Christophe Bobda

Introduction to
Reconfigurable Computing

Architectures, algorithms and applications

Foreword by Reiner Hartenstein

Springer

Introduction to Reconfigurable Computing

Introduction to Reconfigurable Computing

Architectures, Algorithms, and Applications

by

Christophe Bobda
University of Kaiserslautern, Germany

Springer

To Lewin, Jan and Huguette for being so patient

Foreword

"Christophe Bobda's book is an all-embracing introduction to the fundamentals of the entire discipline of Reconfigurable Computing, also seen with the eyes of a software developer and including a taxonomy of application areas.

Reconfigurable Computing is a disruptive innovation currently going to complete the most important breakthrough after introduction of the von Neumann paradigm. On software to FPGA migrations a dazzling array of publications from a wide variety application areas reports speed-up factors between 1 and 4 orders of magnitude and promises to reduce the electricity bill by at least an order of magnitude. Facing the tcyberinfrastructure's growing electricity consumption (predicted to reach 35–50% of the total electricity production by the year 2020 in the USA) also this energy aspect is a strategic issue.

The focal point of worldwide almost 15 million software developers will shift toward new solutions of the productivity problems which stem from programming the coming many-core microprocessors and from educational deficits. Currently Reconfigurable Computing for High Performance computing is even disorienting the supercomputing scene. Their tremulous question is: Do we need to learn hardware design?

In the past, when students asked for a text book, we had to refer to a collection of specialized books and review articles focused on individual topics or special application areas, where Reconfigurable Computing takes only a corner or a chapter, sometimes even treating FPGAs as exotic technology (although in important areas it is mainstream for a decade). The typical style of those books or articles assumes that the reader has a hardware background: a leap too far for the existing software developers community.

The book by Christophe Bobda, however, has also been written for people with a software background, substantially reducing the educational leap bybridging the gap. His book has the potential to become a best-seller and to

stimulate the urgently needed transformation of the software developer population's mindset, by playing a similar role as known from the famous historic Mead-&-Conway textbook for the VLSI design revolution.

<div align="right">

Reiner Hartenstein, IEEE fellow,
Professor, TU Kaiserslautern "

</div>

Contents

Preface

One indicator of the growing importance of Reconfigurable Computing is the large number of events (conferences, workshops, meetings) organized and devoted to this topic in the last couple of years. Also, the growth observed in the market share of programmable logic devices, particulary the FPGAs, is an indicator of the strong interest in reconfigurable logic.

Following this development, teaching reconfigurable computing, which was initiated in many Universities a couple of years before, has gained more importance. The curricula in reconfigurable computing varies from simple seminars to more heavy syllabus including lectures, exercises and labs.

Many people among whom Reiner Hartenstein have been advocating years ago in favour of a normalized reconfigurable computing syllabus. Aware of the importance of a teaching book in this normalization, during a bi-annual meeting on reconfigurable computing held in Dagstuhl in 2003 [11], Hartenstein coined the importance of a text book in reconfigurable computing and proposed the attendees to write one. He suggested to have several people writing to minimize the work and have the book published as faster as possible. Unfortunately, this initiative was not pursued.

A couple of months after this initiative, in the summer term 2004, I started teaching a course in reconfigurable computing at the university of Erlangen-Nuremberg. With the difficulties of acquiring teaching materials and labs, I started writing a script to ease the learning process of students. The positive feedback gained from the student encouraged me to continue writing. Further repetitions of the course in winter term 2004 and in winter term 2006 were used to improve the course contents. It should be mentioned that in a couple of books [153] [220] [134], reconfigurable computing were published in between. However, none of them were found to cover the complete aspects of reconfigurable computing as I use to teach in my course.

My goal in writing this book was to provide a strong theoretical and practical background, as a contribution for a syllabus in reconfigurable computing.

A short overview on the content of each chapter is provided in Section 6 of Chapter 1.

This book targets graduate students and lecturers in computer engineering, computer science and electrical engineering. Also professional in the afore mentioned field can use the book as well. We supply the book with teaching materials (slides and labs) to ease the course preparation for those willing to introduce a reconfigurable computing curricula. The teaching material as well as the labs can be downloaded from the course Web page at www.bobda.net/rc-book.

Finally, despite all the effort place in the review, we cannot be sure that all the mistakes were filtered out. We will therefore be grateful to receive your comments and feedback on possible errors.

Kaiserslautern, June 2007
Christophe Bobda

About the Author

Dr. Bobda received the Licence degree in mathematics from the University of Yaounde, Cameroon, in 1992, the diploma of computer science and the Ph.D. degree (with honors) in computer science from the University of Paderborn in Germany in 1999 and 2003, respectively. In June 2003, he joined the department of computer science at the University of Erlangen-Nuremberg in Germany as post doc. In October 2005, he moved to the University of Kaiserslautern as Junior Professor, where he leads the working group Self-Organizing Embedded Systems in the department of computer science. His research interests include reconfigurable computing, self-organization in embedded systems, multiprocessor on chip and adaptive image processing.

Dr. Bobda received the Best Dissertation Award 2003 from the University of Paderborn for his work on synthesis of reconfigurable systems using temporal partitioning and temporal placement.

Dr. Bobda is member of The IEEE Computer Society, the ACM and the GI. He has also served in the program committee of several conferences (FPL, FPT, RAW, RSP, ERSA, DRS) and in the DATE executive committee as proceedings chair (2004, 2005, 2006, 2007). He served as reviewer of several journals (*IEEE TC, IEEE TVLSI, Elsevier Journal of Microprocessor and Microsystems, Integration the VLSI Journal*) and conferences (DAC, DATE, FPL, FPT, SBCCI, RAW, RSP, ERSA).

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Research in architecture of computer systems has always been a central preoccupation of the computer science and computer engineering communities. The investigation goals vary according to the target applications, the price of the final equipment, the programmability of the system, the environment in which processors will be deployed and many others.

For processors to be used in parallel machines for high-performance computing as it is the case in weather simulation, the focus is placed on high clock rates, parallelism and high communication bandwidth at the expense of power. In many embedded systems, the price of the final equipment is the governing factor during the development. A small microcontroller is usually used to control data acquisition from sensors and provide data to actuators at a very low frequency. In many other embedded systems, in particular in untethered systems, power and cost optimization are the central goals. In those systems, the growing need of more computation power that contradict with power and cost optimization put a lot of pressure on engineers who must find a good balance of all contradicting goals. For an autonomous cart used to explore a given environment, the processing unit must be able to capture images, compress the images and send the compressed images to a base station for control. Parallel to this, the system must perform other actions such as obstacle detection and avoidance. In such a system, power must be optimized to allow the system to run as long as possible. On the other hand, the processor must process image frames as fast as possible, to avoid important frames to be missed. Obstacle detection and avoidance must also be done as faster as possible to avoid a possible crash of the cart. The multiplicity of goals has led to the development of several processing architectures, each optimized according to a given goal. Those architectures can be categorized in three main groups according to their degree of flexibility: the general purpose computing group that is based on the

Von Neumann (VN) computing paradigm; domain-specific processors, tailored for a class of applications having in common a great range of characteristics; application-specific processors tailored for only one application.

1.   General Purpose Computing

In 1945, the mathematician John Von Neumann demonstrated in a study of computation that a computer could have a simple, fixed structure, able to execute any kind of computation, given a properly programmed control, without the need for hardware modification. The VN contribution was universally adopted and quickly became the fundament of future generations of high-speed digital computers. One of the reasons for the acceptance of the VN approach is its simplicity of programming that follows the sequential way of human thinking.

The general structure of a VN machine as shown in figure 1.1 consists of:

- A memory for storing program and data. Harvard architectures contain two parallel accessible memories for storing program and data separately.

- A control unit (also called control path) featuring a program counter that holds the address of the next instruction to be executed.

- An arithmetic and logic unit (also called data path) in which instructions are executed.
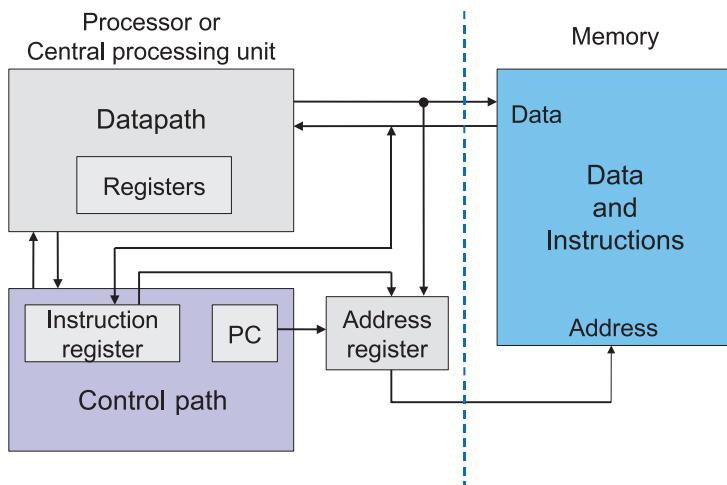
Figure 1.1.   The Von Neumann Computer architecture

A program is coded as a set of instructions to be executed sequentially, instruction after instruction. At each step of the program execution, the next instruction is fetched from the memory at the address specified in the program counter and decoded. The required operands are then collected from the memory before the instruction is executed. After execution, the result is written back into the memory. In this process, the control path is in charge of setting all signals necessary to read from and write to the memory, and to allow the data path to perform the right computation. The data path is controlled by the control path, which interprets the instructions and sets the data path's signals accordingly to execute the desired operation.

In general, the execution of an instruction on a VN computer can be done in five cycles: *Instruction Read* (IR) in which an instruction is fetched from the memory; *Decoding* (D) in which the meaning of the instruction is determined and the operands are localized; *Read Operands* (R) in which the operands are read from the memory; *Execute* (EX) in which the instruction is executed with the read operands; *Write Result* (W) in which the result of the execution is stored back to the memory. In each of those five cycles, only the part of the hardware involved in the computation is activated. The rest remains idle. For example if the IR cycle is to be performed, the program counter will be activated to get the address of the instruction, the memory will be addressed and the instruction register to store the instruction before decoding will be also activated. Apart from those three units (program counter, memory and instruction register), all the other units remain idle. Fortunately, the structure of instructions allows several of them to occupy the idle part of the processor, thus increasing the computation throughput.

1.1 Instruction Level Parallelism

Pipelining is a transparent way to optimize the hardware utilization as well as the performance of programs. Because the execution of one instruction cycle affects only a part of the hardware, idle parts could be activated by having many different cycles executing together. For one instruction, it is not possible to have many cycles being executed together. For instance, any attempt to perform the execute cycle (EX) together with the reading of an operand (R) for the same instruction will not work, because the data needed for the EX should first be provided by R cycle. Nevertheless, the two cycles EX and D can be performed in parallel for two different instructions. Once the data have been collected for the first instruction, its execution can start while the data are being collected for the second instruction. This overlapping in the execution of instructions is called *pipelining* or *instruction level parallelism (ILP)*, and it is aimed at increasing the throughput in the execution of instructions as well as the resource utilization. It should be mentioned that ILP does not reduce the execution latency of a single execution, but increases the throughput of a set

of instructions. The maximum throughput is dictated by the impact of hazards in the computation. Those Hazards can be reduced for example by the use of a Harvard architecture.

If t_{cycle} is the time needed to execute one cycle, then the execution of one instruction will require $5*t_{cycle}$ to perform. If three instructions have to be executed, then the time needed to perform the execution of those three instructions without pipelining is $15*t_{cycle}$, as illustrated in figure 1.2. Using pipelining, the ideal time needed to perform those three instruction, when no hazards have to be dealt with, is $7*t_{cycle}$. In reality, we must take hazards into account. This increases the overall computation time to $9*t_{cycle}$.

The main advantage of the VN computing paradigm is its flexibility, because it can be used to program almost all existing algorithms. However, each algorithm can be implemented on a VN computer only if it is coded according to the VN rules. We say in this case that '*The algorithm must adapt itself to the hardware*'. Also because of the temporal use of the same hardware for a wide variety of applications, VN computation is often characterized as '*temporal computation*'.

With the fact that all algorithms must be sequentially programmed to run on a VN computer, many algorithms cannot be executed with their potential best performance. Algorithms that usually perform the same set of inherent parallel operations on a huge set of data are not good candidates for implementation on a VN machine.
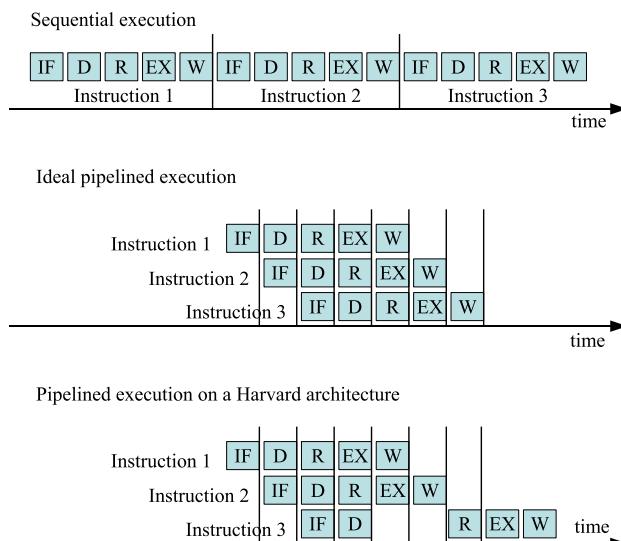
Figure 1.2.   Sequential and pipelined execution of instructions on a Von Neumann Computer

If the class of algorithms to be executed is known in advance, then the processor can be modified to better match the computation paradigm of that class of application. In this case, the data path will be tailored to always execute the same set of operations, thus making the memory access for instruction fetching as well as the instruction decoding redundant. Moreover, the memory access for data fetching and storing can also be avoided if the sources and destinations of data are known in advance. A bus could for instance provide sensor data to the processor, which in turn sends back the computed data to the actuators using another bus.

2. Domain-Specific Processors

A domain-specific processor is a processor tailored for a class of algorithms. As mentioned in the previous section, the data path is tailored for an optimal execution of a common set of operations that mostly characterizes the algorithms in the given class. Also, memory access is reduced as much as possible. Digital Signal Processor (DSP) belong to the most used domain-specific processors.

A DSP is a specialized processor used to speed-up computation of repetitive, numerically intensive tasks in signal processing areas such as telecommunication, multimedia, automobile, radar, sonar, seismic, image processing, etc. The most often cited feature of the DSPs is their ability to perform one or more *multiply accumulate* (MAC) operations in single cycle. Usually, MAC operations have to be performed on a huge set of data. In a MAC operation, data are first multiplied and then added to an accumulated value. The normal VN computer would perform a MAC in 10 steps. The first instruction (multiply) would be fetched, then decoded, then the operand would be read and multiply, the result would be stored back and the next instruction (accumulate) would be read, the result stored in the previous step would be read again and added to the accumulated value and the result would be stored back. DSPs avoid those steps by using specialized hardware that directly performs the addition after multiplication without having to access the memory.

Because many DSP algorithms involve performing repetitive computations, most DSP processors provide special support for efficient looping. Often a special loop or repeat instruction is provided, which allows a loop implementation without expending any instruction cycles for updating and testing the loop counter or branching back to the top of the loop. DSPs are also customized for data with a given width according to the application domain. For example if a DSP is to be used for image processing, then pixels have to be processed. If the pixels are represented in Red Green Blue (RGB) system where each colour is represented by a byte, then an image processing DSP will not need more than 8 bit data path. Obviously, the image processing DSP cannot be used again for applications requiring 32 bits computation.

This specialization of the DSPs increases the performance of the processor and improves the device utilization. However, the flexibility is reduced, because it cannot be used anymore to implement other applications other than those for which it was optimally designed.

3.   Application-Specific Processors

Although DSPs incorporate a degree of application-specific features such as MAC and data width optimization, they still incorporate the VN approach and, therefore, remain sequential machines. Their performance is limited. If a processor has to be used for only one application, which is known and fixed in advance, then the processing unit could be designed and optimized for that particular application. In this case, we say that '*the hardware adapts itself to the application*'.

In multimedia processing, processors are usually designed to perform the compression of video frames according to a video compression standard. Such processors cannot be used for something else than compression. Even in compression, the standard must exactly match the one implemented in the processors. A processor designed for only one application is called an *Application-Specific Processor (ASIP)*. In an ASIP, the instruction cycles (IR, D, EX, W) are eliminated. The instruction set of the application is directly implemented in hardware. Input data stream in the processor through its inputs, the processor performs the required computation and the results can be collected at the outputs of the processor. ASIPs are usually implemented as single chips called *Application-Specific Integrated Circuit (ASIC)*

EXAMPLE 1.1 *If algorithm 1 has to execute on a Von Neumann computer, then at least 3 instructions are required.*

Algorithm 1
if $a < b$ **then**
$\quad d = a + b$
$\quad c = a \cdot b$
else
$\quad d = b + 1$
$\quad c = a - 1$
end if

With t_{cycle} being the instruction cycle, the program will be executed in $3 * 5 * t_{cycles} = 15 * t_{cycle}$ *without pipelining.*

Let us now consider the implementation of the same algorithm in an ASIP. We can implement the instructions $d = a + b$ *and* $c = a * b$ *in parallel. The same is also true for* $d = b + 1, c = a - 1$ *as illustrated in figure 1.3*
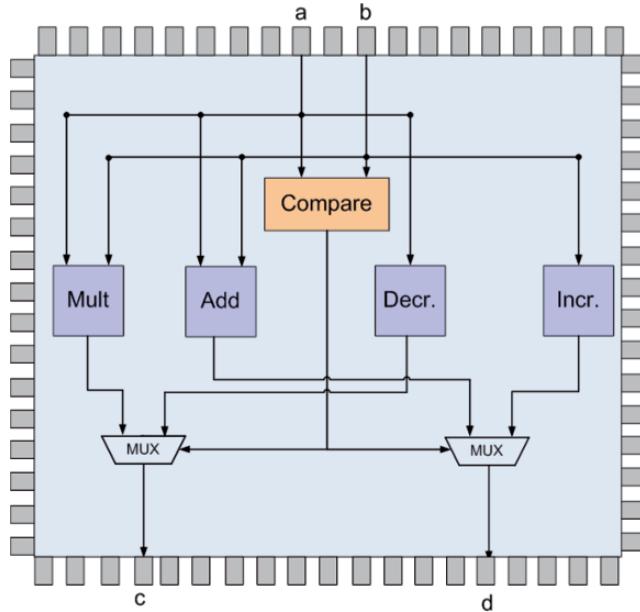
Figure 1.3. ASIP implementation of Algorithm 1

The four instructions $a+b$, $a*b$, $b+1$, $a-1$ as well as the comparison $a < b$ will be executed in parallel in a first stage. Depending on the value of the comparison $a < b$, the correct values of the previous stage computations will be assigned to c and d as defined in the program. Let t_{max} be the longest signal needed by a signal to move from one point to another in the physical implementation of the processor (this will happen on the path Input-multiply-multiplex). t_{max} is also called the cycle time of the ASIP processor. For two inputs a and b, the results c and d can be computed in time t_{max}. The VN processor can compete with this ASIP only if $15 * t_{cycle} < t_{max}$, i.e. $t_{cycle} < t_{max}/15$. The VN must be at least 15 times faster than the ASIP to be competitive. Obviously, we have assumed a VN without pipeline. The case where a VN computer with a pipeline is used can be treated in the same way.

ASIPs use a spatial approach to implement only one application. The functional units needed for the computation of all parts of the application must be available on the surface of the final processor. This kind of computation is called '*Spatial Computing*'.

Once again, an ASIP that is built to perform a given computation cannot be used for other tasks other than those for which it has been originally designed.

4.   Reconfigurable Computing

From the discussion in the previous sections, where we studied three
different kinds of processing units, we can identify two main means to charac-
terize processors: *flexibility* and *performance*.

- The VN computers are very flexible because they are able to compute any
  kind of task. This is the reason why the terminology GPP (General Pur-
  pose Processor) is used for the VN machine. They do not bring so much
  performance, because they cannot compute in parallel. Moreover, the five
  steps (IR, D, R, EX, W) needed to perform one instruction becomes a major
  drawback, in particular if the same instruction has to be executed on huge
  sets of data. Flexibility is possible because 'the application must always
  adapt to the hardware' in order to be executed.

- ASIPs bring much performance because they are optimized for a particular
  application. The instruction set required for that application can then be
  built in a chip. Performance is possible because 'the hardware is always
  adapted to the application'.

If we consider two scales, one for the performance and the other for the flex-
ibility, then the VN computers can be placed at one end and the ASIPs at the
other end as illustrated in figure 1.4.
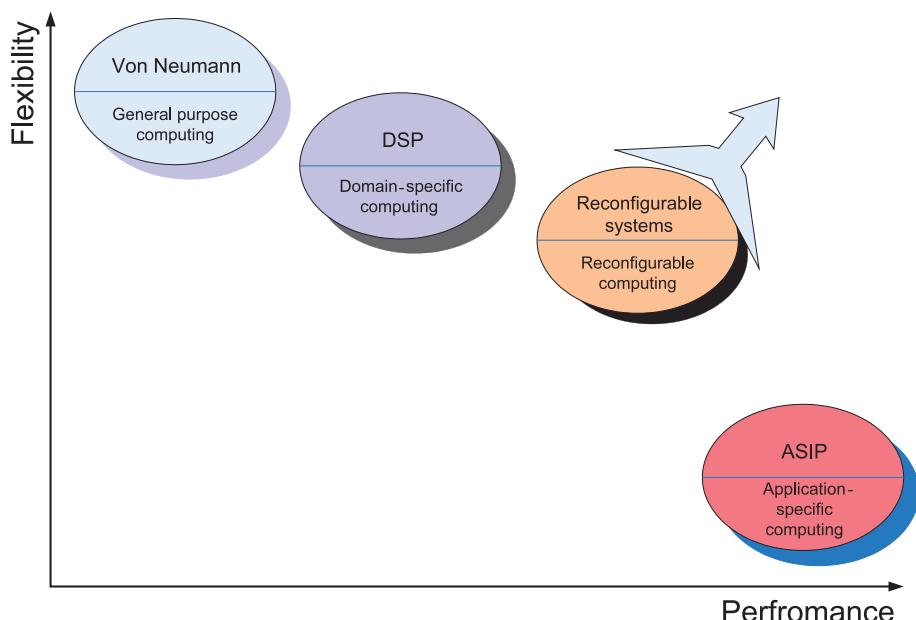
Figure 1.4.   Flexibility vs performance of processor classes

Between the GPPs and the ASIPs are a large numbers of processors. Depending on their performance and their flexibility, they can be placed near or far from the GPPs on the two scales.

Given this, how can we choose a processor adapted to our computation needs? If the range of applications for which the processor will be used is large or if it is not even defined at all, then the GPP should be chosen. However, if the processor is to be used for one application like it is the case in embedded systems, then the best approach will be to design a new ASIP optimized for that application.

Ideally, we would like to have the flexibility of the GPP and the performance of the ASIP in the same device. We would like to have a device able 'to adapt to the application' on the fly. We call such a hardware device a *reconfigurable hardware* or *reconfigurable device* or *reconfigurable processing unit (RPU)* in analogy the Central Processing Unit (CPU). Following this, we provide a definition of the term reconfigurable computing. More on the taxonomy in reconfigurable computing can be found in [111] [112].

DEFINITION 1.2 (RECONFIGURABLE COMPUTING) *Reconfigurable computing is defined as the study of computation using reconfigurable devices.*

For a given application, at a given time, the spatial structure of the device will be modified such as to use the best computing approach to speed up that application. If a new application has to be computed, the device structure will be modified again to match the new application. Contrary to the VN computers, which are programmed by a set of instructions to be executed sequentially, the structure of reconfigurable devices are changed by modifying all or part of the hardware at compile-time or at run-time, usually by downloading a so-called bitstream into the device.

DEFINITION 1.3 (CONFIGURATION, RECONFIGURATION) *Configuration respectively reconfiguration is the process of changing the structure of a reconfigurable device at star-up-time respectively at run-time*

Progress in reconfiguration has been amazing in the last two decades. This is mostly due to the wide acceptance of the Field Programmable Gate Array (FPGAs) that are now established as the most widely used reconfigurable devices. The number of workshops, conferences and meetings dealing with this topics has also grown following the FPGA evolution. Reconfigurable devices can be used in a wide number of fields, from which we list some in the next section.

5. Fields of Application

In this section, we would like to present a non-exhaustive list of fields, where the use of reconfiguration can be of great interest. Because the field is still growing, several new fields of application are likely to be developed in the future.

5.1   Rapid Prototyping

Rapid prototyping is certainly one of the most important fields of application of reconfiguration. The development of an ASIC that can be seen as physical implementation of ASIPs is a cumbersome process consisting of several steps, from the specification down to the layout of the chip and the final production. Because of the different optimization goals in development, several teams of engineers are usually involved. This increases the Non-recurring engineering (NRE) cost that can only be amortized if the final product is produced in a very large quantity. Contrary to software development, errors discovered after the production mean enormous loss because the produced pieces become either unusable or a great adaptation effort must be spent for the deployment. Rapid prototyping allows a device to be tested in real hardware before the final production. In this sense, errors can be corrected without affecting the pieces already produced. A reconfigurable device is useful here, because it can be used several times to implement different versions of the final product until an error-free state. One of the concepts related to rapid prototyping is *hardware emulation*, in analogy to software simulation. With hardware emulation, a hardware module is tested under real operating conditions in the environment where it will be deployed later.

5.2   In-System Customization

Time to market has become one of the main challenges that electronic manufacturers face today. In order to secure market segments, manufacturers must release their products as quickly as possible. In many cases, a well working product can be released with less functionalities. The manufacturer can then upgrade the product on the field to incorporate new functionalities. While this approach works with normal VN processors, it is not the case with ASIPs. A reconfigurable device provides such capabilities of being upgraded in the field, by changing the configuration.

In-system customization can also be used to upgrade systems that are deployed into non-accessible or very difficult to access locations. One example is the Mars rover vehicle in which some FPGAs, which can be modified from the earth, are used.

5.3   Multi-modal Computation

The number of electronic devices we interact with is permanently increasing. Many people hold besides a mobile phone, other devices such as handhelds, portable mp3 player, portable video player, etc. Besides those mobile devices, fixed devices such as navigation systems, music and video players as well as TV devices are available in cars and at home. All those devices are equipped with electronic control units that run the desire application on the

desired device. Furthermore, many of the devices are used in a time multiplexed fashion. It is difficult to imagine someone playing mp3 songs while watching a video clip and given a phone call. For a group of devices used exclusively in a time multiplexed way, only one electronic control unit can be used. Whenever a service is needed, the control unit is connected to the corresponding device at the correct location and reconfigured with the adequate configuration. For instance, a domestic mp3, a domestic DVD player, a car mp3, a car DVD player as well as a mobile mp3 player and a mobile video player can all share the same electronic unit, if they are always used by the same person. However, if several persons have access to the same devices (this can happen in a household with several people), then sharing of the electronic control unit will be rather difficult. In the first case, the user just needs to remove the control unit from the domestic devices and connect them to one car device when going to work. The control unit can be removed from the car and connected to a mobile device if the user decides to go for a walk. Coming back home, the electronic control unit is removed from the mobile device and used for watching video.

5.4 Adaptive Computing Systems

Advances in computation and communication are helping the development of ubiquitous and pervasive computing. Computer anytime and everywhere is now becoming reality.

The design of ubiquitous computing system is cumbersome task that cannot be dealt with only at compile time. Because of uncertainty and unpredictability of such systems, it is impossible, at compile time, to address all scenarios that can happen at run-time, because of unpredictable changes in the environment. We need computing systems that are able to adapt their behavior and structure to change operating and environmental conditions, to time-varying optimizing objectives, and to physical constraints such as changing protocols and new standards. We call those computing systems *Adaptive Computing System*.

Reconfiguration can provide a good fundament for the realization of adaptive systems, because it allows system to quickly react to changes by adopting the optimal behavior for a given run-time scenario.

6. Organization of the Book

The rest of the book is organized in nine chapters, ranging from the architecture to the applications. In those chapters, we try to provide a mixture of the theoretical as well as the practical background needed to understand, develop and deploy reconfigurable systems.

- **Architecture of reconfigurable systems**: After a brief tour in the earlier systems, this chapter considers the technology as well as the coupling possibilities of reconfigurable systems, from the fine-grained look up table

(LUT)-based reconfigurable systems like the field programmable gate arrays (FPGA) to the new coarse-grained technology.

- **Design and implementation**: This chapter deals with the implementation on reconfigurable system. It covers the steps needed (design entry, functional simulation, logic synthesis, technology mapping, place and route and bit stream generation) to implement today's FPGAs. We focus deeply on the logic synthesis for FPGAs, in particular LUT technology mapping.

- **High-Level Synthesis for Reconfigurable Devices**: This chapter considers the high-level synthesis for reconfigurable systems, also known as temporal partitioning. It covers the implementation of large functions, which cannot fit into one RPU. Several temporal partitioning techniques are presented and explained.

- **Temporal placement**: In this chapter, stand-alone reconfigurable systems are considered. We assume that a kind of operating systems for reconfigurable systems is in charge of managing the resources of a given system and allocate space on a device for the computation of incoming tasks. We therefore present several temporal placement approaches for offline as well as online placement.

- **Online Communication**: Modules dynamically placed at run-time on a given device need to communicate with each other in order to exchange data. Therefore, they dynamically create a need of communication channels on the chip. This chapter reviews and explains the different approaches to solve this dynamic intercommunication need.

- **Designing for Partial Reconfiguration on Xilinx Virtex FPGA**: This chapter considers the implementation for partial reconfiguration on the Xilinx FPGAs that are one of the few one on the market with this feature. We present the different possibilities to produce the partial bitstream needed at run-time to reconfigure only part of the device. Also, based on a case study, some hints are provided on the design of viable platforms for partial reconfiguration.

- **System on Programmable Chip**: System on programmable chip is a hot topic in reconfigurable computing. This is mainly the integration of a system made upon some peripheral (UART, Ethernet, VGA, etc.), but also computational (coding, filter, etc.) hardware modules on one programmable chip. We present the current usable solutions. Furthermore, we focus on the development of adaptive multiprocessors on chip. Those are systems consisting of a set of Processors and exchangeable hardware accelerators connected in a network for a parallel implementation of applications on the chip.

■ **Applications**: This section presents applications of reconfigurable systems. It covers the use of reconfigurable system in computer architecture (rapid prototyping, reconfigurable supercomputer, reconfigurable massively parallel computers) and algorithm better adapted for reconfigurable systems (distributed arithmetic, pattern matching, control, software defined radio, cryptography, etc.).

Chapter 2

RECONFIGURABLE ARCHITECTURES

In the previous chapter, we presented the reconfigurable devices as piece of hardware able to dynamically adapt to algorithms. How can it be possible that a hardware device, whose structure is normally fixed at fabrication time and cannot be changed anymore during the lifetime, can be readapted at run-time to dynamically match the application requirements? In this chapter, we bring some light into this issue by presenting the architecture of the commonly used reconfigurable devices.

After taking a brief tour of the evolution of reconfigurable computing architectures, we will cover in the first part of the chapter the so-called *fine-grained* reconfigurable devices. Those are hardware devices, whose functionality can be modified on a very low level of granularity. For instance, the device can be modified such as to add or remove a single inverter or a single two input NAND-gate. Fine-grained reconfigurable devices are mostly represented by programmable logic devices (PLD). This consists of programmable logic arrays (PLA), programmable array logics (PAL), complex programmable logic devices (CPLD) and the field programmable gate arrays (FPGA).

In the second part of the chapter, we cover the pseudo reconfigurable device also called *coarse-grained* reconfigurable devices. Those devices usually consist of a set of coarse-grained elements such as ALUs that allow for the efficient implementation of a dataflow function.

1.   Early Work

Attempts to have a flexible hardware structure that can be dynamically modified at run-time to compute a desired function are almost as older as the development of other computing paradigms. To overcome the non flexibility of the first computer, the ENIAC (electronical numerical integrator and computer) that could be programmed only by handwriting an algorithm, Jon Von

Neumann proposed a first universal architecture made upon three main blocks (memory, datapath and control path), able to run any given and well-coded program. The Von Neumann approach was not intended to provide the most efficient hardwired structure for each application, but a platform to run all type of programs without spending too much effort in the rearrangement of the underlying hardware. The previous effort of Von Neumann was then pursued by other researchers with goal of always having the best computation structure of a given application. We present some of those investigations in this section.

1.1 The Estrin Fix-plus machine

In 1959, Gerald Estrin, a computer scientist of the university of California at Los Angeles, introduced the concept of reconfigurable computing. The following fragment of an Estrin publication in 1960 [81] the fix-plus machine, defines the concept of reconfigurable computing paradigm.

"*Pragmatic problem studies predicts gains in computation speeds in a variety of computational tasks when executed on appropriate problem-oriented configurations of the variable structure computer. The economic feasibility of the system is based on utilization of essentially the same hardware in a variety of special purpose structures. This capability is achieved by programmed or physical restructuring of a part of the hardware.*"

To implement its vision, Estrin designed a computing system, the *fix-plus* machine [81]. Like many reconfigurable computing systems available today, the fix-plus machine consists of three main elements as shown in figure 2.1.

- A high-speed general purpose computer, the *fixed part (F)* that can be implemented on any general purpose processor. In its fix-plus machine, the IBM 7090 were used.

- A *variable part (V)* consisting of various size high-speed digital substructures that can be reorganized in problem-oriented special purpose configurations.

- The *supervisory control (SC)* coordinate operations between the fix module and the variable module.

The variable part (V) was made upon a set of problem-specific optimized functional units in the basic configuration (trigonometric functions, logarithm, exponentials, n-th power, roots, complex arithmetic, hyperbolic, matrix operation).

The basic block modules (figure 2.2) could be inserted into any of 36 positions on a motherboard (figure 2.12(a)) that provides the functionality for a given application. The functionality of the motherboard could be manually
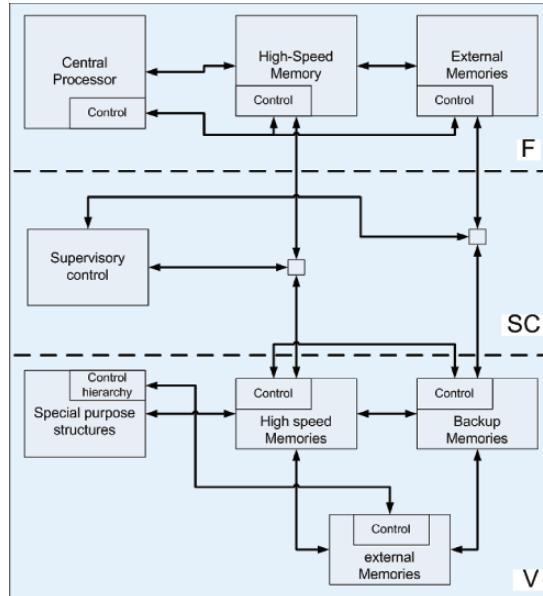
Figure 2.1. Structure of the Estrin Fix-Plus Machine

modified by replacing some basic blocks by new ones. Two types of basic building were available in the fix-plus machine: The *first basic element* that entailed four amplifiers and associated input logic for signal inversion, amplification, or high-speed storage. The *second basic block* used as combinatoric that was made upon ten diodes and four output drivers.
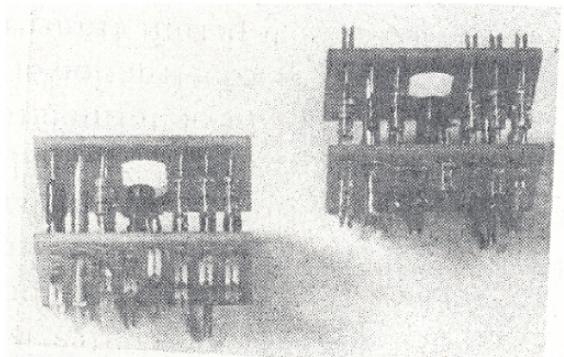
Figure 2.2. The basic building blocks of Fix-Plus Machine
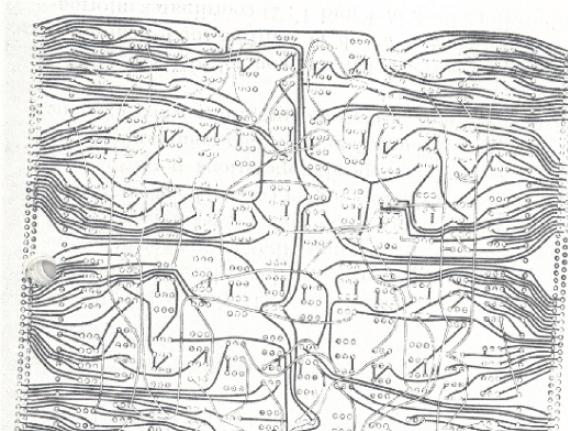
Figure 2.3.   The wiring harness of the Estrin-Machine

The connection between the modules is done by wiring harness (figure 2.3). With this architecture, the reconfiguration was done manually by replacing some modules on the motherboard (figure 2.4) or by changing a wiring harness for a new connection among the existing modules.

The fix-plus machine was intended to be used for accelerating Eigenvalues computation of matrices and has shown a speed gain of 2.5–1000 over the IBM7090 [80] [79]. The available technology at that time however made the use of the fix-plus machine difficult. Reconfiguration had to be done manually, and substantial software efforts were required to implement applications (figure 2.5).
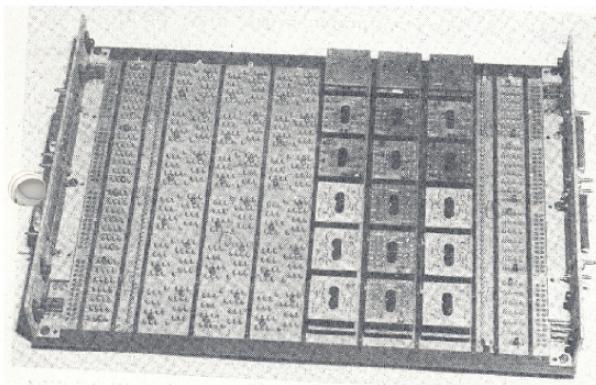
Figure 2.4.   The motherboard of the Estrin's Fix-Plus
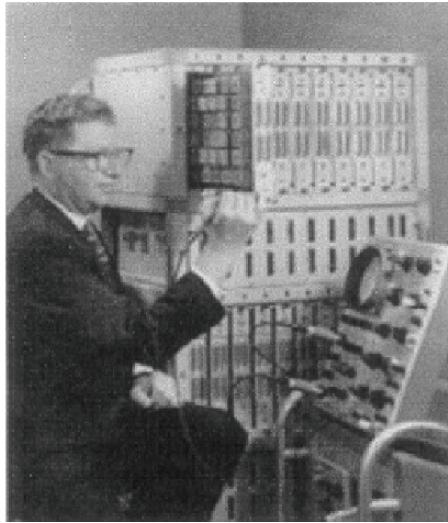
Figure 2.5. Estrin at work: Hand-Controlled Reconfiguration

1.2 The Rammig Machine

In the year 1977, Franz J. Rammig, a researcher at the university of Dortmund proposed a concept for editing hardware [182]. The goal was the "*investigation of a system, which, with no manual or mechanical interference, permits the building, changing, processing and destruction of real (not simulated) digital hardware*".

Rammig materialized his concept by developing a *hardware editor* similar to the today's FPGA architecture. The editor was made upon a set of modules, a set of pins and a one-to-one mapping function on the set of pins. The circuitry of a given function was then defined as a "*string*" on an alphabet of two letters (w = "*wired*" and u = "*unwired*"). To build the hardware editor, *selectors* were provided with the modules' outputs connected to the input of the selectors and the output of the selectors connected to the input of the modules. The overall system architecture is shown in figure 2.6.

The implementation of the {*wired, unwired*} property was done through a programmable crossbar switch, made upon an array of selectors. The bit strings were provided by storing the selector control in registers, and by making these registers accessible from a host computer, the PDP11 in those days. The modules were provided on a library board similar to that of Estrin's Fix-Plus. Each board could be selected under software control. The mapping from module I/Os to pins was realized manually, by a wiring of the provided library boards, i.e. fixed per library board.
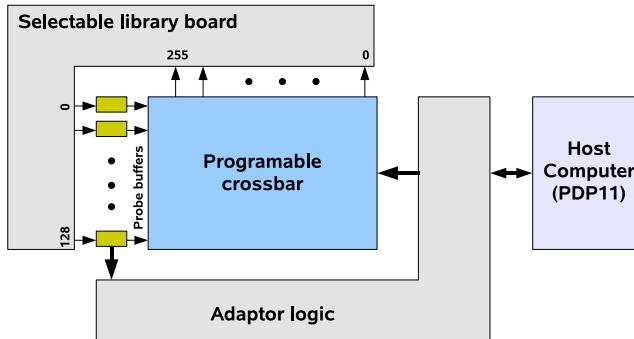
Figure 2.6.   Structure of the Rammig machine

The Rammig machine was heavily used as emulation platform, which is also one of the largest application fields of today's FPGAs. It was therefore possible to control the complete behaviour of a circuit under observation from the software side. This was done by buffering module outputs in registers and transferring the contents of these registers to the host before clocking the system. The system was implemented with a 128×192 crossbar and got the name *META-46 GOLDLAC* (figure 2.7).

1.3   Hartenstein's *XPuter*

The *Xputer*'s [113] [216] concept was presented in early 1980s by Reiner Hartenstein, a researcher at the University of Kaiserslautern in Germany.

Figure 2.7.   META-46 GOLDLAC

The goal was to have a very high degree of programmable parallelism in the hardware, at lowest possible level, to obtain performance not possible with the Von Neumann computers. Instead of sequencing the instructions, the *Xputer* used to sequence data, thus exploiting the regularity in the data dependencies of some class of applications like in image processing, where a repetitive processing is performed on a large amount of data. An *Xputer* consists of three main parts: the *data sequencer*, the *data memory* and the *reconfigurable ALU (rALU)* that permits the run-time configuration of communication at levels below instruction set level. Within a loop, data to be processed were accessed via a data structure called *scan window*. Data manipulation was done by the rALU that had access to many scan windows. The most essential part of the data sequencer was the generic address generator (GAG) that was able to produce address sequences corresponding to the data of up to three nested loops. An rALU subnet that could be configured to perform all computations on the data of a scan window was required for each level of a nested loop.

The general *XPuter* architecture is presented in figure 2.8. Shown is the realization of the *XPuter* as a map oriented machine (MoM). The overall system was made upon a host processor, whose memory was accessible by the MoM.

The rALU subnets received their data directly from local memory or from the host main memory via the MoM bus. Communication was also possible
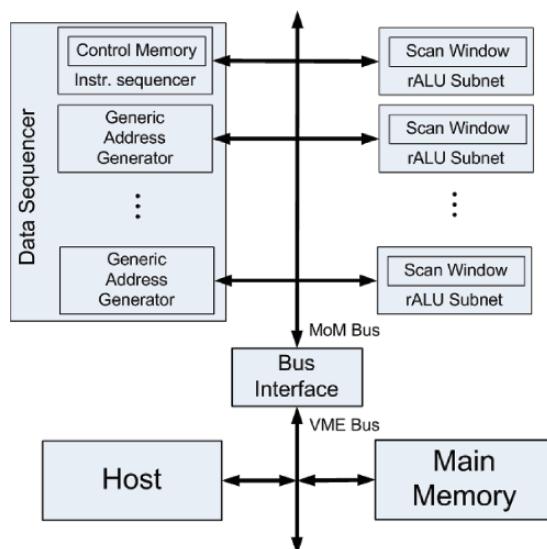
Figure 2.8. General architecture of the XPuter as implemented in the Map oriented Machine (MOM-3) prototype

among the rALUs via direct serial connections. Several *XPuters* could also be connected to provide more performance.

For executing a program, the hardware had to be configured first. If no reconfiguration would take place at run-time, then only the data memory would be necessary. Otherwise, a configuration memory would be required to hold all the configurations to be used at run-time.

The basic building block of the reconfigurable ALU was the so-called reconfigurable datapath unit (rDPU) (figure 2.9). Several rDPUs were used within an rALU for data manipulation. Each rDPU had two registered inputs and two registered outputs with a datawidth of 32 bit. Input data were provided either from the north or from the west, while the south and east were used for the output. Besides the interconnection lines for the rALUs, a global I/O-Bus is available for the connection of designs to the external world. The I/O bus was principally used for accessing the scan windows.

The global view of the reconfigurable datapath attached to a host processor is given on figure 2.9. It consists of two main parts: the *control unit* and a *field of rDPUs*. The register file was used for optimizing the memory access when the GAG operated with overlapping scan windows. In this case, data in the actual scan window position will be reused in the following positions. Those data could therefore be temporary stored in registers and copied back into the memory if they were no more needed.

The control implemented a program that is loaded on reconfiguration to control different units of the rALU. Its instruction set consisted of instructions for loading the data as well as instructions for collecting results from the field.

Application of the *XPuters* was in image processing, systolic array and signal processing.
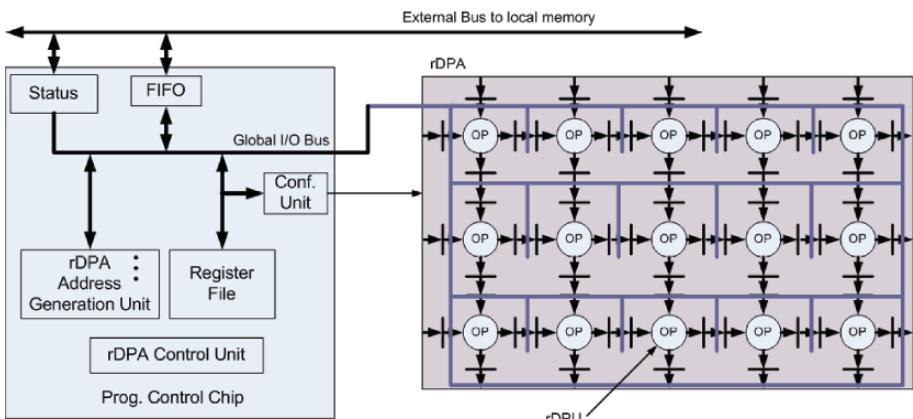
Figure 2.9.   Structure of the XPuter's reconfigurable ALU

The introduction of commercial FPGAs by Xilinx in the mid-1980s increased interest in reconfigurable computing. Many systems in high-performance computing were built with farm of FPGAs on different boards. The "drawbacks" of FPGAs also inspired many research groups that propose new architectures. We briefly present some of the most popular experiments. Included are the PAM developed at the DEC research center in Paris, the SPLASH system of the Supercomputer Research Center, the PRISM, the DISC and the GARP.

1.4 The PAM Machine

Introduced by Bertin et al. [25] from the Digital Equipment corporation, a *programmable active memory (PAM)* is defined as a uniform array of identical cells, the processing array blocks (PAB) that are connected in a given regular fashion. The architecture is built on the FPGA model that will be presented later in this chapter. The basic building block in a PAM is a cell, which consist of a flip flop and a combinatorial element with four inputs and four outputs. The combinatorial element consists of five look-up tables, each of which implements an output as a function of the five inputs. One hundred and sixty bits are, therefore, necessary to configure a combinatorial element. The flip flop provides a memory for the realization of finite state machines but can be used for local data storage. Figure 2.10 shows a PAM consisting of a 4×4 array of PAB.

Like in the FPGAs, the PAM configuration is defined by a bitstream that set the functions of the combinatorial elements as well as the state of the flip flops.

A prototype of the PAM, the Perle was built using a 5×5 array of *Logic Array Cell (LCA)* , a CMOS cell designed by Xillinx. Perle-0 features a VME
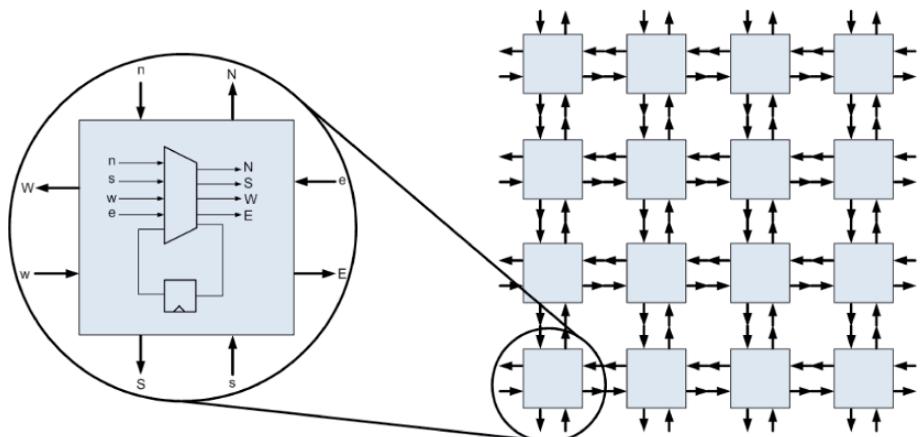
Figure 2.10. Programmable Active Memory (PAM) architecture as array of (PABs)

bus to interface a host CPU to which it is coupled. The host is in charge of downloading the bitstream to the Perle. The configuration controller and the host-bus communication protocol are programmed into two extra LCAs, statically configured at power-up time from a PROM.

Example of applications for which the Perle was used to implement are data compression, cryptography are image processing and, energetic physics.

1.5   The *SPLASH II*

SPLASH II [97], [39] is a computing system made upon a set of FPGA modules solded on different printed circuit boards, connected together to build a massive parallel computer. The system consists of a set of boards, called *array boards* (figure 2.11), that can be reconfigured according to the application currently being implemented. The array boards are connected to a host computer (in this case a SUN SPARC Station 2) via a system Bus (SBUS) and an interface board. Each array board features 17 Xilinx XC4010 FPGAs. Although sixteen of the seventeen FPGAs are used for computation, the seventeenth is use for data broadcasting. Each of the FPGAs has access to a separate 512-Kbyte memory. The SLASH II can be viewed as a large serial data path from the interface board to the array board. The SIMD bus is used to transmit data to the first FPGA (X0) on each board, which can then broadcast the data to the other sixteen FPGAs on its array. Alternatively, the data can also be moved in a linear way on an array board, from FPGA to FPGA. In this case, the data are received by the first FPGA (X1) that serially transmit the data to the next FPGAs on the board. The data are transmitted to the next board by the last FPGA in the chain.

Some examples of applications that have benefited from the SPLASH II performance are image processing, text searching in genetic database and fingerprint matching.

1.6   The *PRISM* Paradigm

PRISM is an acronym for processor reconfiguration through instruction set metamorphosis. Developed by Athanas et al [12], the idea behind the PRISM is to use information extracted at compile-time from a given application and to build a new set of instructions tailored to that application. The so-generated instructions can then be implemented in a reconfigurable hardware unit, tightly coupled to the processor. To ease the programmability of such systems, the identification and synthesis of the operations in hardware should be done automatically and transparent to the user. This task is performed by a *configuration compiler*, which automatically performs a hardware/software partitioning. This results in the generation of a software and a hardware image. The software image is a sequential program to be executed on the processor, whereas the
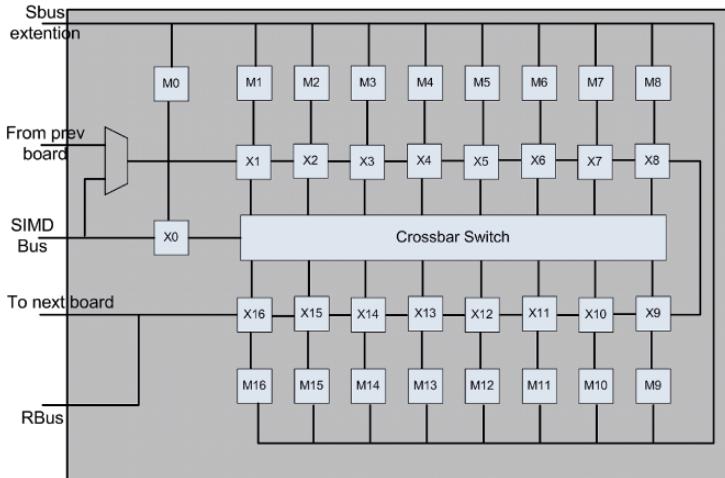
Figure 2.11. Architecture of the SPLASH II array board

hardware image defines the configuration of the attached reconfigurable hardware. As proof of concept, a PRISM prototype consisting of a 10-MHz processor M68010 attached to a board with four Xilinx 3090 FPGAs was constructed and tested.

1.7 The *Garp* Approach

The approach proposed by Hauser and Wawrzynek [114] is very close to today's embedded FPGA systems. The *Garp* was proposed as attempt to overcome the obstacles of reconfigurable computing machines, mostly consisting of a processor attached to a set of FPGA boards. The problems identified were for instance, the small capacity of FPGA, their expensive reconfiguration time, the non availability of internal memory to temporally stored data to be computed and the lack of a clear defined standard for coupling FPGAs to processor. Hauser and Wawrzynek proposed a system consisting of a microprocessor and a reconfigurable array on the same die. The system was not different than those already proposed. However, having the processor on the same die with the reconfigurable device was a great benefit. The time for transferring data to the reconfigurable array could be drastically reduced. Instructions and data caches were also foreseen on the chip. Almost all the FPGAs available today on the market offer a similar structure, with processors that can be synthesized according to the user's need. Despite the nice concept and positive simulation results, a Garp-chip could not be produced and commercialized.

1.8   *DISC*

The purpose of the *dynamic instruction set computer (DISC)* presented in [219] is to support demand-driven instruction set modifications. In contrast to the PRISM approach, where the specific instructions are synthesized and fixed at compiled-time, the DISC approach uses partial reconfiguration of FPGAs to place on the FPGA, hardware modules, each of which implements a given instruction. The relocation was also proposed as a means to reduced defragmentation of the FPGA. Because of its partial reconfiguration capabilities, the national semiconductor configurable logic array (Clay) was chosen for building a prototype consisting of a printed circuit board with two CLA31 FPGA and some memory. Although the first FPGA was used to control the configuration process, the second FPGA was used for implementing the instruction specific hardware blocks. Via an ISA Bus, the board was attached to a host processor running on Linux. A simple image mean filter first implemented as application-specific module, and later as sequence of general purpose instructions, was used to show the viability of the platform.

1.9   The *DPGA*

With similar goals as the *Garp* architecture, Dehon's *dynamically programmable gate array* was proposed as a means to extend the capability of microprocessor [68]. However, DGPAs provide a better implementation of the reconfigurable array proposed in the Garp. The concept proposed by Dehon is used in many coarse-grained devices available today. To reduce the reconfiguration time, one of the main bottlenecks of early FPGAs, the DGPA should be able to quickly switch among preloaded configurations. Therefore, redundant look-up tables are used to broadcast configurations in a local area on a cycle-by-cycle basis, thus allowing a clockwise reconfiguration of the FPGAs.

Despite the recent development of flexible coarse-grained device, almost all systems that require flexibility in the hardware structure rely on the programmable devices to which the PALs and PLAs, the CPLDs and the FPGA belongs. In the next section, we provide a short view on the structure of those devices.

2.   Simple Programmable Logic Devices

Programmable logic arrays (PLA) and programmable array logic (PAL) consist of a plane of AND-gates connected to a plane of OR-gates. The inputs signals as well as their negations are connected to the inputs of the AND-gates in the AND-plane. The outputs of the AND-gates are use as input for the

OR-gate in the OR-plane whose outputs correspond to those of the PAL/PLA. The connections in the two planes are programmable by the user.

Because every Boolean function can be written as a sum of products, the PLAs and PALs can be used to program any Boolean function after fabrication. The *products* are implemented in the AND-plane by connecting the wires accordingly, and the *sums* of the product are realized in the OR-plane. While in the PLAs both fields can be programmed by the user, this is not the case in the PALs where only the AND-plane is programmable. The OR-plane is fixed by the manufacturer. Therefore, the PALs can be seen as a subclass of the PLAs.

EXAMPLE 2.1 *Figure 2.12 shows the PLA and PAL implementations of the functions $F1 = A \cdot C + A \cdot \overline{B}$ and $F2 = A \cdot B + B \cdot C$. While the OR-plane of the PAL is no more programmable, by modifying the connection in the OR-Plane of the PLA, different sums of the products can be generated.*

Further enhancements (feeding the OR-output to a flip flop or feeding back a OR-output to a AND-input) are made on PLAs to allow more complex circuits to be implemented.

PALs and PLAs are well suited to implement two-level circuits; those are circuits made upon the sum of product as described earlier. At the first level, all the products are implemented and all the sum are implemented on the second level.

The main limitation of PLAs and PALs is their low capacity, which is due to the nature of the AND-OR-plane. The size of the plane grows too quickly as the number of inputs increases. Because of their low complexities, PALs and PLAs belong to the class of devices called *simple programmable logic devices* (SPLD).

(a) PAL: The OR-connections are fixed   (b) PLA: Programmable OR-connections
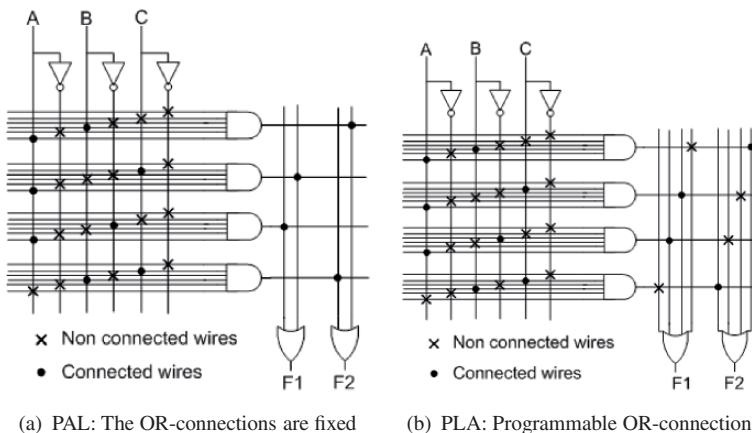
Figure 2.12. PAL and PLA implementations of the functions $F1 = A \cdot C + A \cdot \overline{B}$ and $F2 = A \cdot B + B \cdot C$

3.   Complex Programmable Logic Device

As stated in the previous section, PALs and PLAs are only available in small sizes, equivalent to a few hundred logic gates. For large logic circuits, *complex programmable logic devices (CPLD)* can be used.

A CPLD consists of a set of *macro cells*, *input/output blocks* and an *interconnection network*. The connection between the input/output blocks and the macro cells and those between macro cells and macro cells can be made through the programmable interconnection network (figure 2.13). A macro cell typically contains several PLAs and flip flops. Despite their relative large capacity (few hundreds thousands of logic gates), compared to those of PLAs, CPLDs are still too small for using in reconfigurable computing devices. They are usually used as glue logic, or to implement small functions. Because of their non volatility, CPLDs are used in many systems for configuration of the main reconfigurable device at start up.
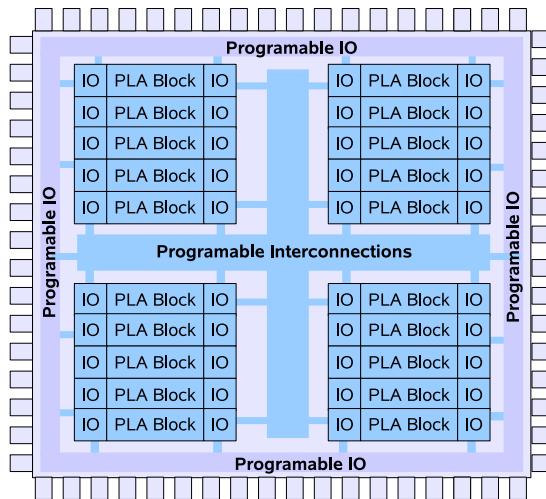
Figure 2.13.   Structure of a CPLD device

4.   Field Programmable Gate Arrays

Introduced in 1985 by the company Xilinx, an FPGA is a programmable device consisting, like the CPLDs, of three main parts. a set of *programmable logic cells* also called *logic blocks* or *configurable logic blocks*, a *programmable interconnection network* and a set of *input and output cells* around the device (figure 2.14). A function to be implemented in FPGA is partitioned in modules, each of which can be implemented in a logic block. The logic blocks are then connected together using the programmable interconnection. All three basic components of an FPGA (logic block, interconnection and input output)
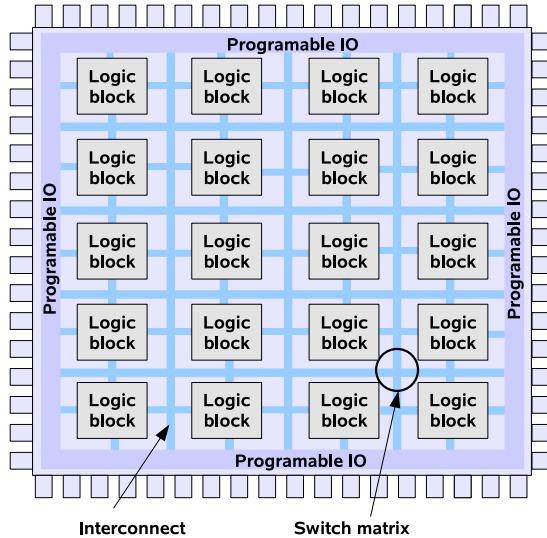
Figure 2.14. Structure of an FPGA

can be programmed by the user in the field. FPGAs can be programmed once or several times depending on the technology used.

4.1 Technology

The technology defines how the different blocks (logic blocks, interconnect, input/output) are physically realized. Basically, two major technologies exist: antifuse and memory-based. Whereas the antifuse paradigm is limited to the realization of interconnections, the memory-based paradigm is used for the computation as well as the interconnections. In the memory-based category, we can list the SRAM the EEPROM and the Flash based FPGAs.

4.1.1 Antifuse

Contrary to a fuse, an antifuse is normally an open circuit. An antifuse-based FPGA uses special antifuses included at each connection customization point. The two-terminal elements are connected to the upper and lower layer of the antifuse, in the middle of which a dielectric is placed (figure 2.15). In its initial state, the high resistance of the dielectric does not allow any current to flow between the two layers. Applying a high voltage causes large power dissipation in a small area, which melts the dielectric. This operation drastically reduces the resistance and a link can be built, which permanently connects the two layers. The two types of antifuses actually commercialized are: *The Programmable Low-Impedance Circuit Element* (PLICE) (2.15(a)), which is

(a) Q-Logic Vialink antifuse   (b) Actel PLICE antifuse
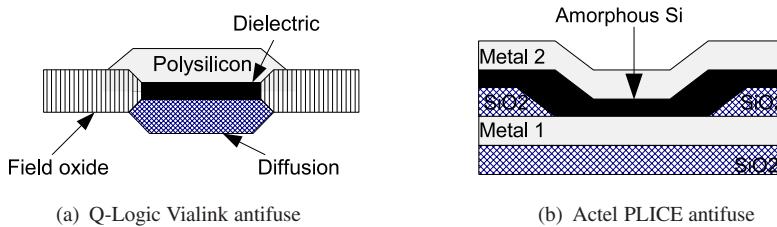
Figure 2.15.   Antifuse FPGA Technology

manufactured by the company Actel and the *Metal Antifuse* also called *ViaLink* made by the company QuickLogic.

The PLICE antifuse consists of an *Oxygen-Nitrogen-Oxygen* (ONO) dielectric layer sandwiched between a polysilicon and an n+ diffusion layer that serves as conductor. The *ViaLink* antifuse (2.15(b)) is composed of a sandwich of very high resistance layer of programmable amorphous silicon between two metal layers. When a programming voltage is applied, a metal-to-metal link is formed by permanently converting the silicon to a low resistance state.

The main advantage of the antifuse chips is their small area and their significantly lower resistance and parasitic capacitance compared with transistors. This *helps* to reduce the RC delays in the routing. However, anti-fuse-based FPGAs are not suitable for devices that must be frequently reprogrammed, as it is the case in reconfigurable computing. Antifuse FPGAs are normally programmed once by the user and will not change anymore. For this reason, they are also known as *one-time programmable* FPGAs.

4.1.2   SRAM

Unlike the antifuse that is used mostly used to configure the connection, a static RAM (SRAM) is use to configure the logic blocks and the connection as well. SRAM-based FPGAs are the most widely used.

In an SRAM-based FPGA, the states of the logic blocks, i.e. their functionality bits as well as that of the interconnections, are controlled by the output of SRAM cells (figure 2.16).

As shown in figure 2.17(a), a connection between two wires can be done by a *pass transistor* connected to the wires (one wire on a terminal). The state of the transistor can then be controlled by the output of an SRAM cell that allows the current to flow or not to flow between the two wires. Copying different values into the SRAM allows for changing the behaviour of the connection.
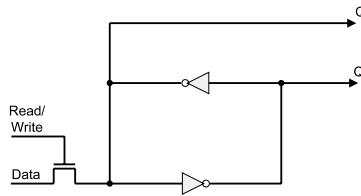
Figure 2.16. A Xillinx SRAM cell

The same principle applies for the programming of the logic modules, the so-called function generators. The selector input of a multiplexer (figure 2.17(b)) as well as the content of a look-up table representing the values of a function for different input combinations (figure 2.17(c)) can be reprogrammed by just copying new values into the corresponding SRAM cells.

The basic SRAM configuration cell (figure 2.16) is constructed from two cross-coupled inverters and uses a standard CMOS-process. The Q-output of the SRAM cell is connected to the module (pass transistor, multiplexer or Look up table) that is controlled by configuration. The read-and-write word line is use either to read or to write data from or to the SRAM through the data line.

The major advantage of this technology is that FPGAs can be programmed (configured) indefinitely. We just need to change the value into the SRAM-cells to realize a new connection or a new function. Moreover, the device can be done in-circuit very quickly and allow the reconfiguration to be done on the fly.

The disadvantages of SRAM-based FPGAs are the chip area required by the SRAM approach is relatively large. The total size of an SRAM-configuration cell plus the transistor switch that the SRAM-cell drives is also larger than the programming devices used in the antifuse technology. Furthermore, the device is volatile, i.e. the configuration of the device stored in the SRAM-cells is lost if the power is cut off. Therefore, external storage or non-volatile devices such

(a) Wire connection   (b) MUX programming   (c) Function generation
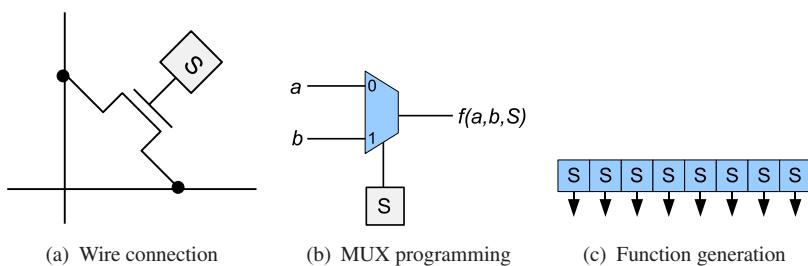
Figure 2.17. Use of SRAM in FPGA-Configuration

as CPLDs, EPROM or Flash devices, are required to store the configuration
and load it into the FPGA-device at power-on.

4.1.3   EPROM

Erasable programmable read only memory (EPROM) devices are based on
a floating gate (figure 2.18(a)). The device can be permanently programmed
by applying a high voltage (10–21 V) between the control gate and the drain
of the transistor (12 V). This causes the floating gate to be permanently and
negatively charged. This negative potential on the floating gate compensates
the voltage on the control gate and keeps the transistor closed.

In an *ultra violet* (UV) *PROM*, the programming process can be reversed
by exposing the floating gate to UV-light. This process reduces the threshold
voltage and makes the transistor function normally. For this purpose, the device
must be removed from the system in which it operates and plug into a special
device.

In *electrically erasable and programmable ROM (EEPROMs)* as well as
in *flash-EPROM*, the erase operation is accomplished electrically, rather than
by exposure to ultraviolet light. A high negative voltage must therefore be
applied at the control gate. This process is faster than using a UV lamp, and
the chip does not have to be removed from the system. In EEPROM-based
devices, two or more transistors are typically used in a ROM cell: one access
and one programmed transistor. The programmed transistor performs the same
function as the floating gate in an EPROM, with both charge and discharge
being done electrically.

In the flash-EEPROMs that are used as logic tile cell in the Actel ProASIC
chips (figure 2.18(b)), two transistors share the floating gate, which store the
programming information. The sensing transistor is only used for writing and
verification of the floating gate voltage whereas the other is used as switch.
This can be used to connect or disconnect routing nets to or from the configured
logic. The switch is also used to erase the floating gate.

(a) EPROM Cell   (b) Actel Flash-EEPROM
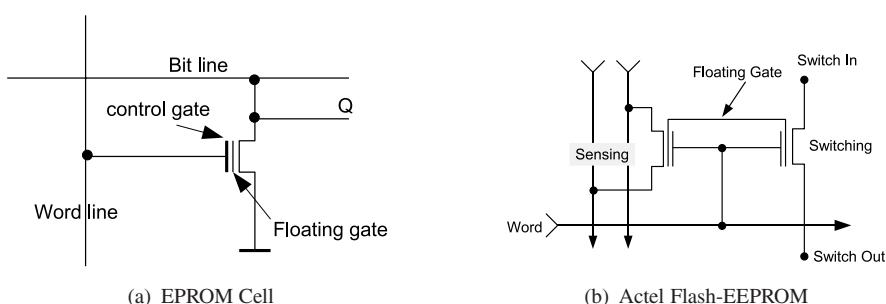
Figure 2.18.   EEPROM Technology

4.2 Function generators

A reconfigurable hardware device should provide the users with the possibility to dynamically implement and reimplement new functions. This is usually done by means of function generators that can be seen as the basic computing unit in the device. Two types of function generators are in use in commercial FPGAs: the multiplexers and the look-up tables.

4.2.1 Multiplexer

A 2:1 (2^n-input-1) *multiplexer* (MUX) is a selector circuit with 2^n inputs and one output. Its function is to allow only one input line to be fed at the output. The line to be fed at the output can be selected using some selector inputs. To select one of the 2^n possible inputs, n selectors lines are required. An MUX can be used to implement a given function. The straightforward way is to place the possible results of the function at the 2^n inputs of the MUX and to place the function arguments at the selector inputs. In this case, the MUX will work like a look-up table that will be explained in the next section. Several possibilities exist to implement a function in a MUX, for instance by using some arguments as inputs and other as selectors. Figure 2.19 illustrates this case for the function $f = a \times b$. The argument a is used as input in combination with a second input 0 and the second argument b is used as selector.

The Shannon expansion theorem can be used to decompose a function and implement it into a MUX. This theorem states that a given Boolean logic function $F(x_1, \ldots, x_n)$ of n variables can be written as shown in the following equation.

$$F(x_1, \cdots, x_n) = F(x_1, \cdots, x_i = 1, \cdots, x_n) \times x_i + \\ F(x_1, \cdots, x_i = 0, \cdots, x_n) \times \overline{x}_i$$

Where $F(x_1, \cdots, x_i = 1, \cdots, x_n)$ is the function obtained by replacing x_i with one and $F(x_1, \cdots, x_i = 0, \cdots, x_n)$ the function obtained by replacing x_i by zero in F. The functions $F_1 = F(x_1, \cdots, x_i = 1, \cdots, x_n)$ and $F_2 = F(x_1, \cdots, x_i = 0, \cdots, x_n)$ are called cofactors.
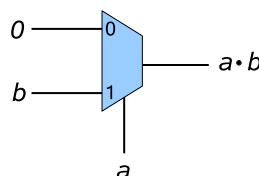
Figure 2.19. Implementation of f=ab in a 2-input MUX

A 2:1 multiplexer with inputs I_0, I_1, output O and selector S is defined by the following equation: $O(S, I_0, I_1) = \overline{S}(I_0) + S(I_1)$. The function F can be implemented on a 2:1 MUX with inputs $I_0 = F_0$ and $I_1 = F_1$, and selector $S = x_i$.

A complex function can be implemented in many multiplexers connected together. The function is first broken down into small pieces. Each piece is then implemented on a multiplexer. The multiplexers will then be connected to build the given function. This process is called technology mapping and is part of the logic synthesis that we consider in the next chapter.

To implement a function on a 2^n:1 MUX, the Shannon expansion theorem should be refined such as to have n fixed variables and 2^n cofactors. For $n = 2$, the function of the multiplexer is defined as follows:

$$O(S_0, S_1, I_0, I_1, I_2, I_3) = \overline{S_0}\overline{S_1}(I_0) + \overline{S_0}S_1(I_1) + S_0\overline{S_1}(I_2) + S_0S_1(I_3) \quad (4.1)$$

This is equivalent to the Shannon refinement of a function with two fixed variables as described below:

$$F(x_1,\cdots,x_n) = F(x_1,\cdots,x_i=1,\cdots,x_n)\cdot x_i + F(x_1,\cdots,x_i=0,\cdots,x_n)\cdot \overline{x}_i$$

$$= [F(x_1,\cdots,x_i=1,\cdots,x_j=1,..x_n)\cdot x_i + F(x_1,\cdots,x_i=0,\cdots,x_j=1,\cdots,x_n)\cdot \overline{x_i}]x_j$$

$$+ [F(x_1,\cdots,x_i=1,\cdots,x_j=0,\cdots,x_n)\cdot x_i + F(x_1,\cdots,x_i=0,\cdots,x_j=0,\cdots,x_n)\cdot \overline{x_i}]\overline{x}_j$$

$$= F(x_1,\cdots,x_i=1,\cdots,x_j=1,\cdots,x_n)\cdot x_i x_j + F(x_1,\cdots,x_i=0,\cdots,x_j=1,\cdots,x_n)\cdot \overline{x}_i x_j$$

$$+ F(x_1,\cdots,x_i=1,\cdots,x_j=0,\cdots,x_n)\cdot x_i \overline{x}_j + F(x_1,\cdots,x_i=0,\cdots,x_j=0,\cdots,x_n)\cdot \overline{x}_i \overline{x}_j$$

$$I_0 = F_0 = F(x_1,\cdots,x_i=0,\cdots,x_j=0,\cdots,x_n),$$

$$I_1 = F_1 = F(x_1,\cdots,x_i=0,\cdots,x_j=1,\cdots,x_n),$$

$$I_2 = F_2 = F(x_1,\cdots,x_i=1,\cdots,x_j=0,\cdots,x_n),$$

$$I_3 = F_3 = F(x_1,\cdots,x_i=1,\cdots,x_j=1,\cdots,x_n)$$

EXAMPLE 2.2 *We consider the implementation of a full adder using 4:1 MUX. Let a_i, b_i be the operand and c_{i-1} be the carry on a previous level, then s_i is the sum and c_i the carry for the next level. According to the truth table of the full adder (table 2.1), we can write s_i and c_i as:*

$$s_i = \overline{a_i}\overline{b_i}c_{i-1} + \overline{a_i}b_i\overline{c_{i-1}} + a_i\overline{b_i}\overline{c_{i-1}} + a_ib_ic_{i-1}$$
$$c_i = \overline{a_i}\overline{b_i} * 0 + \overline{a_i}b_ic_{i-1} + a_i\overline{b_i}c_{i-1} + a_ib_i * 1$$

According to those two equations and to the function of a 4:1 MUX given above, two multiplexers can be configured as shown in figure 2.20.

As stated earlier, an MUX provides much more possibilities than look-up tables, because the inputs of the MUX, which are equivalent of the SRAM

a_i	b_i	c_{i-1}	s_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 2.1. Truth table of the Full adder

output to a look-up table, are not reserved to the function result like as it is the case in look-up tables. Function arguments can be connected to the MUX-input as well as to he selector inputs.

The Actel ACT X Logic module. Multiplexer are used as function genera-tors in the Actel FPGA devices (figure 2.21). In the Actel ACT1 device, the basic computing element is the *logic module*. It is an 8-input 1-output logic circuit that can implement a wide range of functions. Besides combinatorial functions, the logic module can also implement a variety of D-latches. The *C-modules* present in the second generation of Actel devices, the ACT2, are similar to the *logic module*. The *S-modules*, which are found in the second and third generation of actel devices, contain an additional dedicated flip-flop. This avoids the building of flip flop from the combinatorial logic as it is the case in the logic module.

4.2.2 Look-Up Tables

A *look-up tables* (LUT) is a group of memory cells, which contain all the possible results of a given function for a given set of input values. Usually,
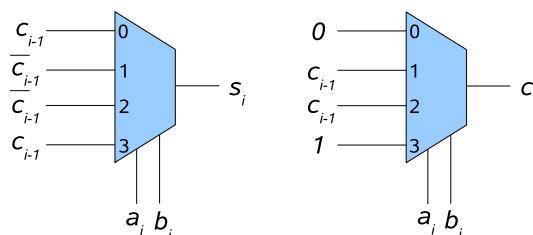
Figure 2.20. Implementation of a Full adder using two 4-input one output MUX
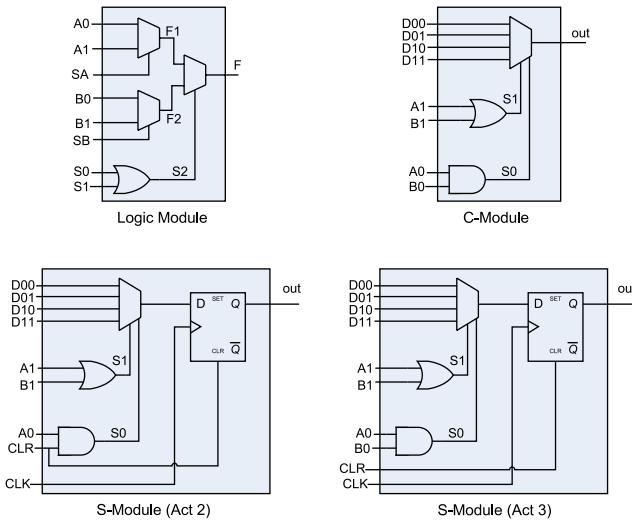
Figure 2.21.   The Actel basic computing blocks uses multiplexers as function generators

the set of possible function values corresponds to the possible combinations of the inputs. The values of the function are stored in such a way that they can be retrieved by the corresponding input values. An n-input LUT can be used to implement up to 2^{2^n} different functions, each of which can take 2^n possible values. Therefore, an n-input LUT must provide 2^n cells for storing the possible values of an n-input function. In FPGAs, an LUT physically consists of a set of SRAM-cells to store the values and a decoder that is used to access the correct SRAM location and retrieve the result of the function, which corresponds to the input combination (figure 2.22).

To implement a complex function in an LUT-based FPGA, the function must be divided into small pieces, each of which can be implemented in a single
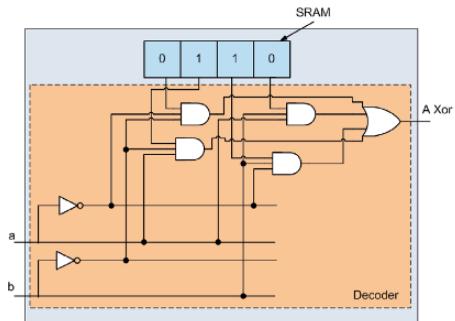
Figure 2.22.   2-input LUT

LUT. The interconnections are used to connect small pieces together and form the complete function

EXAMPLE 2.3 *The full adder of the previous section can be implemented using two 3-inputs, 1 output (3:1) LUT as shown in figure 2.23. The sum is implemented in the first LUT and the carry-out in the second LUT.*

The first three columns of table 2.1 represent the input values. They build the address used to retrieve the function value (corresponding to the value in the third column) from the corresponding LUT location. The content of the fourth and fifth columns of the truth table must therefore be copied in the corresponding LUTs as shown in figure 2.23. The sum values are copied in the upper LUT, while the carry values are compiled in the lower LUT.

SRAM-based LUT is used in the most commercial FPGAs as function generators. Several LUTs are usually grouped in a large module in which other functional elements such as flip flops and multiplexers are available. The connection between the LUTs inside such modules is faster than connections via the routing network, because dedicated wires are used. We consider examples of devices using LUT as function generator, the Xilinx FPGA and those of Altera, and we next explain how the LUT is used in those devices.

The Xilinx Configurable Logic Block. The basic computing block in the Xilinx FPGAs consists of an LUT with variable number of inputs, a set of multiplexers, arithmetic logic and a storage element (figure 2.24).

The LUT is used to store the configuration whereas the multiplexers select the right inputs for the LUT and the storage element as well as the right output of the block.

The arithmetic logic provides some facilities such as XOR-gate and faster carry chain to build faster adder without wasting too much LUT-resources.
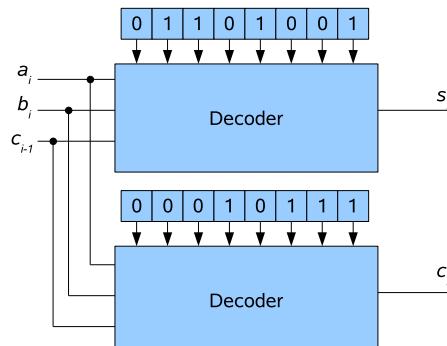
Figure 2.23. Implementation of a full adder in two 3-input LUTs
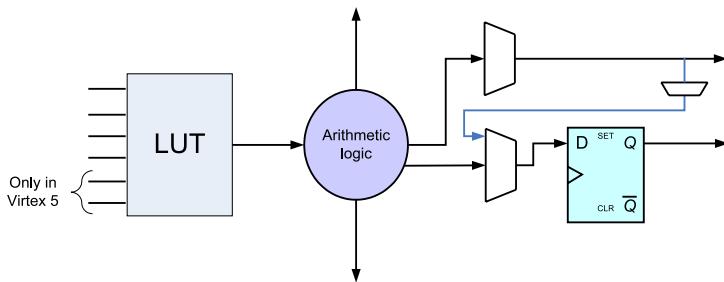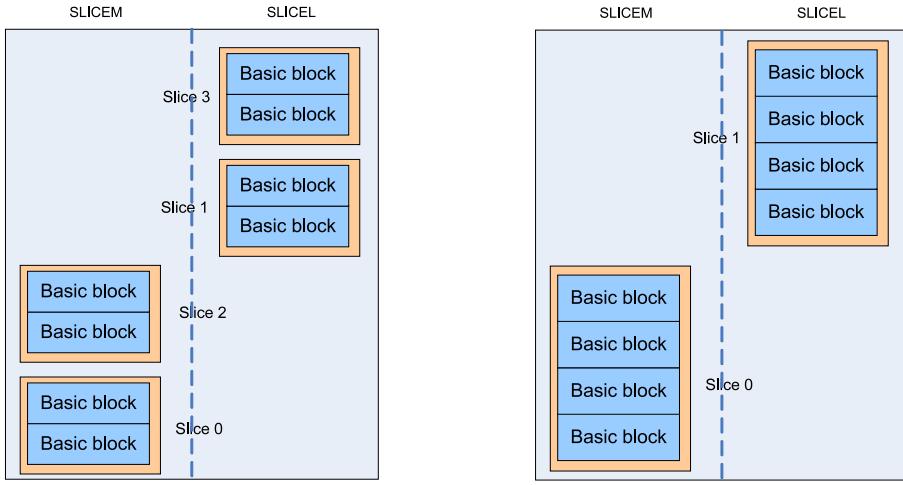
Figure 2.24.   Basic block of the Xilinx FPGAs

Several basic computing blocks are grouped in a coarse-grained element called the *configurable logic block (CLB)*(figure 2.25). The number of basic blocks in a CLB varies from device to device. In the older devices such as the 4000 series, the Virtex and Virtex E and the Spartan devices, two basic blocks were available in a CLB. In the newer devices such as the Spartan 3, the Virtex II, the Virtex II-Pro and the Virtex 4, the CLBs are divided into four slices each of which contains two basic blocks. The CLBs in the Virtex 5 devices contain only two slices, each of which contains four basic blocks.

In the newer devices, the left part slices of a CLB, also called SLICEM, can be configured either as combinatorial logic, or can be use as 16-bit SRAM or

Figure 2.25.   CLB in the newer Xilinx FPGAs (Spartan 3, Virtex 4 and Virtex 5)

as shift register while right-hand slices, the SLICEL, can only be configured as combinatorial logic.

Except for the Virtex 5, all LUTs in Xilinx devices have four inputs and one output. In the Virtex 5 each LUT has six inputs and two outputs. The LUT can be configured either as a 6-input LUT, in which case only one output can be used, or as two 5-input LUTs, in which case each of the two outputs is used as output of a 5-input LUT.

The Altera Logic Array Block.   Like the Xilinx devices, Altera's FPGAs (Cyclone, FLEX and Stratix) are also LUT-based. In the Cyclone II as well as in the FLEX architecture, the basic unit of logic is the *logic element* (LE) that typically contains an LUT, a flip flop, a multiplexer and additional logic for carry chain and register chain. Figure 2.26 shows the structure of the logic element in the Cyclone FPGA. This structure is very similar to that of the Altera FLEX devices. The LEs in the cyclone can operate in different modes each of which defines different usage of the LUT inputs.

In the Stratix II devices, the basic computing unit is called *adaptive logic module* (ALM) (figure 2.27). The ALM is made upon a mixture of 4-input and 3-input LUTs that can be used to implement logic functions with variable number of inputs. This ensures a backward compatibility to 4-input-based designs, while providing the possibility to implement coarse-grained module with variable number (up to 8) inputs. Additional modules including flip flops, adders and carry logic are also provided.

Altera logic cells are grouped to form coarse-grained computing elements called *logic array blocks* (LAB). The number of logic cells per LAB varies from the device to device. The Flex 6000 LABs contains ten logic elements while the FLEX 8000 LAB contains only eight. Sixteen LEs are available for each LAB in the cyclone II while the Stratix II LAB contains eight ALMs.
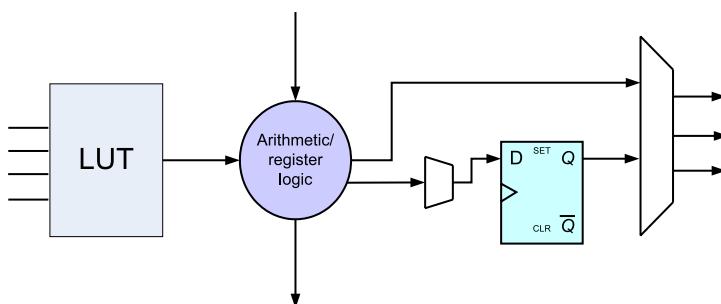
Figure 2.26.   Logic Element in the Cyclone II
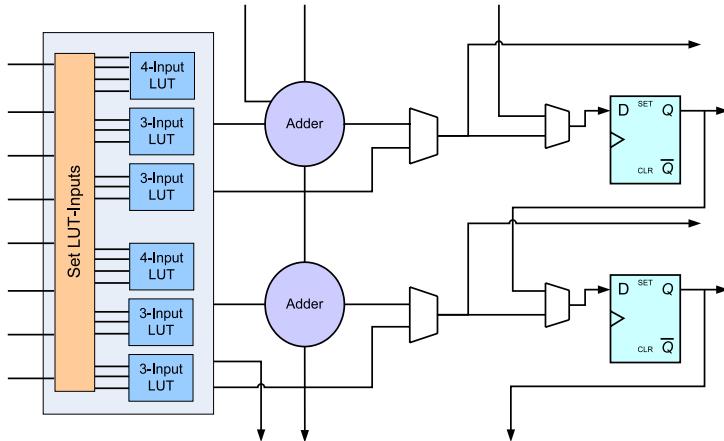
Figure 2.27.   Stratix II Adaptive Logic Module

4.3   FPGA structures

FPGAs consist of a set of programmable logic cells placed on the device such as to build an array of computing resources. The resulting structure is vendor-dependant. According to the arrangement of logic blocks and the inter-connection paradigm of the logic blocks on the device, FPGAs can be classified in four categories: *symmetrical array*, *row-based*, *hierarchy-based* and *sea of gates* (figure 2.28). We next explain each of the mentioned structure based on a commercial example.

4.3.1   Symmetrical Array: The Xilinx Virtex and Atmel AT40K Families

A symmetrical array-based FPGA consists of a two-dimensional array of logic blocks immersed in a set of vertical and horizontal lines. Switch elements exist at the intersections of the vertical and horizontal lines to allow for the connections of vertical and horizontal lines.

Examples of FPGAs arranged in a symmetrical array-based are the Xilinx Virtex FPGA and the Atmel (figure 2.29).

On the Xilinx devices, CLBs are embedded in the routing structure that consists of vertical and horizontal wires. Each CLB element is tied to a switch matrix to access the general routing structure, as shown in figure 2.30(a). The switch matrix provides programmable multiplexers, which are used to select the signals in the given routing channel that should be connected to the CLB terminals. The switch matrix can also connect vertical and horizontal lines, thus making routing possible on the FPGA.

Each CLB has access to two tri-state driver (TBUF) over the switch matrix. Those can be used to drive on-chip busses. Each tri-state buffer has its own tri-
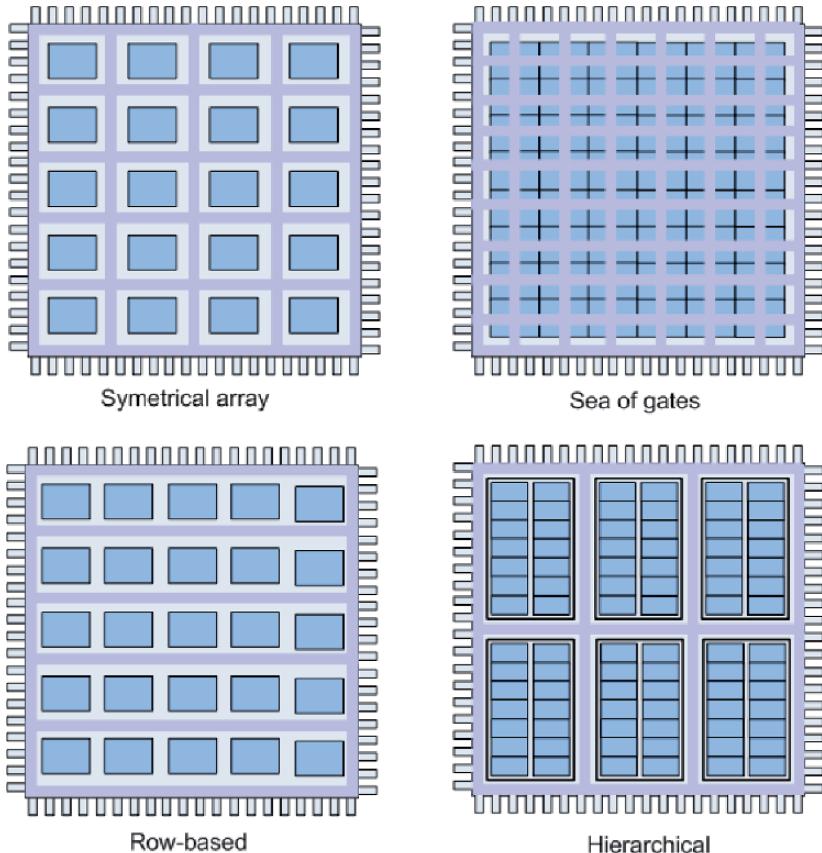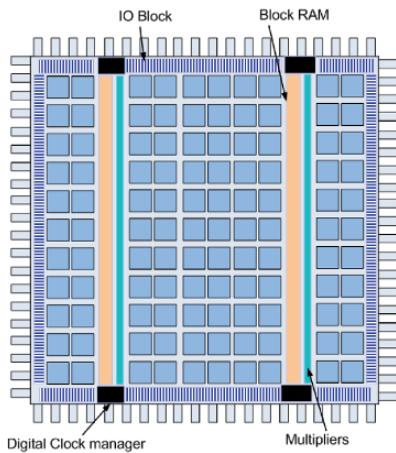
Figure 2.28. The four basic FPGA structures
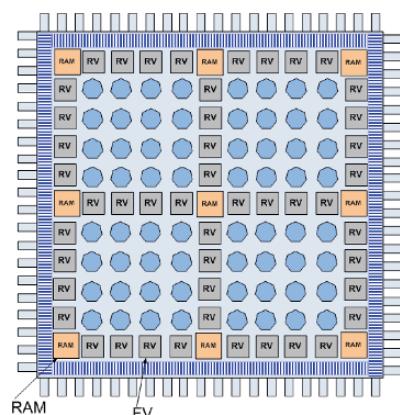
state control pin and its own input pin that are controlled by the logic built in the CLB. Four horizontal routing resources per CLB are provided for on-chip tri-state busses. Each tri-state buffer has access alternately to two horizontal lines, which can be partitioned as shown in figure 2.30(b). Besides the switch matrix, CLBs connect to their neighbours using dedicated fast connexion tracks.

The routing is done on the Atmel chips using a set of *busing planes*. Seven busing planes are available on the AT40K. Figure 2.31 depicts a part of the plane with five identical busing planes. Each plane has three bus resources: a *local-bus resource* (the middle bus) and two *express-bus* resources (both sides).

Repeaters are connected to two adjacent local-bus segments and two express-bus segments. Local bus segments span four cells whereas an express bus segments span eight cells. Long tri-state bus can be created by bypassing a repeater.
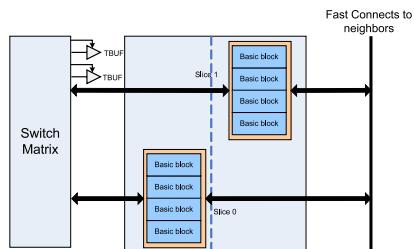
(a) The Xilinx Virtex II
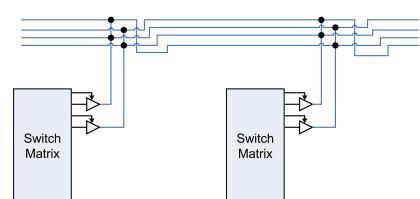
(b) Atmel's symmetrical array arrangement

Figure 2.29.   Symmetrical array arrangement in a) the Xilinx and b) the Atmel AT40K FPGAs

(a) CLB connexion to the switch matrix

(b) Tri-state buffer connection to horizontal lines

Figure 2.30.   Virtex routing resource
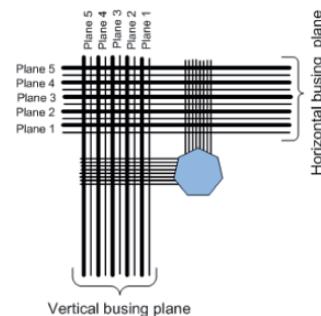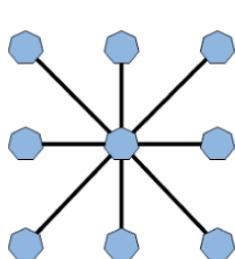
Figure 2.31.   Local connection of an Atmel Cell

Locally, the Atmel chip provides a star-like connection resource that allows each cell (which is the basic unit of computation) to be connected directly to all its eight neighbours. Figure 2.31 depicts direct connections between a cell and its eight nearest neighbours.

4.3.2   Row-Based FPGAs: The Actel ACT3 Family

A row-based FPGA consists of alternating rows of logic block or macro cells and channels (figure 2.32). The space between the logic blocks is called channel and is used for signal routing.

The Actel ACT3 FPGA family (figure 2.32) is an example of row-based FPGA. The macro cells are the C-elements and S-elements presented earlier in this chapter. The routing is done via the horizontal direction using the channels. In the vertical direction, dedicated vertical tracks are used. As shown in figure 2.33, a channel consists of several routing tracks divided into segments. The minimum length of a segment is the width of a module pair and its maximum length is the length of a complete channel, i.e. the width of the device.

Any segment that spans more than one-third of the row length is considered a long horizontal segment. Non dedicated horizontal routing tracks are used to route signal nets. Dedicated routing tracks are used for the global clock networks and for power and ground tracks. Vertical tracks (figure 2.33) are of
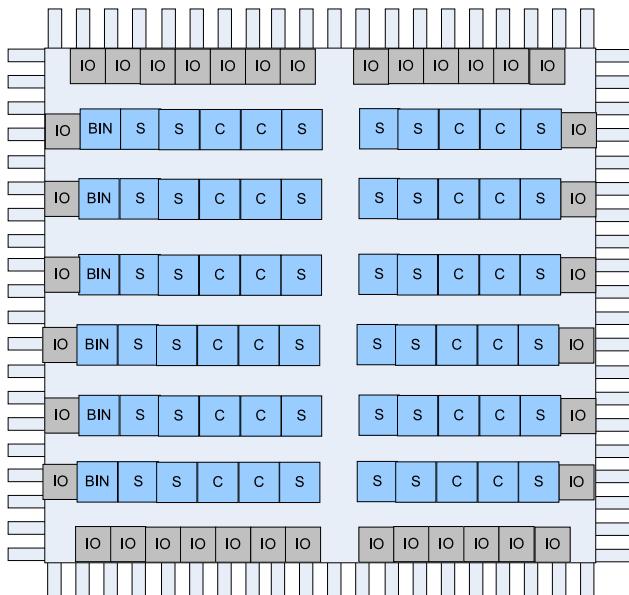
Figure 2.32.   Row based arrangement on the Actel ACT3 FPGA Family
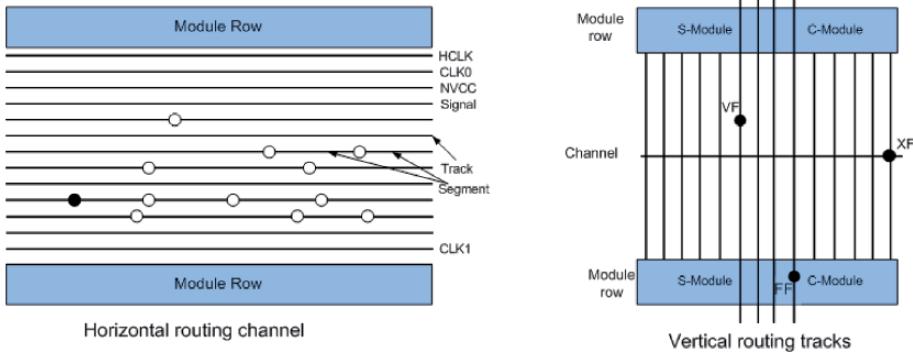
Figure 2.33. Actel's ACT3 FPGA horizontal and vertical routing resources

three types: *input*, *output* and *long*. They are also divided into more segments. Each segment in an input track is dedicated to the input of a particular module, and each segment in the output track is dedicated to the output of a particular module. Long segments are uncommitted and can be assigned during routing. Each output segment spans four channels (two above and two below) except near the top and the bottom of the array. Vertical input segments span only the channel above or the channel below. The tracks dedicated to module inputs are segmented by pass transistors in each module row. During normal user operation the pass transistors are inactive, which isolate the inputs of a module from the inputs of the module above it.

The connections inside Actel FPGAs are established using antifuse. Four types of antifuse connections exist for the ACT3: *horizontal-to-vertical* (XF) connection, *horizontal-to-horizontal* (HF) connection, *vertical-to-vertical* (FF) connection and *fast-vertical* connection (figure 2.33).

4.3.3 Sea-of-gates: The Actel ProASIC family

Like in symmetrical arrays, the macro cells are arranged on a two-dimensional array structure such that an entry in the array correspond to the coordinate of a given macro cell. The difference between the symmetrical array and the sea-of-gate is that there is no space left aside between the macro cells for routing.

The interconnection wires are fabricated on top of the cells. The Actel ProASIC FPGA family is an implementation of the sea-of-gate approach. The ProASIC core consists of a sea-of-gates called *sea-of-tiles*. The macro cells are the EEPROM-based tiles previously seen. The device uses a four level of hierarchy routing resource to connect the logic tiles: the *local resources*, the *long-line resources*, the *very long-line* resources and the *global networks*. The local resources allow the output of the tile to be connected to the inputs of one
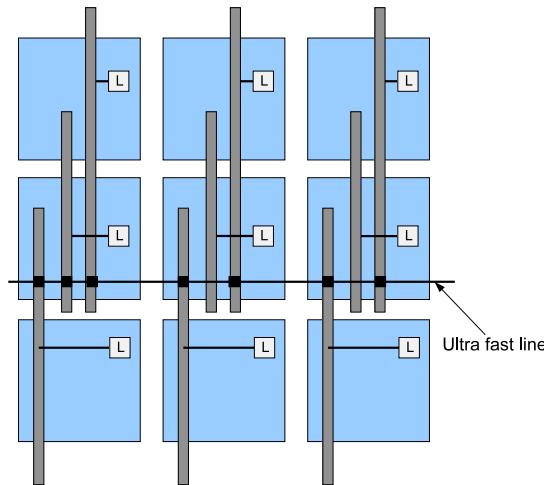
Figure 2.34. Actel ProASIC local routing resources

of the eight surrounding tiles (figure 2.34). The long-line resources provide routing for longer distances and higher fanout connections. These resources, which vary in length (spanning one, two, or four tiles), run both vertically and horizontally and cover the entire device. The very long lines span the entire device. They are used to route very long or very high fanout nets.

4.3.4 Hierarchical-based: The Altera Cyclone, Flex and Stratix families

In hierarchical based FPGAs, macro cells are hierarchically placed on the device. Elements with the lowest granularity are at the lowest level hierarchy. They are grouped to form the elements of the next level. Each element of a level i consists of a given number of elements from level $i - 1$.

Altera FPGAs (FLEX, Cyclone II and Stratix II) have two hierarchical levels. The logic cells (in the Cyclone II and FLEX) and the ALM in the Stratix II are on the lowest level of the hierarchy. The logic array blocks *(LABs)* build the higher level (figure 2.35). Each LAB contains a given number of logic elements (eight for the FLEX8000, ten for the FLEX6000, sixteen for the Cyclone and eight AMLs for the Stratix II). The LABs in turn are arranged as array on the device.

Signal connections to and from device pin are provided via a routing structure called *FastTrack* in the FLEX and *MultiTrack* in Cyclone II and Stratix II). The *FastTrack* as well as the *MultiTrack* interconnects consist of a series of fast, continuous row and column channels that run the entire length and width of the device.
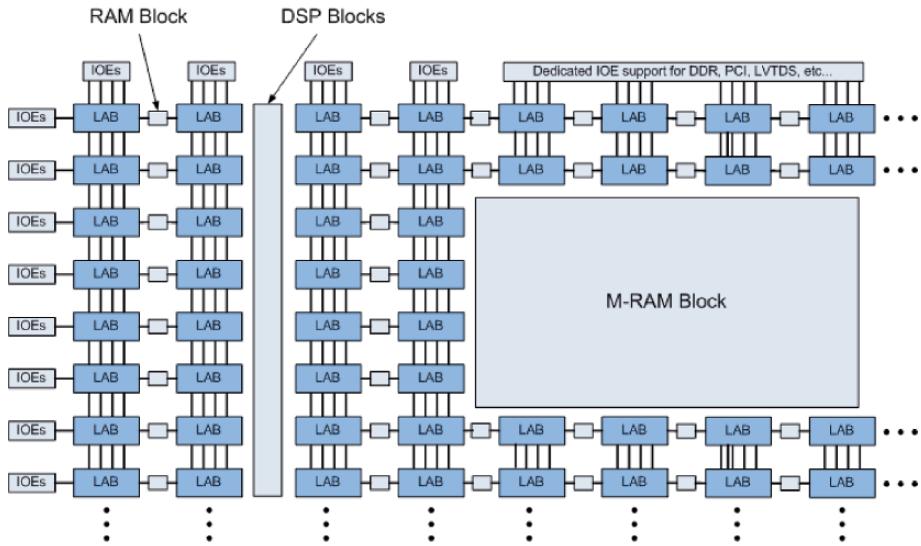
Figure 2.35.   Hierarchical arrangement on the Altera Stratix II FPGA

Signals between LEs or ALMs in the same LAB and those in the adjacent LABs are routed via local interconnect signals (figure 2.36). Each row of a LAB is served by a dedicated row interconnect, which routes signals between LABs in the same row. The column interconnect routes signals between rows and routes signals from I/O pin rows.
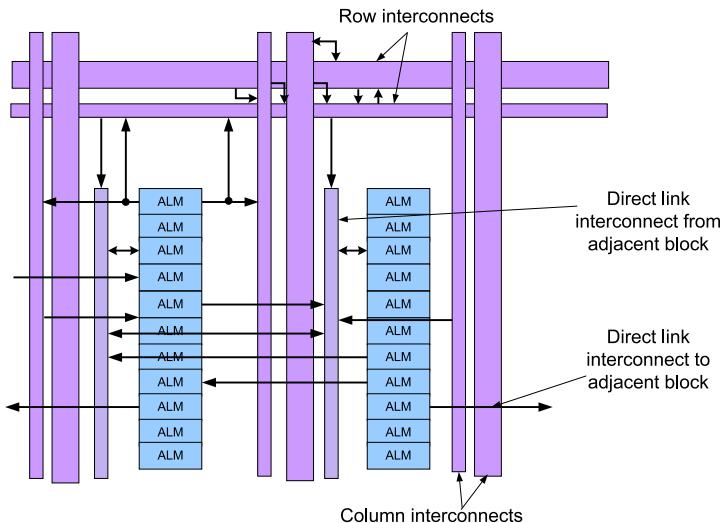
Figure 2.36.   LAB connection on the Altera Stratix devices

A row channel can be driven by an LE (or ALM in the Stratix II) or by one of two column channels. Each column of LABs is served by a dedicated column interconnect. The LEs in an LAB can drive the column interconnect, which can then drive another row's interconnect, to route the signals to other LABs in the device. A signal from the column interconnect must be routed to the row interconnect before it can enter an LAB (figure 2.36).

LEs can drive global control signals. This is helpful for distributing the internally generated clock, asynchronous clear and asynchronous preset signals and high-fan-out data signals.

4.3.5   Programmable I/O

Located around the periphery of the device, I/O components allow for the communication of a design inside the FPGA with off-chip modules. Like the logic cells and the interconnections, FPGA I/Os are programmable, which allows designs inside the FPGA to configure a single interface pin as input, output or bidirectional.

The general structure of an I/O component is shown in figure 2.37: It consists of an input block, an output block and an output enable block for driving the tri-state buffer. Two registers that are activated either by the falling or by the rising edge of the clock are available in each block.

The I/Os can be parameterized for a single data rate (SDR) or a double data rate (DDR) operation mode. Whereas in the SDR-mode, data are copied into the I/O registers on the rising clock edge only, the DDR mode exploits the falling clock edge and the rising clock edge to copy data into the I/O registers. On the input, output, tri-state, one of the double data rate (DDR) register can be
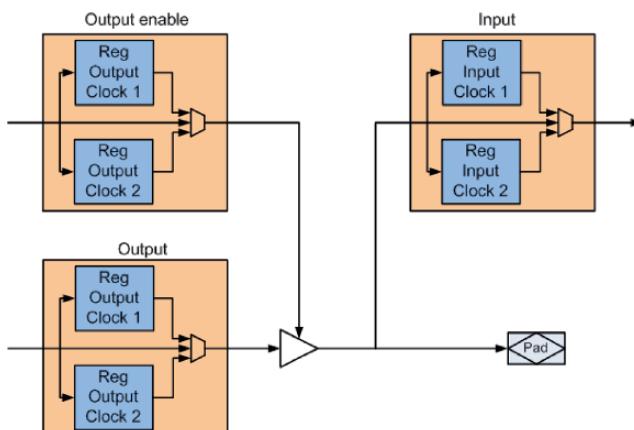
Figure 2.37.   General structure of an I/O component

used. The double data rate is directly accomplished by the two registers on each path, clocked by the rising edge (or the falling edge) from different clock nets.

DDR input can be done using both input registers whereas DDR output will use both output registers. In the Altera Stratix II, the I/O component is called I/O element (IOE). The IOEs are located in I/O blocks around the periphery of the device.

The I/O blocks, which contain up to four IOEs, are used to drive the rows and columns interconnects. They are divided in two groups: The row I/O blocks, which drive row, column or direct link interconnects, whereas the column I/O blocks drive column interconnects.

The Xilinx Virtex I/O components are called I/O block (IOB), and they are provided in groups of two or four on the device boundary. The IOB can be used independent from each other as input and/or output, or they can be combined in group of two to be used as differential pair directly connected to a switch matrix.

4.4   Hybrid FPGAs

The process technology as well as the market demand is pushing manufacturers to include more and more pre-designed and well-tested hard macros in their chips. Resources, such as memory, that are used in almost all designs can be directly be found on the chip. This allows the designer to use well-tested and efficient modules. Moreover, hard macros are more efficiently implemented and are faster than macro implemented on the universal function generators. The resources often available on hybrid FPGAs are RAMs, clock managers, arithmetic modules, network interface modules and processors. The market demand has pushed almost all the manufactures to include hard macros in all their devices. Because the resources required by the users vary with their application class, some manufacturers, such as Xilinx, provided different classes of FPGAs, each of which is suited for a given purpose. The most emerging classes as classified by Xilinx are the system on chip (SoC), digital signal processing (DSP) and pure logic. The system on chip class to which the Virtex 4 FX belongs is characterized by embedded processors directly available on the chip, memory and dedicated bus resources. Reference designs also exist, which efficiently use the chip resource to get the maximum performance. The DSP class, which contains the Virtex 4 SX is characterized by the abundance of multipliers macros and the pure logic class, to which the Virtex 4 LX belongs, is dominated by LUT function generators.

Figure 2.38 depicts the architecture of the Xilinx Virtex II Pro, one of the first FPGAs to include Hard macro, among which complete processors.

The Xililnx Virtex II Pro contains up to four embedded IBM Power PC 405 RISC hard core processors, which can be clocked at more than 300 MHZ. Embedded high-speed serial *RocketIO* transceivers with up to 3.125 Gb/s per
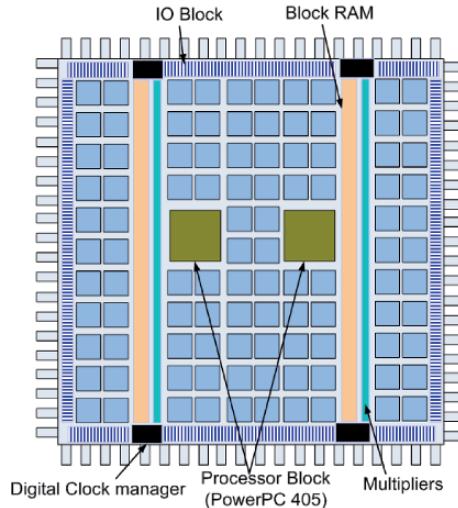
Figure 2.38. Structure of a Xilinx Virtex II Pro FPGA with two PowerPC 405 Processor blocks

channel, internal *BlockRAM* memory module to be used as dual-ported RAM, embedded 18×18-bit multiplier blocks, digital clock manager (DCM) to provide self-calibrating, fully digital solution for clock distribution delay compensation, clock multiplication and division, and coarse-and fine-grained clock phase shifting are available on the chip.

5.   Coarse-Grained Reconfigurable Devices

FPGAs allow for programming any kind of function as far as this can fit onto the device. This is only possible because of the low granularity of the function generators (LUT and MUX). However, the programmable interconnections used to connect the logic blocks reduce the performance of FPGAs. A way to overcome this is to provide frequently used module as hard macro, as it is the case in hybrid FPGAs, and therefore, to allow programmable interconnections only between processing elements available as hard macros on the chip. Coarse-grained reconfigurable devices follow this approach. In general, those devices are made upon a set of hard macros (8-bit, 16-bit or even a 32-bit ALU), usually called *processing element* (PE). The PEs are able to carry few operations such as addition, subtraction or even multiplication. The interconnection is realized either through switching matrices or dedicated busses. The configuration is done by defining the operation mode of the PEs and programming the interconnection between the processing elements.

In the last couple of years, several coarse-grained architectures were built by different companies, some of which went bankrupt. Despite all the hope placed in coarse-grained reconfigurable device, they fail until now to provide

a large acceptance, and their future is not really easy to predict. This does not mean that the philosophy behind coarse-grained device is wrong. Companies investigating coarse-grained reconfigurable devices must also face the FPGA competition, dominated by large companies that provides many coarse-grained elements in their devices according to the market need.

A wide variety of coarse-grained reconfigurable devices that we classify into three categories will be presented in this section: in the first category, the *dataflow machines*, functions are usually built by connecting some PEs to build a functional unit that is used to compute on a stream of data. In the second category are the *network-based* devices in which the connection between the PEs is done using messages instead of wires. The third category are the *embedded FPGA* devices, which consist of a processor core that cohabit with a programmable logic on the same chip.

5.1   Dataflow Machines

Dataflow machines are the most dominating coarse-grained reconfigurable devices. In this section, we present three of those architectures: the PACT-XPP, the NEC-DRP and the PicoChip devices.

5.1.1   The PACT XPP device

The idea behind the PACT XPP architecture [17] [176] is to efficiently compute streams of data provided from different sources such as A/D converters rather than single instructions as it is the case in the Von-Neumann computers. Because the computation should be done while data are streaming through the processing elements, it is suitable to configure the PEs to adapt to the natural computation paradigm of a given application or part of it at a given time. The eXtreme Processing Platform (XPP) architecture of PACT consist of:

- An array of *processing array elements (PAE)* grouped in *processing array (PA)*

- A communication network

- A hierarchical configuration tree

- Memory elements aside the PAs

- A set of I/O elements on each side of the device.

One configuration manager (CM) attached to a local memory is responsible for writing configuration onto a PA. The configuration manager together with PA build the processing array cluster (PAC). An XPP chip contains many PACs arranged as grid array on the device. Figure 2.39 shows an XPP device with four PACs, each of which contains 4 PAEs and surrounded by memory blocks. The CMs at a lower level are controlled by a CM at the next higher level. The
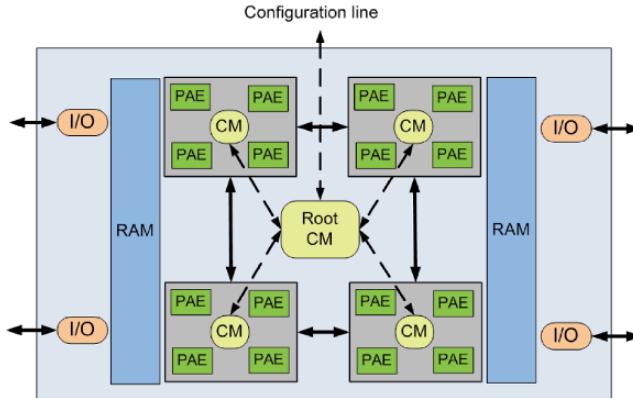
Figure 2.39. Structure of the PACT XPP device

root CM at the highest level is attached to an external configuration memory and supervises the whole device configuration.

The Processing Array Element (PAE). There exist two different kinds of PAEs: the ALU PAE and the RAM-PAE. An ALU-PAE contains an ALU that can be configured to perform basic arithmetic operations, whereas the RAM-PAE is used for storing data. The back-register (BREG) provides routing channels for data and events from bottom to top, additional arithmetic and register functions whereas the forward-register (FREG) is used for routing the signals from top to bottom and for the control of dataflow using event signals. All objects can be connected to horizontal routing channels using switch-objects. Dataflow register (DF-Registers) can be used at the object output for data buffering in case of a pipeline stall. Input registers can be pre-loaded by configuration data and always provide single cycle stall.

A RAM-PAE is similar to an ALU-PAE. However, instead of an ALU, a dual ported RAM is used for storing data. The RAM generates a data packet after an address was received at the input. Writing to the RAM requires two data packet: one for the address and the other for the data to be written. Figure 2.40 shows an ALU-PAE. The structure is the same for a RAM-PAE; however, an RAM is used instead of the ALU.

Routing and Communication. The XPP interconnection network consists of two independent networks: one for data transmission and the other for event transmission (figure 2.40). These two networks consist of horizontal and vertical channels. The vertical channels are controlled by the BREG and FREG whereas connection to horizontal channel is done via switch elements. Besides the horizontal and vertical channels a configuration bus exists, which allows the CMs to configure the PAEs.
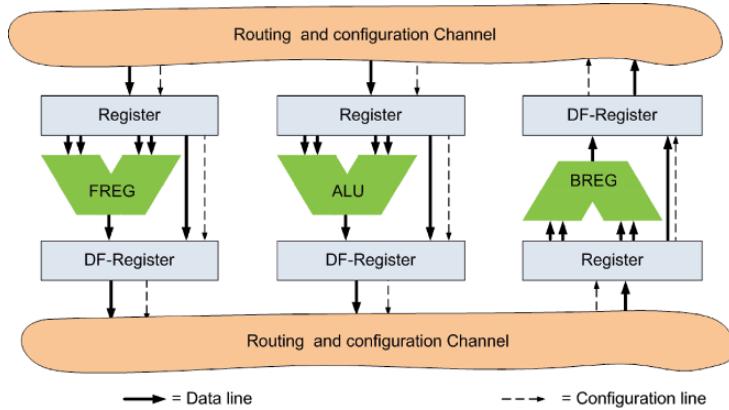
Figure 2.40. The XPP ALU Processing Array Element. The structure of the RAM ALU is similar.

Horizontal buses are used to connect a PAE within a row whereas the vertical buses are used to connect objects to a given horizontal bus. Vertical connections are done using configurable switch objects that segment the vertical communication channels. The vertical routing is enabled using register-objects integrated into the PAEs.

Interfaces. XPP devices provide communication interfaces aside the chip. The number may vary from device to device. The XPP64-A1 for example contain six external interfaces consisting of four identical general purpose I/O interfaces on the chip corner (bottom left, upper left, bottom right and upper right), one configuration manager interface and a JTAG compliant interface for debugging and testing purpose.

The I/O interfaces can operate independently from each other either in RAM, or in streaming mode. In streaming mode, each I/O element provides two bidirectional ports for data streaming. Handshake signals are used for synchronization of data packets to external ports. In RAM mode, each port can access external synchronous SRAMs with 24-bit addresses and 24-bit data. Control signals for the SRAM transactions are available such that no extra logic is required.

The configuration manager interface consists of three subgroups of signals: *code*, *message send* and *message receive*. The code group provides channels over which configuration data can be downloaded to the device whereas the send and receive groups provide communication channels with a host processor.

5.1.2 The NEC DRP Architecture

We now present the NEC dynamically reconfigurable processor (DRP), which operates in a similar way as the PACT. However, the reconfiguration control is
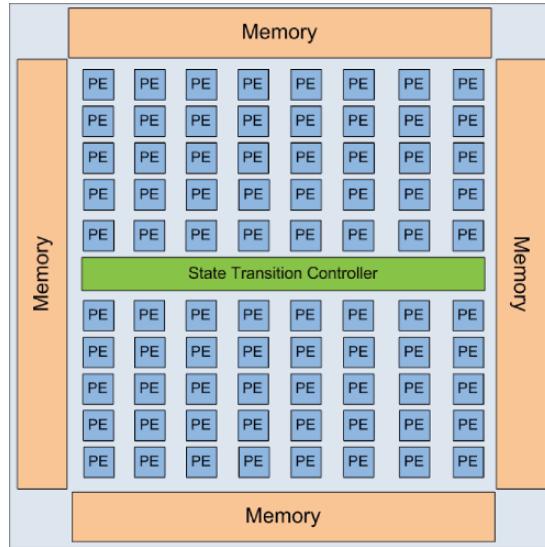
Figure 2.41. Structure of the NEC Dynamically Reconfigurable Processor

not done in a hierarchical fashion such as in the PACT devices. Figure 2.41 shows the overall structure of a NEC DRP. It consist of

- an array of byte-oriented processing elements

- a programmable interconnection network to connect the processing elements

- a sequencer (*State Transition Controller*) which can be programmed as finite state machine to control the dynamic reconfiguration process on the device

- memory blocks to store configuration and computation data. The memory blocks are arranged around the device

- various interfaces and RAM controllers such as PCI, PLL, SDRAM/SRAM

As shown in figure 2.42, a DRP processing element contains an ALU for byte-oriented arithmetic and logic operation, a data management unit to handle byte selects, shift, mask and constant generation. The operand can be fetched from the PE's register file or collected directly from the PE's inputs. The results can be stored in the PE's register file or can be sent to the output. The configuration is done by the *state transition controller* that set a pointer to a corresponding instruction register according to the operation mode of the
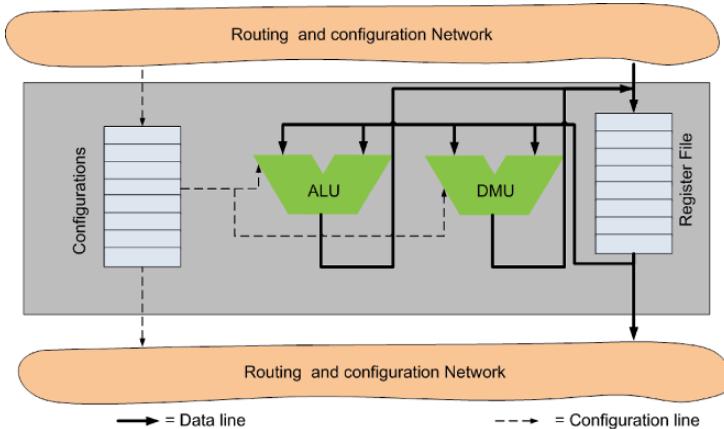
Figure 2.42.   The DRP Processing Element

ALU. Having many configuration registers allow for storing configuration data directly in the proximity of the PEs and allow a fast switching from one configuration to the next one.

5.1.3   The *picoChip* Reconfigurable Device

A *picochip* consists of a *picoArray* core and a set of different interfaces for external connection of various module such as memory and processors. The *picoArray* core has a similar structure such as the NEC DRP and the PACT; however, the connection between the PES is done at the column and row intersection. The PC102 Chip for example contains hundreds of array elements, each with a versatile 16-bit processor with local data memory connected by a programmable interconnect structure (figure 2.43). The architecture is heterogeneous, with four types of processing element all having a common basic structure, but optimized for different tasks: the *standard AE* (STAN), the *control AE* (CTRL), the *Memory* AE (MEM) and the *function accelerator unit* (FAU).

A standard AE type includes multiply-accumulate peripheral as well as special instructions optimized for CDMA operations. The *memory AE* contains multiply unit and additional memory. The *function accelerator* unit is a co-processor optimized for specific signal-processing tasks. The *control AE* is equipped with a multiply unit and larger amounts of data and instruction memory optimized for the implementation of base station control functionality.

Multiple elements can be programmed together as a group to perform particular function. The device can be reconfigured at run-time to run different applications such as wireless protocols. Several interfaces are available for
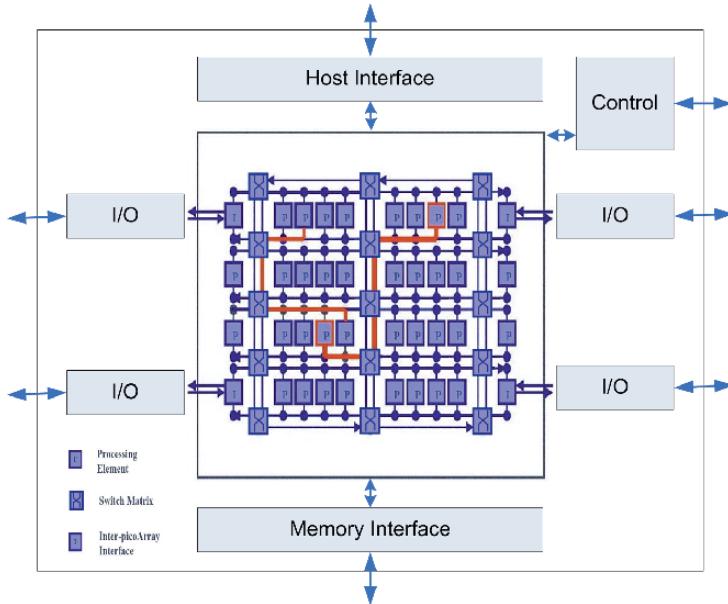
Figure 2.43. Structure of the picoChip device

seamless external connection (figure 2.43): each chip has four inter processor communications links that can be used to build an array of several picoChips to implement larger and complex functions that cannot fit in only one picoChip. The communication on those links is based on a time division multiplexing (TDM) protocol scheme. Besides the inter processor communication, a micro-processor interface is available for connecting an external processor that can be used to configured the device and stream data into the device. External storage (EEPROM, FLASH) can be connected as well to this interface to allow a self reconfiguration of the device on start up. Other interfaces are provided among which a memory interface for connection of external memory and a JTAG interface for debugging purpose.

5.2   Network-oriented architectures

Despite the interest on networks on chip as communication paradigm has grown recently, very few reconfigurable devices rely on message passing for data exchange among the PEs. One of the few companies to have implemented this concept is Quicksilver Tech, a start-up that failed to commercialized their technology and, recently, went bankrupt. Nevertheless, we believe that their architecture merits more attention. We therefore decide to present some details on it in this section.

5.2.1   The Quicksilver ACM Architecture

The adaptive computing machine (ACM) is based on a revolutionary network on chip paradigm and is one of the very few devices that work on such a principle.

The Quicksilver ACM consists of a set of heterogeneous computing nodes hierarchically arranged on a device. At the lowest level, four computing nodes are placed in a cluster and connected locally together. Many clusters at a given level are put together to build bigger clusters at the next higher level (figure 2.44).

An ACM chip consists of the following elements:

- a set of heterogeneous *processing nodes (PN)*

- an homogenous *matrix interconnect network (MIN)*

- a *system controller*

- various I/O interfaces.

The ACM Processing Node structure.   An ACM processing nodes consist of

- an *algorithmic engine* that defines the node type. The node type can be customized at compile-time or at run-time by the user to match a given algorithm. Four types of nodes exist: The P*programmable scalar node (PSN)*
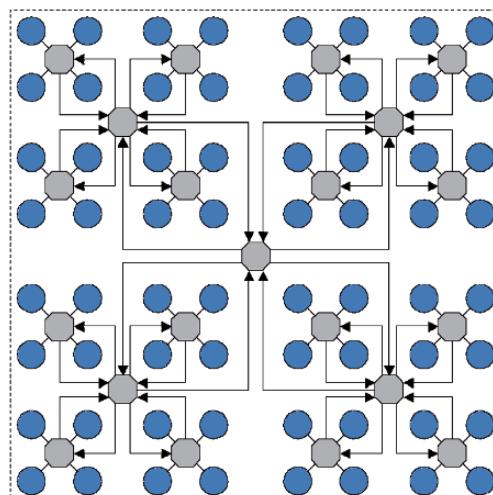
Figure 2.44.   The Quicksilver ACM hierarchical structure with 64 nodes

provides a standard 32-bit RISC architecture with 32 general purpose registers, the *adaptive execution node* (AXN) provides variable word size multiply accumulate (MAC) and ALU operations, the *domain bit manipulation* (DBN) node provides bit manipulation and byte oriented operations, and the *external memory controller* node provides DDRRAM, SRAM, memory random access and DMA control interfaces for off-chip memory access

- The *node memory* for data storage at node level

- A *node wrapper* that hides the complexity of the network architecture. It contains an MIN interface to support communication, a hardware task manager for task managements at node level and a DMA engine. The wrapper envelops the algorithmic engine and presents an identical interface to the neighbouring nodes. It also incorporates dedicated I/O circuitry, memory, memory controllers and data distributors and aggregators (figure 2.45).

The Matrix Interconnect Network (MIN). The communication inside an ACM chip is done via an MIN, which is organized hierarchically. At a given level, the MIN connects many lower level MINs. The top level MIN, the MIN root, is used to access the nodes from outside and to control the configuration of the nodes. The communication among nodes is done via the MIN with the help of the node wrapper. The MIN provides a diversity of services such as point-to-point dataflow streaming, real-time broadcasting, direct memory access and random memory access. The ACM chip also contains various I/O interfaces accessible via the MIN for testing (JTAG) and communication with off-chip devices (figure 2.45(b)).

(a) The ACM node structure

(b) The ACM QS2412 Resources
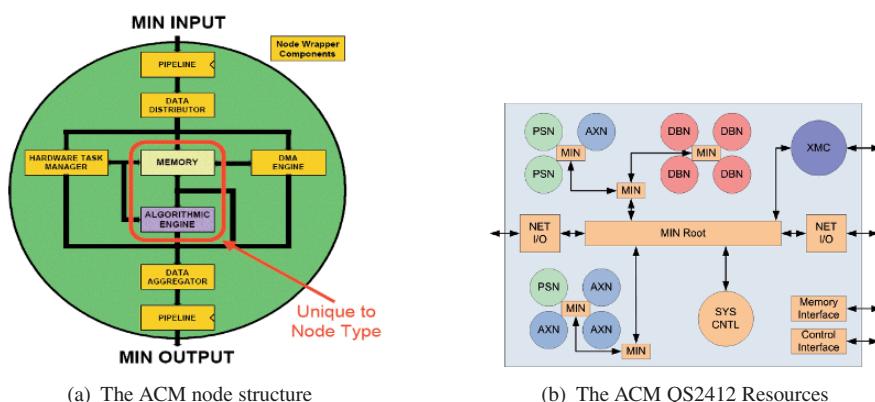
Figure 2.45. ACM Node and Routing resource

System Controller.   The system management is done via an embedded system controller (figure 2.45(b)). The system controller loads tasks into the node's ready-to-run queue for execution, statically or dynamically sets the communication channels between the processing nodes. Any node can be adapted (reconfigured) at run-time by the system controller to perform a new function, in a cycle-by-cycle manner.

5.3   Embedded FPGA

Embedded programmable logic devices, also known under the name *embedded FPGA*, usually integrate a processor core, a programmable logic or FPGA and memory on the same chip. Strictly seen, several existing FPGAs such as the Xilinx Virtex 4 FX fall under this category, as they provides the same elements like any other embedded FPGA device. However, the processor in the Xilinx Virtex 4 FPGAs is immersed in the programmable logic, whereas it is strictly decoupled from the logic in common embedded FPGAs. Two examples of such devices are the DAP/DNA from *IPflex* and the S500 series from Stretch.

5.3.1   The IPflex DAP/DNA Reconfigurable Processor

The DAP/DNA consists of an integrated DAP RISC processor,[1] a *distributed network architecture* (DNA) matrix and some interfaces (figure 2.46). The processor controls the system, configures the DNA, performs computations in parallel to the DNA and manages the data exchange on the device.

The DNA Matrix 2.46 is a dataflow accelerator with more than hundred dynamic reconfigurable processing elements. The wiring among elements can be changed dynamically, therefore, providing the possibility to build and quickly change parallel/pipelined processing system tailored to each application. The DNA configuration data are stored in the configuration memory from where it can be downloaded to the DNA on a clock-by-clock basis. Like in other reconfigurable devices, several interfaces exist for connecting the chip to external devices.

Large applications can be partitioned and sequentially executed on a DAP/DNA chip. While the processor controls the whole execution process, the DNA executes critical parts of the application.

5.3.2   The Stretch Processor

The second example of embedded FPGA that we present is the S5000 series from Stretch. The device consists of a 32-bit Xtensa RISC processor from

---

[1]That is a 32-bit RISC processor operating at 100 MHz with a data cache of 8K byte and an instruction cache of 8K byte.
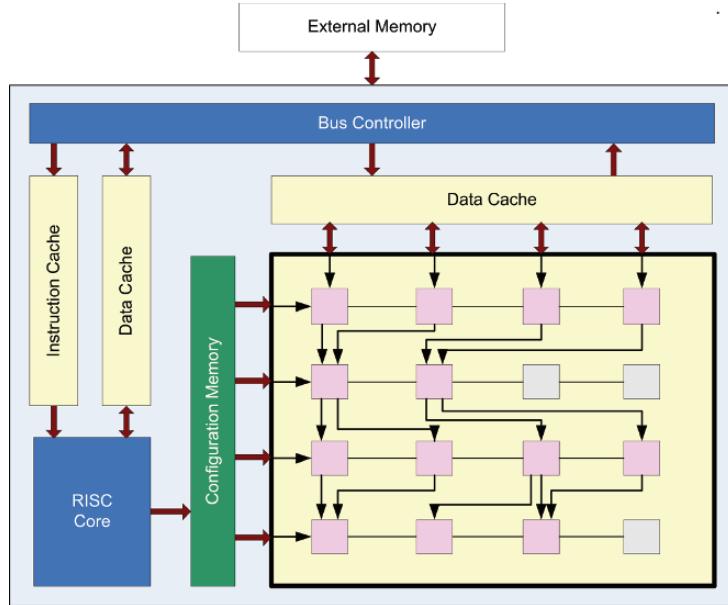
Figure 2.46. IPflex DAP/DNA reconfigurable processor

Tensilica, operating at 300 MHz and featuring a single precision floating point unit, an *instruction set extension fabric (ISEF)*, embedded memory and a set of peripheral control modules (figure 2.47).

Like the DAP/DNA, the processor controls the whole system and configures the ISEF. The ISEF is used to augment the processor capacity by implementing
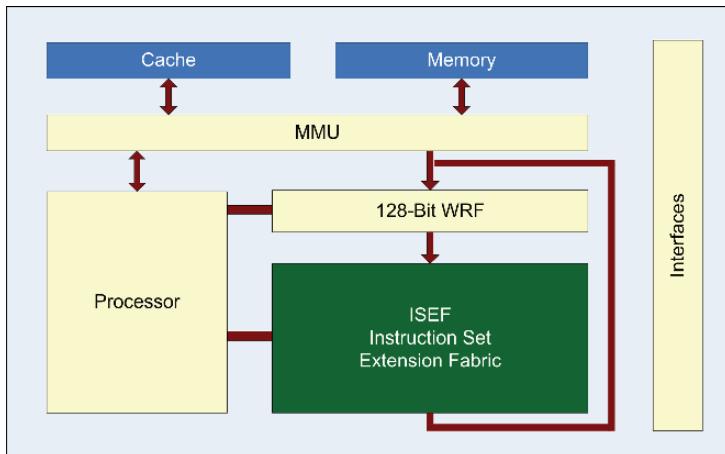
Figure 2.47. The Stretch 5530 configurable processor

additional instructions that are directly accessible by a program running on the processor. The S5 device uses thirty-two 128-bit wide registers coupled with 128-bit wide access to memory to feed data to the ISEF. Several interfaces, connected to a peripheral bus exist for communication with the external world. The S5530 peripherals are divided into three categories as follows: the *low speed peripherals* that includes UART, IrDA, serial peripheral interface (SPI). The second category consists of the *mid-speed peripherals* that include parallel generic interface bus (GIB) for Flash and SRAM connection. The last category are the *high-speed peripherals* that include a 64-bit PCI/PCI-X port and one 64-bit SDRAM port.

5.4   Academic Efforts

Besides commercial coarse-grained reconfigurable devices, several other coarse-grained devices were developed in the academic world. Starting from 1990, several approaches for coarse-grained device, some of which are described in this section, have been published [110].

5.4.1   The MATRIX Architecture

The multiple ALU architecture with reconfigurable interconnect experiment (MATRIX) [159] comprises an array of identical basic functional units (BFUs). Each BFU contains an 8-bit ALU, 256 words of 8-bit memory and control logic. The ALU features the standard set of arithmetic and logic functions and a multiplier. A configurable carry-chain between adjacent ALUs can be used to cascade ALUs for wide-word operations. The control logic can generate local control signals from ALU output by a pattern matcher. A reduction network can be employed for control generated from neighbouring data. Finally, a 20-input, 8-output NOR block may be used as half of a PLA to produce control signals. According to these features, a BFU can serve as an instruction memory, a data memory, a register-file-ALU combination or an independent ALU function. Instructions can be routed over the array to several ALUs. The routing fabric provides three levels of 8-bit buses: eight nearest neighbour and four second nearest neighbour connections of length four, and global lines spanning an entire row or column.

5.4.2   RAW: Reconfigurable Architecture Workstation

The idea of RAW [210] is to provide a simple and highly parallel computing architecture composed of several repeated tiles connected to each other by nearest neighbor connections. The tiles comprise computation facilities as well as memory, thus implementing a distributed memory model. A RAW microprocessor is a homogeneous array of processing elements called tiles. The prototype chip features 16 tiles arranged in a 4×4 array. Each tile comprises

a simple RISC-like processor consisting of ALU, register file and program counter, SRAM-based instruction and data memories, and a programmable switch supplying point-to-point connections to NN. The CPU in each tile of the prototype is a modified 32-bit MIPS R2000 processor with an extended 6-stage pipeline, a floating point unit and a register file of 32 general purpose and 16 floating point registers. Both the data memory and the instruction memory consist of 32-kilobytes of SRAM. While the instruction memory is uncached, the data memory can operate in cached and uncached mode. Interconnection of the RAW architecture is done over nearest neighbour connections being optimized for single data word transfer. Communication between tiles is pipelined over these connections and appears at register level between processors, making it different from multiprocessor systems. A programmable switch on each tile connects the four nearest neighbour links to each other and to the processor. The RAW architecture provides both a static and a dynamic network with wormhole routing for the forwarding of data.

5.4.3 REMARC: Reconfigurable Multimedia Array Coprocessor

REMARC [160] is a reconfigurable coprocessor that is tightly coupled to a main RISC processor. It consists of an 8×8 array of 16-bit programmable logic units called *nanoprocessor*, which is attached to a global control unit. The control unit manages data transfers between the main processor and the reconfigurable array and controls the execution on the nanoprocessors. It comprises an instruction RAM with 1024 entries, 64-bit data registers and four control registers. The nanoprocessor consists of a 32-entry local instruction RAM, an ALU with 16-bit datapath, a data RAM with 16 entries, an instruction register, eight data registers, four 16-bit data input registers and a 16-bit data output register. The ALUs can execute 30 instructions, including addition, subtraction, logical operations, shift instructions, as well as some operations often found in multimedia applications such as minimum, maximum, average, absolute and add. Each ALU can use data from the data output registers of the adjacent processors via nearest neighbor connect, from a data register, a data input register, or from immediate values as operands. The result of the ALU operation is stored in the data output register. The communication lines consist of nearest neighbour connections between adjacent nanoprocessors and additional horizontal and vertical buses in each row and column. The nearest neighbour connections allow the data in the data output register of a nanoprocessor to be sent to any of its four adjacent neighbors. The horizontal and vertical buses have double width (32-bit) and allow data from a data output register to be broadcasted to processors in the same row or column, respectively. Furthermore, the buses can be used to transfer data between processors, which are not adjacent to each other.

5.4.4   *MorphoSys*

The complete *MorphoSys* [194] chip comprises a control processor (*TinyRISC*), a frame buffer (data buffer), a DMA controller, a context memory (configuration memory) and an array of 64 reconfigurable cells (RC). Each RC comprises an ALU-multiplier, a shift unit, and two multiplexers at the RC inputs. Each RC also has an output register, a feedback register and a register file. A context word, loaded from the configuration memory and stored in the context register, defines the functionality of the RC. Besides standard logic/arithmetic functions, the ALU has other functions such as computation of absolute value of the difference of two operands and a single cycle multiply-accumulate operation. There are a total of 25 ALU functions. The RC interconnection network features three layers. In the first layer, all cells are connected to their four nearest neighbours. In the second layer, each cell can access data from any other cell in the same row or column of the same array quadrant. The third layer of hierarchy consists of buses spanning the whole array and allowing transfer of data from a cell in a row or column of a quadrant to any other cell in the same row or column in the adjacent quadrant. In addition, two horizontal 128-bit buses connect the array to the frame buffer.

5.4.5   NISC Processor: No-Instruction-Set Computer

The NISC [92] processor consists of a Controller and Datapath on which any C program can be executed on it. The datapath consists of a set of storage elements (registers, register files, memories), functional units (ALUs, multipliers, shifters, custom functions) and a set of busses. Each component may take one or more clock cycles to execute, each component may be pipelined and each component may have input or output latches or registers. The entire Datapath can be pipelined in several stages in addition to components being pipelined themselves. The Controller defines the state of the processor and issues the control signals for the Datapath. The Controller can be fixed or programmable, whereas the Datapath can be reprogrammable and reconfigurable. Reprogrammable means that the Datapath can be extended or reduced by adding or omitting some components, while reconfigurable means that the Datapath can be reconnected with the same components. To speed up the NISC pipelining, a control register (CR) and a status register (SR) has been inserted between the Controller and the Datapath.

5.4.6   Virtual Pipelines: The *PipeRench*

Pipeline reconfiguration was proposed by Goldstein et al. [98] as a method of virtualizing pipelined hardware application designs. The single static reconfiguration is broken into pieces that correspond to pipeline stages in the application. The resulting configurations are then loaded into the device on a

cycle-by-cycle basis. With pipeline virtualization, an application is pipelined implemented on a given amount of virtual resources. The virtual resources are then mapped to the physical resources in a final step. The physical resources are computing units modules, whose functionality can be changed by reconfiguration.

The process is somehow similar to the implementation on parallel machines, where virtual processors are first used to easily capture the inherent parallel structure of an application. The virtual processors are then mapped to the physical one in the next step. Virtualization provides the designer freedom to better explore the parallelism in the implementation, and he/she does not need to face the resource constraint of a given platform.

Figure 2.48 illustrates the virtualization processing on the mapping of a five-stage virtual pipeline on a three-stage physical fabric. On the top part of this figure, we see the five-stage application and the state of each of the stages of the pipeline in the five consecutive cycles. The bottom half of the figure shows the mapping of the virtual blocks on the physical modules of the device. Because the configuration of each single unit in the pipeline is independent from the other, the reconfiguration process can be broken down to a cycle-by-cycle configuration. In this way, part of the pipeline can be reconfigured while the rest is computing.
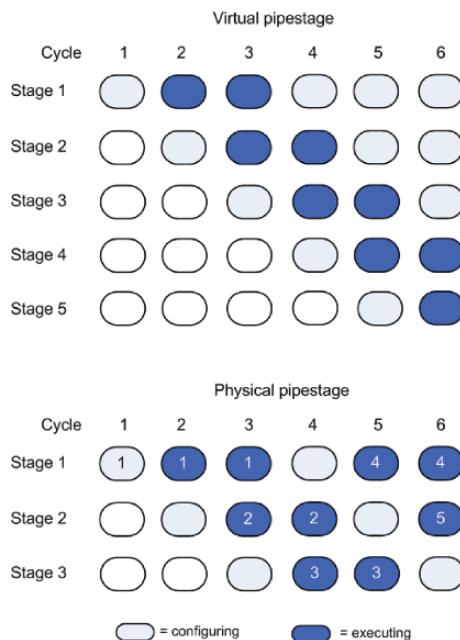
Figure 2.48. Pipeline Reconfiguration: Mapping of a 5 stage virtual pipeline auf eine 3 stage

Based on the pipeline reconfiguration concept, a class of reconfigurable devices called *PipeRench* was proposed in [98] as co-processor in multimedia applications. A PipeRench device consists of a set of physical *pipeline stages* also called *stripes*. A stripe is composed of interconnect and processing elements (PE), each of which contains registers and LUT-based ALUs. The PEs access operands from registered outputs of the previous stripe as well as registered or unregistered outputs of the other PEs in the stripe. All *PipeRench* devices have four global busses, two of which are dedicated to storing and restoring stripe state during hardware virtualization (configuration). The other two are used for input and output.

5.4.7   RaPiD

The last work that we consider in this section is the one of Ebeling et al. [100] whose goal was to overcome the handicap of FPGA.[2] A one-dimensional coarse-grained reconfigurable architecture called *RaPiD*, an acronym for *reconfigurable pipelined datapath* is proposed for this purpose.

The structure of RaPiD datapaths resembles that of systolic arrays. The structure is made upon linear arrays of functional units communicating in mostly a nearest neighbour fashion. This can be used for example, to construct a hardware module, which comprises different computations at different stages and at different times resulting in a linear array of functional units that can be configured to form a linear computational pipeline. The resulting array of functional units is divided into identical cells that are replicated to form a complete array. A RaPiD-cell consists of an integer multiplier, two integer ALUs, and six general purpose registers and three small local memories. Interconnections among the functional units are realized using a set of ten segmented busses that run the length of the datapath. Many of the registers in a pipelined computation can be implemented using the bus pipeline registers.

Functional unit outputs are registered. However, the output registers can be bypassed via configuration control. Functional units may additionally be pipelined internally depending on their complexity.

The control of the datapath is done using two types of signals: the *static control signals* that are defined by the configuration memory as in ordinary FPGAs and *dynamic control* that must be provided on every cycle. To program an application on the RaPiD, a mapping of functional blocks to computing elements of the datapath must be done, which result on the generation of a static programming bitstream. This will be used to construct the pipeline, and the dynamic programming bits are used to schedule the operations of the computation

---

[2]In this case, the large amount of resources deployed to build macro instructions and the difficulty in programming the FPGAs.

onto the datapath over time. A controller is programmed to generate the dynamic information needed to produce the dynamic programming bits.

5.5   Capacity / Chip Size

The last point we deal with in the architecture is the definition of a measure of comparison between the RPUs. The most used measure of comparison is the *device capacity*, which is usually provided by the manufacturer as the number of gates that can be used to build a function into the device. On fine-grained reconfigurable devices, the number of gates is often used as unit of measure for the capacity of an RPU. A *gate equivalent* corresponds to a two-input NAND gate, i.e. a circuit that performs the function $F = \overline{A \cdot B}$. The *gate density defines* the number of gates per unit area. Besides the capacity of an RPU, others factors such as the number of pins and the device speed may also play a big role in the comparison.

Coarse-grained reconfigurable devices do not have any established measure of comparison. However, the number of PEs on a device as well as their characteristics such as the granularity, the speed of the communication link, the amount of memory on the device and the variety of peripherals may provide an indication on the quality of the device.

Although the capacity of the device defines the "*amount of parallelism*", which can be implement in the device, the speed gives an indication on the throughput of data in the device. Designers should however bear in mind that the real speed and real size that a design can achieve depends on the implementation style and the compilers used to produce the designs.

6.   Conclusion

Our goal in this chapter was not to provided all possible details on the single architectures of the reconfigurable computer chips presented. We rather focus on the main characteristics in the technology as well as the operations. More details on the single devices can be found in the corresponding datasheet. Despite the large amount of architecture presented in this section, the market is still dominated by the FPGAs, in particular those from Xilinx and Altera. The coarse-grained reconfigurable device's market has not really taken-off so far, despite the large amount of concept and prototypes developed in this direction.

Research and development in the architecture of reconfigurable computing systems is very dynamic. By the time this book is published, some concept presented in this section will probably be no more actual. However, as we have seen with the FPGAs, the basic structure of reconfigurable device will still remain the same in the future. In the case of FPGA, we will experience some changes in the amount of input of the LUT, modifications in the I/O elements and also the inclusion of various coarse-grained element in the chip.

However, LUTs will still be used as computational connected to each other using programmable crossbar switches. I/O elements will still be used for external communication. Coarse-grained device will still have the same structure consisting of ALU-computing elements, programmable interconnections and I/O components. Understanding the concept presented here will therefore help to understand better and faster the changes that will be made on the devices in the future.

Chapter 3

IMPLEMENTATION

In the first part of this chapter, we present the different possibilities for the use of reconfigurable devices in a system. According to the way those devices are used, the target applications and the systems in which they are integrated, different terminologies will be defined for the systems. We follow up by presenting the design flow, i.e. the steps required to implement an application on those devices. Because the programming of coarse-grained reconfigurable devices is very similar to that of processors, we will not focus on the variety of tool that exists for this purpose. The implementation on FPGA devices is rather unusual in two points. First, the programming is not aimed at generating a set of instructions to be executed sequentially on a given processor. We seek the generation of the hardware components that will be mapped at different time on the available resources. According to the application, the resources needed for the computation of the application will be built as components to be downloaded to the device at run-time. The generation of such components is called logic synthesis. It is an optimization process whose goal is to minimize some cost functions aimed at producing, for instance, the fastest hardware with the smallest amount of resources and the smallest power consumption. The mapping of the application to the FPGA resources is a step of the logic synthesis called technology mapping. The second unusual point with FPGAs is that the technology mapping targets look-up tables rather than NAND Gate as it is the case with many digital devices. In the last part of the chapter, we will therefore shortly present the steps required in logic synthesis and focus in more details on technology mapping for FPGAs.

1.   Integration

Reconfigurable devices are usually used in three different ways:

- **Rapid prototyping**: In this case, the reconfigurable device is used as an emulator for another digital device, usually an ASIC. The emulation process allows to functionally test the correctness of the ASIC device to be produced, sometimes in real operating and environmental conditions, before production. The reconfigurable device is only reconfigured to emulate a new implementation of the ASIC device.

- **Non-frequently reconfigurable systems**: The reconfigurable device is integrated in a running system where it is used as an application-specific processor. These systems are usually stand-alone systems. The reconfiguration is used for testing and mg initialization at start-up and for upgrading purpose. The device configuration is usually stored in an EEPROM of Flash from which it is downloaded at start-up to reconfigure the device. No configuration happens during operation.

- **Frequently reconfigurable systems**: This third category comprises systems, which are frequently reconfigured. Those systems are usually coupled with a host processor, which is used to reconfigure the device and control the complete system.

With the increasing size and speed of reconfigurable processor,[1] it is possible to implement many large modules on a reconfigurable device at the same time. Moreover, for some reconfigurable devices, only a part of the device can be configured while the rest continues to operate. This *partial reconfiguration* capability enables many functions to be temporally implemented on the device. Depending on the time at which the reconfiguration sequence are defined, the computation and configuration flow on a reconfigurable devices can be classified into two categories:

- **Compile-time reconfiguration:** In this case, the computation and configuration sequences as well as the data exchange are defined at compile time and never change during a computation. This approach is more interesting for devices, which can only be fully reconfigured. However, it can be applied to partial reconfigurable devices that are logically or physically partitioned in a set of reconfigurable bins.

- **Run-time reconfiguration:** The computation and configuration sequences are not known at compile time. Request to implement a given task is known at run-time and should be handled dynamically. The reconfiguration process

---

[1]Today, leading edge reconfigurable devices contain million of gates with few hundreds MHz.

exchanged part of the device to accommodate the system to changing operational and environmental conditions. Run-time reconfiguration is a difficult process that must handle side-effect factors such as defragmentation of the device and communication between newly placed modules. The management of the reconfigurable device is usually done by a scheduler and a placer that can be implemented as part of an operating system running on a processor (figure 3.1). The processor can either reside inside or outside the reconfigurable chip.

The scheduler manages the tasks and decides when a task should be executed. The tasks that are available as configuration data in a database are characterized through their bounding box and their run-time. The bounding box defines the area that a task occupies on the device. The management of task execution at run-time is therefore a *temporal placement problem* that will be studied in detail in chapter 5. The scheduler determines which task should be executed on the RPU and then gives the task to the placer that will try to place it on the device, i.e. allocate a set of resources for the implementation of that task. If the placer is not able to find a site for the new task, then it will be sent back to the scheduler that can then decide to send it later and to send another task to the placer. In this case, we say that the task is rejected.
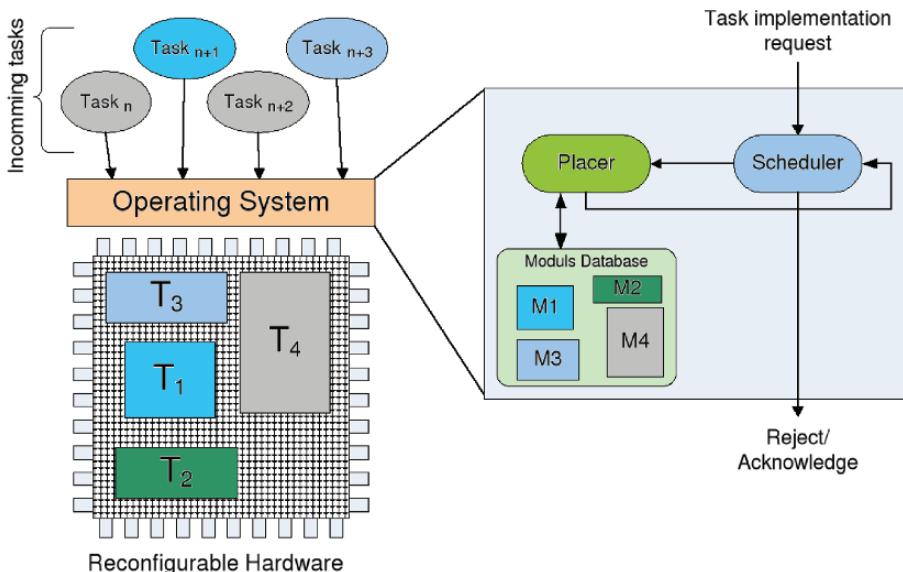
Figure 3.1.   Architecture of a run-time reconfigurable system
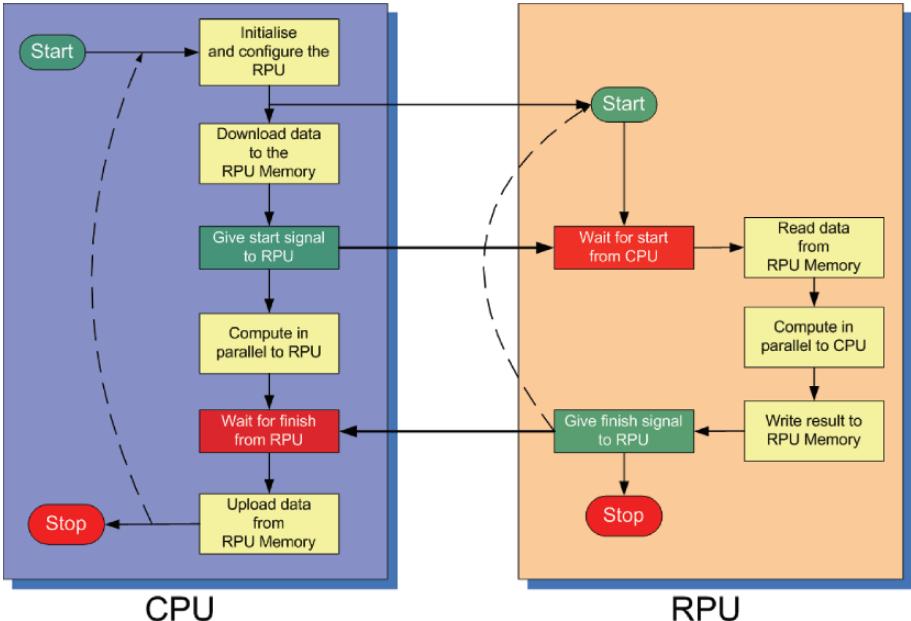
Figure 3.2.   A CPU-RPU configuration and computation step

No matter if we are dealing with a compiled-time or run-time reconfigurable system, the computation and reconfiguration flow is usually the one shown on Figure 3.2. The host CPU is used for device configuration and data transfer. Usually, the reconfigurable device and the host processor communicates through a bus that is used for the data transfer between the processor and the reconfigurable device. In supercomputing, systems are made upon a high-speed processor from Intel or AMD and an FPGA board attached to a bus such as the PCI. Several systems (Cray XD1, SGI, Nallatech SRC Computers), in which FPGAs cohabits and communicate using a dedicated high-speed bus, are available to buy. In embedded systems, the processors are more and more integrated in the reconfigurable devices and are heavily used for management purpose rather than for computation. The RPU acts like a coprocessor with varying instruction sets accessible by the processor in a function call. The computation flow can be summarized as shown in algorithm 2.

At the beginning of a computation, the host processor configures the reconfigurable device.[2] Then it downloads the segment of the data to be processed by

---

[2]It is already possible to trigger the reconfiguration of the Xilinx FPGAs from within the device using their ICAP-port. This might allow a self-reconfiguration of a system from a processor running within the device.

Algorithm 2 CPU-RPU configuration and computation steps

 1: Start
 2: Initialize the RPU
 3: **while** (1) **do**
 4:     Configure the RPU to implement a new task
 5:     Download Data for RPU computation into RPU-memory
 6:     Computes in parallel with the RPU if necessary
 7:     Upload the data computed by the RPU from the RPU-memory
 8: **end while**
 9: Stop

---

the RPU[3] and give the start signal to the RPU. The host and the RPU can then process in parallel on their segments of data. At the end of its computation, the host reads the finish signal of the RPU. At this point, the data (computation result) can be collected from the RPU memory by the processor. The RPU can also send the data directly to an external sink. In the computation flow presented above, the RPU is configured only once. However, in frequently reconfigured systems, several configurations might be done. If the RPU has to be configured more than once, then the body of the while loop must be run again according to the number of reconfigurations to be done before.

The design flow of a dynamic reconfigurable system is primary a hardware/software partitioning process in which:

- The part of the code to be executed on the host processor is determined. This part is usually control dominated.

- The parts of the code to be executed on the reconfigurable device are identified. Those are usually data-dominated parts for which efficient dataflow computation modules are required.

- The interface between the processor and the reconfigurable device is implemented.

The implementation of the control part on the CPU is done in software using the common development languages and tools. It is usually a C-program with system calls to access the device for reconfiguration. We will not focus on the software development details in this book, because enough material that cover software development exist.

No matter if the device is partially reconfigurable or not, the development of the module to be downloaded later on the reconfigurable device follows the

---

[3]The data might also be collected by the RPU itself from an external source.

Manufacturer	Language	Tool	Description
PACT	NML (Structural)	XPP-VC	C into NML and configuration
Quicksilver	SilverC	InSpire SDK	SiverC into Configuration
NEC-DRP	C	DRP Compiler	C into configuration
picoChip	C	Picochip Toolchain	C into configuration
IpFlex	C/MATLAB	DAP/DNA FW	C/MATLAB into configuration

Table 3.1.   Language and tools overview for coarse-grained RPUs

same approach. The goal is to generate a dataflow computing block that best matches the inherent parallelism of the part of the application to be implemented as module.

Almost all manufacturers of coarse-grained reconfigurable devices provide proprietary language and tools to implement such modules. It is usually a C-like language with an appropriate scheduler that defines the function of the processing elements as well as their interconnections as function of the time. Table 3.1 gives an overview of the languages and tools used by some manufacturers.

We will not further consider the programming on coarse-grained devices, but rather refer to the corresponding manufacturers for more details on their implementation languages and tools. In the next section, we will briefly present the standard design flow for implementing digital circuits such as ASIC. We will focus in much detail on technology mapping for FPGA because it differs from that of other digital devices.

2.   FPGA Design Flow

The standard implementation methodology for FPGA designs is borrowed from the ASIC design flow. The usual steps are presented in Figure 3.3. Note that those steps concern only the modules of the application that has been identified by the hardware/software co-design process to be executed on the FPGA. The software part is implemented using standard software development approaches, which are well covered in several textbooks.

2.1   Design Entry

The description of the function is made using either a schematic editor, a hardware description language (HDL), or a finite state machine (FSM) editor. A schematic description is made by selecting components from a given library and connecting them together to build the function circuitry. This process has the advantage of providing a visual environment that facilitates a direct mapping of the design functions to selected computing blocks. The final circuit is built in a structural way. However, designs with very large amount of function will not be easy to manage graphically. Instead, a HDL may be used to capture the design either in a structural or in a behavioral way. Besides VHDL
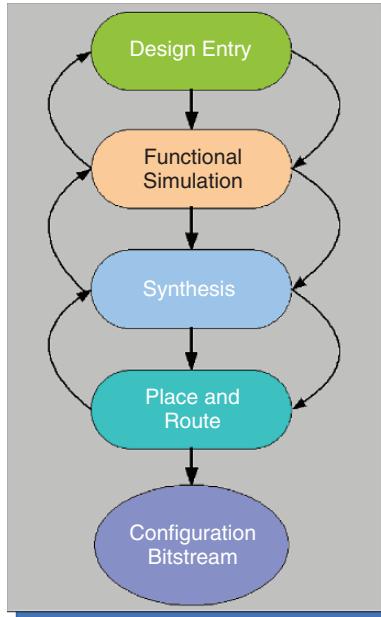
Figure 3.3. The FPGA design flow

and Verilog, which are the most established HDLs several C-like languages, mostly tied to just one compiler manufacturer exist to describe hardware. Here we can cite the languages *Handel-C* [125] and *ImpulseC* [175] and, in some extend, *SystemC* [170].

2.2 Functional Simulation

After the design entry step, the designer can simulate the design to check the correctness of the functionality. This is done by providing test patterns to the inputs of the design and observing the outputs. The simulation is done in software by tools that emulate the behaviour of the components used in the design. During the simulation, the inputs and outputs of the design are usually shown on a graphical interface, which describes the signal evolution in time.

2.3 Logic Synthesis

After the design description and the functional simulation, the design can be compiled and optimized. It is first translated into a set of Boolean equations. Technology mapping is then used to implement the functions with the available modules in function library of the target architecture. In case of FP-GAs, this step is called LUT-based technology mapping, because LUTs are the modules used in the FPGA to implement the boolean operators. The result of the logic synthesis is called the *netlist*. A netlist describes the modules used to

implement the functions as well as their interconnections. There exist different netlist formats to help exchange data between different tools. The most known are the *Electronic Design Interchange Format (EDIF)*. Some FPGA manufacturers provide proprietary formats. This is the case the Xilinx *Netlist Format (XNF)* for the Xilinx FPGAs.

2.4   Place and Route

For the netlist generated in the logic synthesis process, operators (LUTs, Flip-Flopss, Multiplexers, etc.) should be placed on the FPGA and connected together through routing. Those two steps are normally achieved by CAD tools provided by the FPGA vendors. After the placement and routing of a netlist, the CAD tools generate a file called a *bitstream*. A bitstream provides the description of all the bits used to configure the LUTs, the interconnect matrices, the state of the multiplexer and I/O of the FPGA. The full and partial bitstreams can now be stored in a database to be downloaded later according to the paradigm described in section 1.

2.5   Design Tools

The design entry, the functional simulation and the logic synthesis are done using the CAD tools from Xilinx, Synopsys, Synplicity, Cadence, ALTERA and Mentor Graphics. The place and route as well as the generation of configuration data is done by the corresponding vendor tools. Table 3.2 provides some information on the tool capabilities of some vendors.

The logic synthesis step is an important part in FPGA design. It is done by tools, which solve a given number of optimization problems on the path from the design entry to the generation of the configuration data. In the next section, we will take a look inside those tools in order to understand how they work. As the technology mapping process of FPGA differs from that of ASIC, we will pay more attention to this step and present in detail some of technology mapping algorithms, developed for FPGA devices. Each of the algorithms is best adapted for the optimization of a given cost function.

Manufacturer	Tool	Description
Synopsys	FPGA Compiler	Synthesis
Mentor	FPGA	Synthesis, place and route
Synplicity	Sinplify	Synthesis, place and route
Xilinx	ISE	Synthesis, place and route (only Xilinx products)
Altera	Quartus II	Synthesis, place and route (only Altera products)
Actel	Libero	Synthesis, place and route (only Actel products)
Atmel	Figaro	Synthesis, place and route (only Atmel products)

Table 3.2.   Overview of FPGA manufacturers and tool providers

3.   Logic Synthesis

A function, assigned to the hardware in the hardware/software co-design process, can be described as a digital structured system. As shown in Figure 3.4, such a digital structured system consists of a set of combinatorial logic modules (the nodes), memory (the registers), inputs and outputs.

The inputs provide data to the system, whereas the outputs carry data out of the system. Computation is performed in the combinatorial parts and the results might be temporally stored in registers that are placed between the combinatorial blocks. A clock is used to synchronize the transfer of data from register to register via combinatorial parts. The description of a design at this level is usually called *register transfer* description, because of the register to register operation mode previously described.

For such a digital system, the goal of the logic synthesis is to produce an optimal implementation of the system on a given hardware platform. In the case of FPGA, the goal is the generation of configuration data that satisfies a set of given constraints such as the maximal speed, the minimum area, the minimum power consumption, etc. In a structured system, each combinatorial block is a node that can be represented as a two-level function or as multi-level function. Depending on the node representations, the two following synthesis approaches exist:

■ *Two-Level Logic Synthesis*: Two-level synthesis deals with the synthesis of designs represented in two-level logic. Those are representations in which the longest path from input to output, in term of number of gates crossed on the path, is two. Two-level logic is the natural and straightforward approach
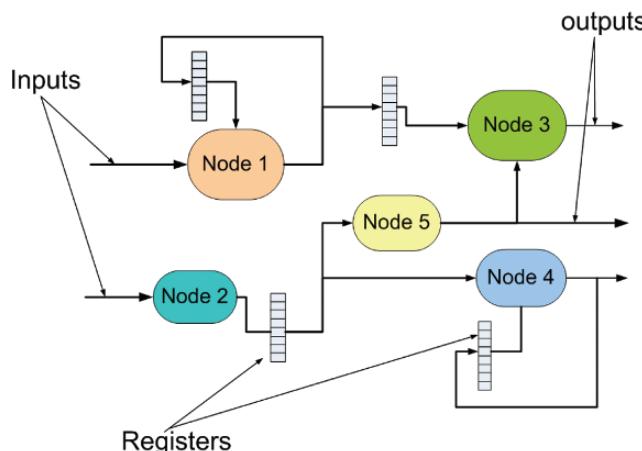
Figure 3.4.   A structured digital system

to implement a Boolean function, because each Boolean function can be
represented as a sum of product terms. In the first level, the products are
built using the AND primitives. The sums of the resulting products are built
in the second level with the OR-primitives.

- *Multi-Level Logic Synthesis*: In the multi-level synthesis, functions are rep-
  resented using a multi-level logic. Those are circuits in which the longest
  path from input to output goes through more than two gates.

Most of the circuits used in practice are implemented using multi-level logic.
Multi-level circuits are smaller, faster in most cases and consume less power
than two-level circuits. Two-level logic is most appropriate for PAL and PLA
implementations, whereas multi-level is used for standard cell, mask-programmable
or field-programmable devices.

We formally represent a node of the structured system as a *Boolean net-
work*, i.e. a network of Boolean operators that reflects the structure and func-
tion of the nodes. A *Boolean network* is defined as a directed acyclic graph
(DAG) in which a node represents an arbitrary Boolean function and an edge
(i, j) represents the data dependency between the two nodes i and j of the
network.

Figure 3.5 shows a Boolean network example for the functions.

3.1   Node representation

The representation of a node is of great importance in the synthesis process.
In two-level logic, the question on how to represent the node of a Boolean net-
work is not relevant, because the final representation is the same sum ofproducts
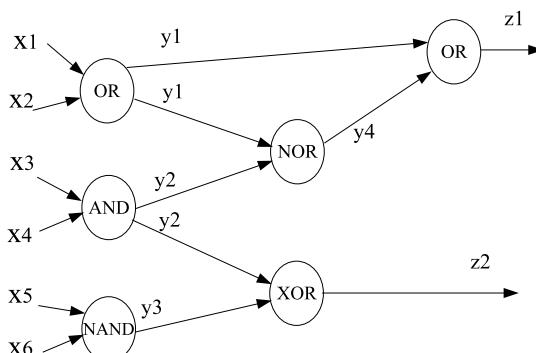
Figure 3.5.   Example of a boolean network with: $y_1 = x_1 + x_2$, $y_2 = x_3 \cdot x_4$, $y_3 = \overline{x_5 \cdot x_6}$,
$y_4 = \overline{y_1 + y_2}$, $z_1 = y_1 + y_4$, and $z_2 = y_2 \oplus y_3$

as the initial representation. Although any valid representation is allowed, in multi-level logic, representation are sought, which efficiently use the memory and for which a correlation with the final representation exists. Furthermore, the representation should be easy to manipulate.

Logic synthesis can be done in two different approaches: the *technology-dependent synthesis* and the *technology independent*. In the first case, only valid gates chosen from the target library are used in the node representation. The final implementation matches the node representation. In the second case, the representation is technology independent, i.e the design is not tied to any library. A final mapping must be done to the final library in order to have an implementation. The technology-independent method is most used, because of the large set of available optimization methods. With a technology-independent representation, synthesis for FPGA devices is done in two steps. In the first step, all the Boolean equations are minimized, independent of the function generators used. In the second step, the technology mapping process maps the parts of the Boolean network to a set of LUTs.

In general, the following choices are made for the representation of a node:

- **Sum of products form:** A Sum of Product (SOP) is the most trivial form to represent a Boolean function. It consists of a sum of product terms, and it is well adapted for two-level logic implementation on PALs and PLAs. Example: $f = x\overline{y}z + \overline{x}yz + wx\overline{y}$.

  This representation has the advantage that it is well understood, and it is easy to manipulate. Many optimization algorithms are available (AND, OR, Tautology, two-level minimizers). The main disadvantage is the non-representativity of the logic complexity. In fact, designs represented as sum of products are not easy to estimate as the complexity of the design decreases through manipulation. Therefore, estimation of progress during logic minimization on SOPs is difficult.

- **Factored form:** A factored form is defined recursively either as a single literal or as a product or a sum of two factored forms: a *product* is either a single literal or the product of two factored forms and a *sum* is either a single literal or the sum of two factored forms. $c\,[\overline{a} + b(d + e)]$ is a product of the factored forms c and $\overline{a} + b(d + e)$, and $\overline{a} + b(d + e)$ is a sum of the factored forms $\overline{a}$ and $b(d + e)$.

  Factored forms are representative of the logic complexity. In many design styles, the implementation of a function corresponds to its factored form. Therefore, factored forms are good estimation of complexity of the logic implementation. Their main disadvantage is the lack of manipulation algorithms. They are usually converted in SOPs before manipulation.

- **Binary decision diagram:** A binary decision diagram (BDD) is a rooted directed acyclic graph used to represent a boolean function. Two kinds of nodes exist in BDDs: *variable* and *constant nodes*.

  - A *variable node* v is a non-terminal having as attribute its argument index[4] $index(v) \in \{1, ..., n\}$ and its two children $low(v)$ and $high(v)$.

  - A *constant node* v is a terminal node with a value $value(v) \in \{0, 1\}$.

  A BDD in which an ordering relation among the nodes exists is called a ordered BDD (OBDD). The non-terminal nodes are ordered from the root to the terminal nodes. Formally, for each non-terminal node v, if $low(v)$ is non terminal, then $index(low(v)) < index(v)$. Similarly, if $high(v)$ is non terminal, then $index(high(v)) < index(v)$.

  The correspondence between a BDD and a Boolean relation is define as follow: A BDD with root v denotes a function f_v.

  - If v is a terminal node, then if $value(v) = 1$, then $f_v = 1$, else $[value(v) = 0]$ $f_v = 0$.

  - If v is a non-terminal node with $index(v) = i$, the Shannon expansion theorem is used to express the function f_v as $f_v = \overline{x_i} f_{low(v)} + x_i f_{high(v)}$, where $f_{low(v)}$ respectively $f_{high(v)}$ denote the function rooted at $low(v)$ respectively $high(v)$. The value of f_v for a given assignment is obtained by traversing the graph from the root terminal according to the assignment values of the nodes.

  A BDD G is a reduced ordered BDD (ROBDD) if the following holds:

  - $low(v) \neq high(v), \forall v$ in G

  - $\forall v, v' \in G$, the subtrees rooted at v and the subtree routed at v' are not isomorphic[5]

  Figure 3.6 shows the optimal BDD representation of the function $f = abc + \overline{b}d + b\overline{c}d$.

  Basically, BDDs are not canonical, i.e. there might exist several BDD representations of the same Boolean function. Reduced ordered BDDs are canonical and compact; thus, they are good replacements of truth tables. For a good ordering, reduced ordered BDDs remain reasonably small

---

[4]An index i defines a variable x_i.

[5]Two BDDs G_1 and G_2 are isomorph $\iff$ there exists a bijective function σ from G_1 in G_2 such that: 1) for a terminal node $v \in G_1$, $\sigma(v) = w$ is a terminal node in G_2 with $value(v) = value(w)$; 2) for a non terminal node $v \in G_1$, $\sigma(v) = w$ is a non-terminal node of G_2 with $index(v) = index(w)$, $\sigma(low(v)) = low(w)$ and $\sigma(high(v)) = high(w)$.
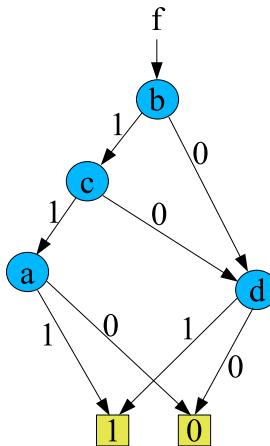
Figure 3.6. BDD-representation of the function $f = abc + \bar{b}d + b\bar{c}d$

for complicated functions. Manipulations of BDDs are well defined and efficient.

3.2 Node manipulation

Having represented the node in one of the formats previously presented optimization algorithms can be applied to the Boolean network. The goal is to generate an optimized network with either a less amount of gates or the lowest depth of the gates. A given number of transformations can be applied to the network for this purpose. Among them, we can cite the following:

- **Decomposition**: The decomposition takes a single Boolean function and replace it with a collection of new expressions. We say that a Boolean function $f(X)$ is *decomposable* if we can find a function $g(X)$ such that $f(X) = f'(g(X), X)$.

  The function $f = ab\bar{c} + a\bar{b}d + \bar{a}c\bar{d} + bc\bar{d}$ for example operates on 12 literals. Using decomposition, f can be written as: $f = a\bar{b}(\bar{c} + d) + (\bar{a} + b)c\bar{d} = a\bar{b}(\bar{c} + d) + \overline{a\bar{b}}(\overline{\bar{c} + d}) = XY + \overline{XY}$ with $X = a\bar{b}$ and $Y = \bar{c} + d$. This decomposition reduces f the operation of f to only 8 literals.

- **Extraction**: The extraction is used to identify common intermediate subfunctions of a set of given functions in order to avoid redundancy.

  Example: $f = (a + bc)d + e$ and $g = (a + bc)\bar{e}$ can be rewritten as $f = xd + e$ and $g = x\bar{e}$ with $x = a + bc$. The output x of the common part of the circuit will be used as common input for f and g.

- **Factoring**: Factoring is the transformation of SOP-expressions in factored form. For example the function $f = ac + ad + bc + bd + e$ can be rewritten as $f = (a + b)(c + d) + e$

- **Substitution**: Substitution replace an expression e within a function f with the value of an equivalent function $g(X) = e$. For example the function $f = (a + bc)(d + e)$ can be rewritten as $f = g(d + e)$, with $g = a + bc$.

- **Collapsing**: Also called elimination, collapsing is the reverse operation of the substitution. Collapsing is used to eliminate levels in order to meet the timing constraints. The function $f = ga + \overline{g}b$ for example will be replaced by $f = ac + ad + b\overline{c}\overline{d}$ with $g = c + d$.

3.3   LUT-based Technology Mapping

The technology-independent optimization phase ends with a reduced Boolean network in which the fanin and fanout of gates vary. The next step consists of allocating LUTs, which are the FPGA library elements for the implementation of the different nodes. Several goals might be followed here. If the goal is to minimize the chip area used by the circuit, then the mapping will try to allocate the less possible amount of LUTs. If the goal is to minimize the delay of the final circuit, then the mapping will try to minimize the depth of the LUTs used. Other goals such as testability and low power might also be followed. Several LUT technology mapping algorithms have been developed in the past. Depending on their optimization goals, those algorithms can be classified in three categories:

- The first category contains the algorithms, whose goal is to minimize the area. This category includes the *Chortle-crf* [87][86], the *MIS-fpga* [163] and the *Xmap* [133]. Because of its popularity and the proof of area optimality for LUTs with less than five inputs, the *Chortle* algorithm will be presented in this section.

- The algorithms in the second category target the delay minimization. Algorithms in this category include the *FlowMap* [53], the *Chortle-d* [86], the *DAG-map* [46] and the *MIS-pga-delay* [164]. The *FlowMap* algorithm was a breakthrough in delay minimization because the authors were able to present an optimal algorithm for LUT technology mapping with delay minimization as well as a proof of the polynomial-time complexity of their algorithm. We present the FlowMap in detail in this section.

- The third category contains algorithms that focuses on maximizing the routability. This category includes the Bhat and Hill [26] work as well as the Schlag [54], Kong and Chang approach [54]. None of those methods will be presented here.

Besides the algorithms presented in this section, a large variety of high-quality LUT technology mapping algorithms have been developed in the last couple of years. Also research on LUT-based technology mapping keeps going on. All those algorithms cannot be presented here. We therefore referred the interested readers to the large amount of available publications. A starting point is the survey provided by Cong et al. [54]. We will first present some definitions needed to better understand the algorithms that we present. We start providing a formal definition of the problem that the algorithms we introduce intend to solve. The LUT technology mapping problem can be stated as follows:

DEFINITION 3.1 [LUT-BASED TECHNOLOGY MAPPING PROBLEM]
Given a Boolean network representing a function f and an integer k. Find an implementation of f using only k-inputs LUTs, such that

1 *The amount of LUTs use is minimal or*

2 *The delay of the resulting circuit is minimal.*

The LUT-based technology mapping is the problem of covering a given graph (the Boolean network) with a set of k-input LUTs. Because the area of the final implementation is defined through the amount of LUT used, the first condition is equivalent to having a covering with a minimal amount of LUTs. The delay in an FPGA is influenced by two main factors: The delay in LUTs and the interconnection delay. Although the delay in LUT is known, the interconnection delay can only be accurately known after the place and route phase. Delay estimation at this stage is done using the depth of LUTs in the design, thus assuming the interconnection delay to be one for each wire. The more LUTs are available on a path from the input to the outputs the higher the delay in the circuit.

DEFINITION 3.2 (PRIMARY INPUT, PRIMARY OUTPUT, NODE LEVEL, NODE DEPTH, FAN-IN, FAN-OUT) *Given a boolean network G, we define the following:*

1 *A **primary input (PI)** node is a node without any predecessor.*

2 *A **primary output (PO)** node is a node without any successor.*

3 *The **level** $l(v)$ of a node v is the length of the longest path from the primary inputs to v.*

4 *The **depth of a network** G is the largest level of a node in G.*

5 *The **fan-in** of a node v is the set of gates whose outputs are inputs of v.*

6 *The **fan-out** of v is the set of gates that use the output of v as input.*

7 *Given a node* $v \in G$ *, input(v) is defined as the set of node of G, which are fan-in of v, i.e. the set of predecessors of v.*

DEFINITION 3.3 (TREES, LEAF-DAG) *1 A **tree** or **fan-out-free circuit** is one in which each node has a maximal fan-out of one.*

2 *A **leaf-DAG** is a combinational circuit in which the only gates with a fan-in greater than one are the primary inputs.*

DEFINITION 3.4 (K-BOUNDED NETWORK) *Given a Boolean network G and a subgraph H of G.*

- *With input (H), we denote the set of all nodes not included in H, which are predecessors of some nodes in H.*

- *G is K-bounded if* $|input(v)| \leq K$ *for all nodes of G.*

A K-bounded Boolean network can be directly mapped to a set of k-inputs LUT by assigning an LUT for each node. However, this straightforward approach may not produce the expected optimal results.

DEFINITION 3.5 (CONE AT A NODE) *Given a Boolean network G,*

1 *A cone* C_v *at a node* v *is the tree with root* v *which spans from* v *to its primary inputs.*

2 *The cone* C_v *is* K-feasible *if:*

- $input(C_v) \leq K$
- *any path connecting two nodes in* C_v *to* v *lies entirely in* C_v

An illustration of K-feasible cone at a node v is given in figure 3.7.

With the previous definition of *K-feasible* cones, the LUT technology mapping becomes the problem of covering the graph with a set of *K-feasible* cones that are allowed to overlap. The technology mapping results in a new DAG in which nodes are *K-feasible* cones and edges represent communication among the cones. Figure 3.8 shows the covering of a graph with 3-feasible cones and the resulting LUT-mapping to 3-input LUTs.

Next, we present some existing LUT-technology mapping algorithms and explain their advantage as well as their drawbacks.

3.3.1 The Chortle Algorithm

The *Chortle* algorithm was developed by Francis et al. [86, 87] at the University of Toronto in 1991 with the aim of minimizing the amount of LUTs in
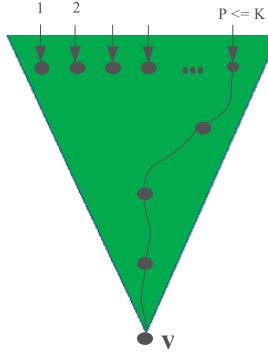
Figure 3.7.   Example of a *K-feasible* cone C_v at a node v
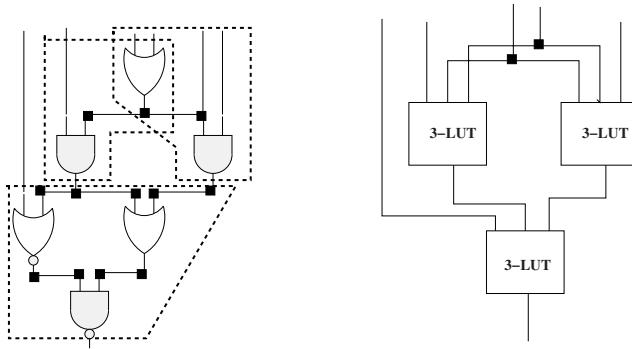
Figure 3.8.   Example of a graph covering with *K-feasible* cone and the corresponding covering with LUTs

the implementation of a given circuit. It operates in two steps: In the first step, the original Boolean network is partitioned into a forest of trees that are then separately mapped into circuits of K-input LUTs. The second step assembles the circuits implementing the trees to produce the final circuit.

The transformation of the original network into a forest is done by partitioning each fan-out node v. Therefore, sub-network rooted at v is duplicated for each input triggered by the fan-out nodes of v. The resulting sub-networks are either trees or leaf-DAGs. The leaf-DAGs are converted in trees by creating a unique instance of a primary input for each of its fan-out edges.

Mapping the trees.   The strategy used by *Chortle* to map a tree is a combination of bin packing and dynamic programming. Each tree is traversed from the primary inputs to the primary outputs. At each node v, a circuit referred to as the *best circuit*, implementing the cone at v extending from the node to the primary inputs of the network, is constructed. The *best circuit* is characterized by

two main factors: The tree rooted at v and represented by a cone must contain the minimum number of LUTs and the output LUT (the root-LUT) implementing v should contain the maximum number of unused input pins. For a primary input p, the *best circuit* is a single LUT whose function is a buffer. Using the dynamic programming, the *best circuit* at a node v can be constructed from its fan-in nodes, because each of them is already optimally implemented. The procedure enforces the use of the minimum number of LUTs at a given node. The *best-circuit* is then constructed from the minimum number of LUTs used to implement its fan-in nodes. The secondary goal is to minimize the number of unused inputs of the circuit rooted at node v.

The construction of the tree is done in two steps. First a *two-level decomposition* of the cone at v is constructed, and then, this decomposition is converted into a *multi-level decomposition*.

Two-level decomposition. The two-level decomposition consists of a single first-level node and several two-level nodes (figure 3.9(a)). Each second level node implements the operation of the node being decomposed over a subset of the fan-in LUTs. The first level node is not implemented at this stage. This will be done in the second phase of the algorithm where the two-level representation is converted into a multi-level implementation.

The two-level decomposition is constructed using a bin packing algorithm approach. In the traditional bin packing algorithm, the goal is to find the minimum number of bins with a given capacity, into which a set of boxes can be packed. In the *Chortle* algorithm, the bins are the second-level LUTs and the boxes are the fan-in LUTs. The capacity of each bin is the number k of LUT inputs. The packing at this stage consist of combining two fan-in LUTs, LUT_1 that realizes the function f_1 and LUT_2 that realizes the function f_2 into a new LUT LUT_r that implements the function $f_1 \oslash f_2$, where $\oslash$ is the operation implemented by the fan-out node. Figure 3.9(a) shows an original graph and its decomposition is shown in 3.9(b). In example, the $\oslash$ is the OR function.

(a) Fan-in LUTs at a node   (b) The two-level decomposition
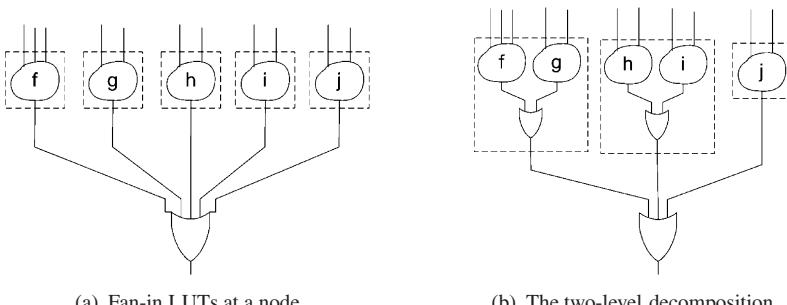
Figure 3.9. Chortle two-level decomposition

Algorithm 3 Chortle's two-level decomposition

　　Start with an empty list of LUTs
　　while there are unpacked fan-in LUTs **do**
　　　　if the largest unpacked fan-in LUT will
　　　　not fit within any LUT in the list **then**
　　　　　　create an empty LUT and add
　　　　　　it to the end of the list
　　　　end if
　　　　pack the largest unpacked fan-in LUT
　　　　into the first LUT it will fit within
　　end while

The pseudocode of the two-level decomposition algorithm is given in algorithm 3.

This algorithm uses a *first-fit-decreasing (FFD)* method, which places a fan-in LUT to be packed into the first LUT it will fit within. However, a *best-fit (BF)* approach that packed the fan-in LUTs into the LUT they best fit in can also be used.

Multi-level Decomposition.　In the second step, the first-level nodes are implemented using a tree of LUTs. The number of LUTs used is minimized by using second-level LUTs that have unused pins to implement a portion of the first-level tree as shown in figure 3.10. The detailed procedure for converting a two-level decomposition into a multi-level decomposition is given in algorithm 4, and figure 3.10 provides an illustration of this process.

The fact that the most filled unconnected LUTs are always selected pushes less filled LUTs to the root of the tree being built. Therefore, the LUTs with the most unused inputs will be found near the root of the tree.

Algorithm 4 Chortle's multi-level decomposition

---
　　while there is more than one unconnected LUT **do**
　　　　if there are no free inputs among
　　　　the remaining unconnected LUTs **then**
　　　　　　create an empty LUT and add
　　　　　　it to the end of the LUT list
　　　　end if
　　　　connect the most filled unconnected LUT
　　　　to the next unconnected LUT with a free input
　　end while
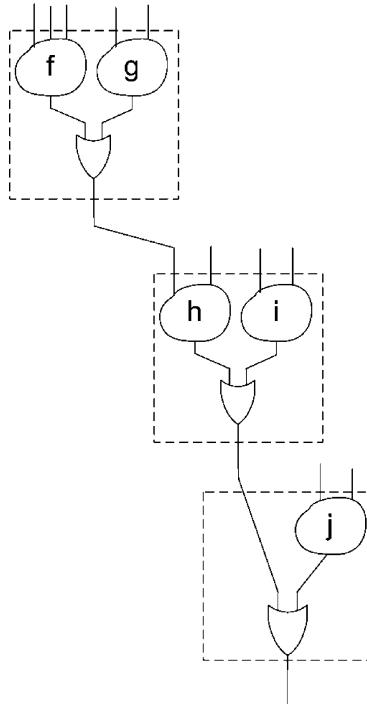
Figure 3.10.   Example of multi-level decomposition

Improvement of the Chortle algorithm.   Before applying the decomposition on the trees in the forest, the *Chortle* algorithm performs a pre-processing step in which De Morgan's Theorem as well as the associativity rules are applied in order to insure that

- the only inverted edges in the trees are those originating from the leaf nodes.

- no consecutive OR and no consecutive AND exist in the trees.

Subject to this, Francis et al. [87] could prove that the Chortle algorithm construct an area optimal algorithm for LUTs with less than or equal five inputs.

Exploiting the reconvergent paths.   A further optimization of the *Chortle* consists of optimizing the reconvergent paths to improve the implementation at a given node. A *reconvergent path* is caused by a leaf node with a fan-out greater than one. This produces two paths in the leaf-DAG that terminate at a given node. The goal of the optimization is to pack the reconvergent paths caused by a given input into just one LUT.
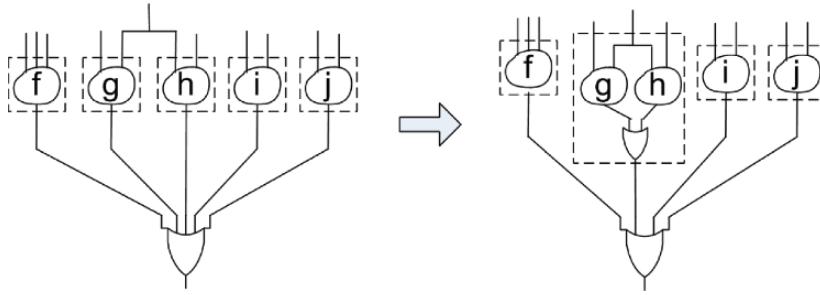
Figure 3.11. Exploiting reconvergent paths to reduce the amount of LUTs used

This strategy is illustrated in figure 3.11. To see are two paths going through the blocks g and h and placed in the same LUT, thus reducing the number of LUTs used from five to four.

If more than one pair of fan-in LUTs share inputs, there will be several pairs of reconvergent paths. To determine which one should be packed in the same LUT, two approaches exist in the *Chortle* algorithm. The first one is an exhaustive approach that first finds all pairs of fan-in LUTs that share inputs. Then every possible combination is constructed by first merging the fan-in LUTs and then proceeding with the FFD bin packing algorithm. The two-level decomposition that produces the fewest bins and the smallest least filled bins is retained as the solution.

A large amount of pairs of fan-in LUTs sharing the same inputs cause the algorithm to be impracticable. To overcome this limitation, a second heuristic called the maximum share decreasing (MSD) can be used. The goal is to maximize the sharing of inputs when fan-in LUTs (boxes) are packed into bins. The MSD iteratively chooses the next box to be packed into bins according to the following criteria: 1) the box has the greatest number of inputs, 2) the box shares the greatest number of inputs with any existing bin and 3) it shares the greatest number of inputs with any of the remaining boxes.

The first criterion insures that the MSD algorithm works like the FFD if no reconvergent path exists. The second and third criteria help to place boxes that share inputs into the same bins. The chosen box is then packed into the bins with which it shares the most input without exceeding the bin capacity, i.e. the number of inputs. If no more bins exist for packing the chosen bin, then a new bin is created, and the chosen box is packed into the new bin.

Logic replication at fan-out LUTs. The replication of the logic at fan-out nodes can also help to reduce the amount of LUTs used. As stated earlier, the *Chortle* technology mapping first decomposes the Boolean network into a forest, then maps each tree separately and finally assembles the final circuit
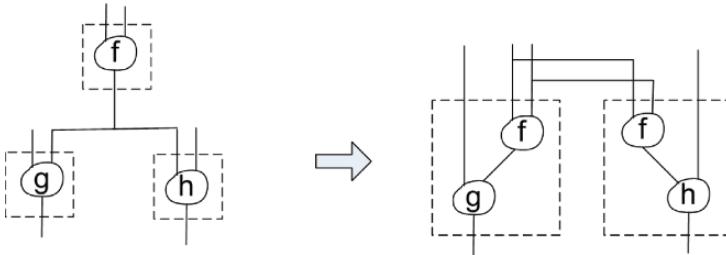
Figure 3.12. Logic replication at fan-out nodes to reduce the number of LUTs used

from the mapped threes. In the assembling phase, logic replication is used to reduce the amount of LUTs in the final circuit as illustrated in figure 3.12.

The chortle algorithm exists in two versions. One with the replication of all nodes and the other without replication. The solution that produces the less amount of LUT is retained.

The *Chortle* algorithm presented in this section is known in the literature [87] as the *Chortle-crf*.[6]

As seen in figure 3.10, the path generated by the *Chorlte* algorithm can become very long if no effort is spent in reducing the delay. This problem is addressed in another version of the *Chrotle* algorithm, the *chortle-d*.[7] Instead of using the bin packing to minimize the amount of LUTs used, the *Chortle-d* focus in the bin packing strategy, on the reduction of the number of levels in the final design. Because we investigate a delay optimal algorithm in the next section, the *Chortle-d* algorithm will not be considered further. The *FlowMap* algorithm that we next present focuses on delay minimization using a network flow approach, a technique that was also use in the MIS-pga algorithm of Murgai et al. [163].

3.3.2 The FlowMap Algorithm

Cong and Ding proposed a polynomial time algorithm for LUT-based technology mapping with the goal of delay minimization [53]. In their algorithm, the *FlowMap* uses the notion of cut in a network to construct an optimal solution in polynomial time.

Because the delay of the final circuit is determined by the delays in the LUTs as well as those of the interconnections, an algorithm seeking to minimize the delay should consider those two factors. However, as the components are not placed and routed on the chip before the technology mapping phase, no real accurate estimation of the interconnection delays can be done. The technology

---

[6]c is for the constructive bin packing, r for the reconvergent path and f for the logic replication.
[7]d stays for delay

mapping with delay minimization as goal can be done only on the basis of the LUT delays. Delay minimization is equivalent to the minimization of the depth of the resulting DAG, i.e. the minimization of the number of LUTs on a path from the primary inputs to the primary outputs of the final circuit.

In contrast to most existing algorithms that first partitioned the Boolean network into a forest of trees and then map each tree separately, the *FlowMap* algorithm can be directly applied to the Boolean network. The precondition is that the network must be K-bounded. If this is not the case, then a preprocessing must be performed on the network to transform it to a K-bounded one. This can be done for example with the same approach used in the *Chortle* to generate the trees.

The *FlowMap* algorithm is a two-step method, which first determines the labels of the nodes in the first phase and then, in the second phase, assembles the nodes in the LUTs according to their level number. We present the two steps in the next paragraphs.

First Phase: Node Labeling. Labeling is based on the notion of cut in a network. We therefore first recall some definitions related to the notion of network and flow in a network.

DEFINITION 3.6 (NETWORK, CUT, LABEL, HEIGHT, VOLUME)

- *Given a network* $N = (V, E)$ *with a source* s *and a sink* t, *a* cut *is a partition* $(X, \overline{X})$ *of the graph* N *such that* $s \in X$ *and* $t \in \overline{X}$).

- *The* cut-size $n(X, \overline{X})$ *of a cut* $(X, \overline{X})$ *is the number of nodes in* X *adjacent to some nodes in* $\overline{X}$

- *A cut* $(X, \overline{X})$ *is K-feasible if* $n(X, \overline{X}) \leq K$

- *The* edge cut-size $e(X, \overline{X})$ *of* $(X, \overline{X})$ *is the sum of the crossing edges capacities.*

- *The label* $l(t)$ *of a node* t *is defined as the* depth *of the LUT, which contains* t *in an optimal mapping of the cone at* t.

- *The* height $h(X, \overline{X})$ *of a cut* $(X, \overline{X})$ *is the maximum label in* X.
  $h(X, \overline{X}) = max\{l(x) : x \in X\}$

- *The* volume $vol(X, \overline{X})$ *of a cut* $(X, \overline{X})$ *is the number of nodes in* X.
  $vol(X, \overline{X}) = |\overline{X}|$

The level of the K-LUT containing the node t in an optimal mapping of N is at least $l(t)$, and the label of all primary outputs of N is the depth of the optimal mapping of N.

The first phase of the algorithm computes the label of the nodes in a topological order, starting with the primary inputs of the graph. The topological order guarantees that each node is processed after all its predecessors have been processed. The labeling is done as follows: Each primary input u is assigned the label $l(u) = 0$. For a given node to be processed at a given level, the subgraph N_t representing the cone at node t is transformed into a network N_t by inserting a source node s which is connected to all inputs of N_t as shown in figure 3.13. For the sake of simplicity, the resulting network is still denoted as N_t

Assuming that t is implemented in the K-LUT $LUT(t)$ in an optimal mapping of N_t, the cut $(X(t), \overline{X(t)})$, where $\overline{X(t)}$ is the set of nodes in $LUT(t)$ and $X(t) = N_t - X(t)$, is a K-feasible cut between s and t and the level of $LUT(t)$ is $l(u) + 1$, where u is the node of $X(t)$ with the maximum level.

With the previous reflection, the K-LUT mapping with minimal delay is reduced to the problem of finding a K-feasible cut with minimum height[8] for each node in the graph. The level $l(t)$ of the node t is therefore given by the following formula:

$$l(t) = min_{\{(X,\overline{X}) \ is \ K-feasible\}}(h(X,\overline{X}) + 1) \qquad (3.1)$$

The following Lemma results from the previous discussion, and therefore, it will be given without proof.
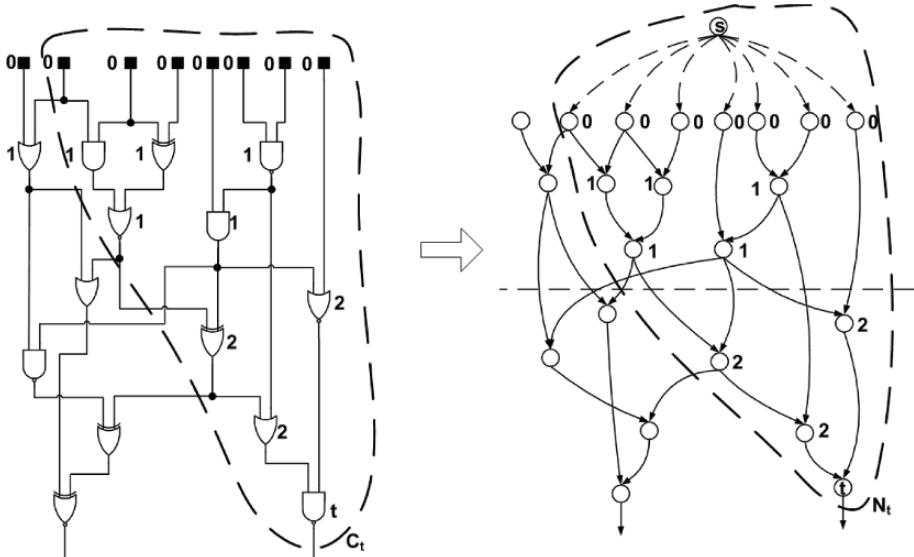
Figure 3.13. Construction of the network N_t from the cone C_t at node t

---

[8]It is assumed here that no cut $(X, \overline{X})$ is computed with primary input nodes in $\overline{X}$.

LEMMA 3.7 *The minimum depth of any mapping solution of N_t is given by:*

$$l(t) = min_{\{(X,\overline{X}) \ is \ K-feasible\}}(h(X,\overline{X}) + 1)$$

Figure 3.14 illustrates the *FlowMap* labelling method. Because there is a minimum height 3-feasible cut in N_t, we have $l(t) = 2$ and the optimal 3-LUT mapping solution for N_t is given in the figure.

The goal of minimizing the delay can be reduced to efficiently compute the minimum height K-feasible cut for each node visited in the graph. The *FlowMap* algorithm constructs a mapping solution with minimum delay in time $O(Km)$, where m is the number of nodes in the network. Further transformations are required on the networks N_t in order to reach this goal. The node labels defined by the FlowMap scheme satisfy the following property.

LEMMA 3.8 *Let $l(t)$ be the label of node t, then $l(t) = p$ or $l(t) = p + 1$, where p is the maximum label of the nodes in $input(t)$.*

Proof : Let t' be any node in $input(t)$. Then for any cut $(X, \overline{X})$ in N_t, either

1 $t' \in X$ or

2 $(X, \overline{X})$ also determines a K-feasible cut $(X', \overline{X'})$ in $N_{t'}$ with
   $h(X', \overline{X'}) \le h(X, \overline{X})$, where $X' = X \cap N_t'$ and $\overline{X'} = \overline{X} \cap N_t'$. Those two cases are illustrated in figure 3.15
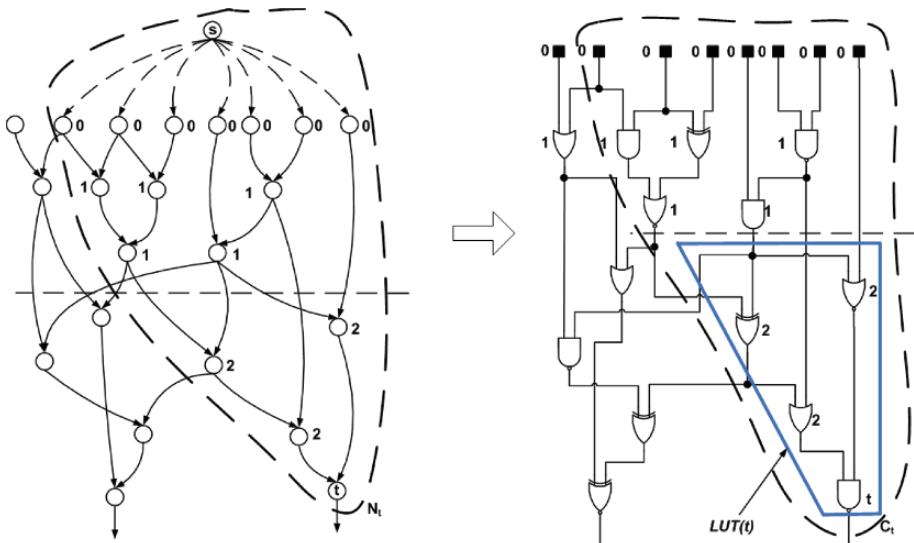
Figure 3.14.   Minimum height 3-feasible cut and node mapping
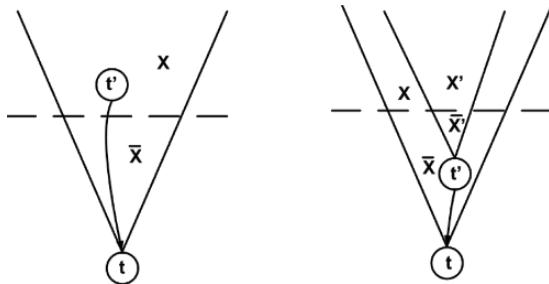
Figure 3.15.   Illustration of the two cases in the proof of Lemma 3.8

In the first case, we have $l(t) = h(X, \overline{X}) + 1 \geq l(t') + 1$ and therefore $l(t) \geq l(t')$

In the second case we have $l(t') - 1 \leq h(X', \overline{X'}) \leq h(X, \overline{X}) = l(t) - 1$, which implies $l(t) \geq l(t')$. Therefore, $l(t) \geq p$. This part of the proof shows that the label of a node cannot be smaller than those of its predecessors.

Because the network N_t is K-bounded, $input(t) \leq K$. Therefore, $(N_t - t, \{t\})$ is a K-feasible cut. As each node in $N_t - \{t\}$ is either in $input(t)$ or is a predecessor of some node in $input(t)$, the maximum label of the nodes in $N_t - \{t\}$ is p. Therefore, $h(N_t - \{t\}, \{t\}) = p$, i.e $l(t) \leq p + 1$.   ∎

According to Lemma 3.8, a delay optimal mapping algorithm could first check whether there is a K-feasible cut $(X, \overline{X})$ of height $p - 1$ in N_t. If such a cut exists, then $l(t)$ is assigned the value p and the node t will be packed in the second phase in a common LUT with the nodes in $\overline{X}$. If such a cut does not exist, then the minimum height of the K-feasible cuts in N_t is p and $N_t - \{t\}, \{t\}$ is such a cut. The value of $l(t)$ is set to $p + 1$ in this case, and a new LUT will be used for t in the second phase of the algorithm. This is the way the *FlowMap* algorithm works.

We now face the next problem, namely finding out if a network has a K-feasible cut with a given height h. In order to provide an answer to this question, further transformations are required. The first one transforms N_t into N'_t as described in Lemma 3.9.

LEMMA 3.9 *Let N'_t be the graph obtained from N_t by applying a transformation which collapses all the nodes in N_t with maximum label $p - 1$ together with t in a new node t'. N_t has a K-feasible cut of height $p - 1$ if N'_t has a K-feasible cut.*

Proof : Let H_t denote the set of node in N_t that are collapsed into t'.
$\Leftarrow$ If N'_t has a K-feasible cut$(X', \overline{X'})$, let $X = X'$ and $\overline{X} = (\overline{X'} - \{t'\} \cup H_t)$, then $(X, \overline{X})$ is a K-feasiblecut of N_t. Because no node in $X'(= X)$ has a

label p or larger, we have $h(X, \overline{X}) \leq p - 1$. Furthermore, according to Lemma 3.8, $l(t) \geq p$, which implies $h(X, \overline{X}) \geq p - 1$. Therefore, $h(X, \overline{X}) = p - 1$.

$\Rightarrow$ If N_t has a cut $(X, \overline{X})$ of height $p - 1$, then X cannot contain any node of label p or higher. Therefore, $H_t \subseteq \overline{X}$, meaning that $(X, (\overline{X} - H_t) \cup \{t'\})$ forms a K-feasible cut of N'_t. ∎

Now that the problem of finding a minimum height K-feasible cut in the network N_t is reduced to the problem of finding a K-feasible cut in N'_t, existing network flow algorithms can be used to compute cuts in N'_t and then check if they are feasible. Unfortunately, for those algorithms, the cuts are defined on crossing edges. Moreover, it is difficult to establish a relation between the number of crossing edges and the number of adjacent nodes in the original network. A second transformation, called *node splitting*, is therefore applied on N'_t. The goal of this second step is to reduce the node cut-size constraint in N'_t to an edge cut-size constraint in the new graph N''_t and then use well-known existing edge-cut computation algorithms.

The *node splitting* transforms the graph N'_t into a graph N''_t as follows:

- For each node v in N'_t other than s and t', two new nodes v_1 and v_2 are introduced and connected by a *bridging edge* (v_1, v_2).

- The source s and sink t' are also introduced in N''_t. For each edge (s, v) respectively (v, t') in N'_t, an edge (s, v_1) respectively (v_2, t') is introduced in N''_t.

- For each edge (u, v) in N'_t with $u \neq s$ and $v \neq t'$, an edge (u_2, v_1) is introduced in N''_t. The capacity of each bridging edge is set to one and those of the non bridging edges is set to infinity.

The constructions of the graphs N'_t and N''_t are illustrated on figures 3.16 and 3.17.

The transformation of the graph N'_t into N''_t insures that if a cut exists in N''_t with capacity less than K, then no edge with infinite capacity will be a crossing one. The only edges that would be crossing the cut are the bridging ones. Because each bridging edge represents a node of N'_t and the capacity of each bridging edge is one, the edge size cut in N''_t is equivalent to the node cut size in N'_t. Based on this observation, the following Lemma can be stated.

LEMMA 3.10 *N'_t has a K-feasible cut if N''_t has a cut whose edge cut size is no more than K.*

The Ford and Fulkerson method [84, 55] can be used to check if a cut with edge cut-size smaller or equal K exists. We first briefly provide some more background on networks, which are important to understand the testing procedure.
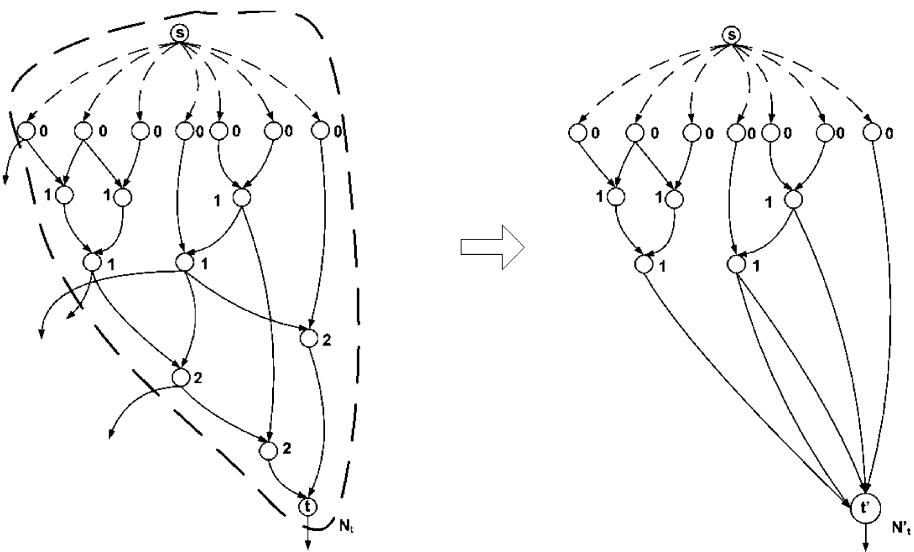
Figure 3.16. Transforming N_t into N'_t by node collapsing

In a network N with source s and sink t, a *flow* can be seen as a streaming of data on different direction on the edges of the network. A node might have data coming in and data going out through its edges, each of which has a given *capacity*. Data streaming in the s-direction (resp. in the t-direction) caused a
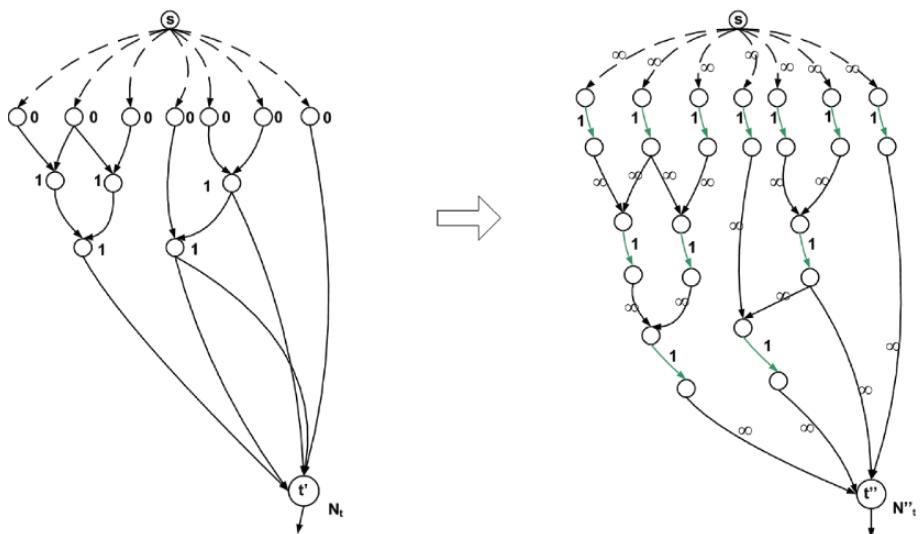
Figure 3.17. Transforming the node cut constraints into the edge cut ones

negative (resp. positive) value of the flow at a given node. The value of the flow in the network is therefore the sum of the flows on the network edges. Formally, a flow f in N can be defined as a function of $N \times N$ in $\mathbb{R}$.

A *residual value* exists on an edge if the flow at that edge has a lower value than the edge capacity. The residual value is then the difference between the capacity and the flow's value. This can be added to the flow in order to saturate the edge. The *capacity of a cut* in the network is defined as the sum of all positive crossing edge capacities. Negative crossing edges do not influence the capacity of a cut. An edge not saturated by the flow is called a *residual edge*. The *residual network* of a network is made upon all the residual edges as well as the nodes that they connect. An *augmenting path* in the residual network is a path from source s to the sink t that contains only residual edges, i.e. a path in the residual network.

A very important relationship between a flow and a cut in a network is given by the following corollary:

COROLLARY 3.11 *The value of any flow in the network is bounded from above by the capacity of any cut in the network.*

This implies the following: *the maximum flow value in the network is bounded from above by the minimum cut of the network.*

Based on the above observation, the notion of cut and that of residual network, the famous *max-flow min-cut theorem* [55] of Ford and Fulkerson states that a flow is maximum in the network if the residual network does not contain any augmenting path. The value of the flow is then equal to the capacity of the minimum cut.

Applying the *max-flow min-cut* theorem to our problem, we can state the following: If a cut with edge cut-size smaller or equal K exists in N_t'', then the maximum value of any flow between s and t'' in N_t'' will be less than K.

Because we are only interested in testing if the value of the cut is smaller than K, the *augmenting path* of Ford an Fulkerson can be applied to compute the maximum flow. The approach starts with a flow f, whose value is set to 0 and then iteratively find some augmenting path P in the residual network and increase the flow on P by the residual capacity $c_f(P)$, that is the value by which the flow on each edge of P can be increased. If no path from s to t exists, then the computing stops and return the maximum flow value.

Because each bridging edge in N_t'' have a capacity of one, each augmenting path in the flow residual graph of N_t'' from s to t'' increases the flow by one. Therefore, the augmenting path can be recursively used to check whether the maximum value for a flow associated to a cut is less than K. For a given cut, if a $K + 1$ augmenting paths could be found, then the maximum flow in N_t'' has a value more than K. Otherwise, the residual graph will not contain a $(K + 1)$th path.

Testing if N_t'' has a cut, whose value is no more than K, can therefore be done through a depth first search starting at s and including in X' all nodes reachable by s. The run-time of the Ford and Fulkerson method is $O(m|f^*|)$, where $|f^*|$ is the value of the maximal flow computed. In the $FlowMap$ algorithm, this value is K, which corresponds to the number of iterations that were performed to find the K augmenting paths. Since finding an augmenting path takes $O(m)$ (m being the number of edges of N_t''), testing if a cut with edge cut-size less or equal K exists can be determined in time $O(Km)$. The resulting cut $(X'', \overline{X''})$ in N_t'' induces a cut $(X', \overline{X'})$ in N_t' which in turn induces a K-feasible cut $(X, \overline{X})$ in N_t.

Because the above procedure must be repeated for each node in the original Boolean network, we conclude that the labelling phase, i.e. computing the label of all edges in the graph, can be done in $O(Kmn)$ where n is the number of nodes and m the number of edges in N.

Node Mapping.   In its second phase, the *FlowMap* algorithm maps the nodes into K-LUTs. The algorithm works on a set L of node outputs to be implemented in the LUTs. Each output will therefore be implemented as a LUT-output.

Initially, L contains all primary outputs. For each node v in L, it is assumed that the minimum K-feasible cut $(X, \overline{X})$ in N_v has been computed in the first phase. A K-LUT LUT_v is then generated to implement the function of v as well as that of all nodes in $\overline{X}$. The inputs of LUT_v are the crossing edges from X to $\overline{X}$ which are less than K, because the cut is K-feasible. L is then updated to be $(L - \{v\}) \cup input(\overline{X})$. Those nodes w belonging to two different cut-set $\overline{X}_u$ and $\overline{X}_v$ will be automatically duplicated. Algorithm 5 provides the pseudocode that summarizes all the steps of the *FlowMap* presented here.

Improvements performed in the *FlowMap* algorithm have the goal of reducing the amount of LUTs used, while keeping the delay minimal. The first improvement is used during the mapping of the nodes into the LUTs. The algorithm tries to find the K-feasible cut with maximum volume. This allows the final LUTs to contain more nodes, thus reducing the area used. The second possibility is the used of the so-called *flow-pack* method, which generalizes the predecessor packing used in the *DAG-Map* [46]. The idea of predecessor packing is to pack K-inputs LUT v in the same K-inputs LUT u, if v is a fan-out free fan-in LUT of u and if the total amount the inputs of u and v is less than K. The *flow-pack* method generalizes the predecessor packing method to the set of predecessors P_u of u (including u) of the node u, provide that P_u has a number of inputs less or equal to K.

Figure 3.18 illustrates the predecessor packing approach. The gate decomposition presented in the *Chortle* algorithm can be used as well for the reduction of the LUT amount.
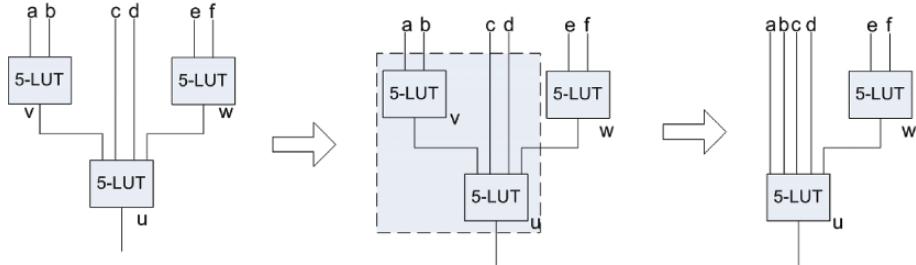
Figure 3.18. Improvement of the FlowMap algorithm through efficient predecessor packing

---

Algorithm 5 Pseudocode of the FlowMap algorithm

/* phase 1: node labeling */
for each PI node v **do**
  $l(v) := 0$
end for
$T :=$ list of non-PI nodes in topological order
while T is not empty **do**
  remove the first node t from T
  construct the network N_t
  let $p = max(l(u)) : u \in input(t)$
  transform N_t into N_t' using collapsing
  transform N_t' into N_t'' using node splitting
  compute a cut $(X'', \overline{X''})$ in N_t'' with $e(X'', \overline{X''}) \leq K$
  using the augmenting path method
  if $(X'', \overline{X''})$ with $e(X'', \overline{X''}) \leq K$ is not found in N_t'' **then**
    $\overline{X_t} := \{t\}$
    $l(t) := p + 1$
  else
    Induce a cut $(X, \overline{X})$ in N_t from the cut $(X'', \overline{X''})$ in $N''t$
    $\overline{X_t} := \overline{X}$
    $l(t) := p$
  end if
end while
/* phase 2: mapping nodes to LUTs */
$L :=$ list of PO nodes
while L contains non-PI nodes **do**
  take a non-PI node v from L
  generate a K-LUT $LUT(v')$ to implement the function of v
  such that $input(v') = input(\overline{X})$
  $L := (L - \{v\}) \cup input(LUT(v'))$
end while

---

4.   Conclusion

In this chapter, the general use of FPGAs in different systems was explained as well as the different steps needed to synthesize the hardware modules that will be executed in the FPGAs at run-time. We have taken a brief tour in logic synthesis with the focus placed on LUT technology mapping, which is the step by which FPGAs differ from other logic devices. Our goal was to present two algorithms that best match the optimization requirements, namely minimal delay and minimal area. Research in FPGA synthesis keep on going and we several LUT-based technology mapping algorithms are expected to be developed in the future. The dynamic development in FPGAs introduces new challenges for the synthesis tools, which has to deal with new elements such as embedded arithmetic modules, embedded memory, different clocking schemes, etc. Therefore, good technology mappers are required to cope with the on-going innovations. Our goal in this chapter was to provide some background to better understand the challenges and provide better solutions for the next generations of FPGAs.

Chapter 4

HIGH-LEVEL SYNTHESIS
FOR RECONFIGURABLE DEVICES

One of the key points for the success of microprocessors is the ease in programming such systems. This is in part due to the maturity of compilers as well as the operation mode of microprocessors, the Von Neumann paradigm that allows any sequential program to be executed on the underlying hardware. In almost three decades of progress in compilers and microprocessors, a huge amount of algorithms have been developed, coded and deployed in high-level languages such as FORTRAN, C, C++, Java. Most of those existing programs can be executed with low modifications and low porting efforts on new platforms. With the very attractive nature of software programming, a very large community of programmers has grown, thus providing software code for the most existing problem. The consequence of this development is that high expectation in programmability is placed on new hardware platforms. A new and highly innovative hardware computing platform, providing the best architectural organization to speed-up applications will certainly fail to be adopted, if its programmability is poor. A less-competitive hardware platform in turn will certainly succeed if the portability of existing algorithms is shown to be easy with a small increase in the computation time. This concept has led to the development of languages and frameworks, which allows for the compiler to be generated for a specified hardware description [168] [187].

Although a certain degree of maturity has been reached in software programming, or better said in sequential programming languages and compilers, parallel programming has not experienced the same advancements. The parallel implementation of a given application requires two main steps. First, the application must be partitioned to identify the dependencies among the different parts and set the parts that can be executed in parallel. This partitioning step requires a good knowledge on the structure of the application.After

partitioning, a mapping phase is required to allocate the independent blocks of the application to the computing hardware resources. For this purpose, a good understanding of the underlying hardware structure is required. As a consequence, the community of programmers is not very enthusiastic whenever parallel programming is concerned.

Reconfigurable computing also faces the disadvantages of parallel computing. To program most of the existing systems, one must write code in a given hardware description language such as VHDL or Verilog and sometimes in a C-like language, which syntax is not so far from VHDL or Verilog. It is therefore not easy to decouple the implementation process from the hardware. The consequence is that FPGA programming is still considered a close field for electrical engineers, despite all the benefits in performance and flexibility that they provided.

High-level synthesis has been introduced for more than a decade as an attempt to describe electronic systems using a high-level language that can be compiled down to the hardware without manual intervention of the user. Having such systems, software developers could be encouraged to develop more applications for reconfigurable systems, and existing applications could be ported on such systems with low effort, thus increasing the popularity of such systems.

High-level synthesis for reconfigurable devices is slightly different from that of other electronic devices in that the resources on which part of the application must be mapped are not fixed. This provides a greater flexibility during the temporal mapping of code segments to computing blocks.

In this chapter, we present the high-level synthesis problem in general and focus on the high-level synthesis for reconfigurable devices, also known as temporal partitioning. Afterwards, we will investigate the algorithms that were developed for this purpose.

1.   Modelling

High-level synthesis (HLS) deals with the specification of a given application at a very high level of abstraction as well as its implementation on a given platform. The starting point is to specify the application in a given high-level language or tool. For this modelling step, several possibilities exist for capturing the behaviour of a systems, from the very simple finite state machines (FSM) and their extensions such as State charts, Control Dataflow Graphs, to very complex tools such as the Petri Nets, each of which has a different level of powerfulness. We first present the dataflow graph that is used as model in this chapter. We also consider two extensions of dataflow graphs, the sequencing graphs and the finite state machine with datapath and explain why those two models are not used in this context.

1.1   Dataflow Graphs

A *dataflow graph (DFG)* provides a means to describe a computing task in a streaming mode. Given a code segment in a high-level language such as C or C++, each operator represents a node in the dataflow graph. The inputs of the nodes are the operand on which the corresponding operator is applied. The output of a node represents the result of the operation on that node. The output of a node can be used as input to other nodes, thus defining a *data dependency* in the graph. Nodes that they depend on others are called *successors* of the nodes on which they depend. Nodes on which other nodes depend are the *predecessors* of the nodes that depend on. Some nodes in the graph have no predecessors and others have no successors. Dataflow graphs might be normalized by inserting in the graph two fake nodes, whose operation has no effect on the dataflow computation. The first fake node is connected as predecessor of all nodes without any predecessor, and the second fake node is connected as successor of all nodes that have no successors.

EXAMPLE 4.1 *As example of dataflow graph, consider the computation of the quadratic root using the formula $x = \frac{(b^2 - 4ac)^{\frac{1}{2}} - b}{2a}$. The corresponding dataflow graph is shown in figure 4.1. Data are streamed through the inputs $a, b, c, d, 4, 2$ and the result is collected at the output of the graph.*

Formally, we define a dataflow graph as follows:

DEFINITION 4.2 (DATAFLOW GRAPH)
Given a set of tasks $T = \{T_1, ..., T_k\}$,

- *a dataflow graph (DFG) is a directed acyclic graph $G = (V, E)$, where $V = T$ is the set of nodes representing operators and E is the set of edges.*

- *An edge $e = (v_i, v_j) \in E$ is defined through the (data)dependency between task T_i and task T_j.*

We assume that for each task, an equivalent hardware implementation exists, which occupies a rectangular area on the chip. Therefore, the nodes as well as the edges in a DFG possess some characteristics such as height, length, area, latency and width that are derived from the hardware resources used later to implement those tasks. Those values are formally defined as follows:

DEFINITION 4.3 (LATENCY, LENGTH, HEIGHT, AREA, WEIGHT OF NODES AND EDGES)
Given a node $v_i \in V$ and its implementation H_{v_i} as rectangular shape module in hardware.

1 *l_i denotes the length and h_i the height of H_{v_i}*

2 *$a_i = l_i \times h_i$ denotes the area H_{v_i}*
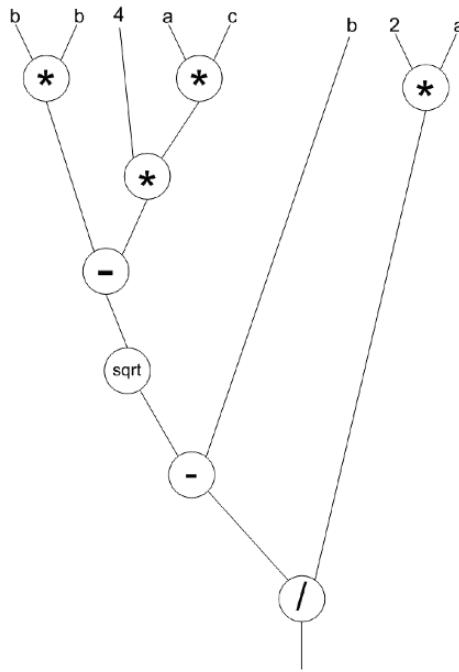
Figure 4.1.   Dataflow Graph for Quadratic Root

3 *The latency t_i of v_i is the time it takes to compute the function of v_i using the module H_{v_i}*

4 *For a given edge $e_{ij} = (v_i, v_j)$, which defines a data dependency between v_i and v_j, we define the weight w_{ij} of e_{ij} as the width of bus connecting two components H_{v_i} and H_{v_j}.*

5 *The latency t_{ij} of e_{ij} is the time needed to transmit data from H_{v_i} to H_{v_j}.*

For sake of simplicity, we will just use the notation v_i to denote a node of the graph as well as its hardware implementation H_{v_i}.

Any program written in a high-level language can be compiled into a dataflow graph, provided that the program is free of loops and branching instructions. This restriction does not match with the reality, as loops and branch instructions are available in most of the programs, for the evaluation of branching conditions at run-time, and to decide on the segment to be executed according to the value of the condition variables. In case of non nested loops, the body of a loop is always a set of instructions that can be represented using a dataflow data structure. Several extensions of dataflow graph exist to capture program with control structures and loop. We will consider two of them in this chapter: the sequencing graphs and the finite state machines with datapath.

1.2   Sequencing Graph

A *sequencing graph* [156] is a hierarchical dataflow graph with two different types of nodes: The *operation nodes* corresponding to normal 'task nodes' in a dataflow graph and the *link nodes* or branching nodes that point to another sequencing graph in a lower level of the hierarchy. Linking nodes evaluate conditional clauses. They are placed at the tail of alternative paths corresponding to possible branches. Loops can be modelled by using a branching as a tail of two paths, one for the exit from the loop and the other for the return to the body of the loop, which is the sub-sequencing graph associated with the link node.

EXAMPLE 4.4 *Figure 4.2 shows an example of sequencing graph with a branching node BR containing two branching paths.*

According to the conditions that node BR evaluates, one of the two sub-sequencing (1 or 2) graphs can be activated. To implement a loop, only one sub-sequencing graph in which the body of the loop is implemented, will be considered. The node BR will then evaluate the exit condition and branch to the next node of the hierarchy level, if the condition holds. Otherwise, the body of the loop is re-entered by reactivating the corresponding sub-sequencing graph.

1.3   Finite State Machine with Datapath

Another extension can be done on a dataflow by integrating a finite state machine in the model to control the execution on a datapath defined by the dataflow graph. The result is the so-called *finite state machine with datapath*
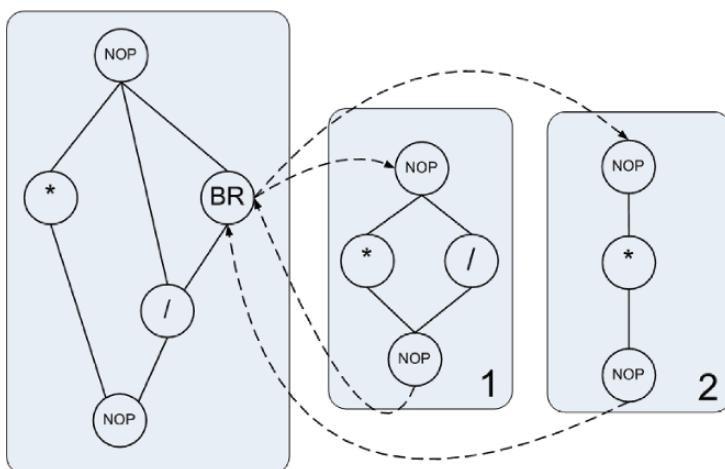
Figure 4.2.   Sequencing graph with a branching node linking to two different sub graphs

(FSMD).[1] We adopt in this section the FSMD definitions and terminology from Vahid et al. [203]. Thereafter, a FSMD can be formally defined as a 7-tuple $< S, I, O, V, F, H, s_0 >$ where:

- $S = \{s_0, s_1 \cdots, s_l\}$ is a set of states,

- $I = \{i_0, i_1 \cdots, i_m\}$ is a set of inputs,

- $O = \{o_0, o_1 \cdots, o_n\}$ is a set of outputs,

- $V = \{v_0, v_1 \cdots, v_n\}$ is a set of variables,

- $F : S \times I \times V \rightarrow S$ is a transition function that maps a tupel (states, input variable, output variable) to a state,

- $H : S \rightarrow O + V$ is an action function that maps the current state to output and variable,

- s_0 is an initial state.

FSMDs have some fundamental differences with traditional finite state machines. First, the transition function operates on arbitrary complex data type like in high-level programming language; second, the transition and action functions may include arithmetic operations rather than just Boolean operations. The arithmetic operations and the complex data types implicitly define a datapath structure in the specification.

The transformation of a program into an FSMD is done by transforming the statements of the program into FSMD states. The statements are first classified into three categories:

- the *assignment statements*: For an assignment statement, a single state is created that executes the assignment action. An arc connecting the so created state with the state corresponding to the next program statement is created.

- the *branch statements*: For a branch statement, a condition state C and a join state J both with no action are created. An arc is added from the conditional state to the first statement of the branch. This branch is labelled with the first branch condition. A second arc, labelled with the complement of the first condition ANDed with the second branch condition is added from the conditional state to the first statement of the branch. This process is repeated until the last branch condition. Each state corresponding to the last statement in a branch is then connected to the join state. The join state

---

[1]In the original definition from Gasky [91], the extension is done on an FSM to support more complex data types and variables as well as complex operators.

is finally connected to the state corresponding to the first statement after the branch.

■ and the *loop statements*: For a loop statement, a *condition state C* and a *join state J*, both with no action are created. An arc, labelled with the loop condition and connecting the conditional state C with the state corresponding to the first statement in the loop body is added to the FSMD. Accordingly, another arc is added from C to the state corresponding to the first statement after the loop body. This arc is labelled with the complement of the loop condition. Finally, an edge is added from the state corresponding to the last statement in the loop to the join state and another edge is added from the join state back to the conditional state.

The transformation steps of a given program to an FSMD is illustrated in figure 4.3.

EXAMPLE 4.5 *Let us model the greatest common divisor (GCD) of two numbers, using an FSMD for as explained [203]. The sequential version of the GCD is given in algorithm 6 and the corresponding FSMD is shown in figure 4.4.*

The loop state (C1) and the branching state within the loop are white and other states are grey. Also, we labelled the states with the action to be taken in those states. An additional label corresponding to the line of the instruction in the program is placed on each state.
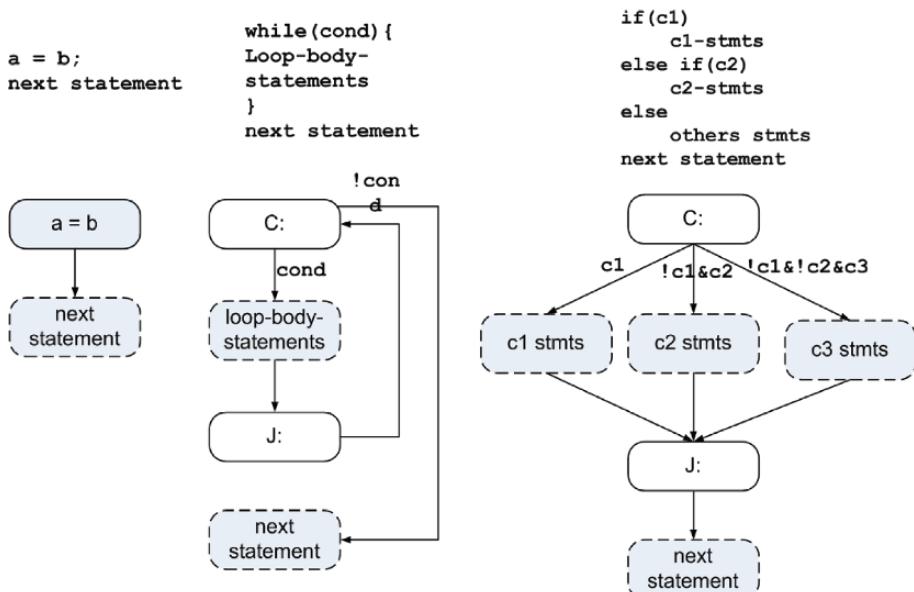
Figure 4.3. Transformation of a sequential program into a FSMD

Algorithm 6 The greatest common divisor sequential algorithm

1: variable a, b, gcd: integer;
2: done := FALSE;
3: **while** (!done) **do**
4:     **if** (a > b) **then**
5:         a := a - b;
6:     **else**
7:         **if** (b > a) **then**
8:             b := b - a;
9:         **else**
10:             done = TRUE;
11:         **end if**
12:     **end if**
13: **end while**
14: gcd := a;
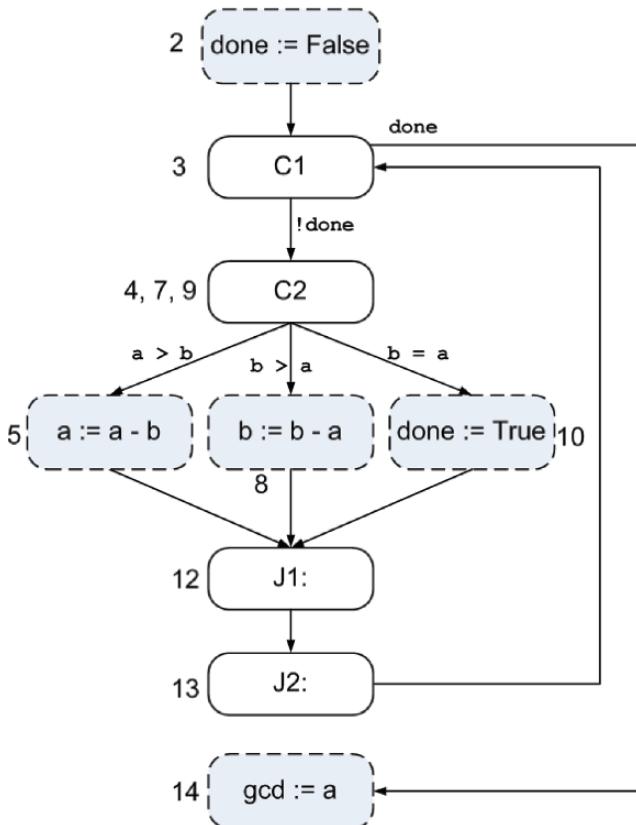
Figure 4.4.   Transformation of the greatest common divisor program into an FSMD

After the program transformation into an FSMD, a datapath must be created that will be controlled by a finite state machine derived from the FSMD. The process of creating the datapath is straightforward. First, a register must be instantiated for each variable in the program. An output port implicitly declares a variable for which a register must be created. In the second step, a functional unit will be created for each arithmetic operation in the state diagram. The third step consists of connecting the ports of the functional unit with those of the variable. In the fourth step, a unique identifier is created for each control input and output of the functional units in the datapath. Sharing the operator can be done using multiplexers.

Figure 4.5 shows the datapath of the GCD programmed together with control finite state machine resulting from the FSMD. The control signals LDA and LDB are to load the two registers A and B with the values coming from the subtractor, the SELL and SELR to select the corresponding value from the multiplexer. Those signals are controlled by the FSM that set them according to its current state. The status signals AGTB and ALTB are used by the FSM to decide about the next state to move to, depending on the value of the comparison between A and B.

Having specified a system using one of the modelling tools previously presented, the next step will consist of the compilation of the specification into a
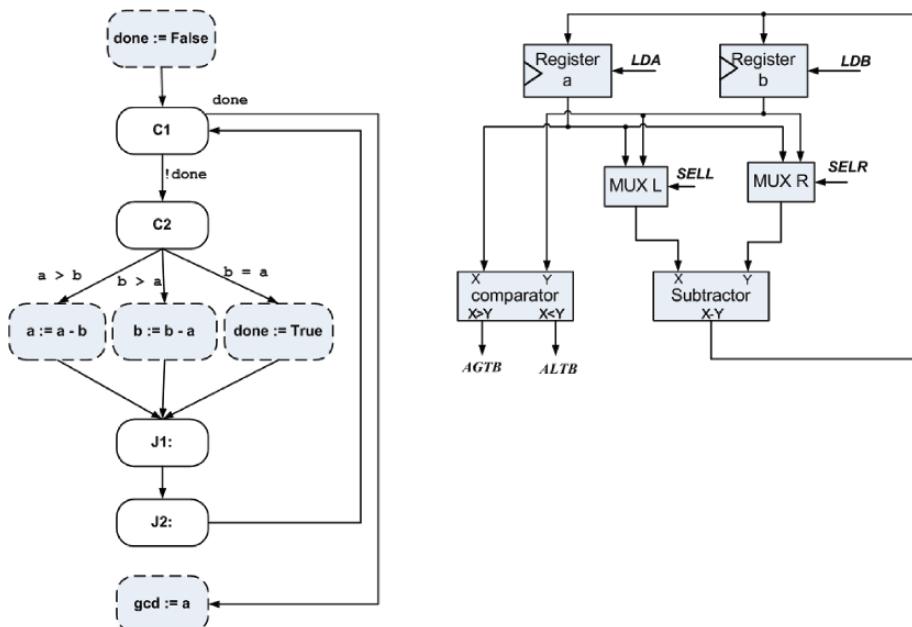
Figure 4.5. The datapath and the corresponding FSM for the GCD-FSMD

set of hardware components. This compilation step, also known as synthesis is usually done in three different steps. The first one is the *allocation*, which defines the type of resources required by the design, and for each type the number of instances. In the GCD case, the types of resources needed are comparators, subtractors, registers and multiplexers. Two registers are instantiated and two multiplexers are instantiated whereas only one subtractor and one comparator are used. Instead of using just one subtractor and two multiplexers, we could choose to use two subtractors and no multiplexers. The next step after the allocation is the *binding*. In this step, each operation is mapped to an instance of a given resource. As many operators can be mapped to the same instance of a resource, a *schedule* is used in the third step to decide on which operator should be assigned a given resource at a given period of time. Formally, allocation binding and scheduling can be defined as follow:

DEFINITION 4.6 (ALLOCATION) *For a given specification with a set of operators or task $T = \{t_1, t_2, \cdots, t_n\}$ to be implemented on a set of resource types $R = \{r_1, r_2, \cdots, r_t\}$. An Allocation is a function $\alpha : R \rightarrow \mathbb{Z}^+$, where $\alpha(r) = z_i$ denotes the number of available instances of resource type r_i*

DEFINITION 4.7 (BINDING) *For a given specification with a set of operators or task $T = \{t_1, t_2, \cdots, t_n\}$ to be implemented on a set of resource types $R = \{r_1, r_2, \cdots, r_t\}$. A binding is a function $\beta : T \rightarrow R \times \mathbb{Z}^+$, where $\beta(t_i) = (r_i, b_i)$, $(1 \leq b_i \leq \alpha(r_i))$ denotes the instance of the resource type r_i on which t_i is mapped to.*

DEFINITION 4.8 (SCHEDULE) *For a given specification with a set of operators $T = \{t_1, t_2, \cdots, t_n\}$, a schedule is a function $\varsigma : V \rightarrow \mathbb{Z}^+$, where $\varsigma(t_i)$ denotes the starting time of the tast t_i.*

1.4   Fundamental differences in HLS for reconfigurable computing

The high-level synthesis for reconfigurable computing differs from that of other architectures[2] in two main points. First, the fundamental problems that must be solved in common high-level synthesis are the binding and scheduling. The allocation is usually not necessary, in particular in those systems in which the resource types and their numbers are already fixed. The binding will just map operator to resources and the scheduling will decide which operator owns the resource at a given time. This is not the case in reconfigurable computing, where the architectural resources are created on the reconfigurable device according to the resource types needed to map the operators at a given time. As

---

[2]Also known as architectural synthesis.

long as basic computing resources are available on the device, new resources can be built. With the uniformity of resources available on a reconfigurable device, it is possible to implement any task on a given part of a reconfigurable device, provided that the available resource are enough. The problem of binding an operator to a given resource then becomes an area assignment problem.

EXAMPLE 4.9 *Let us illustrate this major difference with using dataflow graph for the computation of the functions $x = ((a \times b) - (c \times d)) + ((c \times d) - (e - f))$ and $y = ((c \times d) - (e - f)) - ((e - f) + (g - h))$ as shown on figure 4.6.*

We consider the implementation on a common architecture. Assume an allocation that selects one instance of each resource type (multiplier, adder and subtractor). Consider that the multiplication needs 100 basic resource units. This can be 100 NAND-gate for example, but we express the basic unit of computation in look-up table (LUT). The multiplier therefore needs 100 LUTs, the adder and the subtractor need 50 LUTs each. Because only one instance for each resource type is available, the two subtractions in the first level of the graph corresponding to the nodes 3 and 4 cannot be executed in the same step, although enough basic resources are available. But, those resources were used to implement one adder and one subtractor instead of two subtractors. The adder cannot be used in the first step, because it depends on a subtractor that
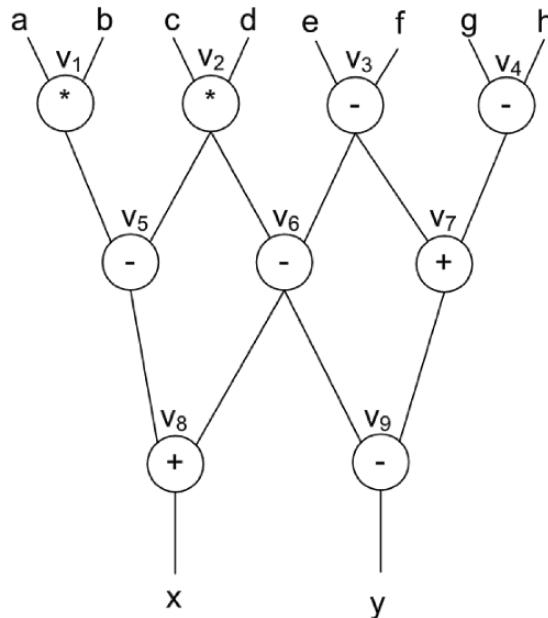
Figure 4.6. Dataflow graph of the functions: $x = ((a \times b) - (c \times d)) + ((c \times d) - (e - f))$ and $y = ((c \times d) - (e - f)) - ((e - f) + (g - h))$
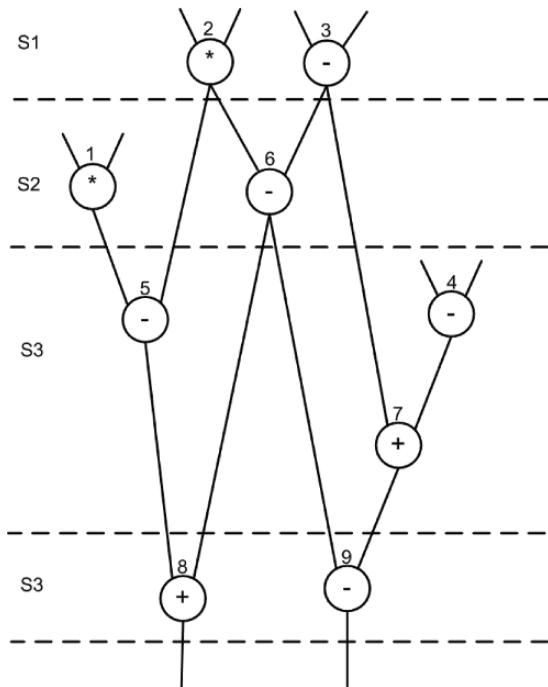
Figure 4.7.   HLS of the graph in figure 4.6 on a an architecture with one instances of the resource types $+, *$ and $-$

must be first executed. The minimum execution delay is four steps if we use chaining[3] in the schedule (figure 4.7).

Consider now the implementation on an reconfigurable device having the same amount of basic resources, namely 200 LUTs. Because the basic resources are not bounded to any operator, they can be configured according to our need. We are therefore able to map one multiplier and the two subtractors in the first step. In the second step, we map one multiplier, one adder and one subtractor. In the third step, the rest of operators are map onto the basic resources. The resulting implementation is shown on figure 4.8. This implementation is one step faster than the previous one, although they use the same amount of resource.

The second major difference concerns the control of the computation steps of a given application. In general high-level synthesis, the application is specified using a structure that encapsulates a datapath (set of computational

---

[3]Chaining operations means sequentially executing a set of operations in the same time slot, provided that their overall delay is less that the clock delay.
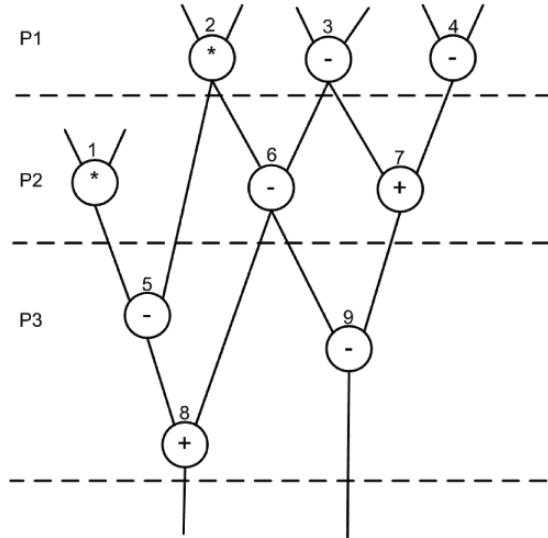
Figure 4.8. HLS of the graph in figure 4.6 on a reconfigurable device

resources and their interconnections) and a control part. The synthesis process then allocates the resources to operators at different time according to a computed schedule. This temporal resource assignment is controlled by a separate part of the system for which a synthesis is required. In reconfigurable devices, a set of hardware modules implemented as datapath normally compete for execution on the chip. Instead of a separate control module to set the control lines of the datapath modules, a processor is used to control the selection process of the hardware modules by means of reconfiguration. The same processor is also in charge of activating the single resources in the corresponding hardware accelerators. For the modelling of the datapath, a dataflow graph is usually sufficient, as loops and branch control is left to the processor. Although the general high-level synthesis has to control each single operator, the high-level synthesis for reconfigurable devices deals with a set of operators with different execution delay at a time. This means that for a given application, the resources on the device are not allocated to only one operator but to a set of operators that must be placed at the same time and removed at the same time. With this, an application must be partitioned in sets of operators. The partitions will then be successively implemented at different time on the device. This process, called *temporal partitioning*, allows an application to be sequentially computed, by allowing a temporal sharing of resource among different sets of operators. We next present the temporal partitioning problem and some of the solution approaches developed to solve it.

1.5   Temporal Partitioning

We assume that for each task $t_i \in T = \{t_1, t_2, \cdots, t_n\}$ a core[4] C_i is available in a given library. This can be a simple description of the task t_i in a hardware description language or the final implementation of the task, i.e. the final partial bitstream for the given task. Recall that the implementation of a given task for a given device describes all the resources used to implement the given task in the device.

In temporal partitioning, complete partitions are downloaded onto the device rather than just a single component. Whenever a new partition is downloaded, it is assigned all the resources onto the device and the partition that was running is destroyed. Two possibilities exist to build the implementation of a given partition. The first one is to compose the partition from the implementations of the single components in the partition. This exercise requires some knowledge about the structure of the bitstream to place the component at the correct locations and perform the right interconnections. The second possibility is to instantiate the HDL description of each single component in the HDL description of the partition and to let the compiler do the rest of the job. Preserving the boundary of each component is usually not required. Therefore, the complete design is usually flattened and optimized across the component boundaries before the technology mapping, the placement and routing phases.

Each task is characterized by its latency, the height, the length and area of its bounding box. Those values are estimated values on the basis of the implementation of the task in the region defined by its bounding box. It may not be the same in the final implementation if the complete partition design is first flattened before optimization. The flattening of a design before optimization provides more room for optimization. The resulting design is usually smaller than the non-flattened. However, placement algorithms tend to place components belonging to the same group in the same region. If some modules in a given component are tied to other modules in another component, then the placer will tend to place tied components in the same group. This may result in an increase in the latency of the two involved components. We next provide some definitions before stating the temporal partitioning problem.

DEFINITION 4.10 (CONFIGURATION) *Given a reconfigurable processing unit H and a set of tasks $T = \{t_1, ..., t_n\}$ available as cores $C = \{c_1, ..., c_n\}$, we define the configuration ζ_i of the RPU at time t_i to be the set cores $\{c_{i1}, ..., c_{ik}\} \subseteq C$ running in H at time t_i. We set $\zeta_i = \{c_{i1}, ..., c_{ik}\}$*

We consider a model in which the execution control is left to the processor that is also in charge of controlling the reconfiguration process on the chip.

---

[4]Hardware module.

We therefore use the dataflow model as entry point for the part of the design to be implemented in the reconfigurable device. Because the control loops are no more available in the design, cycles can be only available in the dataflow graph, if the outputs of some components are used as input to some of their predecessors. This is for instance the case with the GCD-datapath in figure 4.5, where the output of the subtractor is used as inputs for the two registers A and B. In this case, the part of the datapath containing the loop is encapsulated into a single component that will be executed for a given number of clock cycles fixed by the processor. We can also imagine a coarse-grained structure in which a smaller controller supervises the execution process until completion. Upon completion of the execution, status signals are used to notify the processor. If a coarse-grained element containing loops fits into the device, then it may be entirely placed into a partition with other component. However, we may also face the situation, where a coarse-grained element containing a loop is too large to fit onto the device. In this case, we will partition only this coarse-grained element in parts that will be sequentially implemented in the chip. The number of iterations will then be done in the partitioned version as required in the original version. This approach is illustrated in figure 4.9, where the coarse-grained node C with a loop in its implementation is partitioned in two part P_1 and P_2. P_1 will be downloaded before P_2 in one loop iteration.

The rest of the chapter is based on the assumption that an underlying dataflow graph model is available. We therefore redefine the schedule on the basis of the dataflow graph.

DEFINITION 4.11 (SCHEDULE AND ORDERING RELATION) *Given a DFG $G = (V = \{v_1, v_2, \cdots, v_n\}, E)$,*

- *A schedule is a function $\varsigma : V \rightarrow \mathbb{Z}^+$, where $\varsigma(v_i)$ denotes the starting time of the node v_i that implements a task t_i.*
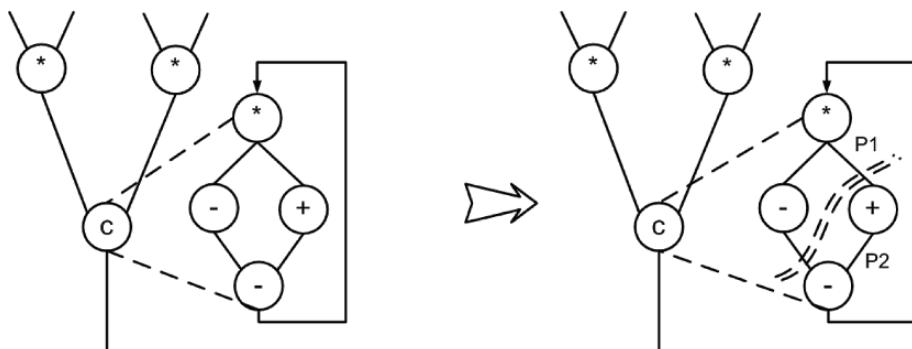
Figure 4.9. Partitioning of a coarse-grained node in the dataflow graph.

A schedule ς is feasible if: $\forall e_{ij} = (v_i, v_j) \in E$, such that e_{ij} defines a data dependency between tasks t_i and t_j, $\varsigma(t_j) \geq \varsigma(t_i) = t_i + t_{ij}$ (t_{ij} denotes the latency of the edge e_{ij} and t_i is the time it takes the node v_i to complete execution).

- *We define an ordering relation $\leq$ among the nodes of G, such that $v_i \leq v_j \iff \forall$ schedule ς, $\varsigma(v_i) \leq \varsigma(v_j)$.*

    $\leq$ is a partial ordering, as it is not defined for all pairs of nodes in G.

DEFINITION 4.12 (PARTITION) *A partition P of the graph $G = (V, E)$ is its division into some disjoint subsets $P_1, ..., P_m$ such that $\bigcup_{k=1}^{m} P_k = V$*

A partition is feasible in accordance to a reconfigurable device H with area $a(H)$ and pin count $p(H)$ if:

- $\forall P_k \subseteq P$: $a(P_k) = (\sum_{v_i \in P_k} a_i) \leq a(H)$
- $1/2(\sum_{\{e_{ij} \in E: (e_{ij} \cap P_k) \neq \emptyset \text{ and } (e_{ij} - P_k) \neq \emptyset\}} (w_{ij})) \leq p(H)$

The *run-time $t(P_i)$* of a partition P_i is the time it takes to complete the execution of that partition. It is the maximum time from the input of the data to the output of the result. The definition of the run-time is limited to only one execution step of the complete partition. If a partition is use in a loop, then we must multiply the run-time of the partition by the number of iterations.

Each partitioning is submitted to the restriction that the total amount of resources assigned to it, i.e. the sum of the resources of all its components does not exceed the available amount of resources. The total area of a partition must therefore be smaller than the device area and the weighted sum of all crossing edges[5] is less than the number of terminals or pins. This restriction on the crossing edges can be relaxed by multiplexing the use of input output pins in order to allow a design with a greater amount of pins to communicate with the external world.

DEFINITION 4.13 (ORDERED PARTITIONS) *We can extend the ordering relation $\leq$ to partitions as follows: $P_i \leq P_j \iff \forall v_i \in P_i, \forall v_j \in P_j$, either $v_i \leq v_j$ or v_i and v_j are not in relation. Thereafter, a partitioning P is ordered $\iff$ an ordering relation $\leq$ exists on P.*

An ordered partitioning is characterized by the fact that for a pair of partitions one can always be implemented after the other with respect to any scheduling relation.

---

[5]A crossing edge is an edge that connects one component in a partition with another component out of the partition

DEFINITION 4.14 (TEMPORAL PARTITIONING) *Given a dataflow graph $G = (V, E)$ and a reconfigurable device H, a temporal partitioning of G on H is an ordered partitioning P of G with respect to the reconfigurable device H.*

Applying a temporal partitioning on a dataflow graph generates another graph that we call *configuration graph*. With the necessary control part of an application, a final schedule must be applied on the configuration graph to compute the original application. Formally, we define a configuration graph as follows:

DEFINITION 4.15 (CONFIGURATION GRAPH) *Given a DFG $G = (V, E)$ and a temporal partitioning $P = \{P_1, ..., P_n\}$ of G, we define a Configuration graph of G relative to the partition P, with notation $\Gamma(G/P)$ to be the graph $\Gamma(G/P) = (P, E^P)$ in which the nodes are partitions in P. An edge $e^P = (P_i, P_j) \in E^P \iff \exists e = (v_i, v_j) \in E$ with $v_i \in P_i$ and $v_j \in P_j$.*

For a given partition P, each node $P_i \in P$ has an associated configuration ζ_i that is the implementation of P_i for the given device H. Because two partitions are never running at the same time, communication data between the partitions must be stored in a memory external to the device. We call this memory that is used for communication between partitions the *communication memory*. Physically, the communication memory might not be on a different chip than the main system memory. It can be a reserved part of the main memory.

The computation steps for two partitions P_i and P_j such that $P_i \leq P_j$ is done as follows: The corresponding configuration for P_i is first downloaded into the device. P_i copies all the data it needs to send to other partitions into the communication memory. Upon completion, the device is reconfigured to implement the partition P_j, which can then access the communication memory and collect the data that were sent to it. By *interconfiguration registers*, we denote those registers that are usually used at the boundary of the device to hold the input and output values. Interconfiguration registers are usually mapped into a processor address space that can be used by the processor for communication with the device. An example of configuration graph with interconfiguration registers is given in figure 4.10.

According to the goal to be reached, several objective functions can be defined for the temporal partitioning problem. One objective could be the minimization of the number of partitions to reduce the overall reconfiguration overhead. Another objective could be the minimization of the computation time. This can be expressed for example through the minimization of the maximum computational delay across all the partitions. A third objective could be the minimization of the overall amount of wasted resources on the chip. One of the major drawbacks of temporal partitioning is the non-efficient use of the device area. Temporal partitioning normally targets non-partial reconfiguration devices. It has the advantage that the resulting partition can be
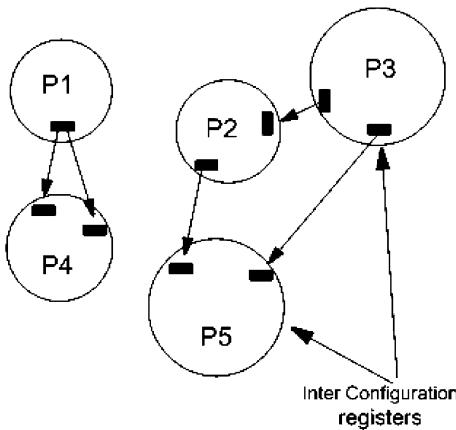
Figure 4.10.   Example of configuration graph

implemented in just one circuit. Therefore, floorplanning efforts are left to the synthesis tools. However, when components with shorter run-times are placed in the same partition with other components with longer run-time, those with the shorter components remain idle for a longer period of time, resulting in a waste of the device resources. The wasted resource of a given partition can be formally defined as follows:

DEFINITION 4.16 (WASTED RESOURCE) *Given a dataflow graph* $G = (V, E)$ *and a temporal partitioning* $P = \{P_1, ..., P_k\}$ *of* G:

■ *The wasted resource* $wr(v_i)$ *of a node* v_i *is the unused area occupied by the node* v_i *during the computation of a partition. It is the area occupied by the node times the idle time of the node.*
  $wr(v_i) = (t(P_i) - t_i) \times a_i$, *where* $t(P_i)$ *is the run-time of the partition. The idle time* $t(P_i) - t_i$ *of* v_i *is the run-time of the partition minus the running time of the component.*

■ *The wasted resource* $wr(P_i)$ *of a partition* $P_i = \{v_{i_1}, .., v_{i_n}\}$ *is defined as the sum of wasted resources of its components:*
  $wr(P_i) = \sum_{j=1}^{n} (t(P_i) - (t_{i_j})) \times a_j$

■ *The wasted resource of a partitioning* P *is the sum of the wasted resource of all its partitions.*
  $wr(P) = \sum_{i=1}^{k} wr(P_i)$

Figure 4.11 graphically illustrates the waste resource of a partition. The run-time of the partition is determined by the component v_1. The shaded area defines the overall wasted resource of the partition.
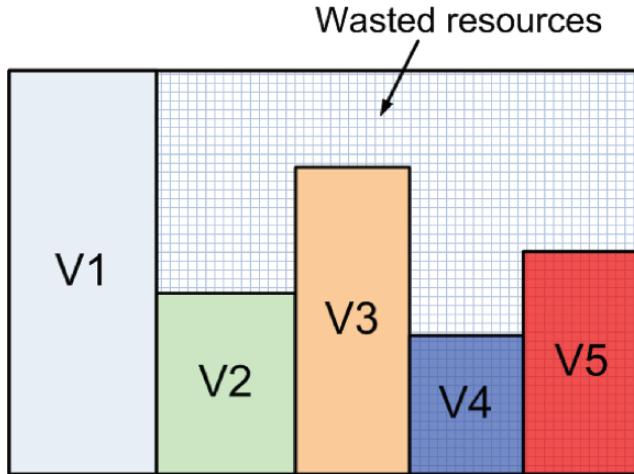
Figure 4.11. Wasted resources

The minimization of the overall run-time is difficult to capture in an equation, as it can be affected by many factors. The reduction of the number of partition automatically reduces the overall configuration time, but not necessarily the overall computation time. If the configuration time is smaller than the computation time of some components, an additional packing of those components in a partition will reduce the number of partition on one side. However, the run-time of the partitions in which the components were packed will increase by a factor greater than the reconfiguration time. Fortunately, this is not the case with the existing technology, where the reconfiguration time is usually higher than the computation time of single components.

The execution time does not only depend on the partition but also on the processor used for reconfiguration and the speed of data exchange. Recall that fully reconfigurable devices are usually coupled with a processor, which manages the complete system. The processor reconfigures the device and manages the data exchange. One of the most important goal to be considered in temporal partitioning is the communication overhead. This goal can be captured in several ways. We present here a simple model to capturing the communication cost during temporal partitioning: the *graph connectivity*.

DEFINITION 4.17 (GRAPH CONNECTIVITY) *Given a dataflow graph $G = (V, E)$, we define the connectivity, $con(G) = \frac{2 \times |E|}{|V|^2 - |V|}$, of G as the rapport between the number of edges in E over the number of all edges that can be built with the nodes of G.*

For a given subset V' of V, the connectivity of V' is defined as the relation between the number of edges connecting the nodes of V' over the set of all edges that can be built with the nodes of V'.

The connectivity of a set provides a means to measure how strongly the components of a set are connected. High connectivity means a strongly connected set, whereas low connectivity reflects a graph in which many modules are not connected together. The connectivity may be used to define how good a partitioning algorithm, whose goal is the minimization of the communication cost, has performed. We formally define the quality of a partitioning as follows:

DEFINITION 4.18 (QUALITY) *Given a dataflow graph* $G = (V, E)$ *and a partitioning* $P = \{P_1, ..., P_n\}$ *of G, we define the quality* $Q(P) = \frac{1}{n} \times \sum_{i=1}^{n} (con(P_i))$ *of P as the average connectivity over all the partitions* $P_i, 1 \leq i \leq n$.

Most of the target architectures for which temporal partitioning is used are made upon a reconfigurable device connected to a host processor by a bus such as, for instance, the PCI.[6] The number of bus lines is limited. Therefore, the communication has to be time-multiplexed on the bus, if many data have to be transported on the bus. This happens for example when a partition has to be replaced. Recall that the communication between the partitions is done by a set of registers inside the reconfigurable devices. All the temporary data in those registers have to be saved in the communication memory before reconfiguration. The device is then reconfigured, and the data are copied back into the reconfigurable device registers. Minimizing the communication overhead can be done by minimizing the weighted sum of crossing edges among the partitions. This will also minimize the set of registers needed to communicate between the generated partitions, thus reducing on one side the size of the communication memory and the communication time on the other side. This goal is likely to be reached if highly connected components are placed in the same partition.

After a partitioning by a given algorithm, the quality of the partition will determine whether the algorithm performed well. If a graph is highly connected and the partitioning algorithm performs with low quality, then there will be more edges connecting different partitions and therefore more data exchange among the partitions. But if the graph is highly connected and the partitioning algorithm performs with high quality, then the components in the partitions are highly connected, and therefore, there will be fewer edges connecting the partitions.

---

[6]Peripheral Component Interconnect.
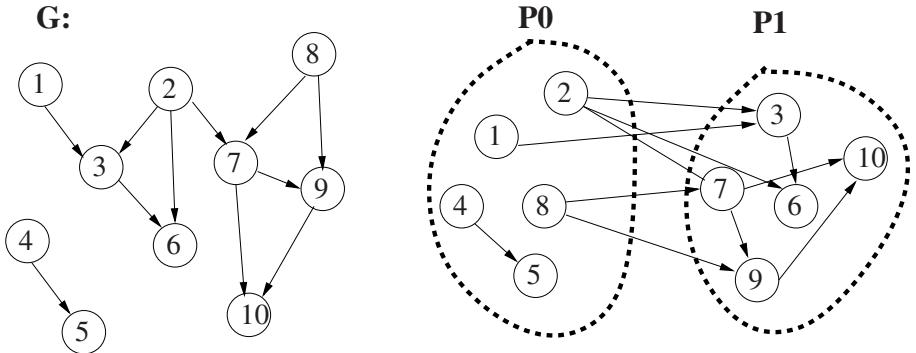
Figure 4.12. Partitioning of the graph G with connectivity 0.24 with an algorithm that produces a quality of 0.25

EXAMPLE 4.19 *Figures 4.12 and 4.13 illustrate the connectivity of a graph and the quality of the algorithm that was used for the partitioning. In figure 4.12, a graph with a connectivity of 0.24 is partitioned by a first algorithm, which produces a quality of 0.25. The same graph is partitioned by another algorithm in figure 4.13 with a quality of 0.45. In the first case, we have six edges connecting the two partitions, while there are only two edges connecting the partitions in the second case. The second case is, therefore, better than the first case for data communication between the partitions.*

In the next section, we present contributions of authors who developed various techniques and methodologies for temporal partitioning. The methods can be grouped into four different categories. In the first category are the *list-scheduling based methods*. The second category encounters *exact methods*,
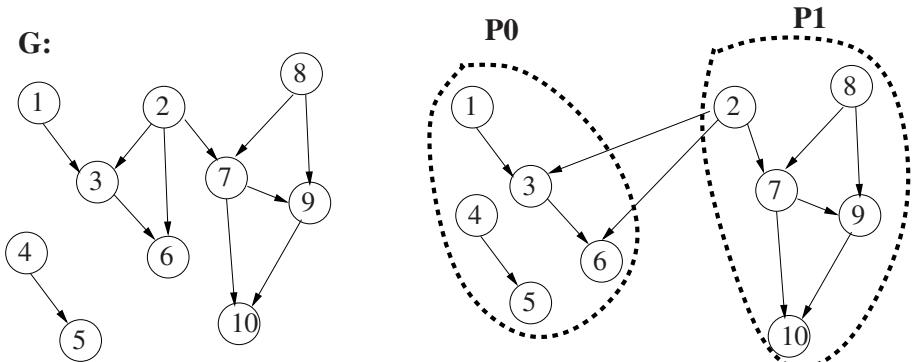
Figure 4.13. Partitioning of the graph G with connectivity 0.24 with an algorithm that produces a quality of 0.45

which use integer linear programming equations for capturing the optimization problem. The solution of the equations provides the exact solution of the temporal partitioning problem. The third category covers *network flow-based algorithms*, whereas, in the fourth category, we have *spectral method* based on the computation of eigenvalues of a matrix derived from the dataflow graph. The methods in the first, third and fourth categories can also be grouped under the umbrella of recursive bipartition, because the partitions are iteratively generated through the bipartition of a remaining set of components.

2. Temporal Partitioning Algorithms

Several heuristics such as the list scheduling algorithm use the mobility range of components to perform further optimizations. This range can be determined on the basis of the earliest and the latest starting time of the corresponding node resulting from the solution of the unconstrained scheduling using the ASAP and ALAP methods. We therefore first present the unconstrained scheduling before focussing on the temporal partitioning methods.

2.1 Unconstrained Scheduling

In general high-level synthesis, *unconstrained scheduling* [156] is the allocation of starting time to components under the assumption that an unlimited amount of resources is available. In the case of reconfigurable devices, this corresponds to placing modules in a device with unlimited size that allows any partition to be implemented. Because the devices in reality have only a limited amount of resources, unconstrained scheduling cannot be used as such. Instead, it is usually used as pre-processing step for other algorithms. It can be used for instance for the computation of the upper and lower bounds on the starting time of operations in a dataflow graph. Whereas the *lower bound* provides the earliest time at which a module can be scheduled, the *upper bound* defines the latest time at which a module can be started. The difference between the upper and the lower bound of a module is its *mobility range*. The mobility interval is given by $[ASAP(v) \ ALAP(v)]$, where $ASAP(v)$ resp. $ALAP(v)$ is the starting time of v computed with the ASAP resp. ALAP-algorithm, that we present later in this section. The starting time of a node cannot be out of its mobility range.

To compute the earliest starting time of each component, the as soon as possible (ASAP)-Algorithm can be used. The ASAP-algorithm traverses the dataflow in topological order starting from the primary input. Each primary input is assigned the starting time 0. A node v_i is processed only when all its predecessors have been processed. The starting time of the node v_i is the highest end time among all its predecessors. The ASAP algorithm idealizes the binding process by assuming an unlimited amount of available resource,

and assigns each operation as soon as it is ready to be executed, therefore providing the lowest starting time of the tasks in the graphs. The pseudocode for the ASAP-algorithm is provided in algorithm 7 for the resulting schedule ς.

Algorithm 7 ASAP-Algorithm for unconstrained scheduling

1: **for** each node $v \in V$ **do**
2:    **if** v has no predecessors **then**
3:       $\varsigma(v) := 0$
4:       $V := V - v$
5:    **end if**
6: **end for**
7: **while** $V \neq \emptyset$ **do**
8:    select a vertex $v_i \in V$ whose predecessors are all scheduled
9:    schedule v_i by setting $\varsigma(v_i) := max_{(v_j,v_i) \in E}(\varsigma(v_j) + t_j)$
10:   $V := V - v_i$
11: **end while**

EXAMPLE 4.20 *Assuming a latency of 100 clocks for the multiplication and 50 clocks for the addition as well as for the subtraction, an example of ASAP-scheduling is provided in figure 4.14. The nodes are labelled with their number as well as their starting time as computed by the algorithm. The label on the edges represent the delay computation delay of the previous node. The data transmission delay is neglected. The number on the nodes represent their starting times.*

Contrary to the ASAP-algorithm, the as late as possible (ALAP)-algorithm traverses the graph from the primary outputs to the primary inputs. A node is processed only if all its successors have been processed. The primary outputs are assigned the maximum starting time, which is an upper bound on the computation time of the dataflow graph. In the simple case, this upper bound is the length (latest starting time − earliest starting time) of the schedule computed by the ASAP-algorithm. The starting time of the node v_i is assigned the minimum value among all the staring times of its predecessor minus the latency of v_i. Algorithm 8 provides the pseudocode of the ALAP-algorithm for the resulting schedule ς and an upper bound λ on the overall computation time.

EXAMPLE 4.21 *With the previous assumption on the module latencies, we schedule the example of figure 4.14 using the ALAP-algorithm with upper bound, thus providing the result of figure 4.15.*
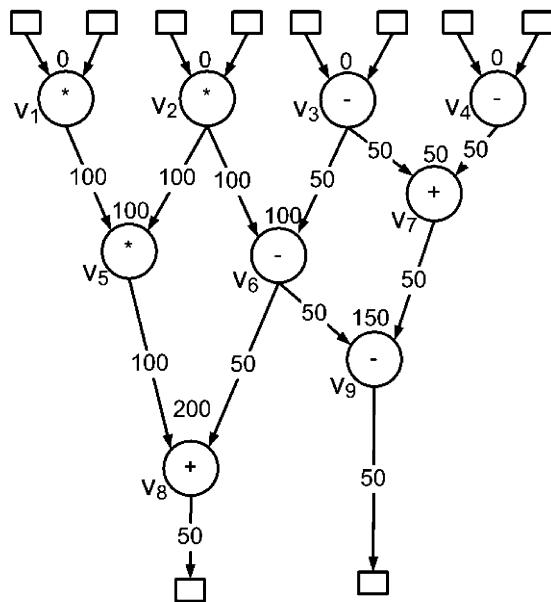
Figure 4.14. Scheduling example with the ASAP-algorithm
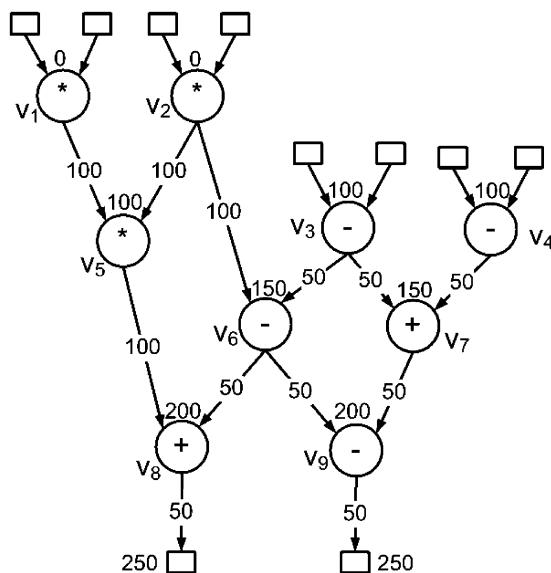
Figure 4.15. ALAP Scheduling example

Algorithm 8 ALAP-Algorithm for unconstrained scheduling

1: **for** each node $v \in V$ **do**
2:      **if** v has no successors **then**
3:          $\varsigma(v) := \lambda$
4:          $V := V - v$
5:      **end if**
6: **end for**
7: **while** $V \neq \emptyset$ **do**
8:      select a vertex $v_i \in V$ whose successors are all scheduled
9:      schedule v_i by setting $\varsigma(v_i) := min_{(v_i, v_j) \in E}(\varsigma(v_j) - t_i)$
10:     $V := V - v$
11: **end while**

The constrained scheduling problem considers the restriction on the architecture. If there are many tasks competing for a given resource at a given time, one of them must be chosen according to a given criteria and the rest will be scheduled later. Sometimes, a task that could be executed at a given time cannot be run because the resource type needed to implement that task is not available at that time. This may delay the execution of the complete set of remaining tasks, if the rest of the tasks depends on that task.

2.2 The List Scheduling Approach

List Scheduling. One of the most used approaches used to solve the temporal partitioning problem is the list scheduling (LS) method. List scheduling was first used as a method for microcode compaction to generate efficient microcode from high-level languages [65]. The idea behind list scheduling in architectural synthesis is to first sort the nodes in topological order and then assign some priority to the node of a dataflow graph. There are several methods to assign a priority to a node. A common strategy is to use the latency weighted depth of the node as its priority [96][143]. The *depth* of a node v is defined as the length (number of nodes) of the longest path from an input to v. The *latency weighted depth* is the same as the latency depth; however, the path from the input to v is weighted using the latency of the operation to be executed by the nodes on the path. Also, the *mobility* of a given node, i.e. the difference between its ALAP-value and its ASAP-value, can be used as its priority.

At any time step t, the so-called *ready set*, that is the set of operations ready to be scheduled, is constructed. The ready set contains operations whose predecessors have already been scheduled early enough to complete their execution at time t. The algorithm checks whether there are enough resources of a given type k to implement all the operations of type k. If so, the operations are

assigned the resources, otherwise, higher priority tasks are assigned the available resources and the rest of the operations will be scheduled later, when some resources will be available. If the mobility of a node is used as priority criteria, it is possible that all operators in the ready list are on a *critical paths*, which means that their mobility is zero. As a consequence, the complete depth of each operators is increased by one, thus increasing the latency of the graph's execution.

EXAMPLE 4.22 *We consider the graph of figure 4.16 on the left side, in which each node is labelled with its priority. The nodes must be scheduled on a resource set made upon an adder and a multiplier. With the priority of a node defined as its depth, the list scheduling is shown on the right side.*

Forced-Directed List Scheduling. The *force-directed list scheduling (FDLS)* from Paulin and Knight [174] is an extension of the list-scheduling algorithm. Instead of computing the priority for each node at the beginning, priorities are dynamically computed at each step. Force-directed list scheduling is a resource-constrained scheduling method aimed at finding a minimum latency schedule for a given resource set. The concept of *force* is used to select
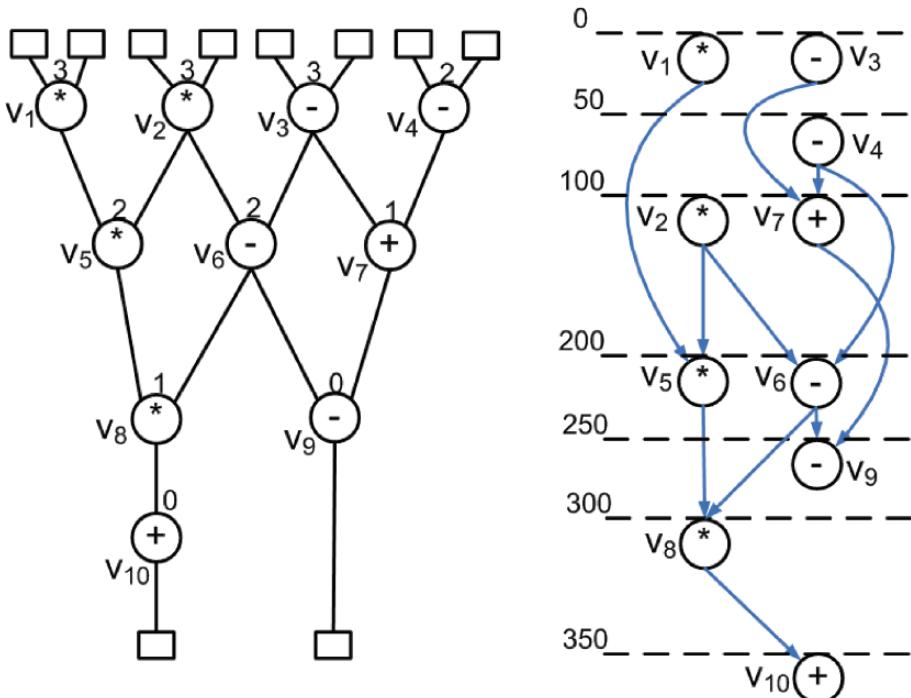
Figure 4.16. An example of list scheduling using the depth of a node as priority

the operation to be assigned a given resource. First, the possible *time frame*, i.e. the mobility interval of each node is computed using the ALAP and ASAP scheduling. Next, the probability $p(v, t)$ of a node v to be scheduled at step t is computed for each node v of the graph. The probability $p(v, t)$ is zero outside the mobility interval of v and is equal the reciprocal of the mobility within the mobility interval. Formally, we have

$$f(n) = \begin{cases} \frac{1}{mob(v)+1} & \text{if } t \in [ASAP(v) \; ALAP(v)] \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where $mob(v) = ALAP(v) - ASAP(v)$.

A distribution function $d(t, k)$ is calculated as the sum of probabilities of all the operations with a resource type k. Formally, we have

$$d(t, k) = \sum_{\{v \in V, \alpha(v) = k\}} p(v, t) \tag{2.2}$$

This can be plotted into a graph called the distribution graph that indicates the concurrency of similar operations over the schedule steps. Whenever many operators compete for a fewer amount of resources, the operations that produce a global increase of concurrency in the graph are selected and assigned the resources.

EXAMPLE 4.23 *Consider once again the graph of figure 4.6 whose ASAP and ALAP time is computed in figures 4.14 and 4.15. Nodes v_1, v_2, v_5 and v_8 have each a mobility of zero. Therefore, $p(v_1, i) = p(v_2, i) = p(v_5, i) = p(v_8, i) = 0, \forall i \in [0 \; , \; 200]$.*

As stated earlier, the scheduling is done on the basis of the force concept, whose role is to attract or repel operators in that scheduling step. Two types of forces are defined: the *self-force* and the *predecessor–successor* force. The self-force is the one that relates an operation to different control steps in which it can be scheduled. For a given node v, the self-force is formally defined as follows:

$$\text{self-force}(v, t) = d(t, k) - \frac{1}{mob(v) + 1} \sum_{m=ASAP(v)}^{ALAP(v)} d(m, k) \tag{2.3}$$

The selection of an operator influences the mobility interval of other operators. This influence can be expressed as a force, called predecessor–successor force that we will not present in detail here. The predecessor–successor force is added to the self-force to build the total forces that are used for the selection process. More details on the computation of forces can be found in [174].

List Scheduling for Reconfigurable Devices. In high-level synthesis for re-configurable devices, the resource type does not play a big role. Only the total amount of basic resources is important. We do not have operators competing for a resource type but for a given surface on the device. Furthermore, we are interested in building complete partitions describing a set of computational steps that will be performed at the same time. Despite the fact that any component has a different starting time and a different end time, only the starting time and the end time of the complete partition is usually considered. Several authors [173, 41, 171, 45, 202, 181] have used this fact to adapt known architectural synthesis algorithm to reconfigurable devices.

The list-scheduling algorithm in reconfigurable devices works in the same way as the common list-scheduling algorithm, by building and updating a list of ready operators. Components are then removed from the ready list and assigned to partitions. The only assignment criterion is that there should be enough places left on the device to accommodate the new component. If this is the case, then the component is placed on the partition currently being built. Otherwise, a new partition is built and the process is repeated until all the nodes of the graph are placed in partitions. The pseudocode for this approach is provided in algorithm 9.

Algorithm 9 List-scheduling algorithm for reconfigurable devices

    sort the nodes of v according to their priorities
    $P_0 := \emptyset$
    while $V \neq \emptyset$ **do**
        select a vertex $v \in V$ with highest priority and whose predecessors are all placed
        if (a partition P_i exists with $s(P_i) + s(v) \leq s(H)$) **then**
           $P_i = P_i \cup \{v\}$
        else
           create a new partition P_{i+1} and set $P_{i+1} = \{v\}$
        end if
    end while

Because the method is an iterative heuristic, several improvements can be done during the construction to generate optimal solutions.

Optimization usually considers the minimization of the overall computation latency of the dataflow graph. The two main factors that influence the overall latency are the latency of each segment and the configuration overhead. If k

segments are generated by an algorithm, then the overall computation time can be formally written as shown in equation 2.4.

$$t_{DFG} = k \times C_H + \sum_{i=1}^{n}(t_{P_i}) \tag{2.4}$$

where t_{DFG} is the overall computation latency for a dataflow graph DFG, C_H is the reconfiguration time of the device H and t_{P_i} is the computation time of partition i.

The minimization of the overall computation time of the given function can be done by tuning with the different parameters in equation 2.4. If the reconfiguration time of the device is too big, then the optimization will tend to minimize the number of partitions to avoid a lot for reconfigurations. However, if the reconfiguration time can be neglected, then only the delay in partitions will be of great interest and a good algorithm will tend to avoid long paths in partitions.

The main advantage of the list-scheduling method is its linear run-time in the number of nodes in the graph. Furthermore the method allows for local optimizations while selecting the nodes to be placed in partitions. This fact is exploited in [173] and [30] to compute a minimum set of resources used by consecutive partitions. In [202], a pair wise interchange is done between adjacent segments to minimize a cost function defined as the sum of squares of the number of nets in each segments.

List scheduling has a major drawback called the *levelization*. Levelization means that the partitions are built on the basis of the level number of the components in the dataflow graph. Modules are assigned to partitions based more on their level number rather than their interconnectivity with other components already placed in the partition. This may have a bad effect on the quality of the partitioning because of a bad connectivity inside the partitions and between the partitions. With this, the goal of minimizing the number of nets connecting two different partitions, i.e. the minimization of the data exchange among partitions becomes difficult to reach.

EXAMPLE 4.24 *Consider the circuit of figure 4.17 partitioned by a list-scheduling algorithm that produces three partitions. Although the components are less connected inside the partitions, the partitions have more edges among each other, thus leading to a poor quality. A better partitioning is shown in figure 4.18 for the same graph. The connectivity is better preserved inside and outside the partitions.*

Despite its drawback in connectivity, the list-scheduling algorithm remains, thanks to its linear computation time, a good temporal partitioning candidate,
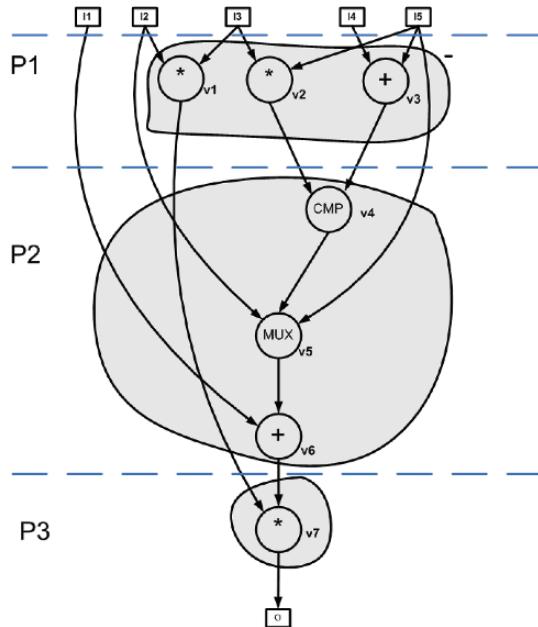
Figure 4.17.   Levelizing effect on the list-scheduling on a dataflow graph
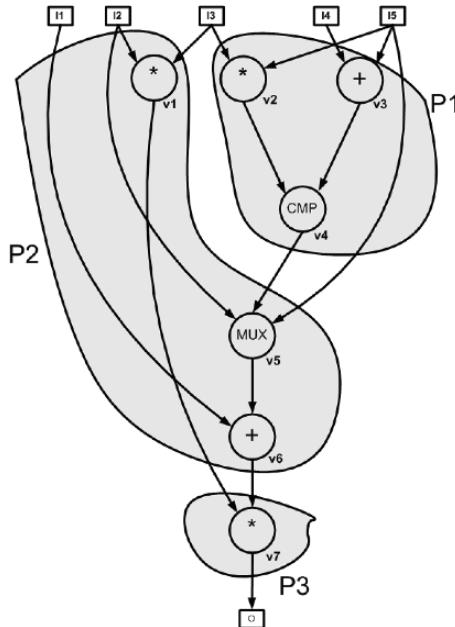
Figure 4.18.   Partitioning with a better quality than the list-scheduling

in particular for low-connected graphs. Also, it can be used to construct initial solutions that can be improved by more complex iterative improvement optimization heuristics.

2.3 Integer Linear Programming

As defined in [49], an instance of an integer linear programming (ILP) problem is defined through the equation 2.5.

$$min \ c^T x \tag{2.5}$$
$$Ax = b$$
$$x \geq 0$$

with A, b and c and x being matrices of integers. It consists of finding a variable x, which minimizes $c^T x$ under the side constraints $Ax = b$ and $x \geq 0$. The ILP method provides an exact approach for solving the general architectural synthesis problem and therefore can be used for the temporal partitioning problem as well.

The approach consists of formulating the temporal partitioning problem as an ILP problem instance and then to use well-known tools to compute the solution to the equation. All problem-related constraints are defined as a set of equations involving only integer numbers. The equations must be solved subject to a given goal to be minimized.

The most important constraints that need to be captured in an ILP problem instance are the *unique assignment constraint*, the *precedence constraint* and the *resource constraint*. The integer requirement in the equation vector entries is due to the fact that we deal with indivisible variables and therefore no real number must be involved.

Before we present the formal description of the constraints, we provide the definitions of the variables that will be used in the equations.

DEFINITION 4.25 (0-1 VARIABLE) *Given a dataflow graph $G = (V, E)$ and a partitioning $P = P_1, ..., P_n$ of G sorted in order of precedence. We defined 0-1 variable y and w as follow:*

- $y_{vi} = 1 \Leftrightarrow v \in P_i$

- $w_{uv} = 1 \Leftrightarrow ((u, v) \in E) \wedge (u \in P_i) \wedge (v \in P_j) \wedge (P_i \neq P_i)$

The role of the y variables is to define the memberships of a given node to a partition. A value of one means that the node is in the corresponding partition, whereas a value of zero means that the node is not in the partition. In the same way, a 0-1 variable for an edge determines if that edge connects two different partitions. The 0-1 variables provide the basis for defining the set of ILP equations that we next formulate.

1 **Unique assignment constraint**: This constraint is used to control the assignment of a node to a single partition. Each node must be assigned exactly to one partition. The unique assignment constraint is formally defined by equation 2.6.

$$\forall v \in V, \sum_{i=1}^{n} y_{vi} = 1 \tag{2.6}$$

The sum of the assignment index of a given variable over all partitions is exactly one, which means that there is exactly one value i for which the y variable is one. This is the index i of the partition in which v is placed.

2 **Precedence constraint**: This constraint is used to control the assignment of data-dependent nodes of the dataflow graph to partitions. A node v that is data dependent from a node u must be placed in a partition with index bigger or equal to that of the partition into which the node u is placed. This constraint is captured by the following equation:

$$\forall (u, v) \in E, \sum_{i=1}^{n} i \times y_{ui} \leq \sum_{i=1}^{n} i \times y_{vi} \tag{2.7}$$

The two sums define the index of the partition in which the components are placed.

3 **Resource constraint**: The resource constraint defines the constraint on the architecture used. This can be limited to the reconfigurable device or to a complete system into which the reconfigurable device is integrated. For a reconfigurable device, the area constraint as well as the constraint on the number of terminals must be specified.

The area constraint states that the total amount of resources assigned to a given partition must not exceed the amount of available resource on the chip. Recall that the computation resources in a reconfigurable device are defined in terms of area occupied on the device. The constraint can therefore be expressed in terms of device area: the total area assigned to a given partition must not exceed the device area. This is formally defined by equation 2.8

$$\forall P_i \in P, \sum_{v \in P_i} a(v) \leq a(H) \tag{2.8}$$

The terminal constraint defined in equation 2.9 states that the total number of input and output in a partition must not exceed the total number of pins on the device.

$$\forall P_i \in P, \sum_{(u \in P_i) \wedge (v \notin P_i)} w_{uv} + \sum_{(u \notin P_i) \wedge (v \in P_i)} w_{uv} \leq p(H) \tag{2.9}$$

where $p(H)$ is the number of terminals (pins) of the device H.

4 **communication memory constraint**: The total amount of data to be temporally saved must not exceed the size of the communication memory used. This constraint is captured by the following equation:

$$\sum_{(u,v)\in E \text{ and } (u\in P_i)\wedge(v\in P_{j(j>i)})} w_{uv} \le M_s \qquad (2.10)$$

where w_{uv} is the width of the edge (u, v) and M_s is the size of the communication memory.

Having formulated the constraint as ILP equations, commercial solvers can be invoked to compute the solutions of the equations. ILP in temporal partitioning was investigated in [76] [135, 136], [93] and [141].

EXAMPLE 4.26 *Considering the partitioning of figure 4.18, let us check if the ILP-constraints hold.*

- *Unique assignment constraint:*
  Partition P_1: $y_{22} = y_{23} = 0$, $y_{21} = 1$ and $y_{32} = y_{33} = 0$, $y_{31} = 1$ and $y_{42} = y_{43} = 0$, $y_{41} = 1$
  Partition P_2: $y_{11} = y_{13} = 0$, $y_{12} = 1$ and $y_{51} = y_{53} = 0$, $y_{52} = 1$ and $y_{61} = y_{63} = 0$, $y_{62} = 1$
  Partition P_3: $y_{71} = y_{72} = 0$ and $y_{73} = 1$

- *Precedence constraint: $1 = \sum_{i=1}^{n} y_{4i} \le \sum_{i=1}^{n} y_{5i} = 2$ and $2 = \sum_{i=1}^{n} y_{6i} \le \sum_{i=1}^{n} y_{7i} = 3$*

- *Resource constraint assuming a device with a size of 200 LUTs, and 100 LUTs for the multiplication, 50 LUTs each for the addition, the comparison and the multiplexer, we have:*
  Partition P_1: $\sum_{u=1}^{7} y_{u1} = (100 + 50 + 50) \le a(H) = 200$
  Partition P_2: $\sum_{u=1}^{7} y_{u2} = (100 + 50 + 50) \le a(H) = 200$
  Partition P_3: $\sum_{u=1}^{7} y_{u3} = (100) \le a(H) = 200$

- *Communication memory constraint: let us assume that a memory with 50 bytes is available for communication and each datum has a 32-bit width. We have communication between partition P_1 and P_2 and between P_2 and P_3.*
  For P_1 to P_2: $w_{45} = 32Bytes \le M_s = 8 \times 50$
  For P_2 to P_3: $w_{67} = 32Bytes \le M_s = 8 \times 50$

The general problem of the ILP approaches for partitioning a DFG is the computation time, which grows drastically with the size of the problem instance.

The algorithm can be applied only to small examples. Branch and bound strategies are sometimes used to limit the search space. In this case, the bounds can be computed using the ASAP/ALAP approach previously described. To overcome this problem, some authors reduce the size of the model by reducing the set of constraints in the problem formulation, but the number of variables and precedence constraints to be considered still remains high. Reconfigurable devices are no longer those tiny devices that could not hold more than four multipliers. Their sizes have increased very fast in the past, and this will continue in the future. Temporal partitioning algorithms should therefore be able to partition very large graphs (graphs with thousands of nodes). Trying to formulate all the precedence constraints with the ILP approach can drastically increase the size of the model, thus making the algorithm intractable.

2.4 Network Flow

The next method that we consider is the network flow approach for solving temporal partitioning. The network flow methodology is based on the Ford and Fulkerson min-cut max-flow theorem [84]. This method has been used in circuit partitioning [231] [123] and also in LUT-technology mapping by the FlowMap method presented in section 3.3.2. The use of the network flow approach to solve temporal partitioning problems was first proposed in [149], [148].

The method is a recursive bipartition approach that successively partitions a set of remaining nodes in two sets, one of which is a final partition, whereas a further partition step must be applied on the second one.

The goal to be reached during partitioning is the minimization of the communication overhead among the partitions, which also means the minimization of the communication memory. The goal is formulated as the minimization of the overall cut-size among the partitions. A little cut-size among the partitions means fewer edges connecting the partitions, less communication and therefore a good partitioning quality.

Formally, we define $\tilde{P}_i$ as the set of nodes to be partitioned at step i. We call $\tilde{P}_i$ the *restgraph* of the original dataflow graph. $\tilde{P}_0 = G$ is the complete graph G. The partitions $P_0, P_1, ..., P_k$ are successively generated using recursive bisection of the sets $\tilde{P}_i$ until the set $\tilde{P}_k$ fulfills the device size and terminal limitations, i.e. the $a(\tilde{P}_k) \leq a(H)$ and $p(\tilde{P}_k) \leq p(H)$.

We consider partitions that generate cycle-free configuration graphs. Configuration graph with cycles can be treated in the same way; however, the implementation of the feedback loops is left to the processor that controls the reconfiguration and the data streaming on the datapaths. With this, the bipartition of the set $\tilde{P}_i$ produces two new sets P_i and $\tilde{P}_{i+1}$ with the following characteristics:

1 $s(P_i) \leq s(H)$ and $p(P_i) \leq p(H)$, i.e. P_i must not be further partitioned.

2 With the ordering of the relation of definition 4.13, only one of the following condition should hold:

  ■ $(P_i \leq \tilde{P}_{i+1})$

  ■ $(\tilde{P}_{i+1} \leq P_i)$

  ■ P_i and $\tilde{P}_{i+1}$ are not in relation.

Those conditions ensure that no cycles exist between P_i and $\tilde{P}_{i+1}$. The network flow method is used in each step of the recursive bipartitioning process to compute a cycle-free bipartition. At each step, the following processing operations are applied to the rest graph $\tilde{P}_i$:

1 $\tilde{P}_i$ is first transformed into a network graph $\tilde{P}'_i$ by introducing two new nodes into the graph $\tilde{P}_i$. The first one is an input-free source node, whose outputs are connected to all the primary inputs. In the same way, a sink node is inserted in the dataflow graph and all the primary outputs are connected as input to it.

2 A second transformation is done on the resulting network $\tilde{P}'_i$ to generate the graph $\tilde{P}_i^2$.

  ■ For an edge $(v_1, v_2) \in \tilde{P}'_i \times \tilde{P}'_i$ two edges $e_1 = (v_1', v_2')$, with a capacity of $c_1 = 1$ and $e_2 = (v_2', v_1')$ with a capacity of $c_2 = \infty$, are added to $\tilde{P}_i^2$.

  ■ For a multi-terminal edge in $\tilde{P}'_i \times \tilde{P}'_i$, a bridging node is added to $\tilde{P}_i^2$. An edge weighted with 1 connects the source node with the bridging node in $\tilde{P}_i^2 \times \tilde{P}_i^2$. For each sink node in the multi-terminal net, an edge weighted with ∞ is added between the bridging edge and the sink nodes and between the sink nodes and the source node.

This transformation is similar to the one done by the FlowMap algorithm in section 3.3.2 before applying the min-cut maxflow theorem. Figure 4.19 illustrates the transformation previously described.

Having transformed the rest graph into a network graph, the maximum flow on the resulting network is computed, leading to a min-cut $(X^2, \overline{X^2})$ of the network $\tilde{P}_i^2$, all the forward edges (from X^2 to $\overline{X^2}$) must be saturated (flow equal to the capacity) and all backward edges (from $\overline{X^2}$ to X^2) should have no flow. If a net is cut, then only bridging edges can connect $\overline{X^2}$ to X^2, thus
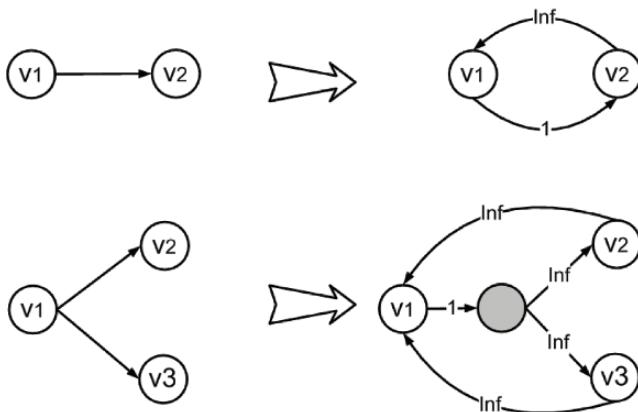
Figure 4.19.   Dataflow graph transformation into a network

preserving the precedence constraints. The computation of the maximal flow is done using the augmenting path method described in [186].

For a computed min-cut $(X^2, \overline{X^2})$ in $\tilde{P}_i^2$, the corresponding cut $(X, \overline{X})$ is induced in $\tilde{P}_i$ by inserting the equivalent nodes from $(X^2, \overline{X^2})$ in $(X, \overline{X})$. The four steps for transformation and partitioning using the network flow approach are illustrated on figure 4.20.

The min-cut max-flow theorem of Ford and Fulkerson is a powerful tool to minimize the communication in a cut in polynomial time. However, the model is constructed by inserting a great amount of nodes and edges in the original graph. The resulting graph $\tilde{P}_i^2$ may grow too big. In the worst case, the number of nodes in the new graph can be twice the number of the nodes in the original graph. The number of additional edges also grows dramatically and become difficult to handle.

2.5   Spectral Methods

In the case where the goal in temporal partitioning is the minimization of communication between partitions, the spectral approach [30][29][28] can be used for temporal partitioning.

The connectivity of a graph can be minimized by placing components in an n-dimensional space in such a way that the sum of the distance between component pairs is minimized. This approach is called the wire length model. As the sum of the distances between component pairs can be minimized if connected components are placed close to each other, the wire length model is likely to provide an optimal placement of the components in an n-dimensional space. Using the wire length model to solve the temporal partitioning with the minimization of communication among the partitions can be done by solving the following two sub-problems:
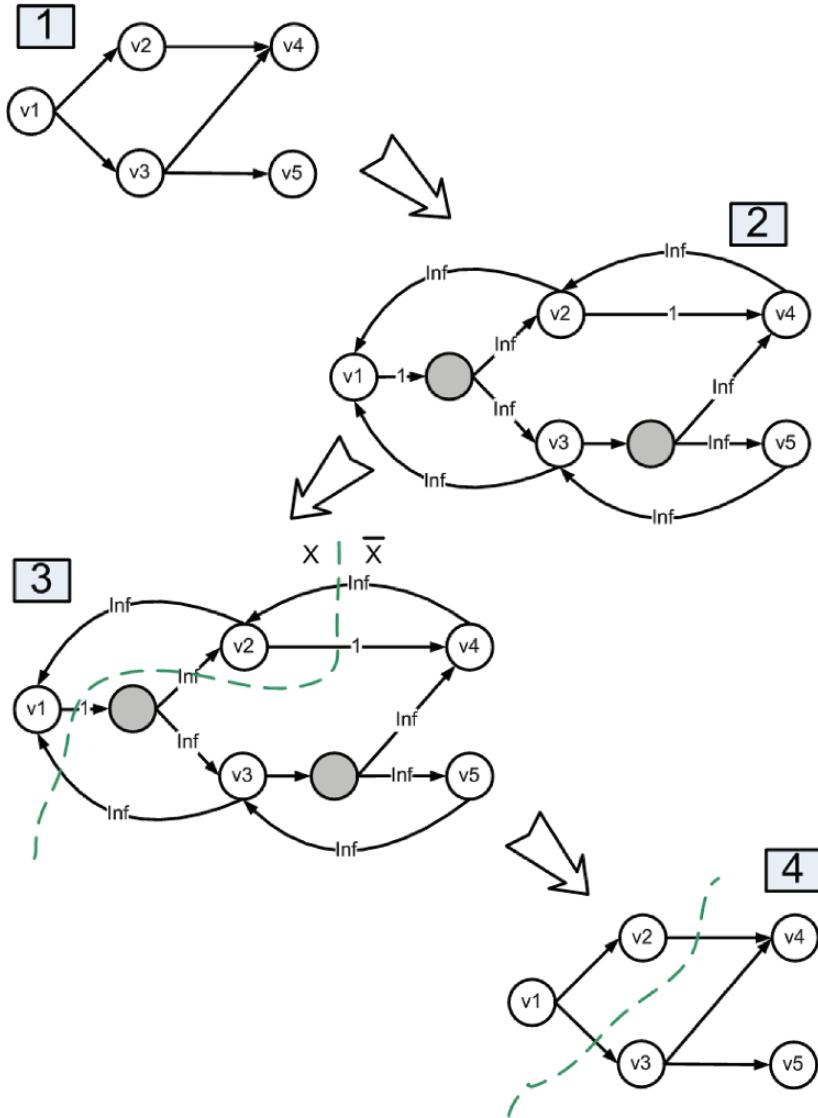
Figure 4.20. Transformation and partitioning steps using the network flow approach

1 Placement of the components in an n-dimensional vector space such as to minimize the sum of the distances between the components. This procedure is known as an n-dimensional spectral embedding [5] [7] [6].

2 Derivation of a partition from an optimal placement that minimizes the sum of the distance between the components.
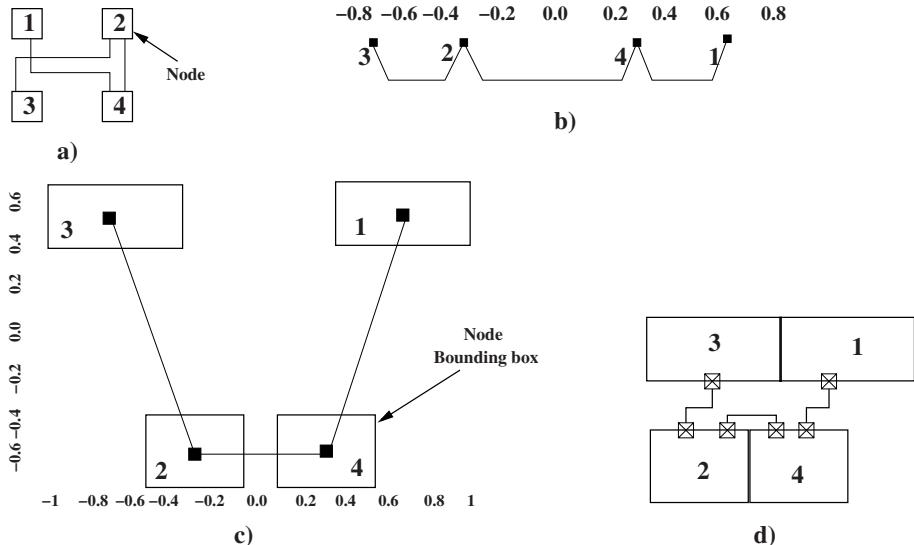
Figure 4.21.   1-D and 2-D spectral-based placement of a graph

We start with the first problem by considering the one-dimensional[7] version as defined by Hall in [107] and illustrated in figure 4.21 [195]. (The subfigure (a) shows an example of a directed graph with four nodes and three edges borrowed from [195], whereas subfigure (b) shows the one-dimensional placement on a line. The small black squares represent the centres of the logic cells. Subfigure (c) shows the two-dimensional placement of the same nodes. The method does not take account of sizes of the logic cells or actual location of logic cell connectors. The scaling of the computed position (by the width and height of the node bounding box). In subfigure (d), a complete layout is made by placing the logic cells on valid locations.

The spectral embedding problem can be stated as follows: Given a dataflow graph $G = (V, E)$, find locations for the $|V|$ nodes that minimize the weighted sum of squared distances between the nodes. If x_i denotes the X-coordinates of node $v_i \in V$ and r denotes the weighted sum of squared distances between the nodes, then the one-dimensional problem is to find the row vector $X^T = (x_1, x_2, ..., x_n)$, which minimizes

$$r = \sum_{i=1}^{n} \sum_{j=1}^{n} (x_i - x_j)^2 w_{ij} \qquad (2.11)$$

---

[7]Placement on a line.

To avoid the trivial case in which $x_i = 0$ for all i, we impose the following condition (normalization):

$$X^T X = 1 \qquad (2.12)$$

We assume that the non-interesting solution $x_i = x_j$ (for all $i, j \in \{1, .., n\}$) is to be avoided. This leads to all components placed at the same location. Next, we define the connection matrix, the degree matrix and the Laplacian or disconnection matrix of G as follows:

DEFINITION 4.27 (CONNECTION MATRIX, DEGREE MATRIX, LAPLACIAN MATRIX) *Given a DFG $G = (V, E)$, we define:*

- *The connection matrix of G as the symmetric matrix $C = (c_{i,j})$ with $(1 \leq i, j \leq\mid V \mid)$ and $c_{i,j} = 1$ if $(v_i, v_j) \in E$ and $c_{i,j} = 0$ otherwise.*

- *The degree matrix of G as the diagonal matrix $D = (d_{i,j})$, $(1 \leq i, j \leq\mid V \mid)$ with $i \neq j \rightarrow d_{i,j} = 0$ and $d_{i,i} = \sum_{j=1}^{|V|} c_{i,j}$.*

- *The Laplacian matrix of G as the matrix $B = D - C$.*

For two nodes v_i and v_j of the DFG connected by an edge, the connection matrix will have an entry one in line i and column j. The degree matrix is a diagonal matrix. An entry in the diagonal (line i, column i) corresponds to the number of nodes adjacent to v_i. The Laplacian matrix is the difference between the degree and the connection. Hall has proved in [107] that:

$$r = X^T B X \qquad (2.13)$$

As B is positive semi-definite ($B \geq 0$) and B is of rank $|V| - 1$, whenever G is connected [107], the initial problem is now reduced to the following:

$$\begin{cases} \text{minimize } r = X^T B X \text{ with } B \geq 0 \\ \text{subject to } X^T X = 1 \end{cases} \qquad (2.14)$$

This is a standard constraint optimization problem that can be solved using the method of the Lagrange multipliers, which is a standard method used to find the extrema of a function $f(x_1, ..., x_n)$ subject to a constraint $g(x_1, ..., x_n) = 0$. An extrema exists if equations (2.15) and (2.16) are satisfied.

$$df = \frac{\partial f}{\partial x_1} dx_1 + \cdots + \frac{\partial f}{\partial x_n} dx_n = 0 \qquad (2.15)$$

$$dg = \frac{\partial g}{\partial x_1} dx_1 + \cdots + \frac{\partial g}{\partial x_n} dx_n = 0 \qquad (2.16)$$

Multiplying (2.16) by a parameter λ to be determined and subtracting the result from (2.15), we obtain equation (2.17):

$$(\frac{\partial f}{\partial x_1} - \lambda\frac{\partial g}{\partial x_1})dx_1 + \cdots + (\frac{\partial f}{\partial x_n} - \lambda\frac{\partial g}{\partial x_n})dx_n = 0 \qquad (2.17)$$

Because the differentials are all independent, we can set any combination equal to zero and the remainder must still be zero. Therefore,

$$(\frac{\partial f}{\partial x_k} - \lambda\frac{\partial g}{\partial x_k})dx_k = 0 \, , \forall k \in 1, ..., n \qquad (2.18)$$

The constant λ to be computed is called the *Lagrange multiplier.*

To solve problem 2.14, we apply the method of the Lagrange multipliers with $f = X^T B X$ and $g = X^T X - 1$. We introduce the Lagrange multiplier λ and form the Lagrangian $L = X^T B X - \lambda(X^T X - 1)$. Taking the first partial derivative of L with respect to X and setting the result equal to zero yields equation 2.19:

$$2BX - 2\lambda X = 0 \iff (B - \lambda I)X = 0 \qquad (2.19)$$

Multiplying equation (2.19) by X^T and applying the constraint (2.12), equation (2.19) yields a non-trivial solution if and only if X is the eigenvector of B which minimizes r and $\lambda(= r)$ is the corresponding eigenvalue. Because the minimum eigenvalue $\lambda_0 = 0$ yields the non-interesting solution $X^T = (1, 1, ..., 1)/\sqrt{(n)}$, the second smallest eigenvalue λ_1 should be chosen. The eigenvector X_1 related to the eigenvalue λ_1 is the solution to the one-dimensional problem (2.14).

For a placement in a k-dimensional vector space, the problem is formulated similar to the one-dimensional version. The entire dimensions involved have to be considered. The following equation must therefore hold.

$$\begin{cases} \text{minimize } R = X_1{}^T B X_1 + X_2{}^T B X_2 + ... + X_k{}^T B X_k \\ \text{subject to } X_1{}^T X_1 = X_2{}^T X_2 = ... = X_k{}^T X_k = 1 \end{cases} \qquad (2.20)$$

(X_i defines the coordinates of the nodes of V in the i-th dimension) has to be solved. Analog to the one-dimensional case, the Lagrange multiplier method will be applied with the k (each for one dimension) Lagrange multipliers $\lambda_1, \lambda_2, ..., \lambda_k$. The solution is the eigenvectors associated to the k smallest non-zero eigenvalues $\lambda_1, \lambda_2, ..., \lambda_k$. This approach is known in the literature as *spectral method*. Spectral methods have been widely used in the past for partitioning and placement [6] [7] [118] [44] [74] [73]. Its run-time is dominated by the computation of the eigenvalues, which can be done using various methods on different architectures. The most used algorithm for computing

the eigenvalues of a matrix is the Golub Kahan method [99] that needs $O(n^3)$ on a single processor for an $n \times n$ matrix. Using $O(n)$ processors, the eigenvalues can be computed with a parallel version of the Hestenes method [200] [172] [119] [38] in $O(n^2 S)$ where S is the number of so-called sweeps [119] [38]. Brent and Luk [38] conjectured that $S = log(n)$ and therefore $S \leq 10$ in general. For sparse and quadratic matrices, the eigenvalues can be computed in $O(n^{1.4})$ using the more efficient Lanczos method [99, 6].

2.6 Application to the Temporal Partitioning Problem

So far we have shown that the spectral method can be used to solve the first part of our problem. The second part, the generation of partitions from a placement with minimum wire length, has to be considered yet. One approach is to consider the placement of the components in a three-dimensional vector space in which the X axis and Y axis represent the device surface and the Z axis represents the time at which each component should be mapped inside the device. In the time dimension, the precedence constraints, $\forall e = (v_i, v_j) \in E, z_i \leq z_j$, between the components of the dataflow graph must hold. A possible solution will consist of first placing the component in the space without taking the precedence constraints into consideration. Thereafter, the partition can be built using an iterative approach in which the constraint can be checked.

Assuming that three-dimensional spectral placement of the component is computed, the partitions $P_0, P_1, \ldots$ and P_k can incrementally be generated using a recursive bisection of the sets $\tilde{P}_i$, which is the set of the modules that still needs to be placed in partitions. Initially we set $\tilde{P}_0 = V$. At step i, the partition $\tilde{P}_i$ is divided into two partitions P_i and $\tilde{P}_{i+1}$. This process is repeated until $\tilde{P}_{i+1} = \emptyset$.

The partition P_i is built at step i by picking components along the Z axis and placing them in P_i until the size of P_i reaches a given limit (the device size). This process creates a bipartition $(P_i, \tilde{P}_i - P_i = \tilde{P}_{i+1})$ of the set of components not yet assigned to a partition. For each computed bipartition $(P_i, \tilde{P}_{i+1})$, we seek exactly one of the following situations: either $(P_i \leq \tilde{P}_{i+1})$, or $(\tilde{P}_{i+1} \leq P_i)$, or there is no relation between P_i and $\tilde{P}_{i+1}$. This means that either no edge exists that connects a component of one partition with a component of another partition or all the edges connecting components between the two partitions have the same direction. This is not always true, as we have used an undirected graph for the placement, and we did not consider the precedence constraint. The bisection has to be improved to fit the cycle-free constraint. We do this by moving nodes from one side of the bisection to the other until a strict order is reached between P_i and $\tilde{P}_{i+1}$.

EXAMPLE 4.28 *To illustrate our temporal partitioning using the spectral method, we consider the graph of figure 4.22 whose laplacian matrix is given in matrix 4.1.*
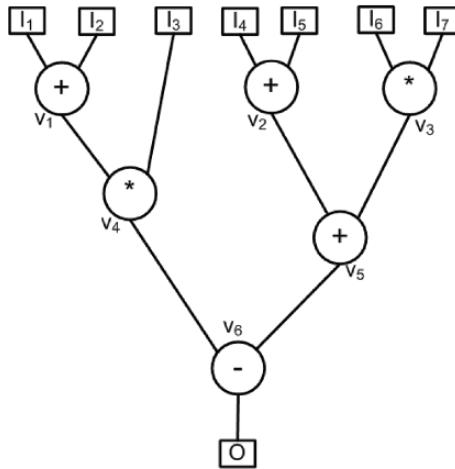
Figure 4.22. Dataflow graph of $f = ((a + b) * c) - ((e + f) + (g * h))$

   The three smallest eigenvalues related to them are $\lambda_6 = 0.112586$, $\lambda_3 = 0.267949$ and $\lambda_1 = 0.438447$ and the related eigenvectors are $(0.0410593, -0.152761, 0.225048, -0.241072, \quad -0.241072, \quad 0.355147), (0.0, 0.0, 0.0, 0.325058, \quad -0.325058, 0.0)$, $(0.34188, \quad 0.0749482, \quad 0.191984, -0.079481, -0.079481, -0.191984)$.

$$
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 3 & 0 & -1 & -1 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 3 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & -1 & 0 & 3 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 3 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 1
\end{pmatrix}
$$

Table 4.1. Laplacian matrix of the graph of figure 4.22
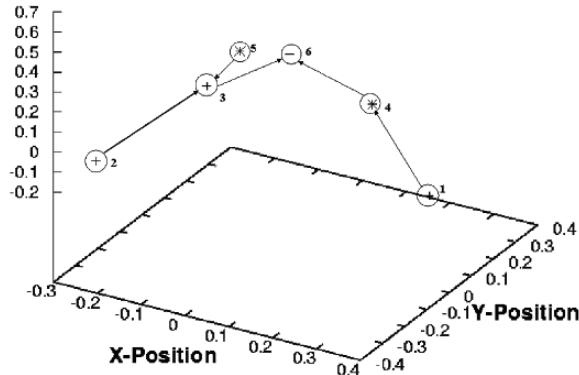
Figure 4.23. 3-D spectral placement of the DFG of figure 4.22

In this example, the primary inputs and primary outputs were taken into account in the building of the matrices. This is useful in that it allows to consider the placement of components in the vicinity of the pins that they use. However, the eigenvectors only define the position of the components in the different dimensions.

Using those three eigenvectors, we computed the three-dimensional placement of figure 4.23.

By picking the component along the Z axis in increasing order of their Z-coordinates, we constructed the two partitions P_0 and P_1 of figure 4.24.
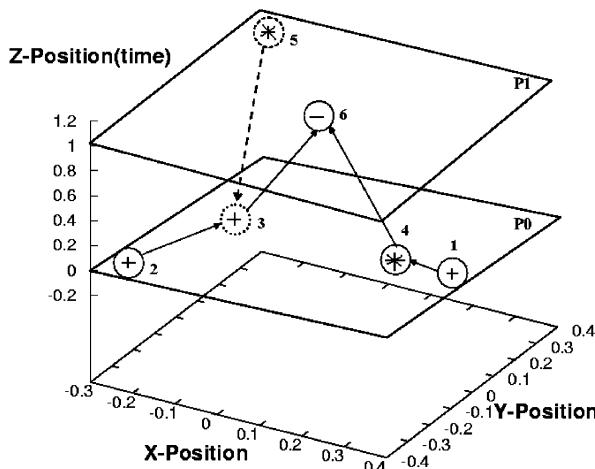
Figure 4.24. Derived partitioning from the spectral placement of figure 4.23

This partition is not ordered as the edges $(3, 6)$ and $(4, 6)$ have their source in P_0 and their destination in P_1, whereas the edge $(5, 3)$ has its source in P_1 and its destination in P_0. This produces a configuration graph with a cycle. By exchanging the nodes 3 and 5 from P_0 to P_1 and vice versa, the partitioning becomes an ordered one.

2.7   Elimination of Cycles in the Configuration Graph

Elimination of cycles in the configuration graph is important in methods that perform an iterative bisection of a rest graph. This is the case for the list-scheduling, the network flow method and the spectral method. The approach that we present here is more general than just limited to one particular method.

The cycle elimination method is based on the well-known iterative improvement procedure of Kernighan and Lin (KL) described in [137] [83] [144]. As the KL-algorithm deals only with undirected graphs and because we target directed graphs in this work, some modifications have to be done on the original KL algorithm.

The fundamental idea behind the KL-algorithm is the definition of a *cut* of a bisection as well as the notion of the *gain* of moving a vertex from one side of the bisection to the other. For an undirected graph, a cut is defined as the weighted sum of all the edges crossing from one partition to another. By moving a node from one partition into the other, the number of crossing edges is also modified and the value of the cut changed. The KL-algorithm allows a series of moves, which reduce the bisection cut. If the gain of moving a vertex is positive, then making that move will reduce the total cost of the cut in the partition. During one iteration of the KL-algorithm, nodes are moved from one side of the bisection and locked on the other side. The cost of swapping unlocked nodes in opposite parts is then computed, and the nodes with the best gain (greatest decrease or less increase of the cut) are swapped. If all the nodes are locked, the lowest cost partition is set to the current computed partition, if it improves the cost of the cut. One iteration of the KL-algorithms is called a *pass*. After one pass, all the nodes are unlocked and a new pass is computed. The iteration terminates if a pass produces no further improvement on the cut. For a more detailed description of the KL-methods and its extension by Fiduccia and Mattheyses, refer to [137] [83] [144].

The KL-method works on undirected graphs and does not differentiate between the direction of edges. It does not matter if an edge crosses from the first partition to the second partition or vice versa. In temporal partitioning, the graph is directed because of precedence constraints; thus, the original KL-algorithm has to be modified to fit our needs. To better explain how the modification can be done on the original KL-method, we first provide more definitions.

DEFINITION 4.29 *Let $G = (V, E)$ be a dataflow graph. For two nodes v_i and v_j in V and a bipartition $\{P, Q\}$ of G, we have:*

- $E_P(v_i) = \sum_{\{(v_i, v_j) | v_j \in P\}} w_{ij}$ *is the weighted sum of the edges from v_i to P, i.e. edges connecting v_i to nodes in P (figure 4.25).*

- $I_P(v_i) = \sum_{\{(v_j, v_i) | v_j \in P\}} w_{ji}$ *is the weighted sum of the edges from P to v_i, i.e. edges connecting nodes in P to v_i (figure 4.25) and*

- $E_{P,Q} = \{(v_i, v_j) \in E \mid v_i \in P \text{ and } v_j \in Q\}$ *is the set of edges crossing from partition P to partition Q, i.e. edges having their sources in P and their destination in Q.*

At step i of the temporal partitioning, a bipartition $(P_i, \tilde{P}_{i+1})$ is generated. The precedence constraint $(P_i \leq \tilde{P}_{i+1}$ or $\tilde{P}_{i+1} \leq P_i)$ can be captured through the following equation: $E_{P_i, \tilde{P}_{i+1}} = \emptyset$ or $E_{\tilde{P}_{i+1}, P_i} = \emptyset$. Intuitively, we would like to combine the two variables $E_{P_i, \tilde{P}_{i+1}}$ and $E_{\tilde{P}_{i+1}, P_i}$ in the definition of our cut and try to decrease one of them to zero using the KL-algorithm. But decreasing one variable could have the negative effect of increasing the second one, thus producing a cycle of improvement and alteration on the cost of the cut. To avoid this, we apply two instances of the KL-algorithm on the same
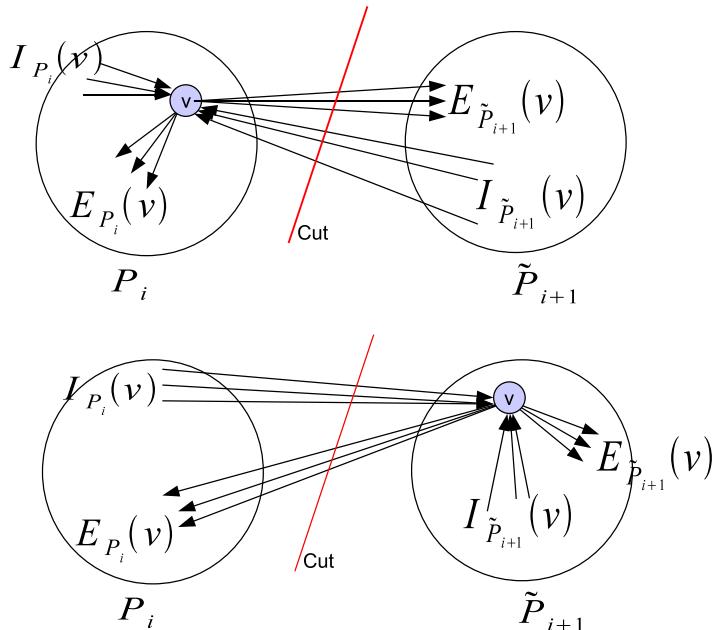

Figure 4.25. Internal and external edges of a given nodes

bisection in parallel. The objective is to have $E_{P_i, \tilde{P}_{i+1}} = \emptyset$ on the first computation path and $E_{\tilde{P}_{i+1}, P_i} = \emptyset$ on the second one. Obviously, the cost of the cut is set to $|E_{P_i, \tilde{P}_{i+1}}|$ on the first path and $|E_{\tilde{P}_{i+1}, P_i}|$ on the second path. After one pass on each path, we check whether the objective has been reached on one path. If this is the case, then the result is set to be the partition generated on that path. Otherwise, a new pass is computed on the two computation paths. The gain of moving a node is defined differently on the two computation paths.

- On the first path where the goal is to have $|E_{P_i, \tilde{P}_{i+1}}| = 0$, the gain of moving a node j from P_i to $\tilde{P}_{i+1}$ is $I_{P_i}(v_j) - E_{\tilde{P}_{i+1}}(v_j)$ and the one of moving a node k from $\tilde{P}_{i+1}$ to P_i is $E_{\tilde{P}_{i+1}}(v_k) - I_{P_i}(v_k)$.

- On the second path where to goal is to have $|E_{\tilde{P}_{i+1}, P_i}| = 0$, the gain of moving a node j from P_i to $\tilde{P}_{i+1}$ is $E_{P_i}(v_j) - I_{\tilde{P}_{i+1}}(v_j)$ and the gain of moving a node k from $\tilde{P}_{i+1}$ to P_i is $I_{\tilde{P}_{i+1}}(v_k) - E_{P_i}(v_k)$.

The gain defined on each computation path is the same as the one defined in the original KL-algorithm. The modified version of the KL-algorithm presented here will produce the desired result on one path. Because the targeted graphs are acyclic dataflow graphs, there exists a partition in which all the edges cross from the first one to the second. Such a partition is provided for example by a list-scheduling algorithm.

2.8   Local optimization: Context Switching, Time Multiplexing and Configuration Switching

Reconfiguration time has always been a major problem in reconfigurable computing. Switching from one configuration to the next can take up to a fraction of a second in large FPGAs, according to the technical realization of the system. With millions of iterations, the reconfiguration overhead becomes too high to allow a use of the device in real-life problem solving. To decrease reconfiguration overhead, the architectural concepts of time-multiplex and context switching FPGAs have been proposed [202] [188]. The general idea behind those two concepts is to store a given number of configurations directly on the chip. This avoids the downloading of a large amount of reconfiguration data when required.

Trimberger [202] proposed an FPGA architecture in which designs are modelled as Mealy state machines. Microregisters are used to temporally hold computation data when switching from one configuration to the next. The combinational logic receives its inputs from the device inputs and from the flip flop outputs, and the device outputs come from combinational logic and from flip flops outputs.In *logic engine* mode, the device emulates a large design in

many *microcycles*. In each microcycle, resources are allocated to a new configuration. A similar architecture has been proposed by Scalera et al. [188]. Data pipes are used here to exchange data between different configurations also called *contexts*. A *data pipe* contains a plurality of context switching logic arrays (CSLA), which can be used to process two 16-bit words. An incoming context can then pick its input data where its predecessor left off by acquiring the intermediate data deposited on the rightmost portion of the pipe and processing it in a pipeline from right to left. Unfortunately, the methods developed have remained in a conceptual stage. Neither time multiplexing nor context-switching FPGAs have ever been commercialized.

A more practical approach has been proposed in [30] to reduce the reconfiguration time. It is based on the observation that traditional list-scheduling based temporal partitioning algorithms can produce a series of configurations based on the same set of operators if the components are 'well ordered'[8] in the list. For example, for two consecutive configurations $\zeta_i = \{C_1, ..., C_{k_i}\}$ and $\zeta_{i+1} = \{C_1', ..., C_{k_{i+1}}'\}$ representing two partitions P_i and P_{i+1}, one can be the subset of the other one, i.e either $\zeta_i \subseteq \zeta_{i+1}$, or $\zeta_{i+1} \subseteq \zeta_i$. If one of those two situations arises, then the reconfiguration overhead can be reduced by implementing the two partitions P_i and P_{i+1} in one configuration $\zeta_{new} = \zeta_i \cup \zeta_{i+1}$. The components of ζ_{new} will then be shared between the two partitions P_i and P_{i+1}. That means, the modules required for both configurations are placed on the device and wired in two different ways. Each way corresponds to one configuration. Figure 4.26 shows a partitioning of a graph into two partitions P_0 and P_1. The set of components required to implement P_1 is a subset of the set of components required to implement P_0.

With the use of multiplexers on the inputs of the operators in and the use of selection signals to connect the corresponding signals to the module inputs, it is then possible to implement the connections defined in configuration ζ_i as well as those defined in ζ_{i+1} together. Switching from the configuration ζ_i to the configuration ζ_{i+1} can be done by setting the corresponding value of the selection signals. The device is therefore 'reconfigured' without changing the physical configuration. We call this process *configuration switching*. With configuration switching, there is no need to save the registers of the FPGA in the processor address space during reconfiguration because no register is altered. With this, configuration switching will help to reduce the data exchange between the processor and the reconfigurable.

Configuration switching can be extended to a series of configurations $\zeta_1, ..., \zeta_r$ by extracting the smallest amount of common operators needed to implement all the configurations and implement the configuration switching on $\zeta_1, ..., \zeta_r$.

---

[8]A well order defines the way components with the same level number should be place in the list before partitioning.
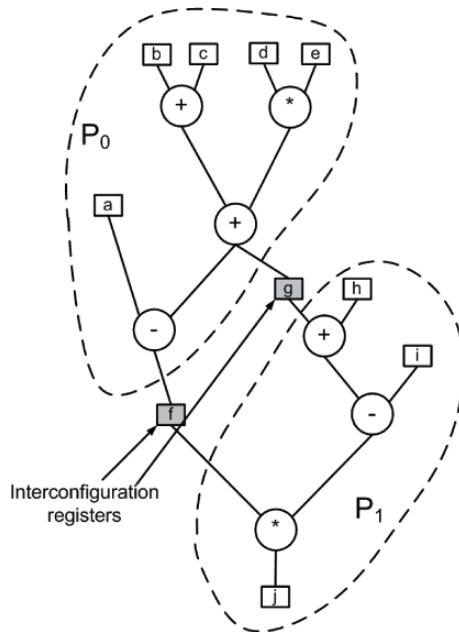
Figure 4.26.   Partitioning of a graph into two sets with common sets of operators

But the amount of multiplexers needed to assign the operators to a particu-
lar configuration as well as the difficulty to route all the possible configura-
tions can jeopardize the implementation of configuration switching for a high
number of configurations. The amount of partitions to be implemented using
configuration switching must be kept small.

EXAMPLE 4.30 *Figures 4.28 shows the implementation of configuration
switching on a device for the two partitions of figure 4.27.*

*During reconfiguration, the corresponding values for the selection signal
are set on the multiplexers. The output of the corresponding partitions are
then selected and sent to the operators.*

Although configuration switching can save time and reduce the data trans-
fer between the processor and the reconfigurable device, its implementation
usually requires additional resources. If the amount of additional resource be-
comes too big, then it makes no sense to have a smaller number of components
implemented, and a larger logic area on the device just to realize the switch.
The tradeoffs between the numbers of additional resources and the number of
configurations to be implemented has to be chosen carefully. The search for
a tradeoff between the number of configurations to reside in the device and
the complexity of the final design is an optimization problem that we do not
address in this book, namely the exploration of the design space.
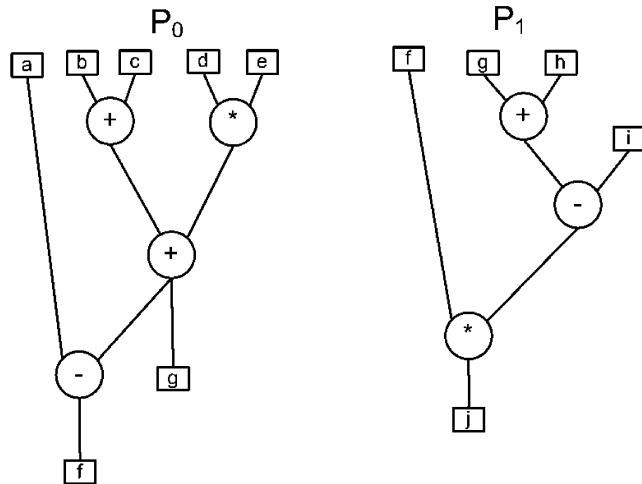
Figure 4.27. Logical partitioning of the graph of figure 4.26
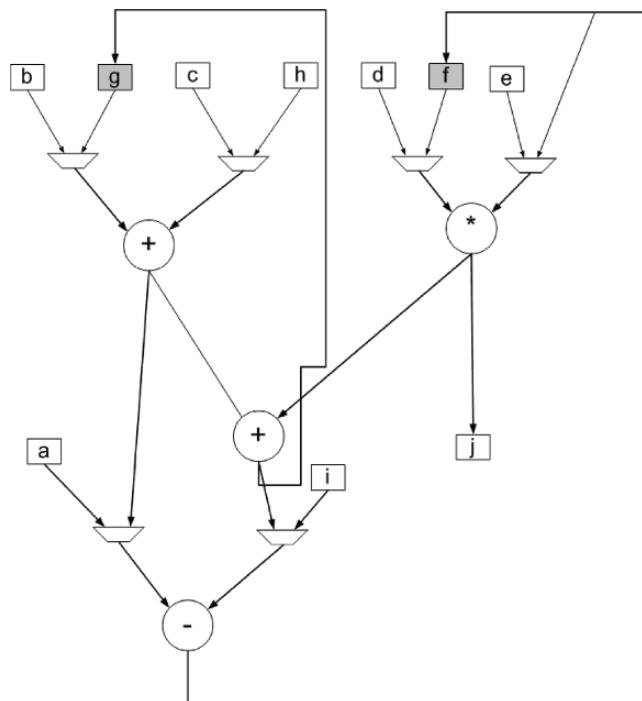
Figure 4.28. Implementation of configuration switching with the partitions of figure 4.27

3.   Conclusion

High-level synthesis for reconfigurable devices has benefitted from the various method provided in the past for the general high-level synthesis problem. The main difference between the two approaches is in the freedom of choice on the operator type in the case of reconfigurable device. However, one must deal with temporal partitioning, which allows the reuse of chip resource over time. Time multiplex reuse of resources was a great matter of concern for a decade, as the capacity of FPGAs was very small to accommodate one of the complete functions to be implemented. Meanwhile, the size of FPGAs has grown a lot, and it is quite difficult to find those functions to be temporally partitioned to exploit chip resources. The interest on temporal partitioning methods has considerably decreased as a consequence of this growth in the size of FPGAs. This does not mean that temporal partitioning is useless. We still have areas of computation such as rapid prototyping where reconfigurable device can provide a very cheap alternative to the very expensive existing systems. Rapid prototyping systems are usually done with expensive machines containing a set of FPGAs. Applications are then partitioned among the FPGAs, which compute each part of the function to be implemented. With temporal partitioning, a few number of FPGAs can be used by several parts of the applications that must not be computed at the same time.

Chapter 5

TEMPORAL PLACEMENT

In the last chapter, we presented the high-level synthesis problem for reconfigurable devices and some solution approaches. The result is a set of partitions that are used to reconfigure the complete device. While the implementation of single partitions is easy, the amount of waste resources in partitions can be very high. Recall that the waste resource of a component is the amount resources occupied by that component multiplied by the time where the component is idle on the device.

Wasting resources on the chip can be avoided if any single component is placed on the chip only when its computation is required and remains on the device only for time it is active. With this, idle components can be replaced by new ones, ready to run at a given point of time. Exchanging a single component on the chip means reconfiguring the chip only on the location previously occupied by that component. This process is called *partial reconfiguration* in contrast to *full reconfiguration* where the full device must be reconfigured, even for the replacement of a single component. To be exploited, the partial reconfiguration must be technically supported by the device, which is not the case for all available devices. While most of the existing devices support full reconfiguration, only few are able to be partially reconfigured.

For a given set of operations to be executed, the resource allocation on the device is a time-dependant process in which not only the placement of the components on the device is defined, but also the time slot in which the execution of the task must be performed. The time-dependant placement of task on the device is called *temporal placement*.

Temporal placement can be graphically illustrated through an arrangement of rectangular boxes in a 3-dimensional container whose base is defined by the surface of the device and the high is the time axis. Each box represents a

components with a given surface placed at a given location on the device at a time defined on the time axis.

EXAMPLE 5.1 *In figure 5.1, a temporal placement of a set of components* $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$ *is given. Component v_1 for example occupies a surface on the defined by its length and height for a time slot 0 to 55, which corresponds to its computation latency. Component v_5 starts its execution at time 90 and occupies among others, part of the device that was perviously occupied by component v_4. Component v_8 occupies a part of the device for the whole computation time.*

The computation of a temporal placement for a given set of components representing some tasks to be implemented can be done offline, at compile-time or online, while the device is executing. In the first case, for an application specified as dataflow graph that must be computed on the device, the computation sequence is defined at compile-time and remains fixed for the given function. In online temporal placement or simple said, online placement, the computation sequence is not known in advance. The creation of task that must be placed on the device is done dynamically at run-time. It is therefore not possible to capture the specification of an algorithm for online placement through a given model at compile-time.
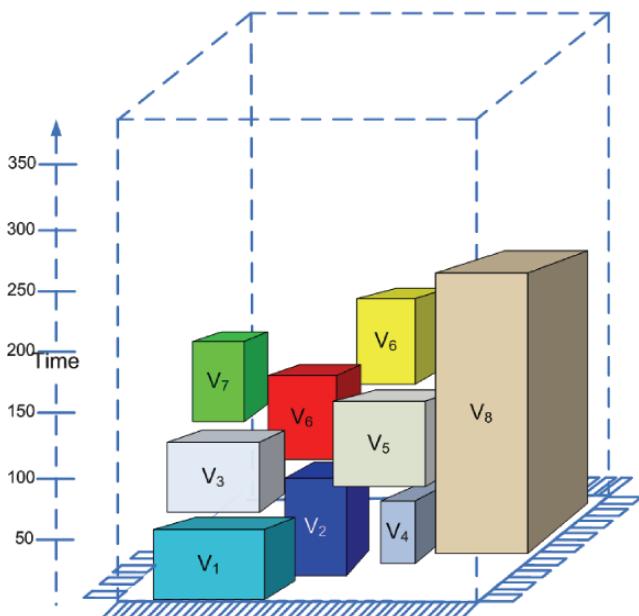

Figure 5.1.   Temporal placement as 3-D placement of blocks

Temporal placement has the advantage of being highly flexible and efficient in terms of device utilization; however, it also has a major drawback. Efficient temporal placement algorithms are difficult and cost-intensive. Online placement requires to solve some computational intensive problems at run-time in a fraction of millisecond. Those are the efficient management of the free space, the selection of the best site for a new component and the management of communication. For a new module to be placed at run-time, it is not sufficient to only compute the best placement site. The communication between the modules running on the chip must also be considered. The communication between modules running on the chip and the external world must be taken into account as well. A reconfigurable device must provide a viable online communication mechanisms to help establish the communication among modules on the chip at run-time. This problem is not easy to solve and requires most of the time, some prerequisites on the device architecture.

Although communication aspects such as the distance among components are considered in some of the methods provided in this chapter, we do not deal with the technical realization of communication here. Chapter 6 is devoted to this topic. We assume that communication among modules placed on the device is somehow possible.

The first part of this chapter addresses the offline temporal placement, whereas the second part deals with the online temporal placement. In both cases, we start with some definitions related to the part being addressed, and then we present existing approaches to solve the corresponding problem.

1. Offline Temporal Placement

We first provide a formal definition of the temporal placement problem before investigating existing solution approaches. For sake of simplicity and compatibility, we use the definitions provided in [199] [82].

DEFINITION 5.2 [(OFF-LINE) TEMPORAL PLACEMENT]
Given a dataflow graph $G = (V, E)$ and a reconfigurable device H with length (H_x and width H_y), a temporal placement is a 3-dimensional vector function: $p = (p_x, p_y, p_t) : V \to \mathbb{R}^3$, where p_t defines a feasible schedule.

For a given node v_i, the values $p_x(v_i)$, $p_y(v_i)$ and $p_t(v_i)$ denote the coordinates of the node v_i in a 3-dimensional vector space. $p_x(v_i)$ and $p_y(v_i)$ are the coordinates of v_i on the device H, while $p_t(v_i)$ defines the starting time of v_i on H.

We next present some solution approaches for the offline temporal placement problem. We first introduce a simple incremental method based on the first-fit/best-fit concept. In the second method, a clustering of components in blocks that can be placed together on the chip is done. Finally, we introduce

an exact method that characterized valid partitioning as a set of constraint equations, used to describe the 3-D packing of boxes in containers.

1.1   First-Fit and Best-Fit Placement

Given a dataflow graph for which a temporal placement must be computed, a simple approach will consist of continuously keeping track of free locations on the chip. The nodes to be placed will then be selected according to their readiness. A node is said to be ready to be placed, if all its predecessors are already placed. For each selected node, one of the free locations on the chip is selected, and the node is placed at that location, provides that the space is large enough to accommodate the selected node. If the selection procedure chooses the first free place on the chip to place the new component, then we call it a *first-fit*.

The *first-fit* procedure is fast and run in linear time in the number of free locations. However, if we assume that each selected location can accommodate only one component, the amount of unused resource created by the component can be very high, if a location is selected whose surface is bigger than that of the component. To avoid this situation, we may choose a best-fit strategy in which the component is placed on the location whose surface is the closest to that of the component. We call this approach a *best-fit*. *Best-fit* is much time consuming than first-fit, because the complete list of free locations must be searched for the best location for the component to be placed.

The definition of a location has a great importance in the first-fit and best-fit strategy. What is a location? A simple rectangle? or is it a shape with an arbitrary contour? For sake of simplicity, we assume that a location is a rectangle, in which case the test of inclusion of a given component in a free location is easy. What happens with the new free locations that results from the placement of a component on a selected free area of the device? Just ignoring those free fragmented locations can be inefficient. In many case, the total amount of fragmented area can be enough to hold some components. If we do not keep track of the fragmented area and if those fragmented area are not consecutive, then many components cannot be placed on the device, although enough free space exist to accommodate those components. Managing the fragmented locations in turn requires a very large book keeping efforts. In Section 2, we will present some algorithms that deal with the fragmentation issues in the context of online temporal placement.

A possibility to keep the first-fit and the best-fit approaches simple is to segment the device in sectors, each of which is defined as a locations. Such a location is able to hold a given amount of components, whose total area cannot exceed the size of that sector. This approach is attractive for many devices in which the partial reconfiguration is done with some restrictions, as it is the case for example with the Xilinx Virtex and Virtex II FPGA, where the

reconfiguration can be done only column wise. Whenever a component has to be exchanged, all the components sharing some columns with that component are affected. The situation is better in the Virtex 4 and Virtex 5 families, where the reconfiguration does not affect the complete device column, but only the complete height of a rectangular block within the chip. By defining the location to span the complete device column in the case of the Virtex and Virtex II and the block height for the Virtex 4 and Virtex 5, we avoid to disturb the components already running on the device during the reconfiguration process.

Given a dataflow graph and a device partitioned in locations, we would then like to place several components at the same time at a given location on the device. The dataflow graph must then be partitioned in sets of components such that each set can fit into a location. Several partitioning strategies exist in the literature. However, most of them does not take the precedence constraints into account. Partitioning strategies are not presented in this section. We refer to the temporal partitioning algorithms presented in the previous chapter that can also be used here. In this case, the size constraint must be the size of a location. To limit the amount of wasted resources on the chip, a clustering algorithms that tries to place components in partitions according to their finishing time was presented in [30]. This has the advantage that all the components finish at the same time, thus minimizing the wasted amount of resources.

Let us now assume that for given a dataflow graph $G = (V, E)$, a partitioning or better say a clustering $C = \{C_1, ..., C_n\}$ is computed such that all the clusters fulfill the size constraints. Algorithm 10 computes a temporal placement of C in a first-fit manner. For a new cluster C_{act} to be placed on the device, the algorithm checks for the cluster C_{top} with the minimum run-time among the clusters allocated to the device for which the precedence constraint $C_{top} \leq C_{act}$ between C_{top} and C_{act} holds. The cluster C_{act} is placed on top of C_{top}. The placement of cluster C_{act} on top C_{top} simply means that the resources allocated to C_{top} will be allocated to C_{act} after C_{top} has completed its execution. The algorithm stops when all the clusters have been placed.

EXAMPLE 5.3 *Figure 5.2 illustrates the firs-fit temporal cluster placement on a set of ten clusters. We assume that the precedence constraints are satisfied*

Algorithm 10 First-fit temporal placement of clusters

   while while all the clusters are not placed do **do**
      Select one cluster C_{act} from the list of *ready to run* clusters
      From the clusters already placed, select the cluster C_{top}
      with the smallest finishing-time such that $C_{top} \leq C_{act}$
   place the cluster C_{act} on top of C_{top}
   end while
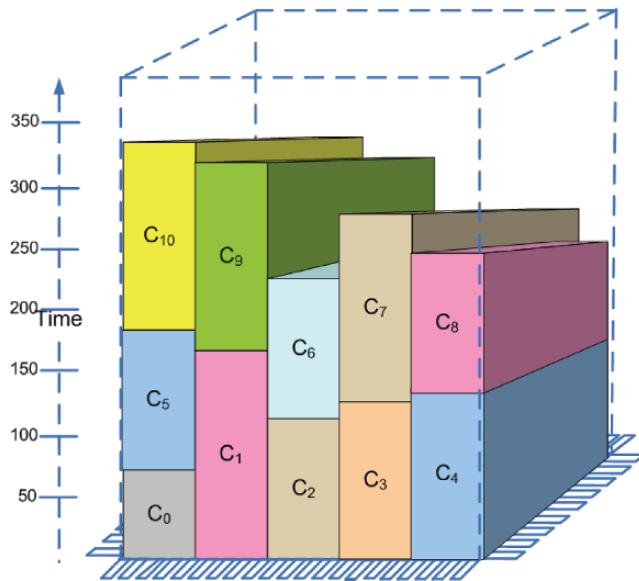
Figure 5.2. First-fit temporal placement of a set of clusters

during the placement. At the beginning, the clusters C_0, C_1, C_2, C_3 to C_4 are completely placed on the device. To place cluster C_5, cluster C_0 with the earliest finish time is selected, and cluster C_5 is placed on top of C_0. Cluster C_6 occupies the space freed by cluster C_2, and cluster C_7 is placed on top of C_3. C_8 is then placed on top of C_4, C_9 on top of C_1 and C_{10} on top of C_5. The configuration sequence produced by this example is ($\{0, 1, 2, 3, 4\} \rightarrow \{5, 1, 2, 3, 4\} \rightarrow \{5, 1, 6, 7, 4\} \rightarrow \{5, 1, 6, 7, 8\} \rightarrow \{5, 9, 6, 7, 8\} \rightarrow \{10, 9, 6, 7, 8\}$).

The first-fit and best-fit approaches provide fast method to compute a temporal placement solution. However, no effort is spent on the efficiency of the space management. Integer linear programming can be used as exact method to solve the temporal placement problem. In this case, a set of constraint equations that must be fulfilled by each solution must be formulated. Solving the equations will then provide a solution to the temporal placement problem. Because the computation of a ILP-solution is computational intensive, integer linear programming is usually suitable only for small-size problems. An exact method that we next presented was proposed in [199] [82] to solve the temporal placement for large-size problems. Equations are formulated in a similar way as in integer linear programming. However, a branch and bound strategy is used to reduce the search space and allow much large problems to be solved.

1.2 Packing Approach for Temporal Placement

The idea of modeling the temporal placement problem as a 3-D packing problem, which is the problem of deciding if a given set of boxes can be placed within a given container of size (x, y, h) representing the length, the width and the height of the container, was first presented in [199] [82]. According to the goals seeked, several variations can be made in the formulation of the problem. A goal can be for example the minimization of the overall execution time or the minimization of the amount of wasted resources for a given algorithm. Let us consider two possible variations of the packing problem: the *base minimization problem (BMP)* and the *the strip packing problem (SPP)*

1 *Base Minimization Problem*: Given a set of boxes B and a height h, the problem of finding a container with minimal size (x, y, h) that can accommodate the set of boxes B is called the BMP. The corresponding goal in temporal placement is to find the device with minimum size (x, y) on which a set of components can be implemented given an overall run-time $t = h$.

2 *Strip Packing Problem*: Given a set of boxes and a base (x, y), the problem of finding the minimum height h container with size (x, y, h) that can hold all the boxes is called the SPP. The analogy with the temporal placement is to find the minimum run-time $t = h$ for a set of components given a reconfigurable device with size (x, y).

With the optimal solution of the BMP, one can select the best reconfigurable device, in term of size, on which a set of tasks can be computed in a limited amount of time. This device is optimized for the given set of task and for the given run-time. In reconfigurable computing, the main task is not necessary to choose an optimal device for a fixed set of tasks, but to have a fixed device on which various set of task can be optimally implemented. The BMP is therefore not the right tool to be used.

The SPP that we next focus on best matches the requirements of having only one device on which different set of tasks that can be implemented at different time. In its original version, the goal of the SPP was to minimize the overall computation delay. This may be changed according to the objectives seek.

As stated earlier in this section, integer linear programming can be used to formulate all constraints that must be fulfilled for a given solution, as a set of equation to be solved. However, this approach can be used only for small-size problems. The concept of packing classes that we next present was used in [199] [82] as a mean to define a feasible solution for the SPP.

1.2.1   Packing Classes

The solution of an instance of the temporal placement problem can first be characterized through a valid packing, using the concept of packing classes. This approach is limited to the pure packing of boxes in a container and does not capture the precedence constraints in the dataflow graph.

Providing an orientation to the packing classes can then be done in a second step to capture the precedence constraints of the dataflow graph in a valid solution. The description provided in [199] [82] considers packing in arbitrary dimension. We are interested only in a 3-D packing and will therefore limit our description accordingly. We start by providing some definitions of the terms needed in this section.

DEFINITION 5.4 (INTERVAL GRAPH) *Given a dataflow graph $G = (V, E)$, a reconfigurable device H and a packing of V into H, i.e a 3-D placement of the components of V on the device H. An interval graph of G is a graph $G_i = (V, E_i), \forall i \in \{1, 2, 3\}$ such that: $(v_k, v_l) \in E_i \iff$ the projections of the nodes v_k and v_l overlap in the i-th dimension, i.e on the i-th axis.*

DEFINITION 5.5 (COMPLEMENT GRAPH) *Given a dataflow graph $G = (V, E)$, a reconfigurable device H and a packing of V into H, The complement graph of an interval graph $G_i = (V, E_i)$ is the graph $\overline{G}_i = (V, \overline{E}_i), \forall i \in \{1, 2, 3\}.$*

The strategy presented here consists of first providing a better characterization of a solution. With this, potential solutions can be checked for validity. The second step will then consists of constructing arrangements, each of which is tested for its validity as solution. The validity of a solution can be enforced with the two following conditions that must be fulfilled by the corresponding interval graphs of a given arrangement:

- Any independant[1] set $S \subseteq G$ is i-admissible. That is $\sum_{v \in S} pr_i(v) \leq h_i$, where $pr_i(v)$ is the length of projection of component v_i on the i-th dimension and h_i is the length of the device in the same dimension.

- $\bigcap_{i=1}^{3} E_i = \emptyset$

The first restriction states that in a direction i, the total length of components placed on the device and pairwise overlapping in the perpendicular dimension to i should not exceed the length of the device in dimension i. To better understand this restriction, consider a perpendicular line to direction i that run through the device. All the components that are cut by the line pairwise overlap in the perpendicular direction to i. The total sum of their width in the

---

[1] $\forall v, w \in S, v$ overlaps with w in dimension $(3 - i), i \in \{1, 2\}$.

i direction should therefore not exceed the line segment which is in the device, i.e. the size of the device in dimension i. Placing many such lines at the different coordinate points in direction i of the device will help to capture all the restrictions stated by the first condition.

With the second restrictions, we avoid placements in which components overlap in all dimensions. A placement can only be valid if for any pair of components, at least 1-D must exists in which the two components do not overlap.

DEFINITION 5.6 (PACKING CLASS) *A 3-tuple of interval graphs that satisfy the two previous defined restriction is called a* packing class.

EXAMPLE 5.7 *The two-dimensional placement of figure 5.3 is a valid packing with the corresponding packing classes G_1 and G_2. In figure 5.4, we have an example of invalid packing for two reasons. First, part of the component v_5 is placed with a part out of the device and second, components v_3 and v_4 overlap in all directions.*

Using the pure packing classes as previously defined help to define the placement aspect of the temporal placement problem, without caring about the constraints in the dataflow graph. Because those constraints must be preserved in a valid temporal placement, a modification of the packing classes must be done to insure that a component depending on another one starts its execution only after the component on which it depends has completed its
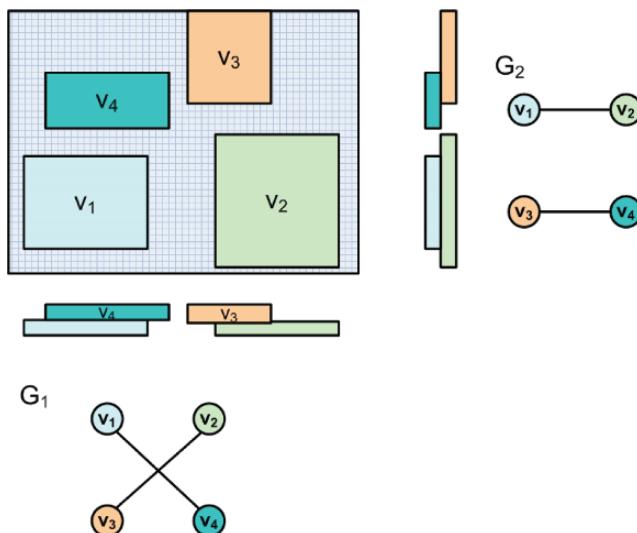

*Figure 5.3.* Valid two dimensional packing
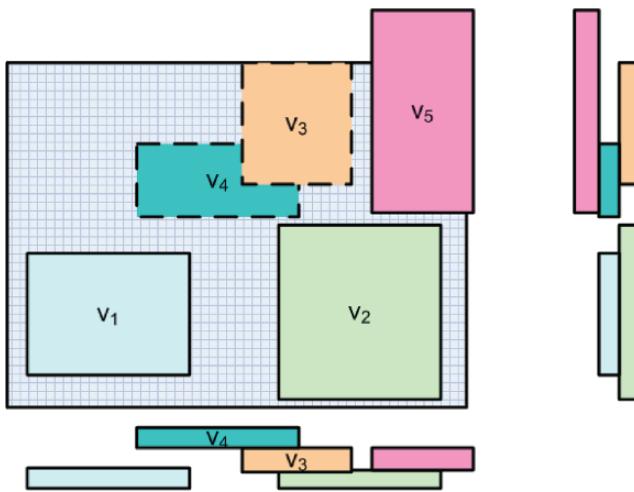
Figure 5.4.   A non valid two dimensional packing

execution. The modification that we present here are the orientation of the
packing classes.

1.2.2   Orientation of Packing Classes

An undirected edge $\{v_1, v_2\}$ in a component graph G_i corresponds to an
intersection between the projections of component v_1 and v_2 in the i-th dimen-
sion. However, it does not provide any information on the relative position of
component v_1 compared to v_2. The two nodes v_1 and v_2 can be compared ac-
cording to their placement in one of the three dimensions. In the x-dimension,
we have the relation 'left to' and 'upper' in the y-dimension. However, we
are only interested in the 3-rd dimension, i.e the time axis, where we define
the relation 'place after'. For each pair of components v_1 and v_2, either the
components v_1 *is placed after* v_2 $(p_t(v_1) + t_1 \leq p_t(v_2))$ or vice versa, if their
placement overlaps in the 3-rd dimension. We use this fact to define the com-
parability graph of an interval graph.

DEFINITION 5.8 (COMPARABILITY GRAPH) *Given   a   dataflow   graph*
$G = (V, E)$, *a reconfigurable device H and a packing of V into H.*
    The comparability graph of an interval graph $G_i = (V, E_i)$, $\forall i \in \{1, 2\}$
is the directed graph $G_i = (V, E_i)$, *where* $(v_k, v_l) \in E_i \iff v_l$ *is 'placed
after'* v_k *in 3-th dimension, i.e.* $(p_t(v_k) + t_k \leq p_t(v_l))$.

The relation *'place after'* defined by the comparability graph is a transitive
relation also known as *transitive orientation* that can be used for orienting the
packing classes.

DEFINITION 5.9 (PACKING CLASS ORIENTATION) *Given a dataflow graph $G = (V, E)$, a reconfigurable device H and a packing of V into H. The orientation of the packing class corresponding to the packing is defined through the comparability graph of the interval graph in the time dimension.*

The orientation of a packing is used to refine information about the placement of components on the time axis. This refinement is the information need to check if the precedence constraint is fulfilled. Precedence constraints do not play a role for the placement of component on the x and y directions. Therefore, a large variety of packing classes exist that lead to different 3-D placements with precedence constraints.

EXAMPLE 5.10 *Figure 5.5 provides a 3-D placement and its characterization through interval graphs, complement graphs and the orientation of the packing class associated to the placement. Shown is only the graph related to the dimensions x and y.*

Now that a good description of a solution is available, we have the possibility to check if a given arrangement of nodes forms a valid packing with some precedence constraints fulfilled. We only need a method to build arrangements on which we can apply the formulas previously described to check whether
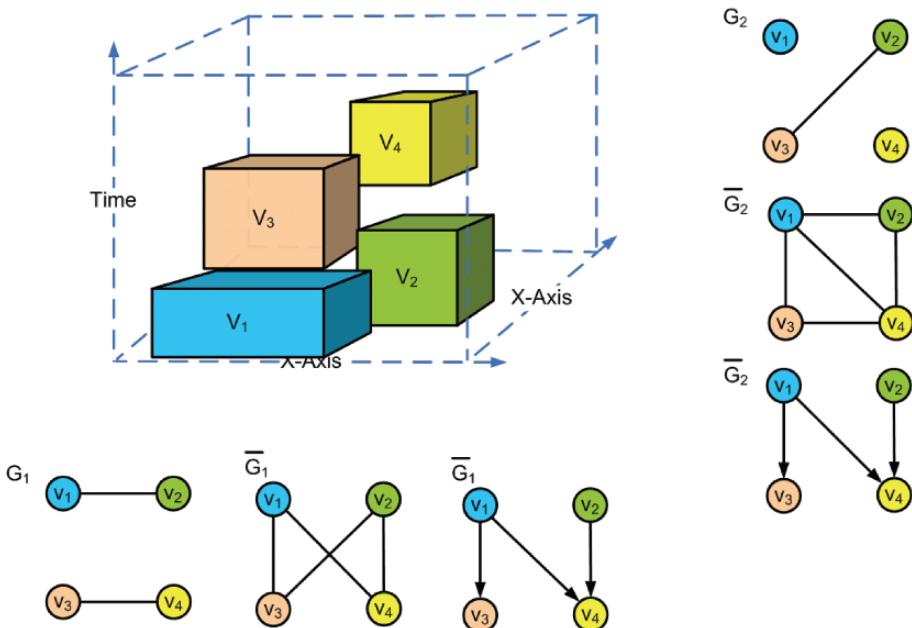

Figure 5.5. 3-D placement and corresponding interval graphs, complement graphs and oriented packing

or not the arrangement is a solution. Moreover, if we have a set of available solutions, we also need to extract the optimal one according to the optimality criteria. A solution or better said an arrangement that can be checked for validity can be constructed by arbitrarily fixing the edges of the different graphs as well as their orientations. The set of all possible arrangements that can be built in this constructive fashion called the *solution space*. This can be extremely large and very difficult to explored, if the size of the problem is big.

In [199] [82], a branch and bound strategy for iteratively prune the set of all possible arrangements is presented. The approach is an incremental that aim at constructing a tree from the root to the leaf. The root of the tree consists of the set of available nodes. The tree is constructed from the root to the leaf by inserting edges to the already existing graph. Two different insertions of edges on the same node lead to two different branches in the tree. In each step of the algorithm, a new edge is included into the graphs, and the validity of the resulting placement is checked. If the resulting placement will not be valid, then all possible arrangement in which the introduced edges exist are discarded from the search. Otherwise, a new edge is added to the constructed graph. Whenever a branch will not lead to valid leaf, the complete subtree resulting from the introduction of a new branch is discarded. This helps to avoid exhaustive and expensive checks on branches that do not lead to a solution. The formal characterization of a valid solution previously done provides at the same time a means to check if the full or partial arrangement constructed can lead to a solution. In [199] [82], the authors use the concept of induced cycle to identify and discard paths that will lead to non-valid solutions.

The method presented in this section target problems well specified at compiled-time with no change on the computational flow at run-time. In many systems, in particular real-time ones, the set of task as well as their interconnectivity is not known at compiled-time. Run-time or online placement possibility must be provided to deal with unpredictable event that creates tasks, which must be executed at run-time. The next sections deal with this issue.

2.   Online Temporal Placement

In Section 1, we presented the architecture of a dynamic run-time reconfigurable system (figure 3.1). The dynamic management of the device resources is done by two main components: a scheduler and a placer, each of which can be implemented as part of an operating system running on a processor. While the scheduler determines which task should be executed on the RPU, the placer tries to place the associated bounding box associated to a task on the device, thus allocating rectangular resource area on the device for the execution of that task. Whenever the placer receives a request from the scheduler to place a task and it is not successful in doing that, the scheduler must decide on what to do

with the task. It can later try again to run the task on the RPU or it can decide to let the task run with a lower speed on the CPU. In any case, not being able to satisfy a placement request creates additional delay in the execution of the program. To avoid this penalty, care should be taken on the design of the placer. The placer should be able to place as much tasks as possible on the device to avoid delay penalties. This goal can be reached if the amount of wasted resource is kept small. We first state the online placement problem and then present some solution approaches.

DEFINITION 5.11 (ONLINE PLACEMENT) *Given a reconfigurable processing unit H at time t with a given configuration C_t. Find an optimal position for a new incoming task v such that v does not overlap with any running component in the configuration C_t.*

As defined in the mathematics, the goal when solving an online problem does not only consist of providing an optimal method at a given time step, but a method that is optimal for a sequence of computation not fixed in advance. The method should be developed to cope with unknown parameters that may arise in the future. A non-optimal partial solution can be preferred to an optimal one at time t, if this partial sub-optimal solution contributes on a global optimal in the future. In this section, we limit the online definition to the stepwise optimization of a partial problem in a given time slot, without caring about the global optimal solution.

We present in the next sections two approaches for online placement of incoming components on the device. While the first approach manages the free space on the device by keeping track of all empty rectangles from which one will be selected to place the new module, the second one keeps only track of the occupied area on the device and try to place a new component in such a way that no overlap with a running module occurs.

3.   Managing the Device's Free Space with Empty Rectangles

One possibility to implement the placer is to always keep track of the free space on the device. On request to place a new component, a free location that can accommodate the new component is chosen and the module is placed at the selected location.

Bazargan et al. [18] proposed the used of a list of empty rectangles on the chip to capture the free locations, an approach that was used in computational geometry to solve rectangle layout problems as it is the case in glass cutting and paper cutting industries [116].

The main task to solve is to place a given set of rectangles, so-called pieces, in a given larger rectangular area, the slot sheet, such as to minimize the amount

of wasted space in the resulting layout. Once again, we start with definitions and then continue with solution approaches.

DEFINITION 5.12 (EMPTY RECTANGLE) *An* Empty Rectangle (ER) *is a rectangle that does not overlap a placed module on th chip. A* maximum empty rectangle (MER) *or* Largest Empty Rectangle (LER) *is an empty rectangle that is not included in any other rectangle than the device bounding box.*

EXAMPLE 5.13 *An illustration of empty rectangles is provided in figure 5.6. (E, D), (A,D) and (E,C) are example of empty rectangles. While (E, D) and (A,D) are maximal, (E,C) is not because it is included in (E,D). (E,F) is not an empty rectangle because it overlaps module v_2.*

On the basis of empty rectangle concept, a strategy to solved the online temporal placement problem was proposed in [18]. The method, which has the name *keeping all maximum empty rectangles (KAMER)*, permanently keeps track of all MERs. Whenever a request for placing a component v arrives, the list of MERs is searched for a rectangle that can accommodate v. Because it is possible to have many MERs in which v can fit, strategies such as the first-fit or best-fit are used to select one rectangle from the set of possible free rectangles. Once a rectangle is chosen, the candidate points that can be chosen as reference point for placing the new component are those that do not allow an overlap with the external part of the rectangle. In [18], the authors use the *bottom left* point as the reference one to place the component.
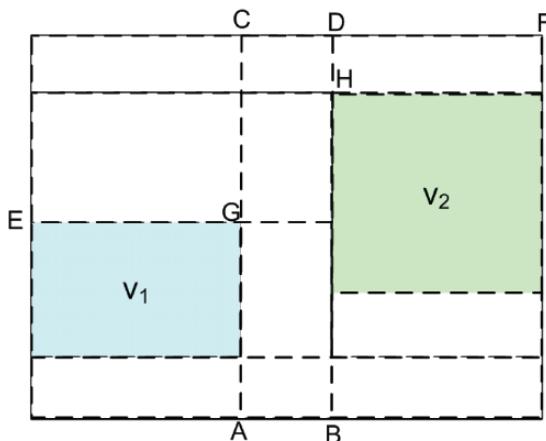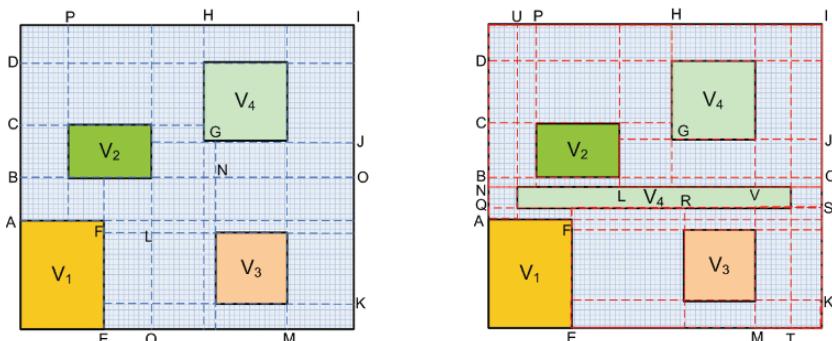

Figure 5.6. Various types of empty rectangles

The great advantage of the KAMER approach is that all the free space on the device is captured, thus providing a good quality[2]. This happens at the cost of efficiency in the computation time. As the following example shows, the number of empty rectangle does not grow linear with the number of components included. Whenever a new component is placed on large, depending on the configuration.

EXAMPLE 5.14 *Consider the configuration shown in figure 5.7(a) with 11 MERs ((D,I), (C,H), (A,P), (A,O), (F,O), (E,N), (Q,G), (Q,H), (E,K), (M,I), (L,J)) before the placement of a new component of the device. After the placement of module v_4, the number of MERs grows to 14 ((D,I), (C,H), (N,P), (A,U), (N,O), (A,S), (F,S), (E,R), (E,K), (L,H), (L,J), (M,S), (V,I), (T,I)), thus increasing the complexity by a factor more than the number of included modules.*

Also the number of free rectangles can be drastically reduced after the removal of a module running on the chip. The insertion of components on the chip as well as their later removal creates large fluctuations on the number of empty rectangles to be managed, thus increasing the complexity of the algorithm. In [116], the run-time of the free rectangle placement is shown to be $O(n^2)$.

To avoid the quadratic run-time of the KAMER, a simple heuristic was proposed in [18], with a lower quality and a linear run-time. The strategy consist of keeping only the *non-overlapping empty rectangles*, thus reducing the number of empty rectangles to be managed. This allows a linear run-time at the cost of the quality. The problem with this approach is that several

(a) The configuration contains 11 modules befog the insertion of a new task

(b) The number of modules grows to 14 after the insertion of a new task v_4

Figure 5.7. Increase of the number of MER through the insertion of a new component

---

[2]If a free space exist in which the module can be placed, then there must be a free rectangle that can accommodate that module, and therefore, the MER approach will find a placement.
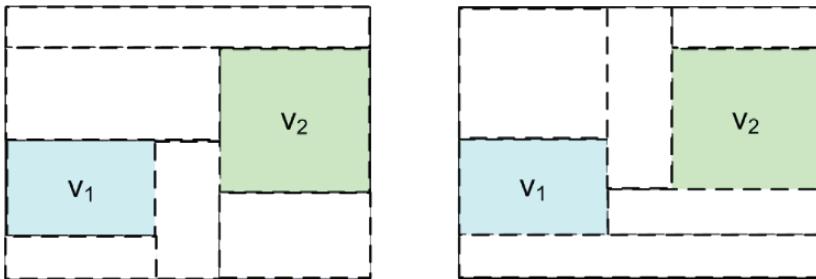
Figure 5.8.   Two different non-overlapping rectangles representations

non-overlapping rectangles representation of a configuration may exists lead-
ing to multiple possible choices in the representation as illustrated in figure 5.8.
Shown are two different representations of the free space with different sets of
non-overlapping free rectangles.

Because the non-overlapping empty rectangles are not necessary maximal, a
module may exists that could fit onto the device, but cannot be placed because
of a bad non-overlapping representation.

As shown in figure 5.9, whenever a new component v_1 is placed in a non-
overlapping rectangle, two possibilities exist to split that rectangle. A horizon-
tal split that is done using segment S_a and a vertical split using the segment S_b.

Choosing to select one the two split directions may have a negative impact
on the placement of the next components. Assuming for example that the al-
gorithm keeps selecting the horizontal split, after the placement of module v_1
(figure 5.9), the free rectangles left are (A,D) and (C,E). Assume now that a
new module, whose width is the same as (C,E) but with a height slightly bigger
than that of (C,E) is chosen. The algorithm will not be able to place the compo-
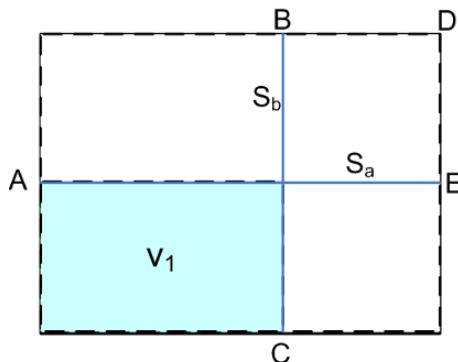

Figure 5.9.   Splitting alternatives after new insertion

nent, although enough free space is available on the chip to accommodate the new module. Bazargan et al. proposed several strategies for choosing one split direction with the goal is to favour those splits that create quadratic space on the chip. To increase the quality of the non-overlapping free rectangle heuristic, a strategy is proposed in [197] that simply consist of delaying the split decision for a number of steps later. This may allow a bad choice that could have been done earlier to avoid.

While the KAMER algorithm always find a rectangle to place a new component, if one exists, the position to place the component within the rectangle must be selected from a set of points whose number is the area of the rectangle in worst case. An optimal algorithm must choose one point from the set of possible points according to some optimization criteria. In [116], the bottom left position is chosen. Because no relation exists between the rectangle to be placed and those already placed, an arbitrary position can be chosen for the new rectangle. In reconfigurable devices where the communication between pairs of tasks on one side and between a task and the device boundary on the other side plays an important role, components should be placed in such a way that the communication will be efficiently realized. If a position in the middle of the selected rectangle is better adapted for optimizing the goal seeked, then the component should be placed there, no mater if the number of empty rectangle increases. An optimal algorithm should consider any single point where the placement is possible and develop a strategy to choose the best position. This approach is followed by the next algorithms that we present. Instead of managing the free space, the strategy consists of managing the occupied space on the device and use the set of running components for computing the best position to place the incoming component.

4. Managing the Device's Occupied Space

The observation that has guided the development of this approach is that the number of placed modules on the device grows linearly contrary to the number of free rectangles that has a quadratic grows. Each insertion of a new component on the chip increases the number of placed components by only one, the same as any removal of a component decreases the list by only one. By managing the number of placed modules, we do not face large fluctuations in the list of modules like it is the case with empty rectangles. It is therefore possible to develop a linear time optimal algorithm for the online placement problem. With this, the temporal placement can be splitted into two subproblems as first presented in [3].

1 Identify the set of potential sites to place the new component.

2 Select the best site to place the component according to a set of given criteria.

Assuming that a set of possible placement sites is identified, several criteria can be used to choose between the feasible positions. Here, we consider the communication cost between the task and its environment as the objective to be minimized. The connections between two different components as well as those between modules on the device and the device boundary are of great importance. While the first one allows two modules to communicate together, the second one allows a modules within the device to communicate with a module out of the device.

The straightforward way to solve this subproblem is to use a brute force Algorithm. For each new component c to be placed, the brute force algorithm solves the first subproblem by scanning all the positions on the device. For each position $p = (x_p, y_p)$, it checks if an overlapping will occur between c and a placed module, if the component c is placed at location p.

Having solved subproblem through the scanning of all possible positions, the optimal placement position is computed from the set of feasible positions by computing the placement cost for each of the locations found in the first step and then select the best one as the optimal solution.

The Brute force requires $O(H \times W \times n)$ time to solve subproblem 1. H being the height, W the width of the reconfigurable device and n is the number of running tasks on the hardware. This approach is not practicable for large reconfigurable devices.

Without loss of generality, we will consider that components are placed relatively to their lower left positions. Later, we will use the middle of the device as reference point. We next provide some definitions that are important to understand the placement strategy explained in this section.

DEFINITION 5.15 (IPR RELATIVE TO PLACED MODULES) *For a new component v to be placed on the device and a placed component v', the Impossible Placement Region (IPR) $I_{v'}(v)$ of v relative to v' is the region on the chip, where v cannot be placed without overlapping with v'.*

For a set V' of placed components, the impossible placement region $I_{V'}(v)$ of v relative to the set V' is the region on the chip, where v cannot be placed without overlapping with an element of V': $I_{V'}(v) = \bigcup_{v' \in V'} I_{v'}(v)$.

Besides the impossible placement region relative to components of the device, an impossible placement region relative to the device itself exists and is defined as follow:

DEFINITION 5.16 (IPR RELATIVE TO THE DEVICE) *The   Impossible Placement Region $I_H(v)$ of a module v relative to a reconfigurable device H is the region of the device where v cannot be placed without overlapping with the external area of the device.*

Having defined the impossible placement region relative to the device and that relative to other components on the chip, we can now define the overall impossible placement region of a component on a device with a running set of components.

DEFINITION 5.17 (IMPOSSIBLE AND POSSIBLE PLACEMENT REGIONS) *The impossible placement region IPR $I(v)$ of a new component v on a reconfigurable device H is the region on the device, where v is not allowed to be placed, if the placement has to be valid. $I(v) = I_{V'}(v) \cup I_H(v)$, where V' is the set of all the components running on the chip*

The possible placement region (PPR) $P(v)$ of a module v is obtained by subtracting the IPR of the module from device from the device area. If L is the set of all locations on the device, then $P(v) = L - I(v)$.

As illustrated in figure 5.10, for a new component v with height h_v and width w_v to be placed on the device and a placed component v', the IPR $I_{v'}(v)$ of v relative to v' can be computed by first defining a left-side margin with size $w_v - 1$ and a bottom margin with size $h_v - 1$ for the component v'.

The IPR of v relative to v' is the sum of the augmented margin and the area of v'.

The computation of the IPR relative to the device is slightly different from that of a component. Instead of computing a left-side and a bottom margin, we compute a right-side and an upper margin with. The right-side margin has the
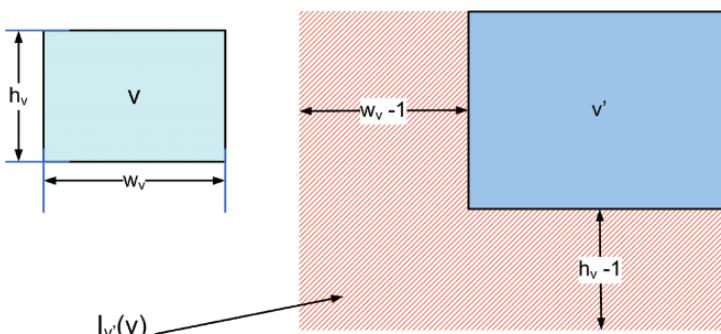

*Figure 5.10.* IPR of a new module v relative to a placed module v'
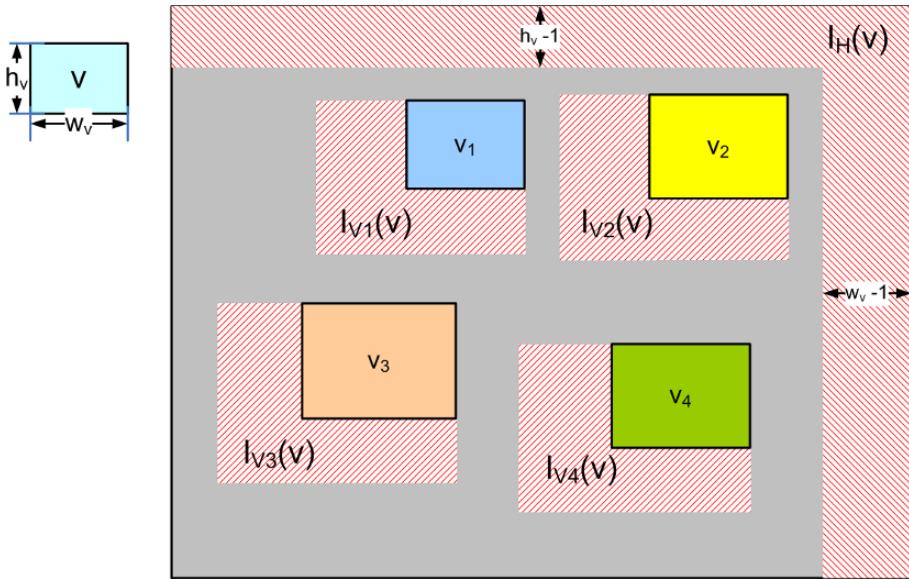
Figure 5.11.   Impossible and possible placement region of a component v prior to its insertion

width $w_v - 1$, whereas the upper margin has the height $h_v - 1$. Figure 5.11 shows the impossible placement region of a component v on a device with four running components v_1, v_2, v_3 and v_4.

Computing the IRP of a given module can be done in linear time $O(n)$, where n is the number of components currently running on the chip. For each component, a constant amount of operations is required to identify the IPR relative to that component.

Now that we know how to compute the region (set of points) where the component can be placed, the next sub-problem consists of choosing the best location for the new component. The position where the new component must be placed should be a feasible position that minimizes the communication cost. A simple approach consists of scanning all the possible placement positions and to compute the communication cost for each position and then selecting the optimal one. This straightforward but inefficient approach requires $O(|PPR| * n)$ where n is the number of placed components and $|PPR|$ is the size (number of point) of the possible placement region. This can be very large according to the current configuration of the device.

Alternatively, we can first compute the point p_{opt} that gives us the optimal placement cost. If p_{opt} is located within the PPR of v, then we have the solution. Otherwise, the optimal position is not in the PPR and we should choose

the closest point to p_{opt} which is located in the PPR and select it as the optimal placement position.

The definition of the optimal point is done according to the communication cost that can be minimized by placing the new component closed to other components with which it communicates. If the component to be placed has off-chip connections, then the boundary of the device that connects to the component is assumed to be a placed module at the corresponding location. The communication cost that we also called *routing cost* is therefore formally defined in terms of weighted distance between modules and device boundary. We consider that routing cost between two modules is defined by the Euclid distance. Later we will investigate the Manhattan distance that is better adapted for the routing cost on a given chip.

DEFINITION 5.18 (ROUTING COST) *For two modules v_i and v_j, we define the routing cost between modules v_i and v_j as follows:*

$$Routing\ Cost(R_{ij}) =$$
$$[(x_j + \frac{w_j}{2} - x_i - \frac{w_i}{2})^2 + (y_j + \frac{h_j}{2} - y_i - \frac{h_i}{2})^2] \times w_{ij}$$

In other words, the routing cost between two modules is the weighted Euclid distance between the two modules. To compute the routing cost, we use the center point of the components instead of using the bottom left corner point as the reference point. For module v_i, this is defined by the pair $(x_i + w_i/2, y + h_i/2)$ where (x_i, y_i) define the lower left position of v_i, w_i the width and h_i the height of v_i . If there is no communication between two modules v_i, v_j, then the routing cost between the two modules is zero.

With $(n-1)$ modules placed on the device and an n-th module to be placed, the routing cost is the sum of the routing cost between the n-th module and each of the $n-1$ modules running on the chip. The optimal point is therefore the one that provides a solution to equation 4.1:

$$min(\sum_{i=1}^{n-1}\{[(x_n + \frac{w_n}{2} - x_i - \frac{w_i}{2})^2 + (y_n + \frac{h_n}{2} - y_i - \frac{h_i}{2})^2] \times w_{in}\}) \quad (4.1)$$

In Equation 4.1, x_n and y_n are variables and other parameters are fixed. Because x_n and y_n are positive and independent from each other, we can replace equation 4.1 by the two equations 4.2 and 4.3.

$$min\{\sum_{i=1}^{n-1}[(x_n + \frac{w_n}{2} - x_i - \frac{w_i}{2})^2 \times w_{in}]\} \quad (4.2)$$

$$min\{\sum_{i=1}^{n-1}[(y_n + \frac{h_n}{2} - y_i - \frac{h_i}{2})^2 \times w_{in}]\} \quad (4.3)$$

The minimums can be computed through the partial derivative of the sums in equations 4.2 and 4.3. We therefore have for x_n

$$\frac{\partial\{\sum_{i=1}^{n-1}[(x_n + \frac{w_n}{2} - x_i - \frac{w_i}{2})^2 \times w_{in}]\}}{\partial x_n} = 0 \qquad (4.4)$$

This leads to the optimal value:

$$x_n = \frac{\sum_{i=1}^{n-1} w_{in}(x_i + \frac{w_i}{2} - \frac{w_n}{2})}{\sum_{i=1}^{n-1} w_{in}} \qquad (4.5)$$

In a similar way, we can compute the optimal value for y_n:

$$y_n = \frac{\sum_{i=1}^{n-1} w_{in}(y_i + \frac{h_i}{2} - \frac{h_n}{2})}{\sum_{i=1}^{n-1} w_{in}} \qquad (4.6)$$

The computation of this optimal weighted sum of Euclid distance is fast. However, the computed point may fall into the IPR of the component. If this is the case, we need a strategy to quickly move out of the IPR and select the closest feasible point to optimal. As shown in figure 5.12, there are four *nearest possible points (NPP)* associated with a component in whose IPR the optimal point falls. Four comparisons are necessary for selecting the closest one.

In the worst case, we may face the situation, where several IPRs overlap such that moving out of the IPR related to a component bring us in the next IPR of another component. In this case, we simple keep moving out of the IPR until we get a point in the PPR. However, the price of doing such a successive move can be very high. A situation could be constructed such that we have to go through all IPRs, thus creating a quadratic run-time $O(n^2)$. Because we need $O(n)$ time to compute all the IPRs and $O(n^2)$ in the worst case for moving out of all IPRs if the computed optimal point falls within one of them, the run-time of the algorithm is quadratic.


●   Optimal point

X   Near possible point
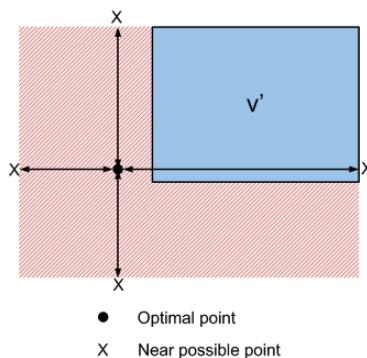
Figure 5.12.   Nearest possible feasible point of an optimal location that falls within the IPR

Algorithm 11 provides a pseudocode for the approach presented here.

Algorithm 11 Pseudocode of the online placement algorithm

1: Compute optimal point (new component)
2: **if** optimal point is feasible **then**
3:     place the component at the optimal point
4: **else**
5:     Find the 4 near points outside the actual IPR, in which the selected point is located
6:     Insert these points into the optimal points list
7:     Select the closest point to the optimal point from the near points list
8:     **if** the selected point is feasible **then**
9:       place the component at the optimal point
10:     **else**
11:       remove it from the list, and go to 5
12:     **end if**
13: **end if**

The implementation of this method can be done using a linked list of size $O(n)$ (for n running tasks) to store the placed and running modules on the reconfigurable device.

A second structure that can be used is a 2-D matrix with the same size as that of the reconfigurable device. This matrix represents the state of the device. An entry in the matrix defines the value of the point with the same coordinate on the device. Whenever a point on the device is occupied by a module, the corresponding element in the matrix will have a pointer to the related module. Otherwise, the point in the matrix shows its emptiness. Also for identifying PPR, the effects of extending and deleting margins will be applied on this matrix. Using the matrix allows to access each element and obtain its state in just one computing step.

For finding optimal or near optimal point for placing a new module, a linked list for saving near optimal points can be used. The next element to be removed from the list is always the one closest to the optimal point. If the removed point is feasible (this is tested by checking the corresponding entry in the matrix), then we have the NPP, otherwise the selected point is occupied by a running module or its margin. In this case, the nearest points, out of the border of that running module are inserted into the list.

EXAMPLE 5.19 *Figure 5.13 shows an example of recursive move out of the IPRs. The optimal point falls in an IPR relative to module A. The four nearest points inserted in the list are $1, 2, 3$ and 4. The closet point to the optimal is 4 that is next selected but falls in the IPR of module B.*
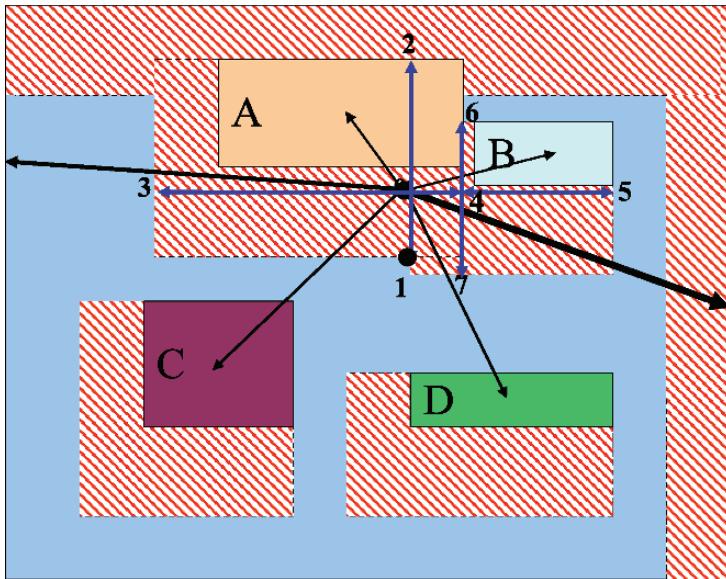
Figure 5.13.   Moving out of consecutive overlapping IPRs

The points 5, 6 *and* 7 *are inserted into the list. The closest point to the optimum after the removal of the point* 4 *is the point* 1 *that is selected and maintained as solution, because it is in the PPR.*

The approach presented here has the drawback of moving from IPR to IPR until a valid placement point is found. In the normal case, we need only a few number of steps. However, the worst case still remains and in the case, it happens, the penalty is a quadratic run-time. A better characterization of the total set of IPRs can help to improve the efficiency as well as the quality of the method. This is the topic of the next section.

4.1   Using the Contour of Rectangles

The main drawback of the approach previously presented is the delay that may arise, when moving a point out of the IPRs. This is because the IPRs of all components are considered separately. If we could merge the complete IPRs of that component in only one region, then moving out of an occupied region can be done in constant time, thus improving the run-time of the algorithm. Several IPRs can be merged in only by computing the contour of the IPRs.

Like in the previous section, the approach presented here is based on the observation that the occupied space consists of very simple geometric objects,

namely n placed rectangular modules. The difference is in the use of the Manhattan Distance metric for capturing the exact routing costs.

The method was first presented in [2] as extension of the work in [3]. The management of the occupied space is done through a modification of the well-known algorithm CONTOUROFUNIONOFRECTANGLES (CUR) [106] [147] [179], for finding the contour of a union of axis-parallel rectangles. As the number of contour segments is linear in n, a running time of $O(n \log n)$ can be achieved. Note that the method does *not* require the contour to be connected, it works even on a set of disjoint contours.

In contrast to the previous section and without loss of generality, the impossible placement region of a component v relative to v' is now computed with the reference to the middle point of v, by blowing up the module v' by half the width and half the height of v. The impossible placement region of v relative to the device H is however computed by shrinking the device half the width and half the height of v.

As defined in the previous section, the new module v is reduced to a point p_v; thus, the original problem of finding free space to place a rectangle becomes the problem of finding a free location to place a point.

EXAMPLE 5.20 *In figure 5.14, the result of the transformation for a new component that must be placed is shown. All points out of the impossible placement regions are feasible placement locations from which the optimal one must be selected.*
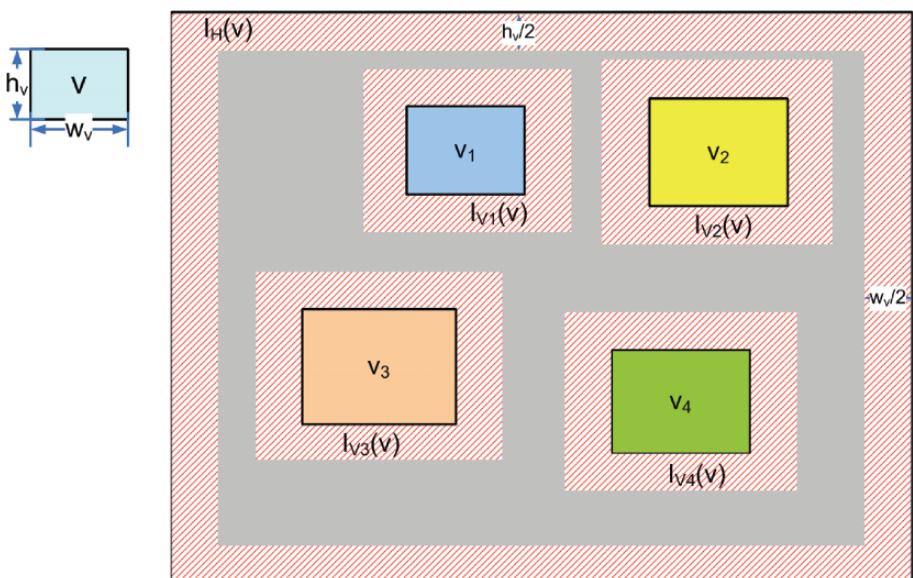

Figure 5.14. Expanding existing modules and shrinking chip area and the new

Among all feasible locations, the points on the contour of the occupied space, i.e those at the boundary of the free space and the occupied space, form a set from which the nearest to optimal point will be selected, if the optimal point falls in an occupied region. This helps to avoid the recursive jumping out of the IPRs of the components as presented in the previous section. The computation of the contour is therefore at the center of our interest.

4.1.1   Computing the Contour of the Occupied Space

In general finding, the contour of a set of n axis-aligned rectangles can be done in $O(n \log n + s)$ by using the CUR algorithm as described in [106] [147] [179]. Here s is the complexity of the resulting contour. The algorithm presented in [2] is not simply an implementation of CUR. There are a few subtleties that must be considered. All differences stem from the above-mentioned fact that the points on the contour are feasible locations. As a consequence, the algorithm has to find free space of height and width 0 as shown on figure 5.15. The CUR returns an exact contour of a set of placed rectangle, whereas the modified algorithm return the exact set of feasible location. Shown is a set of rectangles (left), representing expanded modules (dashed, light). At the center, we have the contour as returned by CUR and the resulting feasible space for placing the center point of the new module (dark, together with the boundary). The right part of the figure shows the correct feasible positions for placing the center point of the new module as returned by the modified CUR (dark, with boundary). The difference is a short segment representing the boundary of two IPR of the two components.
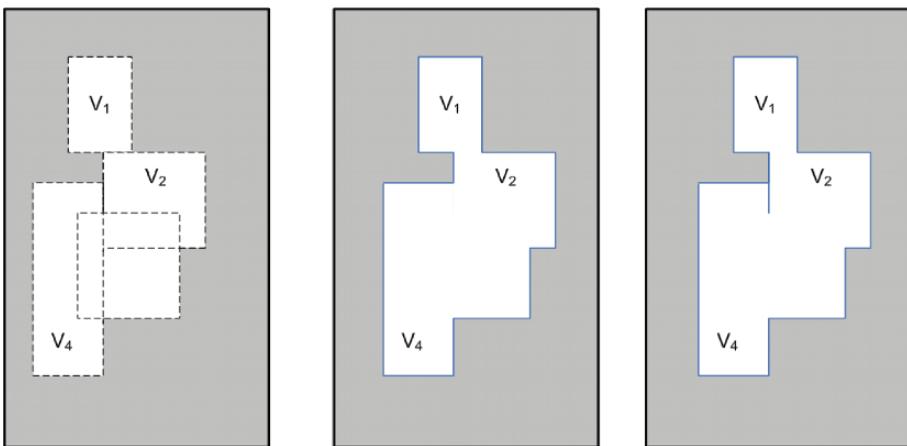

Figure 5.15.   Characterization of IPR of a given component: The set of contours (left), the contour returned by the modified CUR (middle), the contour returned by the CUR (right)

In the following, we will describe CUR and the modification done in [2].

The building block of CUR is an algorithmic technique from computational geometry called *plane sweep* and a data structure called *segment tree*. For an in-depth introduction to both, refer to [66].

A *plane-sweep algorithm* is an algorithm that scans the plane and a set of objects in it: It consists of moving an axis-parallel line in an orthogonal direction across the plane and keep track of the structure of the intersection with the set of objects. The key observation is to notice that updates to this structure only occur at a discrete set of critical positions called *events*. By pre-sorting these events (in time $\Theta(n \log n)$), only the updates have to be performed, which can be done efficiently for all events by using an appropriate data structure. For our purposes, such a data structure is a *segment tree*: This is a balanced binary tree for dynamically storing a set of n intervals. The number of endpoints of these intervals must be known at construction time. Because it is bounded by $2n$, the segment tree can be constructed in $O(n)$. Insertion and deletion of intervals can be done in $O(\log n)$. For more details on segment trees, refer to [24] [22, 179] [23].

One has to be careful when constructing the segment tree. To find free space of height and width 0, we have to make sure that two modules starting or ending on the same coordinate are separated by an elementary interval in the segment tree. This can be done by disturbing the top left corner of each module by a sufficiently small value $\epsilon > 0$.

The algorithm uses two plane sweeps: one horizontal sweep that discovers all the vertical contour segments and one vertical sweep that finds all horizontal segments. As the horizontal and the vertical sweeps differ only in the initialization, only the horizontal one will be described.

For the horizontal sweep, we construct a list L of $2n$ quadruples, denoted by (p_i, t_i, b_i, e_i): for each module $v_i' \in V'$, where V' represents the set of extended modules that form the IPRs of the components on the chip, we add two elements to L — one for the left side $(x_i', Open, y_i', y_i' + h_i')$ and one for the right side $(x_i' + w_i', Close, y_i', y_i' + h_i')$. This list is sorted lexicographically and we assume that $Close < Open$.

In the sweep, we process all $2n$ elements of L. In case of an $Open$ event, the corresponding contour points are retrieved from the segment tree and the segment $[y_j', y_j' + h_j']$ is added to the tree. For a $Close$ event, the segment $[y_i', y_i' + h_i']$ is removed from the tree, and the corresponding contour points are retrieved. Another vertical plane sweep is necessary to discover all horizontal segments.

4.2   Routing-Conscious Placement

After describing how to find *all feasible* locations to place a new module, we turn to find an *optimal* location, which minimizes the weighted communication between the modules. Instead of using the Euclid distance as in the previous

section, the Manhattan Distance is used. The Euclid distance between two components defines the shortest distance between two points. However, this shortest distance may be a diagonal line that cannot be realized as signal on the chip. The Manhattan Distance captures the smallest routing distance between two components on the chip. Such a route is made only upon vertical and horizontal segments weighted by width of the communication segments.

This can still be achieved in time $\Theta(n \log n)$, making use of local optimality properties, the occupied space manager, and another application of plane sweep techniques.

4.2.1   Communication Cost

Given a reconfigurable device H and a component v, the objective of the placer is to find a point in free space that minimizes communication cost for the new module v.

As in the previous section, the communication cost of an additional module m is the sum of the Manhattan distance between the center point of m and the center points of the modules that communicate with m.

Also, we consider communication with the chip boundary as indicated in Figure 5.16. We therefore have to consider the set $C = \{(x_1, y_1), \ldots, (x_k, y_k)\}$ of demand points for communication. We assume that the number k of connections to be established with module m is linear in n, i.e the new component is connected to a running component by a maximum of one link. Obviously, the width w_{ij} of the communication link between modules i and j must be considered, thus, we get the objective function

$$\min\{c(x_i, y_i) = \sum_{j=1}^{k} w_{ij}\|(x_i, y_i) - (x_j, y_j)\|_1\}$$
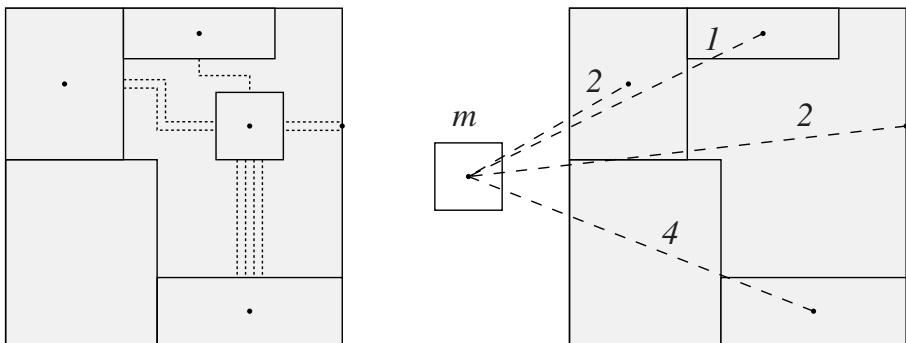

Figure 5.16.   Placement of a component on the chip (left) guided by the communication with its environment (right)

Because we are dealing with the Manhattan metric, this can be reformulated to

$$c(x,y) = c^x(x) + c^y(y)$$

with

$$c^x(x) = \sum_{i=1}^{k} w_{ij}|x_i - x|,$$

and

$$c^y(y) = \sum_{i=1}^{k} w_{ij}|y_i - y|.$$

If we would allow v_i to be placed anywhere on the chip, this can be reformulated to

$$\min\{c^x(x_i) : x_i \in [\frac{l_i}{2}, H_l - \frac{l_i}{2}]\}+$$
$$\min\{c^y(y_i) : y_j \in [\frac{h_i}{2}, H_h - \frac{h_i}{2}]\}.$$

As a consequence, we can consider two separate minimization problems, one for each coordinate. If we ignore feasibility, both minima are attained in the respective weighted medians, so they can be computed in linear time [27]; as we already sort the coordinates for performing plane sweep, this running time is not critical, so we may as well use a trivial method. Note that only medians satisfy unconstrained local optimality, as the gradients for c^x and c^y are simply

$$\nabla c^x(x) = \sum_{x_i < x} w_{ij} - \sum_{x_i > x} w_{ij},$$

the sum of the required bandwidths to the left minus the sum of the required bandwidths to the right and

$$\nabla c^y(y) = \sum_{y_i < y} w_{ij} - \sum_{y_i > y} w_{ij},$$

the sum of the required bandwidths to the bottom minus the sum of the required bandwidths to the top.

4.2.2 Local Optimality

The optimal point that produces the minimal routing cost with the Manhattan distance is called the median (figure 5.17). It is the global optimum that may falls in the impossible placement region. If it is in the occupied space, there are only two other types of points where the nearest possible point to optimum could be located.
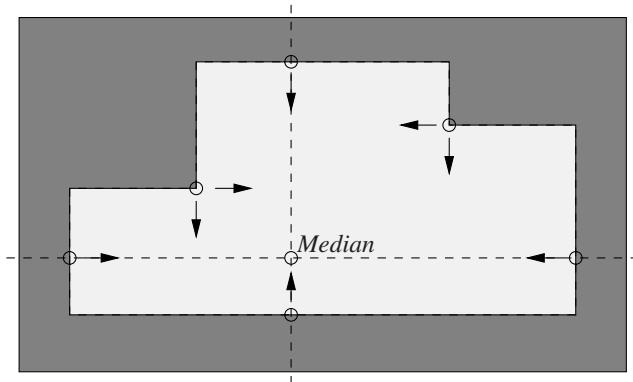

Figure 5.17. Computation of the union of contours. The point on the boundary represent the potentially moves from the median out of the IPR.

All points of the first type can be found by intersecting the contour of the occupied space with the median axes $l_x = \{(x_{med}, y) : y \in [0, H_l]\}$ and $l_y = \{(x, y_{med}) : x \in [0, H_h]\}$. In these points, one of the gradients ∇c^x and ∇c^y vanishes. We cannot move in the direction of a better solution because that way is blocked by either a vertical or a horizontal segment of the contour.

The second type of points are some of the vertices of the contour. These points are the intersections of horizontal and vertical segments forming an interior angle of $\frac{\pi}{2}$ pointing in the direction of the median. In these points, neither of the gradients vanishes. Either of the directions indicated by the gradient are blocked by contour segments.

By simply inspecting all the local optima, one finds the one closest to the global optimum. In the next subsection, we describe how this can be done efficiently.

4.2.3 Algorithm for Global Optimality

The occupied space manager described in the previous section computes the contour of the occupied space in $O(n \log n)$. A simple algorithm that finds the optimal point to place a new module v would compute the median and check its feasibility; if the outcome is positive, we have found the optimum. Otherwise, we need to check communication cost for all other possible local minima,

i.e., for every vertex of the contour and every intersection point of the contour with one of the median axes. Let L denote this set of points. Computing the communication cost for a single point takes $O(n)$, so evaluating all objective values in a brute-force manner would take $O(n^2)$ time. However, by means of two more plane sweeps, we can achieve a complexity of $O(n \log n)$.

For this purpose, we observed that communication cost for the x- and y-coordinate of the contour segments can be computed separately, then add the precomputed values for every point of L. The crucial step is to use the fact that we only need to compute the communication cost for the leftmost x-coordinate and for the bottommost y-coordinate; the other values can be obtained by doing appropriate fast updates during the plane sweep.

5. Conclusion

This chapter has provided some of the most important approaches for temporal placements. While very solid mathematical backgrounds were developed and simulated, a physical investigation on reconfigurable chips has failed so far. This is mainly due to the lack of devices providing partial reconfiguration facilities, a precondition for the implementation of temporal placement algorithms. One and almost the only device type to provide partial reconfiguration is very limited and does not allows for evaluating the large set of available algorithms. Besides the difficulty to designed for partial reconfiguration, many restriction are done on the use of the device resources during and after the reconfiguration. Some hope were placed on coarse-grained reconfigurable devices. However, they have failed so far to take-off.

Chapter 6

ONLINE COMMUNICATION

Several methods were presented in the last chapter for the placement of components at run-time on a reconfigurable device. The last two ones use the communication costs between a component to be placed and its environments, i.e. the set of modules on the chip and out of the chip that exchange data with this component, as a mean to select the best position for the new component. While this communication costs, defined as the average minimum distance of the new component to its neighbour is useful to guide the placement process, it does not tell us how data should be exchanged among the different components on the chip. In this chapter, we provide some answers to this question by presenting some approaches to enable the communication at run-time between modules on the chip. The approaches can be classified on different categories, depending on the way the communication is realized. We can cite the direct interconnection, the communication over third party, bus-based communication, the circuit-switching approach and network on chip-oriented communication.

1.   Direct Communication

Direct communication paradigm allows modules placed on the chip to communicate using dedicated physical channels, configured at compile-time. The configuration of the channels remains until the next full reconfiguration of the device. A configuration defines the set of physical lines to be used, their direction, their bandwidth and speed as well as the terminal, i.e the components that are connected by the lines. Components must be designed and placed on the device in such a way that their ports can be connected to the predefined terminals. Feedthrough channels must also be available in each component to allow signal used by modules aside the component to cross the components.
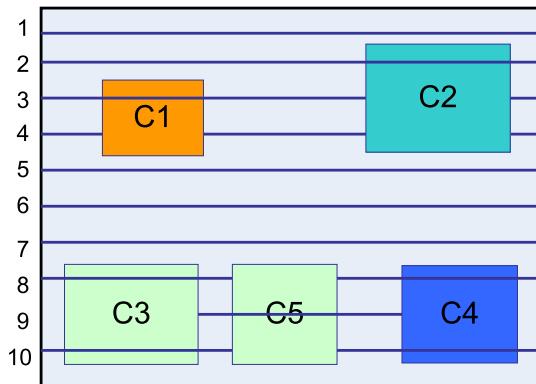
Figure 6.1.   Direct communication between placed modules on a reconfigurable device

EXAMPLE 6.1 *The configuration of figure 6.1 provides an example of 1-D communication using direct lines. For this purpose, a set of 10 predefine channels is fixed and the component must adapt their position and direction to make use of the channels. The physical channels 1, 5, 6 and 7 are not used. Line 3 is used by module C2 for connection with the device pins on the left and right sides. It is fed through component C1 that provides the necessary channel for the signal to cross. Line 4 is used by the two components C1 and C2 for a direct communication. Additionally, the two components use the same line to access the device pins. Lines 8 and 10 are used by component C5 to access the device pins on the left and right sides. They cross components C3 and C4. Line 9 is used to connect the components C3 and C4 and runs through component C5.*

The main disadvantage of this approach is the restriction imposed on the design of components. For each component, dedicated channels must be foreseen to allow signals that are not used by this component in its placement location to cross. This increases the amount of resources needed to implement the component. Also, the placement algorithm must deal with additional restrictions like the availability of signals in a given location. This increases its complexity and makes the approach only possible for an offline temporal placement, where all the configurations can be defined and implemented at compile-time.

2.   Communication Over Third Party

Communication over third party is used for example by Brebner in [37] and Walder et al. in [211] [196] [212]. A central component exists that behaves as a message reflector. Each message is first sent to the central module, which forwards it to the destination. Brebner in [37] uses this approach to allow the communication between a reconfigurable module connected to a processor through

a bus and a user program running on the host processor. The module inputs and outputs are controlled by registers that are mapped into the address space of the processor. This approach can be used not only to allow the communication between a reconfigurable module and a user program but also between several reconfigurable modules connected together through a bus. The system is controlled by an operating system, whose role is to manage the device resources, control the reconfiguration process and allow the communication to happen between the components temporally placed on, or removed from the device. All modules willing to send a message must first copy those messages in their sending register. Thereafter, the operating system copies the message from those registers to the input register of the destination module. In [211] [212], the central module is not an operating system running on a separate processor, but a set of fixed resources on the device. It provides dedicated channels to access peripheral devices and also to connect direct neighbour modules. The communication between non-adjacent modules is done through fixed resources implemented on the device.

3. Bus-based Communication

The communication between the reconfigurable modules on a given device can also be done using a common bus. To avoid the bus resources to be destroyed at run-time by components dynamically placed on the device, predefined slots must be available, where the modules can be placed at run-time, such that no alteration of the bus is possible. On the predefined slots, connection ports must be available to dynamically attach the placed component to the bus. While the predefinition of locations where to place components allows a simplicity of the placement algorithms, it is not flexible at all. Using a common bus reduces the amount of resources needed in the system because only one medium is required for all component. However, the additional delay increased by the bus arbitration can drastically affect the performance of the system. The approaches of Walder et al. in [211] [212] as well as that of Brebner are both based on restricted bus in which no arbitration module is required to manage the bus access.

4. Circuit Switching

Introduced in the 1980s under the name reconfigurable massively parallel computers [146], [205], circuit switching is the art of dynamically establishing a connection between two processing elements (PE) at run-time, using a set of physical lines connected by switches. The system consists of a set of processing elements arranged in a mesh. Switches are available at the column and line intersection to allow a longer connection using the vertical and horizontal lines at an intersection point. In this way, two arbitrary processing elements

can be dynamically connected at run-time by dynamically setting the switches on the path from the first processor to the second one. Once the connection is established, the data are transmitted from the source PE to the destination in just one clock.

Circuit routing was experienced in several systems such as the YUPPIE [146] in the 1980s. It is available today in many systems such as the PACT-XPP device [176]. Also the connection mechanism in FPGA follows the same paradigm. Although dedicated lines exist in some devices to allow connection between remote modules on the device, longer connections are normally established using the switch matrices to connect segmented lines in other to realize the long communication path.

Despite its application in fine-grained parallel image computing systems, where it helped to dynamically change the topology of a parallel computer to accommodate the best computation structure of an application under computation, using the circuit switching in reconfigurable devices presents some drawbacks:

- **Long communication delay**: This will happen, if the connection between two processing elements has to go through many other processors. The number of switches on the communication path therefore increases, thus increasing communication delay and therefore slowing down the clock.

- **Dynamic computation of routes**: In dynamically changing environment, the communication need between components changes with the placement of components. Computing new routes at run-time is very time-consuming, and therefore, the performance of the overall computation may be drastically affected by the run-time routing.

- **Exclusive use of chip space**: Because of the complexity of the synthesis algorithm, components that will be downloaded later onto the reconfigurable device are implemented at compile-time. The synthesis process constrains a module on a given area of the device, where it uses all the resources, i.e. the processing elements and the interconnects in that area. This has the consequence that the placement of a module at run-time in a region, where a route is running, will destroy the route, because the connection used by the route will be assigned to the component. To avoid this, we can place components only at locations where no route will destroy them, a restriction that will however increase the chip fragmentation. Consider the placement of figure 6.2 with three routes to connect PE1 to PE2, PE3 to PE4 and PE5 to PE6. For a new component that needs to use four consecutive processing elements in a quadratic surface, the region where the routes are implemented must be prohibited. No region that can accommodate the new component is therefore left on the device, and the component must be rejected, although enough resources are available on the device.
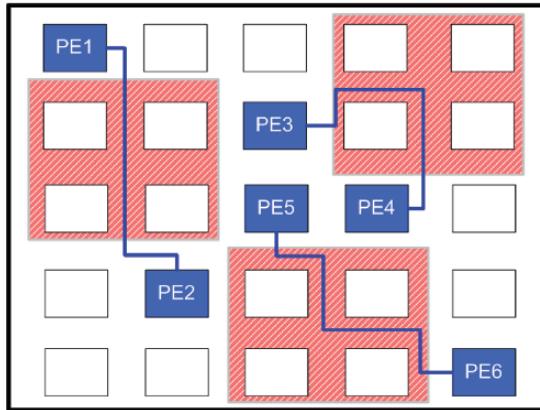
Figure 6.2. Drawback of circuit switching in temporal placement Placing a component using 4 PEs will not be possible, although enough free resources are available

The previous example has shown how circuit switching might increase the device fragmentation in devices allowing a 2-D placement. In device allowing only a 1-D placement like it is the case of Xilinx VirtexII FPGAs, which are only column wise reconfigurable, circuit switching can be used to connect a few number of modules, usually 2–8, and allow dynamic communication to be established between the components running onto the device at run-time.

4.1   1-D Circuit Switching: The RMB

In [4] [32], a 1-D circuit switching model called the reconfigurable multiple bus (RMB), well suited for the column-wise reconfigurable Xilinx Virtex FPGA is designed and implemented. The approach is based on a concept previously introduced in [204] and [77].

As figure 6.3 shows, the communication structure is made upon a set of switches, locally attached to a component using vertical lines. The connection between the switches is done through a bus, which consists of a set of segmented horizontal lines. The bandwidth of the horizontal connection may vary according to the application to be implemented. The function of the switches is to allow the component on which they are connected to dynamically access the bus and establish communication with other components.

The request for a new connection is done in a wormhole fashion. A sender components P_k, located in position k and willing to establish a communication with a receiver component P_t located in position t, must first send a request for communication to its own switch at location k. Switch k forwards the request to switch $k + 1$ in the direction of the receiver, which in turn forwards the request to switch $k + 2$, until the request arrives at the switch t. Each switch on the path from the sender to the receiver checks if the request for a
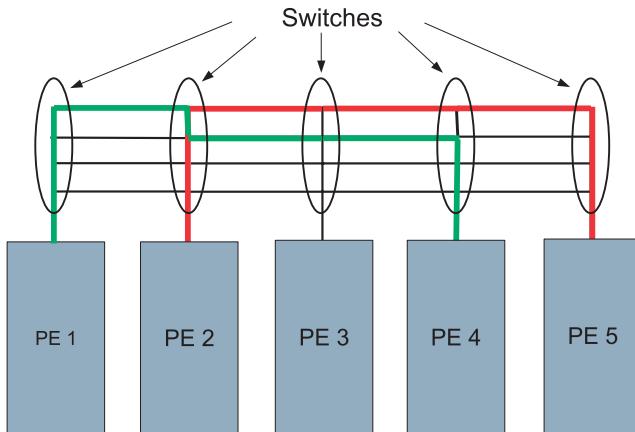
Figure 6.3. The RMBoC architecture

communication line can be satisfied, i.e. if a free communication channel is still available on the switch. If this is not the case, then the request is either queued or rejected. Otherwise, the switch sets the connection and sends an acknowledge. The first acknowledge is generated by the last switch on a path (in this case switch t) and sent back to switch $t - 1$, which in turn send the acknowledgement to switch $t - 2$, the process is repeated until the switch at the sender location. Upon receiving the acknowledgement, the sender can start with the communication. This approach was first presented in [204] and extended later in [77] with a compaction mechanisms for a quick finding of a free segment. A redesign as well as an implementation of this approach, called the RMBoC (RMB on Chip) for the Xilinx FPGA devices was presented in [4] and [32].

In of the RMBoC on the Xilinx Virtex FPGAs, the switch controllers are separated from the processor. In this way, a uniform interface is provided to designers for connecting their modules on the Bus.

To facilitate the partial reconfiguration on Xilinx FPGA devices, the device is segmented in a set of horizontal slots, each of which can accommodate a given module at run-time. The reconfigurable modules can then be placed only on the foreseen slots. Large modules that cannot fit on one slot can be implemented using a given amount of consecutive slots. The segmentation in slots is enforced using dedicated modules, the so-called *bus macros* (figure 6.4), at the boundaries of the slots. A bus macro is a special hardware module that allows established connections between neighbour component not to be destroyed during the reconfiguration process. The crosspoint locally manages the connection of the component to the Bus as well as the setting of switches according to the requests and the availability of the channels.
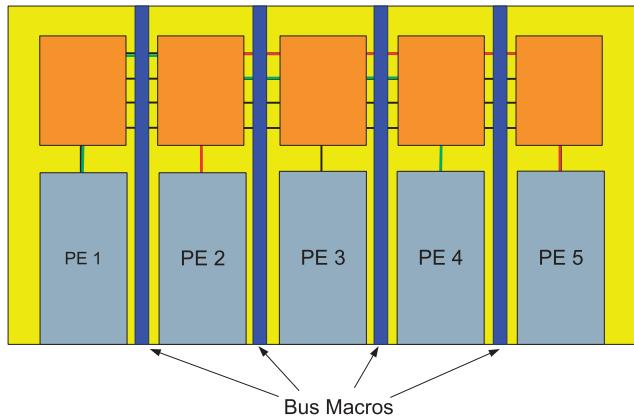
Figure 6.4. RMBoC FPGA implementation
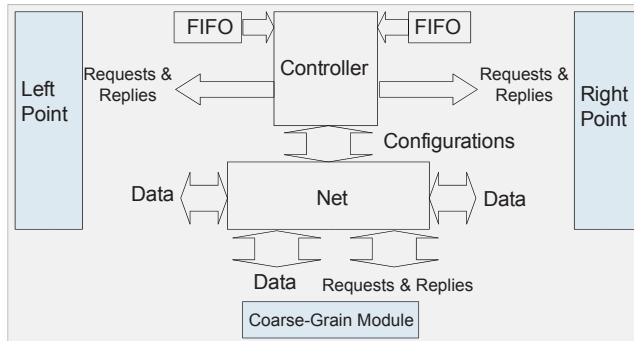
Figure 6.5. Crosspoint architecture

As depicted in figure 6.5, the crosspoint used in the RMBoC consists of three principal components.

■ **Controller:** The controller manages the switch at a local level according to requests from the left or the right crosspoint as well as that of the local module. Four kinds of command (REQUEST, REPLY, CANCEL and DE-STROY) can be locally processed on a single crosspoint, thus defining the interface between a processing node and its crosspoint.

A communication process starts by a REQUEST command from the sender to its local corresponding crosspoint with the destination address. The command is successively sent to the next crosspoint in the destination direction until the receiver location or the earliest crosspoint with no free channel. Upon reaching the destination, a connection will be established and a positive answer (REPLY) is sent back successively until the source. Each
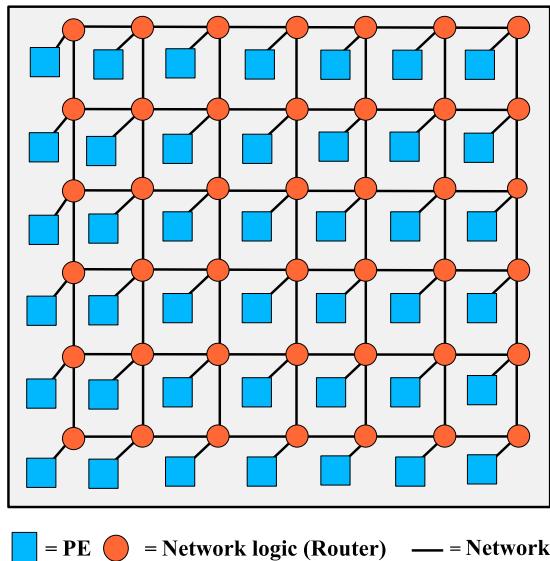
processing element that receives a REPLY sets its switch accordingly. If a processor cannot establish a connection in the destination direction, a negative answer (CANCEL) is sent back and the connection will not be established. Whenever a connection is successfully established, the communication is done on a point-to-point manner. At the end of the communication, the sender sends a DESTROY command to its crosspoint that successively passes it to the crosspoints involved in the communication. Each crosspoint frees the data channel after sending the DESTROY command to the next crosspoint.

- **Data network:** The function of data network is just to connect corresponding data channels according to the configurations modified by the controller. The original RMB foreseen that data must be transferred within one clock cycle from source to destination, once the connection is established. However, to avoid long delay and therefore slow clocks, a pipelined communication is realized in the RMBoC, using registers between the slots.

- **FIFOs:** The purpose of FIFOs is to provide buffer for commands coming from different side of the crosspoint (left, right, local). The FIFO Selector sends the command from the FIFOs of each side to the main FIFO. The policy of arbitration in the FIFO Selector is Round-Robin in the following order: left, right and local.

5.   Network on Chip

Many well-reputed authors [21] [117][60] have predicted that wiring modules on chip will not be a viable solution in the billion transistor chips in the future. Instead, they proposed Network on Chip (NoC) as a good solution to support communication on System on Chip in the future. NoCs encounter many advantages (performance, structure and modularity) towards global signal wiring. A chip employing an NoC consists of a set of network clients such as DSP, memory, peripheral controller, custom logic that communicate on a packet base instead of using direct connection. As shown on figure 6.6, several modules (network client) placed at fixed locations on the chip can exchange packets in the common network. This provides a very high flexibility, because no route has to be computed before allowing components to start communicating. Components just send packets, and they do not care on how the packets are routed in the network. NoC is viewed as the ultimate solution to avoid problems that will arise because of the growing size of the chip. The Quicksilver chip [89] can be seen as an example of NoC-based architecture.

A generic NoC architecture is characterized by the number of routers, each of which is attached to processing elements in the array, the bandwidth of the communication channels between the routers, the topology of the network and the mechanism used for packet forwarding. The topology of the network is

Figure 6.6. A Network on Chip on a 2-D Mesh
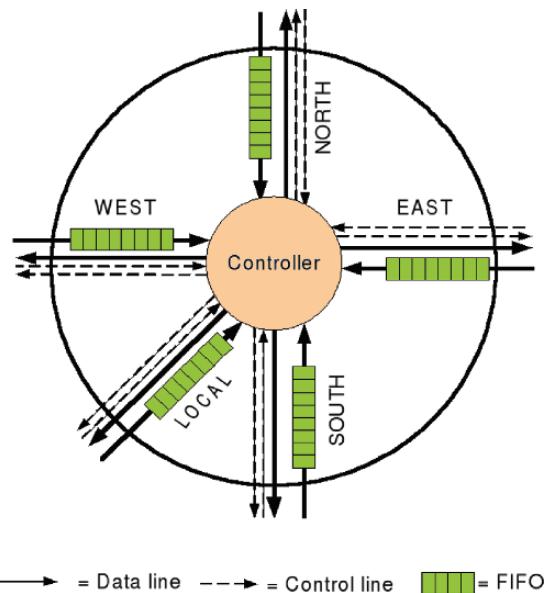
defined through the arrangement of routers and processor on the device and the way those processors are connected together. One of the most used topology is the 2-D mesh network, because it naturally fits the tile-based architecture on the chip.

Compared to typical macro networks, a network on chip is by far more resource limited. To minimize the implementation cost, the network should be implemented with little area overhead. This is especially important for those architectures composed of tiles with fine-level granularity as it is the case in FPGAs. Thus, instead of having huge memories (e.g. SRAM or DRAM) as buffer space for the routers like in the macro network, only few registers are available to be used as buffers for on-chip routers. This leads to a much simpler model with little overhead compared with its macro network peer.

We next take a look on the design and implementation of major NoC components, which are the routers and the processing elements.

5.1 The Router

The basic building blocks of an NoC are routers that consist of a given set of components like buffers to temporarily store packets, a controller that determines how to forward the packet, usually according to the destination address. The performance and the efficiency of the NoC depend on the underlying communication infrastructure, which in turn depends on the performance (latency and throughput) of the on-chip routers. Thus, the design of efficient, high

Figure 6.7.   Router Architecture

performance routers represents a critical issue for the success of the NoC approach. While several possibilities may exist to implement a given router, we focus on a very simple and intuitive model consisting, as depicted in figure 6.7, of the following main components:

1 Five input buffers, usually implemented as FIFOs, to temporally store messages coming from five directions left, right, north, south and local PE, before forwarding. Each router willing to send a message in a given direction copies it into the FIFO of the neighbour router in the direction where the message will be forwarded. The data are therefore placed on the data lines and the control signals are used to perform the handshaking between the two neighbour routers.

2 The controller, which is in charge of managing the dataflow inside the router. It defines in which direction a message should be sent and sets the signals in the router to allow this to happen.

3 Five Output Arbiters, each for one direction and the local PE. The output arbiters manage the assignment of the message to output channels.

   We now provide a much detailed description of the router components:

5.1.1   The FIFO

A first-in, first-out (FIFO) is a data storage to model a queue in such a way that the data that come in first will be read out first. It consists of a set of registers for storing the data. The registers are organized in a serial way that allowed data to be shifted from one register to the next one. Data are pushed (written) on one side and read from the other side. Figure 6.8 provides the general architecture of a FIFO.

It consists of two counters to count the written and read registers. One write enable signals allow data to be written into the registers and one read enable signal to notify about the readiness of the data from the registers. Two signals are also used as flags to inform about the fullness and emptiness of the FIFO. If full, then no more data can be written to the FIFO. An empty FIFO means that no data are available in the FIFO registers. If a common clock is used for reading and writing the data then the FIFO is said to be *synchronous*. Otherwise, there will be two different clocks for reading and writing the data. In this case, we said that the FIFO is *asynchronous*. The FIFO can be parameterized with the data width (number of bits in a register) and FIFO depth (number of registers in a FIFO).

5.1.2   Router Control

Each router is identified through its position in the network. This position is determined by the (*x*,*y*)-coordinate of the processing element on which the router is attached. Messages are sent in packets, each of which consists of the address of destination router, control bits and the payload (data). A possible format for the packet being sent in to the network is shown in figure 6.9
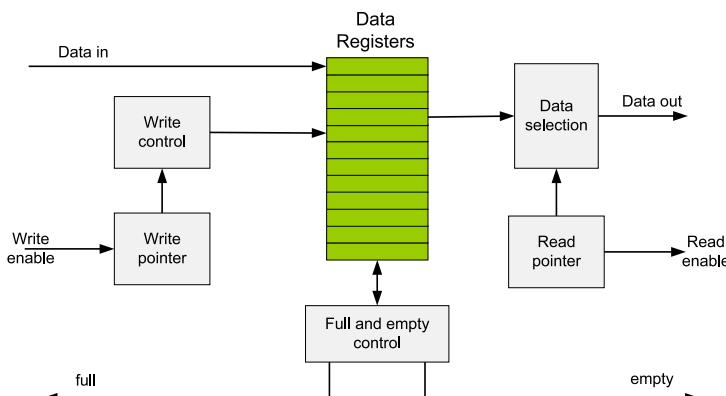

Figure 6.8.   A general FIFO Implementation

Address	Control bits	Payload (data)

Figure 6.9.   General format of a packet

The first part is the packet address, which is used by the router control to determine the direction where to send the packet. The controller has an address decoder that decodes the address into (x,y) coordinate of destination router or PE. In the simple case of XY-Routing as we will see later, a comparator is used to compare the (x,y) coordinate of the destination PE to that of the router to compute the direction (LOCAL, EAST, WEST, SOUTH, and NORTH) where the packet will be sent. The packet is then sent out by writing in the input FIFO of the corresponding neighbor FIFO if this is not full. If the FIFO in the neighbour router is full, then the router can decide to take some action. It can for instance block all incoming packets or send the packet in another direction to decongest a given data line.

5.1.3   Output Arbiter

To increase performance in the router, input FIFOs in all directions must be read concurrently if they are not empty, i.e. there is at least one packet in a FIFO. After reading the packets in parallel, the control logic decides in which directions the packets should be sent. The packets are then written into the neighbour router in the chosen directions. The case where the control decides to forward many packets in the same direction leads to a contention, because only one output data line is available and many packets cannot be written at the same time. For a contention free routing, an arbiter must be provided at every output port. A simple arbiter may consists of a multiplexer and a Finite State Machine, which will make the decision in a Round-Robin fashion. If the data must be written on the local output line, the order can be EAST, WEST, SOUTH, and NORTH as shown in figure 6.10.

The incoming packets from the EAST will be written before the one coming from the WEST, which in turn is written before the packet coming from SOUTH that is written before the one coming from NORTH. Incoming packet from LOCAL is not considered here, because we assume the a packet coming from a given direction will not be sent back in that direction. This assumption is only a design decision that can be made or not. However, we can choose to give the router the freedom to decide where to forward packets. In this case, a packet can be returned in the same direction from where it was sent. This capability may lead in some routing algorithm to a ping-pong game between two routers. However, it may also help to decongest some over-filled channels.
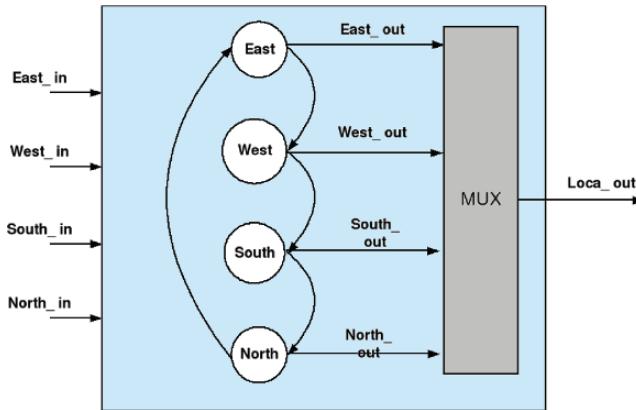
Figure 6.10.   Arbiter to control the write access at output data lines

5.2   The Processing Element

A processing element can be either a processor core, a memory block, an embedded programmable logic or any custom hardware block. Processing elements are usually connected to the upper right router through an interface in the network. Each processing element has a unique address, which is the same address of router to which it is connected. Thus, any PE can be plugged into the network if its design fits into an available slot. To make this process easy, a wrapper is usually used to decouple the network activities from the computation within a processing element. The wrapper controls all the transaction on the network and provides to the component a simple interface to access the network.

5.2.1   Wrapper Design

The PE port is connected through a wrapper to the router. The port width (or channel width) determines the size of the packet (message) that can be moved in the network through routers. For consistency, wrappers use the same port width as the router port width. A wrapper consists of only one FIFO at input port as shown in figure 6.11. The main functionality of the wrapper is to provide an interface to the processing element or the module connected to the network. Its logic includes decoding and encoding the received and sent packets, respectively. Incoming packets to the PE carries the address along with the payload. This address is removed in the wrapper, and only the payload is passed to the PE. In the same way, wrapper gets the payload from the PE, adds the address of the destination PE to the payload and formats the packet before giving it to the connected router. Always an incoming packet is written to the FIFO in the wrapper, and an outgoing packet is written to the FIFO in router to
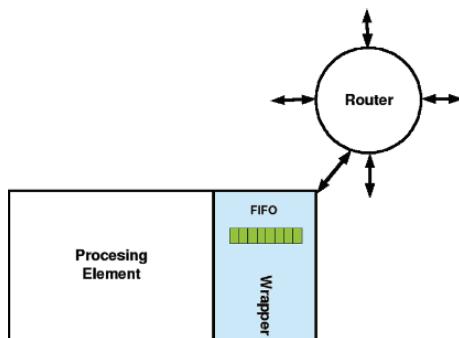
Figure 6.11.   A general wrapper architecture

which the PE is connected. The wrapper reads the packet from the FIFO only when there is a packet in that FIFO. In the implementation, the processing element is instantiated as functional block within the wrapper.

5.3   NoC Design Constraints

During the design of an NoC, important decision must be taken according to a goal seek by the user. Each design decision influences a given parameter in the final implementation. The area overhead, i.e. the amount of resources that the final design consumes as well as the latency, i.e. how long in average it takes to route a packet to destination are the most important parameters that guide the design process.

5.3.1   Area Overhead

When considering a NoC architecture, the area used by the routing hardware is important. Costumers will probably not be interested in those design, where the routing hardware occupies 90% of the device area and only 10% is left for the implementation of the PEs. The area consumed by the routers depends on the bandwidth requirements, i.e. the packet width, the buffer size and the complexity of the control algorithm. The packet size determines the width of connection between routers. This makes it directly proportional to the amount of internal wire required. The buffer size determines the amount of memory used for storing the packets within the router before forwarding. The complexity of the routing algorithm determines how much additional resources the router consumes.

5.3.2   Latency

The latency consists of the time needed to setup a route and the time need to transfer the payload to destination. In circuit switching, route setup time

includes request and acknowledgment latency, but in packet routing, there is no such set up time. Only the address flit takes initial setup time to reach the destination based on the routing algorithm, thereafter for every cycle, the data flit will be delivered to the destination in a deadlock free network. In deadlock-free store-and-forward packet switching, each packet spends one cycle to pass through each router. The total number of routers between source and destination will determine the latency. For example, if the source and destination are placed at diagonally opposite corners of a 4×4 mesh, each packet being transferred will take 8 clock cycles to reach the destination. In destination, IP block perspective, for every clock cycle a packet will be received. The latency completely depends on the chosen routing algorithm (adaptive or deterministic) and the buffering method.

5.3.3 Performance Metrics

The performance of a network provides a mean to compare two different implementations of the network. The two main means used are the latency and the throughput.

- *Latency*: This is the time a message needs from its source to its destination. It is the difference between the time where the last packet of the message arrives at destination and the time when the first packet of the message is output from the source.

- *Throughput*: Often used in conjunction with latency. This is the maximum traffic a network can accept per unit of time, typically measured as bytes or packets per node per cycle.

5.4 Routing Techniques

Routing is one of the most understood area in computer science. A large variety of routing algorithms exist in the literature, each with its advantages and drawback. We provide some of the most used class of routing algorithms in this section.

5.4.1 Circuit Switching

Circuit switching was presented in Section 4. This approach allows a communication path to be created from the source to the destination before transmitting any data. The procedure starts by a routing probe traversing the network and reserving links to transmit the data. This routing probe contains the source and destination addresses. Once the routing probe reaches the destination address, an acknowledgment is sent back to the source address, then the data are transferred at the full bandwidth of the hardware. The circuit remains operational until the end of data to be transmitted. The lock on the links may be released once all the data have reached the destination by sending

back another acknowledgment through the same route to the source. The disadvantage of this technique is the time required to establish a dedicated link from source to destination. It can be advantageous when the time to set up the path is minimal, compared with the transfer time of the messages, and when long messages are present, especially continuous data streams.

5.4.2   Store-and-Forward

At each node, the packets are stored in memory and the routing information examined to determine which output channel to direct the packet. This is why the technique is referred to as store-and-forward (SAF). Additionally, by transferring an entire packet at a time, the latency for a packet is the number of routers through which the packet must travel multiplied by the time to transfer the packet between the routers.

5.4.3   Virtual Cut-Through

Virtual cut-through (VCT) was conceptualized to address the deficiency in SAF-based packet routing, which results from the buffering of messages at each node along the communication path. As the routing information is carried in the header of a packet, the packet should not be stored in the current node's memory if an output buffer is available. The packet simply cuts through the router of the node to an available output channel. This alleviates the need for an excessive amount of memory along the path of a message, but enough memory has to be allocated if an output channel is not available. Of course at high volumes of messages on the network, VCT will behave similarly to SAF packet switching. The unit of message flow control for VCT is the packet [184].

5.4.4   Wormhole Routing

Wormhole routing was conceived to address the deficiency in VCT, that is, if an output channel is not available, the packet must be stored in the current node's memory. Wormhole routing divides a message into smaller flow-control digits than packets, which are called flits. Each message contains one header flit that carries the routing and control information and the remaining data for the message are stored in data flits. The header flit always goes first to allocate a path for the data flits. Thus, smaller memory requirements exist for each node on the network (buffers flits instead of packets). If an output channel is available, the header flit is routed and the remaining data flits follow in a pipeline style fashion. During any instance of a message traversing a network, the flits of a message will be located in multiple routers. This routing technique got its name in that it looks like a worm traversing the network. While this method benefits from very low latency and buffer space, blocking and deadlock

problems can occur. Therefore, other techniques, such as virtual channels, are needed. An advantage of virtual channels is the ability to share a single physical channel.

5.4.5 Deadlock and Livelock

Deadlock is a situation that occurs when a packet is waiting for an event that can never happen because of a circular dependence on resources. Livelock, on the other hand, is a configuration of the network in which packets continue to move, but never reach their destination.

A routing algorithm should determine the optimal routing path to transport packets from source to destination through the network. The optimality is usually defined according to the following parameters: high performance, low overhead, Deadlock and livelock freeness, fault-tolerance and flexibility.

In general, routing can be done in two ways: *deterministic routing* and *adaptive routing*.

5.5 Deterministic Routing

A deterministic routing algorithm provides a unique path from a source to destination. XY-routing also called dimension ordering routing is a simple deterministic routing algorithm, where messages are transmitted fully in each dimension, beginning with the lowest dimension available. In a 2-D mesh network, XY-routing first routes packets along the *X*-axis. Once it reaches the destination's column, the packet is then routed along the *Y*-axis until the destination's line. Therefore, any packet moving in the *Y*-direction can never return to the *X*-direction. The XY-routing algorithm routes the packets based on the destination address, irrespective of the traffic pattern on the link and the link delay. The router compares its own address (Xrouter, Yrouter) to the destination address of a packet. Packets are forwarded based on destination address (Xdest, Ydest) as follows.

1 If Xrouter < Xdest, the packet ist forwarded in the east direction

2 If Xrouter > Xdest, the packet ist forwarded in the west direction

3 If Xrouter = Xdest and Yrouter > Ydest, the packet is sent to the south of the current router

4 If Xrouter = Xdest and Yrouter < Ydest, the packet is sent to the north of the current router

5 If Xrouter = Xdest and Yrouter = Ydest, the packet is sent to the local PE

5.6   Adaptive Routing

In adaptive routing algorithms, the direction where to send an incoming packet is not fixed a priori. The routing algorithm may decides to use more complex approaches to decide on the further direction of a packet. Adaptive algorithms are usually used to improve the performance in the presence of localized traffic or to provide fault-tolerance in the network. Packets are not always routed along the shortest path. The longest routing path may be preferred to the shortest one, if the longest path provides the best better fault tolerance result or if deadlock and livelock problems can be avoided. An example of adaptive routing is the Q-routing, an adaptive routing algorithm based on Q-learning, which is a form of reinforcement learning [36]. It routes packets based on the learnt routing information from its neighbours. Initially, this algorithm builds a routing table based on the delivery times (Q values) of the packets to every router in the network. These delivery times are updated every time a router forwards a packet for a particular destination, which changes depending on the traffic at a given time to the destination.

On the basis of this information, a router can choose an alternative route when the queues are congested in the intermediate routers, thus resulting in faster delivery compared to the XY-routing algorithm.

With reinforcement learning in general and Q-learning in particular, each router is an autonomous structure that learns with the time the efficient route to all possible destinations. On communication request, the best router, i.e. the router through which the fastest communication is possible (from the point of view of that router) is selected and the packet is sent there. With this method, all the routers need to learn the new structure of the network whenever a change is done.

With a frequently changeable network, the router will spend most of its time for learning the new network structure, thus decreasing the network performance.

Adaptive routing algorithms are more complex than deterministic routing techniques, which means that the amount of resources consumed by the router that implements adaptive routing is much higher than the amount of resources consumed by router implementing deterministic routing. Therefore, the complexity of those algorithms does not qualify them to be used on a chip. The XY-algorithm usually performs well in the practice and routes packets according to the lower Manhattan distance.

Network on a chip presents a viable communication infrastructure for communication among task dynamically placed on a reconfigurable device. However, it is still too inflexible to dynamically support the communication among modules in a changing network. When placing a component on a reconfigurable device, we face two possibilities: First, the components fits on one PE, i.e. the resources needed by the component is no more than that available on
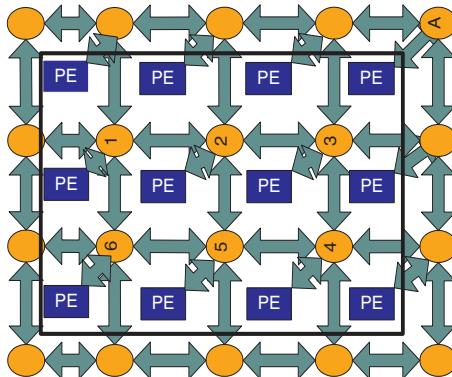
Figure 6.12. Implementation of a large reconfigurable module on a Network on Chip

a PE. In this case, we place the component on one PE and attach it to the corresponding router to allow a communication with other components. In the second case, the component does not fit on one PE, i.e. more than the amount of resources offered by a PE is necessary to implement the component. In this case, the component will be splitted in pieces, each of which can fit on a PE. The communication between the pieces placed on different PE will use the router network for communication. In other words, the communication within a component boundary must be done using packets that are sent and routed in the network. This increases the complexity of the module, and more resources are wasted than needed. This situation is best illustrated in the figure 6.12.

There, a large module needs 12 PEs to be implemented. It is splitted in parts, each of which is implemented on one PE. The connection among the parts is done using routers. A better implementation would connect all the PE using direct wiring. This has no disadvantage, because the PEs are close to each other, and therefore, long connections are avoided. In figure 6.12, the routers inside the area of the components (1,2,3,4,5,6) become redundant.

To overcome this handicap, the concept of dynamic network on chip that we next presented was developed. In the dynamic network on chip, the redundant routers can be used as additional resource to implement an even bigger module.

6. The Dynamic Network on Chip (DyNoC)

A Dynamic Network on Chip (DyNoC) [35] [32] is a Network on Chip whose structure can be dynamically changed at run-time. In a DyNoC, a routers is a programmable element basically configured as router but that can be configured at run-time to implement any function that can fit on it. This concept provides a mean to overcome the deficiency of network on chip as communication paradigm in reconfigurable computing (figure 6.12).

Whenever a module is placed at a given location where it uses the resources of several PEs, the redundant routers within the module boundary can be used as additional resources for the module. The placed module only needs one router to access the network. With this, the routers in the area of a placed component are no more accessible by other components in the network. Upon completion, the module is removed and the network must be reactivated in the area where the component where previously placed. This can be done quickly, because the router are programmable components that can be quickly reset to their basic configuration, i.e. the routers.

With this, we have a network in which some parts may be deactivated at a given period of time and reactivated in the future. In order for such a network to efficiently operate, some prerequisites on the communication infrastructure of the network must be fulfilled. We explain these next.

6.1   Communication infrastructure

The goal is to have a communication infrastructure in which the reachability of packets is insured, independent on the changing topology which occurs when components are placed and removed on the chip. In its basic state, the communication infrastructure is a normal NoC. Processing elements access the network through corresponding routers. Additionally, direct communication lines exist between neighbour PEs. In this way, the routers are only used for communication between non-neighbour PEs. With this modification, component splitted across several PEs with router inbetween do not need the router anymore for communication, because direct lines already exist to connect the neighbour PEs. The direct lines can also be modified to connect two PEs not directly neighbour and allow more flexibility. The cost of realizing such an hybrid architecture with a network on one side and direct routing on the other side will be more than that of a normal NoC; however, the gain in flexibility is enormous.

Routers in those area where a component is placed become useless, and therefore, they can be used to provide more resources (logic and memory) to the component. The realization of such routers on the current programmable logic device is too costly, thus making the realization of a DyNoC communication infrastructure only possible if the router are directly available as hard macros on the device, as it is the case with the switch matrices, the embedded multipliers or the block memories on FPGA devices. Whenever a component is placed in a given region, only one router is necessary for this element to access the network. We use without loss of generality the router attached to the upper right PE of the module (figure 6.12).

Figure 6.13 shows a communication infrastructure as described below on a reconfigurable device. Note that the arrangement of the router is done in such a way that the device is surrounded by a ring of routers. This arrangement
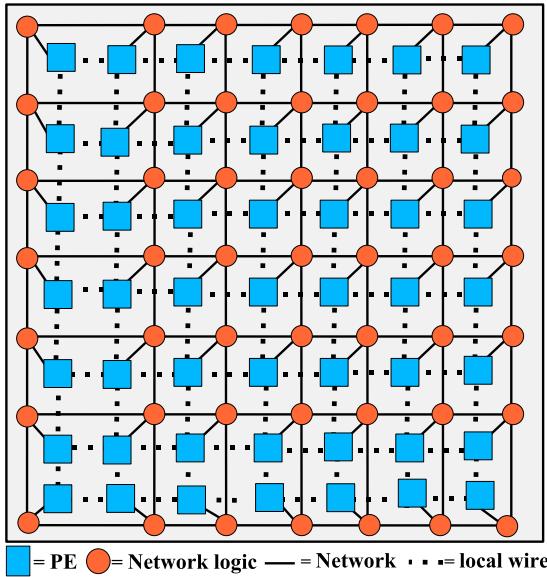
Figure 6.13. The communication infrastructure on a DyNoC

increases the flexibility and, as we will see later, is a prerequisite for a accessibility of placed components.

A further requirements on the structure of the network is that routers should be able to notify their neighbours about their activity. This can be done using an additional *activation line* that is set to one if the router is active, i.e. no component is placed on top of that router and zero if a component is using that router in its internal implementation. In this case, the activation line is controlled by the component, which notifies the routers around on the states of the routers that it internally uses.

6.2 Network access

As stated earlier, we would like to have a network in which all component are reachable, despite the dynamic modification of the network. Let us first recall the way component are developed for partial reconfiguration. Each task that can be executed at run-time is implemented as a component, represented by a rectangular box and stored in a database. The bounding box delimits the area of resources used by that component. In fact, the component is available as partial bitstream that will be downloaded onto the device at run-time, whenever the corresponding task needs to be executed.

Those components are hardware modules that required to be synthesized from a given description. As synthesis is a time-consuming task, it cannot be done online. Therefore, the synthesis of components is done at compiled time.

The result of the synthesis of a component is box that encapsulates the circuit implemented with the resources in a given area (routers logic and PEs).

After the placement of a new component on the device at run-time, its coordinate is set to that of its corresponding router. As the placement destroys some part of the network, we must insure that packets sent to other components will still reach their destination.

6.2.1   Reachability of Components and Pins

In this section, we define some conditions that must be fulfilled by a temporal placement of components on the device to enforce the reachability of all the components on the device an those of the pins around the device.

DEFINITION 6.2 (REACHABILITY OF COMPONENT AND PIN) *Given a reconfigurable device and a configuration, i.e. a set of components actually running on the device, we said that a component (pin) on reconfigurable device at a given time is reachable if each message sent to this component (pin) can reach the component (pin).*

Because the communications between components are established at runtime and because the configuration of the chip is not known in advance, we must insure that all components and pins on the device are reachable at any time during the temporal placement. This condition is fulfilled if at any time the set of components and pins on the device is strongly connected.

DEFINITION 6.3 (STRONGLY CONNECTED CONFIGURATION) *Given a reconfigurable device, we said that a configuration of the device is strongly connected if for each pair of components A and B, a path of active routers that connects the two components, exists on the device.*

One way to enforce the strong connectedness in a DyNoC is to require that each component placed on the chip must always be surrounded by a ring of active routers. This can be reached either by synthesizing components in such a way that when placed on the device, they are always surrounded by a ring of routers. The second possibility is to let the job to a temporal placer. This will considerably increase the complexity of the placer. Besides the computation of free space to place a new component, it must insure that all placements are strongly connected.

THEOREM 6.4 *If each component is synthesized in such a way that it is internally surrounded only by processing elements, then all placements on the reconfigurable device are strongly connected.*

Proof : Assume that a set of components developed as required in theorem 6.4 and placed on the device is not strongly connected. In this case, at least

one pair of components abuts or a component abut the device boundary. Let us consider the first case. The second one can be handled in a similar way. Either the two components overlap or at least one component uses some routers on its internal boundary (this is illustrated in figure 6.14) . The first case is impossible because only overlapping free placements are valid. The second case contradicts our requirement of the theorem, thus completing the proof. ∎

EXAMPLE 6.5 *Consider the placement of figure 6.14 with two abutting components. The first component attached to router A is not implemented according to the guideline of theorem 6.4, because routers are used in its internal boundary. The second component attached to router B is developed according to the rule of theorem 6.4. No route is available between those two component, because the only routers available are consumed by the first component.*

The configuration of figure 6.15 allows the reachability of all components and pins. All the components are implemented according to the guideline of theorem 6.4. Therefore, no matter where a component is placed on the device, it will always be surrounded by a ring of routers.
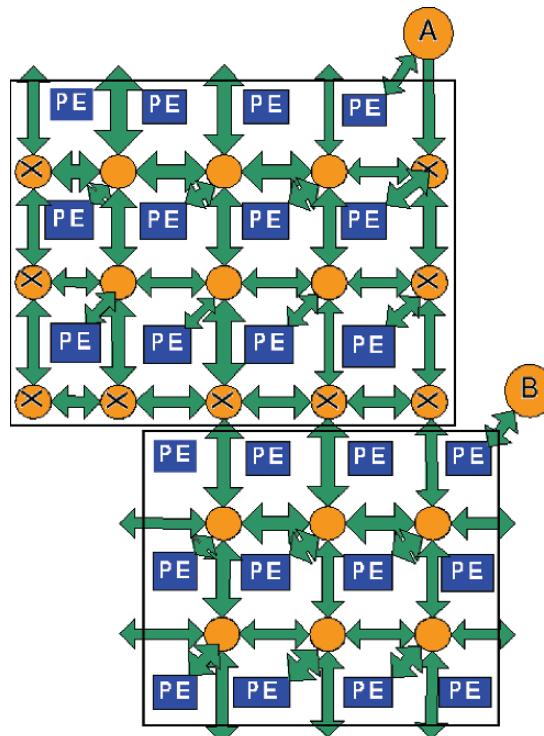
Figure 6.14. A impossible placement scenario
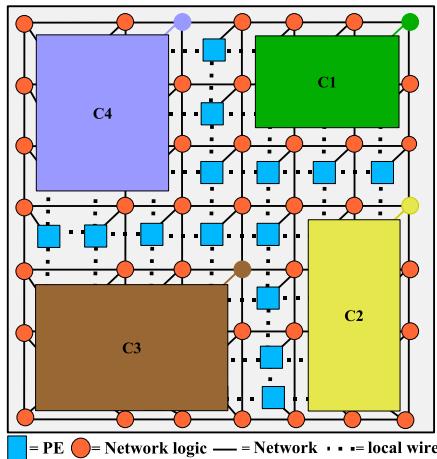
Figure 6.15. A strongly connected configuration on a DyNoC

While in a static NoC, each router always has four active neighbour routers,[1] this is not always the case in the DyNoC presented here. Whenever a component is placed on the device, it covers the routers in its area. As those routers cannot be used, they are deactivate. The component therefore sets the activation signal to the neighbour routers to notify them not to send packets in its direction. Upon completing its execution, the deactivated routers are set to their default state. A routing algorithm used for common NoC cannot work on the DyNoC, without modification. An improvement or an extension of the existing routing algorithms should be done in such a way that packets may be able to surround the components on their way to the destinations.

6.3   Routing Packets

Because the dynamic placement and removal of modules on the chip creates unpredictable obstacles in a DyNoC, adaptive routing algorithms must be used to handle the dynamic changing configuration of the network. The routing algorithm must be fully local-decisive,[2] deadlock free and live lock free. We must insure that each packet sent in the network will reach its destination after a finite number of steps. While very complex and intelligent algorithm may be developed, a very simple strategy was proposed in [35] [32] to allow the packets to surround the obstacle, whenever they encounter one.

---

[1]Routers around the chip does not have four neighbours. The package pins can be considered as further neighbours where the routers can send a message. This leads to the availability of the four neighbour routers everywhere on the chip.

[2]The decision where to send a packet is taken locally.

The strategy is based on the XY-routing, because of its simplicity, its efficiency and its deadlock freeness, the XY-routing algorithm presented in Section 5.5 was adapted for the DyNoC. Recall that in a full mesh, the XY-routing is a deadlock free, shortest path routing algorithm that first routes packets in *X*-direction to the correct *X*-coordinate and then in the *Y*-direction until the correct *Y*-coordinate. The adapted XY-routing, the *S-XY-Routing (Surrounding XY-routing)* operates in three different modes:

- *The N-XY (Normal XY)* mode. In this mode, the router behaves as a normal XY router.

- *The SH-XY (Surround horizontal XY)* mode. The router enter this mode when its left neighbour or its right neighbour is deactivated.

- *The SA-XY (surround vertical XY)* mode. The router enter this mode when its upper neighbour or its lower neighbour is deactivated.

The normal mode is available when all the neighbours of a routers are active. In this case, it is the XY-routing that is used.

6.3.1   The SH-XY mode: Surrounding Obstacles in the X-direction

Assume without loss of generality that a packet moving from right to left is blocked by an obstacle. As shown in figure 6.16, there exist two alternative paths for the packet to reach its destination.

According to the XY-routing, which is the base of this algorithm, the first path is chosen if the *Y*-coordinate of destination of the packet is greater or equal than that of the router and the packet is sent upward. Otherwise, the second path is chosen and the packet is sent downward. One problem occurs when a packet with destination Y_{dest} is sent for example upward and riches a router r with coordinate $Y_{\text{r}} > Y_{\text{dest}}$. According to the previously defined scheme, the packet will be sent downwards to the router with coordinate $Y_r - 1$, which will send it upward, thus producing a 'ping-pong' game. Figure 6.16 illustrates this situation. In this case, the ping-pong game happens between routers 1 and router 2.

To avoid this, a strategy consists of stamping the packet by setting a "stamp-bit" to 1 to notify the next routers that the packet is willing to surround the component, and it should not be sent back. In our example, the packet is stamped by the router 1.

Upon reaching the router upper right of the module to be surrounded, the stamp is removed and the packet is sent left, until its destination column or until another obstacle is found. Because each component is always surrounded by a ring of routers, this algorithm for surrounding a component in the X-direction will always work and a packet will never be blocked. In the example of figure 6.16, path 1 will be chosen.
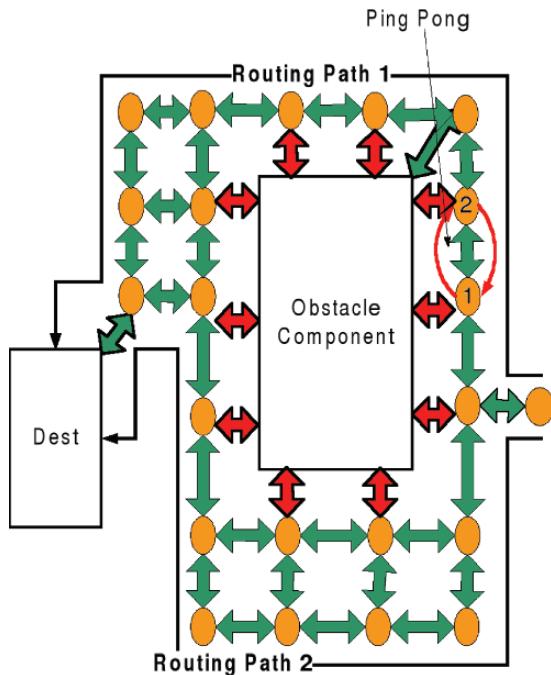
Figure 6.16.   Obstacle avoidance in the horizontal direction

6.3.2   Surrounding Obstacles in the Y-direction

The situation is similar to the one where a packet moving in the *X*-direction is blocked. Assume without loss of generality that a packet moving from top to bottom is blocked by a placed component. Dealing with this case as the previous one, the packet will be sent left or right. No preference is done here, because the packet is already in its right column. Let us assume that the packet is sent to next router right to the current one. Because the basic routing algorithm is the XY-routing, the next router will first compare the X position of the packet with its own position. With the packet X-destination being smaller, it will send the packet back to the router from which it was received. The two routers will keep sending the same packet to each other, thus creating a deadlock. Figure 6.17 in which a 'ping-pong' game will then happen between router 2 and router 4 illustrates this situation.

As in the previous case, the packet is stamped to notify all the routers above the obstacle that the packet is willing to surround the component. In our example, the packet will then be sent right until the last router (5) above the component. There, the router removes the stamp and sends the packet downward. From there on, we have the same situation as defined in the previous section (surrounding obstacles in the *X*-direction).
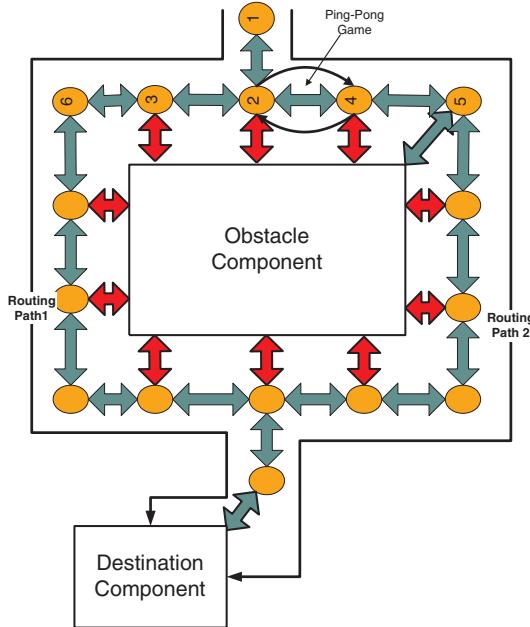
Figure 6.17. Obstacle avoidance in the vertically direction

In the two cases, a path will be always available from the source to destination. With the XY-routing used as underlying routing technique, we conclude that each packet sent will reach its destination in a finite number of steps.

THEOREM 6.6 *With very high probability, the S-XY algorithm presented here is deadlock free and livelock free.*

Proof : We need to first prove that there is always a path from packet to destination. With this, the only possibility to have a livelock is that the packet moves permanently through the network and never reaches its destination, despite the availability of a path.

Second, we must prove that each packet will reach its destination after a finite number of steps. The first requirement is guaranteed through theorem 6.4.

We now assume that a packet never reaches its destination. This will happen only if the packet is blocked or if the packet is looping in a given region. Because a path always exists from one active router to all other active routers, no packet can be blocked in the network, i.e. a packet is looping. As this situation is not possible in the normal XY, it can only arise in the surrounding phase. When a packet is blocked in a given direction, it takes the perpendicular direction. This is done until the last router on the component boundary that is at one corner of the module to be surrounded. From there on, the normal
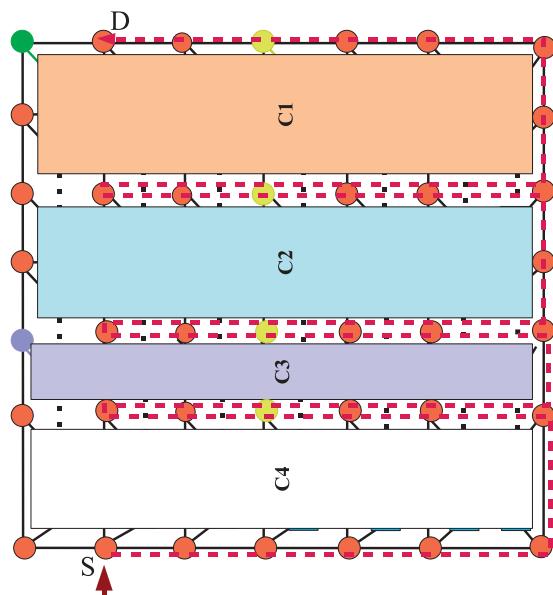
XY-routing resumes. A looping of a packet around a component is therefore not possible.

It may be possible to construct a placement sequence of components that blocks a single packet whenever that packet moves in a given direction, forcing it to take the perpendicular direction. Such a sequence may lead to a packet that will loop in a given region. However, doing this will only block a single packet, and other packets will move to their destination as previously described. This leas to a very low livelocking because only one packet out of an infinite number is livelocking. However, such placement sequences are very unlikely to happen in the reality. ∎

In the S-XY routing, the direction where to send a packet whenever an obstacle is encountered is fixed for all routers a priori. In this particular case, packets are always sent to the right, whenever they are blocked in the vertical direction. This may lead to extreme long routing paths like that of figure figure 6.18. Those extreme paths are not usual, but may occur as result of some placements.

6.3.3   Router Guiding

A simple and efficient strategy to avoid such path consists of instructing the router around a component on the direction where a packet must be sent,

Figure 6.18.   Placement that cause an extreme long routing path
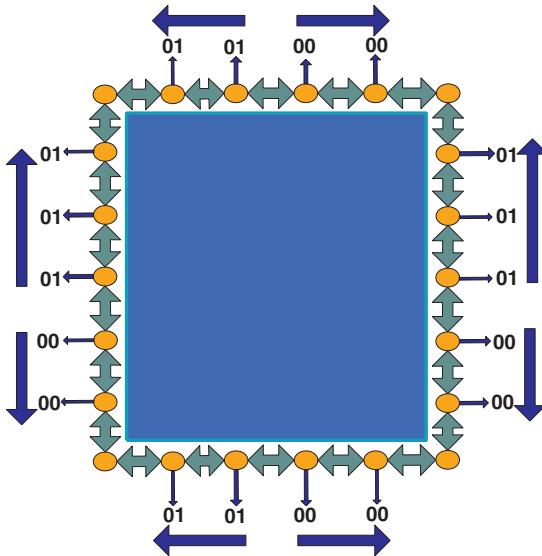
Figure 6.19. Router guiding in a DyNoC

whenever the incoming packet is blocked by the component in a given direction. Only one line is necessary to realize this concept.

Figure 6.19 illustrates this approach that we call the *router guiding*. Besides the activation line, which provides information on the state of a router (1 = activate, 0 = deactivated), an additional line is used for instructing the router on the direction where they should send packet to, in order for them to surround the component. A value of 0 (= east or south) will mean that a packet blocked in the vertical direction must be forwarded east and a packet blocked in the horizontal direction must be forwarded south. A value of 1(= west or north) means that a packet blocked in the vertical direction must be forwarded west, while a packet blocked in the horizontal direction must be forwarded north.

The best direction where to send packets whenever they are blocked by a component can easily be determined by the component shape, a process that can be managed at compile-time.

This approach considerably limits the complexity of the routers, and there is no need for stamping anymore. We call this modification the *router guiding* because the routers are guided by the components.

6.4 Analysis of Efficiency

We define the communication latency of a packet as the number of routers on a path from its source to its destination. On a reconfigurable device with width W and height H, the maximum latency is given by $W + H$. However,

the unpredictable structure of the network that changes with the placement of components creates unpredictable delays. The worst case delay $D_{wc}(p)$ of a packet going through k obstacles from its source to its destination is given by equation

$$D_{wc}(p) = H + W + \sum_{i=0}^{k} D_{wc}^i(p) \qquad (6.1)$$

$D_{wc}^i(p)$ is the worst case additional delay caused by the obstacle i. If w_i is the weight of the components i and h_i its height, then we have $D_{wc}^i(p) = max(w_i, h_i)$. The half perimeter of the component is not added here, because it is already in the sum $H + W$. Only the maximum between the components height and its width is considered. This additional delay is unpredictable and depends on the temporal placement of components on the device.

In a static NoC where each processing element is assigned a fixed place, the delay is also fixed. Packet communication can be seen as pipelined computation in which the length of the pipeline is the packet delay from source to destination. Once the pipeline if filled, one packet is received at the destination on each clock.

In the DyNoC, we have the same situation; however, delays created by newly placed components create new situations for which the path must first be filled with packets. It is even possible that some packets sent later reach their destination before others, which were sent earlier, but which had to go through some obstacles. This makes the problem difficult for a formal analysis.

6.5   DyNoC Implementation on FPGAs

DyNoCs with different size and width were prototyped on two FPGAs [35] [32], the VirtexII-1000 and VirtexII-6000 from Xilinx. While the prototype on the VirtexII-6000 were mostly for statistical (area, latency) purpose, the implementation on the VirtexII-1000 was done on the RC200 FPGA-board from Celoxica [43]. The result is given in table 6.1 in terms of area (A) occupation for different bit-widths, memory (M) usage and speed (S) in MHz.

Two video applications with a VGA controller running at 25 Mhz for normal 640×480 VGA were implemented. In the first one, a colour generator module

Table 6.1.   Router Statistics

	VirtexII-1000	VirtexII-6000
A/M/S(8 bit)	8% /4% / 77.2	1% /0% / 77.2
A/M/S(16 bit)	12% 7% / 75.4	2% /1% / 75.4
A/M/S(32 bit)	21% / 12% / 77.3	3% /2% / 74.9
A/M/S(64 bit)	46% / 28% / 70.1	7% /4% / 73.7

(CG) communicates with the VGA controller (VC). The colour generator gets the X and Y coordinates of the current pixel position from the VGA module, computes the colour to be placed at that position and sent it back to the VGA module that displays the colour at the corresponding position. The CG application is a nice method to detect changes in the communication, because this will directly have a visual effect on the screen. The X and Y positions are each 12 bits wide and the colour is 24 bits wide. Therefore, packets with 32 bits width in each direction were used. Implemented on the RC200 board with the DyNoC, no change was detected in the displayed pattern, event with a full network traffic because of the communication among remaining routers.

The second application is the implementation of a traffic light controller (TLC) containing three modules. A VGA controller (VGA), a Traffic light visual module (LV) and a traffic control module (TC) to capture the pedestrians wishes. As in the first case, the VGA module is used to display the state of a traffic intersection on which the light and the button used by the pedestrians can be seen. The traffic visual module is in charge of building the traffic light infrastructure, which is then displayed by the VGA module. The VGA sends the X and Y pixel scan positions to the traffic visual module and receive a colour to be displayed. According to the pixel positions, the traffic light visual computes the pattern to be placed at that position. This generates the traffic light infrastructure. The last module is a FSM that monitors the pedestrian inputs (two push buttons on the board) and sent a message for the transition of state of the traffic infrastructure to the traffic light visual, which in turn generates the corresponding colour to be seen. The traffic light controller was successfully implemented on 3×3 DyNoC. The router at position (2,2) were disable to enforce a surrounding. All the remaining routers keep communicating with each other to keep the traffic high in the network. Also here the application runs as it uses to do without any interruption and without any malfunction. Table 6.2 provides some statistics on the implementation on VirtexII-1000 and VirtexII-6000. The implementation of the TLC on a 3×3 DyNoC is shown in figure 6.20.

As we can see, the size of the routers are very large, making the used of NoCs very difficult on FPGAs. This is somehow because the routing resources in the router are built on top the programmable resources available (LUT and routing matrices) in an FPGA. A direct access to the available resources in the FPGA will reduce the amount of the resources need to a

Table 6.2. TLC and CG Statistics

	VirtexII-1000	VirtexII-6000
A/M/S(TLC)	53% /32% / 100.2	8% / 5% / 100.2
A/M/S(CG)	47% 33% / 91.2	20% /11% / 121.9
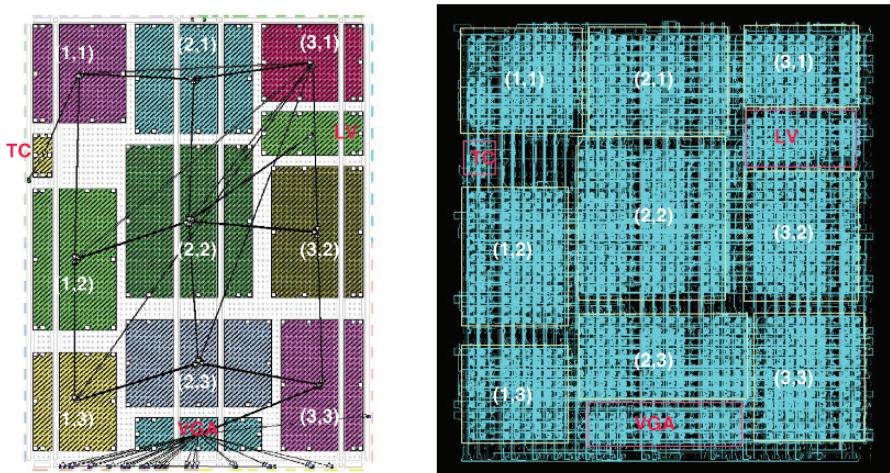
Figure 6.20.   DyNoC implementation of a traffic light controller on a VirtexII-1000

minimum. However, this can be done only if the vendors provide the necessary knowledge to control the resources on a very low level.

7.   Conclusion

In this chapter, we have addressed the communication paradigm needed to allow components dynamically placed at run-time on a reconfigurable devices to communicate with their environment. The most promising approach is the Network on Chip, which has attracted a lot of attention in the research community in the last decade. A large amount of work is available, each with its own level of complexity. The DyNoC is an extension of the NoC paradigm. In our opinion, it provides the best NoC-based approaches to solve the dynamic need of communication in temporal placement. We have also presented a circuit switching approach and show why it is not a good solution in temporal placement. However, the 1-D RMB circuit switching approaches can be used to connect a small number of components in predefined location on a 1-D reconfigurable device.

Chapter 7

PARTIAL RECONFIGURATION DESIGN *

Partial reconfiguration is one the prerequisite in reconfigurable computing, as it allows for swapping modules into and out of the devices without having to reset the complete device for a total reconfiguration. As we will see later in chapter applications, the possibilities offered by the partial reconfigurable devices in the implementation of system adaptivity are enormous. Unfortunately, very few devices on the market have the capability to be partially reconfigured to replace some modules, whereas the rest of the design keeps working. One of the few devices falling in the category of partial reconfigurable chips are the FPGAs from Xilinx, in particular those from the Virtex class.

Designing for partial reconfiguration on Xilinx device is a complex exercise that requires to be well prepared, technically, but also psychologically. This chapter is much like a tutorial for designing partial reconfigurable applications on the Xilinx Virtex FPGA-devices. Our goal in this chapter is to guide the designer in the design process for partial reconfiguration by providing the essential steps and materials in order for him to avoid long search and long introduction phases, which sometimes leads to a possible discouragement.

We therefore explained the three design flows used for the partial reconfiguration on Xilinx devices: The *JBits* approach, the *Modular Design Flow* and the *Early Access Design Flow*. Although the *JBits* approach apply for very few old Virtex FPGA, the modular design and the early access design flows support newer devices. We therefore explain those two design flows on the basis of working examples, whose sources can be downloaded from the book home page.

The development of partial reconfigurable applications on the Xilinx platform is done according to the *Modular Design Flow* provided by Xilinx in [227]. The modular design flow provides a guided flow on how to generate the constraints required by the placement and configuration tools for the generation of full and partial bitstreams representing modules to be downloaded at run-time into the device for a partial or for a full configuration.

Several tutorials for partial reconfiguration on the Xilinx devices exist to be downloaded from the internet. Despite their usefulness, almost all of them target only VHDL and Verilog designs. With the increasing interest in software-like hardware design language such as SystemC and Handel-C, the need for incorporating those languages in partial reconfiguration design flows is high. We therefore describe in this chapter how to incorporate Handel-C designs in a partial reconfiguration design flow.

Finally, to efficiently use the capability of partial reconfigurable chips, the platform that hosts the chip, i.e. the board, must be designed in such a way that the use of the flexibility provided by the reconfigurable chip can be maximized. This issue is addressed at the end of the chapter where we provide some guidelines in the design of partial reconfigurable platforms. As example of such platforms, we present the Erlangen Slot Machine, which was designed to allow the maximal use of partial reconfiguration.

1.   Partial Reconfiguration on Virtex Devices

Virtex FPGAs like other FPGAs from Xilinx are organized as a two-dimensional array of CLBs containing a certain amount of logic. They are configured with configuration data called a *bitstream*, which can be downloaded into the device via the configuration port.

The idea behind partial reconfiguration is to realize reconfiguration by only making the changes needed to bring the device in the desired state. Fragments of the complete bitstream, also called *packets*, are sent to the device to reconfigure the needed part of the device. A copy of the last configuration is maintained in a dedicated part of the processor memory, called the *configuration memory*. Partial reconfiguration is done by synchronization between the configuration memory and the device. Changes made between the last configuration and the present one is marked as dirty packets that are then sent to the device for partial reconfiguration. A packet is a sequence of (command, data) pairs of varying length that are used to read or write internal registers and configuration state data.

Virtex devices are organized in *frames* or column-slices that are the smallest unit of reconfiguration. Partial reconfiguration can then be performed by just downloading the configuration data needed to modify the part of the device needed. Up to the Virtex II Pro, a frame used to span a complete column, with the consequence that the reconfiguration of one component affects all the components, which share a common column with that component. From the Virtex 4 and upwards, the frames do not span a complete column any more,

but only a complete tile. Therefore, the reconfiguration of a component affects only those components in the same blocks, which share a common column with the reconfigurable module.

It is then the matter of the partial reconfiguration design flow to produce the partial bitstream, i.e. the set of data required to configure the needed frames and therefore to allow the replacement of a component on the chip. The extracted partial bitstreams are then used to move the device from one configuration state to the next one without reconfiguring the whole device.

Two possibilities exist to extract the partial bitstream representing a given module: A constructive approach that allows us to first implement any single component separately using common development tools. The so-developed components are then constrained to be placed at a given location using bounding box and position constraints. The complete bitstream is finally built as the sum of all partial bitstreams. This approach is the one followed by the modular design and the early access design flows. The second possibility consists of first implementing the complete bitstreams separately. The fix parts as well as the reconfigurable parts are implemented with components constrained at the same location in all the bitstreams, which differs from each other only on the reconfigurable parts. The difference of two bitstreams is then computed to obtain the partial bitstream needed to move from one configuration to the next one.

EXAMPLE 7.1 *Consider for instance the two designs of figure 7.1, where the full designs are named* Top1 *and* Top2. *The fix part consists of the module*
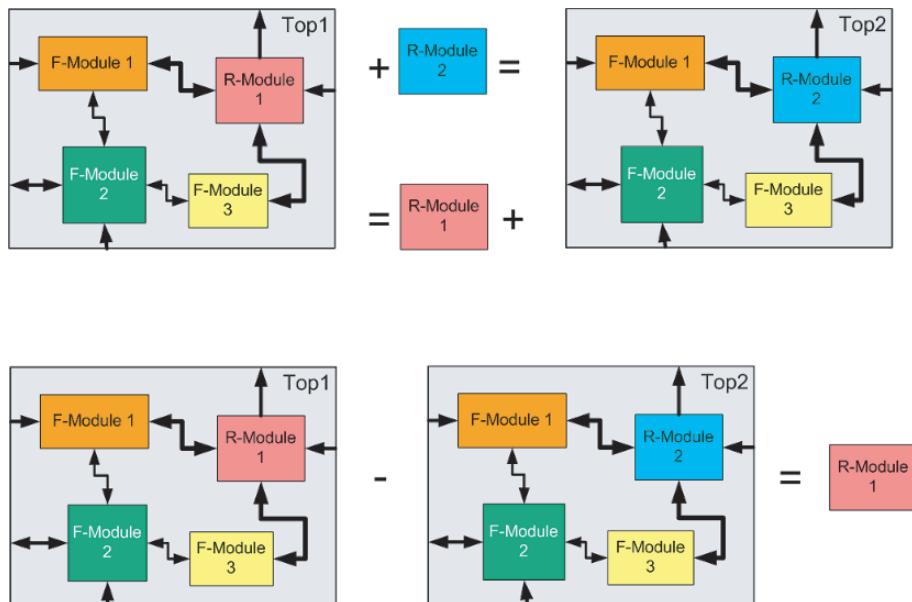
Figure 7.1. Generation of bitstreams for partial reconfiguration

F-Module 1, F-Module 2, *and* F-Module 3, *which are placed at the same lo-
cation in the two designs. In* Top1 *, the reconfigurable module is* R-Module 1,
*which is placed at the same location, with the same bounding box constraints
as the reconfigurable module* R-Module 2 *in* Top2.

The addition of R-Module 2 *to the fix parts of the two bitstreams generate
the full bitstream* Top2. *The same is true for* R-Module 2 *and* Top1. *Also,
as shown in the figure, adding the two bitstreams* Top1 *and* R-Module 2 *will
produce the bitstream* Top2, *because* R-Module 2 *will replace* R-Module 1 *in*
Top1. *The substraction of* Top2 *from* Top1 *produce the partial bitstream for*
R-Module 1.

2.   Bitstream Manipulation with *JBits*

JBits [102] is an application interface developed by Xilinx to allow the end
user to set the content of the LUT and make connections inside the FPGA
without the need for other CAD tools. *JBits* does permit the changes to be
made directly to the device, but to the configuration data.

The *JBits* API is constructed from a set of java classes, methods and tools
that can be used to set the LUT-values as well as the interconnections. This is
all what is required to implement a function in an FPGA. *JBits* also provides
functions to read back the content of a FPGA currently in use.

The content of an LUT can be set in *JBits* using the set function:

$$set(row, col, SliceNumber\_Type, value)$$

which means that the content of the LUT *Type* (*Type* is either the LUT F or
the LUT G of the corresponding CLB) in the slice *SliceNumber* (*SliceNumber*
can be 0 or 1) of the CLB in position *row* and *col* should be set to the value of
the array *value*. The Virtex LUTs have four inputs and 16 entries representing
the 16 possible values a 4-input function can take.

A connection is defined using the function connect:

$$connect(outpin, inpin)$$

This function uses the *JBits* routing capabilities to connect the pin *outpin* to
the pin *inpin* anywhere inside the FPGA. *outpin* and *inpin* should be one of the
CLB terminals.

Connecting the output of an LUT to a CLB terminal can be done with the
function.

$$set(row, col, terminal\_control, LUT\_output)$$

where *terminal_control* sets the correct terminal to be connected to the out-
put *LUT_output* of the corresponding LUT. *JBits* provides hundreds of prede-
fine cores such as adders, subtractors, multipliers, CORDIC Processor, encoder

and decoders, network modules, which can directly be used in designs. They can also be combined to generate more complex cores.

Although any function can be implemented entirely in *JBits* using the basic primitives previously described as well as the predefined library modules, it is usually not the best approach, because of the high complexity in building components from scratch. Instead, the bitstream substraction described in the previous section is used to extract the part of the bitstream needed at run-time for partial reconfiguration.

The approaches used to extract partial bitstream of the component that will be replaced at run-time, consist of implementing as many full bitstream, also called top-level, as required and perform the substraction among them to extract the differences. Those are the parts that will be used to configure the device later. Implementing the substraction using *JBits* is straightforward. The two bitstreams need to be scanned and compared for each element. Only the difference is copied in the resulting bitstream.

3. The Modular Design Flow

One of the main drawback of *JBits* is the difficulty to provide fixed routing channels for a direct connection between two modules. This is important because one must ensure that signals will not be routed on the wrong paths after the reconfiguration. Consider for instance the two full designs of figure 7.2 in which the fixed as well as the reconfigurable components are placed at the same locations. The connection F-Module 1 ↔ R-Module 1 is running on a different path than the connection F-Module 1 ↔ R-Module 2. On reconfiguration, the design will probably not work correctly.

One of the main contribution in the *modular design flow* [128] is the use of the so-called *Bus Macro* primitive to guarantee fixed communication channels
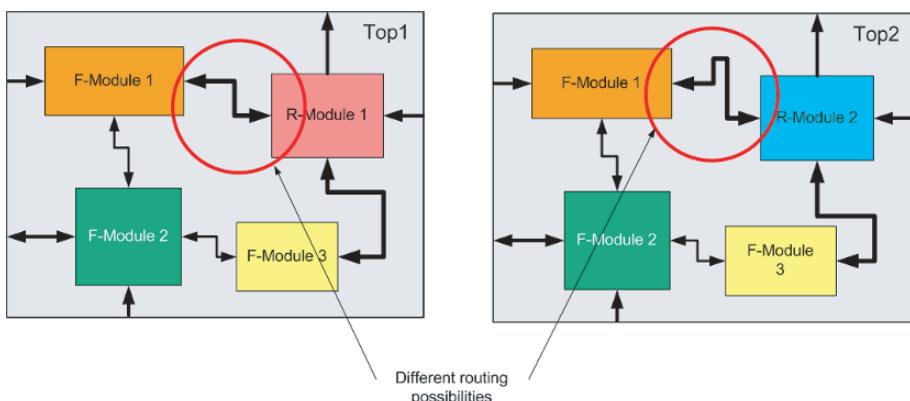
Figure 7.2. Routing tools can route the same signals on different paths

among components that will be reconfigured at run-time and the fixed part of the design.

The *modular design flow* however was not initially developed to support partial reconfiguration. It is an approach that allows several engineers to cooperatively work on the same project. The project leader identifies the components of the whole project, estimates the amount of resources that will be consumed by each component, defines the placement locations for the components on the device and lets the single parts be developed by the different engineers. Finally, the single developed parts are integrated into the whole design to build a workable circuit. The *modular design flow* is a 4-step method (design entry and synthesis, initial budgeting, active implementation and assembly) for which a viable directory structure must be used.

We recommend to use the ISE version 6.3, which is in our opinion the most stable version for implementing partial reconfiguration with the modular design flow.

3.1   Directory Structure

Xilinx recommends a structure of design directories as shown in figure 7.3:

- The *HDL Design* directory (HDL): This directory is used to store the HDL descriptions of the modules and that of the top level designs. The HDL description can be in VHDL, Verilog or any other hardware description language.

- *Synthesis* directory (synth): In this directory the team leader synthesizes the top-level design, which includes the appropriate HDL file, the project file, and project directories. In the same way, each team member will also have a local synthesis directory for his or her synthesized module.

- *Modules* directory (Modules): In this directory, the active implementation of individual modules is done in the respective module's subdirectory.

- *Implementation* directory (Top): It includes two subdirectories, one for the initial budgeting phase and one for the final assembly phase. In the *initial* directory, the team leader performs the initial budgeting operations for the corresponding top-level design. In *final assembly* directory, the team leader assembles the top-level design from the individual modules implemented and published in *PIMs* directory.

- *PIMs (Physically Implemented Modules)* directory (PIMs): The active implementation of individual module creates appropriate module directory in PIMs directory where implemented module files are copied.

We next focus on the single steps of the *modular design flow*:
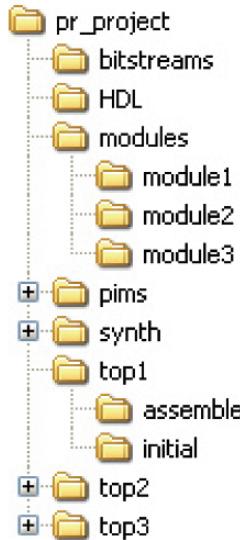
Figure 7.3. The recommended directory structure for the modular design flow

3.2 Design Entry and Synthesis

This step must be carried out before going to the modular design implementation step. In this step, the members of the team develop modules using a hardware description language (HDL) and synthesize them using synthesis tools. The team leader completes top-level design entry and synthesis before moving to the initial budgeting phase. Team members complete individual module design and synthesis before moving to the active module implementation phase for that module. However, the team members can work on their modules whereas the team leader is working on initial budgeting phase of modular design implementation. The restriction that must be done on the top-level design are the following:

- All global logic, including I/Os, must be included in the top-level description.

- All design modules are instantiated as *black boxes* in the top-level design, with only ports and port directions specified.

- Signals that connect modules to each other and to I/O pins must be defined in the top-level design.

Top-level design and individual modules are synthesized as described in the documentation of the synthesis tool selected. In synthesis properties, the *insert*

I/O pad settings must be enabled for top-level design and disabled for each submodule. Separate unique netlists will be created for each of the modules and for the top-level design. For each instantiated module, exactly the same module name must be used as its file name. Some synthesis tools such as the Xilinx Xst will not be able to match the module names specified in the top-level netlist to the module netlists.

3.3   Initial Budgeting

In this stage, the team leader assigns top-level constraints (pin locations, area constraints for each modules, timing constraints, etc.) to the top-level design, which results in *topX.ucf file (user constrain file)*. The same *topX.ucf file* is used for the active module implementation of individual modules.

Only the netlist for the top-level design file (topX.ngc, if compiled with Xilinx Xst) and the constraint file (topX.ucf) are copied to the *initial* directory in *implementation* directory of the corresponding top-level design.

The same top-level design name should be used for both files. The following command is then first executed

<div align="center">

ngdbuild -modular initial *design_name*

</div>

In this case *design_name* is *top.ngc*. In this step, the compiler **ngdbuild** produces two files, *design_name.ngd* and *design_name.ngo*. The *design_name.ngd* file is used during subsequent modular design steps, while *design_name.ngo* is not. The constraint editor that is used to edit design constraints such as clock periods, in the top-level design, can be invoked with the following command:

<div align="center">

constraints editor *design_name.ngd*

</div>

The following command invokes Floorplaner.

<div align="center">

floorplanner *design_name.ngd*

</div>

Using Floorplanner, modules sizes, module locations and information on module ports can be created. The modifications are copied in the top-level constraint file *topX.ucf*. This file can also be directly edited with the constraints as specified in the user manual. Constraints that must be set in the *.ucf* file, either by hand or by the floorplanner are the following:

1 Module area constraints:

- each module must have a minimum width of four-slices.
- a module's width is always a multiple of four slices (e.g. 4, 8, 12)

- the module bounding box must span the full height of the device.
- each module must be placed on an even four-slice boundary.

Besides those constraints, *area group* must be defined for each component and specific properties for the partial reconfiguration design flow must be added to the to the *area group* of components.

2 Input Output Block constraints: The IOB must be constrained within the *columnar space* of their associated reconfigurable modules. This condition is important to avoid the connection between a module and a pin to be destroyed during the reconfiguration of another component. Also, all IOBs must be locked down to exact sites.

3 Global logic constraint: All global logic such as clocks, power and ground lines must be constrained in the top level. No global logic should be locked in the module *topX.ucf* file.

4 Insertion of *Bus macros*

Partial reconfiguration is the process of transiting from one full configuration to a new full configuration using a partial configuration bitstream. This partial bitstream represents a component or a set of components present in the final bitstream, but not in the first one. The two bitstreams have a common fix part, which does not change. The reconfigurable parts are connected to the fix part by defining signal and connecting them to the ports. The path followed by the signals in two designs may be different because of the routing tools. On partial reconfiguration, the resulting application may not work anymore, if the signals connecting the fix and dynamic part are not using the same paths in the two designs. The role of the *bus macros* in the Xilinx FPGAs is to provide fix communication channels, which can be used by reconfigurable module. Those are *tri-state* lines that help to keep the signal integrity of modules being reconfigured. Those channels are not affected by the reconfiguration and the signals keep their integrity, thus ensuring a correct operation of the application.

The approach consists of connecting the two components at the two ends of a dedicated long line segment within the FPGA. The two components use *tri-state buffers* to define the direction of the communication. A *bus macro* is a construct that allows such connection to be specified in the design tool. Each *bus macro* occupies a 1-row by 8-column section of *tri-buffer* site space.

Besides the constraints specified, global-level timing constraints can be created for the overall design. The output of the initial budgeting phase is a *topX.ucf* file containing all placements and timing constraints. Each module is

implemented using this *topX.ucf* file, extended with the module-specific constraint requirements.

To finalize the initial budgeting phase, the following command is run in the initial subdirectory of the corresponding top-level design, after copying the files *topX.ucf* file and *topX.ngc* file to the same directory.

ngdbuild -p *device_type* **-modular initial** *topX.ngc*

3.4   Implementing Active Modules

During this phase, the team members implement the top-level design with only the *active* module expanded using the top-level *topX.ucf* file. Team members can implement their modules in parallel. Once each design has been synthesized, floorplanned and constrained, the implementation (mapping, place and route) of all modules (both fixed and partially reconfigurable) for the design can start. Each module will be implemented separately, but always in the context of the top-level logic and constraints. Bitstreams will be generated for all reconfigurable modules. This section describes the overview on how to independently implement each module.

The following instructions should be performed for each of the module in the design.

1 Copy the module implementation files to the active implementation directories for each module (*active/module_name*) under the module directory in which active module will be implemented:

   ■ Synthesized module netlist file *module_name.ngc*
   ■ The file *topX.ucf*, which the team leader created in the initial budgeting phase must be copied in the module local directory and renamed to *module_name.ucf*.

2 Build the module in active module mode as follows:

ngdbuild -uc *module_name.ngc* **-modular module -active** *topX.ucf* \\
            top_level_design_directory_path/topX.ngo

The output NGD file is named after the top-level design and contains implementation information for both the top-level design and the individual module. The -uc option ensures that the constraints from the local *topX.ucf* file are annotated.

3 Map the module to the library elements using the following command.

map *topX.ngd*

In this step, the logic of the design only with expanded active module is mapped.

4 The place and route of the library elements assigned to the module is done by invoking the placer and router with:

par -w *module_name.ncd top_routed.ncd*

The -w option ensures that any previous versions of the produced files *design_name_routed.ncd* are overwritten. If the area specified for the module cannot contain the physical logic for the module because it is sized incorrectly, then resizing must be done again and the .ucf file generated during the initial budgeting phase must be regenerated with the changes made. The entire initial budgeting phase should be performed again. This would then imply that new .ucf files would need to be copied to each module and that each module needs to be reimplemented. If everything went fine, then the module can be placed and routed at the correct location in the given bounding box.

5 Run trace on the implemented design to check the timing report to verify that the timing constraints are met.

trce *top_routed.ncd*

If necessary, the Floorplanner is used to reposition logic, if the module implementation is unsatisfactory, for example if it does not meet timing constraints. In this case the Map, place and route steps must be rerun again as described in the previous steps.

6 The last step in the active implementation is the publishing of the implemented module file to the central located PIMs directory, which was set up by the team leader. This step is performed using the command:

pimcreate pim *directory_path* **-ncd** *top_routed.ncd*

This command creates the appropriate module directory inside the PIMs directory. It then copies the local, implemented module files, including the .ngo, .ngm and .ncd files, to the module directory inside the PIMs directory and renames the .ngd and .ngm files to *module_name.ncd* and *module_name.ngm*. The -ncd option specifies the fully routed NCD file that should be published.

3.5   Module Assembling

Module assembling is the final phase of modular design. Here the team leader assembles the previously implemented modules into one top-level design. In this phase previously implemented modules, which were published to the PIMs directory, and the top-level design file are used. The resulting full design can be used to generate a bitstream. As before, we provide the necessary steps.

1 To incorporate all the logic for each module into the top-level design, run **ngdbuild** as follows:

> **ngdbuild -modular assemble -pimpath** *pim_directory_path topX.ngo*

Ngdbuild generates an *topX.ngd* file from the top-level *topX.ucf* file, the top-level .ngo file and each PIM's *topX.ngo* file.

2 The logic for the full design can then be mapped as follows:

> **map** *topX.ngd*

MAP uses the *topX.ncd* and *topX.ngm* files from each of the module directories inside the PIMs directory to accurately recreate the module logic.

3 The full design can then be placed and routed using the following command:

> **par -w** *topX.ncd topX_routed.ncd*

The place and route command, par, uses the *topX.ncd* file from each of the module directories inside the PIMs directory to accurately reimplement the module logic.

4 To verify whether timing constraints were met, the trace command must be invoked as follows:

> **trce** *top_routed.ncd*

Partial reconfiguration with the modular design flow can be performed as batch job using one or more scripts that contain the large amount of commands that must be invoked. An example of such scripts and design examples are available for download on the book's website. Also, tools such as Part-Y [71] provide a conformable graphical user interface on top of the modular design.

4. The Early Access Design Flow

With the introduction of the Early Access Design Flow for creating partially reconfigurable designs, Xilinx updated and enhanced the existing *Modular Design Flow* [227]. The package consists of a number of scripts that modify the user's ISE installation. Up to now the update is only viable with the exact ISE version 8.1i with service pack 1. It can be obtained by registered users from [226].

The main changes compared to the Modular Design Flow concern usability and the relaxation of some rigid constraints for partial reconfiguration designs. Also, new bus macros improve constraint setting and allow more possibilities for reconfiguration.

First, Virtex-4 platform support is added to the PR tools. Besides the known Spartan 3, Virtex-II and -II Pro, this enables users to deploy the current top product of Xilinx's linecard.

Partially reconfigurable areas may now also be defined block-wise and do not have to span over the full chip height. This provides the possibility, for Virtex-4 devices and later to reconfigure the tiles rather than whole columns. The Virtex-II and Spartan FPGAs, in contrast, physically lack this ability.

When a small region is reconfigured, the whole column is written, but the configuration controller on the FPGA checks whether the reconfiguration would actually change the content of the configurable logic block and perform reconfiguration only where changes are necessary. The reconfiguration will not touch blocks that would remain unchanged at all as depicted in figure 7.4.
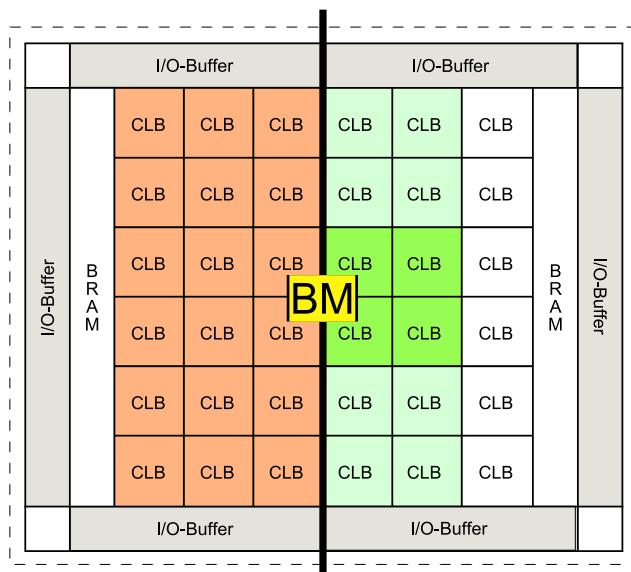
Figure 7.4. Limitation of a PR area to a block (dark) and the actual dimensions (light)
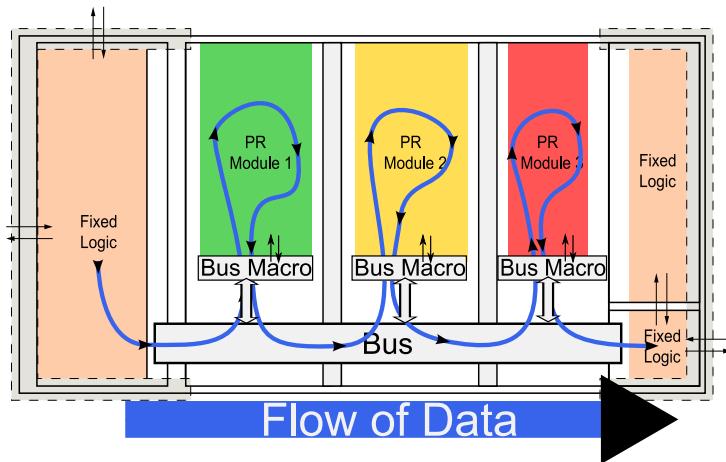
Figure 7.5.   Scheme of a PR application with a traversing bus that may not be interrupted

With this, applications may also be realized that require a bus that is placed through the whole width of the chip and will not be destroyed when reconfiguration occurs somewhere in between. Figure 7.5 shows an example of the design for a video streaming application that uses several stages of filters and a bus that transports the pictures from one module to the next. The single filters in the modules are partially reconfigurable and may be swapped without interrupting and stalling the bus. In this case, we will not want to alter the bus while configuring a module on the chain.

Another relaxation is that signals crossing partially reconfigurable blocks without interacting will not have to be passed through bus macros anymore. The routing algorithm is now capable of determining those signals and using longer range communication lines that will not be touched by a partial reconfiguration. This greatly alleviates timing constraints to be met and simplifies the designing process as a whole. Before, the developer had to build up the design, have the signals be routed and check for signals that had been guided through the partially reconfigurable area. Also, resources that used to lie within this block, but have to be used in another fixed module, would have had to be fed signals through some bus macros. Both cases, given the signal, will not exchange information with the PR area and are now fortunately handled by the new design flow. Now, traversing signals must not be guided through a bus macro, a tedious handwork.

4.1   New directory structure

The Early Access Tools now support a modified, more sensible directory structure for PR projects as depicted in figure 7.6. There, the folders' contents are presented in the order of usage in the design flow.

```
pr_project
    merges
        prm_a1
        prm_a2
    non_pr
        a1
        a2
    reconfigmodules
        prm_a1
        prm_a2
    static
    synth
        base
        prm_a1
        prm_a2
        top
    top
```
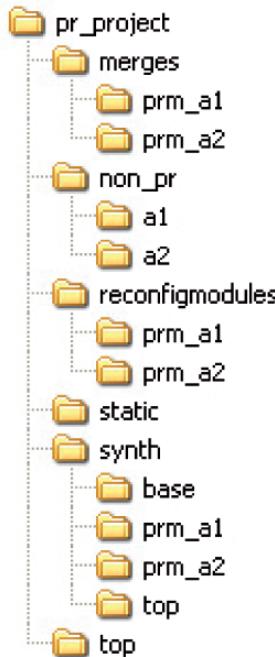
Figure 7.6. Improved directory structure for the Early Access Design Flow

■ The *non_pr* folder should contain a working version of every combination of modules that can be later used for partial reconfiguration. Although not required, it is *strongly* recommended to verify the functionality of the different variants of the design before trying to do any partial reconfiguration!

■ *synth* holds the HDL-definitions of the modules. There should be at least one subfolder for every partially reconfigurable module, a folder *base* for all the fixed modules and *top* for the top-level design. The modules are then synthesized herein. *Synth* thus replaces the folders hdl and synth from the Modular Design Flow.

■ The initial budgeting phase on the top-level module is done in *top* using the synthesized netlist from *synth/top*, the UCF and the Bus Macro definition files.

■ Initial budgeting for the static modules is accomplished in *static*.

■ The activation phase takes place in the *reconfigmodules'* subdirectories for each partially reconfigurable module.

- Finally, the design is assembled in *merges* and full and partial bitstreams are generated.

4.2   Design Entry and Synthesis

Except the slightly changed locations because of the modified directory structure, the general implementation work stays the same as with the *Modular Design*. Thus, top-level modules contain

- all global logic, including I/Os,

- all design modules, instantiated as *black boxes* with only ports and port directions specified and

- signals that connect modules to each other and to I/O pins.

Again, top-level modules include no logic except clock generation and distribution circuits.

4.3   Bus Macros

Xilinx ships a new set of Bus Macros to go with the *Early Access* package. They are now distinguished between device type, direction, synchronicity and width. They are now directed and not deployable in any direction and either clocked to run synchronously with the data or unclocked as the old bus macros. The naming convention yields to terms such as busmacro_*device_direction_synchronicity_width.nmc* where
device can be

xc2vp	–	for a Virtex-II Pro,
xc2v	–	for a Virtex-II or
xc4v	–	for a Virtex-4 FPGA.

direction is either one of

r2l	–	right-to-left
l2r	–	left-to-right
b2t	–	bottom-to-top (Virtex-4 only)
t2b	–	top-to-bottom (Virtex-4 only).

synchronicity may be

sync	–	synchronous or
async	–	asynchronous

width is either

wide	–	spanning four CLBs or
narrow	–	reaching over two CLBs.

A sample name is `busmacro_xc2vp_r2l_async_narrow.nmc`.
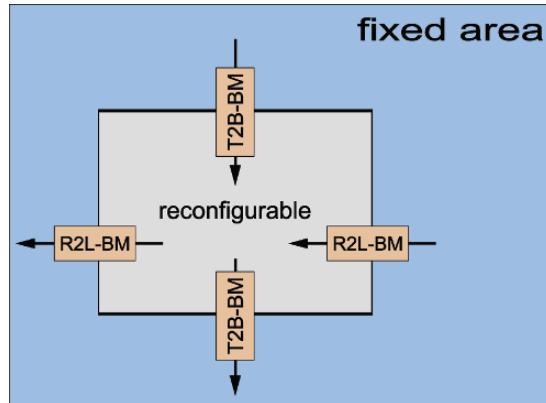
Figure 7.7. Usage of the new EA bus macros

The direction of the macro is to be seen geographically: right-to-left means data flowing from the right side of a border to the left side. If the right side is inside a partially reconfigurable area, the bus macro is an output bus macro, otherwise it is an input bus macro. The same applies to the other cases with left-to-right and the bottom-to-top and top-to-bottom bus macros. The usage is shown in figure 7.7.

Figure 7.8 shows the difference between a wide- and a narrow-type bus macro. The classification concerns only the range of CLBs that one of these bus macros bridges. Both kinds offer eight signal lines to cross a boundary.

The unoccupied CLBs between wide bus macros can be used by user logic or other bus macros. They can also be nested up to three times in the same y-row as depicted in figure 7.9. With the old bus macros, only four signals could traverse the boundary at this position whereas the new macros put through up to 24 lines. The three staggered bus macros do not have to be of the same type or direction.

Especially during the reconfiguration, signals from partially reconfigurable modules might toggle unpredictably. The new bus macros now provide versions that may therefore be disabled when appropriate to output a stable zero instead of the fluctuant signals.

4.4 Initial Budgeting

For the initial budgeting as well as for the activation phase, similar steps have to be taken as with the *Modular Design*. The UCF generation will have a different outcome as area constraints may now be set more freely and bus macros have to be deployed more carefully concerning their type and direction.

(a)

(b)
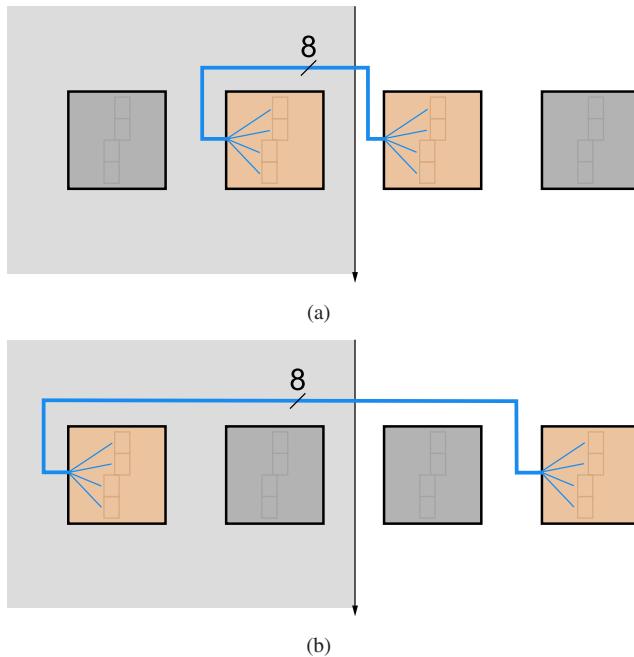
Figure 7.8.   Narrow (a) and wide (b) bus macro spanning two or four CLBs
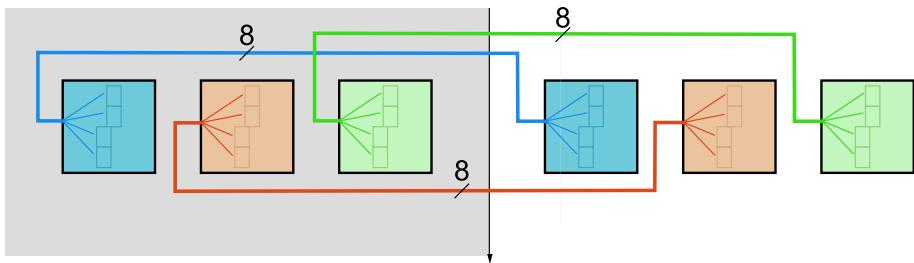
Figure 7.9.   Three nested bus macros

The placement rules are generally defined as follow:

■ the *X*-coordinate of the CLBs on which the bus macros is locked to, has to be divisible by two

■ the *Y*-coordinate has also to be divisible by two

■ Virtex-4 devices require coordinates divisible by four because of the location of block RAMs and DSPs

- bus macros have to traverse a region boundary. Each of the endpoints has to completely lie in a different area.

Further actions for the Initial Budgeting such as setting the timing constraints follow the same guidelines as the *Modular Design*. One must keep in mind that signals that will not interfere with a module can cross that module without having to run through a bus macro.

To close the Initial Budgeting Phase, the top-level design has to be implemented. In directory *pr_project/top*, the command

<div align="center">

ngdbuild -uc *system.ucf* **-modular initial** *system.ngc*

</div>

has to be invoked. *pr_project/static* now budgets the static portion of the design. The *.nmc* files for the used bus macros have to be placed in this directory. The necessary commands are

<div align="center">

ngdbuild -uc *../top/system.ucf* **-modular initial** *../top/system.ngc*

map *system.ngd*

par -w *system.ncd sytem_base_routed.ncd*

</div>

The routed outputs are placed in *pr_project/top* resp. *pr_project/static*. The process yields another file, *static.used*, that contains a list of routes within the partially reconfigurable regions that are already occupied with static design connections. This is used as an input for the placement and routing of the partially reconfigurable modules later on. The file allows for guiding signals through PRMs without using bus macros.

4.5 Activation

In the Activation Phase, the partially reconfigurable modules are implemented. First, the previously generated *static.used* is copied to each of the modules directories (e.g. *pr_project/ reconfigmodules/ prm_a1*) and renamed to *arcs.exclude*. Each module is then run through the known sequence of **ngdbuild**, **map** and **par** as follows for the *prm_a1*. In directory *pr_project/ reconfigmodules/ prm_a1*, do the following:

<div align="center">

ngdbuild -uc *../../top/system.ucf* **-modular module -active** \\
prm_component_name *../../top/system.ngc*

map *system.ngd*

par -w *system.ncd prm_des_routed.ncd*

</div>

Note that *prm_component_name* is the component name and not the instance name of the module. **par** automatically looks for and incorporates the *arcs. exclude* file.

4.6   Final Assembly

The last step is to assemble the static and reconfigurable modules into the top-level design.   This work is greatly facilitated by two new scripts: *pr_verifydesign* and *pr_assemble*.   For this, a routed top-level design, a set of the static modules and one PRM design have to be copied to a separate subdirectory for each PRM under *pr_project/merges*. For every PRM (e.g. in *pr_project/merges/prm_a1*), the following has to be executed:

> **pr_verifiydesign** *sytem_base_routed.ncd   prm_a1_routed.ncd*
>
> **pr_assemble** *sytem_base_routed.ncd   prm_a1_routed.ncd*

It should be checked whether the final netlists (e.g. *pr_project/merges/prm_a1/ static_routed.ncd*) meet timing constraints.   The timing analysis is performed with the *.pcf* file for the static design.

The partial and full bitstreams to reconfigure the targeted FPGA are then found in *merges*. Before using them, the developer has to make sure that the assembly process produced proper files by examining the *.summary* files generated.

4.7   Merging Multiple Partially Reconfigurable Areas

To merge more than one partially reconfigurable region in one design, an iterative invocation of `pr_verifydesign` and `pr_assemble` is needed. With each iteration, a new partially reconfigurable module is merged into the existing overall design.

The corresponding actions are

 1 *Merge each PRM of one reconfigurable area with the static design*
    `pr_verifydesign` and `pr_assemble` generate intermediate designs with just the one PRM contained and partial bitstreams that can be used to reconfigure the FPGA at this particular region afterwards.   The necessary commands are

> merges/prmA/**pr_verifiydesign** *sytem_base_routed.ncd* \\
> *prm_a1_routed.ncd*
>
> merges/prmA/**pr_assemble** *sytem_base_routed.ncd   prm_a1_routed.ncd*
>
> merges/prmA/**pr_verifiydesign** *sytem_base_routed.ncd* \\
> *prm_a2_routed.ncd*
>
> merges/prmA/**pr_assemble** *sytem_base_routed.ncd   prm_a2_routed.ncd*

2 *Merge additional modules for other partially reconfigurable areas*
 For every additional region, the output of the previous proceeding is taken
 as input: the partially reconfigurable module `b1` is to be merged with the
 combined `a1` and base design.

<div align="center">

merges/prmB1withA1/**pr_verifydesign** \\
../../prmA/prm_a1/base_routed_full.ncd \\
../../reconfigmodules/prm_b1/prm_b1_routed.ncd

merges/prmB1withA1/**pr_assemble** \\
../../prmA/prm_a1/base_routed_full.ncd \\
../../reconfigmodules/prm_b1/prm_b1_routed.ncd

</div>

Every combination of partially reconfigurable modules have to be merged
together. This requires for two partially reconfigurable regions accordingly:

<div align="center">

merges/prmB1withA2/**pr_verifydesign** \\
../../prmA/prm_a2/base_routed_full.ncd \\
../../reconfigmodules/prm_b1/prm_b1_routed.ncd

merges/prmB1withA2/**pr_assemble** \\
../../prmA/prm_a2/base_routed_full.ncd \\
../../reconfigmodules/prm_b1/prm_b1_routed.ncd

merges/prmB2withA1/**pr_verifydesign** \\
../../prmA/prm_a1/base_routed_full.ncd \\
../../reconfigmodules/prm_b2/prm_b2_routed.ncd

merges/prmB2withA1/**pr_assemble** \\
../../prmA/prm_a1/base_routed_full.ncd \\
../../reconfigmodules/prm_b2/prm_b2_routed.ncd

merges/prmB2withA2/**pr_verifydesign** \\
../../prmA/prm_a2/base_routed_full.ncd \\
../../reconfigmodules/prm_b2/prm_b2_routed.ncd

merges/prmB2withA2/**pr_assemble** \\
../../prmA/prm_a2/base_routed_full.ncd \\
../../reconfigmodules/prm_b2/prm_b2_routed.ncd

</div>

With the iterations, several files that will be created are equivalent to each other. In this case, for partially reconfigurable area B

```
prmB1withA1/prm_b1_blank.bit,
prmB1withA2/prm_b1_blank.bit,
prmB2withA1/prm_b1_blank.bit and
prmB2withA2/prm_b1_blank.bit
```

are equal. Equivalent bitstreams for module `b1` are

```
prmB1withA1/prm_b1_routed_partial.bit and
prmB1withA2/prm_b1_routed_partial.bit
```

and for module `b2`

```
prmB2withA1/prm_b2_routed_partial.bit and
prmB2withA2/prm_b2_routed_partial.bit.
```

Four variations of the full bitstream are generated representing the four combinations of the two partially reconfigurable modules in this example:

```
prmB1withA1/base_routed_full.bit,
prmB1withA2/base_routed_full.bit,
prmB2withA1/base_routed_full.bit and
prmB2withA2/base_routed_full.bit
```

In practice, only one full bitstream is needed to load the FPGA. The reconfiguration can then be conducted with the partial bitstreams. Which full bitstream to take is incumbent on the user.

5.   Creating Partially Reconfigurable Designs

This chapter gives a detailed description of the steps to take when recreating a design to be partially reconfigurable. The next part summarizes two sample designs to explain the single tasks to accomplish. The understanding of this guide depends on a profound knowledge of VDHL, ISE and EDK. To get a quick glance of VHDL, see [208], for an in depth introduction consult, e.g. [131]. Introductions to ISE and EDK are provided in [229] and [230]. For the *Early Access Reconfiguration Flow* software to work, it is crucial to deploy the correct version of ISE. The current issue of the update package requires ISE

8.1.01i (Service Pack 1) and updates it to ISE 8.1.01i PR8 resp. PR12. The used EDK system has version 8.1.02i.

5.1   *Animated Patterns*

The sample design *Animated Patterns* is a partially reconfigurable pattern generator for the XUP development board [228]. It is based on a project taken from a previously published tutorial [40]. The design creates a simple moving pattern and directs it out to a VGA-compatible screen. Depending on the current pixel position that is drawn on the monitor, the reconfigurable module `Produce Color` evaluates the color to be set for the next pixel. The system with its two areas, reconfigurable and fixed, is depicted in figure 7.10.
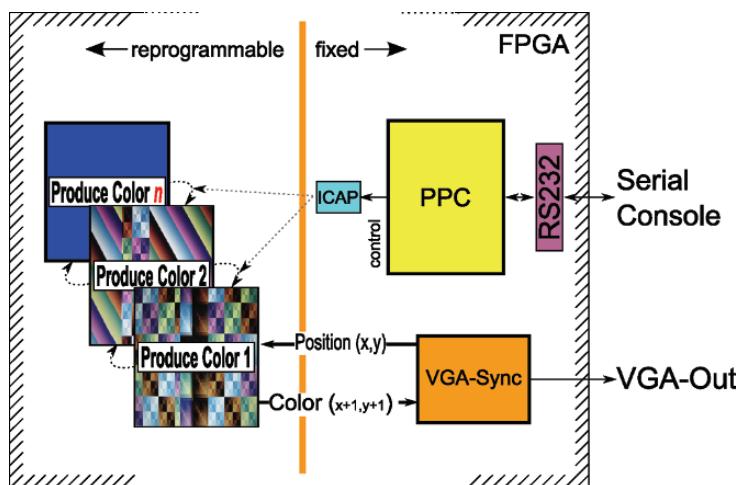
Figure 7.10.   Scheme of *Animated Patterns*

Partially reconfiguring the device at run-time here means exchanging `Produce Color` while the output to the monitor is continuing uninterruptedly. Control for the user is provided through a simple C-program accessible over a serial terminal. If a new partial bitstream is chosen, the program loads the necessary partial bitstream from a Compact-Flash card connected to the development board and transmits the data to the *ICAP*, the *Internal Configuration Access Port*, to reconfigure the device. The *ICAP* allows certain families of FPGAs[1] to be reconfigured from within the FPGA by a processor, in this case the PowerPC sitting in the Virtex II Pro on the XUP development board itself. The design can be found on the book's Web page.

---

[1]Currently, the Xilinx Virtex-II, -II Pro, Virtex-4 and Virtex-5.
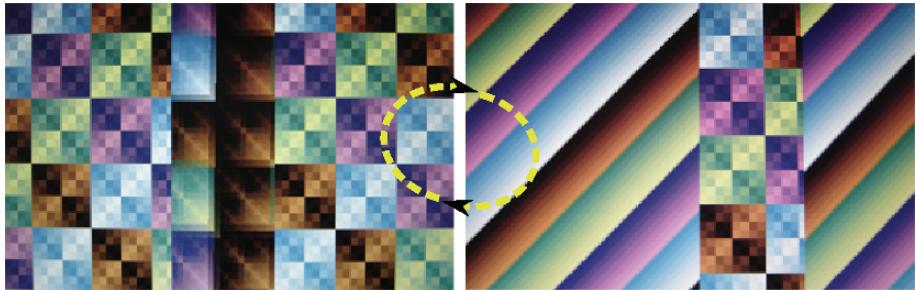
Figure 7.11.   Two patterns that can be created with modules of *Animated Patterns*. Left: the middle beam is moving. Right: the diagonal stripes are moving

5.2   *Video8*

The second sample design is called *Video8* and is used for the code and command execution examples in the following guide. It deploys the Digilent VDEC1 extension card [70] for the Virtex-II Pro XUP Development Board [228] to digitize an analog video stream from a camera. The extension card is connected to the XUP via a proprietary slot and is technically connected through an I^2C Bus.

As shown in figure 7.12, the data stream is taken into the system by *video_in*. The module converts to the RGB color space and hands over to a filter, employing, e.g. a Sobel-implementation for edge detection. This filter module will be
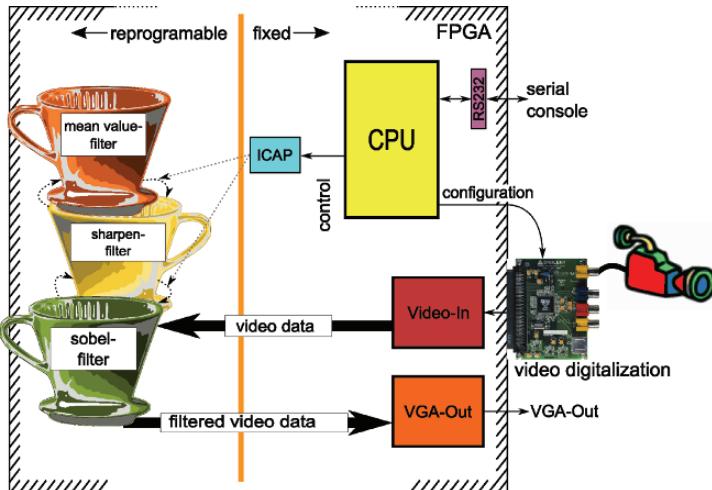
Figure 7.12.   Scheme of *Video8*

reconfigurable at the end of this tutorial. The filtered data are then processed in *vga_out* to be sent to a VGA compliant screen.

The embedded hardcore processor on the FPGA is used to configure the video digitizer card and the ICAP on start up. It is then used to run a simple program to choose which partial bitstream to load next. The input and output is brought to the user's console on a remote PC through the serial port.

5.3   Measures to take for PRM creation

Initially, a completely working project should mark the starting point for the creation of a partially reconfigurable design. The initial project *Video8_non_pr* marks the starting point for the instructions to follow.

1. **Move partially reconfigurable parts to the top-level module**
   Partially reconfigurable modules may not be submodules to other than the top-level module. To move its instatiation to the top-level, generally two approaches are possible:

   ■ Reconstruct the initial design to place the reconfigurable parts and also therewith connected module instantiations in the top-level. On the one hand, the resulting code is much easier to understand and modules can be developed by different engineers according to the modular design flow straight forward. On the other hand, the facilitation that comes for example from using EDK for the basic design might be outweighed by the lot of work involved when extracting and reconnecting all the necessary modules. This militates all the more regarding the synthesis process. The placement tool places modules according to their connectivity with each other and the constraints given in the UCF not according to their actual position in the hierarchy. Thus, reconstructing the design might be useless as the placement is done independently.

   In the *Video8_non_pr* example, *video_in* and *vga_out* are directly affiliated to the partially reconfigurable module to be. As suggested in figure 7.13, these two modules at least would have to be instantiated in the top-level module. Additional difficulties arise from the use of the OPB to connect the filter module. Either the developer has to adapt the filter module and provide other means of connection to the rest of the system or the OPB has to be brought up to the top-level as well. This again affords work and other modules to be changed accordingly.

   ■ The other solution is to only replace the old instantiations of the partially reconfigurable modules with instantiations on the top-level and amend the entities that formerly hosted the modules: Ports have to be added to redirect the input data to the partially reconfigurable modules up the module hierarchy to its new location at the top-level module and
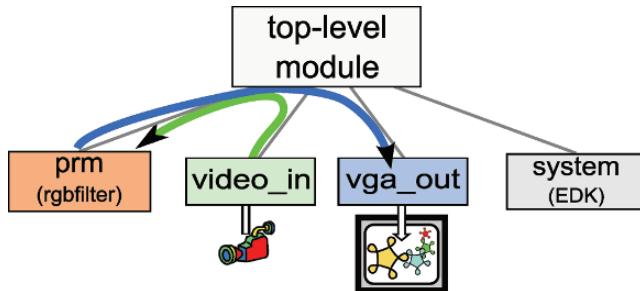
Figure 7.13. Reconstructing example *Video8* to place the partially reconfigurable part and connected modules in the top-level

the output data back down to be fed into the system again. Figure 7.14 depicts the new structures. The big advantage with this approach is the minimal adaption work to be done. An initial EDK design may be used as is and can easily be changed.

The aimed at module for partial reconfiguration, `rgbfilter`, is initially instantiated in entity `system`, the top-level module of the initial EDK design. To place `rgbfilter` in the top-level module according to the second approach, `video_in` has to be expanded as follows:

```
entity video_in is {
...

  //ports going out to the rgbfilter
```
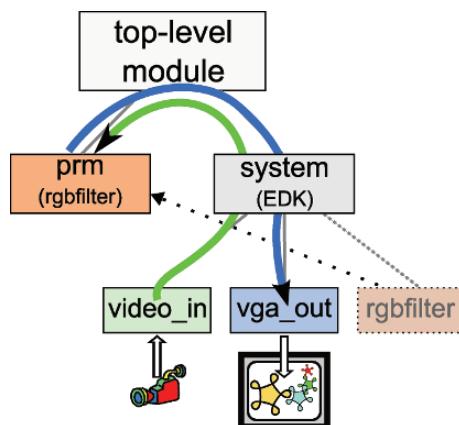

Figure 7.14. Moving a partially reconfigurable module to the top-level design on the example of *Video8*

```
LLC_CLOCK_to_filter : out std_logic;
R_out_to_filter : out std_logic_vector(0 to 9);
G_out_to_filter : out std_logic_vector(0 to 9);
B_out_to_filter : out std_logic_vector(0 to 9);
h_counter_out_to_filter : out std_logic_vector(0 to 9);
v_counter_out_to_filter : out std_logic_vector(0 to 9);
valid_out_to_filter : out std_logic;

//ports coming back from rgbfilter
R_in_from_filter : in std_logic_vector(0 to 9);
G_in_from_filter : in std_logic_vector(0 to 9);
B_in_from_filter : in std_logic_vector(0 to 9);
h_counter_in_from_filter : in std_logic_vector(0 to 9);
v_counter_in_from_filter : in std_logic_vector(0 to 9);
valid_in_from_filter : in std_logic
...
};
```

As the ports of a submodule cannot be connected directly to those of its hosting module, intermediate signals have to be introduced in the hosting module. The entity vide_in from the sample design has to be appended accordingly after defining the signals themselves:

```
...
-- signals out to the filter:
R_out_to_filter <= R1;
G_out_to_filter <= G1;
B_out_to_filter <= B1;
h_counter_out_to_filter <= h_counter1;
v_counter_out_to_filter <= v_counter1;
valid_out_to_filter <= valid1;
-- processed signals in from the filter:
R2 <= R_in_from_filter;
G2 <= G_in_from_filter;
B2 <= B_in_from_filter;
h_counter2 <=  h_counter_in_from_filter;
v_counter2 <=  v_counter_in_from_filter;
valid2 <= valid_in_from_filter;
...
```

The top-level entity must contain a similar construction.

2. **Base design rebuilding**
   Many designs for the Xilinx FPGAs take advantage of the use of EDK and

its advanced wizards. Complete functioning systems on chip including the software controlling can be created with a simple sequence of point-and-clicks. In many cases the static base system, including the microprocessor, the communication over a serial lines, RAM-controller and so forth, is created with EDK. The controlling software that runs on the microprocessor, may also be part of the system, leading to the creation of a bistream file in the `.elf` format. This is a file supported by the EDK, which describe the hardware configuration and the memory mapping with the software that will be executed on the processor.

Up to now, EDK does not support partial reconfiguration directly. The necessary phases for the design flow still have to be done manually. Systems implemented with EDK have to be exported to an ISE-format project, synthesized as required and then be plugged into the top-level module as a submodule.

In the case of *Video8_non_pr* a basic system consisting of the infrastructure for the PowerPC on the Virtex-II Pro, a RS-232 controller, a *Video-in* and a *VGA-out* module. The design is created so that the partial bitstreams to reconfigure the FPGA are loaded from a Compact Flash Card attached to the FPGA.

Because the reconfigurable module in *Video8_non_pr* has been included as a core (*Pcore*) in the EDK project, the additional port definitions previously described have to be applied to the `system` entity generated by the EDK as well. This can be easily accomplished by adding *External Ports* to the project. These changes will be written to the project's MHS file similar to the following sample lines:

```
PORT LLC_CLOCK_OUT = VIDEO_IN_PR_0_LLC_CLOCK_to_filter, DIR = O
PORT R_in = VIDEO_IN_PR_0_R_out_to_filter, DIR = O, VEC = [0:9]
PORT G_in = VIDEO_IN_PR_0_G_out_to_filter, DIR = O, VEC = [0:9]
PORT B_in = VIDEO_IN_PR_0_B_out_to_filter, DIR = O, VEC = [0:9]
PORT h_counter_in = VIDEO_IN_PR_0_h_counter_out_to_filter, \\
DIR = O, VEC = [0:9]
PORT v_counter_in = VIDEO_IN_PR_0_v_counter_out_to_filter, \\
DIR = O, VEC = [0:9]
PORT valid_in = VIDEO_IN_PR_0_valid_out_to_filter, DIR = O
PORT R_out = RGBFILTER_0_R_out, DIR = I, VEC = [0:9]
PORT G_out = RGBFILTER_0_G_out, DIR = I, VEC = [0:9]
PORT B_out = RGBFILTER_0_B_out, DIR = I, VEC = [0:9]
PORT h_counter_out = RGBFILTER_0_h_counter_out, \\
DIR = I, VEC = [0:9]
PORT v_counter_out = RGBFILTER_0_v_counter_out, \\
DIR = I, VEC = [0:9]
PORT valid_out = RGBFILTER_0_valid_out, DIR = I
```

2.1. *Additional Modules*

To make the best use of the EDK, some of the needed components can be added and connected in the EDK. A JTAG-PPC controller can help to debug components connected to the PowerPC core. The `opb_sysace` module is the necessary interface to the *SystemACE* device that is deployed for the management of a Compact Flash Card attached to the FPGA. `opb_timer` is used here to measure the time the reconfiguration takes. Finally, the `opb_hwicap` functions as an interface to the ICAP for self-reconfiguration.

2.2. *Bus Connections*

The needed bus connections for the additional modules can also be set up with EDK using the graphical user interface. This prevents a lot of errors originating from misspelling of variables and names. It also eases the task for less-experienced users. Thus, `opb_hwicap` and `opb_timer` have to be connected to the same opb on which the *SystemACE* module is connected to.

2.3. *Setting Address Ranges and Software Locations*

To ensure the correct function with the parameters set in the controlling software, some of the project's modules have to be set to a certain address as stated in table 7.1.

`docm_ctlr` and `iocm_ctrl` are the controllers for data and instructions to be stored in on-chip block RAMs. The program that controls the reconfiguration of the FPGA in this case is too large to fit in a smaller RAM area. It can generally be put into the DDR-RAM as well, but in this special design, the framebuffer for the VGA output is placed in the DDR-RAM, consuming a great share of the bandwidth of the bus it is connected to. Thus, placing the program in the DDR-RAM as well generates unwanted distortions in the VGA output.

2.4. *Rewriting the Control Software*

The program deployed to control the non-pr design has to be extended

Module Name	Start Address	Size
opb_sysace	0x41800000	64KB
opb_timer	0x41c00000	64KB
opb_hwicap	0x40200000	64KB
docm_ctlr	0xe8800000	32KB
iocm_ctrl	0xffff8000	32KB

Table 7.1. New address ranges to be set in EDK

by the initialization of the ICAP module and a mean to choose which partial bitstream to load next.

For the sample design, the control program `main.c` contains just the initialization of the video decoder card. The initialization routine for the ICAP and the SystemACE has to be added as well as a menu that prompts the user for the next partial bitstream.

2.5. *Export to ISE*

The generated design has to be synthesized to be a submodule to the top-level design. For this, EDK supports the export of a project to the ISE where detailed options for the synthesis can be set. Alternatively, the module can be synthesized using `xst`. To export the project in `Project|Project-Options|Hierarchy and Flow`, the following settings have to be made as follows:

- Design is a sub-module with name `system_i`
- use project navigator implementation flow (ISE)
- do not add modules to an existing ISE-project

3. **Creating a Top-Level Module**

To integrate the different modules, a top-level module has to be created. It should contain just module instantiations and clocks. Here, I/O-buffers are put up and bus macros are instantiated. Input and output ports are simply directed through the appropriate buffer instance. Bidirectional `inout`-signals are split up in in-, out- and a tristate control signals, using three corresponding buffers.

Instances have to be named, especially those that will be static and reconfigurable because their definition is placed in the UCF and the connection to the design is established over the instance name.

The partially reconfigurable module `rgbfilter` requires bus macros for signals R,G and B, `v_` and `h_counter` and `valid` from and to the `rgbfilter`. Tools such as *PlanAhead* or *FPGA-Designer* can facilitate this work. For detailed information on how and why to use bus macros see sections 3.3 and 4.3. An example of the resulting UCF can be found on the book web page.

4. **Synthesis**

The synthesis of the partially reconfigurable project takes place in the subfolder `synth` of the project base folder. Here, corresponding subfolders for static and reconfigurable modules and the top-level module can be found. It should be mentioned again that any submodules, including the partially reconfigurable ones, have to be synthesized without automatically adding I/O-buffers. Thus the option `-IOBUF` has be set to `no`. I/O-buffers only have to be included in the top-level module. The exported project contains the

system as a submodule `system_i` of type `system` and should already include the needed ports for the data flow to and from the reconfigurable module. `system_i` is ordered under a generated super module `system_stub` and can therefore directly be included in a top-level module. These ports should be created by EDK when entering and wiring *external ports* as described above. If not so, the generated HDL files describing and instantiating entity `system` resp. `system_i` (`system.vhd` resp. `system_stub.vhd`) have to be extended with the necessary ports.

For the sample design folders, the reconfigurable modules (`mod_*`) as well as a `top` and an `edk` directory must be created. There is no separate static module, except the EDK system. Therefore, no `static` folder is required. The ports to add to entity `system` are the same as when changing the former super module of the one that should be partially reconfigurable.

5. **Appending the UCF** Taking a UCF built for example from EDK, several amendments have to be made to let the partial reconfiguration work. As stated before, the definition of the area to place the partial and static blocks and the location of the bus macros have to be declared in the UCF.

To set a region to be partially reconfigurable, the UCF is extended by lines such as

```
INST "reconfig_module" AREA_GROUP = "pblock_recon_module";
AREA_GROUP "pblock_recon_module" RANGE=SLICE_X30Y64:SLICE_X45Y95;
AREA_GROUP "pblock_recon_module" MODE=RECONFIG;
```

With this, the reconfigurable module `reconfig_module` is associated with the area group `pblock_recon_module` whose bounding box is defined by the rectangle between slices `X30Y64` and `X45Y95`. It is then marked reconfigurable. A fixed part like a basic system created with EDK, can be set with:

```
INST "system_i" AREA_GROUP = "pblock_fixed_area"; AREA_GROUP
"pblock_fixed_area" RANGE=SLICE_X46Y16:SLICE_X87Y149;
```

Omitting `MODE=RECONFIG` tells the parser that the block will be fixed. Although the UCF is a simple text file and can be edited manually, tools such as *PlanAhead* can be a big help with their architecture-dependant graphical user interfaces.

6. **The Build Process**
The build process starts with the synthesis of all the involved modules. The submodules should not include I/O-buffers, which must all be included only in the top-level module. The initial budgeting is partly done already, and must be complete now. All other phases, from the activation to the final assembly, follow the steps described in section 4.

6.   Partial Reconfiguration using Handel-C Designs

The structure of the directory previously described is the same, if another HDL such as Handel-C or even SystemC have to be used. Using Handel-C, the goal is to provide the implementation of top-level designs and separately the implementation of each module in a much comfortable way. The Handel-C that must code for the entire system must be divided in code for modules, each with its own interface for connecting other modules. The integration is then done by connecting the modules together using signals in the top-level design as shown in figure 7.1. For each module to be reconfigured later, a separate top-level must be produced. The transition from one design to the next one will then be done later using either full reconfiguration or partial reconfiguration with modules representing the difference from one top-level to the next one.

We explain this modular design for Handel-C by means of one example provided below.

We consider a system containing an adder in its first configuration. The system can be partially reconfigured to act implement a subtractor in place of the adder. Both modules (adder and subtractor) have to be implemented separately. The implementation of the adder is provided in the following code segment in Handel-C.

```
void main() {
unsigned 32 res;

// Interface definition
interface port_in(unsigned 32 var_1
        with {busformat="B<I>"}) Invar_a();

interface port_in(unsigned 32 var_2
        with {busformat="B<I>"}) Invar_b();

interface port_out() Outvar(unsigned 32 Res = res
        with {busformat="B<I>"});

// Addition
res = Invar_a.var_1 + Invar_b.var_2; }
```

The first part of the code is the definition of the interfaces for communication with other modules and the second part realizes the addition of the input values coming from the input interface, and the output values are sent to the output interface. We need not provide the implementation of the subtractor, because it is the same like that of the adder, but instead of adding we subtract. Having implemented the two modules separately, each module will be inserted in a separated to-level. Up to the use of the adder or subtractor, the two top-levels

are the same. The design can now be reconfigured later to change the adder against the subtractor. Connecting the module in a top-level design is done as shown in the following code segment.

```
unsigned 32 operand1, operand2;

unsigned 32 result;

interface adder(unsigned 32 Res)

my_adder(unsigned 32 var_1 = operand1,
         unsigned 32 var_2 = operand2)

with {busformat="B<I>"};


void main() {
operand1 = produceoperand( 0);

operand2 = produceoperand( 3);

result = my_adder.Res;

dosethingwithresult(result);
}
```

According to the top-level design being implemented, the adder will be replaced by a subtracter. The next question is how to keep the signal integrity of the interfaces. As there is no bus macro available in Handel-C, bus-macros provided by Xilinx must be used as VHDL-component in a Handel-C design. For instruction on integrating a VHDL code in Handel-C, consult the Celoxica Handel-C reference manuals [125]. Bus Macros are provided with the Xilinx application note on partial reconfiguration [128] [227]. Before setting constraints on Handel-C design, the design has to be compiled first. Afterwards, the resulting EDIF-files must be opened with any text-editor and the longest pattern that contains the name of the module as declared in the top-level module is selected. This is useful because the Handel-C compiler generates EDIF-code and automatically adds some characters to the module name used in the original design. If the original module name is used, it will not be recognized in the following steps of the Modular Design Flow. With the EDIF for the modules and that of the top-level designs, we now have all what we need to run the modular design flow explained in section 3.

7.   Platform design

One of the main factors that have hindered the implementation of on-line placement algorithms developed in the past and presented in chapter 5, is the development process of reconfigurable modules for the Xilinx platforms. As we have seen in the previous section, partial reconfiguration design is done with many restrictions, which make a systematic development process for partial reconfiguration very difficult. Each module placed at a given location on the device is implicitly assigned all the resources in that area. This includes the device pins, clock managers and other hard macro components such as the embedded multipliers and the embedded BlockRAMs.

As illustrated in figure 7.15, a module using resource outside its placement area must use connection running through other modules to access its resources. We call those signals used by a given module and crossing other modules *feed-through signals*. Using *feed-through* lines to access resources have two negative consequences:

■ *Cumbersome design automation*: Each module must be implemented with all possible feed-through channels needed by other modules to reach their resources. This problem was already identified in section 1, where dedicated channels are preconfigured on the device to be used by modules at run-time. Because we only know at run-time which module needs to feed the signal through, many channels reserved for a possible feed-through become redundant.

■ *Non-relocation of modules*: Modules accessing resources, like pins in a given area of the chip, are no more relocatable, because they are compiled for fixed locations where a direct access to their pins must be available.
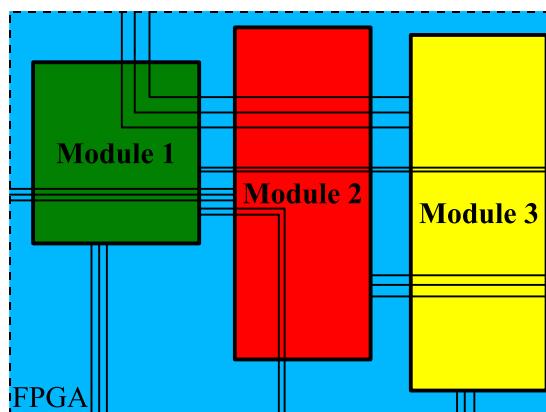
Figure 7.15.   Modules using resources (pins) not available in their placement area (the complete column) must use *feed-through* signals to access those resources

Most of the FPGA platforms available on the market do not provide solution to the problems previously mentioned. Many systems on the market offer various interfaces for audio and video capturing and rendering, for communication and so forth. However, each interface is connected to the FPGA using dedicated pins in a fix location. Modules willing to access a given interface such as the VGA must be placed in the area of the chip where the VGA signals are available such that they can be assigned those signals among other resources. This makes a relocation of modules at run-time impossible. Relocation provides the possibility to have only one bitstream for a module representing the implementation of that module for a given location. At run-time, the coordinate of the module is modified to match the location assigned to the module by the on-line placer. Without relocation, each module must be compiled for each possible position on the device where it can be later placed. A copy of the component bitstream must therefore be available for each location on the chip. The amount of storage needed for such a solution does not make such an approach attractive.

A research platform, the Erlangen Slot Machine (ESM) that we next present, was developed at the university of Erlangen to overcome the drawbacks of existing platforms, and therefore allow unrestricted on-line placement on an FPGA platform. Through the presentation of the ESM, we also hope to highlight some of the requirements in the platform design for reconfigurable computing.

The goals that the ESM-designers had in mind while developing a new platform was to overcome the deficiency of existing FPGA-Platforms by providing

- A new highly flexible FPGA Platform in which each component must not be fixed all the time at a given chip location.

- A suitable tool support which goal is to ease the development process of modules for run-time reconfiguration and communication and to make an efficient use of the architecture.

Although the tooling aspect is important, it is not part of the basic requirements for designing the platform itself. We therefore just focus on the architectural aspect of the platform design in the next sections.

7.0.1 Drawback of existing systems

The practical realization and use of partial and dynamic reconfiguration on the current existing FPGA-based reconfigurable platforms is highly restricted, because of the following factors:

1. **Limitation of reconfiguration on actual FPGAs:** Very few FPGAs allowing partial reconfiguration exist on the market. Those few FPGAs such

as the Virtex series allow only a restricted partial reconfiguration. As stated in section 1, up to the Virtex II Pro, the reconfiguration had to be done column wise meaning that loading a module in a given area will affect all the modules on top and below the module. From the Virtex 4 and upwards, the reconfiguration of a module affects only those components in the same blocks, which share a common column with the reconfigurable module. The block does not span the complete device column, but only a few number of lines.

2. **I/O-Pin-Problematic:** The design of existing FPGA platforms makes it more difficult to design for partial reconfiguration. Most of the existing platforms have the peripheral components such as Video, RAMs, Audio, ADC and DAC connected at fixed location on the device. This has the consequence that a module must be stuck on a given region where it will have access to his pins, and therefore making a relocation very difficult. Another problem related to the pin connection is that the pins belonging to a given logical group such as video, audio, etc... are not physically grouped together. On many platform, they are spread around the device. A module willing to use a device will have to feed many lines through many different components.

   This situation is illustrated on figure 7.16 for the Celoxica RC200 platform. To see are two modules among which a VGA module, which is implemented on the FPGA. The VGA module uses a large number of pins at the bottom part of the device and also on the right side. It is possible to implement a module without feed-through only on the two first columns on the left side. Most of the reconfigurable modules will need more than the two available column of resources to be implemented. Feed-through lines that
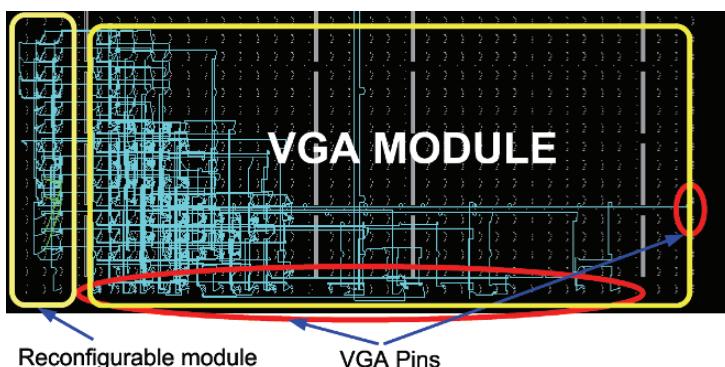
Figure 7.16.   Illustration of th pin problematique on the RC200-Board

run through the reconfigurable module must be used, in order for the VGA module to access the pins cover by the reconfigurable module.

Similar situation is not only present on the Celoxica boards. The XESS boards [56], the Nallatech boards [165], the Alpha boards [8] face the same limitations.

On the XF-Board [177] [211] from the ETH in Zurich, the connection to the peripherals are done on the side of the device. Each module accesses its peripheral through an operating system (OS) layer implemented on the left and right part of the device and between the component swapped in and out. This approach provides only a restricted solution to the problem, because all modules must be implemented with a given number of feed-through lines and interfaces to access the OS layer for communication with the peripheral. The intermodule communication as well as the communication between a module and its peripheral is done via buffers provided by the OS. This indirect communication will affect the performance of the system. Many other existing platforms such as the RAPTOR-Board [132], Celoxica RC1000 and RC2000 [43] are PCI systems, which require a workstation for operation. The use in stand-alone systems as it is the case in embedded is not possible.

3. **Intermodule communication:** One of the biggest problems not considered in the design of reconfigurable platform is the intermodule communication. Modules placed at run-time on the device may need to exchange some data among each other. This request of communication is a dynamic task because of the dynamism in on line placement. For module placed far from each other, it will not be possible to feed communication line through several modules to establish a connection. New paths are then necessary to allow a dynamic communication to take place at run-time.

The previous limitations were the primary motivation in the design of the Erlangen Slot Machine (ESM), whose concept we next present.

7.0.2   The Erlangen Slot Machine

The Erlangen Slot machine (ESM) is made upon a *BabyBoard* mounted on a *MotherBoard*. The separation of the system in two boards allows the *Baby-Board* that contains the reconfigurable module to be used on a wide variety of systems. For the integration of the ESM in a new platform, no redesign of the complete system must be done. Only a new *MotherBoard* must be provided according to the computational requirement in the new environment. A multimedia system for example will provide a *MotherBoard* with multimedia devices such as video and audio. In an automotive system, the MotherBoard will mostly contain sensor and actuators. The *BabyBoard* can be mounted on

a PCI *MotherBoard* in a computer system or on a stand alone *MotherBoard* in an embedded system. Actually, only one variety of *MotherBoard* is available, that is mainly targeted for applications in multimedia and signal processing. In the next, we provide a detailed description of the boards functionality.

7.0.3   The BabyBoard: Computation and Reconfigurable Engine

The reconfigurable engine of the ESM platform is a *BabyBoard*, which features an FPGA Virtex II-6000 FPGA from Xilinx, several SRAMs and a configuration circuit. Because of the restriction[2] in the reconfiguration of the Virtex FPGAs, the architecture was adapted to match the following:

- **Free relocation of modules:** Online placement of modules on a reconfigurable device, in this case the FPGA, is done by downloading a partial bitstream implementing a given task in the FPGA. The full and partial bitstreams represent circuits implemented on a given region of the device at compile time. In order to place a module in another location other than the location for which it was compiled, a relocation of the module must be done. The relocation of a given module therefore modifies the coordinates of all the resources used by the module in its previous location (the location in which the module was constrained at compile time) with the coordinates of the module in the new location. Relocation can be done only if all the resources are available and structured in the same way in the previous location and in the new location. This include the device pins used by the module. If the connection of a module with its peripheral is done via some pins fixed at a given chip location, the module must be constrained at that location. A relocation will be possible, but the module must feed signals through all the other modules between the pins and the new location.

  This problem is solved on the ESM by avoiding a direct connection of peripheral on the FPGA. As shown in figure 7.17, all the bottom pins from the FPGA are connected to an interface controller (crossbar). The Interface devices are connected to the interface controller as well. This makes it possible to establish any connection from one module to its peripheral via the crossbar. No matter where the module resides in the device, the crossbar can be set at run-time to connect the module to its peripheral.

- **Uniform repartition of resource:** The ESM like any other reconfigurable platform is primarily a parallel computing engine in which each module carries its own computation and communicates with other modules. Therefore a certain amount of resources must be allocated to each module for its operation, at least for a given period of time. Memory is very important in

---

[2]The reconfiguration can be done only column wise
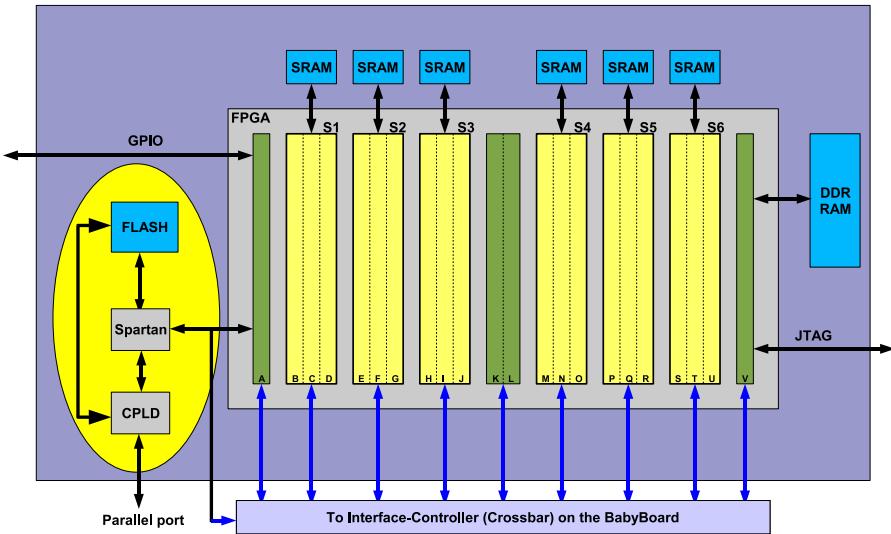
Figure 7.17. Architecture of the ESM-Baby board

application such as video streaming in which a given module must exclusively access a picture at time for computation. However, the capacity of the available BlockRAMs in FPGA is limited. External memory (SRAM or DRAM) must therefore be added to allow the storage of large amount of data by each module. To allow a module to exclusively accesses its memory, the SRAM are connected at the top part of the FPGA. In this way, a module will have its connection to its peripheral from the bottom part, and the top part will be used to temporally store the computation data. No module will need to feed its lines through the other modules for accessing its resources. According to the number of memory banks that can be connected on the top level, the device is then divided into a set of exchangeable slots, each of which have access to the SRAM on the top part and to the crossbar at the bottom.

The name *Erlangen Slot Machine* was chosen because it was developed at the University of Erlangen-Nuremberg in Germany, but mainly because of this organization in slots. This modular organization of the device simplifies the relocation, primary condition for a viable reconfigurable computing system. Each module moved from one slot to another one will find the same resources there. The architecture is illustrated in figure 7.17. The *Baby-Board* is logically divided into columns of 2 CLBs called *micro slots*. The *micro slots* are the smallest allocatable units in the system. Each module must therefore be implemented in a given number of *micro slots*. Because of the number of pins needed to access one external SRAM, each module

willing to use an SRAM module must be implemented in a multiple of 3 micro slots.

7.0.4   The configuration manager

A part from the main FPGA, the *BabyBoard* also contains the configuration circuitry. This consists of a CPLD, a configuration FPGA, that is a small *Spartan II*-FPGA, and a Flash.

- The CPLD is used to download the *Spartan II*'s configuration from the flash on power-up. It also contains board initialization routines for the on board PLL and the Flash.

- The configuration program for the main FPGA is implemented in the *Spartan II*. Because of its small size, this reconfiguration program could be directly implemented in the CPLD, which could configure the main FPGA at power on. But a much larger device was chosen to increase the degree of freedom in the reconfiguration management. As stated before, the relocation is an important operation of the ESM. This can be done in different ways: The first approach is to keep a bitstream for each possible module and each possible position in memory. However, the size of the Flash cannot allow us to implement this solution. The second approach is the online modification of the coordinate of the resources used in the module's bitstream to match the new position. This modification can be done for example through a manipulation of the bitstream file with the Java program JBits [102] previously presented. However, file manipulation is a time-consuming operation that will increase the reconfiguration time. The last and most efficient solution is to compute the new module's position while the bitstream is being downloaded, using a dedicated circuitry. This requires a much larger hardware code than the simple reconfiguration. This is why the *Spartan II* was chosen.

- The Flash provides a capacity of 64 MByte, thus enabling the storage of up to 32 Full configurations and few hundreds partial bitstreams for the FPGA Virtex II 6000 that was used.

7.0.5   Memory

Six SRAM banks with 2 MBytes each are vertically attached to the board on the top side of the device, thus providing enough memory space to six different slots for temporal data storage. The SRAMs can also be used for shared memory communication between neighbour modules, e.g for streaming applications. They are connected to the FPGA in such a way that the reconfiguration of a given module will not affect the access to other modules.

7.0.6   Debug lines

Debugging capabilities are offered through general purpose I/O provided in regular distance between the basic slots. A JTAG port provides debug capabilities for the main FPGA, the CPLD and the Spartan II.

7.0.7   The *MotherBoard*

The *MotherBoard* provides programmable links from the FPGA to all peripherals. The physical connections are established at run-time through a programmable crossbar implemented in a Spartan FPGA on the *MotherBoard*. This crossbar functionality basically solves the I/O pin dilemma of many existing FPGA platforms, thus allowing free relocation of modules requiring I/O pin connectivity. Besides the run-time programmable crossbar, many peripherals for multimedia and communication are available on the board (figure 7.18).

Video capture and rendering interfaces as well as high-speed communication links are available on the *MotherBoard*. This interface allows the platform to be used for instance in autonomous system, where a robot may collect pictures from the environment and send them after a preprocessing step to a central station for evaluation.
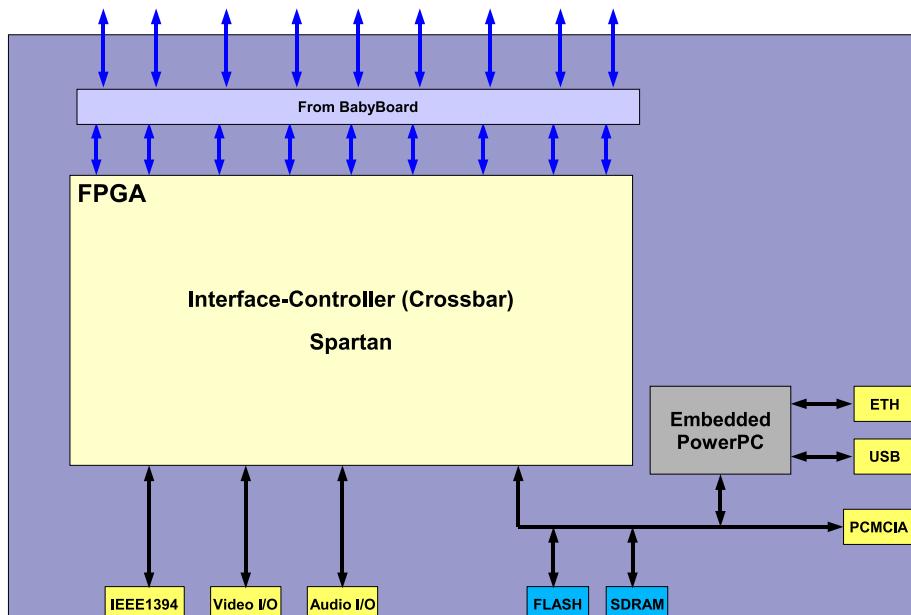
Figure 7.18.   Architecture of the ESM MotherBoard

7.1   Intermodule communication

One of the central points in dynamic reconfiguration is the communication. Each module that is placed on the device must collect its data from a given source and send its results to a given destination. This problem is given an importance only in few systems such as the XF-Board of Zurich [177] [211]. However, the XF-Board implements only a communication through a third party, which is slow because of the long path that messages have to go.

In the ESM, the communication among different modules (figure 7.19) can be realized in three different ways: The first one is a *direct communication* using *bus macros* between adjacent modules. The second one is the *shared memory* using the external SRAMs or the internal BlockRAMs. However, only neighbour modules can use those two communication modes. For modules placed in non-adjacent slots, we provide a dynamic signal-switching communication, the reconfigurable multiple bus (RMB) presented in section 4.1. Also, the crossbar switch can be use as well for communication between modules.
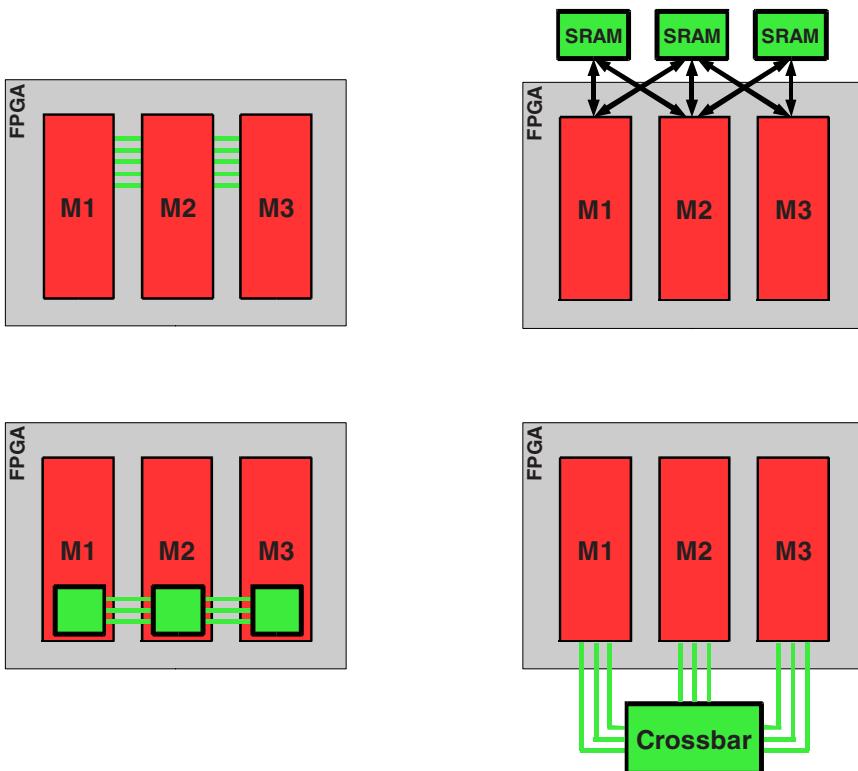
Figure 7.19.   Intermodule communication possibilities on the ESM

7.1.1 Adjacent Communication

Adjacent communication is done between two modules, which abut each other. Bus macros must be used for a direct communication between neighbouring modules. The number of Bus-macros needed is limited by the amount of signal that can go through the Bus-macros and the number of signals to be connected.

7.1.2 Communication via shared memory

The communication between two neighbouring modules can be done in two different ways using shared memory:

- Communication using BlockRAM: The BlockRAMs are dual ported RAMs used in the FPGA mostly to implement FIFOs and other memory. It is a nice possibility to implement communication among two neighbour modules, working in two different clock domains. The sender will just write on one side with its own frequency, whereas the receiver will read the data at the other end with its own frequency.

- Communication using external RAM: The communication through the external RAM is particularly useful in applications in which each module must process a large amount of data and then send the processed data to the next module. This is mostly the case in video-streaming application in which a first module captures a stream, image by image. The images are alternately stored in two different memory, which are accessed by the next module for reading. On the ESM, each SRAM can be accessed by three modules as shown in figure 7.20.
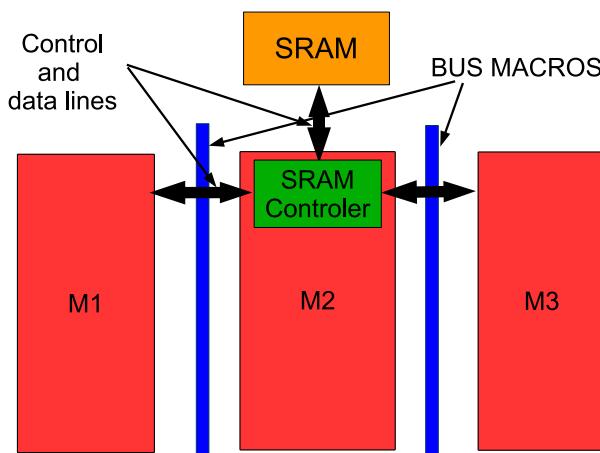
Figure 7.20. SRAM-based intermodule communication on the ESM

Because an SRAM can only be accessed by one module at a time, a controller is then used to manage the concurrent SRAM access between the three modules. Depending on the application, the user may set the priority of accessing the SRAM for the three modules.

7.1.3   Communication via RMB

The communication between non-adjacent modules happens through the 1-D circuit-switching paradigm presented in section 4.1.

7.1.4   Communication via the crossbar

The last possibility to establish a communication among modules is to use the crossbar switch available on the mother board. Because all modules are connected to the crossbar via the pins at the bottom of the FPGA, the communication among the modules can be set in the crossbar as well. This possibility should be used only as emergency solution, because of the high latency caused by the of-chip routing.

8.   Enhancement in the Platform Design

The design of the Erlangen Slot Machine was strongly influenced by the architectural limitations of the Xilinx Virtex II FPGA, in particular their columnwise reconfiguration. This limitation was the primary motivation for the implementation of the crossbar switch on a separate FPGA. All the connections between running modules and their peripherals could be implemented in the crossbar switch and be safe from the reconfiguration of the module. Substantial efforts were therefore placed in the design of the two boards previously described.

With the introduction of the Virtex 4, Xilinx has introduced a new paradigm in which complete column must not be replaced on reconfiguration. This new development can lead to a very great enhancement on the platform design in the two following aspects:

- First, the crossbar switch can now be implemented in the FPGA rather than on a separated device. In this case, we need to attach all the peripheral on one side of the block in which the crossbar is implemented. Modules can then be connected on the other side of the block. We can even think of a system in which all components are attached around the chip. The crossbar can then be implemented as a ring, distributed module around the chip.

- The distribution of the resources, such as the external RAM must not be done in a column-wise manner anymore. Resources should now be homogeneously spread around the device, to allow different modules, which are placed on different blocks to access their own resources.

Figure illustrates the enhancements on the Erlangen Slot Machine, using the Virtex 4 and Virtex 5 FPGAs as previously explained.
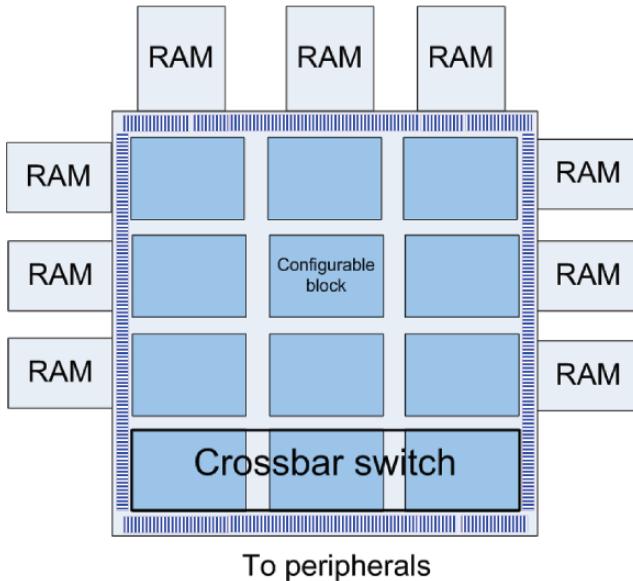
Figure 7.21. Possible enhancement of the Erlangen Slot Machine on the Xilinx Virtex 4 and Virtex 5 FPGAs

Despite the advantages provided by the new class of Virtex devices, a great disadvantage that results from this architectural enhancement is the communication: for the columnwise reconfigurable devices, the 1-D circuit was a simple and efficient possibility to enforce communication among different modules running on the FPGA. In a two-dimensional reconfigurable device as is the case with the Virtex 4 and Virtex 5, the difficulty of implementing the communication increases. In 2-D, we have a quadratic growth in the amount of resources needed to implement the communication infrastructure. That is the amount of resources needed to implement a 2-D communication infrastructure is not only twice the amount needed to implement a 1-D communication infrastructure but four times.

9. Conclusion

We have presented in this chapter the different possibilities to design for partial reconfiguration on Xilinx devices. Our goal was not to rewrite a complete manual on partial reconfiguration, because several descriptions on this exist [128] [227]. The Xillinx manuals as well as the work of Sedcole [190] provide very good descriptions on using the modular design flow and the early access. Our primary motivation was to provide a king of tutorial, based on our experi-

ence and for which a workable design, not too complex, but also not so easy, exists. The designs as well as all the scripts needed for compiling are available to download from the book's web page. The difficulty in designing for partial reconfiguration can be reduced, if the target platform is well designed. One of such platform is the Erlangen Slot Machine that was presented, with the goal to emphasize the challenges in designing such a platform. The ESM is however strongly influenced by the column-wise reconfigurable Virtex. The price we pay for flexibility is very high. With the advent of new Virtex 4 and virtex 5, enhancements can be made in the design to increase the flexibility, while reducing the costs.

One of the main problems is the communication between the module at runtime. While this problem is somehow better solved in a 1-D reconfigurable device through the use of circuit switching and dedicated channels on the module, its extension on a 2-D reconfigurable device is not feasible because of the amount of resources needed. Viable communication approaches such as the DyNoC was presented in chapter 6; however, with the amount of resources needed by those approaches, their feasibility is only possible if manufacturers provide coarse-grained communication element in their devices.

Chapter 8

SYSTEM ON A PROGRAMMABLE CHIP

Developments in the field of FPGA have been very amazing in the last two decades. FPGAs have moved from tiny devices, with few thousands of gates, only able to implement some finite state machines and glue-logic to very complex devices with millions of gates as well as coarse-grained cores. In year 2003, a growth rate of 200% was observed in the capacity of the Xilinx FPGA in less than 10 years, while in the meantime, a 50% reduction rate in the power consumption could be reached with the prices also having the same decrease rate. Other FPGA vendors have faced similar development, and this trend is likely to continue for a while. This development, together with the progress in design tools, has boosted the acceptance of FPGAs in different computation fields. With the coarse-grained elements such as CPU, memory, arithmetic units, available in recent FPGAs, it is now possible to build a complete system consisting of one or more processors, embedded memory, peripherals and custom hardware blocks in a single FPGA. This opportunity limits the amount of components that must be soldered on a printed circuit board to get a FPGA system working.

In this chapter, we present the different possibilities that exist to build those systems consisting of one or more processors, peripherals and custom hardware component.

We start with an introduction in system on programmable chip, and then, we present some of the various elements usually needed to build those systems. We then present at the end of the chapter a design approach for adaptive multiprocessor systems on chip.

1. Introduction to SoPC

A *system on chip* (SoC) is usually defined as a chip that integrates the major functional elements of a complete end product. Figure 8.1 presents the
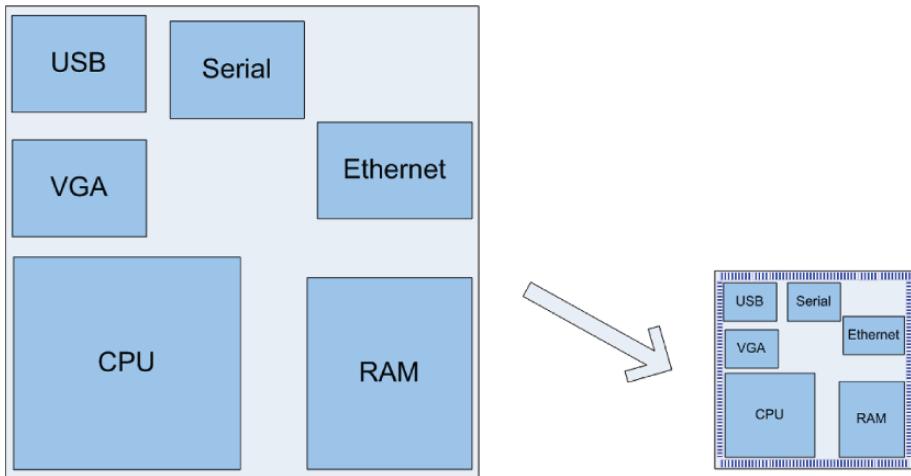
Figure 8.1.   Integration of PCB modules into a single chip: from system on PCB to SoC

achievement of a system on chip, namely the replacement of a complete printed circuit board by a single chip.

A system on chip usually consists of a processor, memory, peripheral modules and custom hardware modules. Contrary to a system on chip that is manufactured on an integrated circuit and cannot be modified again, a *system on programmable chip* (SoPC), or programmable system on chip, is built on programmable logic. Its structure can be modified by the end user, either at compile-time or at run-time by means of full or partial reconfiguration.

In the next sections, we take a brief look on the main components of system on programmable chip and present some key implementations of those components by manufacturers.

1.1   Processor

Processors are the central processing and coordination units in system on programmable chips. Besides the coordination of the complete system, they are in charge of collecting the data from different peripheral modules or from the memory, process those data and store them into the memory or send them to the peripheral modules. Also the processor initializes the peripherals as well as the dedicated hardware modules on the chip and manages the memory. The most widely used processors are those used on Xilinx and Altera devices, because those two companies control the largest part of the programmable device market. Besides the processors offered by those two companies, other platform-independent implementations exist. We list some of the available processors next.

1.1.1 The MicroBlaze Processor Core

The *MicroBlaze* [127] is a soft 32-bit RISC processor core designed and commercialized by Xilinx. The processor is not directly available on a chip. It exists only as reference design, optimized for the Xilinx FPGA. With a clock frequency of up to 200 MHz and a possibility to include a 32-bit single precision floating point unit (FPU) in the IEEE-754 format into the datapath, the *MicroBlaze* is one of the fastest available soft core processor. It consists of a set of fixed components and a set of optional features that can be configured at design time by the user. The main fixed features of the *MicroBlaze* are:

- An Arithmetic and Logic (ALU) Block unit featuring a shifter

- thirty two 32-bit general purpose registers

- Harvard architecture with separate 32-bit address bus and 32-bit data bus

- a 5-stage pipeline in the version 5.1a

- seven *fast simplpex links* (FSL) that can be used to connect the processors to custom hardware

- an interface for connecting the processor to an *on-chip peripheral bus* (OPB)

Besides these fixed features, the *MicroBlaze* processor can be parameterized by the users at compile-time to include additional functionalities. The latest version of the *MicroBlaze*, the version v5.1a, can be configured to support the following additional features:

- a barrel shifter, a multiplier and a divider can be added to the ALU-Block

- a 32-bit single precision FPU in the IEEE-754 format

- A data cache and an instruction cache

- an interface to the *local memory bus* (LMB), a high-speed synchronous bus used primarily to access on-chip block RAM.

- A *CacheLink* for accessing external memory

- debug logic

The *MicroBlaze* soft processor core is available as part of the *Xilinx embedded development kit* (EDK), which includes a comprehensive set of system tools to design an embedded application in a Xilinx FPGA.

1.1.2   The PowerPC 405 Processor

Contrary to the *MicroBlaze* that exists only as reference design compiled to a netlist, the *PowerPC 405* processor [126] is immersed into some Xilinx Virtex FPGAs (the Virtex II Pro, the Virtex 4 and the Virtex 5). The integration of the PowerPC directly into the chip at fabrication allows for a great performance increase compare configurable processor cores.

The *PowerPC 405* processor core used in the Xilinx devices is a 32-bit implementation of the*PowerPC* embedded environment architecture that can be clocked at up to 450 MHZ. It provides a dual mode operation, which allows programs to run either in privileged mode or in user mode. The main components of the *PowerPC 405* are:

- the *central processing unit* (CPU) with the following features:

  - a five-stage (fetch, decode, execute, write-back) pipelined datapath. A fetch queue consisting of two prefetch and a decode buffer is used to queued instructions in case of stalls. Instructions flow directly to the decode buffer if the prefetch buffers are empty.

  - a general purpose register file (GPR) consisting of thirty two 32-bit registers accessible via three reads and two write ports. The five ports on the GPR allow the processor to execute load/store operations in parallel with ALU or MAC operations.

  - an issue execution unit consisting of ALU and a single cycle throughput multiply accumulate unit (MAC).

- an exception handling module for servicing a total of 19 critical and non-critical exceptions that can be caused by error conditions, the internal timers, debug events and the external interrupt controller (EIC) interface.

- a memory management unit, which provides address translation, protection functions, and storage-attribute control for a total of 4 GB non-segmented address space.

- a 16 KB instruction-cache and a 16 KB data-cache, which can be accessed respectively through the data cache and instruction cache units.

- additional resources such as timers and debug units.

- a processor local bus (PLB) interface providing a 32-bit address and three 64-bit data buses attached to the instruction-cache and data-cache units.

- a device control register (DCR) interface for the attachment of on-chip device control, clock and power management interfaces for clock distribution and power management.

- a JTAG debugging port, on-chip interrupt controller interface and on-chip memory controller interface for attaching additional memory.

- an auxiliary processor unit (APU) interface and a APU controller that are used for extending the native *PowerPC 405* instruction set with custom instructions, implemented as coprocessor on the FPGA Fabric. With the APU a tighter integration of application-specific function into the processor is possible. The APU can be used for processor interconnection as well. The *PowerPC 405* does not have a floating point unit; however, Xilinx provides in its module library a dedicated floating point unit that can be attached to the processor through the APU interface.

1.1.3 The Nios Processor Core

Like the *MicroBlaze*, Altera's *Nios II* [124] processor core is available as reference design to be synthesized and to be downloaded into FPGAs. The architecture supports operation in user mode as well as in supervisor mode, thus allowing for a protection of the control registers. The *Nios II* core can be customized by the designer by adding optional functionalities to the basic system. Its main features are

- a register file that consists of thirty two 32-bit general purpose integer registers, and six 32-bit control registers. Floating point registers can be added to the register file by the customization of the processor.

- an ALU supporting the main arithmetic and logic operations as well additional shift and rotate operations.

- a single precision floating point as specified by the IEEE Std 754-1985, which can be added to the core by customization.

- optional, a NIOS core may include one instruction cache and/or one data cache, depending on the user application. Caches can be avoided to keep the design small.

- one data bus selector module and one instruction-bus selector interfaces, each of which can be used to provide memory and I/O access. Each controller provides an *Avalon* master port for connecting the memory via the *Avalon* switch fabric, and an interface to fast memory outside the *Nios II* core.

- various interface for debugging and interrupt handling.

- a custom instruction port that can be used to extend the native instruction set. A set of instructions implemented in a hardware module can be accessible by connecting that hardware module to this port.

1.1.4   The LEON Processor Core

Developed by the company Gaisler Research, the *LEON* processor core [1] is an open source 32-bit processor core based on the SPARC V8 architecture. Several versions of the *LEON* exists; however, we present only the *LEON3* in this section. The *LEON3* is available as a synthesizable VHDL model, which means that it can be implemented on any FPGA platform with the corresponding vendor tool. The complete source code is open and available under the GNU GPL license. The main features of the *LEON3* processor, which can be clocked at up to 125 MHz, are

- Advanced 7-stage pipelined datapath

- Hardware multiply, divide and MAC units

- a fully pipelined floating point unit in the IEEE-754 format

- Separated instruction and data cache, whose size, associativity, and replacement policy, can be configured

- a memory management unit with configurable TLB

- an interface to the *AMBA-AHB* bus

- support for Symmetric Multiprocessor (SMP)

- various interfaces for debugging, clocking and power-down mode

1.2   Memory

A system on programmable chip needs memory for storing instructions and data. Memory elements are available in small amount on almost all modern FPGAs. This is usually used to build the caches directly on the chip. For applications that require more memory, external SRAM or DRAM must be used. On-chip memory can be built from the memory elements, the block RAMs, or from the LUTs. The use of the LUT in the second case to built memory has two drawbacks: First, the amount of available computing resources decreases, thus reducing the complexity of the logic that can be implemented on the chip. Second, the memory built with LUTs is distributed across the chip, meaning that chip interconnection must be used to connect the spread LUTs. This leads to decrease in performance in the memory access. On-chip block RAMs are usually dual-ported. They therefore provide a nice possibility to integrate a custom hardware module, in particular those working in two different clock domains. The memory is used in this case to synchronize the module with each other.

1.3 Peripheral

Peripherals are used in the system for communication with the external world and for debugging purpose. The peripheral components are usually provided by the board manufacturers as ready to the module that can be inserted in the SoPC-design. Because of their lower frequency, compared with the one of processors, peripherals should be connected to the low-speed bus, which in turn can be connected to the processor bus through a bridge. The communication between the processor and the available peripheral can be done either through the I/O mapping paradigm or through the memory map paradigm. In the first case, the processor addresses each peripheral using special instructions that directly address the peripheral on the bus. In the second, the peripheral is assigned an address space in the memory and the communication happens in this space with the processor writing in control register that can be read by the peripheral. The peripheral responds by writing in status registers that can be read by the processor.

1.4 Custom hardware

The final types of component in a system on programmable chip are the custom hardware. Those are hardware components that implement special functions for which one intend to speed-up the computation. Custom hardware modules are usually streaming-based, meaning that the computation is performed on a stream of data that flow through the hardware module. Care should be taken while integrating custom hardware modules in a system on programmable chip. Connecting a custom hardware module on a peripheral bus may result on a performance decrease, if the data rate of the computation is much higher than that of the bus, which is usually the case. Direct connection and communication using dual-ported RAM can help in this case to allow the custom module to work with its maximal frequency.

1.5 Interconnection

The interconnection provides the mechanism for integrate all the components previously described in a workable system on programmable chip. The interconnection mechanism provides the communication channels, the interface specification, the communication protocols and the arbitration policy on the channels. The design of the interconnection infrastructure depends on the target application. In traditional system on chip, where the infrastructure is fixed at chip production, the communication infrastructure must be as general as possible to serve all class of applications that can be implemented on the chip. In programmable devices, the flexibility can be used to modify the interconnection infrastructure according to the type of application to be implemented. In this case, each application can first be analysed to derive and

implement the best communication infrastructure for its computation. Despite the great interest in network on chip in the last couple of years, interconnection on chip is dominated by the SoC communication paradigm which is in most of the case bus-based. Leading existing solutions were previously developed for SoCs before adapted to SoPCs. The general connection mechanism is provided in figure 8.2.

It usually consists of two different buses. A high-performance bus that is used by the processor to access the memory and a slow bus used to connect the peripherals. High-performance dedicated hardware modules are connected to the high-performance bus, whereas low-performance custom hardware components are connected to the low-performance bus. A bridge is used to allow for communication to happen between two modules attached on the two different
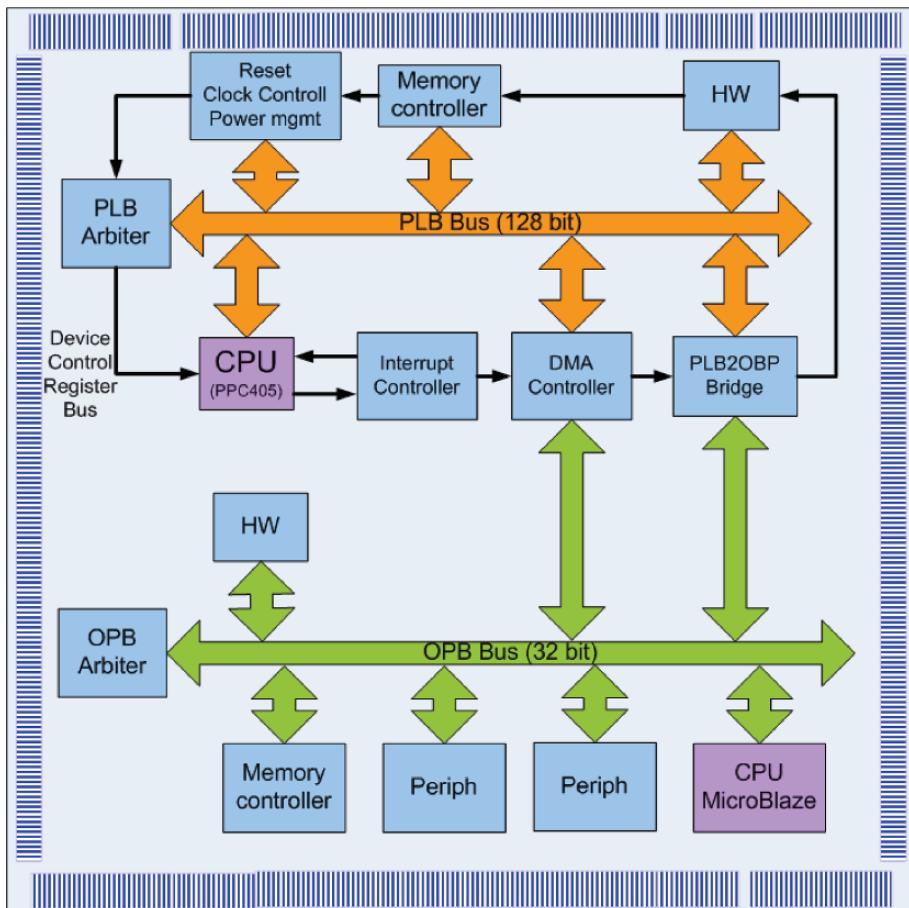
Figure 8.2.   Example of system ingration with CoreConnect buses

busses. Besides those two buses, several possibilities exist to directly connect the component. Dedicated lines or crossbar switches can be used, even in cohabitation with the two previously described buses. The two well-established bus systems in the world of programmable system on chip are the *CoreConnect* from IBM and the ARM *AMBA*.

1.5.1 The IBM *CoreConnect*

As shown on figure 8.2, the IBM *CoreConnect* communication infrastructure consists of three different buses:

- the PLB, which is a high-performance bus, used to connect high-bandwidth devices such as high-performance processor cores, external memory interfaces and DMA controllers.

- the OPB, which is a secondary bus that can be used to decoupled the peripherals from the PLB to avoid a lost of system performance. Peripherals such as serial ports, parallel ports, UARTs, GPIO, timers and other low-bandwidth devices should be attached to the OPB. Access to the peripherals on the OPB bus by PLB masters is done through a bridge, which is used as a slave device on the PLB and as master on the OPB. The bridge performs dynamic bus sizing, to allow devices with different data widths to communicate.

  Figure 8.3 illustrates the implementation of the OPB. Note that no tri-state is required. The address and date buses are instead implemented using a distributed multiplexer. This is a common technique for implementing buses in programmable logic devices. All the master inputs to the bus are ORed, and the result is provided to all the slaves. The arbitration module defines which master is granted the bus. All other masters must then place a zero signal on their output. The ORed is then used to write the value of the bus master on the bus.

- the DCR bus to allow lower performance status and configuration registers to be read and written. It is a fully synchronous bus that provides a maximum throughput of one read or write transfer every two cycles. The DCR bus removes configuration registers from the memory address map, reduces loading and improves bandwidth of the processor local bus.

1.5.2 The *AMBA*

The Advanced Microcontroller Bus Architecture (*AMBA*) from ARM shares many similarities with the IBM *CoreConnect*. Both architectures support data bus widths of 32-bits and higher, they utilize separate read and write data
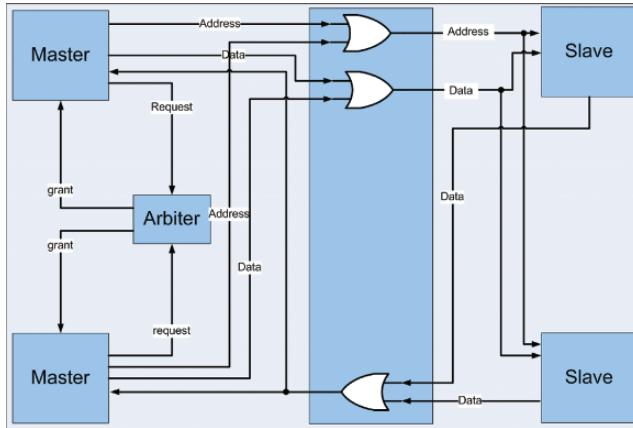
Figure 8.3.   Implementation of the OPB for two maters and two slaves

channels, they allow multiple masters and support split transactions and burst transfers. The *AMBA 2* interconnection infrastructure consists of two main buses:

- the advance high-speed bus (AHB) that is used as high-performance system interconnect to allow communication between high-performance modules and the processor. The AHB plays the same role in the AMBA that the PLB plays in the *CoreConnect*. A bridge from AHB or ASP is required to interface to APB.

- the advance peripheral bus (APB), used to connect the slower peripherals. A bridge is used to interface between the AHB and APB buses and allow low peripheral to be accessed from components hung on the AHB.

2.   Adaptive Multiprocessing on Chip

The continuous performance improvement of the last decades in micropro- cessor is not likely to hold in the future. This positive trend observed in past, and also known Moore's law, was due to two main factors: high-speed clocks and improvement in instruction level parallelism.

While the speed was increased by constantly growing clock frequency, the capacity has always been boosted by the reduction of transistor size. Sequential programs could therefore be speeded-up through instruction level parallelisms (ILP) in a transparent way for the user. The increase in speed is getting more and more difficult to maintain. Enhancement on ILP-design for datapath is not coupled anymore with a performance increase of the system [169] [213]. Moreover, the power is becoming one of the main limitations. To solve this

problem, the research community and the industry have started using the large amount of available chip area to implement several processors, thus creating the so called chip multi processing systems.

Basically, two paradigms exist for parallel computation on multiprocessors. The first one is the *message passing interface (MPI)*, usually available where communication among processors happens in a network. The second paradigm, the *shared memory* or *symmetrical multi processing (SMP)* is applied in bus-based systems where communication happens through a share memory. Multiprocessor on chip has been studied according to the second paradigm. In most of the existing works, the purpose is to have an efficient mapping of a set of threads onto the processors. Two possibilities have been presented that address this purpose: *simultaneous multithreading (SMT)* [75, 120, 103] and *chip multiprocessor (CMP)* [169, 109, 139, 94, 16]. An SMT-Chip is based on a superscalar processor with a set of units for a parallel execution of instructions. Given a set of threads, the resources are dynamically allocated by extracting several instructions to be executed in parallel. Threads that need long memory access are preempted to avoid idle states of processors. Based on an *Alpha 21164* processor with 10 function units and 8 instructions per clock, an SMT-system was shown to provide a speedup of up to 4 compared with common superscalar implementations [75]. In Hirata et al. [120], ray-tracing-application on SMT-architecture could be simulated with considerable performance improvement. Gulati and Bagherzadeh [103] could also provide simulation results of SMT with performance improvements.

Instead of using only one superscalar architecture to execute instructions in parallel, CMP use many processor cores to compute threads in parallel. Each processor has a small first-level local instruction and data cache. A second-level cache is available for all the processors on the chip. CMPs target applications consisting of a set of independent computations that can be easily mapped to threads. This is usually the case in database or webserver where transactions do not rely on each other. In CMPs, threads having long memory access are preempted to allow others to use the processor. Barroso et al. [16] have presented a scalable CMP architecture called *Piranha* in which the processors – in this case alpha processors – use a crossbar switch for communication. Despite the good simulation results no physical chip could be manufactured. A most concrete approach is the work of Lammon and Olukutun that leads to the design of the *Hydra Chip* [109], whose technology is used in the *Niagara* chip [139]. In this architecture, a hardware support for speculation is available, besides the parallel execution of threads. The communication among the processors is done using a two-level on-chip cache. The simulation provides almost a linear performance increase in the number of processors. Other developments in multiprocessor on chip include the IBM *Cell-Chip*, consisting of a 64-Bit Power processor and eight so-called *synergistic processors*. The Intel

Pentium D and the AMD *Dual Core Athlon 64* Processor are other examples of multiprocessor on chip.

The work in [167] proposed an adaptive chip-multiprocessor architecture, where the number of active processors is dynamically adjusted to the current workload needed to save energy while preserving performance. The proposed architecture is a shared memory based, with a fix number of embedded processor cores. The adaptivity results in the possibility of switching the single processors on and off.

Programmable logic devices in general and FPGA in particular have experienced a continuous growth in their capacity, a constant decrease in their power consumption and permanent reduction of their price. This trend, which has increased interest in using FPGAs as flexible hardware accelerators, is likely to continue for a while.

After an unsuccessful attempt in the 1990s to use FPGA as co-processor in computing system, the field FPGA in high-performance computing is experiencing a renaissance with manufacturers like Cray who now produce systems made upon high-speed processors couple to FPGAs that act as hardware accelerators [58]. However, actual FPGA solutions for high-performance computing still use fast microprocessors, and the results are systems with a very high-power consumption and power dissipation.

A combination of the chip multiprocessor paradigm and flexible hardware accelerators can be used to increase the computation speed of applications. In this case, FPGAs can be used as target devices in which a set of processors and a set of custom hardware accelerators are implemented and communicate together in a network.

FPGA manufacturers such as Xilinx have been very active in this field by providing ready to use components (soft of hard-cores processors, bus-based intercommunication facilities, and peripherals) on the base of which, multiprocessor on chip can be built. The *Virtex II Pro* FPGA for example provides up to four *PowerPC* processors and several features like memory and DSP-modules.

The adaptivity of the whole system can be reached by modifying the computation infrastructure. This can be done at compile-time using full reconfiguration or at run-time by means of partial device reconfiguration. This results in a multiprocessor on-chip, whose structure can be adapted at run-time to the computation paradigm of a given application.

Unfortunately, the capabilities of those devices are not exploited. FPGA devices like the Xilinx *Virtex Pro* are usually used only to implement a small hardware component. The two available *Power PC* processors remain most of the time unused. This is in part due to the lack of support for the multiprocessor design.

In this section, we investigate the use of Adaptive Multiprocessor on Chip (AMoC) and present a design approach for such systems. We also present an

environment developed at the University of Kaiserslautern in Germany, for a seamless generation and configuration of the hardware infrastructure to produce a complete system.

One of the main applications of such infrastructure is in high-performance computing in embedded systems. We next present our understanding of the system architecture that must be first generated onto the device. Thereafter, we present a two-step design approach and the design automation for AMoCs.

2.1 Hardware Infrastructure

Many possibilities exist to build a multiprocessor hardware infrastructure on a chip. In this section, we however focus on the general computing infrastructure shown on figure 8.4. For the sake of modularity, the infrastructure consists of a set of *processor blocks*, each having a custom hardware accelerator and memory directly attached the processor, either through the available fast processor bus or through a dedicated communication link. The *processor blocks* are connected in a network and are accessible from off-chip components via peripherals. Additional external RAM might be available.

One of the main tasks in the design and implementation of a system like that of figure 8.4, is the interconnection network. Contrary to other modules (processors, peripherals, memory controllers and some custom hardware) that
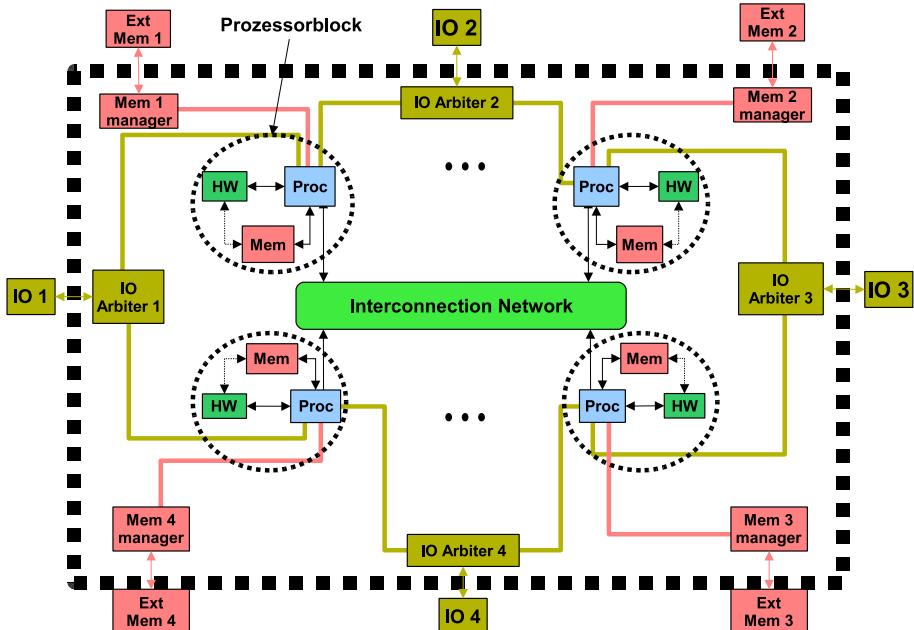
Figure 8.4. General adaptive multiprocessor hardware infrastructure

are available in the vendor libraries, communication facilities, in particular network on chip modules are not available at all. Also, most of the existing tools provide only a very limited support for multiprocessing. We therefore focus on the communication aspect next.

2.2   Communication Infrastructure

The communication between the processor blocks can basically be done in two different ways: Shared-memory and message passing; in the first case, the processor blocks are attached to a common bus on which a memory is also attached. The exchanged of message is done through the common accessible memory. Although such a system can be built with the current vendor tools, the amount of handwork required to have a system working is quite high. Furthermore, some modules needed to build a workable system, for instance a module to handle the cache coherence amount, the processors is not provided and must be designed and implemented by the user. In the second case, where message passing is used, the processor blocks are accessible through a network on which they hung. The network is in charge of routing messages from source to destination. Although message passing can be implemented with shared-memory, we focus in this section on those implementations where a set of independent processors communicate with messages that are sent through a network.

The design of the communication network should be carefully done, to avoid a high consumption of chip resources, while maximizing the speed and the throughput. A network on chip (NoC) normally consists of set of routers each connected to a processor and direct connections between the routing elements. Each router provides a processor an interface to the network for sending and receiving messages in the form of packets. Also, the routers are in charge of placing incoming packets on the best tracks and insure a smooth routing to destination.

While the routers have the advantage of flexibility, thus allowing any topology to be built, their resource consumption has been shown to be quite high in FPGAs, because of the large amount of multiplexers used [32]. To avoid wasting too much chip resources, a limited amount of routers may be used. This can be reached by using a ring architecture. Like a Bus, a ring provides a great advantage in the resource consumption. The disadvantages of the bus, like the exclusive bus access by masters and the lower fault tolerance can be can be compensated by having several masters on different ring segments. In [108], a ring architecture was designed for the interconnection of a set of processors blocks on a FPGA.

As figure 8.5 illustrates, the processors are primarily connected using a ring. However, because a ring allows only one kind of topology to be built, routers are available for building cross-connections over different rings. The routers
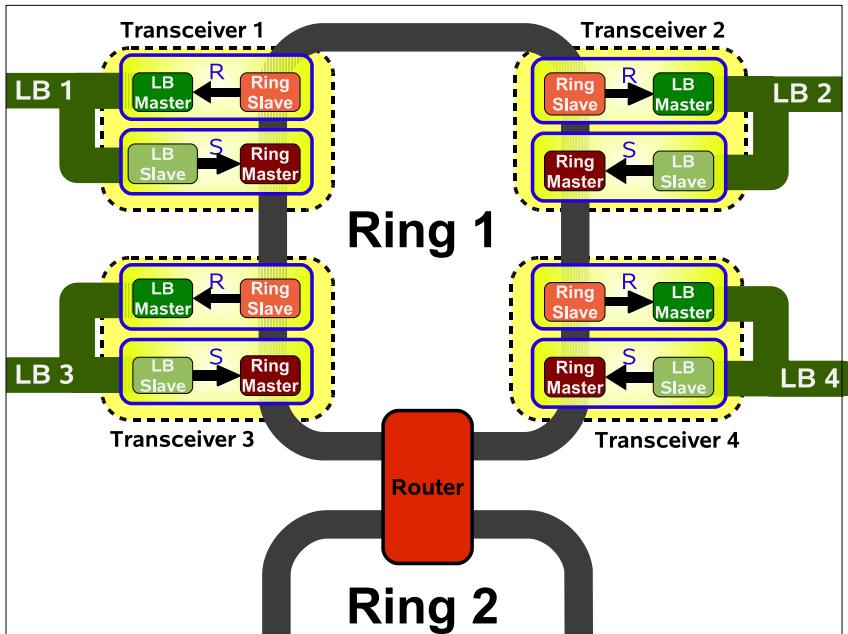
Figure 8.5. Structure of the on chip network

therefore provide a means to build other topologies than the ring. A router is able to connect up to four rings. Building a mesh for example is done by connecting two 'perpendicular and disconnected rings' through the router. In figure 8.5, two rings connected to a router are shown.

A ring has the advantage that components connected to it do not have to deal with routing, and therefore, the amount of multiplexers (one of the most resource consuming element) needed to implement the complete infrastructure decreases drastically. Components only need *transceivers* to access the ring for sending and receiving messages (figure 8.5). Each *transceiver* consists of a *sender* (S) and a *receiver* (R) that operate independent from each other. The receivers behave on the ring as slave and on the local bus as master, whereas the sender behaves on the local bus as slave and on the ring as master. The *ring slave* in the receiver collects data from the ring and passes them to *local bus master* in the same receiver that. The message can then be copied into the local memory through the local bus. The *local bus slave* in the sender collects data from the local bus and send them to *ring master* in the sender, which then place the message onto the ring (figure 8.6).

Within a transceiver, each slave and each master is basically a finite state machine (FSM), which is in charge of initiating and handle the required
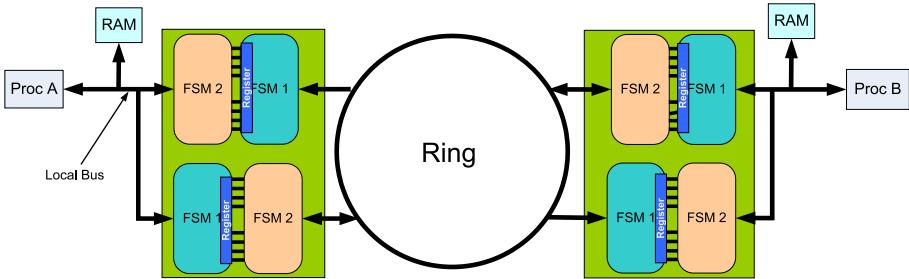
Figure 8.6.   Implementation of the transceiver

transactions (figure 8.6). This modularization helps to decouple the complex bus operations from the cumbersome ring transactions.

The transceivers operate in two different modes: *active copying* and *DMA (direct memory access)*. The active copy mode is used to send small number of packets while DMA deals with the transmission of large data segments. With the DMA capability on transceivers, a remote DMA between two processors is implemented, to quickly transfer large amount of data from one processor to another one, without loss of performance. The communication protocol used in this case is next explained.

2.3   Data Transfer and Communication Protocol

In *active copy* mode, the processors pass each single byte to be sent to the transceiver whereas in DMA mode, the processor just specifies the destination address and provides only the start address of the data block to be sent. The transceiver can then access the memory and transmits the data over the network, while the processor continues its computation.

A sending operation is initiated by a processor by copying the destination address together with some control data to the control register of the FSM acting as *local bus slave*. While this first FSM collects data to be sent from the local bus, the second FSM, acting as master on the ring, builds packets and send them through the ring. In case of an active copy, the data are placed on the bus by the processor, while in DMA, the slave FSM accesses the memory itself to collect data, thus freeing the processor. This second possibility is used to implement a remote DMA (RDMA), which allows a processor to directly copy a chunk of data in the memory of another processor on the network. The destination processor must therefore dynamically allocate enough space in its local memory for incoming data from the sender processors. A processor P_s, willing to send data to another processor P_r, first sends a request of RDMA data transfer to processor P_r. The request specifies the length (in bytes) of

data to transfer. P_r then tries to allocate enough memory for the data and allows P_s to transfer data, if the reservation is successful. The transfer is done by the sender and receiver modules of the corresponding transceivers while the two processors might perform other tasks.

To implement the handshaking between the two processors, a communication area is foreseen on each processor. This is a table with two entries for each processor. The first one specifies the request/acknowledge of data transfer and the second one specifies the amount of bytes to transfer. Each processor performs a request/acknowledge through a remote writing in the table of the sender/destination processor at the corresponding location.

A possible communication scenario is shown in figure 8.7; The network consists of $n + 1$ processors also called clients. $client_0$ and $client_n$ consecutively request a RDMA-transfer of data to $client_1$ (step 1 and step 2). According to a first-request-first-grant mechanisms, $client_1$ first grants RDMA access to $client_0$ by writing the granted-code at the right location in the handshaking table of $client_0$ (step 3). From this point on, the sender transceiver and the receiver transceiver are used to realize the data transfer, while the two processors keep computing (step 4). Upon completion, $client_0$ removes its request (step 5), and $client_1$ removes its grant. $client_n$ is now granted the RDMA-access (step 6) and the data transfer is done according to the scheme previously described (step 7 and step 8).
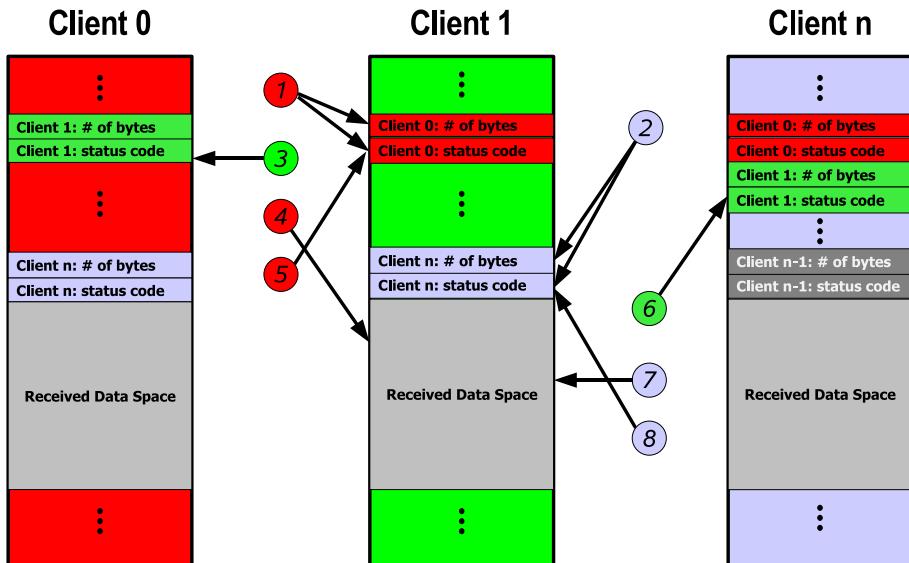
Figure 8.7. The communication protocoll

2.4   Design Automation

The goal in design automation is to ease the implementation of an adaptive multiprocessor system described in figure 8.4, on an FPFA.

The generation of such a system is a complex task whose support through the current vendor tools is very limited. A substantial amount of hand work must be done to have many processors working in an FPGA. The only paradigm supported by the current tools, albeit very limited, is the SMP-like computation paradigm. For the design of MPI-like systems, the user must do everything from scratch. The network components must be provided, also support for cache coherence in the case of SMP-like computation. This requires a great amount of hand work and strong experience in hardware design. To make multiprocessing implementation on chip easier, in particular for software designers, but also for those hardware designers not willing to spend too much time in the hardware part of the system, a seamless approach must be available to generate the hardware infrastructure. The designer can then focus only on parallelizing he's code by defining the part to be executed by each processor and the communication points in the code.

The work in [130] [183] has contributed, besides the development of the communication infrastructure to the design and implementation of a vendor independent framework in which the user, no matter if beginner or experienced, can specify the complete requirements of he's system and let the tool produce the necessary configuration and start-up code.

The approach that followed is a two-step one, consisting of first providing an abstract specification of the system. The abstract specification will then be refined and mapped to a concrete description according to the component library available in the second step.

2.5   Automatic Generation of the Hardware Infrastructure

In the first step, the hardware is automatically generated from an abstract specification. This specification may be provided by the user or it can be the result of an analysis process that generates the best multiprocessor infrastructure as well as the best topology to compute a given application.

As shown in figure 8.8, the description at the highest level is very abstract and specifies only the amount of the processors, their interconnection paradigm and the characteristics of the computing blocks used in the system.

The description is done in XML in conjunction with a document type definition file (DTD). The document type definition describes the syntax as well as valid components and their attributes. At this level, no information about specialized and platform-dependent system components has to be supplied. Valid modules that can be described at this stage are for instance *CPU*, *Memory*, *CommMedium*, *Periphery* and *HWAccelerator*. *CommMedium* is an identifier
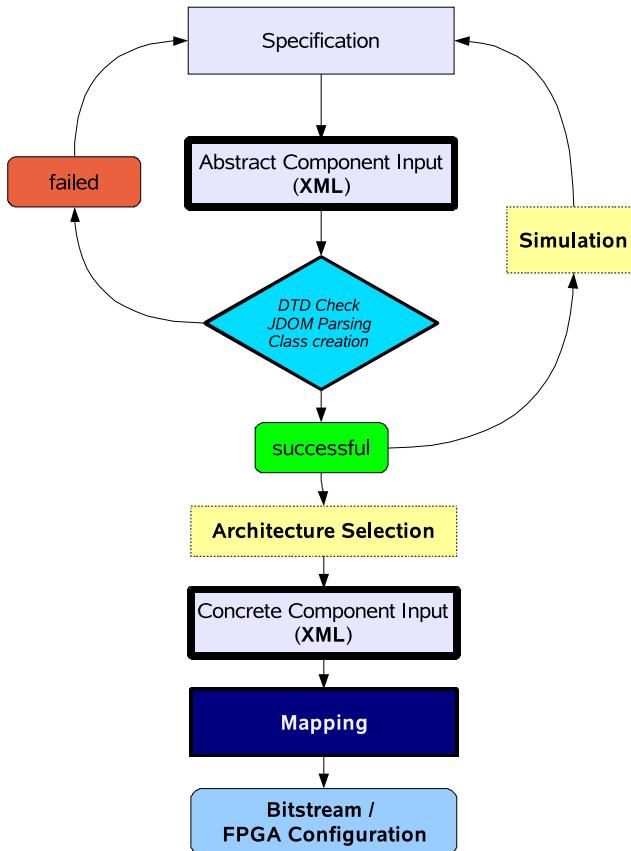
Figure 8.8. Automatic hardware generation flow

for the whole class of communication media. It comprises buses, networks and point-to-point connections. *Periphery* and *HWAccelerator* differ only in their type of connections. While a *HWAccelerator* is used for internal components, *Periphery* is used to describe modules with external connections.

Having specified the system, the description may be functionally simulated to observe the system behaviour and afterwards be validated.

In the last step, the final system can be created by supplying a concrete system description for the target platform.

When transforming an abstract system description into a concrete one, trace-ability must be maintained. Therefore, in each of both steps, the respective XML input file is validated against its document type definition to prevent in-valid component specifications.

Subsequently, the given specification is processed by a Java application consisting of three consecutive phases. In phase one, an XML parser extracts

required information from the XML sources and an internal representation of the resulting XML tree is built up for further use. For each component, an appropriate class is instantiated, which knows how to process its own parameters. Obviously, the application can be easily extended to support new user-designed IP-cores by just adding a corresponding class.

Both parsers – the abstract one as well as the concrete one – share this approach. Whereas the abstract parser has finished its work at this point, the concrete XML parser can now go on processing the gathered information to create the platform-specific hardware information files. Individual mappers are created in the second phase, one for each component and each target platform. In the third and final step, a mapper is invoked to create the desired platform-dependent hardware description files. Again, it has to be emphasized that extending the application to support a new platform can be simply accomplished by adding another mapper class.

Mapper classes for system components are derived from an abstract platform-dependent super class that contains a generic algorithm, which enables derived classes to inherit mapper functions. The current system component that has to be mapped is passed from the corresponding class itself to its mapper class through constructor. The resulting hardware description files mentioned above are then passed to the vendor's tool chain, for example to Xilinx EDK to generate the bitstream for the configuration of the FPGA.

The system does not only enable the user to create hardware information files from a given specification. Existing design can also be loaded from files in which they were saved. Also, system components can be added to or removed from the system in a very comfortable way, and the changes can be rewritten back to the XML file.

A tool called *PinHat* (Platform-independent hardware generation tool) was developed with the goal to integrate all the functionalities and features described above into one single, easy-to-use application. *PinHat* provides a graphical user interface (figure 8.9) that allows the user to comfortably edit the desired system settings and guides him through the process of deriving the concrete system description from the abstract one.

It integrates seamlessly with vendor-supplied toolchains and allows the generation of a bitstream to configure any supported FPGA based on the created concrete specification.

After having adapted the abstract specification to the end-user's needs, *PinHat* offers a transformation wizard, which supports the procedure of refining abstract node elements to concrete ones. First of all, the root element has to be refined by specifying the target toolchain and the target board. This piece of information is used to identify correct specified components when refining all the other abstract nodes. Selecting for instance *Xilinx* as target toolchain and *ML 310* as target board would allow an abstract CPU node to be refined to either a *Microblaze* or alternatively a *PowerPC 405* concrete node element.
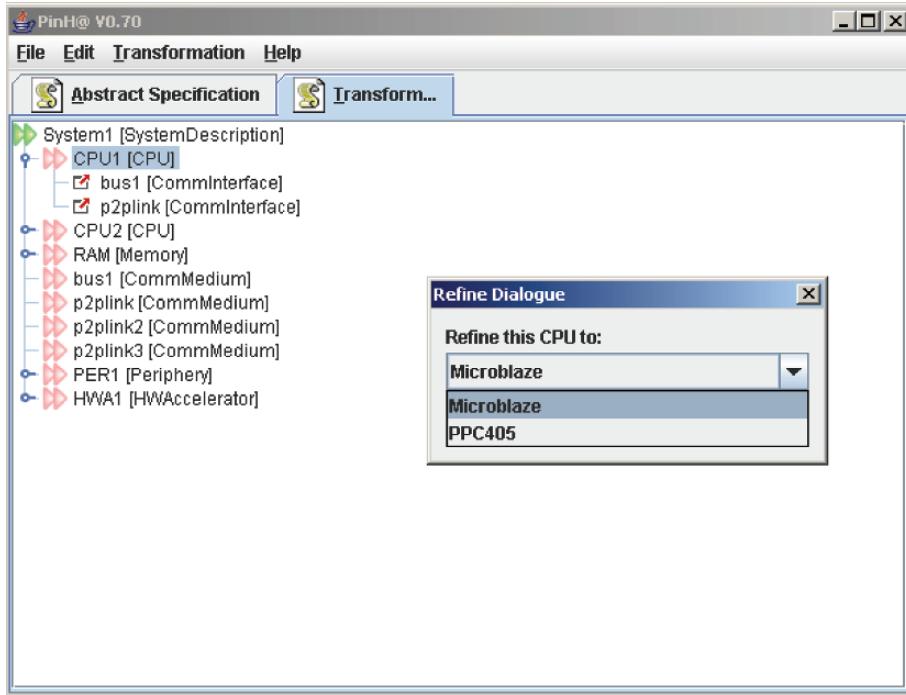
Figure 8.9. The Platform-independent Hardware generation tool (PinHat)

Subsequently, the refinement of all lower-level nodes can be done. *PinHat* has integrated parsers to read out component and board description files where information about peripherals, FPGA pin connections, board architecture, internal component parameters and much more can be stored and thus does not need to be entered manually. This increases the comfort by avoiding the users to enter hundreds of parameters for a single component. Furthermore, this procedure guarantees that no non-existing parameters can be specified. Parameters that are not specified are initialized with standard values that are also part of the parsed description files. The role of *PinHat* in the design flow is illustrated in figure 8.10, where the abstract specification is mapped later to the Xilinx Platform.

2.6 Automatic configuration of the hardware infrastructure

In contrast to the downloading of the bitstream into the FPGA, we understand under automatic configuration the generation of the binary code that will be run on each processor, the configuration of the operating system and the generation of the files needed for system start-up.
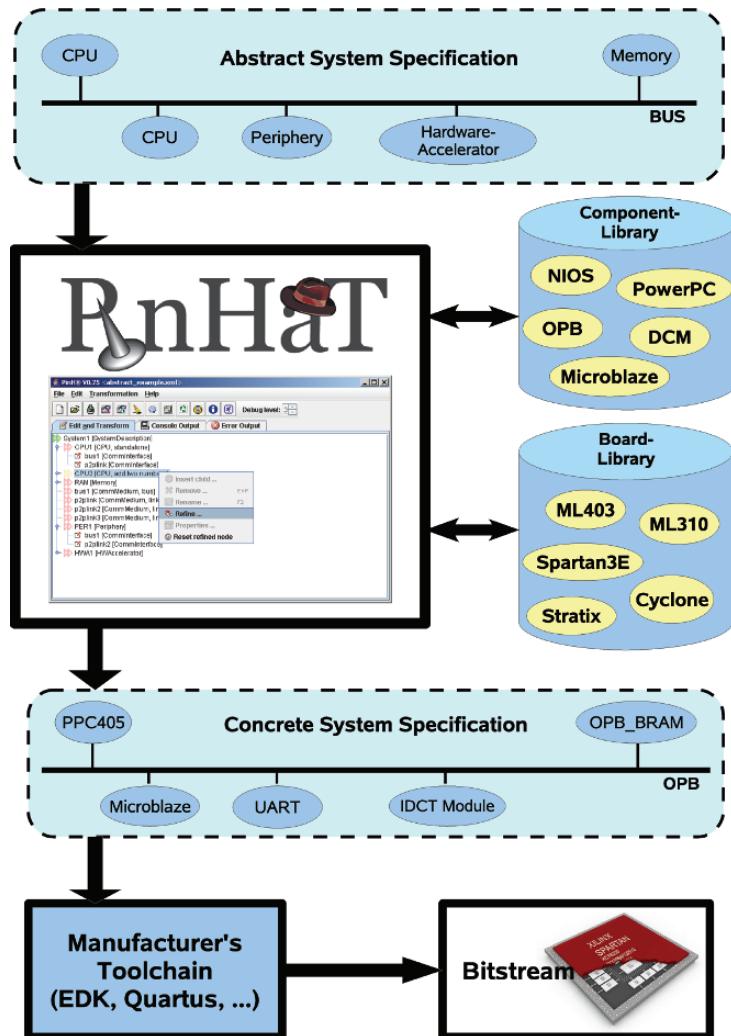
Figure 8.10. The PinHaT framework

Like in the Automatic generation of the hardware infrastructure, we have a *SW-specification*, which may be – as the *HW-specification* – provided by the user or it can be the result of an analysis and code generation process. The SW-specification describes whether a given processor is to be run with a standalone application or if an operating system is foreseen for that processor. For each standalone application, additional information is given about the task it executes; the task allocation results from splitting the application to be computed into a separated set of independent and communicating tasks. The distribution of the jobs and the mapping to the processors is stored in XML, complementing

the abstract component input described in Section 2.5. We are not focusing on the partitioning process here. We assume that this part has been done and that the segments of the overall code that will be implemented on each processor are available.

As the abstract system description is transformed into a concrete one, the mapping 'task → abstract processor' is refined into a mapping 'task → concrete processor' (figure 8.11).

Thereafter, parameters of the software for each processor must be specified: this includes information about either the application (source-code, libraries, etc.) or about the operating system(s) (type, configuration, require functionalities and modules, etc.). Once those information are provided, scripts and compiling files for building the standalone software and the operating systems are generated. Building the standalone software is straightforward because it requires only compiling and linking the files, using vendor tools on the basis of the hardware generated in the previous step. The building of the operating system is however a more complex issue. Depending on the OS chosen, different steps are necessary; for some operating systems, only the kernel-parameters may be configured explicitly, while the file-system and therewith the available programs is pre-build. Other operating systems allow for the complete configuration of the kernel, file-system and the available programs. The configuration of the operating systems is done by generated scripts.

The result of the configuration step is one executable file for each processor in the system. This file can be the same for a pair of processors in the system, if the parameters provided to build the file are the same.

Using vendor tools, the FPGA-configuration bitstream, resulting from the generation of the hardware infrastructure is complemented with the generated executable files. These results in a configuration file containing hardware and software configuration of the whole system for the chosen FPGA.

According to the operation paradigm chosen, we may have the operating system running on all the nodes or on just one node as illustrated in figure 8.12. The last approach makes more sense. In this case, the node attached to the peripherals uses the operating system for a better control of those peripherals. This node will then behave as a master in the system, collecting the data and dispatching them to all the processors involved in the parallel computation of a given application, collect the partial results from the single processors and send them through the peripherals to off-chip components. The operating system can also be made available on many processors, if those processors manage some peripherals separately.

As case study, several implementations were done on FPGA, using the *PinHat* tool. A three processor system consisting of a *PowerPC*, and two *MicroBlaze* processors on the Xilinx ML310 board, featuring a VirtexII Pro 30 FPGA, were built. The primary goal was to test the bandwidth performance
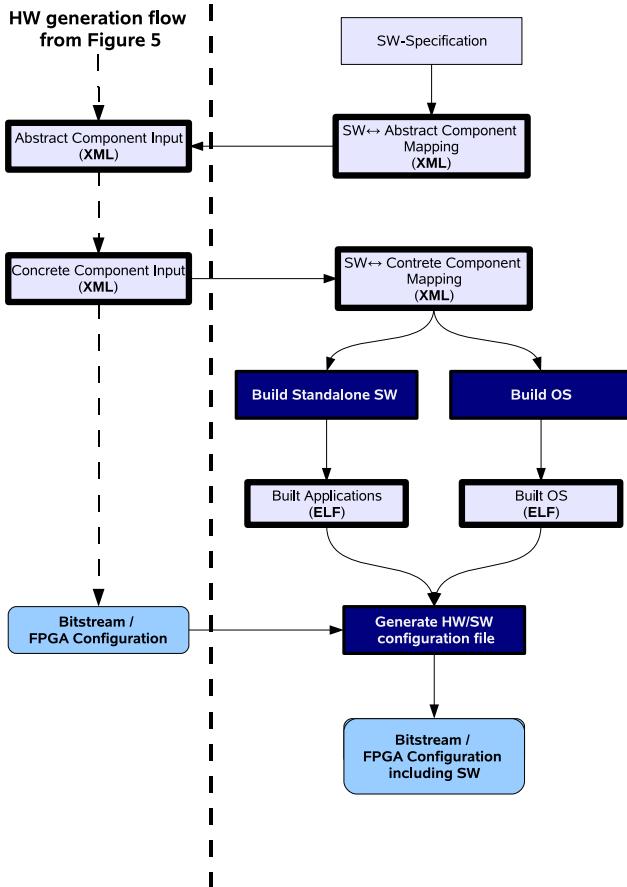
Figure 8.11.   software configuration flow

of a system with multiple rings and a router for the ring interconnection. The design, in a first non-optimized implementation, reveals an average real bandwidth of 36 Mb/s. This includes the complete transaction need to code, send and decode a message.

To have the system running with a real-life application, the singular value decomposition (SVD) on the ML310-Board is implemented, with the number of processors varying from one to eight, and with matrices varying from (4×200) to (128×200). The performance increase because of the use of multiprocessors was shown to be almost linear in the number of microprocessors.

As the bulk of the computations in the SVD is the computation of the dot-products of the column pairs, which is a good candidate for hardware implementation. Custom multiply accumulate (MAC) modules for computing
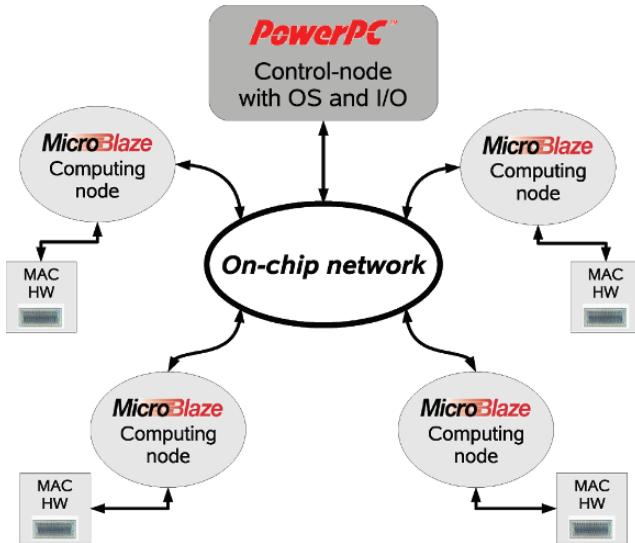
Figure 8.12. 4-Processor infrastructure for the SVD on the ML310-Board

those column-pair multiplications were designed and directly attached con-
nected to the *MicroBlaze* processors through the fast simplex link (FSL) ports
(figure 8.12, to increase the performance of the whole system.

2.7   Adaptivity

As stated at the beginning of the chapter, the adaptivity can be used for
several purposes. According to the application, a physical topology consisting
of a given arrangement of routers and processors may be built to match the best
computation paradigm of that application. The topology can be modified, even
at run-time to better adapt to the changes observed in the application.

On the other hand, the adaptivity of the system can be reached by using
reconfigurable hardware accelerators, provide that the target device supports
partial reconfiguration.

Designing partially reconfigurable systems with unlimited capabilities is a
complex engineering task that requires a lot of hand work. To avoid this and
keep the thing very simple, a template for partial reconfiguration can first be
created for the target device. In this template, fixed locations must be selected,
where the reconfiguration is allowed to take place. It acts as place holder to
accommodate reconfigurable modules at run-time. Any hardware accelerator
can be plugged on the hardware block at run-time by means of partial recon-
figuration. This step can even be performed from within the device using the
appropriated port like the ICAP in the case of Xilinx devices.

3.   Conclusion

In this chapter, we have addressed the system on programmable chip paradigm and provided some advantages of using those architectures. The main components needed to build such a system were presented, with a focus on the leading manufacturers. The goal was not a comparative study on the different system on programmable chip infrastructure. We rather wanted to provide the user a brief view on the existing solution.

In the second part of the chapter, we have presented a design approach for multiprocessor systems on FPGAs. Using the reconfiguration, it is possible to modify the hardware infrastructure to adapt it to the best computation paradigm of the application being computed.

We have presented a framework to specify and implement a complete system without knowledge of the vendor tools. Obviously, all what is feasible with the *PinHat* tool is also feasible with current vendor tools. However, the complexity and the difficulty of using those tools do not allow any new comer to generate design with it. The goal here was to hide the complexity in the design of multiprocessor, which is very limited in the current tool to the user and allow him to focus on the efficient analysis and partitioning of its application.

We believe that multiprocessing on FPGA has a great potential to speed-up computation in embedded system. To facilitate their use, systems like the *PinHat* are welcome to hide the complexity of the hardware generation and let the designer focus on the analysis of the application and the definition of the best adapted structure for its computation.

The adaptivity of the whole system is provided through the use of partial reconfiguration to exchange running hardware modules at run-time. However, to overcome the difficulty of designing for partial reconfiguration with the current tool, a template-based approach is recommended, which foresee predefined location for placing components at run-time.

Chapter 9

APPLICATIONS

Up to now, we have focussed on the technology behind reconfigurable devices, their integration in different systems and their programmability at different level of abstraction. The natural question which arises is the one to know, in which area does reconfigurable computing could be helpful? If areas of application exist, then we would also like to know what the gain of using reconfigurable technology is, compared with other processing devices used before? The gain of using a solution rather another one is usually expressed as the computation speed, or as the improvement in power consumption, or as the price, or even as the easiness in the programmability. In [57], Craven and Athanas provided a performace/price comparative study between FPGA-based high-performance computing machines and traditional supercomputers. On the basis of the gain in accelerating floating-point computations, they came to the conclusion that the price to pay was too high compared with the marginal gain obtained, when using FPGAs in supercomputers. Other researchers have reported amazing gains with FPGA-based computing machines in several fields of computation, and several companies that have been offering FPGA-based computing solutions a decade ago are still operating and even growing. Our goal in this chapter is to present some applications that benefit somehow from an implementation on reconfigurable devices. Almost all the experiments reported were done on FPGA platforms. We will not report all the experiments in all details. We rather focus on the coarse-grained structure of the applications and point out where the reconfiguration can be helpful. We are primary concerned with the speed improvement and the use of flexibility of reconfigurable computing systems.

Despite the very large number of applications that could benefit from an implementation in FPGAs, we have selected only 6 domains of application

that we present here: pattern matching, video streaming, signal processing using distributed arithmetic, control, and super computing.

1.   Pattern Matching

Pattern matching can be defined as the process of checking if a character string is part of a longer sequence of characters. Pattern matching is used in a large variety of fields in computer science. In text processing programs such as Microsoft Word, pattern matching is used in the search function. The purpose is to match the keyword being searched against a sequence of characters that build the complete text. In database information retrieval, the content of different fields of a given database entry are matched against the sequence of characters that build the user request. Searching in genetical database also use pattern matching to match the content of character sequence from a database entries with the sequence of characters that build a given query. In this case, the alphabet is built upon a set of a genomic characters. Speech recognition and other pattern recognition tools also use pattern matching as basic operations on top of which complex intelligent functions may be built, to better classify the audio sequences. Other applications using pattern matching are: dictionary implementation, spam avoidance, network intrusion detection and content surveillance.

Because text mining, whose primary role is the categorization of documents, makes a heavy use of pattern matching, we choose in this section to present the use of pattern matching in text mining and to point out the possible use of reconfiguration.

Documents categorization is the process of assigning a given set of documents from a database to a given set of categories. The categories can either be manually defined by a user or be computed automatically by a software tool. In the first case, the categorization is supervised and in the second case, we have an unsupervised categorization. In categorization, the first step usually consist of indexing the collection of available documents. This is usually done through the so-called vector space representation. In this model, a document is represented as a vector of key words or term present in the document. A complete collection of n documents over a list of m keywords is then represented by a term by documents matrix $A \in R^m \times R^n$. An entry a_{ij} in A represents the frequency of the word i in the document j. The term by documents matrix is then used for indexing purpose or statistical analysis like latent semantic indexing (LSI) [67]. Building a term by documents matrix is done by scanning the documents of the given collection to find the appearance of key words and return the corresponding entry in the matrix for each document. Pattern matching is therefore used for this purpose.

The first advantage of using the reconfigurable device here is the inherent fine-grained parallelism that characterizes the search. Many different words

can be searched for in parallel by matching the input text against a set of words on different paths. The second advantage is the possibility of quickly exchanging the list of searched words by means or reconfiguration.

Pattern matching was investigated in different work using different approaches on FPGAs [52] [85] [104] [151] [101] [180] [34], each with a different overhead in compiling the set of words down to the hardware and different capacities. We define the capacity of a search engine as the number of words that can de searched for in parallel. A large capacity also means a high complexity of the function to be implemented in hardware, which in turn means a large amount of resources. The goal in implementing a search engine in hardware is to have a maximal hardware utilization, i.e. as many words as possible that can be searched for in parallel. We present in this section various hardware implementation of the pattern matching, each with its advantages and drawbacks.

1.1 The Sliding Windows Approach

One approach for text searching is the so-called sliding window (SL) [85] [151]. In the *1-keyword* version, the target word is stored in one register, each character being stored in one register field consisting of a byte. The length of the register is equal to the length of the word it contains. The text is streamed through a separate shift register, whose length is the same as that of the keyword. For each character of the given keyword stored as a byte in one register field, an 8-bit comparator is used to compare this character with the corresponding character of the text, which streams through the shift register. A hit occurs when all the comparators return the value true.

The sliding window can be extended to check a match of multiple patterns in parallel. Each target word will then be stored in one register and will have as many comparators as required. The length of the window (the number of characters in the shift register) is defined by the segment of the text to be processed. Words with length bigger than the length of the window cannot be handled. To overcome this problem, one can define the length of text to be considered as the maximum length over all the target words. In this case, all the words with a length smaller than the maximum length should be filled with do not care characters to be handled correctly. The example of figure 9.1 shows the structure of a sliding windows with three key words.

The main advantage of this approach resides in the reconfiguration. Because each keyword is stored in an independent register, the reconfiguration can happen without affecting the other words, thus providing the possibility to gradually and quickly modify the dictionary.[1]

---

[1]With dictionary, we mean here the set of keyword compiled into the reconfigurable device.
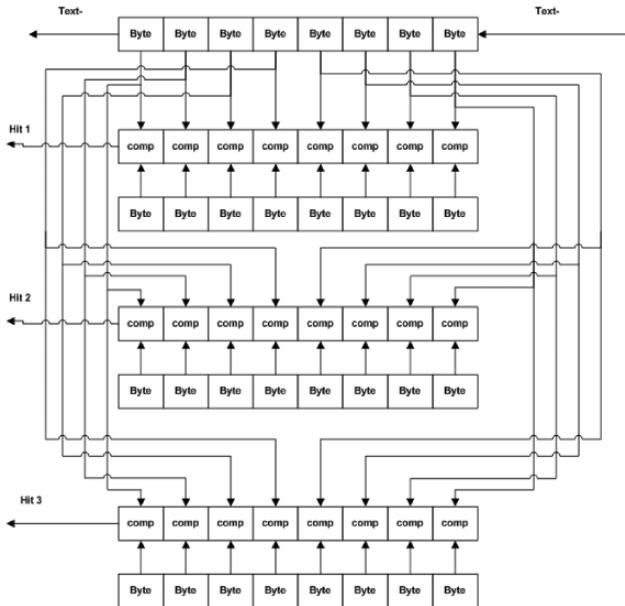
Figure 9.1.   Sliding windows for the search of three words in parallel

This method however has a main drawback, which is the redundancy in the amount of register fields used to store the words as well as the number of comparators. Redundancy in the sliding windows approach reduce the capacity of such implementation, thus making its use not competitive. In the case of the Xilinx Virtex FPGA XCV300 with a maximum of 6144 flip flops, a sliding windows of length 10 needs $10 \times 8 = 80$ flip flops to store one target word in the device. Consider that the device placement and routing permits an utilization of 80%. The number of words that can be folded in such an FPGA will theoretically be in the range of 60, which is small compared with the number of words that can be folded in the same divide with a more efficient implementation. A nice approach would be to just perform the comparison on a given alphabet consisting of the set of common character and to use the result in the evaluation of each single word. This would lead to a reduce amount of register fields and a reduce amount of comparators.

1.2   Hashed Table-Based Text Searching

The hashed table-based text searching approach was implemented on the SPLASH II Platform [180]. The text to be processed is streamed in a pair of consecutive characters called *superbyte*. An incoming *superbyte* is mapped on one character of the target alphabet. Each non-alphabet character is mapped to a delimiter. To determine whether or not a word is in the target list, a hashing

technique is used. A hash table in memory is used to store the value of a presence bit for a particular target word. An entry zero in this table means that the word is not present and a value one means that the word will probably be present. A hash register (with a length of 22 bit in the SPLASH implementation), which is initially set to zero, is incrementally modified by the incoming *superbytes*. At the end of a word marked by a delimiter, the hash register is set to zero to allow the next word to be processed. The hash register is used to address the hash table of 2^{22} pseudo-random mappings of words. The modification of the hash register happens as follows: When a non-delimiter *superbyte* is encountered, the contents of the 22-bit hash register is updated by first XOR-ing the upper 16-bit of the register with the incoming *superbyte* and a value of a hash function. The modified value of the hash register is then circularly shifted by a fixed number of bits. Upon reaching the end of a word, the content of the register is used to address the hash table and determine the value of the presence bit. To reduce the likelihood of false hits, a number of independent hash functions is calculated for each word of the text, with the stipulation that each hash function lookup of a word must result in a hit for that word to be counted as a target word.

Estimations on performance of the SPLASH text searching have been done mainly on the basis of the critical path length returned by the place and route tool. Communication between the host computer and the boards such as memory latency has not been taken into consideration.

1.3   Automaton-Based Text Searching

It is well known that any regular grammar can be recognized by a deterministic finite state machine (FSM). In an automaton-based search algorithm, a finite state machine is built on the basis of the target words. The target words define a regular grammar that is compiled in an automaton acting as a recognizer for that grammar. When scanning a text, the automaton changes its state with the appearance of characters. Upon reaching an end state, a hit occurs and the corresponding word is set to be found. One advantage of the FSM-based search machine is the elimination of the preprocessing step done in many other methods to remove stop words (such as 'the', 'to', 'for' etc., which does not affect the meaning of statements) from documents. In [52], the implementation of the hardware-based retrieval machine is done using a memory to store the transition table. Two registers hold the current state and the incoming character. The logic is only used to decode the address of the next state in memory and to set the correct values in the registers.

The performance (speed) of such an implementation is governed by the path (RAM output → state register → RAM address → RAM output), which can considerably slow down the search process, because of the multiple memory accesses. To increase the speed of an FSM implementation, memory accesses

can be suppressed by compiling the transition table in hardware and having a mechanism to execute the state machine. Folding big transition tables in FPGAs can be difficult because of the limited capacity of those devices. Transition tables for word recognizers are often full of backtracking edges and crossing edges (edge that connects two nodes on different paths in the FSM-tree). With the sequential structure of the memory-based recognizer, it is almost impossible to eliminate the backtracking edges, which constitute in most cases the biggest part of the transition table.

The SPACE (Scalable Parallel Architecture for Concurrency Experiments) machine [104] [105] makes use of the parallel structure of FPGAs to eliminate backtracking edge from the FSM. This search machine is a FSM-like implementation for text searching capable of handling a search over a list of almost 100 target words. The search is performed simultaneously on a board with 16 CAL1024 FPGAs without using a memory to store the transition table. Each key word is compiled in a separate FSM that is mapped to the hardware device. The text is then streamed into the device, character by character in ASCII form. For the case insensitivity purpose the 8-bit characters are mapped by the preprocessor to 5-bit characters. With the incoming characters, which clock the flip flops, all the FSM move simultaneously to their next state. Upon reaching an end state, the corresponding accumulator is incremented and the result is returned. The backtracking edges appearing in the formal representation of the FSM become redundant and will not be taken into consideration. Figure 9.2a shows a transition graph of the FSM for the word 'conte'. The transition table requires 5×6 memory locations to be stored (figure9.2b). For the same FSM, the hardware representation will need only 4 flip flops, 4 AND gates, and 5 comparators (figure 9.2c).

While having the advantage of replacing a word already folded in the FPGA with a new target word of the same length, the SPACE implementation presents two major inconveniences: First, it does not take into consideration words that share a common prefix, thus increasing the amount of resources used to store the character states. Second, it uses one comparator for each character of a target word, which increases the resource needed to implement the comparators. Those two factors lead to redundancy in flip flop and look-up table utilization as it is the case in the sliding window recognizer. On the SPACE board, with a total of 8192 flip flops, it has been possible to implement a list of only 96 target words with average length 12.

When taking into consideration the common prefix of words, it is possible to save a considerable amount of flip flops. For this purpose, the search machine could be implemented as an automaton recognizer, common to all the target words. This approach was first presented in [34]. As shown in figure 9.3, the resulting structure is a tree in which a path from the root to a leaf determines the appearance of a corresponding key word in the streamed text. Words that share

	c	o	n	t	e
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
4	1	0	0	0	5
5	1	0	0	0	0
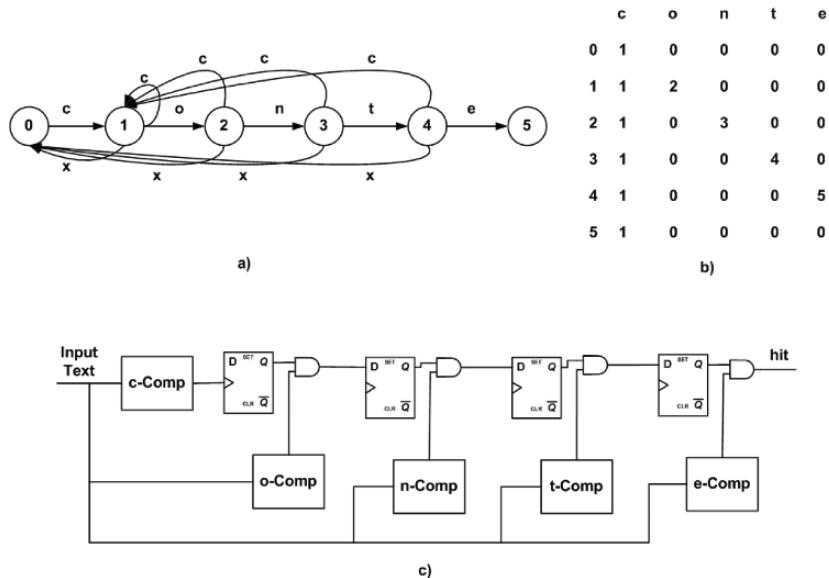
Figure 9.2. FSM recognizers for the word 'conte': a) sate diagram, b) transition table, c) basis structure the hardware implementation: 4 flip flops will be need to code a 5×6 transition table

a common prefix use a common path from the root corresponding to the length of their common prefix. A split occurs at the node where the common prefix ends. In the hardware implementation of a group of words with a common prefix, common flip flops will be used to implement the common path (figure 9.3a).

For each character in the target alphabet, only one comparator is needed. The comparison occurs in this case, once at a particular location in the device.
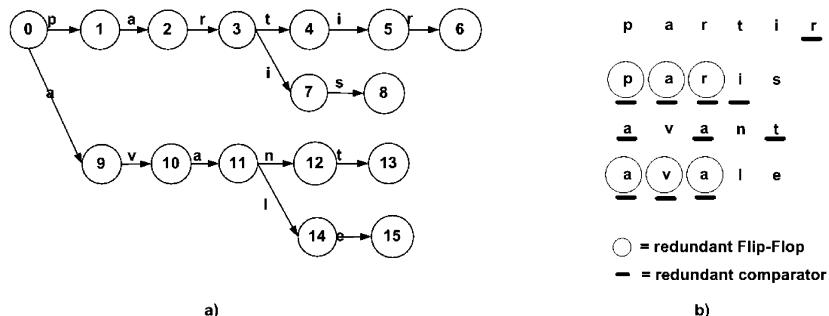
Figure 9.3. a) Use of the common prefix to reduce the number of flip flops of the common word detector for 'partir', 'paris', 'avale','avant'. b) implementation without use of common prefix and common comparator set

Incoming characters can be directly sent to a set of all possible comparators. Upon matching a particular one, a corresponding signal is sent to all the flip flops, which need the result of that comparison. This method will reduce the number of comparators needed almost by the sum of the length of all target words.

The overall structure of the search engine previously explained is given in figure 9.4. It consists of an array of comparators to decode the characters of the FSM alphabet, a state decoder that moves the state machine in the next states and a preprocessor to map incoming 8-bit characters to 5-bit characters, thus mapping all the characters to lower cases. As case insensitivity is considered in most application in information retrieval, the preprocessor is designed to map upper and lower case characters to the binary code of 1 to 26 and the rest of character to 0 [104]. Characters are streamed to the device in ASCII notation. An incoming 8-bit character triggers the clock and is mapped to a 5-bit character that is sent to the comparator array. All the 5-bit signals are distributed to all the comparators that operate in parallel to check if a match of the incoming character with an alphabet character occurs. If a match occurs for the comparator k, the output signal $char k$ is set to one and all the others are set to zero. If no match occurs, all the output signals are set to be zero. Figure 9.5 illustrates this concept on the basis of the word 'tictic'.

The one-hot approach (figure 9.2c) is probably the best implementation for the kind of automaton realized in the state decoder. First, it takes a major advantage toward the regular structure of the FPGA, and it is the fastest way to implement FSM in hardware [142]. Moreover, there is no need to incorporate backtracking edges in a suitable hardware representation. To implement a set of different words, the resulting tree structure of the common FSM is mapped in hardware. All the leafs corresponding to target words are connected to the hit port. The width of the hit port is determined by the number of positions needed to code the number of target words. Upon matching a particular word,
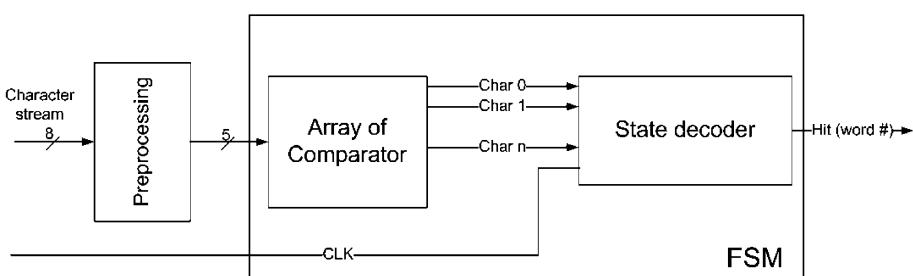
Figure 9.4. Basic structure of a FSM-based words recognizer that exploits the common prefix and a common set of characters
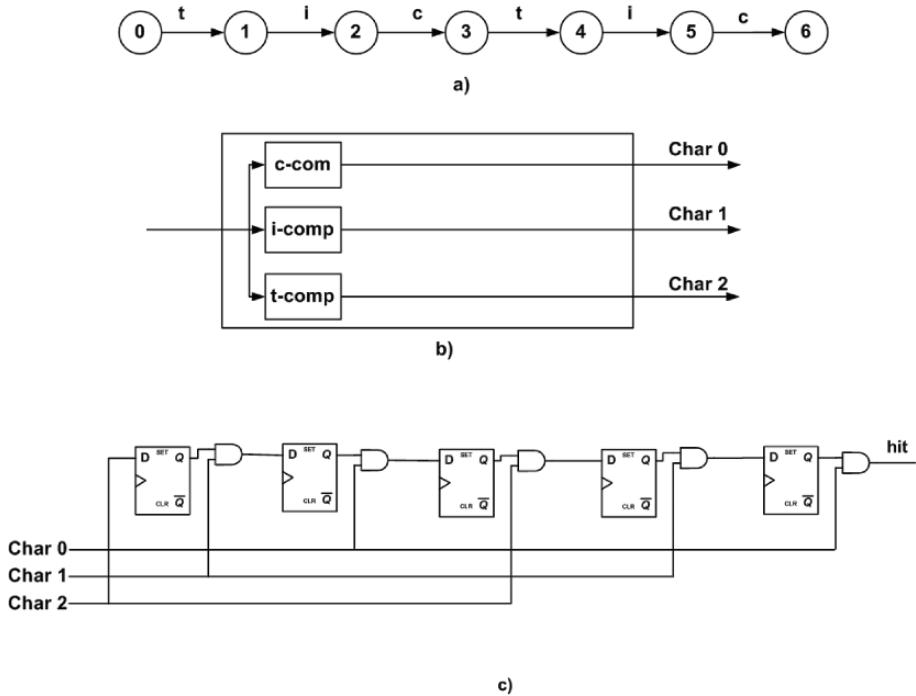
Figure 9.5. Processing steps of the FSM for the word 'tictic'

one leaf of the automaton is reached, and the hit output is set as to the index of the matched target word.

1.4 Automaton generation

A tool was provided in [34] to allow for an automatic generation of a term by document matrix from a document collection with a given list of target words. This tool automatically generates a VHDL description of the search machine, and the synthesis can be done with available vendors or third party tools. The tool is based on the method described in [14] to generate the corresponding tree structure that is then mapped in hardware. The resulting state machine is optimized (in terms of number of states) and is able to solve the Mississippi problem[2] for a set of target words. With the union, concatenation, Kleene star (Kleene closure) operations, all the basic FSMs (FSM related to one word) are used to build one FSM capable of recognizing all the target words. A set of final states is linked to the appearance of target words in a streaming text. The resulting transition table is then translated into VHDL code in such a way that

---

[2]Can the state machine recognize the word 'issip' in a file containing the word Mississippi?[52]

no backtracking and crossing edges appear in the configuration. The resulting VHDL code is then synthesized, mapped, placed, routed and configured to be downloaded into the FPGA with the help of the generated scripts. The document collection is then scanned and a column vector representing the weight of target words in this document is returned for each document. The performance of this approach was evaluated using the Spyder Virtex board from the FZI Karlsruhe [214], a PCI board featuring a Xilinx Virtex XCV300. A list of 330 key words with an average of 10 characters each could then be compiled into the FPGA, leading to an automaton with 2370 states. The utilization of the FPGA was 59% (1817 CLB slices out of 3072 slices), including bus interface. The stream rate of the characters through the FPGA was by 3 million[3] characters per second.

2.   Video Streaming

Video streaming is the process of performing computations on video data, which are streamed through a given system, picture after picture.

Implementation of video streaming on FPGA have attracted several researchers in the past, resulting in the building of various platforms [39] [115] [198] [72] [207] [140] [201] for various computations. Two main reasons are always stated as motivation for implementation video streaming in FPGAs: the performance, which results from using the exploitation of the inherent parallelism in the target applications to build a tailored hardware architecture, and the adaptivity, which can be used to adjust the overall system by exchanging parts of the computing blocks with better adapted ones.

In [193], Richard G.S discusses the idea of parameterized program generation of convolution filters in an FPGA. A 2-D filter is assembled from a set of multipliers and adders, which are in turn generated from a canonical serial-parallel multiplier stage. Atmel application notes [13] discuss 3 x 3 convolution implementations with run-time reconfigurable vector multiplier in Atmel FPGAs. To overcome the difficulties of programming devices with classic Hardware Description Languages (HDL) such as VHDL and Verliog, Celoxica has developed the Handel-C language. Handel-C is essentially an extended subset of the standard ANSI-C language, specifically designed for use in a hardware environment. The Celoxica Handel-C compiler, the *DK1* development environment includes a tool, the *PixelStream*, for an easy generation of video processing functions for FPGA implementations. *PixelStream* offers a set of basic modules and functions that can be (graphically) combined to build a more complex datapath in FPGA. The resulting design can be mapped to one of the Celoxica FPGA development boards.

---

[3]4 million if less than 255 words are compiled into the word recognizer.

We will not focus in this section on the details of video or image processing. A comprehensive description of video processing algorithms and their hardware implementation on FPGA is provided in [153].

The Sonic architecture [115] is a configurable computing platform for acceleration and real-time video image processing. The platform consists of plug-in processing elements (PIPEs), which are FPGA daughter card that can be mounted on a PCI-board plugged on a PCI-slot of a workstation. A PIPEflow bus exists to connect adjacent PIPES, while the available PIPE provide global connection to the PIPES. Sonic's architecture exploits the reconfiguration capabilities of the PIPES to adapt part of the computation flow at run-time.

The Splash [97] architecture was also used in video processing. Its systolic array structure makes it well suited to image processing.

The ESM platform introduced in Section 7.2 presents an optimal pipelined architecture for the modular implementation of video streams. Its organization in exchangeable slots, each of which can be reconfigured at run-time to perform a different computation, makes it a viable platform in video streaming. Moreover, the communication infrastructure available on the platform provides an unlimited access of modules to their data, no matter on which slot they are placed, thus increasing the degree of flexibility in the system.

The computation on video frames is usually performed in different steps, while the pictures stream through the datapath. It therefore presents a nice structure for a pipelined computation. This has led to a chained architecture on the basis of which most video streaming systems are built. The chain consist of a set of modules, each of which is specialized for a given computation. The first module on the chain is in charge of capturing the video frames, while the last module output the processed frames. Output can be rendered on a monitor or can be sent as compressed data over a network. Between the first and the last modules, several computational modules can be used according to the algorithm implemented. A module can be implemented in hardware or in software according to the goals. While software provides a great degree of flexibility, it is usually not fast enough to carry the challenging computations required in video applications. ASICs can be used to implement the computational demanding parts; however, ASIC does not provide the flexibility needed in many systems. On a reconfigurable platform, the flexibility of a processor can be combined with the efficiency of ASICs to build a high-performance flexible system. The partial reconfiguration capability of reconfigurable devices provides the possibility to replace a given module on the chain at run-time.

Most of the video capture modules provide the frames to the system on a pixel by pixel manner, leading to a serial computation on the incoming pixels. As many algorithms need the neighbourhood of a pixel to compute its new value, a complete block must be stored and processed for each pixel. Capturing

the neighbourhood of a pixel is often done using a sliding window data structure with varying size. The size of the sliding window corresponds to that of neighbour region of a pixel needed for the computation of the new value. As shown in figure 9.6, a sliding window is a data structure used to sequentially capture the neighbourhood of pixels in a given image.

A given set of buffers (FIFO) is used to update the windows. The number of FIFOs vary according to the size of the window. In each step, a pixel is read from the memory and placed in the lower left cell of the window. Up to the upper right pixel that is disposed, i.e. outputted, all the pixels in the right part of the window are placed at the queue of the FIFOs one level higher. The processing part of the video is a normal image processing algorithm combining some of the basic operators like:

- Median filtering

- Basic morphological operations

- Convolution

- Edge detection.

In the field of video compression, the processing is usually done in a block by block manner, different from the sliding window. However, the overall structure is almost the same. The neighbourhood must always be saved to provide the necessary data for the computation of a new pixel value.

As explained earlier in this section and as figure 9.7 shows, the architecture for a video streaming system is usually built on a modular and chained basis.

The first module deals with the image captured from an image source. This can be a camera or a network module that collects the picture through a network channel, or any other source. The frames are alternately written to the RAM1 and RAM2 by the capture module. The second module collects the pictures from the RAM1 or RAM2, if this is not in use by the first module, builds
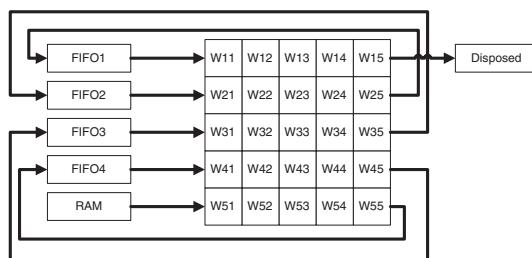
Figure 9.6.   Implementation of a 5 × 5 sliding windows
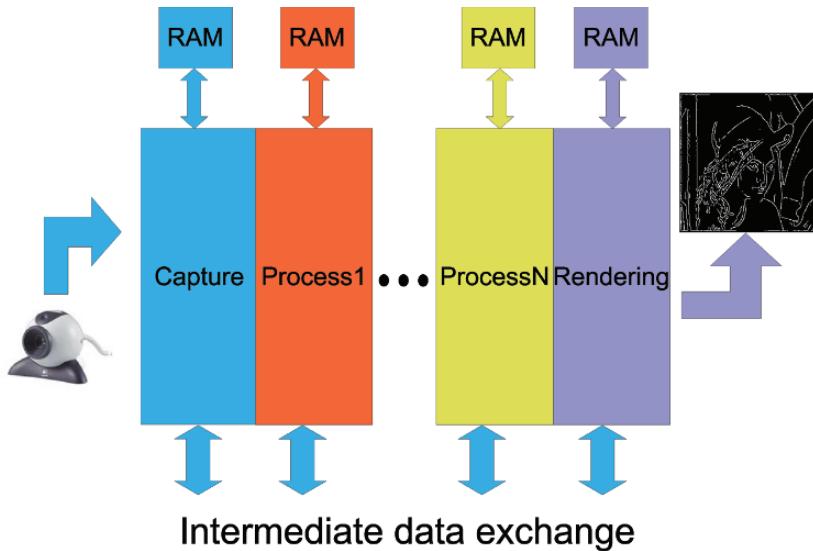
Figure 9.7. A modular architecture for video streaming on the ESM

the sliding windows and passes it to the third module, which processes the pixel and saves it in its own memory or directly passes it to the next module. This architecture presents a pipelined computation in which the computational blocks are the modules that process the data frames. RAMs are used to temporally store frames between two modules, thus allowing a frame to stream from RAM to RAM and the processed pictures to the output.

2.1 Enabling Adaptivity through Reconfiguration

An adaptive video streaming system is characterized by its ability to optimize the computation according to changing environmental condition. In most cases, only one module on the computation chain must be changed while the rest keep running. The video capture module for example can be changed if we want to optimize the conversion of pixel to match the current brightness or the current landscape. It is also possible to change the video source from camera to a new one with different characteristics or the source can be for instance a network channel collecting the frames in a compressed form. In an adaptive system, the functionality of a module on the computation path should be changed very fast and without affecting the rest of the system. Two possibilities exist for this purpose. The first one consists of providing some parameters to the module to instruct it to switch from one algorithm to the next one. However, the structure of the basic algorithms are not always the same. A Sobel filter cannot be changed in a Laplace filter by just changing the parameters. This is also true for a median operator that cannot be replaced

by a Gauss operator by just changing the parameters. The network capture module and the camera capture module require two different algorithms for collecting the pixels and bringing them in the system. The second possibility consists of replacing the complete module at run-time with a module of the same size, but different in its structure, while the rest of the system keeps running. Reconfigurable devices in general and FPGAs in particular fulfill this requirements. Many available FPGAs can be partly configured, while the rest of the system is running. Moreover, many FPGAs provide small size on-chip memories, able to hold part of a frame (the so-called region of interest). It is therefore possible to perform many computations in parallel on different regions of interest, thus increasing the performance and flexibility of video applications.

3.   Distributed Arithmetic

Distributed arithmetic (DA) is certainly one of the most powerful tool for the computation of the product of two vector products, one of which is constant, i.e it consists of constant values. DA exploits the nature of LUT-based computation provided by the FPGAs by storing in an LUT, all the possible results for a set of variable combinations. Computation at run-time only consists of retrieving the results from the LUT, where they were previously stored. The elements of the variable vector are used to address the look-up table and retrieve partial sums in a bit-serial (1-BAAT, 1 Bit At A Time) manner. Investigations in distributed arithmetic have been done in [215] [158] [223] [224] [33] [95] [63] [62] [31].

One of the notorious DA-contribution is the work of White [215]. He proposed the use of ROMs to store the precomputed values. The surrounding logic to access the ROM and retrieve the partial sums had to be implemented on a separate chip. Because of this moribund architecture, distributed arithmetic could not be successful. With the appearance of SRAM based FPGAs, DA became an interesting alternative to implement signal processing applications in FPGA [158] [223] [224]. Because of the availability of SRAMs in those FPGAs, the precomputed values could now be stored in the same chip as the surrounding logic.

In [33], the design automation of DA architecture was investigated, and a tool was designed to help the user in the code generation of a complete DA design, and to perform investigations on the various tradeoffs. Also, an initial attempt to implement DA for floating-point numbers to increase the accuracy was presented. However, the amount of memory required to implement such a solution makes it applicable only on small examples.

The idea behind the distributed arithmetic is to distribute the bits of one operand across the equation to be computed. This operation results in a new one, which can then be computed in a more efficient way. The product of two vectors X and A is given by the following equation:

$$Z = X \times A = \sum_{i=0}^{n}(X_i \times A_i) \tag{3.1}$$

We assume that $A = (A_1, \cdots, A_n)$ is a constant vector of dimension n and $X = (X_1, \cdots X_n)$ is a variable vector of the same dimension. With the binary representation, $\sum_{j=0}^{w-1} X_{ij}2^j$, of X_i, where w is the width of the variables and $X_{ij} \in \{0, 1\}$ is the j-th bit of X_i, equation (3.1) can be rewritten as:

$$Z = \sum_{i=0}^{n} A_i \times \sum_{j=0}^{w-1} X_{ij}2^j = \sum_{j=0}^{w-1} 2^j \sum_{i=0}^{n} X_{ij} A_i \tag{3.2}$$

Spreading equation 3.2 for a better understanding leads to equation 3.3

$$\begin{aligned} Z = \ &(X_{00} \times A_0 + X_{10} \times A_1 + ... + X_{n0} \times A_n)2^0 \\ &+ (X_{01} \times A_0 + X_{11} \times A_1 + ... + X_{n1} \times A_n)2^1 \\ &... \\ &+ (X_{0(w-1)} \times A_0 + X_{1(w-1)} \times A_1 + ... + X_{n(w-1)} \times A_n)2^{(w-1)} \end{aligned} \tag{3.3}$$

Equation 3.2 represents the general form of a distributed arithmetic. It shows how the variable operands are distributed in the equation. Each bit of each variable operand contributes only once to the sum $\sum_{i=0}^{n} X_{ij}A_i$. Because $X_{ij} \in \{0, 1\}$, the number of possible values $\sum_{i=0}^{n} X_{ij} A_i$ can take is limited to 2^n. Therefore, they can be precomputed and stored in a look-up table, provide that enough place is available in the LUT. Lets call such a look-up table a *distributed arithmetic look-up table (DALUT)*. The DALUT has the size $w \times 2^n$ bits. At run-time, the n-tupel $(X_{1j}, X_{2j}, ..., X_{nj})$ will be used to address the DALUT and retrieve the partial sums $\sum_{i=0}^{n} X_{ij} A_i$. the n-tupel $(0, 0, ..., 0)$ will be used to address the location 0, $(0, 0, ..., 1)$ is used to address the location 1, and $(1, 1, ..., 1)$ is used to address the location $2^n - 1$.

The complete dot-product Z requires w steps to be computed in a 1-BAAT manner. At step j, the j-th bit of each variable are used to address the DA-LUT and retrieved the value $\sum_{i=0}^{n} X_{ij} A_i$. This value is then left shifted by a factor j (which corresponds to a multiplication by 2^j) and accumulated. After w accumulations, the values of the dot-product can be collected. The DA datapath for this computation is obvious as figure 9.8 shows.
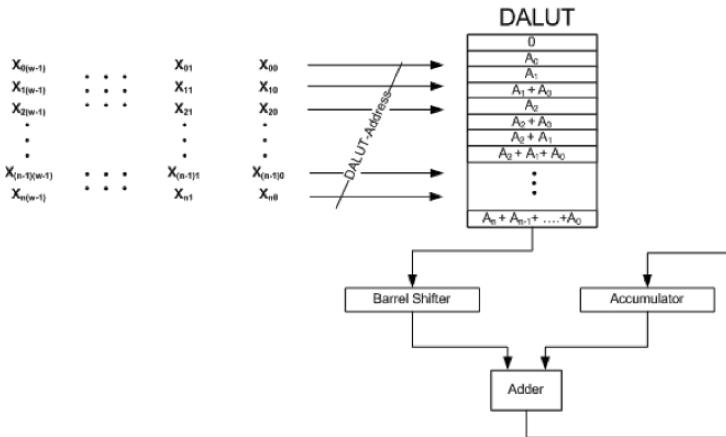
Figure 9.8.   Architecture of a distributed arithmetic datapath

The DALUT address consists of a set of bits, each of which belongs to a component of the input vector. A barrel shifter is used to shift the number retrieved from the DALUT at the j-th step on j positions. The shifted value is then added to the value in the accumulator. After w steps, the results are collected from the accumulator.

Many enhancements can be done on a DA implementation. The size of the DALUT for example can be halved if only positive values are stored. In this case, the sign bit, which is the first bit of a number, will be used to decide whether the retrieved value should be added to or subtracted from the accumulated sum. On the other hand, it is obvious that all the bit operations, i.e. the retrievals of a value from the DALUT are independent from each other. The computation can therefore be performed in parallel. The degree of parallelism in a given implementation depends on the available memory to implement the DALUTs. In the case where w DALUTs and datapaths can be instantiated in parallel, the retrieval of all partial sums can be done in only one step, meaning that the complete computation can be done in only one step instead of w as in the serial case.

In general, if k DALUTs are instantiated in parallel, i.e. for a computation performed on a k-BAAT basis, then w/k steps are required to retrieve all the partial sums, which can be directly accumulated, thus leading to a run-time of w/k steps. Figure 9.9 shows a datapath for the computation of the DA.

The input values are segmented in different fields, each of which is assigned to a datapath for the retrieval of the partial sums from the corresponding DALUT. The retrieved values from the k DALUTs are shifted by the corresponding values and added to the accumulated sum. After w/k steps, the result can be collected from the accumulator.
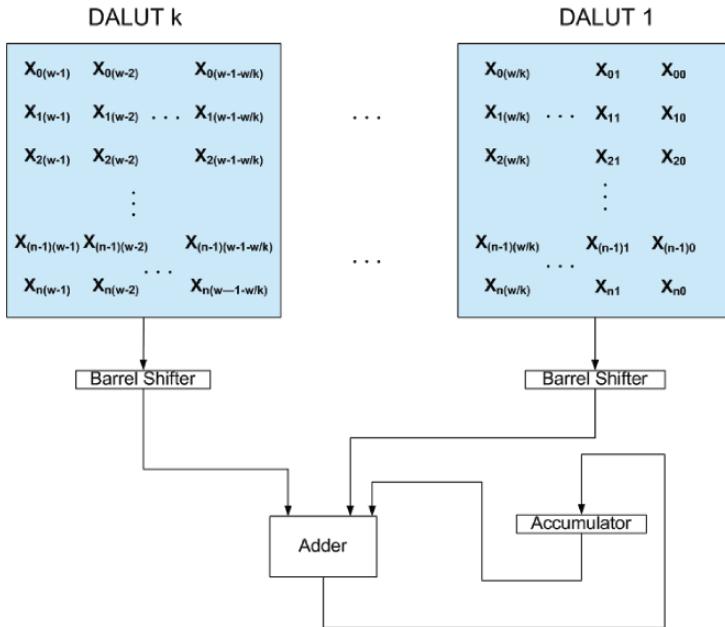
Figure 9.9. k-parallel distributed arithmetic datapath

3.1 Distributed Arithmetic in High Dimension

In many computational areas such as in mechanical control, computations are not only limited to dot-product. Matrix operations are usually required, whose general form can be defined by equation 3.4.

$$
\begin{pmatrix} z_1 \\ z_2 \\ ... \\ z_s \end{pmatrix} = \begin{pmatrix} a_{11} & ... & a_{1r} \\ & ... & \\ a_{1s} & ... & a_{sr} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_r \end{pmatrix} \tag{3.4}
$$

This equation can be implemented using s DALUTs. The i-th ($i \in \{1, .., s\}$) DALUT is used to compute the vector product $z_i = \sum_{j=0}^{r} x_j \times a_{ij}$ with the constants a_{i1} to a_{ir}. If there is enough space on the chip to hold all the DALUTs, then equation 3.4 can be computed in few clocks. As we will see later, partial reconfiguration can also be used to implement many dot-products, if there is not enough memory on the chip to hold all the DALUTs.

3.2 Distributed Arithmetic and Real Numbers

The straightforward approach to handle real numbers in distributed arithmetic is the use of fixed-point. The representation of a real number using the fixed-point approach is done by defining an integer part and a fractional part

for each number. The integer part is separated from the fractional part using a point. Because the point is only an imaginary and not physically available in the number representation, operations on fixed-point numbers are not different than those on normal integer numbers. The datapath previously presented can be used without modifications for real numbers represented as fixed-point. Representing real numbers as fixed-point is advantageous in the computation speed and the amount of resources required to implement the datapath. However, the ranges of fixed-point representations as well as their precisions are small compared to those of a floating-point representations. Therefore, we would like to handle real numbers as floating-point.

In this section, we present a concept first investigated in [33] for handling real numbers, represented as floating-point in the IEEE 754 format, using distributed arithmetic. In IEEE 754 format, a number X is represented as follows:

$$X = (-1)^{S_X} 2^{e_X} \times 1.m_X \tag{3.5}$$

In this equation, e_X is the exponent (we consider that the substraction with the bias is done and the result is e_X), m_X is the mantissa and S_X is the sign of X. Without loss of generality, we will consider the sign to be part of the mantissa; thus, the new representation is $X = 2^{e_X} \times m_X$. With A, X and Z being all floating-point numbers, the floating-point counterpart of equation (3.1) is given by:

$$Z = X \times A = \sum_{i=0}^{n}(X_i \times A_i) = \sum_{i=0}^{n}(2^{e_{A_i}} \times m_{A_i}) \times (2^{e_{X_i}} \times m_{X_i})$$

$$= \sum_{i=0}^{n}(2^{e_{A_i}} \times 2^{e_{X_i}}) \times (m_{A_i} \times m_{X_i}) = \sum_{i=0}^{n}(2^{e_{A_i}+e_{X_i}}) \times (m_{A_i} \times m_{X_i})$$

$$\tag{3.6}$$

The goal here is to compute and provide Z as floating-point number, using the same principle of the partial sums previously described. At each step of the computation, a floating-point value F_i must be read from the floating-point DALUT and added to an accumulated sum, which is also a floating-point number. As the adder used at this stage is a floating-point adder, we assume that issues such as rounding and normalization have been integrated in its implementation. From the last part of equation 3.6, we see that the value $(2^{e_{A_i}+e_{X_i}}) \times (m_{A_i} \times m_{X_i})$ represents a floating-point number with exponent $e_{A_i} + e_{X_i}$ and mantissa $m_{A_i} \times m_{X_i}$. By setting $e_{F_i} = e_{A_i} + e_{X_i}$ and $m_{F_i} = m_{A_i} \times m_{X_i}$, we have the requested values at each computation step. Instead of computing the exponential part $(2^{e_{A_i}+e_{X_i}})$ of F_i as well as its mantissa $(m_{A_i} \times m_{X_{ij}})$ online, the idea here consists of using two floating-point

DALUTS for each constant A_i. The values $e_{A_i} + e_{X_i}$ will then be precomputed and saved in the first DALUT and $m_{A_i} \times m_{X_i}$ in the second one. Let us call the first DALUT that stores the exponents the EDALUT and the second DALUT that stores the mantissas the MDALUT. The size of EDALUT, $size(EDALUT)$, as well as that of MDALUT, $size(MDALUT)$, is given in equations (3.7).

$$size(EDALUT) = n \times 2^{|E|} \times |E| \text{ bits} \qquad (3.7)$$

$$size(MDALUT) = n \times 2^{|M|} \times |M| \text{ bits} \qquad (3.8)$$

$|E|$ is the exponent width and $|M|$ is the mantissa width of the floating-point representation.

The main argument against this approach is the size of the DALUTs used for this floating-point DA implementation, which can be so large that an implementation cannot be possible. However, if we consider a DA implementation involving five variables and five coefficients represented in the IEEE 754 floating-point format with 8 bits exponent and 10 bits mantissa, the total memory requirement for the EDALUTs and the MDALUTs is: $((5 \times 2^8 \times 8) + (5 \times 2^{10} \times 10))/1024 = 60$ Kbits. Therefore, the EDALUTS and the MDALUTs will easily fit into a very small low cost FPGA such as the Xilinx Spartan III 50 with a total of 72 Kbits Block RAM [225].

For large vectors with numbers represented on large mantissas and exponents, this approach will require a large amount of memory, certainly not directly available on the chip. An external memory must therefore be used.

Having the EDALUTs and the MDALUTs for each coefficient, the datapath will not be implemented as in the fixed-point or integer case. The variables are no more input in a bit-serial way. At step i, the variable X_i is used to address the two i-th EDALUT and MDALUT. The bits of e_{X_i} are used to access the EDALUT, while the bits of m_{X_i} are used to access the MDALUT in parallel. The values collected from the EDALUT and the MDALUT are used to build the floating point number F_i. After n steps, the floating-point dot-product is computed. Figure 9.10 shows the datapath for the floating-point DA. As all the DALUTs for the n coefficients are available on the device, they can be accessed in parallel to compute the dot-product in only one step.

Signal processing applications are usually based on the computation of the dot-product of one vector of variables with a vector of constants. They are ideal candidates for a DA implementation in reconfigurable hardware. As stated earlier in this section, to facilitate the design of DA applications, a tool was implemented in [33] for the design automation. Also, the possibility is given to
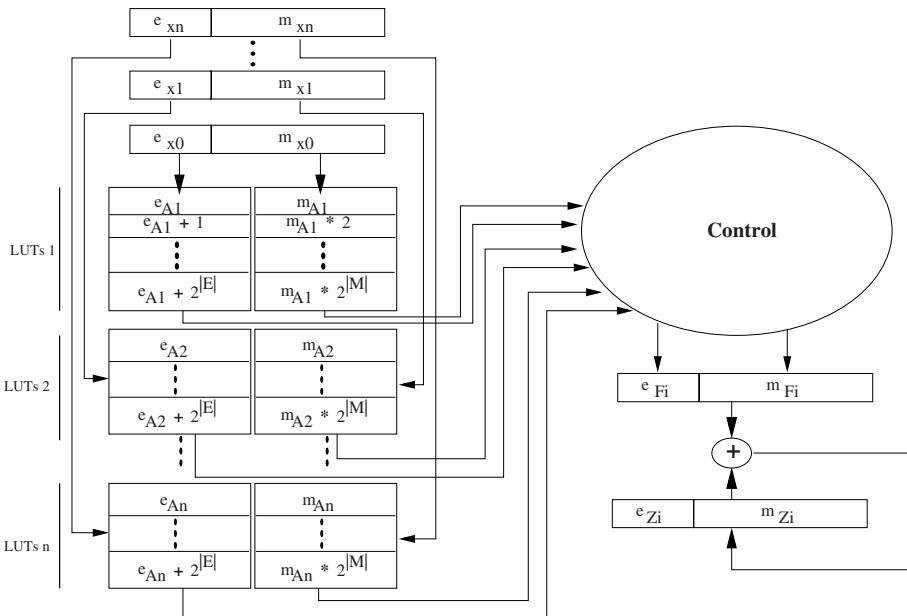
Figure 9.10. Datapath of the distributed arithmetic computation for floating-point numbers

investigate the different tradeoffs for a given application. For a given device and a given set of constants, different DA implementations can be generated, from the full parallel implementation (w-BAAT), which can be computed in only one step, to the full sequential DA (1-BAAT). For each possibility, the user is provided the area and speed of the design. Real numbers can be handled either as fixed-point or as floating-point in the IEEE 754 format with the technique previously defined. The width of the mantissa as well as that of the exponent has to be provided. For the final design, the tool generates a description in the Handel-C language.

3.3   Use of Reconfiguration

The modification of design implemented as distributed arithmetic can be done at run-time through the replacement of the DALUTs. The remaining part of the design must not be changed, because it is the same for all designs. In this case, the designer must not have to deal with all the cumbersome aspects of partial reconfiguration design. However, if the datawidth of the operators changes, then the computing structure (shifter, adder, accumulator) must be replaced as well. In this case, the partial reconfiguration of the device cannot be avoided. In Section 4, we present the use of reconfiguration in DA designs, using the concept of adaptive controller concept, which makes use of DA and partial reconfiguration.

3.4   Applications

In this section, we present one application, the implementation of recursive convolution algorithm for time domain simulation of optical multimode intrasystem interconnects that was substantially speeded-up through the use of distributed arithmetic. Another application that has benefit from the implementation as DA is the adaptive controller. Because Section 4 is devoted to adaptive controller, we explain only the first application here.

3.4.1   Recursive Convolution Algorithm of Time Domain Simulation of Optical Multimode Intrasystem Interconnects

In general, an optical intrasystem interconnect contains several receivers with optical inputs driven by transmitters with optical outputs. The interconnections of transmitter and receivers are made by a passive optical waveguide, which are represented as a multiport (figure 9.11) using ray tracing approach.

The transfer of an optical signal along the waveguide can be computed by a multiple convolution process. Frequency domain simulation methods are not applicable regarding the high number of frequency. Pure time domain simulation methods are more efficient if the pulse responses can be represented by exponential functions. The application of a recursive method for the convolution of the optical stimulus signals at the input ports with the corresponding pulse responses enables a time efficient computation of the optical response signals at the belonging output ports. The recursive formula to be implemented in three different intervals is given by equation 3.9.

$$y(t_n) = f_0 \cdot y(t_{n-1}) + \qquad\qquad (3.9)$$
$$f_4 \cdot x_0 - f_5 \cdot x_1 + f_{24} \cdot x_2 + f_{53} \cdot x_3 .$$

Because the values f_0, f_4, f_4, f_{24} and f_{53} are constants, while t_{n-1}, x_0, x_1, x_2, x_3 are variables, equation 3.9 is best adapted for a DA-implementation.
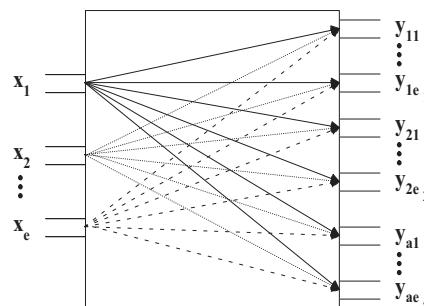
Figure 9.11.   An optical multimode waveguide is represented by a multiport with several transfer paths

Workstation	1 interval	3 intervals
Sun Ultra 10	73.8 ms	354.2 ms
Athlon (1.53 GHZ)	558 ms	1967.4 ms
FPGA (time)	**1 interval**	**3 intervals**
Custom dot-product design	25.6 ms	76.8 ms
Sequential DA	19.4 ms	19.4 ms
3-parallel DA	6.4 ms	6.4 ms
FPGA (area)	**1 interval**	**3 intervals**
Custom dot-product design	could not fit	could not fit
Sequential DA	yes (7%)	yes (14%)
3-parallel DA	yes (14%)	yes (42%)

Table 9.1.   Results of the recursive convolution equation on different platforms

For this equation, different tradeoffs were investigated in the framework presented in [33], for generation and evaluation of DA-trade-off implementation. A Handel-C code was generated, and the complete design was implemented on a system made upon the Celoxica RC1000-PP board equipped with a Xilinx Virtex 2000E FPGA and plugged into a workstation.

Table 9.1 provides the implementation results on different platforms. The workstation is used for sending the variable to the FPGA and collecting the result of the computation. The implementation of equation 3.9 in three intervals occupies about 14% of the FPGA area while running at 65 MHZ. To obtain a maximal parallelism in the computation, 6 DALUT could be implemented in parallel leading to 6 times speed-up. As can be seen on figure 9.12,
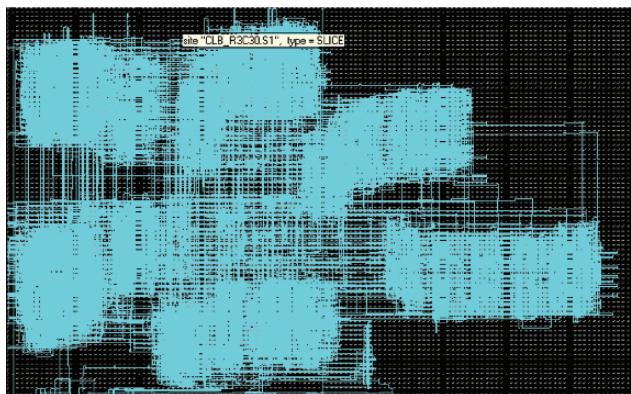
Figure 9.12.   Screenshot of the 6-parallel DA implementation of the recursive convolution equation on the Celoxica RC100-PP platform

the 6-parallel DALUT and the corresponding adder three occupy about 76% of the FPGA area. Enough space is left for the routing.

The same design was implemented without the use of DA. It could not fit into the same FPGA and the run-time was much bigger that of the DA-implementation. The performance as well as the area consumption of our the DA-implementation is more efficient than that of all the other architectures.

4.   Adaptive Controller

In this section, we investigate the next field of application of reconfigurable devices, namely the implementation of adaptive controllers, also identified here as multi-controller, using partial reconfiguration.

We will first introduce the multi-controller concept. Thereafter, we investigate its implementation using the distributed arithmetic. Finally, the implementation of the whole design using partial reconfiguration is shown.

4.1   Adaptive Control

The task of a controller is to influence the dynamic behaviour of a system referred to as *plant*. A control feedback is available, if the input values for the plant are calculated on basis of the plant's outputs.

Adaptive controllers deal with the control of a plant, for instance a complex mechatronic system, in a changing environment. In this case, the control is modeled as a physical process that operates in a limited set of operating regimes. Depending on the environmental conditions, the operation regime of the plant must be changed to adapt to new conditions. With conventional methods, it might be possible to design one robust controller that optimally controls the plant in all operating regimes. According to the conditions, the control regime may be changed using a set of parameters. Besides the possible slow response of such a controller [166], the size can be too big due the large amount of possible cases to be considered. To overcome this handicap, the controller modules can be implemented as separate modules and stored in a database. They will be used at run-time to replace the active controller, whenever a new control algorithm must be implemented. The price to pay for this solution is the amount of memory to store all possible module implementations as well as the time needed to switch from one module to the next one.

The general architecture of an adaptive controller as described in [161] is shown in figure 9.13. It consists of a set of *controller modules (CM)*, each of which is optimized for an operating regime of the plant. A *supervisor* component is used to switch from module to module and set the active one.[4] The decision to switch from one *CM* to the next one is made on the basis of mea-

---

[4]At a given time, the active controller is the one that controls the plant
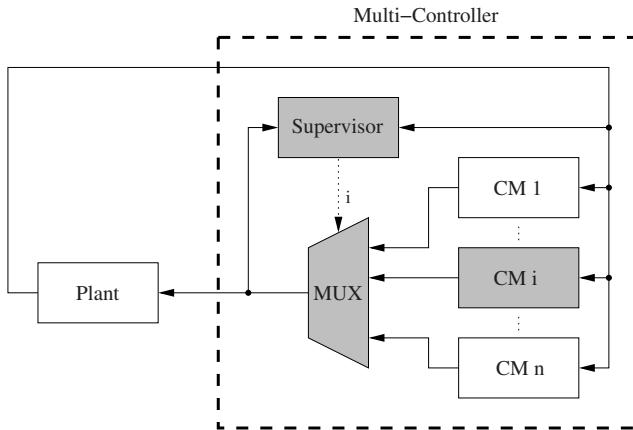
Figure 9.13.   Adaptive controller architecture

surements of physical values of the plant. The strategy of the *supervisor* can vary from simple Boolean combinations of the input values to very complex reasoning techniques [206].

The computation of the plant inputs is normally done in a time discrete manner, using a set of discrete sample points $t_k \ldots t_{k+1}$. The time between two sampling points is constant and defines the sampling period T. The controller application can therefore be seen as a periodic real-time task, which must be performed at every sampling point and must be completed within one period T.

While in a Von Neumann processor-based implementation of the system, each module is implemented as sequence of instructions, in a reconfigurable device, the tasks are implemented as hardware modules to be downloaded onto the device at run-time. Let us take a look on the behaviour of the single modules and their implementations.

4.2   Controller-Module Specification

A common approach is to model the plant as a linear time-invariant system. Based on this model and the requirements of the desired system behaviour, a linear controller is systematically derived using formal design methods.

Let us assume that every controller module CM_i of figure 9.13 is designed as a linear time-invariant system, optimized for a given operating regime of the plant. After a time discretization has been performed, the behaviour can be captured in equation 4.1.

The input vector of the controller module, representing the measurements from sensors of the plant, is represented by $\mathbf{u}$, $\mathbf{y}$ is the output vector of the controller, which is used to drive the actuators of the plant, and $\mathbf{x}$ is the inner

state vector of the controller. The matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and $\mathbf{D}$ are used for the calculation of the outputs based on the inputs.

$$
\begin{aligned}
\mathbf{x}(k+1) &= \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \\
\mathbf{y}(k) &= \mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k)
\end{aligned}
\tag{4.1}
$$

$$
p = dim(\mathbf{u}), \qquad s = dim(\mathbf{x}), \qquad q = dim(\mathbf{y})
$$

The task of the digital controller is to calculate equation 4.1 during one sampling interval. The new state $\mathbf{x}_{k+1}$ and the output $\mathbf{y}_k$ must be computed before the next sampling point $k + 1$.

The state space equations of a digital linear time invariant controller (equation 4.1) can be written as a product of a matrix of constant coefficients and a vector of variables (equations 4.2 and 4.3).

$$
\mathbf{z}_{(s+q,1)} = \mathbf{M}_{(s+q,s+p)}\mathbf{V}_{(s+p,1)}
\tag{4.2}
$$

$$
\begin{bmatrix}
x_1(k+1) \\
\vdots \\
\vdots \\
x_s(k+1) \\
y_1(k) \\
\vdots \\
\vdots \\
y_q(k)
\end{bmatrix}
=
\begin{bmatrix}
a_{11} & \cdots & a_{1s} & b_{11} & \cdots & b_{1p} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
a_{s1} & \cdots & a_{ss} & b_{s1} & \cdots & b_{sp} \\
c_{11} & \cdots & c_{1n} & d_{11} & \cdots & d_{1p} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
c_{q1} & \cdots & c_{qs} & d_{q1} & \cdots & d_{qp}
\end{bmatrix}
\begin{bmatrix}
x_1(k) \\
\vdots \\
x_s(k) \\
u_1(k) \\
\vdots \\
u_q(k)
\end{bmatrix}
\tag{4.3}
$$

$$
\underbrace{}_{\mathbf{z}} \qquad\qquad \underbrace{}_{\mathbf{M}} \qquad\qquad \underbrace{}_{\mathbf{v}}
$$

Equations 4.2 and 4.3 define a classical distributed arithmetic equation in higher dimension. Each of the $s + q$ entries of the resulting vector $\mathbf{v}$ defines a separate DA equation. We therefore need to compute $s + q$ different DA-equations, all independent from each other. The computation of the $s + q$ equations can then be performed in parallel, provided that enough resources are available on the chip to store all the DALUTs. If enough memory is not available for storing all the DALUTs as well as the corresponding logic for the datapath, then the different equations must be implemented sequentially. In general, compromises must be found between the amount of parallelism that one need to have and the speed of the design.

4.3   Use of Reconfiguration

As we explained earlier, the controller modules can be implemented as hardware modules and stored in a database from which they will be downloaded at run-time by the supervisor module.

The straightforward approach to realize this is to build a structure consisting of a fixed part in which the supervisor resides. A *reconfigurable slot* is then foreseen as place holder for the controller module that will be downloaded at run-time (figure 9.14 left). Each controller module is downloaded at run-time
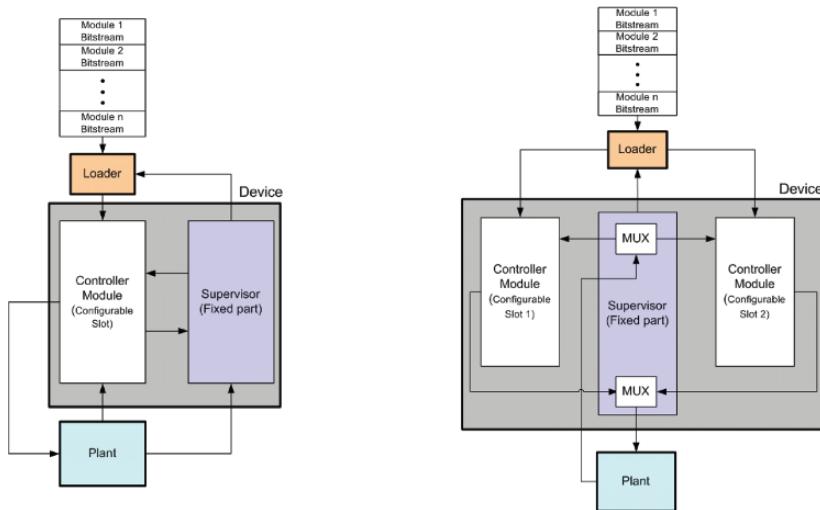
Figure 9.14. Adaptive controller architecture. Left: the one slot implementation, and right: the two slot implemenation

into the reconfigurable slot, which has predefined interfaces to the supervisor and to the plant.

The problem with this approach is the vacuum, which arises on switching from one module to the next one. Because only one slot is available, the reconfiguration of this slot will place the plant in a 'floating state' on reconfiguration, where it is not really controlled. To avoid this situation, the approach developed in [64] was the use of two slots. One active slot and a passive one (figure 9.14 right). The active slot is in charge of controlling the plant, while the reconfiguration takes place in the passive slot. Whenever the supervisor decides to replace a controller module, the reconfiguration will first take place on the passive slot. Thereafter, the control of the plant will be given to the configured module, which becomes the active one, while the former active becomes passive.

5.   Adaptive Cryptographic Systems

Sensitive data exchange is always coupled with security issues. Cryptography provides a means to allow two communicating entities to exchanged data in such a way that a third party that intercepts the data will not be able to understand their meaning. A protocol is therefore required, which allows the sender to crypt the data in such a way that only the receiver will be able decrypt them.

Considerable advances have been done in communication in the past, and this trend is likely to continue. The development of the internet has also pushed the development in several branches where a network processing could not be

tough off few years ago. Applications like e-commerce, e-government, virtual private network, on-line banking must provide a high degree of security.

Over years, a large variety of standards, such as Triple-DES, Advanced Encryption Standard (AES), Data Encryption Standard (DES), RSA, OpenPGP, CipherSaber, IPsec, Blowfish, ANSI X9.59, CAST, RC4 and RC6, have been developed to provide high security. With this large variety of standards and the customized implementation possibilities for each standard, cryptography can be seen as one of the most versatile application domains of computer science. Depending on criteria such as speed, degree of flexibility and degree of security, single implementations of cryptography application were developed either as software or as intellectual property component.
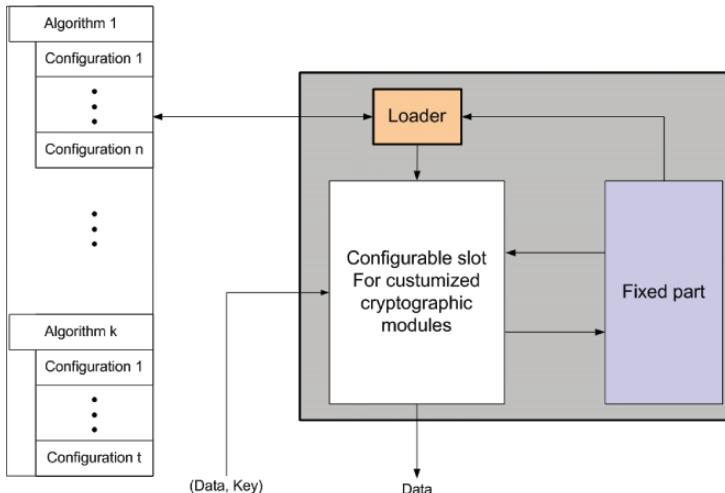
In [221] the advantages of using reconfigurable hardware in cryptography are listed. The author focus on the traditional advantage of flexibility and performance. The flexibility is so far important because it offers the possibility to use the same hardware to switch from one algorithm to the next one at run-time, according to factors such as the degree of security, the computational speed, the power consumption. Also, according to some parameters, a given algorithm can be tuned. Moreover, algorithms that has been broken and where the security is no more insured can be changed by means of reconfiguration. The system can easily be upgraded to include new standards, developed while the system was already deployed. The corresponding algorithm can therefore be compiled and included in the library of bitstreams for the device configuration. The second advantage provided by reconfigurable hardware, namely the performance, can be used to efficiently implement the components, by using the inherent parallelism and building efficient operators for computing Boolean operation on a very large amount of data. This results on a large throughput and a cost efficiency. Experiments reported a throughput of 12 GBit/s for an implementation of the block cipher AES on an FPGA Virtex 1000, using 12,600 slices and 80 RAMs [90], while an ASIC implementation, the Amphion CS5240TK [150] clocked at 200 MHz, could provided twice the throughput of the FPGA solution. The same algorithm, implemented on a DSP TI TMS320C6201 provided a throughput of 112.3 Mbits/s [222], while a throughput of 718.4 Mbit/s could be reached on a counterpart implementation on a on a Pentium III [10].

The general architecture of an adaptive cryptographic engine proposed by Prasanna and Dandalis [61] [178] basically consists of a database to hold the different configuration that can be downloaded at run-time onto the device, like an FPGA for instance, to perform the computation and a configuration controller to perform the reconfiguration, i.e. downloading the corresponding bitstream form the database into the FPGA. Each bitstream represents a given algorithm implementing a given standard and tuned with some parameters according to the current user's need.

With the recent development in FPGA, it is possible to have a complete system on programmable chip (processor, peripherals, custom hardware components, interconnection) implemented on an FPGA. The configuration controller therefore must no more resides off chip. It can be implemented as custom on-chip hardware module or as software running on an embedded processor. Also the possibility to reconfigure only part of the chip opens new possibilities. In the architecture presented in [61] [178], the whole device must be reset on re-configuration, thus increasing the power consumption because of the amount of the data that must be downloaded on a chip. Power consumption is usually a big problem in mobile environments, and the implementation must consider such issues. Besides this power saving, partial reconfiguration also provides the possibility of keeping a skeleton structure into the device and perform only few modifications at run-time on the basic structure to move from one algorithm to the next one. In [42], a cryptographic application is implemented as exchangeable module of a partial reconfigurable platform. The system, which is based on the AES algorithm consumes only 954 Virtex slices and 3 block RAMS. The cryptographic algorithm is used in this case just as a block to test the partial reconfigurable platform, instead of using the partial reconfiguration to enhance the flexibility of the cryptographic algorithm.

Figures 9.15 presents a possible architecture of an adaptive cryptography system, using the previous mentioned advantages of partial reconfigurable devices.

The main difference with the architecture presented in [61] is the use of partial reconfiguration, which allows for an integration of all components on a

Figure 9.15. Architecture of an adaptive cryptographic system

single chip. Also, in contrast to the adaptive architecture for a control system presented in figure 9.14, the loader module resides into the device. Depending on the implementation chosen, the loader can reside inside or outside the device. If a Xillinx Virtex chip is used and the configuration takes place via the ICAP port, then the loader, which is the *ICAP* module in this case, is automatically available inside the device. However, if the configuration happens through the normal *SelectMap* port, then we need an external loader module for collecting configuration data from the database and copy them on the configuration port.

In figure 9.15, the architecture is logically divided into two main blocks. A fix one, which remains continuously on the chip. It consist of the parts, which are common to all the cryptographic algorithms in general or common to the algorithms in a given class only. On the figure, we show only one reconfigurable slot; however, it can be implemented as set of configurable blocks, each of which can be changed by means of reconfiguration to realize a given customized standard.

The last point concerns the development of building blocks that will be compiled in bitstreams to be downloaded into the device at run-time. A designer is no more required to focus on the hardware implementation of the cryptographic algorithm. A lot of work was done in this direction, and the results are available. We need mostly to focus on the architecture of the overall system and find out how a viable partitioning can be done, according to the reconfiguration scheme.

Most of the work have focussed in various implementations of a given approach like the RSA [185] in [155] [88] [154] [232] [50] or the implementations described in [78] [19] [20] [47] [145] [152] [15], mostly parameterizable and based on the Elliptic Curve approach [138] [157]. Generators for producing a customized description in a hardware description language have been developed for example in [47]. This can be used to generate various configurations that will be used at run-time to move from one implementation to the next one.

6. Software Defined Radio

Software defined radio (SDR) was defined in 1991 by Joe Mitola [129] as follow:

A software radio is a radio whose channel modulation waveforms are defined in software. That is, waveforms are generated as sampled digital signals, converted from digital to analog via a wideband DAC and then possibly upconverted from IF to RF. The receiver, similarly, employs a wideband Analog to Digital Converter (ADC) that captures all of the channels of the software radio node. The receiver then extracts, downconverts and demodulates the channel waveform using software on a general purpose processor.

As it is the case in many new technologies, SDR was initiated by the military in an attempt to alleviate numerous problems associated with traditional radio systems [162]. In the project *SPEAKeasy* [162], which can be seen as precursor of the SDR technology, several military entities (DARPA, AIR FORCE/AFRL, ARMY/CECOM, NAVY/NRaD/SPAWAR, NSA) joined efforts in the development of 'programmable' radios to demonstrate the feasibility of a Multiband, Multimode Radio (MBMMR) and to develop technologies that facilitate programmability and implementation of MBMMR. The achievement was a system with two programmable channels, the Texas Instrument quad-TMS320C40 multi-chip module for digital signal processing, and a SUN Sparc 10 workstation for interfacing with human.

The main goal in SDR is to move the analog–digital signal conversion as closer as possible to the antenna and to process the digital signals in software rather than using a set of dedicated hardware components. The expected benefits of using SDR have drained great attention and interest from governmental entities, from the academia and from the industry.

SDR is the next evolutionary stage of wireless technology. Many believe that it is just a question of time up to the adoption of SDR in the broad range of communication facilities, and several initiatives like [121] [189] were born with the goal of accelerating the proliferation of SDR.

The advantages of SDR can be categorized as follows:

- **Interoperability** to provide support of multiple standards through multimode, multiband radio capabilities

- **Flexibility** for an efficient shift of technology and resources

- **Adaptivity** to provide a faster migration towards new standards and technologies through programmability and reconfiguration

- **Sustainability** for increasing resource utilization through generic hardware platforms

- **Reduced ownership costs** because of the reduced infrastructure, the lower maintenance and the easier deployment

The general architecture of a SDR is presented in figure 9.16.

At least one digital to analog (DAC) and one analog to digital converter (ADC) must ba available in a SDR system. While the ADC performs conversion of the incoming analog signals to digital signals, the DAC performs the inverse operation.

The signal processing part collects digital signals from the ADC or from another source like a base station input, process them and send them to the DAC.
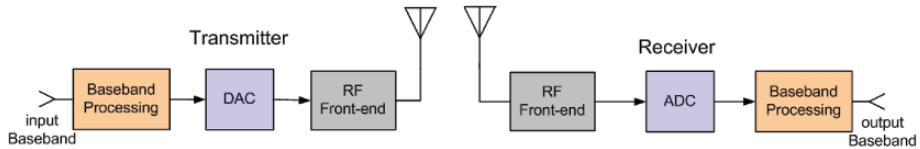
Figure 9.16. Architecture of a software defined radio system

The ideal realization of a SDR would push the ADC and DAC as close as possible to the antenna. The practical realization is, however, as explained in [217], very challenging.

6.1 Use of Reconfiguration

Reconfigurable devices provide the best prerequisites to fulfill the expectations (interoperability, flexibility, adaptability, sustainability, and reduced ownership costs) placed on SDR systems. As explained in the previous section, it is possible to realize a complete system consisting of a processor, peripherals and reconfigurable custom hardware modules as system on programmable chip. In such a system, the maximal flexibility will be insured by the processor, while lower degree of flexibility can be realized by means of reconfiguration. For many algorithm in signal processing, software alone will not be able to provide the required performance. A migration of functions in hardware is then required. The simple approach as realized in [9] [122] [48] is to provide a single coprocessor for computation intensive tasks, like the CORDIC [209] implementation, directly attached to the processor. This solution provides the possibility to configure the system as whole, either to implement a new functions at run-time or event to upgrade the system structure according to the availabilities of new standards. The use of partial reconfiguration [69] will be welcomed, to allow the upgrading to take place without interrupting the system, while saving a considerable amount of power.

7. High-Performance Computing

One of the first and most investigated fields of application of FPGAs in the 1990s was high-performance computing. The goal of many platform built at that time was to provide acceleration of application such as searching in genomic databases, signal and image processing, matrix factorization. Several architectures were developed, mostly consisting of a workstation on which one or more FPGA were attached. By reconfiguration of the FPGAs, custom instructions could be added to the computing system, sometimes even at runtime. While remarkable speed-ups were reported for some applications, the resulting custom computing machines (CCMs) could not be widely adopted as viable computing platforms. This was mainly due to architectural limitations

and lack of high-level programming tools. The capacity of FPGAs was too small to allow real migration of software components in hardware. Furthermore, the performance bottleneck posed by the slow communication between host processor and FPGAs often busted speed-ups. Finally, there were no useful software-like description languages that could encourage software designers to write programs for such machines and implement efficient libraries for reusable data stream management. Programming CCMs meant tedious low-level hardware design, which was not attractive at all.

The last couple of years brought a pivotal change. FPGA's logic densities increased dramatically, now allowing for massive bit-level parallelism. Available high-level languages like Handel-C [125] and ImpulseC [175] as well as progress in compilation technology makes the logic resources available to high-level application programmers.

Boosted by the recent engagement of companies such as Cray [58], SRC Computers [59], and ClearSpeed [51], Nallatech [165], and SGI [191], CCMs are experiencing a renaissance. Despite the use of new high-speed interfaces and efficient datastream management protocols between the components in the new systems, the new architectures are built on the same models like the old ones in the 1990s. It usually consist of a set of FPGAs, on the same board with one or more processors. The processors control the whole systems and configured the FPGAs at run-time with dedicated function to be accelerated in hardware.

While considerable progress has been done, the rentability of the machines provided by the firms previously mentioned still have to be proven. The rentability cannot be evaluated anymore only on the basis of speed-ups observed in some class of applications. It is not sure that a comparison made in terms of performance/$, performance/J, and performance/sqm will favour further deployment of those systems. Craven and Athanas recently provided in [57] a study on the viability of the FPGA in supercomputers. Their study is based on the cost of realizing floating-point into FPGAs. The cost includes not only the performance gain but also the price to pay. The authors came to the conclusion that the price to pay for a FPGA-based supercomputer is too high for the marginal speed-ups.

High-precision floating-point arithmetic is usually the bulk of computation in high-performance computing. Despite the progress done in developing floating-point units in hardware, Floating-point computation and FPGA is still a difficult union. The advantage provided by FPGA in customized implementation of floating-point as described Shirazi [192] is unlikely to help here because the datapath must provide customization that consume the largest amount of resources. Coarse-grained elements like the embedded multipliers in FPGAs or coarse-grained reconfigurable device provide the best prerequisites for the use of reconfigurable device in supercomputers.

8.   Conclusion

We have presented in this chapter a couple of applications that can take advantage of the flexibility as well as performance of reconfigurable devices. For each application targeted, we placed the focus mostly on the architectural organization because the goal was not to necessary present the detailed implementation of an application. However, we have presented a comprehensive description of the pattern-matching application, while keeping the other presentation short. This is due to the low attention paid on pattern matching in hardware in available textbooks. With the large amount of literature in other presented topics such as image processing and control, we choose not to focus in details of those applications.

A lot of work have been done in the last two decades, and a large number of applications were implemented in FPGA, which is the main representative of reconfigurable devices. We could not present nor cite all available implementations here. We have rather focussed on few ones where we could show the usefulness of reconfigurability. Finally, we would like to emphasize that despite two decades of research, the existence of 'killer applications' could not be shown for reconfigurable device, thus limiting the acceptance of reconfiguration in the industry. The existence of a killer application will certainly boost the development of reconfigurable devices, leading to new class of devices, tools and programmers. Despite this missing step, research keeps going, and the support of the industry is needed more than it have ever be.

References

[1] G. R. AB, *MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 8.2i*, 2005, http://www.gaisler.com. [Online]. Available: http://www.gaisler.com

[2] A. Ahmadinia, C. Bobda, S. Fekete, J. Teich, and J. van der Veen, "Optimal routing-conscious dynamic placement for reconfigurable computing," in *14th International Conference on Field-Programmable Logic and Application*, ser. Lecture Notes in Computer Science, vol. 3203.   Springer-Verlag, 2004, pp. 847–851, available at http://arxiv.org/abs/cs.DS/0406035.

[3] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, "A new approach for on-line placement on reconfigurable devices," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS) / Reconfigurable Architectures Workshop (RAW)*, 2004.

[4] A. Ahmadinia, C. Bobda, J. Ding, M. Majer, J. Teich, S. P. Fekete, and J. C. van der Veen, "A practical approach for circuit routing on dynamic reconfigurable devices," in *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*.   Washington, DC, USA: IEEE Computer Society, 2005, pp. 84–90.

[5] C. Alpert and A. Kahng, "Geometric embeddings for faster and better multi-way netlist partitioning," 1993.

[6] C. J. Alpert and A. B. Kahng, "Multi-way partitioning via spacefilling curves and dynamic programming," in *Design Automation Conference*, 1994, pp. 652–657.

[7] C. J. Alpert and S.-Z. Yao, "Spectral partitioning: The more eigenvectors, the better," in *Design Automation Conference*, 1995, pp. 195–200.

[8] *ADM-XRC-II Xilinx Virtex-II PMC*, Alpha Data Ltd., 2002, http://www.alpha-data.com/adm-xrc-ii.html.

[9] Altera, "Software defined radio," www.altera.com/literature/cp/fpga-cores-for-sdr.pdf. [Online]. Available: www.altera.com/literature/cp/fpga-cores-for-sdr.pdf

[10] K. Aoki and H. Lipmaa, "Fast Implementations of AES Candidates," in *The Third Advanced Encryption Standard Candidate Conference*.   New York, NY,

USA: National Institute of Standards and Technology, 13–14 Apr. 2000, pp. 106–120, entire proceedings available from the conference homepage `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3conf.htm`. [Online]. Available: citeseer.ist.psu.edu/aoki00fast.html

[11] P. Athanas, J. Becker, G. Brebner, and H. ElGindy, "Dynamically reconfigurable architectures," in *Dagstuhl Semiar 03301*, 2003.

[12] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11–18, 1993, citeseer.nj.nec.com/athanas93processor.html. [Online]. Available: citeseer.nj.nec.com/athanas93processor.html

[13] Atmel, *AT6000 FPGA configuration guide*.   Atmel Inc.

[14] R. Baeza-Tates, *Handbook of Algorithms and Data Structures*.   Addison-Wessley, 1991.

[15] S. Bajracharya, C. Shu, K. Gaj, and T. El-Ghazawi, "Implementation of elliptic curve cryptosystems over gf(2n) in optimal normal basis on a reconfigurable computer," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*.   New York, NY, USA: ACM Press, 2004, pp. 259–259.

[16] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A scalable architecture based on single-chip multiprocessing," 2000. [Online]. Available: http://citeseer.ist.psu.edu/barroso00piranha.html

[17] V. Baumgarte, G. Ehlers, F. May, A. Nueckel, M. Vorbach, and M. Weinhardt, "PACT XPP A self-reconfigurable data processing architecture," *J. Supercomput.*, vol. 26, no. 2, pp. 167–184, 2003.

[18] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *In IEEE Design and Test - Special Issue on Reconfigurable Computing*, vol. January-March, pp. 68–83, 2000.

[19] M. Bednara, M. Daldrup, J. Shokrollahi, J. Teich, and J. von zur Gathen, "Tradeoff analysis of fpga based elliptic curve cryptography," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS-02)*, Scottsdale, Arizona, U.S.A, May 2002.

[20] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, and J. Teich, "Reconfigurable implementation of elliptic curve crypto algorithms." in *IPDPS*, 2002.

[21] L. Benini and G. Micheli, "Network on chips: A new soc paradigm," *IEEE Computer*, January 2001.

[22] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[23] J. L. Bentley, "Solutions to Klee's rectangle problems," Carnegie-Mellon Univ., Pittsburgh, PA, Technical Report, 1977.

[24] J. L. Bentley and D. Wood, "An optimal worst case algorithm for reporting intersections of rectangles," *IEEE Trans. Comput.*, vol. C-29, pp. 571–577, 1980.

[25] P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to programmable active memories," in *Systolic Array Processors*, J. McCanny, J. McWhirter, and E. Swartzlander, Eds. Prentice Hall, 1989. [Online]. Available: citeseer.ist.psu.edu/bertin89introduction.html

[26] N. B. Bhat and D. D. Hill, "Routable technologie mapping for lut fpgas," in *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*. Washington, DC, USA: IEEE Computer Society, 1992, pp. 95–98.

[27] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Linear time bounds for median computations," in *Proc. Fourth Annual ACM Symposium on Theory of Computing*, 1972, pp. 119–124.

[28] C. Bobda, "Temporal partitioning and sequencing of dataflow graphs on reconfigurable systems," in *International IFIP TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002)*. Montreal, Canada: IFIP, 2002, pp. 185–194.

[29] C. Bobda and N. Steenbock, "A rapid prototyping environment for distributed reconfigurable systems," in *13th IEEE International Workshop On Rapid System Prototyping(RSP'02), Darmstadt Germany*. IEEE Computer Society, 2002.

[30] C. Bobda, "Synthesis of dataflow graphs for reconfigurable systems using temporal partitioning and temporal placement," Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Entwurf Paralleler Systeme, 2003, p. 35,-, ISBN 3-935433-37-9.

[31] C. Bobda, "Coremap: a rapid prototyping environment for distributed reconfigurable systems," *International Journal of Embedded Systems*, vol. 2, no. 3-4, pp. 274 – 290, 2006.

[32] C. Bobda and A. Ahmadinia, "Dynamic interconnection of reconfigurable modules on reconfigurable devices." *IEEE Design & Test of Computers*, vol. 22, no. 5, pp. 443–451, 2005.

[33] C. Bobda, A. Ahmadinia, and J. Teich, "Generation of distributed arithmetic designs for reconfigurable application." in *ARCS Workshops*, 2004, pp. 205–214.

[34] C. Bobda and T. Lehmann, "Efficient building of word recongnizer in fpgas for term-document matrices construction," in *Field Programmable Logic and Aplications FPL 2000*, R. Hartenstein and H. Grünbacher, Eds. Villach, Austria: Springer, 2000, pp. 759–768.

[35] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, and J. Teich, "A dynamic noc approach for communication in reconfigurable devices," in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL)*, ser. Lecture Notes in Computer Science (LNCS), vol. 3203. Antwerp, Belgium: Springer, Aug. 2004, pp. 1032–1036.

[36] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: A reinforcement learning approach," in *Advances in Neural Information*

Processing Systems, J. D. Cowan, G. Tesauro, and J. Alspector, Eds., vol. 6. Morgan Kaufmann Publishers, Inc., 1994, pp. 671–678. [Online]. Available: citeseer.ist.psu.edu/boyan94packet.html

[37] G. J. Brebner, "A virtual hardware operating system for the xilinx xc6200," in *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*. London, UK: Springer-Verlag, 1996, pp. 327–336.

[38] R. P. Brent and F. T. Luk, "The solution of singular-value and eigen-value problems on multiprocessor arrays," *SIAM J. Sci. Stat. Comput.*, vol. 6, no. 1, pp. 69–84, 1985.

[39] D. A. Buel, J. M. Arnold, and W. J. Kleinfelder, *Splash 2 FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, Los Alamitos, California, 1996.

[40] C. Bobda, M. Huebner, A. Niyonkuru, B. Blodget, M. Majer, A. Ahmadinia, "Designing partial and dynamically reconfigurable applications on xilinx virtex-ii fpgas using handelc," in *International Workshop on Field Programmable Logic and Applications (FPL), Antwerp, Belgium*.

[41] J. M. P. Cardoso and H. C. Neto, "An enhance static-list scheduling algorithm for temporal partitioning onto rpus," in *IFIP TC10 WG10.5 10 Int. Conf. on Very Large Scale Integration(VLSI'99)*. Lisboa, Portugal: IFIP, 1999, pp. 485 – 496.

[42] J. Castillo, P. Huerta, V. López, and J. I. Martínez, "A secure self-reconfiguring architecture based on open-source hardware," *reconfig*, vol. 0, p. 10, 2005.

[43] *RC2000 Development Board*, Celoxica Ltd., 2004, http://www.celoxica.com/products/boards/rc2000.asp.

[44] P. K. Chan, M. D. F. Schlag, and J. Y. Zien, "Spectral-based multi-way fpga partitioning," in *FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM Press, 1995, pp. 133–139.

[45] D. Chang and M. Marek-Sadowska, "Partitioning sequential circuits on dynamicaly reconfigurable fpgas," in *International Symposium on Field Programmable Gate Arrays(FPGA 98)*. Monterey, California: ACM/SIGDA, 1998, pp. 161 – 167.

[46] K.-C. Chen, J. Cong, Y. Ding, A. B. Kahng, and P. Trajmar, "Dag-map: Graph-based fpga technology mapping for delay optimization," *IEEE Design and Test of Computers*, vol. 09, no. 3, pp. 7–20, 1992.

[47] R. C. C. Cheung, N. J. Telle, W. Luk, and P. Y. K. Cheung, "Customizable elliptic curve cryptosystems." *IEEE Trans. VLSI Syst.*, vol. 13, no. 9, pp. 1048–1059, 2005.

[48] F. Christensen, "A scalable software-defined radio development system," *Xcell Journal*, Oct. 2004.

[49] K. S. Christos H. Papadimitrio, *Combinatorial Optimization*. Englewood Cliffs, NY: Prentice Hall, 1982.

[50] M. Ciet, M. Neve, E. Peeters, and J.-J. Quisquater, "Parallel fpga implementation of rsa with residue number systems - can side-channel threats be avoided? - extended

version," Cryptology ePrint Archive, Report 2004/187, 2004. [Online]. Available: http://eprint.iacr.org

[51] p. ClearSpeed Technology, *http://www.clearspeed.com/*.

[52] W. Cockshott and P. Foulk, "A low-cost text retrieval machine," *IEEE PROCEEDINGS*, vol. 136, no. 4, pp. 271–276, July 1989.

[53] J. Cong and Y. Ding, "Flowmap: an optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[54] J. Cong and Y. Ding, "Combinational logic synthesis for lut based field programmable gate arrays," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, no. 2, pp. 145–204, 1996.

[55] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[56] X. Corp, `http://www.xess.com`.

[57] S. Craven and P. Athanas, "Examining the viability of fpga supercomputing," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. Article ID 93 652, 8 pages, 2007, doi:10.1155/2007/93652.

[58] Cray, "Cray xd1 supercomputer," GNU's home page. [Online]. Available: http://www.cray.com/products/xd1/

[59] I. CRC Computers, "General purpose reconfigurable computing systemsr." [Online]. Available: http://www.srccomp.com/

[60] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the Design Automation Conference*, Las Vegas, NV, Jun. 2001, pp. 684–689.

[61] A. Dandalis and V. K. Prasanna, "An adaptive cryptographic engine for internet protocol security architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 3, pp. 333–353, 2004.

[62] K. Danne, "Distributed arithmetic FPGA design with online scalable size and performance," in *Proceedings of 17th SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI04)*. ACM Press, New York, NY, USA, 7 - 11 Sep. 2004, pp. 135–140.

[63] K. Danne and C. Bobda, "Dynamic reconfiguration of distributed arithmetic controllers: Design space exploration and trade-off analysis," *ipdps*, vol. 04, p. 140a, 2004.

[64] K. Danne, C. Bobda, and H. Kalte, "Run-time exchange of mechatronic controllers using partial hardware reconfiguration," in *Proc. of the International Conference on Field Programmable Logic and Applications (FPL2003), Lisbon, Portugal*, Sep. 2003.

[65] S. Davidson, D. Landskov, B. Shriver, and P. W. Mallett, "Some experiments in local microcode compaction for horizontal machines." *IEEE Trans. Computers*, vol. 30, no. 7, pp. 460–477, 1981.

[66] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd ed.   Berlin, Germany: Springer-Verlag, 2000.

[67] S. Deerwester, S. Dumai, G. Furnas, T. Landauer, and R. Harshmann, "Indexing by latent semantic analysis," *Journal of American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[68] A. DeHon, "DPGA-coupled microprocessors: Commodity ICs for the early 21st century," in *IEEE Workshop on FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek, Eds.   Los Alamitos, CA: IEEE Computer Society Press, 1994, pp. 31–39. [Online]. Available: citeseer.ist.psu.edu/dehon94dpgacoupled.html

[69] J. P. Delahaye, G. Gogniat, C. Roland, and P. Bomel, "Software radio and dynamic reconfiguration on a dsp/fpga platform," 2004. [Online]. Available: http://hal.ccsd.cnrs.fr/ccsd-00089395/en/

[70] Digilent Inc., "The vdec1 video decoder board." [Online]. Available: http://www.digilentinc.com/Products/Detail.cfm?Prod=VDEC1

[71] F. Dittmann, A. Rettberg, and F. Schulte, "A y-chart based tool for reconfigurable system design," in *Workshop on Dynamically Reconfigurable Systems (DRS)*.   Innsbruck, Austria: VDE Verlag, 17 Mar. 2005, pp. 67–73.

[72] P. S. B. do Nascimento, M. E. de Lima, S. M. da Silva, and J. L. Seixas, "Mapping of image processing systems to fpga computer based on temporal partitioning and design space exploration," in *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*.   New York, NY, USA: ACM Press, 2006, pp. 50–55.

[73] W. E. Donath, "Hierarchical placement method," *IBM Technical disclosure Bulletin*, vol. 17, no. 10, pp. 3121–3125, 1975.

[74] W. E. Donath and A. J. Hoffman, "Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices," *IBM Technical disclosure Bulletin*, vol. 15, no. 3, pp. 938–944, 1972.

[75] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, 1997. [Online]. Available: http://citeseer.ist.psu.edu/eggers97simultaneous.html

[76] Ejnioui and N. Ranganathan, "Circuit scheduling on time-multiplexed fpgas." [Online]. Available: citeseer.nj.nec.com/17858.html

[77] H. A. ElGindy, A. K. Somani, H. Schroeder, H. Schmeck, and A. Spray, "RMB - A Reconfigurable Multiple Bus Network," in *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, San Jose, California, Feb. 1996, pp. 108–117.

[78] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blümel, "A reconfigurable system on chip implementation for elliptic curve cryptography over gf(2n)," in *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*.   London, UK: Springer-Verlag, 2003, pp. 381–399.

[79] G. Estrin and C. R. Viswanathan, "Organisation of a "fixed-plus-variable" structure computer for computation of eigenvalues and eigenvectors of real symetric matrices," *Journal of the ACM*, vol. 9, pp. 41–60, 1962.

[80] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel processing in a restructurable computer system," *IEEE Transactions on Electronic Computers*, vol. 12, no. 5, pp. 747–755, 1963.

[81] G. Estrin and R. Turn, "Automatic assignment of computations in a variable structure computer system," *IEEE Transactions on Electronic Computers*, vol. 12, no. 5, pp. 755–773, 1963.

[82] S. P. Fekete, E. Köhler, and J. Teich, "Optimal fpga module placement with temporal precedence constraints," Technische Universität Berlin, Tech. Rep. 696.2000, 2000.

[83] C. M. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of the 19th Design Automation Conference*, 1982, pp. 175–181.

[84] L. Ford and D. Fulkerson, *Flows in Networks*. Princeton University Press, 1962.

[85] P. Foulk, "Data-folding in SRAM configurable FPGA," in *IEEE Workshop on FPGAs for Custom Computing Machines*. IEEE, 1993, pp. 163–171.

[86] R. FRANCIS, "Technology mapping for lookup-table based fieldprogrammable gate arrays," Ph.D. dissertation, University of Toronto, 1992. [Online]. Available: citeseer.ist.psu.edu/francis93technology.html

[87] R. J. Francis, J. Rose, and K. Chung, "Chortle: a technology mapping program for lookup table-based field programmable gate arrays," in *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*. New York, NY, USA: ACM Press, 1990, pp. 613–619.

[88] J. Fry and M. Langhammer, "Rsa and public key cryptography in fpgas." [Online]. Available: www.altera.com/literature/cp/rsa-public-key-cryptography.pdf

[89] F. C. Furtek, E. Hogenauer, and J. Scheuermann, "Interconnecting heterogeneous nodes in an adaptive computing machine." in *Proceedings of International Conference on Field-P rogrammable Logic and Applications (FPL)*, ser. Lecture Notes in Computer Science (LNCS), vol. 3203. Antwep, Belgium: Springer, Aug. 2004, pp. 125–134.

[90] K. Gaj and P. Chodowiec, "Comparison of the hardware performance of the aes candidates using reconfigurable hardware." in *AES Candidate Conference*, 2000, pp. 40–54.

[91] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Des. Test*, vol. 11, no. 4, pp. 44–54, 1994.

[92] D. D. Gajski and M. Reshadio, "Application and advantages," CECS, UC Irvin, Technical Report 04-12, 2004.

[93] C. H. Gebotys, "An optimal methodology of synthesis of dsp multichip architectures," *Journal of VLSI Signal Processing*, vol. 11, pp. 9–19, 1995.

[94] L. Geppert, "Sun's big splash," *IEEE Spectrum Magazine*, pp. 21–29, M, January 2005 2005.

[95] J. Gerling, K. Danne, C. Bobda, and J. Schrage, "Distributed arithmetics for recursive convolution of optical intercannects," in *EOS Topical Meeting, Optics in Computing (OIC)*, Engelberg (Switzerland), Apr. 2004, pp. 65–66.

[96] P. B. Gibbons and S. S. Muchnick, "Efficient instruction scheduling for a pipelined architecture," in *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*.   New York, NY, USA: ACM Press, 1986, pp. 11–16.

[97] M. Gokhale, W. Holmes, A. Kopser, D. Kunze, D. P. Lopresti, S. Lucas, R. Minnich, and P. Olsen, "Splash: A reconfigurable linear logic array." in *ICPP (1)*, 1990, pp. 526–532.

[98] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: A coprocessor for streaming multimedia acceleration," in *ISCA*, 1999, pp. 28–39. [Online]. Available: citeseer.ist.psu.edu/goldstein99piperench.html

[99] G. H. Golub and C. F. V. Loan, *Matrix Computations*.   North Oxford Academic Publisching, 1983.

[100] C. E. D. C. Green and P. Franklin, "RaPiD – reconfigurable pipelined datapath," in *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, R. W. Hartenstein and M. Glesner, Eds.   Darmstadt, Germany: Springer-Verlag, 1996, pp. 126–135. [Online]. Available: citeseer.ist.psu.edu/ebeling96rapid.html

[101] S. Guccione, "Programming fine-grained reconfigurable architechture," Ph.D. dissertation, The University of Texas at Austin, May 1995.

[102] S. Guccione, D. Levi, and P. Sundararajan, "Jbits: A java-based interface for reconfigurable computing," 1999.

[103] M. Gulati and N. Bagherzadeh, "Performance study of a multithreaded superscalar microprocessor," in *Proc. Intl. Symp. High-Performance Computer Architecture, ACM*, 1996, pp. 291–301.

[104] B. Gunther and G. Milne, "Accessing document relevance with run-time reconfigurable machines," in *IEEE Workshop on FPGAs for Custom Computing Machines*.   Napa California: IEEE, 1996, pp. 9–16.

[105] B. Gunther and G. Milne, "Hardware-based solution for message filtering," school of computer and information science, Tech. Rep., 1996.

[106] R. H. Güting, "An optimal contour algorithm for iso-oriented rectangles," *J. Algorithms*, vol. 5, pp. 303–326, 1984.

[107] K. Hall, "An r-dimensional quadratic dimensional quadratic placement algorithm," *Journal of Management Science*, vol. 17, no. 3, pp. 219–229, 1970.

[108] T. Haller, "Adaptive multiprocessing on reconfigurable chip," 2006, master Thesis.

[109] L. Hammond, B. A. Nayfeh, and K. Olukotun, "The hydra chip," *IEEE MICRO Magazine*, pp. 71–83, March-April 2000. [Online]. Available: http://citeseer.ist.psu.edu/287939.html

[110] R. Hartenstein, "A decade of reconfigurable computing: A visionary Retrospective," in *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*. IEEE Computer Society, March 2001, pp. 290–295.

[111] R. Hartenstein, *Morphware and Configware*, A. Y. Zomaya, Ed. New York: Springer-Verlag, 2006.

[112] R. Hartenstein, *Basics of Reconfigurable Computing*, S. P. J. Henkel, Ed. New York: Springer-Verlag, 2007.

[113] R. Hartenstein, A. Hirschbiel, and M.Weber, "Xputers - an open family of non von neumann architectures," in *11th ITG/GI Conference on Architektur von Rechensystemen*. VDE-Verlag, 1990.

[114] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1997, pp. 12–21. [Online]. Available: citeseer.nj.nec.com/hauser97garp.html

[115] S. D. Haynes, J. Stone, P. Y. K. Cheung, and W. Luk, "Video image processing with the sonic architecture," *Computer*, vol. 33, no. 4, pp. 50–57, 2000.

[116] P. Healy and M. Creavin, "An Optimal Algorithm for Rectangle Placement," Dept. of Computer Science and Information Systems, University of Limerick, Limerick, Ireland, Tech. Rep. UL-CSIS-97-1, Aug. 1997.

[117] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist, "Network on chip: An architecture for billion transistor era," in *In Proceeding of the IEEE NorChip Conference, November 2000.*, 2000.

[118] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *SIAM Journal on Scientific Computing*, vol. 16, no. 2, pp. 452–469, 1995. [Online]. Available: citeseer.nj.nec.com/hendrickson95improved.html

[119] M. R. Hestenes, "Inversion of matrices by biorthogonalization and related results," *J. Soc. Indust. Appl. Math.*, vol. 6, no. 1, pp. 51–90, 1958.

[120] Hirata et al., "An elementary processor architecture with simultaneous instruction issuing from multiple threads," in *Proc. Intl. Symp. Computer Architecture, Assoc. of Computing Machinery*, 1992, pp. 136–145.

[121] G. home page, "Gnu radio - the gnu software radio," http://www.gnu.org/software/gnuradio/. [Online]. Available: http://www.gnu.org/software/gnuradio/

[122] J. Huie, P. DŠAntonio, R. Pelt, and B. Jentz, "Synthesizing fpga cores for software-defined radio." [Online]. Available: www.altera.com/literature/cp/fpga-cores-for-sdr.pdf

[123] S. Iman, M. Pedram, C. Fabian, and J. Cong, "Finding uni-directional cuts based on physical partitioning and logic restructuring," in *Fourth International Workshop on Physical Design*. IEEE, 1993.

[124] A. Inc, *Nios II Processor Reference Handbook*, November, 2006. [Online]. Available: http://www.altera.com/literature/lit-nio2.jsp

[125] C. inc, *Handel-C Reference Manual*, 2000, www.celoxica.com/techlib/files/CEL-W0410251JJ4-60.pdf. [Online]. Available: www.celoxica.com/techlib/files/CEL-W0410251JJ4-60.pdf

[126] X. Inc, *PowerPC 405 Processor Block Reference Guide: Embedded Development Kit*, July 20, 2005. [Online]. Available: www.xilinx.com/bvdocs/userguides/ug018.pdf

[127] X. Inc, *MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 8.2i*, June 1, 2006. [Online]. Available: www.xilinx.com/ise/embedded/mb_ref_guide.pdf

[128] X. Inc, *XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based*, September 9, 2004. [Online]. Available: www.xilinx.com/bvdocs/appnotes/xapp290.pdf

[129] J. M. Joseph Mitola, *Software Radio Architecture: Object-Oriented Approaches to Wireless Systems Engineering*. Wiley, John & Sons, Incorporated, 2001.

[130] S. Jung, "Entwurf eines verfahrens und einer umgebung zur durchgängigen konfiguration adativer on-chip multiprozessor," 2006, master Thesis.

[131] Jürgen Reichardt, Bernd Schwarz, *VHDL-Synthese*, 3rd ed. Oldenbourg-Verlag, Dec 2003. [Online]. Available: http://users.etech.fh-hamburg.de/users/reichardt/buch.html

[132] H. Kalte, M. Pormann, and U. Rueckert, "Rapid prototyping system f"ur dynamisch rekonfigurierbarer hardware struckturen," in *AES 2000*, 2000, pp. 149–157.

[133] K. Karplus, "Xmap: A technology mapper for table-lookup field-programmable gate arrays," in *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*. New York, NY, USA: ACM Press, 1991, pp. 240–243.

[134] R. Kastner, A. Kaplan, and M. Sarrafzadeh, *Synthesis Techniques and Optimizations for Reconfigurable Systems*. Amsterdam: Kluwer Academic Publishers, 2003.

[135] M. Kaul and R. Vemuri, "Optimal temporal partitioning and synthesis for reconfigurable architectures," 1998.

[136] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss, "An automated temporal partitioning tool for a class of dsp applications," 1998. [Online]. Available: citeseer.nj.nec.com/160625.html

[137] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[138] N. KOBLITZ, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[139] P. Kongetira, K. Aingaran, and K. Olukotun, "The hydra chip," *IEEE MICRO Magazine*, pp. 21–29, March-April 2005. [Online]. Available: http://citeseer.ist.psu.edu/287939.html

[140] H. Kropp, C. Reuter, M. Wiege, T.-T. Do, and P. Pirsch, "An fpga-based prototyping system for real-time verification of video processing schemes." in *FPL*, 1999, pp. 333–338.

[141] K. Kucukcakar and A. Parker, "Chop: A constraint-driven-system-level partitioner," in *Design Automation Conference*, 1991, pp. 514–519.

[142] P. Kurup and T. Abbasi, *Logic Synthesis Using Synopsys*. Kluwer Akademic publisher, 1997.

[143] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett, "Local microcode compaction techniques," *ACM Comput. Surv.*, vol. 12, no. 3, pp. 261–294, 1980.

[144] T. Lengauer, *Combinatorial Algorithm for Integrated Circuit Layout*. Teubner Stuttgart, 1990.

[145] K. H. Leung, K. W. Ma, W. K. Wong, and P. H. W. Leong, "Fpga implementation of a microcoded elliptic curve cryptographic processor," in *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2000, p. 68.

[146] H. Li and Q. Stout, *Reconfigurable Massively Parallel computers*, H. Li and Q. Stout, Eds. Glasgow, UK: Prentice-Hall, 1991.

[147] W. Lipski, Jr. and F. P. Preparata, "Finding the contour of a union of iso-oriented rectangles," *J. Algorithms*, vol. 1, pp. 235–246, 1980, errata in 2(1981), 105; corrigendum in 3(1982), 301–302.

[148] H. Liu and D. F. Wong, "Network flow-based circuit partitioning for time-multiplexed FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design*, 1998, pp. 497–504.

[149] H. Liu and D. F. Wong, "Circuit partitioning for dynamicaly reconfigurable fpgas," in *International Symposium on Field Programmable Gate Arrays(FPGA 98)*. Monterey, California: ACM/SIGDA, 1999, pp. 187 – 194.

[150] A. S. Ltd., *High Performance AES Encryption Cores*. [Online]. Available: http://http://www.jat.co.kr/newsletter/2001_11_24/ds5210-40.pdf

[151] S.-M. Ludwig, "Hades-fast hardware synthesis tools and a reconfigurable coprocessor," Ph.D. dissertation, Swiss Federal Institute of Technologie, Zürich, 1997.

[152] J. Lutz and A. Hasan, "High performance fpga based elliptic curve cryptographic co-processor," in *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*. Washington, DC, USA: IEEE Computer Society, 2004, p. 486.

[153] P. S. G. Maya Gokhale, *Reconfigurable Computing: Accelerating Computation with Field-programmable Gate Arrays*. Berlin: Springer, 2005.

[154] A. Mazzeo, L. Romano, G. P. Saggese, and N. Mazzocca, "Fpga-based implementation of a serial rsa processor," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10582.

[155] A. Michalski and D. Buell, "A scalable architecture for rsa cryptography on large fpgas," in *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 331–332.

[156] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[157] V. S. Miller, "Use of elliptic curves in cryptography," in *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*. New York, NY, USA: Springer-Verlag New York, Inc., 1986, pp. 417–426.

[158] L. Minzer, "Programmable silicon for embedde signal processing," *Embedded Systems Programming*, pp. 110–133, March 2000.

[159] E. Mirsky and A. DeHon, "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1996, pp. 157–166. [Online]. Available: citeseer.ist.psu.edu/mirsky96matrix.html

[160] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable multimedia array coprocessor (abstract)," in *FPGA*, 1998, p. 261. [Online]. Available: citeseer.ist.psu. edu/miyamori98remarc.html

[161] A. Morse, "Control using logic-based switching," *Trends in Control, Springer, London*, 1995.

[162] M. C.-G. H. Mr. Adam Harrington and D. S. S. Jones, "Software-defined radio: The revolution of wireless communication," *Annual Review of Communications*, vol. 58, 2005.

[163] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*. New York, NY, USA: ACM Press, 1990, pp. 620–625.

[164] R. Murgai, N. V. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Performance directed synthesis for table look up programmable gate arrays." in *ICCAD*, 1991, pp. 572–575.

[165] I. Nallatech, `http://www.nallatech.com`.

[166] Narendra and Balakrishnan, "Adaptive control using multiple models: Switching and tuning," *Yale Workshop on Adaptive and Learning Systems*, 1994.

[167] M. Nikitovic and M. Brorsson, "An adaptive chip-multiprocessor architecture for future mobile terminals." in *CASES*, 2002, pp. 43–49.

[168] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation," in *DAC '02: Proceedings of the 39th conference on Design automation*. New York, NY, USA: ACM Press, 2002, pp. 22–27.

[169] K. Olukotun and L. Hammond, "The future of microprocessors," *ACM Queue*, vol. 3, no. 7, pp. 26–29, September 2005. [Online]. Available: http://doi.acm.org/10.1145/1095408.1095418

[170] OSCI, *SystemC Reference Manual*, 2003. [Online]. Available: www.systemc.org

[171] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures," in *IPPS/SPDP Workshops*, 1998, pp. 31–36. [Online]. Available: citeseer.nj.nec.com/ouaiss98integrated.html

[172] Y. Pan and M. Hamdi, "Singular value decomposition on processors arrays with a pipelined bus system," *Journal of Network and Computer Applications*, vol. 19, pp. 235–248, 1996.

[173] A. Pandey and R. Vemuri, "Combined temporal partitioning and scheduling for reconfigurable architectures," in *Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844*, J. Schewel, P. M. Athanas, S. A. Guccione, S. Ludwig, and J. T. McHenry, Eds. Bellingham, WA: SPIE – The International Society for Optical Engineering, 1999, pp. 93–103.

[174] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of asic's." *IEEE Transactions on CAD*, vol. 6, no. 8, pp. 661–679, 1989.

[175] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Prentice Hall,, April 2005.

[176] M. Petrov, T. Murgan, F. May, M. Vorbach, P. Zipf, and M. Glesner, "The XPP architecture and its co-simulation within the simulink environment." in *Proceedings of International Conference on Field-P rogrammable Logic and Applications (FPL)*, ser. Lecture Notes in Computer Science (LNCS), vol. 3203. Antwep, Belgium: Springer, Aug. 2004, pp. 761–770.

[177] M. Platzner and L. Thiele, "XFORCES - executives for reconfigurable embedded systems." [Online]. Available: http://www.ee.ethz.ch/\~platzner

[178] V. K. Prasanna and A. Dandalis, "Fpga-based cryptography for internet security." [Online]. Available: halcyon.usc.edu/~pk/prasannawebsite/papers/dandalisOSEE00.pdf

[179] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. New York, NY: Springer-Verlag, 1985.

[180] D. Pryor, M.R.Thisle, and N.shirazi, "Text searching on splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines*. IEEE, 1993, pp. 172–177.

[181] K. M. G. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 579–590, 1999.

[182] F. J. Rammig, "A concept for the editing of hardware resulting in an automatic hardware-editor," in *Proceedings of 14th Design Automation Conference*, New Orleans, 1977, pp. 187–193.

[183] D. Rech, "Automatische generierung und konfiguration von adaptiven on-chip symmetrical multiprocessing (smp) systemen," 2006, master Thesis.

[184] E. Rijpkema, K. G. W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10350.

[185] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[186] R. L. Rivest and C. E. Leiserson, *Introduction to Algorithms*. New York, NY, USA: McGraw-Hill, Inc., 1990.

[187] C. Rowen and S. Leibson, "Flexible architectures for engineering successful socs." in *DAC*, 2004, pp. 692–697.

[188] S. M. Scalera and J. R. Vázquez, "The design and implementation of a context switching fpga," in *IEEE Symposium on FPGAs for Custom Computing Machines*. Napa Valey, CA: IEEE Computer Society Press, April 1998, pp. 78–85.

[189] SDR, "The sdr forum's website." [Online]. Available: http://www.gnu.org/software/gnuradio/

[190] P. Sedcole, "Reconfigurable platform-based design in fpgas for video image processing," Ph.D. dissertation, Imperial College, London, January 2006.

[191] SGI, *http://www.sgi.com/products/rasc/*.

[192] N. Shirazi, A. Walters, and P. M. Athanas, "Quantitative analysis of floating point arithmetic on fpga based custom computing machines." in *FCCM*, 1995, pp. 155–163.

[193] R. G. Shoup, "Parameterized convolution filtering in an FPGA," in *Proceedings of International Conference on Field Programmable Logic and Arrays*, 1993, pp. 274–280.

[194] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, "Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[195] M. J. S. Smith, *Application-specific integrated circuits*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[196] C. Steiger, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1393–1407, 2004, member-Herbert Walder and Member-Marco Platzner.

[197] C. Steiger, H. Walder, M. Platzner, and L. Thiele, "Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices," in *Proceedings of the 24th International Real-Time Systems Symposium (RTSS'03)*, December 2003.

[198] M. A. Tahir, A. Bouridane, and F. Kurugollu, "An fpga based coprocessor for glcm and haralick texture features and their application in prostate cancer classification," *Analog Integr. Circuits Signal Process.*, vol. 43, no. 2, pp. 205–215, 2005.

[199] J. Teich, S. P. Fekete, and J. Schepers, "Optimizing dynamic hardware reconfiguration," Angewante Mathematik Und Informatik Universität zu Köln, Tech. Rep. 97.228, 1998.

[200] G. R. G. S. J. Thomas, "An optimal parallel jacobi-like solution method for the singular value decomposition," in *Proc. Int. Conf. on Parallel Processing*, January 1988.

[201] C. Torres-Huitzil and M. Arias-Estrada, "Fpga-based configurable systolic architecture for window-based image processing," *EURASIP Journal on Applied Signal Processing*, vol. 2005, no. 7, pp. 1024–1034, 2005, doi:10.1155/ASP.2005.1024.

[202] S. Trimberger, "Scheduling designs into a time-multiplexed fpga," in *International Symposium on Field Programmable Gate Arrays(FPGA 98)*. Monterey, California: ACM/SIGDA, 1998, pp. 153 – 160.

[203] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[204] R. Vaidyanathan and J. L. Trahan, *Dynamic Reconfiguration: Architectures and Algorithms*. IEEE Computer Society, 2003.

[205] R. Vaidyanathan and J. L. Trahan, *Dynamic Reconfiguration: Architectures and Algorithms (Series in Computer Science (Kluwer Academic/Plenum Publishers).)*. Plenum Publishing Co., 2004.

[206] A. van Breemen and T. de Vries, "An agent-based framework for designing multi-controller systems," *Proc. of the Fifth International Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology, pp. 219-235, Manchester, U.K*, Apr. 2000.

[207] C. Veciana-Nogués and J. Domingo-Pascual, "Adaptive video on demand service on rsvp capable network," in *ECMAST '99: Proceedings of the 4th European Conference on Multimedia Applications, Services and Techniques*. London, UK: Springer-Verlag, 1999, pp. 212–228.

[208] "Vhdl online." [Online]. Available: http://www.vhdl-online.de/

[209] J. E. Volder, "The birth of cordic," *J. VLSI Signal Process. Syst.*, vol. 25, no. 2, pp. 101–105, 2000.

[210] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997. [Online]. Available: citeseer.ist.psu.edu/waingold97baring.html

[211] H. Walder, S. Nobs, and M. Platzner, "Xf-board: A prototyping platform for reconfigurable hardware operating systems." in *ERSA*, 2004, p. 306.

[212] H. Walder and M. Platzner, "A runtime environment for reconfigurable hardware operating systems." in *FPL*, 2004, pp. 831–835.

[213] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, vol. 26, no. 4. New York, NY: ACM Press, 1991, pp. 176–189. [Online]. Available: http://citeseer.ist.psu.edu/wall90limits.html

[214] K. Weiss, T. Steckstor, C. Ötker, I. Katchan, C. Nitsch, and J. Philipp, *Spyder - Virtex - X2 User's Manual*, 1999.

[215] S. A. White, "Application of distributed arithmetic to digital signal processing: A tutorial review," *IEEE ASSP Magazine*, pp. 4–19, July 1989.

[216] G. Wigley and D. Kearney, "The development of an operating system for reconfigurable computing," in *Proceedings of the 9th IEEE Symposium Field-Programmable Custom Computing Machines(FCCM'01)*. IEEE-CS Press, April 2001.

[217] S. E. G. William Lehr, Fuencisla Merino, "Software radio: Implications for wireless services, industry structure, and public policy," Massachusetts Institute of Technology, Program on Internet and Telecoms Convergence, Tech. Rep., Aug. 2002.

[218] J. W. Williams and N. Bergmann, "Embedded linux as a platform for dynamically self-reconfiguring systems-on-chip," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2004, pp. 163–169. [Online]. Available: http://espace.library.uq.edu.au/view.php?pid=UQ:410

[219] M. Wirthlin and B. Hutchings, "A dynamic instruction set computer," in *IEEE Symposium on FPGAs for Custom Computing Machines*, P. Athanas and K. L. Pocek, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1995, pp. 99–107. [Online]. Available: citeseer.nj.nec.com/wirthlin95dynamic.html

[220] P. L. E. Wolfgang Rosenstiel, *New Algorithms. Architectures and Applications for Reconfigurable Computing*. Berlin: Springer, 2005.

[221] T. Wollinger, J. Guajardo, and C. Paar, "Security on fpgas: State-of-the-art implementations and attacks," *Trans. on Embedded Computing Systems*, vol. 3, no. 3, pp. 534–574, 2004.

[222] T. J. Wollinger, M. Wang, J. Guajardo, and C. Paar, "How well are high-end dsps suited for the aes algorithms? aes algorithms on the tms320c6x dsp." in *AES Candidate Conference*, 2000, pp. 94–105.

[223] Xilinx, "A guide to using field programmable gate arrays (fpgas) for application-specific digital signal processing performance," *http://www.xilinx.com*, 1995.

[224] Xilinx, "The role of distributed arithmetic design in fpga-based signal processing," *http://www.xilinx.com*, 2000.

[225] Xilinx, "Spartan-3 fpgas," *http://www.xilinx.com*, 2000.

[226] Xilinx Inc., "The early access partial reconfiguration lounge," (registration required). [Online]. Available: http://www.xilinx.com/support/prealounge/protected/index.htm

[227] Xilinx Inc., "Early access partial reconfiguration user guide," xilinx User Guide UG208, Version 1.1, March 6, 2006. [Online]. Available: http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf

[228] Xilinx Inc., "Virtex-II Pro XUP Development Board," http://www.xilinx.com/univ/xupv2p.html.

[229] Xilinx Inc., "Xilinx ISE 8 Software Manuals and Help - PDF Collection," 2005. [Online]. Available: http://toolbox.xilinx.com/docsan/xilinx8/books/manuals.pdf

[230] Xilinx Inc., "Edk platform studio documentation," 2007. [Online]. Available: http://www.xilinx.com/ise/embedded/edk_docs.htm

[231] H. Yang and D. F. Wong, "Efficient network flow based min-cut balanced partitioning," in *International Conference on Computer-Aided Design*, 1994.

[232] Y. Yang, Z. Abid, and W. Wang, "Two-prime rsa immune cryptosystem and its fpga implementation," in *GLSVSLI '05: Proceedings of the 15th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM Press, 2005, pp. 164–167.

Appendix A
Hints to Labs

This chapter gives a step-by-step guide in note form to create a partially reconfigurable system. It has been developed while reconstructing the *Video8* example and is intended to give hints to the reader on how to create own designs. A more detailed description of the demonstration project can be found in 5.2 on page 236. The sources of the guide can be found on the book's Web page. Following the directory and entity names, refer to the *Video8* project. A profound knowledge of VHDL, ISE and EDK will be needed to comprehend the instructions.

Tutorial
Creation of partially reconfigurable designs
On the Example of *Video8*
Version 1.0

1. Prerequisites

Following basic conditions should be present when accomplishing this guide. In case of deviations the given approach might not be feasible.

- ISE 8.1.01i PR8 or PR12 (*Early Access Partial Reconfiguration* patch for ISE)
- EDK 8.1.02i

Abbreviations:

PR	partially reconfigurable
PRM	partially reconfigurable module
TL	top-level
TLM	top-level module
DIR	directory
\<base\>	root directory of the project

2. Reorganization of the Project `Video8_non_pr`

1. elevate PRM (`rgbfilter`) to the TLM (`top.vhd`): initially `rgbfilter` is instantiated in entity `video_in`. But PRM may not be sub-modules of a static part. This change now is only the first step. `rgbfilter` will have to be brought up completely to the top-level module.

   1.1. In `video_in.vhd`: additional ports have to be adjoined to the entity declaration of `video_in` so that `rgbfilter` can be instantiated outside this entity but the rest may stay the same as before.

```
entity video_in is {
        ...
        LLC_CLOCK_to_filter : out std_logic;
        R_out_to_filter : out std_logic_vector(0 to 9);
        G_out_to_filter : out std_logic_vector(0 to 9);
        B_out_to_filter : out std_logic_vector(0 to 9);
        h_counter_out_to_filter : out std_logic_vector(0 to 9);
        v_counter_out_to_filter : out std_logic_vector(0 to 9);
        valid_out_to_filter : out std_logic;
        R_in_from_filter : in std_logic_vector(0 to 9);
        G_in_from_filter : in std_logic_vector(0 to 9);
        B_in_from_filter : in std_logic_vector(0 to 9);
        h_counter_in_from_filter : in std_logic_vector(0 to 9);
        v_counter_in_from_filter : in std_logic_vector(0 to 9);
        valid_in_from_filter : in std_logic
        ...
};
```

1.2. In `video_in.vhd`: modules that formerly accessed the `rgbfilter` instance have to be redirected to outward ports. For example simply by rewriting corresponding signals:

```
-- signals out to the filter:
R_out_to_filter <= R1;
G_out_to_filter <= G1;
B_out_to_filter <= B1;
h_counter_out_to_filter <= h_counter1;
v_counter_out_to_filter <= v_counter1;
valid_out_to_filter <= valid1;
-- processed signals in from the filter:
R2 <= R_in_from_filter;
G2 <= G_in_from_filter;
B2 <= B_in_from_filter;
h_counter2 <=  h_counter_in_from_filter;
v_counter2 <=  v_counter_in_from_filter;
valid2 <= valid_in_from_filter;
```

2. making the PRM available to EDK (not necessary; good for testing purposes)

2.1. adapt .mpd and .pao (PCORE-definition file und -synthesizing directives) for the `video_in`-module:

- create .mpd from .vhd with:

```
psfutil -hdl2mpd video_in_pr.vhd -bus opb m \\
                -lang vhdl -o VIDEO_IN_PR_v2_1_0.mpd
```

- In .pao: delete `rgbfilter` since it's a PRM. It'll be synthesized separately.

```
lib VIDEO_IN_v1_00_a rgbfilter
```

löschen.

- rename `video_in` to `video_in_pr`

2.2. Create an own PCORE for `rgbfilter`:

- copy `video_in`
- fit .pao:

```
lib RGBFILTER_v1_00_a rgbfilter
```

- use only `rgbfilter.vhd` and create .mpd from it

2.3. rename `VIDEO_IN_v1_00_a` to `VIDEO_IN_PR_v1_00_a`

2.4. include `Video_in_pr` and `rgbfilter` in the EDK project. The custom PCOREs can be found under 'IP Catalog', 'Project Repository'. Erase the PCORE `video_in` from the EDK project. Now wire the ports: `rgbfilter` receives several signals from `video_in`. `video_in` takes in the results from `rbgfilter`.

2.5. Attention has to be paid to the correct address ranges of opb2plb-bridge :
0x0 - 0x0fffffff
The modules accessible over these addresses in this block will be supplied to the bus participants on every side of the bridge. This is important because the data flows from `video_in` to the frame buffer situated in the RAM (→ Memory-Mapped I/O).

2.6.   Synthesis in EDK (for testing). If errors occur like

```
ERROR:HDLParsers:3317 - "D:/murr/edkProjects/ \\
Video8_81_PR/hdl/video_in_pr_0_wrapper.vhd" Line 10.
Library VIDEO_IN_PR_v1_00_a cannot be found.

ERROR:HDLParsers:3014 - "D:/murr/edkProjects/ \\
Video8_81_PR/hdl/video_in_pr_0_wrapper.vhd" Line 11.
Library unit VIDEO_IN_PR_v1_00_a \\
is not available in library work.
```

then the `.pao` files have not been altered correctly as stated above (e.g. not all occurrences of `video_in` have been changed to `video_in_pr`)

2.7.   add additional filters if desired (e.g. a mean-value filter)

2.8.   place the address of the frame buffer at the end of the DDR-RAMs (at address 0x00001111111000000000000000000000) to allow the Linux System to start from 0x0

→ adapt files, since `Video8` unfortunately received hard-coded addresses from its author:

- In `rgbstream2framebuffer.vhd`:
  ```
  next_wb_counter <= ram_out(24 to 33) + \\
  (ram_out(34 to 43) * "00000000001010000000") ;
  ```

  changes to:
  ```
  next_wb_counter <= ram_out(24 to 33) + \\
  (ram_out(34 to 43) * "00000000001010000000") + \\
  "00001111111000000000000000000000" ;
  ```

- adapt `simple_timer_unit.vhd`:
  ```
  reader_addr_plb_unsynced <= (others => '0');
  ```

  changes twice to:
  ```
  reader_addr_plb_unsynced <= \\
  "00001111111000000000000000000000"
  ```

3.   prepare the EDK-project for PR:

   3.1.   supply additional IP-Cores:
   - `JTAG-PPC` controller (debugging)
   - `opb_sysace`
   - `opb_timer` (to measure the time the reconfiguration process takes)
   - `opb_hwicap`

   3.2.   Bus connections
   - connect `opb_timer` with the opb the `sysace` is already placed at
   - connect `opb_hwicap` with the same opb

   3.3.   fill in addresses:

- sysace: 0x41800000 - 64K
- opb_timer: 0x41c00000 - 64K
- opb_hwicap: 0x40200000 - 64K
- docm_ctlr: 0xE8800000 - 32K → so that the program fits into the BRAMs. If put in the RAM unwanted deviations occur in the resulting picture.
- iocm_ctrl: 0xFFFF8000 - 32K /

3.4. extend the .ucf with sysace-pinouts. Entries of the form:

```
Net fpga_0_SysACE_CompactFlash_SysACE_CLK_....
```

→ see appendix B

3.5. connect ports:

- opb_sysace:
    ```
    OPB_Clk - sys_clk_s
    SysACE_MPA -> make external
    SysACE_CLK -> make external
    SysACE_MPD -> make external
    SysACE_MPD_I -> no connection
    SysACE_MPD_O -> no connection
    SysACE_MPD_T -> no connection
    SysACE_CEN -> make external
    SysACE_OEN -> make external
    SysACE_WEN -> make external
    SysACE_MPIRQ -> no connection
    ```
    → not to forget to adapt the pins in the .ucf
- opb_timer:
    ```
    OPB_clk -> sys_clk_s
    ```

- also fit the external ports to the ones in the .ucf

3.6. Extend the C-program for PR with ICAP:

- copy xhwicap* files from John William's ICAP driver [218]
- adapt main.c: first configure the video decoder card, then present the PR menu
- can e.g. sysace_stdio.h not be found add the xilfatfs-library under Software|Software Platform Settings|Platform Settings
- create a linker script for the application (if possible put all the program parts to iocm- resp. docm-memory (→ RAM very often causes problems ⇒ if necessary extend the address space of the corresponding controller)

3.7. Testing: completely compile with EDK and see if the system works so far

3.8. enter ports for the PRM in the port declaration of entity system as external ports:

- close the EDK project
- redirect ports to and from rgbfilter from the local instance to the corresponding external ports:
    → add to system.mhs at the external port (at the beginning of the file):
    ```
    PORT LLC_CLOCK_OUT = VIDEO_IN_PR_0_LLC_CLOCK_to_filter,
    DIR = O
    ```

```
                    PORT R_in = VIDEO_IN_PR_0_R_out_to_filter, DIR = O, VEC
                    = [0:9]
                    PORT G_in = VIDEO_IN_PR_0_G_out_to_filter, DIR = O, VEC
                    = [0:9]
                    PORT B_in = VIDEO_IN_PR_0_B_out_to_filter, DIR = O, VEC
                    = [0:9]
                    PORT h_counter_in = VIDEO_IN_PR_0_h_counter_out_to_
                    filter, \\
                        DIR = O, VEC = [0:9]
                    PORT v_counter_in = VIDEO_IN_PR_0_v_counter_out_to_
                    filter, \\
                        DIR = O, VEC = [0:9]
                    PORT valid_in = VIDEO_IN_PR_0_valid_out_to_filter, DIR = O
                    PORT R_out = RGBFILTER_0_R_out, DIR = I, VEC = [0:9]
                    PORT G_out = RGBFILTER_0_G_out, DIR = I, VEC = [0:9]
                    PORT B_out = RGBFILTER_0_B_out, DIR = I, VEC = [0:9]
                    PORT h_counter_out = RGBFILTER_0_h_counter_out, \\
                        DIR = I, VEC = [0:9]
                    PORT v_counter_out = RGBFILTER_0_v_counter_out, \\
                        DIR = I, VEC = [0:9]
                    PORT valid_out = RGBFILTER_0_valid_out, DIR = I
```

- ■ reopen EDK project
- ■ erase instance `rgbfilter` in `System Assembly|Bus Interface`

3.9. export to ISE:

→ in `Project|Project-Options|Hierarchy and Flow`:

- ■ Design is a sub-module: name: `system_i`
- ■ do not synthesize non-Xilinx IPs
- ■ use project navigator implementation flow (ISE)
- ■ do not add modules to an existing ISE

4. Synthetizing the project with ISE

4.1. copy the content of the EDK-directory to `<base>/synth/edk`

4.2. Verify the ISE project:

→ open `system.ise` in `base/synth/edk/projnav` with ISE:

→ verify if the design is structured as follows:

- ■ `system_stub.vhd` (`system_stub`)
  - `system.vhd` (`system_i`)
    - `ppc405_0_wrapper` (`ppc405_0`)
    - `ppc405_1_wrapper` (`ppc405_1`)
    - `jtagppc_0_wrapper` (`jtagppc_0`)
    - `reset_block_wrapper` (`reset_block`)
    - ⋮
    - `system_stub.bmm`

→ if the structure is not like the above, check for the correct export settings in EDK. Mind the 'do not add modules to an existing ISE project'-setting!

4.3. mark and rightclick `system_stub`, select 'Synthesize - XST'-> properties: Xilinx Specific Options-> do not add I/O Buffers (since the system is to be a sub-module)

4.4. create netlists for the EDK-module with doubleclicking 'Synthesize - XST'

4.5. the suiting Block-RAM memory mapping file (`system_stub_bd.bmm`) can be created with doubleclicking 'Generate Programming File' (every step necessary will be done, including synthetization)

→ if that does not work out there is *probably* something wrong (a workaround *might* be compiling with I/O-buffers to get the `system_stub_bd.bmm` and then again without to generate the appropriate `.ngc`-files)

5. Changes to the skript framework: in directory `synth`
   The example `Video8` comes with a bunch of scripts to support up the PRM generation. This part gives hints on how to change those appropriately.
   Directories:

   - `mod_*` directories according to the modules (sobel, sharpen, mean-value, unfiltered) ändern

   - `top`-directory remains

   - `static` - not needed (in this case contained in `edk`

   - `edk` - copy previously generated design here

   Files:

   - adapt `doit.cmd` → descend in new DIR and execute `xst`

   - in these DIR: adapt `.prj` and `.xst` (are the same in all the DIR)

6. Changes to the project files

   6.1. `top.vhd` (takes the longest time to accomplish):

      - adapt the entity-definition (ports) that are connected to the I/Os and further to the `ucf`

      - not to forget the I, O and I/O buffers → IBUFs resp. OBUFs remain simple, IOBUFs are extended to I, O and T (tristate)-line (e.g. for `Sda_decoder`: one pin to the outside, but `*_I`, `*_O` and `*_T` towards the entity `system_i` → very time and work consuming!

      - add/erase components that are (not) needed:
        → adapt component `system` ⇒ comes up to the `system` from the EDK project → can be copied from `<base>/synth/edk/hdl/system.vhd` (`system_stub.vhd` here is only a dummy to be able to instantiate `system.vhd` as a sub-module in the ISE project)
        → guide the PRM signals as ports out in entity `system`

      - add/remove corresponding instances

      - care for intermediate signals if signals have to run directly from component instances to I/Os

      - insert bus macros for signals coming from/going to the PRM (here: `R,G` u. `B` to and from `rbgfilter`, `v_` and `h_counter` to and from `rgbfilter` and `valid` to and from `rgbfilter`
        → again, add intermediate signals

      - additional instances, that are to be static or reconfigurable later on have to be named → the definition will be done in the `ucf` (e.g. `recon_module: rgbfilter portmap( ....)`

6.2. `top.ucf`:

- place bus macros (PlanAhead or FPGA-Designer can pose a big help here)
- assign areas for static and reconfigurable modules ($\rightarrow$ in PlanAhead this can be done with drag-and-drop manner!)
- assign which instance from `top.vhd` is member of which group (static $\leftrightarrow$ reconfigurable)

7. The Build Process:

7.1. synthesize all files:

- if not done yet: synthesize the ISE project in
  `<base>/synth/edk/projnav`
  (`system_stub_bd.bmm`) may not be omitted!)
- execute `<base>/synth/synthesize_all.cmd`

7.2. begin with the *Early Access Partial Reconfiguration Design Flow*:
$\rightarrow$ execute `buildit.cmd` $\Rightarrow$ the full and partial bitstreams should be built if the script has been adapted properly.

Appendix B
Example of a User Constraints File

The following chapter gives an excerpt of a `User Constraints File` (UCF). It has been created for the partial reconfiguration of the *Video8* sample design. The complete file can be found on the book's Web site.

```
# area group for reconfig module
INST "recon_module" AREA_GROUP = "pblock_recon_module";
AREA_GROUP "pblock_recon_module" RANGE=MULT18X18_X3Y8:MULT18X18_X3Y11;
AREA_GROUP "pblock_recon_module" RANGE=RAMB16_X3Y8:RAMB16_X3Y11;
AREA_GROUP "pblock_recon_module" RANGE=SLICE_X30Y64:SLICE_X45Y95;
AREA_GROUP "pblock_recon_module" RANGE=TBUF_X30Y64:TBUF_X44Y95;
AREA_GROUP "pblock_recon_module" MODE=RECONFIG;



# INST "fixed_area" AREA_GROUP = "pblock_fixed_area";
INST "system_i" AREA_GROUP = "pblock_fixed_area";
INST "alibi" AREA_GROUP = "pblock_fixed_area";
INST "dcm_25_MHZ" AREA_GROUP = "pblock_fixed_area";

AREA_GROUP "pblock_fixed_area" RANGE=SLICE_X46Y16:SLICE_X87Y149;


# Clock and reset constraints
Net sys_clk_pin LOC=AJ15;
Net sys_clk_pin PERIOD = 10000 ps;

Net reset LOC=AG5 | IOSTANDARD = LVTTL; # Center
Net reset TIG;

Net sys_rst_pin LOC=AH5;
Net sys_rst_pin IOSTANDARD = LVTTL;
```

```
INST "dcm_25_MHZ" loc="DCM_X2Y0";



# leds
net LED_0 loc = AC4 | IOSTANDARD = LVTTL;
net LED_1 loc = AC3 | IOSTANDARD = LVTTL;
net LED_2 loc = AA6  | IOSTANDARD = LVTTL;
net LED_3 loc = AA5  | IOSTANDARD = LVTTL;



# push buttons
net button_start loc = AH2 | IOSTANDARD = LVTTL; # Right
net button_stop loc = AH1 | IOSTANDARD = LVTTL; # Left



#### Diligent general purpose expansion port (where the VDEC1 is
connected)
NET "YCrCb_in*  LOC = "AA8" | IOSTANDARD = LVTTL ;
.
.
.


NET "LLC_CLOCK"  LOC = "B16" | IOSTANDARD = LVCMOS25 ;
NET "RESET_VDEC1_Z"  LOC = "AF6" | IOSTANDARD = LVTTL  | DRIVE = 8 ;
NET "VDEC1_PWRDN_Z"  LOC = "G1" | IOSTANDARD = LVTTL   | DRIVE = 8 ;
NET "VDEC1_OE_Z"  LOC = "AF3" | IOSTANDARD = LVTTL   | DRIVE = 8 ;
NET "Sda_decoder_pin"  LOC = "AE5" | IOSTANDARD = LVTTL   | DRIVE = 8 |
PULLUP = TRUE;
NET "Scl_decoder_pin"  LOC = "AB8" | IOSTANDARD = LVTTL   | DRIVE = 8 |
PULLUP = TRUE;



#### VGA
NET "BLANK_Z" LOC = A8 | DRIVE = 12 | SLEW = SLOW | IOSTANDARD
= LVTTL;
NET "COMP_SYNC" LOC = G12 | DRIVE = 12 | SLEW = SLOW | IOSTANDARD
= LVTTL;
NET "H_SYNC_Z" LOC = B8 | DRIVE = 12 | SLEW = SLOW | IOSTANDARD
= LVTTL;
NET "V_SYNC_Z" LOC = D11 | DRIVE = 12 | SLEW = SLOW | IOSTANDARD
= LVTTL;
NET "PIXEL_CLOCK" LOC = H12 | DRIVE = 12 | SLEW = SLOW | IOSTANDARD
= LVTTL;

INST "red_out_DAC*
INST "green_out_DAC*
INST "blue_out_DAC*
```

```
#### Bus Macros
INST "BM_1" LOC = SLICE_X44Y94;
INST "BM_3" LOC = SLICE_X44Y92;
INST "BM_2" LOC = SLICE_X44Y90;
INST "BM_4" LOC = SLICE_X44Y88;
INST "BM_5" LOC = SLICE_X44Y86;
INST "BM_6" LOC = SLICE_X44Y84;
INST "BM_7" LOC = SLICE_X44Y82;
INST "BM_8" LOC = SLICE_X44Y80;
INST "BM_9" LOC = SLICE_X44Y78;
INST "BM_10" LOC = SLICE_X44Y76;
INST "BM_11" LOC = SLICE_X44Y74;
INST "BM_12" LOC = SLICE_X44Y72;
INST "BM_13" LOC = SLICE_X44Y70;
INST "BM_14" LOC = SLICE_X44Y68;
INST "BM_1" AREA_GROUP = "AG_BM_1";
INST "BM_2" AREA_GROUP = "AG_BM_2";
INST "BM_3" AREA_GROUP = "AG_BM_3";
INST "BM_4" AREA_GROUP = "AG_BM_4";
INST "BM_5" AREA_GROUP = "AG_BM_5";
INST "BM_6" AREA_GROUP = "AG_BM_6";
INST "BM_7" AREA_GROUP = "AG_BM_7";
INST "BM_8" AREA_GROUP = "AG_BM_8";
INST "BM_9" AREA_GROUP = "AG_BM_9";
INST "BM_10" AREA_GROUP = "AG_BM_10";
INST "BM_11" AREA_GROUP = "AG_BM_11";
INST "BM_12" AREA_GROUP = "AG_BM_12";
INST "BM_13" AREA_GROUP = "AG_BM_13";
INST "BM_14" AREA_GROUP = "AG_BM_14";


#### Module RS232_Uart_1 constraints
Net fpga_0_RS232_Uart_1_RX_pin LOC=AJ8;
Net fpga_0_RS232_Uart_1_TX_pin LOC=AE7;


#### Module SysACE_CompactFlash constraints
Net fpga_0_SysACE_CompactFlash_SysACE_*
.
.
.


#### Module DDR_256MB_32MX64_rank1_row13_col10_cl2_5 constraints
Net fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_*
.
.
.
```

Appendix C
Quick Part-Y Tutorial

This quick tutorial gives an introduction to the basic features of the Part-Y [71] tool developed by Florian Dittmann from the 'Heinz Nixdorf Institute' at the University of Paderborn.

It is intended to guide a novice to Part-Y and partial reconfiguration through a first example of a PR design. The targeted system was the Xilinx XUP Development Board with a Virtex-II Pro FPGA.

- Prerequisites:
  - running version of Part-Y (this tutorial was created with version 1.2)
  - the tutorial_sources.zip file
  - ISE 6.1 or higher installed (tutorial is known to work on 7.1.04i)
- Notes:
  - always hit OK before changing the tab in a view – changes will not be saved otherwise
  - StdOut will be directed to Miscellaneous‖Standard Output
  - you might have extended the Part-Y Project by the proper definition for your FPGA. These can be found under
    de.upb.cs.roichen.party.device.concrete
  - encountering problems you might find Part-Y_FAQ.txt useful
- Proceeding:
  1. Open the level configuration window and ensure that the following items from the different views are checked:
     - BehavioralModuleSelectionController
     - BehavioralTopLevelController
     - BehavioralDownloadController
     - StructureTopAssemblyController
     - StructureTreeLevelController
     - GeoSystemLevelController

- MiscStorageController
- MiscGenerateBitstreamsController
- MiscOutputController

2. In Module Selection import, the modules rt1.vhd, rt2.vhd and fixed.vhd from tuto-rial_sources.zip. Add them to current modules and mark each of them as reconfig-urable. Hit OK!

3. In Top, import global.vhd and global2.vhd. Add global.ucf to each of the Top-Level-Designs marking one at a time and adding the .ucf separately. OK!

4. Top Assembly: mark Top0_global and Top1_global2 each at a time and add every module to both of them. You should see the added modules at the 'modules instances of selected top'-window.

5. If everything worked out up to now, you will see a hierarchy like the one exposed in figure [C.1] on page 350. in Structural View|Tree.
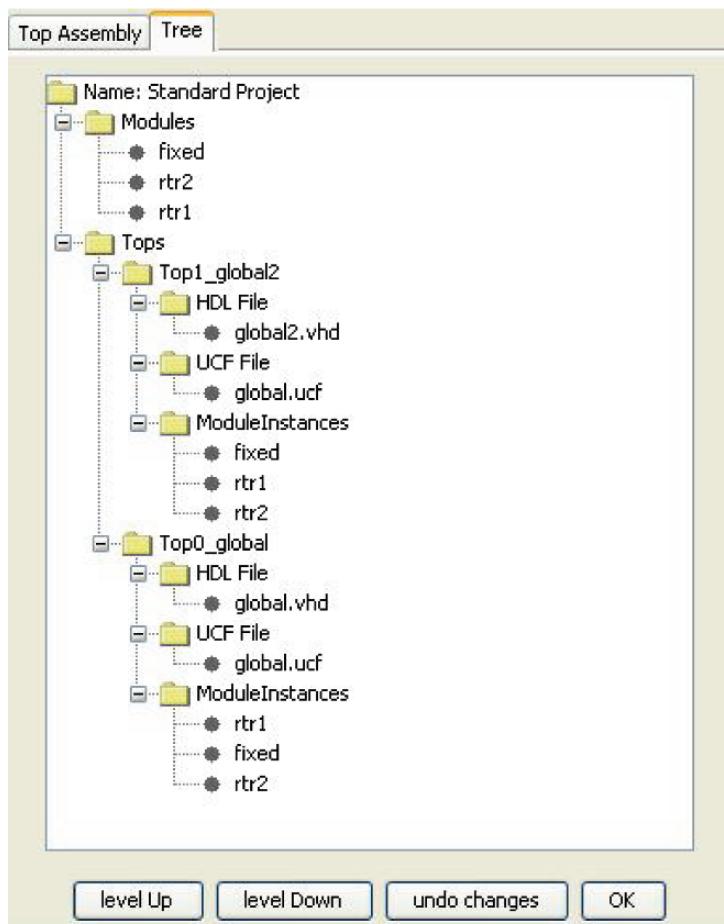
Figure C.1. Tree View after Top Assembly

6. In Geometrical View|System: set the correct target platform and a proper bus macro file.

7. Miscellaneous|Storage: set a name for the Part-Y project and a project path. Clicking OK will create a file hierarchy at the given spot, INTEGRATE will copy the vhdl files and others to the correct places. Be sure not to use spaces in path names because Part-Y as well as ISE cannot cope with them properly. If everything went right, there should be a bunch of subfolders in your specified directory containing copies of the vhdl files you declared and a couple of additional files.

8. In Behavioral View|Synthesis, all of the VHDL-files have to be synthesized.

9. With Miscellaneous|Bitstream Generation, you can conduct the initialization phase, the activation as well as the final assembly phase by clicking on the run buttons. For some reason, this will not look pressed when they are – might be a bug in the GUI. You can check if the action is taking place in the 'standard output' window.

10. After everything has been accomplished, the bitstreams can be downloaded to the device with Behavioral|Download using iMPACT.

Index