Field-Programmable Gate Arrays Explained

A high-level introduction to FPGAs

CONTENTS

1. An Introduction to FPGAs 1
2. FPGA Architecture 8
3. FPGA Design Flow 33
4. FPGA Tools and Development Environments 43
5. FPGA Programming Languages 51
6. Design Techniques and Best Practices 71
7. Testing and Verification 77
8. Intellectual Property (IP) Cores and Design Reuse 84
9. Real-World Case Studies 90

Appendix 95

©2024

Chapter One: An Introduction to FPGAs

This Handbook provides a high-level introduction into FPGAs and is split into four main parts: Definition and Overview; Historical Evolution; Applications and Use Cases; and Advantages and Limitations. This section aims to give a general idea of what FPGAs are, where they come from, and where they are used.

Field-Programmable Gate Arrays (FPGAs) represent a versatile and powerful class of integrated circuits that offer a unique blend of flexibility and performance. Unlike traditional Application-Specific Integrated Circuits (ASICs), FPGAs are programmable at the hardware level after manufacturing. This characteristic allows users to configure the chip's functionality to suit specific application requirements. FPGAs consist of an array of configurable logic blocks interconnected by programmable routing resources. These logic blocks can be customized to perform a wide range of digital functions, making FPGAs well-suited for tasks such as digital signal processing, image and video processing, networking, and more.

The design process for FPGAs involves creating a hardware description using Hardware Description Languages (HDLs). Commonly used HDLs include VHDL and Verilog. This hardware description is then synthesized and implemented using specialized tools, generating a configuration bitstream that defines the interconnections and functionality of the FPGA. This ability to reconfigure hardware dynamically makes FPGAs ideal for rapid prototyping, iterative design, and applications where adaptability is critical.

FPGAs find applications across various industries, including telecommunications, automotive, aerospace, and consumer electronics. Their parallel processing capabilities, low-level hardware customization, and ability to implement complex algorithms in hardware make them indispensable for addressing computational challenges in a diverse range of fields. As technology continues to advance, FPGAs remain at the forefront of innovation, playing a key role in the development of cutting-edge solutions for today's complex digital systems.

What to expect in Chapter One:

- Definition and Overview
- Historical Evolution of FPGAs
- Application and Use Cases
- Advantages and Limitations

Definition and Overview

FPGAs, are semiconductor devices that offer a unique and reconfigurable approach to digital circuit design. Unlike traditional fixed-function integrated circuits, FPGAs provide a blank canvas of logic elements and programmable interconnects that users can configure and reprogram to implement a wide variety of digital functions.

At the core of an FPGA's versatility is its architecture, typically composed of an array of Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), programmable interconnects, and other essential components. Configurable Logic Blocks contain Look-Up Tables (LUTs) and flip-flops, allowing users to define and implement digital logic functions. IOBs manage input and output connections, enabling seamless interaction with external devices. This is illustrated in Figure 1 below.

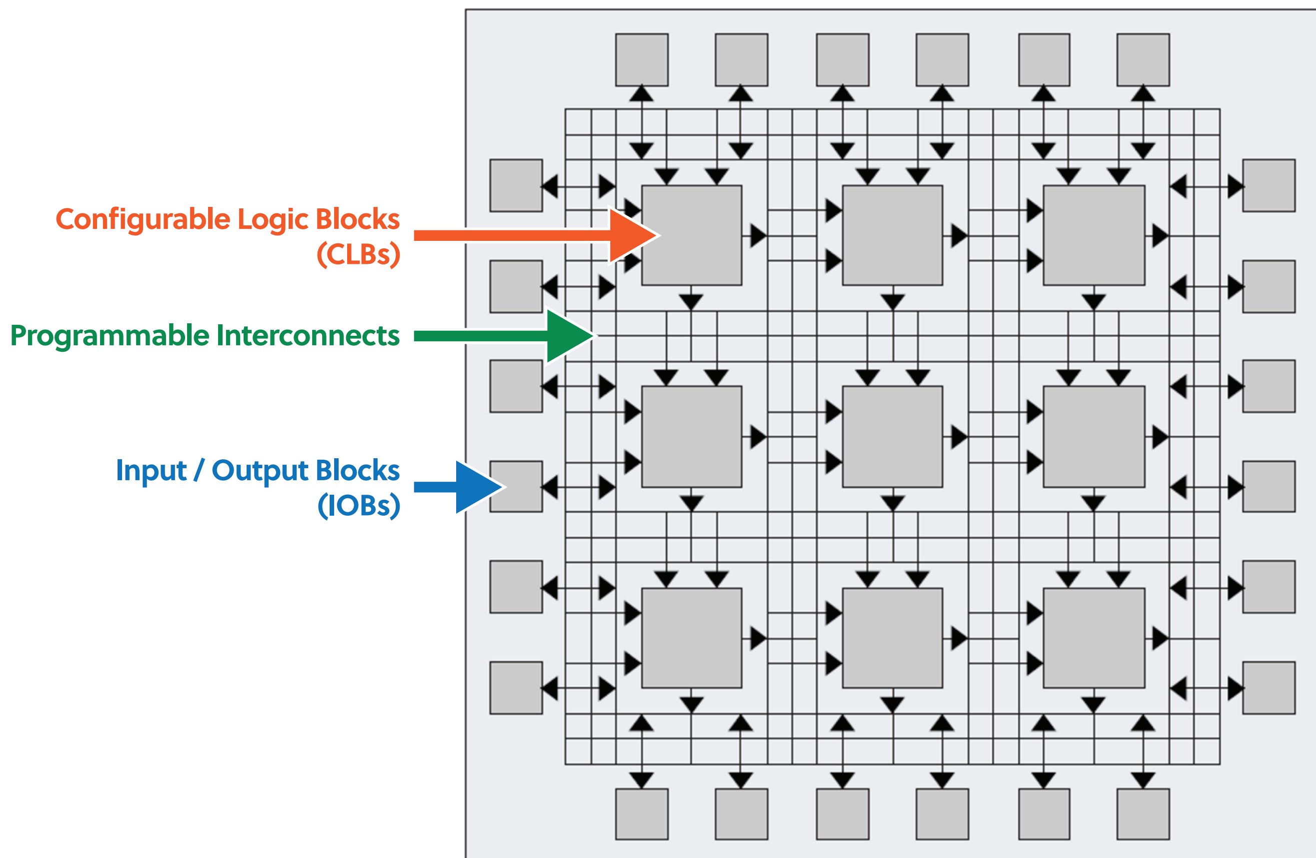


Figure 1: An abstract view of an FPGA; Control Logic Blocks are embedded in a general routing [1]

FPGAs consist of a versatile internal architecture designed for digital circuit implementation. At the core are CLBs housing Logic Elements (LEs) capable of both combinational and sequential logic operations. Interconnects form a grid-like structure, incorporating a Switching Matrix for flexible signal routing across CLBs. IOBs interface with external signals, supporting various standards, while Block RAM (BRAM) provides both distributed and dedicated memory resources. Dedicated Digital Signal Processing (DSP) Blocks, equipped with specialized Multiply-Accumulate (MAC) units, optimize the implementation of signal processing algorithms. Clock management features,

including buffers and Phase-Locked Loops (PLLs), facilitate precise timing control. Configuration memory stores the bits defining FPGA functionality, and a configuration interface enables reprogramming. This reconfigurability, coupled with a diverse set of resources, makes FPGAs suitable for applications ranging from rapid prototyping to specialized high-performance computing tasks. Section 2 of this Handbook gives a more detailed view of the FPGA's architecture.

The key distinguishing feature of FPGAs is their programmability. Designers use hardware description languages such as VHDL or Verilog to create a hardware-level description of the desired digital circuit. Through a series of design steps, including synthesis and place-and-route processes, this description is translated into a configuration bitstream. This bitstream, when loaded onto the FPGA, effectively "programs" the device, defining its internal connections and functionality.

FPGAs find applications across a broad spectrum of industries due to their adaptability and performance. They are particularly valuable in prototyping and development stages of electronic systems, where rapid iteration and modification are essential. Additionally, FPGAs play a crucial role in applications requiring parallel processing, real-time signal processing, and tasks demanding hardware acceleration. FPGAs were not always as advanced and complex as the ones widely available today. The next section gives an interesting look at the historical evolution which lead to the FPGA we know today.

Historical Evolution

This chapter looks deeper into the historical evolution of these dynamic devices. Building upon the foundations laid in the introductory chapter, we will navigate through pivotal moments and key milestones that have shaped the trajectory of FPGA development. From the rudimentary origins of programmable logic arrays to the emergence of Configurable Logic Blocks (CLBs) and the birth of true FPGA architecture, this chapter unfolds the narrative of innovation and adaptation. By tracing the footsteps of industry leaders, exploring technological breakthroughs, and understanding the driving forces behind each evolutionary leap, the aim is to provide a comprehensive narrative that not only captures the historical nuances but also sheds light on the transformative impact of FPGAs on the digital landscape. There is a rich history to be unravelled, exploring the threads that have woven together to create the sophisticated programmable devices we know today.

Origins and Early Concepts

The concept of programmable logic dates to the 1970s when researchers began exploring ways to create flexible digital circuits. Pioneering works continued through the late 1970s, leading to the development of programmable logic arrays (PLAs). These early experiments laid the groundwork for the evolution of more sophisticated programmable devices.

Emergence of PALs and CPLDs

In the 1980s, Programmable Array Logic (PAL) devices and Complex Programmable Logic Devices (CPLDs) emerged as precursors to FPGAs. PALs offered fixed OR arrays with programmable AND

arrays, while CPLDs introduced more complex architectures with multiple PLAs and interconnection resources. These developments marked significant steps toward the reconfigurable logic landscape.

Birth of FPGA Architecture

The true FPGA era began in the late 1980s and early 1990s with the introduction of devices like the Xilinx (now part of AMD) XC2064. These early FPGAs featured configurable logic blocks and programmable interconnects, allowing users to implement custom digital circuits. The shift towards Look-Up Tables (LUTs) as configurable elements brought unprecedented flexibility, enabling the realization of complex designs.

Rapid Advancements in the Late 20th Century

The late 1990s and early 2000s witnessed a rapid evolution of FPGA technology. Xilinx and Altera (now part of Intel) emerged as industry leaders, introducing successive generations of FPGAs with increased logic density, improved performance, and additional features such as embedded memory blocks and Digital Signal Processing (DSP) resources. These advancements spurred the adoption of FPGAs in various applications, including telecommunications, aerospace, and signal processing.

Rise of High-Performance FPGAs

As the demand for high-performance computing increased, FPGAs evolved to include dedicated resources for specialized tasks. The integration of DSP blocks, high-speed transceivers, and hardened IP cores for specific functions like PCIe and Ethernet communication enhanced the suitability of FPGAs for a broader range of applications.

Reconfigurable Computing and Parallel Processing

In the 21st century, FPGAs found a niche in reconfigurable computing and parallel processing. Their ability to adapt to specific algorithms and computational tasks made them attractive for applications in machine learning, image processing, and scientific computing.

Ongoing Trends and Future Prospects

Today, FPGAs continue to evolve with advancements such as heterogeneous integration of processors (SoC FPGAs), enhanced security features, and increased focus on power efficiency. The dynamic nature of the FPGA landscape suggests a promising future, with ongoing research and development aimed at addressing emerging challenges and expanding the scope of FPGA applications.

The historical evolution of FPGAs reflects a journey from basic programmable logic concepts to sophisticated, highly configurable devices that play a pivotal role in modern digital systems. As technology advances, FPGAs are poised to remain at the forefront of innovation, adapting to new challenges and unlocking possibilities in diverse fields.

Applications and Use Cases

Earlier in this chapter, this handbook provided a brief description of the inner functionality of the FPGA. Given this, you may notice that the inner workings of the FPGA give rise to parallelism when it comes to digital processing. This parallelism provides an excellent tool for applications where speed is of the essence. FPGAs have become integral components in a wide array of applications due to their flexibility, reconfigurability, and high-performance capabilities. This includes their role in telecommunications, networking, signal processing, and real-time applications. This chapter takes a look at this diverse range of applications and use cases where FPGAs play a crucial role, showcasing their adaptability in addressing complex challenges across various industries. As technology continues to advance, FPGAs are expected to play an even more pivotal role in shaping the landscape of digital systems.

Communications and Networking

FPGAs have long been instrumental in the telecommunications and networking sectors. From implementing high-speed data interfaces like PCIe and Ethernet to enabling the development of customizable and efficient network processors, FPGAs have played a vital role in the evolution of communication technologies. Their ability to handle real-time processing and adapt to changing standards makes them ideal for routers, switches, and communication infrastructure.

Digital Signal Processing (DSP) and Audio/Video Processing

The parallel processing capabilities of FPGAs make them well-suited for DSP applications. FPGAs can be tailored to implement custom signal processing algorithms, making them valuable in areas such as audio and video processing, image recognition, and compression. Their ability to handle parallel data streams efficiently has led to widespread adoption in broadcasting, multimedia, and video surveillance.

Embedded Systems and IoT

FPGAs find a growing presence in embedded systems and Internet of Things (IoT) devices. As embedded processors within larger systems or standalone FPGA-based systems, they offer rapid prototyping, adaptability, and real-time processing. In IoT applications, FPGAs contribute to energy-efficient processing, sensor interfacing, and protocol customization.

Aerospace and Defence

In the aerospace and defence industries, where reliability and performance are critical, FPGAs are deployed in radar systems, communication modules, and avionics. Their ability to handle complex algorithms, rapid reconfiguration, and resistance to radiation make them suitable for harsh environments. FPGAs contribute to the development of advanced radar signal processing, secure communication systems, and electronic warfare applications.

Medical Imaging and Healthcare

FPGAs play a vital role in medical imaging applications, powering devices such as ultrasound machines, CT scanners, and MRI systems. Their parallel processing capabilities enhance image processing speed and accuracy. FPGAs also find application in wearable medical devices, providing real-time processing and customization for patient-specific requirements.

High-Performance Computing (HPC) and Acceleration

As accelerators, FPGAs are increasingly used in high-performance computing environments. They excel in applications requiring parallel processing, such as scientific simulations, computational finance, and artificial intelligence. FPGAs can be integrated with traditional processors to create heterogeneous systems, providing a balance between flexibility and performance.

Automotive Electronics

In the automotive industry, FPGAs contribute to advanced driver assistance systems (ADAS), in-vehicle infotainment, and control systems. Their adaptability allows for the implementation of evolving standards, and their parallel processing capabilities enhance real-time processing for safety-critical applications.

Advantages and Limitations of FPGAs

FPGAs, with their unique blend of flexibility and reconfigurability, offer myriad advantages that have propelled them to the forefront of digital design. Yet, alongside these strengths, FPGAs come tethered to certain limitations that demand careful consideration in the design and implementation process. On the positive side, FPGAs offer unparalleled flexibility and reconfigurability, allowing designers to tailor hardware to meet specific application requirements. This reprogramming capability enables iterative design processes and facilitates quick updates without necessitating hardware changes. FPGAs excel in parallel processing tasks, providing high performance and hardware acceleration for computationally intensive applications.

The low-latency nature of FPGAs makes them well-suited for real-time processing in fields like communications and control systems. Additionally, FPGAs can be power-efficient when optimized for specific functions, and their integration of IP cores expedites development by incorporating pre-designed functional blocks. However, FPGAs also come with limitations, including finite resources that must be carefully managed in complex designs. Cost considerations, programming complexity, and a potential learning curve for hardware description languages (HDLs) can pose challenges. While FPGAs are adept at parallel tasks, they may not be as efficient for purely sequential operations, and security concerns regarding bitstream protection need to be addressed. Long development cycles and vendor dependence on specific tools and libraries are additional factors that should be considered when choosing FPGAs for a particular application.

Chapter One Summary

Chapter One has provided a comprehensive overview of Field-Programmable Gate Arrays, spanning their fundamental definition, historical evolution, diverse applications, and a detailed examination of their advantages and limitations. FPGAs, with their reconfigurable nature, have evolved from early programmable logic concepts to sophisticated devices pivotal in modern digital systems. The historical overview traced the trajectory from Programmable Array Logic (PAL) devices to the emergence of FPGAs with Configurable Logic Blocks (CLBs) and programmable interconnects. The exploration of applications showcased the versatility of FPGAs across industries, from communications and signal processing to healthcare and high-performance computing. The advantages highlighted their flexibility, parallel processing capabilities, and suitability for specific tasks, while acknowledging limitations such as resource constraints, cost considerations, and programming complexity. This holistic understanding of FPGAs sets the stage for subsequent chapters, delving deeper into their architecture, programming methodologies, and advanced features, providing readers with a robust foundation for further exploration into the realm of programmable logic.

In the next chapter, we're going to take a close look at how FPGAs are put together. After discussing their history, uses, and pros and cons, we're now going to explore how these flexible devices actually work. The chapter will explain the different parts of FPGAs, like CLBs, interconnects, IOBs, Block Ram (BRAM), and more. We'll go through the pathways that can be programmed, understand the logic elements, and look at special blocks like Digital Signal Processing (DSP) units. By focusing on the basic structure, this chapter aims to help readers understand how FPGAs turn digital designs into physical results. Keep reading as we uncover the details of FPGA architecture, revealing the complexities that make these devices important in the world of programmable logic.

Chapter Two: FPGA Architecture

In this chapter, we embark on a detailed exploration of FPGA architecture to get a closer look at the intricacies that define these reconfigurable marvels.

What to expect in Chapter Two:

- FPGA Chip
- Configurable Logic Blocks (CLBs)
- Input/Output Blocks (IOBs)
- Programmable Interconnect
- Embedded Resources
- Clock Management Resources
- Configuration and Programming
- System-on-Chip Architecture (SoC)

The FPGA Chip

In this visual exploration, we present an abstract view of the internal architecture of a FPGA. Figure 2 provides a simplified representation, capturing the fundamental components that constitute the core of an FPGA. Here, you will encounter Configurable Logic Blocks (CLBs) – the versatile units where programmable logic is configured to execute specific functions. Input/Output Blocks (IOBs) act as gateways, facilitating communication between the FPGA and the external world. Interconnect pathways weave through the architecture, forming the intricate network that enables data flow between various components.

It is essential to note that this abstract view serves as a simplified representation. FPGA architectures are significantly more complex, featuring additional elements, specialized resources, and advanced functionalities. This chapter seeks to take a detailed look the intricacies of the basic components – CLBs, IOBs, and interconnects together with other very important parts in modern FPGAs – providing an understanding of their roles and interactions within the broader FPGA framework.

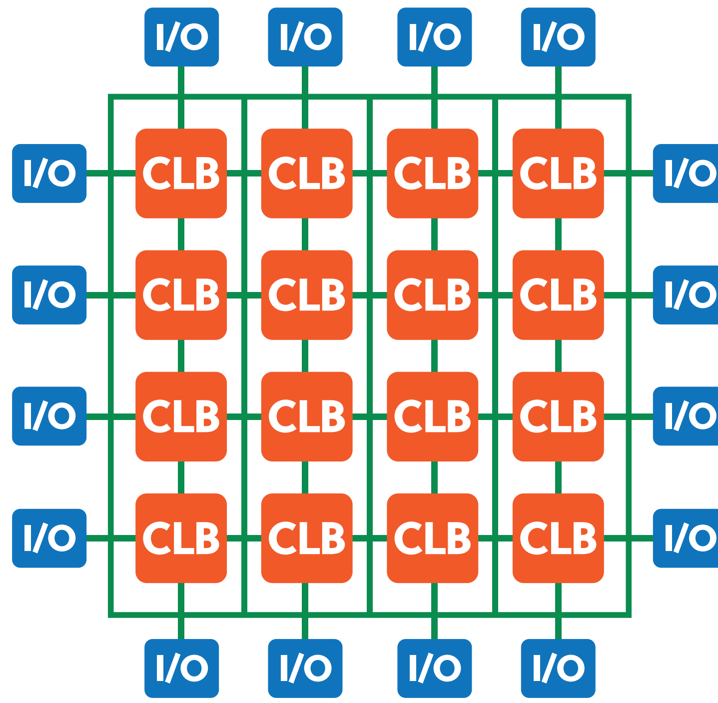


Figure 2: High-Level View of the FPGA Architecture [2]

In the landscape of FPGAs, Configurable Logic Blocks (CLBs) have a pivotal role in modifying digital logic to suit specific applications.

Configurable Logic Blocks (CLBs)

CLBs are the dynamic hubs where logic synthesis transforms abstract designs into tangible functionality. They house Look-Up Tables (LUTs) for combinational logic, flip-flops for sequential operations, and interconnects that constitute the fabric of programmable logic. Let's look at their composition, internal structure, and the ingenious mechanisms that empower users to craft tailored digital circuits.

According to digital logic fundamentals, any computation can be articulated as a Boolean equation, and in certain instances, as a Boolean equation where inputs rely on prior results—fear not, as FPGAs can indeed retain state. Consequently, every Boolean equation finds expression in a truth table. From these foundational principles, structures can be built to perform arithmetic operations like addition and multiplication as well as decision-making processes that assess conditional statements, exemplified by the classic if-then-else structure. By amalgamating these elements, we can articulate complex algorithms succinctly through the utilization of truth tables.

The Lookup Table

Considering this fundamental insight into digital logic, the truth table emerges as the computational core of the FPGA. Specifically, a hardware component adept at embodying a truth table is the lookup table, or LUT. LUTs are the key building blocks within a CLB. They are small, configurable memory

units that allow the storage and retrieval of logic functions. From a circuit implementation standpoint, a LUT can be easily fashioned through an N:1 (N-to-one) multiplexer and an N-bit memory. Conceptually, a LUT systematically enumerates a truth table. Thus, leveraging LUTs furnishes an FPGA with the flexibility to execute arbitrary digital logic. Figure 3 illustrates a standard N-input lookup table commonly found in contemporary FPGAs, with the majority of commercial FPGAs adopting the LUT as their fundamental building block. Even though many different CLB architectures do exist, this handbook will focus on the LUT-based CLB since it is the most popular implementation in FPGAs.

The LUT possesses the capability to compute any function of N inputs by programming the lookup table with the truth table corresponding to the desired function. As depicted in Figure 3, implementing a 3-input exclusive-or (XOR) function with a 3-input LUT (often denoted as a 3-LUT) involves assigning values to the lookup table memory in a manner that aligns the pattern of select bits with the correct row's "answer." Consequently, each "row" produces a result of 0, except in the four instances where the XOR of the three select lines yields 1.

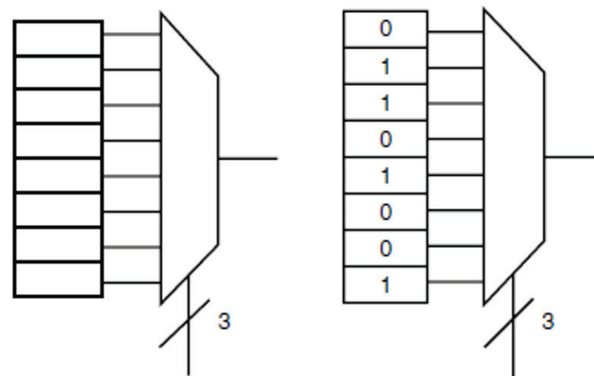


Figure 3: A 3-LUT schematic (a) and the corresponding 3-LUT symbol and truth table (b) for a logical XOR. [1]

Of course, more complicated functions – and functions of a larger number of inputs – can be implemented by aggregating several lookup tables together. For example, one can organize a single 3-LUT into an 8×1 ROM, and if the values of the lookup table are reprogrammable, an 8×1 RAM – but the basic building block, the lookup table, remains the same.

Flip-Flops and Multiplexers

We must now ask ourselves if this is enough to implement all the functionality we want in our FPGA. Indeed, it is not. With just LUTs, there is no way for an FPGA to maintain any sense of state, and therefore we are prohibited from implementing any form of sequential, or state-holding, logic. To remedy this situation, a simple single-bit storage element in our base logic block in the form of a D flip-flop is needed. A flip-flop is a fundamental building block in digital electronics, primarily used for storing binary information. It belongs to the category of bistable multivibrators, meaning it has two stable states and can store one bit of data. The two stable states are often denoted as "0" and "1". There are various types of flip-flops, with the most basic being the D flip-flop. The D flip-flop (Data

or Delay flip-flop) has a single data input (D), a clock input (C or CLK), and two outputs: Q (the normal output) and Q' (the inverted output). The data input (D) determines the state that the flip-flop will assume when the clock signal transitions from one level to another (e.g., from low to high).

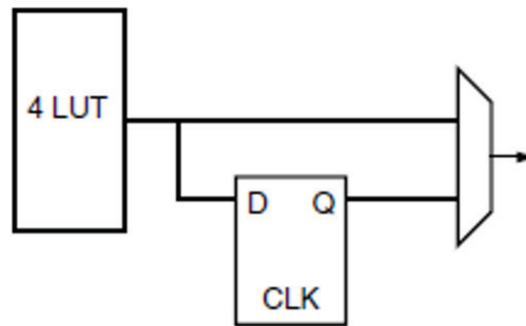


Figure 4: A simple lookup table logic block. [1]

Our logic block now adopts a configuration resembling that depicted in Figure 4. The output multiplexer makes a choice between the result derived from the function generated by the lookup table and the stored bit in the D flip-flop. A multiplexer, often abbreviated as "MUX," is a digital circuit component that plays a crucial role in data routing and selection within electronic systems. It is designed to take multiple input data lines and selectively route a particular input to the output based on control signals. In practice, this logic block closely mirrors those found in certain commercial FPGAs.

Most modern FPGAs are composed not of a single LUT, but of groups of LUTs and registers (flip-flops) with some local interconnect between them. Figure 5 illustrates a CLB with multiple LUTs. There has been research and ongoing debate over logic blocks containing groups of LUTs and their respective shapes and forms. Regarding the density and the speed produced by the CLBs. A particular study has shown that an FPGA containing two-thirds 4-input LUTs and one-third 2-input LUTs reduced the number of bits within the LUTs by 22% and the number of logic block pins by 10% when compared to FPGAs with only 4-input LUTs. [3]

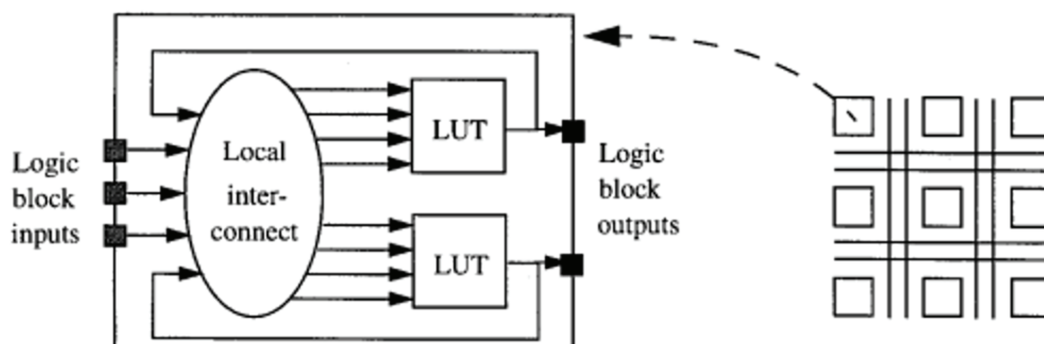


Figure 5: Abstract View of a Multiple LUT CLB [3]

Programmability

Examining the logic block depicted in Figure 4, it becomes straightforward to pinpoint all the programmable elements. These encompass the contents housed within the 4-LUT, the select signal directing the output multiplexer, and the initial state configuration of the D flip-flop. Presently, prevailing commercial FPGAs leverage volatile static-RAM (SRAM) bits linked to configuration points for FPGA configuration. Consequently, the configuration of the entire FPGA can be established by simply writing a value to each configuration bit.

Within our logic block, the 4-LUT comprises 16 SRAM bits, with each dedicated to an individual output; the multiplexer utilizes a solitary SRAM bit, and the initialization value for the D flip-flop can also be stored in a single SRAM bit. The way these SRAM bits are initialized within the broader context of the FPGA will be covered in subsequent chapters.

7-Series FPGA Example

Now that we have looked at the inner-workings of the CLB, we can look at the actual implementation of CLBs in modern AMD 7-Series FPGAs. An important point to note is that 7-Series AMD CLBs are made up of two individual slices. These slices contain different logic elements from each other, but both incorporate the technologies discussed above.

Every slice contains:

- Four logic-function generators (or look-up tables)
- Eight storage elements
- Wide-function multiplexers
- Carry logic

These elements are used by all slices to provide logic, arithmetic, and ROM functions. In addition, some slices support two additional functions: storing data using distributed RAM and shifting data with 32-bit registers. Slices that support these additional functions are called SLICEM; others are called SLICEL [4]. The complete schematic of the slices can be seen in [4].

A CLB element contains a pair of slices, and each slice is composed of four 6-input LUTs
and eight storage elements.

- SLICE(0) – slice at the bottom of the CLB and in the left column
- SLICE(1) – slice at the top of the CLB and in the right column

These two slices do not have direct connections to each other, and each slice is organized as
a column. Each slice in a column has an independent carry chain.

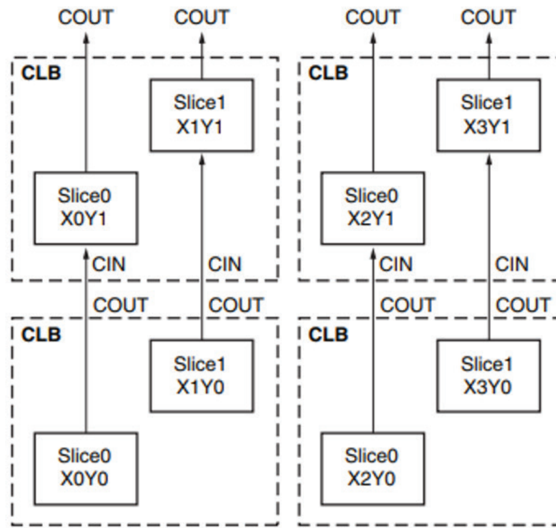


Figure 6: Shows four CLBs located in the bottom-left corner of the die [4]

7-Series CLB Physical Arrangement

The CLBs are arranged in columns in the 7-series FPGAs. The 7-series is the fourth generation to be based on the unique columnar approach provided by the ASMBL™ architecture. AMD introduced the Advanced Silicon Modular Block (ASMBL) architecture with the aim of creating FPGA platforms tailored to diverse application domains, each with optimized feature mixes. This innovative approach expands the range of available devices, empowering customers to choose FPGAs with the precise combination of features and capabilities that best suit their specific design requirements.

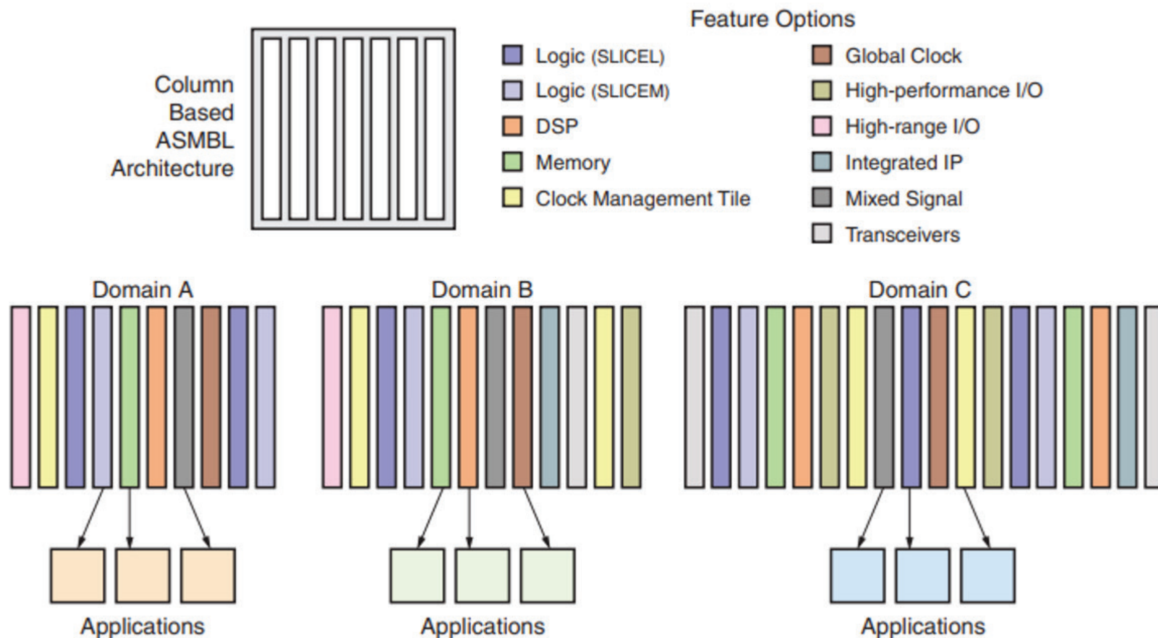


Figure 7: ASMBL Architecture [4]

Illustrated in Figure 7 is an overview of the various column-based resources associated with the ASMBL architecture. What sets ASMBL apart is its ability to overcome traditional design limitations by eliminating geometric layout constraints, such as dependencies between I/O count and array size. Additionally, it enhances on-chip power and ground distribution by allowing flexible placement anywhere on the chip. ASMBL further enables the scaling of disparate integrated IP blocks independently of each other and the surrounding resources, providing a flexible and versatile design framework.

Input/Output Blocks (IOBs)

Think of IOBs as gatekeepers or translators. They handle the incoming and outgoing signals, making sure they follow the right rules. Imagine different lanes on a road, each with its own set of traffic rules – IOBs ensure that signals of the same type follow the same rules.

IOBs are programmable, meaning you can customize their behaviour based on the needs of your specific application. They're designed to adapt to various electrical standards and interface with different external devices, making FPGAs highly flexible and suitable for a wide range of projects.

Today's FPGAs provide support for dozens of I/O standards thus providing the ideal interface bridge in your system. I/O in FPGAs is grouped in banks with each bank independently able to support different I/O standards. Today's leading FPGAs provide over a dozen I/O banks, thus allowing flexibility in I/O support.

IOB implementations vary from FPGA to FPGA and vendor to vendor, and thus in the next section a look at how 7-series FPGAs handle I/Os is given.

SelectIO

The I/O system on 7-series FPGA is called SelectIO™ and is defined in AMD User Guide UG471. The 7-series FPGAs have different types of I/O banks. There are the High-Performance (HP) ones and the High-Range (HR) ones. The HP banks are made to work better with fast memory and inter-chip connections. They handle voltages up to 1.8V. On the other hand, the HR banks support a broader range of input/output standards and can handle voltages up to 3.3V. So, depending on what you need, you can choose the type of I/O bank that fits best for your project. It is important to note that IO bank voltage depends on the application and the circuitry surrounding the FPGA. Some development boards may support changing the voltage on some banks, while other have a specific use-case and therefore use fixed bank voltages.

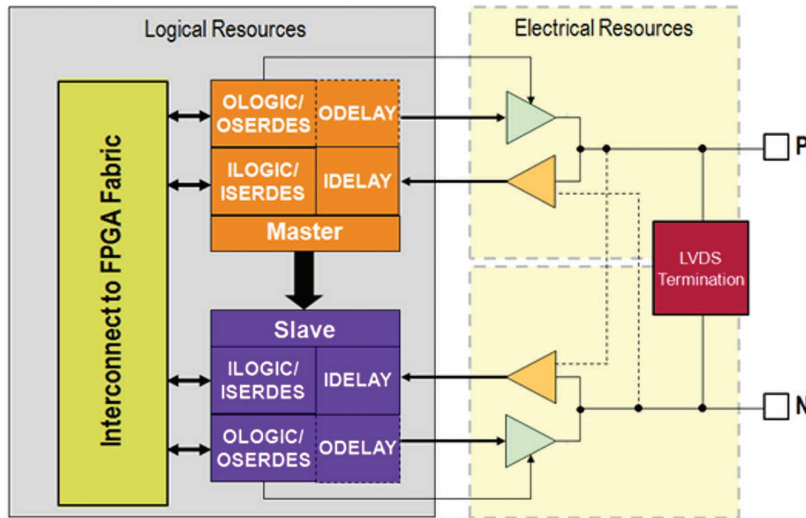


Figure 8: SelectIO Block Diagram

Programmable Interconnect

In the world of FPGAs, the programmable interconnect is like the wiring system that connects different parts of the FPGA. It's a network of pathways that you can configure and adjust based on what your electronic project needs.

Imagine it as the roads in a city. You can change the routes to connect different places, and you can do the same with the programmable interconnect in an FPGA. This flexibility is what makes FPGAs powerful. Instead of having a fixed layout like in traditional circuits, FPGAs allow you create your own pathways, allowing you to build custom electronic circuits tailored to your specific requirements. In the city analogy, FPGA programmable interconnect is like a customizable road system, giving the freedom to create and these roads in a fashion that is the most efficient, allows the most volume, or less distance, etc.

Now that we have an idea of how logic computation is achieved in FPGAs, we will go through the programmable interconnect and its functional description within FPGAs. Figure 9 below illustrates the current most popular implementation architecture in FPGAs, commonly called island-style architecture.

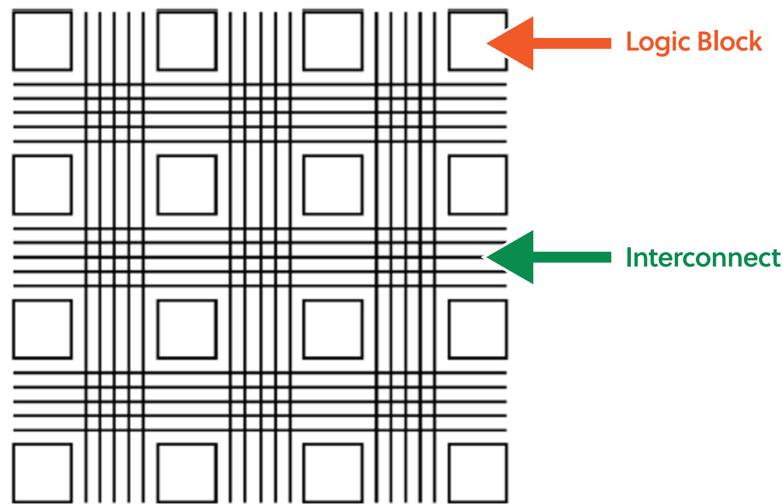


Figure 9: The island-style FPGA architecture. (The interconnect shown here is not representative of structures actually used.)

In this plan, there are puzzle-like building blocks scattered in a two-dimensional pattern and connected in a certain way. These building blocks are like islands, and they kind of float in a network of connections.

This layout lets us do calculations in a partitioned way on the FPGA. Big calculations are split into smaller pieces the size of a 4-LUT (a basic logic element) and put into these physical building blocks. The connections are set up to guide signals between the building blocks in the right way. If we have enough of these building blocks, we can make our FPGAs do any kind of calculation we want. It's like having many small parts that work together to create a big and powerful system. Looking at Figure 9 we can deduce that this is simply a visual placeholder to give us an idea of the internal structure of the FPGA. The actual internal architecture of FPGAs is more complex. In this section interconnect structures present in today's FPGAs are introduced.

Nearest Neighbour

Nearest-neighbour communication is as straightforward as it sounds. Imagine a 2x2 arrangement of logic blocks, just like in Figure 10. In this setup, each logic block only needs to connect with its immediate neighbours in four directions: north, south, east, and west. This means that every logic block can directly talk to the ones right next to it.

Figure 10 shows one of the simplest routing architectures possible. Even though it might seem basic, some older commercial FPGAs actually used this approach. However, this simple design has its drawbacks. It has issues with connectivity delays. Think of it this way: if instead of a small 2x2 setup, you had a huge 1024x1024 array, the delay would increase as you move further away. The signal must travel through many cells and switches to reach its final destination, causing delays and connectivity problems.

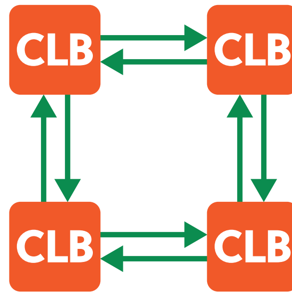


Figure 10: 2×2 Array of Logic Blocks

Here is where the need to bypass logic blocks arises. Without the ability to bypass logic blocks in the routing structure, all routes that are more than a single hop away require traversing a logic block. With just one pair of connections that work in both directions, there's a restriction on how many signals can cross in and out. Signals that are moving through must not interfere with signals that are actively being used and generated.

Because of these limitations, the nearest-neighbour structure isn't commonly used all by itself. However, it's almost always included in current FPGAs. Usually, it's combined with other techniques to overcome the challenges posed by its simplicity. One of the techniques that the nearest-neighbour structure is combined with is the segmented structure.

Segmented

Most of today's FPGA designs are less like Figure 10 and more like Figure 11. In Figure 11, we bring in what's called a Connection Block (CB) and a Switch Box (SB). This makes the routing structure more versatile and mesh-like.

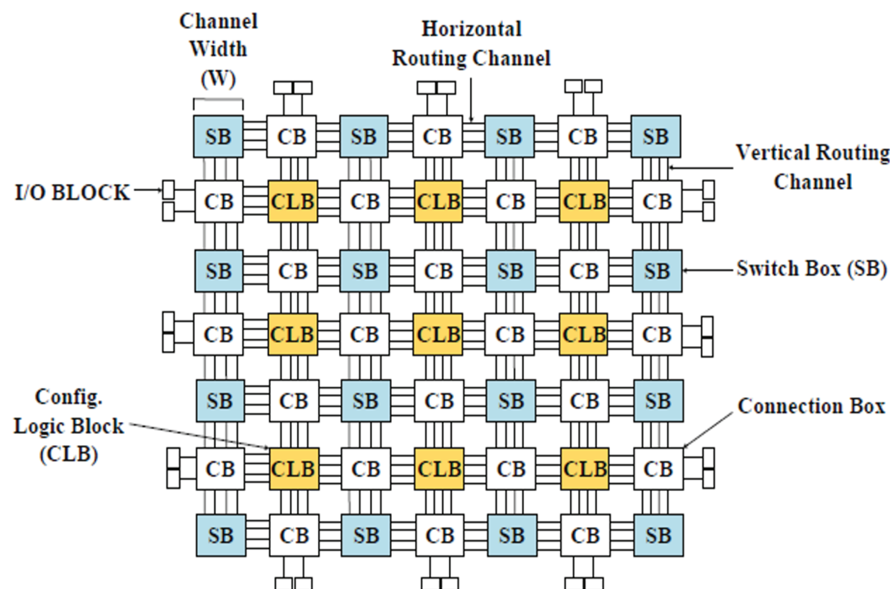


Figure 11: Illustration of a traditional island-style (mesh based) FPGA architecture with CLBs; the CLBs are "islands in a sea of routing interconnects". The horizontal and vertical routing tracks are interconnected through switch boxes (SB) and connection boxes (CB) connect logic blocks in the programmable routing network, which connects to I/O blocks. [2]

Here's how it works: the logic block can communicate with nearby resources through the connection block. This block links the input and output points of the CLB to the routing resources using programmable switches or multiplexers. The connection block, which you can see in more detail in Figure 12, lets you assign the inputs and outputs of the logic block to different horizontal and vertical tracks. This boosts the flexibility of how signals move around in the FPGA.

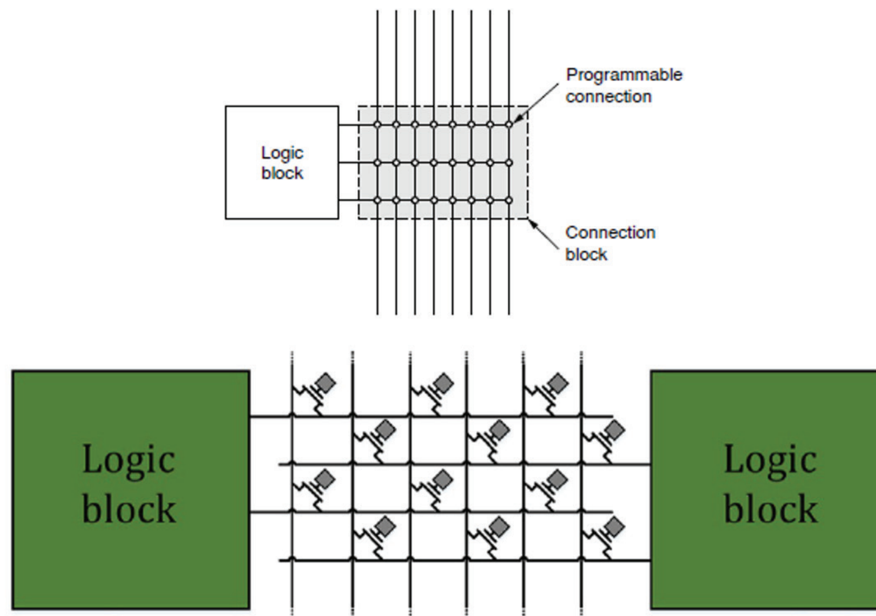


Figure 12: Detail of a connection block.

The switch block appears where horizontal and vertical routing tracks converge as shown in Figure 13. In the most general sense, it is simply a matrix of programmable switches that allow a signal on a track to connect to another track. SBs are placed at the intersection points of vertical and horizontal routing channels. Routing a net from a CLB source to the target CLB sink necessitates passing through multiple tracks and SBs, in which an entering signal from a certain side can connect to any of the other three directions based on the switch matrix (matrix of SBs) topology. The popular SB topologies in commercial FPGA architectures are Wilton, Disjoint, and Universal which are shown in Figure 14.

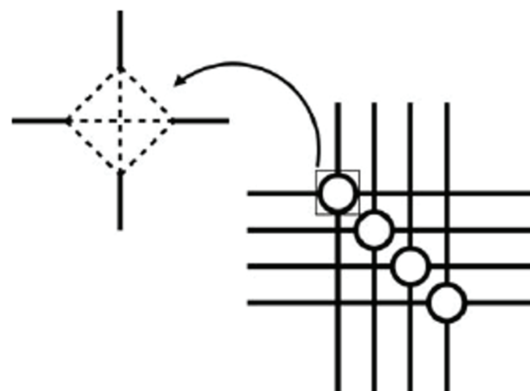


Figure 13: An example of a common switch block architecture.

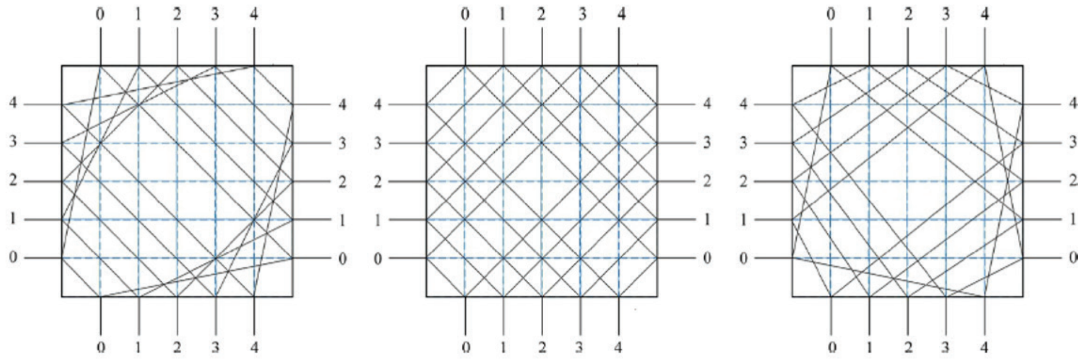


Figure 14: Three prevalent SM topology—Disjoint (left), Universal (middle), and Wilton (right).

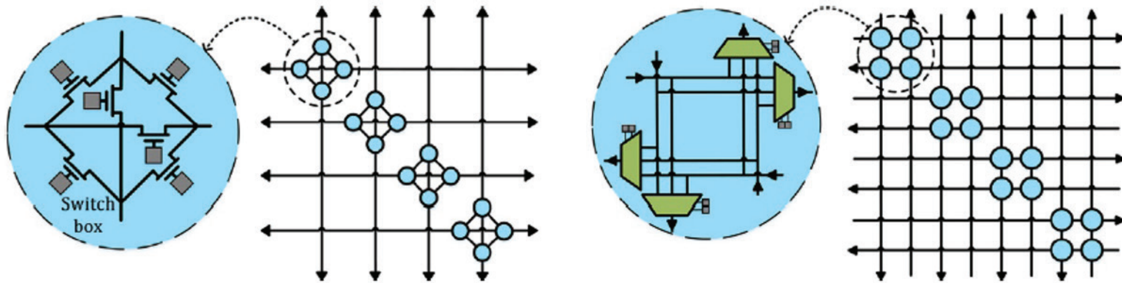


Figure 15: The structure of uni (left) and bi (right) Universal SM.

The connections between different parts of the FPGA can either be one-way (unidirectional) or two-way (bidirectional), and you can see examples of both in Figure 15. However, in modern FPGAs, the main setup is with one-way tracks. These one-way tracks can be either short or long. For instance, a wire that spans two Configurable Logic Blocks (CLBs) is a two-segment wire. Longer wires might take a bit more time to get through the multiplexer (SB) but are good for connecting things globally across the FPGA. On the other hand, shorter tracks have less delay, making them better for connecting things that are close by. So, depending on whether you need to connect things far or near, you might choose longer or shorter tracks.

Hierarchical

Here's another way to make long wires faster: a hierarchical approach. Look at the structure in Figure 16. At the lowest level, we group together 2x2 arrays of logic blocks into a single cluster. Inside this cluster, the routing is limited to local, nearest-neighbour connections. Now, we create a higher level by forming a 2x2 cluster of these smaller clusters, making a group of 16 logic blocks. At this level, longer wires at the edges of the smaller 2x2 clusters connect each cluster of four logic blocks to the other clusters in the higher-level group. We keep repeating this pattern at even higher levels, with larger clusters and even longer wires.

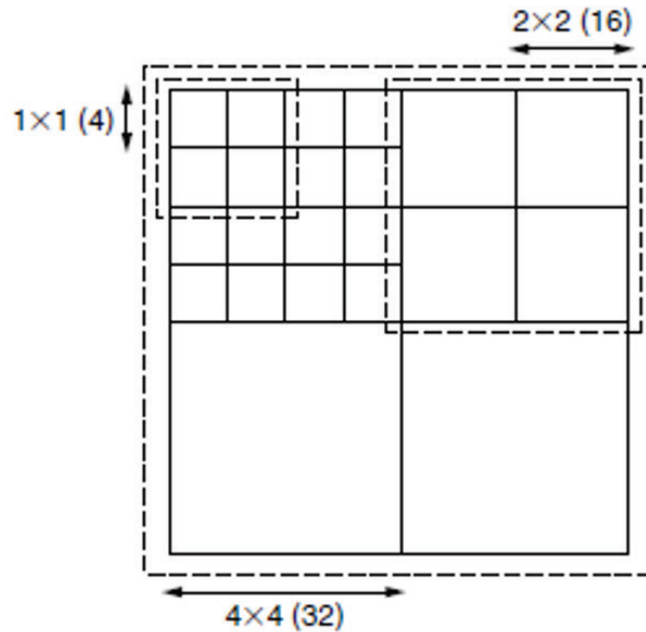


Figure 16: Hierarchical routing used by long wires to connect clusters of logic blocks.

This interconnect design relies on the idea that a well-designed (and well-placed) circuit mostly has local connections, and only a few connections need to travel long distances. By offering fewer resources at the higher levels, this design stays efficient in terms of space while still having some longer wires to speed up signals that need to cross large distances.

Hard Cores

Many new FPGA devices come with added features like specialized building blocks such as memory blocks (single or dual-port RAMs), multipliers and other arithmetic operations, and Digital Signal Processors (DSPs). These DSP and other dedicated blocks are designed and built into the devices to make it easier to implement specific functions. Without these specialized blocks, you would need a much larger number of Look-Up Tables (LUTs) to achieve the same functionality. They also provide a way to handle applications with high memory requirements.

Figure 17: The embedded DSP and Hard blocks inside FPGAs

However, certain design choices, like how often certain blocks are repeated in the architecture (as shown in Figure 17), are crucial. This repetition frequency is a key design parameter that affects the overall performance and energy efficiency of the FPGA. The way these architectural elements are configured plays a significant role in determining how well the FPGA can perform specific tasks and how efficiently it uses energy.

Embedded Resources

Even though CLBs are a very important and powerful tool in FPGAs, they can easily be overused when trying to implement structures such as memory, shift registers and arithmetic operations. That is why all modern FPGAs have specific embedded resources that target these challenges.

Data Storage

Data storage is very common and important in digital system design. Apart from the SLICEMs in the CLBs of 7-series FPGAs, which can be used as memories or shift registers, FPGAs also have something called Block RAMs (BRAM) embedded in their hardware. These are bigger storage parts. In the 7-series, all the parts have 36 Kb BRAM, each of which can be divided into two 18 Kb BRAMs. The table below shows how much BRAM is in the parts on the suggested development boards. These BRAMs are not exclusively found in 7-series FPGAs but are common to all modern FPGAs. The following table lists the BRAM resources available on two Digilent boards.

Board	Device	Total 36Kb BRAM
Basys 3	XC7A35T	50
Nexys 7 A7	XC7A100T	135

Table 1: BRAM Available in Two FPGA Boards

These BRAMs can all be configured as follows:

- A single port.
- True dual port memories – Two read/write ports.
- Simple dual port memories – 1 read/1 write. In this case, a 36 Kb BRAM can be up to 72 bits wide and an 18 Kb BRAM up to 36 bits wide.

The information stored in BRAMs can be set up upon initialization, and you can adjust it using a file or a special part in the code. This comes in handy when creating things like ROMs or setting up initial conditions.

In 7-series devices, the BRAMs also have some built-in logic to create FIFOs (First-In First-Out). This is useful because it helps save resources in the CLBs, and it makes the design process smoother by avoiding some technical issues.

All 36 Kb BRAMs come with something called Error Correction Code (ECC) functions. This is more about ensuring things work reliably, like in medical, automotive, or space applications. However, we won't get into the details of that in this handbook.

In addition to the embedded BRAMs 7-series FPGAs also offer an on-chip high speed memory interface which goes up to 1,866 Mb/s on Virtex-7 FPGAs.

Digital Signal Processing Blocks

Performing arithmetic operations in an FPGA can be quite costly in terms of the FPGA resources discussed until now. In Application Specific Integrated Circuits (ASICs), the most expensive and time-consuming task is usually multiplying numbers, while adding is quicker and less demanding. To handle this, FPGA makers have been putting in dedicated arithmetic cores directly into the fabric of the FPGA for many years. This flips things around in an FPGA – now, adding numbers can become the slower task, especially when dealing with wider numbers. This happens because the multiplication process has been turned into a complex and pipelined operation. One of the most common arithmetic operations in digital signal processing (DSP) is called the Multiply and Accumulate (MAC) operation. This is used extensively in DSP, one perfect example would be filtering, FIR filter implementations in FPGA hardware make use of MAC operations, the higher order the filter, the more MAC operations are needed.

FPGAs have special parts called DSP blocks or slices. These DSP blocks help speed up common tasks like fast Fourier transforms (FFTs) and finite impulse response filtering (FIR), which are related to

processing signals and require a large number of arithmetic operations (Multiply, Divide, Add, Subtract, etc.). DSP slices are not only for multiplying numbers—they can do more. The DSP slices make lots of things work faster and better in many applications, not just digital signal processing. They help with tasks like handling big buses that move data around, creating memory addresses, combining different data paths, and dealing with input and output registers that are linked to memory locations.

You can also perform operations such as multiplication using regular logic (LUTs and flip-flops), but it uses up a lot of resources. Using the special DSP blocks for multiplication makes sense because it's better for performance and using logic efficiently. That's why even small FPGAs set aside space for DSP blocks.

FPGA	DSP Block Multiplier Width
Altera Cyclone V	27 x 27 bit
Lattice iCE40UP (SB_MAC16)	16 x 16 bit
Lattice ECP5 (sysDSP):	18 x 18 bit
AMD 7-series (DSP48E1)	25 × 18 bit
AMD Ultrascale+ (DSP48E2)	27 x 18 bit

Table 2: DSP Block Multiplier Width for Various FPGAs

DSP48E1

FPGAs excel in digital signal processing (DSP) tasks because they can use special, fully parallel methods that are customized for specific needs. DSP operations often involve a lot of binary multiplication and accumulation, and FPGAs have dedicated parts called DSP slices that are perfect for these tasks. In the 7-series FPGAs, there are plenty of these custom-designed, low-power DSP slices that are fast, compact, and still flexible for designing different systems. The figure below [5] illustrates the basic DSP48E1 Slice functionality in 7-series FPGAs.

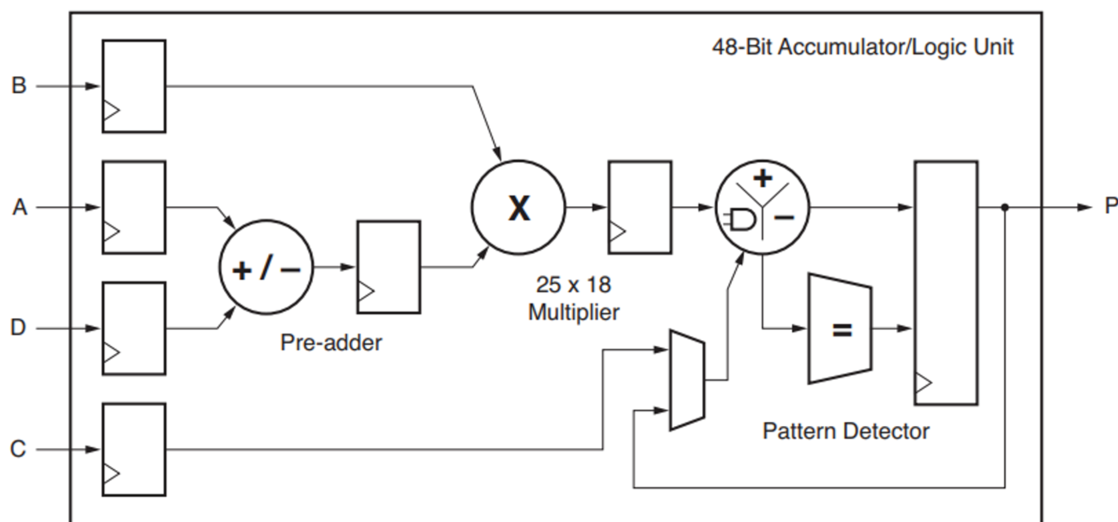


Figure 18: Basic DSP48E1 Slice Functionality [5]

The DSP functionality in question offers several notable features. First, it includes a 25 × 18 two's-complement multiplier with dynamic bypass and a 48-bit accumulator that can function as a synchronous up/down counter. There's also a power-saving pre-adder designed to optimize symmetrical filter applications and reduce DSP slice requirements. The single-instruction-multiple-data (SIMD) arithmetic unit supports dual 24-bit or quad 12-bit add/subtract/accumulate operations, along with an optional logic unit capable of generating ten different logic functions of the two operands. Additional capabilities involve a pattern detector, convergent or symmetric rounding, and the ability to perform 96-bit-wide logic functions when used alongside the logic unit. Advanced features such as optional pipelining and dedicated buses for cascading further enhance the versatility of this DSP functionality.

Peripherals

Peripherals in FPGAs refer to external devices or components that can be connected to the FPGA to enhance its functionality. These peripherals can include input/output interfaces, communication ports, memory modules, sensors, and other hardware components that extend the capabilities of the FPGA. They enable the FPGA to interact with the external world, process data from various sources, and perform specific tasks based on the application's requirements. Integrating peripherals allows FPGAs to be customized for a wide range of applications and makes them adaptable to different tasks and environments. Modern FPGA chips are also incorporating peripherals so that implementation of certain functions such as inter chip communications. These do not have to be implemented using the available logic in the FPGA.

Connectivity

Connectivity in modern digital processing platforms, particularly in 7-series FPGAs, extends beyond the chip itself to encompass peripheral circuitry connected to FPGA I/Os. In the realm of digital design, establishing communication with external devices is often facilitated by incorporating components like Ethernet PHYs and USB controllers. These peripherals offer simpler interfaces to the FPGA, enabling seamless connectivity with various devices. For instance, utilizing a USB controller can simplify the implementation of interfaces like USBUART serial communication, reducing the burden on FPGA resources. This strategic offloading of functionalities to dedicated peripheral circuitry not only streamlines the design process but also optimizes the utilization of valuable FPGA resources for more complex and specialized tasks. In essence, the integration of these peripheral components enhances the overall connectivity of the FPGA-based system, fostering efficient communication with the outside world.

Analog to Digital Converters

Analog-to-Digital Converters (ADCs) are electronic devices that convert continuous analog signals into discrete digital representations. In other words, they transform real-world signals, such as those from sensors, audio devices, or other analog sources, into digital data that can be processed by digital systems like microcontrollers, computers, or digital signal processors. 7-series FPGAs include an on-chip user configurable analog interface (XADC), incorporating dual 12-bit 1MSPS

analog-to-digital converters with on-chip thermal and supply sensors. If the specifications of on-chip ADCs are insufficient for certain applications, external ADCs are commonly used.

Clock Management Resources

FPGA clock management resources play a crucial role in controlling and distributing clock signals throughout the device. These resources help manage the timing of various components within the FPGA, ensuring synchronization and proper operation. Clock management features include phase-locked loops (PLLs) and delay-locked loops (DLLs) that enable precise control over clock frequencies, skew, and jitter. PLLs offer flexibility by allowing the generation of multiple clock frequencies from a single reference clock. This capability is vital for accommodating diverse requirements within a design. FPGA designers leverage these clock management resources to optimize performance, meet timing constraints, and enhance the overall reliability of digital circuits implemented on the FPGA.

One very important aspect to consider when designing with FPGAs is the clock skew. Clock skew refers to the variation in arrival times of a clock signal at different points within a digital system. In other words, it's the difference in time it takes for the clock signal to reach different parts of a circuit. In synchronous digital systems, various components rely on the same clock signal to coordinate their operations. However, due to factors such as differences in wire lengths, routing paths, and environmental conditions, the clock signal may not reach all components simultaneously.

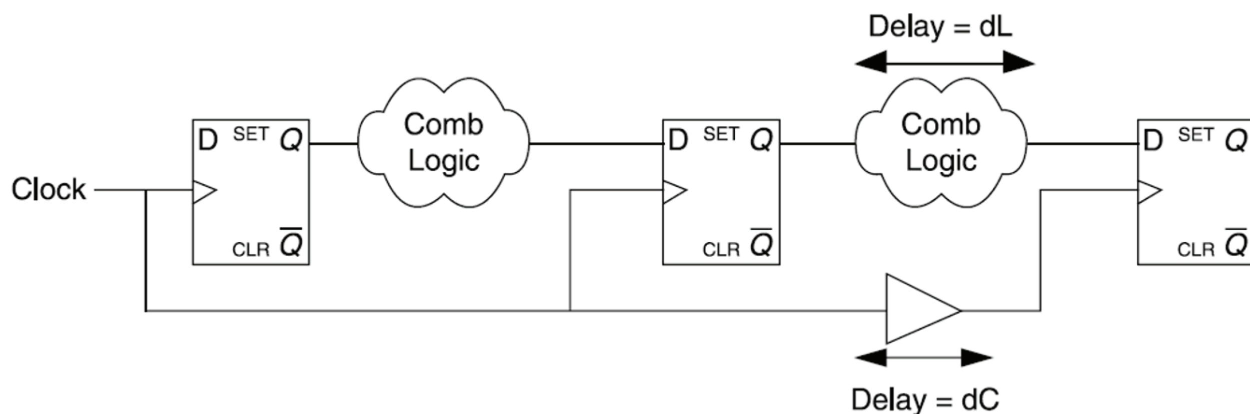


Figure 19: Clock Skew

Clock skew can lead to timing issues and negatively impact the reliability and performance of a digital circuit. Excessive clock skew may result in some components latching data at different times, causing data corruption and errors. Designers use techniques like careful routing, buffer insertion, and clock tree synthesis to minimize clock skew and ensure that the clock signal reaches different parts of the circuit as simultaneously as possible. Minimizing clock skew is particularly important in high-performance digital systems to maintain accurate synchronization.

It is relevant to mention here that in the context of AMD Vivado, "negative slack" refers to a critical

timing issue in your FPGA design. Slack is a measure of the timing margin, indicating how much time is available (positive slack) or how much time is violated (negative slack) between the arrival of a signal at a destination (e.g., a flip-flop) and the required time for the signal to meet setup or hold requirements.

Negative slack occurs when the design fails to meet timing constraints, meaning that certain paths in the design are not meeting the required timing specifications. This can be problematic because it may lead to incorrect functionality, reduced performance, or even complete failure of the design.

FPGA clock management resources when used correctly enable the designer to meet timing constraints and make sure to minimize these effects in FPGA systems. Sometimes designers tend to not prioritize the clock effects in their design. But the effects of mismanaged clock signals throughout an FPGA can lead to designs being ineffective and show intermittent errors which are not desirable. In the following sections the clock management tools available in modern FPGAs are discussed.

Clock Sources

Internal oscillators within FPGAs offer on-chip solutions for generating clock signals, eliminating the need for external clock sources and simplifying the design process. These on-chip oscillators provide stable and precise clock signals with low jitter and configurable frequencies to meet the timing requirements of digital circuits within the FPGA. They contribute to power efficiency and play a pivotal role in defining clock domains, enabling heterogeneous designs with varied clock frequencies. While internal oscillators provide a convenient option for many applications, it's important to note that in certain cases, especially those with stringent timing requirements or specialized needs, most FPGAs still utilize external oscillators for greater precision or specific frequency characteristics. The choice between internal and external oscillators depends on the specific design considerations and performance criteria of the FPGA application.

External oscillators for FPGAs serve as standalone clock sources positioned outside the FPGA device itself. These oscillators are preferred in applications that demand a high degree of precision and stability in clock signals. Selected based on specific frequency requirements, external oscillators are characterized by low jitter and accurate frequency control, making them suitable for scenarios where standard on-chip oscillators may not meet the desired frequencies or specialized clock characteristics. Often taking the form of crystal oscillators, these external sources utilize crystal resonators to generate highly stable clock signals. They are frequently integrated into the broader clock distribution network, providing a common reference for multiple FPGA devices or other components within a larger system. External oscillators play a crucial role in applications requiring synchronized clocks across different components or boards, contributing to coherent operation in systems with distributed timing needs. Configuration options allow designers to tailor external oscillators to the specific frequency and settings requirements of their FPGA application. While internal oscillators within FPGAs offer simplicity and integration, the choice between internal and external oscillators depends on the precision and customization demands of the particular

application. All Digilent boards rely on external oscillators.

Phased-Locked Loops

A Phase-Locked Loop (PLL) is an electronic feedback system designed to control the phase of an output signal in relation to a reference signal. It is commonly used in electronics and communication systems for tasks such as clock synchronization, frequency synthesis, and demodulation. PLLs offer clock multiplication and division, phase shifting, programmable duty cycle, and external clock outputs, allowing system-level clock management and skew control.

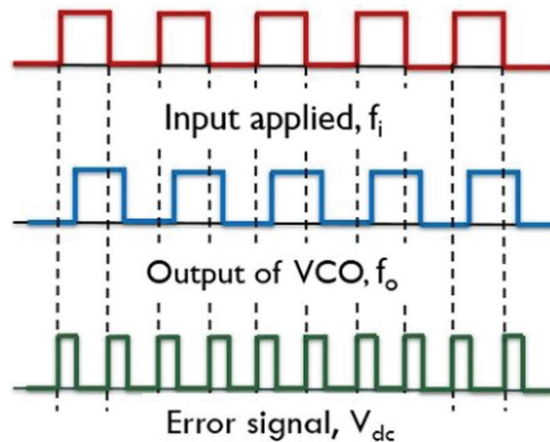


Figure 20: Example Waveforms for a VCO Error Signal

Clock Buffers

Clock buffers are essential components in digital systems, tasked with the efficient distribution of clock signals across various components of a circuit. Their primary function is to replicate an input clock signal and deliver it to multiple output locations within a system. This is particularly critical in large-scale digital designs where synchronous operation is paramount. Clock buffers are designed to handle multiple outputs, a property known as fanout, without compromising the integrity of the clock signal. They come in different types, including non-inverting and differential buffers, each serving specific purposes. Non-inverting buffers maintain the same logic level as the input, while differential buffers transmit the clock signal as a complementary signal pair, offering enhanced noise immunity. Clock buffers play a crucial role in minimizing clock skew, the variation in arrival times of the clock signal at different points in the circuit, ensuring synchronized operation. Additionally, they may provide control over edge rates, influencing the speed of transitions between logic levels in the output signals. In FPGA and ASIC designs, where precise clock control is essential, clock buffers contribute to meeting timing requirements and facilitating reliable digital system operation.

Clock Regions

Clock regions in FPGAs serve as designated areas within the device where clock resources are organized and managed. As FPGAs comprise numerous programmable logic cells and dedicated

clock distribution networks, clock regions play a vital role in orchestrating the distribution of clock signals. In practical terms, they help establish and control various clock domains, ensuring synchronous operation within specific regions. This is particularly important for managing clock skew, the variation in arrival times of clock signals at different locations. Clock regions enable the optimization of clock distribution for different parts of the FPGA, accommodating diverse clock frequencies within a single device. Design tools provided by FPGA vendors leverage information about clock regions to enhance the placement and routing of logic elements and clock resources, ultimately contributing to better performance, efficient resource utilization, and the successful fulfilment of timing constraints in complex systems.

Why are Clock Regions so Important?

Clock management resources in FPGA design are crucial for ensuring the reliable and efficient operation of digital circuits. Let's examine a practical example to illustrate why understanding and effectively utilizing these resources are paramount, especially for new users.

Consider the implementation of DDR (Double Data Rate) memory interfaces using Memory Interface Generator (MIG) in AMD Vivado. DDR memory interfaces, commonly used in many applications for higher data transfer rates, have stringent timing requirements. These interfaces demand precise control over clock signals to avoid errors in tooling and maintain correct design functionality. In DDR interfaces, the data is transferred on both the rising and falling edges of the clock signal, doubling the effective data transfer rate. This introduces challenges related to clock skew, where the arrival times of clock signals at different points in the system need to be tightly controlled to meet the timing constraints imposed by the DDR standard.

Here's where clock management resources become instrumental:

1. Phase-Locked Loops (PLLs):
   a. DDR interfaces often require precise control over the clock frequency to ensure data is sampled correctly. PLLs in FPGAs allow designers to generate stable and precisely controlled clock frequencies, meeting the tight requirements of DDR timing.
2. Clock Buffers:
   a. Clock buffers play a crucial role in minimizing clock skew. In DDR designs, where synchronization is critical, clock buffers help replicate and distribute the clock signal efficiently across the memory interface, ensuring that the rising and falling edges align properly with data transitions.
3. Clock Regions:
   a. DDR interfaces often operate at different clock domains within an FPGA. Clock regions help manage and organize these domains, optimizing the distribution of clock signals. This is essential for avoiding issues related to clock domain crossings and ensuring reliable DDR interface operation.

In the context of DDR timing, mismanagement of clock resources can lead to errors in the tooling process, incorrect design functionality, and compromised performance. New users, in particular,

should pay attention to leveraging PLLs, clock buffers, and understanding clock regions to meet the specific requirements of DDR memory interfaces.

By effectively utilizing FPGA clock management resources, designers can navigate the intricacies of DDR timing, optimize performance, and ensure a robust and error-free implementation of memory interfaces. This example underscores the practical significance of mastering clock management in FPGA design, especially for applications with stringent timing constraints like DDR interfaces.

Clock Management in 7-series FPGAs

The clocking resources in 7-series FPGAs effectively manage a spectrum of complex and straightforward clocking requirements, utilizing dedicated global and regional I/O and clocking resources. Clock Management Tiles (CMT) play a crucial role in providing functionalities such as clock frequency synthesis, deskew, and jitter filtering. It is emphasized that non-clock resources, including local routing, are discouraged in designs focused on clock functions. The global clock trees enable synchronous element clocking across the entire device, while I/O and regional clock trees allow clocking of up to three vertically adjacent clock regions. Each CMT, located in the CMT column next to the I/O column, houses one Mixed-Mode Clock Manager (MMCM) and one Phase-Locked Loop (PLL). The division of each 7-series device into clock regions is fundamental for clocking purposes, with the number of clock regions scaling with the device size, ranging from one in the smallest to 24 in the largest. A clock region encompasses all synchronous elements, spanning 50 CLBs and one I/O bank, with a central Horizontal Clock Row (HROW). Each clock region extends 25 CLBs up and down from the HROW and horizontally across each side of the device.

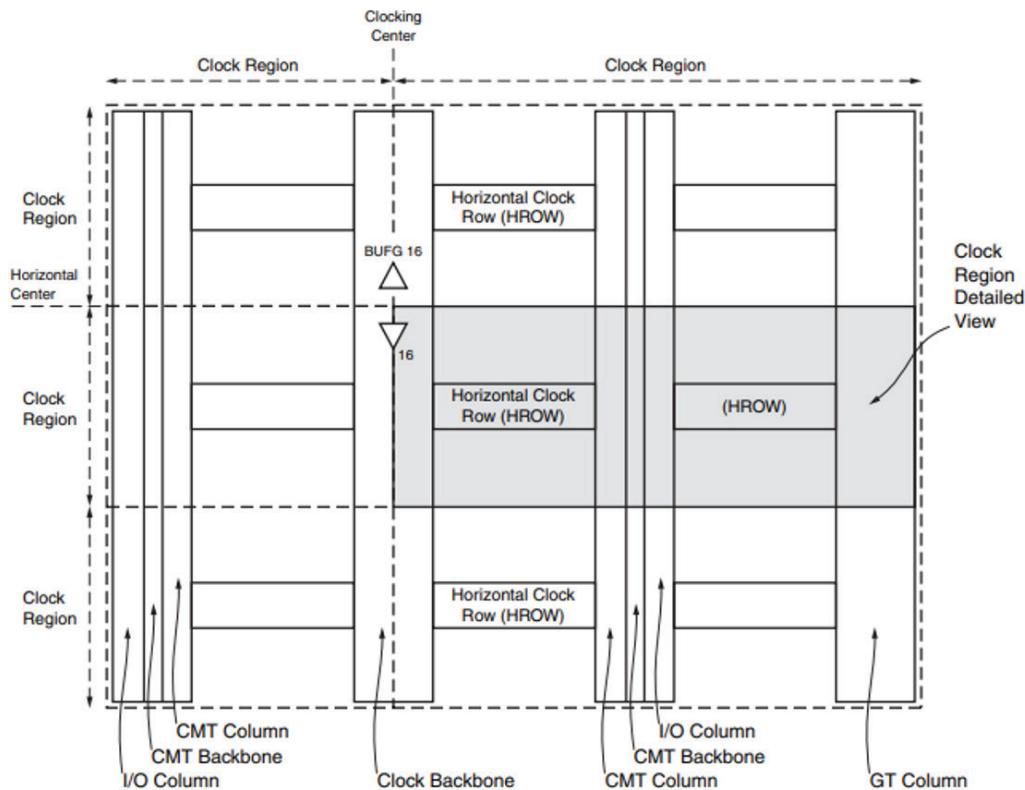


Figure 21: 7-series FPGA High-Level Clock Architecture View [6]

The MMCM and PLL are a lot alike. They can both help synthesize different frequencies and clean up the timing of incoming signals. Inside both is something called a voltage-controlled oscillator (VCO), which speeds up or slows down based on the input it gets. There are three sets of programmable dividers: D, M, and O. D makes the input slower, M makes it faster, and O divides the output into smaller parts. You need to pick the right values for D and M so that the VCO stays in the right frequency range. The VCO has eight different phases, and you can choose one of them to divide and create the output signal. Both the MMCM and PLL have three options to filter out jitter in the input: low, high, or optimized. Low is best for reducing jitter, high is best for reducing timing errors, and optimized lets the tools figure out the best setting.

Each 7-series FPGA provides six different types of clock lines (BUFG, BUFR, BUFIO, BUFH, BUFMR, and the high-performance clock) to address the different clocking requirements of high fanout, short propagation delay, and extremely low skew.

In every 7-series FPGA (except XC7S6 and XC7S15), there are 32 global clock lines with the widest reach, capable of extending to every flip-flop clock, clock enable, set/reset, and many logic inputs. Within each clock region, governed by horizontal clock buffers (BUFH), there are 12 global clock lines. Each BUFH can be independently enabled or disabled, offering the flexibility to turn off clocks within a specific region, providing precise control over power consumption. These global clock lines can be connected to global clock buffers, which have the additional capabilities of glitchless clock multiplexing and managing clock enable functions. The source of global clocks is often the Clocking Management Tile (CMT), which has the potential to entirely eliminate basic clock distribution delays.

Regional clocks can drive all clock destinations in their region. A region is defined as an area that is 50 I/O and 50 CLB high and half the chip wide. 7-series FPGAs have between two and twenty-four regions. There are four regional clock tracks in every region. Each regional clock buffer can be driven from any of four clock-capable input pins, and its frequency can optionally be divided by any integer from 1 to 8. To repeat, it's rarely advised to use signals that are not driven by clocking-related resources as if they are clocks!

For more details on the 7-series FPGAs clocking resources take a look at UG472 [6].

Configuration and Programming

Up until this section, this handbook has given a detailed look into the inner workings of FPGAs. Configuration and programming are the steps used in order to use the internal hardware resources of the FPGA to implement custom digital applications effectively and efficiently.

In FPGA terminology, configuration refers to the process of initializing the FPGA with a specific set of instructions. These instructions, stored in a configuration memory, dictate the interconnections and functionalities of the FPGA's internal logic elements. Configuration occurs during startup and is crucial for defining the FPGA's operational characteristics. Remember FPGAs lose their configuration on loss of power, so they must be configured upon every power cycle. That is why some type of

configuration memory is required.
Programming an FPGA involves a series of systematic steps:

1. Define the Objective: Clearly articulate the desired functionality or task the FPGA is intended to perform. This forms the basis for subsequent programming steps.

2. Hardware Description Language (HDL): Utilize HDL, such as Verilog or VHDL, to describe the desired circuitry and behavior. HDL serves as the intermediary language between human-readable code and the low-level hardware description.

3. Compilation Process: The HDL code undergoes synthesis and implementation processes using specialized tools. Synthesis translates the high-level HDL code into a netlist, representing the logical structure. Implementation maps this netlist onto the physical resources of the FPGA, considering factors like timing and resource utilization.

4. Bitstream Generation: The compiled design is converted into a bitstream – a binary file containing configuration data. This bitstream is analogous to the firmware for configuring the FPGA.

5. Configuration Upload: The bitstream is loaded onto the FPGA's configuration memory, effectively programming the device. This step is typically carried out during the power-up sequence.

A more detailed description of the configuration and programming of FPGAs is given in section 3 FPGA Design Flow.

The Zynq and other SoC

FPGAs are amazing and when used well can beat most of the available digital platforms in terms of speed and efficiency. However, the need for more peripherals and to have additional digital processing power have arisen as time went on. In fact, several System on Chip (SoCs) have made it to market. These SoCs normally incorporate processor cores coupled with an FPGA and various peripherals. This provides the ease of use of sequential processors with peripherals and the power of the FPGA.

The Zynq system-on-chip (SoC) is a notable example of a versatile and powerful integrated circuit that combines the capabilities of both a traditional processor system and programmable logic within a single chip. Developed by AMD, the Zynq SoC family integrates a Processing System (PS) based on ARM Cortex-A9 cores with programmable logic (PL) in the form of an FPGA (Field-Programmable Gate Array). This unique combination enables designers to harness the flexibility of programmable logic alongside the processing power of traditional CPUs, making it well-suited for a broad range of applications.

Multiplexed IO (MIO) and Extended Multiplexed IO (EMIO) are pivotal features in AMD Zynq FPGAs, enhancing connectivity and integration capabilities. MIO enables flexible routing of Peripheral I/O (PIO) pins from the PS to on-chip peripherals, facilitating versatile interfacing with external components. On the other hand, EMIO takes this flexibility a step further by enabling communication between PS peripherals and the FPGA fabric, allowing for seamless integration of PS peripherals, such as UARTs, with custom IP blocks instantiated in the FPGA. A practical application of EMIO is using a PS UART to communicate with a UART module instantiated in the FPGA, providing a bridge between PS and FPGA logic and enabling innovative system-level designs. These features empower designers to create adaptable and integrated systems by offering flexible routing options for PS peripheral I/O pins and facilitating communication between PS peripherals and the FPGA fabric, enhancing overall connectivity in Zynq-based FPGA designs. UART communication with Digilent Zynq boards typically occurs through PS UART peripherals instead of through FPGA pins.

The ARM Cortex-A9 cores within the Zynq SoC handle general-purpose processing tasks, running operating systems such as Linux or other real-time operating systems (RTOS). These cores are responsible for executing high-level software applications, interfacing with peripherals, and managing system-level operations. Concurrently, the programmable logic section of the Zynq chip provides a customizable hardware platform that can be tailored to specific tasks or applications, offering a performance boost for parallelizable and compute-intensive operations.

Other system-on-chip architectures share a similar integration concept, combining processing elements with programmable logic to create a holistic solution. Examples include Intel's Cyclone and Arria SoC families, which integrate ARM Cortex-A9 or ARM Cortex-A53 cores with FPGA fabric, allowing for diverse application implementations. These SoCs find applications in fields ranging from telecommunications and automotive to industrial automation and edge computing.

The integration of processing cores and programmable logic in SoCs has become a trend in modern embedded systems, providing a balance between the flexibility of software and the performance of dedicated hardware. Designers can optimize their systems by leveraging the strengths of both components, tailoring solutions to meet specific requirements and achieve a competitive edge in terms of performance, power efficiency, and adaptability.

Chapter Two Summary

At the core of FPGA architecture lies a versatile framework that combines programmable logic blocks with configurable routing and clock management resources. This unique blend allows designers to implement custom digital circuits efficiently and rapidly adapt to evolving project requirements. The architecture supports diverse applications through the integration of peripherals like communication interfaces, memory controllers, and specialized IP cores. Understanding FPGA architecture empowers engineers to leverage its flexibility for a wide range of tasks, from digital signal processing to embedded system design.

Chapter Three: FPGA Design Flow

The FPGA design flow is a systematic process that transforms a conceptual hardware description into a fully functional and optimized FPGA implementation. This journey involves a series of well-defined stages, each contributing to the realization of a digital design within the constraints and capabilities of an FPGA. From conceptualization to synthesis, place and route, and finally bitstream generation, the FPGA design flow encompasses various critical steps. Each stage involves intricate decisions related to architecture, timing, power, and resource utilization, requiring designers to strike a balance between performance, flexibility, and efficiency. In this section, we delve into the intricacies of the FPGA design flow, exploring the key processes and considerations that engineers navigate to bring their digital designs to life on programmable hardware.

What to expect in Chapter Three:

- An overview of design flow
- Hardware Description Languages
- Synthesis and Optimization

Overview of Design Flow

The FPGA design flow is a step-by-step process that transforms a high-level hardware description into a configuration bitstream that can be loaded onto a FPGA. Here's an overview of the FPGA design flow:

Source Code:

The design process begins with the creation of a hardware description using a hardware description language (HDL) such as Verilog or VHDL. This source code describes the intended functionality of the digital circuit. An initial design may also be further abstracted above HDLs using other tools, such as block diagrams or high-level synthesis.

Logic Synthesis:

Logic synthesis is the process of converting the high-level HDL code into a netlist of logical gates and flip-flops. This stage involves optimizing the design for factors such as performance, area, and power.

Technology Mapping:

Technology mapping involves mapping the logical gates in the synthesized netlist to specific

resources available in the target FPGA technology. This step considers the architecture of the FPGA and aims to match the design's logic to the available programmable resources.

Placement:

Placement involves determining the physical location of each logic element on the FPGA. The goal is to place critical elements close to each other to minimize delays and optimize performance. Proper placement is crucial for meeting timing requirements.

Routing:

After placement, the routing step involves creating the interconnections (wires) between the placed logic elements. The router determines the optimal paths for signals, considering factors such as signal delays, avoiding congestion, and meeting timing constraints.

Bitstream Generation:

Once the design is placed and routed, the final step is to generate the bitstream. The bitstream is a binary file that contains configuration information for the FPGA. It specifies how the programmable elements (look-up tables, flip-flops, etc.) should be configured to implement the desired logic.

Throughout the FPGA design flow, designers use various tools, such as synthesis tools, place-and-route tools, and vendor-specific tools provided by FPGA manufacturers like AMD (or Altera) The iterative nature of the design flow allows designers to refine and optimize their designs at each stage, balancing factors like performance, resource utilization, and power consumption.

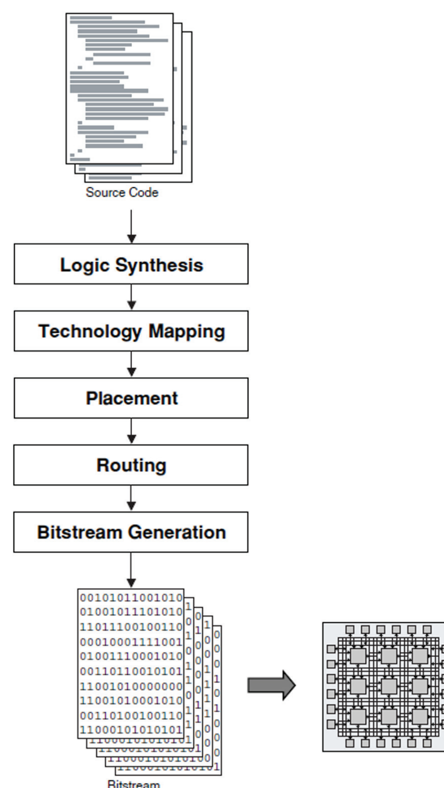


Figure 22: A typical FPGA mapping flow. [1]

Hardware Description Languages (HDLs)

Hardware Description Languages (HDLs) are specialized programming languages crucial for digital design, providing a comprehensive framework for expressing the behavior and structure of electronic circuits. These languages, such as Verilog and VHDL, serve different communities and industries but share fundamental principles. Verilog, with its C-like syntax, and VHDL, with a more verbose and standardized syntax, offer designers a choice based on preference and application requirements.

HDLs support multiple levels of abstraction. Behavioral HDLs focus on specifying the functionality of a system without detailing its implementation, leveraging constructs like processes and always blocks. Structural HDLs, on the other hand, allow designers to describe the physical interconnections and arrangement of hardware components using modules or entities. Register-Transfer Level (RTL) HDLs, like both Verilog and VHDL, capture the flow of data between registers and are widely used for describing digital systems. Concurrency is a fundamental feature of HDLs, allowing designers to model operations happening simultaneously. This concurrency is expressed through constructs like processes or concurrent signal assignments, enabling a more natural representation of digital circuit behavior. HDLs support simulation, a crucial aspect of the design process. Simulation tools allow designers to verify the correctness and performance of their designs before moving to the physical implementation stage. Additionally, HDLs can be synthesized into netlists of gates, flip-flops, and other hardware elements. This synthesis process is vital for the implementation of designs on FPGAs or Application-Specific Integrated Circuits (ASICs).

Standard libraries in HDLs serve as a repository of predefined modules and functions, encompassing commonly used elements like logic gates, flip-flops, and arithmetic units. This not only expedites the design process but also promotes code reuse by providing a foundation of well-tested and established components. In the context of AMD Vivado, designers can further enhance their efficiency through the utilization of "Language Templates," a feature integrated into the tool. These templates offer predefined structures for common coding patterns, ensuring consistency and adherence to best practices. Moreover, Vivado provides a comprehensive suite of tools, including simulators, synthesizers, and waveform viewers, supporting the entire design flow from conceptualization to implementation. This integration, coupled with language templates, streamlines the development process and contributes to the robustness of FPGA and ASIC designs. In essence, HDLs provide engineers with a powerful set of tools to navigate the intricate landscape of digital design. They enable the expression of design concepts, facilitate simulation and verification, and ultimately empower the creation of efficient and effective digital systems. The choice between Verilog and VHDL, along with the rich features and toolsets they provide, underscores the significance of HDLs in modern electronic design processes. VHDL will be discussed in more detail in Chapter Four.

Synthesis and Optimization

The first stage of synthesis converts the circuit description, which is usually in a hardware description language or schematic form, into a netlist of basic gates. Then, the logic synthesis

process converts this netlist of basic gates into a netlist of FPGA logic blocks such that the number of logic blocks needed is minimized and/or circuit speed is maximized. Logic synthesis is sufficiently complex that it is usually broken into two or more subproblems, in this handbook we will use three-stage synthesis flow as shown in Figure 22.

Technology-independent logic optimization removes redundant logic and simplifies logic wherever possible. The optimized netlist of basic gates is then mapped to look-up tables. Both of these problems have been extensively studied and good algorithms and tools capable of targeting the FPGAs we are interested in studying are publicly available, so this book does not study these phases of the synthesis process.

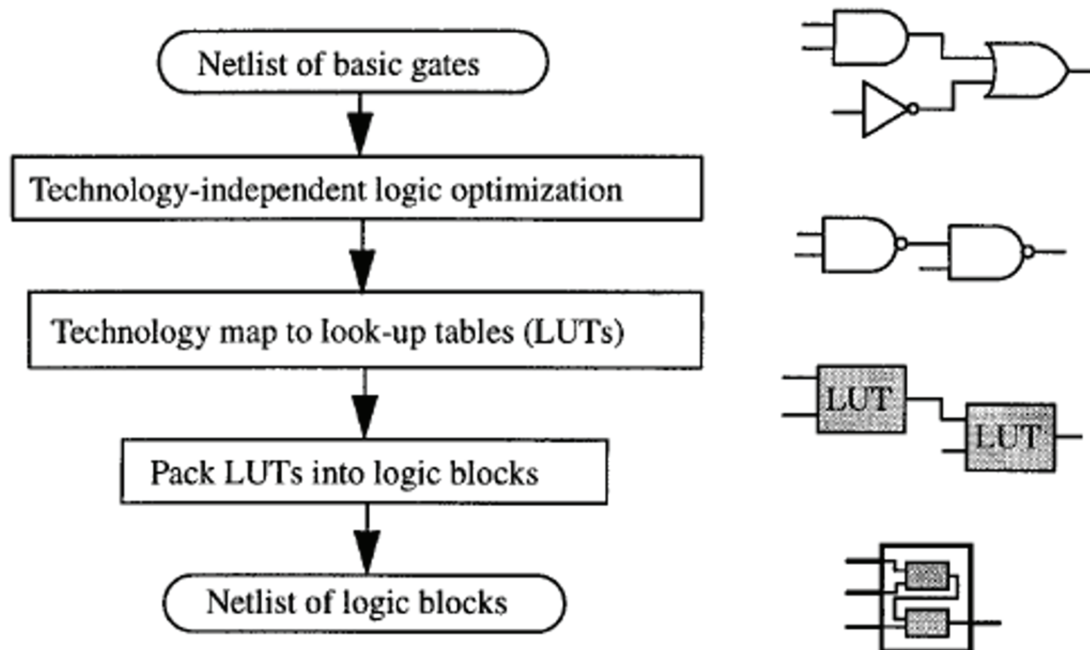


Figure 23: Details of Synthesis Procedure

The third synthesis step in Figure 22 is necessary whenever an FPGA logic block contains more than a single LUT. Logic block packing groups several LUTs and registers into one logic block, respecting limitations such as the number of LUTs a logic block may contain, and the number of distinct input signals and clocks a logic block may contain. The optimization goals in this phase are to pack connected LUTs together to minimize the number of signals to be routed between logic blocks, and to attempt to fill each logic block to its capacity to minimize the number of logic blocks used.

This problem is a form of clustering. Clustering and partitioning are essentially the same problem; divide a netlist into several pieces, such that certain constraints, such as maximum partition size, are respected, and some goal, such as minimizing the number of connections that cross partitions, is optimized. When a circuit is to be divided into only a few pieces, the problem is called partitioning. When a circuit is to be divided into many small pieces in one step (as opposed to recursively partitioning into a few partitions in each step), the problem is usually called clustering.

Place and Route: Place

A placement algorithm for FPGAs requires two main inputs: a netlist that outlines the functional blocks and their connections, and a device map specifying where each functional unit can be located. The goal is to choose a legal position for each block to optimize the circuit wiring. The constraints for legality and the criteria for optimization depend on the specific FPGA architecture in use. Figure 23 shows the FPGA placement problem with both the legality constraints and the optimization metric (what constitutes a "good" arrangement of functional blocks) depend on the FPGA architecture being targeted.

Achieving a good placement is crucial for FPGA designs, as a poor placement can hinder successful routing and result in lower operating speeds and increased power consumption. Finding an optimal placement is challenging, especially for large commercial FPGAs with around 500,000 functional blocks, leading to an enormous number of possible placements. Due to the computational complexity of the problem, exhaustive evaluation of all placement options is impractical. Consequently, the development of fast and effective heuristic placement algorithms is a significant area of research.

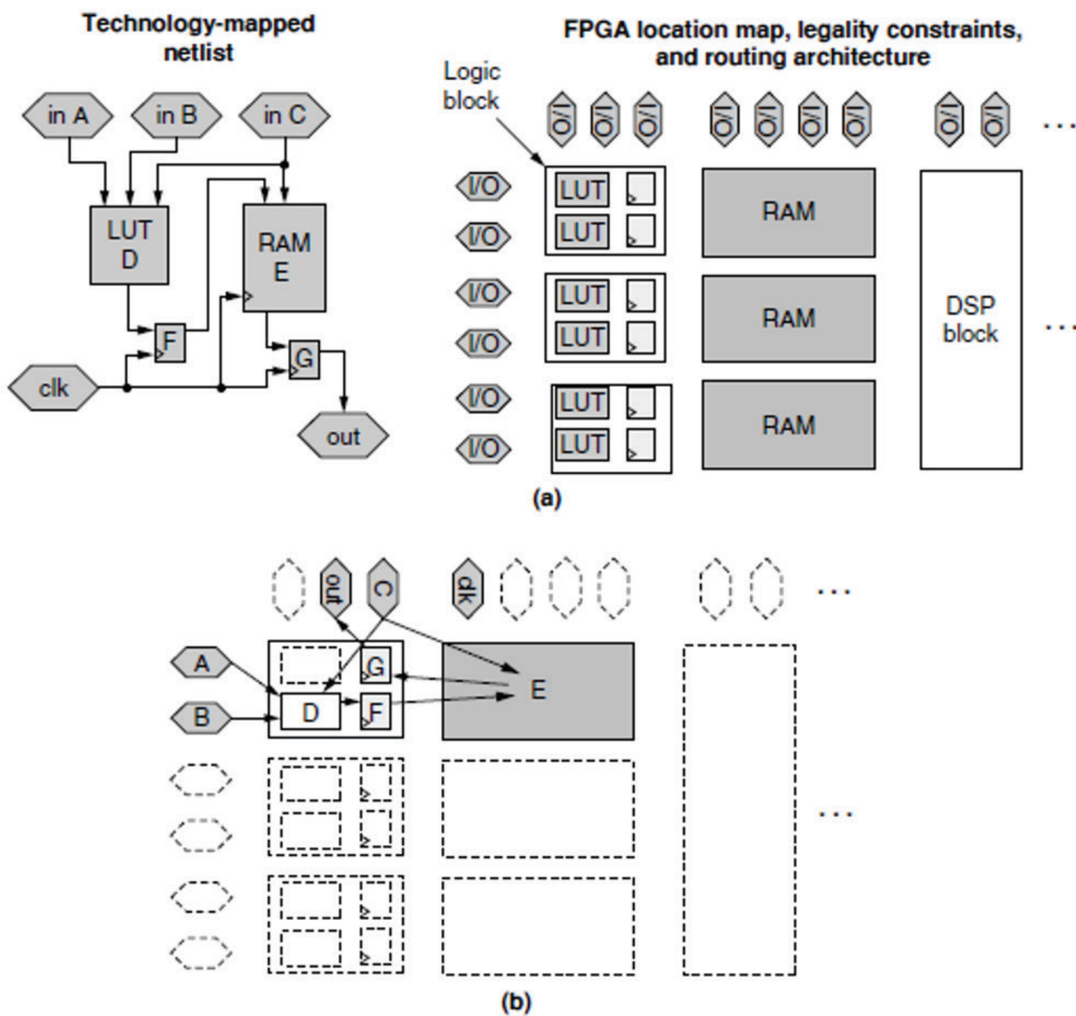


Figure 24: Placement overview: (a) inputs to the placement algorithm, and (b) placement algorithm output—the location of each block.

Simulated Annealing

Simulated annealing is a commonly used placement algorithm for FPGAs. It works like the process used to make strong metal alloys. In the beginning, blocks can move quite freely, but as the "temperature" drops (similar to cooling metal), they settle into a high-quality arrangement.

Initially, a placement is created, usually of low quality, by assigning each block to the first legal location found. Then, it gets better over time by suggesting and checking changes, or "moves." A move involves relocating a few blocks to new places, and a cost function is used to see how each move affects the overall arrangement. Moves that make the arrangement better are always accepted.

VPR Placement Tool

VPR is a popular tool for placing components on FPGAs using a technique called simulated annealing. People often use it with T-VPack, a clustering algorithm, or a similar one, that groups logical elements into allowed blocks. VPR is handy because it can adjust itself to different FPGA setups, as long as they use a specific kind of routing.

Unlike some other methods, VPR doesn't stick to fixed temperatures and cooling rates. Instead, it figures out its annealing schedule based on the situation during placement. This flexible approach helps it produce high-quality results for various design sizes, FPGA setups, and cost considerations. That's why it's often preferred over methods with more rigid, predetermined schedules.

Timing-Driven FPGA Placement

Instead of using quick estimates to figure out how well the placement of components on an FPGA will work, some methods use a router to plan out the connections for each suggested placement during the simulated annealing process. These methods can get detailed information about the wiring, congestion, and timing directly from the circuit's routing.

One advantage of this approach is that it allows for the development of a placement algorithm that can automatically adjust to a broader range of FPGA setups. This is because the algorithm doesn't rely on too many assumptions about the device-routing architecture in its cost calculations. However, a downside is that using a router in the cost calculation takes a lot of computer processing time. Checking the cost after each move is very demanding, making it hard to evaluate enough moves quickly for large circuits.

PROXI is an example of a timing-driven FPGA placement algorithm that uses a router for its cost calculations. The cost in PROXI is a weighted sum of the number of nets that haven't been connected and the delay of the most critical path in the circuit. After each placement change, PROXI disconnects all the nets connected to blocks that have moved and plans new routes for them using a fast, directed-search maze router.

Partition-Based Placement

Another way to place components on an FPGA involves splitting the circuit into smaller sections and assigning each section to a specific part of the FPGA. Usually, each splitting step divides a previous larger section into two parts, or it splits the component into two pieces. Some methods do more complex splitting to create multiple smaller sections in one step. The goal of these methods is to minimize the number of connections that go between the sections or cross them.

Since each section of the circuit ends up in a different area of the FPGA, this approach reduces the number of connections going out of each area. This indirectly helps optimize the amount of wiring needed for the design. This method can handle large problems because there are good, efficient algorithms for splitting the circuit into sections. However, for certain FPGA setups, this method has a downside. It doesn't directly optimize the timing of the circuit or the amount of routing needed for the placement. Hierarchical FPGAs are good candidates for partition-based placement, since their routing architectures create natural partitioning cut lines.

Place and Route: Route

Routing plays a vital role in transferring circuits onto FPGAs. When dealing with extensive circuits that use numerous FPGA resources, it becomes challenging and time-consuming to effectively connect all the signals. Furthermore, the effectiveness of the mapped circuit relies on routing critical and nearly critical paths with the least possible interconnect delays. One drawback of FPGAs is their slower speed compared to ASIC counterparts, emphasizing the need to minimize every potential nanosecond of delay in the routing process.

Once the positions for all the logic blocks in a circuit are chosen, a route is planned to decide which switches in the FPGA should be activated to connect the input and output pins of the logic blocks needed by the circuit. In FPGA routing, the usual way is to picture the routing structure of the FPGA as a directed graph. Each wire and each pin on a logic block becomes a point in this graph, and possible connections become the lines between them. While some past research has treated FPGAs as undirected graphs, a directed graph is necessary when modeling directional switches like tri-state buffers and multiplexers accurately.

Routing a connection means finding a path in this graph between the points representing the pins of the logic blocks that need to be connected. To use as few of the limited number of wires in an FPGA as possible, the goal is to keep this path short. It's also crucial that the routing for one connection doesn't use up the routing resources needed by another connection. That's why most FPGA routers have some method to avoid congestion and resolve conflicts over routing resources. Another goal is to make connections on or near the critical path speedy by using short paths and fast routing resources. Routers aiming to optimize timing this way are called timing-driven, while delay-oblivious routers focus purely on routability. Since most of the delay in FPGAs comes from the programmable routing, timing-driven routing is important for achieving good circuit speeds.

FPGA routes can be split into two types. Combined global-detailed routers determine a complete

routing path in one go, while two-step routing algorithms first do global routing to decide which logic block pins and channel segments each connection will use within each of the specified channel segments. A channel segment is the length of a routing channel that spans one logic block – a channel spanning M logic blocks contains M channel segments. The task of an FPGA detailed router is often tricky because FPGA routing has limited flexibility, and the detailed router is highly constrained by the decisions the global router made about which channel segments each connection must use. Combined global-detailed routers have the potential to more fully optimize the routing since they are not bound by such constraints.

Bitstream Generation

A reconfigurable logic device is a bit like a mix between a fixed hardware device and a programmable instruction set processor. What sets them apart is how they're set up and programmed. Both use "software" for programming, but they handle it in different ways.
In an instruction set processor, the programming is a set of binary codes fed into the device while it's running. These codes make the processor change its internal logic and routing on every cycle based on the input of these binary codes.

On the flip side, a reconfigurable logic device, like an FPGA, is built differently. It has a two-dimensional array of programmable logic elements connected by a programmable interconnection network. The significant difference is that an FPGA is usually programmed as a complete unit, with all its internal components working together at the same time. Unlike an instruction set processor, the programming data for an FPGA is loaded into the device's internal units before it starts operating, and typically, no changes are made to the data while the device is running.

The data used to program a reconfigurable logic device is commonly called a "bitstream," although this term is somewhat misleading. Unlike an instruction set processor where the configuration data are continuously streamed into the internal units, an FPGA usually loads its data only once during setup. The format of the bitstream is often kept as a trade secret by manufacturers, making it less accessible for experimentation with new tools and techniques by third parties. While most users of commercial reconfigurable logic devices are okay with the vendor-supplied tools, those interested in the internal structure find trade secrecy to be an important issue.

The bitstream is like a map that shows how different small hardware parts come together in a reconfigurable logic device to create a working digital circuit. While there's no strict limit to the types of units in a reconfigurable logic device, two basic structures are common in most modern FPGAs: the lookup table (LUT) and the switch box.

Similar to switch boxes, the configuration bitstream data for Input/Output Blocks (IOBs) consists of bits that set flip-flops within them to choose specific features. In newer generations of FPGA devices, there are also special-purpose units like block memory and multiplier units. The actual data bits may be part of the bitstream, initializing the BlockRAM during power-up. However, to keep the bitstream size smaller, this data might be absent, and the internal circuitry could be needed to reset

and initialize the BlockRAM.

Apart from IOBs, other internal data, like BlockRAM, is connected to the switch boxes in different ways, and deciding its location and interfacing in the interconnection network is a significant architectural choice in modern reconfigurable logic device design. Many other features in the FPGA, such as global control related to configuration and reconfiguration, ID codes, and error-checking information, have control bits in the bitstream. Implementation of these features can vary widely among different device families.

Basic control for bit-level storage elements, like flip-flops on the LUT output, is a common feature. Control bits often set circuit parameters, such as the type of flip-flop (D, JK, T) or the clock edge trigger type (rising or falling edge). Being able to change the flip-flop into a transparent D-type latch is also a popular option. Each of these bits contributes to the configuration data, with one set of flip-flop configuration settings per LUT being typical.

Programming and Configuration

The FPGA configuration bitstream, which is like the instructions telling the FPGA how to work, is usually saved outside in a nonvolatile memory, like an EPROM. The data are typically loaded into the device soon after it's turned on, usually bit by bit. This loading method might be why many engineers see the configuration data as a "stream of bits." The reason for loading bit by bit is mostly about keeping costs low and being convenient. Since there's usually no rush to load the FPGA configuration data when it's powered up, using just one pin for this data is the easiest and cheapest way. After the data are fully loaded, this pin might even be used for regular input/output (I/O) tasks, so the configuration downloading doesn't take up valuable I/O resources on the device.

Most FPGAs use a serial way of loading the configuration, but some have a parallel option that uses eight I/O pins to load the data all at once. This can be useful for designs using an 8-bit memory device or for applications where the FPGA needs to be reprogrammed often and speed is crucial—like when it's controlled by another processor. Just like the serial approach, the pins can go back to regular I/O tasks once the downloading is done. Quad SPI flash devices for boot are common across most Digilent boards. The programmer essentially loads the bitstream into a flash memory which the FPGA boots from.

In the factory, during the testing of FPGA devices after they are made, having a high-speed configuration can be extremely helpful. Testing FPGAs can be expensive because of the time spent connected to test equipment. So, making the configuration download faster can mean the FPGA manufacturer needs fewer pieces of test equipment, saving a lot of money during manufacturing. The need for high-speed download is more about making the testing process more efficient than meeting any customer requirements for changing how the FPGA works while it's running.

There's also a kind of device that uses non-volatile memory, like Flash-style memory, instead of RAM and flip-flops for the internal logic and control. These devices, like those from companies such as Actel, only need to be programmed once and don't need to reload configuration data when

powered up. This can be essential in systems that need to power up quickly. They are also more resistant to soft errors, making them popular in tough environments like space and military applications.

Chapter Three Summary

The FPGA design flow is a structured process that translates a conceptual hardware description into a fully optimized and functional FPGA implementation. This journey unfolds through distinct stages, each crucial in shaping the digital design to fit the constraints and capabilities of the FPGA. From initial concept to synthesis, place and route, and finally generating the bitstream, every step demands careful consideration of architecture, timing, power efficiency, and resource utilization. Engineers must carefully balance performance, flexibility, and efficiency throughout this process to successfully realize their digital designs on programmable hardware.

Chapter Four: FPGA Tools and Development Environments

Within the domain of FPGA development, understanding the landscape of vendors, families, development environments, simulation tools, and programming languages is pivotal for proficient design and implementation. This section undertakes a detailed examination of these fundamental components, starting with an exploration of popular FPGA vendors and families. Focusing notably on AMD and pertinent open-source software that aids in the development for their components, this subsection offers insights into the prevailing industry standards and the tools available to FPGA developers. By delving into the specifics of vendor offerings and the associated development ecosystem, readers will garner a nuanced understanding of the foundational elements shaping contemporary FPGA design methodologies.

What to expect in Chapter Four:

- Popular FPGA Vendors
- IDEs
- Simulation Tools

Popular FPGA Vendors and Families

The FPGA market is replete with diverse offerings from various vendors, each possessing its unique strengths and characteristics. Notable among these are AMD, Intel, Microchip Technology (formerly Microsemi), and Lattice Semiconductor.

AMD: Formerly known as Xilinx, AMD continues to hold a prominent position in the FPGA market, offering a comprehensive portfolio of families catering to a wide range of applications. The Spartan series addresses entry-level and cost-sensitive applications, while the Artix series extends this versatility to mid-range applications with enhanced performance and logic capacity. The Kintex series delivers heightened performance and scalability, suitable for industrial and aerospace applications, while the Virtex series offers unparalleled performance and integration features, ideal for data center acceleration and high-performance computing. Digilent uses AMD FPGAs.

Intel: As a major competitor to AMD, Intel – formerly Altera – boasts a formidable line-up of FPGA families. The Stratix series targets high-performance computing and data center applications, while the Cyclone series caters to cost-sensitive and low-power applications, providing a comprehensive range of options for FPGA developers.

Microchip Technology (formerly Microsemi): Microchip Technology offers the SmartFusion series, blending FPGA fabric with integrated ARM Cortex processors for embedded applications. Additionally, their PolarFire series provides low-power and high-reliability FPGA solutions suitable for a wide range of industrial and aerospace applications.

Lattice Semiconductor: Lattice Semiconductor's ECP and MachXO families target low-power and compact form-factor applications, providing alternatives to traditional FPGA offerings with a focus on power efficiency and compactness.

Free and Open Source Software Tools for FPGA Work

In the world of FPGA development, using open-source (free to use and share) software is very important. These free tools, like simulators, synthesizers, linters, and programs that write code, help a lot in making FPGA technology easier to use for everyone. This includes students, engineers, and anyone interested in working with FPGAs. Let's talk about how these free projects help in FPGA development.

Projects like IceStorm and Project X-Ray are great examples of free tools that help people work with FPGAs made by companies like Lattice Semiconductor and AMD. Besides these, there are more tools that are very useful:

Yosys is a free tool that helps turn the design code (Verilog) into something the FPGA can understand. It's very important for making FPGA designs work.

SymbiFlow wants to be like the GCC (a very popular free software compiler) but for FPGAs. This means it wants to help people make FPGA designs in a standard way, no matter what type of FPGA they are using.

NextPNR is a tool that takes the design and fits it onto the actual FPGA chip. It's part of the bigger SymbiFlow project and is important for making sure the design can work in real life.
Tools like GHDL and Verilator let people test their FPGA designs on a computer before they try them on a real chip. This is very helpful for finding and fixing mistakes.

Working with Company-Made Platforms

Even though they're not technically free (typically requiring paid licenses), platforms like Vitis and Vivado from AMD, and ROCm, are starting to work well with free tools. This means people can use the best parts of both free and company-made tools together, making FPGA development better.

Having these tools makes it easier for more people to work with FPGAs, meaning more projects and ideas can come to life. It also helps students learn better and lets everyone share their knowledge and help one another. The world of FPGA development is getting a big boost from these free software projects.

Integrated Development Environments (IDEs)

IDEs serve as indispensable tools, providing a unified platform for design, synthesis, and verification. The IDE's role is to implement the functionality required to develop, simulate, and program designs for FPGAs by essentially taking care of all the steps mentioned in Chapter 3. Among the leading IDEs tailored for FPGAs, Vivado Design Suite stands out as a comprehensive and feature-rich environment designed to streamline the entire development workflow.

Vivado Design Suite

Introduced as a successor to the Xilinx ISE (Integrated Synthesis Environment), the Vivado Design Suite represents a significant leap forward in FPGA design capabilities. Designed with the latest FPGA architectures in mind, such as the UltraScale and UltraScale+ series, Vivado streamlines and optimizes the design process through advanced algorithms and a modern user interface.

Key Features:

High-Level Synthesis (HLS): Vivado allows designers to model FPGA circuits in higher-level programming languages such as C, C++, and SystemC, dramatically reducing the design complexity and time. Intellectual property (IP) cores developed in Vitis HLS can be used in Vivado designs.

IP Integrator: This feature enables the rapid composition of IP cores and custom modules into a single design canvas, facilitating a block design methodology that enhances productivity.

Logic Simulation: Integrated logic simulators provide the capability to test and verify design behavior before hardware implementation.

Implementation Tools: Vivado offers advanced place and route algorithms that optimize the physical layout on the FPGA fabric, ensuring the best possible performance and resource utilization.

Link: *Getting Started With Vivado*

Vitis Unified Software Platform

Vitis emerged as AMD's answer to the growing demand for a unified software platform that not only supports FPGA design but also bridges the gap between hardware acceleration and software development. Vitis integrates and replaces the SDK (Software Development Kit) development environment, offering a comprehensive ecosystem for developing applications across AI, software, and hardware domains.

Seamless Integration with Vivado: Vitis works hand-in-hand with Vivado, allowing designs and IP generated in Vivado to be easily imported and utilized within the Vitis environment for software acceleration.

High-Level Synthesis (HLS): Like Vivado, Vitis supports HLS, enabling software developers to write code in C/C++ that is compiled into FPGA logic. Vitis HLS can be used to develop and test IP cores that can be used in Vivado designs. Alternatively, Vitis can be used to integrate HLS kernels into complex software projects at compile time, including both Linux software components and HLS components intended to run in FPGA fabric. This tool leverages Vivado build functionality under the hood.

AI Engine: For AI and machine learning applications, Vitis provides specialized support for AI model development and deployment, leveraging the adaptable compute architecture of AMD FPGAs. Digilent boards generally don't support this, as AI engines are a feature of higher cost silicon.
Comprehensive Software Libraries: Vitis includes optimized libraries for a range of applications, including data analytics, image processing, and financial computations, allowing developers to leverage these pre-built modules for rapid application development. Note that some of these libraries may be difficult to set up on Digilent FPGA development boards, as not all of the required PetaLinux support is available.

Comprehensive Software Libraries: Vitis includes optimized libraries for a range of applications, including data analytics, image processing, and financial computations, allowing developers to leverage these pre-built modules for rapid application development. Note that some of these libraries may be difficult to set up on Digilent FPGA development boards, as not all of the required PetaLinux support is available.

Link: *Getting Started With Vitis*

Vitis is for writing software to run in an FPGA, and is the combination of a couple of different AMD tools, including what was AMD SDK, Vivado High-Level Synthesis (HLS), and SDSoC. The functionality of each of these is now merged together under Vitis. To break each of these down:

1. AMD SDK (Vitis): Write C/C++ to run on a processor in a design you created in Vivado. This code often ends up being at least partially used to configure and control elements of the hardware design – it's easier to rebuild, tweak, and debug than the hardware portion is.

2. Vivado HLS (Vitis HLS): Write C/C++ to be built into a block which you can include in a Vivado project. This block can often be reused in multiple projects, and even potentially be loaded up in Vivado for manual optimization.

3. SDSoC (Vitis): Write C/C++ to be built into a block which the tool stitches into a previously created Vivado design. You take a platform with some I/O built in, and start accelerating certain data processing functions of your software design by building them into the hardware (while still writing them in software languages).

Model Composer is a Xilinx tool that integrates with MATLAB. It's generally downloaded with the Vitis platform and streamlines the design and streamlines the design of digital signal processing

(DSP) systems for FPGAs. It enables designers to create high-level block diagrams that can be directly translated into synthesizable HDL code, bypassing the need for detailed knowledge of VHDL or Verilog. This accelerates the development process for FPGA-based projects, from communications to consumer electronics, by allowing rapid prototyping and testing within the familiar MATLAB environment.

The tool's key features include efficient HDL code generation, comprehensive simulation and verification capabilities, and support for both fixed-point and floating-point models. This makes it easier to balance performance, accuracy, and resource use. By simplifying the transition from conceptual models to hardware-ready designs, Model Composer makes FPGA technology accessible to a wider range of engineers, enhancing productivity and innovation in DSP applications.

Link: *What's different between Vivado and Vitis?*

Vivado – Vitis Tool Flow

Vivado and Vitis are not standalone tools - they can each be used in different stages of a larger workflow. In a typical embedded software flow, a hardware design targeting FPGA fabric is created in Vivado, which then creates handoff files that are used in constructing a hardware platform supporting that design for use in Vitis. At that point, Vitis is then used to develop software for any processors that exist in the platform. This section describes the process of creating that platform in further detail.

1. Design Entry

Project Creation: Designers initiate the process by creating a new project within Vivado, specifying project settings such as device family, device part or target board, and simulation language.

Design Sources: Designers add source files, including HDL (Hardware Description Language) files, constraints, and IP (Intellectual Property) cores to the project.

2. Synthesis

RTL Synthesis: Vivado synthesizes the RTL (Register Transfer Level) code to generate a logical netlist, optimizing the design for target performance and area constraints.

High-Level Synthesis (HLS): Optionally, designers can leverage HLS to synthesize C/C++ code into hardware-accelerated functions.

3. Implementation

Floorplanning: Designers allocate physical resources on the FPGA device through

floorplanning, optimizing placement and routing for critical design elements.

Place and Route: Vivado performs automated placement and routing, mapping the logical netlist onto the physical FPGA fabric while meeting timing and resource constraints.

4. Bitstream Generation

Bitstream Creation: Vivado generates the bitstream—a binary file containing configuration data—representing the finalized FPGA design.

Configuration File: The bitstream, along with other necessary configuration files, is prepared for deployment onto the target FPGA device.

5. Verification and Validation

Timing Analysis: Vivado conducts timing analysis to verify that design requirements, such as clock frequency and setup/hold times, are met.

Simulation: Designers can simulate the synthesized design using Vivado's built-in simulator to validate functionality and behaviour.

6. Hardware Definition

Vivado also exports a hardware component called the Hardware Definition File. The hardware definition file serves as a comprehensive container holding all the necessary information to construct a platform for any targeted AMD device. Within this container, a key component is the HWH, or Hardware Handoff File. This file emerges from the execution of output products on a Block Design, essentially a detailed map showing the interconnection and functionality of various components. The HWH file's knowledge is confined to the scope of the Block Design.

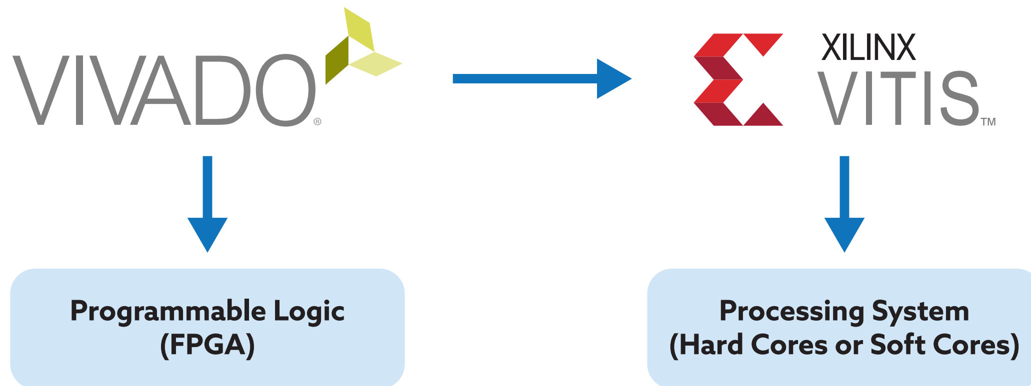
This HWH file plays a crucial role for software tools, as it encapsulates all the requisite details for tailoring an application specifically for the device in question. It delineates the architecture of the device, including the central processing units (CPUs), data pathways (buses), integral components (IP), and the interfaces for external communication (ports and pins) such as interrupt signals.

In scenarios involving a Zynq Ultrascale+ device (or a Zynq 7-series), the hardware definition file package is enriched with additional files: psu_init.c/h and tcl scripts. These files are instrumental in configuring the device's Processing Subsystem according to the parameters defined in Vivado, the design tool utilized for the project. The psu_init scripts are particularly pivotal during the First Stage Boot Loader (FSBL), ensuring the system boots up with the correct configurations. The tcl script, on the other hand, aids in debugging by facilitating the same configuration tasks.

Furthermore, if the project incorporates a block RAM (BRAM) system that is directly addressable in the Programmable Logic (PL), or in cases involving FPGA setups not integrated into a System on

Chip (SoC), an MMI file comes into play. In MicroBlaze-based systems, this file is essential for using the UpdateMEM tool to load the BRAM with the executable file (ELF) generated in Vitis, thereby completing the configuration for the device's application.

The hardware definition file is then loaded into Vitis and all the information necessary regarding the hardware design is loaded. Various header files and functions are then ready to use from the software part of the project.



While both Vivado and Vitis offer high-level synthesis capabilities, their primary distinction lies in their target audience and application focus. Vivado is tailored for hardware engineers focusing on FPGA circuit design, offering detailed control over hardware aspects. In contrast, Vitis targets software developers and data scientists looking to leverage FPGA acceleration without delving into the complexities of hardware design. It's also important to note that Vitis HLS will come up in the embedded software flow before Vivado block design applies.

Simulation Tools

Simulation constitutes a vital aspect of FPGA development, enabling designers to verify the functionality and performance of their designs before deployment. AMD offers a suite of simulation resources integrated within the Vivado Design Suite, providing comprehensive support for behavioral and timing analysis of FPGA designs.

AMD's simulation resources encompass a range of tools and utilities tailored to meet the diverse simulation needs of FPGA developers. The Vivado Simulator, also known as Xsim, is a built-in feature of the Vivado Design Suite, offers advanced capabilities for RTL simulation, enabling designers to validate their designs at the register-transfer level. With support for industry-standard languages such as VHDL and Verilog, as well as advanced verification methodologies such as SystemVerilog Assertions (SVA), the Vivado Simulator facilitates thorough and efficient verification of FPGA designs.

Additionally, AMD provides support for third-party simulation tools such as ModelSim from Mentor, a widely-used simulator in the FPGA industry. ModelSim offers advanced debugging features,

waveform viewing capabilities, and support for a wide range of simulation languages and methodologies, making it a preferred choice for FPGA developers seeking comprehensive simulation solutions.

In addition to proprietary simulation tools, open-source VHDL simulation frameworks such as GHDL and Icarus Verilog offer viable alternatives for FPGA developers. GHDL, a free and open-source VHDL simulator, provides support for IEEE standard VHDL language constructs and offers seamless integration with popular development environments such as Visual Studio Code (VS Code) through the use of appropriate extensions. Similarly, Icarus Verilog offers a robust simulation environment for Verilog designs, with support for mixed-language simulation and advanced debugging features.

By leveraging AMD's simulation resources and exploring alternative simulation tools such as ModelSim and open-source VHDL simulators, FPGA developers can ensure the thorough validation of their designs and mitigate potential errors and performance bottlenecks early in the development process. Through rigorous simulation and verification, designers can enhance the reliability and robustness of their FPGA-based solutions, ultimately delivering superior performance and functionality to end-users.

Chapter Four Summary

In the realm of FPGA development, familiarity with various vendors, FPGA families, development environments, simulation tools, and programming languages is crucial for effective design and implementation. By detailing vendor offerings and the associated development ecosystems, you should now have a comprehensive understanding of the foundational elements influencing modern FPGA design methodologies.

Chapter Five: FPGA Programming Languages

Central to the harnessing of FPGA capabilities lies the mastery of Hardware Description Languages (HDLs), with VHDL emerging as a preeminent choice among engineers and designers. This chapter aims to serve as a comprehensive guide to FPGA programming, with a particular emphasis on VHDL. It will navigate through the fundamental concepts, delve into advanced techniques, and explore practical applications, ensuring a robust understanding of FPGA development.

What to expect in Chapter Five:

- VHDL
- Sequential Statements
- Concurrent Statements
- Synchronous and Asynchronous Logic
- Hierarchical Design and Module Instantiation
- VHDL Data Types and Conversions
- Advanced VHDL Techniques

Introduction to VHDL

VHDL, an acronym for Very-High-Speed-Integrated-Circuit Hardware Description Language, is a robust and versatile language instrumental in specifying the behaviour and structure of digital systems. Originating from the U.S. Department of Defense's Very High-Speed Integrated Circuit (VHSIC) program, VHDL has ascended to become an industry-standard language for FPGA and ASIC design. This section initiates the journey into VHDL with a comprehensive exploration of its basic syntax, data types, control structures, and modelling principles. With VHDL it is always important to remember that one is describing hardware, and therefore all the issues one may expect of having with digital hardware circuits, one will also find when implementing VHDL in FPGAs.

Basic Syntax

VHDL employs a structured syntax resembling natural language constructs, facilitating the expression of complex digital designs in a concise and comprehensible manner. The basic syntax of VHDL comprises entity-architecture pairs, where entities define the interface of a hardware component, while architectures describe its internal behaviour. The following VHDL snippet shows how Module Instantiation is done, which is discussed further in following sections.

```vhdl
-- Example of a simple VHDL entity-architecture pair
entity AND_GATE is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Y : out STD_LOGIC);
end AND_GATE;

architecture Behavioral of AND_GATE is
begin
    Y <= A and B; -- VHDL code for AND gate functionality
end Behavioral;
```

This VHDL code defines an AND gate, a fundamental digital logic component that performs a logical AND operation on its input signals. Let's break down the code for beginners:

Entity Declaration (AND_GATE): In VHDL, an entity represents a hardware component's interface. It defines the inputs and outputs of the component. AND_GATE is the name given to this entity.

Port Declaration: Inside the entity, the Port keyword is used to declare the input and output ports of the AND gate. A and B are declared as input ports (in), while Y is declared as an output port (out). STD_LOGIC represents a single-bit signal, which can take on values '0', '1', 'U' (undefined), 'X' (unknown), 'Z' (high impedance), or 'W' (weak unknown).

Architecture Declaration (Behavioural): The architecture keyword defines the internal behaviour or functionality of the entity. Behavioural is the name given to this architecture.

Behavioural Code: Inside the architecture, Y <= A and B; is the behavioural code that describes the AND gate's functionality. The symbol <= is the signal assignment operator, indicating that the output signal Y is assigned the result of the AND operation between input signals A and B. The and keyword represents the logical AND operation, which yields '1' (true) only when both inputs are '1'. Otherwise, it results in '0' (false). This way a designer can implement digital logic circuitry inside a black called the entity. In subsequent sections the use of multiple entities together is discussed.

Data Types

VHDL offers a rich set of data types catering to various levels of abstraction in digital design. These data types encompass scalar types, composite types, and enumerated types, each serving distinct purposes in modelling digital systems. Data types and converting between them can be a common stumbling-block for those first learning VHDL, as it is a strongly-typed language and requires explicit conversion between types. For example, the addition operator found in the IEEE numeric standard package, which is extremely commonly used, takes two signed or unsigned inputs and outputs the corresponding signed or unsigned type. To mix and match data types, or use the more-portable standard_logic_vectors commonly used for I/O interfaces, a designer must use type-conversion functions.

Scalar Types: Scalar types represent single values and include BOOLEAN, INTEGER, and REAL. These types are essential for expressing individual signals and variables within a design.

```vhdl
-- Boolean Type Example
signal flag : BOOLEAN := true;

-- Integer Type Example
signal count : INTEGER := 0;

-- Real Type Example
signal voltage : REAL := 3.3;
```

Composite Types: Composite types combine multiple values into structured data objects, enabling the representation of complex data structures. ARRAY and RECORD are common examples of composite types in VHDL.

```vhdl
-- 1D Array Example
type Word_Array is array(0 to 7) of STD_LOGIC_VECTOR(7 downto 0);
signal word_data : Word_Array;

-- 2D Array Example
type Matrix_Array is array(0 to 3, 0 to 3) of INTEGER;
signal matrix_data : Matrix_Array;


-- Array Type Example
type Byte_Array is array(7 downto 0) of STD_LOGIC;
signal data_byte : Byte_Array;

-- Record Type Example
type Person_Record is record
    Name : STRING(1 to 50);
    Age : INTEGER;
    Height : REAL;
end record;

signal person_info : Person_Record;
```

Enumerated Types: Enumerated types facilitate the definition of discrete sets of values, enhancing code readability and maintainability. Enumerated types are particularly useful for modelling state machines and finite state systems.

```vhdl
-- Enum Type Example
type State_Type is (Idle, Running, Stopped);
signal current_state : State_Type := Idle;
```

Control Structures

VHDL encompasses a variety of control structures for specifying the flow of execution within a design. These control structures include sequential statements and concurrent statements, each serving distinct purposes in describing the behaviour of digital systems.

Sequential Statements: Sequential statements define actions that occur in sequence within a process or block. Examples of sequential statements include SEQUENCE, WAIT, and ASSERT, which enable the modelling of sequential logic and state transitions.

```
architecture Behavioral of Example is
begin
    process (clk)
    begin
        if rising_edge(clk) then
            -- Sequential assignment
            output <= input1 and input2;
            -- Another sequential assignment
            counter <= counter + 1;
        end if;
    end process;
end Behavioral;
```

In this example, the statements inside the process block are executed sequentially. First, the output signal is assigned the result of the AND operation between input1 and input2. Then, the counter is incremented by 1.

Concurrent Statements: Concurrent statements describe operations that occur concurrently or simultaneously within a design. Processes, generate statements, and component instantiations are common examples of concurrent statements in VHDL, facilitating the parallel execution of logic and data processing tasks.

```
architecture Behavioral of Example is
begin
    -- Concurrent assignment
    output <= input1 and input2;
    -- Another concurrent assignment
    counter <= counter + 1 when rising_edge(clk);
end Behavioral;
```

In this example, the assignments to output and counter occur concurrently. The output signal is continuously updated based on the AND operation between input1 and input2, while the counter is incremented only on the rising edge of the clock (clk). Both assignments can happen simultaneously, reflecting the concurrent nature of VHDL.

The ***fundamental distinction*** between sequential and concurrent statements in VHDL lies in their execution order and timing behaviour. Sequential statements follow a predefined order of execution, where each statement is processed in sequence as they appear within a process or block. In contrast, concurrent statements are executed simultaneously, without any predetermined order. This concurrent execution enables multiple actions to occur concurrently, allowing for parallel behaviour within the design. This is the real power of the FPGA!! The following sections will explore this argument with further detail. More particularly section VHDL Data Types and Conversions.

The Process

At the heart of VHDL are processes, which allow designers to model the behaviour of digital circuits. Processes in VHDL are primarily used to describe the behaviour of sequential logic.

Sensitivity List: A sensitivity list in VHDL specifies the signals that the process is sensitive to. It essentially tells the simulator or hardware which events should trigger the execution of the process. The process will execute whenever any of the signals in the sensitivity list experiences a change in value. For example:

```
process (signal1, signal2)
begin
  -- Process body
end process;
```

In this example, the process will be triggered whenever either signal1 or signal2 changes.

Signal Assignment: In VHDL, signals represent physical connections between different parts of a digital circuit. Signal assignment within a process involves updating the value of a signal based on certain conditions or expressions.

```
process (clk)
begin
  if rising_edge(clk) then
    signal_out <= signal_in1 and signal_in2;
  end if;
end process;
```

In this example, the signal_out is updated whenever there is a rising edge on the clock signal (clk). The value assigned to signal_out is the logical AND of signal_in1 and signal_in2. A very important thing to keep in mind is that signals do NOT immediately adopt the value assigned to them, they are updated when the process finishes.

Variable Assignment: Variables in VHDL are used for temporary storage within a process and are only accessible within the process in which they are declared. Unlike signals, variables are not bound by the event-driven model of signal changes. Variable assignments occur sequentially within the process, meaning they execute one after the other and they are updated instantly.

```
process (reset)
  variable count : integer := 0;
begin
  if reset = '1' then
    count := 0;
  elsif rising_edge(clk) then
    count := count + 1;
  end if;
end process;
```

The key concept to grasp (and a common source of confusion) is that variables instantly adopt the value assigned to them, whereas signals' behaviour varies depending on whether they are utilized in combinational or sequential code (such as in a process). In combinational code, signals promptly assume the value assigned to them. Conversely, in sequential code (such as in a process), signals are employed in constructing flip-flops, which inherently do not promptly adopt the value of their assignment; they require one clock cycle. In general, I advise beginners to steer clear of variables. They tend to generate confusion and can be challenging to synthesize with the tools.

In the next chapter sequential statements are discussed. These are the tools available to the designer in a process.

Sequential Statements

Variables

Variables serve as containers for storing intermediate values between sequential VHDL statements. They are restricted to processes, procedures, and functions, and remain local to these constructs. The assignment operator ":=" is utilized when assigning a value to a variable.

```
signal Grant, Select: std_logic;
process(Rst, Clk)
    variable Q1, Q2, Q3: std_logic;
begin
    if Rst = '1' then
        Q1 := '0';
        Q2 := '0';
        Q3 := '0';
    elsif (Clk = '1' and Clk'event) then
        Q1 := Grant;
        Q2 := Select;
        Q3 := Q1 or Q2;
    end if;
end process;
```

Note: Both signals and variables transport data within a design. However, signals are required for conveying information between concurrent elements of the design. The following are examples of the most common sequential statements available to the designers.

If-then-else Statement

The following is a high-level example of the If-then-else Statement.

```
if Boolean_expr_1 then
    sequential_statements;
elsif Boolean_expr_2 then
    sequential_statements;
elsif Boolean_expr_3 then
    ...
else
    sequential statements;
end if;
```

The following includes Boolean expressions commonly implemented in the If-then-else Statement.

```
process ( a, b, m, n)
begin
    if m = n then
        r <= a + b;
    elsif m > 0 then
        r <= a - b;
    else
        r <= a + 1;
    end if;
end process;
```

Case Statement

The following is a high-level example of the Case Statement.

```
case sel is
    when choice_1 =>
        sequential_statements;
    when choice_2 =>
        sequential_statements;
    ...
    when others =>
        sequential_statements;
end case;
```

The following includes sequential statements commonly implemented in the Case Statement.

```
case sel is
    when "00" =>
        r <= a + b;
    when "10" =>
        r <= a - b;
    when others =>
        r <= a + 1;
end case;
```

For Loop

The following is a high-level example of the for loop.

```
for index in loop_range loop
    sequential statements;
end loop;
```

The following includes sequential statements commonly implemented in the for loop.

```
constant MAX: integer := 8;
signal a, b, y: std_logic_vector(MAX-1 downto 0);
...
for i in (MAX-1) downto 0 loop
    y(i) <= a(i) xor b(i);
end loop;
```

While Loop

The following is a high-level example of the while loop.

```
loop_name: while (condition) loop
    -- repeated statements
end loop loop_name;
```

The following includes sequential statements commonly implemented in the while loop.

```
while error_flag /= '1' and done /= '1' loop
    Clock <= not Clock;
    wait for CLK_PERIOD/2;
end loop;
```

Concurrent Statements

Any statement placed in architecture body is concurrent. Only one type of conditional statements is allowed as concurrent which are shown here. Remember these statements happen at the same time!

Conditional Signal Assignment

Syntax:

```
signal_name <= value_expr_1 when Boolean_expr_1 else
               value_expr_2 when Boolean_expr_2 else
               value_expr_3 when Boolean_expr_3 else
               ...
               value_expr_n;
```

Example: 4-to-1 Multiplexer (Mux):

```
z <= a when (s="00") else
     b when (s="01") else
     c when (s="10") else
     d when (s="11") else
     'X';
```

A more concise approach:

```
z <= a when (s="00") else
     b when (s="01") else
     c when (s="10") else
     d;
```

Selected Signal Assignment

Syntax:

```
with select_expression select
    signal_name <= value_expr_1 when choice_1,
                   value_expr_2 when choice_2,
                   ...
                   value_expr_n when choice_n;
```

Example: 4-to-1 Multiplexer (Mux):

```
with s select
    z <= a when "00",
         b when "01",
         c when "10",
         d when others;
```

Synchronous and Asynchronous Logic

Understanding the principles of synchronous and asynchronous logic is paramount for effective FPGA design. This chapter delves into the intricacies of these two fundamental types of logic, discussing their principles, applications, and considerations within the FPGA development environment.

Synchronous Logic

Synchronous logic relies on clock signals to synchronize operations, ensuring predictable and deterministic behaviour. This section explores the foundational concepts of synchronous logic, including the role of clock signals, clock domain crossing, and the impact of clock skew and jitter on system performance. Additionally, it discusses techniques for designing synchronous circuits and mitigating potential timing hazards.

```
-- Example of a synchronous D flip-flop in VHDL
entity D_FLIP_FLOP is
    Port ( D : in STD_LOGIC;
           CLK : in STD_LOGIC;
           Q : out STD_LOGIC);
end D_FLIP_FLOP;

architecture Behavioral of D_FLIP_FLOP is
begin
    process (CLK)
    begin
        if rising_edge(CLK) then
            Q <= D;
        end if;
    end process;
end Behavioral;
```

Clocking Considerations

In synchronous logic design, employing good patterns for using clocks is crucial for ensuring reliable and efficient operation of digital circuits. One fundamental practice is maintaining a single clock domain throughout the design. By using a single primary clock signal to synchronize all sequential elements within the FPGA, timing analysis is simplified, and the risk of timing violations is reduced.

For instance, consider the following VHDL snippet, where all sequential elements are clocked by the same primary clock signal:

```
entity MyDesign is
    Port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        data_in : in STD_LOGIC_VECTOR(7 downto 0);
        data_out : out STD_LOGIC_VECTOR(7 downto 0)
    );
end MyDesign;

architecture Behavioral of MyDesign is
begin
    process (clk, reset)
    begin
        if reset = '1' then
            -- Reset logic here
        elsif rising_edge(clk) then
            -- Sequential logic here
        end if;
    end process;
end Behavioral;
```

Additionally, when interfacing between different clock domains, proper clock domain crossing techniques should be employed to synchronize signals and avoid metastability issues. For instance, the following VHDL code demonstrates a double synchronization technique using two flip-flops to transfer data safely between two clock domains:

```
process (clk1, clk2)
begin
    if rising_edge(clk1) then
        data_sync1 <= data_in;
    end if;
end process;

process (clk2)
begin
    if rising_edge(clk2) then
        data_sync2 <= data_sync1;
        data_out <= data_sync2;
    end if;
end process;
```

*For more information on properly handling clock domain crossing and commonly used design techniques, Clifford E Cummings' paper for Sunburst Design Inc, **here**.*

Moreover, optimizing the clock tree is essential to minimize clock skew and jitter, ensuring consistent clock signals across the FPGA. Proper placement and routing of clock signals, along with clock skew analysis, contribute to improved timing closure and overall performance.

Utilizing clock gating techniques, such as the following VHDL example, can dynamically enable or disable clock signals based on specific conditions, reducing power consumption and improving energy efficiency in FPGA designs:

```
process (clk, enable)
begin
    if enable = '1' then
        clk_gated <= clk;
    else
        clk_gated <= '0';
    end if;
end process;
```

Lastly, incorporating clock enable signals allows for finer control over data capture by enabling or disabling flip-flops or registers based on specific conditions. This helps reduce unnecessary power consumption when not actively processing data. By following these good patterns for using clocks, designers can optimize the utilization of clock signals in synchronous logic designs, ensuring robustness, reliability, and performance in FPGA-based systems.

Resetting

Proper handling of reset signals is paramount in synchronous logic design to ensure reliable initialization and operation of digital circuits. This subsection explores the concepts of synchronous and asynchronous resets, highlighting their distinctions and offering examples of their implementation in VHDL.

Synchronous Reset

Synchronous resets are synchronized to the clock signal, guaranteeing that the reset operation occurs at a known point relative to the clock edge. This synchronization mitigates potential timing hazards and ensures consistent behaviour across different clock domains. In VHDL, synchronous resets are typically realized using a flip-flop with a reset-enable input. Consider the following VHDL snippet illustrating the implementation of a synchronous reset:

```
process (clk)
begin
    if rising_edge(clk) then
         if reset = '1' then
        -- Synchronous reset: reset flip-flop to '0' on active edge of 
clock
           flip_flop <= '0';
         else
        -- Update flip-flop state on rising edge of clock
           flip_flop <= data_in;
         end if;
    end if;
end process;
```

Asynchronous Reset:

In contrast, asynchronous resets are not synchronized to the clock signal and can occur independently of the clock edge. While providing immediate circuit initialization, asynchronous resets pose timing challenges, such as metastability. In VHDL, asynchronous resets are implemented using dedicated reset signals. Here's a VHDL example demonstrating the application of an asynchronous reset:

```
process (reset, clk)
begin
    if reset = '1' then
        -- Asynchronous reset: reset flip-flop to '0' immediately
        flip_flop <= '0';
    elsif rising_edge(clk) then
        -- Update flip-flop state on rising edge of clock
        flip_flop <= data_in;
    end if;
end process;
```

Notice how the reset signal here must be in the sensitivity list of the process, as such the process will be triggered both with a logical change of the clock and the reset. This way the reset signal is completely independent of the clock and thus is asynchronous. Understanding the principles of synchronous and asynchronous resets and their VHDL implementations empowers designers to effectively manage reset signals, ensuring the robustness and reliability of their digital designs.

*Digilent provides some common modules for handling reset synchronization and some basic CDC (Clock Domain Crossing) techniques, in the vivado-library repository on GitHub, **here**.*

Asynchronous Logic

Asynchronous logic, on the other hand, operates independently of clock signals, introducing potential timing hazards and metastability issues. Asynchronous logic in VHDL refers to digital circuitry that operates independently of a clock signal, unlike synchronous logic which relies on clock signals for synchronization. In asynchronous logic design, circuit elements respond immediately to changes in their inputs, without waiting for a clock signal to trigger their actions. This approach offers advantages in certain scenarios where strict timing requirements are not critical or where responsiveness to input changes is paramount.

In VHDL, asynchronous logic can be implemented using processes sensitive to input signals, allowing for immediate response to changes in input values. For example, an asynchronous D flip-flop can be designed to update its output whenever the input signal changes, rather than waiting for a clock signal.

While asynchronous logic offers flexibility and responsiveness, it also introduces challenges such as metastability, where flip-flops may capture uncertain values due to input changes occurring near

clock edges. To mitigate metastability and ensure reliable operation, designers employ synchronization techniques such as multiple flip-flops in series or handshaking protocols.

Overall, asynchronous logic in VHDL provides a versatile approach to digital circuit design, offering responsiveness and simplicity in certain applications where strict timing synchronization is not required. However, careful consideration of timing hazards and appropriate design techniques is necessary to ensure the robustness and reliability of asynchronous circuits.

Hierarchical Design and Module Instantiation

Hierarchical design involves breaking down a complex digital system into smaller, manageable modules or components. Each module encapsulates specific functionality, allowing for easier development, debugging, and maintenance. Hierarchical design promotes reusability and modularity, enabling designers to efficiently build and manage large-scale digital systems.

Module Parameterization / Generics

Module parameterization, also known as generics in VHDL, allows modules to be instantiated with configurable parameters. This feature enhances the flexibility and versatility of modules, enabling them to adapt to different requirements without modification. Generics enable designers to create reusable components that can be easily customized for various applications. The following is an example of a generic counter module in VHDL.

```vhdl
entity Counter is
    generic (
        WIDTH : integer := 8  -- Default width of 8 bits
    );
    Port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        count : out STD_LOGIC_VECTOR(WIDTH-1 downto 0)
    );
end Counter;

architecture Behavioral of Counter is
    signal counter_reg : STD_LOGIC_VECTOR(WIDTH-1 downto 0);
begin
    process (clk, reset)
    begin
        if reset = '1' then
            counter_reg <= (others => '0');  -- Reset the counter
        elsif rising_edge(clk) then
            counter_reg <= counter_reg + 1;  -- Increment the counter
        end if;
    end process;

    count <= counter_reg;
end Behavioral;
```

Top Level Modules

Top-level modules, also known as the main modules or the top-level entities, serve as the primary interface for the entire digital design. These modules typically encapsulate the entire functionality of the system and coordinate interactions between various submodules or components. In VHDL, a top-level module is defined as an entity and implemented as an architecture, incorporating all necessary inputs, outputs, and internal signals required for the system's operation.

The top-level module acts as a centralized control hub, receiving external inputs, processing them through the system, and generating corresponding outputs. It orchestrates the flow of data and control signals throughout the design, facilitating communication between different modules and managing overall system behaviour.

Designers often use top-level modules to instantiate and interconnect lower-level modules or components, organizing the design into a hierarchical structure. This hierarchical approach simplifies the design process, enhances modularity, and promotes code reuse by breaking down complex systems into smaller, more manageable units.

Additionally, top-level modules may include configuration parameters or generics to enable flexibility and customization of the design. These parameters allow designers to tailor the behaviour and functionality of the system without modifying the underlying implementation, enhancing the versatility and scalability of the design.

The following is an example of a top-level module which is instantiating the counter in the previous VHDL example.

```vhdl
entity Top_Module is
    Port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        count : out STD_LOGIC_VECTOR(7 downto 0)
    );
end Top_Module;

architecture Behavioral of Top_Module is
    component Counter is
        generic (
            WIDTH : integer := 8  -- Default width of 8 bits
        );
        Port (
            clk : in STD_LOGIC;
            reset : in STD_LOGIC;
            count : out STD_LOGIC_VECTOR(WIDTH-1 downto 0)
        );
    end component;
begin
    counter_inst : Counter generic map (
        WIDTH => 8  -- Set width parameter to 8 bits
    ) port map (
        clk => clk,
        reset => reset,
        count => count
    );
end Behavioral;
```

By leveraging hierarchical design and module parameterization in VHDL, designers can create scalable and reusable digital systems that are adaptable to diverse requirements and facilitate efficient development processes. Overall, top-level modules play a pivotal role in digital design, serving as the foundation upon which complex systems are built. They provide a centralized interface for system control and integration, facilitating efficient development, testing, and maintenance of digital designs.

VHDL Data Types and Conversions

Understanding VHDL data types and conversions is essential for developing robust and efficient digital designs. This chapter explores the various data types available in VHDL and provides insights into performing data conversions between different types.

VHDL Data Types

VHDL offers a rich set of data types to represent different kinds of digital signals and values. These data types include:

1. **Scalar Types:** Scalar types represent single values and include basic types such as BIT, BOOLEAN, INTEGER, REAL, and TIME.

2. **Composite Types:** Composite types allow grouping of multiple values into a single object. Examples of composite types include arrays (ARRAY) and records (RECORD).

3. **Enumeration Types:** Enumeration types define a set of named values, such as ENUMERATION and SUBTYPE.

4. **Access Types:** Access types provide references to objects in memory, enabling dynamic memory allocation and manipulation.

5. **File Types:** File types are used for file I/O operations within VHDL designs. (Not synthesisable)

Each data type in VHDL serves specific purposes and offers unique capabilities for digital design. Understanding the characteristics and usage scenarios of each type is crucial for effective design implementation.

Conversions

Data and type conversions in VHDL involve transforming data from one type to another. These conversions can be implicit or explicit, depending on the context and compatibility between the source and target types. Examples of data conversions include:

1. **Type Conversion:** Type conversion operations enable transforming data from one type to another using explicit casting operations. These conversions are essential for ensuring compatibility between different types in hardware designs. For example:

```
signal integer_value : INTEGER := 10;
signal bit_vector_value : STD_LOGIC_VECTOR(3 downto 0);
bit_vector_value <= STD_LOGIC_VECTOR(to_unsigned(integer_value,
bit_vector_value'length));
```

2. **Numeric Conversion:** Numeric conversions allow converting between different numeric types, such as INTEGER, REAL, and BIT_VECTOR, suitable for hardware synthesis.
For instance:

```
signal real_value : REAL := 3.14;
signal integer_value : INTEGER;
integer_value <= to_integer(real_value);
```

3. **Bit-Wise Conversion:** Bit-wise conversions involve extracting or concatenating bits from different signals or variables, facilitating data manipulation at the bit level. These conversions are fundamental for implementing bitwise operations in hardware designs. Example:

```
signal data_in : STD_LOGIC_VECTOR(7 downto 0);
signal msb : STD_LOGIC;
msb <= data_in(7);  -- Extracting the most significant bit
```

4. **Enum-to-Integer Conversion:** Enum-to-integer conversions transform between enumeration types and their underlying integer representations, suitable for synthesisable logic. This conversion is useful for interfacing with external systems or performing arithmetic operations. Example:

```
type my_enum is (A, B, C);
signal enum_value : my_enum := B;
signal integer_value : INTEGER;
integer_value <= to_integer(enum_value);
```

Conversions play a vital role in VHDL designs, enabling the transformation of data between different types while maintaining synthesizability for hardware implementation. By leveraging these conversion techniques effectively, designers can develop robust and efficient digital systems suitable for hardware synthesis.

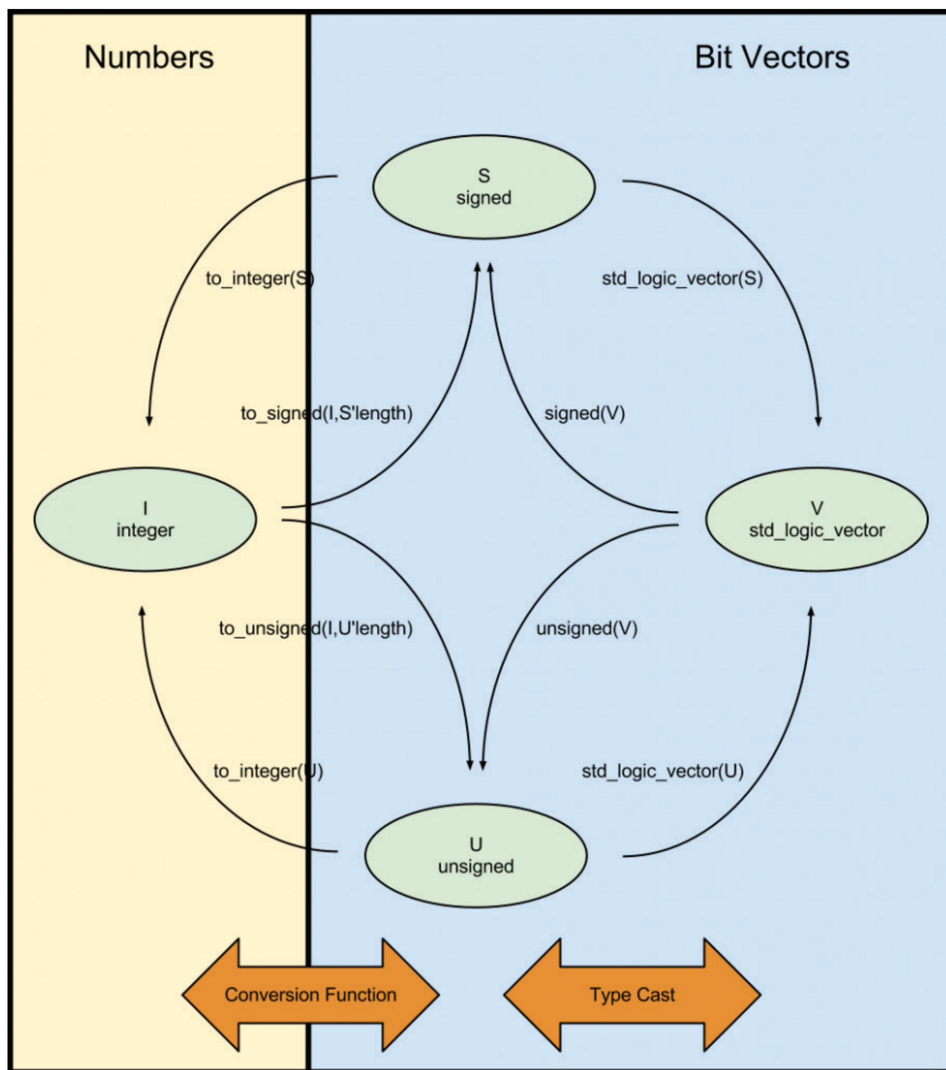


Figure 25: A diagram illustrating how to convert between the most common VHDL types. This diagram was taken from a great resource for VHDL type conversions on ***bitweenie.com***.

Advanced VHDL Techniques

Advanced RTL coding techniques in VHDL enable designers to create highly optimized and scalable register-transfer level (RTL) designs. This section explores advanced RTL coding methodologies:

Pipeline Design: Pipeline design techniques enhance system throughput and performance by breaking down complex operations into smaller stages, enabling parallel processing and reducing latency.

FIFO Design: First-In-First-Out (FIFO) design methodologies facilitate efficient data buffering and management, ensuring smooth data flow and preventing data loss or overflow.

Finite State Machine (FSM) Design: FSM design techniques enable the implementation of complex control logic and state-dependent behaviour, enhancing system functionality and versatility.

Clock Domain Crossing (CDC) Mitigation: CDC mitigation strategies address timing issues arising from data transfer between different clock domains, ensuring reliable and synchronized operation across the entire system.

Advanced Synthesis Optimization Techniques

Advanced synthesis optimization techniques empower designers to maximize the performance and efficiency of their designs during the synthesis process. This section covers advanced optimization methodologies, including:

Resource Sharing: Resource sharing techniques reduce hardware resource utilization by identifying and consolidating common logic elements, minimizing area overhead and improving design efficiency.

Clock Gating: Clock gating strategies optimize power consumption by selectively enabling or disabling clock signals based on specific conditions, reducing dynamic power dissipation in the digital system.

Area Optimization: Area optimization methodologies focus on minimizing the physical footprint of the design by optimizing logic placement, routing, and resource allocation to achieve compact and efficient designs.

Timing Closure Techniques: Timing closure techniques ensure that the design meets timing constraints and achieves reliable operation by optimizing critical paths, balancing clock skew, and resolving timing violations.

Chapter Five Summary

At the heart of leveraging FPGA capabilities is the proficiency in Hardware Description Languages (HDLs), and VHDL stands out as a preferred option among engineers and designers. We covered essential principles of VHDL, investigated advanced methodologies, and examined real-world applications, providing a solid foundation in FPGA development.

Chapter Six: Design Techniques and Best Practices

In this chapter of the FPGA Handbook the essential design techniques and best practices for developing efficient and reliable digital systems is discussed.

What to expect in Chapter Six:

- RTL Design
- Finite State Machines
- Timing Constraints
- Pipelining and Parallelism
- Power Optimization

RTL Design

RTL design serves as the foundation for describing digital circuits using registers and combinational logic. It involves partitioning the design into data path and control logic, ensuring synchronous operation, and adhering to coding guidelines for clarity and maintainability. The following is an example of a Register-Transfer Level Flip-Flop:

```
entity D_FF is
    Port (
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        d : in STD_LOGIC;
        q : out STD_LOGIC
    );
end D_FF;

architecture RTL of D_FF is
begin
    process (clk, rst)
    begin
        if rst = '1' then
            q <= '0';  -- Reset the flip-flop
        elsif rising_edge(clk) then
            q <= d;  -- Update the flip-flop state
        end if;
    end process;
end RTL;
```

In this example, we define a D flip-flop entity with clock (clk), reset (rst), data input (d), and output (q) ports. In the architecture, a process sensitive to the clock and reset signals is used to update the output (q) based on the input data (d) on the rising edge of the clock signal (clk), while also handling reset (rst) to initialize the flip-flop state.

Finite State Machines (FSMs)

FSMs are fundamental models in digital design used to describe systems with sequential behaviour. They are particularly useful for representing systems that transition between different states based on inputs and internal conditions. FSMs can be classified into two main types: Mealy machines and Moore machines.

Mealy Machines: In a Mealy machine, both the outputs and the state transitions are dependent on the current state and the inputs. The output is a function of both the current state and the input. This type of FSM is characterized by its ability to produce outputs that can change asynchronously with respect to the inputs.

Moore Machines: Unlike Mealy machines, Moore machines have outputs that are only dependent on the current state. The state transitions are determined solely by the current state and the inputs. This type of FSM is known for its synchronous output behaviour, where the output changes only at the clock edge.

FSMs are commonly represented using state transition diagrams, where nodes represent states, and directed edges represent transitions triggered by inputs. Each state is associated with specific outputs or actions that occur when the system is in that state. These diagrams provide a visual representation of the system's behaviour and aid in understanding and designing FSMs.

FSMs find applications in various areas of digital design, including control systems, protocol implementations, and stateful data processing. They are versatile tools that allow designers to model complex behaviour in a systematic and structured manner, facilitating the development of efficient and reliable digital systems. Understanding FSMs is essential for anyone involved in digital design, as they form the basis for many advanced design techniques and methodologies.

```
-- Example: Moore Finite State Machine
entity Moore_FSM is
    Port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        input : in STD_LOGIC;
        output : out STD_LOGIC
    );
end Moore_FSM;

architecture RTL of Moore_FSM is
    type state_type is (S0, S1, S2);
    signal state : state_type := S0;
begin
    process (clk, reset)
    begin
        if reset = '1' then
            state <= S0;  -- Reset the FSM state
        elsif rising_edge(clk) then
            case state is
                when S0 =>
                    if input = '1' then
                        state <= S1;
                    else
                        state <= S0;
                    end if;
                when S1 =>
                    state <= S2;
                when S2 =>
                    state <= S0;
            end case;
        end if;
    end process;

    process (state)
    begin
        case state is
            when S0 => output <= '0';
            when S1 => output <= '1';
            when S2 => output <= '0';
        end case;
    end process;
end RTL;
```

In this example, we define a Moore FSM entity with clock (clk), reset (reset), input (input), and output (output) ports. The architecture comprises two processes: one for state transition and another for output generation based on the current state. The FSM transitions between states based on input conditions and generates outputs accordingly.

Timing Constraints

Timing constraints are essential for ensuring correct operation and timing closure of digital designs. They define the timing requirements and constraints for signals in the design, guiding the synthesis and place-and-route processes to meet timing objectives.

```
-- Example: Timing Constraint for Clock Period
create_clock -period 10 -name clk
```

This timing constraint specifies a clock signal named clk with a period of 10 units of time. It guides the synthesis and place-and-route tools to optimize the design to meet this timing requirement, ensuring proper operation of synchronous elements in the design. A large number of timing constraints exist and are very important in complex designs.

It is important to note that methods and syntax for timing constraints are largely specific to different vendors. Take a look at [UG903 Vivado Design Suite User Guide: Using Constraints for AMD FPGAs](#).

Pipelining and Parallelism

Pipelining and parallelism techniques enhance system performance and throughput by breaking down operations into smaller stages or executing multiple tasks concurrently. They are essential for optimizing the efficiency and speed of digital systems.

```
-- Example: Pipelined Adder
architecture RTL of Pipelined_Adder is
begin
    process (clk)
    begin
        if rising_edge(clk) then
            -- Pipeline stage 1
            sum_1 <= a + b;
            -- Pipeline stage 2
            sum_2 <= sum_1 + c;
            -- Pipeline stage 3
            result <= sum_2;
        end if;
    end process;
end RTL;
```

In this example, we implement a pipelined adder with three stages: input addition (sum_1), intermediate addition (sum_2), and final result (result). Each stage operates on the rising edge of the clock signal (clk), enabling concurrent execution of multiple additions and improving overall system throughput.

Power Optimization

Power optimization techniques aim to minimize power consumption while maintaining performance and functionality in digital designs. They include strategies such as clock gating, voltage scaling, and resource optimization to achieve power-efficient designs.

```vhdl
-- Example: Clock Gating
architecture RTL of Power_Optimized_Design is
begin
    process (clk, enable)
    begin
        if enable = '1' then
            -- Enable the clock signal
            clk_gated <= clk;
        else
            -- Disable the clock signal
            clk_gated <= '0';
        end if;
    end process;
end RTL;
```

In this example, we implement clock gating to selectively enable or disable the clock signal (clk_gated) based on the enable signal. By gating the clock when it is not needed, power consumption is reduced, improving overall power efficiency in the design.

AMD devices incorporate dedicated clock networks designed to offer large-fanout, low-skew clocking resources. The inclusion of fine-grained clock gating techniques in HDL code can impair functionality and hinder the effective utilization of these dedicated clocking resources. Consequently, AMD advises against implementing clock gating constructs in the clock path when writing HDL for their devices. Instead, it is recommended to manage clocking by employing coding techniques that infer clock enables to deactivate sections of the design, whether for functionality or power optimization purposes. (UltraFast Design Methodology Guide for FPGAs and SoCs *(UG949)*).

In VHDL, optimizing for size involves implementing strategies to reduce the hardware resources required by the design. One approach is to adopt a modular design methodology, breaking down the system into smaller, reusable modules. This not only enhances organization but also includes redundancy, leading to a more compact overall design. Parameterization of modules allows for configurability, enabling the reuse of modules across different contexts and mitigating the need for multiple similar modules, thus further optimizing size. Moreover, careful consideration of data types based on the required range and precision of signals or variables can contribute to size reduction, as employing smaller data types where feasible diminishes the overall design footprint. Efficiency in VHDL code is paramount; designing code that is concise and rid of unnecessary operations or redundancy aids in minimizing design size.

Additionally, eliminating extra and unneeded signals and variables, streamlining state machines, optimizing design hierarchy, and fostering resource sharing between modules all play pivotal roles in reducing the size of VHDL designs. Ensuring sequential logic elements are properly clocked and avoiding the use of latches further aids in size optimization. Lastly, leveraging synthesis optimization options provided by synthesis tools can automate certain optimization processes, aiding in the quest for a compact VHDL design while still meeting functionality and performance criteria.

Chapter Six Summary

This chapter looked at fundamental elements and techniques of digital design, like Parallelism and Timing Constraints.

Chapter 7: Testing and Verification

FPGA verification ensures that the FPGA design meets functional requirements, operates correctly under various conditions, and meets timing constraints before it is manufactured and deployed in the target application.

What to expect in Chapter Seven:

- Simulation-Based Testing
- Testbench Development
- On-Chip Logic Analyzer

Simulation-Based Testing in FPGAs

Simulation-based testing represents a pivotal phase in the development cycle of FPGAs, allowing designers to rigorously verify the functionality and performance of their designs prior to hardware deployment. This method entails creating a virtual model of the FPGA design and executing simulations to identify and rectify potential issues.

Fundamental Concepts in Simulation-Based FPGA Testing

Design Entry: The initial phase of FPGA design involves capturing the intended functionality using hardware description languages (HDLs) such as VHDL or Verilog. This phase may also include the integration of Intellectual Property (IP) cores and instantiated blocks.

Simulation Models: These models serve as abstract representations of the FPGA design, utilized during simulations to emulate the behavior of the actual hardware. Simulation models facilitate the prediction of design performance under various operational scenarios.

Testbenches: Testbenches are HDL codes designed to provide a controlled environment for stimulating and verifying the design. They encompass stimulus generation (input signals), the instantiation of the design under test (DUT), and mechanisms to compare the outputs against expected results. This is illustrated graphically in Figure 25.

Functional Simulation: This type of simulation focuses on verifying the logical correctness of the design, ensuring that it performs the intended operations without considering timing constraints.

Timing Simulation: Timing simulations account for propagation delays, setup and hold times, and other timing constraints, ensuring that the design operates correctly at the desired clock speeds.

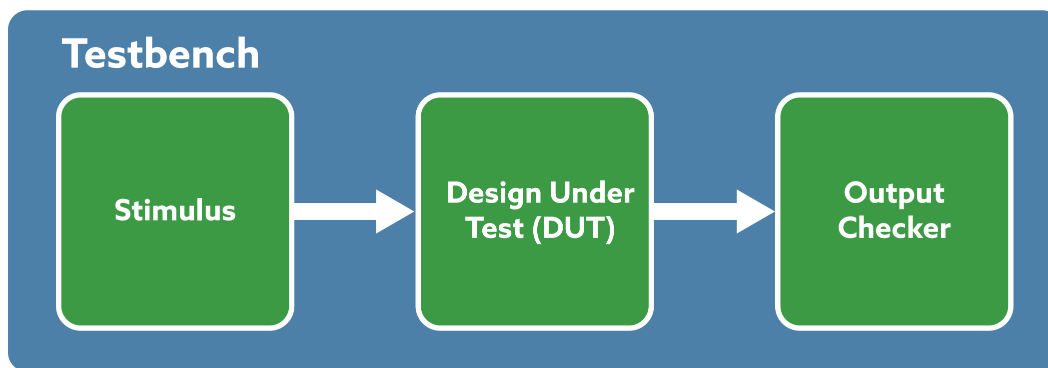


Figure 26: Testbench High level View

Tools for Simulation-Based Testing in FPGA Design

Various tools are employed for simulation-based testing of FPGAs, ranging from generic HDL simulators to specialized tools integrated within FPGA development environments.

ModelSim

ModelSim is a widely used HDL simulator that supports both functional and timing simulations, accommodating VHDL, Verilog, and SystemVerilog.

Features:

1. Advanced debugging capabilities, including waveform viewing, breakpoints, and signal tracing.
2. Integration with numerous FPGA design tools and environments.
3. Comprehensive support for multiple HDL languages.
4. Efficient handling of large designs and complex testbenches.

AMD Vivado Simulator

Vivado, AMD's integrated development environment (IDE) for FPGA design, includes the Vivado Simulator.

Features:

1. Seamless integration with the Vivado Design Suite, providing a unified workflow from design entry to simulation.
2. Support for mixed-language simulations, including VHDL, Verilog, and SystemVerilog.
3. A rich set of debugging tools, such as waveform viewers and logic analyzers.
4. Capabilities for both functional and timing simulations, ensuring thorough verification.
5. Free of Charge

A tutorial for using the AMD Vivado simulator is linked ***here***.

AMD ISE Simulator (ISim)

For older AMD FPGA families, the Integrated Synthesis Environment (ISE) offers the ISim simulator.

Features:

1. Support for VHDL and Verilog simulations.
2. Integration within the ISE design suite, facilitating a smooth transition from design to simulation.
3. Suitable for both functional and timing simulations.
4. Basic debugging tools, though less advanced compared to Vivado.

Simulation Workflow in the AMD Ecosystem

Various tools are employed for simulation-based testing of FPGAs, ranging from generic HDL simulators to specialized tools integrated within FPGA development environments.

1. Design Entry: The FPGA design is created using Vivado or ISE, involving HDL code writing, IP core integration, and constraint definition.

2. Testbench Creation: A testbench is developed to apply stimulus and verify the outputs of the design. The testbench is typically written in the same HDL as the design.

3. Simulation Setup: The simulation environment is configured in Vivado or ISE, which includes selecting the appropriate simulator (Vivado Simulator or ISim), specifying the testbench, and setting up simulation parameters.

4. Running Simulations:
   - Functional Simulation: Initial simulations are conducted to verify the logical correctness of the design. Functional errors are debugged using waveform viewers and other tools.
   - Timing Simulation: Post-synthesis and implementation, timing simulations are performed to ensure the design meets timing constraints. This step verifies the design's correctness under real-world timing conditions.

5. Debugging and Verification: Debugging tools provided by the simulator are used to trace signals, set breakpoints, and inspect waveforms. The design and testbench are iteratively refined to resolve issues.

6. Validation: Upon successful simulation, the design proceeds to further stages such as synthesis, implementation, and eventual hardware testing on the FPGA.

Simulation-based testing is an important component of FPGA design, providing a controlled environment for verifying the functionality and performance of designs prior to hardware implementation. Tools such as ModelSim and the simulators within the AMD ecosystem (Vivado Simulator and ISim) offer comprehensive features for both functional and timing simulations,

ensuring robust and reliable FPGA designs. By thoroughly testing designs in a simulated environment, developers can identify and resolve issues early in the development process, thereby saving time and resources.

Hardware/Software Cosimulation

Hardware/software cosimulation is an essential technique in the design and verification of complex systems, particularly for AMD devices. This approach integrates the simulation of hardware components with software execution, ensuring both interact correctly and perform optimally within a unified environment. Tools like the AMD Vivado Design Suite, which includes the Vivado Simulator and System Generator, facilitate this process by allowing for concurrent simulation of HDL designs and embedded software. Additionally, the AMD Vitis Unified Software Platform and Vitis Model Composer enable comprehensive cosimulation by providing environments where hardware accelerators can be simulated alongside software. This methodology emphasizes a co-design approach, where hardware and software are developed concurrently with iterative testing to ensure optimal integration. Proper partitioning of tasks between hardware and software is crucial, with high-performance, parallel tasks typically assigned to hardware, and control-oriented, complex algorithms handled by software. This integrated approach helps in identifying and addressing issues early in the design process, thereby reducing development time and costs.

Testbench Development

Testbench development is a fundamental aspect of the verification process in VHDL design. A testbench provides a controlled environment to apply stimuli to the DUT and observe its behaviour, ensuring that the design meets its specifications. This review explores the key components and methodology of developing testbenches in VHDL, along with a simplified example. The key components of a testbench in VHDL are given below.

Key Components of a VHDL Testbench

Entity Declaration: Unlike the DUT, the testbench entity typically has no ports because it is a self-contained verification environment.

Architecture Body: This section contains the actual testbench implementation. It includes signal declarations, instance of the DUT, stimulus generation, and output checking mechanisms.
Signal Declarations: Signals are used to connect the DUT and to generate stimulus inputs and capture outputs.

Instance of the DUT: The DUT is instantiated within the testbench architecture, connecting internal signals to its ports.

Stimulus Generation: This involves creating the necessary input signals to exercise the DUT. Stimuli can be applied using concurrent statements (processes) that generate waveforms.

Output Checking: The responses from the DUT are monitored and compared against expected results to verify correct behaviour. This can be done manually or using automated checking mechanisms.

Methodology of Testbench Development

Define Objectives: Clearly outline what aspects of the DUT need to be verified, such as functionality, performance, and timing.

Plan Test Cases: Develop a comprehensive set of test cases to cover all possible scenarios, including normal operation, boundary conditions, and erroneous inputs.

Write the Testbench: Implement the testbench in VHDL, ensuring that it accurately reflects the planned test cases. This includes creating processes for stimulus generation and output checking.

Run Simulations: Use a VHDL simulator to execute the testbench and observe the behaviour of the DUT. Record the simulation results for analysis.

Analyse Results: Compare the observed behaviour against expected outcomes. Identify and debug any discrepancies to ensure the DUT operates correctly under all tested conditions.

Example of a Simple VHDL Testbench

Consider a simple DUT that performs a binary addition. The following example illustrates a basic VHDL testbench for this DUT.

DUT:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder_vhdl_code is
 Port ( A : in STD_LOGIC;
 B : in STD_LOGIC;
 Cin : in STD_LOGIC;
 S : out STD_LOGIC;
 Cout : out STD_LOGIC);
end full_adder_vhdl_code;

architecture gate_level of full_adder_vhdl_code is

begin

 S <= A XOR B XOR Cin ;
 Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B) ;

end gate_level;
```

Test Bench:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder_tb is
end full_adder_tb;

architecture behavior of full_adder_tb is

    -- Signal declarations for testbench
    signal A : STD_LOGIC := '0';
    signal B : STD_LOGIC := '0';
    signal Cin : STD_LOGIC := '0';
    signal S : STD_LOGIC;
    signal Cout : STD_LOGIC;

    -- Component declaration for the Unit Under Test (UUT)
    component full_adder_vhdl_code
        Port (
            A : in STD_LOGIC;
            B : in STD_LOGIC;
            Cin : in STD_LOGIC;
            S : out STD_LOGIC;
            Cout : out STD_LOGIC
        );
    end component;

begin
    -- Instantiate the Unit Under Test (UUT)
    uut: full_adder_vhdl_code
        Port map (
            A => A,
            B => B,
            Cin => Cin,
            S => S,
            Cout => Cout
        );

    -- Stimulus process
    stim_proc: process
    begin
        -- Test case 1: 0 + 0 + 0
        A <= '0'; B <= '0'; Cin <= '0';
        wait for 10 ns;
        assert (S = '0' and Cout = '0') report "Test case 1 failed"
severity error;

        -- Test case 2: 0 + 0 + 1
        A <= '0'; B <= '0'; Cin <= '1';
        wait for 10 ns;
        assert (S = '1' and Cout = '0') report "Test case 2 failed"
severity error;

        -- Test case 3: 0 + 1 + 0
        A <= '0'; B <= '1'; Cin <= '0';
        wait for 10 ns;
        assert (S = '1' and Cout = '0') report "Test case 3 failed"
severity error;

        -- Test case 4: 0 + 1 + 1
        A <= '0'; B <= '1'; Cin <= '1';
        wait for 10 ns;
```

On-Chip Logic Analyser

An On-Chip Logic Analyzer is a powerful debugging and verification tool used within FPGA and SoC designs to monitor and capture internal signals during operation. This tool is integral for real-time analysis and troubleshooting, allowing developers to observe the behaviour of their digital designs in situ, which significantly enhances the debugging process compared to traditional external logic analysers. In particular, AMD provides a sophisticated On-Chip Logic Analyzer known as the Integrated Logic Analyzer (ILA) as part of its Vivado Design Suite.

The AMD ILA is embedded within the FPGA design and offers deep visibility into the internal state of the FPGA without needing to route signals to external pins. The ILA core can be instantiated within the FPGA design, in fact multiple ILAs can be instantiated in the same design, where it can probe and capture internal signal states based on user-defined trigger conditions. The captured data is then analysed using the Vivado toolset, which provides a graphical interface for viewing and interpreting the signal waveforms. This in-situ debugging capability is especially valuable for complex designs where external access to internal signals is limited or impractical.

One of the key features of the AMD ILA is its configurability. Users can specify which signals to monitor, set trigger conditions, and determine the depth of the capture memory. This allows for tailored debugging sessions focused on specific areas of interest within the design. The ILA supports a wide range of triggering options, including basic edge triggers, logical combinations of signals, and even sequential triggers, enabling the capture of complex scenarios that may lead to bugs or unexpected behaviour. Moreover, the ILA core can operate at the full speed of the FPGA, ensuring that high-speed signals are accurately captured and analysed.

The integration of the ILA within the Vivado Design Suite further enhances its utility. Vivado provides a seamless workflow for inserting the ILA core into the design, compiling the FPGA configuration, and subsequently analysing the captured data. The tool's interface allows for real-time interaction with the running design, enabling users to adjust trigger conditions and probe settings on-the-fly. This dynamic capability is crucial for iterative debugging and refinement, allowing developers to quickly hone in on issues and verify fixes without lengthy design re-implementations. A user guide of the AMD ILA is titled UG936, chapter 10 has a very interesting example of this.

Integrated Logic Analyzers (ILAs) can be built into intellectual property (IP) cores, like the video converter IPs provided by Digilent, such as dvi2rgb or rgb2dvi. These ILAs use a generic and a generate statement, or similar methods, allowing users to easily enable them by ticking a box in the IP configuration. This adds ILAs to important signals, such as phase-locked loop (PLL) locks, making it easier to monitor and debug the system.

Chapter Seven Summary

FPGA verification is the process of ensuring that a design behaves as intended and meets its functional requirements before it is fabricated and deployed. Verification involves testing the design thoroughly to detect and correct errors, bugs, or unintended behaviors that could affect the performance or reliability of the FPGA in its intended application.

Chapter 8: Intellectual Property (IP) Cores and Design Reuse

IP (Intellectual Property) cores in FPGAs are pre-designed, reusable blocks of logic or functions that simplify and expedite the FPGA design process. These cores encapsulate specific functionalities, allowing designers to incorporate complex operations without having to design them from scratch. The use of IP cores not only accelerates development but also reduces cost and risk by leveraging proven and verified components. This approach is especially beneficial in modern FPGA development, where time-to-market pressures and design complexities continue to increase.

What to expect in Chapter Eight:

- Types of IP Cores
- AXI Interfaces
- Licensing and Integration
- Design and Reuse Strategies
- High-Level Synthesis (HLS)

Types of IP Cores

Basic Functional Cores are the fundamental building blocks used in FPGA designs. These include simple arithmetic operations like adders, multipliers, and accumulators, as well as essential components like counters, shift registers, and logical gates. These cores provide the basic functionalities required in almost every digital system, serving as the foundation upon which more complex systems are built. By utilizing these pre-verified components, designers can focus on higher-level design and system integration tasks, rather than spending time on elementary logic design. Additionally, infrastructure like AXI interconnects and processor subsystem resets further support the efficient integration and operation of these cores. Clocking wizard is a basic function that simplifies the configuration of complex clocking primitives as well.

Peripheral Cores provide interfaces to standard peripheral devices, facilitating communication between the FPGA and external hardware. Common peripheral cores include UART (Universal Asynchronous Receiver-Transmitter), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), and GPIO. These cores enable the FPGA to interact with sensors, actuators, storage devices, and other peripherals, making them crucial for embedded system applications. By integrating peripheral IP cores, developers can easily extend the functionality of their FPGA designs and ensure compatibility with a wide range of external devices.

Communication Cores are designed to handle various communication protocols and standards, enabling high-speed data transfer and networking capabilities. These cores include Ethernet for networking, CAN (Controller Area Network) for automotive applications, USB (Universal Serial Bus) for device interfacing, and PCIe (Peripheral Component Interconnect Express) for high-speed data exchange. Additionally, memory interface cores such as DDR (Double Data Rate) and Flash memory controllers are essential for managing data storage and retrieval. Communication IP cores are critical in applications that require reliable and efficient data exchange between the FPGA and other system components or networks.

Processor Cores provide embedded processing capabilities within FPGA designs. These can be soft processors like the AMD MicroBlaze, which are implemented using the FPGA's programmable logic, or hard processors like the ARM cores embedded in AMD Zynq SoCs. Soft processors offer flexibility, allowing customization to specific application needs, while hard processors provide higher performance and efficiency. Processor cores enable the FPGA to execute software programs, making them suitable for complex applications that require both hardware acceleration and software programmability.

Benefits and Integration

The integration of IP cores into FPGA designs is facilitated by design tools provided by FPGA vendors, such as AMD's Vivado Design Suite. These tools offer a library of IP cores and provide a graphical interface for configuring and integrating them into the design. The use of IP cores helps streamline the design process, allowing designers to focus on system-level challenges rather than low-level implementation details. Moreover, IP cores from reputable vendors are thoroughly tested and optimized, ensuring reliability and performance in the final design.

In summary, IP cores are a vital component of FPGA design, offering reusable, pre-verified building blocks that enhance development efficiency and reduce time-to-market. By leveraging a wide range of available IP cores—from basic functional units to complex communication and processing systems—designers can create sophisticated and robust FPGA-based applications more effectively.

AXI Interfaces, AXI4, AXI4-Lite, AXI4-Stream

The AXI (Advanced eXtensible Interface) protocol, part of ARM's AMBA (Advanced Microcontroller Bus Architecture) specification, is extensively used in FPGA designs for interconnecting functional blocks within a SoC. AMD, a leading FPGA manufacturer, incorporates AXI interfaces in its design tools to facilitate high-performance, flexible, and scalable communication. The AXI protocol family includes several variants—AXI4, AXI4-Lite, and AXI4-Stream—each tailored to specific applications and performance requirements.

AXI4 (Advanced eXtensible Interface 4)

AXI4 is the most comprehensive version of the AXI protocol, designed for high-performance

memory-mapped communication. It supports burst transactions, allowing multiple data transfers with a single address phase, significantly enhancing data throughput. The flexibility of AXI4 is evident in its support for a wide range of data widths, from 8 bits to 1024 bits, and various addressing modes, including 32-bit and 64-bit. Additionally, AXI4 can handle burst lengths of up to 256 data transfers per burst. This protocol separates the address and data phases, enabling independent control and data transfer channels, which facilitates pipelining and improves overall efficiency. AXI4's ability to manage multiple outstanding transactions allows for high concurrency and optimal use of bus bandwidth, making it ideal for complex, high-speed data transfer applications in FPGA designs.

AXI4-Lite

AXI4-Lite is a simplified subset of the AXI4 protocol, optimized for scenarios where simplicity and minimal resource usage are more critical than high data throughput. It is designed primarily for accessing control and status registers in peripherals. Unlike AXI4, AXI4-Lite supports only single data transfers per address, which reduces complexity but limits throughput. This streamlined approach results in reduced resource utilization, making AXI4-Lite ideal for low-bandwidth interfaces and control paths within an SoC. The ease of implementation associated with AXI4-Lite allows designers to quickly and efficiently integrate simple peripherals and control logic into their FPGA designs without the overhead of managing complex data transfers.

AXI4-Stream

AXI4-Stream is specialized for high-speed, streaming data applications, where continuous, high-bandwidth data flow is essential without the need for addressing overhead. Unlike AXI4 and AXI4-Lite, AXI4-Stream eliminates the address phase, simplifying the protocol and reducing latency, which is critical for applications such as video processing, data acquisition, and network data streams. The protocol supports flexible data widths, enabling efficient use of available bandwidth and accommodating various data formats. Additionally, AXI4-Stream incorporates flow control mechanisms, allowing the receiver to manage data flow effectively and prevent buffer overflows. This makes AXI4-Stream particularly suitable for applications requiring real-time data streaming and high-throughput data processing. Furthermore, AXI4-Stream is significantly simpler, making it an excellent starting point for learning about other AXI interfaces. It relies on a straightforward handshake mechanism, with the core protocol consisting only of data, ready, and valid signals. In contrast, AXI4 and AXI4-Lite utilize the same handshake mechanism but with multiple channels, interdependencies between channels, and various sideband signals, adding complexity to their protocols.

Licensing and Integration

Licensing of IP Cores

Licensing is a crucial consideration when incorporating IP cores into FPGA designs. IP cores can be sourced from various providers, including FPGA vendors, third-party developers, or open-source

communities, each with distinct licensing models. Vendor-provided IP cores, such as those offered by AMD within the Vivado Design Suite, often come with licenses tied to the use of the vendor's tools and devices. Some of these cores are included free of charge with the development tools, while others require separate licensing fees, which may come with support and regular updates. Third-party IP cores, on the other hand, typically have their own licensing agreements that might involve upfront costs, royalties, or subscription fees. These third-party cores often offer specialized functionalities or optimizations that complement vendor offerings. Open-source IP cores, licensed under terms like GPL, LGPL, the MIT license or Apache, allow free use and modification but may impose conditions on distribution and derivative works. Just like when using third-party source code or libraries in software, understanding these licensing terms is essential to ensure compliance and to leverage the full potential of the IP cores within the FPGA design.

Integration of IP Cores

Integrating IP cores into FPGA designs is a streamlined process facilitated by modern FPGA design tools. The first step involves selecting and configuring the required IP cores from the available catalogue, tailoring them to meet specific design requirements such as data bus width or memory type. FPGA design tools, like AMD's Vivado Design Suite, include IP integrator tools that provide a graphical interface for seamless integration. These tools enable designers to connect IP cores and custom logic effortlessly, using drag-and-drop functionality and automated connection suggestions, ensuring compatibility and proper signal routing. This graphical interface is often referred to as a Block Design or Block Diagram, which visually represents the interconnected IP cores and custom logic.

After the integration, the design is synthesized to generate a netlist, followed by simulation to verify the integrated system's functionality. This simulation step is critical for identifying and resolving any integration issues, ensuring that the IP cores and the overall system meet performance and functional specifications. Once verified, the design undergoes implementation, which involves placement and routing to optimize performance and resource usage. The final step is testing the implemented design on actual hardware, using debugging tools like the Integrated Logic Analyzer (ILA) to monitor internal signals and troubleshoot issues.

Maintenance and Updates

Post-integration, maintaining and updating IP cores is essential to ensure the FPGA design remains robust and up-to-date. IP core providers frequently release updates that include bug fixes, performance enhancements, and compatibility adjustments for new FPGA devices and software versions. Regularly integrating these updates into the design helps maintain its reliability and efficiency. Designers must manage these updates diligently, re-integrating and re-testing the IP cores as necessary to ensure continuous optimal performance and to address any emerging issues or improvements in the IP core functionality.

Design Reuse Strategies

Design reuse strategies play a pivotal role in FPGA development, offering significant benefits in terms of efficiency, reliability, and time-to-market. These strategies involve the systematic organization, documentation, and abstraction of design elements to facilitate their reuse across multiple projects or within the same project. Here are some key design reuse strategies commonly employed in FPGA development:

IP Core Libraries: Building and maintaining libraries of reusable IP cores is a fundamental approach to design reuse. These libraries contain verified and validated functional blocks, such as processors, memory controllers, controllers for communication interfaces, and custom logic modules. IP cores within these libraries are typically designed to be parameterizable and configurable, allowing them to be easily adapted to different applications and project requirements. FPGA vendors like AMD often provide extensive IP core libraries as part of their development tools, supplemented by third-party and open-source offerings. Similarly. Board vendors and vendors for other chips commonly used alongside FPGAs (Analog Devices ADCs) also provide libraries.

Modular Design Approach: Adopting a modular design methodology involves breaking down complex systems into smaller, self-contained modules that can be independently developed, tested, and reused. Each module encapsulates a specific functionality or feature, with well-defined interfaces for interaction with other modules. This modular approach promotes design scalability, maintainability, and reusability, as modules can be easily integrated, replaced, or modified without affecting the overall system architecture.

Design Templates and Frameworks: Design templates and frameworks provide reusable structures, architectures, and design patterns tailored to specific application domains or design methodologies. These templates encapsulate best practices, design guidelines, and implementation methodologies, enabling designers to jumpstart their projects and streamline the development process. Templates may include predefined configurations, scripts, and constraints to expedite the setup and implementation of common FPGA designs, such as signal processing algorithms or digital signal processing (DSP) applications.

Parameterized Design Blocks: Parameterization enables the customization of design blocks by specifying parameters such as data widths, memory depths, clock frequencies, and interface protocols at design time. By parameterizing design blocks, designers can create generic, flexible components that can be reused across different projects or instantiated with varying configurations within the same project. Parameterized design blocks enhance design flexibility, reduce duplication of effort, and promote consistency across projects.

Design Abstraction Levels: Design abstraction involves representing complex functionalities at higher levels of abstraction, such as algorithmic or behavioural descriptions, before refining them into lower-level implementations. By abstracting designs at higher levels, designers can focus on system-level functionality and performance requirements without getting bogged down in low-level implementation details. High-level abstractions, such as algorithmic descriptions written in languages like MATLAB or SystemC, can be reused and refined across multiple projects, speeding up

development and facilitating design exploration.

Documentation and Knowledge Management: Effective documentation and knowledge management practices are essential for capturing, cataloging, and disseminating reusable design assets, including IP cores, modules, templates, and design guidelines. Documenting design decisions, implementation details, and best practices enables designers to leverage existing knowledge and experience, fostering collaboration, knowledge sharing, and continuous improvement across design teams and projects.

High-Level Synthesis (HLS)

HLS is a pivotal technology in FPGA development, and AMD offers robust HLS tools as part of its Vivado Design Suite. Engineers transitioning from microcontroller and C programming backgrounds may find AMD's HLS tools particularly beneficial, as they allow for the synthesis of C, C++, and SystemC code directly into FPGA hardware implementations. This abstraction streamlines the design process, enabling designers to focus on algorithmic development and system-level optimization without the need to delve into low-level hardware description languages like VHDL or Verilog.

Alternatively, AMD Model Composer is a powerful tool within the AMD Vivado Design Suite aimed at accelerating the development of complex signal processing algorithms and models for FPGA implementation. It offers a comprehensive environment for algorithm exploration, modelling, and verification, allowing designers to seamlessly transition from high-level algorithm development to FPGA implementation.

With AMD Model Composer, engineers can develop and simulate signal processing algorithms using MATLAB or Simulink, industry-standard tools for algorithm development and simulation. The tool provides extensive support for FPGA-specific optimizations and constraints, enabling designers to achieve optimal performance and resource utilization in their FPGA implementations. By integrating seamlessly with Vivado HLS and the Vivado Design Suite, AMD Model Composer empowers designers to rapidly prototype, refine, and deploy sophisticated signal processing algorithms on AMD FPGAs, significantly reducing time-to-market and accelerating innovation in domains such as wireless communication, digital signal processing, and image processing. More information on AMD's High Level Design tools can be found here.

Chapter Eight Summary

In FPGA development, pre-designed , pre-verified IP cores and design reuse play crucial roles in accelerating and enhancing the efficiency of designing complex digital systems.

Chapter Nine: Real-World Case Studies

What to expect in Chapter Nine:

- Signal Processing
- Automative and Aerospace Applications
- Cryptography and Security

FPGA-based Signal Processing

Signal processing is surely one of the main applications when it comes to FPGAs. The parallelism and ability to increase throughput of sample processing gives FPGAs n advantage over sequential processing-based systems such as microcontrollers. A system which I have worked on myself included wireless transmission of audio signals from one FPGA based SDR to another with minimum delay. The block diagrams below show how these transmission and reception system where fit into a ZYNQ 7000 SoC. The SDR hardware was designed as a daughter board to the Eclypse Z7 from Digilent.

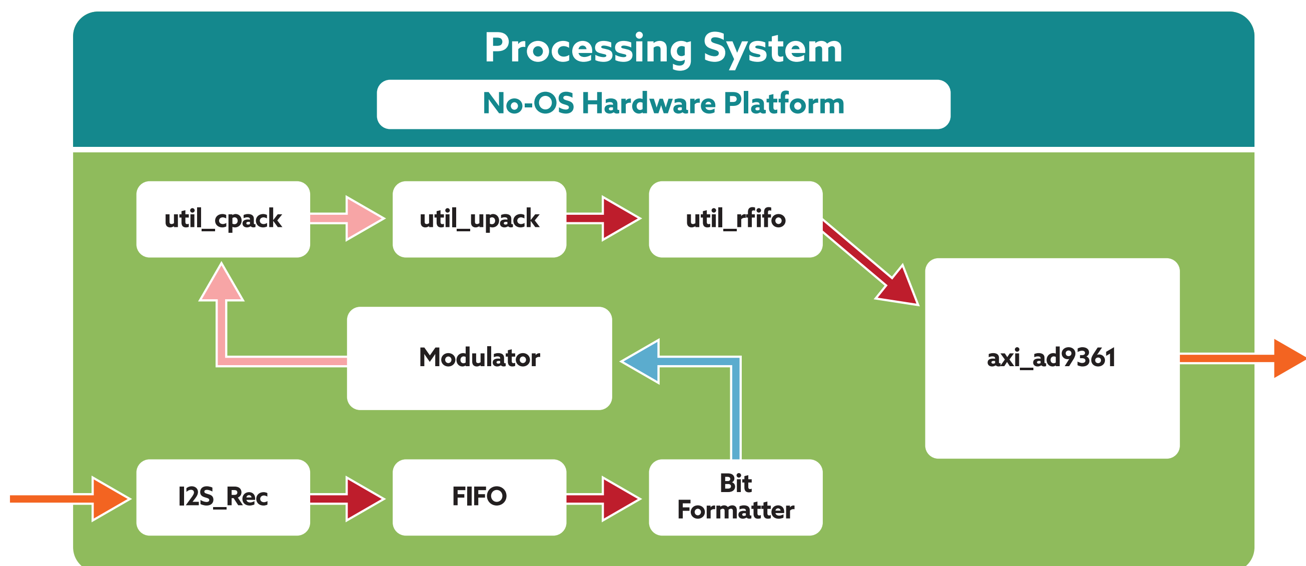


Figure 27: Transmitter Block Diagram

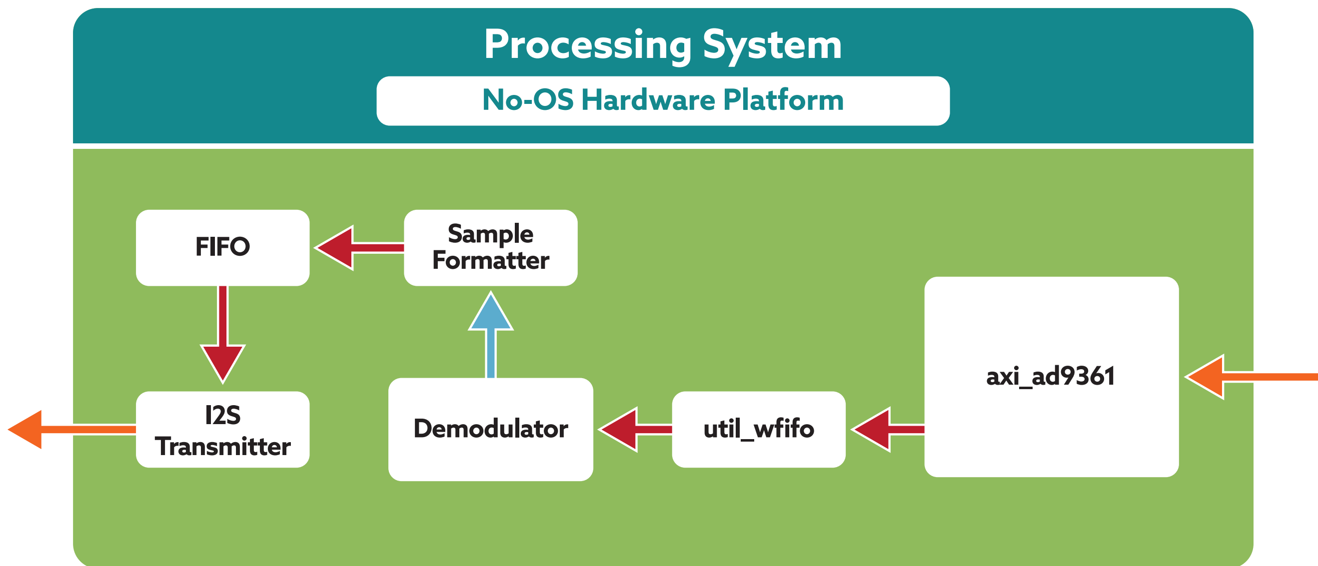


Figure 28: Receiver Block Diagram

The main reason for using FPGAs was the need for high-speed signal processing, Having the parallelism provided by FPGAs helps quite a lot with high-speed processing. The ability to rapidly tweak an algorithm that can run at hardware speeds and the design modularity provided by FPGAs lead to the decision to go for FPGAs quite easy.

Input filtering and averaging in an oscilloscope acquisition chain (Analog Discovery 3, most Analog Discovery Pro devices) help to reduce noise and improve signal analysis. Hardware and software filters are available within the Scope instrument.

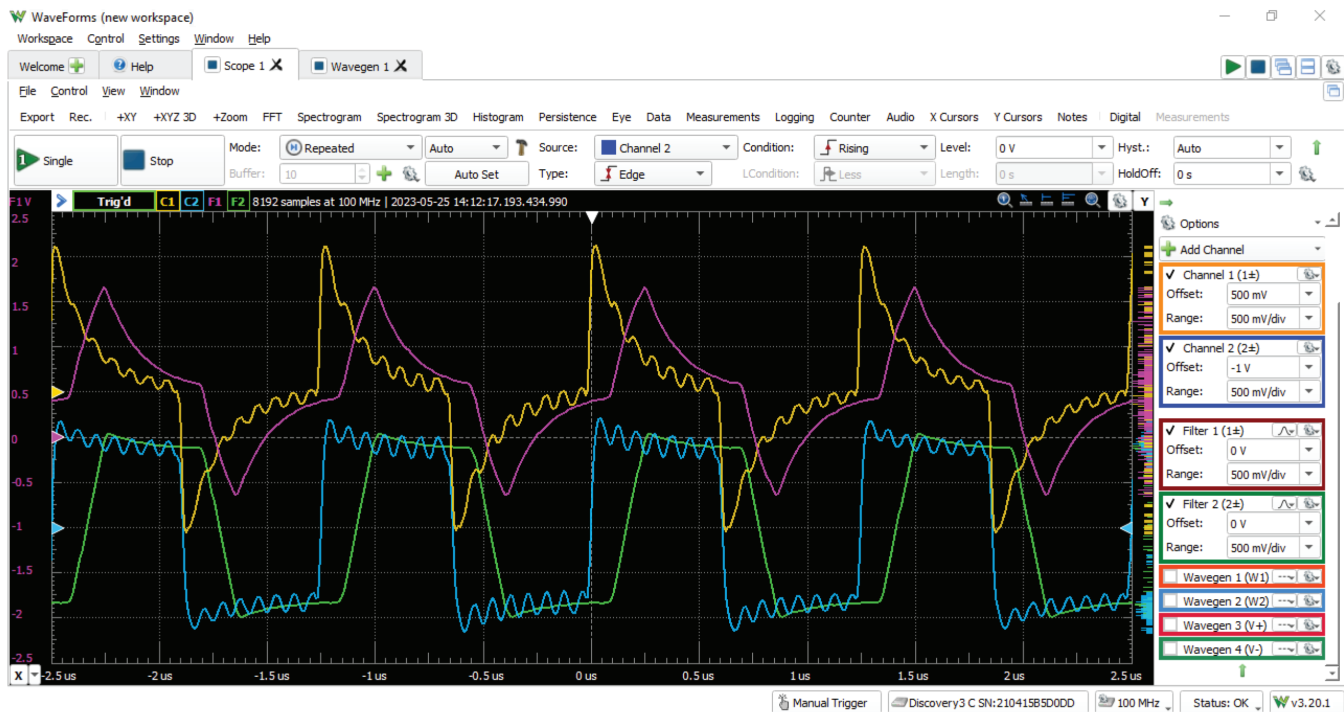


Figure 29: Waveforms Software Filtering

Automotive and Aerospace Applications

FPGAs have always been important in aerospace, offering exceptional flexibility and performance. As technology advances, FPGAs are expected to significantly influence the future of aerospace. FPGAs allow for low latency control loops; they have enhanced processing capabilities. This is extremely important in systems which require significant processing.

Enhanced Processing Capability

A major advancement in FPGA technology is improved processing power. Newer FPGAs have higher logic densities and faster speeds, making them ideal for handling complex algorithms and calculations needed in aerospace. This improvement means FPGAs can now perform tasks that previously required hardware, resulting in more efficient and cost-effective solutions.

Improved Dependability and Safety

Reliability and safety are critical in aerospace because any failure can have serious consequences. FPGAs are known for their reliability, and recent advancements have increased their fault tolerance and error detection capabilities. This makes FPGAs suitable for safety-critical applications like flight control systems and avionics.

Integration with Artificial Intelligence (AI) and Machine Learning (ML)

AI and ML are transforming aerospace by improving flight operations and mission planning. FPGAs are well-suited for AI and ML tasks due to their ability to process multiple tasks quickly and simultaneously. When combined with AI and ML algorithms, aerospace systems can make decisions based on complex data, enhancing safety and efficiency.

Considerations for Size, Weight, and Power (SWaP)

Meeting size, weight, and power (SWaP) requirements is essential in aerospace. Although FPGAs have traditionally consumed more power than other processors, the development of energy-efficient FPGAs has addressed this issue. These low-power FPGAs enable aerospace systems to meet SWaP demands without compromising performance. Power is extremely critical in an environment where you're running on batteries that also take up substantial space, think of an FPGA design in a satellite.

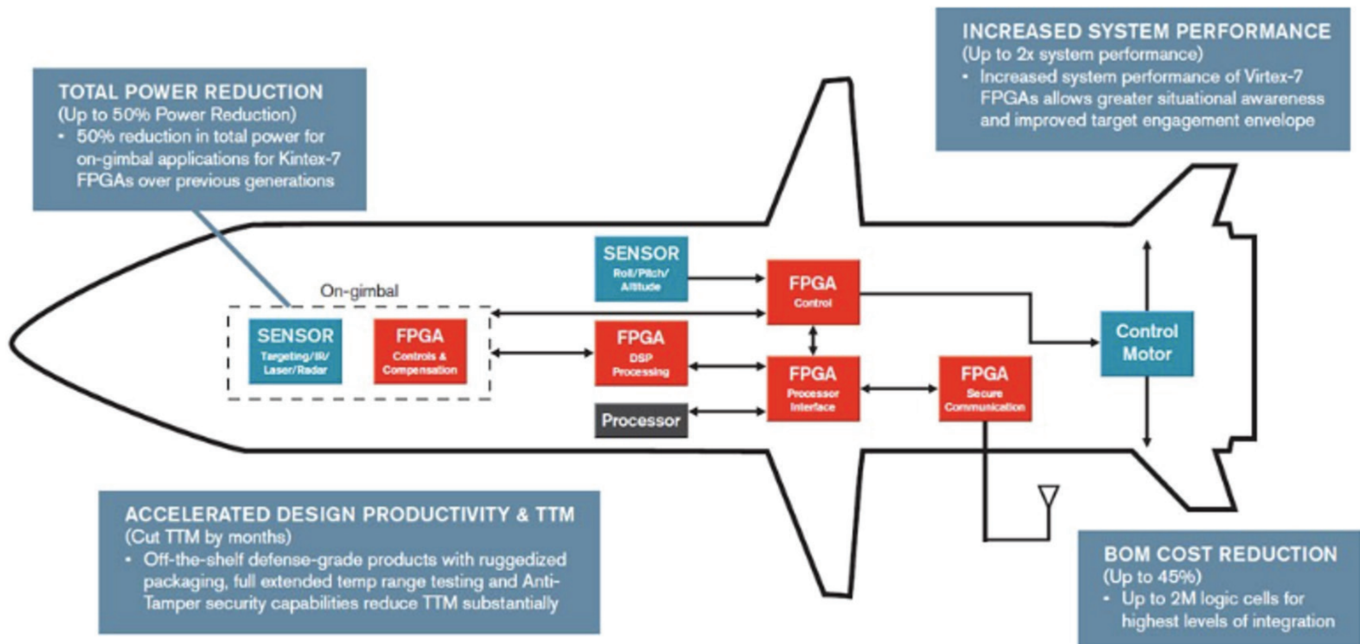


Figure 30: FPGA Applications in Aerospace and Defence (AMD)

Designs like this often require substantial additional effort to actually put that dependability and safety in place. design techniques like keeping three copies of each safety-critical register so that single-event upsets like a cosmic ray flipping a bit in a satellite can be detected and recovered from quickly.

Cryptography and Security

Cryptography is the practice of protecting data and communication from unauthorized access, often referred to as adversaries. It is crucial for modern information security, using cryptographic algorithms and protocols for various purposes such as securing communication over untrusted networks, preventing unauthorized access to stored data, and authenticating users. Cryptography ensures specific security goals, as outlined in the Table below. FPGAs are also used in Networking applications such as directly interfacing with Ethernet PHYs and look for specific patterns in incoming packets, and potentially reroute or block things with a much faster than software might be able to.

SECURITY GOAL	DESCRIPTION
Confidentiality (secrecy)	Ensures that information is accessible only to authorized parties (e.g., the legitimate sender and receiver).
Integrity	Protects information from accidental or intentional changes.
Authenticity	Confirms that an entity is who it claims to be or that data comes from its stated origin.
Non-repudiation	Prevents an entity from denying its previous actions or commitments.

Most current security implementations use cryptographic protocols in software, often through third-party cryptographic libraries on general-purpose processors with known operating systems. While this method is common, there's a growing trend to implement security directly in hardware, particularly in critical embedded systems.

Software-based security has a large "attack surface," meaning many potential targets for attacks, including:

- Operating systems
- Device drivers
- Cryptographic libraries
- Compiler optimizations and microarchitectural changes
- Depth of the software stack
- Cache and memory management
- Key management (e.g., buffer overflow bugs)
- Incomplete control over security algorithms

Software implementations may also struggle with performance (throughput and latency) and power consumption. Additionally, maintaining software security through continuous updates over the system's lifetime can be very challenging and expensive. IoT devices, for example, need ongoing updates for bug fixes throughout their lifecycle, increasing the total cost of ownership.

Recent security concerns have also emerged regarding the underlying processor architecture. Assumptions about the inherent security of processors have been questioned due to vulnerabilities in performance optimizations. Though many issues have been patched, new vulnerabilities may still arise.

Given these challenges, there is a growing trend towards hardware-based security solutions, particularly using FPGAs, which offer greater control and security.

Chapter Nine Summary

FPGAs find application across various industries and sectors due to their versatility, performance, and reconfigurability. The number of industries that FPGAs are implemented in is growing.

Appendix: FPGA Resources and Communities

Books and References

"FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version" by Pong P. Chu - Amazon Link

"Digital Design and Computer Architecture" by David Money Harris and Sarah L. Harris - Amazon Link

AMD Documentation and User Guides - AMD Documentation

VHDL: Pedroni, 1st ed. [3] (now there is a 3rd edition)

VHDL: Armstrong, 1st ed. [4] (I think there are recent editions with more author(s))

Verilog: Thomas & Moorby, 4th ed. [5] (there is a 5th edition)

Verilog: Palnitkar, 1st ed. [6] (there is a 2nd edition)

Verilog: Lee, 2nd ed., [7] (very good read, there is a 3rd edition)

Verilog + FPGA: Stavinov, [8] (very good read)

Nand Land – Website, Linked here

Online Forums and Communities

Digilent Forum: The Digilent Forum is an online community platform where users can discuss and seek support for Digilent products, including FPGAs, microcontrollers, and various development boards. It serves as a valuable resource for students, hobbyists, and professionals to share knowledge, troubleshoot issues, and collaborate on projects.

FPGA Reddit Community: A subreddit dedicated to FPGA technology and discussions. You can find it at https://www.reddit.com/r/FPGA/.

FPGA Central Forums: An online community focused on FPGA discussions, projects, and resources. You can visit the forums at https://www.fpgacentral.com/forum.

AMD Community Forums: AMD's official community forums where you can find support, discussions, and resources related to AMD/Xilinx FPGAs and tools. You can access it at https://support.xilinx.com/.

Altera Forums (now Intel FPGA Forums): Intel's official community forums for discussions on Intel (formerly Altera) FPGAs and development tools. You can find the forums at https://www.intel.com/content/www/us/en/programmable/support/support-resources/support-centers/support-community.html.

VHDL Reddit Community: A subreddit for VHDL programming enthusiasts. You can join discussions on VHDL at https://www.reddit.com/r/VHDL/.

FPGA Developer Forum: A platform for FPGA developers to exchange ideas, ask questions, and share knowledge about FPGA design and development. You can visit the forum at https://www.fpgadeveloper.com/forum.

Intel FPGA (Altera) Forum on Element14: A community forum hosted on Element14 where you can find discussions, resources, and support related to Intel FPGAs (formerly Altera). You can access the forum at https://www.element14.com/community/community/fpga.

VHDL Cafe Forum: An online forum dedicated to VHDL programming language discussions, tutorials, and projects. You can participate in VHDL discussions at http://www.vhdlcafe.com/forum.

FPGA Groups on LinkedIn: Join FPGA-related groups on LinkedIn such as "FPGA Design," "FPGA Engineers," and "VHDL Developers Forum" to network with professionals, share insights, and stay updated on industry trends.

FPGA and VHDL Discord Channels: Explore various Discord channels dedicated to FPGA development and VHDL programming. Joining these channels can help you connect with enthusiasts and professionals in real-time discussions.

Industry Conferences and Events

FPGA Conference (FPGA Forum): An annual conference focused on FPGA technology, trends, and applications. It typically features industry experts, workshops, and showcases of the latest FPGA innovations. You can find more information at https://www.fpga-conference.org/.

International Conference on Field-Programmable Technology (FPT): FPT is a premier conference in the Asia-Pacific region for researchers, engineers, and practitioners interested in reconfigurable technology. Visit their website for event details: http://www.fptconf.com/.

FPGA World Conference: A global conference series that brings together FPGA professionals, researchers, and enthusiasts to share knowledge and insights on FPGA design, applications, and advancements. Check their website for upcoming events: https://www.fpgaworld.com/.

IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM): FCCM is a leading conference for showcasing cutting-edge research in the field of reconfigurable computing. For conference dates and details, visit https://www.fccm.org/.

Embedded Systems Conference (ESC): While not specifically focused on FPGAs, ESC is a significant event where FPGA technology often plays a crucial role in embedded system design. It's a great place to explore the latest trends in embedded systems and FPGA integration. Check out https://esc.embedded.com/ for more information.

Additionally, there are numerous other conferences around the world that tangentially relate to the FPGA field. These events provide valuable opportunities for learning and networking within the broader context of FPGA technology and its applications.