Rozprawy
Monografie 306

306 ROZPRAWY
MONOGRAFIE

PAWEŁ RUSSEK

Przetwarzanie dużych zbiorów danych w układach FPGA

WYDAWNICTWA AGH

KRAKÓW 2015

DISSERTATIONS
MONOGRAPHS 306

PAWEŁ RUSSEK

Data-intensive processing on FPGAs

AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY PRESS KRAKOW 2015

Published by AGH University of Science and Technology Press

Reviewers: *prof. dr hab. inż. Jacek Kitowski*
prof. dr hab. inż. Roman Wyrzykowski

Author of the monograph is an employee of
AGH University of Science and Technology
Faculty of Computer Science, Electronics and Telecommunications
Department of Electronics
al. Mickiewicza 30
30-059 Krakow, Poland

Desktop publishing: Author
The printing was carried out from the materials prepared by Author

ISSN 0867-6631
ISBN 978-83-7464-800-4

Publisher's office

Wydawnictwa AGH
al. Mickiewicza 30, 30-059 Krakow
tel. 12 617 32 28, tel./fax 12 636 40 38
e-mail: redakcja@wydawnictwoagh.pl
http://www.wydawnictwa.agh.edu.pl

Contents

PAWEŁ RUSSEK

Data-intensive processing on FPGAs

Summary

In this paper, the author presents his experiences in the area of the use of FPGA devices for data-intensive computing. Processing of the large volumes of data plays a significant role in today's Internet and computing services. Also, big data processing has become a new pillar of science as a tool for knowledge discovery. Traditionally, high-performance computers help scientist to perform simulations and modelling that employ extensive calculations. Input data sets are relatively small, and processing dominates in these applications. Data-intensive problems set new requirements of computers' users, as they require high data throughput rather than high computing power of the computing system. Now, users expect that, besides processing power, their computers provide the highly efficient data movement as well. Notably, these demands are often accompanied by concerns for energy-efficiency.

The solution that is commonly implemented to fulfill all those claims simultaneously is an introduction of heterogeneous computing architectures. Accordingly, commodity computers are enhanced by the task-oriented accelerators such as GPGPUs, DSPs, FPGAs, and other specialised devices. This technique leads to the improvement of computing capability of a computer system. The role of FPGA accelerators is discussed exclusively in this work. An important part of the text covers downsized computing platforms that combine energy-efficient processors with reconfigurable logic and particularly suit for data-intensive calculations.

It is always critical to characterise a class of algorithms that benefits from a use of a particular type of an accelerator device. The author identifies processing tasks that gain when they are ported to an FPGA-accelerated computers. Mainly, sorting and searching algorithms are elaborated in this paper. The author presents hardware structures for well-known, software-based algorithms but also gives new original solutions that particularly satisfy the FPGA-enabled systems.

The practical examples are reported as results of processing performance and energy consumption in the paper. They show that on average over two-fold processing speedup and order of magnitude reduction of electric power is possible in FPGA-enhanced architectures if compared to traditional CPU-only solutions.

PAWEŁ RUSSEK
Przetwarzanie dużych zbiorów danych w układach FPGA

Streszczenie

W monografii autor opisuje swoje doświadczenia dotyczące przetwarzania dużych zbiorów danych przy wykorzystaniu układów FPGA. Analiza danych odgrywa dziś wiodącą rolę w realizacji usług internetowych i przetwarzaniu informacji. Metody określane terminem *Big Data Discovery* uzyskały status jednego z głównych narzędzi nauki służącego do poznania i rozmienia procesów zachodzących w świecie. W przeszłości naukowcy wymagali, aby komputery dostarczały dużej mocy obliczeniowej, która umożliwiała symulację i modelowanie procesów fizycznych. Dzisiaj użytkownicy przetwarzający duże zbiory danych wymagają dodatkowo dużej przepustowości operacji wejścia-wyjścia. Ponadto powyższe oczekiwanie często idą w parze z koniecznością ograniczenia energii zasilania komputerów.

Spełnienie powyższych oczekiwań można osiągnąć, wdrażając architektury heterogeniczne. W tym rozwiązaniu komputery są dodatkowo wyposażane w karty akcelerujące zawierające urządzenia GPGPU, DSP, FPGA czy inne układy specjalizowane. Prowadzi to do poprawy wydajności wspieranych przez akcelerator zadań. Niniejsza praca ogranicza się do zagadnień związanych z układami FPGA. Ważna jej część dotyczy platform obliczeniowych o zredukowanej wydajności, które łącząc energooszczędne procesory i układy rekonfigurowalne, nadają się do zadań związanych z przetwarzaniem dużych zbiorów danych.

W przypadku wykorzystania akceleratorów istotne jest, aby scharakteryzować grupę algorytmów, które mogą zyskać na ich zastosowaniu. Autor przedstawia zadania obliczeniowe, których realizacja w układach FPGA może przynieść korzyści. Treść pracy dotyczy głównie zagadnień sortowania i wyszukiwania. Autor proponuje architektury sprzętowe odpowiadające znanym algorytmom software'owym, oraz wprowadza nowe oryginalne rozwiązania bazujące na zastosowaniu rekonfigurowalnych układów FPGA.

Przedstawione aplikacje dostarczają parametrów wydajnościowych i energetycznych. Pokazują one, że dzięki technologii FPGA, w porównaniu z rozwiązaniami bazującymi wyłącznie na CPU, jest możliwe uzyskanie średnio ponaddwukrotnego przyspieszenie i zredukowanie zużycia energii elektrycznej o rząd wielkości.

List of symbols and abbreviations

ASIC – Application-Specific Integrated Circuit
ASM – Algorithmic State Machine
BRAM – Block RAM
BWT – Burrows-Wheeler Transform
CFG – Control Flow Graph
CDFG – Control and Data Flow Graph
CPU – Central Processing Unit
DDG – Data Dependency Graph
DFA – Deterministic Finite-state Automata
DFG – Data Flow Graph
DLP – Data-Level Parallelism
DP – Double Precision (floating-point arithmetic)
EDA – Electronic Design Automation
FIFO – First In, First Out
FPGA – Field-Programmable Gate Array
FSB – Front Side Bus
FSM – Finite State Machine
FSMD – Finite State Machine with Data
GPGPU – General-Purpose Computation on Graphics Processing Unit
GPP – General Purpose Processor
GPU – Graphics Processing Unit
DSP – Digital Signal Processor/Processing
h – bit-width of the hash function value
HBT – Hash Binary Tree
HDL – Hardware Description Language
HLL – High-Level Language
HLS – High-Level Synthesis
HPC – High-Performance Computing
HT – Hyper Transport (bus)

ILP – Instruction-Level Parallelism
k – number of hash functions in the Bloom filtering
k_{opt} – optimal number of hash functions to minimise a probability of the false-positive in the Bloom filter algorithm
l – index of a level in the tree/trie structure
L – number of levels in the binary tree; maximum length of subsequences in short-read alignment
L_{max} – upper bound value of an HBT height to reduce memory requirements of the Bloom filter
l_{sat} – theoretical value of l at which the trie stops expanding its width in short-read alignment
LUT – Look-Up Table
m – bit size of memory in the Bloom filter
MEM_{size} – memory requirement for a trie level in short-read alignment
MS_l – size of memory for level l in the trie custom processor
n – number of reference patterns in the patterns dictionary
NDFA – Non-Deterministic Finite Automaton
NIC – Network Interconnect Controller
NIDS – Network Intrusion Detection System
NN_l – maximum number of trie nodes at level l in short-read alignment
NS – number of subsequences in short-read alignment
NUMA – Non-Uniform Memory Architecture
p_{err} – probability of the false-positive in the Bloom filter
PCIe – PCI Express (bus)
PL – Programmable Logic (Zynq-7000)
PS – Processing System (Zynq-7000)
RTL – Register Transfer Level
SAB – Sequential-Access Buffers
SoC – System-On-Chip
SP – Single Precision (floating-point arithmetic)
SPMD – Single Program Multiple Data
SSL – Short Subsequences List (short-read alignment)
SPT – Subsequences' Positions Table (short-read alignment)
TL – length of the template genome in short-read alignment
TDP – Thermal Design Power
TLP – Thread-Level Parallelism
UMA – Unified Memory Architecture
WS_l – the size of the memory word at trie level l in the trie custom processor

'Home is where one starts from'

T.S. Eliot

Dedicated to my parents

Introduction

The role of Field-Programmable Gate Arrays (FPGA) in data-intensive processing is being discussed in this work. At present, data-intensive processing has a growing significance in Internet and Computer Technology (ICT) because manipulation and exploration of the massive data sets have become a pillar of modern e-Society. Data-intensive processing plays an important role in e-Business, e-Commerce, e-Learning, e-Government, e-Democracy, e-Health, *etc.* The FPGA devices have been around the electronics industry for a while. Tha applicability of FPGAs in various ICT applications has been deeply exploited since the 90s when the technology was first introduced to the market. In this paper, the FPGA devices are discussed as an implementation platform for the technology of the custom computing processors, or to be more precise, for architectures of the accelerating co-processors. This work is particularly focused on the computing architectures for data-intensive processing.

The custom processors allow a designer of processing systems to meet the most severe project constraints. Constraints are usually delivered in terms of processing speed, energy efficiency, and power consumption. The custom processors offer dedicated architectures that allow the designer to reduce the number of functional resources, optimize data movement and introduce parallel processing also. The custom processing architectures for data-centered computing are surveyed in this text. The special advantage of pipelined architectures is underlined because of their strength to overcome the severe problem of data throughput that limits the processing capability of each computing architecture.

The function of the proposed customized processing architectures is to enhance the operation of the CPU-central systems. A model of FPGA-accelerated computing that will be considered in this paper allows the CPU to offload the most exhaustive computing operations to the specialized co-processor. In that way, the processing is done faster and is more energy-efficient. The sorting, searching and browsing algorithms will be identified as basic core operations for the data-intensive calculations, and consequently, this study will discuss the appropriate hardware structures that are capable of enhancing these tasks.

In Chapter 1, the background topics are presented in a very concise way. First, FPGAs are introduced in a context of other processing technologies. Next, the present challenges and limitations for the CPU-based computing are brought out. We will recall the three walls of CPU technology that limit further general-purpose processor development. Then, the benefits of parallel processing in the computer clusters are characterized. Massive parallel processing is introduced as the current trend in high-performance system development. The architecture of the heterogeneous node for the clusters is given later, and it is explained how it fits the concept of FPGA-accelerated computing. The scheme of the in-node acceleration is underlined. Afterward, data-intensive processing is put in the wider context of today's commonly met computing patterns. Consequently, the FPGA devices are revisited to elaborate how they suit the processing of data-intensive problems. Later, the contemporary claims and approaches for the energy efficient computing platforms in the context of the needs for the data-intensive applications are given. Finally in Chapter 1, the architectures of FPGA accelerator cards that are available today are presented.

The processing models for custom computing processors are elaborated in Chapter 2. The discussion starts with the well-recognized model of the sequential custom processor. Its architecture is discussed at the registers transfer level. An example of a simple custom processor for the SQL operations is delivered to illustrate the concept better. The construction and operation of the processor's controller and data path are explained. Later, the problem of scheduling of the algorithm operations is highlighted. Scheduling is an important topic because it allows a processor designer to introduce concurrent processing. A tool of the Data Dependency Graph is briefly referred to for that purpose. A technique of pipelining, which is crucial for discussion in this work, is regarded afterward. Particularly a procedure of loop pipelining is examined in that context. Next, the problem of loop dependencies and control statement pipelining is viewed specifically. To summarize the subject, pipelining of the SQL processor is presented as an example. Afterward, the problem of the use of the FPGA's memory blocks is taken up in the chapter. First, the methods of explicit data caching are presented, and a technique of the Sequential-Access Buffering is proposed. At the end of Chapter 2, the practical characteristics of algorithms that match FPGA-based processing well are summarized.

In Chapter 3, the algorithms and architectures for sorting and searching in FPGAs are discussed. The chapter starts with a method of the sorting nets. Then, the merge sort tree structure is shown in relation to custom architectures. A practical sorting system that use the merge sort tree implemented in FPGA is characterised. The searching algorithms are regarded next. First, the sequential and parallel version of the Bloom filter is considered. Afterward, the problem of 'false-positives' in the Bloom filter architecture is recalled, and methods for its mitigation proposed. In that

context, Cuckoo hashing, and its modified version are discussed later. The problem of a shifting substring search scenario that is common in real-time search systems is highlighted afterward. Next, the algorithm and the architecture of a custom processor that implements the binary tree is given in the chapter. Later, as an alternative to the binary tree, the prefix tree is proposed. The pipelined architecture of the prefix tree processor is presented. Chapter 3 is ended by the section that regards the well-known Aho-Corasik algorithm. The problem of the shifting window search is revisited, and examples of the respective custom processor architectures of the Aho-Corasick algorithm are referred to.

Both Chapter 4 and Chapter 5 are a part of the paper that presents author's practical solutions for data-intensive processing in FPGAs. In Chapter 4, the problem of mapping of the text terms to term IDs is regarded. The algorithm of the Hash Binary Tree (HBT) is proposed. The HBT and the modified Cuckoo hashing are discussed together. Also, the memory requirement of the Hash Binary Tree scheme is compared to the corresponding Bloom Filter memory size. The advantages of the proposed approach are highlighted in the chapter, and the results of its implementation in a reconfigurable System-On-a-Chip are given.

Chapter 5 discusses the application of DNA short-read alignment, where DNA short reads are matched with a sequence of the reference genome. The modified architecture of the prefix tree is proposed to enhance the search for inexact matches. Both sequential and pipelined architectures are discussed. The appropriate data representation for the set of DNA short reads is proposed, and the modified architecture of the prefix tree processor is implemented in FPGA. Also, the implementation results are given. Finally, as the size of the prefix tree that can be accommodated by the capacity of the available reconfigurable logic is too small, the method of CPU-FPGA co-processing is proposed to handle the real-life DNA matching workloads.

Acknowledgements

The author wants to thank all the people of the Reconfigurable Computing Group at AGH University of Science and Technology with whom he has been working. There would be no personal achievements without the whole team's long collaboration. Special thanks go to Prof. Kazimierz Wiatr whose leadership allows the group and author to advance. Notably, this paper would never have been written without the Professor's support and encouragement. A lot of people have cooperated with me, but some have to be mentioned by their names: Dr. Ernest Jamro, Dr. Agnieszka Dąbrowska-Boruch, Dr. Maciej Wielgosz, and Dr. Marcin Pietroń. Thank You!

1. Computing and FPGAs

1.1. Basics of FPGA devices

Field Programmable Gate Arrays (FPGAs) are a kind of semiconductor devices that are arrays of unwired digital electronic elements. Early FPGAs provided simple elements that were combinatorial logic and flip-flop components only. Today, as FPGA have evolved, they offer an abundance of various electronic blocks. Nonetheless, the most distinctive feature of the FPGA devices are the configurable routing resources that allow an FPGA designer to configure the hardware array into a custom hardware architecture. Just like a processor that requires a software binaries, an FPGA needs a configuration bitstream to gain useful functionality. Consequently, the programming of FPGAs means planning its interconnection routing and designing the internal configurations of its digital components. The configuration and routing are downloaded to the FPGA's configuration memory as a configuration bitstream. Once configured, an FPGA can act as any digital electronic circuit. Their only limits are the number of available logic resources and the clock frequency. The FPGA bitstream and processor software binaries bring those devices to run, but they are very different in their service.

In digital electronics, devices that are not software running processors or memory chips are called Application-Specific Integrated Circuits (ASICs). In difference to a processor's Central Processing Unit (CPU), ASICs perform operations that are wired in their architecture. ASICs are just like FPGAs with that feature. However, unlike FPGAs, ASICs have a locked functionality. Thanks to the fixed service and hooked structure, ASICs are typically more efficient in the sense of speed and energy consumption if compared to processors. For example, ASICs optimize data movement and data representation when performing built-in operations. The FPGA solutions derive from the ASIC's processing methods. The most significant difference is that the FPGA is configured by an end user, and the ASIC's architecture is settled by its manufacturer. On the commercial market, the most important family of the FPGAs is a family that is manufactured in the well-developed SRAM semiconduc-

tor technology. A distinctive feature that comes with the SRAM-based technology is that, like SRAM memories, FPGAs can be reconfigured many times. These SRAM devices require a download of a configuration bitstream every power-up cycle; therefore, they can change their functionality relatively quickly. One can see this feature as a downside, but it can be exploited in a positive way also. For instance, the same logic resources can provide the different functions during the separate phases of the device's run-time. It is feasible that several algorithms or its parts share the same logic in a time domain. The name of this ability is run-time reconfiguration. Another interesting option is partial reconfiguration that allows it to modify only a selected part of the FPGA while conserving the remaining part of the device. To highlight these essential features, we also call FPGAs the reconfigurable logic devices.

The ability to reconfigure is the fundamental advantage of the FPGA technology over the ASIC technology. It shortens development time of new projects, reduces Non-Recurring Engineering (NRE) costs, and improves the project's maintainability and its flexibility for updates. Unfortunately, reconfiguration feature does come at a price. The estimation leads to the conclusion that routing resources occupy 80% of a silicon area of the FPGA chip [1]. Additionally, the switches that provide reconfiguration ability introduce an extra delay to signal propagation. Consequently, the FPGA's clock frequency is ten times lower than that of the corresponding ASIC. Thus, as the FPGAs offer five times less the amount of usable logic and work ten times slower than ASICs, they perform 50 times fewer operations at the same time. To mitigate this problem, the FPGAs that are provided to their users are manufactured in state-of-the-art semiconductor technology. It has economic foundations because, unlike the ASICs, the FPGAs are off-the-shelf devices. Only high volume manufacturing can make up for the NRE costs that the introduction of each new technology brings on. Today, FPGAs are commodity devices, as their sales volume is estimated to be 250 million in 2015. This quantity can be compared to 230 million of discrete Graphics Processing Units (GPUs) that are expected to be sold at the same time. The main highlight of the FPGAs' properties, when compared to the ASICs, is the fact that they often serve as an initial prototyping platform for the latter.

The SRAM-based FPGAs take advantage of static memory semiconductor technology; therefore, they implement combinatorial logic as memory elements that are called Look-up Tables (LUTs). One LUT and one flip-flop constitute the basic FPGA cell that is referred to as Logic Cell (LC) or Logic Element (LE) (the actual name is vendor specific). These cells are sufficient to build any sequential synchronous digital device. However, resource and processing overhead that is introduced by routing resources influences the evolution of the FPGA internal architecture also. Thus, FPGA architectures offer more complex elements than logic cells today. Manufacturers have been including components that offer more specific and advanced function-

ality as FPGA architecture evolves. These are fast carry logic for arithmetic, memory blocks, multipliers, serial communication transceivers, and CPUs for example. The goal is to deliver to a user the most typical and commonly used components because specialized hard-wired blocks perform faster and require fewer transistors.

Broadcast, networking, wireless, and telecom equipment has been relying on FPGA technology for years. As a result, those markets have been shaping FPGA architecture. At present, FPGAs contain such embedded components as synchronous dual-port RAMs (Block RAMs), Digital Signal Processing blocks (multiply and accumulate), multi-core CPUs, Gigabit transceivers, Ethernet MACs, and PCI Express interfaces. Today's FPGAs are platforms that target several application domains. They exist in variants. For example, FPGAs that offer a lot of DSP blocks, and transceiver blocks are optimized for communication-based, DSP-centric applications found in wire line, military, broadcast, and High-Performance Computing markets. Similarly, devices that provide the highest logic density are designed for ASIC and system emulation, diagnostic imaging, and instrumentation. The FPGAs that contain CPUs and memory blocks are suited for the customer, automotive, and industrial markets. There is also a variety of FPGA sizes for each type available.

References

Older books offer a valuable insight into the origins of FPGA technology; for example [2], [3], and [4]. John Villasenor proposed the very first idea of so-called reconfigurable computing in 1997 [5]. Since then, as the technology of FPGA reconfiguration evolves, designers have been constantly developing the idea of FPGA resources reuse by the non-overlapping system functions.

A huge bibliography in the FPGA subject exists. Here, we will refer to only a few works that are related to our topic of data-intensive processing. For example, Horta *et al.* [6] implemented high-speed Internet packet processing circuits that were Dynamic Hardware Plugin (DHP) modules. Oliver *et al.* [7] use customization opportunities available at run-time to dynamically scan bio-sequence database. Dynamically reconfigurable Content Addressable Memory (CAM) for classifying IP packets into categories is proposed by Ditmar *et al.* [8]. Divyasree *et al.* [9] present dynamically reconfigurable generic block that implements the regular expression matching. Ruta *et al.* [10] proposed integrated development platform that features a programmable FPGA board, where computations of different nature and purpose are logically distributed among a sequential soft-core processor program, a massively parallel accelerator, and an independent communication module.

The methodology of FPGA-based prototyping and a comparison of FPGAs and ASICs is given by Amos *et al.* [11]. Kuon *et al.* [12] deliver experimental measurements of the differences between a 90-nm FPGA and CMOS ASICs in terms of

logic density, circuit speed, and power consumption. Behrooz Zahiri [13] assesses structured ASICs that are the bridge between the two competing semiconductor technologies. A good review of FPGA technology, tools, designs, and applications is also given by Rodriguez-Andina *et al.* [14].

1.2. The challenges of efficient data processing

Descending from von Neuman's architecture, computer naturally performs tasks in a sequential way. The system runs one program after another. The processor executes instructions in a one-by-one manner. And finally, the CPU processes instructions in the sequence: 'fetch', 'analyse', 'load data', 'execute' and 'store results'. For faster processing, today's computers feature the parallelism at the different levels of code execution. Respectively, Thread-Level Parallelism (TLP), Instruction-Level Parallelism (ILP), Data-Level Parallelism (DLP), and Instruction Pipelining are possible.

Thanks to TLP, a user can run many processes (or threads) at the same time. Multitasking is a feature of the Operating System (OS), and it enables TLP to distribute tasks to many processors and thus speed-up the overall execution time. It is an important feature that threads can communicate and synchronize their work during a run-time. Desktop PCs usually run different codes for each process because processes are separate programs, but computers can also execute single code in independent thread instances. Sharing a program by threads is referred to as Single Program Multiple Data (SPMD) model [15].

A processor's CPU executes more than one instruction at a time in ILP while DLP allows it to perform the same instruction for different data simultaneously. Additionally, the CPU executes each instruction in the pipeline to increase program throughput. However, the correct scheduling of instruction has to be done to enable ILP, and adequate data partitioning is necessary for DLP respectively. In practice, those tasks are performed either by a software engineer, who schedules instructions and organizes data during program development, or automatically by a CPU at the time of program execution. The first approach requires enhancement of software binaries for new processor architectures while the second one allows the CPU to execute sequential legacy programs more efficiently without additional programming effort. This automation is possible thanks to architecture advancements of the new processors. Efficient ILP that is driven by a hardware requires advanced mechanisms. Techniques that are in common use include multiple instructions issue, deep pipelines, out- of-order execution, speculative execution, and data pre-fetching. Needless to say that sophisticated ILP automation requires additional hardware resources, but advances in semiconductor technology get over this barrier. Unfortu-

nately, the current commodity processors have reached the point where further advancement in processor's architecture produces no significant performance improvement of ILP. This obstacle is called the ILP wall.

As the enhancements of the CPU's architecture had become barren, multi-core processors came to the market. Putting more CPUs into a single processor did become a method to take advantage of excessive semiconductor resources. Consequently, manufacturers made a decisive shift towards TLP while still maintaining ILP to work for a favor of each CPU. Multi-tasking OS could instantly exploit TLP for the benefit of a computers' users who run many applications on their laptops or desktops simultaneously. The same time, the users of High-Performance Computing (HPC) systems took advantage as well because multi-core processors provided more CPUs to a single socket of a computer motherboard.

Present processors integrate a majority of computer subsystems on a single chip, but the main system memory remains off-chip. There is a technological and economical reason for this separation. A cheap high-capacity semiconductor memory is fabricated in the dynamic RAM technology, but processor chips need the fast CMOS technology. The two technologies are different, and they cannot be integrated into a single chip. That is the reason that processor's chips contain only static RAM cache memory. The size of the cache memory is three orders of magnitude smaller than the capacity of the main memory because static RAM technology is expensive. The processor needs an on-chip memory interface to access the external memory, and the cores share this interface in a multi-core architecture. Apparently, this sharing leads to a communication bottleneck and CPU waiting states. This phenomenon constitutes the so-called memory wall problem. The processor can not use its full processing power because the data delivery from the memory is too slow. The memory wall problem exists for both the one-core and multi-core processors. Due to high latency and limited throughput of read/write operations even an exclusive CPU access to the memory is not sufficient to avoid data starvation of the processor. Modern processors use data caching to lessen the problem, but caching techniques assume data reuse *i.e.* data that has been read, will be read again soon. Data reuse is not always the case, and we will discuss this serious topic later.

Speculative execution techniques can provide a significant performance advantage but at the cost of increased power consumption. Transistor down-sizing continually allows to enlarge resources of the single chip. That scaling process leads to an increase in the clock frequency and, inevitably, to the higher power density of the chip also. However, dissipated power cannot rise beyond the capabilities of the available cooling techniques. Processors' designers had been compensated the increasing transistor density and clock speed by lowering the supply voltage. Unfortunately, the decreasing of the supply voltage is no longer easily possible due to the physical properties of silicon. This limitation is known as the power wall.

Despite that technology improvements mitigate the three walls of today's computing, the conventional wisdom tells how to avoid computational limits. First, lowering the clock frequency prevents processors from hitting the power wall and improves the balance of the CPU and memory performance. The better performance balance of system components mitigates the memory wall problem also. Next, processors of simpler architectures outperform more complicated ones when it comes to electric power consumption. That is the reason the simple RISC processors work more efficiently than highly sophisticated superscalar CISCs. Similarly, many-core accelerators, like the GPGPUs, outperform the commodity processors in terms of computing power per watt. The GPGPUs feature many simple processing cores, and this gives them an advantage over the commodity processors. Designers of HPC clusters, who follow the rules of CPU simplification and clock frequency reduction, achieve the best energy efficiency of their systems [16].

References

The full coverage of the topic discussed in this chapter can be found in the classic book of Hennessay and Patterson [17]. One can find a practical overview of TLP, ILP, and DLP in works of Akhter and Jason [18] and Herlihy and Shavit [19]. The group from Berkeley had formulated a serial processing performance problem: 'the power wall + the memory wall + the ILP wall = a brick wall' [20]. The evolutionary change in conventional computing that is introduced by multi-core processors is presented by Gepner and Kowalik [21]. Mahapatra *et al.* [22] examine the problem of the memory bottleneck. The issue of CMOS technology for microprocessors scaling that leads to performance improvement, transistor density increase, and power reduction is well approached by Borkar [23].

1.3. Massively parallel computing

Amdahl's argument that parallel processing has limited use for the practical computations is known as Amdahl's Law [24]. Any computing system that is designed for parallel computing is useless in a consequence of that law. Inappropriately, HPC parallel systems thrive in science and technology because they are necessary to solve many real-life modelling, simulations, and analysis. *In silico* research and development would not exist without high-performance parallel supercomputers. Supercomputers successfully use many processors to solve significant computational problems because Amdahl's reasoning neglected the practical experience of HPC computations, which states that parallel systems are essential to solving exceptionally big problems. The term 'big' corresponds to the fact that they are characterized by large sets of input parameters which model real-life objects.

Computer scientists use a simple model of the computing algorithm to demonstrate the imperfection of Amdahl's law. This model is provided in Listing 1.1.

Listing 1.1. A model of inner and outer loop

```
for (element in inputSet) do
   initOps(element)  /* Do some entry outer loop operations */
   for (part in element) do
      innerLoopBody(part) /* Do heavy inner loop operations */
   end for
   cleanUpOps(element)     /* Do other outer loop operations */
end for
```

According to that model, the real algorithms exhibited an inner and outer loop. One can notice that the outer loop is easy to parallelize on many working processors. Amdahl's law considered the existence of the inner loop only and forgot that as the input set grows the number of computations that could be done concurrently also grows. Obviously, the greater size of the input set, the better quality and accuracy of obtained results in practice. For example, a finer mesh for modelling of real objects is possible, or a more complex system can be simulated as a single entity. Consequently, Gustafson's law was formulated to supplement Amdahl's law [25]. It takes a more practical experience and states that as the input set grows, it is possible to distribute and process elements in parallel to reduce computation time.

Today, most supercomputers are computer clusters [26]. A cluster is an assembly of connected computing nodes that compute in the group. Cluster's nodes usually have an architecture of the commodity computer servers, and they are built using commodity computer parts. While calculating, they exchange data and communicate to synchronize their work. Fast and efficient communication is possible thanks to the dedicated interconnect network. The aggregate power of all processors and the combined memory capacity of all the nodes allows the clusters to solve severe computing problems. The computer clusters enable the performance of Massive Parallel Computing.

One can find more information regarding HPC in Morse's [27] and Loshin's [28] books. The modern heterogeneous computational infrastructure that is used by today's scientific community is presented by Kitowski *et al.* [29].

1.4. Heterogeneous computing platforms

Symmetry and homogeneity of is an advantage of a big computing system architecture. It is also convenient that nodes of the computer cluster are identical. A homogeneous cluster structure simplifies system programming and management.

Therefore, all cluster nodes use the same hardware in most circumstances. Thanks to the uniform structure of the cluster, it is possible to develop and use a concise programming model. Though, today's clusters are uniform, as one node is similar to another, they are unfortunately heterogeneous when one analyzes the node's internal architecture.

First of all, there is no single memory model in the cluster. The CPU cores work using the Unified-Memory Architecture (UMA) at the processor level. However, the processors of a multi-socket system use the Non-Unified-Memory Architecture (NUMA) at the server motherboard level. Despite that, the system memory model is distributed at the cluster level. The rationale for such a troublesome structure is practical from a hardware engineer's point of view. The supercomputer industry shares as many components as possible with the commodity servers' business. Cluster nodes that are built using commodity server's chipset inherit a shared memory architecture. Servers that feature more than one processor socket fall into the NUMA model as the present multi-core processors integrate a memory controller on the chip. The memory model is very different at the cluster node level, where the memory model is a distributed because the clusters use a computer network to communicate.

Accelerators are another element that breaks the homogeneity of a cluster node structure and in result a uniformity of its programming model. The commodity market drives a use of accelerators in supercomputing clusters. The urge for computing power in HPC is high, and the supercomputer industry tries to absorb every solution that allows it to raise computing capabilities at a competitive price. Lower costs usually come with the huge production volume and only the mass customer market accessories meet this condition. Thus, accelerators are usually the devices that have been adopted for supercomputing from other ICT segments. Just like GPGPUs that are derivatives from the market of computer games for example.

The accelerators are incorporated into the computing nodes to enhance the computing capabilities, but they usually reinforce processing for a selected class of algorithms only *i.e.* some types of calculations gain a speedup or energy efficiency. The architecture of an accelerator is customized to suit particular computing problems. Thanks to this, they can perform calculations faster and are more energy efficient than General Purpose Processors (GPP). Accelerators benefit from a precise adjustment of their hardware architecture to such algorithm's elements as data representation, the type of performed operations, and the data flow. Specialized functionality and tuning for needs of individual algorithms often decreases the logic resources that are necessary to implement the processing element. Consequently, the size reduction of processing elements allows it to replicate them within the same area of the chip silicon and in the same transistor budget. The multiplication of computing cores naturally leads to Thread-Level Parallelism and provides the gain in processors performance.

Data movement is a source of significant energy consumption in a processor architecture. The optimization of the data flow and communication paths provides lower energy consumption, as register-to-register data movement is naturally reduced.

The idea to use the accelerator to speed up the execution of the inner program loop seems to be natural. In practice, acceleration yields when it concerns the code that is repeatedly executed. The practical observation, known as the 90/10 law, states that the processor spends most of the program execution time (90%) in a small portion of the code (10%). The law is just a trend that has been derived by computer scientists from the Pareto 80/20 rule [30], but it implies that 10% of the code constitutes the so-called computational kernel. These kernels are the primary targets for various efforts of the software optimization. The migration of the kernel's code to the accelerator is an optimization that requires a substantial engineering effort because accelerator programming is always harder than CPU programming. It is tedious and time consuming because it often uses a low-level, hardware-aware and parallel programming model. Apparently, such extensive programming labor should be reserved for a part of the program code, which is as small as possible and simultaneously produces substantial program execution time. This observation leads to the concept that acceleration should be exclusively applied to speed up the program code of the inner loop body or its selected part.

Programming of a computing node that is composed of different computing parts is a challenging task. The reason is that accelerators typically accompany the main processor as IO extension cards. Accelerators are semi-detached, and they belong to a different program execution level than the host's CPU. They usually require separate programming *i.e.* a software engineer must distinguish a program that is intended for the accelerator from a code that belongs to the CPU. Additionally, accelerators feature a private memory that is used to store data during processing. That data has to be explicitly transferred from/to the host computer.

1.4.1. Graphics cards

The computer accelerators can be either off-the-shelf or fully customized devices. General-Purpose Computation on Graphics Processing Unit (GPGPU) cards are an example of the off-the-shelf accelerators. The first graphics processors were non-programmable ASICs that were capable of performing graphics operations only. Later, as graphics cards evolved, GPGPU's manufacturers allowed for more flexibility and introduced some programming capability. Today, GPGPUs are fully programmable, many-core software processors that fall into the SPMD programming model. Graphics processors drew widespread interest throughout the computing community when they became fully programmable and began to support double-precision floating point calculations.

One can find conventional features of an accelerator in GPGPU architecture. These characteristics are:

- it provides a significant number of simplified computing elements,
- it exists as an IO extension of the host computer,
- it uses local memory that is separated from the host's main memory.

The architecture of the GPGPUs suits best the image processing algorithms, but it is also appropriate for other computing tasks that exhibit locality of data access. They provide the programming model that allows it to distribute input data to the array of massively parallel processors. As GPGPUs are derivatives of graphics cards, they contribute mostly to single-precision (SP) floating point calculations. Narrowed precision, which is acceptable in graphics processing, allows GPGPU's designers to increase the number of processing elements and computing power. Present GPGPUs have matured to double-precision (DP) operations but the SP-based computations are still the most profitable. Another example of off-the-shelf accelerators are the Digital Signal Processing (DSP) processors but they are not as popular for general purpose computations as the GPGPU cards.

1.4.2. FPGA accelerators

As it was stated, construction of computer clusters relies on commodity computer elements mainly. However, some customization of cluster's components exists and applies primarily to the Network Interconnect Controller (NIC) chipset. Efficient interconnect is crucial in supercomputing architectures; therefore, clusters often use a proprietary network interface that is implemented as an ASIC device. The cost of a fully-customized ASIC is prohibitive, and it is appropriate for high quantity products only. That holds for NICs, but not for custom processors that are designed for a distinct user's algorithm exclusively. Additionally, permanent wiring of the custom processor for a selected algorithms is problematic because real programs often need regular improvements and alternations.

The above remark might suggest that custom-built accelerators are not used for general purpose computing. However, thanks to FPGA devices, that is not the truth. The IO extension cards that integrate FPGAs allow the computers' users to run personalized accelerators. The custom computing processors are delivered to the FPGA chip as an Intellectual Property Cores (IPCores), and they are run as custom FPGA configurations. The IPCores are ready to use, hardware designs that can be seen as reusable components from the library. IPCores take their name after proprietary rights they carry, as they are usually purchased from the hardware vendor. In theory, the IPCores can be technology independent and might be adapted to any FPGA or ASIC technology, but such consolidation requires professional knowledge. Modifications

of soft IPCores are relatively easy, and so the evolution of the custom algorithm is also possible. One can download the IPCore's configuration directly to the device in the case of FPGA technology. The ability to handle the FPGA configurations in a way the software programs are managed reduce the costs of custom computing acceleration substantially.

Heterogeneous computers that are enhanced by accelerator cards contribute to the HPC landscape today. In-node accelerators deliver cheaper computing power to the supercomputing centers, and this results in higher processing power that is available for the HPC users. The obvious advantage is the shorter execution time of applications; thus, the users can perform more simulations at the same time. Another point of interest at the computing centers is the power saving that is delivered by accelerators. Thanks to the reduction of resources and optimization of data movement, custom processing elements tend to be more energy efficient than CPUs. Even if the accelerators' and CPUs' performance is similar, it is better to use a customized device as it is usually less power hungry. Especially, FPGA cards are energy efficient. The energy consumption of FPGAs is an order of magnitude lower than the energy consumption of server processors and substantially lower than the power dissipation of mobile, low-power processors.

References

The research and practical use of GPGPUs and FPGAs for computing applications have returned a flood of publications. Here, we will review only a few papers that have been selected because of the author's contribution.

Kuna *et al.* [31] describe and briefly compare dedicated hardware accelerators, such as GPGPUs, IBM Cell processor, and ClearSpeed processor, to contemporary GPP architectures. It concludes that CPUs are not sufficient for large matrix or vector computations, where they are outperformed by massively fine-grained data-parallel devices. Paćko *et al.* demonstrate how graphical processing units can be used efficiently for large models of elastic wave propagation in complex media. The obtained results indicate significant speedup factors compared to calculations using central processing units or different modelling approaches.

Pietroń *et al.* [32] parallelized SQL operations. The results showed that SELECT WHERE and SELECT JOIN operations on the GPGPU were faster than the sequential ones that were run on the CPU. The primary intention of Dąbrowska *et al.* [33] was to present the results of several cases, where the FPGA technology had been used in high-performance applications. The article gave selected metrics, results and conclusions that were derived from the implementation of several functions. In conclusion, the authors stated that adequate computing technology had to be selected according to the characteristics of the computing problem.

Cryptography, data mining and life science applications were recognized as areas for the most successful use of FPGA accelerators. Jamro *et al.* [34] present implementation results of algorithms that are computational kernels in cryptography and data analysis. Gielata *et al.* [35] investigate hardware implementation of AES-128 cipher standard on FPGA technology. They proposed the pipeline architecture of AES modules. The paper reports throughput of 21.2 Gbit/s and 16.6 Gbit/s for coder and decoder respectively. It was compared to 77 Mbit/s and 74 Mbit/s accordingly on Pentium II 450 MHz.

Russek and Wiatr [36] focus on the analysis of technical challenges of reconfigurable computing in multi-user, multi-threaded systems. More specifically, Russek and Wiatr [37] present their approach to the custom matrix multiplication implementation. The presented architecture is dedicated to the SGI Altix 4700 supercomputer system. The paper claims that the 200 MHz clock system gained computing power of 9.6 GFLOPS.

FPGA implementation of the double precision exponential function module is presented by Wielgosz *et al.* [38]. The function is accelerated on an SGI RASC board with two Virtex-4 LX200 FPGAs. The authors predict the final algorithm execution speedup to be 4× as compared to a sequential implementation on a 2 GHz Intel Itanium2 microprocessor. Wielgosz *et al.* [39] present FPGA acceleration and implementation results of a hardware module for generating an orbital function that is used in quantum chemistry calculations. The computational procedure presented in the paper is part of an algorithm for generating exchange-correlation potential, and it is also recognized as one of the most computationally intensive routines in quantum chemistry calculations. The paper of Wielgosz *et al.* [40] presents an FPGA implementation of a calculation module for the exponential part of Gaussian Type Orbital (GTO). The hardware implementation revealed significant speed-up for the calculation of the finite sum of the exponential products, ranging from 2.5× to 20× in comparison to a general-purpose CPU version.

1.5. Data-intensive computing

The term data-intensive computing is used to describe applications that are IO bound. They can be identified by evaluating the number of bytes of data processed per one floating-point operation. Today, data-intensive applications require the manipulation of huge volumes of data. Therefore, data-intensive computing denotes also the use of a computer to process a large amount of input data. The large volume of data means terabytes (TB) or even petabytes (PB) of data in the era of the Internet that is rich with information. However, data-intensive computing is not an equivalent of Big Data computing. Big Data directly refers to large data sets with sizes beyond

the ability of commonly used software tools to load, store, manage, and process data within tolerable elapsed time. Methods and algorithms, which are being developed, to handle Big Data are often data-intensive to minimise processing time. However, we want to put the term data-intensive in a data-size-independent context here in the paper. We will judge the domination of data with the relation to a quantity of operation within a framework of an analyzed algorithm.

Let's explore terms of data-intensive and compute-intensive calculations in this section. It is necessary to confront the size of the input dataset with the number of performed processor operations to do that. Consequently, we will be able to distinguish data-intensive and compute-intensive algorithms despite the actual size of the input dataset.

1.5.1. The big O notation

The assessment of the calculation effort with respect to the size of the input data is provided by a theory of computational complexity. It assigns an appropriate difficulty to the analysed algorithm by the use of the big O notation. This notation defines a complexity to the algorithm by a measure how it responds to the change in the size of the input. In other words, the big O notation describes a growth rate of a function that characterize data size and a calculation complexity relationship. More precisely, big O provides an expansion rate of a function that express a number of operations for a given input size. The complexity of a scalar vector product and matrix multiplication, where N is the vector/matrix dimension, can be given as $O(N)$ and $O(N^3)$ respectively. The complexity measure allows it to compare different algorithms computational difficulty. The more complex the algorithm is, the more computationally intensive it is.

Despite the observation that data-intensive processing invokes low-difficulty algorithms, it is vital for our discussion that the volume of data still matters in data-intensive computing in practice. Computers could not tackle big-size problems using high-complexity algorithms in a reasonable short time. Therefore, computationally cheap algorithms are used in practical applications if processing of big input volumes is necessary. The algorithm that is successful for processing of input data of modest size has to be simplified if the size of entry grows. It is even a case when it produces the lower accuracy of the results. Consequently, computer science recognizes data-intensive problems as tasks that create modest computing intensity for a big amount of input data.

Different to compute-bound operations, data-intensive operations are IO bound. Whereas compute-bound operations can be done faster if a CPU runs faster, the performance of IO-bound program relays on IO speed. A term of IO can be extended to the main memory in many cases because, as it has been already mentioned, modern

CPUs suffer the memory wall. IO throughput is not a limitation for the compute-bound algorithms, as they can exploit the CPU's performance thanks to the cache memory. Once a processor reads data, it is stored in the cache that performs an order of magnitude faster than the main memory.

1.5.2. Computational patterns

Patterson's group from Berkeley University identified application areas for computing platforms [41]. They pointed out areas like embedded systems, general-purpose computing, databases and browsing, games, artificial intelligence and machine learning, computer-aided design, and high-performance computing. Also, the group determined twelve computational patterns for these applications. One can find dense linear algebra, sparse linear algebra, spectral methods (e.g. FFT), N-Body methods, combinatorial logic, and Finite-State Machines among them. A use of FPGAs for data-intensive processing have a significant support of this work because the group qualified combinatorial logic and FSM as the patterns suitable for the application area of Database and Browsing (D&B). The relevance of D&B applications to conventional digital circuits operations induces a value of FPGAs for data-intensive computing!

The most fundamental ability of big data systems is storing and searching. Apparently, these functionalities cannot satisfy all existing needs of data processing but they are basic. Today, internet browsing and searching is most prevalent and dominant operation. Web browsers retrieve search results from servers of service providers, where the information is stored. Extremely big data repositories must be kept in data warehouses.

The primary purposes are data-mining and information retrieval applications when someone considers data-intensive services. Those applications employ sophisticated, computationally intensive algorithms of artificial intelligence, machine learning, and statistics. However, a simple database search goes ahead of any data-mining analysis to limit the volume of input data that has to be handle.

A clever organization and structure of data significantly helps in efficient processing. The structure of a binary tree reduces the computational complexity of a binary search algorithm to $O(\log N)$ for example. Just like R. Pike [42] stated: "Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming". Therefore, the sorting operation is essential to put data in well-organized structures. Consequently, the sorting is also an important process in databases. The average difficulty of the fastest sort algorithms such as heapsort, quicksort, and mergesort is $O(\log N)$ only. It is even smaller in some particular cases when one can implement bucket-sort or radix sort.

Systems usually inspect information that is stored locally but sometimes it is on-line data, delivered in real-time, that is analysed. Such data is raw, unstructured, and must be analysed *ad-hoc*. The searching of unstructured data exhibit a linear complexity $O(N)$.

1.5.3. Distributed databases

The growing importance of browsing weakened the role of relational databases in today's computing environment. Relational databases have been used for storing data for years. Such databases give an advantage of saving of storage space because they store each data element only once. It is possible also to create any complex table dynamically when it is necessary. Nonetheless, distributed databases gain popularity in the era of data warehouses and petabytes storages. In such a databases, data is not centralized but distributed to many system nodes. Thus, a computer cluster is a natural platform to store a distributed database where each node is capable of storing a part of information. This approach allows it to perform database operations in parallel.

The popularity of distributed databases comes together with the importance of the MapReduce model. MapReduce is a model used for parallel data processing. A key primitive to explore such collections is MapReduce. The essence of MapReduce is a couple of functions: a map function and reduce function. A single map function executes in parallel on independent data sets. The data sets are usually physically distributed in the system. The output from the map functions is combined and eventually reduced to form a single or the small number of results.

References

A good example of the applications of data processing, which employ suboptimal algorithms to replace their ideal counterparts, is real-time video compression. A video camera delivers a substantial data stream that must be compressed to be further handled. Motion estimation and coefficients quantization tasks are computationally expensive in video compression. The research that had been carried out by the author and author's colleagues provides a good outline of the problem. The method of video compression that is presented by Wiatr and Russek [43] is the modified Embedded Zero Wavelet (EZW) algorithm. It was enhanced, by the authors, to simplify quantization. Next, Dąbrowska and Wiatr [44] discussed the motion estimation algorithm. Precisely, the work presents the modification of the Efficient Three-Step Search motion estimation algorithm to suit an FPGA implementation.

The paper of the Berkeley group [41], the one that had been already cited in the text, has been followed another important publication of that group which was a report [20]. The goal of the work was to challenge the conventional wisdom that had preserved the programming paradigms of the past, and re-invent cornerstones of

computing to simplify the efficient programming of parallel systems. Among others, the report contains statements that are vital for a discussion in this paper:

- The essential function of modern databases is hashing.
- The sort is at the heart of modern databases.
- It is important to have efficient interfaces between IO and main memory to sort large files fast.
- The future of databases was large data collections typically found on the Internet. A key primitive to explore such collections is MapReduce. The essence is a single function that executes in parallel on independent data sets, with outputs that are eventually combined to form a single or a small number of results.
- Among 'dwarfs' of computing, the eighth dwarf is combinatorial logic, and the thirteenth dwarf is Finite-State Machine processing.

The book of Han *et al.* [45] provides a survey of data mining. It gives introductions to concepts of database and data mining, with emphasis on data analysis. Additionally, it covers the concepts and techniques of classification, prediction, association, and clustering.

J.Gray *et al.* [46] prepared the report, which showed that sort operation is IO-bound. The authors presented an underutilization of the CPU for the tested sort algorithms. Also, authors keep results of sorting benchmarks on the web page [47]. The list shows the influence of the performance of IO subsystem on the sorting speed that is obtained by the tested computer systems.

1.6. FPGAs in data-intensive applications

Performance is an important parameter of a computing system, but owners of the big computing facilities care for energy consumption also. More precisely, they are interested in the best performance per watt ratio. Energy-efficiency is an important system quality measure for big computer clusters. Thus, FPGA devices come in useful as they outperform other computing devices in performance and energy savings in selected tasks.

One can distinguish perfect algorithm's features that lead to successful FPGA implementation. Non-standard data representation, simple logic like operations, and fine-grain parallelism are the most general characteristics of them. The above attributes might serve as a guideline of a system designer when he considers FPGAs to reinforce CPUs. They mark the perfect candidate that probably does not exist in practice. However, algorithms exist that partly meet the above criteria. It is important for the discussion in this paper that some of them can assist browsing and searching in data-intensive applications.

CPUs use integers and IEEE-754 floating-point data representation for arithmetic. Correspondingly, processor registers have a definite binary size. Meanwhile, database's data representation is far more rich. Databases use character string, date, time, and binary formats for instance. Also, the size of database fields is arbitrary. It does not come naturally to manipulate data elements of an unconstrained type and size in CPU. Consequently, the end performance is affected. It is wasteful to use 32 or 64-bit registers to deal with eight-bit characters for example. Although, data can be re-expressed to fit the internal data representation of CPU better, that solution is not always the best approach because it requires constant data conversion between human-friendly and internal computer format. Also, it may be necessary to keep native data representation to allow for some types of data manipulation; like comparison for example. Finally, it is not possible to use an auxiliary representation for an *ad hoc* data processing.

Contrary to CPUs, FPGAs can be configured to conform to random data formats and data size. FPGAs naturally handle and process information in its native form. They can instantly produce adequately formatted, human-friendly output data. It is possible, for instance, to encode 'A', 'C', 'G', and 'T' nucleotides in a two-bit field for DNA processing. Such reduction of register size to two-bits spares functional resources and allows the designer to introduce more parallelism.

Components that support floating-point arithmetic consume much logic resources of a processor. Floating-point registers and arithmetic units occupy silicon area that could be used to introduce additional parallelism. Contrary, such data manipulations like compare, bitwise logic operation, register bits swap, arbitrary register shift, and rotation are cheap in hardware implementations. Shift and swap operations do not require any logic because they need only wiring resources for example. Additionally, an FPGA designer of the custom processor can combine the above operations into any user-defined function if necessary.

Data formatting that is necessary to prepare an output file often requires a lot of data movement. It extensively employs strings manipulations such as extractions, truncations, and insertions. These types of activity are simple in an FPGA. It requires a customized register-to-register data flow only. Data manipulation of long data records requires many read/write operations in a CPU. It should be noted that data movement is a source of significant energy consumption in electronic devices. When the task of data formatting is put into an FPGA, the speedup of processing might not be experienced, because of the IO transfer limitation, but it leads to substantial energy savings.

Regular expression matching is a form of non-exact matching where the search pattern is given in the form of the character string, and it is a common task for data browsing. The string is enhanced by control symbols like: 'match any character',

'match character from the list', *etc.* The regular expression (RegEx) can be also expressed in a form of Finite-State Machine (FSM), and FSMs are used to run complex regular expression searches in software applications. Tools exist that transform RegEx to FSM graphs that can be later used by a software or hardware application. Additionally, it is possible to compile a set of many regular expressions to a single FSM. Such FSMs can lead to the complex automata with many states and transitions. The use of a powerful CPU to implement the FSM and traverse through its states is possible, but it is a waste of processor capabilities, as the most natural environment for the FSMs is logic. Consequently, the FSMs, which run in FPGAs are faster and more energy-efficient than their software counterparts.

FPGAs can be helpful to manipulate bitmap indexes. The bitmap index stores information in a binary coded format where certain attributes are marked as a bit field. The binary index is useful for enumerated field types such as a gender field: 'male', 'female' for example. This technique allows it to aggregate data records to a single binary field, and to improve further data processing. It introduces a powerful advantage for simple data types because most queries can be performed by bitwise logical operations. FPGA devices can produce any logic condition that can be written as a boolean expression to manipulate the bitmap index.

The hashing operation, which is valuable for data search algorithms, is essential in modern databases. It is a very simple function that involve a combinatorial logic or integer number operations. The hash function maps a record of data of arbitrary size to a binary value of fixed size. The binary size of the output hash is always smaller than the size of input data. Hash tables, which store hash codes, are used for rapid lookup operations where the hash is used to address the table. The hash index is another name for the hash table because hashing is one of the techniques used to index databases. It is an important experience for the idea of browsing and searching in FPGA that hashing can be very efficiently implemented in logic.

The search process is usually supported by appropriate search data structures in databases. The most common search structures that are used by software applications are binary trees, linked lists, heaps, hash tables, and tries. These structures significantly reduce the complexity of a search tasks. It will be presented in this work how some of the mentioned structures can be implemented in hardware. However, the most intensive is a linear search that is usually required when a processor scans *ad-hoc* data. *Ad-hoc* data is unstructured data, and such processing exists when network packets are analysed for example. Despite that *ad-hoc* data processing is less common than processing of indexed data, the FPGA accelerator should be appreciated for that purpose. An example of an operation that needs an exhaustive search is the SELECT JOIN operation of unsorted (*ad-hoc*) data. Also, when a search query is complex, it can be additionally parallelized in FPGA.

In conclusion, it must be stated that FPGAs are excellent for data pre-processing and low-level manipulation. It is a general remark that covers the use of FPGAs in many different areas of processing (*e.g.* video and image processing). Today, the main reason to replace CPUs by FPGAs is the substantially lower power consumption of the latter. Data-intensive calculations are IO-bound, so only a moderate speedup (about two-times) is possible in practical applications.

References

Mueller and Teubner [48] focus on the FPGA use in database systems. Their paper demonstrates the potential of reconfigurable logic, but it also recognizes some hurdles that need to be solved before the FPGA-accelerated database systems can go mainstream. The paper unveils limitations that the hardware-accelerated database processing faces.

An excellent example of the use of FPGA in search applications is provided by Putnam *et al.* [49]. The authors present reconfigurable fabric to accelerate portions of large-scale software services for Microsoft's Bing web search engine. The system ranks the candidate documents that are results from users' queries. When a server wishes to rank a document, it performs the software portion of the scoring, converts the text into a format suitable for FPGA evaluation, and then injects the material to its local FPGA. The authors argument that as data center services evolve rapidly, non-programmable hardware alternatives were impractical for the purpose. The authors report that, thanks to FPGAs, they were able to increase throughput by a factor of two in the number of documents ranked per second per server.

Another use of FPGAs to enhance web search engines is presented by Yan *et al.* in [50]. In that paper, the authors investigate FPGAs as an implementation platform for power efficient inverted index search engines, as well as a host for the accelerator of an inverted list compressor/decompressor, matcher, and ranker. Matching, which is implemented as a full binary tree, traverses the inverted lists and applies boolean operations (intersection, union, and subtraction) to the matched documents according to the input queries. The FPGA-based hardware index server achieves up to 19.52× power efficiency and 7.17× price efficiency over a commodity server processor.

Leber *et al.* [51] present a work that shows *ad-hoc* data processing in FPGA. Their paper presents the design of an application specific hardware for accelerating High-Frequency Trading applications. By using FPGAs the authors, could offload protocol and data decoding tasks from the CPU to optimized hardware blocks. It shows a 4× latency reduction in comparison to the software based approach.

Halstead *et al.* [52] investigate the use of FPGAs for relational joins. The paper presents a hash-join engine that performs hashing, conflict resolution and joining on a PCIe-attached system, achieving greater than 11× speedup over the software.

Many papers discuss the implementation of Finite-State Machines in FPGA devices. The problem splits into two approaches: Deterministic Finite-state Automata (DFA) and Non-deterministic Finite-state Automata (NFA) algorithms. NFAs are smaller in size since the number of states in an NFA is usually comparable to the number of characters presented in its regular expressions. On the other hand, the DFA works faster by proceeding one character each clock cycle, but the DFA has the potential state explosion problem for complex regular expressions.

Sidhu and Prasanna [53] present an efficient method for finding matches to a given regular expression using FPGA. Good results were obtained due to the use of NFA. Sourdis *et al.* [54] give the design methodology of regular expression pattern matching that assumes the use of a tool that automatically generates the circuitry for the given regular expressions and outputs the HDL representations ready for logic synthesis. Kumar *et al.* [55] introduce a new representation for regular expressions, called the Delayed Input DFA (D2FA), which substantially reduces space requirements as compared to a Deterministic Finite Automation (DFA). Bispo and Cardoso [56] present the synthesis of regular expressions with the aim of achieving high-performance engines for FPGAs. Hutchings *et al.* [57] developed a "regular-expression to the FPGA circuit" module generator that creates an FPGA bitstream. The authors claim that the FPGA-based string matcher exceeds the performance of the software-based system by 600× for large patterns.

1.7. Architectures for energy-efficient computing

The task to select the best architecture for the general purpose computing platform involves many trade-offs. Unfortunately, it is possible to compose the optimized computing system only if target algorithms are well-defined. Optimization often concerns the system's price/performance ratio, where price involves both purchase and maintenance costs.

The excessive CPU computing power is unnecessary in the case of memory-bound processing. Thus, power-hungry processors, which are valuable for fast execution of sequential, compute-intensive applications are not a perfect choice for data-oriented platforms. Despite, system architects often choose powerful state-of-the-art processors for data-analytic systems, and it holds because a complete, end-to-end data analysis and data-mining require compute-intensive algorithms as well. Although, data analysis involves data browsing and searching at the early processing stage, and browsing is used for the selection and preparation of data, data-mining requires machine learning and Artificial Intelligence (AI) algorithms also. However, the necessity for compute-intensity is not always the case, and many real-life services rely on IO-bound operations only.

The performance of the elements that constitute complex system should always fit. There is no rationale to put high-performance components in cooperation with slower devices, as the quality of the whole system is a quality of its worst element. Thus, the system should be balanced. In the case of a computer system, the throughput of the data storage and the throughput of the processor need to correspond to certain algorithm needs. The throughput of the processor depends on the algorithm's computational requirement so if an algorithm is IO-bound, the computing power of the CPU can be reduced with no overall system performance loss.

Thus, it is probably more rational to use less complex CPUs that are usually more energy-efficient to perform data-intensive tasks. Such processors are also simpler and cheaper. Downsizing of servers allows the increase of a number of cluster's computing nodes within the same energy, space, and monetary budget. It is possible to put more nodes into a single computer chassis and increase a node's density if lower-range processors are installed, as the cooling system efficiency limits the maximum electric power of the system rack. Consequently, the advantages are more servers in a server room and the reduced maintenance costs.

It is always uncertain to compare CPUs of different architectures, but we will compare processors of two distinct IT segments to illustrate how processor selection influences the computing node measures. We will compare an embedded system processor and server processor. Both processors provide Intel Ivy Bridge CPUs, which are fabricated in Intel's 22 nm technology. Ivy Bridge is highly sophisticated superscalar CISC architecture Table 1.1 presents a number of CPU cores, memory bandwidth, IO bandwidth and other parameters. It is clear that the embedded processor outperforms the server processor in terms of power, price, and bandwidth (memory and IO) per core. An embedded system processor has fewer cores (two vs. four), the lower clock frequency (1.7 GHz vs. 2.3 GHz), and only two memory channels (Xeon has four). It has lower maximum memory bandwidth (25.6 GB/s vs. 68 GB/s), but it offers higher memory bandwidth per core (12.8 GB/s vs. 5.6 GB/s). Power consumption of an embedded processor is 17 W, whereas a server processor requires 105 W. The more striking power efficiency difference could be expected if Ivy Bridge would be compared to a processor of a simpler, RISC-like, architecture.

The HP Moonshot system is an example of a cluster solution that comprises of light-weight computing nodes [58]. It uses a new type of server nodes that can address specific workloads. Nodes are built from chips that are more commonly found in tablets and smartphones, but allow the servers to deliver reduced energy use and a high-density footprint, at a significantly lower cost. The servers use energy-efficient CPUs like ARM's Cortex A9 or Intel's Atom processors. They draw less energy, uses less space and costs less than full-size servers. Additionally, these small-size servers can use DSPs, FPGAs, and GPUs as well.

Table 1.1. A comparison of embedded and server processors. The CPU architecture is Ivy Bridge and 22nm technology for both architectures

Processor's characteristic	Embedded	Server
Processor type	Core i7-3517UE	Xeon E7- 8850 v2
Number of cores	2	12
CPU clock frequency [GHz]	1.7	2.3
Memory type	DDR3	DDR3
Memory clock [MHz]	1333/1600	1066/1333/1600
Number of memory channels	2	4
Max Memory Bandwidth [GB/s]	25.6	68
Number of PCIe v 3.0 lines	16	32
Max PCIe Bandwidth [GB/s]	15.8	31.5
Memory bandwidth per core [GB/s]	12.8	5.6
PCIe bandwidth per core [GB/s]	7.9	2.7
Power (TDP) [W]	17	105
Price	$330	$3000

Accelerators experience obstacles that are similar to those of CPUs. Just like CPUs, they suffer a data throughput bottleneck. The GPGPU and FPGA accelerators are the most successful in speeding up compute-intensive algorithms because, in that case, the inner parallelism can be fully exploited. Simply, data is not delivered fast enough to cover all the accelerator's computing power in data-intensive problems. Consequently, the number of accelerator processing elements could be reduced with no system performance penalty and an accelerator chip that is smaller in size could be used to fit the available data transfer capabilities.

The idea of simpler CPUs and smaller accelerators meet the notion of embedded systems. Thanks to the expansion of mobile devices, semiconductor devices that integrate all computer elements on a chip are now available. They constitute so-called System-On-Chip (SoC) devices. The growth of the SoC market makes these devices an attractive alternative also for building data-processing clusters. The SoC can integrate a CPU with a GPU and recently a CPU with a DSP or an FPGA structure. Today, all the advantages of heterogeneous computing platforms are available on a chip.

The problem of the integration of an accelerator with a host system diminishes for the SoC. Close integration is now possible through an embedded system bus that connects an accelerator with the CPU subsystem and memory. No additional PCB space is necessary, and on-chip integration improves overall system reliability. For example, the HP Moonshot architecture offers a 32-bit ARM option from

Texas Instruments with an integrated DSP chip. Further, Altera announced recently that its FPGA devices would incorporate a high-performance, quad-core 64-bit ARM Cortex-A53 processor.

References

Russek and Wiatr [59] describe and compare a regular expression matching system on SoC device. The system matches a big set of regular expressions from an anti-virus database simultaneously. The authors present an algorithm that was devised for the execution on an FPGA-accelerated platform. The system offloads the majority of computations to the custom hardware, and only a small part is left for the CPU. Thanks to the FPGA accelerator, a single ARM Cortex-A9 core is sufficient to perform the software part of the algorithm for the input data stream of 1 Gbps.

Also, Russek and Wiatr [60] present an algorithm for a regular expression pattern matching system. The article focuses on the comparison of the mobile processor with the server processor. The authors offer a SoC solution that comprises of dual-core ARM Cortex-A9 CPU and FPGA structure. Thanks to FPGA, the SoC could perform as fast as Intel Xeon E5645 2.4GHz (12MB Cache) CPU. The work highlights that the problem is memory-bound, and the server processor is underutilized. The total power consumption of the SoC (2×CPU+FPGA) solution was about 2.8 W in comparison to 80 W of Thermal Design Power (TDP) of the Intel processor.

Wang *et al.* [61] considered the platform with mobile processors and FPGA. In their paper, the authors proposed a practical study of the Xilinx Zynq board for the problem of short-read mapping. Short-read mapping is the DNA sequencing problem that recently attracted attention due to the approach of the Next-Generation Sequencing (NGS) technology. The speedup analysis on a Hadoop cluster was evaluated. The authors claim that their architecture and methods have a speedup of more than 112× and is scalable to the number of accelerators.

The next paper describes an integration of the FPGA with the ARM processor inside the Xilinx Zynq SoC [62]. An eight-slave Zynq-based Hadoop cluster was built, and a customized hardware accelerator for a standard FIR filter was implemented to demonstrate the effectiveness of hardware acceleration. The input data sent to the FIR was a dataset of ASCII text files. The ARM+FPGA system achieved a 2.4× speedup compared to the ARM processor alone.

1.8. Energy efficiency of FPGAs

The higher density of transistors on the same chip makes power consumption one of the major challenges of semiconductor design. Effectively, for the applicable number of available gates, FPGAs consume a higher number of transistors compared to

their closest alternative *i.e.* ASICs. The same rule is evident as one compares FPGAs to processors. Power consumption in the CMOS technology, which includes SRAM-based FPGAs, is a sum of static and dynamic power [63]. Static power is caused by leakage currents inside transistors while dynamic power originates from the transistor toggling operation.

Switching activity causes dynamic power by the charging and discharging of load capacitance. Also, short-circuit currents that flow when the transistors toggle contributes to dynamic power. Dynamic power is given by the equation

$$P_{\text{dynamic}} = \alpha * C * V^2 * f.$$

The formula is a linear dependency on the clock frequency f and a quadratic dependency on the supply voltage V. Additionally; it takes into account the total input capacitance C of logic gates in the design. In an FPGA, the load capacitance depends on the number of logic and routing elements used in the architecture. The constant α is the toggle rate of gates and is dependent on the design and its input stimuli.

Static power is the power consumed by the FPGA when no signals are toggling. The sources of static leakage current are mainly subthreshold leakage and gate direct tunneling leakage in transistors. The static power is between 25-40% of total power dissipation, depending on the temperature, device, running frequency, and design [64].

In ASICs, power correlates to the area used. However, it is different for FPGAs, where static power correlates to the total FPGA area, and dynamic power correlates to the used FPGA area. According to Shang *et al.* [65], the FPGA's power dissipation share of routing, logic, and clocking resources are 60%, 16%, and 14%, respectively. Power dissipation of FPGA may vary significantly depending on the input switching activity.

The total electric power is defined as the sum of static power (which is dependent on temperature T) and dynamic power (which rises with clock frequency f):

$$P_{\text{total}} = P_{\text{static}}(T) + P_{\text{dynamic}}(f).$$

The total energy consumption of electronic chips, including CPUs and other computing devices, comes from their electric power and the time of operation. One can distinguish the *idle* and *active* states in the device operation. It is important to note, that the power dissipation of modern devices is significantly lower in the *idle* state than in the *active* state. This experience comes with CMOS technology characteristics (dynamic and static power) and the device's structural improvements (low-power modes, sleep states, functional unit block power downs). Therefore, the term

of average power P_{avg} is in use, which is design power time averaged over a period of time t. The total energy consumption of the processing element E_{total} can be calculated according to the formulas

$$E_{\text{total}} = P_{\text{avg}} * t,$$

and

$$E_{\text{total}} = P_{\text{idle}} * (t - t_{\text{active}}) + P_{\text{active}} * t_{\text{active}},$$

where P_{idle} and P_{active} are power dissipation in the *idle* and *active* states respectively, and t_{active} is time spent by the device on the actual processing. As P_{idle} is smaller than P_{active}, one can conclude that the speed-up of computing leads to energy savings for a given task.

This conclusion is not straightforward when the faster calculation comes from the addition of a co-processor that enhances the CPU. Such a co-processor reduces the calculation time but also contributes to the total system power. However, the lightweight custom architecture usually characterises co-processors, so their extra contribution to dissipated power is easily mitigated by the shorter processing time. The engineering practice indeed confirms this experience. The observation is both true for the System-on-Chip and board level designs. For example, Putnam *et al.* [49] introduced FPGA accelerators into their servers. To measure the maximum power overhead of introducing FPGAs, they ran a 'power virus' bitstream, *i.e.* the configuration that maximise the area and gate activity factor. Consequently, they measured a modest power consumption of approximately 23 W. The added FPGA compute boards increased power consumption by 10%, but the throughput in the production search infrastructure increased by 95% if compared to a software-only solution.

The general comparison of the FPGAs and the GPPs in terms of the power consumption is hardly feasible. It can be conducted only when a particular application is considered. However, even for the same processing task implemented on the CPU and FPGA, it is unreasonable to draw general conclusions. For example, the CPU and FPGA may differ in semiconductor technology, and no one can guarantee the optimality of the compared FPGA and CPU designs.

More difficulties exist in the assessment of the energy-efficiency of the FPGA-enabled systems. One should consider that the CPU power consumption is only about 30% of the total computer server power [66]. Other sources include AC/DC conversion losses (25%), memory (11%), DC/DC losses (10%), fans (9%), hard disk drives (6%), and others (9%). Therefore, the exclusive power comparison of the processor and FPGA device is not sufficient in practice. On the other hand, the FPGA accelerators also integrate other components. The external FPGA memory, which is usually the DDR SRAM, is a standard part of the accelerator cards for instance.

Measurement of the power consumption of separated devices in an electronic system is troublesome. Today's chips require that the power supply is connected to many different pins simultaneously, and the PCBs provide many independent power tracks to fulfill that. Usually, there are also a couple of different supply voltages required for the same device. Furthermore, the chip's pins are often out of mechanical reach, and the measuring equipment cannot be connected easily. Thus, specialised accessories are required. Consequently, the measurement of power dissipation of the whole system is an option that is frequently used in the basic elaboration of power savings.

Although phsyical measurements are difficult, a theoretical estimation of power consumption is possible. Hardware designers have access to a gate level simulator to assess the power of the given hardware architecture. The hardware description has to be provided for the simulator in an HDL to get the expected power consumption, which is calculated with respect to the given semiconductor technology. For the convenience of the designer, the higher level simulators are available in the case of FPGA technology also.

The Xilinx's Power Estimator [67], which is a calculation spreadsheet, allows a quick power elaboration. One must provide details of the FPGA design, the respective clock frequencies and the expected toggle rate to get a rough power value from the power estimator. For instance, the details of an inspected design for the Xilinx Power Estimator is provided in the form of a computer file, which is the output of the FPGA synthesis and implementation tools. The file contains the facts about the utilization of resources of a different type and thus allows the tool to determine expected power dissipation. The Xilinx Power Estimator provides the dynamic and static power separately and reports the amount of energy consumed by various FPGA resources (clocks, BRAMs, *etc.*).

The assessment of power consumption in the case of FPGAs is not a complicated task, as it can be easily done thanks to the power estimators. Unfortunately, corresponding tools, which allow the designer to emulate the processor's power consumption for a given program are not in common use. Manufacturers can estimate processor's power at the gate level, but the tools are not available for the users of commodity CPUs like Intel's and ARM's. Instead, Thermal Design Power (TDP) P_{tdp} is offered for the processors users [68]. TDP is the maximum sustained power, across a set of realistic applications, drawn under normal operating conditions, nominal voltage, and a realistic ambient temperature. There is also idle power P_{idle} and maximum power P_{max} defined for processors. P_{max} is power of the worst case (V, T) scenario, executing the worst case (synthetic) instruction. Compared to P_{max}, P_{tdp} gives some notion of the expected power consumption of the user application. Consequently, the TDP is in use when a comparison of the CPU to other computing devices is requested.

1.9. Types of FPGA-enabled architectures

A CPU is central to contemporary computing systems. The CPU is superior to every device that is present on the computer because it executes an operating system (OS). Consequently, the operation of the accelerator must be also coordinated by the processor in any hybrid computer architecture. The processor sets up tasks for the accelerator(s) and controls a process of data sending and receiving from it.

Various approaches to integration of acceleration with the rest of a computer exist. A model of the accelerator fusion decides how tight is the integration with the host system. The two-fold distinction is probably the most principal in that matter. The first approach makes the accelerator act as an IO device, and the second method assumes it is a co-processor. However, despite how close is an integration of the accelerator with the system, the CPU always initiates the accelerator's actions. Below, we will discuss several FPGA integration solutions in more detail.

The type of accelerator interface decides if it is classified as a co-processor or an IO device. Accelerators can use either a peripheral bus or a processor's system bus to exchange data. Devices that use a system bus play a role of co-processor. Otherwise, they are peripheral accelerators and act like IO devices. A co-processor is more tightly coupled with the system, and the selection of the system bus instead of IO-bus helps to reduce communication latency. A small latency is most important when many independent write/read transactions are performed, and host-accelerator data delivery incorporates short chunks of data. However, the system bus solutions are rare, and high-speed IO interconnects prevail in practical solutions.

HyperTransport (HT) interface is an example of a system bus that enables co-processor-like integration. The direct link from the CPU to the FPGA is possible thanks to the HT bus. Accelium Co-processor of DRC Computer Corporation is an example of a system that uses the HT link [69]. Accelium (Fig. 1.1) is a solution where an FPGA resides in a CPU socket, and it is highly integrated with the system. The FPGA accelerator accesses system resources like a CPU does. It can read and write from the corresponding memory banks of the host's motherboard, and it can initiate and accept read/write operations on three associated HT links. In addition, the Accelium Co-processor locally integrates Module DRAM and Low-latency RAM. The processor accesses a Local DRAM and RAM through HT links. The Accelium Co-processor of the DRC Computer Corporation is presented briefly in works by Wielgosz *et al.* [40] and Cichoń and Gorgoń [70].

The Convey HC-1 hybrid computer is another example of the FPGA empowered architecture [71]. It also uses a system bus for accelerator integration, but in contrast to the point-to-point HT connection, it uses the multi-drop Intel's Front Side Bus (FSB) interface. There are a couple of distinctive features of HC-1. First of all, there are 14 FPGAs on the coprocessor (see Fig. 1.2).

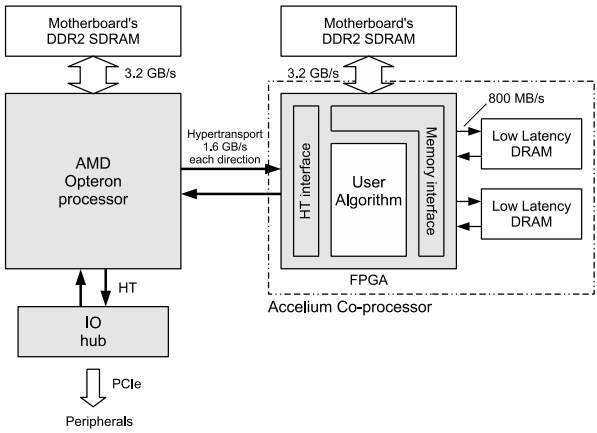


Figure 1.1. A system organization of the Accelium co-processor from DRC Computer Corporation

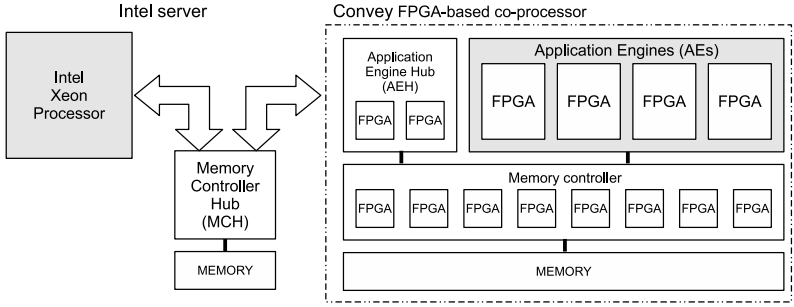


Figure 1.2. The Convey HC-1 architecture

Four FPGA chips serve as the user Application Engines (AEs), two FPGAs comprise the Application Engine Hub (AEH) that handles communication to and from the host, and eight FPGAs build the controller that provide the very fast memory interface in the Convey HC-1. The HC-1 integrates four FPGAs, which execute a user algorithm. These application engines are connected to eight local memory controllers, which provide a highly parallel and high bandwidth connection between the AEs and the co-processor physical memory. Another virtue is that the system incorporates a cache coherent Non-Uniform Memory Access (ccNUMA) across the whole system. There are two physical memory subsystems (one for the host and one for the AEs co-processor), and all physical memory is addressable by all processing elements. The ccNUMA allows all CPUs and AEs to access data in the CPU's memory and the accelerator device memory. The Convey HC-1 hybrid computer as a platform for the research in the numerical simulation was presented by Augustin *et al.* [72].

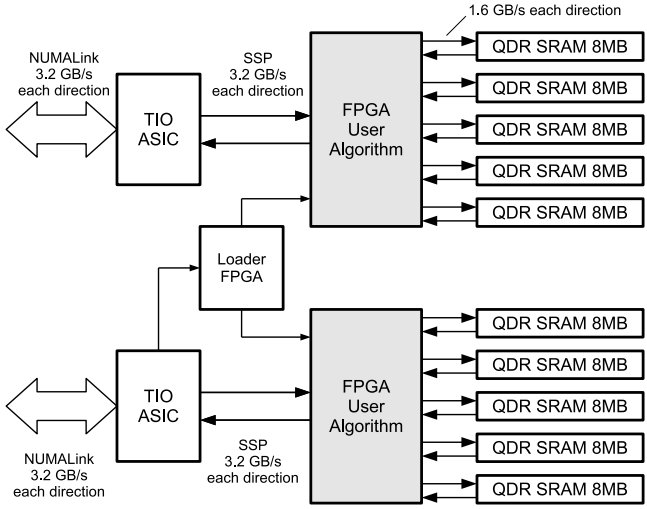


Figure 1.3. A functional block diagram of the SGI's RASC accelerator

The SGI's RASC (Reconfigurable Application-Specific Computing) is an example of the accelerator module that is integrated with a host system using a proprietary interfacing [73]. The RASC is intended for Silicon Graphics ccNUMA systems; specifically for SGI Altix systems that use the high bandwidth and low latency NUMALink 4 interconnect fabric. To address performance issues, RASC connects FPGAs into the NUMAlink fabric making them a peer to the microprocessor. Thanks to NUMALink, RASC's FPGAs are located inside the coherency domain of the computer system. The RASC basic structure is depicted in Figure 1.3. It consists of two independent FPGA modules; each features its NUMA interface and synchronous SRAM memory block. The RASC hardware module is based on an ASIC called TIO. The TIO the Scalable System Port (SSP) port that is used to connect the FPGA to the rest of the Altix system. The maximum bandwidth is up to 6.4 GB/s/FPGA, however, it gains approximately 4.0 GB/s in practical application.

Barriers exist that make the processor system bus a rare choice for the integration of an FPGA accelerator. HT is an open standard specification, but proprietary rights of the other commonly used processors interconnect become an obstacle for the vendors to integrate FPGAs as coprocessors. Further, part of the interface logic that put the FPGA to work as accelerator must be implemented in the reconfigurable array, and implementation of the complicated bus standard in the FPGA also requires a substantial amount of the FPGA's resources. Additionally, OS support for the solution that put the FPGA and CPU on an equal footing does not exist. In the mentioned DRC solution, the operations are possible thanks to a low-level function library only.

Today, the PCI Express bus (PCIe) is a standard to unify a discrete FPGA card with a host server. Opposite to the HT, the PCI Express interface is ubiquitous and hard-wired in some FPGA devices. PCIe hard-wired integration saves more FPGA's logic for user's algorithms. The PCIe also ensures cross-platform compatibility. The data throughput value of the PCIe, a high-speed communication interface, satisfies physical IO capabilities of FPGA. The data throughput, between the host and FPGA card, is not limited by the host's capabilities but by the FPGA boundary. However, PCIe-based accelerators feature higher transaction latencies in comparison to their counterparts that use the system bus.

Usually, an FPGA chip is accompanied by local memory on an accelerator board. The reason for that is that the access of the IO device to the host's main memory is restrained. The local memory is particularly important for the FPGA accelerators because their internal memory size is very modest. Also, the local memory helps to synchronize data exchange between the host and the accelerator. The memory acts as a high-capacity local cache, so the technology of fast static RAM is favourable for the purpose. One can use the cheaper and larger dynamic RAM also, but static memory is significantly faster.

Despite the technology the external memory chips are fabricated, the FPGA accelerators suffers a memory bottleneck phenomenon. Therefore, a sequential access of large memory blocks is preferable to gain good data transmission speed. The memory that is local to the accelerator buffers data transmitted from the host memory and thus allows it to aggregate read/write operations for faster communication. A multi-buffering technique allows for further throughput improvements. In the multi-buffering solution, the system allocates independent data buffers in the local memory. The host transmits data to the one buffer while the FPGA processes another buffer that was previously loaded. The FPGA and host swap buffers when the cycle finishes. The work of Wielgosz *et al.* [74] presents the advantages of the multi-buffering technique in IO-based accelerators.

Systems-on-a-Chip devices from Xilinx and Altera that implement heterogeneous CPU-FPGA platforms became popular recently. These platforms combine a dual-core ARM Cortex-A9 processor, peripherals, and memory interfaces with the FPGA fabric using a high-bandwidth interconnect bus (see Figure 1.4). Additionally, Altera SoCs offer a shared memory controller that can be accessed directly from FPGA logic. This controller feature allows for the easy integration of local memory with a co-processor. On-chip integration of a processor and an FPGA accelerator offers better reliability, more energy efficiency, and system downsizing. ARM AMBA/AXI is a bus standard that dominates the SoC solutions. The AXI interfaces provide high bandwidth, low latency connections between the CPU part and programmable part of the device. Both Xilinx's Zynq and Altera FPGA SoC solutions use this bus.

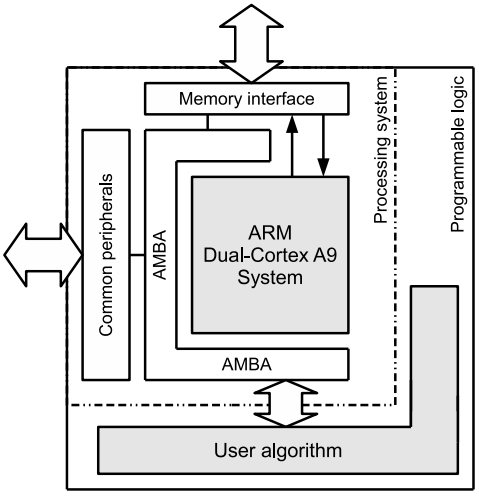


Figure 1.4. An architecture of the FPGA-enabled System-on-Chip device

Embedded solutions lead to the construction of ultra-low-power clusters. Admitting that full-scale computer servers outperform their SoC alternatives, it is interesting to notice that it is easy to migrate a design from SoC to a regular server. That simple shift is possible because both SoCs and servers use the same OSs and IPCores.

Figure 1.5 presents the generic structure of the modern IO based FPGA accelerator. Two blocks of user algorithms, depicted in the figure, are actual processing elements that perform calculations in the structure. It is usually only one algorithm block in the real systems, but two user algorithms highlight two different processing modes that are use in practical approaches. The structure contains an optional local DMA module which can improve overall system performance by offloading the CPU from performing IO read/write operations. The DMA module autonomously copies data between the host's memory and the accelerator's memory. The FPGA reads input data from the local memory in this scenario. That functionality is implemented by 'User algorithm 1' block in Figure 1.5. It is called the memory mode. It is also possible that, skipping accelerator memory, the DMA feeds host's data directly to the algorithm block. This mode is a streaming mode. The system simultaneously reads, processes, and writes data in the streaming mode. Additional FIFOs (not presented in Figure 1.5) usually makes the process steady in this mode.

A very efficient way of processing involves 'Interconnect interface' and 'User algorithm 2' (Fig. 1.5). Like an IO device, the accelerator reads data through the 'Interconnect interface', and it performs 'User algorithm 2' before data is sent to the host's memory. Direct processing of the networking packet is possible in this way for example. That kind of processing is exploited by Leber *et al.* [51].

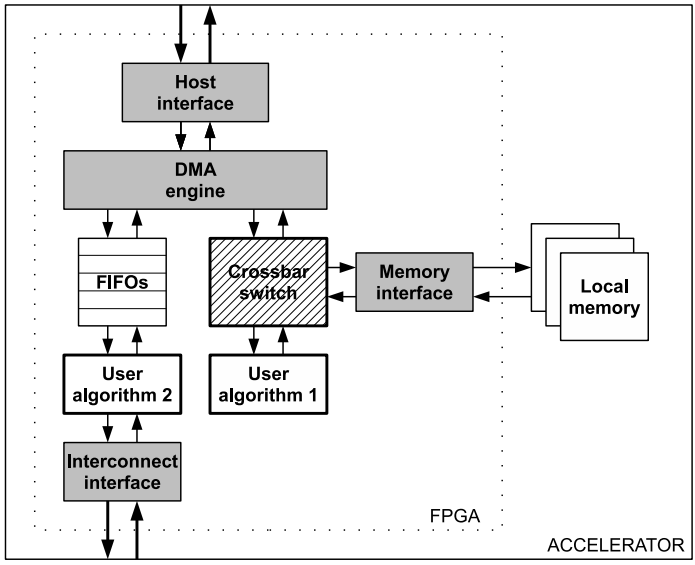


Figure 1.5. The generic structure of a modern FPGA accelerator

It is worth to note that the presented in Figure 1.5 design represents the architecture of such state-of-the-art FPGA accelerators, which are offered on the market today. For example, the architecture of the cards from Maxeler Technologies, Nallatech, Terasic, Gidel, Picocomputing, and other reconfigurable computing companies fall into the presented scheme.

Finally, the Convey HC-2 Computer is a solution worth mentioning to visualize the evolution of hybrid computing nodes. The Convey HC-2 is a derivative of the HC-1, and it integrates four FPGAs, which are devoted to executing a user application. Just like the HC-1, it also implements the globally addressable shared memory architecture. However, an important difference to the HC-1 is that the shared Non-Uniform Memory Architecture (NUMA) is implemented logically across the PCI Express connection. It utilizes the memory mapped IO (MMIO) function of the host x86 processor to map memory references to/from the co-processor's physical memory. Thus, the integration of the FPGA accelerator by the system bus was abandoned by the Convey Corporation, and IO-bus was incorporated instead.

2. Custom processor design in FPGAs

2.1. The general architecture of a custom processor

An FPGA accelerator requires a configuration bitstream that is downloaded to the reconfigurable logic array. The configuration materializes a co-processor structure that was prepared by a hardware designer. The co-processor has customized architecture that meets user's requirements. The custom processor is usually employed if a GPP cannot convey those requirements. As it was stated earlier, it is the sole distinction between a custom processor and a general-purpose processor that a custom processor has a user's algorithm wired into its structure. In other words, the structure of the custom processor reflects algorithms operations and its data flow. The principles of custom processor designs for a generic algorithm are outlined in this section.

2.1.1. Algorithm selection

The algorithm shapes the architecture of the custom processor, but not every algorithm suits well hardware implementation. Therefore, the successful hardware architecture starts with the mindful algorithm selection. The most critical stage of the custom processor design is the algorithm planning. The proper algorithm choice determines the gain of the requested goals. The good knowledge of FPGA technology helps the designer to select the algorithm that produces expected results.

The methods of custom processor design and optimization are already well-known. They have been presented and widely discussed in many works. The schemes that allow a hardware designer to convert a formal algorithm description into hardware architecture are available. Consequently, the process of custom processor design, when the algorithm is defined, can be done, in theory at least, automatically. The only problem is that the above statement is the truth only when an unconstrained design is carried. Resource and performance constraints make the design task much more complicated in practice. Constrained design is an optimization problem that might not have the solution if the requirements are too strict.

Thanks to the advancement and development of Electronic Design Automation (EDA) tools, designers no longer have to deliver hardware architecture in a structural form. At present, behavioral algorithm description in HLLs is also acceptable. Although, programming (or coding) of an algorithm for a digital circuit still requires skills of a hardware designer, it has become much easier now. Consequently, during the creation of the custom processor, the designer should focus on the algorithm rather than on coding process. That algorithmic effort is most significant because algorithms that are the easiest to be found in the literature were devised for software, not hardware, applications. Consequently, they rarely exploit advantages and abilities of the hardware implementation. The algorithms theory and practice of software solutions is well ahead compared to their hardware counterparts. The basic fact of the importance of the sole hardware algorithm creation becomes evident then.

The creation of a hardware specific algorithm is a necessary and most important step in hardware design. An algorithm formalization is a real act of creation, and the following stages are more or less procedural. The correct algorithm selection requires knowledge of target semiconductor technology. The designer must be well aware of FPGA technology in our case.

2.1.2. An example of the SQL custom processor

Following the statement that an inference of the hardware structure from the formal algorithm description is the definite task, for the clarity of further discussion, we will recall the most important steps of custom processor deduction. We will discuss a design of a custom processor for a simple SQL method as an example.

Figure 2.1 presents relationship of data in an example database. The database operations are conducted on the database tables that are named ‘employee’ and ‘bonus’. The task is to calculate a total value of the tax that is imposed on the bonuses that the company paid to employees in the country represented by an identifier ten. In our example, columns of ‘employee’ table are limited to employee’s identifiers and the affiliated country only. The ‘bonus’ table contains employee identifiers and the corresponding values of bonus. The procedure starts with an inner join operation of the input tables. That process creates ‘bonus10’ table that contains the bonus values for the employees in the chosen country. Later, ‘tax’ table is created, where tax is calculated as 20% of the bonus value minus 1250. The total tax value is summed up in the end.

Declarative languages, such as SQL, can not be directly converted to the hardware structure. A procedural description is necessary for that migration. Accordingly, the C-like code that is equivalent to the presented SQL is given in Listing 2.1. The ‘main’ procedure from the listing performs the consecutive database processing operations.

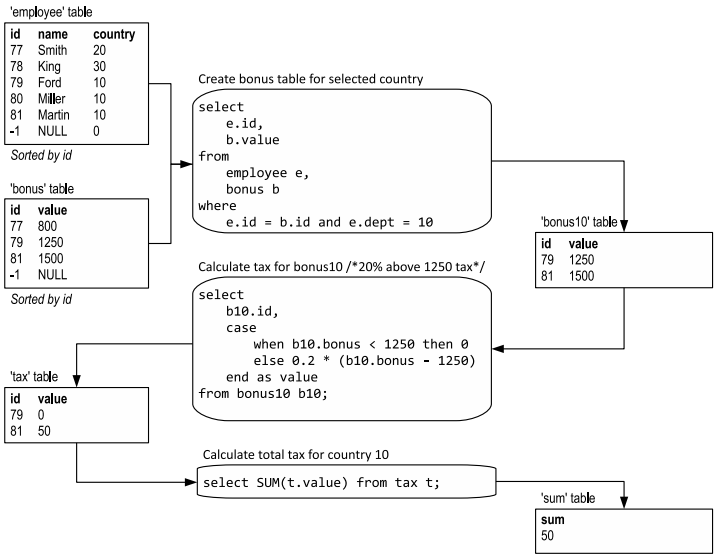


Figure 2.1. The description of an SQL problem

The database operations from Figure 2.1 that can be found in Listing 2.1 are:

- table inner join,
- selection of the country,
- tax calculation,
- the summation of the total value.

All input/output operations are programmed as an access to IO registers in our example. A direct use of the IO registers is essential because our code will end up as a hardware structure. The IO registers are represented as C-style pointers and they mimic IO ports that allow the custom processor to read and write external data.

Listing 2.1. C code for the SQL example

```
typedef struct {
    int id;
    int country;
} TEmployee;

typedef struct {
    int id;
    float value;
} TData;
```

```
TEmployee *E_in; /* IO port for emploee, bonus, and tax */
TData *B_in, *T_out;

main(){ /*1*/
/* Variable declaration */
TEmployee e;      /* To read employee record */
TData d;          /* To read bonus record and store tax values */
int tax, sum=0;

    while ( 1 ) { /*2*/
        e=*E_in; /* Read next elements ports E and B */
        d=*B_in;
        /*3*/ /* JOIN tables sorted by id. */
        while( e.id != d.id && d_id != -1) { /* Find id match */
            if( e.id < d.id )
                e=*E_in; /*4*/ /* Read next employee */
            else
                d=*B_in; /*5*/ /* Read next bonus */
        }
        if ( d.id == -1 )
          break; /* Last element is marked -1. Stop loop  */

        /* e.id == d.id, so check country */
        /* SELECT country */
        if ( e.country != 10 ) { /*6*/
            continue /* Match fail. Go for next elements  */
        }
        else {
        /* CALCULATE tax value */
            if( d.value < 1250 ) { /* Calculate tax value */
                tax = 0; /*7*/
            else
                tax = 0.2 * (d.value - 1250); /*8*/ /*9*/
        }
        d.value=tax; /*10*/
        *T_out=d;  /* Write tax data to output port */
        /* SUM */
        sum+=tax;  /* Total tax value */
    }
}
```

Among other ways a processor retrieves data for processing, the most standard method is a random access that is used to read and write RAM. However, the most efficient and frequently used scheme for FPGA co-processors is a sequential read-/write method. Therefore, each IO port implements a queue in our program. Consecutive records of ‘employee’ and ‘bonus’ tables are obtained by reading input ports E,

and B. Port T is an output port for the ‘tax’ table and the processor writes a sequence of results to port T. The necessary assumption for the join operation, which will be implemented, is that records of the input tables are sorted by the ‘id’ field. Our algorithm is a good candidate for a custom processor because it contains an outer ‘while’ loop. The loop operations are repeated many times, and that makes the loop body a computational kernel.

2.1.3. The Finite-State Machine with Data

The custom processor design starts with a conversion of a candidate algorithm to the Finite-State Machine with Data (FSMD). The FSMD is an extension of the FSM concept, and it can be easily derived from a sequential program code. The FSM does not allow for variables and arithmetic operations, so the FSMD adds these statements to conform with sequential program representation. The FSMD can be represented in a form of an Algorithmic State Machine (ASM). Figure 2.2 gives the ASM for our algorithm. It consists of states (rectangles) and state transitions (arrows).

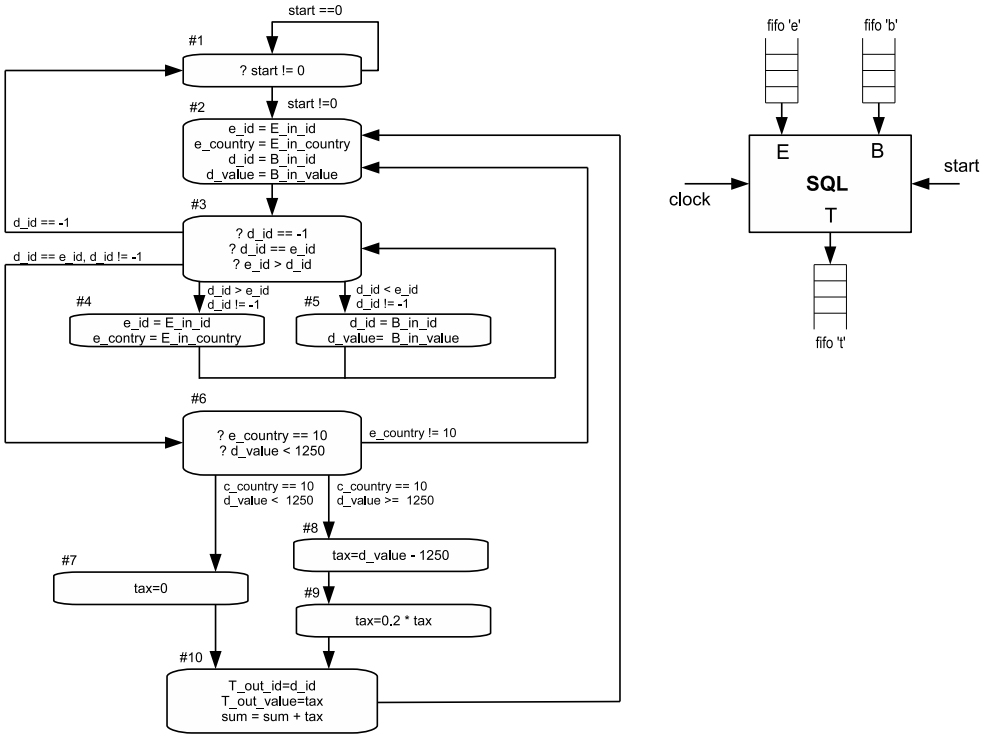


Figure 2.2. An algorithmic state diagram for an example application

The state symbols contain arithmetic operations that are scheduled for the state execution. The processor performs all the operations that are assigned to the corresponding state when it is reached. Although, the FSMD can be automatically taken from a program's sequential code, a designer should govern the conversion process. The FSMD denotes parallelism if many code statements are assigned to a single state. The designer can introduce concurrent execution of selected instructions if he wants to exploit parallelism. The FSMD creation gives a designer an opportunity to deploy his ideas about algorithm's parallelism and hugely influence the processor performance. Parallel execution leads to a performance gain, but it usually requires more processor's resources. Explicit parallelism control allows it to regulate resources and performance trade-offs. In theory, any sequential code portion without data dependency among statements can be executed in parallel but IO bottlenecks are often a limitation in practice. The HLS languages usually allow a designer to control concurrent statement execution by an insertion into the code of the special programming directives.

A classic custom processor does not execute a software code. Instead, the program execution is controlled by the FSM that is obtained from the algorithm's FSMD. The FSM governs the data path that carries arithmetic and logic operations that are encoded in the FSMD. The FSM uses control signals to activate selected functional units and switching resources (*e.g.* multiplexers) in the data path, and it uses status signals from the data path as arguments to its 'next state' function.

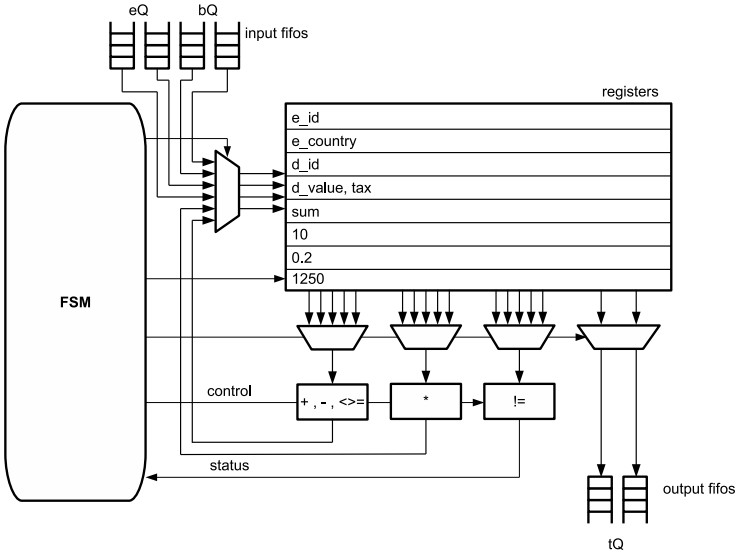


Figure 2.3. The architecture of an SQL processor

Figure 2.3 presents the custom processor architecture for our SQL example. It contains the FSM and the data path. The data path consist of registers (‘e_id’, ‘e_country’, *etc.*), multiplexers, and functional units (‘+,-,<>=’; ‘*’; ‘!=’).

The structure like presented in Figure 2.3 is usually taken from the Register Transfer Level (RTL) description of the hardware operations [75] in practice. A custom processor designer delivers the RTL definition that is formalized by Hardware-Description Languages (HDL). However, the processor architecture was created manually from the FSMD in our case. Today, the behavioral description is also possible in HDL, but there is no designer's control over the operation scheduling and resource allocation in that case. We are not discussing behavioral synthesis here.

2.1.4. The controller and data path

The data path consists of registers, functional units, multiplexers, IO ports, and wiring connections. The registers are used to store algorithm's variables. The functional units perform arithmetic and logic operations. The multiplexers connect the registers' outputs to the functional elements' inputs and the functional elements' outputs to the registers' inputs. Also, multiplexers wires input/output ports with the registers and the functional units.

Data movement in the data path is governed by the FSM's output lines. The control is possible thanks to the control inputs of the data path. Let's consider state ten of our ASM. To add a content of ‘tax’ register to the content of ‘sum’ register and store the result back in ‘sum’‘ register, the following must be set up by the controls:

- the ALU's input multiplexer must be set to feed outputs of ‘tax’ and ‘sum’ registers to the ALU,
- the ALU must be put in an ‘add’ mode,
- the ALU's output multiplexer must be set to direct an output of the ALU to an input of ‘sum’ register,
- ‘load enable’ control of the ‘sum’ register must be active,
- the clock cycle must be executed.

Status signals are a feedback from the data path to the FSM. They allow it to implement conditional program statements. For example, the comparator's outputs are FSM's input signals that control next state logic of the Finite-State Machine in the states three and six.

There are ten states in our ASM for the SQL algorithm. Each FSMD state corresponds to a single execution step of the sequential custom processor. The processor performs all operations denoted inside an ASM's state rectangle simultaneously. The capacity of the data path must allow for a concurrent execution of a few statements

in each state. For example, the country identifier is compared to ten and the bonus identifier is compared to 1250 value in state six. Thus, two separate comparator units in the data path are necessary. Besides, an extended comparator (with implemented 'equal' and 'greater-than' functions) and simple comparator (with 'equal' function only) are necessary to perform operations in state three. The extended comparator gives the result of 'd_id' and 'e_id' comparison in a form of two outputs: 'greater-than' and 'equal'. The simple comparator compares 'd_id' variable to -1 value.

The table of operations usage helps to determine functional units that should be implemented in a data path [76]. The operation usage for our example is presented in Table 2.1. The table summarizes the functions performed at each algorithm state. The table exposes that as much as two operations are performed in a single algorithm's state. Thus, the minimum number of functional units for a data path is two also. Such a simple assignment is possible when multi-functional units are possible. Similarly to an ALU, multi-functional units can perform more than one type of operation. The introduction of multi-functional units allows it to reduce custom processor resources. It is a consequence of functions grouping into single units, and it improves resources utilization of the processor during program execution because it eliminates idle states of processing elements. That technique is called functional units sharing.

However, a functional unit cannot perform an arbitrary operation that is requested by the algorithm. An adder cannot multiply, and multiplier cannot compare values for example. This limitation leads to the increase of the number of necessary functional units in the data path. Table 2.1 shows that state three requires at least two separate comparators for example. Additionally, states eight, ten, and nine require a subtraction, addition, and multiplication respectively. One could easily group a subtraction, addition, and comparison to be the single unit. However, the second compare units would have to implement the multiplication to meet the requirements for two functional units. The processing elements that integrate a multiplication and a comparison are impractical in digital design practice. FPGAs implement multipliers as a separate dedicated DSP blocks for instance. Thus, an independent multiplier is more practical here.

A variable lifetime table determines the number of required registers in the data path. Table 2.2 gives the variable usage for the SQL example. We denote the variable to be alive (active) in all states between its first assignment and last usage. The total number of variables can be higher than the number of registers because sharing of registers is possible. For example variables 'tax' and 'd_value' have non-overlapping lifetimes, and they can share a register. The maximum number of simultaneously active variables in the lifetime table determines a maximum number of necessary registers. The sharing of registers is an important tool for custom processor optimizations. Properly planned assignments of variables to registers allow it to reduce the number of necessary connections in a data path.

Table 2.1. The table of operation usage in the SQL example

Cycle	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
compare			2			2				
subtract								1		
addition										1
multiply									1	
Total			**2**			**2**		**1**	**1**	**1**

Table 2.2. The table of variable lifetime for the SQL example

Cycle	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
e_id		✓	✓	✓						
e_country		✓	✓	✓	✓					
d_id		✓	✓	✓	✓	✓	✓	✓	✓	✓
d_value		✓	✓	✓	✓	✓	✓	✓		
tax									✓	✓
sum		✓	✓	✓	✓	✓	✓	✓	✓	✓
Total		**5**	**5**	**5**	**4**	**3**	**3**	**3**	**3**	**3**

The throughput and latency are well-known parameters that characterize the performance of processing systems. The latency provides a measure that tells how long it takes, for a processor, to finish the processing of input data. The processor of our interest (Fig. 2.3) has a variable processing time. It consumes from two to six clock cycles to complete one processing round. The sequence of states two, three, six, eight, nine and ten constitute the six clock cycle processing path. The pairs of states [three, four] and [three, five] process an input record in two clock cycles only. Correspondingly, the throughput is an amount of input data elements accepted by a processor during a given period of processing. The latency and throughput do not correspond as the processor may be able to take a new data unit before it finishes previously started rounds. Our SQL processor receives a new record only when it completes prior processing. Thus, the latency and throughput are equivalent here. The minimum throughput is one data element per six clock cycles in our example.

References

Books by Gajski & Kleinsmith [76], Vahid & Givargis [76] and Giovanni de Micheli [77] present the practical approach to custom processor design. Micheli's book covers techniques for synthesis and optimization of digital circuits thoroughly, and other books offer a more descriptive approach. Gajski's book [76] delivers various optimization techniques additionally. One can find the overview of strategies for the reduction of the number functional units, registers, and data path connections.

Gajski and Ramachandran [78] introduce the Finite-State Machine with Data model, which forms the basis for hardware synthesis. One can also find an introduction to FSMD and Algorithmic State Machine (ASM) in Baranov's work [79].

Vahid's book [75] offers modern approach the role of Register Transfer Level (RTL) description in contemporary digital circuits design.

Scheduling and resource allocation problems were stated first by Hafer and Parker [80] but a good overview is also offered by [77]. Leive and Thomas [81] sparked a research on a problem of modules selection for use in an automated digital system design.

2.2. Algorithm scheduling

A sequential algorithm consists of instructions that are executed by a processor in defined order. However, a superscalar processor can change the sequence of instructions to gain better performance. Superscalar processors introduce out-of-order execution to improve the use of multiple execution units and avoid their idle states. A processor amends the order of instructions dynamically during its run-time. Similarly, the hardware designer may want to reschedule algorithm's instructions to increase the efficiency of the custom processor. The scheduling that is done prior to custom processor design is static; in difference to scheduling that is performed by a superscalar processor.

The order of instructions in the algorithm can be changed if a result of the previous instruction is not an argument for the later instruction. In other words, there is no data dependency between swapped instructions. It is also worth noting that data-independent instructions can also be executed in parallel.

One builds the Data Flow Graph (DFG) to discover parallelism that is hidden in the sequential algorithm. The DFG is also called the Data Dependency Graphs (DDG). The DDG is a direct graph whose vertexes represent algorithm statements and edges show the data dependence of adjacent instructions. No instruction can start until its every ancestor finishes its operation. It is important to say that the Data Dependency Graph covers assignment expressions only. It does not represent the algorithms' control statements that are the branch and conditional constructs.

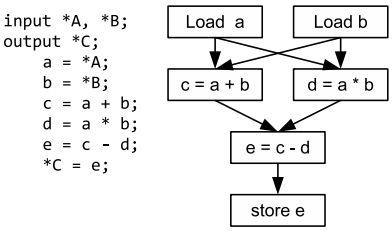


Figure 2.4. A sample code and its Data Dependency Graph

A simple sequence of assignments and their data dependency graph is presented in Figure 2.4. For example, statements ‘c=a+b’ and ‘d=a*b’ are independent and can be executed in parallel, but assignment ‘e=c-d’ has to wait until their completion.

Instruction scheduling assigns instructions to clock cycles in custom processor design. The DDG allows it to schedule the instruction in the correct order. The example plan of the instruction order for the algorithm is proposed in Figure 2.5a. For the synchronous digital circuits, the scheduler assigns each instruction to the particular clock cycle. However, data dependency is not the only constraint that drives the scheduling. It is also the number and type of available functional units that influence how instructions are ordered. An assignment of two additional instructions for the same clock cycle requires that at least two ‘add’ units are present in the data path. The second additional operation has to be postponed if only one ‘add’ unit is available. Figure 2.5a shows an unconstrained schedule where the limit of functional units does not exist. Unconstrained scheduling is often a case for FPGA technology as a designer can freely customize a data path for an algorithm.

The scheme in Figure 2.5a assumes that instructions are completed in one clock cycle. That is rarely the case, so the next scheduling example assumes that multiplication takes two clock cycles. Figure 2.5b gives the corresponding solution.

The As-Soon-As-Possible (ASAP) schedule is presented in Figure 2.5b. In ASAP, each instruction is scheduled to start immediately when all its predecessors are ready. As-Late-As-Possible (ALAP) scheduling scheme is also possible. In that scenario, a statement is delayed as long as the delay does not postpone its derivative instructions. Figure 2.5c outlines the difference between ASAP and ALAP processes. Both ASAP and ALAP are unconstrained scheduling methods. Total execution time for ASAP and ALAP are the same, and it provides the minimum possible latency of the algorithm. However, each instruction can be scheduled differently in ASAP and ALAP. The comparison of ASAP and ALAP schedules give execution boundaries for each instruction for the constrained scheduling. The constrained schedule gains minimum algorithm’s latency if it meets these boundaries for each instruction. The algorithm is executed in the fastest possible time in that case [77].

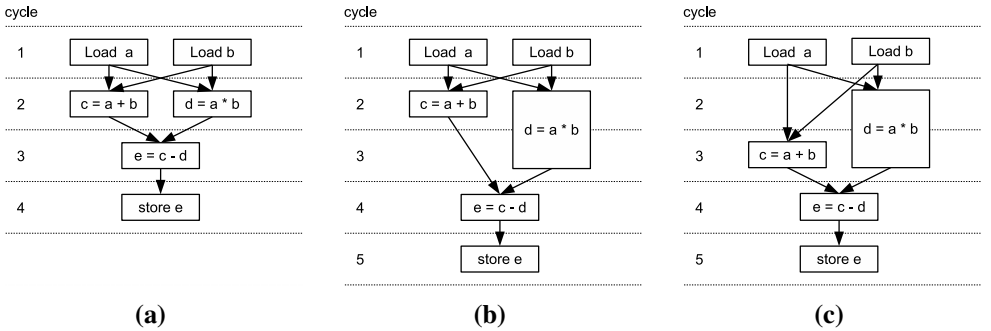


Figure 2.5. Examples of instruction scheduling: a) Unconstrained; b) ASAP; c) ALAP

References

The reader can find the definition of the DDG and the formal algorithms for ASAP and ALAP methods in [77].

The very first works about scheduling regarded software. Fernandez and Lang [82] considered the scheduling of a set of tasks with precedence constraints and known execution times into a set of identical processors. Fisher [83] developed "trace scheduling" as a proposal for the code scheduling problem.

For hardware, scheduling problem was formalized by Hafer and Parker [84]. Authors designed the allocator that selects registers and data operators and interconnects them with data paths to implement the specified behavior. An Integer Linear Programming (ILP) model for the scheduling problem is presented by Hwang and Lee [85]. In addition to time-constrained scheduling and resource-constrained scheduling, a scheduling problem called feasible scheduling is constructed.

2.3. Loop pipelining

Loops mark the most computationally intensive parts of algorithms. Many loop optimization techniques for parallel computing exist. However, loop parallelization is possible if data dependency between subsequent loop iterations does not exist. One can consider so-called loop dependency (or dependence in loops) property [86].

Available data throughput decides whenever loop parallelization improves system performance. For example, the loop: **for** (i=0; i<N; i++) $output[i]=i^2$; can be executed simultaneously by N processing elements in a single processor step. All square values can be available in one clock cycle, but it does not lead to a performance gain if the system is capable of performing one output operation per clock cycle only.

Likewise the parallelism, the pipelining is another method to run more than one instruction simultaneously. Opposite to the parallelism, pipelining is not IO throughput dependent. Pipelining is a form of parallelism that works despite a bottleneck of IO channels. Pipelining allows it to reduce register-to-register data movement and the processor's clock frequency. These features are followed by the energy efficiency of the custom processor.

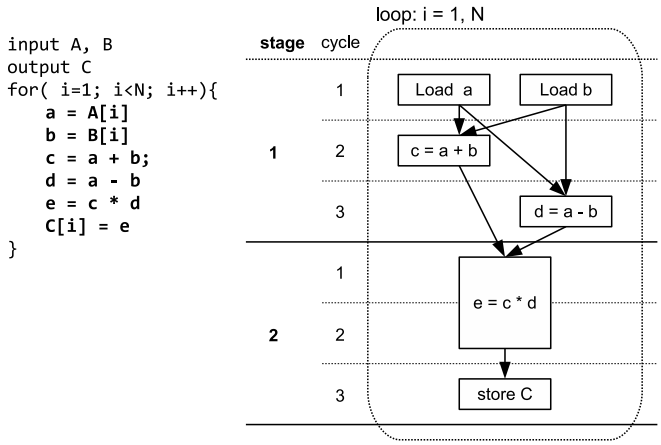
The execution of independent sets of input data overlaps in algorithm pipelining. Algorithm's outer loop makes the particular space for pipelining. One can consider loop pipelining as a simultaneous overlapping execution of the separate loop statements for the consecutive loop iterations. In hardware, the loop pipelining induces the introduction of the additional storage and functional units elements if compared to sequential loop implementation. Most of the stream processing algorithms, which receive input data in the form of a stream and generate the stream of output data, are suitable for pipelining optimization.

We will consider the loop and its code schedule that is given in Figure 2.6a. The designer divided the loop body into two separate pipeline stages. The loop processes N sets of input pairs (a_i, b_i). It is convenient to denote the variables according the iteration number in pipeline analysis ('a' became 'a_i' for example) because iterations are separated in time. This approach is universal for arrays and single variables that are used in the code. The progress of work is as follows:

1. Stage one starts the calculation for the first pair (a_1, b_1),
2. stage one passes the pair (c_1, d_1) to stage two and starts to process the pair (a_2, b_2) after three clock cycles,
3. stage two finishes with the final result e_1 and stage one with the result of the pair (c_2, d_2) after next three cycles,
4. stage one passes the pair (c_2, d_2) to stage two and starts to process the pair (a_3, b_3),
5. *etc.*

A table in Figure 2.6b gives the detailed explanation of the pipeline work. The table contains the iteration number processed by functional units in each clock cycle.

The design of the pipelined system requires that the designer creates a separate processing unit for each processing stage. The output ports of the previous stage are connected to the inputs of the next stage, and so the system works in a passage. Consequently, a pipelined solution requires more resources than a sequential one. However, the expense of additional resources benefits in the higher overall throughput of the system. The throughput of the sequential system is one input record per six clock cycles in our example. The pipelined, two-stage system processes one input set in three clock cycles only. Though, the latency is six clock cycles for both the solutions.



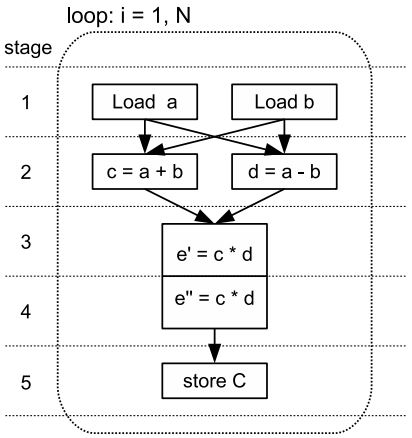
(a)

i	1	2	3	4	5	6	7
Load a(i)	1			2			3
Load b(i)	1			2			3
c(i)=a(i)+b(i)		1			2		
d(i)=a(i)-b(i)			1			2	
e'(i)=c(i)*d(i)				1			2
e''(i)=c(i)*d(i)					1		
Store C(i)						1	

(b)

Figure 2.6. An example of a loop and its two-stage pipelining: a) the program code and the DDG of a loop body; b) processing of input sets in each clock cycle

Figure 2.7 introduces the five-stage pipeline architecture for our example case. Its throughput is one data set per one clock cycle. Thanks to the new scheduling scenario, where addition and subtraction are performed in parallel, the latency is reduced to five clock periods. Additionally, the method of functional unit pipelining is introduced in this architecture. The multiplier that previously required two clock cycles to complete the operation is pipelined now. The multiplier has the throughput of one operation per one clock cycle and the latency of two clock cycles. Opposite to algorithm pipelining, functional unit pipelining requires the new hardware library of functional units. Combinatorial processing elements, sufficient for sequential custom processors and algorithm pipelining, must be redesigned now. One must put registers across its internal data paths and make them work in a pipeline. Boundaries of algorithm pipeline stages are cut across processing elements in functional unit pipelining.



Iteration performed by instructions in the pipeline

i	1	2	3	4	5	6	7
Load a(i)	1	2	3	4	5	6	7
Load b(i)	1	2	3	4	5	6	7
c(i)=a(i)+b(i)		1	2	3	4	5	6
d(i)=a(i)-b(i)		1	2	3	4	5	6
e'(i)=c(i)*d(i)			1	2	3	4	5
e''(i)=c(i)*d(i)				1	2	3	4
Store C(i)					1	2	3

Figure 2.7. An example of an algorithm and its five-stage loop pipelining

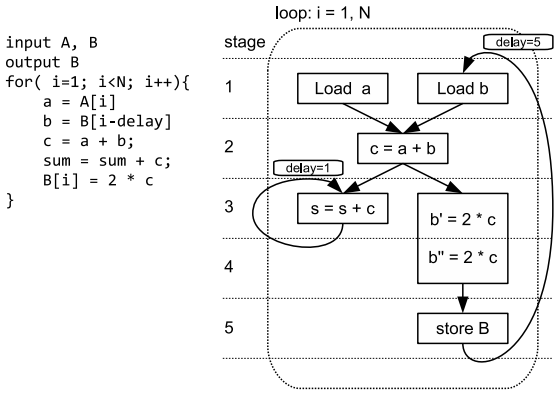
Loops with dependencies

In special cases, loop pipelining is also possible if dependencies exist in the loop *i.e.* circular paths are present in a DFG of the loop body. Let's consider the code from Figure 2.8a.

Variable 'b' gets the result of the preceding loop iteration in line six. The variable 'sum' also takes its previous result. The above settlement causes a circular dependency in the data flow graph. Obviously, the arguments must be already calculated when they are assigned during the following loop cycle. Pipeline processing introduces delay D between subsequent iteration.

When stage number S_a executes iteration number I_a, stage number $S_b = S_a + D - 1$ executes iteration $I_b = I_a - D + 1$. In can be seen in the table in Figure 2.8b that when stage one processes iteration five ($S_a = 1$, $I_a = 5$), stage five processes iteration one ($S_b = S_a + D - 1 = 5$, $I_b = I_a - D + 1 = 1$, $D = 5$). The iteration delay is defined as $D = D_{\text{iter}} = I_a - I_b + 1$. The pipeline delay is defined as $D = D_{\text{pipe}} = S_b - S_a + 1$. One can derive the iteration delay from an assignment statement 'b=B[i−delay]' in the example code, where 'delay' is equivalent to the iteration delay D_{iter}. The iteration and pipeline delays have very practical implications for pipelining feasibility. Loop pipelining cannot be introduced if the pipeline delay is higher than the iteration delay.

Let's consider line four of the loop body in the example. It is trivial but the usual case of the circular dependency. The statement 'sum=sum+c' can be elaborated as 'sum[i]=sum[i−1]+c'. Thus, the iteration delay is one. Further, S_a and S_b are equal, so pipeline delay is also one. The pipeline delay does not exceed the iteration delay, so loop pipelining is possible.



(a)

i	1	2	3	4	5	6	7
Load a(i)	1	2	3	4	5	6	7
Load b(i-5)	1	2	3	4	5	6	7
c(i)=a(i)+b(i)		1	2	3	4	5	6
s(i)=s(i-1) + c			1	2	3	4	5
b'(i)=3 * c(i)			1	2	3	4	5
b"(i)=3 * c(i)				1	2	3	4
Store B(i)					1	2	3

(b)

Figure 2.8. An example of the pipelining of a loop with data dependency: a) the program code and DDG of a loop body; b) processing of the input sets in each clock cycle

References

Pietroń, Russek, and Wiatr [87] discuss the loop pipelining. The paper considers loop profiling and data dependency to implement algorithms efficiently in FPGA circuits. Loop pipelining for hardware synthesis is discussed in Fingeroff's book [88]. Among other issues, the author also presents the problem of data feedback (cycle dependency) during loop pipelining. The reader can find the descriptions of the algorithm pipelining and the functional unit pipelining in the book of Gajski and Kleinsmith [76]. Selected algorithms for scheduling pipelined functional elements are presented in Micheli's book [77].

The problem of scheduling a loop in a pipelined fashion such that the iteration time (turnaround time) is minimized is considered in the paper of Lee *et al.* [89]. Jeon and Choi [90] present a hardware-software partitioning algorithm that exploits a loop pipelining technique. The proposed loop pipelining technique is an adaptation of a compiler optimization technique for instruction level parallelism. The authors

considered the resource-constrained scheduling of loops with inter-iteration dependencies. Chao *et al.* [91] present rotation scheduling for scheduling cyclic DFGs using loop pipelining. A loop is modelled as a data flow graph, where edges are labelled with the number of iterations between dependencies.

The paper of Gielata, Russek, and Wiatr [35] gives the practical example of the technique of the algorithm pipelining. The authors pipelined Rijndael's cryptographic algorithm to achieve the spectacular throughput. The architecture processes data in independent blocks of 128 bytes. The latency is eleven clock cycles.

An example of the pipelined functional element, one can find in the paper of Wielgosz *et al.* [92]. The authors present the implementation of the double precision exponential function that is pipelined to enhance the throughput.

2.4. Control statements pipelining

We considered program codes that did not contain any jumps or their targets in Section 2.2 and 2.3. The codes, so far, were a straight piece of a processor's program that contained assignment statements only. Besides simple assignments, the conditional and loop statements are possible also. A compiler generates assembly jumps and conditional branch instructions from the constructs 'if', 'case/switch', 'while', 'do-until', and 'for', which are available in high-level programming languages. A fully functional program can use the simple assignments and branches only. Consequently, it is enough for a custom processor to implement those types of instructions.

One can organize a program code into blocks, where each block starts with a jump target and ends with a jump instruction. It is possible to construct a Control Flow Graph (CFG) for such an organized program code. The graph vertexes represent the code blocks in the CFG, and the graph edges denote the possible program flow. When the CFG and the DFGs of its blocks are combined, the Control and Data Flow Graph (CDFG) can be created (see Fig. 2.9b).

Now, we will consider the feasibility of pipelining of the loop body that contains the control statements. As we already stated, algorithms pipelining is crucial in designing of custom processors for data-intensive problems because it is immune to IO bottleneck.

2.4.1. Conditional statement pipelining

Let's discuss pipelining of a conditional statement first. Figure 2.9a gives the example of a code of the outer loop where a loop body contains the 'if-else' construct. The loop body can be expressed as the CDFG that is presented in Figure 2.9b.

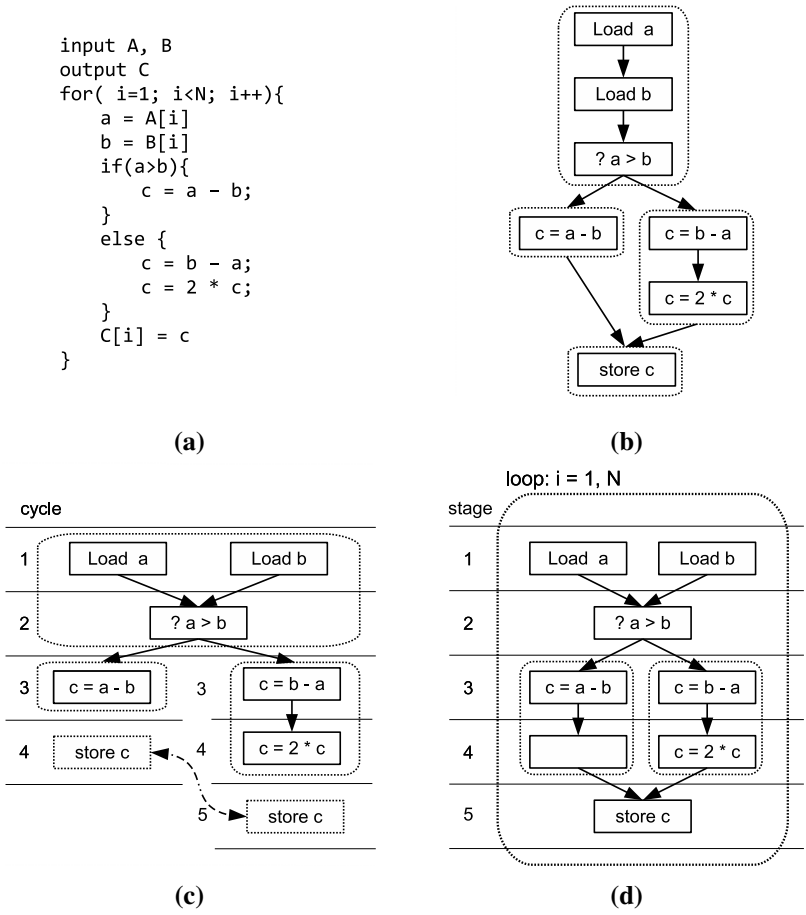


Figure 2.9. The scheduling of control statements: a) the loop with a control statement; b) the Data Flow Graph of a loop; c) a variable loop latency; d) a constant loop latency

There are two independent program execution paths that start with '? a>b' compare instruction in listing in Figure 2.9a. Figure 2.9c presents the schedule of the instructions of the CDFG. Apparently, the introduction of the conditional jump into a program can result in the variable execution time of the single loop iteration. The execution of the whole loop body takes either four or five clock cycles. The variable execution length of the program is an obstacle that inhibits pipelining. The 'Store c' node has to consume exactly one data element at each clock cycle. In other words, the 'Store c' instruction must be assigned to a particular pipeline stage. For this reason, an 'if- then-else' or 'case/switch' constructs can be pipelined when a pipeline length of each conditional branch is the same. The length of both branches of the condition

'? a>b' have to fit to fix the pipe's position for 'Store c' instruction in the example. The equalization of a program's execution paths is possible by the addition of 'do-nothing' instructions. An empty instruction was added after 'c=a−b' instruction in the scheduling that is presented in Figure 2.9d.

2.4.2. Loop statement pipelining

When the outer loop contains the inner loop statements, it is possible, in selected cases, to pipeline its body. The loops that are a part of an outer loop body are referred to as inner loops. The serialization of the inner loop is possible thanks to loop unrolling. In loop unrolling, the loop statement is removed, and each loop iteration is explicitly expressed in the program code. Consequently, the number of the loop iteration must be static to enable unrolling. Figure 2.10 gives the simple example of the unrolling. The example program invokes two 'for' loops (Fig. 2.10a). Interestingly, the first 'for' statement features no loop dependency but the second 'for' statement contains loop dependence. When unrolled (Fig. 2.10b), iterations of the first loop could be executed in parallel, but iterations of a second loop cannot. The DFG of the code allows to explore parallelism (Fig. 2.10c). Despite that, both the first and second loop can be pipelined. A decision, whenever to execute the unrolled instructions in parallel or pipeline, takes place during the instruction scheduling process. The possible schedule for the example is given in Figure 2.10d. We can notice that despite there is no data dependency between these operations, assignments 'c1=a+2' and 'c2=a+3' run one after another in our example.

The unrolling of static loops is possible if the iteration count is modest. The long loop may consume the prohibitive amount of hardware resources when unrolled. A partial loop unrolling is possible in some situations. For example, a loop

```
for( i=0; i<N; i++) {
  /* loop body */
}
```

is equivalent to the two nested loops

```
for( i=0; i<N; i+=step)
  for( j=0; j<step; step++) {
    /* loop body */
  }.
```

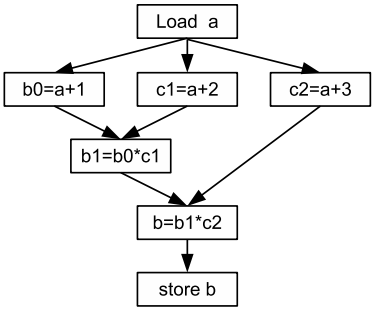
Now, the inner loop in the second expression can be unrolled and then parallelized. The outer loop is executed sequentially while the inner loop is executed in parallel, so the scenario decreases the latency by a factor of 'step'. Unfortunately, the custom processor has to execute a loop in sequence if the loop has a dynamic iteration range.

```
input A
output b
for( i=1; i<N; i++){
array b,c
    for(j=0;j<3;j++)
        c[j]=A[i]+j+1;
    B[0]=c0;
    for(k=1;k<3;k++)
        B[k]=B[k-1]*c[k];
    b=B[2];
}
```

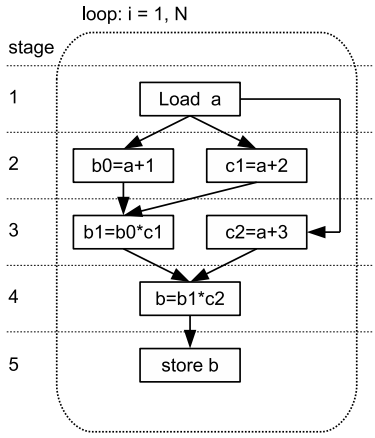
(a)

```
input A
output b
for( i=1; i<N; i++){
var c0,c1,c2,b0,b1
    c0=A[i]+1;
    c1=A[i]+2;
    c2=A[i]+3;
    b0=c0;
    b1=b0*c1;
    b=b1*c2;
}
```

(b)



(c)



(d)

Figure 2.10. An illustration of loop unrolling: a) the loop with loop statements; b) the code after loop unrolling; c) DFG for the unrolled code; d) scheduling of the unrolled loop

2.4.3. Pipelining of the CFG

Although, pipelining conserves the system's latency, it increases the throughput; therefore, it is a very valuable for performance enhancement. The algorithms that exhibit a clear outer loop are good candidates for pipeline execution. Unfortunately, the control statements that are included in the majority of algorithms do not allow a designer to introduce pipelining freely. The algorithms with control statements can be still pipelined; however, with an additional constraint that control statements are entirely incorporated into the single pipeline stage. Thus, every pipeline stage performs its code sequentially and passes results to the next stage. In other words, CFG can be cut across the edge that completely separates vertexes of a newly created CFG. Figure 2.11 shows how to divide an example CFG. When the edge is removed (the dashed line), no path can exist between nodes of the upper and the lower graph.

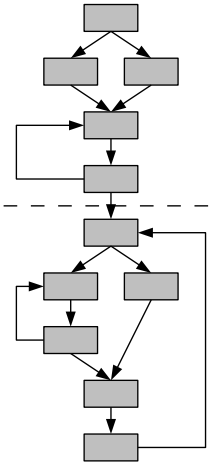


Figure 2.11. An example of the Control Flow Graph

It is important to balance throughputs of the pipeline blocks. Otherwise, we can expect underutilization of faster blocks. As we know, a designer can adjust the processor's performance by controlling the number of its processing units, so the throughput of pipeline stages can be equalized when a processor's architecture is planned. However, problems with inter-stage communication emerge if blocks contain control statements. Due to the variable execution path in code branches every stage in such a pipeline has the fluctuating throughput and latency. The communication channel between blocks requires handshake signalling in that case. It can also be helpful to include FIFOs between the pipeline stages to make the communication smooth. Figure 2.12 presents the two-stage pipeline realization of the SQL processor that was introduced in Section 2.1.

References

Kavi, Buckles, and Bhat [93] presented the formal definition of the DFG. Jong [94] defined a new data flow graph concept. It was established to be used for architectural synthesis, as well as the verification of a system. Amellal and Kaminska [95] outlined a new representation of the behavioral specification of algorithms. Also, Amellal and Kaminska [96] presented a CDFG model for system representation that includes a representation of conditional branches. Wu *et al.* [97] present a hierarchical CDFG model. The hierarchical feature can be straightly obtained by extending the definition of nodes. It is demonstrated how to build basic control constructs of branches and loops. The hierarchical CDFG model can capture the design information from the source file specified by VHDL or C language. Loop unrolling for FPGA implementation is discussed by Pietroń *et al.* [87] also.

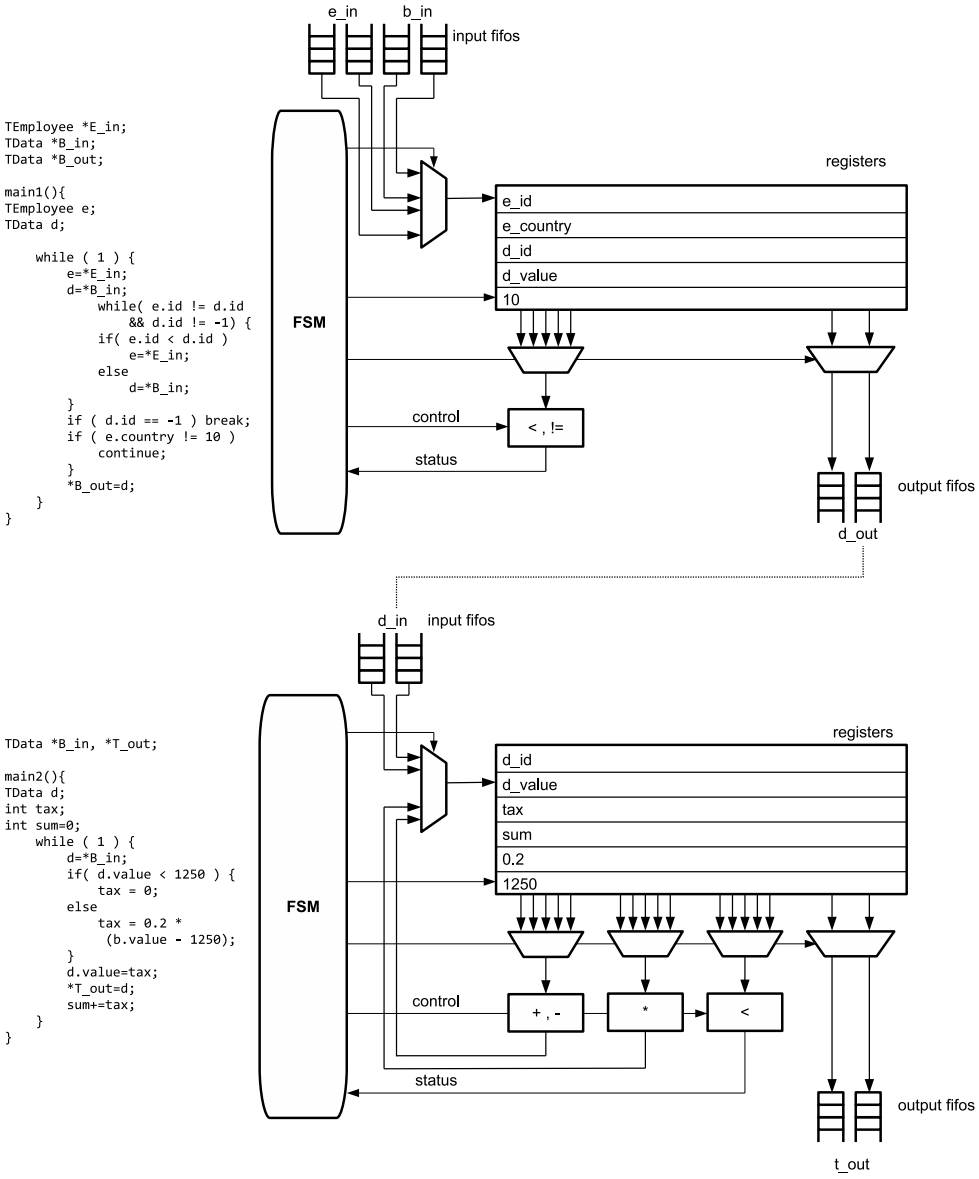


Figure 2.12. A two-stage SQL processor

2.5. Memory handling

We have already stated in Section 1.2 that the performance of the computer memory is one of the most important factors that impact the overall system performance. Computational power is delivered to the user by the CPU, but it needs user data to make calculations useful. The system of computer memory that stores data is hierarchical. The hierarchy typically consists of the register, cache, RAM, and mass storage levels. The significant role of the memory also applies to other computational elements, including the FPGA accelerators. Each processing element has to retrieve data for its calculations, and it should be able to do it fast. Good speed is the first reason to optimize memory access. Another aim is energy efficiency because data movement is the most energy-consuming action in the computation. Therefore, the memory architecture and memory access strategies for FPGA technology are the sole topic of this section.

An FPGA accelerator relies on a memory system that is, similarly to a CPU, organized in a hierarchy, but the FPGA memory architecture is different then its CPU equivalent. At the time of writing this text, the state-of-the-art Intel Xeon E7-8890v2 processor that is fabricated in 22-nm technology, offers 296 Mb (37 MB) of cache memory. At the same moment, Xilinx Virtex-7 XC7V2000T that is a high-performance 28-nm FPGA device contains in total 47.5 Mb of the internal memory only. The FPGA's internal memory can be considered as the CPU cache counterpart. It is organized in blocks that are hard-wired, 36 kb, dual-port, synchronous SRAM elements. These FPGA memory blocks are called Block RAMs (BRAMs). XC7V2000T delivers 2,584 of BRAMs for example. Each BRAM act separately or it is combined with other BRAMs into larger memory blocks by the FPGA implementation software. It is always difficult to compare processors and FPGA devices because both exist in a variety of different types, but we can conclude that FPGAs offer an order of magnitude smaller internal memory than processors.

Shortage of SRAM is among major FPGA's features that cause difficulties in making FPGA accelerators more competitive to CPU processors. Tasks that exhibit repeated access to the same portion of data reveal the lack of big local SRAM memory in FPGAs. Computing problems that easily fit the CPU cache exceed the available FPGA BRAM size. On the other hand, the BRAM memory has an advantage over the CPU cache. Each BRAM has a separate read/write interface, and that allows a hardware designer to implement fine-grain memory architecture. Independent access is possible to each BRAM, and the designer can take advantage of the massive total memory bandwidth. Thanks to that, parallel execution units do not suffer data starvation once data is located in that distributed RAM. However, the above characteristic is usually in favour of computationally-intensive problems.

2.5.1. Local arrays of data

Some program variables have a short lifetime, and they are necessary only to compute a single or few algorithm's statements. In other words, they are local to a selected group of code statements. Consequently, some variables are used exclusively in distinctive CDFG blocks. It should be noted here, that the term variable that we use covers program arrays as well. It is convenient to keep array variables in a local memory block. When a CDFG block or its part become a pipeline stage, the corresponding pipeline processor uses local memory to keep local data. Thus, we have a pipeline of processors that execute separate CDFG blocks in our model. The pipeline blocks feature the local memories if necessary, and that memories are implemented as FPGA BRAMs.

If a designer or a hardware compiler discovers the locality of the algorithm's variables, he or she can benefit from the FPGA architecture hugely. However, like the CPU cache, BRAMs must be preloaded with data from an external memory first, so data reuse is also necessary to benefit at all. Figure 2.13 shows the example of the pipeline stage processor architecture. The BRAM enhances the processor to store local data. In the presented solution, the memory variables must be loaded to the register first to perform calculations.

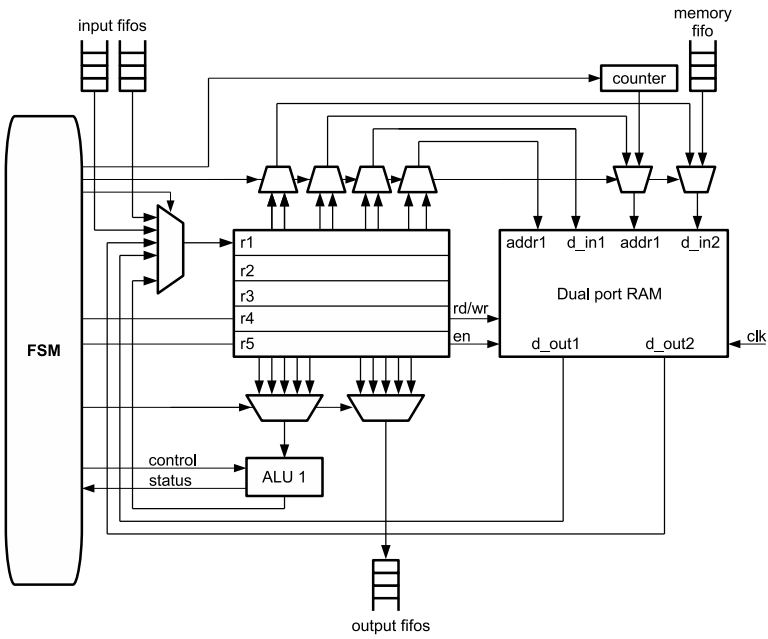


Figure 2.13. An FPGA processor with local BRAM memory

Two independent memory operations are possible thanks to the two separate read/write ports of BRAM. It is useful in many applications and provides an efficient mechanism that allows it to preload the local memory with user's data for example. For this purpose, the separate FIFO stream ('memory fifo') is implemented in Figure 2.13. The processor can quickly load successive data thanks to the automatic address increment that is enabled by the additional counter. Preloaded data are used during the principal algorithm execution. The local memory is useful not only to store constant arrays that are required by an algorithm but also to provide explicit caching of data that otherwise would have to be repeatedly sent through the input interface. The explicit caching scenario assumes memory data replacement more than once during a program execution.

Please, also refer to the architecture of the binary tree processor in Section 3.6.1 for an example application of local memories.

2.5.2. Explicit data caching

The mechanism of data caching is a standard enhancement of CPUs for faster execution of programs. Data is cached automatically by GPPs, as caching is not programmed by a software engineer. Cache automation is very convenient, and it shortens the programming process. Caching relies on two assumptions. The first is that the data already used will be used again soon. The second is that if an array's element 'i' is used, then an element '$i + 1$' will be used as well. The first assumption is called a data reuse policy. The benefits of reuse are evident because it keeps data locally in a low-latency memory to make access faster. The consequence of the second rule is that the cache acts as a data buffer for an external memory. Buffering is beneficial because the external DRAM features a decent throughput but introduces high access latency. That is the reason memory controllers access data in chunks (lines) to carry them between a cache and DRAM. Reading and writing data in portions of adjacent memory addresses is favourable because latency applies to every new memory transaction.

Preferably, custom processors should also exploit cache policies, and they should implement both data reuse and buffering. The only problem is that caching must be planned by hardware designer manually in that case. Unlike CPU programmers, designers of custom processors organize the strategy of data caching explicitly. Although the explicit planning of buffering and reuse is tedious, it is usually more effective than implicit cache work.

Note, that implicit caching is also available in FPGAs, as a cache implementation requires logic and memory resources that are available in FPGAs. The implicit cache is limited in its size, and more complicated than the explicit cache, but it is feasible in FPGAs. An FPGA implementation of the implicit cache consumes ad-

ditional hardware resources and works slower than the dedicated CPU cache. It is necessary to use slow reconfigurable interconnects when BRAMs are combined into large memory components like a cache. The final performance is affected because, large memories require many data and address lines and use a lot of routing resources in FPGAs. Additionally, the logic of an implicit cache is complicated.

More straightforward buffering solutions than implicit caching are favourable for FPGA custom processors. The storing of auxiliary data in an FIFO-style cache is a very efficient technique in searching and browsing applications for example. The FIFO-style cache uses queues to keep data locally. Every method that involves FIFOs is very convenient for FPGA designs because FIFOs are FPGA components that are hard-wired. An FIFO mode is one of the BRAM's operation mode, and a BRAM is configured to be an FIFO at a designer's request.

The most advantageous feature of an FIFO-style cache is that an FIFO does not require lines for an address bus. The lack of memory addressing simplifies both wirings of a hardware and operation of a system. On the other hand, the use of queues requires the careful organization of data because FIFOs can be accessed on a one-by-one manner only. Figure 2.14 presents how the concept of the FIFO-style cache works for the custom architecture. The processor can access only the first element from the FIFO queue and once the operation is done the item is not available anymore.

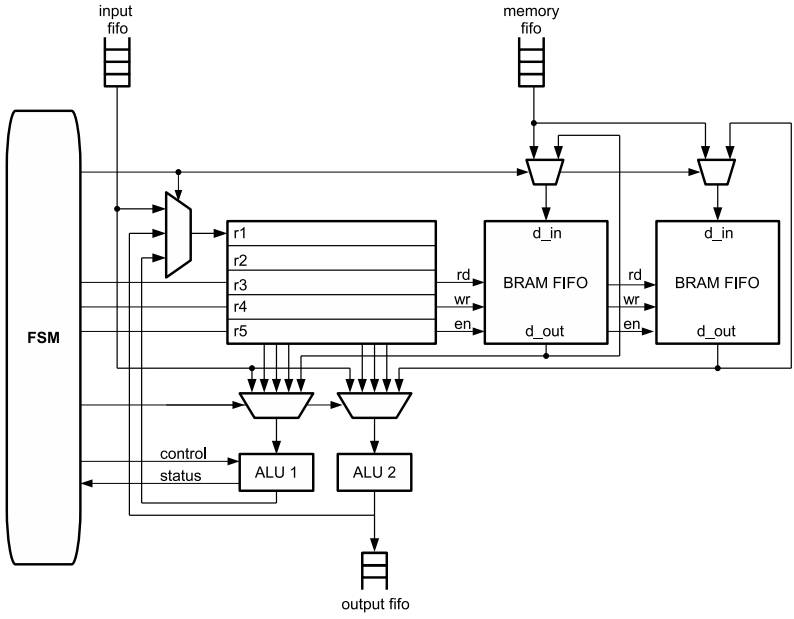


Figure 2.14. An FPGA processor with local FIFO storages

The next advantage of FPGA FIFOs is that they are synchronous digital elements and offer registered outputs. Consequently, the variables do not need to be transferred to the registers before functional units can process them. Figure 2.14 provides the design that offers processing of data that is latched by FIFO's output. Data elements are sequentially sent from the FIFO to an ALU directly, and the ALU stores its result in a selected flip-flop register. However, there are cases that data reuse is requested. A data loop-back path that links an FIFO's output with its input is necessary then. The data loop-back is presented in the Figure 2.14. Data that has been already processed is put back to the FIFO queue and waits for the next data processing cycle.

Thanks to the existence of the two ALUs and two FIFOs, powerful parallel processing is possible in the design that is presented in Figure 2.14. Each ALU can process data that is directly read from the FIFO. The two-fold architecture that is presented in the figure is just a presentation of the forceful concept that can be scaled up to process in parallel many local data streams that are stored in the independent FIFO caches.

In practice, it is not a rare scenario that a processor has to combine an input data stream with multiple independent local memory streams, and every such a couple produces a separate result. This scheme fits matrix-vector product, for example, where the vector is an input stream, and matrix rows are kept locally. The overall algorithm that explains that plan is given in Listing 2.2.

Listing 2.2. An example of the algorithm that process an input stream and many local streams simultaneously

```
Algorithm(){
  input A;
  array b1, b2, b3;
  output res1, res2, res3;
    for( i=1; i<N; i++){
      a = A[i];
      res1+=function1 (a, b1[i], res1);
      res2+=function2 (a, b2[i], res2);
      res3+=function3 (a, b3[i], res3);
  }
}
```

Consecutive elements of input 'A' are used for independent calculations: 'function1', 'function2', ... Each calculation is separated in a sense that it uses a distinct function and a different 'b*n*' array. The arrays of 'b*n*' elements are stored in the local FIFO caches. Importantly, sequential access to elements of 'b*n*' is sufficient to perform computations. Functions 'function1', 'function2', ... can perform the same or different calculations.

The architecture in Figure 2.14 also presents another interesting feature that is common to data processing in FPGA devices. It enables immediate data processing from an input port. Previously, the presented architectures required that data be stored in a register before any further processing is possible. Here, we have reduced a processor's latency thanks to the execution of input element that is taken directly from the input port.

2.5.3. Sequential-Access Buffering

We have introduced an FIFO as a cache supplement that provides a mechanism for data reuse and buffering of an external memory. Another buffering technique will be presented now, and it exploits, just like a CPU cache, a data locality property. The advantage of that method is that it allows a custom processor to access the data arrays efficiently even if they are located in the external DRAM. It possible to work with data sets that do not fit the FPGA's BRAMs size. Here, in this paper we call the method Sequential-Access Buffering.

Similarly to a processor, an FPGA device needs a memory controller to connect to the external DRAM memory. The FPGA's memory controller can be hard-wired inside a reconfigurable structure, or it can be implemented as the IPCore that uses reconfigurable resources. The hard-wired version consumes less transistors, but it wastes silicon area if it is not used. Despite its architecture, just like a CPU's memory controller, FPGA's controller suffers the problem of the high data access latency. It can read efficiently data in large chunks, but accessing single memory elements is time-consuming.

The Sequential-Access Buffers (SABs) are the solution that mitigate the problem of the DRAM's latency. They help to access many individual, unbroken regions of external memory faster. Each region has to be accessed sequentially, but accesses to different regions can interleave in random order. In the other words, a processor can skip randomly between regions but the region elements must be scanned one element after another.

A designer declares areas of external memory that an FPGA has to access in the SAB method. The complete system of the Sequential-Access Buffers is presented in Figure 2.15. The FIFO buffers go between a custom processor and the SAB controller. The processor accesses the next region element by an appropriate FIFO operation. When, during a read operation, the FIFO becomes almost empty, the SAB controller reads a new chunk of data from external memory. As we know, accessing external memory data in bulk spares unnecessary latency. Data is ready in the corresponding FIFO in advance, and the processor meets no wait-states. A similar scenario is valid for write operations, but the controller initiates write of a chunk of data when the FIFO is almost full.

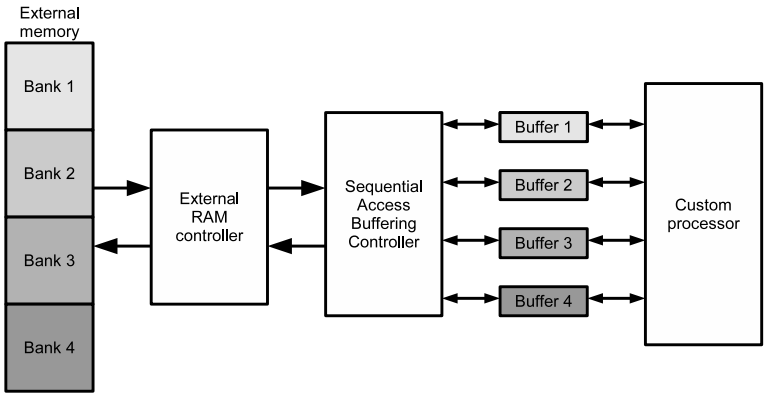


Figure 2.15. A system of the four Sequential-Access Buffers

References

Local BRAM memories are used in the architecture presented by Russek and Wiatr [59]. The work describes the solution for regular expressions matching systems. The solution presents the new approach to the problem. The authors introduce the original multi-stage algorithm for incremental data screening. The algorithm is intended for the reconfigurable SoC devices as it gains from the execution of its initial stage in the custom coprocessor. The paper gives the architecture of the pipelined custom processor that uses local memory blocks of data to store dictionary patterns.

Karwatowski *et al.* [98] presented the architecture for document similarity calculations. Searching a large database requires repeated execution of the same operations for the various data, so the number of operations can be easily parallelized to reduce the time required for the processing. The text vectors from the database are sent through the dedicated communication pipe to FPGA and compared online to many reference vectors simultaneously. The design uses FIFO cache to calculate cosine similarity for eight documents in parallel.

2.6. The FPGA-oriented algorithms

The essential goal of the discussion that is provided in this paper is to highlight potential benefits of custom processor technology for data-intensive computing. The particular focus is put on reconfigurable devices because they play a practical role in general purpose computing as a part of reconfigurable accelerator cards that are available on the market today.

We mentioned in Section 2.1.1 that the careful algorithm selection is the most important part of the custom processor design. Algorithms that work successfully

on general purpose CPUs rarely suit implementation as custom hardware. The usual approach to finding a suitable architecture for the custom processor is to start with the original definition of the problem and then try to devise a solution that exploits strengths of custom processing. The designer's knowledge of the hardware's characteristics allows one to recognize correctly the distinctive attributes of the algorithm that suits hardware implementation.

We will enumerate eight unique algorithm's features that warrant to benefit from the migration of the processing from CPU to FPGA.

1. The algorithm constitutes the outer loop. Thanks to the outer loop, the processor executes the same program code many times. This perpetual operation leads to better hardware utilization and enables the pipelining method.

2. The algorithm exposes concurrency, but the introduction of the parallelism does not lead to data starvation.

3. The maximum number of alive variables is limited. A large number of variables disturb pipelining because all active arguments have to be shifted along from one stage to another. Moving variables along a pipe consumes a lot of register resources.

4. The control-intensive algorithms (many ifs) are troublesome. The alternative execution branches that exist in condition statements introduce underutilization of resources because only one branch is chosen at a time. Also, 'if' and 'switch' constructs limit pipelining.

5. The dynamic-range inner loops do not exist. Static range loops are easier to enhance because unrolling is possible. Dynamic-range loops must be executed sequentially.

6. The algorithms access data in sequence. It is possible to organize the order of input data to allow the processor to access the memory consecutively. Accessing memory data in chunks hugely improves the average data throughput. Algorithms that process data that is organized in streams offer the best performance.

7. Fix-point arithmetic is preferred. Functional elements consume fewer resources for fix-point and integer arguments. Consequently, more processing elements fit the FPGA device.

8. The algorithms perform bit-wise and logic operations. Basically, CPUs implement arithmetic and logic operations only. Peculiar bit manipulations are faster in custom hardware.

Over years, research in algorithms and computing methods has been focused on general-purpose processors mainly. Methods for custom hardware have been proposed only recently. These methods usually derive from the same stem as their CPU counterparts, but they have developed independently. The reason is that a software designer focuses on finding the best algorithm for fixed processor architecture whereas a hardware engineer looks for the hardware architecture, which is the best problem solution. However, designing of the custom processors must involve the algorithm considerations as well.

Fortunately, software concepts that directly fit hardware design exist. Algorithms, data structures, and software methods, together with their variations that are suitable for hardware implementation, in an area of browsing and searching will be presented, in the next chapter.

References

Mueller *et al.* [99] study how data processing can be accelerated using an FPGA. The results indicate that efficient usage of FPGAs involves the right computation model, the careful implementation that balances all the design constraints, and the proper integration strategy to link the FPGA to the rest of the system.

3. Data-intensive algorithms for FPGAs

3.1. Sorting and searching

Sorting and searching are principal operations for database systems. A variety of data organization schemes for the database systems exist. There are relational, NoSQL, centralized, distributed, to mention only a few types that are in common use. Nonetheless, despite the high-level database organization, sorting and searching remain the kernel operations for every database. We have already stated in Section 1.6 that the speed of data-intensive activities is not CPU-bound, but an IO-bound parameter. Therefore, the expected speedup of FPGA-based acceleration of sorting and searching is limited. However, FPGA devices are still beneficial for the improvement of energy efficiency. It takes effect because data movement optimization, resource usage optimization, and system clock reduction is possible when custom processors are employed.

A CPU reads data from its operation memory. If the data is not available in the main memory, it must be transferred to RAM from the secondary storage which is usually a disk drive. The IO subsystem performs the transfer from the mass storage device to the main memory. The Direct Memory Access (DMA) mechanism is employed to transfer data to the memory for the best performance. The DMA engine usually transfers data in blocks; therefore, it can copy information much more efficiently than a CPU. When data is already available in RAM, a CPU moves data, forth and back, to its registers to complete calculations. Additionally, data manipulation that is performed by a CPU always involves data caching, whenever it helps to improve the processing speed or not. When processing is finished, the results are transferred back to the local or remote disk storage by the DMA. We can recognize a lot of data movement that takes part in CPU-based computations in the above description. Therefore, we might prefer to use the FPGA accelerator that is capable of processing IO data directly and consequently more energy efficient.

We have already stated in Section 1.9 that two major modes of data processing are available for IO FPGA accelerator. The first, more common way, is to use an

accelerator to processes data which is already available in the main memory. For processing, data is transferred from the main memory to the accelerator in this mode. The second mode of data processing uses an accelerator to conduct data processing on-the-fly, during its transfer from an IO device (*e.g.* Ethernet card or discs storage), via FPGA to the main memory. The first mode is referred as a central mode, and the second mode is an IO mode.

The work of an accelerator in the central mode resembles the CPU's operations, but it usually minimizes data movement. First, data is transferred from the host's memory to the accelerator's memory. Then, it is read, processed, and sent back to the accelerator's memory. Finally, the results are transferred to the host's memory. It is also possible that accelerator's local memory is bypassed. In that scenario, the accelerator processed the data directly while it is transmitted from the host's memory, and the results are immediately sent back to the host. The second method requires no local storage, but it is not suitable for every algorithm. The elimination of unnecessary data movement reduces energy consumption.

CPU-style implicit caching is not necessary if data reuse is a rare, as in the case of simple searching, for example. FPGA can dismiss the cache and start to exploit the phenomenon of data locality by the introduction of the simple data buffering method (see Section 2.5). Further, the accelerator can process data as a stream if an algorithm is carefully devised and data is well-organized. The stream processing minimizes unnecessary copying of data; therefore, custom architectures that attain stream processing reduce energy consumption. It is worth noting that an accelerator's IO mode of operation while data is read from a mass storage device, for example, reduces data copying even more. Unfortunately, that kind of processing is troublesome due to the lack of support of contemporary OS.

Under those circumstances, porting of some sorting and searching activities to the customized hardware is worth a try. Interestingly, several well-known software algorithms have their hardware counterparts. Such algorithms are ready to act as custom processors immediately without an additional reworking. Among such algorithms, the sorting nets and merge tree are known for sorting, for example. Accordingly, the Bloom filter, binary tree, and prefix tree are ready to perform searching as custom processors. Those architectures will be discussed in the following sections of this paper.

3.2. The sorting nets

The sorting networks, also called sorting nets, are a concept that was developed to provide deterministic sorting algorithms. A characteristic of the sorting nets algorithms is that they define a fixed sequence of ordering operations which can be

presented as a structure of wires and ordering elements. The sorting networks originate from the software theory, but they provide a perfect definition for sorting architectures in hardware. They present a perfect example of software algorithms that directly define hardware implementations and explicitly characterize the design of custom processors. The sorting nets inherently provide information of parallelism and pipelining. In the previous chapter of this work, we presented how to derive custom processors from the algorithms' sequential codes but those procedures are not necessary for sorting nets. It is exceptional that custom architecture can be taken directly from the algorithm's native definition. The approaches that skip intermediate problem representation and lead naturally to hardware solution provide better final results.

The sorting network consists of many ordering elements that are connected by wires. The ordering element performs the ordering operation. Figure 3.1a presents the symbol for the ordering element. The input arguments that are given to the inputs of the ordering element are compared and swapped if necessary. An ordering element sends a smaller argument to its upper output and a bigger argument value to its lower output. It is convenient to recognize that an ordering element is itself a two-input sorting network.

The structure of the sorting network defines which operations can be performed in parallel, and which has to be run in sequence. Obviously, all operations that are defined by the network can be carried in a sequence. However, the main reason to implement a sorting network algorithm is its inherent parallelism because faster than sorting nets sequential sorting algorithms exist. That feature explains the appreciation of sorting networks for sorting in GPGPUs. Unfortunately, a designer has to provide all the sorting elements at once to complete the sorting in a minimum number of steps; which is equivalent to the exploitation of the maximum possible parallelism. It is unique in a computing system that all necessary input elements are available at the same time for the big sorting net. Thus, the sorting net algorithm is typically executed in a mixed, parallel-and-serial manner. The available architectures of an FPGA accelerators offer limited IO throughput and inhibit the direct use of a fully parallel sorting network. Therefore, in practice, the algorithm is more useful as a part of SoC where an internal source of input data exists and input sequence length is moderate.

Various schemes are known to construct sorting networks. Figure 3.1b presents the method for iterative construction of the bitonic sorting net [100]. We present an N-element bitonic sorter structure that consist of four $N/2$-sorters and one N-merger. The name 'bitonic' comes from the definition of bitonic sequence that is not ascending first and then not descending. In the other words, the bitonic sequence is created by a juxtaposition of increasing and decreasing sequences. In Figure 3.1b, we can recognize the concatenation of the two sorted sequences at the merger inputs. It can

be seen as the rearranged bitonic sequence where the second sequence has to be reversed to get the true bitonic string. Figure 3.1c shows the structure of a merger that regards the rearrangements of the input shuffled bitonic sequence. The merger operation has a property that when a bitonic sequence is sent to its inputs, each element in the first half of the output sequence is not bigger than any element in the second half. Thus, an additional pair of output sorters is necessary to compose the correct result of the N-element sorter (see Figure 3.1b).

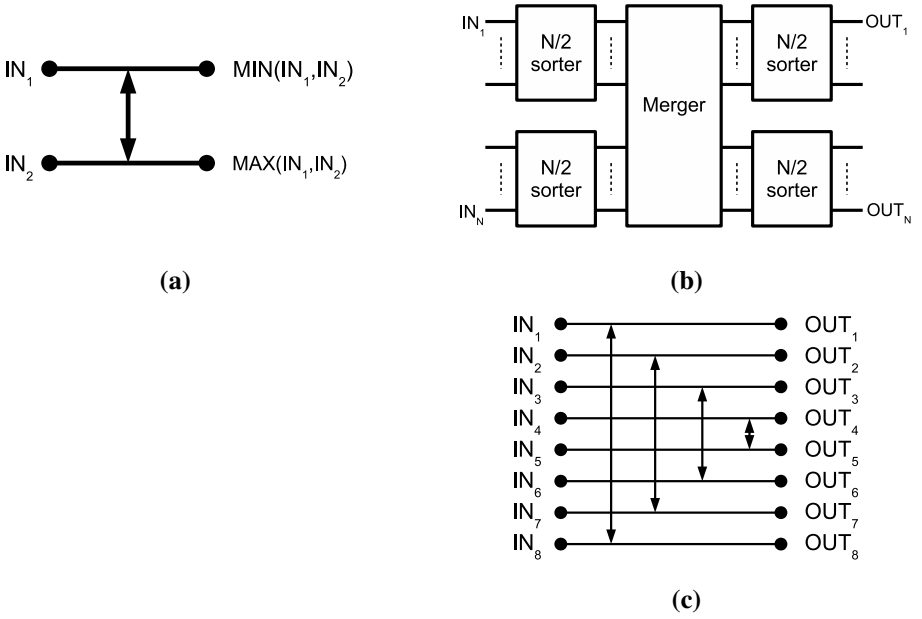


Figure 3.1. The architecture of bitonic sorting nets: a) the ordering element; b) the overall architecture; c) the merger construction

Russek and Wiatr [101] gave method and example results of the construction of bitonic sorting networks in FPGAs. Their work regarded implementations of the bitonic sorting nets in Xilinx's Virtex XCV1000 device. The authors had developed a software tool that generated a VHDL description of a sorting network for a given parameter. The tool read a desired quantity and bit-width of inputs and outputted an RTL description. The sorting network that was generated considered input arguments to be integer numbers. The paper provided the necessary number of ordering elements for a different size of the network (Tab. 3.1). The number grows exponentially with the size of the network, so the networks that fit FPGA had a very limited span. The paper presented implementation results to support that thesis. In conclusions, the article suggested the use of an FPGA accelerator as a pre-processing step of sorting.

In that scheme, the sorted sequence was split into shorter subsequences first and, then the hardware accelerator sorted the subsequences. Afterward, the merge sort algorithm was applied on the CPU.

Table 3.1. The growth of resource requirements for the bitonic net

Number of inputs	4	8	16	32	64
Number of ordering elements	6	24	80	240	672

The work of Russek and Wiatr also delivered resource utilization of the pipelined version of the nets. As expected, the introduction of pipeline registers did not influence utilization of resources. It happened because every FPGA's logic element has an assigned flip-flop that is wasted if it is not used. The article also offered an idea of inputs serialization. Serialized FPGA inputs accepted one bit per clock cycle, and arguments were shifted bit by bit. The serialized solution required significantly fewer input wires, but it degraded the input throughput and consumed more flip-flops.

References

The reader can find comprehensive details about the theory and architectures of sorting networks in [102]. Batcher describes networks that have a fast ordering capability in [103]. Batcher's sorting network could order 2^p words in $\frac{1}{2} * p * (p+1)$ steps. Ajtai *et al.* [104] give a sorting network with $c*n*\log(n)$ comparisons, where n is a number of elements and c is some constant. The algorithm can be performed in $c * \log(n)$ parallel steps as well, where in a parallel step we compare $n/2$ disjoint pairs. Thus, authors propose a sorting algorithm working in $O(\log(n))$ parallel steps, but the constant value that is hidden in O notation is huge.

Zhang and Zheng [105] shows the parallel sorting architecture, denoted as MMSORT (min-max sort). The proposed architecture took advantage of a sorting network and the distributed processors coupled with the local memory. A small sorting network can be used iteratively to sort a large number of elements. The major upside of the idea is that it can be executed in a pipeline if only some conditions are fulfilled. There is no need to wait for the results of the previous iteration to start the next one. However, a distributed memory of a large size is necessary to this solution. It holds because the complete input dataset must fit simultaneously into the sorting processor's memory. The authors didn't deploy their proposal in practice, so no implementation results are known.

Huang *et al.* [106] discuss hardware design approach to Batcher's sorting network. A list of elements is partitioned first into shorter lists. Batcher's network is

used to sort those lists. The shorter lists are then merged into the final complete list by the merge sort algorithm. The merge sort is performed sequentially on the elements that are stored in the local memory.

Greß and Zachmann [107] present a novel approach for parallel sorting on stream architectures. It is based on adaptive bitonic sorting. The approach achieves the complexity $O(n \log(n))/p)$ for sorting n values utilizing p stream processor units.

Mueller *et al.* [108] focus on the problem how an FPGA can be exploited when they are used for sorting network implementation. The paper provides a set of guidelines how to make design choices. Two application scenarios that are provided are based on sorting networks. The first is a median operator; the second is an HW/SW Co-Design case. Authors' experiments showed that FPGAs are a useful component of modern data processing.

Peters *et al.* [109] present a high-performance, in-place implementation of Batcher's bitonic sorting networks for CUDA-enabled GPUs. The authors adapted bitonic sort for arbitrary input length and assigned operations to threads in a way that decreases low-performance global memory access.

3.3. The merge sort tree

A merge sort algorithm creates a single sorted list out of elements of two (or more) sorted lists [102]. At each step, the algorithm compares the head elements of the input lists and moves the smallest one to the output list. It is also possible to sort an unordered set of elements, but the merge sort has to be applied several times iteratively. In that case, the procedure starts with merge sort of one-element lists. First, it joins elements into couples; next, couples are linked into fours, fours into eights, and so on. Joining sequences of equal length minimizes the computational complexity of the algorithm, but merging sequences of unequal length is also possible and inevitable in practice if the size of the sorted set is not a power of two.

The merge sort is the plainest algorithm that is used for sorting. It is deterministic and sorts N input elements in $N * \log(N)$ steps. In practice, it allows the build sort solutions of the highest performance. The merge sort requires more algorithm steps than a quicksort or heapsort algorithms, for example, but it is popular because it outperforms other algorithms in real-life applications. The speed of the merge sort algorithm comes from its regularity and sequential data access.

The merge sort can be combined with the quicksort or heapsort, to enhance sorting. In that method, a big set of elements is split into many smaller sets. The small sets are sorted independently by the quicksort or heapsort, and then they are merged into the sorted output. The drawback of a merge sort algorithm is its memory requirement because it needs additional space to store all elements of the output sequence.

The simplicity makes the merge sort to be a perfect candidate for implementation as the custom processor. Figure 3.2a shows the proposed architecture of the merge sort hardware. The inputs and output are buffered in FIFOs in order to make data access easy. The system is synchronous, but the clock is not present in the figure. The system picks the smaller value and transfers it from the input FIFOs to the output FIFO at each clock cycle. The read signal 'rd' activates a read operation of the selected FIFO. The output FIFO acknowledges it can accept next element by asserting its ready signal 'rdy'.

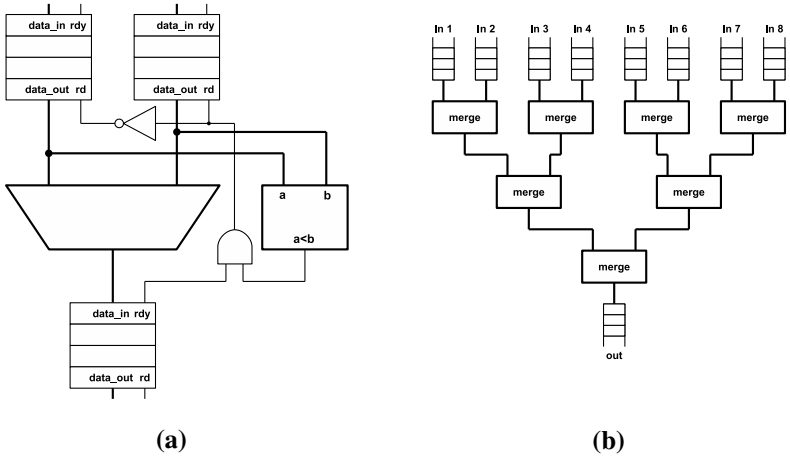


Figure 3.2. The construction of a merge sort processor: a) the basic merge element; b) an architecture of the merge tree

Using the structure that is given in Figure 3.2a as a building block, one can construct the bigger tree. The multi-input merge sort tree that is presented in Figure 3.2b merges many lists simultaneously. It is not necessary to add FIFO buffering to the internal tree nodes. The tree can work correctly without those FIFOs if the input ready 'rdy' signal is short-cut with the output read 'rd' signal of each removed FIFO. The removal of internal FIFOs leads to a combinatorial logic. The combinatorial version of the merge tree contains FIFOs only on inputs and output. However, a lack of register buffering may lead to a long combinatorial path and cause degradation of the maximum clock frequency. Effectively, the two groups of combinatorial paths exist in the combinatorial version of the tree. The first group consists of the paths that start with the output 'rdy' signal and ends at the input 'rd' signals. The second group consists of the paths that start with the input 'data_in' signals and stops at the output 'data_out' signal. If many levels of the tree exist, the combinatorial path latency is significant; therefore, the tree should be pipelined *i.e.* registers should be put at each

tree node. On the other hand, if the sorted elements have a substantial bit-width, the registers may employ additional resources. However, the problem is not crucial for FPGAs because a quantity of logic and flip-flop resources is balanced. Logic gates (LUTs) that are used to create multiplexers have associated registers that are available to pipeline the tree.

The merge sort tree is a perfect example of an algorithm that gains when it is combined with Sequantial-Access Buffering that is presented in Section 2.5.3. Each input list are emptied in a sequential order, one element after another, but the merge sort processor reads its input ports in a random sequence. It is beneficial to use the Sequantial-Access Buffers if input lists are kept in separate but continuous regions of the external memory. In this scenario, the SABs replaced the input FIFOs and perfectly buffer random access to the external RAM.

3.3.1. The FPGA-accelerated sorting system

Russek and Wiatr [110] tested an idea of building a hybrid sorting system. Their paper compares the efficiency of the merge sort custom processor with an efficiency of the merge sort application that was run by a CPU. The authors proposed to break an input data set into smaller groups of elements, and then to use a CPU to sort each group separately. The idea was to make the size of the group to match the size of the available CPU cache. The CPU's work was followed up by the tasks of the merge sort accelerator in FPGA. That scheme had a proper background in an observation that the best results are achieved by hybrid solutions that combine different types of sorting algorithms [47]. The approach also had a support in the paper given by Gray and Nyberg [46], which compared the efficiency of different sorting architectures and scenarios.

Additionally, Russek and Wiatr presented the implementation and its results for the merge sort tree processor in the XC4VLX200 device that is a high-performance Xilinx's FPGA with 200 K logic cells and enlarged memory resources. The achievement was significant because it applied to the HPC system. The heart of the solution was the RASC accelerator. The RASC was the integral component of the SGI's Altix 4700 HPC system. To provide a good throughput, proprietary NUMALink interconnect linked the accelerator with the system. Similarly to the standard accelerator, the RASC offered external memory to provide data exchange between the host and the FPGA device. In the merge sort application, sorted lists were loaded into one memory region and results were read from another memory area.

The paper highlighted the problem of the consumption of FPGA register resources also. The authors proposed to use BRAMs, instead of flip-flops when sorted elements have a meaningful bit-width. Consequently, two different sorting architectures were compared. The paper gave the performance and resources utilization of the

two solutions. Both architectures worked similarly in the terms of performance, but their clock speed was not limited by a delay of the critical path but by RASC maximum available clock frequency. Nonetheless, the BRAM version was able to sort 64 lists of 64-bit elements, and the flip-flop version worked with 32 lists of 32-bit elements only.

The accelerated merge sort algorithm that was presented in the paper outperformed the CPU-only solution by a factor of two only. However, HLL, specifically Mitrion-C [111], was used to create the hardware accelerator. In the authors' opinion, the use of HLL could cause a degradation of the performance with respect to the design that could be prepared using low-level HDL. However, one can notice that the solution did not take advantage of the SAB technique. Consequently, the hardware processor had accessed addresses of an external memory in an inefficient way. The SABs would speed the processing up, but they were not inserted due to the HLL's limitations.

Additionally in their work, Russek and Wiatr assessed the energy requirements of the FPGA and CPU chips. According to Xilinx's Power Estimator tool, the FPGA's power dissipation was 3.1 W only. Due to the lack of proper equipment, the authors did not measure the energy consumption of the 1.5 GHz Itanium2 CPU processor. So, the energy was derived from TDP parameter that was provided by the processor's manufacturer. It was 104 W accordingly. It was obvious that the FPGA and CPU differ an order of magnitude in the power consumption. It should be noted that the performance and energy comparison that were presented in the paper were done with respect to Itanium2 processor that was not a state-of-the-art processor at the time. More advanced CPUs were available, but the Itanium was manufactured in the same 90nm technology as the Virtex4 FPGA device that was used for the experiment. In that sense, the comparison was fair.

References

Koch and Torresen [112] analysed the different hardware sorting architectures in FPGAs. It was proven that a combination of an FIFO-based merge sorter and a tree-based merge sorter resulted in the best performance at low cost.

Harkins *et al.* [113] compared the execution speed of hardware modules which were implemented in FPGA with the speed of the functions executed by the microprocessor. Algorithms for sorting: quicksort, heapsort, radix sort, bitonic sort, and odd/even merge sort were tested. Results showed that the merge sort performed the best, both for the CPU and the FPGA.

Marcelino *et al.* [114] presented and evaluated three hardware sorting units for embedded computing systems implemented in FPGAs. They compared Batcher's odd-even sorting network, insertion sorting, and FIFO-based merge sorting. The

study showed that the best is a hybrid of an insertion sorting and an FIFO-based merge sorting. The sorting speedup was between 1.6 and 15-fold in comparison to a pure software solution of the quicksort.

3.4. The Bloom filter

The Bloom filter algorithm belongs to the class of search algorithms. The method was first proposed by B. H. Bloom [115]. The algorithm analyses items those are processed as b-bit long messages. Bloom proposed to use a memory array to verify the existence or non-existence of a given message in a pre-defined set.

The Bloom filter operation consists of two steps:

1. The deterministic generation of a pseudorandom number for the message. That number is called hash.
2. The lookup in the memory at the address given by the hash. If the memory cell is empty, the message does not belong to the set.

The hashing converts a b-bit input message into an h-bit output hash, where h is smaller than b. Different types of the hash function can be used for the Bloom filter algorithm. One can adapt a checksum algorithm that returns the remainder of a polynomial division for example [116]. The generation of a hash is a one-way operation because different messages end up with the same hash value ($h < b$). Consequently, a message can produce a non-empty memory cell despite it does not exist in a reference set. Thus, the Bloom filter returns, so called, false-positives; therefore, it is not entirely trustworthy. On the other hand, false-negatives do not occur as the algorithm correctly rejects messages that are outer to the set. A probability of a false-positive can be calculated for a given dimension of the set of patterns and the h value.

One can minimize the likelihood of the false-positive when the two-step procedure, described above, is repeatedly applied for different hashing functions. In this multi-hash scenario, the message is considered the member of a set if every hash round returns a non-empty memory cell. The algorithm rounds share the single memory array for different hash functions in the classic Bloom filter. Obviously, the memory content must be prepared beforehand to put the Bloom filter into work. Hash values are calculated for all hashing functions and all messages in a reference set in the preparation phase. Then corresponding locations in the memory are marked.

The probability of a false-positive in the sequential Bloom filter is given by formula

$$p_{\text{err}} = \approx [1 - e^{-\frac{kn}{m}}]^k,$$

where n is a number of searched patterns; m is a bit size of memory and k is a number of hash functions (the algorithm's rounds).

It is possible to select the optimal value of k, for a given m and n, to minimize the hazard of the false-positive. The optimal value of k can be calculated by formula

$$k_{\text{opt}} = ln(2\frac{m}{n}),$$

and its corresponding optimal probability by equation

$$p_{\text{opt}} \approx -2^{-k}.$$

Bit-wise manipulations such as logic operations or shift and swap transformations are the basis for many hash functions that are used in practice. Those elementary operations make hardware that is necessary for hashing to be very easy to implement. Similarly, the memory operation of the Bloom filter is a simple, very much favoured by hardware designers, the look-up table operation. Additionally, phases of hashing and look-up can run in the pipeline to generate one result of message matching at each clock cycle. That makes the Bloom filter another algorithmic candidate that naturally suits for a custom processor design. Accordingly, many applications that are reported in the literature, which perform various kind of search operations, take advantage of the Bloom filter. The papers give its example applications in Network Intrusion Detection Systems (NIDS) [117] [118], anti-virus systems [59] or firewall solutions [119]. Web search [120] and database search [121] are other examples of the Bloom filter utilization.

3.4.1. The parallel Bloom filter

The original Bloom filter algorithm which is implemented in software is sequential. Parallel execution of the algorithm is requested to take full advantage of a hardware version of the Bloom filter. To meet this requirement, Jamro *et al.* [122] present multi-fold parallel version of the Bloom filter as a custom processor architecture. Among other enhancements, authors proposed to perform rounds of separate hash functions concurrently. According to their idea, a single memory block of size m has to be substituted by k memory blocks of size $\frac{m}{k}$. In that parallel scheme, each hash function checks up cells in the independent memory block. The transition of the Bloom filter that uses two hashes to its two-way parallel counterpart is given in Figure 3.3. The total memory bit-size of both solutions is equal. The hash length h can be reduced for the parallel version because its memory blocks are smaller. For example, parallel hashes are one bit shorter for $k = 2$, as presented in Figure 3.3b. In their paper, Jamro *et al.* [122] show that for constant m and the small size of the reference set n, the probability of a false-positive for the sequential and the parallel solution is the same.

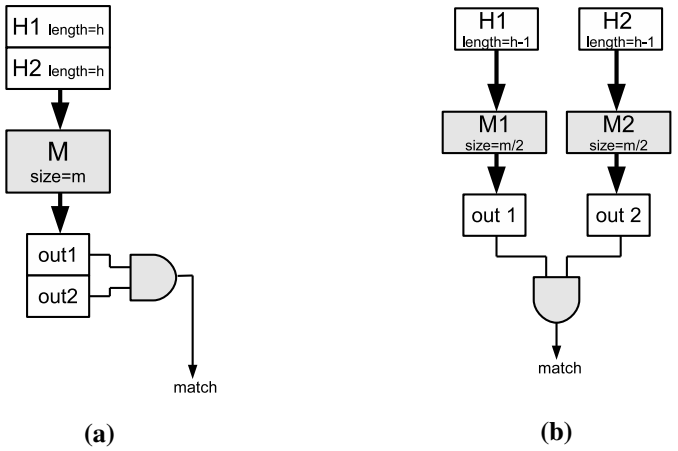


Figure 3.3. Hardware configurations for the Bloom filter: a) sequential configuration; b) parallel configuration

The Bloom filter custom processor is a very compact solution that consumes a very moderate amount of FPGA resources. The implementation results of the Bloom filter that are delivered by Jamro *et al.* [122] and Russek and Wiatr [60] demonstrate that the number of BRAM resources is the only limitation for the filter implementations in FPGA devices. The logic elements and flip-flops are in minimal use. Both works present the designs that take advantage of many parallel Bloom filter structures that are put into the single FPGA architecture. These structures work unison to analyse concurrently overlapped messages that are extracted from a long string of characters. It is necessary to employ more than one matching structure to process data seamlessly with no stalls of the input channel when more than one character of the message arrives in the system at each clock cycle (see also Section 3.5).

References

Dharmapurikar *et al.* [123] considered the implementation of the Bloom filter that fetches one byte of input data at each clock cycle. A parallel Bloom filter operation was performed for the patterns of the different lengths in that work. The throughput achieved in that solution was 100 Mbyte/s which was sufficient for an inspection of data in a Fast Ethernet network.

Suresh *et al.* [118] presented an automatic VHDL code generation for the Bloom filter. A 'C' program was developed for this purpose. The detection of viruses was the main goal of that project. The speed of data processing was 3.1 GBytes/s for the Virtex XC2V8000 FPGA, but only a single word could be localized with this performance.

A completely different solution was presented by Jacob and Gokhale [124]. The goal was to detect the language of the text that was presented to the processor. A set of different Bloom filters was implemented. Each filter was programmed to detect a certain language. Due to IO bandwidth limitation, a theoretical throughput of 1.4 GBytes/s was reduced to 500 Mbyte/s. The eight bytes packet could be read at each clock cycle (the clock frequency was 194 MHz). The data processing performance and the fast programmability of the searched patterns distinguished that solution from the others.

3.4.2. Enhancement of the Bloom filter

The Bloom filter algorithm needs enhancement in applications where the exact recognition of the matching pattern is required or 'false-positives' are not acceptable. Thus, other compression and memory based ideas were proposed. The simplest solution to the Bloom filter's limitations is to follow the Bloom's procedure by a simple comparison algorithm. In that approach, patterns that are thrown out by the filter are looked up in the dictionary. The additional lookup process starts when the Bloom filter reports a membership.

The enhancement that can be introduced is the improvement of the lookup speed for that scheme. One can use hash values that come from the Bloom filter to enhance the verification process by the use of a hash index. The process is presented in Figure 3.4. It uses the hash index table for indirect addressing of patterns in the dictionary. The method is similar to the one utilized in [125]. The hash addresses a position in the index table that contains an address(es) of the pattern(s) in the dictionary. In practice, as hash values overlap, the lookup deals with a list of candidate patterns.

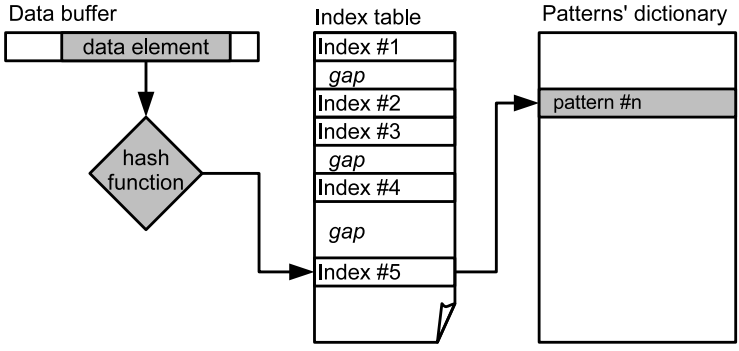


Figure 3.4. A dictionary lookup using the hash index

Additional improvement can be achieved by the reduction of the number of the dictionary items that have to be verified. The verification of the single pattern only would be the best and fastest scenario. The most straightforward method to speed up the dictionary lookup is to introduce the perfect hashing scheme [125]. The perfect hash function delivers one-to-one mapping of the pattern space into the hash space. This feature means that no overlapping hashes exist, and the mapping is not ubiquitous. In consequence, one can use the hash to point directly to the value in the dictionary.

It can be easily noticed that the index table can contain unused locations that are called gaps in Figure 3.4. These gaps can lead to inefficient memory utilization. Therefore, some applications [126] do not implement an index table and use a hash value to address the patterns memory directly. For memory efficiency, this approach requires a very careful hash function and pattern mapping to avoid underutilization of the patterns memory. The best implementation results can be achieved when the minimal perfect hash function is used. The minimal perfect hash function maps m keys to m consecutive integers – usually $[0, .., m-1]$ or $[1, .., m]$. However, construction of a perfect hash function is cumbersome and if possible in practice, applies only to static dictionaries. It is unpractical if the set of elements changes dynamically.

It is worth noting that the perfect hashing contradicts the idea of the Bloom filter, which allows it to reject at once some of the candidate patterns. The hash table is full in the minimal perfect hashing, so there are no 'miss' events. The Bloom filter can detect membership and perfect hashing simplifies exact matching. In other words, the perfect hashing is a fast index generator. The perfect hashing was exploited in the works of Sourdis *et al.* [125] and Papadopoulos *et al.* [126], for example.

Even non-minimal perfect hashing is very difficult to design in applications. Thus, other algorithms are in practical use. One is Cuckoo hashing [127]. Cuckoo hashing relies on k hash functions, and it uses k distinct hash tables. As in the perfect hashing, the hash tables are index tables to the patterns dictionary. The reason to use a k table is to allow for hash function imperfections. When hash values overlap for different hash functions, different index tables are utilized by them. The process of preparing tables looks as follows. When a new pattern 'p1' overlaps with a previous pattern 'p2' in the hash table 1, the 'p2' is moved out to the hash table 2. The position in the table 1 and table 2 are calculated according to hash functions 'h1' and 'h2' respectively. If a collision occurs again in the table 2, the previous occupant is expelled and moved to the table 3. It is possible, however, that the process gets caught in an infinite loop *i.e.* the set-up process fails, and another set of hash functions need to be used for a given dictionary. A lookup requires inspection of a definite number of locations in the dictionary, which takes constant time in the worst case. This is in contrast to many other hash table algorithms, which may not have a constant worst-case time to do a lookup.

The disadvantage of Cuckoo hashing is that during the matching process k index tables and k locations of the patterns must be looked up. There is also a hash size dilemma in the Cuckoo algorithm. The big hash space allows it to map easily the patterns dictionary but cause large memory requirements for the index tables. Although the hash table construction is easier in Cuckoo hashing than in perfect hashing, its memory requirements are higher. Cuckoo hashing was used in the work of Thinh *et al.* [128].

3.4.3. Modified Cuckoo hashing

The author presents his original method of modified Cuckoo hashing in this section. This new method also uses k hash functions, but it requires one hash table only. Furthermore, modified Cuckoo hashing is proposed to eliminate the need for k location lookups that was required in the original Cuckoo scheme. Thanks to the new hash allocation method, it is possible to check only one dictionary location and thus spare the processor's operations.

Like Cuckoo hashing, modified Cuckoo hashing uses the k different hash functions that are necessary to prepare, similarly to the Bloom filter algorithm, a content of a single memory array. In the preparation phase, patterns are hashed, and appropriate locations are marked in the memory. If the new hash conflicts with a previously located hash, the memory location is emptied and marked as forbidden. Conflicting patterns are relocated to the new locations using the next available hash function. The memory's locations that are marked as forbidden must remain empty in the final memory map. Therefore, if the preparation algorithm meets the forbidden location during its work, it has to relocate a pattern to another location using the next available hash function. The procedure stops when the k-th hash function fails or when all patterns are allocated in the memory array. Thus, like Cuckoo hashing, the new scheme may also fail.

An example of memory array design in the modified Cuckoo hashing method is given in Figure 3.5. Patterns 'p1', 'p2', and 'p3' will be mapped to the memory. The example uses three hash functions 'Hash h', Hash g' and 'Hash f'. The hash values are denoted as 'h*i*', 'g*i*' and 'f*i*' respectively. In the example, the memory array offers six positions only. The patterns and their hash values are given in the table of hash values in Figure 3.5. The preparation procedure starts with 'Hash h' function. Hash values 'h1', 'h2', and 'h3' are located in the hash memory in Step 1. We see that 'h2' and 'h3' overlaps in the third cell. Thus, the third memory cell is marked as forbidden and patterns 'p2' and 'p3' are relocated using 'Hash g' function. Next, we see overlapping 'h1' and 'g3' in Step 2, so the second cell is forbidden, and 'p1', 'p3' are relocated. 'P1' is moved using 'Hash g' and 'p3' is moved using 'Hash f'. Luckily, there are no conflicts in Step 3.

Table of hash values

Patterns	**p1**	**p2**	**p3**
Hash h	h1=2	h2=3	h3=3
Hash g	g1=5	g2=4	g3=2
Hash f	f1=4	f2=5	f3=1

Hash table

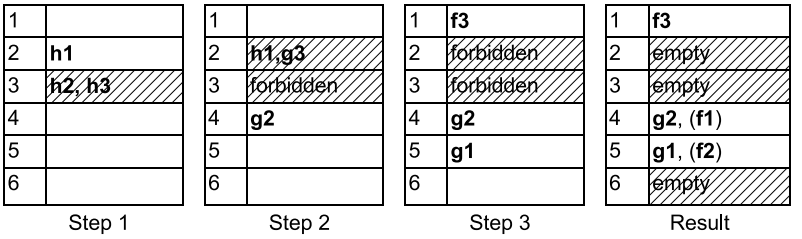


Figure 3.5. The procedure of hash table mapping in modified Cuckoo hashing

The resulting memory has three empty cells and the hash values ‘g1’, ‘g2’ and ‘f3’ are allocated. It is worth noting that ‘f1’ still overlaps ‘g2’, and ‘f2’ overlaps ‘g1’. However, those conflicts are not an obstacle to performing one lookup operation in the matching phase of modified Cuckoo hashing.

In the matching phase, a processor can resolve the matching pattern using only one hash lookup operation. It is possible to meet up to k matches, but there is no need to look up all matching locations. If a match happens for more than one location, one choose the highest priority hash function. The first hash function has the precedence over the rest of the hashes; the second hash function has the precedence over all but first hash function, *etc.* Let’s consider pattern ‘p1’ as an example. If the algorithm finds both ‘g1’ in cell five and ‘f1’ (which is ‘g2’ really) in cell four then ‘Hash g’ has to be look up only. That holds because ‘Hash g’ has a precedence over ‘Hash f’. The ‘Hash f’ function has never been used in the preparation step. Next, a dictionary lookup can be done. Each hash function has a separate index table, so we will get the pattern location from the index table of ‘Hash g’ in our example.

As an example application for modified Cuckoo hashing, the dictionary of 58,109 English was processed to develop the unique 24-bits hashes. The two-way ($k = 2$) modified Cuckoo hashing was adopted to map the dictionary. The CRC32 hash algorithm was chosen. The CRC32 is easy to implement and parallelize in FPGAs. The first hash function hashed bytes of input words while the second function hashed inverted bytes of the words. The hashes were truncated to 24 bits. That procedure allowed it to map of all but eight words from the English dictionary. Modified Cuckoo hashing is an important part of the application that is presented in Chapter 4.

3.5. A shifting substring search

Custom acceleration is most beneficial when the computation needs prevails over the capability of data delivery. Consequently, matching and searching problems are not particularly suitable candidates for processing in FPGAs. However, there are circumstances and additional conditions those make the above statement to become less strict. As it has been already stated, the searching in unstructured data is one coincidence, and the huge volume of searched patterns is another. We will consider the problem of matching of shifting data in this section. Shifting substring searching occurs when analysed tokens are overlapping substrings of a long character string. The number of characters that the processor must acquire to run consecutive matching tasks is reduced in the shifting search scenario. The processor starts by reading all the characters that constitute the first substring and analyses it. Then, it takes the next character from the input, adds it at the end of the previously analysed substring, discards the string's first character and perform the next match. The cycle repeats until the last input character is reached. It may be the case that the relatively complex operations are associated with a single character read operation. One can recognise presented scenario as a sliding window search that is depicted in Figure 3.6. Obviously, the longer window (*i.e.* substring), the more complex matching.

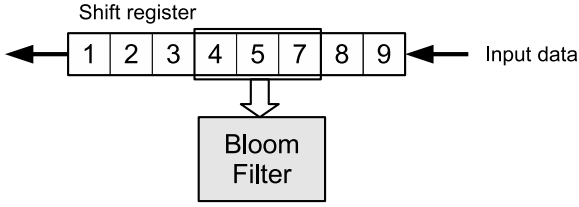


Figure 3.6. A shifting substring search with the Bloom filter

The shifting substring search is a problem for many practical applications, such as a search of anti-virus software or Network Intrusion Detection Systems (NIDS) for example. Occasionally, when word tokenization is not applicable, it can also be useful for text analysis in the Internet, databases or other repositories. Unfortunately, exhaustive processing of an input string, in a character by character manner, is troublesome for software. The Knuth-Morris-Pratt and Boyes-Moor algorithms exist to lessen the number of necessary comparisons. Another fix that is proposed for the shifting search is the Rabin-Karp algorithm. Similarly to the Bloom filter, this algorithm takes advantages of substring hashing.

The Knuth-Morris-Pratt, Boyes-Moor, and Rabin-Karp algorithms do not suit hardware structures due to complex control operations. Thus, the Bloom filter is the best option to search using a shifting window if many patterns exist simultaneously.

The architecture and data flow of the parallel Bloom filter have the capability to process one input message at each clock cycle. It allows the processing of the input data stream without any stalls. The window that is presented in Figure 3.6 moves in pace with input data.

The input data stream bit-width exceeds the size of a single character size in some situations. Today, 32-bit or 64-bit bus interconnects are a standard and the input/output can deliver four or eight characters at a time. In that case, custom processor architecture can handle smooth data processing if four or eight parallel Bloom filters work together concurrently.

Russek and Wiatr [60] exploited the concept of parallel execution of the Bloom filter wider. The custom processor for the Bloom filtering was created in reconfigurable hardware. It is implemented as dual-port synchronous RAM memory. Hashes are sent directly to the BRAM's address bus. The size of the bus fits the bit-width of hashes. To achieve the highest throughput, the execution of the incoming data is performed both in a parallel and pipelined manner. The input buffer is analysed with one-byte resolution. To fit that requirement within the 32-bit architecture of the host system, the Bloom processors consist of four parallel block modules. The modules inspect the input data with a different byte offset (ranging from 0 to 3). The concept is presented in Figure 3.7.

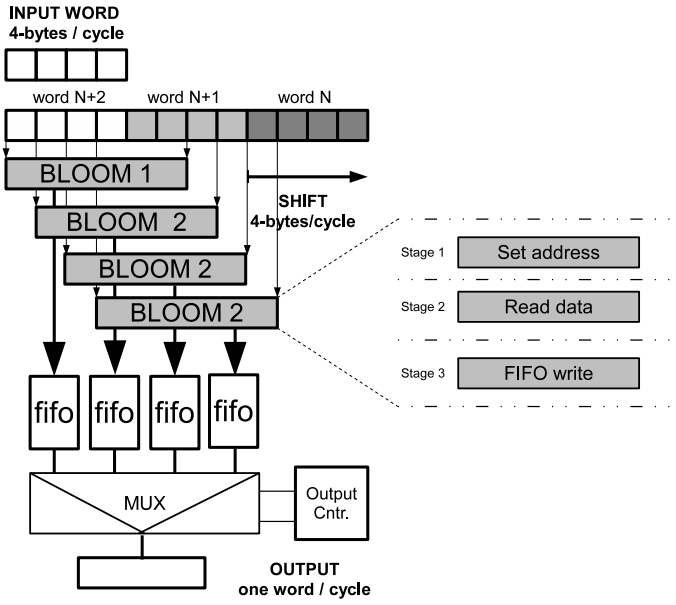


Figure 3.7. A four-way shifting substring search in 32-bit architecture

3.6. The binary tree

The binary search that is also named half-interval search is presented in Knuth's book [102] for example. The algorithm is used to find a match for a random data element in a set of reference patterns. An architecture of the binary tree is the hardware counterpart of the half-interval search algorithm, and it is introduced by Cormen *et al.* [100].

The binary tree is depicted in Figure 3.8. It can be seen as a tree of the consecutive decisions. The sorted dictionary patterns have to be properly located in the tree's nodes. We assume that a set of reference data counts $2L - 1$ elements, where L is a natural value. The node denoted as n_i corresponds to i-th pattern from the sorted set. The input element propagates down from the tree's stub. At each node, it is compared against the corresponding reference patterns and continues left or right according to the comparison result.

The tree is organized into levels, and there are L levels that are enumerated from 0 to $L - 1$ in Figure 3.8. The tree processing can be easily pipelined, and L data elements can be processed by the tree concurrently. A single input data element is processed by the each tree level at every algorithm step. It is possible to feed the tree serially by the elements of an input data stream. One element enters and one element leaves the tree at each algorithm step. The processing latency is L. The 'hit/miss' flag and the index of the matching pattern are the results of tree processing. The result is denoted as 'HIT info' in Figure 3.8.

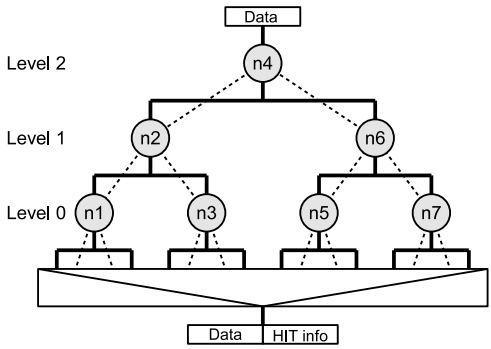


Figure 3.8. The operation concept of a binary tree processor

Figure 3.9 presents the structure of the tree node. The operation of the node is described in pseudo-code given in Listing 3.1. The 'Data' and 'Pattern' elements, depicted in the figure are registers; the rest of the hardware requires a comparator and minor logic. The 'H' field contains the 'hit/miss' flag and a matching leaf index. The 'H' and 'Data' elements propagate together down the structure of the tree.

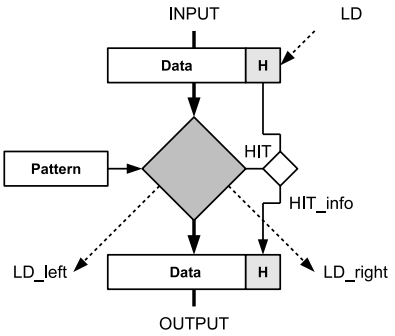


Figure 3.9. The structure of a binary tree node

Listing 3.1. The example algorithm for the single global stream and many local streams.

```
if(LD == true)
  Data:=INPUT;
  if(Data <= Pattern)
    LD_left:=true;
  end if;
  if(Data > Pattern)
    LD_right:=true;
  end if;
  if(Data == Pattern)
    HIT:=true;
  end if;
end if;
```

3.6.1. The binary-tree processor

By studying the pipeline execution concept described in the previous section, one can easily notice that just single node is busy at each level of the tree in each processing step. It can be hardly accepted when hardware efficiency is the priority. If only one node per a tree level is active, the rest of the hardware is idle, and it causes resource underutilization.

The hardware synthesis technique, known as register ports sharing can be used to optimize the tree [76]. That optimization gathers all registers that belong to the same tree level and puts them into a single memory block. Thanks to that, the hardware of the nodes that belong to the same tree level can be reduced to the single processing unit. The register ports sharing allows nodes that belong to a single level of the tree to share hardware. Additionally, this approach gives the opportunity to utilize the

FPGA's memory resources and to free flip-flops for the other algorithm's needs. In practice, the use of BRAMs allows a designer to fit bigger trees into the FPGA device. The architecture that comes from the idea is presented in Figure 3.10. It consists of four processing stages that correspond to the four tree levels. The indexes of nodes that are assigned to the RAMs and presented in the figure are taken from Figure 3.11.

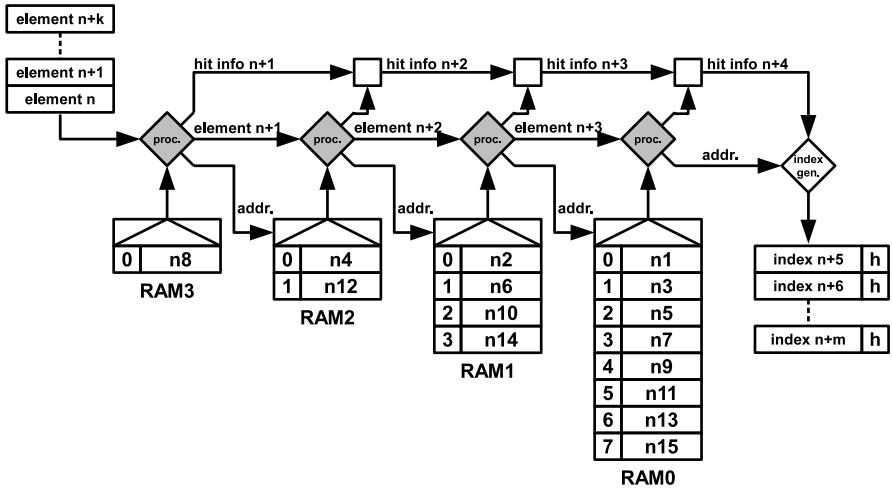


Figure 3.10. The pipeline architecture of a binary tree custom processor

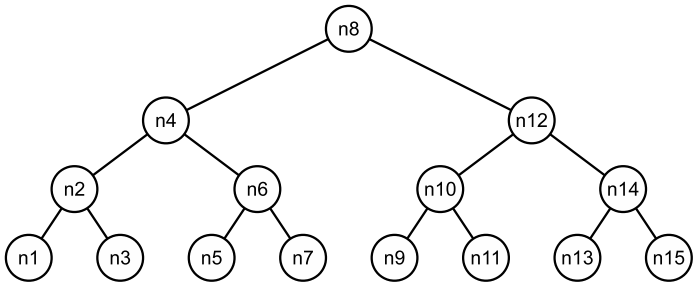


Figure 3.11. The location of patterns' indexes in a binary tree

At each clock cycle, every processor's stage module reaches a single node value from its RAM to perform a comparison. A memory address generation unit is necessary for each stage module of the binary tree processor. The memory address generation is necessary to access the right node value from the RAM cells. The module can be integrated with the level processing unit (module 'proc.'). The address for the stage's RAM is generated according to the comparison results that were performed

by the stages that had run prior to the considered stage. Let's assume that c_l is a comparison result at the level l. If $Data > Pattern$, then c_l is '1', otherwise c_l is '0'. The cell address ($addr_l$) for RAM at level l can be expressed as given by equation

$$addr_l \leftarrow c_{L-1}\&c_{L-2}\&...\&c_{n+2}\&c_{l+1},$$

where '&' is a symbol for bits concatenation.

If the match occurs at a certain level, the hit information is placed in the 'hit info' register. The register consists of the two fields: the level number l and matching RAM address for that level. One can use the level number and address to recover the matching node index. The index recovery procedure is described in the next paragraph. The task of the node index calculations can be performed in the hardware. The index derived from the tree can be directly used to access the matching pattern in the dictionary. Thanks to that feature, it is possible to determine the matching pattern immediately which is an advantage if compared to the lack of this functionality offered by the Bloom filter algorithm.

3.6.2. Mapping of patterns to memories

The binary search that is implemented as the binary tree processor requires that patterns are properly located in the stages' memories. Therefore, the processor's architecture should offer an adequate mechanism that maps the patterns that are downloaded during a configuration phase to the correct address at the right RAM block. The tree processor offers a self-organize mechanism of patterns in the proposed architecture. The sorted reference patterns can be sequentially downloaded to the device, and each element is automatically stored in the correct RAM cell. The addresses are internally generated; therefore, no external addressing of the nodes is necessary. The only requirement that applies is that the data has to be sorted in the downloaded stream. Also, the reverse mechanism guarantees that the appropriate pattern index is produced during the matching process. The output pattern index is derived from the matching level number and memory block address.

As it is presented in Figure 3.12, the level number l and the memory block address $addr$ can be relatively simply derived from the pattern (node) index $n_{bn...b2,b1,b0}$, where $bn...b2, b1, b0$ is a binary representation of the node index. Level number l can be derived from the less significant bits of the index. For level l, we need to decode $l + 1$ less significant bits. All bits but the first must be '0' in the less significant $(l + 1)$-bit field of the index. The rest of bits (most significant) can be used to address the selected memory block. That observation can be used by the hardware to address the RAMs properly on configuration, and to generate the right pattern index on the output during operation.

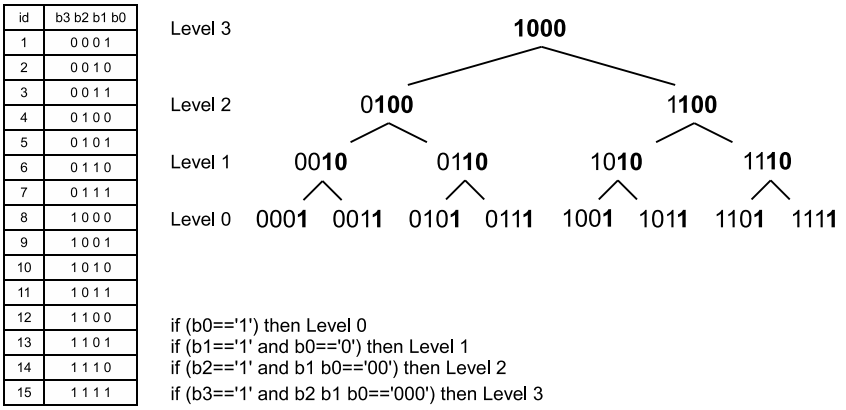


Figure 3.12. The method for mapping of patterns' indexes to the tree level and memory address

References

Le and Prasanna [129] presented the string matching for NIDSs. It was implemented in the Virtex-5 FX200T FPGA. A novel method of memory-efficient architecture for string matching was proposed. The method was based on a pipelined binary search tree. The implementation results showed a sustained throughput of 3.2 Gbps. The dictionary update involved only rewriting the memory content, which could be done quickly without reconfiguring the chip. Very efficient memory usage in this solution was achieved.

3.7. The prefix tree

The prefix tree, also called the trie, is a data organization structure that helps to retrieve elements from a defined data set. Like the binary tree, it consists of leafs (nodes) that are connected by branches (edges). Formally, the tree is a directed graph where exactly one path exists between any two nodes. Also, the tree distinguishes one of the nodes that is the root. As the root is the end node of any graph path, the edges naturally point from the root to other nodes, and each node is the root's descendant. In contrary to the binary tree, where nodes are associated with elements of the reference set, each node of a trie represents a single character. One can restore the strings of the set running all paths down the trie and collecting characters on the way from the root to the bottom leafs. Each path from the root node to the bottom node represents a single element from the data set. The example of the trie is given in Figure 3.13a. For simplicity, an alphabet is reduced to '0' and '1' symbols and the dictionary consists of binary strings.

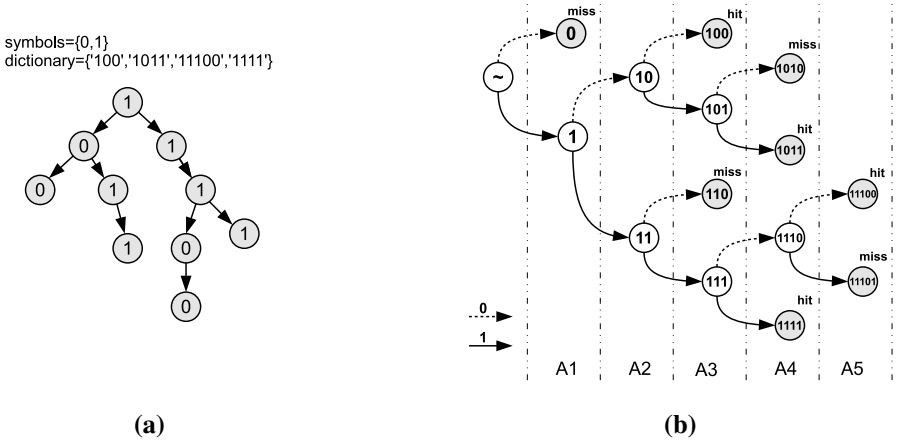


Figure 3.13. The design method of a pipelined automata for a prefix tree: a) an example of the prefix tree; b) an FSM automata for the prefix tree

The searching in the trie is easy: one takes a string and follows down the tree, taking branches according to the string's consecutive symbols. The pattern fits if it reaches the trie's bottom. Thanks to a tree structure, all dictionary's patterns are searched simultaneously. The root node usually represents an empty symbol (no character) because all the elements of the set rarely start with the same character. The trie compresses information because element patterns share nodes that keep their prefixes. For the trie, to have a good effect on compression and search, the dictionary strings should share prefixes, and that is where the prefix tree takes its name.

For a hardware designer, the prefix tree resembles an FSM diagram. FSM's input symbols are characters of the alphabet, and the output are 'hit', and 'miss' symbols. The trie can easily be converted to the FSM. Interestingly, the trie structure is an acyclic graph; therefore, it is possible to run FSM as a pipelined system. The known method of automata division [130] lies behind the concept of pipelining. The state transition path is the one-way route from the root to a bottom leaf in the case of the tree-like state diagram. Accordingly, it is possible to process many independent input symbols in one FSM hardware simultaneously. We divided nodes into five groups in Figure 3.13b. Each group consists of nodes (or FSM states rather) that can be reached in the same number of FSM transitions, starting from the root. The groups constitute the independent FSMs that process the character of the definite position from the input string. The procedure goes as follows. The first FSM (A1) takes the first symbol from the input string, selects the state and outputs to the second FSM (A2) its state information together with 'hit/miss' symbol. A2 takes the second input string's character and the state info from A1, processes and sends results to A3. At the same

time, A1 takes the first symbol of the next input pattern. The procedure continues, and effectively the system analyzes five input patterns at a time. The block diagram of an architecture for the trie processor is given in Figure 3.14.

The presented design fits perfectly into a shifting string search scenario because serialized characters can be sent one-by-one to all stages of the system simultaneously. Also, the searched patterns are localized in any position in the input string. That property is similar to that of the Aho-Corasick algorithm (see Section 3.8.) The parallel pattern delivery requires additional delay registers to synchronize data on the stages' inputs.

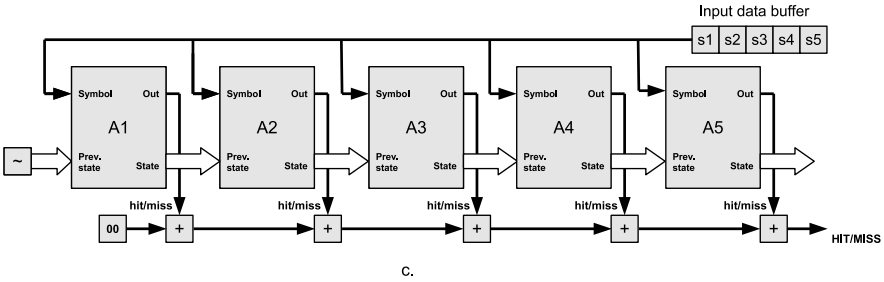


Figure 3.14. A pipelined prefix tree processor

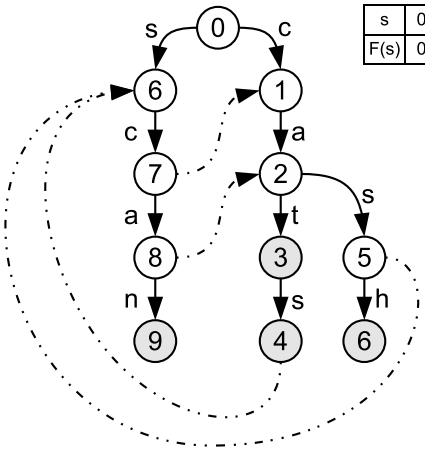
3.8. The Aho-Corasick algorithm

Back in 1975, Alfred V. Aho and Margaret J. Corasick [131] proposed the algorithm that efficiently solved the problem of the shifting sub-string search for a large set of patterns. Computational complexity of the Aho-Corasick Algorithm (ACA) is linearly proportional to the input string length l plus the total length of the patterns p plus the number of output matches m: $O(l + p + m)$. In fact, the authors proposed two versions of the algorithm in their paper. Both versions had derived from FSM, but only the second version was true DFA that made exactly one transition for each input symbol at each step. The DFA version makes the algorithm be another example of a software method that is easily applicable to the hardware.

Thanks to the additional transition links between the various internal nodes, the ACA is an enhancement of the presented in the previous section trie-based method. Like the pipelined trie search algorithm, ACA scans the shifting input string and finds occurrences of patterns in a single pass. In difference to the trie-based approach, the FSM for the ACA does not need pipelining. The ACA takes the single-pass ability from algorithmic, not hardware, property. Thus, the ACA suits also a sequential execution by a CPU. The second version of ACA, proposed in [131], is a true DFA that is ready for hardware application. Comparing the custom hardware for the ACA and

pipelined trie, both FSMs require the same number of states but the number of transitions is higher for the ACA. On the other hand, a pipelined trie requires passing the 'Previous state' (see Figure 3.14) information which consumes additional resources.

The Aho-Corasick method involves the construction of the three functions: Goto function, Failure function, and Output function. Construction of the Goto function is equivalent to the construction of a prefix trie. If we denote Goto function as $G(s, c)$, where s is an FSM state and c is an input symbol, $G()$ returns the next state in the prefix tree. Figure 3.15 gives an example diagram for the ACA if the patterns are: 'scan', 'cat', 'cats', and 'cash'. The arrows denote the Goto function, dashed arrows represent the Failure function, and filling of nodes gives the Output function; 'gray' filling is for the 'hit' output.



s	0	1	2	3	4	5	6	7	8	9
F(s)	0	0	0	0	6	6	0	1	2	0

Figure 3.15. A diagram of an example FSM for the Aho-Corasick algorithm

For example, $G(2, \text{'s'})$ returns '5' in Figure 3.15. The picture shows only correct transitions, but a mismatch to expected symbols is possible at each node also. For example at node '2', symbols 't' and 's' are valid only. Symbols that are not correct cause the machine to execute the Failure function. The failure means that the algorithm swaps the states and tries to execute the inadequate symbol at a new FSM position. What the Failure function does, it points to the next state that should be checked when the failure occurs. In most cases, the failure brings algorithm realization to the '0' state which is the root state.

The construction of the Failure function for the ACA expects that one adds additional transitions that cover common substrings of patterns in the dictionary. More specifically, patterns that contain a prefix of another pattern should be regarded. For example, pattern 'scan' contains the substring 'ca' that is a prefix of the pattern 'cat'. Consequently, the inclusion of the substring 'cat' causes the Failure function $F(s)$ to

return state ‘1’ for state ‘7’ ($F(7) = 1$) and state ‘2’ for state ‘8’ ($F(8) = 2$). Dashed lines denote Failure function transitions in Figure 3.15.

The Output function signalizes when the algorithm meets the match of any dictionary pattern and the input string. The Output function returns the value of the detected match for states that are gray circles in Figure 3.15.

The performance of the Aho-Corasick algorithm can be improved by its implementation in hardware. It is a standard to use memory blocks for realizing next-state and output logic for large state machines. It is also a case for the ACA automata, but the problem of alphabet size exists in practical applications. For the eight-bit character string, it is necessary to resolve 256 transition vertexes for each state. Despite the fact that the majority of the transitions in the ACA algorithm are transitions to the root state, the hardware logic has to implement those passings. Repetitive encoding of the default state transition results in redundancy in the memory that implements logic of the next-state function.

The ACA is a choice for search applications if the alphabet is small in comparison to the size of the dictionary. That scenario leads to prefix sharing and excellent utilization of the ACA properties. The opposite scheme, where the alphabet is rich, and the number of patterns is modest, suggests considering the binary tree rather.

References

Tan and Sherwood [132] were the first to propose the memory spare ACA hardware architecture. They compress the state machine by converting it to many tiny state machines. In the method, each state machine analyses a distinctive portion of bits for each symbol. All tiny state machines run in parallel, and the match arises when all machines agree. Jung, Baker, and Prasanna improved the method further in their work [133].

4. The Hash Binary Tree

4.1. Hashing of the binary tree patterns

Functional elements and registers that are implemented in matching custom processors offer arbitrary bit-width. Thus, comparators perform a comparison operation that takes a one clock cycle despite the patterns' length. The advantage of the custom architecture is that the size of comparators and registers implemented in hardware can fit the size of the matched patterns. The length has to be reasonable, but it can be much longer than a CPU's register size if it is necessary. This capability of custom processing exploits the low-level parallelism. However, very long comparators introduce inefficient utilisation of resources in the case of pattern comparison. A CPU divides a long pattern into fragments that fit its registers and performs a full comparison in several clock cycles. It often happens that a CPU determines that the pattern does not match after comparing its very first characters only. If the beginning chunk of the pattern does not match, a CPU also completes a comparison in one clock cycle. This phenomenon leads to the conclusion that the implementation of long comparators in hardware processors is a waste of computational resources, and low-level parallelism is not very attractive in that case. The observation brought the author to the idea of pattern compression to spare register flip-flops and comparator gates.

An interesting functionality of the binary tree processor that is presented in Section 3.6.1 can be achieved if patterns of the tree nodes are replaced by their hashes. The procedure of preparing an appropriate tree structure is very simple for the given dictionary of patterns. It is enough to compress (hash) the patterns, sort the hash values and download them to the tree processor. The binary tree that holds hashes is named the Hash Binary Tree (HBT) in this work. Like the Bloom filter, the HBT requires the verification of the results to reject 'false-positives'. However, the clear advantage of the HBT is a simplification of the results validation. The Bloom filter requires an additional index retrieval procedure before one can fetch the original pattern to check up a 'false-positive'; in contrary, the HBT offers quick pattern verification because the original pattern index is known from the tag of the matching node.

4.2. Two-fold pipelined HBT architecture

The binary tree processor allows for pipeline execution, and the same property applies to the HBT processor that is the binary trie processor with reduced memory requirements. The pipeline execution of the HBT introduces a latency. However, the latency usually does not matter for the execution of sequential data; and therefore, pipelining is particularly useful when a stream of input data must be processed. It is possible to execute simultaneously L input elements at each level of the tree. The bigger the patterns' dictionary, the more tree levels have to be used, and deeper parallelism is possible. However, the big disadvantage of the tree is that it is necessary to double the number of nodes to add another level to the tree. That can lead to quick saturation of memory resources if HBT is implemented in the FPGA. Consequently, an excessive amount of memory might be necessary to achieve a good degree of pipeline parallelism.

Additionally, the SIMD parallelism can be achieved when HBT is implemented on the FPGA platform. Thanks to the dual port property of the FPGA's Block RAMs, the simultaneous processing of two data streams is possible. One can use that property to process two streams of different hash functions simultaneously. The concept is presented in Figure 4.1.

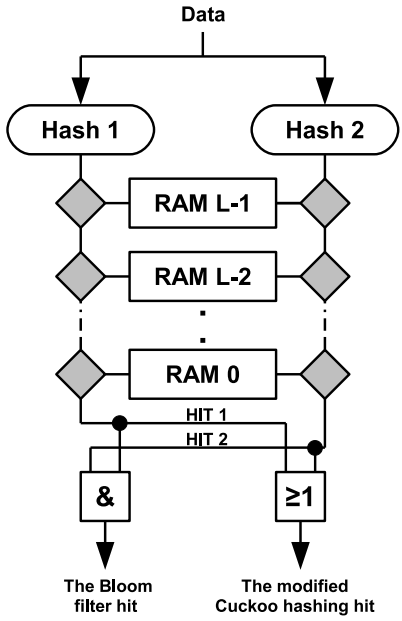


Figure 4.1. The two-fold configuration of the HBT for the modified cuckoo hashing and Bloom filter

The concurrent execution of two hash streams can be used in the two-fold HBT architecture. It can be used either to implement the Bloom filter with two hash functions or the two-way modified Cuckoo hashing. It is worth to note that the core architecture of the HBT processor is the same for the Bloom filter and the Cuckoo hashing implementation. The difference lays in the algorithm that is used to prepare the hash values. The two results from the HBT streams are marked the 'Hit 1' and 'Hit 2'. These results are logically AND-ed when the Bloom filtering is used, and they are OR-ed for the processing with the modified Cuckoo hashing algorithm.

4.3. Memory requirements of the HBT

The advantage of the proposed HBT solution is that it offers a matching result in every clock cycle. Besides, in contrary to the Bloom filter, the HBT provides not only membership verification but also a matching pattern identification. This functionality of pattern's index recognition is similar to that offered by the index hash table. In this sense, the HBT can serve as the hash index where it is necessary. We will see that the HBT requires fewer memory resources than the hash index in this section. Additionally, it will be presented that the HBT needs fewer bits of memory than the Bloom filter for the selected system parameters.

The Bloom filter is very competitive to the HBT approach in operations where only membership verification is required. In some applications, however, the HBT is a better choice because of lower memory size requirements. The memory requirement of the Bloom filter is 2^h bits. The HBT consumes $h * 2^L$ bits, where h is the hash bit-width and L is the number of tree's levels. Let the value of L_{max} is an upper bound value of L when the HBT filter requires fewer memory bits than the Bloom filter. The formula for L_{max} is given by

$$L_{\text{max}} = \frac{h\log(2) - \log(h)}{\log(2)},$$

that is a solution of the equation

$$2^h = h * 2^L.$$

The upper bound of L for selected values of h is also presented in Figure 4.2. For example, in the figure one can see that if the hash width is twenty bits, and the number of levels of the HBT is smaller than fifteen, the HBT requires fewer memory resources than the Bloom filter. That means that the architecture can match up to $2^{15} = 32,767$ different hashes. That is equivalent to the minimum number of $32,767$ patterns if $k = 1$. The number of patterns and the number of hashes are not equal because some patterns may share common hashes.

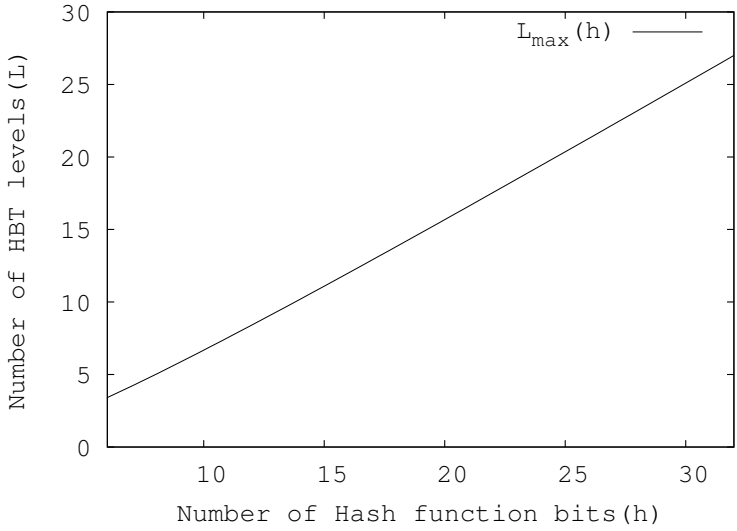


Figure 4.2. The value of L for equal memory requirements of the HBT and Bloom filter

Just like the hash table, the HBT algorithm returns the position of the pattern in the dictionary. Thus, the HBT can be also used instead of the hash index. It is interesting to compare memory requirements of both structures then. The hash table requires $L * 2^h$ memory bits, where $L = \lceil \log_2(n) \rceil$ and n is the number of patterns. Because the HBT consumes $h * 2^L$ bits, the memory requirements of the HBT and the hash index are equal if $h = L$. This equation is valid for the best hashing scenario *i.e.* the minimal perfect hashing (see Section 3.4.2). That leads to the conclusion that the HBT always requires less memory than the hash index table.

4.4. The example application

The HBT processor was used by the author to implement the problem of mapping text terms to term's unique identifiers (Ids). The goal of mapping is to replace each text words with an integer number. This procedure is used as the pre-processing step in the text analysis applications for example. It processes input text data and replaces each word (term) by a unique identification number. The dictionary of 58,109 English words was processed to develop hashes that were downloaded to the HBT. The HBT had sixteen levels and kept 24-bit hash in each node.

The two-way ($k = 2$) modified Cuckoo hashing was used to derive hashes from the dictionary words. The CRC32 function was chosen to create the hashes. The reason for the use of the CRC32 function was that it is easy to implement and parallelize

in FPGA logic. Two similar CRC32 hardware modules generated data for the two HBT paths. The first module calculated hash values of the input words while the second module hashed inverted bytes of the words. Hashes were truncated to 24 bits in order to fit the HBT memory words. That procedure allowed the mapping of all but eight words from the English dictionary to their unique 16-bit Ids.

The proposed hardware was implemented on the Zedboard platform [134]. The ZedBoard development kit uses the Xilinx XC7Z020 FPGA from the Zynq-7000 family of All Programmable SoCs [135]. Zynq-7000 integrates two ARM Cortex-A9 CPUs with an FPGA structure. The HBT processor was located in Zynq's Programmable Logic (PL). The 32-bit, high-performance AXI4 interface was used to couple the HBT with the ARM Processing System (PS). The HBT processed 32-bits of input text data at one read cycle. In order to hash words correctly, the text alignment module was necessary to place single words in 32-bits compatible boundaries. The system clock frequency was 100 MHz, so the theoretical throughput is 400 MB/s. The AXI DMA [136] module was used to transfer data between the HBT and the Zedboard's operation memory (32-bit 533 MHz DDR2 RAMs). The DMA ran in the simple mode, and the real throughput was measured to be 398 MB/s. The FPGA resource utilization is summarized in Table 4.1. The RAMB36E1, and RAMB18E1 respectively are 36 kbit and 18 kbit, the BRAM blocks. The Xilinx's Design Suite 14.6 toolchain was used for design, synthesis and implementation of the system.

Table 4.1. The logic utilization of the HBT processor ($L = 16$, $h = 24$) in programmable logic of Zynq XC7Z020. The two CRC32 hash modules and word alignment module were included

Resource	Utilization	Available	Utilization
Register	4,042	106,400	3%
LUT	4,213	53,200	7%
Slice	1,728	13,300	12%
RAMB36E1	96	280	34%
RAMB18E1	8	280	2%

The HBT consumed *ca.* $96 * 36$ kbit $+ 8 * 18$ kbit $= 3,686,400$ bits of memory. The regular hash table would require $16 * 2^{24} = 256$ Mbits of memory, for the presented terms mapping problem. It is worth noting that such memory resources are not available in today's FPGAs. Thus, it is a clear advantage of the HBT scheme. The perfect hashing requirements would be 2^{16}, but it is probably not feasible to find an appropriate hash function to implement this method in the terms mapping application.

The power consumption of the XC7Z020 FPGA SoC was 2.8 W. The power was estimated by Xilinx's Power Estimator tool [67].

For reference, the algorithm of mapping terms to term's IDs was also run on a computer server with two Intel's Xeon 5650, and 16 GB of memory. Twelve CPUs were available. Thanks to OpenMP, the problem was parallelised to six threads, and run on six CPUs. The input text and index table were located in the operating memory, and the text was divided into six separate buffers, which were processed by individual threads. To improve the CPU performance, the CRC32 function was replaced by the SAX hash function. SAX has good statistical properties also, but it is faster when implemented in software. The Intel's C++ Compiler (icc) was used. The maximum measured performance of Xeon's six CPUs was *ca.* 460 MB/s. The reason to chose six CPUs was that it is equivalent to one processor socket and one memory bank that is similar to SoC's configuration. Another processor socket has own memory bank in the server architecture. A single CPU core was tested for the mapping application also, and it achieved the performance of approximately 80 MB/s. The Thermal Design Power of Xeon 5650 is 95 W.

4.5. Conclusions

The performance of multi-core processors cannot be fully utilized in the case of IO-bound problems. That was the reason that the 3 W SoC came out to be equivalent to a high-performance server. The mapping is a strongly data-dependent problem, and the data-dependent algorithms play an important role in the web's computing infrastructure.

The energy consumption and power availability are the major problems when the building of a new computer infrastructure is considered. The infrastructure maintenance, not the purchase costs play the most important role. The cut of power dissipation is the priority for the further growth of resources that are available at computing centers. Many techniques have become popular recently, but most of them focus on more efficient cooling methods. An alternative method is the employment of energy-efficient processors, which is not easy because it requires the parallelization of existent algorithms.

Efficient parallelization requires the development of new algorithms and methods. In the author's opinion, the existence of FPGA computing should not be neglected when the new methods are considered.

The existing FPGA design tools have matured enough, to allow for fast hardware architecture development. The software can be developed faster than hardware, but the design of good quality parallel software is also complicated and time-consuming. That should increase the importance and attractiveness of custom hardware solutions.

The development of the VHDL code for the HBT architecture was much faster than had been expected. It took approximately three man-months. Obviously, it is not feasible to rebuild all existing codes to migrate them to dedicated computer architectures; however, the shift of some kernel computations towards an energy-efficient environment is seriously considered in the industry.

The module that was presented here may be a part of the hardware library of IPCores that can be used in the hybrid cluster of energy-efficient and FPGA-enabled nodes.

5. Acceleration of genome matching

5.1. Short-read alignment

The new DNA sequencing technologies generate an enormous amount of, so-called, short reads. That calls for fast read alignment programs [137]. The application that is presented in this chapter gives an example of the use of the trie processor for a problem of genome matching. Thanks to the use of a custom processor, it is possible to enhance a process of DNA short-read alignment in the computer system that contains an FPGA accelerator. A real-life DNA matching problems are huge and requires a big memory to keep DNA data. The enormous size of the input makes the FPGA a perfect solution to perform data pre-processing and to act as a co-processor of a CPU. Although, both the hardware and software part is necessary to deliver a complete solution to the problem, it will be the hardware component that is presented in this chapter only. The reader will be provided with references that allow him/her to study the concept of the software for the system.

The short-read alignment belongs to a class of genome matching problems. An introduction to the problem of DNA short-read alignment that is sufficient for a hardware designer is provided by Heng Li and Richard Durbin [138]. The authors delivered a problem description and a successful implementation of DNA short-read alignment as a software application. The authors took advantage of the Burrows-Wheeler transform (BWT) to achieve the solution that is fast and spares memory space.

In short, the problem of short-read alignment is to match a set of short genome reads that come from a genome sequencer with a long DNA reference sequence. Usually, the source of short reads are commercially available the Next-Generation Sequencing (NGS) instruments. The genome is represented as a sequence of the four-letter alphabet (‘A’, ‘C’, ‘T, ‘G’). The sequence “AGCATGCTGCAGTCATGCTTAGGCTA” is a trivial example the reference genome sequence that DNA matching algorithms have to cope with. The single short read can match the reference genome sequence at any letter position. In practice, the reference sequence can be as long as a couple of Gbp (‘bp’ stands for base pair *i.e.* a letter). The set of short reads can

comprise of 50-100 M elements, 32-100 bp each. The most important difficulty in the DNA alignment is that it accepts inexact matches as well. There letter insertions, deletions, and mismatches are acceptable in genome matching. Also, the number of acceptable errors for the match is limited, and it is a parameter that is set for a given matching run.

5.2. Subsequences

The design of the co-processor that is proposed in this chapter starts with an extraction of all subsequences of length L from the reference genome. For instance, for the reference sequence that was proposed in Section 5.1 and the length $L = 4$, we obtain the following set of subsequences (given in the order of appearance):
{'AGCA', 'GCAT', 'CATG', 'ATGC', 'TGCT', 'GCTG', 'CTGC', 'TGCA', 'GCAG', 'CAGT', 'AGTC', 'GTCA', 'TCAT', 'CATG', 'ATGC', 'TGCT', 'GCTT', 'CTTA', 'TTAG', 'TAGG', 'AGGC', 'GGCT', 'GCTA'}.

In theory, the choice of L depends on the maximum expected short-read length. However, a value of L is mainly limited by a quantity of available FPGA resource in the case of the implementation of short-read alignment as a custom processor. The processor would provide a definite location of a short read in the reference sequence if a short-read length was smaller than L. As a short read is longer than L, the processor will point many positions of candidates that share the common prefix of length L. Those candidates have to be further verified and resolved by a CPU. We may say that a screening of possible short read locations within the reference DNA is the main task of the FPGA accelerator in our scheme. That will be further explained later.

The trie of subsequences is necessary to create the short-read alignment processor. To create the trie, one has to sort the subsequences first. Repetitions are possible, and they are highlighted in bold letters in the following sorted list:
('AGCA', 'AGGC', 'AGTC', 'ATGC', '**ATGC**', 'CAGT', 'CATG', '**CATG**', 'CTGC', 'CTTA', 'GCAG', 'GCAT', 'GCTA', 'GCTG', 'GCTT', 'GGCT', 'GTCA', 'TAGG', 'TCAT', 'TGCA', 'TGCT', '**TGCT**', 'TTAG').

The probability of read repetitions is minimal for long enough subsequences. Unfortunately, it is a significant phenomenon when FPGA acceleration is considered because the value L is limited and rather small in that case. Nonetheless, the repetition can be handled easily. Simply, when a multiple match occurs, the CPU will check all reference sequence positions that start with the common prefix of length L. Here, we will remove the repetitions to build the trie for the co-processor. We have the list:
('AGCA', 'AGGC', 'AGTC', 'ATGC', 'CAGT', 'CATG', 'CTGC', 'CTTA', 'GCAG', 'GCAT', 'GCTA', 'GCTG', 'GCTT', 'GGCT', 'GTCA', 'TAGG', 'TCAT', 'TGCA', 'TGCT', 'TTAG').

The sorted list of subsequences will be referred to as the Sorted Subsequences List (SLL) in this paper. Additionally, for the discussion of the custom processor and CPU cooperation, it is useful to create a table that contains locations of the SLL items within the reference genome. The table will be called the Subsequences' Positions Table (SPT). In our example, the SPT looks as follows:
(1, 21, 11, 4, 10, 3, 7, 18, 9, 2, 23, 6, 17, 22, 12, 20, 13, 8, 5, 19).

5.3. The trie

The sorted list of subsequences is used to build the search trie. The trie's root represents an empty '_' symbol, and each trie node has an associated letter 'A','C','G' or 'T'. At the lowest level of the trie, each node corresponds to the single SSL entity. The subsequence that corresponds to the trie node can be determined from letters those make a path from the root to the node. The letters which constitute the paths from the root to the nodes of the lowest level make the sequences for the SLL members.

The trie can be used for fast matching of short reads with the reference genome. Each match starts at the root and traverses the trie, according to the short-read letters. If the short read matches any genome's subsequence, it will find a valid path in the trie. For the short reads of the length $l < L$, one can locate the read in l steps of letter matching.

If a matching short read is shorter than L, it corresponds to the group of subsequences. For example, the node denoted as '1' in Figure 5.1 corresponds to the group {'GCTA', 'GCTG', 'GCTT'} and the node '2' corresponds to the sequence 'TTAG' only. An algorithm knows where to find the matching subsequence in the genome if it knows how to locate the group in SPT. It is an important property that the subsequences that share a common prefix are adjacent on the SLL. Thus, the positions of subsequences that correspond to a given node in genome sequence can be easily retrieved from the SPT. For example, if the search stops at the node '1' then the matching group boundaries on the SLL are <11, 13>. It reflects ('GCTA', 'GCTG', 'GCTT') group and genome locations {23, 6, 17} in the SPT.

As we stated, the real-life short reads are longer than a value of L that can be implemented in FPGA. That is a reason to build the additional SSL and SPT for subsequences of length $l > L$ in software. These tables are to be used by the CPU, which cooperates with the FPGA co-processor in genome location checkup procedure. Each subsequence of length L on the hardware SSL corresponds to many subsequences of length $l > L$ on the software SLL. Accordingly, each lowest level node represents a group of subsequences of length $l > L$ that share the common prefix of length L in the software SSL. The subsequences of the software SSL that share a common trie node (and similarly the common prefix) are adjacent. That property is exploited in

a procedure of the resolution of short read locations that is performed by the CPU. It will be discussed in Section 5.10 how to follow Li and Durbin [138] and replace the software SSL and SPT with BWT's structures for memory efficiency.

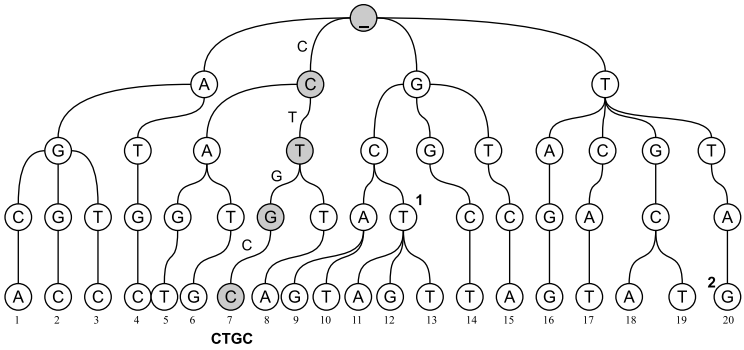


Figure 5.1. An example of the trie for short-read alignment

5.4. The sequential co-processor

Figure 5.2 presents a custom processor for the sequential traversing of the trie. The solution uses a single block of memory, and it resembles a well-known, RAM-based implementation of an FSM automata. The figure explains principles of processor's operations in a form of a block diagram.

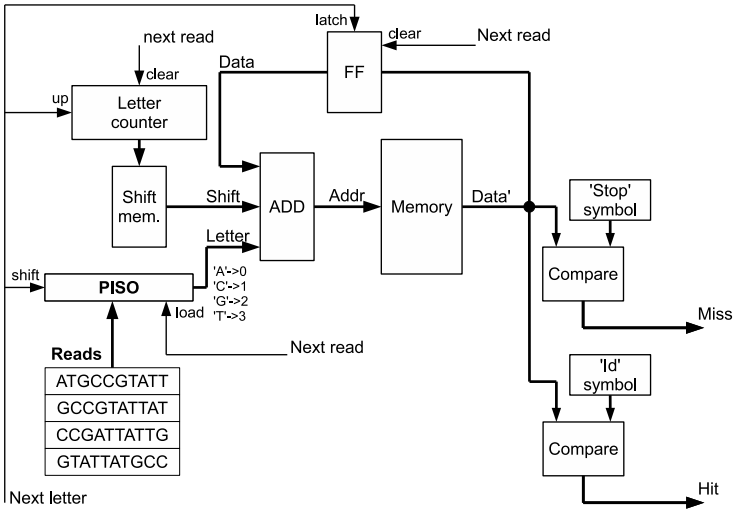


Figure 5.2. A block diagram of the sequential processor for short-read alignment

			Shift=4
Addr	First letter	Data	Data+Shift
0	A	0	4
1	C	4	4
2	G	8	8
3	T	12	12

			Shift=20
Addr	Second letter	Data	Data+Shift
4	AA	stop	
5	AC	stop	
6	AG	0	20
7	AT	4	24
8	CA	8	28
9	CC	stop	
10	CG	stop	
11	CT	12	32
12	GA	stop	
13	GC	16	36
14	GG	20	40
15	GT	24	44
16	TA	28	48
17	TC	32	52
18	TG	36	56
19	TT	40	60

			Shift=64
Addr	Third letter	Data	Data+Shift
20	AGA	stop	
21	AGC	0	64
22	AGG	4	68
23	AGT	8	72
24	ATA	stop	
25	ATC	stop	
26	ATG	12	76
27	ATT	stop	
28	CAA	stop	
29	CAC	stop	
30	CAG	16	80
31	CAT	20	84
32	CTA	stop	
33	CTC	stop	
34	CTG	24	88
35	CTT	28	92
36	GCA	32	96
37	GCC	stop	
38	GCG	stop	
39	GCT	36	100
40	GGA	stop	
41	GGC	40	104
42	GGG	stop	
43	GGT	stop	
44	GTA	stop	
45	GTC	44	108
46	GTG	stop	
47	GTT	stop	
48	TAA	stop	
49	TAC	stop	
50	TAG	48	112
51	TAT	stop	
52	TCA	52	116
53	TCC	stop	
54	TCG	stop	
55	TCT	stop	
56	TGA	stop	
57	TGC	56	120
58	TGG	stop	
59	TGT	stop	
60	TTA	60	124
61	TTC	stop	
62	TTG	stop	
63	TTT	stop	

Addr	Fourth letter	Data
64	AGCA	Id=0
65	AGCC	stop
66	AGCG	stop
67	AGCT	stop
68	AGGA	stop
69	AGGC	Id=1
70	AGGG	stop
71	AGGT	stop
72	AGTA	stop
73	AGTC	Id=2
74	AGTG	stop
75	AGTT	stop
76	ATGA	stop
77	ATGC	Id=3
78	ATGG	stop
79	ATGT	stop
80	CAGA	stop
81	CAGC	stop
82	CAGG	stop
83	CAGT	Id=4
84	CATA	stop
85	CATC	stop
86	CATG	Id=5
87	CATT	stop
88	CTGA	stop
89	CTGC	Id=6
90	CTGG	stop
91	CTGT	stop
92	CTTA	Id=7
93	CTTC	stop
94	CTTG	stop
95	CTTT	stop
96	GCAA	stop
97	GCAC	stop
98	GCAG	Id=8
99	GCAT	Id=9
100	GCTA	Id=10
101	GCTC	stop
102	GCTG	Id=11
103	GCTT	Id=12
104	GGCA	stop
105	GGCC	stop
106	GGCG	stop
107	GGCT	Id=13
108	GTCA	Id=14
109	GTCC	stop
110	GTCG	stop
111	GTCT	stop
112	TAGA	stop
113	TAGC	stop
114	TAGG	Id=15
115	TAGT	stop
116	TCAA	stop
117	TCAC	stop
118	TCAG	stop
119	TCAT	Id=16
120	TGCA	Id=17
121	TGCC	stop
122	TGCG	stop
123	TGCT	Id=18
124	TTAA	stop
125	TTAC	stop
126	TTAG	Id=19
127	TTAT	stop

Id	Read
0	AGCA
1	AGGC
2	AGTC
3	ATGC
4	CAGT
5	CATG
6	CTGC
7	CTTA
8	GCAG
9	GCAT
10	GCTA
11	GCTG
12	GCTT
13	GGCT
14	GTCA
15	TAGG
16	TCAT
17	TGCA
18	TGCT
19	TTAG

Figure 5.3. The contents of the sequential processor ‘Memory’ for the trie of Figure 5.1

The architecture works as follows. The short reads are processed sequentially in the read by read and letter by letter manner. The processing of a new short read is initialized by the 'next read' signal. The 'next read' signal loads the new read into the Parallel Input Serial Output (PISO) register, clears the 'FF' data register, and resets the 'Letter counter'. The 'Next letter' signal triggers processing of read's consecutive letter. The address for the 'Memory' in the next cycle is calculated as a sum of the 'Letter', 'Shift' and 'Data' values. 'Letter' values are 0, 1, 2 and 3 for 'A', 'C', 'G' , and 'T' respectively. 'Shift' is kept in 'Shift memory', and it is an address offset to the next letter region in the 'Memory' table. 'Data' is stored in the 'FF' register and it provides an FSM's next-state info that was read from the 'Memory'. The contents of the 'Memory' is given in Figure 5.3.

The data that is stored in the 'Memory' can be either the next-state address, 'stop' word or 'id' word. The 'stop' word means that the next letter of the short read does not match any available trie's path. The 'id' word signals that a match occurs, and it delivers an index of the matching node. If the data is the next-state address, the search proceeds to the next short read's letter.

The 'Shift' value is different in every letter iteration. It is stored in the 'Shift Memory' which is addressed by the 'Letter counter'. In a sense, the 'Shift' value introduces the compression of the data in the 'Memory' because the address that is kept in this RAM is smaller of the 'Shift' value. In the other words, the values in 'Shift memory' mark the 'Memory' addresses where the data for the consecutive letter iteration begins.

5.5. The pipelined co-processor

The pipelined version of the short-read alignment processor will be proposed in this section. The advantage of the pipeline execution is that, in effect, a single short read is processed in a single algorithm step.

The pipelined architecture that is presented can be easily mapped into the FPGA structure. The single 'Memory' block that is a part of the sequential alignment processor is split into several memory blocks in the pipelined architecture. Each memory block is bond to a single pipeline stage. The block diagram of the architecture is given in Figure 5.4. The additional components that are associated with each stage are similar to those of the sequential processor version.

Serialized pipeline stages process the successive letters of the short reads. The first letter is processed by the 'First Letter Memory', the second letter is processed by the 'Second Letter Memory', *etc*. The processor processes as many letters as the number of stages at each clock cycle. According to a basic principle of outer loop pipelining, simultaneously processed letters belong to different short reads.

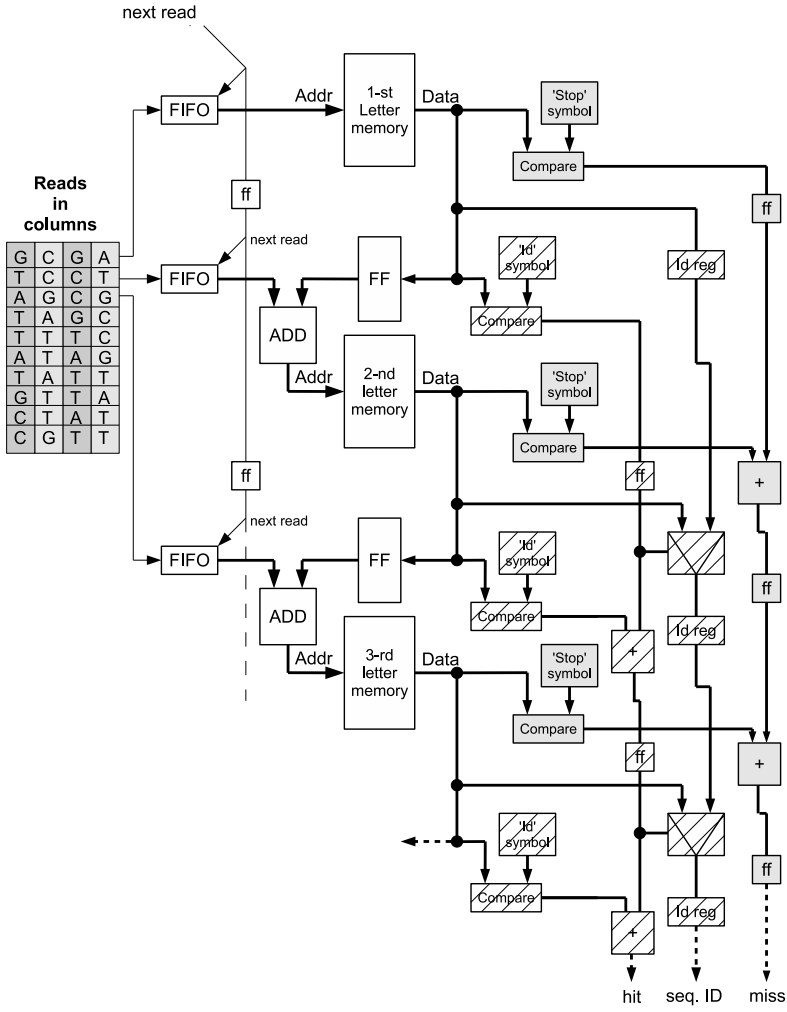


Figure 5.4. A block diagram of the pipelined processor for short-read alignment

Just like in the sequential version, the processor's memories contain next-state address data, 'stop' words and 'id' words. The addresses are passed from one stage to the another. The 'Shift' value of the sequential version is no longer necessary because memory blocks have separate address buses. However, the 'stop' word and 'id' word requires an additional propagation path in the pipelined architecture. Components of the 'stop' path are grayed, and elements of the 'id' path are lined in the figure. The 'id' word consists of the 'id' flag and 'id' value which holds matching node identification. The flag and value are propagated together by the 'id' path. The matching identification is latched in the 'Id reg' register.

The ‘id’ path is not necessary if the matches occur only in the last stage of the algorithm. It can be discarded if only the memory block of the final stage contains the ‘id’ words. That is a usual situation when the length of real short reads is longer than the length of subsequences L. According the assumptions taken in Section 5.3, it is always the case when the alignment processor works as a CPU’s co-processor. The size of the trie that suits that practical short-read range (the length of short reads starts with 30 bp) is out of the reach of today’s FPGA devices. Such big tries just do not fit into the size of available FPGAs. Consequently, the last stage carries ‘id’ symbols only.

5.6. Inexact matching

The exact search checks for the existence of a substring in a string template. In fact, exact matching does not apply to genomics. Accordingly, k-mismatch search is used instead in short-read alignment. The task of k-mismatch search is to locate substrings that have a maximum of k mismatches to a matching element. One can calculate the number of mismatches as a minimal number of edit operations that has to be performed on the substring, to match the string correctly. Three basic types of edit operation are in use: a substitution, insertion, and deletion of a letter.
The illustrations of the three edit operations are:

- substitution ‘AGC**A**TG’→‘AGC**G**TG’,
- insertion ‘AGCAT’→‘AGC**C**ATG’,
- deletion ‘AG**C**ATG’→‘AGATG’.

The naïve search algorithm would check for all possible modification of the string. For example, the series of subsequences that must be checked for ‘AGCT’ in one-mismatch search is:
{‘AGCT’, ‘CGCT’, ‘GGCT’, ‘TGCT’, ‘AACT’, ‘ACCT’, ‘ATCT’, ‘AGAT’, ‘AGGT’, ‘AGTT’, ‘AGCA’ ‘AGCG’, ‘AGCC’, ‘AGCG’, ‘AAGCT’, ‘ACGCT’, ‘AGGCT’, ‘ATGCT’, ‘AGACT’, ‘AGCCT’, ‘AGGCT’, ‘AGTCT’, ‘AGCAT’, ‘AGCCT’, ‘AGCGT’, ‘AGCTT‘, ‘GCT’, ‘ACT‘, ‘AGT’, ‘AGC’}.

The example illustrates a computational complexity of the k-mismatch search. In the series, we have the original string and a set of the twelve substitutions, twelve insertions, and four deletions. Thus, the series of exact matching operations must be performed to fulfill the k-mismatch search. Some of the sequences in the set are repetitions (‘AGCCT’ for example).

The series of modified subsequences can be generated dynamically during the matching process when a custom processor is employed. The idea to combine the

generation of mismatches with the matching operation limits the number of performed checkups. For example, there is no point to check for the 'AGCG' modified sequence if the 'AGC' prefix of 'AGCT' does not exist in the reference genome.

The concept of the edit-and-check algorithm is presented in Listing 5.1.

Listing 5.1. A pseudocode for the k-mismatch search

```
/* K-mismatch edit and search for a short-read sequence    */
/* Arguments:                                              */
/* trie - represents reference genome                      */
/* shortRead - input read                                  */
/* Return:                                                 */
/* Locations of matching mismatches                        */

LocationsType MismatchSearch(TrieType trie,
                             SequenceType shortRead){

/* Declare a list of candidate mismatch reads */
SequenceType readsList;
/* Declare container for mismatch locations in genome     */
LocationsType locations;

   /* Empty list of candidate mismatches */
   readsList.empty();
   /* Add single empty string to the list of candidates   */
   readsList.add("");
   /* Repeat for each letter in the input short-read      */
   for(letter: letters in shortRead) {
       /* Repeat for each candidate in mismatch read list */
       for(r: reads in readsList) {
       /* Get a next read from readList                   */
          read=r;
       /* Delete this candidate from the list             */
         readsList.delete(read);
         /* Check if read in the trie                     */
         if ( !trie.contain(read)) then {
            /* goto next candidate                        */
            break;
         }
         /* Add the next letter of the short-read to candidate */
         read.cat(shortRead.pos(letter));
         /* Add a new, longer candidate to the list       */
         readsList.add(read);
         /* Generate mismatches for candidate if possible */
         if (read.nbrOfMismatches < MAX_NBR_OF_MISMATCH) {
            /* Create candidate with last character deleted */
            candidate=readDelete(read);
            /* Add to the list */
```

```
                readsList.add(candidate);
                /* Create replacements and add to the list          */
                candidate=readSubstitute(read,'A');
                readsList.add(candidate);
                candidate=readSubstitute(read,'C');
                readsList.add(candidate);
                candidate=readSubstitute(read,'G');
                readsList.add(candidate);
                candidate=readSubstitute(read,'T');
                readsList.add(candidate);
                /* Create insertions and add to the list            */
                candidate=readInsert(read,'A');
                readsList.add(candidate);
                candidate=readInsert(read,'C');
                readsList.add(candidate);
                candidate=readInsert(read,'G');
                readsList.add(candidate);
                candidate=readInsert(read,'T');
                readsList.add(candidate);
            }
        }
    }
    /* Return positions of remaining candidates */
    for(read: reads in readsList){
        locations.add(read);
    }
    return positions;
}
```

The algorithm is simple, regular, and its outer loop can be easily pipelined. The inner loop can be implemented as the automata circuit in hardware respectively. The scheme of the appropriate hardware architecture is presented in Figure 5.5.

In comparison to the exact search architectures, the key modification is the introduction of the ‘Letter Edit and Match‘ (LEM), ‘FIFO Search Data‘, and ‘FIFO Edit Data’ blocks. Both the ‘stop’ and ‘id’ propagation hardware is not included in the figure. The LEM block generates all possible letter mismatches in a single short-read position. Each LEM block requires an access to its position letter and the letter’s neighbourhood letters. The neighbourhood is necessary for the insertions and deletions. For example, if the first stage of the LEM inserts a letter, the actual first letter must be shifted to the second stage LEM. Similarly, if the first LEM deletes the letter it needs to fetch the second letter to address the ‘First Letter Memory’. Moreover, the accumulated number of performed insertions and deletions has to be propagated between stages. For instance, if the first stage deletes the letter and the second stage

inserts one then, because of accumulation, the third stage uses the third letter. A balance of insertions and deletions is transferred through the 'FIFO Edit Data'.

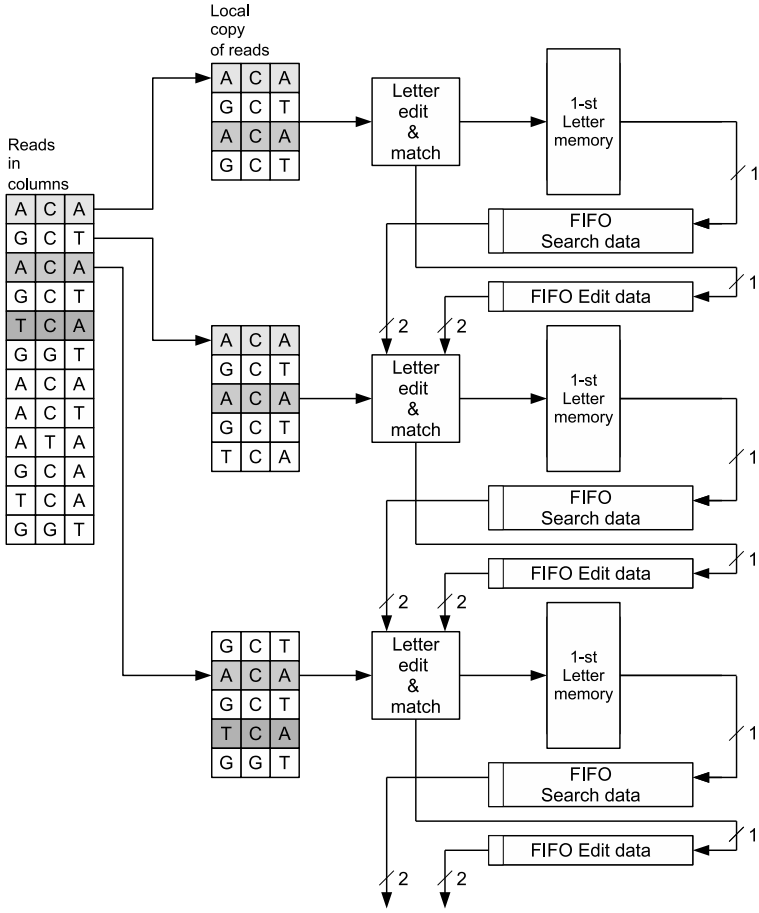


Figure 5.5. A block diagram of the pipelined custom processor for *k*-mismatch short-read alignment

The size of the neighbourhood that has to be accessed by the single stage depends on the maximum number of allowed edit operations. Also, the 'FIFO Edit Data' carries the number of edits that have been performed on the short read on its way. The 'LEM' does not introduce a new edit operation if the number of edits reached a programmed limit. Additionally, to keep track of short reads, the 'FIFO Edit Data' passes the 'id' of the currently processed item. Each stage generates a variable number of mismatches for a single short read because some of the candidates disappear on the way throughout the pipeline.

5.7. The control block

Each pipeline stage features its own ‘Letter Edit and Match’ element. The architecture and operations of the LEM element will be described in more details. Figure 5.6 presents the LEM block and its ports. The names of ports and their functions are as follows:

- ‘Data_in’ receives data that has been read from the ‘Memory’ of the preceding stage. The data is an address or a ‘stop’ symbol. The address and the input letter value are summed up to calculate the access location in the stage’s ’Memory’ block. The ‘stop’ symbol indicates the termination of the trie traversal *i.e.* operations of the remaining stages are abandoned. The ‘match’ symbols exist in the ‘Memory’ of the final stage only, so the LEM block never meets this symbol on its ‘data_in’ port.
- ‘Data_out’ propagates to ‘Data_in’ of the next stage.
- ‘Next_in’ and ‘Next_out’ are used to keep track of the short reads those are currently processed by the stage. An existence of ’stop’ symbols causes that the number of mismatches that are generated by the edit operation for a given short read is not deterministic. So, the LEM fetches a next read from the short-read buffer when it detects an active the ‘Next_in’ signal.
- ‘Edit_in’ is used by the LEM to control the number of edit operations that had been performed until the short read arrived at the corresponding pipe’s stage. Additional edit operations are possible only if the number of edits does not reach the preprogrammed limit. The binary coded value of ‘Edit_in’ is equal to the number of edits committed.
- ‘Edit_out’ propagates to ‘Edit_in’ of the next LEM. If a new edit operation is introduced by the LEM, ‘Edit_out’ equals ‘Edit_in’ plus one. Otherwise, ‘Edit_in’ copies ‘Edit_out’. If ‘Edit_in’ has reached the maximum number of allowed edits, edit operations are not allowed, and ‘Edit_out’ follows ‘Edit_in’.
- ‘Shift_in’ and ’Shift_out’ reflect the balance of the insert and delete operations that had been performed so far. The LEM uses the ‘Shift_in’ value to pick the letter from a correct short read’s position. When an insert operation is performed, the next stages should get a letter that is left positioned to the actual stage number position. Similarly, the right positioned letter should be taken when a delete operation has been performed in the previous stage. The insert and delete operations cancel each other. ’Shift_in’ is used to shift left or right a fetch position in the short-read buffer. Consequently, the LEM increments/decrements a value of ‘Shift_in’ when the delete/insert operation is performed. The ‘Shift_in’ value can be positive or negative, but its absolute value is always less or equal the value of ‘Edit_in’. Altered ‘Shift_in’ is sent to the ‘Shift_out’ port.

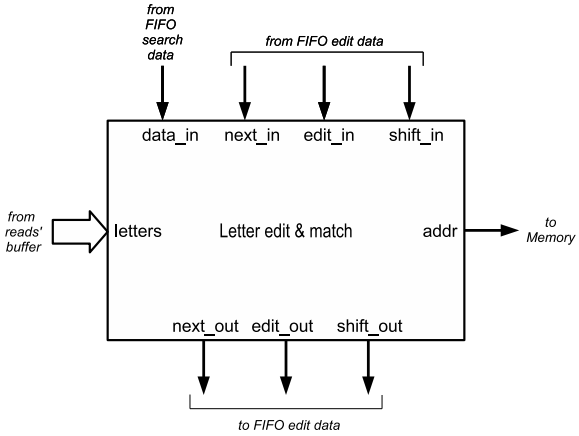


Figure 5.6. Input-output ports of the 'Letter Edit and Match' block

5.8. Resource requirements

The trie nodes are organized in levels. We will denote the maximum number of trie nodes at level $l = \{0, 1, 2, \ldots\}$ as NN_l, where $NN_l = 4^l$. The single top node '_' is localized at level $l = 0$. The memory block at each pipeline's stage consist of words, and the number of words is $4 * NN_l$.

As data that is kept in memory at level l addresses memory at level $l+1$, the size of a word at level l is

$$WS_l = 2 + \log_2 (NN_{l+1}).$$

If we denote the memory size at level l as MS_l, we have

$$MS_l = 4 * NN_l * WS_l.$$

The capacity of the trie level *i.e.* the number of its nodes grows very fast with a value of l. Consequently, it reaches the total number of available subsequences quickly. The number of subsequences is $NS = TL - L \approx TL$, where TL is a genome template length, and L is a subsequence length. The approximation is valid because TL is much bigger than L. It is possible to define l_{sat} which is the smallest integer value that satisfies $4^{l_{\text{sat}}} > NS$. The trie stops to expand its breadth at the level l_{sat} and the number of nodes saturates at l_{sat} level, where the maximum number of TL nodes reside. The expected memory requirement for $l > l_{\text{sat}}$ is

$$MS_l = 4 * TL * \log_2(TL).$$

The above discussion has an impact on the size of 'Memory' block that should be allocated for individual pipeline stages of the short-read alignment processor. The trie data is downloaded to 'Memory' blocks during processor operation, but the proper BRAM size has to be reserved in the design process. Figure 5.7 gives the theoretical and practical notion for memory size requirements.

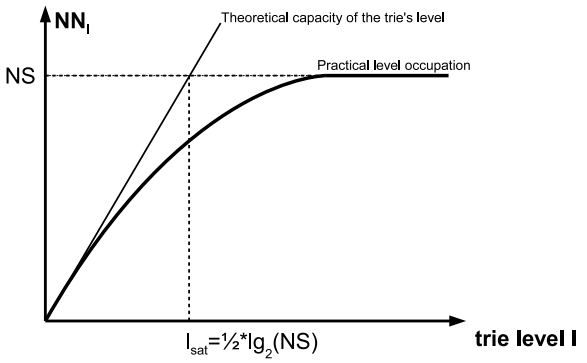


Figure 5.7. A visualization of memory requirement for the short-read alignment processor. The NN_L is a number of nodes at the trie level l, and the l_{sat} is the tree level, where a theoretical number of nodes is greater than the number of genome's subsequences

The FPGA coprocessor is not able to perform a complete short-read search. Today's FPGA devices are not capable of storing all nodes that are necessary to implement the trie that accommodates the real-life DNA data. The equations that were provided in this section allows it to derive the memory requirement for the single trie level l. The appropriate formula is given by equation

$$MEM_{\text{size}} = 4NN_l * WS_l = 4 * 4^l(2 + \log_2 4^{l+1}) = 4^{l+1}(2l + 4). \tag{5.1}$$

However, for levels $l > l_{\text{sat}}$, the formula takes l_{sat} instead of l. The real-life problems of DNA alignments require the short-read length of 30-100 bp, and a typical length of a genome ranges from 100 Mbp to 1.5 Gbp. Using Equation 5.1, one can calculate the memory requirements for $L = 30$ and $TL = 100,000,000$ bp ($l_{\text{sat}} = 14$). It reaches the prohibitive value of 207 Tb!

The disadvantage of the trie, when it is used for short-read alignment, is a huge expansion of the genome data size during SSL creation. The algorithm scans the reference genome to extract the set of the subsequences that overlap in the reference genome. Contrary, each subsequence is coded separately in the SSL. Thus, the refer-

ence genome that occupies $2 * TL$ bits of memory generates the set of subsequences that requires roughly $2 * TL * L$ bits! Although, the sequences share their prefixes in the trie, and this property mitigates the explosion of data, it compensates the problem only partially. Intuitively, the trie wastes most memory to implement its levels for $l > l_{\text{sat}}$ because no prefix sharing exists for those nodes. Therefore, it seems to be rational to implement the trie coprocessor that features up to l_{sat} levels only.

5.9. Reducing software memory requirements

Here, it is necessary to mention Li and Durbin's [138] work again. The authors proposed the fast match algorithm for CPU that needs $2 * TL$ bits to keep the reference genome. Their algorithm keeps the genome data in a form that is an output of Burrows-Wheeler Transform. The BWT of the reference genome is defined as B array in [138]. Additionally, the method of Li and Durbin needs two extra tables, which allow it to search in B array in a manner that resembles trie traversal. Thanks to the extra tables, the algorithm sees the reference genome as a list of sorted prefixes. That approach gives the algorithm speed and modest memory requirements. The extra tables are referred as $C(\cdot)$ and $O(\cdot, \cdot)$ in [138]. Array $C(\cdot)$ requires four words only and can be neglected in the discussion. However, the complete $O(\cdot, \cdot)$ array needs $4TL\lceil \log_2(TL) \rceil$ bits, which is a huge memory. Therefore, Li and Durbin store $O(\cdot, k)$ for k that is a factor of 128 and calculate the rest of the elements using the content of the stored BWT array B at program runtime.

Each trie's node in an FPGA custom processor represents a prefix that can be identified by a certain position in the SPT (see Section 5.2). Similarly, taking Lee and Durbin's approach, the node index can be used to select the initial value from a precalculated $O(\cdot, n_i)$ table, where n_i is the node's index. The value n_i is a starting point for a CPU to perform Li and Durbin's traversal in B array. That scheme allows the system to adopt the successful BWT method for the CPU to finish up the matching process that is initialized by the short-read alignment processor in FPGA.

5.10. Implementation results

Balcerak *et al.* [139] devoted their work to the problem of DNA short-read alignment. The authors implemented in an FPGA the short-read alignment processor that is presented in this chapter. To develop, programme, run and test their concept, they used the ZedBoard platform [134]. The ZedBoard development kit uses the Xilinx Zynq-7000 All Programmable SoC [135], and it is a cheap System-On-a-Chip hardware platform that is perfect to develop Xilinx's Zynq applications. Zynq, which is all programmable SoC, integrates two ARM Cortex-A9 CPUs with an FPGA structure.

In their experiments, they made the custom coprocessor enhance the CPU's work. The idea of CPU-FPGA co-processing assumes the use of a trie processor for short reads pre-screening. In the trie that is truncated to l_{sat} levels the bottom level nodes represent the group of short reads that share a common prefix of length l_{sat}. The FPGA coprocessor returns the node's index in order to point to the common prefix group in SSL and SPT (see Section 5.3), or in the $O(\cdot, n_i)$ table (see Section 5.9). The CPU follows up the FPGA matching process for the selected subset only.

One can assess the memory requirement of the trie processor as roughly 20 Mb for $TL = 10,000,000$ bp and $L = 9$. That size of BRAM memory is available in today's FPGAs. If the work was distributed in the cluster of FPGA-enabled nodes, it would be possible to process genomes that fit the length of real-life DNA applications. The value of l_{sat} for 10,000,000 gnome base pairs is twelve. Accordingly, an expected number of subsequences that share the common prefix is $4^{12-9} = 64$. Thus, each trie's node of the lowest level represents on average 64 common prefix subsequences. In other words, 64 checkups would have to be done by a cooperating CPU for each FPGA hit.

Balcerak *et al.* implemented the trie of 8 levels in the XC7Z020 FPGA SoC. They experimented with the genome size of 4 kbp. They assumed that l_{sat} was 7 for a given genome length. Consequently, the size of local memory for levels eight and higher was constant. The experiment showed that the CPU-FPGA couple performed 1.7 times faster than the CPU alone. The authors concluded that the implementation of the trie with a depth that is greater than l_{sat} gave no advantage of the CPU-FPGA pair over CPU processing. That is because FPGA matches against the single subsequence effectively when it goes beyond the l_{sat} level, and no advantage of parallel matching of many subsequences occurs. The implementation results of Balcerak *et al.* are given in Table 5.1. The clock frequency for the FPGA was 100 MHz.

Table 5.1. The use of XC7Z020 FPGA resources for the trie processor ($L = 8$, $l_{\text{sat}} = 7$)

Resource type	Utilization
Flip-Flops	5,239 out of 106,400 (4%)
LUTs	5,778 out of 53,200 (10%)
Slices	2,648 out of 13,300 (19%)
BRAMs (RAMB36E1/FIFO36E1)	90 out of 140 (64%)
BRAMs((RAMB18E1/FIFO18E1)	29 out of 280 (10%)

The implementation result of the short-read alignment processor in a large, state-of-the-art FPGA is given in Table 5.2.

Table 5.2. The use of XC6VSX475T resources for the trie processor ($L = 10$, $l_{\text{sat}} = 9$)

Resource type	Utilization
Flip-Flops	552 out of 595,200 (<1%)
LUTs	1,136 out of 297,600 (<1%)
Slices	492 out of 74,400 (<1%)
BRAMs (RAMB36E1/FIFO36E1)	910 out of 1,064 (85%)
BRAMs(RAMB18E1/FIFO18E1)	12 out of 2,128 (<1%)

Table 5.2 regards Xilinx's XC6VSX475T FPGA of the Virtex 6 family. The FPGA is big enough to fit the trie of eleven levels ($L = 10$) and $l_{\text{sat}} = 9$.

5.11. Conclusions

The author proposed the hardware solution to the problem of the inexact short-read alignment. The architecture is derived from a pipelined prefix tree processor that is proposed in Section 3.7. The automatic mismatch generation module (Letter Edit and Match) was added to the original solution to provide the inexact genome matching. Genome short reads are input to the processor and candidate matching positions in the reference genome are its outputs. An implementation of the architecture on the FPGA-enabled SoC platform allowed the test and validation of the concept. Thanks to the pipelined architecture several operations are performed in parallel, and the data movement is minimized. That allows it to improve the processing efficiency if compared with the CPU-only solutions. The combination of an ARM and an FPGA outperformed the CPU-only solution mentioned in the Zynq experiment.

The extent of real genome matching problems is too big to fit a capacity of today's FPGAs. On the other hand, it was shown that parallel processing is available to some breadth of the prefix tree and tree size reduction has its rationale. Only the pre-processing of the input short reads is possible and practical in FPGA. That leads to the conclusion that a complete practical system of the best choice is a processing platform that combines both a CPU and an FPGA. This observation corresponds to the general remark of this work.

The author came up with the BWT-based software algorithm to conveniently integrate the custom alignment processor with an efficient CPU-dedicated solution that is fast and with spare memory. The FPGA provide data screening and reduce the complexity of CPU-bound operations in the proposed scheme. As it was presented, the capacity of the present state-of-the-art FPGA allows it to approach short-read alignment problems of reasonable and practical size.

6. Final remarks

Obstacles in FPGA-based computing

The FPGA devices play an insignificant role in the computing infrastructure today. However, we have analyzed the methods and pointed out the potential advantages of FPGA-accelerated computing for data-dominant problems in this work. We underlined the clear benefits of FPGAs in the context of CPU-central systems, and we denoted obstacles that cause that FPGA technology is still minor in the computer industry. The FPGA and CPU devices share the same semiconductor technology but differ significantly in the design methods and principles of operation.

The discussion provided in this paper made it clear that processors deliver a well-proven and accessible solution to data processing, but FPGAs offer powerful, although still underutilized, technology for computing applications. Unfortunately, at present, the processing architectures that are established by CPU-based solutions cannot be easily replaced by other processing schemes *e.g.* the heterogeneous platforms that include GPGPUs, DSPs, and/or FPGAs.

The unprecedented flexibility of general purpose processors makes them a universal platform that can be used to solve any practical problem quickly. At the moment, the GPPs are the most affordable, cheap, popular, stable, and easy to develop technology that have been present on the market for many years. The position of the traditional sequential computing is also well-established, and multi-CPU processors even have difficulties to make way for quick replacement of the single-CPU applications by their more efficient parallel counterparts. Programmers meet barriers to delivering the scalable concurrent equivalents for the sequential software programs because the preparation of a parallel, multi-threaded application requires substantial efforts, programming skills, and adequate algorithms (the need for the adequate algorithm should be particularly highlighted here). Consequently, the design process is long, difficult, and expensive. Additionally, the final result is often uncertain. The same or even higher barriers apply to GPGPU programming. Those hindrances make multi-core and many-core processing reserved for the class of most demanding high-performance applications.

Just like the multi-core processors and GPGPUs, the FPGAs require parallel programming. Besides, in the case of reconfigurable devices that is the only available option. FPGA applications are developed exclusively using the model of concurrent processing. Furthermore, the parallel programming that is involved in an FPGA design covers very fine grain parallelism (system activities that are planned by a designer must be synchronized at a level of logic signals), but also the parallelism of functional units and processors apply in FPGAs. For that reason, the complexity of a design process is higher in the case of FPGAs than for multi-CPU processors.

At the very beginning of the design process, a designer has to choose correct processor technology for his/her solution. First of all, one considers all options that allow them to meet design constraints. Afterwards, programming simplicity has its direct impact on the selection of the suitable solution and rejection of others. General-purpose processors do not meet design constraints in rare cases of special applications, and only exclusive projects require parallel processing in practice. As a consequence, the most troublesome technologies, including FPGAs, are less popular, and they are seldom in the real-life service. However, the above experience does not imply that there is no space to develop methods for accelerated computing.

FPGA devices aroused interest and high expectations of the computing industry at the very beginning of the 21-st century. Major vendors of high-performance computing systems, like Cray, SGI, and Convey, incorporated custom built FPGA accelerators into their high-performance systems. Later, they found that ready to run applications for the FPGAs hardly existed. The programmer community was not prepared for the new technology, and the costs of the FPGA-based solutions was not compensated for by the adequate rise of a computer's performance. That caused a big disappointment in FPGAs and impacted on the recession in the further development of reconfigurable computing systems. Additionally, GPGPUs rose in popularity. As their offers included appropriate free development tools and application libraries, GPGPUs drew the rest of the community's interest away from the FPGAs.

As it was stated, FPGA technology was not the only possible platform for the custom computing processors. A variety of ASIC technologies is also available, but they do not suit where the programmability of the computing system is required. Although the use of ASIC technology can reduce electric power consumption even more than FPGAs, ASICs exclude system programmability, and this is prohibitive when the general purpose systems are concerned.

The future of FPGA-based computing

The lack of popularity of FPGAs in the computing industry is compensated by their great acceptance by other ICT branches; such as communication networking and real-time processing for example. Reconfigurable technology thrives in these fields.

As a result, the production volume of FPGAs is significant, and it is comparable to those of graphics processors at present. That coincidence is a drive for the further development of FPGA methods, and it helps to promote the popularity of FPGA in computing applications. The migration of FPGAs from the telecommunication and military industries to other branches is an undoubtful fact. Unfortunately, it is an evolution that takes a substantial amount of time.

Today, the topic of FPGA-accelerated computing is advancing in an evolutionary way. Reconfigurable accelerators are more and more tightly integrated with CPUs, successful FPGA applications are reported in the literature, and design tools have been evolving to make FPGA programming easier. Additionally, we are experiencing a kind of hardware unification for reconfigurable technology standards as FPGA accelerators have adopted commodity interfaces; like the PCI-E interface or an ARM's AMBA peripheral bus. The most important advantage is that the new algorithms for custom hardware are developed and proposed by researchers. In that sense, studies and research in the area of FPGA processing are vital to the advancement of computing systems for the future.

The lack of high-level design tools for development in FPGAs is often identified as the main reason for the limited popularity of the custom processor technology. However, in the author's opinion that influence is overestimated. Obviously, the quick design process is a critical circumstance when the processing platform is selected for a new application. Although HLS tools are available for hardware development today, they do not revolutionize the FPGA popularity. In fact, the majority of FPGA projects are still developed as Register Transfer Level (RTL) description because it best suits hardware designers. One can argue that HLS, due to their immaturity, lead to degradation of performance and still more time is necessary for HLLs to catch up with speed and efficiency of RTL-based designs. That argument cannot be accepted for two facts. The first experience is that, according the author's practice, HLLs allow a designer to gain performance that is similar to that of the HDLs, and one can notice a higher resource consumption in HLL-based designs only. The second observation is that high-level languages lead to performance degradation of software applications as well, and that does not interfere with the GPP processor's popularity. The lack of development tools for concurrent programming cannot be raised as an argument of low popularity in the case of multi-core and GPGPU processors because they are widely available.

In the authors' opinion, FPGAs are underutilized due to the lack of algorithms and methods that benefit from fine grain parallelism. For years, von Neuman's sequential machine allows computer scientist to solve information processing problems in the plainest possible way. In consequence, that approach dominates in software programming. The sequential algorithms allow it to work out any processing and computing problem. In contrast, parallelism makes its way to commodity processing

with problems. Simply, parallel processing is a supreme technique that is too expensive for the development of the commodity applications. The custom processors and FPGAs are on the parallel side of computing technology, and one cannot measure their value by their popularity only.

In this work, the author highlighted that data-intensive processing becomes an essential part of today's Internet services. That type of processing is a severe concern of the web search service providers for example. The commodity server solutions are not optimal for browsing and searching tasks, as the data-intensive processing exhibits a computing power imbalance of the commodity computer components. The infrastructure for web services has become a significant part of the ICT market, so new server solutions have become available to better suit that purpose. As we have seen, mobile processors have become a new option for server infrastructure where energy-efficiency is a major constraint. We underlined that the memory wall phenomenon and the IO-bottleneck limited the real-life performance of the CPUs. Today, large-scale processing solutions that are offered on the market for web-services use mobile processors instead of server processors. The necessity of high-performance processing is replaced by the requirement of energy efficiency. FPGA devices fit that trend because the custom processors that are implemented in reconfigurable logic consume an order of magnitude less power than general-purpose processors.

The role of programmable logic devices for browsing and searching had already been noticed earlier, as Finite-State Machine and logic manipulation were pointed as 'dwarves' of the computing by the analytics of Berkeley University. The strength of FPGAs in computing comes from fine grain parallelism. Most successful custom processor architectures are solutions of parallel processing. Parallel processing holds if all processing elements perform without stalls. We have noted that such a condition can be easily met for compute-intensive algorithms. Data-intensive problems in IO-bound systems make it hard to utilize FPGAs in a way that leads to a higher yield in speed compared to CPUs.

At the very beginning of this work, it was stated that there was not enough algorithmic research effort for custom computing processors. Particularly there is a d

eficit of pipeline algorithms. Pipelining is not applicable to multi-core general purpose processors or GPGPUs. It is an exclusive FPGA feature to perform processing in the pipe. Exclusively, FPGA-oriented pipelining algorithms must be developed. It is a particularly important observation as pipelining techniques overcome the IO-bottleneck problem.

The author believes that the use of the FPGA as a CPU co-processor delivers the highest value for custom processor technology. Today, the method of FPGA-accelerated computing has strong support in technology development. We have seen the examples of two applications that were developed for mobile SoC devices. Those SoC chips combined two-core ARM Cortex A9 devices and the Xilinx's FPGA struc-

ture. The tight integration of CPUs and FPGA within a single silicon structure makes constant progress. One can shortly expect 64-bit processors, both Intel's and ARM's, which integrates state-of-the-art reconfigurable structures and CPUs on a single chip. The announcement of such solutions indicates an additional incentive for custom architectures development. Although on-chip integration of CPU and FPGA does not solve the problem of communication throughput, it undoubtedly makes FPGA-accelerated solutions more affordable and reliable. Thus, the significance of proper algorithmic solutions for custom processing grows. The presented solutions of the Hash Binary Tree and short-read alignment custom processors, which were presented in this work, belong to that category.

The exact role that FPGAs play in computing comes with their applicability. According to the author's experience, despite the application area, FPGAs are best suited to data pre-processing. This conclusion applies to the image processing, video compression, network processing, signal processing, *etc.* Accordingly, FPGAs are designated for sorting, searching and formatting in data-mining applications. CPUs are the better choice than FPGAs for control-intensive algorithms of artificial intelligence and machine learning. However, most practically used algorithms need the data pre-processing step, and that makes the CPU+FPGA couple a very flexible solution. A good example of FPGA-accelerated pre-processing was presented in this work. The trie custom processor could enhance the software application of short-read alignment, where an FPGA alone would not be capable of performing the complete, full-size DNA matching process.

Today's computing platforms use the operating system to interact with the user and peripherals and to orchestrate information processing. As an operating system needs a processor, processor-based platforms that are enhanced by FPGA-accelerators seems to be the most rational solution at present. The CPU is necessary for OS to run, and an FPGA helps to improve system performance.

In the summary, we should conclude that custom computing processors are undoubtedly an attractive opportunity for the future of computing. Although, the platform technology for the future custom processors is yet unknown, FPGAs are the basis for custom computing of the present day. FPGAs deliver programmable switching resources for experiments, algorithm development, and real-life solutions. Thanks to FPGA technology we have experienced the progress in parallel algorithms, automatic synthesis, and development tools. Those solutions can be immediately offered to the market and applied to real applications. Custom computing processor technology should be separated from switching technology. The expense of reconfigurable routing resources is too high in FPGAs to allow them to gain more over CPUs, but the future alternatives of reconfigurable semiconductor technology might provide a change to that predicament.

The author's contribution

The elements of the original author's contribution to the topic of FPGA-based data-intensive computing has been scattered throughout the text of this paper. They were put in a wider context, and it might be hard to distinguish and isolate them. Now, the author's contribution will be gathered and emphasised here in this section. The improvements should be recognised in three significant areas. There is an algorithmic and theoretical effort that allows the author to propose the new solutions for FPGA custom computing, and it is the primary and most significant advancement. The second one is the work of designing and implementing the custom processor architectures for sorting and searching. Finally, the third field is constituted out of the experiments that allow the author to compare the performance and efficiency of the proposed solutions to the existing ones.

In this paper, the author presented custom architectures for browsing and searching that gain from pipeline processing. It was demonstrated that pipeline processing overcomes the ubiquitous IO-bottleneck and memory wall. We have found how to maximize the length of the pipe to perform more instructions in a single clock cycle. Solutions that have been presented in this work avoided control-intensive algorithms because they disrupt the powerful concept of pipelining. It was shown how the inner loop body could be pipelined and converted into the hardware structure.

It was stated in Section 3.2 that author developed the tool for automatic generation of the parametrised architectures for bitonic sorting networks in VHDL [101]. This approach allowed the author to verify the performance and estimate the amount of resources that were necessary to implement that kind of a sorting solution in FPGAs. Both parallel and sequential input-output methods were examined. In the work [101], an important observation was derived which highlighted a limitation for practical use of sorting networks in real-life systems. The IO bandwidth bottleneck caused that problem.

The custom processor for merge sorting was discussed in Section 3.3. The corresponding structure was implemented and examined by author on SGI's RASC accelerating platform [110]. The custom architecture was developed in the Mitrion-C HLS language (that was a state-of-the-art solution at that time) to verify the emerging methods for FPGA development. The conclusions were taken that the FPGA accelerator should act as a co-processor for a CPU in real-life sorting applications. The algorithm that combined an efficient software-based sorting with the FPGA co-processor was benchmarked. Explicit caching was necessary to enhance processing in the case of the merge sort architecture. This conclusion was a spark to the idea of Sequential-Access Buffering (see Section 2.5.3).

The Bloom filter is the essential and most efficient browsing and searching tool for data processing acceleration. The work by Jamro *et al.* [122] that was mentioned

in Section 3.4 has been conducted by the team with the author's contribution. The collective group effort delivered the multi-fold parallel Bloom filter architecture which featured unprecedented data throughput on the accelerator implemented on SGI's RASC platform. Exclusively, the author developed the formulas for a probability of the false-positive for a parallel version of the Bloom filter. The detailed discussion can be found in [122]. We have seen the significant role of the internal dedicated Block RAM memory in the design for FPGAs. The authors demonstrated that the number of BRAM resources was the only limitation for the Bloom filter implementations in FPGA devices.

Another author's work [60], also mentioned in Section 3.4, takes advantages of Bloom filtering in the SoC environment. The paper demonstrated the application of the anti-virus system that was implemented in the FPGA-enabled SoC. The experiment that was conducted by the author showed that an energy-efficient processor, when supported by an FPGA accelerator, is capable of gaining an efficiency of the server's CPU for the data-intensive problem. The meaningful experience was taken as the CPU&FPGA solution consumed an order of magnitude less energy in this attempt.

Another novelty that was introduced by the author is the Hash Binary Tree that works as the Bloom filter and the hash index simultaneously. It offers the solution that is fast, accurate and spares hardware memory resources. Accordingly, this author's remarkable achievement is presented in Chapter 4. The solution is the consequence of the original concept of the custom binary tree processor that was presented in Section 3.6.1. In that section, the author gave his BRAM-based architecture of the binary tree processor and proposed the algorithm to restore the matching pattern index (see Section 3.6.2). This smart index restoring method allows the binary tree to work as the fast index table. As it was proved by the author in Section 4.3, for a few patterns the HBT offers memory savings when compared to the index table and the Bloom filter.

The author proposed the modified Cuckoo hashing scheme (see Section 3.4.3) that eliminates the problem of pattern verification completely if the dictionary is well-defined. Also, it reduces a check to a single dictionary comparison if the analyzed data is allowed to contain a random pattern. The original Cuckoo method offered deterministic verification time only. The modified Cuckoo hashing scheme works well with the HBT, which was presented in Section 4.4.

Chapter 5 presents the original authors contribution. Together with Balcerak *et al.* [139], the author worked on the problem of short-read alignment. The author devised the appropriate pipelined architectures for the custom processor and proposed the solution to fuse the FPGA accelerator with the Burrow-Wheeler Transform-based matching algorithm (see Section 5.10). The algorithm conveniently integrates the custom alignment processor with an efficient CPU-dedicated solution that is fast and

with spare memory. The FPGA provide data screening and reduces the complexity of the CPU-bound operations in the proposed scheme. The performed experiments showed that an FPGA can provide acceleration of short-read alignment. Also, the author's further analysis, which was presented in the chapter, leads one to believe that state-of-the-art FPGAs provide enough resources to enhance genome processing.

Further work

The works presented in this paper is a result of the author's interest in customized computing architectures. These structures enhance calculations and information processing in practical applications. At present, the choice of FPGA technology, as the most flexible and affordable for the purpose, is straightforward. For years, FPGAs and CPUs constituted separate design domains, and their integration was a very challenging design task. These difficulties made the practical hybrid solutions of CPUs and FPGAs rare. Today, with the help of new EDA tools and reprogrammable SoC devices, co-design in the software and hardware domain is becoming easier and constantly more popular.

Custom architectures that were presented in this dissertation were implemented and optimized in the FPGA domain mainly. An additional focus is necessary to provide seamless integration of custom architectures with CPUs to allow practical benefits. For example, the sorting and searching solutions should become a part of the libraries that encapsulate the hardware functionality. Preferably, the library interface should be compatible with the commonly recognized standards (like C++ <algorithm> for example). The proposed hardware architectures often require a redesign of the existing software solutions. This is the case of the custom trie processor and BWT-based short-read alignment algorithm. The adjustments of the existing software have to be done to make essential use of the new ideas. The effort that is required to perform the necessary integration of the hardware and software will be, in the author's opinion, significant and time-consuming.

Other promising algorithms for the FPGA-based processing exist in the data-intensive processing domain. Text similarity discovery is another research area of the group the author works with. As text similarity calculation is a useful tool for many practical large-scale applications, it has become a goal to port a part of the exhaustive processing to an FPGA. The paper of Karwatowski *et al.* [98], from the author's research group, presents preliminary results of the cosine-similarity-measure processor for text classification. Remarkably, the similarity processor powerfully implements the solutions described in this monograph *i.e.* deep pipeling and execution of parallel tasks for the single input data stream.

At this time, the MapReduce model is a standard for data processing of large datasets in distributed databases. Hadoop is the first and most popular framework

for the MapReduce calculations. For the problem of the text similarity calculations, it is worth to combine the two: an FPGA accelerator and a MapReduce distributed processing model. Russek *et al.* [140] present preliminary results of the effort to develop an energy-efficient cluster for text search and comparison applications. Hadoop classes to delegate data execution to the hardware accelerator are presented in the paper. The authors' long-term goal is to introduce an alternative FPGA-enhanced solution that enables green computing in the infrastructure for a search in the Internet or huge repository.

Bibliography

[1] Lin M., El Gamal A., Lu Y.-C., Wong S.: *Performance benefits of monolithically stacked 3-D FPGA*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 26, no. 2, 2007, 216–229

[2] Oldfield J. V., Dorf R. C.: *Field-programmable gate arrays*. John Wiley & Sons, 1995

[3] Trimberger S.: *Field-programmable gate array technology*. Springer Science & Business Media, 1994

[4] Salcic Z., Smailagic A.: *Digital systems design and prototyping: using field programmable logic and hardware description languages*. Springer Science & Business Media, 2000

[5] Villasenor J., Mangione-Smith W. H.: *Configurable computing*. Scientific American, vol. 276, no. 6, 1997, 54–9

[6] Horta E. L., Lockwood J. W., Taylor D. E., Parlour D.: *Dynamic hardware plugins in an FPGA with partial run-time reconfiguration*. [In:] *Proceedings of the 39th annual Design Automation Conference*. ACM, 2002, 343–348

[7] Oliver T., Schmidt B., Maskell D.: *Hyper customized processors for bio-sequence database scanning on FPGAs*. [In:] *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. ACM, 2005, 229–237

[8] Ditmar J., Torkelsson K., Jantsch A.: *A dynamically reconfigurable FPGA-based content addressable memory for Internet protocol characterization*. [In:] *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*. Vol. 1896, Lecture Notes in Computer Science, Springer, 2000, 19–28

[9] Divyasree J., Rajashekar H., Varghese K.: *Dynamically reconfigurable regular expression matching architecture*. [In:] *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*. IEEE, 2008, 120–125

[10] Ruta A., Brzoza-Woch R., Zielinski K.: *On fast development of FPGA-based SOA services-machine vision case study*. Design Automation for Embedded Systems, vol. 16, no. 1, 2012, 45–69

[11] Amos D., Lesea A., Richter R.: *FPGA-Based Prototyping Methodology Manual*. Happy About, 2011

[12] Kuon I., Rose J.: *Measuring the gap between FPGAs and ASICs*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 26, no. 2, 2007, 203–215

[13] Zahiri B.: *Structured ASICs: opportunities and challenges*. [In:] *Computer Design, 2003. Proceedings. 21st International Conference on*. IEEE, 2003, 404–409

[14] Rodriguez-Andina J. J., Moure M. J., Valdes M. D.: *Features, design tools, and application domains of FPGAs*. Industrial Electronics, IEEE Transactions on, vol. 54, no. 4, 2007, 1810–1823

[15] Darema F.: *The SPMD Model: Past, Present and Future*. [In:] *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Vol. 2131, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, 1–1

[16] Gara A., Blumrich M. A., Chen D., Chiu G.-T., Coteus P., Giampapa M. E., Haring R. A., Heidelberger P., Hoenicke D., Kopcsay G. V. et al.: *Overview of the Blue Gene/L system architecture*. IBM Journal of Research and Development, vol. 49, no. 2.3, 2005, 195–212

[17] Hennessy J. L., Patterson D. A.: *Computer architecture: a quantitative approach*. Elsevier, 2012

[18] Akhter S., Roberts J.: *Multi-core programming*. Vol. 33, Intel Press Hillsboro, 2006

[19] Herlihy M., Shavit N.: *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012

[20] Asanovic K., Bodik R., Catanzaro B. C., Gebis J. J., Husbands P., Keutzer K., Patterson D. A., Plishker W. L., Shalf J., Williams S. W., Yelick K. A.: *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical report, EECS Department, University of California, Berkeley, Dec 2006

[21] Gepner P., Kowalik M. F.: *Multi-core processors: New way to achieve high system performance*. [In:] *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*. IEEE, 2006, 9–13

[22] Mahapatra N. R., Venkatrao B.: *The processor-memory bottleneck: problems and solutions*. Crossroads, vol. 5, no. 3es, 1999, 2

[23] Borkar S.: *Design challenges of technology scaling*. Micro, IEEE, vol. 19, no. 4, 1999, 23–29

[24] Amdahl G. M.: *Validity of the single processor approach to achieving large scale computing capabilities*. [In:] *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, 483–485

[25] Gustafson J. L.: *Reevaluating Amdahl's law*. Communications of the ACM, vol. 31, no. 5, 1988, 532–533

[26] Deng Y., Zhang P., Marques C., Powell R., Zhang L.: *Analysis of Linpack and power efficiencies of the world's TOP500 supercomputers*. Parallel Computing, vol. 39, no. 6, 2013, 271–279

[27] Morse H. S.: *Practical parallel computing*. Academic Press, 2014

[28] Loshin D.: *High Performance Computing Demystified*. Academic Press, 2014

[29] Kitowski J., Turała M., Wiatr K., Dutka Ł.: *PL-Grid: foundations and perspectives of national computing infrastructure*. [In:] *Building a National Distributed e-Infrastructure–PL-Grid*. Vol. 7136, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, 1–14

[30] Ultsch A.: *Proof of Pareto's 80/20 Law and Precise Limits for ABC-Analysis*. Technical Report 2002/c, DataBionics Reseach Group, University of Marburg, 2002

[31] Kuna D., Jamro E., Russek P., Wiatr K.: *Using standard hardware accelerators to decrease computation times in scientific applications*. Computer Science, vol. 10, 2009, 65–74

[32] Pietroń M., Russek P., Wiatr K.: *Accelerating Select where and select join queries on a GPU*. Computer Science, vol. 14, no. 2, 2013, 243

[33] Dąbrowska-Boruch A., Jamro E., Janiszewski M., Kuna D., Machaczek K., Russek P., Wiatr K., Wielgosz M.: *Utilization of FPGA Architectures for High Performance Computations*. Computational Methods in Science and Technology, 2010, 63–69

[34] Jamro E., Janiszewski M., Machaczek K., Russek P., Wiatr K., Wielgosz M.: *Computation acceleration on SGI RASC: FPGA based reconfigurable computing hardware*. Computer Science, vol. 9, 2008, 21–34

[35] Gielata A., Russek P., Wiatr K.: *AES hardware implementation in FPGA for algorithm acceleration purpose*. [In:] *Signals and Electronic Systems, 2008. ICSES'08. International Conference on*. IEEE, 2008, 137–140

[36] Russek P., Wiatr K.: *The prospect of computing acceleration using reconfigurable logic technology in huge computational power systems*. [In:] *Proceedings of the IFAC Workshop on Programable Devices and Embedded Systems, PDeS 2006, Brno, Czech Republic*. 2006, 44–49

[37] Russek P., Wiatr K.: *Dedicated architecture for double precision matrix multiplication in supercomputing environment*. [In:] *Design and Diagnostics of Electronic Circuits and Systems, 2007. DDECS'07. IEEE*. IEEE, 2007, 1–4

[38] Wielgosz M., Pietroń M., Jamro E., Russek P., Wiatr K.: *Two electron integrals calculation accelerated with double precision exp () hardware module*. Reconfigurable Systems Summer Institute, RSSI proceedings, 2007

[39] Wielgosz M., Mazur G., Makowski M., Jamro E., Russek P., Wiatr K.: *Analysis of the Basic Implementation Aspects of Hardware-Accelerated Density Functional Theory Calculations*. Computing and Informatics, vol. 29, no. 6, 2012, 989–1000

[40] Wielgosz M., Jamro E., Żurek D., Wiatr K.: *FPGA Implementation of the Selected Parts of the Fast Image Segmentation*. [In:] *Intelligent Tools for Building a Scientific Information Platform*. Springer, 2012, 203–216

[41] Asanovic K., Bodik R., Demmel J., Keaveny T., Keutzer K., Kubiatowicz J., Morgan N., Patterson D., Sen K., Wawrzynek J. et al.: *A view of the parallel computing landscape*. Communications of the ACM, vol. 52, no. 10, 2009, 56–67

[42] Pike R. *Notes on Programming in C*. `http://doc.cat-v.org/bell_labs/pikestyle/`, 1989. Accessed: 2015-09-03

[43] Wiatr K., Russek P.: *Embedded zero wavelet coefficient coding method for FPGA implementation of video codec in real-time systems*. [In:] *Information Technology: Coding and Computing, 2000. Proceedings. International Conference on*. IEEE, 2000, 146–151

[44] Dąbrowska A., Wiatr K.: *Modyfikacja algorytmu E3SS estymacji ruchu na potrzeby implementacji w układach FPGA*. Automatyka/Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, vol. 10, 2006, 345–353

[45] Han J., Kamber M., Pei J.: *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 3rd edition, 2011

[46] Gray J., Coates J., Nyberg C.: *Performance/price sort and pennysort*. Technical report, Technical Report MS-TR-98-45, Microsoft, 1998

[47] Nyberg C., Shah M., Govindaraju N. *Sort benchmark home page*. `http://sortbenchmark.org`. Accessed: 2015-09-03

[48] Mueller R., Teubner J.: *FPGA: what's in it for a database?* [In:] *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, 999–1004

[49] Putnam A., Caulfield A. M., Chung E. S., Chiou D., Constantinides K., Demme J., Esmaeilzadeh H., Fowers J., Gopal G. P., Gray J. et al.: *A reconfigurable fabric for accelerating large scale datacenter services*. [In:] *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, 13–24

[50] Yan J., Zhao Z.-X., Xu N.-Y., Jin X., Zhang L.-T., Hsu F.-H.: *Efficient Query Processing for Web Search Engine with FPGAs*. [In:] *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, 97–100

[51] Leber C., Geib B., Litz H.: *High frequency trading acceleration using FPGAs*. [In:] *Field Programmable Logic and Applications (FPL), 2011 International Conference on*. IEEE, 2011, 317–322

[52] Halstead R. J., Sukhwani B., Min H., Thoennes M., Dube P., Asaad S., Iyer B.: *Accelerating join operation for relational databases with FPGAs*. [In:] *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE, 2013, 17–20

[53] Sidhu R., Prasanna V. K.: *Fast regular expression matching using FPGAs*. [In:] *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 2001, 227–238

[54] Sourdis I., Bispo J., Cardoso J. M., Vassiliadis S.: *Regular expression matching in reconfigurable hardware*. Journal of Signal Processing Systems, vol. 51, no. 1, 2008, 99–121

[55] Kumar S., Dharmapurikar S., Yu F., Crowley P., Turner J.: *Algorithms to accelerate multiple regular expressions matching for deep packet inspection*. ACM SIGCOMM Computer Communication Review, vol. 36, no. 4, 2006, 339–350

[56] Bispo J., Cardoso J. M.: *Synthesis of regular expressions for FPGAs*. International Journal of Electronics, vol. 95, no. 7, 2008, 685–704

[57] Hutchings B. L., Franklin R., Carver D.: *Assisting network intrusion detection with reconfigurable hardware*. [In:] *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*. IEEE, 2002, 111–120

[58] Ranganathan P.: *The new (system) balance of power and opportunities for optimizations*. [In:] *Proceedings of the 2014 international symposium on Low power electronics and design*. ACM, 2014, 331–332

[59] Russek P., Wiatr K.: *FPGA-accelerated algorithm for the regular expression matching system*. International Journal of Electronics, vol. 102, no. 1, 2015, 71–88

[60] Russek P., Wiatr K.: *The Regular Expression Matching Algorithm for the Energy Efficient Reconfigurable SoC*. [In:] *Parallel Processing and Applied Mathematics*. Springer, 2014, 545–556

[61] Wang C., Li X., Zhou X., Chen Y., Cheung R. C.: *Big data genome sequencing on zynq based clusters*. [In:] *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM, 2014, 247–247

[62] Lin Z., Chow P.: *ZCluster: A Zynq-based Hadoop cluster*. [In:] *Field-Programmable Technology (FPT), 2013 International Conference on*. IEEE, 2013, 450–453

[63] Jamieson P., Luk W., Wilton S. J., Constantinides G. et al.: *An energy and power consumption analysis of FPGA routing architectures*. [In:] *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009, 324–327

[64] Altera Corp. *Reducing Power Consumption and Increasing Bandwidth on 28-nm FPGAs*. `http://www.altera.com`, 2012. Accessed: 2015-09-03

[65] Shang L., Kaviani A. S., Bathala K.: *Dynamic power consumption in Virtex-II FPGA family*. [In:] *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*. ACM, 2002, 157–164

[66] Intel Corp. *Power Management in Intel Architecture Servers*. `http://download.intel.com`, 2010. Accessed: 2015-09-03

[67] Xilinx. *Xilinx Power Estimator*. `http://www.xilinx.com`. Accessed: 2015-04-25

[68] Viswanath R., Wakharkar V., Watwe A., Lebonheur V. et al. *Thermal performance challenges from silicon to systems*. `http://mprc.pku.edu.cn`, 2000. Accessed: 2015-09-03

[69] DRC Corp. *DRC Accelium Coprocessors Datasheet*. `http://drccomputer.com`, 2014. Accessed: 2015-04-25

[70] Cichoń S., Gorgoń M.: *FPGA-based dvcpro hd decoder implementation using impulse C*. Computer Science, vol. 14, no. 4, 2013, 531–546

[71] Bakos J. D.: *High-performance heterogeneous computing with the Convey HC-1*. Computing in Science & Engineering, vol. 12, no. 6, 2010, 80–87

[72] Augustin W., Heuveline V., Weiß J.-P.: *Convey HC-1 Hybrid Core Computer: The Potential of FPGAs in Numerical Simulation*. KIT, 2010

[73] Silicon Graphics. *Reconfigurable Application-Specific Computing User's Guide*. `http://techpubs.sgi.com`. Accessed: 2015-07-28

[74] Wielgosz M., Jamro E., Wiatr K.: *Accelerating calculations on the RASC platform: A case study of the exponential function*. [In:] *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2009, 306–311

[75] Vahid F.: *Digital Design with RTL Design, VHDL and Verilog VHDL*. John Wiley & Sons, 2010

[76] Gajski D. D., Kleinsmith J.: *Principles of digital design*. Vol. 42, Prentice Hall New York, 1997

[77] Micheli G. D.: *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994

[78] Gajski D. D., Ramachandran L.: *Introduction to high-level synthesis*. Design & Test of Computers, IEEE, vol. 11, no. 4, 1994, 44–54

[79] Baranov S. I., Tehnikaülikool T.: *Logic and system design of digital systems*. TUT Press Tallinn, 2008

[80] Hafer L. J., Parker A. C.: *A formal method for the specification, analysis, and design of register-transfer level digital logic*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 2, no. 1, 1983, 4–18

[81] Leive G., Thomas D. E.: *A technology relative Logic Synthesis and Module Selection system*. [In:] *Proceedings of the 18th Design Automation Conference*. IEEE Press, 1981, 479–485

[82] Fernandez E. B., Lang T.: *Scheduling as a graph transformation*. IBM Journal of Research and Development, vol. 20, no. 6, 1976, 551–559

[83] Fisher J. A.: *Trace scheduling: A technique for global microcode compaction*. IEEE Transactions on Computers, vol. 30, no. 7, 1981, 478–490

[84] Hafer L. J., Parker A. C.: *Automated synthesis of digital hardware*. Computers, IEEE Transactions on, vol. 100, no. 2, 1982, 93–109

[85] Hwang C.-T., Lee J.-H., Hsu Y.-C.: *A formal approach to the scheduling problem in high level synthesis*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 10, no. 4, 1991, 464–475

[86] Randy A., Kennedy K.: *Optimizing Compilers for Modern Architectures*. Morgan Kauffman, 2001

[87] Pietroń M., Russek P., Wiatr K.: *Loop profiling tool for HPC code inspection as an efficient method of FPGA based acceleration*. International Journal of Applied Mathematics and Computer Science, vol. 20, no. 3, 2010, 581–589

[88] Fingeroff M.: *High-level synthesis blue book*. Xlibris Corporation, 2010

[89] Lee T.-F., Wu A.-H., Gajski D. D., Lin Y.-L.: *An effective methodology for functional pipelining*. [In:] *Computer-Aided Design, 1992. ICCAD-92. Digest of Technical Papers., 1992 IEEE/ACM International Conference on*. IEEE, 1992, 230–233

[90] Jeon J., Choi K.: *Loop pipelining in hardware-software partitioning*. [In:] *Design Automation Conference 1998. Proceedings of the ASP-DAC'98. Asia and South Pacific*. IEEE, 1998, 361–366

[91] Chao L.-F., LaPaugh A. S., Sha E.-M.: *Rotation scheduling: A loop pipelining algorithm*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 16, no. 3, 1997, 229–239

[92] Wielgosz M., Jamro E., Wiatr K.: *Highly efficient structure of 64-bit exponential function implemented in FPGAs*. [In:] *Reconfigurable Computing: Architectures, Tools and Applications*. Vol. 4943, Lecture Notes in Computer Science, Springer, 2008, 274–279

[93] Kavi K. M., Buckles B. P., Bhat U. N.: *A formal definition of data flow graph models*. Computers, IEEE Transactions on, vol. 100, no. 11, 1986, 940–948

[94] De Jong G. G.: *Data flow graphs: system specification with the most unrestricted semantics*. [In:] *Proceedings of the conference on European design automation*. IEEE Computer Society Press, 1991, 401–405

[95] Amellal S., Kaminska B.: *Scheduling of a control data flow graph*. [In:] *Circuits and Systems, 1993., ISCAS'93, 1993 IEEE International Symposium on*. IEEE, 1993, 1666–1669

[96] Amellal S., Kaminska B.: *Functional synthesis of digital systems with TASS*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 13, no. 5, 1994, 537–552

[97] Wu Q., Wang Y., Bian J., Wu W., Xue H.: *A hierarchical CDFG as intermediate representation for hardware/software codesign*. [In:] *Communications, Circuits and Systems and West Sino Expositions, IEEE 2002 International Conference on*. Vol. 2. IEEE, 2002, 1429–1432

[98] Karwatowski M., Koryciak S., Jamro E., Dąbrowska-Boruch A., Wiatr K.: *Cosine similarity metric calculation on low power heterogeneous computing platform*. [In:] *KU KDM 2015 : eigth ACC Cyfronet AGH users' conference : Zakopane, 11-13 Mar 2015*. 2015, 111–112

[99] Mueller R., Teubner J., Alonso G.: *Data processing on FPGAs*. Proceedings of the VLDB Endowment, vol. 2, no. 1, 2009, 910–921

[100] Cormen T., Leiserson C., Rivest R., Stein C.: *Introduction to Algorithms MIT Press*. Cambridge, MA, 2003

[101] Russek P., Wiatr K.: *Hardware acceleration of sorting algorithms using reconfiguration technics*. [In:] *IFAC : proceedings of IFAC workshop on Programmable Devices and Systems PDS 2004 : international conference : Cracow, November 18th-19th, 2004*. 2004

[102] Knuth D. E.: *The art of computer programming: sorting and searching*. Vol. 3, Pearson Education, 1998

[103] Batcher K. E.: *Sorting networks and their applications*. [In:] *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, 307–314

[104] Ajtai M., Komlós J., Szemerédi E.: *Sorting in clogn parallel steps*. Combinatorica, vol. 3, no. 1, 1983, 1–19

[105] Zhang Y., Zheng S.: *An efficient parallel VLSI sorting architecture*. VLSI Design, vol. 11, no. 2, 2000, 137–147

[106] Huang C.-Y., Yu G.-J., Liu B.-D.: *A hardware design approach for merge-sorting network*. [In:] *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*. Vol. 4. IEEE, 2001, 534–537

[107] Greß A., Zachmann G.: *GPU-ABiSort: Optimal parallel sorting on stream architectures*. [In:] *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, 45–55

[108] Mueller R., Teubner J., Alonso G.: *Sorting networks on FPGAs*. The VLDB Journal-The International Journal on Very Large Data Bases, vol. 21, no. 1, 2012, 1–23

[109] Peters H., Schulz-Hildebrandt O., Luttenberger N.: *Fast in-place sorting with cuda based on bitonic sort*. [In:] *Parallel Processing and Applied Mathematics*. Vol. 6067, Lecture Notes in Computer Science, Springer, 2010, 403–410

[110] Russek P., Wiatr K.: *The enhancement of a computer system for sorting capabilities using FPGA custom architecture*. Computing and Informatics, vol. 32, no. 4, 2014, 859–876

[111] Mohl S.: *The Mitrion-C programming language*. Mitrionics Inc, 2006

[112] Koch D., Torresen J.: *FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting*. [In:] *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, 45–54

[113] Harkins J., El-Ghazawi T., El-Araby E., Huang M.: *Performance of sorting algorithms on the SRC 6 reconfigurable computer*. [In:] *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 2005, 295–296

[114] Marcelino R., Neto H., Cardoso J. M.: *Sorting units for FPGA-based embedded systems*. [In:] *Distributed Embedded Systems: Design, Middleware and Resources*. Vol. 271, IFIP-The International Federation for Information Processing, Springer, 2008, 11–22

[115] Bloom B. H.: *Space/time trade-offs in hash coding with allowable errors*. Communications of the ACM, vol. 13, no. 7, 1970, 422–426

[116] Peterson W. W., Brown D. T.: *Cyclic codes for error detection*. [In:] *Proceedings of the IRE*. 1961, 228–235

[117] Dharmapurikar S., Krishnamurthy P., Sproull T., Lockwood J.: *Deep packet inspection using parallel bloom filters*. [In:] *High performance interconnects, 2003. proceedings. 11th symposium on*. IEEE, 2003, 44–51

[118] Suresh D. C., Guo Z., Buyukkurt B., Najjar W. A.: *Automatic compilation framework for bloom filter based intrusion detection*. [In:] *Reconfigurable Computing: Architectures and Applications*. Springer, 2006, 413–418

[119] Maccari L., Fantacci R., Neira P., Gasca R. M.: *Mesh network firewalling with bloom filters*. [In:] *Communications, 2007. ICC'07. IEEE International Conference on*. IEEE, 2007, 1546–1551

[120] Jain N.: *Using Bloom filters to refine web search results*. [In:] *Proc. 7th WebDB*. 2005, 25–30

[121] Raykova M., Vo B., Bellovin S. M., Malkin T.: *Secure anonymous database search*. [In:] *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM, 2009, 115–126

[122] Jamro E., Russek P., Dąbrowska-Boruch A., Wielgosz M., Wiatr K.: *The implementation of the customized, parallel architecture for a fast word-match program*. International Journal of Computer Systems Science & Engineering, vol. 26, no. 4, 2011, 285–292

[123] Dharmapurikar S., Krishnamurthy P., Sproull T. S., Lockwood J. W.: *Deep Packet Inspection using Parallel Bloom Filters*. IEEE Micro, vol. 24, no. 1, 2004, 52–61

[124] Jacob A., Gokhale M.: *Language classification using n-grams accelerated by FPGA-based Bloom filters*. [In:] *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*. ACM, 2007, 31–37

[125] Sourdis I., Pnevmatikatos D., Wong S., Vassiliadis S.: *A reconfigurable perfect-hashing scheme for packet inspection*. [In:] *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, 644–647

[126] Papadopoulos G., Pnevmatikatos D.: *Hashing+ memory= low cost, exact pattern matching*. [In:] *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, 39–44

[127] Devroye L., Morin P.: *Cuckoo hashing: further analysis*. Information Processing Letters, vol. 86, no. 4, 2003, 215–219

[128] Thinh T. N., Kittitornkun S., Tomiyama S.: *Applying cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS*. [In:] *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*. IEEE, 2007, 121–128

[129] Le H., Prasanna V. K.: *A memory-efficient and modular approach for string matching on fpgas*. [In:] *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010, 193–200

[130] Katz R. H., Borriello G.: *Contemporary logic design*. Pearson Prentice Hall, 2005

[131] Aho A. V., Corasick M. J.: *Efficient string matching: an aid to bibliographic search*. Communications of the ACM, vol. 18, no. 6, 1975, 333–340

[132] Tan L., Sherwood T.: *A high throughput string matching architecture for intrusion detection and prevention*. [In:] *ACM SIGARCH Computer Architecture News*. Vol. 33. IEEE Computer Society, 2005, 112–122

[133] Jung H.-J., Baker Z. K., Prasanna V. K.: *Performance of FPGA implementation of bit-split architecture for intrusion detection systems*. [In:] *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, 8–pp

[134] Avnet. *ZedBoard Hardware User's Guide*. `http://www.zedboard.org`. Accessed: 2015-09-03

[135] Xilinx. *Zynq-7000 All Programmable SoC Overview*. `http://www.xilinx.com`. Accessed: 2015-05-07

[136] Xilinx. *LogiCORE IP AXI DMA v6.03a. Product Guide*. `http://www.xilinx.com`. Accessed: 2015-09-03

[137] Metzker M. L.: *Sequencing technologies – the next generation*. Nature Reviews Genetics, vol. 11, no. 1, 2010, 31–46

[138] Li H., Durbin R.: *Fast and accurate short read alignment with Burrows–Wheeler transform*. Bioinformatics, vol. 25, no. 14, 2009, 1754–1760

[139] Balcerak M., Strzebak T., Russek P., Koryciak S., Wiatr K.: *Pattern Searching Scheme Using CPU & FPGA*. [In:] *Cracow Grid Workshop 2015 (CGW 15)*. Vol. 14, 2015, 59–60

[140] Russek P., Karwatowski M., Wielgosz M., Frączek R., Wiatr K.: *Documents similarity calculation in the low-power cluster*. [In:] *KU KDM 2015 : eigth ACC Cyfronet AGH users' conference : Zakopane, 11-13 Mar 2015*. 2015, 37–38