ulm university universität
uulm

A Coprocessor for Fast Searching in Large Databases: Associative Computing Engine

DISSERTATION

submitted in partial fulfillment of
the requirements for the degree of

Doktor-Ingenieur (Dr.-Ing.)

to the Faculty of Engineering Science
and Informatics of the University of Ulm

by

Christophe Layer
from Dijon

First Examiner:	Prof. Dr.-Ing. H.-J. Pfleiderer
Second Examiner:	Prof. Dr.-Ing. T. G. Noll
Acting Dean:	Prof. Dr. rer. nat. H. Partsch
Examination Date:	June 12, 2007

Abstract

As a matter of fact, information retrieval has changed considerably in recent years with the expansion of the Internet and the advent of always more modern and inexpensive mass storage devices. Due to the enormous increase in stored digital contents, multimedia devices must provide effective and intelligent search functionalities. However, in order to retrieve a few relevant kilobytes from a large digital store, one moves up to hundreds of gigabytes of data between memory and processor over a bandwidth-restricted bus. The only long-term solution is to delegate this task to the storage medium itself.

For that purpose, this thesis describes the development and prototyping of an associative search coprocessor called ACE: an Associative Computing Engine dedicated to the retrieval of digital information by means of approximate matching procedures. In this work, the two most important features are the overall scalability of the design allowing the support of different bit widths and module depths along the data path, as well as the obtainment of the minimum hardware area while ensuring the maximum throughput of the global architecture.

After simulation, the correctness of the results delivered by the FPGA prototyping board could be verified using a text encoding scheme based on hashing theory. The performance of the hardware platform has been compared with the same application running in software on a general purpose processor. Accordingly, a speed-up of three to four orders of magnitude is achieved for the FPGA system versus many recent personal computers with different internal hardware configurations.

Acknowledgments

First and foremost, I express my gratitude to Prof. H.-J. Pfleiderer for his constant support, motivation and encouragements. Thank you for welcoming me in the Institute of Microelectronics at the University of Ulm and giving me the opportunity to pursue my research interests.

Furthermore, I am very grateful to Prof. T. G. Noll of the RWTH Aachen for his interest in my work and his agreement to report about it. Thank you for the extremely helpful feedback, for the suggestions about the completion of the manuscript and for advising on this dissertation.

I would also like to thank Prof. P. Ruján for his continuous encouragements and for pursuing my work over further international projects. Dr. G. Lapir is gratefully acknowledged for the discussions at an earlier stage of this project and for providing information and materials about the associative search method.

I would like to thank my colleagues from the Institute of Microelectronics, especially M. Bschorr, C. Günter, O. Pfänder, J. Rauscher, W. Schlecker, K. Schmidt, R. Schreier and E. Schubert for all the constructive discussions and comments, as well as for the very comfortable atmosphere at work. Nonetheless, I am indebted to G. Kirilov for his expertise and contribution to the synthesis of the memory controller.

Finally, special thanks go to Mrs. Höfer for her kindness, help and support.

Ulm, October 18, 2007 *Christophe Layer*

Table of Contents

List of Figures

List of Tables

Chapter I

Introduction

WITHIN the scope of this work, we aim to demonstrate how a complex problem, e.g., retrieving information rapidly, can be efficiently solved using various well-established techniques as well as modern design methodologies. Our research has focused on the improvement of different algorithms and new computing methods that yielded highly significant results, as it will be presented in this document. But first of all, this chapter is dedicated to a global introduction to the concept of information retrieval, targeting the implementation of a generic text database system. Motivated by the noticeably long searching time needed by software applications even on the most recent computers, we foresee the necessity to move to hardware implementation and expose the envisioned development. Hence the detailed description of the organization of this thesis is depicted in the last section, where we give an overview of the completed research work.

1 Information Retrieval Systems

Fundamentally, Information Retrieval (IR) deals with the storage, the organization of and the access to some items, providing users an easy way to the information they are interested in. As a part of an IR system, data retrieval consists mainly in determining which documents of a collection contain the keywords of a query which is in this context a quite complex process based on a trade-off between speed and accuracy. As the algorithmic complexity is mastered by always more intelligent software programs, speed is basically the problem we want to address at the electronic hardware level in this thesis: How can we accelerate the process of retrieving information in a huge database system?

According to Baeza-Yates and Ribeiro-Neto [Bae99], three fundamental changes have occurred due to the advances in modern computer technology and the boom of the Internet. First it became a lot cheaper to have access to various sources of information,

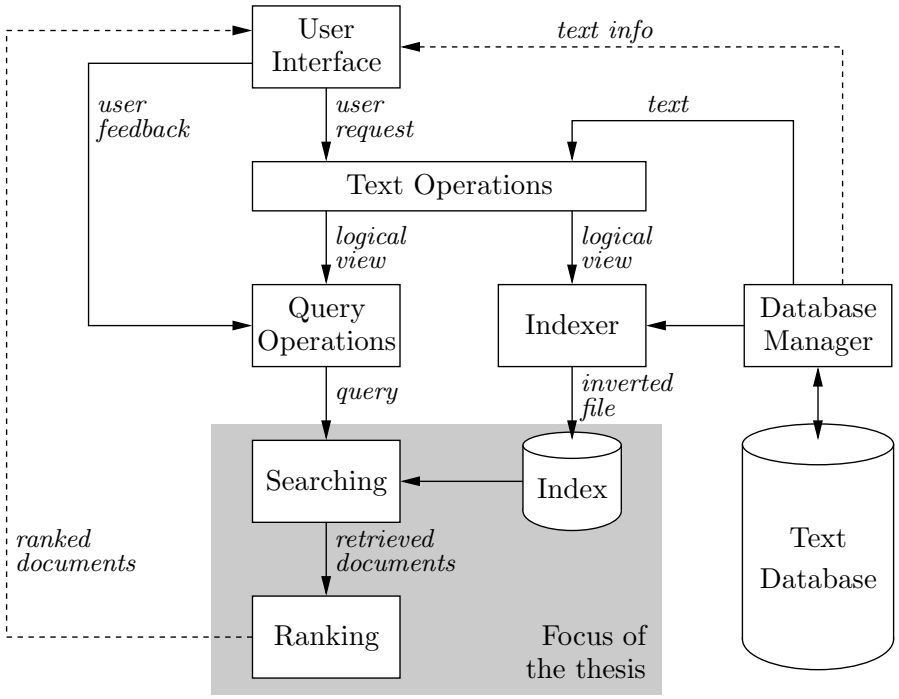


Fig. 1.1. A generic representation of the process of retrieving information from [Bae99] interpreted in terms of functional components. Our work in the current thesis focuses on the highlighted zone including the search within an index and the ranking of the retrieved documents.

reaching a wider audience than ever. Second, the advances in all kinds of digital communication provided greater access to various networks, implying that information sources are available and rapidly accessed even if distantly located. Third, the freedom to post and publish any kind of information to be shared has greatly contributed to the popularity of the World Wide Web (WWW).

Fig. 1.1 shows a generic architecture of an information retrieval system in terms of component subprocesses from [Bae99]. The system is divided into two parts sharing the same text operations, i.e., the user interface for the queries and the system part that provides text resources to be indexed for the search. The text operations transform the original documents and generate a logical view of them by specifying the structure of the elements that can be retrieved. Once the logical view of the documents has been performed, an index is built in order to allow fast searching over the large volume of data. In most of the cases, the index owns the structure of an inverted file for which each entry gives a word and a list of texts, possibly with locations within the text where the word occurs. Once the document database has been indexed, the user initiates the retrieval process by specifying a query which is parsed and transformed by the same operations applied to the text from the database. The retrieved documents obtained after processing of the query during the searching phase have to be sorted according to their order of relevance in the ranking phase before being sent to the user.

Hence, the explanatory model of an IR system must be considered as an introduction to the real problem of using hardware resources efficiently in order to accelerate the search process. There exist many algorithms which can be used for textual search.

However, we intend hereby to analyze promising computing strategies which can handle long queries of misspelled words. Even the biggest search engines have a limited input string length in their inquiry process, e.g., Google with a maximum of ten words per query in 2005. Since query requests are inherently vague, retrieved documents might not be the expected answer and must be ranked according to their relevance to the query. As depicted in Fig. 1.1, sorting constitutes one of the three main areas related to IR systems that we address in this work, as well as searching and the use of a huge index file stored in dedicated memory devices. These domains are glued together through the realization of a efficient hardware platform for the acceleration of the whole retrieval process.

2 The Problem with Storage Devices

This section exposes briefly the problem of today's computerized systems to face an always more active user community greedy for information. Hence we aim to justify the need of a hardware support through scientific historical observations reported in various publications and through the evolution of the hereby concerned technologies.

2.1 Running into the Memory Wall

Undoubtedly, Moore's law keeps stressing processor technologies [Moo03, Olu05]. As we will see in Chapter II, the von Neumann paradigm which consists in improving fairly straightforward single processor architecture by increasing the clock frequency, already shifts towards Single Instruction Multiple Data (SIMD) or directly parallel processing technologies. However, while microprocessors follow an explosive growth in performance, DRAM-based memory systems fall behind creating the so-called memory wall [Bur96, McK04, Wul95]. In the beginning of the 90s, it has been estimated that the Central Processing Unit (CPU) speed of the fastest available microprocessors is increasing at approximately eighty percent per year [Bas91], while the speed of memory devices is growing at only about seven percent per year [Hen90]. Regarding our target application, it becomes clear that the most important hardware components dedicated to the storage of information within computer systems are the main memory and the Hard Disk Drive (HDD).

As seen in Fig. 1.2, the physical areal densities for both semiconductor memories and magnetic discs are exponentially growing while their price is continuously falling. According to Grochowski *et.al.* [Gro03], a significant evolution in disk drive form factor has occured over the past twenty years, as reported in Fig. 1.3-a, leading to an always smaller size with nonetheless higher capacity. Lesk's report [Les97] made estimates of disk sales and predicted necessary media sizes for storing multimedia information.

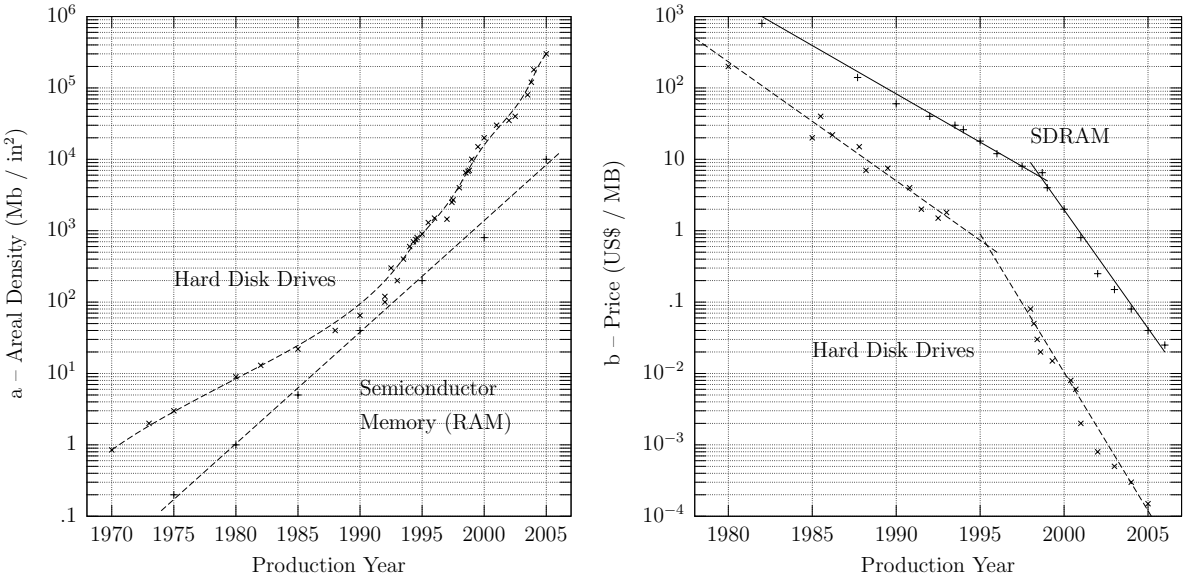


Fig. 1.2. Trends in **a)** areal density and **b)** price for storage systems including magnetic hard disk drives and semiconductor memory devices from [Bur96, Gro03].

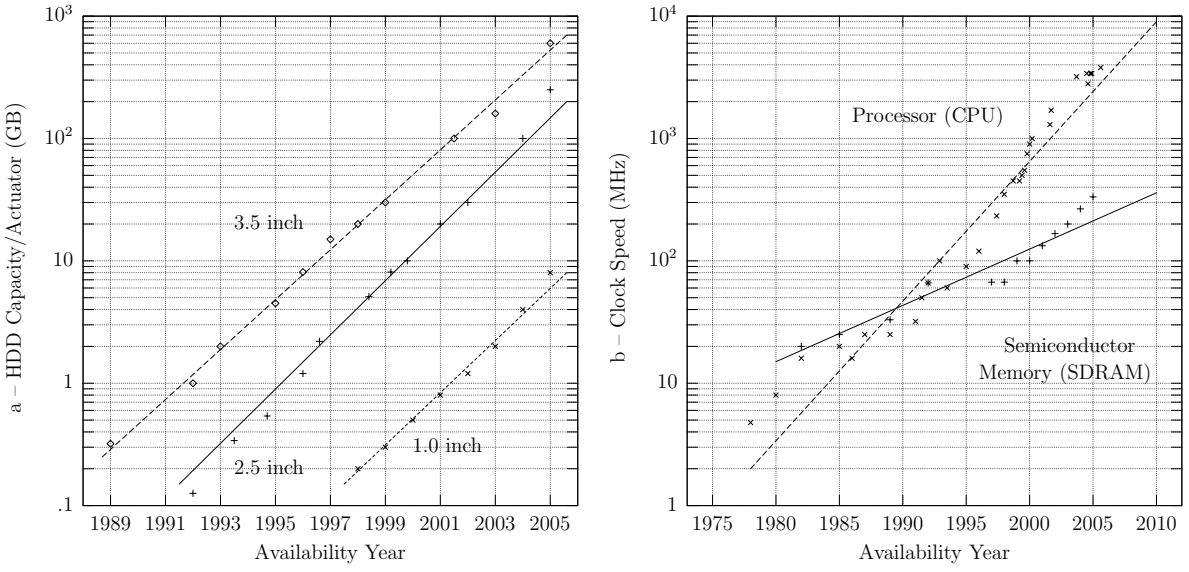


Fig. 1.3. Evolution of **a)** the capacity of hard disk drives according to [Gro03] and **b)** of the clock frequency for semiconductor memory devices from [Ito01, Itr06, Rab03] and processors from [Bur96, Rab03] creates the memory wall [McK04, Wul95].

According to Lyman *et.al.* [Lym03], the world was producing more than one exabyte (1EB= 10^{18}B) of unique information per year in 2000 and five in 2002. Furthermore in 2004, over fifty million web pages were new or changed everyday [Cas04].

Finally, Fig. 1.3-b plots the integration density of both processors and memory as a function of time. The memory wall [Bur96, McK04, Wul95], i.e., the growing gap between the curves on Fig. 1.3-b, is being an always more important problem in digital systems, as RAMs are not able to feed CPUs with data as fast as those can process

them. Due to increases in storage capacities, soon it will be technologically possible for an average person to access virtually all information ever recorded. However, even though we are able to store millions of electronic documents on magnetic media, the problem of retrieving information is more linked to active devices such as memory and processors than to HDDs.

2.2 Need for Hardware Support

The development of larger and larger Random Access Memories (RAMs) during the last fifty years has made the problem of searching a very interesting and well investigated realm. On the one hand, large databases tend to make the retrieval process more complex and more intricate, since having more data available means a bigger probability of matching and thus a longer list of possible responses to a stated query. On the other hand, large databases stimulate people to ask arbitrarily any complicated question about anything. Composing a very precise enquiry may regroup words with an explicit order of appearance and eventually exclude some other terms, as we know from our use of Internet search engines. More importantly, people might also be incited to perform big block searches, like entire paragraphs of text, with the ability to locate items when only a part of the key information is specified, hoping that the matching will be significant enough for the most relevant addresses to be returned. As we found out yet that there is no really efficient generic solution to this problem, we dedicate this thesis to the development of a hardware search engine for large databases.

Intelligent data manipulation, storage, retrieval, and interpretation are some of the most ubiquitous functionalities any IT device must provide. Regarding the problem of searching information, from a logical point of view, it is very inefficient to move in and out several hundreds of gigabytes (GBs) between memory and processor over a restricted size bus in order to retrieve the few relevant kilobytes (KBs). From a financial point of view, hardware manufacturers have to constantly add features to their devices in order to stay ahead of the competition. Besides data integrity and corruption control, a very relevant example of in-device tools a disk drive could be a low-level search functionality [Hug02]. However in today's systems, storage intelligence still resides on the computer side of the drive interface. In our opinion, the solution is to delegate this task where it belongs, i.e., to the storage medium itself or as close to it as possible. While this is practicable only by redesigning the memory chips or the HDD controllers themselves, there are simple ways of short-cutting the path between storage and CPU by using dedicated search coprocessors.

3 Organization of the Thesis

The organization of this thesis is essentially oriented towards the development of a hardware search engine based on an associative computing paradigm for textual queries. Figure 1.4 shows the relationship between all the subjects treated in this document, starting from the global concept of information retrieval down to the finalization of our Associative Computing Engine (ACE) on a hardware platform that provides a high speed-up potential for textual query applications.

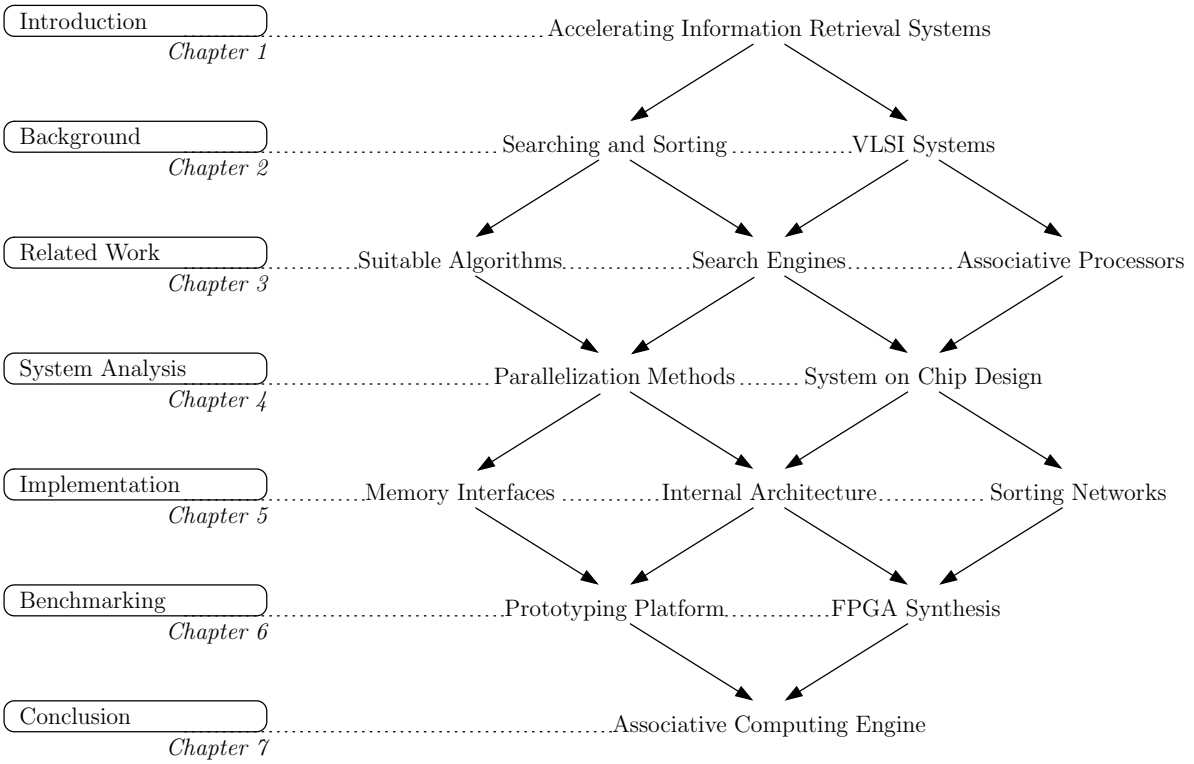


Fig. 1.4. Hierarchical organization of the thesis: as the chapters follow the different steps in the development of the ACE, a logical flow relates the main ideas treated in this dissertation.

For that matter, after an introduction to IR systems in general and the necessity of making searches faster as given in chapter one, the second chapter sets the basis of this work by presenting state-of-the-art techniques for searching and sorting, as well as an overview on Very Large Scale Integration (VLSI) system design. These background principles provide the fundamental ideas from which the mainstream of the thesis is derived. Past and current research performed in related fields is reviewed in the third chapter. Not only does it confirm the importance of our contribution, but it also raises the expectations regarding the final result. A method relying on associative memory accesses [Lap92] is also presented in more detail, since we think it is the one with the biggest potential for being efficiently implemented as our database-searching accelerator.

With an emphasis on hardware design, chapter four is dedicated to the profiling of the selected algorithm and to the application of different methods which make the implementation extremely promising, using primarily parallelization techniques and large

bus widths. The profiling of the software model based on the method described in the previous chapter permits an ideal partitioning of the operations between hardware and software and guides us through the parallelization of the algorithm. Moreover, a study on System on Chip (SoC) designs and associative processors must allow us to develop the best suited architecture for the porting of the previously chosen algorithm onto hardware. However, in the development of such a complex system, it is ineluctable to consider every single interface and data transfer at the lowest level of abstraction particularly for the memory subsystem. The choice of the storage media for data retention directly influences the overall performance of our database system. In chapter five, the selection of a given memory device class and the high data throughput necessary for obtaining the requested short searching durations brings hard area and time constraints into the system, especially in the second half which deals with the arrangement of the best matching candidates. We show how this problem can be solved in hardware using different design techniques and a well-advised mix of sorting algorithms, as introduced in chapter two.

Chapter six is porting the complete design of the ACE onto an FPGA-based prototyping platform in order to benchmark the system. A proof-of-work as well as a performance analysis is obtained through the comparison with software models running on different computers and other simulated hardware platforms. Finally, an overview of the achieved work is given in chapter seven, concluding on the performance of the ACE. Future work and further potential improvement directions for the developed system are proposed afterwards considering more up-to-date technological advancements.

Chapter II

Background

HAVING provided a rather large contextual description of the environment in which this thesis has been realized, we dedicate this second chapter to an overview of the state of the art in Very Large Scale Integration (VLSI) design as well as sorting and searching methods. The presented algorithms constitute the background knowledge which illustrates the theories related to information retrieval and allows a better appreciation of the work in progress. For the concrete realization of an Information Retrieval (IR) hardware accelerator, it is necessary to start with digital electronic system design. Secondly, the best known searching algorithms shall be reviewed in order to depict classical techniques used in IR based on text data. Finally, as sorting and searching are relatively close to each other, various algorithms shall be presented in order to provide a large overview of the different techniques that could be used for the realization of the final system.

1 Designing Digital Systems

Hardware accelerators are separate units close to Central Processing Units (CPUs) that perform some functions faster than a software program running on general purpose processors. They are usually made out of Integrated Circuits (ICs) communicating together over dedicated interfaces. The design of an acceleration platform requires not only a good knowledge about all the different kinds of chips that can be used to build a complete system, but also about how they can and must be combined in order to achieve top performance with a maximum efficiency.

1.1 Integrated Circuits Implementation Strategies

According to the literature [DeM94, Kun88, Rab03, Smi97, Wes93, Wol02], digital ICs are typically classified into three main categories. These include programmable standard architectures such as common microprocessors, programmable logic architectures such as Complex Programmable Logic Devices (CPLDs) or Field Programmable Gate Arrays (FPGAs) and application specific logic architectures highly optimized in terms of functionality, performance, power and cost.

1.1.1 The Choice of an Architecture

Fig. 2.1 represents the computational efficiency of different classified groups of ICs versus their flexibility, i.e., the inverse of the time required to implement a design, and power dissipation, according to [Syd03]. On the one side of the IC spectrum, programmable standard architectures emphasize General Purpose Processors (GPPs) as well as Digital Signal Processors (DSPs) or super-scalar processors. These are highly flexible generic devices, the functionality of which is determined by loaded software code. However, since this code is executed sequentially, the processing time of a function can be quite long. Such devices are therefore typically used in applications which allow a longer response time. As we are building a specific acceleration unit, we might want to trade flexibility for performance.

On the other side of the IC spectrum, application-specific logic architectures emphasize many kinds of Application Specific Integrated Circuits (ASICs) which differ mostly from the way they were designed. The highest degree of optimization is obtained with full custom implementation, while semi-custom devices offer quicker design processes.

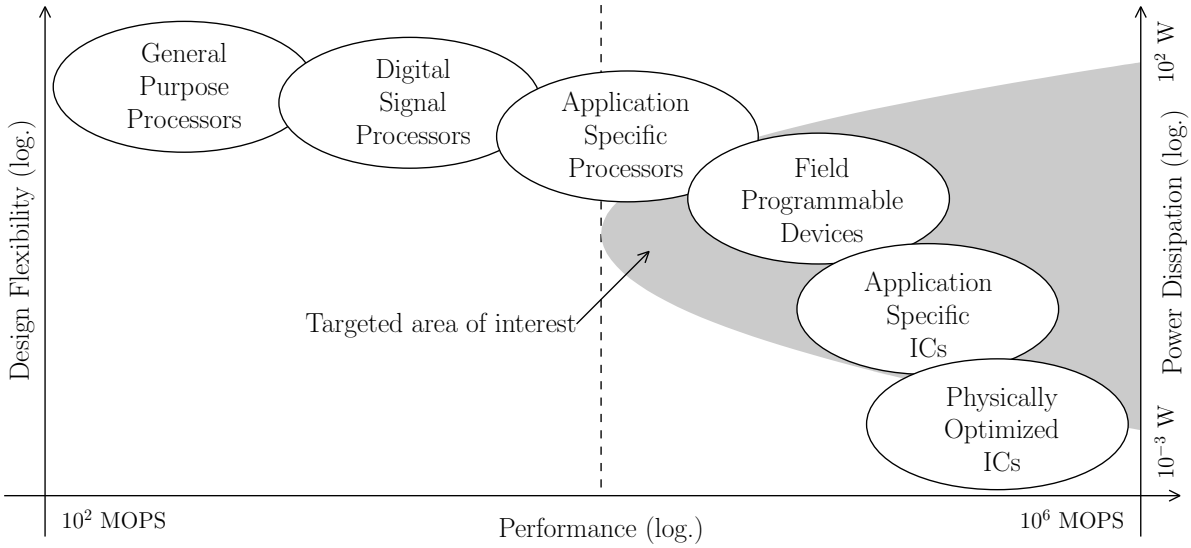


Fig. 2.1. More flexibility has to be paid off by a larger power dissipation from [Syd03]. As we strive for performance, our area of interest includes FPGA and ASIC devices allowing an averaged power consumption for a respectable design flexibility.

Located in the middle of the spectrum, as a compromise between flexibility and performance, field programmable devices are gate arrays which provide highly standardized means to implement digital integrated circuit designs. They are manufactured as regular arrays of patterned blocks of transistors which can be interconnected to form logic elements such as gates, flip-flops and multiplexers [Bet99].

1.1.2 Pipelining Techniques

Digital systems are usually controlled by one or a set of clock signals, which ease the design as well as the development and the verification of complex architectures through a synchronous operation mode. Dividing a system into many smaller sub-circuits, hence separated by registers placed on their respective outputs, permits a global synchronization of the data flow with respect to the system clock triggering these registers. However in a synchronous work process, the highest possible system clock frequency and thus the maximum data throughput is conditioned by the operating time of the slowest sub-circuit of the system. Propagation delay compensation between slower and faster parts of the system is not possible within one clock domain.

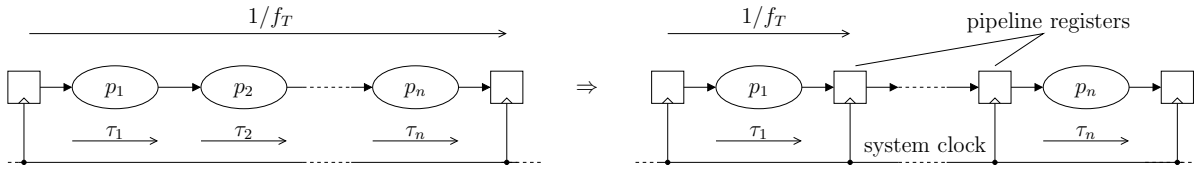


Fig. 2.2. Pipelining consists in inserting registers between the different parts of a computational path. This method increases the system frequency f_T and the data throughput as well.

There exist different methods to increase the throughput of digital synchronous systems. Pipelining consists in inserting levels of registers within a series of logic blocks in order to break the critical path, i.e., the longest signal path of the design created by the logic gates between two registers. As seen in the left part of Fig. 2.2, for n chained asynchronous logic elements with a respective processing time $\tau_1, \tau_2, \ldots, \tau_n$ where registers controlled by a system clock of frequency f_T are located at the extremities of the chain, f_T should respect the condition

$$f_T \leq \frac{1}{\tau_1 + \tau_2 + \ldots + \tau_n} \tag{2.1}$$

so that the correct propagation of the logic levels is assured. The intrinsic delay caused by the typical setup and hold time of the registers is hereby neglected. The insertion of additional registers, as seen in the right part of Fig. 2.2, permits a pipelined processing of the data, that is to say the result of a part p_i of the system can be safely processed by the following part p_{i+1} while the next dataset from p_{i-1} is input to p_i. Hence, the resulting system clock frequency f_T depends on the longest propagation delay of all the parts of the system only, such that

$$f_T \leq \frac{1}{\max\left(\tau_1, \tau_2, \ldots, \tau_n\right)}. \tag{2.2}$$

Theoretically, if all the logic blocks between the pipeline registers have approximately the same propagation delay, the pipelined circuit outperforms the original one by a factor equal to the number of pipeline register levels. However, the overall latency of the complete circuit grows with the depth of the pipelining. This latency, expressed in terms of clock cycles, is defined through the processing time of a pipelined system between the reading of the input dataset and the presence of the result at the output register.

In this work, we make large use of the pipelining technique in order to be able to handle the data fetched from the memory rapidly enough. Not only are we able to process the data faster through an increased system clock frequency, but also work on a huge amount of data concurrently and make a very efficient use of the hardware resources.

1.1.3 Cut-Set Technique

Pipelining is very practical for signal processing algorithms since these are typically applied to a continuous data flow, in comparison to data-processing algorithms in which conditional branches depend on the information currently treated. Widening the principle of pipelining, the cut-set technique [Kun88] is a method permitting the redistribution of propagation delay within an electronic circuit without having to change the actual implementation of the algorithm. Practically, it consists in grouping a set of different processing elements of an architecture together in a virtual box where every incoming signal will be delayed by one clock period (T) and every outgoing signal will be given the opposite negative delay ($-T$). In Fig. 2.3 showing a synchronous accumulator subjected to pipelining, these delays are respectively represented by an empty circle and a full colored circle.

As negatively delaying register cannot exist physically, a supplementary block of registers must be inserted for the synchronization of the output signals. Moreover, if

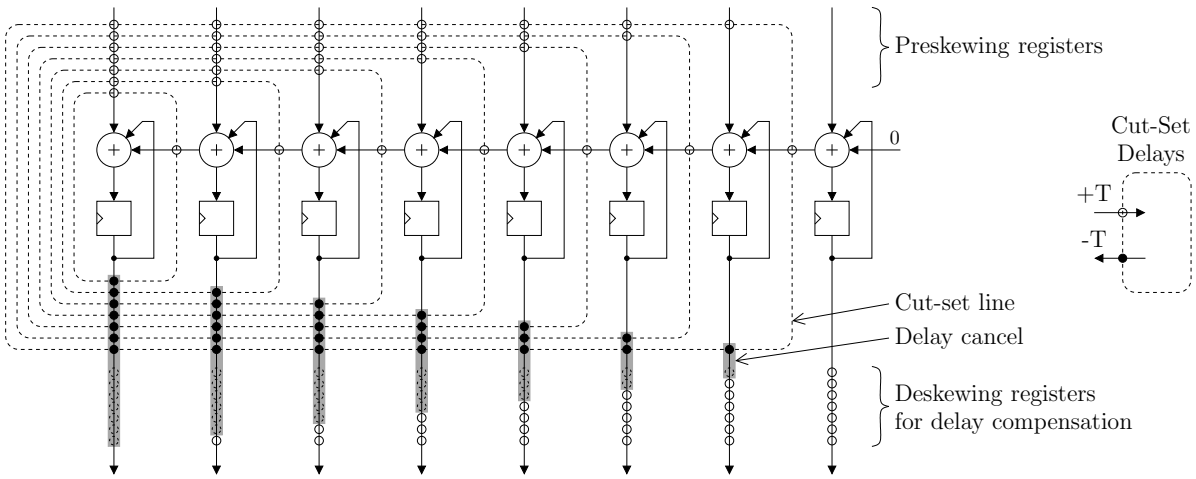
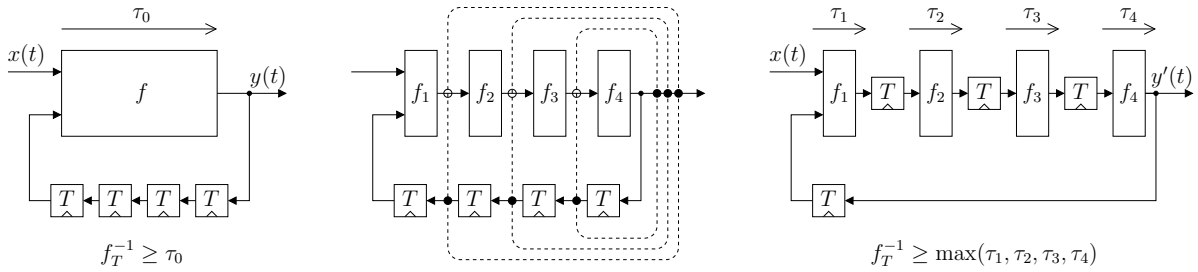


Fig. 2.3. Design of an accumulator using the cut-set technique in order to increase the clock frequency. The critical path composed by the carry signal of the full-adders is intersected by each cut-set line, the number of which equals the final latency in clock cycles.

the output of the accumulator has to be processed word-wise, these deskewing registers are necessary for data consistency. They permit canceling the negative delay $n \times (-T)$ earned through n cut-set lines and hence allow a physical realization of the architecture. Applying the cut-set technique consequently ensures a functional equivalence of the thus pipelined processing network. As an example, Fig. 2.3 assumes that the effect of the registers with the negative delay $(-T)$ has been compensated by the dotted deskewing registers in the grey area. By showing how the method works at the bit level, we can easily extend the idea to the system level, i.e., when a bit line becomes a bus. Moreover, when cut-set lines interface directly inputs or outputs, negative delays can be ignored if the timings are reconsidered, i.e., the input word must be presented earlier or the output has to be read later accordingly to the number of delays neglected.

1.1.4 Retiming Recurrent Architectures

Pipelining increases the internal processing clock frequency f_T of a system through the insertion of intermediate registers on the data path. As a direct consequence, the latency of the processing part strongly influences the pipelining of recurrent structures in which data is fed from the output back to the input. The solution remains in the compensation of the resulting latency of the data path through the functional delay of the feedback loop. In this case, an architectural retiming based on the cut-set technique can provide an increase of the throughput.



a) Original recursion
$y(t) = f\,(x(t), y(t-4T))$

b) Cut-set technique
$f = f_4 \circ f_3 \circ f_2 \circ f_1$

c) Result after retiming
$y'(t) = y(t-3T)$

Fig. 2.4. Retiming a recurrent architecture is possible when the latency of the data path after pipelining can be compensated by the functional delay of the feedback loop.

Fig 2.4 shows the example of a system f with a propagation delay τ_0 which includes a feedback loop for a computation $y(t)$ based on an input value $x(t)$ and a previous result $y(t-4T)$ with $T = f_T^{-1}$. When a recursive function f owns a loop delay of $n \times T$, it becomes possible to part the system in maximally n sub-functions $f_1, \ldots, f_n$ with a respective propagation delay $\tau_1, \ldots, \tau_n$ and increase the system clock frequency f_T in relation to (2.2). With the condition that the loop delay stays larger than T, it can be minimized while the throughput gets maximized.

1.2 Microprocessor Design Techniques

The original goal of a processor is to perform mathematical calculations in a very fast, flexible and reliable manner. With the advances of technology, they reach always higher performance levels in terms of speed, while the power consumption has continuously been reduced. If greater connectivity helps build larger and more parallel systems, more architectural specialization is required for specific markets such as multimedia processing or pattern matching. In this section, we aim to give an evolutionary overview on processors in general, because we make wide use of these design concepts in our work. For further reading, we would like to refer to Hennessy and Patterson text books [Hen90, Pat97a] which provide a broad framework for computer architectures.

1.2.1 Instruction Set Architecture

Considered as a generic term, a microprocessor is a digital semiconductor IC which typically serves as a Central Processing Unit (CPU) in a computer system and generally includes different components with specific functions such as the data path, the control path, some memory units, a clock distribution network and transceiver circuitry. The history of GPPs began more than fifty years ago, where CPUs were simply programmed using machine code. As industrial applications became more complex, the concept of portability permits to move up to larger computers and stay compatible at program and data level. In order to make an entire family of computers run the same software, the adopted solution was to produce a quite complete and capable reference instruction set. The Complex Instruction Set Computer (CISC) was able to fetch fewer instructions from the main memory that could remain slow and less expensive. The evolution led to the development of better and more efficient compilers to bridge the semantic gap between the high-level language of the algorithm and the machine code.

However, most compilers use only a small subset of the available instructions. Redesigning the processor for supporting a simpler instruction set, with less addressing modes but with more internal registers, made it faster and less expensive at the same time. Hence the resulting Reduced Instruction Set Computers (RISCs) yield in a very simple core CPU running at very high speed, supporting the exact same sorts of operations the compilers were using on a register basis [Hen90].

1.2.2 Parallelization of the Operations

There exist mainly two different standard architectures in processor design, both with advantages and drawbacks. For the Von Neumann machines, programs and data are mixed in a single memory device, requiring sequential accessing which produces a bottleneck. For the Harvard machine, they occupy separate memory devices and can be accessed simultaneously. These design variations remain a trade-off between internal complexity of the CPU and throughput to the program memory.

A very important break-through began with the use of instruction pipelining, in which the processor works on multiple instructions in different stages of completion.

With this technique, processors can be clocked almost as many times faster as there are pipelining stages. However, in the typical case of a conditional branch command, the processor must know the result of the previous operation and hence has to wait for the pipeline to be empty.

Another improvement of the structure of the processor came with the introduction of additional Arithmetic Logic Units (ALUs) within the data path in order to execute multiple instructions in parallel, implying that these are not dependent on each other. The resulting architecture is qualified as super-scalar processor and provides a well-suited and efficient platform to applications with a high level of instruction parallelism. Following this idea, the parallelization of the core of the processor resulted in a new conceptual idea which consists in combining different kinds of instructions into an internal single Very Long Instruction Word (VLIW). Provided with a diversity of specialized execution units, the CPU is capable of executing all of the instructions contained in the instruction word, in parallel.

1.2.3 Parallelization of the Data

A vector processor is a CPU designed with a Single Instruction Multiple Data (SIMD) architecture that is able to operate on multiple data elements simultaneously, taken into account the possibility to pipeline the instructions and the data as well [Pat97a]. Computing architectures where many functional units perform different operations on different data simultaneously are qualified of Multiple Instruction Multiple Data (MIMD). A recent industrial example that shows an extensive use of this concept presented with the Cell processor in [Dho05, Moo06] was designed to bridge the gap between conventional desktop processors and more specialized high-performance processors such as graphics chips. Its architecture includes a main multi-threaded processor core, eight fully functional coprocessors and a specialized high-bandwidth circular connecting data bus. The main processor is not intended to perform all primary processing for the system, but rather to act as a controller for the other eight coprocessors which handle most of the computational workload.

Nonetheless, from the commercial point of view, the cost of multi-processor devices must be justified by the applications that run on it. When designed for high-performance markets, these devices require customization of each internal element to achieve frequency, power dissipation and chip area goals. As power consumption essentially has made progress stop in conventional processor core development, manufacturers try to find new ways to utilize the ever-increasing transistor budget more effectively. Olukotun *et.al.* [Olu05] describe the advent of Chip Multi-Processors (CMPs) and depict the next system development steps through the necessity to port latency-critical software into multiple parallel threads of execution.

1.3 Field Programmable Gate Arrays

Since their introduction in the mid 80s, Field Programmable Gate Arrays (FPGAs) have become one of the most popular implementation media for digital circuits. Similarly to Complex Programmable Logic Devices (CPLDs), FPGAs are digital devices based on configurable logical cells and configurable interconnect structures. They have the ability to implement any circuit by being appropriately programmed [Bet99, Wol04]. As depicted in Fig. 2.5, FPGAs typically include Complex Logic Blocks (CLBs), routing lines of different lengths for global and local communication interconnected through switch matrices, line buffers and programmable interconnects that incorporate the logic into the routing network and thus form larger circuits. Grouped in clusters, CLBs are used to implement various logic functions of a larger number of inputs by means of Look-Up Tables (LUTs) or multiplexers. They include registers which permits the design of synchronous circuits and the implementation of local memory blocks. Programming bits configure the output response of the CLBs as well as the pass transistor switches for the routing.

As they are manufactured using the latest technologies, they provide a very high-capacity in equivalent ASIC gates which makes them a good alternative for large designs. Following Moore's law, the number of gates and features inside FPGAs has increased so dramatically that they have progressed to a point where System-on-Chip (SoC) [Wol02] designs can be built on a single device. They compete with capabilities that have traditionally been supplied through ASIC devices only and offer a lot of advantages compared to DSPs or general-purpose processors. Not only do they give system designers the possibility to parallelize their architectures, but they provide an excellent density of hardware Intellectual Property (IP) and lower the price per Giga Operations Per Second (GOPS) rapidly. Moreover, it has been shown that they achieve greater performance per unit of silicon area than processors [DeH00b]. This gain however has to be paid

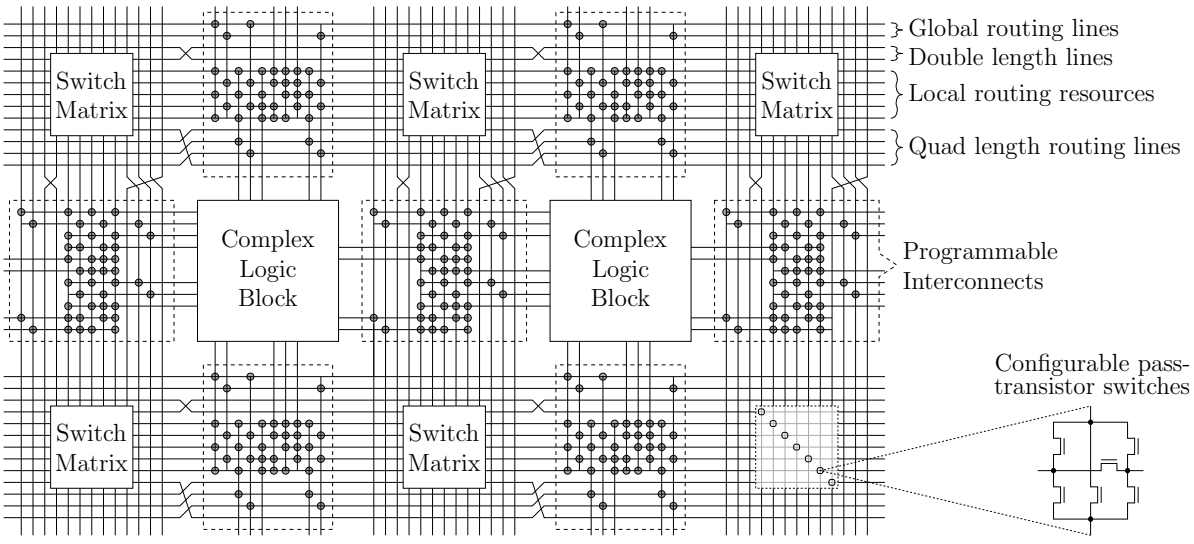


Fig. 2.5. Representation of a generic island-style FPGA which includes CLBs, switch matrices, programmable interconnects and routing lines of different lengths.

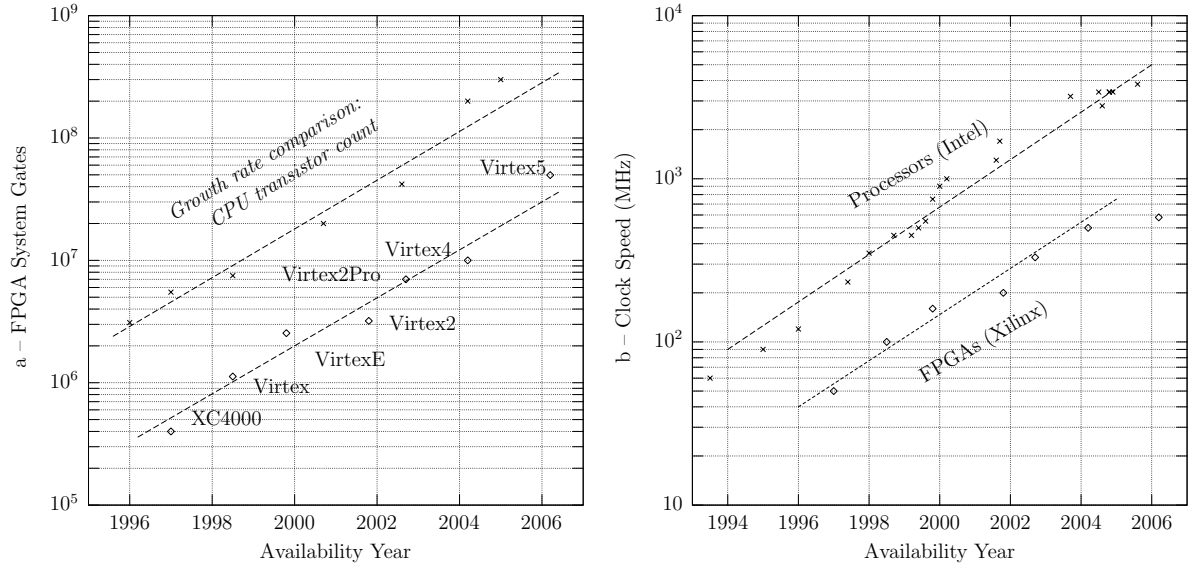


Fig. 2.6. General-purpose processors can be compared with programmable logic devices. In **a)** as for CPU [Int06], FPGA [Xil06] industry trend has an exponential growth in density and **b)** in clock frequency.

off by a relatively larger power consumption. Given for comparison in Fig. 2.6-a, the transistor count in processors grows with the same rate as the amount of system gates in the FPGA devices. Fig. 2.6-b reports that the technology behind FPGAs allows the realization of digital systems that can be clocked only four to five times slower than general purpose processors.

Often seen as their rivals although they are not meant to supersede them, FPGAs provide many advantages against ASICs. As a fast procedure for implementing hardware while allowing a rapid functional verification, they yield an improved time-to-market and a low NRE (Non-Recurring Engineering) costs [Bet99, Rab03] so that they are extremely adapted for small to medium volume productions. However, due to a high signal delay and power consumption, they cannot permit an optimal utilization of the silicon area and come at the expense of lower performance and integration density. In addition, routing problems can limit the flexibility and the implementability of rather complex or highly interconnected systems, as we have experienced in this work, e.g., with the sorting networks. Moreover, physical constraints such as potential clock-skew may appear in late design phases, especially through the communication with peripheral units or in systems with multiple local or global clock domains, e.g., with the use of external synchronous memory chips.

In most of the applications, the ultimate goal is to reach the maximum performance of a system in terms of speed and throughput. On the one hand, the flexibility offered by microprocessor or DSP clusters through their high-level programmability is very attractive, but multi-processor programming is extremely hard, especially for real-time applications. Compared to FPGAs, they dispose of limited I/O capabilities, a high power consumption for a low computational density. On the other hand, with a lack of flexibility and long design cycles, ASICs are meant for large volume productions. As they

are not error-tolerant during system conception, they would not allow any application change or functional design improvement. Therefore, we estimated that FPGAs would be the most suitable choice for the implementation and the verification of our present research work.

1.4 Semiconductor Memory Devices

When it comes to memory devices, there are several different technologies available for use in modern integrated systems. However, not all of them are suitable for the implementation of a fast database search engine. The type of memory unit which is preferable for a given application is a function of the required memory size, the time it takes to access the stored data, the access pattern, the application itself and the system requirements [Rab03]. Depending on the level of abstraction in a design, the size of the memory needed in the system might be expressed in different manners. On the one hand, from the side of the application developers, the ACE platform deals with signature files of a given length, such that the total amount of memory needed would be expressed in terms of signatures or words, e.g., the database must be able to store up to one million signatures. On the other hand, hardware system architects are more likely to refer to this quantity using bytes or megabytes, e.g., the size of the database must be of at least ten gigabytes.

Basically related to the functionality implemented in a digital system with data storage, there exist different classes of memories, as shown in Fig. 2.7. First, a distinction is made between ROM (Read Only Memories) and RWM (Read & Write Memories) [Rab03]. Secondly, whether it belongs to the nonvolatile or volatile class, i.e., whether the contents are kept or lost when power is turned off, is also an early selection criterion. Then a distinction can be made for the volatile RWM devices between the ones with a restricted access resulting in either faster access times, smaller area or a special functionality such as FIFO, LIFO, shift registers and the ones with a random access (RAMs). The latter split into the static and the dynamic RAM categories. The primary difference between them is the lifetime of the data they store. SRAM retains its contents as long as electrical power is applied to the chip. If the power is turned off or lost temporarily, its contents will be lost for ever. By contrast DRAM has an extremely short data lifetime, typically about a few milliseconds, and they require a dedicated controller refreshing the data periodically before it expires. Flash memory combines the best features of the nonvolatile RWM devices described above. They are high density, low-cost, nonvolatile, fast in reading, and electrically reprogrammable. However, a high throughput to a huge database for the ACE platform would be reached at a much higher cost than with a RAM solution.

When deciding which type of RAM to use, a system designer must consider access time and cost. Timing parameters play an important role in the choice of a memory. In an electronic design, one has to consider the read access time, i.e., the delay between the read request and the moment the data is available at the output, the write access

time i.e., the delay between the write request and the final writing of the input data into the memory and the cycle time, i.e., the minimum time required between two successive read or write operations. Depending on the technology of the memory devices, these timings present huge variations [Rab03]. SRAM devices offer extremely fast access times but are much more expensive to produce. Generally, SRAM is used only where access speed is extremely important. A lower cost-per-byte makes DRAM attractive whenever large amounts of RAM are required. Many embedded systems include both types: A small block of SRAM, in the region of a few hundreds kilobytes, along a critical data path and a much larger block of DRAM, perhaps tens of megabytes, for general-purpose storage.

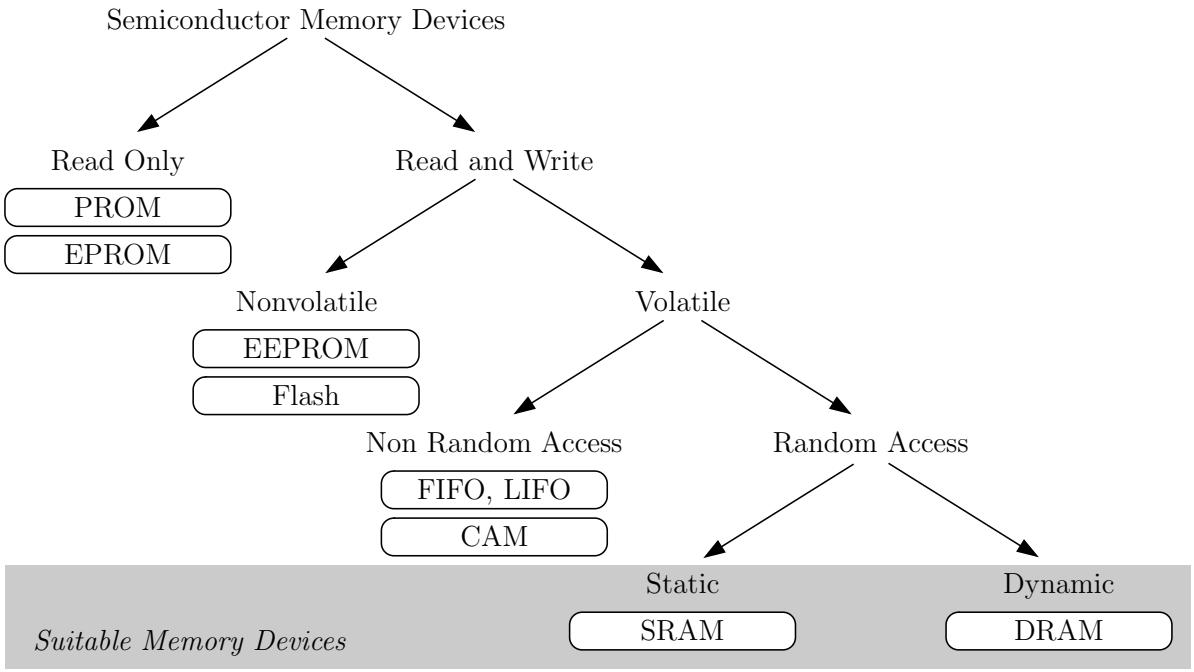


Fig. 2.7. Classification of semiconductor memories. Devices suitable for the realization of a search engine within a digital system using an external database are static and dynamic Random Access Memories (RAMs).

When massive amounts of storage are needed, semiconductor memories tend to become too expensive so that more cost-effective technologies such as magnetic and optical disks should be used instead. However, although they provide extensive storage capabilities at a low cost per bit, they tend to be quite slow, and they can definitely not be a part of a SoC design, which makes them unsuitable for the ACE platform. If both SRAM and DRAM are suitable for the realization of the ACE, it should be clear that the real challenge lies in the conception of a platform based on the much more complex DRAM devices compared to the relatively simple SRAM type.

2 Pattern Matching in Strings

Introduced in Chapter I, searching is a popular problem. In the last decades, many algorithms have been developed, studied, published, e.g., in [Bae99, Hyy02, Hyy04, Hyy05, Knu97, Nav02, Nav05, Sed88, Sun90], and still constitute an excellent reference in the field. As a matter of fact, texts remain today the main form of information exchange although data is stored in many advanced formats. Since our target is the retrieval of information hidden in texts, we concentrate in this section on the most relevant string matching algorithms.

2.1 Classical Algorithms

A text is basically any sequence of symbols or characters drawn from an alphabet Σ, such that when a query string Q of length m is to be found in a text T of length n from a collection, string matching requires to point out all the occurrences of Q in T [Bae99]. Classical string matching algorithms are based on character comparisons. First of all, the Brute-Force (BF) algorithm suggests itself naturally with an $\Theta(mn)$ complexity[1]. It consists in comparing every single character of Q with one of T sequentially from the left to the right. If there is a match, then the next characters are respectively compared until there is a mismatch or a complete match. In either case, the pattern Q is shifted one position to the right and the sequence of comparisons starts from the beginning. The algorithm ends when the remaining number of characters in T is lower than m and returns the locations where Q was found in T if any. The odds are its simplicity and the absence of preprocessing.

By taking advantage of the pattern alignment knowledge obtained during previous comparisons, the Knuth-Morris-Pratt (KMP) algorithm [Knu77] reduces the time for searching Q in T to $\Theta(n)$. However, a preprocessing time in $\Theta(m)$ is necessary to analyze the searched pattern Q in order to know after each mismatch the next possible beginning of Q in T. The algorithm makes its greatest gain over a naive string matching when it can skip the most characters at a time.

Similarly to KMP, the Boyer-Moore (BM) algorithm [Boy77] uses a backward comparison scheme for the search pattern Q while scanning the text T from left to right. The key of the method is that in case of a mismatch with a character that doesn't appear in the pattern at all, then it can be shifted of m positions to the right, as there is no chance that the pattern starts within the next m characters. The advantage is that in the best case, only one in m characters needs to be checked, so that it can run in $O(\frac{n}{m})$ in average. Moreover, small amounts of iterations are reached when the alphabet Σ gets bigger and the length of the pattern m smaller [Sed88]. An example of pattern matching using the BM algorithm is illustrated in Fig. 2.8. There exist many improved variations of the method in the literature and we would like to point to the bibliographical references for more details.

[1] According to Knuth's notations from [Knu76].

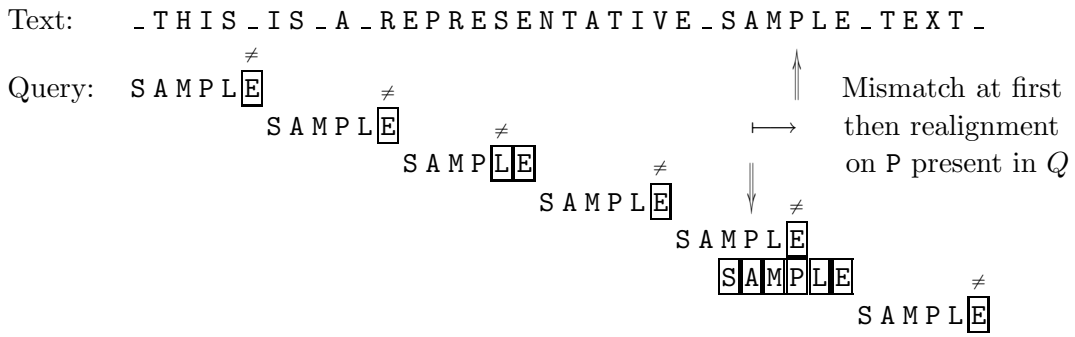


Fig. 2.8. Example of pattern matching using the Boyer-Moore algorithm. The query word "sample" is compared backwards character per character to the text from the left to the right. 13 comparisons marked ▯ were performed within 7 pattern positions, i.e., 6 shifts.

When large texts must be scanned for a search pattern, a sequential method is not a good option since its time complexity depends in the best case linearly on the size of the texts. An ordering relation such as a sorted table or a tree would reduce the time needed for retrieval to a logarithmic complexity. For frequent queries, the cost of the preprocessing, i.e., the sorting of the words of the text into a table, becomes negligible as it must only be performed once. In binary search trees for instance, every node's left subtree has keys less than the node's key, and every right subtree has keys greater than the node's key. Searching for a specific value consists in a recursive process that is performed beginning by examining the root, finding the median, determining whether the desired value comes before or after it, and then proceeding to the remaining half in the same manner, e.g., in the way of the divide-and-conquer algorithm and the dichotomic search [Knu97, Sed88]. Practically best suited for software solutions, trees are usually implemented with pointers in order to be flexibly extended but may yield a variety of constraints on how they are composed. However in this thesis, we are looking for a much more flexible and faster method for retrieving textual information in huge pools of texts, which shall turn out to be extremely well suited for a later hardware implementation.

2.2 Flexible Pattern Matching

Owning more flexibility means being able to find approximate matches to a pattern in a text string. Practically, an approximate matching algorithm must be given a variable that specifies the biggest number of misspelled characters in the query words. Compared to exact matching, it allows a certain number k of differences at the item level, e.g., characters in the case of texts. This measure of dissimilarity is known as edit distance and encompasses several popular measures such as Levenshtein or Hamming distance [Nav01]. In information theory, the Hamming distance between two strings is the number of positions for which the corresponding symbols are different with the restriction that these are of the same length. As an extension to this measure, the edit

distance is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character.

Most approximate matchers used for text processing are based on regular expressions. Per definition, these are strings that describe a set of patterns without listing them all, according to certain syntax rules which provide a lot of flexibility in a search. They are commonly used in computer programs that deal with texts in order to manipulate strings efficiently. For instance, the expression "`^(ex|s)amples?$`" matches the words "example" or "sample" in singular or plural, when these are not followed or preceded by any other character. However, writing regular expressions is a quite difficult task for unexperienced users and even though tools exists that simplify the work, the method loses efficiency when queries becomes too large or too fuzzy.

For the implementation of regular expressions, finite state automata can be designed to recognize patterns in strings. As seen in Fig. 2.9, a deterministic finite state automaton that matches the patterns "sample", "samples", "example", and "examples" includes eleven states with one input and one successful output state. Depending on the character read, a match causes a transition to the next state and a mismatch exits a global failure stopping the search process.

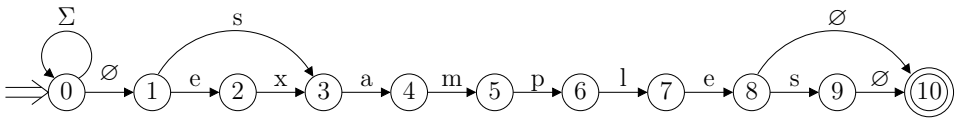


Fig. 2.9. A deterministic finite state automaton for the recognition of the expression "`^(ex|s)amples?$`". Each arrow consumes one input and ∅ stands for any spacing character.

With this method, we can dispose of a system for flexible retrieval that permits an approximate pattern matching in strings to the cost of a dynamically programmable state machine. However, the complexity linked to a hardware realization is hardly measurable as it depends on the way the reconfigurability is supported in the architecture. In this case, the degrees of freedom owned by the user to define a query impact the quantity of resources necessary for the implementation of the finite state automaton in terms of memory and computational power.

2.3 Dynamic Programming

As a technique to facilitate the solution to sequential problems, dynamic programming progressively builds a set of scores utilizing the decision made at a given stage of the algorithm as the conditions governing the succeeding stages [Cor01, Sed88]. Based on an algorithm presented by Wagner *et.al.* [Wag74], this principle can be used to calculate the edit distance between two strings, as depicted in Fig. 2.10. For this, it is necessary to define a cost function $c_{i,j}$ at the position (i, j) in the matrix of results, where $c_{i,j}$ depends on the previously calculated costs and on the local distance $d(x_i, y_j)$. An exemplary cost

M_1	∅	m	a	t	c	h	i	n	g
∅	0	1	2	3	4	5	6	7	8
m	1	0	1	2	3	4	5	6	7
e	2	1	1	2	3	4	5	6	7
a	3	2	1	2	3	4	5	6	7
n	4	3	2	2	3	4	5	5	6
i	5	4	3	3	3	4	4	5	6
n	6	5	4	4	4	4	5	4	5
g	7	6	5	5	5	5	5	5	4

M_2	∅	G	A	A	T	T	C	A	G	T
∅	0	1	2	3	4	5	6	7	8	9
G	1	0	1	2	3	4	5	6	7	8
G	2	1	1	2	3	4	5	6	6	7
A	3	2	1	1	2	3	4	5	6	7
T	4	3	2	2	1	2	3	4	5	6
C	5	4	3	3	2	2	2	3	4	5
G	6	5	4	4	3	3	3	3	3	4
A	7	6	5	5	4	4	4	3	4	4

Fig. 2.10. Examples of the calculation of the edit distance using the method of dynamic programming for textual string matching in the matrix M_1 and DNA sequences in M_2.

function that computes the edit distance between two character strings $x_{1...i}$ and $y_{1...j}$ can be defined by

$$c_{i,j} = \min\left(c_{i-1,j-1} + d(x_i, y_j),\, c_{i-1,j} + 1,\, c_{i,j-1} + 1\right) \qquad \text{with} \quad c_{0,0} = 0 \tag{2.3}$$

where $d(x_i, y_j) = 0$ if $x_i = y_j$ else 1, according to [Nav02]. A parameterization of (2.3) is possible using weighted costs for the penalty of a substitution, an insertion and a deletion [Blü00a]. However, due to the symmetry of the problem, the insertion in one string is equivalent to the deletion in the other.

The path highlighted in the two examples of Fig. 2.10 that apply (2.3) is not exclusive since alternative solutions also result in a maximal global edit distance of four for both M_1 and M_2. Depending on the implementation, the complexity of the method varies from $O(n)$ time using $O(m)$ processors up to $O(mn)$ with one single processor. One advantage of the method resides in the absence of any preprocessing step. However, pattern matching based on dynamic programming has the major problem of being inapplicable for searching small or non-aligned patterns in large texts. Even though a string might be entirely included in another one, the length difference is directly estimated as a mismatch. Therefore this method is more appropriate for searching keys or items in a list of formatted words than for querying wildly in a pool of textual unsorted information.

2.4 Hash Functions and Text Signatures

Hashing is used as a table search method in which the position of a key within the data structure is computed directly from the value of the key using a hash function. Once addressed, data linked to a key can be directly accessed providing a constant time $O(1)$ lookup on average, overcoming the $O(\log n)$ time complexity yield by a retrieval in binary search trees. From the practical point of view, hashing techniques always imply a compromise between time complexity and area consumption. Formally, the retrieval method applies a function h to find an entry key x assumed in our case to be character

string, defined as

$$h(x) = \big(x[0] \cdot B^{m-1} + x[1] \cdot B^{m-2} + \cdots + x[m-1]\big) \bmod W \tag{2.4}$$

where m is the length of the key, B a number that serves as a polynomial basis and W the width of the hash table. Usually, W is an arbitrarily large prime number, the goal of which is to obtain a fairly uniform distribution of $h(x)$ over W for any x. The computation of $h(x)$ which gives the position of the entry key x in the table with $0 \leq h(x) < W$ can be performed in $O(1)$ time complexity [Nav02]. A well known example is the Karp-Rabin (KR) algorithm [Kar87] which uses hashing to seek a pattern within a text. Its average and best case running time is $O(n)$, but the worst case performance is $O(nm)$, where m and n stand for the length of the query Q and the text T respectively. In any hardware or software implementation however, as the table needs to be stored in a physical memory with a fixed size, W is limited and h never remains an injective application. For this reason, the use of this method leads to collisions or false drops which need to be resolved further on. Many ways are found in the literature to deal with collisions, as for example resolution using chained lists, double hashing, open addressing, or in some cases even collision ignoring when applicable. For a more detailed analysis of various searching algorithms suitable for string matching in general, we refer to the published work of Baeza-Yates *et.al.* [Bae99], Cormen *et.al.* [Cor01], Knuth [Knu97], Navarro *et.al.* [Nav01, Nav02], Sedgewick [Sed88], Witten *et.al.* [Wit99] and many more [Col98, Die94, Lom83].

Based on such hashing techniques, signature files are index structures which use a hash function mapping text features, e.g., n-grams, to a bit mask of W bits. The encoding is applicable to a query string Q or a text T from a collection, and consists in regrouping all the hash values $h(x)$ that were extracted from Q or T within one single signature. Hence the resulting file is formed by the sequence of bit masks of all texts T_i plus an entry pointer e_i to this text. The typical search operation can be carried out by comparing signatures s_Q and s_T of Q and T bitwise. The Hamming distance $d(Q,T)$ between the masking $(s_Q \& s_T)$ and the query signature s_Q corresponds to the degree of matching of Q in T and can be estimated using simple logical operations through

$$d(Q,T) = \text{bitcount}\big\{\,(s_Q \;\&\; s_T) \oplus s_Q\big\} \tag{2.5}$$

where & and $\oplus$ symbols correspond to the bitwise AND and XOR respectively. The AND operation masks the positions in s_T which must match the ones in s_Q. The XOR operation isolates the bits in s_T after the previous bit-masking where zeroes were found instead of ones. Using simple boolean arithmetics, we can simplify (2.5) to

$$d(Q,T) = \text{bitcount}\big\{s_Q \;\&\; \overline{s_T}\big\} = \sum_{i=0}^{W-1} \overline{s_T[i]} \quad \forall\; s_Q[i] \neq 0 \tag{2.6}$$

where $\overline{s_T}$ represents the bitwise negated signature of the text T and $s_T[i]$ the bit at the position i in this signature with $0 \leq\; i < W$. Even though a thus calculated zero-distance

cannot ensure an exact matching due to the probability of collision, this method is very fast. For inverted files, if the risk of false drop is accepted within a given application, the necessary traversal verification [Bae99] can be avoided.

3 Sorting Techniques and Algorithms

As they both fundamentally imply comparing operations, it is safe to say that sorting and searching are closely related to each other. In [Knu97], Knuth provides an excellent reference of classical computer techniques for sorting as a very comprehensive and detailed survey. We would like to refer to this resource explicitly as Knuth's seminal work has been widely used to build this section. Also thoroughly reviewed by Sedgewick [Sed88], Martin [Mar71] and Cormen *et.al.* [Cor01], sorting algorithms represent an important part of our work, therefore the most significant ones are reviewed in this section.

3.1 Sequential Sorting

Here we consider that records consist in a body of data plus a key which is used to control the sorting. Per definition, an algorithm for sorting is a procedure that rearranges a file of records so that these are in ascending or in descending order. Before presenting fast networks for sorting in the next section, we review the elementary procedures that will help understand the basic concepts and optimize other algorithms later within our work.

3.1.1 Insertion Sort

Probably the most obvious method for ordering playing cards, insertion sort consists in a repetitive operation of inserting a record from an unsorted list at its final relative place in a new sorted list until no record is left in the old list. This algorithm constructs the final result sequentially, one element at a time.

Algorithm I *(Insertion Sort).* Records R_1 to R_N are rearranged in place such that their keys K_1 to K_N will be in order ($K_1 \leq \cdots \leq K_N$) after sorting is complete. Considering virtually that $R_0 = -\infty$ and $R_{N+1} = \infty$, let i be the index pointing to the record currently processed.

I1. [Loop on i.] Perform steps I2 through I5 for $i = 2, \ldots, N$.

I2. [Set up R.] Set temporary record $R \leftarrow R_i$.

I3. [Find position j.] Select j such that $K_j \leq K_i < K_{j+1}$.

I4. [Move records.] Shift one place $\{R_{j+2} \cdots R_i\} \leftarrow \{R_{j+1} \cdots R_{i-1}\}$.

I5. [Insert record.] Set $R_{j+1} \leftarrow R$. ∎

It should be noticed that algorithm I includes one macro-step I3 which consists in finding the position of the record to be inserted in the sorted list. Depending on how it is implemented, this operation may require many comparisons to find the position in the sorted list. Regarding step I4, it may be possible in hardware to move a part of the records in $O(1)$ time using a dedicated shift register, whereas in software all the records which have to be displaced must be taken one by one.

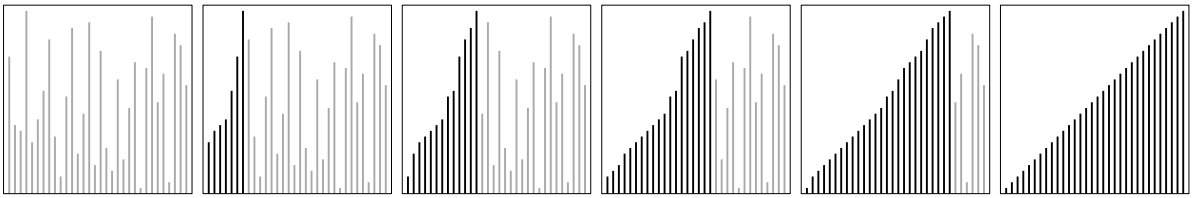


Fig. 2.11. Insertion sort algorithm: Items are selected successively from the left to the right and inserted at their relative final position in the sorted list situated left of the selection.

Fig. 2.11 shows the progression of the insertion sort algorithm at different intervals of time. In the worst case, each step i requires a shift of $i-1$ elements for the insertion so that the runtime complexity remains $O(N^2)$. However, with a given area and a high degree of parallelism, this algorithm could run in $O(N)$ time, e.g., if we are able to find the position of j, i.e., step I3, in $O(1)$ using $O(N)$ comparators.

3.1.2 Selection Sort

The selection method requires all of the input items to be present before sorting can proceed and generates the final output one by one in a sequence. There exist many versions of the selection sort algorithm [Knu97], however we propose one which is based on the selection of the smallest element of a list first, then the second smallest, etc., as illustrated in algorithm S.

Algorithm S *(Selection Sort).* Records R_1 to R_N are rearranged in place such that their keys K_1 to K_N will be in order ($K_1 \leq \cdots \leq K_N$) after sorting is complete. Let i be the index pointing to the last record currently in place.

S1. [Loop on i.] Perform steps S2 and S3 for $i = 1, \ldots, N-1$.

S2. [Find minimum.] Select R_m so that $K_m = \min(K_i, \ldots, K_N)$.

S3. [Exchange.] Interchange $R_i \leftrightarrow R_m$. ∎

The selection sort algorithm is practically the opposite of Algorithm I whereas it requires fewer data movements than the insertion sort but more comparisons. It has the same runtime for any set of N elements independently of their values or order, since finding the minimum or the maximum element of a list of N elements requires $N-1$ comparisons. Its final runtime complexity remains $O(N^2)$. Fig. 2.12 shows this procedure in time, from the leftmost to the rightmost snapshot, where the items are placed from the smallest to the largest.

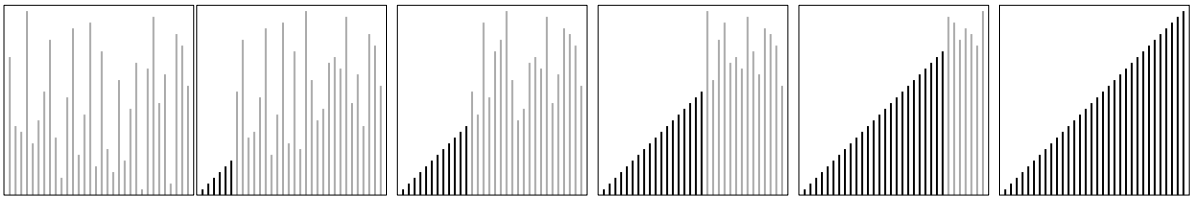


Fig. 2.12. Selection sort algorithm: Items are selected, from the smallest one to the biggest one, and placed progressively at their final position until no item remains unselected.

3.1.3 Exchange Sort

Arranging a file of N records R_1 to R_N according to the bubble sort method consists in repeating a sequence of exchange operations that interchange adjacent records R_i and R_{i+1} if their keys K_i and K_{i+1} are out of order, with $1 \leq i < N$. In this sequence, records with large keys move up and the record with the largest key will become R_N. Repetitions of the process will get appropriate records into positions $R_{N-1}, R_{N-2}, \ldots$, so that all records will ultimately be sorted [Knu97]. After each sequence, all records above and including the last one to be exchanged are in their final position and do not need to be examined on subsequent passes.

Algorithm B *(Bubble Sort).* Records R_1 to R_N are rearranged in place such that their keys K_1 to K_N will be in order ($K_1 \leq \cdots \leq K_N$) after sorting is complete. Let k be the index of the highest record not in its final position, beginning with $k = N$.

B1. [Initialize.] Set $k \leftarrow N$.

B2. [Loop on i.] Set $j \leftarrow 0$. Perform step B3 for $i = 1, \ldots, k-1$.

B3. [Loop on j.] Set $j_{\max} \leftarrow 0$. Perform step B4 for $j = 1, \ldots, k-1$.

B4. [Compare.] If $K_j > K_{j+1}$, interchange $R_j \leftrightarrow R_{j+1}$ and set $j_{\max} \leftarrow j$.

B5. [Return.] If $j_{\max} = 0$ terminate else set $k \leftarrow j_{\max}$ and return to step B2. ∎

One of the most interesting properties of this algorithm is that the highest values are sorted first, so that in i passes, at least the i records with the highest keys are in place. In the worst case, each of the N elements requires to bubble up or down between 1 to $N-1$ places, such that the runtime complexity yields $O(N^2)$. Fig. 2.13 depicts the progression of the records during a sorting sequence at different time intervals. Even though the algorithm can terminate in less than N^2 steps, it remains inefficient and is rarely used. However, the bubble sort can be easily turned into a parallel sorting network, as we will see in Sec. 3.3.

Based on a divide-and-conquer strategy, quicksort [Hoa62] is probably the most used sorting algorithm in computerized applications. After having chosen a pivot element, a partitioning of the data set relatively to the pivot splits the file of records into two subsets. The procedure is applied recursively until all the records are in nondecreasing order. As the most complex issue in quicksort algorithm, the pivot has to be chosen as the median value in order for the algorithm to work in $O(N \log N)$ against $O(N^2)$ for an

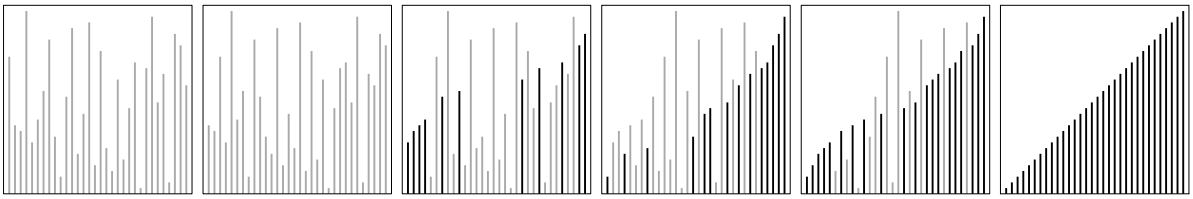


Fig. 2.13. Bubble sort algorithm: Adjacent items are repeatedly compared and interchanged so that the records with the largest key move to the right and vice versa.

infortunate choice. Beside its low average time complexity, one of the big advantages of this method remains that only a small amount of memory is necessary to perform the partitioning when implemented in software.

Based on the insertion sort, shell sort [She59] improves sorting time by comparing and exchanging elements separated by a gap of several positions. Without any explicit rule, the length k of the gap is reduced at every pass on the list of records such that it reaches unity at the end. During this procedure, the data set gets k-sorted and consists in k independent arranged sequences until $k = 1$. Even though the optimal sequence of increments remains an unresolved mathematical problem [Knu97], the running time of the algorithm has been proven to be $O(N^{3/2})$.

3.2 Bit-Level Structures

As opposed to the so far described word-oriented sorting procedures, bit-level algorithms, such as the radix-exchange sort, rank-order or median filter, make use of the binary representation of the keys they have to arrange. The median filter belongs to the class of nonlinear rank order filters and is frequently used in various signal and image processing applications [Hen98, Yin96]. However, depending on its implementation, it may finally not be considered as an actual sorting algorithm, since the processed data set is subject to intrinsic modifications during the filtering, and since the output is by definition a single value.

According to the literature, there exist different algorithmic forms of the radix sort [And94, Cor01, Knu97, Sed88] which are generally classified into two categories, i.e., Least Significant Bit/Digit (LSB/D) and Most Significant Bit/Digit (MSB/D) radix sorts. The sorting principle however remains the same. We describe in this section a stable binary radix sort using LSB first. Based on bit-level manipulations, Algorithm R includes two phases. The first one is the partition phase, in which records having the same value for a given bit-position are consecutively placed into the same bucket. The second one is the gathering phase, where the buckets are combined together, according to the chosen priority rule, i.e., lowest first in this case. The run-time complexity of Algorithm R is in $O(NM)$, where N is the size of the data set and M the length of a key, expressed in digits but usually in bits on binary computers. As opposed to the standard key comparison and exchange operation, e.g., as seen later in Fig. 5.17, it owns the particularity to work in backward mode, i.e., LSB first.

Algorithm R *(Binary Radix Sort).* Records R_1 to R_N are rearranged in place such that their keys K_1 to K_N of length M will be in order ($K_1 \leq \cdots \leq K_N$) after sorting is complete. Let B_1 and B_0 be two temporary buckets in which records can be stored successively with a First-In First-Out (FIFO) behaviour. Let i be the currently processed bit-position within a key and j the index of the currently processed record.

R1. [Initialize.] Set $i \leftarrow 0$.

R2. [Loop on j.] Perform step R3 for $j = 1, \ldots, N$.

R3. [Partition.] If $K_j[i] = 0$ put R_j in B_0 else in B_1.

R4. [Concatenate.] Combine B_0 followed by B_1 into a new list of records.

R5. [Repeat.] Set $i \leftarrow i + 1$. If $i = M$ terminate else return to step R2. ∎

By contrast, Fig. 2.14 depicts the evolution of the list of records during a sorting sequence at different time intervals when the records are compared MSB first. Even though the algorithm can terminate in less than $N \times M$ iterations, it remains a sequential algorithm that compares well to Quicksort [Hoa62] which is also based on the idea of partitioning.

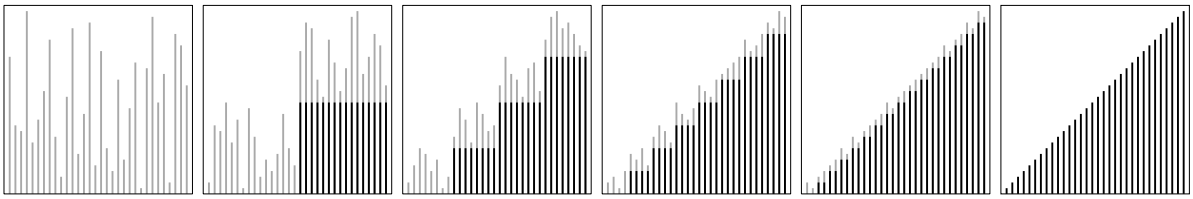


Fig. 2.14. Radix-sort algorithm: Based on a partitioning scheme, the keys are sorted progressively starting from their MSBs first. Here, the set of $N = 32$ keys is partitioned recursively, using up to 5 bits for the comparison.

Nonetheless, radix sort has many drawbacks when it comes to hardware implementation. First, the body of the algorithm consists in moving data back and forth, from the list to one of the two buckets and vice versa. The size of the underlying data transfer structure will badly scale up with the length and the amount N of records in the data set, as opposed to a processor based software implementation where only the size of the required memory would be affected. Second, even though the records are finally rearranged in the initial list, the algorithm does not run "in place" since a double file, i.e., the buckets, is needed to store the records temporarily outside of their original memory space during processing. Moreover, the size of each bucket is determined at run-time and can reach N in the worst case. Third, since the order and the final output position of each record is not known during the partition phase, algorithm R cannot be directly implemented as a sorting network where parallelism would be efficiently exploited.

3.3 Sorting Networks

Sorting networks are decision structures that take into account the possibility to process many records at the same time, involving only simultaneous nonoverlapping comparisons and exchanges. Due to the intrinsinc parallelism of a network, the sorting time can be reduced according to the resources used in the implementation. The concurrent access to the data set to be arranged is a prerequisite that makes sorting networks faster than sequential procedures. However, in most of the cases, the size of the data set N cannot be changed dynamically, as opposed to software solutions, and each network owns a maximal sorting capacity.

3.3.1 Odd-Even Merging Algorithms

Per construction, a sorting network consists in comparator steps executing comparisons and exchanges of exclusively distinct records in parallel. The number of steps within the network determines its time complexity whereas the area complexity is linked to the amount of comparisons. Accounted as the most trivial construction possible, the bubble sort network can be considered as the parallelization of Algorithm B on page 27 where comparisons and exchanges are performed pair-wise on adjacent records. For sorting a set of N records, its time complexity is in $O(N)$ and its size in $O(N^2)$. According to text's conventions for drawing sorting networks [Knu97], Fig. 2.15 represents two networks that respectively implement the bubble sort and the odd-even transposition sort algorithms.

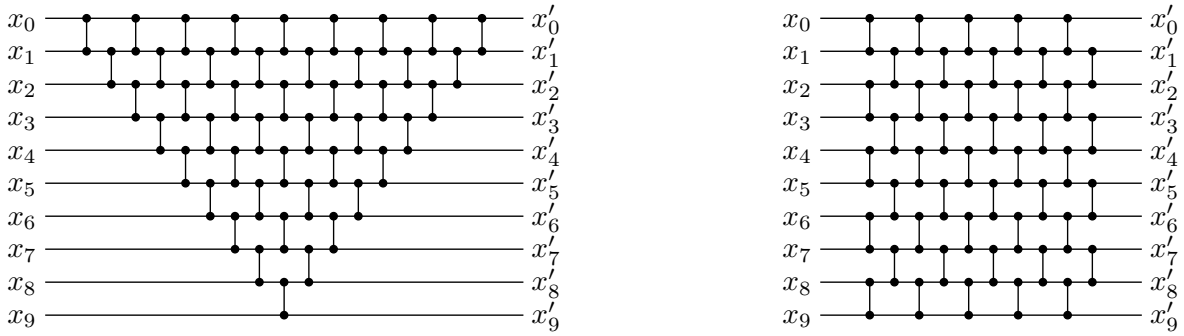


Fig. 2.15. Network implementation of the bubble sort (left) and odd-even transposition (right) algorithms represented in a Knuth diagram for an input of $N = 10$ keys.

The odd-even transposition network can be seen as an improved version of the bubble sort in that the processing time is half as long, since the comparison sequence starts with all the records from the beginning on. Using the zero-one principle [Bat68, Knu97], one can easily prove the correctness of these networks.

3.3.2 Bitonic Sorting Algorithms

One of the best known sorting algorithms is Batcher's bitonic merger and sorter [Bat68] which was discovered in 1968. This algorithm is based on a divide-and-conquer strategy

transforming recursively two bitonic sequences into one monotonic sequence. Per definition, a bitonic sorter of order N is a comparator network that is capable of sorting any bitonic sequence of length N into nondecreasing order, whereas a sequence $\langle x_1, \ldots, x_N \rangle$ is bitonic if $x_1 \geq \cdots \geq x_k \leq \cdots \leq x_N$ for some k, $1 \leq k \leq N$ [Knu97]. The problem of merging $x_1 \leq \cdots \leq x_N$ with $y_1 \leq \cdots \leq y_N$ is a special case of the sorting problem, since merging can be done by applying a bitonic sorter of order $2N$ to the sequence $\langle x_N, \ldots, x_1, y_1, \ldots, y_N \rangle$.

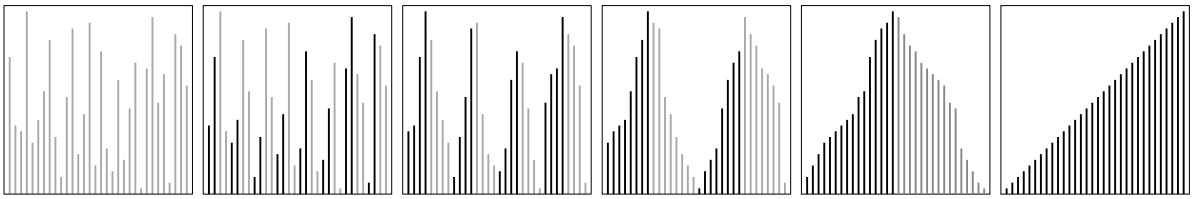


Fig. 2.16. Batcher's Bitonic sort algorithm: In this set of $N = 32$ records, adjacent items are progressively 2^i-sorted, with $0 \leq i \leq \log_2 N$.

Given as an illustration for the sorting of $N = 32$ records according to the bitonic algorithm, Fig. 2.16 depicts the evolution of the set of data at different time intervals after the merging of 2^i adjacent records, with $0 < i \leq \log_2 N$. One merging step of the network, as represented in a Knuth diagram in Fig. 2.17, transforms one bitonic sequence $\langle x_0 \ldots x_7 \rangle$ into one monotonic sequence $\langle x'_0 \ldots x'_7 \rangle$. For convenience reasons, we have chosen N as a power of two, even though arbitrary sequences of any length can be sorted using this iterative sorting-by-merging rule. The bitonic scheme for sorting $N = 2^n$ elements requires $\frac{n}{2}(n+1)$ levels of $\frac{N}{2}$ elements each with a corresponding complexity in $O(\log^2 N)$ time and $O(N \log^2 N)$ for the area.

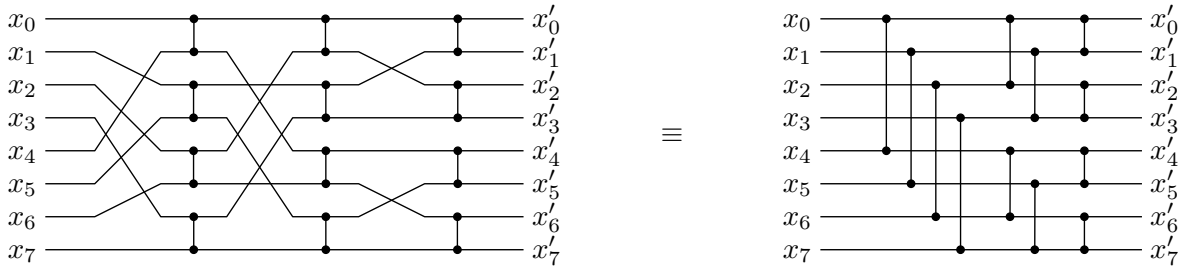


Fig. 2.17. Network implementation of the bitonic merging algorithm represented in a Knuth diagram for an input of $N = 8$ keys in two different but equivalent ways. This network transforms one bitonic sequence $\langle x_0 \ldots x_7 \rangle$ into one monotonic sequence $\langle x'_0 \ldots x'_7 \rangle$.

3.4 Summarization

Sorting algorithms used in computer science are often classified by computational complexity regarding the size of the list of records which have to be ordered. As we have seen in this section, though many different sorting algorithms have been invented, studied

and improved, only a few of them are suitable for a low-level hardware implementation. Ajtai *et.al.* have presented in [Ajt83] an algorithm working in $O(\log N)$ delay time for a sorting network processing N keys with a complexity in $O(N \log N)$ which however remains impracticable due to its large hidden constant factor. Furthermore, Leighton's algorithm presented in [Lei85] for sorting N keys in $O(\log N)$ time with N processors is not appropriate for hardware realizations and is much slower in practice than other parallel sorting algorithms. Nonetheless, Batcher's sorting techniques [Bat68], even renamed otherwise sometimes [Sch89], seem to be the most suitable for actual implementations in hardware or in multi-processor systems and mesh connected computers.

Chapter III

Related Work

SEARCH engines are normally always linked to one or many data containers in which the requested information has to be found. The ensemble of containers constitutes a database, the size of which may vary over many orders of magnitude. It might be either a tiny directory located on one's computer desktop, or the World Wide Web (WWW), an unlimited ever-growing collection of online documents on Internet servers worldwide. Hence, this chapter begins by presenting a typical Internet search engine as well as the standard string matching utilities within computer systems. The second section of this chapter deals with the description of Lapir's Associative Access Method (AAM) which relies on an index built on text signatures. Finally, the third section summarizes related work in the area of hardware search engines and associative processors.

1 State of the Art in Software

The aim of this section is to give an overview on the complexity of some typical search applications that run at the software level using standard processors as support platform. These include Internet search engines as well as utility programs on standard Personal Computers (PC).

1.1 Web Search Engines

The amount of information on the WWW is growing rapidly, as well as the interested amount of users performing any kind of research. After a very long evolution time, there subsist mainly two types of Internet search engines, i.e., the human and the automatically maintained ones. On the one hand, IR systems based on human maintained indices provide a high quality web link graph. They cover popular topics effectively but

are subjective and expensive to build and to maintain. On the other hand, automated search engines rely on keywords and usually return too many low quality matches. Their index provides a view of the retrieval problem which is much more related to the system itself than to the user need [Bae99, Kob00].

Given as an example, Google began as an academic research project and was developed by Brin and Page [Bri98] at Stanford University becoming its official search engine. Their system uses spider programs that traverse the hypertext structure of the web by recursively retrieving all documents automatically, starting from a random address. Supported by huge computational power, Google remains a large scale Internet search engine which makes heavy use of the structures present in web documents to produce and sort results according to a patented page ranking technique [Pag98]. The citation and link graph helps determine how well-established a page is, mainly by calculating how many backward links point to it. Hence Internet search engines profit from their multi-user accessibility to help classify the results according to their popularity.

1.2 Software Functionalities for Computers

Sorting and searching are the two most important operations related to the management of an Operating System (OS). Not only is it a necessity to be able to search a file or a directory, but also to find information inside a file. As a very representative example, every Unix-based OS provides various utility programs to help users work efficiently with text files, such as `locate` which comes with an index database, i.e., a file which is created locally on the hard disk drive of a computer and which is automatically updated. The database also stores file permissions and ownership so that users will not see files they do not have access to. The search pattern is a plain string that can contain metacharacters and the output is a list of file names that contain the pattern. Though `locate` can only display the file names that match the pattern exactly, regular expressions, as seen in Sec. 2.2 in Chapter II, can be used for approximate matching. Another example is the command `find` that searches the directory tree rooted at the issued argument for a given file name. Without using an index, `find` is much slower than `locate`, however no index update is necessary. A related command is `grep` which searches files for lines that match a given pattern or any extended expression. Though a little slower than exact matching programs, its variant `agrep` has the ability to search for approximate patterns with a large number of options. Depending on the querying context, it selects one of its built-in Levenshtein distance based algorithms to maximize the speed. Moreover, many Unix programs such as `awk`, `diff`, `sed`, `sort`, `strings`, `vi`... permit an efficient text processing with find and replace functionalities including pattern scanning and editing through the use of regular expressions and meta characters. For more details and an exhaustive list of options, it is practically wise to refer to the Unix/Linux manual pages (`man`) distributed with the respective programs.

2 Hardware Accelerators

As the era of the billion transistors architecture is well established [Bur97a, Bur04], the major portion of the transistor budget of a processor is still used for cache memory. This is mainly due to the fact that processor clocks are nearly twenty times faster than memory device clocks. In order to overcome the problem of the memory wall as introduced in Chapter I, different research groups have come to many architecture proposals that we want to present briefly in this section. The relation to our work is globally the concern with the need of a rapid access to a big memory space and the necessity to create an efficient processing system according to earlier architectural decisions.

2.1 Associative and Parallel Processors Systems

With reference to the literature, as an alternative to traditional SIMD and MIMD systems, associative processors [Bat76, Cut78, Kun88, Ruh85, Tav94] are particularly suitable for massive data searching and manipulation [Kri94, Par73a, Par73b, Sch92]. They are typically composed of an array of Processing Elements (PE) having their own associated memory which respond to a single or multiple instruction streams. Developed at Kent State University, the Multiple Associative Computing (MASC) model [Pot94, Wal01] is an associative MIMD system which consists in an array of 8-bit RISC PEs with local memory, and a smaller set of control units sharing a bus to every cell. Depending on the executed program, PEs can be reallocated and execute associative operations according to a small instruction set [Wan03]. The scalable architecture, as shown in Fig. 3.1-a at a high level of abstraction, has been implemented on FPGAs for different sizes, making it well-suited for expansions in terms of specifications and functionalities. It provides a high flexibility through its programmability but lacks in data throughput as no external mass memory is available.

Developed especially for string matching and sequence alignments in computational biology, the Kestrel datapath from the University of California at Santa Cruz consists in an SIMD parallel coprocessor system with a linear array of hundreds of 8-bit PEs distributed over eight IC devices. Fig. 3.1-b gives a schematic representation of the processing system where systolic shared registers are placed between adjacent PEs to allow flexible data processing. Kestrel suffers since its inception of a low clock frequency caused by the absence of pipeline registers in order to maintain a certain programming intuitiveness [DiB05]. However, due to its highly parallel architecture, the ASIC realization provides a consequent speed-up against general-purpose processors clocked more than one order of magnitude faster.

Work performed in the Galileo project at the University of Wisconsin at Madison includes memory issues and multiprocessor system design [Bur97b]. Especially for parallel computing, they examined the impact of embedded memory on processor as memory bandwidth limits the performance of standard processors. In order to gain performance

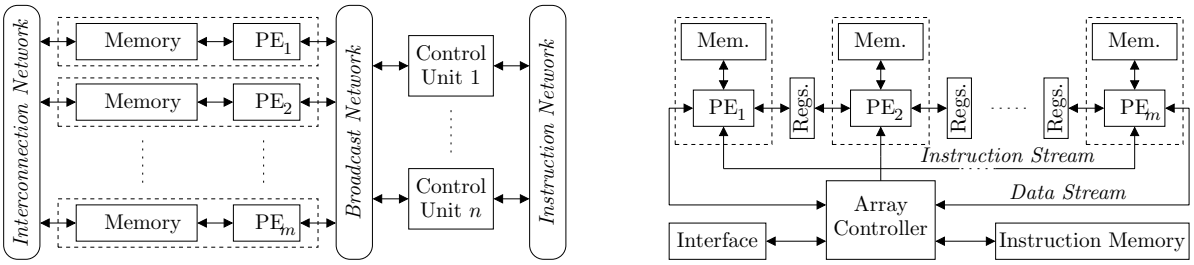


Fig. 3.1. Architecture of different associative processors based on **a)** an MIMD computing model with PEs controlled by multiple instruction streams as in [Wal01] and **b)** an SIMD computing model with PEs organized in a linear array as in [DiB05].

in general-purpose processors, SRAM caches exploit the spatial and temporal locality of the data and tend to compensate the devastating latency of standard DRAMs. With a rising tendency, they occupy from 50% up to 80% of the processor die [Koz02]. However, for streaming applications such as multimedia signal processing or data mining, the cache principle remains inappropriate. Associative and parallel processors present a presumably very suitable architecture for the implementation of string matching algorithms with an inherent structural concurrency. However, even by reducing the programmability to a minimum set of special instructions, e.g., [Fau91, Vui96], the matrix of PEs within an associative processor will always suffer the trade-off between flexibility and data throughput. Therefore one calls for more efficient architectures.

2.2 Merging Logic and Memory

Since more than a decade, the purpose of embedded memory design is to produce new kinds of architectures which provide better performance than standard processors as found in personal computers. The idea is to benefit from a higher bandwidth by eliminating the narrow external data bus, as well as a lower latency by customizing the organization of memory banks. However, the pairing of storage and computing is not trivial, since DRAM and logic processes are fundamentally different. As seen in Sec. 1.4 in Chapter II, DRAM is optimized for minimum cost and maximum capacity. Its process produces thick-gated low-leakage transistors, whereas a logic process aims for thin-gated fast switching ones.

Predicting in the Intelligent RAM (IRAM) project [Pat97b, Per99] at the University of California at Berkeley that logic and memory will be merged onto a single chip to remove the performance gap in future computer architectures, they investigate the placement of a processor along with DRAM on a single chip. Furthermore, the integration of an SIMD array of PE in the DRAM provides a high bandwidth for vector processing with a low power consumption [Koz00]. Targeting the consumer market and portable devices in particular, long term development aims to reduce the overall costs by eliminating the expenses of external discrete components providing a System on Chip

(SoC) solution. However, due to the fabrication process, embedded DRAM cannot be as dense as off-chip DRAM and still remains more expensive than on-chip SRAM [Koz02].

Similarly, the Reconfigurable Architecture Workstation (RAW) processor [Wai97] developed at the Massachusetts Institute of Technology consists of a square reconfigurable interconnected matrix of pipelined RISC processors with distributed SRAM instruction and data caches on a single chip. The emphasis was placed on a fine repartition of the different logic and storage parts over the chip instead of having a large centralized memory. In addition to the important amount of external interfaces available, the efficiency of the processor is mostly due to the quality of the software tools which must be able to provide partial communication reconfiguration of the architecture statically at compile time and dynamically at run time. While RAW has more memory per chip, Asynchronous Simple Array of Processors (ASAP) [Yu06] developed at the University of California at Davis has a larger number of pipelined RISC processors on a single die which communicate asynchronously through FIFO buffers. Individually clocked, the PEs can adapt their working frequency to fill the computational needs and reduce the overall power consumption. However, this hardware overhead is neither negligible nor justified for a single application processor.

The Smart Memories Project (SMP) from Stanford University aims to implement a single chip multi-processor system with coarse grain reconfiguration capabilities supporting different application mappings [Mai00]. Its regular architecture is made up of many processing tiles, each containing local memory, local interconnects and a processing core, as well as on-die SDRAM tiles. According to the requirements of the application, the tiles are connected through a dynamically routed network. Flexibility remains a key element of the design to support regular data-parallel stream-based applications as well as irregular memory access pattern. Concentrating on the memory accesses, the Impulse project at the University of Utah in Salt Lake City aims to build a configurable main memory controller including embedded DRAM that increases processor cache and system bus utilization [Zha01]. The main benefit of the architecture is due to a physical address remapping that makes memory accesses appear consecutive as in a burst mode and hence reduces the penalizing latency.

Many other research groups have recognized the advantages of merging memory and logic onto the processor die such as the Processing in Memory (PIM) project [Sun96] from University of Notre Dame in Indiana. Combining processor and memory macros on a single chip basically always implies an array of PEs, connected together over various network topologies with DRAM, that perform SIMD or MIMD operations in parallel. In the most frequent cases, these chips can be chained to form a multidimensional processing system, depending on the computational needs.

The purpose of embedded logic onto memory devices is, as opposed to embedded memories, to build commodity DRAM able to perform some processing locally. Developed at the University of Illinois at Urbana-Champaign, FlexRAM [Yoo00] aims to place a superscalar RISC processor onto a 64MB DRAM device. The chip targets its insertion inside a workstation or a server, replacing the main memory in order to unload the processor for the memory-intensive parts of the applications, such as data min-

ing or multimedia processing [Kan99]. Beside the compatibility with general purpose processors, FlexRAM achieves low programming costs, so that even though internal reconfigurability is not foreseen, the memory subsystem is able to default tasks to plain DRAM when the application is not enabled for intelligent memory.

A similar approach was adopted by the Computational RAM (CRAM) architecture [Ell92] from the University of Toronto which integrates SIMD processors into a standard DRAM at the sense amplifiers, and by the Data Intensive Architecture (DIVA) chip [Hal99, Dra02] from the University of South California in Los Angeles that benefits from a large datapath enabling data parallelism. Besides, the Reconfigurable Architecture DRAM (RADRAM) project [Osk98] from the University of California at Davis was based upon the integration of DRAM and FPGA technologies to provide programmability at the logic level. Later however, they replaced the reconfigurable logic through a Very Long Instruction Word (VLIW) processor to gain more performance using fine grained parallelism.

2.3 Special Purpose Coprocessors

Special purpose coprocessors are hardware architectures basically designed to achieve extremely high performance for a specific application. Beside mathematical Floating Point Units (FPUs) and graphics accelerators which are the most popular acceleration devices, there is also a need in domains such as cryptography, genome analysis, biosequence mining, network attacks detection, text database searching,..., indeed problems that often reduce to a simple string matching paradigm. Therefore we aim to review a few architectures that seem to address our querying problem at first glance.

The architecture of Blüthgen *et.al.* [Blü00a, Blü00b] is a parallel realization of a coprocessor based on the standard dynamic programming algorithm, as seen in Sec. 2.3 in Chapter II, and can perform approximate string matching for variable edit costs. It belongs to a retrieval system that performs content-based on-line searches in a large database of multimedia documents [Kno98]. Build as an array of 64 PEs, the processor primarily computes the edit distance between an eight characters query word and a text, whereas this limitation can be overcome by chaining the appropriate number of PEs accordingly to the search pattern. Due to the chosen algorithm, no pre-indexing is necessary when documents are stored into the database or when a query is started. As an improvement, text recoding and wildcard handling extend the classic algorithm to a more flexible one. However, for a fault tolerant full-text search engine, the implementation is constrained to a matching at the word level and not at the sentence level. The design provides a high computational throughput rate but was not foreseen for handling huge amounts of data, as only a single SDRAM chip serves as an on-board data and result memory [Blu00]. The overall performance is adapted downwards to the PCI interface which provides the data from the CPU of the system on which the coprocessor board is plugged. Similarly, Sastry *et.al.* [Sas95] described the design and implementation of a linear systolic array chip for computing the edit distance between two strings. However,

as this measure is related to the size length of the query, dynamic programming remains only efficient for short pattern.

Accompanied by a prolific literature, Baker *et.al.* [Bak04, Bak05, Bak06] currently propose the realization of a systolic architecture based on an FPGA design for the implementation of the KMP algorithm as seen in Sec. 2.1 of Chapter II. Focusing on network intrusion detection and aligned text mining, they try to minimize the hardware resources in order to maximize the internal parallelization. The use of FPGAs allows the parameterization of designs at the hardware level, an optimal dimensioning of the internal memory blocks and an easier scalability of the architecture than for ASIC design. However, they do not consider neither the use of approximate matching techniques nor the eventuality of extremely large databases.

Focusing on multimedia and image processing, Stanford's Imagine [Ahn04, Kha02] coprocessor architecture supports ensembles of ALUs organized as tiled SIMD clusters. The computational efficiency of the system is due to a high bandwidth SDRAM interface that can simultaneously feed data to many PEs using stream transfer instead of single load and store operations, and thus exploit the parallelism and locality of streaming media applications. The flexibility of the system is provided by two programming languages developed on this purpose that implement kernel programs for the embedded microcontroller as well as stream level functions for manipulating data. In fact, even though the system could be qualified of general purpose multimedia stream coprocessor, the architectural overhead and the programming complexity for the execution of a single specific task such as string matching would not legitimate the use of such a hardware platform.

2.4 Summarization

String matching algorithms can normally be optimized for approximate text search on general purpose processors [Bae99, Knu97, Nav05, Wu92]. Furthermore, using parallel and associative processors permits the parallelization of the data and the operations during the processing, and results in drastically reduced execution times [Dav86, Lee91, Wu02]. However, despite a very high programming flexibility, these solutions suffer from the rather low performance of the memory subsystem.

Integrating processors and main memory in a single chip is a promising approach to increase system performance. For all the research performed in the last decade, the main goal was to find a way how to overcome the memory wall. Such an integration provides a very high memory bandwidth that can be exploited efficiently by vector operations. As a consequence, proprietary extensions and special compilers have to be built in order to guarantee the usability and the programmability of these systems. However, for custom tasks with critical specifications, it is still possible to trade off flexibility for performance in order to meet the time constraints.

Bringing processing on the storage chip that uses the same interface as conventional memory devices offers an increased memory bandwidth of many orders of magnitude

compared to standard systems. As a consequence, avoiding the delays associated with communicating off chip due to internal memory accesses reduces the overall latency. Multi-processor implementations are even capable of executing multiple threads in the memory. From the practical point of view, functions can be in some cases simply invoked through memory-mapped writes and the results obtained through standard read procedures. However, one of the main difficulties remains to maintain cache coherency between the main processor and the smart memory devices.

Through all the different implementations reported in this section, we have seen that the programmability was still a major concern in the design of the architecture of parallel processors. Even if more bandwidth can be extracted from the memory, the processor-based system may be difficult to program. For most of them, not only the type of application but also the quality of the custom compilers determine the overall performance. By using application specific architectures such as ASICs or dedicated coprocessing FPGA implementations, the efficiency can be maximized in terms of speed, hardware area, throughput and power consumption, at the cost of lower flexibility. In this sense, building a system optimized for a specific application will always be faster than configuring a general machine for that task.

3 Associative Access Method

The Associative Access Method (AAM) belongs to the class of the filtering algorithms. Based on the use of text signatures and binary attributes, it permits a non-exact pattern matching and thus an extremely flexible retrieval. Although it has been developed and first published by Lapir *et.al.* [Ber98, Lap92, Lap02] more than a decade ago, there is still ongoing research and applications [Lay05a, Urb02] relying on this method. In this section, we recall the main functionalities of the AAM and provide the algorithmic basis for the development of the Associative Computing Engine (ACE).

3.1 Building the Signature File

As a probabilistic method for indexing text, the most important element of the AAM is the signature file. It is produced using a hash function $h(x)$ which maps every feature x of a given text T of a page p to bit masks of a constant size W. These features can be of any kind as long as they have only two possible states in the coded object, i.e., present or absent. Our research application focuses on text retrieval where the considered objects are text pages and the features are trigrams.

Per definition, trigrams are ordered triplets of characters from an alphabetic writing system Σ, e.g., "`the`". As a special form of q-gram, they led to the development of

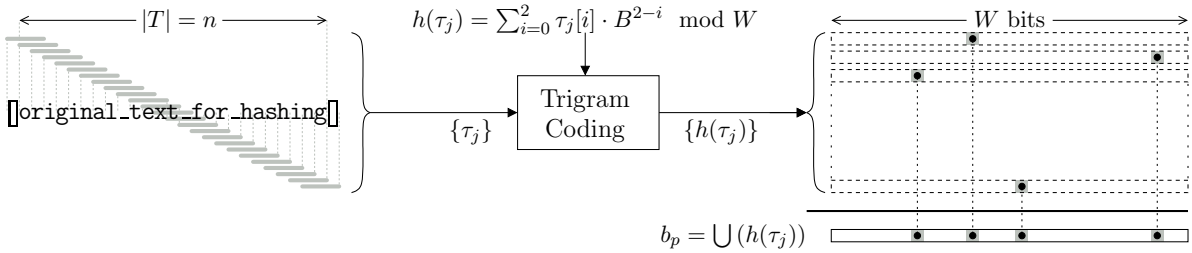


Fig. 3.2. The signature of a page p is created by hashing all the trigrams present in a character string T and computing the union of all the thus obtained bit masks. Considering a non-injective hash function, n bits or less out of W are set in the signature b_p.

various fast methods for approximate string matching that share the principle of filtering [Bur02]. According to (2.4), Fig. 3.2 shows how the signature of a page p is created by hashing all the trigrams present in a character string T and computing the union of all the thus obtained bit masks. Since $h(\tau)$ is not an injective function on purpose, it causes a certain degree of hash collisions so that $n = |T|$ bits or less out of W are set in the signature b_p. The most delicate part of the design of a signature file remains to ensure that the probability of a false drop is low enough while keeping the signature file as short as possible [Bae99].

The Bit Attribute Matrix (BAM) [Lap92, Lap02] is the regular $W \times L$ bit file formed by the superposition of L signatures that constitutes the entry point of our hardware design. Fig. 3.3 represents the off-line procedure of the creation of the BAM which includes two successive steps. First, documents of a textual database are segmented in equally sized pages using ASCII filters which remove images and other meta data [Bee03]. They perform a normalization of the characters, i.e., lowering the case and removing accents for the mapping on a finite alphabet Σ. Then the contents of the texts are stored as a set of binary features in the BAM using the trigram hashing mentioned above. In this matrix, as a consequence of the lossy transformation, no information is available on the position of the binary features in the original text. Once the BAM has been created, the original text is compressed and stored for future use so that the final database consists of the plaintext data and its corresponding binary image.

Corresponding to the retrieval granularity of the AAM, the size of a page and the length W of the signature specify the hit ratio in the BAM, i.e., the density of ones in the matrix. Considering that each of the n trigrams of a page set at most n random bits in a signature, the probability of a hit p_h and a miss p_m are such that $p_h = 1 - p_m \leq \frac{n}{W}$. For a random query Q of length m, the global matching probability yields $(p_h)^m$ and can be used as a comparison criterion to perform a fast page selection. The system architect which uses the AAM has the possibility to parameterize n and W for the construction of the index database. On the one hand, if $n \gg W$, the probability of having retrieval collisions gets very high since $p_m \gg \frac{1}{2}$. On the other hand, if $n \ll W$, the size of the BAM becomes huge, since reducing n or enlarging W respectively increases the number or the length of the necessary signatures. There is a good reason, however, why W should correspond to twice the number of characters within a page. Shannon's entropy

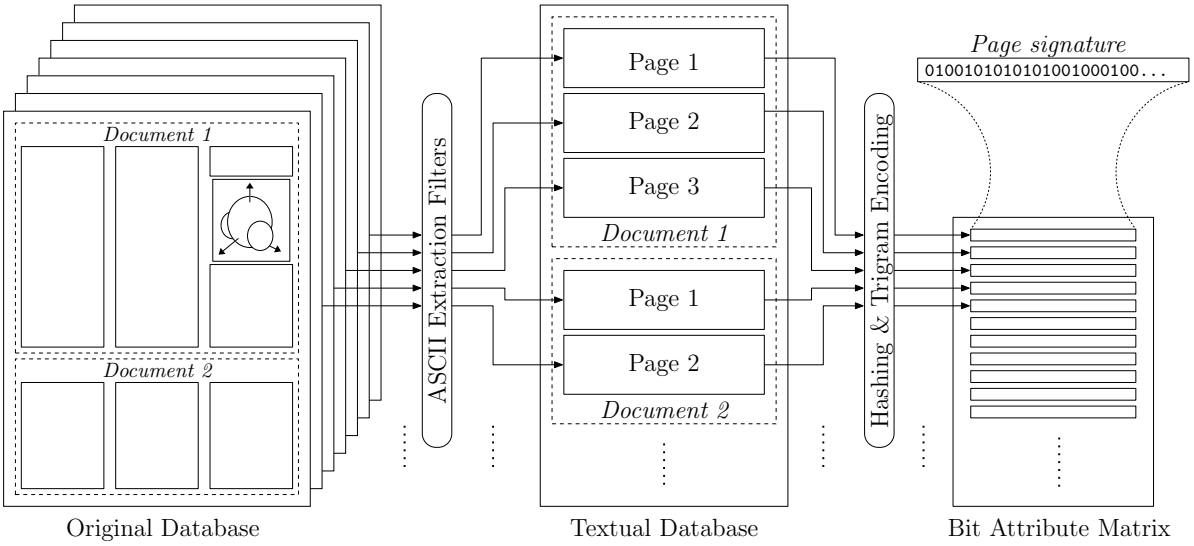


Fig. 3.3. The compilation of the BAM from the original database occurs through a filtered textual database. The hash function encodes equally sized text pages into a fixed length signature vector.

is a common concept used in information theory defined by

$$S = -\sum_{i=1}^{N} p_i \cdot \log_2 p_i \qquad \text{with} \qquad \sum_{i=1}^{N} p_i = 1 \tag{3.1}$$

where p_i is the probability characterizing one of N possible states. It is the lowest bound on the number of bits needed to describe the system and reflects its information content. Entropy is also found in data compression, where one makes use of the non-uniform occurrence of bit patterns in some quantized schemes [Jai89]. An optimal coding of the BAM is achieved when approximately half of the bits are set, since this theoretically conforms to the most efficient arrangement [Cri91]. Moreover, the amount of possible W bits signatures $\binom{W}{n}$ is maximum for $n = W/2$, therefore the density of ones set for the availability of a given trigram is optimized and yields one half.

For example, we can set the retrieval granularity of the AAM to pages sizes of 1kB, i.e., approximatively 1000 characters requiring a hashing in $W = 2000$ bits signatures. Taking $W = 2039$, the highest prime number lower than 2^{11}, as modulo in (2.4) states a very practical reason. Aligning the signatures on the top of each other leads to an efficient occupancy of standard hardware memory devices so that when addressing the position $h(x)$ over 11 bits, only a minimal portion of the address range is left unused. We can express the practical utilization u of a given memory, the width of which is set to $2^{11} = 2048$ bits with

$$u = \frac{2039}{2^{11}} = 0.9956 \qquad \text{(i.e., waste less than 0.5\%)}. \tag{3.2}$$

Nonetheless, for the insertion of supplementary administrative functionalities such as user access restriction or signature validity, the nine $(2048 - 2039 = 9)$ remaining bits are very welcome to be used.

Obviously, the probability of false drops within a signature is very high. If we consider the 26 alphabet symbols equiprobable in texts, the collision factor of each trigram set in the 2039 bits signature is about $26^3/2039 \approx 8.6$. Taking 40 symbols into account for Σ, as we might do later for the functional verification of our work, would even raise it to over 30. However, linguistic properties hidden in documents make the method work fine. On the one hand, depending on the considered language, e.g., English, an interesting fact is that when presented with the letter "`t`" in the initial position of a word, there is a very high probability that the letters "`h`" and "`e`" will follow. Without further measure, this would cause the position of the trigram "`the`" to be always set and thus useless. On the other hand, many trigrams don't even exist in a given language, so that the actual collision factor remains much lower than as calculated above.

In summary, the major advantage of the hashing procedure, when it comes to hardware realizations, is that it drastically reduces the amount of memory needed for the storage of the signature files. The choice of an optimal hash function $h(x)$ implies a trade-off between memory requirements and collision factor. Further discussion remains beyond the scope of the thesis, since such a decision has to be made by the application designer.

3.2 The Retrieval Process

Basically, filter algorithms such as the AAM consist of at least two phases so that a fast but cursory inspection of the signature file is performed at first, selecting potential candidates for the following slower verification [Bur02]. This results in a significant speedup when compared to conventional approximate string matching algorithms.

3.2.1 A Two Phases Search Algorithm

Depicted in Fig. 3.4, the retrieval algorithm includes two matching phases. First the filtering phase returns a list of relevant documents from the BAM using approximate matching, then the evaluation phase computes the relevance factor of the documents regarding the actual query through a more precise matching. These two phases are performed after the preprocessing of the query text Q which includes ASCII filtering, fragmentation of long queries, trigram encoding into query slots and weighting of the words depending on their importance at the semantic level. A typical example of query encoding is given in Sec. 3.2.2.

In the approximate matching phase, the query signature is compared with the attributes of the BAM to identify the records having a high probability of containing the desired query information [Ber98]. In this rough but very fast procedure used to exclude a large number of non-relevant documents from the search, a similarity measure is computed for each page signature to produce what we call page score, as explained in Sec. 3.2.3. Fig. 3.5 depicts different responses from the filtering phase for which the amount R of returned results depends on a threshold value θ set to around two thirds

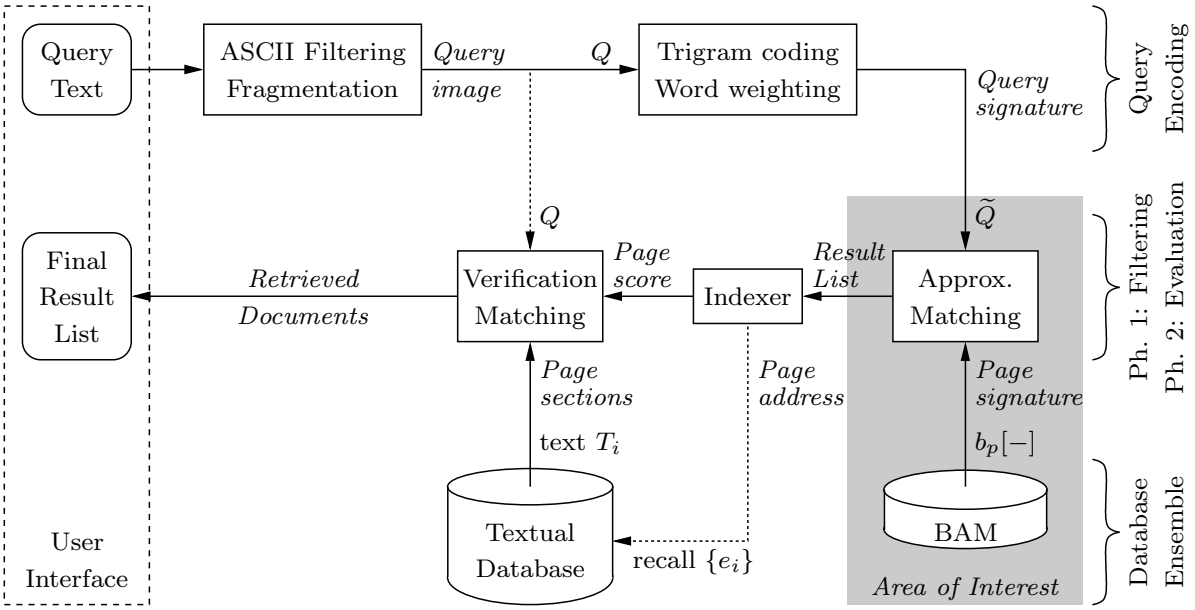


Fig. 3.4. The filtering and evaluation phases of the AAM correspond to an approximate matching and an exact matching. They take place after the signature encoding of the query string.

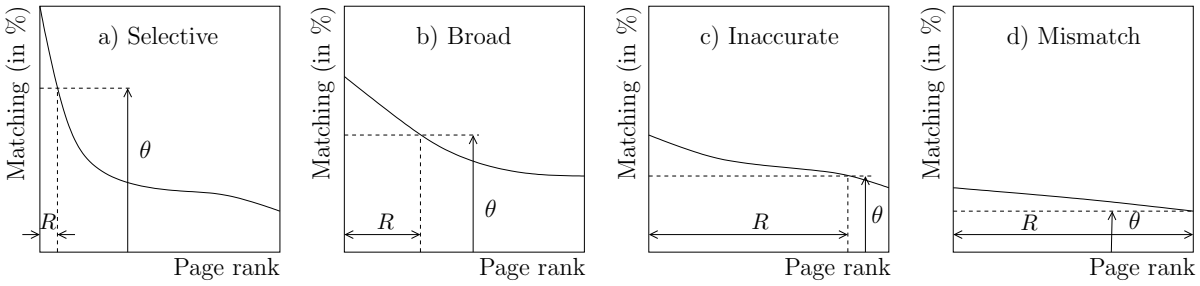


Fig. 3.5. Different distributions of the results after a a) selective, b) broad, c) inaccurate and d) totally mismatching filtering phase. A dynamic threshold value θ, of about two thirds of the highest score, sets the amount of pages R to be reviewed in the second phase of the query.

of the maximum score. In the selective filtering of Fig. 3.5-a, a special query topic is found inside a few pages only and the rest of the database entries match poorly and yield low scores. In the broad and inaccurate filtering of Fig. 3.5-b and c, the query is only partially found, so that the maximum possible score of 100% is not reached. A certain degree of collision is also present in the filtering and raises the scores erroneously. The records from the first phase, matching at the bit attribute level, must consequently be evaluated during the verification phase using more rigid pairing procedures. Furthermore, in the case of Fig. 3.5-d, the scores are so low that the second phase should be avoided. The reasons for this are twofold. First, performing exact matching on all the returned records would take a very long time, and secondly, the correct answer to such a query is "failed".

Concentrating on the filtering phase, the remaining R candidates are sorted in descending order of relevance and passed onto the second phase, the purpose of which is to refine the results and remove the false positives, i.e., the pages with a good score due

to the collisions. This process can be very short if the approximate matching phase was selective enough, as seen in the first case of Fig. 3.5. Though raising the threshold θ reduces the amount of returned candidates R and vice-versa, it has been experienced and advised that depending on the quality of the feature engineering, the second phase might even be entirely skipped. Therefore we will mainly focus our work on the filtering phase of the AAM.

3.2.2 Encoding Query Strings

In this section, we present the method used to overcome the computational limitations due to the basic bitwise signature comparisons as known from the theory as in (2.5). First we take into account the order of appearance of the trigrams in the query string using sequential processing of the query slots. Secondly, partitioning long queries Q into many fragments F allows better matching at the string level and permits documents recovery in which only parts of the original query appear such that

$$Q = \{F_1, F_2, \ldots, F_f\}. \tag{3.3}$$

After marking the fragments, the decomposition of a query string Q of length m into a list of trigrams can be performed using a sequential character selection. Each character item q_j of Q is mapped a trigram τ_j so that

$$q_j \in Q \longmapsto \tau_j = \left\{q_{j-1}, q_j, q_{j+1}\right\} \in \dddot{Q}, \quad \forall j \in [0; m-1] \tag{3.4}$$

with $\dddot{Q}$ the trigram encoding of the query string Q. As seen in Fig. 3.2, the items q_{-1} and q_m are both initialized to the spacing character usually marking the separation between two words. As a result, the number of trigrams that can be formed based on a string of m characters equals its length. Thus we obtain the list of query slots after the encoding of the trigrams into a hash value through a hash function as in (2.4).

$$Q = \left\{q_j |\ j \in [0; m-1]\right\} = \left\{q_0, q_1, \ldots, q_{m-1}\right\} \tag{3.5}$$

$$\Big\downarrow \text{Trigram formation}$$

$$\dddot{Q} = \left\{\tau_j |\ j \in [0; m-1]\right\} = \left\{\tau_0, \tau_1, \ldots, \tau_{m-1}\right\} \tag{3.6}$$

$$\Big\downarrow \text{Hashing } h(x)$$

$$\widetilde{Q} = \left\{h(\tau_j) |\ j \in [0; m-1]\right\} \tag{3.7}$$

Given as an example, Fig. 3.6 represents the coding of a long query which requires different filtering steps preceding the application of the hash function. The assignment of the weights ω_s to each single slot s is constrained by the filtering at the word level. Dedicated filters are supposed to recognize important or unimportant words in a fragment and respectively emphasize them with a higher weight or neglect them with a zero-weight. Experimentally, it can be proven that the quality of the filtering impacts the quality of the results and the overall duration of the search. Moreover in this sense,

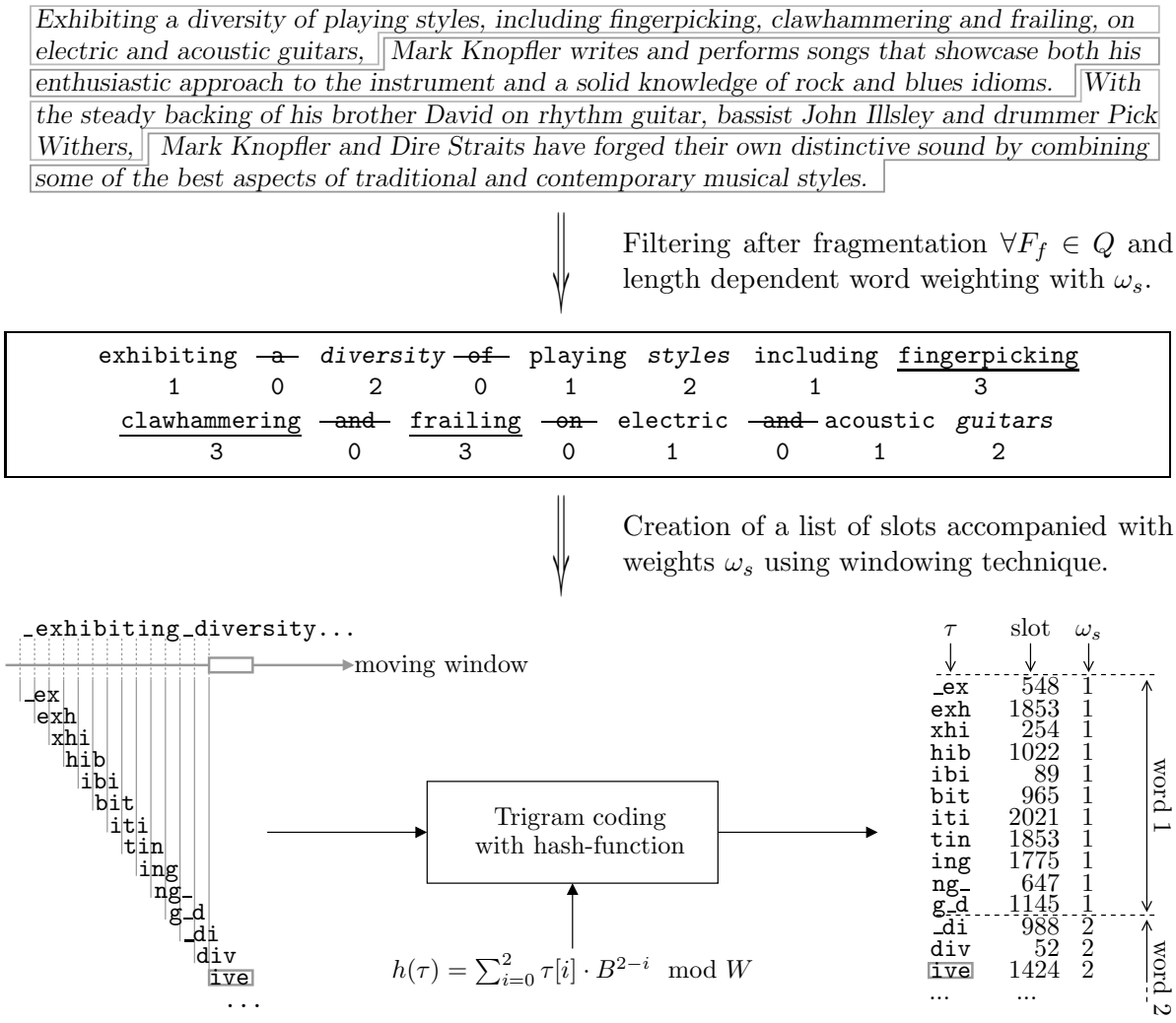


Fig. 3.6. The coding of long query strings requires different filtering steps preceding the application of the hash function $h(\tau)$ on each trigram τ. The obtained list keeps the order of appearance of the trigrams in the query string.

removing too frequent words that carry no information such as “and” or “the” in English does not only improve the coding of the query and the BAM signatures, it also reduces the length of the list of slots and hence the searching time.

3.2.3 Associative Matching Computations

In this section, we provide an overview on the calculation method of the score of a page as it has to be implemented later in our hardware accelerator. By converting a string Q to a set of hash values $\widetilde{Q}$, a vector space is created, thus allowing a sequence to be compared to other sequences in a transformed manner, e.g., the dot product, similarly to the correlation of two signals. The method described by (2.5) in Sec. 2.4 of Chapter II relies on Hamming distance between two signatures. This very basic measure does not take into account any temporal information such as the order of appearance of the trigrams in the string. The AAM, however, provides an improvement of this method

as it considers the query as a sequence of items, i.e., the list of query slots and not an orderless set.

Practically, by addressing the binary signature b_p of a document or a page p from the file with the function $h(x)$, one can know whether the key x is included in the text related to this signature up to a certain extent, depending on the probability of collision in the hashing:

$$b_p[h(x)] = \begin{cases} 0 & \text{if all the features are absent,} \\ 1 & \text{if one of the features is present.} \end{cases} \tag{3.8}$$

The coding in (3.8) does not take into account if a given trigram occurs several times or if it is mapped into an already existing slot. Finding a 0 means that none of the trigrams from the addressed set is present in the page, whereas a 1 reflects the fact that at least one of the trigrams is included. Hence, in terms of communications theory, this shows that a 0 carries much more information than a 1 within the signature of a page. The direct impact on the search method is such that the logical matching statements must be negated. In other words, the part of the algorithm dealing with the signature file has to privilege the treatment of 0s over the 1s which might even be ignored, as they do not bring much information in bit processing.

Based on the information present in the page signatures b_p and in the query Q, instead of awarding a page which maybe owns the searched trigram, non-matching pages are punished more or less intensively, according to the weights given to the different words composing a query fragment F_f. A mismatch penalty $\psi(n)$ is assigned if a page does not hold the binary feature x_n, i.e., the trigram τ_n, which is present in the query. A switch penalty $\sigma(n)$ is given when features appear or disappear along the processing at position n in the query, because when successive features $(\ldots x_{n-1}, x_n, x_{n+1} \ldots)$ are found in the page signature, they probably belong to the same sequence in the original page. Hence, the bigger the similarity of one query fragment F_f to a BAM page p, the smaller the corresponding penalty $\mathcal{P}_p(f)$ defined by

$$\mathcal{P}_p(f) = \sum_n \left(\sigma(n) + \psi(n)\right) \times \omega_n \tag{3.9}$$

where ω_n is the weight associated to the query-slot n. Over all the query fragments, the penalties acquired for one page p must be combined in order to provide an overall rating of the page. The final score $\mathcal{S}_p$ is adjusted after each query fragment f by summing up the logarithm of the penalty $\mathcal{P}_p(f)$ complemented to a constant $\mathcal{K}$ that prevents an overflow so that

$$\mathcal{S}_p = \sum_f \left(\mathcal{K} - \log_2 \mathcal{P}_p(f)\right). \tag{3.10}$$

The purpose of the logarithm is to sum up big values only for very small fragment penalties. Otherwise, in case of a linear function, large penalties of bad matching fragments would impact the result and suppress the small penalties of good matching fragments [Lap92].

Moreover, to ease and accelerate the analysis of the results after processing the first phase, we generate besides each score a Fragment Hit Table (FHT) for each page where good fragments are marked. By comparing the current fragment penalty $\mathcal{P}_p(f)$ with a previously dynamically loaded threshold value $\mathcal{T}_f$, a bit (H_f) is set in the table $\{H_1, H_2, ..., H_i\}$ at the corresponding location in case the penalty is lower than the threshold with

$$H_f = \text{sign}\left(\mathcal{P}_p(f) - \mathcal{T}_f\right). \tag{3.11}$$

As the fragments composing a query might not have the same length or the same importance, it makes sense to be able to provide for each fragment F_f a new threshold $\mathcal{T}_f$ during the encoding of the query. The reasons for the FHT are twofold. First, it allows the removal of bad fragments from the verification phase, as described in the previous section and secondly, it permits a better localization of the score in the uncompressed textual page. Finally, the remaining processing in phase one is the sorting of all the pages according to their score $\mathcal{S}_p$ and the selection of the R best ones only.

Chapter IV

System Level Analysis

NOW that we have given an overview of the thesis in the first chapter, recalled basic implementation strategies related to digital system as well as standard techniques for sorting and searching in the second chapter, and ultimately reviewed state of the art software and alternative hardware solutions proposed by other research groups worldwide in the third chapter, we dedicate this key chapter to the detailed analysis of the AAM search algorithm and discuss our motivation and expectations in this work. Consequently, we provide a parallel revision of the chosen algorithm which shall be best suited for a hardware implementation.

1 Motivation and Expectations

Searching remains one of the most time consuming tasks of many computerized applications. Because of the exponentially growing quantity of digital information, the duration of a search usually scales up more rapidly with the size of the database than it scales down with the always increasing hardware performance of the newest computers. Therefore it makes sense not only to develop and implement alternative algorithms with a reduced complexity, but also to provide a suitable hardware acceleration platform performing record-breaking retrieval times.

1.1 Problem Statement

The substitution of a very accurate and rather complex search algorithm for a simpler but less precise one often leads to a substantial increase in speed. Introduced in Sec. 3 of Chapter III, the Associative Access Method (AAM) is hence an efficient method for information retrieval benefiting from a trade-off between accuracy and speed. Due to

the encoding scheme it bases on, not only is it adaptable to any kind of search, including objects codable with a set of binary descriptors, but it is also extremely well suited for very long queries in huge databases. Based on approximate string matching, it makes use of an index file, the Bit Attribute Matrix (BAM), generated by means of a hash function to compute a kind of edit-distance, as referred to in Sec. 2.3 of Chapter II.

Considering a huge textual database which can be intrinsically divided into small pages, the searching task consists in locating the ones which are the most similar to an issued query string. According to the AAM, once a query string has been encoded, the search is performed in two successive phases that include a filtering of the best matching pages at the trigram level and a more exact evaluation phase at the sentence level. Unfortunately, the filtering phase takes too much time, most probably because too many signatures must be scanned, implying a lot of random accesses to the memory subsystem on which the BAM is stored. Moreover, the associative computing task described in Sec. 3.2.3 of Chapter III costs precious processing time and an enormous amount of data transfers between the main memory, the secondary storage media and the central processor. The reality of the situation is such that soon, due to the exponentially increasing amount of information available world wide stored for later retrieval, it will be impossible to address the problem of long queries in large databases with standard computers without dedicated hardware components.

On the one hand, regarding a single user application, it may not be worthwhile to accelerate the process of searching a couple of items in a small database, as the operations in this case already appear to be instantaneous. However, if the database is stored remotely on a server over which many people are supposed to perform different queries with a relatively high sustained frequency, the information retrieval system must benefit from a very fast search functionality even for small queries, in order to satisfy each user. On the other hand, when it comes to huge databases, the query time is, beside the retrieval quality, the most important factor that makes a search engine a good search engine. Therefore, as we have deeper studied Lapir's AAM algorithm, we intend to build a hardware solution based on this method in such a way that it might even become a part of any standard computer in the future.

1.2 Proposed Research

The very first decision which had to be made at an earlier stage of the development in our work was related to the choice of the most suitable search algorithm that could address the problem of finding information in huge databases while handling very long query strings. Based on a textual paradigm, the AAM presented in Sec. 3 of Chapter III is an application of an approximate matching algorithm that relies on the theory described earlier in Sec. 2.4 of Chapter II. As many single processes compose the retrieval procedure of the AAM, we have to profile the whole algorithm in different situations, i.e., with different query lengths and various database sizes, in order to determine which portion is to be implemented in hardware, even though we know from the description of the

system as seen in Fig. 3.4 on page 44 that the filtering phase is the one processing the most data.

Once we have decided which part of the search algorithm to port to hardware, we have to concentrate on its internal functionalities and analyze them in such a way that we can reach the highest degree of parallelism. This follows the idea developed in Sec. 1.2.2 of Chapter II about handling multiple data concurrently with multiple processing units, which permits the design of a high-performance hardware accelerator. In order to carry out the realization of the search engine efficiently, we have chosen a top-down approach where each part of the system is refined by designing it in more detail. By contrast, in a bottom-up design, individual parts of the system are specified in detail and linked together to form larger components until a complete system is fully described. In our case, the top-down approach emphasizes a complete understanding of the system as a whole, i.e., the search method, while permitting to concentrate also on the particular refinement of dedicated parts, e.g., the memory subsystem or the sorting method. No coding can begin until a sufficient level of detail has been reached in the design, therefore we carefully planned all the modules necessary for the architecture using a Hardware Description Language (HDL). Through a top-down design methodology, we can provide a detailed description of a complex system at a very low level of abstraction without actually knowing the final target technology, e.g., ASIC or FPGA.

When designing a System on Chip (SoC), it is important to carefully choose each element of the system, that is to say not only the ones found on the processing dataflow but also the peripheral ones, i.e., the memory subsystem and the external I/O interfaces. As seen in Sec. 1.4 of Chapter II, only a restricted choice is offered within a wide range of memory devices. The quality of the connection in terms of throughput, data width and transfer frequency extremely influences the overall performance of the final system. Therefore designing the interfaces remains a very delicate part of the work, which has to be handled at an early stage of the whole development.

Sorting plays a major role in the design of our information retrieval system. As we have seen in Sec. 3 of Chapter II, there exist many algorithms which have been thoroughly studied and reviewed [Knu97, Sed88] since the advent of computer science. Various techniques seem to be well suited for our problem and able to settle the sorting and merging tasks within the hardware constraints which appeared during the development of the acceleration platform. Our research focuses on the design and the implementation of an efficient hardware sorting unit with an enormous data throughput, as it is to be expected in a highly parallel system architecture, while caring about the resulting circuit area and its realizability.

Once a design has been elaborated at the system level, it needs to be refined considering suitable speed-up techniques in order to make it more efficient in terms of processing speed and data throughput. At the Register Transfer Level (RTL), pipelining, as introduced in Sec. 1.1.2 in Chapter II, has to be applied onto the system architecture in order to speed-up the whole design. Basically, pipelining reduces the cycle time of a processing unit and hence increases its throughput, that is to say the number of instructions and the amount of data that can be executed in a unit of time. However, despite

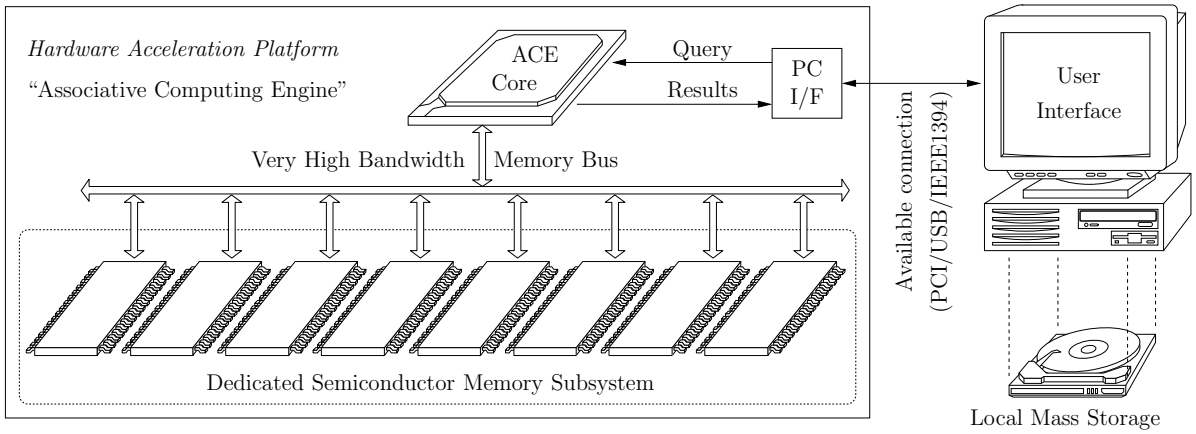


Fig. 4.1. An overview of a hardware accelerator platform supporting an information retrieval system where the database index is deported to a dedicated semiconductor memory subsystem.

this benefit, the method requires a completely new scheduling of the tasks and implies additional controls and synchronization.

Introduced in Chapter II, FPGA devices provide a highly parallel structure that supports spatial computing models instead of a time-multiplexed ones. Not only do FPGAs approximate sequential processors in size while providing orders of magnitude greater performance, also a high flexibility and short design cycles compared to ASICs can be reached through millions of customizable logic gates on a chip [DeH00b]. FPGAs are well suited for the design and test of complex digital circuits, especially computationally intensive signal processing data paths. For these reasons, we choose this kind of hardware platform to implement and benchmark our system, the final realization of which is depicted in Fig. 4.1. It includes an FPGA referred to as "ACE Core" that handles the algorithmic computations, a dedicated memory subsystem providing a very high bandwidth interface where the BAM is stored, and a connection to a standard PC over which query requests are received and results are sent. On the software side, the PC supports the user interface and the driver programs that permit the communication with the hardware platform through a given protocol, e.g., Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB). As we know from Sec. 3 of Chapter III, since the original text database can easily reach many gigabytes in size and does not need to be frequently accessed, it can be stored on the local mass storage devices, e.g., HDD.

2 Profiling and Analysis

This section is based on a software model written in C++ of the complete AAM search algorithm, the functionalities of which are described in Chapter III, Sec. 3.2.1. Originally implemented by G. Lapir, we extracted with his agreement the most relevant

functions of the program and reimplemented them for research purposes without graphical user interface for an open-source Operating System (OS). As Chapter VI later gives an overview of the encoding parts of the algorithm we have needed in order to simulate and benchmark our work, this chapter explains partially the constructs of the matching procedure while focusing on its hardware realization.

2.1 Exploration of the Software Model

Even though we have theoretical knowledge of the computations of an algorithm, it is not enough to guess which parts of it are more likely to be accelerated for the benefit of the whole system. Measurement is a crucial component for performance improvement. As systems become more complex, the need for machine-assisted performance analysis grows. Profiling is a widely used method for the analysis of algorithms in order to determine the cost of an implementation in terms of hardware resources and computational power needed. Hence, in order to identify the bottlenecks in our program, we decided to rely on the Linux OS which is well-served in terms of development tools including a wide selection of profiling packages.

According to the description of the matching computations in Chapter III, more precisely around Fig. 3.4 on page 44, the AAM algorithm is composed of many separate phases which are preformed sequentially. Coarsely, we can group all the operations into three main tasks. The first one includes the query operations, the second one the filtering operations, i.e., the first phase of the search algorithm, and the third one the evaluation operations of the selected pages, i.e., the second phase of the search algorithm. Fig. 4.2 shows the time taken by each task according to different query and database parameters. For later comparisons in our database model, as they scale linearly with each other, a textual database that contains L entries yields a BAM including L lines with a total size of $\frac{L}{4}$kB, considering 2 kb signatures. Four cases are depicted for which the size of the database is increased tenfold while the query becomes slightly longer. It appears clearly that the relative duration of the different tasks within the complete search process varies with both the length of the query and the size of the database. Although every individual duration as well as the total time of the algorithm increase, the filtering phase remains the most time-consuming task of the search algorithm. It varies from about $650\mu s$, for a thousand entries database and a five words query as seen in the top of Fig. 4.2, though representing 40% of the total time, up to about $1.6s$ for a million entries and twenty words query, which corresponds to over 99% of the total time as seen in the last case of Fig. 4.2.

According to Amdahl's law [Amd67], it is possible to find the maximum expected improvement of an overall system when only part of it is accelerated. In this concern, the final speed-up s achievable from the improvement of a computation that affects a

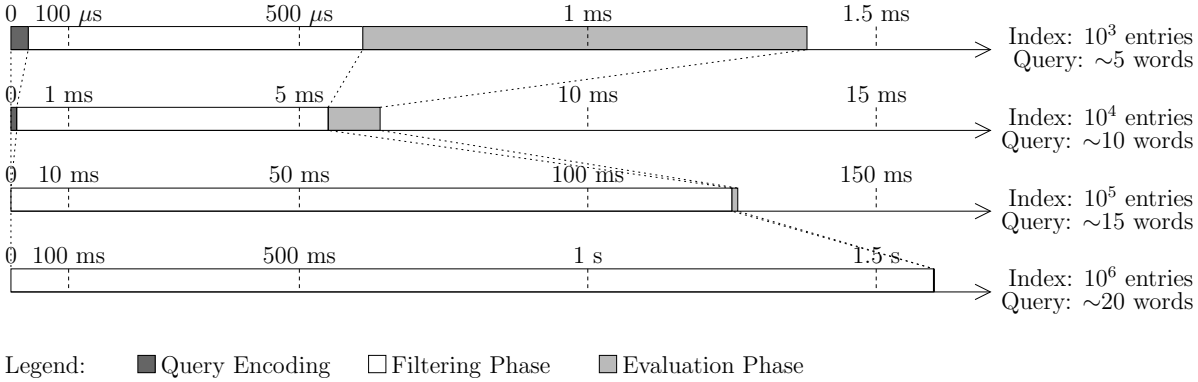


Fig. 4.2. Software profiling of the entire search algorithm helps measure the three phases performed sequentially. As the execution time scales up with the size of the database and the length of the query, it appears clearly that the filtering phase is the most time consuming one.

proportion P_i with an individual speed-up of S_i is expressed by

$$s = \left(P_0 + \sum_{i=1}^{N} \frac{P_i}{S_i} \right)^{-1} \qquad \text{with} \qquad \sum_{i=0}^{N} P_i = 1 \tag{4.1}$$

where P_0 is the fraction of a calculation that is sequential and cannot benefit from parallelization, or in other words the part of the program that remains unimproved. If we are able to speed-up the filtering phase consuming between $P_f = 40\%$ and almost $P_f = 100\%$ of the total time with a hypothetical factor $S_f = 100$, then we can expect a total speed-up s_A for the AAM of

$$s_A = \left(1 - P_f + \frac{P_f}{S_f} \right)^{-1} \qquad \text{such that} \qquad 1.66 \leq s_A < 100 \tag{4.2}$$

where $1 - P_f = P_0$ from (4.1). Even though 1.66 does not appear to be too much at first, the greatest speed-up will be achieved in the situations we need it most, i.e., for combinations of huge BAM sizes and large queries. This confirms our speculations about the AAM from Sec. 3 of Chapter III and guides us through the realization of our hardware accelerator. We know that it does make sense, in order to accelerate the whole search process, to port the filtering into hardware and neither the query encoding nor the evaluation phase. For the reminder of the thesis, we will refer to the AAM filtering phase as Associative Access Filter (AAF).

2.2 Sequential Algorithm Analysis

We consider here the AAF algorithm in its basic form, apart from administrative functionalities such as page selections or query refinements. Given a Bit Attribute Matrix (BAM) of width W and length L with respect to the conventions of the third chapter, we are able to decompose the AAF algorithm into different operations related to the equa-

tions described in Sec. 3 of Chapter III. Fig. 4.3 depicts the AAF in terms of algorithmic steps while these are described in more detail through the listing of Algorithm A. This constitutes a comprehensive basis for the later hardware implementation of the AAF and the choice of dedicated speed-up techniques.

Algorithm A *(Associative Access Filter).* This algorithm explains how to recall the R pages that match the best a given query Q encoded in a set of slots using a trigram based hash function. Let p, f and s index the currently processed page, fragment and slot respectively. Let $N_f(Q)$ and $N_s(f)$ be the number of fragments in the query Q and the number of slots in a given fragment f respectively.

A1. [Initialize.] Set $p \leftarrow 1$ (first page signature of the BAM).

A2. [Page loop.] Reset $f \leftarrow 0$, $s \leftarrow 0$. Process page loop $\forall p \mid 1 \leq p \leq L$.

A3. [Fragment loop.] Reset $s \leftarrow 0$. Process fragment loop $\forall f \in Q$.

A4. [BAM access.] Get query-slot s and read $b_p[s]$ according to (3.8).

A5. [Last slot in fragment?] If $s < N_s(f)$, then increase s and go back to A4.

A6. [Calculate penalty.] Calculate $\mathcal{P}_p(f)$ according to (3.9).

A7. [Calculate hit.] Complete the FHT according to (3.11).

A8. [Last fragment in query?] If $f < N_f(Q)$, then increase f and go back to A3.

A9. [Calculate score.] Update the score $\mathcal{S}_p$ of the page p according to (3.10).

A10. [Last page in BAM?] If $p < L$, then increase p and go back to A2.

A11. [Sort pages.] Sort the L pages coded in the BAM according to $\mathcal{S}_p$. Return the R best ranked pages with respective score and FHT. ∎

Regarding the organization of the nested loops as seen in Fig. 4.3, we know from the functional point of view that certain steps within Algorithm A will either be run through lots of times, e.g., the loop over step A4, or will take a considerable amount of time, e.g., step A11. In order to verify our predictions, the first phase of the retrieval algorithm has been modeled using the C++ programming language and optimized at the instruction level [Lap92]. A listing of the most relevant functions implemented in this model is given in Table 4.1. It shows for each function which part of Algorithm A it refers to.

In order to verify the resource consumption of the different functions composing the AAF model, we rely on the profiling tools of the Linux operating system and evaluate the amount of time and the number of calls of each step. The measured values are reported in Figures 4.4 and 4.5 for the most significant functions according to variations of the length of the query and the size of the BAM. However, it should be noticed that, due to the measurement of very short time periods and the physically impossible absolute accuracy, the position of the points in the following graphs might be subject to a small vertical visual correction.

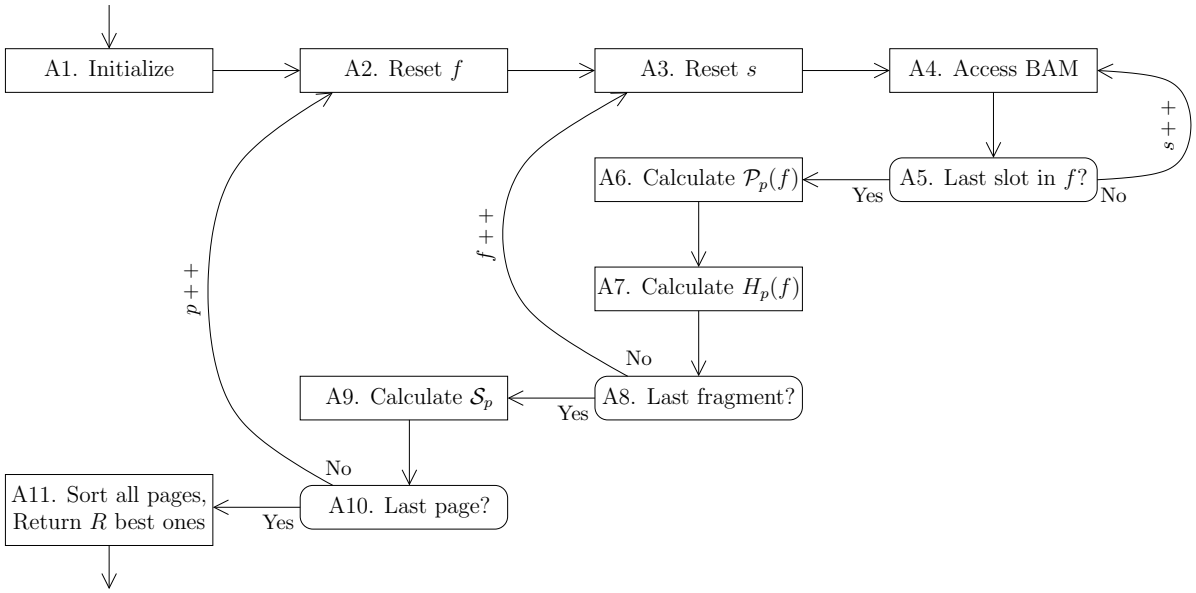


Fig. 4.3. Flow chart of the filtering phase of the AAM according to Algorithm A highlighting three nested loops for the processing of pages, fragments and slots.

Fig. 4.4 demonstrates the global linear dependency of the AAF duration against the query length. Fig. 4.4-a reflects the ratio of computational power needed by the different tasks when the number of searched items varies. As a result, the function `calc_pnlty()` clearly dominates all the others. This effect is amplified through the increasing of the length of the query string, since the implied function corresponds to the most inner loop of the AAF, as seen in Fig. 4.3. Surprisingly, the size of the BAM influences all the functions equally such that we can draw proportionality lines regardless of this parameter. On the other hand, Fig. 4.4-b shows as expected that the time needed by the sorting algorithm implemented in the function `sort_res` does not depend on the length of the query. Moreover, it shows that doubling the size of the BAM also doubles the search time of the functions `calc_pnlty()` and `sort_res()`. This is due to the fact that the width W of the BAM is constant and that only the number of entries in the database sets the length L and consequently the size of the BAM.

Function name	Algorithm steps	Description
`load_bam()`	A1	Open BAM file and get BAM parameters
`read_qry()`	A2, A3	Open query file and get query parameters
`calc_pnlty()`	A4, A5, A6	Read BAM data and calculate penalty
`calc_hit()`	A7, A8	Compare fragment penalty with threshold value
`calc_score()`	A9, A10	Transform fragment penalties into page score
`sort_res()`	A11	Perform internal sort in main memory

Table 4.1. Listing of the most relevant functions composing the software model of the filtering phase of the AAM according to Algorithm A.

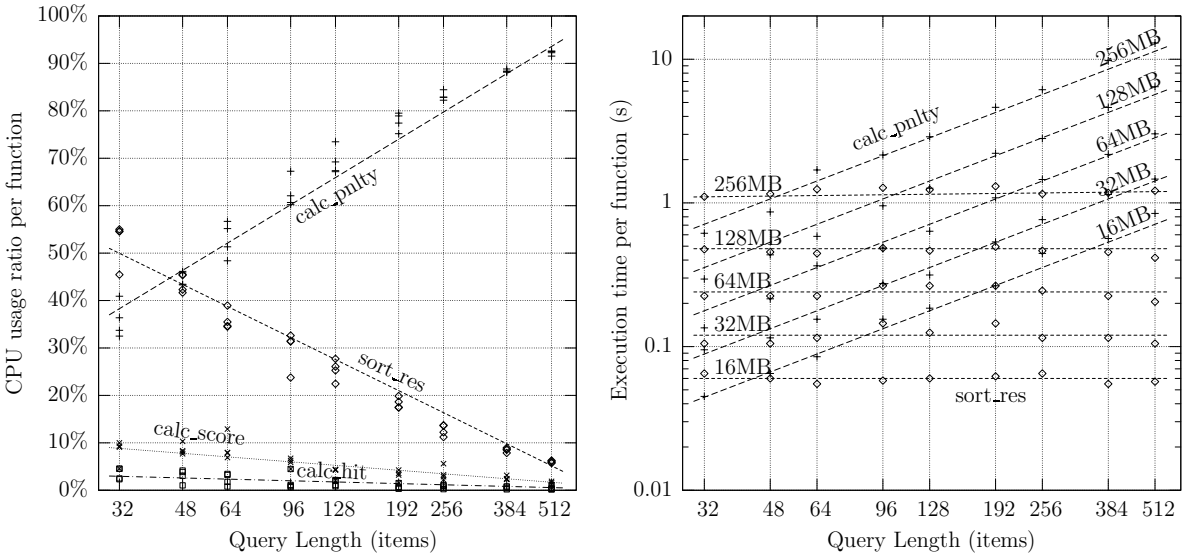


Fig. 4.4. Measurements against the query length of **a)** the CPU usage ratio for the four most time-consuming functions of the algorithm and **b)** the absolute search time in seconds with the size of the BAM as parameter varying from 16MB up to 256MB.

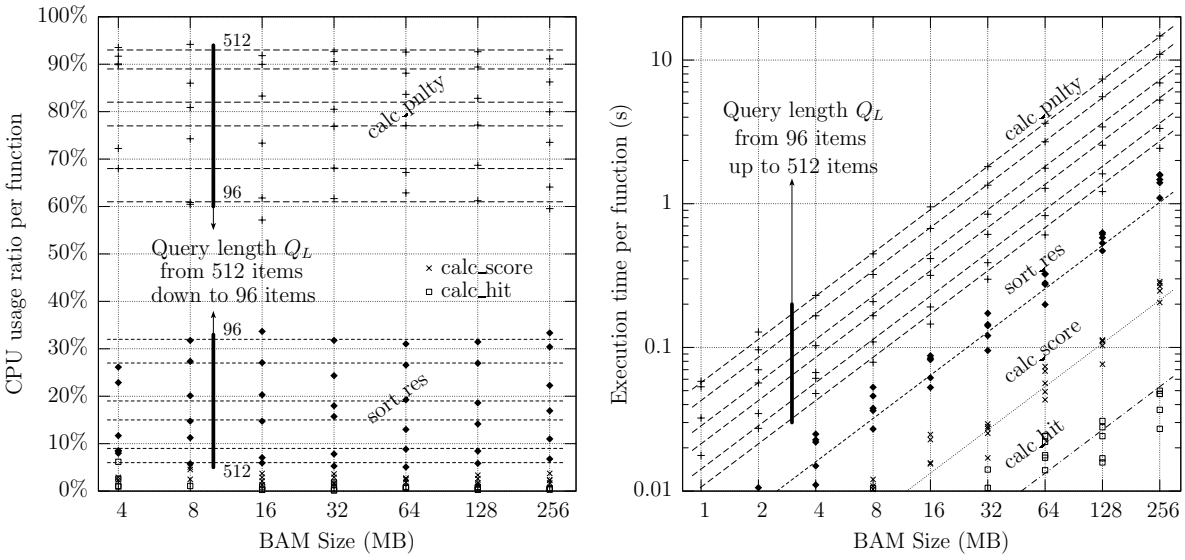


Fig. 4.5. Measurements against the size of the BAM of **a)** the CPU usage ratio for the four most time-consuming functions of the algorithm and **b)** the absolute search time in seconds with the length of the query as parameter varying from 96 items up to 512 items.

Fig. 4.5 demonstrates the linear dependency of the global AAF time against the size of the BAM. As expected, Fig. 4.5-b shows that the `calc_pnlty()` function is the only one to be really affected by the length of the query and confirms Fig. 4.5-a. Even though function `calc_score()` depends on the amount of fragments in the query according to Algorithm A, the profiling graphs show that this effect is almost negligible. It is to be noticed that the data reported in the curves of Figures 4.4 and 4.5 reflects measurements on the most powerful PC available at the beginning of the year

2006. However, no further information on the hardware supporting the AAF algorithm is revealed here on purpose, since the profiling of the algorithm focuses on the relative duration of the successive tasks. A detailed performance analysis considering different computer architectures will be given in Chapter VI.

In conclusion, it is safe to say that only four of the main functions composing the algorithm, as listed in Tab. 4.1, require more than 99% of the total time of the AAF algorithm. We have seen that the size of the BAM plays a secondary role since all the profiled functions are equally influenced by this parameter, i.e., the runtime of the software version of the AAF depends on it linearly. The forthcoming parallelization of the AAF algorithm can hence be based on an arbitrary BAM length L. As one function (`calc_pnlty()`) dominates all the others already for small queries, it has become a key element in the realization of the hardware accelerator to implement it extremely efficiently. The impact of this theory on the rest of the algorithm is thoroughly studied in this thesis.

3 Hardware Accelerator Design

3.1 Parallelization of the Algorithm

While building an acceleration platform, we have to increase the concurrency and reduce overheads due to parallelization, in order to maximize the potential speedup. More than a standard methodology, parallelizing a serial algorithm requires a real understanding of the functionalities of the algorithm and a precise overview on how the data is handled throughout the processing. It is important to consider not only the computational complexity of a process, but also the memory overhead caused by any change in the order of the operations.

3.1.1 From Temporal to Spatial Locality

The basic idea permitting the parallelization of the different tasks composing an algorithm is based on a domain transformation on which time sequential operations are mapped onto an orthogonal physical space. In the translation of a software program to a hardware specific architecture, we hereby trade off area for execution delay. Unavoidably, a redesign of the algorithm becomes necessary, in which a special care of the data and operation dependencies is taken.

As mentioned in Chapter II Sec. 1.2.2, the concept of locality has also been developed for computer architectures that were able to treat different instructions and/or different data in parallel, using multiple computational units. Correct processing becomes possible only if the dependency graph of each data member is respected. Depicted in Fig. 4.6, a small number of hardware resources in the temporal implementation are

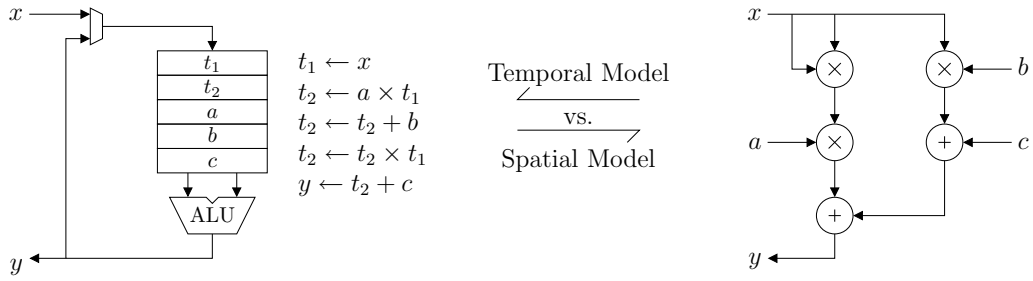


Fig. 4.6. Temporal (left) versus spatial (right) locality with the exemplary calculation of $y = ax^2+bx+c$ from [DeH99b] reflects the trade-off between time and area.

reused in time whereas each operator unit exists at different points in the spatial implementation [DeH99b, DeH00b]. As exploiting parallelism permits to achieve high throughput with low computational latencies, our goal is to find a suitable architecture for the implementation of the AAF.

3.1.2 Parallelizing the Tasks

An algorithm is basically composed of discrete tasks that involve both data and various operators, as well as a plot that guide their execution. Parallelizing the tasks in order to speed up the algorithm requires a certain independency among the components of the processed dataset, not only in space but also in time. On the one hand, operations implying different members of a dataset can be inherently performed at the same time because there is no risk of inconsistency within the data. If a hardware operator f is spatially available n times, then the dataset $\{y_0 \dots y_{n-1}\} = f(\{x_0 \dots x_{n-1}\}) = \{f(x_0) \dots f(x_{n-1})\}$ can be calculated concurrently. The degree of parallelism is conditioned by n. On the other hand, the elements of a dataset computed through a series of operations can benefit from the propagation delay τ of each single operator if they can be fed in sequentially. If a variable $y(t)$ must be calculated using k operators such that $y(t) = f(x(t))$ with $f = f_0 \circ \dots \circ f_{k-1}$, then a temporal processing of $\{y(t_0) \dots y(t_{n-1})\}$ can be performed in $O(n\tau + k)$ time complexity considering that the k operators are available in hardware. With a trade-off between latency ($\propto k$) and throughput ($\propto n\tau$ with $\tau \propto k^{-1}$), the degree of parallelism is conditioned by k. This idea corresponds to the pipelining principle explained in Sec. 1.1.2 of Chapter II applied at the algorithmic level and implies that no feedback is present in the processing of a single task.

Regarding the AAF described in Algorithm A, we first extract the possible algorithmic transformations by studying the data dependencies along the tasks. The whole process is linear and implies three nested loops that fetch the BAM data, evaluate the fragment penalty plus hit, and calculate the score of the current page. In the presence of deterministic conditions, loops are usually unrolled in hardware to permit a parallel processing of the internal data. On the contrary, the most inner task, i.e., the BAM access A4, can only be performed in a sequential manner due firstly to physical realization constraints within memory arrays, and secondly to the unknown length of the query string. It compels a long processing time for the following tasks and removes the possibility for the loop to be unrolled.

The sequential processing of the data as seen in Fig. 4.3 comes from the fact that Algorithm A originally runs on a unidimensional computing machine, i.e., a single processor computer. For instance in the function `calc_score()` described by (3.10) for step A9 that transforms the fragment penalties into a page score and the function `calc_hit()` described by (3.11) for step A7 that compares a fragment penalty with a threshold value, the input data is the same penalty $\mathcal{P}_p(f)$ of the fragment f for a given page p. In other words, A7 and A9 can be performed in parallel, considering that enough hardware resources are available. Furthermore, a certain degree of freedom is assigned to the AAF algorithm around these two functions, as they appear in two different loops although they own the same input constraints. The reason is that the processing occurs progressively in A7 where a single bit in the FHT is updated after the calculation of a fragment penalty, whereas the score computation is performed when all the fragment penalties have been calculated. It must be noticed that due to the associativity of the addition in (3.10) and the vectorization in (3.11), the placement of A7 and A9 in either loop is allowed.

In summary, with the availability of enough hardware resources, it is possible to perform all the processing tasks in parallel, if we respect a few conditions. When tasks imply the same input data, then there is no problem to make them truly parallel. Otherwise we have to apply the pipelining principle at the task level with a special care on the synchronization. Furthermore, the sorting task A11 can be split into two parallel subtasks A11a and A11b, where the first one would sort a few values while the second would merge the previously ordered values into a list of R final results. The advantage is that the sorting function does not need to wait until the end of the processing to start its duty, hence removing $O(L)$ delay on the AAF execution time.

3.1.3 Parallelizing the Data

According to Sec. 3.1 of Chapter III, the BAM is defined as an $W \times L$ matrix with W its width and L its length. In this matrix, the rows correspond to textual page signatures which are functionally independent and theoretically uncorrelated. The consequence is that an arbitrary number of signatures N can be processed in parallel without changing anything in Algorithm A as described in Fig. 4.3. As the required start information for the evaluation of a page in step A4 corresponds to one bit according to (3.8), the input of a parallelized version of the algorithm becomes a bit vector of length N.

A calculation technique related to superscalar CPUs and SIMD machines, as seen in Sec. 1.2.2 of Chapter II, which handles one single bit of many data variables at the same time is referred to as vertical computing. In [Ber98, Lap92, Lay05a], a vertical counting method is used to solve the problem of accumulating single bits in parallel over the fixed size data-bus of a standard processor. For the hardware design of our accelerator, we can reuse this idea without being constrained by any processor-related typical internal data-width. Therefore we take N as the input parameter that specifies the number of bits that can be processed concurrently, or in other words the number of pages.

Basically, the time needed to treat L pages from the BAM becomes N times shorter if we use N parallel PEs (Processing elements). However, this is true for the computing

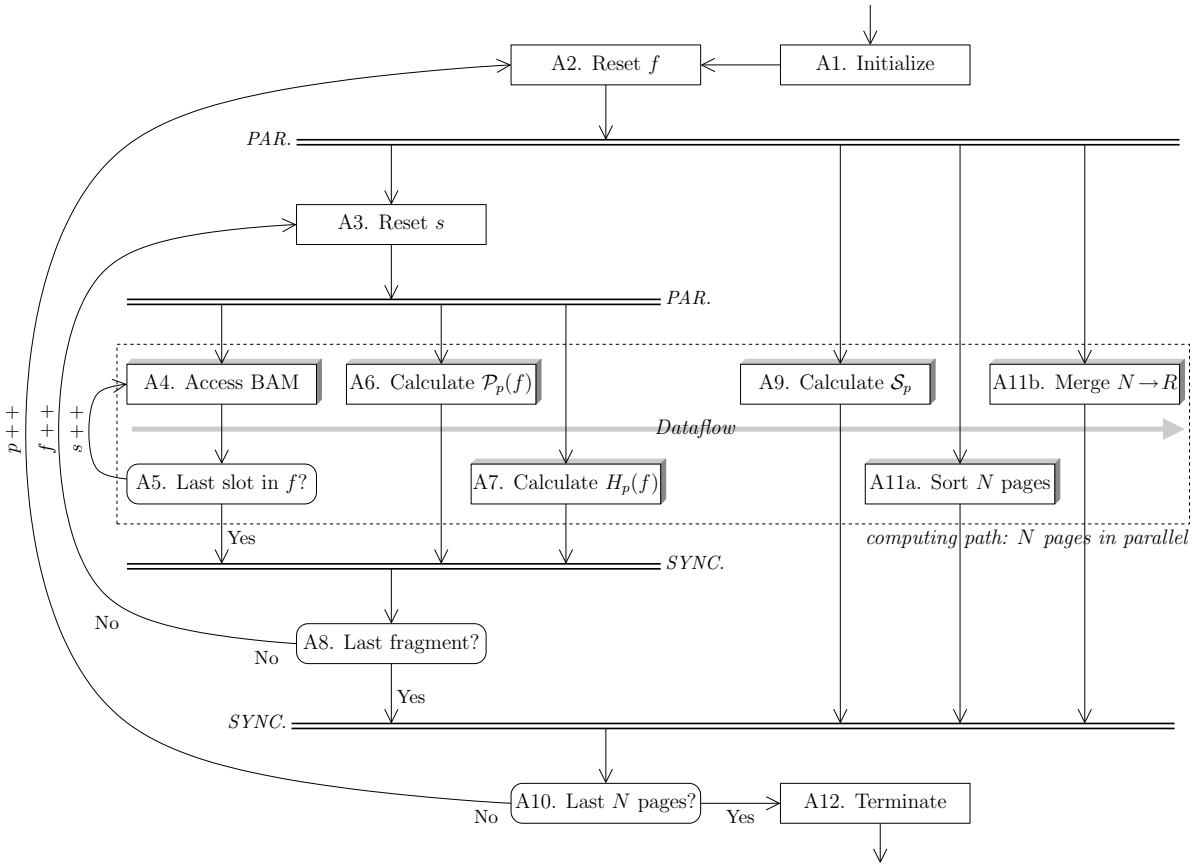


Fig. 4.7. Parallelization of the flow chart of the AAF. The double lines are used for the synchronization of groups of tasks. Data processing flows horizontally over the computing path through A4, A5, A7, A8, A11a and A11b. Task A12 makes sure the merging is entirely finished.

operations, i.e., steps A1 to A10, but not for A11, as sorting is not an inherently parallelizable task. Referring to Sec. 3 of Chapter II, we have to use a method for sorting which allows a concurrent access to the data.

3.1.4 Resulting Parallel Algorithm

Described in Sec. 2.2, the sorting task, i.e., step A11 of algorithm A consists in ordering the addresses of all L pages according to their respective score after the necessary computations and returning the R best ones. As mentioned in Sec. 3.1.2, we transposed A11 into a sorting task that processes N pages in parallel and a merging task that progressively keeps the R best records among all the paths of N pages. According to all the previous considerations, Fig. 4.7 represents the resulting parallelized flow chart of the AAF algorithm. This solution constitutes the starting point in our hardware design and includes a computing path through which data is processed concurrently over N pages. The parallel starting (*PAR.*) and synchronization (*SYNC.*) bars ensure the coherency of the data towards this path. They start the tasks with available input data, e.g., they let the sorting wait at the beginning, and resume the merging as often as necessary even if no data is read from the BAM any more. As a supplementary but necessary feature, the

parallel version of the AAF includes a final synchronizing step A12 which is essential in pipelined multi-tasking systems in order to ensure a complete processing of all the data.

3.2 Modular System Architecture

In the realization of a hardware coprocessor for the acceleration of the AAF, we have to decide on the amount of resources we can spend for the parallelization of the tasks related to the data processing, as seen in Fig. 4.7. Our goal is to obtain a modular architecture able to process N pages of the BAM in parallel with a very high throughput, from the memory access to the sorted list of results.

3.2.1 Parallel Processing Elements

Introduced in [Lay03] and refined in [Lay05c], our model is an SIMD kind of associative processor with many levels of specialized PEs arranged in an irregular systolic array. These are assigned specialized tasks and different data widths in order to be mapped optimally on the parallel AAF algorithm. The entire architecture is driven by a global controller that schedules the different units by interpreting the hashed query string into simple instructions. We hereby give up some flexibility for much more performance in creating a specialized associative processor. Since we can group the main tasks into five functional levels as depicted in Fig. 4.8, we designed a system composed of five processing units:

- The BAM interface (BIF) is the first block of the processing, the purpose of which is to perform a very fast access to the BAM information required by the next module. It has to transform the hash values of the query string into physical addresses to a memory we don't know the type of yet, since it depends on the targeted hardware platform.

- The Penalty Calculating Unit (PCU) is responsible for the calculation of the fragment penalty according to (3.9). It must process the data coming at high speed from the BAM with the query information, i.e., the weight of each query word, given by the global controller.

- The Score Calculating Unit (SCU) owns two processing tasks. Using the fragment penalty passed on by the PCU, it first sets a bit in the FHT according to (3.11) and updates the score of the page in an incremental manner according to (3.10). The threshold value needed to mark a fragment hit must be given by the global controller, according to the importance of this fragment in the query.

- The Result Sorting Unit (RSU) has the duty to sort the pages in their order of relevance to the query according to the page scores. It includes operations that cannot be performed horizontally, e.g., comparisons and exchanges, therefore an internal network is necessary in order to distribute the values vertically among the PEs.

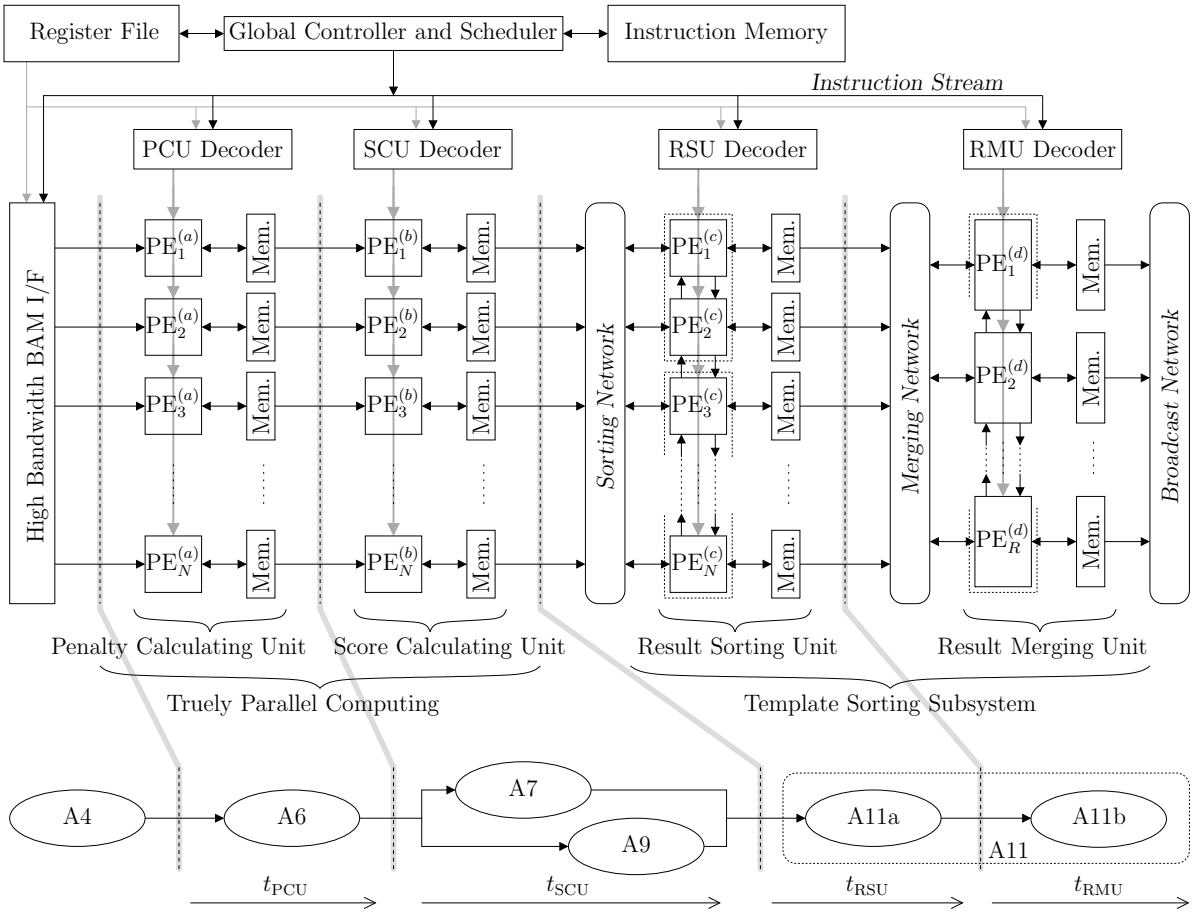


Fig. 4.8. Symbolic representation of our modular SIMD system architecture composed of four levels of computation and a memory interface. This associative processor is controlled by a global scheduler that transforms a query string into a list of instructions for the different PEs.

- The Result Merging Unit (RMU), as the last processing unit, is similar to the RSU. Its purpose is to merge the newest sorted results into the final result list of length R. It is made of a different number of PEs and a different kind of network that helps the merging of the data. The broadcast network permits the access to the result list in a serial manner by an external interface.

Clearly, Fig. 4.8 depicts only a basic structure for the ACE where the PEs conform more to the operational parallelism than the hardware realization. Represented by vertical arrows, it is foreseeable that the RSU and the RMU will require different kinds of connections between the PEs since they basically implement sorting and merging algorithms, as seen in Chapter II, that rely on comparison and exchange operations.

3.2.2 Theoretical Timing Constraints

The logical dataflow in our design goes globally from left to right, i.e., from the BAM to the result list. As our main goal remains the highest performance possible, we have to make sure that every unit is faster than its predecessor, so that no data congestion can occur in the middle of the design. A direct consequence to this philosophy is that the

processing bottleneck of the design moves out of the system to the data input port, i.e., the BAM access interface. In this sense, we have to respect the theoretical conditions

$$t_u \leq t_{\mathrm{PCU}} \qquad \forall\ t_u \in \{t_{\mathrm{SCU}}, t_{\mathrm{RSU}}, t_{\mathrm{RMU}}\} \tag{4.3}$$

where t_u corresponds to the complete data processing time t in the unit u, in order to benefit from the maximum throughput available. Practically however, during the hardware implementation of the sorting subsystem in the next chapter, a reevaluation of t_{RSU} and t_{RMU} becomes necessary in order to ensure a high scheduling flexibility regarding the task parallelism.

Interface:	BIF→PCU	PCU→SCU	SCU→RSU	RSU→RMU
Frequency:	$f_{\text{memory bus}}$	f_{fragment}	f_{query}	f_{query}
Data width:	$N \times 1$ BAM bit	$N \times \lvert\mathcal{P}_p(f)\rvert$	$N \times \{\lvert\mathcal{S}_p\rvert + \lvert\mathrm{FHT}\rvert\}$	$N \times \{\lvert\mathcal{S}_p\rvert + \lvert\mathrm{FHT}\rvert + \lvert\mathrm{addr}(p)\rvert + \lvert\mathrm{rank}(p)\rvert\}$

Table 4.2. The description of the interfaces in the modular design reports functionally the transfer frequency and the type of data being exchanged between the different units.

At the algorithm level, Table 4.2 reports the frequency and the data types at the interfaces in each block of the design as seen in Fig. 4.8. These vary oppositely, i.e., the frequency decreases over the units of the datapath as the data width increases, and influences the architectural throughput in a hardly predictable way as some parameters remain dynamic or unknown, e.g., the length of the query string. Hence the most difficult part in the hardware realization of the ACE will be to make the different units fast enough to keep the bottleneck at the BAM interface in order to ensure the maximum physical throughput and performance possible.

Chapter V

Architectural Hardware Design

FROM the BAM to the result list, a huge amount of data is processed over many computing units under the supervision of a global controller according to the AAF described by the parallel version of algorithm A. In this chapter, we present the details of the datapath in three sections. First we discuss the choice of a memory and its controlling for the most efficient data access possible. Secondly we review the parallel calculating units that we intend to keep fast and small. Finally, we depict the evolution of our sorting blocks and their optimization using different sorting methods.

1 System Management and Peripherals

Every complex processing system owes its final performance to both a powerful data processing and an efficient scheduling of the operations. Therefore in this section, we initially consider the data path as an external entity and concentrate on the control path of the ACE. The hardware implementation of the processing units will be presented in more detail in the next section according to the computational arithmetic described in the previous chapter.

1.1 Operation Scheduling

SIMD processors usually rely on the instructions of general purpose processors for the supervision of the execution of a program, e.g., loop control or subroutines, and provide special instructions for handling data in parallel. In the ACE, we base the operation flow on a listing of dedicated instructions which can be executed by one or more processing

units. In each unit, as seen in Fig. 4.8 in the previous chapter, an instruction decoder recognizes on the instruction bus the commands it has to process and distributes control signals internally. On the top of the design, we use a state machine called Global Controller and Scheduler (GCS) to perform the execution of the listing present in the instruction memory. The register file used by the controller to set temporary values and other local parameters such as the fragment threshold or the current page offset can be read by the instruction decoders of the processing units. Even though the BIF (BAM Interface) remains the only block which is not a computing unit but a peripheral interface, it still needs to access some registers, e.g., to set the correct data path in the BAM memory.

Instruction	Description	Effect on
`ACC WGT ADR`	accumulate weighted penalties for BAM bit at b_p[`ADR`]	BIF PCU
`CMP FRG THR`	set $H_{\texttt{FRG}}$ in FHT if $\mathcal{P}_p(\texttt{FRG}) <$ threshold `THR`	SCU
`VAL USR`	read administrative row `USR` and reset scores of unauthorized pages or user when valid bit not set	BIF, RSU
`BNZ REG ADR`	test value of register and branch to `ADR` iff `REG` $\neq 0$	GCS
`INI BEG END`	initialize the path counters for a limited search	GCS
`SRT`	start sorting process and result list insertion	RSU, RMU
`NOP`	no operation (wait one instruction cycle)	GCS

Table 5.1. The GCS is driven by a set of generic 16 bit instructions which affects different units of our associative coprocessor. It must be noticed that a typical thus programmed query includes about 95% of the `ACC WGT ADR` instruction.

For an efficient scheduling of the operations during a query, we developed a reduced instruction set. The resulting list is given in Table 5.1 with a short description and the affected units in the ACE. Each instruction is basically coded with 16 bits, 4 of which consist in the opcode and 12 are used for the operand. An exception is made with the `ACC` command that codes the weight in the opcode and leaves 12 bits for the BAM column address. The resulting instruction is then in the format $[\texttt{O}^1\texttt{W}^3\texttt{S}^{12}]$ with `O` the opcode set to 0, `W` the weight and `S` the query slot, and permits a BAM width of $W = 4096$. Note that the `NOP` instruction coded $[\texttt{O}^4\texttt{X}^{12}]$ directly derives from the `ACC` with zero weight and an ignored address `X`. Nonetheless, the instruction set depicted in Table 5.1 can be easily extended. It is quite evident that the software driver, which allows the connection between the ACE and the host computer, plays a role in the correct formatting of the list of slots into an understandable program transferred to the instruction memory.

1.2 Designing the Memory Interface

As introduced in Chapter II, electronic memory devices are available in many different forms and types. For a fast and random access to the data, as we wish to obtain for the storage of the BAM, there are actually only two alternatives. The first one is the fast but

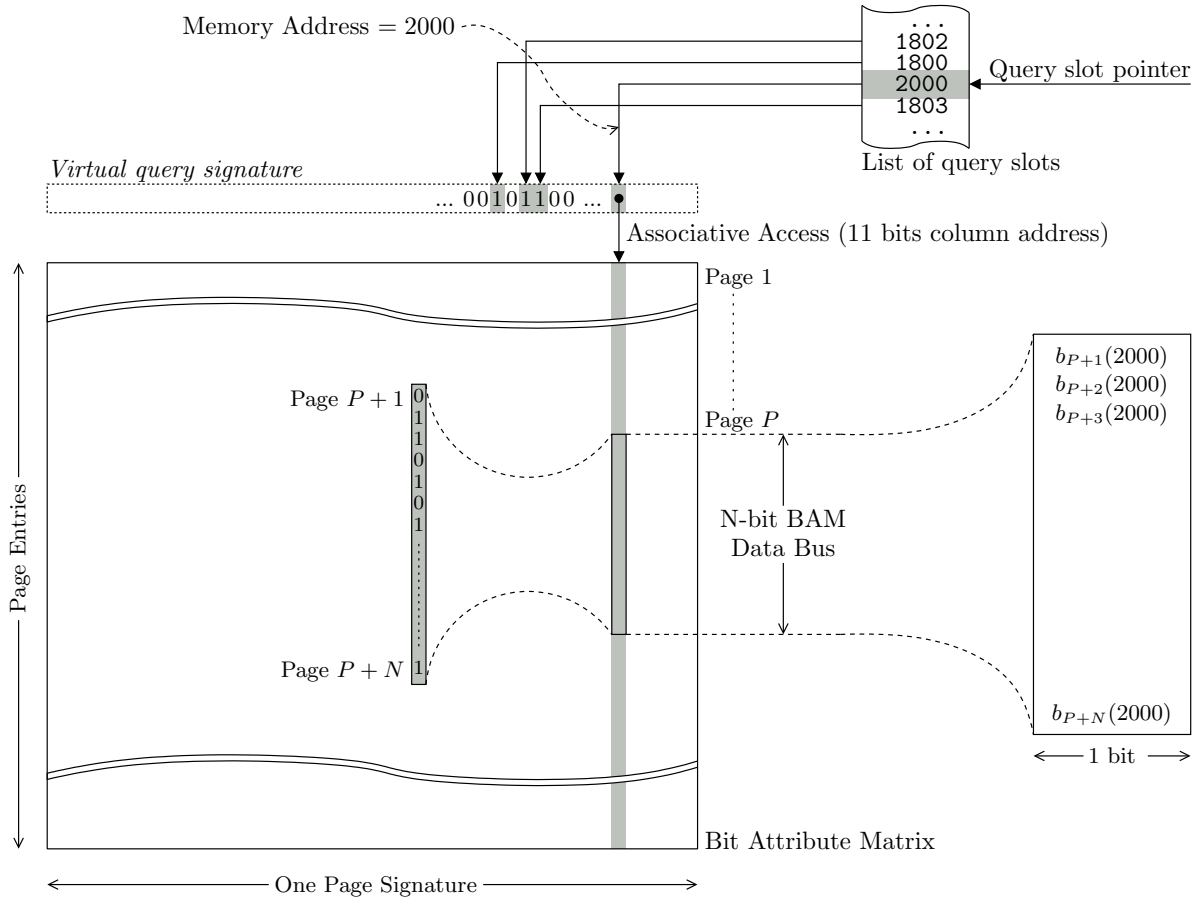


Fig. 5.1. Vertical accesses to the BAM permit a full usage of the N bit information read, as well as the parallel processing of N pages. Note that the virtual query signature is only represented for understanding purposes and never really exists in this vector form.

quite expensive Static Random Access Memory (SRAM) and the second one is the very dense but rather complex Synchronous Dynamic Random Access Memory (SDRAM). Both are volatile memories and can be read or written over an arbitrarily wide data bus. However, trying to access them with the highest throughput possible continuously has created a new challenge in this work. This constitutes in fact the bottleneck of the whole system, as referred to in Sec. 3.2 of Chapter IV.

1.2.1 Accessing the Bit Attribute Matrix

As starting point of the data flow, the BAM interface must provide a flexible and fast access to the data from the BAM. The key of our system is that the BAM is accessed vertically bitwise, i.e., by attribute instead of horizontally by signature. This feature enables the reading of a vector of N bits, as seen in Fig. 5.1, and hence the parallel processing of N pages through the computational units.

The role of the BAM interface is to transform the list of query slots into a physical address for the memory media in terms of abscissae, i.e., the actual slot position in the virtually created query signature, and in terms of ordinates, i.e., the address of page P+1. In this sense, the next N bits data word read from the BAM can be either at the

next pointed column in the list of query slots for pages P+1 to P+N, i.e., an horizontal move in Fig. 5.1, or at the following pages P+N+1 to P+2N for the same attribute, i.e., a vertical move within the same column. This choice is made by the global controller and depends on the characteristics of the used memory device. We will see later that a mix of both directions resulting in a zig-zag pattern is necessary to provide an optimal use of the memory interface.

1.2.2 Choosing the most suitable devices

As explained in Sec. 1.1 and 1.2 of Chapter IV, the target devices for the storing of the BAM are RAMs. If the access speed were the only matter, we would chose the SRAM type for its speed and its interfacing simplicity. However, if we had only to consider the size of the BAM that can reach many giga-bytes, then we would utilize DRAM devices for their density and their price. To answer the question, it is necessary to know how these devices are working internally so that they can be used in the most efficient way possible.

The information storage in memory is based on a positive feedback for SRAM and on a capacitive charge for DRAM. Fig. 5.2 represents the basic building of a CMOS SRAM cell on the left-hand side and of a single transistor DRAM cell on the right-hand side, both for the storage of one information bit. The SRAM cell requires six transistors per

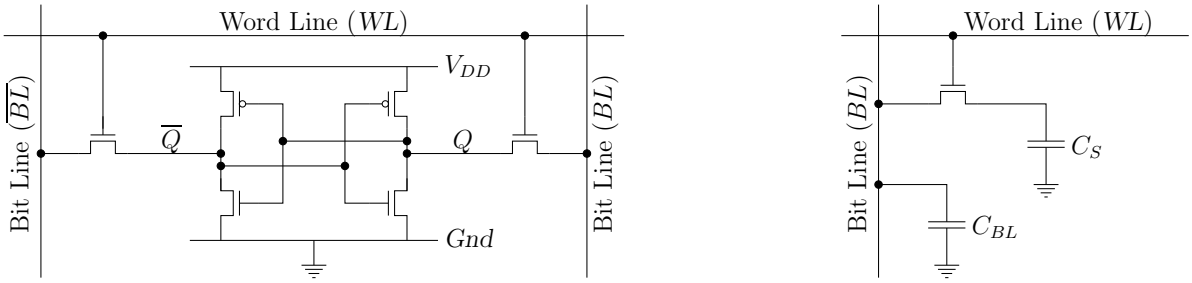


Fig. 5.2. Representation of a six transistors CMOS SRAM cell (left) and of a one transistor DRAM cell (right) for the storage of one data bit. The selection of the cell occurs through the activation of the word line and the storage information is accessed through the bit line [Rab03].

bit, four of which constitute two cross-coupled CMOS inverter structures for the storage of a bit in its two polarities Q and $\overline{Q}$, and two pass transistors shared between the read and write operations that access the cell through the bit line (BL) by enabling the word line (WL). Providing both states of the bit on the two bit lines improves noise margins during the operations to the cell, as analyzed in [Rab03].

The more area-efficient single transistor DRAM cell represented in Fig. 5.2 is composed of one transistor and one capacitance as it is found in today's very large semiconductor memories [Rab03]. During a write cycle, the data value is placed on the bit line (BL) and the word line (WL) is raised enabling the charge or discharge of the capacitance C_S. During a read cycle, the bit line is precharged, typically to $V_{DD}/2$, and the word line is asserted. As the charge are redistributed between C_S and C_{BL}, the direction of the change determines the value of the data stored. As for the SRAM cell, the

sense-amplifiers required for each bit line speed up the readout of the cell. However, the readout of the 1T DRAM cell is destructive, so that the original value must be restored after each read operation. In addition to that, the dynamic memory has the necessity for each cell to be refreshed periodically, due to the unavoidable leakage effects affecting the charge of C_S. Even though these functionalities are supported in commercial devices to ease the building of the memory controllers, e.g., the refresh counter that automatically increments the internal addresses, the timings must be carefully studied and respected in order to obtain high performance data exchanges with the memory, leading to more efficient applications. Apparently, both DRAM and SRAM devices are suitable for the storage of the BAM on our ACE system. The main differences are in the interfacing complexity, the speed in terms of clock frequency and the internal density. Even though SRAMs are more simple and faster to access, DRAMs present a higher density for a lower price. Regarding our application that must deal with gigabytes of text signatures, it might be almost impossible to find or build an affordable hardware platform that provide such a huge amount of SRAM. Therefore we finally chose to target DRAM devices, despite all their constraints.

1.2.3 High Throughput Memory Controller

Designing the control and timing circuitry is a demanding task that requires extensive simulation and design optimizations. According to Rabaey *et.al.* [Rab03], it is an integral but often overlooked part of the memory design process and has a major impact on both memory reliability and performance. As the bottleneck of the system has been identified in Sec. 3.2 of Chapter IV, the challenge here is to map the data of the BAM into the SDRAM devices in order to obtain the highest read throughput possible. In a clocked system, this means a new data word per clock cycle for Single Data Rate (SDR) devices and twice this frequency for Double Data Rate (DDR) devices.

In comparison to SRAM devices that can provide any random data word at every clock cycle, SDRAM devices suffer from a high access latency and from intermediate clock cycle penalties due to a highly constrained addressing procedure which is due to the internal structure of the devices themselves. As seen in Fig. 5.3, an SDRAM is typically divided into four banks of bidimensional matrices of N bit cells. Each bank must be addressed sequentially by selecting first a row which is loaded into the sense amplifiers, and then by reading from or writing into this row at the wished column. Each cell address must be passed to the internal decoding logic in two cycles, i.e., first the activation step where the row address buffer gets the bank and the row address, and then the read or write operation where the column address buffer gets the offset of the cell in the row. At the end of a transaction, the data loaded into the I/O registers connected to the data bus must be written back to the cell in a supplementary precharge cycle. For these reasons, SDRAM devices are usually equipped with a column address counter that can be programmed to perform burst accesses. This avoids the continuous loading of new column addresses for the reading or writing of consecutively stored data words. Notice that paradoxically for more clarity in the BAM mapping representation

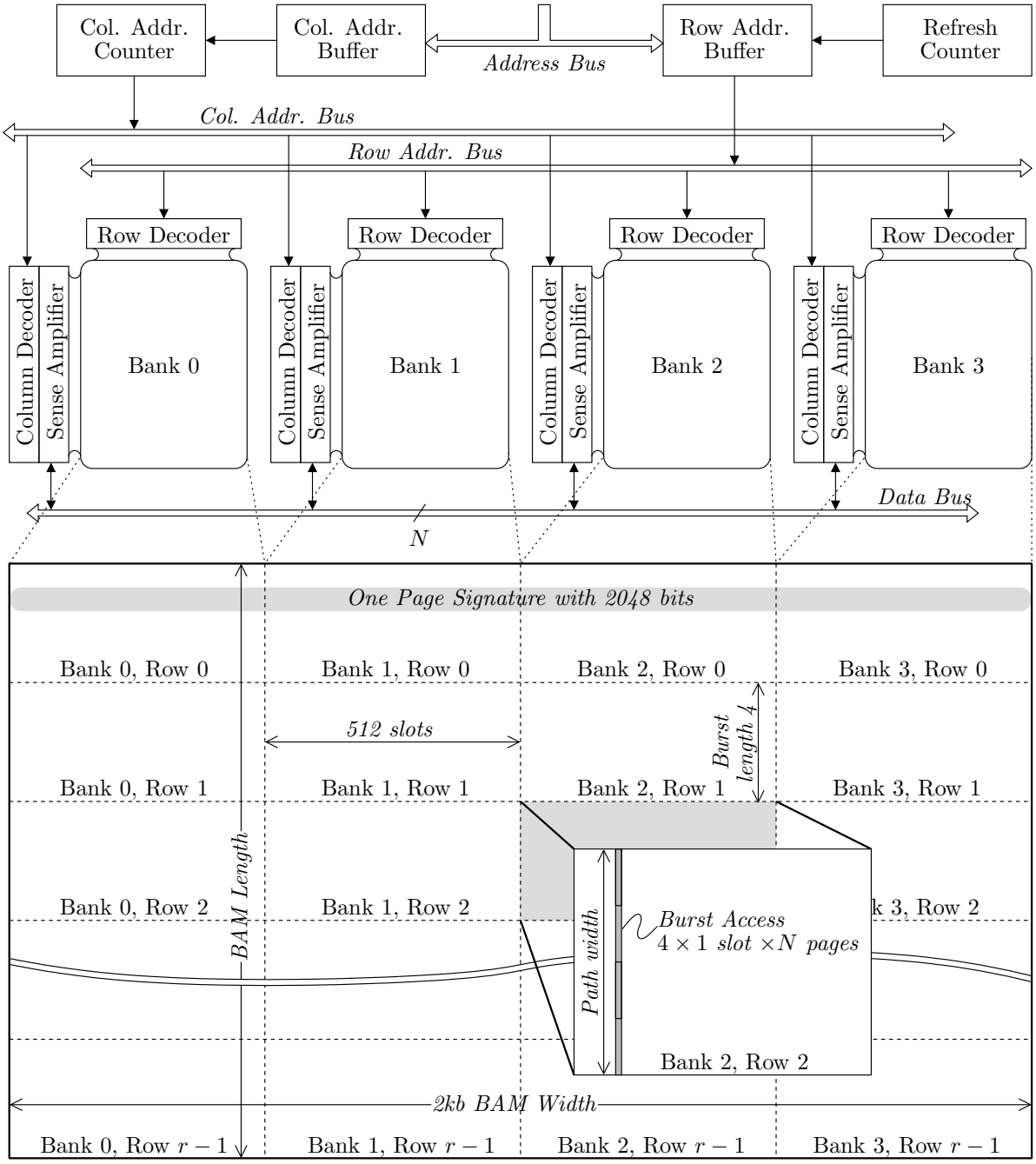


Fig. 5.3. Mapping of the BAM onto standard four-banked SDRAM memory devices. The memory subsystem is accessed through an N bits wide databus on the basis of a burst-4 addressing mode. The device is considered to have 2048 columns per row. The width N can be distributed over an arbitrarily number of chips.

in Fig. 5.3, the rows and the columns of the SDRAM are respectively vertical and horizontal.

Apart from the necessary frequent refresh of the capacitive charges in each cell, this rather complex double addressing mode forces transactions to be delayed on the bus between bank activation, read or write column accesses and row precharges. As

standard SDRAM controllers are designed to provide more addressing flexibility than data throughput, we have to analyze the feasibility and the performance of a custom controller. Our motivation is to map the BAM the most efficiently onto the available memory cells in such a way that no latency delay penalizes the processing of the data by the ACE. To this end, considering the abilities of the SDRAM devices and a BAM width of 2 kbits, we found out that the most suitable and beneficial arrangement of the data into the SDRAM banks is as shown in Fig. 5.3. The main features of the mapping, which permit to keep the maximum throughput as average sustained throughput, can be summarized in using a burst access of length four and an interleaved bank addressing. Hence, the ACE can perform the processing of $4 \times N$ pages, i.e., the width of the working path, within 4 clock cycles per addressed slot. The reason is that four clock cycles are necessary to read data from an arbitrary cell, while the previously loaded row can be precharged. It is not more than four in order to reduce the amount of internal accumulators within the ACE, as penalties and scores must be remembered during the processing of the $4 \times N$ pages in the data path. We refer to Appendix B for a detailed description of the SDRAM controller and its state machine, as well as the timing diagrams.

2 Building the Computational Data Path

This section presents in detail the Penalty Calculating Unit (PCU) and the Score Calculating Unit (SCU) according to [Lay05b, Lay05d]. With the arithmetical operations described in the previous chapters, these units are able to process data coming from the memory subsystem in such a way that no congestion is produced on the data path. Because data is processed with a high degree of parallelism, the challenge remains to keep the architecture as small as possible.

2.1 Penalty Calculating Unit

In the whole design of the ACE, we intend to use a single clock domain, i.e., either a single clock or clocks that have constant phase relationships. In order to avoid metastability of clock domain crossing through asynchronous signals, we base the whole computational architecture of the different modules on the BAM clock frequency. The consequences on the sorting units will be studied in the following section.

2.1.1 Description

Being the first processing unit in the data path, the PCU calculates the fragment penalty of N pages in parallel according to (3.9). Its input is a bit vector of length N that

arrives at the memory frequency in burst mode of length four and its output a $4N$-vector of unsigned integers of arbitrary length that represent the penalties of $4N$ pages. However, the burst accesses, caused by the way data is read from the BAM by our custom SDRAM controller, have an impact on the whole computing architecture. The $4N$ pages are uncorrelated and must be treated independently. In order to handle the data efficiently, we can either demultiplex the burst data and send it to four equivalent ALUs or process it serially given that the ALU is fast enough.

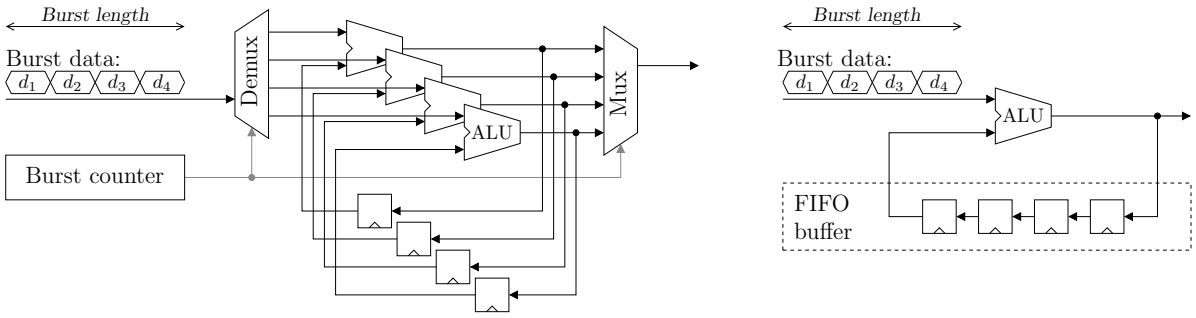


Fig. 5.4. There are two possibilities to handle the burst accesses. Left, a parallel architecture with replication of the hardware allows a longer processing of the data in each ALU. Right a time-shared solution requires much less hardware.

As seen in Fig. 5.4, the first solution (left) allows a longer processing of each data in the duplicated hardware structures, whereas the second solution (right) provides a more area-efficient hardware design based on the use of synchronous First In First Out (FIFO) buffers that let the data circulate in the same architecture. At this point, we would like to remind the architectural retiming ideas developed in Sec. 1.1.4 in Chapter II for the design of fast digital VLSI structures. Although more difficult to achieve, we go in our work with the second idea and settle for the smaller hardware solution.

2.1.2 Functional Computations

As the trigrams τ_n are processed sequentially, the bit $b_p[h(\tau_n)]$ read from the page p is the information which is needed to compute the similarity value between the query and the text from the database at time $t = t_n$. Neglecting the influence of the weight ω_n, we calculate first the mismatch penalty $\psi(n) \in \{0; 1\}$ as in (5.1) where only one bit of the vector is considered so that

$$\psi(t) = 1 - b_p(t) \xrightarrow{\text{logic.}} \psi(n) = \neg\, b_p(n) \tag{5.1}$$

and then the switch penalty $\sigma(n) \in \{0; 1\}$ which reflects the variations of the signal $\psi(t)$ in terms of edges as in (5.2) through

$$\sigma(t) = \left| \frac{d\psi(t)}{dt} \right| \xrightarrow{\text{logic.}} \sigma(n) = \psi(n) \oplus \left(\psi(n) \cdot z^{-1}\right) \tag{5.2}$$

where the $\oplus$ symbol corresponds to the modulo 2 addition, and z^{-1} to a delay element.

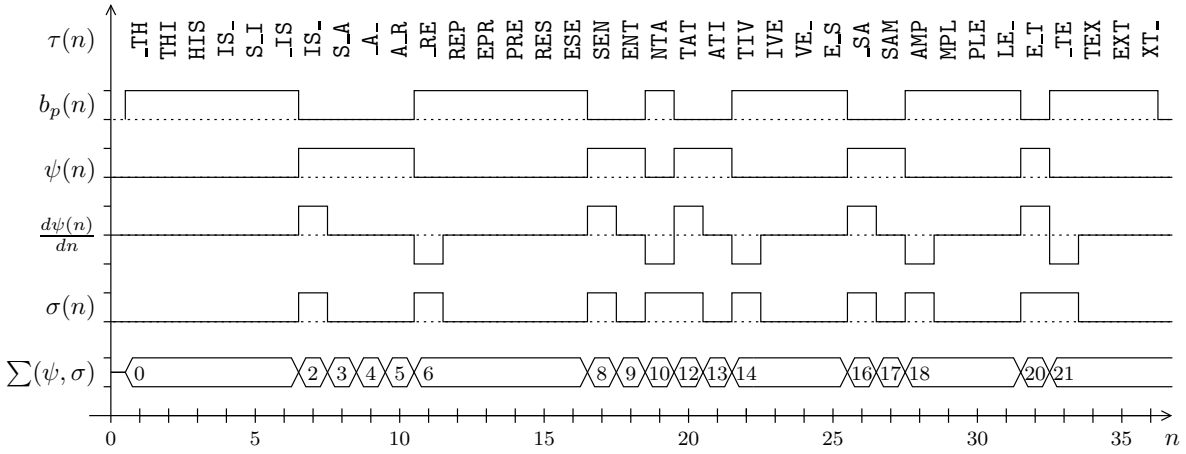
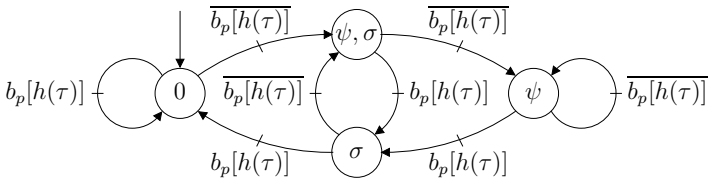


Fig. 5.5. Overview of the signal processing after the BAM access for one bit data $b_p(n)$ with the query string "`this is a representative sample text`", considering that $\omega_n = 1$ for all the slots in the query.



- $b_p[h(\tau)]$: BAM bit of the page p for query trigram τ.
- ψ : Mismatch penalty.
- σ : Switch penalty.

Fig. 5.6. DFSA for the calculation of the penalty of one fragment according to the binary value read from the BAM. The attributed penalties are written in the current state. The final result is obtained through summation of all the penalties.

The evolution of the processing of a query string at the bit level is depicted in Fig. 5.5. A sample text is transformed into a list of trigrams $\tau(t)$ using a time moving window of three characters. Synchronized on the processing clock of the system, the information bits $b_p(t)$ read from the BAM are converted into penalties considering an individual weighting ω_n of each slot n, here arbitrarily set to $\omega_n = 1$ for all n.

2.1.3 RTL Design

This section presents the RTL design of the PCU for the calculation of one fragment penalty $\mathcal{P}_p(f)$ based on a BAM bit $b_p(n)$ and the corresponding query weight $\omega(n)$. According to (3.9) and the previous section, this value can be obtained by determining sequentially the current penalties σ and ψ, and summing them up along the processing of each fragment. The Vertical Fragment Accumulator (VFA) contains the fragment penalty and can saturate automatically at $2^{n_p} - 1$. As seen in Fig. 5.6, we use a Discrete Finite State Automaton (DFSA) to calculate the penalties for the current query slot. It includes four states that correspond to the combinations of the two mismatch and switch penalties described in Sec.3.2.3 of Chapter III.

As seen in Fig. 5.7, the DFSA implementation is very small in terms of hardware resources and one register is sufficient for the four possible states, considering the current value of $\omega(n)$ and $b_p(n)$. Moreover, we remind that when a word in a query must not

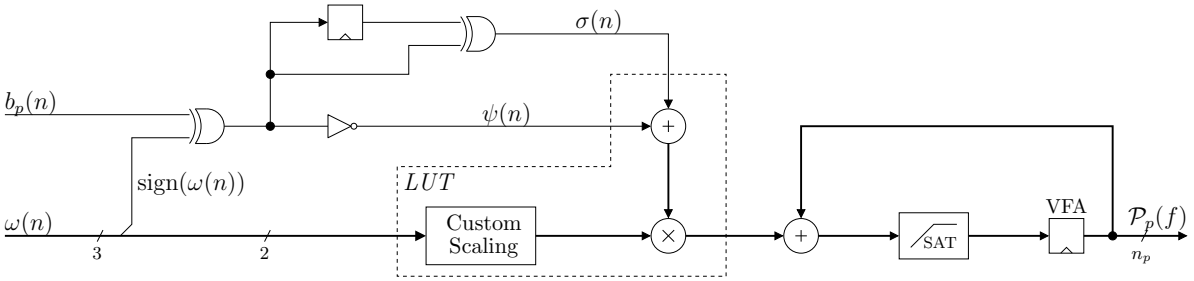


Fig. 5.7. Functional RTL design of the PCU corresponding to one single PE. This architecture is duplicated N times for the whole processing unit. The architecture must be extended according to Fig. 5.4 for a burst support.

be found, it obtains a negative weight. Therefore, the sign bit of $\omega(n)$ is taken to invert the BAM bit $b_p(n)$ using an XOR gate and penalizes the fragment when such items are found.

The custom parameterizability of the impact of the weights over the value of the penalty is set by a user-defined Look-Up Table (LUT) implementation of the adder and the multiplier. Although n_p is set to eight bits for a prototyping model, we keep on describing the design with scalable parameters, as changing the bit widths within the system is not a problem for hardware designers. The optimal values must be estimated by the algorithm designers and other information retrieval specialists.

An important point in the design is that the registers depth for the state machine and for the VFA must be set to four, according to the burst access mode in the BIF. In this case, data is processed at the BAM memory access clock frequency and $4N$ pages can be treated in four clock cycles using N PEs in the PCU in parallel.

2.2 Score Calculating Unit

Besides the functionalities of the SCU, this section presents the realization of a scalable architecture, the Negative Logarithmic Function (NLF), for the integer calculation of nonlinear functions based on a method for estimating the logarithm developed in [Lay04a]. A deeper analysis of the function is given in Appendix A and shows how to implement the desired analog logarithm and its reciprocal function with very little logic and a maximizable accuracy.

2.2.1 Description

The SCU is the second processing unit in the datapath of the ACE, the purpose of which is twofold. On the one hand, it builds the hit-table that records the fragments having a penalty lower than a given threshold. On the other hand, it transforms the fragment penalties of a query compared to a BAM page into a page score using a logarithmic function. However, calculating a logarithm in hardware is not a trivial task.

Investigations have been made in the Logarithmic Number System (LNS) [Kor93, Kos91, Tay88], particularly in the fields of massively parallel systems, Digital Signal Processing (DSP) [Lew95] and data processing for the easy computation of roots, powers, trigonometric functions, products and quotients [Hal70, Hoe91, Lew94, Mit62]. The use of logarithms accelerates multiplications and divisions by turning them into additions and subtractions. However, as we cannot achieve an infinite precision, a compromise has to be found between accuracy and complexity, so that a more tolerant application may benefit from trading off precision against speed and area for the evaluation of logarithms.

2.2.2 Functional Computations

The purpose of the NLF is to take the logarithm of an input value, using a flexible method in terms of scalability and relative accuracy. We design an architecture which fits best to the calculation of the page scores, as described in (3.10), therefore the output response of the NLF shall be described by

$$f_{\mathrm{NLF}}(x) = A_i - B_i \cdot \log_2(x+1), \tag{5.3}$$

where A_i and B_i are positive coefficients which only depend on the width i of the input x in order to scale the function correctly. They translate and resize the desired part of the curve into the upper right quadrant so that both x and $y = f_{\mathrm{NLF}}(x)$ values remain positive. For the hardware realization of the function, we use a piecewise linear approximation of the corresponding analog curve. This method is based on the Taylor/Mac Laurin series of the natural logarithm truncated after the first order through

$$-\log_2(1+x) = \frac{1}{\ln 2} \cdot \sum_{n=1}^{\infty} (-1)^n \cdot \frac{x^n}{n} \approx -\frac{x}{\ln 2} \tag{5.4}$$

which apply for $x \approx 0$. However, the method is easily extendable, as demonstrated through the graphical explanations in Appendix A. The factor $\ln 2$ appears in the slope of each tangent to the curve, at the abscissae which are powers of two, i.e., $x = 2^n, n \in \mathbb{N}$, as depicted in Fig. 5.8-b. By majorating the function $y = -\log_2(x+1)$, we eliminate the $\ln 2$ factor and obtain straight lines, the slopes of which are powers of two.

An example is given in Fig. 5.8-a for the NLF where the width i of the input x is 8 bit. In order to build the function, we split the x-axis into logarithmically increasing parts and divide the y-axis into constant equally sized parts. Afterwards we join the opposite corners of each thereby obtained rectangle through a straight line to form a continuous curve, as seen in Fig. 5.8-a. In this case, the abscissae and the ordinates are divided in $i = 8$ domains which characterize the eight segments composing the function.

As a matter of fact, a similar interpolation method has been suggested by Mitchell [Mit62] and expanded by Hall *et.al.* [Hal70] to a multiple piecewise linear approximation in order to improve the accuracy, but with more complexity and difficulty regarding its implementation. As Abed *et.al.* [Abe03a, Abe03b] presented the realization of a logarithm and antilogarithm converter using combinational logic only, the NLF proposes

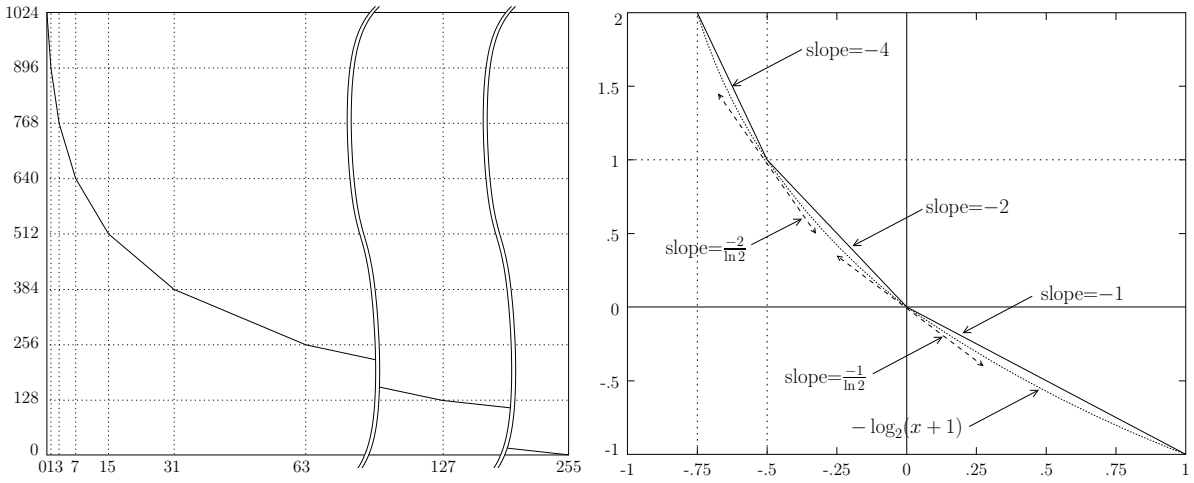


Fig. 5.8. Linear approximation of **a)** the NLF with an 8 bit input, which is based upon **b)** a first order approximation of the function $y = -\log_2(x+1)$ in the interval $[-.75; 1]$.

some interesting advances beyond Mitchell's approach, with the hardware implementation of a sequential architecture minimized in terms of gates and thus optimized for power and area sensitive applications.

Following the example given in Figure 5.8, the digitally approximated logarithm function is created by repeatedly cutting the remaining part of the x-axis into halves starting from the right to the left and stopping when the length of the last interval is one unit. The range of x is covered in i iteration steps, so that the maximum value of x reached yields

$$x_{\max} = \sum_{n=0}^{i-1} 2^n = 2^i - 1. \tag{5.5}$$

On the y-axis, we divide the range into i equal intervals, the length Δ_y of which only depends on the width i of x. We set this length to a constant equal to $\Delta_y = 2^{i-1}$. Since the NLF progresses from one step in both $\vec{x}$ and $\vec{y}$ directions per iteration, the maximum value of y is also reached in i steps and gives the upper bound

$$y_{\max} = i \cdot \Delta_y = i \cdot 2^{i-1}. \tag{5.6}$$

This relation shows that the method does not only combine input and output widths but also has an effect on the precision of the approximation, which is discussed in Appendix A. Thus we can estimate the amount j of bits needed in y to output the result considering the value $y_{\max}$. Using the rounding ceil function $\lceil x \rceil$ which computes the smallest integral value not less than x, we obtain the width j of y with

$$j = \lceil \log_2 (y_{\max} + 1) \rceil = \lceil \log_2 \left(i \cdot 2^{i-1} + 1 \right) \rceil. \tag{5.7}$$

For the next equations describing the NLF, we introduce a parameter n, the role of which is to simplify the writings and allow a better understanding. We attribute the index n to each interval, beginning with 0 for the start interval, i.e., the biggest one the

length of which equals the sum of all the others. The value of x only determines this index with

$$n = \lfloor i - \log_2(x+1) \rfloor = i - \lceil \log_2(x+1) \rceil \tag{5.8}$$

where the floor function $\lfloor x \rfloor$ computes the largest integral value not greater than x. Inside each interval, we generate a straight line which can be calculated through the parametric equation

$$y_{n,i}(x) = -2^n \cdot (x+1) + \Delta_y \cdot (n+2) \tag{5.9}$$

were i is included in Δ_y as well. This equation constitutes the basis for the hardware realization of a logarithmic function as it will be used for the hardware implementation of (3.10). As reported in Appendix A, the comparison of (5.3) with (5.9) gives us more information about the error due to the linear approximation.

2.2.3 NLF Architecture

We begin the RTL description of the SCU with the hardware implementation of the NLF based on the creation of an algorithm that computes a logarithm according to (5.9). The important point here is the use of simple operations that can reduce to additions and shifts, not even requiring any multiplication.

Algorithm N *(Negative Logarithmic Function).* This algorithm calculates the approximation of the binary logarithm according to (5.3) and (5.9). Let i and j be the respective lengths in bits of x and y according to (5.7).

N1. [Counting.] Count in n the number of leading zeroes in x.

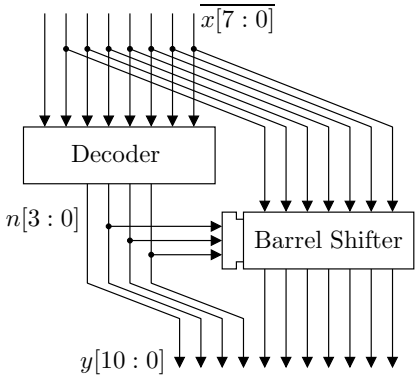
N2. [Inversion.] Invert all the bits of the input vector $x \rightarrow \bar{x}$.

N3. [Shifting.] Shift $\bar{x}$ of n bits to the left and keep $i-1$ LSBs.

N4. [Grouping.] Concatenate n and the result to obtain y. ∎

We propose two manners to realize the NLF. Though both are based on the logarithm through shifting principle, the first one is made of combinational logic and has been introduced in [Hoe91] whereas the second one uses a sequential method to calculate the logarithm. Following the development of the equations from the previous part, we discuss our architectures with the example of an $i = 8$ bit input vector x, as already seen in Fig. 5.8.

Fig. 5.9 shows the realization of the NLF using a decoder and a Barrel shifter. The decoder is a logical block which counts the number of insignificant zeros in x. The result is a four bit index n which gives an approximated logarithm of x as seen in (5.8) that builds the logarithmic offset for the upper bits of the output y. Because of the big amount of "don't care" values marked x in the truth table of the decoder, the remaining logical equations related to n are quite simple and require few logic gates. After the Barrel shifter, the shift of x of n bit to the left yields a multiplication of x



Decoder truth table

input x (8 bits)	n (4 bits)	y range (11 bits)
00000000	8: 1000	1024
00000001	7: 0111	896 – 1023
0000001x	6: 0110	768 – 895
000001xx	5: 0101	640 – 767
00001xxx	4: 0100	512 – 639
0001xxxx	3: 0011	384 – 511
001xxxxx	2: 0010	256 – 383
01xxxxxx	1: 0001	128 – 255
1xxxxxxx	0: 0000	0 – 127

Fig. 5.9. Realization of the NLF with a Barrel shifter for an 8 bit input x which leads to an output y of width 11 bits, according to the truth table of the decoder. The intermediate value n is used to control the amount of bits from which the input x is shifted to the left.

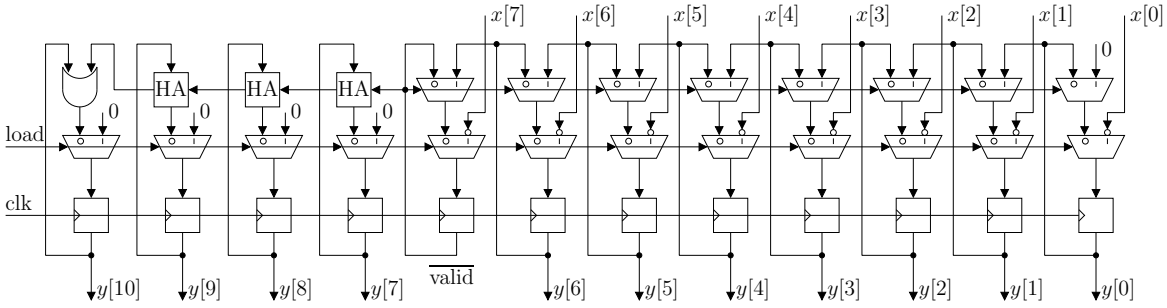


Fig. 5.10. Realization of the NLF for an 8 bit input with a recursive process where the output is recalculated as long as a shift is required. The processing stops automatically and sets the valid flag when finished.

with 2^n according to the linear approximation described in (5.9). The final result is the concatenation of the two values obtained above.

The second method is a recursive process where a clock is needed. The amount of logic needed is much smaller than for the first method. Figure 5.10 shows the realization of the NLF for an $i = 8$ bit input which yields an $j = 11$ bit output according to (5.7) and (5.9). To calculate a logarithm, the architecture needs at most as many clock periods as there are bits i in the input x. The multiplexers build a sequential shifter and can shift only one bit at a time, from the right to the left. Whether a shift is performed is decided by the upper bit (MSB) of x. On the one side, as long as this bit remains true, x is multiplied by 2, or 2^n after n iterations. On the other side, the counter formed by the half-adders (HA) and the OR-gate on the left of the design over the upper bits i to $j-1$ of y is incremented and yields $n \cdot 2^{i-1}$ after n iterations, conform to (5.9). The final result is the concatenation of the two values obtained above. Depending on the value of x, the result can be calculated in less than i clock cycles. The "valid" bit indicates that the shift operations are not over when high or that the result is valid when low. The purpose of such a signal is to report to the system that the calculation is terminated, because it takes a shorter time, i.e., only one clock period, to calculate the logarithm of big numbers ($x > 2^{i-1}$) than for small number ($x < 2^{i-1}$) where up to i clock periods may be needed. This process can be generalized for any width i of x.

2.2.4 RTL Design

The SCU calculates the score of a page and its FHT. Described in Fig. 5.11, the RTL design of a PE shows the 2 independent datapaths which include an accumulator, a comparator and a few registers besides the NLF module. On the top part of the design, the score $\mathcal{S}_p$ of a page p is calculated in the VQA (Vertical Query Accumulator) after processing of the penalty $\mathcal{P}_p(f)$ of a fragment f through the NLF module. The saturation of the VQA is necessary because the number of fragments is unknown and varies from one query to the other. For instance, we can set the width n_s of $\mathcal{S}_p$ so that four fragments with zero penalty produce the maximal score of $2^{n_s} - 1$. On the bottom part of the design, a digital comparator, i.e., a subtracter, outputs one bit which is stored in the registers of the FHT in a serial manner. Note that the reading of the register bank is performed in parallel for the processing in the following sorting units. Although we have to set the maximum number of allowed fragments in the query at design time in order to dimension the FHT, this decision again must be taken by the application designers. In any case, the values of n_h and n_s do not influence the architecture of the ACE at this level of development since the design principles stay the same.

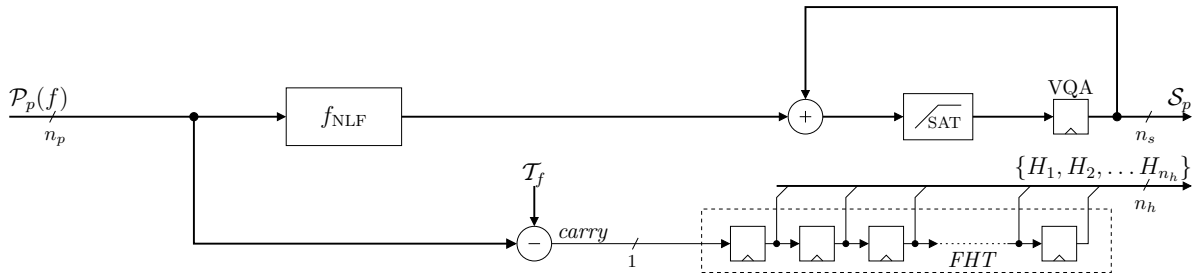


Fig. 5.11. The computational path of the Score Calculating Unit (SCU) is divided into two paths that independently actualize the score and the Fragment Hit Table (FHT) of a page regarding a query, based on the fragment penalty output by the PCU.

The accumulators present in both the PCU and the SCU are implemented with a carry-ripple structure as seen in Fig. 2.3 on page 12. Using pipelining, their speed can be increased to the system clock frequency, so that the throughput reaches the necessary value, allowing the direct processing of the data read from the memory subsystem. Aware of the power consumption reduction trend, Wilton *et.al.* have investigated experimentally the quantitative impact of pipelining on energy per operation and shown in [Wil04] that pipelining an FPGA design could reduce the energy consumption drastically. To this aim, Sec. 1.1.4 in Chapter II presents the necessary transformations which were applied to the architecture of the PCU and SCU modules. With these considerations, one of the main task of the scheduler is to ensure that $4N$ logarithms can be calculated with N PEs in the SCU within the time needed by the PCU to calculate $4N$ fragment penalties with its N PEs.

3 Hardware Sorting and Merging

This section is dedicated to the realization of a sorting engine in hardware, considering the embedded constraints in size, speed and throughput highlighted in the previous section for the implementation of the computational path of the ACE. We define a record as a data structure which includes a key and a body. In the ACE computations, the key of a record is the score of a page, and the body of the record includes the page number and the query-related FHT. However for clarity reasons in the pictures, we discuss the processing of keys instead of records in the RSU and the RMU.

3.1 Parallel Sorting with Bitonic Networks

Parallel sorting has been a well studied paradigm for both computerized applications in multiprocessor systems and VLSI realizations. For related surveys, we would like to refer to the bibliography, especially [Ajt83, Bat69, Knu97, Lay04b, Lei85, Ola99, Ola00, Par99, Tho83]. As a particularly interesting case seen in Sec. 3.3.2 of Chapter II, Batcher's bitonic merger and sorter [Bat68] has the capability of sorting N keys in $O(\log^2 N)$ time with $O(N \log^2 N)$ comparators. Although it owns a very regular architecture, it reveals itself not optimal in terms of area and depth, or in other words delay time. However despite a few dimensioning constraints, it remains the most suitable one regarding its structural properties.

3.1.1 Fast Sorting Networks

The bitonic sorter is based on a comparison network scheme in which many compare and exchange operations are performed in parallel. A particularity of this network is that the

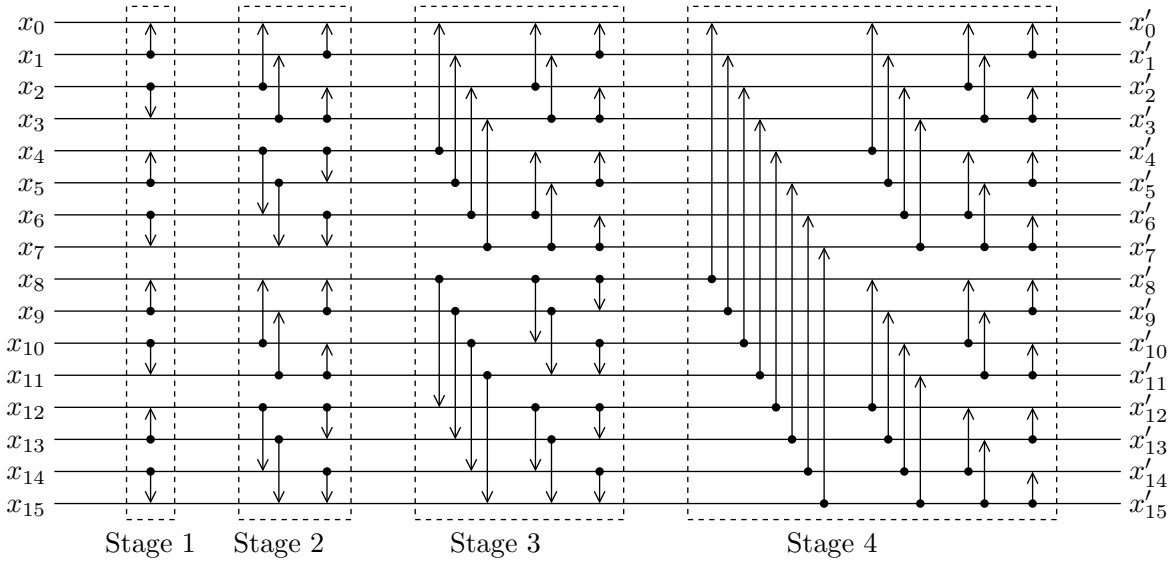


Fig. 5.12. Batcher's bitonic sorting network represented in a Knuth diagram for an input of $N = 16$ keys has an $O(\log^2 N)$ time complexity.

sequence of comparisons is absolutely deterministic and depends neither upon the initial state of the file of keys to be sorted, nor upon the result of the previous comparison step. Fig. 5.12 represents Batcher's bitonic sorting network in a Knuth diagram for an input of $N = 16$ keys, using both ascending and descending comparators. We call comparator $[i : j]$ a module comparing and interchanging its two inputs x_i and x_j if the minimum or maximum is not at the desired position on its two outputs x'_i and x'_j. According to text conventions for drawing networks [Knu97], an arrow from x_i to x_j is used to indicate that the larger number goes to the point of the arrow and the smaller to the base.

In such a sorting network, the $N = 2^n$ horizontal lines correspond to the inputs and the arrows to the switch comparators. The sorting consists of n successive merging phases i with $1 \leq i \leq n$, in which pairs of sorted sequences of length 2^{i-1} are presented in oppositely sorted order and then merged together, according to the bitonic principle [Bat68]. As many research groups have enhanced the bitonic architecture [Eve98, Mut99, Sto71, Tho83], we chose one which allows further modifications of both hardware size and internal scheduling.

3.1.2 Network Implementation Using Recirculation

Stone [Sto71] showed that a sorting network for $N = 2^n$ elements could be constructed by following a regular pattern, as illustrated in Fig. 5.13 for $n = 4$. Each of the n steps in this scheme consists of a perfect shuffle of the first $2^{n-1} = \frac{N}{2}$ elements with the last $\frac{N}{2}$, followed by simultaneous operations performed on $\frac{N}{2}$ pairs of adjacent elements. As seen in Fig. 5.13, each of the latter operations is either a "$\varnothing$" for no operation, a "$\uparrow$" for an ascending comparison, or a "$\downarrow$" for a descending comparison.

During a stage s, for $s < n$, $n - s$ steps are operated with "$\varnothing$". They are followed by s steps in which the operations within step t consist alternately of 2^{t-1} "$\uparrow$" followed by 2^{t-1} "$\downarrow$". During the last stage, all operations are "$\uparrow$" and constitute in fact an N

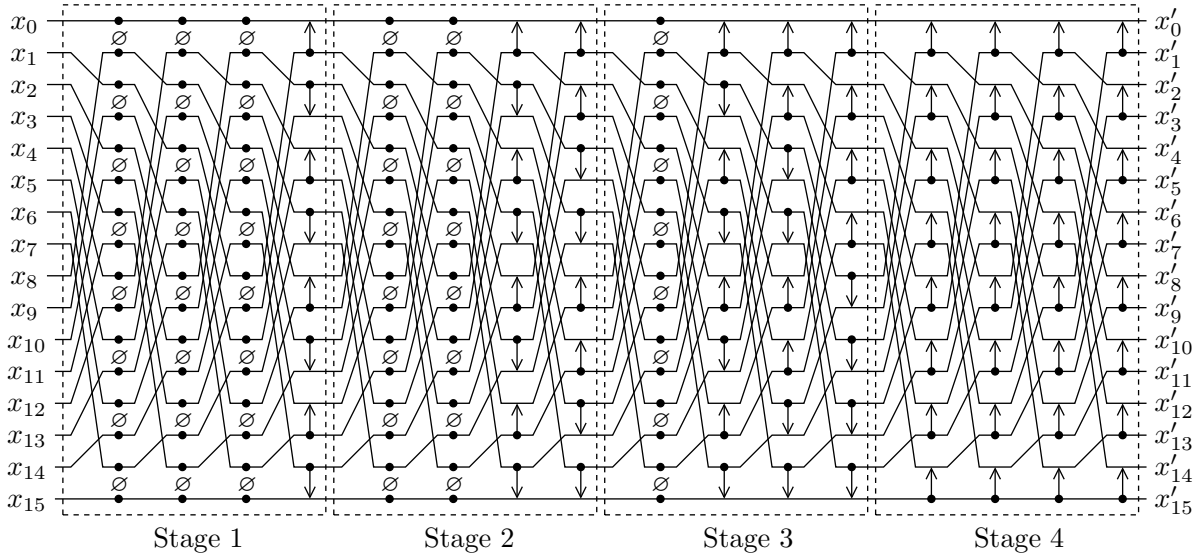
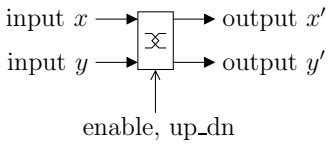


Fig. 5.13. Representation in a regular structure of the bitonic sorting network in a Knuth diagram using Stone's perfect shuffle on $\log_2 N$ stages of $\log_2 N$ steps.

Fig. 5.14. Set of comparator elements including the computational ascending ↑ and descending ↓ comparators, as well as the direct run-through and crossing elements which are used for a symbolic representation of the networks.

keys bitonic merger for two monotonic series of length $\frac{N}{2}$. The different operations are summarized in Fig. 5.14. From the hardware point of view, it appears clearly that this network structure can be simplified to a single column of $\frac{N}{2}$ special switch comparators executing a predefined sequence of operations, at the cost of $\log_2 \frac{N}{2}$ basically neutral but necessary steps, i.e., the ones performed by the comparators switched in run-through mode in the middle of the network.



enable	up_dn	x'	y'
0	0	x	y
0	1	y	x
1	0	$\min(x, y)$	$\max(x, y)$
1	1	$\max(x, y)$	$\min(x, y)$

Fig. 5.15. Controlled switch comparators are processing elements that permit the realization of the four operations necessary in sorting networks as seen in Fig. 5.14. Two control signals are sufficient to program the comparator.

For this reason, we created a new type of switch comparator to permit the realization of a recurrent bitonic sorting network, as shown in Fig. 5.15. In order to perform the perfect shuffle, a butterfly network is built symmetrically by connecting regularly the half of the inputs to each second output, from $i = 1$ to $\frac{N}{2}$ and mirrored from $i = N$ to $\frac{N}{2} + 1$. This sorter processes N keys in $O(\log^2 N)$ time with only $\frac{N}{2}$ elements. An additional external counter generating the control signals "enable" and "up_dn", supervises the sorting procedure. Moreover, considering that the first run-through comparisons can be disregarded, the controller can skip the insignificant steps of the first stage to accelerate the whole sorting. The final remaining time for this model is then expressed in the following equation in terms of clock cycles with

$$t_{\text{sort}}(N) = (\log_2 N)^2 - \log_2 N + 1. \tag{5.10}$$

In this sorting scheme, the keys are input at once and processed in parallel. Their length L does not influence t_{sort}. While the time complexity $O(\log^2 N)$ of the sorter in Fig. 5.16 is the same as the original bitonic sorting network in Fig. 5.12, its complexity in terms of logic gates and comparators has been drastically reduced from $O(N \log^2 N)$ to $O(N)$. However, for large N and L, as it is the case in most applications, the practical realization of such a network yields a moderate performance due to the high routing density. Therefore, we have to consider further optimization methods for an efficient implementation of the bitonic merger into physical devices, e.g., FPGAs.

3.2 Optimization Methodologies

Introduced in Sec. 1.1.2 and 1.1.3 of Chapter II, some methods for optimizing the performance of digital circuits allow system architectures to run faster and increase the data throughput. Applying these methods to our sorting schemes which own a feedback loop, require special attention in the retiming procedure as explained in Sec. 1.1.4 of Chapter II.

3.2.1 Network Retiming

When it comes to the hardware implementation of such a sorting network that includes a perfect-shuffle mapping, a price has to be paid in terms of routing resources, especially in FPGAs. Previous studies [DeH00a, Eve98, Mut99, Szy97, Yeh00, Yeh03] have shown the influence of the butterfly pattern over the complete architecture of the system in VLSI designs. The recurrent bitonic sorting network presented in the previous section is scalable in the sense that it is adaptable to any kind of bus width, as long as the number of key to sort is a power of two ($N = 2^n$). Slightly diverging realizations of redistributing networks for sorting were found in the literature where the main idea was either to use special design environments to verify the lengths of intermediate wires [Cla01] or to resort to a parity strategy to reduce the communication within the sorter [Lee00].

The introduction of a new parameter p representing the degree of parallelism inside the network means a trade-off between area and throughput for the sorter. It corresponds to the granularity at which the keys are internally processed in the comparators

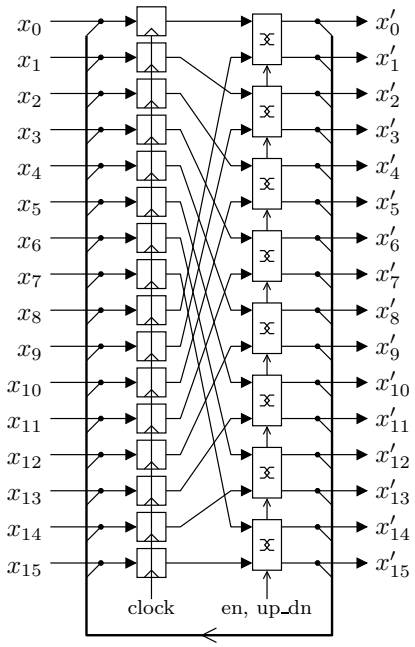


Fig. 5.16. An $N = 16$ keys recurrent bitonic sorting network uses controlled switch comparators according to a predetermined shuffle and operate scheme in $O(\log^2 N)$ time complexity. The feedback permits the recirculation of N keys of L bits in parallel.

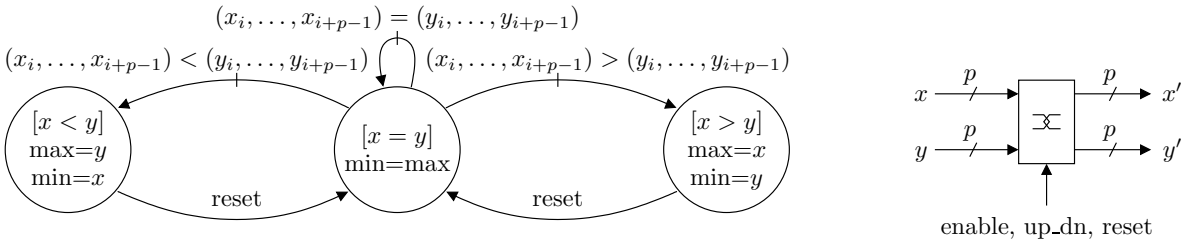


Fig. 5.17. Retiming of the comparator modules with an internal degree of parallelism p for the pseudo-serial processing. An internal finite state automaton is used to remember the result of previous comparisons on partial words.

and within the network. Fig. 5.17 shows the state diagram for a pseudo serial comparison element in which the keys are inserted Most Significant Bit (MSB) first. As a consequence, with L the length of one key, if $p = 1$ corresponds to the completely serial solution and $p = L$ the fully parallel solution, then (5.10) becomes

$$t_{\text{sort}}(N,p) = \left((\log_2 N)^2 - \log_2 N + 1\right) \cdot \left\lceil \frac{L}{p} \right\rceil \tag{5.11}$$

whereas this duration can only be obviously an entire amount of clock periods. The main benefit is for the area of the sorter, as it varies in $O(p^2N^2)$. Though the throughput is accordingly reduced, it is foreseeable that the global routing delay might at our advantage be shortened as well.

3.2.2 Network Recombination

Although a partitioning of the standard bitonic sorter works with arbitrary or mixed sizes [Nak89], we chose to implement an N keys sorting engine based on two $\frac{N}{2}$ keys bitonic sorters and one N keys bitonic merger, all with recirculation. Not only do they have different widths and hence different sizes and areas, they also require unequal sorting and merging time.

As seen in Fig. 5.18, we built a sorter based on these two kinds of networks that handle N keys in parallel. The condition for achieving the maximum throughput is that the time $t_{\text{sort}}(\frac{N}{2}, p_1)$ in $O\left(\log^2 N\right)$ needed to sort $\frac{N}{2}$ keys with a granularity p_1 is at most equal to the time $t_{\text{merge}}(N, p_2)$ in $O\left(\log N\right)$ needed to merge the N resulting keys. As we try to minimize the overall area of the architecture, i.e., with only one level of comparators, we obtain a relationship between p_1 and p_2 with

$$\underbrace{\left(\left(\log_2 \frac{N}{2}\right)^2 - \log_2 \frac{N}{2} + 1\right) \cdot \left\lceil \frac{L}{p_1} \right\rceil}_{\frac{N}{2}\text{keys sorting time}} \geq \underbrace{(\log_2 N) \cdot \left\lceil \frac{L}{p_2} \right\rceil}_{N\text{keys merging time}} \tag{5.12}$$

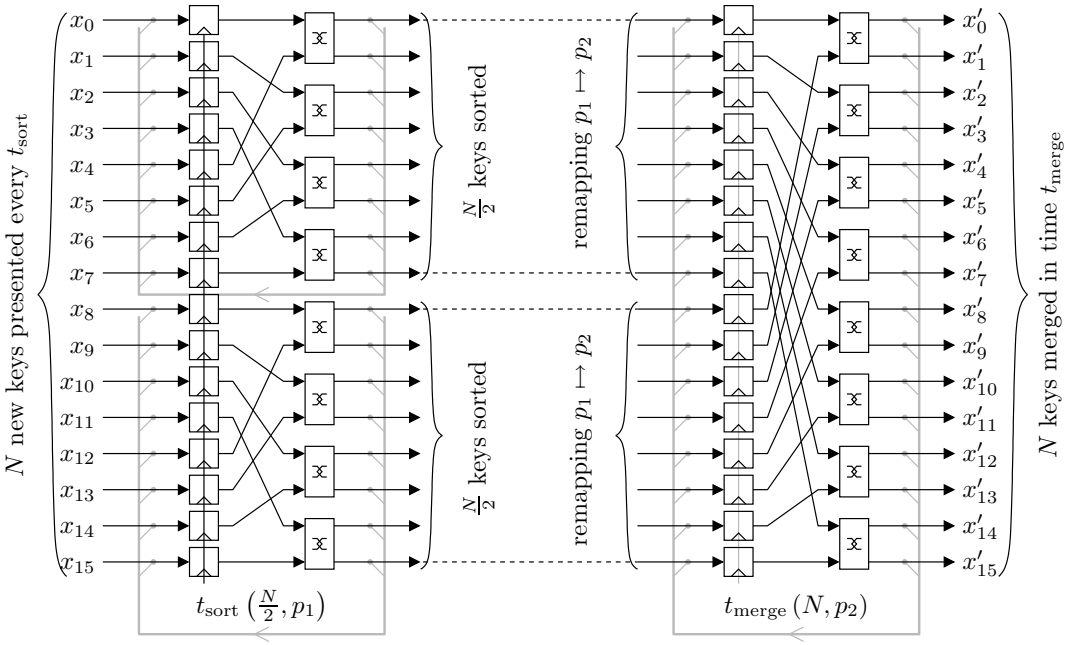


Fig. 5.18. Recombination of two recurrent 8 keys bitonic sorters followed by a recurrent 16 keys bitonic merger. The degree of bit parallelism in the processing of the keys is expressed by the variables p_1 and p_2 respectively.

whereas since both are integers, p_2 might be set to the next suitable value. Hence we can resolve (5.12) for expressing p_2 as a function of p_1 with

$$p_2 \geq \frac{\log_2 N \cdot p_1}{\left((\log_2 N)^2 - 3\log_2 N + 3\right)}. \tag{5.13}$$

As a rule of thumb, we can choose $p_2 \geq \lceil p_1 / (\log_2 N - 2) \rceil$. Between sorters and merger, the remapping $p_1 \mapsto p_2$ is simply done using registers. If necessary, the serial insertion of zero-bits can be used to resynchronize a bitonic network without influencing the sorting functionality.

3.3 Hardware Merging Solutions

In order to realize the RSU plus RMU ensemble, we have studied different sorting and merging possibilities at the algorithmic level. Regarding our application, the hardware constraints lie in the throughput of the system, i.e., in the size of the incoming data, in the frequency and in the the length of the result list, that is to say the amount of records which have to be returned from the ACE to the main application running on the host PC.

3.3.1 Proposal based on the Bitonic Sort Algorithm

The first proposal refers to an extension of the architecture presented in Fig. 5.18 where instead of merging two groups of $\frac{N}{2}$ keys, we could use a feedback channel around the bitonic merger to sort N new keys together with R old keys already present in the sorted list [Lay03]. As seen in Fig. 5.19, the method works fine, but its realization is quite costly in terms of hardware resources, as the butterfly network of size $N+R$ present in the RMU must be fast. The use of retiming parameters for a more serial processing of the keys as explained in Sec. 3.2.1 might badly slow down the processing, i.e., minimize the throughput, and make this approach rather unsuitable within the ACE for non small values of R or N.

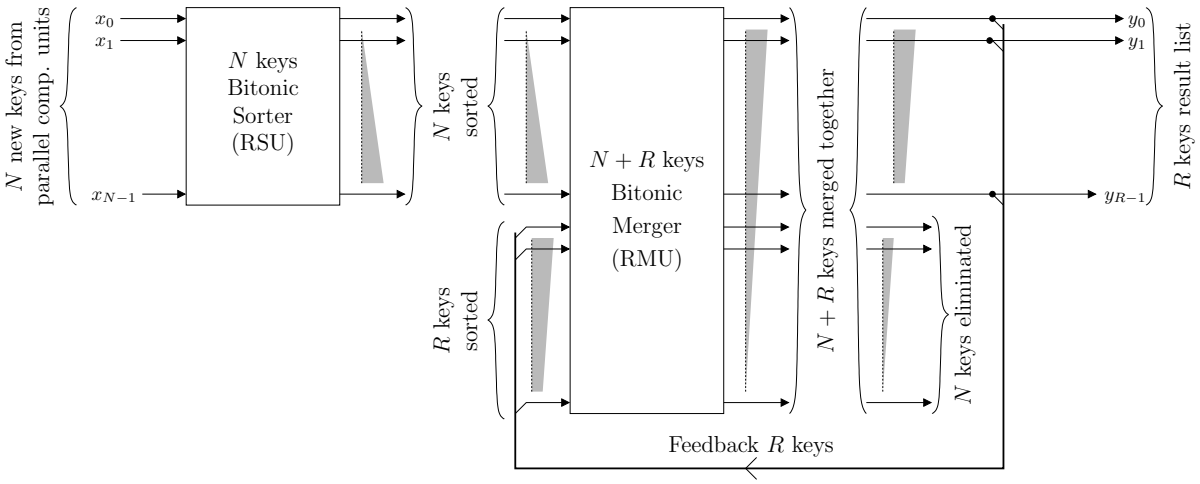


Fig. 5.19. The output of the RSU composed of N sorted keys enters the RMU which uses a large bitonic sorter with a feedback of R keys to regenerate the result list.

3.3.2 Proposal based on the Insertion Sorting Algorithm

Based on the insertion sort algorithm of Chapter II Sec. 3.1.1, we propose a new architecture for merging data in a freely scalable result list. The idea rests upon a time for area tradeoff and the stipulation that after a short amount of iterations, since the pages are randomly distributed in the BAM, the probability is extremely low that one of the N sorted pages from the RSU needs to be inserted in the result list by the RMU. This justifies the choice of an algorithm based on insertion instead of a full $N+R$ merging network. In this case, we rely on a sequential sorting procedure which compares the highest key of the N sorted list and insert it in the R result list if necessary. Only R comparisons are necessary to find the position of the new record in the list. Moreover, the complete N-set of records from the RSU can be ignored if the highest key of this set is lower than the lowest key of the result list.

A further hardware improvement is allowed considering the time available to perform the merging in the RMU. Instead of executing one comparison, chucking out the whole N bunch of low RSU records and waiting for the next arrival as it is very likely to happen after a few iterations, we can use the time to perform more comparisons, and

hence reduce the hardware complexity of the sorting in the RSU. As a result, the data from the RSU is not N-sorted anymore, but becomes $\frac{N}{M}$-sorted. Firstly this decreases the size of the butterfly networks present in the bitonic sorters of the RSU and secondly reduces the duration of the sort itself.

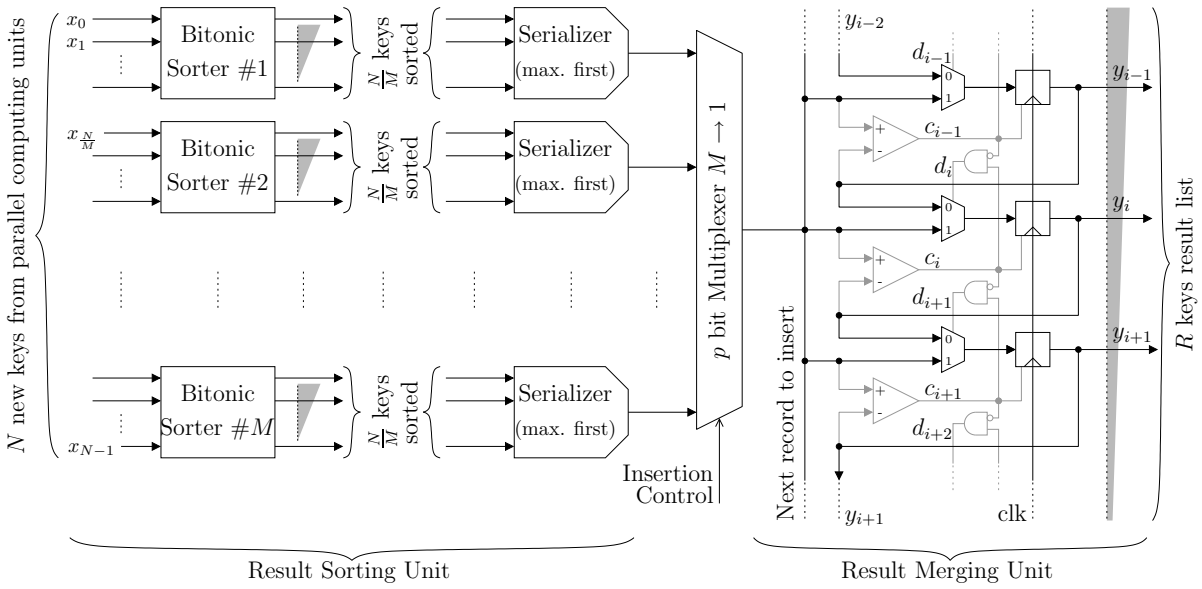


Fig. 5.20. The RSU is composed of M bitonic sorters that present their highest score serially to the RMU over a multiplexer. If the selected key is greater that the lowest one from the result list, then a direct insertion and shift is performed.

Fig. 5.20 shows the details of the architecture of the RSU and of the RMU. A small state machine is necessary for correctly advancing the records in both the serializers and the multiplexer. For a better comprehension of the architecture, the control logic within the RMU which enables the shifting appears in grey and the data lines are black. In the RMU, the R bit vector c in the form $\{c_0 \dots c_{R-1}\} = \{0 \dots 01 \dots 1\}$ is the result of R parallel comparisons and enables the corresponding registers for shifting, whereas d in the form $\{d_0 \dots d_{R-1}\} = \{0 \dots 010 \dots 0\}$ indicates the position of the insertion of the new record in the list. At each clock, the new value can be placed into the result list which shifts out its lowest record y_{R-1} if necessary.

Considering a hardware realization, the main drawback of this proposal is the huge fan-out at the output of the register storing the next record to be inserted. Its key is presented to every single comparator in order to decide directly of the position where to insert the record. As the problem in this design does not come from a propagation delay over a critical path, the cut-set technique cannot be usefully applied here. The remaining solution would be to oversize the drivers of the large fan-out bus, hence increasing the power consumption and the signal propagation time. Although the model has a functional sorting time complexity in $O(N)$, hardware synthesis results have demonstrated that such an alternative is unsuitable for large R as the whole design suffers from a low clock frequency.

3.3.3 Proposal based on the Bubble Sort Algorithm

To replace the insertion sorter in the RMU, our final solution is based on Algorithm B presented page 27 in Sec. 3.1.3 of Chapter II. Though not the fastest, the bubble sort is the most simple algorithm realizable in hardware in both parallel or serial forms. A sorter based on the odd-even transposition algorithm has been presented in [Hen98] and integrated in a reconfigurable system [Blu00]. They use a bit serial processing and a time sharing architecture which drastically reduce the area of the hardware implementation.

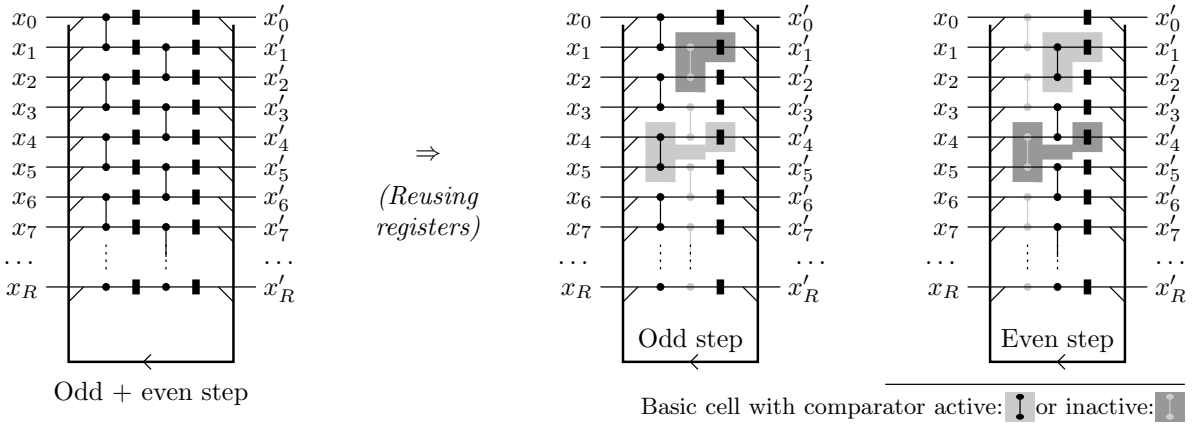


Fig. 5.21. Recurrent sorting network based on the odd-even transposition network (left) can be simplified if every second comparison can be alternatively disabled (right). A basic cell includes one comparator element and one register.

Fig. 5.21 (left) represents a recurrent sorting network based on the odd-even transposition, as seen in Fig. 2.15-b on page 30. A very regular structure can be obtained by implementing two levels of comparators plus two levels of registers. Depending on the parity of R, although the last even step might be unnecessary, it remains effectless, as opposed to a bitonic step. With the ability of reusing the same registers for both odd and even steps, as well as disabling every second comparator accordingly, the size of the structure can be reduced, as seen in Fig. 5.21 (right). As a result, it becomes possible to build an arbitrarily long result list composed of one single type of basic cell including a register with an input selector and a controlled comparator element.

The RTL structure of Fig. 5.22 represents the sorting unit based on the parallel bubble sort principle which is implemented in the RMU. The difference is in the continuous sequential insertion of the records at one extremity of the chain, as seen in Fig. 5.23, instead of a parallel insertion and a following $O(N)$ processing. The upper part of the design including the multiplexers and the registers constitutes the storing structure, for which the data-busses have the width of the records which have to be sorted. The bottom part of the design controls the data-flow of the storage structure on a biphased comparison process, i.e., first compare and then exchange. In this case, the time complexity is linear in $O(N + R)$ for the insertion of N records in a result list of length R. According to Fig. 5.21, this procedure corresponds to an input at the position x_R and a consequent feedback x_0 to x_{R-1} The result list corresponds to the register outputs x'_0 to x'_{R-1}, whereas the minimum x'_R of the list is rejected.

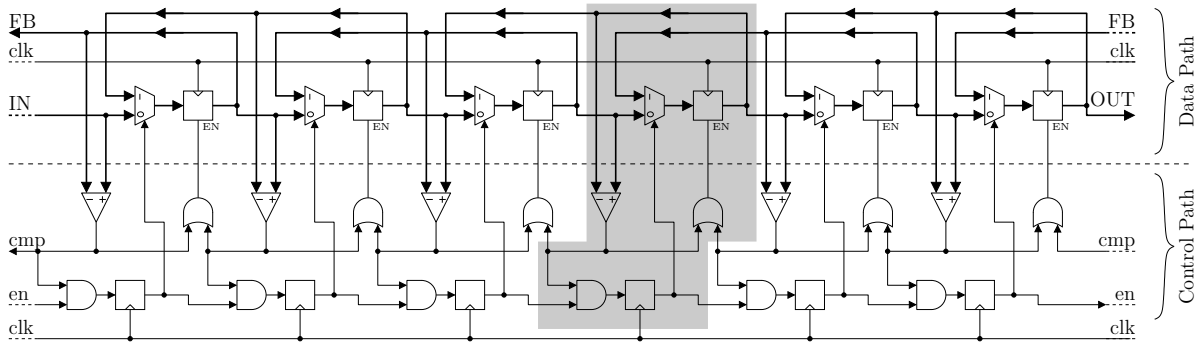


Fig. 5.22. Bidirectional structure for the result list based on the bubble sort algorithm which allows the registered values on the upper part of the design to move up and down in the list, i.e., right or left in the picture.

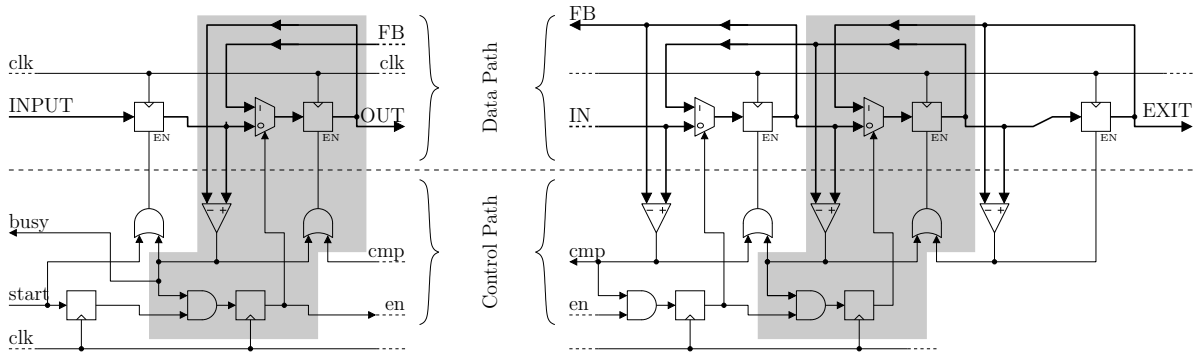


Fig. 5.23. Starting and ending extremities of the bidirectional structure (direct bubbling and feedback) for the result list based on the bubble sort algorithm.

In our architecture, we have foreseen the next step in the search procedure which consists in returning the best records stored in the result list to the host PC. In order to simplify the pictures, a special signal has been omitted in the representation of the result list on Fig. 5.22 and Fig. 5.23. It is the control signal used to read and empty the list, which has the effect of switching all the multiplexers onto the "0" position such that the registers form a straight forward chained list. On the activation of this signal, all the registers are enabled, and the control part of the design with the comparators is disabled. Hence with each clock period, data goes from the input register to the final output register, pushing out the contents of the list through the exit port.

In summary, we have seen that the bitonic sorting network permits a relatively fast sorting in $O(\log^2 N)$ time by using a very regular structure which eases its hardware implementation. From the algorithmic point of view, it is almost impossible to say which solution is best suited for our sorting paradigm, as the whole computing path influences both the speed and the amount of data that must be processed in the RSU and RMU. However, the bubble sort based RMU is the most promising realization in terms of flexibility, speed and area, when it is placed after a subdivided parallel implementation of recurrent bitonic sorting networks.

Chapter VI

Results and Evaluation

WITHIN Chapter V, we have presented innovative methods for performing the necessary associative computations, as well as the subsequent sorting and merging operations that require extremely short durations while using a minimal amount of hardware, according to the functionalities of the Associative Access Method (AAM) algorithm introduced in Chapter III and analyzed in Chapter IV. However, the so far described abstract architecture must be scaled according to the targeted hardware platform. Hence, in this chapter, we provide the ground rules for dimensioning the internal core of the ACE following a synchronization strategy which allows the maximum data throughput at the BAM interface while keeping the circuit design area as small as possible.

1 Hardware Implementation

This section is dedicated to the implementation of the ACE on a typical FPGA platform for the realization of a prototyped system which permits to verify practically the efficiency of our system. In designing a scalable and flexible architecture, we provided a solution suitable for most of the hardware platforms currently available on the market, while keeping the possibility to build one ourselves based on either FPGA or ASIC technology. Moreover, in the development of the high-throughput memory controller, we focused for various reasons on standard SDRAMs which are currently the most popular kind of memory devices for applications requiring a large storage space.

1.1 Adaptation to the Development Platform

According to Fig. 4.1 on page 52, the necessary hardware platform must provide a large memory subsystem connected through a very high bandwidth bus to a customized logic

chip, e.g., an ASIC or an FPGA, further on connected to a host PC over a generic interface such as Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB).

1.1.1 Prototyping Board

Developed at the University of Paderborn for the implementation of microelectronics circuit prototypes, the modular rapid prototyping system Raptor 2000 [Kal02] integrates all the important components to realize circuits and system designs with a complexity of up to 60 million transistors. The main properties of the board are the ability of coupling the system with a PC through the PCI interface, the possibility to mount many extension modules on the same motherboard and the presence of an additional bus, i.e., a broadcast bus, which releases the multi-master Local Bus providing the PCI connection. As seen in Fig. 6.1, a hardware module includes an FPGA of the Xilinx VirtexE family [Xil06] which can emulate circuits with up to 2.5 million system gates as well as 128 MB of SDRAM connected to the FPGA over a 32 bit wide data bus. Additionally, the FPGA requires a regulated power supply, a few clock signals for its internal architecture and the synchronous peripherals at typically 50 MHz, some LEDs for status indication and a Joint Test Action Group (JTAG) interface for its programming.

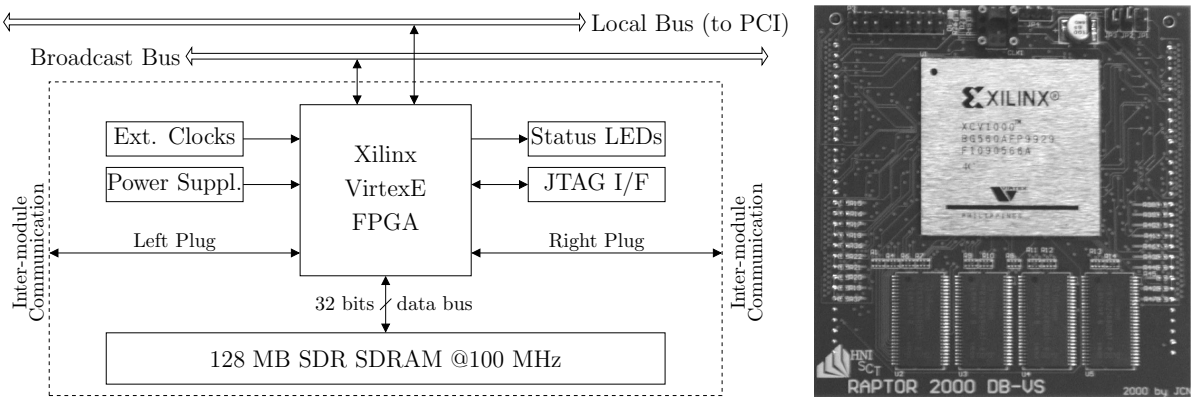


Fig. 6.1. The hardware prototyping module includes a Xilinx FPGA of the VirtexE series and four memory chips for a total of 128 MB SDRAM accessible over a 32 bit data bus at 100 MHz.

As of 2007, this platform can be safely qualified as outdated, since technology has progressed greatly, not only in terms of transistor sizes but also in terms of standards. According to Sec.1.3 in Chapter II, today's FPGA devices have become much faster and an order of magnitude larger than five years ago. With the use of very elaborated Electronic Design Automation (EDA) tools, they are able to implement designs efficiently, benefiting from additional internal hardware resources such as multipliers, Digital Signal Processors (DSPs) and microprocessor cores. Moreover, following the needs of computerized applications, memory devices have seen their capacity increase, as well as the format and speed of their interfaces evolving from SDR (Single Data Rate) to DDR (Double Data Rate) and QDR (Quad Data Rate). On the peripheral side, the PCI interface is becoming obsolete in that technical evolution leads to changes in data

width from 32 to 64 bits, clock frequency from 33 to 166 MHz and transfer mode from a parallel to a multi-channel serial protocol, thus gaining in crosstalk noise and skew immunity. However, in choosing one of the most basic platforms available with down-scaled hardware resources, we set the lowest bound in terms of performance which can be expected from any later dated FPGA implementation.

1.1.2 System on Chip Design

Though fully functional, the core of the ACE is not ready for a bigger system integration, as depicted in Fig. 4.8 in Chapter IV. It needs to be connected to the peripheral devices that provide the communication with the other hardware elements present on board and with the software programs driving the execution of user commands.

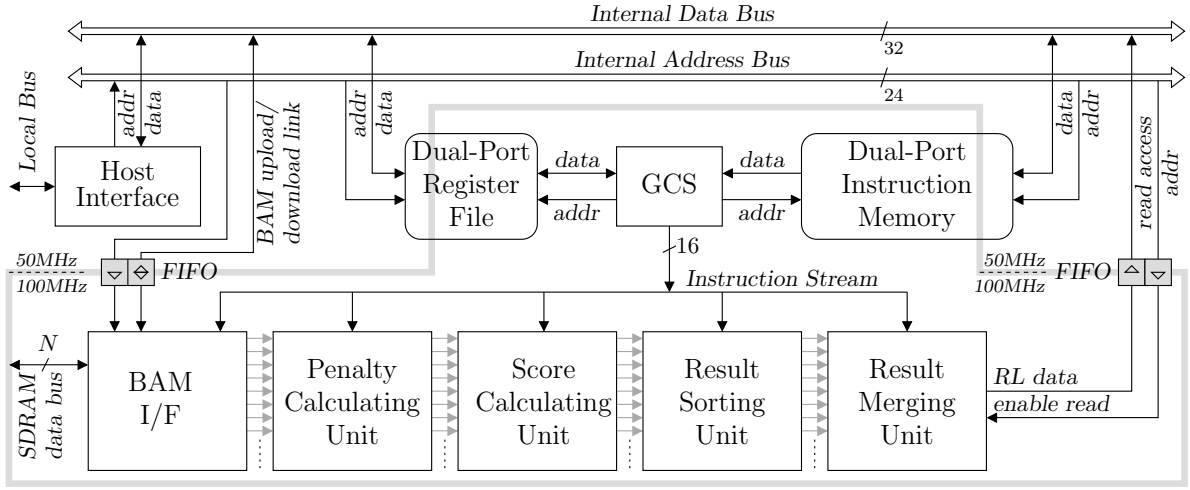


Fig. 6.2. The 100MHz processing core delimited by the grey area is included in a system on chip environment that provides peripheral connections to the hardware resources available on board, i.e., PCI through the Local Bus interface and SDRAM.

As seen in Fig. 6.2, our SoC (System on Chip) design includes the computational path of the ACE core which is clocked with the SDRAM frequency of 100 MHz, allowing a synchronous data exchange between all the modules from the BAM interface to the RMU. The host interface is clocked at 50 MHz using a different source synchronized on the 100 MHz clock through DLLs (Delay Locked Loops). The implementation of dual-ported RAM for the register file and for the instruction memory permits the concurrent access on two different sides of the registers at two different clock frequencies. However, the transfer of the data to the SDRAM for uploading the BAM, or from the RMU for reading the result list, must occur through synchronizing FIFO buffers. Although the controlling of which costs a few supplementary logic gates, it prevents reading the FIFOs when empty and writing them when full.

1.2 Synchronizing the Processing Units

As we have specified in Chapter IV, the ultimate goal remains the fastest processing of the data at the lowest possible hardware costs. Therefore it is essential to plan an efficient synchronization strategy. Since we cannot achieve a lower power consumption in an FPGA design by tweaking the size of the transistors, we can impact the overall switching activity in reducing the amount of gates involved in the computational path. Paradoxically, we transition on the one hand from a temporal to a spatial design, i.e., raising the number of transistors in the design, in order to increase the degree of parallelism and hence the data throughput. On the other hand, we have to reuse the available resources within the concurrently operating units when timing constraints allow it, e.g., using feedback or recirculation, in order to lower the size of the hardware structures.

1.2.1 Dimensioning the Data Path

The four units of the ACE are coordinated within a functional pipelined model that must allow the highest possible throughput at the BIF (BAM Interface). Compared to Fig. 4.8 on page 63, there subsists a certain difficulty to directly identify the PEs in the RSU and in the RMU according to Fig. 5.18 and 5.22. This is due to the fact that the represented layout only aims to highlight the functionally separated blocks and propose a high-level representation of the entire system at the transaction level. The RTL design of the different units composing the ACE is reported in Fig. 6.3 in Chapter V. With a vertically repeated structure, PCU and SCU work on N data in parallel. As the BAM vectors are read in a burst of length four, the PCU accumulates $4N$ penalties to be transfered to the SCU over four buffers that form a synchronous FIFO register in order to cancel the burst effect. Hence the SCU treats the penalties independently without noticing the way data is read at the memory interface. Composed of respectively $2M$ and M times the same structure, the RSU is split into two functional subunits, according

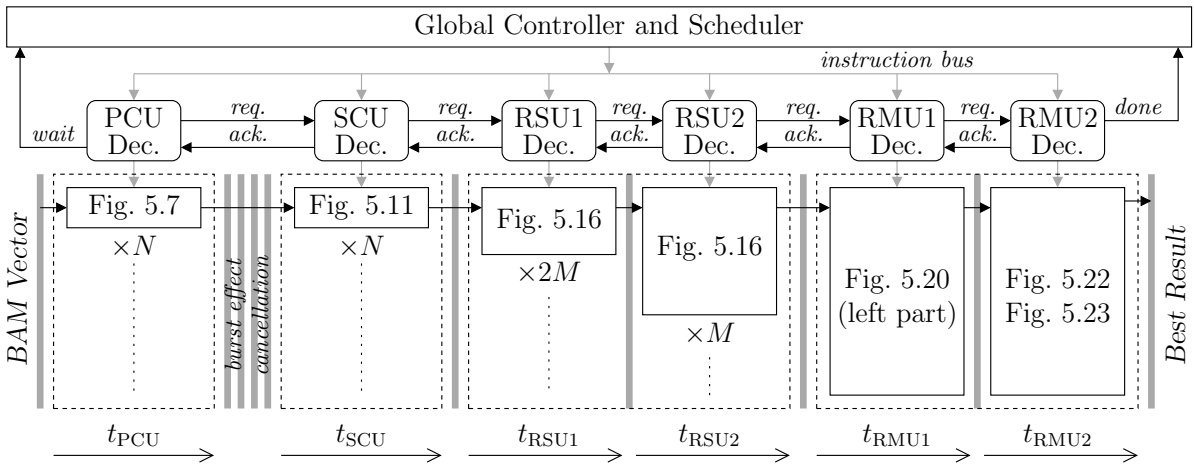


Fig. 6.3. The self-timed data path includes data transfers registers (grey lines) around each processing unit, as well as a burst effect canceling FIFO between PCU and SCU. The decoders synchronize the instructions from the GCS using a simple request/acknowledge protocol.

to the RTL design implementation we have described in Sec. 3.1 of Chapter V. As well, the RMU includes two main processing steps that perform first the select and compare operations and then the eventual bubble sorting. The synchronization of each unit occurs according to a self-timed control path where the instructions from the GCS (Global Controller and Scheduler) can be delayed by each unit decoder upon availability of the processing resources in the following unit by using a simple request-acknowledge protocol.

Synchronizing a parallel computing system including different processing units is not a trivial task. Fortunately, the ACE benefits by construction from precise and deterministic timings in almost all its computing modules. The two unknown parameters during the processing are first the length of the query string, which influences the duration of computing in the PCU, and second the amount of candidates at the output of the RSU to be inserted in the result list, which impacts the duration of the merging in the RMU.

With the frequency f_{clk} at which the N-bits vectors are read by the BAM interface, we can express the processing time of each unit. Hence, we define the functional time t_{PCU} as the time needed by the PCU to provide a single fragment penalty $\mathcal{P}_p(f)$ through

$$t_{\text{PCU}} = s \cdot T_{\text{clk}} \tag{6.1}$$

with s the amount of slots in a fragment f and $T_{\text{clk}} = f_{\text{clk}}^{-1}$ the period of the global clock of the data path. The functional time t_{SCU} corresponds to the time needed by the SCU to transform a fragment penalty into a page score $\mathcal{S}_p$ in the worst case through

$$t_{\text{SCU}} = n_p \cdot T_{\text{clk}} \tag{6.2}$$

where n_p is the number of bits in $\mathcal{P}_p(f)$. Further on, t_{RSU1} and t_{RSU2} correspond to the time needed by the RSU in a pipelined manner to respectively sort $\frac{N}{2M}$ and merge $\frac{N}{M}$ page scores according to (5.11) and (5.12) so that

$$t_{\text{RSU1}} = \left(\left(\log_2 \frac{N}{2M} \right)^2 - \log_2 \frac{N}{2M} + 1 \right) \cdot \left\lceil \frac{L}{p_1} \right\rceil \cdot T_{\text{clk}} \tag{6.3}$$

$$t_{\text{RSU2}} = \left(\log_2 \frac{N}{M} \right) \cdot \left\lceil \frac{L}{p_2} \right\rceil \cdot T_{\text{clk}} \tag{6.4}$$

with p_1 and p_2 two temporary internal processing widths according to (5.13). The functional time t_{RMU1} is the time needed by the RMU to check at least M and at most N page scores for insertion into the result list bounded by

$$2M \cdot T_{\text{clk}} \leq t_{\text{RMU1}} \leq 2N \cdot T_{\text{clk}}. \tag{6.5}$$

The time t_{RMU2} of the RMU2 includes the bubbling procedure of the inserted records in the result list in $O(R)$ time complexity, with R the length of the result list. Functionally however, this supplementary delay can be considered as a final latency since the list sorts itself automatically and does not make the ACE wait, even when the currently processed

data needs to bubble up to the top of the list. Hence, neglecting t_{RMU2}, t_{RMU1} is upper-bounded through N when all of the N records are validated candidates and must be inserted in the list, and lower-bounded through M such that

$$t^*_{\mathrm{RMU}} = \lim_{t\to\infty} t_{\mathrm{RMU1}} = 2M \cdot T_{\mathrm{clk}} \tag{6.6}$$

when all the values at the output of the M sorters of the RSU2 gets rejected due to a too low score. We remember here that the coefficient 2 comes from the select-and-compare way of working of the RMU.

1.2.2 Exploitation of the Timing Diagrams

At first for a full task parallelism within the ACE, we have to consider that none of the four units makes its predecessor wait, that is to say that the processing shall never be interrupted, especially at the BAM interface. To this aim, we rely on an "as soon as possible" synchronization strategy that allows data to be handled immediately as the processing resource becomes available. Compared to a brute-force synchronization where each task is started concurrently at the frequency of the slowest one, our strategy permits an automatic adaptation of the duration of the tasks and a flexible manipulation of the possible overhead in case of the RMU1.

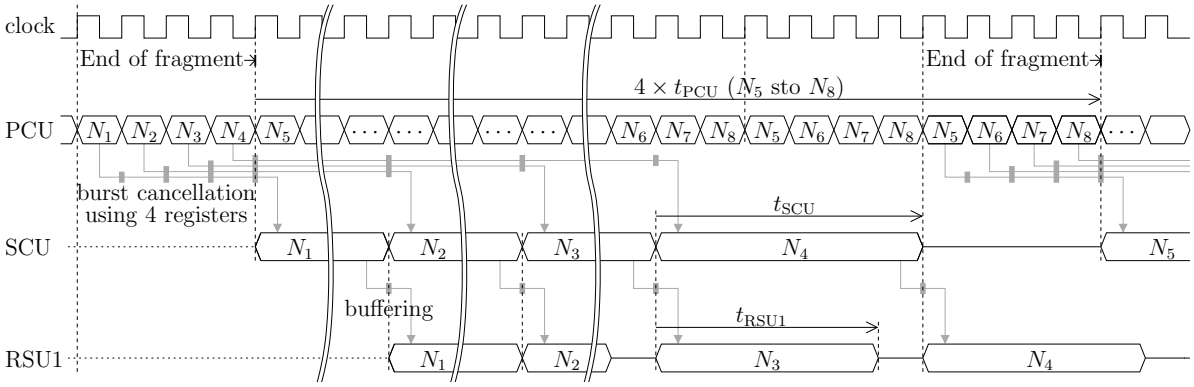


Fig. 6.4. During the processing of one query fragment, N bits data vectors are read from the BAM. The four registers between PCU and SCU as seen in Fig. 6.3 allow the processing of the data in a demultiplexed way at the end of each fragment.

Based on a single clock frequency at which N-bits vectors are read from the BAM using a burst access of length four, Fig. 6.4 represents the synchronization process between the PCU, the SCU and the RSU. Between the end of two fragments, $4N$ pages are processed by the PCU, i.e., N_5 to N_8, in an N pages multiplexed way as explained in Sec. 2.1.1 of Chapter V. While the penalties of four fragments are calculated in $4 \times t_{\mathrm{PCU}}$, the scores of the $4N$ preceding pages, i.e., N_1 to N_4, can be calculated in $4 \times t_{\mathrm{SCU}}$ or less. Therefore, every fragment penalty at the output of the PCU passes through four buffers in order to cancel the burst effect due to the read access mode of the BAM interface. In the same time duration, although delayed of t_{SCU}, four sets of N values must be ordered in the RSU.

Based on the timings of the SCU, Fig. 6.5 shows the evolution of the data sets N_1 to N_4 along the sorting subsystem over the two levels of the RSU before entering the RMU. As seen in Chapter V, the execution time of the bitonic sorters composing the RSU1 and RSU2 modules is deterministic by construction and set on purpose shorter than t_{SCU}. However, t_{RMU1}, which corresponds to the time the RMU needs to scan the potential candidates from the output of the RSU, is functionally non-deterministic, i.e., short if the scores of the pages are low and quite long if these are rather high. The two cases are depicted in Fig. 6.5, whereas the latter eventually causes the preceding units to hold the execution for a few clock periods until resources are available again for processing the next set of data. In the worst case, the PCU might theoretically happen to be waiting, e.g., at the very beginning of the processing of a query, when the result list is empty and when the RMU is accepting any result from the RSU.

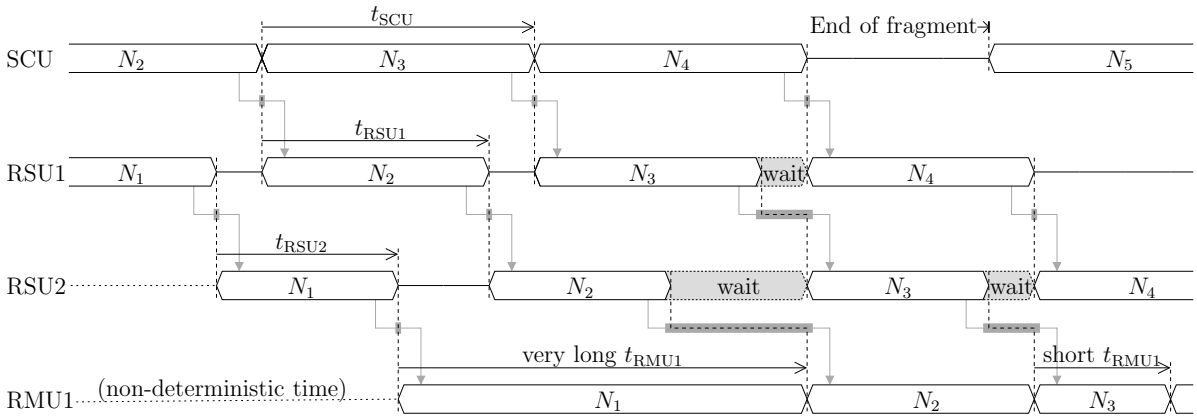


Fig. 6.5. A very long processing time t_{RMU1}, e.g., for N_1, can force the preceding units to wait, whereas a short t_{RMU1}, e.g., for N_3, makes the RMU wait for data.

Different approaches are foreseeable for reducing the probability of blocking the processing in the PCU while the RMU is inserting eventual candidates in the result list. One possibility would be to insert a large FIFO buffer between RSU2 and RMU1, i.e., a temporary storage between parallel processing and sequential sorting, to give the RMU more time to process. The size of the FIFO is application-dependent and must be estimated using functional simulations of the system in the considered boundary conditions. However, a register chain at this location in the data path would yield a large data width and its implementation might not be justified in terms of hardware costs, as we try to keep the size of the architecture as small as possible. Another possibility would be to reconsider the minimum bounding time t_{SCU} to be longer, allowing not only a redimensioning of the RSU into slower and hence smaller partial bitonic sorters, but also making the insertion of more eventual candidates in the result list by the RMU feasible.

1.3 Synthesis Results

The scalable architecture of the ACE must be dimensioned in order to fit into the target FPGA, not only in size as the hardware resources are not infinite, but also in terms of frequency as we want to require only one single clock domain in the design of the data path. On the one hand, enlarging the size of the design tends to maximize the data parallelism and hence the throughput. On the other hand, it increases the switching activity as well as the routing complexity and hence badly impacts the maximum clock frequency, especially in the sorting part of the design.

1.3.1 Numerical Analysis

The condition that rules the most efficient implementation in terms of computational performance was described by (4.3) without taking the consequences of the burst accesses to the memory subsystem into consideration. Satisfying these constraints with (6.1) and (6.2) implies that the PCU must be waiting if $n_p > s$. The average length[1] of the words included in the standard dictionary of a Linux spell checker yields 9.7 characters per existing word. For this reason, we set the lower bound of the processing time through the SCU to $t_{\mathrm{SCU}} \leq 10 \cdot T_{\mathrm{clk}}$ so that $n_p \leq s_{\min} = 10$.

With $N = 32$ bits for the data bus on our FPGA platform as seen in Fig. 6.1, we can estimate the parameters used within the scalable data path necessary for the correct processing of the BAM pages, especially for the sorting subsystem. Arbitrarily, the length L of each record processed within the RSU includes the $n_s = 14$ bits score of a page, the $n_F = 5$ bits FHT and the relative $\log_2 N$ bits page number so that $L = 24$ bits. According to (4.3), the problem is now to dimension the RSU in order to ensure that $t_{\mathrm{RSU2}} \leq t_{\mathrm{SCU}}$ and $t_{\mathrm{RSU1}} \leq t_{\mathrm{SCU}}$, whereas the unknown variables are the amount M of partial bitonic sorters in the blocks RSU1 and RSU2, as well as their respective internal processing width p_1 and p_2. Hence using (6.3), (6.4) and (6.5), we can solve numerically the values of p_1, p_2 and M.

M	$\frac{N}{M}$	$\frac{N}{2M}$	$t_{\mathrm{RSU1}} \cdot f_{\mathrm{clk}}$	$t_{\mathrm{RSU2}} \cdot f_{\mathrm{clk}}$	$p_{1\min} \rightarrow p_1$	$p_{2\min} \rightarrow p_2$	$t^*_{\mathrm{RMU}} \cdot f_{\mathrm{clk}}$
1	32	16	$312/p_1$	$120/p_2$	~~32~~	12→ ?	2
2	16	8	$168/p_1$	$96/p_2$	17→ 24	10→ 12	4
4	8	4	$72/p_1$	$72/p_2$	8 →8	8 →8	8
8	4	2	$24/p_1$	$48/p_2$	3→ 3	5→ 6	~~16~~

Table 6.1. Dimensioning the Result Sorting Unit for the FPGA platform with $N = 32$ records of $L = 24$ bits to be sorted in parallel using two sorting levels. These numbers are functional and must be verified after the hardware synthesis.

In the RSU, the remapping of the p_1 bits of the first sorting level to the p_2 bits of the second merging level is possible for any width using arbitrarily complex state machines. However, this becomes very simple when $p_1 = k \cdot p_2$ with k or $k^{-1} \in \mathbb{N}$, and absolutely

[1] obtained with the shell command "`wc /usr/share/dict/words | awk '{print $3/$2}'`" by dividing the total number of characters by the number of words

trivial if $k = 1$. Reported in Table 6.1, the lower bounding values of p_1 and p_2 vary along different parameters such as the expected duration t^*_{RMU} for the merging in the result list according to (6.6). Using $M = 1$ is not an option since p_1 cannot be larger than L, as well as for $M = 8$ where $t^*_{\mathrm{RMU}} > t_{\mathrm{SCU}}$. In the best case, $L = k_1 \cdot p_1 = k_2 \cdot p_2$, which implies for $M = 2$ to take $p_1 = 2 \cdot p_2 = L$ for practical reasons. However, when $p_1 = 24$, the area of the sorter in the RSU1 is considerably large and the 100 MHz frequency requirement cannot be achieved. Therefore, as the last but not least solution, with $M = 4$, all the constraints are respected, and plus, the hardware realization of the mapping between RSU1 and RSU2 remains simple with $p_1 = p_2 = \frac{L}{3}$. Hence, Table 6.2 recapitulates the different bit widths used in the design.

Param.	Bits	Description
N	32	Width of the data bus on board, i.e., length of the BAM vector.
$s_{\min}$	10	Amount of slots in the query for minimal time processing.
n_p	8	Width of $\mathcal{P}_p(f)$ in PCU and SCU, i.e., fragment penalty.
n_s	14	Width of $\mathcal{S}_p$ in the SCU, i.e., page score.
L	24	Length of a record in the RSU, i.e., page number, score, and FHT.
p_1	8	Width of the $\frac{N}{2M}$ bitonic sorter in the RSU1.
p_2	8	Width of the $\frac{N}{M}$ bitonic merger in the RSU2.
L_r	32	Length of a record in the RMU, i.e., page address, score, and FHT.

Table 6.2. Width of the various signals composing the data path of the ACE within its FPGA based hardware implementation on the Raptor 2000 prototyping platform.

The final length L_r of the records at the output of the ACE was set to 32 bits not only for commodity reasons, but also because it fits the address of a 128 MB BAM composed of 512 thousand pages, a scaled version of the page score on eight bits, as well as a complete five bits FHT. The position of the pages in the result list does not need to be transferred as a value, since the list is read, i.e., emptied, best score first until no item is left inside.

1.3.2 FPGA Implementation

Table 6.3 gives a detailed list of the hardware resources needed in the FPGA implementation of the ACE in terms of slices, LUTs and maximum frequency, for the processing blocks only. The communication blocks and the internal system bus are quite small, hence fast, and were not considered in the listing as relevant parts of the ACE. The implementation results do not only show that the architecture has a very small area but also that the system is suitable for high speed memory devices such as the 100 MHz SDRAM chips. The amount of on-chip SRAM memory is not reported in the table and accounts for 2 KB, due to the instruction memory where the translated query string is stored. Remembering that it is freely extendable, the length of the result list was arbitrarily set to $R = 100$ places in our implementation. Moreover, the global area is in addition minimally influenced by the rest of the logic in the system such as the local bus interface to the PCI bridge and the communication FIFO for the frequency

XCV2000E	BIF	PCU	SCU	RSU	RMU	GCS
Flip-Flop Slices	225	465	698	1850	2339	71
4 input LUT	365	680	1284	3164	4171	63
Clock load	359	978	1172	2458	3307	93
FPGA Usage	1%	2%	3%	9%	12%	0%
Frequency (MHz)	116.3	129.3	175.2	125.3	128.4	140.3

Table 6.3. Listing of the hardware resources needed for the FPGA implementation of the ACE after individual synthesis of the different modules on a Xilinx Virtex 2000E.

adaptation, as well as the placement and routing overhead. Regarding the prototyping board presented in the previous section, it is safe to say that our goal has been reached in that the final synthesis of the ACE which includes all the functional blocks of Fig. 6.2 yields a frequency of 108 MHz.

2 Evaluation of the Hardware Model

2.1 Benchmarking Environment

In the context of hardware and software, formal verification is the act of proving or disproving the correctness of a system with respect to given specifications or properties. Although the nature of our model is known by construction, we intend in this section to demonstrate the correct functionality of the system we have built.

2.1.1 Validation of the hardware model

It seems easy to provide a system like the ACE that apparently performs the correct operations, but we have to make sure that it fulfills its purpose regarding real text queries in huge text databases. Therefore we tried the implementation of a coding method for a personal verification of the results, permitting a validation of the hardware platform. For this we use a method basically issued from the hash theory described in Sec. 2.4 of Chapter II. In order to practically limit the computational complexity, we have chosen an alphabet of valid characters, ciphers and signs composed of 40 items. With these considerations, (2.4) on page 24 becomes

$$h(x) = \big(x[0] \cdot 40^2 + x[1] \cdot 40 + x[2]\big) \mod 2039. \tag{6.7}$$

Relying on the AAM as explained in Chapter III, we use this function to generate the text signatures and also encode the query string. Furthermore, the ASCII filtering process during the creation of the BAM relies on regular expressions, as explained in Sec. 1.2 in Chapter III. After removing special signs, it maps any character of a text onto the corresponding item in our selected alphabet.

The encoding phase of the query described in Sec. 3.2.2 of Chapter III includes the assignment of weights and the fragmentation of long text strings. The value of a weight which is given for a word in the query string depends on the length of the words, since long words are supposed to be more important, and on the language used, e.g., the word "`the`" which appears too often in English might be automatically removed from a query. Hence, the resulting weighting ω_s of a slot s is described through

$$\omega_s = \begin{cases} 0 & \text{if word length } < 3 \text{ or word too frequent} \\ 1 & \text{if } 3 \leq \text{ word length } < 5 \\ 2 & \text{if } 5 \leq \text{word length} < 9 \\ 3 & \text{otherwise} \end{cases} \tag{6.8}$$

so that only two bits are needed for the coding, plus one bit to specify the sign of the weight, assigning it negative for words that shall not appear in the retrieved pages, as seen in Fig. 5.7 on page 74.

In the case of long queries which need to be split into fragments in order to increase the precision of the retrieval, a condition is necessary to decide when to set the fragment separation slot. A trivial approach would be to count the number of slots and to terminate the fragment at word boundaries. However, this method is too simple and does not take into account the eventual relevance of important words in a searched fragment. Therefore, we implemented a different method that counts the absolute weights instead. The condition for the slots s_i to s_j to belong to the same fragment F is that

$$\sum_{s=i}^{j} \omega_s \leq G_{max} \quad \forall s \in F. \tag{6.9}$$

In other words, we make sure that the sum of the weights ω_s in a given fragment F doesn't exceed the rough maximum granularity G_{max}, which we happen to set to 100 in our test implementation. However, as it only serves for testing purposes, it must be clear that this distribution of the weights has been arbitrarily chosen and cannot compete in any way with professional implementations which account idiomatic properties.

2.1.2 Functional Verification

With reference to (3.1) in Chapter III, it has been reported that a signature file is optimized when its entropy is maximized [Wit99]. As we know that our hash function in (6.7) is experimentally created without any working guarantee, we can measure the distribution of the bit attributes in the BAM and appreciate the quality of the coding using a frequency analysis. The magnitude of the two dimension Fast Fourier Transform (FFT) of the BAM in spatial frequency domain is shown in Fig. 6.6 where

$$\mathcal{B}(f_x, f_y) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} B(x,y) \cdot e^{-j2\pi\left(\frac{x \cdot f_x}{M} + \frac{y \cdot f_y}{N}\right)} \tag{6.10}$$

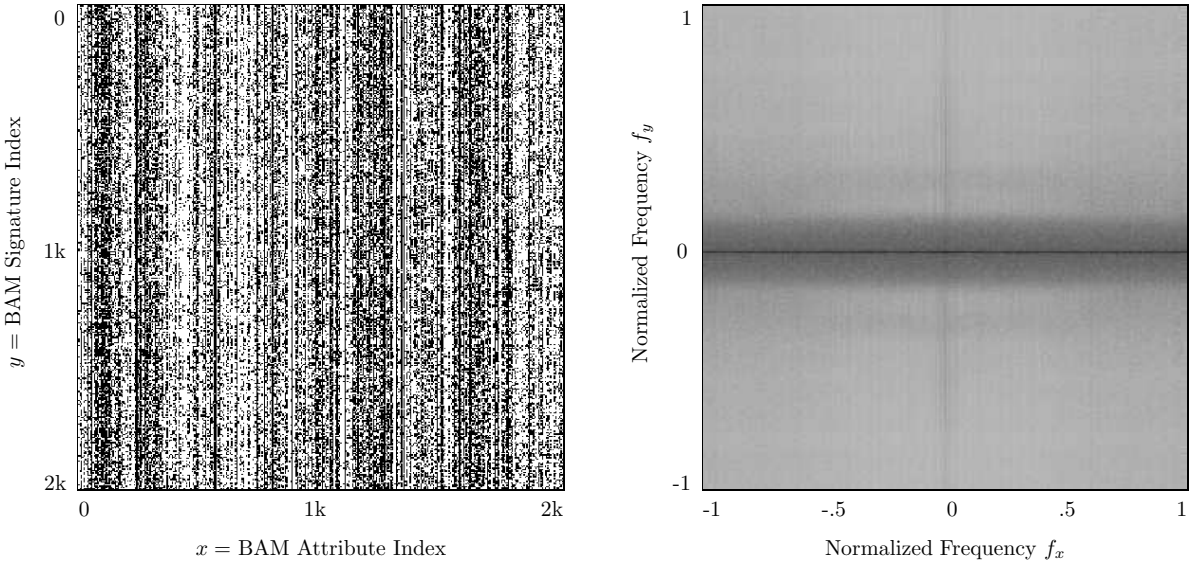


Fig. 6.6. The Bit Attribute Matrix represented as a bitmap (left) yields the spatial frequency spectrum (right) after a two dimensional Fourier transform.

with $B(x, y)$ and $\mathcal{B}(f_x, f_y)$ standing for the BAM and its image respectively. In order to perform the FFT instead of the much slower Discrete Fourier Transform (DFT), the image has been transformed so that the width and height are an integer power of two. The axis carry normalized frequencies, for which the value 1 represents the maximum frequency inside the BAM, i.e., the alternation of zeroes and ones, or respectively black and white dots on the left of Fig. 6.6. On the one hand, we can say that the distribution of the bits is apparently quite uniform. On the other hand however, the horizontal white line in the spectrum reveals a vertical dependency in the real BAM. This can be explained by the fact that the chosen texts, for creating the BAM, came from an English repository where the same finite amount of most frequent trigrams appear repetitively. Another reason could also be that our hash function is too rudimentary compared to a specialized one which would take into account linguistic properties of the texts from the database.

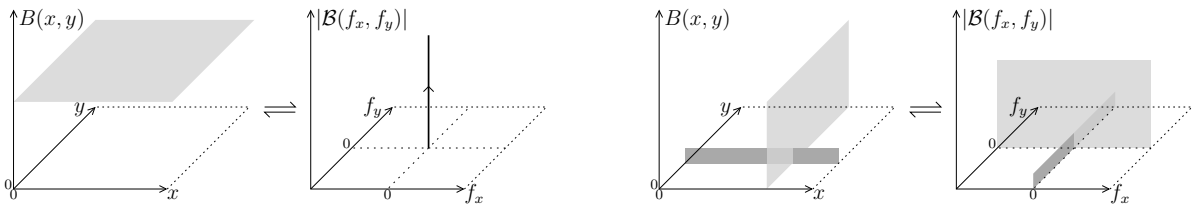


Fig. 6.7. In a two dimensional Fourier analysis, a 2D plane (left) transforms to a delta function and, neglecting the phase information, a horizontal ($y = 0$) or vertical ($x = 0$) line of delta functions (right) to a line of delta functions $f_x = 0 \,\forall f_y$ or $f_y = 0 \,\forall f_x$ respectively.

As seen in Fig. 6.7, instead of the expected uniform distribution of the bits in both x and y direction in the BAM (left), a normalized frequency analysis (right) reports the rather constant presence of some bit-attributes, i.e., vertically in $B(x, y)$, through the amplitude of $\mathcal{B}(f_x, f_y)$ for $f_y = 0$. Less obvious, the fact that some page signatures

in the BAM are empty or composed of less trigrams in case of a small page makes horizontal lines visible in Fig. 6.6 (left), and hence causing the light grey vertical spectral ray at $f_x = 0$. To all intents and purposes, the main idea is here to prove that our model of coding works properly with the ACE and that our implementation is not only theoretically correct by construction, but also that a prototype permits real performance measurements beyond the simulation level.

2.2 Scaling the Design

Within our work, we took special care in designing a flexible parameterizable system that does not only provide a solution for different hardware platforms but also allows its scalability for higher performance requirements as well as wider or narrower bus sizes. Compared to Sec. 1.3, a synthesis of the VHDL RTL code followed by a place and route implementation step for the next class of FPGA, i.e., the Xilinx Virtex2 family devices [Xil06], yields the same area requirements for almost twice the clock speed, reaching the symbolic 200 MHz internal frequency. In such a case, the memory subsystem must be scaled up as well, therefore we would use (Double Data Rate) DDR SDRAM devices that provide the double data throughput.

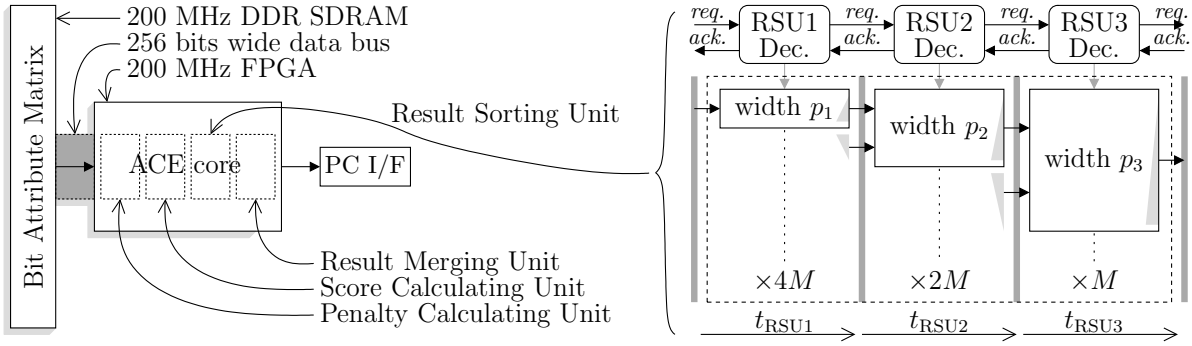


Fig. 6.8. Detail of a three level implementation of the RSU in a simulated version of the ACE with a 200 MHz Double Data Rate (DDR) connection to the BAM through a 256 bit data bus that yields internally $N = 512$ bits.

Fig. 6.8 represents a possible integration of the data path of the ACE on a simulated hardware platform composed of a Xilinx FPGA, e.g., Virtex4 or Virtex5 that can be easily clocked at 200 MHz, and a large memory subsystem, the size of which does not play any role in the performance measurement. The key element of the design is the large data bus between the processing units and the BAM where 256 bits can be transferred at each edge of the clock typically specified at 200 MHz, reaching a record breaking throughput of 12.8 GB/s. As a direct consequence of the bus widening and of the DDR access, increasing the internal degree of parallelism $N = 2 \times 256$ in the processing units PCU and SCU requires the following RSU to be able to sort a larger amount of page scores in a time still set by a theoretical minimum query length, e.g., $s = 16$. Then we have to redimension the sorting subsystem and extend it with at least one supplementary

M	$t_{\mathrm{RSU1}} \cdot f_{\mathrm{clk}}$	$t_{\mathrm{RSU2}} \cdot f_{\mathrm{clk}}$	$t_{\mathrm{RSU3}} \cdot f_{\mathrm{clk}}$	$p_{1\min} \rightarrow p_1$	$p_{2\min} \rightarrow p_2$	$p_{3\min} \rightarrow p_3$	$t^*_{\mathrm{RMU}} \cdot f_{\mathrm{clk}}$
1	$1032/p_1$	$192/p_2$	$216/p_3$	~~65~~	$12 \rightarrow ?$	$14 \rightarrow ?$	2
2	$744/p_1$	$168/p_2$	$192/p_3$	~~47~~	$11 \rightarrow ?$	$12 \rightarrow ?$	4
4	$504/p_1$	$144/p_2$	$168/p_3$	~~32~~	$9 \rightarrow ?$	$11 \rightarrow ?$	8
8	$312/p_1$	$120/p_2$	$144/p_3$	$20 \rightarrow$ **24**	$8 \rightarrow$ **12**	$9 \rightarrow$ **12**	**16**
16	$168/p_1$	$96/p_2$	$120/p_3$	$11 \rightarrow 12$	$6 \rightarrow 12$	$8 \rightarrow 12$	~~32~~
32	$144/p_1$	$72/p_2$	$96/p_3$	$5 \rightarrow 6$	$5 \rightarrow 6$	$6 \rightarrow 6$	~~64~~
64	$120/p_1$	$48/p_2$	$72/p_3$	$2 \rightarrow 3$	$3 \rightarrow 3$	$5 \rightarrow 6$	~~128~~

Table 6.4. Dimensioning the RSU for a simulated FPGA platform with $N = 512$ records of $L = 24$ bits to be sorted in parallel using three sorting levels ensuring $t^*_{\mathrm{RMU}} \leq 16 \cdot T_{\mathrm{clk}}$. In the table, $p_{\min} \rightarrow p$ indicates the minimum size of p in order to respect the required throughput and then the chosen size of p which makes sense to be implemented considering $p \leq L$.

merging step, i.e., the RSU3, hence taking into account the increasing of the total area of the architecture. In the RSU, the internal degree of parallelism is upper-bounded by L and the subsequent merging time t^*_{RMU} constrained by M. According to Tab. 6.4, the best choice ensuring $t^*_{\mathrm{RMU}} \leq 16 \cdot T_{\mathrm{clk}}$ for a customized FPGA platform with $N = 512$ records of $L = 24$ bits to be sorted in parallel using three sorting levels remains the combination $M = 8$ with $p_1 = 2 \cdot p_2 = 2 \cdot p_3 = 24$.

Resorting to the same methodology while trading speed for area, it can be shown that if we allow $t^*_{\mathrm{RMU}} \cdot f_{\mathrm{clk}} \leq 32$, i.e., in the case of a search engine for queries of at least 32 slots, then the whole architecture of the RSU can be drastically reduced using $M = 16$ with $p_1 = p_2 = p_3 = L/4$.

2.3 Performance Appraisal

Speedup is a measure of hardware performance which can be defined as the running time of a sequential algorithm available on a given machine over the running time of the parallel algorithm executed by a dedicated accelerator such as the ACE. In order to evaluate this measure, we rely on the software implementation of the Associative Access Filter (AAF) algorithm presented in Sec. 2 of Chapter IV which is brought to run on different personal computers. Tab. 6.5 lists the most relevant properties of these PCs in terms of Central Processing Unit (CPU) speed, main memory (RAM), and Hard Disk Drives (HDD). Typically, this reflects the evolution of the most significant computer hardware parts within the last five years and might help conclude on future trends.

On the hardware side of the benchmarking, the ACE has been synthesized for two FPGA devices with different architecture sizes. One has a 32 bit data bus to a 100 MHz SDR Single Data Rate (SDR) SDRAM memory subsystem and the other a 256 bit data bus to a 200 MHz Double Data Rate (DDR) SDRAM memory subsystem. The first implementation has been tested and evaluated on the prototyping board described in Sec. 1.1, whereas the second model was only simulated and synthesized for a virtual hardware model of the board components.

Configuration	Software 1 (PC1)	Software 2 (PC2)	Software 3 (PC3)
Processor Type	Intel PIII	AMD Athlon	Intel PIV
CPU Frequency	933 MHz	1875 MHz	2926 MHz
L2 Cache	512 KB	512 KB	1 MB
Release Year	2001	2003	2005
Memory Type	SDR SDRAM	DDR SDRAM	DDR SDRAM
RAM Frequency	133 MHz	166 MHz	200 MHz
RAM Size	256 MB	512 MB	1024 MB
HDD Size / Cache	80 GB / 8 MB	80 GB / 2 MB	120 GB / 8 MB
HDD Bus Type	ATA 100	ATA 100	SATA 150
HDD Access Time	12 ms	9 ms	8 ms
HDD Properties	5400 rpm - 2.5”	7200 rpm - 3.5”	7200 rpm - 3.5”

Table 6.5. Listing of the most relevant parameters of the different software environments, reporting the processor data, main memory parameters and the features of the HDD.

Fig. 6.9 reports the execution time of the AAF algorithm on different platforms including the three PCs described in Tab. 6.5 storing the BAM either on their Hard Disk Drive (HDD) or into their RAM, as well as the two hardware implementations. As expected, the query time responds linearly to the two varying parameters, i.e., the size of the BAM (left) and the length of the query (right). Unfortunately, for small values of the query length and the BAM size, the profiling tools have to measure extremely short durations, even though the BAM was set constant to 128 MB and the query to 128 items, i.e., rather large in the appropriate measurement scenario. In such a case, the influence of a multi-tasking Operating System (OS) becomes noticeable and tends to falsify the results. Moreover, the HDD are mechanical devices which require a certain amount of time to position their heads on the correct tracks and sectors. As data is usually scattered randomly all over the disks, repositioning the heads at a new physical address costs supplementary latency penalties. When a file is accessed on the hard disk, a cache system permits to accelerate the data transfers to or from the main memory and the processor of the computer. In this sense, when the BAM is read from the drive, it is partially copied into the cache, hoping for a locality advantage. However, the vertical accesses to the columns of the matrix, as described in Sec. 1.2 of Chapter V, forces the BAM file to be read colum by column so that the main memory of the computer serves as a cache for a few successive columns. When the query gets longer, as seen in Fig. 6.9 right, the probability becomes higher that a column indexed by the query is already stored in the memory due to an earlier neighboring access. In such a case, the data is fetched from the memory instead of the HDD. These reasons explain why the behavior of the curves based on the HDD measurements is sublinear compared to the variations of the length of the query.

When comparing Tab. 6.5 with Fig. 6.9, it becomes clear where the real speed-up of the algorithm really comes from. Apparently, the speed of the processor of a PC only plays a minor role in the duration of a query. It appears that using a CPU which is three times faster, i.e., PC3 versus PC1, only reduces the searching time to half. Advances in

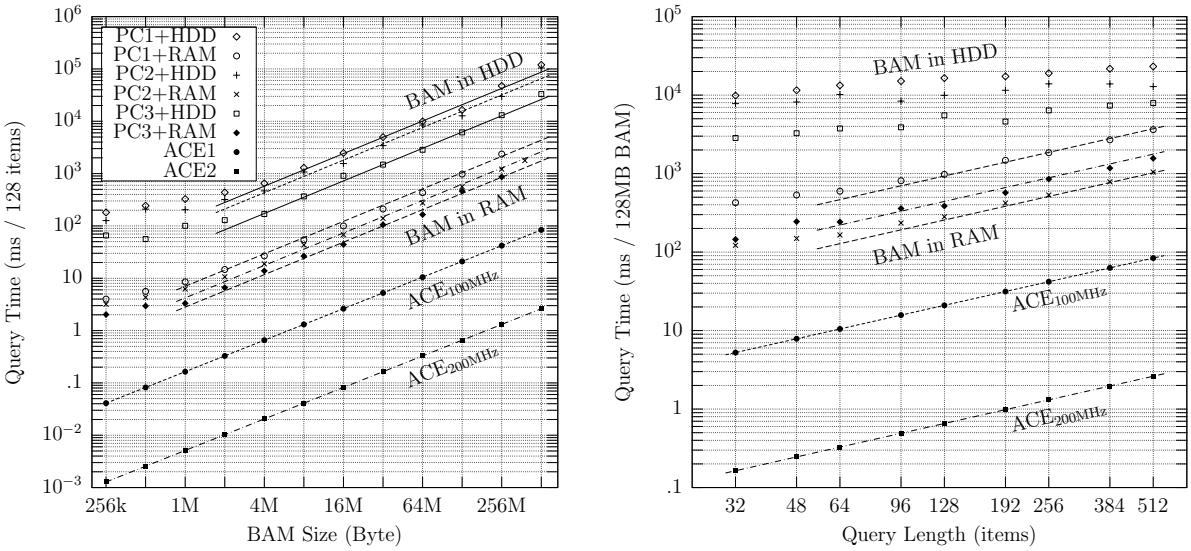


Fig. 6.9. Representation of the complexity of the AAF algorithm executed by the two ACE models and three different computer architectures against the size of the BAM for a query of 128 slots (left) and against the length of the query for a BAM of 128 MB (right). When the BAM is read from the HDD, during long queries (right), local information tends to be re-accessed from the main memory of the computer that serves as a cache instead of being accessed from the disk.

the HDD technologies within the last five years permit a reduction of a query duration with respect to the AAF algorithm down to one fifth of the original time. Moreover, when the BAM is stored in the main memory of the system, i.e., in RAM instead of on HDD, a query can be accelerated by a factor of about 20.

The technological evolution of memory devices in terms of access frequencies and data rates, as known from Fig. 1.3 in Chapter I on page 4, makes the biggest difference in our benchmarking environment. Not only can we explain the gain in time between the different software solutions, but also between the two hardware models. This proves that a fast memory with a fast processor is not enough to obtain a very high-performance system. As presented in this thesis, it is their clever association which makes an old and

Appliance	Processing Unit	BAM Location	Speed-up
Software 1	} PIII 933 MHz {	ATA 100 Hard Disk	~ 25000
Software 1'		SDR SDRAM 133 MHz	~ 1500
Software 2	} Athlon 1875 MHz {	ATA 100 Hard Disk	~ 18000
Software 2'		DDR SDRAM 166 MHz	~ 800
Software 3	} PIV 2926 MHz {	SATA 150 Hard Disk	~ 5000
Software 3'		DDR SDRAM 200 MHz	~ 700
Hardware 1	VirtexE 100 MHz	SDR SDRAM 100 MHz	32
Hardware 2	Virtex4 200 MHz	DDR SDRAM 200 MHz	*Reference*

Table 6.6. Summarization of the average speed-up obtained by the simulated ACE architecture (Hardware 2) against the different test platforms for queries composed of a variable number of items using a BAM ranging from 256KB up to 256MB.

slow 100 MHz FPGA with a simple standard SDR SDRAM (Hardware 1) be more than twenty times faster that one of the fastest currently available PCs.

In conclusion, Tab. 6.6 reports the average speed-up obtained by the simulated ACE architecture (Hardware 2) against the measured test platforms ranging from the different software solutions where the BAM is stored in RAM or on HDD, to the prototyping platform using standard almost obsolete memory devices.

Chapter VII

Conclusion and Outlook

BASED on an associative computing paradigm, our research has shown how intensive data processing algorithms can be efficiently mapped onto elementary hardware structures under drastic speed and area constraints. The two most important issues were first the overall scalability of the design allowing the support of different bit widths and module depths along the data path, and second the obtainment of the minimum hardware area while ensuring the maximum throughput of the global architecture. Hence, in this chapter, we give a brief recapitulation of the ideas developed in this thesis which follow the development of the Associative Computing Engine (ACE). Moreover, we present in a further section the possible improvements to our work and some open applications for additional research fields.

1 On the Associative Computing Engine

Beside the matching quality, one of the most important problems in information retrieval systems remains their performance in terms of searching time. With the exponentially increasing amount of data available on different multimedia networks, finding information has become a real challenge for both software and hardware solutions. As today's General Purpose Processor (GPP) based implementations cannot achieve the required performance, it is necessary to port the problem onto dedicated hardware systems. In this thesis, we addressed particularly the memory wall issue through the realization of a hardware search engine based on approximate matching computations.

First of all, in the development of the ACE, the basic task preceding any hardware implementation was the choice of an algorithm suitable for many data types which also allows the processing of long query strings in huge databases. A dedicated profiling has shown that Lapir's Associative Access Method (AAM) is very promising if we can extract the hidden degrees of parallelism and start on a low-level analysis of the algorithm. Not only was a functional task parallelization of the algorithm possible, but also

a parallelization of the data and local operations in order to increase the theoretical throughput of the system. However, drastic constraints had to be observed during the design of the architecture. On the one hand, the final goal is to obtain the smallest area possible in order to reduce the switching activity and the power consumption. On the other hand, we target the maximum clock frequency for the synchronous data path to ensure the highest possible computational throughput.

In parallel, we performed the exploration of different hardware domains for the storage of the text database index, i.e., the Bit Attribute Matrix (BAM), with the aim to dispose of a fast and non-expensive memory compatible with the architectural conception of the system. The decision of supporting the dense but complex dynamic memory devices versus the simple but more expensive static ones had heavy consequences on the operations and the synchronization of the different tasks along the data-path. Not only did we have to implement a special SDRAM controller for being able to access the data fast enough, but also bring the corresponding modifications to the computational units to handle the flow of data correctly, e.g., demultiplexing the burst accesses.

Hence, in our modular architecture, the data-path includes two major parts which essentially differ in the way they handle data. As developed in Chapter IV and V, the first part composed of the Penalty Calculating Unit (PCU) and the Score Calculating Unit (SCU) is fully parallel. This basically deals with the implementation of fast operations, e.g., penalty accumulation, as well as more complex mathematical operations. In this sense, nonlinear functions such as the logarithm must be computed using logic hardware structures. The second part of the data-path composed of the Result Sorting Unit (RSU) and the Result Merging Unit (RMU) is dedicated to a special problem for which not only the throughput but also the scalability play a major role. Sorting algorithms can be classified into two categories, i.e., the parallel algorithms where the complexity in terms of area depends very strongly on the amount of data to be handled, and the sequential ones, the speed and the complexity of which make them unsuitable for high-throughput systems. In studying sorting networks and different implementation methods that allow variations of the bit widths and of the number of records to sort, we have refined an advanced architecture that permits the adaptability of a sorting subsystem to any kind of hardware platform, following a trade-off between minimum necessary area and required throughput.

In order to test the ACE functionally and measure the duration of the search compared to a standard Personal Computer (PC), we used a prototyping board including a rather old and slow Field Programmable Gate Array (FPGA) and a large amount of external dedicated memory, which was available at our institute. Firstly, we were able to verify practically the correctness of the results as predicted by the simulation. For this we only had to recreate a text encoding model based on the hashing theory described in Chapter II, suitable for the AAM as described in Chapter III. Secondly, we were able to compare in real-time the performance of the board with the same application running in software on a GPP. Hence we have measured up to three orders of magnitude for the duration of a given search between our outdated prototyping board and many recent PCs with different hardware configurations.

Finally, the last part of our work has been to simulate the ACE on a more recent FPGA platform with a newer technology, a more advanced memory type, i.e., Double Data Rate (DDR) instead of Single Data Rate (SDR), and last but not least, a much wider dedicated data bus used for the BAM accesses. Based on hardware synthesis results and post place-and-route simulations, the resized ACE core was able to run synchronously with the memory subsystem at the expected frequencies. Due to the scalability of the system and to the modular implementation methodology, simulation results have demonstrated a huge gain in speed up to more than four orders of magnitude for the acceleration platform based on the ACE against a PC, where a BAM of a respectable size could be stored on the Hard Disk Drive (HDD) only.

2 Future Work

The AAM constitutes the original algorithm which was implemented with the objective of reaching the maximum performance available on any arbitrary hardware platform. Based on signature files, it theoretically permits the retrieval of any object type which can be coded using a hash function, or in other words anything from which binary attributes can be extracted. The main advantage is that the AAM performs an error-tolerant matching allowing an approximate or fuzzy feature coding. Another possible improvement for text searching might be to refine the hash function and the encoding method by using a combination of bigrams and trigrams. Typical examples would be in the domains of the optical character recognition (OCR), biometric minutiae matching such as fingerprints, iris scans or faces, as well as multimedia applications requiring text processing. According to related research work with reference to Chapter III, there is nowadays a growing demand for accelerated computations in genome analysis, biosequence mining and network attack detections among others.

From the hardware point of view, it is safe to say that the Very Large Scale Integration (VLSI) structure presented in this thesis is extremely suitable for an Application Specific Integrated Circuit (ASIC) implementation, regarding both the parallel computing part composed of the PCU and the SCU, as well as the parallel sorting part composed of the RSU and the RMU. Within this thesis, we have shown that the scalability of each module can guarantee the implementation of the system for various bit widths and data throughput. As a consequence, the ACE could become a part of any intelligent system related to database searching applications, either in form of a separate accelerator for a PC, or as an embedded hardware module into a System on Chip (SoC), or as a standard extension for HDD controller chips and memory modules.

Appendix A

On the Hardware Implementation of Logarithms

Presented in Chapter V, the NLF (Negative Logarithmic Function) is a function approximating the binary logarithm of a positive integer in $\mathbb{N}^+$. The method applied here permits the calculation of any form of logarithmic function that can be described by

$$f(x) = A + B \cdot \log_2 (C \cdot x + D) \tag{A.1}$$

theoretically, whereas the choice of A, B, C and D impacts the complexity of the hardware realization as well as the resizing of the function into a given range. However due to the fact that we restricted the processing in the ACE to positive integers, the NLF is a special case of (A.1) that deals with integer numbers in the positive range ($\mathbb{N}^+ \rightarrow \mathbb{N}^+$) including zero as described in (5.3). In our quest of simplicity in the hardware realization, we will try to handle only coefficients that are powers of two, as multiplications and divisions turn into binary left or right shifts respectively. Hence according to [Lay04a], we discuss in this appendix the efficiency of the method and propose some improvements to minimize the approximation error.

1 Error Analysis

Let x and y be some integers with the respective widths i and j, where $x = [x_0, x_1, \ldots x_{i-1}]$ and $y = [y_0, y_1, \ldots y_{j-1}]$. Assuming that x and y are positive, we can represent them using the corresponding binary weighting

$$x = \sum_{n=0}^{i-1} x_n \cdot 2^n \qquad \text{and} \qquad y = \sum_{m=0}^{j-1} y_m \cdot 2^m \tag{A.2}$$

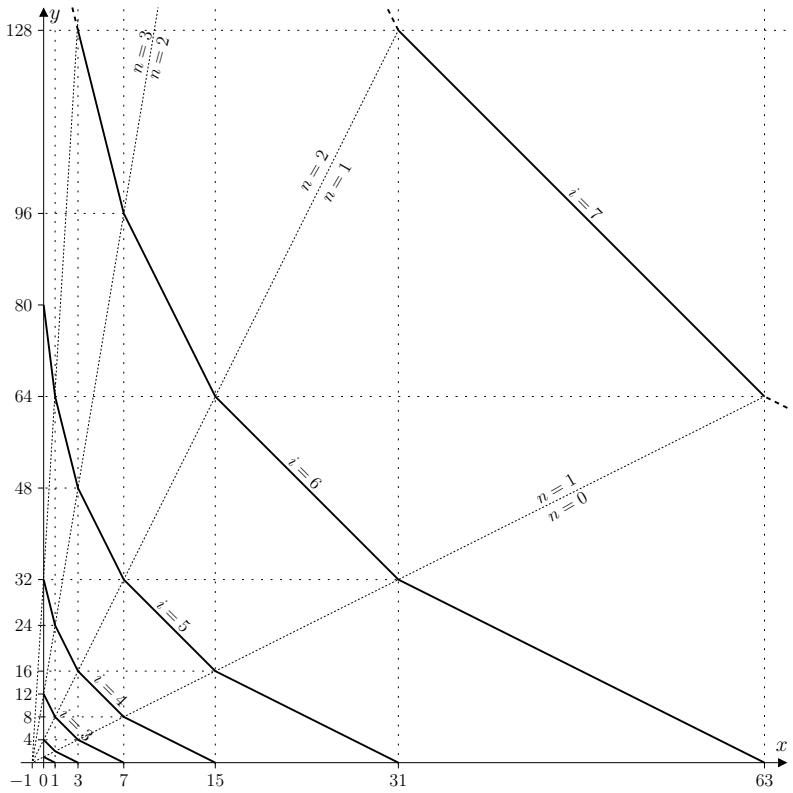


Fig. A.1. Details of the scalability of the NLF for different input widths i that show the y to x dependency according to Algorithm N. In each linear approximation, the parameter i corresponds to its number of fragments.

as in our model. However, in creating a bijective function suitable for the ACE with the most compact range of values on x and y, we obtained the following equation derived from (5.6) and (5.9)

$$y_{n,i}(x) = -2^n \cdot (x+1) + 2^{i-1} \cdot (n+2) \tag{A.3}$$

where i and j are linked together. Hence when i increases, the amount of segments composing the piece linear approximation of the function increases respectively. Fig. A.1 shows partially the characteristics of the NLF realized for different values of i varying from 1 to 7, where each curve is dilated from the previous one with a factor two per added bit in x and extended of one segment. The dilation center is located at ($x = -1$; $y = 0$), and for each value of i, all the segments carrying the same index n, when they exist, are parallel.

We carry now our interest onto the error made in the linear approximation of the logarithm by comparing the NLF and its corresponding analog function. Following the example of Fig. 5.8-a on page 76, the curve $f(x)$ can be recovered by replacing n in (A.3) through a continuous value, i.e., without the rounding ceil function. The only remaining parameter is i, the width of the input x, so that (A.1) becomes

$$f(x) = 2^{i-1} \cdot (i - \log_2(x+1)) \tag{A.4}$$

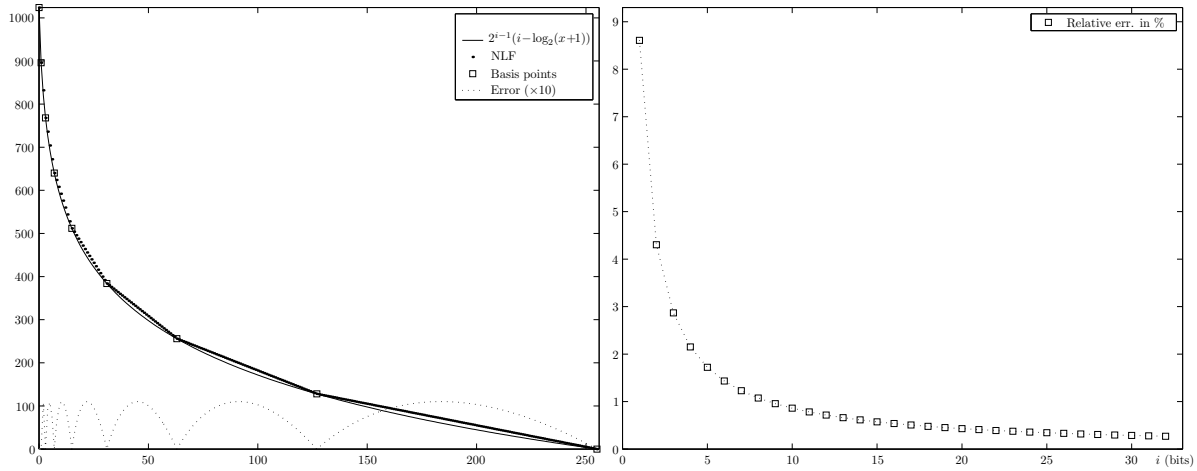


Fig. A.2. a) Comparison of the analog and digitally piece-wise linear approximated logarithm functions and drawing of the absolute error function for an 8 bits input. **b)** Relative error made by the piece-wise linear approximation depending on the width i of the input.

while resolving the coefficients A, B, C and D. Fig. A.2-a shows the analog function $f(x)$ compared to the digital one, using squares to represent the basis points of the NLF. They correspond to the extremities of the segments described in Fig. A.1 and are common to both curves. The absolute error made by this approximation method considering an input width of $i = 8$ bits is magnified with a factor ten and plotted in Fig. A.2-a. Well known by mathematicians [Knu97], this error remains always positive and can be upper-bounded. Since the analog curve is located under the linear approximation, the error can be estimated with the following equation

$$y_{n,i} - f(x) = 2^{i-1} \cdot (\log_2(x+1) + n + 2 - i) - 2^n \cdot (x+1) \tag{A.5}$$

where $y_{n,i}$ corresponds to the straight lines of the NLF. The local maximum between two basis points can be found by differentiating the function in (A.5)

$$\frac{d}{dx}\left[y_{n,i}(x) - f_1(x)\right] = \frac{2^{i-1}}{\ln 2 \cdot (x+1)} - 2^n \tag{A.6}$$

$$\text{maxima:} \qquad x_M = \frac{2^{i-n-1}}{\ln 2} - 1 \tag{A.7}$$

In these equations, x, i and n are linked together as specified in (5.8). The number of maxima equals i and they are almost located in the middle of each interval, whereas the exact value yields $[\ln^{-1} 2 - 1] \approx 0.44$ times the length of the interval. Thus replacing the maxima x_M in (A.5) yields a result independent from n so that

$$err_{\max} = 2^{i-1} \cdot \frac{\ln 2 - \ln\ln 2 - 1}{\ln 2} \approx 2^{i-1} \cdot 0.086 \tag{A.8}$$

explaining the drawing of the absolute error function in figure A.2-a, conform to Mitchell's error analysis [Arn03, Mit62] of the logarithm obtained for i set to 1. Referring to the y

range, the relative error decreases inversely proportional to the width i of the input vector, improving the accuracy as shown in figure A.2-b. The relative error in percentage terms can be estimated to $err_{\text{rel}} \approx 0.086/i$, which is much better than other already presented techniques like in [Hoe91].

2 Error Correction

We propose here a method to reduce this error up to a factor 4.6, with only a few supplementary logic gates and a subtracter. Approximating a logarithm function through straight lines yields an arch shaped error as seen in Fig. A.2-a, which can be itself approximated using simple logic. This error is always positive, upper-boundable and takes effect on the $i-1$ least significant bits of the output. For this reason, the $j-i$ most significant bits pass through and remain unchanged after the correction.

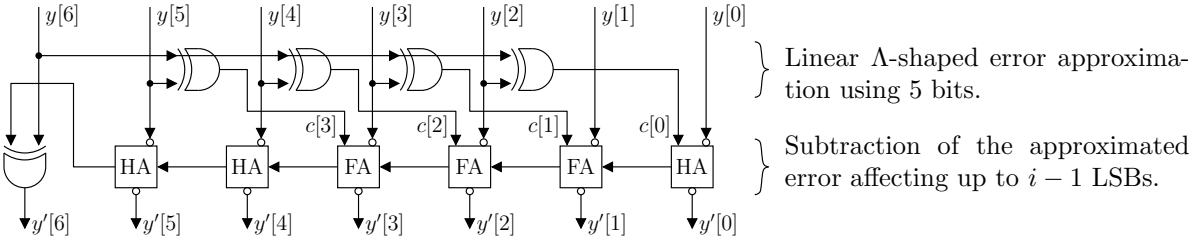


Fig. A.3. Correction of the NLF using a row of XOR gates that creates a Λ-shaped linear approximation of the error using 5 bits.

The idea is to subtract a Λ-shaped approximation of the error within the interval between two basis points. The correction is calculated in such a way that it can not affect the bit whose indices are bigger than i-1, otherwise an unsupported overflow would occur in the subtracter. To generate the correcting value with the fewest logic possible, the XOR gates controlled by the upper bit $y[6]$ create a symmetrical distribution of the four lower bit ($c[3]$ to $c[0]$) which are subtracted from the result after being divided by four through the two bit shifting to the right. As a result, the error made is 2.5 times smaller than without correction as seen in Fig. A.5. The look-up table (LUT) in Fig. A.4 is a complement to the XOR based methodology described in Fig. A.3. It approximates symmetrically the arch shaped error over a four bit word ($d[3]$ to $d[0]$) which is then subtracted to the final result. The logical equations related to the correction signal d

$$d[3] = c[3] \vee c[2] \wedge c[1] \tag{A.9}$$

$$d[2] = \overline{c[3]} \wedge (c[2] \oplus c[1]) \tag{A.10}$$

$$d[1] = c[3] \wedge c[1] \vee c[2] \wedge \overline{c[1]} \tag{A.11}$$

$$d[0] = c[0] \tag{A.12}$$

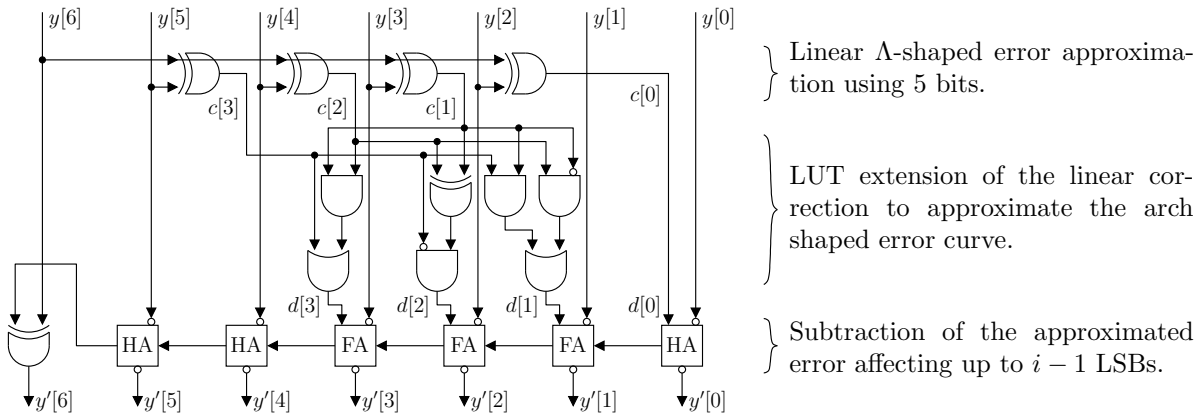


Fig. A.4. Correction of the NLF using a LUT to approximate the arch shaped error curve after a first correction through the linear Λ-shaped error approximation.

are deliberately simple. Compared to the NLF without any correction, the remaining error is here 4.6 times smaller. The cost is the $i-1$ bit long carry path in the full/half-adder chain, and a supplementary clock period to register the data if necessary.

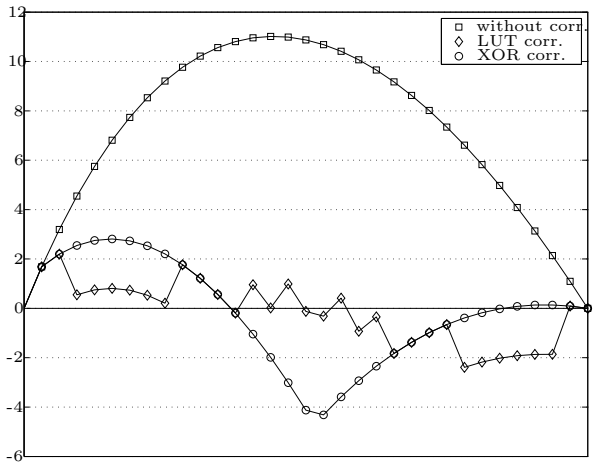


Fig. A.5. Comparison of the absolute error between two basis points for the linear approximation without error correction, for the approximation with a linear correction, and for the approximation with correction using a small look-up table.

In Fig. A.5, the first curve with the squares shows the progression of the absolute error of the NLF between two basis points. The second curve with the circles corresponds to the error after the corrector based on the four XOR gates. The error doesn't stay positive but crosses the zero line, so that there are two more theoretical common points to the analog curve per interval. The third curve with the diamonds represents the error after the second corrector based on the small LUT. The error oscillates around the zero line and crosses it six times in between, so that the approximated curve is closer again to the analog one.

In conclusion to the correction of the integer approximation of the logarithm function, it is possible to use the LUT correction method to reduce the absolute error below one

unit. The result would be a large LUT with almost as many entries as there are bit i in x. For instance, in order to calculate the logarithm of an eight bit number with an absolute error less than unity, we would need a LUT with 6 entries and 4 outputs. It might be easier to implement a ROM in this case, but this is actually what we try to avoid by using the NLF. Finally, further details about the realization of the function are available in [Lay04a].

3 Exponential Function

The architecture in Fig. A.6 realizes the reverse transformation of the NLF, following the example of an 8-bit output value x. With the same properties as described in the previous sections, the corresponding output response yields

$$x = f^{-1}(y) = 2^{\left(i - y \cdot 2^{1-i}\right)} - 1. \tag{A.13}$$

Complementary to the architecture proposed in Fig. 5.10 on page 78, this structure is composed of the same basis elements, namely a loadable register row preceded by simple logic gates. On the right side of the *valid* signal, the first $i-1$ bits of y are shifted from the left to the right, and on the left side of the design, the registers preceded by the half-adders build a decounter. More precisely, the value 1, which is input to the left-most half-adder, increments the four registers preloaded with the negated values $y[7]$ to $y[10]$, i.e., $2^4 - \sum_{k=0}^{3} 2^k \cdot y[7+k] - 1$.

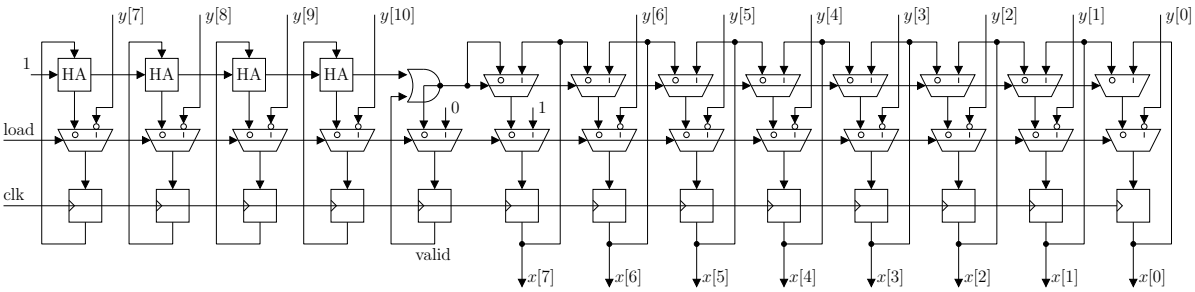


Fig. A.6. Realization of the reverse transformation of the NLF for an 8-bit input with a recursive process based on the sequential implementation of Algorithm N described in Chapter V.

In this structure as well, the valid signal indicates that the computing has been terminated and that the result x can be read from the output registers. In comparison with Fig. 5.10 on page 78, the processing time depends on the value of the input y and the valid signal is not inverted.

Appendix B

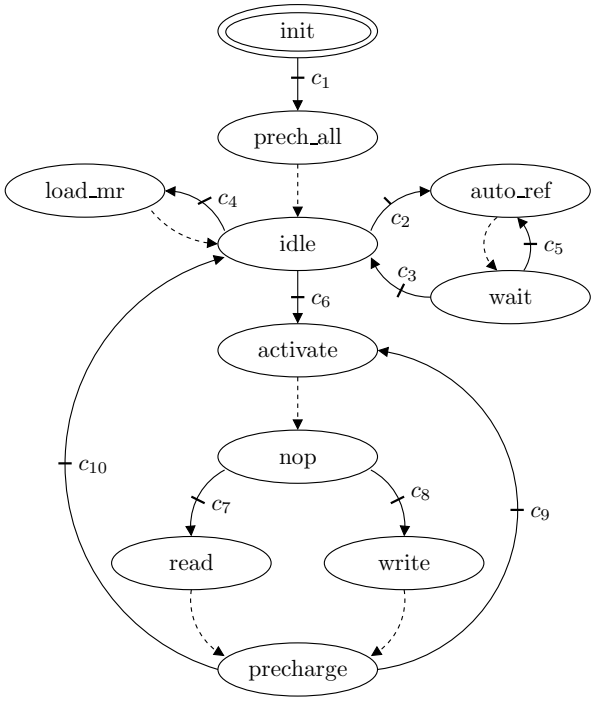
High Throughput Memory Controller

STANDARD memory controllers, available in form of parameterizable Intellectual Property (IP) cores, allow various access types and addressing modes. Although they provide a high design flexibility, they often show a lack of performance for a specific application such as the Associative Access Filter (AAF) algorithm. Therefore in this work, we had to address the problem more in details and develop a custom controller dedicated to a single application that would remain small and efficient though providing a unique access mode. Hence this chapter describes the basics for the realization of a Synchronous Dynamic Random Access Memory (SDRAM) controller suitable for the Associative Computing Engine (ACE), without any unnecessary features. With reference to Chapter IV, the design has been implemented into a Xilinx VirtexE Field Programmable Gate Array (FPGA) and tested in a system running at 100 MHz, thus providing a peek and average bandwidth of 400 MB/s.

1 Finite State Machine

To compete with static RAMs, SDRAM devices offer features to enhance overall bandwidth performance, such as multiple internal banks, burst mode access, and pipelining of operation executions. Multiple internal banks enable accessing one bank while precharging or refreshing the other banks. Burst-mode access allows the controller to write or read multiple words to the memory without paying the penalty of Column Access Strobe (CAS) access for each word written or read. Of course, the bank must be active a few clock periods before the column access command can be issued on a row previously selected, a constraint known as Row Access Strobe (RAS) to CAS delay.

Typically, an SDRAM device includes four internal banks in the same chip and shares parts of the internal circuitry to reduce the chip area. Restricting the activation of four



- c_1: power-up timer terminated
- c_2: need auto-refresh
- c_3: all banks refreshed
- c_4: mode register not configured
- c_5: bank counter running
- c_6: initiation of a burst access
- c_7: current operation is read
- c_8: current operation is write
- c_9: a bank is still open or another address is programmed
- c_{10}: access is terminated

Fig. B.1. In this simplified state machine of the memory controller, the transactions are based on a burst read or write cycle of length four which permit a full time utilization of the data bus.

banks simultaneously, the device allows a time-multiplexed processing of the operations on any bank, as long as no timing violation occurs. Hence during a burst read or write, the controller performs four types of operations repetitively:

- **Activate:** selection of a random row in an idle bank.
- **NOP:** a necessary clock period to respect the RAS-to-CAS delay.
- **Read/write:** access to the column in the selected row of the activated bank.
- **Precharge:** deactivation of the row which was activated in a previous cycle.

In case of a write, data words must be provided on the bus in the same clock cycle as the CAS signal is asserted, i.e., during the write command. In case of a read, the first output appears a CAS-latency number of clock cycles on the data bus after the read command has been issued.

2 SDRAM Timings

Regarding SDRAM timings, some ground rules have to be respected. On the one hand, they concern the physical temporal constraints such as the well known setup and hold times [Rab03, Smi97, Wes93]. On the other hand, the internal controller of the memory

device needs some time to perform logical operations and move data words to or from the sense amplifiers, e.g., the CAS-latency of two or three clock cycles. Tab. B.1 summarizes the typical delays of a standard 100 MHz SDR SDRAM device.

Name	Duration	t_{clk}	Description
t_{clk}	10 ns	1	Clock cycle time (reference) considering $f_{\text{clk}} = 100$ MHz
t_{RCD}	20 ns	2	Row to column delay time (RAS to CAS delay)
t_{RP}	20 ns	2	Row precharge time, i.e., latency before data is written back
t_{CAS}	20 ns	2	Column access time (CAS latency)
t_{CCD}	10 ns	1	Column to column addressing delay within the same row
t_{RC}	70 ns	7	Row cycle time during which a row must stay active
t_{WR}	20 ns	2	Write recovery time (data input to precharge)
t_{REF}	64 ms	$6.4 \cdot 10^6$	Maximum refresh period for the capacitive loads

Table B.1. Typical functional timing characteristics for a standard 100 MHz SDR SDRAM device. The given time values are minima and only ought to provide an order of magnitude according to the actual industry standards.

Fig. B.2 represents a typical read access of 16 words in burst mode. Thereby, one data word of N bits is fetched to the memory data bus synchronously to the clock.

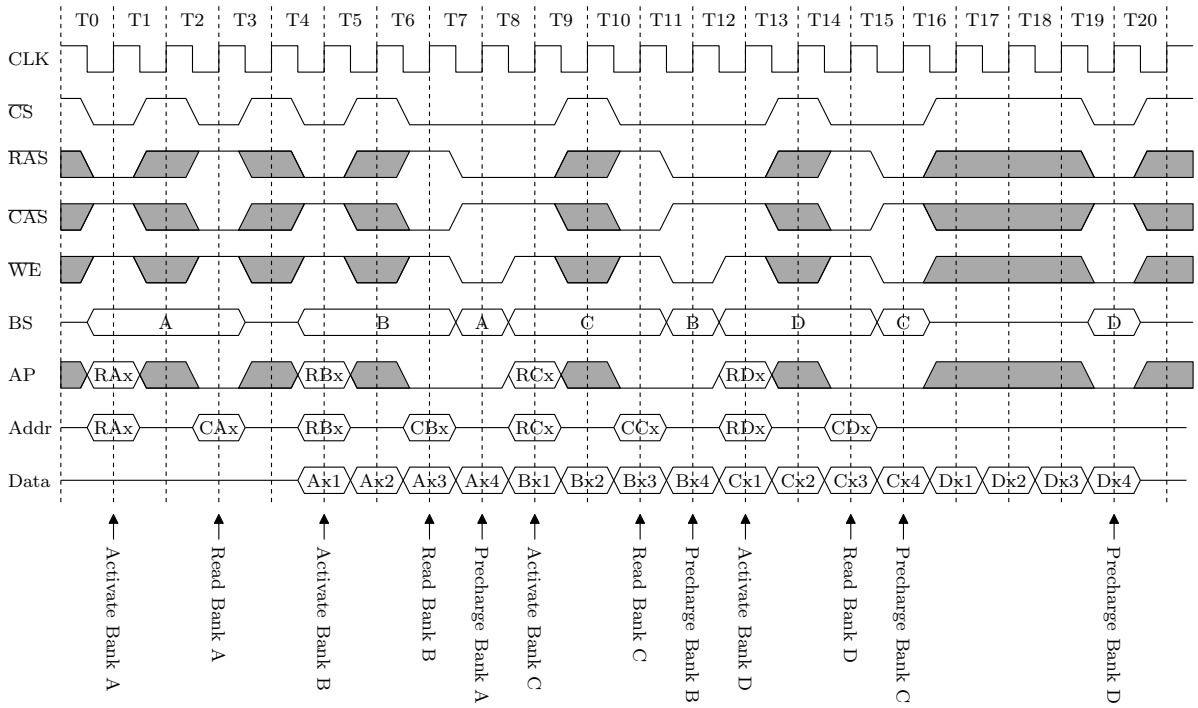


Fig. B.2. SDRAM timings diagram for the BAM controller. The transactions are performed in a random row read with precharge while interleaving the banks. The length of the burst access is set to four for a CAS latency of two cycles.

Pin	Name	Description
CLK	Clock	Inputs are sampled on the rising edge of CLK
CS	Chip Select	Enables the operations on the device
RAS	Row Acc. Str.	Initiates a row access in combination with CS and WE
CAS	Col. Acc. Str.	Initiates a column access in combination with CS and WE
WE	Write Enable	Indicates a read or a write operation
BS	Bank Select	Addresses the operated bank
AP	Autoprecharge	Forces a precharge of the row at the end of the burst
Addr	Address Bus	Multiplexed bus for row and column address
Data	Data Bus	Bidirectional bus for data transfer

Table B.2. Description of the most relevant signal pins used for interfacing a standard SDR SDRAM memory device.

List of Abbreviations

AAM	Associative Access Method – Lapir's information retrieval algorithm based on approximate matching using the BAM and text signatures.
AAF	Associative Access Filter – The filtering phase of the AAM which is being implemented in hardware.
ACE	Associative Computing Engine – Hardware accelerator supporting the AAF.
ALU	Arithmetical and Logical Unit – A part of a CPU (beside registers and control unit) executing the mathematical and logical operations.
ASCII	American Standard Code for Information Interchange – 8 bit standard for the code numbers of all the Latin letters, numbers and punctuation.
ASIC	Application Specific Integrated Circuit – A chip designed for a particular application as opposed to a general purpose circuit.
BAM	Bit Attribute Matrix – Arrangement of vectors coding the availability of binary features in textual documents.
BIF	BAM Interface – Memory interface to the BAM which permits a vertical access to the data, i.e., access by attributes.
CMOS	Complementary Metal Oxide Semiconductor – A process that uses both N and P channel devices in a complementary fashion.
CPU	Central Processing Unit – Processor chip executing the arithmetic and control portions of a sequential computer.
DRAM	Dynamic Random Access Memory – RAM that stores each bit of data in a separate capacitor.
FHT	Fragment Hit Table – A bit-vector that indicates if a query fragment has a penalty bigger than a preset threshold value.
FIFO	First In First Out – A type of data buffering that prevents data loss during high-speed communications.
FPGA	Field Programmable Gate Array – General purpose IC customized by interconnecting an array of programmable logic elements.
GCS	Global Controller and Scheduler – State machine scheduling the operations of the ACE using a fix set of micro-instructions.
HDD	Hard Disk Drive – Mechanical device storing a high density information magnetically on large capacity spinning discs.
IC	Integrated Circuit – Tiny complex of electronic components and their connections usually produced in silicon.
IP	Intellectual Property – Refers here to a precompiled module, the code of which is protected by copyrights and not available for modifications.
JTAG	Joint Test Action Group – IEEE Standard 1149.1 specifying how to control and monitor the pins of compliant devices on a printed circuit board.
LUT	Look-Up Table – A fast way of implementing any precalculated mathematical function based on a RAM paradigm.

PCI	Peripheral Component Interconnect – Local bus standard developed by Intel Corporation supporting up to 64 bit transfers at 66 MHz.
PCU	Penalty Calculating Unit – First module of the data-path, it computes fragment penalties based on the information given by the BAM and the query.
RAM	Random Access Memory – IC memory chip allowing information to be stored or accessed directly in any addressing order.
RMU	Result Merging Unit – Last module of the data-path, it saves the R best matching pages of the BAM.
ROM	Read Only Memory – A memory with unmodifiable contents.
RSU	Result Sorting Unit – Third module of the data-path, it sorts N pages from the BAM in parallel according to their respective scores.
RTL	Register Transfer Level – Modeling style corresponding to digital hardware synchronized by clock signals.
SCU	Score Calculating Unit – Second module of the data-path, it transforms the fragment penalties into page scores.
SDRAM	Synchronous Dynamic RAM – A cheap and compact form of RAM which needs to be periodically refreshed in order to retain its contents.
SIMD	Single Instruction Multiple Data – Parallel computer architecture consisting of a single instruction stream and multiple data stream.
SOC	System On Chip (also SoC) – Highly integrated device composed of multiple functional blocks, including on-chip memory and a processor.
SRAM	Static RAM – A fast access memory that retains data as long as power is being supplied, without having to be periodically refreshed.
VFA	Vertical Fragment Accumulator – Internal register structure for the accumulation for penalties calculated within one query fragment.
VHDL	VHSIC Hardware Description Language – IEEE 1076-standard high-level language for designing and simulating electronic systems.
VHSIC	Very High Speed Integrated Circuit – Abbreviation coined by the US Department of Defense in the 1980s for the development of VHDL.
VLIW	Very Long Instruction Word – Packing many simple RISC-like instructions into a much longer internal instruction word format.
VLSI	Very Large Scale Integration – Process of placing more than hundred thousands transistors on a single chip.
VQA	Vertical Query Accumulator – Internal SCU register structure for the calculation of page scores during one query.

Notations: According to Knuth [Knu76], big O, big Ω and big Θ notations are such that when $n \to \infty$

- $O(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants k and n_0 with $|g(n)| \leq k \cdot f(n)$ for all $n \geq n_0$;
- $\Omega(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants k and n_0 with $g(n) \geq k \cdot f(n)$ for all $n \geq n_0$;
- $\Theta(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants k_1, k_2, and n_0 with $k_1 \cdot f(n) \leq g(n) \leq k_2 \cdot f(n)$ for all $n \geq n_0$.

Bibliography

[Abe03a] K. H. Abed, R. E. Siferd, "VLSI Implementation of a Low-Power Antilogarithmic Converter", *IEEE Trans. Computers*, Vol. 52, No. 9, pp. 1221-1228, 2003.

[Abe03b] K. H. Abed, R. E. Siferd, "CMOS VLSI Implementation of a Low-Power Logarithmic Converter", *IEEE Trans. Computers*, Vol. 52, No. 11, pp. 1421-1433, 2003.

[Ahn04] J. Ahn, W. Dally, B. Khailany, U. Kapasi, A. Das, "Evaluating the Imagine Stream Architecture", *Proc. IEEE Int. Symp. Computer Architecture (ISCA'04)*, pp. 14-25, 2004.

[Ajt83] M. Ajtai, J. Komlos, E. Szemeredi, "An $O(N \log N)$ Sorting Network", *Proc. ACM Int. Symp. Theory of Computing (STOC'83)*, pp. 1-9, 1983.

[Amd67] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *Proc. AFIPS Spring Joint Computer Conf.*, Vol. 30, pp. 483-485, 1967.

[And94] A. Andersson, S. Nilsson, "A New Efficient Radix Sort", *Proc. IEEE Symp. Foundations of Computer Science (FOCS'94)*, pp. 714-721, 1994.

[Arn03] M. Arnold, T. Bailey, J. Cowles, "Error Analysis of the Kmetz/Maenner Algorithm", *Journal of VLSI Signal Processing*, Vol. 33, pp. 37-53, 2003.

[Bae99] R. Baeza-Yates, B. Ribeiro-Neto, "Modern Information Retrieval", *ACM Press / Addison Wesley*, ISBN 0-201-39829-X, 1999.

[Bak04] Z. Baker, V. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs", *Proc. ACM Int. Symp. Field Programmable Gate Arrays (FPGA'04)*, pp. 223-232, 2004.

[Bak05] Z. Baker, V. Prasanna, "Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs", *Proc. IEEE Symp. Field Programmable Custom Computing Machines (FCCM'05)*, pp. 3-12, 2005.

[Bak06] Z. Baker, V. Prasanna, "An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing System", *Proc. IEEE Symp. Field Programmable Custom Computing Machines (FCCM'06)*, pp. 67-75, 2006.

[Bas91] F. Baskett, "Keynote Address", *Int. Symp. Shared Memory Multiprocessing*, 1991.

[Bat68] K. E. Batcher, "Sorting Networks and Their Applications", *Proc. AFIPS Spring Joint Computer Conf.*, Vol. 32, pp. 307-314, 1968.

[Bat69] K. E. Batcher, "Means for Merging Data", *US Patent*, No. 3428946, 1969.

[Bat76] K. E. Batcher, "Solid State Associative Processor Organization", *US Patent*, No. 3936806, 1976.

[Bee03] P. van Beek, J. Smith, T. Ebrahimi, T. Suzuki, J. Askelof, "Metadata-Driven Multimedia Access", *IEEE Signal Processing Magazine*, Vol. 20, No. 2, pp. 40-52, 2003.

[Ber98] S. Berkovich, E. El-Qawasmeh, G. Lapir, "Organization of Near Matching in Bit Attribute Matrix Applied to Associative Access Methods in Information Retrieval", *Proc. IASTED Int. Conf. Applied Informatics*, pp. 62-65, Feb. 1998.

[Bet99] V. Betz, J. Rose, A. Marquardt, "Architecture and CAD for Deep Submicron FPGA", *Kluwer Academic*, ISBN: 0-7923-8460-1, 1999.

[Blü00a] H.-M Blüthgen, T. Noll, "A Programmable Processor for Approximate String Matching With High Throughput Rate", *Proc. IEEE Int. Conf. Application Specific Systems, Architectures, and Processors (ASAP'00)*, pp. 309-316, 2000.

[Blü00b] H.-M Blüthgen, P. Osterloh, H. Blume, T. Noll, "A Hardware Implementation for Approximate Text Search in Multimedia Applications", *Proc. IEEE Int. Conf. Multimedia and Expo (ICME'00)*, pp. 1425-1428, 2000.

[Blu00] H. Blume, H.-M. Blüthgen, C. Henning, P. Osterloh, "Integration of High-Performance ASICs into Reconfigurable Systems Providing Additional Multimedia Functionality", *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures and Processors (ASAP'00)*, pp. 66-75, 2000.

[Blu01] H. Blume, H. Feldkämper, H. Hübert, T. Noll, "Analyzing Heterogeneous System Architectures by Means of Cost Functions: Acomparative Study for Basic Operations", *Proc. European Solid-State Circuits Conf. (ESSCIRC'01)*, pp. 424-427, 2001.

[Boy77] R. Boyer, J. Moore, "A fast String Searching Algorithm", *Comm. ACM*, Vol. 20, No. 10, pp. 762-772, 1977.

[Bri98] S. Brin, L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine", *Proc. Int. World Wide Web Conf.*, Vol. 30, pp. 107-117, 1998.

[Bur96] D. Burger, J. Goodman, and A. Kägi, "Memory Bandwidth Limitations of Future Microprocessors", *Proc. ACM Int. Symp. Computer Architecture (ISCA'96)*, pp. 78-89, 1996.

[Bur97a] D. Burger, J. Goodman, "Billion-Transistor Architectures", *IEEE Computer*, Vol. 30, No. 9, pp. 46-47, 1997.

[Bur97b] D. Burger, "System-Level Implications of Processor-Memory Integration", *Mixing Logic and DRAM Workshop at Int. Symp. on Computer Architecture (ISCA'97)*, 1997.

[Bur04] D. Burger, J. Goodman, "Billion-Transistor Architectures: There and Back Again", *IEEE Computer*, Vol. 37, No. 3, pp. 22-28, 2004.

[Bur02] S. Burkhardt, "Filter Algorithms for Approximate String Matching", *Ph.D. Thesis*, Universität des Saarlandes, Germany, 2002.

[Cas04] S. Cass, "A Fountain of Knowledge", *IEEE Spectrum*, Vol. 41, No. 1, 2004.

[Cla01] K. Claessen, M. Sheeran, S. Singh, "The Design and Verification of a Sorter Core", *Proc. Conf. Correct Hardware Design and Verification Methods (CHARME'01)*, LNCS Vol. 2144, pp. 355-369, 2001.

[Col98] R. Cole, R. Hariharan, "Approximate String Matching: A Simpler Faster Algorithm", *Proc. ACM-SIAM Symp. Discrete Algorithms (SODA'98)*, pp. 463-472, 1998.

[Cor01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, "Introduction to Algorithms", Sec. Edition, *MIT Press & McGraw-Hill*, ISBN: 0262032937, 2001.

[Cri91] J. K. Cringean, G. A. Manson, R. England, P. Willett, "Nearest Neighbor Searching in Files of Text Signatures Using Transputer Networks", *Electronics publishing, J. Wiley & Sons*, Vol. 4, pp. 185-203, 1991.

[Cut78] R. M. Cutler, "Associative Processors", *US Patent*, No. 4068305, 1978.

[Dav86] W. A. Davis, D. L. Lee, "Fast Search Algorithms for Associative Memories", *IEEE Trans. Computers*, Vol. C-35, No. 5, pp. 456-461, 1986.

[DeH99a] A. DeHon, "Balancing Interconnect and Computation in a Reconfigurable Computing Array", *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays (FPGA'99)*, pp. 69-78, 1999.

[DeH99b] A. DeHon, J. Wawrzynek, "Reconfigurable Computing: What, Why, and Implications for Design Automation", *Proc. ACM/IEEE Design Automation Conf. (DAC'99)*, pp. 610-615, 1999.

[DeH00a] A. DeHon, "Compact, Multilayer Layout for Butterfly Fat-Tree", *Proc. ACM Symp. Parallel Algorithms and Architectures (SPAA'00)*, pp. 206-215, 2000.

[DeH00b] A. DeHon, "The Density Advantage of Configurable Computing", *IEEE Computer*, Vol. 33, No. 4, pp. 41-49, 2000.

[DeM94] G. De Micheli, "Synthesis and Optimization of Digital Circuits", *McGraw-Hill*, ISBN: 0070163332, 1994.

[Dho05] S. H. Dhong, O. Takahashi, M. White, T. Asano, T. Nakazato, J. Silberman, A. Kawasumi, H. Yoshihara, "A 4.8GHz Fully Pipelined Embedded SRAM in the Streaming Processor of a CELL Processor", *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC'05)*, Vol. 1, pp. 486-612, 2005.

[DiB05] A. Di Blas, D. Dahle, M. Diekhans, L. Grate, J. Hirschberg, K. Karplus, H. Keller, M. Kendrick, F. Mesa, D. Pease, E. Rice, A. Schultz, D. Speck, R. Hughey, "The UCSC Kestrel Parallel Processor", *IEEE Trans. Parallel and Distributed Systems*, Vol. 16, No. 1, pp. 80-92, 2005.

[Die94] M. Dietzfelbinger, A. Karlin, K. Melhorn, F. Meyer, H. Rohnert, R. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds", *SIAM Journal of Computing*, Vol. 23, No. 4, pp. 738-761, 1994.

[Dra02] J. Draper, J. Sondeen, C. Kang, "Implementation of a 256-bit Wide Word Processor for the Data-Intensive Architecture Processing-In-Memory Chip", *Proc. IEEE European Solid-State Circuit Conf. (ESSCIRC'02)*, pp. 77-80, 2002.

[Ell92] D. Elliott, M. Snelgrove, M. Stumm, "Computational RAM: A Memory-SIMD Hybrid and its Application to DSP", *Proc. IEEE Custom Integrated Circuit Conf. (CICC'92)*, pp. 30.6.1 -30.6.4, 1992.

[Eve98] S. Even, S. Muthukrishnan, M. Paterson, S. Sahinalp, "Layout of the Batcher Bitonic Sorter", *Proc. ACM Symp. Parallel Algorithms and Architectures (SPAA'98)*, pp. 172-181, 1998.

[Fau91] P. Faudemay, M. Mhiri, "An Associative Accelerator for Large Databases", *IEEE Micro*, Vol. 11, No. 6, pp. 22-34, 1991.

[Fel02] H. Feldkämper, T. Gemmeke, H. Blume, T. Noll, "Analysis of Reconfigurable and Heterogeneous Architectures in the Communication Domain", *Proc. IEEE Int. Conf. Circuits and Systems for Communications (CCSC'02)*, pp. 190-193, 2002.

[Gro03] E. Grochowski, R. D. Halem, "Technological Impact of Magnetic Hard Disk Drives on Storage Systems", *IBM Systems Journal*, Vol. 42, No. 2, pp. 338-346, 2003.

[Hal70] E. L. Hall, D. D. Lynch, S. J. Dwyer, "Generation of Products and Quotients Using Approximate Binary Logarithms for Digital Filtering Applications", *IEEE Trans. Computers*, Vol. 19, No. 2, pp. 97-105, 1970.

[Hal99] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture", *Proc. ACM/IEEE Conf. Supercomputing (SC'99)*, Art. No. 57, 1999.

[Hen90] J. L. Hennessy, D. A. Patterson, "Computer Architecture: a Quantitative Approach", *Morgan-Kaufman*, San Mateo, CA, 1990.

[Hen98] C. Henning, T. G. Noll, "Architecture and Implementation of a Bitserial Sorter for Weighted Median Filtering", *Proc. IEEE Custom Integrated Circuits Conf. (CICC'98)*, pp. 189-192, 1998.

[Hoa62] C. A. R. Hoare, "Quicksort", *Computer Journal*, Vol. 5, No. 1, pp. 10-15, 1962.

[Hoe91] B. Höfflinger, M. Selzer, F. Warkovski, "Digital Logarithmic CMOS Multiplier for Very-High-Speed Signal Processing", *Proc. IEEE Custom Integrated Circuits Conf. (CICC'91)*, pp. 16.7.1-16.7.5, 1991.

[Hug02] G. F. Hughes, "Wise Drives", *IEEE Spectrum*, Vol. 39, No. 8, 2002.

[Hyy02] H. Hyyrö, G. Navarro, "Faster Bit-Parallel Approximate String Matching", *Proc. Annual Symp. Combinatorial Pattern Matching (CPM'02)*, LNCS Vol. 2373, pp. 203-224, 2002.

[Hyy04] H. Hyyrö, K. Fredriksson, G. Navarro, "Increased Bit-Parallelism for Approximate and Multiple String Matching", *Proc. Workshop on Efficient and Experimental Algorithms (WEA'04)*, LNCS Vol. 3059, pp. 285-298, 2004.

[Hyy05] H. Hyyrö, G. Navarro, "Bit-Parallel Witnesses and their Applications to Approximate String Matching", *Algorithmica*, Vol. 41, No. 3, pp. 203-231, 2005.

[Int06] Intel Corporation, *http://www.intel.com*, 2006.

[Ito01] K. Itoh, "VLSI Memory Chip Design", *Springer Verlag*, ISBN: 3-540-67820-4, 2001.

[Itr06] International Technology Roadmap for Semiconductors, *http://public.itrs.net*, 2006.

[Jai89] A. K. Jain, "Fundamentals of Digital Image Processing", *Prentice Hall*, ISBN: 0-13-336165-9, 1989.

[Kal02] H. Kalte, M. Porrmann, U. Rückert, "A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs", *Proc. IEEE Workshop on Heterogeneous Reconfigurable Systems on Chip*, 2002.

[Kan99] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System", *Proc. IEEE Int. Conf. on Computer Design (ICCD'99)*, pp. 192-201, 1999.

[Kar87] R. M. Karp, M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms", *IBM Journal of Research and Development*, Vol. 31, No. 2, pp. 249-260, 1987.

[Kha02] B. Khailany, W. Dally, A. Chang, U. Kapasi, J. Namkoong, B. Towles, "VLSI Design and Verification of the Imagine Processor", *Proc. IEEE Int. Conf. Computer Design (ICCD'02)*, pp. 289-296, 2002.

[Kno98] A. Knoll, C. Altenschmidt, J. Biskup, H.-M. Blüthgen, I. Glöckner, S. Hartrumpf, H. Helbig, C. Henning, R. Lüling, B. Monien, T. G. Noll, N. Sensen, "An Integrated Approach to Semantic Evaluation and Content-Based Retrieval of Multimedia Documents", *Proc. European Conf. Research and Advanced Technology for Digital Libraries*, LNCS Vol. 1513, pp. 409-428, 1998.

[Knu76] D. E. Knuth, "Big Omicron and Big Omega and Big Theta", *ACM SIGACT News*, Vol. 8, Issue 2, pp. 18-24, 1976.

[Knu77] D. E. Knuth, J. H. Morris, V. R. Pratt, "Fast pattern matching in strings", *SIAM Journal on Computing*, Vol. 6, No. 1, pp. 323-350, 1977.

[Knu97] D. E. Knuth, "The Art of Computer Programming, Volume 3: Sorting and Searching", Sec. Edition, *Addison Wesley*, ISBN: 0-201-89685-0, 1997.

[Kob00] M. Kobayashi, K. Takeda, "Information Retrieval on the Web", *ACM Computing Surveys*, Vol. 32, No. 2, pp. 144-173, 2000.

[Kor93] I. Koren, "Computer Arithmetic Algorithms", Sec. Edition, *A. K. Peters, Natick, MA*, ISBN: 1-56881-160-8, 2002.

[Kos91] D. K. Kostopoulos, "An Algorithm for the Computation of Binary Logarithms", *IEEE Trans. Computers*, Vol. 40, No. 11, pp. 1267-1270, 1991.

[Koz00] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, K. Yelick, "Vector IRAM: A Media-Oriented Vector Processor with Embedded DRAM", *Proc. Hot Chips Conf.*, 2000.

[Koz02] C. Kozyrakis, "Scalable Vector Media Processors for Embedded Systems", *Ph.D. Thesis*, Technical Report UCB-CSD-02-1183, University of California at Berkeley, 2002.

[Kri94] A. Krikelis, C. C. Weems, "Associative Processing and Processors", *IEEE Computer*, Vol. 27, No. 11, pp. 12-17, 1994.

[Kun88] S. Y. Kung, "VLSI Array Processors", *Prentice Hall*, ISBN: 013942749X, 1988.

[Lap92] G. M. Lapir, "Use of Associative Access Method Information Retrieval Systems", *Proc. Pittsburgh Conf. Modeling and Simulation*, Vol. 23, No. 2, pp. 951-958, 1992.

[Lap02] G. M. Lapir, H. Urbschat, "Associative Memory", *WIPO Patent*, IPN. WO 02/15045A2, 2002.

[Lay03] C. Layer, H.-J. Pfleiderer, P. Ruján, G. Lapir, "High Performance System Architecture of an Associative Computing Engine Optimized for Search Algorithms", *Proc. IFIP Int. Conf. Very Large Scale Integration of SoC (VLSI-SOC'03)*, pp. 74-79, 2003.

[Lay04a] C. Layer, H.-J. Pfleiderer, C. Heer, "A Scalable Compact Architecture for the Computation of Integer Binary Logarithms Through Linear Approximation", *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS'04)*, Vol. 2, pp. 421-424, 2004.

[Lay04b] C. Layer, H.-J. Pfleiderer, "A Reconfigurable Recurrent Bitonic Sorting Network for Concurrently Accessible Data", *Proc. Int. Field Programmable Logic Conf. (FPL'04)*, LNCS 3203, Springer-Verlag, pp. 648-657, 2004.

[Lay05a] C. Layer, H.-J. Pfleiderer, "Vertical Sorting Techniques Accelerating Associative Accesses Based Information Retrieval Systems", *Proc. IASTED Int. Conf. Applied Informatics*, pp. 411-416, 2005.

[Lay05b] C. Layer, H.-J. Pfleiderer, "Efficient Hardware Search Engine for Associative Content Retrieval of Long Queries in Huge Multimedia Databases", *Proc. IEEE Int. Conf. Multimedia and Expo (ICME'05)*, pp. 1034-1037, 2005.

[Lay05c] C. Layer, H.-J. Pfleiderer, "A Scalable Highly Parallel VLSI Architecture Dedicated to Associative Computing Algorithms", *Proc. IEEE Ph.D. Research in Microelectronics and Elecronics (PRIME'05)*, Vol. 2, pp. 214-217, 2005.

[Lay05d] C. Layer, H.-J. Pfleiderer, "Hardware Implementation of an Approximate String Matching Algorithm Using Bit Parallel Processing for Text Information Retrieval Systems", *Proc. IEEE Int. Conf. Signal Processing Systems (SIPS'05)*, pp. 193-198, 2005.

[Lee00] J. D. Lee, K. E. Batcher, "Minimizing Communication in the Bitonic Sort", *IEEE Trans. Parallel and Distributed Systems*, Vol. 11, No. 5, pp. 459-474, 2000.

[Lee91] K. C. Lee, T. M. Hickey, V. W. Mak, G. E. Herman, "VLSI Accelerators for Large Database Systems", *IEEE Micro*, Vol. 11 , No. 6, pp. 8-20, 1991.

[Lei85] F. Leighton, "Tight Bounds on the Complexity of Parallel Sorting", *IEEE Trans. Computers*, Vol. 34, No. 4, pp. 344-354, 1985.

[Les97] M. Lesk, "How Much Information Is There In The World?", *Technical Report*, 1997.

[Lew94] D. M. Lewis, "Interleaved Memory Functions Interpolators with Application to Accurate LNS Arithmetic Unit", *IEEE Trans. Computers*, Vol. 43, No. 8, pp. 974-982, 1994.

[Lew95] D. M. Lewis, "A 114 MFLOPS Logarithmic Number System Arithmetic Unit for DSP Applications", *IEEE Journal Solid State Circuits*, Vol. 30, pp. 1547-1553, 1995.

[Lom83] D. B. Lomet, "A High Performance, Universal, Key Associative Access Method", *ACM SIGMOD Record Archive*, Vol. 13, No. 4, pp. 120-133, 1983.

[Lym03] P. Lyman, H. R. Varian, "How Much Information?", *Technical Report*, School of Information Management and Systems, U. C. Berkeley, http://www.sims.berkeley.edu/research/projects/how-much-info, 2003.

[Mai00] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, M. Horowitz, "Smart Memories: A modular Reconfigurable Architecture", *Proc. Int. Symp. Computer Architecture (ISCA'00)*, pp. 161-171, 2000.

[Mar71] W. A. Martin, "Sorting", *ACM Computing Surveys*, Vol. 3, No. 4, pp. 147-174, 1971.

[McK04] S. A. McKee, "Reflections on the Memory Wall", *Proc. ACM Conf. Computing Frontiers*, pp. 162, 2004.

[Mit62] J. N. Mitchell, "Computer Multiplication and Division Using Binary Logarithm", *IRE Trans. Electronic Computers*, Vol. EC-11, pp. 512-517, 1962.

[Moo03] G. E. Moore, "No Exponential Is Forever: But Forever Can Be Delayed", *IEEE Int. Solid State Circuits Conf. (ISSCC'03)*, Digest of Technical Papers, Vol. 1, pp. 20-23, 2003.

[Moo06] S. K. Moore, "Multimedia Monster", *IEEE Spectrum*, Jan. 2006.

[Mut99] S. Muthukrishnan, M. Paterson, S. Sahinalp, T. Suel, "Compact Grid Layouts of Multi-Level Networks", *Proc. ACM Symp. Theory of Computing (STOC'06)*, pp. 455-463, 1999.

[Nak89] T. Nakatani, S.-T. Huang, B. W. Arden, S. K. Tripathi, "K-Way Bitonic Sort", *IEEE Trans. Computers*, Vol. 38, No. 2, pp. 283-288, 1989.

[Nav01] G. Navarro, "A Guided Tour to Approximate String Matching", *ACM Computing Surveys*, Vol. 33, pp. 31-88, 2001.

[Nav02] G. Navarro, M. Raffinot, "Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences", *Cambridge University Press*, ISBN: 0521813077, 2002.

[Nav05] G. Navarro, "Text Databases", *Encyclopedia of Database Technologies and Applications*, pp. 688-694, ISBN: 1-59140-560-2, 2005.

[Ola99] S. Olariu, M. C. Pinotti, S. Q. Zheng, "How to Sort N Items Using a Sorting Network of Fixed I/O Size", *IEEE Trans. Parallel and Distributed Systems*, Vol. 10, No. 5, pp. 487-499, 1999.

[Ola00] S. Olariu, M. C. Pinotti, S. Q. Zheng, "An Optimal Hardware-Algorithm for Sorting Using a Fixed-Size Parallel Sorting Device", *IEEE Trans. Computers*, Vol. 49, No. 12, pp. 1310-1324, 2000.

[Olu05] K. Olukotun, L. Hammond, "The Future of Microprocessors", *ACM Queue*, No. 9, 2005.

[Osk98] M. Oskin, F. Chong, T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory", *Proc. Int. Symp. Computer Architecture (ISCA'98)*, pp. 192-203, 1998.

[Pag98] L. Page, S. Brin, R. Motwani, T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web", *Stanford Digital Library*, Technologies Project, 1998.

[Par73a] B. Parhami, "Associative Memories and Processors: An Overview and Selected Bibliography", *Proc. IEEE*, Vol. 61, Iss. 6, pp. 722-730, 1973.

[Par73b] B. Parhami, A. Avizienis, "Design of Fault-Tolerant Associative Processors", *Proc IEEE Int. Symp. Computer Architecture (ISCA'73)*, pp. 141-145, 1973.

[Par99] B. Parhami, D. M. Kwai, "Data-Driven Control Scheme for Linear Arrays: Application to a Stable Insertion Sorter", *IEEE Trans. Parallel and Distributed Systems*, Vol. 10, No. 1, pp. 23-28, 1999.

[Pat97a] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", Sec. Edition, *Morgan Kaufmann*, ISBN: 1-55860-491-X, 1997.

[Pat97b] D. A. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick, "Intelligent RAM (IRAM): Chips that Remember and Compute", *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC'97)*, 1997.

[Per99] S. Perissakis, Y. Joo, J. Ahn, A. DeHon, J. Wawrzynek, "Embedded DRAM for a Reconfigurable Array", *Proc. IEEE Int. Symp. VLSI Circuits (ISVLSI'99)*, pp. 145-148, 1999.

[Pot94] J. Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, C. Asthagiri, "ASC: An Associative Computing Paradigm", *IEEE Comp.: Special Issue on Associative Processing and Processors*, pp. 19-25, 1994.

[Rab03] J. M. Rabaey, A. Chandrakasan, B. Nikolic, "Digital Integrated Circuits: A Design Perspective", Sec. Edition, *Prentice Hall, NJ*, ISBN 0-13-120764-4, 2003.

[Ruh85] S. Ruhman, I. Scherson, "Associative Processors Particularly Useful for Tomographic Image Reconstructions", *US Patent*, No. 4491932, 1985.

[Sas95] R. Sastry, N. Ranganathan, K. Remedios, "CASM: A VLSI Chip for Approximate String Matching", *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 17, No. 8, pp. 824-830, 1995.

[Sch89] I. D. Scherson, S. Sen, "Parallel Sorting in Two-Dimensional VLSI Models of Computation", *IEEE Trans. Computers*, Vol. 38, pp. 238-249, 1989.

[Sch92] I. D. Scherson, D. Kramer, B. Alleyne, "Bit-Parallel Arithmetic in a Massively-Parallel Associative Processor", *IEEE Trans. Computer*, Vol. 41, No. 10, pp. 1201-1210, 1992.

[Sed88] R. Sedgewick, "Algorithms", Sec. Edition, *Addison Wesley Longman*, ISBN 0-201-06673-4, 1988.

[She59] D. L. Shell, "A High-Speed Sorting Procedure", *Communications ACM*, Vol. 2, No. 7, pp. 30-32, 1959.

[Smi97] M. Smith, "Application-Specific Integrated Circuits", *Addison Wesley*, ISBN: 0-201-50022-1, 1997.

[Sto71] H. S. Stone, "Parallel Processing with the Perfect Shuffle", *IEEE Trans. Computers*, Vol. C-20, No. 2, pp. 153-161, 1971.

[Sun96] T. Sunaga, P. Kogge, "A Processor in Memory Chip for Massively Parallel Embedded Applications", *IEEE Journal Solid-State Circuits*, pp. 1556-1559, 1996.

[Sun90] D. M. Sunday, "A Very Fast Substring Search Algorithm", *Communications ACM*, Vol. 33(8), pp. 132-142, 1990.

[Syd03] T. von Sydow, H. Blume, T. G. Noll, "Performance Analysis of General Purpose and Digital Signal Processor Kernels for Heterogeneous Systems-on-Chip" *URSI Advances in Radio Science*, Vol. 1, pp. 171-175, 2003.

[Szy97] T. H. Szymanski, "Design Principles for Practical Self-Routing Nonblocking Switching Networks with $O(N \cdot \log N)$ Bit-Complexity", *IEEE Trans. Computers*, Vol. 46, No. 10, pp. 1057-1069, 1997.

[Tav94] D. Tavangarian, "Flag Oriented Parallel Associative Architectures and Applications", *IEEE Trans. Computers*, Vol. 27, No. 11, pp. 41-52, 1994.

[Tay88] F. J. Taylor, R. Gill, J. Joseph, J. Radke, "A 20 Bit Logarithmic Number System Processor", *IEEE Trans. Computers*, Vol. 37, No. 2, pp. 190-200, 1988.

[Tho83] C. D. Thompson, "The VLSI Complexity of Sorting", *IEEE Trans. Computers*, Vol. C-32, No. 12, pp. 1171-1184, 1983.

[Urb02] H. Urbschat, G. Lapir, "Associative Memory", *European Patent Application*, IPN. EP 1182577A1, 2002.

[Vui96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Trans. VLSI Systems*, Vol. 4, No. 1, pp. 56-69, 1996.

[Wag74] R. A. Wagner, M. J. Fischer, "The String-to-String Correction Problem", *Journal ACM*, Vol. 21, No. 1, pp. 168-73, 1974.

[Wai97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua. J. Babb, S. Amarasinghe, A. Agarwal, "Baring It All to Software: Raw Machine", *IEEE Computer*, Vol. 30, No. 9, pp. 86-93, 1997.

[Wal01] R. Walker, J. Potter, Y. Wang, M. Wu, "Implementing Associative Processing: Rethinking Earlier Architectural Decisions", *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS'01)*, 2001.

[Wan03] H. Wang, R. Walker, "Implementing a Scalable ASC Processor", *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS'03)*, pp. 267, 2003.

[Wes93] N. Weste, K. Eshraghian, "Principles of CMOS VLSI Design", *Addison Wesley Longman*, Sec. Edition, ISBN 0-201-53376-6, 1993.

[Wil04] S. Wilton, S.-S. Ang, W. Luk, "The Impact of Pipelining on Energy per Operation in Field-Programmable Gate Arrays", *Proc. Int. Conf. Field Programmable Logic and its Applications (FPL'04)*, LNCS 3203, Springer-Verlag, pp. 719-728, 2004.

[Wit99] I. Witten, A. Moffat, T. Bell, "Managing Gigabytes", Sec. Edition, *Morgan Kaufmann Publishers*, ISBN: 1-55860-570-3, 1999.

[Wol02] W. Wolf, "Modern VLSI Design: System-on-Chip Design", *Prentice Hall*, Thi. Edition, ISBN: 0-13-061970-1, 2002.

[Wol04] W. Wolf, "FPGA-Based System Design", *Prentice Hall*, ISBN: 0-13-142461-0, 2004.

[Wul95] W. Wulf, S. McKee, "Hitting the Memory Wall: Implications of the Obvious", *SIGARCH Computer Architecture News*, Vol. 23, No. 1, pp. 20-24, 1995.

[Wu92] S. Wu, U. Manber, "Fast Text Searching Allowing Errors", *Communications ACM*, Vol. 35, No. 10, pp. 83-91, 1992.

[Wu02] M. Wu, R. Walker, J. Potter, "Implementing Associative Search and Responder Resolution", *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS'02)*, pp. 246-253, 2002.

[Xil06] Xilinx Corporation, "http://www.xilinx.com", *Xilinx FPGA Family Reference Guide*, 2006.

[Yeh00] C. Yeh, B. Parhami, E. Varvarigos, H. Lee, "VLSI Layout and Packaging of Butterfly Networks", *Proc. ACM Symp. Parallel Algorithms and Architectures (SPAA'00)*, 2000.

[Yeh03] C. Yeh, "Optimal Layout for Butterfly Networks in Multilayer VLSI", *Proc. IEEE Int. Conf. Parallel Processing (ICPP'03)*, pp. 379-388, 2003.

[Yin96] L. Yin, R. Yang, M. Gabbouj, Y. Neuvo, "Weighted Median Filters: A Tutorial", *IEEE Trans. Circuits and Systems*, Vol. 43, Nr. 3, pp. 157-192, 1996.

[Yoo00] S. Yoo, J. Renau, M. Huang, J. Torrellas, "FlexRAM Architecture Design Parameters", *Technical Report CSRD-1584*, University of Illinois at Urbana-Champaign, 2000.

[Yu06] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, B. Baas, "An Asynchronous Array of Simple Processors for DSP Applications", *IEEE Int. Solid-State Circuits Conf. (ISSCC'06)*, pp. 428-430, 2006.

[Zha01] L. Zhang, Z. Fang, M. Parker, B. Mathew, L. Schaelicke, J. Carter, W. Hsieh, S. McKee, "The Impulse Memory Controller", *IEEE Trans. Computers*, Vol. 50, No. 11, pp. 1117-1132, 2001.