Universidade de Aveiro
2009

Departamento de Electrónica, Telecomunicações e Informática

BRUNO FIGUEIREDO PIMENTEL

Síntese de aceleradores baseados em FPGAs implementando algoritmos recursivos

Synthesis of FPGA-based accelerators implementing recursive algorithms

Universidade de Aveiro
2009

Departamento de Electrónica, Telecomunicações e Informática

BRUNO FIGUEIREDO PIMENTEL

Síntese de aceleradores baseados em FPGAs implementando algoritmos recursivos

Synthesis of FPGA-based accelerators implementing recursive algorithms

dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutorado em Engenharia Informática, realizada sob a orientação científica do Dr. Valeri Skliarov, Professor Catedrático do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro, e co-orientação da Dr.ª Iouliia Skliarova, Professora Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Apoio financeiro da FCT e do FSE no âmbito do III Quadro Comunitário de Apoio.

Dedico este trabalho aos meus pais – Noémia e João.

o júri

presidente

Prof. Dr. Paulo Jorge de Melo Matias Faria de Vila Real
professor catedrático da Universidade de Aveiro

Prof. Dr. António Manuel de Brito Ferrari Almeida
professor catedrático da Universidade de Aveiro

Prof. Dr. Valeri Skliarov
professor catedrático da Universidade de Aveiro (orientador)

Prof. Dr. Horácio Cláudio de Campos Neto
professor associado do Instituto Superior Técnico da Universidade Técnica de Lisboa

Prof. Dr. Henrique Manuel Dinis Santos
professor associado da Universidade do Minho

Prof.ª Dr.ª Iouliia Skliarova
professora auxiliar da Universidade de Aveiro (co-orientadora)

agradecimentos

Chegado ao fim desta importante etapa, quero expressar o meu agradecimento:

Ao Prof. Dr. Valeri Skliarov e à Prof.ª Dr.ª Iouliia Skliarova, pela sábia orientação, pelo dedicado contributo para este trabalho e pela querida amizade que deles recebi ao longo deste período;

Ao Prof. Dr. António de Brito Ferrari e ao Prof. Dr. António Rui Borges, pelas palavras amigas e de grande motivação que me foram dirigindo durante este trabalho;

Ao Prof. Dr. Rui Tomaz Valadas e ao Prof. Dr. Luís Seabra Lopes, pela simpatia demonstrada, nomeadamente com a escrita das cartas de referência;

À Prof.ª Dr.ª Ana Maria Tomé e ao Prof. Dr. Rui Manuel Escadas, pelo interesse demonstrado pelo andamento deste trabalho e pelas palavras amigas que me dirigiram em várias ocasiões;

À Fundação para a Ciência e a Tecnologia por todo o apoio financeiro no âmbito da Bolsa de Doutoramento que me concedeu;

Aos membros de diferentes órgãos da Universidade de Aveiro e ao Instituto de Engenharia Electrotécnica e Telemática de Aveiro pelas boas condições que me proporcionaram;

Ao meu pai, pelos importantes conselhos orientadores que me deu desde muito cedo;

À minha família, em particular à minha mãe, ao meu irmão César, à Ana, ao Alexandre e à Helena, por todo um confortante suporte familiar que me deram;

Ao meu amigo Aneesh Chauhan e ao meu irmão César, pelo companheirismo e pelas inúmeras e produtivas trocas de ideias;

Ao meu colega e amigo Manuel Almeida, pela saudável cooperação e companheirismo que tornaram confortável o laboratório que partilhámos.

palavras-chave

FPGA, algoritmos recursivos, computação reconfigurável, desenvolvimento de sistemas computacionais

resumo

O desenvolvimento de sistemas computacionais é um processo complexo, com múltiplas etapas, que requer uma análise profunda do problema, levando em consideração as limitações e os requisitos aplicáveis. Tal tarefa envolve a exploração de técnicas alternativas e de algoritmos computacionais para optimizar o sistema e satisfazer os requisitos estabelecidos. Neste contexto, uma das mais importantes etapas é a análise e implementação de algoritmos computacionais.

Enormes avanços tecnológicos no âmbito das FPGAs (*Field-Programmable Gate Arrays*) tornaram possível o desenvolvimento de sistemas de engenharia extremamente complexos. Contudo, o número de transístores disponíveis por *chip* está a crescer mais rapidamente do que a capacidade que temos para desenvolver sistemas que tirem proveito desse crescimento. Esta limitação já bem conhecida, antes de se revelar com FPGAs, já se verificava com ASICs (*Application-Specific Integrated Circuits*) e tem vindo a aumentar continuamente.

O desenvolvimento de sistemas com base em FPGAs de alta capacidade envolve uma grande variedade de ferramentas, incluindo métodos para a implementação eficiente de algoritmos computacionais. Esta tese pretende proporcionar uma contribuição nesta área, tirando partido da reutilização, do aumento do nível de abstracção e de especificações algorítmicas mais automatizadas e claras. Mais especificamente, é apresentado um estudo que foi levado a cabo no sentido de obter critérios relativos à implementação em *hardware* de algoritmos recursivos *versus* iterativos. Depois de serem apresentadas algumas das estratégias para implementar recursividade em *hardware* mais significativas, descreve-se, em pormenor, um conjunto de algoritmos para resolver problemas de pesquisa combinatória (considerados enquanto exemplos de aplicação). Versões recursivas e iterativas destes algoritmos foram implementados e testados em FPGA. Com base nos resultados obtidos, é feita uma cuidada análise comparativa.

Novas ferramentas e técnicas de investigação que foram desenvolvidas no âmbito desta tese são também discutidas e demonstradas.

keywords

FPGA, recursive algorithms, reconfigurable computing, design of computational systems

abstract

Design of computational systems is a complex multistage process which requires a deep analysis of the problem, taking into account relevant limitations and constraints as well as software/hardware co-design. Such task involves exploring competitive techniques and computational algorithms, enabling the system to be optimized while satisfying given requirements. In this context, one of the most important stages is analysis and implementation of computational algorithms.

Tremendous progress in the scope of FPGA (Field-Programmable Gate Array) technology has made it possible to design very complicated engineering systems. However, the number of available transistors grows faster than the ability to meaningfully design with them. This situation is a well known design productivity gap, which was inherited by FPGA from ASIC (Application-Specific Integrated Circuit) and which is increasing continuously.

Developing engineering systems on the basis of high capacity FPGAs involves a wide variety of design tools, including methods for efficient implementation of computational algorithms. The thesis is intended to provide a contribution in this area by aiming at reuse, high level abstraction, automation, and clearness of algorithmic specifications. More specifically, it presents research studies which have been carried out in order to obtain criteria regarding implementation of recursive vs. iterative algorithms in hardware. After describing some of the most relevant strategies for implementing recursion in hardware, a selection of algorithms for solving combinatorial search problems (considered as application examples) are also described in detail. Iterative and recursive versions of these algorithms have been implemented and tested in FPGA. Taking into consideration the results obtained, a careful comparative analysis is given.

New research-oriented tools and techniques for hardware design which have been developed in the scope of this thesis are also discussed and demonstrated.

Index of Contents

Index of Figures

Index of Tables

Glossary of Abbreviations

ASIC	Application-Specific Integrated Circuit
BV	Binary Vector
CAD	Computer-Aided Design
CLB	Configurable Logic Block
CNF	Conjunctive Normal Form
CPLD	Complex Programmable Logic Device
DABM	Dual Access Binary Matrix
DATM	Dual Access Ternary Matrix
DSP	Digital Signal Processor
EDIF	Electronic Design Interchange Format
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GCD	Greatest Common Divisor
GPPL	General-Purpose Programming Language
HDL	Hardware Description Language
HFSM	Hierarchical Finite State Machine
HT	Hardware Template
IP	Intellectual Property
ITRS	International Technology Roadmap for Semiconductors
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
PBM	Prototyping Board Manager
PC	Personal Computer
RAM	Random Access Memory
RHFSM	Recursive Hierarchical Finite State Machine

RTL	Register Transfer Level
SABM	Single Access Binary Matrix
SAT	Boolean Satisfiability
SATM	Single Access Ternary Matrix
SLSL	System-Level Specification Language
SOC	System-On-Chip
TCP	Transmission Control Protocol
TV	Ternary Vector
UML	Unified Modeling Language
USB	Universal Serial Bus
VEW	Virtual Execution Workbench
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large-Scale Integration

1. Introduction

1.1. Motivation

Design of computational systems is a complex multistage process which requires a deep analysis of the problem, taking into account relevant limitations and constraints as well as software/hardware co-design strategies. These factors are essential to achieve the required functionality while optimizing the most important system's characteristics (*e.g.* maximizing the performance or minimizing the needed hardware resources).

The system requirements are the constraints whose satisfaction is to be guaranteed. Typical constraints concern maximum respond time to different requests, maximum power consumption, etc. So long as the requirements are met, the computational system can be optimized in terms of complementary goals, such as minimizing the hardware resources, providing clearness of specifications, simplifying system maintenance, design reuse, opportunities for further updates and improvements, etc. Trade-offs between such system characteristics often take place, and determining the most appropriate choices involves exploring competitive techniques and computational algorithms, which is a process that can be seen as design space exploration. Analysis and implementation of computational algorithms is therefore a very important step to guarantee that the system functions in strong conformity with the given requirements and to achieve a good compromise between mutually-dependable system characteristics.

Algorithmic structure plays a very important role in the development of computational systems and it has direct relationship with important issues, such as: how well the algorithms are organized; how the algorithms are implemented; how clearly the algorithms are described; how different parts of the algorithms can be reused; how easily the algorithms can be modified and improved if required; etc. In case these features are carefully taken into account, it becomes possible to optimize algorithms,

to simplify their implementation, to shorten their development lead time, and to increase their effectiveness.

The thesis is dedicated to the problem of optimizing computational algorithms, and it explores and compares two known alternative ways to implement them, namely recursive and iterative. Additionally, important algorithmic features such as modularity, reusability, clearness, and verifiability are carefully studied.

1.1.1. General approach to hardware/software co-design

Hardware/software co-design of a computational system requires an answer to the following general question: Which parts of that system should be implemented in hardware and which parts of it should be implemented in software? In order to answer this question, it is necessary to consider multiple sub-questions, namely: What exactly is software and what exactly is hardware? For instance, software can be considered for general-purpose computers, for application-specific computers, for application-specific microcontrollers, for built-in 'hard/soft' cores such as the FPGA (Field-Programmable Gate Array) Power PC processor [EETimes02] which is built-in to FPGA or the Micro Blaze soft core [Xilinx], etc. Nevertheless, all these types of software have a number of common features, such as sequential processing of machine instructions, and implementation of fundamental concepts like procedure calls, interrupts, etc. Hardware can also be 'hard', like ASICs (Application-Specific Integrated Circuits), and 'soft', like FPGAs. Comparing with software, hardware is significantly more heterogeneous, and it is either difficult or even impossible to indicate a number of common features like for software. The main objective of this thesis is to explore hardware implementation of different algorithms. Due to complexity, not all of them can be realized entirely in hardware, urging software/hardware co-design to be employed. Exploring this topic also constitutes an objective of this thesis.

Special attention should be paid to reconfigurable computing. Indeed, the market for FPGAs and other programmable logic devices is expected to grow from $3.2 billion in 2005 to $6.7 billion in 2010, according to Gartner Dataquest [EETimes06a]. Figure 1.1 demonstrates the increasing reconfigurability (see percentage in vertical axis) of SOCs (Systems-On-Chip) from 2007 to 2022 [Roadmap07].

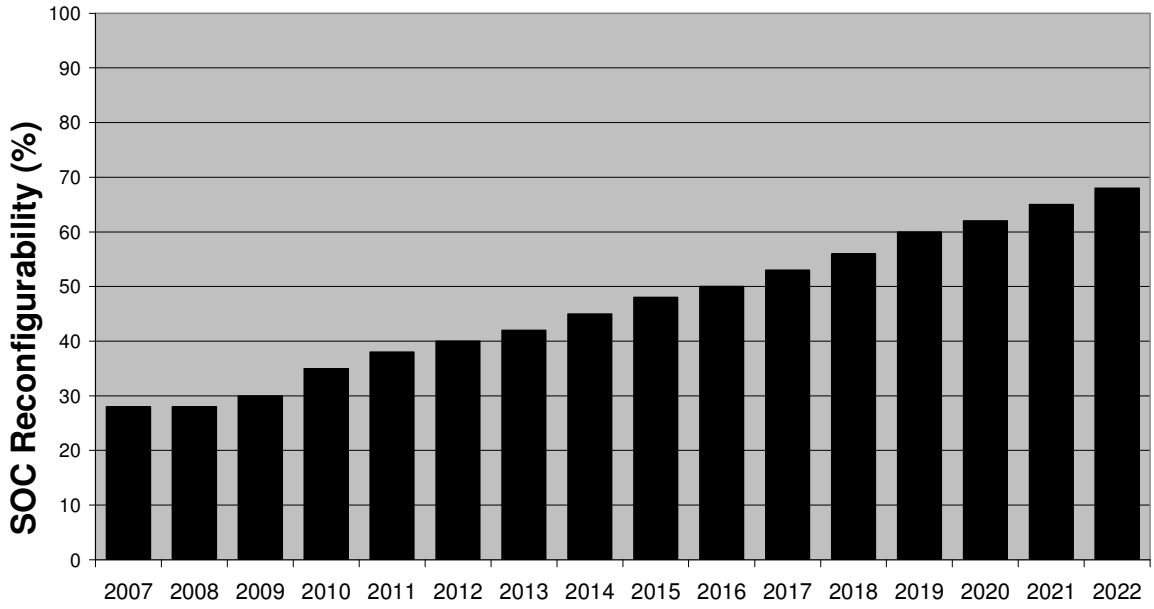


Figure 1.1 – SOC reconfigurability from 2007 to 2022

Such increase of SOC reconfigurability is expected because "the growing system complexity will make it impossible to ship designs without errors in the future. Hence, it is essential to be able to fix errors after fabrication" [Roadmap07]. These circumstances lead to extensive on-going research in digital circuit test and diagnosis [Ubar07, Jutman07], as well as in fault detection and fault tolerance strategies [Raik07, Ubar08]. Moreover, the increase of SOC reconfigurability is also due to the fact that "reconfigurability increases reuse, since existing devices can be reprogrammed to fulfill new tasks" [Roadmap07].

Since a forecast of importance of reconfigurable systems in general, and FPGAs in particular, is very promising for the future, this technology is assumed for implementation of computational algorithms in hardware within this work. Particularities of FPGA-based systems are considered in detail in the next subsection.

1.1.2. FPGA-based digital systems and reconfigurable computing

Tremendous progress in the scope of FPGA technology has made it possible to evolve configurable microchips from simple gate arrays that appeared on the market in the mid-1980s, to multi-platform FPGAs containing more than 10 million system gates and targeted to the design of very complicated engineering systems. Today, the way to evolve high performance computing from a general-purpose computer, proposed more than 50 years ago [Estrin60], has finally been implemented in reality. As mentioned in

the previous subsection, the market for FPGAs and other programmable logic devices is expected to grow from $3.2 billion in 2005 to $6.7 billion in 2010 [EETimes06a].

Developing engineering systems on the basis of high capacity FPGAs involves a large variety of design tools, including methods for efficient implementation of computational algorithms. The thesis is intended to provide significant contribution in this area.

An analysis presented in [D&R06] clearly demonstrates that the largest FPGA consumers will be in engineering, with numerous applications in the scope of electronic system design, from glue logic to high-complexity application-specific (ASIC-type) devices. Pioneering products such as Xilinx's Virtex or Altera's Stratix FPGA families will find their main applications in the development of high-volume products. Figure 1.2 demonstrates how FPGAs have been employed in different industries [Turley05]. Furthermore, Light Reading Inc.'s *Components Insider* conducted a worldwide survey in which 91 industry professionals participated, including equipment-manufacturing engineers, product developers and managers from more than 50 major equipment makers, and "90 percent of survey respondents said their company now uses FPGAs" [EETimes06c].

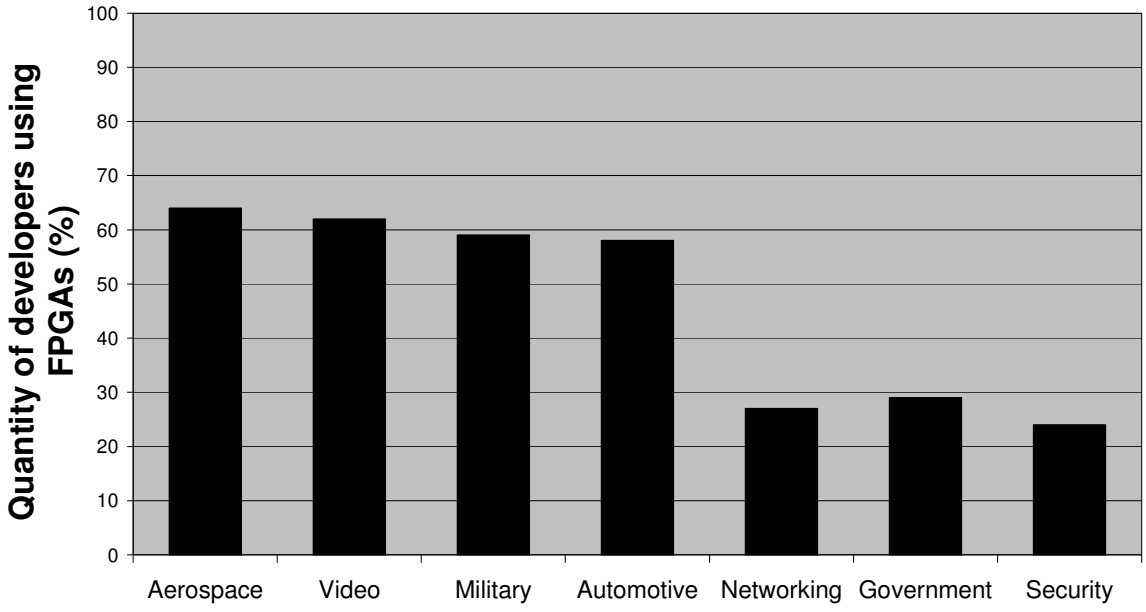


Figure 1.2 – FPGA usage in industry

In particular, FPGAs have been intensively used in the areas of mobile computing [Sridharanand05, Jung07] and multimedia. For example, Xylon company combines

Xilinx FPGAs of Spartan-3 family with the logicBRICKS IP (Intellectual Property) cores library [Kovacec05], allowing to quickly customize system designs running on generic FPGA development platforms into specialized multimedia products. Xilinx multimedia solutions provide the programmable hardware platforms, design tools, intellectual property, and reference designs which are needed to develop real-time video and image processing systems for a wide diversity of applications, such as video broadcasting and video conferencing, surveillance cameras, medical imaging, home gateway and digital TV [Newswire05].

The Xilinx Virtex-4 programmable technology enables the developers to rapidly implement state-of-the-art DSP (Digital Signal Processor) systems with high performance. Using FPGA-based reconfigurable processors for computation-intensive multimedia functions was considered in [Panainte04], reporting significant reduction in the number of clock cycles. Announced in 2006, Xilinx Virtex-5 FPGAs are a programmable alternative to custom ASIC technology and offer the best solution for addressing the needs of designers in the scope of high-performance logic, DSP, and embedded systems with unprecedented logic, hard/soft microprocessor, and connectivity capabilities [Xilinx06]. Virtex-5 microchips are built upon advanced 65nm triple-oxide technology with speed on average 30 percent higher and with capacity increased 65 percent over previous generation 90nm FPGAs.

The enormous potential of reconfigurable devices that recently appeared on the market for the design of complex systems can be seen from the example of the XC5VLX330 FPGA (Virtex-5 family) [Xilinx06]. This chip contains 25,920 configurable logic blocks (CLBs), 192 DSP slices, 10,368 Kb of block RAM (including 18 Kb and 36 Kb blocks), and 6 devices for advanced clock management. The plenary talk by Mike Butts in FPL'03 (the International Conference on Field-Programmable Logic and Applications, 2003), entitled 'Molecular Electronics: All chips will be reconfigurable', reports that future project densities are likely to be upwards of 100 billion devices per square centimeter and argues that cheap molecular-scale reconfigurable logic, memory, and interconnect are likely to become the predominant digital technology a decade hence. The advances and promising applications of reconfigurable systems given above clearly demonstrate future prospects of FPGA technology and its challenging capabilities for both industrial needs [Salcic06, Aimé07, Du07, Zhuang07] and research activity.

Announced in 2008, Xilinx Virtex-6 and Spartan-6 FPGAs can be seen as one more example demonstrating rapid progress in the scope of reconfigurable computing. Table 1.1 presents some of the characteristics of these two recent FPGA families [Xilinx09].

Table 1.1 – Details of Xilinx Virtex-6 and Spartan-6 FPGA families

Feature	Virtex-6	Spartan-6
Logic Cells	74,500 – 759,000	3,400 – 147,000
Distributed RAM (Kb)	1,045 – 8,280	32 – 1,358
Block RAM (Kb)	5,616 – 38,304	144 – 4,824
DSP Slices	288 – 2,016	4 – 182

Developing digital systems on the basis of high capacity FPGAs requires the extensive use of computer-aided design (CAD) tools. In fact, the electronic design automation business has profoundly influenced the integrated circuit business and vice versa, *e.g.* in the scope of design methodology, verification, libraries, and intellectual property [MacMillen00]. Traditionally, FPGA-targeted CAD systems support schematic and hardware description language-based design flows involving model-specific tools (such as those for synthesizing finite state machines (FSMs) from graphical specifications) and IP core generators based on parameterization or templates. Recently, commercial CAD tools which allow digital circuits to be synthesized from system-level specification languages (such as Handel-C and SystemC) as well as high-level programming languages (such as C) have appeared on the market. The domain of reconfigurable systems design turns out to be very dynamic and many-sided.

Designers of FPGA-based systems must wade through several layers of design before programming the actual device. The typical FPGA flow includes five major phases illustrated in Figure 1.3: design entry; synthesis; mapping, placement and routing; FPGA programming; and verification. The latter may occur at different levels, such as behavioral simulation, functional simulation, static timing analysis, post-layout timing simulation and, finally, in-circuit verification. If we focus our attention on the design entry, four different specification methods can be envisioned: schematic entry, hardware description languages, system-level specification languages (SLSLs) and, finally, general-purpose programming languages (GPPLs) [Sklyarov07b].

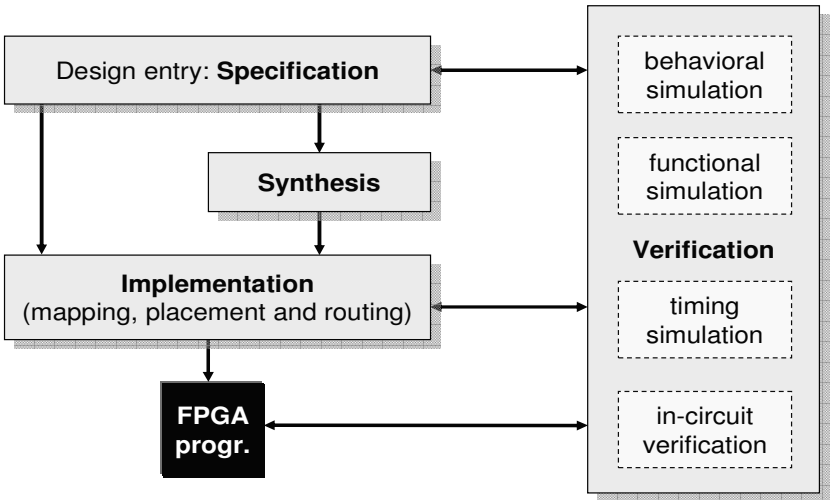


Figure 1.3 - Typical FPGA design flow

The schematic-based approach is nowadays not very appropriate for specifying the functionality of modern systems because, instead of thinking in terms of algorithms and data structures, it forces the designer to deal directly with the hardware components and their interconnections. Contrariwise, the hardware description languages (HDLs), such as VHDL and Verilog, are widely used for design specification since they typically include means for describing structure and functionality at a number of levels, from the most abstract algorithmic level, down to the gate level.

Recently, commercial tools for synthesizing digital circuits from system-level specification languages, such as Handel-C and SystemC, have appeared on the market. In this area, C and C++, with application-specific class libraries and with the addition of inherent parallelism, are emerging as the dominant languages in which system descriptions are provided. This fact allows the designer to work at a very high level of abstraction, virtually without worrying about how the underlying computations are executed. Consequently, even computer engineers with a limited knowledge of the targeted FPGA architecture are capable of rapidly producing functional, algorithmically-optimized designs.

An even higher level of abstraction is achieved with general-purpose programming languages, such as C or Java. During the last years, commercial tools (*e.g.* Catapult Synthesis from Mentor Graphics and CoDeveloper from Impulse) started appearing on the market, allowing the respective high-level descriptions to be automatically converted to HDL descriptions, which are then used for synthesis. In this case, the code portions that can be executed in parallel are automatically identified by the design tools. In addition to the design specification methods mentioned, there are

other available tools, such as vendor libraries, graphical finite state machine editors, parameterizable IP cores, and so on.

In the graph of Figure 1.4 [Sklyarov07b], different design specification methods are assessed according to performance, FPGA resource usage, portability, ease to learn, ease to change and maintenance, and development time (for the first five groups of vertical bars: the higher, the better; for the last group: the lower, the better).

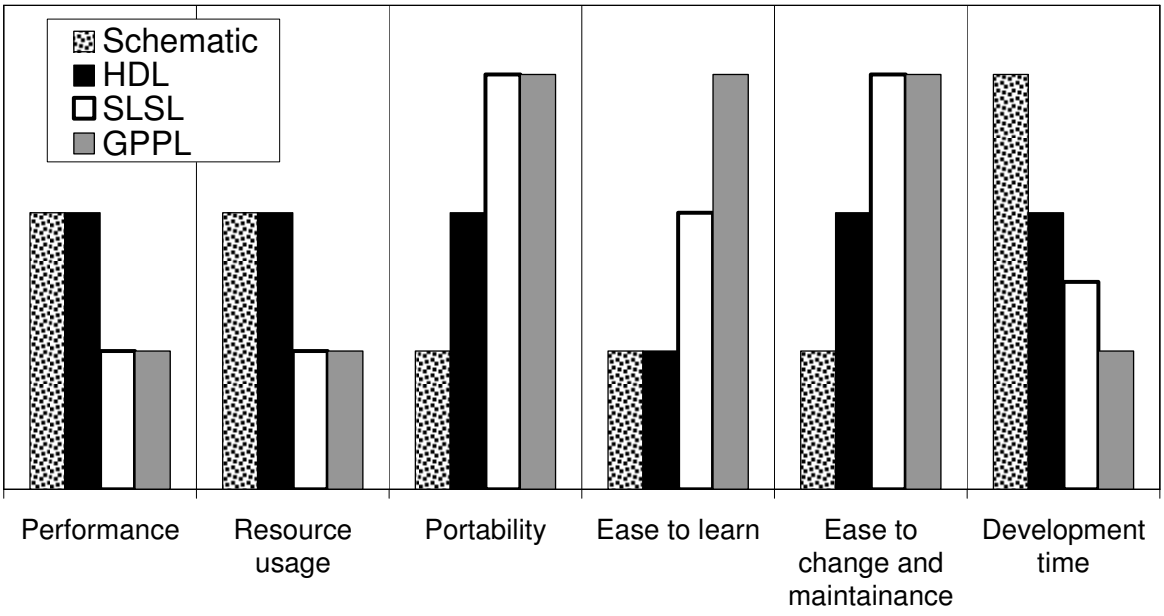


Figure 1.4 - Comparison of specification methods (from [Sklyarov07b])

From the graph, we can see that the schematic-based approach leads to circuits with very good performance and efficient resource usage. However, when we consider portability and ease to learn, change and maintenance, and the associated development time, schematic entry is an obvious outsider. As mentioned in an Electrical Engineering Times survey, "the days of designing FPGA with schematics are gone" [EETimes06b].

Hardware description languages are currently the golden mean of the design entry methods [Sklyarov07b]. They allow creating high-performance circuits that are optimized from the resource usage point of view, the associated development time is not too long, and design changes are not so difficult. The only weak point is that it's not very easy to learn HDLs.

System-level and high-level languages possess the highest portability and the highest level of abstraction. Of course, the higher level of abstraction leads to some

performance degradation and not very efficient resource usage. On the other hand, SLSLs and GPPLs have important advantages such as ease to learn, ease of change and maintenance, and a very short development time. We can therefore expect that, as the tools responsible for generating hardware from high-level source code advance, the SLSLs and GPPLs may become the predominant hardware description methodology, in the same way as general-purpose high-level programming languages have already supplanted microprocessor assembly languages [Skliarova06a]. Due to such advantages, system-level specification languages and the relevant synthesis tools are considered within this work to be basic instruments for comparing alternative recursive and iterative implementations of computational algorithms.

According to Moore's law [Moore65], chip complexity grows exponentially with time. But more important is that the number of available transistors grows faster than the ability to meaningfully design with them. This situation is a well known design productivity gap, which was inherited by FPGA from ASIC and which is increasing continuously. Therefore, the design productivity will be the real challenge for future systems. It is believed that platform FPGAs could alleviate this problem since they offer the flexibility, time-to-market, and the bandwidth requirements to rapidly bring electronic systems to market. With such highly programmable platforms that include one or more programmable processors and/or reconfigurable logic, derivative designs may be created without fabricating a new system-on-chip (SOC) [Roadmap05]. Platform customization for a particular SOC derivative then becomes a constrained form of design space exploration: the basic communications architecture and platform processor choices are fixed, and the design team is restricted to choosing certain customization parameters and optional IPs from a library [Roadmap05].

In order to increase the design productivity, three important strategic directions must be followed. First of all, design reuse must be encouraged. Reusable, high-level functional blocks (such as IP blocks) offer great potential for productivity gains because design effort for the reused logic is only a portion of the effort needed for newly designed logic. According to International Technology Roadmap for Semiconductors (ITRS), reuse rate for system-level design will increase from 35% in 2007, to 58% in 2022 [Roadmap07].

The second strategic line concerns design abstraction levels, which must be raised. Higher levels of abstraction allow many forms of verification to be performed much earlier in the design process, reducing time-to-market and lowering cost by

discovering problems earlier [Roadmap05]. As previously mentioned, tools which allow for hardware design at a very high level of abstraction are currently emerging.

And last, the third strategic direction is to increase the level of automation and clearness of algorithmic specification, which will inevitably allow the number of design iterations to be reduced. In case of platform-based design, further improvements in automated software/hardware partitioning tools are strongly required.

It is now clear that reconfigurability will certainly be the key aspect of future systems, since it will be required for fault tolerance, *e.g.* for molecular-scale systems, and for development of adaptive and self-correcting or self-repairing circuits. In addition, reconfigurability increases reuse, since existing devices can be reprogrammed to fulfill new tasks. According to what ITRS estimates (see Figure 1.1), more and more SOC functionality will become reconfigurable [Roadmap07].

Another important aspect of SOC design is the exploration of efficient methods for implementation of computational algorithms which allow for clearness of specification, reuse and effectiveness of future implementation. Contributing to this topic is the primary target of the thesis. The following section discusses widely used ways of implementing computational algorithms.

1.1.3. Recursive implementation of computational algorithms

It is known that recursion is an extremely powerful problem-solving technique [Carrano95] that permits a problem to be decomposed into smaller sub-problems that are of exactly the same form as the original problem.

Many examples that demonstrate advantages of recursion are presented in [Kernighan88, Carrano95, Maruyama99, Sklyarov04]. However this technique is not always appropriate, particularly when a clear efficient iterative solution exists [Carrano95, Sklyarov04]. This fact is primarily due to the large amount of states that are accumulated during deep recursive calls. Besides, in most high-level programming languages, a function call incurs a bookkeeping overhead. Recursive functions magnify this overhead because a single initial call to the function might generate a large number of recursive invocations of the function.

The paper [Sklyarov04] provides significant contribution to solve this problem and proves that recursion can be implemented in hardware more efficiently than in

software. This achievement resulted from combining any activation of a recursive subsequence of operations with the execution of the operations that are required by the respective algorithm. The same combination takes place when any recursive sub-sequence is being terminated, *i.e.* when control has to be returned to the point after the last recursive call, and the following operation of the executing algorithm has to be activated.

The number of states that are required for the execution of recursion in hardware can be made smaller than in software, but it is still greater than for iterative solutions. However, codes for such states are accumulated on stacks that are typically implemented on built-in memory blocks, which are very regular and relatively cheap. The results obtained for some known methods for implementing recursive calls in hardware, such as a technique based on multi-thread and speculative execution [Maruyama99], have shown that hardware circuits can be faster than software programs running on general-purpose computers, with respect to this matter. Moreover, it is known that a recursive algorithm can be implemented in hardware with the aid of a hierarchical finite state machine (HFSM) [Sklyarov84, Sklyarov99] and this strategy is explored in this thesis.

Note that recursive algorithms have a wide scope of practical applications (see, for example, [Sklyarov04, Carrano95, Maruyama99]). However, they are most often employed for various kinds of binary search and this is a notable exception, even when implemented in software [Carrano95], because the recursive solutions are quite efficient in this area. There are many examples of recursive binary search and we will briefly discuss just a few of them.

Let us consider a binary tree whose nodes contain four fields, which are: a pointer to the left child node, a pointer to the right child node, a counter, and a value (let us say an integer or a pointer to a string). The nodes are maintained so that, at any considered node, the left sub-tree contains only values that are smaller than the one at the considered node, and the right sub-tree contains only values that are bigger than that. The counter indicates the number of occurrences of the value associated with the respective node. It is known that such a tree can be constructed and used for sorting various types of data [Kernighan88]. In order to build such a tree for a given set of values, we have to find the appropriate place for each incoming node in the current tree. In order to sort the data, we can apply a special technique [Kernighan88] using forward and backtracking propagation steps that are exactly the

same for each node. Thus a recursive procedure is very efficient. Sorting of this type was considered in [Sklyarov04] as a working example.

Other useful applications can be encountered in the area of lossless data compression [Sklyarov04]. Many techniques have been proposed in this context, such as Huffman coding, arithmetic coding, run-length coding, and Lempel-Ziv compression algorithms (see, for example, the Internet site [EFF], which collects many useful publications, methods, and software tools). They combine components for modeling (classified by statistical methods and dictionary methods [Nunez03]) and coding.

Recursive algorithms are quite efficient for such applications and we will show two examples taken from [Sklyarov04]. Huffman coding requires a sequential invocation of two procedures: data sorting, and incremental construction of a Huffman binary tree [Rosen00]. The latter contains information about Huffman codes with different lengths. We have already mentioned that recursive algorithms can be efficiently employed for data sorting. However, they can also be used for constructing a Huffman tree. Moreover, these two procedures can be combined in a single recursive procedure. Dictionary methods often require a content-addressable memory, which is resource-consuming [Nunez03]. On the other hand, searching in dictionaries can be performed using recursive methods that are employed for software applications [Carrano95]. Thus the considered technique can be helpful. This is especially important today because many data compression algorithms need to be implemented in hardware, in general, and in reconfigurable hardware (such as FPGA), in particular [Nunez03]. One potential example of applying recursive algorithms for Huffman coding was examined in [Sklyarov04].

Another important application area that can be addressed is in the scope of combinatorial optimization [Sklyarov04, Skliarova04a, Skliarova08]. Combinatorial search algorithms that are widely used in this area have two distinctive features. Firstly, as a rule they require a huge number of different feasible solutions to be considered. Secondly, these feasible solutions can be ordered and examined with the aid of a search tree that provides an efficient way for handling intermediate solutions. The search tree is constructed during the search process and it is traversed starting from the root. Typically, this is an N-ary tree [Rosen00] with $N \geq 2$. Note that a recursive search can also be efficiently applied to N-ary trees and this has been demonstrated in [Sklyarov03b] on an example of discovering a minimal column cover of a binary matrix. A similar approach can be used for solving many other combinatorial problems, such as Boolean satisfiability, graph coloring, etc. Two

examples from this scope which can make use of recursive calls (namely the knapsack and the knight's tour problems) were discussed in [Maruyama99].

Let us note that many combinatorial algorithms deal with a huge amount of data which have to be transferred between a host computer and a hardware accelerator [Skliarova04b]. In many circumstances, due to the complexity, the problem cannot be completely solved in hardware, and combined hardware/software solutions are therefore employed. This is a typical way of hardware/software co-design and it involves multiple time consuming data transfers. Thus recursion can be employed [Sklyarov04] on the one hand for the data compression/decompression operations mentioned above (enabling the amount of data and consequently the data transfer time to be significantly reduced), and on the other hand for the combinatorial algorithms themselves, allowing more efficient solutions for tree search problems to be provided (see, for instance, some assessments in [Maruyama99, Sklyarov03b]).

As already mentioned, FPGA-based systems are going to be used for implementation and evaluation of the considered computational algorithms. Thus, it is necessary to analyze the basic distinctive features of FPGA-based systems and to take advantage of them. The relevant features of such systems are the following:

- Can be seen as 'soft' ASICs;
- Introduce a new computing paradigm;
- Eliminate the necessity for the von Neumann architecture although such architecture can be used if required;
- Enable the designers to implement algorithms directly in silicon;
- Make parallelism a key feature;
- Permit any required interface with external devices to be established.

1.2. Design prototyping

There are a number of available prototyping boards that support various experiments with FPGA-based circuits [Xilinx, Celoxica, Trenz]. These boards permit to implement digital systems in FPGAs and to provide for an interaction of these systems with both onboard microchips and external devices (such as static RAM and micro controllers), which might be connected through expansion headers. The use of such boards

significantly simplifies the design of new FPGA-based systems and allows the development lead time to be shortened.

Prototyping boards are widely employed in engineering practice, in research activity, and in education. When choosing an FPGA-based prototyping board, it is necessary to find a compromise between the required hardware/software resources and the price; but with the large number of available boards, it becomes difficult to make the best choice for a particular application. Taking into account that the majority of prototyping boards include many typical components (memories, LCDs, standard interfaces, etc.), it is very difficult to find a board that contains only elements that are required and nothing else, which only occupies the space and increases the cost. Moreover, it is necessary to develop software targeted to the desired experiments, taking into account numerous particularities of the developed algorithms. However, it is either difficult or even impossible to satisfy all the requirements mentioned above due to unavailability of detailed technical documentation and hardware support projects implemented by relevant manufacturers. As a rule, such materials are not supplied.

Thus, an extendable set of hardware/software tools have been proposed. Hardware tools have been developed in [Almeida06, Almeida08], and software tools have been designed and explored within the scope of this research. It is important that any particular problem can be solved using only the subset of hardware/software components that are required (from the considered extendable set), excluding all the other available components. In case the desired components are not available, they can easily be constructed and integrated/attached.

In general, the suggested tools have to provide prioritized support for the following distinctive functionality:

- Configuration of the core FPGA using wired (USB) and wireless (Bluetooth) interfaces, the latter making the prototyping system ideal for remote applications;
- Dynamic onboard reconfiguration and remote wireless reconfiguration and/or interaction;
- Implementation and comparison of recursive and iterative algorithms in hardware and software/hardware partitioning;
- Versatile, efficient, user-friendly workflows (by integration with other CAD tools) for system design on the basis of hardware description languages (VHDL in

particular), system-level specification languages (Handel-C in particular), templates, design libraries and IP cores.

1.3. Main objectives

The three main objectives of this research are the following:

1. Implementation of recursive algorithms in reconfigurable hardware and comparison of recursive and iterative implementations. Analysis of the design space where recursive/iterative algorithms are more advantageous taking into account the design objectives and target requirements;

2. Exploration of a reuse technique, in hardware design, on the basis of parameterizable, reprogrammable architecture and generic IP modules;

3. Development of software tools for hardware/software co-design and co-simulation of FPGA-based reconfigurable prototyping systems.

In order to pursue the first main objective, it is necessary to address the following tasks:

- Answering the question: How to implement recursion in hardware? Note that known hardware and system-level specification languages do not provide support for implementing recursion;
- Exploring hardware architectures enabling recursive algorithms to be implemented in hardware;
- Designing system components, such as IP modules, which support the development of hardware from recursive specifications;
- Considering particular design examples allowing to compare alternative recursive and iterative algorithms;
- Experiments and comparisons of recursive and iterative algorithms. Determining design space for recursive and iterative algorithms.

To satisfy the second objective it is necessary to address the following tasks:

- Explore the relationship between recursion and modularity in hardware design (indeed, recursion assumes modularity);

- Explore and compare different opportunities for the design of reusable modules;
- Provide a set of experiments and recommendations;
- Analyze a relationship between modularity and dynamic reconfigurability.

To attain the third objective, it is necessary to carry out the following set of tasks:

- Analyze potential ways to explore such type of software/hardware co-design and co-simulation, which enable designers to easily explore digital systems with either more software and less hardware or vice versa;
- Suggest an FPGA-based prototyping system suitable for such purposes;
- Develop software oriented to the comparison and implementation of alternative FPGA-based accelerators;
- Provide a set of experiments based on the developed methods and tools.

1.4. Thesis structure

This thesis is organized in seven chapters. Chapter 2 starts with describing background concepts which are essential for understanding the remainder of the thesis (namely recursion, combinatorial problems, and backtracking search algorithms) and then presents the state of the art relevant to the thesis area, addressing known results on the comparison of recursive and iterative algorithms, and strategies for implementing recursion in hardware. The last section of chapter 2 summarizes the main aspects of the background and strategies considered.

Chapter 3 analyzes computationally intensive problems which are taken mainly from the scope of combinatorial search. The latter is relevant because both targeted techniques, *i.e.* recursive and iterative, can rationally be applied. For each of six selected problems, the following is provided: problem description, application domains, an algorithm for solving it, and a detailed illustration in which the given algorithm is applied to solve the problem on the basis of a practical example. Four particular problems are solved with the aid of backtracking search algorithms, namely: set covering, Boolean satisfiability, graph coloring, and knapsack. Two supplementary problems (tree-based data sorting and discovering of a greatest common divisor) are also studied. A generic approach to backtracking search algorithms is described and discussed.

Chapter 4 describes the developed prototyping system and software tools that enable experiments with hardware accelerators and comparisons of alternative recursive and iterative algorithms to be carried out easier and more efficiently. The system is based on the DETIUA-S3 prototyping board, featuring wired and wireless interfaces with a host computer, and on software tools proposed and implemented in the scope of this thesis. These software tools provide user-friendly interface with the board (including wireless interaction) and high-level support for many different experiments which are required for the hardware accelerators considered. Virtual peripheral devices, modules for software/hardware co-simulation, and procedures for extracting intermediate results for analysis are examples of the software tools developed. A more advanced technique assumes the application of the developed tools through the Internet in such a way that allows different users to configure and to interact with the remotely accessed prototyping board. Although this work was not initially planned, many tools have been developed, implemented and tested, permitting to conclude that the proposed system can efficiently be used for remote interactions.

Chapter 5 provides details of reconfigurable hardware implementation of iterative and recursive algorithms for the selected problems. Every algorithm was first modeled in software in order to simplify the design process, and the respective object-oriented classes and activity diagrams are presented. After modeling, some of the algorithms were described in a system-level specification language (Handel-C) and a hardware description language (VHDL). The respective specifications were finally synthesized and implemented in commercially available FPGAs and carefully analyzed.

Chapter 6 presents the details and results of the various sets of experiments which were carried out, followed by careful analyses. The first set of experiments addresses the comparison of iterative and recursive implementations in hardware. Results are not only shown in tables with the relevant numerical results but put into perspective by means of graphical charts, allowing for an easier analysis. Relevant remarks beyond the observation of the results are made in order to complement the comparison. The second subsection of this chapter describes the validation and analysis of the architecture for generic matrix-oriented solvers, providing an overview of the key data structure and functional block usage amongst different matrix-based backtracking search algorithms. The third section provides an assessment of the prototyping tools which were developed in the scope of this thesis and a summary of relevant potential applications.

Chapter 7 summarizes the author's contribution, lists the most important results and suggests future work in the considered area.

2. Background and State of the Art

This chapter is composed of the following three sections: background (section 2.1) which describes recursion, combinatorial problems, and backtracking search algorithms; state of the art relevant to the thesis area (section 2.2), presenting known results in comparison of recursive and iterative algorithms as well as strategies for implementing recursion in hardware; and finally a conclusion (section 2.3).

2.1. Background

2.1.1. Recursion

Something is said to be *recursive* if it partially consists or is defined in terms of itself [Wirth86]. Recursion can be applied and observed in many fields and, in problem solving, it is known to be an extremely powerful technique [Carrano95] which permits to decompose a problem into smaller sub-problems that are of the same form as the original problem [Sklyarov04].

Within the context of algorithm implementation, recursion is mainly used in the definition of procedures (see Figure 2.1-a) and structured data types (Figure 2.1-b). The thesis focuses on the procedure-oriented kind of recursion but, in fact, recursive algorithms are particularly appropriate when the data to be processed and the problem to be solved are defined in recursive terms [Wirth86].

a)

```
procedure(...)
{
   ...              // some eventual operations
   procedure(...)   // self invocation
   ...              // other eventual operations
}
```

b)

```
datatype is composed of
{
   ...                // some eventual data fields
   datatype field_n   // field of the type being defined
   ...                // other eventual data fields
}
```

Figure 2.1 - Recursive definitions for procedures (a) and data types (b)

Recursion can be *direct* and *indirect*. A procedure that includes an explicit invocation of itself is said to be *directly recursive* (see Figure 2.1-a). On the other hand, an *indirectly recursive* procedure is one that invokes some other procedure which directly or indirectly invokes the first one. For example, most recursive algorithms developed for solving combinatorial problems are directly recursive. Nonetheless, both kinds of recursion present essentially the same implementation challenges. It should also be noted that, although recursion is often very useful, it has been proven that any recursive algorithm can be re-expressed non-recursively [Kruse87].

In order to illustrate the applicability and advantages of recursive algorithms when they are implemented in hardware for solving computationally intensive problems, we will explore combinatorial search problems as an example. The subsequent two sections present general characteristics of such problems and a technique widely used to solve them.

2.1.2. Combinatorial problems

Combinatorics is a branch of mathematics with increasing importance which can be described as the study of how discrete sets of objects can be arranged, counted, and constructed, according to specified constraints [Cameron94, Erickson96].

Combinatorial search problems are divided in four types, depending on the kind of solution that is required [Kreher99, Skliarova04a]:

- *Decision problems*, in which a question is to be answered 'yes' or 'no';
- *Search problems*, in which a question is to be answered 'yes' or 'no' and, in case the answer is 'yes', an *n*-tuple $[x_1,...,x_n]$ that verifies the given constraints is to be provided;
- *Enumeration problems*, in which the number of different *n*-tuples $[x_1,...,x_n]$ that verify the given constraints is to be found;
- *Optimization problems*, in which an *n*-tuple $[x_1,...,x_n] \in \{0,1\}^n$ which maximizes the value of a specified profit-evaluating function (or minimizes the value of a specified cost-evaluating function) is to be provided.

A significant characteristic of combinatorial problems is their vast applicability, which is also the reason for their increasing importance. Algorithms for solving such problems are therefore getting a lot of attention today [Zakrevskij08]. Applications of combinatorial problems can be found in Boolean expression simplification [Breuer70]; resource allocation [Rubin73, Walker74, Gleeson94, Rodin90, Bodin91, Henig90]; mathematical logic, artificial intelligence, VLSI engineering, and computing theory [Gu97]; automated reasoning, computer-aided design, computer-aided manufacturing, machine vision, database, robotics, integrated circuit design automation, computer architecture design, embedded systems, and computer network design [Gu97, Goossens97, Subramonian04]; microprogramming for application-specific embedded microprocessors and resource distribution [Culberson, Wu93]; cutting stock systems [Gilmore61, Hahn68, Madsen79, Seth87]; cryptography [Merkle78, Chor88, Jan93], broadband communications [Ross89, Gavious94], etc. More detailed application examples of particular combinatorial problems will be considered in chapter 3, when describing the problems which will be used in the scope of this thesis.

2.1.3. Backtracking search algorithms

Most algorithms for solving combinatorial problems have a top-down approach based on search trees [Zhang89]. Search trees are typically implemented by means of a *backtracking* mechanism [Golomb65, Floyd67, Bitner75, Cohen79, Skliarova04a]. In this context, the search consists of a multi-stage decision process in which some choice is made at each stage [Helsgaun95]. At every stage, a solvability test which

takes the earlier choices into account is also performed and, under certain circumstances, such test can determine that some of those choices cannot lead to a solution. If this is the case, the algorithm restores the context belonging to the previous stage, *i.e.* it *backtracks*, in order to make an alternative choice. If all alternative choices based on that context have already been tried, the algorithm backtracks again. The process continues until the whole search tree is traversed or, in case the given problem is not an optimization problem, when a satisfactory solution is found. In either case, algorithms which follow this general strategy are called *backtracking search algorithms*.

The eight queens problem [Ball60] is a classic combinatorial problem that is very appropriate for illustrating backtracking search algorithms. Let us consider a simplified version which consists of finding a way to place 4 queens in a 4 by 4 chessboard in such a way that no queen is able to attack another. In this problem, there is no distinction between white and black queens. Thus, in order to achieve a solution, no pair of queens can be placed in the same row, column, or diagonal.

In order to solve this problem, Bitner and Reingold chose the following strategy [Bitner75]: Because exactly one queen must be placed in each column, a solution can be represented as a tuple $[x_1, x_2, x_3, x_4]$ in which x_i represents the row of the queen placed in the ith column. They do not consider all possible combinations of queen placements; only those with one queen in each column. This way, all combinations with more than a queen per column (which are obviously not solutions) are excluded from the beginning.

Using this strategy, one can conceive a search process with 4 stages in which the value of x_i is chosen at stage i. Each choice is made amongst 4 possible values: 1 to 4 (which identify the 4 rows). As a result, a quaternary search tree with a depth of 4 levels is obtained. Figure 2.2 depicts the part of that search tree which is actually traversed by a backtracking search algorithm that would follow this approach. At the root of the search tree, the 4-tuple variable which will provide the solution is completely unassigned. At each level, one of its elements is assigned and hence the variable becomes completely assigned when a leaf is reached. White circles correspond to legal partial queen placements which are therefore explored further. On the other hand, grey circles illustrate partial or complete queen placements which are illegal and therefore trigger a backtrack movement. The black circle is the leaf of the search tree in which a solution is found (see Figure 2.2-t).

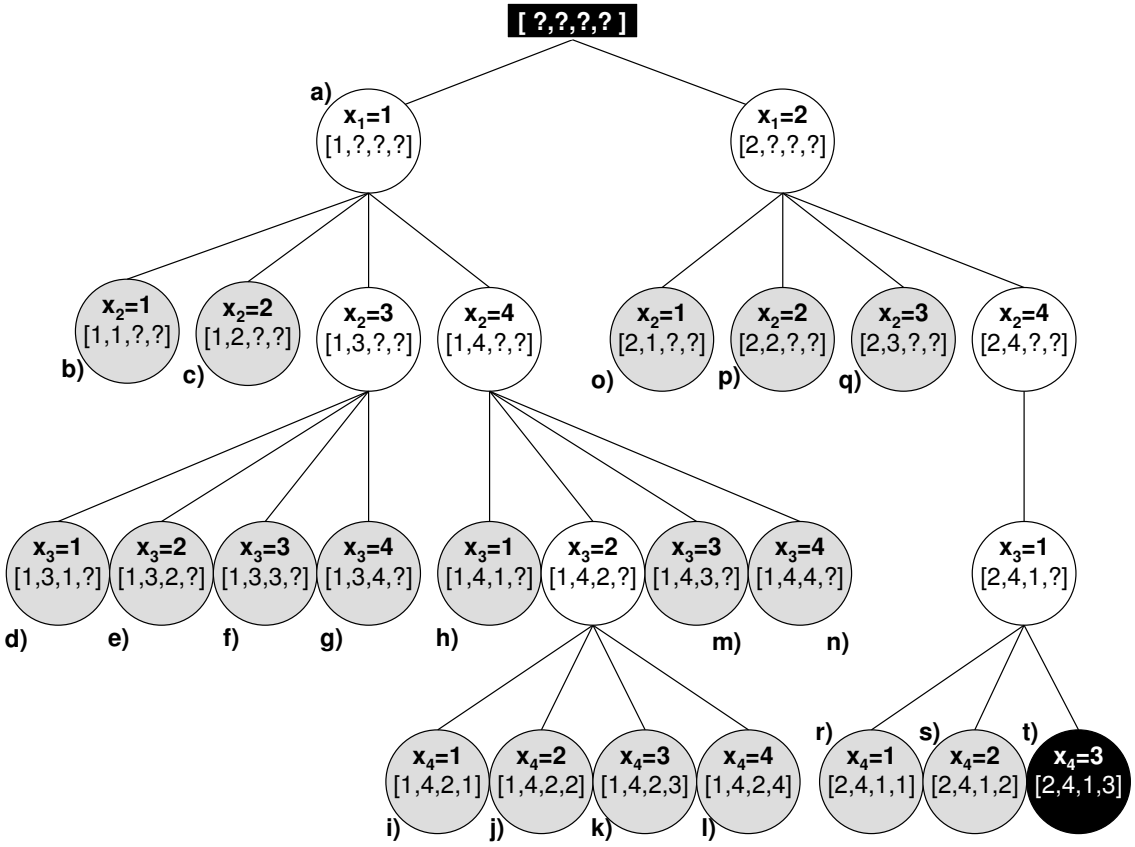


Figure 2.2 – Traversed part of the search tree for solving the four queens problem

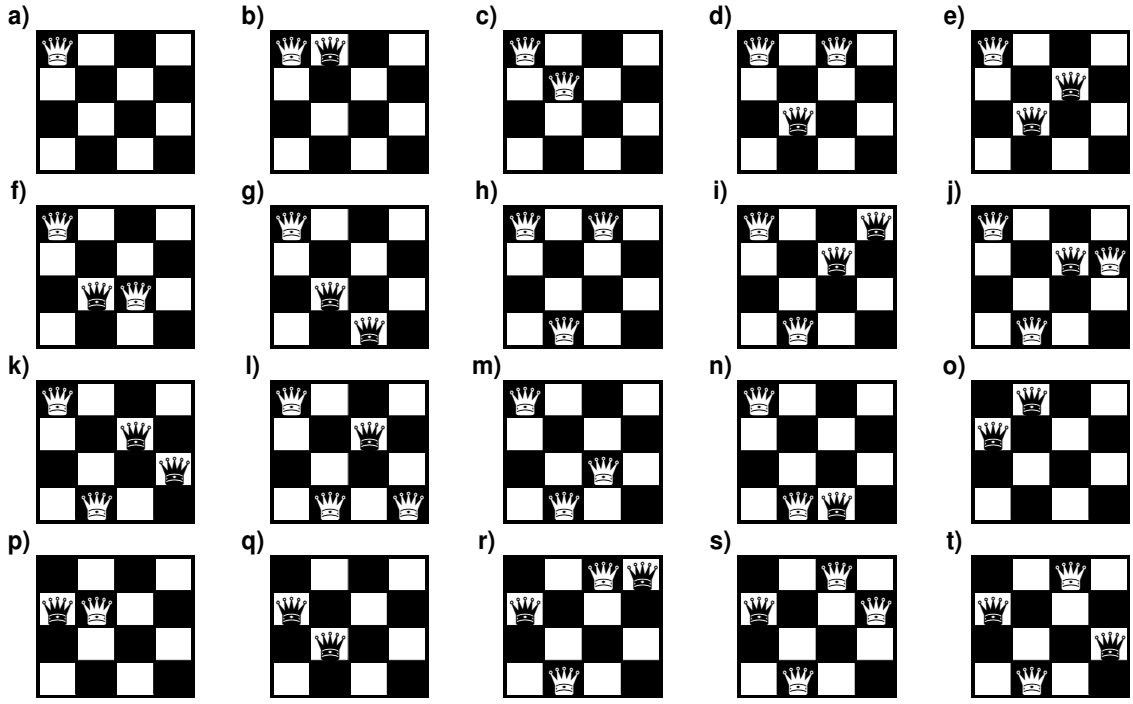


Figure 2.3 - Queen placements represented by the 4-tuples in Figure 2.2

In order to better understand the traversed search tree and the reason for which some of its nodes correspond to illegal placements, the queen placements that are represented by the 4-tuples in Figure 2.2-a to Figure 2.2-t are depicted in Figure 2.3-a to Figure 2.3-t, respectively.

Recursion is often used to support backtracking. At any branch point of the search process, *i.e.* any non-leaf node of the search tree, choosing one of the alternative search branches corresponds to making a recursive call, whereas backtracking corresponds to returning from a recursive call. When backtracking, all the variables which define the search context must be re-assigned in order to restore the values they had when the corresponding recursive call was made.

2.2. State of the art

2.2.1. Comparison of recursive and iterative algorithms

Comparing recursive and iterative algorithms can be carried out with two perspectives [Pimentel09]. From the point of view of the designer, there are pros and cons concerning the design process, such as design time, ease of modification, etc. These characteristics are generally independent from implementation issues (like programming/description language or computational platform) because they are related only to how the algorithm is described at a high-level of abstraction. These *design-based comparison criteria* are often considered subjective, as there are currently no known methods for evaluating them objectively. However, despite their subjectiveness, these criteria can be of great relevance to designers.

On the other hand, one can compare characteristics of the resulting solution, such as execution time, area/memory usage, etc. In this perspective, comparison results are very dependent on the implementation issues. Moreover, available CAD tools and the measurability of the solution properties grant objectiveness to these *solution-based comparison criteria*.

Within the software applications domain, recursive and iterative algorithms have been subject to comparison for a long time and therefore concrete comparison results in this area are already well-known. On the one hand, it is widely accepted that, for certain classes of algorithms, recursion provides clean, concise, elegant, and robust designs that are easy to conceive, understand, and modify with minimal design costs

(namely design time). On the other hand, recursive algorithms in software are generally considered slow and very memory-consuming [Ninos08] when compared to iterative ones. Although some authors believe that inappropriate examples are sometimes used to reinforce such disadvantages [Noble03], the latter are widely accepted. As a consequence, the use of recursion in software is quite often avoided, even when implementing algorithms that are inherently recursive. In fact, methods for transforming general recursion into iteration have been extensively studied [Arsac82, Backus85, Partsch90, Harrison92, Kfoury97, Liu99, Tang06].

Despite this widely accepted heuristic indicating that recursion is generally less advantageous than iteration, in software, the suitability of recursion (versus iteration) has been found highly dependent on the class of the implemented algorithm. When applying a divide-and-conquer approach, the original problem is replaced with similar smaller problems. With this approach, recursion is known to be most efficient [Noble03] and therefore arguably advantageous when compared to iteration.

Implementing recursion in hardware deals with platform features and limitations which are different from those dealt with in software. For instance, general purpose computers generally offer wide allocable memory space (which is of great use for keeping ever-changing size stacks) but, on the other hand, do not support parallel execution (which speeds up the completion of sets of independent operation sequences). However, the opposite scenario unfolds for hardware implementations. This means that the pros and cons of using recursion (versus iteration) in hardware applications can be quite different from the results known in software applications.

However, strategies for implementing recursion in hardware [Maruyama99, Maruyama00, Sklyarov99, Ferizis06, Ninos08] have started to be proposed only in 1999 and therefore very few results are available for comparison. Furthermore, these strategies implement recursion in different ways, which means they may lead to different iteration-versus-recursion comparison results.

2.2.2. Strategies for implementing recursion in hardware

In software, recursion has a standard support already provided by programming language compilers, which implement it transparently on the basis of procedure calls which make use of stacks. In contrast, different strategies for hardware implementation of recursion are still being proposed and discussed. Let us have an overview of some important proposals regarding this topic.

2.2.2.1. Maruyama, Takagi, and Hoshino

Tsutomu Maruyama, Masaaki Takagi, and Tsutomu Hoshino have proposed the following two techniques for implementing recursion in hardware: *multi-thread execution* and *speculative execution* [Maruyama99]. Both of them are aimed at the implementation and optimization of recursion in backtracking search algorithms on the basis of pipelining and with the use of a logic stack.

Analogously to Bondalapati and Prasanna's proposal on mapping loops onto reconfigurable architectures [Bondalapati98] (optimized later in [Bondalapati00]), each of the pipeline stages is activated for a different recursive call and therefore all stages are activated simultaneously, and idle cycles are avoided.

While *multi-thread execution* is more appropriate for algorithms which require traversing the whole search tree (searching for the optimal solution), *speculative execution* is better suited for finding any solution (the first that is found).

The research [Maruyama99] has shown that multi-threaded execution of recursion calls leads to higher performance than simple sequential execution with negligible hardware resource usage and clock frequency overheads.

Later on, Maruyama and Hoshino have developed a compiler for generating pipeline circuits on the basis of loops and recursive programs written in the C programming language [Maruyama00]. Stacks are implemented using FPGA internal memory blocks, when available.

Known limitations regarding these authors' proposals concern the following issues:

a) the maximum speedup equaling the pipeline's depth [Maruyama99, Ninos08];

b) the speed increase obtained at the expense of area utilization [Ninos08];

c) the efficiency when handling recursive functions that call themselves multiple times [Ferizis06].

2.2.2.2. Sklyarov

Having the objective of developing an approach to the design of virtual control devices that would provide the properties of extensibility, flexibility, and reusability, Valery

Sklyarov has proposed a technique for synthesizing finite state machines from hierarchical behavioral specifications (namely hierarchical graph schemes) [Sklyarov99].

Later on, Sklyarov has addressed some disadvantages of this *hierarchical finite state machine* model and proposed an enhanced version: the *Recursive HFSM* (RHFSM) model [Sklyarov04]. The typical time overhead that is caused by recursive invocations was reduced by means of executing the algorithm-related operations and the flow control operations in parallel (see Figure 2.4).

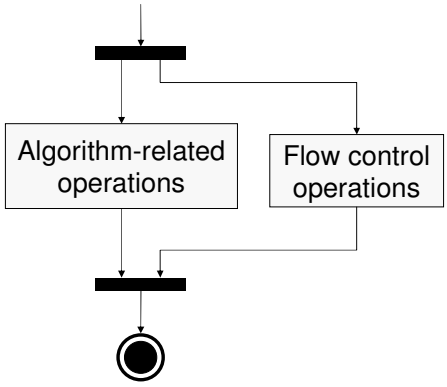


Figure 2.4 - Parallel execution of algorithm-related and flow control operations

The RHFSM model requires three stacks for storing and restoring module identifiers, state identifiers, and context data when performing or returning from hierarchical calls. As shown in practical applications of this model [Sklyarov05, Sklyarov06a], these stacks can be implemented on built-in memory blocks, significantly reducing the use of FPGA logic.

The RHFSM model allows for correct implementation of both directly and indirectly recursive calls at the same time that it provides the advantages of modular and hierarchical algorithm decompositions, which are generalized in software algorithm design.

Drawbacks that have been pointed out on the RHFSM model are as follows:

a) The use of three stacks per algorithm implementation suggests greater utilization of logic or block memory vs. a conventional single stack solution [Ninos08];

b) Either the stacks for keeping state and module identifiers are as large as the data stack, incurring significant area overhead; or the hierarchical invocation depth must be bounded to a lower stack size [Ninos08].

Both drawbacks loose their significance with algorithms which require higher amounts of context data to be kept in the data stack and also when the number of modules and states per module is lower. For instance, if the storage of a context data entry requires 72 bits and there are 3 modules and 8 states per module, then all information (context data, module identifier, and state identifier) could be stored in 77 bits. Moreover, if one wants to reduce the number of stacks (at the expense of algorithm clarity), then a single 77-bit wide stack can be used.

2.2.2.3. Ferizis and El Gindy

George Ferizis and Hossam El Gindy have proposed a method for mapping recursive functions to reconfigurable hardware which does not require stacks [Ferizis06]. This method consists of unrolling recursive functions by means of runtime reconfiguration and placing them into a pipeline.

Mapping a loop into a pipelined linear array is the basis of the methods earlier proposed by Bondalapati and Prasanna [Bondalapati98, Bondalapati00] (who also inspired Maruyama *et al.* [Maruyama99, Maruyama00]) and by Weinhardt and Luk [Weinhardt99]. Ferizis and El Gindy pushed the idea further in such a way that the pipelines created by recursive functions can be mapped as trees instead of arrays in case they contain multiple recursive calls. This approach led to a series of significant challenges which the authors had to address with techniques that are rather complex and resource consuming.

In order to compare this method's performance with that of regular stack implementations, its authors have carried out experiments on the basis of two algorithms (quicksort and force approximation) and reported high speedups [Ferizis06]. However, it seems that the stack implementations have not been subject to any kind of optimization, and some design details, such as the synthesis tools and the language(s) that have been used were left unidentified.

The drawbacks which have been found in Ferizis and El Gindy's approach are the following:

a) Algorithm-generality limitation: area-related problems arise when mapping recursive functions with a process growth rate greater than 1 [Ferizis06];

b) Platform-specificity: it relies on run-time reconfiguration, which is only available for certain FPGAs;

c) High concept-to-implementation time span [Ninos08]: complex preliminary algorithm-specific analysis must be carried out by the designer, as there is no known CAD tool available for the task, so far;

d) Error-prone design: the complexity and diversity of the techniques that have to be used for solving algorithm-specific challenges raises the probability of making mistakes [Ninos08];

e) Incompatibility with System-On-Chip (SOC) design [Ninos08]: because of implementation requirements, the design cannot coexist in the same reconfigurable device with other designs.

2.2.2.4. Ninos and Dollas

In contrast with the previous processing-oriented approaches to implement recursion in hardware, Spyridon Ninos and Apostolos Dollas have proposed a data-oriented solution [Ninos08].

The method is based on a *recursion simplification* procedure which requires, for each recursive call state, the following preliminary tasks:

1. Identification of the *condition for recursion*, *i.e.* the condition which determines whether that recursive call is to be activated;

2. Identification of the *local data*, *i.e.* the context values which must be stored onto the data stack when that recursive call is activated and restored when returning.

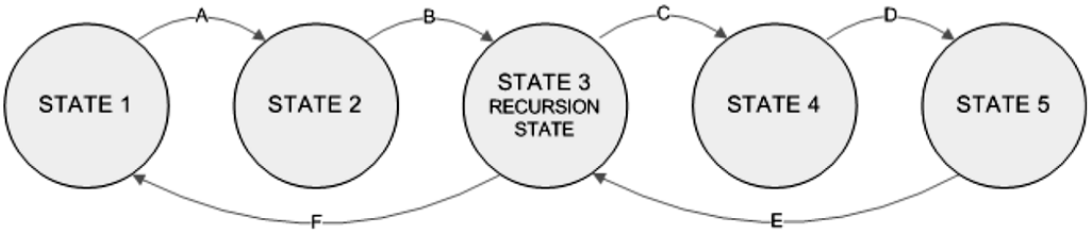


Figure 2.5 – An originally recursive state diagram after recursion simplification (from [Ninos08])

After the recursion simplification procedure, recursion can be thought of as conditional flow (see Figure 2.5): when a condition for recursion is met, upon testing in a state with a recursive invocation, local data are pushed onto the stack and execution is sent back to the initial state (transition F in Figure 2.5); after a recursive call has activated the final state, local data are restored from the stack and execution is brought back to the invocating state (transition E in Figure 2.5).

Ninos and Dollas have reported (i) speedups of this method compared to software implementations of up to 2.86 and (ii) relevant area occupation and clock speed enhancements compared to Sklyarov's HFSM-based approach [Ninos08].

However, this data-oriented approach provides no support for indirect recursion. In fact, not even modularity is supported. Designers must describe the whole algorithm as a single function that is implemented by means of an FSM. Thus, unless the algorithm which the designer needs to implement consists of a simple directly recursive function, clarity and readability of the description become compromised.

Moreover, the recursion simplification stage can constitute a complex and time-consuming task for many algorithms. In particular, complex scenarios can lead to challenging identification of the local data. Thus, while there is no well-defined set of rules which assures correctness of this process, it is impossible to develop software tools for automatic (and, ideally, transparent) recursion simplification. This hard task must therefore be carried out by the designers, which might lead to mistakes.

2.3. Conclusion

Something is said to be recursive if it partially consists or is defined in terms of itself. This thesis focuses on the procedure-oriented kind of recursion, although recursive data structures are used therein. Recursion can be direct and indirect.

In order to illustrate the applicability and advantages of recursive algorithms when they are implemented in hardware for solving computationally intensive problems, we will explore combinatorial search problems as an example.

Combinatorics is a branch of mathematics which can be described as the study of how discrete sets of objects can be arranged, counted, and constructed, according to specified constraints. Combinatorial search problems are divided in four types: decision problems, search problems, enumeration problems, and optimization problems.

Search trees are typically implemented by means of a backtracking mechanism. In this context, the search consists of a multi-stage decision process in which some choice is made at each stage. In case the algorithm detects that the previous choices can not lead to a solution, the context belonging to the previous stage is restored, in order to try alternative choices. Algorithms that use this technique for solving search problems are called backtracking search algorithms and they are often described recursively. Choosing one of the alternative search branches corresponds to making a recursive call, whereas backtracking corresponds to returning from a recursive call.

When implementing any algorithm in software, the use of recursion instead of iteration can be better or worse, depending on the criteria that are chosen and the class of algorithm. It is widely accepted that, for certain classes of algorithms, recursion provides clean, concise, elegant, and robust designs that are easy to conceive, understand, and modify with minimal design costs. However, recursive algorithms in software are generally considered slow and very resource-consuming when compared to iterative ones. Strategies for implementing recursion in hardware have started to be proposed only recently and therefore very few results are available for comparison. Furthermore, these strategies implement recursion in different ways, which means they may lead to different iteration-versus-recursion comparison results.

Tsutomu Maruyama, Masaaki Takagi, and Tsutomu Hoshino are amongst the first known authors to propose a strategy for implementing recursion in hardware. *Multi-*

thread execution and *speculative execution* are two techniques that aim for the implementation and optimization of recursion in backtracking search algorithms on the basis of pipelining and the use of a logic stack. Known limitations regarding these authors' proposals concern (i) the maximum speedup equaling the pipeline's depth, (ii) the speed increase obtained at the expense of area utilization, and (iii) the efficiency when handling recursive functions that call themselves multiple times.

Sklyarov has proposed the RHFSM model that allows for correct implementation of both direct and indirect recursive calls at the same time that it provides the advantages of modular and hierarchical algorithm decompositions (which are generalized in software algorithm design). It uses three stacks but prevents the typical time overhead that is caused by recursive invocations by means of executing the algorithm-related operations and the flow control operations in parallel. Two drawbacks that have been pointed out on the RHFSM model have been found. One suggests that it requires significant utilization of logic or block memory when compared to a conventional single stack solution. The second states that either the support stacks are as large as the data stack, incurring significant area overhead; or the hierarchical invocation depth must be bounded to a lower stack size. However, a few remarks which render the relevance of these drawbacks low have been presented.

George Ferizis and Hossam El Gindy have proposed a method for mapping recursive functions to reconfigurable hardware which does not require stacks and consists of unrolling recursive functions by means of runtime reconfiguration and placing them into a pipeline. This proposal has been inspired by research carried out by Bondalapati and Prasanna and some details are not very clear. Several disadvantages of Ferizis and El Gindy's proposal have been pointed out.

Spyridon Ninos and Apostolos Dollas have proposed a data-oriented solution for implementing recursion in hardware which is based on a preliminary *recursion simplification* procedure. For each recursive call state, the *condition for recursion* and the *local data* must be identified. The drawbacks of Ninos and Dollas' strategy includes the time-consuming and error-prone recursion simplification procedure, which is presently impossible to automate, and lack of support for indirect recursion and modularity, forcing designers to describe whole algorithms as single functions that are implemented with an FSM.

3. Design Space Exploration

This chapter analyzes computationally intensive problems which are taken mainly from the scope of combinatorial search, in which both recursion and iteration can rationally be applied. For each of six selected problems, the following is provided: problem description, application domains, an algorithm for solving it, and a detailed illustration in which the given algorithm is applied to solve the problem on the basis of a practical example. Four particular problems are solved with the aid of backtracking search algorithms, namely: set covering, Boolean satisfiability, graph coloring, and knapsack. Two supplementary problems (tree-based data sorting and discovering of a greatest common divisor) are also studied. A generic approach to backtracking search algorithms is proposed and discussed.

3.1. Introduction

A primary objective of this research is the comparison and evaluation of alternative recursive and iterative implementations for different algorithms. For this purpose, it is necessary to select a set of algorithms which can be described both iteratively and recursively. For instance, if we want to calculate the factorial of a non-negative integer using either an iterative algorithm or a recursive algorithm, we can use the pseudocode depicted in Figure 3.1-a and Figure 3.1-b, respectively.

a)

```
it_fact(n)
{
   fact = 1
   for each i from 1 up to n
      fact = fact * i
   return fact
}
```

b)

```
rec_fact(n)
{
   if n < 2
      return 1
   else
      return n * rec_fact(n – 1)
}
```

Figure 3.1 - Pseudocode for calculating the factorial iteratively (a) and recursively (b)

3.2. Backtracking search algorithms

A class of algorithms that can be implemented on the basis of recursive descriptions is *backtracking search algorithms*. A set of problems has been selected in order to study the advantages and disadvantages of recursive algorithms in comparison with iterative algorithms.

3.2.1. Generic approach to backtracking search algorithms

A basic structure for backtracking search algorithms has been explained in detail in [Skliarova04a] and used in [Pimentel07] (see Figure 3.2).

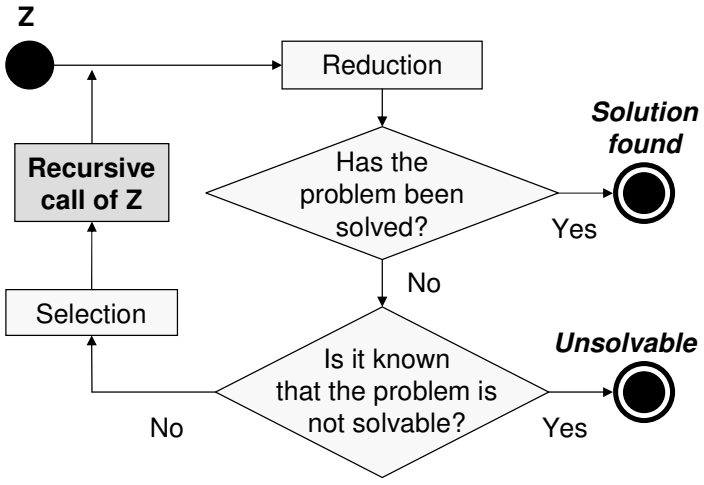


Figure 3.2 - Basic structure for backtracking search algorithms

The structure expresses a recursive procedure which is executed at every node of the search tree and it determines which nodes to visit next. The process starts by simplifying the current problem instance using a set of *reduction* operations. When no further reduction is possible, a *resolution test* is performed to determine whether the problem has been solved. In case it has been solved, the process ends and the solution is provided. Otherwise, a *solvability test* is carried out to verify if the current problem is unsolvable. In case the problem is found unsolvable, the process ends with no solution. Otherwise, the solver might have to try alternative paths in the search tree in order to check whether there is one which leads to a solution. The set of operations that determines which path to follow is called *selection*. When a chosen search path fails to provide a solution, the algorithm backtracks and selects another one, if available.

The implementation of each stage of this process, *i.e.* the reduction, the resolution test ('Has the problem been solved?'), the solvability test ('Is it known that the sub-problem is not solvable?') and the selection, depends on the particular algorithm which is executed. Small adaptations of the structure itself can also be required.

The basic algorithmic structure is not the only characteristic that different backtracking search algorithms can share. In fact, a common data structure can be used to specify problem instances for a variety of such algorithms. Most combinatorial search problems can be expressed in several equivalent mathematical formulations based on different standard data structures, such as matrices, graphs and Boolean functions. However, matrices can be found very appropriate for hardware implementations [Skliarova04a] because matrices can easily be stored and processed in both software and hardware, and because most combinatorial search problems can efficiently be formulated over matrices. Thus, without loss of generality, we have selected matrices as the data structure to be used for specifying the combinatorial problem instances, when applicable.

Let us now address the combinatorial problems which were selected for this research and how the considered basic algorithmic structure can be used to implement the correspondent solving algorithms.

3.2.2. The set covering problem

Given a group G of finite sets whose union is a universe U, a *cover* is a group $C \subseteq G$ of sets whose union is still U. The *set covering problem* consists of finding a cover with the minimum number of subsets [Bäck95].

Applications of the set covering problem can be found in Boolean expression simplification [Breuer70], resource allocation [Rubin73, Walker74] and committee forming endeavors (as illustrated next).

Let us solve the set covering problem for a simple practical example of choosing a group of specialists to hire for a scientific research expedition. In order to make both its scientific research component and its logistic support component possible, the expedition requires skills in 6 different fields of specialization: telecommunications, mountaineering, pilotage, zoology, botany, and climatology. There are 8 available specialists, each one skilled in at least one of those fields. In order to reduce the expedition costs, we want to hire the minimum number of specialists required to cover the 6 fields of specialization. The Euler diagram shown in Figure 3.3 identifies the universe of required specialization fields (rectangle U) and the set of specialization fields of each available specialist (ellipses G_1 to G_8). The same diagram reveals a minimum cover for this scenario: the ellipses with a white circumference (G_2, G_5 and G_8).

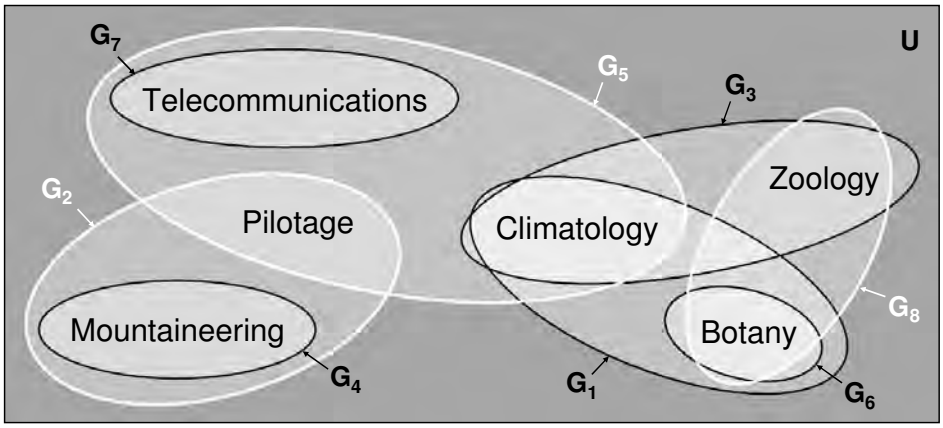


Figure 3.3 - Practical example diagram for the set covering problem

In this example, the universe for which a cover must be determined is the set of required specialization fields $U = \{t, m, p, z, b, c\}$, where t, m, p, z, b and c stand for telecommunications, mountaineering, pilotage, zoology, botany, and climatology,

respectively. The given group of sets which can be used to compose the cover is $G=\{G_1,G_2,...,G_8\}$, where $G_1=\{b,c\}$, $G_2=\{m,p\}$, $G_3=\{z,c\}$, $G_4=\{m\}$, $G_5=\{t,p,c\}$, $G_6=\{b\}$, $G_7=\{t\}$, and $G_8=\{z,b\}$.

Instances of the set covering problem can be expressed by matrices in such a way that a solution can be found by means of some algorithm operating over those data structures. For this problem, binary matrices are used [Zakrevskij08]. Figure 3.4 illustrates the conversion steps for the given example. The relevant details of each step are emphasized with a black background.

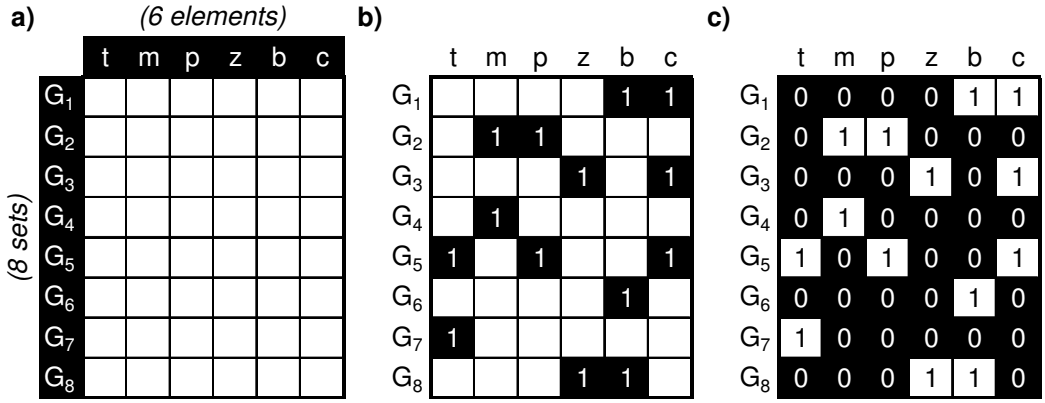


Figure 3.4 - Converting a set covering problem instance to a binary matrix

If the given family of sets G is composed of S sets, and the universe U, which is the union of all sets in G, contains E elements, then a binary matrix with S rows and E columns is required for this conversion (see Figure 3.4-a). Thus, each row corresponds to a given set and each column to an element of U. For each time an element of U belongs to a set in G, the correspondent cell must be filled with 1 (see Figure 3.4-b), *i.e.* we must guarantee:

$$\forall i \in \{1,2,...,S\}, \forall j \in \{1,2,...,E\}, U_j \in G_i \Leftrightarrow M_{ij}=1$$

where M_{ij} is the matrix cell in row i, column j. For instance, the cells of column z in rows G_3 and G_8 must be filled with 1 because z is element of sets G_3 and G_8. Finally, all empty cells must be filled with 0 (see Figure 3.4-c).

By analogy, in the scientific research expedition example, each row corresponds to an available specialist and the values 1 in that row indicate the specialization fields in which he or she is skilled.

After the problem instance has been converted to a binary matrix, solving the set covering problem corresponds to finding the minimal number of rows that include at least one value 1 in each column. This description is actually that of the *matrix covering problem* [Zakrevskij71]. The approximate algorithm proposed in [Zakrevskij81] to solve this problem is depicted in the diagram of Figure 3.5. When the algorithm finishes, the solution is the set of rows that have been removed.

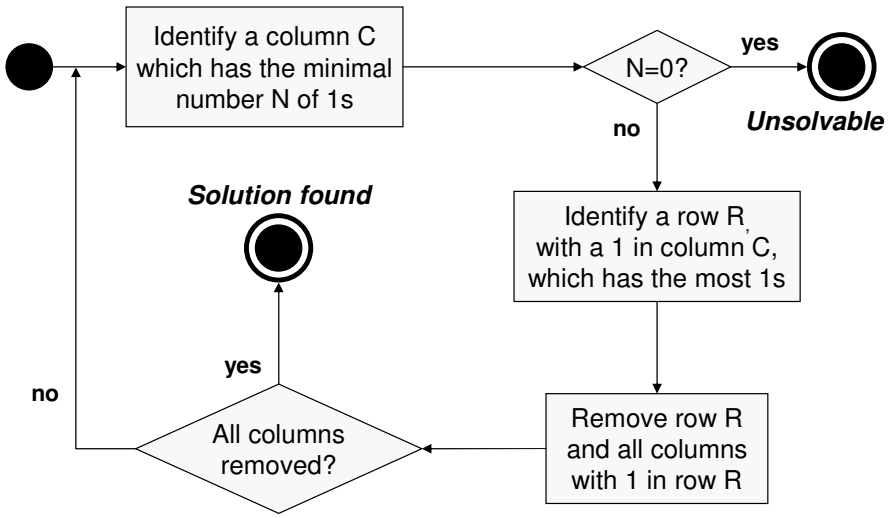


Figure 3.5 - Iterative approximate algorithm to solve the matrix covering problem

Note that this algorithm includes a solvability test, which is shown at the top right-hand corner of the diagram in Figure 3.5. In fact, unsolvable instances for the matrix covering problem emerge if the given matrix contains a column with no values 1. However, the set covering problem description given in the beginning of this section leads to solvable instances only, and therefore the algorithm could simply continue choosing the next given set (row) to include in the cover and a solution would eventually be found. In the worst case scenario, the solution found would be a cover which would include all given sets (all rows).

Figure 3.6 demonstrates the resulting steps of the algorithm when applied to the binary matrix obtained for the given example (see Figure 3.4), depicting the three iterations of the algorithm which lead to the solution. The rows and the columns which are removed at each iteration are presented with a black background while a grey background indicates previously removed matrix parts.

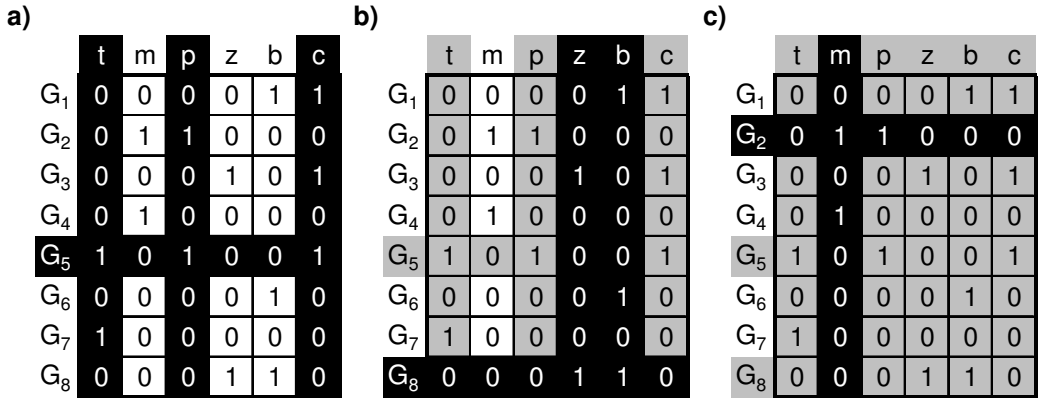


Figure 3.6 - Solving a set covering problem instance

At the first iteration, row G_5 is removed because no other row contains more values 1, and then columns t, p and c are removed because these contain a value 1 in that row. After two more iterations with the same procedure, the algorithm reveals the solution composed of the rows which were removed: G_2, G_5 and G_8.

In the context of the given problem instance, this solution points to hiring the specialists skilled in: mountaineering and pilotage; telecommunications, pilotage and climatology; and zoology and biology. These correspond to the ellipses with a white circumference in Figure 3.3.

The solution found for the given example is optimal, as no cover with less than 3 sets can be found. However, the approximate algorithm considered does not guarantee an optimal solution for an arbitrary problem instance. Furthermore, it should be noticed that the algorithm does not include a backtracking mechanism.

Let us now consider how an exact algorithm for solving the set covering problem can make use of backtracking in order to obtain an optimal solution in all cases, and how the basic structure for backtracking algorithms (depicted in Figure 3.2) can be used for that purpose.

The exact algorithm for solving the matrix covering problem proposed in [Zakrevskij81] was adapted to the basic structure for backtracking algorithms in [Skliarova04a]. The reduction rules, selection rules, solvability test and resolution test which were applied, in fact, provide minimum covers composed of columns. Let us rephrase these rules and tests in such a way that permits covers composed of rows to be obtained.

The adjusted resolution and solvability tests correspond to those presented in Figure 3.5, *i.e.*: a solution has been found if all columns have been erased; and the current instance is unsolvable if the minimum number of values 1 in a column is 0.

The reduction rules become the following:

R1 - For every pair of columns col_i and col_j in the matrix, where $i \neq j$, if $col_i \wedge col_j = col_i$, then col_j must be removed. For instance, if the matrix would contain columns $[0,0,1]$ and $[0,1,1]$, then the latter would be removed because $[0,0,1] \wedge [0,1,1] = [0,0,1]$. This procedure is called *subsumption for columns*.

R2 - For every pair of rows row_i and row_j in the matrix, where $i \neq j$, if $row_i \wedge row_j = row_i$, then row_i must be removed. For instance, G_7 (in Figure 3.6) should be removed because $G_5 \wedge G_7 = G_7$. This procedure is called *subsumption for rows*.

On the other hand, the selection rules become:

S1 - For every column that contains a single element with value 1, the row which has this element must be included in the cover.

S2 - When all columns contain multiple elements with values 1, a column C with the minimum number of values 1 must be selected. Then, for every row R which includes an element with value 1 in column C, the same algorithm must be called to continue constructing the cover after including row R. When a first cover becomes complete, it is stored as 'the best cover', *i.e.* as the cover with the minimum number of rows. After that, 'the best cover' is replaced every time a cover which includes a lower number of rows is found. On the other hand, if the number of rows included in a cover under construction reaches that of 'the best cover', the current algorithm invocation must be discontinued.

Additionally, every time a row is included in the cover under construction, it is also removed from the matrix. All columns with value 1 in that row are removed as well.

After finding a cover, the exact algorithm backtracks in order to find an optimal solution (a cover with the minimum number of rows) which might be found in others leaves of the search tree.

3.2.3. The Boolean satisfiability problem

The *Boolean satisfiability problem* (also known as *SAT*) consists of determining whether it is possible to assign values to the variables of a given Boolean formula in such a way as to make the formula evaluate to true [Micheli94, Zakrevskij08].

In fact, any Boolean formula is said to be either:

- **i)** *contingent*, if its value depends on the values of the variables;
- **ii)** a *tautology*, if it always evaluates to true;
- **iii)** a *contradiction*, if it always evaluates to false.

When a Boolean formula is a contradiction, the corresponding SAT problem instance is *unsatisfiable*. Otherwise, it is *satisfiable*.

In the context of the Boolean satisfiability problem, the Boolean formulae are usually presented in the Conjunctive Normal Form (CNF), *i.e.* as a conjunction of clauses. A *clause* is a disjunction of literals and a *literal* is a variable or its negation. The formula $\left(x_1 \vee x_2 \vee x_3\right) \wedge \left(x_2 \vee \overline{x_4}\right) \wedge \left(x_2\right) \wedge \left(\overline{x_1} \vee \overline{x_4} \vee x_5\right)$ is an example of a Boolean formula in CNF.

There are multiple versions of the SAT problem deriving from the original version. For example, the *3-SAT* problem can be obtained by restricting the maximum number of literals in each clause to 3. It is known that the conversion of SAT problem instances into 3-SAT problem instances is achievable in polynomial time [Zhong99].

An overview of the most well-known applications of SAT and an outline of several other successful applications of SAT is presented in [Marques-Silva08]. The SAT problem has direct applications in mathematical logic, artificial intelligence, VLSI engineering, and computing theory [Gu97]. Furthermore, problems such as constraint satisfaction problems and constrained optimization problems can be transferred to SAT [Gu04]. In fact, methods to solve SAT formulae play an important role in solving many problems in automated reasoning, computer-aided design, computer-aided

manufacturing, machine vision, database, robotics, integrated circuit design automation, computer architecture design, and computer network design [Gu97].

Let us consider an example of a Boolean formula in CNF:

$$\left(x_1 \vee \overline{x_4}\right) \wedge \left(\overline{x_1} \vee \overline{x_4}\right) \wedge \left(x_1 \vee x_2 \vee x_4\right) \wedge \left(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4\right) \wedge \left(x_3\right)$$

This particular formula is satisfiable because it evaluates to true, for instance, with the following set of variable assignments:

$$\begin{cases} x_1 = true \\ x_3 = true \\ x_4 = false \end{cases}$$

Boolean formulas can be converted to matrices in such a way that allows the SAT problem to be solved with some algorithm operating over those matrices [Gu97]. Figure 3.7 illustrates this conversion, emphasizing the relevant details of each step with a black background, for the given Boolean formula.

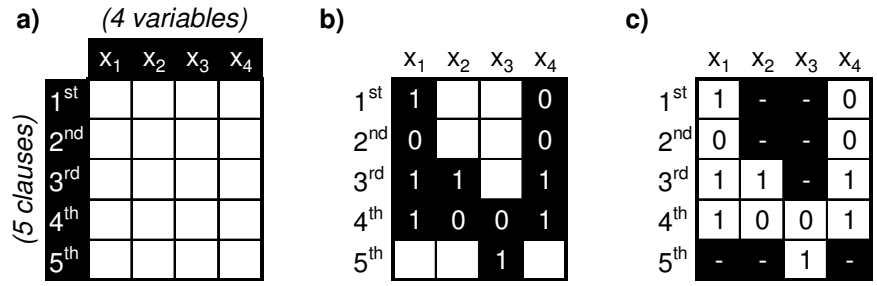


Figure 3.7 - Converting a Boolean formula to a ternary matrix

As mentioned before, a SAT problem instance is usually a Boolean formula in CNF. If this formula is composed of C clauses and includes V distinct variables, the ternary matrix which can be used to express the corresponding problem instance must have C rows and V columns (see Figure 3.7-a). Thus, each row corresponds to a clause and each column corresponds to a variable. In other words, the i^{th} cell in the j^{th} matrix row corresponds to the occurrence of the i^{th} variable in the j^{th} clause of the Boolean formula.

For each time a variable appears in the Boolean formula, the corresponding matrix cell must be filled in with 0 if the variable is negated, or with 1 if it's not (see Figure 3.7-b). For example, the forth cell in the second row must be filled in with 0 because the

second clause contains the forth variable negated, *i.e.* the second conjunction contains the literal $\overline{x_4}$.

After all the literals in the Boolean formula have been mapped onto the ternary matrix using this method, every empty cell must be filled in with '-', which stands for the *don't-care* value (see Figure 3.7-c).

Two equally sized ternary vectors u and v are considered *orthogonal* if there is an index i for which $\{u_i, v_i\} = \{0,1\}$. For instance, vectors $[1,1,0,0,0]$ and $[1,-,0,-,1]$ are orthogonal because their fifth elements are 0 in one vector and 1 in the other. Without this pair of homologous elements, these two vectors would not be orthogonal. On the other hand, if two equally sized ternary vectors are not orthogonal, they intersect in the Boolean space. In such case, the ternary vector (w) which represents the intersection of those two vectors (u and v, with n elements each) is determined as follows: for each index $i = 1, 2, ..., n$, if $u_i = v_i$ then $w_i = u_i = v_i$; otherwise, either 1) u_i is a *don't-care* value, in which case $w_i = v_i$, or 2) v_i is a *don't-care* value, in which case $w_i = u_i$. For instance, the intersection of $[0,0,-,-,-1,1]$ and $[0,-,0,-,1,-,1]$ is $[0,0,0,-,1,1,1]$. Figure 3.8 summarizes how to assign the i^{th} element of the intersection vector as a function of the i^{th} elements of the intersecting vectors u and v. Black cells correspond to values of u_i and v_i that are not valid for intersecting vectors.

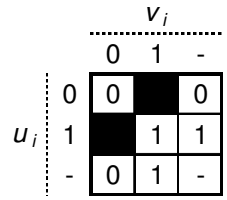


Figure 3.8 - Determining the i^{th} element of the intersection of ternary vectors u and v

Solving the Boolean satisfiability problem corresponds to finding a ternary vector which is orthogonal to every row in the ternary matrix that was built using the described method [Skliarova04a]. If such a vector is found, the solution is obtained in two more steps:

1. Negate every non-*don't-care* element in the vector that was found (*i.e.* replace its 0s and 1s respectively with 1s and 0s);

2. For each 1 or 0 obtained in step 1, assign the corresponding variable to true or false, respectively (*e.g.* if the second element of the negated vector is 0, x_2 must be assigned to false).

In order to apply the basic structure for backtracking algorithms which was previously mentioned (see Figure 3.2), it is necessary to define which operations are executed at each stage. In the case of the exact SAT solving algorithm presented in [Skliarova03], the resolution test ('Has the problem been solved?') is satisfied in case all matrix rows have been deleted; and the solvability test ('Is it known that the problem is not solvable?') is satisfied in case the problem has not yet been solved and all matrix columns have been deleted. The reduction operations used implement the following rules:

R1 - If a column contains just *don't-care* values, it must be deleted from the matrix;

R2 - All rows that are orthogonal to an intermediate vector w (that incrementally forms a solution) must be removed from the matrix; and all columns that correspond to the components of vector w with values 1 and 0 must be removed from the matrix;

R3 - If the matrix contains a row with just one component 1 (0) with an index i, then the i^{th} element of vector w must be assigned value 0 (1), *i.e.* the negated value;

R4 - If there is a column j in the matrix without values 1 (0) then the j^{th} element of w can be assigned value 1 (0).

Finally, the selection operations implement the following rules:

S1 - A column C_{max} containing a maximum number of non-*don't-care* values is selected. Let us designate the number of values 1 and the number of values 0 in column C_{max} respectively N_1 and N_0;

S2 - If $N_1 \geq N_0$, then value 0 for column C_{max} is included in w. If $N_1 < N_0$, then value 1 for column C_{max} is included in w. This creates a sub-matrix that will be examined at the next iteration;

S3 - If this path fails, it is necessary to backtrack and repeat the attempt including the alternative value for column C_{max} in vector w.

Let us now combine these algorithm stage descriptions with the basic structure for backtracking algorithms. Figure 3.9 depicts the resulting search steps for the matrix example obtained in Figure 3.7. Above each step illustration, the reduction and selection rules which are applied are indicated using labels *R1* to *R4,* and *S1* to *S3*, respectively. The rows and columns which are removed at each step are presented with a black background, while a grey background indicates previously removed matrix parts. When an element of vector w is assigned a value, it is also highlighted with a black background.

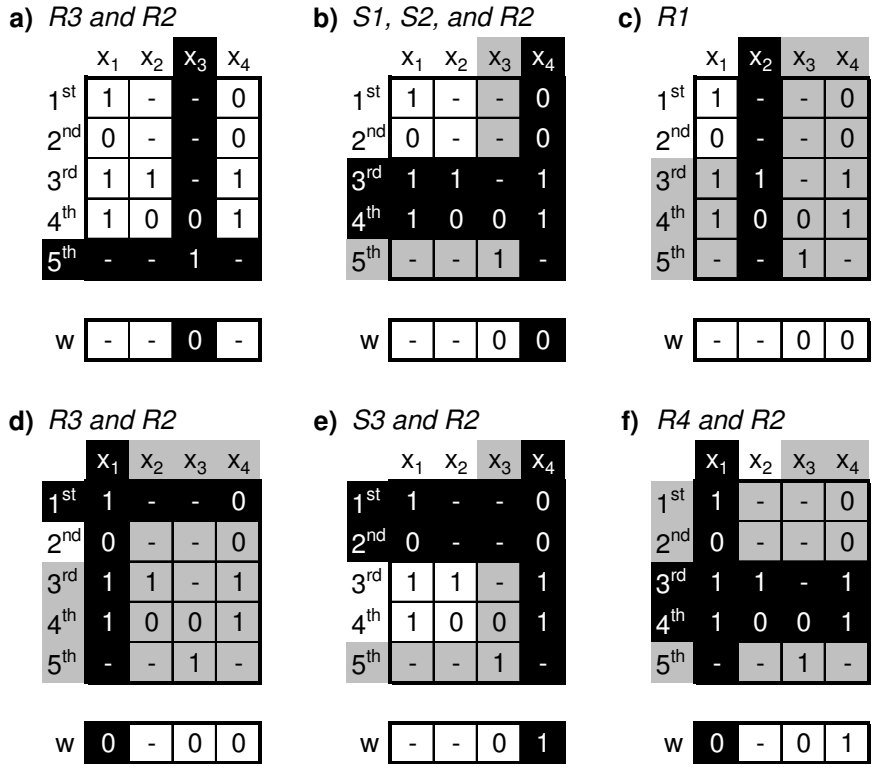


Figure 3.9 - Solving a Boolean satisfiability problem instance

Figure 3.10 depicts the search tree used to solve the given SAT problem instance. Each of the six steps illustrated in Figure 3.9 (labeled from 'a' to 'f') is identified in Figure 3.10 using the same letter. Throughout the search process, reduction operations prune certain branches (nodes represented in gray), and make deterministic assignments to elements of vector w (nodes represented in white). On the other hand, when there are alternative search branches (nodes represented in black), selection operations determine which branch to try next.

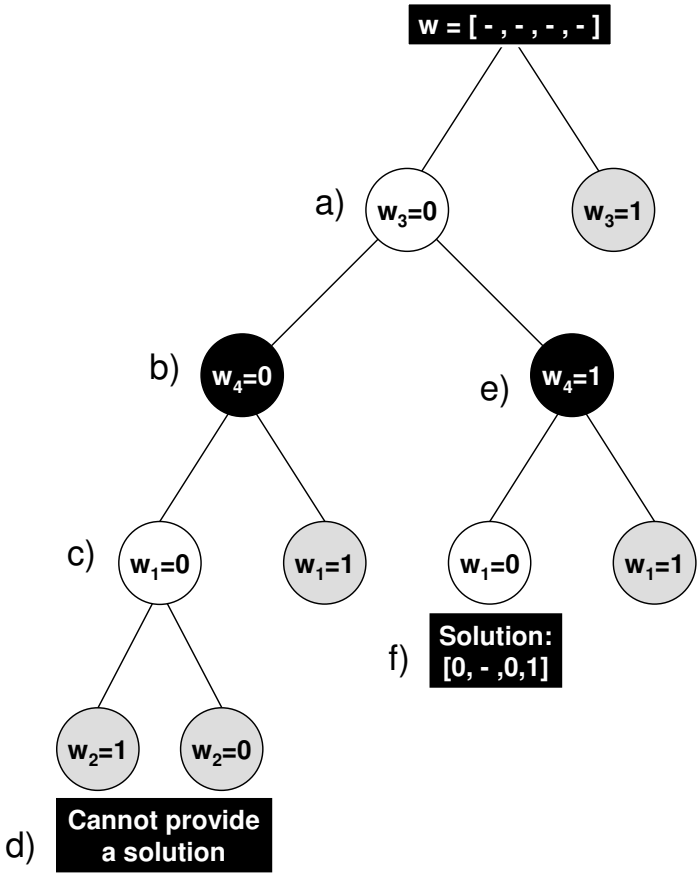


Figure 3.10 – Search tree for the SAT problem example

The search starts with a first invocation to the Z module depicted in Figure 3.2 and with the *don't-care* value assigned to every element of the solution vector w. After Figure 3.9-a, no more reduction can take place and there are still rows and columns left, so in Figure 3.9-b value 0 for column x_4 is included in vector w according to the selection procedure. Then, some reduction takes place and, after the step depicted in Figure 3.9-d, there is still one row left (the second row), which means a solution was not yet found; but all columns have been removed, meaning this search path cannot provide a solution.

Hence, the search must backtrack in order to try the search path alternative to the one chosen in Figure 3.9-b. This time (see Figure 3.9-e), value 1 for column x_4 is included in vector w and then reduction rules are applied again. Finally, in Figure 3.9-f, all rows have been removed, meaning a solution has been found. Vector w ($[0,-,0,1]$, at the end) has been constructed throughout this process and is now orthogonal to all given matrix rows.

As previously mentioned, the solution is obtained by assigning the negation of each 1 or 0 element in vector w to its corresponding variable, *i.e.*:

$$\begin{cases} x_1 = true \\ x_3 = true \\ x_4 = false \end{cases}$$

3.2.4. The graph coloring problem

There are various *graph coloring problems*, such as the vertex coloring problem, the edge coloring problem and the face coloring problem. The most usual is the *vertex coloring problem* which consists of coloring the vertices of a graph using the minimum number of colors and making sure that no two adjacent vertices get the same color [Zakrevskij00, Diestel00].

Graph coloring algorithms are widely used for solving different engineering problems in robotics and embedded systems [Goossens97, Subramonian04, Ezick], microprogramming for application-specific embedded microprocessors, resource distribution, etc. [Culberson, Wu93].

Let us solve the graph coloring problem applied to the practical example of coloring a map of Portugal provinces. Historically, Portugal has been divided in eleven provinces delineated in Figure 3.11-a: Minho (A), Trás-os-Montes e Alto Douro (B), Douro Litoral (C), Beira Alta (D), Beira Litoral (E), Beira Baixa (F), Estremadura (G), Ribatejo (H), Alto Alentejo (I), Baixo Alentejo (J), and Algarve (K).

In order to solve this problem, the map of provinces must be converted to a graph in which: each province in the map is represented by a vertex; and vertices corresponding to contiguous provinces are connected by edges, *i.e.* are adjacent. For instance, vertex J must be connected by edges to vertices G, I and K because Baixo Alentejo has common borders with Estremadura, Alto Alentejo and Algarve.

The resulting province adjacency graph is depicted in Figure 3.11-b, in which black, gray and white colors reveal an optimal coloring. In fact, the minimum number of colors for this example is 3: one color for Minho, Beira Alta, Ribatejo, and Baixo Alentejo; another color for Trás-os-Montes e Alto Douro, Beira Litoral, Alto Alentejo, and Algarve; and a third color for Douro Litoral, Beira Baixa, and Estremadura.

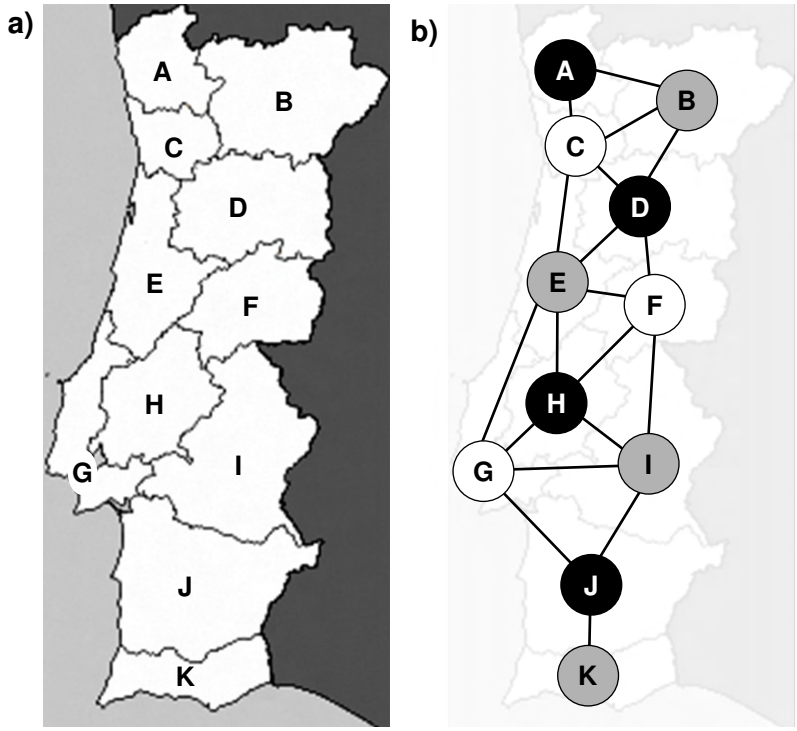


Figure 3.11 - Portugal's historical province map (a) and the corresponding province adjacency graph (b)

Conversion of a graph to a ternary matrix is illustrated in Figure 3.12, for the province adjacency graph example, and it consists of the following steps:

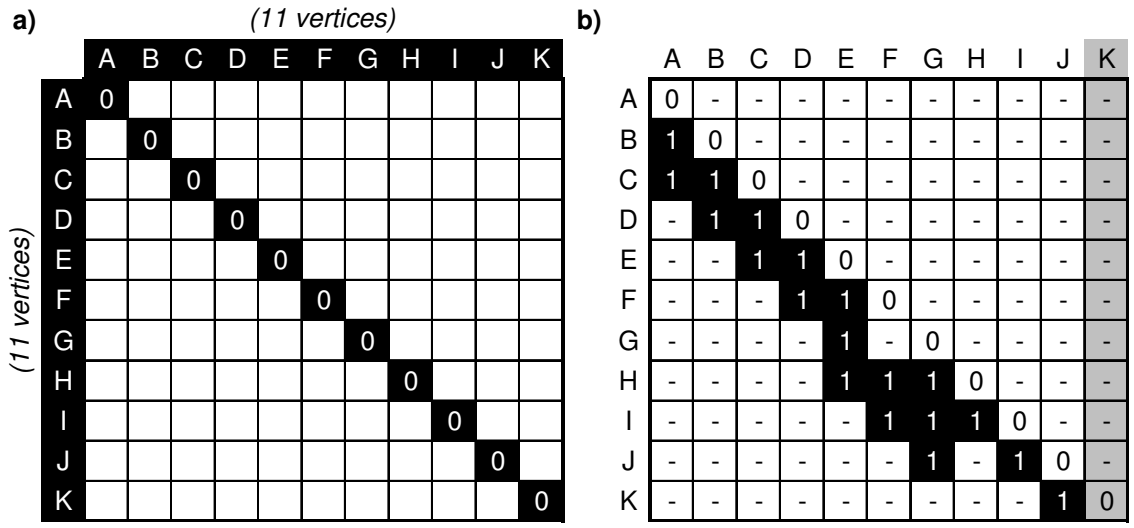
1. Creating a matrix with N rows and N columns, where N is the number of vertices in the graph (see Figure 3.12-a);

2. Inserting value 0 in every cell which belongs to the main diagonal of the matrix (see Figure 3.12-a);

3. In each cell below the main diagonal (because the cells above it won't be used), inserting a value 1 if and only if the cell's coordinates correspond to adjacent vertices (see Figure 3.12-b). For instance, the cell in row J and column G must have value 1 because there is an edge connecting vertex G to vertex J;

4. Inserting *don't-care* values in all empty matrix cells;

5. Removing every column which has no value 1 (*e.g.* column K, in Figure 3.12-b), keeping track of which vertex each column corresponds to.



	A	B	C	D	E	F	G	H	I	J	K
A	0										
B		0									
C			0								
D				0							
E					0						
F						0					
G							0				
H								0			
I									0		
J										0	
K											0

	A	B	C	D	E	F	G	H	I	J	K
A	0	-	-	-	-	-	-	-	-	-	-
B	1	0	-	-	-	-	-	-	-	-	-
C	1	1	0	-	-	-	-	-	-	-	-
D	-	1	1	0	-	-	-	-	-	-	-
E	-	-	1	1	0	-	-	-	-	-	-
F	-	-	-	1	1	0	-	-	-	-	-
G	-	-	-	-	1	-	0	-	-	-	-
H	-	-	-	-	1	1	1	0	-	-	-
I	-	-	-	-	-	1	1	1	0	-	-
J	-	-	-	-	-	-	1	-	1	0	-
K	-	-	-	-	-	-	-	-	-	1	0

Figure 3.12 - Converting a graph coloring problem instance to a ternary matrix

It should be noticed that, as a result of these conversion steps, if two vertices are adjacent, the correspondent matrix rows are orthogonal. For instance, vertices F and I are connected by an edge and the correspondent matrix rows (respectively $[-,-,-,1,1,0,-,-,-,-]$ and $[-,-,-,-,-,1,1,1,0,-]$) are indeed orthogonal, namely regarding their sixth elements. Therefore, solving the vertex coloring problem corresponds to finding a minimum number of row subsets which satisfy the following conditions [Sklyarov06b]:

a) Each subset contains no orthogonal pair of rows;

b) Every matrix row must belong to exactly one subset.

At the end, the number of compiled subsets expresses the minimum number of colors that the given graph requires, while rows grouped in each subset correspond to vertices assigned the same color [Pimentel07].

Let us now consider which solvability and resolution tests and which reduction and selection operations should be embedded in the basic structure for backtracking search algorithms, in order to implement an exact algorithm for solving the vertex coloring problem.

There are no unsolvable instances for the vertex coloring problem. The worst case scenario corresponds to graphs in which every vertex is connected to all other vertices and even such instances are solvable. The solution, in these cases, consists of

assigning a different color to each vertex. Thus, the outcome of a solvability test is constant: always 'solvable'.

The resolution test is satisfied in case the matrix is empty, *i.e.* if all rows have been removed.

The method of condensation proposed in [Zakrevskij81] can be used in the implementation of the graph coloring algorithm [Sklyarov06b, Sklyarov07a] as a basis for the reduction and selection rules. Let us consider the reduction rules:

R1 - After selecting a new color, all matrix columns which contain no value 0 or no value 1, must be removed;

R2 - At any intermediate step of the algorithm, all matrix rows which contain only *don't-care* values must be removed and included in the subset under construction (meaning the correspondent vertices are assigned the current color);

R3 - All rows included in the constructed subsets must be removed from the matrix.

Finally, the selection rules used implement the following sequence of algorithm steps:

1. Choose a new color (*i.e.* create a new, initially empty subset);

2. Apply reduction rules R1 and R2;

3. Consider the topmost row m_i in the matrix;

4. Include row m_i in the constructed subset and remove it from the matrix (reduction rule R3);

5. Find out all other rows intersecting (*i.e.* not orthogonal) with vector m_i;

6. Select the first row found during step 5 which has not been tried yet (let us designate this m_j), include it in the constructed subset and then delete it from the matrix;

7. Reassign m_i to the intersection of m_i and m_j and repeat steps 5 to 7 if this is possible. If this is not possible, go to step 8;

8. If the intermediate matrix is not empty repeat steps 1 to 7. Otherwise, store the solution found, then backtrack to the nearest branching point (set at step 6), and try to find a better solution by repeating steps 6 to 8.

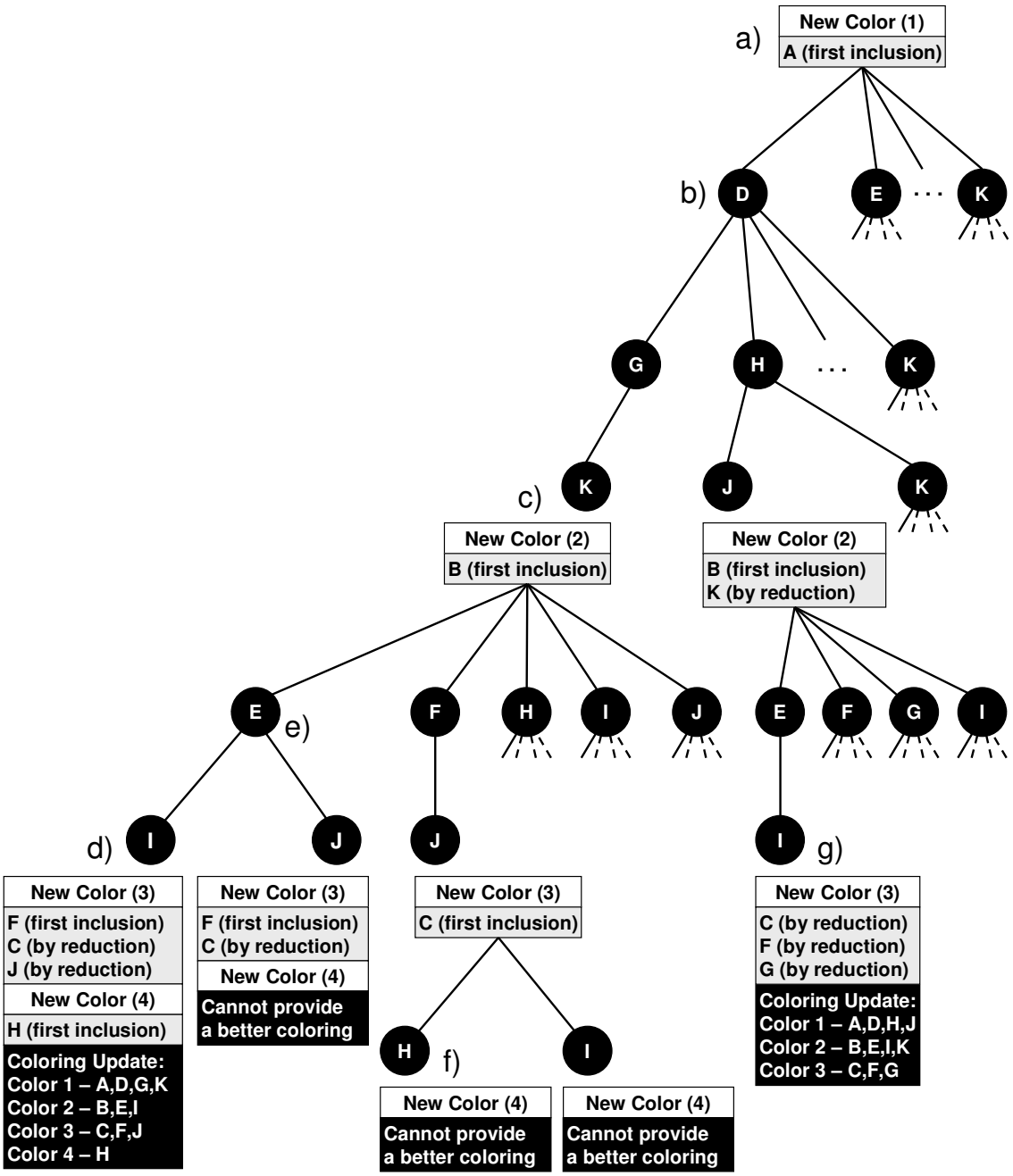


Figure 3.13 – Part of the search tree for the vertex coloring problem example

Figure 3.13 depicts a part of the search tree obtained when applying the described exact backtracking search algorithm to the graph example in Figure 3.11-b. Text

boxes contain operations which are deterministic at the current branch point, namely: starting a new color (a new subset of rows); including a graph vertex (a row) either as a first pick or by reduction; updating the stored coloring; and branch pruning. On the other hand, when a branch point is reached, current data must be stored in stacks in order to allow for alternative search branches to be explored. Each alternative branch starts with a selection-based vertex inclusion represented by a black circle in Figure 3.13. When the algorithm must backtrack, the branch point data are restored and other vertices are selected and analyzed.

A first subset (color 1) is initialized at the root of the search tree and a first row (A) is included, *i.e.* a first vertex is assigned color 1 (see Figure 3.13-a). A branch point with 8 alternatives is reached because there are 8 rows (D to K) which are not orthogonal to any of the rows in the current subset under construction: $\{A\}$.

As a first attempt, D is selected and included in the subset, and a new branch point is reached (see Figure 3.13-b). The subset under construction is now $\{A, D\}$ and there are 5 rows which are not orthogonal to any of its rows: G to K. The same procedure continues until every remaining row is orthogonal to at least one row in the subset under construction. This occurs for the first time when the first subset under construction becomes $\{A, D, G, K\}$ (see Figure 3.13-c). Then, a new subset of rows must be initialized and the search proceeds until all rows have been included in the constructed subsets (see Figure 3.13-d). A leaf of the search tree has been reached, having all vertices colored. The algorithm backtracks to the nearest branch point (see Figure 3.13-e) and alternative row J is selected and included in the subset under construction, which has been restored meanwhile.

The need to start a fourth subset renders the current search branch useless (see Figure 3.13-f) because a solution consisting of 4 colors has already been found. This search branch is therefore pruned and the algorithm backtracks again. The search continues and, when a new coloring is found having fewer subsets than the stored one, the solution is updated. Eventually, an optimal solution is stored (see Figure 3.13-g); the search continues until all search branches are implicitly tested but the solution is obviously not replaced again.

When the algorithm finishes, the solution is composed of the following 3 subsets of rows: $\{A, D, H, J\}$, $\{B, E, I, K\}$, and $\{C, F, G\}$. Finally, each group of vertices

corresponding to one of these row subsets is assigned a different color, as illustrated in Figure 3.11-b.

3.2.5. The knapsack problem

Given a set of items, each with a profit and a weight, and a knapsack with a certain weight capacity, the *0-1 knapsack problem* consists of selecting a subset of items whose total weight does not exceed the knapsack capacity and whose total profit is as large as possible [Beier04].

Applications of the knapsack problem can be found in a variety of resource allocation tasks [Gleeson94, Rodin90, Bodin91, Henig90], in cutting stock problems [Gilmore61, Hahn68, Madsen79, Seth87], cryptography [Merkle78, Chor88, Jan93], broadband communications [Ross89, Gavious94], etc.

Typical knapsack problem solvers make use of backtracking search. On the other hand, matrices are not a suitable data structure in this context.

Let us solve the knapsack problem on the basis of an example in which:

a) there are 4 items: 1 to 4;

b) p_i and w_i designate respectively the profit and weight of item i, $i = 1,2,3,4$;

c) vectors p and w are composed of p_1 to p_4, and w_1 to w_4, respectively;

d) $p = [7,6,4,6]$ and $w = [4,7,5,4]$;

e) the weight capacity of the knapsack is $c = 10$.

In order to provide the solution, a binary vector x composed of x_1 to x_4 must be constructed. If an item i is to be inserted in the knapsack then the i^{th} element of vector x (x_i) must be assigned to 1; otherwise to 0. For instance, if the solution would correspond to the insertion of items 2 and 3, then solution vector x should be assigned to $[0,1,1,0]$.

The exact algorithm which has been used in [Skliarova05] for solving the Knapsack problem makes no use of matrices but it is based on the basic structure for backtracking search algorithms (see Figure 3.2). The reduction rules, selection rules, resolution test, and solvability test have been defined in such a way that the algorithm executed is the one described in Figure 3.14.

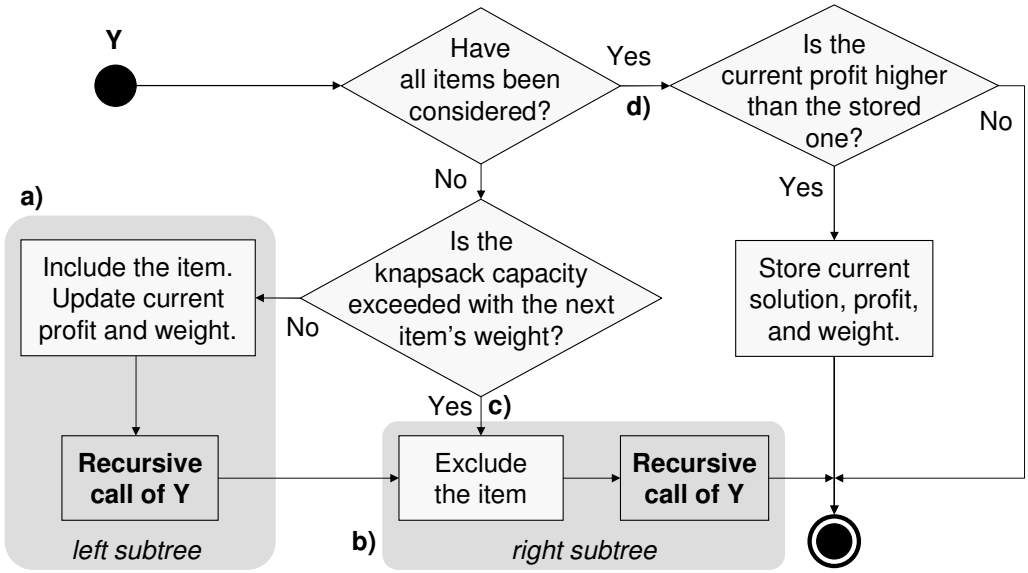


Figure 3.14 - Recursive exact algorithm to solve the knapsack problem

The search process is triggered with the invocation of module Y which includes potential recursive invocations of itself. Each recursive invocation corresponds to processing a subtree of the current search tree node. Left subtrees correspond to including the considered item (see Figure 3.14-a), while right subtrees correspond to excluding it (see Figure 3.14-b).

This algorithm also features pruning of search branches. Indeed, if the inclusion of an item implies an accumulated weight which exceeds the knapsack capacity (see Figure 3.14-c), the item cannot be included and thus the left subtree is not processed.

When a leaf in the search tree is reached (see Figure 3.14-d), the current (accumulated) profit is compared with the one previously stored (initially 0). If the current profit is higher, the current solution, profit and weight replace the stored ones.

Figure 3.15 depicts the search tree which results of applying this exact algorithm to solve the given knapsack problem example. The inclusion of an item i, in the solution vector under construction (x), is indicated as $x_i = 1$. Analogously, $x_i = 0$ indicates its

exclusion. The accumulated profit and weight are shown at the bottom of each node. Gray nodes indicate pruned subtrees. The algorithm carries out a pre-order traversal, *i.e.*, starting from the root, the processing sequence is: first the node, then the left subtree, and finally the right subtree.

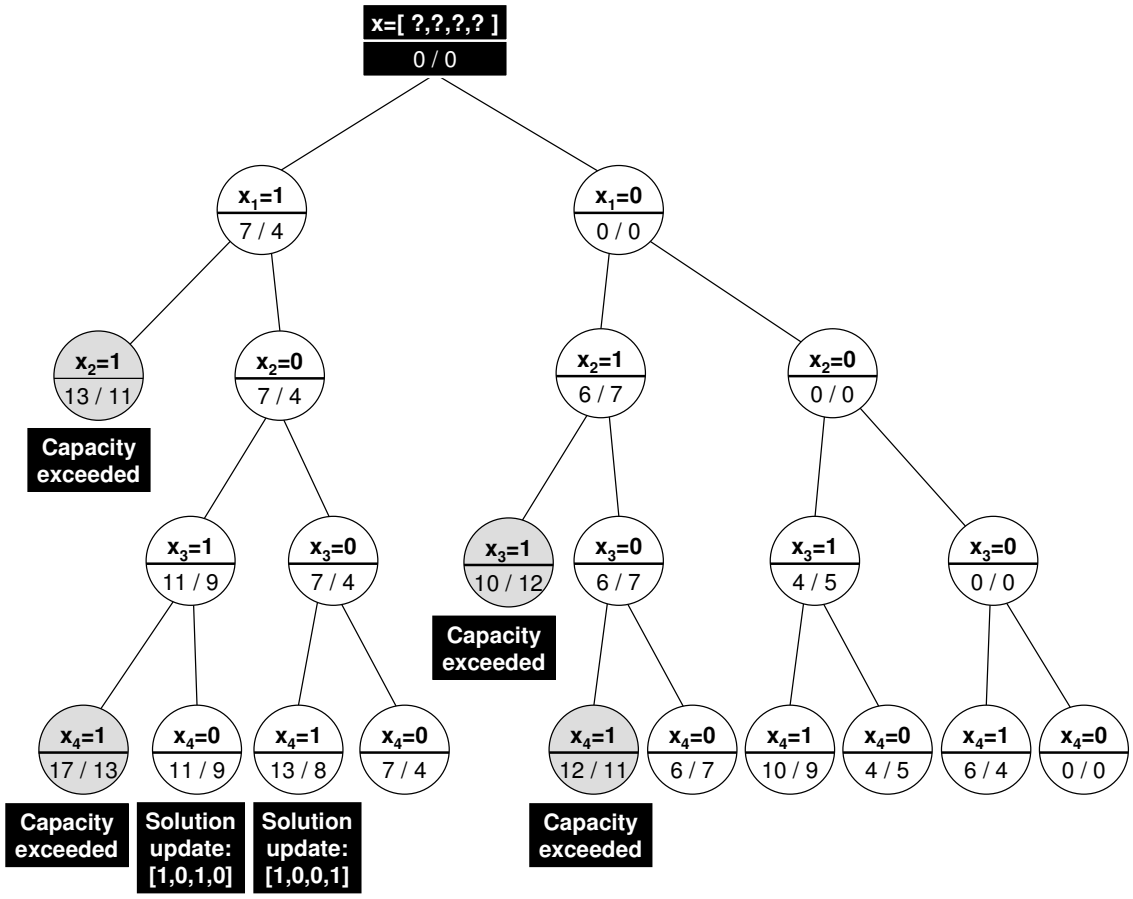


Figure 3.15 - Search tree for the knapsack problem example

After the second update, the stored solution vector is $[1,0,0,1]$, which means that items 1 and 4 are to be selected. The corresponding profit and weight is 13 and 8, respectively. The search continues, until the whole search tree gets traversed, but no other leaf provides a more profitable set of items.

3.3. Other selected algorithms

Two other classical problems, which are not usually addressed with backtracking search algorithms, have been selected for this research: sorting; and the calculus of the greatest common divisor.

3.3.1. Sorting

Sorting consists of rearranging the elements of a given set using a specific ordering criterion. For instance, a given group of people can be sorted by alphabetic order of their first names, or by numerical order of their ages, etc. However, for implementation purposes, any sorting criterion is usually converted to numerical ordering.

Sorting facilitates the search of elements within the given set and it is therefore very often executed within a broad range of more complex algorithms. In fact, its applicability is so wide that the list of sorting algorithms keeps growing, as designers try to reach higher efficiency. Some of the most famous are Bubble sort [Astrachan03], Insertion sort [Astrachan03], Binary Tree sort [Wirth86], and Merge sort [Kernighan88].

Sorting based on a binary tree (also known as *binary tree sort*) was selected for comparison of alternative recursive and iterative implementations. This algorithm includes the following two stages:

1. Construction of an ordered binary tree, using the numerical value of each given element to form a tree node. The node insertion is performed in such a way that:
   a. the left subtree of any node contains only values less than the node's value;
   b. the right subtree of any node contains only values greater than or equal to the node's value.
2. Retrieval of every tree node using in-order traversal, *i.e.*, with the root node as the starting point, performing the following three steps:
   a. in-order traversing the left subtree, if there is one;
   b. retrieving the current node;
   c. in-order traversing the right subtree, if there is one.

Figure 3.16 illustrates, step by step, the tree construction stage of the algorithm. The given sequence of integers used as an example in this illustration is 4-2-9-5-9. Black

circles are used to identify the tree node inserted at each step, while white circles represent previously inserted nodes. Symbols < and ≥ can be used to easier understand the location of each inserted node.

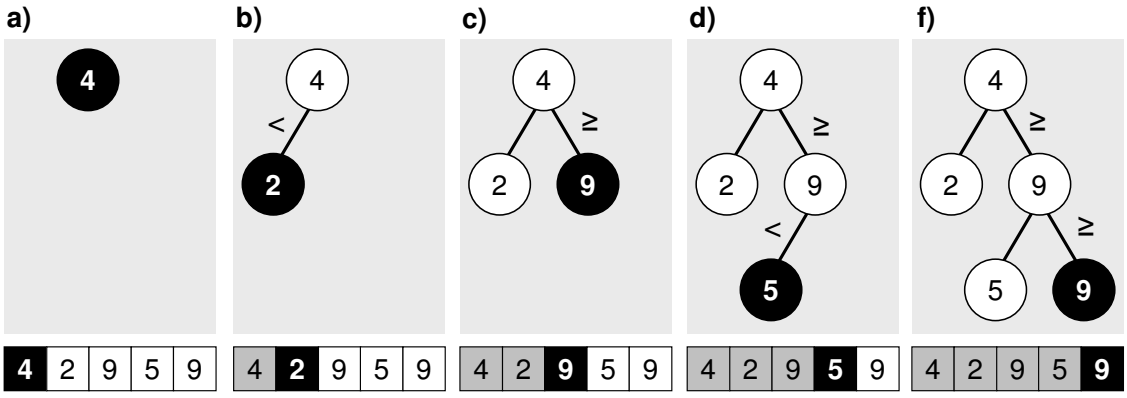


Figure 3.16 - Constructing an ordered binary tree

The tree node in-order retrieval stage of the algorithm is illustrated in Figure 3.17. The ordered binary tree used for this illustration is the one constructed in the previous example (see Figure 3.16-f). Within each step illustration (Figure 3.17-a to Figure 3.17-f), black circles identify the tree node being retrieved, while the resulting ordered sequence is updated below the tree. Gray circles represent nodes already retrieved, while white circles correspond to nodes yet to be retrieved. With the fifth step (see Figure 3.17-f), the last integer is retrieved, thus completing the sorted sequence: 2-4-5-9-9.

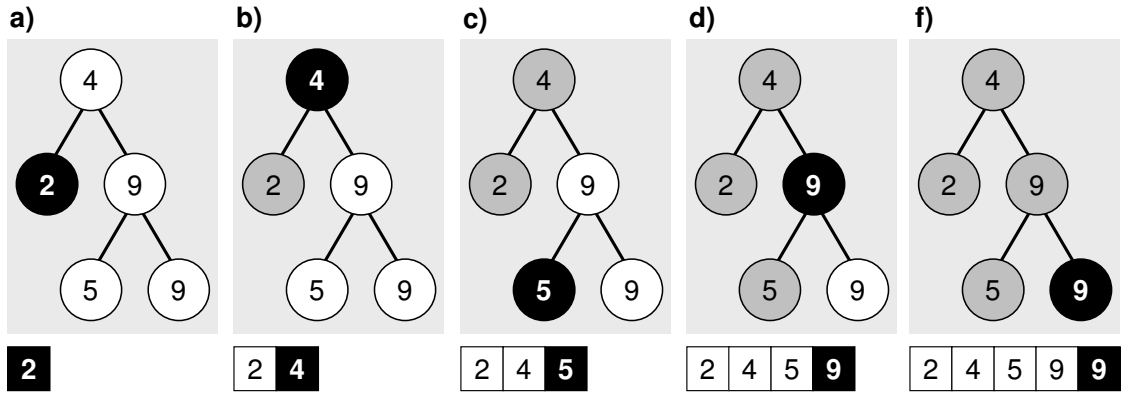


Figure 3.17 - Retrieving ordered binary tree nodes

3.3.2. The greatest common divisor

The *greatest common divisor* (*GCD*) of 2 integers *a* and *b*, not both zero, is written as $\gcd(a,b)$ and defined to be the largest integer that divides both *a* and *b* with no remainder [Knuth97, Abelson96]. This definition can be extended to three or more integers, in which case the greatest common divisor is the largest integer that divides each of them with no remainder. The greatest common divisor is also known as greatest common factor and as highest common factor.

Applications of the GCD can be found in rational arithmetic and in multiple-precision arithmetic [Knuth97].

The GCD of two integers can be efficiently calculated using Euclid's algorithm, which is over two thousand years old [Knuth97]. Figure 3.18 describes Euclid's algorithm both iteratively and recursively, by means of pseudocode. Keyword 'mod' represents the *modulo* operation, which calculates the remainder of dividing the left operand by the right operand. For instance, expression '10 mod 3' evaluates to 1, as the remainder after dividing 10 by 3 is 1.

a)

```
it_gcd(a, b)
{
   while b ≠ 0
   {
      temp = b
      b = a mod b
      a = temp
   }
   return a
}
```

b)

```
rec_gcd(a, b)
{
   if b ≠ 0
      return rec_gcd(b, a mod b)
   else
      return a
}
```

Figure 3.18 - Pseudocode for calculating the GCD of two integers iteratively (a) and recursively (b)

3.4. Conclusion

A primary objective of this research is the comparison and evaluation of alternative recursive and iterative implementations for different algorithms. For this purpose, it

was necessary to select a set of algorithms which can be described both iteratively and recursively.

A class of algorithms which is often implemented on the basis of recursive descriptions is *backtracking search algorithms*. Such algorithms can be developed on the basis of a generic algorithmic structure which expresses a recursive procedure to be executed at every node of the relevant search tree. The process starts by simplifying the current problem instance using a set of *reduction* operations. When no further reduction is possible, a *resolution test* is performed. In case the problem has been solved, the process ends, and the solution is provided. Otherwise, a *solvability test* is carried out. In case the problem is found unsolvable, the process ends with no solution. Otherwise, the solver might have to try alternative paths in the search tree in order to check whether there is one which leads to a solution. The set of operations that determines which path to follow is called *selection*. When a chosen search path fails to provide a solution, the algorithm backtracks and selects another one, if available.

A common data structure can be used to specify problem instances for different backtracking search algorithms. Most combinatorial search problems can be expressed in several equivalent mathematical formulations based on different standard data structures. Matrices can be stored and processed easily in both software and hardware and most combinatorial search problems can efficiently be formulated over matrices. For these reasons, and without loss of generality, we have selected matrices as the data structure to be used for specifying the combinatorial problem instances, when applicable.

In order to compare and evaluate recursive and iterative implementations in hardware, six problems have been considered for experiments: 1) set covering, 2) Boolean satisfiability, 3) graph coloring, 4) knapsack, 5) tree-based sorting, and 6) calculating the greatest common divisor.

We have demonstrated in detail the applicability of backtracking search algorithms operating over matrices for solving the first four selected problems. For each of the four corresponding algorithms, a particular method for converting problem-instances to matrices is used. The same applies for the reduction rules, selection rules, resolution test, and solvability test.

Algorithms for tree-based sorting and the calculus of the greatest common divisor have also been presented and demonstrated in detail. For solving these problems, specific data structures were used.

4. Software/Hardware Tools for Prototyping and Experiments

This chapter describes the developed prototyping system and software tools that enable experiments with hardware accelerators and comparisons of alternative recursive and iterative algorithms to be carried out easier and more efficiently. The system is based on the DETIUA-S3 prototyping board with wired and wireless interface with a host computer developed at the department of Electronics, Telecommunications and Informatics of Aveiro University, as well as on software tools proposed and implemented within this thesis. The software tools establish user-friendly interface with the board (including wireless interaction) and provide high-level support for many different experiments required for the considered hardware accelerators, such as the developed virtual peripheral devices, modules for software/hardware co-simulation which simplifies hardware/software partitioning, procedures which extract intermediate results for analysis, etc. A more advanced technique assumes the application of the developed tools through the Internet in such a way that allows different users to configure and to interact with the remotely accessed prototyping board. Although this work was not initially planned, many tools have been developed, implemented and tested, which permits to conclude that the proposed system can efficiently be used for remote interactions. In the end, an overview of the advantages and potential applications of the prototyping system is provided.

4.1. Prototyping system

This section presents the set of developed hardware/software components, namely an FPGA-based prototyping board, the board-targeted software, and useful tools mainly needed for fulfilling the objectives of this thesis.

4.1.1. The DETIUA-S3 FPGA-based prototyping board

The FPGA-based prototyping board named DETIUA-S3 [Almeida08] (see Figure 4.1) has been developed in such a way that permits to provide the following main features:

- Programming and data transferring from a PC (personal computer) through USB or Bluetooth interface;
- Powering the board through a USB port or using an external power source;
- Keeping bitstreams for the FPGA in a flash memory, which allows to use the board as an autonomous device, without any connection to a PC, and only external powering has to be provided;
- Keeping more than one bitstream in the flash memory for dynamic reconfiguration of the FPGA. The capacity of the selected flash memory permits to store up to 8 bitstreams. This is very practical not only for run-time reconfiguration but also for verification of different types of alternative and competitive implementations;
- User-friendly interface (see section 4.1.2) for programming the board and data exchange with a PC;
- Expansion headers for interacting with application-specific, externally connected devices.

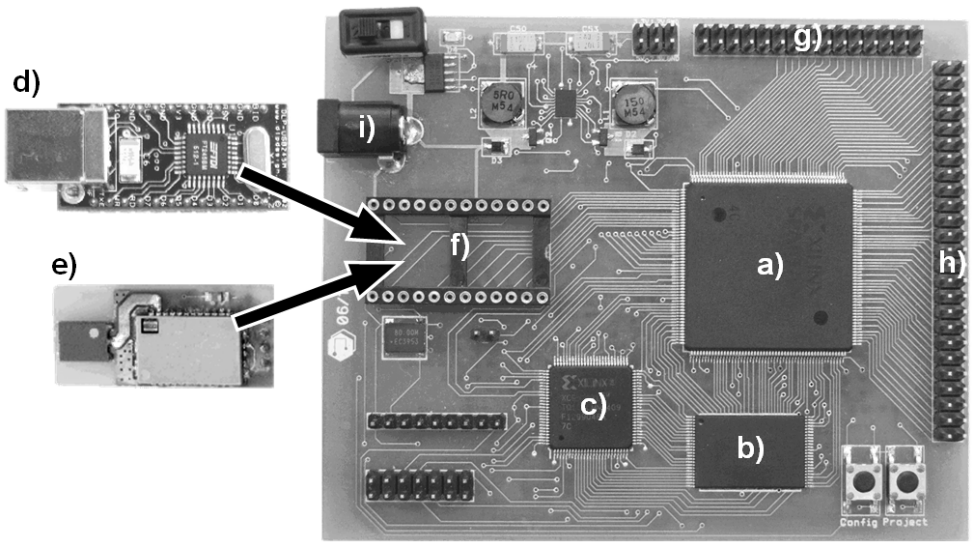


Figure 4.1 – The DETIUA-S3 board with interface module alternatives

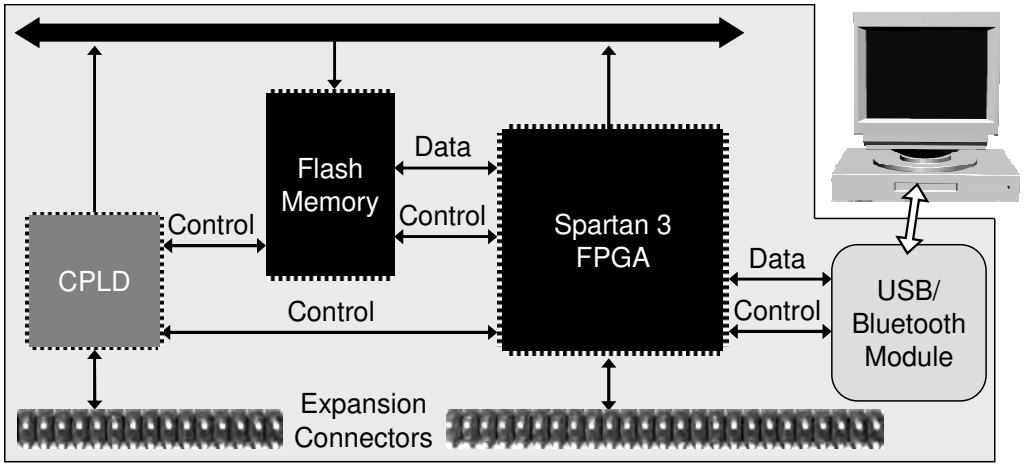


Figure 4.2 – The DETIUA-S3 board basic architecture

The basic architecture outlined for DETIUA-S3 (see Figure 4.2) includes the following main components:

- An FPGA of Xilinx Spartan-3 family (see Figure 4.1-a), namely XC3S400 [Xilinx], based on 90 nm technology, with 400.000 system gates, 288 Kb of block RAM, 16 embedded multipliers and 264 inputs/outputs;

- A flash memory of AMD (see Figure 4.1-b), namely Am29LV160D [AMD], divided into three logical sections, as shown in Figure 4.3. The first section contains the *default bitstream*. This bitstream has to be pre-loaded to the FPGA in order to allow the following set of operations: 1) transferring an application-specific bitstream to the second logical section; 2) erasing flash memory sectors; 3) transferring data from a host device to the third section of the flash memory and vice versa. This technique has already been used in Trenz prototyping boards [Trenz]. The second logical section stores an application-specific bitstream (*user bitstream*) for subsequent quick loading into the FPGA (pressing the 'project' pushbutton available on the board). The third memory section enables the designer to keep additional bitstreams for configuring the FPGA or any arbitrary user data;

- A CPLD (Complex Programmable Logic Device). This component (see Figure 4.1-c) is needed for controlling the flash memory and pushbuttons assembled on the board, because the FPGA cannot execute these functions during configuration. The CPLD also generates an initial reset signal for FPGA circuits as soon as a new configuration is completed;

- Either of two available interface modules, USB (see Figure 4.1-d) and Bluetooth (see Figure 4.1-e), can be plugged into the board interface socket (see Figure 4.1-f). The selected interface can be established with any compatible device, allowing for any required bi-directional data exchange (for instance, to supply the board with user bitstreams). Connecting the board to a computer using the USB module eliminates the need for another power source and provides higher bandwidth, while Bluetooth has the advantage of a wireless communication and portability (a small battery-based source can be used for powering);

- Expansion connectors (see Figure 4.1-g and Figure 4.1-h) permit to attach any application-specific external hardware, such as mini boards with extra components, human interaction peripherals and even other FPGA-based boards (other DETIUA-S3 boards, for instance).

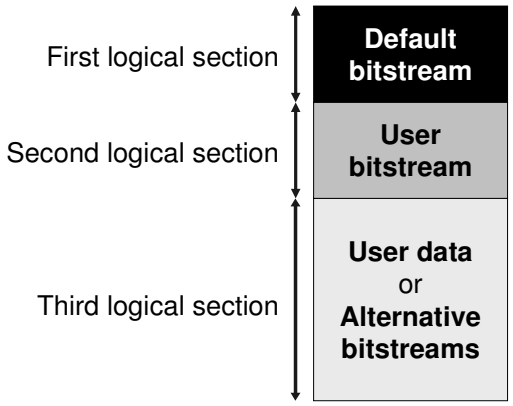


Figure 4.3 - Logical division of the flash memory in DETIUA-S3

4.1.2. The PBM system software for DETIUA-S3

A software application called PBM (Prototyping Board Manager) has been developed to provide important functionality with respect to DETIUA-S3 with a convenient user-friendly interface. Basic PBM tools, possible workflows, and system integration scenarios that are achievable with DETIUA-S3 and PBM are revealed in Figure 4.4 [Almeida06].

The most basic function of PBM is uploading a user bitstream into the second section for quick configuration of the FPGA (by pressing the 'project' pushbutton). This technique is the most appropriate to integrate design workflows for single-bitstream projects.

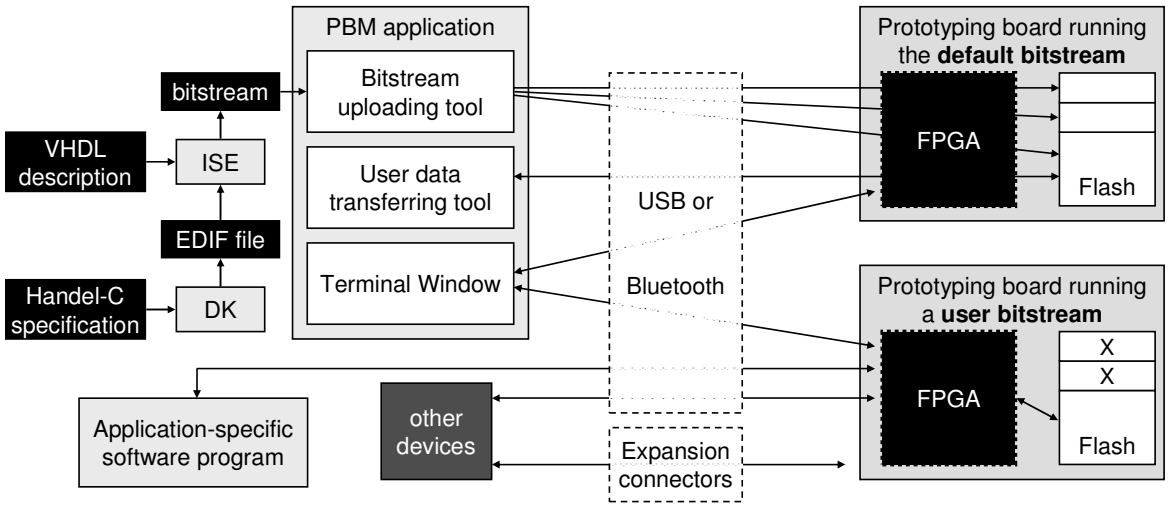


Figure 4.4 - Examples of DETIUA-S3 and PBM prototyping capabilities

PBM also features a terminal window for run-time data exchange between the user and the prototyping system, thus constituting an integrated input/output peripheral, which is very convenient for project monitoring and testing.

A more advanced function allows to send multiple bitstreams (let us refer to them as *alternative bitstreams*) and to store them in the third logical section of the flash memory (see Figure 4.3). The latter is logically divided in six predefined subsections for storing alternative bitstreams, which can be used for FPGA reconfiguration by the following means:

a) Attaching a simple additional switch through expansion connectors and pressing the 'project' pushbutton, the board reconfigures the FPGA with the bitstream that is stored in the subsection indicated by the switch;

b) Any circuit running in the FPGA can send to the CPLD a request for run-time reconfiguration, using techniques such as those described in [Shirazi98, Sklyarov98], indicating which bitstream has to be loaded.

c) Sending any required bitstream through expansion connectors to another prototyping board in order to configure its FPGA.

The software application includes a user manual in English and Portuguese (available online [Pimentel]) which gives detailed information on how to take full advantage of all the available functionality.

In order to be able to work with PBM, either through USB or Bluetooth interface, a user must first press the board's 'configuration' button. This operation loads the default bitstream from the first logical section of the flash memory into the FPGA, configuring it to establish the protocol which PBM uses to manage the board. Each function available to the user generates a sequence of basic operations supported by this protocol, such as: erase a sector, read from a specified range of addresses, or write a sequence of bytes.

PBM also provides compatibility with new boards which may include different FPGAs, flash memories, etc. The basic rules to develop PBM-compatible prototyping boards are provided in the user manual [Pimentel].

4.1.3. Remote interaction

A conceptual framework called *Remote Lab* [Pimentel08] has been proposed, enabling a remote interaction with DETIUA-S3. We assume that the following conditions have to be satisfied:

a) DETIUA-S3 is connected through either USB or Bluetooth interface to a server PC;

b) The server PC is running PBM in *Remote Lab Server Mode*;

c) The client PC (on which the remote user is) can reach the server PC through the Internet;

d) The client PC is running PBM in *Remote Lab Client Mode*.

The developed software tools provide remote users with most of the PBM functionality through the Internet (see Figure 4.5). In addition, co-simulation tools have been developed, enabling remote users to construct digital systems in such a way that they are partially implemented in FPGA and partially modeled in software of a user computer. The developed software enables only one user to work with the board at the same time. As soon as communication between a user's computer and the board is terminated, the latter becomes available for another user. Such system is very helpful for educational process.

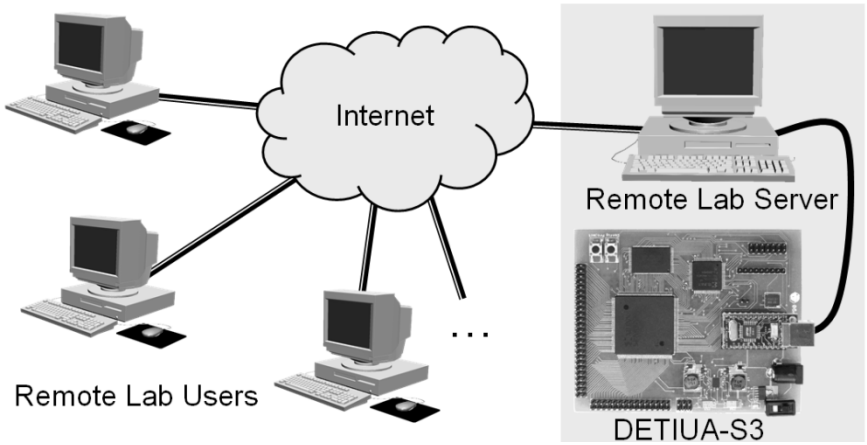


Figure 4.5 - Remote access to DETIUA-S3

Potential applications of this remote interaction system are the following:

- Remote design;
- Using virtual peripheral devices [Sklyarov08a];
- Co-simulating the developed and virtual components.

Additional details will be given in section 4.2.

4.1.4. Hardware/software co-simulation

As a rule, the considered hardware accelerators are parts of larger circuits which, in combination with some other components, make up a complete system. Traditional approaches to the design of such circuits, such as top-down, bottom-up and combined, assume decomposition of the entire system into sub-systems, which at different levels of decomposition are of varying complexity. For many practical problems it is necessary to examine the communication between relatively autonomous sub-systems in order to assess the characteristics of the system, such as the correctness of the functionality, the adequacy of the performance, the accuracy of execution, and so on. Note that this assessment has to be done at a point when all the components of the system have not yet been implemented. Paper [Sklyarov02b] proposes a combined software/hardware model of a system that consists of a control part that is mainly implemented in hardware (in an FPGA) and a datapath that is modeled partially in hardware and partially in software. The hardware and software components communicate through a pre-established interface.

The reasons for choosing such decomposition are the following:

1. Very often, a control algorithm (such as a micro-program) operates primarily with individual bits of data in such a way that it analyzes a predefined subset of bits from a given set (such as the flags of a processor) and generates control signals that can also be considered as a subset of a set of individual bits (such as the signals that affect data flow). These operations can be implemented in hardware much faster than in software. Note that control can be considered at different levels. A high level control is usually implemented in software. A low-level control might be realized more efficiently in hardware.

2. An execution unit deals mainly with words of data that have a predefined size. Operations on these words can be performed with the aid of general-purpose processors. For many practical applications, implementing execution units in FPGAs is very profitable because it allows the speed of data flow to be accelerated. However, a software model of a datapath enables us to estimate how profitable it may ultimately be, and provides the information needed to decide whether to partition an execution unit into a software component plus hardware. Note that this question is especially interesting in the educational context.

3. Using the proposed approach makes it possible to implement a distributed control, which combines autonomous control circuits that directly affect the external FPGA devices, and a more sophisticated control that requires the use of a datapath emulated in software. Such control devices are widely needed in embedded systems requiring the considered hardware accelerators.

4. Reconfigurable hardware can improve the performance of datapath by using parallelism and pipelined execution. The proposed approach does not exclude this possibility. We have already mentioned that the boundary between hardware/software components is adjustable and we can combine a hardware implementation of highly parallel parts of an execution unit with the remainder implemented in software.

5. A control unit can be implemented as a virtual device. As a result, it can have dynamically modifiable functionality. One interesting approach in this context is to implement only part of a control algorithm in the dynamically modifiable area of an FPGA, and to reload other parts of the control algorithm when required, with the aid of software tools. The developed software tools for the DETIUA-S3 board provide support for such opportunity.

We consider a similar technique but will not restrict types of circuits implemented in software and hardware (such as control and execution units).

A top-level architecture of an FPGA-based virtual system [Sklyarov02b] consists of three primary components: virtual hardware (modeled in software), physical hardware (implemented in an FPGA), and a software/hardware interface. This architecture permits to examine various alternative implementations by shifting the boundary between the hardware and software parts, *i.e.* by examining the systems with more hardware and less software or vice versa. This is especially important for hardware accelerators because it permits to select the most favorable fragments of the studied algorithms requiring acceleration and also the relevant increase in performance.

A software/hardware interface enables connections and the interchange of signals to be established between hardware/software components shown in Figure 4.6.

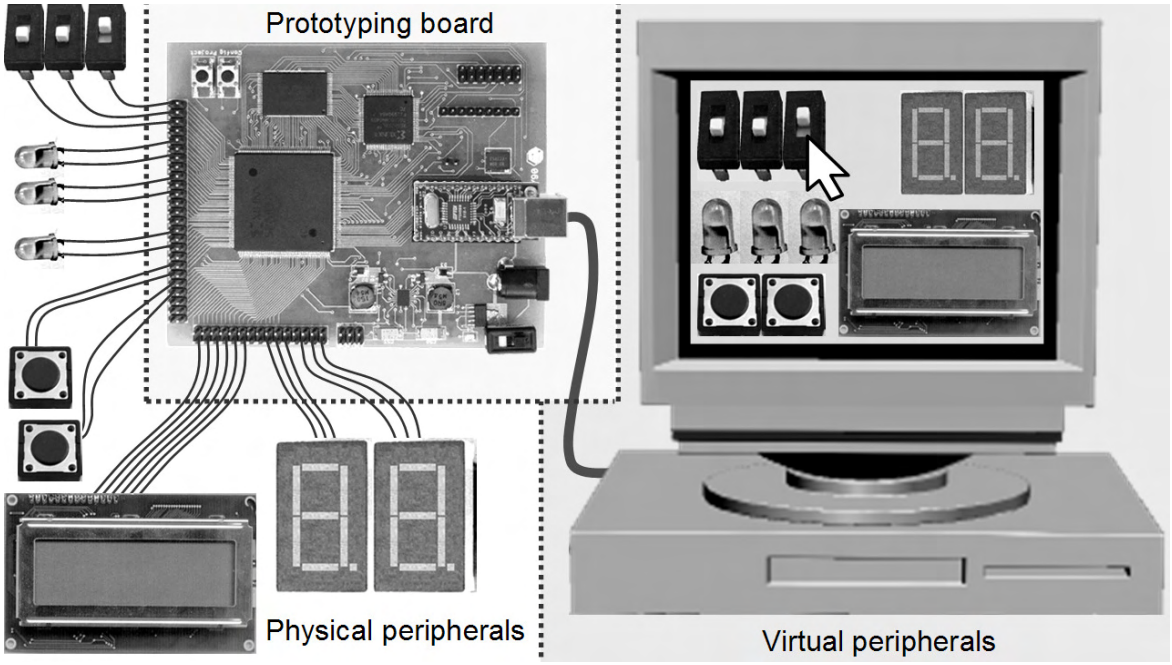


Figure 4.6 – Demonstrating virtual and physical peripheral devices

In our case, the software models have been developed on a PC. The hardware part has been implemented in DETIUA-S3. The interface was partially implemented in software and partially in an FPGA, and it provides for the transmission of signals through either USB or Bluetooth interface. The following subsections (0 and 4.1.4.2) present different proposed and implemented tools which provide support for the hardware/software co-simulation considered above.

4.1.4.1. Interaction with virtual peripheral devices

One of the most important components of the hardware/software co-simulator is a virtual visual sub-system [Sklyarov08a], which enables the designer to verify the functionality of the developed system in a visual mode using the host computer monitor (see Figure 4.6). It is important that the virtual visual environment allows the creation of a vast variety of virtual peripheral devices for the FPGA-based prototyping core considered in section 4.1.1. Indeed, it permits to visualize and to virtually attach peripheral devices, providing data input and output and modeling typical (push buttons, LCD, etc.) and application-specific (stack watchers, function calls watchers, etc.) devices. The number of potential devices is, indeed, unlimited because any newly required one can be modeled and included in the existing virtual peripherals library.

A system depicted in Figure 4.6 displays virtual peripheral devices (namely pushbuttons, an LCD panel, a segment display, and LEDs), and communications with such devices are organized much like communications with physical peripheral devices. The proposed technique can be very efficiently used in the scope of design space exploration. For example, there are many practical applications that require solving combinatorial search problems. It is possible to design a reusable circuit [Sklyarov08a] that might be customized for solving many problems from the area of combinatorial computations. Such a reusable circuit can be entirely modeled in software. However, it may be beneficial to implement this circuit in FPGA. One of the easiest ways is a sequential conversion of the software model to hardware implementation in such a way that the required hardware is incrementally extended replacing more and more software. This, in particular, significantly simplifies testing and debugging of hardware circuits. Besides, the considered hardware/software architecture opens practically unlimited capabilities for experiments in the scope of design space exploration. For example, we can:

- Use debugging facilities of software in order to test different operations available for the circuit;

- Verify different algorithms step by step, either in software or in hardware, assessing the results in a convenient visual mode (see example in Figure 4.6);
- Provide hardware/software partitioning and execute algorithms partially in software and partially in hardware, changing the boundary between software and hardware. This is indeed design space exploration because we can check if hardware implementation is really capable to improve performance and other characteristics of the system.

The described system for hardware/software co-simulation has been implemented using the following technique. DETIUA-S3 has two expansion buses connected to the FPGA (see Figure 4.1-g and Figure 4.1-h) with a total of 80 connectors that can be used to attach physical external devices. The developed agent module routes the corresponding 80 circuit input/output signals from/to a computer, instead of the board expansion connectors, in order to establish full interaction with the virtual user peripheral components.

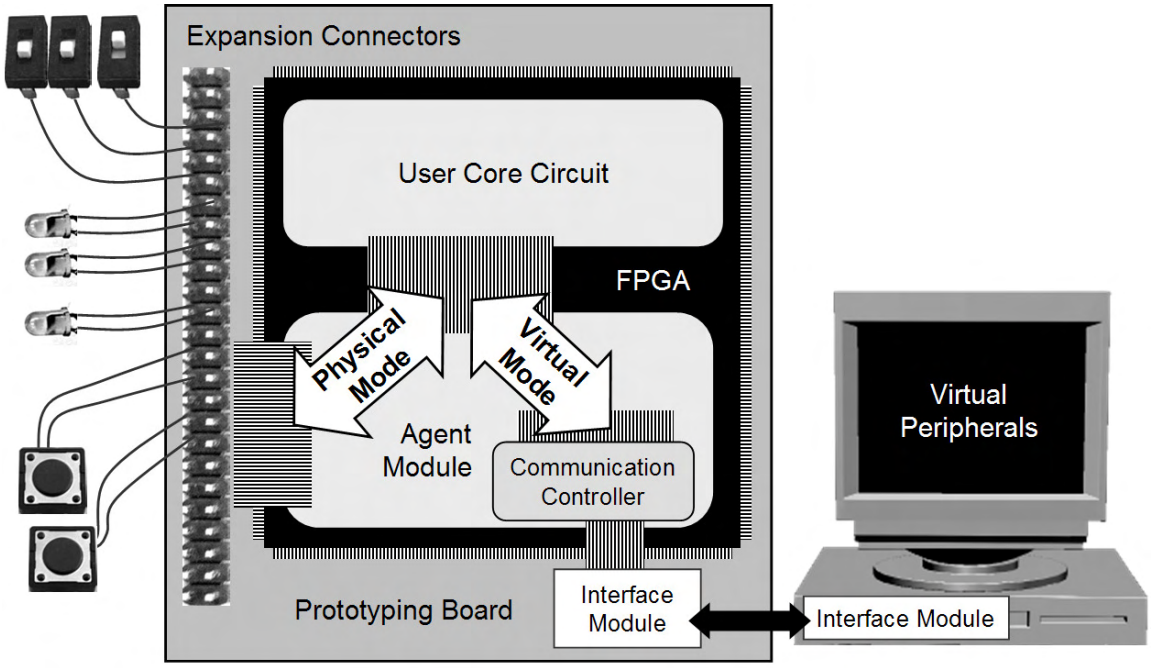


Figure 4.7 - Signal routing with the agent module

To allow user projects to interact with both physical and virtual peripherals, the agent module has two running modes: physical and virtual. In *physical mode*, signals are simply transferred to/from the expansion connectors. In *virtual mode*, signals pass through the interface (either USB or Bluetooth) using an exchange protocol created for that purpose. Figure 4.7 illustrates these routing capabilities. The agent module starts

in physical mode and switches to virtual mode in case the software application requests it through the established interface.

For a more complete understanding of how the designed tools work, let us now look at the developed object-oriented classes that provide support for virtual peripheral devices (see the class diagram depicted in Figure 4.8).

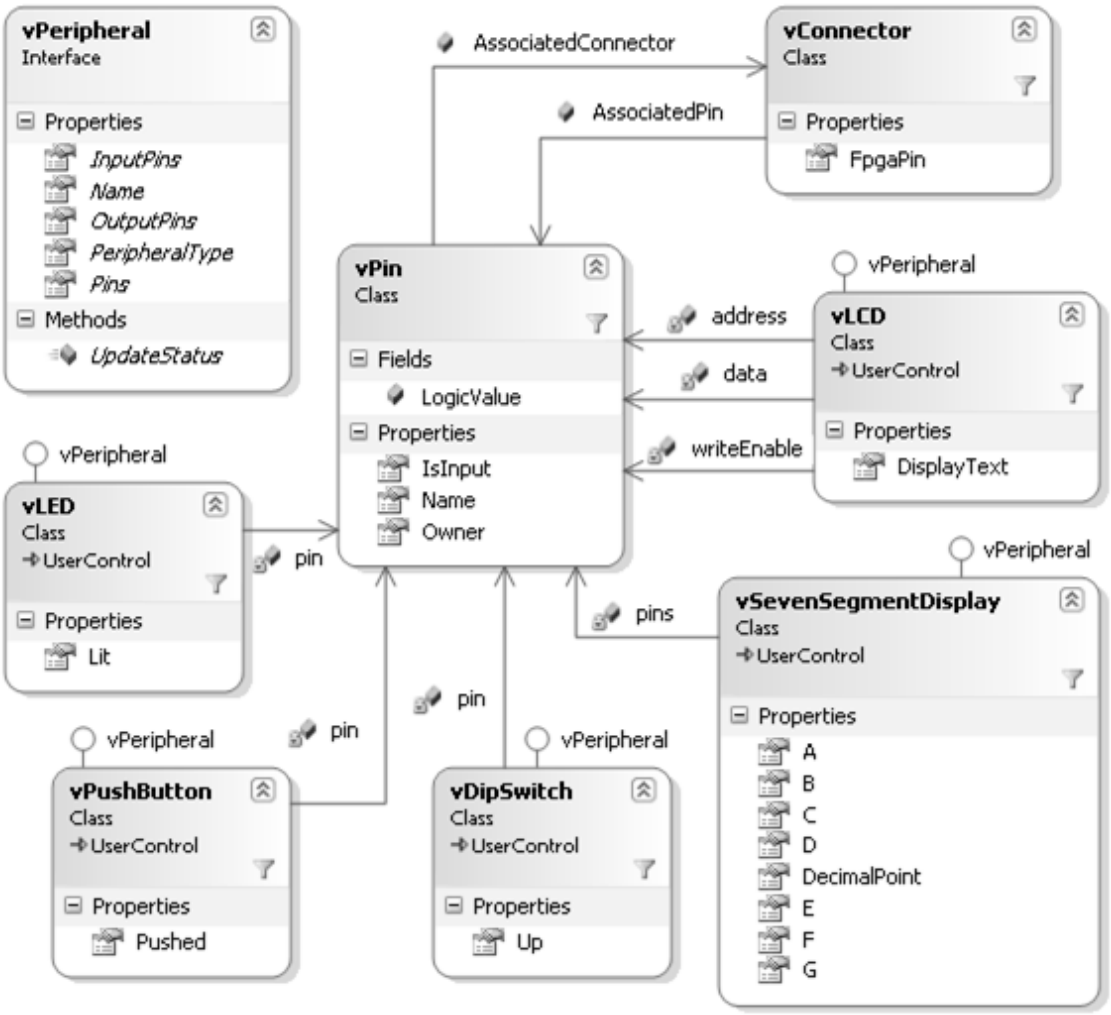


Figure 4.8 - Partial class diagram used in the software for running virtual peripheral devices

The functionality of the most frequently used peripherals is emulated by object-oriented classes which have been described in C#. The developed application can manage any required number of instances of each virtual peripheral class.

All classes for emulating physical peripheral devices conform to the following:

a) They are derived from the class *UserControl* (from the .NET library), providing functions for graphical visualization based on images of the corresponding physical devices;

b) They implement the *vPeripheral* interface (see the top left-hand corner of Figure 4.8) which contains a set of methods common to all virtual peripherals (such as *GetName*, *GetOutputPins*, etc.), thus taking advantage of the object oriented paradigm.

Two special classes named *vPin* and *vConnector* (respectively at the center and top right corner of Figure 4.8) are the basis for signal propagation. Each of the 80 connectors mentioned before is represented by a *vConnector*; and every pin in the set of real peripherals is represented by a *vPin*. Before running the project, the user must associate *vPins* with *vConnectors*. Such association corresponds to connecting real peripheral pins to DETIUA-S3 expansion connectors, according to the scenario the user needs to emulate.

When the project is running, any signal change in a virtual peripheral output (caused for instance by user action) triggers a signal update in the associated *vPin*. Such a change is propagated to the associated *vConnector*, updating its value. The continuous cycle of signal exchange between the application and the agent module keeps sending the last signal value stored in each *vConnector*, through the interface module. Every time a signal exchange packet reaches DETIUA-S3, the agent module updates the user circuit inputs with new signal values.

The user circuit output signals are propagated basically in the same way but in the opposite direction. When a signal update reaches a *vPin* for input, the application invokes the *UpdateStatus* method on the owner *vPeripheral* and the user can visualize the resulting feedback on a monitor screen.

The developed tools can easily be combined with remote interaction through the Internet, which permits to execute similar functions in a distant mode. The remote interaction requires constant signal exchange between the application and DETIUA-S3 over the Internet. For this purpose, a dedicated PBM tool establishes a new TCP (Transmission Control Protocol) connection with the remote client through which all data (including PBM operation commands) sent by the board are forwarded to the remote client and vice versa (see Figure 4.5).

Note that the components emulated with this framework are peripherals for interaction with humans. The delays inherent in this kind of interaction are significantly long and irregular, considering the response time of digital circuits. For this reason, the delay overhead which is inflicted by the whole signal propagation process can be considered tolerable. In fact, the signal propagation delay overhead ranges from tens of milliseconds, when the user's computer is directly connected to the board, to a few seconds, when remotely connected across the Internet.

4.1.4.2. Reprogrammable FSM-based architecture

This section describes the model proposed for validation of different types of interactions between the execution and control units. Paper [Sklyarov02b] proves that such interaction is needed for many practical applications and discusses a number of examples. The main contribution is in the adaptation of a reprogrammable FSM model described in [Sklyarova02a].

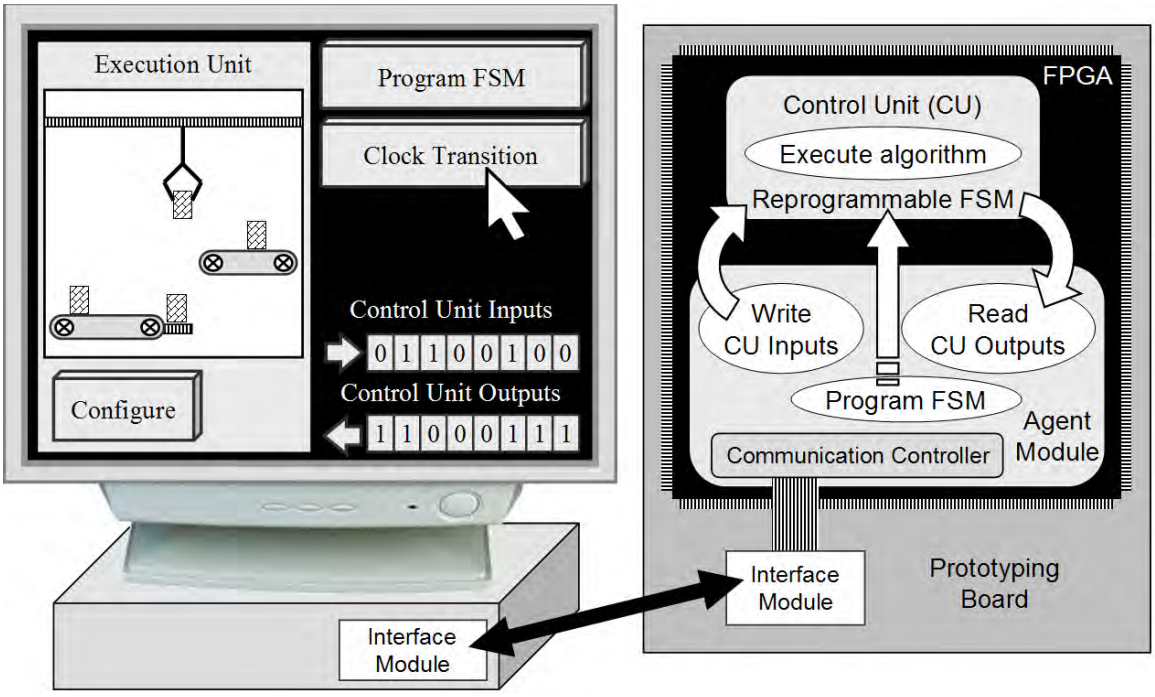


Figure 4.9 – Using the proposed reprogrammable FSM-based model

We assume the following characteristics for the designed system (see Figure 4.9 as a potential example):

a) Control units are modeled by a reprogrammable FSM implemented in hardware;

b) Execution units and application-specific peripheral devices are modeled in software;

c) Input, output, reset, and clock signals of the control unit are propagated at each clock transition triggered by the user, allowing for step-by-step control and monitoring of the developed system, from the computer.

The reprogrammable FSM can be implemented with the aid of a hardware template (HT) [Sklyarov06d]. An *HT* is a circuit that contains elements with functions that can be changed and which are initially undefined. All the external connections of elements are fixed and they cannot be modified. The customization of the HT is carried out by programming (reprogramming) its elements with changeable functions. In order to construct the HT, it is necessary to estimate all the likely constraints for future applications. In other words, we should define a class of applications and the constraints for that class.

For an FSM, these constraints might be the maximum numbers of the input variables (L_{max}), the output variables (N_{max}), the states (M_{max}) and the transitions from any state; also the maximum size of state codes (R_{max}), etc. Figure 4.10 shows an example of an HT for an FSM.

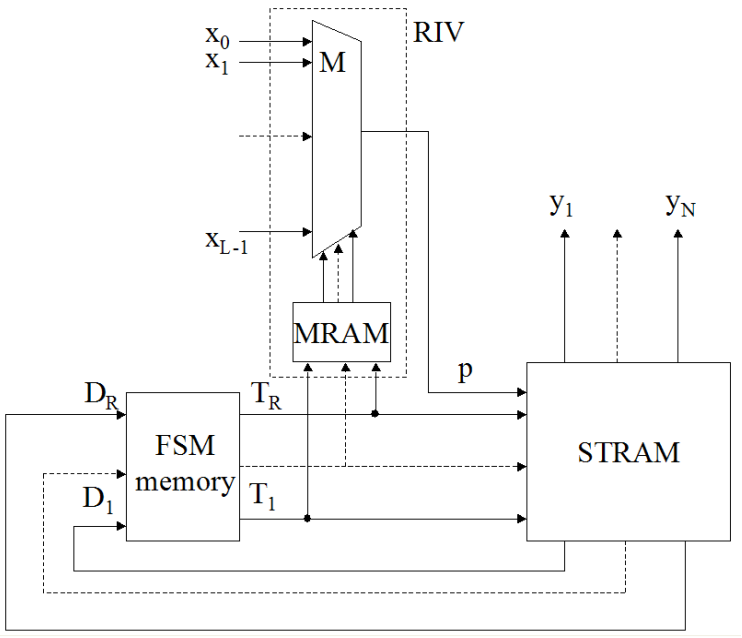


Figure 4.10 – An example of a hardware template

It is composed of two RAM blocks, an FSM memory and a multiplexer. MRAM permits any input $x_l \in \{x_0,...,x_{L-1}\}$ of the multiplexer to be selected in such a way that $p=x_l$ and value l can be specified by MRAM. For example, if the FSM codes are binary values of state subscripts and if input x_9 affects transitions from state a_5 then, at the address 101, MRAM forms outputs 1001 which control the 16:1 multiplexer. If, for another application, the transitions from a_5 are affected by x_3 then at the address 101, MRAM has to form outputs 0011. Clearly we can provide any correspondence between states $a_0,...,a_{M-1}$ and inputs $x_0,...,x_{L-1}$. The RIV (Replacement of Input Variables) block shown in Figure 4.10, in the dashed rectangle, permits any variable from set $X=\{x_0,...,x_{L-1}\}$ to be replaced by a single variable p. State transition RAM (STRAM) enables us to generate codes for the next states and outputs. For example, if R=4 and we have state transitions $a_{10}x_2 \Rightarrow a_7$ and $a_{10}\overline{x_2} \Rightarrow a_4$ then, at the address 10101, STRAM contains the code $(D_1,...,D_4) = 0111$ and, at the address 10100, the code $(D_1,...,D_4) = 0100$. Obviously any subset of output signals $y_1,...,y_N$ can be arbitrarily generated in any state transition.

By modifying the contents of MRAM and STRAM we can implement any desired FSM behavior within the scope of the constraints predefined for the HT. In case of the HT depicted in Figure 4.10, there is a very significant constraint: any state transition can only be affected by a single input variable. Different ways to solve this problem and many details regarding RAM-based reprogrammable FSMs are considered in [Sklyarov06d].

The developed hardware/software tools can be employed in different areas enabling the designers to partition the developed system in such a way that one part of the system will be implemented in reconfigurable hardware and another part will be modeled in software. Suppose we need to design a reprogrammable FSM which implements different algorithms over ternary vectors whose elements have one of three possible values: 1, 0 and – (*don't-care*). Different algorithms permit to execute such operations as: testing if the given vector contains N successive 1s (0s, *don't-cares*); if the vector does not have values 1 (0s, *don't-cares*); if the number of 1s in the vector is greater than the number of 0s, etc. Such operations are frequently required for numerous combinatorial search problems [Skliarova06b] and we would like to examine the execution time for different algorithms and the ability of FSM to be efficiently reprogrammed (using, for example, methods described in [Sklyarov02a]). Suppose an initial vector has to be entered from either pushbuttons or DIP switches and the results of the selected operation together with the execution time have to be

displayed on an LCD. The designed circuit includes the following two primary blocks (see Figure 4.11): the reprogrammable FSM and the handler making it possible to customize the FSM in such a way that enables the desired algorithm to be realized.

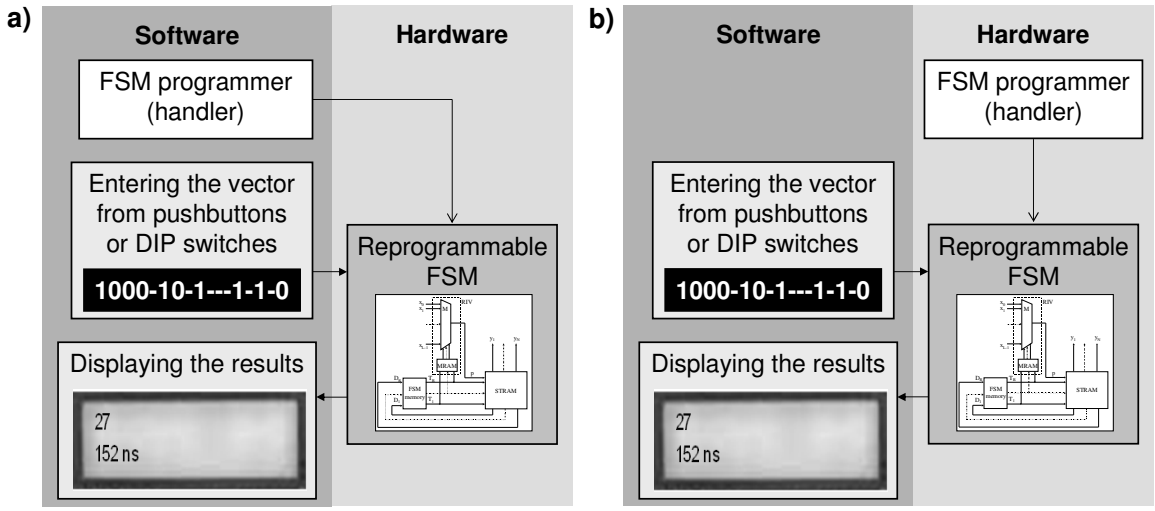


Figure 4.11 – Incremental circuit design with the proposed technique

Suppose that in the beginning (see Figure 4.11-a) the FSM is implemented in FPGA and the handler is modeled in software of a host computer. After the FSM has been tested, both blocks (*i.e.* the FSM and the handler) can be implemented in FPGA (see Figure 4.11-b). Thus, the considered technique enables the circuit to be designed incrementally. Dependently on the availability of peripheral devices (such as the LCD shown in Figure 4.11), either physical or virtual interaction with such devices can be employed (see also Figure 4.6).

It should be noted that the described technique is very useful not only for accelerators considered in this thesis. It is applicable to many other applications outside the thesis area and thus, it is rather universal. Let us consider an example from [Pimentel08] in which the developed tools are used to simulate an assembly line whose basic functionality is depicted in Figure 4.12. Incoming items, which are brought by the left-hand side conveyer, must be passed onto the right-hand side conveyer with the aid of a magnetic crane.

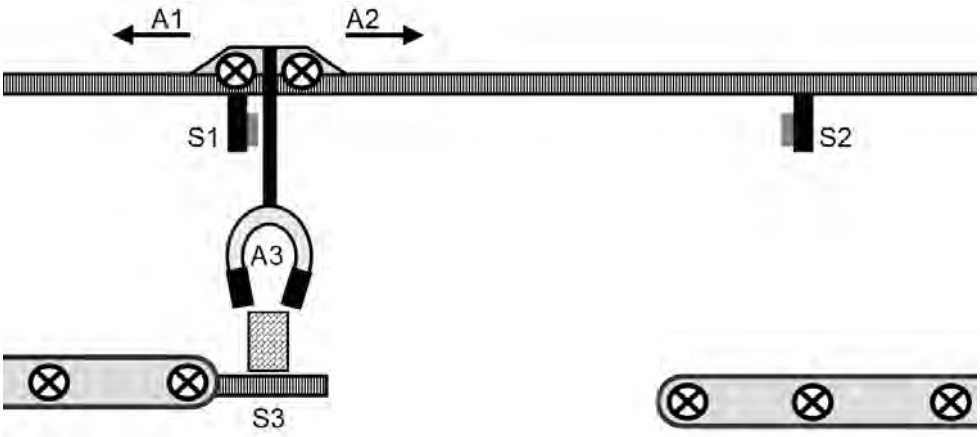


Figure 4.12 - Assembly line scenario

The crane can check sensors S1 to S3, and switch actuators A1 to A3 (see Figure 4.12), whose roles are listed in Table 4.1. Each sensor indicates that its specific condition test is verified using value 1, and otherwise using value 0. For instance, the value of sensor S3 becomes 1 as soon as an item arrives from the left hand side conveyer, and it turns to 0 when the magnetic crane pulls it off. Analogously, each actuator is turned on with value 1, and off with value 0. Therefore, the control signal of actuator A3 must be set to 1 and 0 in order to turn the magnetic pull on and off, respectively.

Table 4.1 - Sensor and actuator roles in the assembly line scenario

Sensor roles		Actuator roles	
S1	Crane is at the left end	**A1**	Move crane to the left
S2	Crane is at the right end	**A2**	Move crane to the right
S3	Item is on platform	**A3**	Grab item

The FSM depicted in Figure 4.13 defines a feasible crane behavior based on the given sensor and actuator signals.

The presented example demonstrates potentialities of the developed hardware and software tools for visual simulation based on virtual graphical models of physical objects. Similar techniques can be used (and have been used) for representing systems implemented just in hardware (such as a SAT solver).

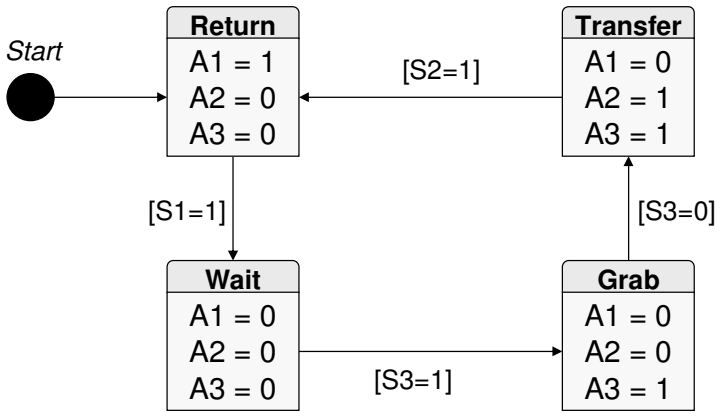


Figure 4.13 - Feasible FSM for controlling the assembly line component

4.2. Advantages and applicability of the designed prototyping tools

It is known that there are many prototyping boards available on the market. Why has one more board been designed? First of all, the board was planned to be used by undergraduate and postgraduate students of electronics, telecommunications and computer engineering curricula. These students have to acquire profound knowledge and abundant experience in the scope of electronic circuit design and software engineering. Therefore, we would like to use open-source hardware/software tools which are completely understandable without any hidden feature. This requirement is also very important for the thesis area because we would like to avoid any misunderstanding in both software and hardware used for experimental purposes. Besides, such tools have to satisfy all necessary functional requirements. The most appropriate solution was to develop the board in the department by postgraduate students, which can easily spread the required knowledge and experience to other students. It was done in [Almeida08] and the following benefits have been obtained:

- The board has become an ideal platform for the development of both electronic devices and software which interacts with hardware. Indeed it does not have any hidden or unknown element or source code. Such open hardware and software is very uncommon for commercial prototyping systems, *i.e.* source code for FPGA configuration, communication with host computers, etc. are usually not provided.

- The board is a very suitable element for remotely controlled embedded systems, mainly because it supports wireless interface. The latter is not widely available for FPGA-based prototyping systems.

- The board is very flexible and easily extendable; it can be customized for many practical applications in such a way that the developed board-based system will include only the required components. This feature is also not so common for the majority of commercially available boards, which contain many auxiliary devices that are not required for particular user applications.

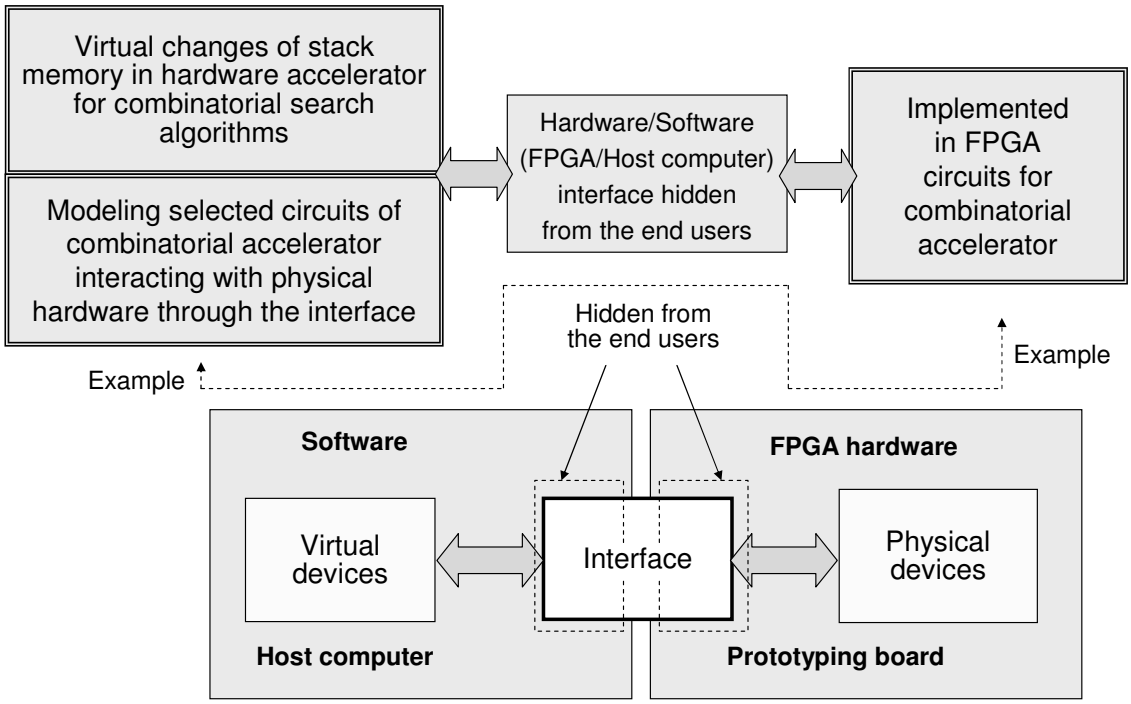


Figure 4.14 – Overview of the virtual visual environment

Figure 4.14 summarizes the main characteristics of a virtual visual environment:

1. Virtual devices are implemented in a host computer. They are virtual devices because they are implemented in software and provide functionality that is very similar to physical devices. They are visual because we are able to observe the functionality (such as different changes in stack memory during forward and backward propagation steps in combinatorial search algorithms) in visual mode on a monitor screen (or possibly in some other connected peripheral devices). They are easily controllable because we can carry out numerous functional and timing scenarios, for example, test only selected fragments of the

implemented algorithms, execute algorithmic steps faster or slower, etc.

2. Physical devices are implemented in FPGA and they interact with virtual devices in such a way that allows to make up the designed system, *i.e.*: *physical devices* + *virtual devices* = *the designed system*. Such system is flexible and extendable, because functionality of both software and reconfigurable hardware can be altered.

3. Software/reconfigurable hardware interface providing interaction between the virtual and physical devices are hidden from the end users.

The proposed technique permits:

- To verify the accelerator entirely in software;
- To implement the accelerator partially in software and partially in hardware;
- To carry out hardware/software co-simulation with adjustable boundary between hardware and software (*i.e.* to analyze the accelerator with either more software and less hardware or vice versa).

Let us consider one more example. Suppose we have to design the combinatorial accelerator shown in Figure 4.15. The main idea is to verify if this accelerator can be reused for solving different combinatorial problems formulated over Boolean and ternary matrices (such as that discussed in [Skliarova06b]). The part shown with grey background is projected to be reusable and the control unit is intended to be reprogrammable in such a way that allows implementing different combinatorial algorithms (such types of combinatorial accelerators are described in detail in [Skliarova06b]). Let us model the reusable part (*i.e.* the part shown with grey background) in software, and implement the control unit in FPGA. Suppose that a request for reprogramming the control unit has to be done from a host computer, which knows a particular problem that must be solved. Examples of such problems might be the SAT, binary matrix covering, etc. In order to model the considered reusable circuit in the host computer, it is necessary to develop a program using a library of classes which model the relevant hardware parts, such as matrices, stacks, registers (see Figure 4.15), etc. These classes will be described in detail in chapter 5. Suppose the control unit is modeled by a reprogrammable FSM whose functionality can be changed through reloading the FSM's RAM blocks. Methods for synthesis of reprogrammable FSMs are proposed in [Sklyarov02a]. Interaction between software

and hardware parts can be provided with the aid of the developed interface components (the agent module and the PBM), which establish links between the designated inputs and outputs of the circuit implemented in hardware (in FPGA) and hardware parts modeled in software (see Figure 4.15). Finally, we can test the circuit for a particular algorithm (for example for the Boolean satisfiability) with the aid of class functions which visually demonstrate the functionality of the simulated hardware parts in a monitor screen.

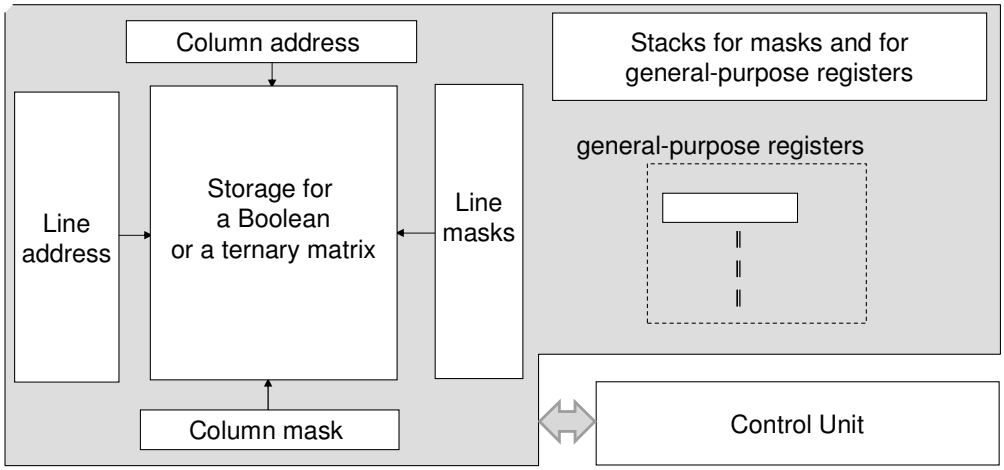


Figure 4.15 – Structure of a combinatorial accelerator

Let us now consider how to change the circuit functionality in order to examine different algorithms. For such purposes, it is necessary to implement a handler which is able to alter the algorithm of the control unit (we assume that the execution unit, shown with grey background, is exactly the same). Thus, we have to apply the same technique that is demonstrated in Figure 4.11. In the beginning, the handler can be modeled in software. After the handler has been tested, it can be implemented in FPGA. Incrementally, other blocks of the explored system (see Figure 4.15) might be converted from software to hardware. This technique gives vast opportunities for hardware/software co-simulation and consequently for the design space exploration. Obviously, this task is very interesting and helpful for students. The system presented above has not been completely finished yet, although the majority of basic components (such as the DETIUA-S3 board, software libraries for virtual peripheral devices and for numerous hardware objects, remote interaction with the board through the Internet) have been implemented and tested. The results of testing demonstrate good capabilities of the developed components for remote monitoring and design of reconfigurable systems. Many developed elements have been used for prototyping and experiments (see chapter 6).

4.3. Conclusion

The proposed tools for hardware/software co-simulation of reconfigurable systems, including remote monitoring and design, are very promising in a vast scope of practical applications, such as virtual design space exploration, rapid dissemination of different models and methods in the scope of hardware design, comparison of alternative and competitive circuit implementations using the Internet facilities, education, engineering training, etc. These tools possess the following distinctive features:

1. Prototyping board managing through either wired (USB) or wireless (Bluetooth) interface;
2. Remote design and monitoring of reconfigurable systems;
3. Software/reconfigurable hardware co-simulation through the developed interfaces supported by the relevant hardware projects and software tools.

All the developed software tools were modeled in C#, while the developed hardware tools have been implemented in hardware on the basis of Handel-C specifications or VHDL descriptions. Dependently on the specific component, Microsoft Visual Studio with the .NET framework, the Celoxica DK, the Xilinx ISE, and the Mentor Graphics ModelSim have been used.

A new prototyping system which includes an FPGA-based board (called DETIUA-S3) has been developed (with Manuel Almeida [Almeida08]). The board architecture is based on five main components: an FPGA, a flash memory, a CPLD, an interface module (either USB or Bluetooth), and expansion connectors. A new software application (called PBM) was developed to take full advantage of the board, allowing for its configuration and for data-exchange with a general-purpose computer.

A framework called Remote Lab has been under development to support remote interaction with DETIUA-S3. When finished, a server computer connected to a DETIUA-S3 board will be able to provide most of the PBM functionality to users through the internet.

Two hardware/software co-simulation frameworks have been developed based on input and output signal exchange between circuits implemented in FPGA and virtual

environment software applications and they can be combined with the Remote Lab framework. They are the following:

1. A virtual visual sub-system allows virtual peripherals to be used instead of real ones. The visual and the internal behaviors of the most typical peripherals, such as LEDs, pushbuttons, dipswitches, seven-segment displays, and LCDs, have been modeled in software to emulate real peripherals, and more can be added.

2. A reprogrammable FSM-based partition architecture allows designers to implement control units in FPGA using a reprogrammable FSM, to model execution units and application-specific peripheral devices in software, and to monitor projects with step-by-step capabilities.

Figure 4.16 discloses the key characteristics of the developed hardware/software co-simulation models. Light-gray indicates hardware implementation, while dark-gray indicates software emulation.

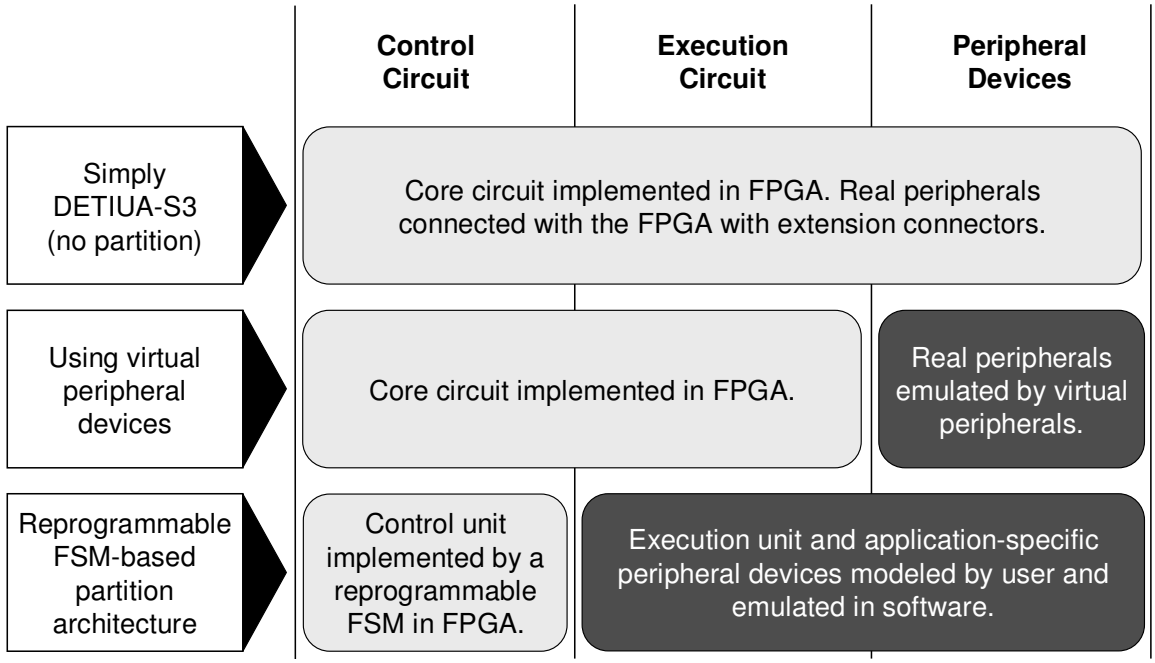


Figure 4.16 - Using different hardware/software co-simulation frameworks

The developed tools have been used for the majority of experiments provided for analysis and comparison of recursive and iterative algorithms studied in this thesis.

5. Algorithm Modeling and Implementation

The algorithms considered in chapter 3 have one common characteristic: they require generation and evaluation of a huge number of different variants before a solution is found. It was shown that feasible solutions can be generated with the aid of a search tree, whose nodes represent different situations that are reached during the search for results, and whose edges specify steps of the algorithm that have to be performed. A distinctive feature of this approach is that, at each node of the search tree, the same problem is being solved. The only thing that changes from node to node is the input data. This means that the whole problem can simply be solved by repeating an often large number of times the execution of a single limited set of operations over a periodically modified set of data.

It has already been shown by various researches that reconfigurable hardware can provide some benefits (over software) when solving these problems. This is mainly due to the possibility to parallelize execution of some repeated operations, as well as to tailor memory interface to the required data structures. The main objective of this work is however not to find the best reconfigurable hardware implementation (in terms of the required resources or performance). Instead, the main idea is to assess the relative cost of using iterative and recursive algorithms in hardware.

This chapter therefore provides details of reconfigurable hardware implementation of iterative and recursive algorithms for the selected problems. To simplify the design process, every algorithm was first modeled in software (the first part of chapter 5 is devoted to this topic). Then, descriptions of some of the algorithms were created in a system-level specification language (Handel-C) and a hardware description language (VHDL). The second part of chapter 5 gives all the details. The respective specifications were finally synthesized and implemented in commercially available

FPGAs and carefully analyzed (the respective results being reported in the next chapter).

5.1. Modeling in software

In order to estimate relative effectiveness of recursive and iterative specifications of different algorithms, as well as to check their correctness, all the algorithms were first modeled in software and only after that implemented in hardware. The subsequent sections will provide all necessary details.

5.1.1. Data structures

It was indicated in chapter 3 that discrete matrices were selected as the primary mathematical model because of two reasons: matrices can easily be stored and processed in both software and hardware; and the majority of the considered combinatorial search problems can efficiently be formulated over matrices. Taking into account this decision, relevant data structures need to be created so as to provide support for storing and manipulation of matrices, by the respective algorithms.

We suppose that a discrete matrix is composed of a set of discrete vectors (either rows or columns). Therefore data structures are needed for representing both vectors and matrices. An object-oriented design approach was followed and, as a result, several classes were created. These will be represented with the aid of class diagrams which were generated in Microsoft Visual Studio .NET. Note that, for this reason, the Unified Modeling Language (UML) regulations are not strictly followed. A relevant difference is the use of two-headed arrows to identify arrays of objects.

5.1.1.1. Common classes

The two most basic classes which are required by the majority of the selected algorithms allow the storing of vectors and masks. Class *Vector* can keep general-purpose vectors, as well as matrix rows and columns, and they can be either binary or ternary. Class *Mask* keeps a series of binary values which are used to mark their indexes, for instance, as deleted/not deleted or as selected/not selected.

Let us consider *Vector* and *Mask* classes in detail. The class diagram in Figure 5.1 reveals the most relevant functionality which is implemented by these two classes.

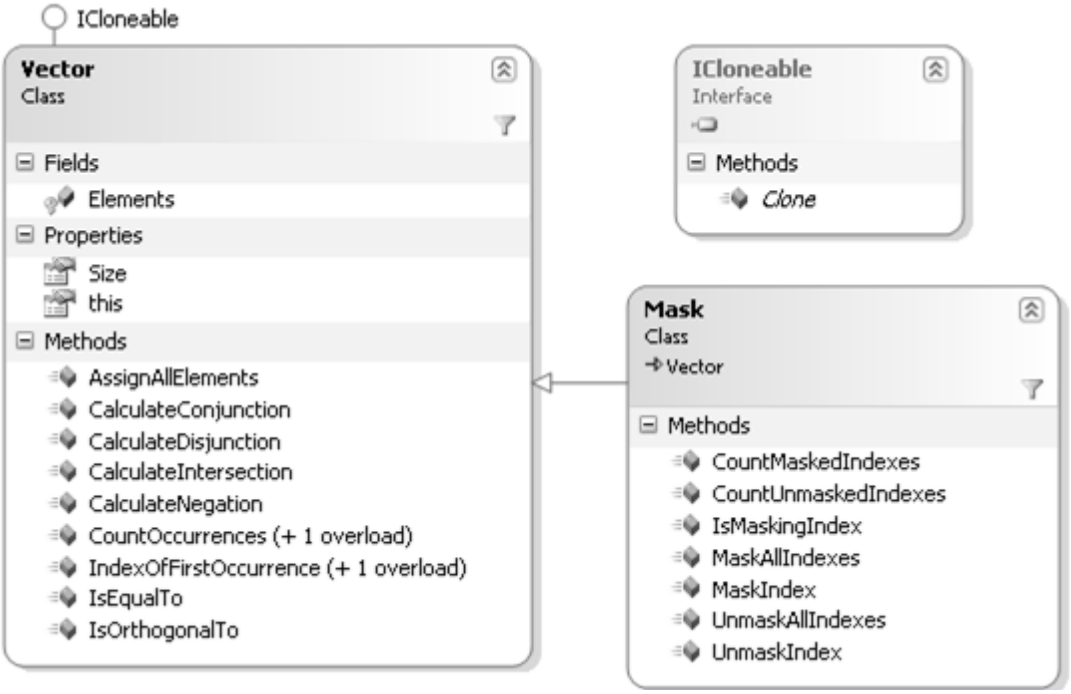


Figure 5.1 - Class members of *Vector* and *Mask*

Vector class contains a single data field, which is an integer array for storing the vector elements. Although integers have a wide range of possible values, the selected algorithms require only values 0, 1, and, when using ternary values, *don't-care*. The latter is coded with integer value 2. Class *Vector* provides methods for calculating the conjunction, the disjunction, and the intersection with another *Vector*, etc.

Class *Mask* is derived from *Vector*, as the data structure it requires is that of a binary vector. The purpose of class *Mask* is to allow for an intuitive set of methods for dealing with index masking. Hence, it provides methods for index masking and unmasking, checking whether an index is masked, and also counting masked and unmasked indexes. In practice, masked indexes correspond to values 1, whereas unmasked indexes correspond to values 0. A masked index can denote a deleted row/column, when applying deletion masks to a matrix (see Figure 5.2-a). This technique keeps deletion operations simple, in opposition to actual memory deallocation. Eventual row and column recovering is equally simple. The only emerging requirement is that some operations over *Vectors* must take these masks (provided as parameters) into account. Furthermore, the *Mask* class can also be used as selection masks for choosing subsets of rows or columns (see Figure 5.2-b), *e.g.* in set covering, SAT, and graph coloring algorithms.

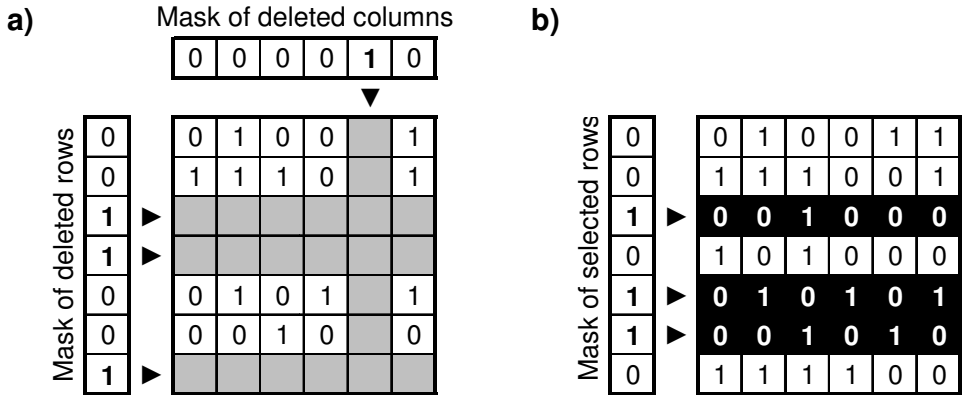


Figure 5.2 - Using deletion (a) and selection (b) masks

Class *Matrix* provides general-purpose properties and methods which are inherited by classes implementing matrix-based algorithms. The class diagram in Figure 5.3 depicts the relevant properties, methods and derived classes of *Matrix*.

Figure 5.3 - Properties, methods, and derived classes of *Matrix*

As previously mentioned, once the original problem matrix has been stored, the considered matrix-based algorithms do not require methods for writing rows, columns,

or elements. Thus, elements are written only within the class constructor, and only reading and deleting methods are provided. Throughout the execution of an algorithm, matrix rows eventually have to be deleted. At any step of the algorithm, two matrix properties provide the original and the current number of undeleted rows in the matrix (respectively *OriginalNumberOfRows* and *CurrentNumberOfRows*); and the same scheme applies to columns. Methods *RowIsDeleted* and *ColumnIsDeleted* can be used to test whether a specific row or column has been deleted.

The class diagram in Figure 5.4 reveals private data fields on which *Matrix* operates in order to implement its functionality.

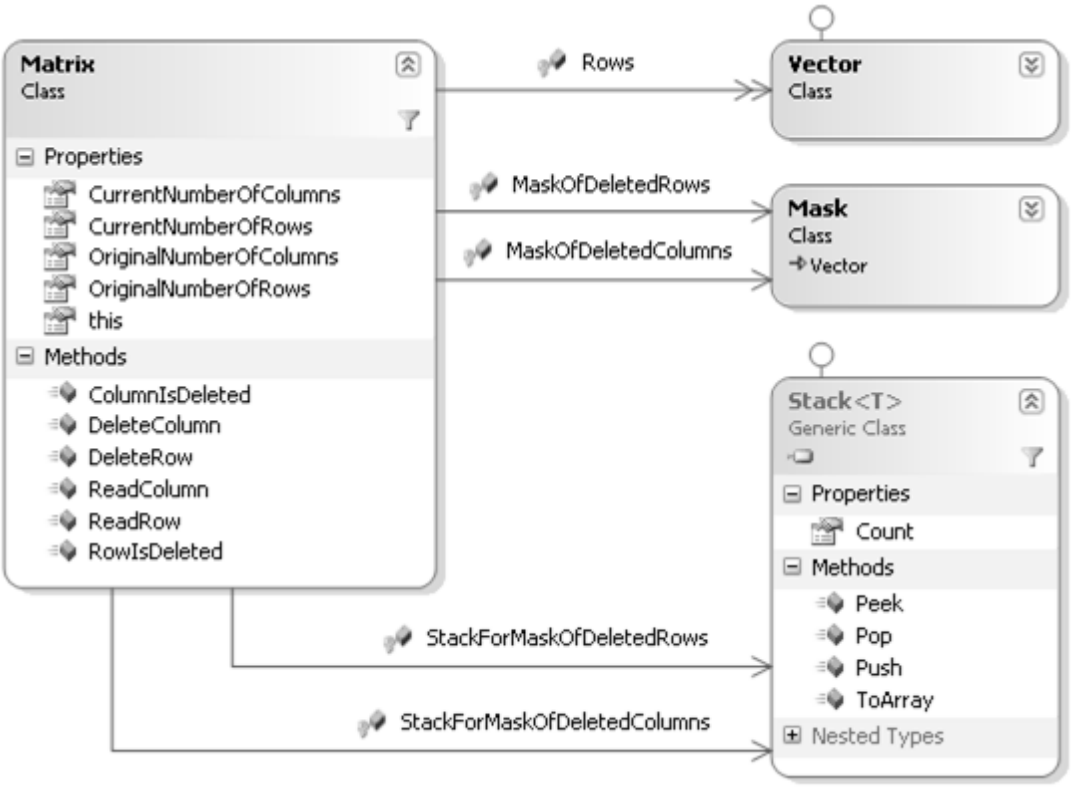


Figure 5.4 - Class members of *Matrix*

Matrix elements are organized by rows, as an array of *Vectors* (the *Rows* field). Two fields of class *Mask* are used for masking the subset of matrix rows and the subset of matrix columns which have already been deleted. Matrix properties *CurrentNumberOfRows* and *CurrentNumberOfColumns* provide the current number of unmasked indexes of those *Masks*. In order to support backtracking, class *Matrix* also features *Mask*-storing *Stacks* which permit to store the current state of the matrix and to recover matrix states that have previously been stored.

The classes described in this section were augmented with methods that permit all the selected algorithms to be modeled in software. As a result, a new set of classes was created (including those which are shown at the bottom of Figure 5.3) and their detailed descriptions are provided in the following sections.

5.1.1.2. Classes for set covering algorithms

The *SetCoveringMatrix* class is derived from the *Matrix* class and features the additional fields and methods which are required for running set covering algorithms. The relevant members of this subclass are depicted in the class diagram of Figure 5.5.

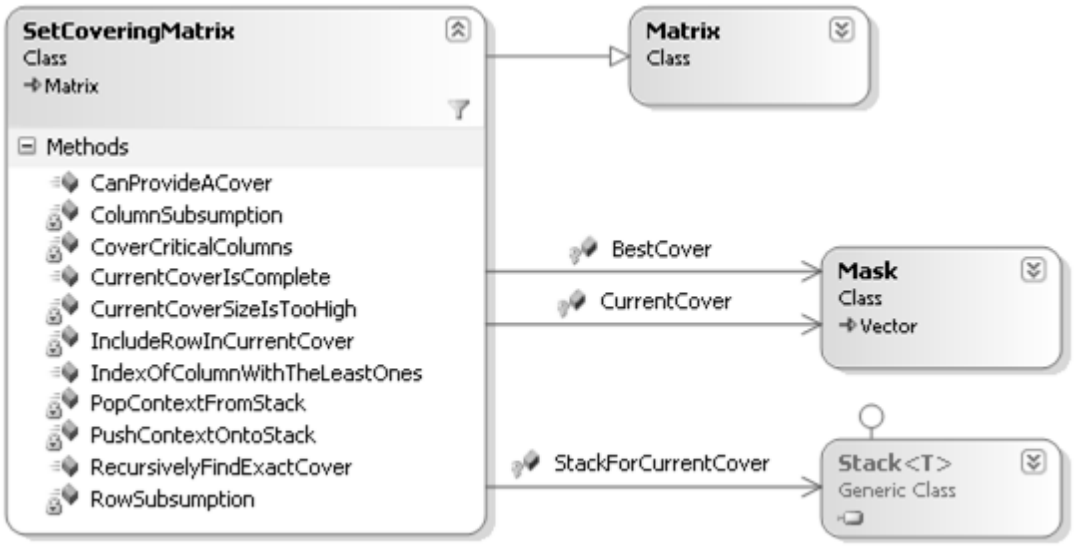


Figure 5.5 - Class members of *SetCoveringMatrix*

Two *Masks* are used for keeping the current and minimum row covers, masking the indexes of the rows included therein. One *Stack* is used for storing and restoring the current cover throughout the search tree of the set covering algorithm. Auxiliary methods model the functionality of the solvability and resolution tests, and the reduction and selection rules that were presented in section 3.2.2.

5.1.1.3. Classes for SAT solving algorithms

Also deriving from the *Matrix* class, the class *SATSolvingMatrix* contains the additional functionality which is required for running Boolean satisfiability algorithms. The class diagram in Figure 5.6 depicts the relevant members of this new class.

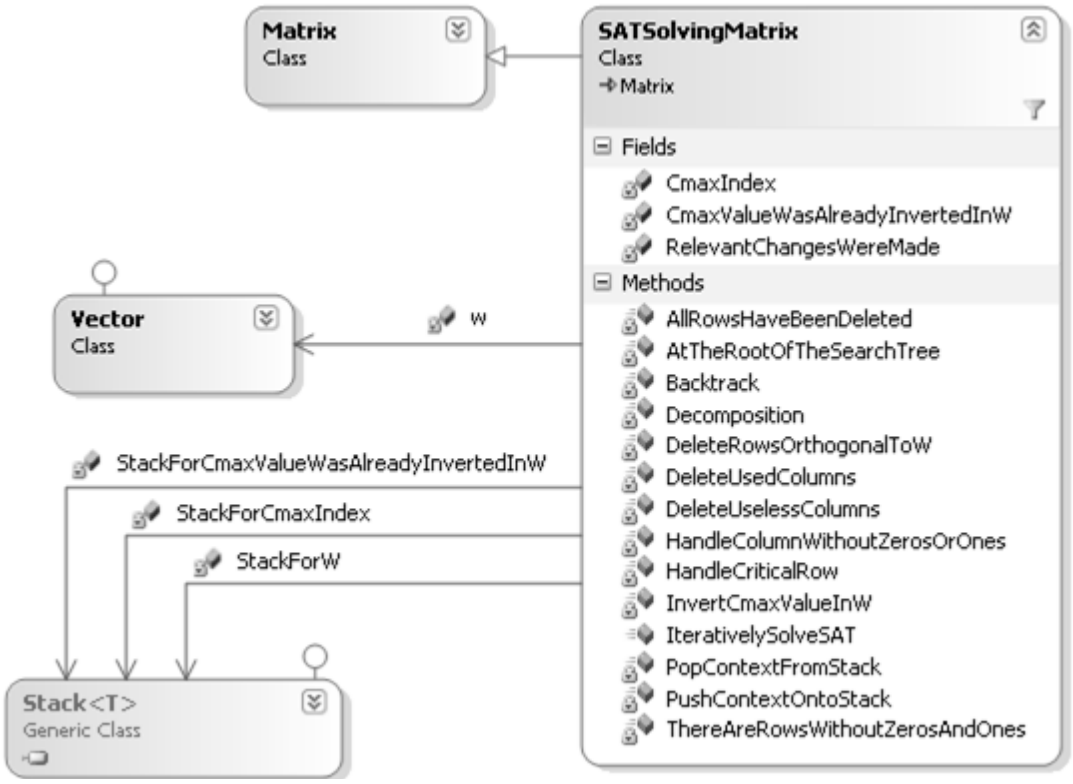


Figure 5.6 - Class members of *SATSolvingMatrix*

Vector w is used for incremental construction of the solution vector. Three *Stacks* are required for storing and restoring the context while traversing the search tree. Three additional attributes provide support for correct algorithmic flow. Last, several private methods carry out the required reduction and selection rules, and the resolution and solvability tests which were presented in section 3.2.3.

5.1.1.4. Classes for graph coloring algorithms

One more subclass of *Matrix* – *GraphColoringMatrix* – contains the necessary class members for running graph coloring algorithms. The relevant methods and properties of this new class are depicted in the class diagram of Figure 5.7.

A *Mask*-storing *Stack* denominated *CurrentColoring* (see lower right-hand side corner of Figure 5.7) is used to incrementally compose the coloring currently under construction. When a new color is required, a new *Mask* is pushed onto that *Stack*. When a row R (corresponding to a graph vertex) is to be assigned the current color, the *Mask* at the top of that *Stack* is used for masking the index of row R.

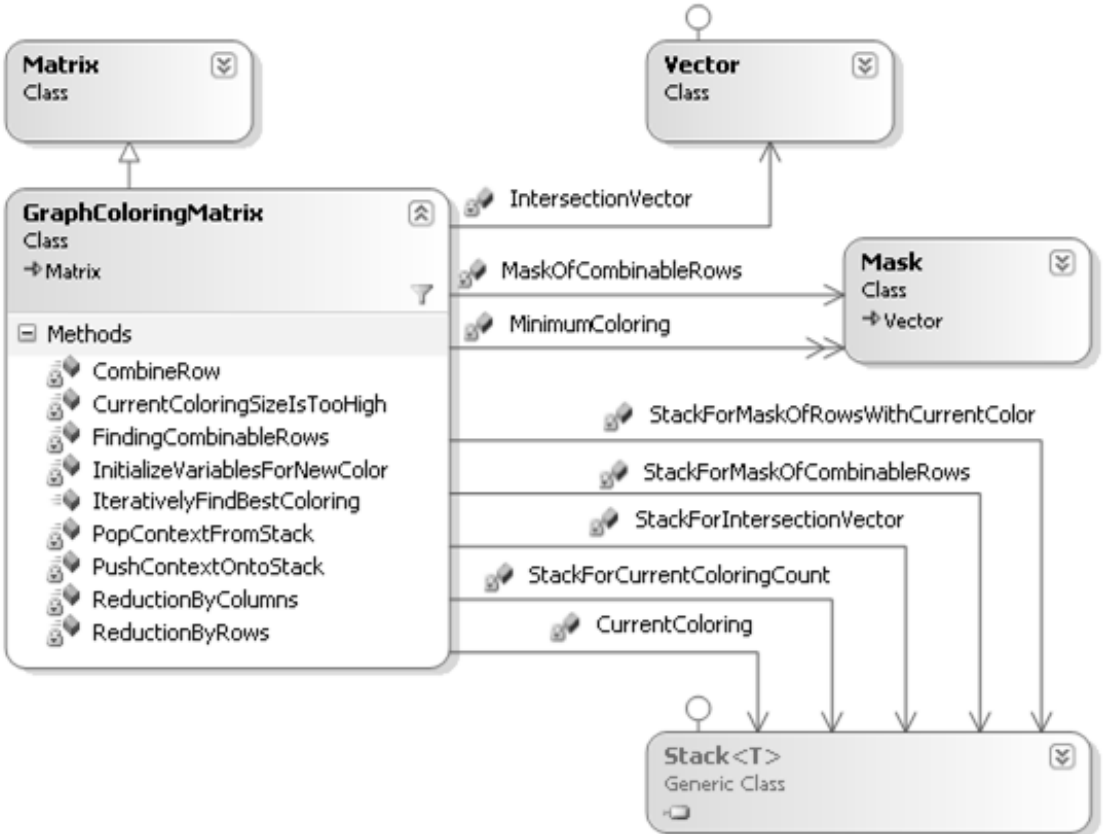


Figure 5.7 - Class members of *GraphColoringMatrix*

On the other hand, an array of *Masks* denominated *MinimumColoring* (see right-hand side of Figure 5.7) is used to keep the minimum coloring, *i.e.* the complete coloring witch has the minimum number of colors found so far. If the current coloring becomes complete and contains fewer colors (fewer *Masks*) than the minimum coloring, the contents of the *CurrentColoring Stack* are used to replace those of the *MinimumColoring Mask* array.

A *Mask* denominated *MaskOfCombinableRows* is used to identify the indexes of rows which can be assigned the current color. When a row S is assigned the current color, the *IntersectionVector* is updated so as to keep the result of the intersection of all rows assigned the current color (including S).

Four other *Stacks* are used for context storing and restoring, by means of *GraphColoringMatrix* pushing and popping methods, respectively (see left-hand side of Figure 5.7). These methods support the backtracking mechanism.

Other auxiliary methods provide the functionality of the solvability and resolution tests, and the reduction and selection rules presented in section 3.2.4.

5.1.1.5. Classes for solving the knapsack problem

Three classes have been created for solving the knapsack problem. The relevant class diagram is depicted in Figure 5.8.

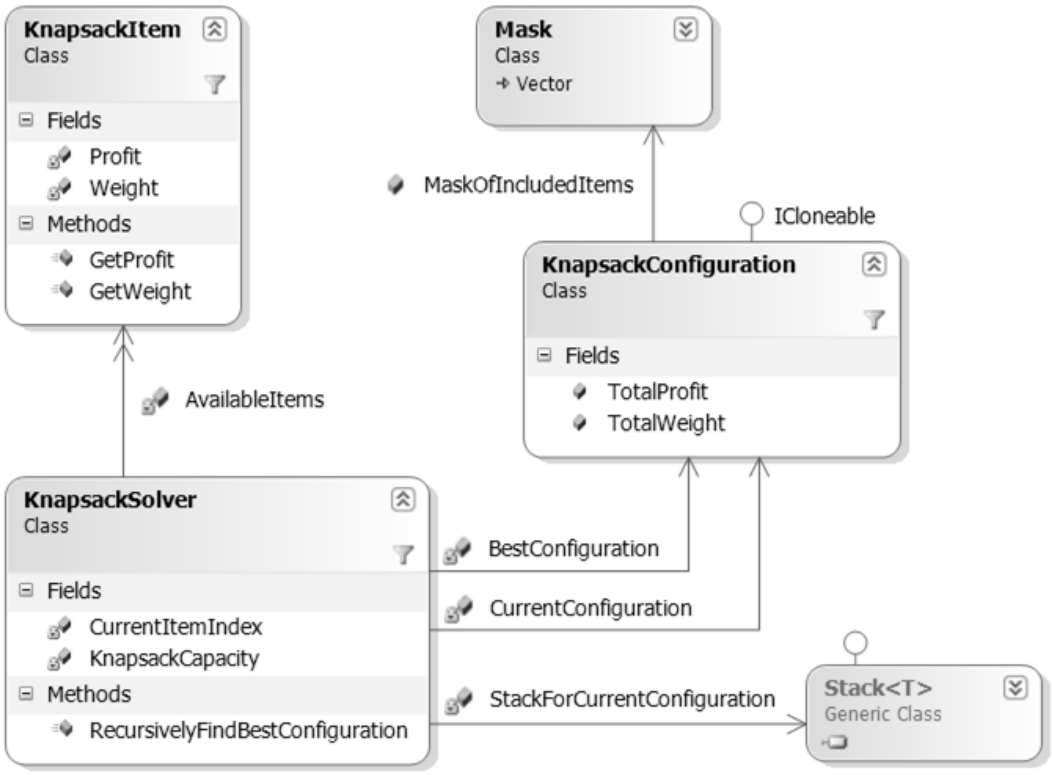


Figure 5.8 - Class diagram for knapsack-solving algorithms

The *KnapsackItem* class is used for representing the available items, each one holding its own profit and weight, and methods for reading those (see top left-hand corner of Figure 5.8).

An instance of the *KnapsackConfiguration* class represents a selection of such items. A *Mask* is used for determining which items are included in the knapsack, and two other fields provide the total profit and weight (see top right-hand corner of Figure 5.8).

The constructor method for the *KnapsackSolver* class takes an array of *KnapsackItems* and the knapsack weight capacity as parameters. These parameters characterize the problem instance. When searching for the most profitable knapsack configuration, the *CurrentConfiguration* field is used for building up, one by one, all the feasible solutions to the problem. The *BestConfiguration* field keeps the most profitable knapsack configuration found so far. In order to iterate through the array of available items, an

integer attribute is incremented at each stage of the algorithm, keeping track of the current item's index.

A *Stack* is used for storing and restoring the knapsack configuration under construction throughout the algorithm search tree (see bottom right-hand corner of Figure 5.8).

5.1.1.6. Classes for tree-based sorting algorithms

Tree-based sorting algorithms require modeling of binary tree nodes, whose instances assume different values to be sorted. Figure 5.9 presents the class diagram which was built for this purpose.

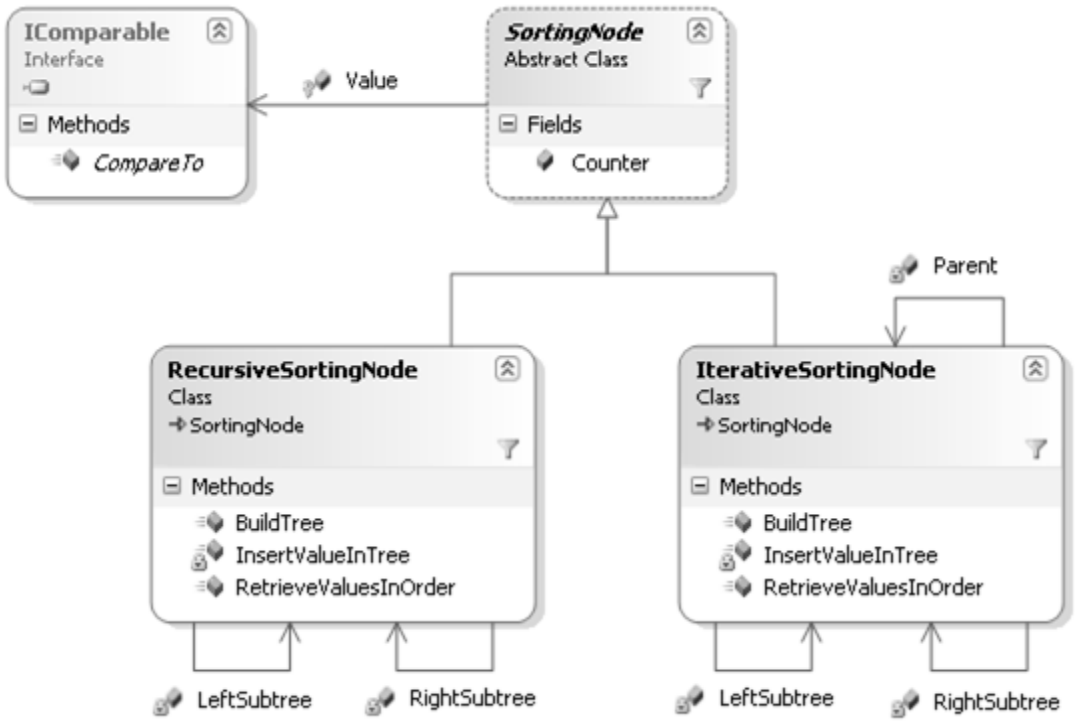


Figure 5.9 - Class diagram for iterative and the recursive tree-based sorting

The iterative and the recursive sorting algorithms have different requirements regarding the node class members. For this reason, a *RecursiveSortingNode* and an *IterativeSortingNode* classes have been modeled, inheriting the class members which they share from the same class: the *SortingNode* abstract class.

Any object which implements the *IComparable* interface can be the assignment of *SortingNode*'s *Value* field. The utilization of this interface allows the developed

algorithms to sort objects of different classes, as long as they implement the required comparison method.

Integer field *Counter* is used for keeping track of the number of occurrences of the assigned value. In fact, this field leads to sorting algorithms somewhat more enhanced than the algorithm which was presented in section 3.3.1. The differences take place in the insertion and the retrieval of repeated values. In the tree construction stage, when inserting a value which has already been inserted in a node N, the *Counter* of N can simply be incremented, instead of constructing a new node. Moreover, in the retrieval stage, the value of each node must then be retrieved as many times as stated by the *Counter*. Figure 5.10 and Figure 5.11 respectively illustrate the tree construction and value retrieval stages, when integrating this strategy into the tree-based sorting algorithm. The value of the nodes' *Counter* is revealed within parenthesis, when different from 1 (see Figure 5.10-d). The sequence of values which is given in the example is 4-2-7-5-7-9, the sorted result being 2-4-5-7-7-9.

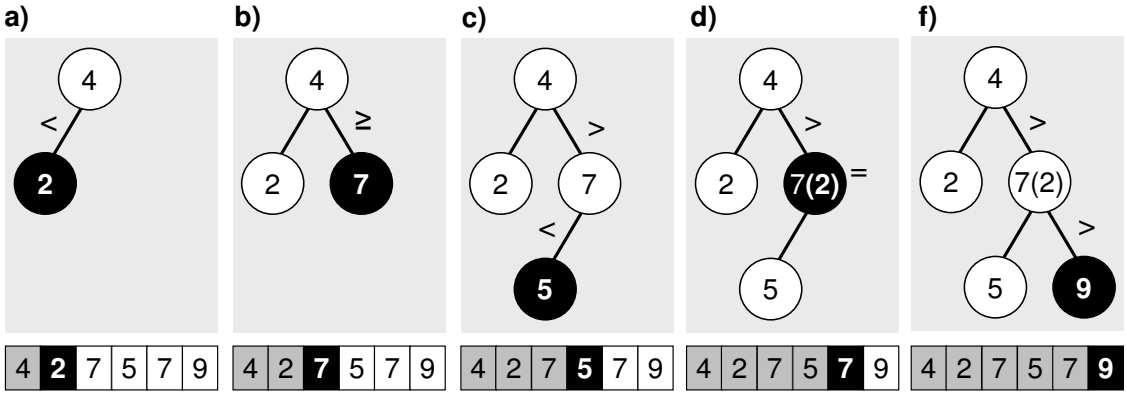


Figure 5.10 - Constructing a sorting tree with occurrence accumulation

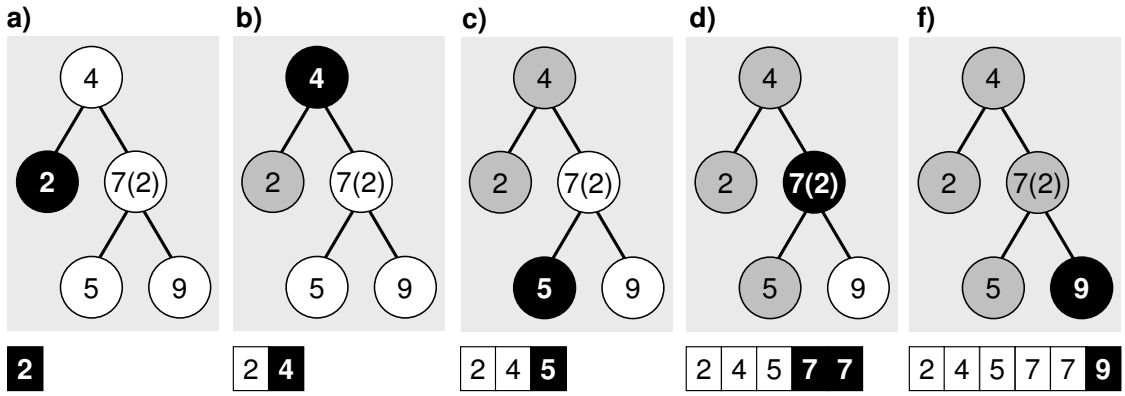


Figure 5.11 - Retrieving the values of a sorting tree with occurrence accumulation

Moving on to node fields, although there is a *LeftSubtree* and a *RightSubtree* fields in both the derived classes of *SortingNode*, these fields do not belong to that base class because their type (class) is distinct. In fact, sub-trees of *RecursiveSortingNodes* should be composed solely of nodes of the same type, and the same applies for an *IterativeSortingNode*.

The most relevant differences between the *RecursiveSortingNode* and the *IterativeSortingNode* classes are:

1. the recursive vs. iterative nature of their methods for:
   a. inserting a value in the tree;
   b. in-order retrieving the tree values;
2. the *IterativeSortingNode*'s *Parent* field which is required for the iterative algorithm to backtrack in the tree towards its root.

The Microsoft Visual Studio built-in *Queue* class features easy-to-use queuing and dequeuing of generic objects. For this reason, the *BuildTree* public method of both *SortingNode*'s sub-classes takes as a parameter a *Queue* of the *IComparable* objects which are to be sorted. One-by-one, each of these objects is dequeued and inserted in the sorting tree by means of the *InsertValueInTree* private method.

5.1.1.7. Classes for calculating the greatest common divisor

A simple *GCDCalculator* class (see Figure 5.12) has been created to provide recursive and iterative public methods for calculating the greatest common divisor of two integers.

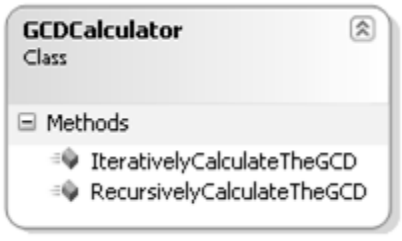


Figure 5.12 - The *GCDCalculator* class

Both GCD-calculating methods take two integer parameters for providing the values between which the GCD is to be calculated. No auxiliary methods or composite types are required.

5.1.2. Algorithmic flows

This section presents the UML Activity Diagrams which were prepared when modeling the selected algorithms in software. Algorithmic flows described here were later reused as a basis for algorithm implementation in hardware as well.

5.1.2.1. The set covering algorithm

Section 3.2.2 presents the reduction and selection rules which are employed in the exact set covering algorithm. Figure 5.13 and Figure 5.14 respectively depict the recursive and iterative top-level activity diagrams built on the basis of those rules.

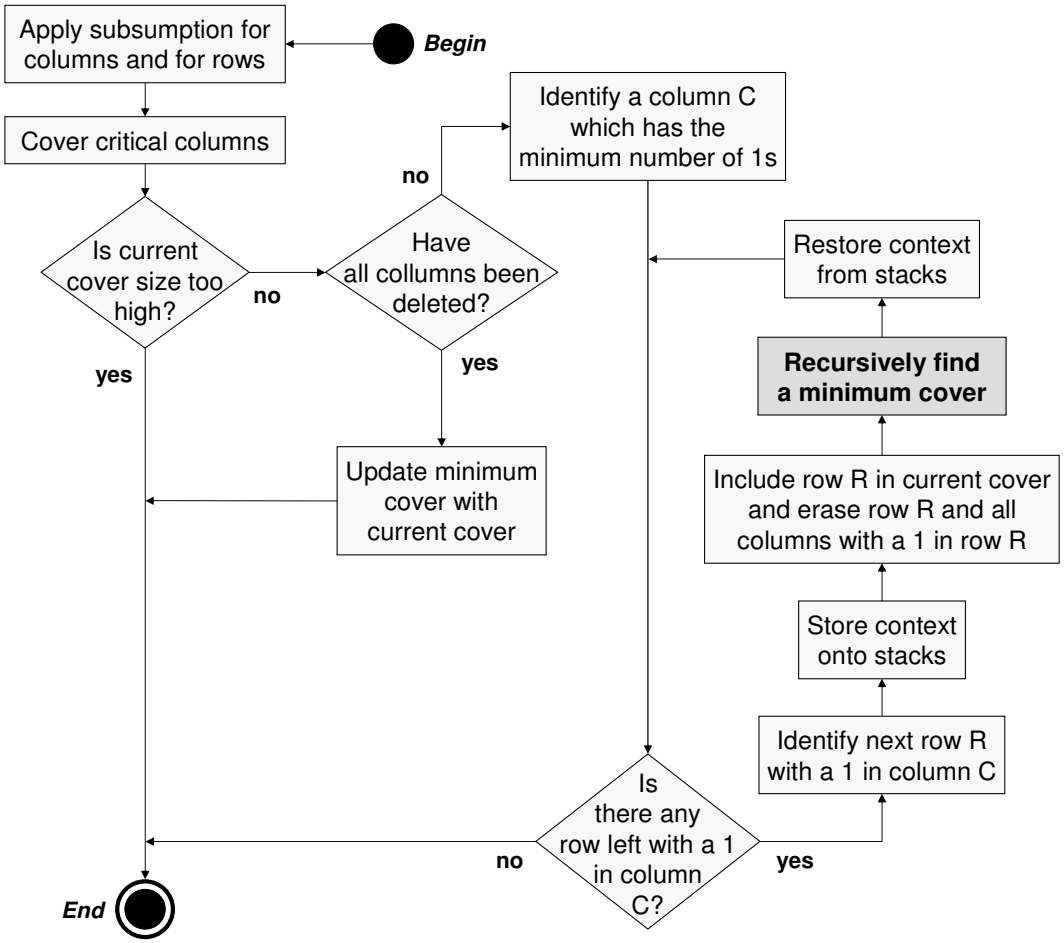


Figure 5.13 - Recursive method for finding an exact cover

When starting a new branch of the search tree, the recursive algorithm applies reduction rules R1 and R2 (see section 3.2.2), *i.e.* carries out subsumption for columns and subsumption for rows until it is no longer possible. At the second step, selection rule S1 is applied, *i.e.* for any column C containing a single value 1, the row which includes this value is included in the current cover. The remaining part of the diagram is dedicated to application of the selection rule S2 (see section 3.2.2).

The darkened node in Figure 5.13 corresponds to a recursive invocation call. The search for a minimum cover, using the recursive algorithm, is therefore carried out by re-executing exactly the same method. The search tree backtracking mechanism is supported by the use of stacks, onto which context variables are stored before starting a new search tree branch, and from which they are restored when backtracking.

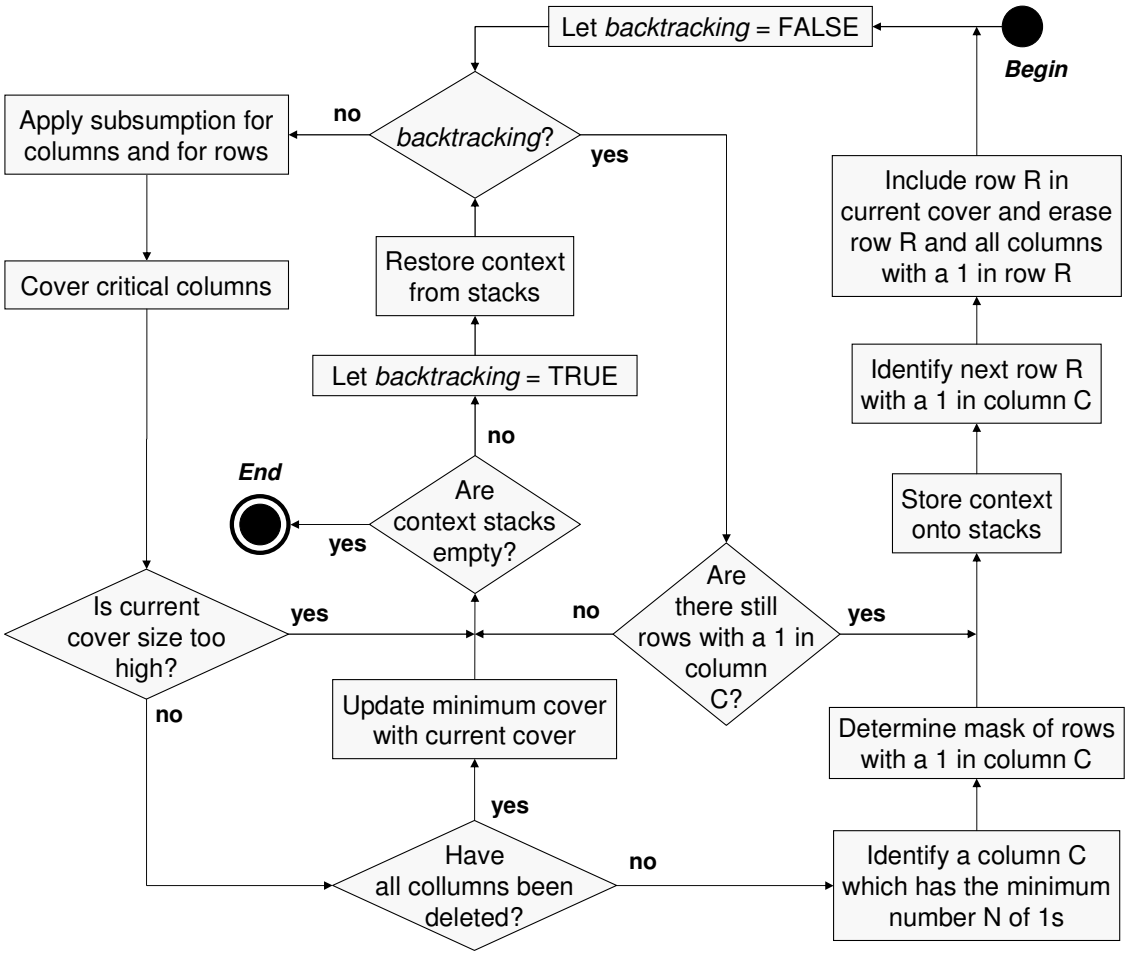


Figure 5.14 - Iterative method for finding an exact cover

As explained in section 3.2.2, the set covering problem description given in the beginning of that section leads to solvable instances only. Moreover, both the

recursive and the iterative algorithms which are presented in this section do not include any operation which would render the processed matrix uncoverable. For these two reasons, no solvability test has been included in the proposed algorithms. Nevertheless, if we require considering unsolvable instances (*i.e.* uncoverable matrices), executing a simple solvability test prior to running any of the proposed algorithms would suffice. Such test would consist of determining whether the given matrix contains a column without values 1 and finding such a column would indicate that the given instance is unsolvable.

Let us now compare the recursive activity diagram depicted in Figure 5.13 with the iterative one in Figure 5.14. Although, both algorithms follow the same reduction and selection rules indicated in section 3.2.2, the superior clearness and simplicity of the recursive algorithm are rather obvious.

5.1.2.2. The SAT solving algorithm

The reduction and selection rules and the solvability and resolution tests which can be used to solve the Boolean satisfiability problem are described in section 3.2.3. The activity diagrams in Figure 5.15 and Figure 5.16 (created on the basis of those descriptions) depict the recursive and iterative algorithms for solving the SAT problem. Note that, although no initialization operation is shown is those diagrams, all elements of vector w must be assigned to *don't-care* before running any of those two algorithms. In both algorithms, if vector w becomes orthogonal to all rows, its negation is calculated and stored in the *solution* vector. If the latter is still unassigned (*i.e.,* if all its elements are still *don't-cares*) when the algorithm finishes, it means that the problem instance is unsatisfiable.

One of the areas identified with a gray background (see Figure 5.15-a) comprehends the sequence of operations which implement the reduction rules (see section 3.2.3). When reduction rule R2, R3, or R4 carries out any changes, the latter might create the conditions for some more reduction rules to be applicable. Thus, when at least one of those rules is effectively applied, the whole reduction sequence is repeated. The iterative algorithm proposed includes exactly the same reduction cycle (see Figure 5.16).

After reduction, if both the resolution and the solvability tests fail — *i.e.*, if w is not yet orthogonal to all rows and it is not known that the current search subtree cannot provide a solution —, then selection rules are applied (see section 3.2.3). At this point,

a branching in the search tree has been reached. An assignment to a chosen element of vector w is made. Then, a recursive call to the same method continues the search through the corresponding subtree. If the *solution* vector is not assigned when returning from this recursive call, a second assignment to the chosen element of vector w is made using the inverted value, and another recursive call is made. Once again, the forward and backward steps in the search tree are supported by storing/restoring context variables onto/from stacks.

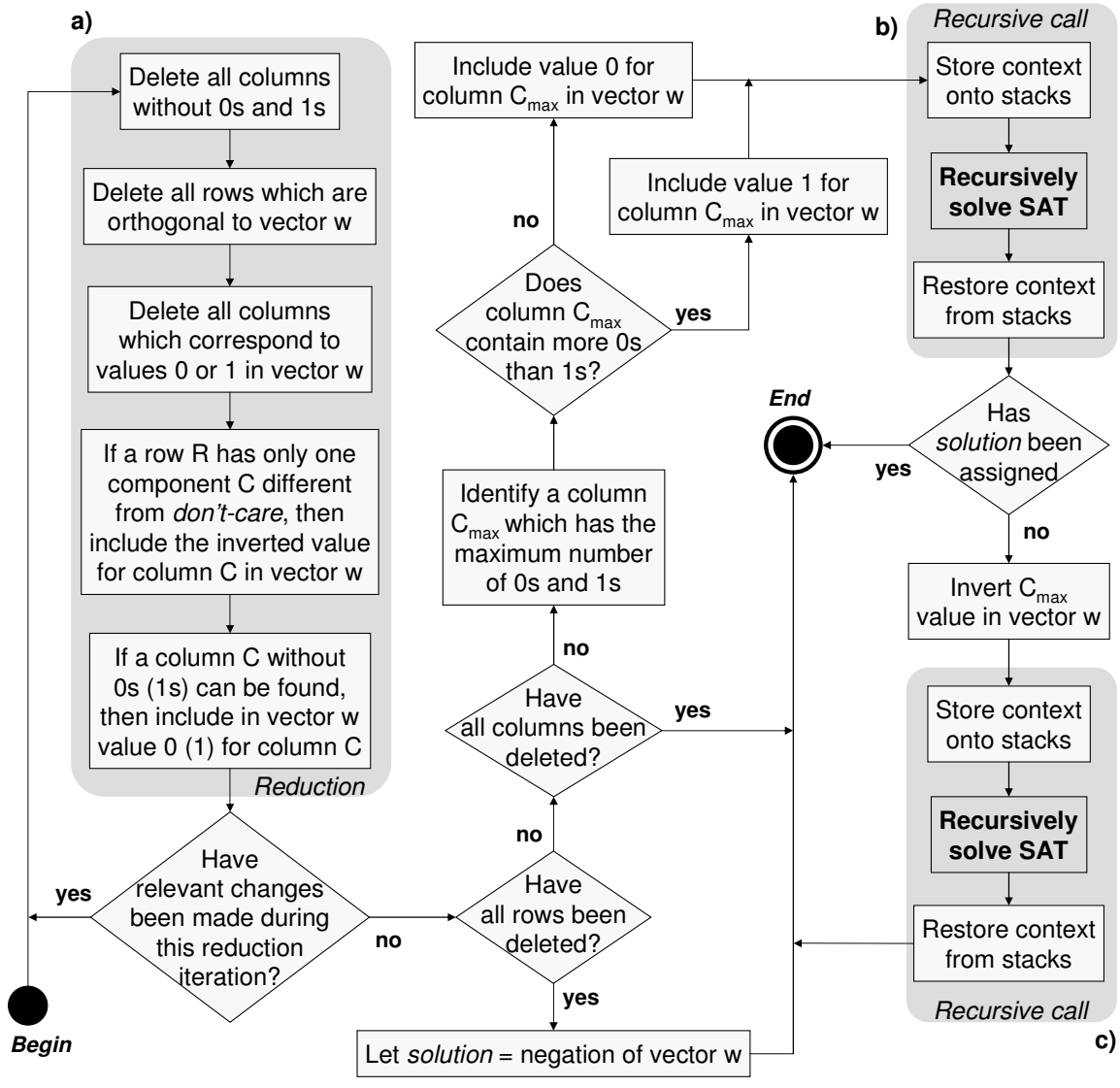


Figure 5.15 – Recursive method for solving the Boolean satisfiability problem

The proposed iterative algorithm for solving the SAT problem implements the same reduction and selection rules as the recursive algorithm. Although these two algorithms were not implemented in hardware, their modeling in software permits to validate the presented activity diagrams which, in turn, allows for a valuable

comparison between recursive and iterative descriptions, regarding an important set of design issues. Indeed, differences in their description's clearness, structural simplicity, and ease of modification can be identified by comparing the activity diagrams in Figure 5.15 and Figure 5.16.

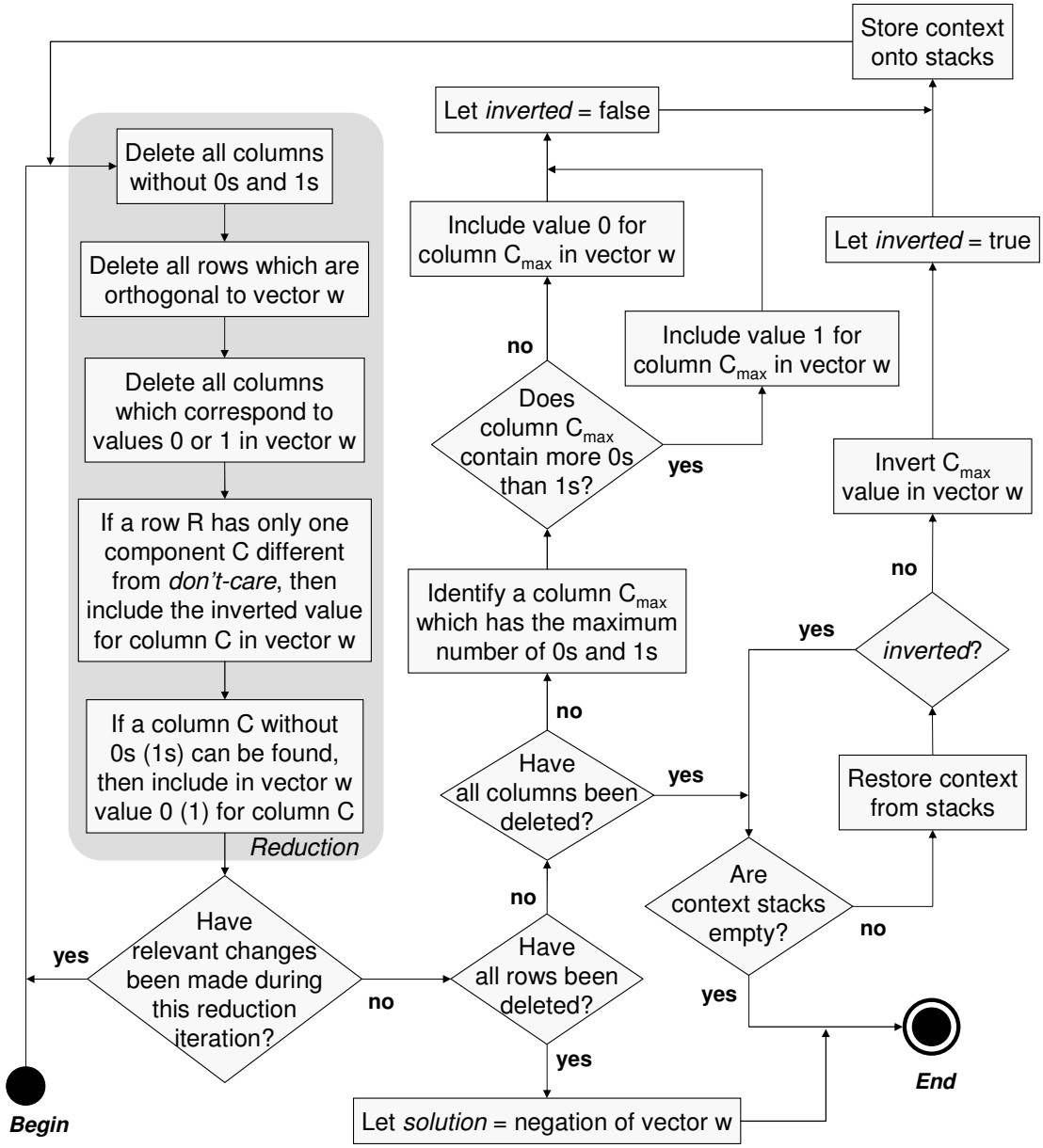


Figure 5.16 – Iterative method for solving the Boolean satisfiability problem

A relevant difference is related to the need for assigning and testing of auxiliary variables (namely *inverted* and the state of the stacks). The iterative algorithm requires such operations to correctly traverse the search tree, whereas the recursive

one does not. As a consequence of this drawback, the overall structure of the iterative diagram becomes more complex, lowering its clearness.

Note that the whole reduction cycle in both activity diagrams could be wrapped up inside a module. In the case of the recursive algorithm (see Figure 5.15), other parts of the diagram have well-defined and context-intuitive boundaries too. Thus, hierarchical modularity could be applied to such parts as well. Additionally, the use of modules enables reuse, and such is the case of the diagram blocks in Figure 5.15-b and Figure 5.15-c. Contrariwise, the complex algorithmic flow obtained with the iterative diagram (see Figure 5.16) prevents any useful modularity to be applied and, as a consequence, reuse is also not possible.

In fact, these disadvantages of using iteration are generally detected when comparing the pair of activity diagrams of all the selected algorithms (see sections 5.1.2.1-5.1.2.6).

5.1.2.3. The graph coloring algorithm

The top-level activity diagram in Figure 5.18 describes a recursive exact algorithm for solving the graph coloring problem. This activity diagram is based on the reduction and selection rules, and the solvability and resolution tests which were presented in section 3.2.4.

As previously mentioned, the graph coloring problem has no unsolvable instances. Instead of a solvability test, a condition determines whether the current (under construction) coloring may still converge into a complete coloring which would be smaller (*i.e.* include fewer colors) than that previously stored as the minimum coloring. This condition is tested in the 'Is the current coloring size too high?' node, at the top-left corner of Figure 5.17.

In the context of this algorithm, a matrix row is said to be *combinable* if it is not orthogonal to the *combination vector*. The *combination vector* stores the intersection of all the rows that were assigned the current color. In practice, *combining a row R* consists of three operations:

1. Assign row R to the current color;
2. Update the combination vector to the intersection of the current combination vector and row R;

3. Remove row R from the matrix (see reduction rule R3 in section 3.2.4).

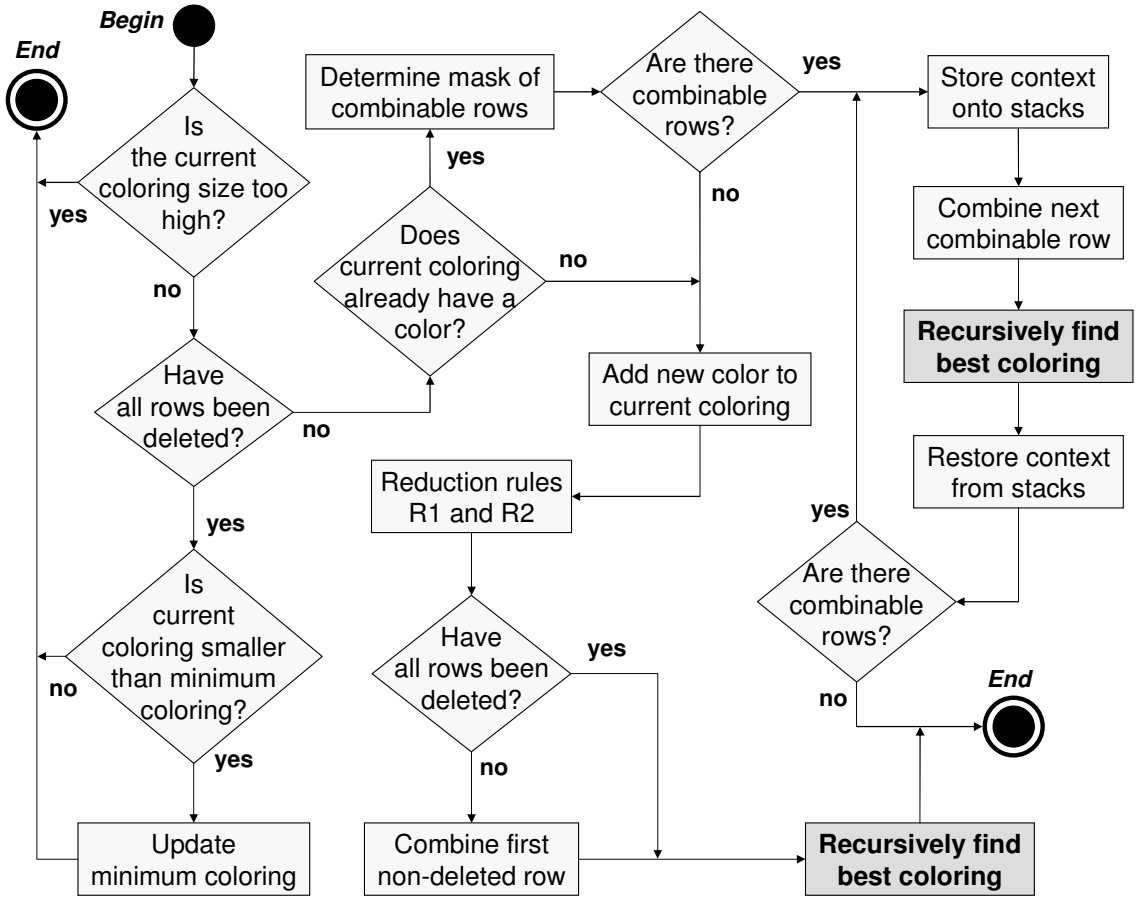


Figure 5.17 - Recursive method for finding an exact vertex coloring

The overall contour of the activity diagram conforms to the step sequence outlined in section 3.2.4.

Once again, the backtracking mechanism is supported by stacks, onto which context variables are stored before starting a new search tree branch, and from which they are restored when backtracking.

Let us now consider the top-level activity diagram in Figure 5.18. On the basis of the same reduction and selection rules, and solvability and resolution tests presented in section 3.2.4, this activity diagram describes an exact iterative algorithm for solving the graph coloring problem.

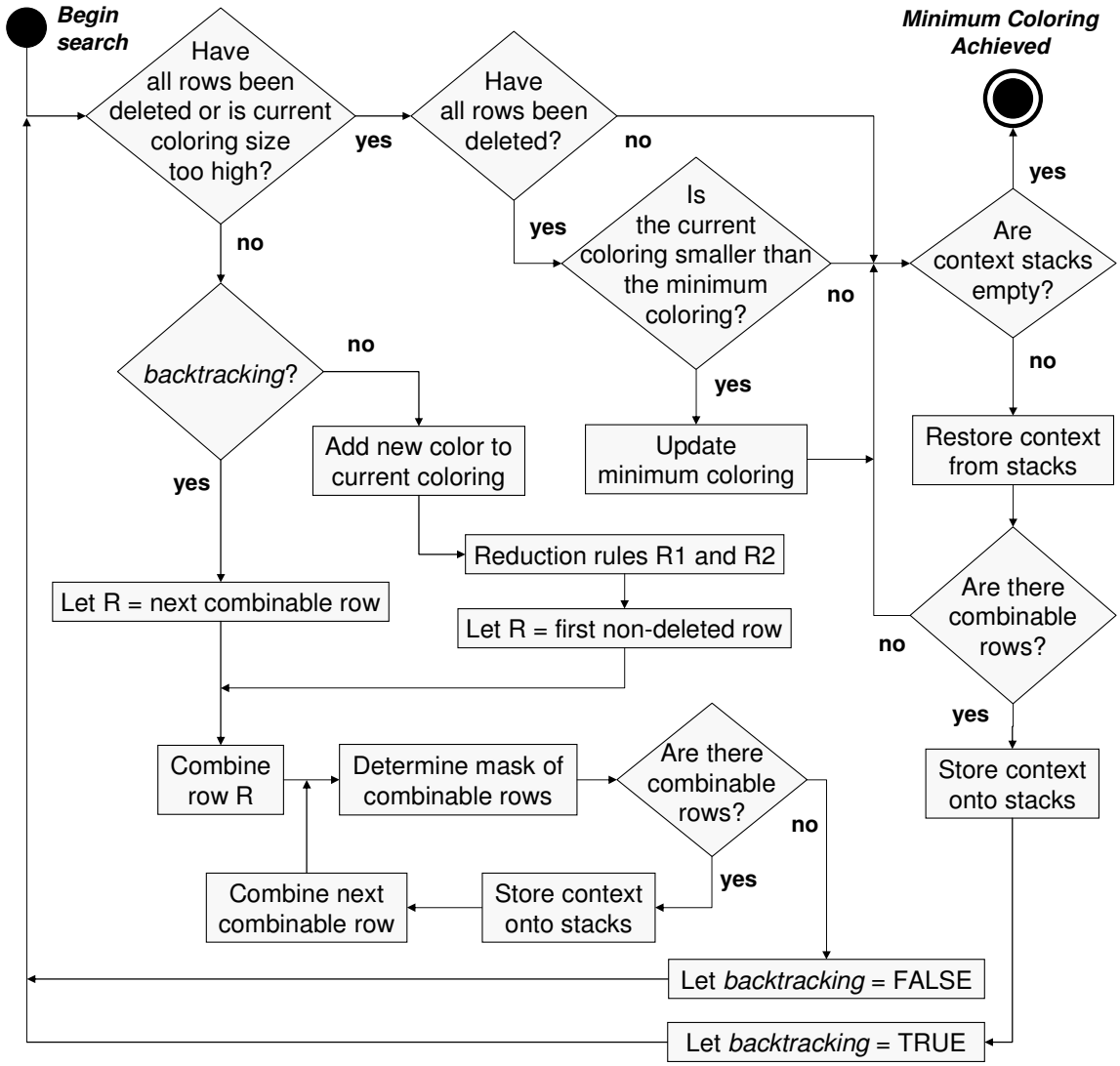


Figure 5.18 - Iterative method for finding an exact vertex coloring

5.1.2.4. The algorithm for solving the knapsack problem

The activity diagram in Figure 3.14 describes the main algorithmic flow of the recursive method for finding the most profitable knapsack configuration. Let us now examine the same algorithm in more detail with the aid of the activity diagram in Figure 5.19.

Any new item that is considered (see the 'Let I = next item' node in Figure 5.19) starts masked as not included in the current configuration. In case its weight exceeds the knapsack capacity, the inclusion path is skipped in order to try only knapsack configurations which do not include this item.

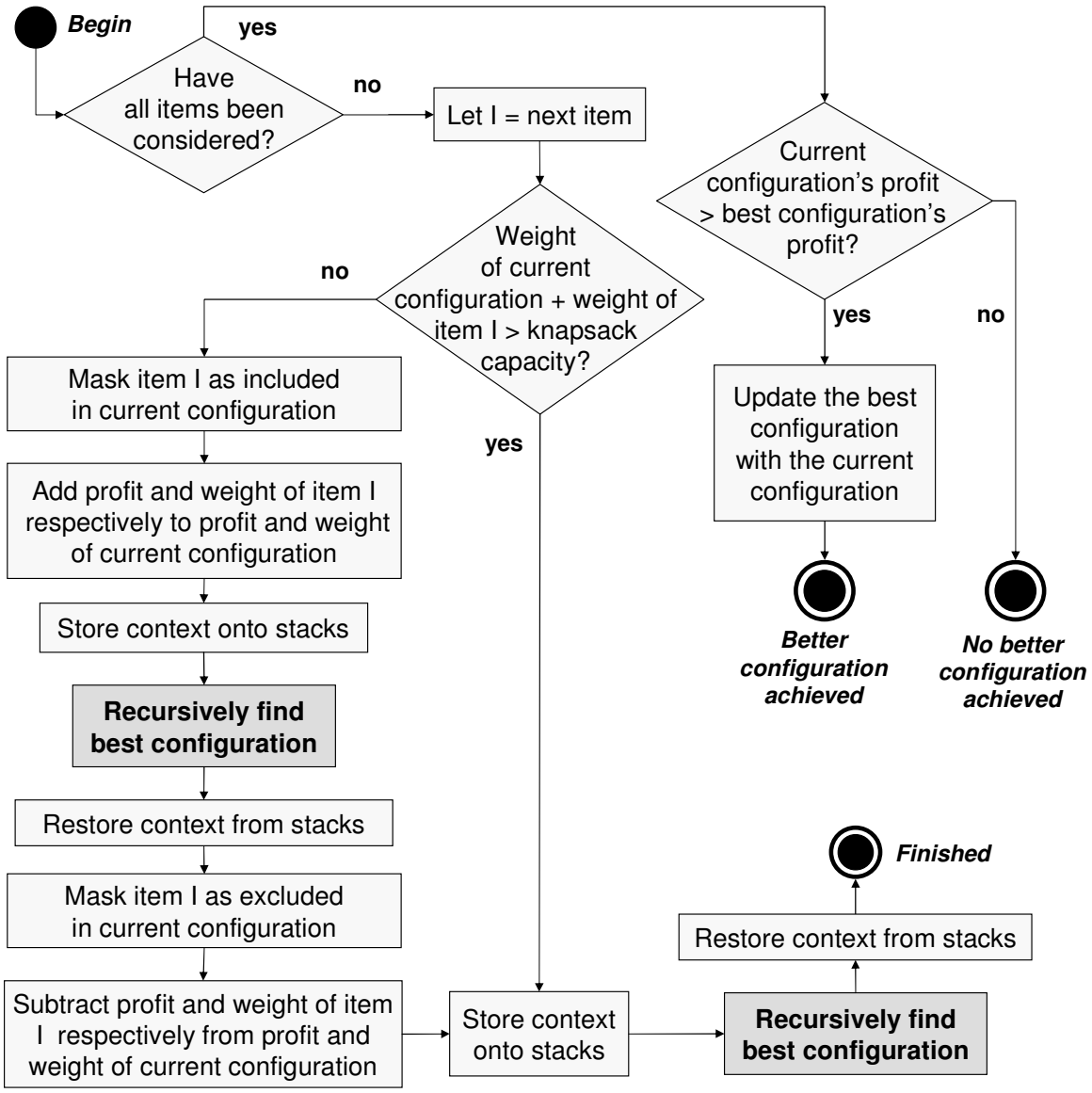


Figure 5.19 - Recursive method for finding the most profitable knapsack configuration

Analogously to the previous algorithms, backtracking is supported by executing operations for context storing and restoring respectively before and after each of the two recursive invocations.

5.1.2.5. The tree-based sorting algorithm

As previously mentioned, tree-based sorting comprehends two stages: first, build a binary sorted tree using the given data; then retrieve those values by means of in-order traversing. Each node holds a value to be sorted, a counter for accumulating multiple occurrences of that value, and two nodes which are the roots of the left and

right subtrees. If any of these two nodes is unassigned, it means the holding node does not have the corresponding subtree.

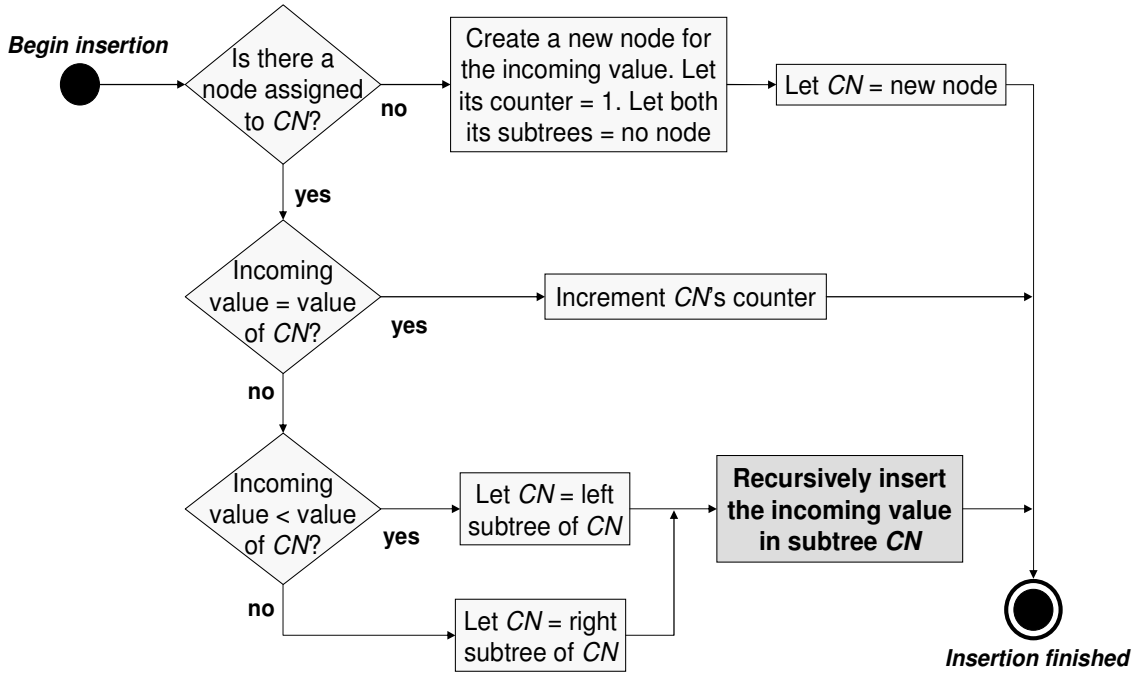


Figure 5.20 - Recursive method for inserting a value in a sorted tree

Figure 5.20 and Figure 5.21 depict the activity diagrams respectively for the recursive and iterative methods for inserting an incoming value in a sorted tree. Let us notice that both these methods make use of a variable CN ('CN' stands for 'current node') in which the node to be processed is stored. Prior to each insertion, CN is assigned the root of the sorted tree. When an incoming value must be inserted in one of the subtrees, variable CN is assigned to that subtree's root. In case this root has not been assigned (*i.e.* there is no subtree), a new node is created, thus initiating a subtree (see Figure 5.20).

The iterative method, additionally, makes use of a variable PN ('PN' standing for 'parent node'), assigning it to CN before processing one of its subtrees. This way, in case the subtree to process does not exist, it is possible to assign PN's left subtree or right subtree (whichever is the case) to a new node then created: CN (see Figure 5.21).

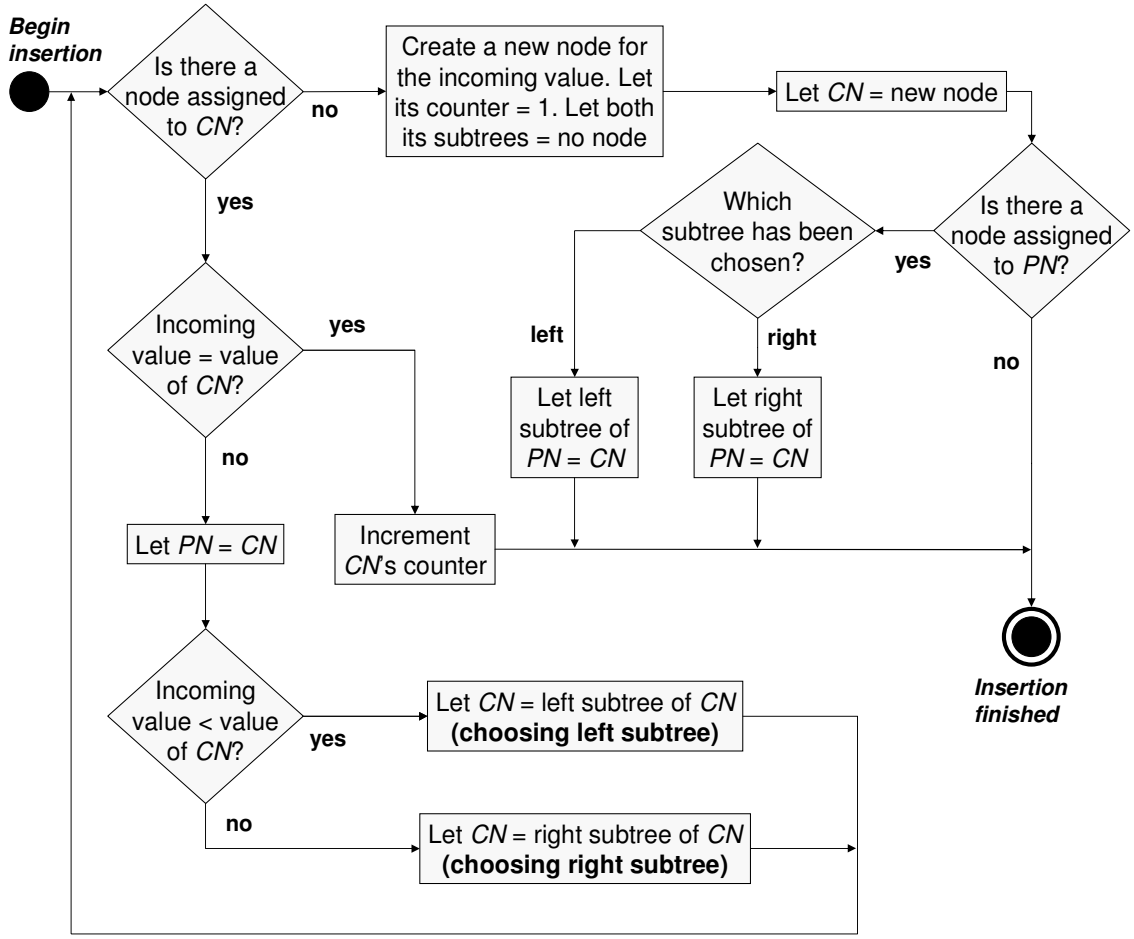


Figure 5.21 - Iterative method for inserting a value in a sorted tree

Figure 5.22 and Figure 5.23 depict the activity diagrams respectively for the recursive and the iterative methods that are used for retrieving the tree values. In both cases, the retrieval starts with variable CN assigned to the tree root.

For the iterative method, nodes must hold not only a value, a counter, and the root nodes of both its subtrees, but also a third node: the parent node (see top right corner of Figure 5.23). In this context, a tree root is therefore a node with an unassigned parent node.

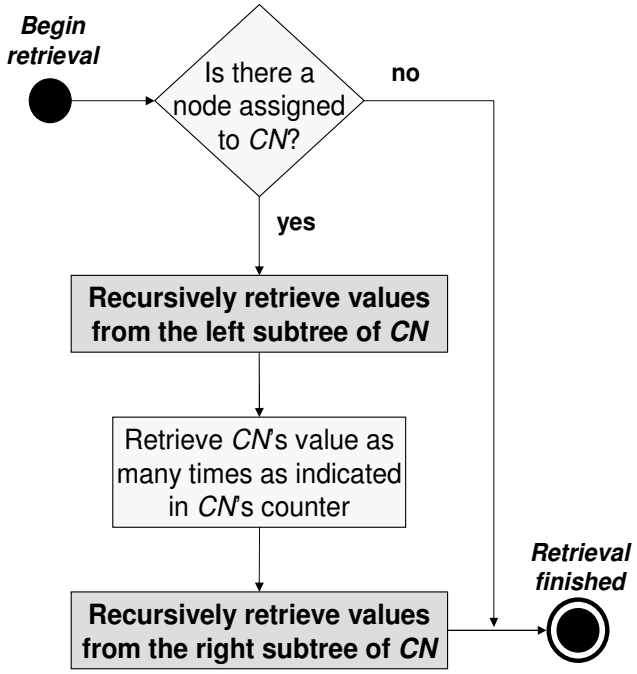


Figure 5.22 - Recursive method for retrieving tree values

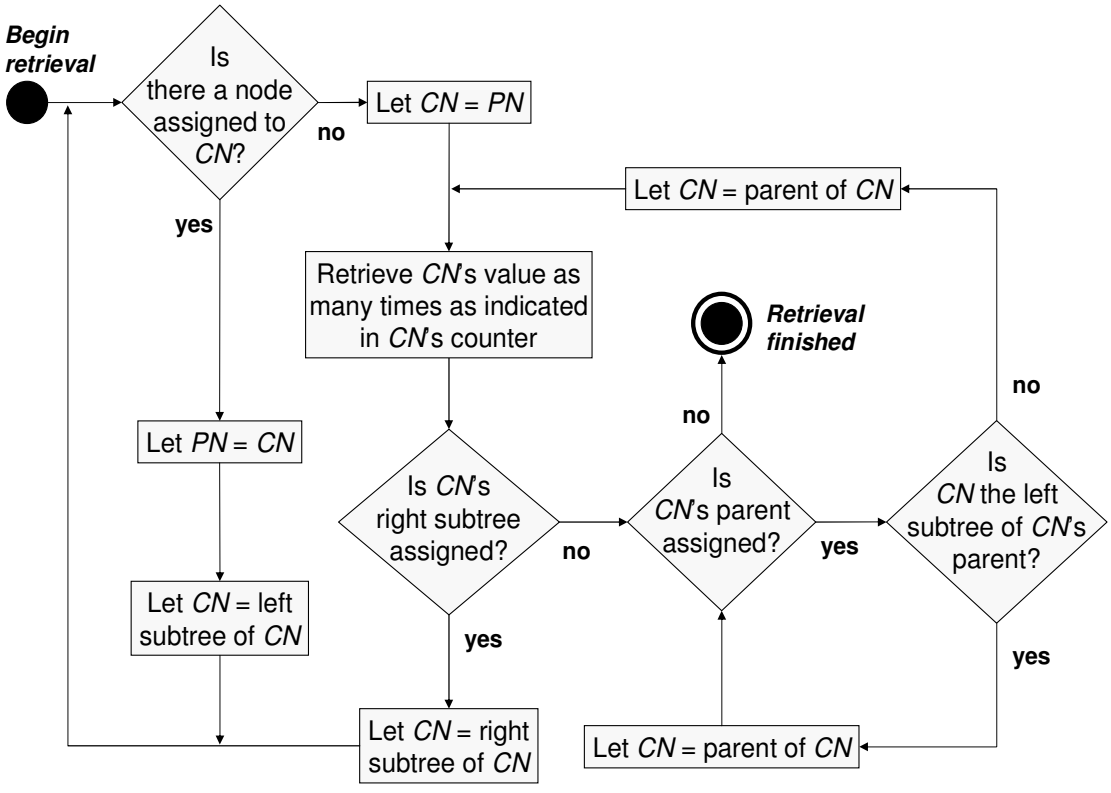


Figure 5.23 - Iterative method for retrieving tree values

5.1.2.6. The algorithm for calculating the GCD

The pseudocode for calculating the greatest common divisor of two integer values which is presented in Figure 3.18 leads to the activity diagrams in Figure 5.24. The recursive algorithm is described in Figure 5.24-a, whereas the iterative one is in Figure 5.24-b. Let us recall that keyword *mod* represents the modulo operation, which calculates the remainder of dividing the first operand by the second operand. The iterative algorithm uses an auxiliary variable T (*temp* in Figure 3.18's pseudocode) for swapping the values of A and B.

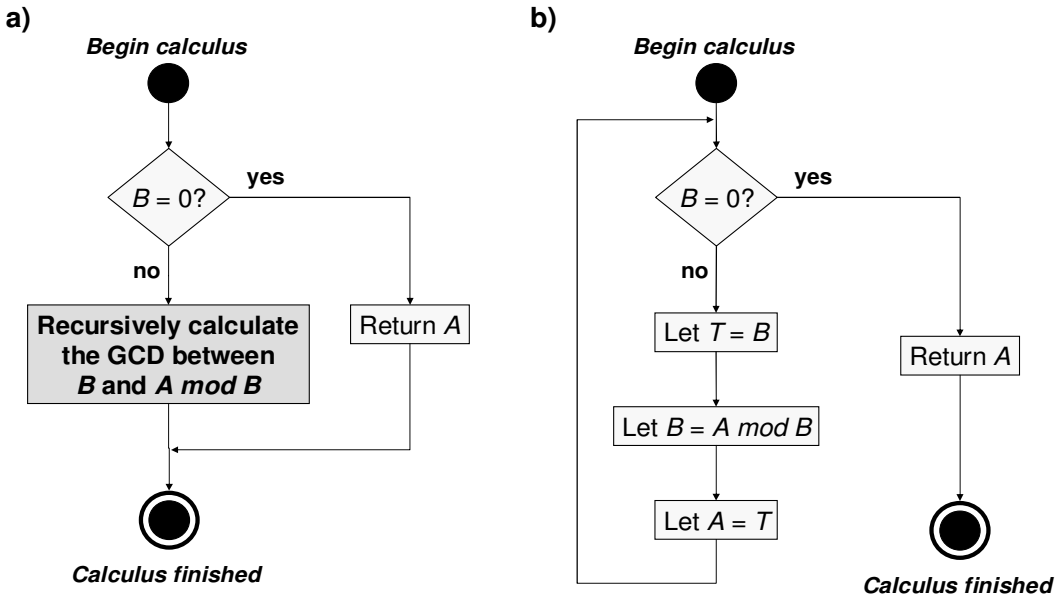


Figure 5.24 - Recursive (a) and iterative (b) algorithms for calculating the GCD of two integers A and B

5.2. Implementation in hardware

Most of the selected algorithms were implemented on the basis of the general hardware architecture depicted in Figure 5.25. A centralized control unit coordinates the execution of the required sequence of algorithmic steps. For the majority of the implemented algorithms, the data storage block is matrix-oriented and all operations over individual rows and columns are executed in the processing unit. When implementing other kinds of algorithm, different data storage blocks were used.

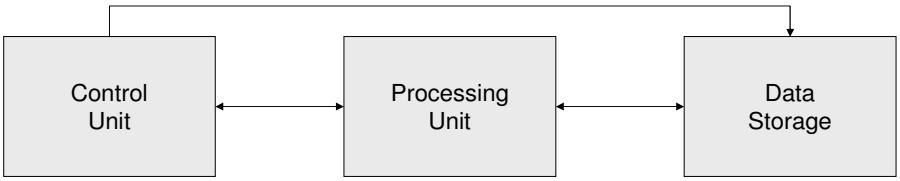


Figure 5.25 – General architecture of hardware solvers

This architecture has been used for implementing instance-specific solvers. Using such approach, the results of experiments can more expressively highlight the differences between the compared systems (for instance, regarding the resources required) because there is less interference caused by secondary system components. Thus we believe it leads to a more reliable comparison between recursive and iterative implementations.

The following subsections present the three blocks of the architecture.

5.2.1. Data storage

The architectural block for storing problem data has to be able to keep basic data structures, such as binary and ternary vectors. Let us consider such components first.

5.2.1.1. Binary vectors and ternary vectors

For storing binary vectors, the developed circuits keep arrays of bits explicitly stating the binary values. Two logic vectors are used for storing ternary vectors: one marking the position of values 0 and the other marking the position of values 1, while positions marked by none of those values correspond to *don't-care* values (see Table 5.1, in which '-'s represent *don't-care* values).

Table 5.1 - Representing binary and ternary vectors

	Binary vector	Ternary vector
Vector to store:	1 0 0 1 0 1	1 0 1 - 0 -
Storing logic vectors:	1 0 0 1 0 1	zeros: 0 1 0 0 1 0 ones: 1 0 1 0 0 0

5.2.1.2. Binary matrices and ternary matrices

Some matrix-based algorithms require binary matrices, while others use ternary matrices. Analogously to the ternary vector-storing technique (see the last column in Table 5.1), two binary matrices can be used to compose a ternary matrix. Figure 5.26 shows how a ternary matrix can be coded by two binary matrices.

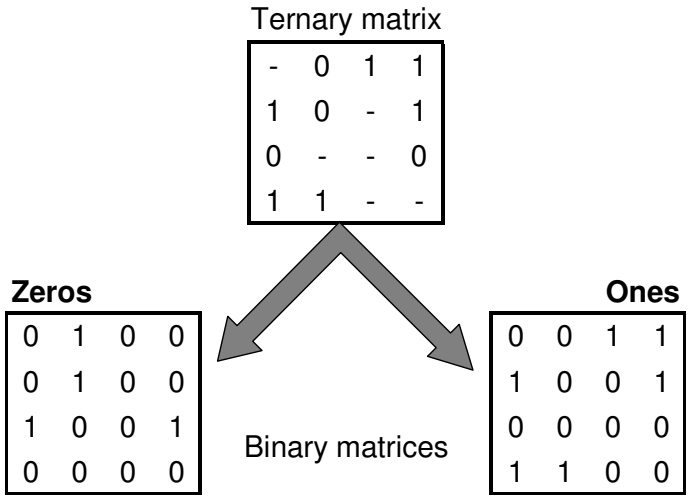


Figure 5.26 – Coding of a 4x4 ternary matrix by two binary matrices

A second criterion regarding the matrices which are required by matrix-based algorithms also divides these algorithms in two groups: one with simple access to the matrices (by either rows or columns) and the other one requiring dual access (by both rows and columns).

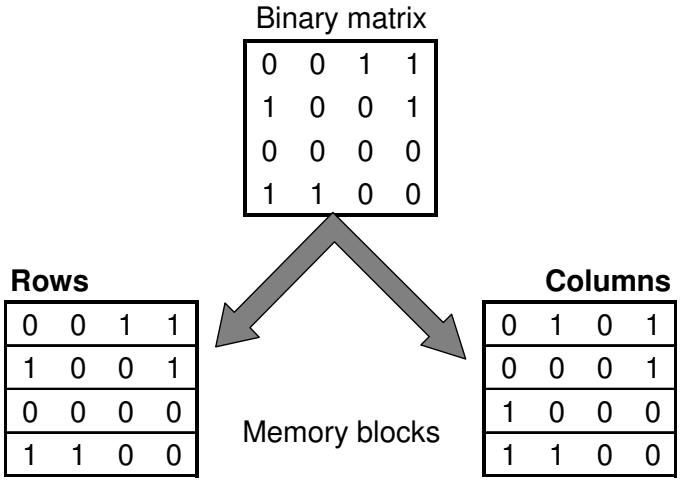


Figure 5.27 - Representation of a 4x4 binary matrix in two memory blocks

Using embedded memory blocks for keeping matrices, if we want to achieve good performance, then dual access to a matrix requires a replication of its data: one memory block can be organized as an array of rows; and the other as an array of columns. Figure 5.27 presents an example.

Let us notice that replicating a ternary matrix implies the replication of both binary matrices which result from the ternary-to-binary decomposition (see 'zeros' and 'ones' in Figure 5.26). This means that up to four memory blocks may be required to store a single ternary matrix.

Combining the two presented criteria, four solver classes emerge with direct correspondence to four kinds of matrix:

a) Single Access Binary Matrix (SABM);

b) Single Access Ternary Matrix (SATM);

c) Dual Access Binary Matrix (DABM);

d) Dual Access Ternary Matrix (DATM).

The number of embedded memory blocks used to implement binary and ternary matrices in function of the access type is presented in Table 5.2.

Table 5.2 - Number of embedded memory blocks in function of matrix and matrix access types

Access	Binary Matrices	Ternary Matrices
Simple	**SABM:** 1 memory block required	**SATM:** 2 memory blocks required
Dual	**DABM:** 2 memory blocks required	**DATM:** 4 memory blocks required

5.2.1.3. Supplementary problem-oriented data structures

With respect to the tree-based sorting algorithm implementation in hardware, every node-referencing variable contains an address which indicates the node's position within a memory block. Let us look at an example on the basis of the sorting tree built in section 5.1.1.6 (see Figure 5.10-f). Figure 5.28 illustrates the storage of the respective nodes' data in a simplified memory block.

	Value (0-15 value)	Counter (0-7 value)	Parent (ref. to a node)	Left Subtree (ref. to a node)	Right Subtree (ref. to a node)
111					
110					
101					
100	1001 (9)	001 (1)	0*010*	1111	1111
011	0101 (5)	001 (1)	0*010*	1111	1111
010	0111 (7)	010 (2)	0*000*	0*011*	0*100*
001	0010 (2)	001 (1)	0*000*	1111	1111
000	0100 (4)	001 (1)	1111	0*001*	0*010*

Figure 5.28 – Memory block with sorting tree nodes' data

The three rightmost fields of each memory word are used to store addresses of other related nodes in the same memory. Only 3 bits are required to address any of the 8 memory words, but using an extra bit permits to represent the equivalent to a null pointer ('1111' in this example), indicating that the respective child or parent node does not exist. Thus, node-referencing fields are 4-bit wide. In case one of these fields does not contain '1111', then the three rightmost bits of this field constitute the memory address at which the pointed node is stored. In Figure 5.28, numbers in parenthesis are the decimal equivalents of the stored binary numbers.

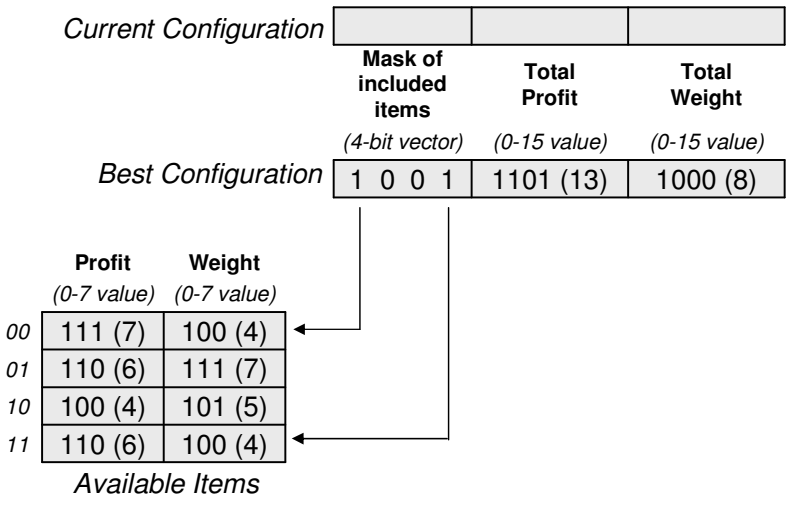


Figure 5.29 – Simplified hardware data structures for solving the knapsack problem

For solving the knapsack problem, the available items are stored in a memory block (see a simplified example at the bottom of Figure 5.29), each memory word containing the profit and weight of an item. On the other hand, the current and best knapsack configurations (see top of Figure 5.29) keep not only the total profit and

total weight but also a logic vector which is used as a mask of included items. If an item i must be included in the configuration, then the i^{th} bit of the mask is assigned to 1. Figure 5.29 illustrates the use of these data structures when applied to the Knapsack problem instance that was given as an example in section 3.2.5. Again, numbers in parenthesis are the decimal equivalents of the stored binary numbers.

5.2.2. Control unit

Independently of the implementation (*i.e.* either in software or in hardware), recursive calls invoke operations over stacks in such a way that the states of the algorithm (where recursive invocations have happened) are saved onto a stack and the stack pointer is incremented to address the storage for a recursively called sub-algorithm. When the recursive sub-algorithm ends, the stack pointer is decremented in order to restore the state of the interrupted algorithm. If we consider an equivalent iterative algorithm, such stack is not required and computations are performed in a loop, which ends as soon as some conditions are satisfied.

Iterative algorithms can be described and implemented using either *flat* or *hierarchical* specifications. In the first case, we can recur to the traditional FSM model and employ any suitable language (such as VHDL, Verilog, or Handel-C) for specifying the algorithmic steps.

The hardware FSM model is shown in Figure 5.30 [Skliarova08]. The FSM consists of a combinational circuit (that produces the primary outputs and calculates the next state on the basis of the input values and the current state) and a register that stores the current FSM state. Figure 5.30 includes a VHDL template illustrating how the combinational circuit and the FSM register can be described with the aid of two processes. The template is parameterizable and can therefore be used for describing functionality of any FSM. Figure 5.31 presents a Handel-C template for the same FSM model.

In the case of an *iterative hierarchical* specification, the algorithm description is decomposed in modules (for example, a module implementing reduction rules, a module for testing the quality of solutions, etc.). The resulting modular descriptions have a number of advantages over traditional FSMs which can be justified as follows. It is well known that the best way to simplify the problem solving process is to divide the initial problem into small, manageable parts [Skliarova08]. The resulting design will contain modules, which are self-contained circuits. Besides of simplifying the

design process, such an approach provides a direct support for reusability, since the developed modules might be reused in different parts of the project, as well as in other projects. Another very important aspect is design's modifiability. Imagine that the initial problem specification is changed after some period of time. When the project is divided in modules, incorporating changes into a single module is a simpler task than changing the implementation of the whole circuit.

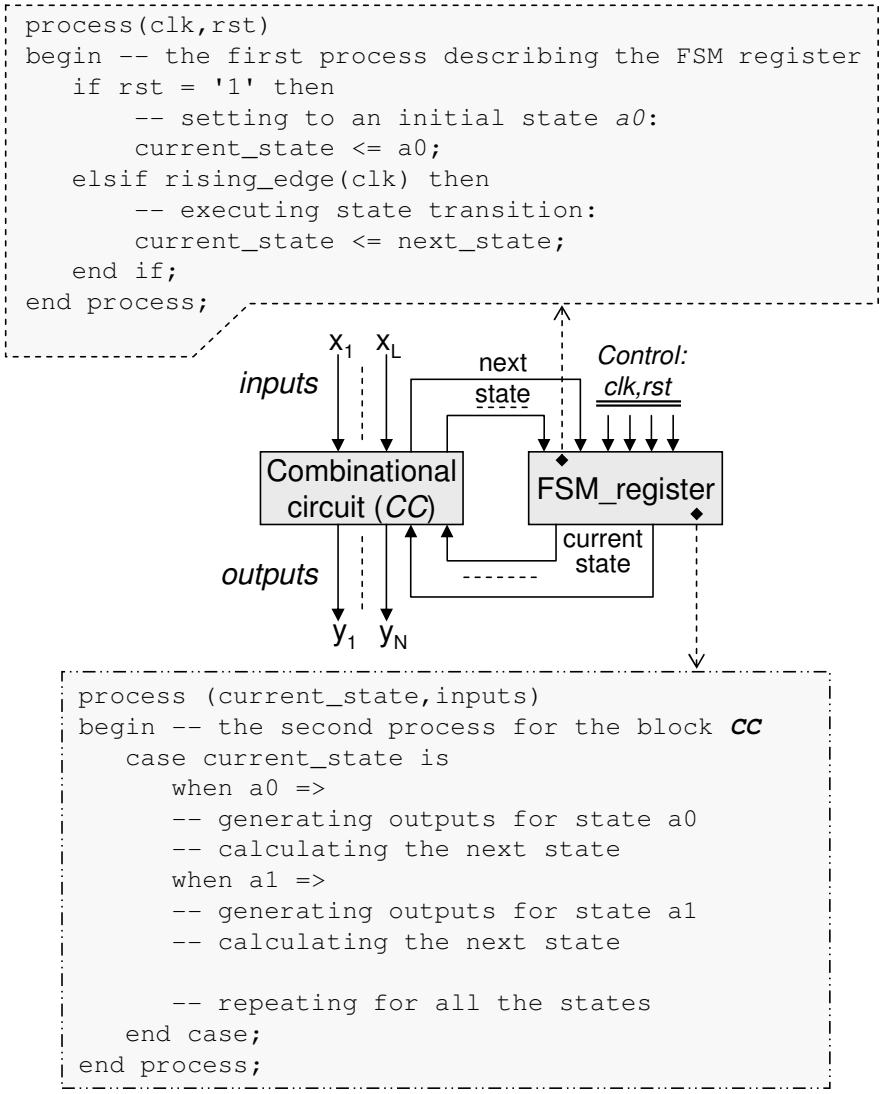


Figure 5.30 – Design template for an FSM and VHDL description

Recursive algorithms are also constructed from modules but, in this case, each module is allowed to call itself. It is well known that hardware description languages (such as VHDL) and system-level specification languages (such as Handel-C) do not provide direct support for recursive algorithms [Skliarova08]. However, recursion can be

implemented in a hierarchical finite state machine [Sklyarov99] and the latter can easily be described in hardware and system-level specification languages.

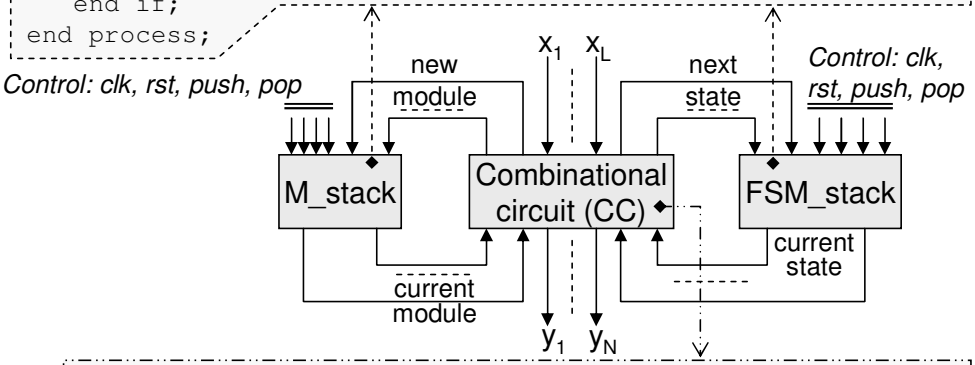
```
void FSM()
{  STATE_ID  state;
   BOOLEAN   done;
   //set to an initial state and initialize
   do
   {  switch(state)
      {  case a0:
            par
            {  //algorithm-related operations for a0
               //calculate next state
            }
            break;
         case a1:
            par
            {  //algorithm-related operations for a1
               //calculate next state
            }
            break;
         //repeating for all states
         default:
            delay;
      }
   } while(!done);
}
```

Figure 5.31 – Design template for an FSM described in Handel-C

The hardware HFSM model is depicted in Figure 5.32. The HFSM consists of a combinational circuit and two stacks (that keep track of hierarchical module invocations), one for states (*FSM_stack*) and the other for modules (*M_stack*).

The stacks are managed by a combinational circuit that is responsible for new module invocations and state transitions in any active module that is designated by the outputs of *M_stack*. Any non-hierarchical transition is performed through a change of a code only on the top register of *FSM_stack*. Any hierarchical call alters the states of both stacks in such a way that *M_stack* will store the code for the new module and two values will be pushed onto *FSM_stack*: first, the code of the next state in the calling module and then the code of the first state in the called module. Any hierarchical return just activates a pop operation without any change in the stacks. As a result, a transition to the state following the state where the terminated module was called will be performed. The stack pointer (*stack_ptr*) is common to both stacks.

```
process(clk,rst) -- the first process for the blocks
begin            -- M_stack and FSM_stack
   if rst = '1' then
       -- setting to an initial state and initializing
   elsif rising_edge(clk) then
       -- test for possible errors
       -- executing transitions of the following types
       -- a) between states within the same module
       -- b) between states that belong to different modules
   end if;
end process;
```




```
process (current_module,current_state,inputs)
begin     -- the second process for the block CC
   case M_stack(stack_ptr) is
       when z0 =>
           case FSM_stack(stack_ptr) is
               -- state transitions in the module z0
               -- generating outputs for the module z0
           end case;
       when z1 =>
           case FSM_stack(stack_ptr) is
               -- state transitions in the module z1
               -- generating outputs for the module z1
           end case;
       -- repeating for all the modules
end process;
```

Figure 5.32 – Design template for an HFSM and VHDL description

Figure 5.32 illustrates an example of VHDL code for an HFSM, which makes it possible to describe modular and recursive algorithms [Sklyarov04]. There are two concurrently executing VHDL processes in Figure 5.32. The first process describes two stacks (the stack of modules and the stack of states) and the second process describes the combinational circuit, which is able to manage transitions between the FSM modules and FSM states. It is important that the second process can easily be customized for executing any desired hierarchical algorithm [Skliarova08]. A similar template is presented in Figure 5.33 using Handel-C.

```
void HFSM()
{  MODULE_ID   module;
   STATE_ID    state;
   BOOLEAN     done;
   //set to an initial module and state and initialize
   do
   {  par
      {  module = M_stack(stack_ptr);
         state = FSM_stack(stack_ptr);
      }
      switch(module)
      {  case z0:
            switch(state)
            {  //algorithm-related operations in the module z0
               //stack management operations in the module z0
            }
            break;
         case z1:
            switch(state)
            {  //algorithm-related operations in the module z1
               //stack management operations in the module z1
            }
            break;
         //repeating for all modules
         default:
            delay;
      }
   } while(!done);
}
```

Figure 5.33 – Design template for an HFSM described in Handel-C

Although modular design incurs some overhead and therefore occupies more resources, it is not slower than non-modular dedicated design (as will be evidenced by the results of experiments given in chapter 6). Moreover, if the stacks are constructed on the basis of memory blocks embedded in FPGA, the additional FPGA resources required for stack management are negligible.

5.2.3. Processing unit

5.2.3.1. Similarities amongst matrix-based backtracking search algorithms

Matrix-based backtracking search algorithms have similar characteristics. One of them is the execution of problem-specific operations, and another one is the traversal of a search tree, starting from the root, by involving such procedures as forward search and backtracking. Any branching point can be considered as extracting a sub-tree with a local root. Moreover, because the data structures that they manipulate are basically the same, the operations used as basic blocks to implement those algorithms are, in

fact, very much the same. Examples of generally used micro-operations are the following:

- Remove a row/column;
- Read a row/column;
- Check whether two binary/ternary vectors are orthogonal;
- Intersect two binary/ternary vectors.

Let us notice that each operation that is used by matrix-based search algorithms can have variations, such as: use or not the contents of a mask register; store or not store the result; use just one vector of a binary matrix or two vectors of a ternary matrix.

There are also composed operations (groups of micro-operations) which are still very commonly used, such as:

- Find the row/column with the most/fewest 0s/1s in a matrix;
- Find the index of the first 0/1 in a binary/ternary vector;
- Count the number of rows/columns which have no 0s/1s in a matrix;
- Count 1s/0s in a binary/ternary vector;
- Check whether there are matrix rows/columns orthogonal to some binary/ternary vector;
- Intersect all rows/columns of a matrix with some binary/ternary vector.

In the end, matrix-based search algorithms possess several common features identified in [Skliarova06b]:

1. They can be formulated both recursively and iteratively.

2. They do not change the initial data (*i.e.* the initial matrix) because the matrix reduction can be provided by masking some rows/columns and using just the remainder of the matrix.

3. They invoke a very limited number of operations (such as reduction and selection operations), which have to be applied to a large volume of data.

4. Subsets of the required operations are usually not the same for different combinatorial problems.

5. In order to perform forward and backward propagation we can use a stack memory that stores and restores intermediate results (such as the values of mask registers) in branching points.

6. The algorithms can be decomposed into two levels of control operations. The top-level sequence is the same (or similar) for different algorithms. The bottom-level sequence permits the problem-specific operations over Boolean and ternary vectors to be executed.

These features make it possible to select a number of reusable blocks for constructing the processing unit. So let us first consider the hardware implementation of stacks and then address the functional blocks which are required by the processing unit architecture.

5.2.3.2. Stacks

Stacks for storing the context (masks of deleted rows, masks of selected rows, etc.) are implemented in block RAM. Multiple context variables can be stored in a single embedded memory block. For example, one block RAM of Spartan-3 FPGAs can hold 18 Kb and has a configurable width/depth ratio. Therefore, if an algorithm requires a stack with a depth up to 512 levels, 36 bits can be stored at one level. Figure 5.34-a illustrates these settings.

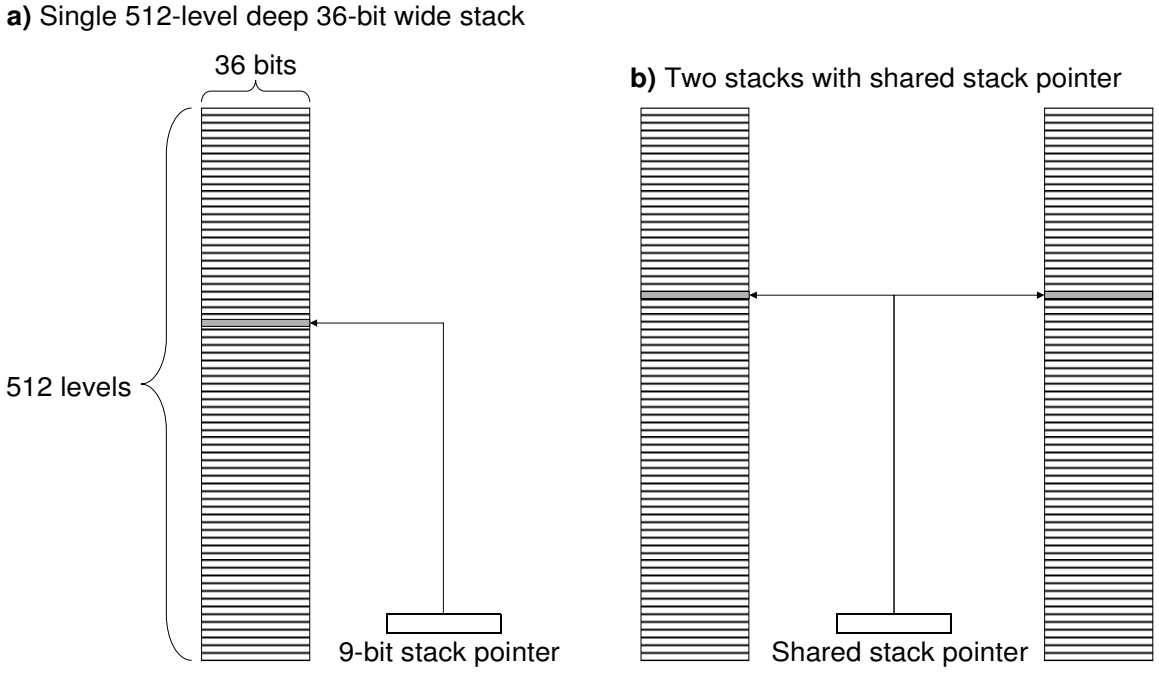


Figure 5.34 – Stacks with dedicated (a) and shared (b) stack pointers

If this number of bits is not sufficient for keeping all the context variables, multiple block RAMs have to be employed. A single stack pointer bus can be shared by different embedded memory blocks because all context variables, belonging to the same level in the search tree, must be stored/restored simultaneously (as a group). Figure 5.34-b illustrates the use of two memory blocks in order to duplicate the quantity of context data which can be stored, using a shared stack pointer bus.

5.2.3.3. Architecture for the processing unit

The architecture for the processing unit is depicted in Figure 5.35.

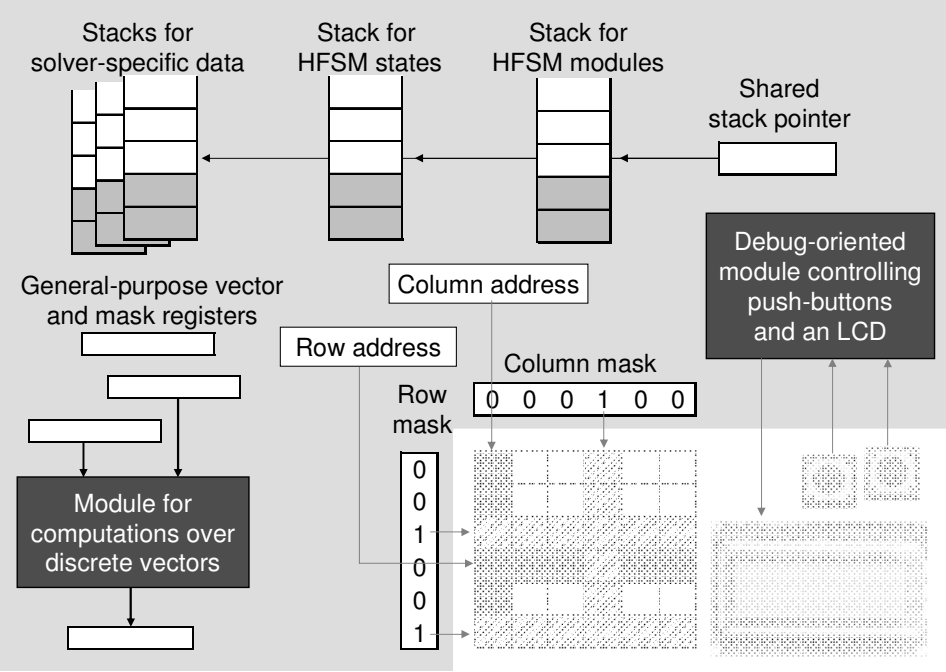


Figure 5.35 – Overview of the processing unit

The following functional blocks have been selected on the basis of analysis of different search algorithms and their primary operations:

1. Mask registers allowing to use the same storage for handling the initial matrix and all the sub-matrices, which have to be constructed during the search for results.

2. Stacks for managing forward and backward propagation steps.

3. General-purpose registers for keeping vectors.

4. A device for computations over discrete vectors.

5. Additional auxiliary circuits for testing, debugging and interacting with the hardware processor.

To make the blocks considered above reusable, we have to provide them with the property of parameterization. This property allows for scalability in such a way that the considered blocks can be applicable to matrices of different dimensions (*i.e.* with different number of rows and columns). All the implemented blocks are parameterizable in VHDL with the aid of generic and *generate* statements and in Handel-C by means of statements with the *#define* directive and parameterized macro expressions.

The proposed functional blocks take into account many specific features of the search algorithms analyzed and they have been optimized for the considered problems. It makes possible to provide block-based high-level design, *i.e.* to concentrate the efforts of the designer on the considered algorithms, avoiding (or at least minimizing) the details of hardware implementation. Since the proposed reusable blocks were implemented as a set of Handel-C macros and VHDL library modules, it allows considering either the design flow on the basis of a system-level specification language or a widely-used hardware description language.

5.2.4. Proposed architecture for a generic matrix-oriented solver

As previously mentioned, the solvers which were implemented for comparing recursion and iteration in hardware were instance-specific because such approach guarantees a more reliable comparison. However, when designing combinatorial search hardware accelerators, it is desirable to implement a problem-specific solver only once and then use it to solve different problem instances. Furthermore, a reprogrammable generic solver with the ability to implement different problem-solving algorithms can be very useful. This section suggests an extended version of the architecture depicted in Figure 5.25 for implementing systems which can be reprogrammed to execute different matrix-based algorithms for solving problem instances transferred from a host computer (see Figure 5.36). This generic architecture presents three new characteristics:

1. There is now an interface module which communicates with a computer to receive the required reconfiguration data as well as matrices to be processed, and to return the results;

2. The control unit is now reprogrammable, with the aid of a reprogrammable HFSM which has changeable functionality and can therefore be customized for implementing different algorithms.

3. The data storage block is now matrix-oriented.

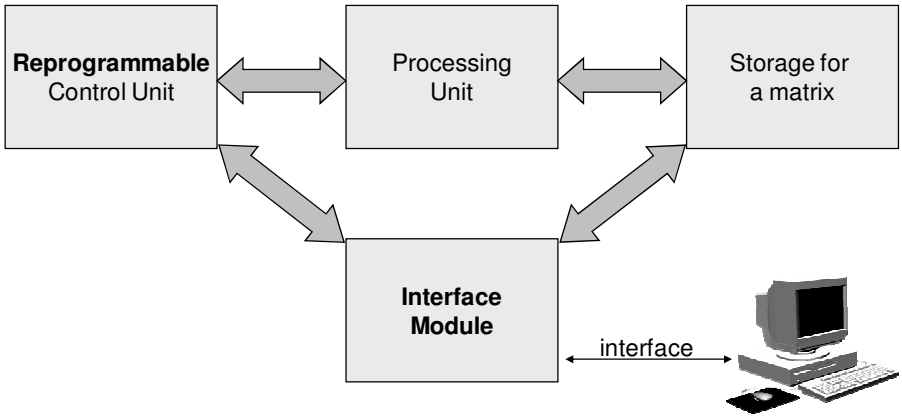


Figure 5.36 – Proposal for a generic solver architecture

In order to establish communication between the interface module and a general-purpose computer, from/to which the matrices, algorithms, and results would be received/sent, a new tool was included in PBM. This tool manages data transfer through USB or Bluetooth and it can be used in any system which requires run-time data exchange between computer and DETIUA-S3. The required hardware IP module is reusable. When designing the reprogrammable HFSM-based generic solver, debugging data can also be sent using this communication channel instead of using extra components, such as LCDs.

The reprogrammability of the control unit can be achieved with the aid of a reprogrammable HFSM, whose basic structure is depicted in Figure 5.37 [Sklyarov06c]. Reloading the RAM blocks allows for the reconfiguration of the combinational circuit's functionality, thus enabling different algorithms to be implemented. This generic solver architecture has been only partially implemented. Nevertheless, software validation and partial hardware implementation was carried out, permitting to consider Figure 5.36 as a valid architecture [Pimentel07].

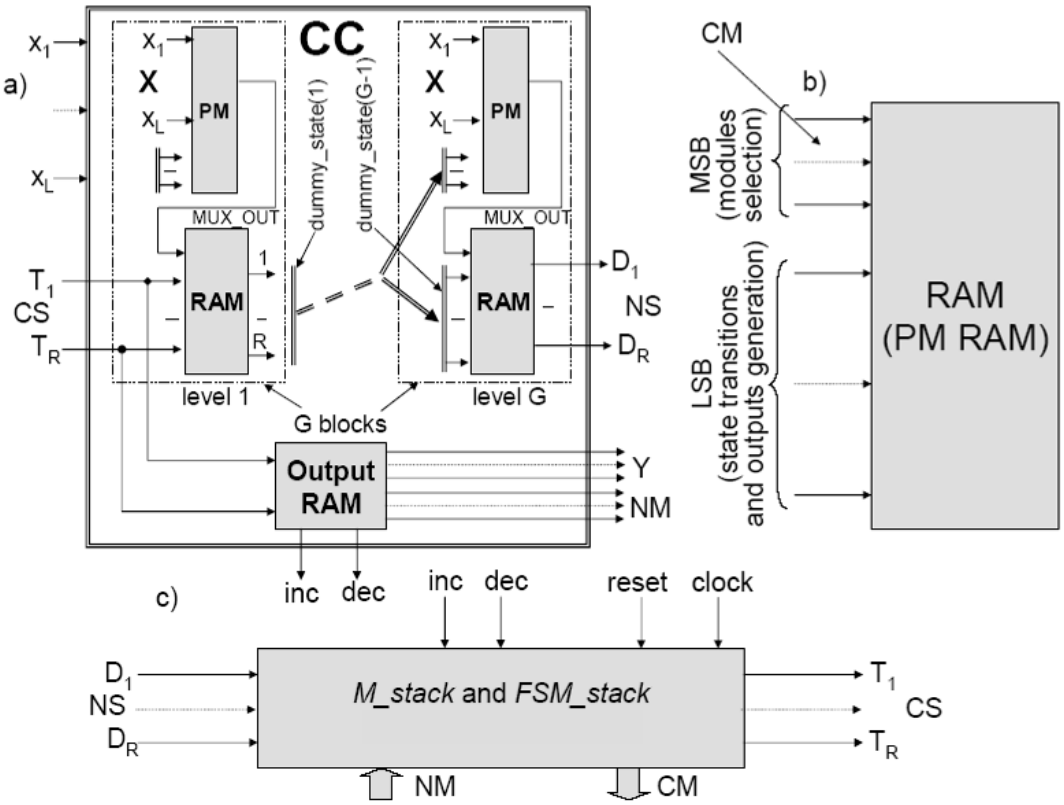


Figure 5.37 – Hardware model of a reprogrammable HFSM (from [Sklyarov06c])

The processing unit and all interaction between its functional blocks have been validated in a software application programmed in C# in which each block was described by a class that emulates the behavior expected from its hardware implementation. A special class emulated the reprogrammable control unit to validate the execution of those different algorithms.

After software validation, the architecture was implemented and successfully tested using Handel-C and the DETIUA-S3 prototyping board. The four binary vector arrays for storing the matrix were implemented using the FPGA's embedded block RAM. USB interface was used for data exchange.

The reprogrammable control unit and the interface module were then implemented in VHDL, while simulating the processing unit in C# for monitoring purposes. Some experiments with simpler algorithms were carried out and the expected run-time reprogramming was successfully achieved.

Since the proposed generic architecture has not been completely validated and tested, we plan to continue working in this direction. For now, the following has been entirely implemented, validated and tested:

- the reprogrammable HFSM. The same hard-wired RHFSM was applied for solving different problems and the customization of its behavior was achieved just through reprogramming its primary RAM blocks;
- remote customization of RHFSM functionality through wired (USB) and wireless (Bluetooth) interface;
- interaction between the customizable control unit (the RHFSM) and software models of combinatorial problem solvers. This, in particular, permits to compare iterative and recursive algorithms applied to processing units modeled in software.

5.3. Validation and implementation of the hardware accelerators

In order to study and compare different hardware implementations of the selected algorithms, the latter have first been validated in software. For this purpose, each algorithm was modeled using a high-level programming language. Only after validation, hardware solvers were synthesized from specifications in system-level and hardware description languages.

In order to validate the proposed specific architectures for implementing matrix-based backtracking search algorithms, the interaction between the most important functional blocks was also described and analyzed in a high-level language and modeled in software, allowing to estimate expected hardware behavior.

Table 5.3 lists the languages and CAD tools which have been chosen for design at different abstraction levels. Software implementations have been developed using C# and Microsoft Visual Studio with the .NET framework. Design sequence for hardware implementations based on system-level specifications included:

1. Specification in Handel-C;
2. Synthesis producing an EDIF file in Celoxica DK [Celoxica];

3. Generating a bitstream from EDIF files in Xilinx ISE implementation tools (mapping, placement and routing).

Table 5.3 - Languages and CAD tools chosen for design at different abstraction levels

Platform	Abstraction level	Languages	CAD tools
Software	High-Level Programming	C#	Microsoft Visual Studio with the .NET framework
Hardware	System-Level Specification	Handel-C	Celoxica DK and Xilinx ISE
	Register Transfer Level	VHDL	Xilinx ISE and ModelSim

Implementations based on RTL (Register Transfer Level) descriptions recurred to VHDL projects, Xilinx ISE synthesis and implementation software, and simulation tools available from ModelSim [MentorGraphics].

Two kinds of mixed specifications have also been examined. The first one combines hardware descriptions with system-level specifications. The second one relies on software/hardware co-design (co-simulation). Such mixed systems require communication mechanisms between software and hardware.

Mixed specifications allow proper selection of the most appropriate abstraction level for each system component independently. In fact, some components may require a low level description in order to achieve good performance, while others need higher level decompositions to cope with hierarchical complexity. However, the latter makes it harder to study and to compare algorithm implementations in terms of execution and design time, capabilities for modification, etc. Hence, in the scope of experiments, mixed specifications were only considered for transferring data to hardware implementations and monitoring the latter using a proper software interface running on general-purpose computers.

5.4. Conclusion

In order to estimate relative effectiveness of recursive and iterative specifications of different algorithms, as well as to check their correctness, all the algorithms were first modeled in software and only after that implemented in hardware.

When modeling in software, an object-oriented design approach was followed and thus several classes were created. Class *Vector* can keep general-purpose vectors as well as matrix rows and columns, and they can be either binary or ternary. Class *Mask* keeps a series of binary values which are used to mark their indexes, for instance, as deleted/not deleted or as selected/not selected. The use of deletion masks keeps deletion operations simple, in opposition to actual memory deallocation, and eventual row and column recovering is equally simple. Class *Matrix* provides general-purpose properties and methods which are inherited by classes implementing matrix-based algorithms, namely class *SetCoveringMatrix*, class *SATSolvingMatrix*, and class *GraphColoringMatrix*, each one providing functionality for solving its specific problem. Specific classes have also been created for the knapsack problem solver, tree-based sorting, and the calculus of the greatest common divisor. Built-in *Stack* and *Queue* class templates are used for handling any data type required by the different algorithms. In general, the data fields required by the iterative solvers outnumber those required by their recursive counterparts.

Recursive and iterative algorithmic flows were described in detail for each selected algorithm, rendering obvious the fact that, generally, the iterative algorithmic structures are more complex (and thus less clear). Furthermore, iterative algorithms often reveal the need for auxiliary variables to correctly traverse the relevant search tree, whereas the recursive ones do not. Finally, recursion favors modularity and consequently reuse, whilst the iterative algorithmic structures are often too complex to do the same.

The selected algorithms were then described in Handel-C and in VHDL, and implemented in instance-specific solvers. With such approach, the results of experiments can more expressively highlight the differences between the compared systems because there is less interference caused by secondary system components. Thus we believe it leads to a more reliable comparison between recursive and iterative implementations. Most of the solvers are based on a general hardware architecture which consists of a *control unit*, a *processing unit* and a *data storage* block.

For storing binary vectors, the developed circuits keep arrays of bits explicitly stating the binary values. Two logic vectors are used for storing ternary vectors: one marking the position of values 0 and the other marking the position of values 1, while positions marked by none of those values correspond to *don't-care* values. Two binary vectors can be used to compose a ternary vector. Analogously, two binary matrices can be used to compose a ternary matrix. Some matrix-based algorithms require simple access to the matrices (by either rows or columns) while others require dual access (by both rows and columns). Using embedded memory blocks for keeping matrices, the need for dual access to a matrix leads to a replication of its data: one memory block can be organized as an array of rows; and the other as an array of columns. Replicating a ternary matrix implies the replication of both binary matrices which result from the ternary-to-binary decomposition.

For implementing the tree-based sorting algorithm in hardware, every node-referencing variable contains an address which indicates the node's position within a memory block. The node data is organized in five fields: *value*, *counter*, *left subtree*, and *right subtree*. The last three fields are, again, addresses which indicate node positions within that same memory block. For solving the knapsack problem in hardware, the available items are stored in a memory block, each memory word containing the profit and weight of an item. The current and best knapsack configurations keep not only the total profit and total weight, but also a logic vector which is used as a mask of included items.

It is well known that neither hardware description languages nor system-level specification languages provide direct support for recursive algorithms. However, recursion can be implemented in a hierarchical finite state machine. The control unit is therefore based on a hardware model of an HFSM, consisting of a combinational circuit and two stacks that keep track of hierarchical module invocations: one stack for states and one stack for modules. The stacks are managed by a combinational circuit that is responsible for new module invocations and state transitions in any active module that is designated by the outputs of the stack for modules. Non-hierarchical transitions require access to the stack for states only, whereas hierarchical calls and returns access both stacks. A single stack pointer is used for addressing both stacks. Such a control unit allows to describe not only modular and hierarchical but also recursive algorithms.

Matrix-based backtracking search algorithms have similar characteristics. One of them is the execution of problem-specific operations, and another one is the traversal of a

search tree, starting from the root, by involving such procedures as forward search and backtracking. Any branching point can be considered as extracting a sub-tree with a local root. Moreover, because the data structures that they manipulate are basically the same, the operations used as basic blocks to implement those algorithms are, in fact, very much the same. Such similarities make it possible to select a number of reusable blocks for constructing the processing unit.

Stacks for storing the context are implemented in block RAM. Multiple context variables can be stored in a single embedded memory block. However, if one is not sufficient, multiple block RAMs have to be employed. A single stack pointer bus can be shared by different embedded memory blocks because all context variables, belonging to the same level in the search tree, must be stored/restored simultaneously.

The chosen architecture for the processing unit includes mask registers, stacks, general-purpose registers, a device for computations over discrete vectors, and additional auxiliary circuits for testing, debugging and interacting with the hardware processor. All the implemented blocks are parameterizable in VHDL with the aid of generic and *generate* statements and in Handel-C by means of statements with the *#define* directive and parameterized macro expressions.

The proposed functional blocks provide support for block-based high-level design. Since the proposed reusable blocks were implemented as a set of Handel-C macros and VHDL library modules, it allows considering either the design flow on the basis of a system-level specification language or a widely-used hardware description language.

The solvers which were implemented on the basis of the hardware blocks described for comparing recursion and iteration in hardware are instance-specific because such approach guarantees a more reliable comparison. In addition, an extended architecture has been suggested for designing reprogrammable generic solvers with the ability to implement different matrix-oriented algorithms. On the basis of the instant-specific architecture used, the extended version also includes an *interface module*, which communicates with a computer, and its control unit is now reprogrammable. Such reprogrammability can be achieved with the aid of the Reprogrammable HFSM model. Reloading the RAM blocks allows for the reconfiguration of the combinational circuit's functionality, thus enabling different algorithms to be implemented.

6. Experiments, Results, and Analysis

This chapter presents the results of experiments and analysis. The following general approach is used. In the beginning, a limited number of problems are examined and a comparison of relevant recursive and iterative algorithms is done. Synthesis was carried out from system level-specification (namely Handel-C) and hardware-level description (namely VHDL) languages. Similar comparison for the same problems was done in software using a general-purpose programming language (C#). The obtained results were compared and it was found that a similar tendency is taking place for particular algorithms described at different levels of abstraction. For example, the results of synthesis has revealed that recursive VHDL-based implementations are either equally or more advantageous than iterative VHDL-based implementations. On the other hand, recursive implementations of the same algorithms in software were always worse (in terms of execution time) when compared to iterative implementations in software. Analysis of different algorithms permitted to draw out algorithmic characteristics that allow potential benefits to be predicted. For example, particularities of hardware implementations permit to benefit from fast stack unwinding. We can measure potential acceleration taking into account some algorithmic features (e.g. the number of unwinding steps) and this can be done just in software. This tendency has always taken place for the considered backtracking search algorithms. Thus, for many experiments it was possible to avoid very complicated design and implementation steps required for synthesis of hardware and rely considerably on modeling in software. Finally, only some of the studied problems (see chapter 3) are implemented and tested in hardware and the remaining problems are modeled just in software in order to validate correctness of the respective algorithms.

6.1. Experiments and comparison of iterative and recursive implementations in hardware

In order to compare iterative and recursive hardware implementations, a set of experiments executing a subset of the selected algorithms have been carried out [Sklyarov05]. The four algorithms which have been implemented within the scope of this set of experiments are identified in Table 6.1:

Table 6.1 – Algorithms implemented in hardware for comparison

Algorithm	Describing section
Algorithm for **sorting** based on a binary tree	3.3.1
Approximate algorithm for solving the **set covering** problem	3.2.2
Exact algorithm for solving the **knapsack** problem	3.2.5
Calculus of the **greatest common divisor** between two integers	3.3.2

Each of the four algorithms of this experiment set has been implemented on the basis of both recursive and iterative descriptions. After careful modeling and debugging in C#, each of the eight resulting algorithms has been specified both in VHDL and in Handel-C, with the exception of the approximate set covering algorithm which was specified solely in Handel-C.

Moreover, different versions of VHDL-based implementations have been prepared in order to obtain some additional criteria regarding:

a) design modularity, *i.e.*, hierarchical decomposition of the algorithm in self-contained sub-tasks that are performed in sequence by means of an HFSM;

b) embedded memory usage, in order to compare implementations which use block RAM vs. distributed RAM vs. pure logic (no RAM).

On the other hand, Handel-C projects differ only in the addressed problem, and in whether they implement the algorithm recursively or iteratively.

It should be noticed that recursive solutions must always be modular, as recursion is achieved by means of an HFSM (see section 2.2.2).

The projects provide generic parameters for initial data and can therefore be customized. Note that each circuit includes not only components that are needed for comparison, but also auxiliary blocks for visualizing the results. Let us designate by *experiment* the set of different implementations of a particular algorithm, described in a particular language (*i.e.* in either VHDL or Handel-C). It is important to notice that, for each experiment, the auxiliary components used in its different implementations are exactly the same. This way, the results such as the amount of required resources and the execution time that are obtained in each experiment constitute valid data for comparison [Sklyarov05].

At the time of these experiments, not all software and hardware tools which are mentioned in chapter 4 were available. In order to provide an accurate context for result analysis, Table 6.2 presents the tools which have actually been used to carry out this experiment set.

Table 6.2 - Prototyping tools used for algorithm implementation and comparison

<table>
<tr><th></th><th>Algorithm</th><th>Synthesis CAD tool</th><th>Implementation CAD tool</th><th>Prototyping board</th><th>FPGA</th></tr>
<tr><td rowspan="3">VHDL</td><td>Tree-based Sorter</td><td rowspan="3" colspan="2">Xilinx
ISE</td><td rowspan="3">Trenz
TE-XC2Se
[Trenz]</td><td rowspan="3">Xilinx
xc2S400e-6ft256
(Spartan-IIE family)</td></tr>
<tr><td>Knapsack Problem Solver</td></tr>
<tr><td>GCD Calculator</td></tr>
<tr><td rowspan="4">Handel-C</td><td>Tree-based Sorter</td><td rowspan="4">Celoxica
DK</td><td rowspan="4">Xilinx
ISE</td><td>Celoxica
RC200</td><td>Xilinx
xc2v1000-4fg456
(Virtex-II family)</td></tr>
<tr><td>Set Covering Problem Solver</td><td rowspan="3">Celoxica
RC100</td><td rowspan="3">Xilinx
xc2s200-5fg456
(Spartan-II family)</td></tr>
<tr><td>Knapsack Problem Solver</td></tr>
<tr><td>GCD Calculator</td></tr>
</table>

6.1.1. Experiment results

The results of the experiments based on VHDL and on Handel-C are presented in Table 6.3 and Table 6.4, respectively.

Table 6.3 - VHDL-based experiment results

Algorithm	Modularity	Memory usage	Algorithm description	Number of occupied FPGA slices	Maximum attainable clock frequency (MHz)	Number of clock cycles to solve the problem	Time required to solve the problem (ns)
Tree-based Sorter	No	Pure logic	Iterative	443	35.1	70	1994
	Yes		Recursive	623	74.8	72	963
			Iterative	599	76.0	87	1145
		Block	Recursive	474	52.2	72	1379
			Iterative	473	70.2	87	1239
		Distributed	Recursive	477	58.8	72	1224
Knapsack Problem Solver	No	Pure logic	Iterative	153	59.9	88	1469
	Yes		Recursive	165	37.3	62	1662
			Iterative	-	-	-	-
		Block	Recursive	149	40.3	62	1538
			Iterative	-	-	-	-
		Distributed	Recursive	150	43.1	62	1438
GCD Calculator	No	Pure logic	Iterative	448	41.3	9	217
	Yes		Recursive	515	42	11	261
			Iterative	-	-	-	-
		Block	Recursive	454	43.5	11	252
			Iterative	-	-	-	-
		Distributed	Recursive	454	42.4	11	259

The greatest common divisor calculator has been tested for many pairs of unsigned integers. Although different numbers produce different results, the ratio between recursive and iterative implementations for each measured parameter is nearly the same. For this reason, Table 6.3 and Table 6.4 present the results regarding a single pair of numbers: 189 and 135 (27 being the result). A similar approach has been used for the tree-based sorting experiments and the corresponding data that are shown in those tables refer to the following input sequence: 30-14-9-7-13-37-2-8-17-21. The presented results regarding the knapsack problem implementations are given for 5 objects. The chosen binary matrix dimensions for the set covering problem are 128

rows by 128 columns. In total, 9 different randomly generated problem instances have been tested, generating similar results. For this reason, the results for only one problem instance are shown in Table 6.4.

Table 6.4 - Handel-C-based experiment results

Algorithm	Algorithm description	Number of occupied FPGA slices	Maximum attainable clock frequency (MHz)	Number of clock cycles to solve the problem	Time required to solve the problem (ns)
Tree-based Sorter	Recursive	1293	37.3	73	1953
	Iterative	750	45.5	61	1340
Set Covering Problem Solver	Recursive	5118	25.1	182700	7.28×10^6
	Iterative	5118	25.1	182688	7.28×10^6
Knapsack Problem Solver	Recursive	624	31.5	265	8407
	Iterative	228	36.7	474	12915
GCD Calculator	Recursive	242	16.4	6	365
	Iterative	234	16.3	6	367

6.1.2. Result analysis

The comparison between implementations based on hardware description specifications and implementations based on system-level specifications is not a target of the experiment set. In fact, the auxiliary circuits synthesized in VHDL projects are very different from those synthesized in Handel-C projects. Besides, different prototyping boards and FPGAs have been used for VHDL and Handel-C projects (see Table 6.2). Thus, only the comparison between implementations based on the same language is relevant.

6.1.2.1. Experiments based on hardware description specifications

The graph in Figure 6.1 helps comparing the number of FPGA slices that are occupied by different circuits which have been implemented on the basis of VHDL descriptions.

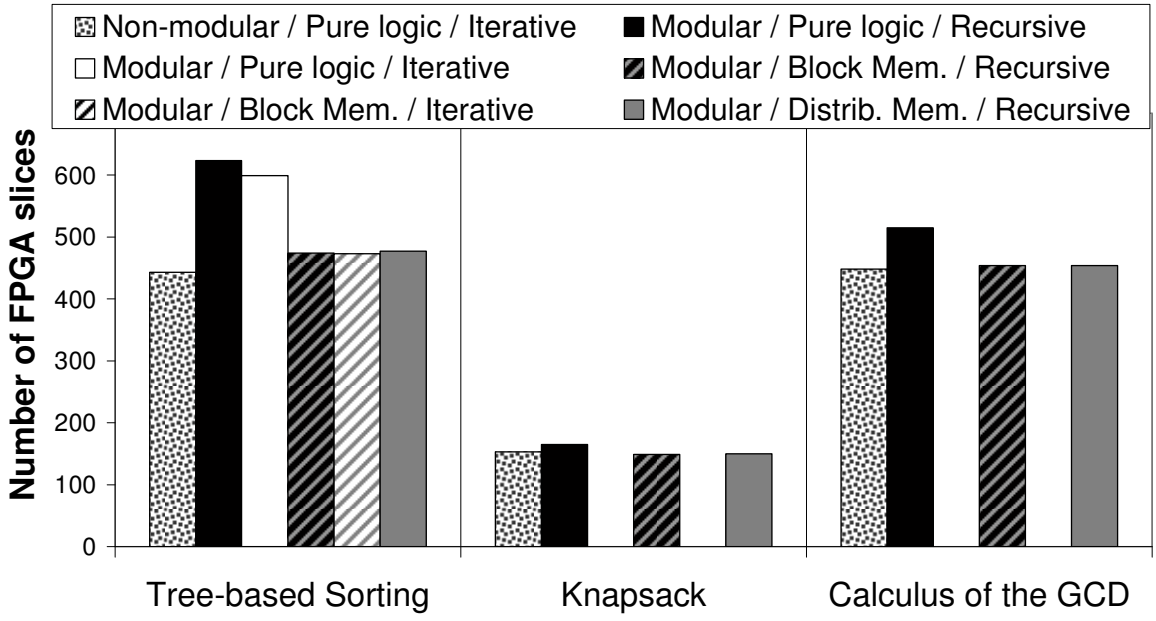


Figure 6.1 - Number of FPGA slices occupied by VHDL-based implementations

The results depicted in Figure 6.1 show no relevant differences between the number of occupied FPGA slices of recursive and iterative implementations.

However, let us notice that, amongst the pure logic implementations, the modular ones present significant overheads in the number of occupied FPGA slices. This drawback can be imputed to the circuitry supporting the HFSM, namely the stacks for storing module and state codes. However, such overhead practically disappears when making use of block or distributed memory, to which most of this support circuitry is synthesized.

The graph in Figure 6.2 highlights the differences in the maximum clock frequency which is attainable for the various VHDL-based experiments.

The results that have been obtained for the maximum clock frequency do not reveal obvious dependency on either modularity, use of recursion, or memory usage. Thus, no algorithm-independent criteria for this parameter can be identified.

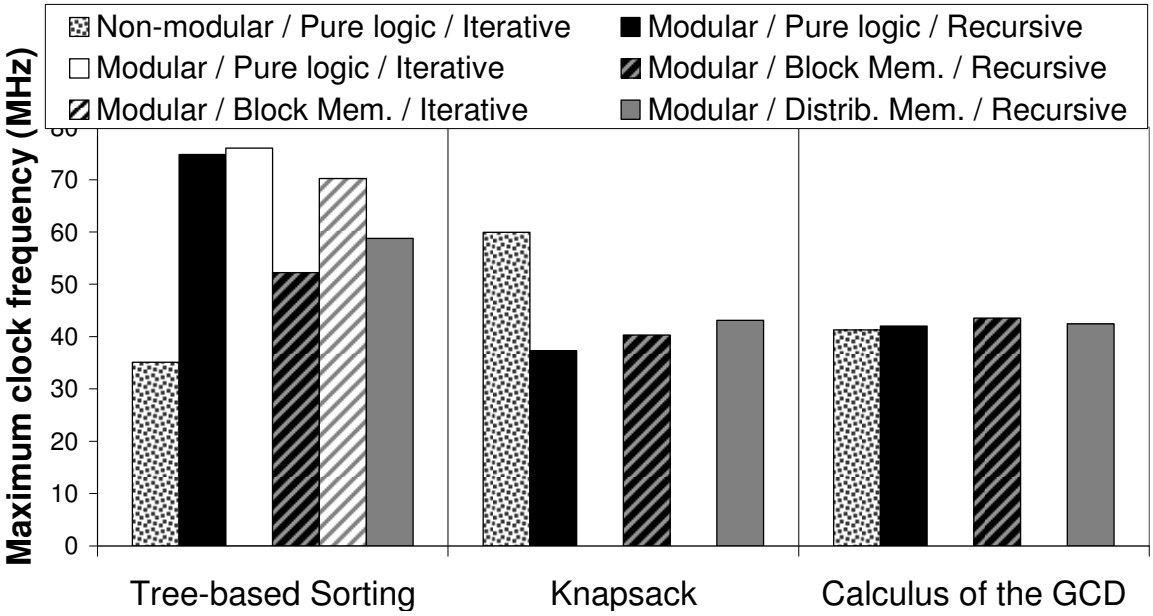


Figure 6.2 - Maximum clock frequency allowed on the VHDL-based implementations

The number of clock cycles which are required to solve the problem instances in each VHDL-based experiment is set side by side in Figure 6.3 for comparison.

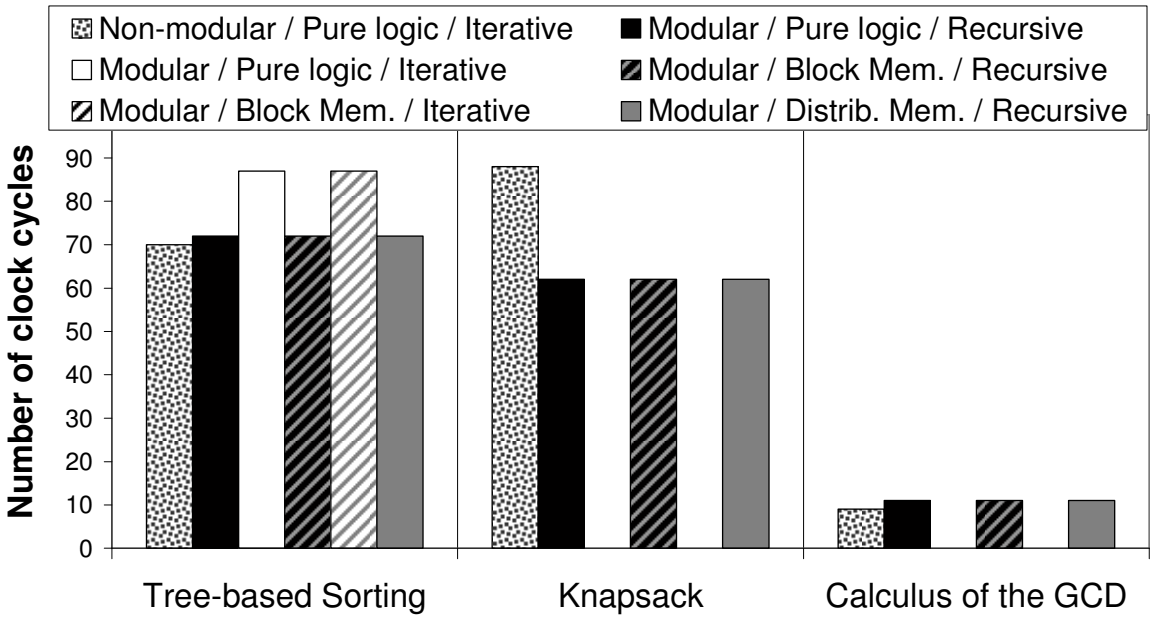


Figure 6.3 - Number of clock cycles used for solving the problem on the VHDL-based implementations

As one would expect, the use of pure logic, distributed memory, or block memory had no influence on the number of clock cycles that are required for solving a problem. On the other hand, modularity reveals some influence in respect to this parameter. However, no algorithm-independent criteria can be deduced on the basis of design modularity because the kind of influence depends on the associated problem.

Recursive implementations of tree-based algorithmic flows required fewer clock cycles to solve problems than iterative ones. For example, the modular implementations of tree-based iterative sorters required nearly 21% more clock cycles than their recursive equivalents, whether using block memory or pure logic. For solving the Knapsack problem, the pure logic non-modular iterative implementation required practically 42% more clock cycles than any of the three recursive versions.

Last, the graph in Figure 6.4 emphasizes the differences in time that is necessary for solving the problem on the various VHDL-based experiments.

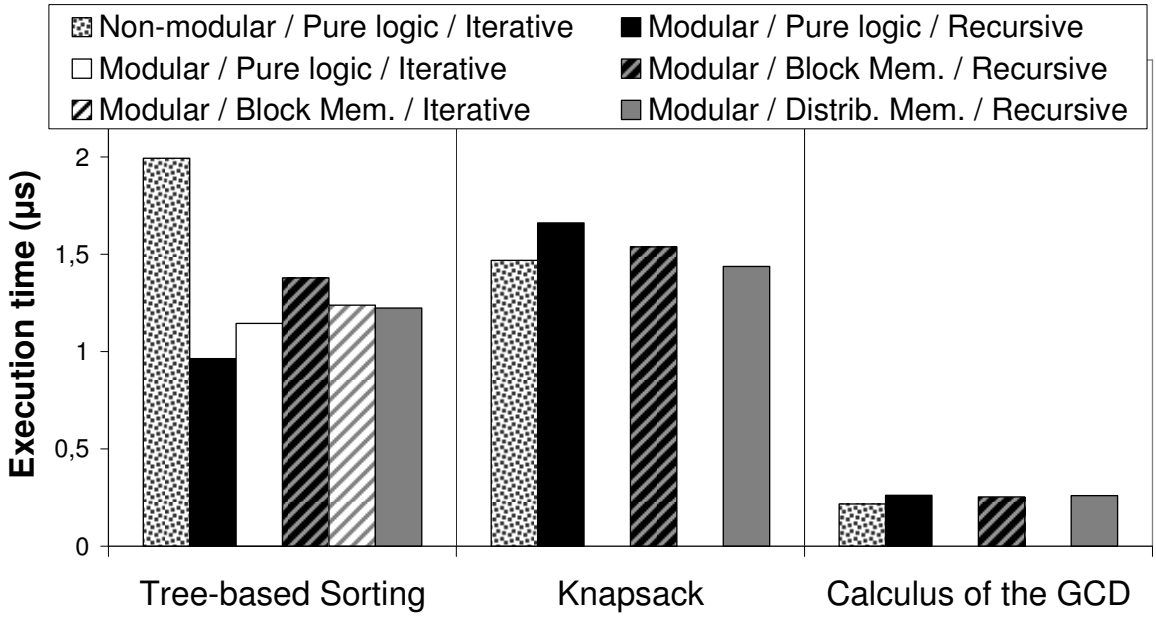


Figure 6.4 - Time required by the VHDL-based implementations for solving the problem

The results regarding the time required for problem solving reveal no dependency on either modularity, use of recursion, or memory usage. For this reason, no general criteria can be inferred for this parameter.

6.1.2.2. Experiments based on system-level specifications

The graph in Figure 6.5 permits to compare the numbers of FPGA slices that are occupied by the different circuits which have been implemented on the basis of Handel-C descriptions.

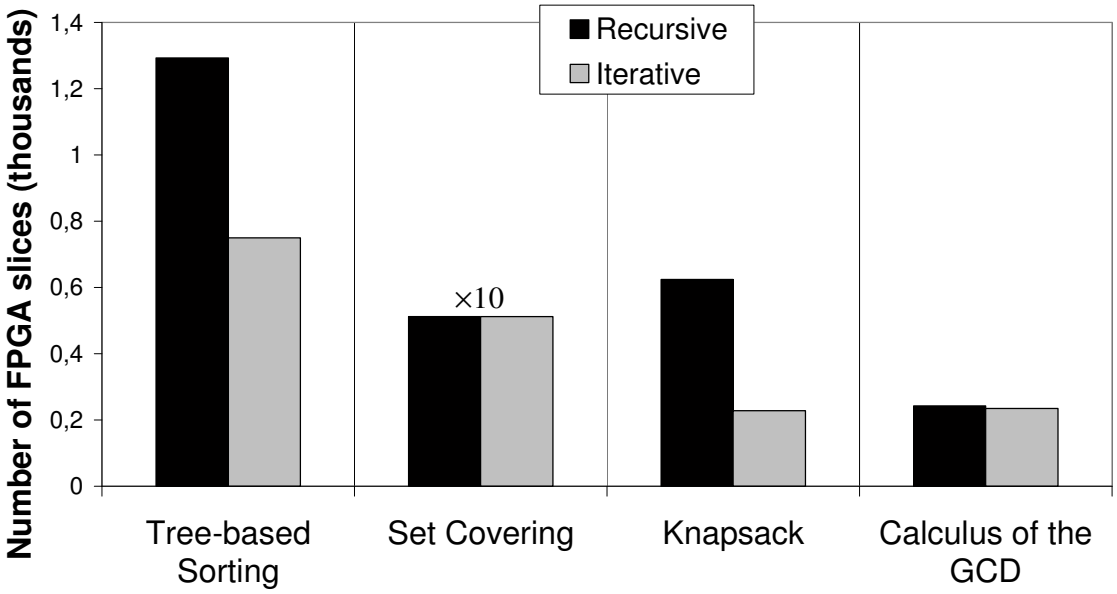


Figure 6.5 - Number of FPGA slices occupied by Handel-C-based implementations

The results of the Handel-C projects reveal that the number of FPGA slices that are required for non-backtracking algorithms (namely sorting, knapsack, and CGD) is higher in recursive implementations than in iterative ones. This drawback seems to be related to the use of stacks. In the case of the algorithm for calculating the CGD, this difference is rather insignificant, probably due to the very small stack dimensions required by the HFSM which was used in the recursive implementation. However, in the recursive implementations of the tree-based sorting and the knapsack algorithms, the dimensions of the HFSM-supporting stacks were considerably higher and thus the significant difference in the required FPGA resources. On the other hand, both the recursive and the iterative implementations of the set covering algorithm made use of stacks because both required a backtracking mechanism. For this reason, the number of required FPGA slices is the same for both versions.

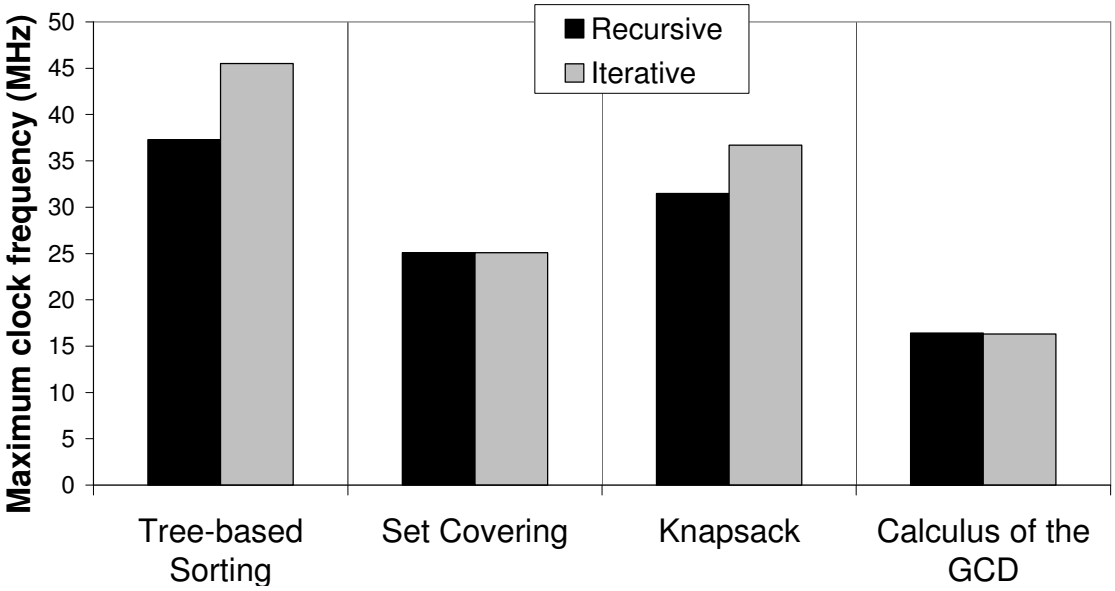


Figure 6.6 - Maximum clock frequency allowed on the Handel-C-based implementations

The graph in Figure 6.6 highlights the differences in the maximum clock frequency which guarantees correct circuit behavior for the various Handel-C-based experiments.

In maximum clock frequency, iterative projects have revealed:

- **i)** to be more advantageous (allowing higher frequencies) than recursive ones, when implementing non-backtracking tree-based algorithms;
- **ii)** virtually no difference from recursive projects, when implementing backtracking or cyclic algorithms.

The number of clock cycles which are required to solve the problem instances in each Handel-C-based experiment is set side by side in Figure 6.7 for comparison.

In number of clock cycles that are required to solve the problems, iterative and recursive Handel-C projects (unlike the VHDL projects) have revealed:

- **i)** to be equally advantageous, when implementing backtracking or cyclic algorithms;
- **ii)** no algorithm-independent differences, when implementing non-backtracking tree-based algorithms.

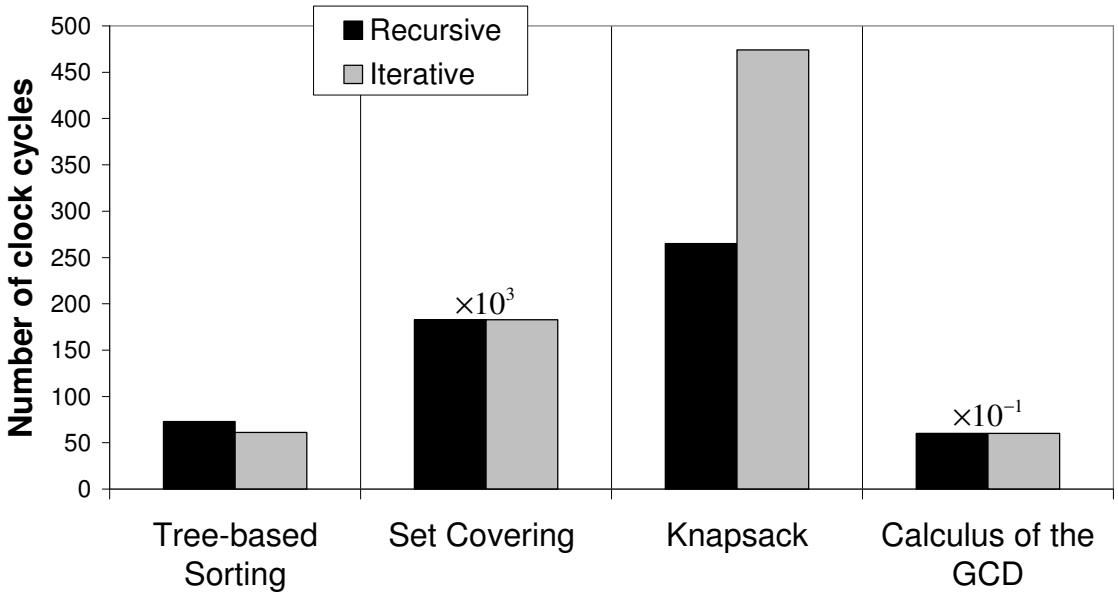


Figure 6.7 - Number of clock cycles used for solving the problem on the Handel-C-based implementations

Last, the graph in Figure 6.8 emphasizes the differences in time that is necessary to achieve a solution on the various Handel-C-based experiments.

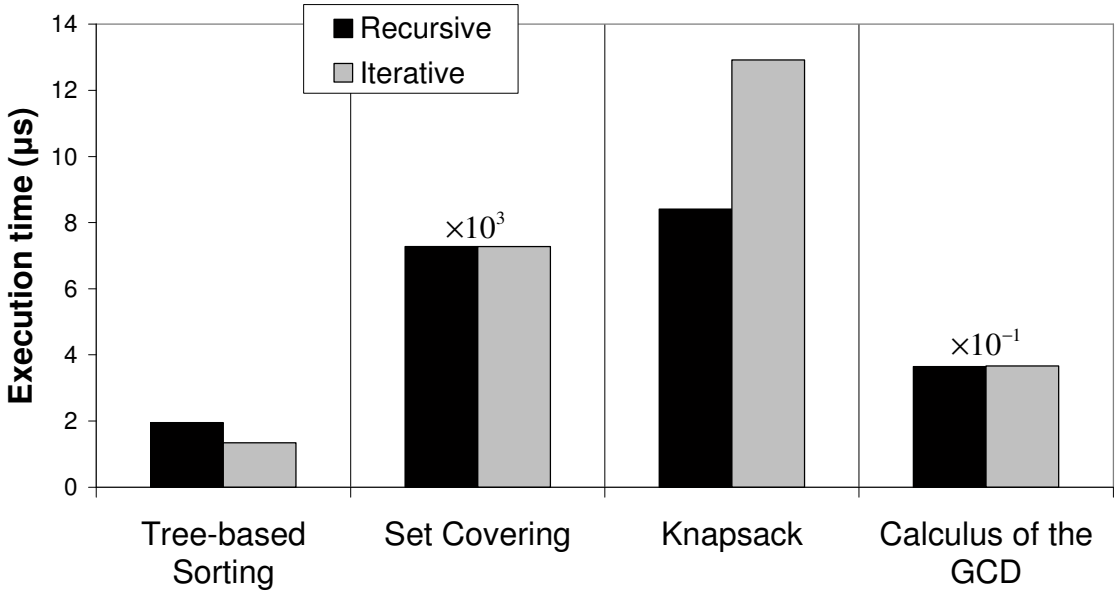


Figure 6.8 - Time required by the Handel-C-based implementations for solving the problem

The relative results regarding the time required for solving the problems, using the maximum attainable clock frequency for each implementation, are very similar to

those regarding the number of clock cycles spent on the same task. Thus, also in execution time, iterative and recursive Handel-C projects have revealed:

i) to be equally advantageous, when implementing backtracking or cyclic algorithms;

ii) no algorithm-independent differences, when implementing non-backtracking tree-based algorithms.

6.1.2.3. Summary and further discussion

Some of the general criteria which have been achieved by means of this experiment set can be summarized in Table 6.5.

Table 6.5 - Summary of general criteria achieved with this experiment set

<table>
<tr><th colspan="3">Algorithm scope</th><th>Number of FPGA slices</th><th>Maximum clock frequency</th><th>Number of clock cycles</th><th>Execution time</th></tr>
<tr><td rowspan="3">VHDL</td><td colspan="2">Cyclic</td><td rowspan="3">Less advantageous when using pure logic and modularity simultaneously</td><td colspan="3">No general criteria</td></tr>
<tr><td rowspan="2">Tree-based</td><td>Backtracking</td><td rowspan="2">No General criteria</td><td rowspan="2">Recursive implementations generally more advantageous</td><td rowspan="2">No general criteria</td></tr>
<tr><td>Non-backtracking</td></tr>
<tr><td rowspan="3">Handel-C</td><td colspan="2">Cyclic</td><td colspan="2" rowspan="2">No general criteria</td><td colspan="2" rowspan="2">Iterative and recursive equally advantageous</td></tr>
<tr><td rowspan="2">Tree-based</td><td>Backtracking</td></tr>
<tr><td>Non-backtracking</td><td colspan="2">More advantageous in iterative implementations</td><td colspan="2">No general criteria</td></tr>
</table>

Furthermore, from examining the complexity of the resulting circuits one concludes that the use of embedded (block or distributed) memory significantly reduces the amount of FPGA resources that are occupied by the solvers. This fact allows recursive algorithms to be nearly as resource-demanding as iterative ones. Each new FPGA family put on the market has usually more embedded memory than the previous ones. This memory can therefore be used to store more stack data virtually without increasing the number of occupied FPGA slices, and this allows much more complex algorithms to be implemented. The only additional resources that are required for

recursive algorithms are those keeping stack pointers. Thus, one can expect modular implementations to be significantly more advantageous for very complex problem instances. Moreover, a highly integrated stack memory for HFSMs might potentially be included as an embedded block in future generations of FPGAs [Sklyarov05]. Such feature would encourage the generalization of hierarchical implementations (recursive ones included) which, in turn, would encourage FPGA manufacturers to generalize that feature.

Besides those criteria which are obtained from quantifiable results, a more subjective deliberation can also lead to important conclusions. Examples of important project characteristics that are hard to quantify include design time, clarity of the algorithm's description (compare *e.g.* Figure 5.22 and Figure 5.23), ease of modification, etc. These characteristics can be improved through strategies such as divide and conquer (hierarchical specification), modularity, and design of reusable components. In fact, hierarchical modular specifications provide direct support for reusability, as a given module can be included multiple times in the same or even different algorithms. This possibility obviously leads to significant reductions in the design time and, in some cases, in the required hardware resources. Furthermore, if hierarchy is applied on the basis of an HFSM, recursive calls are inherently supported without any additional hardware, as previously mentioned. In such cases, the use of recursion or iteration should be assessed on the basis of algorithm clearness. For all these reasons, designers should keep in mind that HFSM-based hierarchical modular implementations possess strong design advantages, which are particularly important when developing complex projects.

6.2. Validation and analysis of the architecture for generic matrix-oriented solvers

For easy reference in this section, let us divide the architecture for generic matrix-oriented solvers in two components:

- The control component, composed of the reprogrammable control unit and the interface module, together with the software application;
- The operational component, which is composed of all the other functional blocks and implements the whole set of operations over the considered data.

The operational component and all interaction between its functional blocks have been validated in a software application which was modeled in the C# programming language. Each functional block was described by a class that emulates the behavior which is expected from its hardware implementation. When the application starts, objects of those classes are instantiated and they are reused to solve as many number of problem instances as required. Furthermore, each instance can correspond to any of the three matrix-based backtracking search algorithms described in section 5.1.2, *i.e.* those regarding set covering (DABM class solver), Boolean satisfiability (DATM class solver), and graph coloring (DATM class solver). A special class emulates the reprogrammable control unit behavior in order to validate the execution of different algorithms. Experiments were carried out using different sequences of problem instances and the application was able to correctly solve them.

After validation, the architecture's operational component was implemented and successfully tested using the Handel-C system-level specification language and the DETIUA-S3 prototyping board, which incorporates a Xilinx Spartan-3 FPGA (namely a XC3S400). A USB interface was used for data exchange between hardware and software.

The hardware reprogrammable control unit and the user agent module were designed using VHDL, whilst a software application to interact with the user agent was developed in C#. The expected run-time control unit reprogramming, for implementing different algorithms, was tested and successfully achieved.

Table 6.6 presents a summary of the data structures' usage in each implemented combinatorial search algorithm, permitting to assess the reusability of the structures amongst this type of solvers. For each of the three algorithms, Table 6.6 indicates whether each of the data structures is either explicitly declared by the designer, implicitly employed as a building block (for those which are explicitly declared), or not used at all. When a data structure is used both explicitly and implicitly, its usage is labeled as explicit.

Table 6.6 - Data structure usage in different matrix-based backtracking search algorithms

Data Structure	Set Covering	SAT	Graph Coloring
Binary Vector	Explicit	Explicit	Explicit
Ternary Vector	None	Explicit	Explicit
Mask	Explicit	Explicit	Explicit
Stack	Explicit	Explicit	Explicit
Simple Access Binary Matrix	Implicit	Implicit	Implicit
Dual Access Binary Matrix	Explicit	None	None
Simple Access Ternary Matrix	None	Implicit	Implicit
Dual Access Ternary Matrix	None	Explicit	Explicit

Table 6.6 reveals that the most basic data structures, such as binary vectors, masks, and simple access binary matrices, are explicitly or implicitly used across the three implemented combinatorial search algorithms. Stacks are also thoroughly employed because they are required for supporting backtracking, which is typical amongst this type of algorithms. On the other hand, more complex structures are less often used; as is the case of dual access binary matrices, which are exploited only in one of the implemented algorithms.

Table 6.7 indicates whether each of the different functional blocks is: explicitly employed by the designer; implicitly constructed as a building block (for those which are explicitly employed); or not used, in each of the analyzed combinatorial search algorithms. In the table, abbreviations BV, TV, SABM, DABM, SATM, and DATM respectively stand for binary vector, ternary vector, simple access binary matrix, dual access binary matrix, simple access ternary matrix, and dual access ternary matrix. When a functional block is used both explicitly and implicitly, its usage is labeled as explicit.

The use of operations for accessing matrix rows and columns, which can be binary or ternary vectors, is not considered in Table 6.7, as it can directly be deduced from the matrix type. It should be noticed that writing rows and columns is only needed for initialization of problem instances because the solving algorithms do not require changing matrix contents in order to find a solution.

Table 6.7 - Functional block usage in different matrix-based backtracking search algorithms

Method description	Input variant	Set Covering	SAT	Graph Coloring
Calculate the **number of zeros** (ones*) in a vector.	BV	None	None	None
	TV	None	None	None
	BV, Mask	Explicit	None	None
	TV, Mask	None	Explicit	None
Determine whether 2 vectors are **orthogonal**.	TV, TV	None	Implicit	Explicit
	TV, TV, Mask	None	Explicit	None
Calculate the **intersection** of 2 ternary vectors.	TV, TV	None	None	Explicit
Determine whether a ternary vector is constituted by **only *don't-care*** values (ones*, zeros*).	TV	None	Implicit	Implicit
	TV, Mask	None	Explicit	Explicit
Determine whether a ternary vector has **no zeros** (ones*)	TV	None	None	Implicit
	TV, Mask	None	None	Explicit
Create a mask which identifies the **position of *don't-care*** (non-*don't-care**) values in a ternary vector.	TV	None	None	None
	TV, Mask	None	None	None
Identify the **position of the first zero** (one*) in a vector/mask.	BV	Explicit	None	Explicit
	Mask	None	None	Explicit
	BV, Mask	Explicit	None	None
	TV, Mask	None	Explicit	None
Given a mask *m* and two vectors *a* and *b*, create a **new vector, copying values** from *a*, for positions that are masked by *m*; and from *b*, for positions that are not masked by *m*.	BV, BV, Mask	None	None	Implicit
	TV, TV, Mask	None	None	Explicit
Build the **transpose** of a matrix.	SABM	Implicit**	Implicit**	Implicit**
	DABM	Explicit**	None	None
	SATM	None	Implicit**	Implicit**
	DATM	None	Explicit**	Explicit**
Push (pop*) a vector/mask onto (from) the top of a stack.	Stack, BV	Explicit	Implicit	Explicit
	Stack, Mask	Explicit	Explicit	Explicit
	Stack, TV	None	Explicit	Explicit

* in another version of the method (typically used in combination in an algorithm)
** only for problem instance initialization purposes (not part of the solving algorithm)

Table 6.7 reveals that some of the developed functional blocks have not been used. Nevertheless, they can be helpful for implementing matrix-based combinatorial search algorithms which have not been addressed here.

6.3. Assessment of the developed prototyping tools and summary of potential applications

PBM has been tested in various ways. Furthermore, this tool is subject to an on-going continuous testing and improvement process. The software has been made available through the internet [Pimentel] and installed in classroom computers at the Department of Electronics, Telecommunications and Informatics of the University of Aveiro. Some of the disciplines taught at the department already use several DETIUA-S3 boards and PBM as working tools, and important feedback is accessible from students.

The fact that DETIUA-S3 and PBM have been used in practical classes reveals to some extent the reliability and the practical potential of this set of tools. Many projects of undergraduate students consisted of extension boards which implement different interfaces for peripherals, such as a keyboard, a mouse, and a VGA monitor.

Moving on to usability issues, some experiments have been carried out in order to determine the expected execution time for PBM's most basic operations with respect to the DETIUA-S3 board. Several measurements have been made through USB and Bluetooth interfaces, using a few different computers.

Table 6.8 - Average execution time in function of task and interface used

Task \ Interface	USB	Bluetooth
Erase a sector	0.7 sec	1 sec
Read an entire sector	0.4 sec	11 sec
Write an entire sector	1.5 sec	28 sec
Write a bitstream	5.5 sec	1 min 27 sec

Table 6.8 presents the average time by task and by interface which resulted from those measurements. When reading the table values, one should take into consideration the following:

a) Each flash memory sector has 64 KB;

b) Writing a bitstream involves 4 sectors;

c) Writing tasks require previous erasure of the targeted sectors, and this overhead is included in the time values presented.

As previously mentioned, PBM offers the possibility of sending multiple bitstreams to DETIUA-S3 and storing them in the third logical section of the flash memory (see Figure 4.3) in a format that is ready to be loaded onto an FPGA. This feature allows for the following potential applications:

1. Autonomous experiments with different single bitstream projects without connection to a host computer. In particular, this mode allows for the comparison and validation of alternative implementations. A simple additional switch, attached through expansion connectors, can be used to select the logic subsection that keeps the bitstream to be loaded to the FPGA.

2. FPGA run-time reconfiguration, permitting to implement circuits that require more resources than the resources available in the FPGA.

3. Programming FPGAs installed on additional extension boards. In this case, the core FPGA is considered to be a controller (manager) for a runtime reconfigurable system which includes multiple FPGAs.

The first two possibilities have already been tested and found successful.

6.4. Conclusion

In the beginning, a limited number of problems were examined and a comparison of relevant recursive and iterative algorithms was done. Synthesis was carried out on the basis of Handel-C and VHDL languages. Similar comparison for the same problems was done in software using C#. The obtained results were compared, revealing that a similar tendency was taking place for particular algorithms described at different levels. Analysis of different algorithms permitted to draw out algorithmic characteristics that allow potential benefits to be predicted, as a tendency always took place for the considered backtracking search algorithms. Thus, for many experiments, it was possible to avoid very complicated design and implementation steps required for synthesis of hardware and rely considerably on modeling in software. Finally, only some of the studied problems were implemented and tested in hardware and the

remaining problems were modeled just in software in order to validate the correctness of the respective algorithms.

In order to compare iterative and recursive hardware implementations, a set of experiments was carried out using the following four algorithms: sorting based on a binary tree, approximate algorithm for solving the set covering problem, exact algorithm for solving the knapsack problem, and calculus of the greatest common divisor between two integers. The first and third use context data stacks, whereas the other two do not.

The recursive and iterative versions of each of these four algorithms was carefully modeled and debugged in C#, and then specified both in VHDL and in Handel-C, with the exception of the approximate set covering algorithm which has not been specified in VHDL. Moreover, different versions of each of the VHDL-based implementations have been prepared in order to obtain some additional criteria regarding design modularity and embedded memory usage. Recursive solutions must always be modular, as recursion is achieved by means of an HFSM.

The results of the experiments indicate the following general criteria:

1. The number of FPGA slices occupied by VHDL implementations is higher if using pure logic and modularity simultaneously;

2. When designing solvers with non-backtracking tree-based algorithms in Handel-C, both the average number of occupied FPGA slices and the average maximum allowed clock frequency are more advantageous when implemented iteratively;

3. When designing solvers with tree-based algorithms in VHDL, the average number of clock cycles required to reach a solution is lower if implemented recursively;

4. When designing solvers with cyclic or backtracking tree-based algorithms in Handel-C, iterative and recursive implementations require the same average number of clock cycles and the same average time to reach a solution.

From examining the complexity of the resulting circuits one concludes that the use of embedded memory allows recursive algorithms to be nearly as resource-demanding as

iterative ones. Each new FPGA family put on the market has usually more embedded memory than the previous ones. This memory can therefore be used to store more stack data virtually without increasing the number of occupied FPGA slices, and this allows much more complex algorithms to be implemented. Thus, one can expect modular implementations to be significantly more advantageous for very complex problem instances. Moreover, a highly integrated stack memory for HFSMs might potentially be included as an embedded block in future generations of FPGAs, encouraging the generalization of hierarchical implementations (recursive ones included) which, in turn, would encourage FPGA manufacturers to generalize that feature.

Other important project characteristics, such as design time, clarity of the algorithm's description, and ease of modification, can be improved through strategies such as divide and conquer, modularity, and design of reusable components. HFSM-based hierarchical modular implementations possess strong design advantages, which are particularly important when developing complex projects. They provide direct support for reusability, significantly reducing the design time and, in some cases, the hardware resources required. Furthermore, if hierarchy is applied on the basis of an HFSM, recursive calls are inherently supported without any additional hardware.

The operational component of the architecture for generic matrix-oriented solvers, and all interaction between its functional blocks have been validated in a software application modeled in C#. Each functional block was described by a class that emulates the behavior which is expected from its hardware implementation. A special class emulates the reprogrammable control unit behavior in order to validate the execution of different algorithms. The application was able to correctly solve different sequences of problems (from set covering, Boolean satisfiability, and graph coloring) and problem instances, reusing its functional block-emulating objects instantiated only once. After validation, the architecture's operational component was implemented and successfully tested using Handel-C and DETIUA-S3. A USB interface was used for data exchange. Hardware reprogrammable control unit and user agent modules were designed using VHDL, whilst a software application to interact with the user agent was developed in C#. The expected run-time control unit reprogramming, for implementing different algorithms, was tested and successfully achieved.

Detailed analysis reveals that binary vectors, masks, simple access binary matrices and stacks are explicitly or implicitly used across the three implemented combinatorial search algorithms, whereas dual access binary matrices are used less often. Some of

the developed functional blocks have not been used but they can be helpful for implementing matrix-based combinatorial search algorithms which have not been addressed here.

Prototyping Board Manager has been tested in various ways and it is subject to an on-going continuous testing and improvement process. Some of the disciplines taught at the Department of Electronics, Telecommunications and Informatics of the University of Aveiro already use several DETIUA-S3 boards and PBM as working tools and important feedback is accessible from students. Many projects of undergraduate students consisted of extension boards which implement different interfaces for peripherals. We can therefore claim that this software has a significant practical usefulness.

The possibility of sending multiple bitstreams to DETIUA-S3 and storing them in the flash memory, ready to be loaded onto an FPGA allows for: autonomous experiments with different single bitstream projects without connection to a host computer; FPGA run-time reconfiguration (permitting to implement circuits that require more resources than the resources available in the FPGA); and programming FPGAs installed on additional extension boards. The first two possibilities have already been tested and found successful.

7. Conclusion

This chapter summarizes the author's contribution, lists the most important results and suggests future work in the considered area. When some results are described, the relevant references to the thesis chapters (where the proper contribution is presented) are done.

7.1. Contributions

Basic contributions of the thesis are provided within the following three areas:

1. Analysis of recursive and iterative implementations of computational algorithms in hardware.

2. Synthesis and FPGA-based prototyping of computationally intensive algorithms applying recursive and iterative techniques. Note that the analysis of previous results, obtained in software development, has allowed us to select an area (namely tree-based computations) where recursive algorithms might be better than the iterative ones. The thesis presents analysis and implementation of tree-based computations that are used in combinatorial search algorithms applied to binary and ternary matrices. In addition, some other applications (such as tree-based data sorting) are studied.

3. Software tools for an FPGA-based prototyping system with the primary objective of satisfying research-specific requirements. Such tools are convenient for implementing circuits, demonstrating its advantages and carry out various experiments with them. The hardware of the FPGA-based prototyping system has been designed

by Manuel Almeida [Almeida08], whereas all the necessary software and relevant experiments were made within this thesis.

The most important results within each area of the thesis indicated above are as follows.

Within the first area:

1.1. Analysis of recursive and iterative implementations of different algorithms and comparison of their advantages and disadvantages (chapter 2).

1.2. Review of known approaches and presenting the state of the art in the scope of hardware implementation of recursive algorithms (chapter 2).

1.3. Applying recursive and iterative techniques to the computationally intensive algorithms selected, namely set covering, Boolean satisfiability, graph coloring, and data sorting. Some simple computational algorithms (such as discovering the greatest common divisor of integers) have been presented for illustrative purposes, making it easier to demonstrate implementation and other necessary details (chapter 5).

1.4. Prototyping and experiments with the algorithms mentioned in the previous point. Results and conclusions allow to estimate which technique (recursive or iterative) is more likely to be the most advantageous for particular applications (chapter 6).

Within the second area:

2.1. Selection and analysis of computationally intensive algorithms for further design space exploration targeted to recursive and iterative implementations and comparison of the relevant characteristics (chapter 3).

2.2. Software modeling and analysis of recursive and iterative implementations for algorithms mentioned in point 2.1 (chapter 5).

2.3. Hardware implementation and analysis of recursive and iterative algorithms referenced in point 2.1 on the basis of Handel-C specifications and VHDL descriptions (chapter 6).

2.4. Conclusions from experiments. Given the importance of these conclusions, they will be separately emphasized after presenting the results in the scope of area 3 below.

Within the third area:

3.1. A set of software tools incorporated in PBM has been designed for FPGA-based prototyping system DETIUA-S3 [Almeida08] developed at the department of Electronics, Telecommunications and Informatics of Aveiro University. The tools include the necessary drivers as well as user-friendly interface for configuring and interacting with the system and for experimental purposes. Support for both wired and wireless interactions between a host computer and the system is provided. Contributions of point 3.1 are presented in chapter 4.

3.2. The tool set also provides support for co-simulation, enabling local and remote users to construct digital systems in such a way that they are partially implemented in FPGA and partially modeled in software of a user computer.

3.3. A set of tools that provide remote users with most of the PBM functionality through the Internet. The tool set also provides support for co-simulation, enabling local and remote users to construct digital systems in such a way that they are partially implemented in FPGA and partially modeled in software of a user computer. This work was not initially planned for the thesis. Since this work has not been finished yet (and the completion is not required by the thesis objectives), we consider it as a useful direction of future work.

3.4. A set of experiments that have been done with the aid of the proposed methods and software/hardware tools (chapter 4).

The results of experiments and analysis of alternative recursive and iterative specifications and the relevant circuits permit to present the following summary for the considered hardware implementations:

a) The number of FPGA slices occupied by VHDL implementations is higher if using pure logic and modularity simultaneously. This drawback can be imputed to the circuitry supporting the HFSM. Such overhead practically disappears when making use of block or distributed memory, to which most of this support circuitry is synthesized.

b) When designing solvers with tree-based algorithms in VHDL, the average number of clock cycles required to reach a solution is lower if implemented recursively. However, the corresponding execution time is not necessarily shorter.

c) When designing solvers with non-backtracking tree-based algorithms in Handel-C, both the average number of occupied FPGA slices and the average maximum allowed clock frequency are less advantageous when implemented recursively. Because these drawbacks emerge with the use of stacks, recursive and iterative versions of either cyclic or backtracking tree-based algorithms lead to equally advantageous results.

d) When designing solvers with cyclic or backtracking tree-based algorithms in Handel-C, iterative and recursive implementations require the same average number of clock cycles and the same average time to reach a solution.

e) The use of embedded (block or distributed) memory significantly reduces the amount of FPGA resources that are occupied by solvers. This memory can be used to store stack data virtually without occupied more FPGA slices, allowing recursive algorithms to be nearly as resource-demanding as iterative ones. Overall, one can expect modular implementations to be significantly more advantageous for very complex problem instances.

f) For many applications, additional circuit complexity is not as important as clearness of the algorithm. In particular, we can benefit

from using a divide-and-conquer strategy, which can be applied by means of hierarchical specification. Hierarchical modular specifications provide direct support for reusability, which leads to significant reductions in the design time and, in some cases, in the required hardware resources. If hierarchy is applied on the basis of an HFSM, recursive calls are inherently supported without any additional hardware. In such cases, the use of recursion or iteration should be assessed on the basis of algorithm clearness. For the majority of tree-based algorithms, recursion leads to clearer and more easily understandable specifications.

g) Analysis that was carried out in previous publications [Sklyarov05] and summarized in the thesis has shown that extra hardware complexity for recursive calls usually appears due to the implementation of stack memories (especially for allowing deep recursive calls). However, this memory is very regular and it can be constructed from FPGA embedded memory blocks. We found out that this technique significantly reduces the number of FPGA slices for such implementations. A highly integrated stack memory for HFSMs might potentially be implemented as an embedded block in future generations of FPGAs and this would attract additional attention to hierarchical and even recursive, implementations.

The results of the design and implementation of FPGA-based hardware accelerators can be reused in the following directions:

- The general architecture for hardware solvers described in section 5.2 is reusable and thus, can be selected for potential future algorithms which require the relevant search techniques.

- The use of dynamically reprogrammable HFSMs enables to change the algorithm which is executed, providing a base for the design and implementation of customizable hardware accelerators. This work is very promising and can be postponed for future (see the next section).

- Many object-oriented software classes which were developed, tested, and described in chapter 5 can be reused in future applications. They are a useful basis for developing algorithms such as those discussed in chapter 5.

- Many developed software/hardware tools have been successfully used in educational process at the Department of Electronics, Telecommunications and Informatics of Aveiro University for students of two specialties, namely Electronics and Telecommunications, and Computers and Telematics. These tools include the prototyping FPGA-based system, prototyping board manager (PBM), and hardware/software partition frameworks. We can mention various student publications, such as [Silva09], [Dias08], and [Silva08], which explicitly indicate the use of the developed tools, confirming their usefulness and successful utilization. Furthermore, reusable library modules which are developed by students for communication between DETIUA-S3 FPGA and peripheral devices are made available in a dedicated online repository [Sousa].

7.2. Future work

We believe that the following directions are important for future work:

- Developing automatic software tools for detecting recursive fragments in hardware description code and system-level specifications and generating the respective hierarchical finite state machines. The integration of this functionality in CAD tools could promote the use of recursion and consequently motivate reconfigurable hardware manufacturers to include highly integrated stack memories for hierarchical finite state machines as an embedded block in future generations of FPGAs.

- Exploring parallel architectures of hardware-accelerators on the basis of recently proposed parallel hierarchical finite state machines [Sklyarov08b].

- Examining the results of the thesis for other types of hardware accelerators implementing tree-based computations. This is interesting in two following aspects: additional validation of the thesis results; and exploring potential extensions of applicability for recursive algorithms in hardware implementations.

- Developing a full set of tools that enable the designed hardware/software systems (FPGA-based prototyping system, PBM, etc.) to be used remotely (see chapter 4). Many basic results have been obtained in this thesis but this work has not been finished yet.

References

[Abelson96] H. Abelson, G. J. Sussman, with J. Sussman, *Structure and Interpretation of Computer Programs*, Second edition, Cambridge, Massachusetts: The MIT Press, 1996. [Online]. Available: The SICP Web Site, http://mitpress.mit.edu/sicp. [Accessed January 6, 2009].

[Aimé07] M. Aimé, G. Gateau, T. Meynard, "Implementation of a Peak-Current-Control Algorithm Within a Field-Programmable Gate Array," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 1, pp. 406-418, February 2007.

[Almeida06] M. Almeida, B. Pimentel, V. Sklyarov, I. Skliarova, "Design Tools for Rapid Prototyping of Embedded Controllers," in *Proceedings of the Third International Conference on Autonomous Robots and Agents*, 2006, pp. 683-688.

[Almeida08] M. Almeida, "Métodos e Ferramentas para Reconfiguração de FPGAs Remotamente," M.Sc. thesis, University of Aveiro, Portugal, 2008. (In Portuguese)

[AMD] *Advanced Micro Devices, Inc*. [Online]. Available: http://www.amd.com. [Accessed February 26, 2009].

[Arsac82] J. Arsac, Y. Kodrato, "Some techniques for recursion removal from recursive functions," *Association of Computing Machinery Transactions on Programming Languages and Systems*, vol. 4, no. 2, pp. 295-322, April 1982.

[Astrachan03] O. Astrachan, "Bubble sort: an archaeological algorithmic analysis," in Proceedings of the Thirty-fourth SIGCSE Technical Symposium on Computer Science Education, 2003, pp. 1-5.

[Bäck95] T. Bäck, M. Schütz, S. Khuri, "A Comparative Study of a Penalty Function, a Repair Heuristic and Stochastic Operators with the Set-Covering Problem," in *Artificial Evolution*, vol. 1063 of Lecture Notes in Computer Science, Berlin / Heidelberg: Springer, 1995, pp. 320-332.

[Backus85] J. Backus, "From function level semantics to program transformation and optimization," in *Mathematical Foundations of Software Development*, vol. 185 of Lecture Notes in Computer Science, Berlin / Heidelberg: Springer, 1985, pp. 60-91.

[Ball60] W. Ball, *Mathematical Recreations and Essays*, Eleventh edition, revised by H. Coxeter, New York: Macmillan, 1960.

[Beier04] R. Beier, B. Vöcking, "Probabilistic Analysis of Knapsack Core Algorithms," in *Proceedings of the Fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004, pp. 468-477.

[Bitner75] J. Bitner, E. Reingold, "Backtrack Programming Techniques," *Communications of the Association of Computing Machinery,* vol. 18, no. 11, pp. 651-656, November 1975.

[Bodin91] L. Bodin, A. Kashani, "The zone hopping problem," *Computers and Operations Research*, vol. 18, no. 1, pp. 75-86, January 1991.

[Bondalapati98] K. Bondalapati, V. Prasanna, "Mapping loops onto reconfigurable architectures," in *Field-Programmable Logic: From FPGAs to Computing Paradigm*, vol. 1482 of Lecture Notes In Computer Science, London: Springer-Verlag, 1998, pp. 268–277.

[Bondalapati00] K. Bondalapati, V. Prasanna, "Loop pipelining and optimization for run time reconfiguration," in *Proceedings of the Fifteenth IPDPS 2000 Workshops on Parallel and Distributed Processing*, vol. 1800 of Lecture Notes In Computer Science, London: Springer-Verlag, 2000, pp. 906–915.

[Breuer70] M. Breuer, "Simplification of the covering problem with application to Boolean expressions," *Journal of the Association of Computing Machinery*, vol. 17, no. 1, pp. 166-181, January 1970.

[Cameron94] P. J. Cameron, *Combinatorics: Topics, Techniques, Algorithms*, Cambridge University Press, 1994.

[Carrano95] F. M. Carrano, *Data Abstraction and Problem Solving with C++*, Redwood City, California: The Benjamin / Cummings Publishing Company, Inc., 1995.

[Celoxica] *Celoxica: Low-latency and accelerated computing solutions for Capital Markets*. [Online]. Available: http://www.celoxica.com. [Accessed February 26, 2009].

[Chor88] B. Chor, R. L. Rivest, "Knapsack-type public key cryptosystem based on arithmetic in finite fields," *IEEE Transactions on Information Theory*, vol. 34, no. 5, pp. 901-909, September 1988.

[Cohen79] J. Cohen, "Non-deterministic algorithms," in *Association of Computing Machinery Computing Surveys*, vol. 11, no. 2, pp. 79-94, June 1979.

[Culberson] J. Culberson, *Graph Coloring Page*. [Online]. Available: http://www.cs.ualberta.ca/~joe/Coloring/index.html. [Accessed: Jan. 7, 2009].

[D&R06] "FPGA Market Will Reach $2.75 Billion by Decade's End," *Design And Reuse*, May 24, 2006. [Online]. Available: http://www.design-reuse.com/news/13441/fpga-market-reach-2-75-billion-decade-end.html. [Accessed: January 8, 2009].

[Dias08] N. Dias, S. Tafula, "Implementação em FPGA de um ordenador numérico recursivo com interface gráfica," *Electrónica e Telecomunicações*, vol. 4, no. 9, 2008, pp. 1006-1009.

[Du07] H. Du, H. Qi, X. Wang, "Comparative Study of VLSI Solutions to Independent Component Analysis," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 1, pp. 548-558, February 2007.

[EETimes02] "Xilinx FPGAs integrate PowerPC processor," *EE Times Asia*, March 8, 2002. [Online]. Available: http://www.eetasia.com/ART_8800212173_1034362_NP_14f8c950.HTM. [Accessed: January 8, 2009].

[EETimes06a] "Semiconductor FPGA/PLD market to grow 14% in '06, says Gartner," *EE Times Asia*, May 30, 2006. [Online]. Available: http://www.eetasia.com/ART_8800419580_499485_NT_07a6c7d3.HTM. [Accessed: January 13, 2009].

[EETimes06b] R. Goering, "FPGA users rank challenges, tasks," *EE Times*, July 31, 2006. [Online]. Available: http://www.eetimes.com (search key: 191600017). [Accessed: January 8, 2009].

[EETimes06c] D. McGrath, "FPGAs top choice for some telecom equipment, survey says," *Embedded.com*, October 8, 2006. [Online]. Available: http://www.eetimes.com (search key: 191901726). [Accessed: January 8, 2009].

[EFF] EFF, *Data-Compression*. [Online]. Available: http://www.data-compression.com. [Accessed: January 8, 2009].

[Erickson96] M. J. Erickson, *Introduction to combinatorics*, New York: Wiley-Interscience, 1996.

[Estrin60] G. Estrin, "Organization of Computer Systems – The Fixed Plus Variable Structure Computer," in *Proceedings of the Western Joint Computer Conference*, 1960, pp. 33-40.

[Ezick] J. Ezick, *Robotics*. [Online]. Available: http://dimacs.rutgers.edu/REU/1996/ezick.html. [Accessed: January 13, 2009].

[Ferizis06] G. Ferizis, H. El Gindy, "Mapping recursive functions to reconfigurable hardware," in *Proceedings of the Field Programmable Logic and Applications International Conference*, 2006, pp. 1-6.

[Floyd67] R. W. Floyd, "Nondeterministic algorithms," *Journal of the Association of Computing Machinery*, vol. 14, no. 4, pp. 636-644, October 1967.

[Gavious94] A. Gavious, Z. Rosberg, "A restricted complete sharing policy for a stochastic knapsack problem in b-isdn," *IEEE Transactions on Communications*, vol. 42, no. 7, pp. 2375-2379, July 1994.

[Gilmore61] R. Gilmore, R. E. Gomory, "A linear programming approach to cutting-stock problem," *Operations Research*, vol. 9, no.6, pp. 863-888, November-December 1961.

[Gleeson94] M. Gleeson, J. Ottensman, "A decision support system for acquisitions budgeting in public libraries," *Interfaces*, vol. 24, no. 5, pp. 107-117, September-October 1994.

[Golomb65] S. Golomb, L. Baumert, "Backtrack programming," *Journal of the Association of Computing Machinery*, vol. 12, no. 4, pp. 516-524, October 1965.

[Goossens97] G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, P. G. Paulin, "Embedded software in real-time signal processing systems: design technologies," in *Proceedings of the IEEE*, vol. 85, no. 3, pp. 436-454, March 1997.

[Gu97] J. Gu, P. W. Purdom, J. Franco, B. W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 35, pp. 19-151, 1997.

[Gu04] J. Gu, *Constraint-Based Search*, New York: Cambridge University Press, 1994.

[Hahn68] S. G. Hahn, "On the optimal cutting of defective sheets," *Operations Research*, vol. 16, no.6, pp. 1100-1114, November-December 1968.

[Harrison92] P. G. Harrison, H. Khoshnevisan, "A new approach to recursion removal," *Theoretical Computer Science*, vol. 93, no. 1, pp. 91-113, February 1992.

[Helsgaun95] K. Helsgaun, "CBack: a simple tool for backtrack programming in C," *Software - Practice & Experience*, vol. 32, no. 8, pp. 905-934, August 1995.

[Henig90] M. Henig, "Risk criteria in a stochastic knapsack problem," *Operations Research*, vol. 38, no. 5, pp. 820-825, September-October 1990.

[Jan93] J. Jan, S. Wang, "A dynamic access control scheme based upon the knapsack problem," *International Journal of Computers & Mathematics with Applications*, vol. 26, no. 12, pp. 75-86, 1993.

[Jung07] S. Jung, S. su Kim, "Hardware Implementation of a Real-Time Neural Network Controller With a DSP and an FPGA for Nonlinear Systems," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 1, pp. 265-271, February 2007.

[Jutman07] A. Jutman, A. Tsertov, A. Tsepurov, I. Aleksejev, R. Ubar, H. Wuttke, "BIST Analyzer: a Training Platform for SoC Testing," in *Proceedings of Frontiers in Education Conference*, 2007, pp. S3H-8 - S3H-13.

[Kernighan88] B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

[Kfoury97] A. J. Kfoury, "Recursion versus iteration at higher-orders," in *Foundations of Software Technology and Theoretical Computer Science*, vol. 1346 of Lecture Notes in Computer Science, Berlin: Springer-Verlag, 1997, pp. 57-73.

[Knuth97] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, Third edition, Vol. 2, Addison Wesley, 1997.

[Knuth98] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Second edition, Vol. 3, Addison Wesley, 1998.

[Kovacec05] D. Kovacec, "A Multimedia Platform for Automotive and Consumer Markets," *Xcell Journal* [Online]. Available: http://china.xilinx.com. [Accessed: January 12, 2009].

[Kreher99] D. L. Kreher, D. R. Stinson, Combinatorial algorithms: generation, enumeration, and search, Florida: CRC Press, 1999.

[Kruse87] R. L. Kruse, *Data Structures and Program Design*, Second edition, Prentice-Hall, 1987.

[Liu99] Y. A. Liu, S. D. Stoller, "From recursion to iteration: what are the optimizations?," in *Proceedings of the ACM SIGPLAN Notices*, vol. 34, no. 11, pp. 73-82, November 1999.

[MacMillen00] D. MacMillen, M. Butts, R. Camposano, D. Hill, T. W. Williams, "An Industrial View of Electronic Design Automation," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1428-1448, December 2000.

[Madsen79] O. B. G. Madsen, "Glass cutting in a small firm," *Mathematical Programming*, vol. 17, no. 1, pp. 85-90, December 1979.

[Marques-Silva08] J. Marques-Silva, "Practical Applications of Boolean Satisfiability," in *Proceedings of the Ninth International Workshop on Discrete Event Systems*, 2008, pp. 74-80.

[Maruyama99] T. Maruyama, M. Takagi, T. Hoshino, "Hardware Implementation Techniques for Recursive Calls and Loops," in *Field Programmable Logic and Applications*, vol. 1673 of Lecture Notes in Computer Science, Berlin / Heidelberg: Springer, 1999, pp. 450-455.

[Maruyama00] T. Maruyama, T. Hoshino, "A C to HDL Compiler for Pipeline Processing on FPGAs," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 101-110.

[MentorGraphics] Mentor Graphics, *ModelSim – a comprehensive simulation and debug environment for complex ASIC and FPGA designs*. [Online]. Available: http://www.model.com/. [Accessed: February 24, 2009].

[Merkle78] R. Merkle, M. Hellman, "Hiding information and signatures in trapdoor knapsack," IEEE Transactions on Information Theory, vol. 24, no. 5, pp. 525-530, September 1978.

[Micheli94] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, USA: McGraw-Hill, Inc., 1994.

[Moore65] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114-117, April 1965.

[Newswire05] PR Newswire, "Xilinx Delivers Highest Performance DSP Solutions for Multimedia, Video and Imaging Applications", in *FPGA and Structured ASIC Journal*, October 24, 2005. [Online]. Available: http://www.fpgajournal.com (search key: 20051024_01). [Accessed: January 13, 2009].

[Ninos08] S. Ninos, A. Dollas. "Modeling recursion data structures for FPGA-based implementation," in *Proceedings of the Eighteenth International Conference on Field-Programmable Logic and its Applications*, 2008, pp. 11-16.

[Noble03] J. V. Noble. "Recurses!," *Computing in Science & Engineering*, vol. 5, no. 3, pp. 76- 81, May-June 2003.

[Nunez03] J. L. Nunez, S. Jones, "Gbit/s Lossless Data Compression Hardware," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 11, no. 3, pp. 499-510, June 2003.

[Panainte04] E. M. Panainte, K. Bertels, S. Vassiliadis, "Multimedia Reconfigurable Hardware Design Space Exploration," in *Proceedings of the Sixteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, 2004, pp. 398-403.

[Partsch90] H. A. Partsch, Specification and Transformation of Programs - A Formal Approach to Software Development, Berlin: Springer-Verlag, 1990.

[Pimentel] B. Pimentel, *Homepage of Bruno Figueiredo Pimentel*. [Online]. Available: https://sites.google.com/site/brunofigueiredopimentel/. [Accessed February 26, 2009].

[Pimentel07] B. Pimentel, "A Dynamically Reprogrammable CSA-Generic Platform Architecture," in *Proceedings of the Fourth FPGAworld Conference*, 2007, pp. 16-21.

[Pimentel08] B. Pimentel, V. Sklyarov, M. Almeida, "Virtual and Remote Laboratories for Circuit Design e-Learning," in *Proceedings of the International Conference on Interactive Computer Aided Learning*, 2008.

[Pimentel09] B. Pimentel, "Recursion in Hardware: Applicability and Implementation Strategies," in *Proceedings of the Second International Conference on Advances in Circuits, Electronics and Micro-electronics*, 2009.

[Raik07] J. Raik, R. Ubar, A. Krivenko, M. Kruus, "Hierarchical Identification of Untestable Faults in Sequential Circuits," in *Proceedings of Tenth IEEE EUROMICRO Conference on Digital System Design*, 2007, pp. 668-671.

[Roadmap05] "International Technology Roadmap for Semiconductors. ITRS 2005 Edition: Design," 2005. Available: http://www.itrs.net/reports.html.

[Roadmap07] "International Technology Roadmap for Semiconductors. ITRS 2007 Edition: Design," 2007. Available: http://www.itrs.net/reports.html.

[Rodin90] E. Y. Rodin, D. Geist, "Flight and fire control with logic programming," *Computers & Mathematics with Applications*, vol. 20, no. 9/10, pp. 15-27, 1990.

[Rosen00] K. H. Rosen, J. G. Michaels, J. L. Gross, J. W. Grossman, D. R. Shier, *Handbook of Discrete and Combinatorial Mathematics*, CRC Press, 2000.

[Ross89] K. W. Ross, D. Tsang, "The stochastic knapsack problem," *IEEE Transactions on Communications*, vol. 37, no. 7, pp. 740-747, July 1989.

[Rubin73] J. Rubin, "A technique for the solution of massive set-covering problems with applications to airline crew scheduling," *Transportation Science*, vol. 7, pp. 34-48, 1973.

[Salcic06] Z. Salcic, J. Cao, S. K. Nguang, "A Floating-Point FPGA-Based Self-Tuning Regulator," *IEEE Transactions on Industrial Electronics*, vol. 53, no. 2, pp. 693-704, April 2006.

[Seth87] A. Seth, "Wastage reduction in wood cutting," *Opsearch*, vol. 24, no. 2, pp. 94-105, 1987.

[Shirazi98] N. Shirazi, W. Luk, P. Y. K. Cheung, "Run-time management of dynamically reconfigurable designs," in *Proceedings of the Eighth International Workshop on Programmable Logic and Applications*, 1998, pp. 59-68.

[Silva08] B. Silva, "Especificação, síntese e implementação em VHDL de um processador MIPS Single Cycle simplificado," *Electrónica e Telecomunicações*, vol. 4, no. 9, 2008, pp. 998-1005.

[Silva09] B. Silva, "Um Processador com Arquitectura MIPS para ensino," M.Sc. thesis, University of Aveiro, Portugal, 2009. (In Portuguese).

[Skliarova01] I. Skliarova, A. B. Ferrari, "Synthesis of reprogrammable control unit for combinatorial processor," in *Proceedings of the Fourth IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2001, pp. 179-186.

[Skliarova03] I. Skliarova, A. B. Ferrari, "The Design and Implementation of a Reconfigurable Processor for Problems of Combinatorial Computation," *Journal of Systems Architecture: the EUROMICRO Journal*, vol. 49, no. 4-6, 2003, pp. 211-226.

[Skliarova04a] I. Skliarova, "Reconfigurable Architectures for Problems of Combinatorial Optimization," Ph.D. dissertation, University of Aveiro, Portugal, 2004.

[Skliarova04b] I. Skliarova, A. B. Ferrari, "Reconfigurable Hardware SAT Solvers: A Survey of Systems," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1449-1461, November 2004.

[Skliarova05] I. Skliarova, "Implementation of Recursive Search Algorithms in Reconfigurable Hardware," in *Proceedings of the Fourth Winter International Symposium on Information and Communication Technologies*, 2005, pp. 142-147.

[Skliarova06a] I. Skliarova, "Intelligent Systems Engineering with Reconfigurable Computing," in *International Federation for Information Processing: Professional Practice in Artificial Intelligence*, vol. 218, Boston: Springer, 2006, pp. 161-170.

[Skliarova06b] I. Skliarova, V. Sklyarov, "Design Methods for FPGA-based implementation of combinatorial Search Algorithms," in *Proceedings of the International Workshop on SoC and MCSoC Design*, Fourth International Conference on Advances in Mobile Computing and Multimedia, 2006, pp. 359-368.

[Skliarova08] I. Skliarova, V. Sklyarov, "Recursive versus Iterative Algorithms for Solving Combinatorial Search Problems in Hardware," in *Proceedings of the Twenty-First International Conference on VLSI Design*, 2008, pp. 255-260.

[Sklyarov84] V. Sklyarov, *Synthesis of Finite State Machines Based on Matrix LSI*, Minsk, Belarus: Science & Techniques, 1984. (In Russian)

[Sklyarov98] V. Sklyarov, A. da Rocha, A. B. Ferrari, "Synthesis of Reconfigurable Control Devices Based on Object-oriented Specifications," in *Advanced Techniques for Embedded Systems Design and Test*, Kluwer Academic Publishers Group, 1998, pp 151-177.

[Sklyarov99] V. Sklyarov, "Hierarchical Finite-State Machines and Their Use for Digital Control," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 7, no. 2, pp. 222-228, June 1999.

[Sklyarov00] V. Sklyarov, "Synthesis of Control Circuits with Dynamically Modifiable Behavior on the Basis of Statically Reconfigurable FPGAs," in *Proceedings of the Thirteenth Symposium on Integrated Circuits and Systems Design*, 2000, pp. 353-358.

[Sklyarov02a] V. Sklyarov, "Reconfigurable models of finite state machines and their implementation in FPGAs," *Journal of Systems Architecture*, vol. 47, no. 14-15, pp. 1043-1064, August 2002.

[Sklyarov02b] V.Sklyarov, "Hardware/Software Modeling of FPGA-based Systems," in *Parallel Algorithms and Applications* (ISSN 1063-7192), vol. 17, no. 1, 2002, pp. 19-39.

[Sklyarov03a] V. Sklyarov, I. Skliarova, "Design of Digital Circuits on the Basis of Hardware Templates", in *Proceedings of the International Conference on Embedded Systems and Applications*, 2003, pp. 56-62.

[Sklyarov03b] V. Sklyarov, I. Skliarova, "Architecture of a Reconfigurable Processor for Implementing Search Algorithms over Discrete Matrices," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2003, pp. 127-133.

[Sklyarov03c] V. Sklyarov, I. Skliarova, A. Oliveira, A. B. Ferrari, "A Dynamically Reconfigurable Accelerator for Operations over Boolean and Ternary Vectors," in *Proceedings of the EUROMICRO Symposium on Digital System Design*, 2003, pp. 222-229.

[Sklyarov04] V. Sklyarov, "FPGA-based implementation of recursive algorithms," *Microprocessors and Microsystems*, vol. 28, no. 5-6, pp. 197-211.

[Sklyarov05] V. Sklyarov, I. Skliarova, B. Pimentel, "FPGA-based Implementation and Comparison of Recursive and Iterative Algorithms," in *Proceedings of the Fifteenth International Conference on Field-Programmable Logic and its Applications*, 2005, pp. 235-240.

[Sklyarov06a] V. Sklyarov, I. Skliarova, "Recursive and Iterative Algorithms for N-ary Search Problems," in *Professional Practice in Artificial Intelligence*, vol. 218 of IFIP International Federation for Information Processing, Boston: Springer, 2006, pp. 81-90.

[Sklyarov06b] V. Sklyarov, I. Skliarova, B. Pimentel, "Modeling and FPGA-based implementation of graph coloring algorithms," in *Proceedings of the Third International Conference on Autonomous Robots and Agents*, 2006, pp. 443-448.

[Sklyarov06c] V. Sklyarov, I. Skliarova, "Reconfigurable Hierarchical Finite State Machines," in *Proceedings of the Third International Conference on Autonomous Robots and Agents*, 2006, pp. 599-604.

[Sklyarov06d] V. Sklyarov, I. Skliarova, B. Pimentel, "Synthesis of FSMs on the Basis of Reusable Hardware Templates," *WSEAS Transactions on Systems*, vol. 5, no. 11, pp. 2548-2553, November 2006.

[Sklyarov07a] V. Sklyarov, I. Skliarova, B. Pimentel, "FPGA-based Implementation of Graph Colouring Algorithms," in *Studies in Computational Intelligence: Autonomous Robots and Agents*, vol. 76, Berlin / Heidelberg: Springer-Verlag, 2007, pp. 225–231.

[Sklyarov07b] V. Sklyarov, I. Skliarova, "Reuse Technique in Hardware Design," in *Proceedings of the IEEE International Conference on Information Reuse and Integration*, 2007, pp. 36-41.

[Sklyarov08a] V. Sklyarov, I. Skliarova, B. Pimentel, M. Almeida, "Multimedia Tools and Architectures for Hardware/Software Co-Simulation of Reconfigurable Systems," in *Proceedings of the Twenty-First International Conference on VLSI Design*, 2008, pp. 85-90.

[Sklyarov08b] V. Sklyarov, I. Skliarova, "Design and Implementation of Parallel Hierarchical Finite State Machines", in *Proceedings of the Second International Conference on Communications and Electronics*, 2008, pp. 33-38.

[Sousa] R. Sousa, *FPGA Design Repository*. [Online]. Available: http://sweet.ua.pt/~a16360/. [Accessed February 26, 2009].

[Sridharanand05] K. Sridharanand, T. K. Priya, "The Design of a Hardware Accelerator for Real-Time Complete Visibility Graph Construction and Efficient FPGA Implementation," *IEEE Transactions on Industrial Electronics*, vol. 52, no. 4, pp. 1185-1187, August 2005.

[Steiger04] C. Steiger, H. Walder, M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393-1407, November 2004.

[Subramonian04] V. Subramonian, H. M. Huang, G. Xing, C. Gill, C. Lu, R. Cytron, "Middleware specialization for memory-constrained networked embedded systems," in *Proceedings of the Tenth IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004, pp. 306-313.

[Tang06] P. Tang, "Complete inlining of recursive calls: beyond tail-recursion elimination," in *Proceedings of the Forty-fourth Annual Southeast Regional Conference*, 2006, pp. 579-584.

[Trenz] Trenz Electronic. *Products*. [Online]. Available: http://www.trenz-electronic.de. [Accessed February 26, 2009].

[Turley05] J. Turley, "Survey: Who uses custom chips," *Embedded Systems Design*, January 8, 2005. [Online]. Available: http://www.embedded.com (search key: 166404172). [Accessed: January 13, 2009].

[Ubar07] R. Ubar, S. Kostin, J. Raik, T. Evartson, H. Lensen, "Fault Diagnosis in Integrated Circuits with BIST," in *Proceedings of Tenth IEEE EUROMICRO Conference on Digital System Design*, 2007, pp. 604-610.

[Ubar08] R. Ubar, S. Devadze, M. Jenihhin, J. Raik, G. Jervan, P. Ellervee, "Hierarchical Calculation of Malicious Faults for Evaluating the Fault-Tolerance," in *Proceedings of the Fourth IEEE International Symposium on Electronic Design, Test & Applications*, 2008, pp. 23-25, 2008.

[Walker74] W. Walker, "Using the set-covering problem to assign Fire companies to Fire houses," *Operations Research*, vol. 22, no. 2, pp. 275-277, March-April 1974.

[Weinhardt99] M. Weinhardt, W. Luk, "Pipeline vectorization for reconfigurable systems," in *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999, p. 52-62.

[Wirth86] N. Wirth, *Algorithms and Data Structures*, Prentice-Hall, 1986.

[Wu93] Y. L. Wu, M. Marek-Sadowska, "Graph based analysis of FPGA routing," in *Proceedings of the European Design Automation Conference*, 1993, pp. 104-109.

[Xilinx] Xilinx, Inc. *Products and services*. [Online]. Available: http://www.xilinx.com. [Accessed February 26, 2009].

[Xilinx06] Virtex-5 LX Platform Overview. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf. [Accessed: January 13, 2009].

[Xilinx09] Virtex-6 and Spartan-6 FPGA Families Brochure. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/Virtex6_Spartan6_Product_Brief.pdf. [Accessed: February 24, 2009].

[Zakrevskij71] A. Zakrevskij, *Algorithms of Discrete Automata Synthesis*, Moscow: Nauka, 1971. (In Russian)

[Zakrevskij81] A. Zakrevskij, *Logical Synthesis of Cascade Networks*, Moscow: Nauka, 1981. (In Russian)

[Zakrevskij00] A. Zakrevskij, "Graph Coloring and Decomposition of Boolean Functions," *Logical Design*, no. 5, 2000. (In Russian)

[Zakrevskij08] A. Zakrevskij, Yu. Pottosin, L. Cheremisinova, *Combinatorial Algorithms of Discrete Mathematics*, Tallinn, Estonia: Tallinn University Press, 2008.

[Zhang89] Y. Zhang, R. Karp, "Parallel algorithms for combinatorial search problems," Ph.D. dissertation, University of California, Berkeley, California, 1989.

[Zhong99] P. Zhong, "Using Configurable Computing to Accelerate Boolean Satisfiability," Ph.D. dissertation, Princeton University, New Jersey, 1999.

[Zhuang07] H. Zhuang, K. S. Low, W. Y. Yau, "A Pulsed Neural Network With On-Chip Learning and Its Practical Applications," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 1, pp. 34-42, February 2007.