Design Synthesis for Dynamically Reconfigurable Logic Systems

Milan Vasilko

A thesis submitted in partial fulfilment of the requirements of Bournemouth University for the degree of Doctor of Philosophy

October 2000

Bournemouth University

Design Synthesis for Dynamically Reconfigurable Logic Systems

Abstract

Design Synthesis for Dynamically Reconfigurable Logic Systems

Milan Vasilko
Bournemouth University

Dynamic reconfiguration of logic circuits has been a research problem for over four decades. While applications using logic reconfiguration in practical scenarios have been demonstrated, the design of these systems has proved to be a difficult process demanding the skills of an experienced reconfigurable logic design expert.

This thesis proposes an automatic synthesis method which relieves designers of some of the difficulties associated with designing partially dynamically reconfigurable systems. A new design abstraction model for reconfigurable systems is proposed in order to support design exploration using the presented method. Given an input behavioural model, a technology server and a set of design constraints, the method will generate a reconfigurable design solution in the form of a 3D floorplan and a configuration schedule. The approach makes use of genetic algorithms. It facilitates global optimisation to accommodate multiple design objectives common in reconfigurable system design, while making realistic estimates of configuration overheads and of the potential for resource sharing between configurations. A set of custom evolutionary operators has been developed to cope with a multiple-objective search space.

Furthermore, the application of a simulation technique verifying the

results of such an automatic exploration is outlined in the thesis.

The qualities of the proposed method are evaluated using a set of benchmark designs taking data from a real reconfigurable logic technology. Finally, some extensions to the proposed method and possible research directions are discussed.

To Maria, Dominika, Viktoria

and my parents,

for their love, patience and support.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ASIC	Application Specific Integrated Circuit
CAD	Computer Aided Design
CDFG	Control/Data Flow Graph
CDS	Configuration and Data Store
CM	Clock Morphing
CMOS	Complementary Metal Oxide Silicon
DFG	Data Flow Graph
DRL	Dynamically Reconfigurable Logic
FPGA	Field Programmable Gate Array
FPL	Field Programmable Logic
FSM	Finite-State Machine
FSMD	Finite-State Machine Datapath
GA	Genetic Algorithm
HDL	Hardware Description Language
ILP	Integer Linear Program(ming)

OTP	One-Time Programmable
RCU	Reconfiguration Control Unit
RL	Reconfigurable Logic
RLU	Reconfigurable Logic Unit
RS	Reconfigurable System(s)
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SLU	Swappable Logic Unit
SRAM	Static Random Access Memory
VLSI	Very Large Scale Integration
VHDL	VHSIC HDL
VHSIC	Very High Speed Integrated Circuit
WCS	Writeable Control Store

Acknowledgements

This thesis would have not existed without the efforts of many kind individuals. It gives me a great pleasure to acknowledge their contribution here.

I would like to thank my first supervisor, Graham Benyon-Tinker, for his advice and encouragement, generous support, and dedication to see this project through to its successful end. I am very grateful to my second supervisor, David Long, for his help, constructive criticism and support on this and other related projects. Special thanks to Jim Roach and the rest of the management team of the School of Design, Engineering & Computing who have provided generous support throughout my studies.

My special gratitude goes to Djamel Ait-Boudaoud who was the first to suggest a research topic in the field of FPGAs, and later offered his support and advice as my first supervisor in the early years of my PhD studies. Professor Sa'ad Medhat also provided encouragement and support throughout this period.

I am grateful to my past and present colleagues, Steve Holloway, Petr Voleš, Radovan Čemeš, David Cabanis, Darrell Gibson for their friendship, help and an excellent working environment they have provided over the years.

Many individuals from the community of scientists, engineers and in-

dustrialists have helped to provide suggestions, guidance and shape the ideas presented in this thesis. While I cannot name all here, I would like to express my gratitude to Patrick Lysaght from University of Stathclyde and Wayne Luk from Imperial College for their encouragement and discussions on the topics of dynamically reconfigurable logic from the very beginning. I am also indebted to Patrick Schaumont and Serge Vernalde at IMEC, for their views and suggestions on modelling aspects of reconfigurable systems.

I am grateful to Tom Kean, John Gray, Jason Feinsmith and Patrick Kane and the companies they have been representing, namely Xilinx Development Corporation and Xilinx, Inc., for their advice, support and donations, which made work on aspects of this research possible.

Financial support provided during my PhD studies by the UK CVCP Overseas Research Award Scheme, the NATO ASI grant and the EC LSF programme grant is gratefully acknowledged.

My work on this research would have never been possible without the support from my family. I am grateful to my parents for their continuing support and encouragement in my pursuit of the university studies and the scientific career. Above all, I am indebted to my wife Maria and daughters Dominika and Viktoria, who have firmly supported me throughout my PhD studies and have patiently suffered all the consequences.

Chapter 1

Introduction

Techniques for the design of computing systems have been attracting interest since the invention of the first computing machines. Early design techniques have relied on the engineering excellence of computing pioneers and offered little assistance with the laborious procedures involved in the construction of even the simplest system components.

Advances in microelectronic technologies and system integration in the second half of the 20th century have moved computing machines from specialised research laboratories into everyday-life products. Many thousands of designers are developing embedded computing products today in an environment very different from that of fifty years ago.

Short product life cycles and increased competition are forcing design teams to minimise the 'time-to-market' of their products. While only a few years ago it was common for the development of an embedded computer product to take 1-2 years, the current market situation for many products is forcing design teams to deliver increasingly complex and powerful systems within only several months.

In this market environment, product design techniques and method-

ologies are of paramount importance. Computer-aided design tools and methodologies have helped to automate the laborious and repetitive design tasks, while increasingly higher levels of abstraction are being used to cope with system complexity. 'Push-button' design methodologies are being used to generate large portions of designed systems automatically, thus dramatically reducing the design time. Short development cycles demand 'first-time right' design methodologies which can guarantee the correctness of the automatically generated designs and minimise or completely eliminate design iterations. Objectives such as design time and performance now often dominate product development, rather than the previously desired silicon area efficiency.

Furthermore, many computing systems today are subjected to modifications during their product life cycles. These modifications can be caused by changes in protocol or interface standards, changes in user requirements, correction of product faults, and others. The period of such changes may vary from years to months, when products need to be upgraded for example with a new version of an embedded operating system. Changes may be also required in matters of seconds or milliseconds, when systems have to react to requirements changing in real-time. For example, where the type of data to be processed depends on system locality, or coding protocol depends on the type of processed data.

This flexibility has traditionally been provided via modifications of system software components—programs stored in re-writable memories. Such programs can be modified or completely replaced without the need to physically replace the memory devices. However, this 'software' approach to design flexibility is limited to processor-based components. While these can achieve high performance, timing-critical system components often

have to be implemented using custom integrated circuits.

With the commercial availability of reconfigurable logic (RL) devices, a similar level of flexibility can be provided in custom high-performance circuits.

The following sections outline some of the benefits and uses of reconfigurable logic technology and illustrate further the motivation for the presented work. This chapter concludes with a brief summary of the topics covered in the thesis.

1.1 Reconfigurable Logic

With *in-field programmable* or *configurable* logic technology it is possible to construct logic circuits in-field, after the technology devices have been manufactured.

A typical configurable logic architecture is composed of *logic blocks* surrounded by the *routing wires* (Fig. 1.1). Boolean functions and storage elements implemented in the individual logic blocks, together with the connectivity of reconfigurable routing elements, can be configured via a set of configuration switches. The on/off state of these switches is controlled by individual memory cells contained within a device *configuration memory*. The collection of states stored within the configuration memory is called a (hardware or device) *configuration*. It is the type and organisation of the configuration memory cells which determine the level of flexibility available in these technologies.

For example, one-time programmable (OTP) technologies store hardware configurations in distributed programmable read-only memories. Once a selected configuration is transferred into an OTP device, the con-

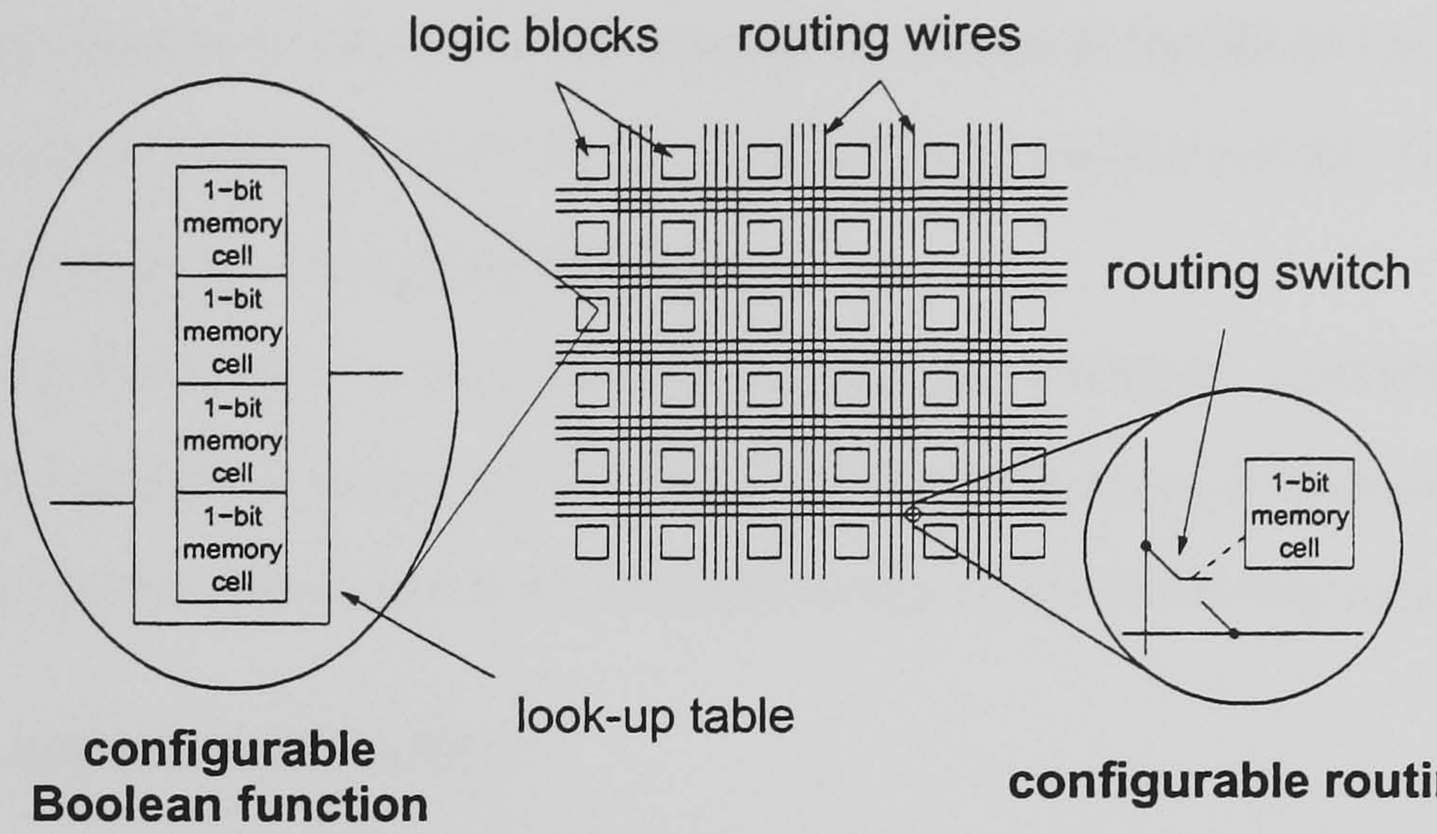


Figure 1.1: Typical architecture of a configurable logic device.

figuration becomes permanent and cannot be modified[1].

Reconfigurable technologies store hardware configurations in a re-writable memory. While many different reconfigurable logic technologies have been developed, most of the contemporary RL technologies use distributed configuration memories based on conventional 5-transistor static CMOS memory cells (Trimberger, 1994). In the remainder of this text we will refer to this type of RL technology as 'SRAM reconfigurable logic'.

The suitability of a reconfigurable logic technology for a specific application is determined by the technology characteristics, including its architecture, granularity of logic cells, interconnection structures, availability of special architectural blocks (memories, clock generators, interface cores, etc.) and others. Detailed discussions of various such reconfigurable architectures can be found in field-programmable logic textbooks (e.g. (Oldfield and Dorf, 1995; Trimberger, 1994)).

With SRAM reconfigurable logic it is possible to change the configura-

[1]In some OTP technologies it is possible to change the state of configuration memory cells, if these were left unprogrammed (in their default states). However, the state of programmed memory cells cannot be changed.

tion in a manner similar to how a program stored in the SRAM of a processor system can be changed. Thus it is possible to load or modify the circuits implemented in reconfigurable logic.

This flexibility of reconfigurable logic has attracted considerable research interest in recent years. Much of the excitement is focused around the promising application of this technology in the following areas:

- software acceleration
- hardware virtualisation
- fault tolerance
- in-field and remote hardware modification

1.1.1 Software Acceleration

In his early work Estrin (1960) has highlighted the potential of computer systems with 'flexible' architectural structures. If the processor architecture could be adapted to each computation executed on it, the speed of each computation could be improved.

Later technological advances based on work by Minninck (1964) and others have led to the development of reconfigurable technologies capable of supporting these flexible architectural features. Commercial availability of reconfigurable devices, such as SRAM Field-Programmable Gate Arrays (FPGAs) in the 80's, has allowed the practical implementation of systems based on Estrin's ideas.

Software acceleration aims to achieve computational speed-up for a specific algorithm/program by using a processor with its architecture tailored to the computational requirements of the input algorithm. The speed-up is measured as the ratio of the algorithm execution time on a processor

designed for a variety of different algorithms (*general-purpose* processor) to the execution time on a processor with its architecture customised for the accelerated algorithm (*custom* processor).

Processor architectures based on reconfigurable logic technologies can be customised for a large variety of algorithms. Some impressive speed-up results have been demonstrated for many algorithms running on reconfigurable computing machines. The early Splash-I system (a 32 FPGA linear array accelerator) was reported to outperform a standard general-purpose processor in a Sun3 workstation by a factor of 2700, while a single Cray-2 processor was outperformed by a factor of up to 300 (Gokhale et al., 1991).

High speed-up figures have been demonstrated with many specialised algorithms, however, the architectural limitations of custom-computer platforms and inherent limitations on the computational speed-up (Amdahl's law, (Amdahl, 1967)) do not allow for significant speed-up figures to be achieved for many other algorithms (Albaharna et al., 1994; Guccione, 1995). Speed-up figures in the order of tens to hundreds are being routinely reported in the custom computing community.

1.1.2 Hardware Virtualisation

With reconfigurable logic it is possible to implement designs with sizes larger than the available hardware logic resources through hardware reconfiguration. This concept of 'hardware virtualisation' is similar to the concept of virtual memory available in most modern operating systems (Silberschatz and Galvin, 1998). Hardware virtualisation is also known as logic 'time-sharing' (Lautzenheiser, 1986).

The maximal size of a program which can be executed on a given processor is limited only by external factors, such as program storage size.

Similarly, the size of a hardware circuit implemented on a reconfigurable logic device is limited only by the size of storage for RL configurations.

Hardware virtualisation has been explored for the implementation of high-performance applications of large sizes. Several implementations of neural networks on small reconfigurable logic devices have been demonstrated (e.g. (Lysaght et al., 1994; Eldredge and Hutchings, 1994)). In this approach, the computations in the neural networks are evaluated in small incremental configurations. Jones and Lewis (1995) have demonstrated a system capable of emulating large logic circuits partitioned over a number of configurations on a reconfigurable FPGA emulation platform.

1.1.3 Fault Tolerance

Early pioneers of computer science have envisaged difficulties with the operation of complex computing machines. In his theoretical work, von Neumann (1966) argued that complex computing machines will have to tolerate unreliability of their individual components as a normal part of their operation. He discussed self-repair and self-reproduction mechanisms, which could improve the reliability of complex systems.

With reconfigurable logic, the structure and connectivity of implemented circuits can be changed during the life-time of a reconfigurable system. Provided that it is possible to identify the failure of a circuit component, the functionality of such a component can be 'replaced' via hardware reconfiguration. This suggests that a significant amount of spare reconfigurable resources need to be available in the reconfigurable logic for such a system to continue to function correctly. Such redundancy, however, is commonly incorporated in the design of systems demanding high reliability (Lala, 2000), although not at such a fine-grained level.

Successful implementation of 'defect-tolerant' systems based on reconfigurable logic was demonstrated by Culbertson et al. (1997) and others.

The feasibility of self-reproduction and self-repair mechanisms in a reconfigurable logic system were demonstrated with a purpose-built dynamically reconfigurable logic system (Mange et al., 1995).

The applications of systems based on the above principles range from mission-critical or life-support systems to computing machines based on unreliable computational structures (e.g. chemical nanostructures (Heath et al., 1998) or biomolecular technologies (McCaskill and Wagler, 2000)).

1.1.4 In-Field and Remote Hardware Modification

The ability to modify reconfigurable logic designs allows for errors to be corrected and new features to be added after the product was released. These updates can be performed in the field and even remotely over a distributed network. Such features become vital in the environment where short time-to-market and short product life cycles force application developers to accommodate new features late in the development cycle or during the product life-time (Kean, 1999).

Slow standardisation efforts fail to keep up with the customer demand for new products. This situation is forcing manufacturers to develop products with incomplete draft standards and then update the final products to comply with the final standard documents.

The above features of reconfigurable logic have gained much importance with the recent development of large-complexity and single-chip systems. In these systems the reconfigurable logic will provide the required flexibility to cope with the uncertainties of product design, reliability, stan-

dards or feature demand.

Remote changes of reconfigurable logic circuits will provide benefits to many systems requiring hardware changes during their life cycle. These systems include reconfigurable communication systems, consumer electronics products (e.g. digital TV), automotive systems, and others. One other example of such a promising application are the systems using hardware reconfiguration in remote and harsh environments (Brebner and Bergmann, 1999).

Although the above features of reconfigurable logic have gained considerable interest only recently, this concept has been known to processor architects for many years. Early implementations of IBM 370 and DEC VAX computer architectures developed in the 70's utilised a Writeable Control Store (WCS)—an SRAM which could hold custom microcode (Hennessy and Patterson, 1990; Edwards, 2000). Through WCS it was possible to 're-place' faulty instructions with the new ones, thus allowing corrections of hardware bugs after the entire computer system was manufactured. Similar features can be found in many contemporary processors today.

1.2 Reconfigurable Logic in Real-World Applications

Recent technological advances in the area of reconfigurable logic have produced technologies capable of integrating designs with over one-million gates operating at system clock frequencies of hundreds of megahertz. In contrast, other ASIC technologies (gate arrays, standard cells, full-custom) can provide an order of a magnitude better performance and capacity. Despite these limitations reconfigurable logic technologies are becoming increasingly popular for design implementation in low to medium volumes. However, to date applications using hardware reconfiguration in practical,

real-world scenarios are rare.

Let us examine two principal difficulties which contemporary users of reconfigurable logic are confronted with:

- **Limited reconfiguration performance.** The reconfiguration latency overhead of current reconfigurable technologies is prohibitive for many practical applications. While the system clock period of contemporary general-purpose processors reduces below 1 ns, the reconfiguration time for typical commercial devices remains in the order of tens of microseconds and more.

  Furthermore, the flexibility gain of reconfigurable technology can often be outweighed by alternative approaches using microprocessor or non-reconfigurable technologies. Several technological approaches have been proposed to reduce the reconfiguration overheads (e.g. (Brown et al., 1994; Vasilko and Ait-Boudaoud, 1996b)), however, a practical commercial implementation of these technologies is yet to be demonstrated.

  Chapter 2 discusses further the various approaches used for the construction of reconfigurable logic devices.

- **Lack of suitable CAD tools and methodologies.** Most of the design tools and methodologies available for reconfigurable systems are based around static (i.e. non-reconfigurable) design flows.

  Using such design flows for the design of reconfigurable systems can lead to a large number of design iterations due to an inability to accommodate reconfiguration overhead metrics early in the design flow. Automation of this process has been limited due to its computational complexity and the difficulties in estimating the low-level

design metrics at higher abstraction levels. A more detailed discussion of the topic of automatic design for reconfigurable logic, and a review of the past work in this area, is provided in Chapter 3.

1.3 This Thesis

This thesis addresses the need for an automatic design flow for dynamically reconfigurable systems. A new approach is proposed, which allows simultaneous searching through the design space of a reconfigurable system at behavioural and layout levels, while attempting to satisfy the overall design constraints.

The presented approach uses a new model for the process of reconfigurable system synthesis, which permits technology-dependent and solution-dependent characteristics (such as position-dependent configuration latency) to be calculated and inserted into the model during the solution search. Therefore realistic estimates of reconfiguration overheads can be considered during the solution search. This approach guarantees that the automatically produced design solutions are feasible, i.e. they can be implemented using the target reconfigurable device without the occurrence of resource or dependency conflicts.

This thesis is composed as follows. **Chapter 2** provides background for reconfigurable systems with a particular emphasis on the reconfigurable logic technologies. The chapter examines the characteristics of several technological approaches used for the construction of reconfigurable logic devices in order to demonstrate how the technology-specific features complicate the design considerations during the reconfigurable system design. **Chapter 3** outlines the various techniques used for the design of reconfigurable systems, summarises the previous work significant to the area

of automatic synthesis for reconfigurable systems and highlights some of the shortcomings of the previously proposed approaches. **Chapter 4** introduces a new model for the process of synthesising a design for a reconfigurable system. The model provides the formalism necessary for the discussion of the design techniques presented in the following chapters. **Chapter 5** presents a new experimental CAD framework developed to support the work presented in this thesis. **Chapter 6** presents an automatic synthesis technique for a restricted instance of this problem based on the use of genetic algorithms. The qualities of this approach have been tested using a number of benchmark problems. These experimental results are presented in **Chapter 7**. The contribution of this thesis is summarised in **Chapter 8**, which also provides suggestions for future work in this area.

Chapter 2

Reconfigurable Systems: Background

Reconfigurable systems combine features which have previously existed individually in either software or hardware systems. Reconfigurable systems provide flexibility similar to that of processor-based software systems, while their performance is close to custom hardware circuits. Reconfigurable systems are therefore often viewed as a 'missing link' between software and hardware systems.

The previous chapter has discussed the historical motivation for the development of reconfigurable logic systems. This chapter provides a background for reconfigurable systems and reconfigurable logic technologies. The design difficulties resulting from the availability of different technological approaches for supporting reconfiguration are discussed using a simple example at the end of this chapter.

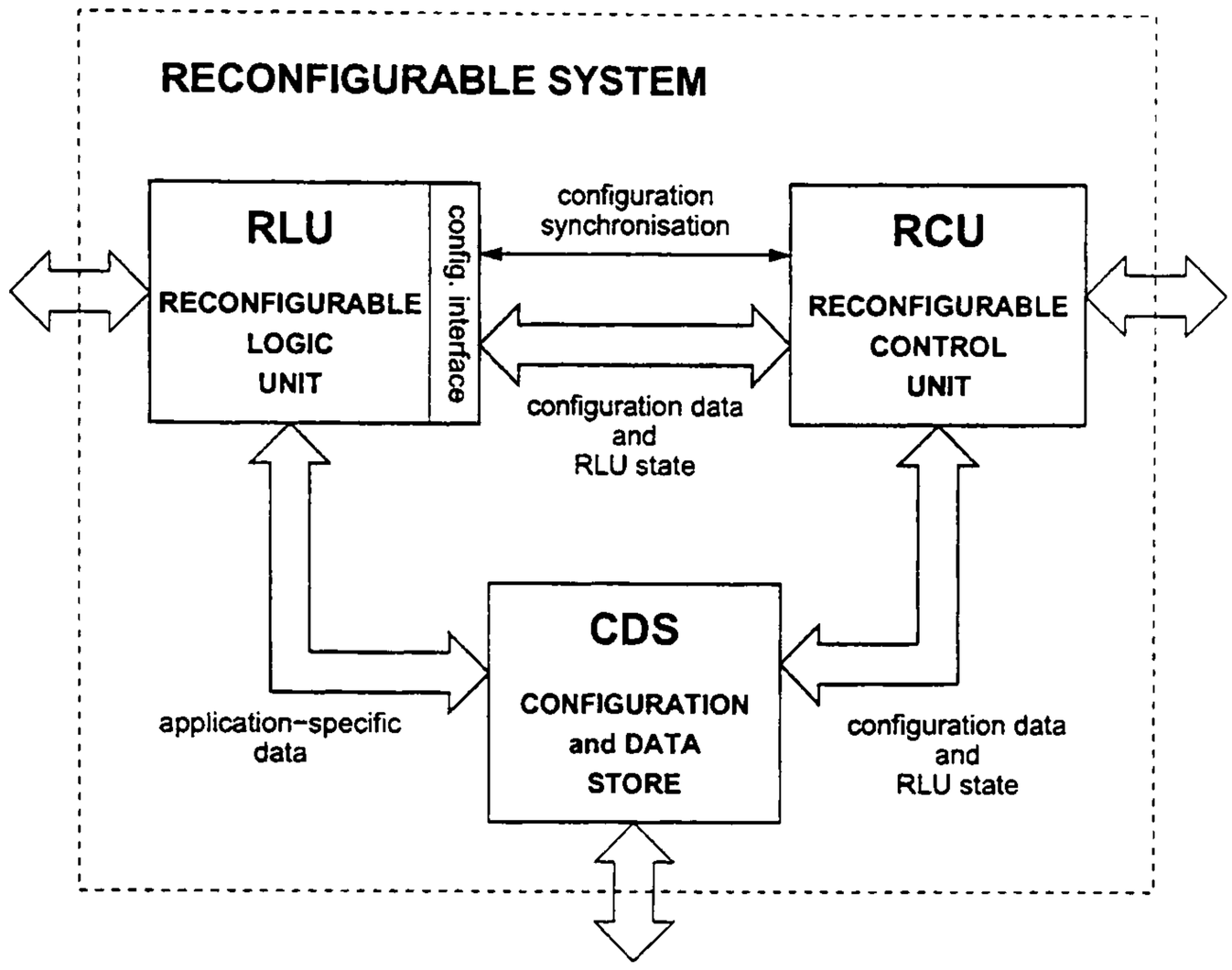


Figure 2.1: Typical architecture of a reconfigurable logic system.

2.1 Typical Architecture of a Reconfigurable System

A variety of reconfigurable systems and applications has been demonstrated, including those exploiting the less traditional concepts of self-reconfiguration (e.g. (French and Taylor, 1993; Sidhu et al., 1999; McGregor and Lysaght, 1999)) and hardware evolution (e.g. (Thompson, 1996; Tangen, 2000)). The majority of these systems are built around a typical architecture shown in Fig. 2.1.

A typical reconfigurable system architecture includes:

- **The Reconfigurable Logic Unit (RLU)** which contains reconfigurable logic and routing resources, but may also provide memory blocks, configurable clock generators, etc. The state of all RLU resources can be accessed via the RLU's configuration interface. In many contemporary systems, the RLU is represented by a single FPGA device,

while the FPGA configuration interface provides access to its configuration memory.

- **The Reconfiguration Controller Unit (RCU)** which manages the RLU reconfiguration. The RCU typically allows loading and retrieval of the RLU resource data states and configurations, although the availability of these features depend on the RLU-specific capabilities.

  The RCU unit can be implemented in a hardware controller or a processor. It can operate either as a unit dedicated to the task of configuration control, or its functionality can be provided by units shared between other system functions (e.g. on a processor running a real-time operating system (RTOS)). Furthermore, the RCU may be integrated within the RLU. In such a scenario, the RCU can either control the RLU's reconfiguration via a single configuration interface or the RCU may provide a distributed reconfiguration control mechanism.

- **The Configuration and Data Store (CDS)** which provides memory storage for both the RLU configuration and state data, and application-specific data.

  There are many options for the CDS implementation, including a single memory block, two memory blocks (one for configuration data and one for application data), hierarchical memory structures, and others.

A reconfigurable system can operate either as a self-contained unit or it can be embedded in a larger system. The complexity of the individual RS units and the complexity of the overall operation will vary greatly with the application requirements.

Simple reconfigurable systems may implement the RLU on a single

FPGA device, the RCU on a dedicated configuration controller with an address counter and control generator, while the CDS can store the RLU configurations in a read-only memory. Reconfiguration of such a system can be initiated by an external control signal and may occur more or less frequently.

Complex reconfigurable systems may be part of a larger real-time system, implementing the RLU using several FPGA devices and field programmable crossbar switches, the RCU on a processor running a real-time operating system, responsible for the management of real-time event initiated reconfiguration, while also servicing other system functions. The CDS may be placed on a shared system memory with the system's RTOS being responsible for the management of memory sharing between system components.

2.2 Reconfigurable Logic Technology

Several technological approaches are available for the implementation of the RLU discussed in the previous section. The technological features of reconfigurable technologies can be separated into two main categories:

- **Architectural features:**
  - logic block functionality and granularity
  - structure of routing resources
  - architectural organisation of logic and routing
  - availability of application-specific architectural components (fast local memory, fast carry-chains, etc.)
  - availability, characteristics and number of external connections

- **Configuration capabilities:**
  - organisation of configuration memory and data
  - reconfiguration interface throughput and characteristics

Reconfigurable logic has evolved from technologies developed for FPGAs. Therefore architectural tradeoffs for the implementation of reconfigurable logic are similar to those of FPGAs. A detailed discussion of the architectural tradeoffs for FPGA architectures can be found in a number of FPGA technology textbooks (e.g. (Trimberger, 1994; Oldfield and Dorf, 1995)).

A number of techniques have been developed to provide configuration interfaces optimised for reconfigurable logic. The main techniques are summarised in the following section.

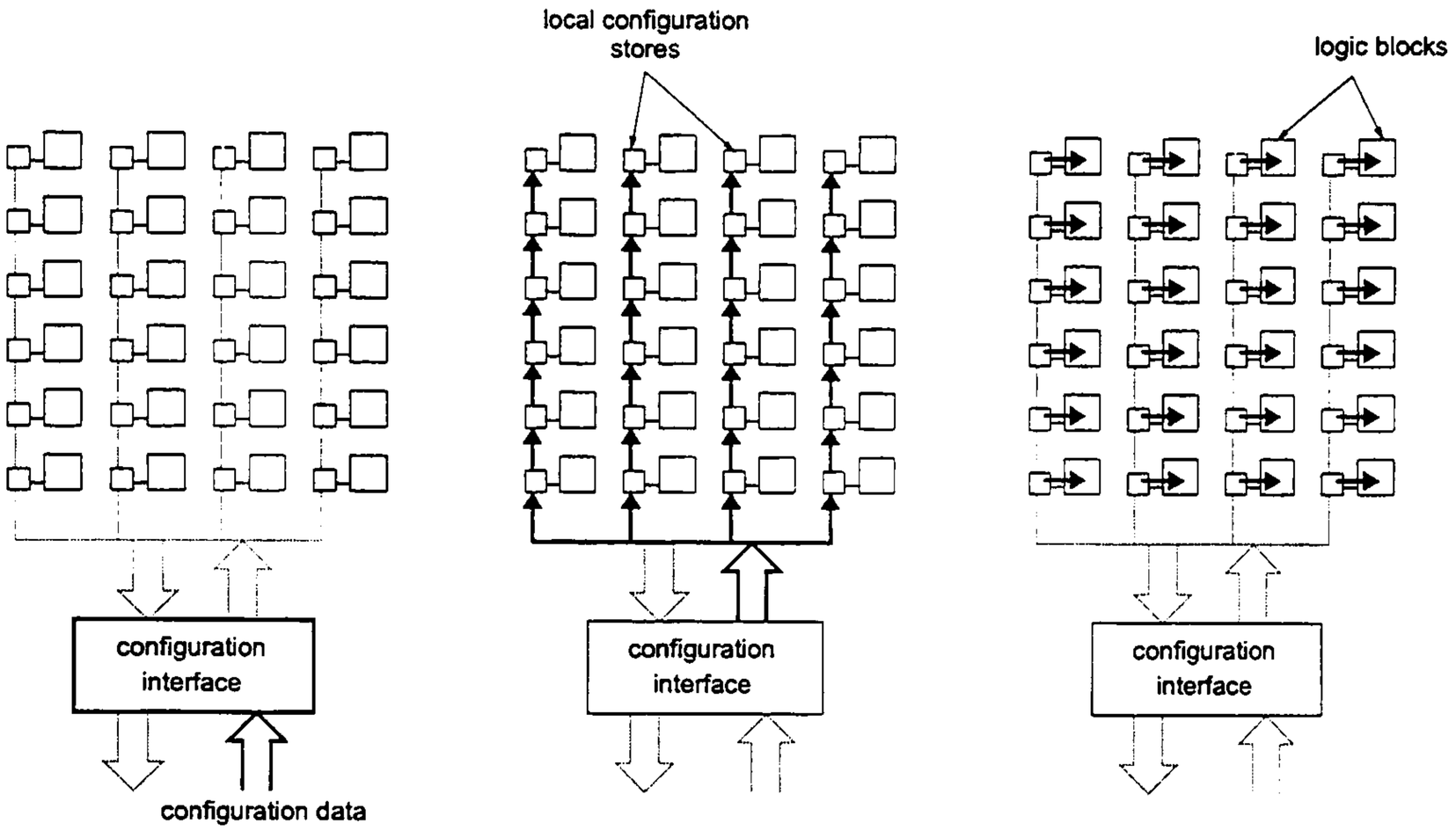
2.2.1 Support for Reconfiguration

This section summarises the main techniques used in reconfigurable technologies for the construction of circuits supporting reconfiguration. The performance of reconfiguration for a specific type of reconfigurable technology is determined by the performance of its individual components:

1. **External configuration interface**, which transfers configuration data from the external sources (e.g. CDS in Fig 2.1) to the internal structures of a reconfigurable device (Fig. 2.2(a)).

2. **Configuration data distribution network**, which transports the configuration data to the individual configuration memory cells (Fig. 2.2(b)).

3. **Configuration activation scheme**, which activates the configuration data by connecting the selected configuration memory cells with the configurable components of a reconfigurable system (Fig. 2.2(c)).



(a) external transfer (b) distribution (c) activation

Figure 2.2: Propagation of the configuration data through the reconfiguration subsystem.

Although in the following the implementation of these components in FPGA technologies is discussed, these considerations are applicable generally to the implementation of the RLU; regardless of whether the RLU contains a single FPGA device or a combination of FPGA and other reconfigurable technologies.

The *configuration performance* depends on the throughput of all of the above components. Unbalanced configuration subsystems require configuration buffering between the configuration components with different throughput. The overall bandwidth of a reconfiguration subsystem is limited by the throughput of its slowest component.

2.2.2 Configuration Interface

The connection of the configuration interface to the external world determines the speed at which the configuration data can be transferred to the reconfigurable device. A high-throughput *parallel configuration interface* connection is well suited for fast configuration transfers. However, it requires that a number of device I/O ports are assigned to this function. In those implementations where external ports are in demand or when fast reconfiguration is not required, the parallel interface might not be desired. In such cases, the *serial configuration interface* provides a low-throughput and low-cost alternative. Most of the contemporary reconfigurable devices provide a dual parallel/serial interface allowing users to select the interface best suited for the targeted application.

After the configuration data has been received from an external source, the configuration interface circuitry will arrange the data in the format suitable for distribution within the reconfigurable logic array.

2.2.3 Configuration Data Distribution

Similar tradeoffs between serial and parallel access apply for the distribution of the configuration data in the reconfigurable logic array. The following are two examples of common serial and parallel distribution mechanisms.

2.2.3.1 Serial distribution

Early commercial reconfigurable logic devices were designed for configuration only during the system power-up. This scenario required a configuration interface, which is simple and has only minimal area and pin overhead in the reconfigurable logic array.

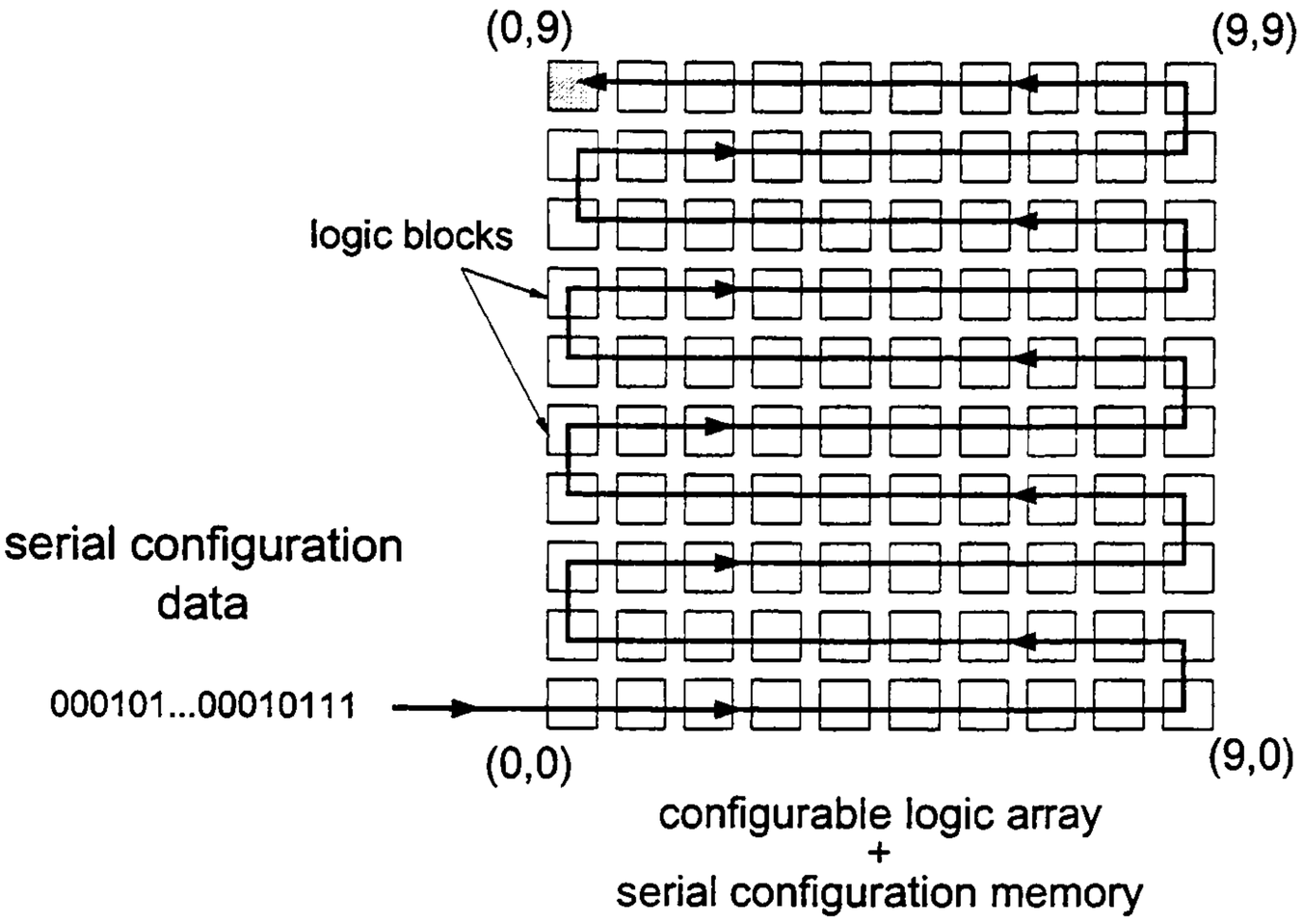


Figure 2.3: Serial configuration data distribution.

A *serial* configuration distribution (Fig. 2.3) was used in these technologies. Typically, the configuration memory is arranged as a long shift-register. In order to program a device, the configuration data is shifted from the configuration interface to the configuration store bit-by-bit. The device is activated for normal operation once the entire configuration memory has been filled with the configuration data. Xilinx XC2000/3000/4000 (Xilinx, 1994) and Altera FLEX 8k/10k (Altera, 1995) FPGA families are examples of the technologies with serial distribution of the configuration data.

In order to improve the throughput of the configuration interface these families also provide parallel access to their configuration interfaces. These configuration interfaces internally reformat the parallel configuration data into a serial configuration 'bitstream'.

Early implementations of configuration distribution mechanisms required that with each reconfiguration the *entire* configuration memory must be filled with configuration data before the new configuration can be used.

The term *full reconfiguration* is used to denote this type of reconfiguration.

Later configuration distribution mechanisms have allowed for only a portion of the reconfiguration memory to be modified, while the rest of the system remains unchanged. This type of reconfiguration is called *partial reconfiguration*.

The term *dynamic reconfiguration* is used to refer to cases when it is possible to perform reconfiguration while the system remains in operation. In this text, the term 'dynamic reconfiguration' denotes a *temporal* quality of a reconfigurable system, while the terms 'partial/full reconfiguration' denote its *spatial* quality[1].

The Atmel AT6000 (Atmel, 1994) is an example of an FPGA technology with serial configuration data distribution, which supports both partial and dynamic reconfiguration[2].

The main drawback of the serial configuration data distribution mechanism is its low configuration data throughput. Although partial reconfiguration can reduce average configuration time, in the worst case it is still necessary to shift the configuration data through the entire array (e.g. consider the configuration of the block at position (0,9) in Fig. 2.3).

On the other hand, this type of reconfiguration distribution mechanism is simple to implement as it does not incur a large area overhead. It provides sufficient performance and flexibility for many applications which do not require rapid reconfiguration.

2.2.3.2 Parallel distribution

With the development of reconfigurable computing, it became desirable to provide a faster and more direct method of accessing configuration data

---

[1]The use of this terminology is summarised in the Glossary on page 161.

[2]Atmel use the term *Cache Logic*™ to refer to this feature.

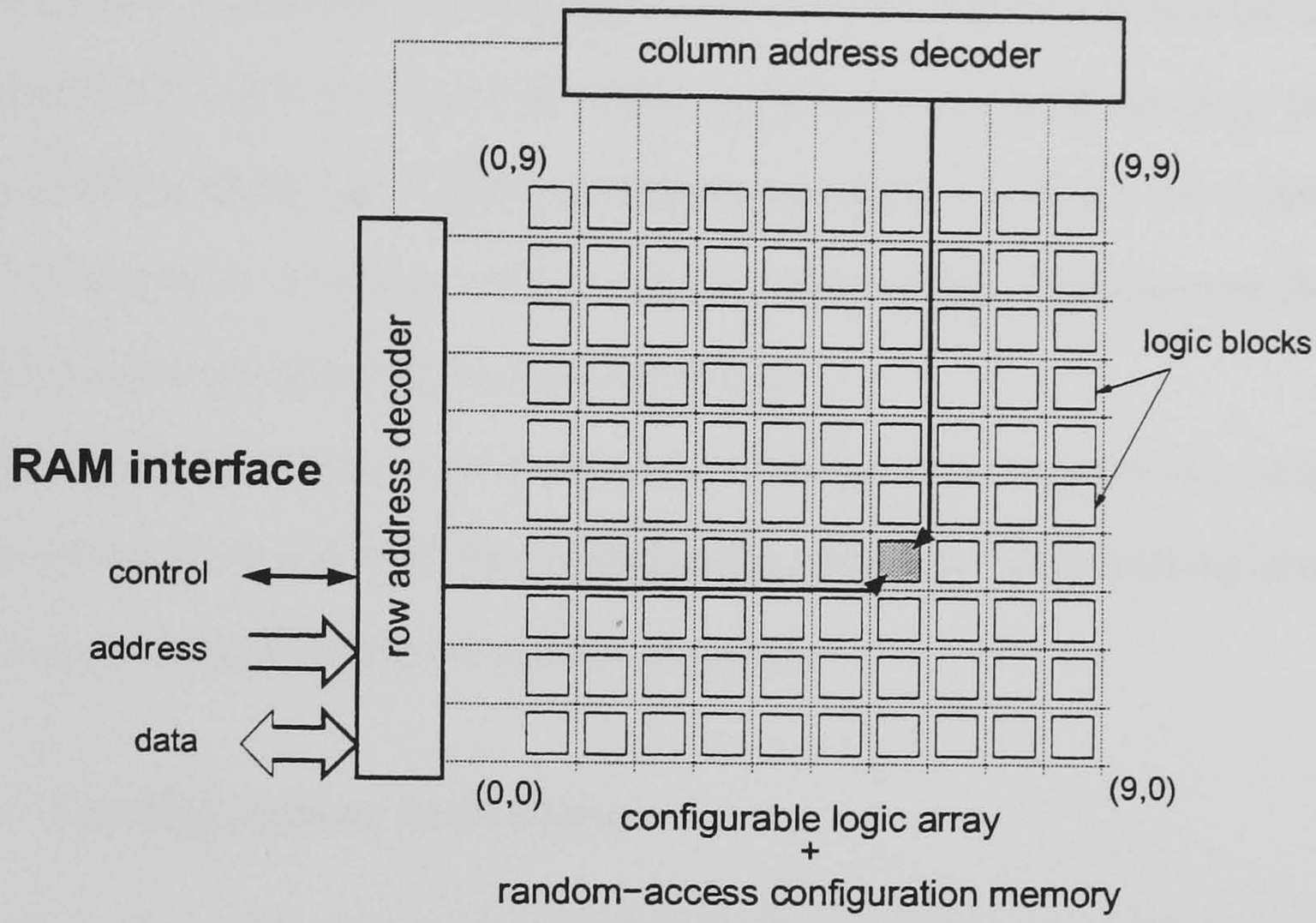


Figure 2.4: Random-access configuration memory.

and the RLU state (e.g. from a coupled microprocessor). A configuration distribution mechanism allowing random access to the RLU configuration memory was developed in response to these needs (introduced by Kean (1988), prototyped in the Algotronix CAL1024 device (Algotronix, 1991) and later improved in the Xilinx XC6200 FPGA family (Churcher et al., 1995; Xilinx, 1997b)).

This is an example of a *parallel* configuration distribution mechanism, where the internal configuration memory is organised in a structure similar to that of a conventional single-port RAM (Fig. 2.4). The configuration data and the RLU state can be set and retrieved through addressing the appropriate location in a random-access configuration memory.

Through this configuration distribution mechanism, the configuration memory appears as an ordinary RAM, which can be easily interfaced to standard processor-based systems.

The main advantage of this approach is that individual words in the configuration memory can be accessed quickly at any address (e.g. in Xilinx XC6200 technology, a 32-bit configuration data word can be written in the configuration memory within a 30 ns write cycle). This approach also provides a partial reconfiguration capability.

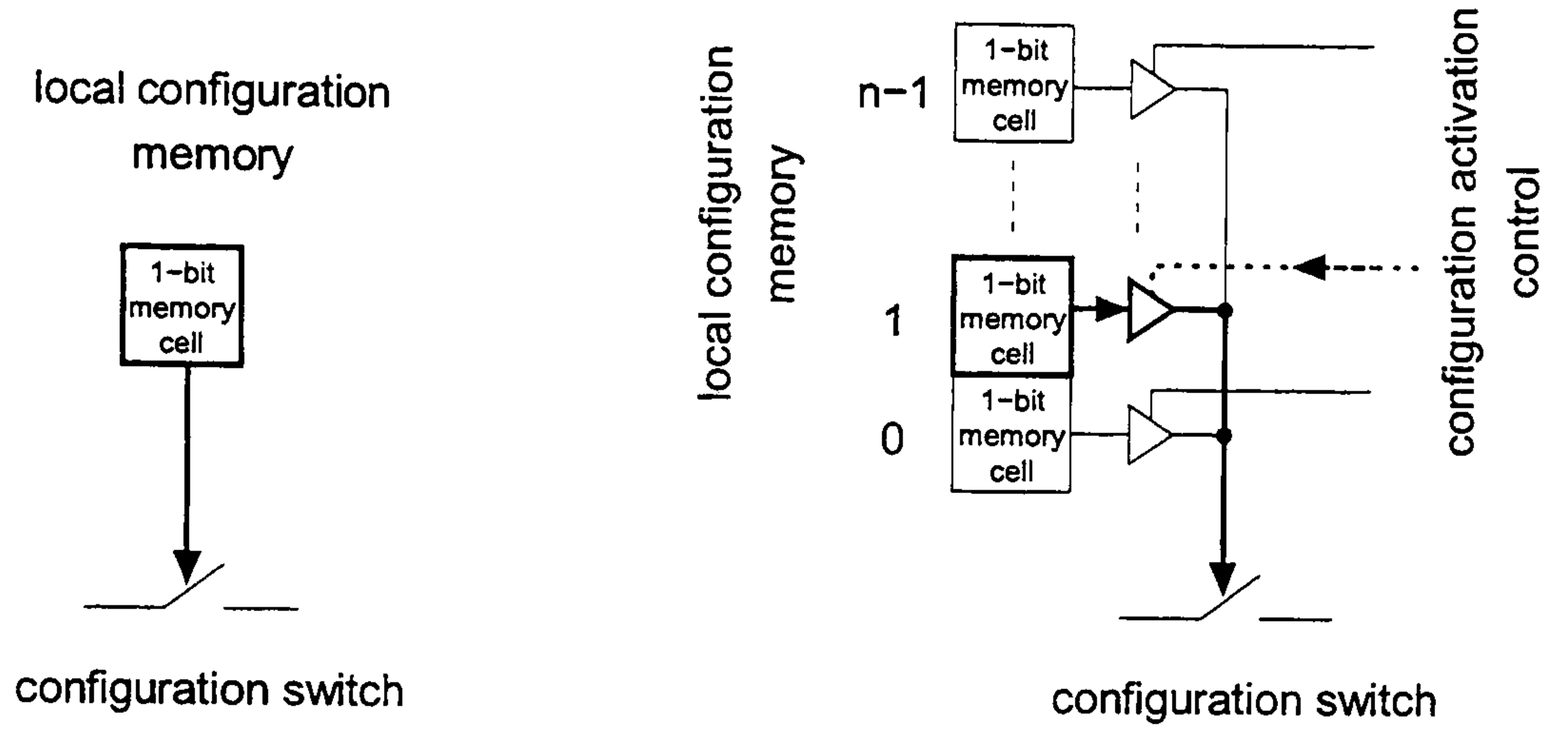
The approach requires an increased reconfigurable logic array area due to the necessity to provide the configuration address/data routing and the logic supporting the random memory access.

2.2.4 Configuration Activation

A configuration activation scheme determines how the distributed configuration data is transferred to the programmable switches within the reconfigurable array. The following are examples of two approaches used for the activation of the configuration data:

2.2.4.1 Direct one-to-one activation

Traditional reconfigurable logic devices provide one configuration memory cell for each configurable switch in the array (Fig. 2.5(a)). In this case, the configuration memory cell is directly connected to the configurable switch. The new configuration is activated immediately after the configuration memory cell has been written with the new data. Examples of reconfigurable devices with direct one-to-one activation, include Xilinx XC2000/3000/4000 (Xilinx, 1994), XC6200 (Xilinx, 1997b) and Altera FLEX 8k/10k (Altera, 1995) FPGA families.



(a) one-to-one

(b) many-to-one (n-to-1)

Figure 2.5: One-to-one versus many-to-one configuration activation.

2.2.4.2 Many-to-one activation

Configuration sub-systems connecting several configuration cells to one configuration switch were proposed to accelerate the speed of reconfiguration (e.g. (Brown et al., 1994; Trimberger et al., 1997)). These systems provide a *multiple-context* configuration memory, which contains more than one 'layer' of the configuration data storage (Fig. 2.6). Once the configuration data has been pre-loaded into the context layers, the configuration can be activated by selecting configuration cells in one of the memory layers (Fig. 2.5(b)). This process is often referred to as a 'context switch' because of its similarities to CPU context switching in multi-process operating systems.

The main advantage of this system is the speed of reconfiguration. Provided that all configuration memory contexts can be *pre-loaded* with the desired configuration, the configuration for the entire device context can

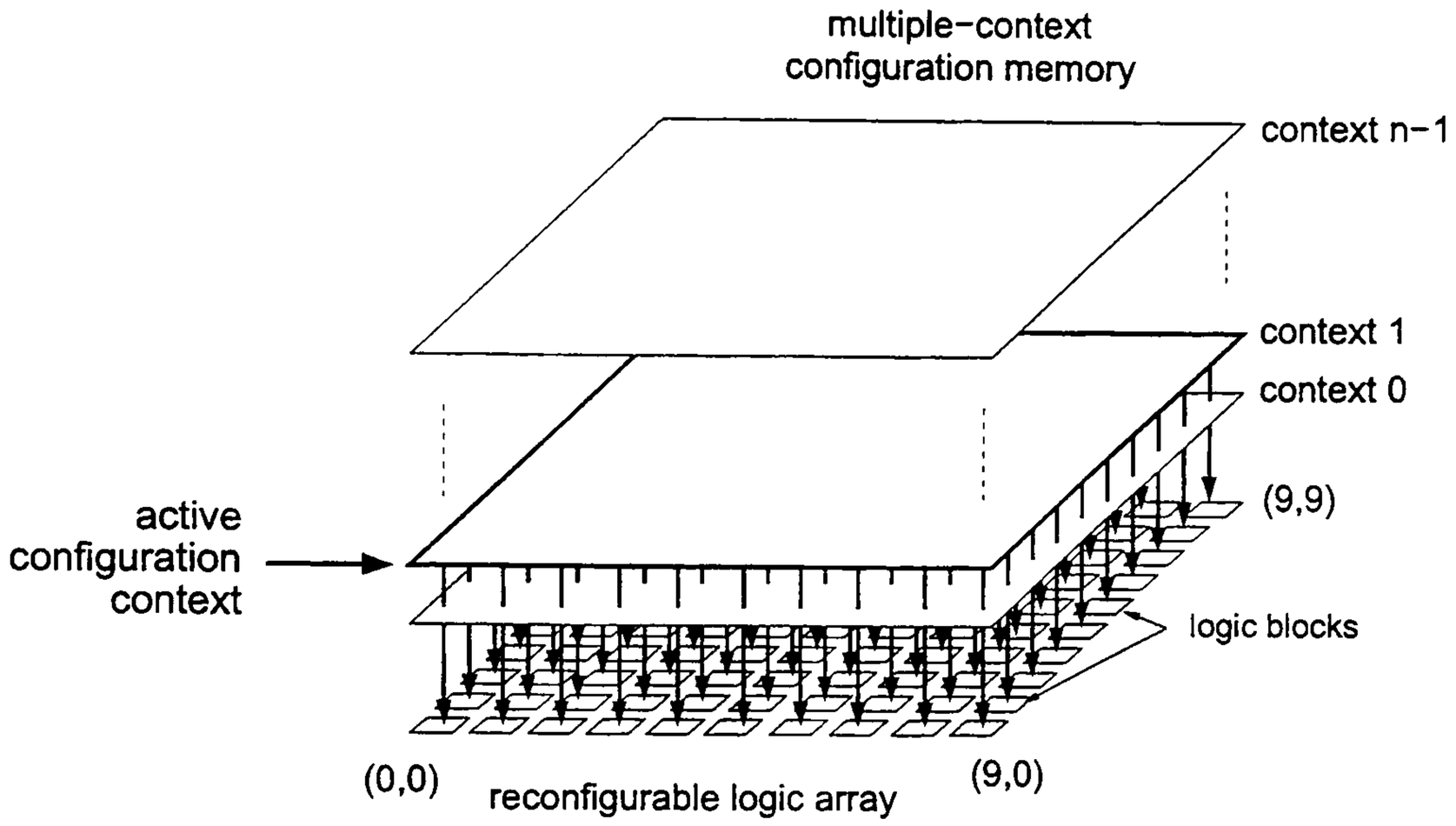


Figure 2.6: Multiple-context configuration memory.

be changed within several nanoseconds (e.g. 3 ns in MIT DELTA prototype (DeHon, 1996) and 30 ns in Xilinx Time-Multiplexed FPGA (Trimberger et al., 1997)).

The main drawbacks of this approach are a large silicon area overhead required for the configuration context memory and its control signals, longer routing delays due to the increased spacing between the logic blocks and unpredictable power consumption during reconfiguration (Trimberger et al., 1997).

2.3 Reconfiguration Latency

The various reconfiguration mechanisms provide a tradeoff between the configuration throughput and the area overhead required for the implementation of a reconfiguration sub-system. In particular, the time needed for the configuration of a design module will vary with the technology. For

partially reconfigurable technologies, the required configuration time will also vary with the current contents of the configuration memory and therefore is dependent on the placement of modules in the design solution and their mapping to primitive device elements. This is demonstrated using the following example.

Consider a reconfigurable logic array of size 31 × 31 with an architecture similar to that of the Xilinx XC6200 FPGA family (Xilinx, 1997b). The reconfigurable logic array has a fine-grained architecture and offers several different configuration modes. The relevant details of this technology and the modules used in this experiment are summarised in Appendix A.

In the example, first the content of the configuration memory was cleared, and then a 4-bit adder was configured at coordinate (0,0). We will examine the configuration latency required for the configuration of a 4-bit subtractor onto this array at different coordinates within a 17 × 17 square area (Fig. 2.7). The reconfiguration overhead was measured as the minimal number of configuration cycles required to complete the configuration of the entire 4-bit subtractor (C_{sub}).

Reconfiguration overhead routines calculate the differences between the two configurations (i.e. an empty XC6200 device + adder configuration versus subtractor configuration) for all model XC6200 technology configuration subsystems, while accommodating the addressing restrictions of their respective configuration interfaces. The configuration differencing routine compares the configuration of XC6200 technology primitive elements (i.e. logic and routing multiplexors), while ignoring the elements which are not used in the subtractor module circuit.

The entire experiment was implemented within the DYNASTY Framework (Chapter 5). Reconfiguration using the following configuration sub-

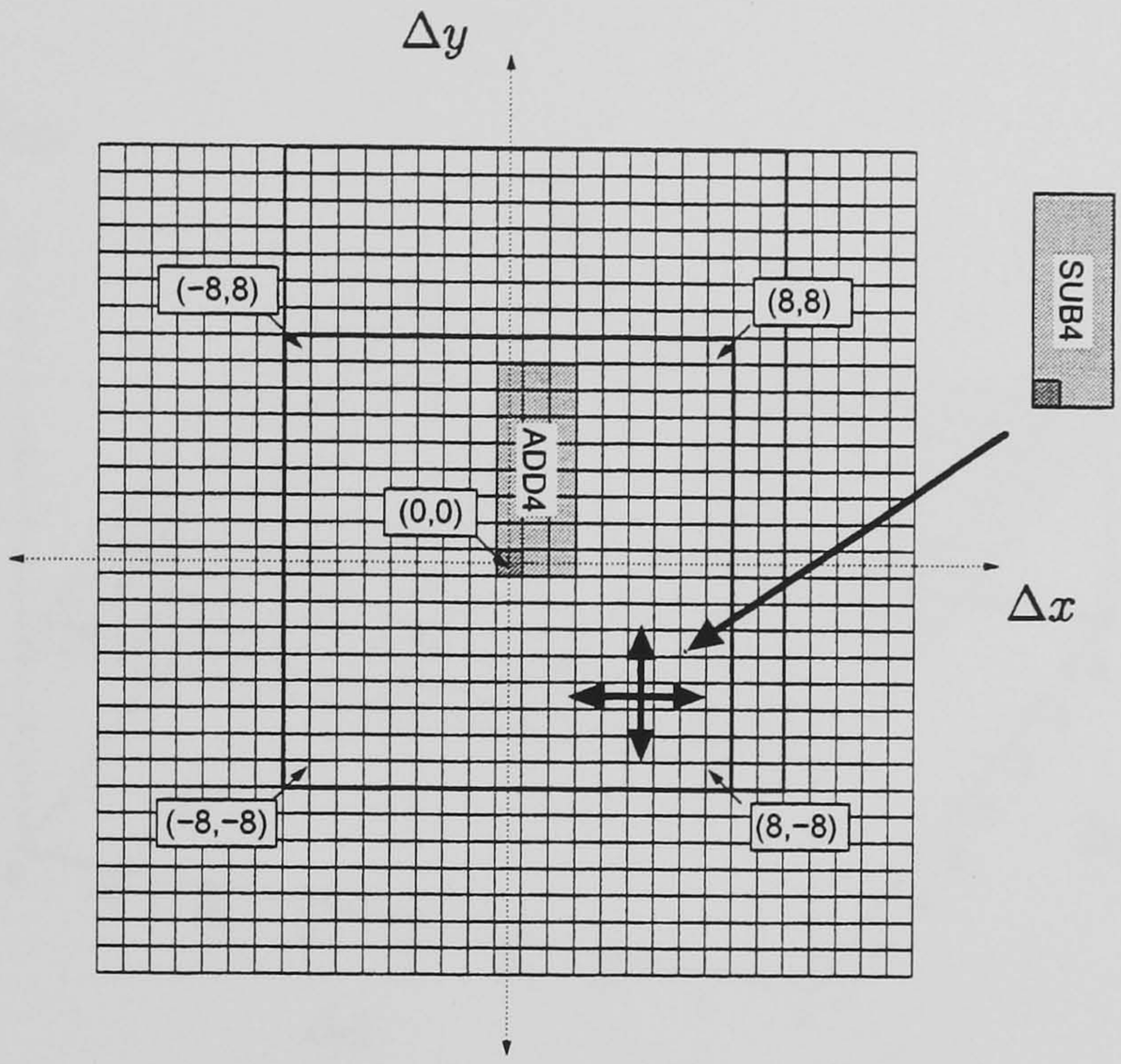


Figure 2.7: 4-bit subtractor configuration experiment.

systems was examined:

1. Partial reconfiguration via an 8-bit parallel random access configuration interface: Figs 2.8 and 2.9 show C_{sub} as a function of the offset from the base coordinate $(0, 0)$. Due to similarities between the adder and subtractor modules, the minimal reconfiguration latency is at offset $(0, 0)$. Configuration latency varies with the subtractor module position between 15 and 54 configuration cycles.

2. Partial reconfiguration via a 32-bit parallel random access configuration interface (Figs 2.10 and 2.11). Compared to the previous case, the reconfiguration interface offers a higher configuration throughput (32 configuration data bits/cycle). The minimal reconfiguration latency is lower and also varies with the module position. Note that the latency also varies in the locations with empty configuration memory

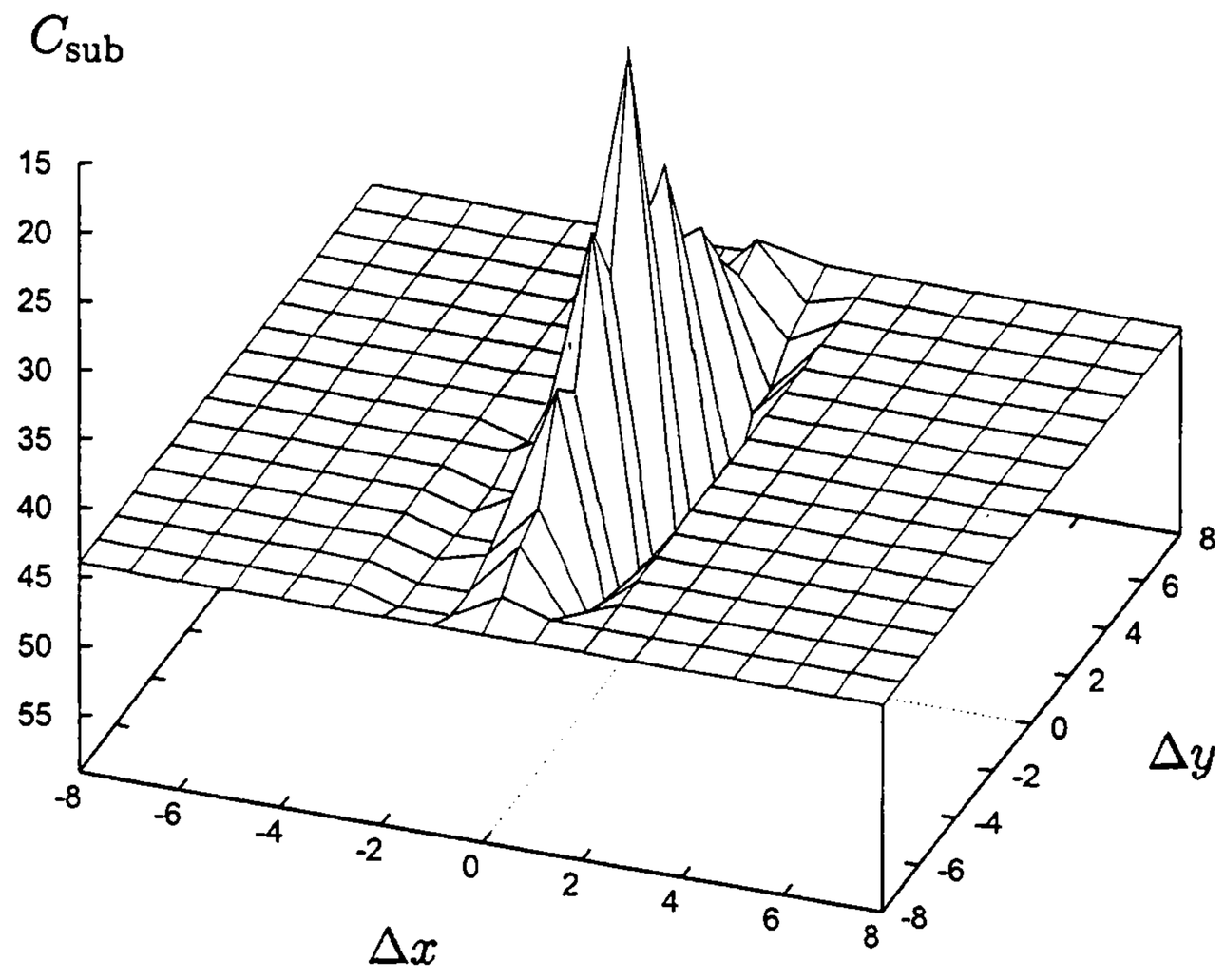
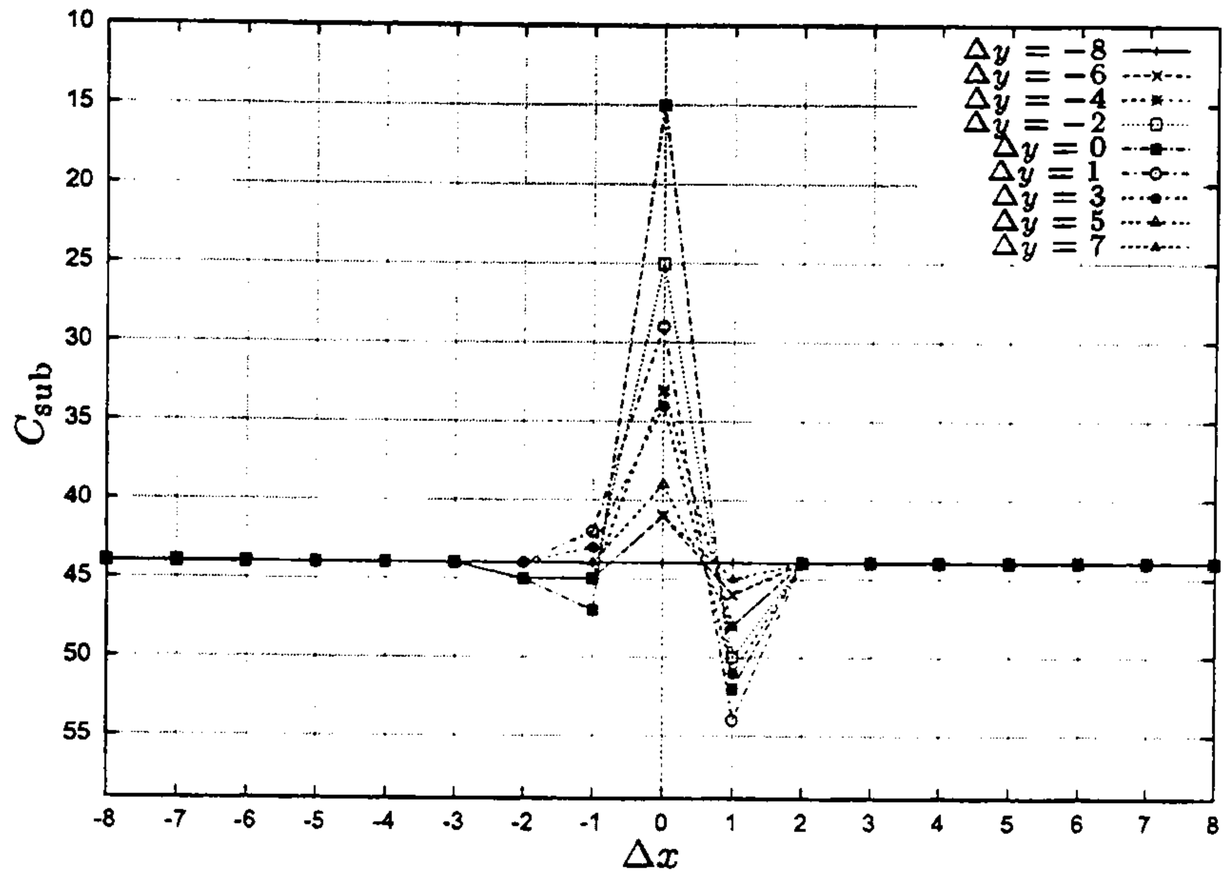


Figure 2.8: $C_{\text{sub}} = f(\Delta x, \Delta y)$, subtractor module configuration latency C_{sub} as a function of the offset $(\Delta x, \Delta y)$ against the adder module (8-bit parallel random access configuration interface).

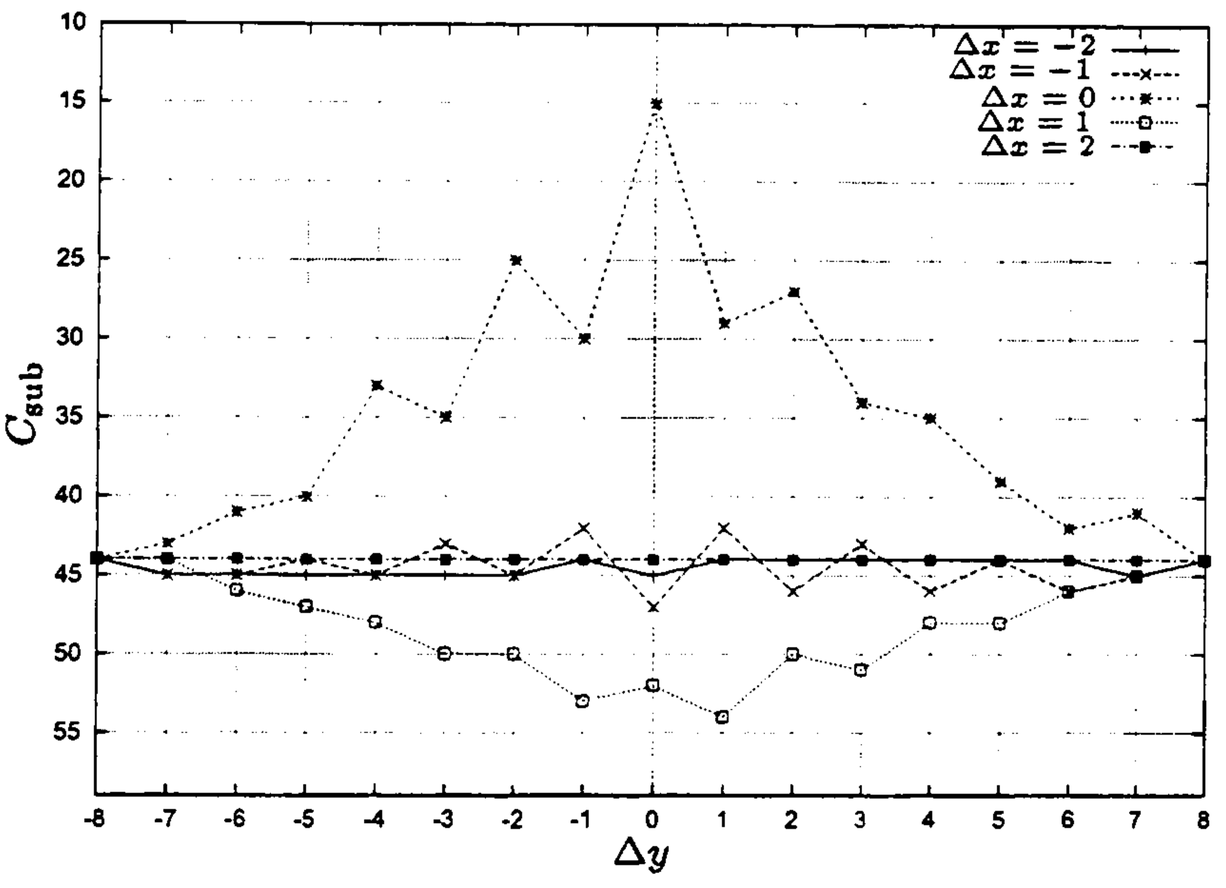
(e.g. $\Delta y = 2$ in Fig. 2.11(b)). This is caused by the column-based alignment of the 32-bit configuration words within the configuration memory.

3. Partial reconfiguration via a serial column access configuration interface (Fig. 2.12). Throughput of the configuration interface is lower then in the previous two cases (less then 1 configuration data bits/cycle) resulting in much higher absolute reconfiguration latency. The addressing structure of the configuration memory requires that the entire column is reconfigured even if only one bit in a column needs to be changed.

In the case of the subtractor module reconfiguration, the reconfiguration time is reduced only at coordinate (0,0). Also the reconfiguration



(a) $C_{\mathrm{sub}} = f(\Delta x), \quad \Delta y = \{-8, -6, -4, -2, 0, 1, 3, 5, 7\}$



(b) $C_{\mathrm{sub}} = f(\Delta y), \quad \Delta x = \{-2, -1, 0, 1, 2\}$

Figure 2.9: X-Y cross-sections for the diagram shown in Fig. 2.8 (8-bit parallel random access configuration interface).

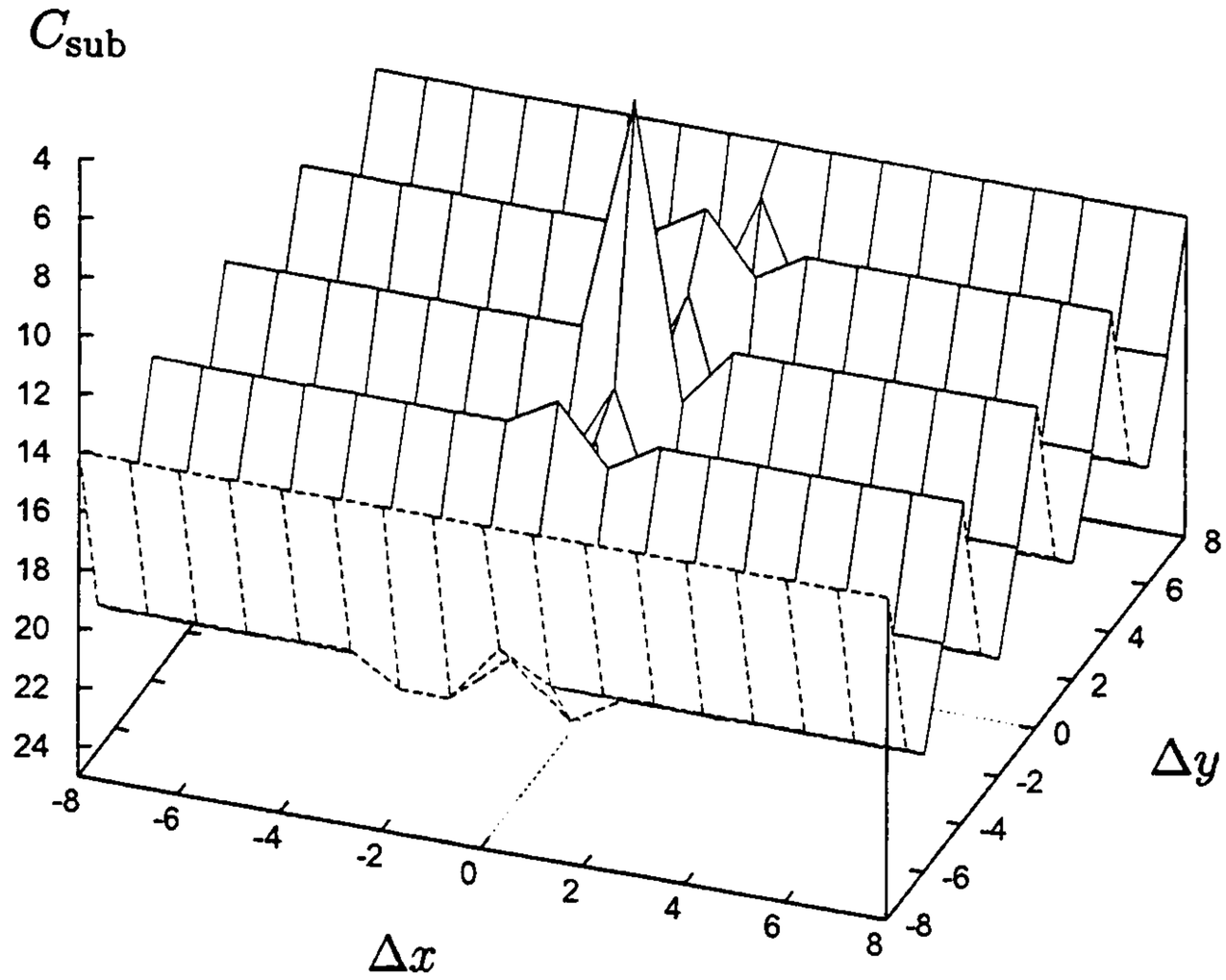
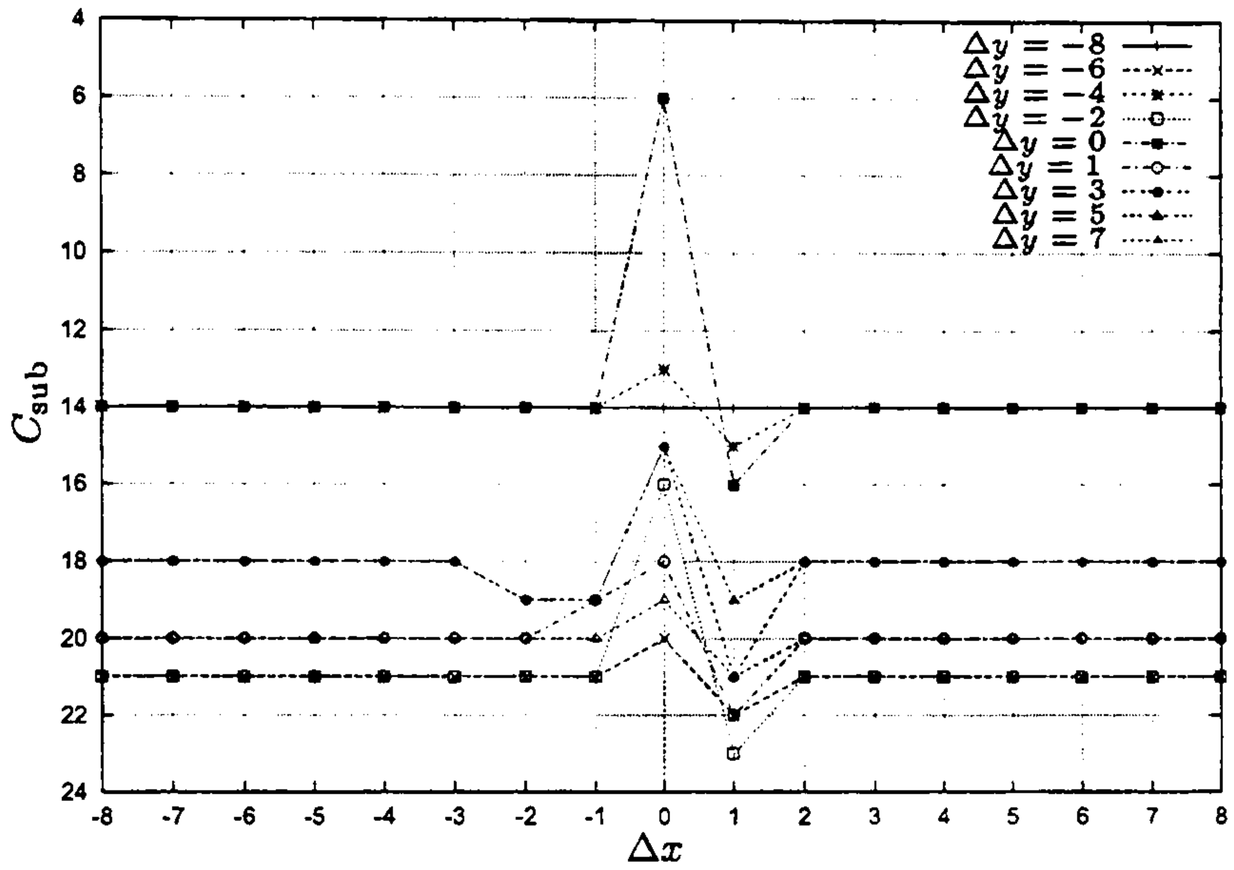


Figure 2.10: $C_{\text{sub}} = f(\Delta x, \Delta y)$, subtractor module configuration latency C_{sub} as a function of the offset $(\Delta x, \Delta y)$ against the adder module (32-bit parallel random access configuration interface).

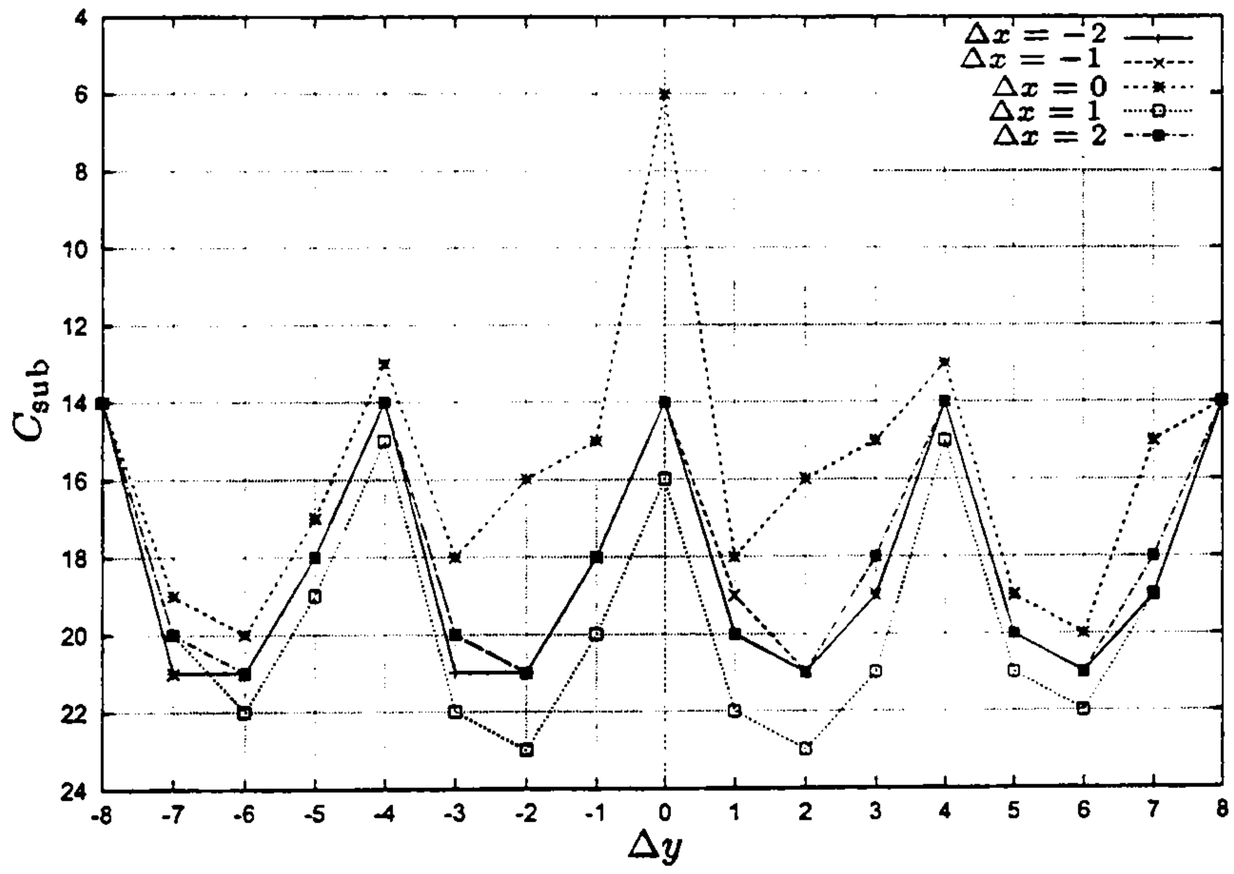
latency function has been simplified considerably.

4. Full reconfiguration with multi-context configuration memory, pre-loaded with the subtractor module configuration data (Fig. 2.13). With this type of reconfiguration memory, it is possible to switch between the pre-loaded context within one configuration cycle. If the full context switch is performed, the time required for reconfiguration is constant regardless of module position or previous contents of the configuration memory.

   However, if the new configuration has to be loaded via the external configuration interface during the run-time, the performance of the reconfiguration will decrease considerably. Such a situation is not considered in this example.



(a) $C_{\mathrm{sub}} = f(\Delta x), \quad \Delta y = \{-8, -6, -4, -2, 0, 1, 3, 5, 7\}$



(b) $C_{\mathrm{sub}} = f(\Delta y), \quad \Delta x = \{-2, -1, 0, 1, 2\}$

Figure 2.11: X-Y cross-sections for the diagram shown in Fig. 2.10 (32-bit parallel random access configuration interface).

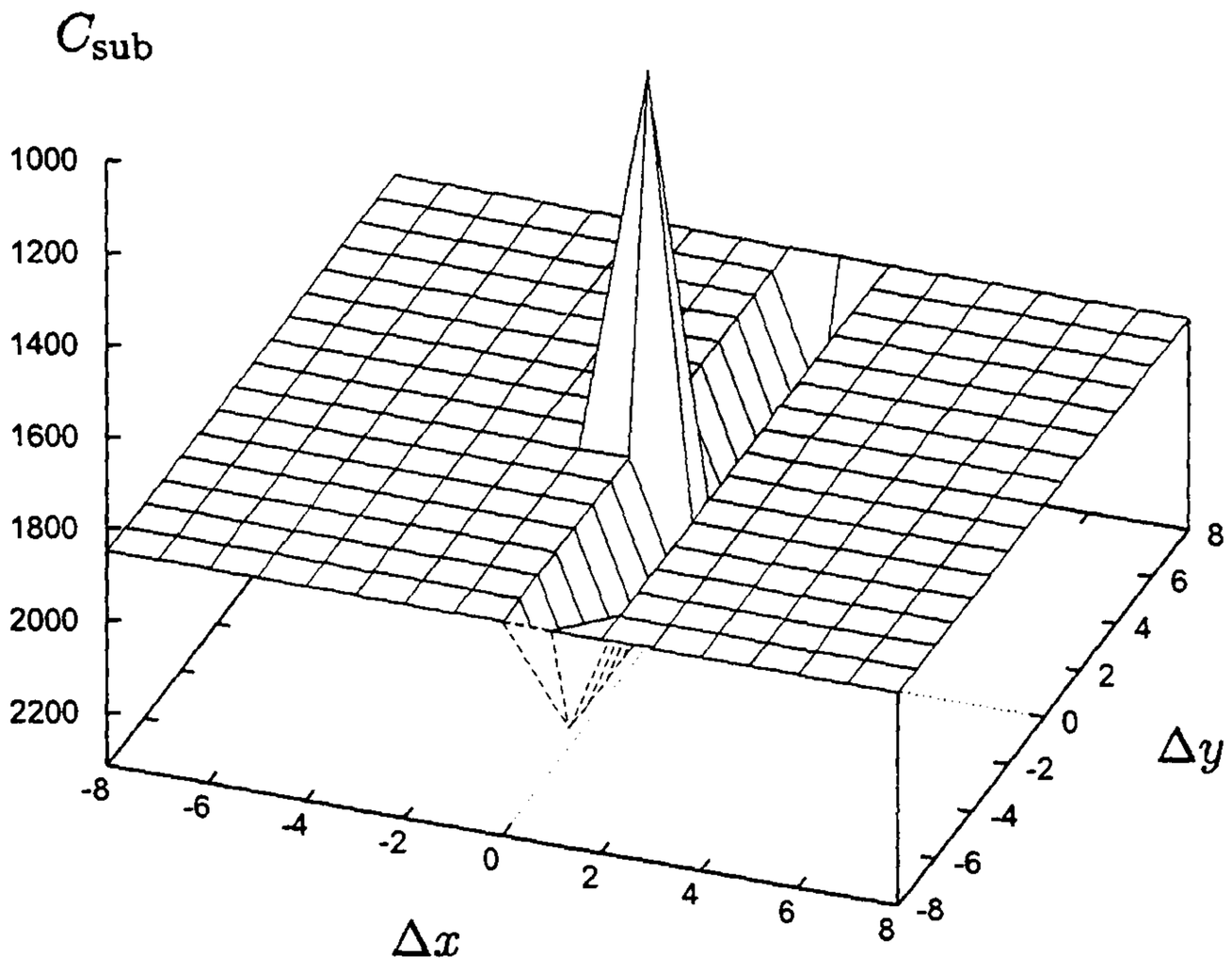


Figure 2.12: Subtractor module configuration latency C_{sub} as a function of the offset $(\Delta x, \Delta y)$ against the adder module (serial column-access configuration interface).

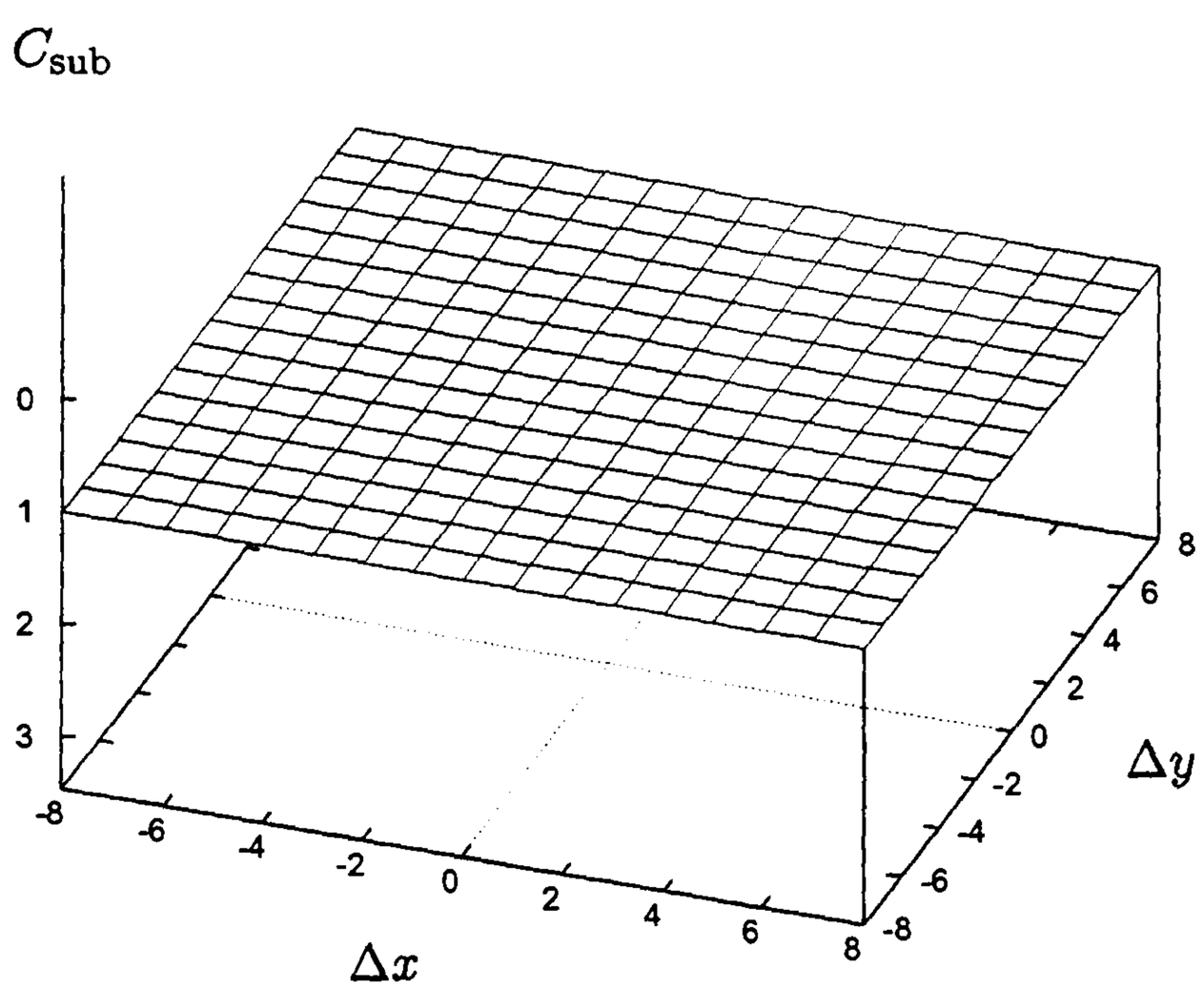


Figure 2.13: Subtractor module configuration latency C_{sub} as a function of the offset $(\Delta x, \Delta y)$ against the adder module (multiple-context configuration memory pre-loaded with the configuration data).

The above examples demonstrate that the reconfiguration latency varies greatly with the type of reconfiguration sub-system. In the simplest case, the reconfiguration latency is constant and does not depend on the module position or previous contents of the configuration memory (Fig. 2.13). Therefore the reconfiguration latency can be calculated accurately at a high-level.

In the most complex case, the reconfiguration latency is a function of module position and the previous contents of the device configuration memory (Fig. 2.8). The reconfiguration latency function becomes a complex non-linear function, which can be accurately determined only after physical synthesis has been completed. The corollary is that simple reconfiguration latency estimating techniques used prior to accurate knowledge of the physical design layout can be seriously misleading.

The availability of alternative configuration sub-systems has fuelled research into techniques allowing minimisation of the configuration latency for specific technologies. For example Hauck et al. (1998) and Shirazi et al. (1998) have investigated various such techniques for Xilinx XC6200 technology.

2.4 Summary

The approaches presented for the construction of the reconfiguration support circuitry provide tradeoffs between the reconfiguration speed and the size of reconfigurable array devoted to the reconfiguration circuitry. Reconfigurable technologies may combine these approaches to provide a balance suitable for a selected application domain.

For example, a Xilinx Virtex FPGA (Xilinx, 2000b) uses a serial configuration interface allowing partial reconfiguration. The configuration sub-

system provides an addressed access to configuration memory at coarse granularity (column-based). This solution provides a tradeoff between fast partial random-access to the configuration memory and its slow serial reconfiguration.

As many different approaches exist for the implementation of the configuration interface, configuration data distribution and activation, the speed of reconfiguration will depend greatly on the characteristics of each such implementation in a specific reconfigurable logic technology. In general, the reconfiguration latency is a non-linear and technology-dependent function.

Considering that in practical systems the period of the configuration cycle is often comparable to the period of the system clock cycle, the impact of the selected reconfiguration mechanism on the design execution latency will vary considerably with the *technology*, but for some technologies also with the module *placement*. This dependency on the low-level layout characteristics makes it very difficult to consider realistic configuration latency during reconfigurable system design. A solution to this problem is one of the principal results of this thesis.

Chapter 3

Previous Work on Reconfigurable System Design

Reconfigurable systems have evolved to bridge the gap between flexible processor-based programmable systems and high-performance systems with 'static' hardware functionality. Research in automatic reconfigurable systems design therefore has origins in two scientific disciplines: program compilation and high-level synthesis. This chapter examines the relevant work of researchers in both of these fields to date.

First the design process for non-reconfigurable systems is summarised in the following section in order to facilitate the comparison between the reconfigurable and non-reconfigurable design approaches.

3.1 Design for Non-Reconfigurable Systems

Design methodologies for non-reconfigurable systems have been the subject of active research since the invention of electronic circuits. Automatic synthesis of digital circuits from high-level descriptions became viable with

the development of numerous optimisation algorithms combined with design flows based on hierarchical design abstraction and hardware description languages (HDL).

3.1.1 Synthesis Design Flow

A typical design flow for non-reconfigurable systems is shown in Fig. 3.1. This is typically a hierarchical top-down process, composed of individual transformation and optimisation steps performed in a sequence.

During *behavioural* (also *architectural*) synthesis an abstract behavioural design model is translated into one of the possible architectural design models, while attempting to meet the design constraints. The architectural model is a collection of interconnected computational blocks and a system's controlling finite state machine(s) (FSM). This model is often referred to as *register-transfer level (RTL) architecture* to emphasise that at this level data transfers between each block's registers become visible.

In the following step, the architectural model is translated into a gate-level model. The gate-level model is a netlist capturing the connectivity between the instances of cells from the target technology library. This model is equivalent to a schematic diagram in a traditional schematic-based design flow. Whereas the behavioural and architectural models are independent of the target technology, the gate-level model is target-technology specific. The process of automatic transformation of an architectural design model into a gate-level model is known as *logic* or *RTL synthesis*. This process involves the mapping of all computational design blocks into their gate-level representation and the synthesis and optimisation of the design FSMs.

The design flow is completed by mapping the gate-level design model

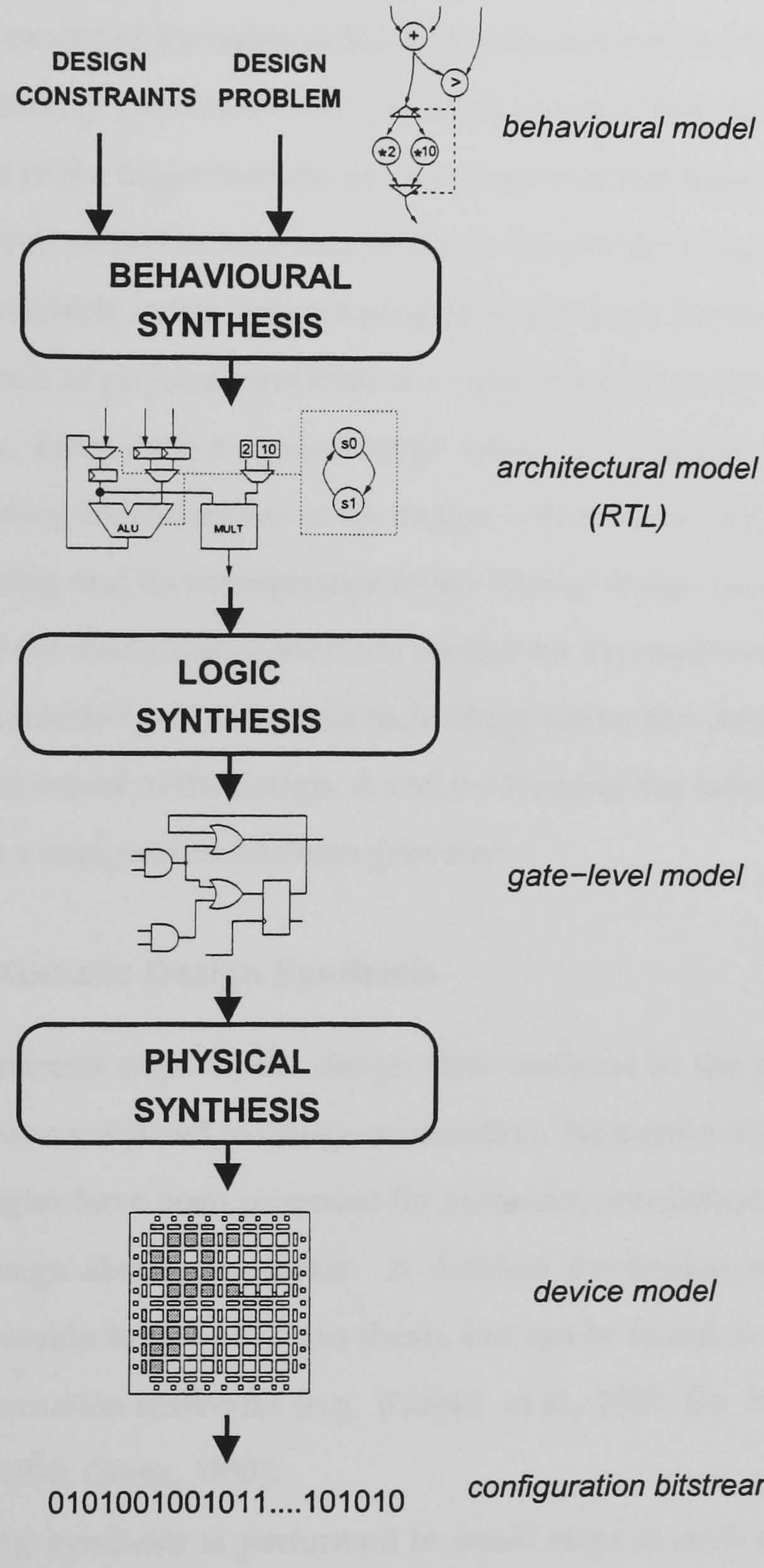


Figure 3.1: A typical design flow for non-reconfigurable systems.

onto a target technology physical model. This process is known as the *placement & routing (P&R) stage* or *physical* or *layout synthesis*, and normally involves placement of the target technology cells and their routing within the target technology physical model. The exact nature of this process depends on the type of the target technology. For programmable logic technologies, the gate-level model has to be mapped onto the primitive logic and routing elements available in the targeted programmable logic device.

The result of physical synthesis is a design implemented in the target technology. From such a *physical design model*, it is possible to extract the detailed timing characteristics of the design, which can be used to verify the overall timing and its conformance to the timing design constraints. The contents of the configuration memory needed for the implementation of the design in a selected programmable technology can be also determined from the physical model of the design. A tool performing this task is commonly denoted as a *configuration bitstream generator*.

3.1.2 Automatic Design Synthesis

All the synthesis steps of the design flow outlined in the previous section have been subjected to design automation. Numerous algorithms and methodologies have been proposed for automatic translation between the various design abstraction levels. A detailed description of these algorithms is outside the scope of this thesis, but can be found in several VLSI design automation textbooks (e.g. (Gajski et al., 1992; De Micheli, 1994; Sherwani, 1995; Gerez, 1999)).

Typically, synthesis is performed in small steps at each of the design abstraction levels. Accurate design metrics, such as design size, power consumption, or detailed timing characteristics, are not known before the

design flow has been completed. However, these metrics can be estimated using a variety of techniques available at each abstraction level.

Such a design approach does not normally lead to the most optimal results. Often design iterations are necessary to accommodate strict timing constraints. However, in many cases the inefficiency of the automatic design techniques can be tolerated for a large part of the designed system. Automatic design techniques offer a productivity gain, which in many real-world situations out-weighs the design inefficiency for all but a very small proportion of a designed system.

3.2 Design for Reconfigurable Systems

With the introduction of dynamically reconfigurable systems, design techniques capable of supporting the dynamic operation of these systems became desirable. Initial efforts were focused on techniques known from non-reconfigurable system design with the expectation that these techniques could be adapted for reconfigurable systems. Later new approaches to the design for reconfigurable systems were proposed to address specific problems not present in the traditional non-reconfigurable design methodologies.

The design process for reconfigurable systems can also be seen as a task with synthesis steps identical to those of non-reconfigurable systems (Fig. 3.1). The difference is in the type of the intermediate results produced at various abstraction levels. During the design for reconfigurable systems, the goal is to partition the design model into temporal segments so that the set of input design constraints could be satisfied.

This problem of *temporal partitioning* is different to the problem of partitioning into multiple FPGA devices (spatial partitioning). While both prob-

lems address the partitioning of design computational and design storage elements, temporal partitioning must also consider the temporal relationships between the individual design partitions to ensure that no dependency violations or other conflicts occur during execution.

Temporal partitioning can be performed at either behavioural level, register-transfer level, or at gate-level. The previous work in the area of design techniques for reconfigurable systems, and specifically that which addresses the automatic design of dynamically reconfigurable systems, is summarised in the following sections.

3.2.1 Evolution of Design Methodologies for Reconfigurable Systems

The difficulties with the design of reconfigurable systems have been highlighted by the work of several researchers. Hadley and Hutchings (1995) have described a manual design methodology for partially reconfigurable systems, noting the difficulties of using the conventional tools designed for non-reconfigurable systems.

The DISC system (Wirthlin and Hutchings, 1995) used a library of custom 'instructions' created using the standard FPGA CAD tools. The instructions were required to align with a dedicated communication and control architecture provided by the DISC system. The encapsulation of units of computations in such well-characterised instructions have allowed for dynamic instruction reconfiguration and placement to be managed during run-time.

The approach has been generalised by Brebner (1997), who introduced the Swappable Logic Unit (SLU) as a new computing paradigm to support dynamic reconfiguration and placement in reconfigurable computer sys-

tems.

This 'library-based' approach has been further advocated in the reconfigurable systems design methodologies in order to reduce the difficulties in designing reconfigurable systems (e.g. (Luk et al., 1996; Luk et al., 1997b)).

In such methodologies, a low-level library of target-technology modules is provided as a part of the design flow. Using these modules, which were pre-placed and pre-routed using the target reconfigurable technology, it is possible to estimate both the computational performance and worst-case configuration latency of the system with a good degree of accuracy for many reconfigurable technologies.

However, for many partially reconfigurable technologies, the actual reconfiguration latency is a non-linear function of module size, shape and the previous content of the configuration array (e.g. Xilinx XC6200 or Atmel AT6000 FPGA families), which is difficult to estimate even with the library-based approach.

For example, consider the experiment from Section 2.3. While the worst-case configuration latency for a 4-bit subtractor module configured via the 8-bit random access interface is 67 configuration cycles (Table A.3 in Appendix A), in the best case the latency reduces to only 15 (Fig. 2.8). This is a considerable reduction of 78%, which can be determined only after the place & route stage. These difficulties are forcing designers to iterate through the entire design flow several times in order to quantify the reconfiguration latency.

Luk et al. (1997b) and Robinson et al. (1998) reported design methodologies, which attempt to find the individual temporal design partitions in a sequential and iterative design process. Govindarajan and Vemuri (2000)

describe SPARCS—a system capable of performing both temporal and spatial (multi-FPGA) partitioning within one design flow.

These sequential design methodologies suffer from their inability to consider the strong interdependencies between the design decisions at high and low levels. For all but the simplest designs, these problems will lead to an excessive number of design iterations, which in turn will require several passes through the physical design tools.

Automatic temporal partitioning has been shown to be possible at various abstraction levels, although the quality of such partitioning varies depending on the applied method. The following sections summarise the past achievements in automatic temporal partitioning and run-time floorplanning for reconfigurable systems.

3.2.2 Partitioning at Behavioural Level

Conceptually this process is depicted in Fig. 3.2. Starting from a behavioural design model and a set of constraints, the temporal partitioning is performed directly on the behavioural model. As the partitioning is performed at high-level, it is possible to explore the tradeoffs between the problem implementations using different architectural options.

The product of such a temporal partitioning after the behavioural synthesis is a set of reconfigurable system partitions and a configuration controller for the design.

Automatic techniques which fall in the category of behavioural level temporal partitioning have been reported by several authors. This work includes methods based on heuristics and exact combinatorial optimisation.

Ling and Amano (1993a) have implemented a priority-list scheduling

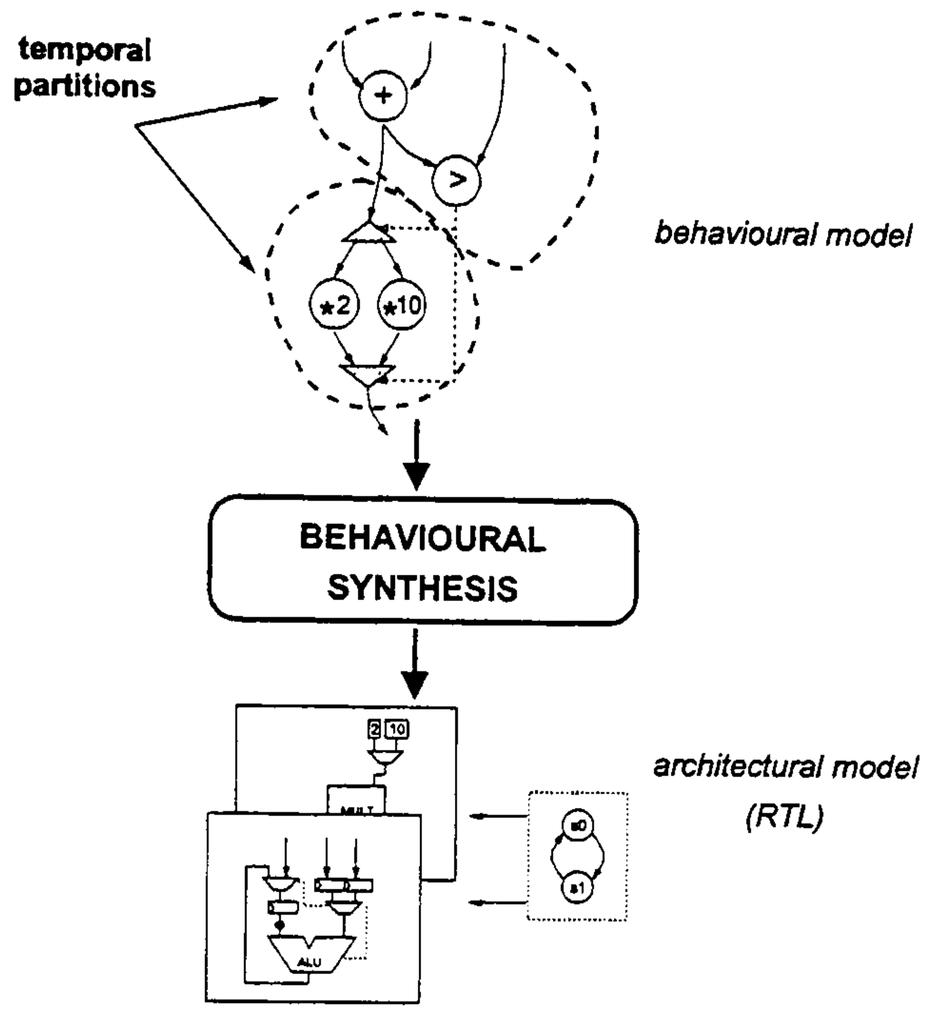


Figure 3.2: Temporal partitioning at behavioural level.

based technique for partitioning data-flow problems into multiple configurations. The technique has been further improved by Takayama et al. (2000). These heuristic techniques specifically target the WASMII platform which offers multiple-context memory for the configuration storage. These techniques do not consider partial reconfiguration.

Gokhale and Marks (1995) have proposed a partitioning approach for reconfigurable computing platforms, where each program function is partitioned into a separate FSM. This allows large programs to be executed on a limited reconfigurable computing platform. Each such FSM was implemented on a single FPGA.

A list scheduling based synthesis technique was developed by Vasilko and Ait-Boudaoud (1996a), which allows synthesis for partially reconfigurable systems. The technique assumes a constant reconfiguration time, while a simple approach was used for the high-level estimation of the avail-

able reconfigurable device area. Both of these considerations limit the practicality of this technique with the current reconfigurable technologies. Furthermore, architectural-level sharing between the design modules is not considered in this approach.

GajjalaPurna and Bhatia (2000) have presented two different heuristic methods for temporal partitioning and scheduling data-flow graphs for reconfigurable computing. These heuristics partition an input design problem into a set of full partitions, while using a simple area metric to express the reconfigurable technology resource constraint. The two techniques provide tradeoff between maximum parallelism and minimum communications cost. A correction factor based on the FSM communication cost is devised in order to deal with the routability problems.

Sels (1996) has addressed the problem of temporal partitioning for a reconfigurable device using integer-linear programming (ILP). However, only a one-dimensional model of the reconfigurable system was used in order to simplify the ILP problem formulation.

Kaul and Vemuri (1998) have presented an ILP-based temporal partitioning technique which optimises the communication and memory bandwidth between the individual configurations. The technique has considered a target system with reconfigurable devices of fixed size, while no partial reconfiguration was considered.

More recently Zhang et al. (2000) have proposed a temporal partitioning technique based on Constraint Logic Programming, which permits the design modules to be shared between configurations.

The problem of temporal and spatial partitioning has been also considered in the SPARC system by Govindarajan and Vemuri (2000). The SPARC partitioning process does not consider device-level partial reconfiguration.

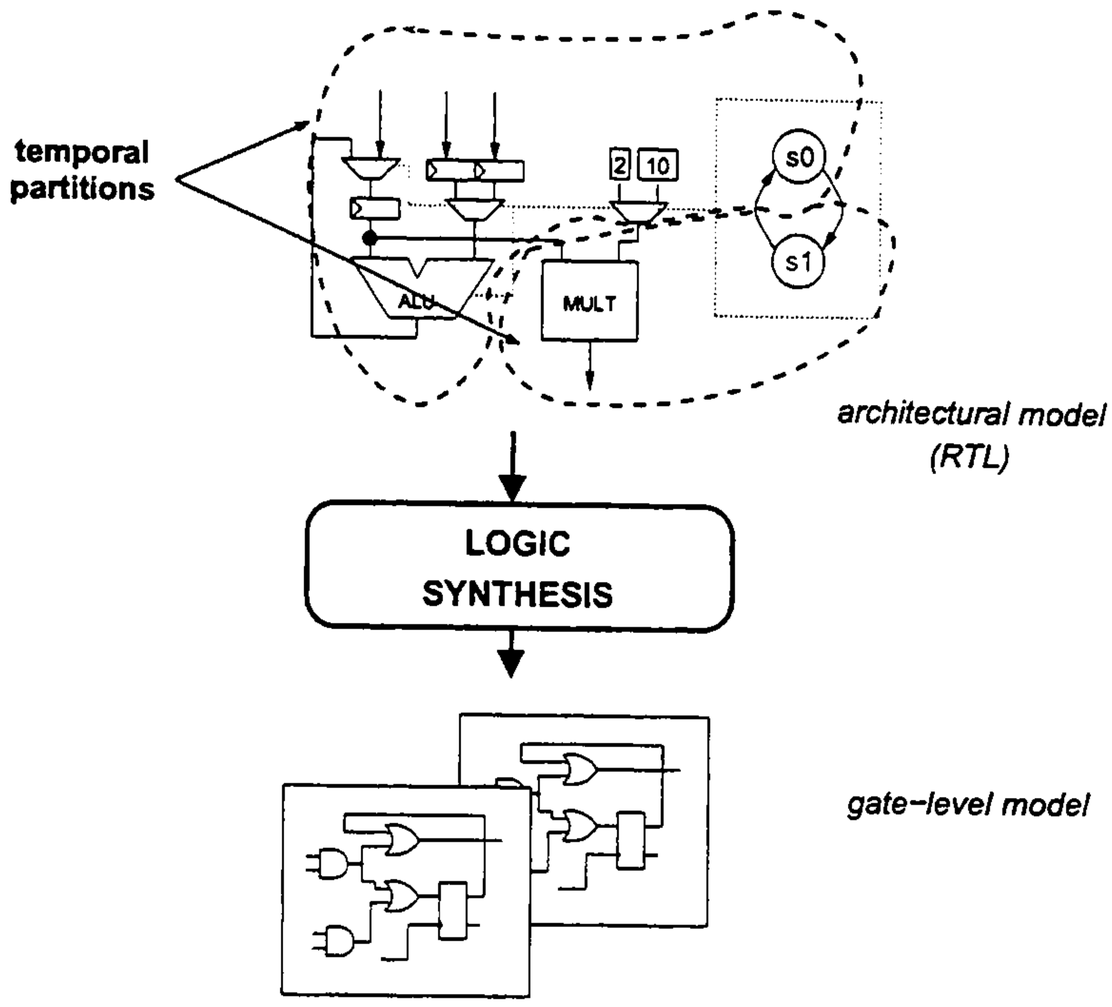


Figure 3.3: Temporal partitioning at RTL.

3.2.3 Partitioning at Register-Transfer Level

Conceptually this process is shown in Fig. 3.3. Given a non-reconfigurable register-transfer level architecture, the RTL temporal partitioning will produce a gate-level reconfigurable implementation. This approach explores the partitioning of both the design structure and its finite-state machine(s) for *one* RTL design architecture.

The RTL partitioning alone is unlikely to be a practical approach for synthesis of reconfigurable systems. This is because at RTL several design decision have been already fixed (including design module allocation, binding and scheduling) without any consideration for the design reconfiguration. While the RTL design architecture can be partitioned into separate configurations, the design optimisation is limited at this level to FSM optimisation/partitioning.

However, in a combination with other approaches (e.g. a reconfigurable system synthesis from behavioural level), the RTL partitioning could provide a valuable optimisation technique allowing optimisation of the design architecture and FSM partitioning at the same level.

While no previous work has been reported to date which directly addresses this problem, the use of reconfigurable finite-state machines was considered in (Skylarov and de Brito Ferrari, 1998; Oliveira et al., 1998).

3.2.4 Partitioning at Gate Level

Once a gate-level design model has been generated, it is not possible to modify the architecture or execution schedule of a design. Temporal partitioning at gate-level becomes attractive if the final gate-level model cannot fit into the target device. In such a case, one possibility is to 'fold' the implementation of the gate-level netlist over multiple design configurations (Fig. 3.4).

This approach has been shown to be beneficial in the field of logic emulation where hardware emulation resources are limited (Jones and Lewis, 1995; Trimberger et al., 1997). Both of these techniques are based on priority list scheduling heuristics operating on a gate-level design netlist.

Shirazi et al. (1998) describe an optimisation technique based on graph bi-partitioning, capable of optimising the layout in two configurations. The algorithm maximises the overlap of similar blocks in two configurations in order to minimise the overhead associated with the reconfiguration of the partitions.

Cantó et al. (1999) describe a heuristic gate-level bi-partitioning technique. The technique will split the gate-level circuit into two configurations, which can be mapped on the target device with two configuration

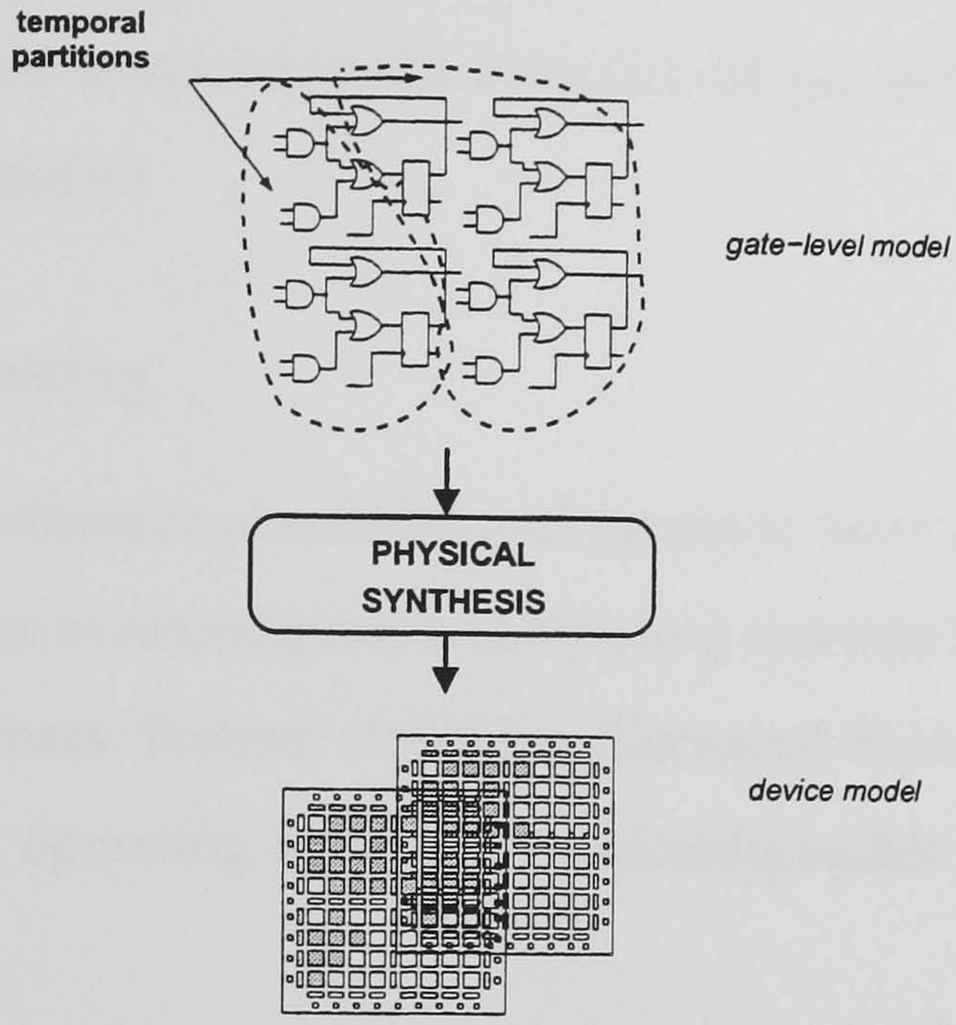


Figure 3.4: Temporal partitioning at gate level.

context layers.

Other authors have improved partitioning techniques at gate-level, including recent works by Liu and Wong (1999) and Chang and Marek-Sadowska (1998).

The advantage of gate-level partitioning is that the granularity of the design structure representation is close to that of the target architecture. It is therefore possible to estimate whether the design will fit into the target device and predict how successful routing will be.

The limitation of this approach is that the architectural implementation of the system, as captured in the gate-level netlist, cannot be changed to match the properties of the target reconfigurable technology. The original gate-level netlist was produced while targeting a *non-reconfigurable* system under a set of constraints intended for non-reconfigurable design implementation. Whilst considering the properties of the target reconfigurable

system (such as reconfiguration latency dependencies or resource limitations), the gate-level temporal partitioning cannot guarantee that these constraints will be satisfied.

3.2.5 Floorplanning

The problem of automatic placement and dynamic rearrangement of computational modules in reconfigurable computing systems has been addressed by several researchers. Brebner (1996) has discussed SLUs and their use in the context of an operating system for a reconfigurable computing platform.

A method for automatic placement of SLUs in a 3D floorplan based on simulated annealing has been proposed by Bazargan et al. (1999). A model reconfigurable system is used to conduct their experiments, while assuming a constant reconfiguration latency for each of the configured blocks.

Diessel et al. (2000) have proposed three different scheduling techniques for dynamic placement and rearrangement of tasks in reconfigurable computer systems. In this approach, the partial reconfiguration latency is assumed to be a linear function of the module size.

3.3 Solution Feasibility

The above approaches provide useful methods for temporal partitioning at behavioural and gate levels. However, few of these techniques consider partial reconfiguration. Even if the partial reconfiguration is considered, its properties are modelled using simplified and inaccurate models. In most cases, the above techniques attempt to reduce the complexity of the search space by working with a very simplified model of the reconfigurable architecture, often over-simplifying the impact of reconfiguration, ignoring

partial reconfiguration and routing implications.

If a partitioning technique for this design problem is to produce practical results, it must use a realistic model of the target reconfigurable technology. Furthermore, the advanced features of the configuration interfaces which allow reduction of the configuration overheads must be considered as a part of the partitioning process. For example, Luk et al. (1997b) have demonstrated that the latency required for reconfiguration of a 32-bit adder to become a 32-bit subtractor can be reduced 8-fold (from 32 to 4 configuration cycles) if the wildcard feature of the target XC6200 FPGA technology is used.

A further difficulty for the design of dynamic systems is that there are tight temporal and spatial interdependencies between the design entities at all abstraction levels. For example, a small modification of a floorplan at the physical level may cause a violation of the behavioural data dependencies due to an increased configuration latency. The interdependencies between the individual design problems for reconfigurable systems are further discussed when a formal model for this design process is introduced in Chapter 4.

From the above discussion it is apparent that the design of reconfigurable systems is much more difficult that that of non-reconfigurable systems. Indeed, non-reconfigurable system design can be perceived as a special case of reconfigurable system design, but where the entire design solution can be fitted into one configuration.

3.3.1 Synthesis for Full versus Partial Reconfiguration

The design of reconfigurable systems is further complicated by the variety of available reconfigurable technologies. Section 2.2 discussed the various

approaches for the implementation of reconfiguration sub-systems (partial vs full reconfiguration, random-access vs serial distribution mechanisms, various configuration activation techniques) and the resulting tradeoffs.

When temporal partitioning is performed for a technology offering full reconfiguration only, the goal of the partitioning is to split an input problem model into a number of configuration 'pages'. Each such page must fit within the resources available in the targeted reconfigurable device. Traditional design techniques used for non-reconfigurable systems can often be used to synthesise each such configuration page.

While these techniques cannot guarantee that each page can be fully placed and routed for the targeted reconfigurable device, simple methods can be used to avoid such problems. For example, temporal partition techniques may be permitted to use only a proportion of the total resources available in order to provide redundancy in case of placement or routing difficulties (e.g. used in (Govindarajan and Vemuri, 2000) or (Takayama et al., 2000)).

If the latency of full reconfiguration is prohibitive, a technology supporting partial reconfiguration would be a preferred option.

Temporal partitioning for partially reconfigurable technologies is a much harder problem. Additional constraints and features must be considered, such as ensuring that partially reconfigured modules do not overlap, whether the state of device flip-flops can be shared between configurations, the position of the design blocks to minimise the configuration overheads, and others. The design optimisation is further complicated by the fact that in many partially reconfigurable technologies, the partial reconfiguration latency is a non-linear function of the position of a design module and the previous contents of the configuration memory (see Section 2.3 for further

discussion on the reconfiguration latency function).

3.4 Summary

Various technological approaches have been developed for the design of reconfigurable systems offering tradeoffs between device area, reconfiguration speed and other design metrics. The reconfiguration performance of reconfigurable systems is dependent on a specific technology and is determined by the speed of its reconfiguration sub-system.

However, the review of the published work to date reveals that there is not yet a solution to the problem of automatic design synthesis which can *reliably* exploit the features of partially and dynamically reconfigurable logic systems.

The design of reconfigurable systems differs from the design for non-reconfigurable systems, because it must consider the temporal partitioning of the input design problem and technological dependencies of the design metrics associated with the selected target technology. Partially reconfigurable systems offer many technological advantages over fully reconfigurable systems. However, their features together with placement-dependent reconfiguration latency further complicates the design process.

Given the availability of various reconfigurable logic technologies, it is important that a synthesis methodology for reconfigurable systems considers the *dependency* of the reconfiguration latency function at a high level. This thesis present an example of one such approach to reconfigurable system synthesis.

Chapter 4

Reconfigurable System Synthesis Problem Formulation

High-level synthesis is a multiple-level transformation process involving translation of an initial design problem represented by a behavioural model into a design implementation model, while considering both design performance constraints and the constraints imposed by the selected target technology.

The success of a solution search for any optimisation problem is determined by the qualities of the model characterising the problem. The quality of any such model is measured by its ability to capture the fundamental problem characteristics, while simplifying or neglecting the factors, which have only a minor or no contribution to the success of the solution search.

This chapter presents a new formulation of the problem of synthesis for reconfigurable systems. The formulation captures the low-level technology-dependent characteristics and can therefore guarantee the feasibility of a generated solution, while permitting the optimisation algorithm to explore the solution search space efficiently.

The following section summarises the initial assumptions about the reconfigurable system synthesis problem and models used in its formulation. The formulation of the synthesis problem for reconfigurable systems is provided in the reminder of this chapter. The formulation highlights the impact of the technology-dependent design characteristics on the synthesis process.

4.1 Fundamental Assumptions

Various techniques can be used to implement circuits using reconfigurable logic technologies, working in various operational modes. These range from systems implemented without any reconfiguration (non-reconfigurable systems), systems which will reconfigure more or less frequently, systems with a separate configuration control, self-repairing, self-reproducing or self-reconfiguring circuits, circuits constructed and controlled via simulated run-time natural evolution and possibly many others.

This section summarises the fundamental assumptions made about the problem of high-level synthesis for reconfigurable systems. These assumption also characterise the target computational model for reconfigurable systems considered in this thesis. While this formulation is restrictive and does not cover all possible applications of reconfigurable logic, the presented formulation is feasible for many reconfigurable systems of practical interest.

4.1.1 Input to Reconfigurable System Synthesis

The process of reconfigurable system synthesis considered here begins with a behavioural *problem model* and a set of *design constraints*. However, real-world design projects rarely start at this level. This level of abstraction

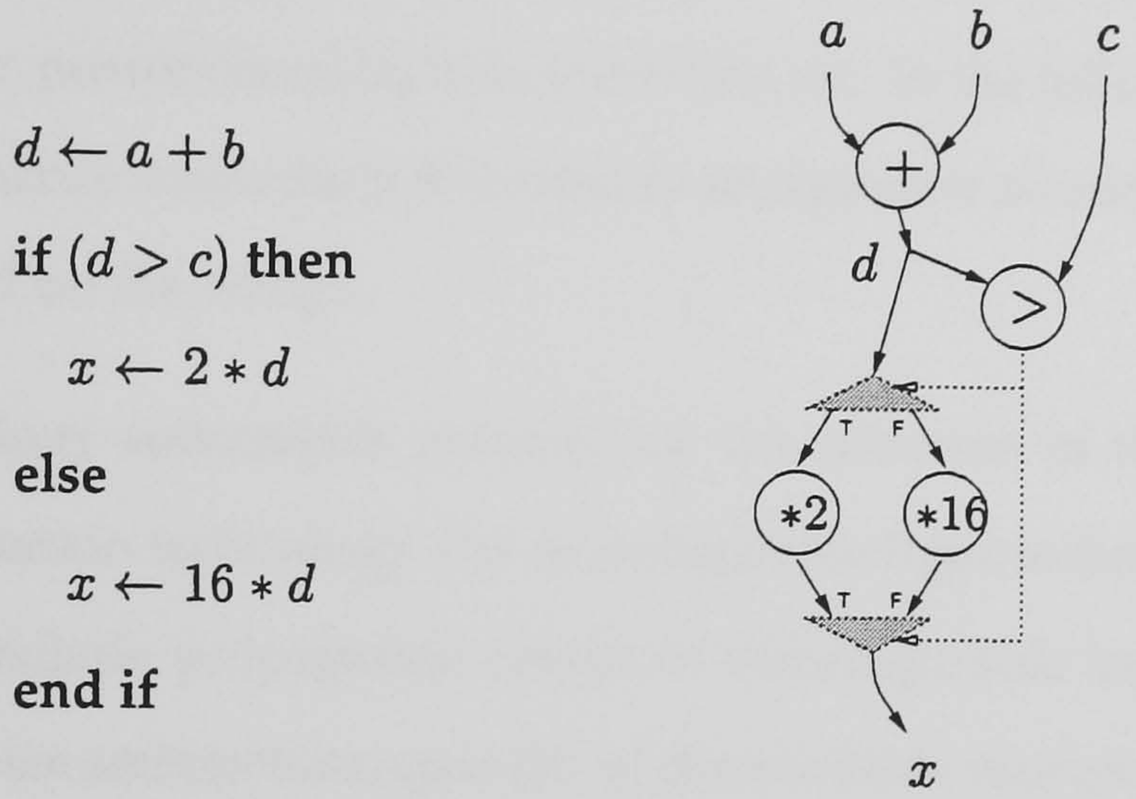
is normally preceded by a design problem analysis leading to a formulation of a system specification. From the system specification it is possible to derive a system-level model, which can be used to establish and verify the desired system functionality at this level. Once the system-level functionality has been established, a behavioural design model together with design constraints can be extracted in a suitable form. If the system is to be implemented on a heterogeneous platform, including hardware, software, reconfigurable hardware, etc., this step has to be preceded by system-level partitioning. This problem is not considered here.

4.1.1.1 Design Problem Model

A variety of different design models have been developed in the field of high-level synthesis research. A Control/Data Flow Graph (CDFG) model (developed by McFarland et al. (1990) and others) was selected here to represent the design behaviour. This choice was motivated by the model's ability to capture both data and control characteristics of an input design problem in a single design model. The CDFG model is popular in high-level synthesis tools because of this ability. Similar models capturing both control and data characteristics of an input design problem could have been used as alternatives.

Definition 4.1.1 (Control/Data Flow Graph) A CDFG is a directed graph $G(V, E)$, where the set of vertices V represents a set of operations and the set of edges E represent dependencies between the pairs of operations.

The vertex set V can be decomposed into a set of data vertices V_d representing behavioural data operations (multiplication, subtraction, increment, etc.) and the set of control vertices V_c representing control



(a) code fragment

(b) CDFG

Figure 4.1: Example of a Control/Data Flow Graph model with the corresponding behavioural code fragment.

operations (control flow fork/join), where $V_d \cup V_c = V$. Similarly, the edge set E can be decomposed into a set of data edges (those carrying data token values) E_d and control edges (carrying control tokens) E_c, where $E_d \cup E_c = E$.

In the following, where the distinction between V and E is not important, the set $B = V \cup E$ is used to denote *all* elements of a CDFG model.

An example of a CDFG model and its corresponding behavioural procedure is shown in Fig. 4.1.

4.1.1.2 Design Constraints

Design constraints restrict the set of possible design implementations to a set of solutions which can accommodate them. Design constraints fall into two categories:

- **Performance constraints** which represent bounds on the desired per-

formance of the implementation, such as execution latency, throughput, size, power consumption, testability, etc. In the following, a set of performance constraints Ψ is used to encapsulate all such constraints imposed on the design.

- **Technology constraints** enforced by the selection of the target implementation technology. For reconfigurable logic technologies these might include propagation delays of reconfigurable logic and routing, device architecture, quantity of the available resources, limitation on module placement, throughput and capabilities of the reconfiguration interface, etc. A set of technology constraints is referred to as Θ.

4.1.2 Design Goal

It is assumed that the goal of reconfigurable system synthesis is to construct a design implementation (also called *design solution*) using the selected target technology, such that all design constraints are satisfied. It is further assumed that the aim is to implement the *entire* input design model using the selected reconfigurable logic technology. The problem of hardware/software partitioning for the input design problem is not considered here.

There is no *a priori* assumption that the design problem is to be implemented using dynamic reconfiguration. Given the set of design constraints and the selection of the target technology, the synthesis process may result in either a reconfigurable or a non-reconfigurable design implementation.

4.1.3 Target Architectural Model

The target architecture for the reconfigurable system synthesis problem considered is the architecture discussed in Section 2.1. It is assumed that

each of the main architectural components is represented by a single device (spatial partitioning between multiple reconfigurable devices, controllers or memories is not considered). No initial assumptions are made about the architecture and capabilities of the targeted reconfigurable logic technology.

4.2 Reconfigurable System Design Synthesis Transformations

This section presents a formal framework for the definition of design problems in reconfigurable system design. The definitions presented here are derived from the traditional definitions of high-level synthesis problems for non-reconfigurable systems (Gajski et al., 1992; De Micheli, 1994; Gerez, 1999), whilst new or modified definitions are provided for problems which exist in the synthesis of reconfigurable systems. The aim is to provide a macroscopic view of the entire synthesis process, while highlighting a new formulation of problems resulting from the use of reconfigurable systems. The following formulations do not make any assumptions about whether the synthesis is performed during *compile-time* or *run-time*, rather they provide a formalism for the problems which need to be addressed by either of these two approaches.

A design solution is constructed by finding associations between the elements of a design model, library and target technology device elements at several abstraction levels. This process is illustrated in Fig. 4.2.

There are number of individual transformation tasks (problems) which need to be solved during the reconfigurable system synthesis. The individual problems are inherently interdependent. In the following, the individ-

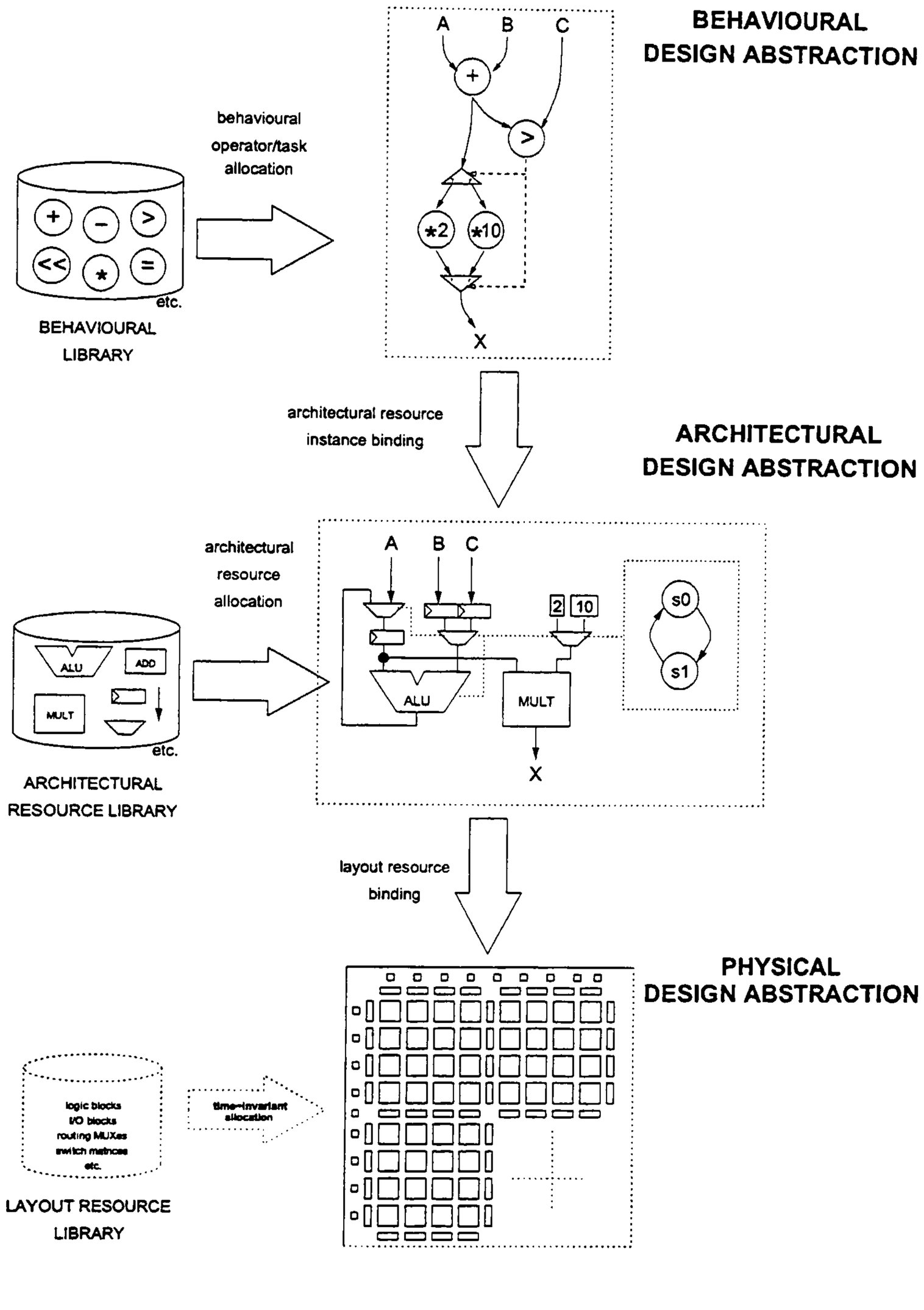


Figure 4.2: Transformation of a reconfigurable design during synthesis.

ual transformations are described in a hierarchical order.

4.2.1 Behavioural → Architectural Level

During transformation from a behavioural to an architectural abstraction level it is necessary to perform the operations of resource allocation, resource binding and scheduling.

4.2.1.1 Architectural resource allocation

This selects a set of resources from the resource types available in the target technology library at the architectural level. This selection must ensure that resources providing the implementation for all types of behavioural model elements are available. In the following, the term 'architectural resource library' is used to refer to a collection of (parametrisable) functional resources available for the target technology at architectural level (Fig. 4.2).

Resource allocation involves allocation of library computational resources for behavioural model vertices (e.g. multipliers, subtractors, comparators, etc.) and allocation of library connectivity resources for behavioural model edges (e.g. buses, permanent and 'virtual' registers, register files, FIFOs and other memory elements, etc.).

Definition 4.2.1 (Resource allocation problem) Given a behavioural input model $G(V, E)$ with a set of behavioural model elements B and a set of resource types R from the architectural resource library, find a set of architectural resource instances A such that the following condition is satisfied for all $b \in B$ (resource availability):

$$\exists a \in A\,.\;\; F(b) \subseteq F(a) \tag{4.1}$$

where $F(b)$ and $F(a)$ represent the set of operations performed by

the behavioural element b and the set of operations performed by the architectural resource instance a respectively.

The result of resource allocation can be characterised by mapping $\alpha : A \rightarrow R$, which represents associations between all architectural resource instances and their corresponding library resource types.

The technology-specific architectural resource library provides resources specific to the features of the targeted technology. For example, if a reconfigurable technology permits the transfer of register states via the configuration interface, the architectural library will provide a 'virtual register'[1] resource available for allocation to behavioural model edges.

4.2.1.2 Architectural resource binding

Architectural resource binding creates a mapping between the set of architectural resource instances and the behavioural model elements. The problem of resource binding at architectural level can be defined as follows:

Definition 4.2.2 (Architectural resource binding problem) Given an input behavioural model $G(V, E)$, its set of behavioural elements B, and the set of resource instances A find mapping $\beta : B \rightarrow A$ such that the following condition is satisfied (functional compatibility):

$$\forall b \in B\,.\;\; F(b) \subseteq F(\beta(b)) \tag{4.2}$$

Once both vertices and edges from the behavioural model have been allocated and bound to specific architectural instances, it is possible to determine the implementation characteristics associated with the behaviou-

[1] *Virtual register* is a register, whose state is transferred via the configuration interface to/from an external memory storage (CDS in Fig. 2.1). Virtual registers are used in reconfigurable systems to transfer the values stored in hardware registers between either different configurations or different modules/ports in one configuration. Such registers can be also used in 'virtual pipelines' (Luk et al., 1997c).

ral model elements (e.g. latency, pipeline stages, area, signal quantisation, buffering, etc.), while other characteristics can be only quantified in the later stages.

The resource allocation may result in some of the architectural resources in the set A to be *shared* between the elements of the behavioural model $G(V, E)$:

Definition 4.2.3 (Architectural-level resource sharing) The resource sharing at architectural level occurs when given the behavioural model $G(V, E)$, the set of its behavioural model elements B and the resource binding mapping β, the following condition is satisfied (architectural-level resource sharing):

$$\exists b \neq b' \,.\; \beta(b) = \beta(b') \tag{4.3}$$

where $\beta(b) = \beta(b') \in A$ is the shared resource.

4.2.1.3 Scheduling

Scheduling can be performed once the timing characteristics of architectural resources associated with the behavioural model elements can be estimated. Scheduling can be performed in a variety of scenarios, depending on the type of the design constraints. In order for a design implementation to be *feasible*, the behavioural model has to be scheduled such that no violations of data or control dependencies occur.

In the following, the integer delay function $Delay(\beta(v_j))$ represents the latency[2] of the resource(s) bound to v_j and the token transport latencies from the edges connecting from v_j to v_i, where v_j is a predecessor to v_i.

The integer setup function $Setup(\beta(v_i))$ represents the latency required to setup the resource bound to v_i. This may include configuration of the

[2]the integer latency represents the latency relative to the system control step period

library resource instances and associated routing resources, and other tasks which need to be completed before the operation v_i can be executed.

A general constrained scheduling problem for reconfigurable systems can be then formulated as follows:

Definition 4.2.4 (General constrained scheduling problem) Given a set of operations V and a partial order on operations E, find an integer labeling of operations $\sigma, \rho : V \rightarrow \mathbb{Z}^+$, representing a design *execution schedule* σ and a design *configuration schedule* ρ, while the following condition is satisfied for all i, j such that $(v_j, v_i) \in E$ (schedule feasibility condition):

$$\sigma(v_i) \geq \sigma(v_j) + Delay(\beta(v_j)) \quad \wedge \quad \sigma(v_i) \geq \rho(v_i) + Setup(\beta(v_i)) \quad (4.4)$$

and the set of performance constraints Ψ is satisfied.

The execution schedule time $\sigma(v_i)$ represents the execution start time of the resource bound to the behavioural operation v_i, while $\sigma(v_j)$ represents the execution start time of the resource bound to v_j.

The configuration schedule time $\rho(v_i)$ represents the configuration start time of the resource bound to v_i.

An example of a feasible schedule for the design behavioural model $G(V, E)$ is shown in Fig. 4.3.

The integer labels in σ, ρ represent system 'control steps', which usually correspond to the system clock cycles. Thus labeling ρ represents the reconfiguration schedule in terms of system time units. The detailed timing of the reconfiguration process can be controlled by a separate synchronisation mechanism.

In the case when $Setup(\beta(v_i)) = 0$, there is either no need to setup the resource associated with the operation v_i or the contribution of the setup

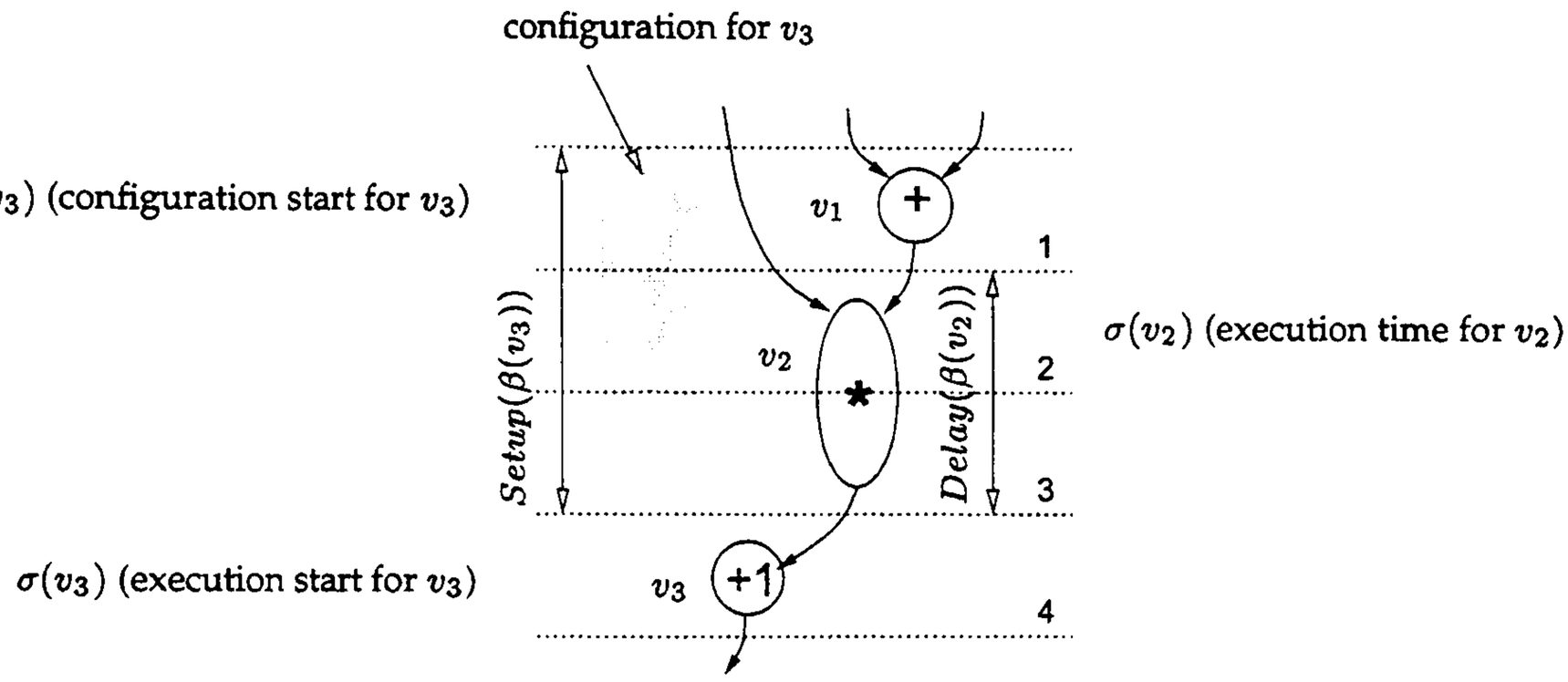


Figure 4.3: Example of a feasible schedule in a reconfigurable system. Configuration latency for v_3 resources (represented by $Setup(\beta(v_3))$) has no temporal impact on the design execution latency as the configuration is performed in parallel with execution of operations v_1 and v_2.

latency is considered as a part of the system clock cycle.

For this special case, the Eq. 4.4 becomes:

$$\sigma(v_i) \geq \sigma(v_j) + Delay(\beta(v_j)) \quad \wedge \quad \sigma(v_i) \geq \rho(v_i) \tag{4.5}$$

As was demonstrated in Section 2.3, the reconfiguration latency is a technology and design-dependent function, which may vary greatly with the physical design characteristics (e.g. design module floorplan position or overlap). In traditional design approaches, no physical design characteristics are available at the time of scheduling. If for a selected target technology the actual *Setup*() function is not known at the time of scheduling, then only an *estimate* of the reconfiguration latency can be made. It is therefore difficult to ensure that the schedule feasibility condition (Eq. 4.4) is satisfied at this stage.

The result of behavioural synthesis for a specific set of performance constraints Ψ is a 4-tuple (A, β, σ, ρ), representing the design architectural

model with the selection and binding of its architectural elements, and their execution and reconfiguration schedules. The target technology in this process is represented by the architectural resource library providing the set of resource types R and their $Setup()$ functions.

4.2.2 Architectural → Physical Level

The transformation process from architectural to physical level has to translate the architectural abstract model into a physical design model, which can be mapped onto the target technology device. The following problems need to be solved during this process.

4.2.2.1 Logic synthesis

This translates architectural model elements into a set of connected, technology-specific primitive cells representing a technology cell netlist. The following is a macroscopic definition of the logic synthesis problem for reconfigurable systems.

> **Definition 4.2.5 (Logic synthesis problem)** Given an architectural design model (A, β, σ, ρ) find a design logic model (U, N) representing the set of target technology primitive logic blocks U and nets N such that the model (U, N) preserves the functionality of architecture A, execution schedule σ and reconfiguration schedule ρ of the architectural model.
>
> The process of logic synthesis may involve the following tasks:
>
> - synthesis and optimisation of the logic representation for each element of the architecture A (if the logic representation of architectural elements in the target technology is not known)

- synthesis and optimisation of the design state-machines which implement the schedule defined by σ
- synthesis and optimisation of the configuration controller automata implementing the schedule ρ. Configuration controller synthesis will produce a cycle-accurate reconfiguration schedule ρ_c, indicating the exact activity of the reconfiguration interface in each configuration clock cycle

A description of the individual tasks involved in logic synthesis is outside the scope of this thesis. Details can be found in the logic synthesis literature (e.g. (De Micheli, 1994; Murgai et al., 1995; Gerez, 1999)).

The reconfiguration controller can be implemented by a dedicated logic circuit using traditional finite-state machine synthesis methods, or its operation can be provided by a processor-based system which implements the reconfiguration schedule ρ_c. The implementation of dedicated reconfiguration controller circuits has been studied by Robinson and Lysaght (1999) and others.

Once the design logic model is obtained, the netlist elements can be mapped onto a target technology device.

4.2.2.2 Physical synthesis

This involves finding a solution to both placement and routing problems. In the case of reconfigurable systems this can be generally defined as follows:

Definition 4.2.6 (Physical synthesis problem) Given the design logic model (U, N) with the set of logic model elements $L = U \cup N$, the target technology device with a limited set of resources D, the design execution schedule σ and cycle-true configuration schedule ρ_c, find

the mapping $\phi : L \rightarrow D$ such that the functionality of the logic model L is preserved and the schedule feasibility condition (Eq. 4.4) is not violated.

If the design implementation is *non-reconfigurable* then the device resources are not shared between the elements of the logic model:

$$\forall i \neq j \,.\; \phi(l_i) \neq \phi(l_j) \tag{4.6}$$

For a reconfigurable design implementation it is possible to share the device resources between the elements of the logic model (physical resource sharing):

$$\exists i \neq j \,.\; \phi(l_i) = \phi(l_j) \tag{4.7}$$

Once the mapping between the design logic model and the physical device is known, it is possible to construct a configuration controller implementing the cycle-accurate reconfiguration schedule ρ_c for the design.

4.2.2.3 Solution feasibility

The feasibility of the final design solution is defined as follows:

> **Definition 4.2.7 (Solution feasibility)** The design solution is feasible if and only if it can be implemented on the target technology device resources D with behavioural functionality identical to that of an input behavioural model $G(V, E)$, while the schedule feasibility condition (Eq. 4.4) is not violated.

In the above definition, the solution feasibility itself does not imply that the design solution meets all of the performance constraints Ψ.

The result of physical synthesis for the target device is a 4-tuple $(L, \phi, \sigma, \rho_c)$. This physical design model can be analysed for physical timing character-

istics and used to generate device configuration data necessary for the implementation of the design functionality on the device resources D.

4.3 Comparison with a Traditional High-Level Synthesis Formulation

This section compares the problem formulation introduced in Section 4.2 with the traditional formulation of high-level synthesis for non-reconfigurable systems (e.g. (Gajski et al., 1992; De Micheli, 1994; Gerez, 1999)).

The main differences between the two formulations are as follows:

- The scheduling problem (Definition 4.2.4) for reconfigurable systems considers the latency required to setup the architectural resources bound to their respective behavioural model elements. The schedule feasibility condition (Eq. 4.4) ensures that the data and control dependencies between the behavioural operations are not violated, and the architectural resources are setup before they can perform any computations. Furthermore, the function $Setup()$ expresses the interdependence between the design schedule (high-level characteristic) and the setup latency (typically a low-level technology-dependent characteristic).

The formulation of the scheduling problem for non-reconfigurable systems requires that only the data and control dependencies between the behavioural operations are not violated. When such a formulation is applied to reconfigurable system synthesis, the reconfiguration latency is *not* considered at the time of scheduling. Therefore the design schedule may be invalidated at a later stage in the design flow, when the target technology-specific reconfiguration latency is inserted into

the design schedule.

- The presented formulation of a physical synthesis problem (Definition 4.2.6) allows for a clear distinction to be made between synthesis for reconfigurable and non-reconfigurable systems (Eq. 4.7 versus Eq. 4.6). In the context of the presented formulation, the reconfiguration is viewed as an instance of a resource sharing problem; note similarities between the architectural resource sharing (Eq. 4.3) and device resource sharing (Eq. 4.7). This observation suggests that similar methods could be used to search for solutions to these problems, while working at different abstraction levels.

- Reconfigurable technologies may provide new possibilities of 'connecting' behavioural model operations V. For example, it might be possible to connect two operations using 'virtual registers' or using a pair of overlapping registers which share their contents[3]. The presented formulation allows for these special features to be considered as a normal part of the synthesis transformations for $B = V \cup E$ (Definitions 4.2.1–4.2.4). These special features are supported via the availability of specific connectivity resource types in R. Timing characteristics of such resources may be considered as a part of $Setup()$ and $Delay()$ functions.

In the traditional high-level synthesis formulation, the problems of resource allocation and binding are considered only for behavioural model operations V. This is because the implementation of behaviou-

[3]For a pair of *overlapping registers* the register state is transfered directly between the overlapping registers, i.e. without the need for the state to be transfered via the configuration interface to the external memory storage (as for virtual registers). For example by overlapping the count register between up-counter and down-counter configurations, it is possible for the counters in two different configurations to share their count values (Vasilko and Cabanis, 1999).

ral model edges E (i.e. wires, multiplexors and registers) at the later stage is assumed not to introduce a significant timing overhead into the design schedule.

- A separate reconfiguration schedule ρ_c was introduced in the presented formulation to capture the configuration cycle-true activity of the configuration interface. No equivalent of a reconfiguration schedule exists for non-reconfigurable systems.

The presented problem formulation generalises the problem of system synthesis. The synthesis problem for non-reconfigurable systems is viewed as a special case of this general formulation.

4.4 Summary of the Model Features

The following are the main features of the formulation presented for the problem of reconfigurable system synthesis as compared to other approaches to reconfigurable system synthesis discussed in Chapter 3:

- Resource sharing with architectural granularity only is considered during behavioural synthesis. This type of resource sharing is identical to that of non-reconfigurable behavioural synthesis, where for example one functional unit (e.g. ALU) can be shared between the behavioural operations which require that functionality (it must satisfy the condition of functional compatibility, Eq. 4.2).

- Resource sharing via reconfiguration is considered at a fine-grained physical level. This allows sharing of primitive physical components at the layout level. Therefore an optimisation algorithm working within this framework will be able to evaluate not only sharing be-

tween architectural elements with identical architectural functionality, but also sharing between unrelated architectural elements based on the *similarities* of their configurations.

- Contrary to other approaches discussed in Sections 3.2.2–3.2.4 no 'configuration partitioning' is performed during the behavioural → architectural translation. Rather the temporal dependencies resulting from reusing the architectural elements via reconfiguration are annotated as a part of the configuration schedule ρ.

A cycle-accurate configuration schedule is generated based on the mapping of the logic design model to the physical device model generated during physical synthesis. This schedule represents the set of reconfigurations performed at the granularity determined by the capabilities of the target technology reconfiguration sub-system.

- As there is no *a priori* assumption that the design should be implemented as a reconfigurable system, the optimisation techniques working within this framework are free to explore the tradeoffs between the reconfigurable and non-reconfigurable implementations. Therefore, the result of the synthesis is a solution which accommodates the input design constraints and which may be either a reconfigurable or a non-reconfigurable design implementation.

Chapter 5

DYNASTY Framework

This chapter presents DYNASTY—an experimental CAD framework developed as a part of this project to support the research on reconfigurable system design techniques. This Framework was used throughout the work presented in this thesis for the development of the presented algorithms, techniques, but also for the experimentation with the target technologies, new simulation techniques and methodologies.

The presentation of the Framework in this chapter is based on the author's previous publications from this project (Vasilko et al., 1999; Vasilko, 1999; Vasilko, 2000).

The following section provides the details about the Framework features and its implementation. In Section 5.2 a user's view of the DYNASTY design flow is presented, while Section 5.3 illustrates the Framework capabilities using a simple design example. The chapter concludes with the summary of the DYNASTY Framework's features in Section 5.4.

5.1 Introduction

DYNASTY Framework is an extensible generic CAD tool-suite, designed to support research of reconfigurable system design techniques and methodologies.

The Framework fully supports the problem formulation presented in Chapter 4. The design methodology currently implemented in the DYNASTY Framework is based around the *temporal floorplanning* (Vasilko, 1999), which allows *simultaneous* reconfigurable design space exploration at multiple levels of design abstraction in both spatial and temporal design dimensions.

The Framework implements several novel concepts, including temporal floorplanning, technology server[1] based design methodology and a variety of the design visualisation techniques allowing designers to interact with the design process throughout the entire design flow.

5.1.1 Architecture

The overall architecture of the DYNASTY Framework is shown in Fig. 5.1. The core of the Framework is the internal design representation providing design model view at (i) behavioural, (ii) architectural, and (iii) layout abstraction levels.

Design manipulation tools are provided to facilitate construction of the design solution. Design analysis tools help to evaluate the quality of the constructed solution. Design visualisation techniques provide visual feedback about the design structure, characteristics and performance. Interfaces to third party tools are provided to allow importing designs into the

[1]Some of author's previous publications use the term *library server* to refer to a technology server.

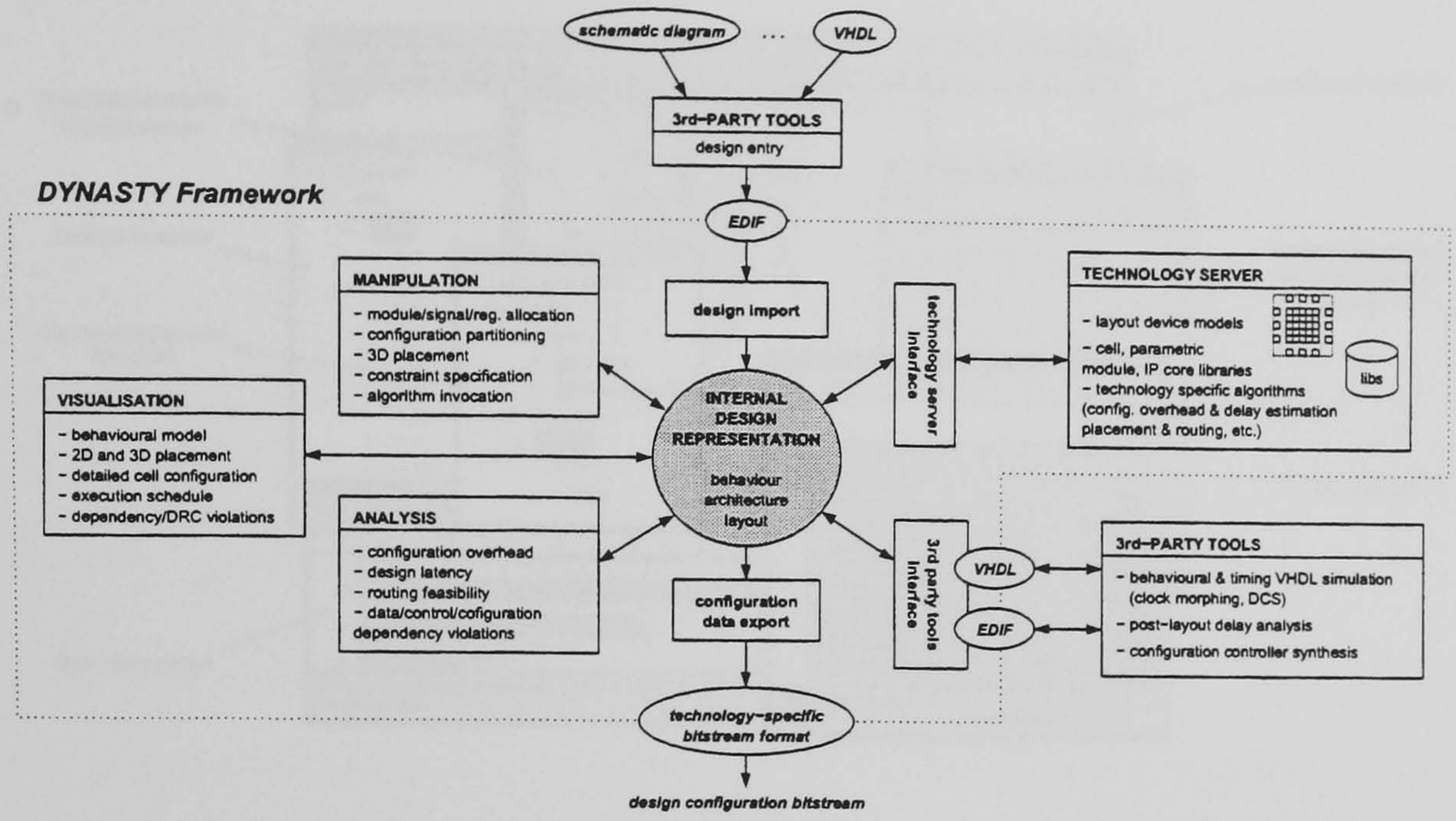


Figure 5.1: DYNASTY Framework architecture.

Framework, communication with a VHDL simulator and other technology-specific analysis tools.

A selectable technology server provides technology-specific libraries, device models and algorithms (estimation, placement & routing). Other DYNASTY components not shown in Fig. 5.1 include design and technology server database, and an internal Tcl command interpreter. A typical DYNASTY design session is shown in Fig. 5.2.

The internal design representation allows a combination of design views during a reconfigurable design exploration. For example during temporal floorplanning various portions of the design could be available in the architectural and behavioural views. Incomplete design representations are also supported to facilitate late insertion of configuration controllers or other static circuits.

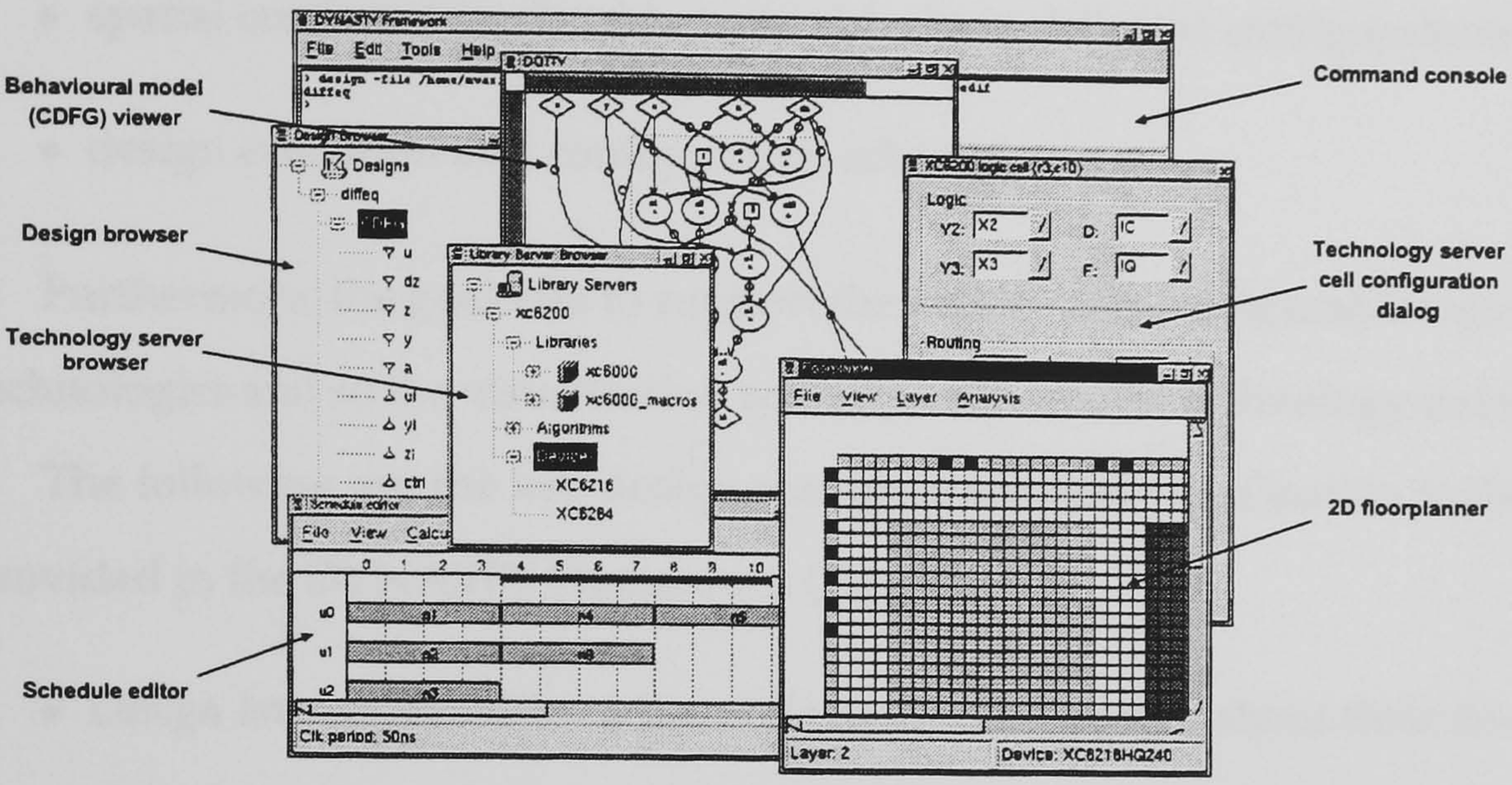


Figure 5.2: Typical DYNASTY session (not all tools shown).

5.1.2 Design Manipulation and Visualisation.

Unlike non-reconfigurable system designers, the designers of dynamically reconfigurable systems are required to analyse numerous design characteristics simultaneously. The search for a good design solution requires analysis of various temporal and spatial design properties, including design latency, throughput, configuration time, spatial conflicts, sharing of reconfigurable resources in different configurations, impact of placement on the RLU reconfiguration time, size of configuration data, power consumption, etc.

A set of novel design visualisation techniques have been developed to support design visualisation and manipulation within the DYNASTY Framework. The aim was to achieve visualisation of the following reconfigurable design properties:

- reconfiguration overhead effects
- configuration partitioning

- spatial conflicts (overlaps) between blocks in different configurations
- design execution and configuration schedule

Furthermore, the goal was to support the variety of reconfigurable logic technologies and so the visualisation techniques should be technology-independent.

The following are the key design manipulation and visualisation tools provided in the DYNASTY Framework (Fig. 5.2):

- *Design browser* provides a list of designs, with details about their design elements at all abstraction levels. The browser allows manipulation of the individual design elements in order to perform allocation, module placement, invoke specific design algorithms, etc. The following models are used to represent the design at three abstraction levels:
  - Control/Data Flow Graph (CDFG) represents design behaviour
  - Finite-State Machine Datapath (FSMD) represents design architecture.
  - 3D structural netlist is used to represent the design layout
- *Technology server browser* provides list of supported technology servers, with the details of their libraries, devices and technology-specific algorithms. The most suitable algorithm for a given design stage can be chosen interactively. This is typically used to chose an estimation technique with the desired accuracy/run-time tradeoff.
- *Behavioural model viewer* provides a CDFG view of the system behaviour.
- *Schedule Editor* allows tracking the execution and reconfiguration schedule for the system during the design space exploration.

- *Floorplanner* provides design structure visualisation in either spatial 2D or 3D floorplan. It provides two views:

  - *Configuration view* (e.g. Fig. 5.3(b)) represents partitioning of the design into individual configurations. Such a view is useful in the early design stages when a designer needs to perform this partitioning on a behavioural design model. At this stage, only sequencing of design block execution is determined, while the cycle-accurate execution schedule will be calculated at a later stage.

  - *System clock view* (e.g. Fig. 5.4(b)) is a cycle-true display of the design activity. This view includes visualisation of both execution and configuration processes for all design blocks. The cycle-true schedule can be recalculated by the technology server as the design is being manipulated.

- *Cell configuration dialog* allows manipulation of the configuration for the detailed layout elements, e.g. routing and logic switches.

The structure and parameters of the reconfigurable logic design solution can be directly manipulated via the DYNASTY Framework graphical user interface. A change in any view will be propagated to the relevant design representation in the other design views. For example, when a design module is placed in a configuration which results in a violation of data dependencies in the design CDFG, the execution and configuration schedules can be automatically recalculated to reflect the resulting design latency and reconfiguration overhead.

5.1.2.1 Reconfiguration Overhead Effects

In the DYNASTY Framework, the reconfiguration overhead is calculated by a technology-specific algorithm provided by the technology server. In the current implementation, the Framework supports reconfigurable logic designs with one configuration controller. The period of reconfiguration is identified in the Schedule Editor using a red bar on the top of schedule display (seen as dark-grey in Figure 5.2). In the Floorplanner tool, the configuration of individual blocks is indicated using a pyramid (3D Floorplanner) or a triangle (2D Floorplanner). The number of pyramids/triangles in the direction of the z-axis indicates the configuration interface activity during the system clock cycles.

With these techniques designers can assess the configuration overheads for the current placement, partitioning and clock/configuration cycles of a reconfigurable logic design.

5.1.2.2 Execution and configuration schedule for the design

Due to the interdependencies between the execution and configuration design scheduling, both schedules have been merged into a single Schedule Editor tool. Here the overall execution schedule is displayed, which combines the execution and configuration latencies of the individual design blocks. Schedule steps are identical to the system clock cycles. If the configuration clock is different from the system clock, the configuration latencies are scaled to the system clock units.

5.1.2.3 2D versus 3D Floorplanner

The Floorplanner has been designed to provide design visualisation in both 2 and 3 dimensions. While a 3D floorplan view represents the overall de-

sign structure and partitions well, its manipulation may become tedious for large designs. With the 2D Floorplanner, designers can examine each of the layers individually and also the locations which are 'difficult to reach' in a 3D view. The 2D Floorplanner is also better suited for the exploration of the desired sharing between configuration layers (users can display selected number of layers to examine their similarities).

5.1.3 Technology Server

Available reconfigurable technologies support a wide variety of reconfiguration mechanisms and device architectures as was discussed in Section 2.2.1. The use of technology servers in our Framework offers technology independence as the technology-specific features can be provided as a 'plug-in' technology server.

In its basic configuration, the technology server includes the following components:

- *A set of target-technology cell and module libraries.* These are a common part of modern FPGA design tools and provide a selection of technology-specific components which can be used in the design.

- *Reconfigurable architecture device models* provide a detailed model for each of the available devices. Such models contain all logic, routing and configurable resources available in the target technology.

- *Technology-specific algorithms.* These include algorithms for estimation of configuration overheads at various levels, placement and routing algorithms, delay estimation, and other routines required to support the specific features of the target reconfigurable technology.

Compared with the other approaches used to implement a generic tech-

nology support, the technology server is unique in providing technology-specific algorithms along with the technology libraries and device models.

5.1.4 Design Simulation

The simulation of reconfigurable logic designs in DYNASTY is supported at two abstraction levels:

- A VHDL simulation model can be generated for the design at any stage during the design. Clock Morphing simulation (Vasilko and Cabanis, 1999) was selected as a primary simulation method in the Framework for its ability to provide simulation of a reconfigurable design at various abstraction levels.

- The completed design can be exported (via the EDIF third-party tools interface) to Xilinx XACT6000 (XC6200 P&R tool (Xilinx, 1997a)), where a detailed timing model can be generated and then simulated using a third-party VHDL simulator.

5.1.5 Third-Party Interfaces

A design can be imported into the Framework in the EDIF 200 format (Stanford and Mancuso, 1990), which can be exported from many popular design entry tools. A design behavioural model can be stored in EDIF as a structural representation of the design CDFG.

The design can be exported from the DYNASTY Framework in both EDIF and VHDL formats. This can be used for simulation, synthesis or delay analysis using third party tools. The reconfigurable design configuration data can be generated using a technology-specific bitstream generator, which produces design configuration files.

5.1.6 Synthesis of Configuration Controllers and Static Design Modules.

Automatic synthesis of configuration controllers is not directly supported by the DYNASTY Framework. However, the Framework can generate a configuration control schedule in a text file, from which such controllers can be constructed using standard ASIC/FPGA design tools or processor compilation tools.

The Framework could provide a library of various reconfiguration controllers suitable for the selected target reconfigurable logic technology. Such a library could be used by the technology server configuration overhead estimation algorithms in order to provide estimates on non-deterministic metrics such as overheads due to random interrupts or memory contention, etc.

DYNASTY's built-in Tcl language command interpreter allows for all of the technology server components to be defined using Tcl commands. Such a capability allows for the technology server to reside on a network and communicate the technology-specific characteristics and algorithms to the DYNASTY Framework remotely.

5.2 Designing with the DYNASTY Framework

From the user's perspective the DYNASTY Framework provides a collection of tools allowing the designer to construct and analyse various reconfigurable logic design solutions in an interactive environment. A typical design sequence in the Framework includes the following steps:

1. *Design capture* using either schematic or HDL design entry.

2. *Selection of the static parts of the design* which should not be subjected

to reconfigurable logic design exploration. These are marked as static throughout the design flow.

3. *Design exploration of a reconfigurable design search space* using temporal floorplanning. This involves the use of the tools described in Section 5.1.2. Typically, a good candidate solution is created first and various implementation and scheduling options are then explored in order to meet the design criteria.

   An initial solution can be created by allocating modules from the technology libraries to nodes in the design CDFG (using Design browser) and placing these modules in the design floorplan (using 3D floorplanner). The design performance can then be estimated (using Schedule Editor). Design exploration is performed by gradual modification of design parameters (module allocation & placement, execution and reconfiguration schedule, spatial and temporal partitioning, etc.). Execution and configuration schedules are analysed throughout the design exploration in order to (i) verify design performance and (ii) ensure that no data, control and configuration dependencies have been violated.

   Once a satisfactory design solution has been created, it can be exported from the Framework for a detailed timing analysis. Any violations of timing constraints are used to adjust design solution parameters until all design constraints are met.

4. *Generation of final solution.* The configuration bitstreams are generated for the final design.

5.3 Design Example

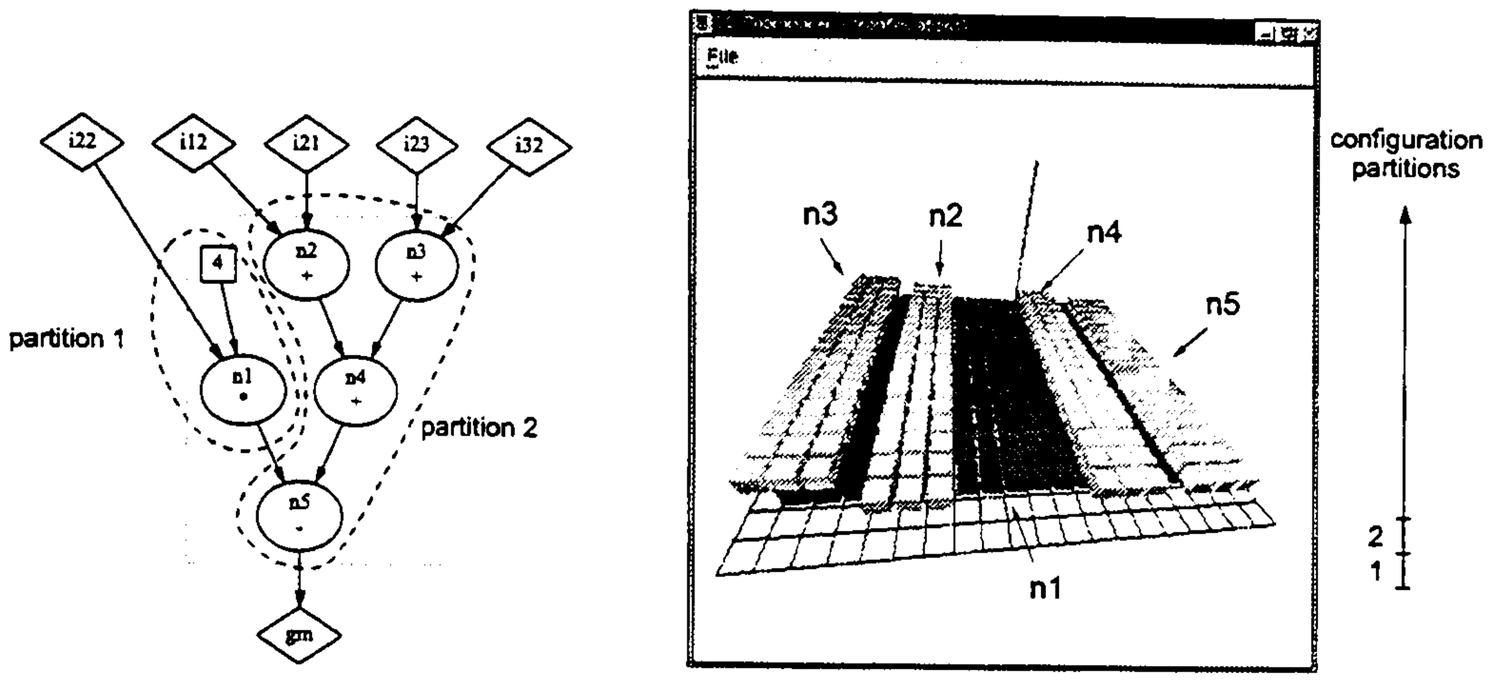
A simple example is used in this Section to demonstrate some of the capabilities of the DYNASTY Framework. Other examples are provided in the core text of this thesis. In the following only the 3D Floorplanner visualisation tool will be used. The example will be implemented on a model dynamically reconfigurable FPGA architecture, derived from the Xilinx XC6200 FPGA technology (Appendix A).

A Laplace transform operator mask design is used here as an example to demonstrate the design flow (its data-flow graph is shown in Fig. 5.3(a)). This design is also used later in the thesis (Chapter 7) to benchmark the performance of the developed synthesis technique.

Let us consider an implementation of the Laplace operator on a resource-limited FPGA architecture (20 × 20 array). The size of the reconfigurable array does not allow for the entire Laplace operator to be implemented in a single configuration. A designer may opt to consider an alternative implementation where the data-flow computation is 'folded' over several configurations.

In this case the designer would construct a 3D floorplan from the available design blocks. The main design objective in most cases will be to minimise latency of the execution for the entire design. The latency is determined by both module execution latency and the configuration latency[2]. While the module execution latency is fixed for a given module type, the configuration latency can be reduced if module resources can be shared between configurations. The designer needs to identify design solutions,

[2]In order to maintain clarity of the presented example, the configuration clock frequency was chosen so that the number of system clock cycles needed for configuration does not exceed four. Selection of ratio between the system and configuration clock will normally depend on design objectives and constraints.



(a) Data-flow graph.

(b) 3D floorplan (configuration partition view).

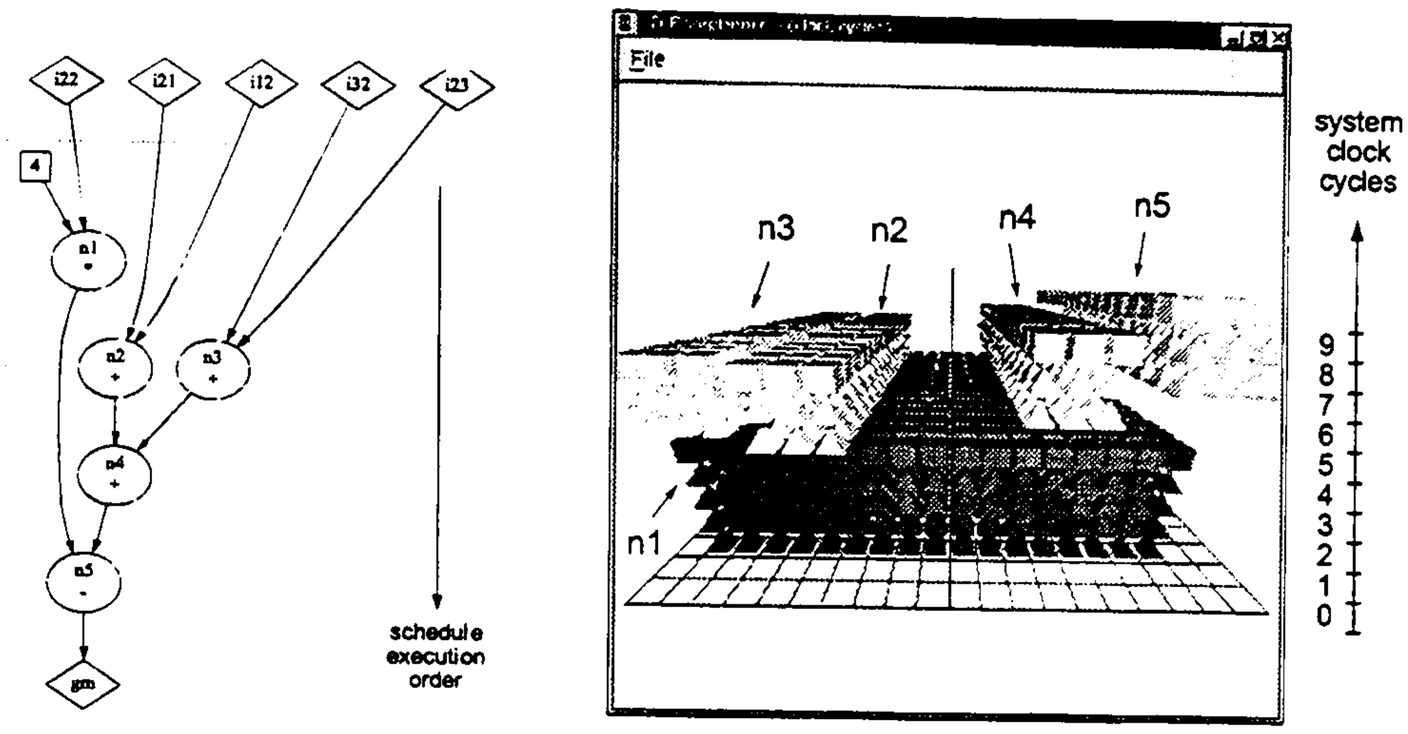
Figure 5.3: Laplace operator 3D floorplan and data-flow graph after scheduling. Each layer in the 3D floorplan represents one design *configuration* as partitioned by a designer.

where such resource sharing is maximised.

First the design modules can be partitioned into individual configurations. The 3D Floorplanner tool in a configuration view can be used to visualise such an initial solution (Figure 5.3(b)). Once initial partitioning was decided, the designer would aim to minimise configuration overhead with a module placement which would maximise module sharing. The actual execution latency can be measured in the Schedule Editor and seen in the 3D Floorplaner using a system clock view (Figure 5.4(b)).

5.4 Conclusions

The DYNASTY Framework provides a combination of techniques, which allow simultaneous exploration of a reconfigurable design search space in both temporal and spatial dimensions. Technology-dependent features are provided by a technology server, which is unique in providing device mod-



(a) Sequenced data-flow graph.

(b) 3D floorplan (system clock view).

Figure 5.4: Laplace operator 3D floorplan and data-flow graph after scheduling. Each layer in the 3D floorplan represents one *system clock cycle;* a pyramid indicates that a block is being reconfigured and a cube denotes its execution.

els and technology-specific algorithms along with the cell and module libraries.

The experience with using DYNASTY for the design of various partially-reconfigurable circuits in the XC6200 technology, confirms that temporal floorplanning leads to a considerable reduction of the design time compared to the other iterative XC6200 design methodologies (e.g. (Robinson et al., 1998)).

Such a reduction can be attributed to the capabilities of the Framework, which provide designers with an immediate visual feedback about design characteristics throughout the entire design flow, and allow design manipulation at multiple abstraction levels simultaneously. Bad design decisions can be identified early, while the feasibility of the final design solution is guaranteed through checking the dependency violations.

The Framework allows an expert human designer to control the entire

reconfigurable logic design process, while automatic design and estimation techniques can provide guidance and acceleration of computationally intensive tasks. Further 'push-button' techniques for automatic exploration of a multiple-level reconfigurable design search space will provide a fast design route in scenarios where design time is a primary objective, while possible implementation inefficiencies can be tolerated. One such automatic technique is presented in this thesis.

Although further development of automatic synthesis and estimation algorithms for reconfigurable systems can be expected to reduce the emphasis on manual reconfigurable logic design manipulation, the presented visualisation techniques will still be able to provide an intuitive visual framework for the analysis and manipulation of auto-generated design solutions.

Chapter 6

Synthesis of Dynamically Reconfigurable Systems with Evolutionary Algorithms

The theoretical model for the synthesis of reconfigurable systems presented in the previous chapter provides a framework in which various optimisation algorithms can search for design solutions.

In order to confirm the viability of this formulation an evolutionary algorithm based optimisation technique has been developed for the synthesis of reconfigurable systems, which is presented in this chapter.

The following section defines a *restricted* problem for the synthesis of reconfigurable systems, which is considered by the presented evolutionary optimisation technique. Section 6.2 presents a newly developed *temporal floorplanning* representation of this problem. Section 6.3 discusses the suitability of various optimisation algorithms for the solution search using the temporal floorplanning representation. Genetic algorithms are briefly introduced in Section 6.4. The implementation of the newly developed re-

configurable system synthesis technique based on genetic algorithms is described in Section 6.5.

6.1 Restricted Problem for Synthesis of Reconfigurable Systems

The formulation of the reconfigurable synthesis problem presented in Chapter 4 defines a generalised and complex set of interdependent transformations. In order to simplify the initial search for algorithms capable of solving this problem, the formulation was restricted into a simplified instance of the original reconfigurable synthesis problem.

This section presents the restricted formulation which constrains the type of reconfigurable system into a system, which can be synthesised using the presented approach. Furthermore, the assumptions about the design methodology within which this algorithm operates provide a simplification of several processing steps to allow practical verification of the presented technique.

The following are the assumptions and practical considerations which restrict the RS synthesis problem considered here:

1. **Only acyclic data-flow problems are considered.** With respect to Definition 4.1.1, in the input behavioural model $G(V, E)$ the sets of control vertices and control edges are assumed to be empty:

$$V_c = \emptyset \quad \wedge \quad E_c = \emptyset \tag{6.1}$$

and thus for $G(V, E)$, it is assumed that $V = V_d$ and $E = E_d$. In this case $G(V, E)$ represents a data flow graph.

This assumption ensures that no cycles and no conditional execution branches exist in the input problem. Therefore the design execution

and reconfiguration schedules can be determined and fixed during synthesis.

While only acyclic graphs are considered, it should be noted that cyclic data-flow graphs can be easily transformed into acyclic graphs representing algorithm iterations with non-overlapping schedule. This approach is commonly used in high-level synthesis (Gerez, 1999, Chapter 12).

2. **The input design problem is deterministic.** It is assumed that the input design problem is fully specified prior to synthesis and therefore all behavioural design characteristics and dependencies can be determined during the synthesis.

3. **Logic synthesis is performed via direct mapping to 'hard' library modules.** It is assumed that the architectural resource library for the targeted technology provides a set hard macro modules which are available during synthesis.

   The term 'hard macro modules' refers to target technology modules, in which primitive cells were placed relative to the module origin and routed locally. The use of only local routing allows for modules to be placed in more than one location within the target technology array.

   In respect to Definition 4.2.5, given the design architectural model (A, β, σ, ρ), the design logic model (U, N) can be constructed by replacing the architectural resource instances by their corresponding 'hard' macro modules from the target technology library.

4. **Architectural modules can be connected only via register transfers.** It is assumed that the architectural module input/output register values are transferred between the modules via the RLU's configura-

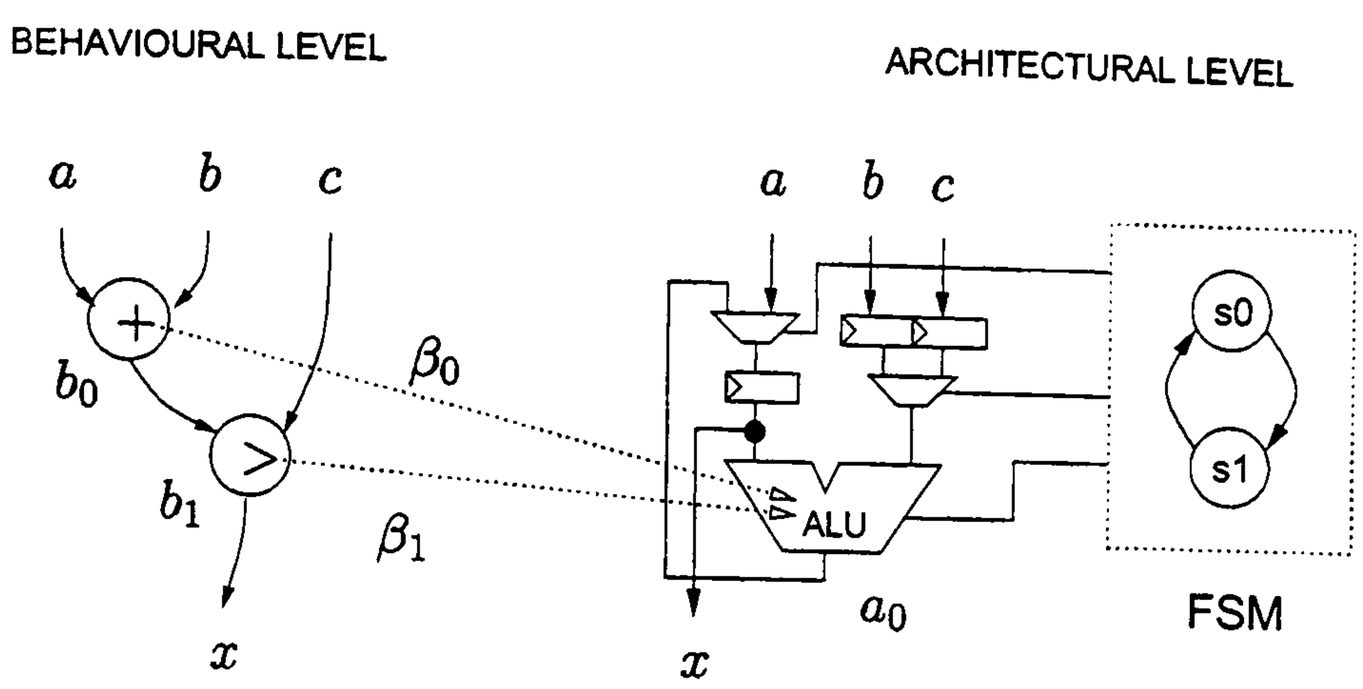


Figure 6.1: Architectural-level resource sharing controlled by an FSM.

tion interface. No physical wiring is allowed *between* the architectural modules.

In order to allow for inter-module wiring to exist, the synthesis process would have to address the routing problem for reconfigurable systems. This is a complex problem, for which solutions have not yet been proposed and therefore the routing problem is not considered here. The routing problem in reconfigurable systems is further discussed in Section 8.2.

5. **Only a restricted case of architectural-level resource sharing is considered.**

   Implementation of resource sharing at the architectural level may require that a finite-state machine (FSM) is constructed for each instance where such sharing occurs. This FSM controls the access of input/output signals and registers to the ports of the shared architectural resource (Fig 6.1). The existence of such a FSM would require that both the FSM and its supporting logic are synthesised as a part of the RS synthesis. Such FSM synthesis is not considered here.

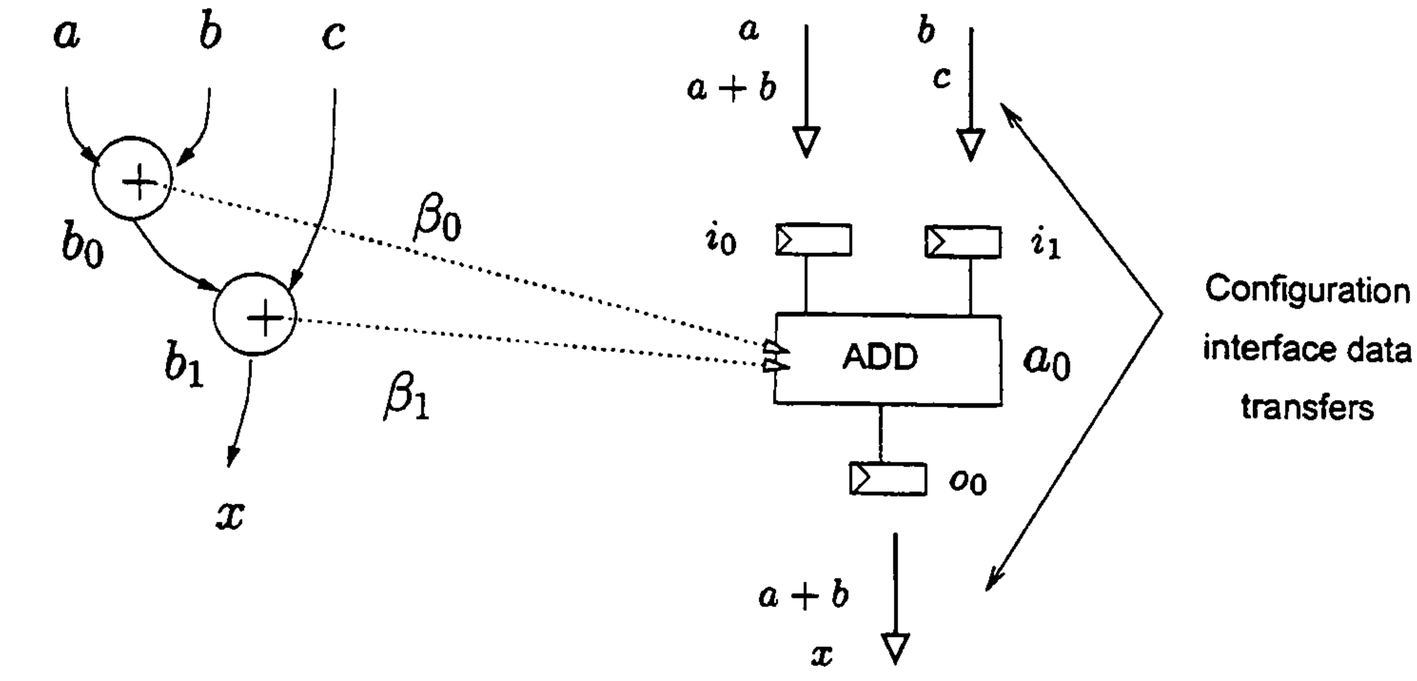


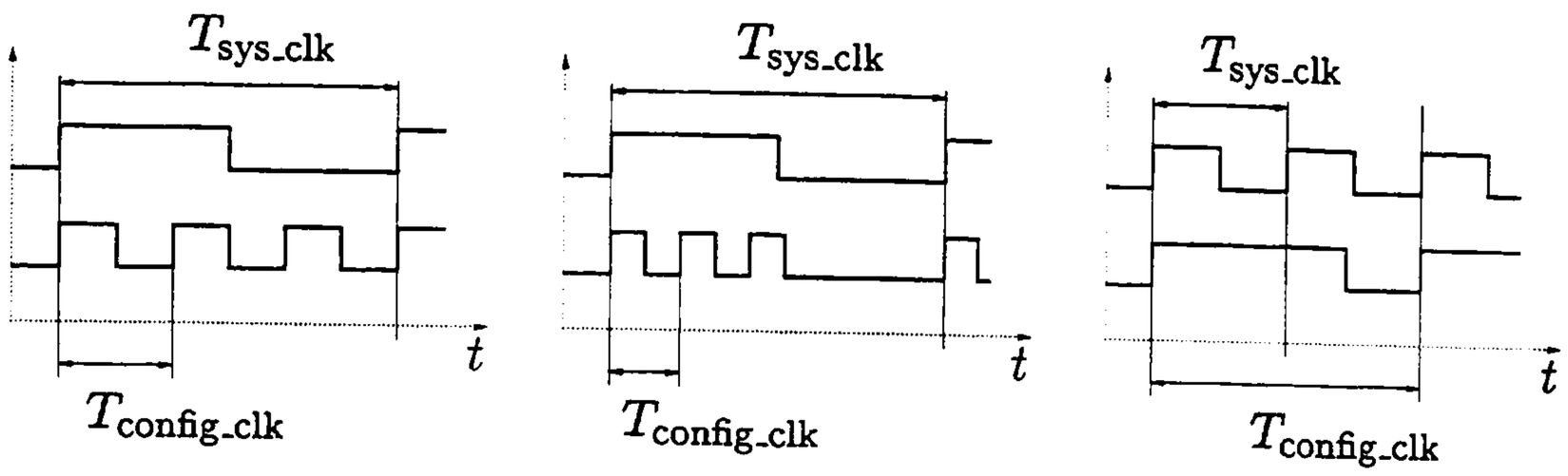
Figure 6.2: Architectural-level resource sharing with module a_0 shared between behavioural computations b_0 and b_1. The values stored in registers i_0, i_1, o_0 are transferred via the configuration interface.

However, an architectural module can be shared between two behavioural computations without any additional logic in the arrangement shown in Fig. 6.2. In this case, the shared module remains configured in the RLU, while its input/output registers are programmed with new values via the configuration interface.

This type of architectural-level resource sharing is considered by the presented restricted synthesis problem. Resource sharing at physical level is also considered

6. **Target architecture assumptions.** It is assumed that synthesis is targeting the reconfigurable system architecture depicted in Fig. 2.1, with the following restrictions/features:

   - the target architecture is a *synchronous* system. Both the computations in the RLU and the system's RCU are synchronised with one *system clock* signal. The communication between the RLU and RCU is controlled by a separate *configuration clock*, which



(a) $T_{\text{sys\_clk}} = 3 \times T_{\text{config\_clk}}$ (b) $T_{\text{sys\_clk}} > T_{\text{config\_clk}}$ (c) $T_{\text{sys\_clk}} < T_{\text{config\_clk}}$

Figure 6.3: Relationship between the system and configuration clock signals. $T_{\text{config\_clk}}$ and $T_{\text{sys\_clk}}$ are the periods of the configuration clock signal and system clock signal respectively.

is synchronised with the system clock at the beginning of the system clock cycle, although it may run asynchronously within the system cycle. Examples of possible timing relationships between the system and configuration clock signals are illustrated in Fig. 6.3.

This ensures that the relationship between the system and configuration clock signals is well defined.

- the RLU is implemented on a single reconfigurable logic device and therefore partitioning between multiple reconfigurable devices will not be considered
- reconfiguration is controlled by a dedicated RCU, which is external to the RLU and therefore all RLU resources are available for the design implementation
- the RCU-CDS and RCU-RLU interfaces are implemented as dedicated interfaces which are not shared between other system components. The data transfers across these interfaces will be fully

deterministic and therefore there is no need to consider delays due to bus sharing and arbitration.

- the configuration and data stored within the CDS can be read/written within one configuration clock cycle. Furthermore, the CDS has a storage capacity sufficient for the storage required for the configuration, RLU state and application-specific data.

7. **Synthesis goals.** It is assumed that there are two main goals for reconfigurable logic system synthesis:

   - The primary goal is to produce a *feasible* reconfigurable logic system (Definition 4.2.7).
   - The secondary goal is to produce a system which meets the constraint on its execution latency.

   These goals guarantee that even if the solution violates the secondary design goal (execution latency constraint), it will be guaranteed to operate correctly in the target reconfigurable system. Therefore the approach can be used for feasibility evaluation of the given design constraints (e.g. design latency and the targeted technology).

The above assumptions restrict the application domain for which the presented synthesis technique will be applicable. However, these assumptions are representative of many practical reconfigurable systems implementing real-time applications, such as digital signal and image processing, but also other applications which use data-flow computations.

6.2 Synthesis Process Overview (Temporal Floorplanning)

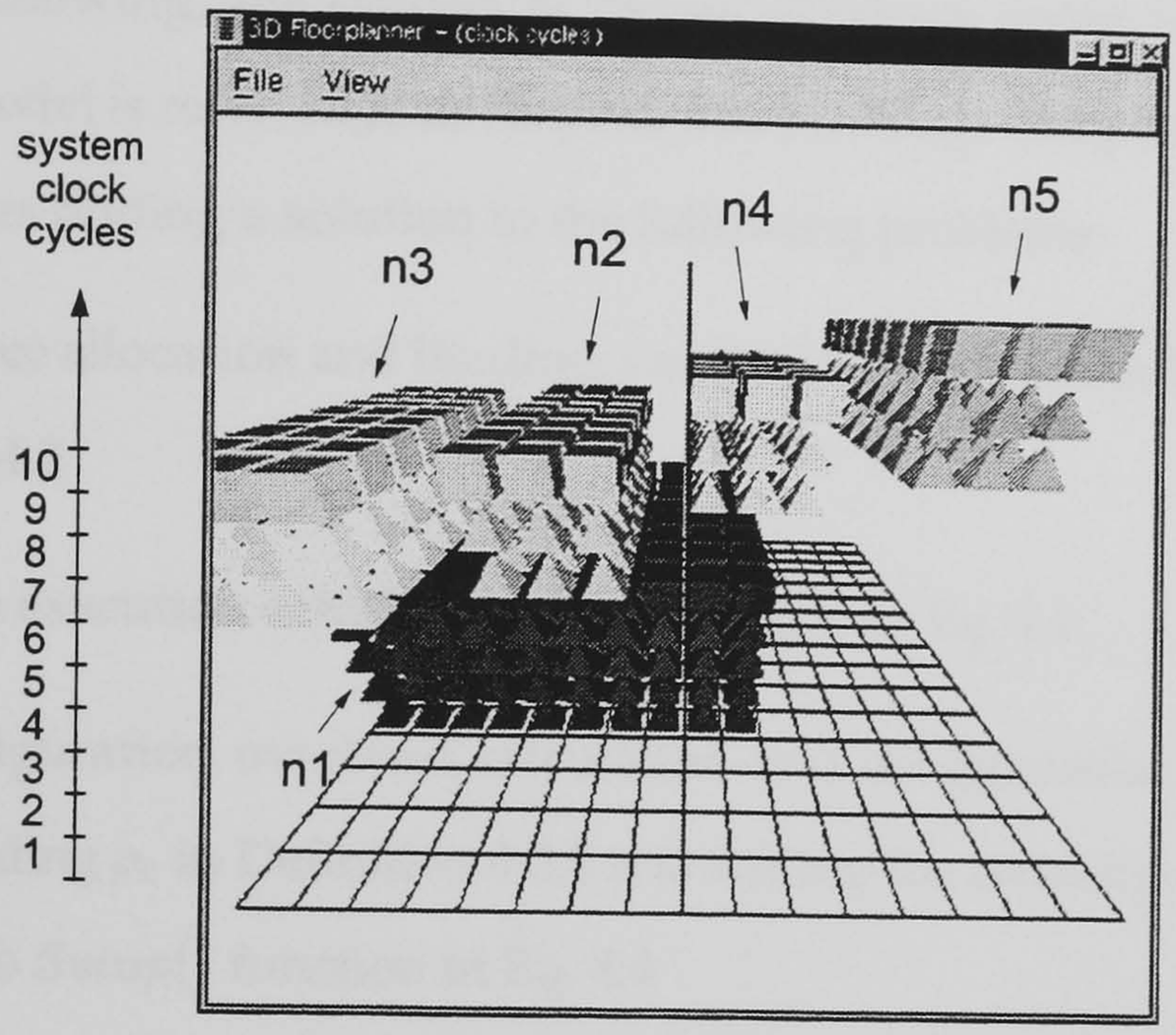
The first problem in the reconfigurable synthesis formulation is resource allocation and binding. Then a scheduling problem can be considered. The feasibility of a reconfigurable design schedule for the selected type of target technology may depend greatly on the feasibility of the design layout, where there should be no spatial or temporal conflicts between the design configurations.

The restricted RS synthesis problem assumes that the logic synthesis is replaced by direct mapping of the architectural modules onto the target technology hard macro modules. Such an approach allows for the physical characteristics of the technology modules, e.g. module dimensions and latency, to be known at a high level. Furthermore, as no wiring is considered, physical synthesis translates to a problem of design module placement.

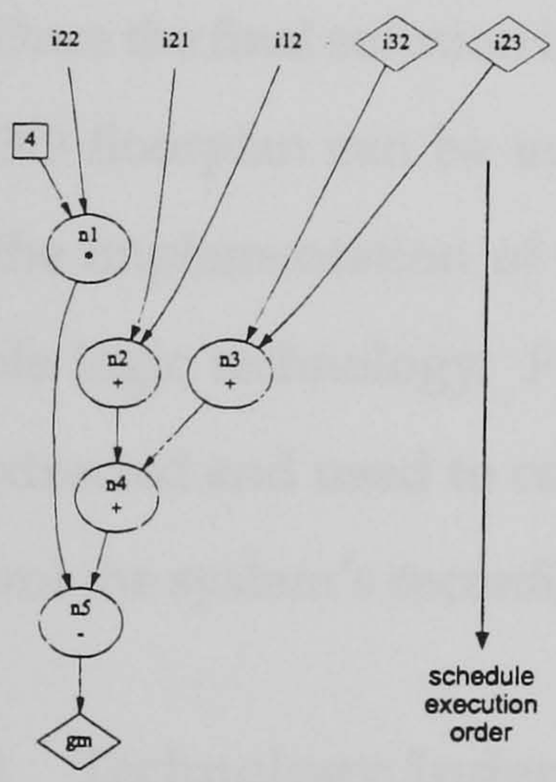
Due to the interdependent relationship between the above problems, i.e resource allocation, resource binding, design scheduling and module placement, it would be useful to consider the problems together. This is possible within a 3D floorplan model (see Fig. 6.4(a)).

In the 3D floorplan the horizontal x/y-coordinates represent spatial positions of the design modules, while vertical z-coordinate represent the system clock cycles. In Fig. 6.4(a) the execution of the system proceeds from the bottom ($z = 0$) towards the top of the floorplan. The 3D dimensions of the design module represent its spatial and temporal characteristics, extracted from the library cell linked with the design block as a result of resource allocation and binding.

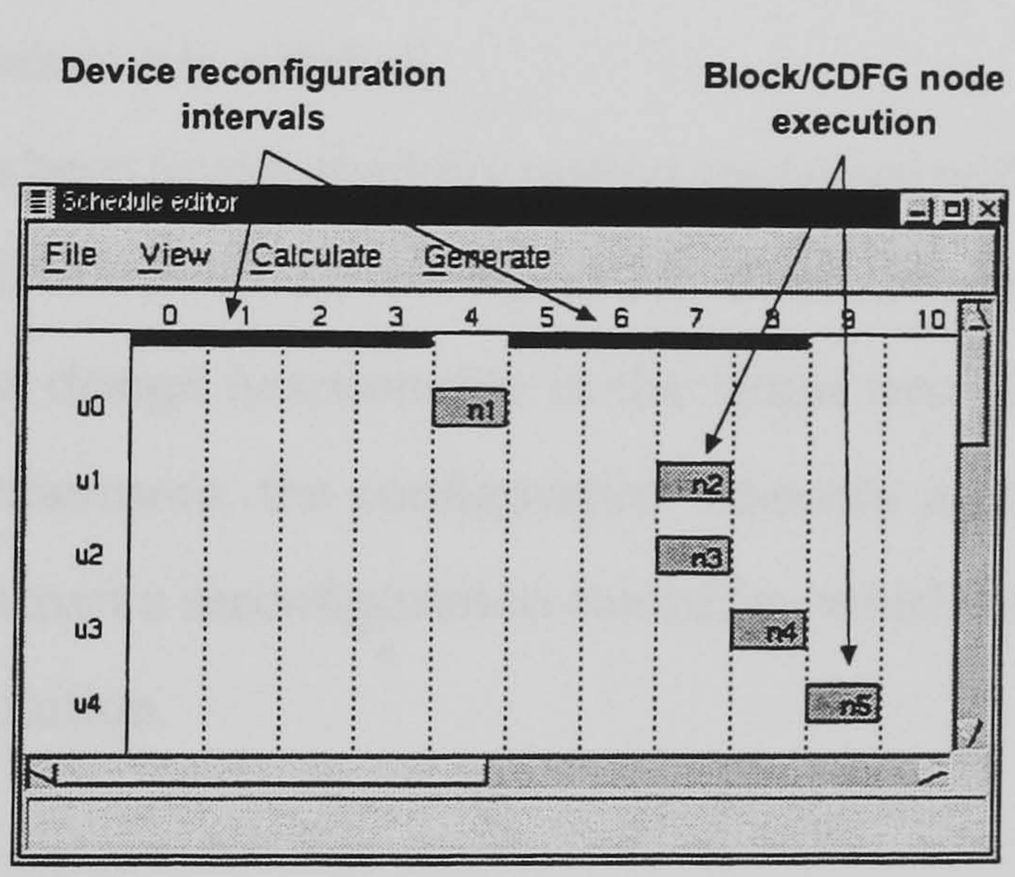
The 3D floorplan can be therefore represented as a 4-tuple (A, β, σ, ϕ). Considering the above interpretation of the 3D floorplan, the 4-tuple can



(a) 3D floorplan.



(b) Scheduled DFG.



(c) Design execution and configuration schedule.

Figure 6.4: A Laplace operator mask 3D floorplan and data-flow graph during temporal floorplanning. Each layer in the 3D floorplan represents one *system clock cycle;* a pyramid indicates that a block is being reconfigured while a cube denotes its execution.

be expressed as (A, β, z, x, y)

In the following, the process of design synthesis using the above 3D floorplan model is referred to as *temporal floorplanning*. Temporal floorplanning involves finding a solution to the following problems:

- resource allocation and binding, i.e. finding the set A in Eq. 4.1 and β in Eq. 4.2
- design execution scheduling, i.e. finding σ in Eq. 4.4
- reconfiguration overhead calculation and configuration scheduling, i.e. finding ρ_c in Definition 4.2.5 and scaling the reconfiguration overhead to $Setup()$ function in Eq. 4.4
- design block placement, i.e. finding ϕ in Definition 4.2.6

while the solution feasibility condition is not violated (Definition 4.2.7) and the set of performance constraints ψ is satisfied.

Once the final solution has been found, the information contained within the 3D floorplan can be used to extract the configuration data necessary for the implementation of the design functionality in the target reconfigurable logic technology. Furthermore, the configuration schedule ρ_c can be extracted and used to construct a reconfiguration controller, which will control the system's reconfiguration.

6.2.1 Technology Independence

In the 3D floorplan model, the base 2-D floorplan array is composed of blocks which encapsulate the primitive components of the target technology. For example in Fig. 6.4(a) a primitive floorplan block corresponds to a logic block and a set of routing multiplexors in the Xilinx XC6200 technology (see Fig. A.1 in Appendix A, page 149). While different reconfigurable

devices provide different primitive physical components, their abstraction in the 3D floorplan is the same. Therefore, the 3D floorplan abstraction allows algorithms to operate within one common model, while targeting different technologies.

6.3 Optimisation Algorithm Selection

Chapter 4 has formulated the problem of synthesis for reconfigurable systems. Although only a restricted case of this problem is considered for automatic synthesis here, the formulation demonstrates tight interdependence between the individual tasks. Most importantly, the technology-specific features such as partial reconfiguration and sophisticated features of reconfigurable logic configuration interfaces, can further complicate these interdependencies.

In the search for a suitable optimisation algorithm for this problem a number of options were examined.

In non-reconfigurable system synthesis, simple and fast heuristics are used to solve the individual synthesis problems. These techniques cannot guarantee the feasibility of the generated solution because they operate on simplified problem models, which do not consider the low-level design characteristics and technology constraints. In the case of reconfigurable systems, examples which use simple heuristic search techniques include (Ling and Amano, 1993b) and also the author's previous work (Vasilko and Ait-Boudaoud, 1996a).

Given the complex formulation of the problem of synthesis for reconfigurable systems, and the tight interdependence between the individual synthesis problems, techniques capable of considering all the interdependencies should be more appropriate. The following techniques were con-

sidered:

- **Integer Linear Programming (ILP)** (e.g. (Nemhauser and Wolsey, 1988)) is a robust optimisation technique capable of finding a globally optimal solution for any problem which can be formulated as a set of linear relations. However, the run time of these techniques is prohibitive for large and interdependent problems. Examples of using ILP optimisation on simplified models of reconfigurable system synthesis include (Sels, 1996; Kaul and Vemuri, 1998).

- **Simulated Annealing** (Kirkpatrick et al., 1983) is a stochastic optimisation method based on a mathematical model of annealing in natural systems. The technique gained its popularity through its ability to climb out of local minima.

  Although the run time of simulated annealing based optimisation can be significantly shorter than that of an ILP method, it can still be prohibitive for large problems. The long run times can be attributed to the technique using relatively small steps to progress towards new solutions during the search. A simulated annealing based algorithm for the synthesis of reconfigurable systems in a simple 3D floorplan was demonstrated by Bazargan et al. (1999).

- **Evolutionary Algorithms** (Holland, 1975; Goldberg, 1989) are a global optimisation technique based on a model of a Darwinian evolution. Like simulated annealing, this technique is capable of 'hill-climbing'. Evolutionary algorithms operate on a large set of solutions rather than on a single solution. This allows for a large number of alternative solutions to be examined within a short period of time. The application of evolutionary algorithms to a simplified problem of re-

configurable system synthesis was considered by Zhang et al. (1998).

Evolutionary algorithms were selected as a suitable candidate for the evaluation of reconfigurable system synthesis using the presented problem formulation. One category of evolutionary algorithms, *genetic algorithms (GAs)*, have attracted a considerable interest over recent years for their applications to complex real-world problems.

Evolutionary algorithms have also been demonstrated to successfully solve multi-objective and interdependent problems in VLSI CAD (e.g. (Dodhi et al., 1995; Ohmori, 1995; Morris and Nowrouzian, 1996; Sait et al., 1996)).

The ability of genetic algorithms to search a large pool of very different solutions early during the solution search will be beneficial for reconfigurable system synthesis. While the overall objective of reconfigurable system synthesis is to produce a feasible solution which meets the performance constraints, it is often only necessary to quickly evaluate the feasibility of the target technology or a performance constraint for a given input problem. A genetic algorithm will produce many different solutions early during the solution search, allowing designers to assess the suitability of the target technology from these early results.

6.4 Genetic Algorithms

Genetic algorithms (Goldberg, 1989; Holland, 1975) are a stochastic optimisation technique based on principles of natural evolution. They operate using a mathematical model of natural evolution based on the 'survival of the fittest' strategy, which is similar to the process thought to occur in nature, and which can lead to the selection of the best solution for a given set of environmental conditions.

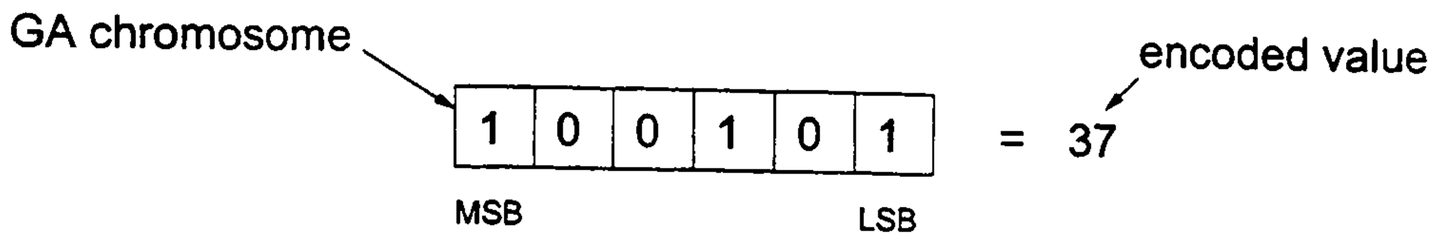


Figure 6.5: An example of a chromosome coding in a genetic algorithm (1-dimensional binary string). The binary value encoded in the chromosome is linked to the system variable under optimisation.

An optimisation problem which is to be considered by the genetic algorithms is encoded so that it can be manipulated by the algorithm. The individual solutions are coded in a data structure representing *chromosomes*. The values encoded in the chromosome are linked to system variables, the values of which are the subject of GA optimisation. For example in Fig. 6.5, the solution is represented by a binary string of a fixed size. Its encoded integer value can be linked to a single one-dimensional system variable, e.g. an angle, velocity, position, priority, etc.

The chromosomes are grouped to form a *population* of possible solutions, which is manipulated by two types of *genetic operators* during the evolutionary process: *crossover* and *mutation*.

The crossover operator selects pairs of chromosomes from the old generation ('parents') and performs their 'mating' to generate two new chromosomes ('children') in the new generation (Fig. 6.6). These children may combine some of the 'good' characteristics of their parents. Therefore there is a likelihood that the crossover operator will produce at least some new chromosomes which will have better characteristics then the parents.

The mutation operator selects a chromosome from the old generation and introduces a random change to the chromosome which is then stored in the new generation (Fig. 6.7). This allows for the generation of new 'offsprings' which do not inherit the entire set of parents' characteristics.

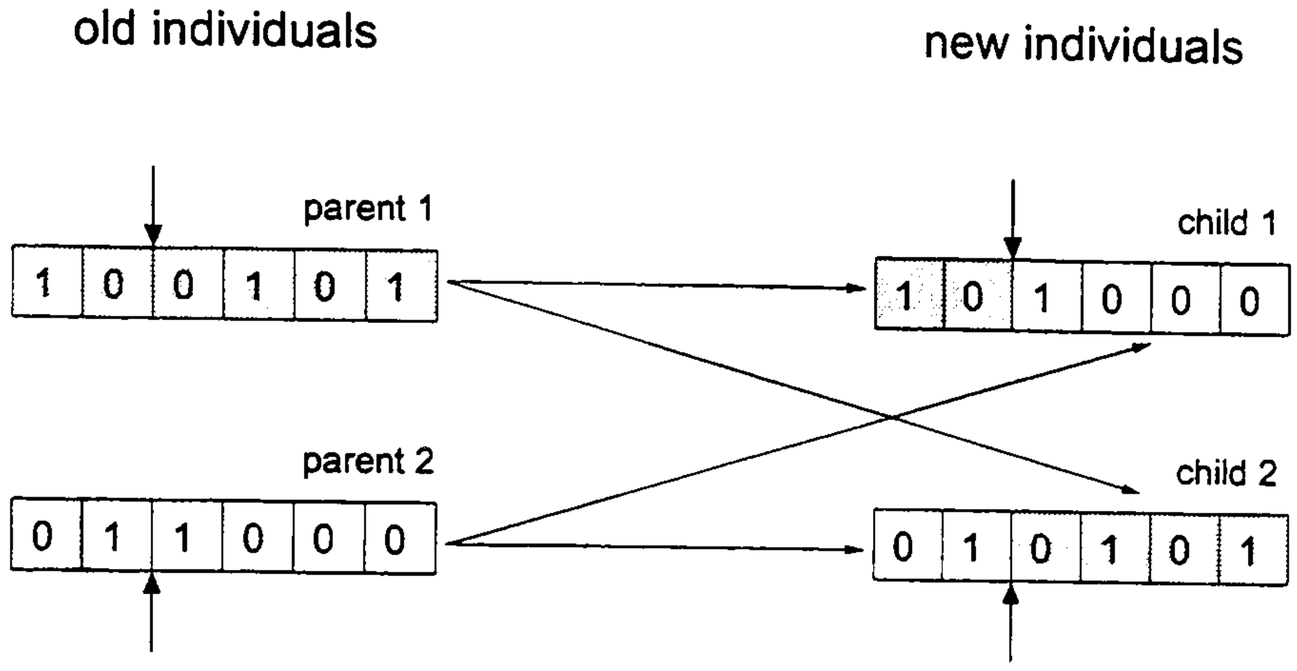


Figure 6.6: An example of a crossover operator (one-point crossover).

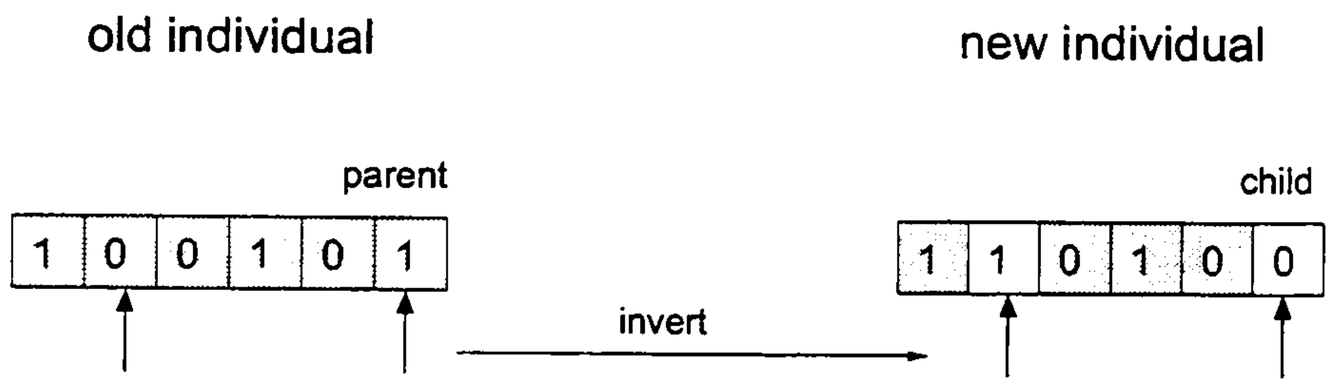


Figure 6.7: An example of a mutation operator (random 'flip' mutation).

This is the mechanism which allows genetic algorithms to escape the 'local minima' during the evolutionary search process. There is a possibility that new characteristics introduced by the mutation operator may provide some chromosomes with different and better characteristics than those of the parents.

A generic simple genetic algorithm can be expressed by the procedure shown in Algorithm 6.1.

Initialisation of a population creates a set of individuals with population size selected by one of the parameters of the genetic algorithms. This initialisation is often performed by a simple random assignment of the chromosome values.

Algorithm 6.1 Simple Genetic Algorithm

initialise a population

evaluate the population fitness

while stopping criteria is not satisfied **do**

select individuals for the next generation

apply crossover

apply mutation

evaluate the population fitness

end while

After initialisation the genetic algorithm enters into an iterative loop which simulates natural evolution by performing a sequence of genetic operations.

First, the *fitness* of each individual in the population is evaluated. The fitness provides a measure of the individual's quality and is traditionally represented by a scalar value.

The fitness evaluation is followed by a *selection* procedure, which selects the individuals which will survive in the next generation. This selection is based on the individuals' fitness: those individuals with high fitness are more likely to survive than the individuals with low fitness. After selection a new generation of individuals is stored as the current population to generate a new generation.

Then both crossover and mutation operators are applied to individuals randomly selected from the current population.

The entire process repeats until one or more *stopping criteria* are satisfied. Various stopping criteria have been used with genetic algorithms. The limit on the total number of generations, and population convergence (similarity of individuals within the entire population), are most commonly

used.

6.5 Implementation of an Automatic Reconfigurable System Synthesis

When genetic algorithms are considered for the optimisation of any given problem, it is necessary to examine several problem-specific issues:

- problem representation
- population initialisation
- selection of genetic operators (crossover and mutation)
- fitness function
- selection of a genetic algorithm procedure and control parameters

6.5.1 Problem Representation

Conventional GA-based optimisation techniques use simple data structures to represent the problem variables. Examples include binary (Fig. 6.7) or integer strings, arrays and trees.

The problem of reconfigurable system synthesis represents a collection of complex and interdependent relationships. Although temporal floorplanning provides a simplification of this process, it still involves a solution search for several interdependent problems.

In order to use genetic algorithms successfully, the problem representation and the genetic operators must match well the characteristics of the problem. For the problem of temporal floorplanning a problem-specific representation was developed together with a set of problem-specific genetic operators.

Algorithm 6.2 Population initialisation

Require: $G(V, E)$, n - population size

1. Create n individuals in population P

for all $p_i \in P$ **do**

2. Perform resource allocation (Algorithm 6.3)

3. Place the design in a 3D floorplan (Algorithm 6.4)

4. Calculate the reconfiguration latency and correct the 3D floorplan using a technology-specific procedure (Algorithm 6.13).

end for

A composite chromosome was designed, where the chromosome genes are linked with behavioural model elements and represent the following behavioural element properties: (Fig. 6.8):

- resource binding, represented as a link to the library or a shared design module implementing the desired behavioural functionality
- 2-D position in the target technology device represented as a pair of integer floorplan x/y coordinates
- temporal position which represents the system's execution schedule time slot

6.5.2 Population Initialisation

An individual in a population is initialised by the procedure in Algorithm 6.2.

First a population of the required size is created. Then all individuals in the population are initialised to hardware modules using a greedy 'first come - first served' allocation and binding algorithm (Algorithm 6.3), followed by a random placement of the design modules in a 3D floorplan (Algorithm 6.4). The placement is followed by a procedure (Algorithm 6.4)

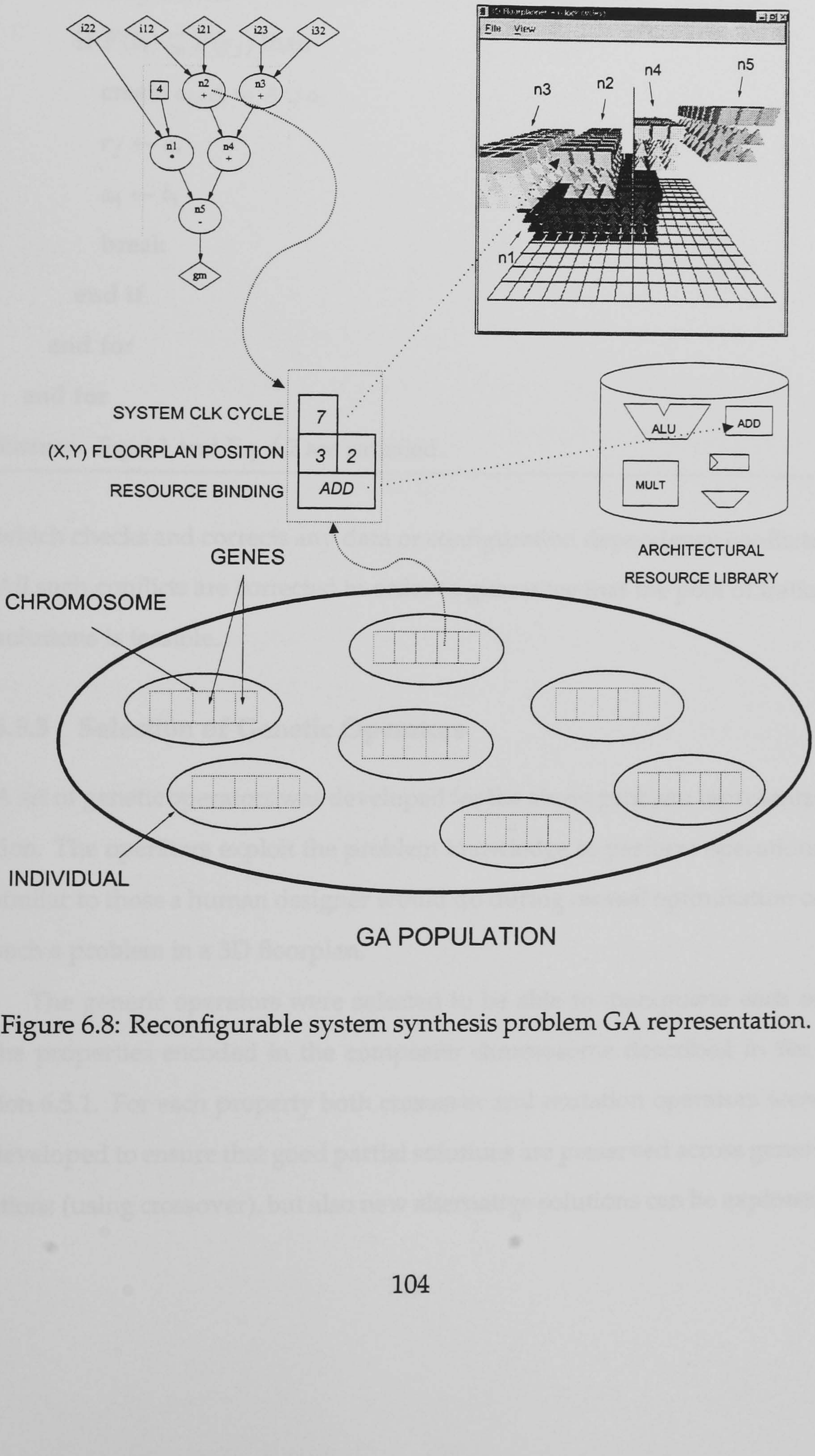


Figure 6.8: Reconfigurable system synthesis problem GA representation.

Algorithm 6.3 First-come first-served allocation and binding

Require: $G(V, E), B = V \cup E, R, A = \emptyset$

for all $b_i \in B$ **do**

for all $r_j \in R$ **do**

if $F(b_i) \subseteq F(r_j)$ **then**

create a_i, $A = A \cup a_i$

$r_j \leftarrow a_i$

$a_i \leftarrow b_i$

break

end if

end for

end for

Ensure: Eq. 4.1 and Eq. 4.2 are satisfied.

which checks and corrects any data or configuration dependency conflicts. All such conflicts are corrected in order to guarantee that the pool of initial solutions is feasible.

6.5.3 Selection of Genetic Operators

A set of genetic operators was developed for the above problem representation. The operators exploit the problem knowledge to perform operations similar to those a human designer would do during *manual* optimisation of such a problem in a 3D floorplan.

The generic operators were selected to be able to manipulate each of the properties encoded in the composite chromosome described in Section 6.5.1. For each property both crossover and mutation operators were developed to ensure that good partial solutions are preserved across generations (using crossover), but also new alternative solutions can be explored

Algorithm 6.4 3D floorplan placement

Require: $G(V, E)$, A, β

for all $a_j \in A$ **do**

1. Randomly select the next module $a_j = \beta(v_j)$ such that all the modules of v_j's predecessors were already placed.
2. Place a_j at a random (x, y) position in the next available system clock cycle such that Eq. 4.4 is satisfied assuming $Setup() = 0$ and the module fits within the target device.

end for

to escape local minima (using mutation).

A special mutation operator (Section 6.5.5.4) was developed after it was observed that many solutions could be improved by slightly changing the locations of the design modules. These slight changes would allow modules to 'descend' into lower locations in a 3D floorplan, which could create new solutions with reduced design latency.

The operation and implementation of all generic operators is further discussed in the following Sections 6.5.4 and 6.5.5.

The probability of application of the operators is controlled by an evolution control strategy designed for this algorithm (Section 6.5.9).

6.5.4 Crossover Operators

The crossover operator simulates 'mating' between two parent individuals which produces two child individuals in the new generation. The following problem-specific crossover operators were developed:

Algorithm 6.5 Module binding crossover

Require: parent1, parent2, $p_{\text{x\_bind}}$ (crossover probability)

for all $v_i \in V$ **do**

if toss_coin($p_{\text{x\_bind}}$) **then**

exchange $\beta(v_i)$ between v_i in parent1 and parent2

end if

end for

Algorithm 6.6 2D floorplan crossover

Require: parent1, parent2, $p_{\text{x\_2D}}$ (crossover probability)

for all $v_i \in V$ **do**

if toss_coin($p_{\text{x\_2D}}$) **then**

exchange (x, y) coordinates between v_i in parent1 and parent2

end if

end for

6.5.4.1 Module binding crossover

This exchanges modules bound to identical CDFG nodes in parent solutions (Algorithm 6.5). This operator aims to preserve binding of modules between the generations.

6.5.4.2 2D floorplan crossover

This exchanges X-Y positions between the modules in one floorplan layer. This operator will preserve the relative spatial positions between the modules in a 3D floorplan (Algorithm 6.6).

6.5.4.3 3D floorplan crossover

This exchanges the positions of a randomly-sized group of modules between the two parent 3D floorplans. This will copy the X-Y-Z positions of

Algorithm 6.7 3D floorplan crossover

Require: parent1, parent2, p_{x_3D} (crossover probability)

for all $v_i \in V$ **do**

if toss_coin(p_{x_3D}) **then**

exchange (x, y, z) coordinates between v_i in parent1 and parent2

end if

end for

the selected modules (both horizontal and vertical placement). This operator preserves both spatial and temporal relationship between the design modules in a 3D floorplan. Relative module position in Z-direction characterises the possibility for physical resource sharing between the blocks (Algorithm 6.7).

6.5.5 Mutation Operators

A mutation operator simulates random changes to one or more individuals. The following problem-specific mutation operations were developed:

6.5.5.1 Module binding mutation

This changes the module bound to a given CDFG node to a module of a different type, but with the same functionality (e.g a ripple-carry adder can be swapped for a carry-lookahead adder). Both the architectural module library and the modules already instantiated in the design are searched for new architectural modules with identical functionality. This allows for either new module types to be introduced in the design solution or for modules with identical functionality to be shared (Algorithm 6.8).

Algorithm 6.8 Module binding mutation

Require: parent, $p_{\text{mut\_bind}}$ (mutation probability)

for all $v_i \in V$ **do**

module_selection_set $\leftarrow$ lib_modules($F(v_i)$) $\cup$ design_modules($F(v_i)$)

if toss_coin($p_{\text{x\_bind}}$) **then**

remove old $\beta(v_i)$ from parent

select m_j from module_selection_set at random

create new binding $\beta(v_i) = m_j$

end if

end for

Algorithm 6.9 2D floorplan mutation

Require: parent, $p_{\text{m\_2D}}$ (mutation probability))

for all $v_i \in V$ **do**

if toss_coin($p_{\text{m\_2D}}$) **then**

calculate_valid_XY_range($\beta(v_i)$)

change (x, y) coordinates for v_i at random (within the valid range)

end if

end for

6.5.5.2 2D floorplan mutation

This randomly changes the X-Y coordinates of the selected module. This operator changes the module position(s) relative to the entire design floorplan (Algorithm 6.9).

6.5.5.3 3D floorplan mutation

This changes the X-Y-Z coordinates of the selected module(s) in a 3D floorplan layer.

Algorithm 6.10 3D floorplan mutation

Require: parent, p_{m_3D} (mutation probability))

for all $v_i \in V$ **do**

if toss_coin(p_{m_3D}) **then**

calculate_valid_XYZ_range($\beta(v_i)$)

change (x, y, z) coordinates for v_i at random (within the valid range)

end if

end for

6.5.5.4 3D floorplan 'shaking'

This special mutation operator was developed to simulate the effect of randomised 'shaking' of the entire 3D floorplan. This is a greedy algorithm, which generates new X-Y coordinates for randomly selected modules in the solution.

The floorplan 'shaking' may lead to further compaction of the 3D floorplan in Z direction as modules from bottom layers may create sufficient space to allow modules from higher layers to 'descend'.

6.5.6 Overall Synthesis Procedure

The overall procedure using the proposed technique is outlined in Algorithm 6.12.

6.5.7 Solution Feasibility

Application of each of the genetic operators introduced above may lead to a spatial or temporal conflict. In order to guarantee the feasibility of each individual solution in the population, the procedure in Algorithm 6.13 recalculates the design schedule for the entire 3D floorplan.

Algorithm 6.11 Floorplan 'shaking' mutation

Require: parent, $p_{\text{m\_shake}}$ (mutation probability)

if toss_coin($p_{\text{m\_shake}}$) **then**

for all $v_i \in V$ **do**

$V_{tmp} \leftarrow V$

while $V_{tmp} \neq \emptyset$ **do**

choose integer j at random from interval $\langle 1, n_{V_{\text{tmp}}} \rangle$

Xdir = toss_coin()

Ydir = toss_coin()

calculate_valid_XY_range($\beta(v_j)$, Xdir, Ydir)

change (x, y) coordinates for $\beta(v_j)$ at random (within the valid range)

remove v_j from V_{tmp}

end while

end for

end if

Algorithm 6.12 Overall GA-based synthesis procedure.

Require: $G(V, E)$, $\lambda_{\max}$, T, n, $p_{\text{x\_bind}}$, $p_{\text{x\_XY}}$, $p_{\text{x\_XYZ}}$, $p_{\text{m\_bind}}$, $p_{\text{m\_XY}}$, $p_{\text{m\_XYZ}}$, $p_{\text{m\_shake}}$

1. Read $G(V, E)$ and analyse its dependencies

2. Initialise target technology device

3. Initialise GA:

P_{old} = initialise(G, T, n) {initial population}

F = evaluate_chromosome_fitness(P_{old})

4. Do the evolution:

while stopping criteria is not satisfied **do**

P_{new} = select_individuals(P_{old}, F)

P_{new} = apply_crossover(P_{new})

P_{new} = apply_mutation(P_{new})

F = evaluate_chromosome_fitness(P_{new})

end while

5. Store the best solution

s = select best individual(P_{new}, F)

extract 3D floorplan positions & reconfiguration schedule ρ_c from s

Algorithm 6.13 3D floorplan correction and reconfiguration latency calculation

Require: $B = V \cup E$, A, L (3D floorplan model)

1. Perform ASAP resource-constrained scheduling $\forall b_i \in B, a_i \in A$ so that (Eq. 4.4) is satisfied, while assuming $Setup(a_i) = 0$.
2. Calculate reconfiguration latency using a technology-specific procedure

$block\_list \leftarrow A$

while $block\_list \neq 0$ **do**

select available block a_i from $block\_list$ (selected at random if 2 or more are available)

$c_{a_i} \leftarrow$ calculate_reconfig_latency(a_i, L) {**Technology-specific**}

$\sigma, \rho \leftarrow$ update_design_schedules(a_i, c_{a_i}, σ, ρ, L)

remove a_i from $block\_list$

end while

First the design blocks are rescheduled using the 'as soon as possible' (ASAP) scheduling algorithm constrained to operate within the 3D floorplan. This removes any behavioural dependency violations between the modules positioned in the 3D floorplan. The ASAP scheduling affects only z-coordinates of the design modules.

In the second step, the actual reconfiguration latencies for all modules in the design are calculated. The procedure selects modules from the list of 'available'[1] modules and then calculates the reconfiguration latency for the selected module. A technology-specific function evaluates the actual reconfiguration latency. If two or more modules are available, one of the available modules is selected at random.

In the presented approach a simple greedy algorithm from the DYNASTY Framework's XC6200 technology server was used to evaluate the module reconfiguration latencies. The algorithm inserts additional system clock cycles for the violating modules until all conflicts in the design schedule are resolved. The algorithm uses an intermediate content of the configuration memory to evaluate the possibility of reusing previous configurations. Alternatively, a more sophisticated approach could have been used to explore specific features of the target technology (e.g. (Hauck et al., 1998)).

After the reconfiguration latency has been calculated, the cycle-accurate reconfiguration schedule ρ_c is updated and scaled to the execution schedule σ and configuration schedule ρ. This ensures that the accurate reconfiguration overheads are considered throughout the temporal floorplanning process.

[1]Design module is 'available' if and only if reconfiguration latency of all its predecessors was already calculated.

6.5.8 Problem-Specific Fitness Function

A simple fitness function extracts the overall execution latency of the generated solutions from the 3D floorplan. The fitness is calculated as the ratio of the desired execution latency to the latency extracted from the 3D floorplan.

Other design characteristics could be examined during the fitness evaluation, including the size of the configuration data, power consumption, device usability, etc. These are not considered here.

6.5.9 Selection of a Genetic Algorithm Procedure and Control Parameters

The simulated evolution is being controlled by a core steady-state genetic algorithm with tournament selection. A supplementary monitoring algorithm is implemented, which is used to control the frequency of application of the selected genetic operators. These can change probabilities dynamically in response to population convergence changes during the course of the evolution. The control function and the individual probabilities can be changed by the designer.

The overall strategy is to apply the operators which produce big changes (e.g. 3D floorplan crossover) in the design solution early during the evolution (when the population divergence is large). As the confidence in the generated solutions increases (observed as decreasing population divergence) the probability of the fine-tuning operators (e.g. 2D floorplan mutation) increases at the expense of operators producing big changes. The evolution terminates once a solution satisfying the design objectives has been generated or on command from the designer.

6.5.10 Implementation

The genetic algorithm based synthesis technique presented in this chapter was implemented using the MIT GAlib library (Wall, 1996) and the DYNASTY Framework (Chapter 5).

6.5.11 Summary

A new genetic algorithm based technique has been developed for a specific case of the complex problem of reconfigurable systems design. A problem-specific chromosome representation was constructed together with a set of problem-specific genetic operators.

The presented approach represents a combination of technology-specific heuristics responsible for ensuring the solution feasibility, and knowledge-based and problem-specific genetic algorithm manipulations within a design 3D floorplan.

In contrast to traditional applications of genetic algorithms, the chromosome problem representation is 'corrected' after the application of genetic operators in order to rectify possible conflicts or inefficiencies resulting from a different reconfiguration overhead distribution in the newly created individuals.

The correction procedure evaluates the reconfiguration overheads and reschedules the execution of the problem in order to guarantee the feasibility of each such potential design solution.

The correction procedure can completely change all absolute 3D floorplan coordinates. This, however, does not destroy the main design characteristics, which are determined by the relative placement of the design modules in the 3D floorplan. The relative 3D placement represents the module overlaps at fine-grained level, but also their relative execution de-

pendencies resulting from the schedule feasibility condition (Eq. 4.4).

A number of experiments were conducted using the presented synthesis technique in order to ascertain its capabilities. The results from these experiments are discussed in the following chapter.

Chapter 7

Experimental Results

The previous chapter has outlined a reconfigurable systems synthesis technique, which uses genetic algorithms to search for a solution in a complex, multi-dimensional search space. This chapter presents a selection of the experimental results used to assess the capabilities of the technique.

7.1 Benchmark Problems

No standard set of benchmarks exists for the evaluation of synthesis techniques for dynamically reconfigurable logic. Therefore, different methods for the evaluation of the qualities of previously reported techniques have been used. For example, Chatha and Vemuri (1999) use a single benchmark of a JPEG algorithm to explore the partitioning between software and reconfigurable hardware, Bazargan et al. (1999) use a set of random graphs to illustrate the capabilities of their 3D floorplanning technique, Lysaght and Stockwood (1996) and Luk et al. (1997b) use a simple pattern matcher circuit to demonstrate the capabilities of the low-level modelling features of their respective techniques, while Vasilko and Ait-Boudaoud (1996a) and Zhang et al. (2000) use a high-level synthesis benchmarks originally devel-

Benchmark	Graph	Source
Laplace filter operator	Fig. 7.1	(Heron and Woods, 1996)
Differential equation solver	Fig. 7.2	(Paulin et al., 1986)
Elliptic wave filter	Fig. 7.3	(Dewilde et al., 1985)

Table 7.1: Behavioural benchmarks used in the synthesis evaluation.

oped for non-reconfigurable systems.

In the evaluation of the approach presented in this thesis, the following requirements were placed on the selection of benchmark problems:

- all benchmarks should be behavioural design problems, which satisfy the restricted formulation presented in Section 6.1

- a reference design implementation designed by hand or other near-optimal method should exist for at least some of the benchmarks used. This will allow assessment of the quality of the synthesised results

- benchmarks should be of various sizes so that the presented synthesis technique can be exercised for problems of different sizes

- benchmarks should be composed of functions for which modules exist in the target technology library

Table 7.1 shows the set of behavioural design problems selected to demonstrate the performance of the presented synthesis technique (their data-flow graphs are shown in Figs 7.1–7.3). The implementation of these benchmarks using a dynamically reconfigurable system is considered here.

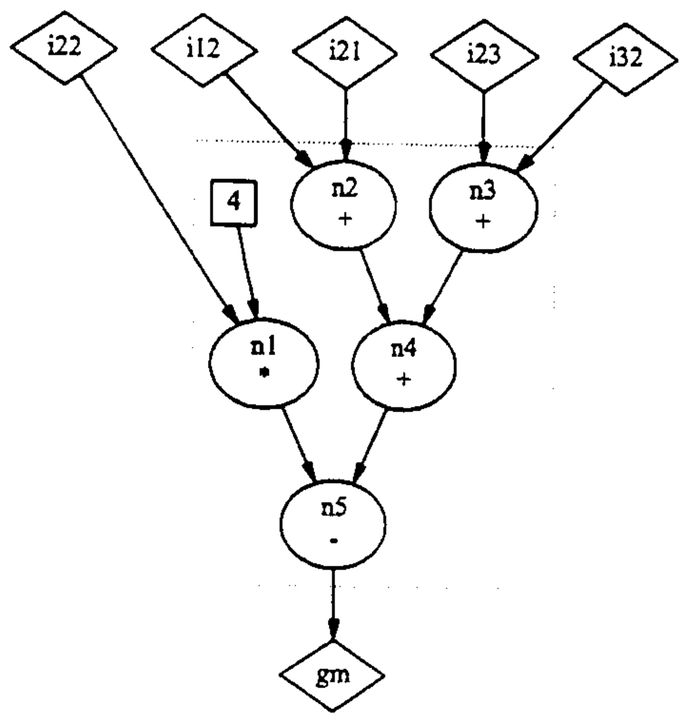


Figure 7.1: Laplace operator data-flow graph.

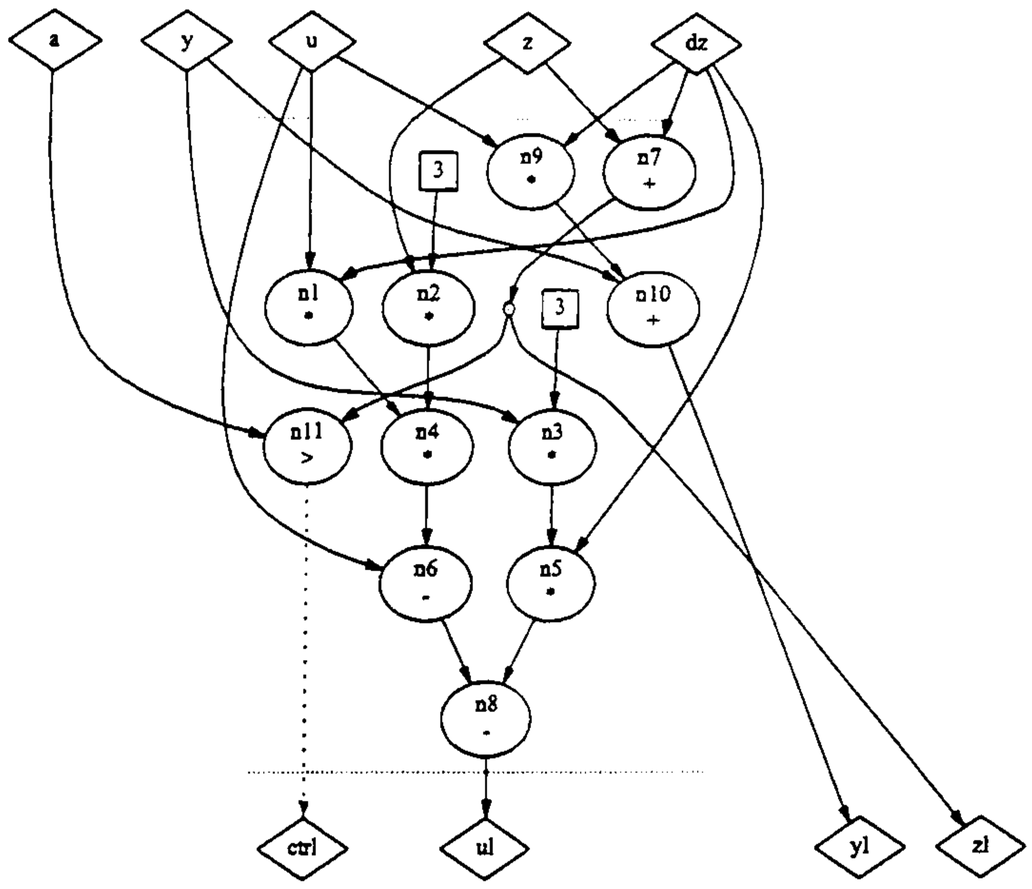


Figure 7.2: Differential equation solver data-flow graph.

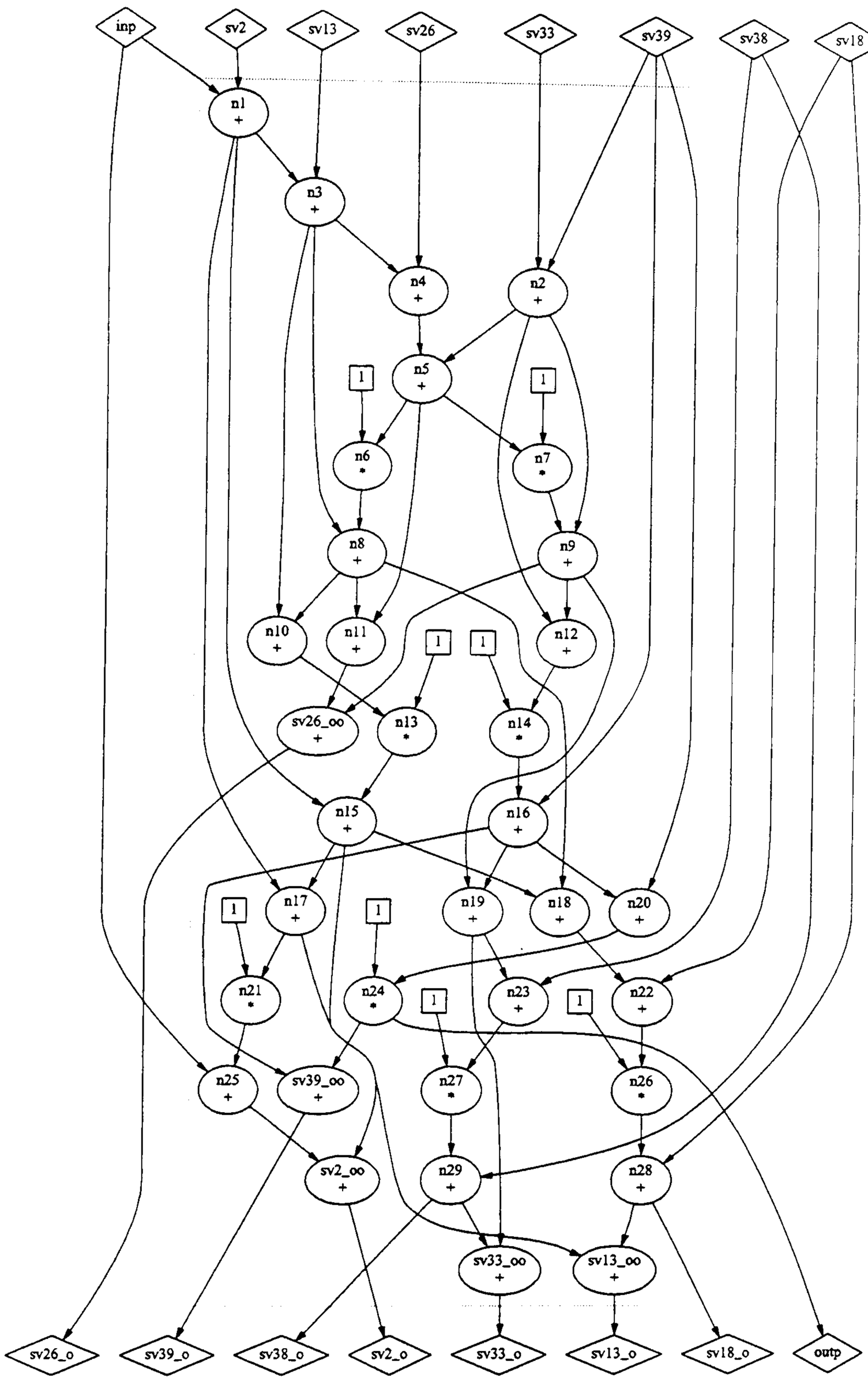


Figure 7.3: Elliptic wave filter data-flow graph.

7.2 Target Technology

A model technology for the implementation of the target RLU was used and implemented for this purpose in the DYNASTY Framework (Chapter 5). The technology is based on the Xilinx XC6200 series of partially reconfigurable FPGAs (Xilinx, 1997b), while the original technology was enhanced to include several different reconfiguration subsystems. Appendix A summarises the relevant details of the implementation of the model target technology.

The availability of different reconfiguration subsystems (Section A.2) allows comparison of the tradeoffs between the implementations using different types of subsystem within the same logic array architecture.

A small library of modules suitable for the experiments using the presented synthesis approach was derived from the existing XC6200 module libraries (Section A.3).

Note that because in the presented approach a design solution is constructed only from the modules available in the target technology module library, the efficiency of design implementation is limited by the implementation of the library modules. For example, the constant multiplication by 4 in the Laplace operator benchmark (Fig. 7.1) could be implemented using a simple shift operation. However, because only a full A*B multiplier is available in the target technology module library (Table A.1), the multiplication will be implemented using this multiplier module.

7.3 Experimental Procedure

Each of the benchmark problems was synthesised using the algorithm presented in Chapter 6. A selection of target reconfigurable device array sizes

Module	Operation	Relative latency = $\lceil \frac{T_{\text{module}}}{T_{\text{sys\_clk}}} \rceil$
ADD	+	1
SUB	−	1
GTN	>	1
MULT	*	3

Table 7.2: Relative module latencies used during the synthesis of examples.

have been used. Library modules with 8-bit operands were used for all of the presented experiments. All experiments were conducted using the model XC6200 technology with a 8-bit parallel random access configuration subsystem (Section A.2).

A system clock cycle of 50 ns was used. The functional unit module latencies (shown in Tables A.2–A.5 in Appendix A) scaled to the system clock cycle of 50 ns are shown in Table 7.2.

The configuration clock cycle was assumed to be equal to the system clock cycle ($T_{\text{config\_clk}} = T_{\text{sys\_clk}}$).

The genetic algorithm synthesis technique was run until the population converged and no further improvement to the best design solution in the population was found.

The following configuration subsystems of the targeted model XC6200 technology (Section A.2) were used to compare the algorithm performance with different types of configuration mechanisms:

- 8-bit parallel random access (original Xilinx XC6200)
- pre-loaded multiple contexts (similar to MIT DPGA)

After automatic synthesis, the benchmark design implementations were sample tested for cycle-accurate functionality using the following method.

7.3.1 Design Verification

A simulation model for each benchmark design was created in order to verify the correctness of the design reconfiguration schedule. A 'clock morphing' (CM) simulation model (Vasilko and Cabanis, 1999) has been used for this purpose.

The simulation model provides behavioural models for all design modules. These are connected to a global reconfiguration controller (RCU) model via separate clock signals (Fig. 7.4). The RCU implements the design reconfiguration schedule. During the simulation the module reconfiguration is modelled by assigning special signal values to the module clock signals. A direct connection of the system clock signal to the module clock signal indicates that the module is active ($n1$ and $n4$ in Fig 7.4). Clock signals of inactive design modules ($n2$, $n3$ and $n5$ in Fig 7.4) are driven to special 'V' values, which indicate that these modules are in *virtual* states.

The CM simulation model was preferred to other approaches to reconfigurable system simulation because of its flexibility and the capabilities to highlight the problems with design reconfiguration scheduling and resource sharing.

Only one VHDL model had to be written for each benchmark design. In order to simulate the various design solutions, the design modules in the simulation model are annotated with their respective spatial floorplan positions during the simulation. There is no need to recompile or regenerate a new simulation model for each design solution.

The functionality of the simulation model required to support the virtual state propagation has been implemented in a modified version of the IEEE `std_logic_1164` VHDL package. Details of this implementation can be found in (Vasilko and Cabanis, 1999).

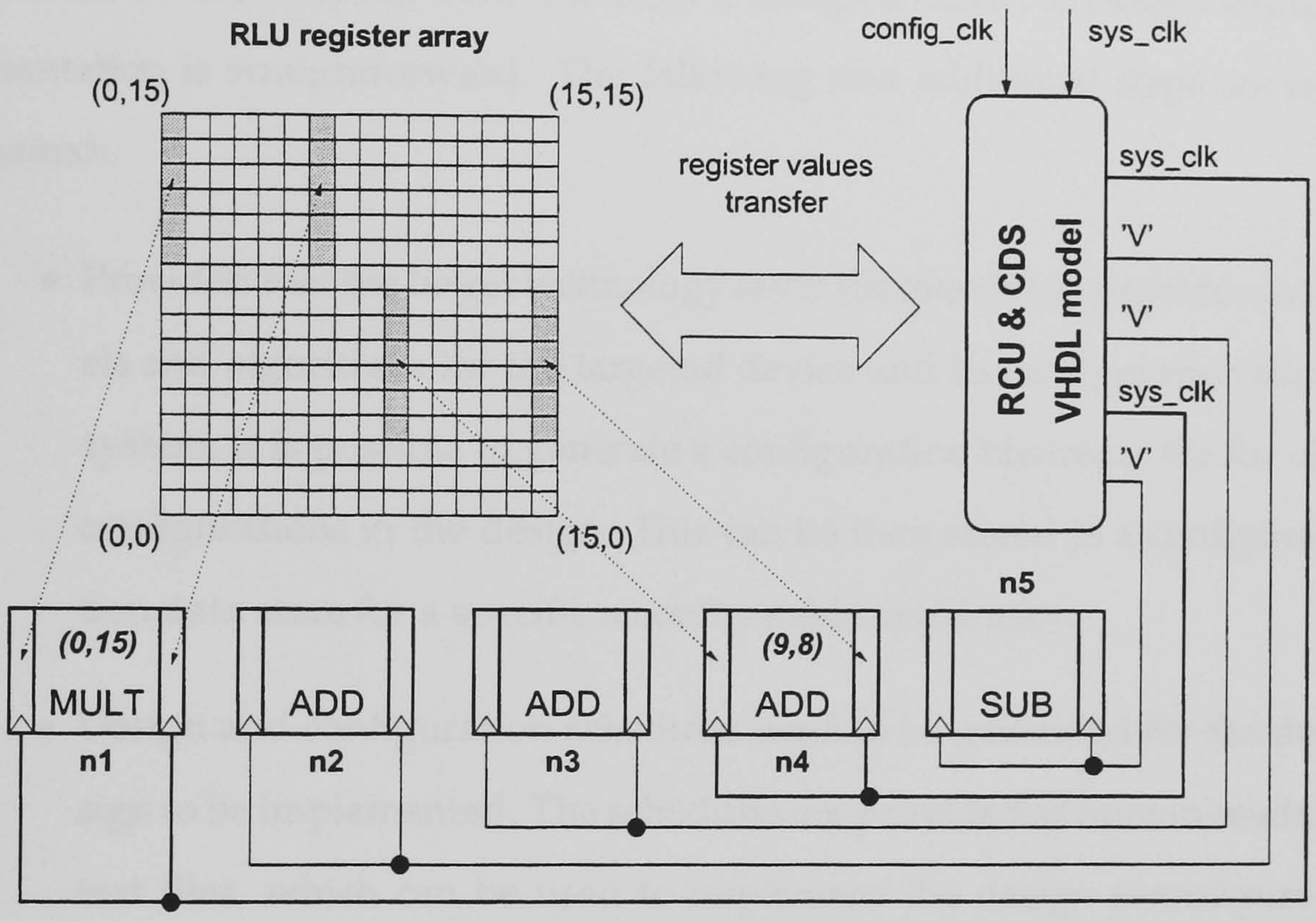


Figure 7.4: An example of a CM-based simulation model used during verification (Laplace operator design). The modules $n1$ and $n4$ are active, positioned at specific coordinates in the RLU and their registers are connected to the RLU array registers. Inactive modules are disconnected from the RLU register array by driving their respective clock inputs to the *virtual* state 'V'. The operation of the entire system is controlled by an RCU model.

A small set of test vectors was used to confirm the functionality of the reconfigurable design implementation.

7.3.2 Design Implementation

While the results from the experiments in this chapter were not implemented on real FPGAs, the route from a design solution to design implementation is straightforward. The following two additional steps are required:

- Provided that the target technology sever (Section 5.1.3) provides models and algorithms for the targeted device and its configuration subsystem, it is possible to generate a configuration bitstream file for all configurations in the design. This can be then stored in a configuration data store for a specific reconfigurable application.
- Design and configuration schedules need to be generated for the design to be implemented. The schedules are provided as human-readable text files, which can be used to implement the design reconfiguration controller in either hardware or software. The configuration controller synthesis is not directly supported by the DYNASTY Framework (see Section 5.1.6).

7.4 Summary of Results

Tables 7.3–7.4 display a selection of results generated using the benchmarks from Table 7.1. These were selected as best results out of specific 10 runs of the synthesis algorithm. The following symbols are used in the following tables (Fig. 7.5):

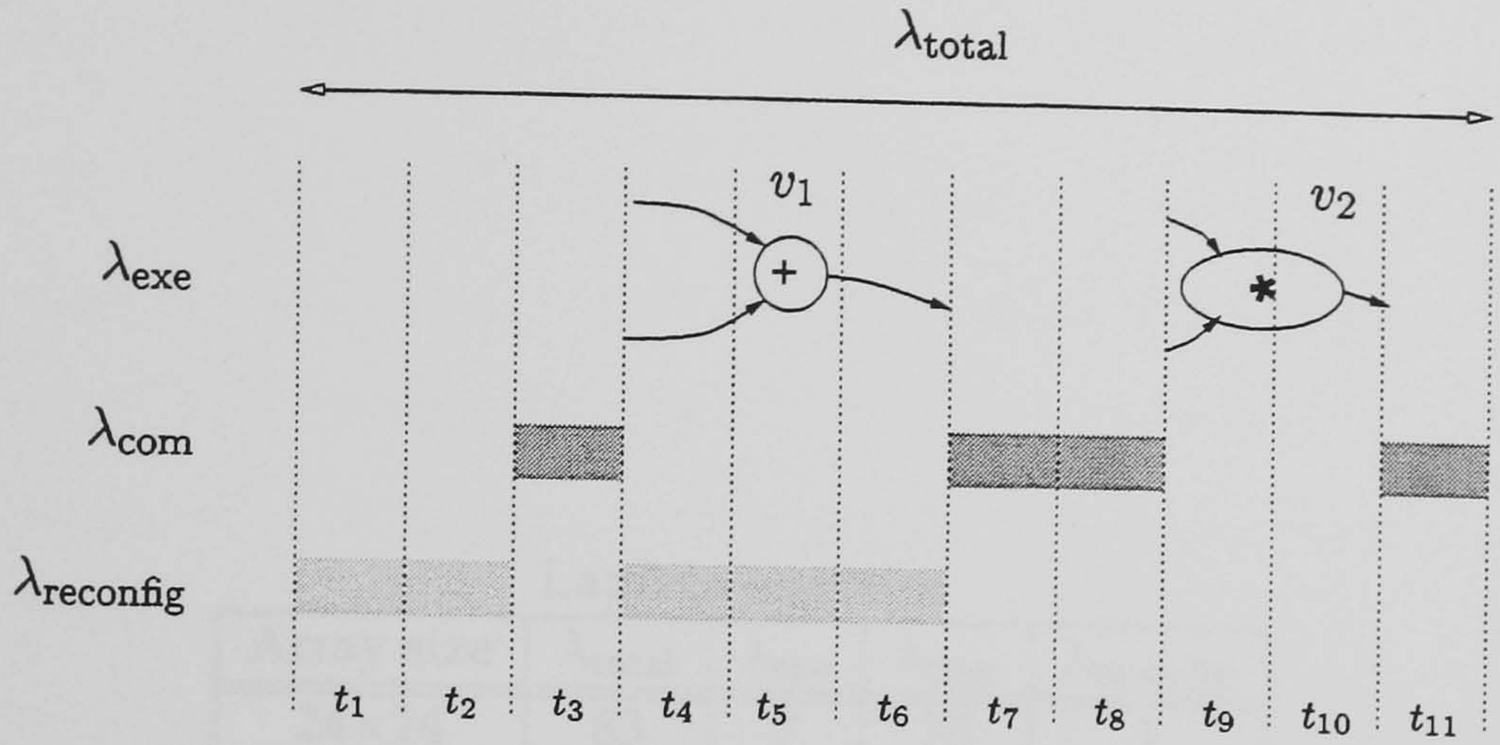


Figure 7.5: Design schedule example. $\lambda_{exe} = 3$ (t_5, t_9, t_{10}), $\lambda_{com} = 4$ (t_3, t_7, t_8, t_{11}), $\lambda_{reconfig} = 5$ (t_1, t_2, t_4, t_5, t_6).

Laplace operator

Array size	λ_{total}	λ_{exe}	λ_{com}	$\lambda_{reconfig}$
24×24	979	7	75	897
32×32	82	7	75	0
48×48	82	7	75	0
64×64	82	7	75	0

Differential equation

Array size	λ_{total}	λ_{exe}	λ_{com}	$\lambda_{reconfig}$
24×24	1337	23	336	978
32×32	1303	23	336	944
48×48	1281	23	336	922
64×64	359	23	336	0

Elliptic wave filter

Array size	λ_{total}	λ_{exe}	λ_{com}	$\lambda_{reconfig}$
24×24	1584	50	564	970
32×32	1747	50	564	1133
48×48	1663	50	564	1049
64×64	1782	50	564	1168

Table 7.3: Synthesis results for an 8-bit parallel random access configuration subsystem (XC6200).

Laplace operator

Array size	λ_{total}	λ_{exe}	λ_{com}	$\lambda_{reconfig}$
24×24	83	7	75	1
32×32	82	7	75	0
48×48	82	7	75	0
64×64	82	7	75	0

Differential equation

Array size	λ_{total}	λ_{exe}	λ_{com}	$\lambda_{reconfig}$
24×24	364	23	336	5
32×32	366	23	336	7
48×48	361	23	336	2
64×64	360	23	336	1

Elliptic wave filter

Array size	λ_{total}	λ_{exe}	λ_{com}	$\lambda_{reconfig}$
24×24	627	50	564	13
32×32	626	50	564	12
48×48	620	50	564	6
64×64	618	50	564	4

Table 7.4: Synthesis results for multiple contexts configuration subsystem (DPGA).

λ_{exe} is the number of system clock cycles spent executing the individual computations in the design

λ_{com} is the number of system clock cycles spent transferring RLU register values from/to the RLU (used for the transfer of arguments and retrieval of the results of computations implemented in the RLU). The data transfer can be performed in parallel with execution of the computations.

$\lambda_{reconfig}$ is the number of system clock cycles spent reconfiguring the RLU. The RLU configuration can be performed in parallel with execution of computations.

λ_{total} denotes the total design execution latency as the number of system clock cycles required to complete the entire design computation. As the register and configuration data transfers can be performed in parallel with the computations, the following relationship holds

$$\lambda_{total} \leq \lambda_{exe} + \lambda_{com} + \lambda_{reconfig} \tag{7.1}$$

An example of the results for the Laplace operator benchmark over 10 runs of the synthesis algorithms is shown in Fig. 7.6. For each run, the initial population was generated using a different random seed. This demonstrates that genetic algorithms cannot guarantee that an identical solution will be found with the same design problem and constraints. Close examination of the generated results reveals that the displayed variation is due to the algorithm's inability to share one adder resource for all add operations (some solutions provide 1 adder block while others provide 2) and also the adder and subtractor resources did not fully overlap in some cases (an example of a solution with 2 adder blocks, 1 subtractor and 1 multiplier is shown in Fig. 7.7(b)).

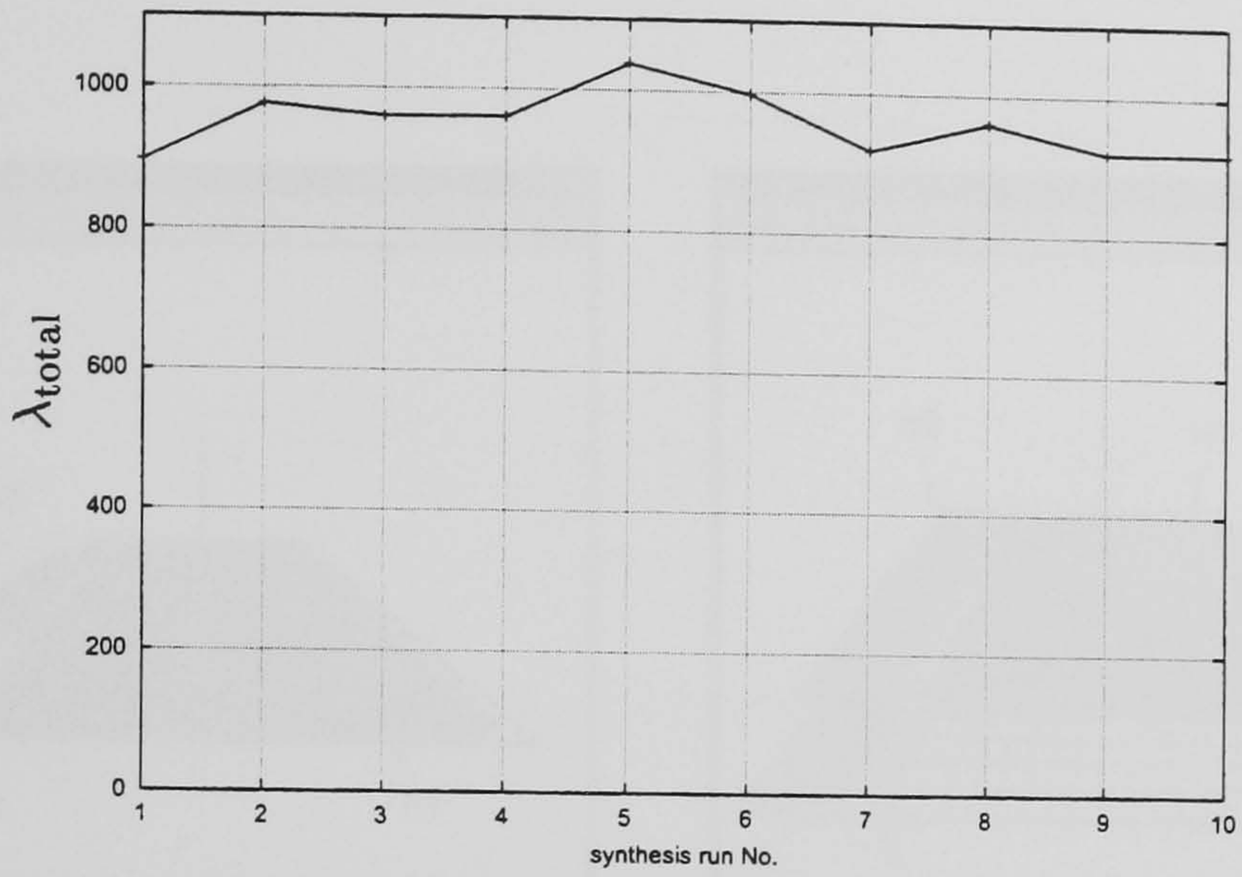
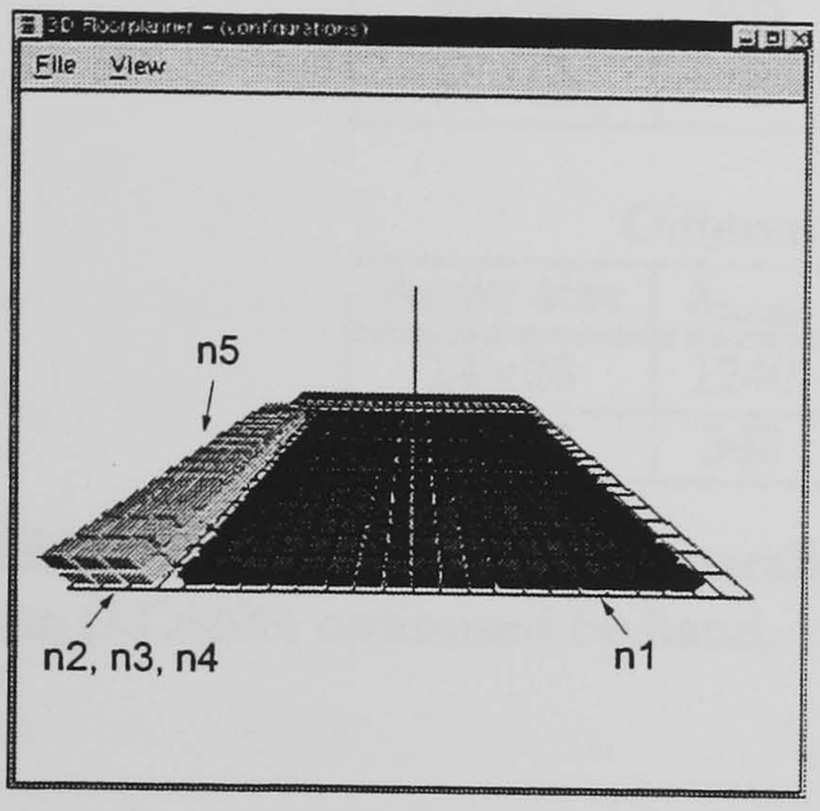


Figure 7.6: Solution stability over 10 GA-synthesis runs (Laplace operator benchmark, 24 × 24 array, 8-bit parallel random access configuration subsystem).

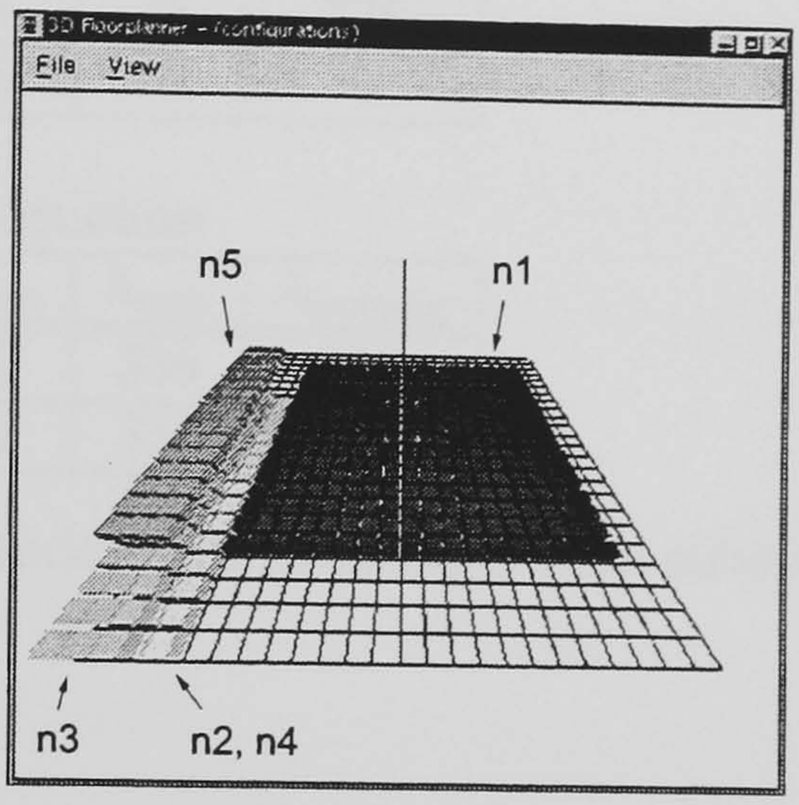
The effect of a multiple-level resource sharing can be also observed in Fig. 7.7. The design optimised by hand (Fig. 7.7(a)) requires only 3 modules (adder, subtractor and multiplier), while an example of the automatically generated solution (Fig. 7.7(b)) requires 4 modules (1 additional adder) due the inability of the synthesis algorithm to fully overlap all 3 adder operations.

Architectural-level resource sharing allows for 2 behavioural add operations (n3 and n4) to be bound to a single adder module (Fig. 7.7(b)). Physical-level resource sharing allows for the subtractor (n5) to share routing and logic resources with this adder module. The effects of resource sharing from both architectural and physical levels are combined in the design 3D floorplan.

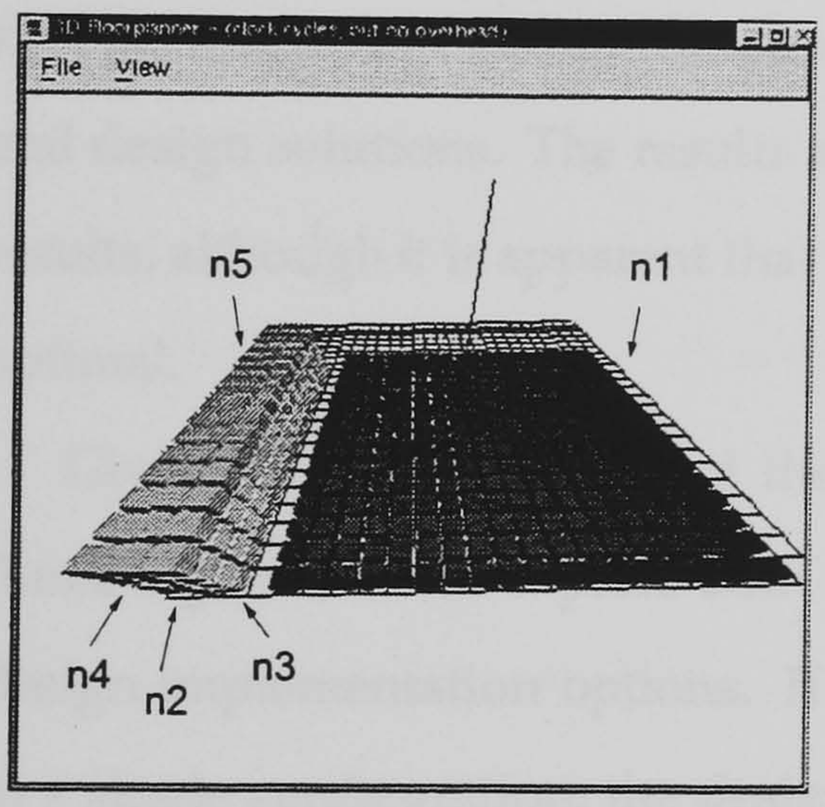
Some of the designs presented above were optimised by hand to provide the most efficient implementation given the size of an array, module library and an input design problem. These results are summarised in Ta-



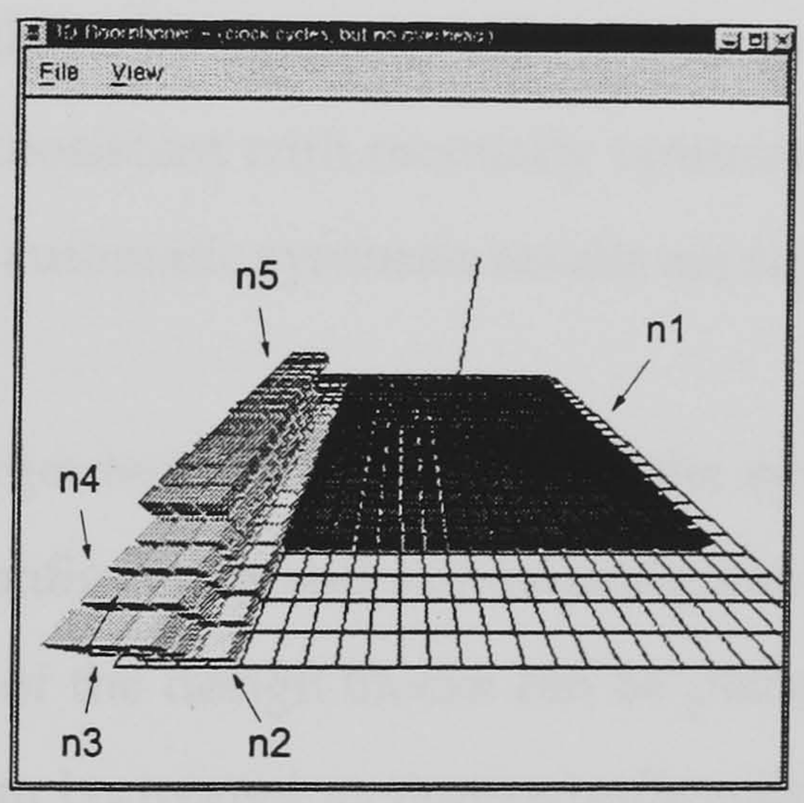
(a) optimised by hand (modules)



(b) automatically synthesised (modules)



(c) optimised by hand (schedule)



(d) automatically synthesised (schedule)

Figure 7.7: Comparison of a manually constructed design solution with a design obtained automatically (Laplace operator benchmark, 24 × 24 array, 8-bit parallel random access configuration subsystems, no configuration cycles are shown). (a)-(b) show the placement of the design modules, (c)-(d) show the design execution schedule.

Laplace operator

Array size	λ_{total}	λ_{exe}	λ_{com}	$\lambda_{reconfig}$
24×24	803	6	75	724
64×64	78	5	75	0

Differential equation

Array size	λ_{total}	λ_{exe}	λ_{com}	$\lambda_{reconfig}$
24×24	1240	23	336	888
64×64	342	8	336	0

Table 7.5: Results for an 8-bit parallel random access configuration subsystem (XC6200) optimised by hand.

ble 7.5.

The results demonstrate that given the size of the target technology device (technology resource constraint) and the desired design latency (design performance constraint), the synthesis technique is able to find several design solutions. The results are consistent with manually optimised results, although it is apparent that the automatic synthesis results are sub-optimal.

Given the performance and the target technology constrains the synthesis algorithm will explore both reconfigurable and non-reconfigurable design implementation options. If all of the design blocks can be placed in a single configuration, the design can be treated as non-reconfigurable (represented by $\lambda_{reconfig} = 0$).

The results from using the XC6200-compatible reconfiguration interface (Fig. 7.3) demonstrate that the large overhead required for the reconfiguration of the design modules ($\lambda_{reconfig}$) and communication of design data (λ_{com}) results in a very inefficient design implementations.

The results from using the reconfiguration subsystem with pre-loaded multiple-context configuration memory (Fig. 7.4) show that although the

reconfiguration latency is negligible, the communication latency (λ_{com}) limits the performance of the generated design implementations.

The imbalance between λ_{exe} and $\lambda_{com}/\lambda_{reconfig}$ (Tables 7.3–7.5) suggests that if a reconfigurable design implementation using the above configuration subsystems is to be efficient, the design modules should spend much longer time in executing useful computation than in configuration or communication. In this context, a reconfigurable design is more efficient when the reconfiguration of design modules can be performed while others are executing useful computations. Thus long reconfiguration and communication latencies could be amortised over many execution cycles. Alternatively, if the design implementation can operate with a clock period much longer than that of configuration clock, the reconfiguration and communication overhead could be accommodated during the system clock period.

One difficulty with using genetic algorithms for the optimisation of complex problems is that the algorithm runtime may vary considerably between optimisation runs. For the experiments presented in this chapter, the synthesis algorithm usually converged to a final solution within only several minutes on a PentiumIII/400MHz PC, while runtime increased with larger problems. However, in some cases the synthesis algorithm runtime exceeded one hour and the synthesis had to be terminated, while accepting the current best solution.

An advantage of genetic algorithms over other global optimisation techniques is that a large pool of different solutions is generated early in the algorithm execution. If it is not necessary to search for the most optimal solution (for example when it is only necessary to access the suitability of the target technology for the implementation of the design problem), it is possible to terminate the synthesis early and use the best solution gener-

ated so far. Although the quality of such a solution is low, this may provide sufficient indication of the implementation feasibility, while the algorithm runtime is greatly reduced.

For the practical implementation of the presented synthesis algorithm the runtime must be improved. Several opportunities exist here, including the combination of GA with knowledge-based heuristics and other optimisation techniques (such as simulated annealing). These would allow genetic algorithm to converge faster once a global minimum is thought to be found, but also could lead to more optimal results.

Chapter 8

Conclusions

This chapter summarises the contribution of the presented work and outlines the areas for further improvements. Possible future directions of this research are outlined at the end of this chapter.

8.1 Summary of the Contribution

This thesis has presented a new formulation for the problem of synthesis for dynamically reconfigurable logic systems. A new synthesis and optimisation algorithm working on a restricted formulation of this problem was presented to demonstrate the feasibility of a special case of the proposed model.

The features provided by the presented problem formulation were discussed at the end of Chapter 4 and are summarised below:

- *A generic framework for the construction of various synthesis/compilation techniques.* The problem formulation allows encapsulation of a variety of techniques for reconfigurable system synthesis. Given the constraints and assumptions about the targeted technology and the im-

plementation methodology, a specific problem instance can be characterised using this formulation (as demonstrated in Section 6.1).

Synthesis and optimisation techniques can then operate within such a problem instance. For example, design for non-reconfigurable systems can be treated within this framework as a special instance of the problem model (Eq. 4.5). Another example is the technique presented in Chapter 6, which assumes compile-time synthesis and emphasises detailed analysis of the reconfiguration overheads. Other techniques aiming to achieve different synthesis objectives at either compile-time or run-time can be characterised through a definition of a specific instance of this general problem.

- *Multiple-level resource sharing.* The presented formulation allows for resources in the design to be shared at both architectural and fine-grained physical level. It is therefore possible to consider the reduction of the reconfiguration overhead not only through sharing of architectural modules between identical behavioural operations, but also through sharing of primitive physical resource configurations between the modules implementing *different* behavioural operations.

The synthesis technique presented in Chapter 6 provides an example of a technique operating on an instance of the problem characterised by the assumptions in Section 6.1. The following are the main features of this technique:

- *Solution feasibility guarantee.* As the approach combines the exploration of low-level design characteristics (layout position, reconfiguration overhead, etc.) with the high-level considerations (design scheduling, resource allocation and binding) it can guarantee that

the implementation of the synthesised solution will be feasible. This, however, is possible only if the reconfiguration overhead calculation function provides an estimate of the reconfiguration latency with sufficient accuracy.

The solution feasibility ensures that no design flow iterations are required in order to generate an implementation which will operate on the selected target technology. However, design iterations while selecting different target technologies or performance constraints may be required if the implementation with the originally chosen target technology cannot satisfy the original performance constraints.

- *Tradeoff analysis between reconfigurable and non-reconfigurable design implementation.* As there is no *a priori* assumption that the design implementation has to be reconfigurable, the optimisation technique can freely explore design implementations for these two options. Whether the design solution will operate as a reconfigurable or non-reconfigurable system depends on the selected target technology and the design constraints.

- *Technology independence.* The temporal floorplanning process, together with a model of a 3D floorplan, provides an abstraction which allows design optimisation for many common reconfigurable logic technologies. The technology architecture abstraction offered by the 3D floorplan model allows for complex technology-specific features to be 'hidden' from the synthesis technique, while these features can be exploited through the availability of a suitable technology server. The technology-specific considerations (such as the availability of specific library modules, specific technology resources or features, reconfiguration overhead calculation procedures, etc.) can be then 'plugged-

in' to the synthesis technique. Therefore it can be expected that many other reconfigurable logic technologies, such as Xilinx Virtex, Atmel AT40k and others, can be used within such a framework.

Furthermore, the results from the synthesis experiments presented in Chapter 7 appear to provide the expected results. Although no exact analysis of the optimality of these solutions has been conducted, in several known cases the results are consistent with previous hand implementations.

8.1.1 Applications of the Proposed Approach

The synthesis technique presented in Chapter 6 can be used in several applications:

- *Synthesis for reconfigurable systems.* As was demonstrated in Chapter 7, the developed technique can be used to produce working reconfigurable implementations. The practicality of these implementations depends greatly on the features provided by the target technology. Contemporary commercial reconfigurable logic devices suffer from large reconfiguration overheads, which makes reconfigurable design implementations practical only (i) when system reconfiguration does not occur very often or (ii) when a relatively long system clock cycle can accommodate many configuration cycles, thus reducing the impact of the reconfiguration overhead on the design latency. Future improvements may provide technologies with faster reconfiguration times and therefore make frequent reconfiguration a more practical design option.

- *Reconfigurable technology architecture/features analysis.* The approach

can be also used to evaluate the suitability of a specific reconfigurable logic architecture, reconfiguration subsystem and other technological features for the implementation of design problems from a specific application domain. Given an input set of typical design problems it would be possible to synthesise their implementation over a set of variations of the target technology. A domain-specific reconfigurable logic technology could be then derived from such experiments.

- *Retargetable compilation for reconfigurable computing platforms.* Many different reconfigurable computing platforms and technologies have been developed for use in software acceleration. The design of a reconfigurable implementation for a specific algorithm and a specific type of reconfigurable computing platform can be difficult, especially when partial reconfiguration is considered as a part of this process. Even if a high-level languages such as VHDL, C or Java were used to implement the algorithm on a specific reconfigurable platform, the implementation of the reconfiguration-related functionality will be *specific* to a given reconfigurable computing platform. Porting such an application to a different platform may involve a complete redesign of such algorithm hardware implementation (consider for example porting an algorithm implementation from a Xilinx XC6200 FPGA to Xilinx Virtex FPGA based platform).

The synthesis technique presented in Chapter 6 offers a technology-independent model of the synthesis/compilation process. Technology-specific architecture and reconfiguration capabilities are provided through a technology server. For a compiler targeted to a reconfigurable computing platform, the technology server can be represented as a part of the target platform architecture model. If such a model is pro-

vided for each targeted reconfigurable computing platform, the porting of an algorithm to these platforms involves mere re-compilation (re-synthesis) of the algorithm with a different architectural model.

The guaranteed feasibility of the implementation will ensure that an input algorithm will operate on the target reconfigurable computing platform, while the actual latency of the computation will depend on its speed and reconfiguration features.

8.2 Areas for Improvement and Future Directions

The presented synthesis technique limits the category of technologies (as detailed in Section 6.1) and applications which can be considered using this approach. Some of the possible areas for improvement are identified below:

8.2.1 Composite Cost Function

While the synthesis technique presented in Chapter 6 uses latency as a measure of quality for the generated solutions, the fitness evaluation may include further considerations. These might include constraints on the size of the storage available for the configuration and application data, constraints on power consumption, constraints on the individual placement and scheduling of the design modules, and others.

8.2.2 Evaluation with Large and Multi-cycle Modules

The synthesis examples presented in Chapter 7 provide only a small set of benchmarks, limited to data-flow problems with behavioural graph elements being primitive arithmetic computations. It is necessary to bench-

mark the performance of this technique using larger problems with complexity approaching that of industrial applications. While primitive arithmetic operations of data-flow graphs provide many opportunities for architectural exploration, without the ability to synthesise finite-state machines and route these primitive operations, it will not be possible to construct reconfigurable system design implementations at such a fine-grained level.

It is foreseeable that the presented technique will be useful for synthesis from other system-level models. For example, task graphs are being used successfully in hardware/software co-design for reconfigurable systems (e.g. (Chatha and Vemuri, 1999)). Each task in the model can be represented by a complex hardware module. In this scenario, the presented algorithm should be able to explore any similarities between the configuration in order to produce systems which share primitive reconfigurable technology resources. This remains a topic for further investigation.

8.2.3 Routing Consideration

In the presented approach all data transfers between the architectural design modules are performed via registers. It would be desirable to evaluate the opportunities for direct wire routing between the modules in a reconfigurable array. Routing may reduce the overheads required for the transfer of the computational data and also offer more implementation options.

While this is a desirable feature, routing for reconfigurable systems is a very difficult problem. Routing algorithms have not only to select the routing paths with limited reconfigurable routing resources, but also consider the impact of the routing on future re-configuration and the impact of the overhead due to configuration of routing switches on the overall execution latency.

Furthermore, direct routing in a reconfigurable system may not always be a desirable option. If the design implementation reconfigures frequently, it might be preferred to transfer register data via the configuration interface, rather than reconfigure many distributed routing resources. For example, a 32-bit data word in the Xilinx XC6200 FPGA technology can be transfered via its configuration interface in only 1 configuration interface cycle (in the best case). The number of configuration interface cycles required for the configuration of routing resources for a 32-bit wired bus depends on the position of the source and destination of the data transfer and the availability of routing resources. This latency may be considerably longer than that of an equivalent register value transfer via the configuration interface. Such tradeoffs must be considered during the routing process.

The routing problem is a part of physical synthesis and can be viewed within the framework presented in Chapter 4. The contribution of the routing configuration and the delay is considered as a part of the *Setup*() function in Eq. 4.4.

In the presented synthesis technique, the routing could be considered as a part of the configuration correction procedure presented in Section 6.5.7. Routing techniques, however, must provide a worst case estimate of the impact of the routing on the execution latency so that the solution feasibility could be maintained.

8.2.4 Architectural-Level Resource Sharing

The presented technique has considered only one specific type of resource sharing at the architectural level (assumption 5 in Section 6.1). With the availability of routing algorithms suitable for dynamically reconfigurable systems it will be possible to consider architectural-level resource sharing

in scenarios when other components (e.g. FSM and its control logic) need to be placed and routed into the design.

8.2.5 Register Allocation, Pipelining and Retiming

If the synthesis approach can consider the routing problem, it will also be possible to consider different arrangements for the distribution of registers and data transfers in the design. This opens possibilities for the exploitation of different approaches aimed at the improvement of throughput and resource usage of the design implementation.

Furthermore, the possibilities for design module pipelining and their reconfiguration through 'pipeline morphing' (Luk et al., 1997c) could be considered as an extension to the presented approach.

While various design optimisation techniques could be applied to the presented synthesis method, there are several options how these could be implemented: (i) provided as new genetic algorithm operators, (ii) included as a part of the 3D floorplan correction routine (Algorithm 6.13), (iii) included into a fitness calculation routine, or (iv) provided as a pre- or post-processing algorithms. Further work is needed to establish suitable implementations of various optimisation techniques in the context of the presented synthesis method.

8.2.6 Summary

The presented technique allows automatic design synthesis for a category of reconfigurable systems. The result of such a synthesis is a reconfigurable system implementation represented as a set of configuration data for run-time reconfiguration of the RLU and a schedule for the control of this reconfiguration process from the RCU.

Many problems in the design automation for reconfigurable systems remain unsolved. While all areas of improvement listed in Section 8.2 should receive attention in the future, its is the routing problem for reconfigurable systems which demands high priority. Understanding of routing in reconfigurable systems and its impact on the system performance, will allow for future synthesis approaches to consider wire routing as a central part of the reconfigurable system synthesis.

Improvements to the presented synthesis method are necessary to improve its efficiency and speed, and to include more advanced optimisation transformations. Furthermore, the interdependence between the reconfigurable architectures, technologies, modelling and design tools needs to be studied to improve our understanding of reconfigurable systems and their applications.

The synthesis results using the model XC6200 technology demonstrate some of the limitations of the current reconfigurable technologies. While better reconfigurable technologies are needed, these have to be developed to consider the targeted application domain. Furthermore, the reconfigurable technology features should be developed in conjunction with the design tool development to ensure that the efficient algorithms can be constructed which will use the technology features efficiently.

In many practical systems where reconfigurable logic is considered as one of the implementation options, the system will include a combination of software implemented on an embedded processor(s), fixed function hardware and reconfigurable hardware. In order to explore the high-level partitioning of the system's functionality between these three options, the system-level tools need to estimate the expected performance of each target technology option for various partitioning scenarios.

The presented technique for the design of reconfigurable systems should integrate with other design tools operating at a system-level. Such an integration would allow system-level designers to consider reconfigurable logic implementation as one of the options equivalent to those of fixed hardware and processors, but providing different implementation trade-offs.

Appendix

Appendix A

Model Reconfigurable Logic Technology

A Xilinx XC6200 FPGA (Xilinx, 1997b) based model reconfigurable logic technology was used in the experiments presented in this thesis. To differentiate between this model technology and the original Xilinx XC6200 technology, the model technology is referred to as a *model (XC6200) technology*, while the *original (XC6200) technology* denotes the technology as originally developed by Xilinx.

The model XC6200 technology was implemented as a technology server in the DYNASTY Framework (Chapter 5). The implementation includes:

- logic arrays of various sizes
- a set of libraries, including primitive technology cell libraries and macro libraries
- various configuration subsystems and supporting configuration latency estimation algorithms
- a low-level device simulation model

The model XC6200 technology does not implement the full set of features available in the original XC6200 technology. While preserving the overall architecture and the basic functionality of the original configuration interface, the model technology was further enhanced to include other reconfiguration subsystems, which are not available in the original XC6200 technology.

This appendix highlights the features and incompatibilities of the model XC6200 technology as used in this thesis. Further detailed description of the original XC6200 technology and its configuration interface can be found in the relevant literature (Xilinx, 1997b; Churcher et al., 1995).

A.1 Architecture

A compatible subset of the original XC6200 architecture was implemented in the model XC6200 technology, thus allowing for many of the circuits developed for the original XC6200 technology to be implemented in the model technology. The aim of the architectural implementation was to preserve the architectural features required by the XC6200 module library, but also to maintain functionality similar to the original XC6200 technology, so that the newly developed design techniques could use a realistic target technology model.

A.1.1 Device Size

The original Xilinx XC6200 technology provides a selection of devices with array sizes of 48×48, 64×64, 96×96 and 128×128. The model XC6200 technology implementation provides additional devices of other sizes, including 8×8, 16×16, 24×24, 32×32. While these smaller devices are probably not useful for practical real-world applications (other than small coproces-

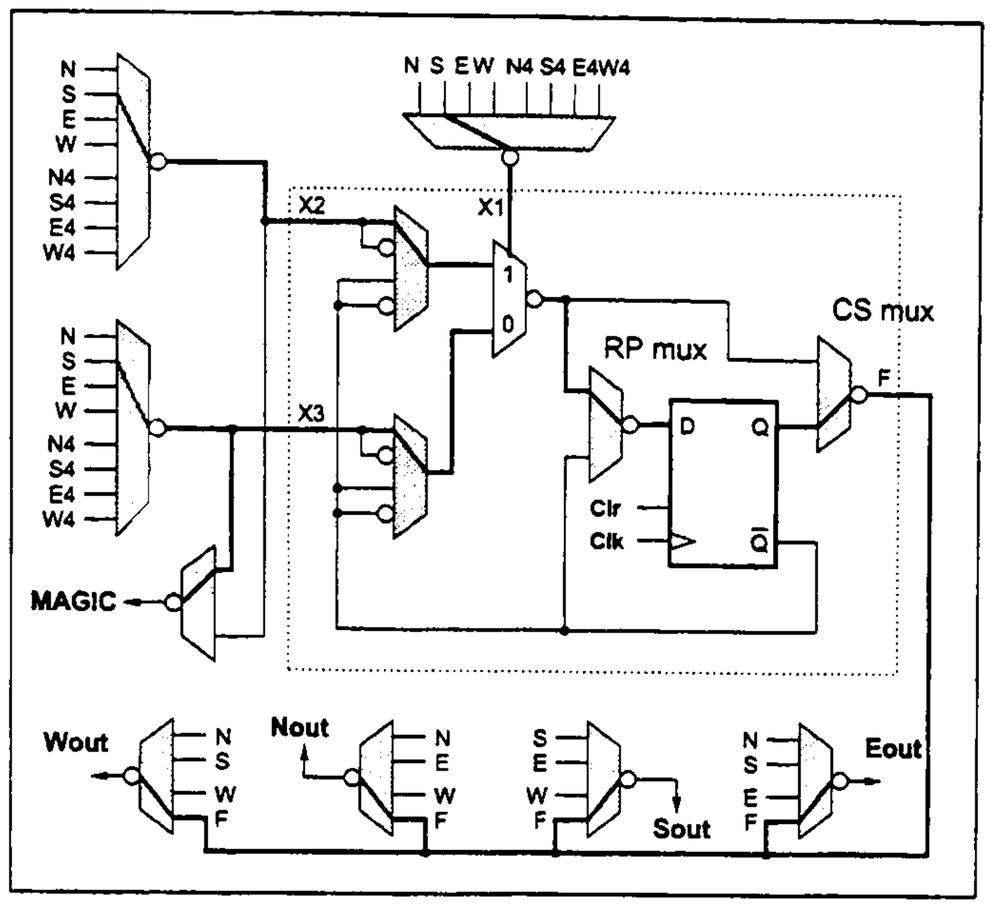


Figure A.1: XC6200 logic block. The configuration multiplexers (shown in grey) are in their default states after the device reset.

sors or reconfigurable ALUs), these devices were used in the experiments presented in Chapters 2 and 7 to enforce tighter resource constraints for small design problems.

A.1.2 Logic Block

Xilinx XC6200 logic block architecture (including its functional unit and the local routing multiplexers) is depicted in Fig. A.1. An identical logic block was implemented in the model XC6200 technology.

The XC6200 logic block provides two 2-input look-up tables with outputs connected to a 2:1 multiplexer, one D flip-flop, and a number of routing multiplexers. The special RP multiplexer can disable access to the D flip-flop from within the array, thus allowing for the flip-flop value to be read/written only via the XC6200 configuration interface.

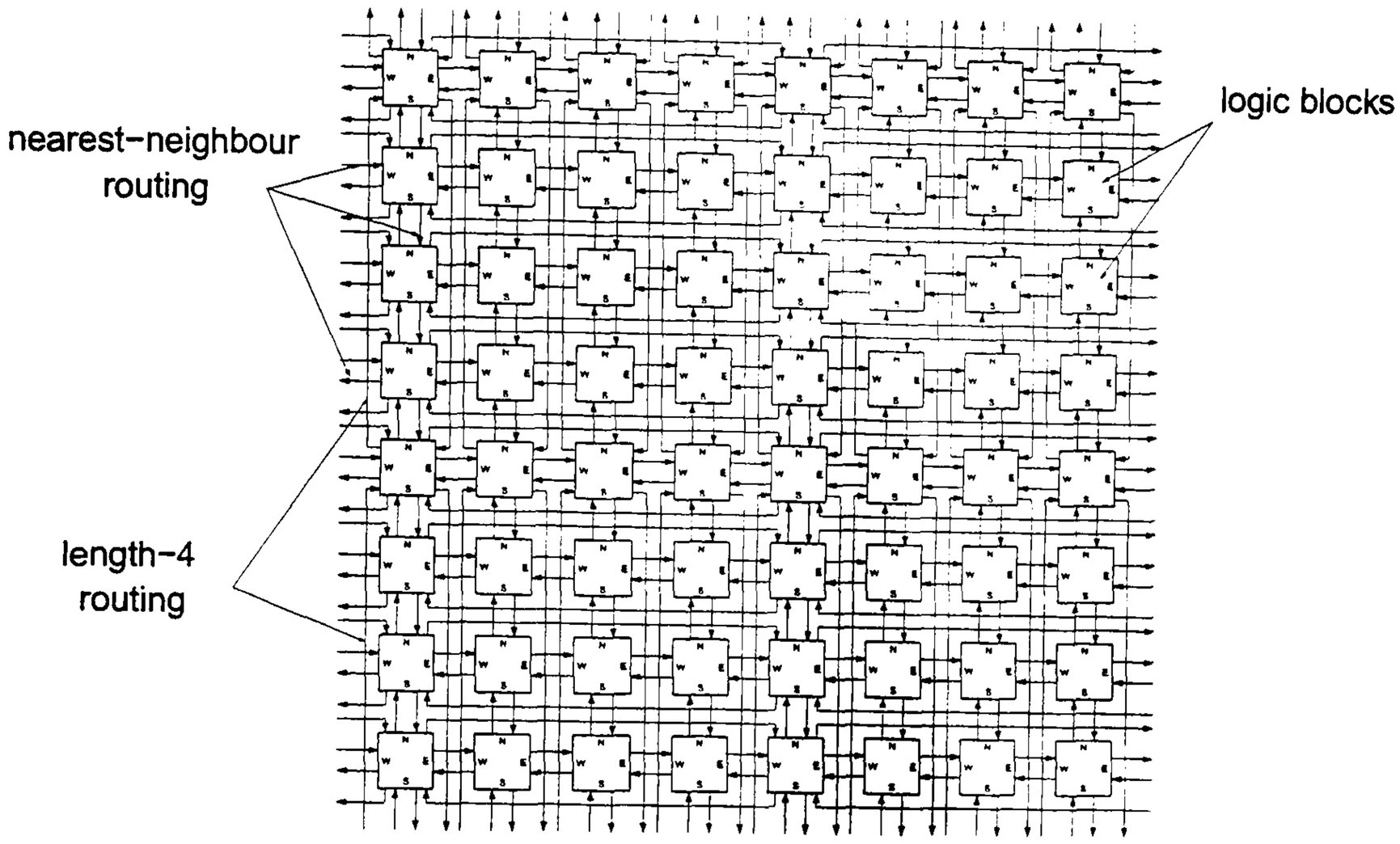


Figure A.2: Model XC6200 technology logic array (not all length-4 connections are shown).

A.1.3 Routing Resources

Only the nearest-neighbour and length-4 interconnections were provided in the model XC6200 technology. While other longer interconnections from the original XC6200 technology (length-16 and chip-length) could have been easily implemented, this was unnecessary as the modules in the model XC6200 parametrised library do not use interconnections other than nearest-neighbour and length-4. As no inter-module routing is used in the approach presented in this thesis, the longer routing resources would remain unused.

An example of the implemented model XC6200 logic array is shown in Fig. A.2.

The model XC6200 technology does not provide any input/output blocks. The routing lines connecting to these blocks in the original XC6200 technology were left unconnected in the model XC6200 technology. Also MAGIC

routing connections originating in the logic blocks (Fig. A.1) were left unconnected.

A.2 Configuration Subsystem

The original XC6200 technology provides a combined serial and parallel configuration interface, with parallel random-access configuration data distribution and one-to-one activation mechanism[1].

The model XC6200 technology was enhanced to provide several configuration subsystems with different configuration speed. These were provided to facilitate testing of new synthesis algorithms while targeting technologies with different configuration subsystems. The following configuration subsystems were implemented as typical representatives of the current trends:

1. A subset of the original Xilinx XC6200 configuration subsystem. Only the parallel configuration interface is provided; 8, 16 and 32-bit configuration data words are supported. The configuration distribution does not support the wildcard feature available in the original XC6200 technology. Also many of the original configuration memory locations are inactive due to lack of the corresponding configurable resources.

2. A Xilinx Virtex-like configuration subsystem (Xilinx, 2000a) providing frame-based configuration data distribution. The individual frames are aligned with the device columns. There are 3 frames for each column. The frame length depends on the device size (8-bit control and address word + configuration data):

[1]This categorisation of reconfigurable technologies was introduced in Section 2.2.1 on page 17.

$$8 + 8 \times \text{logic blocks per 1 column} \quad \text{[bits]}$$

The subsystem further provides a serial configuration interface and direct one-to-one configuration activation.

3. A subsystem with a multiple-context configuration memory (similar to MIT DPGA (Brown et al., 1994)) providing an unlimited number of layers (many-to-one configuration activation). The memory contexts can be accessed through the original XC6200 parallel random-access distribution mechanism and the parallel configuration interface.

In all of the above cases, the access to the device cell register values is assumed to be through the original XC6200 configuration interface.

Reconfiguration latency calculation algorithms have been provided for each of the above configuration subsystems. Given a design solution represented as a 3D floorplan these calculate the number of reconfiguration cycles required for the design implementation using the selected configuration subsystem.

A.3 Library Modules

The model XC6200 technology uses a library of parametric modules derived from the original XC6200 macro library (Luk et al., 1997a; Xilinx, 1998).

A selection of arithmetic modules shown in Table A.1 was adopted for the model technology library. All model XC6200 technology library modules support signed numbers (negative numbers are represented in a 2's complement system). The modules were provided with input and output registers to allow data transfer from and to the modules via the configuration interface.

Name	Function	Description
ADD	A + B	signed ripple-carry adder
SUB	A − B	signed ripple-carry subtractor
GTN	A > B	signed greater than comparator
MULT	A * B	signed multiplier

Table A.1: A selection of XC6200 library modules used in experiments described in Chapter 7.

In the original XC6200 macro library, the relative placement of logic blocks within each macro module was fixed using the positional constraints, while the detailed routing was performed by the Xilinx XACT6000 place & route tool. In the model XC6200 technology module library, each module was fully routed when the library was constructed. This ensures that the configuration of all routing multiplexors and look-up tables in the module is known during the synthesis process. Therefore the reconfiguration latency, which considers the configuration of both logic and routing resources, can be calculated accurately.

As only the local and length-4 routing wires were used for the module routing, the modules can be positioned at any array location (if only the local routing wires were used) or locations with coordinates which are multiple of 4 (when the length-4 wires were used).

Examples of fully placed and routed modules for 4-bit adder and subtractor modules are shown in Figs. A.3–A.4 and Figs. A.5–A.6 respectively. Detailed configuration data for these modules can be derived from the configuration of logic and routing resources shown in Figs. A.4 and A.6.

The summary of module characteristics for adder, subtractor, comparator and multiplier modules are shown in Tables A.2–A.5.

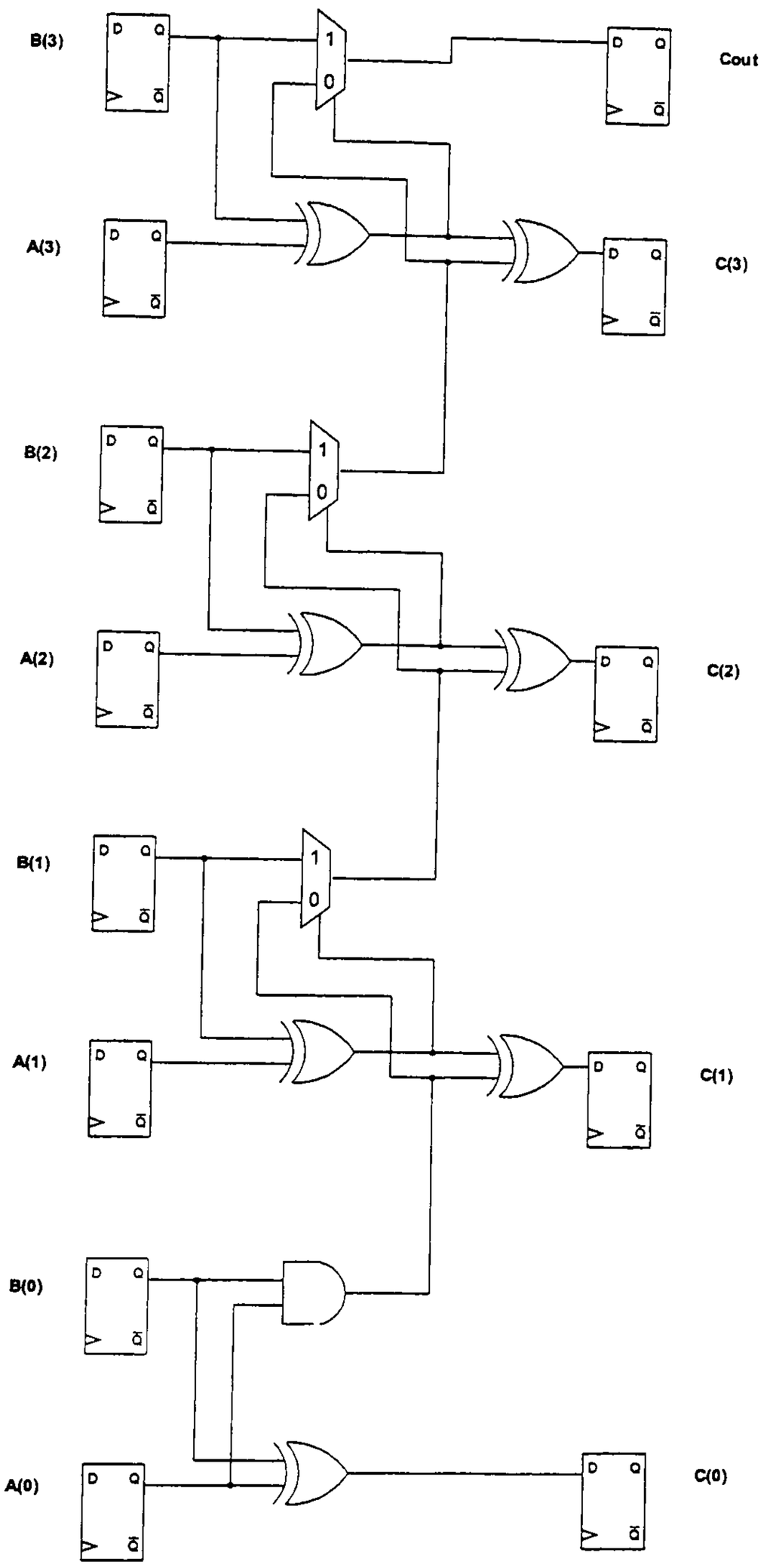


Figure A.3: 4-bit adder $(a + b)$: schematic diagram.

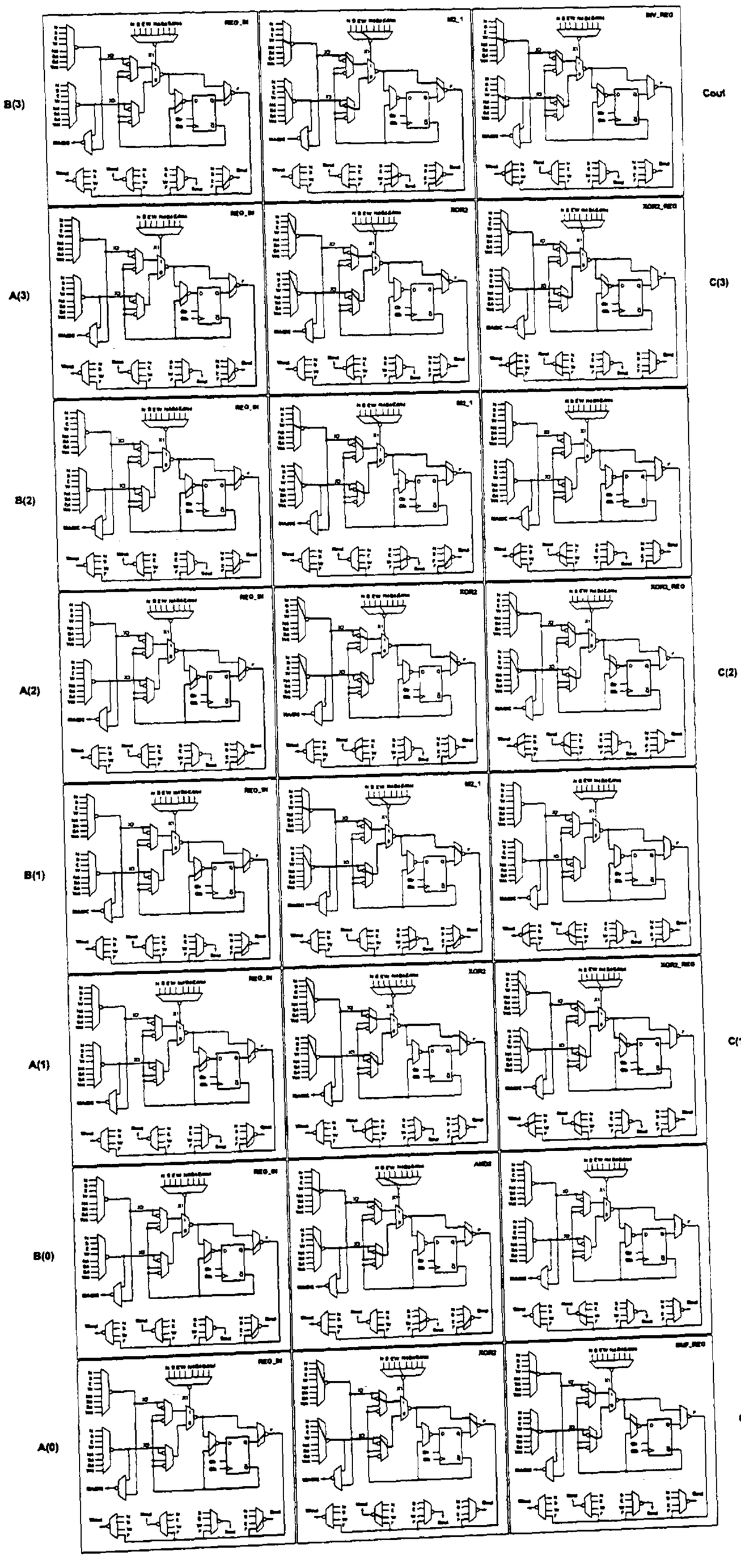


Figure A.4: 4-bit adder $(a + b)$: detailed layout.

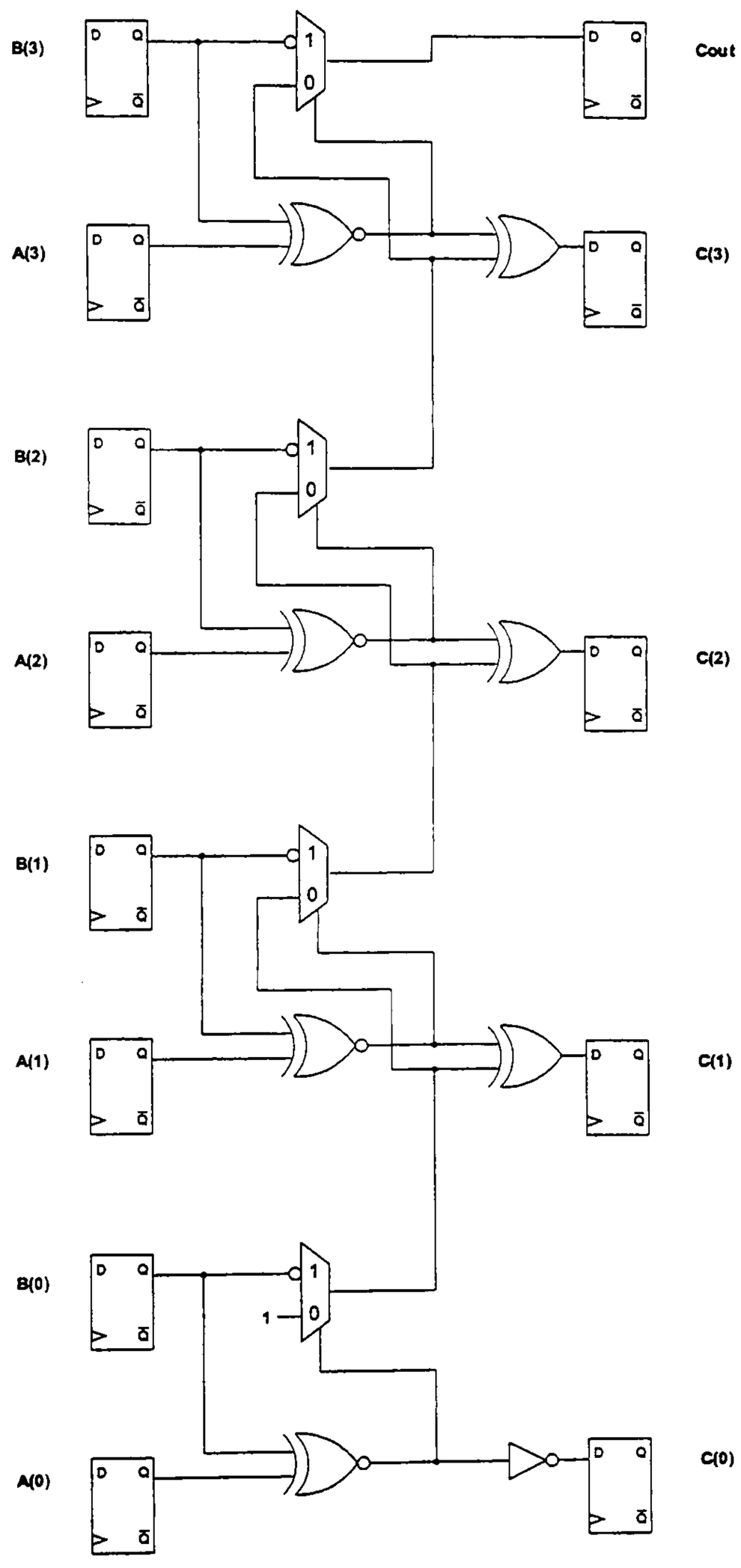


Figure A.5: 4-bit subtractor ($a - b$): schematic diagram.

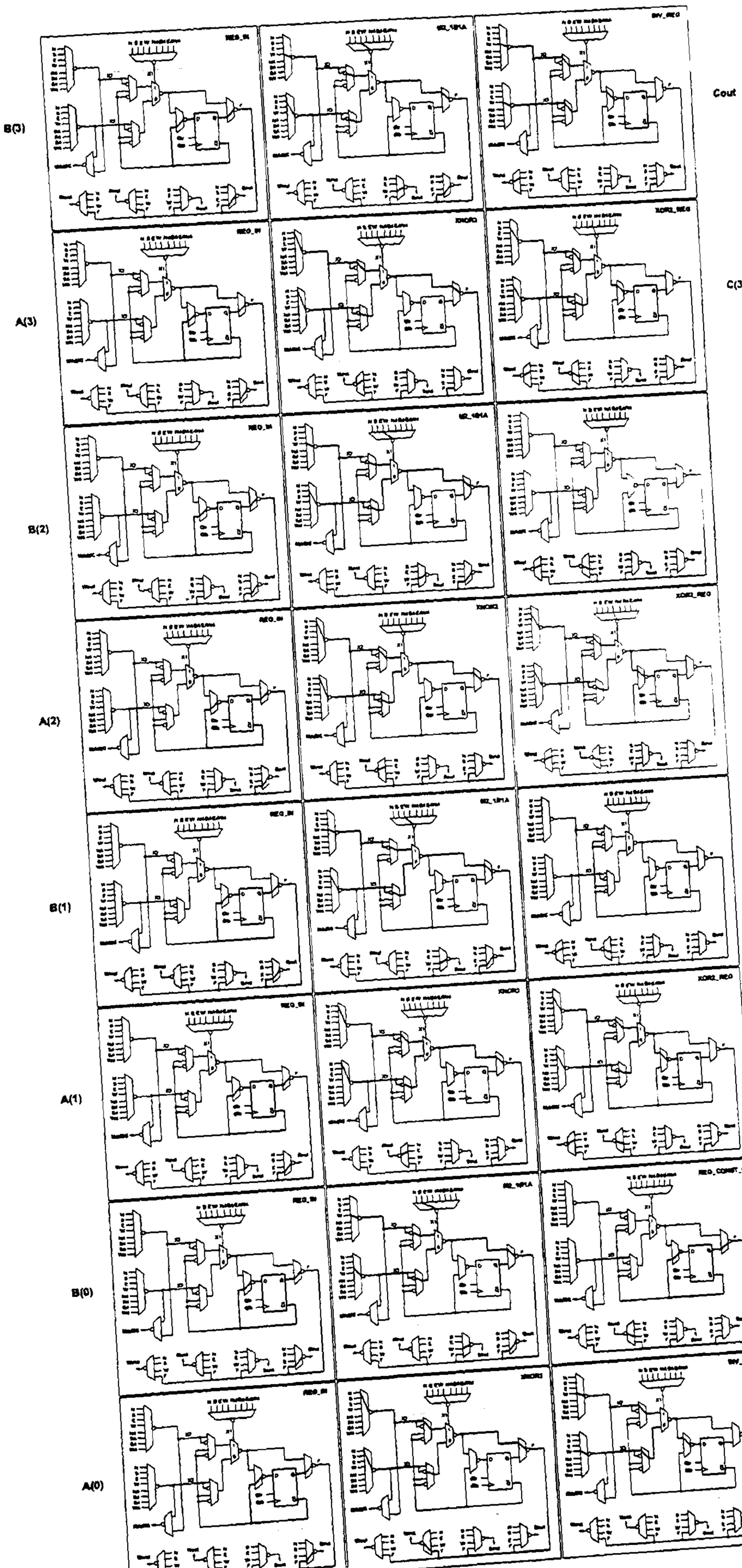


Figure A.6: 4-bit subtractor $(a - b)$: detailed layout.

	4-bit ADD	**8-bit ADD**
size	3 × 8	3 × 16
execution latency	17 ns	37 ns
data retrieval latency (in/out)	4/2 c/cycles	4/2 c/cycles

Worst-case configuration latency (in c/cycles)		
Configuration interface	**4-bit ADD**	**8-bit ADD**
8-bit XC6200	65	129
16-bit XC6200	44	80
32-bit XC6200	27	45
frame-based (Virtex)	2376	2376
multiple contexts (DPGA)	1	1

Table A.2: Characteristics for 4-bit and 8-bit adder modules.

	4-bit SUB	**8-bit SUB**
size	3 × 8	3 × 16
execution latency	22 ns	42 ns
data retrieval latency (in/out)	4/2 c/cycles	4/2 c/cycles

Worst-case configuration latency (in c/cycles)		
Configuration interface	**4-bit SUB**	**8-bit SUB**
8-bit XC6200	67	131
16-bit XC6200	44	80
32-bit XC6200	27	45
frame-based (Virtex)	2376	2376
multiple contexts (DPGA)	1	1

Table A.3: Characteristics for 4-bit and 8-bit subtractor modules.

	4-bit GTN	**8-bit GTN**
size	3 × 8	3 × 16
execution latency	22 ns	44 ns
data retrieval latency (in/out)	4/2 c/cycles	4/2 c/cycles

Worst-case configuration latency (in c/cycles)		
Configuration interface	**4-bit GTN**	**8-bit GTN**
8-bit XC6200	64	136
16-bit XC6200	41	77
32-bit XC6200	27	45
frame-based (Virtex)	2376	2376
multiple contexts (DPGA)	1	1

Table A.4: Characteristics for 4-bit and 8-bit 'greater than' comparator modules.

	4×4-bit MULT	**8×8-bit MULT**
size	10 × 9	18 × 17
execution latency	62 ns	141 ns
data retrieval latency (in/out)	13/14 c/cycles	25/26 c/cycles

Worst-case configuration latency (in c/cycles)		
Configuration interface	**4-bit MULT**	**8-bit MULT**
8-bit XC6200	204	790
16-bit XC6200	124	442
32-bit XC6200	84	246
frame-based (Virtex)	7920	14256
multiple contexts (DPGA)	1	1

Table A.5: Characteristics for 4×4-bit and 8×8-bit multiplier modules.

A.4 Support for Design Verification

A low-level VHDL model of the device architecture was implemented for the model XC6200 technology server. The model can be used to verify the validity of the configuration data produced by the design synthesis.

The VHDL device model provides a model of the configuration interface and a structural netlist of primitive device blocks (look-up tables, routing multiplexers, wires, etc.). The individual primitive blocks are modelled at behavioural level. The configuration of all primitive elements can be read from a configuration text file during the VHDL simulation. The configuration text file can be changed by another program (e.g. another simulator or a debugger), which allows co-simulation of software and reconfigurable hardware system components.

Glossary

compile-time refers to time during program (or design) compilation (or synthesis). *Compile-time* is an opposite to *run-time*.

configurable is used here as a synonym for *(in-field) programmable*.

dynamically reconfigurable refers to a quality of a system of being changed during its own operation as opposed to requiring the system power-down. Note that the term *dynamic* is used here to denote a temporal quality only. *Dynamic reconfigurability* does not automatically imply partial reconfigurability, which is a spatial characteristic. *Dynamically reconfigurable* is a synonym for *run-time reconfigurable*.

Note that currently two interpretations of this terminology are in common use. The above interpretation is consistent with Lautzenheiser (1986) and other early works in this field.

The other interpretation of *dynamically reconfigurable* implies both temporal and spatial characteristics (e.g. (Lysaght and Dunlop, 1994)), i.e. systems which can be changed during their operation and *partially*.

dynamically reconfigurable logic is a logic system, which is dynamically reconfigurable.

full reconfiguration refers to a type of reconfiguration, when only the en-

tire reconfigurable system can be modified in one configuration. *Full reconfiguration* is an opposite of *partial reconfiguration*.

(in-field) programmable refers to a quality of a system (or a device, circuit, sub-system, etc.) such that the system's configuration can be changed away from the system vendor's manufacturing facility.

partial reconfiguration refers to a type of reconfiguration, which permits that only a portion of the reconfigurable system is modified. Depending on the context where this term is used, the 'system' may include a single reconfigurable logic device, but also a complex reconfigurable system with several reconfigurable logic devices, and other components. *Partial reconfiguration* is an opposite of *full reconfiguration*.

reconfigurable system is a system which can be configured more than once.

run-time is time during system's operation. *Run-time* is an opposite to *compile-time*.

run-time reconfigurable is a synonym of *dynamically reconfigurable*.

References

Albaharna, O. T., Cheung, P. and Clarke, T. (1994). Virtual hardware and the limits of computational speed-up, In: *IEEE International Symposium on Circuits and Systems*, pp. 159–162.

Algotronix (1991). *CAL1024 Datasheet*, Algotronix Ltd. Version 004.

Altera (1995). *1995 Data Book*, Altera Corporation, 2610 Orchard Parkway.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities, In: *Proc. AFIPS 1967 Spring Joint Computer Conference*, Atlantic City, NJ, April, pp. 483–485.

Atmel (1994). *Configurable Logic Design and Application Book 1994/1995*, Atmel Corporation, 2125 O'Nel Drive, San Jose, CA, 95131.

Bazargan, K., Kaster, R. and Sarrafzadeh, M. (1999). 3-D floorplanning: Simulated annealing and greedy placement methods for reconfigurable computing systems, In: *Proceedings of the IEEE Workshop on Rapid System Prototyping (RSP'99)*, Clearwater, FL, USA, June 16–18.

Brebner, G. (1996). A virtual hardware operationg system for the xilinx XC6200, In: R. W. Hartenstein and M. Glesner (editors), *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers (FPL '96 Proceedings)*, LNCS 1142, Springer-Verlag, pp. 327–336.

Brebner, G. (1997). The swappable logic unit: a paradigm for virtual hardware, In: *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, USA, April 16–18, pp. 77–86.

Brebner, G. and Bergmann, N. (1999). Reconfigurable computing in remote and harsh environment, In: P. Lysaght, J. Irvine and R. Hartenstein (editors), *Field-Programmable Logic and Applications*, LNCS 1673, Springer-Verlag, Glasgow, UK, August 30–September 1, pp. 195–204.

Brown, J., Chen, D., Eslick, I., Tau, E. and DeHon, A. (1994). DELTA: Prototype for a first-generation dynamically programmable gate array, *Transit Note 112*, MIT Artificial Intelligence Laboratory.

Cantó, E., Moreno, J. M., Cabestany, J., Faura, J. and Insenser, J. M. (1999). A bipartitioning algorithm for dynamic reconfigurable programmable logic, In: P. Lysaght, J. Irvine and R. Hartenstein (editors), *Field-Programmable Logic and Applications*, LNCS 1673, Springer-Verlag, Glasgow, UK, August 30–September 1, pp. 134–143.

Chang, D. and Marek-Sadowska, M. (1998). Partitioning sequential circuits on dynamically reconfigurable FPGAs, In: *International Symposium on Field Programmable Gate Arrays*, Monterrey, CA, February, pp. 161–167.

Chatha, K. S. and Vemuri, R. (1999). Hardware-software codesign for dynamically reconfigurable architectures, In: P. Lysaght, J. Irvine and R. Hartenstein (editors), *Field-Programmable Logic and Applications*, LNCS 1673, Springer-Verlag, Glasgow, UK, August 30–September 1, pp. 175–184.

Churcher, S., Kean, T. and Wilkie, B. (1995). The XC6200 FastMap^{TM} processor interface, In: W. Moore and W. Luk (editors), *5th International Work-*

shop on Field Programmable Logic and Applications, LNCS 975, Springer-Verlag, Oxford, UK, August 29–September 1, pp. 36–43.

Culbertson, W. B., Amerson, R., Carter, R. J., Kuekes, P. and Snider, G. (1997). Defect tolerance on the teramac custom computer, In: *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA (USA), April, pp. 116–123.

De Micheli, G. (1994). *Synthesis and Optimisation of Digital Circuits*, McGraw-Hill.

DeHon, A. (1996). E-mail conversation.

Dewilde, P., Deprettere, E. and Nouta, R. (1985). Parallel and pipelined VLSI implementation of signal processing algorithms, In: S. Kung, H. Whitehouse and T. Kailath (editors), *VLSI and Modern Signal Processing*, Prentice Hall, pp. 257–264.

Diessel, O., ElGindy, H., Middendorf, M., Schmeck, H. and Schmidt, B. (2000). Dynamic scheduling of tasks on partially reconfigurable FPGAs, *IEE Proceedings Computer and Digital Techniques* 147(3): 181–188.

Dodhi, M. K., Hielschner, F. H., Storer, R. H. and Bhasker, J. (1995). Datapath synthesis using a problem-space genetic algorithm, *IEEE Transactions on CAD of Integrated Circuits and Systems* 14(8): 934–944.

Edwards, C. (2000). Vax all, folks, *Electronics times*. September 11, pp. 42–44.

Eldredge, J. G. and Hutchings, B. L. (1994). RRANN: A hardware implementaion of the backpropagation reconfigurable FPGAs, In: *Proceedings of the IEEE World Conference on Computational Intelligence*, IEEE, Orlando, Florida, June, pp. 77–80.

Estrin, G. (1960). Organization of computer systems—the fixed plus variable structure computer, In: *Proc. Western Joint, Computer Conference*, San Francisco, CA, USA, May 3–5, pp. 33–40.

French, P. C. and Taylor, R. W. (1993). A self-reconfigurable processor, In: D. A. Buell and K. L. Pocek (editors), *IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE Comput. Soc. Press, Napa, CA, USA, April 5–7, pp. 50–59.

GajjalaPurna, K. M. and Bhatia, D. (2000). Temporal partitioning and scheduling data flow graphs for reconfigurable computers, *IEEE Transactions on Computers* 48(6): 579–590.

Gajski, D. D., Dutt, N. D., Lu, A. C. and Lin, S. Y. (1992). *High-level synthesis: Introduction to Chip and System Design*, Kluwer Academic Pubishers.

Gerez, S. H. (1999). *Algorithms for VLSI Design Automation*, John Wiley & Sons.

Gokhale, M. and Marks, A. (1995). Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays, In: W. Moore and W. Luk (editors), *5th International Workshop on Field Programmable Logic and Applications*, LNCS 975, Springer-Verlag, Oxford, UK, August 29–September 1, pp. 399–408.

Gokhale, M., Holmes, W., Kopser, A., Lucas, S., Minnich, R. and Sweely, D. (1991). Building and using a highly parallel programmable logic array, *IEEE Computer* 24(1): 81–89.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimisation, and Machine Learning*, Addison-Wesley.

Govindarajan, S. and Vemuri, R. (2000). Tightly integrated design space exploration with spatial and temporal partitioning in SPARC, In: R. W. Hartenstein and H. Grünbacher (editors), *Field Programmable Logic and Applications (FPL 2000 Proceedings)*, LNCS 1896, Springer-Verlag, Villach, Austria, August 27–30, pp. 7–18.

Guccione, S. A. (1995). *Programming Fine-Grained Reconfigurable Architectures*, PhD thesis, University of Texas at Austin.

Hadley, J. D. and Hutchings, B. L. (1995). Design methodologies for partially reconfigured systems, In: *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, USA, April 19–21, pp. 78–84.

Hauck, S., Li, Z. and Schwabe, E. (1998). Configuration compression for the xilinx XC6200 FPGA, In: *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*, Napa, CA, USA, April 15–17, pp. 138–146.

Heath, J. R., Kuekes, P. J., Snider, G. S. and Williams, R. S. (1998). A defect tolerant computer architecture: Opportunities for nanotechnology, *Science* 280: 1716–1721.

Hennessy, J. L. and Patterson, D. A. (1990). *Computer architecture: a quantitative approach*, Morgan Kaufmann Publishers.

Heron, J. P. and Woods, R. F. (1996). Architectural strategies for implementing an image processing algorithm on XC6200 FPGA, In: R. W. Hartenstein and M. Glesner (editors), *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers (FPL '96 Proceedings)*, LNCS 1142, Springer-Verlag, pp. 317–326.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*, The University of Michigan Press, Ann Arbor, Michigan, USA.

Jones, D. and Lewis, D. M. (1995). A time-multiplexed FPGA architecture for logic emulation, In: *Proceedings of IEEE Custom Integrated Circuits Conference*, pp. 495–498.

Kaul, M. and Vemuri, R. (1998). Optimal temporal partitioning and synthesis for reconfigurable architectures, In: *Design, Automation and Test in Europe Conference*, Paris, France, February 23–26.

Kean, T. (1988). *Configurable Logic: A Dynamically Programmable Celluar Architecture and its VLSI Implementation*, PhD thesis, University of Edinburgh, Dept. of Computer Science.

Kean, T. (1999). FPL in the era of system level integration, In: *FPL'99 Special One-day Seminar*, ISLI, The Alba Campus, Scotland, September 2.

Kirkpatrick, S., Gelatt, C. D. and Vecchi, M. P. (1983). Optimisation by simulated annealing, *Science* 220(4598): 671–680.

Lala, P. K. (2000). *Self-Checking and Fault-Tolerant Digital Design*, Morgan Kaufmann Publishers.

Lautzenheiser, D. P. (1986). Using dynamic reconfigurable logic in a XC2064 logic cell array, *Electro'86 and Mini/Micro Northeast Conference* pp. 26/2/1–10.

Ling, X.-P. and Amano, H. (1993a). Performance evaluation of WASMII: a data driven computer on a virtual hardware, In: *Proceedings 5th International PARLE Conference*, LNCS 694, pp. 610–621.

Ling, X.-P. and Amano, H. (1993b). WASMII: A data driven computer on a virtual hardware, In: D. A. Buell and K. L. Pocek (editors), *IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE Comput. Soc. Press, Napa, CA, USA, April 5–7, pp. 33–42.

Liu, H. and Wong, D. F. (1999). Circuit partitioning for dynamically reconfigurable FPGAs, In: *International Symposium on Field Programmable Gate Arrays*, Monterrey, CA, February, pp. 187–194.

Luk, W. et al. (1997a). Parametrised libraries for Xilinx 6200 FPGAs, *Preliminary documentation*, Dept of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, United Kingdom. Version 2.1.

Luk, W., Guo, S., Shirazi, N. and Zhuang, N. (1996). A fremework for developing parametrised FPGA libraries, In: R. W. Hartenstein and M. Glesner (editors), *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers (FPL '96 Proceedings)*, LNCS 1142, Springer-Verlag, pp. 24–33.

Luk, W., Shirazi, N. and Cheung, P. Y. (1997b). Compilation tools for run-time reconfigurable designs, In: *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, Napa, CA, USA, April 16–18, pp. 56–65.

Luk, W., Shirazi, N., Guo, S. R. and Cheung, P. Y. K. (1997c). Pipeline morphing and virtual pipelines, In: W. Luk, P. Y. K. Cheung and M. Glesner (editors), *Field Programmable Logic and Applications (FPL '97 Proceedings)*, LNCS 1304, Springer-Verlag, pp. 111–120.

Lysaght, P. and Dunlop, J. (1994). Dynamic reconfiguration of FPGAs, In: W. R. Moore and W. Luk (editors), *More FPGAs*, Abingdon EE&CS Books, pp. 82–94.

Lysaght, P. and Stockwood, J. (1996). A simulation tool for dynamically reconfigurable field programmable gate arrays, *IEEE Transactions on VLSI Systems* 4(3): 381–390.

Lysaght, P., Stockwood, J., Law, J. and Girma, D. (1994). Artificial neural network implementation on a fine-grained FPGA, In: *4th International Workshop on Field Programmable Logic and Applications*, LNCS 849, Springer-Verlag, Prague, Czech Republic, September 7–9, pp. 421–431.

Mange, D., Durand, S., Sanchez, E., Stauffer, A., Tempesti, G., Marchal, P. and Piguet, C. (1995). A new self-reproducing automaton based on a multi-cellular organization, *Technical Report No. 95/114*, Logic Synthesis Laboratory, Dept of Computer Science, Swiss Federal Institute of Technology, Lausanne, Switzerland.

McCaskill, J. and Wagler, P. (2000). From reconfigurability to evolution in construction systems—spanning electronic, microfluidic and biomolecular domains, In: R. W. Hartenstein and H. Grünbacher (editors), *Field Programmable Logic and Applications (FPL 2000 Proceedings)*, LNCS 1896, Springer-Verlag, Villach, Austria, August 27–30.

McFarland, M. C., Parker, A. and Camposano, R. (1990). The high-level synthesis of digital systems, *IEEE Proceedings* pp. 301–318.

McGregor, G. and Lysaght, P. (1999). Self controlling dynamic reconfiguration: A case study, In: P. Lysaght, J. Irvine and R. Hartenstein (editors),

Field-Programmable Logic and Applications, LNCS 1673, Springer-Verlag, Glasgow, UK, August 30–September 1, pp. 144–154.

Minninck, R. (1964). Cutpoint cellular logic, *IEEE Transactions on Electronic Computers* EC-13: 685–698.

Morris, R. and Nowrouzian, B. (1996). A novel technique for pipelined scheduling and allocation of data-flow graphs based on genetic algorithms, In: T. J. Malkinson (editor), *1996 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Theme: Glimpse into the 21st Century (Cat. No.96TH8157)*, Vol. 1, Calgary, Alta., Canada, May 26–29, pp. 429–432.

Murgai, R., Brayton, R. K. and Sangiovanni-Vincentelli, A. (1995). *Logic Synthesis for Field-Programmable Gate Arrays*, Kluwer Academic Pubishers, Boston.

Nemhauser, G. L. and Wolsey, L. A. (1988). *Integer and Combinatorial Optimization*, Chichester & Wiley.

Ohmori, K. (1995). High-level synthesis using genetic algorithm, In: *1995 IEEE International Conference on Evolutionary Computation (Cat. No.95TH8099)*, Perth, WA, Australia, November 29–December 1, pp. 209–213.

Oldfield, J. V. and Dorf, R. C. (1995). *Field Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*, John Wiley and Sons.

Oliveira, A., Lau, N. and Sklyarov, V. (1998). Synthesis of VHDL code from the hierarchical specification of control circuits for dynamically recon-

figurable FPGAs, In: *Proceedings of VUIF Fall '98*, Orlando, Florida, October 26–28.

Paulin, P. G., Knight, J. P. and Girczyc, E. F. (1986). Hal: A multi-paradigm approach to automatic data path synthesis, In: *Proc. 23rd IEEE Design Automation Conference*, Las Vegas, NV, USA, July, pp. 263–270.

Robinson, D. and Lysaght, P. (1999). Modelling and synthesis of configuration controllers for dynamically reconfigurable logic systems using the dcs cad framework, In: P. Lysaght, J. Irvine and R. Hartenstein (editors), *Field-Programmable Logic and Applications*, LNCS 1673, Springer-Verlag, Glasgow, UK, August 30–September 1, pp. 41–50.

Robinson, D., McGregor, G. and Lysaght, P. (1998). New CAD framework extends simulation of dynamically reconfigurable logic, In: R. W. Hartenstein and A. Keevallik (editors), *Field Programmable Logic and Applications (FPL '98 Proceedings)*, LNCS 1482, Springer-Verlag, pp. 1–8.

Sait, S. M., Ali, S. and Benten, M. S. T. (1996). Scheduling and allocation in high level synthesis using stochastic techniques, *Microelectronics Journal* 27(8): 693–712.

Sels, P. (1996). *Scheduling for dynamically reconfigurable FPGAs*, Master's thesis, Keble College, Oxford University.

Sherwani, N. (1995). *Algorithms for VLSI Physical Design Automation*, 2nd edn, Kluwer Academic Pubishers.

Shirazi, N., Luk, W. and Cheung, P. Y. (1998). Automating production of run-time reconfigurable designs, In: *Proceedings of 6th IEEE Symposium*

on Field-Programmable Custom Computing Machines (FCCM'98), Napa, CA (USA), April 14–17, pp. 147–156.

Sidhu, R. P. S., Mei, A. and Prasanna, V. K. (1999). Genetic programming using self-reconfigurable FPGAs, In: P. Lysaght, J. Irvine and R. Hartenstein (editors), *Field-Programmable Logic and Applications*, LNCS 1673, Springer-Verlag, Glasgow, UK, August 30–September 1, pp. 301–312.

Silberschatz, A. and Galvin, P. B. (1998). *Operating System Concepts*, 5th edn, Addison-Wesley Longman.

Skylarov, V. and de Brito Ferrari, A. (1998). Design and implementation of control circuits based on dynamically reconfigurable FPGA, In: *Proc. of IEEE Internatinal Conference on Electronics, Circuits and Systems*, Lisbon.

Stanford, P. and Mancuso, P. (editors) (1990). *EDIF Electronic Design Interchange Format Version 2 0 0*, 2nd edn, Electronic Industries Association.

Takayama, A., Shibata, Y., Iwai, K. and Amano, H. (2000). Dataflow partitioning and scheduling algorithms for WASMII, a virtual hardware, In: R. W. Hartenstein and H. Grünbacher (editors), *Field-Programmable Logic and Applications (FPL 2000 Proceedings)*, LNCS 1896, Springer-Verlag, Villach, Austria, August 27–30, pp. 685–694.

Tangen, U. (2000). Self-organisation in micro-configurable hardware, In: M. A. Bedau et al. (editors), *Artificial Life VII: Proceedings of the 7th International Conference*.

Thompson, A. (1996). An evolved circuit, intrinsic in silicon, entwined with physics, In: *Proc. 1st Int. Conf. on Evolvable Systems (ICES 96)*.

Trimberger, S., Carberry, D., Johnson, A. and Wong, J. (1997). A time-multiplexed FPGA, In: *Proc. IEEE Symposium on FPGAs for Custom*

Computing Machines (FCCM'97), Napa, CA, USA, April 16–18, pp. 22–28.

Trimberger, S. M. (1994). *Field-Programmable Gate Array Technology*, Kluwer Academic Publishers.

Vasilko, M. (1999). DYNASTY: A temporal floorplanning based CAD framework for dynamically reconfigurable logic systems, In: P. Lysaght, J. Irvine and R. Hartenstein (editors), *Field-Programmable Logic and Applications (FPL'99 Proceedings)*, LNCS 1673, Springer-Verlag, pp. 124–133.

Vasilko, M. (2000). Design visualisation for dynamically reconfigurable systems, In: R. W. Hartenstein and H. Grünbacher (editors), *Field-Programmable Logic and Applications (FPL 2000 Proceedings)*, LNCS 1896, Springer-Verlag, Villach, Austria, August 27–30, pp. 131–140.

Vasilko, M. and Ait-Boudaoud, D. (1996a). Architectural synthesis techniques for dynamically reconfigurable logic, In: R. W. Hartenstein and M. Glesner (editors), *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers (FPL '96 Proceedings)*, LNCS 1142, Springer-Verlag, pp. 290–296.

Vasilko, M. and Ait-Boudaoud, D. (1996b). Optically reconfigurable FPGAs: Is this a future trend ?, In: R. W. Hartenstein and M. Glesner (editors), *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers (FPL '96 Proceedings)*, LNCS 1142, Springer-Verlag, pp. 270–279.

Vasilko, M. and Cabanis, D. (1999). A technique for modelling dynamic reconfiguration with improved simulation accuracy, *IEICE Transactions*

on Fundamentals of Electronics, Communications and Computer Science E82-A(11): 2465–2474.

Vasilko, M., Gibson, D., Long, D. and Holloway, S. (1999). Towards a consistent design methodology for run-time reconfigurable systems, In: *IEE Colloquium on Reconfigurable Systems, Digest No.99/061*, Glasgow, Scotland, March 10, pp. 5/1–4.

von Neumann, J. (1966). *Theory of Self-Reproducing Automata*, University of Illinois Press.

Wall, M. (1996). *GAlib: A C++ Library of Generic Algorithm Components, version 2.4*, MIT, available from `http://lancet.mit.edu/ga/`, August.

Wirthlin, M. J. and Hutchings, B. L. (1995). A dynamic instruction set computer, In: *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, USA, April 19–21, pp. 99–107.

Xilinx (1994). *The ProgrammableLogic Data Book*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124.

Xilinx (1997a). *XACTstep Series 6000 User Guide*, Xilinx, Inc.

Xilinx (1997b). *XC6200 Field Programmable Gate Arrays*, Xilinx, Inc., April 24 (Version 1.10). Product Description.

Xilinx (1998). *Parametrised Library for XC6200*, Xilinx, Inc., January. Velab documentation.

Xilinx (2000a). *Virtex Series Configuration Architecture User Guide*, Xilinx, Inc., September 27. XAPP151 (v1.5).

Xilinx (2000b). *Virtex™ 2.5 V Field Programmable Gate Arrays*, Xilinx, Inc., September 19. DS003 (v2.3).

Zhang, X.-j., Ng, K.-w. and Luk, W. (2000). A combined approach to high-level synthesis for dynamically reconfigurable systems, In: R. W. Hartenstein and H. Grünbacher (editors), *Field Programmable Logic and Applications (FPL 2000 Proceedings)*, LNCS 1896, Springer-Verlag, Villach, Austria, August 27–30, pp. 361–370.

Zhang, X.-j., Ng, K.-w. and Young, G. H. (1998). High-level synthesis using genetic algorithms for dynamically reconfigurable FPGAs (Abstract), In: *Proceedings of the 1998 ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA'98)*, ACM, Monterey, CA (USA), February 22–25, p. 258.