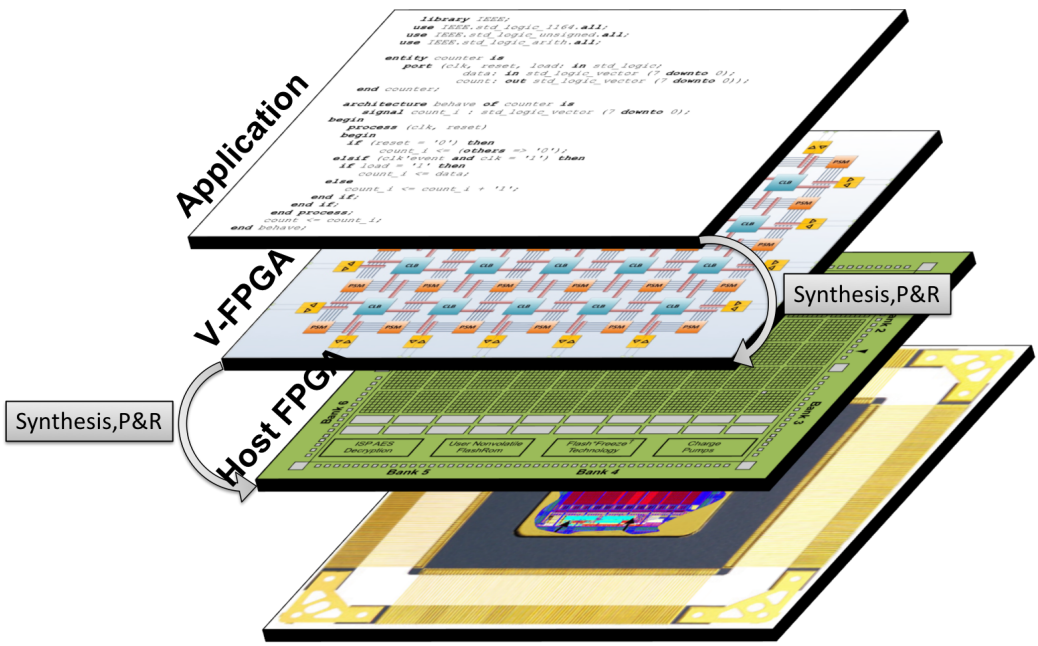
The Customizable Virtual FPGA:
Generation, System Integration and Configuration of Application-Specific Heterogeneous FPGA Architectures

Răzvan Peter Figuli



The Customizable Virtual FPGA: Generation, System Integration and Configuration of Application-Specific Heterogeneous FPGA Architectures

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS (Dr.-Ing.)

von der Fakultät für
Elektrotechnik und Informationstechnik
des Karlsruher Institut für Technologie (KIT)
genehmigte

DISSERTATION

von

Dipl.-Ing. Razvan Peter Figuli

geboren in Hermannstadt (Rumänien)

Tag der mündlichen Prüfung:
21.09.2017

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr.-Ing. Klaus Hofmann

Abstract

During the past three decades the evolution of Field Programmable Gate Arrays (FPGAs) has been strongly influenced by Moore's Law, process technology (scaling) and commercial markets. State-of-the-art FPGAs are moving closer and closer to general purpose on the one hand, but on the other hand, now that FPGAs have superseded more and more traditional Application-Specific Integrated Circuit (ASIC) domains, the efficiency expectations are growing. With the end of Dennard scaling, efficiency improvements can no longer rely on technology scaling alone. These facets along with trends towards reconfigurable System-on-Chips (SoCs) and new low-power applications such as cyber physical systems and internet of things require a better fit of target FPGAs, which can be facilitated through customization. Collaterally, trends towards mainstream deployment of FPGAs in day-to-day consumer products and services, the latter especially with the recent developments to employ FPGAs in data centres and cloud services, necessitate instant portability of applications across current and future FPGA devices. In this context, hardware virtualization can be a seamless vehicle for platform independence and portability. Candidly, purposes of customization and of virtualization are in a field of conflict as customization is intended for efficiency improvement yet virtualization adds additional area overhead. However, virtualization not only benefits from customization but also adds more flexibility to it as the architecture can be altered anytime. This peculiarity can be exercised for adaptive systems.

Both, customization and virtualization of FPGA architectures are predominantly unaddressed by the industry. Despite a few existing academic works they can be considered unexplored and are emerging areas of research.

The main goal of the work presented in this thesis is to expedite the generation of custom FPGA architectures that are tailored towards an efficient befitting of the application. In contrast to the usual approach with commercial off-the-shelf FPGAs, where the FPGA architecture is considered as a given constraint and the application is mapped onto the available resources, this work follows a new paradigm in which the application or application class is considered as given and the target architecture is tailored to efficiently accommodate the application. This results in customized application-specific FPGAs. The three pillars of this thesis are the aspects of *virtualization*, *customization* and the *framework*.

The central element is an extensively parameterizable virtual FPGA architecture, called *V-FPGA*, with the primary scope to be mapped onto any commercial off-the-shelf FPGA, while applications are executed on the virtual layer. This ensures portability and migration even on bitstream level as the specification of the virtual layer can persist, while the hosting physical platform can be exchanged. Furthermore, this technique is utilized to enable dynamic and partial reconfiguration on platforms that don't support it natively. Apart from virtualization, the *V-FPGA* architecture is further intended to serve as an embedded FPGA integrated into an ASIC, delivering efficient yet flexible system-on-chip

solutions. Therefore, target technology mapping methodologies for both, virtualization and physical implementation are addressed and an example for physical implementation in a 45 *nm* standard-cell approach is carried out.

The highly flexible *V-FPGA* architecture can be tuned by more than 20 parameters, including Lookup Table (LUT) size, clustering, 3D stacking, routing structure and many more. The effects of the parameters on area and performance of the architecture are studied and an extensive analysis of over 1400 benchmark runs reveals a high parameter sensitivity with variances up to $\pm 95.9\,\%$ in area and $\pm 78.1\,\%$ in performance, which proves the high significance of customization when it is up to efficiency. To systematically tune the parameters towards the application's needs, a parametric design space exploration methodology based on suitable area and delay models is proposed.

A challenge of custom architectures is their design effort and the need for custom tools. Therefore, this work comprises a framework for architecture generation, design space exploration, application mapping and evaluation. Above all, the *V-FPGA* is designed in a fully synthesizable generic Very High Speed Integrated Circuit Hardware Description Language (VHDL) code, being highly flexible yet eliminating the need for external code generators. System designers can benefit from different types of generic SoC architecture templates to reduce design time. All necessary design steps for application development and mapping onto *V-FPGA* are supported by a tool-flow for electronic design automation, that exploits a collection of existing commercial and academic tools, customized by suitable models and complemented by a new tool called *V-FPGA Explorer*. This new tool not only acts as back-end tool for application mapping onto the *V-FPGA* but is also a graphical configuration and layout editor, a bitstream generator, an architecture file generator for the place & route tools, a script generator and a testbench generator. A specialty is the support of just-in-time compilation with fast algorithms for in-system application mapping.

Finally, this thesis resolves with the closure of *V-FPGA* being so far employed in use-case applications in the fields of industrial process automation, medical imaging, adaptive systems and education.

Zusammenfassung

In den vergangenen drei Jahrzehnten wurde die Entwicklung von Field Programmable Gate Arrays (FPGAs) stark von Moore's Gesetz, Prozesstechnologie (Skalierung) und kommerziellen Märkten beeinflusst. State-of-the-Art FPGAs bewegen sich einerseits dem Allzweck näher, aber andererseits, da FPGAs immer mehr traditionelle Domänen der Anwendungsspezifischen integrierten Schaltungen (ASICs) ersetzt haben, steigen die Effizienzerwartungen. Mit dem Ende der Dennard-Skalierung können Effizienzsteigerungen nicht mehr auf Technologie-Skalierung allein zurückgreifen. Diese Facetten und Trends in Richtung rekonfigurierbarer System-on-Chips (SoCs) und neuen Low-Power-Anwendungen wie Cyber Physical Systems und Internet of Things erfordern eine bessere Anpassung der Ziel-FPGAs. Neben den Trends für den Mainstream-Einsatz von FPGAs in Produkten des täglichen Bedarfs und Services wird es vor allem bei den jüngsten Entwicklungen, FPGAs in Rechenzentren und Cloud-Services einzusetzen, notwendig sein, eine sofortige Portabilität von Applikationen über aktuelle und zukünftige FPGA-Geräte hinweg zu gewährleisten. In diesem Zusammenhang kann die Hardware-Virtualisierung ein nahtloses Mittel für Plattformunabhängigkeit und Portabilität sein. Ehrlich gesagt stehen die Zwecke der Anpassung und der Virtualisierung eigentlich in einem Konfliktfeld, da die Anpassung für die Effizienzsteigerung vorgesehen ist, während jedoch die Virtualisierung zusätzlichen Flächenaufwand hinzufügt. Die Virtualisierung profitiert aber nicht nur von der Anpassung, sondern fügt auch mehr Flexibilität hinzu, da die Architektur jederzeit verändert werden kann. Diese Besonderheit kann für adaptive Systeme ausgenutzt werden.

Sowohl die Anpassung als auch die Virtualisierung von FPGA-Architekturen wurden in der Industrie bisher kaum adressiert. Trotz einiger existierenden akademischen Werke können diese Techniken noch als unerforscht betrachtet werden und sind aufstrebende Forschungsgebiete.

Das Hauptziel dieser Arbeit ist die Generierung von FPGA-Architekturen, die auf eine effiziente Anpassung an die Applikation zugeschnitten sind. Im Gegensatz zum üblichen Ansatz mit kommerziellen FPGAs, bei denen die FPGA-Architektur als gegeben betrachtet wird und die Applikation auf die vorhandenen Ressourcen abgebildet wird, folgt diese Arbeit einem neuen Paradigma, in dem die Applikation oder Applikationsklasse fest steht und die Zielarchitektur auf die effiziente Anpassung an die Applikation zugeschnitten ist. Dies resultiert in angepassten anwendungsspezifischen FPGAs.

Die drei Säulen dieser Arbeit sind die Aspekte der Virtualisierung, der Anpassung und des Frameworks. Das zentrale Element ist eine weitgehend parametrierbare virtuelle FPGA-Architektur, die V-FPGA genannt wird, wobei sie als primäres Ziel auf jeden kommerziellen FPGA abgebildet werden kann, während Anwendungen auf der virtuellen Schicht ausgeführt werden. Dies sorgt für Portabilität und Migration auch auf Bitstream-Ebene, da die Spezifikation der virtuellen Schicht bestehen bleibt, während die physische

Plattform ausgetauscht werden kann. Darüber hinaus wird diese Technik genutzt, um eine dynamische und partielle Rekonfiguration auf Plattformen zu ermöglichen, die sie nicht nativ unterstützen. Neben der Virtualisierung soll die V-FPGA-Architektur auch als eingebettetes FPGA in ein ASIC integriert werden, das effiziente und dennoch flexible System-on-Chip-Lösungen bietet. Daher werden Zieltechnologie-Abbildungs-Methoden sowohl für Virtualisierung als auch für die physikalische Umsetzung adressiert und ein Beispiel für die physikalische Umsetzung in einem 45 *nm* Standardzellen Ansatz aufgezeigt.

Die hochflexible V-FPGA-Architektur kann mit mehr als 20 Parametern angepasst werden, darunter LUT-Grösse, Clustering, 3D-Stacking, Routing-Struktur und vieles mehr. Die Auswirkungen der Parameter auf Fläche und Leistung der Architektur werden untersucht und eine umfangreiche Analyse von über 1400 Benchmarkläufen zeigt eine hohe Parameterempfindlichkeit bei Abweichungen bis zu $\pm 95,9\%$ in der Fläche und $\pm 78,1\%$ in der Leistung, was die hohe Bedeutung von Anpassung für Effizienz aufzeigt. Um die Parameter systematisch an die Bedürfnisse der Applikation anzupassen, wird eine parametrische Entwurfsraum-Explorationsmethode auf der Basis geeigneter Flächen- und Zeitmodellen vorgeschlagen.

Eine Herausforderung von angepassten Architekturen ist der Entwurfsaufwand und die Notwendigkeit für angepasste Werkzeuge. Daher umfasst diese Arbeit ein Framework für die Architekturgenerierung, die Entwurfsraumexploration, die Anwendungsabbildung und die Evaluation. Vor allem ist der V-FPGA in einem vollständig synthetisierbaren generischen Very High Speed Integrated Circuit Hardware Description Language (VHDL) Code konzipiert, der sehr flexibel ist und die Notwendigkeit für externe Codegeneratoren eliminiert. Systementwickler können von verschiedenen Arten von generischen SoC-Architekturvorlagen profitieren, um die Entwicklungszeit zu reduzieren. Alle notwendigen Konstruktionsschritte für die Applikationsentwicklung und -abbildung auf den V-FPGA werden durch einen Tool-Flow für Entwurfsautomatisierung unterstützt, der eine Sammlung von vorhandenen kommerziellen und akademischen Werkzeugen ausnutzt, die durch geeignete Modelle angepasst und durch ein neues Werkzeug namens V-FPGA-Explorer ergänzt werden. Dieses neue Tool fungiert nicht nur als Back-End-Tool für die Anwendungsabbildung auf dem V-FPGA sondern ist auch ein grafischer Konfigurations- und Layout-Editor, ein Bitstream-Generator, ein Architekturdatei-Generator für die Place & Route Tools, ein Script-Generator und ein Testbenchgenerator. Eine Besonderheit ist die Unterstützung der Just-in-Time-Kompilierung mit schnellen Algorithmen für die In-System Anwendungsabbildung.

Die Arbeit schliesst mit einigen Anwendungsfällen aus den Bereichen industrielle Prozessautomatisierung, medizinische Bildgebung, adaptive Systeme und Lehre ab, in denen der *V-FPGA* eingesetzt wird.

Preface

The presented work has been conceived during my time as research associate / PhD student at the Institute for Information Processing Technologies (ITIV) at the Karlsruhe Institute of Technology (KIT) from February 2011 to January 2017. Initially, my main motivation to pursue a PhD was to take the challenge and exceed my own boundaries. Indeed, during my time at the KIT I have grown a lot in my capabilities and gained invaluable experiences, not only in my actual research but also in teaching, project planning, writing project proposals, project management, scientific reviews, publishing, networking, collaborations and international affairs. Furthermore I was able to travel to new places for conferences, workshops and collaborative projects and to meet new people. Above all, the greatest achievement in my life happened during my time at KIT when I met my now wife Shalina and married her. So I can truly say that the time at KIT had a significant impact on my life and I have plenty of reasons to be grateful.

First and above all I am thanking God Almighty for all His goodness, grace and mercy upon my life, without Whom I couldn't have reached this position in my life. He has been my source of strength to Whom I always looked to. He carried and lifted me all throughout my good days and desperate days. Time and again He showed His sovereignty and faithfulness to me and proved that He has the final say over everything.

I want to express my sincere gratitude towards my supervisor Prof. Dr-Ing. Dr. h. c. Jürgen Becker for enabling me to pursue a PhD at his Institute, for supporting me in all possible ways and for his guidance even during my tough times. Especially I appreciate his trust on me from the very beginning and that he granted me the freedom to explore the research fields that I like and to develop therein.

I am thanking Prof. Dr.-Ing. Klaus Hofmann for being the second referee, for all his support and for the very interesting talks that we had during the preparation of the PARFAIT project proposal. I really enjoyed the fruitful time we had spent as a team together with Tillmann Krauss.

Special thanks go to Prof. Dr.-Ing. Michael Hübner who not only injected the idea of virtual FPGAs in me but also persistently urged me to pursue a PhD. Even though I refused initially as it was not part of my plans, his perseverance was like a call and made me to rethink it seriously and to chose this path. Sometimes we need destiny helpers in our lives and he was definitely my destiny helper in this regard.

I want to thank Prof. Dimitrios Soudris, Prof. Kostas Siozios and Dr. Charalampos (Harry) Sidiropoulos from the National Technical University of Athens (NTUA) for the great and very fruitful collaboration that we had. The CAD tools from NTUA and the V-FPGA architecture from KIT suddenly fitted perfectly together and built a symbiosis that enabled "crazy" research ideas and resulted in numerous joint publications in reputed conferences and journals and a "best paper" award.

Furthermore I would like to thank all the students who contributed to the implementation of the *V-FPGA* either as part of their thesis or through HiWi jobs. Thomas Bruckschlögl contributed with the dynamic defragmentation strategy and the implementation of the Snapshot functionality. Michael Mechler was involved in the implementation of the clustering and the 3D extension. Marc Bo Hartmann helped with the tile based structure. Weiqiao Ding was involved in conducting experiments for design space exploration, in integrating the clustering with the tile based structure and in the implementation of the fractional input MUX.

I'm so much grateful for all the love, care, support and sacrifices of my parents Georg and Ileana Figuli. They have been my constant source of comfort and encouragement at all times even when the sun didn't shine through. My special thanks goes to my brother's family Daniel, Roxana and Sofia Figuli for their continual advice, support and help all throughout my PhD career. My niece Sofia always brought joy and warmth to my heart that made me to press on.

I have not enough words to express my thanks to my wife Shalina. She was a great source of encouragement and took me out even when I was drawn in desperation. She stood with me side by side through all the tough times and through sleepless nights. She believed in me when I did not. She supported me in all possible ways a wife can do. In all situations she showed me her love and I love her so much. My warm thanks extend to also my parents-in-law George and Vasantha Ford for upholding me with their love and prayers. I'm also grateful to my sister-in-law's family Suvitha and Alfred for being there for me though they are far away.

Last but not least I'm indebted to all my friends and relatives who made my journey more beautiful, enjoyable and lively. It won't be complete if I don't thank all those who persevered with me through their prayers.

Karlsruhe, July 2017
Peter Figuli

Contents

1. Introduction

Field Programmable Gate Arrays (FPGAs) as the prime example of reconfigurable architectures are no longer just prototyping and research devices, but are meanwhile experiencing mass-deployment in products and applications. In their infancy they were almost abandoned due to their area inefficiency compared to ASICs, yet now their density is high enough to accommodate very complex circuits and the flexibility that they offer is so sweet and compelling that they have superseded ASICs in many domains despite the ASIC gap. Even though FPGAs are a platform for implementing application specific digital circuits, they are actually general purpose off-the-shelf devices. As things get more ambitious one might realize that *"one size fits all"* probably fits none really well.
This thesis goes one step further and advocates a new paradigm of application or domain specific FPGAs. As such, the main focus is the customization of FPGA architectures towards the needs and peculiarities of the applications that they are going to implement. After a reflection of the trends in reconfigurable architectures (Section 1.1), the *raison d'être* for this new paradigm is deduced in Section 1.2.
A second emphasis in this thesis is laid on virtual FPGAs, which add a new dimension of flexibility and are motivated in Section 1.3. Thereby, to be upfront with it, virtualization is not a must for a custom FPGA and customization is not a must for a virtual FPGA. However, the area hungry virtualization benefits from customization. In return, virtualization is a vehicle for prototyping, exploring and exchanging custom architectures and can ensure seamless and instant portability across diverse custom FPGAs or commercial off-the-shelf FPGAs.

1.1. Trends in Reconfigurable Architectures

To understand the trends and to anticipate the future it is always wise to reflect first the past. Reconfigurable architectures in context of Field Programmable Gate Arrays (FPGAs) have to date a history of three decades and their evolution has been mainly driven by Moore's Law [76], CAD tools, economic aspects, commercial exploitation, application fields and the entering into new markets. In [102] Trimberger divides the evolution of FPGAs into three epochs: the age of invention, the age of expansion, and the age of accumulation, which are summarized from Trimberger's paper and further references in the following.

Age of invention (AD 1984-1991): The first commercial FPGA, the Xilinx XC2064, was invented in 1984 by Bill Carter [78]. It featured 64 logic blocks with each one 3-input LUT and one register, and 58 I/O blocks. The FPGA was designed without CAD tools in a modified CMOS logic style with around 85000 transistors by repetitive use of a modular

CLB and a modular I/O block. The XC2064 dies were manufactured in a 2.5 μm process technology with two metal layers on a Seiko fab and the die size measured around 300 mil x 300 mil. Consequently the first FPGA with only 64 logic cells suffered greatly area inefficiency and the large die was prone to low yield and high costs per chip. Furthermore, the programming bitstream needed to be stored on an external memory chip (initially PROM) and loaded during startup. This means that functionality per dollar and space was rather low and it was not competitive to the alternatives. This focussed the attention towards increasing area efficiency as a high priority in further developments and process technology.
Actel was able to make a big step in area efficiency by introducing the anti-fuse technology for programming the FPGA, eliminating the need for configuration memory cells and external PROM. However, this came at the expenses of one-time programmability.
Even though it turned out early that a granularity of 4 inputs per LUT provides a good trade-off between area and delay, in practical applications they suffered from low utilization ratios (e.g. due to unused inputs and configurations), wasting area. Consequently this set off a wave towards very fine-grained logic cell architectures with two-input function generators ([5], [40], [77]) or even at the granularity of individual transistors with programmable interconnections through anti-fuse [66].
In contrast to PALs, interconnection architectures in FPGAs followed 2D topologies with short interconnect segments between adjacent blocks and pass transistor switches, yielding a more efficient wiring. However, the pass transistor networks led to increased signal delay due to their accumulated series resistance and large capacitance. Furthermore, since interconnect networks consumed area which is not accountable for logic, they were integrated very stingily by the FPGA architects, leading to routing congestions and challenges in utilization of the FPGA resources.

Age of expansion (AD 1992-1999): The age of expansion began with an aggressive addition of interconnects to FPGAs due to the launch of chemical-mechanical polishing (CMP), which made the birth of stacked metal layers possible thereby trailing certain significant effects in the era of expansion. Area was no longer the precious commodity as it could be traded off for performance, features and ease-of-use. On one hand the immense growth in capacity with decrease in costs propelled large FPGA applications whereby manual automation was no longer in discussion, paving way for automated synthesis, placement and routing as part and parcel of the design process. On the other hand, upgradation in process scaling as well as the cheaper metal usage made automated placement more precise. It also promoted performance to a great extent by its longer segmented interconnects making physically distant logics logically closer and eliminating exact alignment of logic cells, which are done for the sake of higher performance. Opposing to the payoff is the wastage of unused parts of the metal wire even though it is in the acceptable range. This era put an end to logic-waste-based fine-grained architectures, brought efficient architectures which are based on starving interconnects to an end and doomed time-multiplexed devices. The only thriving companies were Altera, AT&T and Xilinx. The surge in Moore's law pressurized the FPGA vendors to take possession of the leading-edge process technology to keep up the balance between doubling capacity and halving costs, throwing companies with technologies such as EPROM, Flash and antifuse into chaos due to their technology dependent architectures. Though in the mid-1990s LUT-

based architectures were "synthesis unfriendly", they survived and started to dominate once its simplicity in mapping, efficient layout in silicon as memories and interconnect saving capabilities have been explored and exploited persuading companies like Xilinx, Altera and AT&T/Lucent to adopt LUT architectures along with distributed-memory-cell architectures. The latter due to its architectural freedom and its proficiency in giving the FPGA vendors a nearly universal access to process technology. During the age of expansion, the FPGA's capacity growth started to address majority of ASIC applications and fought its way through EDA and ASIC technologies.

Age of accumulation (AD 2000-2007): The birth of new millennium brought FPGAs a huge market in the data communications industry. Their high-throughput, real-time computation and generality earned them a wide range of applications including control and automotive systems. Though hiking capacity growth escalated market growth, increased product cost started to weigh down the customers. This conflicting challenge was tackled by producing lower-capacity, lower-performance "low-cost" FPGAs for low-end markets and apprising the high-end markets with libraries of soft logic (IP) for important functions like microprocessors (Xilinx MicroBlaze and Altera Nios), memory controllers and various communication protocol stacks. For example, Ethernet MAC was implemented as soft core in Virtex-II before it was available as transistors in Virtex-4. FPGAs were not only expected to increase the logic size but were anticipated by the users to adhere to their system standards by keeping the cost and power efficiency under check, thereby compelled the inclusion of logic blocks such as large memories, microprocessors, multipliers, flexible I/O and source-synchronous transceivers. Built from custom-designed transistors made them often more efficient than ASIC implementations. The resultant "Platform FPGA" was a collection of programmable logic, soft core IPs and logic blocks. Especially the launch of logic generators like System Generator from Xilinx and DSP Builder from Altera assisted the users to build their own functionality from the available dedicated functions and soft logic. Embedded (System) Design Kits made possible to have Linux OS running on FPGA processor. Finally, the age of accumulation came to an end with Moore's law being under a subject of discussion as performance came to a slowdown in trade-off with power even though improvements in cost and capacity continued to step-up.

Today one of the major technological trends in reconfigrable computing is the formation of programmable SoC hybrids combining programmable logic, microprocessors, on-chip networks and interfaces in a single device. The integration of two or more circuit classes (e.g. FPGA, processor, dedicated HW logic, etc.), that classically have been treated and interconnected as separate chips, has tremendous benefits in terms of efficiency. For instance, on chip interconnects require less area, are more performant and allow a much higher complexity than off-chip interconnects, meanwhile reducing the pin count and the silicon- and power-hungry IO buffers. Recently, major FPGA vendors have realized this potential and are showing serious efforts towards heterogeneous SoC platforms [110] [73] [52].
With respect to new workloads, we can see a hype of extending the classical application fields of FPGAs by machine learning, artificial intelligence, search engine indexing, encryption and data compression [47]. We are currently witnessing a new major

breakthrough of FPGAs by employing them in data centers and cloud farms to accelerate workloads. Prominent projects are, e.g. Intel Xeon+FPGA Integrated Platform [45] or Microsoft's Catapult project [83]. Recently, Amazon started employing FPGAs in cloud farms, where integrated FPGA instances can be rented to accelerate applications in the cloud [9].
With the rising complexity and the demand for faster time to market high-level synthesis and growing IP libraries are the most obvious trends in EDA for FPGAs. This is especially essential for entering into markets that are dominated by CPU or GPU devices and where large pools of developers are trained for and used to utilize higher level programming languages and models. But also in terms of CPU-FPGA hybrid solutions the process of outsourcing of CPU workloads to FPGAs, preferably during runtime, benefits from abstraction layers that hide the underlying hardware. One of the most recent trends is the support of OpenCL in this direction and Altera (now part of Intel) and Xilinx put lots of efforts in it.

1.2. Need for Specialization and Customization

While devices like the Xilinx ZYNQ or Altera Cyclone V SoC are getting closer to general purpose computing, there is also an increasing need for specialized devices for high volume, low cost and low power applications, e.g. in the fields of mobile devices, Cyber-Physical System (CPS), Internet of Things (IoT), sensor networks, wearables and others. The tight power budgets of such devices are hard to meet by general purpose FPGAs. Wireless devices need to operate for a long time from battery, some need even to be self-sustained through energy harvesting. Even wired devices can have tight power limitations. For instance, intelligent sensor nodes that use the same one wire for power supply and for Highway Addressable Remote Transducer (HART) communication have a current limit of <4mA for their internal electronics in order to not disturb the 4-20 mA current-modulated communication which is on the same wire. Apart from power also form-factor, size and price play an important role, which are hard to meet with current general purpose Commercial off-the-Shelf (COTS) FPGAs. If FPGAs are to supersede ASICs in these domains, then they need more specialization.

With respect to high performance computing, the recent embracement of FPGAs by large scale contemporary and upcoming markets (Datacenter, Cloud, Ai, Security) presumably sets the volumes high enough to afford specialized FPGAs for certain application classes. According to [47] it is unlikely that FPGAs employed in data centers will take the role of general purpose computing. Instead, they are seen as a companion to CPUs accelerating certain workloads. Meanwhile, power matters more and more also in data centers, which according to [45] is a strong motivation to replace GPUs by FPGAs. With power, performance, hyper workload classes and the high volumes in mind, certainly specialization could pay out in these highly competitive markets, where chip giants are currently reshuffling the cards by stacking up with strategic acquisitions and fusion of technologies.

From a high level viewpoint, another more general need for specialization is coming from the limitations that the semiconductor industry is running against. In former decades Moore's law (originally formulated in 1965 in [76] and revised ten years later) predicted

the doubling of transistors per area every two years, which used to be a quite reliable self-fulfilling prophecy. Also the formulations of Dennard [29], which in essence conclude that power requirements are proportional to actual area, which in conjunction with Moore's law means that performance-per-watt would increase when transistor density increases, were valid in former decades. This way, the industry was able to deliver generation by generation more value to their customers and an increased efficiency by roughly 40% per generation through scaling alone [19].
Today, while Moore's law is threatened but probably still continues for some time, Dennard Scaling is already invalid. It came to an end roughly one decade ago because at feature sizes below 65 nm the exponential growth of leakage current is not neglect-able anymore and also threshold voltage is difficult to reduce further. This impedes the down-scaling of operating voltage and increases the power and heat density with shrinking feature sizes. The consequence is that scaling alone doesn't automatically lead to efficiency improvements anymore. Yet the market is used to get more and more value with every new generation, which is also the reason why they have been investing again and again in the latest technology. This puts now pressures on increasing the efficiency of integrated circuits through other means.
One option to achieve this is through specialization on architecture level, in order to obtain a better fit of the architecture to the application and consequently a higher efficiency. Naturally, specialization demands a high price in terms of effort, especially if things need to be developed from scratch. This can be considerably mitigated through generic architecture models (and corresponding tools) that can be customized to fit the application.

1.3. Virtualization from different viewpoints

Portability. FPGAs are gaining more and more popularity and their large scale deployment surely will make them more and more affordable as the NRE costs will be shared by more units. If this movement continues at this rate, FPGAs will become mainstream in the near future and indispensable in our day-to-day systems and applications such as entertainment, communication, assistance, automation, cyber-physical systems, Internet of Things (IoT) devices, cloud services, monitoring, controlling, and many more. There will be the situation that FPGA based devices and applications change more often than how it is today, thereby making it necessary to loosen the bond between application and the execution platform [34].
Virtualization is a key for instant portability and migration of applications even on bitstream level without redesigning or recompiling. Thereby, an optimized reconfigurable architecture as a virtual layer can be mapped onto an existing COTS FPGA, while the application itself will be executed on the virtual layer, thus being independent of the underlying physical platform. The eminent advantage is that the specification of the virtual architecture can persist, while the hosting physical platform can be exchanged by another one. Not only can this be used for task migration scenarios during runtime but is also a strong aspect for the industry to have a *"second source"* in order to overcome disruptive situations like the discontinuation of the employed COTS FPGA platform. In such cases, the same bitstream can be executed on another hosting COTS FPGA from another vendor through virtualization.

Partial and Dynamic Reconfiguration. The configuration mechanisms of a virtual FPGA can be decoupled from the configuration mechanisms of the underlying physical platform. This can be exploited to enable partial and dynamic reconfiguration on platforms that don't support it natively. This is a very powerful aspect of FPGA virtualization as it actually extends the capabilities and functionality of a COTS FPGA through the virtual layer. Dynamic partial reconfiguration in particular is a desirable feature as it allows to reuse parts of the chip area for temporal exclusive functions on demand [18] [103] [50] [35]. It virtually increases chip area compared to static solutions that need to accommodate all functions at the same time even though they are not needed simultaneously. Many COTS FPGAs, especially low power devices such as Actel's ProASIC3, Igloo, etc., don't offer dynamic reconfiguration. In [50] and [35] this limitation is canceled through virtualization.

Adaptivity. The added level of flexibility through virtualization enables to adapt the custom FPGA architecture to the workloads. Being mapped onto a programmable platform, the architecture can be altered any time and in case the underlying platform supports dynamic partial reconfiguration, this can happen during run time operation. The adaptivity can be exploited to tune the architectural parameters of a custom FPGA for an optimized befitting with respect to the application, leveraging the Reconfigurable Computing paradigm to new dimensions in space and time and following a new paradigm in which the application comes first and the architecture follows as opposed to the old off-the-shelf approaches.

Prototyping and Emulation. Virtualization serves also as a vehicle for prototyping custom physical FPGA architectures, be it stand-alone or embedded in systems, and associated runtime environments. The custom FPGA and other system(-on-chip) components along with monitoring and debug logic can be mapped on a COTS FPGA or a multi-FPGA platform to test the design. Since virtualization exploits the same degree of parallelization as the intended ASIC realization, this makes the testing and benchmarking of such complex systems much faster than through simulation on a PC, which is mainly sequential. Following the observations of [56] on the ASIC gap, we can presume this type of virtualization/prototyping to be only around 3.4 x to 4.6 x slower than a targeted standard-cell implementation of the custom FPGA architecture. One could take advantage on this to initially release an early fully-functional *"light"* variant of a new product with a COTS FPGA virtualizing the new reconfigurable system-on-chip architecture. Later, the product line can be updated once the new higher-performance chip is fabricated. This strategy not only reduces time-to-market of new products but can also help making the digital portion of the intended chip mature at the first manufacturing attempt, because already the experiences and customer feedbacks from the virtualized version can be used to improve the design early enough before chip manufacturing. Furthermore, the risk of the product is lower as the initial virtualized version can be altered through reconfiguration in case of faults. This helps to cope with infancy in the initial product phase.

Accessibility. Another benefit of virtualization is accessibility. Novel FPGA architectures can be accessed and made tangible through virtualization without the need of expensive

chip implementation. This way, researchers and students can conduct functional experiments on custom FPGA architectures that are accommodated by regular COTS FPGAs, which are usually present in any university. Same applies to educational workshops and training sessions. Furthermore, tools and application developers can start developing and debugging for custom FPGAs in parallel to the chip development and prior to production, as they have access to the new FPGA architecture through virtualization. This not only affects the applications to be mapped onto the new FPGA, but also the interplay with other system components, such as microprocessor, memory, network interfaces, etc..

1.4. Challenges and Proposed Solutions

Customization and virtualization come with a number of principle challenges that are systematically addressed by the work of this thesis:

1. The **design efforts** to create custom FPGA architectures and programmable SoCs are very high if everything needs to be developed from scratch. This might impede the acceptance of application specific custom FPGAs.
   -> Proposed solution: *The design efforts are minimized by generic architecture templates, that can be customized through parameters. This way system architects don't need to develop the FPGA architecture, but just change the parameters to obtain an optimized suitable architecture for the application. The more parameters the generic architecture offers, the better is the achievable fit.*
2. Architectural **design choices** can get rather complex due to interdependency of parameters and trade-offs. Especially the introduced freedom can be also a pain if the number of parameters is overwhelming.
   -> Proposed solution: *The process of choosing the right design parameters is assisted by model based parametric design space exploration.*
3. Custom FPGA architectures require **custom EDA tools** for application mapping and design space exploration.
   -> Proposed solution: *Flexible toolflow that seamlessly adapts to the customized architectures through parameterizable architecture models.*
4. Static analysis is not sufficient for **evaluating** dynamic systems with applications or benchmarks mapped onto the custom FPGA. Event or time driven simulation is able to do so but is too slow to observe a relevant time frame in acceptable tool runtime. Furthermore, the interplay with other system components that are not modelled is not possible in simulation.
   Proposed solution: *Prototyping through virtualization on a COTS FPGA exploits the same degree of parallelism as the target, thus is much faster than simulation. Existing external components can be interconnected and don't need to be modelled.*
5. Virtual FPGAs have the drawback of a high **area overhead** and also performance is degraded compared to physical FPGAs due to the additional layer.
   -> Proposed solution 1: *As sure as there is an ASIC gap for physical FPGAs, similarly the overhead for the virtual layer can not be eliminated. There is a price that has to be paid for the added flexibility and the benefits. However, this can be mitigated through customization.*

A better fit leads to a better utilization leads to less logic elements needed to accommodate the application. Since virtualization happens on top of a reconfigurable platform, it comes for free and can be updated on a per-application basis.
-> Proposed solution 2: *If there are temporal exclusive functions/applications that the virtual FPGA has to execute, the partial and dynamic reconfiguration feature of the custom virtual FPGA reduces the area requirements because not all functions need to be present at the same time. Thus they can be loaded on demand while others are discarded. In case the number of temporal exclusive functions is greater than the overhead factor, this technique compensates the area overhead.*

6. Due to dependency on underlying platform and coarse scaling quantization, virtual architectures don't follow the same area and performance patterns as their physical counterparts. Consequently **existing models** adopted from physical FPGAs are inaccurate for virtual FPGAs.
-> Proposed solution: *Introduction of new area and delay models based on minimum sized basic elements, taking into account the underlying platform.*

1.5. Contribution

Despite a few related works ([58], [65], [20]), the field of Virtual FPGAs is considered unexplored. Neither reconfigurable computing nor the idea of virtualization are new topics. However, to the best of my knowledge, the framework presented in this document incorporates the first concrete 3D Virtual FPGA that is able to execute applications.

Furthermore, the framework satisfies the different views on virtualization presented above.

Architecture wise there are a number of new innovations contributed, such as *Loopback-Propagation* for emulating bi-directional wires, *CoreFusion* to merge adjacent *V-FPGA* cores, slice-level dynamic and partial reconfiguration mechanisms, *Snapshot* mechanisms for fast dynamic migration of applications from one region to another during runtime, inclusion of *ViSA* cores as programmable heterogeneous blocks for efficient implementation of complex arithmetic functions or control flow.

In contrast to related works that have a rather fix architecture, the *V-FPGA* architecture presented in this thesis can be customized by more then 20 parameters.

For guiding the selection and adjustment of parameters towards objectives, a customization methodology is proposed, whereby starting with an analysis of the application, suitable architectural parameter values are determined through design space exploration.

The *V-FPGA* has a rather complete tool support for EDA of applications, architecture level DSE and customization. Existing solutions were focused on the exploration of hypothetical architectures at abstract level. Within this work a new tool called *V-FPGA Explorer* was developed that closes the gap between abstract layout and actual reconfiguration by importing the abstract layout information and mapping it onto the programmable resources and configuration mechanisms. Apart from bitstream generation it features graphical configuration editing, testbench generation, architecture file generation and tool control script generation.

In this work new area and delay models along with transferrable metrics suitable for virtual FPGAs are introduced. Prior works relied on borrowed models from physical FPGAs for DSE and area/timing driven application mapping. However, those transistor-level models were not suitable for virtual FPGAs as they have different base units. This gap is now closed with the new models presented in this thesis.

A detailed analysis of the effects of architectural parameters on area and performance is contributed in this thesis. Similar analysis were carried out for physical FPGAs by prior art related works, however this was missing for virtual FPGAs.

In addition, the transition from virtual to physical is considered and can be emulated in an early design stage. A physical design methodology with hierarchical layout is introduced for the *V-FPGA* in case a mapping onto standard-cell ASIC is targeted.

A unique type of educational support is provided by the TEAChER framework which is based on and utilizes the Virtual FPGA.

1.6. Outline

The rest of the thesis is organized as follows:

In chapter 2, apart from a definition of efficiency, the most important fundamental basics about reconfigurable architectures, partial and dynamic reconfiguration, 3D integration, electronic design automation steps for FPGAs are covered. This chapter is intended to provide a common understanding about the terms and topics throughout the thesis. Readers that are already confidently familiar with these topics can skip this chapter.

Chapter 3 surveys related works in context of architectural efforts for efficiency increase, custom embedded or embedded FPGAs, virtual FPGAs, generic CAD tools and 3D FPGA architectures and discusses briefly how the *V-FPGA* differentiates or complements these works.

Chapter 4 describes in detail the *V-FPGA* architecture which is the center piece of the framework. Thereby it includes also analysis of design parameters to justify customization of such parameters.

In chapter 5 the complete toolflow for mapping applications onto the *V-FPGA* considering that it is a custom architecture is described. Apart from existing customizable tools also the new *V-FPGA Explorer* tool is described which closes the gap between abstract layout and actual configuration.

Chapter 6 is devoted to the customization process itself by presenting generic SoC architecture templates and concepts and methodologies to determine the right parameters for a good fit of the application and architecture. Furthermore it includes new area related metrics targeted at the peculiarities of virtualization.

Chapter 7 covers the mapping of the *V-FPGA* onto various underlying platforms, including virtualization and physical implementation in a standard-cell approach. Closely related to the target technology mapping is the characterization which is needed for area and delay models. Therefore a characterization flow is also described in this chapter.

Chapter 8 presents various use cases with the *V-FPGA* employed.
Finally, in chapter 9 the conclusion of this work is summarized.

2. Fundamentals

2.1. Efficiency in Context of This Work

One of the most important optimization goals (but not the only) of this work is efficiency improvement. In [31] a to the point definition of the term "efficiency" as quoted in the following is given:

> Efficiency is the (often measurable) ability to avoid wasting materials, energy, efforts, money, and time in doing something or in producing a desired result. In a more general sense, it is the ability to do things well, successfully, and without waste. In more mathematical or scientific terms, it is a measure of the extent to which input is well used for an intended task or function (output). It often specifically comprises the capability of a specific application of effort to produce a specific outcome with a minimum amount or quantity of waste, expense, or unnecessary effort.

Efficiency can have different aspects. The following ones are relevant for this work:

Energy efficiency and the widely accepted synonym *performance per watt* (note that energy is power integrated over time) refers here to the ability to perform computations with low energy or power consumption. This is especially important for mobile and battery powered devices, however it has gained extreme importance also in high performance computing since Moore's Law began to run into the power wall leading to extreme power (and heat) densities in microprocessors nearly comparable to those of nuclear reactors [81].

Area efficiency refers to function density per area or performance per area. Often it is expressed in logic density or gate equivalents per die area, however this is not always a meaningful measure, since function density not only depends on the circuit complexity but also on the architecture type. An example of higher area efficiency with less logic density is presented in [36].

Cost efficiency as function of possible performance per chip cost (or per system cost) is related to area efficiency, but also depends on the technology node, integration, production volume and manufacturing process.

Mapping efficiency is used here to express how well an application can be mapped on the resources of an underlying platform in terms of resource utilization. The goal is to

utilize the allocated resources as much as possible. An example of an inefficient mapping is the realization of a 2-input NAND gate by a 6-input LUT: 67% of the LUT inputs and 95% of the LUT area remain unutilized and are wasted. Increasing the mapping efficiency can have also a positive effect on the area efficiency because the same application fits in a smaller area.

Throughout the rest of the thesis, if not otherwise specified, the term efficiency refers implicitly to one or more of the above mentioned aspects depending on the context that it is used in.

2.2. Reconfigurable Architectures

Reconfigurable architectures in context of this work are circuit architectures for semiconductor devices that can be configured by the user after the manufacturing process has been completed. The primary target is the realization of application specific integrated circuits, whereby in contrast to classical ASICs the flexibility and ability to alter the circuits is given and is the main differentiation. For this purpose reconfigurable architectures generally consist of arrays of flexible and programmable logic cells, programmable I/O cells as well as programmable interconnects. The purpose of each logic cell is to realize a logic function that has a limited amount of variables, i.e. a logic cell is a function generator. The programmable interconnects are there to combine logic cells in a way that the composite realizes a function of higher complexity out of several functions of little complexity. Furthermore, they route signals from the logic cells to the I/O cells for off-chip communication.
Reconfigurable architectures can be classified in at least two groups, fine grain and coarse grain, depending on the size and complexity of the programmable components. Compton and Hauck [26] distinguish the granularity by the bit size of the logic cells, i.e. a 3-input LUT would be fine-grained and a 6-input LUT coarse grained. Shannon [89] however distinguishes by bit level data manipulation and interconnect (fine-grained) vs. word-level data manipulation and interconnect (coarse-grained). Throughout this thesis the classification by Shannon is followed because the classification by Compton and Hauck suffers a blurred discrimination and would fail as soon as we talk about adaptive LUTs with variable size.

2.2.1. Fine Grain Reconfigurable Architectures

Field Programmable Gate Arrays fall in the category of fine grained reconfigurable architectures. The essence of FPGAs are programmable logic cells and programmable interconnect networks in a regular structure to realize digital circuits through individual programming of these resources. Apart from this, modern derivates contain also optimized macro-blocks of common functions, such as Block Random Access Memorys (BRAMs), Digital Signal Processing (DSP), I/O serializer/de-serializer. Fundamentally, the most important differentiators between existing FPGAs are the types of logic cells, the routing topologies for the interconnects and the programming technologies, which are explained in the following subsections.

2.2.1.1. General Topologies

The established fundamental topologies are island-style, row-based and sea-of-gates.

Island-style FPGAs are characterized by the fact that all logic cells are surrounded by routing channels (see Figure 2.1). Typically, in such architectures the inputs and outputs of the logic cells are distributed on all four sides and are connected to the tracks of routing channels via connection boxes. At the intersection points of the vertical and horizontal routing channels there are switching matrices that can interconnect tracks from different routing channels. This is the dominating topology of commercial FPGAs.

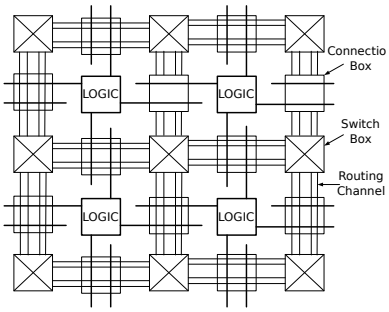


Figure 2.1.: Island-style FPGA topology [10]

Row-based. In row-based FPGAs, the logic cells are arranged in a row-shaped manner (see Figure 2.2). Between these rows are horizontal routing channels. The tracks are segmented in the routing channels and can also have different lengths. Furthermore, there are vertical tracks that pass through some logic cells and cross the horizontal routing channels. In this way, tracks from different rows can be connected to each other. Row-based architectures are used, for example, in Actel ACT3 FPGAs [72].

Sea-of-gates. In "sea-of-gates" based architectures (see Figure 2.3) there are no routing channels between the logic cells. Here, the routing is established by local connections between adjacent logic cells. The SX family of Actel for example, uses such a topology.

2.2.1.2. Logic Cells

The most common type of logic cells uses LUTs as function generators. This approach relies on precomputing a boolean function with a fixed number of variables under all possible combinations of values during the design time and storing the results in a table. If K is the number of variables, then the LUT needs to contain 2^K single bit storage elements for the results. The inputs of a LUT correspond to the function variables and drive the

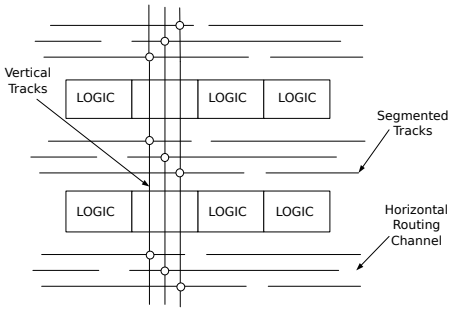


Figure 2.2.: Row-based FPGA topology [10]

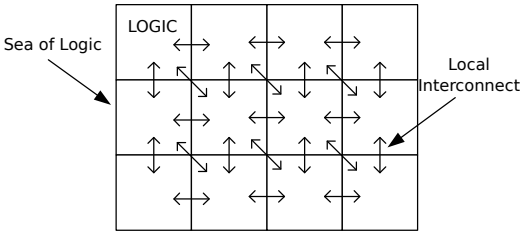


Figure 2.3.: Sea-of-gates FPGA topology [10]

select signals of a MUX, thus each value combination addresses one of the stored results to drive the output as depicted in Figure 2.4. With a K-input LUT there are 2^{2^K} possible functions realizable. Furthermore, logic cells usually contain also a flip-flop at the output of a LUT in order to realize sequential processes, while it can be bypassed if only combinational logic is to be implemented. Figure 2.5 shows the structure of a basic LUT based logic cell. Comercial versions of LUT-based logic cells can differ greatly from manufacturer to manufacturer. For instance, not only the number of inputs of a LUT can vary, but also a plurality of LUTs (and possibly flip-flops) can be clustered in different ways and, if necessary, locally interconnected, along with providing extra signals, both internally and externally, for realizing e.g. carry chains.

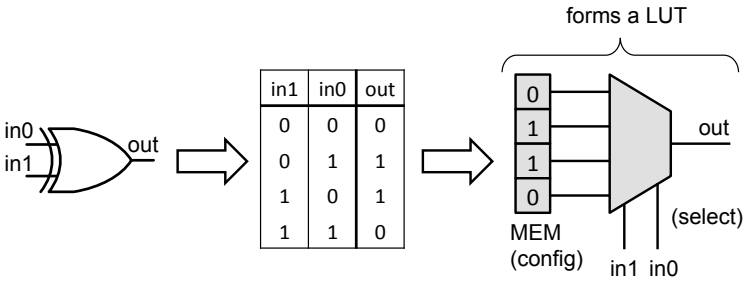


Figure 2.4.: Principle of lookup table (LUT) as programmable logic

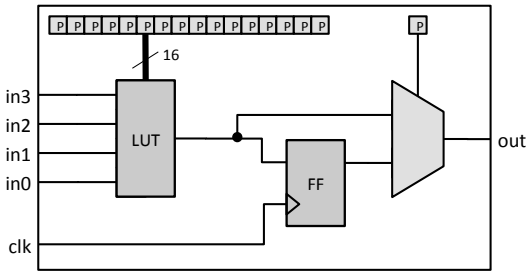
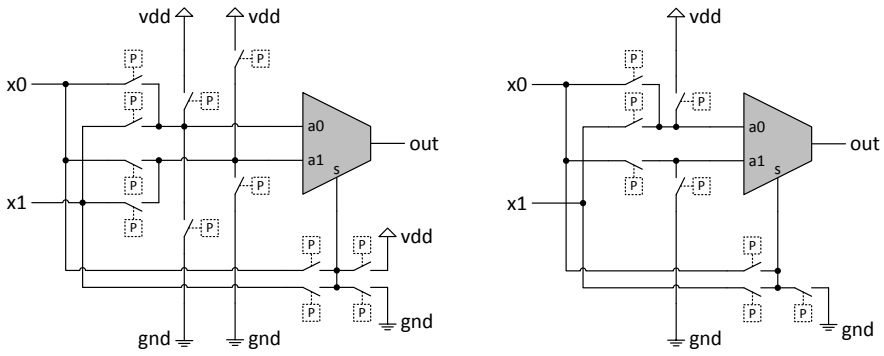


Figure 2.5.: Logic cell containing a LUT, a flip-flop and a bypass MUX, programmable through SRAM cells P

An alternative to LUT is multiplexer-based logic. The principle of MUX based logic is that a 2:1 MUX can generate the boolean functions listed in Table 2.1 if each input as well as the select signal can be tied by programmable switches to any of the function variables (*x0, x1*) or to *gnd* or to *vdd* according to Figure 2.6a. Having two MUX inputs (*a0, a1*) and one select signal *s*, there are $4^3 = 64$ possible combinations. However, some of the combinations lead to the same results and are redundant. Removing the redundant combinations, there remain the 11 possible functions from Table 2.1. The same redundancy leaves also room for minimizing the number of required switches, e.g. as shown in Figure 2.6b. For comparison, it is to mention that a 2-input LUT can realize 16 possible functions, i.e. MUX based logic misses 5 functions. However, with the functions presented in table 2.1 any other boolean function can be realized through combination of several MUX based logic cells.

Table 2.1.: Functions realizable with a 2:1 MUX

s	a1	a0	function
x0	gnd	x0	0
x1	gnd	x0	(NOT x1) AND x0
x1	gnd	vdd	NOT x1
x0	gnd	x1	(NOT x0) AND x1
x0	gnd	vdd	NOT x0
x1	gnd	x1	x0 AND x1
x0	x0	x0	x0
x1	x0	vdd	(NOT x1) OR (x1 AND x0)
gnd	gnd	x1	x1
x0	x0	x1	x1 OR x0
x0	x0	vdd	1



(a) fully flexible MUX based logic

(b) optimized MUX based logic

Figure 2.6.: Multiplexer based logic: (a) fully flexible (b) optimized. Both alternatives can implement the same function set (see Table 2.1)

2.2.2. Heterogeneous Reconfigurable Architectures

The need for heterogeneity arises from the fact that functions implemented on *specific-purpose logic* causes significant reduction in area when they are left unused in the target application, even though when utilized leads to higher performance. Thereby, the urge to have a heterogeneous mixture of *general-purpose* and *specific-purpose* logic blocks peaked in the last decade. To perceive the importance of heterogeneity, it is important to understand the terms "hard-core" or "hard circuit structure" and "soft core" or "soft logic fabric". Both of them are an array of combinational logic elements and each Logic Element (LE) consists of a logic function implemented as a gate of LUTs. The difference between hard-core and soft-core is being the fact that the latter one connects the LEs through programmable routing fabric, while in the former case the routing is made fixed in the fabric and cannot be reconfigured.

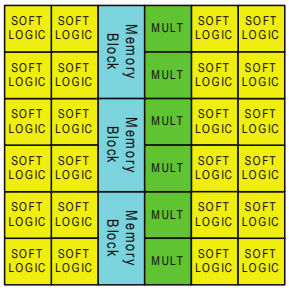


Figure 2.7.: An example of tile-based heterogeneity [57]

According to [57] there are two kinds of heterogeneity: *soft fabric heterogeneity* and *tile-based heterogeneity*. The name *soft fabric heterogeneity* by itself suggests that the FPGAs are constructed from an array of identical tiles, each containing the basic soft logic block and soft fabric heterogeneous elements. When distinct tiles, each having dedicated hard circuit structures, are included along with the soft logic blocks in the same substrate as illustrated in Figure 2.7, they fall into the latter category.

2.2.2.1. Soft Fabric Heterogeneous Elements

By definition, flip-flops are dedicated logic blocks and hence are hard circuit structures. Researches were carried out to emulate flip-flops as soft logic blocks, yet the significant impact on area efficiency has led to the inclusion of "hard" flip-flops within logic elements in most commercial FPGAs [1]. These are typically edge-triggered and include a variety of set, reset, load, enable and clocking capabilities, which depending upon the FPGA family can have fixed or programmable signal selection. Other explicit soft fabric elements are addition/subtraction/carry logic (carry lookahead and carry-skip) to make adder and subtraction units smaller and faster. Small memory units can also be built from LUTs which in turn can be connected together to form larger memories. [57]

2.2.2.2. Hard Structure Heterogeneous Elements

Block RAMS are one of the first hard-core heterogeneous tiles with the flexibility to have different aspect ratio configurations, which can be suited for different applications with varying memory requirements. With the addition of a small amount of soft logic, they can be combined to form larger memory blocks. These block RAMS can support dual functionality of reading and writing simultaneously and can be configured to have First-in First-out (FIFO) functionality. The unused memory blocks can be utilized efficiently by converting them into large LUTs. Multiplier is another such heterogeneous computation-oriented tile. When unused, they offer little benefit, which can be dealt with by creating

[1] One exception is Actel's *VersaTile* technology used in the ProASIC3 and Igloo devices [2], where a core cell can be configured either as a 3-input logic function or as a flip-flop or latch

sub-families within the FPGA family using the same basic architecture but with different ratio of hard and soft logic. In order to improve performance, microproccessors have been often made hard-core in spite of the challenges that arise in the interface layer between the processor, memory system and soft fabric. Though having them as soft-core means lower performance and larger area, they are customizable, which is very beneficial for applications with varying resource requirements. [57]

2.2.3. Programming Technologies

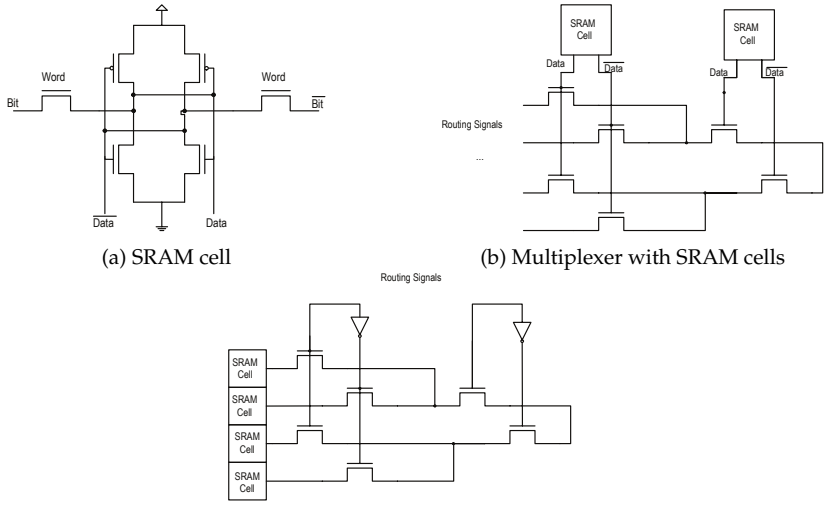
In FPGAs there have been mainly three relevant programming technologies established, that are commercially and successfully employed until today: SRAM, flash and anti-fuse.

SRAM. The static memory cells like the one shown in Figure 2.8a form the basis of SRAM programming technology. They are used to either interconnect signals by setting the select lines to multiplexers or to store data in LUTs as illustrated in Figure 2.8b and Figure 2.8c. Their re-programmability and their competence to make use of CMOS process technology made their mark in FPGAs and are employed by vendors like Xilinx and Altera. On the other hand, this approach is marked down because of the following demerits:

1. The SRAM cell along with the programmable element requires at least 6 to 7 transistors.
2. Its necessity for external storage devices during power down degrades the cost effectiveness of the FPGAs.
3. The on-resistances of pass transistors, which are used to implement multiplexers lead to capacitive load.

Flash. Figure 2.9 depicts the functionality of a flash memory cell. The programming transistor programs the floating gate (as it "floats" above the transistor) and the switching transistor acts as the programmable switch. Though the flash-based programming technology doesn't need to wait for the loading of configuration data, omits the usage of external devices for storing data and exhibits non-volatility and more area efficient than SRAM cells, it aids in area overhead because of the inclusion of high and low voltage buffers in programming circuitry, adds design complexity in switching transistor to keep up the source-drain voltage from being injected into the floating gate, suffers from high resistance and capacitance of transistor-based switches and finally the charge buildup in the oxide prevents them to be reprogrammed infinite number of times. The emerging trend of using flash cells in combination with SRAM cells provides non-volatility with infinite reconfigurability with the price being paid in area overhead.

Anti-fuse, an alternative non-volatile programming technology, can be implemented using either dielectric (Figure 2.10) or metal-to-metal (Figure 2.11) based approach. The former approach is largely replaced by the latter one, which sandwiches an insulating



(a) SRAM cell

(b) Multiplexer with SRAM cells

(c) Lookup table with SRAM cells

Figure 2.8.: SRAM programming technology [57]

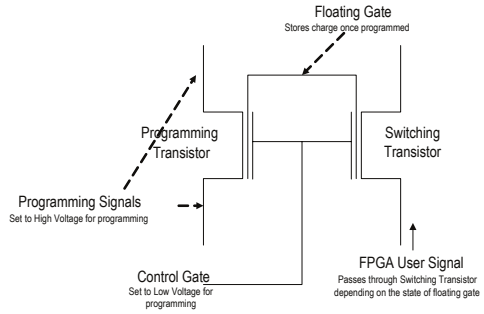


Figure 2.9.: Flash programming technology [57]

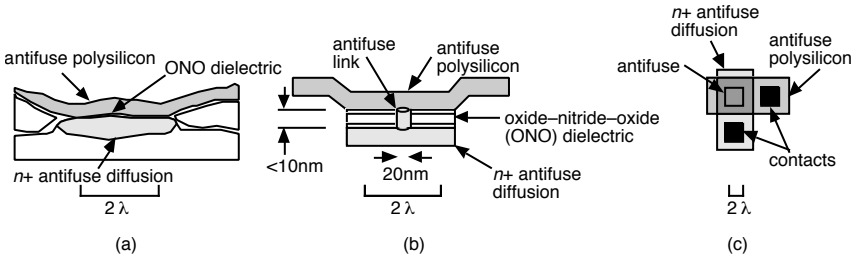


Figure 2.10.: Dielectric anti-fuse technology [11]

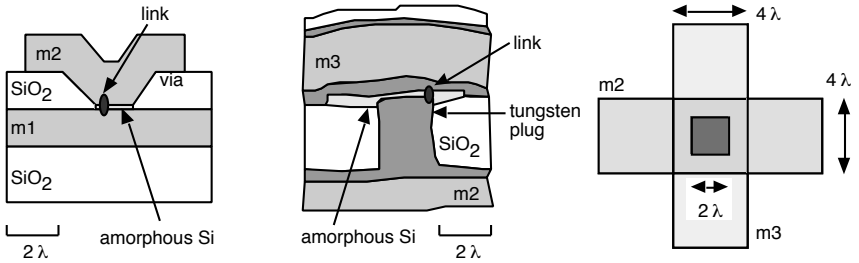


Figure 2.11.: Metal-to-metal anti-fuse technology [11]

material like amorphous silicon or silicon oxide between two metal layers and by applying a high voltage breaks down the anti-fuse and thereby establishes a conduct with a resistance of 20 to 100 ohms. Since the fuse itself doesn't require silicon area, the area overhead of programming circuitry is significantly reduced, which is slightly neutralized by the programming transistors as they need to deliver large currents to program the anti-fuse. Non-volatility reducing the system costs by eliminating external storage devices, low resistance and capacitance enabling inclusion of more switches per device and immediate operation after power up adds up to the advantage of anti-fuse technology. However the scalability of anti-fuses is in question as the most advanced devices use only 0.15 μm technology. Additionally, their permanent link establishment makes them invalid for applications where reconfigurability is highly essential.

2.3. Partial and Dynamic Reconfiguration

FPGAs with SRAM or Flash technology are reconfigurable. The implemented function of the circuit can be changed as often as required. To achieve this, earlier FPGAs had to be stopped and "reprogrammed" completely via a programming device. It is also said that such FPGAs are statically reconfigurable. This is still true today for most FPGAs. During the past decade, dynamic and partially reconfigurable architectures have been developed that can perform reconfiguration during runtime to a particular portion of the array, while the remaining portion can work uninterruptedly. In this way, functionality

can be exchanged at runtime and the array reused. Usually, different configurations are loaded into a non-volatile memory and are mapped onto the hardware on demand. This reactionary dynamic adaptivity of the hardware provides completely new possibilities. Therefore, this technique is currently the subject of intense research activities around the world. Dynamic and partial reconfiguration is explained in detail in [49].

2.4. 3D Integration

Chip stacking through 3D integration has the potential not only to reduce physical area but also to afford improvements in performance label. An overview of this new emerging 3D technology has been explained by W. Rhett Davis et.al. in [28]. In the following subsections, a walk-through of different 3D techniques has been briefed and compared to bring out the specialty of each technique. Wire bonded, microbump, through vias and contactless interconnection are the different 3D interconnect techniques available and their method of assembly, the maximum number of tiers (no. of chips in a stack) they can bond, the pitch of the vertical interconnect and their routing resource usage have been compared in Table 2.2. The technologies are described in the following subsections.

Table 2.2.: Comparison of 3D IC Technologies [28]

		Microbump		Through via		Contactless	
	Wire bonded	**3D package**	**Face-to-face**	**Bulk**	**SOI**	**Capacitive**	**Inductive**
Assembly level	Die	Die	Die	Wafer	Wafer	Die	Die
Tier limit	Assembly process	Heat	Assembly	Heat, yield	Heat, yield	Assembly process	Heat
Vertical pitch (μm)	35 to 100	25 to 50	10 to 100	50	5	50 to 200	50 to 150
Metal layers blocked by pad	All	Top 1 to 2	Top 1 to 2	All, top	All, top	Top	Top 1 to 2

2.4.1. Wire bonded

Each individual die in the stack are connected to other dies through wires as shown in Figure 2.12. Even though the wires need to run back and forth from each single die, they accomplish the purpose of connecting them together. This most commonly used approach goes out of bounds whenever the number of I/Os in the chip stack increases as they are limited in their wire resolution. The interconnect density is yet another factor that gets badly affected as the bonding is possible only on the peripheral of the chips. Also it is important to mention that in terms of routing resources, even though all the metal layers are used for bonding, it is likely that the devices underneath the pad get destroyed due to the mechanical stress and pressure caused by wire bonding.

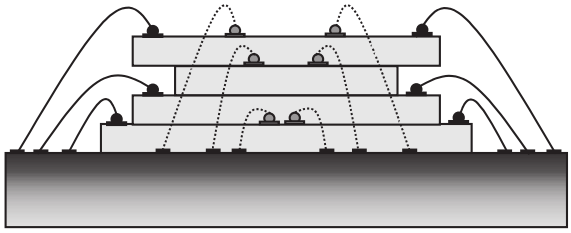
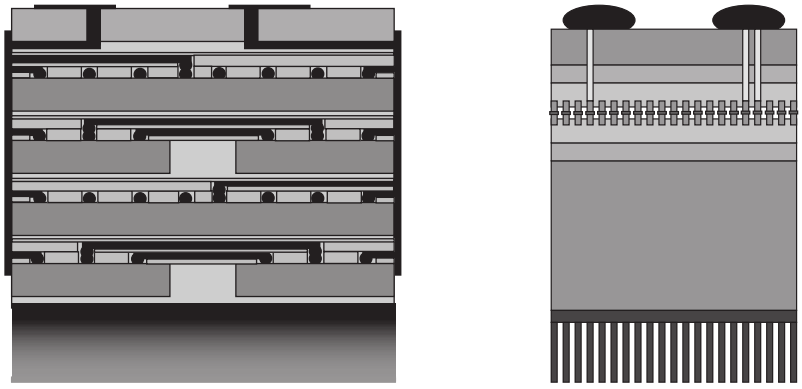


Figure 2.12.: Illustration of wire bonded 3D technology [28]

2.4.2. Microbump

Gold bumps or the use of solder on the dieÕs surface gets them the name "microbumps". The typical bump pitch is about 50 to 500 μm. Unlike wire bonded, microbump requires a maximum of two metal layers for bonding with minimum mechanical stress. The 3D package technology as shown in Figure 2.13a disburses greater interconnect density as they can embed fabricated dies into a set of carrier wafers enabling a much tighter cube structure. The process is as follows: each die-carrier tier is bonded to the epoxy routing tier through a layer of microbumps; the tiers are laminated and the routing tiers are then connected through metallization which is added to the sides of the cube. Since the microbumps are on the periphery of the tiers, the signals have to always pass through the periphery before reaching their destination inside the cube and thereby cannot significantly impact the effect of to parasitic capacitance. On the other hand, 3D package approach makes it achievable to interconnect chips made of different fabrication technologies with the drawback of having limitation in the number of tiers due to the heat produced inside the cube. Figure 2.13b shows the face-to-face microbump technique. This technique shows improvement in performance as the parasitic capacitances are downsized by the employment of short wires between tiers. Though only a maximum of two tiers can be bonded, this approach is in conjunction with wire bonded and through-via technologies.

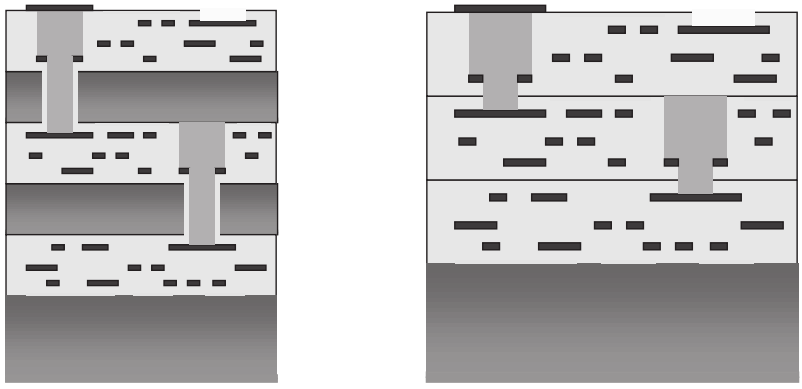


(a) (b)

Figure 2.13.: Illustration of microbump 3D technology: (a) 3D package, (b) face-to-face [28]

2.4.3. Through Via

The greatest advantage of this technique is itÕs interconnect density though it comes at a higher price. The assembly process is a simple one and does not act as a limiting factor when it comes to the maximum number of tiers that can be bonded. As shown in Figure 2.14, the second wafer is placed face down onto the first wafer (face-to-face) and the subsequent wafers are placed face down on top of the second wafer (face-to-back) thereby increasing the tier count. In order to facilitate connectivity between tiers, holes are etched right from the upper to the lower wafers to be filled with tungsten. By the time the next chip needs to be placed, the back of the previously etched chip is already thinned through polishing. The tungsten-vias in the top most tier are not polished but rather left protruded with cuts to have power, ground and I/O connections. Like 3D package, heat delimits the number of tiers that can be bonded. Additionally, all the layers in the upper tier and the top layer of the lower tier are required for routing. The difference between bulk technology and Silicon-on-Insulator (SOI) technology is that, the former coats the hole with an insulator of pitch 50 μm while the latter polishes the substrate layer completely till the buried oxide and reaches an inter-tier pitch of 5 μm.

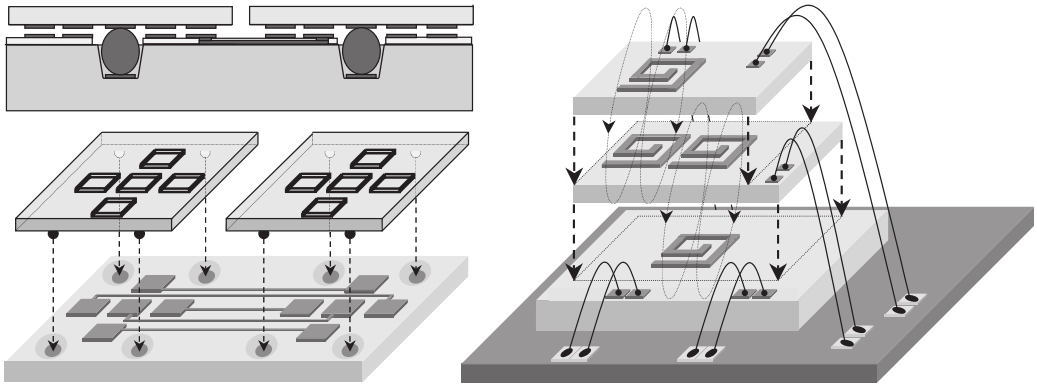


(a) (b)

Figure 2.14.: Illustration of through via 3D technology: (a) bulk, (b) silicon on insulator [28]

2.4.4. Contactless

This approach also called as "AC-coupled" interconnects tiers using capacitive or inductive coupling and hence knocked out inter-tier DC connectivity processing steps and got rid of longer wire lengths as periphery routing is no longer required. It is also cost effective compared to microbump and through-via techniques as chip thinning requires only minimal processing steps. The half capacitors which are formed from the top level of metal are used to couple tiers in "capacitive coupling" approach. The distance between the tiers, the fall and rise time of the technology and the dielectric constant of the gap determine the interconnect density. In order to have DC connectivity between chips or between chip and substrate, solder bumps are used and this created gap between the chips. For sufficient coupling to happen between the two half plates of the inter-chip capacitor, this gap should be relatively small to that of the plate. Though high-k dielectric underfill can be used to fill the gap, the most preferred technique is AC-Coupled Interconnection (ACCI), which cuts trenches in the substrate to let the solder bumps get deep enough to make contact between the chip and the substrate. Figure 2.15a shows the cross sectional view of a Multi-Chip Module (MCM) through buried bump technology. This technology not only increases the manufacturing yield due to less coupling failure but when combined with high-k dielectric underfill technology offers various advantages and one such benefit is stress relief between chip and substrate. In a stack of three or more chips, where separation between coupling elements is determined by the chip thickness, communication between the chips throughout the stack can be provided by inductors. An example of three-tier stack using inductive coupling technique is shown in Figure 2.15b. This technique is inexpensive and easy to construct as each tier is placed face-to-back and their DC power and ground connections are furnished by wire bonds.



(a) (b)

Figure 2.15.: Illustration of contactless 3D technology: (a) capacitive, (b) inductive [28]

2.5. Design Steps for Application Mapping onto FPGAs

The necessary steps for mapping a logic circuit onto an FPGA are shown in Figure 2.16. The logic circuit is described at high abstraction level in a hardware description language such as VHDL or Verilog. Predesigned IP cores can also be integrated, which simplify the design. The implemented code can be verified in advance by means of a behavioral simulation. The code is then synthesized. The synthesis tool generates a technology-independent gate-level netlist consisting of generic library cells such as AND, OR, XOR, and flip-flops. At the same time, the design is transformed into a structural VHDL description, which also contains the generic primitives from the synthesis library. In addition, timing data (e.g. estimated propagation delays for the generic gates) is already generated, which is taken into account in a post-synthesis simulation. This is followed by the technology mapping process. The synthesized gate-level netist is converted into a technology-dependent network list consisting of logic cells of the target technology. With the placement step, each node in the technology-dependent network list is assigned a place on the physical array. During the routing step signal paths are determined. The path lengths and thus the resulting signal delays are optimized both during placement and during the route. After the place & route steps, the path delay times are also known and an adequate post-layout simulation can be carried out, which is then very close to reality. Finally, the bit streams are generated with which the desired logic circuit is implemented by (re-)configuration of the FPGA.

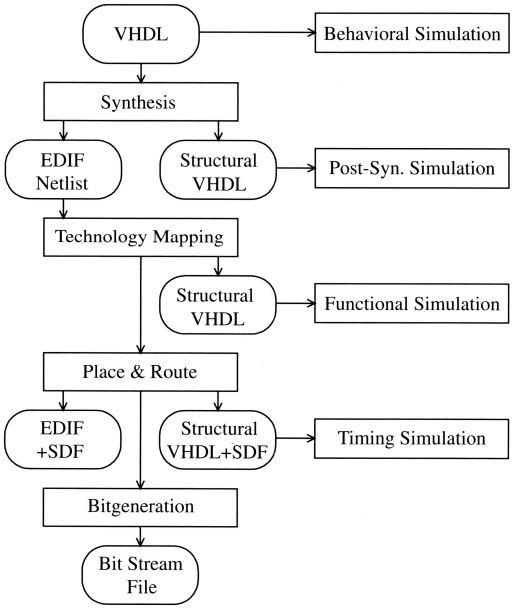


Figure 2.16.: Design flow for FPGAs [53]

3. Related Work and State of the Art

This chapter surveys the related works and discusses how they are differentiated or complemented by the work of this thesis. In particular existing works related to low-power FPGA architectures, custom embedded FPGAs, FPGA virtualization, customizable CAD tools and 3D FPGA architectures are eyed.

3.1. Architectural Efforts to Reduce Area and Power Consumption in FPGAs

On architecture level there have been several approaches presented to optimize area and power consumption in FPGAs.

The tuning of basic architectural design parameters such as LUT size, cluster size and channel width in typical island style FPGAs have an impact on the utilization ratio and area efficiency of FPGAs while generally a trade-off with delay has to be made. In [4] Ahmed and Rose studied the effect of LUT and cluster size on area and delay within the design space of 2 to 7 LUT inputs and 1 to 10 LUTs per cluster. From their published experimental results over the MCNC benchmarks we can observe a variance of approx. 41% in total FPGA area and 67% in total critical path delay for a 0.18 μm 1.8 V CMOS process, which demonstrates the significance of design parameters. Ahmed and Rose conclude that a LUT size of 4 to 6 and a cluster size of between 3-10 provide in average the best area-delay product. This is reflected also by various commercial FPGAs.

Further improvements could be obtained by more efficient logic cell and interconnect architectures. Hu et al. at the Xilinx research lab examined in [48] the effects on logic area when mixing LUTs and macro-gates in heterogeneous programmable logic blocks. The experimental results over an extensive set of benchmarks achieved in average a performance gain of 7%, while the logic area could be reduced by 15 % as compared to homogeneous 6-input LUTs. However, a major drawback of this technique is a reduced flexibility in the routing architecture as the proposed macro-based architecture does not allow full pin-permutation. This means also that it can be applied only to fully populated clusters, which compensate for this by their full connectivity in the internal local routing of a logic block, but this in turn comes at higher area costs for the local routing infrastructure.

Further techniques to reduce static power dissipation by clock gating and power gating have been addressed for instance in [44] and [51] respectively.

Despite the examples above present valid approaches on architecture level to reduce in average area and power in FPGA, they are studied each in a mainly isolated manner. Furthermore, the majority of subsequent works and commercial devices rely on parameter trade-offs based on the main target to improve the average efficiency over a complete

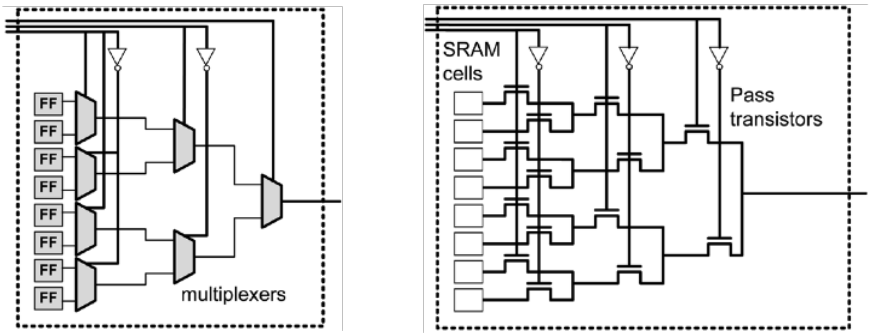
benchmark suite, while single benchmarks can suffer a degradation. This is a typical approach for general purpose architectures, where the average performance matters and a wide field of applications is targeted by one device. However, with the diversity of applications and the need for more specialization as described in chapter 1.2, more distinct optimization and customization strategies with evaluation of individual requirements, consideration of runtime effects and characteristics of the underlying technology are becoming essential for extreme and evolving applications that can not be satisfied by general purpose architectures.

3.2. Custom and Embedded FPGAs

To reduce the ASIC gap while still offering flexibility, there have been a few approaches to embed an FPGA fabric in an ASIC, yielding heterogeneous System-on-Chips (SoCs) with reconfiguration capabilities. While devices like the Xilinx ZYNQ or Altera Cyclone V SoC are getting closer to general purpose computing, there is also an increasing need for specialized devices for high volume, low cost and low power applications, e.g. in the fields of mobile devices, Cyber Physical Systems (CPS), Internet-of-Thigs (IoT). Hence the focus here is on solutions to embed customized FPGA fabrics that satisfy the applications demands in a more fitted and consequently efficient way. The most obvious way to customize an FPGA fabric is to tune its size (i.e. complexity) while there are different architectural parameters that can be adjusted, such as number of logic blocks, granularity of LUTs, cluster size, routing channel width, etc.. But also the inclusion or exclusion of special elements, such as DSP blocks, RAM blocks, serializer/deserializer, based on the individual needs can have a significant effect on performance and area.

There are two reasonable ways to embed an FPGA in an ASIC design, either as hard IP core or as soft IP core. The main difference is that hard IP cores are pre laid out and added as fixed hard macros in the layout process of IC design, while soft IP cores are supplied at RTL or netlist level and need to be synthesized, placed and routed as standard cells by the ASIC designer. While hard IP cores benefit from an optimized layout (possibly full-custom) with superior circuit performance yet smaller effort for the SoC designer (due to pre-layout and -verification), soft IP cores can offer parameterization, a better fit for the application, unchained placement with variable aspect ratio, and the possibility to migrate the design to a new technology node.

Wilton et al. present in [107] some general design considerations for embedding FPGA fabrics by the means of soft IP as opposed to hard IPs. The most obvious difference on circuit level is that the soft IP cores use standard cell libraries and are built of NANDs, NORs, inverters, flip-flops and multiplexers, while hard IP cores can have a more custom implementation. For instance as shown in Figure 3.1, in a soft IP core with standard cells a LUT is built of flip-flops and multiplexers, while in a hard IP core it is realized more area efficient using SRAM cells and pass transistors. Wilton et al. observed in their experiments an area overhead of 6.4x of a soft IP FPGA core compared to a hard IP FPGA core. However, this doesn't take into account that application wise a hard IP FPGA with a pre-defined size and aspect ratio has obviously a lower utilization of its programmable resources (thus a lower mapping efficiency) compared to a fitted soft IP FPGA. Another



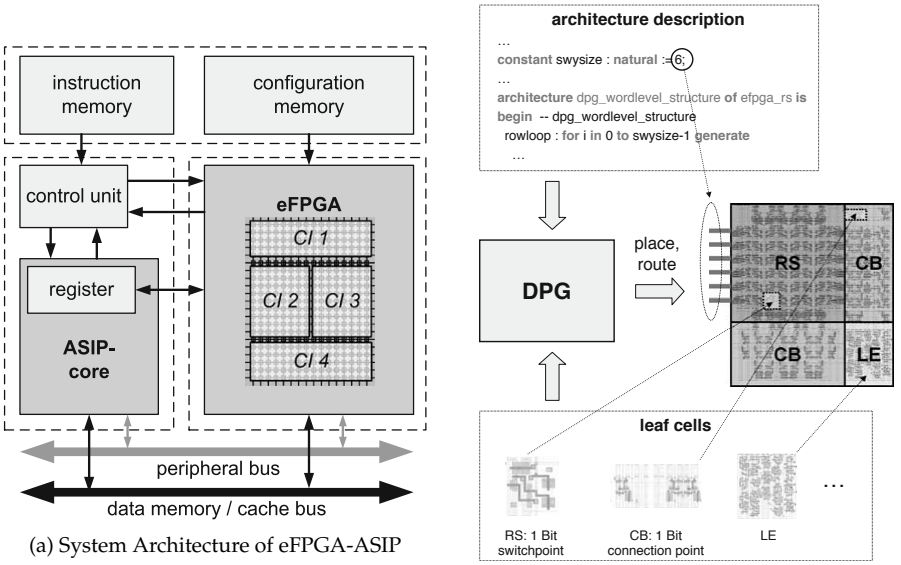
(a) (b)

Figure 3.1.: Comparison of a LUT implementation in a) a soft IP core using standard cells and b) a hard IP core [107]

observation is that, due to the nature of FPGAs with their flexible routing capabilities, CAD tools that are engaged in synthesis, place, route and timing verification steps of the soft IP FPGA core may have problems with combinational loops. Wilton et al. suggest to use FPGA architectures with unidirectional routing infrastructure (i.e. directed from one side of the core to the other side) in that case, however this strongly limits the placement and routing of applications and can have negative effects on performance and area utilization. The architectures proposed in [107] consequently are intended only for small combinational circuits, such as the next state logic in a state machine.

One of the early commercial solutions for embedded FPGA IP cores is FlexEOS from the french startup company M2000 (later AboundLogic) [23]. FlexEOS was a hard macro in GDS-II for embedding FPGA fabric in an ASIC and was used e.g. in the MORPHEUS platform [82]. The company doesn't exist anymore and detailed information about the architecture is not available. Based on [23] and [82] it is known that the FlexEOS architecture relied on basic 4-input LUTs and SRAM, non-uniform routing infrastructure and was optimized for high density. The FlexEOS FPGA core was supplied as hard IP in GDS-II format.
Before M2000, the same people have founded the company Meta Systems in 1996 that was soon acquired by Mentor Graphics Corp. in the same year. Mentor Graphics uses a custom FPGA architecture in their chip Crystal2 for emulation and testing systems and Rizzatti mentions in [85] that the architecture originates in the work of Meta Systems.

Neumann et al. present in [79] an extensible ASIP architecture that utilizes an embedded FPGA core to extend the instruction set of an ASIP as depicted in Figure 3.2a. The embedded FPGA is based on a parameterizable template and is generated by a data path generator (DPG) that uses hand-designed basic cells (so-called leaf cells) to compose and generate a complete macro. This methodology is illustrated in Figure 3.2b. A leaf cell can be either a logic element (LE), a part of a routing switch (RS) or a connection box (CB) and is pre laid out prior to the generation of the embedded FPGA macro by the DPG. This



(a) System Architecture of eFPGA-ASIP

(b) Generation of eFPGA macro with DPG tool

Figure 3.2.: The flexible ISA ASIP by Neumann et al. [79] combining an ASIP with an embedded FPGA

methodology is somewhere between a hard IP and a soft IP approach, yielding a better efficiency than a soft IP and offering more customization than the hard IP.

The company Menta offers an embedded FPGA Core IP to be used in SoC, ASIC and ASSP designs. As shown in figure 3.3 the architecture consists of embedded logic blocks (eLB), optional embedded customer blocks (eCB), optional embedded memory blocks (eMB), programmable IO interfaces and configuration interface. Pre-defined eFPGA cores in the range from 7k to 60k equivalent logic gates (i.e. 512 to 4032 equivalent LUT6) are offered as hard macro cells. Additionally, custom eFPGA cores can be designed on demand. Therefore, the embedded FPGA Core IP is scalable and customizable by a tool called Menta Origami Designer™ in terms of number of eLBs, number and type of eCBs and eMBs, number of clocks, and number of IOs. Furthermore, the aspect ratio of the core can be tailored.

Another commercial solution is EFLX by the company Flex Logix [32] that offers FPGA IP cores for embedding in SoC designs. The EFLX architecture relies on two 4-input LUTs per logic block and a hierarchical routing based on the work presented by Yuan et al. in [112], where a modified folded-Beneš network is realised. Clock gating and power gating mechanisms reduce power comsumption. EFLX comes as predefined and laid out cores in GDS-II format, along with LIB, LEF, CDL and encrypted Verilog files. There are two different cores with fixed size available, i.e. EFLX-100 with 120 LUTs, EFLX-2.5K with 2520 LUTs. Scaling can be done through concatenating multiple cores via expendable network I/Os. Also a mix with DSP cores and RAM blocks is possible.

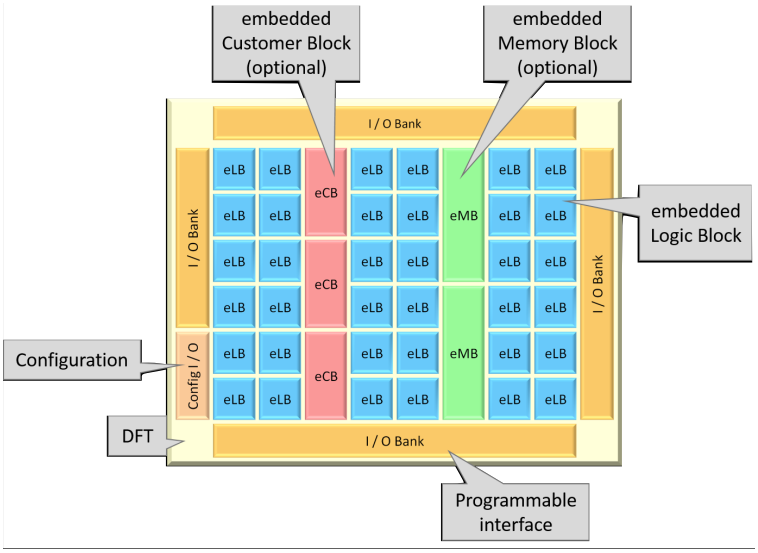
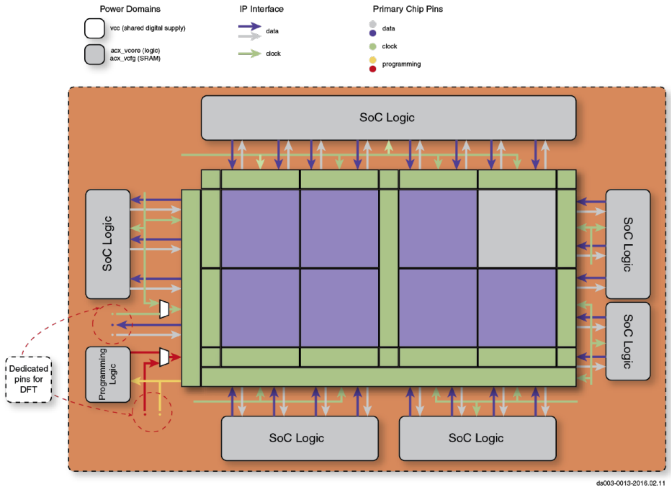


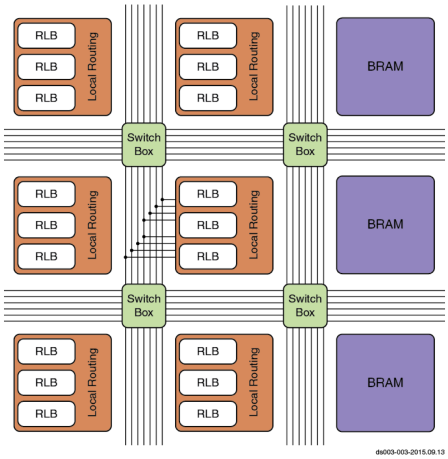
Figure 3.3.: Menta eFPGA Core IP architectural features [71]

Achronix offers with Speedcore an eFPGA IP core based on the Speedster22i FPGA [1] that can be integrated in SoC designs as hard IP, as illustrated in Figure 3.4a. Unlike other commercial IP solutions, Achronix provides more details about the architecture in public. As illustrated in Figure 3.4b it is an island style FPGA with reconfigurable logic blocks (RLBs), block RAMs (BRAMs), logic RAMs (LRAMs) and DSP blocks and a uniform global interconnect with routing channels and switch boxes. As it can be seen in Figure 3.4c a reconfigurable logic block (RLB) contains four 4-input LUTs, four flip flops, a 4-bit ALU and MUXes for the internal interconnects. At the boundary of the IP block, there are interfaces for data signals, clock inputs and programming to integrate with the rest of the ASIC logic. Customization is done by chosing the quantity of logic blocks, logic RAM, BRAM and DSP blocks, aspect ratio and submitting these requirements to Achronix. Achronix delivers a GDS-II of the customized Speedcore IP that they can directly integrate into their SoC, along with Verilog definition of logical connectivity at boundary, Liberty timing lybrary for timing closure at boundary and LEF files defining the physical floorplan, pins and metal blockages. In addition a custom version of design tools for design, mapping and programming of application onto the FPGA fabric is supplied.

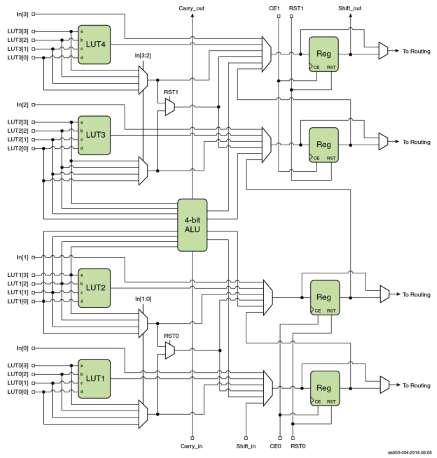
Unlike M2000, Menta, Flex Logix and Achronix, the french company ADICSYS (btw. founded by 4 persons that worked before at AboundLogic aka M2000) goes the way of soft IP core and offers an embedded FPGA core as synthesizable programmable core (SPC) on RTL level [3]. The IP comes with the programmable core, a configuration controller and a built-in self-test (BIST) unit for manufacturing testing or in-system self-testing, that are interconnected as depicted in Figure 3.5a. SPC uses a proprietary FPGA architecture with



(a) Speedcore sample instantiation



(b) General topology

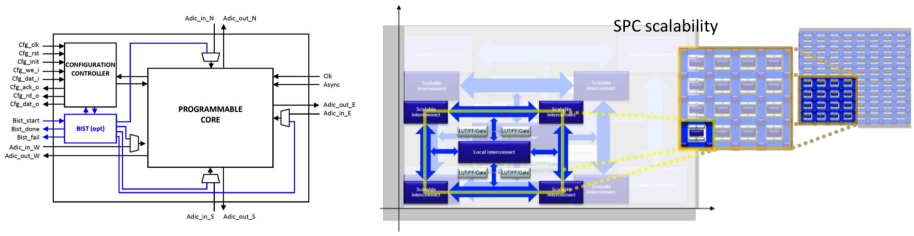


(c) Schematic of reconfigurable logic block (RLB)

Figure 3.4.: Speedcore eFPGA by Achronix [1] - a) Instantiation in SoC design, b) topology and c) reconfigurable logic block

scalable interconnects (see Figure 3.5b) and supports 100 to 100k LUTs per core and multiple SPC per chip. Detailed information about the architecture is not published.

For the *V-FPGA* presented in this thesis, though intended primarily for virtualization, the soft IP approach is recommended when integrating into ASICs as embedded FPGA. The reason for this is that a strong aspect and the main focus of the *V-FPGA* lies in the flexibility and highly fitted customization through a very rich set of parameters beyond the



(a) SPC block diagram

(b) Scalability of SPC

Figure 3.5.: The synthesizable programmable core (SPC) by ADICSYS [3]

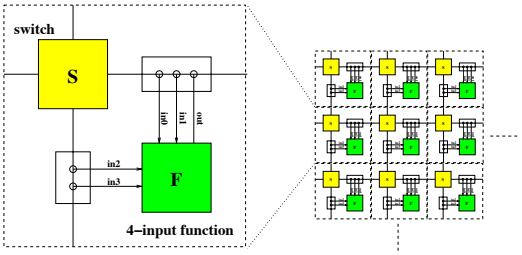
works presented above. This includes amongst others CLB count, aspect ratio, LUT size, cluster size, routing channel width, switch box structure, layers and TSV distribution in 3D integration, configuration scheme, fine grain and coarse grain partial reconfiguration. As hard IP it would compromise on mapping efficiency and flexibility.

3.3. Virtualization in Context of Reconfigurable Architectures

In [39] Fornaciari and Piuri introduced the term "Virtual FPGA" to present the idea of resource sharing of an FPGA in a heterogeneous system, where an operating system that is executed on a General-Purpose Processor (GPP) shares an FPGA among tasks through dynamic reconfiguration. Thereby, applications have a virtual view of the FPGA that is then mapped on the available physical device by the operating system, in a way similar to the virtual memory. The FPGA is virtualized by time multiplexing its physical components for all application tasks that need to realize part of their computation in hardware. This is done by downloading the corresponding configuration on the FPGA itself. The FPGA can be therefore treated as any other shared hardware resource in the general-purpose multitasking system. The synchronization among tasks to use the shared FPGA is managed and enforced by the operating system as it is accomplished for all other resources. For this purpose Fornaciari and Piuri proposed a methodology that relies on the principles of dynamic loading, partitioning, overlaying, segmentation, pagination, input and output multiplexing.
We need to differentiate that Fornaciari uses the term virtualization in the context of operating and runtime systems representing the software point of view, rather than in context of FPGA architectures. In contrast, the virtualization methodology presented in this thesis refers to hosting or incorporating a custom FPGA architecture by a commercial off-the-shelf FPGA that features a different architecture.

A virtualization concept in terms of hardware architectures is introduced in [58] by Lagadec et al., where the authors present a toolset for generic implementation of virtual architectures. The scope is to automatically generate programmable hardware architectures that are mapped on an FPGA. The methodology relies on the description of cells and interconnects in a high level representation that are replicated as an array (as de-

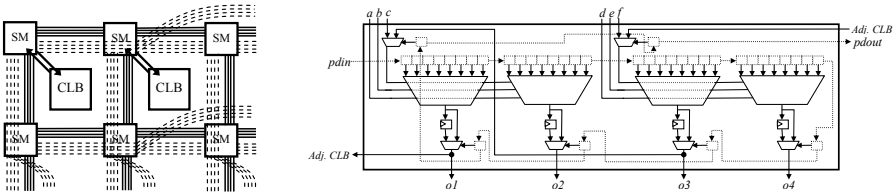
picted in Figure 3.6). A toolset performs application mapping by placement and routing of a netlist. Thereby, the netlist itself is the design entry, while the nodes must match the inputs and outputs of the specified cells. Starting with an initial random placement, simulated annealing is used for iterative otimization. A router based on a PathFinder algorithm determines the interconnection paths. There is no synthesis from hardware description languages mentioned, thus it is not clear which level of complexity the application mapping can support. The authors present an example that demonstrates the generation of a systolic processor for DNA comparison, while a virtual cell as part of the systolic processor was designed and synthesized on a Xilinx Virtex FPGA. The design steps from the high level representation of the virtual architecture to its mapping on the underlying physical FPGA are not described, which leads to the assumption that it is a custom hand coded process for each type of cell. The approach and the examples are closer to coarse grain reconfigurable architectures (CGRA) than to FPGA (or fine grain) architectures. This is also indicated by the statement that "a set of wires (a bus) is routed as a single entity". That approach can lead to performance benefits since the architecture can be specialized to a very narrow application or circuit class, such as systolic arrays. However, the drawback is then a rather low flexibility. A strong focus of the paper lies on the generic tool flow for architecture representation and place&route of application netlists onto various virtual architectures. The virtualization aspects from hardware perspective, including the mapping onto the underlying platform as well as programming mechanisms and configuration management remain predominantly unaddressed.




```
(ARRAY ((DOMAIN 0 99 0 99)
(COMPOSITE (WIRE wireH) (WIRE wireV)
(SWITCH switch (Type disjoint))
(FUNCTION func (IN in0 in1 in2 in3) (OUT out))
(CONNECT
  'func out connectTo: wireH'
  'wireH connectTo: funv in0'
  'wireH connectTo: funv in1'
  'wireV connectTo: func in2'
  'wireV connectTo: func in3'
  'wireH connectTo: switch east'
  'wireH connectTo: (self relativeAt: 1@0)
  s                    wi tch  we st '
  'wireV connectTo: switch south'
  'wireV connectTo: (self relativeAt: 0@–1)
  s                    wi tch  nor th
  )))
```

Figure 3.6.: Lagadec et al.: 2D array of identical cells. A cell is composed of one switch and one 4-input boolean function. [58]

Lysecky et al. introduced in [65] a simple fine grain virtual FPGA that is specifically designed for fast place and route. The architecture has a mesh structure with configurable logic blocks (CLBs) surrounded by switch matrixes (SMs) that are interconnected by routing channels as illustrated in Figure 3.7a. Thereby, a CLB is connected directly to a single switch matrix as opposed to architectures where logic blocks connect directly to the surrounding routing channels - the latter allow shorter routing paths since a logic block can connect to all surrounding channels rather than to a single switch matrix. As depicted in



(a) General structure (b) CLB

Figure 3.7.: Structure of the virtual FPGA by Lysecky et al. [65]: a) general topology and b) configurable logic block (CLB)

Figure 3.7b, the logic blocks have a fixed size and consist of two 3-input-2-output LUTs per logic block (note that a 3-input-2-output LUT is made of two 3-input LUTs that share the same input signals). Each logic block has 6 main inputs and 4 main outputs in addition to extra ports for carry logic. Lysecky et al. support two types of switch matrixes, a tristate-buffer based switch matrix (TSM) or a multiplexor based switch matrix (MSM), see Figure 3.8 and Figure 3.9 respectively. The channels between switch matrices have a fixed width. The CLBs as well as the switch matrices are controlled by configuration bits that are stored in flip flops. The architecture is designed with synthesizable and portable VHDL code that can be mapped on COTS FPGAs. However, it is to mention that TSMs are not feasible on all underlying platforms, since many FPGAs offer tristate-buffers only at the IOs. Lysecky et al. report an area overhead of 100x for their virtual architecture mapped on a Xilix Spartan-II device, which is rather high. The largest share on the area consumption have the MSMs. This is also obvious when we recall the structure in Figure 3.9, where each output of the MSM features a 12:1 MUX. The synthesis and mapping of one 12:1 MUX on a Xilinx Spartant-II device that features 4-input LUTs would require around 8 LUTs (while utilizing additionally 3 of the slice-internal MUXF5 primitives). Regarding the choice of the architectural parameters (LUT size, cluster size, CLB I/Os, channel width, switch block flexibility) it is not fully clear which objectives were targeted. The extensive studies and experiments of Ahmed, Rose and Betz (for instance in [4] and [15]) indicate different values for a good trade off between area and delay. The *V-FPGA* architecture proposed in this thesis follows a similar granularity and architecture class as the work of Lysecky et al.. However, the *V-FPGA* has a flexible architecture with higher complexity that features a rich set of variable parameters for customization, scalability and objective related optimizations.

The ZUMA architecture by Brant et al. [20] targets to reduce the area overhead of the virtualization layer by utilizing LUTRAMs of the underlying platform to implement LUTs and multiplexers with fewer resources compared to generic HDL descriptions. This technique also reduces the number of required flip flops for configuration bits. Architecture wise, ZUMA is a clustered LUT based FPGA architecture with island style topology. The local interconnect within a cluster is controlled by a 2-level crossbar. Brant et al. demonstrated that the exploitation of LUTRAM can bring up to two thirds area reduction compared to generic HDL. However, the approach has also limitations. ZUMA is incompatible to bidirectional routing. The realization in [20] doesn't support sequential circuits, however this problem was recently addressed by Wiersama et al. in [105] and ZUMA

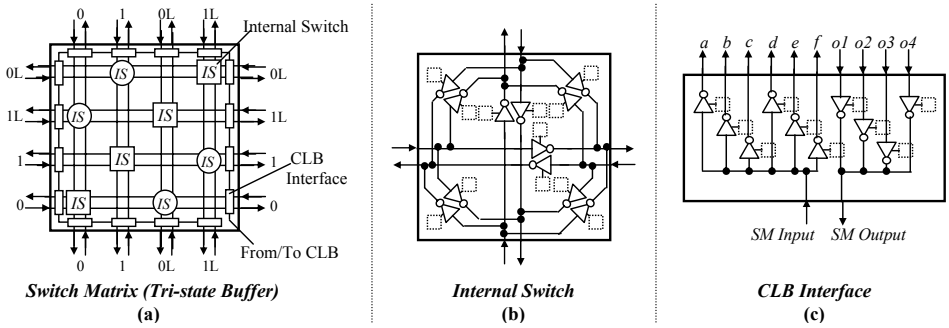


Figure 3.8.: Lysecky et al.: schematic of a tristate-buffer based switch matrix (TSM) [65]

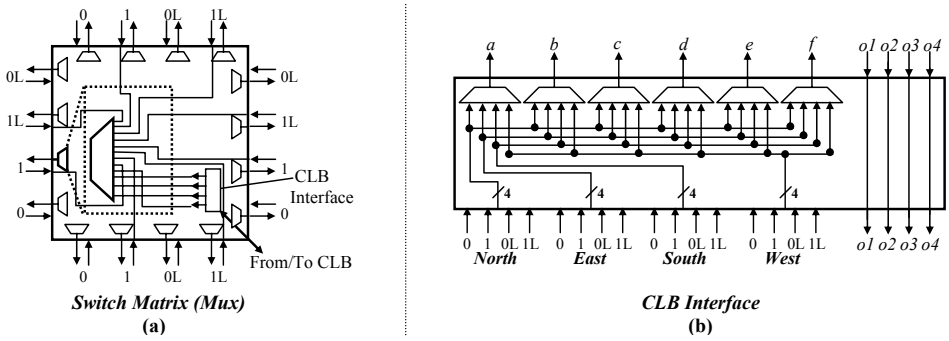


Figure 3.9.: Lysecky et al.: schematic of a multiplexor based switch matrix (MSM) [65]

accordingly extended. A noticeable limitation is that the design compiles only on Xilinx and Altera FPGAs while LUTRAMs are not supported by all FPGAs, thus the portability aspect of virtualization is only partially fulfilled. Furthermore, a change of parameters such as LUT size might need a manual matching of virtual resources to physical resources, which makes parameterization difficult.
We need to differentiate that the ZUMA approach follows a different philosophy than *V-FPGA*. While the *V-FPGA* aims at being portable across various underlying FPGA platforms as well as being easily mapped on an ASIC process, the methodology of Brant et al. utilizing target specific and exclusive elements in ZUMA would fail here the purpose of the *V-FPGA*. Furthermore, the LUTRAM approach would lose its efficiency when mapped onto an ASIC which is the second target of the *V-FPGA*. Nevertheless, due to the modular hardware model of the *V-FPGA*, the utilization of exclusive platform specific resources is not prohibited.

The major drawbacks of all virtual FPGA architectures, including the one presented in this thesis, are that they require a higher chip area and introduce larger path delays compared to physical FPGAs. The reason lies in the fact that one virtual logic cell is realized

by a multitude of programmable logic cells of the underlying physical platform. This is the price that needs to be paid for portability and higher flexibility and essentially there is an analogy to the comparison of FPGA vs. ASIC. Thereby, the factor of area overhead of a virtual FPGA over its underlying physical FPGA platform depends mainly on the granularity of the underlying platform as well as how well the virtual resources can be matched by the physical resources. This factor is individual for each combination of virtual architecture and underling platform. Thus, the same Virtual FPGA has a different area efficiency on one underlying platform than on another and a change in the design parameters of the virtual FPGA can turn the game. That's why it has been difficult to compare the few existing virtual FPGA architectures with each other and any context-less conclusion about the superiority of one virtual architecture over the others is of limited validity, not only due to the lack of transferable quantification but also due to different purposes and abilities of the existing solutions. To bring more transparency regarding area efficiency and to facilitate future comparisons, Chapter 6.3 introduces transferable metrics that can be applied on virtual architectures for various target technologies.

3.4. Generic CAD Tools for FPGAs

The basic purpose of CAD tools for FPGAs is to map an application onto an FPGA architecture and to generate a bitstream with which it is possible to configure the resources of an FPGA in a way that it creates a respective circuit to run the application.

Existing vendor tools for COTS FPGAs generally don't support custom FPGA architectures. During the past two decades a number of academia driven efforts were carried out to provide parameterizable tools for exploring custom architectures and mapping applications onto them.

SIS, a system for sequential circuit synthesis brought forth in [88] is a framework for testing different algorithms and for synthesizing and optimizing sequential circuits by receiving any one of the following as input: State transition table, signal transition graph and logic-level description. On one hand it generates an optimized net-list of the underlying technology and on the other hand it maintains the input-output behavior.

In 2006, a technique called And-Inverter Graph (AIG) was introduced in [75] which represents the combinational logic using a network of two-input ANDs and inverters. By switching between AIG rewriting and AIG balancing, area optimization without increasing delay and delay optimization without increase in area are obtained respectively. Implemented on the sequential logic synthesis and verification tool ABC, this technique was able to be faster than SIS and MVSIS yet offering a better quality, which will be beneficial for applications like hardware emulation, estimation of design complexity and equivalence checking.

Quartus Integrated Synthesis (QIS) made known under Quartus University Interface Program (QUIP) [7] can be used in two modes, either as a comparison tool or as a front end to convert VHDL/Verilog design codes into formats used by the academic tools. This tool accepts input not only as VHDL/Verilog code but also as schematic, instantiated LPM modules and as IP cores. It supports constructs like FOR, GENERATE, GENERIC, etc,

finite-state-machines, RAM and multipliers by converting them into synthesizable subsets, logic, embedded memory blocks and DSP blocks respectively.

[14] describes a CAD tool called Versatile Place and Route (VPR) for FPGA architectures, which can perform placement and routing either as global routing or as a combination of global and detailed routing by taking the mapped netlist and architectural description of the targeted FPGA as its input file. The output file is also helpful in determining the utility of routed wire length, track count and maximum net length. Though their architecture description does not include segments with more than one logic block, they are highly flexible to add new routing architecture features.

Followed by [14], VPR 5 [64] introduces four new compelling features to the VPR tool such as single-driver routing, modeling heterogeneous logic blocks like hard memory and multipliers, optimization of electrical models in different process technologies and a set of regression tests needed to verify the functionality and quality of the output results in order to maintain robustness of the tool.

A rich toolset called MEANDER, which consists of non-modified academic tools (FreeHDL, SIS, T-VPACK), modified academic tools (E2FMT, ACE, VPR) and new tools (DIVINER, DRUID, DUTYS, DAGGER), was presented in [96]. By accepting inputs as VHDL design files of the application, all the necessary steps from elaboration, format translations, synthesis, logic optimization, activity estimation, packing, placement and routing onto custom island style FPGA architectures can be done with this toolset. It includes also a tool for bitstream generation, which however is not compatible to custom FPGA architectures other than the AMDREL FPGA. Another specialty of MEANDER is a web interface to operate the tools on a remote server (currently hosted at [69]) from any web browser or through ssh without the need of on-site installation.

NAROUTO, a framework for having architecture-level exploration in terms of delay, area and power/energy estimation in heterogeneous FPGAs [93] is an open-source tool. In order to automate the annotation of the generated net-list, a new toolset called Heterogeneous Support Toolset (HST) has also been developed. One of the merits of this framework is its ability to handle designs with IP cores more efficiently.

Even though the above named tools are flexible and cover most of the steps needed for application mapping, there are still some parts missing for a complete toolflow from design entry to the final bitstream. The employed architecture models are very abstract, which is good for design space exploration but somewhat decoupled from actual implementation. Most critical, none of the existing tools is able to create executable bitstreams for custom FPGAs because they were intended mainly for exploration purposes and lack a bistream generation tool as backend or there is no way to bring in details about custom configuration mechanisms and organization. Note that MEANDER contains the tool DAGGER for bitstream generation onto the AMDREL FPGA architecture, however the tool is not suitable for other custom and virtual FPGAs. Furthermore, a way of detailed manipulation and verification of the application mapping results with GUI is missing in the existing tools. To close these gaps, the proposed framework within this thesis contains a new tool called *V-FPGA Explorer*. It complements the above named tools by making a bridge between abstract layout and actual configuration. It transforms the textual synthesis and layout results of the other tools to an object oriented graphic capable representation, considering the custom architectural mechanisms, and generates the final bitstreams to con-

figure a custom *V-FPGA*. At any time the layout and function can be altered through a GUI or through XML files. It supports a rich set of parameters for architecture customization of the *V-FPGA* and is capable to generate custom architecture abstracts (so-called architecture files), that are required by the place & route and DSE tools. Additional features are testbench generation for simulation and script generation for parameter sweeps and for running VPR in batch to automate benchmarking and extend DSE capabilities. In conjunction with QUIP, ABC, SIS, VPR and MEANDER it forms a powerful and rather complete toolflow for application mapping onto custom *V-FPGA* architectures.

3.5. 3D FPGA Architectures

Since routing resources and interconnect have the majority share on area and delay, there is more and more focus towards 3D interconnects and routing architectures by 3D stacking of multiple die layers with through silicon vias or microbumps. Thereby, in most cases the vertical connections between layers are established either in switch boxes or in logic blocks.

The Rothko 3D-FPGA introduced in [70] and [59] is based on stacked layers of sea-of-gates architecture with metal interconnections (called interlayer vias) between layers. The vertical connections are made through the unified RLBs (Routing & Logic Blocks), i.e. as shown in Figure 3.10 a RLB can connect to its adjacent neighbours within a layer plus to an RLB above and an RLB below from other layers. Each layer allows horizontal routing in one direction only. Since two layers are stacked face-to-face and thus have opposite routing directions, the routing direction of a path can be changed by changing the layer through the vertical interconnects. Logic-wise an RLB contains a 3-input LUT.

In [33] a 3D pipelined asynchronous FPGA is presented where vertical connections between stacked layers are made through the SBs (switch boxes) of the routing infrastructure using pipelined 3D switches.

A model based study of monolithically stacked 3D FPGA is presented in [61] and [62], where a stacking is envisioned to take place within the same die (see Figure 3.11), thus allowing a higher density of vertical interconnects compared to chip or wafer stacking. However the monolithic approach is limited to adding only switch and configuration memory layers on top of the usual layers of a 2D FPGA, i.e. the logic still remains the same in a CMOS layer while only switch transistors and configuration memory cells are shifted to additional mask layers.

A 2-layer 3D-FPGA approach where routing and logic area are mostly separated on different layers is presented in [113]. As shown in Figure 3.12, logic and a small part of routing are on the first layer, while the second layer contains only routing. In contrast to most other approaches the 3D connections are not on the switch blocks but on the inputs and outputs of the logic block. They showed that it is possible to achieve for the MCNC benchmarks in average a 57% reduction of channel width by building 3D connections on the input and output pins of logic blocks.

Various 3D switch box designs are presented for instance in [43], [84], [99].

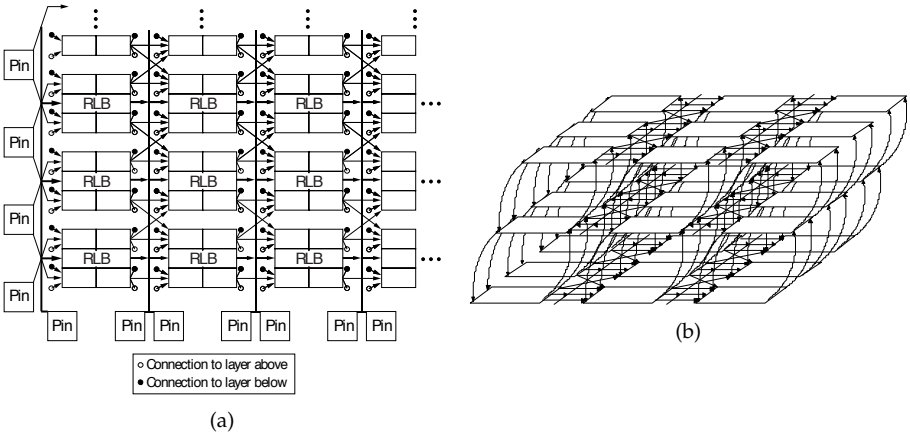


Figure 3.10.: Rothko 3D FPGA [59]: a) routing structure of a layer, b) connectivity between RLBs

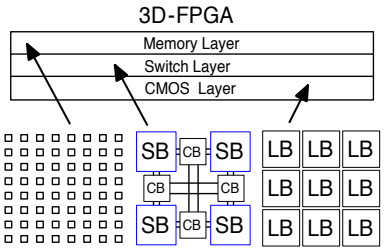


Figure 3.11.: Lin et al.: Monolithically stacked 3D FPGA [61]

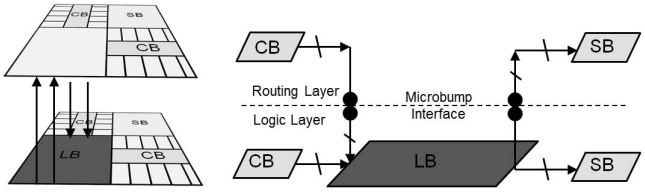


Figure 3.12.: Two-layer 3D FPGA by Zhao et al. [113]

The architecture level exploration of Siozios et al. in [94] indicates that stacking of FPGA layers with TSVs in switch boxes can bring in average for the MCNC benchmarks reductions of 13% in total wire length and 32% in power consumption while overall performance is increased by 35%. Further improvements in [97] with a heterogeneous mix of 2D and 3D switch boxes and different regions allow to reduce the number of vertical interconnects without penalty, whereby a reduction in area by 37% is possible compared to homogeneous 3D FPGAs. Compared to 2D FPGA, they show an improvement of 41% in delay, 32% in total power consumption and 36% in total wire-length.

Similarly as [43], [84], [99], [94] and [97], the 3D *V-FPGA* architecture presented in this thesis uses TSVs through SBs to establish connections between stacked FPGA layers, which in the first place is a choice justified by tool support, scalability and the possibility to support heterogeneous layers if needed, when compared to the other option of providing vertical interconnects through logic blocks. However, a difference in 3D *V-FPGA* compared to the prior art is that each port of a PSM can have an exclusive TSV connecting an identical port of a PSM from a different layer. A technique called *LoopbackPropagation* (see Section 4.1.4) allows then to route the TSV signal also to other ports of the same PSM. This can reduce the overall number of switches and configuration bits. Furthermore, the amount and distribution of TSVs can be parameterized in the 3D *V-FPGA* architecture.

4. V-FPGA: Virtual Field Programmable Gate Array

The key element of this work is the customizable virtual FPGA (*V-FPGA*) architecture that was first published in [50] and [35]. In fact, the first seed for this architecture was sown in the year 2010 as a part of my diploma thesis "Heterogene FPGA basierende DSP System on Chip Architektur" at the Karlsruhe Institute of Technology, where a simple uncustomizable first version of the *V-FPGA* was developed. This has been then re-designed for customization capabilities and continuously extended and improved during the course of this work.

The *V-FPGA* is a generic LUT-based FPGA architecture with mesh topology that can be mapped on existing commercial off-the-shelf (COTS) FPGAs, such as Xilinx, Altera, Microsemi, etc. Thereby, as illustrated in Figure 4.1, the applications will be mapped and executed on the virtual layer rather than the logic layer of the underlying COTS FPGA. The advantage with this approach is that the specification of the virtual FPGA stays unchanged, independent to the underlying hardware and provides therefore features, which the exploited physical host FPGA cannot provide. Due to the independence from the native configuration capabilities of the underlying platform, a special feature of the presented virtual FPGA amongst others is the dynamic reconfigurability which is for example not available with all off the shelf FPGAs. This was first demonstrated with a heterogeneous SoC architecture, integrating a *V-FPGA* fabric, mapped on a flash based Actel ProASIC3 FPGA for a low-power dynamic reconfigurable solution [50]. Furthermore the concept of FPGA virtualization enables the re-use of hardware blocks on other physical FPGA devices and enables portability of unaltered bitstreams among different

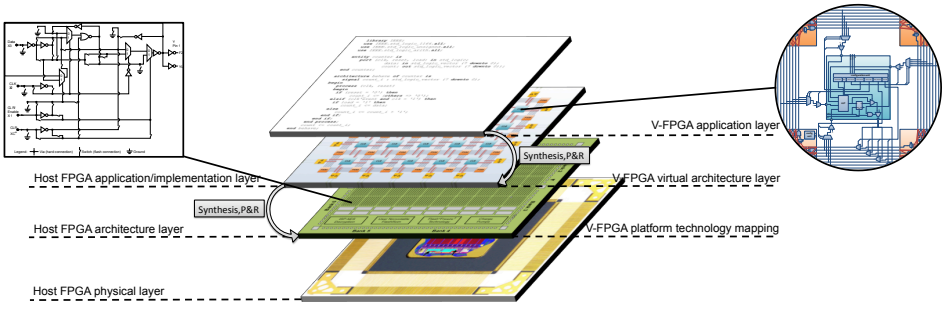


Figure 4.1.: Layer model of the *V-FPGA* approach with island based virtual architecture hosted by an Actel (now Microsemi) COTS FPGA

FPGA manufacturers and device families, e.g. in order to overcome the problem of device discontinuation.

Architecture wise, the *V-FPGA* can take the form of a 2D (see Section 4.1) or a 3D (see Section 4.2) FPGA. Apart form this the internal structure is highly customizable through a rich set of parameters in favour of a better fit for the application and consequently a higher efficiency when compared with stiff architectures. Table 4.1 introduces in advance an overview of the supported parameter set, while more details are provided in the following subsections.

The extensively scalable and parameterizable architecture is implemented in a mostly structural and fully synthesizeable HDL code, utilizing hierarchy, modularity and generics. Even though the *V-FPGA* was initially developed for virtualization, it can be also adopted to various target technologies and mapped on an ASIC flow as embedded FPGA. This makes it a suitable corner stone for customizable reconfigurable SoC architectures.

Table 4.1.: Parameter set of *V-FPGA*

Parameter	Description
L	Number of stacked 2D *V-FPGA* layers in a 3D *V-FPGA* (see Figure 4.2.1, p. 72)
TSVpC	Number of TSVs per channel in a 3D *V-FPGA* (see Figure 4.2.1, p. 72)
X	Number of CLB columns (see Section 4.1, p. 46)
Y	Number of CLB rows (see Section 4.1, p. 46)
K	LUT size; number of inputs per LUT (see Section 4.1.1.1, p. 48)
N	Cluster size; number of LUTs per CLB (see Section 4.1.5.1, p. 63)
I_left	Number of cell inputs on left side of CLB (see Section 4.1.5.1, p. 63)
I_top	Number of cell inputs on top side of CLB (see Section 4.1.5.1, p. 63)
I_right	Number of cell inputs on right side of CLB (see Section 4.1.5.1, p. 63)
I_bot	Number of cell inputs on bottom side of CLB (see Section 4.1.5.1, p. 63)
I	Number of total CLB inputs (see Section 4.1.5.1, p. 63) $I = K/2 \cdot (N+1)$ in auto mode $I = I\_left + I\_top + I\_right + I\_bot$ in custom mode
O_left	Number of cell outputs on left side of CLB (see Section 4.1.5.1, p. 63)
O_top	Number of cell outputs on top side of CLB (see Section 4.1.5.1, p. 63)
O_right	Number of cell outputs on right side of CLB (see Section 4.1.5.1, p. 63)
O_bot	Number of cell outputs on bottom side of CLB (see Section 4.1.5.1, p. 63)
O	Number of total CLB outputs (see Section 4.1.5.1, p. 63) $O = N$ in auto mode $O = O\_left + O\_top + O\_right + O\_bot$ in custom mode
F_inMUX_mode	Mode of BLE_inMUX flexibility (see Section 4.1.5.1, p. 63) 0: Fractional ($N + \lceil I/K \rceil$) 1: Fully-connected ($N + I$)
F_outMUX_mode	Mode of CLB output flexibility (see Section 4.1.5.1, p. 63) 0: direct connection 1: N:1 MUXs
W	Channel width; number of tracks per routing channel (see Section 4.1.1.2, p. 50)
SBtype	Switch box type (see Figure 4.1.3, p. 54) 0: Wilton 1: Universal 2: Subset/Disjoint

4.1. 2D Generic Architecture

As illustrated in Figure 4.2 the *V-FPGA* layer is an island based architecture with lookup tables (LUTs) in configurable logic blocks (CLBs) surrounded by routing channels. To ensure connectivity between distributed CLBs there are two types of programmable interconnects involved, Programmable Switch Matrices (PSMs) and Connection Boxes (CBs). The Connection Boxes (CBs) establish the connections of a CLB with its surrounding routing channels while the PSMs at the intersections of vertical and horizontal routing channels establish the global routing. I/O Blocks (IOBs) on the perimeter of the array access the outer channels through CBs and are the interface to the outbound systems. All these elements are described in detail in the following subsections. The total count of CLBs in an array can be scaled through the parameters X (number of CLB columns) and Y (number of CLB rows), i.e. $\#CLB = X \cdot Y$. This will indirectly affect also the count of PSMs and IOBs, i.e. $\#PSM = (X+1) \cdot (Y+1)$ and $\#IOB = 2 \cdot (X+Y)$ respectively.

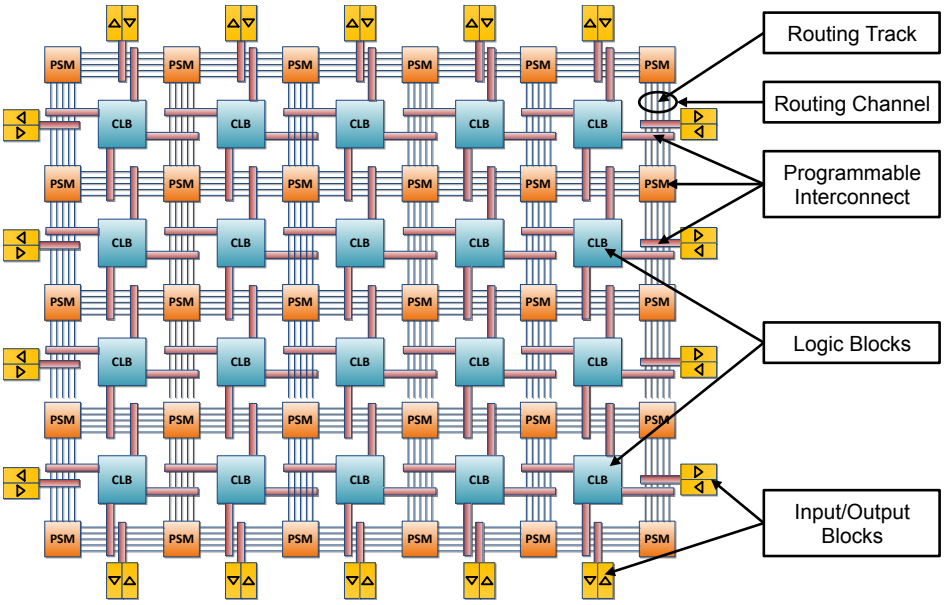


Figure 4.2.: Structure of *V-FPGA*

4.1.1. Configurable Logic Blocks

The Configurable Logic Blocks (CLBs) of the *V-FPGA* are LUT-based as depicted in Figure 4.3. The advantage of LUT-based logic cells opposed to multiplexer-based logic is that more complex functions can be implemented with a single LUT, as long as the number of function variables doesn't exceed the inputs of the LUT.

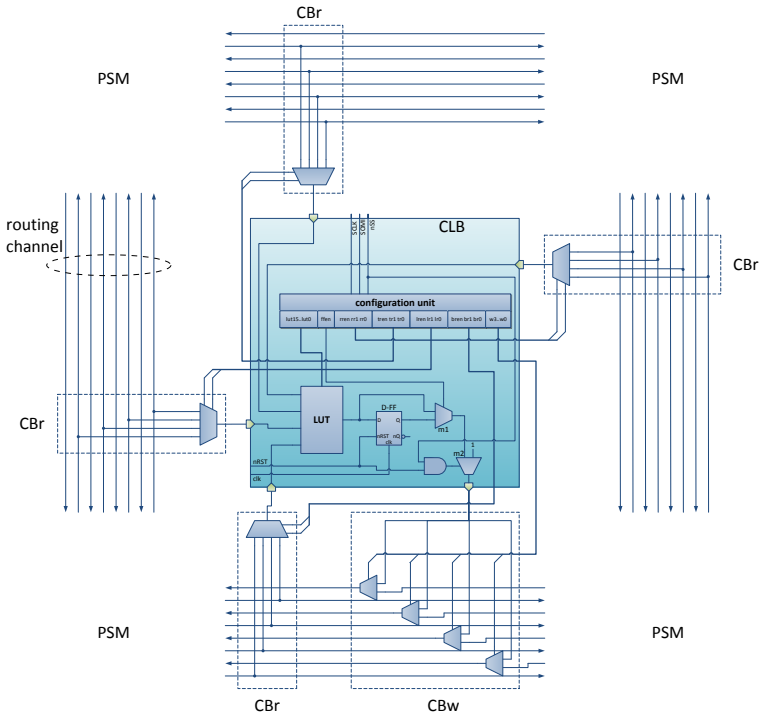


Figure 4.3.: Unclustered CLB with connection boxes

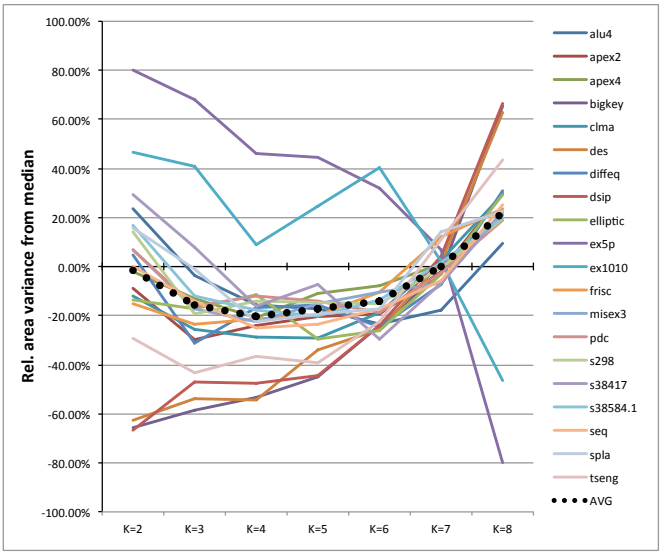
To support synchronous circuits and sequential processes, the output of a LUT leads to a clock edge controlled D-flip-flop. In the event that an asynchronous and purely combinatorial function should be implemented, the flip-flop can be bypassed by a 2:1 multiplexer ($m1$).
During a configuration or a reset, i.e. when nss='0' or $nRST$='0', the output of the CLB is set to logical '1' via a second multiplexer ($m2$), to produce a defined state and avoid glitches during configuration. In normal operation it selects the output of $m1$. This mechanism, which requires an additional MUX and AND gate, is optional and can be further optimized by simply modifying the nss and $nRST$ signals to be high active - an OR gate at the output would be sufficient then. However, this optimization has not been implemented due to legacy and compatibility reasons and furthermore the logic optimization step during synthesis is usually able to perform a similar level of reduction automatically. It should be mentioned that Figure 4.3 shows the minimal version of a CLB. In Section 4.1.5, a more complex version with flexible clustering is introduced, where a CLB can contain a multitude of LUTs locally interconnectable.

4.1.1.1. Intermezzo: LUT Size Tuning and the Effect on Area and Performance

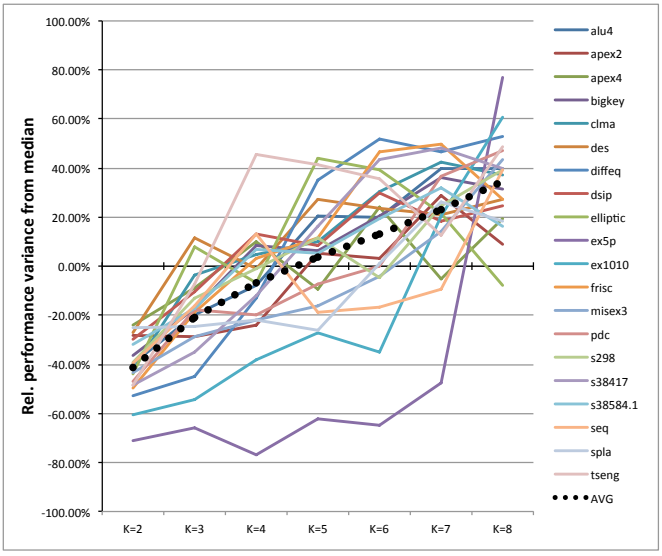
The LUT shown in Figure 4.3 has 4 inputs and can realize any boolean logic function with 4 variables, however in *V-FPGA* the number of LUT inputs can be actually varied by the parameter K , which has an effect on area efficiency and performance. Thereby not only the logic area needs to be considered, but also the routing area since K influences the decomposition and matching and indirectly also the interconnects and channel width. [86] and [4] indicate that a K between 3 and 4 provides the best area efficiency, while K=6 gives the best performance. [42] shows similar results through a theoretical model, while [101] indicates that $K = 6$ is the best choice for area, delay and area-delay product in nanometer technology.
However, those results are an average and furthermore it is very important to note that in virtual architectures the situation gets more complicated because both, area efficiency and performance, depend also on how efficient a K-input LUT can be realized by the underlying platform. Thus the findings in [86], [4], [42] and [101] and the concluded recommendations are not accurately applicable in case of virtual architectures. To demonstrate this, Figure 4.4 shows the variance of area and delay for the 20 largest MCNC benchmark circuits mapped on the *V-FPGA* (hosted by Actel's VersaTile technology [2]) with different LUT sizes in the range of $K = 2..8$. Additionally, Figure 4.5 shows the variance of area-delay product in a similar fashion. For each benchmark the variance is displayed as relative values compared to the median centred between the best and the worst result of the respective benchmark. This allows to compare the parameter sensitivity among all benchmarks, irrespective of their sizes and scales. Additionally a dotted curve is added, that represents the average over all benchmarks.

The analysis of Figure 4.4 and 4.5 reveals some interesting findings:

1. The average curve of area variance has a smooth bathtub characteristic with a wide optimum that stretches from $K = 3$ to $K = 6$ without noticeable difference, while outside of this range there is a degradation of area efficiency in both directions. A LUT size within the range of $K = 3$ to $K = 6$ will yield in average the best area efficiency for the general purpose case.
2. The different benchmarks differ greatly in area variance over LUT size K. Some of the benchmarks show a much higher parameter sensitivity than others. This manifests not only by the variance range and steepness that each benchmark shows but also by the smoothness of the curves. Furthermore they don't follow the same trends, i.e. some of the benchmarks are trending a higher area efficiency with rising K, while some others show the opposite trend and again others follow a bathtub characteristic. This shows that there is plenty of room for optimization through application specific costumization and parameter tuning.
3. Regarding the performance variation, all benchmarks show a trendline with increasing performance for increasing K. Consequently the average curve increases from $K = 2$ to $K = 8$ almost linearly. Some of the benchmarks show relatively smooth curves that are oriented around the average curve. In contrast, some others show a very disturbed and inconsistent curve with alternating change of trend and a few show a nearly exponential shape.



(a) relative area variance



(b) relative performance variance

Figure 4.4.: Effects of LUT size *K* on (a) area and (b) performance, shown through variance relative to medians of series within the range K=2..8

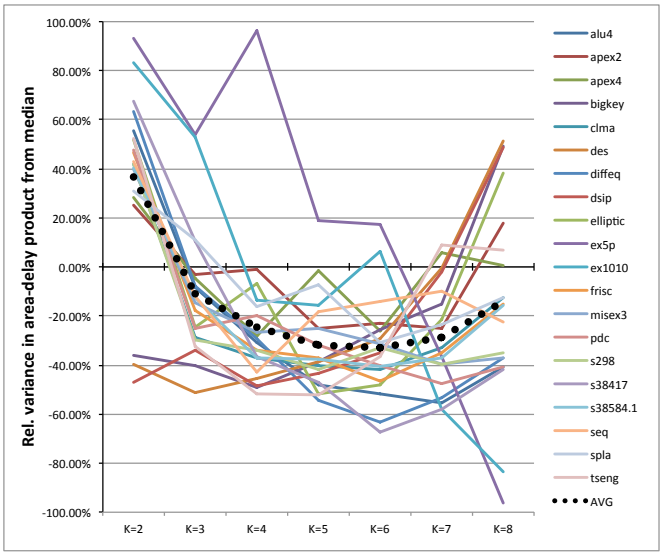


Figure 4.5.: Effects of LUT size K on area-delay product, shown through variance relative to medians of series within the range K=2..8

4. Around 5 % of the benchmarks have a performance maximum at $K = 5$, 10 % at $K = 6$, 40 % at $K = 7$ and 45 % at $K = 8$, which indicates that $K = 7$ to $K = 8$ are in average the recommended choices for performance optimization in the general purpose case. This differs from [86], [4], [42] and [101] and shows that their findings based on transistor level modelling should not be blindly adopted to virtual architectures.
5. The unequal trends suggest that application specific customization is also beneficial for performance optimization, even though the effect might not be same as high as compared to area optimization.
6. Considering the variance of area-delay product in Figure 4.5, LUT sizes between K=5 and K=7 show the best trade-off between area and performance.

In conclusion, all these findings motivate to keep K parameterizable in *V-FPGA* in order to achieve a better fit of both, the intended application and the virtualization.

4.1.1.2. Connectivity to Routing Channels

Connection boxes around the CLB connect the inputs and outputs to tracks from the surrounding routing channels. Thereby, the number of tracks per routing channel is determined through the parameter W . The connection boxes consist mainly of multiplexers and the respective select signals also are controlled by configuration registers to allow a

programming of local interconnects. The connection boxes of the inputs (CBr) connect one track from the respective routing channel. The output can be connected to several tracks at the same time, increasing routing flexibility. Multiplexers of the respective output connection box (CBw) puts through either the output signal of the CLBs or the signal coming from the PSM (here from right to left). The flexibility of the connection boxes, both for inputs and outputs, is $F_C = W$, i.e. connections can be made with each track from the channel. It is also possible to make the connection flexibility $F_C < W$ or fractional (i.e. a relative fraction of W), yet choosing of $F_C = W$ provides the highest routing flexibility and is recommended in [15] for unclustered CLBs. Since W highly impacts routing area (which has the dominant share on total area) it is favourable to keep W low, however a too low W can lead to routing congestions and unroutable designs.

4.1.1.3. Configuration Unit

The contents of the LUT, the control signals for the bypass multiplexer $m1$ and the control signals for the connection boxes are stored in configuration registers that are programmed by a configuration unit. The configuration unit inside a CLB receives a serial bitstream during configuration and sets the configuration registers that control the programmable elements (LUT, bypass MUX, connection boxes) according to Table 4.2. The configuration size of a CLB depends on the channel width W and the LUT size K and is:

$$LEN(CLB_{conf.}) = 2^K + 1 + K \cdot \lceil \log_2(W) \rceil + W \tag{4.1}$$

Table 4.2.: Configuration register set for an unclustered CLB

Field	Length in bits	Description
LUT	2^K	K-input lookup-table (Solution set of implemented function).
ffen	1	Flip-flop enable (controls the bypass MUX $m1$). 0: Flip-flop is bypassed 1: Flip-flop is used
rr	$\lceil log_2(W) \rceil$	Controls the right side input connection box. Contains the binary number of the track from the routing channel, which is to be connected with the right side input.
tr	$\lceil log_2(W) \rceil$	Controls the top side input connection box. Contains the binary number of tracks from the routing channel, which is to be connected with the top side input.
lr	$\lceil log_2(W) \rceil$	Controls the left side input connection box. Contains the binary number of tracks from the routing channel, which is to be connected with the left side input.
br	$\lceil log_2(W) \rceil$	Controls the bottom side input connection box. Contains the binary number of tracks from the routing channel, which is to be connected with the bottom side input.
w	W	Controls the output connection box. Determines which tracks from the routing channel output will be connected. Each track number is assigned a seperate bit (e.g. bit position 0 for track0, bit position 1 for track1,...). A bit set to '1' leads to driving the corresponding track by the output signal of the CLB. Is it set to '0', the signal coming from the PSM (from the right side in Figure 4.3 is forwarded.

4.1.2. I/O Blocks

IOBs on the perimeter of the array work in a similar way like the connection boxes of the CLBs. As depicted in Figure 4.6, an IOB has exactly one input and one output. A multiplexer connects one of the tracks from the routing channel to the virtual output pad. In th eevent that an output is not occupied, the level '0' is issued through an AND gate connected to the MUX and to a respective configuration register bit *ren* that enables or disables the output. A signal at the virtual input pad can be connected in favour of a higher routability with several tracks at the same time. Respective 2:1 multiplexers forward either the input signal of the virtual I/O block or the signal coming from the PSM (here from right to left).

Analog to the CLBs the multiplexers of the IOBs are controlled by configuration registers, that are filled by the configuration unti during programming. Table 4.3 describes the contents of the configuration register of IOBs. The configuration size of an IOB is:

$$LEN(IOB_{conf.}) = 1 + \lceil \log_2(W) \rceil + W \tag{4.2}$$

Table 4.3.: Configuration register set for an IOB

Field	**Length in bits**	**Description**
ren	1	Enables the output. 0: output disabled (unassigned); Signal set to '0'. 1: output enabled (assigned to a track from the routing channel)
r	$\lceil log_2(W) \rceil$	Contains the binary number of the track that should be connected to the virtual output pad.
w	W	Determines the tracks that an IOB should drive. Each track number is assigned a seperate bit (e.g. bit position 0 for track0, bit position 1 for track1,...). A bit set to '1' leads to driving the corresponding track by the virtual input pad of the IOB. Is it set to '0', the signal coming from the PSM is forwarded.

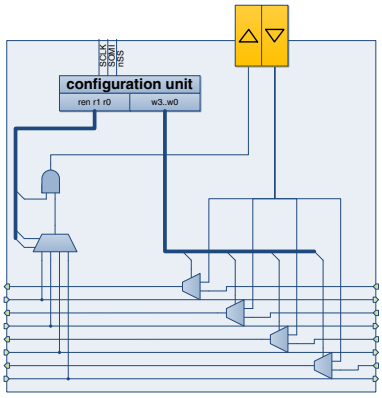


Figure 4.6.: Schematic of I/O block in *V-FPGA*

4.1.3. Programmable Switch Matrix

The Programmable Switch Matrices (PSMs) realize the routing of the signal paths by connecting tracks from different channels at the intersections. Therefore a 4:1 MUX is located at each output of a PSM as shown in Figure 4.7. On the left and bottom side of the PSM, the first position of the MUX is the logic level '1', which is the defined idle value of the routing infrastructure. This position is selected when an output track should not be connected to any other track, i.e. if there is no routing intended in this direction. The three remaining positions are associated each with an input from one of the three adjacent sites. The two select lines of the MUX are controlled by configuration registers, that are filled by the configuration unit during programming, thus the connections between routing channels are programmable. On the top and right side of the PSM, the inputs can be fed back to the outputs of the same sides by selecting the first position of the respective output multiplexers. This technique, which we call *loopback propagation* enables the emulation of bi-directional tracks using uni-directional tracks and is described in Section 4.1.4.

From Figure 4.7 and the fact, that only 4:1 MUX are used, it is clear that the PSM is not fully-connected. For each output there is only exactly one track per adjacent side that can be switched, i.e. the flexibility is effectively $F_s = 3$ (note that the first position of the 4:1 MUX is used for idle operation and loopback propagation, but not for routing). This choice is made for the purpose of area efficiency and bitstream reduction since the routing infrastructure has a significant share on area (see Figures A.1 to A.20 in Appendix A.1) and a fully-connected PSM would be by far too excessive.

There are three types of switch block structure supported, which can be selected through the parameter *SBtype* : *Wilton* [106], *Universal* [24] and *Disjoint* aka. *Subset* [108]. All have the same effective flexibility, yet the differences are in the patterns of track numbers that can be interconnected as defined in Table 4.4 to 4.6 and visualized in Figure 4.8. The *Disjoint* pattern is symmetric and allows connections only among tracks with the same track number, e.g. left-side track 1 can be connected only to top-side track 1 or right-side

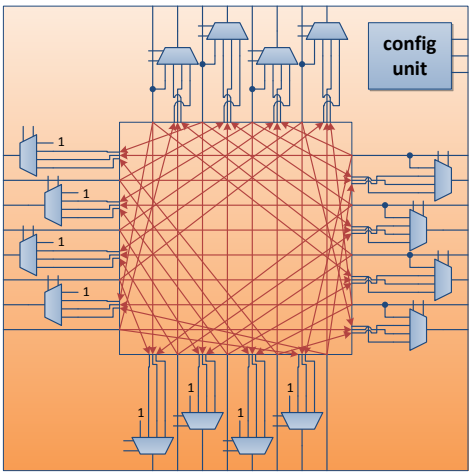
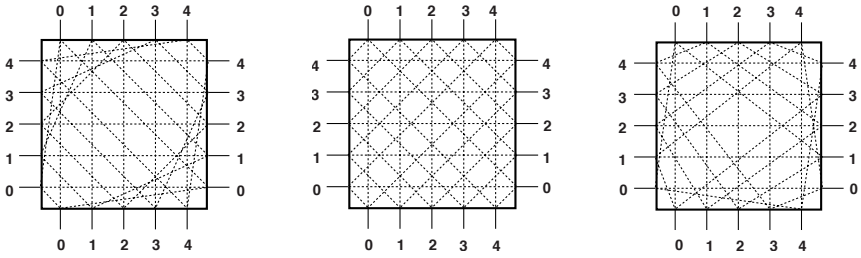


Figure 4.7.: Schematic of PSM in *V-FPGA*, here with Wilton structure

track 1 or bottom-side track 1. This leads to a partition of the complete routing infrastructure into disjoint routing domains. The routes are more predictable, yet the routability suffers from the limitation that complete paths are restricted to the same track domain. The *Universal* pattern, which was intended for maximizing the number of simultaneous connections, is asymmetric in a way that one of the diagonals can be connected to inverse track numbers, while the other two sides can be connected to tracks with the same track number. The *Wilton* pattern eliminates the domain limitation of the *Disjoint* pattern by rotating diagonal connections by one 1 track.

The effects of the switch block pattern, combined with other parameters such as LUT size and cluster size, on area efficiency and performance are studied in Section 4.1.5.2. There



(a) Disjoint (b) Universal (c) Wilton

Figure 4.8.: Structure of (a) *Disjoint*, (b) *Universal* and (c) *Wilton* switch block patterns [67]

exist further switch block structures, yet the choice to include the three mentioned above in the *V-FPGA* was made based on their popularity and more importantly because they are supported by the employed CAD tools for application mapping. However, this can be easily extended to other or custom switch block structures by modifying the patterns. As described in Table 4.7, the MUXs of a PSM have each two select signals which can be controlled by a configuration unit. The total configuration size per PSM is:

$$LEN(PSM_{conf.}) = 8 \cdot W \tag{4.3}$$

Table 4.4.: Pattern definition for *Wilton* switch block structure with channel width W and track number $i \in \{0, ..., W-1\}$

	top in	**right in**	**bottom in**	**left in**
top out	-	$(i+1) \mod W$	i	$(W-i) \mod W$
right out	$(W-1+i) \mod W$	-	$(W-2-i) \mod W$	i
bottom out	i	$(W-2-i) \mod W$	-	$(i+1) \mod W$
left out	$(W-i) \mod W$	i	$(W-1+i) \mod W$	-

Table 4.5.: Pattern definition for *Universal* switch block structure with channel width W and track number $i \in \{0, ..., W-1\}$

	top in	**right in**	**bottom in**	**left in**
top out	-	i	i	$W-1-i$
right out	i	-	$W-1-i$	i
bottom out	i	$W-1-i$	-	i
left out	$W-1-i$	i	i	-

Table 4.6.: Pattern definition for *Disjoint* switch block structure with channel width W and track number $i \in \{0, ..., W-1\}$

	top in	**right in**	**bottom in**	**left in**
top out	-	i	i	i
right out	i	-	i	i
bottom out	i	i	-	i
left out	i	i	i	-

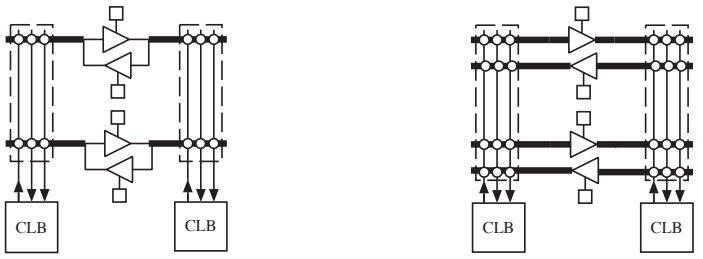
Table 4.7.: Configuration register set for a PSM

Field	**Length in bits**	**Description**
left(W-1)	2	Select signal-vector for left-side MUX on track *W*-1 "00": '1' (idle value) "01": signal from top side "10": signal from right side "11": signal from bottom side
bottom(W-1)	2	Select signal-vector for bottom-side MUX on track *W*-1 "00": '1' (idle value) "01": signal from left side "10": signal from top side "11": signal from right side
right(W-1)	2	Select signal-vector for right-side MUX on track *W*-1 "00": loopback propagation (see Section 4.1.4) "01": signal from bottom side "10": signal from left side "11": signal from top side
top(W-1)	2	Select signal-vector for top-side MUX on track *W*-1 "00": loopback propagation (see Section 4.1.4) "01": signal from right side "10": signal from bottom side "11": signal from left side
... ...		
left(0)	2	Select signal-vector for left-side MUX on track number 0 ...
bottom(0)	2	Select signal-vector for bottom-side MUX on track number 0 ...
right(0)	2	Select signal-vector for right-side MUX on track number 0 ...
top(0)	2	Select signal-vector for top-side MUX on track number 0 ...

4.1.4. Bi-directional Tracks

The routing of signal paths can be eased by bi-directional tracks, that can propagate a signal in both directions of a line as opposed to unidirectional wiring. In physical FPGAs bi-directional tracks are realized using two anti-parallel tristate buffers driving the same wire as shown in Figure 4.9a. On the other hand, the study in [60] shows that unidirectional single-driver wiring as shown in Figure 4.9b can reduce area and delay, since regular buffers require less transistors than tristate buffers and also the load capacitance is reduced. For the *V-FPGA* both techniques are considered.

While unidirectional wiring is a trivial matter, unfortunately bidirectional wires are naturally not possible with virtual FPGAs because the signal flow of configured underlying resources is always directional (except for the device I/Os that incorporate tristate buffers). A bi-directional track must therefore be modeled by two uni-directional wires. The propagation of a signal in both directions and the sharing of the track by a plurality of potential sources (e.g. 2 PSMs and 1 CLB) can then be implemented, for example, by an arrangement as shown in Figure 4.10, which enables the emulation of tristate signals by additional AND gates. However, since in virtual architectures the efficient use of underlying resources matters, another technique has been developed for the *V-FPGA* that uses less resources and is presented in Section 4.1.4.1. The pleasant specialty is that it allows the emulation of bi-directional wiring without additional resources. Furthermore, without modifications it can be used as uni-directional and as bi-directional track.



(a) Bi-directional wiring (b) Uni-directional wiring

Figure 4.9.: Routing tracks using (a) bi-directional or (b) uni-directional wiring [60]

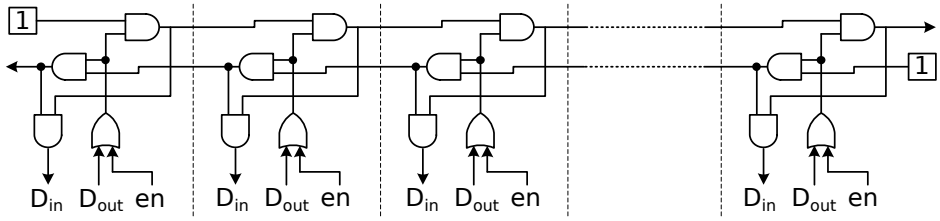


Figure 4.10.: AND based tristate emulation [54]

4.1.4.1. Loopback Propagation

Defining a track as a pair of two oppositely directed wires, the idea behind the presented technique, hereinafter called *Loopback Propagation*, is to feed a signal (e.g. from an output of a CLB) only into one of the two wires of a track and at the end of the track to replicate the signal onto the opposite direction. An example is depicted in Figure 4.11, where a signal on a wire in right-to-left direction can be replicated by a MUX in the PSM onto a wire in left-to-right direction as highlighted in red colour, carrying the signal in two directions simultaneously. Thereby the rules defined in the following must be obeyed:

1. Read and write accesses by connection boxes on a track occur on different wires, i.e. *CBw*s write (drive wires) only in one direction, while *CBr*s read from the opposite wires that carry the replicated signals.
2. The definition, in which direction the original signal is written and from which direction the replicated signal is read, must be applied globally to all tracks, e.g.:
   a) In horizontal tracks, always write on lines leading from right to left, and read from lines leading from left to right.
   b) In vertical tracks, always write on lines leading from top to bottom, and read from lines leading from bottom to top.
3. The idle value of unused wires must be '1'.
4. [A track is not driven by two sources simultaneously].

While the first three rules are strict, the last one depends on the capabilities of the CAD tools with respect to the routing steps. In case that the CAD tools support multimode tracks (i.e. a track with two wires can be seen either as one bi-directional track or as two separate uni-directional tracks), then the last rule may be violated in favour of a more efficient routing, allowing one wire of a track to be driven by a *CBw* and the opposite wire to be driven by an independent signal through the PSM that has been routed to the same track.

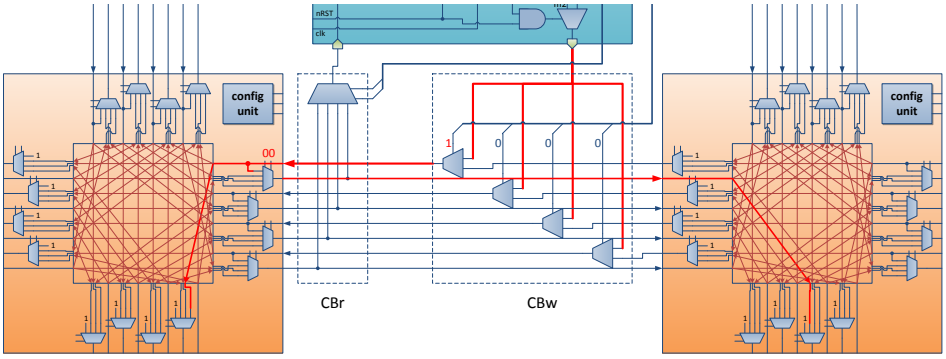
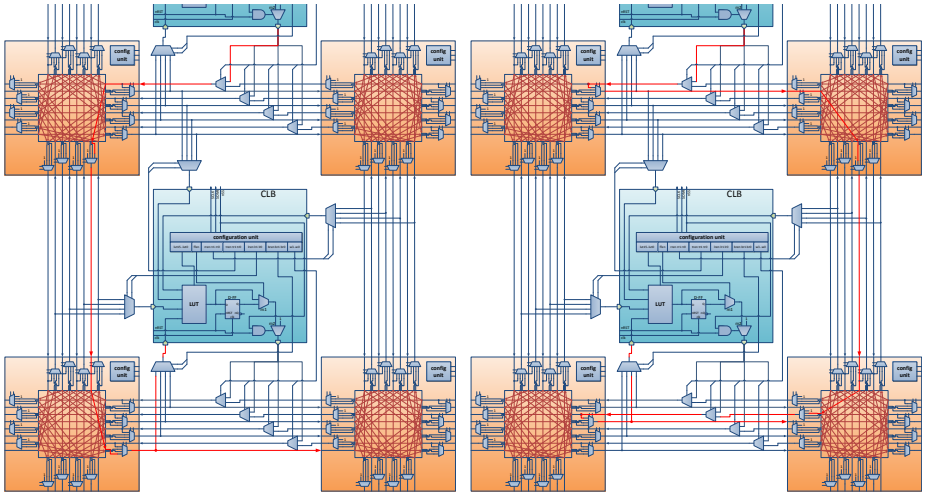


Figure 4.11.: Example of *Loopback Propagation* in bi-directional tracks



(a) Path A (b) Path B

Figure 4.12.: Connection between two CLBs through two different paths

As it can be seen in Figure 4.11, the ability to replicate a signal comes at no extra costs as the employed 4:1 MUXs in the PSMs are anyway needed for the routing. The difference is that on the right side and on the top side of a PSM the first position of a MUX selects not the idle value '1' but the incoming signal of the corresponding track, thus forming a loopback. Without *Loopback Propagation* the first position would be reserved for the idle value '1' which is selected when a track is unused. With *Loopback Propagation* the idle value of an unused track in right-to-left (or top-to-bottom) direction is replicated in left-to-right (or bottom-to-top) direction, thus having equivalent functionality for the idle case plus extending the functionality of normal operation by bi-directionality without additional resources. Compared to the version in [50] it saves $2W$ AND gates per PSM and compared to the tri-state emulation in [54] it saves W AND-gates per CLB-input and substitutes $2W$ AND-gates + W OR-gates by W MUX2 per CLB-output. The disadvantage of *Loopback Propagation* is that replicated signals have an additional delay since they need to pass through another MUX. This leads to asymmetric propagation delays depending on the direction of signal flow. For instance, Figure 4.12a and Figure 4.12b show two different paths for connecting two CLBs traversing the same number of routing channels, whereby path B has a longer delay than path A because it relies on signal replication to change the direction. CAD tools need either to be aware about this peculiarity by considering different wire models for different directions or alternatively one common worst case wire model (representing the loopback case) needs to be applied for all directions.

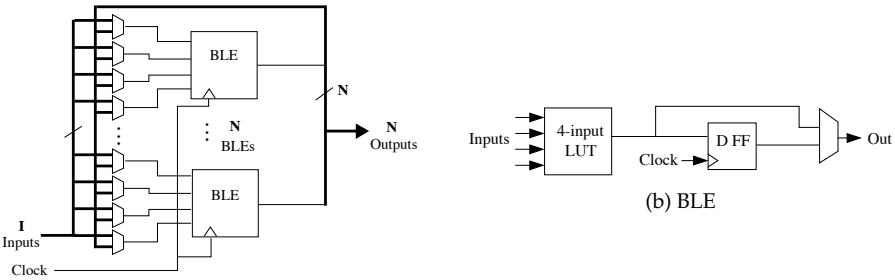
4.1.5. Clustering

Increasing the size of the LUTs is one way to reduce the routing complexity, as the application circuit is partitioned into fewer but larger logic blocks that need to be interconnected. However, the complexity of a LUT grows exponentially with the number of variables (i.e. inputs) and furthermore large LUTs suffer from low utilization ratios in practical applications where logic functions underly strong variance concerning their variable count. A more efficient way is to increase the size of a logic block by clustering several reasonably sized LUT + flip-flop pairs (called Basic Logic Elements (BLEs)) into one logic block and interconnect them by local routing as depicted in Figure 4.13. Compared to the approach of increasing the LUT size, clustering yields a higher flexibility and hence higher utilization because one logic block can realize either one large logic function or multiple smaller independent logic functions.

Further benefits of clustering are:

- improved locality of interrelated signals yields higher performance, e.g. in vector operations
- circuits with multiple operations on the same data (e.g. the half adder performs *a XOR b* for the sum and *a AND b* for the carry) experience a reduced number of total CLB ports and connection boxes
- when reasonably sized, routing complexity *can* be reduced due to hierarchical routing
- consequently reduction of routing area is possible

The downside of clustering, however, is that it requires additional input (and eventually also output) multiplexers inside CLBs, which can become quite large and add up on the path delay from CLB inputs to BLE inputs, depending on cluster size and LUT size, especially when a large multiplexer is composed of several 2:1 multiplexers. Thus, if not appropriately tuned, it can cancel all the promising benefits.



(a) Logic cluster

(b) BLE

Figure 4.13.: General structure of a logic cluster combining a collection of basic logic elements (BLE) that are flexibly interconnected by multiplexers [13]

To maximize the benefits of clustering, the following objective driven strategies are essential:

- try to pack connected LUTs together
- minimize the signals to be routed between logic blocks
- attempt to fill each logic block to its capacity in order to minimize the number of logic blocks

Clustering is essentially part of the partitioning problem in VLSI which has been studied extensively on netlist or graph level, e.g. in [46], [6] and [27]. Commercially, clustering was first used in Altera 8K and Xilinx XC5200 FPGAs. Betz et al. studied this technique further on architecture level in [13] to find out the optimal cluster size with 4-input LUTs and to determine a relationship between cluster size and required number of inputs per CLB. Later, Ahmed and Rose extended this study in [4] by varying both, LUT size K and cluster size N in the range of $K = 2..7$ and $N = 1..10$. Analyzing area and delay for the 20 largest MCNC benchmarks (and additionally 8 others) within these ranges, they came to the conclusions that $K = 4..6$ and $N = 3..10$ provide the best trade-off between area and delay. Furthermore, limiting the number of inputs per CLB to $I = K/2 \cdot (N + 1)$ is sufficient to reach a 98 % utilization and leads to area savings compared to a full $I = K \cdot N$ setup.

As for the *V-FPGA* presented in this work, clustering can have a significant effect on area and performance. While it is one of the most sensitive architectural parameters, we need to be aware that it can be a blessing or a curse, depending on how well it is tuned. The findings in [4] have become a widely accepted reference and guideline in parameter choice and the baseline for many academic FPGA architectures. However, for the *V-FPGA*, following these recommendations blindly might lead to a mistuning of architectural parameters and additional penalty in performance and area for the following reasons:

1. Ahmed et al. used in their experiments an area and delay model on transistor level including CMOS buffers, pass transistors, RC wire models, etc. based on SPICE simulations of a 0.18 μm CMOS process. However, the *V-FPGA* architecture is platform independent and in case of virtualization the base units are multiplexers and flip-flops realized by underlying logic blocks. Furthermore, while Ahmed et al. use a 6-transistor SRAM cell model for programming, the *V-FPGA* makes use of flip-flops. These differences can lead to unmatched proportions in logic area, local routing area and global routing area as well as in respective path delays, potentially invalidating the cost functions and consequently the quality of the packing (i.e. partitioning), placement and routing.
2. The architecture model used for the experimental results in [4] allows full permutation of CLB inputs through fully connected input multiplexers on each BLE and LUT input. This comes at the expenses of large area for the input multiplexers and respectively local routing. The *V-FPGA* architecture makes use of a LUT reordering technique for a more efficient full permutation, which cuts down the size of the input multiplexers by approx. K and reduces the path delay by approx. $\lceil log_2(K) \rceil$ without any penalty. Thus, relying on the results in [4] would not take full advantage on this improvement and again the proportions of local and global routing would be distorted.

3. The recommendations in [4] are based on averaging results across 28 benchmark circuits. While this will provide in average good results for general purpose, the customization approach presented in this thesis calls for a more application specific tuning of architectural parameters to increase efficiency.

For these reasons it is necessary to re-evaluate the LUT vs. cluster size problem with suitable area and delay models in order to take full advantage of the customization and optimization strategies followed in this work. Subsection 4.1.5.1 presents a generic and parameterizable clustering architecture for the *V-FPGA*. Section 5.6 provides area and delay models that are used in Section 4.1.5.2 to analyze the effects of varying K and N on area and delay of circuits from the MCNC benchmarks when mapped on the *V-FPGA*.

4.1.5.1. Generic Clustering Architecture for *V-FPGA*

The *V-FPGA* features a generic clustering architecture with parameterizable cluster size N and LUT size K as depicted in Figure 4.14. The union of a K-input LUT, a flip-flop and a bypass MUX forms a Basic Logic Element (BLE). A CLB contains N BLEs. As proposed in [4], a CLB with N BLEs of K-input LUTs contains $I = K/2 \cdot (N+1)$ inputs and $O = N$ outputs. There are two modes for the location pattern of the in- and outputs of a CLB: *auto* mode and *custom* mode. The *auto* mode aims an equal distribution on the perimeter of a CLB, as this improves routability. Starting on the bottom side with the first input, subsequent inputs are placed with each a clockwise rotation (bottom->left->top->right) from the previous one. After all inputs are placed, the procedure continues in the same way for the outputs. In custom mode it is possible to set the number of in- and outputs for each side of the CLB individually through the parameters I_left , I_top , I_right , I_bot , O_left , O_top , O_right and O_bot respectively.

Input multiplexers for each BLE input can select signals from the CLB inputs or any of the BLE outputs. Thereby, two versions are implemented selectable through the parameter F_inMUX_mode , one with fully-connected multiplexers (i.e. all CLB inputs and all BLE outputs are selectable by any BLE input) and another one where each input per BLE can connect only to a fraction of $1/K$ CLB inputs and to all BLE outputs. For an equal distribution of CLB inputs to LUT inputs, the latter version uses an overlap function if I/K is not an integer. The version with fractional input-MUXs is more area efficient than the version with fully-connected input-MUXs but is also more dependant on the outer routing.The multiplexers at the outputs of a CLB are optional through the parameter *F_outMUX_mode* and in case that they are employed they can select from any of the BLE outputs. They can slightly ease the outer routing, but cost additional area. It is possible to omit them and instead use a direct wiring of BLE outputs to CLB outputs, whereby the outer routing can be facilitated by a reordering of BLEs if the CAD tools support this optimization step. Each BLE also holds a configuration unit that sets the bits of the LUT and controls the bypass MUX. Additionally, there is a seperate configuration unit that controls the input- and output-multiplexers as described in Table 4.8 and 4.9. All configuration units within a clustered CLB are daisy-chained, starting with the configuration unit for the MUXs followed by the configuration units of the BLEs.

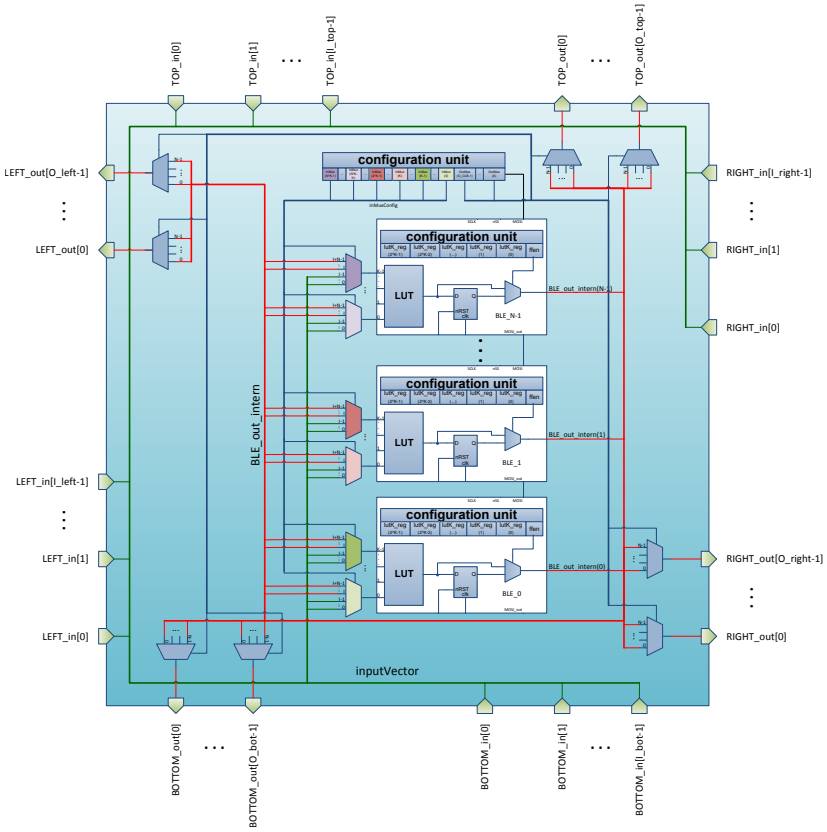


Figure 4.14.: Clustering of BLEs within a CLB of *V-FPGA*

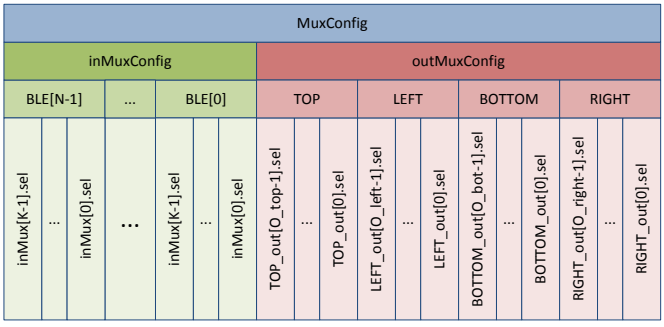


Figure 4.15.: Organization of configuration data for controlling input and output multiplexers in *V-FPGA* CLBs with clustering

Table 4.8.: Configuration bits for controlling the input and output multiplexers within a clustered CLB in fully-connected mode

Field	Length (Bit)	Description
MuxConfig	$N \cdot K \cdot \lceil \log_2(I + N) \rceil + O \cdot \lceil \log_2(N) \rceil$	This field controls all input and output multiplexers within a CLB. It is composed of the fields inMuxConfig and outMuxConfig which are described below.
inMuxConfig	$N \cdot K \cdot \lceil \log_2(I + N) \rceil$	This field controls all multiplexers at the inputs of the BLEs. There are N BLEs and each BLE has K input multiplexers.
outMuxConfig	$O \cdot \lceil \log_2(N) \rceil$	This field controls all multiplexers at the outputs of a CLB. There are N BLEs and each BLE has K input multiplexers.
BLE[N-1]	$K \cdot \lceil \log_2(I + N) \rceil$	This field controls all input multiplexers of the respective BLE. Each BLE has K input multiplexers.
...		
BLE[0]		
inMux[K-1].sel	$\lceil \log_2(I + N) \rceil$	This field controls the select signals of the respective $(I + N) : 1$ multiplexers at the respective BLE inputs. As depicted in Fig. 4.14, the multiplexer can select any CLB input or any BLE output to be fed the respective BLE input.
...		
inMux[0].sel		

Table 4.9.: Configuration bits for controlling the input and output multiplexers within a clustered CLB in fractional mode

Field	Length (Bit)	Description
MuxConfig	$N \cdot K \cdot \lceil \log_2(\lceil I/K \rceil + N) \rceil + O \cdot \lceil \log_2(N) \rceil$	This field controls all input and output multiplexers within a CLB. It is composed of the fields inMuxConfig and outMuxConfig which are described below.
inMuxConfig	$N \cdot K \cdot \lceil \log_2(\lceil I/K \rceil + N) \rceil$	This field controls all multiplexers at the inputs of the BLEs. There are N BLEs and each BLE has K input multiplexers.
outMuxConfig	$O \cdot \lceil \log_2(N) \rceil$	This field controls all multiplexers at the outputs of a CLB. There are N BLEs and each BLE has K input multiplexers.
BLE[N-1]	$K \cdot \lceil \log_2(\lceil I/K \rceil + N) \rceil$	This field controls all input multiplexers of the respective BLE. Each BLE has K input multiplexers.
...		
BLE[0]		
inMux[K-1].sel	$\lceil \log_2(\lceil I/K \rceil + N) \rceil$	This field controls the select signals of the respective $(\lceil I/K \rceil + N) : 1$ multiplexers at the respective BLE inputs, using overlapping of inputs for an equal distribution.
...		
inMux[0].sel		

4.1.5.2. Design Space Exploration for Optimized Tuning of Cluster Size and LUT Size

For adequate parameter tuning we re-evaluated the 20 largest MCNC benchmarks by extensive design space explorations spanning all combinations of LUT inputs in the range $K = 2..8$ and cluster sizes in the range $N = 1..10$. The employed DSE methodology is described in Section 6.4. The analysis required 1400 benchmark runs with the steps *logic optimization*, *LUT mapping*, *packing*, *placing* and *routing*. Logic optimization and LUT mapping were eased by reusing the pre-mapped versions of the MCNC benchmarks within the VTR package [104]. The steps of packing, placing and routing were done all with the academic tool VPR version 7.0 (contained in the open source VTR design flow package [63]) and with special architecture files representing the *V-FPGA* architecture and including area and delay models. Since LUT size K and cluster size N are specified in the architecture file and have effects on other parameters as well, there were 70 separate architecture files with combinations of K and N required. Those files, along with scripts for automated benchmark execution, were generated by the *V-FPGA* Explorer tool presented in this thesis (see Chapter 5, Section 5.4). Table 4.10 presents the best and worst solutions for the objectives of area and performance optimization, whereby Figure 4.16 and Figure 4.17 give an overview over the variance for all combinations of K and N within the design space for each benchmark. The dotted curves represent the average variances over all benchmarks. Since the benchmarks differ greatly in their size and structure, area and performance results are each expressed as variations relative to the median of the respective benchmark. More detailed results are presented in Appendix A.1, Figure A.1 to Figure A.20, where we can observe also the change in proportions of logic area, routing area and IO area with variation of K and N. The evaluation shows a strong parameter sensitivity with different and irregular trends. Up to $\pm$95.9% variance in area and up to $\pm$78.1% variance in performance are observed. This sensitivity is remarkably high considering that the circuit functionality and the resource types are the same and K and N determine only the granularity of logic blocks which has indirectly also an effect on the routing infrastructure. Furthermore, the different benchmarks differ greatly in their results, which leads to the conclusion that application specific customization can yield high optimizations, rather than relying on average values for the parameterization of the architecture.

Table 4.10.: Max. variance of area and performance in MCNC benchmarks when tuning cluster size N and LUT size K

	Best area			Worst area			Best performance			Worst performance			Best area-delay product			Area variance	Performance variance
	K	N	area	K	N	area	K	N	f_max	K	N	f_max	K	N	ADP		
alu4	4	7	4.56E+05	8	9	7.50E+05	8	9	7.71E+06	2	2	2.27E+06	6	7	7.11E-02	±24.4 %	±54.5 %
apex2	3	2	6.57E+05	8	10	1.34E+06	7	9	6.39E+06	3	1	2.63E+06	6	7	1.32E-01	±34.2 %	±41.7 %
apex4	4	6	4.73E+05	8	9	8.80E+05	7	10	7.15E+06	2	4	2.10E+06	6	5	8.66E-02	±30.0 %	±54.7 %
bigkey	2	1	4.93E+05	8	10	2.20E+07	8	2	9.48E+06	2	1	3.78E+06	4	1	1.04E-01	±95.6 %	±43.0 %
clma	5	1	3.29E+06	8	10	6.33E+06	8	10	4.03E+06	2	1	9.76E+05	7	10	1.34E+00	±31.5 %	±61.0 %
des	2	1	7.42E+05	8	10	3.13E+07	7	4	4.88E+06	2	1	2.67E+06	3	1	2.23E-01	±95.4 %	±29.3 %
diffeq	3	1	3.65E+05	8	10	1.53E+06	7	6	9.77E+06	2	1	2.33E+06	6	5	4.43E-02	±61.4 %	±61.5 %
dsip	2	1	4.76E+05	8	10	2.25E+07	5	2	9.30E+06	2	2	4.06E+06	4	1	1.06E-01	±95.9 %	±39.2 %
elliptic	3	4	1.10E+06	8	10	1.01E+07	8	9	5.57E+06	2	1	1.54E+06	5	1	2.96E-01	±80.3 %	±56.8 %
ex5p	8	1	6.05E+04	8	10	5.90E+05	8	2	2.29E+07	4	1	2.82E+06	8	1	2.80E-03	±81.4 %	±78.1 %
ex1010	8	5	7.34E+05	3	3	2.50E+06	8	10	8.84E+06	2	1	1.12E+06	8	5	1.08E-01	±54.6 %	±77.4 %
frisc	2	4	1.34E+06	8	10	3.37E+06	7	5	5.53E+06	2	1	1.27E+06	5	5	3.35E-01	±43.0 %	±61.4 %
misex3	4	5	4.59E+05	8	9	9.37E+05	7	10	7.61E+06	2	1	2.47E+06	6	7	7.12E-02	±34.3 %	±50.9 %
pdc	6	1	2.16E+06	8	1	3.15E+06	7	10	5.28E+06	2	1	1.07E+06	7	10	4.87E-01	±18.7 %	±66.2 %
s298	4	5	5.41E+05	8	9	1.05E+06	8	9	6.22E+06	2	1	1.64E+06	7	10	1.28E-01	±31.9 %	±58.2 %
s38417	6	2	1.27E+06	2	8	3.60E+06	7	5	8.81E+06	2	1	2.01E+06	6	4	1.86E-01	±47.7 %	±62.8 %
s38584.1	5	1	1.58E+06	8	10	1.81E+07	6	9	7.17E+06	2	1	2.67E+06	7	2	3.46E-01	±83.9 %	±45.8 %
seq	4	5	6.02E+05	8	9	1.24E+06	8	10	7.49E+06	2	1	2.20E+06	6	6	1.02E-01	±34.8 %	±54.7 %
spla	4	1	1.57E+06	8	9	2.61E+06	7	10	5.29E+06	5	1	1.73E+06	7	10	4.08E-01	±25.0 %	±50.7 %
tseng	3	2	2.25E+05	8	10	4.35E+06	7	4	1.02E+07	2	1	2.26E+06	5	1	4.42E-02	±90.2 %	±63.8 %

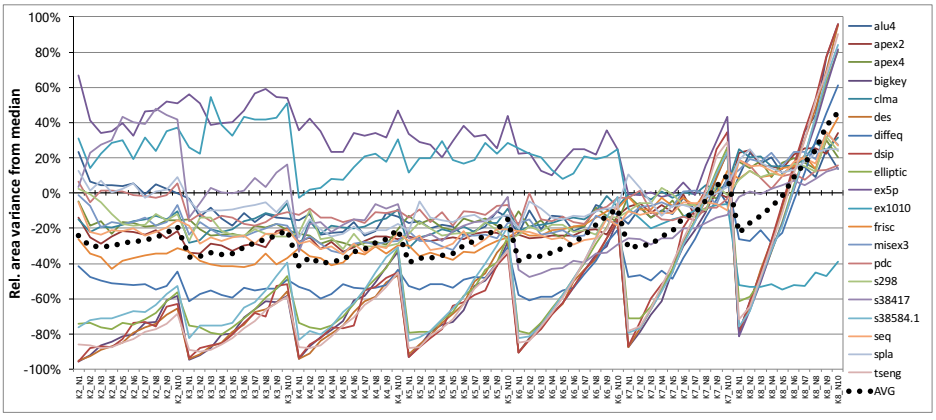


Figure 4.16.: Effects of Lut size K and cluster size N on area, shown through variance of area relative to median of series for combinations of K and N

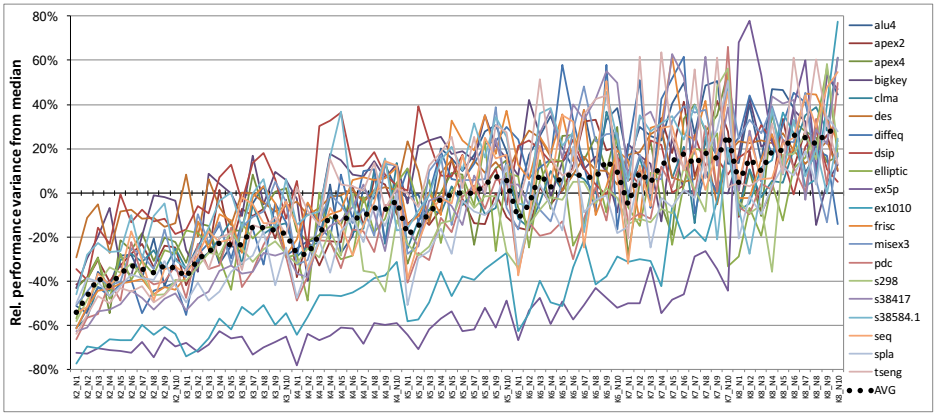


Figure 4.17.: Effects of LUT size K and cluster size N on performance, shown through variance of performance relative to median of series for combinations of K and N

4.1.6. Tile Based Structure

The *V-FPGA* is developed in mainly structural VHDL making use of modularization, hierarchy, automated instantiation and wiring by the means of GENERATE loops and parameterization through GENERICS. In Figure 4.18 a reduced hierarchy diagram shows the composition of components out of subcomponents of the next lower hierarchy step.

The leafs are MUX2 (2:1 multiplexer), MUX4 (4:1 multiplexer), AND2 (2-input AND gate) and DFF (D-flip-flop). On a higher hierarchy level, the *V-FPGA* structure is partitioned into recurring tiles as illustrated in Figure 4.19. There are 9 types of tiles of which the *V-FPGA* is constructed: one for the inner area, 4 for the borders and 4 for the corners. Compared to a flat structure, the use of tiles has a number of benefits:

- The synthesis process is faster when preserving hierarchy because once a tile is synthesized the same netlist can be reused for all instances of the same tile.
- In case of physical implementation, a tile can be placed & routed as a macro block.
- The problem of combinational loops in static timing analysis is mitigated since the analysis can be performed per tile/macro whereby the portion of routing channels present in a tile can not form loops (in contrast to an overall flat structure).
- The reuse of macros in physical implementation not only reduces the design time but offers also a higher regularity with more homogenous characteristic throughout the chip area.
- The characterization effort is reduced.
- Homogeneous and unified timing and area characteristics and accordingly derived delay and area models ease the place & route efforts for application mapping.

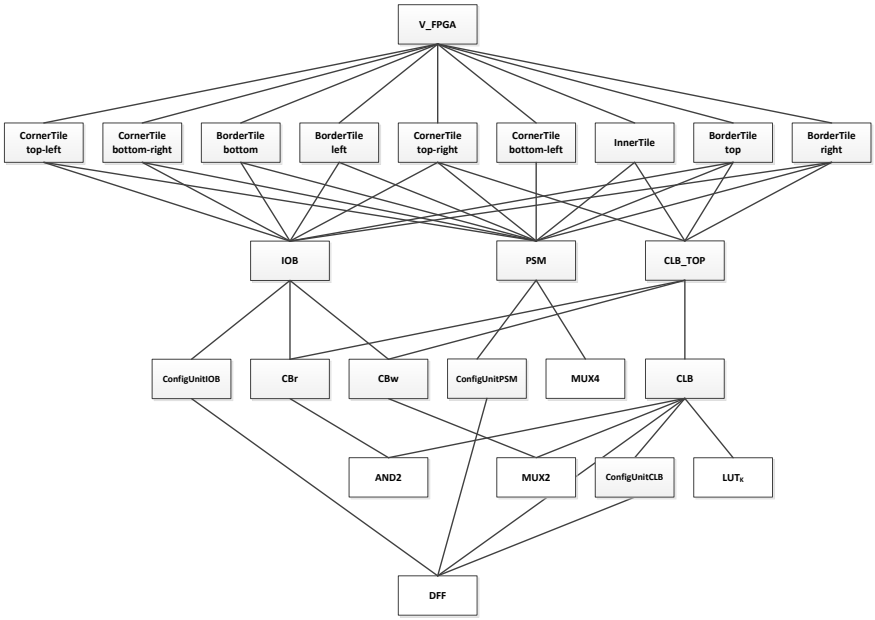


Figure 4.18.: Hierarchy of *V-FPGA* components and subcomponents

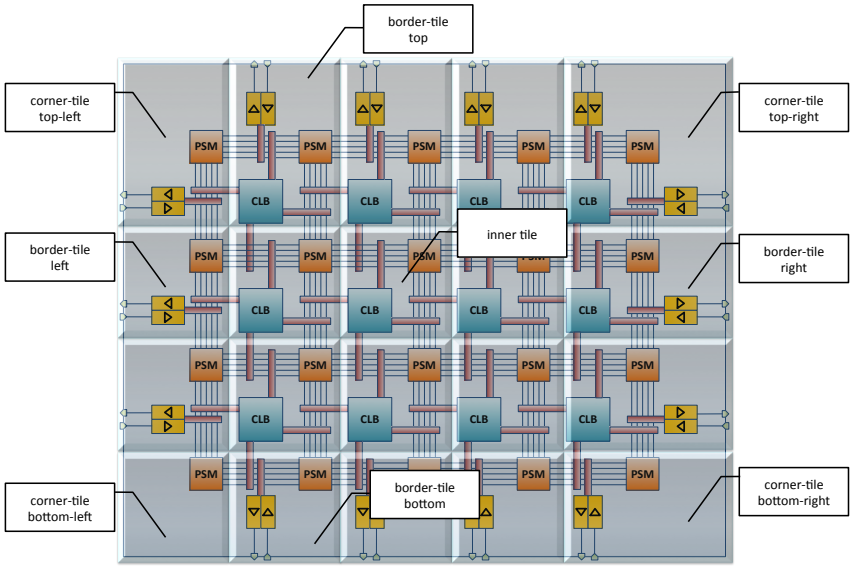


Figure 4.19.: Partitioning of *V-FPGA* structure into recurring tiles

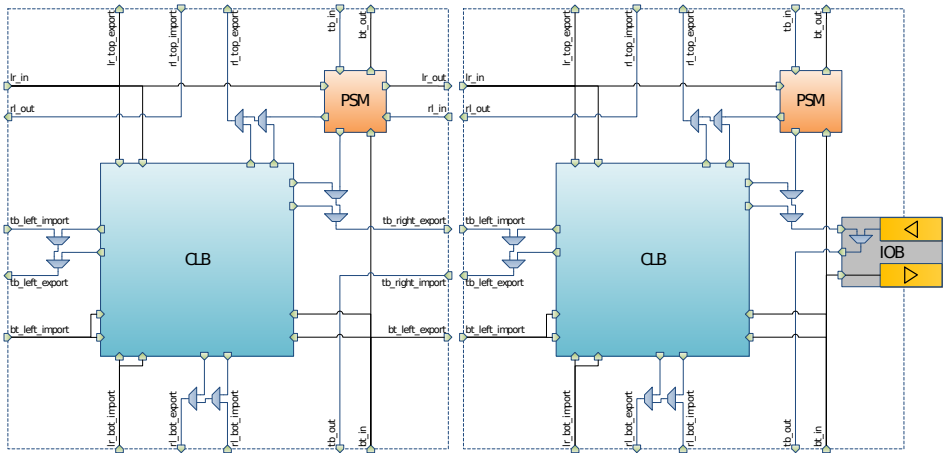


Figure 4.20.: Routing channels within tiles

Figure 4.20 shows a more detailed view of routing channels within tiles. Apart from the channels flowing in and out of PSMs there are so-called import- and export-channels between tiles. These channels are needed because a routing channel from a PSM is shared by two tiles and the export- and import-ports allow to connect a routing channel within one tile also by a neighbouring tile.

4.2. 3D Extension

The interconnect challenge imposed by increased total wirelength and dominating RC delays in highly integrated circuits calls for new routing architectures exploiting the third dimension. There is a promising trend towards 3D FPGAs with the aim to vertically stack and interconnect homogeneous or heterogeneous layers (each on a separate die), thereby reducing the total wirelength, increasing yield, mixing technology nodes (each layer can be manufactured in a different process and feature size), and/or increasing the total area within a chip. This has scalability, performance, and power benefits as discussed in [80], [94] and in the related works [70], [33], [62], [113], [43], [84], [99] and [97], which are briefly described in Section 3.5.

The related works show only a few ways of designing 3D-FPGA architectures. In fact the new dimension and the added degrees of freedom largely extend the architecture design space, which can be considered mainly unexplored despite existing works and experiences from 3D memories. With the vertical stacking of FPGA layers also new challenges arise, such as e.g. heat dissipation of inner layers, limited number of vertical connections, efficient partitioning, placement & routing in 3 dimensions, task migration, heat balancing and runtime management, etc.. These aspects can be influenced by the architecture

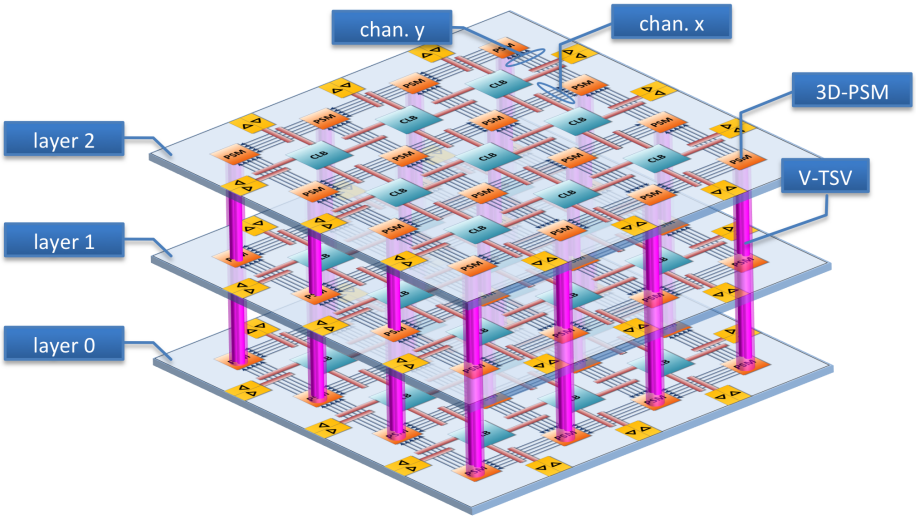


Figure 4.21.: Structure of 3D *V-FPGA*

of a 3D FPGA, and therefore customization becomes important, especially since it is an emerging research field.

4.2.1. The customizable 3D *V-FPGA* Architecture

The *V-FPGA* framework allows the generation of custom 3D FPGA architectures composed of several stacked layers of 2D FPGAs that are interconnected by the means of Through-Silicon Vias (TSVs) as illustrated in Figure 4.21. Therefore, the PSMs are extended by vertical connections to establish paths between layers through 3D-PSMs. While TSVs are passive elements with bi-directional signal flow, the so-called Virtual Through-Silicon Vias (V-TSVs) in the *V-FPGA* are modelled as union of two antiparallel uni-directional vertical wires. The structure of a 3D-PSM with V-TSVs is illustrated in Figure 4.22. For the purpose of clarity only one output multiplexer is shown. The colour gradients from red to blue are meant to visualize the directions of signal flows through the PSM-internal wires.

On the first sight the connectivity pattern of this approach might look different from the existing 3D switch boxes mentioned in the works above and the question might be *"why not simply adopting the existing structures?"*. This design choice was made for the following reasons:

- In case of virtualization the approach in Figure 4.22 is more area efficient because a vertical connection itself doesn't need switches. Instead the flexibility is achieved

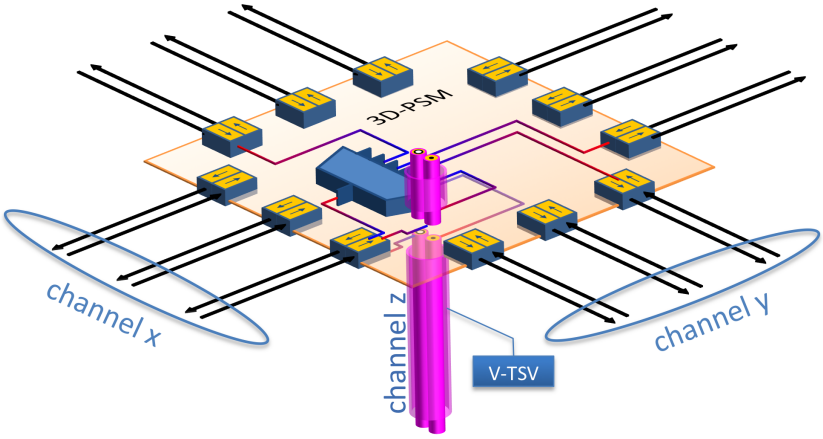
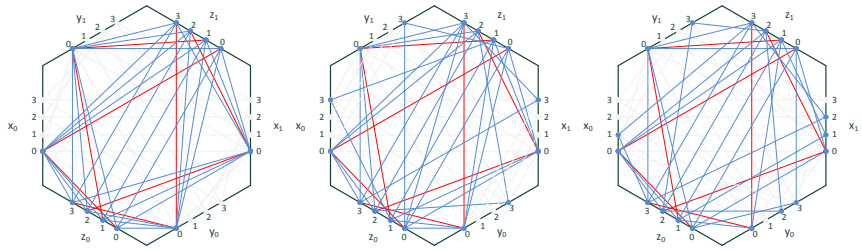


Figure 4.22.: Simplified scheme of 3D-PSM in *V-FPGA*

by the output MUXs of the 3D-PSM. Note that a 3D-switch has a 2.5x area-increase over a 2D-switch, while the 6:1 MUXs in the 3D-PSM have only a 1.67x area-increase over the 4:1 MUXs in the 2D-PSM. Furthermore, the virtualization of single switches is overall less efficient than the virtualization of MUXs.

- The CAD routing- and bitstream-generation-efforts presumably can be reduced as the absence of switches in vertical connections mean a reduction of switchable segments.
- The 3D PSMs are suited not only for true 3D-, but also for 2.5D- and 2D-architectures. In the latter case, the V-TSVs can be seen as long-lines that provide fast connections between regions of the array.
- Even though it might be not obvious at first sight, the combination of the employed 3D-PSM approach in conjunction with other techniques such as *Loopback Propagation* or in-outs leads to a similar flexibility ($F_S = 5$) as the existing 3D SBs. This becomes clearer when we have a look at the possible resulting interconnect pattern graphs in Figure 4.23a, 4.23b and 4.23c. For the sake of clarity, horizontal connections are faded in order to give highlight to the vertical connections, which are represented by red lines (direct vertical connections) and blue lines (indirect vertical connections).

The 3D *V-FPGA* extends the parameter set by two additional parameters. The parameter L determines the number of stacked layers and the parameter $TSVpC$ refers to the number of V-TSVs per channel (per PSM side). Starting with track number 0 the TSVs are allocated in ascending order of track number until $TSVpC$ is exhausted. Alternative to the $TSVpC$ parameter there is a hidden parameter *int_sel* for the 3D-PSMs that enables to set arbitrary location patterns for TSVs. This is useful when a heterogeneous interconnect fabric is envisioned (e.g. such as in [97]). In Figure 4.23 the parameters $W = 4$ and $TSVpC = 1$



(a) (b) (c)

Figure 4.23.: 3D-PSM connectivity patterns based on (a) 2D Disjoint, (b) 2D Universal and (c) 2D Wilton origin horizontal patterns

are selected along with $SBtype = 2$ for Disjoint, $SBtype = 1$ for Universal and $SBtype = 0$ for Wilton. For better compatibility to existing 3D SBs the parameter *int_sel* needs to be employed customizing the vertical pattern style.
All the 18 parameters of the 2D *V-FPGA* persist also in the 3D *V-FPGA*, offering in total 20+1 ways (and thousands of combinations apart from scaling) to tune the architecture, which gives a large room for exploration and customization second to none of the related works identified in Chapter 3.

4.3. Configuration Mechanisms

The *V-FPGA* is (re)configured by loading bitstreams into the configuration registers of the programmable resources. Thereby, the *V-FPGA* supports partial and dynamic reconfiguration at coarse as well as at very fine granularity. Furthermore, it incorporates special features such as fast dynamic task migration and defragmentation at run-time. The mechanisms and the trade-offs are described in the following subsections.

4.3.1. Configuration Units

The reconfigurable resources in the *V-FPGA* come each with an own configuration unit. A configuration unit contains storage elements and logic for the reception and transport of configuration data. As shown in Figure 4.24, the storage elements are realized by flip-flops in case of virtual FPGAs as these are the available bit-wise storage resources of the underlying COTS FPGA platform. The flip-flops within a configuration unit are daisy chained and form a shift register. Daisy-chaining of multiple configuration units extends the shift register. This yields a very area efficient transport mechanism as no additional resources are employed for this purpose apart from the storage elements. The transport and storage of configuration data is controlled through clock-gating of the flip-flops. During a (re)configuration the clock is enabled and configuration data as a bitstream is pushed serially into the first flip-flop of the daisy-chain through a single Master Out Slave In (MOSI). At the same time the nSS (low-active slave-select) signal is driven low. With ev-

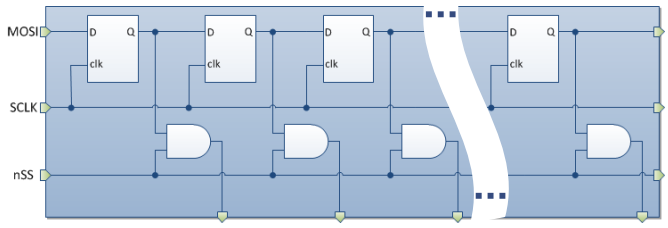


Figure 4.24.: Schematic of a configuration unit within the *V-FPGA*

ery clock-cycle a bit is shifted one step further in the daisy-chain. When all bits are shifted in, the transport is stopped by gating the clock and the flip-flops hold their values. Since each flip-flop controls something in the programmable resources of the *V-FPGA*, shifting bitstream through the daisy-chain would lead to glitches and arbitrary activity in the *V-FPGA* during a reconfiguration. To avoid this and to keep the *V-FPGA* in a defined state, the outputs of the configuration registers are conjunct with nSS by AND-gates and thus driven low during programming.

4.3.2. Configuration Controller

The task of the configuration controller is to retrieve configuration data from memory and distribute it to the configuration units of the CLBs, PSMs and IOBs, which fill the corresponding configuration registers. Since the daisy-chained configuration units adopt a standardized interface protocol, the configuration controller for the *V-FPGA* can be user defined. This gives more flexibility to embedded systems. It is also possible to build the configuration controller completely in software, utilizing Serial Peripheral Interface (SPI) peripherals of a microcontroller or GPIOs, yet the best performance is achieved with hardware solutions.

The *V-FPGA* framework contains a reference design of a hardware configuration controller that can be interfaced by microprocessor cores through standardized Advanced Microcontroller Bus Architecture (AMBA) busses. The configuration controller uses three separate RAM blocks for configuration data of the CLBs, PSMs and IOBs. These RAM blocks are part of the underlying target technology, and are integrated as hardware macros. Therefore, unlike the rest of the *V-FPGA*, the memory part of the configuration controller is platform-dependent. For porting to other underlying platforms, the instances of the RAM blocks used here must be replaced by instances of the RAM macros supported by the target platform. Fortunately, the RAM blocks of different target platforms are very similar with respect to their interface, which simplifies the efforts of migration to the creation of simple wrappers for the memory modules per platform. The communication with the memory controller takes place via an AMBA Advanced High-Performance Bus (AHB), whereby the configuration controller is connected as master. If the memory is shared between several system components and has only one port, a bus arbiter is required to grant access from multiple masters to the memory controller. However, the

Table 4.11.: Instruction set of configuration controller

Adress-Offset	Type	Name	Description
0x0000	write	CON	Selects a configuration and starts the (re-)configuration process
	read	ST	Status register
0x0800 ... 0x0BFC	write	CRAM	Writes data in CLB RAM
	read	-	Reserved
0x1000 ... 0x13FC	write	PRAM	Writes data in PSM RAM
	read	-	Reserved
0x2000 ... 0x23FC	write	IRAM	Writes data in IOB RAM
	read	-	Reserved

underlying platforms support dual-ported RAM blocks, which eliminates the need of an arbiter in setups with two masters. The configuration controller also includes an AMBA APB slave interface through which it can be controlled externally (e.g., by a microprocessor core). For instance, via this interface, configuration data can be copied to the RAM blocks from outside, configuration processes can be started and status information can be queried. Table 4.11 shows the supported instruction set of the interface. For writing the configuration data to the RAMs, the configuration controller contains a bridge that decodes the APB addresses and writes the data to the correct RAMs, thereby utilizing the already existing APB signals that can be reused for the RAM interfaces.

When sending a word to the address 0x0000, the signal $reconf$ is set to '1'. This starts the reconfiguration process, which is controlled by three Finite State Machines (FSMs) (one each for CLBs, PSMs and IOBs) as shown in Figure 4.25. In the idle state, the FSMs wait for the reconfigure command ($reconf =' 1'$). Then they jump to the *start* state, where they check whether any of the FSMs are still in an old configuration process that has not yet been completed. If this is not the case, the FSMs jump to the *load* state. Thereby, the signals *clb_nPROG*, *psm_nPROG* and *iob_nPROG* are set to '0' by the individual FSMs. As a result, the global low-active slave select signal nSS of the virtual FPGA core is also set to '0', thus the configuration units in the CLBs, PSMs and IOBs are ready to receive new configuration data, and the anti-glitch logic in the CLBs is active. In the *load* state, the read-enable signal of the corresponding RAM block is set to '1' for each FSM and is held until all configuration data have been transferred. As a result, a new word can be read from the RAM with each clock. Addressing takes place via variable b, which is incremented in each clock cycle. The (read) data bus signal of the RAMs lead directly to the data signals of the configuration bus. The arrangement of the bits in the data words is selected in such a way that a 1:1 assignment of the bit positions to the column numbers of the virtual FPGA array takes place. The bit streams on the individual data lines of the configuration bus are thus created by reading the data words from the RAM one after the other. In case the *V-FPGA* has more columns than the data width of the RAMs, the easiest solution is to segment the *V-FPGA* and daisy-chain the segments. This way there is no limitation in the scaling of the array. The read-enable signals of the RAM block are also used to gate the configuration clocks. Clock gating is needed control the shifting of the

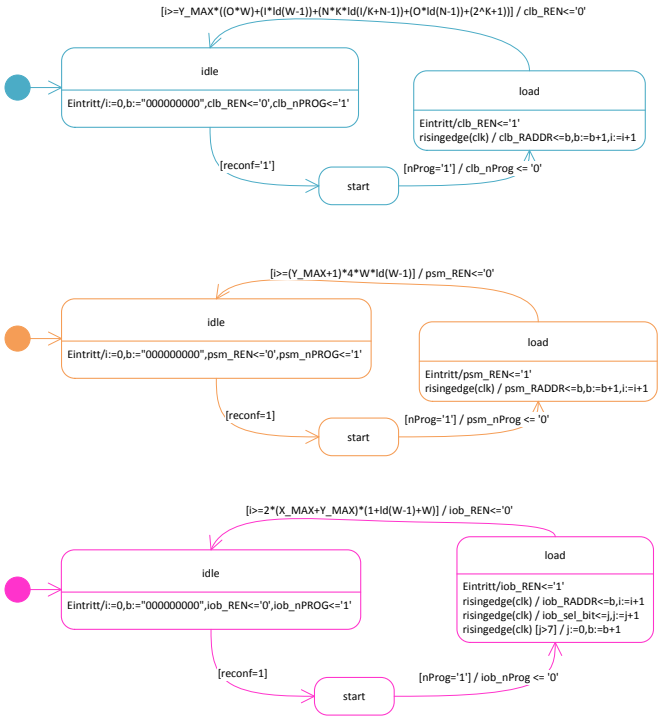


Figure 4.25.: Finite State Machines of bitstream loader

bitstreams through the configuration registers. If the amount of data to be transmitted is reached in the *load* state by the counter variable i, then the respective FSM jumps to the *idle* state and the read enable signal of the corresponding RAM is reset to 0, which stops the reading from the RAM and also the configuration clock by clock-gating. In the idle state, the corresponding $*\_nPROG$ signal is then set to '1' again. If all three FSMs have reached the idle state, all three $*\_nPROG$ signals are '1' and so also the global low-active slave select signal. This completes a reconfiguration.

Figure 4.26 shows a simulation of the configuration process as described above. At the marker position the configuration controller receives the *CON* instruction (address 0) with the data word 0x01, which tells the configuration controller to configure the array with the first configuration in the memory. Then the state machines read data from the memories and drive the configuration signals accordingly. The configuration related signals are marked with three different colours, cyan for the CLB columns, orange for the PSM columns and magenta for the IOB ring. The beginning and the end of the bitstream loading for each resource type is marked by the events in the $nPROG$ signals. The configuration clocks are gated accordingly.

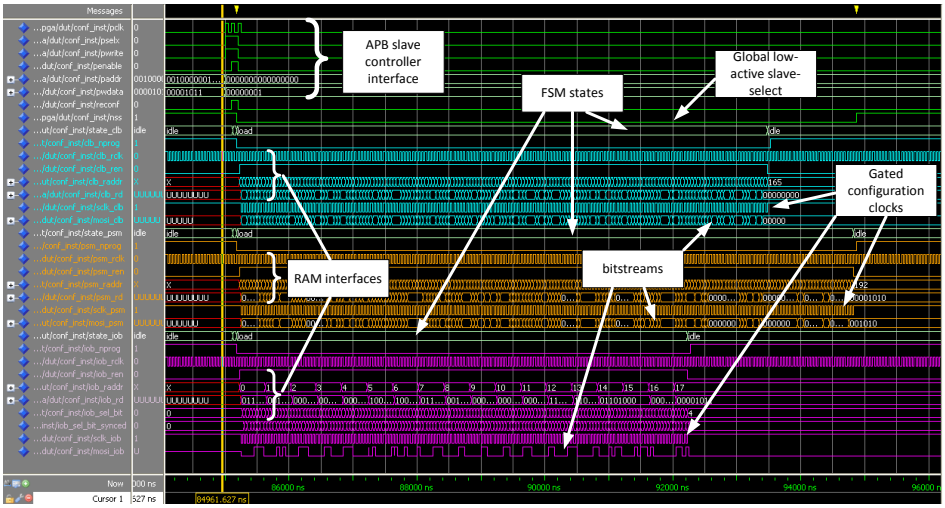


Figure 4.26.: Simulation of a configuration process

4.3.3. Area vs. Speed Trade-offs in Configuration Infrastructure Topologies

The topology of the configuration infrastructure can be tuned by splitting and combining the configuration daisy-chains. In dynamically reconfigurable FPGAs there is a serial/parallel trade-off to be considered when thinking about configuration infrastructure. Runtime reconfiguration has to be fast and from this point of view a parallel bus seems to be the best choice. But a purely parallel bus occupies a lot of area and resources especially in virtual architectures. If you take into account that this occupied area has a very low utilization while it degrades the system's routability, a purely parallel solution should be avoided when possible. The virtual FPGA is programmed in a semi-parallel manner as visualized in Figure 4.27. This means that all elements of the same type within a column are programmed serially but all columns (and all different element types) have separate serial busses which are driven concurrently by the configuration controller. The configuration speed depends on the amount of rows whereas the occupied bus area depends on the amount of columns. By varying the parameters X (columns) and Y (rows) in the top level of the RTL, the trade-off can be tuned in favor of speed or area [50].

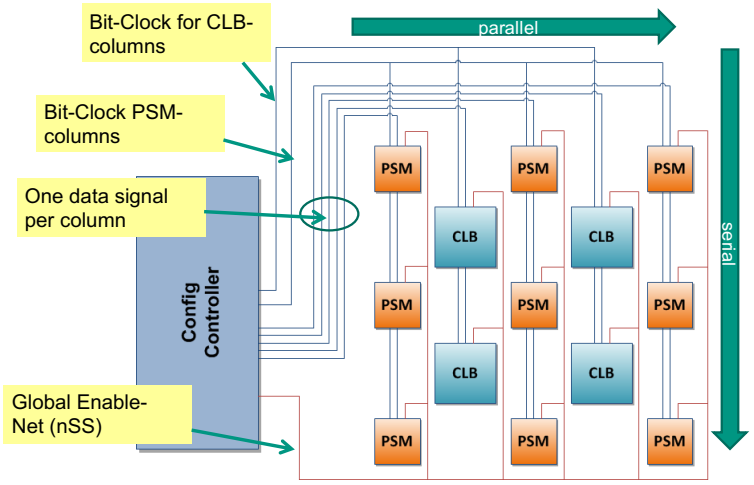


Figure 4.27.: Semi-parallel configuration infrastructure in *V-FPGA*

4.3.4. Coarse-Grain Partial Reconfiguration

Partial Reconfiguration (PR) is needed in order to reconfigure a part of the *V-FPGA* while the other parts are operating and should not be disturbed. Thereby, in many cases it is needed to reconfigure large contiguous areas rather than individual spots. An efficient way to enable coarse-grain partial reconfiguration at minimum costs is to split the *V-FPGA* area into smaller independent *V-FPGA* cores that can be configured individually. Thereby each core would accommodate an application and multiple cores would execute multiple applications in parallel. However, this is only efficient if the cores closely matches the application size. In practical cases this not given as different applications (or circuits) have different area requirements and are not equally sized. In such cases the size of all cores would be chosen based on the largest application that should fit in a core, which however would lead to unutilized area in cores that execute smaller applications. To overcome this problem and increase area efficiency the *V-FPGA* features a unique technique called *Core Fusion* [35]. The idea is to partition the *V-FPGA* into relatively small cores and to merge adjacent cores to a bigger core on demand whenever an application wouldn't fit into one as illustrated in Figure 4.28.

This technique is basically realized by adding routing channels between the outer PSMs of two adjacent cores to extend signal paths from one core to another. As a result of using the same channel width and the same routing structure inside the PSMs there are no bottlenecks between the merged cores. The routing and the behavior are quite the same like one single natively double sized core would be used, except that there is a middle stripe without CLBs.

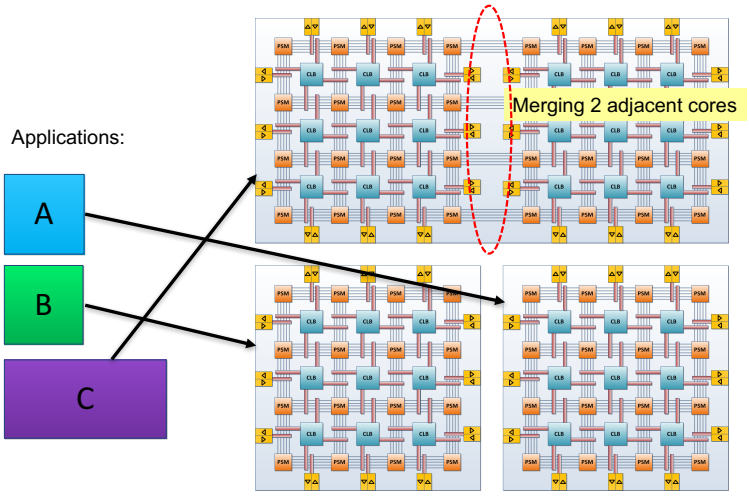


Figure 4.28.: The *CoreFusion* idea, merging adjacent *V-FPGA* cores [35]

Thanks to *CoreFusion* the size of the cores doesn't need to depend on the applications as it can be adapted on demand by merging. Instead the size can be selected based on the desired granularity of partial reconfiguration.

4.3.5. Fine-Grain Partial Reconfiguration

To maximize flexibility in dynamic application mapping and to speed-up context-switching, the *V-FPGA* supports slice-level partial reconfiguration. This unique feature allows to dynamically re-configure single resources (BLEs, CBs, PSMs, IOBs) during runtime while the others are still operating. In contrast to state-of-the-art devices that can perform partial reconfiguration only at the coarse granularity of entire frames, the very fine grained partial reconfiguration of the *V-FPGA* allows:

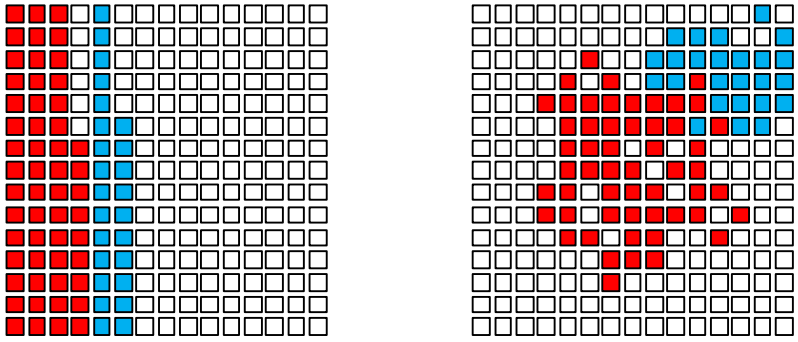
- **arbitrary aspect ratios** for application mapping, as PR regions are not limited to rectangular vertical frames (Figure 4.29a), but are rather free-form (Figure 4.29b).
- **better routability and shorter paths**. It is clear that the aspect ratio has an impact on the routeability and path delays. With slice-level PR this is optimized by timing-driven free-form placement.
- **faster context switching**, as fewer resources need to be reconfigured compared to entire frames.
- **higher utilization** due to finer area quantization. In state-of-the-art devices, if a frame is not fully utilized by an application, the rest of it is waste. Not so with slice-level reconfiguration, where other applications can utilize the rest.

Slice-level PR is very powerful in conjunction with the Just-in-Time Place&Route methodology described in Section 5.5. It allows to change form-factors and locations of applications during runtime, depending on the momentary distribution of free resources when the application is requested.

Architecture wise, slice-level reconfiguration is enabled by additional by-pass selector logic in the configuration units of the *V-FPGA*. The idea is to maintain the same configuration infrastructure, but to completely bypass the registers of a configuration unit if it is not selected for configuration. This is achieved by the circuit shown in Figure 4.30.

Based on the nSS signal the circuit enters in one of two modes. If $nSS =' 1'$, the configuration unit is in an idle phase. During this phase the configuration registers are disabled by driving the $conf\_en$ signal low and hold their Q value irrespective of signal changes on the D-input. This ensures that running applications on the *V-FPGA* are not disturbed by signals on the configuration bus. Nevertheless, during this phase a bitstream can be fed into the configuration unit to configure a by-pass MUX. This type of configuration goes through a seperate flip-flop that is enabled only during this phase. The output Q of this flip-flop drives the signal $conf\_en\_stdby$ which is forwarded to the output MUX of the configuration unit in order build a daisy-chain with the other configuration units. When $nSS =' 0'$ this separate flip-flop is disabled and holds its Q value, while the configuration unit is in an active phase. If $conf\_en\_stdby =' 1'$ in conjunction with $nSS =' 0'$, then the configuration registers are enabled for configuration. Otherwise, they are disabled and by-passed by the incoming bitstream. This way, only selected resources are re-configured while the by-passed ones keep doing their business and are not interrupted.

As it can be seen in Figure 4.30, the hardware overhead for slice-level reconfigurability is very low. The configuration bus stays the same, only the configuration units are extended by additionally one flip-flop, one inverter, one AND-gate and one 4:1 MUX.



(a) (b)

Figure 4.29.: PR regions and the effects on placement: (a) rectangular regions in frame-granular PR, (b) free-form in fine-grained slice-level PR

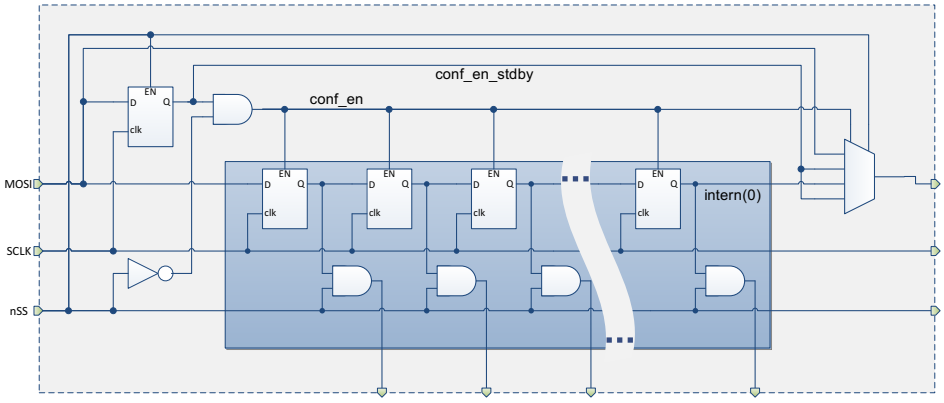


Figure 4.30.: Configuration unit with by-pass selector logic for enabling slice-level partial reconfiguration in *V-FPGA*

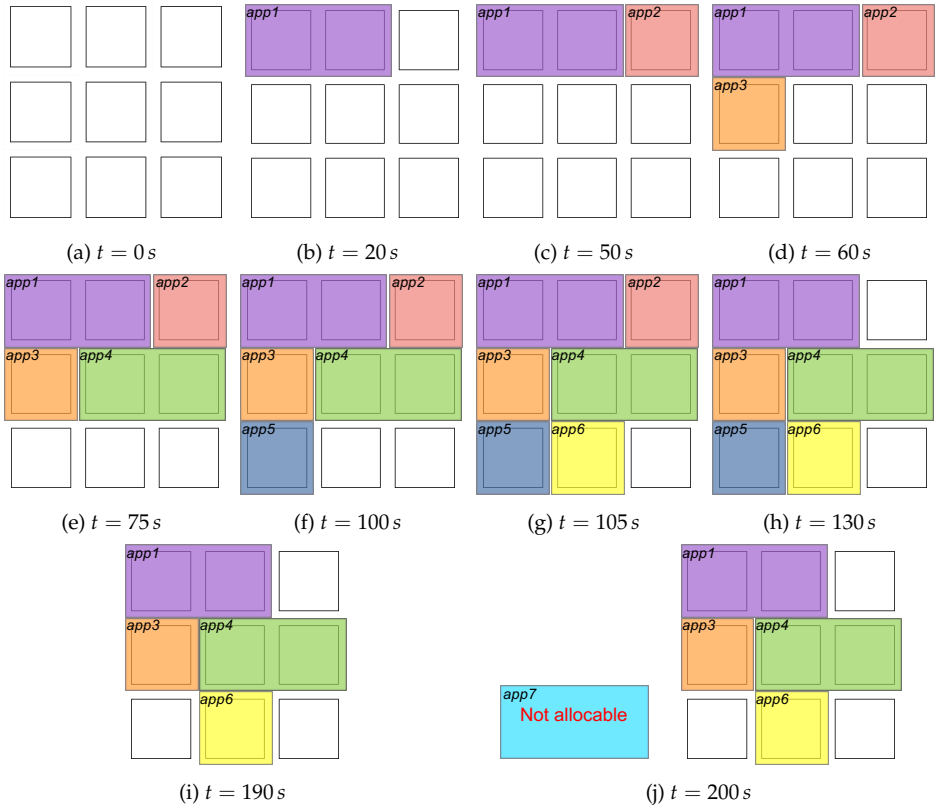
4.3.6. Runtime Task Migration and Defragmentation

In partial and dynamic reconfigurable architectures the perpetual allocation and de-allocation of applications or tasks during runtime can lead to fragmentation of the reconfigurable fabric similarly to the fragmentation of hard disk drives in PCs. Fragmentation of FPGAs has been long neglected, but think of FPGAs in the cloud where different applications or service requests fly in and out on demand in an unpredictable way. If several such transient applications are concurrently mapped on the same device, fragmentation effects can occur quite quickly. The consequence is that inspite unoccupied ressources are available, they are scattered and thus can not be utilized by applications that require large contiguous area. A simple example is illustrated in Figure 4.31.

Assuming a system with 9 *V-FPGA* cores and *CoreFusion* capabilities, during runtime the core array is allocated and deallocated by various applications of different size. At $t = 200\,s$ the system reaches a state in which despite of sufficient unoccupied resources *app7* can not be mapped on the array as it needs a contiguous area of two adjacent *V-FPGA* cores.

To overcome such situations, the *V-FPGA* architecture is extended by mechanisms that allow a defragmentation of the core array during runtime [35]. In the example of Figure 4.31 *app6* would be migrated to the top row to free sufficient contiguous area in the bottom row to accommodate *app7*.
The challenge thereby is to perform the migration during runtime in a non-destructive and practically non-disruptive way for the running applications, including the one that is to be moved. Non-destructive means that the running application should maintain its state during migration. Non-disruptive means that realtime behaviour/response of running applications should not be disturbed by migration, which is especially important e.g. in streaming applications, where data samples should not be lost.

In fact, it is not possible to move a running application without any interruptions, but the interruptions can be kept negligibly small so that they don't do any harm even in realtime applications. In *V-FPGA* this is achieved by a four-step migration process with fast hardware mechanisms for reading and restoring instant snapshots of the application's state.



(a) $t = 0\,s$ (b) $t = 20\,s$ (c) $t = 50\,s$ (d) $t = 60\,s$

(e) $t = 75\,s$ (f) $t = 100\,s$ (g) $t = 105\,s$ (h) $t = 130\,s$

(i) $t = 190\,s$ (j) $t = 200\,s$

Figure 4.31.: Fragmentation scenario in runtime partial reconfigurable architectures

4.3.6.1. Snapshot Functionality in *V-FPGA* Architecture

Taking a snapshot means that the output signals of all signal sources in the *V-FPGA* need to be stored instantly at a certain point in time.

The signal sources of a core are all CLBs and the peripheral unit. Because the CLBs already contain a flip-flop at the LUT's the modifications of the *V-FPGA* architecture to support snapshots are minor and the hardware overhead negligible (see Figure 4.32 where modifications are marked in red colour).

With the additional signal *nSnap* that is connected to the *EN* input of the already existing flip-flop the output can be freezed when $nSnap =' 0'$. As there is one global *nSnap* signal for all CLBs inside a core this simple modification realizes the snapshot functionality at nearly no additional hardware costs.

In order to support the transport of the snapshot data a daisy-chain approach similar to the configuration mechanism is used. If snapshot data needs to be stored or loaded then the output flip-flops of all CLBs within a column are daisy-chained by a multiplexer at their inputs and build a shift register that transports the data synchronously as a bitstream. Again, as shown in Figure 4.33 each column has its own daisy-chain and all daisy-chains transport the data at the same time. Similar to the configuration data the daisy-chains of the snapshot data are connected to the data busses of a RAM block where each daisy-chain an according bit position in the words of the memory is assigned. The beginnings of the daisy-chains are linked to the write data bus and their ends to the read data bus. Thus the storing and the loading of snapshot data work the same way. The address bus of the RAM is driven by the configuration controller and incremented every clock cycle during the transport.

The snapshot functionality of the peripheral unit is also implemented at nearly no additional hardware costs. As the outputs of the peripheral unit are anyway registered, the corresponding flip-flops can be freezed by the global *nSnap* signal that is connected to the *EN* inputs of the flip-flops. The integrated communication controller supports the storing and loading of snapshot data via the AMBA APB bus.

Apart from migration and defragmentation, the snapshot functionality further enables some new possibilities. E.g. a running application can be interrupted and the core it is running on can be freed if a higher priority application is requested but all cores are occupied. When the higher priority application terminates then the interrupted application can be mapped again on the core and resume its work. This method further enhances the overall utilization as idle cores can be used for low priority background check applications but still be available for mapping higher priority applications as they are interruptible.

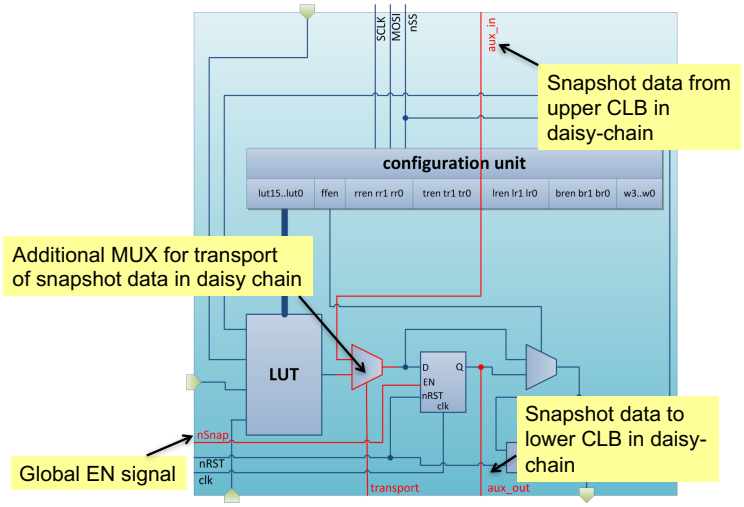


Figure 4.32.: Snapshot mechanisms in *V-FPGA* [35]

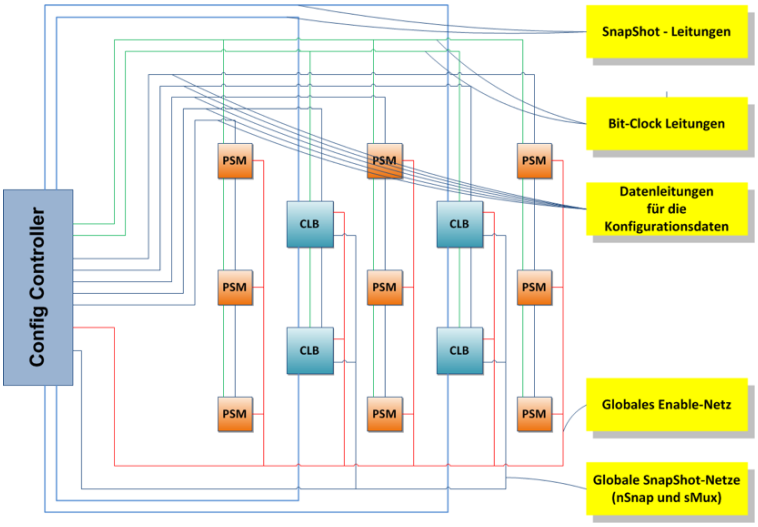


Figure 4.33.: Snapshot net in *V-FPGA* [21]

4.3.6.2. Task Migration Process for *V-FPGA*

To migrate an application or task from on *V-FPGA* core to another four steps are performed in the following order:

1. The configuration controller configures the destination core with the same application bitstream as the source core.
2. Once the configuration process of the destination core is finished, a snapshot of the source core's state is taken and stored.
3. In a next step the stored snapshot has is loaded into the destination core.
4. Finally, in software the address of the application needs to be remapped to the new core. Note that it is assumed that all *V-FPGA* cores are equipped with a peripheral unit with communication controller as described in Section 6.1 and thus is addressable for data exchange with a microprocessor through busses. this explains why a remapping of the base-address is necessary when an application moves to another core.

From the moment the snapshot is taken until the moment the address is remapped, all data exchange between software and mapped application is invalid and should be avoided in software. However, this doesn't cause any serious performance problems as the snapshot data is pretty small and can be restored quite fast inside the destination core.

The simulation results in Figure 4.34 show the behaviour and timing when migrating an application from one *V-FPGA* core to another. As it can be seen, the duration of taking and storing snapshots as well as restoring them on another core is very short in the range of a few microseconds. The example application is a CRC calculation over multiple data samples and is migrated during it's runtime to another core. The final result of the calculation is correct.

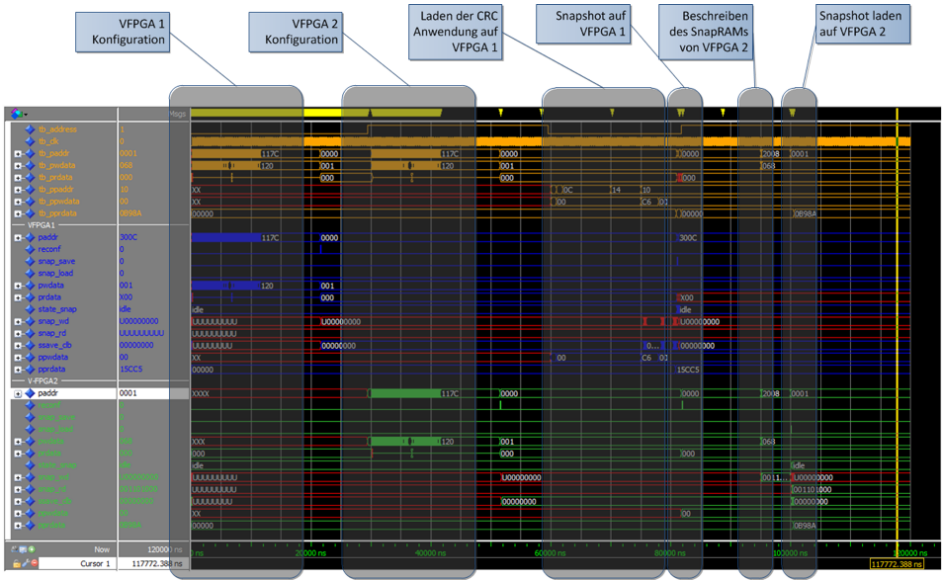


Figure 4.34.: Simulation of a migration of a running application from one *V-FPGA* core to another [21]

4.3.6.3. Dynamic Defragmentation Strategy

The monitoring of the fragmentation degree, the decision whether to defragment or not as well as the strategy of moving and swapping applications through the core array are done in software that is executed on the microprocessor. This was contributed by Thomas Bruckschlögl in his student research project [21].

Since a defragmentation should be performed in-system dynamically during runtime, a fast heuristic is developed that performs this task in acceptable time. The calculation of a new allocation map must be completed before a core application terminates its execution. Otherwise, if a defragmentation process is not completed by this time, space is reserved for an application that is no longer required. In addition, it could be that the defragmentation would not be necessary any more, since by completion of an application a sufficiently large contiguous area in the array could become free.

These time constraints can be exacerbated by adding the reconfiguration duration to the calculation duration of a new allocation map or even performing a reconciliation between the reconfiguration time of a core and the remaining completion time of an application. A threshold could then ensure that certain cores are marked as non-movable, if the time expenditure is no longer worthwhile.

With respect to the defragmentation strategy, a greedy-heuristic is applied, which starting with the largest configuration attempts to fill the array of *V-FPGA* cores from the upper left corner to the lower right corner. The width and height of a configuration is defined by the number of fused *V-FPGA* cores in x and y direction, respectively.

The steps are as follows:

1. Sort the list of configurations by size
2. Select the largest, unplaced and non-discarded configuration
3. Find a suitable region
   a) If height > width of configuration, find suitable region first as far to the left as possible and then as far to the top as possible.
   b) If height <= width of configuration, find suitable region first as far to the top as possible and then as far to the left as possible.
4. If sufficient area is found, then
   a) mark the configuration as placed
   b) mark the required area as occupied
5. If no adequate area is found, then
   a) mark the configuration as discarded
   b) repeat steps 2 to 5 until all configurations have been placed or discarded

Figure 4.35 illustrates the sequence of the algorithm graphically. First, startup configurations with initial assignment of a fragmented *V-FPGA* array are assumed. The initial assignment is shown on the left, while the result is the right-hand side after defragmentation. This example is intended to illustrate how the algorithm works, showing that the

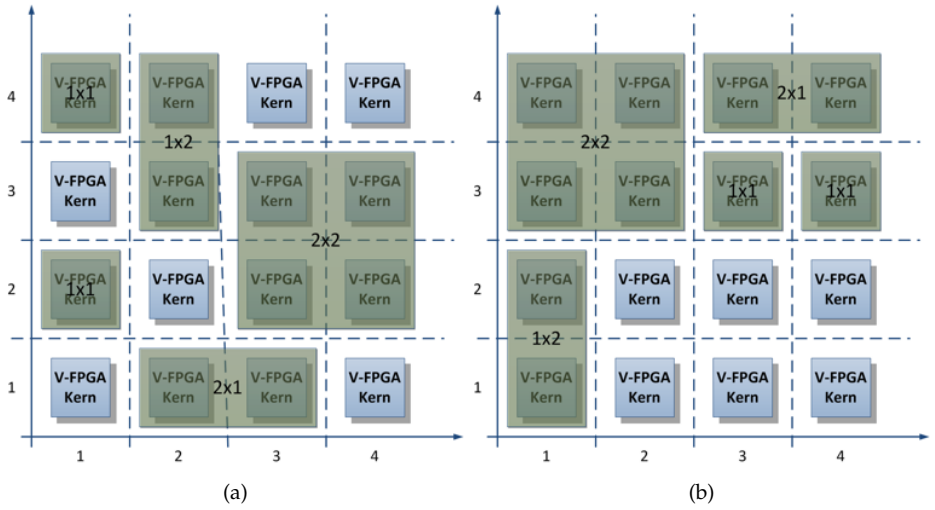


Figure 4.35.: Example of a fragmented *V-FPGA* core array (a) and the result after defragmentation (b) [21]

largest configuration is located at the top left corner, and that the next two larger configurations are placed with priority left (configuration 1x2) or priority top (configuration 2x1).

4.4. Multi-granular Virtual Reconfigurable Architecture

A special case of *V-FPGA* is the ViSA and V-FPGA Integrated System (ViSA-VIS). The specialty is that some or all of the logic cells are composed of a scalable microarchitecture that is able to run programs. The objective is to reduce the control overhead of applications that are not fully data-flow driven.

4.4.1. ViSA: VLIW Inspired Slot Architecture

VLIW inspired Slot Architecture (ViSA) is a generic and scalable microarchitecture that can serve either as a loosely coupled programmable accelerator core in a system (as demonstrated in [36]) or be integrated as special macro cells in the *V-FPGA*, thereby taking advantage of the surrounding programmable logic and interconnects. The latter approach can be seen as an extension of what is being today offered in FPGAs as dedicated multipliers or DSP slices. The most important differences to those units are:

- *ViSA* can execute programs. It supports also a configurable data path mode to be used in a similar way as DSP slices for control-less streaming applications. Thus it's more versatile.

- *ViSA* is generic and scalable. Thus it can be customized to cover a custom range of operations, from simple multiply, add/subtract to complex and custom operations.
- In contrast to DSP slices, *ViSA* can implement control flow.
- *ViSA* is an efficient alternative to FSMs as it moves the entire control flow complexity to memory rather than expensive logic.

4.4.1.1. The Idea Behind the *ViSA* Approach

The original idea [36] behind the VLIW inspired Slot Architecture was driven by two key demands:

1. In contrast to typical FPGA hardware designs based on FSMs, we were looking for an alternative approach to drastically reduce chip area without degrading performance.
2. We aimed to develop a generic, reusable development approach that can be easily tailored towards application-specific demands.

Thus, *ViSA* provides highly efficient yet flexible custom computing solutions at fast time-to-market.

A widely used design method for FPGAs is to implement parallel arithmetic computing units, e.g., multipliers, adders, and control them by FSMs, or even to have all computations inside large FSMs. However, this is not always a very efficient approach because precious resources are often wasted for purposes that are dedicated more to controlling and not directly to computation. This is even more severe when resource sharing is needed due to limited amount of computing units, e.g., multipliers.

Therefore, the proposed overlay architecture is focused on reducing the control flow overhead to a minimum by using similar techniques as found in small processors. However, a big difference to conventional processor architectures is that the same amount of parallel computing units as in pure hardware designs is used. The parallelism is distributed in a hierarchical manner, e.g., locally in parallel computation slots like in VLIW and globally by multiple instances of *ViSA* cores, which could be realized as homogeneous or heterogeneous multicores.

4.4.1.2. Generic Structure

As depicted in Figure 4.36, the generic VLIW-inspired Slot Architecture mainly consists of functional units, registers, multiplexers, a program counter (PC), data and instruction memories and a VLIW-like instruction word which resides inside the instruction memory. In contrast to conventional processors, there are neither pipeline stages nor a decoding unit, dispatcher, reservation station, etc.. *ViSA* is a strictly reduced architecture with focus on efficiency optimization like area, power and performance. Typical function units in the *ViSA* architecture are adders, multipliers, logic units, load/store units, comparators, but could also be more complex units like square root if beneficial for the application.

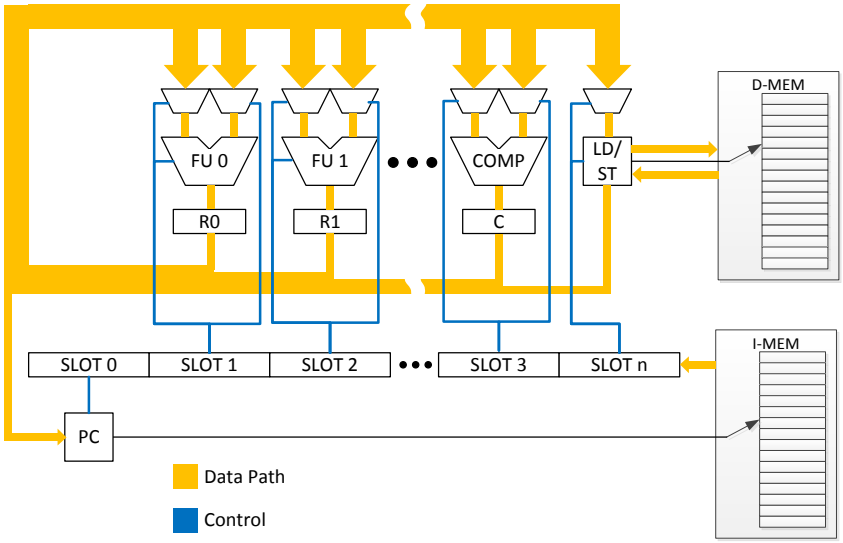


Figure 4.36.: Generic VLIW-inspired Slot Architecture consisting of multiple functional units, registers, multiplexers, a program counter, data and instruction memory

The outputs of the function units are fed into registers and then back to the input multiplexers, which control the data paths. The *ViSA* instruction words directly control the input multiplexers, the modes of the function units as well as the program counter, e.g., the single signals of the instruction memory's data bus are directly connected to the respective units. Every unit with its multiplexers has a dedicated control slot in the *ViSA* instruction word. This requires almost no control logic making the approach very efficient. In every clock cycle, the PC determines the address of the next *ViSA* instruction word to be loaded, such that the program is executed. Besides an incremental mode, the PC supports also conditional and unconditional jumps. The mode is controlled by the current *ViSA* instruction word. In case of conditional jumps, the result of the comparator unit is used to decide whether to jump directly to a PC-relative address or to increment the current address.

The simplicity of the VLIW-inspired Slot Architecture not only reduces chip area but can additionally increase the maximum obtainable operating frequency, since the combinational paths are significantly shorter compared to complex processor FSMs. The VLIW-inspired Slot Architecture is generic and scalable in the means that depending on the multi-objective application needs and the desired degree of parallelism, the number and the type of the function units can be adapted in order to get a highly efficient specifically tailored solution. Furthermore, the data widths, memory depths and number of parallel *ViSA* cores inside a system are parametrizable, such that application-specific many core systems can be efficiently implemented.

4.4.1.3. Programming

The types and numbers of function units within a custom *ViSA* core can be customized which defines an application specific instruction set and the number of parallel slots. Programs can be written in parallel assembly code, each assembly instruction is separated into parallel sections, one for each functional unit. An extract of an exemplary assembler code is shown in Listing 4.1. Parallel sections are represented by a vertical bar. For sake of simplicity the example architecture features only 4 parallel slots: program counter w. address generation, multiply-add/subtract unit, logic unit, load/store unit.

```
1  _sqrt_branch1:            ; positive case
2        INC | CLR ACC | SRA R0 8 R0 | STOR sqrt_k R0
3        INC | CLR ACC | SRA R0 8 R0 | LOAD mask_0x1 LD
4        INC | ADD R0 LD ACC | MOV R0 R0 | LOFF 0 R0 LD
5        INC | MOV ACC ACC | NOT LD R0 | LOFF 0 R0 LD
6        INC | MOV R0 ACC | MOV LD R0 | LOAD mask_0x1 LD
7        INC | ADD ACC LD ACC | MOV R0 R0 | LOAD sqrt_k LD
8        INC | ADD ACC R0 ACC | MOV LD R0 | LOAD mask_0x00FFFF LD
9        INC | MOV ACC ACC | AND R0 LD R0 | LOAD sqrt_k LD
10       INC | MUL ACC R0 ACC | SRA LD 8 R0 | LOAD sqrt_k LD
11       INC | MOV ACC ACC | SRA LD 8 R0 | LOAD sqrt_k LD
12       INC | MOV ACC ACC | SRA ACC 8 R0 | LOFF 0 R0 LD
13       INC | MOV LD ACC | SRA ACC 8 R0 | LOAD scale_Qy LD
14       INC | ADD ACC R0 ACC | MOV R0 R0 | LOAD @_sqrt_branch4 LD
15       JPOS R1 LD | MOV ACC ACC | NOT ACC R0 | LOAD mask_0x1 LD
```

Listing 4.1: Extract of a parallel assembly code for a custom *ViSA*

In the end, an assembler and linker tool translates the application-specific data flow into the needed *ViSA*-specific machine code. The code is divided into sections for instructions and data. One part will be the hex-code for the instruction memory and the other part is the hex-code for the data memory.

4.4.1.4. Integrated Tracing

After all parts of the VLIW-inspired Slot Architecture are implemented, the verification phase is conducted. Therefore, the whole design is simulated. For better usability of the workflow and a faster time-to-market, a comprehensive debug unit is implemented into the *ViSA* source code. Trace 4.37 shows an example for the integrated tracing. During simulation, the debug unit prints the integrated tracing. Now, each state of *ViSA* is printed out enabling a detailed step-by-step reproduction of the execution, e.g., the states of the registers, the address of the instruction memory and the disassembly is given. This is much more convenient and minimizes the effort of analyzing simulation waveform traces.

```
# ------------------- Step: 70-------------------
# ACC: 1000111110110010000100001 [-0.5..0.5]:
# R0: 1000111110110010000110011
# LD: 00000000000000000000001101
# R1: 00000000000000000000000100
# iMEM_ADDR: 00001101
# Disassembly: INC | MOV ACC ACC | MOV R0 R0 | LOAD @_end LDD
# VLIW: 00000001111001010000000101011
# ->ADDR: 0000101011
# ->LDST_MODE: 00
# ->LDST_sel: 00
# ->logic_B_sel: 01
# ->logic_A_sel: 01
# ->logic_mode: 110
# ->alu_B_sel: 11
# ->alu_A_sel: 00
# ->alu_mode: 00
# ->pc_mode: 000
```

Figure 4.37.: Integrated tracing output during simulation with register states, addresses of instruction and data memory, disassembly and *ViSA* instruction

4.4.2. ViSA-VIS: ViSA and *V-FPGA* Integrated System

The integration of *ViSA* cores into the *V-FPGA* fabric as illustrated in Figure 4.38 yields a ViSA and V-FPGA Integrated System (ViSA-VIS) with extended capabilities for customization. Thereby, CLBs and ViSA cores coexist face-to-face in the *V-FPGA* array and share the same routing topology. Similarly, memory blocks are integrated to provide instruction memory to the *ViSA* cores if needed and shared data memory, though instruction memory is not needed in cases where *ViSA* cores are used as data-path processing elements. As shown in Figure 4.38 *ViSA* cores have the width of a CLB yet the height of a multitude of CLBs depending on the complexity and size relative to each other. Same applies to the memory blocks. The limitation to align the width of a *ViSA* core to a CLB column is imposed by the place&route tools that allow to model only the height of such heterogeneous blocks, yet not the width. However, there are no objections to extend *ViSA-VIS* to integrate multi-column wide *ViSA* cores if the place&route tools will support it one day.

The purpose of heterogeneous integration in the *ViSA-VIS* is manyfold:

- Arithmetic functions benefit from a coarser cell architecture with word granularity rather than bit granularity. In essence, this is because the configuration overhead or programming overhead per bit is rather small compared to CLBs as the function set of basic arithmetic operations (multiplier, adder, subtractor, etc.) is considerably smaller than the function set of LUTs.
- The inclusion of dedicated highly optimized function units for basic arithmetic operations is an effective way to mitigate the ASIC gap. This has been recognized by FPGA manufacturers since the nineties and today most commercial FPGAs include multipliers and adders as dedicated full-custom hard macros, that can be clocked much higher than CLBs [102].

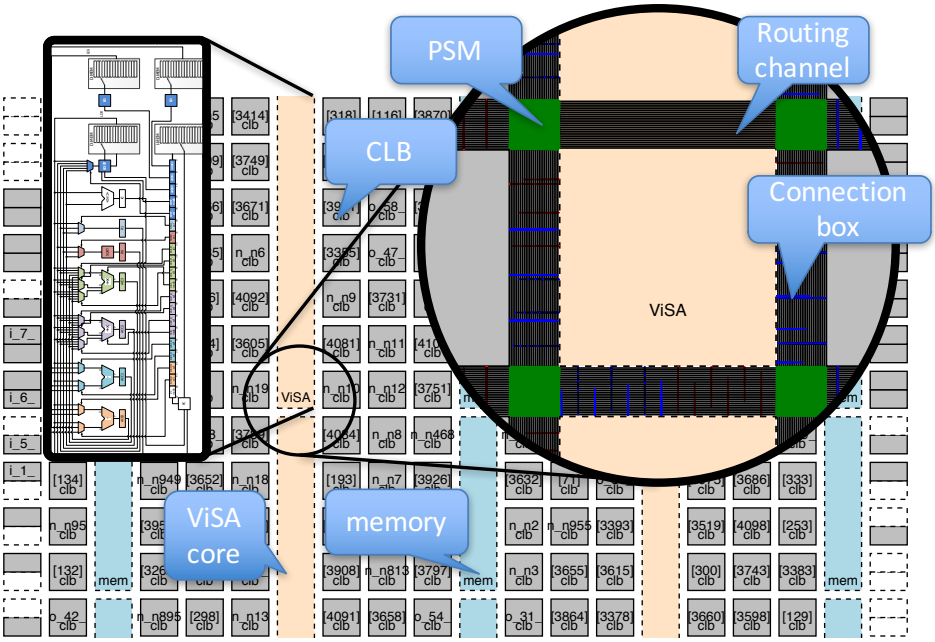


Figure 4.38.: Structure of ViSA and V-FPGA Integrated System (ViSA-VIS) with CLBs and *ViSA* cores aranged face-to-face in the *V-FPGA*

- *ViSA-VIS* goes one step further as the *ViSA* cores can execute small programs. This allows to efficiently execute complex arithmetic operations by breaking them down into conditional sequences of basic operations.
- The *ViSA* concept with its mechanisms to custom integrate dedicated special function units in a well defined and scalable way proposes the customization of such cores to efficiently target certain application classes. For instance, multidimensional ultra-sound based imaging often relies on calculating the euclidian distance between two points in cartesian coordinates. The most critical arithmetic operation therein is the square root. Thus, a custom *V-FPGA* targeted at various applications of ultrasound based imaging would benefit from the inclusion of a square root engine in the *ViSA* cores as discussed and demonstrated in [36].

Table 4.12 compares *ViSA-VIS* with COTS FPGAs in terms of heterogeneous resources. The main differences are that *ViSA-VIS* has custom function units in its *ViSA* cores and that they can execute program code if needed.

Table 4.12.: Heterogeneous processing blocks in FPGAs - a qualitative comparison of *ViSA-VIS* with COTS devices

		Virtex-7 [109]	Stratix V [8]	ProASIC-3 [2]	Igloo 2 [74]	*ViSA-VIS*
Logic blocks		CLB with 4 fractionable LUT-6	ALM with adaptive LUT	VersaTile (LUT-3 equivalent)	LUT-4	CLBs with custom LUT and cluster sizes
Arithmetic blocks	type	DSP48E1	DSP	-	Math Blocks	*ViSA*
	multiply	yes	yes	-	yes	yes
	multiply-add	yes	yes	-	yes	yes
	multiply-accumulate	yes	yes	-	yes	yes
	logic	yes	yes	-	no	yes
	custom	no	no	-	no	yes
	microcode execution	no	no	-	no	yes

4.5. Conclusion

In contrast to the state-of-the-art the proposed generic *V-FPGA* architecture is highly flexible and can be customized by more than 20 generic parameters. With this flexibility the *V-FPGA* can be tuned towards the application's needs and thus the efficiency and/or performance can be maximized. The most important parameters are LUT size K and cluster size N. In fact, the extensive study with parameter sweeps through over 1400 benchmark runs showed a very high sensitivity for these parameters, resulting in area variances of up to $\pm 95.9\%$ and performance variance of up to $\pm 78.1\%$ between the best and worst solutions per benchmark circuit. Interestingly, individual benchmarks showed individual preferences with regard to parameter choice, that differed from the average case. This demonstrates the significance of parameter tuning and customization.

Apart from its flexibility, the proposed *V-FPGA* incorporates a number of unique architectural features.
LoopbackPropagation is a circuit technique that enables bi-directional routing tracks even though the underlying physical platform offers only uni-directional resources. Compared to tri-state emulation of related works, *LoopbackPropagation* saves W AND-gates per CLB-input and substitutes $2 \cdot W$ AND-gates + W OR-gates by W MUX2 per CLB-output, whereby W is the number of tracks per routing channel. This makes it the most area efficient technique for virtualizing bi-directional routing tracks.
The inclusion of 3D PSMs with V-TSVs makes *V-FPGA* the first 3D virtual FPGA known so far. Of course the number of stacked layers as well as the number of V-TSVs per PSM are parameterizable in order to explore various 3D topologies.
The *V-FPGA* supports partial and dynamic reconfiguration even on physical underlying platforms that don't offer this feature natively. This is achieved by configuration mechanisms that are independent form the configuration infrastructure of the underlying platform. *CoreFusion* is a unique feature of the *V-FPGA* that allows two merge two adjacent *V-FPGA* cores. This allows coarse-grained partial reconfiguration. A by-pass selector logic in the configuration units enables partial reconfiguration at a very fine grain of individual LUTs, PSMs and IOBs, while others are continuing operation without interruption. In contrast to the state-of-the-art, that allows only column-wise partial reconfiguration, fine-grained partial reconfiguration inherently allows arbitrary aspect ratios, better routability, faster context switching and higher utilization.
The fast *Snapshot* functionality of the *V-FPGA* enables to dynamically migrate an applica-

tion from one *V-FPGA* core to another during its runtime. This can be used for runtime defragmentation of an array of *V-FPGA* cores in order to free contiguous reconfigurable area that otherwise is scattered due to fragmentation effects that arise from the perpetual on-demand allocation and de-allocation of applications during runtime. This is especially relevant for FPGAs in the cloud, where fragmentation effects can happen quite quickly as result of unpredictable on-demand service requests. A heuristic for quick defragmentation exists for the *V-FPGA*. Another use of the *Snapshot* functionality is to interrupt low-priority tasks by higher-priority tasks that need the occupied area and to resume the low-priority tasks once the high-priority tasks are finished.

In *ViSA-VIS*, a heterogeneous extension of the *V-FPGA*, the classical CLBs are mixed with *ViSA* cores. *ViSA* is a generic customizable microarchitecture, exploiting superscalarity. This heterogeneous mix goes one step further than related works that include multipliers and adders in their FPGAs, with the differences that *ViSA* can contain custom function units and can execute programs. *ViSA* was designed with the aim of minimum area requirement as an alternative to FSMs, thus complex multi-cycle arithmetic functions can be realized very efficiently with *ViSA*. In this tandem, the CLBs can be used then for glue-logic or other purposes.

5. Application Mapping and Toolflow

The procedure of implementing an application onto an FPGA - from design entry to the final programming binaries realizing the desired circuits - we call *application mapping*, whereby the involved CAD tools fall in the category of Electronic Design Automation (EDA). The mapping of applications onto the *V-FPGA* requires similar steps as the procedures for COTS FPGAs.

Starting with a *design entry* usually in a hardware description language like VHDL or Verilog, a technology independent gate-level netlist is generated through *synthesis*. This step usually involves also multiple phases of logic optimizations to reduce the area and/or to increase the performance. Next follows the *technology mapping*, whereby the gate-level netlist is transformed into another netlist with its nodes representing K-input LUTs and flip-flops of the target architecture. In case the target architecture supports clustering, then a so-called *packing* step needs to take place, resulting in a new hypergraph netlist whereby each node consists of a subgraph with up to N BLEs (i.e. LUT + flip-flop pairs) and their interconnects. Next is the *placing* step in which each node of the technology-mapped and evtl. packed netlist is assigned a location within the CLB array or the IOB perimeter. The subsequent *routing* step determines the paths through the routing channels for interconnecting the placed nodes that are spread throughout the FPGA. Finally the *bitstream generation* follows, which determines the programming bits for the logic-, routing- and I/O-resources within the FPGA, in order to incorporate the results of synthesis, place & route, thus realizing the circuit. Due to the complexity of applications and devices, tool-support for all these steps is indispensable.

EDA for *V-FPGA* to perform the above named steps faces a number of challenges:

1. Developing a complete set of EDA tools tailored for the *V-FPGA* would take years of effort.
   Proposed solution: Exploit existing tools as much as possible.
2. The *V-FPGA* has a custom architecture. Consequently none of the commercial tool flows for COTS FPGAs can be exploited for the technology-dependent steps (from technology-mapping to bitstream generation).
   Proposed solution: Perform technology-dependent steps with adoptable academic tools.
3. Academic tools have limited support of design entry methods.
   Proposed solution: Use commercial tools whenever advanced design entry is needed or favoured.
4. Theoretically, the technology independent steps (design entry, synthesis, logic optimization) can be performed by any commercial tool flow for COTS FPGAs (e.g. Xilinx ISE/Vivado, Microsemi Libero, Altera/Intel Quartus, ...). Practically, however, those tools produce only the technology-mapped netlist (which is incompatible to

V-FPGA) and intermediate steps/netlists are inaccessibly encapsulated. Exception: Altera/Intel offers the Quartus University Interface Program to access the intermediate netlists for research purposes.
Proposed solution: Perform synthesis with either vendor independent tools, academic tools or Altera Quartus + QUIP.

5. The *V-FPGA* is highly customizable. EDA tools need to be parameterizable to take advantage of this flexibility.
Proposed solution: Use academic tools (e.g. MEANDER [96], VTR [63], ...), that can be exploited for the steps from design entry, over synthesis till place & route. If design entry methods of academic tools are unsatisfactory, then complement those tools with Altera Quartus + QUIP.
6. Existing academic tools work on a very abstract level and there is no way to bring in additional information regarding the realization of the programmable resources and their configuration techniques. Consequently they can not generate bitstreams for the *V-FPGA*.
Proposed solution: New tool as back-end to perform bitstream generation.
7. *V-FPGA* is an emerging technology and needs additional tool support for verification, manipulation, graphical editing, etc.
Proposed solution: New tool with graphical editing and automated testbench generation.
8. Parameters of academic CAD tools need to be tuned whenever *V-FPGA* gets customized, which is time consuming to do by hand.
Proposed solution: New tool that generates automatically architecture files for academic CAD tools.

Based on these considerations a flexible tool-flow for EDA of application mapping onto *V-FPGA* is proposed in Figure 5.1. The tool-flow is composed of academic tools, commercial tools and a new tool (*V-FPGA Explorer*) developed for the *V-FPGA*.
From the VTR package [63], the tool *ODIN II* can be used for synthesis. Logic optimization and technology-mapping can be done with the tool *ABC*, while *VPR 7* can perform the packing, place & route steps.
From the MEANDER framework, the *VHDLParser* can be used for syntax check. The tool *DIVINER* can be employed for synthesis, followed by compatibility modifications and format conversions with the tools *DRUID* and *E2FMT* respectively. The tool *SIS* can be used for logic optimization and technology-mapping. The tool *POWER MODEL* can facilitate power estimation in conjunction with *EX-VPR* by generating switching activity profiles. *T-VPACK* tool can be used for the packing step and *EX-VPR* for place & route and power estimation.
Altera's QUIP (Quartus University Interface Program) enables to use *Quartus II* as a front-end for design entry and for synthesis and logic optimization.
The new tool *V-FPGA Explorer* can be used as a back-end for bitstream generation with import. Furthermore it can generate the architecture files needed in VTR and MEANDER, testbenches for simulation, scripts for running the VTR tools and extracting the results. It can also act as a graphical editor for function mapping, place & route.
Modelsim can be used for simulation of both, the *V-FPGA* and the application running onto it.

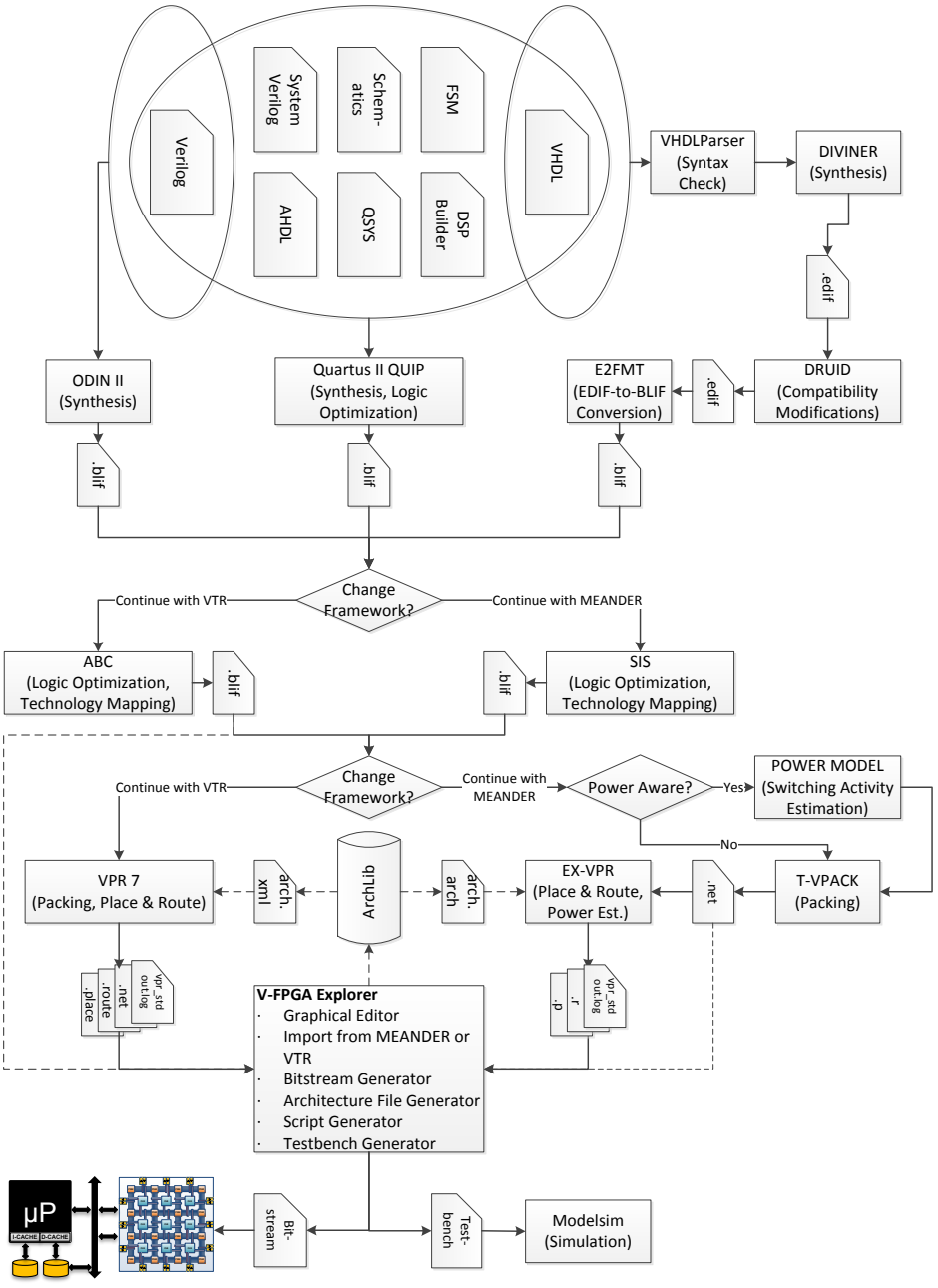


Figure 5.1.: Tool-flow for application mapping onto *V-FPGA*

The tool-flow has four different flavours:

- Flow 1: Start with Verilog as design entry; perform synthesis, logic optimization, technology mapping, packing, place and route with tools from the VTR package; use *V-FPGA Explorer* as the back-end for bitstream generation.
- Flow 2: Start with VHDL as design entry; perform synthesis, logic optimization, technology mapping, packing, place and route with tools from the MEANDER flow; use *V-FPGA Explorer* as the back-end for bitstream generation.
- Flow 3: Use Quartus as the front-end for advanced design entry (supporting VHDL, Verilog, System Verilog, AHDL, Schematics, QSYS, FSM, DSP Builder) and synthesis and logic optimization; use VTR as the middle-ware for technology mapping, packing, place and route; use *V-FPGA Explorer* as the back-end for bitstream generation.
- Flow 4: Use Quartus as the front-end for advanced design entry (supporting VHDL, Verilog, System Verilog, AHDL, Schematics, QSYS, FSM, DSP Builder) and synthesis and logic optimization; use MEANDER as the middle-ware for technology mapping, packing, place and route; use *V-FPGA Explorer* as the back-end for bitstream generation.

Since intermediate results are generated after each step, the flows can be also mixed changing the framework at any time after synthesis (as shown by the conditional branches in Figure 5.1), e.g. performing synthesis with VTR, then technology mapping with MEANDER, then packing, place & route with VTR, etc..
In the following subsections, the tools and the procedures for application mapping onto *V-FPGA* are described in more detail.

5.1. Design Entry and Synthesis with Altera Quartus II and QUIP

In principle, academic toolflows such as MEANDER framework or VTR package have built in synthesis engines that can synthesize a VHDL or Verilog file into a technology independent netlist, that is needed for the further steps. These tools however are not yet as mature and comfortable to use as commercial IDEs of FPGA vendors, that offer a higher functionality, various design entry methods and user interfaces, mixed hardware description language support, support of advanced HDL constructs (GENERATE loops, generics, etc.), support of graphical editors, libraries, code generators, syntax highlighting, error checks/location, elaboration utilities, etc.. On the other hand, almost all commercial IDEs can be used only with the respective COTS FPGAs, because the outputs are vendor and part specific. An exception is Altera Quartus II, the IDE for Altera FPGA devices. With the Quartus University Interface Program (QUIP) [7] it offers the possibility to interface with academic CAD tools for research purposes by generating intermediate and technology independent netlists in Berkeley Logic Interchange Format (BLIF) format after synthesis, optimization or LUT-mapping steps. This behaviour is disabled by default but can be activated by modifying the *.qsf project file in a text editor and adding commands for dumping the desired netlists. The supported commands are explained in [7].

Thanks to this interface, the *V-FPGA* framework uses Quartus II as a front-end for design entry and technology independent logic synthesis, i.e. the goal is to transform a design into a technology independent BLIF netlist which will be used in other tools for further steps of the application mapping methodology. Fortunately the QUIP approach supports an extended set of design entry methods including:

- coding in VHDL, Verilog, SystemVerilog or AHDL
- drawing schematics
- instantiating LPM modules
- instantiating IP cores
- graphically modeling state machines
- code generators
- system integration with QSYS
- high level synthesis from Matlab using Altera DSP Builder

The procedure to design a circuit in Quartus II and to obtain the desired *.blif netlist is the following:

1. Create a new design project, whereby the name of the project should correspond to the top-entity name of the design.
2. Select any of the available devices as target, e.g. Stratix EP1S10F484C5. It doesn't matter which device is selected because we are interested only in the technology independent netlist before the target mapping.
3. Design the circuit using any of the above named design methods (or combinations of multiple methods). Thereby ensure that the top module name corresponds to the project name.
4. Open the *.qsf project file in a text editor and add the command lines in Listing 5.1:

```
1  set_global_assignment -name DSP_BLOCK_BALANCING "LOGIC ELEMENTS"
2  set_global_assignment -name AUTO_SHIFT_REGISTER_RECOGNITION OFF
3  set_global_assignment -name INI_VARS "no_add_ops=on;
4    dump_blif_with_blackboxes=off;
5    dump_blif_after_optimize=on;
6    opt_dont_use_mac=on"
```

Listing 5.1: QUIP settings for obtaining the technology-independent netlist

Line 5 is the command to output the blif netlist after technology independent synthesis and optimization, but before technology mapping. This infers usually the steps:

- VHDL/Verilog/SystemVerilog/AHDL analysis (parsing)
- Elaboration (binding and instantiation)
- RTL inferencing and synthesis

- LPM instantiation (common functions such as barrel shifters are converted into optimized library functions)
- Technology-independent multi-level optimization (algebraic and functional techniques to optimize the netlist)

The other lines 1, 2, 3, 4 and 6 effectuate that device specific properties and primitives such as DSP-blocks, shift registers, adder carry-chains, blackboxes and MAC units are avoided in the synthesis steps. Instead logic elements are used.

5. Finally start the *compile* process in Quartus II. This process will usually fail at some point in time and report errors, because it can not perform the technology mapping due to missing or incompatible settings. Nevertheless, by this time the desired technology independent netlist would have been already generated and saved as *.blif file in the project directory and is ready to be used by the other tools.

At this point it is important to note that while Altera (now part of Intel) endorses the use of this flow for research purposes, according to [7] *"it is a violation of your license agreement to use Quartus to synthesize designs for commercial purposes (e.g. as the front-end to a competitive product to Altera's FPGAs or as the front-end to a commercial EDA tool under development)"*! In case of commercialization of the *V-FPGA*, for legal reasons surely another front-end needs to be employed if the underlying platform is not an Altera/Intel device. A special case is the commercial use of the *V-FPGA* as virtualization layer on an Altera/Intel device. It that case it needs further clarification (and eventually additional agreements) with Intel whether it is permitted to utilize Quartus as front-end tool for the virtual layer on an Altera/Intel device.

5.2. Technology Mapping, Place & Route with MEANDER Framework

The MEANDER framework [96] consists of several well-known academic tools that have been adapted as well as newly developed tools. The tools are arranged in a toolflow-based Graphical User Interface (GUI) (see Figure 5.2), but can also be used independently. Each tool makes the results available in readable text files after it has been executed. A unique specialty of MEANDER is that it can be remotely operated online in a web-browser [69] and the tools will run on a web-server, thus it can be used anytime and anywhere without the need for local installation, supposed there is an internet connection. Apart from the GUI the tools can be operated also in command line manner through a Secure Shell (SSH) terminal.

The *UPLOAD* tool allows to upload files that are used as input to the tools. At least the VHDL files of an application are required. However, it is also possible to upload already synthesized netlists and to skip the MEANDER synthesis if the use of a more powerful synthesizer is favoured.

The *VHDL parser* analyzes the VHDL files and performs a syntax check. In case of syntax errors it is also able to locate the errors and to suggest corrections.

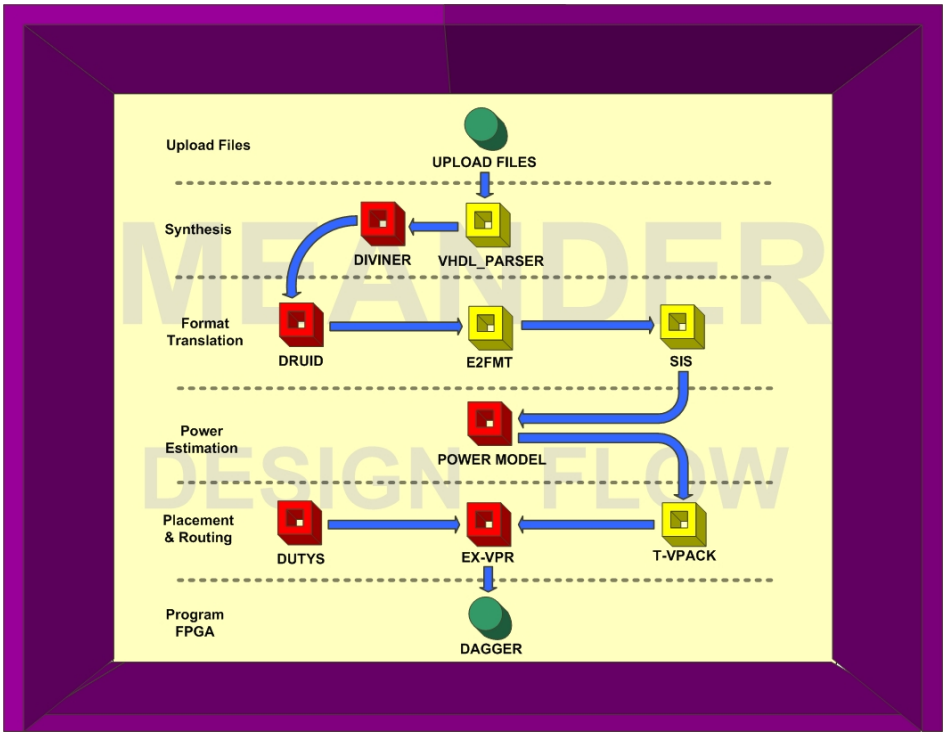


Figure 5.2.: Main screen of GUI-featured MEANDER framework [69]

The *DIVINER* tool performs the synthesis and generates a technology-independent netlist in the EDIF format. The tool supports only a subset of the VHDL standard and the circuits should consist only of logic and flip-flops. This can be a limitation. In that case it is recommendable to use instead Altera Quartus or another powerful external synthesis tool.

DRUID prepares the netlist for the subsequent tools by changing names, simplifying the structure of the network list, and rebuilding elements, that are not included in the libraries of the subsequent tools, using composite library elements. All these modifications are necessary to ensure compatibility with subsequent tools.

The *E2FMT* tool converts the EDIF network list into the Berkeley Logic Interchange Format, which is described in [17].

SIS performs logic minimization and technology mapping. It creates a technology-dependent netlist of LUTs and flipflops in the BLIF format. The number of inputs of the LUT as well as optimization strategies can be specified.

POWER MODEL in conjunction with *EX-VPR* can estimate the power. For the *V-FPGA*, however, in case of virtualization the result is not meaningful because the power depends on the underlying platform and the place & route of the virtual architecture. A higher

accuracy can be obtained by estimating the power for the virtual layer by the vendor tools of the underyling devices.

The tool *T-VPACK* is needed for cluster-based CLBs. It groups a LUT and a flip-flop to a BLE and forms a cluster of several BLEs in a CLB, if this option is activated. The output is a hypergraph netlist in .net format and an updated activity file for power estimation.

The *EX-VPR* tool performs the place & route. The settings options are particularly extensive. For example, you can specify the size of the virtual FPGA (columns, rows, channel width), or leave the corresponding fields empty, and then an architectural exploration is carried out, in which the optimal size is determined automatically. You can specify an I / O mapping (assignment of the network names to the I / O blocks) or this is carried out automatically, minimizing the path lengths to the I/Os. You can influence the place & route strategies by various parameters. *EX-VPR* requires a so-called architecture file, that contains information about the architecture such as switch block type, IO ratio, channel width ratio, LUT size, clustering, location pattern of CLB-inputs and -outputs, as well as process technology related parameters that are needed by the timing and area models. The primary scaling parameters X, Y and W are not part of the architecture file as they are variable and can be left unspecified in order the tool to find the optimum values for a given application.

5.3. Technology Mapping, Place & Route with VTR Design Flow

The VTR (Verilog To Routing) project [87, 63] offers an open-source command-line toolset for FPGA architecture and CAD research. The *V-FPGA* framework uses the VTR tools Odin II, ABC and VPR 7 as alternative to the MEANDER toolset for the tasks of synthesis, technology mapping, packing, place & route. The reasons to employ two similar toolsets for the same tasks are extended support of features and to pick the best raisins from both cakes. For instance, MEANDER supports design entry in VHDL only, while VTR supports only Verilog. MEANDER offers a GUI and remote operation from a web browser, while VTR has the fresher tools with support of heterogeneous blocks and experiences more frequent updates. A good thing about both the tool sets is that they are modular with similar interfaces, thus they can be mixed to benefit from the strengths of both. This is illustrated in Figure 5.1 where the tools can be changed in the flow at certain steps, e.g. after synthesis or after optimization.

In the framework of VTR, Odin II Verilog elaboration front end has the following four vital roles:

1. Converting Verilog syntax into logical netlist
2. Synthesizing constructs directly into 'hard logic' blocks and making use of their logical properties to make them physically realizable
3. Being responsive to the architecture description such as routing architecture, internal structure and global pattern of logic blocks and I/O structure of the FPGA
4. Providing framework to verify the correctness of the software flow

The inputs are a configuration file and the verilog file. The resulting netlist is a BLIF file.

ABC is a synthesis tool operating on And-Inverter Graph (AIG) with a network of two-input AND gates and inverters. AIG represents soft logic, while the hard blocks (e.g., memories and multipliers) received from Odin II are represented as black boxes in ABC. This tool is used for

1. technology independent logic synthesis: ABC's resyn2 script is used for this purpose to minimize the maximum number of AND gates in any combinational path by iteratively calling ABC commands to optimize AIG.
2. technology mapping of AIG into K-input LUTs: Using WireMap technique to produce depth-optimal mappings and attempting to reduce the number of used inputs in the resulting LUTs to benefit its routability, power and efficient packing into dual-output fracturable LUT architectures.

ABC takes as input a BLIF file and a LUT library that defines the inputs and individual propagate delays of a LUT. The resulting netlist can be written in various output formats, while for the *V-FPGA* framework BLIF is most relevant.

In the VPR tool, the architecture file describes interconnection of primitives inside logic blocks along with multiple modes of operation within each piece of the unlimited hierarchical layer. One of the important aspects is its timing-driven physical synthesis in which the timing graph generator inside the timing analyzer generates timing graphs that reflect the arbitrary graph of connectivity inside the complex logic blocks using timing-driven packing algorithm and correctly models timing numbers for each mode of operation thereby making VPR placement and routing timing-driven. VPR needs an architecture file and a BLIF netlist to operate on. The relevant output files for the *V-FPGA* framework are the resulting placement file *.place, routing file *.route and the hierarchical netlist *.net. These files are later imported by the *V-FPGA Explorer* tool to generate the bitstream.

5.4. The *V-FPGA Explorer*

Existing tools are not capable to generate bitstreams for programming the *V-FPGA* for the following reasons:

- Commercial tools (e.g. Altera Quartus II) support only the respective vendor devices.
- Academic tools like VPR were primarily intended for design space exploration and evaluation of CAD algorithms for application mapping and abstract architecture features. Consequently they are working on an abstract representation of hypothetical architectures, that is uncoupled from the actual mechanisms of the architectures.
- Existing tools have no knowledge about the bitstream organization of *V-FPGA*, nor do they offer interfaces for bringing in this information.

A new tool called *V-FPGA Explorer* was developed within this work to close the gap between abstract layout and actual configuration. The primary purpose of the tool is to generate bitstreams out of the textual synthesis, place & route results. Furthermore, a graphical editor allows to edit configurations at the level of CLBs, PSMs and IOBs. This

is also useful for validating the toolchain and the architecture of the *V-FPGA*. Apart from this, the tool provides functions to generate architecture files and scripts for the place & route tools. In order to facilitate simulation of applications mapped onto *V-FPGA*, the tool can generate testbenches that instantiate the device under test with the correct parameters and configure the device, while application specific stimuli can be added in the same testbench file.

The core of the *V-FPGA Explorer* is a graphical representation of the *V-FPGA* as shown in Figure 5.3, which is object-oriented. Each graphical element corresponds to a programmable element in the *V-FPGA* architecture and is realized by an object and contains attributes that describe its properties. The configuration data or the information relevant to the configuration are managed decentrally in the attributes of the affected objects. An array can be addressed either by its instance within an array of objects of the same class or by a unique key in a hash table *HT*. This key corresponds to the name of the object instance which is composed of the typename followed by the identifier, e.g. the 16th object of the class *CLBShape* has the name *clb15*. The use of a hash table eliminates the need to search within the array of objects for a particular object. For instance, the permissible neighbourhood of an object of the class *BoxShape* (which represents a port of a PSM) is defined in the attribute-array *neighbour()*, which contains the names of the valid neighbours according to the SBtype. Thus, finding a neighbour and reading or modifying its attributes (e.g. *source*, *orientation*, *id*, etc.) is simplified by using its name as key in the hash table.

5.4.1. Graphical Editor

The graphical configuration editor is operated by click actions on the graphical objects. To edit the routing in the PSMs, the yellow boxes from the BoxShape class are clicked (first source, then destination). This calls the procedure *ExplorePath()*, whose algorithm is illustrated in Figure 5.5. The first box, which is clicked, is interpreted as an input (or source), the second as an output (or destination). Thus, a directed connection is established from an input to an output of the PSM. When the first box (source) is clicked, the neighbouring boxes according to the selected switch block type (Wilton, Universal or Disjoint), which are not yet occupied, are highlighted in green (see Figure 5.4). Thereby, this neighbourhood relationship, which is different for each switch block type, is stored within the *neighbour()* attribute of a box, i.e. each box knows its three valid neighbours to whom it is allowed to connect. All other boxes that are not valid neighbours are blocked, so they can not be clicked and are not connectable.

At the same time, all accessible paths are identified and highlighted by a path-exploration phase with depth-first search algorithm as in Listing 5.2. Adding existing connections to the *Selection* list avoids collisions (such as two sources driving the same wire) as the list is checked before establishing a new connection. A second list called *Preselection* is used by the *preselect* algorithm for two purposes: one is for the colouring of accessible paths, and the other is in order to avoid infinite loops during the depth-first search of paths through the cyclic graph. This path exploration is performed whenever a source box is clicked and the so-called *PathAssist* option is activated in the settings. When the next box is clicked (destination), a connection is established and the *source* attribute of the new box receives

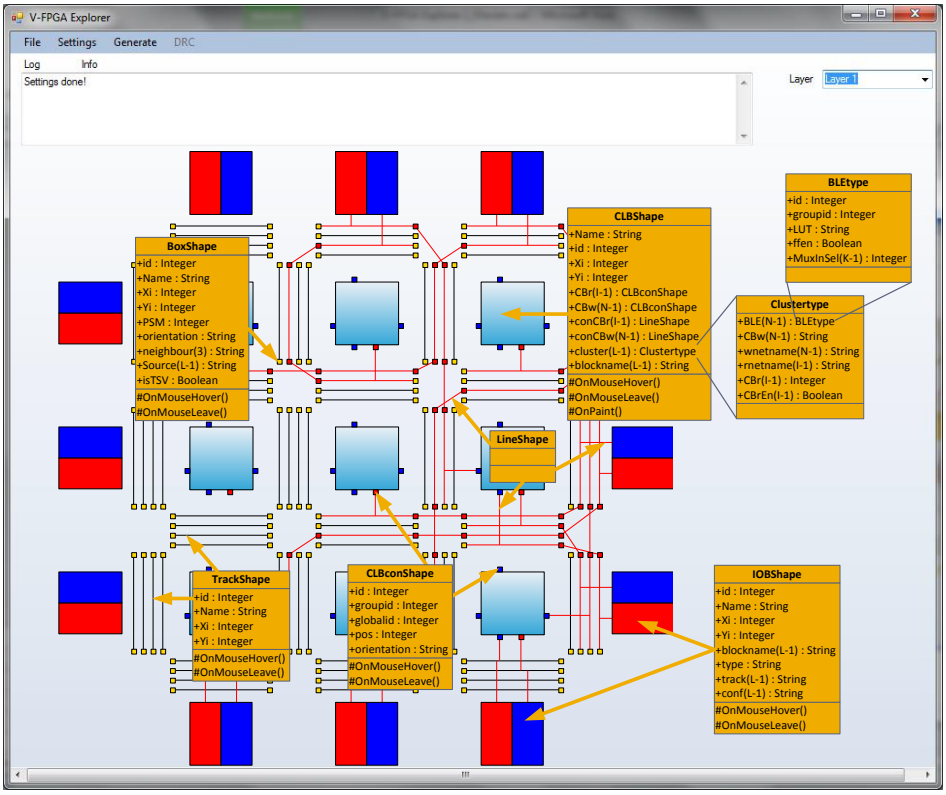


Figure 5.3.: View of object oriented graphical representation in *V-FPGA Explorer* with respective classes and relevant attributes

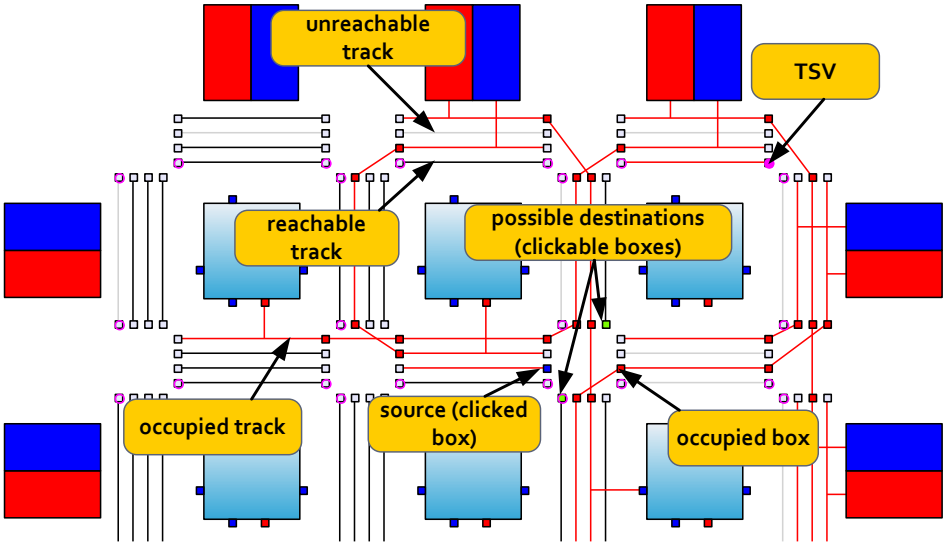


Figure 5.4.: Editing the routing graphically with *V-FPGA Explorer*

the name of the previous box as the value, thus knowing its predecessor. All boxes that are already connected are noted in a selection set. These are then considered occupied and are coloured red, as are the associated tracks. An existing connection can be released by clicking the same box twice. Then the box and the associated track are removed from the selection set and their colours set to their initial values (yellow for the box and black for the track).

```
1  Private Sub Preselect(ByVal sender As System.Object)
2          Dim ThisBox As BoxShape = CType(sender, BoxShape)
3          Dim FictiousSource As String
4          For i As Integer = 1 To 3
5              If ThisBox.neighbour(i) <> "" Then
6                  If (LSelection(selectedL).Selection.Contains(ThisBox.neighbour(i)
                       ) = False) And (Locked.Contains(ThisBox.neighbour(i)) =
                       False) And (Preselection.Contains(ThisBox.neighbour(i)) =
                       False) Then
7                      'Add the neighbour of this box to the preselection list:
8                      Preselection.Add(ThisBox.neighbour(i))
9                      'use a hash table to find the oposite box, that shares the
                           same track associated to the neighbour of this box, and
                           make it a fictious source:
10                     If (HT(ThisBox.neighbour(i)).orientation = "left") Then
11                         FictiousSource = boxR(HT(ThisBox.neighbour(i)).id).Name
12                     ElseIf (HT(ThisBox.neighbour(i)).orientation = "right") Then
13                         FictiousSource = boxL(HT(ThisBox.neighbour(i)).id).Name
14                     ElseIf (HT(ThisBox.neighbour(i)).orientation = "top") Then
```

```
15                          FictiousSource = boxB(HT(ThisBox.neighbour(i)).id).Name
16                      ElseIf (HT(ThisBox.neighbour(i)).orientation = "bottom") Then
17                          FictiousSource = boxT(HT(ThisBox.neighbour(i)).id).Name
18                      End If
19                      'if neither the selection list nor the preselection list
                            contains yet the fictious source, add it to the
                            preselecion list and call this procedure recursively
                            with it:
20                      If (LSelection(selectedL).Selection.Contains(FictiousSource)
                            = False) And (Preselection.Contains(FictiousSource) =
                            False) Then
21                          Preselection.Add(FictiousSource)
22                          Preselect(HT(FictiousSource))
23                      End If
24                  End If
25              End If
26          Next
27      End Sub
```

Listing 5.2: The *Preselect* procedure in *V-FPGA Explorer*

In case of 3D *V-FPGA* the connections are established in a similar way. Some of the PSM-associated boxes feature a magenta-coloured circle, which represents a TSV. To make a vertical connection between two layers, first a TSV needs to be clicked and then the layer needs to be changed using the layer selector. This results in a vertical connection between the previous and the new layer and the associated boxes and tracks are considered occupied (are added to the selection set) and are coloured respectively. A vertical connection causes the *source* attribute of the connected box to be set to either *"zdown"* or *"zup"* depending on whether the connection was established with a lower layer or with a higher layer.

IOBs and CLBs are connected in a similar way with tracks in proximity. A click on a box of the *CLBconShape* class or of the *IOBShape* class followed by a click on a track will establish the connection. The connection can be released by double click on the same box.

The editing of CLB configuration is solved by a seperate table-based form as shown in Figure 5.6, that pops up upon a click on a CLB shape. The first table sets the configurations within the BLEs of a CLB. Therein, each row is associated with one BLE. The first column is the identifier of a BLE. The second column is the block name, which usually corresponds to the name of the net that is connected with the BLE output (which can be also interpreted as a node in the netlist). This field is especially important for associating a function mapping to a placed node during import from external tools. Therefore it is also added to a hash table as key for an associated BLE, i.e. the BLE can be now also addressed by the associated node of a mapped application netlist (usually the .net hyper-graph in *T-VPACK* or *VPR tool*). The third column is the LUT configuration, whereby the leftmost bit represents the highest address in the LUT and the rightmost bit the lowest address. The fourth column controls the bypass MUX for the flipflop at the output of the LUT. The other columns control the input multiplexers of the BLEs, whereby each BLE has *K* inputs and input multiplexers the establish connections with either CLB inputs or with BLE outputs. A graphic shows a preview of the connections set in the first table and gets updated with

every change. The second table controls the CLB in- and outputs, i.e. the corresponding connection boxes *CBr* and *CBw*. It is an alternative yet fully linked approach to the graphical editing through click actions on the associated boxes. The connections are established by entering the desired track number or pattern. The first, third and fourth columns are not editable as they refer to the identifiers and locations of the associated I/Os of a CLB. The second column refers to the name of a net (of the application) connected to a CLB port. The fifth column selects the tracks that should be connected, whereby CLB inputs are set to one of the track numbers within a routing channel and CLB outputs are set to a pattern the connects to one or more tracks within a routing channel simultaneously. The sixth columns enables or disables the inputs. In contrast to [50] this is not an explicit feature of the new *V-FPGA* architecture but is solved through bit manipulation in the LUT, i.e. there is always a track (e.g. track0) connected to an input, yet when a LUT input is unused then it is considered as *don't care* in the synthesis of the LUT results. It is to be noted that the structure of the CLB configuration form, including the tables and the preview, is not fixed but gets generated according to the architectural parameters that are set or imported. All changes (whether manually or through import form external tools) are also updated in the attributes of the internal object-oriented graphical representation.

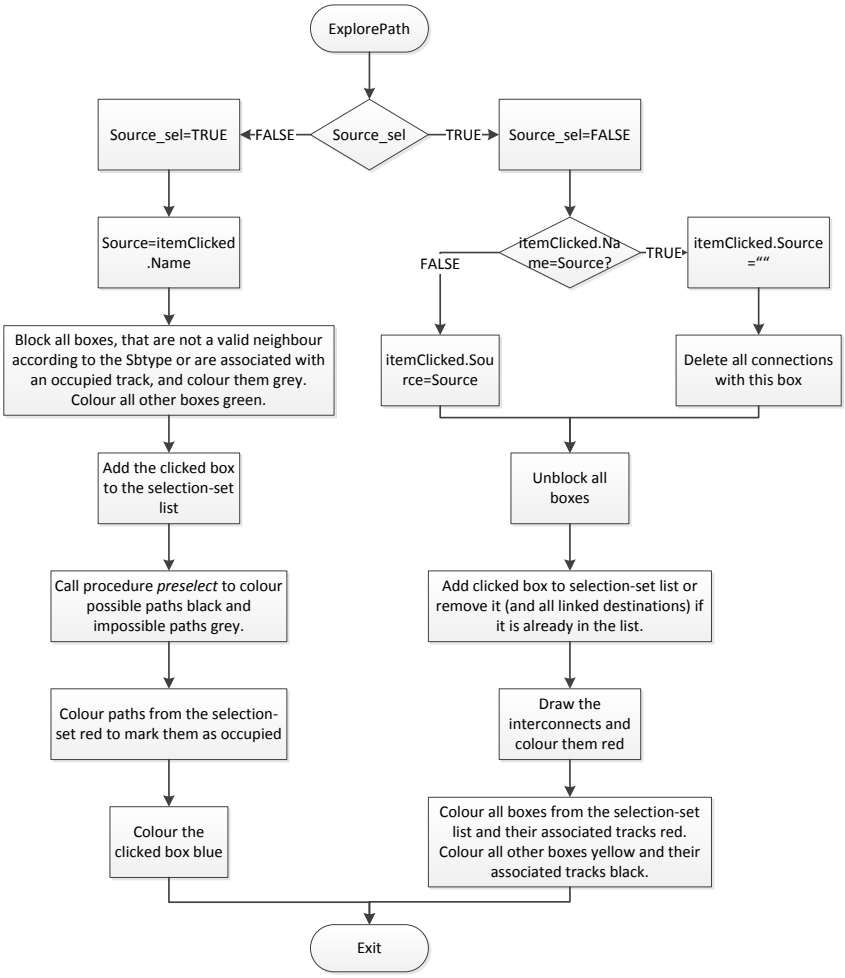


Figure 5.5.: Flow chart of *ExplorePath* algorithm in *V-FPGA Explorer*

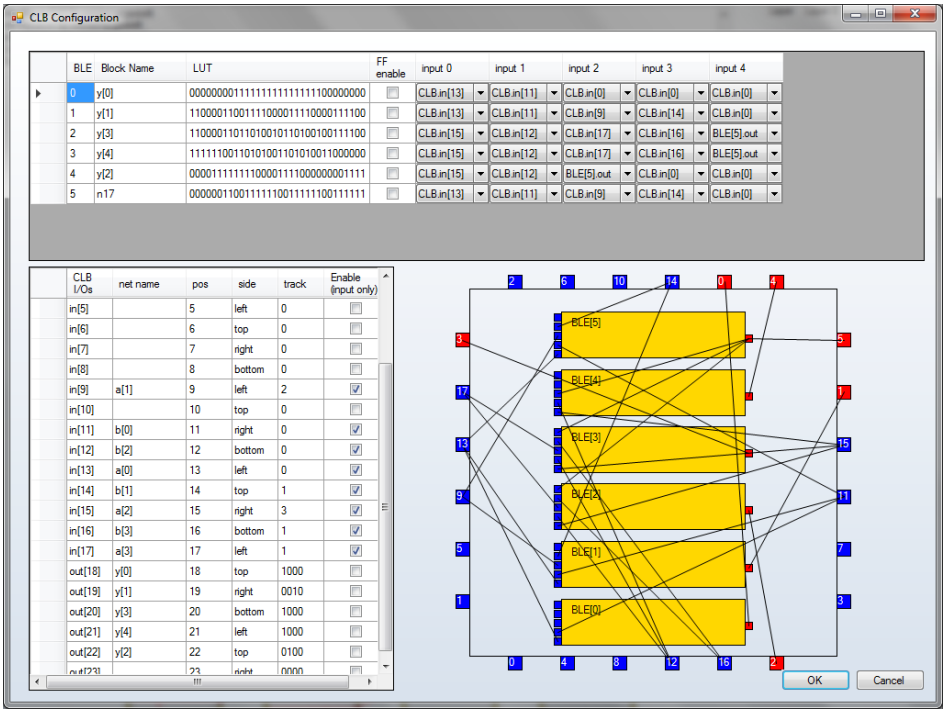


Figure 5.6.: Editing of CLB configuration with *V-FPGA Explorer*

5.4.2. Import from External P&R Tools

With *V-FPGA Explorer* it is possible to import synthesis and layout results from tools like *VTR* or *MEANDER*, which will be then transformed into the internal and graphical representation, which can be afterwards edited and further transformed into configuration bitstreams for programming the *V-FPGA*. As it can be seen from the import dialog in Figure 5.7, the required files are *.place (placement), *.route (routing), *.blif (LUT-mapped netlist), *.net (packed hypergraph netlist) and vpr_stdout.log (log file with summary of results and parameters).

The import procedure is divided in four steps in the following order, which interestingly is exactly the inverse order of the steps in *VTR* or *MEANDER*:

1. **Construct.** This step extracts the architectural parameters and generates the graphical representation with the corresponding data structure (see Section 5.4.2.1 for more details).
2. **Route.** During this step, global routing information is extracted and the paths realising it are set up (see Section 5.4.2.2 for more details). The reason why this step comes before the placement and the mapping imports is that the routed signals have net names that can be used to identify the associated CLB IOs and thus to correctly allocate BLEs, derive the correct MUX select settings and reorder the LUTs if necessary.
3. **Place.** This step associates placed nodes with BLEs, CLBs and IOBs at the corresponding locations (see Section 5.4.2.3 for more details).
4. **Map.** This step sets the contents of the placed LUTs (see Section 5.4.2.1 for more details).

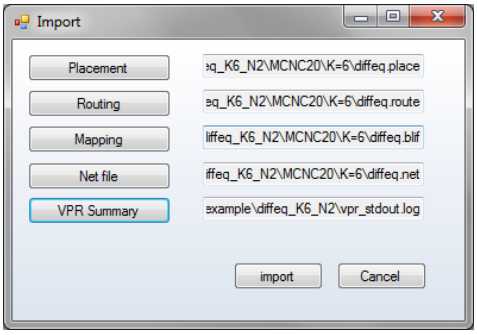


Figure 5.7.: Dialog for file import from *VTR* or *MEANDER* into *V-FPGA Explorer*

5.4.2.1. Construct

In order to prepare the data structure and the graphical representation of the *V-FPGA*, first the architectural parameters (see Table 4.1) upon which the application mapping was based need to be extracted from the outputs of the external tools. Fortunately, all the relevant output files generated by the employed tools in Figure 5.1 are in human readable ASCII format, which makes it easy to the locate and understand the needed information without the knowledge of data structure specification. For the *V-FPGA Explorer* this means that the extraction of information is done by string search and manipulation mechanisms. This is not as efficient as reading the information from the exact address in a known data structure, yet it is the most flexible way that can cope with multiple tool versions and data formats. It is thinkable to outsource the keyword definitions in an external file in order to allow easy adoption to future output formats. Up to now, the outputs of *MEANDER* and *VTR* have been tested, which have a similar format.

The parameters X and Y are obtained from the routing file by searching for the string that contains "Array size: <*X_parameter*> x <*Y_parameter*> logic blocks" and extracting the X and Y parameters. Since this is usually the first line in a routing file (see Listing 5.3) it is found very quickly. The parameter W is extracted from the string "Best routing used a channel width factor of <*W_parameter*>" and the parameter $SBtype$ from the string "RoutingArch.switch_block_type: <*SBtype_parameter*>", both found in the log file of *VPR*. K and N parameters are encapsulated in the architecture file name and are extracted from the string "Architecture file: arch_VFPGA_K <*K_parameter*>_N<*N_parameter*>" which is found in the place file. Other parameters such as I and O are derived from K and N as described in Section 4.1.5.1.

Based on the obtained parameters the graphical representation of an empty *V-FPGA* is set up by instantiating the objects depicted in Figure 5.3 and setting their size and locations. For the objects of the class *BoxShape* also the permissible neighbourhood relationship according to the extracted $SBtype$ parameter and the corresponding pattern schemes (see Table 4.4 to 4.6) is set.

5.4.2.2. Route

In order to realise the paths of the calculated routing, the routing file, which describes chains of traversed coordinates, is analysed and the attributes of the routing objects in the graphical representation are derived accordingly. Listing 5.3 shows an extract of a routing file. A net always starts with a source and ends in one or more sinks. In between, the traversed channels (CHANX in x-direction and CHANY in y-direction) with their coordinates and track numbers are listed. Coordinates are given in brackets with the format (x,y), whereby the coordinate system in Figure 5.8a is used. In case a net has two sinks, after reaching the first sink the path towards the second sink is continued from the branching location. For instance, in line 13 the first sink at (12,2) is reached, followed by line 14 that is a jump back to the branching coordinate CHANY(10,2) and then continuing towards the second sink which is located in (13,3). Sources and sinks can be either CLBs or IOBs. *V-FPGA Explorer* differentiates this by the coordinates, i.e. coordinates on the perimeter indicate IOBs while the other coordinates indicate CLBs. Compared to the *VPR* coordinate system, the graphical representation in *V-FPGA Explorer* has an offset of -1 in

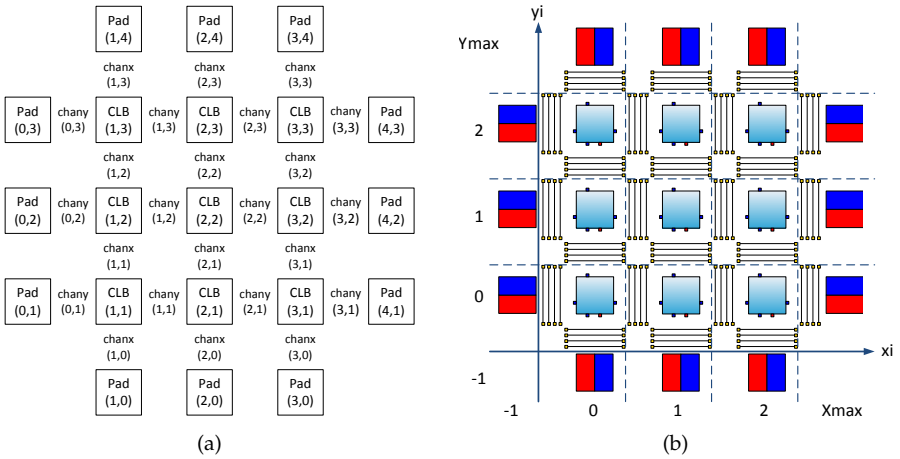


Figure 5.8.: Difference between coordinate systems in (a) *VPR* and (b) *V-FPGA Explorer*

both x- and y-direction (see Figure 5.8). Therefore, a transformation of coordinates needs to take place.

The routing import in *V-FPGA Explorer* is divided in two phases:

1. Derive the global routing through the PSMs (see Figure 5.9): This phase is based on trailing changes in the channel coordinates, identifying predecessor and successor tracks of connections through intersections and setting the source attributes of associated objects of the *BoxShape* class accordingly while checking validity of neighbourhood with respect to the selected switch block type.
2. Derive the local routing through the connection boxes (see Figure 5.10): During this phase, connection boxes and IOBs are identified (based on coordinates and pin numbers) and configured to connect with tracks in their proximity according to the routing information. Furthermore, connection boxes are associated with net names, which is needed for the next steps in order to correctly make the connections of BLE inputs with CLB inputs. Even though BLE connections make also part of the local routing, they are established during the map step described in Section 5.4.2.4 because they are related to the function mapping and the LUT ordering.

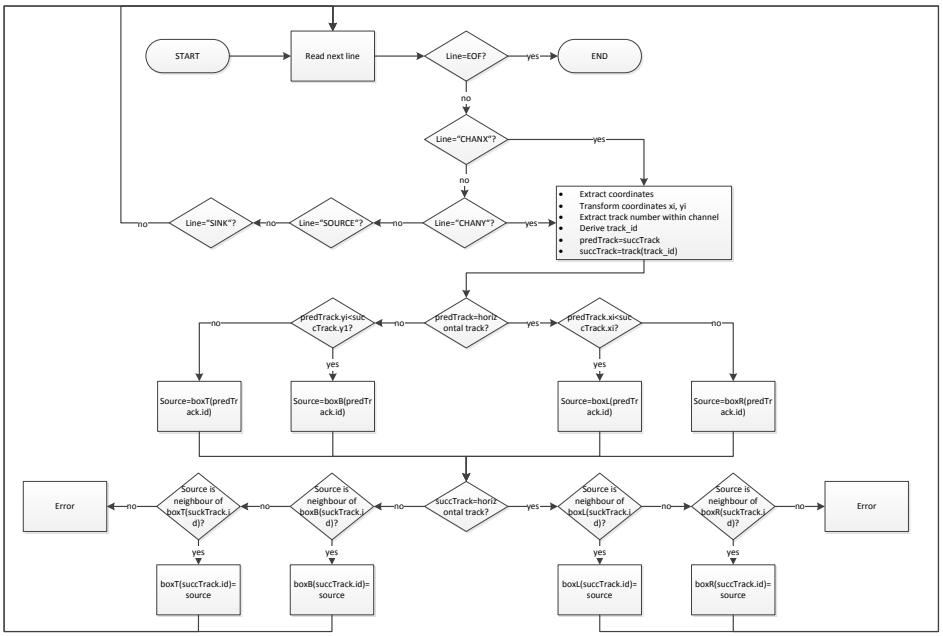


Figure 5.9.: Import of global routing information in *V-FPGA Explorer*

```
1  Array size: 21 x 21 logic blocks.
2
3  Routing:
4
5  Net 6 (n_n4198)
6
7  Node: 3881  SOURCE (11,2)  Class: 1
8  Node: 3893    OPIN (11,2)  Pin: 9
9  Node: 17499  CHANY (10,2)  Track: 13
10 Node: 8498   CHANX (11,1)  Track: 0
11 Node: 8512   CHANX (12,1)  Track: 0
12 Node: 4248     IPIN (12,2)  Pin: 4
13 Node: 4240     SINK (12,2)  Class: 0
14 Node: 17499  CHANY (10,2)  Track: 13
15 Node: 17513  CHANY (10,3)  Track: 13
16 Node: 3901     IPIN (11,3)  Pin: 1
17 Node: 3896     SINK (11,3)  Class: 0
```

Listing 5.3: Extract of a routing file generated by *VPR 7*

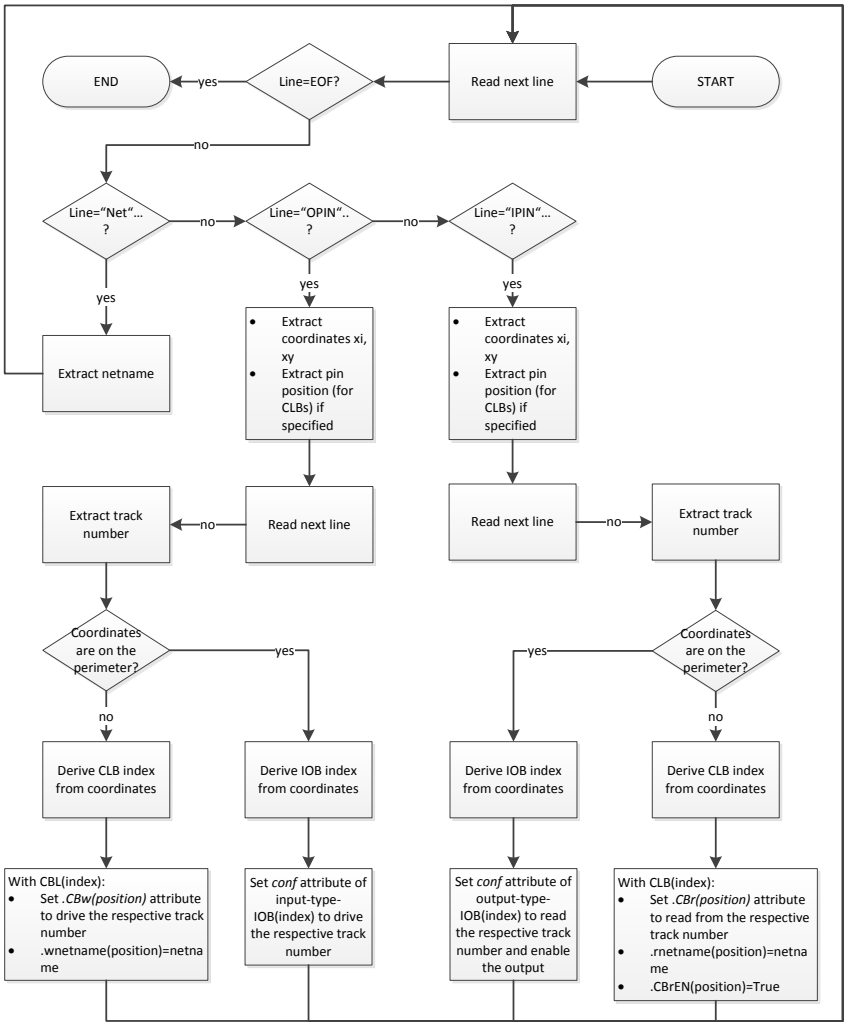


Figure 5.10.: Import of local routing information in *V-FPGA Explorer*

5.4.2.3. Place

The purpose of this step is to associate node names (also called *blocknames* by the external tools) within the netlist with CLBs, BLEs and IOBs.

The placement information for CLBs and IOBs imported from the placement file. An exemplary extract of a placement file is given in Listing 5.4. The placement information is structured in a table format. The first column contains the block names. The second and the third column contain the coordinates of the placed blocks. The fourth column contains sub-block identifiers and is only needed for IOBs that contain more than one I/O per block. The last column contains block numbers, that are used by *VPR* internally instead of the block names, and is not relevant for the import into *V-FPGA Explorer*. During the import process this table is read row by row, whereby the block name of a node is extracted as well as its coordinates. The coordinates are transformed into the coordinate system of *V-FPGA Explorer*. Depending on the location it is distinguished whether the node is placed on in an IOB (if the coordinates point on the perimeter of the array) or on an CLB and based on the type the identifier is determined. In case of an IOB, the differentiation whether a node is an input or an output is made by the prefix of the block name, i.e. the block names of outputs start always with *"out:"* while inputs have no prefix. Finally the *blockname* attribute of an associated CLB or input-type IOB or output-type IOB at the specified location is updated with the extracted block name from the placement table. Additionally the block name is also added to the hash table as a key for the associated object.

```
1  Netlist file: MCNC20/K=6/diffeq.net   Architecture file: arch_VFPGA_K6_N2.xml
2  Array size: 21 x 21 logic blocks
3
4  #block name x y subblk  block number
5  #---------- -- -- ------ ------------
6  n_n3847    11  3 0 #0
7  n_n3859    13  1 0 #1
8  n_n3403    12  4 0 #2
9  [1347]     11  2 0 #3
10 [841]    12  2 0 #4
11 [7392]     11  4 0 #5
12 n_n3925    14  6 0 #6
13 n_n3888    15  7 0 #7
```

Listing 5.4: Extract of a placement file generated by *VPR 7*

In case of clustering, the locations of nodes mapped onto BLEs within CLBs are obtained from the packed hypergraph netlist *.net*. An example is shown in Listing 5.5 (note that some contents are replaced by "..." for space reasons). The data structure is hierarchical, so that BLE blocks are nested within a CLB block. There is no explicit location information because the packed netlist is generated before the placement process takes place. However, the CLB block names correspond to the ones used in the placement file and due to their inclusion in the hash table as keys for the CLB objects there is already an association. Consequently the remaining task is to associate BLEs with the placed CLBs in the right

order. For instance, in the example from Listing 5.5 the following steps take place during the import:

1. Extract the block name of a CLB from lines that contain the keywords *block name* and *mode="clb"* (e.g. from line 5).
2. Use the hash table to get the id of the CLB that is already associated with that block name.
3. From subsequent lines that contain the keywords *block name* and *mode="ble"* (e.g. from line 15 and line 28) extract the block names associated with the nested BLEs and the BLE indices.
4. Update the attributes *clb(id).ble(index)=ble_blockname*.
5. Add the BLE block names to the hash table as key for the associated BLE objects.
6. Repeat steps 1 to 5 with the next CLB blocks until the end of file is reached.

```
1  <block name="MCNC20/K=6/diffeq.net" instance="FPGA_packed_netlist[0]">
2    <inputs> ... </inputs>
3    <outputs> ... </outputs>
4    <clocks> ... </clocks>
5    <block name="n_n3847" instance="clb[0]" mode="clb">
6      <inputs>
7        <port name="I">[7392] n_n3912 [797] [1347] open n_n3403 n_n4069 open
             n_n4198 </port>
8      </inputs>
9      <outputs>
10       <port name="O">open ble[1].out[0]->clbouts1  </port>
11     </outputs>
12     <clocks>
13       <port name="clk">pclk </port>
14     </clocks>
15     <block name="[1348]" instance="ble[0]" mode="ble">
16       <inputs>
17         <port name="in">ble[1].out[0]->crossbar  clb.I[1]->crossbar  clb.I[2]->
               crossbar  clb.I[8]->crossbar  clb.I[5]->crossbar  clb.I[6]->crossbar
                 </port>
18       </inputs>
19       <outputs>
20         <port name="out">lut[0].out[0]->mux1  </port>
21       </outputs>
22       <clocks>
23         <port name="clk">open </port>
24       </clocks>
25       <block name="[1348]" instance="lut[0]" mode="lut"> ... </block>
26       <block name="open" instance="ff[0]"/>
27     </block>
28     <block name="n_n3847" instance="ble[1]" mode="ble">
29       <inputs>
30         <port name="in">clb.I[3]->crossbar  ble[0].out[0]->crossbar  clb.I[0]->
               crossbar  open open open </port>
31       </inputs>
32       <outputs>
```

```
33          <port name="out">ff[0].Q[0]->mux1  </port>
34        </outputs>
35        <clocks>
36          <port name="clk">clb.clk[0]->clks  </port>
37        </clocks>
38        <block name="n_n3847" instance="lut[0]" mode="lut"> ... </block>
39        <block name="n_n4197" instance="ff[0]"> ... </block>
40      </block>
41    </block>
```

Listing 5.5: Extract of a packed netlist file generated by *VPR 7*

5.4.2.4. Map

The file with the suffix .blif is the result of the technology mapping by the tool *SIS* in MEANDER or the tool *ABC* in VTR. This ASCII file contains a netlist in the BLIF [17] format with LUTs (keyword *.names*), flipflops (keyword *.latch*), inputs (keyword *.inputs*) and outputs (keyword *.outputs*):

```
1  .model top
2  .inputs tin_puport_10_10_ tin_puport_0_0_ tin_pxport_10_10_ tin_pxport_0_0_ \
3  pdxport_4_4_ tin_puport_1_1_ tin_pxport_1_1_ tin_puport_2_2_ tin_pyport_10_10_ \
4  ...
5  .outputs puport_10_10_ puport_0_0_ pxport_10_10_ pxport_0_0_ puport_1_1_ \
6  pxport_1_1_ puport_2_2_ pyport_10_10_ pyport_0_0_ pxport_2_2_ puport_3_3_ \
7  ...
8  .latch    n_n3032 n_n3033 re pclk 2
9  .latch    n_n2959 n_n2960 re pclk 2
10 ...
11 .names tin_puport_10_10_ n_n3430 n_n4068 puport_10_10_
12 10- 1
13 -11 1
14 .names tin_puport_0_0_ n_n3381 n_n2982 puport_0_0_
15 -11 1
16 1-0 1
17 ...
18 .end
```

Listing 5.6: LUT-mapped netlist in BLIF format

Thereby, LUTs are simply defined by their onsets (1) or offsets (0), whereby the use of *don't cares* ("-") reduce the file size.

In *V-FPGA Explorer* the import from the .blif file has basically two goals:

1. Identify sequential nodes: Lines that start with the keyword *.latch* refer to flip-flop nodes. Thereby, there are two nodes specified: the input of a flip-flop (that is the node name of the LUT that drives the data-in signal) and the output of the flip-flop. However, in *V-FPGA Explorer* the BLE as the union of a LUT and a flip-flop has only

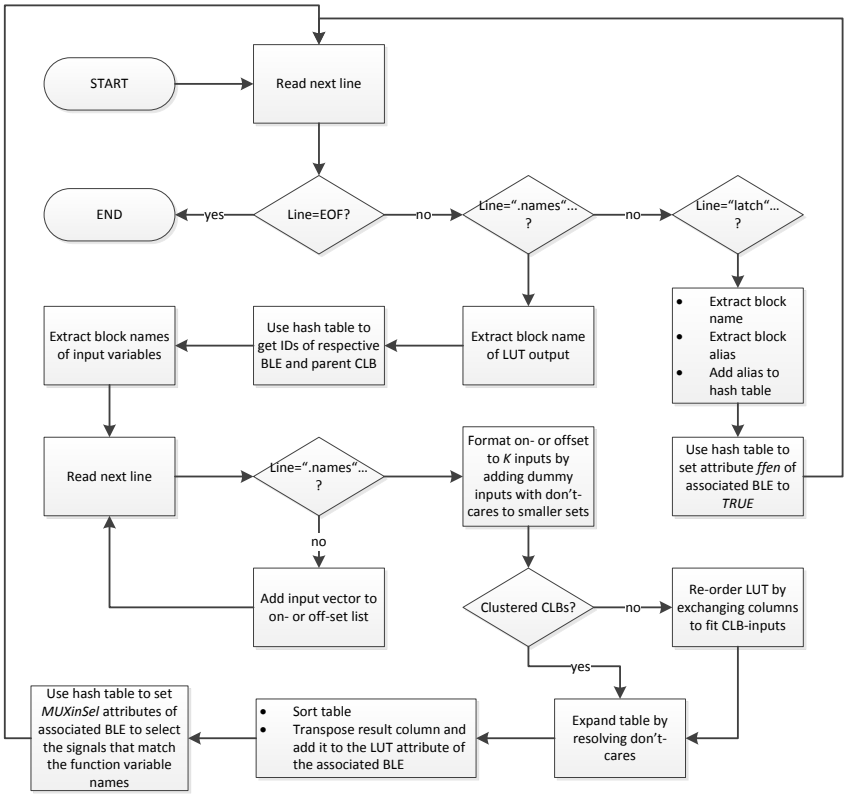


Figure 5.11.: Import of function mapping information in *V-FPGA Explorer*

one block name. Therefore, aliases are created and added to the hash table in order to associate both node names with the same BLE.

2. Derive the LUT contents: Lines that start with the keyword *names* contain the node names of the input variables followed by the node name of the LUT output. The subsequent lines contain the compressed on- or off-set of a LUT. This set is formatted to K variables (i.e. missing variables are substituted by dont-cares) and expanded to resolve the don't-cares and obtain the full table. I case of non-clustered CLBs a re-ordering of the full table takes place, in order to comply with the direct-connected CLB inputs.

Figure 5.11 illustrates the steps that are done in *V-FPGA Explorer* for these purposes.

After the function mapping import for all nodes is done, the import process is finished. Further editing, bitstream generation etc. will be done directly on the internal data structure encapsulated in the graphical representation of the *V-FPGA Explorer*. The graphics output can be disabled in the settings in order to remarkably speed-up tool-time for large

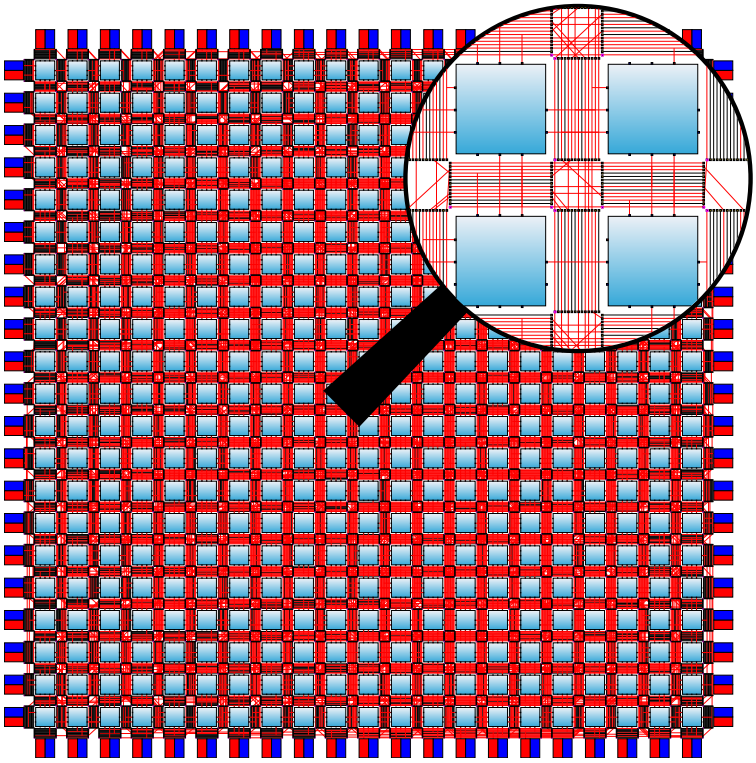


Figure 5.12.: *V-FPGA Explorer*: Layout of *diffeq* circuit mapped onto *V-FPGA* after import from *VPR 7*

circuits. Yet the graphical representation can be exported anytime, irrespective of enabling or disabling the graphics output.

Figure 5.12 shows the import results of the *diffeq* circuit from the MCNC benchmark suite [111], which was mapped with the *VPR 7* tool onto the *V-FPGA* with the parameters $K = 6$, $N = 2$, $SBtype = Wilton$, $W = 14$, $X = 21$, $Y = 21$. The resulting files of *VPR* were then imported into *V-FPGA Explorer*. The shown graphic is exported from *V-FPGA Explorer* tool in a Scalable Vector Graphics format. Thus, in the digital version of this document it is possible to zoom inside for a more detailed view.

5.4.3. Bitstream Generation

Bitstream generation works by evaluating the attributes of the graphical objects relevant to the configuration data and concatenating them as a string corresponding to the bit positions in the configuration registers (see Table 4.2, 4.3, 4.7 and 4.8).

The bit stream for a CLB with the index h on the layer *lindex* is composed as shown in Listing 5.7. First the selects of all *CBw* are concatenated, followed by the selects of all *CBr*. Then the selects of all BLE input multiplexers are concatenated. Finally the BLE bitstreams are added, that contain the LUT result and a bit for controlling the bypass MUX. The order is such that the element with the highest index is left. That's why the loop variables are counted down.

```
1  For n As Integer = parN - 1 To 0 Step -1
2    CBw_config = CBw_config & clb(h).cluster(lindex).CBw(n)
3    For in_i As Integer = parI - 1 To 0 Step -1
4      CBr_config = CBr_config & dec2bin(clb(h).cluster(lindex).CBr(in_i), ld(parW -
           1))
5    Next
6    For k As Integer = parK - 1 To 0 Step -1
7      BLE_inMux_config = BLE_inMux_config & dec2bin(clb(h).cluster(lindex).BLE(n).
           MuxInSel(k), ld(parN + parI - 1))
8    Next
9    BLE_config = BLE_config & clb(h).cluster(lindex).BLE(n).LUT
10   If clb(h).cluster(lindex).BLE(n).ffen = True Then
11     BLE_config = BLE_config & "1"
12   Else
13     BLE_config = BLE_config & "0"
14   End If
15 Next
16 LcConfig(lindex).cConfig(h) = CBw_config & CBr_config & BLE_inMux_config &
       BLE_config
```

Listing 5.7: Pseudocode for assembling the bitstream of one CLB

The bit stream of an IOB with the index h is composed of the *.conf* attribute of the output with the index h and the *.conf* attribute of the input with the index h:

```
1  LioConfig(lindex).ioConfig(h) = output(h).conf(lindex) & input(h).conf(lindex)
```

Listing 5.8: Pseudocode for assembling the bitstream for one IOB

The bit stream of a PSM requires more effort in its generation, as shown Listing 5.9. Initially, all bits of the PSM bitstreams $.pConfig(h)$ are set to '0'. Then the objects from the BoxShape class are evaluated. Each box represents one of many ports (with input and output) of a PSM. The associations of a box with a PSM is given by the attribute *PSM* in the box. Each box is responsible for exactly 2 bits in 2D *V-FPGA* or 3 bits in 3D *V-FPGA* of the configuration register of a PSM. These are the select signals for the output multiplexer of the corresponding port (see Table 4.7). These bits are determined by evaluating the *source* attribute of the box. For instance, in a 3D *V-FPGA*, if *source* is empty, the bits at the corresponding locations in the bitstream are set to "000". If it is "zdown" or "zup", i.e. sourced from a lower or an upper layer through a TSV, then the bits are set "100" or "101" respectively. If *source* is identical to the value from the neighbour attribute (1), the bits are set to "001". If *source* is identical to the value from the neighbour attribute (2), the bits are set to "010". If the *source* is identical to the value from the neighbour attribute (3), the

bits are set to "011". In case of 2D *V-FPGA* there are only 2 select bits per MUX and they correspond to the lower 2 bits of of the 3D case. Therefore, in the 2D case the bitstream is reduced such that every third bit is deleted.

```
For h As Integer = 0 To UBound(boxL)
  ' left orientation
  LpConfig(lindex).pConfig(boxL(h).PSM) = LpConfig(lindex).pConfig(boxL(h).PSM).
      Remove(((12*parW)-1)-(11+(12*(boxL(h).id Mod parW))),3)
  If boxL(h).Source(lindex) = "" Or boxL(h).Source(lindex) = boxL(h).Name Then
    sel="000"
  ElseIf boxL(h).Source(lindex) = "zdown" Then
    sel="100"
  ElseIf boxL(h).Source(lindex) = "zup" Then
    sel="101"
  Else
    Select Case boxL(h).Source(lindex)
      Case boxL(h).neighbour(1)
        sel="001"
      Case boxL(h).neighbour(2)
        sel="010"
      Case boxL(h).neighbour(3)
        sel="011"
      Case Else
        sel="000"
    End Select
  End If
  LpConfig(lindex).pConfig(boxL(h).PSM) = LpConfig(lindex).pConfig(boxL(h).PSM).
      Insert(((12*parW)-1)-(11+(12*(boxL(h).id Mod parW))),sel)

  ' right orientation
  LpConfig(lindex).pConfig(boxR(h).PSM) = LpConfig(lindex).pConfig(boxR(h).PSM).
      Remove(((12*parW)-1)-(5+(12*(boxR(h).id Mod parW))),3)
  ...
  ...
  LpConfig(lindex).pConfig(boxR(h).PSM) = LpConfig(lindex).pConfig(boxR(h).PSM).
      Insert(((12*parW)-1)-(5+(12*(boxR(h).id Mod parW))),sel)
Next
For v As Integer = 0 To UBound(boxT)
  ' top orientation
  LpConfig(lindex).pConfig(boxT(v).PSM) = LpConfig(lindex).pConfig(boxT(v).PSM).
      Remove(((12*parW)-1)-(2+(12*(boxT(v).id Mod parW))),3)
  ...
  ...
  LpConfig(lindex).pConfig(boxT(v).PSM) = LpConfig(lindex).pConfig(boxT(v).PSM).
      Insert(((12*parW)-1)-(2+(12*(boxT(v).id Mod parW))),sel)

  ' bottom orientation
  LpConfig(lindex).pConfig(boxB(v).PSM) = LpConfig(lindex).pConfig(boxB(v).PSM).
      Remove(((12*parW)-1)-(8+(12*(boxB(v).id Mod parW))),3)
  ...
  ...
  LpConfig(lindex).pConfig(boxB(v).PSM) = LpConfig(lindex).pConfig(boxB(v).PSM).
      Insert(((12*parW)-1)-(8+(12*(boxB(v).id Mod parW))),sel)
```

```
42  Next
43  If parL < 2 Then    ' reduce bitstream if only 1 layer is present in order to
        target 4:1 Muxes for 2D instead 6:1 Muxes for 3D
44    For i As Integer = 0 To UBound(LpConfig(lindex).pConfig)
45      For b As Integer = 0 To (4 * parW) - 1
46        LpConfig(lindex).pConfig(i) = LpConfig(lindex).pConfig(i).Remove((b * 2),
              1)
47      Next
48    Next
49  End If
```

Listing 5.9: Pseudocode for assembling the bitstream for one PSM

There is each an array in which the bitstreams of all CLBs, all PSMs and all IOBs are stored for internal purposes during the generation of the bit stream (*cConfig*, *pConfig* and *ioConfig*). Bitstream files in different output formats for configuration controller, simulation, etc. are created from the internal arrays.

5.4.4. Architecture File Generator

VPR requires a so-called architecture file that describes the topology of the target architecture as well as technology parameters. This is needed for packing, timing driven place&route and estimation of area and performance. An example of such an architecture file for the *V-FPGA* is given in Section A.2. Usually, architectural parameters such as LUT size, switch block type, cluster size, I/Os per CLB and their distribution, etc. are defined in an architecture file. In case of *V-FPGA* some of the device parameters are also dependent on the architecture parameters. Only simple scaling parameters such as channel width, total rows and columns of CLBs can be determined by VTR during its DSE phase and do not need to be included in the architecture file. The disadvantage of fixing parameters in the architecture file is that they reduce the design space that is explored. The extended DSE methodology presented in Section 6.4 however needs to sweep through these parameters in order to extend the design space and improve the quality of customization. Especially LUT size and cluster size are two very sensitive parameters that have a significant impact on performance and area as we have shown and discussed in [34], therefore they need to be variable in DSE. For the *V-FPGA* this is solved by multiple architecture files, each with a different combination of these parameters, thus DSE is extended by multiple runs each with another architecture file.

To facilitate the creation of such files, that can be very numerous depending on the intended design space, the *V-FPGA Explorer* incorporates an architecture file generator. A set of few primary parameters (K, N, SB_{type}), their sweep-ranges and the MSBE characteristics (e.g. normalized area of MUX2 and D-type flip-flop, propagation delays of MUX2 and average net segments, setup-time and clock-to-Q dealy of a flip-flop) are defined by the user in a GUI mask as shown in Figure 5.13 with bold font.

The other parameters can be also defined by the user or can be derived from the primary parameters by pressing the *AutoFill* button. Then they are derived as follows:

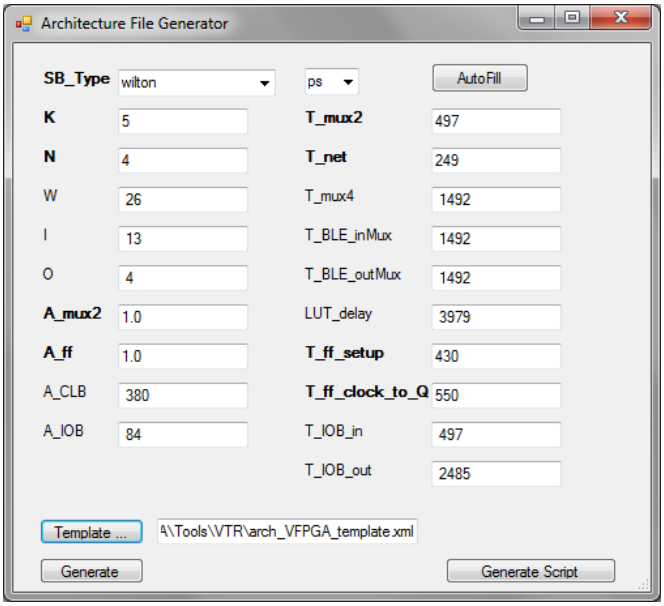


Figure 5.13.: Settings for architecture file generation in *V-FPGA Explorer*

- Following the recommendations of [4], the inputs I per clustered CLB are derived from K and N as $I = \left\lceil \frac{K}{2} \cdot (N+1) \right\rceil$.
- The outputs O per CLB are identical to the cluster size.
- The channel width W is more difficult to derive as it actually depends on the application mapping and is determined by VPR. The reason why it is needed here is that the area and timing of IOBs and CBs mainly depend on the channel width. A curve fitting function that approximates the W from the average over the 20 largest MCNC benchmarks is derived from K and N as follows:
  $W = \left\lceil 5 + (0.7 \cdot K) + \left((3.2 - (0.35 \cdot K)) \cdot \left\lceil \frac{K}{2} \cdot (N+1) \right\rceil \right) \right\rceil$
  As already mentioned, this number will not be accurate for each application, but is a good starting point for an iterative refinement as stipulated by the DSE methodology in Section 6.4 where a feedback loop updates this value with the actual one after the first iteration.
- The rest parameters are derived from the MSBEs by applying the area and delay models described in Section 5.6.

An architecture file template contains tags that mark positions where the parameters should be inserted during the architecture file generation. If K and/or N are specified as ranges, then the architecture file generator automatically sweeps through these ranges, re-derives the other parameters for each K and N combination and generates each a separate

architecture file. Furthermore, a script file is generated to automatically run all necessary steps of VTR with the various architecture files in batch mode when executed.

5.4.5. Automated Testbench Generation with *V-FPGA Explorer*

The *V-FPGA Explorer* generates VHDL testbench files with instantiation of a *V-FPGA*with the correct parameters and with stimuli of the AMBA APB signals with which the generated bit stream data is copied into the RAM blocks of the configuration controller and then the reconfiguration command is given. Users can specify a prefix for the signal names, the clock period of the APB clock PCLK, the start of the transmission, the basic addresses of the RAM blocks and the offset in the word alignment of the addressing scheme (see Figure 5.14.

Furthermore, it is possible to import an existing testbench as a frame. The stimuli data is then automatically inserted there, the test signals are declared, and the generic scaling parameters are also specified when the virtual FPGA is instantiated. For this purpose, a testbench frame must contain corresponding tags (see Section A.3) as comments in the right places. The *post synthesis* checkbox allows to specify whether the testbench is used for pre-synthesis or post-synthesis (and post-layout) simulation. Te difference is that in the post-synthesis simulation no generics are allowed to be used during the instantiation in the testbench because the netlist is already synthesized and therefore in no generics are available in the entity.

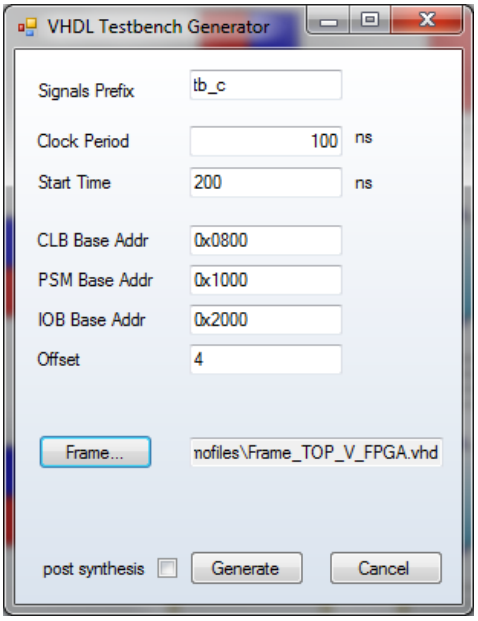


Figure 5.14.: Settings for testbench generation in *V-FPGA Explorer*

5.5. Just-in-Time Compilation

In collaboration with the National Technical University of Athens (NTUA), a Just-in-Time (JIT) compilation methodology for in-system dynamic application mapping onto *V-FPGA* was developed [91], [92]. The idea is to perform fast application mapping during run-time in order to target adaptive systems and to allow fine-grain defragmentation. NTUA focused on JIT floor-planning, placement and routing and KIT contributed with JIT bit-stream generation and with the fine-grained reconfigurable *V-FPGA* architecture.

The ability to dynamically modify blocks of logic without interrupting the ongoing logic operation being referred as partial reconfiguration not only adapts algorithm to share hardware resources, but also to improve resource utilization and to provide continuous device sharing. Though partial reconfiguration by itself has to deal with challenges regard to identifying an appropriate region on the FPGA which has sufficient amount of contiguous unutilized hardware resources and poses minimum possible blockage to upcoming tasks for the placement of configuration data, enabling run-time reconfiguration for increased flexibility introduces overhead in execution run-time and in FPGA fragmentation. Thereby, just-in-time compilation aims at faster partial configuration by not only altering FPGA functionality on the fly but also its specialty is to perform task placement and routing at run-time which lowers hardware fragmentation significantly.

Significance of JIT compilation can be best understood by considering allocation algorithm of the existing approaches. Figure 5.15 depicts an allocation of five tasks namely

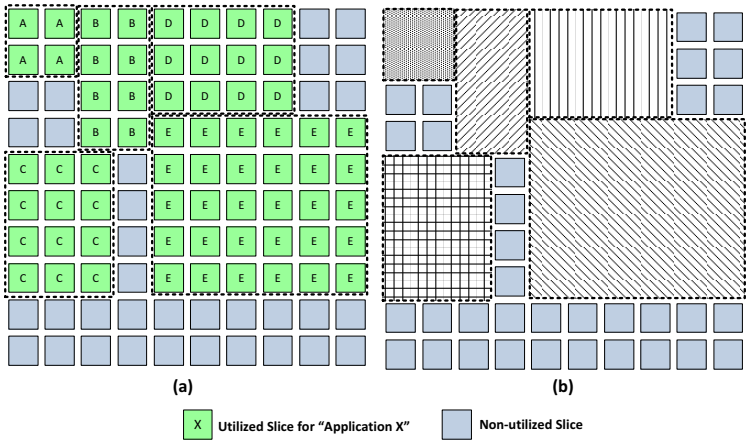


Figure 5.15.: Fragmentation problem introduced by typical task allocation: a) example allocation, b) respective area blockages [92]

A, B, C, D and E onto FPGA with focus on maximizing the resource utilization in vertical or horizontal axis. When a sixth task say E with a size of 4x4 slices needs to be mapped, existing approaches could not handle this even though there are sufficient amount of left over resources as they assume the configuration files to occupy orthogonal or rectangular regions. This approach of handling the tasks as pre-computed macro blocks with predefined area requirements leads to fast application implementation on one hand but on the other hand results in higher fragmentation of hardware resources. With regard to JIT, being performed at slice level with configuration files computed at run-time by technology packing, placement and routing solves the aforementioned resource fragmentation problem at the cost of slow application implementation when computed with existing CAD tools, that were intended of off-line application mapping at design time. Therefore, the goal is to significantly speed-up application mapping to be executed during runtime of the system. Experimental results in [91] confirmed the placement to be the most time consuming step in VPR. This led to the decision to develop and include a much faster placer in the JIT framework.

5.5.1. Target Architecture

A System-on-Chip architecture consisting of multiple *V-FPGA* cores, a microprocessor, a configuration controller, an external memory and AMBA busses for on-chip communication and configuration has been selected as the target platform with each *V-FPGA* core being an array of 100x100 slices. One of the salient features of Virtual FPGAs is their competence even in devices where partial reconfiguration is not supported natively. Thus the JIT methodology can be exercised also on existing COTS FPGAs through the virtual layer. Second importantly, *V-FPGA* provides flexibility at finest granularity level while the state-of-the-art approaches perform reconfiguration at column level.

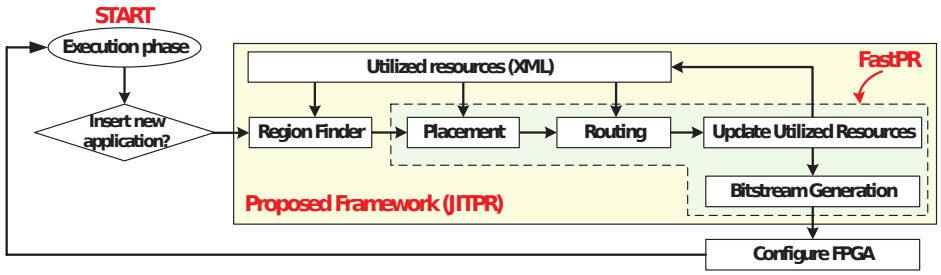


Figure 5.16.: JIT compilation framework for performing fast application implementation onto FPGA devices [92]

5.5.2. JIT Compilation Flow

The corresponding netlist of a new task, which needs to be mapped onto the target architecture, goes through 'Region Finder' before entering into JIT compilation framework consisting of Technology mapping, Placement and Routing as shown in Figure 5.16. The task of the Region Finder to derive the suitable floorplan for the application is a critical step as it preserves minimum fragmentation ratio and maximizes performance. In order to distinguish between the available hardware resources and the ones which are already utilized by previous tasks, an XML file furnished with information about resource types (e.g., logic, routing etc.,) and spatial location of the utilized resources is inputted to JIT framework through an external storage and thereby enabling JIT to perform P&R only with non-utilized resources. This fine-grain reconfiguration supported not only by JIT compilation but also by *V-FPGA* overruns column-based reconfiguration by performing resource isolation at slice and routing track level. The outcome of JIT compilation provides information to compute the bitstream file for this new task and furnishes details about resources in the target architecture upon which this new task will be allocated. With this information the bitstream tool then generates the corresponding bitstream file of required granularity and the XML file is annotated to represent the current state of available resources before it is stored in external storage. The generated partial reconfiguration bitstream file configures the Virtual FPGA with the new task. In case of task de-allocation, the XML file is updated to mark the resources occupied by this task as non-utilized and these hardware resources are then configured with an empty bitstream file.

5.5.3. Run-time Enhancements

The JIT framework is supported by 2-D MEANDER [96]. Appropriate tuning of this toolset significantly reduces the run-time overhead caused by technology mapping and P&R without degrading the results in terms of operating frequency and power consumption. The first tool being the RegionFinder operates with the goal (i) to improve the available resource utilization and (ii) to reduce the execution runtime of the application's P&R. Thereby, it incorporates an advanced heuristic methodology to quantify multiple candidate regions simultaneously rather than identifying regions of rectangular or square

shape over the target architecture. The RegionFinder takes in the application's netlist and two XML files describing the target architecture and the utilized resources as inputs. Initially it assigns a number of uniformly distributed seeds which then expands in x and y directions in the ratio of 1:1, 1:2 and 2:1. Majority of the run-time enhancement is imposed by the employment of a new fast placer which follows fast simulated annealing approach similar to the one in the VPR tool. The placer starts swapping pairs of logic blocks of an initial placement with the goal to find lower overall cost. In simulated annealing, one of the critical parameters that affects the quality of the derived placement is the number of moves per temperature value. While all the available placers enforce $10 \times N^{4/3}$ swaps per temperature value, JIT framework performs $10 \times N_{blocks}$ swaps of logic blocks, without performance degradation and with mentionable lower run-time overhead.

The functionality of the router, which is based on PathFinder negotiated congestion algorithm [68], is to identify the routing segments and switches to connect all the nets in the circuit. The run-time of the router is improved at architecture level by employing Virtual FPGAs which provide wider routing tracks in the most congested regions (e.g., in the center of the architecture). The *V-FPGA* supports this heterogeneity through the *CoreFusion* technique [35] (see Section 4.3.4 for more details), defining and merging center and surrounding cores with different channel width *W*.

The JIT bitstream generator is adopted from *V-FPGA Explorer*. To improve the runtime and reduce memory requirements, the graphical representation is omitted and the object oriented structure is replaced by aggregated composite data types. Commodity functions, manual editing, path exploration and logs are disabled in the JIT version. Furthermore, the source code is ported from VisualBasic to native C that compiles on various platforms with or without operating system, including embedded systems. For parsing the textual output files of the JIT place & route tools, the JIT bitstream generator contains custom string functions.

5.5.4. Experimental Results

Finally, the efficiency of JIT framework is tested on a target Virtual FPGA platform, which consists of 100x100 slices with 50 routing tracks in each channel, for the 20 biggest MCNC benchmarks. Table 5.1 shows the execution time of JIT PR for mapping these benchmarks onto the *V-FPGA*. Compared to the original VPR the proposed JIT PR is at peak 80.75x and in average 53.49x faster (for further details and a break-down of execution time refer to [92]).

In Table 5.2 the runtime for JIT bitstream generation of the majority of these applications is listed, which is in the range of fractions of a second on an Intel i5 processor in single threaded mode. Thereby, most time is consumed in parsing the textual output files from JIT PR. In a productive system, the time could be considerably reduced by providing the JIT PR results in structured binary formats.

The overall time for application mapping and bitstream generation is in a range that makes it feasible to map applications on demand. In conjunction with the slice-level reconfigurability of the *V-FPGA* fragmentation effects are kept to a minimum. It is thinkable to provide such JIT application mapping by services operating in the cloud. Yet an

Table 5.1.: Execution time of JIT PR compared to original VPR [92]

Benchmark	Execution time (*ms*)		Speedup
	Original VPR	Proposed JIT	
alu4	54647	1041	52.49×
apex2	70287	1190	59.06×
apex4	49987	619	80.75×
bigkey	78898	1845	42.76×
des	80134	1432	54.96×
diffeq	62303	1079	57.74×
dsip	61171	1197	51.10×
elliptic	148379	3828	38.76×
ex1010	174274	3517	49.55×
ex5p	47559	695	68.43×
frisc	235350	5895	39.92×
misex3	52441	756	69.37×
pdc	226810	5122	44.28×
s298	77759	1515	51.33×
s38417	337944	7378	45.80×
s38584	273533	7371	37.11×
seq	66485	1097	60.61×
spla	141456	2861	49.44×
tseng	50928	824	61.81×
Average:	**120544**	**2592**	**53.49**×

Table 5.2.: Execution time of JIT bitstream generator for *V-FPGA*

Benchmark	CLB cols	CLB rows	Channel width	Execution time (*ms*) on Intel Core i5 (@2.5GHz)
alu4	32	32	9	160
apex2	37	37	11	230
apex4	31	31	12	160
bigkey	54	54	7	360
des	63	63	7	420
ex1010	40	40	11	270
frisc	59	59	14	770
misex3	31	31	10	150
pdc	52	52	16	607
s38417	60	60	8	686
seq	37	37	13	234
spla	51	51	16	593

autarkic in-system self-adaptive application mapping is possible as well, provided the embedded system is equipped with a processor sub-system and enough memory.

5.6. Area and Delay Models for Optimized Application Mapping and DSE

Typically, the overall area requirement and performance of an application mapped onto a virtual FPGA is revealed after the synthesis, place and route steps of both, the application and the virtual FPGA. Since the place and route steps are area and timing driven, area and delay models are required in order to find an optimized solution. In the past, place & route tools involved in application mapping unto virtual FPGAs have been mainly used with area and delay models that were intended for physical FPGAs. For instance, the initial *V-FPGA* (presented in [50]) used the timing and area related physical parameters of the architecture file templates contained in MEANDER toolflow [69], which are based on a 180 nm technology. Similarly, the technology parameters of ZUMA architecture in [114] are very similar to the ones used in the 90 nm k4_n4_90nm.xml architecture template contained in the VTR toolflow package [63]. While the application mapping will be still valid and the circuits operable, this practice might be deceitful for the purpose of design space exploration and optimization as the ratios of e.g. logic area to routing area or local routing to global routing will suffer accuracy. This might lead to non-optimal parameter choices and reduced mapping efficiency.

To overcome this situation we derive area and delay models for the *V-FPGA*, based on the utilized resource types of the underlying platform. The idea is to decompose the *V-FPGA* into basic elements of minimum size, to characterize these elements and in a bottom up approach to derive area and delay models of the architecture in a hierarchical way that are also dependent on the parameters K, N, W, I, and O.
The programmable resources of *V-FPGA* are BLEs, CLBs (including connection boxes), PSMs and IOBs. A BLE is composed of 2^K:1 MUX (for the LUT), 2:1 MUX (for the bypass) and flip-flops (2^K for the configuration unit and one that can be bypassed at the LUT output). The remaining CLB circuitry requires $(N + \lceil I/K \rceil)$:1 MUXs for the multiplexers at the inputs of the BLEs, optionally N:1 MUXs at the outputs and D-FFs for the configuration unit. Additionally, the connection boxes require 2:1 MUXs for CBw and W:1 MUXs for CBr. Each PSM is composed of 4:1 MUXs for routing and D-FFs for configuration. An IOB needs a W:1 MUX and an AND gate for the output, 2:1 MUXs for the input and D-FFs for the configuration unit.
Hence the Minimum Size Basic Elements (MSBEs) are 2:1 MUX, 2-input AND gate and D-FF. All other elements are composed of these MSBEs. If we consider for instance Actel ProASIC3 as underlying platform, then the MSBEs are realized by the so called *VersaTiles*. A *VersaTile* can realize either a 3-input function or a flip-flop (see [2] for more details). Consequently, the MSBEs have the following area sizes:

$$A_{MUX2} = 1\,VersaTile \tag{5.1}$$

$$A_{AND2} = 1\,VersaTile \tag{5.2}$$

$$A_{FF} = 1\,VersaTile \tag{5.3}$$

With the respective areas of the MSBEs A_{MUX2}, A_{AND2} and A_{FF}, the areas of the other elements based on their composition described above are derived as follows:

$$A_{BLE} = \left(2^K + 1\right) \cdot A_{FF} + \left(\left(2^K - 1\right) + 1\right) \cdot A_{MUX2} \tag{5.4}$$

$$A_{BLE\_inMUX} = \left(N + \left\lceil \frac{I}{K} \right\rceil - 1\right) \cdot A_{MUX2} + \left\lceil log_2\left(N + \left\lceil \frac{I}{K} \right\rceil\right)\right\rceil \cdot A_{FF} \tag{5.5}$$

$$A_{BLE\_outMUX} = (N-1) \cdot A_{MUX2} + \lceil log_2(N) \rceil \cdot A_{FF} \tag{5.6}$$

$$A_{CLB} = N \cdot K \cdot A_{BLE\_inMux} + N \cdot A_{BLE} + O \cdot A_{BLE\_outMUX} \tag{5.7}$$

$$A_{CBr} = (W-1) \cdot A_{MUX2} + \lceil log_2(W) \rceil \cdot A_{FF} \tag{5.8}$$

$$A_{CBw} = W \cdot (A_{MUX2} + A_{FF}) \tag{5.9}$$

$$A_{PSM} = 4 \cdot W \cdot (3 \cdot A_{MUX2} + 2 \cdot A_{FF}) \tag{5.10}$$

$$A_{IOB} = (2W-1) \cdot A_{MUX2} + A_{AND2} + (W + \lceil log_2(W) \rceil + 1) \cdot A_{FF} \tag{5.11}$$

The delays are obtained through characterizations of the MSBEs in a placed and routed design with the help of a timing analyzing tool, which is usually part of the IDE for the underlying platform (e.g. Actel *SmartTime*). Additionally to the MSBEs we need also the average delay of a short net. For demonstration purpose the following delays are obtained on an Actel ProASIC3 FPGA:

$$T_{MUX2} = 0.497\,ns \tag{5.12}$$

$$T_{AND2} = 0.497\,ns \tag{5.13}$$

$$T_{FF\_setup} = 0.430\,ns \tag{5.14}$$

$$T_{FF\_clock\_to\_Q} = 0.550\,ns \tag{5.15}$$

$$T_{net} = 0.249\,ns \tag{5.16}$$

With the respective MSBE delays T_{MUX2}, T_{AND2}, T_{FF_setup}, $T_{FF\_clock\_to\_Q}$ and T_{net}, the relevant delays of the other elements are estimated as follows:

$$T_{MUX4} = 2 \cdot T_{MUX2} + T_{net} \tag{5.17}$$

$$T_{LUT} = T_{net} + K \cdot (T_{MUX2} + T_{net}) \tag{5.18}$$

$$T_{BLE\_inMUX} = \left\lceil log_2\left(N + \left\lceil \frac{I}{K} \right\rceil\right)\right\rceil \cdot (T_{MUX2} + T_{net}) \tag{5.19}$$

$$T_{BLE\_outMUX} = \lceil log_2(O) \rceil \cdot (T_{MUX2} + T_{net}) \tag{5.20}$$

$$T_{IOB\_in} = T_{MUX2} + T_{net} \tag{5.21}$$

$$T_{IOB\_out} = (\lceil log_2(W) \rceil - 1) \cdot (T_{MUX2} + T_{net}) + T_{AND2} + T_{net} \tag{5.22}$$

The resulting values are sufficient for the place & route tool to estimate the path delays and are also incorporated to the architecture file as virtual technology parameters. These models target a fine grained underlying platform (e.g. the 3-input VersaTiles in Actel ProASIC3) and need to be slightly modified when the underlying platform changes. For

instance, for an underlying platform with 6-input LUTs, a 4:1 MUX will have the same area and timing as a 2:1 MUX (both can be realized by 1 LUT). In that case the 4:1 MUX becomes an MSBE.

5.7. Conclusion

A challenge for the custom *V-FPGA* architecture was to find a suitable CAD toolset that is able to map applications onto it, especially considering the various architectural parameters. This was solved in the *V-FPGA* framework by employing and combining a selection of existing academic, commercial and new own tools, that are or have been made compatible to each other. The resulting tool-flow is rather complete (i.e. it supports all the required steps), allowing various state-of-the-art design entry methodologies and including powerful synthesis engines, logic optimizers, technology mappers, placers, routers, a custom bitstream generator and a simulator.

A central piece thereby is the newly developed *V-FPGA Explorer* tool, which is a graphical configuration editor, a bitstream generator, an architecture file generator, a script generator and a testbench generator. Especially it is to note that it closes the gap between abstract layout and actual configuration.

In collaboration with researchers from the National Technical University of Athens (NTUA), a unique just-in-time compilation toolset for the *V-FPGA* was developed. It includes JIT placer&router (developed by NTUA) and JIT bitstream generator (own development). Compared to state-of-the-art tools, we were able to achieve in average a 53.49x faster execution time of these tools. With the JIT compilation flow, MCNC20 benchmark applications are each mapped (including placement, routing and bitstream generation) onto the *V-FPGA* within a few seconds (in average 2.6 s). This makes it feasible to map applications on demand, which is very attractive for adaptive systems and cloud services.

In order to deal with the peculiarities of virtual FPGAs, suitable area and delay models based on characterized Minimum Size Basic Elements (MSBEs) were developed. These models are used in design space exploration as well as to generate architecture files for the employed place&route tools.

6. Concepts and Methodologies for Customizing Reconfigurable Architectures

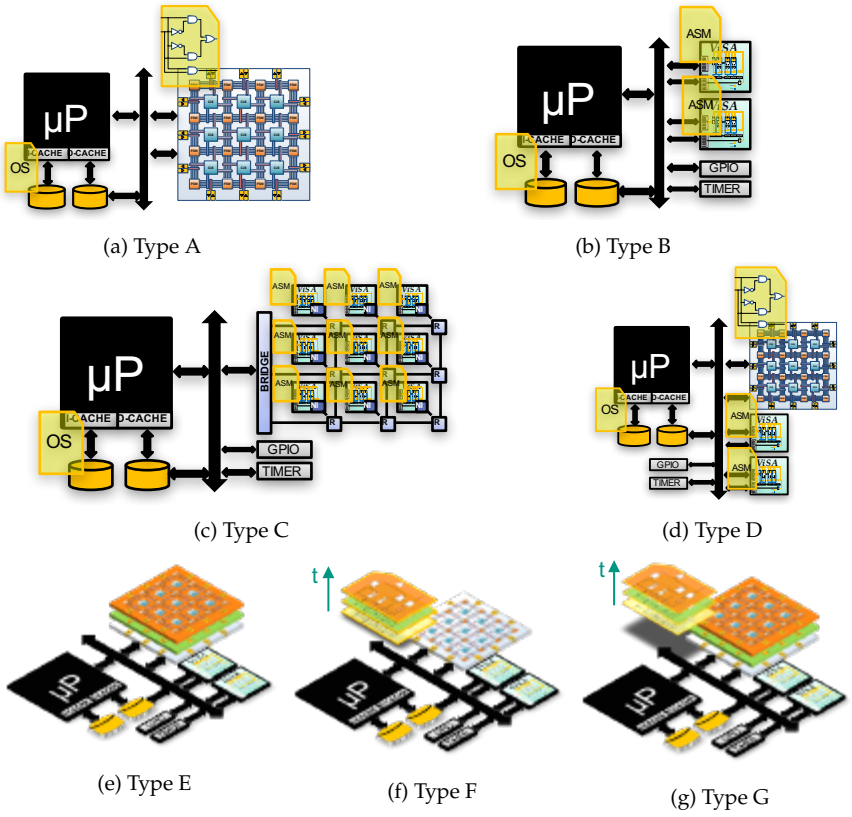
This chapter concerns the customizing of the generic *V-FPGA* architecture to obtain a good fit of the application. On system-level this is achieved by selection of suitable SoC architecture templates. On core-level the architectural parameters are tuned based on design space exploration. Furthermore, suitable metrics are introduced to consider the virtualization aspects in early estimations.

6.1. Generic SoC Architecture Templates

To facilitate the development of custom reconfigurable SoC platforms, the *V-FPGA* framework incorporates customizable SoC templates. This elevates the spirit of customization from core level to system level architecture and enables multi-level optimization. On the system level, the concept is to allow various combinations of heterogeneous cores/peripherals interacting through a unified infrastructure. Figure 6.1 illustrates a set of SoC template concepts, whereby all types have in common a microprocessor core, memories and further heterogeneous cores interconnected through an AMBA bus.

Type A integrates the *V-FPGA* as a peripheral block. Here the *V-FPGA* is equipped with a peripheral unit (see Figure 6.2), that contains a communication controller to map parts of the I/O Blocks to the bus interface. This allows the processor core to exchange data with the FPGA in an addressable way. For the processor it makes no difference whether it writes to a memory or to the *V-FPGA* core. Furthermore, the peripheral unit contains timer and serializer in order these frequently used function units not to occupy the more precious reconfigurable logic. The *V-FPGA* core contains a second bus interface for its configuration controller, which can be accessed by the processor core (or another bus master) to copy configuration data, select configurations, trigger the configuration process and read the status. Typically the microprocessor would run an operating system and more control oriented tasks, while the *V-FPGA* core is a companion accelerating timing critical and parallelizable workloads or acting as flexible glue logic. This template is used in [50] and [35] to create a custom low-power heterogeneous reconfigurable SoC for industrial process automation.

In type B the *V-FPGA* core is replaced by one or more custom *ViSA* cores. The *ViSA* cores are equipped with dual ported memories for instructions and data, whereby one port is mapped to the AMBA bus and the other port is accessed by the load/store unit and the forwarding network inside the *ViSA* core. A *ViSA* core can have multiple data memories, each with a separate load/store unit thus simultaneously accessible by the *ViSA* core for Single Instruction Multiple Data (SIMD) operations. The microprocessor core can access

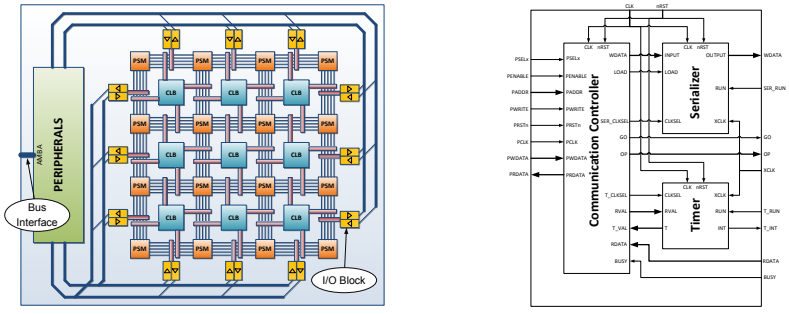


(a) Type A (b) Type B

(c) Type C (d) Type D

(e) Type E (f) Type F (g) Type G

Figure 6.1.: Generic System-on-Chip templates



(a) (b)

Figure 6.2.: Peripheral unit to connect the *V-FPGA* to an AMBA bus: a) mapping to IOBs of the *V-FPGA*, b) structure with additional frequently used function units

the memories through the AMBA bus, to exchange data with the *ViSA* cores or to upload the programs for the *ViSA* cores. The type B template is suitable for accelerating arithmetic dominated workloads of little to medium parallelism and with control flow content (e.g. numeric solver algorithms, multi-cycle operations, complex operations, etc.). In [36] we applied this template to accelerate a measuring chain for industrial process automation.

The acceleration of massive parallel arithmetic workloads (e.g. SAFT, image processing, neuronal networks, etc.) is addressed by the type C template. Here the parallelism is scaled at two levels: local parallelism within a *ViSA* core and global parallelism by an array of many *ViSA* cores. The bus structure of the other types has limitations in terms of scalability and concurrent traffic which makes it not suitable for many-core architectures. Therefore, the *ViSA* many-core array of the type C template utilizes a scalable mesh type Network on Chip (NoC) communication infrastructure. Each *ViSA* core is equipped with a NoC interface and thus is a node in this network. A bridge allows the microprocessor to access the NoC through the AMBA bus. In [36] we demonstrated the efficient acceleration of a highly parallel SAFT algorithm for 3D ultrasound tomography, utilizing a 176 many-core *ViSA* array.

Type D is a combination of types A and B, where *V-FPGA* and *ViSA* cores coexist along with a processor core and dedicated peripherals in a heterogeneous SoC. This targets multi-tasking workloads, where tasks have diverse characteristics and are accelerated each on the most suitable core in the system.

Types E is similar as type D but extends the *V-FPGA* core to the 3rd dimension by stacking multiple 2D *V-FPGA* layers and interconnecting the layers with TSVs (see Section 4.2). In type F the 3rd dimension is time, i.e. temporal exclusive applications are mapped on an overutilized *V-FPGA* core and are loaded on-demand through dynamic reconfiguration. Type G is a combination of the types E and F, exploiting 4 dimensions in the *V-FPGA* core: 2 dimensions for the horizontal pane, 1 dimension for the vertical stacking and time is the 4th dimension.

The chart in Figure 6.3 classifies the suitability of the different SoC template types according to the degree of parallelism and the content of control flow in the portions of the application that should be accelerated. Type A offers a wide range of parallelism, yet is not well suited for control flow dominated tasks, because control flow doesn't benefit much from parallelism yet results in large multiplex logic when implemented as finite state machine logic in an FPGA. Control flow is more efficiently handled in Type B where it can be realized by sequential programs with conditional jumps running on the *ViSA* microarchitecture, i.e. the control complexity is mapped in cheap memory rather than in expensive reconfigurable logic. However, the parallelism in type B is limited by the number of parallel function units and the length of the instruction word. Experiments in context of the use cases in [36] have shown that with longer instruction word the achievable clock frequency is reduced because instruction memory needs then to be partitioned into several blocks that are cascaded, thus the skew and path delays for memory access increase. Therefore, type C is better suited for applications with control flow content and a high degree of parallelism, as it scales better than type B by employing multiple *ViSA* cores in parallel, each with reasonable instruction word length and local parallelism. Types D-G, which are combinations of the other types, consequently are classified somewhere in the centre region, while tendencies can be tuned by the ratio of *ViSA* cores to *V-FPGA* cores.

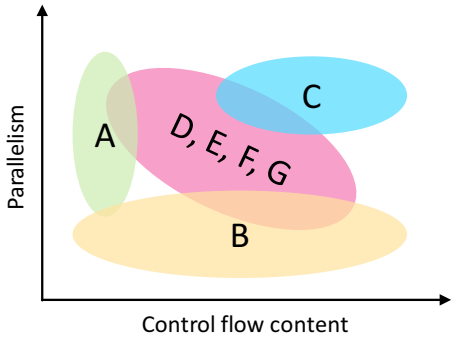


Figure 6.3.: Classification of type A-G SoC templates based on parallelism and control flow suitability

For demonstration purposes Figure 6.4 shows a template of type A, realized with the Actel Libero IDE and the *Canvas* utility. The SoC consists of an ARM Cortex M1 processor core, a *V-FPGA* core, an interrupt controller, six GPIO cores, two timer cores, a PWM core, an SRAM memory block and a memory controller for external memory. The bus system is cascaded, whereby the processor core is master of an AMBA AHB bus with the SRAM and the memory controller as slaves. An AHB-to-APB bridge is the third slave component in the AHB bus and acts as master of an Advanced Peripheral Bus (APB) bus. The remaining cores are connected as slaves to this bus. There are three unoccupied slave ports in the APB bus that can be used to extend the SoC by additional cores. Each slave port has an associated address range to be accessible by the processor core. Cores can be added and removed according to the needs. Once all system components are assembled, a VHDL top level file of the SoC can be generated, containing all the necessary instantiations and signals. If needed, the *V-FPGA* core can be further customized by tuning its architectural parameters. Then the design can be synthesized, placed and routed. Thus, in response to *Challenge 1* formulated in Section 1.4, a custom SoC architecture with custom embedded FPGA and heterogeneous cores can be obtained with minimum efforts. In fact it is not even necessary to write VHDL code.

The fundamental difference between COTS programmable SoC platforms ([110], [52], [73]) and the proposed customizable templates is that in COTS solutions the SoC platform is given, whereby in the proposed solution it is multi-level customizable for the purpose of specialization and tailoring towards the application.

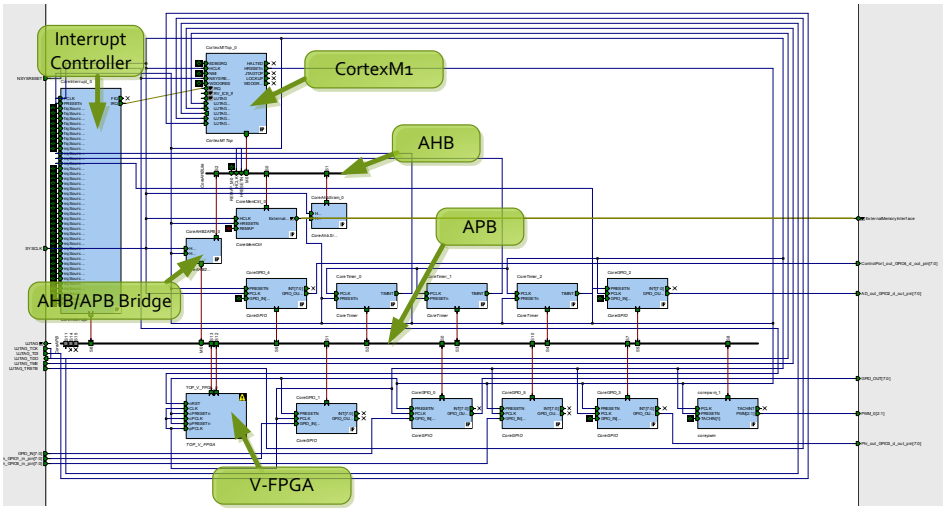


Figure 6.4.: Screenshot of graphical type A SoC template, created with Actel Libero IDE

6.2. Application Specific and Objective Driven Specialization

With the overall goal to obtain specialized reconfigurable platforms that are tailored to fit the application efficiently, the proposed design flow as a whole follows a multi-level customization methodology for system level and core-level architecture, as shown in Figure 6.5. In contrast to the state of the art, the application does not bend to a given off-the-shelf platform, but comes first and defines the characteristics of the custom target platform.

Despite this thesis is dedicated to custom FPGA architectures, the strategy is not to map the entire application onto programmable logic alone but rather to partition the application and take advantage of the diverse core types and their strengths in a heterogeneous SoC. Then the cores can be customized and their architectural parameters tuned to suit their portion of the application as good as possible.

Starting with the application as a model, algorithm, code, pseudocode or any other detailed form to describe its intent, an analysis (1) should reveal the critical portions. If executable software code is already there, profiling methodologies on target processor models can be engaged for this purpose. Usually it will be a performance profiling, yet in [41] and [100] we extended virtual platform processor simulation models by micro-benchmarked power models per instruction to obtain also power profiling during program execution.

Based on the analysis/profiling the application can be partitioned (2) in critical and uncritical parts. The uncritical part is to be mapped as software implementation (3) on a microprocessor. Typically this is the operating system and less demanding control ori-

ented tasks, that are well suited by a processor core. Demanding tasks and bottlenecks that wouldn't meet the constraints in software are the critical portion and need to be accelerated in hardware. While hardware/software partitioning is an active field of research, it is not the scope of this thesis to further immerse in this topic. It is assumed that the partitioning problem is solved based on respective partitioning algorithms, classification of workloads, rational decisions or Artificial Intelligence (Ai).

The critical tasks are further partitioned 4 in parts that are suited on *V-FPGA* architectures and parts that are better suited on *ViSA* cores. The classification in Figure 6.3 can guide this step based on characteristic workloads. Then the core level customizations take place, which are described in Section 6.2.1 for the *V-FPGA* approach and in Section 6.2.2 for the *ViSA* approach.

The primary outcome of the *V-FPGA* customization is a set of architectural parameters tuned to efficiently implement the intended circuits. Similarily, the *ViSA* customization defines the numbers and types of parallel functional units as well as memory blocks and load/store units per *ViSA* core. With respect to the uncritical path on the microprocessor, after software implementation and compilation, the memory footprint for the processor core is obtained, e.g. from the listing file or the Executable and Linkable Format (ELF) file. All these parameters together are used during instantiation of the cores at system level to yield the custom programmable SoC platform that is tailored towards specific applications.

6.2.1. *V-FPGA* Customization

The customization of the *V-FPGA* starts with the application or portions of it that are to be accelerated on the *V-FPGA*. With respect to nomenclature, for a better distinction from the overall application, in the following these portions are referred to as *circuits*. There can be several circuits mapped concurrently or temporally exclusive onto a *V-FPGA* core, which is related to partial and dynamic reconfiguration.

First the design entry of the circuits takes place, which is usually VHDL coding 5 yet other methods such as graphical modelling, drawing schematics, IP based design, high level synthesis, etc. are supported as well by the tools engaged in the *V-FPGA* framework (see Section 5.1).
Behavioural simulation (pre-synthesis) is used to verify the circuits 6 and if necessary the design is refined.

Prior to synthesis, the circuits are combined for later analysis steps. I.e. concurrent circuits are aggregated 7 into one top-level VHDL module, while temporally exclusive circuits should stay in separate files to consider dynamic reconfiguration. Then the circuits are synthesized 8 to obtain technology mapped BLIF netlists. This is repeated for a range of LUT sizes *K*.

To obtain the architectural parameters *K, N, W, SBtype, etc.* that fit a circuit or aggregation best, a Design Space Exploration (DSE) 9 based on the methodology presented in Section 6.4 is performed. This is repeated for all aggregations. Then, depending on the objectives, the aggregation with the most critical timing slack or with the largest area is taken as reference and the *K, N, W, SBtype, etc.* parameters are selected for the whole core.

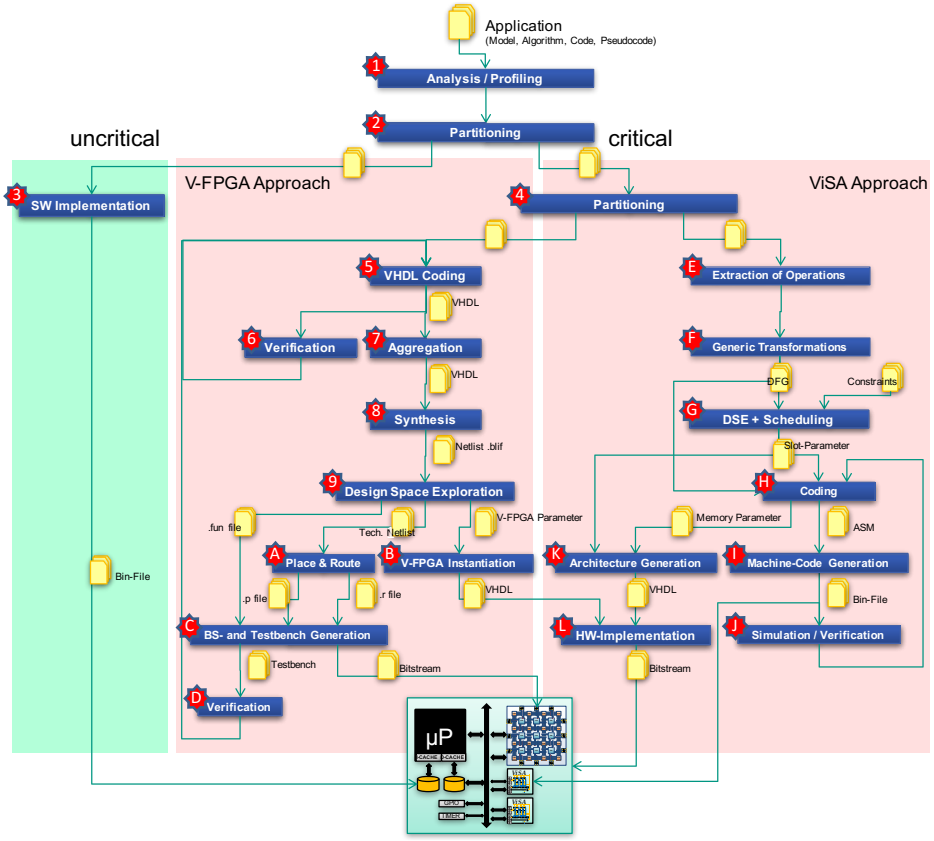


Figure 6.5.: Customization flow for heterogeneous SoC architectures based on the *V-FPGA* framework

To make the *V-FPGA* core more robust for future circuit changes and avoid congestions W can be increased by a certain margin, e.g. 10%.

Then again all aggregations are placed and routed with the fixed reference parameters and the sizing of the array (X columns and Y rows of CLBs) is determined for each aggregation. The size of the largest aggregation plus a safety margin is taken as reference and a final place & route A is performed for all aggregations with the fixed parameters, followed by bitstream and testbench generation C.

The generated testbench is used to simulate D both, the *V-FPGA* core and the circuits mapped on to it. The results of the verification can be taken as feedback loop to refine the circuits if necessary.

The actual realization of the *V-FPGA* core happens by instantiation B of the generic core, overriding of the generic parameters at top-level with the fixed parameters determined during DSE, and physical (or virtual) hardware implementation L as described in Chapter 7. Thereby an integration with other cores takes place to form the custom SoC.

6.2.2. *ViSA* Customization

For customizing the *ViSA* cores a hardware/software co-design methodology is employed that leads to a highly application-specific shape of *ViSA*, which is optimized towards meeting the application needs in terms of functionality and performance with minimal overhead [36].

The first step is to analyze the algorithms of the application and to extract E the set of arithmetical and logical operations used, e.g., multiplication, addition, division, square root, comparison, logic shift, AND, OR, XOR. This will determine the types of function units to be deployed in the application-specific *ViSA* architecture. Thereby, common function units are already available in the work library. Special units, e.g. hardware accelerated square root unit, need either to be developed from scratch or reused from existing designs (e.g. vendor IP or open source IP catalogs). The library keeps growing and the degree of reuse increases with every new design.

After specifying the necessary function units, a generic transformation F takes place to obtain the Data Flow Graph (DFG) from the algorithms and to perform a design space exploration with scheduling G in order to efficiently exploit the degree of parallelism that *ViSA* provides.

Therefore, the data flow exploration of the application solves the dependencies of the data and the operations. Figure 6.6 shows an example DFG for the realization of a slot architecture with all possible executable operations. The graph will show the minimal used operations in case the application is fully partitioned into the basic functional units introduced above. This provides in turn the requirements for the data to be available from memory.
From the pre-built library, introduced above, the area utilization is known for every component and thus the total area for the application-specific *ViSA* can be estimated beforehand. The obtained values for total area are worst-case estimations. Experiments have shown that due to global optimization in the synthesis process of the system, the actual

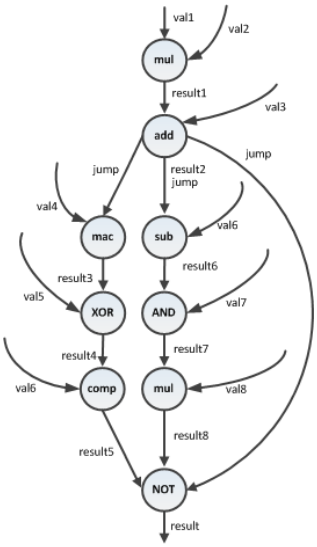


Figure 6.6.: Dataflow graph of an example application scheduled for *ViSA* with all executable operations

area will be in average around 5 % smaller and never larger than estimated. Furthermore, from simulations and/or data sheets the performance in clock cycles is known for every unit in the library. This helps to schedule the operations within the DFGs with the goal to meet given constraints while reducing the number of parallel units that are not utilized. It turned out that the as late as possible (ALAP) scheduling strategy minimizes the number of necessary load/store operations as an intermediate value is calculated by a function unit just-in-time before it is needed by another unit. Although the overall performance can be estimated based on the scheduled DFGs, this will be only an ideal best case value, because memory accesses are not modeled in the DFG. However, a more accurate estimation can be done later in the design process after the software has been coded.
The outcomes of the DSE phase are the number of parallel instances for every type of function units used, the data widths of input and output ports, the ports of the input multiplexers, the number of parallel load/store units and memory blocks accordingly.

In the next step the software can be developed in parallel assembly code H, based on the scheduled DFGs and the derived instruction set. For every slot in the *ViSA* architecture there is also a slot in the assembly line. An assembler & linker tool generates the machine code I and the separate instruction and data memory mapping out of the assembly code. Furthermore, it determines the parameters for the actually needed memory widths and depths of the separate instruction and data memories.

The functional verification J of the implemented application running on a *ViSA* core is preferably done in presynthesis simulation. In order to speed up this phase and shorten the time-to-market, a comprehensive debug unit is integrated inside *ViSA* as non-synthesizing VHDL code, which is able to stream out a detailed tracing during simulation.

The parameter values obtained in the previous steps are entered in the generic *ViSA* description (K) and the application specific architecture is synthesized, placed & routed (L) and a bitstream is generated.

6.3. Metrics

Apart from identifying the required resources, the customization approach also aims at optimizing the *V-FPGA* architecture for one or more objectives depending on the application needs. To quantify the effects and compare solutions, suitable metrics are required. However, virtualization obscures the usual metrics as the basis is not fixed. This makes it also difficult to judge the efficiency of the *V-FPGA* because it is not only a relationship of application and architecture alone, but at the same time also a relationship between the architecture and the underlying platform. For instance, assuming that an application is optimally mapped on a customized architecture, i.e. the utilization ratio on the virtual architecture is maximized, it still can be the case that the virtual layer itself is poorly mapped onto the underlying platform. To overcome this situation the traditional metrics (e.g. gate equivalent, critical path delay, physical area etc.) are extended by further platform dependent and platform independent metrics. Using platform dependent metrics helps improving the quality of CAD and the confidence level of the results, while platform independent metrics allow quick-and-dirty cross-platform transferability by generic base units that in a post-process can be translated into platform-dependent metrics. The following subsections present a set of metrics and discuss their meaning and purpose.

6.3.1. Average Utilization Ratios

Ultimately, chip area is a measure with direct impact. However, for architecture exploration and analysis it can be a quite unhandy measure, not only because it takes timely efforts to obtain it accurately but also because it gives no hint about the potential optimization room unless it is compared among all solutions in the design space. Even then it doesn't explain the reason why the parameters of a solution led to less area consumption than other solutions. While this more or less brute-force approach was done for the experiments in Section 4.1.5.2 to obtain a large data set of high significance for fundamental analysis of parameter sensitivity, for the purposes of customisation the aim is to get an earlier estimate whether the architecture gives a good fit to the application and thus is an efficient solution.

Rather than absolute area numbers it is of interest how well resources (that are proportional to area) are utilized. This can be expressed as utilization ratio. Thereby, utilization ratio can have different meanings. In design flows for application mapping onto COTS FPGAs it is common to report for each resource type (LUT, register, multiplier or DSP slices, BRAM, I/Os) a utilization ratio by the means of how many elements of a resource type were utilized compared to the total available resources of that type in the FPGA. Because the area of a COTS FPGA is fixed, the utilization ratios are indicators whether there is still space for additional logic. In certain cases they can also help to decide on design alternatives. For instance, if an FFT application overutilizes the DSP resources while LUTs

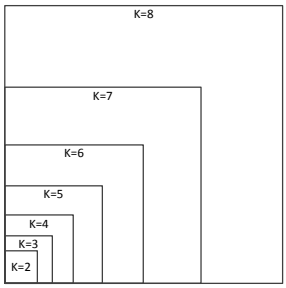


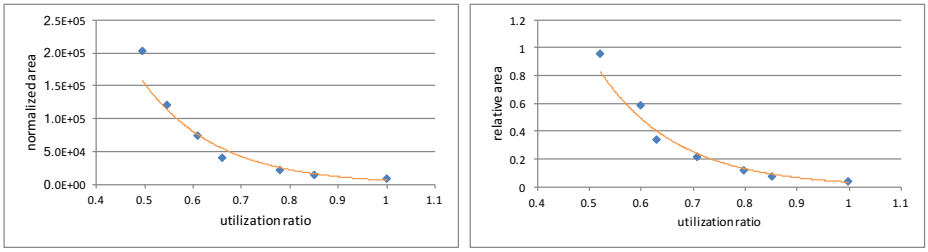
Figure 6.7.: Relative comparison of area required for implementing LUTs of different input sizes K

are hardly utilized, it will make sense to implement a part of the demanding arithmetic in logic. In [37] and [38] we have done such considerations to analyze and balance the resources in reconfigurable digital signal processing hardware architectures for audio and communication applications.

For hardware architects, that need to decide on parameters, it is useful to have more detailed types of utilization ratios. In particular for FPGA architectures, one of the most sensitive parameters is the LUT size. A small function with two inputs (e.g. c <= a AND b) utilizes a 2-input LUT more efficiently than a 6-input LUT. The latter will waste a 18x higher chip area, which becomes clear when comparing the area for different LUT sizes K as illustrated in Figure 6.7. Thus, a logic utilization ratio, that expresses the active portion of the used logic area compared to the total available logic area, is useful to assess the logic related mapping efficiency. The reason is that the more each individual logic cell can be utilized, the less logic cells are required to fit the application and the higher is the logic efficiency.

This claim can be backed up by plotting the normalized total LUT area vs. the utilization ratio for all solutions as it is exemplarily shown in Figure 6.8a for the *alu4* benchmark circuit in particular and in Figure 6.8b for the average over the 20 largest MCNC benchmarks. The experiments confirm that with increasing utilization ratio the required logic area decreases. Note the exponential character which comes from the fact that with linearly rising LUT-inputs the LUT-size grows exponentially (recall Figure 6.7).

In virtual FPGA architectures this gets a bit more complicated because there are two levels of mapping efficiency. It is not only important how well an application is mapped onto the architecture of the virtual FPGA, but also how well the virtual FPGA is mapped onto the underlying platform. Each of these two views can be expressed by an utilization ratio. The Virtual Logic Utilization Ratio (VLUR) and the Platform Logic Utilization Ratio (PLUR) are introduced as follows:



(a) (b)

Figure 6.8.: Relationship between logic area and utilization ratio for (a) *alu4* circuit and (b) for the average over MCNC20 benchmark set

$$VLUR = \frac{\sum_{i=0}^{netlist\_LUTs-1}\left(\frac{2^{c_i}-1}{2^K-1}\cdot w_{comb}\right)+\left(\frac{2^{c_i}}{2^K}\cdot w_{seq}\right)}{total\_LUTs\cdot\left(w_{comb}+w_{seq}\right)} \tag{6.1}$$

$$\text{with } c_i = VFPGA.LUT(i).connected\_inputs \tag{6.2}$$

$$\text{and } total\_LUTs = \begin{cases} netlist\_LUTs & \text{after technology mapping} \\ X\cdot Y\cdot N & \text{after placement} \end{cases} \tag{6.3}$$

$$PLUR = \frac{\sum_{i=0}^{utilized\_LUTs_{platform}-1}\left(\frac{2^{\hat{c}_i}-1}{2^{\hat{K}}-1}\cdot w_{comb}\right)+\left(\frac{2^{\hat{c}_i}}{2^{\hat{K}}}\cdot w_{seq}\right)}{utilized\_LUTs_{platform}\cdot\left(w_{comb}+w_{seq}\right)} \tag{6.4}$$

$$\text{with } \hat{c}_i = Platform.LUT(i).connected\_inputs \tag{6.5}$$

$$\text{and } \hat{K} = \text{LUT size or equivalent of platform} \tag{6.6}$$

6.3.1.1. Virtual Logic Utilization Ratio (VLUR)

VLUR can be measured by averaging the utilized areas of LUTs divided by their total area. The utilized area is related to the number of connected LUT-inputs. Regarding a LUT as composition of 2:1 MUXs, the required number of these MUXs per LUT is $2^{LUT\_inputs}-1$. Of course the number of registers to store the LUT contents needs to be counted as well, which is 2^{LUT_inputs} but needs to be weighted by a factor w to normalize it to the size of a 2:1 MUX. With Actel's VersaTile technology we can set $w = 1$ because a VersaTile cell can implement either a 3-input function (a 2:1 MUX is such a function) or a flip-flop. More generally, as done in Equation 6.1, we can introduce a weighting factor w_{comb} for the combinational share on area and another weighting factor w_{seq} for the sequential share (i.e. flip-flops) if these two resource types have different sizes. VLUR can be obtained either after technology mapping or after placement, with the latter considering also autosizing effects. Figure 6.9 shows the VLUR values for the 20 largest MCNC benchmarks after

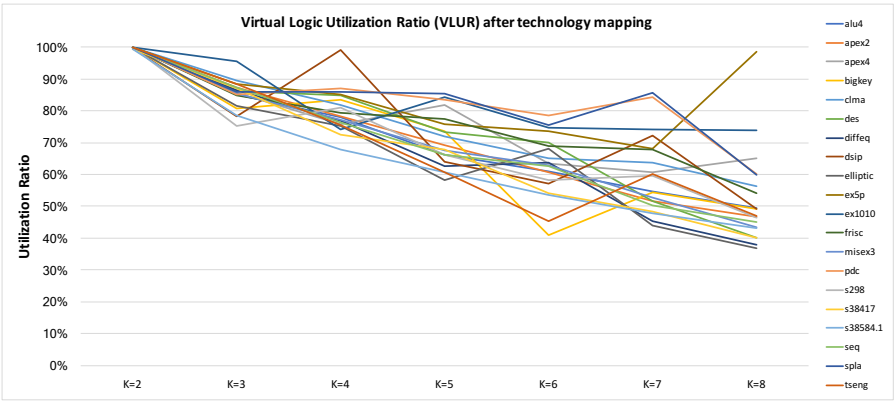


Figure 6.9.: Virtual Logic Utilization Ratio (VLUR) of MCNC20 benchmarks for $K = 2..8$ after technology mapping

technology mapping on different LUT sizes between $K = 2..8$. The general trend therein is that with rising K the VLUR decreases. This makes sense as a logic function with a given number of variables can be decomposed in 2 or more smaller sub-functions, yet the other way will always lead to underutilized LUTs.

In Figure 6.10 we can see the utilization ratio after placement. This differs from the pre-placement results for the following reasons:

- The array on which the nodes of the technology mapped netlist are placed has a rectangular aspect ratio. This can cause whitespaces of fully unutilized LUTs because of quantization effects of the sizing. For instance, the sizing of a LUT array for placing a netlist with 23 nodes leads to 5x5=25 LUTs for a square aspect ratio, i.e. 2 LUTs remain unutilized.
- Similarily, clustering can cause additional whitespace within a CLB cluster.
- Autosizing considers also the I/O blocks which have a dependancy on the array sizing when allowing only a fixed number of I/Os per column (which is the usual case). For applications that are dominantly I/O-bound it can happen that the autosizing tool will artificially increase the array size (which would have been sufficient for the logic alone) in order to increase the perimeter and fit more I/Os. The same can also happen with other resource types (e.g. DSP slices, BRAMs, etc.) in heterogeneous FPGA architectures.

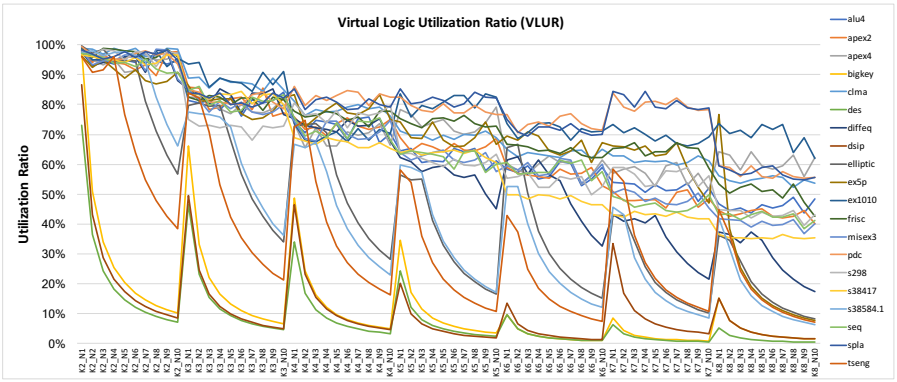


Figure 6.10.: Virtual Logic Utilization Ratio (VLUR) of MCNC20 benchmarks for $K = 2..8$ and $N = 1..10$ after placement on autosized arrays

6.3.1.2. Platform Logic Utilization Ratio (PLUR)

PLUR is a function of utilized LUT-inputs of the underlying platform, whereby the reference design is a BLE of the virtual layer (BLE_v) since we are interested in the logic related utilization ratio. Thereby we need to consider also the registers required to store the LUT function table of the virtual BLE_v, as well as an additional flip-flop for sequential logic and the bypass multiplexer along with an associated programming register for selecting the mode. Fortunately all these elements are part of the BLE_v unit and due to the regular structure of the *V-FPGA* it is sufficient to synthesize one unit alone and map it on the technology of the target platform in order to get all required data for deriving the PLUR. From the technology mapped netlist we gain knowledge about all allocated platform resource macros and their input and output connectivity. This information we can use in Equation 6.4 to calculate the PLUR. Figure 6.11 shows a comparison of PLUR values for LUT sizes of the virtual layer varying from K=2..8 when mapped on an Actel ProASIC3 or a Xilinx Virtex-7 underlying platform. In case of ProASIC3, it is no surprise that the PLUR is constantly at 100%. This comes from the fact that the rather fine-grained VersaTile architecture is fully utilized when accommodating functions with 3 input variables - a perfect size for 2-to-1 MUXs. Since any LUT size can be symmetrically decomposed in 2-to-1 MUXs there is no underutilization of the allocated VersaTile macros. The Virtex-7 platform shows smaller PLUR values depending on the K due to the coarser granularity of LUT resources that leads partially to underutilization.

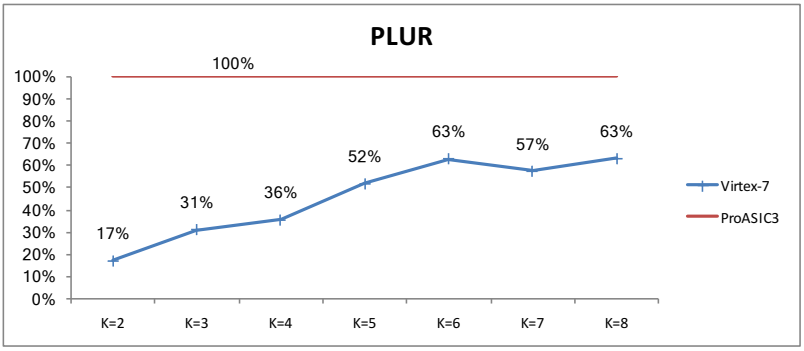


Figure 6.11.: Platform Logic Utilization Ratio (PLUR) on Actel ProASIC3 and Xilinx Virtex-7 for $K = 2..8$

6.3.1.3. Multi-level Utilization Ratio (MLUR)

The product of VLUR and PLUR creates the Multi-level Utilization Ratio (MLUR) and can be used to assess the effective overall logic efficiency:

$$MLUR = VLUR \cdot PLUR \tag{6.7}$$

Figure 6.12 plots the MLUR of the 20 largest MCNC benchmarks when they are mapped onto *V-FPGA* with LUT size K in the range of 2 to 8, whereby the *V-FPGA* itself is mapped onto a Xilinx Virtex-7 underlying platform. In average the sweet-spot concentrates around $K = 6..7$, however individual applications reach their maximum multi-level utilization ratio at other K values. For Actel ProASIC3 the MLUR is equivalent to VLUR (see Figure 6.9) because PLUR is constantly at 100 % on this very fine grained platform.
Thus, MLUR can be used to tune the *V-FPGA* parameters in a way that not only the application is well mapped on the virtual layer but also the *V-FPGA* is well mapped onto the physical platform or to find a trade-off that improves the overall area efficiency. It can be also used to identify the best combination of *V-FPGA* and underlying platform in terms of logic mapping efficiency.

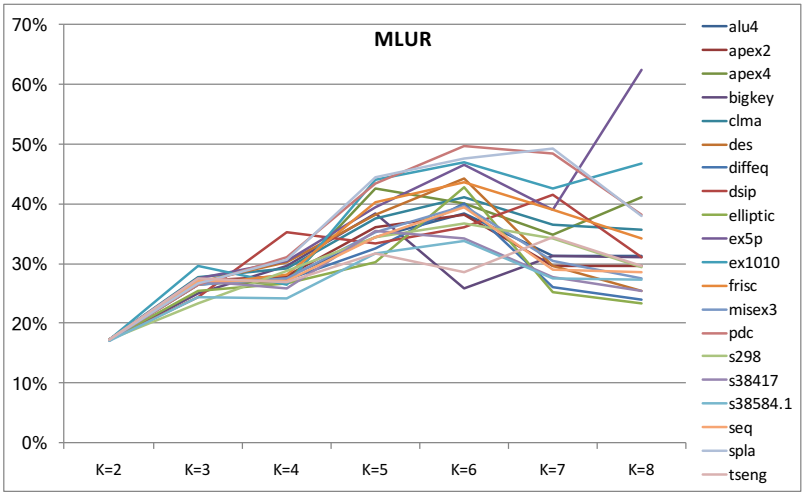


Figure 6.12.: Multi-level Utilization Ratio (MLUR) of MCNC20 benchmarks after technology mapping onto *V-FPGA* for $K = 2..8$, which is mapped onto Xilinx Virtex-7 underlying platform

6.3.2. Generic Equivalents

The utilization ratios introduced above are relative metrics for estimating the mapping efficiency. Absolute metrics are used to assess the overall complexity, area or performance, including also the routing related portion of the *V-FPGA* as opposed to logic utilization ratio. However, they are obtained only after mapping and placement of the *V-FPGA* architecture onto a certain target platform, which can be time consuming and needs to be repeated if the target platform changes. For faster estimation prior to actual implementation a set of generic equivalent metrics is introduced. Expressing the area of a *V-FPGA* in $MUX2$ equivalents, LUT_K equivalents and Flip-flop count is independent from the target platform and doesn't need to be repeated if the platform changes. A metric for platform dependent estimation is given through the introduction of weighted minimum size basic elements, which is a more meaningful metric. In the end the platform dependent metrics really matter. So what is the purpose of platform independent metrics? The answer is that platform independent metrics can be translated to platform dependent metrics through characterization, thereby bypassing the timing intensive implementation steps for the purpose of quick estimations.

6.3.2.1. $MUX2$ Equivalent (M2E)

MUX2 Equivalent (M2E) is a technology independent metric for expressing the complexity of a logic circuit by 2:1 MUXs. It is an important metric in virtual FPGA architectures as MUXs are the dominating types of elements. LUTs and almost the entire routing struc-

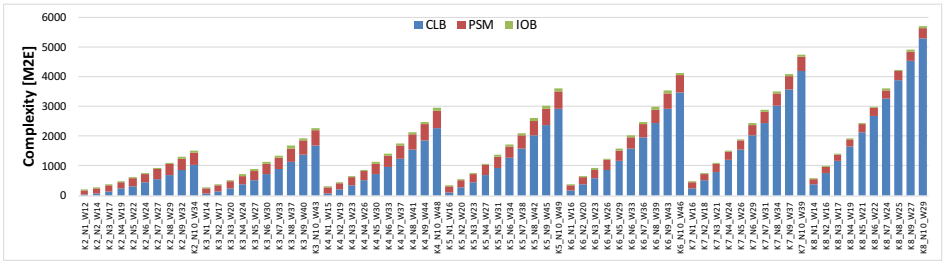


Figure 6.13.: MUX2-equivalent module-level complexity of *V-FPGA* for various K, N and W combinations

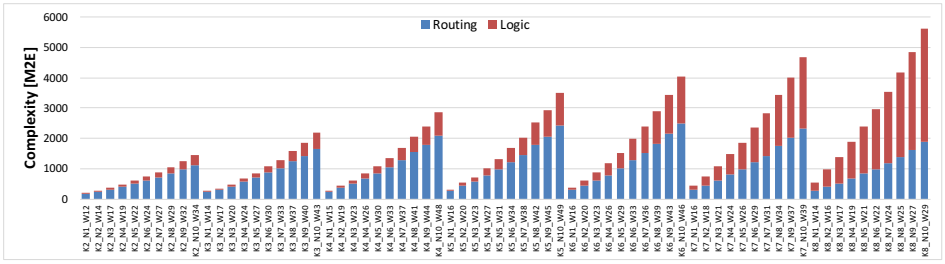


Figure 6.14.: Logic vs. routing complexity of *V-FPGA* expressed in MUX2-equivalents

ture are composed of MUXs. The rather simple decomposition of higher order MUXs into 2:1 MUXs makes M2E a handy baseline unit. Recalling Table 2.1 from the fundamentals Chapter 2 any of the other required logic functions can be realized with 2:1 MUX. Thus, it is a fully legitimate technology-independant area metric. However, it shouldn't be the only area related metric in the system as it doesn't account for sequential storage elements. Therefore it needs to be complemented at least by flip-flop count to cover the entire relevant area of a *V-FPGA*.

Figure 6.13 plots the M2Es against various combinations of K, N and W for CLB, PSM and IOB components of the *V-FPGA* architecture. Thereby W was taken based on the average channel width for the MCNC20 benchmarks. In most cases the complexity of CLBs dominates, however it should be noted that it contains also the local routing. A breakdown in logic and routing complexity is given in Figure 6.14. As expected, at finer granularity of CLBs the routing complexity dominates and vice versa.

6.3.2.2. LUT_K Equivalent (LKE)

To decouple LUT complexity from its realization the technology independent LUT_K Equivalent (LKE) is introduced as an abstract metric. Therein, the subscript K is a variable that represents the number of inputs per LUT. For instance $4LUT_3$ represents the complexity of four LUTs with each 3 inputs. This metric can be used two-fold:

- The complexity of any combinational logic circuit can be expressed in LUT_K. This can be translated in platform dependent measures once the target platform is known.
- Synthesizing an application into a technology independent BLIF netlist yields a structure with variable K-input LUTs as nodes. Counting the LUTs for each K and weighing them with an area factor proportional to LUT area gives a hint about the overall complexity.

It should be noted that the translation into other metrics or a fixed K can cause some loss of accuracy because some parts of a circuit can not be combined into a greater LUT if they are on independent paths.

6.3.2.3. Flip-flop Count (FFC)

Flip-flop Count (FFC) is a mainly technology independent metric for estimating the complexity of synchronous elements in the *V-FPGA* architecture. It is as simple as counting all the flip-flops in the architecture. In virtual FPGAs FFC accounts dominantly for configuration area as the configuration bitstream is loaded in the configuration units that consist of flip-flop chains. Flip-flops are also used in BLEs for realizing sequential circuits, however their count there is much less than in the configuration units.

6.3.2.4. Minimum Size Basic Elements (MSBE)

Minimum Size Basic Element (MSBE) are introduced as technology dependent metrics. The idea is to defines the smallest elements of the architecture out of which the other elements can be composed and to associate the number of underlying resources required to implement it. Area and delay of each MSBE are obtained through characterization. Then the area of each MSBE type is normalized to the area of the smallest MSBE type present in the platform. Elements of higher complexity are estimated through composition and respective summing of MSBEs. MSBE is a multidimensional metric depending on the different resource types (LUTs, FFs, etc.). A special case applies on Actel devices as a VersaTile cell can be either a 3-input LUT or a flip-flop and thus both have the same normalized size 1. Manufacturing-independent metrics such as M2E, LUT_K and FFC can be mapped onto MSBE metrics, meanwhile this tandem presenting an easy way to transfer metrics from one platform to another. The accuracy of MSBE is rather high. Table 6.1 shows a comparison of area estimation through MSBEs vs. actual area that is reported after synthesis and technology mapping of an inner tile of a *V-FPGA*. The target platform is an Actel (now Microsemi) ProASIC3 device. The estimated area is hierarchically divided in sub-modules. The actual area is reported only for the whole inner tile. Between estimated area and actual area the relative error is only around 0.7 %.

Table 6.1.: MSBE based area estimation vs. actual area reported by Microsemi Libero IDE on a ProASIC3 device for an inner tile of *V-FPGA* with the parameters $K = 4$, $N = 7, W = 25$

	est. area per module	**qty**	**actual area (synthesized and mapped)**	**error**
CLB	623	1		
BLE	33	7		
LUT	31	1		
BLE_inMux	14	28		
cbR	29	16		
cbW	50	7		
PSM	500	1		
total	**1937**		**1950**	0.7 %

6.4. Design Space Exploration Methodology

To find the optimum parameters an architecture level design space exploration is performed with combinations of varying cluster size N and LUT size K. Parts of either MEANDER toolflow or of VTR toolset [63] can be used for this purpose and are complemented by custom scripts and architecture file generators in the *V-FPGA* framework. However in Figure 6.15, it is aimed to describe the flow of the involved CAD steps in a more general view independent from the actual tools. Starting with the smallest $K = 2$, technology independent netlists of presynthesized benchmark circuits are translated into netlists of equally sized K-input LUTs. Proceeding this is the process of packing, where N LUTs are clustered into one CLB with an initial value of $N = 1$. The hypergraph nodes of the resulting netlist are placed onto an array of CLBs, whose size is not known at the beginning. One of the optimization goals of this placing step is to determine the required number of CLB columns and rows with minimum area consumption. The placed nodes are then swapped for timing driven optimizations, aiming for minimum distance between connected nodes. The next step is to route the signal paths between the placed nodes by considering the routing capabilities of the architecture (PSMs, connection boxes, in- and output multiplexers). The channel width W is not fixed and is determined iteratively in an area driven mode attempting to route the design with minimum required tracks per channel. To speed up this process, W is estimate beforehand based on the parameters K and N. This is not an accurate estimation since the minimum W depends also on the application. However, this is good enough to start an initial routing attempt, followed by iterative bisection of the estimated W to converge towards the actual minimum channel width with a reduced number of routing attempts. Once the minimum channel width is found, the usual timing driven optimizations follow.
The steps of packing, placement and routing require information about the target virtual FPGA architecture and the parameters and constants related to area and delay models.

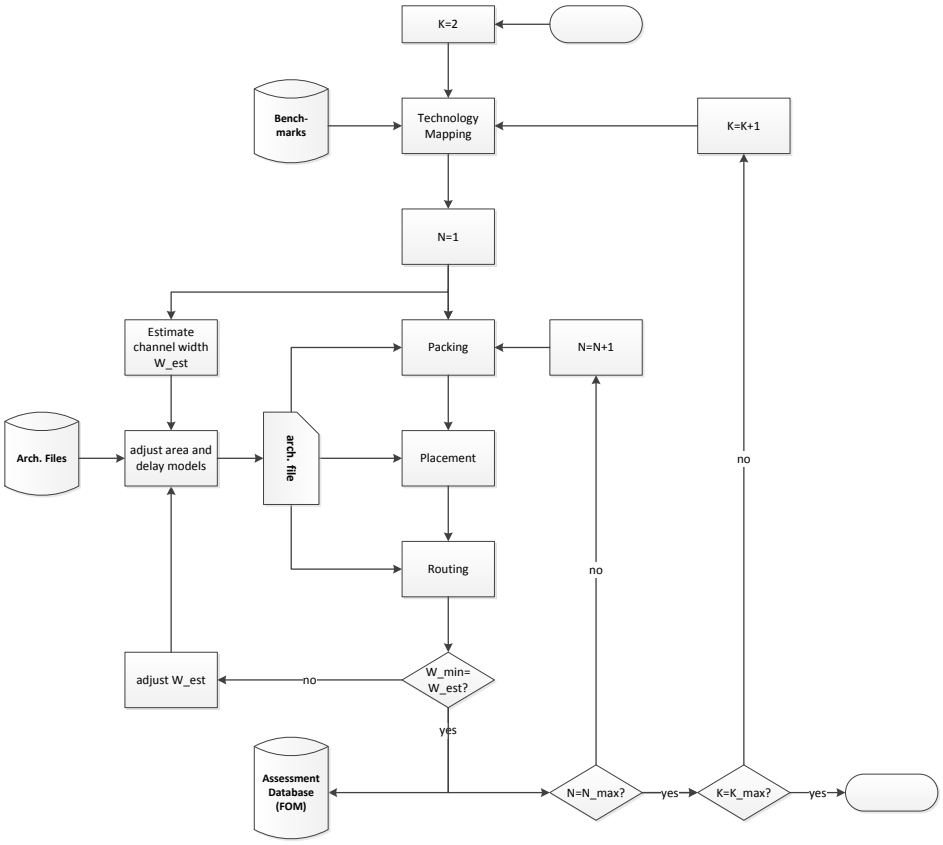


Figure 6.15.: Concept of parametric DSE flow

This information is provided through architecture files. As we can see in Section 5.6 some equations of the area and delay models are dependent on W, which is known only after the routing process. Thus initially the estimated W is used. For an improved accuracy, a feedback is needed to update the architecture file with the actual channel width W and to re-run the area- and/or timing-driven place & route processes , because the exact logic depth and the area of the multiplexers employed in the connection boxes depend on this parameter, which is known only after the routing process. The results in terms of array size, channel width, area, critical path delay are stored in a data base for assessing the figures of merit (FOM). Then the process is repeated with other combinations of N and K in a nested loop to span the design space of interest. The procedure can be repeated with other parameter variations, e.g. switch block type $SBtype$ and/or number of layers L in a 3D *V-FPGA*, to extend the analysis.

6.5. Conclusion

To reduce design time, 7 customizable SoC templates with embedded *V-FPGA* and/or *ViSA* cores that target a wide field of applications characterized by parallelism and control-flow content were proposed in this chapter.

A customization flow that starts with the analysis of the application and gradually tunes the architectural parameters of the *V-FPGA* or *ViSA* based on a new design space exploration methodology was proposed to obtain application specific optimized *V-FPGA* and *ViSA* cores to embed in the custom SoC.

Furthermore, a set of new area related metrics (VLUR, PLUR, MLUR, M2E, LKE, FFC and MSBE) for virtual FPGAs was developed in order to quantify and assess how well an application is mapped onto a custom *V-FPGA* and how well a custom *V-FPGA* is mapped onto a COTS FPGA. Experiments have shown a connection between virtual logic utilization ratio and area. With the PLUR metric it is also interesting to observe the additional waste of area in combinations of custom *V-FPGA* onto COTS FPGA that results form underutilization of platform resources. Experiments have shown that this depends a lot on the granularity of the underlying platform. For instance, the resources of the very fine-grained Actel ProASIC3 platform are perfectly utilized at 100% per allocated cell while resources of the coarser-grained Xilinx Virtex-7 platform are underutilized at 17-63% in average per allocated cell depending on the LUT size of the *V-FPGA*. I.e. there is more waste of area on the Xilinx platform due to quantization effects. MLUR combines PLUR and VLUR to assess the overall mapping efficiency of application mapped onto the *V-FPGA* mapped onto a COTS FPGA. This helps to understand which combinations go well together, even before full implementation of the application. However, it is only one part of the puzzle, because routing area is not considered therein. In the end, what really matters is the overall area, which is known only after implementation. The use of MSBEs as metric in conjunction with the area models from Section 5.6 yields a quick yet quite accurate area estimate without the need to synthesize the *V-FPGA*. Experiments have shown errors less than 1%. This speeds up the process of DSE and customization remarkably, when the *V-FPGA* doesn't need to be synthesized for each combination of architectural parameters.

7. Target Technology Mapping and Characterization Flow

This chapter discribes and discusses ways of mapping a *V-FPGA* architecture onto an underlying platform. Section 7.1 concerns virtualization, i.e. the mapping of the *V-FPGA* onto physical COTS FPGAs. Section 7.2 covers the physical implementation of the *V-FPGA* onto an ASIC process with the aim to embed custom FPGA fabrics into heterogeneous SoCs. This leads to loss of virtualization yet gains significant area and performance improvements.

Closely related to target platform technology is the characterization of the *V-FPGA*, which is significant for area and delay models and thus to improve the accuracy of DSE and application mapping tools. The characterization flow is described in Section 7.3.

7.1. Virtualization

A major aspect of the *V-FPGA* is platform independency. Hence it is designed to be mapped out of the box on any COTS FPGA. The key component for achieving this is a design methodology with self-generating VHDL models exploiting *GENERATE* loops in conjunction with generic parameters. Thus, despite the extensive capabilities for customization, the vendor IDE tools supplied with the COTS FPGA are sufficient for mapping the virtual layer and there is no need for additional tools like e.g. code generators for scaling and customizing the architecture. Irrespective of the vendor (Xilinx, Microsemi, Altera (now Intel), etc.), the steps are fundamentally the same:

1. Create a new project and select the target device.
2. Add the VHDL source files of the *V-FPGA* to the project.
3. Set the generic parameters either in the top-level module or in the package *initconfig*.
4. [Optional]: introduce platform specific optimizations if not inferred automatically, such as
    - Replace generic elements by supported macros, where the tools don't do it automatically. E.g., multipliers or adders in integrated *ViSA* cores are generic to be supported by Actel devices, yet Xilinx or Altera devices contain dedicated hard macros that are more efficient than generic logic implementations of such operations.
    - In Xilinx FPGAs slice-internal MUXF BELs can be used as a replacement for the generic LUT-based MUXs [25].

- Manually insert clock buffers, global net buffers and define clock regions to reduce skew and to enhance timing in flip-flops and configuration signals.

While these techniques are a few examples of improving overall area efficiency and performance, they suffer portability if they need to be added manually. Adequate replacements need to be found and the architecture re-implemented when changing the platform.

5. If desired, integrate the *V-FPGA* module with other system modules either using SoC templates as introduced in Section 6.1 or in a fully custom system architecture. For easier integration, the *V-FPGA* sources include an optional peripheral unit with AMBA APB bus interface that allows a memory-mapped access of *V-FPGA* I/Os by microprocessors or other modules with compatible bus interface (see Section 6.1).
6. Define the clock and I/O constraints for the system. The *V-FPGA* is a multi-clock architecture with separate clocks for configuration and logic. Therefore, separate clock constraints are possible. Generally, the configuration infrastructure can be clocked higher than the logic as it is fully pipelined with short paths due to daisy-chaining of configuration flip-flops.
7. Synthesize the design with the integrated synthesis tools. Each COTS FPGA vendor integrates at least one synthesis tool in their IDE, but provides also interfaces for coupling external synthesis tools.
8. Run place & route process with the provided vendor tools. A challenge in timing driven routing is that the *V-FPGA* (and most other FPGA architectures) inherently contain possible combinational loops in the routing infrastructure due to flexibility. Actually, the *V-FPGA* will be configured in a way that such loops will not occur unless intended by the application. Yet, the place & route tools of the underlying platform, being not aware of the applications that are mapped on the *V-FPGA*, need to assume such loops as possible paths. Most vendor tools are capable to break such loops automatically. In cases where this fails, *false path* constraints need to be introduced.
9. Generate the bitstream and program the device.

Figure 7.1 shows an exemplary toolflow for mapping the *V-FPGA* onto an Actel/Microsemi FPGA, that supports all the steps listed above. It exploits the Libero IDE toolset from the manufacturer Microsemi. On the left side there is the hierarchical project view with components and subcomponents. The top-level *V-FPGA* component (*TOP_V_FPGA*) is integrated with a microprocessor subsystem through APB and AHB busses. On the right side the toolflow is interlinked with arrows indicating the order of the steps.

It is clear that virtualization and the flexibility that it offers come at the price of high area overhead. As far as related work is concerned, Lysecky et al. report for their virtual FPGA an area overhead of roughly 100x [65]. Brandt et al. achieved an area overhead of 40x by employing platform exclusive LUTRAMs in their ZUMA [20] architecture while sacrificing portability. Thereby both related works define the overhead as number of physical resources that are required to realize virtual resources (including logic block, associated routing circuitry and configuration memory) of a similar complexity.

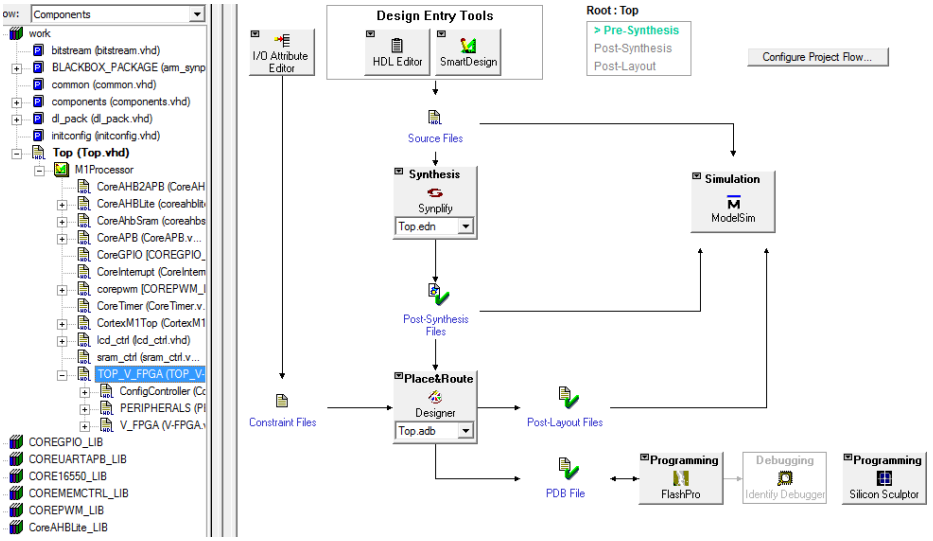


Figure 7.1.: Toolflow for mapping *V-FPGA* onto an Actel/Microsemi COTS FPGA

Table 7.1.: Module-level resource utilization when mapping one tile of a *V-FPGA* core with $K = 3$, $N = 4$ and $W = 4$ onto a Xilinx Spartan-3 device

	Slice Reg	**LUTs**
CLB	108	116
PSM	32	16
total	140	132
per *V-FPGA* LUT	35	33

Lysecky et al. count the utilized resources of a Xilinx Spartan-2E when implementing one virtual CLB and one virtual Switch Matrix. Then they devide that number by 4, which is the number of LUTs per virtual CLB. The complexity of the virtual LUTs, being 3-input LUTs, is actually less than the complexity of the platform's 4-input LUTs. However, Lysecky et al. generously round up the overhead so that a factor of 100x seems more or less reasonable in the end.

For a direct comparison, following the same definition for resource overhead, a *V-FPGA* with $K = 3$, $N = 4$ and $W = 4$, being tuned for a similar complexity as the virtual FPGA by Lysecky et al., is analyzed. Table 7.1 concludes an area overhead of 35x when mapped on a Xilinx Spartan-3[1] device. This is around 2.8 times less than Lysecky's virtual FPGA.

The resource overhead for the ZUMA architecture is reported on a Xilinx Virtex-5 device. To compare the *V-FPGA* with ZUMA the same baseline platform is chosen and the logic

[1]Spartan-3 was used as hosting platform instead of Spartan-2E because the latter was not supported anymore by the employed vendor tools. Since both are based on 4-input LUTs, the results are assumed to be transferrable.

Table 7.2.: Module-level resource utilization when mapping one tile of a *V-FPGA* core with $K = 6$, $N = 8$ and $W = 55$ onto a Xilinx Virtex-5 device

	Slice Reg	**LUTs**
CLB	?	?
PSM	?	?
total	784	634
per *V-FPGA* LUT	98	79

Table 7.3.: Key architectural parameters in virtual FPGAs - a comparison of the *V-FPGA* with related works

	V-FPGA	**Lysecky [65]**	**ZUMA [20]**
LUT size	K	3	6
Cluster size	N	4	8
Channel width	W	4 bi-dir	112 uni-dir
SB topology	Wilton, Universal, Disjoint with $f_s = 3$ or $f_s = 5$	fixed with $f_s = 12$	fixed
3D stacked layers	L	-	-

related parameters of the *V-FPGA* are tuned to match a similar logic complexity as ZUMA, i.e. LUT size $K = 6$ and cluster size $N = 8$. With respect to channel width, the two architectures have different routing infrastructure, thus there is no direct comparison possible. However, Brandt et al. state that they have chosen their channel width 112 based on the minimum requirements to route all the MCNC20 benchmarks. In *V-FPGA* with $K = 6$ and $N = 8$ the full routability of all the MCNC20 benchmarks is already achieved at a channel width of $W = 55$. With these parameters the utilization for mapping the *V-FPGA* onto Xilinx Virtex-5 is reported in Table 7.2 from which we can conclude a resource overhead of around 98x, which is around 2.4 times more than ZUMA, which used platform exclusive elements. However, this comparison doesn't consider neither the portability aspects nor application specific parameter tuning of the *V-FPGA*. In [34] we have shown a possible variance of up to $\pm 95.9\,\%$ on the same target platform and for the same application just by tuning of parameters.

As summarized in Table 7.3, the *V-FPGA* has the most flexible architecture. In contrast to other virtual FPGAs, the LUT size, cluster size, channel width, switch block topology and 3D stacking are parameterizable, to name only a few. This advantage can and should be used to increase mapping efficiency, which improves the overall area efficiency.

Finally, it is of interest how well the virtualization aspects are fulfilled in comparison with related works. This is summarized in Table 7.4 with the *V-FPGA* excelling in most aspects. In terms of accessibility ZUMA is better as the source files are openly available in the internet.

Table 7.4.: Fulfillment of virtualization aspects

	V-FPGA	Lysecky [65]	ZUMA [20]
Portability	+++	+++	-
Partial and dynamic reconfiguration	+++	-	-
Adaptivity	+++	0	0
Prototyping and emulation	+++	+	+
Accessibility	+	+	+++

7.2. From Virtual to Physical

To radically enhance area efficiency and performance the *V-FPGA* can be mapped in an ASIC, which can be also more cost efficient if the sales volume permits. Then the parameterizable virtual FPGA becomes a specialized physical FPGA. Combining the *V-FPGA* fabric with dedicated hard macro blocks such as microprocessor cores, DSPs, etc. yields heterogeneous reconfigurable SoCs. Meanwhile, there is a big variety of these macros available as licensable or open source IP cores. Bringing the highly customizable *V-FPGA* in the game yields practically almost unlimited possibilities for specialized yet flexible SoCs. This is obviously attractive especially for Cyber-Physical Systems (CPSs), sensor networks and Wireless Sensor Networks (WSNs) and IoT for the following reasons:

- These types of devices have typically tight requirements on area, power and costs.
- Having pre-determined tasks, the requirements can be easier met on specialized SoCs than on general purpose SoCs.
- Configurability yields adaptivity in the field. The option to alter parts of the circuit during runtime eventually avoids the necessity to change parts in some cases and reduces service costs. For instance, changing the communication protocol of a sensor node is a typical scenario for the reconfigurable fabric. Another example is the reuse of the same SoC for a number of different sensors and the reconfiguration of the signal conditioning whenever the sensor changes. A non-reconfigurable ASIC or SoC would need replacement in these cases.
- Being emerging technologies, time-to-market is important to acquire and secure market share. Building upon libraries of parameterizable IPs (including *V-FPGA*) saves development time.

Hence, this subsection governs the physical implementation of the *V-FPGA* to offer a flexible solution for embedding customized FPGA cores in SoC ASICs. The focus is on a soft-IP approach utilizing standard-cells as it allows full-blown customization of the FPGA architecture and reduces design time compared to a pre-laid-out full-custom hard-IP approach, accepting that the latter approach would likely yield higher optimization on the physical layer yet not on the architecture layer. From a SoC perspective the soft-IP approach gives also more flexibility to the floorplan and reduces whitespace through a) tuning the aspect ratio or b) defining a custom outline shape of the FPGA core or c) performing a flat layout. Furthermore it allows feedthrough insertion to improve routability and timing closure of signals crossing different partitions of the SoC.

Figure 7.2 illustrates the flow for mapping the *V-FPGA* onto an ASIC. Generally we can divide the flow in two parts, that are explained in the following: I) mapping of the RTL VHDL models onto standard cells of a given technology library and II) creating the layout of the resulting schematic (technology-mapped netlist).

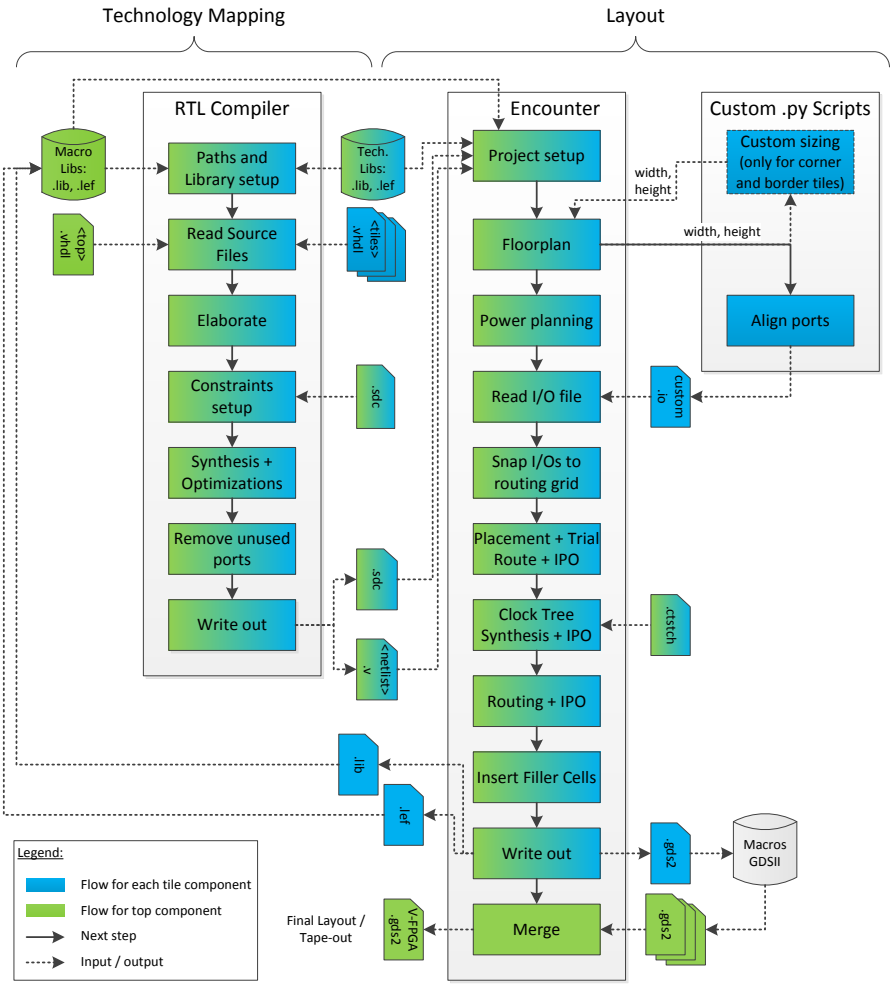


Figure 7.2.: Physical design flow for mapping the *V-FPGA* onto an ASIC

7.2.1. Standard-cell Mapping

A nice design peculiarity of the *V-FPGA* is that the source files can remain always the same no matter how the architecture is tuned and what the target technology is. Apart from the source files there are technology libraries and timing constraints needed. In the following examples the 45 nm GPDK libraries from Cadence are used for demonstration purposes. The design steps are performed with the tool *Encounter RTL Compiler* from Cadence [22], yet other similar tools such as *Design Compiler* from Synopsis can be engaged as well.

After setting up the library and source file paths, the first step is to import and elaborate the VHDL source files of the *V-FPGA*. Depending on the desired layout strategy (flat, hierarchical top-down or hierarchical bottom-up) either all source files are imported or only the tile components with all their recursive sub-components. The preferred layout strategy is hierarchical bottom-up as described in Section 7.2.2. Therefore, we start with the tile components, which after their layout become library macros that can be used in a second run for the top-level design.
During the elaboration phase the syntax and semantics of the source files are checked, the design is translated into an internal data structure of the tool, registers are inferred and higher-level optimizations (e.g. elimination of dead code) are performed. Furthermore it is possible to overwrite generic parameters during elaboration through the command option *-parameters {<par1> <par2> ...}*. This is an alternative to changing the parameters in top VHDL file.

In a next step constraints are applied. This includes user constraints that are provided in a *.sdc file as well as design rule constraints that are inferred from the technology libraries. The minimum user constraints for the *V-FPGA* are clock constraints. The *V-FPGA* is a multi-clock design and supports different clocks for logic and CLB-, PSM- and IOB-configuration infrastructures.

Then the synthesis is performed that transforms the design into a gate-level structure with standard cells from the technology libraries. First a generic gate-level netlist is created and a number of technology independent optimizations are performed, such as datapath synthesis, resource sharing, mux optimization, structuring and removing of redundancy. Then the actual mapping to standard-cells from the technology libraries takes place, including restructuring, buffer-insertion, etc.. In RTL Compiler this is called *Global Focus Mapping* and is a multi-objective task that considers timing, area and power in the overall solution space. Global optimizations follow, that include sizing of cells and optimization of buffer trees. Further incremental optimizations take place to fix design rule violations and improve timing and area. More details on this can be obtained from [22].

The next step is then to remove unused (i.e. unconnected) ports from the design. This is not done by default in RTL Compiler but needs to be invoked by a command. This step is only necessary for the 2D *V-FPGA* as it contains a few unconnected ports for the following reason:
The *V-FPGA* design has the same source files for 2D and 3D flavours which are distinguished by the parameter *L*. In the 3D flavour each layer features additional ports for TSVs. The 2D *V-FPGA* is actually a layer within the 3D *V-FPGA*, however it doesn't require TSVs if it's purely 2D. Internal signals for TSVs are conditionally unconnected in the 2D *V-FPGA* through *if...generate* statements, yet these kind of statements can not be

applied to port definitions and therefore the unneeded ports persist in the RTL design and need to be removed from the synthesized netlist through the command *delete_unloaded_undriven -all*. Figure 7.3 shows the hierarchical RTL schematic of a tile component after synthesis and optimization. In that example, the parameters $K = 3$, $N = 2$ and $W = 2$ were chosen, which are intentionally rather small in order to reduce the complexity in favour of a better readability of the schematics. The upper hierarchy levels show blackboxes of

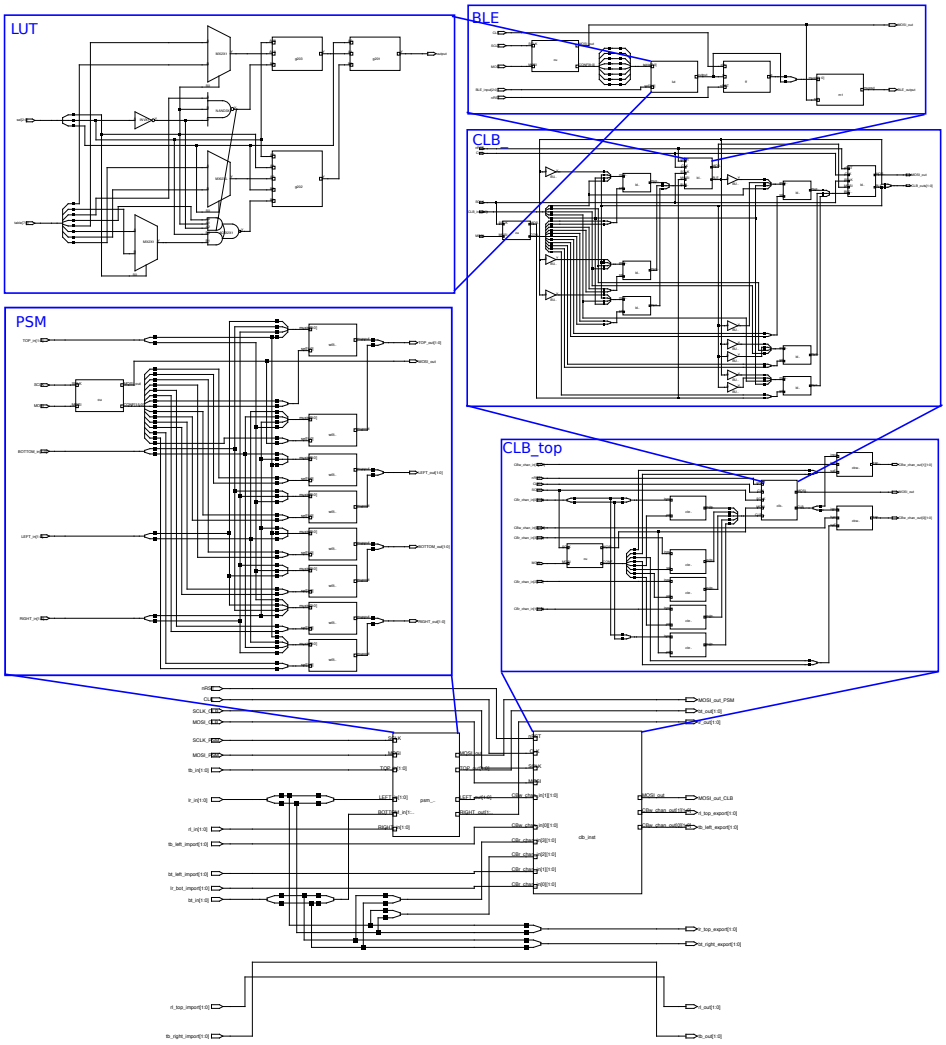


Figure 7.3.: Hierarchical schematic of a tile component after synthesis with RTL Compiler

sub-components and interconnects. The deeper we dive in the hierarchy, the more black boxes are resolved into more concrete elements. The schematic of the LUT component already shows elements from the technology library. Furthermore we can see that the tool has also inserted buffers as part of the optimization steps.

Finally the technology mapped and optimized netlist (in a verilog file) and a constraint file (*.sdc) can be exported. RTL compiler is also able to generate script files that automatically set up the design and imports the files needed for the layout in the next tool *Encounter*, though these settings can be done also manually.

7.2.2. Hierarchical Layout

Due to the regular structure of the *V-FPGA*, the layout can be facilitated by a hierarchical flow. The largest recurring component in the *V-FPGA* architecture is a tile. There are 9 different tiles - 4 corner tiles, 4 border tiles and 1 inner tile - whereby the most recurring one is the inner tile. The proposed design methodology follows a bottom-up strategy, whereby in a first design round (see Section 7.2.2.1) it lays out each of these 9 tiles as macro blocks with well-defined dimensions and I/O locations. Then, in the second round (see Section 7.2.2.2), multiple instances of the same macro blocks are placed relative to each other and interconnected. This approach has a number of advantages:

- shorter implementation time
- equal layouts and characteristics of macro tiles
- more predictable over-all performance and area
- efforts can be concentrated on optimizing the few macros

7.2.2.1. Layout of Macro Tiles

Starting with the inner-tile, the project is set up and the synthesized and technology mapped design described in Section 7.2.1 is imported. This includes settings for paths to the netlist, technology libraries and constraint file as well as setting constraint modes, timing analysis modes and the selection of the top cell, which in this case is the inner tile. Furthermore the global nets for VDD and VSS are defined and connected.

Then the floorplan is specified. Rather than entering absolute dimension values, we specify the target aspect ratio and utilization for the inner-tile which the tool uses to derive absolute dimensions. An aspect ratio of 1.0 and a target utilization of 80 % are a good start for the inner tile, though higher utilization is possible as well. Note that the resulting dimensions might not satisfy exactly the target aspect ratio, because standard cells have a fixed height and the height of the module dimension corresponds to multitudes of standard-cell rows with equal heights. For instance, applying an aspect ratio of 1.0 and a utilization of 80 % to an inner-tile with the parameters $K = 4$, $N = 5$ and W=23 results in a floorplan with the dimensions 88.2 μm x 85.5 μm in a 45 nm technology.

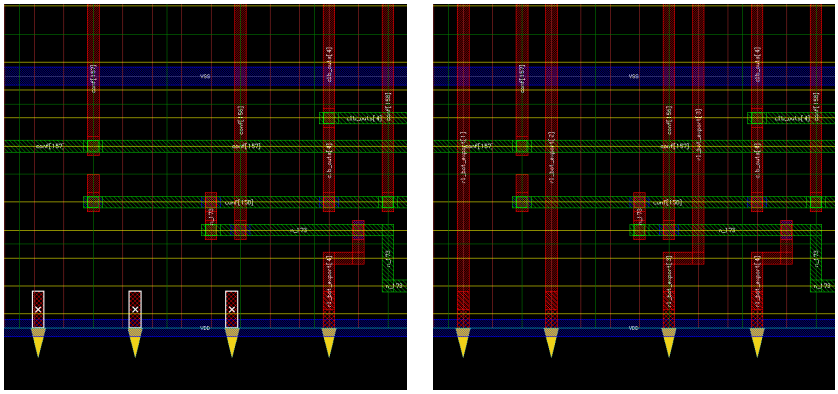
The next step is the power planning to add vertical power stripes for VDD and VSS at the left and right borders of the block. In order to avoid standard-cells from being placed

on these stripes, placement blockages have to be set around the power stripes. Then the standard cell rows in that areas can be cut. Note that this will effectively increase the density as the available area for standard cell placement is now slightly smaller than initially specified in the floor planning due to the placement blockage around the power stripes and the cutting of standard cell rows. Depending on the technology process it might be also necessary to place endcap cells at the ends of the rows for a correct bias of P+ and N+ substrates.

Before placing the standard cells, the locations of the block I/Os need to be specified. If unspecified, the placement tool will automatically assign locations to the I/Os. However at this design stage the tool doesn't know how the top level design of the *V-FPGA* will be assembled and what will be the relative positions of the tile macros to each other. Thus it can happen that I/Os, that were intended to connect a tile macro with another one left to it, will be placed e.g. on the bottom side and cause a poor routing. In order to avoid this, the *V-FPGA* framework contains a python script called *align_ports.py* (see Section A.4) to generate explicit I/O location constraints for the tile blocks with the correct orientation and optimal alignment that reduces path lengths for inter-tile connections. The required parameters are channel width W, width and height of the tile macro block. The result is a custom I/O file with offset positions of I/Os on all four sides, that can be imported into the physical design tool (here Cadence Encounter). However, with user defined placement of I/Os there is a high chance that many of the I/Os are not properly aligned with the manufacturing grid. This can lead to connectivity problems in a later design phase because signals are routed only with alignment to the routing grid. This problem is shown in Figure 7.4a where the router fails to connect I/Os that are off the grid. One possibility to overcome this problem is to consider this in the python script by rounding the I/O locations to the nearest valid track on the routing grid. However this requires exact knowledge about the spacings and offsets of the routing grid. A more elegant solution is to issue the *legalizePin* command after any user defined I/O placement. This will snap the I/Os to the routing grid by slightly shifting misaligned I/Os to valid locations that are in line with the grid. Furthermore it will dissolve other violations such as overlaps, etc.. Figure 7.4b shows the result of this correction that was applied on the initial user defined placement of Figure 7.4a.

In a next step the standard cells are placed initially, followed by trial route and In Place Optimization (IPO). IPO at this stage tries to fix possible setup time violations by moving of standard cells. Thereby, the timing analysis for IPO relies on the parasitics estimated after the trial route, which is a quick and dirty routing attempt for rough estimations on wiring and congestions. Figure 7.5 shows a placed design of an inner tile. Left and right are the red power stripes with placement blockage around them. The blue standarad cells are placed in equidistant rows. The amoeba view in Figure 7.6 shows how the PSM-related and CLB-related cells are distributed. The CLB area seems larger than the PSM area, however this should not be taken as an indicator for logic area vs. routing area, because the local routing resources (the connection boxes and the input-MUXs for the BLEs) are within the CLB-instance.

Once the placement is fixed, the Clock Tree Synthesis (CTS) can take place, which creates the clock trees. The clock trees provide the standard cells with the clocks. The branches of a clock tree are typically H-shaped and all leafs have approximately the same path



(a) (b)

Figure 7.4.: Routability in user defined alignment of I/Os: a) locations as defined, b) after snapping to routing grid

length to the root point. This attempts to keep the delay times on the clock lines as equal as possible in order to reduce clock skew. Additional buffers and inverter cells can also be inserted, so that in some places equal delay times can be also achieved without equal path lengths. The target specifications (MaxDelay, MinDelay, SinkMaxTran, BufMaxTran, MaxSkew) as well as permissible buffers that can be inserted are defined in and provided by an external .ctstch specification file. Figure 7.7 shows the clock trees (white lines) after CTS. There are three different clocks in the inner tile module: a clock for the CLB configuration, a clock for the PSM configuration and a clock for the sequential logic flip-flop. Accordingly there are three clock trees. After CTS another IPO is executed. Thereby, the cells can be still shifted slightly to provide as accurate clock paths as possible.

The next major step is the routing of signal wires. Before that, in a special route process, the power wires are routed from the horizontal and vertical power strips to the standard cells to provide the voltage supply and ground connection. Then the actual routing of the connections between the cells and to the pins is carried out in two phases. The global routing phase partitions the design into a set of smaller routing problems (bins) and creates rough routes for the connections between the bins. During detailed routing, several iterative routing tasks are carried out in the individual networks. After the routing another IPO is performed to optimize hold- and setup time slacks. When everything is fixed, then finally filler cells need to be inserted. These cells fill the empty gaps between the standard cells to create a planar surface as the basis for the overlying metal layers. One could add the filler cells also in an earlier design phase, yet it makes more sense to add them in the end when all the placements and shifts are finalized. Figure 7.8 shows the final layout of the inner tile macro, that is then exported as black box macro. We can see many straight and long horizontal lines in the upper area and many straight and long vertical lines in the right area. This comes from the fact that the PSM is placed in the upper right quarter of the inner tile and the routing channels go through the PSM.

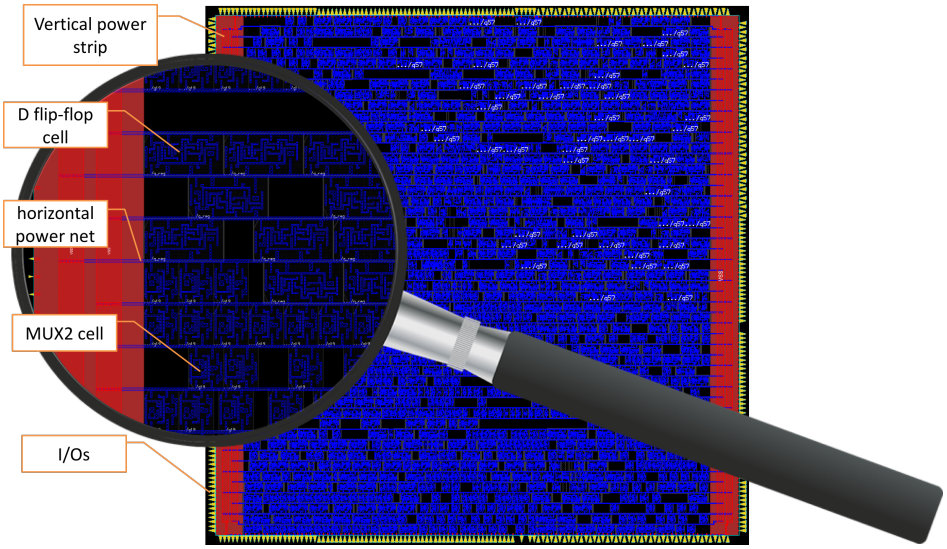


Figure 7.5.: Standard cell placement of *V-FPGA* inner tile macro in a 45 nm ASIC process

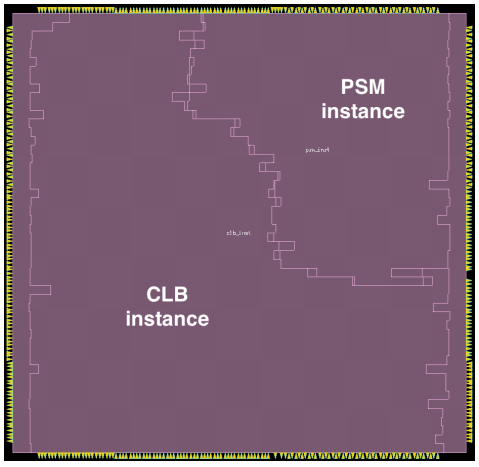


Figure 7.6.: Amoeba view of a placed inner tile macro showing the outlines of CLB and PSM instances

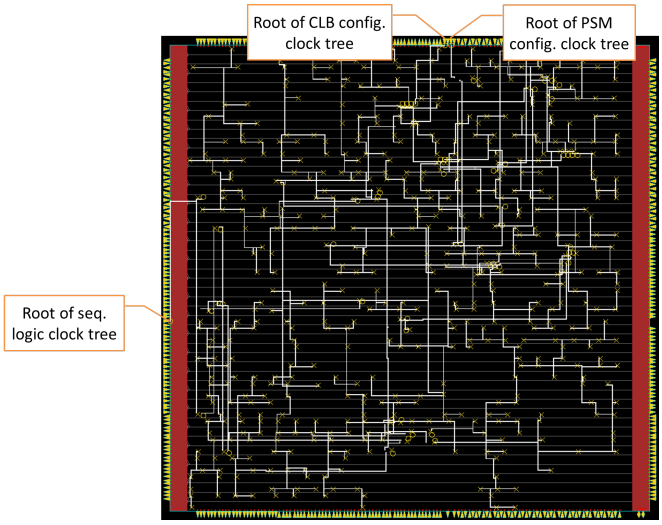


Figure 7.7.: Result of Clock Tree Synthesis (CTS) in inner tile macro of *V-FPGA*

Then the layout is exported in a standard stream out format such as Graphics Database System (GDSII) or Open Artwork System Interchange Standard (OASIS). Furthermore, an abstract of the macro layout is generated in Library Exchange Format (LEF) as well as a .lib timing library file. These two files are needed in the top level design to represent the macro and to consider the associated delays.

Then all these steps are repeated for the other tiles. However, the floorplanning of the other tiles is done with absolute dimensions in order to achieve a unified alignment of I/Os for optimized inter-tile connections. A Python script derives the absolute dimensions of the other tiles from the dimensions of the inner tile based on the equations in Table 7.5. Therein, w_i and h_i are the width and height of the inner tile macro. The variables a_i, a_{bl}, a_b, a_{br}, a_r, a_{tr}, a_t, a_{tl} and a_l are the estimated area requirements for inner tile, corner-tile bottom-left, border-tile bottom, corner-tile bottom-right, border-tile right, corner-tile top-right, border-tile top, corner-tile top-left and border-tile left respectively, that are obtained from the synthesis reports. Basically, the strategy behind this sizing methodology is to keep the edges between adjacent tiles equal in order to exactly match the I/O alignment for the inter-tile connections.

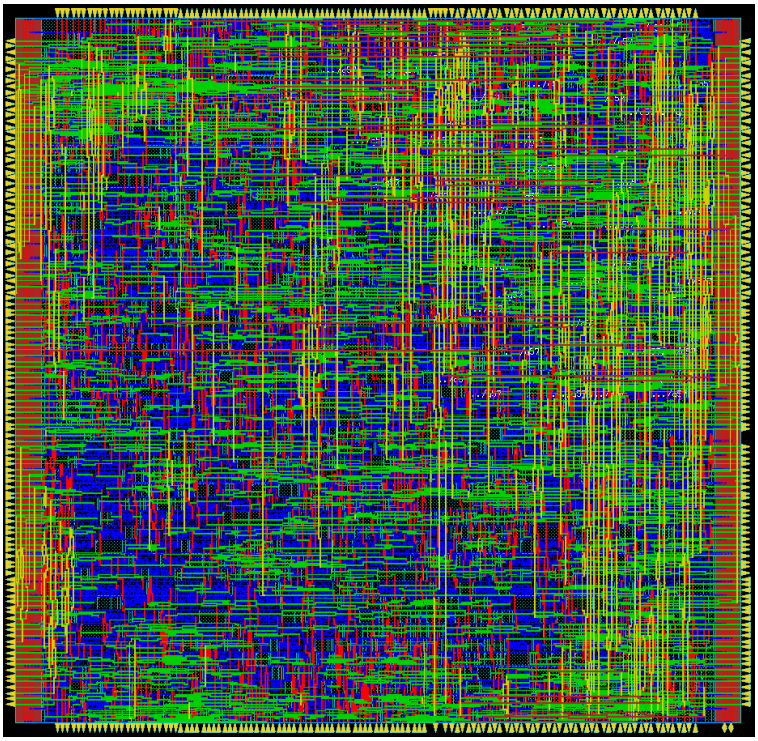


Figure 7.8.: Layout of *V-FPGA* inner tile macro after routing

	width	**height**
inner tile	w_i	h_i
border-tile bottom	$w_b = w_i$	$h_b = max(\frac{a_b}{a_i} \cdot h_i, \sqrt{\frac{a_{bl}}{a_i}} \cdot max(h_i, w_i))$
border-tile left	$w_l = max(\frac{a_l}{a_i} \cdot w_i, \sqrt{\frac{a_{bl}}{a_i}} \cdot max(h_i, w_i))$	$h_l = h_i$
corner-tile bottom-left	$w_{bl} = w_l$	$h_{bl} = h_b$
border-tile right	$w_r = \frac{a_r}{a_i} \cdot w_i$	$h_r = h_i$
corner-tile bottom-right	$w_{br} = w_r$	$h_{br} = h_b$
border-tile top	$w_t = w_i$	$h_t = \frac{a_t}{a_i} \cdot h_i$
corner-tile top-right	$w_{tr} = w_r$	$h_{tr} = h_t$
corner-tile top-left	$w_{tl} = w_l$	$h_{tl} = h_t$

Table 7.5.: Sizing of remaining tile macros based on the width and height of the inner tile macro

7.2.2.2. Layout of *V-FPGA* Top-Level Module

Once all tiles are completely laid out, the top level module that assembles the *V-FPGA* out of the tile macros can be implemented. The proposed methodology to obtain a top-level netlist that binds all tile macros is to repeat all steps from Section 7.2.1, yet this time employing the following modifications:

- Extend the library setup to contain apart from the technology libraries also the .lib and .lef files of the tile macros that were created in the previous steps.
- Read in only the top-level VHDL source file of the *V-FPGA*. All other source files are not needed as the sub-modules should be replaced by the laid out macros that have become library components.

With these modifications all synthesis and technology mapping related steps are performed in the same way as it was done for the tile components. The resulting netlist contains instances of the tile macros and additionally buffers from the technology library to improve the performance of inter-tile connections.

The layout procedure for the top-level module involves similar steps as for the macro tiles with some differences in the project setup and the power planning. The project setup needs now to import the newly created Verilog netlist of the top-level module and the corresponding .sdc file. Furthermore, the library paths need to be extended to the .lib and .lef files of the tile macros in addition to the technology libraries. All other project settings (constraint modes, timing analysis modes, global VDD and VSS nets) are equivalent to the settings for the tile macros.

The floorplan can be specified by aspect ratio and target utilization ratio. The aspect ratio can be set close to the actual aspect ratio of the inner tile macro. The utilization ratio can be set rather high ($> 90\,\%$) as the array of tiles follows a regular structure and the inter-tile connections are rather easy to route due to the custom alignment of tile ports that was done exactly for this purpose.

The power planning is done with each an outer power ring for VDD and VSS around the core and with equidistant vertical and horizontal power stripes inside the core forming a power grid as shown in Figure 7.9.

The next step is to place the macro tiles inside the core, followed by trial route and IPO. In many designs macros are placed manually in order to take control of the relative positions of macros to each other, which has a big impact on the quality of the routing. However, the *V-FPGA* tile macros have custom I/O alignments with the purpose of high quality inter-tile routing when placed as intended relative to each other. Knowing that the placer targets a good routeability, the assumption is that the auto-placer should be able to place all instances of the tiles in an array as intended with the correct orientation and the correct relative locations to each other, because this yields undoubtly the best routing results and requires only 1-2 metal layers. Indeed, the experiment in Figure 7.10a with 361 tiles demonstrates that all tiles were placed correctly in an array of 19x19 tiles. Thus, we can conclude that the custom alignment strategy for the I/Os of the tile macros facilitates placement and routing of the array of tiles. Having a closer look at the results we can see that most of the tiles have straight connections to their neighbours, yet there are a few tiles that have a slight offset causing angular routes and change of metal layers (see

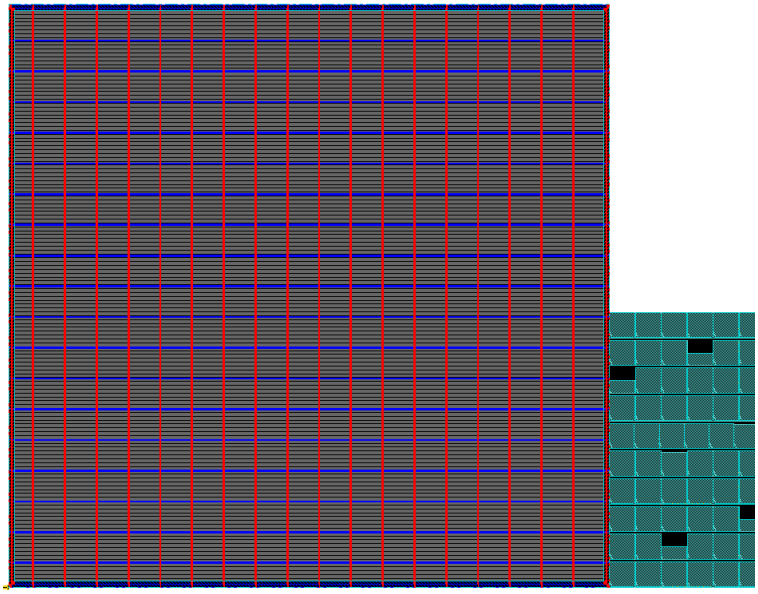
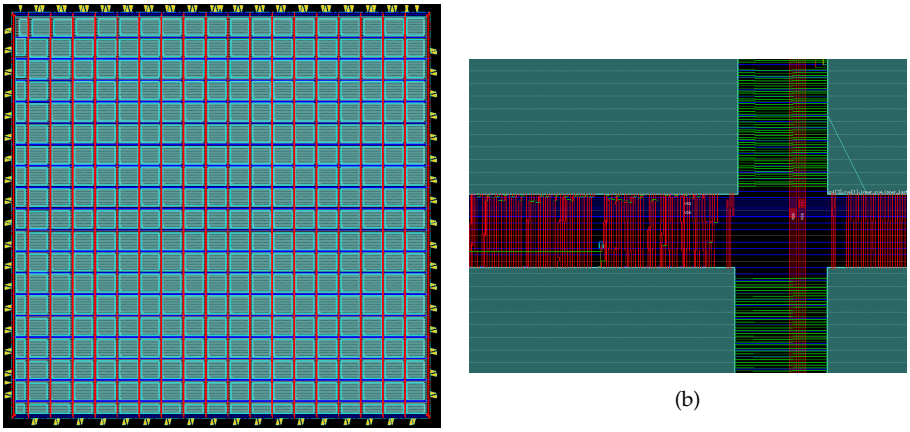


Figure 7.9.: Floor plan of *V-FPGA* with power rings and power grid

Figure 7.10b). Still the routes are very short, yet perfectionists might desire optimum face-to-face alignment of adjacent tile macros with straight connections in all directions. The alternatives are then either to write a tcl script that defines absolute locations for all tiles as intended or to execute the *relativeFPlan* command with user defined array constraints. The latter approach relies on specifying offset, anchor point and orientation of each macro relative to another macro.

The next steps are the same as for the tile macros, including CTS, post-CTS IPO, routing, post-route IPO and filler cells. Figure 7.11a shows the clock trees of the *V-FPGA* array. The macros, power nets and other signal wires are hidden for a better visibility of the clock trees. The X shapes mark the connection points of the trees (mainly the leafs) and the circles mark buffers that were inserted during CTS. Figure 7.11b shows the complete layout after routing. The resulting core size is 2000 μm x 1938 μm for an array of 19 x 19 tiles corresponding to 18 x 18 CLBs, each with $N = 5$ LUTs of size $K = 4$ at a target frequency of 500 MHz. The routing infrastructure has a channel width of $W = 23$ tracks per channel.

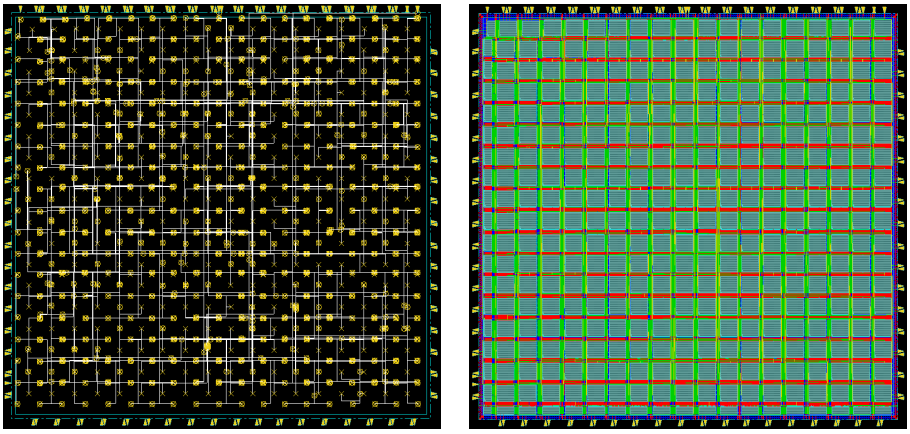
Note that the macro tiles are represented by their abstracts only as the imported .lef files don't contain layout information. To obtain all layout information in one file for manufacturing, the stream-out of the top layer is merged with the GDSII files of the macro tiles. The result is shown in Figure 7.12, where now we can see the complete layout. While the previous figures were produced with Cadence Encounter tools, this one was produced



(a) (b)

Figure 7.10.: Automated placement of macro tiles in *V-FPGA*: a) array of 19x19 tiles, b) detailed view of inter-tile connections

with the tool KLayout [55] from the final GDSII stream-out file. This explains the different colour scheme.



(a) (b)

Figure 7.11.: a) Clock trees in *V-FPGA*, b) layout with abstract tile macros after routing

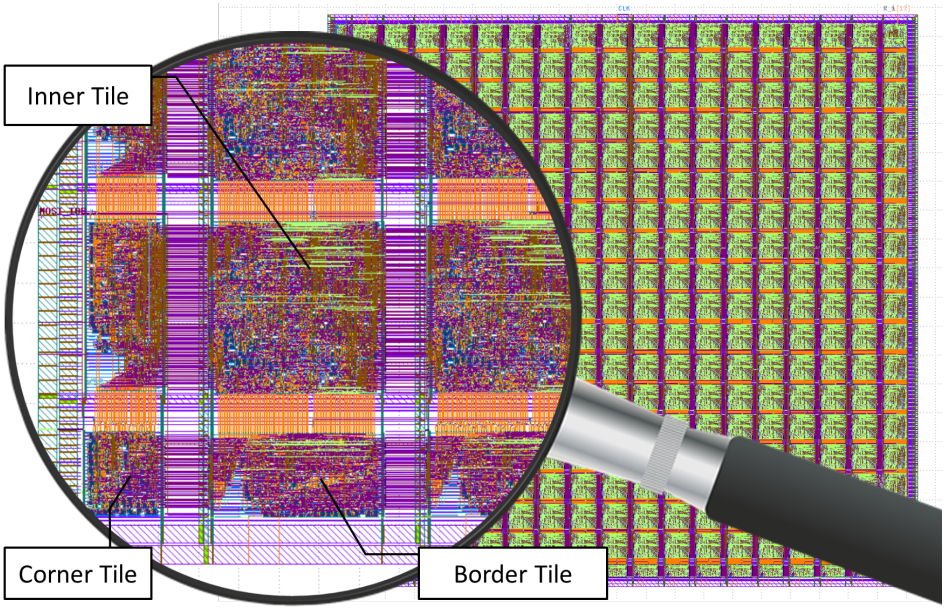


Figure 7.12.: Merged layout of *V-FPGA* top level and tile macros

Table 7.6 shows a qualitative comparison with existing embedded FPGA approaches. Hard IP cores can not be customized by the system designer, but some manufacturers offer to build custom cores per order. The proposed *V-FPGA* in a standard-cell soft-IP approach is the most flexible and accordingly allows unparalleled application-specific customization with more than 20 architectural parameters.

Table 7.6.: Embedded FPGAs - a comparison of the *V-FPGA* with related works

	FlexEOS [82]	**Neumann eFPGA [79]**	**MENTA eFPGA [71]**	**EFLX [32]**	**Achronix SpeedCore [1]**	**ADICSYS [3]**	***V-FPGA***
IP type	hard IP	mixed	hard IP	hard IP	hard IP	soft IP	soft IP
technology	90 *nm*	?	28 *nm*, 14 *nm*	40 *nm*	22 *nm*	any	any
logic cells	LUT4	clusters of LUT2 + adder logic	?	2x LUT4	4x LUT4 + 4-bit ALU	?	parameterizable *N*xLUT-*K*
heterogeneous blocks	-	-	DSP, memory	optional mix with DSP cores and memory	DSP, memory	?	*ViSA* cores, memory
customization	-	combination of leaf cells	per order only: aspect ratio, number and types of blocks	concatenation of cores	per order only: aspect ratio, qty. of cells	?	20+ parameters (see Table 4.1 in Chapter 4)

7.3. Characterization Fow

Characterization not only reveals the area requirements and the maximum possible performance of the *V-FPGA* mapped on a specific target platform technology, but is also important for the accuracy of design space exploration and application mapping. This is because place & route are area- and time-driven, relying on the provided area and delay models with parametrizable technology dependent constants. Borrowing the constants from a different target technology than the intended one will still result in a functionally correct mapping. However, this commodity, that with respect to virtual FPGAs has been followed by the prior art, can be misleading for architectural parameter choices as we have shown and discussed in [34]. The apparently optimum parameter choice maybe actually sub-optimal if the technology related parameters and constants are too far from being accurate. Therefore, the proposed characterization flow aims to properly assess the relevant characteristics. It should not be confused with standard cell characterization which follows a different scope and a different methodology with analog simulation and signal sweeps. This is not necessary as the utilized standard cell libraries are already characterized. Instead, the characterization described in this chapter has the purpose to extract only the characteristics that are needed for the area and delay models employed in the DSE and application mapping flows.

Irrespective of the underyling platform, the general characterization flow for the *V-FPGA* described in the following:

1. Select and parameterize one tile to characterize, which is representative for most of the other tiles. Obviously this is the inner tile which has the highest occurrence and complexity. Architectural parameters that are already certain can be specified

accordingly in the generic map of the tile. Other parameters that are subject to DSE can be set to an average value. Generally, the parameters selected for characterization have a negligible impact on the accuracy of the area models, because the models are based on MSBEs which due to their fine granularity are rather independent on the parameters. However, the accuracy of delay models can be affected by net delays that to some extent depend on the complexity of a tile.

2. Synthesize, place and route this tile alone with the methodologies and tools described in Section 7.1 and Section 7.2.
3. Extract the relevant area information from the generated reports. Since the area models rely on Minimum Size Basic Elements, they form the minimum set of elements for which we require area information. Particularly the area information of a MUX2, MUX4, AND2, D-flipflop and in case of physical implementation also of a buffer are needed. All other elements are derived from these MSBEs. In case an architectural parameter such as the LUT size K is fixed, the accuracy can be improved by extracting the actual area of a LUT rather than deriving it from the MSBEs. Yet, experimental results have shown that the difference - if any - is negligible and has no effect on the DSE and application mapping results. It should be also noted that it makes no difference whether actual geometric area figures are used or normalized numbers. For parameter tuning the ratio alone is important.
4. Extract the relevant timing information. We need the delays through MSBEs, the worst case delay through a LUT, the setup-time and CLK-to-Q delay of a D-flipflop and the delay of an average net. These numbers can be obtained through static timing analysis with vendor timing analysis tools, which are able to report delays for specific paths.
5. Include the obtained area and delay figures in the architecture file arch.xml that is used by the EDA tools for application mapping and DSE. This is done by the architecture file generator within the *V-FPGA Explorer* tool, that is also able to derive required macro characteristics (e.g. for large MUXs, CLBs, IOBs) from MSBEs and architectural parameters.

7.4. Conclusion

The generic structure of the *V-FPGA* with standard VHDL code makes it possible to implement it either as virtual FPGA mapped onto a COTS FPGA or as embedded FPGA mapped onto an ASIC process. Both ways including the necessary steps were discussed in this chapter.

Virtualization adds another level of flexibility because the *V-FPGA* can be altered anytime and also new features such as dynamic reconfiguration, JIT compilation, etc. can be added to the system even though the underlying platform doesn't support it natively. This flexibility, however, comes at the price of high area overhead. This is not only for the *V-FPGA* the case, but also for all other virtual FPGAs. Compared to related works, following their definition of overhead and tuning the *V-FPGA* at a similar complexity for a fair comparison, the *V-FPGA* has a 2.8 times lower overhead than the virtual FPGA of

Lysecky et al. and 2.4 times higher than the ZUMA architecture. However, the ZUMA architecture uses platform-exclusive resources that prohibit direct portability onto other platforms, yet portability is one of the main aspects of virtualization. Furthermore, in this comparison we did not consider the application specific parameter tuning capabilities of the *V-FPGA* to reduce area. Based on benchmark results it is expected that the *V-FPGA* improves its area efficiency for individual applications through parameter tuning. This is not possible with related works as they have a fixed architecture that can't be customized towards applications. In general in virtual FPGAs the virtualization overhead is 2-3 orders of magnitude. This high overhead can be compensated only by dynamic reconfiguration that reuses the area dynamically by mapping an equal number of temporal exclusive applications. Nevertheless, area efficiency was never the purpose of virtualization but rather its flexibility and independence are its values. The *V-FPGA* fulfils all the 5 aspects of virtualization formulated in Section 1.3: portability, partial and dynamic reconfiguration, adaptivity, prototyping and emulation, accessibility.

The virtualization overhead is eliminated through physical implementation by mapping the *V-FPGA* onto an ASIC at the penalty of losing adaptivity of the architecture. A hierarchical bottom-up standard-cell approach with all the necessary steps was described in this chapter, whereby first the 9 different recurring tiles of the *V-FPGA* architecture were implemented as macros and then multiple instances of these macros were instantiated and aligned multiple times in a top-level design. The standard tool flows for physical design were extended by custom scripts for automated sizing of tile abstracts and unified alignment of IOs in a way that adjacent tiles can be attached to each other with minimum wire utilization. Experiments have shown that due to these sizing and port alignment strategies even the auto-placer manages to place instances of the tiles to an almost perfectly aligned array without the need of relative placement constraints. For demonstration purposes a *V-FPGA* with the parameters $K = 4$, $N = 5$, $W = 23$, $X = 18$ and $Y = 18$ was implemented on a 45 nm standard cell approach with a target frequency of 500 MHz. The resulting overall area for this 19 x 19 tiles array is only around 3.9 mm^2. Such custom *V-FPGA* arrays can be embedded and integrated with other modules to form application specific reconfigurable SoCs targeting certain application classes. Thereby, this standard-cell soft-IP approach has the advantage that the *V-FPGA* architecture can be customized by more than 20 parameters to meet the application's demands and maximize mapping efficiency resulting in area efficiency. None of the related works offers such a flexibility.

8. Use Cases

The following subsections present a selection of use cases with the *V-FPGA* employed. Of course, the list of possible use cases would be very large due to the customization approach, yet here only the ones that are peer-reviewed and published in [50], [35], [36], [90], [98] and [95] are mentioned.

8.1. Industrial Process Automation

The low-power domain is characterized by well-defined tight requirements for area, current and power. Furthermore, especially in the process automation field, real-time requirements need to be met and a violation of the response time or the maximum current rating can lead to severe consequences, especially in explosive environments. The process automation industry is interested in the use of low-power FPGAs to realize reconfigurable heterogeneous SoCs in intelligent industrial sensors as a power- and area-efficient 1-chip alternative to classical solutions with microcontrollers and additional ICs.

A research project with an industrial partner identified flash-based low-power FPGAs from Actel's ProASIC3 or Igloo families as suitable devices from a power and area perspective for their application. However, these devices lack the desired feature of dynamic reconfiguration, which they don't support natively. Section 8.1.1 show-cases how this limitation is cancelled by employing the dynamic reconfigurable *V-FPGA* architecture as a virtual layer on the Actel ProASIC3 FPGA.

The lowest-power flash-based Actel devices are comparably small in terms of programmable resources that they offer. In Section 8.1.2 the area efficient implementation of a complex measuring chain is achieved by employing the *ViSA* architecture as overlay to reduce control overhead compared to classical FSM implementations. Remarkably, *ViSA* reduces the resource utilization from 172.7% to 62.5%, thus it enables the use of a tiny Actel Igloo Nano device as platform for the measuring chain which was not possible before *ViSA* due to severe overutilization.

8.1.1. Enabling Dynamic Reconfiguration on Flash-Based Low-Power FPGA

A heterogeneous multicore platform greatly profits power constrained applications. Depending upon the application requirement and the available hardware blocks of the multicore platform, the application can be partitioned into modules to enable optimized mapping and parallel processing on different cores, thereby leading to lower power consumption. Though heterogeneity poses challenges to developers in the sense of having deeper knowledge about each specific hardware architecture on the chip, the huge design space

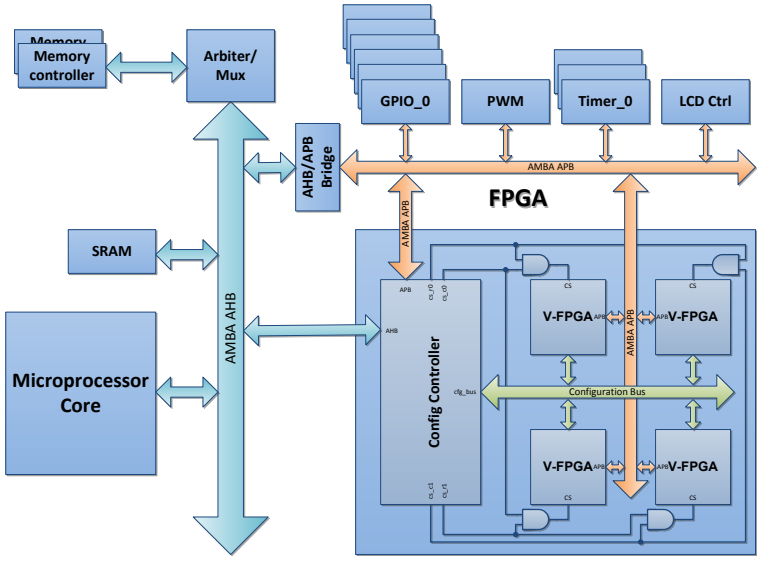


Figure 8.1.: Schematic view of the heterogenous SoC architecture with embedded *V-FPGA* cores

in turn can be utilized to map each partitioned-application-module onto a specific hardware block/architecture which can meet the precise requirements (e.g., parallelism, performance) of each application module. Custom virtual FPGA architectures due to their catering of dynamic reconfigurability even in off-the-self FPGAs, where this feature is not available, and their independency to the underlying physical platform makes them a good candidate for being embedded in heterogeneous hardware platforms. With their use, features which are application specific like special functional and I/O blocks can be exploited. This is exploited in [50].

The schematic view of such a system-on-chip architecture with *V-FPGA* cores is shown in Figure 8.1.

Thereby the SoC template type A, which is described in Section 6.1, is adopted. The system design is composed of the following blocks based on the customization flow described in Section 6.2:

- A microprocessor core: It is best suited for control oriented tasks and interfacing. It copies the configuration file from an external flash memory onto the internal RAM of the configuration controller.
- One or more *V-FPGA* cores: It accelerates computation by parallel data processing. Each *V-FPGA* core is an abstraction of the underlying physical FPGA realized by its logic cells and runs applications described in VHDL or Verilog. *V-FPGA* being used as an abstraction layer is of great benefit as the system design does not require any additional changes even when it comes down to change of physical FPGA to

ASIC. Additionally, *V-FPGA* cores promote dynamic run-time configuration even though such a feature is not supported on the physical FPGA. For example, in the case of Actel FPGA, their flash or antifuse based technique prohibits run-time configuration as their configuration memory cannot be reconfigured during run-time. Despite such a restriction, when integrated with virtual FPGAs such a feature is made available to all the applications mapped onto the core. Thereby, virtual cores enable efficient dynamic and partial reconfiguration.

- Configuration controller: The microprocessor specifies the configuration file which needs to be loaded into the respective virtual core. The configuration controller then accesses the memory controller to fetch the specific configuration data.
- Memory controller: It accesses the configuration file from an external non-volatile memory.
- Advanced Microcontroller Bus Architecture (AMBA) busses: The on-chip communications are realized using AMBA busses. The Microprocessor accesses virtual cores through AMBA Advanced Peripheral Bus (APB) bus and communicates with the configuration controller and memory controller through AMBA Advanced High-performance Bus (AHB) bus. Since both the microprocessor and configuration controller require access to the memory controller, a bus arbiter is implemented. Furthermore, a common configuration bus for all virtual cores enables transfer of configuration data from configuration controller to the respective virtual core. In order to configure each core separately, chip select signals are enforced.

Another aspect that needs to be examined is the size of virtual cores. It is of great importance as each application requires different amount of CLBs. Having a core size big enough to accommodate the largest application will lead to inefficient utilization of core area. Hence, a technique called *CoreFusion* is introduced in [35]. Initially, a relatively small core size is chosen and depending upon the CLB demand of each application the core can be merged with adjacent cores in order to fit the application.

As the virtual FPGA is provided on Register Transfer Level (RTL) as pure VHDL code and does not require any proprietary hardware macros, it is realized using the logic cells of Actel ProASIC3 M1A3P1000. From Table 8.1, the efficiency of virtual FPGAs can be viewed as two sides of a coin. On one side, virtual FPGA allocates far more physical logic cells than when an application has been directly mapped onto the physical FPGA. On the other side of the coin, beneficiary features like dynamic reconfiguration and reuse of chip area for temporal exclusive functions can be embraced even for physical FPGAs which do not natively support such vital features. Altogether, the more the amount of temporal exclusive applications that need to be mapped, the higher raises the efficiency of virtual FPGAs.

In this use-case, the *V-FPGA* core accelerates tasks such as CRC-check, memory tests, etc. that fit in a 5x5 *V-FPGA* core. Figure 8.2 breaks down the module level resource utilization in the hetogeneous SoC. On the left side the system consists of an ARM Cortex M1 soft-core processor with debug interface, dedicated hardware blocks, and the *V-FPGA*. On the right side, the resource hungry debug interface is removed from the ARM Cortex M1 core and instead a custom DSP core is integrated. In both cases the SoC fits in the Actel M1-enabled ProASIC3 M1A3P1000 COTS FPGA. The 5x5 *V-FPGA* occupies 27% of the

Table 8.1.: Utilization of physical logic cells on Actel's ProASIC3 for implementing a *V-FPGA* core with $K = 4$, $W = 4$ and various core sizes [50]

***V-FPGA* Size**	**Cells or Tiles**			**Utilization M1A3P1000**
	Seq.	**Comb.**	**total**	
2x2	480	643	1123	4.57%
3x3	902	1401	2303	9.37%
4x4	1456	2354	3810	15.50%
5x5	2142	3564	5706	23.22%
6x6	2960	5055	8015	32.61%
7x7	3910	6796	10706	43.56%
8x8	4992	8740	13732	55.88%
9x9	6206	10957	17163	69.84%
10x10	7552	13510	21062	85.70%

available resources, which is around 4% more than in Table 8.1. The differences come from the peripheral unit that is not included in the table, yet accounted for in the figure.

The simulation in Figure 8.3 demonstrates the dynamic recofiguration of a *V-FPGA* on a flash-based Actel ProASIC3 device. Here the sample application is a CRC (Cyclic Redundancy Check) algorithm, which is implemented on the custom *V-FPGA* architecture utilizing the application mapping flow described in Chapter 5. The dynamic reconfiguration process takes around 192 clock cycles. After that the application is active and calculates the correct values.

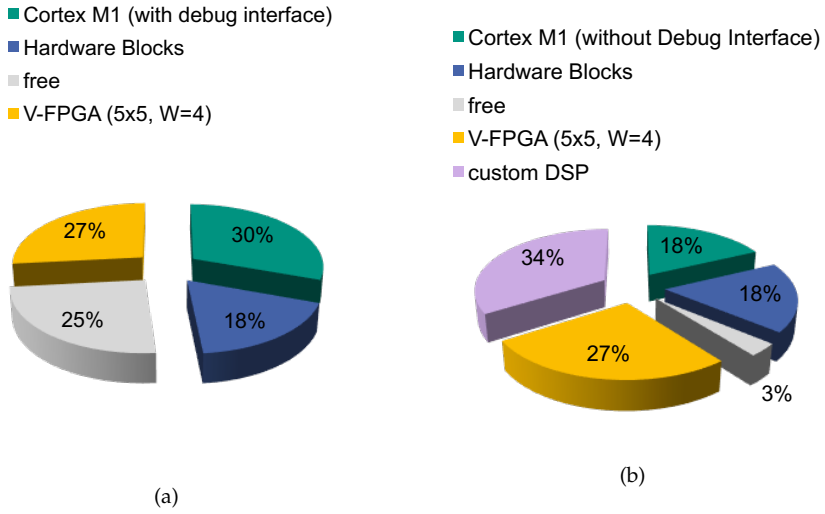


Figure 8.2.: Resource utilization for implementing the hetorogeneous reconfigurable SoC on an Actel M1A3P1000 device: (a) SoC with microprocessor, debug unit and *V-FPGA*, (b) SoC with microprocessor, *V-FPGA* and custom DSP

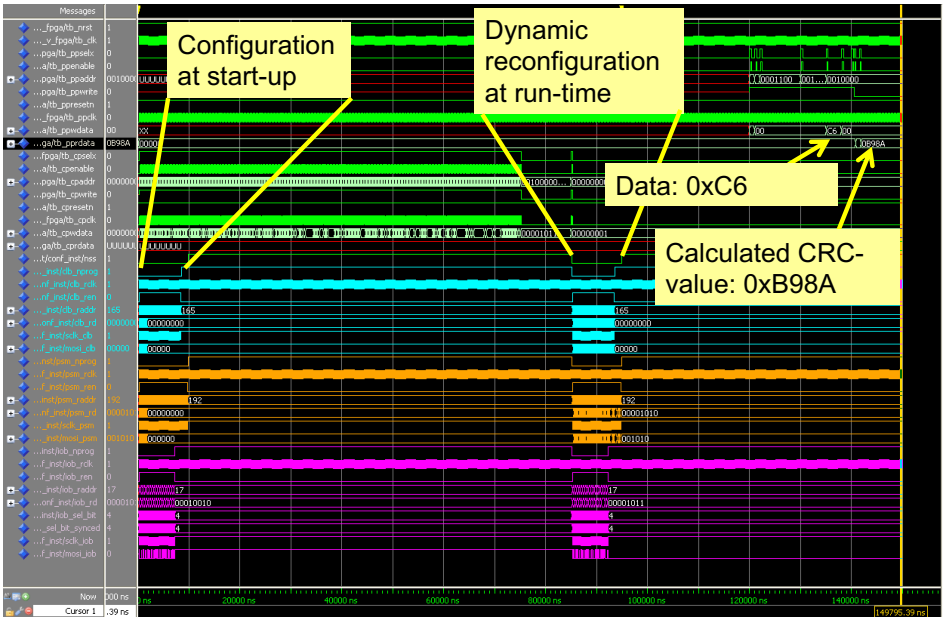


Figure 8.3.: Simulation of dynamic reconfiguration of a *V-FPGA* core that is hosted by an Actel ProASIC3 FPGA

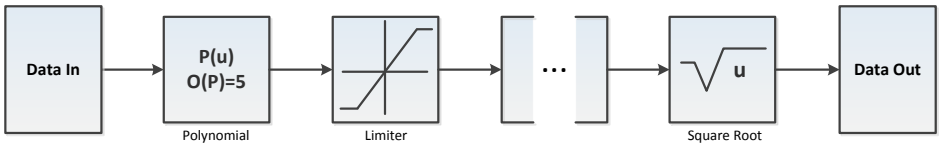


Figure 8.4.: Measuring chain with exemplary function blocks like polynomial, limiter and square root functions

8.1.2. Enhancing Efficiency by Employing the *ViSA* Architecture

In [36] we examine a real-world application scenario from where a complex measurement data processing chain needs to be implemented in a device with a foot print of $5x5\,mm^2$. The power budget is below $1\,mW$ and the real-time requirement is given by a data rate of $100\,samples/s$. The functional requirements include a sequence of several processing blocks like non-linear transfer function, polynomial solver or square root. An exemplary extract of this chain is depicted in Figure 8.4. Furthermore, the device should be able to serve additionally as level shifter and as interface to industrial busses.

These requirements cannot be met completely by available off-the-shelf microcontrollers and DSPs. ASICs are too expensive for the required volume, therefore we pick the ultra-low power Microsemi Igloo Nano AGLN250 FPGA as target device. The limitation of only 6144 cells (logic or D-Flip-Flops) is very challenging for mapping the required functionality on this device. A $24x24\,bit$ multiplier already utilizes around 34 % of the logic cells, hence resource sharing needs to be applied extensively.

In a typical FSM based implementation the resource utilization is 172.7 % and consequently the design does not fit inside the selected target device. Due to the area and power limitations, it is not possible to use another FPGA target device. This is with resource sharing applied that is managed by a lightweight round robin scheduler. Therefore, in contrast to the FSM based alternative, we investigate the *ViSA* approach presented in Section 4.4, which allows a more efficient implementation. Applying the hardware/software co-design methodology introduced in Section 6.2, a microarchitecture, as depicted in Figure 8.5, is generated, which is specifically tailored towards the application needs with focus on area minimization.

The basic function units are a multiply and accumulate (MAC) unit, a logic unit (shift left, shift right, AND, OR, XOR, NOT), a comparator and a load/store unit. The MAC unit is realized by switching the output of the multiplier directly to the input of the adder when requested. Especially the polynomial function, the transfer function and the square root benefit from multiplying and accumulating in one clock cycle. A comparator is used to enable conditional jumps and is only connected with the program counter (PC). This saves resources for the input multiplexers on the other units.

In a FSM based realization, the same application utilized 172.7 % of the available logic at the same data widths. This over utilization resulted after applying manual resource sharing, e.g., only one multiplier in the whole design, and several time-consuming optimization iterations. In contrast with the proposed methodology implemented on the

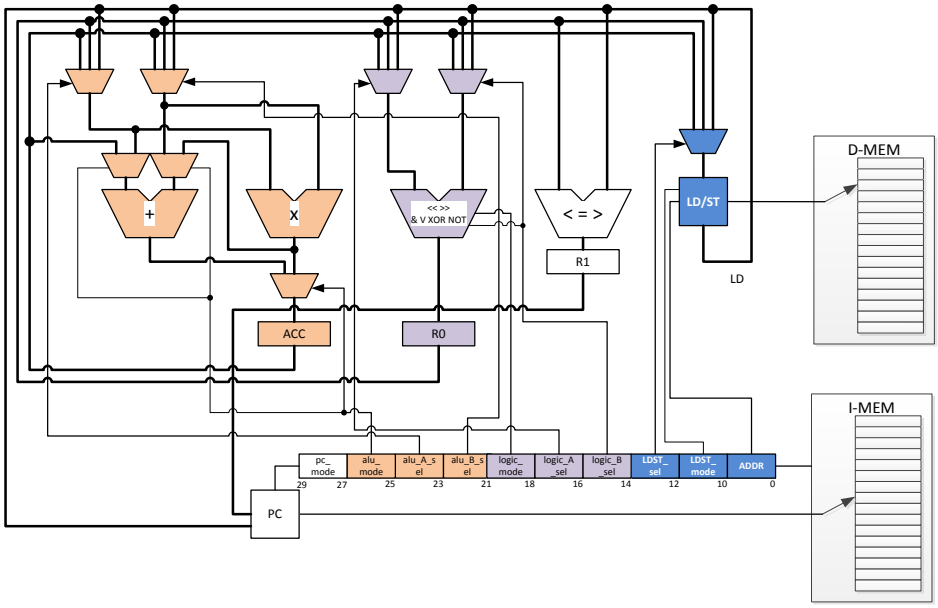


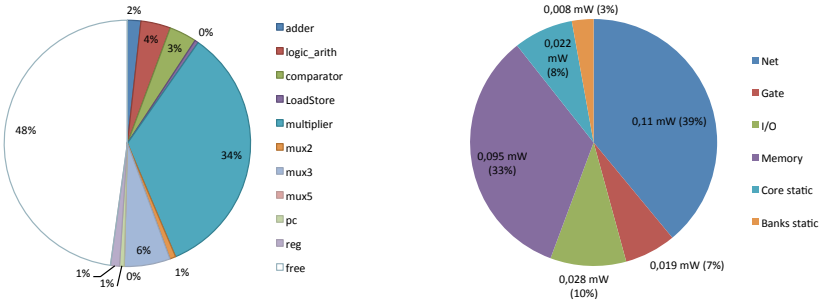
Figure 8.5.: Ultra low power VLIW-inspired Slot Architecture for measuring chains in process automation implementing a program counter, a MAC unit, a logic and a load/store slot

Actel Igloo Nano AGLN250, the specific VLIW-inspired Slot Architecture occupies only 52 % of VersaTiles[1] and 62.5 % of block RAMs. Figure 8.6a shows the composition of the resource utilization on module level for 24 *bit* data width. Since the Actel Igloo Nano does not contain dedicated hardware multipliers, it occupies 34 % of the available logic and is the largest part of the design. The 3 : 1 multiplexers contribute in total with 6 % to the logic utilization. As the pie chart indicates (see Figure 8.6a), other components are negligible in terms of area. Furthermore, maintainability is much better in the *ViSA* approach, since changes in the algorithms can be applied much faster enabling faster verification with the built in tracing.

The execution time for the processing of new input data is 90 clock cycles. Having a data rate of 100 $samples/s$ at the inputs, the minimum required clock frequency for the VLIW-inspired Slot Architecture is 9 kHz.

As shown in Figure 8.6b, the Actel SmartPower tool estimates a total power consumption of only 251 μW at 10 kHz clock frequency for the implemented *ViSA* core, while the most switching contribution is by nets and memory. Based on these results, we can conclude that *ViSA* is a highly efficient implementation for ultra-low power applications requiring a small footprint.

[1] A VersaTile can be either a flip-flop or a 3-input LUT equivalent



(a) Relative module level resource utilization on Actel Igloo Nano. The multiplier uses most of the VersaTiles.

(b) Estimated power consumption of *ViSA* on Actel Igloo Nano at 10 kHz with the Actel SmartPower tool.

Figure 8.6.: Resource utilization and power consumption of *ViSA* in measurement processing chain

8.2. 3D Ultrasound Computer Tomography

While Section 8.1 showed examples for employing *V-FPGA* and *ViSA* in low-power applications, this section shows the opposite corner case and deals with high-performance medical imaging, where a customized many-core *ViSA* architecture - as a virtualization of resources of an underlying Xilinx Virtex-6 FPGA - efficiently concentrates on the demanding highly parallel arithmetic computation. The application targets at detecting breast cancer with a 3D Ultrasound Computer Tomography (3D USCT) device as shown in Figure 8.7.

Ultrasound based medical imaging in 3D is characterized by a huge amount of data that needs to be processed. In turn it allows a high degree of parallelization, which makes it very suitable for GPUs and FPGAs. The basic principle of ultrasound based imaging in 3D is to use a 3D arrangement of ultrasound emitters and receivers surrounding the object under test. The emitters sequentially send ultrasonic wave fronts, while all receivers record the amplitude variation over time (A-Scan), including the transmitted signal as well as its delayed echo scattered by the object under test. In case of [12], depending on the resolution, 20 GB or more of raw measurement data is acquired that needs to be processed in order to reconstruct one 3D image. This amount of data is not taking into account the additional internally produced data within the calculations. On an Intel Core i7 CPU running at 2.67 GHz, the computation for a 3D image with the resolution $1024x1024x64$ takes around 15 hours [12].

The most compute intensive part of the image reconstruction is the Synthetic Aperture Focusing Technique (SAFT) algorithm [30]. Adapted for 3D, the algorithm is described by Equation 8.1 where $P(x,y,z)$ is the position of the investigated voxel, $E(x,y,z)$ the position of the ultrasound emitter, $R(x,y,z)$ the position of the ultrasound receiver, v_s the

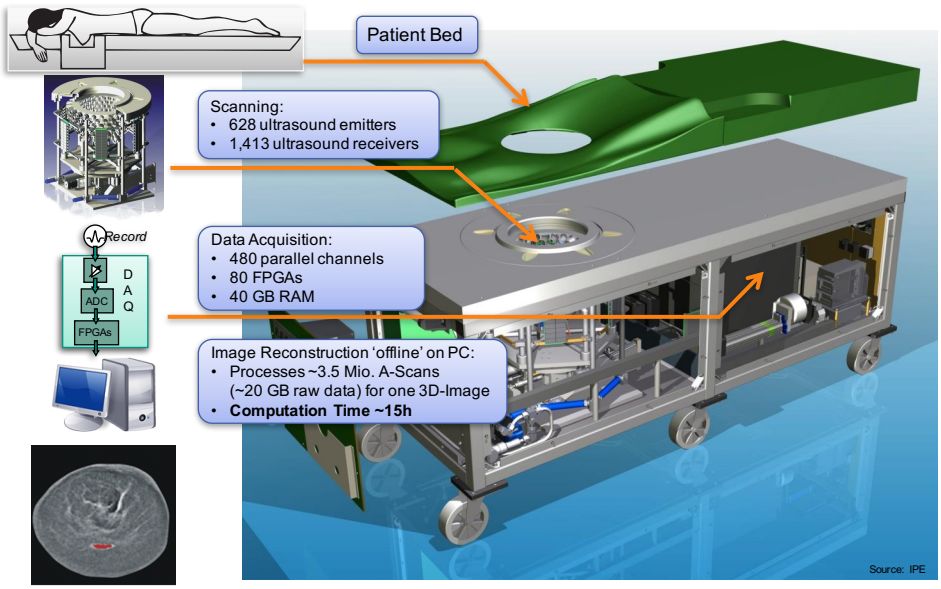


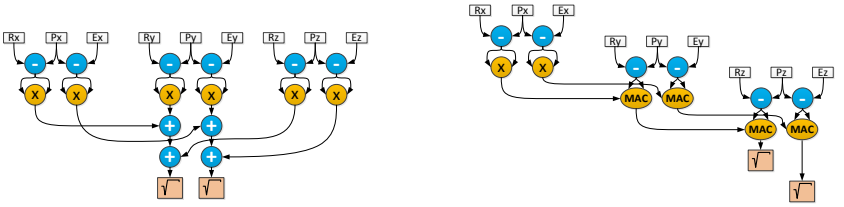
Figure 8.7.: Setup of 3D Ultrasound Computer Tomography for breast cancer detection

speed of sound, and $A(t)$ the value of the according A-Scan at the determined point in time t.

$$
\begin{aligned}
I(\vec{P}) &= \sum_{\forall(j,k)} A(t = (d_j + d_k) * \frac{1}{v_s}) \\
d_j &= \sqrt{(P_x - E_x)^2 + (P_y - E_y)^2 + (P_z - E_z)^2} \\
d_k &= \sqrt{(P_x - R_x)^2 + (P_y - R_y)^2 + (P_z - R_z)^2}
\end{aligned}
\tag{8.1}
$$

The result is an intensity of the investigated voxel, depending on how much this voxel is scattering the emitted wave fronts. If this voxel was located on a hard surface, the intensity would be high. If the voxel was located on a soft surface, the intensity would be low. This algorithm is applied for all voxels within the region of interest, giving a 3D contrast image in the end.

The SAFT algorithm is highly parallelizable, such that many voxels can be investigated at the same time. Additionally, the calculations per voxel can be parallelized as well. Thus, it is a good benchmark application for demonstrating the high performance capabilities of *ViSA*. A large Xilinx Virtex-6 (XC6VSX475T) is used as the target hardware platform. The strategy is to have many *ViSA* cores running in parallel on the FPGA, while each *ViSA* core has a reasonable configuration of parallel functional units. The reasons for multi-core realizations rather than one big single core are on the one hand a better scalability by the means of exploiting the parallelism of application. Furthermore, the complexity can be reduced by a divide & conquer strategy. This also enhances the place & route results on the target platform. Since smaller cores also have smaller instruction words, the per-



(a) Unscheduled SAFT data flow graph

(b) Scheduled SAFT data flow graph with efficient use of resources

Figure 8.8.: Dataflow graphs for SAFT algorithm

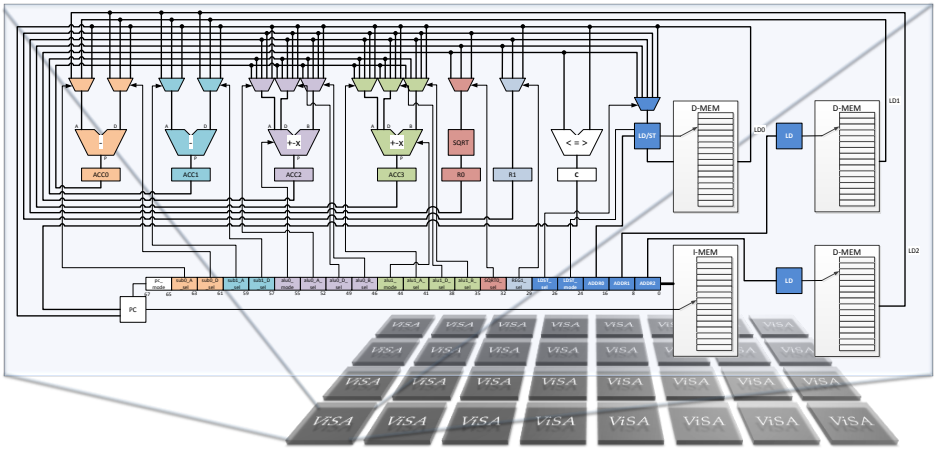


Figure 8.9.: VLIW-inspired Slot Architecture for high performance medical imaging implementing a program counter, several arithmetic and logic slots as well as a load/store slot. *ViSA* is realized as a homogeneous multicore system.

formance of the memory system is improved because in contrast to a big single core no extensive cascading of distributed native BRAMs is required, shortening the delays and consequently the critical path.

The data flow graph for the algorithm is set up and analyzed to find an efficient scaling (see Figure 8.8a). The *ViSA* core can access three memories concurrently, each for one of the coordinates $\vec{P}$, $\vec{E}$, $\vec{R}$. With this boundary, a schedule for the tasks is found, such that all required units are highly utilized, thus having a high efficiency (see Figure 8.8b). Since the positions of emitters and receivers are fixed, ROMs can be used instead of RAMs for the memories, saving additional logic and especially routing resources. From this analysis, our tailored *ViSA* core needs one Load/Store Unit, two Load Units, two subtracters, two ALUs (including MAC functionality within one clock cycle), one square root unit and one delay register.

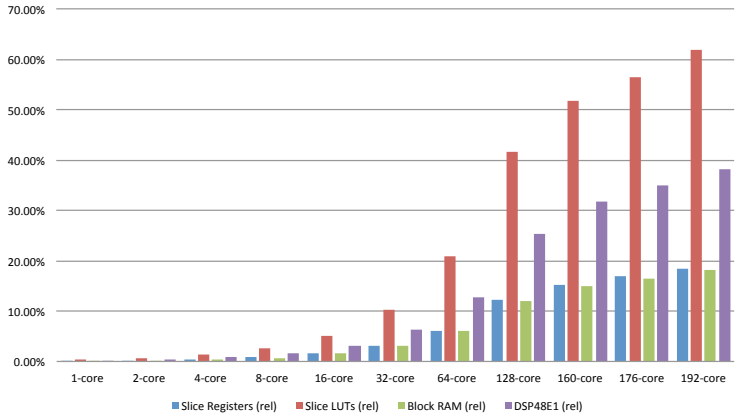


Figure 8.10.: Resource utilization of *ViSA* manycore on Xilinx Virtex-6

Accordingly, the structure of the *ViSA* core is set up as depicted in Figure 8.9. In the implementation of the VLIW-inspired Slot Architecture on Virtex-6, the integrated DSP48E1 blocks are utilized for realizing the subtracters and the ALUs. CORDIC soft cores from the Xilinx IP catalog are instantiated for implementing the square root units. The integrated block RAMs realize all memories. A light-weight linear NoC (Network-on-Chip) enables global access to the local data RAMs in case of multicore configurations. Therefore every core has a NoC interface, which incorporates packet handling, parsing and DMA (direct memory access).

The VLIW-inspired Slot Architecture is implemented in various multicore configurations at 18 *bit* data width on the Xilinx Virtex-6 (XC6VSX475T) target platform as shown in Figure 8.9. The resulting resource utilization is examined as well as the achievable peak performance for executing the SAFT algorithm. As metric for performance $VoxelAScans/s$ is used as defined in [16]. This is the number of processed AScans per second for reconstructing voxel intensities. Figure 8.10 shows the resource utilization in relation to the number of *ViSA* cores.

Since the resource utilization is straightly proportional to the number of cores, this indicates that the ViSA multicore approach is efficiently scalable in terms of area. A more detailed view of the relative module level utilization of a *ViSA* core in Figure 8.11 shows that the main part of the utilization is by the square root unit, which is implemented by the CORDIC soft core from the Xilinx IP library, and by the NoC interface. The other functional units cause very little utilization of LUTs and registers, since they are realized by the integrated DSP48E1 blocks.

Figure 8.12 shows the maximum operating frequency and the obtainable peak performance for various multicore configurations after we applied loop optimisations in the program flow. The maximum achievable operation frequency is around 230 MHz for the single-core and around 220 MHz for configurations with up to 8 cores, but decreasing slightly for larger manycore configurations. There are some stronger variations for

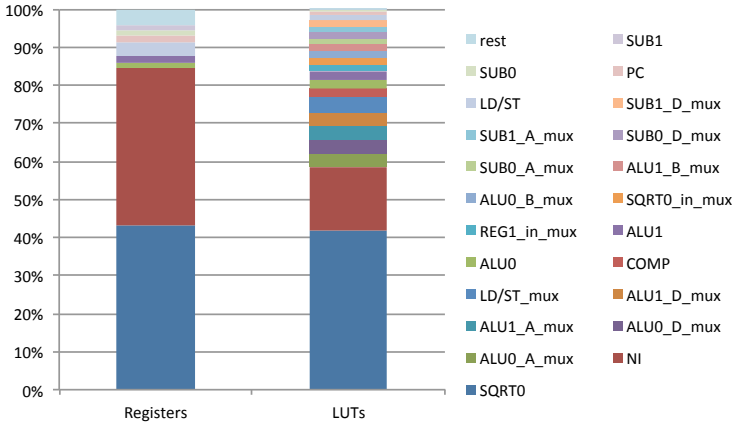


Figure 8.11.: Relative module level resource utilization on Xilinx Virtex-6. The square root unit (SQRT0) is the unit with the most resources, followed by the NoC interface (NI).

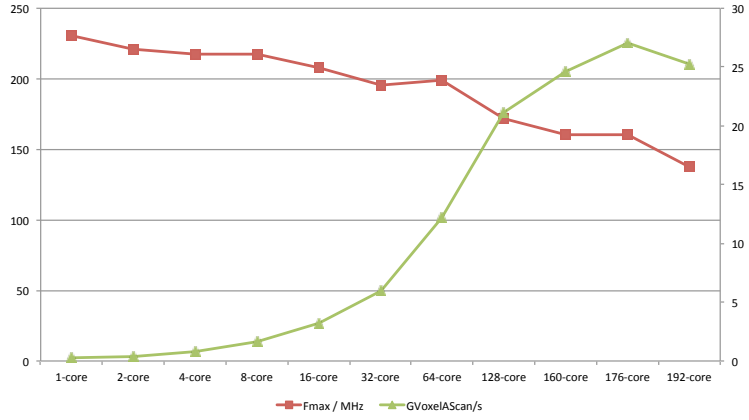


Figure 8.12.: Maximum frequency and throughput of *ViSA* manycore on Xilinx Virtex-6. The optimum configuration contains 176 *ViSA* cores at a maximum frequency of 160.4 MHz, which gives a peak performance of up to 27.0 $GigaVoxelAScans/s$.

larger configurations, which may arise from low place & route tool optimization efforts. However, due to parallelism the achievable performance increases significantly with the number of cores up to 27.0 $GigaVoxelAScans/s$ for the configuration with 176 *ViSA* cores operating at 160.4 MHz. If the number of cores is further increased, there is a slight degradation in terms of performance, e.g., with 192 *ViSA* cores, only 25.2 $GigaVoxelAScans/s$ are achieved because of a significant decrease in operating frequency to 137.1 MHz due to high resource utilization on the Virtex-6 FPGA and place and route problems caused by Xilinx ISE. From the timing reports, we realize that the critical paths limiting the maximum operating frequency are in the memory structure. It is expected that the performance will increase beyond this point if Virtex-7 FPGA is used in conjunction with Xilinx Vivado.

Using the presented design space exploration, we conclude that the optimum configuration contains 176 parallel *ViSA* cores. Furthermore, we still have a huge amount of free resources (82.9 % slice registers, 43.4 % slice LUTs, 83.4 % BlockRAMs and 65.0 % DSP48E1s) available for other tasks or for incorporating a *V-FPGA* region.

8.3. Dynamic Platform-Independant Application Mapping

The *V-FPGA* was employed in [90] to enable dynamic platform-independent application mapping through virtualization.

FPGAs with their scalability and power efficiency though promise an alternative solution for implementing parallel applications, the application complexity and hardware incompatibilities make them less flexible when compared to CPUs and GPUs. Thanks to virtualization, which makes application mapping platform-independent, and thereby enables its portability across different FPGA devices without additional modification. One of the next burdensome tasks is Placement and Routing (P&R) because of its time consuming nature. Though this cumbersome step can be optimized with fast placement and routing algorithms, their inability to accommodate multiple applications concurrently is one of the greatest stumbling blocks with regard to execution time. In *V-FPGA*s, since the applications are mapped onto the virtual architecture rather than onto the physical platform, the P&R step of each application is independent of each other. Hence, *V-FPGA*s sanction the intelligence to execute parallel P&R steps for different applications and thereby significantly increases the throughput of the toolflow. The application mapping onto *V-FPGA*s follows the toolflow shown in Figure 8.13.

The toolflow is started with the assumption that knowledge about resource requirement of the future application and the available resources at future time point is unknown. When a new application requests for resources, it enters a queue system. The first block called scheduler manages this queue by sending the new tasks serially to the next tool, the CoreMapper, removes the finished tasks from the queue and also checks for the availability of resources. The CoreMapper then allocates an area on the SoC for the application to be mapped and sends this information to P&R step. For building SoCs using *V-FPGA* cores, "CoreFusion" methodology is employed, where the size of each *V-FPGA* core is not big enough to accommodate the largest application but has a smaller size which when needed depending upon the resource requirement of the application will get merged with

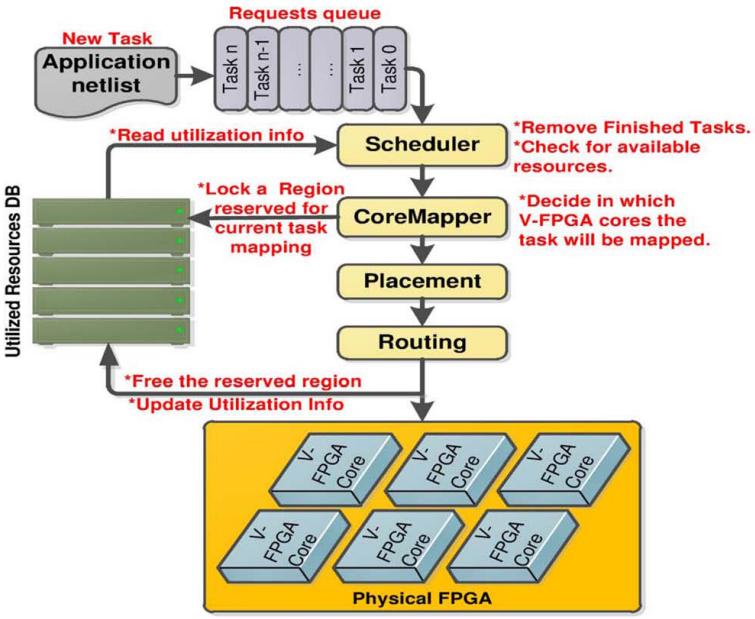


Figure 8.13.: Methodology for application mapping onto *V-FPGA* cores [90]

the adjacent cores to form a bigger core. This technique not only provides flexibility but also improves the overall resource utilization and enhances the number of applications that can run in parallel. Because of virtualization, the CoreMapper is aware of the available resources in slice level granularity and not only allocates the area accordingly but also locks that area in the resource database from usage by other applications. This tool ensures that the P&R step of each application is independent from each other. The placer tool follows a fast-simulated annealing algorithm. The initial logic block placement is optimized by swapping pairs of blocks and only certain percentage of moves is allowed. The execution time of the placer is significantly improved by: i) virtualization which through CoreMapper allocates a predefined area thereby narrowing the solution space for placement and ii) decreasing the number of moves. The router is based on the PathFinder negotiated congestion algorithm and is tuned for faster execution time. The output of P&R step is a partial bitstream file per *V-FPGA* which configures the LUTs, PSMs and I/O blocks.

The 20 biggest MCNC benchmarks are implemented by a single Intel Xenon CPU with 8 cores and 8 Gb of RAM using the aforementioned toolflow. A *V-FPGA* core of 8 x 8 CLBs is mapped unto Xilinx Virtex-7 (xc7v2000tflg 1925-2), Altera Stratix V (5SEEBF45I2) and Actel ProASIC3 (A3P15000). In order to evaluate the efficiency of the toolflow, a queue consisted of 200 circuits (from 20 MCNC benchmarks) is considered along with an array of four identical 75 x 75 *V-FPGA* cores, a total array of 150 x 150 CLBs and 50 routing tracks per channel. The flexibility to reconfigure the virtual architecture at slice level granularity and the ability to map several applications onto multiple *V-FPGA* cores will affect the

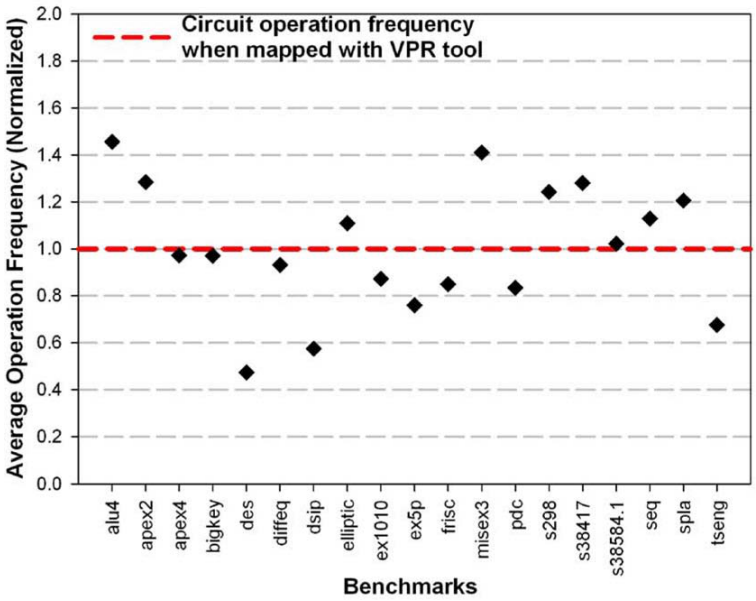


Figure 8.14.: Average operation frequency for multiple benchmark mappings with our framework, normalized to VPR solution [90]

operating frequency of each circuit. At the same time, the resource utilization and the physical position of the available resources play a major role in the final mapping step.

Each benchmark is mapped several times and their normalized average maximum frequency in comparison to the reference solution (when each application is mapped onto the FPGA by VPR tool) is depicted in Figure 8.14. The variations in results are due to the parallel execution of P&R for different applications and in average has a negligible penalty of 0.26% Fragmentation due to dynamic reallocation of resources on the four *V-FPGA* cores is shown in Figure 8.15. An average ratio of 24% is achieved with the toolflow in trade-off for high quality mapping of multiple applications.

The scalability of the toolflow is then tested on a multicore system with two, four and eight cores by measuring the number of tasks that can be mapped per minute. Up to 14 applications per minute can be mapped onto a single *V-FPGA* core, which then increases to 25 mappings per minute for dual core (1.77x), 44 for four cores (3.13x) and 72 for eight cores(5.23x). Thereby, proving that the toolflow can scale efficiently and can be a viable solution to support large heterogeneous systems with some addition of hardware resources.

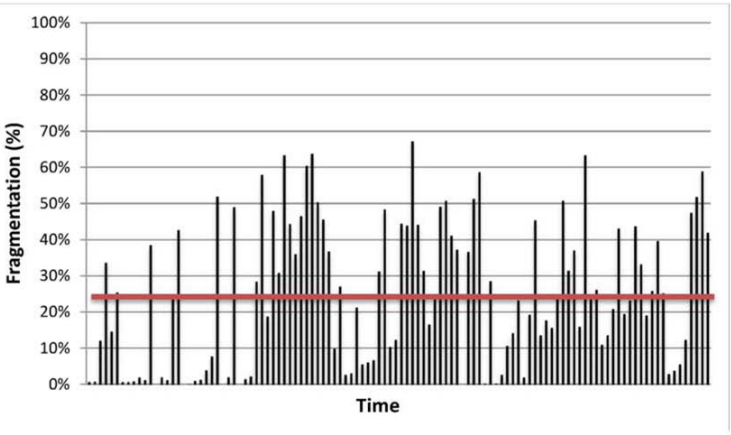


Figure 8.15.: Fragmentation ratio on the *V-FPGA* cores through multiple applications mappings [90]

8.4. Self-Aware Reconfigurable Platforms

An inherent advantage of the virtual FPGA approach is the ability to alter the architecture of the virtual layer on which an application will be mapped. In cases where the underlying physical platform supports dynamic reconfiguration, the architecture of the virtual layer can be altered even during run-time. In [98] we are exploiting this flexibility to enable self-aware reconfigurable platforms that adapt the virtual layer depending on the applications that are to be executed onto it. The motivation behind this approach is that different applications have different demands and constraints on the architectures, leading to high variations in efficiency due to parameter sensitivity as we have studied in depth in [34]. In order to mitigate this problem and improve efficiency the idea is to select and instantiate the most suitable virtual FPGA architecture for an application during runtime.

A challenge thereby is that the applications might not be known a-priori and can change. For instance, think of FPGAs in the cloud e.g. as recently deployed by Amazon [9], where it is expected that applications are not known by the host and fly in and out based on the rented compute time and area by various clients. This requires an online analysis of the applications, exploration of the supported architectural design space and selection of the most suitable architecture. However, the methodology in Section 6.4, which is primarily designed for a high accuracy, is time-consuming. Therefore, collaborating researchers from the National Technical University of Athens have developed in [98] a fast approximate methodology based on neural networks.

The target architecture platform (see Figure 8.16) in this study is similar to the low-power reconfigurable SoC platform used in Section 8.1.1. It's a SoC architecture with multiple *V-FPGA* cores, a microprocessor core, a configuration controller, on- and off-chip memories and busses for on-chip communication and configuration. The difference however is that

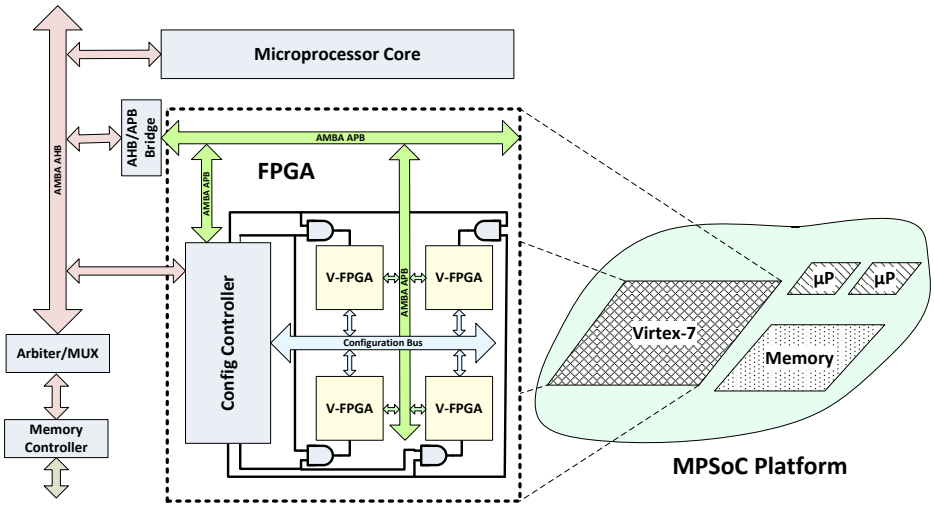


Figure 8.16.: System architecture of the self-aware platform

the *V-FPGA* cores can be altered during runtime by partial and dynamic reconfiguration of the underlying physical FPGA, whereby each core can have different properties. By appropriately tuning parameters of LUT size, cluster size, routing channel width, etc. it is not only possible to improve the mapping efficiency but also to address different objectives, e.g. area, performance, power.

The proposed methodology for performing application mapping onto self-aware reconfigurable platforms is depicted in Figure 8.17. As input to this framework the synthesized application's description is used.

Whenever a new application is requested, the proper *V-FPGA* is instantiated onto which the application is to be mapped. A profiling of the application's netlist is done to extract a number of application-oriented parameters such as number of nodes, number of edges, number of primary inputs/outputs, average and maximum fanout. The profiling results provide a first estimate about the desired architecture for the target FPGA device and are used to derive the architecture's parameters for customizing the *V-FPGA*.

State-of-the-art approaches to appropriately fuse these results (e.g. brute force, simulated annealing, genetic algorithms, etc) impose mentionable run-time overhead, which is not affordable especially for systems that have to be executed at run-time. For this purpose, our methodology incorporates a fast yet accurate multi-objective optimization technique based on neural network. While the focus of this thesis is not on neural networks, a more detailed explanation can be found in [98]. The outcome is the most suitable set of architectural parameters in terms of LUT size K, cluster size N, channel width W, etc.. These parameters are handled by our framework in order to automatically instantiate the desired *V-FPGA* template.

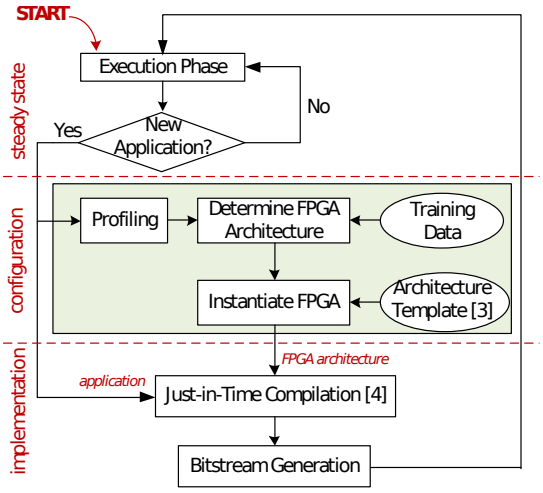
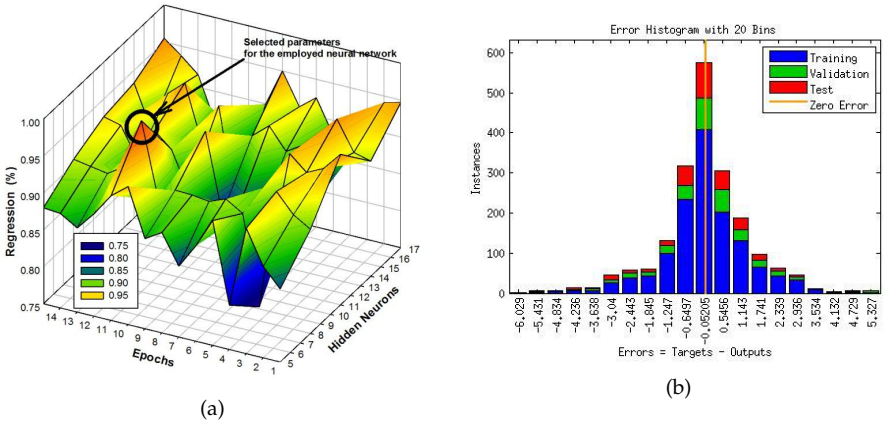


Figure 8.17.: Methodology for supporting self-aware reconfigurable platforms [98]

A critical task for deriving this optimized architecture affects the proper training of neural network with a representative set of benchmarks. For this purpose, at design-time, the employed neural network is trained with a variety of applications from different benchmark suites (MCNC, LGSynth93, QUIP). Hence, we can almost safely guarantee that the architectural properties derived from our solution are close enough to the optimum device FPGA.

Then, we perform technology mapping, placement and routing (P&R) with the usage of the JIT compilation framework [92]. The outcome from JIT framework contains the partial bitstream file for the new task and the resource map where this task has to be allocated. Finally, the computed bitstream file configures the *V-FPGA* with the new task.

In [98] the employed neural network was first modeled in Matlab, whereas after determining the optimal parameters for training (e.g. number of neurons), the network was also developed in C++ to allow integration of the developed network into the MEANDER design framework. A critical parameter that affects the efficiency of the designed neural network is the regression. Figure 8.18a summarizes the metrics of this parameter, as we vary the number of hidden neurons and epochs (determine the maximum number of iterations for training). Another important parameter that quantifies the efficiency of the derived neural network affects its error. This parameter is computed by finding the error between the network's output and the target value over all the example pairs (targets - outputs). The output of this analysis is summarized in Figure 8.18b. Specifically, this figure plots the error histogram for the selected neural network. The red color lines denote the optimal solution retrieved after brute-force exploration.



(a) (b)

Figure 8.18.: Experimental results: (a) Exploration results for designing neural network, (b) Error histogram for the employed neural network [98]

Due to the neural networks a sufficient tuning of architectural parameters for delay, power and area metrics can be achieved during run-time in order to adapt the *V-FPGA* in a self-aware reconfigurable platform.

8.5. The TEAChER Framework

The new paradigm of application/platform co-design, where the platform (FPGA) is not limited to given COTS solutions but can be specifically tailored towards the application to be developed and deployed onto it, needs to be early addressed and made tangible also in the education of young engineers and researchers. Here virtualization plays a special role for accessibility and experimentation of custom architectures. The *V-FPGA* has been employed in an educational pilot project called TEAChER (TEach AdvanCEd Reconfigurable architectures and tools) [95].

This collaborative project between Karlsruhe Institute of Technology (KIT) and National Technical University of Athens (NTUA) was a happenstance to birth new pioneering ideas in emerging technologies like 3-D integration, optical on-chip interconnects and system virtualization with the continuing expansion of reconfigurability in FPGAs and architecture exploration being the key factor. The main focus of this framework, as the name suggests, is to teach and educate the younger generation, engineers and researchers about FPGA architectures and how to reconfigure, compile and synthesize them by using a user-friendly virtual lab, which to some extent even allows the students to work remotely. This transfer of knowledge is carried out through various means of teaching techniques like providing educational materials for programming and designing reconfigurable architectures, conducting summer schools to give lessons about state-of-the-art reconfigurable architectures and CAD tools, incorporating these advanced topics in university lessons, catering for an online portal to allow students to work remotely with the experiments and to disseminate the developed tools to the community of reconfigurable architectures. The TEAChER framework addresses the application oriented topics as displayed in Figure 8.19. The first step being the architectural-level exploration is an important step to identify the key features like memory requirements, demand for high-speed connectivity as well as the need for complex arithmetic operations specific to the application. This initial determination lays down the sketch of the reconfigurable architecture and the analyzed parameters are included in the generic HDL-level architectural files. For example, for 2-D FPGAs, the VHDL description of the Virtual FPGA (*V-FPGA*) is mapped onto the existing off-the-shelf platform provided by vendors like Altera or Xilinx and the application being mapped onto the *V-FPGA* using the extended version of 2-D MEANDER flow. In case of 3-D FPGAs, due to the constraints caused by die stacking, the *V-FPGA* is tuned accordingly by taking into consideration the existence of vertical interconnects. The *V-FPGA* is then mapped onto the physical platform which is aware of 3-D integration (for example Virtex-7) and the application on top of the virtual layer as similar to that of 2-D mapping.

8.5.1. 3-D Reconfigurable Architectures

The era of Moore's law is slowly coming to an end as shrinking the transistor size to accommodate more of them in a smaller area has led to frequency wall limitation and thereby imposing a struggle of not being able to achieve an operating frequency of more than 1 GHz in FPGAs. This set back in performance emphasis the criticality of having alternate solutions and optimized methodologies. In this context, 3-D architectures prove

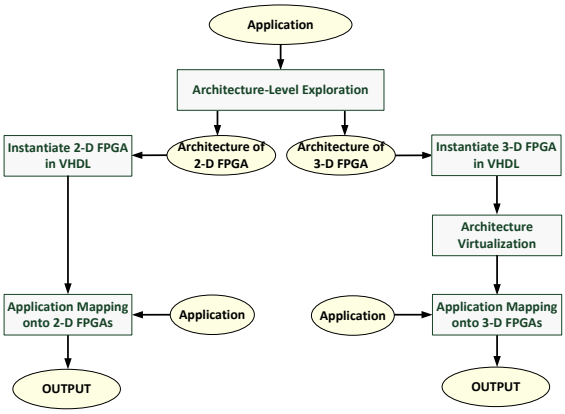


Figure 8.19.: Flow diagram for the topics addressed during the TEAChER project [95]

to be an emerging field with promising outcomes to deal with the aforementioned challenges. Though stacking multiple dies in vertical axis using Through Silicon Vias (TSVs) guarantees reduced signal propagation delay due to shorter interconnects, complexity in dealing with the 3rd dimension along with the need for having new software tools slow down the growing pace of 3-D FPGAs, which is being addressed by TEAChER framework by using virtual FPGAs as virtualization of the hardware platform (processing cores, memories, interconnects etc.,) in the form of a simulator makes the architectural layer fully customizable depending upon the application requirement.

8.5.2. Virtual FPGAs

The virtual FPGA employed in the TEAChER framework is an island style topology with the CLBs being surrounded by routing channels, which are being interconnected through Programmable Switch Matrices (PSMs). Architectural details can be read in Chapter 4. As shown in Figure 8.16, the *V-FPGA* implemented on top of the physical Virtex-7 FPGA platform acts as the reconfigurable architecture for the application, which is mapped on top of the *V-FPGA* and is thereby independent of the physical platform. In this fashion, virtualization substantiates not only partial reconfiguration at slice level but also breaks down the complexity of interconnects in 3-D FPGAs. In TEAChER framework, *V-FPGA* enables students to play around with architectural features and their impact on the performance of the applications. This concept has been employed so far in labs of two pilot summer schools (the first one for 2D and the second one for 3D FPGA architectures). As not all the students will be able to have access to expensive physical FPGA boards, the concept of virtual laboratories has been introduced. Any student or researcher at any time and from any place through remote infrastructure can send and receive information to and from the virtual framework as sketched in Figure 8.20. In this way executing their customized models and evaluating their experimental results are made possible without the physical hardware.

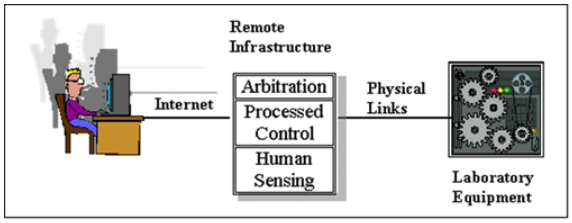


Figure 8.20.: Concept of virtual laboratories [95]

8.6. Conclusion

Peer-reviewed and published use cases for *V-FPGA* and *ViSA* in low-power applications with constricted area and current requirements, in real-time high-performance medical imaging application, in making applications platform-independent, for self-aware reconfigurable platforms and in a special framework called TEAChER were presented in this chapter.

The fast dynamic and partial reconfiguration ability of *V-FPGA* promotes dynamic runtime reconfiguration in Actel ProASIC3 device even though this option is not natively supported by the device. For a sample Cyclic Redundancy Check (CRC) application the reconfiguration time is only around 192 clock cycles and is fast enough to seamlessly switch context during runtime and thus increase efficiency by reusing of the same area for multiple temporal exclusive applications. On the other hand, the competence of *ViSA* architecture for a real-time application requiring lower power and smaller footprint of 1 mW and 5x5 mm2 respectively was investigated on a Microsemi Igloo Nano AGLN250 device which results in a resource utilization of only 52% of VersaTiles and 62.5% of block RAMs when compared to 172.7% overutilization in a classical yet already area-optimized FSM based realization. In a particular project in the field of industrial process automation, where state-of-the-art design methodologies failed to meet the area constraints, *ViSA* proved to be an enabling technology and rescued the feasibility of employing such small low-power devices for complex measurement tasks. An opposite corner case of utilizing many-core *ViSA* architecture in high-performance medical imaging application of detecting breast cancer using 3D Ultrasound Computer Tomography (3D USCT) was also studied. Its implementation on Xilinx Virtex-6 reports having 176 *ViSA* cores (each with 8-fold superscalarity) as the optimum configuration and achieves a peak performance of up to 27.0 *GigaVoxelAScans/s* at a maximum frequency of 160.4 *MHz*. *V-FPGA* cores being mapped onto three different platforms of Xilinx Virtex-7, Altera Stratix V and Actel ProASIC3 along with the 20 biggest MCNC benchmarks being dynamically mapped onto *V-FPGA* by optimized CAD tools running on a single Intel Xenon CPU not only makes application-mapping platform independent but also exploits the ability to map several applications onto multiple *V-FPGA* cores. Mapping of 14 applications per minute onto a single core can be increased to 25 mappings per minute for dual core (1.77x), 44 for four cores (3.13x) and 72 for eight cores (5.23x). One other important competence of *V-FPGA* is their flexibility to enable self-aware reconfigurable platforms where the virtual layer can be altered depending upon the application and this facility has been studied in

a collaboration with the National Technical University of Athens as a use case utilizing an accurate multi-objective optimization technique which is based on neural networks in order to reduce the run-time overhead. Finally in this chapter, *V-FPGA* being employed in an educational pilot project called TEAChER to address the growing issues of 3-D FPGAs in a direct link between research and education was presented. In two summer schools the *V-FPGA* framework was employed as an experimentation platform in hands-on labs to instantly explore and prototype custom 2D and 3D FPGA architectures without the pains of actual physical implementation.

9. Conclusion and Future Work

Driven by the needs for specialization and customization an extensive framework for the systematic generation of custom FPGA architectures and reconfigurable SoCs, that are explicitly tuned towards the application demands, and for the application mapping onto them was developed.

The overall strategy is to analyze first the application or application class and then derive values of supported architectural parameters that achieve an optimal fit of the architecture and a high mapping efficiency of the application onto it.

The center-piece thereby is the generic and highly parameterizable 2D and 3D *V-FPGA* architecture coded in technology independent VHDL. It offers more than 20 parameters that can be tuned to customize the architecture. This architecture was designed to be either virtualized onto arbitrary commercial of the shelf FPGAs or to be integrated as embedded FPGA into SoCs on an ASIC process by the means of soft-IP standard-cell approach. Both targets, the mapping and realization methodologies were described and demonstrated. Especially the physical implementation is detailed more as it is less straight-forward than the mapping onto COTS FPGAs and involves a custom hierarchical layout methodology for the *V-FPGA*.

The main purposes of virtualization in this context are portability, upgrading to partial and dynamic reconfiguration capabilities where not natively supported, adaptivity, prototyping/emulation and accessibility. All these aspects are fulfilled by the presented framework and were demonstrated in a number of use cases.
The draw-back of virtualization is its high area overhead as it is realized by programmable resources of the underlying platform. This problem is eliminated by mapping the *V-FPGA* on an ASIC when adaptivity of the architecture can be sacrificed. An exemplary physical implementation of a *V-FPGA* with the parameters K = 4, N = 5, W = 23, X = 18 and Y = 18 on a 45 nm standard cell process with a target frequency of 500 MHz resulted in an area of only 3.9 mm^2.

No matter whether the *V-FPGA* is realized as virtual layer or as physical layer, customization of its architecture improves area efficiency and/or performance. This has been shown in this work by an extensive study of effects of architectural parameters on area and performance. Therefore, more than 1400 benchmark application mapping runs with parameter sweeps were executed and analyzed. The study showed high sensitivities of the parameters LUT size and cluster size which resulted in area variances of up to $\pm$95.9% and performance variances of up to $\pm$78.1%. An even more interesting observation is that individual benchmark applications favour different parameter values, i.e. some benefit from a smaller than average case optimum LUT and/or cluster size while some others benefit from a larger than average case optimum LUT and/or cluster size. This proves the significance of customizing such parameters.

Apart form customization aspects and flexibility, the *V-FPGA* architecture comes with a number of innovations. *LoopbackPropagation* is an efficient circuit technique for emulating bi-directional wires. *CoreFusion* is a mechanism to merge two adjacent *V-FPGA* cores. Partial and dynamic reconfiguration mechanisms are supported at various granularity ranging down to the reconfiguration of an individual CLB while all others continue operation during this time. In conjunction with the new *Snapshot* mechanism, the instant migration of active applications from one region to another is supported at run-time. This has been exercised along with a new defragmentation heuristic to free contiguous reconfigurable area by moving scattered applications during runtime. Mixing CLBs with custom *ViSA* cores (a novel generic microarchitecture enabling multi-objective ASIP cores) goes one step further than related works that include simple multipliers and adders in their FPGAs, with the differences that ViSA can contain custom function units and can execute programs. The very high area efficiency and performance of *ViSA* cores has been proven in various use cases from low-power industrial process automation to high-performance medical imaging.

Customization and virtualization impose a number of challenges that were systematically addressed in this work. Design efforts are minimized by generic architecture templates that are easily customized by simply adjusting parameters at top level or in a package file. No code changes are necessary. The complex process of choosing the right design parameters is assisted by novel model based parametric design space exploration methodologies with new metrics that were introduced to capture peculiarities and effects of adding a virtual layer. To cope with the challenges of mapping applications onto custom architectures, a flexible toolflow employing and integrating existing commercial, academic and new own tools was developed that supports parameters of the *V-FPGA*. Therein the new *V-FPGA Explorer* tool closes the gap between abstract layout and actual configuration. New is also the support of JIT compilation for the *V-FPGA* that was developed in collaboration at the National Technical University of Athens. It allows to map applications 53.49x faster than the state-of-the-art, resulting around 2.6 *s* per benchmark application, which is fast enough to compile applications on demand during runtime. This is very attractive for adaptive systems and cloud services. For the employed place&route tools as well as for DSE new area and delay models based on Minimum Size Basic Elements were developed to target virtual architectures. In contrast to prior works that relied on misleading area and delay models at transistor-level that were actually intended for physical FPGAs, the new models are better suited for virtual FPGAs as they consider the correct base units of the underlying platform. The dynamic evaluation of custom FPGA architectures is solved through virtualization, without the need for physical design. This has been also demonstrated in the pilot TEAChER project, where the *V-FPGA* has been employed in labs for this purpose.

The main differentiators of the *V-FPGA* compared to the few existing works in the context of virtual and custom FPGA architectures are especially the high flexibility, customization, portability and the novel architectural features mentioned above. Furthermore, this framework is characterized by interlocked co-design of all relevant aspects, including not only architecture generation, but also tooling and customization methodologies.

The *V-FPGA* framework, due to its flexibility and modularity is a platform that can be used for further research activities.

An interesting idea is to explore very fine-grained underlying reconfigurable resources such as ambipolar transistors. In this direction we proposed the project PARFAIT which was accepted and launched recently. The idea is to exploit a novel ambipolar field-effect-transistor called DeFET for power-aware and flexible FPGA architectures. An interesting property of this transistor is its intrinsic reconfigurability through the back-gate, i.e. it is possible to change its polarity and also to tune the threshold voltage on-demand. This can be beneficial for instance in the routing infrastructure and for dynamic performance and energy optimizations through threshold-voltage-scaling. It is planned to use the *V-FPGA* as baseline architecture in this project and to extend it to make use of the intrinsic reconfigurability of the DeFET. Having already an existing modular and tool-supported platform helps concentrating on the new challenges of large scale DeFET integration and their area and energy efficient utilization through transistor level reconfiguration.

Another idea is to target the generation of power and heat aware specialized 3D-FPGA architectures. The *V-FPGA* already has 3D PSMs with parameterizable distribution of TSVs for 3D stacking. A spatiotemporal power and heat profiling through simulation of the *V-FPGA* while applications are mapped and running onto it and through extraction of the actual switching activity profile can be used to identify local hotspots. The analysis can be used to find an optimum distribution of TSVs in a way that minimizes such hotspot generation.

A. Appendix

A.1. Evaluation of MCNC Benchmarks Mapped on *V-FPGA* with Various LUT and Cluster Sizes

Figures A.1 to A.20 present resulting area and performance of the 20 largest MCNC benchmarks mapped on *V-FPGA* with all the various LUT and cluster size combinations in the ranges of $K = 2..8$ and $N = 1..10$. As underlying host platform of the *V-FPGA* the Actel ProASIC3 architecture was chosen and the *V-FPGA* accordingly characterized, upon which 70 architecture model files (needed for the mapping tool VPR) were derived. In total 1400 combinations of benchmarks on architectures were synthesized, placed and routed, which represents a rather extensive evaluation. The area is split in logic area, routing area and IO area in order to visualize how the ratios change with parameter variation. The area values are expressed in VersaTiles, that are the smallest units by which the *V-FPGA* is realized when mapped on Actel ProASIC3.

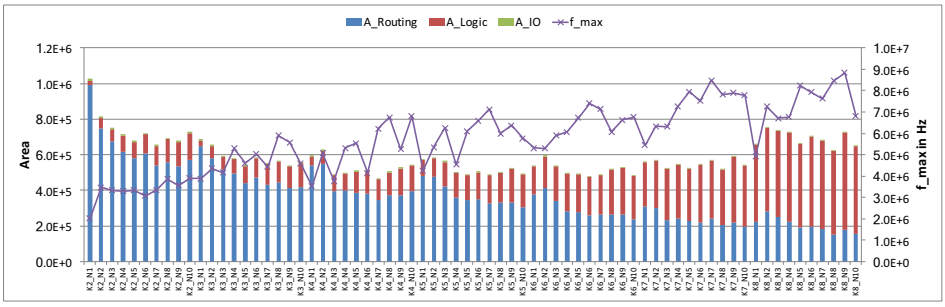


Figure A.1.: Area and performance of *alu4* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

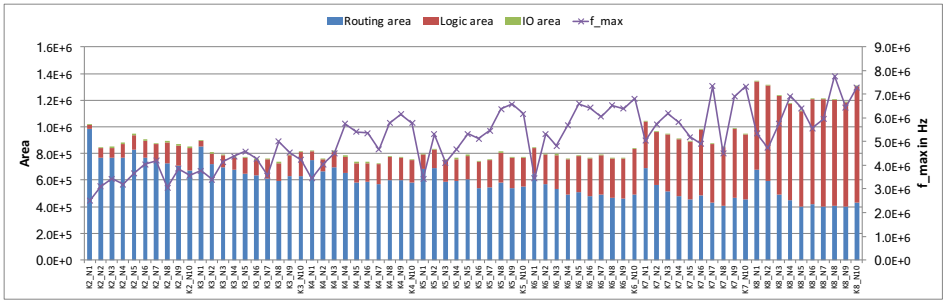


Figure A.2.: Area and performance of *apex2* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

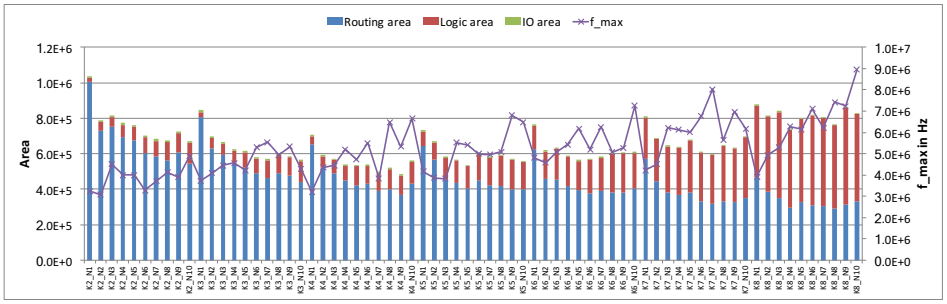


Figure A.3.: Area and performance of *apex4* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

Figure A.4.: Area and performance of *bigkey* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

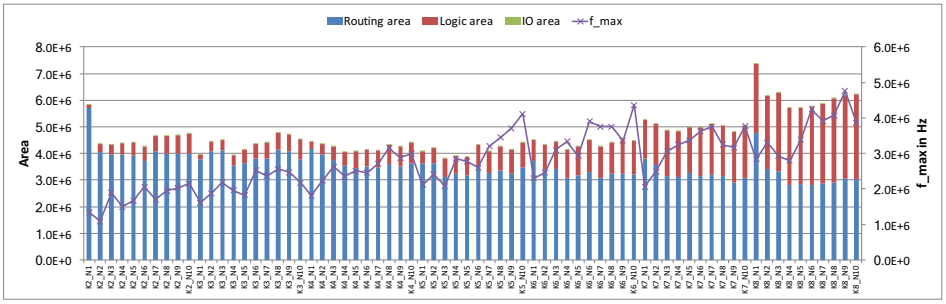


Figure A.5.: Area and performance of *clma* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

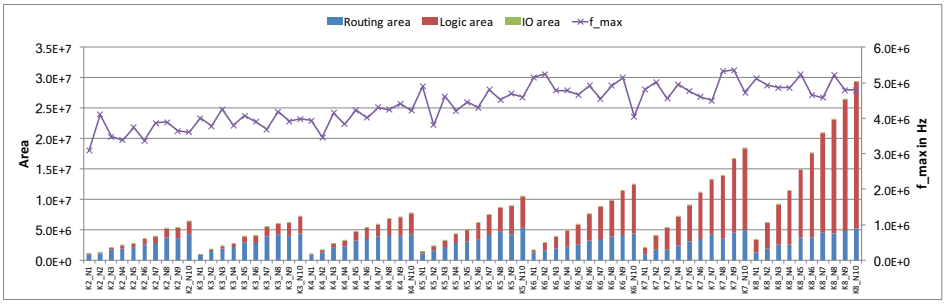


Figure A.6.: Area and performance of *des* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

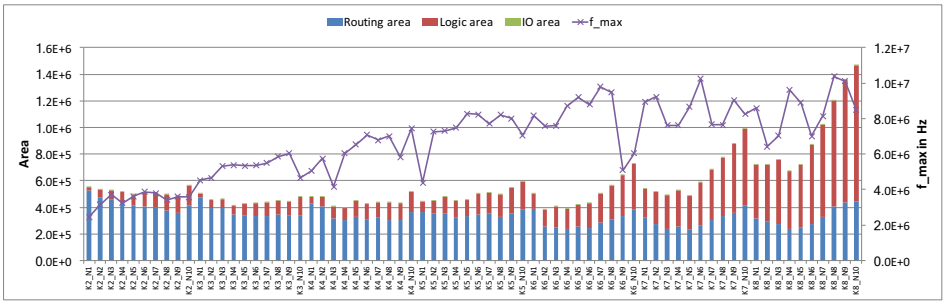


Figure A.7.: Area and performance of *diffeq* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

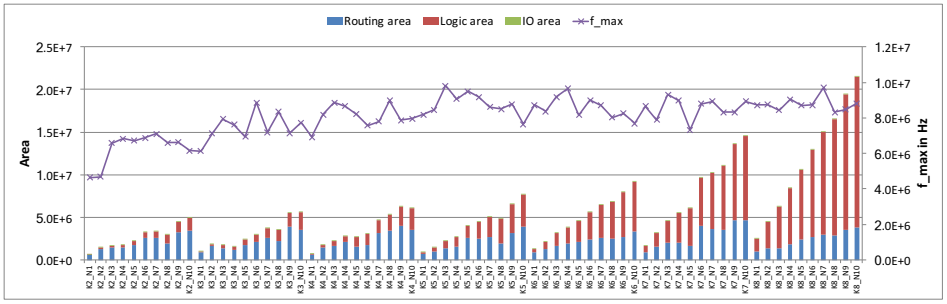


Figure A.8.: Area and performance of *dsip* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

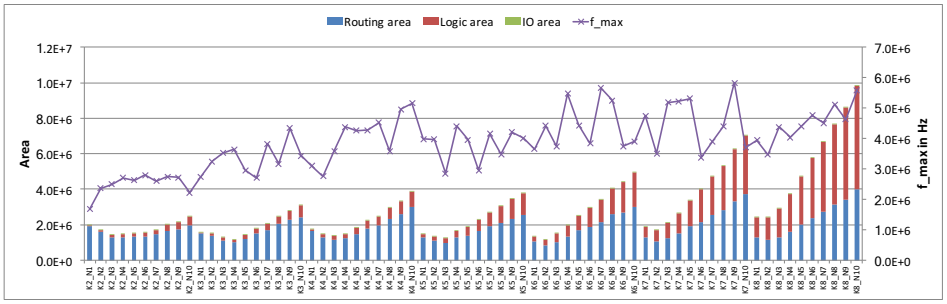


Figure A.9.: Area and performance of *elliptic* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

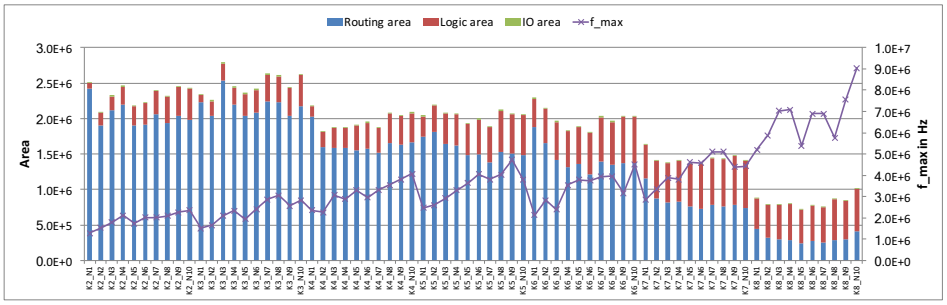


Figure A.10.: Area and performance of *ex1010* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

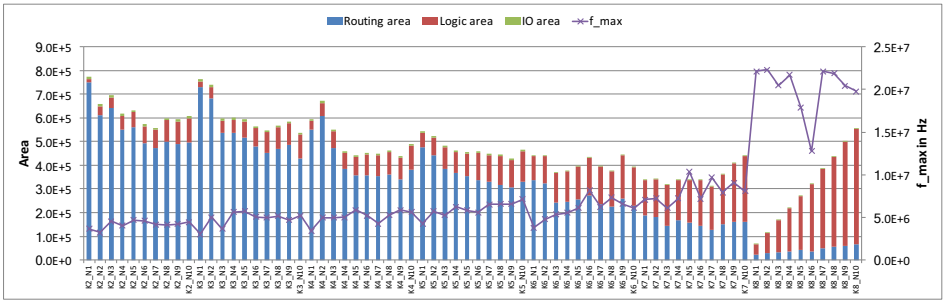


Figure A.11.: Area and performance of *ex5p* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

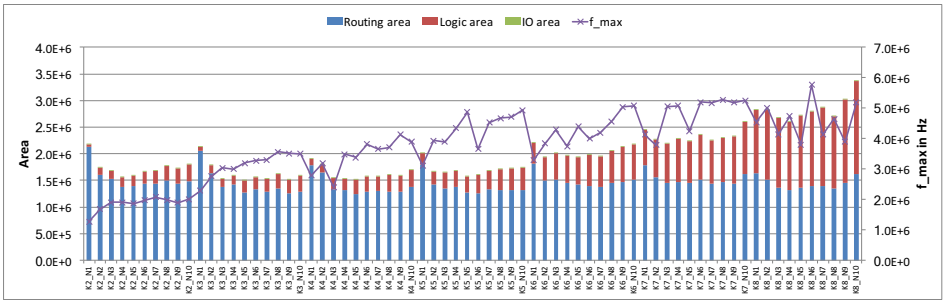


Figure A.12.: Area and performance of *frisc* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

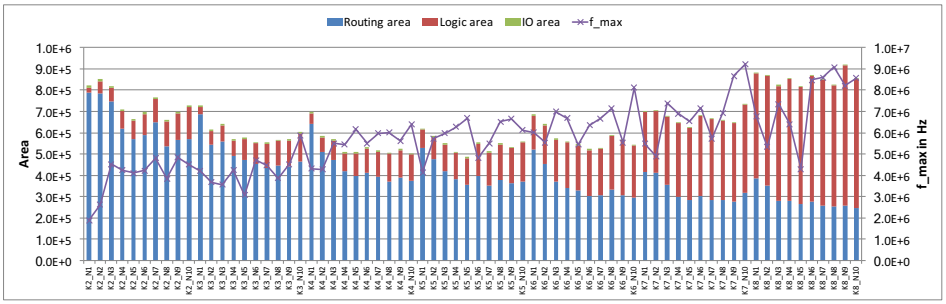


Figure A.13.: Area and performance of *misex3* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

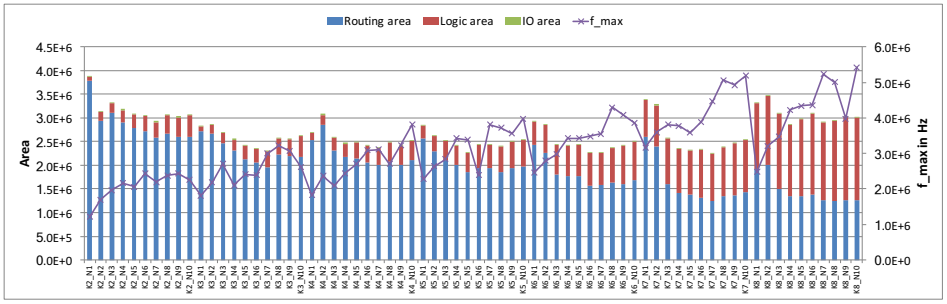


Figure A.14.: Area and performance of *pdc* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

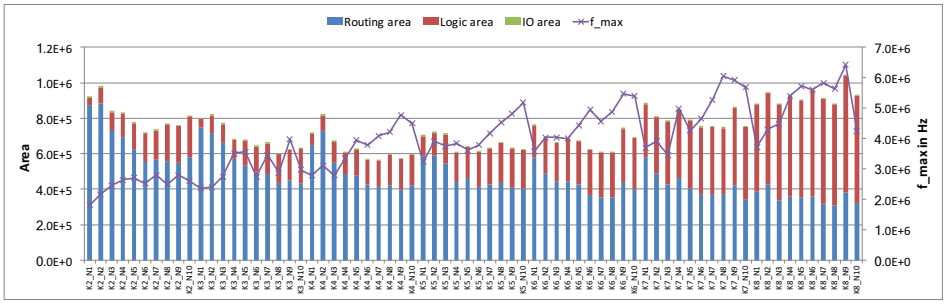


Figure A.15.: Area and performance of *s298* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

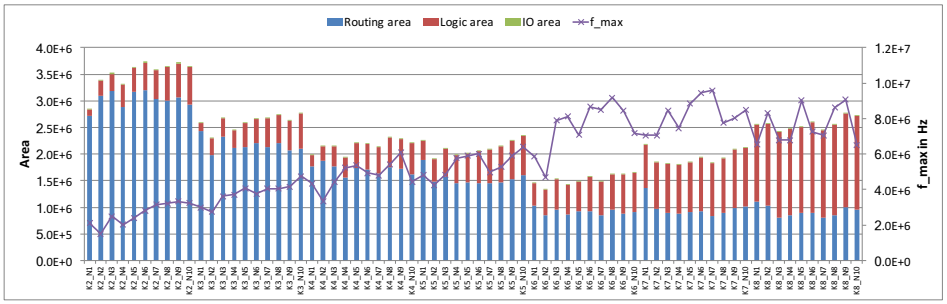


Figure A.16.: Area and performance of *s38417* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

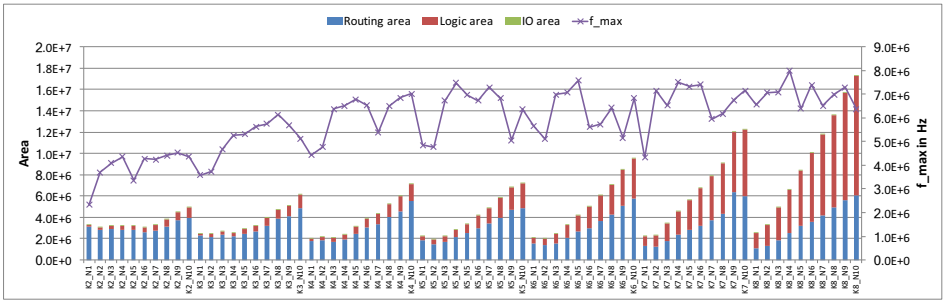


Figure A.17.: Area and performance of *s38584.1* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

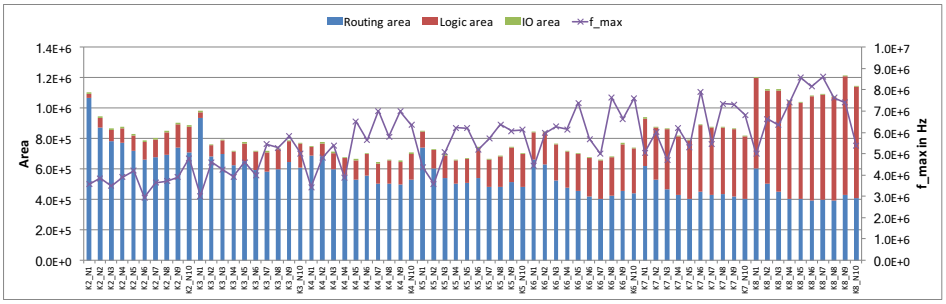


Figure A.18.: Area and performance of *seq* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

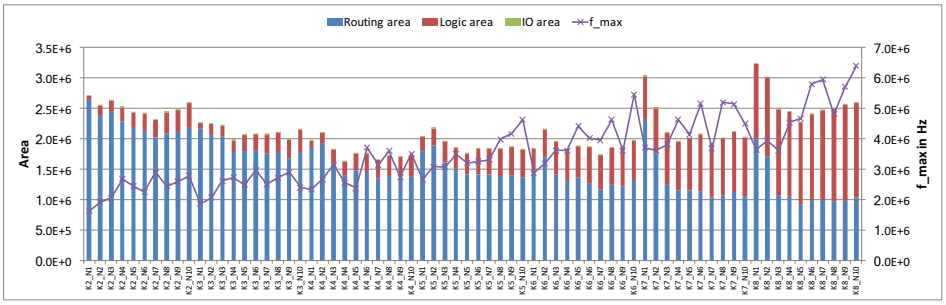


Figure A.19.: Area and performance of *spla* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

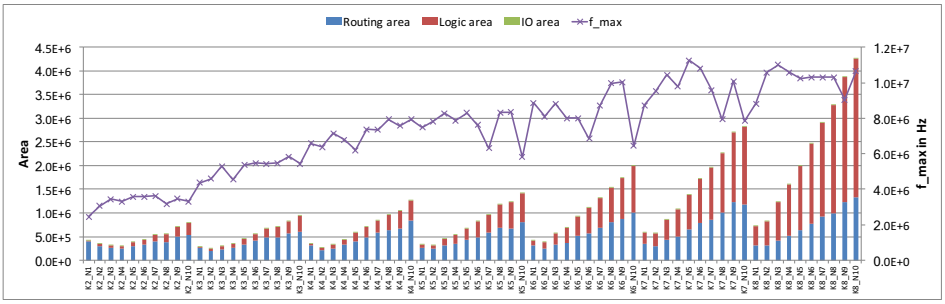


Figure A.20.: Area and performance of *tseng* benchmark mapped on *V-FPGA* with various LUT and cluster sizes

A.2. Architecture File Template

The following template is used by the *V-FPGA Explorer* tool to generate so-called architecture files that are needed by the *VPR* tool to perform packing, place & route of applications onto the *V-FPGA*.

```
<!-- V-FPGA Architecture Template for VTR 7.0. -->

<architecture>

  <!--
       ODIN II specific config begins
       Describes the types of user-specified netlist blocks (in blif, this
            corresponds to
       .model [type_of_block]) that this architecture supports.

       Note: Basic LUTs, I/Os, and flip-flops are not included here as there are
       already special structures in blif (.names, .input, .output, and .latch)
       that describe them.
  -->
  <models>
  </models>
  <!-- ODIN II specific config ends -->

  <!-- Physical descriptions begin -->
  <layout auto="1.0"/>

  <device>
    <!-- ipin_mux_trans_size is calculated from the area of a 2:1 MUX (including
         the logic and the programming) devided by the number of transistors in
         the area model of VPR for 2-level mux (6*ld(N_inputs)+2*N_inputs-2).
         Caution: this area model includes also the area of SRAM cells for
         programming! However in V-FPGA flipflops are used for programming. This
         introduces some error, but the error is too small to affect the P&R.
T_ipin_cblock is aproximated to the delay of a 4:1 mux for simplicity. Depending
     on the channel width and Fc there can be variations.
-->
    <sizing R_minW_nmos="8926" R_minW_pmos="16067" ipin_mux_trans_size="0.25"/>
    <timing C_ipin_cblock="1.47e-15" T_ipin_cblock="1492e-12"/>

    <!-- The grid_logic_tile_area below will be used for all blocks that do not
         explicitly set their own (non-routing)
        area; set to 0 since we explicitly set the area of all blocks currently
             in this architecture file.
    -->
    <area grid_logic_tile_area="0"/>
    <chan_width_distr>
      <io width="1.000000"/>
      <x distr="uniform" peak="1.000000"/>
      <y distr="uniform" peak="1.000000"/>
    </chan_width_distr>
    <switch_block type="#SB_Type#" fs="3"/>
```

```
38    </device>
39    <switchlist>
40    <!-- mux_trans_size should be similar like ipin_mux_trans_size, however values
           < 1.0 lead to a crash during routing area estimation -->
41      <switch type="mux" name="0" R="0" Cin=".77e-15" Cout="4e-15" Tdel="#T_mux4#"
            mux_trans_size="1" buf_size="0"/>
42    <switch type="buffer" name="1" R="551" Cin=".77e-15" Cout="4e-15" Tdel="1492e
          -12" buf_size="27.645901"/>
43    </switchlist>
44    <segmentlist>
45    <!-- Rmetal and Cmetal are selected in a way that the resulting delay
           corresponds to a net delay of the underlying platform -->
46      <segment freq="1.000000" length="1" type="bidir" Rmetal="11066" Cmetal="22.5e
            -15">
47        <wire_switch name="1"/>
48        <opin_switch name="1"/>
49        <sb type="pattern">1 1</sb>
50        <cb type="pattern">1</cb>
51      </segment>
52    </segmentlist>
53
54    <complexblocklist>
55
56      <!-- Define I/O pads begin -->
57      <!-- Capacity is a unique property of I/Os, it is the maximum number of I/Os
             that can be placed at the same (X,Y) location on the FPGA -->
58      <pb_type name="io" capacity="2" area="#A_IOB#">
59        <input name="outpad" num_pins="1"/>
60        <output name="inpad" num_pins="1"/>
61        <clock name="clock" num_pins="1"/>
62
63        <!-- IOs can operate as either inputs or outputs. -->
64        <mode name="inpad">
65          <pb_type name="inpad" blif_model=".input" num_pb="1">
66            <output name="inpad" num_pins="1"/>
67          </pb_type>
68          <interconnect>
69            <direct name="inpad" input="inpad.inpad" output="io.inpad">
70              <delay_constant max="#T_IOB_in#" in_port="inpad.inpad" out_port="io.
                    inpad"/>
71            </direct>
72          </interconnect>
73        </mode>
74        <mode name="outpad">
75          <pb_type name="outpad" blif_model=".output" num_pb="1">
76            <input name="outpad" num_pins="1"/>
77          </pb_type>
78          <interconnect>
79            <direct name="outpad" input="io.outpad" output="outpad.outpad">
80              <delay_constant max="#T_IOB_out#" in_port="io.outpad" out_port="
                    outpad.outpad"/>
81            </direct>
82          </interconnect>
```

```
83        </mode>
84
85        <!-- Every input pin is driven by 100% of the tracks in a channel, every
               output pin is driven by 100% of the tracks in a channel -->
86        <fc default_in_type="frac" default_in_val="1.0" default_out_type="frac"
               default_out_val="1.0"/>
87
88        <pinlocations pattern="custom">
89          <loc side="left">io.outpad io.inpad io.clock</loc>
90          <loc side="top">io.outpad io.inpad io.clock</loc>
91          <loc side="right">io.outpad io.inpad io.clock</loc>
92          <loc side="bottom">io.outpad io.inpad io.clock</loc>
93        </pinlocations>
94
95        <!-- Place I/Os on the sides of the FPGA -->
96        <gridlocations>
97          <loc type="perimeter" priority="10"/>
98        </gridlocations>
99
100       <power method="ignore"/>
101     </pb_type>
102     <!-- Define I/O pads ends -->
103
104
105     <!-- Define general purpose logic block (CLB) begin -->
106     <pb_type name="clb" area="#A_CLB#">
107       <input name="I" num_pins="#I#" equivalent="true"/>
108       <output name="O" num_pins="#O#" equivalent="false"/>
109       <clock name="clk" num_pins="1"/>
110
111       <!-- Define BLE -->
112       <pb_type name="ble" num_pb="#N#">
113         <input name="in" num_pins="#K#"/>
114         <output name="out" num_pins="1"/>
115         <clock name="clk" num_pins="1"/>
116
117
118   <!-- Define LUT -->
119   <pb_type name="lut" blif_model=".names" num_pb="1" class="lut">
120     <input name="in" num_pins="#K#" port_class="lut_in"/>
121     <output name="out" num_pins="1" port_class="lut_out"/>
122     <!-- LUT timing using delay matrix -->
123     <delay_matrix type="max" in_port="lut.in" out_port="lut.out">
124       #LUT_delay_matrix#
125     </delay_matrix>
126   </pb_type>
127   <!-- LUT definition ends here -->
128
129   <!-- Define flip-flop -->
130   <pb_type name="ff" blif_model=".latch" num_pb="1" class="flipflop">
131     <input name="D" num_pins="1" port_class="D"/>
132     <output name="Q" num_pins="1" port_class="Q"/>
133     <clock name="clk" num_pins="1" port_class="clock"/>
```

```
134     <T_setup value="#T_ff_setup#" port="ff.D" clock="clk"/>
135     <T_clock_to_Q max="#T_ff_clock_to_Q#" port="ff.Q" clock="clk"/>
136   </pb_type>
137   <!-- flip-flop definition ends here -->
138
139   <interconnect>
140     <direct name="direct1" input="ble.in" output="lut.in"/>
141     <direct name="direct2" input="lut.out" output="ff.D">
142       <delay_constant max="#T_net#" in_port="lut.out" out_port="ff.D"/>
143       <!-- Advanced user option that tells CAD tool to find LUT+FF pairs in
              netlist -->
144       <pack_pattern name="ble" in_port="lut.out" out_port="ff.D"/>
145     </direct>
146     <direct name="direct3" input="ble.clk" output="ff.clk"/>
147     <mux name="mux1" input="ff.Q lut.out" output="ble.out">
148       <delay_constant max="#T_mux2#" in_port="lut.out" out_port="ble.out" />
149       <delay_constant max="#T_mux2#" in_port="ff.Q" out_port="ble.out" />
150     </mux>
151   </interconnect>
152       </pb_type>
153       <!-- BLE definition ends here -->
154
155       <interconnect>
156         <complete name="crossbar" input="clb.I ble[#N-1#:0].out" output="ble[#N
                -1#:0].in">
157           <delay_constant max="#T_BLE_inMux#" in_port="clb.I" out_port="ble[#N-1#
                :0].in" />
158           <delay_constant max="#T_BLE_inMux#" in_port="ble[#N-1#:0].out" out_port
                ="ble[#N-1#:0].in" />
159         </complete>
160         <complete name="clks" input="clb.clk" output="ble[#N-1#:0].clk">
161         </complete>
162
163         <direct name="clbouts1" input="ble[#N-1#:0].out" output="clb.O[#N-1#:0]"/
              >
164       </interconnect>
165
166       <!-- Every input pin is driven by 100% of the tracks in a channel, every
              output pin is driven by 100% of the tracks in a channel -->
167       <fc default_in_type="frac" default_in_val="1.0" default_out_type="frac"
              default_out_val="1.0"/>
168
169       <pinlocations pattern="custom">
170     #PIN_LOC_LIST#
171     </pinlocations>
172
173       <!-- Place this general purpose logic block in any unspecified column -->
174       <gridlocations>
175         <loc type="fill" priority="1"/>
176       </gridlocations>
177     </pb_type>
178     <!-- Define general purpose logic block (CLB) ends -->
179
```

```
180
181     </complexblocklist>
182     <power>
183       <local_interconnect C_wire="2.5e-10"/>
184       <mux_transistor_size mux_transistor_size="3"/>
185       <FF_size FF_size="4"/>
186       <LUT_transistor_size LUT_transistor_size="4"/>
187     </power>
188     <clocks>
189       <clock buffer_size="auto" C_wire="2.5e-10"/>
190     </clocks>
191   </architecture>
```

Listing A.1: arch_VFPGA_template.xml

A.3. Frame File Template for Testbench Generation in *V-FPGA Explorer*

```
--------------------------------------------------------------------------------
-- Company: KIT ITIV
--
-- File: tb_TOP_V_FPGA.vhd
-- File history:
--      <Revision number>: <Date>: <Comments>
--      <Revision number>: <Date>: <Comments>
--      <Revision number>: <Date>: <Comments>
--
-- Description:
--
--      Testbench for Virtual FPGA
--
--
-- Targeted device: any FPGA
-- Author: GENERATED AUTOMATICALLY BY V-FPGA EXPLORER!
--
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use work.common.all;

entity tb_TOP_V_FPGA_light is
end tb_TOP_V_FPGA_light;

architecture behavioural of tb_TOP_V_FPGA_light is

    component TOP_V_FPGA_light is
-- TAG_GENERIC
    port (
        nRST        :IN std_logic;
        CLK         :IN std_logic;

        -- AMBA APB Slave for ConfigurationController
        cPSELx      :IN std_logic;
        cPENABLE    :IN std_logic;
        cPADDR      :IN std_logic_vector(15 downto 0);
        cPWRITE     :IN std_logic;
        cPRESETn    :IN std_logic;
        cPCLK       :IN std_logic;
        cPWDATA     :IN std_logic_vector(8 downto 0);
        cPRDATA     :OUT std_logic_vector(8 downto 0);

        -- Pads
    INPADS     :IN std_logic_vector((2*X_MAX*P) + (2*Y_MAX*P) - 1 downto 0);
    OUTPADS    :OUT std_logic_vector((2*X_MAX*P) + (2*Y_MAX*P) - 1 downto 0)
    );
    end component;
```

```
50
51
52
53      signal tb_nRST        :std_logic:='0';
54      signal tb_CLK         :std_logic:='0';
55      signal tb_INPADS   :std_logic_vector((2*X_MAX*P) + (2*Y_MAX*P) - 1 downto 0);
56    signal tb_OUTPADS    :std_logic_vector((2*X_MAX*P) + (2*Y_MAX*P) - 1 downto 0);
57
58  -- TAG_SIGNAL_CONF_APB
59
60
61  begin
62
63      dut: TOP_V_FPGA
64  -- TAG_GENERIC_MAP
65          port map(
66              nRST      => tb_nRST,
67              CLK       => tb_CLK,
68
69  -- TAG_PORT_MAP_CONF_APB
70
71              INPADS    => tb_INPADS,
72              OUTPADS   => tb_OUTPADS
73          );
74
75
76
77  end behavioural;
```

Listing A.2: arch_VFPGA_template.xml

A.4. Python Script for Aligning Block I/Os of Tile Macros

The following Python script generates I/O location constraints for the tile blocks with the correct orientation and optimal alignment that reduces path lengths for inter-tile connections in the physical design stage of the *V-FPGA*.

```
width= 84.0
height= 84.0
f2=open('InnerTile_3D.custom.io', 'w');
W = 23;
f2.write('(globals')
f2.write('\n')
f2.write('  version = 3')
f2.write('\n')
f2.write('  io_order = default')
f2.write('\n')
f2.write(')')
f2.write('\n')
f2.write('(iopin')
f2.write('\n')
f2.write('  (top')
f2.write('\n')
offsetstep=width/(5.0*W+4+2)
offset=offsetstep
for i in range(0,W):
  f2.write('  (pin name="rl_top_import[' + str(i) + ']"  offset=' + str(offset) +
       ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
  f2.write('\n')
  offset=offset+offsetstep
for i in range(0,W):
  f2.write('  (pin name="rl_top_export[' + str(i) + ']"  offset=' + str(offset) +
       ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
  f2.write('\n')
  offset=offset+offsetstep
for i in range(0,W):
  f2.write('  (pin name="lr_top_export[' + str(i) + ']"  offset=' + str(offset) +
       ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
  f2.write('\n')
  offset=offset+offsetstep
f2.write('  (pin name="SCLK_CLB"   offset=' + str(offset) + ' layer=2 width
    =0.0800 depth=0.2500 place_status=placed  )')
f2.write('\n')
offset=offset+offsetstep
f2.write('  (pin name="MOSI_CLB"   offset=' + str(offset) + ' layer=2 width
    =0.0800 depth=0.2500 place_status=placed  )')
f2.write('\n')
offset=offset+offsetstep
f2.write('  (pin name="SCLK_PSM"   offset=' + str(offset) + ' layer=2 width
    =0.0800 depth=0.2500 place_status=placed  )')
f2.write('\n')
offset=offset+offsetstep
```

```
40  f2.write('  (pin name="MOSI_PSM"   offset=' + str(offset) + ' layer=2 width
        =0.0800 depth=0.2500 place_status=placed  )')
41  f2.write('\n')
42  offset=offset+offsetstep
43  for i in range(0,W):
44    f2.write('  (pin name="tb_in[' + str(i) + ']"  offset=' + str(offset) + ' layer
          =2 width=0.0800 depth=0.2500 place_status=placed  )')
45    f2.write('\n')
46    offset=offset+offsetstep
47    f2.write('  (pin name="bt_out[' + str(i) + ']"   offset=' + str(offset) + '
          layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
48    f2.write('\n')
49    offset=offset+offsetstep
50  f2.write('  )')
51  f2.write('\n')
52  f2.write('  (left')
53  f2.write('\n')
54  offsetstep = height/(5.0*W+1+2)
55  offset=offsetstep
56  for i in range(0,W):
57    f2.write('  (pin name="tb_left_export[' + str(i) + ']"   offset=' + str(offset)
          + ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
58    f2.write('\n')
59    offset=offset+offsetstep
60  for i in range(0,W):
61    f2.write('  (pin name="tb_left_import[' + str(i) + ']"   offset=' + str(offset)
          + ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
62    f2.write('\n')
63    offset=offset+offsetstep
64  f2.write('  (pin name="nRST"   offset=' + str(offset) + ' layer=2 width=0.0800
        depth=0.2500 place_status=placed  )')
65  f2.write('\n')
66  offset=offset+offsetstep
67  f2.write('  (pin name="CLK"  offset=' + str(offset) + ' layer=2 width=0.0800
        depth=0.2500 place_status=placed  )')
68  f2.write('\n')
69  offset=offset+offsetstep
70  for i in range(0,W):
71    f2.write('  (pin name="bt_left_import[' + str(i) + ']"   offset=' + str(offset)
          + ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
72    f2.write('\n')
73    offset=offset+offsetstep
74  for i in range(0,W):
75    f2.write('  (pin name="lr_in[' + str(i) + ']"  offset=' + str(offset) + ' layer
          =2 width=0.0800 depth=0.2500 place_status=placed  )')
76    f2.write('\n')
77    offset=offset+offsetstep
78    f2.write('  (pin name="rl_out[' + str(i) + ']"   offset=' + str(offset) + '
          layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
79    f2.write('\n')
80    offset=offset+offsetstep
81  f2.write('  )')
82  f2.write('\n')
```

```
83  f2.write('  (bottom')
84  f2.write('\n')
85  offsetstep=width/(5.0*W+4+2)
86  offset=offsetstep
87  for i in range(0,W):
88    f2.write('  (pin name="rl_bot_export[' + str(i) + ']"  offset=' + str(offset) +
          ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
89    f2.write('\n')
90    offset=offset+offsetstep
91  for i in range(0,W):
92    f2.write('  (pin name="rl_bot_import[' + str(i) + ']"  offset=' + str(offset) +
          ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
93    f2.write('\n')
94    offset=offset+offsetstep
95  for i in range(0,W):
96    f2.write('  (pin name="lr_bot_import[' + str(i) + ']"  offset=' + str(offset) +
          ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
97    f2.write('\n')
98    offset=offset+offsetstep
99  offset=offset+offsetstep
100 f2.write('  (pin name="MOSI_out_CLB"   offset=' + str(offset) + ' layer=2 width
      =0.0800 depth=0.2500 place_status=placed  )')
101 f2.write('\n')
102 offset=offset+offsetstep
103 offset=offset+offsetstep
104 f2.write('  (pin name="MOSI_out_PSM"   offset=' + str(offset) + ' layer=2 width
      =0.0800 depth=0.2500 place_status=placed  )')
105 f2.write('\n')
106 offset=offset+offsetstep
107 for i in range(0,W):
108   f2.write('  (pin name="tb_out[' + str(i) + ']"   offset=' + str(offset) + '
        layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
109   f2.write('\n')
110   offset=offset+offsetstep
111   f2.write('  (pin name="bt_in[' + str(i) + ']"  offset=' + str(offset) + ' layer
        =2 width=0.0800 depth=0.2500 place_status=placed  )')
112   f2.write('\n')
113   offset=offset+offsetstep
114 f2.write('  )')
115 f2.write('\n')
116 f2.write('  (right')
117 f2.write('\n')
118 offsetstep=height/(5.0*W+2+2)
119 offset=offsetstep
120 for i in range(0,W):
121   f2.write('  (pin name="tb_right_import[' + str(i) + ']"  offset=' + str(offset)
        + ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
122   f2.write('\n')
123   offset=offset+offsetstep
124 for i in range(0,W):
125   f2.write('  (pin name="tb_right_export[' + str(i) + ']"  offset=' + str(offset)
        + ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
126   f2.write('\n')
```

```
127     offset=offset+offsetstep
128   offset=offset+offsetstep
129   offset=offset+offsetstep
130   for i in range(0,W):
131     f2.write('  (pin name="bt_right_export[' + str(i) + ']"  offset=' + str(offset)
          + ' layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
132     f2.write('\n')
133     offset=offset+offsetstep
134   for i in range(0,W):
135     f2.write('  (pin name="lr_out[' + str(i) + ']"   offset=' + str(offset) + '
          layer=2 width=0.0800 depth=0.2500 place_status=placed  )')
136     f2.write('\n')
137     offset=offset+offsetstep
138     f2.write('  (pin name="rl_in[' + str(i) + ']"  offset=' + str(offset) + ' layer
          =2 width=0.0800 depth=0.2500 place_status=placed  )')
139     f2.write('\n')
140     offset=offset+offsetstep
141   f2.write('  )')
142   f2.write('\n')
```

Listing A.3: align_ports.py

List of Figures

List of Tables

Acronyms

3D USCT 3D Ultrasound Computer Tomography.

ACCI AC-Coupled Interconnection.
AHB Advanced High-Performance Bus.
Ai Artificial Intelligence.
AIG And-Inverter Graph.
ALAP as late as possible.
AMBA Advanced Microcontroller Bus Architecture.
APB Advanced Peripheral Bus.
ASIC Application-Specific Integrated Circuit.

BLE Basic Logic Element.
BLIF Berkeley Logic Interchange Format.
BRAM Block Random Access Memory.

CB Connection Box.
CLB Configurable Logic Block.
COTS Commercial off-the-Shelf.
CPS Cyber-Physical System.
CTS Clock Tree Synthesis.

DFG Data Flow Graph.
DSE Design Space Exploration.
DSP Digital Signal Processing.

EDA Electronic Design Automation.
ELF Executable and Linkable Format.

FCCM Field-Programmable Custom Computing Machine.
FFC Flip-flop Count.

FPGA Field Programmable Gate Array.
FSM Finite State Machine.

GDSII Graphics Database System.
GPP General-Purpose Processor.
GPU Graphics Processor Unit.
GUI Graphical User Interface.

HART Highway Addressable Remote Transducer.

IOB I/O Block.
IoT Internet of Things.
IPO In Place Optimization.

JIT Just-in-Time.

LE Logic Element.
LEF Library Exchange Format.
LKE LUT_K Equivalent.
LUT Lookup Table.

M2E MUX2 Equivalent.
MCM Multi-Chip Module.
MISD Multiple Instruction, Single Data.
MOSI Master Out Slave In.
MSBE Minimum Size Basic Element.

NoC Network on Chip.

OASIS Open Artwork System Interchange Standard.

PR Partial Reconfiguration.
PSM Programmable Switch Matrix.

QUIP Quartus University Interface Program.

SIMD Single Instruction Multiple Data.
SoC System-on-Chip.

SOI Silicon-on-Insulator.
SPI Serial Peripheral Interface.
SSH Secure Shell.
SVG Scalable Vector Graphics.

TCL Tool Command Language.
TSV Through-Silicon Via.

V-FPGA Virtual Field Programmable Gate Array.
V-TSV Virtual Through-Silicon Via.
VHDL Very High Speed Integrated Circuit Hardware Description Language.
ViSA VLIW inspired Slot Architecture.
ViSA-VIS ViSA and V-FPGA Integrated System.
VLIW Very Long Instruction Word.

WSN Wireless Sensor Network.

Bibliography

[1] Achronix Semiconductor Corporation. Speedcore eFPGA Datasheet (DS003), 2016. URL: https://www.achronix.com/product/speedcore/.

[2] Actel Corporation. ProASIC3 FPGA Fabric User's Guide, 2010. URL: http://www.actel.com/documents/PA3_UG.pdf.

[3] ADICSYS. Synthesizable Programmable Core, 2012. URL: http://www.adicsys.com/efpga-products/.

[4] E. Ahmed and J. Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):288–298, Mar. 2004. doi:10.1109/TVLSI.2004.824300.

[5] Algotronix. A Brief History of Algotronix' Configurable Array Logic (CAL) Technology, 2007. URL: http://www.algotronix.com/company/Photo%20Album.html.

[6] C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration, the VLSI Journal*, 19(1-2):1–81, Aug. 1995. doi:10.1016/0167-9260(95)00008-4.

[7] Altera. Synthesis Design Flows Using the Quartus University Interface Program (QUIP). Document quip_synthesis_interface.pdf in the QUIP package, 2007. URL: https://www.altera.com/support/support-resources/software/download/altera_design/quip.html.

[8] Altera. Stratix V Device Handbook, 2016. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf.

[9] Amazon EC2 F1-Instances, 2017. URL: https://aws.amazon.com/de/ec2/instance-types/f1/.

[10] AMDREL Consortium. Deliverable D9 - Survey of Existing Fine-Grain Reconfigurable Hardware Platforms, 2002. URL: http://proteas.microlab.ntua.gr/amdrel/pdf/d9_final.pdf.

[11] ASICs...the course, Chapter 4. Programmable ASICs. URL: http://www.csit-sun.pub.ro/resources/asic/CH04.pdf.

[12] M. Balzer, M. Birk, R. Dapp, H. Gemmeke, E. Kretzek, S. Menshikov, M. Zapf, and N. Ruiter. 3D Ultrasound Computer Tomography for Breast Cancer Diagnosis. In *18th IEEE-NPSS Real Time Conference (RT)*, 2012. doi:10.1109/RTC.2012.6418198.

[13] V. Betz and J. Rose. Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size. In *Proceedings of the IEEE 1997 Custom Integrated Circuits Conference*, pages 551–554, May 1997. doi:10.1109/CICC.1997.606687.

[14] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, FPL '97, pages 213–222, London, UK, UK, 1997. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=647924.738755.

[15] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[16] M. Birk, M. Balzer, N. Ruiter, and J. Becker. Comparison of Processing Performance and Architectural Efficiency Metrics for FPGAs and GPUs in 3D Ultrasound Computer Tomography. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2012. doi:10.1109/ReConFig.2012.6416735.

[17] Berkeley Logic Interface Format (BLIF), Feb. 2005. URL: http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/links/blif.pdf.

[18] B. Blodget, S. McMillan, and P. Lysaght. A Lightweight Approach for Embedded Reconfiguration of FPGAs. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 399–400, 2003. doi:10.1109/DATE.2003.1253642.

[19] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011. doi:10.1145/1941487.1941507.

[20] A. Brant and G. G. F. Lemieux. ZUMA: An Open FPGA Overlay Architecture. In *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, Apr. 2012. doi:10.1109/FCCM.2012.25.

[21] T. Bruckschlögl. Dynamische Defragmentierung in Dynamisch und Partiell Rekonfigurierbarem Multicore V-FPGA. Studienarbeit IL-960, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2011.

[22] Cadence Design Systems Inc. Using Encounter® RTL Compiler, Product Version 14.1, Sept. 2014.

[23] A. Cataldo. M2000 Seeks to Merge FPGAs, ASICs. Article in EE Times, 2004. URL: http://www.eetimes.com/document.asp?doc_id=1240024.

[24] Y.-W. Chang, D. F. Wong, and C. K. Wong. Universal Switch Modules for FPGA Design. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1(1):80–101, Jan. 1996. doi:10.1145/225871.225886.

[25] K. Chapman. Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources, Oct. 2014. URL: https://www.xilinx.com/support/documentation/application_notes/xapp522-mux-design-techniques.pdf.

[26] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys (CSUR)*, 34(2):171–210, June 2002. doi:10.1145/508352.508353.

[27] J. Cong, H. P. Li, S. K. Lim, T. Shibuya, and D. Xu. Large Scale Circuit Partitioning with Loose/Stable Net Removal and Signal Flow Based Clustering. In *Computer-Aided Design, 1997. Digest of Technical Papers., 1997 IEEE/ACM International Conference on*, pages 441–446. IEEE, 1997.

[28] W. R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. M. Sule, M. Steer, and P. D. Franzon. Demystifying 3D ICs: the Pros and Cons of Going Vertical. *IEEE Design & Test of Computers*, 22(6):498–510, Nov. 2005. doi:10.1109/MDT.2005.136.

[29] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct. 1974. doi:10.1109/JSSC.1974.1050511.

[30] S. Doctor, T. Hall, and L. Reid. SAFT - The Evolution of a Signal Processing Technology for Ultrasonic Testing. *NDT International*, 19:163 – 167, 1986.

[31] Efficiency, 2016. URL: https://en.wikipedia.org/wiki/Efficiency.

[32] EFLX™ Reconfigurable RTL Blocks for your SOC/MCU, 2015. URL: http://www.flex-logix.com.

[33] D. Fang, S. Peng, C. Lafrieda, and R. Manohar. A Three-Tier Asynchronous FPGA. In *Proceedings of the International VLSI/ULSI Multilevel Interconnection Conference*, 2006. doi:10.1.1.139.147.

[34] P. Figuli, W. Ding, S. Figuli, K. Siozios, D. Soudris, and J. Becker. Parameter Sensitivity in Virtual FPGA Architectures. In S. Wong, A. C. Beck, K. Bertels, and L. Carro, editors, *Applied Reconfigurable Computing*, pages 141–153. Springer International Publishing, Cham, 2017.

[35] P. Figuli, M. Hübner, R. Girardey, F. Bapp, T. Bruckschloegl, F. Thoma, J. Henkel, and J. Becker. A Heterogeneous SoC Architecture with Embedded Virtual FPGA Cores and Runtime Core Fusion. In *NASA/ESA 6th Conference on Adaptive Hardware and Systems (AHS 2011)*, June 2011. doi:10.1109/AHS.2011.5963922.

[36] P. Figuli, C. Tradowsky, N. Gaertner, and J. Becker. ViSA: A Highly Efficient Slot Architecture Enabling Multi-Objective ASIP Cores. In *System on Chip (SoC), 2013 International Symposium on*, pages 1–8, Oct. 2013. doi:10.1109/ISSoC.2013.6675270.

[37] P. Figuli, C. Tradowsky, J. Martinez, H. Sidiropoulos, K. Siozios, H. Stenschke, D. Soudris, and J. Becker. A Novel Concept for Adaptive Signal Processing on Reconfigurable Hardware. In K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, editors, *Applied Reconfigurable Computing*, pages 311–320. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-16214-0_26.

[38] S. Figuli, A. Sonnino, P. Figuli, and J. Becker. A Variable FPGA Based Generic QAM Transmitter with Scalable Mixed Time and Frequency Domain Signal Processing. In *39th International Conference on Telecommunications and Signal Processing*, June 2016.

[39] W. Fornaciari and V. Piuri. Virtual FPGAs: Some Steps Behind the Physical Barriers. In J. Rolim, editor, *Parallel and Distributed Processing*, pages 7–12. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/3-540-64359-1_665.

[40] F. Furtek. Programmable Logic Cell and Array, May 28 1991. US Patent 5,019,736. URL: http://www.google.com/patents/US5019736.

[41] G. Shalina P. D., T. Bruckschlögl, P. Figuli, C. Tradowsky, G. M. Almeida, and J. Becker. Article: Bringing Accuracy to Open Virtual Platforms (OVP): A Safari from High-Level Tools to Low-Level Microarchitectures. *IJCA Proceedings on In-*

ternational Conference on Innovations in Intelligent Instrumentation, Optimization and Electrical Sciences, ICIIIOES(10):22–27, Dec. 2013.

[42] H. Gao, Y. Yang, X. Ma, and G. Dong. Analysis of the Effect of LUT Size on FPGA Area and Delay Using Theoretical Derivations. In *Sixth International Symposium on Quality Electronic Design (ISQED'05)*, pages 370–374, Mar. 2005. doi:10.1109/ISQED.2005.20.

[43] A. Gayasen, V. Narayanan, M. Kandemir, and A. Rahman. Designing a 3-D FPGA: Switch Box Architecture and Thermal Issues. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(7):882–893, July 2008. doi:10.1109/TVLSI.2008.2000456.

[44] A. Gayasen, Y. Tsai, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and T. Tuan. Reducing Leakage Energy in FPGAs Using Region-Constrained Placement. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, pages 51–58, New York, 2004. ACM. doi:10.1145/968280.968289.

[45] P. K. Gupta. Accelerating Datacenter Workloads. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2016.

[46] L. Hagen and A. B. Kahng. A New Approach to Effective Circuit Clustering. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '92, pages 422–427, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. URL: http://dl.acm.org/citation.cfm?id=304032.304146.

[47] N. Hemsoth and T. Morgan. *FPGA Frontiers: New Applications in Reconfigurable Computing, 2017 Edition*. Next Platform Press, 2017.

[48] Y. Hu, S. Das, S. Trimberger, and L. He. Design, Synthesis and Evaluation of Heterogeneous FPGA with Mixed LUTs and Macro-Gates. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 188–193, Nov. 2007. doi:10.1109/ICCAD.2007.4397264.

[49] M. Hübner. *Dynamisch und Partiell Rekonfigurierbare Hardwaresystemarchitektur mit Echtzeitfähiger On-Demand Funktionalität*. PhD thesis, 2007. Karlsruhe, Univ., Diss., 2007.

[50] M. Hübner, P. Figuli, R. Girardey, D. Soudris, K. Siozos, and J. Becker. A Heterogeneous Multicore System on Chip with Run-Time Reconfigurable Virtual FPGA Architecture. In *18th Reconfigurable Architectures Workshop (RAW)*, May 2011.

[51] S. Huda, M. Mallick, and J. H. Anderson. Clock Gating Architectures for FPGA Power Reduction. In *2009 International Conference on Field Programmable Logic and Applications*, pages 112–118, Aug. 2009. doi:10.1109/FPL.2009.5272538.

[52] Intel. User Customizable SoC FPGAs, 2017. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/br/br-soc-fpga.pdf.

[53] D. Jansen and G. Albert. *Handbuch der Electronic-Design-Automation*. Hanser, 2001.

[54] B. Kettelhoit. Architektur und Entwurf Dynamisch Rekonfigurierbarer FPGA-Systeme, 2009. URL: http://digital.ub.uni-paderborn.de/ubpb/urn/urn:nbn:de:hbz:466-20090402033.

[55] M. Koefferlein. KLayout - High Performance Layout Viewer and Editor. URL: https://www.klayout.de.

[56] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, Feb. 2007. doi:10.1109/TCAD.2006.884574.

[57] I. Kuon, R. Tessier, and J. Rose. FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, Feb. 2008. doi:10.1561/1000000005.

[58] L. Lagadec, D. Lavenier, E. Fabiani, and B. Pottier. Placing, Routing, and Editing Virtual FPGAs. In G. Brebner and R. Woods, editors, *Field-Programmable Logic and Applications: 11th International Conference, FPL 2001 Belfast, Northern Ireland, UK, August 27-29, 2001 Proceedings*, pages 357–366. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. doi:10.1007/3-540-44687-7_37.

[59] M. Leeser, W. M. Meleis, M. M. Vai, S. Chiricescu, W. Xu, and P. M. Zavracky. Rothko: A Three-Dimensional FPGA. *IEEE Design & Test of Computers*, 15(1):16–23, Jan. 1998. doi:10.1109/54.655178.

[60] G. Lemieux, E. Lee, M. Tom, and A. Yu. Directional and Single-Driver Wires in FPGA Interconnect. In *IEEE International Conference on Field Programmable Technology (FPT)*, pages 41–48, Dec. 2004. doi:10.1109/FPT.2004.1393249.

[61] M. Lin, A. El Gamal, Y.-C. Lu, and S. Wong. Performance Benefits of Monolithically Stacked 3D-FPGA. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 113–122, New York, NY, USA, 2006. ACM. doi:10.1145/1117201.1117219.

[62] M. Lin, A. E. Gamal, Y. C. Lu, and S. Wong. Performance Benefits of Monolithically Stacked 3-D FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):216–229, Feb. 2007. doi:10.1109/TCAD.2006.887920.

[63] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6:1–6:30, June 2014.

[64] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose. VPR 5.0: FPGA CAD and Architecture Exploration Tools with Single-Driver Routing, Heterogeneity and Process Scaling. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 133–142, New York, NY, USA, 2009. ACM. doi:10.1145/1508128.1508150.

[65] R. Lysecky, K. Miller, F. Vahid, and K. Vissers. Firm-Core Virtual FPGA for Just-in-Time FPGA Compilation. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 271–271, New York, NY, USA, 2005. ACM. doi:10.1145/1046192.1046247.

[66] D. Marple. An MPGA-Like FPGA. *IEEE Design & Test of Computers*, 9(4):51–60, Dec. 1992. doi:10.1109/54.173333.

[67] M. I. Masud and S. J. E. Wilton. A New Switch Block for Segmented FPGAs. In P. Lysaght, J. Irvine, and R. Hartenstein, editors, *Field Programmable Logic and Applications (FPL'99)*, pages 274–281. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. doi:10.1007/978-3-540-48302-1_28.

[68] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays*, FPGA '95, pages 111–117, New York, NY, USA, 1995. ACM. doi:10.1145/201310.201328.

[69] MEANDER Design Framework. http://proteas.physics.auth.gr/meander/, 2017 (accessed January 25, 2017).

[70] W. M. Meleis, M. Leeser, P. Zavracky, and M. M. Vai. Architectural Design of a Three Dimensional FPGA. In *Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI '97)*, ARVLSI '97, pages 256–, Washington, DC, USA, 1997. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=786452.786697.

[71] eFPGA Core IP Product Brief, 2015. URL: http://www.menta-efpga.com.

[72] Microsemi Corporation. Accelerator Series FPGAs - ACT 3 Family, 2012. URL: https://www.microsemi.com/document-portal/doc_view/130668-accelerator-series-fpgas-act-3-family.

[73] Microsemi Corporation. SmartFusion2 SoC FPGA, Product Brief), 2016. URL: https://www.microsemi.com/document-portal/doc_download/132721-pb0115-smartfusion2-soc-fpga-product-brief.

[74] Microsemi Corporation. IGLOO2 Product Information Brochure, 2017. URL: https://www.microsemi.com/document-portal/doc_download/132013-igloo2-product-information-brochure.

[75] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-Aware AIG Rewriting: a Fresh Look at Combinational Logic Synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 532–535, July 2006. doi:10.1145/1146909.1147048.

[76] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965. URL: http://www.intel.com/technology/mooreslaw/index.htm.

[77] H. Muroga, H. Murata, Y. Saeki, T. Hibi, Y. Ohashi, T. Noguchi, and T. Nishimura. A Large Scale FPGA with 10k Core Cells with CMOS 0.8 µm 3-Layered Metal Process. In *Custom Integrated Circuits Conference, 1991., Proceedings of the IEEE 1991*, pages 6.4/1–6.4/4, May 1991. doi:10.1109/CICC.1991.164125.

[78] D. Nenni and P. McLellan. *Fabless: The Transformation of the Semiconductor Industry*. Createspace Independent Pub, 2014.

[79] B. Neumann, T. von Sydow, H. Blume, and T. G. Noll. Application Domain Specific Embedded FPGAs for Flexible ISA-Extension of ASIPs. *Journal of Signal Processing Systems*, 53(1):129–143, 2008. doi:10.1007/s11265-008-0211-9.

[80] V. F. Pavlidis and E. G. Friedman. *Three-Dimensional Integrated Circuit Design*. The Morgan Kaufmann Sseries in Systems on Silicon. Elsevier Morgan Kaufmann, Amsterdam, 2009.

[81] E. Pop, S. Sinha, and K. E. Goodson. Heat Generation and Transport in Nanometer-Scale Transistors. *Proceedings of the IEEE*, 94(8):1587–1601, Aug. 2006. doi:10.1109/JPROC.2006.879794.

[82] G. Pulini and D. Hulance. FlexEOS Embedded FPGA Solution. In N. S. Voros, A. Rosti, and M. Hübner, editors, *Dynamic System Reconfiguration in Heterogeneous Platforms: The MORPHEUS Approach*, pages 39–47. Springer Netherlands, Dordrecht, 2009. doi:10.1007/978-90-481-2427-5_4.

[83] A. Putnam. Large-Scale Reconfigurable Computing in a Microsoft Datacenter. In *Proceedings of the 26th IEEE HotChips Symposium on High-Performance Chips (HotChips 2014)*. IEEE, Aug. 2014. URL: https://www.microsoft.com/en-us/research/publication/large-scale-reconfigurable-computing-microsoft-datacenter/.

[84] S. A. Razavi, M. S. Zamani, and K. Bazargan. A Tileable Switch Module Architecture for Homogeneous 3D FPGAs. In *2009 IEEE International Conference on 3D System Integration*, pages 1–4, Sept. 2009. doi:10.1109/3DIC.2009.5306586.

[85] L. Rizzatti. What's the Difference Between FPGA and Custom Silicon Emulators? Article in Electronic Design, 2014. URL: http://electronicdesign.com/eda/what-s-difference-between-fpga-and-custom-silicon-emulators.

[86] J. Rose, R. J. Francis, D. Lewis, and P. Chow. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency. *IEEE Journal of Solid-State Circuits*, 25(5):1217–1225, 1990.

[87] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 77–86, New York, NY, USA, 2012. ACM. doi:10.1145/2145694.2145708.

[88] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html.

[89] L. Shannon. Reconfigurable Computing Architectures. In S. Hauck and A. DeHon, editors, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, pages 29–46. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[90] H. Sidiropoulos, P. Figuli, K. Siozios, D. Soudris, and J. Becker. A Platform-Independent Runtime Methodology for Mapping Multiple Applications onto FPGAs Through Resource Virtualization. In *2013 23rd International Conference on Field Programmable Logic and Applications*, pages 1–4, Sept. 2013. doi:10.1109/FPL.2013.6645564.

[91] H. Sidiropoulos, K. Siozios, P. Figuli, D. Soudris, and M. Hubner. On Supporting Efficient Partial Reconfiguration with Just-In-Time Compilation. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 328–335, May 2012. doi:10.1109/IPDPSW.2012.40.

[92] H. Sidiropoulos, K. Siozios, P. Figuli, D. Soudris, M. Hübner, and J. Becker. JITPR: A Framework for Supporting Fast Application's Implementation onto FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 6(2):7:1–7:12, Aug. 2013. URL: http://doi.acm.org/10.1145/2492185.

[93] H. Sidiropoulos, K. Siozios, and D. Soudris. NAROUTO: An Open-Source Framework for Supporting Architecture-Level Exploration at Heterogeneous FPGAs. In *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, pages 527–530, Dec. 2010. doi:10.1109/ICECS.2010.5724565.

[94] K. Siozios, A. Bartzas, and D. Soudris. Architecture-Level Exploration of Alternative Interconnection Schemes Targeting 3D FPGAs: A Software-Supported Methodology. *International Journal of Reconfigurable Computing*, 2008, 2008. doi:10.1155/2008/764942.

[95] K. Siozios, P. Figuli, H. Sidiropoulos, C. Tradowsky, D. Diamantopoulos, K. Maragos, S. P. Delicia, D. Soudris, and J. Becker. TEAChER: TEach AdvanCEd Reconfigurable Architectures and Tools. In K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, editors, *Applied Reconfigurable Computing*, pages 103–114. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-16214-0_9.

[96] K. Siozios, G. Koutroumpezis, K. Tatas, N. Vassiliadis, V. Kalenteridis, H. Pournara, I. Pappas, D. Soudris, A. Thanailakis, S. Nikolaidis, and S. Siskos. A Novel FPGA Architecture and an Integrated Framework of CAD Tools for Implementing Applications. *IEICE Transactions on Information and Systems*, E88-D(7):1369–1380, July 2005. doi:10.1093/ietisy/e88-d.7.1369.

[97] K. Siozios, V. F. Pavlidis, and D. Soudris. A Novel Framework for Exploring 3-D FPGAs with Heterogeneous Interconnect Fabric. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 5(1):4:1–4:23, Mar. 2012. doi:10.1145/2133352.2133356.

[98] K. Siozios, H. Sidiropoulos, D. Diamantopoulos, P. Figuli, D. Soudris, M. Hübner, and J. Becker. On Designing Self-Aware Reconfigurable Platforms. In *Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS)*, Jan. 2012.

[99] J. S. Soofiani and N. Masoumi. Area Efficient Switch Box Topologies for 3D FPGAs. In *2011 IEEE 9th International New Circuits and systems conference*, pages 390–393, June 2011. doi:10.1109/NEWCAS.2011.5981252.

[100] E. Sotiriou-Xanthopoulos, G. Shalina P. D., P. Figuli, K. Siozios, G. Economakos, and J. Becker. A Power Estimation Technique for Cycle-Accurate Higher-Abstraction SystemC-Based CPU Models. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 70–77. IEEE, 2015.

[101] X. Tang and L. Wang. The Effect of LUT Size on Nanometer FPGA Architecture. In *2012 IEEE 11th International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 1–4, Oct. 2012. doi:10.1109/ICSICT.2012.6467767.

[102] S. M. Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103(3):318–331, Mar. 2015. doi:10.1109/JPROC.2015.2392104.

[103] M. Ullmann, M. Hübner, B. Grimm, and J. Becker. An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 135–, Apr. 2004. doi:10.1109/IPDPS.2004.1303106.

[104] Pre-mapped MCNC20 benchmarks contained in Verilog to Routing (VTR) package, 2016. URL: https://docs.verilogtorouting.org/en/latest/vtr/benchmarks/#mcnc20-benchmarks.

[105] T. Wiersema, A. Bockhorn, and M. Platzner. An Architecture and Design Tool Flow for Embedding a Virtual {FPGA} into a Reconfigurable System-on-Chip. *Computers and Electrical Engineering*, pages –, 2016. URL: http://dx.doi.org/10.1016/j.compeleceng.2016.04.005.

[106] S. J. E. Wilton. Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory, 1997. URL: www.eecg.toronto.edu/~jayar/pubs/theses/Wilton/StevenWilton.pdf.

[107] S. J. E. Wilton, N. Kafafi, J. C. H. Wu, K. A. Bozman, V. O. Aken'Ova, and R. Saleh. Design Considerations for Soft Embedded Programmable Logic Cores. *IEEE Journal of Solid-State Circuits*, 40(2):485–497, Feb. 2005. doi:10.1109/JSSC.2004.841038.

[108] Xilinx. *Xilinx: The Programmable Logic Data Book*. Xilinx, 1994. URL: https://books.google.de/books?id=2TG-OwAACAAJ.

[109] Xilinx. 7 Series FPGAs Data Sheet: Overview, 2016. URL: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[110] Xilinx. Zynq-7000 All Programmable SoC, Product Brief, 2016. URL: https://www.xilinx.com/support/documentation/product-briefs/zynq-7000-product-brief.pdf.

[111] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0. Technical report, MCNC Technical Report, Jan. 1991.

[112] F. L. Yuan, C. C. Wang, T. H. Yu, and D. Markovi. A Multi-Granularity FPGA with Hierarchical Interconnects for Efficient and Flexible Mobile Computing. *IEEE Journal of Solid-State Circuits*, 50(1):137–149, Jan. 2015. doi:10.1109/JSSC.2014.2372034.

[113] Q. Zhao, Y. Iwai, M. Amagasaki, M. Iida, and T. Sueyoshi. A Novel Reconfigurable Logic Device Base on 3D Stack Technology. In *3D Systems Integration Conference (3DIC), 2011 IEEE International*, pages 1–4, Jan. 2012. doi:10.1109/3DIC.2012.6263022.

[114] ZUMA repository. https://github.com/adbrant/zuma-fpga/tree/master/source/templates, 2016 (accessed November 25, 2016).

Supervised Student Research

[AI13] AL-IMAM, AHMAD: *Untersuchung und Konzeptentwurf für eine Energieautarke und Mobile Fahrgastinformationsanzeige*. Diplomarbeit ID-1655, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.

[Bap11] BAPP, FALCO: *Evaluation einer Virtuellen Rekonfigurierbaren Heterogenen System-on-Chip Architektur*. Studienarbeit IL-948, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2011.

[Bra13] BRAUNSCHWEIGER, ROBERT: *FPGA-basierte Sparse-Matrix-Vektor Multiplikation zur Beschleunigung der Transmissionsrekonstruktion in der 3D Ultraschall Computertomographie*. Bachelor-Arbeit ID-1766, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.

[Bru11] BRUCKSCHLÖGL, THOMAS: *Dynamische Defragmentierung in Dynamisch und Partiell Rekonfigurierbarem Multicore V-FPGA*. Studienarbeit IL-960, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2011.

[Bru12] BRUCKSCHLÖGL, THOMAS: *Evaluierung und Optimierung von Embedded Systems Durch Profiling*. Diplomarbeit ID-1524, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2012.

[Des13] DESSECKER, JÜRGEN: *Evaluation of a Low-Power FPGA System Compared to a Microprocessor Based Approach*. Master-Arbeit ID-1762, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.

[Gut16] GUTMANN, STEFAN: *Realisierung eines Mehrfrequenzmessverfahrens für die Kapazitive Füllstandmesstechnik*. Master-Arbeit ID-2162, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2016.

[Hel13] HELD, FELIX: *Vergleich von Modellbasierten High-Level Beschreibungen für FPGA Zielarchitekturen Anhand von Audio-Signalverarbeitungsalgorithmen*. Bachelor-Arbeit ID-1710, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.

[Hub12] HUBER, MARTIN: *Parallele FPGA Online Rekonfiguration für die Datenverarbeitung bei der 3D Ultraschall- Computertomographie*. Bachelor-Arbeit, Hochschule Karlsruhe - Technik und Wirtschaft, Fakultät für Elektro- und Informationstechnik, 2012.

[Kar13] KARCHER, NICK: *High-Level Modeling of a HMPSoC with Just-In-Time Support*. Bachelor-Arbeit ID-1741, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.

[Kle16] KLEINER, THOMAS: *Entwicklung einer FPGA-basierten Prototyping-Umgebung für Hardware-in-the-Loop Tests von Videoverarbeitungsmodulen*. Master-Arbeit ID-1948,

Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2016.

[Mar13] MARTINEZ, JOSÉ ANTONIO LUCIO: *Adaptive Digital Audio Signal Processing on Partial and Dynamic Reconfigurable Hardware*. Master-Arbeit ID-1762, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.

[Mec13] MECHLER, MICHAEL: *Design und Implementierung von Debugmethoden für den V-FPGA*. Bachelor-Arbeit ID-1670, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.

[San13] SANDJONG, WULLIAM TCHOUMY: *Entwicklung eines Heterogenen System-on-Chip mit Laufzeitadaptivem Beschleuniger*. Master-Arbeit ID-1700, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.

[Sch16] SCHMITT, ERIC: *Validation of Mechatronic Systems*. Master-Arbeit ID-1896, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2016.

[Sei12] SEIDENSPINNER, ERIK: *Design und Implementierung von Signalverarbeitungsalgorithmen auf Unterschiedlichen Zielarchitekturen*. Bachelor-Arbeit ID-1615, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2012.

[Son15] SONNINO, ALBERTO: *Performance Driven Optimizations in FPGA Based QAM Systems*. Master-Arbeit ID-2034, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2015.

[Wan16] WANKMÜLLER, SEBASTIAN: *Fehlertolerante Gigabit Datenübertragung für Vibrometersysteme*. Master-Arbeit ID-2173, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2016.

[Wel16] WELLER, DENNIS: *Partial Differential Equations Implemented on FPGAs using OpenCL* . Master-Arbeit ID-2125, Karlsruher Institut für Technologie, Chair of Dependable Nano Computing (CDNC), 2016.

Own Publications

[BKF+16] BIRK, MATTHIAS, ERNST KRETZEK, PETER FIGULI, MARC WEBER, JÜRGEN BECKER and NICOLE V. RUITER: *High-Speed Medical Imaging in 3D Ultrasound Computer Tomography*. IEEE Transactions on Parallel and Distributed Systems, 27(2):455–467, Feb 2016.

[FDF+17] FIGULI, PETER, WEIQIAO DING, SHALINA FIGULI, KOSTAS SIOZIOS, DIMITRIOS SOUDRIS and JÜRGEN BECKER: *Parameter Sensitivity in Virtual FPGA Architectures*. In WONG, STEPHAN, ANTONIO CARLOS BECK, KOEN BERTELS and LUIGI CARRO (editors): *Applied Reconfigurable Computing*, pages 141–153. Springer International Publishing, Cham, 2017.

[FFB17] FIGULI, SHALINA PERCY DELICIA, PETER FIGULI and JÜRGEN BECKER: *A Reconfigurable High-Speed Spiral FIR Filter Architecture*. In *40th International Conference on Telecommunications and Signal Processing (TSP)*, pages 532–537, July 2017. **Best Paper Award**.

[FFSB17] FIGULI, SHALINA PERCY DELICIA, PETER FIGULI, ALBERTO SONNINO and JÜRGEN BECKER: *A Generic Reconfigurable Mixed Time and Frequency Domain QAM Transmitter with Forward Error Correction*. International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems (IJATES), 6(2):80–88, 2017.

[FHG+11] FIGULI, PETER, MICHAEL HÜBNER, ROMUALD GIRARDEY, FALCO BAPP, THOMAS BRUCKSCHLOEGL, FLORIAN THOMA, JÖRG HENKEL and JÜRGEN BECKER: *A Heterogeneous SoC Architecture with Embedded Virtual FPGA Cores and Runtime Core Fusion*. In *NASA/ESA 6th Conference on Adaptive Hardware and Systems (AHS 2011)*, June 2011.

[FSFB16] FIGULI, SHALINA PERCY DELICIA, ALBERTO SONNINO, PETER FIGULI and JÜRGEN BECKER: *A Variable FPGA Based Generic QAM Transmitter with Scalable Mixed Time and Frequency Domain Signal Processing*. In *39th International Conference on Telecommunications and Signal Processing (TSP)*, pages 453–457, June 2016.

[FTGB13] FIGULI, PETER, CARSTEN TRADOWSKY, NADINE GÄRTNER and JÜRGEN BECKER: *ViSA: A Highly Efficient Slot Architecture Enabling Multi-Objective ASIP Cores*. In *International Symposium on System on Chip (SoC)*, pages 1–8, Oct 2013.

[FTM+15] FIGULI, PETER, CARSTEN TRADOWSKY, JOSE MARTINEZ, HARRY SIDIROPOULOS, KOSTAS SIOZIOS, HOLGER STENSCHKE, DIMITRIOS SOUDRIS and JÜRGEN BECKER: *A Novel Concept for Adaptive Signal Processing on Reconfigurable Hardware*. In SANO, KENTARO, DIMITRIOS SOUDRIS,

MICHAEL HÜBNER and PEDRO C. DINIZ (editors): *Applied Reconfigurable Computing*, pages 311–320. Springer International Publishing, Cham, 2015.

[GBF+13] G., SHALINA PERCY DELICIA, THOMAS BRUCKSCHLÖGL, PETER FIGULI, CARSTEN TRADOWSKY, GABRIEL MARCHESAN ALMEIDA and JÜRGEN BECKER: *Bringing Accuracy to Open Virtual Platforms (OVP): A Safari from High-Level Tools to Low-Level Microarchitectures*. IJCA Proceedings on International Conference on Innovations in Intelligent Instrumentation, Optimization and Electrical Sciences, ICIIIOES(10):22–27, December 2013.

[GFB15] G., SHALINA PERCY DELICIA, PETER FIGULI and JÜRGEN BECKER: *Parametric Design Space Exploration for Optimizing QAM Based High-Speed Communication*. In *2015 IEEE/CIC International Conference on Communications in China (ICCC)*, pages 1–5, Nov 2015.

[HFG+11] HÜBNER, MICHAEL, PETER FIGULI, ROMUALD GIRARDEY, DIMITRIOS SOUDRIS, KOSTAS SIOZOS and JÜRGEN BECKER: *A Heterogeneous Multicore System on Chip with Run-Time Reconfigurable Virtual FPGA Architecture*. In *18th Reconfigurable Architectures Workshop (RAW)*, May 2011.

[SFS+13] SIDIROPOULOS, HARRY, PETER FIGULI, KOSTAS SIOZIOS, DIMITRIOS SOUDRIS and JÜRGEN BECKER: *A Platform-Independent Runtime Methodology for Mapping Multiple Applications onto FPGAs through Resource Virtualization*. In *23rd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2013.

[SFS+15] SIOZIOS, KOSTAS, PETER FIGULI, HARRY SIDIROPOULOS, CARSTEN TRADOWSKY, DIONYSIOS DIAMANTOPOULOS, KONSTANTINOS MARAGOS, SHALINA PERCY DELICIA, DIMITRIOS SOUDRIS and JÜRGEN BECKER: *TEAChER: TEach AdvanCEd Reconfigurable Architectures and Tools*. In SANO, KENTARO, DIMITRIOS SOUDRIS, MICHAEL HÜBNER and PEDRO C. DINIZ (editors): *Applied Reconfigurable Computing*, pages 103–114. Springer International Publishing, Cham, 2015.

[SSD+12] SIOZIOS, KOSTAS, HARRY SIDIROPOULOS, DYONISIOS DIAMANTOPOULOS, PETER FIGULI, DIMITRIOS SOUDRIS, MICHAEL HÜBNER and JÜRGEN BECKER: *On Designing Self-Aware Reconfigurable Platforms*. In *Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS)*, January 2012.

[SSF+12] SIDIROPOULOS, HARRY, KOSTAS SIOZIOS, PETER FIGULI, DIMITRIOS SOUDRIS and MICHAEL HÜBNER: *On Supporting Efficient Partial Reconfiguration with Just-In-Time Compilation*. In *26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 328–335, May 2012. **Best Paper Award**.

[SSF+13] SIDIROPOULOS, HARRY, KOSTAS SIOZIOS, PETER FIGULI, DIMITRIOS SOUDRIS, MICHAEL HÜBNER and JÜRGEN BECKER: *JITPR: A Framework for Supporting Fast Application's Implementation onto FPGAs*. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 6(2):7:1–7:12, August 2013.

[SXGF+15] SOTIRIOU-XANTHOPOULOS, EFSTATHIOS, SHALINA PERCY DELICIA G., PETER FIGULI, KOSTAS SIOZIOS, GEORGE ECONOMAKOS and JÜRGEN BECKER:

A Power Estimation Technique for Cycle-Accurate Higher-Abstraction SystemC-Based CPU Models. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 70–77. IEEE, 2015.

[TFS+13] TRADOWSKY, CARSTEN, PETER FIGULI, ERIK SEIDENSPINNER, FELIX HELD and JÜRGEN BECKER: *A New Approach to Model-Based Development for Audio Signal Processing*. In *134th Audio Engineering Society (AES) Convention*, May 2013.