System Level Communication Challenges of Large FPGAs

by

Mustafa Abbas

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

System Level Communication Challenges of Large FPGAs

Mustafa Abbas
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2019

As FPGA capacity and the complexity of the designs implemented on them have grown, so too have system-level communication challenges. At the design level the typical latency of long distance interconnect grows making them the limiting factor in reaching timing for a design. We explore the use of latency-insensitive design (LID) as a solution. Our best approach gained 2x area efficiency and 18% less critical path delay over traditional LID, and come at a minimal speed overhead of 3% compared to a latency-sensitive design.

At the architecture level the number of clocks has grown. It is challenging to supply global clocks to all modules efficiently. Therefore, the clock architecture must be flexible to allow for smaller and more power efficient networks to be built. We present a flexible clock modeling format, built into open source CAD tool VPR, to allow clock impacts and architecture options to be explored.

Acknowledgements

I would like thank my supervisor Prof. Vaughn Betz for guiding me throughout the course of my Masters at the University of Toronto (UoT). In each one of our conversations whether it was on the way to getting a coffee or over a phone meeting I have always learned something new and valuable. Not only that but your friendliness, patience, and encouragement has made the course of my Masters a more enjoyable experience.

I would also like to thank my lab members: Kevin Murray, Andrew Boutros, Sadegh Yazdanshenas, Ibrahim Ahmed, Abed Yassine, Henry Wong, Sameh Attia, Linda Shen, Mohamed Eldafrawy, Mathew Hall, and Yasmin Afsharnajad for their technical feedback throughout my work. I would like to especially thank Kevin Murray who was always willing to give up his time to engage in insightful discussions that were very valuable for this work. I would also like to thank my brother Mohamed Abdelfattah who I have always looked up to and who's mentorship during my masters has helped me improve upon my work.

I would also like to thank my defense committee: Prof. Jason Anderson and Prof. Jonathan Rose, for there valuable feedback on my work.

Finally, I would like to thank my family whose value in academics and constant love and support motivates me to excel in my work everyday.

Funding for my Masters was supported by NSERC/Intel Industrial Research Chair in Programmable Silicon.

Contents

Chapter 1

Introduction

1.1 Motivation

Modern Field-Programmable Gate Arrays (FPGAs) have evolved to be more like Systems-on-Chip (SoC). When first introduced, the architecture of an FPGA was a simple array of logic clusters (LCs) connected together using configurable routing interconnect (multiplexers and wires), surrounded by basic input/output (I/O) buffers on the periphery of the FPGA die, and a few clock signals. By contrast, modern FPGAs are considerably more heterogeneous; in addition to LCs, hardened blocks are interspersed amongst the FPGA fabric, such as block random access memory (BRAM), digital signal processing (DSP) units, and processors. Modern FPGAs also contain high-speed I/Os, such as double data rate (DDR) controllers, PCIe controllers, and Gigabit Ethernet ports. Finally, a large number of clock domains both globally across the chip and in local regions are distributed to clocked elements throughout the FPGA fabric. Not only has the diversity of resources on the FPGA increased, but process technology scaling has allowed for significantly higher logic capacity and speed [78]. The increased variety of resources coupled with an increased logic capacity and speed has enabled designers to integrate large complete applications with many modules and multiple clock domains on a single chip.

To make these applications operate at high speed it is vital for system-level interconnect to keep up with the global communication demand of such large designs. However, while logic speed and capacity on a chip has continued to increase, wire speed for the same *physical* length of wire is not scaling [28]. The mismatch in logic delay relative to interconnect delay causes global connections that traverse a large area of the chip to strongly impact the critical path of designs. For instance, using our own experiments for a large Stratix 10 chip it is not possible to cross the chip at speeds above 100 MHz although the logic can run at 1 GHz. Therefore, it is imperative for the designer to break up long paths in the interconnect by adding pipeline stages. As easy as this sounds, this is a challenge for FPGA designers since a designer is thinking *logically* about the design, assuming that each path between two registers occurs in one clock cycle. Therefore, it is not obvious to the designer that specific signals may need to traverse long wires, thereby requiring extra pipelining. Furthermore, the problem is exacerbated because the final operating speed of the design is not apparent until the final stage of the FPGA computer-aided design (CAD) flow. This means that adding pipeline stages requires many stages of redesign and many iterations of the CAD flow, where each CAD run can be very time consuming and can range from multi-hour to multi-day run-times. In addition, the designer faces the same timing closure problem, ensuring that the

design meets all its timing requirements, again when reusing their intellectual property (IP) on a new FPGA chip, since different process technology and FPGA architecture affect the timing of the design.

There are two ways to solve this issue: one way is to explore architecture changes to the FPGA interconnect fabric, and another is to map designs using a system-level design methodology that can be used to automate the interconnect pipelining late in the CAD flow, when the final timing is known. We explore the latter of the two, by looking at the latency insensitive design (LID) methodology [15] to ameliorate the timing closure and IP reuse issues for the system level.

Another important FPGA design consideration in order to meet the high-speed demand of large applications is that of the clock network architecture. Large complete systems contain many clock domains, requiring the distribution of multiple clocks throughout the FPGA to clocked logic and I/O interfaces. Building clock networks of such scale can consume a large part of the FPGA area as well as its power budget [24, 31]. On the other hand, there is a push for applications to run at higher speeds, which requires carefully designed networks with low skew. A carefully designed high-speed and low-skew clock network will provide a noticeable effect on the operating speed of the FPGA application, whereas a poorly designed one can decrease the amount of the clock period available for useful work causing major timing degradation on the application. Other constraints on the designer can be a direct result of the clock network architecture as well; for example, depending on the flexibility of the network the placed area of the design and wire-length can be directly affected [42]. To explore architecture trade-offs between power, area, skew, and flexibility for clock network architectures, as well as to evaluate different novel clock architectures, we extend the FPGA architectural description in an existing academic FPGA CAD tool Verilog-to-Routing (VTR) [52] to include clock networks.

1.2 Contributions

Process technology scaling has allowed designers to create large complete applications on FPGAs; however, poor wire scaling and the increased number of clocks domains bring about system level communication challenges for modern FPGAs. The two main topics we explore to solve these challenges are system level design methodologies and clock network architecture models. The contributions are as follows:

- *System Level Design:* A proposal of two different LID styles that can better target FPGAs than traditional Carloni-based LID proposed in [16]. Traditional Carloni-based LID uses advanced pipelining elements with wide multiplexers and high-fanout enable signals. We develop alternatives that use simplified pipelining elements to reduce the area and improve the speed of pipelining for LID systems.
- *Clock Network Architecture:* The VTR open-source academic CAD tool for FPGAs [52] has no model for clocks, they are considered ideal and clock nets are not routed. We added a clock model for chip-wide global clocks and the ability to route these clocks using clock trees and/or normal routing resources. This allows us to quantify the area, power, skew and flexibility trade-offs of adding clock networks of different sizes, as well as directly see their affect on the placement area, routing area, and operational speed of the design. Additionally, this is the first step of a clock network architectural exploration and evaluation which allows benchmarking using a full CAD flow.

1.3 Organization

The thesis is organized as follows. Chapter 2 provides background on FPGA architecture and CAD, including clock networks and new architectures such as registered routing. In addition, Chapter 2 also reviews FPGA future trends that motivate our work as well as present a study of prior work presented on system level design and clock architecture modeling for FPGAs. Chapter 3 and Chapter 4 contain our contributions on latency insensitive design and clock architecture modeling respectively. Finally, Chapter 5 highlights conclusions and proposes future work.

Chapter 2

Background

2.1 FPGA Architecture

The reconfigurable architecture of an FPGA contains LCs as well as commonly used hardened elements for arithmetic and memory. LCs are the most common element in an FPGA and are usually arranged in an array-like structure as seen in Figure 2.1. Also seen in Figure 2.1 are hardened arithmetic, digital signal processing (DSP) blocks, and memory, block-RAM (BRAM) blocks, which are usually arranged in columns every few LCs. To communicate between blocks, configurable interconnect made up of various length wires and programmable multiplexers are used. Some commercial interconnect architectures include registered routing, a resource which we exploit when designing our LID system. Another important part of the FPGA architecture that we discuss in this section is the clock network, which distributes clock signals to clocked elements (registers) in regions of the chip or globally throughout the chip.

2.1.1 Logic, Memory and Routing

Basic Logic Elements (BLEs), shown in Figure 2.2, make up the primary building blocks of FPGAs [11]. The most common BLEs contain a k-input look-up table (LUT) and a flip-flop (FF) and may also contain an adder. In the BLE, the FF can be reset using synchronous and asynchronous clear signals and a multiplexer in the BLE is configured to pick either the registered or unregistered output of the LUT. Many BLEs are grouped together to form a logic cluster [11]. Within the cluster inter-cluster wiring and multiplexer choices are made to connect the inputs and outputs of the cluster to any or all BLE inputs. Logic clusters in the FPGA make up the *soft logic* of the FPGA, which is capable of implementing any sequential or combinational circuit logic. BLEs and logic clusters of modern FPGAs are slightly more complex: they include fracturable LUTs [51, 45], chain adders [53] and can even be used to implement shallow memory [46].

LUTs can implement a small ROM, and hence the truth table of any function, with no more inputs than the LUT provides. With a small amount of extra circuitry, LUTs can also implement small read/ write RAMs, called LUT RAM. To implement shallow memory the LUT configuration bits provide data memory storage. Small modifications made to the LUT architecture enable dynamic write capabilities, while input multiplexers from the logic cluster are reused as read addresses [46]. For example, in the commercial Stratix IV FPGA architecture each LUT contains 64 configuration bits which are used to

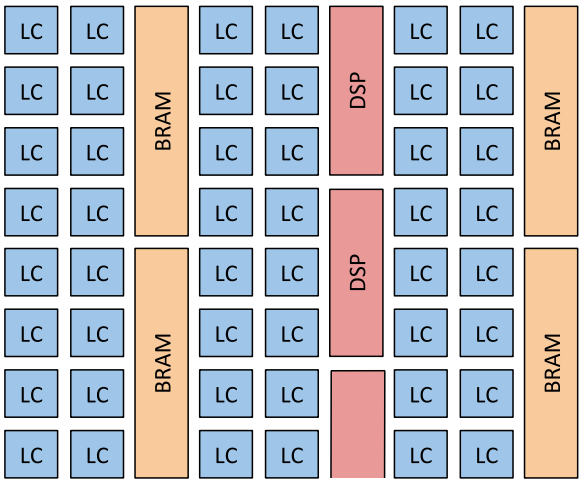


Figure 2.1: A high-level schematic of the FPGA layout. Hardened arithmetic DSP blocks and memory BRAM blocks are distributed amongst the LCs in columns.

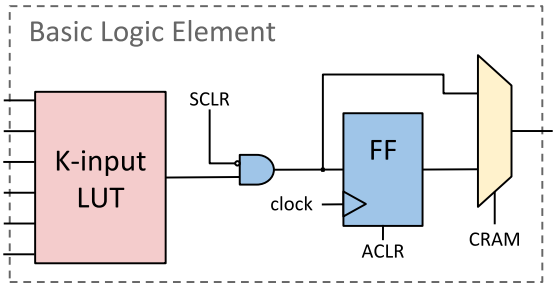


Figure 2.2: Basic Logic Element (BLE) containing a k-input LUT and a flip-flop (FF). The FF receives synchronous clear (SCLR) and asynchronous clear (ACLR) reset control signals. A multiplexer picks between the registered or unregistered output of the LUT using the configuration-RAM (CRAM) cells.

implement 64 word deep by one bit wide (64x1) RAM or a 32x2 RAM. There are 10 LUTs within a Stratix IV cluster enabling the cluster to implement 64x10 or 32x20 RAM [46]. Hardened memory blocks called BRAM can implement much deeper RAMs. For example, the Stratix V FPGA supports 512x40 BRAM which can be reconfigured to support additional word depth at the expense of smaller data-width [48].

Hardened blocks and soft fabric signals are routed throughout the FPGA using an interconnect structure similar to that shown in Figure 2.3 of a traditional island style FPGA [11]. Horizontal and vertical routing channels that contain various length wire segments make up the FPGA wiring. Switch blocks (SB) located on the intersection of vertical and horizontal channels contain programmable routing switches that allow for wire segments to form longer routes. SB patterns can be anywhere from random patterns or specific patterns like the Wilton SB pattern found in [58]. In most modern FPGAs, SB are made of pass-gate multiplexers and depending on the pattern and the number of switches needed, trade-offs between area and routing flexibility are possible [47, 11]. To wire signals from soft or hardened logic pins onto the FPGA interconnect, a connection block (CB) is used. The connection block consists of programmable switches that connect block I/O pins to the adjacent routing channels. The number of routing wires in a routing channel is called the channel width and is denoted by the symbol W, and the number of connections from a block pin to the routing channel is called the connection block flexibility

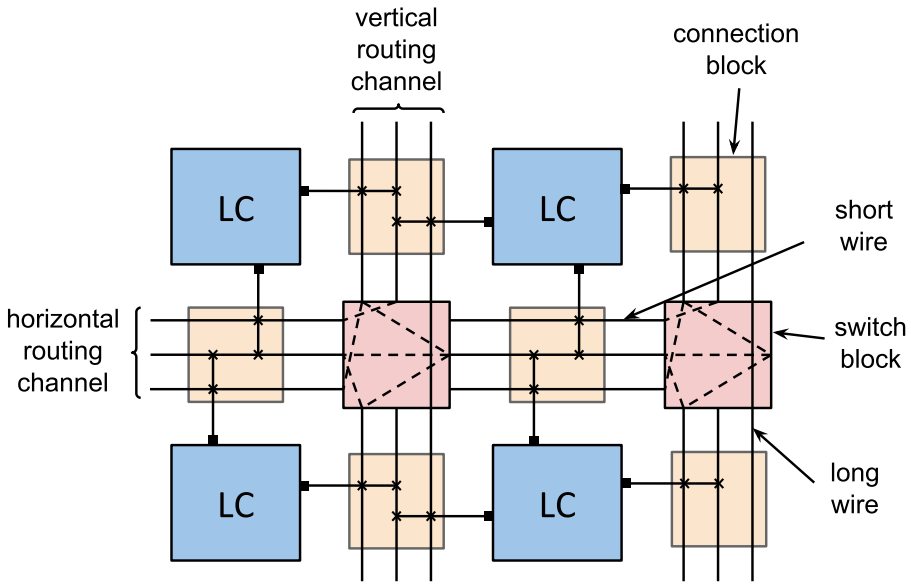


Figure 2.3: FPGA routing architecture for an island-style FPGA. Figure adapted from [11].

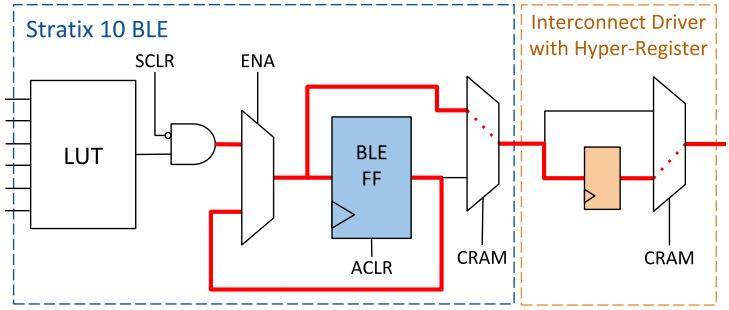


Figure 2.4: Stratix 10 BLE control signals [49]; highlighted path (red) shows the use of register control signals with a hyper-register.

and is denoted by the symbol F_c [72, 11]. Depending on the F_c value another trade-off between area and routing flexibility is possible [44].

2.1.2 Registered Routing in Stratix 10

The routing architecture of the recently released Stratix 10 FPGA includes registered routing providing an important FPGA resource which we utilize in Chapter 3 where we evaluate different latency insensitive design styles that better target new FPGAs.

The Stratix 10 pipelined architecture was introduced to mitigate the impact of long range routing delay (we discuss this problem in Section 2.3) and to be well suited for high-speed pipelined designs. This was done by adding a bypassable hyper-register to every routing multiplexer in the FPGA [49]. Hyper-registers enable the migration of a register in a design to anywhere on the interconnect, allowing for greater flexibility in pipelining long connections.

A hyper-register is implemented as a pulse latch, which logically acts as an edge triggered flip-flop (FF). Unlike a conventional FF, it has no direct access to control signals like clock enable and clear ports

[49]. Therefore, hyper-registers are most easily used as simple pipeline registers in a design that does not need these control signals. Registers in a design that use clock enable (ENA) or synchronous clear (SCLR) can still benefit from hyper-registers, as shown in Fig. 2.4. A BLE register is used to implement the ENA and/or SCLR functionality, and a simple register duplicate is implemented as a hyper-register by appropriate configuration-RAM (CRAM) cell settings. This can improve timing for registers with ENA or SCLR, but cannot reduce area by saving BLE registers. As for the asynchronous-clear (ACLR) port, it cannot be used together with the hyper-register as seen in Fig. 2.4.

Hyper-registers are targeted during the retiming stage of the Stratix 10 FPGA CAD flow. Retiming is the method used by the CAD tool to maximize the frequency of the design by moving registers to timing critical locations. The retiming stage is made more flexible due to hyper-registers since they can be enabled at the last stage of the CAD flow. This is when final timing is known; and hyper-registers can be configured or bypassed using CRAM cells without the need for change in placement that may require legalization. To get the most of the retiming engine it is recommended to minimize the use of register control signals [49]. Although, an example of a valid approach of retiming with ENA and SCLR signals was shown in Fig. 2.4, it is not desirable to route-through the BLE as there is a delay associated with this entry and exit. In addition, hyper-registers are not limited to LAB intra-block wiring, but instead bypassable hyper-registers are present on all routing multiplexers on horizontal and vertical wires including BRAM and DSP block input ports. Therefore, it is easier to target them anywhere on the chip if they are not tied to ALM control signals.

2.1.3 Clock Architecture

Unlike the general-purpose inter-block routing architectures described above, dedicated interconnect routing architectures, *clock networks*, are built specifically to route clock signals. Clock signals play an important role in synchronous systems by supplying a common time frame to logic elements throughout the chip. Any difference in the time at which one element is clocked compared to another contributes to clock skew which directly affects the frequency at which computations can occur. If large enough, clock skew can also lead to hold violations which prevent the correct operation of a design at any speed. To maximize system performance, clock networks are carefully designed to operate at high speed and with low-skew [81].

Poor wire scaling compared to logic density has made it increasingly difficult to design clock networks to operate at higher speeds. Clock networks must now be able to reach a larger number of logic elements even though wire speed has not improved [28]. Making high speed clock networks still more difficult to design is the fact that clock networks should include some programmability so that they can adapt to the clocking needs of different applications. The programmability is achieved by including a large number of small networks or a small number of large networks. Recent FPGAs such as Stratix 10 and Ultrascale include more flexible (programmable) clock networks than earlier FPGAs to better match application needs [24, 60].

Typical clock architectures on FPGAs are made up of multiple local and global clock networks [6]. Global clock networks span the entire chip and are sourced from the center of the device. Local networks are smaller – a common type of local network spanning a quarter (or quadrant) region of the FPGA. They are consequently sourced from the center of the quadrant clock region and reach only to logic elements in that quadrant. On the other hand, in the latest FPGA architectures, clock network regions are not as rigid. Instead of typical clock regions that span quadrants of the FPGA, the clock regions Stratix 10

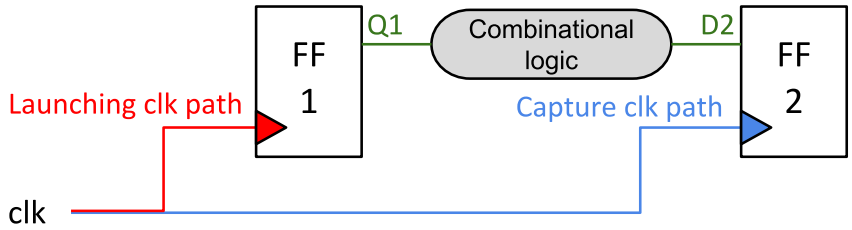


Figure 2.5: A timing path between launching FF-1 and capture FF-2. Figure adapted from [81]

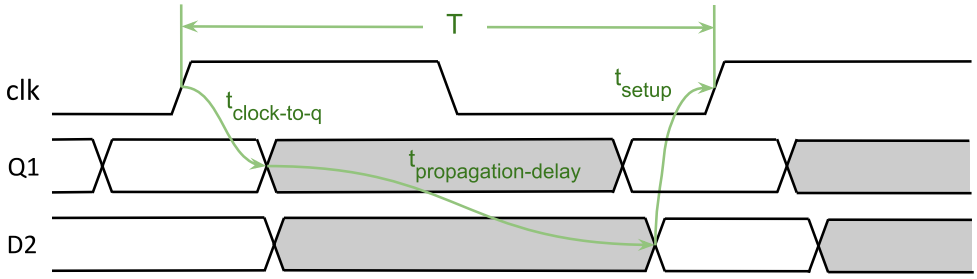
and Ultrascale FPGAs are divided into small tiles from which the clock can be sourced. The tiles are combined together using a routable clock distribution network to customize clock region size [29, 85]. This means that smaller clock networks will be used more often; thereby, reducing the use of large global clock networks which accumulate more skew and therefore slow down operational frequency [24, 60]. Another commercial FPGA clock architecture, Xilinx's UltraScale+, improves timing by introducing intentional delays, using programmable hardware delay chains, in clock network wires to enable time borrowing (beneficial skew) between slow and fast datapaths [27].

The programmable and routable clock architecture in modern FPGAs allows modules running on different clock domains to have much more flexibility in terms of where they can be placed on the chip, allowing for easier integration. This is important since FPGAs need to be flexible enough to support designs with multiple clock domains, especially at IO interfaces. Support for many clocks and flexible clock networks comes with a direct trade-off as it increases the size of the clock network layout area. Additionally, another trade-off for these large clock networks is the power they consume. In fact, due to the clock network's constant switching activity, it is one of the biggest contributers to dynamic power-consumption on an FPGA [31, 76]. The dynamic power-consumption of the clock network can be reduced using clock gating whereby parts of the clock network are disabled in clock cycles if the registers they feed do not need to toggle. Clock gating has the benefit of reducing the switching activity of the design which correlates to a reduction in dynamic power [31, 80]. On the other hand, adding clock gates in the clocking architecture can increase the area of the clock network. Given the complexity of all these interacting clock network design choices, it is important that there is a framework in which different clock network design trade-offs can be explored, and hence one of the contributions of this thesis is the creation of such a framework.

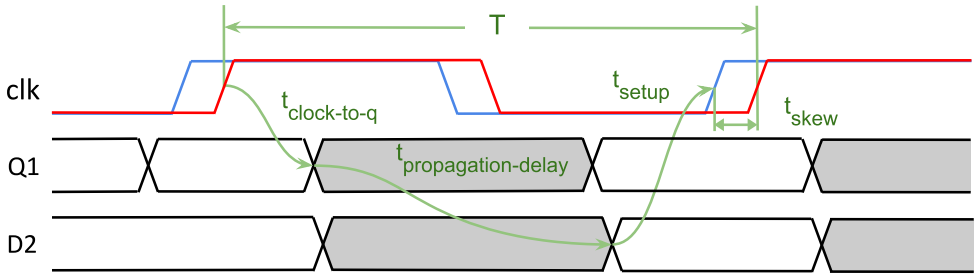
In the rest of this section we describe clock timing constraints and elaborate on the the design trade-offs and components of a clock network, we give examples of commercial FPGA clock networks and finally we also discuss prior work and methods that have been used to evaluate and model clock network architectures.

Timing Constraints

Timing constraints help us understand how clock skew affects the operating frequency of a design by cutting into the time that is available for computation. When designing a sequential circuit, the designer sets a timing constraint on the clock with the desired operating frequency. A timing analyzer then checks that all timing paths meet their timing requirements. A timing path begins at the output of one clocked element (a register) and may pass through an arbitrary amount of combinational logic until the signal reaches another clocked element that terminates the path as seen in Figure 2.5 [81].



(a) No clock skew: Clock arrives at both registers at the same time



(b) With clock skew: Clock arrives late to register-1 (red) and arrives early at the register-2 (blue)

Figure 2.6: Max delay timing constraints. Figures adapted from [81]

There are two timing requirements on a timing path. The first is the *setup requirement*: the time for which a signal must have settled before the rising edge of the clock. Figure 2.6a shows setup time as it relates to the clock period of the design [81]. It is described as follows:

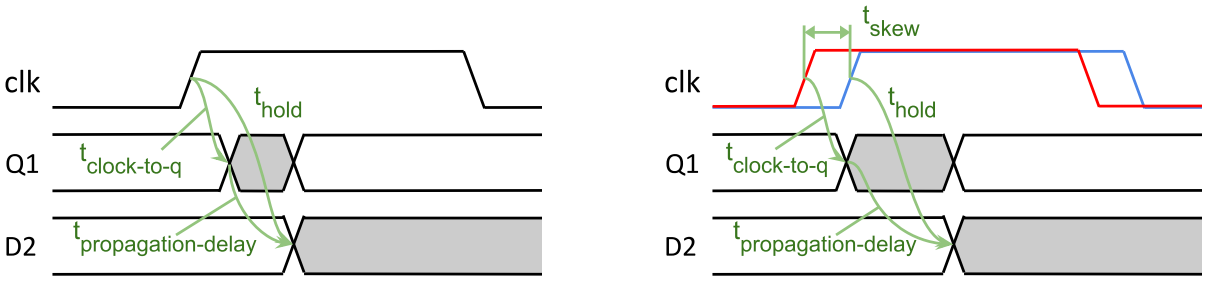
$$T \geqslant t_{clock-to-q} + t_{propagation-delay} + t_{setup} \tag{2.1}$$

where T is the design clock period, $t_{clock-to-q}$ is the delay from the clock edge until data appears on the output $Q1$ of the first register, $t_{propagation-delay}$ is the propagation delay of the data signal to travel from one register through some combinational logic to the next register, and t_{setup} is the time for which data must remain stable at input $D2$ of the second register to guarantee correct operation. The setup time constraint is also called the max delay constraint since it is the maximum possible propagation delay allowed for the signal to travel from one register to the next. The timing diagram in Figure 2.6a assumes that the clock arrives at both registers at the same time. However, this is usually not the case: Figure 2.6b shows the worst case scenario for max delay where the launching register-1 receives its clock late while the capture register-2 receives its clock early. An imbalance of clock arrival time at different registers are called skew. We account for clock skew in Equation 2.1 and 2.3 as follows:

$$T \geqslant t_{clock-to-q} + t_{propagation-delay} + t_{setup} + t_{skew,src-dst} \tag{2.2}$$

$$t_{skew,src-dst} = t_{clk,src} - t_{clk,dst} \tag{2.3}$$

where $t_{skew,src-dst}$ is the difference in arrival time of the rising clock edge from the source (src) register Q1 to the destination (dst) register D2 in Figure 2.6b. In this case, skew subtracts from the time available for work, i.e. propagation delay, and forces the designer to either reduce their combinational



(a) No clock skew: Clock arrives at both registers at the same time

(b) With clock skew: Clock arrives early to register-1 (red) and arrives late to register-2 (blue)

Figure 2.7: Min delay timing constraints. Figures adapted from [81]

logic on timing critical paths or slow down their entire system.

The second timing constraint relates to hold time: the time for which a signal must remain stable after the rising edge of the clock. The hold time constraint can be seen in Figure 2.7a and is described as follows:

$$t_{hold} \leqslant t_{clock-to-q} + t_{propagation-delay} \tag{2.4}$$

where t_{hold} is the hold time required for the data signal to remain stable at input $D2$ after the rising clock edge to be ensure correct operation. The constraint in Equation 2.4 is also called the min delay constraint since its the minimum possible propagation delay needed for the signal to travel through one resister to the next. In ideal conditions hold time (t_{hold}) is small enough that a designer will not come across a hold time failure. However, in the presence of skew, hold time can effectively increase. Figure 2.7b shows the worst case scenario for min delay where the launching register-1 receives its clock early and the capture register receives its clock late. The min delay constraint when accounting for clock skew can be described in Equation 2.5 and 2.6 as follows:

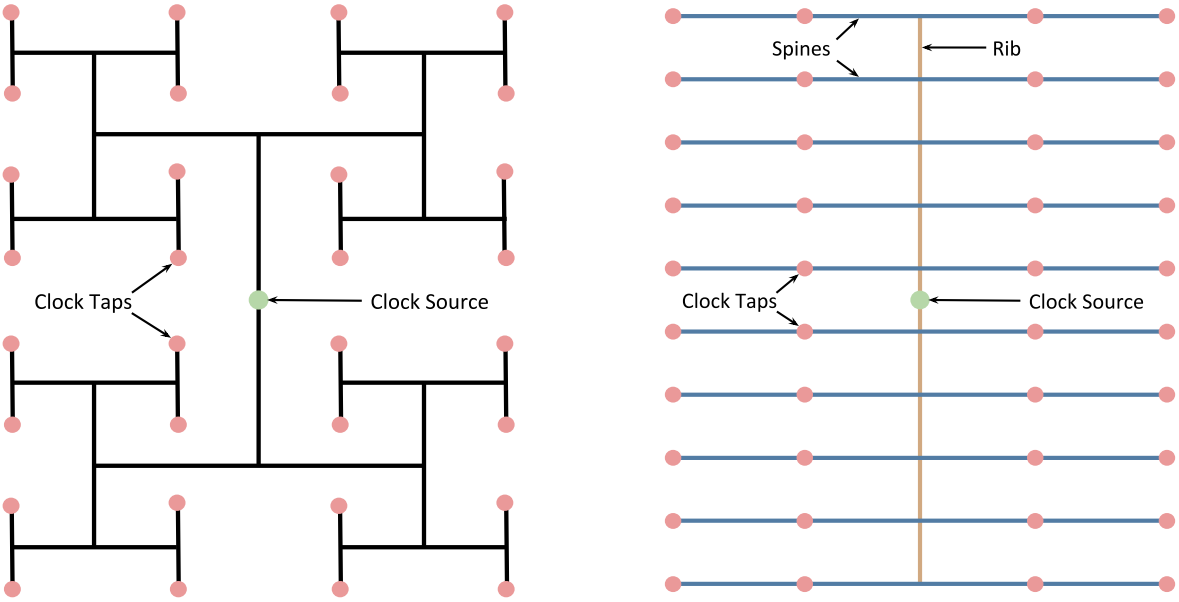
$$t_{hold} + t_{skew,dst-src} \leqslant t_{clock-to-q} + t_{propagation-delay} \tag{2.5}$$

$$t_{skew,dst-src} = t_{clk,dst} - t_{clk,src} \tag{2.6}$$

where $t_{skew,dst-src}$ is the difference in clock arrival time from the destination register D2 to the source register Q1 in Figure 2.6b. Equation 2.5 shows us how skew adds to hold time making for a tighter timing constraint. If the min-delay constraint is not met, no change in clock period can fix the design, making it a particularly problematic failure. The FPGA designer's only option is to increase combinational logic or routing delay between timing critical registers to meet the hold-time constraint. To make this inefficient solution less necessary the clock network should be carefully designed to minimize skew making both min and max delay failures less likely.

Design Constraints

In addition to clock skew [24, 27], other important design considerations when designing a clock network include: clock insertion delay (clock latency), clock network power consumption [31, 80], network area, and flexibility/ adaptability of the network [41, 24].



(a) H-tree

(b) Rib and Spine

Figure 2.8: Fundamental clock network topologies.

Clock skew: We have seen how *clock skew* can cause setup and hold time violations. There are four sources of clock skew: systematic, random, drift and jitter [81]. *Systematic skew* occurs if the length of the clock distribution wires from the clock source to clocked elements is not equal, or if the loads at the end of distribution wires are not balanced. Take the fundamental clock network typologies in Figure 2.8 as an example: Figure 2.8b shows a Rib and Spine network; a trace from the clock network source to the taps does not always traverse an equal length of wire segments. The unbalanced delay from the clock source to different tap locations leads to different clock arrival times and contributes to systematic skew. Even though rib and spine clock networks introduce systematic skew, they are frequently seen in FPGA clock networks as their regular design can be more simply incorporated in the gird-like layout of FPGAs [6, 83]. On the other hand, Figure 2.8a shows an H-tree clock network; a trace from the clock source to any of the clock taps/ clocked elements yields an equal length of wiring traversed. If all the clock taps have equal or symmetric loads the system effectively has zero systematic skew. Due to the re-configurable nature of FPGAs, balanced load capacitance on clock network taps is not a gurantee and is up to FPGA placement and packing algorithms to balance loads [80, 42, 88, 60].

Whereas systematic skew happens under ideal conditions and is easily measurable, *Random skew* is different from one chip to the next making it harder to account for. Uncertainty during the chip manufacturing step causes on-die variation that leads to skew. Variations include: differences in wire width and spacing which directly effect the resistance and capacitance of the wire, and differences in transistor characteristics which effect the delay and current through the transistor [24]. These changes, though different from chip to chip, could in principle be measured during speed testing at manufacturing time. It would then be possible to account for this and other sources of skew in an FPGA using programmable delays in combination with buffers along or at the end of clock network distribution wires [81, 27]. The skew component from *Drift* is any time-dependent change in the device. Skew from drift is harder to account for and any measurements of it to enable a programmable delay compensation would

have to be repeated periodically. The most prevalent causes of drift skew are temperature variation caused by hot spots in the device, and aging. Finally, skew from *Jitter* varies by cycle and comes from noise in the system. Primary sources of noise can come from voltage variation on the power supply and crosstalk between signals.

Clock Insertion Delay: or clock latency is the delay for the clock signal to travel from the clock drive point to the clock sink. Minimizing the insertion delay tends to help minimize random skew, as the clock network has a smaller delay that is subject to random variation. Unfortunately, insertion delay has only increased with process scaling, because wire speed for the same length of wire has not scaled while chips are getting larger [24].

Power Consumption: There are two sources power consumption on FPGAs: static power and dynamic power. Static power is consumed in the absence of circuit switching activity. In SRAM-based FPGAs static power is due to leakage current [80]. Sources of leakage current are: transistor sub-threshold conduction, non-ideal gate dielectric, and reverse-biased p-n junction current [81]. Dynamic power on the other hand is directly correlated to circuit switching activity. The average dynamic power consumption across all nets in a design is described as follows in Equation 2.7 [31]:

$$P_{avg} = \frac{1}{2} \sum_{i=1}^{n} f_i \times C_i \times V^2 \tag{2.7}$$

Where $i = 1$ to n is the sum over the total number of nets in the design, f_i is the average toggle rate on net i, C_i is the capacitance, and V is the power supply voltage. To compute clock network power we are only interested in clock nets. The work in [76] showed that on a complete design 22% of dynamic power came from the clock network. Techniques to reduce dynamic power have looked at factoring in clock power during placement to reduce clock spine usage, as well as clock gating [31, 80]. Clock gating involves selectively moving register enable signals to instead disable buffers along the clock network. This will dynamically turn off the clock, thereby reducing the average switching frequency of the clock net.

Area: Clock network area comes from wires and clock buffers. Wire delay increases quadratically with length; therefore, buffers are used to break-up long wires into smaller segments and achieve a linear increase in delay with wire length. Different types of buffers are possible: for example, to allow clock gating [24, 60], to increase flexiblity using routable clocks [31, 80], and to adjust skew [27]. Depending on the types of buffers used, a direct trade-off in area, delay and power can be made. The number of clock networks available on the device can also significantly increase the clock network area. For example, the Stratix IV FPGA allows up to 71 clock signals per chip quadrant [7].

Flexibility: The number of clock domains needed on a chip has increased as a result of increased logic capacity and an increased number of I/O interfaces available. Early FPGAs contained a modest number of fixed chip wide global networks that could be used to synthesize many clocks. However, global clocks quickly become inefficient due to large size and higher insertion delay, which accumulates significant skew. Later FPGAs contained both chip-wide clocks and also a number of smaller quadrant clocks. This enabled the use of smaller localized clocks. On the other hand, modern FPGAs are considerably more flexible by allowing custom sized clocks to be built anywhere on the chip through the use of routable clock network architectures [85, 24].

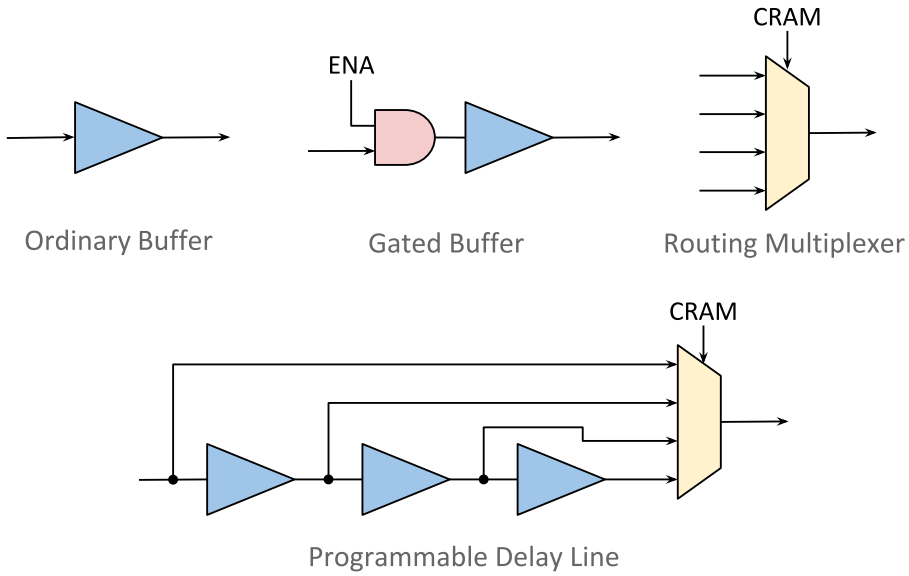


Figure 2.9: Types of clock buffers and routing resources

Design Components

To build fast and reliable clock networks a balance between flexibility, area, and power as well as the other design constraints discussed above must be made. At a high level there are three main components that make up a clock architecture on a chip: high-speed I/Os that link external clocks to the FPGA, internal clock generation units like phase-locked loops (PLL) that match the frequency and phase of the internal clocks relative to external clocks, and clock distribution networks that supply the internal clocks to clocked elements [81]. Out of the three components, the distribution network is most unique in FPGAs versus ASICs.

Clock distribution networks can be described in three stages [41]:

1. From the I/Os, PLLs, and other clock generators to clock sources: Clock sources are the root of the clock trees and are available at the center of clock regions. Traditionally global clock regions occupy the full chip and local regions are available in each quadrant. In modern FPGAs clock regions are much smaller; for example, the largest Stratix 10 device contains 80 clock regions arranged in a grid pattern [37, 29].

2. From the clock sources to logic clusters: Fixed clock networks or custom routed clock networks are distributed to tap points throughout the chip.

3. From logic clusters to individual registers (BLEs): To balance flexibility and area, BLEs can connect to all or some clock networks available at the logic cluster inputs.

The second and third stage clock distribution networks require the use of clock buffers along wire segments. Figure 2.9 shows different types of buffers that are used in clock networks: regular buffers can only improve the delay of wires, gated buffers are used to turn off portions of the clock, full routing multiplexers are used to create programmable custom clock networks, and programmable delay lines can be used to adjust the clock delay and hence skew.

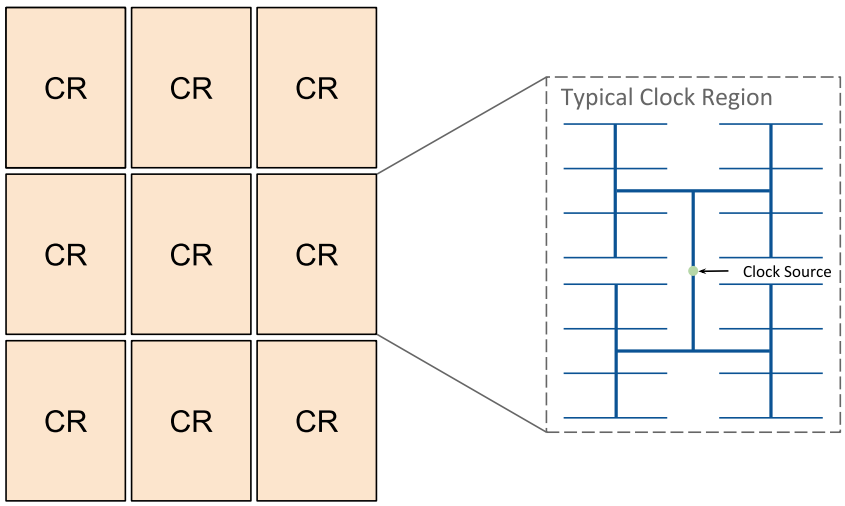


Figure 2.10: High level view of clock region layout on FPGAs. Zoomed in is a typical clock region distribution network.

Commercial FPGA Clock Networks

The clock architecture in commercial FPGAs is separated into clock regions (CRs), as seen in Figure 2.10. Each clock region is sourced from its center and is distributed on fixed clocks to logic clusters. Distribution networks are often built out of rib and spine topologies as seen in Figure 2.8b, for example in Virtex II FPGAs [83], or are a mix between H-trees and rib and spine topologies, as seen in Figure 2.10, like in Stratix V devices [8]. In typical FPGAs, each clock source is driven by dedicated routes from high speed I/Os, PLLs, as well as general routing and can generate local clocks.

Global clocks are also available on FPGAs. They span the entire chip instead of being confined to clock regions. Typically, global clocks are distributed separately than local clocks. However, hierarchical clock networks like those in Stratix IV and Stratix V devices allow for sharing of resources between the global and local clocks [7, 8]. In a hierarchical network, a mux exists at the clock source of each clock region that allows driving the clock from either the global clock, or other clock generators.

In older FPGAs, like Stratix V or Virtex 5, there were four to six clock regions per device respectively [8, 80]. By contrast, modern FPGAs like Stratix 10 has up to 80 clock regions available spanning 25K LC each [24]. A local clock network will often connect together more than the 25K logic clusters available in a clock region. This is acceptable because clock regions in Stratix 10 can be connected together through a routable clock distribution network. Conversely, in historical FPGAs if a clock network needs to connect together more than the logic available in a clock region it would end up using global clock networks with large insertion delay, which will contribute more skew from drift, jitter and on-die variation.

A high level diagram of a Stratix 10 clock region can be seen in Figure 2.11. A rib and spine network is used to distribute the clock sourced from the center to logic clusters. At the border of the clock region is a routable clock distribution network. The routable distribution network is made out of 32 horizontal and vertical bidirectional wires with programmable switches at each edge of the clock region. Through a tap point all vertical distribution wires are able to drive the clock spine. The routable distribution wires contain parallel networks: one network is used for stage one distribution from clock generators to the center of the custom clock network, and the second network is for stage two distribution which routes custom built H-trees to clock tap points in each clock region. The Ultrascale routable clocking

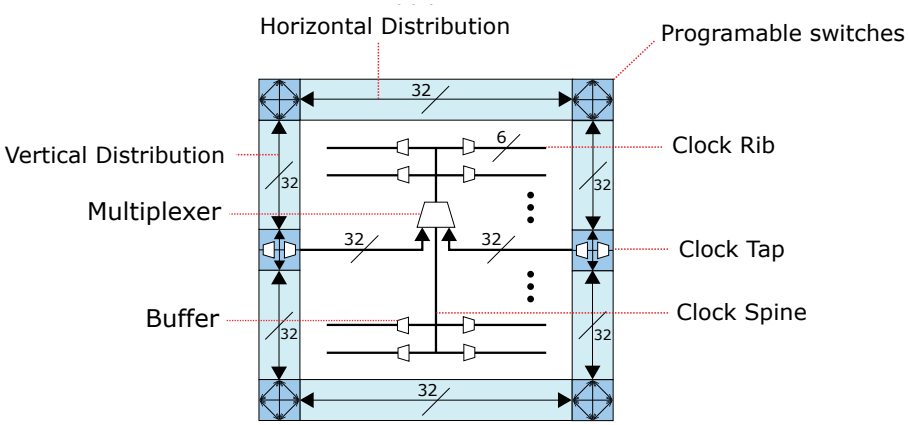


Figure 2.11: Stratix 10 clock region. Figure taken from [33]

architecture is also similar [60, 85]

In addition to the choice of distribution networks, commercial architectures also include different types of clock buffers chosen to improve power and/or speed. For example, the Virtex-5 FPGA uses gatable buffers to improve dynamic power [80]. A type of gated buffer using an AND gate can be seen in Figure 2.9. Unlike ordinary clock buffers, the output of gated buffers can be turned off using an enable signal which enables clock gating. Clock gating aims to reduce dynamic power consumption by disabling portions of the clock feeding a set of registers at times when the register outputs are inconsequential to the application result [31, 80]. Functionally the circuit with or without clock gating remains the same; however, physically the switching frequency of the parts of the clock network decreases, thus reducing dynamic power as shown in Equation 2.7. [31] has shown that the use of clock gating at a fine granularity can improve clock dynamic power consumption by 50%

Another replacement to ordinary buffers is programable delay lines used to improve design speed. Programable delay lines, like the one seen in Figure 2.9, are made out of a chain of buffers and a mux. The mux selects from one the outputs to the buffer chain. Each additional buffer adds to the insertion delay of the clock, allowing the ability to selectively delay the clock at fixed increments. To see how programable delay lines can improve design speed, we use the timing diagram in Figure 2.6b as an example. In this scenario skew is caused because the clock arrives late to register-1 and early to register-2. By inserting a programmable delay in the capture clock path of register-2 we have reduced skew and loosened the setup time constraint. Note however that because we delayed the clock to register-2 we will have changed skew of the paths going out of register-2. This essentially borrows timing slack from one path and distributes it to the other path and is the reason this technique of improving speed is called time-borrowing [81]. So long as there was enough timing slack available in the connection from register-2 to subsequent registers, this would be beneficial in the example of Figure 2.6b. Devices such as the Ultrascale+ FPGAs use time borrowing at a granularity of 16 time borrowing circuits per clock region. It has been shown to improve design frequency by 5.5% on average while only using 0.1% area overhead to add the programmable delays [27].

Clock Modeling Prior Work

To study the how different clock architectures change flexibility, power, and speed, clock network models are created. Depending on the model it can evaluate all or some of the clock architecture design constraints.

In [31] the authors studied the advantages of clock gating. They generated clock power models for StratixIII-like and Virtix-5-like FPGA clock architectures and varied clock gating at two different levels of granularity. The first level gates an entire clock region, and the second level is a much finer granularity which gates individual ribs and spines within a clock region. They ran their experimentation using VPR, an open source CAD software tool [11]. They modified VPR's placement engine to reduce clock spine usage, and estimated the resulting dynamic power, including clock gating effects, for several benchmarks. They did not actually route the clock network, which means they cannot change the clock network to more complex topologies, nor can they evaluate the clock network area or delay.

In [41] the authors created a clock architecture model that evaluates the area and power of a variety of rib-and-spine topology clock networks. They allowed flexibility in changing the number of clock regions, the number of instances of clock networks, the number of spines from which each rib could select a clock signal, and how logic clusters select from available clock instances. While their models compute the implementation cost (area and power) of the various networks, they do not develop CAD tools which can implement circuits using the different networks, so they cannot draw conclusions on the utility of different clocking architectures from an applications perspective.

CAD tools provide a method to complete design implementations and directly extract their quality; a flexible CAD tool that can map to a variety of architectures allows direct comparison of the quality of those architectures over a set of benchmarks [11]. Ideally, a CAD tool would take as an input a clock network architecture model and output area, power, and speed results based on a set of benchmarks. Overall, we have noticed that there is a gap: there is no available flexible CAD system can that model both a variety of clock architectures and all their design constraints.

2.2 FPGA CAD

The mapping of an application onto an FPGA architecture requires automation through the use of computer aided design (CAD) tools. Choices made in the CAD flow are important and directly affect the final area, power, and operating speed of the application. The typical FPGA CAD flow consists three main phases: elaboration, logic synthesis, and physical synthesis [73].

Elaboration

The elaboration step takes an input a definition of an application, written in a hardware description language (HDL) like verilog, and translates it into a netlist. The netlist consists of basic digital elements like logic gates, multipliers, registers, and memory, as well as the signals connecting the elements together [38, 82].

Logic Synthesis

There are two steps to logic synthesis: technology independent optimizations and technology mapping [9]. Technology independent optimizations involve reducing logic in the design through logic minimization

which can improve design speed, area, and power. Technology mapping involves mapping of technology independent logic to the basic block primitives present in an FPGA, such as K-input-LUTs, multipliers, memory blocks, and registers.

Physical Synthesis

Physical synthesis is usually divided into three steps: packing, placement, and routing [52]. Packing groups together logic into larger blocks to simplify the placement problem. For example, it groups LUTs and registers into logic clusters, small RAM primitives into larger RAM blocks that exist on the FPGA, multiplication and (in some architectures) other arithmetic primitives into DSP blocks. Placement determines the location of blocks (LCs, BRAMs, and DSPs) on the FPGA. It aims to minimize the wire-length between connected blocks to optimize for delay, area, and power. Routing determines the wires and switches on the FPGA routing network that signals will use to implement all the connections required between the placed blocks.

2.2.1 Analysis and Verification

After running through the CAD flow, static timing analysis tools verify that the design has met its timing constraints [63]. If the design has not met its timing constraints the designer has to modify the design and/or CAD tool settings, and re-implement the modified design through the CAD flow. For large designs many modify/re-implement iterations are often required, and this is costly especially since these large designs present a large optimization problem that leads to long run-times. As FPGA capacity has grown with technology scaling, ever larger designs can fit on a single FPGAs, magnifying this design productivity problem.

2.3 FPGA Scaling Characteristics and Challenges

Field-Programmable Gate Arrays (FPGAs) continue to scale in capacity with the introduction of every new process generation. For example, the recently released Stratix 10 FPGA has as many as 5-million Logic Elements on a chip; this is 5x greater than the amount present in Intel's previous generation Arria 10 FPGA [37, 35]. While transistor density on a chip continues to increase, wire speed for the same length of wire is not scaling [28]. Thus, global connections that traverse a large area of the chip strongly impact the critical path of large designs. In fact, over five Stratix FPGA process generations, from 130 nm to 28 nm, interconnect that spans the length of the chip had no speed gain [61]. The same trend continues with the 14 nm Stratix 10, which like the previous generation FPGA achieves ∼100 MHz register to register operating frequency for connections spanning the chip according to our experiments. This global wiring scaling problem exacerbates one of the major challenges facing hardware designers today, timing closure: ensuring the design meets all its timing requirements.

2.3.1 Timing Closure and IP reuse Problem

Currently, a designer using the common synchronous design methodology must perform many costly iterations of the entire FPGA CAD flow (synthesis, place, and route) to reach timing closure. First, the designer must manually add pipeline registers to critical paths of the design and update design logic to maintain functionality given these modifications. Then they must re-run the CAD flow and

verify that the design was able to meet timing. The designer has to iterate these two steps until the design has reached timing closure while remaining functionally correct. This is a time consuming task, since running through the CAD flow requires multi-hour-to-multi-day runtimes [62]. In addition, the entire problem is reintroduced if the designer chooses to move their design to a new FPGA chip, as a different FPGA architecture and process technology affects the timing of the design. Hence, even if the designer has validated the IPs in their design on the new FPGA, the designer must tediously fine tune the interconnect to meet timing again.

Recent demand for FPGAs in the datacenter have put pressure on designers to be more productive [19]. In the data center IP cores will be combined into accelerators in more diverse ways. This adds more motivation to move to a system level design methodology with forward compatibility that can reach timing closure with low effort.

2.4 System Level Design Approaches

There are many proposed system-level design approaches which we discuss in the following sections. One approach that is particularly favorable to solving the timing closure problem is Latency Insensitive Design (LID).

2.4.1 Latency Insensitive Design

LID is the process of assembling design modules using communication links that are allowed to transfer data with arbitrary latencies. Each design module's only guarantee is the synchronization on communication channels and order of the arriving and sent data.

The benefit of LID is that it allows the hardware engineer to design their system in two separate stages:

- *Computation:* The design of the IP cores' functionality.
- *Communication:* The interconnect architecture used for communication between computational cores.

The separation of the IP from interconnect allows the insertion of additional pipeline stages on latency insensitive links, at any stage in the CAD flow, without changing the functionality of the design [15]. Additionally, it makes it feasible to develop CAD tools that can automatically pipeline system-level interconnect later in the CAD flow, thereby reducing time-consuming iterations of the design. Finally, the simplified method of reaching timing closure that LID provides, improves forward compatibility (migration of systems to future FPGAs).

2.4.2 System Integration Tools

A system can be integrated at the HDL level but this is time consuming so many designers now use system integration tools like Quartus Platform Designer and Vivado IP Integrator [36, 86]. These tools help to abstract the design of the interconnect. However, unlike LID these tools require latency to be specified by the designer, leading to tedious exploration of latency options. For example, using the Altera Qsys system integration tool, it took multiple CAD runs using the max tool effort to achieve a

timing-closed placement for both small and large designs [1]. The academic tool in [70] is similar but supports a wider range of network topologies which can lead to area savings.

Some work to minimize the design effort for interconnect timing closure through automation has been done in [71]. It showed 43% area can be saved using a linear programming method to synchronize communication, as opposed to FIFO based synchronization. However, this work assumes fixed latency, and only applies to streaming communication. Other work to ease the burden on developers focuses on fine-grained interconnect [69] and not on the global wiring delay problem.

2.4.3 Globally Asynchronous Locally Synchronous Design

Many architectural solutions have been proposed for solving the global wiring problem. One solution is through Globally Asynchronous Locally Synchronous Architectures (GALS) [74, 39]. This FPGA architecture independently clocks global and local interconnect. It assumes the use of asynchronous communication for global signals, while maintaining synchronous communication locally. Global signals are those that span an area larger than the defined local regions. Unfortunately, this architecture requires radical clock network changes as well as major CAD tool changes.

2.4.4 Networks On Chip

Networks-on-Chip (NoC) is an approch that uses packet-switched routers within the FPGA for system level communication. The routers consist of arbiters, virtual channels, and credit-based FIFOs and are linked by high-speed dedicated wiring. They are introduced as a solution to global interconnect speed problems and are ideally used as a latency-insensitive system (through the use of lightweight LI wrappers with size less than 10 equivalent LABs). Embedded Hard NoCs show significant gains including reducing routing utilization by 40% and improving frequency up to 80% [2, 50]. They have been incorporated in some recent FPGAs like Xilinx Versal [77, 26] and Achronix Speedster7t [4].

FPGA-optimized soft NoCs such as hoplite [40], CONNECT [66], and split-merge [30] also present a latency insensitive interface. They are efficient when modules' communication patterns vary over time; however, they require a large area when used for fixed communication, and therefore small LIDs can be more efficient [87].

Chapter 3

Latency Insensitive Design

Long distance interconnect delays are not scaling well with process technology, thereby leading to long routes strongly impacting the critical path of large FPGA designs. This forces the designer to pipeline long connections, which necessitates time consuming logic redesign in traditional latency-sensitive systems.

One way to simplify timing closure and IP reuse for the system level, is for the designer to use a latency insensitive design (LID) methodology [15]. LID separates the design of the interconnect from the computational modules. By doing so it allows the designer to add as many pipeline stages in the interconnect as needed to meet the desired clock frequency without breaking the functionality of the design. Additionally, it makes it possible to develop CAD tools that can automatically pipeline the interconnect later in the CAD flow, thereby reducing time-consuming iterations of the design.

Generally LID has been based on a design style introduced by Carloni [16] which uses a structure called a Relay Station for pipelining. Work in [61] quantified the cost and benefits of using Carloni-based LID on FPGAs. It concluded that the area and timing overhead is manageable, given the benefits LID provides for connecting large designs in FPGAs. We note however that this design style was developed for ASICs and the cost of circuitry on FPGAs is significantly different. Registers and multiplexers on the FPGA are relatively expensive and block-RAMs are plentiful making solutions that make heavier use of FIFOs more attractive. Recently, there has been a major change in FPGA architecture with Stratix 10, which includes registers (hyper-registers) within the FPGA routing interconnect. The commercial introduction of hardened interconnect registers increases gains for deep pipelining by using the less costly interconnect registers for pipelining, making it a good fit for LID. However, hyper-registers do not have direct access to enable or clear ports, motivating new LID styles that can efficiently exploit such registers [49]. We develop two different LID styles that can easily target hyper-registers on communication links. We compare them to Carloni-based LID which requires more advanced pipelining structures with wide multiplexers and high-fanout enable signals to implement Relay Stations. Our comparison is performed on both Arria 10, a more traditional FPGA architecture, and Stratix 10, which includes hardened interconnect registers.

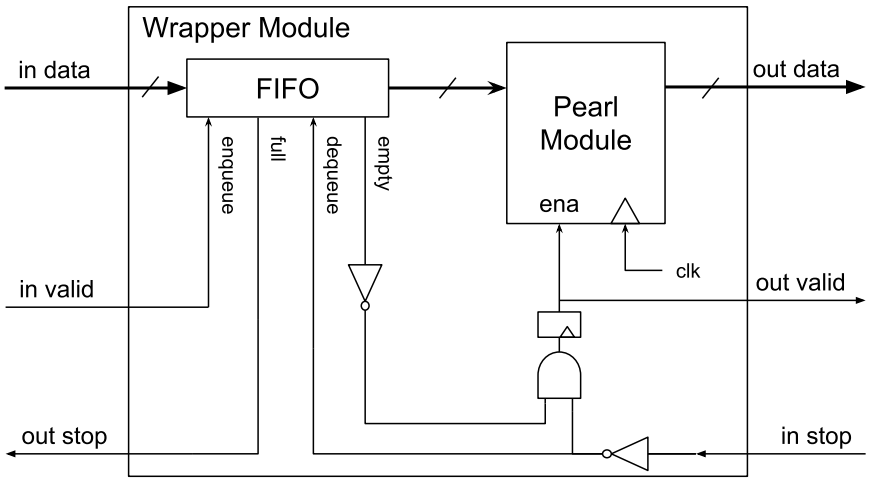


Figure 3.1: Carloni based LI shell encapsulation

3.1 Three Different Latency Insensitive Styles

LID separates the design of the IP module from the communication channels. The separation of design of the module functionality is done via shell encapsulation of the module (usually referred to as the pearl) using a latency insensitive wrapper. The wrapper can be automatically generated and is comprised of buffering stages and control logic used to synchronize data on communication channels as well as to control stalling of the pearl module using a backpressure signal. The only requirement of the pearl module is that it is stallable.

For a module to be stallable it must be able to freeze input and output data in the event of a stall. This is a simple modification and can be implemented by sending a clock enable signal to all flip-flops inside the pearl module as implemented in [61]. The wrappers in turn make the module *patient*. A patient module is one that behaves correctly regardless of the interconnect latency. The resulting design greatly simplifies the interconnect architecture allowing additional pipelining of communication wires between patient modules [15]. We study the use of three different approaches to LID that utilize different pipeline elements and wrappers. However, the abstraction all three LID approaches present is the same such that they are equally easy to integrate into designs.

3.1.1 Carloni-Based Latency Insensitive Design

Carloni and Vincentelli introduced a wrapper and pipelining element design for LID in [18]. Variations of their designs can be seen in [61, 14]. We adapted the original work with the addition of input port buffering queues for the wrapper data channels. Although the buffering queues are not necessary for the functionality of the system, they can optimize throughput in systems where wrappers have multiple input channels [17]. The wrapper design can be seen in Fig. 3.1, and the pipelining element, called a Relay Station, can be seen in Fig. 3.2 along with its control finite state machine (FSM) in Fig. 3.3.

The shell interface consist of data signals, a valid signal, and a backpressure stop signal for each channel of communication. The backpressure stop signal as well as the state of the FIFO determine when to enable the pearl module. If the pearl is not taking in valid data, i.e. the FIFO is empty or a

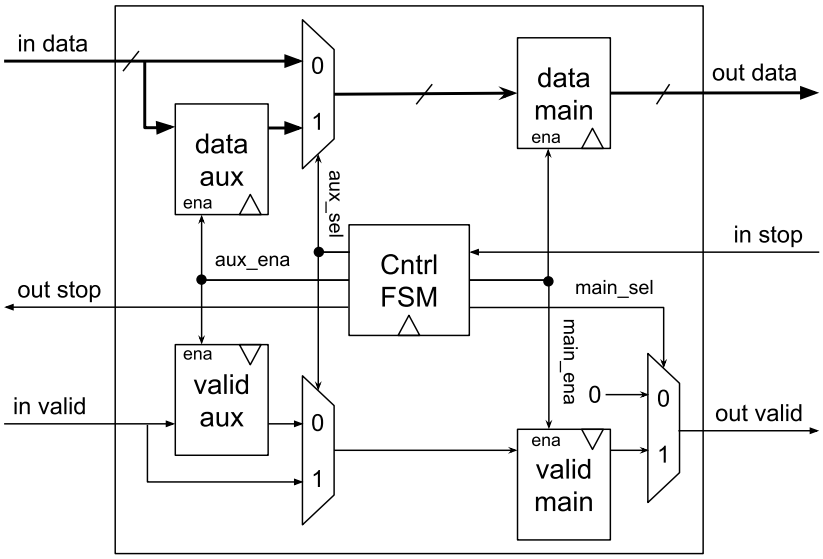


Figure 3.2: Relay Station Design used for Carloni LID pipelining

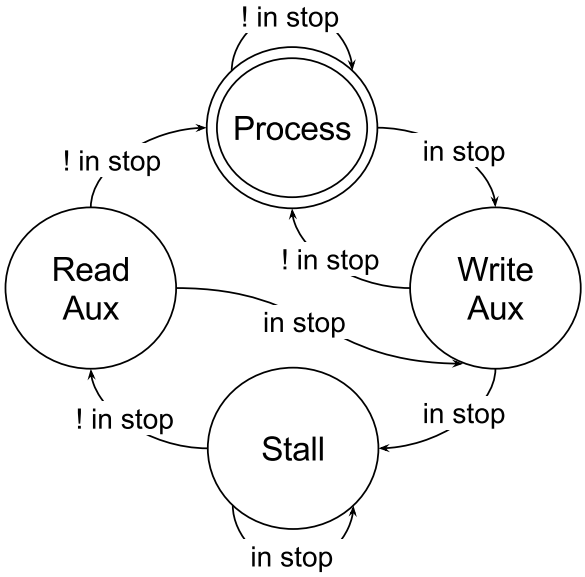


Figure 3.3: Relay Station FSM used for Carloni LID pipelining

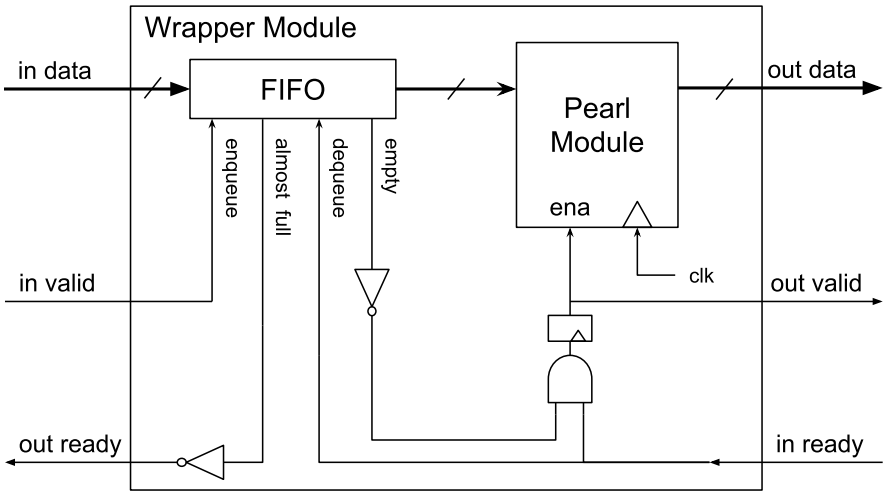


Figure 3.4: Ready-Valid based LI shell encapsulation

stop signal has been raised from upstream, the downstream wrapper will send out invalid data, which is referred to as a void token.

The Relay Station is also patient; it allows for the use of two word deep buffers for pipelining the communication between shells. In the event of a stall the current data is stored in the main register bank along with its valid bit and immediately the output data is voided using the multiplexer in front of the main valid register bank as seen in Fig. 3.2. Any incoming value is stored in the auxiliary register bank along with its valid bit. When exiting a stall the output value is read from the main register bank, then the next value read is from the auxiliary register bank provided there is no additional stall.

The progression of these events can be seen in the Relay Station control FSM in Fig. 3.3. There are four total states. The *Process* state allows data to go through the relay station. The *Stall* state stops data and transmits void packets instead. *Write-Aux* and *Read-Aux* states guide the transitions from Process and Stall by storing data in auxiliary registers in the event of a stall and reading data from auxiliary registers when exiting a stall. All registers in the design are controlled through clock enable signals, and selecting between auxiliary or main storage is controlled using multiplexer select signals seen in Fig. 3.2.

3.1.2 Ready-Valid Based Latency Insensitive Design

The ready-valid system is analogous to the request-acknowledge handshake signals that are adapted as an open standard in AXI4 and Avalon interconnect [32, 84]. The wrapper design can be seen in Fig. 3.4. This design uses FIFO queues to buffer data in the event of a stall. The FIFO size determines the number of pipeline registers. As many pipeline stages as half of the amount of words in the FIFO can be added to the interconnect. This means that unlike the Carloni-based LID system, the FIFO is necessary and must have a depth of at least 2x the number of pipeline stages between communicating channels for correct functionality – it must accommodate the n-cycle back-pressure latency plus the n-cycles of data in flight that cannot be paused. If the FIFO is empty, then the module is stalled and the wrapper outputs void data. On the other hand, if the FIFO is almost-full, with the value of almost-full set as the round trip latency of pipelined interconnect subtracted from the number of words in the FIFO, a

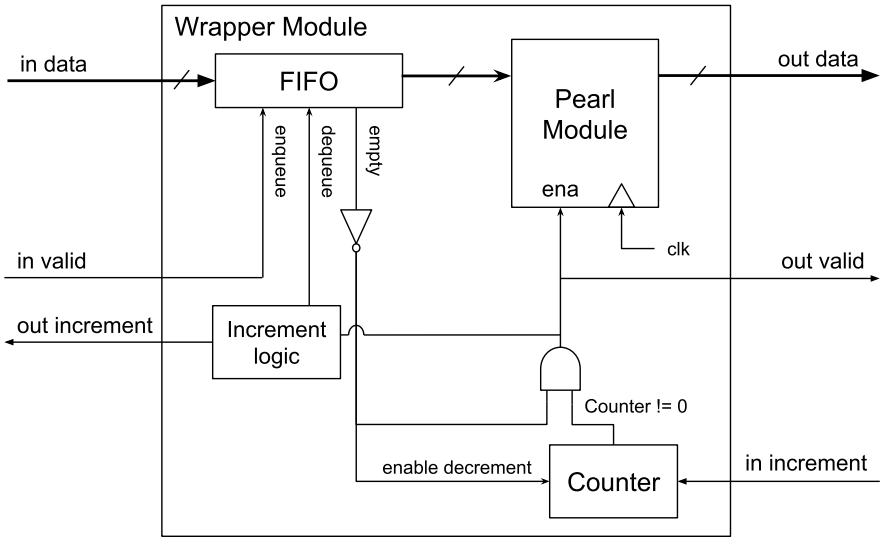


Figure 3.5: Credit based LI shell encapsulation

downstream active low ready signal is sent for back-pressure stalling.

One major advantage over the Carloni-based system is that we can use regular registers instead of relay stations which allows it to easily target the hyper-registers in Stratix 10. This is because the ready-valid FIFO is required to have enough space to store the contents of the pipeline in case of a stall without the need for distributed storage elements, such as relay stations, to perform this action.

3.1.3 Credit-Based Latency Insensitive Design

Networks-on-Chip frequently use a credit-system flow control method to avoid data loss [23] and thus can be considered as a form of LID. We develop a simple system based on the same ideas of credit-based flow control used in NoCs as a latency insensitive wrapper as shown in Fig. 3.5. The interface control signals are a valid bit and an increment count bit. A counter in each wrapper is used to store the number of words available in the upstream FIFO, called credits. If the counter runs out of credits then the pearl is stalled to avoid sending valid data downstream. A simple increment counter module is also placed in each wrapper to let the downstream module know that it can send more data by updating the downstream credit every time a word is dequeued from the FIFO buffer. In order to maintain functionality, the number of words in the FIFO must be greater than or equal to the round-trip latency (the sum of the forward path pipeline stages and backpressure path pipeline stages) between communicating modules.

The credit-based system can also use conventional pipeline registers instead of relay stations. It also avoids the use of enable signals, allowing it to easily target hyper-registers.

3.2 Results

The following sections compare the area and speed of the three different LID styles in two FPGA families: Arria 10, which has traditional interconnect and Stratix 10 which has optional pipeline registers in its interconnect.

We gathered results using the Quartus Prime Pro 17.1 CAD tool. The target FPGAs used in our

Table 3.1: Area overhead of 32-bit wide latency intensive design wrappers

Resource	Stratix 10 Designs			Arria 10 Designs		
	Carloni	Credit	Ready-Valid	Carloni	Credit	Ready-Valid
ALMs (ALUT + FF pairs)	23	29	23	14	20	15
ALUT	40	54	41	32	41	32
FF	34	42	34	26	32	26
M20K	1	1	1	1	1	1
Equivalent ALMs	63	69	63	54	60	55

Table 3.2: Area of adding ten 32-bit wide pipeline stages for each method of LID as well as a non-LI implementation

Resource	Stratix 10 Designs				Arria 10 Designs			
	Carloni	Credit	Ready-Valid	Non-LI	Carloni	Credit	Ready-Valid	Non-LI
ALMs	327	90	98	78	338	173	172	157
ALUT	405	0	0	0	71	0	0	0
FF	588	175	200	164	717	341	340	320
Hyper-registers	264	165	162	156	–	–	–	–
Equivalent ALMs	327	90	98	78	338	173	172	157

experiments are the Arria 10 10AS066H1F34E1SE, and the Stratix 10 101SX210HU3F50E1VG. Both are mid-size FPGAs and operate at the fastest speed grade. All reported results are averaged over 5 seeds to minimize noise due to the stochastic nature of CAD algorithms.

To compare the total layout area between LIDs, which use hardened RAM blocks, we summarize their area in terms of equivalent ALMs (eALMs). In [68] it is reported that cost of a M20K RAM block is 40 eALMs for the Stratix V architecture. Since Arria 10 and Stratix 10 use the same M20K block and similar ALM design to Stratix V we assume the ratio stays the same. In addition, when reporting Stratix 10 results we assume the use of hyper-registers does not add any area. This is because they are plentiful in the chip and always available along the current routing path. Hence, logic and RAM utilization are what will limit implementable design size.

For optimal hyper-register usage in the Stratix 10 designs we followed the suggestions in [34] including correctly setting Quartus CAD assignments and using their verilog templates.

3.2.1 Wrapper Area

Table 3.1 shows the relative area of the latency-insensitive wrappers for Stratix 10 and Arria 10 respectively. The reported resources are ALUTs, flip-flops (FF), and M20K blocks. ALUTs and FFs are merged into ALMs; finally ALMs and M20K blocks values are used to calculate the final eALM result. To gather the area results we used a simple pearl module consisting of a registered 32-bit wide datapath. The results shown in the tables exclude the area of the pearl module and show only the area overhead of a single wrapper. In Table 3.1 we see that the area overhead of all the latency intensive wrappers is relatively small, less than 70 eALMs.

Additionally, the area of the wrappers does not increase with interconnect pipelining depth. The M20K Block RAM (BRAM) can store 512 words. For the credit-based and ready-valid wrappers that

require the buffering depth to be at least twice the number of interconnect channel pipeline stages, this means that you can add as many as 256 pipeline elements without the need for additional BRAM. This is many more than needed to cross even the largest FPGAs at high frequency. The only other possible area increase due to pipelining is the size of the credit counter for the credit-based wrapper, which is relatively small. The area of the credit-based wrappers reported in Table 3.1 include a 7-bit counter for the Stratix 10 design and a 5-bit counter for the Arria 10 design. These are sufficient for 128 pipeline stages and 32 pipeline stages respectively. Their area overhead is small, ∼6 ALMs for both cases.

Although the wrappers can handle more pipeline stages with no increase in area, this is not the case for increasing the port width. While the pearl control logic does not grow with datapath width, the FIFO width does. The M20K BRAM can handle a maximum of 40-bit wide words. Increasing the datapath beyond 40-bits requires the addition of extra BRAM blocks. Since the area of our wrappers is mostly dominated by the area of M20K blocks this can become expensive. For example, having a 64-bit datapath would require two BRAMs, nearly doubling the area of the LI wrappers. If we move to a very large datapath of 512-bits we would require a total of 13 BRAMs and increase the wrapper area to 590 eALMs which is 8.4x the size of the LI wrappers at 32-bits.

Increasing the number of input channels also increases the overhead of the wrappers due to the addition of the extra BRAM blocks needed for each input. A small amount of control logic for the FIFO and pearl enable is also added with each extra input, but this area increase is trivial compared to the area of the BRAM blocks.

3.2.2 Pipeline Area

To fully compare the LID styles, we consider not only the area of the module wrappers but also the area of the pipeline stages between them. For both Stratix 10 and Arria 10 we compare the area of adding 10 pipeline elements between two modules floorplaned to opposite corners of the FPGA.

Table 3.2 shows the area of pipelining for Stratix 10 and Arria 10 respectively. Recall that the area cost is determined by the number of ALMs used because we assume hyper-registers are plentiful thus they are free. The first big trend we see is that pipelining is cheaper for credit-based and ready-valid systems compared to the Carloni system. The Carloni system requires 1.9x the area for pipeline elements compared to other LI solutions for the Arria 10 FPGA. This is because relay stations contain 2x the registers verses the other systems and also need additional area for multiplexers and control signals. The area gap is much greater on the Stratix 10 FPGA at 3x. Compared to non-LI, the credit-based and ready-valid system have a small area overhead of less than one equivalent LAB. This is because the feedback signal in the LI systems is also pipelined.

We also see that pipelining is cheaper for all systems, except the Carloni relay stations, for Stratix 10 than it is in Arria 10. The area decrease due to the availability of plentiful hyper-registers in Stratix 10 is significant: ∼50 % of the eALM area. The Carloni-based system still uses a significant number of hyper-registers but this is done purely for timing. It does not save any area because the registers inside the relay stations use an enable signal which means that an ALM duplicate register is needed as shown in Sec. 2.4.

We expected that when pipelining is all simple registers, the pipelining area would be near zero. Since the interconnect for the credit-based, ready-valid, and non-LI systems is made up of only simple registers with no enables, they should mostly be targeting hyper-registers with minimal ALM flip-flops.

However, we found that Quartus typically only used 2-3 hyper-registers in series before going through an ALM register, perhaps to avoid hold time violations that may arise from chaining closely spaced pulsed latches.

3.3 Speed Results

3.3.1 Pipeline Scaling

To study the speed of the LI wrappers we start off by analyzing the efficiency of communication over long distances using the previously mentioned 32-bit wide simple circuit, floorplanned to opposite corners of the FPGA. For the Stratix 10 design we added a phase-locked loop (PLL) with a 1 GHz output clock to avoid having clock pin I/O delays limit the critical path. Fig. 3.6 shows the operating frequency of the design at various levels of pipeline depth. A surprising result is the substantial number of pipeline stages needed for full frequency communication across the chip in Stratix 10 compared to Arria 10. Sixty pipeline stages are needed to operate at the best achieved frequency of 750 MHz across the chip. This is 6x the number needed for Arria 10 highlighting the increased importance of high-latency communication as process technology scales.

In Fig. 3.6 we can also see that the credit-based and ready-valid system achieve the same frequencies as the non-LI system. Whereas previous work showed that there is a speed overhead of 17% for LI systems on FPGA [61], using credit-based and ready-valid systems we were able to close the speed gap. This is because they use the same type of interconnect pipelining as the non-LI system and the wrappers are optimized for speed. On the other hand, the Carloni systems' operating frequency plateaus before all the other systems and is is clearly inferior for high levels of pipelining.

The average Fmax difference of Carloni-based LID verses the FIFO-buffering-based LI solutions (ready-valid and credit-based) across all pipelining depths is 60 MHz for the Arria 10 and 90 MHz for the Stratix 10. The best Fmax difference is even greater, 66 MHz on the Arria 10 and 164 MHz on the Stratix 10. The bigger difference on the Stratix 10 verses Arria 10 can be attributed to Stratix 10's hyper-registers in the interconnect which are better exploited by the credit-based and ready-valid wrappers.

3.3.2 Width Scaling

We also studied the effect of port data widths on speed. To do this we again used the the previously mentioned 32-bit wide simple circuit, floorplanned to opposite corners of the FPGA. We varied the width of the input and output channels of the wrapper/pearl module pair. For each width we averaged the speed result across multiple levels of pipelining from 0-100 pipeline stages. Fig. 3.7 shows this result.

As the data width increases, the speed of all systems decrease. The credit-based and ready-valid system frequencies decrease slightly more than the non-li system. This is because the critical path is increasingly within the LI wrappers due to the higher fanout FIFO control signals for high widths. The Carloni-based system's frequency is significantly lower than those of the other systems at all widths and also decreases with width at about the same rate as the other LI systems. The Carloni-based system's critical path though is due to the high-fanout control signals inside the relay stations which are proportional to the change in width. The high-fanout control signals of the relay-station grow in delay as the data width increases due to the additional signals needed to control an increasing amount

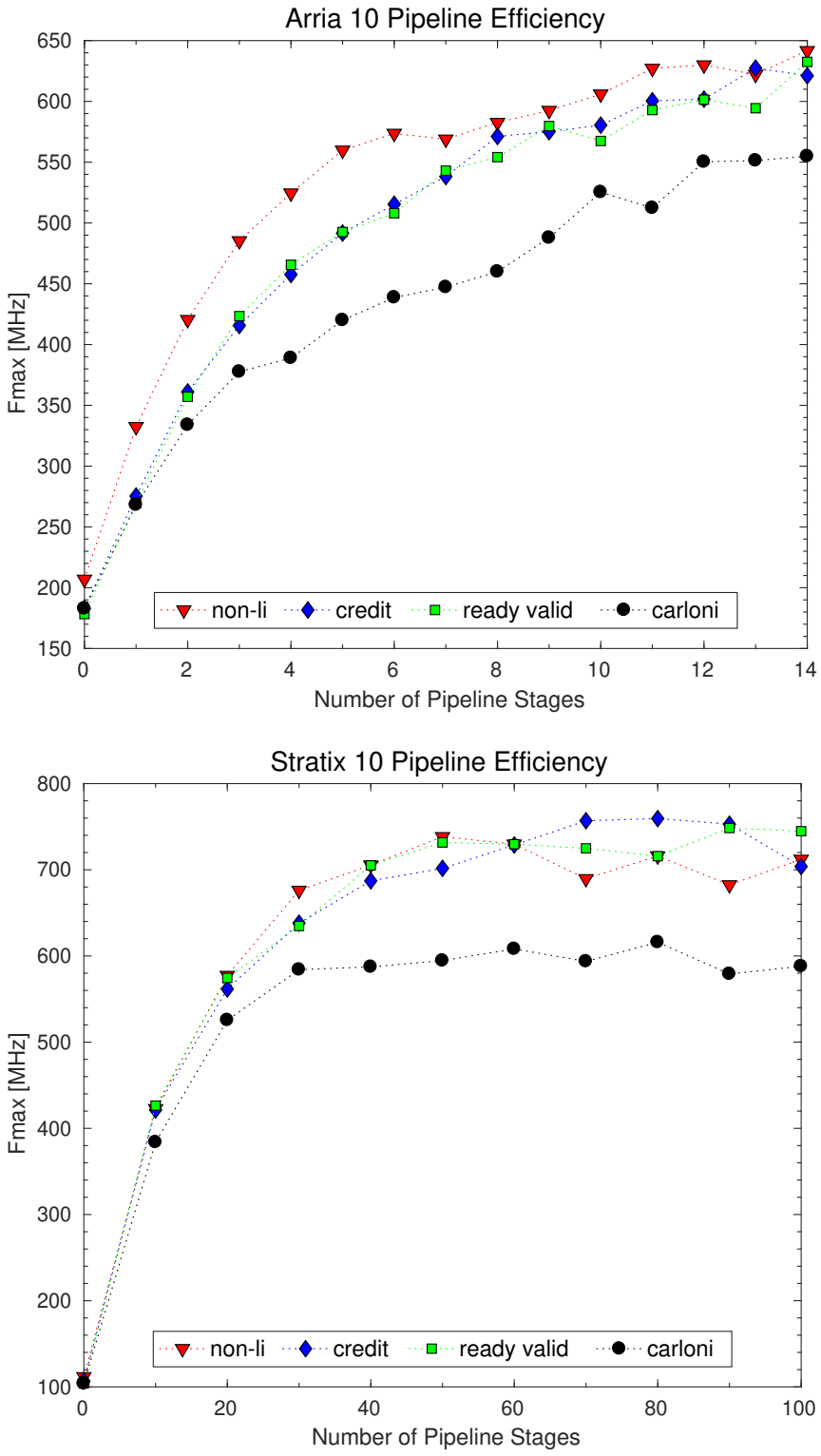


Figure 3.6: Average operating frequency across five seeds for corner-to-corner communication as the amount of pipelining increases.

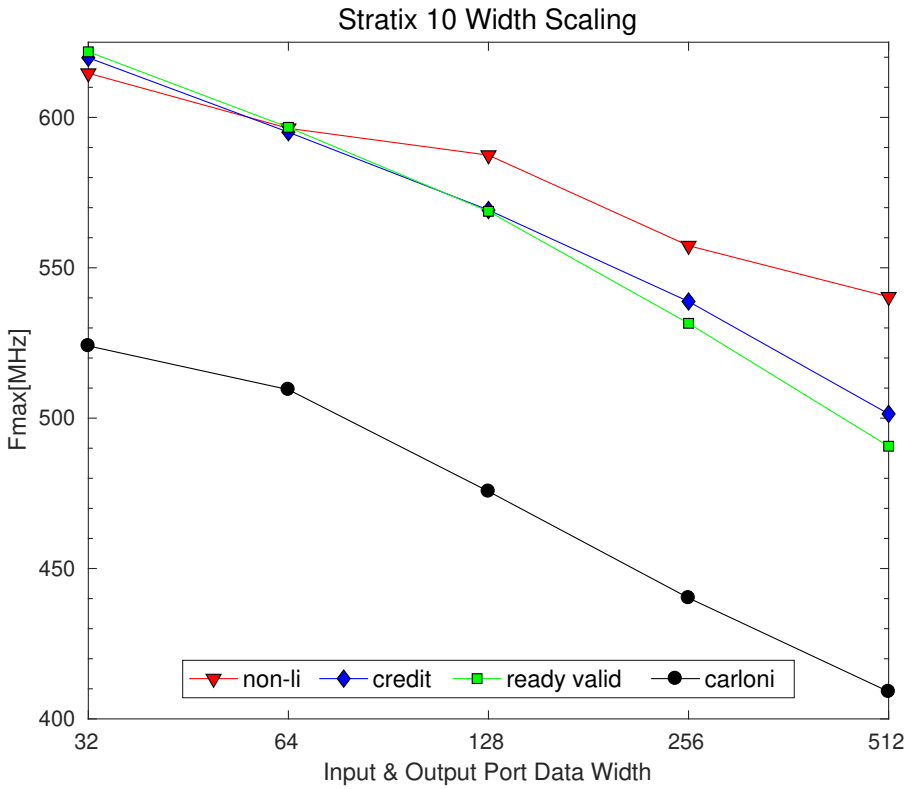


Figure 3.7: Average operating frequency across 10 different pipeline depths ranging from 0-100 as port size increases.

Table 3.3: Average operating frequency for different design styles when varying pipeline depths and data-widths

Design	Fmax [MHz]	Difference to Non-LI
Non-LI	579	–
Credit	565	-2.4%
Ready-Valid	562	-2.9%
Carloni	471	-20.6%

of registers in the relay station.

We average the frequency data of Fig. 3.7 across the various port data widths to obtain the overall frequency comparison of the four design styles shown in Table 3.3. Overall the speed reduction for credit-based and ready-valid systems is minimal at 3% slower than the non-li system, while the Carloni-based system average 20% slower than the non-li system.

3.4 Design Scaling using Circuit Example

We use a finite impulse response (FIR) cascade as a benchmark to test the efficiency of the LID systems as the chip is filled. A cascaded FIR module is a number of FIR modules connected together in a feedforward fashion. In our case each FIR module is the pearl inside of a LID wrapper. In between each wrapper we have latency-insensitive communication channels where we can insert pipeline elements. For the non-LI system, manual pipelining between FIR modules could be easily done. Since we are not using this design example as a comparison of design effort between LID and non-LI design but instead as a high speed design example to compare design style efficiency it is appropriate to use it as a benchmark, as also was done in [61].

The FIR module consists of 13 chained DSPs. Each DSP in the Stratix 10 and Arria 10 FPGA contain two multiplies and an adder tree that connects the output to the chained input of the adjacent DSP. The filter has 51-tap symmetric coefficients; therefore, chain adders in the soft logic are used to implement a pre-adder for the inputs before going into the multiplier. The module's inputs and outputs are 17-bits wide. The module was designed to operate at the fastest speed of the respective DSP block on the FPGAs used: this was 550 MHz for Arria 10 and 750 MHz for Stratix 10.

Fig. 3.8 shows operating frequency as the number of FIR modules increases to fill the chip. The plot shows the result for both no pipelining elements between FIR modules (dashed lines) and with the addition of pipelining elements (solid lines). Since Stratix 10 is a bigger device we tested the designs using four pipeline stages between FIR modules, while for Arria 10 we use two. For the Arria 10 design at zero pipeline stages the non-LI system does the best and the LI-systems cost some speed overhead. Beyond 20 modules we see a frequency drop; however, as we pipeline more we can handle larger designs. With two pipeline stages, designs with 40 FIR modules can gain a significant frequency improvement. As the design size increases the frequency drops rapidly suggesting that more pipeline stages would be necessary to improve operating frequency. This highlights the importance of decoupling the design of the interconnect from the design of the IP-blocks so that it is easier to place pipeline elements on long connections.

In addition, at two pipeline stages in the Arria 10 design the ready-valid system performs nearly as

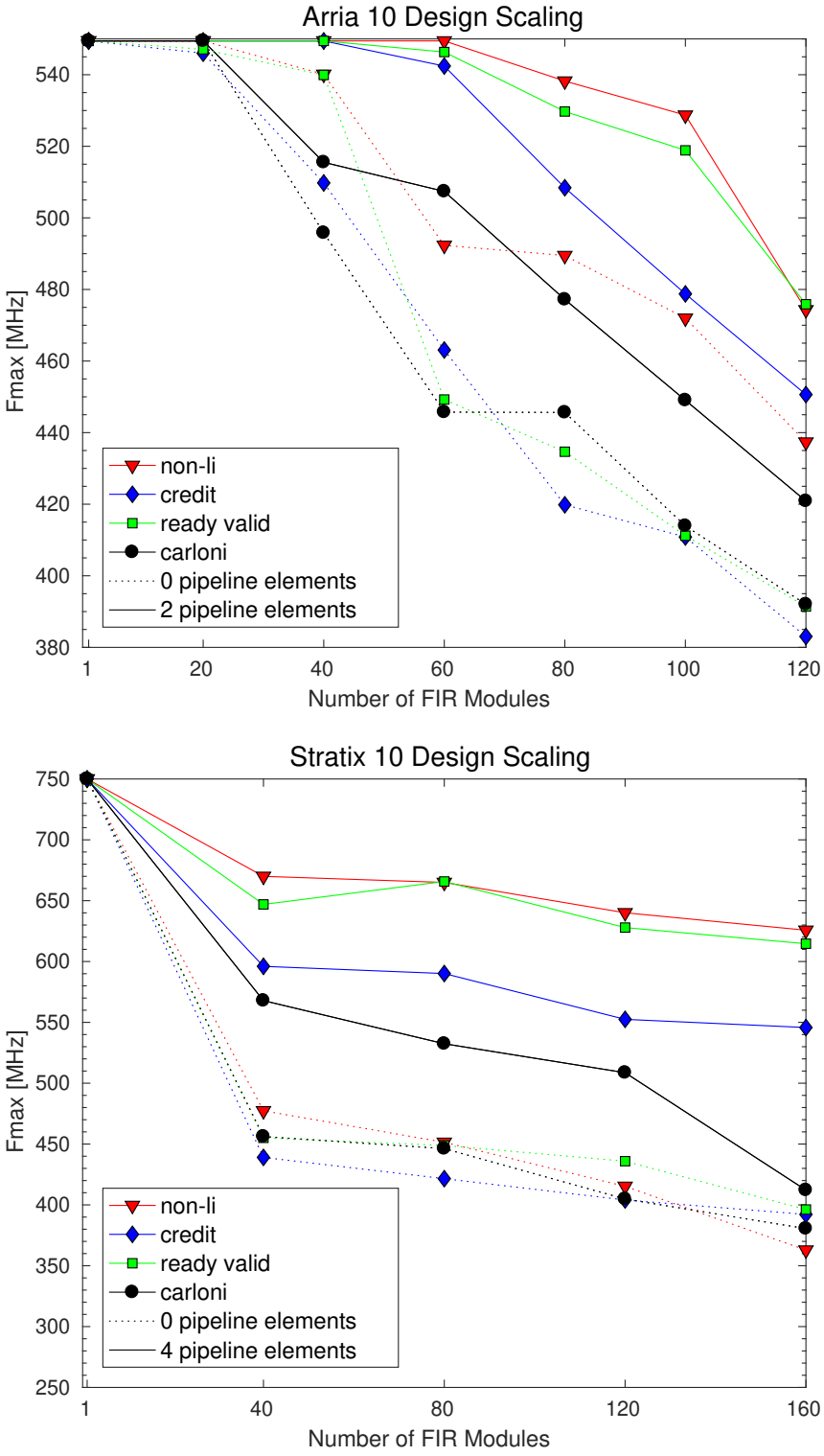


Figure 3.8: Average operating frequency across five seeds as design size increases with and without system-level pipelining.

well as the non-LI system. The credit-based system performs worse than the ready-valid system because of the additional logic needed for the credit counters. While Sec. 3.3.1 showed that the credit based system can run at high frequency, the counter logic slightly increases the size of the placement problem and appears to lead to less optimal placement in these larger designs. Finally, the Carloni-based system has the lowest performance, matching the trends of Sec. 3.3.1.

On the Stratix 10 design scaling plot in Fig. 3.8 we see a similar difference between the performance of the credit-based system and the ready-valid system. We also still see that the Carloni-based system again has lower performance than the other solutions. The only difference verses Arria 10 is with zero pipeline stages, all systems perform relatively similar. This may be due to better placement of BRAMs used for FIFOs.

Overall, the trends seen using the FIR benchmark are similar to those seen in Sec. 3.3: ready-valid can achieve a similar speed to the non-li design and Carloni-based LID achieves the slowest speed. One result that is different is that the credit-based system, while still better than Carloni-based now performs worse than the ready-valid system.

Chapter 4

Clock Architecture Modeling

The reconfigurable nature of FPGAs have made them a popular acceleration platform for applications such as: wireless communication systems [22], real-time video and image processing [65], compression [3], biomedical applications like protein identification [12], amongst many others. After recent demand for FPGAs in the datacenter [67], FPGAs are being more widely used in complex high performance computing applications such as, web search engines [67] and real-time artificial-intelligence using neural networks [25]. These complex applications include multiple independent components communicating at the system-level, each of which may be using it's own distinct clock signal. For that reason, it is important that the FPGA clock network is flexible to match to the clocking demands of a diverse set of applications. Furthermore, these applications are expected to operate at ever higher speeds with each new process technology node. Faster transistors and larger chips have enabled these large systems to be integrated on a single device. On the other hand, wire speed has scaled poorly, leading to increased insertion delay of large clock networks. The clock network needs to be flexible such that it is able to form multiple small clocks for different parts of a design. This is to avoid the creation of large clock networks which have larger loads and whose delay will be subject to more skew from temperature, on-die process variation, and voltage variation, thereby decreasing the time per clock cycle available for computation. Not only is the clock network a fundamental limiting factor in high speed design, but their high toggle rates of hundreds of MHz to GHz makes them dissipate significant dynamic power.

To cope with the clocking challenges of large FPGA designs, we need to rethink the design of clock networks. Commercially, various clocking architectures have been explored to make the clock distribution network more flexible and programmable [24, 85] and to change the granularity and types of clock network buffering [80, 27] to improve design speed and power while maintaining a reasonable clock network area.

However, academically there is no available flexible modeling format that can evaluate the effect of different clock architectures on power, area, and speed. Additionally, since academic FPGA CAD tools currently neglect and abstract away the clocking of a design, the impact of implementing clocking is not understood or considered during CAD stages like placement and routing, yet these stages need to consider clocking in real chips. With the recent interest in open-source FPGA CAD tools for targeting commercial devices, the lack of clock support is also a major limitation that prevents open-source CAD tools from implementing complete designs in commercial FPGAs. We present a flexible clock architecture modeling format that is built into the widely used open source CAD tool VTR [52], along with enhancements to

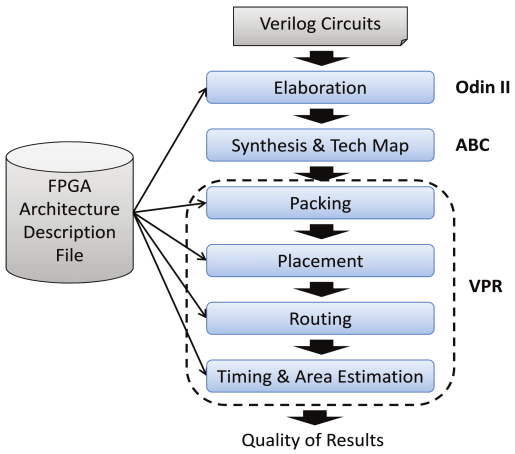


Figure 4.1: VTR flow. Figure taken from [52]

VTR's routing algorithm to efficiently exploit these networks. Together these enhancements allow the assessment of the impact of clock network changes on design performance and chip area, and enables the implementation of complete designs on existing FPGAs. The modeling format is flexible and can be expanded to include more features.

4.1 Language and Capabilities

We built our clock architecture modeling format into the publicly available CAD tool VTR (Verilog to Routing). This tool chain can perform the full suite of basic CAD steps, as shown in Figure 4.1. The flow takes an input Verilog circuit description and performs elaboration, logical and physical synthesis, and finally timing analysis and area estimation. A major advantage of VTR is that it can take as its input a custom FPGA architecture description and supply it to different parts of the CAD tool. It outputs not only final placement and routing information for the input architecture and circuit but important metrics that measure the quality of results, such as the number of resources (LC, BRAM, DSPs, ect.) used, the total routing wire-length, and the final critical path delay of the design. The ability to rapidly modify the architecture and re-evaluate the quality of results for a set of benchmark circuits has enabled a wide range of architecture studies in different areas such as redesigning logic [56, 5, 21], routing [10, 43], and introducing novel switches [58]. VPR has also been used as a framework for a wide range of CAD studies, such as creating a timing-aware placer [57], studing different packing algorithms [55, 20], and improving routing algorithms [75]. Our goal is to allow easy specification and automatic exploitation of clock network architecture studies, and CAD tool studies that include the important impact of clock implementation.

In VTR the FPGA architecture description language is based on XML, and we extended it to describe basic clock networks. An example of our clock architecture description is shown in Listing 4.1. We will walk through this listing to describe the full language we built to describe clock networks and its capabilities. Before we proceed, the following is some fundamental terminology and syntax that we use to describe XML [13]:

– Elements in XML are described using start <tag_name> and end tags \<tag_name>

- Elements can contain sub-elements, each with their own start and end tags
- The element start tag can contain one or more attributes: <tag_name attribute="value">
- The lowest level elements do not use a closing tag; instead they end with \>. For example:

  <tag_name attribute1="value1" attribute2="value2"\>
- The syntax for comments is as follows: <!-- comment -->
- We use "{option1 | option2}" to denote options within attribute values

The element <clocknetworks> contains three sub-elements that collectively describe the clock architecture: the wiring parameters <metal_layers>, the clock distribution <clock_network>, and the clock connectivity <clock_routing>.

The <metal_layers> element describes the per unit length electrical parameters used to implement the clock distribution wires. There can be one or more wiring implementation options (metal layer, width and spacing) that are used by the later clock network specification and each is described in a separate <metal_layer> sub-element. The syntax of the wiring electrical information is:

```
<metal_layer name="string" Rmetal="float" Cmetal="float"/>
```

Where the attribute name denotes a unique string for the metal layer that can be used to reference it later, and the attributes Rmetal and Cmetal describe the wire parameters. The value of Rmetal is the resistance in Ohms and the value of Cmetal is the capacitance in Farads per unit block in the FPGA architecture; a unit block usually corresponds to a logic cluster.

The <clock_network> element contains sub-elements that describe the clock distribution wires for the clock architecture. There could be more than one <clock_network> element to describe separate types of distribution wires. The high-level start tag for a clock network is as follows:

```
<clock_network name="string" num_inst="integer">
```

It contains two attributes: The first is a unique name for reference. The second is num_inst which describes the number of parallel instances of the clock distribution types described in the <clock_network> sub-elements. The supported clock distribution types are <spine> and <rib>:

```
<spine metal_layer="string" x="integer" starty="integer" endy="integer"
    repeatx="integer" repeaty="integer"/>
```

```
<rib metal_layer="string" y="integer" startx="integer" endx="integer"
   repeatx="integer" repeaty="integer"/>
```

Spine is used to describe vertical clock distribution wires. Its attributes begin by referencing a metal_layer that sets the unit resistance and capacitance of the distribution wire over the length of the wire. The wire runs parallel to the y-axis from starty and ending at endy, inclusive. The location of the spine with respect to the x-axis is denoted by the integer value inside the attribute x. repeatx and repeaty are optional attributes used to describe the horizontal and vertical repeat factor of the spine along the device grid. In Listing 4.1, we define two spines, and neither repeats as each spans the entire height of the device and is locally at the horizontal midpoint of the device.

Listing 4.1: Example clock architecture: global rib and spine.

```
<clocknetworks>
    <metal_layers>
        <metal_layer name="global_spine"
            Rmetal="50.42" Cmetal="20.7e-15"/>
        <metal_layer name="global_rib"
            Rmetal="50.42" Cmetal="20.7e-15"/>
    </metal_layers>

    <!-- Full Device: Center Spine -->
    <clock_network name="spine1" num_inst="2">
        <spine metal_layer="global_spine"
            x="W/2" starty="0" endy="H">
            <switch_point type="drive" name="drive"
                yoffset="H/2" buffer="drive_buff"/>
            <switch_point type="tap" name="tap" yoffset="0" yincr="1"/>
        </spine>
    </clock_network>

    <!-- Full Device: Each Grid Row -->
    <clock_network name="rib1" num_inst="2">
        <rib metal_layer="global_rib" y="0" startx="0"
         endx="W" repeatx="W" repeaty="1">
            <switch_point type="drive" name="drive"
                xoffset="W/2" buffer="drive_buff"/>
            <switch_point type="tap" name="tap"
                xoffset="0" xincr="1"/>
        </rib>
    </clock_network>

    <clock_routing>
        <!-- connections from inter-block routing to central spine -->
        <tap from="ROUTING" to="spine1.drive" locationx="W/2"
            locationy="H/2" switch="general_routing_switch" fc_val="1.0"/>
        <!-- connections from spine to rib -->
        <tap from="spine1.tap" to="rib1.drive"
            switch="general_routing_switch" fc_val="0.5"/>
        <!-- connections from rib to clock pins -->
        <tap from="rib1.tap" to="CLOCK"
            switch="ipin_cblock" fc_val="1.0"/>
    </clock_routing>
</clocknetworks>
```

Rib is used to describe a horizontal clock distribution wire. Its attributes are similar to those of the spine but are rotated 90 degrees. Note that since VTR allows the device size to change automatically to accommodate the design being implemented, you can specify the spine and rib locations as well as the repeat increments relative to the device size. An FPGA grid (array of block locations) is defined to span the range from (0,0) to (W, H) so a value of "W/2" means a point at the center of the device width, for example.

Along each spine and rib is a group of switch points:

```
<switch_point type="{drive | tap}" name="string" yoffset="integer"
   xoffset="integer" xinc="integer" yinc="integer" buffer="string">
```

Switch points are used to describe drive or tap locations along the clock distribution wire. Drive points are where the clock distribution wire can be driven by a routing switch or buffer and tap points are where it can drive a routing switch or buffer to send a signal to a different clock_network or logic block. Depending on whether the switch point describes a rib or spine, the location offset in the x or y direction, respectively. For taps (but not drive points) you can describe the repeat factor, xinc or yinc, to describe a series of evenly spaced tap points with one switch_point element. For drive points you are required to specify the name of the buffer (a routing switch defined elsewhere in the architecture file) that will be used at the drive point. Our clock generator uses two buffers to drive the two portions of this clock_network wire when it is split at the drive point.

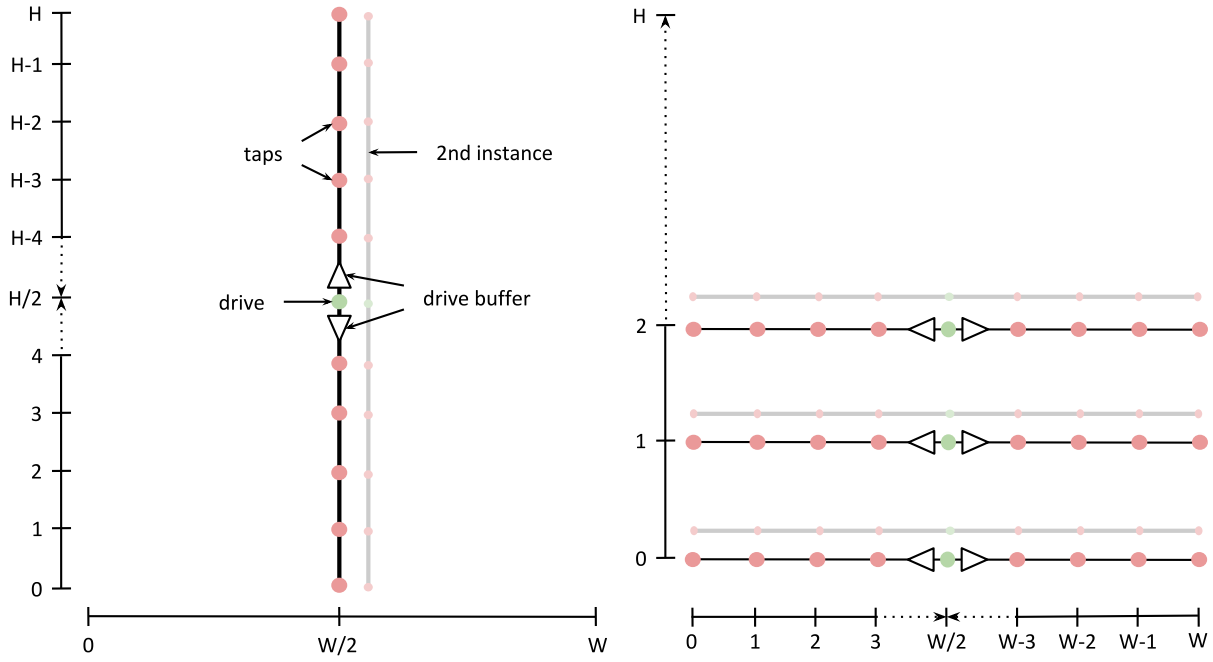
Lastly the <clock_routing> element consists of a group of tap elements that each separately describe the connectivity from one routing resource (pin or wire) to another. The tap element and its attributes are as follows:

```
<tap from="string" to="string" yoffset="integer" locationx="integer"
   locationy="integer" switch="string" fc_val="float">
```

The to and from attributes reference the name of the routing resources to connect together. To connect together clock distribution (clock_network) wires, a tap from one wire has a switch or a buffer that can drive the drive point of another distribution wire. The referencing schema can be seen in Listing 4.1 where the unique clock network name is followed by a period and then the switch point name. For example, in the listing the spine tap points "spine1.tap" sources the rib drive points "rib1.drive" at (x,y) locations where there is both a spine tap and a rib drive. Additionally, the from and to attributes can reference a connection from the general inter-block routing at a particular (locationx, locationy) coordinate using the special literal "ROUTING" Another special literal "CLOCK" describes connections from clock network tap points that supply the clock to clock pins on blocks at the tap locations; these clock inputs are already specified on blocks in the VTR architecture file. fc_val is a decimal value between 0 and 1 representing the connection block flexibility between the connecting routing resources; a value of 0.5 for example means that only 50% of the switches necessary to connect all the matching tap and drive points would be implemented. The switch attribute specifies the name of the routing switch used for the connection; it can reference any switch defined in the architecture file.

These components allow us to describe a range of clock architectures. We can describe:

- Architectures with arbitrary number of rib and spine networks that span the chip (global clocks).
- Smaller clock networks that span only a portion of the chip.



(a) Vertical Distribution (spine)

(b) Horizontal Distribution (rib)

Figure 4.2: Clock network distribution wires of Listing 4.1

- Architectures where a larger number of spines must multiplex down to a smaller number of ribs.
- A multilevel network of ribs and spines.
- An H-tree, or a portion of an H-tree that eventually drives a rib and/or spine network at the lower levels of the clock distribution.
- combinations of the above

While the architecture specification and parser are quite general, we have tested only a subset of these network styles through the clock routing resource graph generator (see Section 4.2) and enhanced routing algorithm we created. Specifically we have tested different numbers of global rib and spine network, different network to block clock pin connectivities, and some H-tree topologies.

We use the architecture of Listing 4.1 for the clock routing algorithm presented in this chapter. This clock architecture consists of two independent chip-wide spine and rib network, each of which can drive the clock pins on every block in the FPGA. Figure 4.2 shows the layout of the spine and rib distribution wires of Listing 4.1. The vertical distribution wire in Figure 4.2a is sourced at the center of the device at coordinates ($W/2$, $H/2$) and spans the full chip vertically. Tap locations are located in increments of one vertical unit along the wire. This distribution network is replicated for a second instance at the same location. The horizontal distribution wires are shown in Figure 4.2b; while Listing 4.1 has only one line specifying a rib, it repeats in increments of one vertically to form multiple ribs of Figure 4.2b. Each tap point on the spine network drives a rib drive point, when their coordinates match. The tap points on the rib distribution wire are repeated every increment of one unit, and the F_c to clock pins on blocks is 1.0, so each FPGA clock can receive signals from both clock network instances. The clock signal is driven onto the root of the clock network via the general inter-block routing; all wires in both

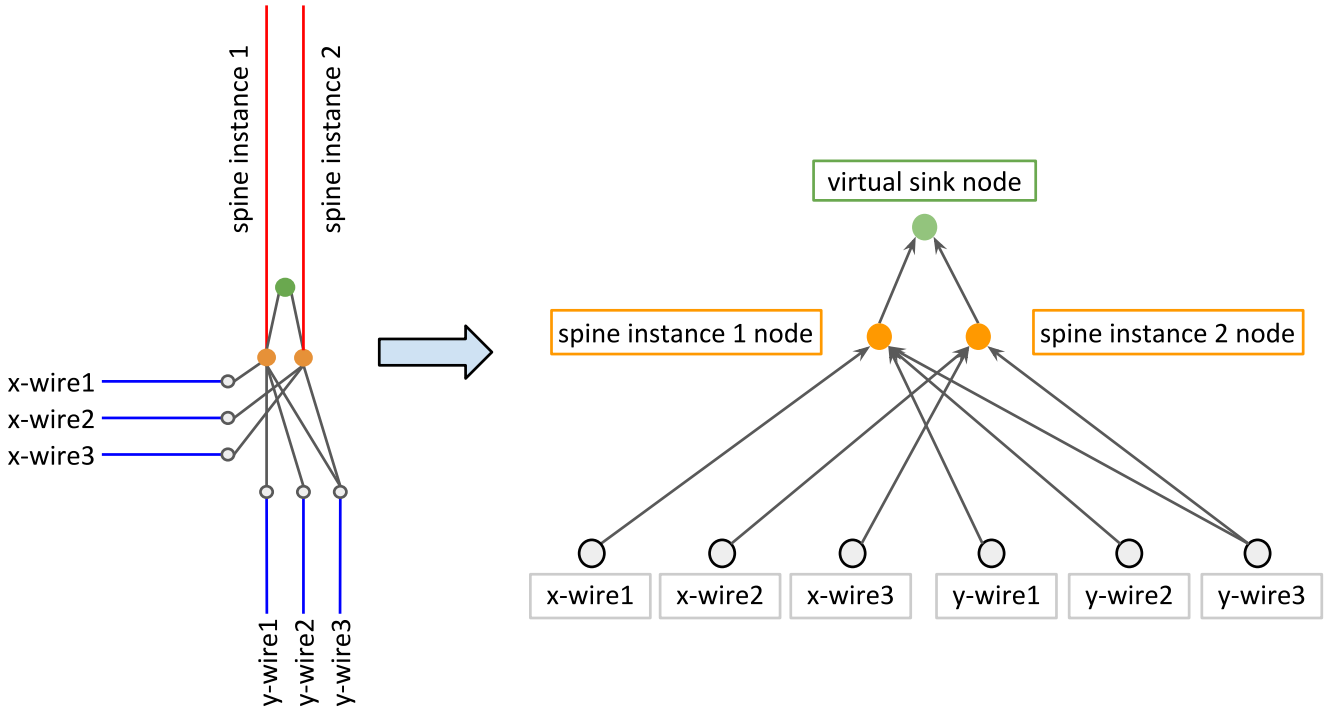


Figure 4.3: The clock network routing architecture (left) and its representation as a routing resource graph (right). This specifically shows the connectivity from inter-block routing to the clock network.

x and y directions in the routing channel segment at the center of the chip can drive both the clock network spines via a routing mux, as specified by line 33 of Listing 4.1.

4.2 Clock Net Routing

The routing algorithm in VTR uses a directed graph to describe the routing resources available in an FPGA. The nodes in the routing resource graph represent block pins and wires and the edges represent switches that can be turned on to implement connections between nodes [11]. The clock architecture is considered as another routing resource in the graph where each clock distribution wire is represented as a node with edges to other clock distribution wires, inter-block routing, and/or clock pin nodes.

In FPGAs, clock pins can often be sourced, or driven, from either the general FPGA inter-block routing or the clock network. Consequently, it is possible for clock nets to route all or a portion of their fan-out using just inter-block wires, thereby avoiding the clock network completely or partially. Generally such a routing is undesirable as it leads to higher skew clock than would be achieved if the global clock networks were used as much as possible. To aid in routing from FPGA I/O clock sources to the drive point of the clock network which requires use of the general FPGA routing, while also forcing use of the global clock network to route as many clock net fanouts as possible to minimize skew, we created a two stage routing algorithm in VPR. The first stage routes from the clock net source pin to one of the drive points of the clock network instances. The example clock network we use in Listing 4.1 contains two instances of the clock network, i.e. two drive points. The second stage routes the pins of the clock net normally using the drive point as the new root of the clock fan-out net.

To accomplish the first stage routing we create a virtual sink that connects the clock network drive points that originate from inter-block routing. An example routing resource graph representation for the virtual sink we create is seen in Figure 4.3. The virtual sink gives us a target for the first routing phase that connects from the clock generation point to a root of some clock global clock network, and allows the router to select which clock network root is the best target from a wirelength and congestion perspective. Assume we have two clock nets. For the example in Figure 4.3 the first net will route to the virtual sink node and will pass through one of the clock spine instances. The second node will also route to the virtual sink and may pass through the same clock spine instance as the first net. However, the VPR router uses a negotiated congestion-based router [59] which will eventually make the cost of passing through a wire that can only accept one signal more costly forcing the later net to use the second of the two spine instances. This is simular to the way logical pin equivalence is implemented for logic blocks in VPR [54].

Listing 4.2: C++ Pseudo-code of two stage clock routing algorithm.

```
bool route_net(ClusterNetId net_id) {
   /* Initialized from a previous routing iteration or null */
   rt_node* rt_route = initialize_route_tree(net_id); // route tree

   /* New code: Pre-route to clock root for clock nets */
    if(net_is_global){ // only nets marked as global nets
      int sink_node = virtual_clock_network_sink_idx; // stored globaly
      float criticality = max_criticality; // Max Criticality = 0.99
      pre_route_sink(net_id, sink, criticality, rt_root);
    }

    /* Existing code: Route net sinks in order of decreasing criticality */
    for (int pin: ordered_pins){
      float criticality = pin_criticality[pin];
      route_sink(net_id, pin, criticality, rt_root);
    }
}

pre_route_sink(
   ClusterNetId net_id, int sink, float criticality, rt_node* rt_root){
   heap_ptr = route_sink_from_route_tree(net_id, sink, criticality, rt_root);
   /* For the found connection: build the route tree and the net traceback */
   update_net_traceback(heap_ptr, net_id);
   update_route_tree(heap_ptr, route_tree);

   /* Remove virtual sink from route tree and traceback and fix route tree */
   remove_traceback_sink(net_id, sink);
   fix_route_tree_and_remove_route_tree_sink(route_tree, sink);
}
```

Listing 4.2 shows pseudo code that describes the two stage router algorithm. `Route_net()` is the main routine that routes a net from its source pin to all its destination pins on the FPGA. If the net we are routing is a global net we start the two stage router, as shown in line 6. Note that we assign clock nets to be global, but other timing critical high-fanout nets can similarly be assigned. The first step of the two stage router is to pre-route the net from its source to the clock driver node (`virtual_clock_network_sink_idx`). We start by assigning a `criticality` value to the virtual sink. Criticality is a number from 0 to 0.99 that sets how timing critical a connection is; the more timing critical the connection the higher the criticality value. Based on the level of criticality the negotiated congestion-based router gives the connection a higher or lower priority when choosing which connections to rip-up and re-route. Unlike pins where timing analysis can assign a slack or criticality to the connection we have no clear way of defining the criticallity of the virtual clock network sink. We choose to set it to max criticality which ensures that we minimize clock insertion delay by selecting a more direct root from the clock net source to the clock network root.

We pass the criticallity value and the virtual clock sink to the pre-route routine (`pre_route_sink()`). This is the first stage routing which results is a partial route from the clock net source to the clock network driver. To do this, the router updates a heap with the nodes explored to reach the virtual clock sink. The function `route_sink_from_route_tree()` returns a heap pointer to the found sink. From this heap pointer we are able to create a route tree and net traceback which are used to store information on the partial routing. The route tree is especially used to give the router information on where to branch from for routing of remaining net pins. In the route tree, we mark all the nodes except the clock network driver to not be re-expanded so as not to go back onto the heap as branch points (`fix_route_tree_and_remove_route_tree_sink()`). As we loop through all the net sinks in line 13 - 16 of Listing 4.2 we have ensured that we are only putting the clock network root back onto the heap, and that the routing expansion starts from there.

4.3 Results

We evaluate the quality of our new clock routing algorithm by implementing a set of benchmarks through the VTR CAD suite with various clock routing options. The target FPGA architecture is a modified version of the Comprehensive Architecture example (k6_frac_N10_frac_chain_mem32K_40nm.xml) in VTR [52]. The Comprehensive Architecture is similar to that of Stratix IV and Virtex 7 FPGAs; it contains DSPs, BRAMs, and LCs [52] and is based on the 40 nm process technology node. The LCs contain 10 fracturable 6-LUTs with built in carry-chain arithmetic, and the routing wire length is built up of length-4 unidirectional wire segments [52]. We build into this architecture the global rib and spine clock network that we described in Listing 4.1. We used the delay of the longest wire segment (length-4) in the architecture as an approximation to the unit resistance and capacitance of the clock network distribution wires. For input benchmarks we used the heterogeneous benchmark set from [52]. The set includes 21 benchmarks that implement a variety of real applications. The benchmarks contain up to 100k 6-LUTs, 564 mulipliers, 29k adders, and 2 clocks; the full listings and key characteristics of the designs can be found in [52]. We run the trunk (VTR 8.0) development code with the default options, plus our added clock network generation code and for some experiments our enhanced router code.

We gather results on the routed wire-length, route time, and final critical path delay to evaluate the

Table 4.1: Geometric mean of the heterogenous benchmark set [52] for different clocking methods. Averaged against 5 placement seeds where the channel width was relaxed to be 1.3 times that of the minimum for each benchmark.

Clocking Method	Wire-length [# LCs crossed]	Route Time [s]	Critical Path Delay [ns]
Ideal	40394	27.31	10.37
Routed	42063 (+4.1%)	35.68 (+30.6%)	11.96 (+15.4%)
Dedicated	41615 (+3.0%)	31.00 (+13.5%)	11.78 (+13.6%)
Dedicated (clock pin $F_c = 0$)	41215 (+2.0%)	35.69 (+30.7%)	10.66 (+ 2.8%)
Two Stage Router	41187 (+2.0%)	29.53 (+ 8.1%)	10.68 (+ 3.0%)

difference between use of our clock network to that of ideal clocks (not routed, and modeled as 0 skew) that are currently being used during placement and routing in VPR.

VPR can automatically select the minimum FPGA grid size and the routing channel width required by each design. We allow VTR to select the necessary grid size for all benchmarks. In our first experiment we ran the router in binary search mode to find the minimum channel width for each circuit per clock modeling method, ideal vs dedicated clock etc. We then re-ran the router with a fixed channel width of 1.3 times that of the minimum channel width for each benchmark. We repeated this for 5 placement seeds. Table 4.1 shows the quality of results for that run.

In Table 4.1 Clocking Method details the resource types available and algorithms used for routing clocks. *Ideal* is the baseline, which it ignores all clock nets, does not route them, and assumes their skew is 0. *Routed* routes all clock nets using general inter-block routing resources using the usual VPR negotiated congestion routing algorithm. *Dedicated* includes the clock network as part of the routing resources available for the routing of clock nets but does not run the two stage router to guarantee the use of the clock network instead of local routing; it instead uses the conventional VPR router. Note that we only use the clock network to drive clock pins so regular nets will not be able to use the dedicated network. *Dedicated (clock pin $F_c = 0$)* forces the router to use the dedicated clock network by making the F_c of clock pins to the general inter-block routing zero; this means that the only routing choice for global clocks is the clock routing network. Finally *Two Stage Router* includes the dedicated clock network in its routing resource graph as well as routing resource edges from regular routing to the block clock pins and uses the two stage routing algorithm to force clock nets to use the dedicated network.

From Table 4.1 we see that routing the clock using only inter-block routing increases the wire length and critical path delay of the designs significantly. Using a dedicated clock network without a two stage router still degrades the critical path as much as using only the general inter-block routing. This is because clock nets do not always find a dedicated clock network to route all their fanouts; instead the router usually uses the general inter-block routing for some clock net fanouts. This adds systematic skew and affects the critical path accordingly. Forcing the clock pin F_c to inter-block routing to 0 alleviates this issue; however, it removes the flexibility of allowing block clock pins to connect to both general inter-block and dedicated clock network routing wires. Commercial FPGAs usually include this flexibility, as it allows small clocks to be routed using conventional routing if the dedicated clock networks are fully used. Our two stage routing algorithm performs as good in critical path delay overhead and wire-length as forcing the block clock pin F_c to inter-block routing to be zero without removing the flexibility to use regular routing for clocks. It achieves this while also having a faster run time than *Dedicated (clock*

Table 4.2: Comparison of ideal vs dedicated clocks using the two stage router. The same placement was used for both the Ideal clock results and the Two Stage Router results. The results are the geometric mean of the heterogeneous benchmark set [52] with the channel width relaxed to be 1.3 times that of the minimum

Clocking Method	Wire-length [# LCs crossed]	Critical Path Delay [ns]
Ideal	36918	10.40
Two Stage Router	37431 (+1.4%)	10.56 (+1.6%)

pin F_c = 0), showing that our two phase routing algorithm guides the router to use the appropriate resources efficiently. In contrast to the *Dedicated (clock pin F_c = 0)* result, for which the router wastes time exploring local inter-block routing which will never lead to a valid route as it routes each fanout of the clock net.

It is worth noting that there is still some timing overhead to using a dedicated clock network with two stage routing verses an ideal clock network. The overhead in wire-length (3.0% from Table 4.1) is understandable since the dedicated clock network wires used count towards the final wire-length. However, we expect that the critical path delay to be very close to that of using an ideal network. This is because the VPR timing analyzer models wire delay as a lumped delay, meaning that the delay of a wire will be the same regardless of where taps are located [11]. This means that even though we tested using a rib and spine network, which should include some systematic skew, we would not actually see it reflected in the final critical path of the design.

To test and check if the critical path delay overhead of the *Two Stage Router* is caused by routing we ran both the *Ideal* clocking method and the *Two Stage Router* stating with the same placement. The results are reflected in Table 4.2. This reduces, but does not fully eliminate, the timing overhead. We still see a small critical path delay overhead of 1.6%. This indicates that we are losing some optimization quality by miss-estimating the clock network delay during placement; updating the placement delay estimator to understand the presence of low skew network should recapture this delay. We suspect that the remaining 1.6% overhead has to do with the way pin criticality is calculated in the router which affects the cost of routing nets. Pin criticality is a number between 0 and 1 that represents how timing critical a net is. The more critical it is the less likely it will be detoured to avoid congestion (increasing its delay) by the negotiation-based router. The way pin criticality is computed is proportional to the inverse of the total data plus clock path length [79]. If the clock is ideal, criticality is just a ratio of the data delay to the critical path delay. However, since the clock latency went from an ideal latency of zero to now the delay of a global network (about 1.8ns for a typical grid size) it has the effect of confining criticality to a narrower range, thereby changing the router's behavior. We believe that reformulating the router and timing analyzer formulation of criticality should recapture this timing penalty, but we leave this as future work.

Another interesting result that we can capture because the clock networks are part of the routing resource graph is the routing area or the clock network used. To mesure routing area VPR counts the number of Minimum-Width Transistor Areas [11]. The minimum-width transistor area is the area of the smallest transistor used for the process node plus its spacing. The area model involves summing the number of routing switches, with respect to their number of minimum-width transistor areas, traversed to route nets in a design. Table 4.3 compares *Ideal* verses *Two Stage Router* routing area used. With

Table 4.3: Minimum transistor width area for geometric mean of the heterogenous benchmark set [52] for different clocking methods. Averaged against 5 placement seeds where the channel width was relaxed to be 1.3 times that of the minimum for each benchmark.

Clocking Method	Routing Area [Minimum-Width Transistor Areas]
Ideal	3532783
Two Stage Router	3617723 (+2.4%)

respect to geometric mean across all benchmarks the *Two Stage Router* uses 2.4% more area than the ideal clock network. This overhead is due to the clock network area. It is important to note that the computing area based on minimum-transistor width areas assumes that the area is transistor limited. This is because we are only computing the area of the switches and buffers used in the routing network. However, because the clock network contains long wires we expect that the area would be metal limited. We consider using a different area model to account for clock network wires as important future work.

Chapter 5

Conclusions and Future Work

5.1 Latency Insensitive Design Summary

The increasing reliance on latency for global communication on FPGAs to achieve timing closure motivates investigation of FPGA-friendly LID. As we have seen we require 60 stages of pipelining to communicate across a Stratix 10 chip at maximum frequency. This number will only increase with newer process generations.

While the motivation for LID is clear, the best implementation on an FPGA is not. We compare three types of Latency insenstive design that can be used on FPGAs to ease the timing closure problem. The FIFO buffering solutions for LID (credit-based and ready-valid) are 2x more area efficient and 18% more speed efficient than the Carloni-based system on the traditional Aria 10 FPGA. Architectures with hyper-registers benefit even more. On Stratix 10, the FIFO buffering solutions for latency insensitive design are 3x more area efficient than Carloni-based ones. Compared to non-LI, LID systems only add a small area of less than 7 equivalent LABs to encapsulate the design modules for input port widths of 40-bits or less. FIFO buffering solutions add only a small (20%) area to pipeline elements compared to non-LI systems; this extra area arises from the need to pipeline a back-pressure signal.

The FIFO buffering solutions can be used at a small area overhead; however, with regards to operating speed we have found that one FIFO buffering solution is better than the other. The ready-valid design style performs the best. It is able to maintain little to no speed difference compared to the non-li system as design size scales, whereas the credit-based style degrades in frequency due to placement effects.

We have shown that with the right latency-insensitive style, LID can be efficiently used in FPGAs without significant area overhead and with little to no timing overhead.

5.1.1 Latency Insensitive Design Future Work

To take full advantage of LID, automated pipelining tools should be built in to the CAD flow to increase designer productivity. The efficiency of these pipeline tools can be studied in terms of design throughput.

One way to study the effects of automated pipelining in the post placement and routing stage is to expand the VTR architecture description to include piplinable routing, similar to Stratix 10. A post routing step to pipeline timing critical paths that are marked to be latency insensitive should then

be added to the CAD flow. Additionally, creating a latency insensitive benchmark set by adding LID wrappers around high-level computing modules in existing academic benchmark sets, such as the VPR benchmark sets [52] or the Titan benchmark set [64], would be necessary to evaluate the automated pipelining step.

5.2 Clock Architecture Modeling Summary

As wires continue to scale poorly with process technology advancement, it is important not only to study ways to improve inter-module communication styles like LID to simplify timing closure, but also how to build clock networks that produce low skew and are efficient in their wiring usage while still providing the flexibility needed to integrate many sub-systems communicating on many clock domains.

We created a clock modeling framework and integrated it into VPR sand developed a full end-to-end CAD flow with a routed global rib and spine clock network example. Our clock network modeling language is flexible enough to allow for future enhancements that can describe many commercial FPGA architecture clock networks.

5.2.1 Clock Architecture Modeling Future Work

There are four areas of future work that will aid in the better evaluation of a broader set of clock network designs: extending language support, extending router support, enhancing the router and placer, and creating better area and delay models.

Extending Language Support

To enable the evaluation of clock architecture ideas in current and future FPGAs a set of language enhancements can be added to the clock modeling framework:

- Adding simplified support for h-tree clock distribution can help easily define Stratix-V-like clock architecture which contain h-trees at the root of the clock network and ribs at the leaf. Note that we can already create h-trees using existing `rib` and `spine` elements but it requires multiple definitions of `rib` and `spine` elements, increasing the complexity of the architecture specification that must be provided.
- Adding support to describe buffer sizing and placement along a wire. Buffer placement at defined spacing along long wires breaks up the delay of long wires.
- Adding switch block patterns to the clock network description will make it possible to describe detailed routing connection patterns such as those in the Stratix 10 and Ultrascale routable architectures.
- Adding bidirectional `switch_points` would also aid in the creation of routable clocks similar to Stratix 10 and Ultrascale. Currently there is only support for `tap` and `drive` points.
- Allowing the clock network to drive block I/Os in addition to clock pins. This will enable the clock network to route high fanout timing critical nets that are not necessarily clock signals.

Extending Router Support

Currently the two stage routing algorithm we created only works with global clocks that have one location for clock network roots. To support multiple clock roots in different locations on the chip, for example global vs quadrant clock networks we must add an allocation step to our router. The allocation step would allocate the first stage of routing of clock nets to one of many virtual sinks that are at the locations of clock network roots.

This would involve having a data structure which, for each virtual sink, stores information on its (x,y) coordinates on the chip, and the number of clock drive points connecting to that virtual sink. Then depending on the number of clock nets and the number of clock networks available at each virtual sink, we would allocate the clock nets to ensure they route using a clock network.

If there are more global nets than there are available clock networks, i.e. there is unresolved congestion at a virtual sink, then we would start moving clock nets to be routed over general inter-block routing. In addition, if we allocated to a local network that cannot reach all logic blocks we would route excess fanout over local routing and alert the user.

To accompany this we would have to add placement constraints to constraint logic on the same clock domain into regions of the FPGA. For routable clocks like those of Stratix 10 and Ultrascale the constraints can be more relaxed to just tightly place logic on the same clock domain together in order to synthesize smaller clocks.

Enhancing the Placer and Router

We have seen from our clock results, Section 4.3 that we are increasing the critical path delay when routing clock nets using dedicated clock networks with our modified router. To recapture timing and some wiring we must update the placement delay estimator to better account for high-fanout low-skew clock nets. Currently, the placer delay model optimizes for the shortest delay from source to sinks. Additionally, updating the router criticality formulation to not penalize low-skew clocks because of their large insertion delay should also recover some of the small timing quality loss in the current formulation.

Better Area and Delay Modeling

To better evaluate clock networks in VPR it is necessary to create better delay modeling that properly estimates systematic skew. This requires addition of a distributed wire delay model to VPR. In addition, the clock network should have the minimum and maximum delays to account for manufacturing variation, as this directly affects the range of possible clock delays, which in turn directly affect the setup and hold times achieved by a design. VPR's timing analyzer models minimum and maximum delays, and the delays within blocks can have a different minimum and maximum delays in the VTR 8 trunk. However, routing wire and switch delays are currently a single number; they should be expanded to ranges in the future, particularly for clock networks.

Another area where better modeling will lead to better results is in calculating routing area. Currently, VPR computes the routing according to the minimum width transistor area. This assumes that FPGAs are transistor limited. While such an assumption can be defended for general inter-block routing we believe that the long wires of the clock network would cause area to be metal limited. Therefore, accounting for wire area would yield more accurate results for clock networks.

Bibliography

[1] Mohamed S Abdelfattah and Vaughn Betz. The Case for Embedded Networks on Chip on Field-Programmable Gate Arrays. *IEEE Micro*, 34(1):80–89, 2013.

[2] Mohamed S Abdelfattah, Andrew Bitar, and Vaughn Betz. Take the Highway: Design for Embedded NoCs on FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 98–107. ACM, 2015.

[3] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, page 4. ACM, 2014.

[4] Achronix. Speedster7t FPGAs Product Brief. *Achronix (PB033)*, 2018.

[5] Elias Ahmed and Jonathan Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):288–298, 2004.

[6] Altera. Stratix II Device Handbook, Volume 1. *Altera SII5V1-4.5*, 2011.

[7] Altera. Stratix IV Device Handbook, Volume 1. *Altera SIV5V1-4.8*, 2016.

[8] Altera. Stratix V Device Handbook, Volume 1. *Altera SV-5V1*, 2019.

[9] ABC Berkeley. A System for Sequential Synthesis and Verification, 2009.

[10] Vaughn Betz and Jonathan Rose. FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 59–68. ACM, 1999.

[11] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. Architecture and CAD for deep-submicron FPGAs. Kluwer Academic Publishers, 1999.

[12] István A Bogdán, Jenny Rivers, Robert J Beynon, and Daniel Coca. High-Performance Hardware Implementation of a Parallel Database Search Engine for Real-Time Peptide Mass Fingerprinting. *Bioinformatics*, 24(13):1498–1502, 2008.

[13] Bary Bray, Jean Paoli, C.M Sperberg-McQueen, Eve Maler, and Francois Yergeau. XML 1.0 Specificaiton (Fifth Edition). 2008.

[14] Luca P Carloni. From Latency-insensitive Design to Communication-based System-level Design. *Proceedings of the IEEE*, 2015.

[15] Luca P Carloni, Kenneth L McMillan, Alexander Saldanha, and Alberto L Sangiovanni-Vincentelli. A Methodology for Correct-by-Construction Latency Insensitive Design. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 309–315. IEEE Press, 1999.

[16] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 20(9):1059–1076, 2001.

[17] Luca P Carloni and Alberto L Sangiovanni-Vincentelli. Performance Analysis and Optimization of Latency Insensitive Systems. In *DAC*, 2000.

[18] Luca P Carloni and Alberto L Sangiovanni-Vincentelli. Coping with Latency in SoC Design. *MICRO*, 2002.

[19] Adrian M Caulfield et al. A Cloud-scale Acceleration Architecture. In *MICRO*, 2016.

[20] Gang Chen and Jason Cong. Simultaneous Timing Driven Clustering and Placement for FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 158–167. Springer, 2004.

[21] Charles Chiasson and Vaughn Betz. Should FPGAs Abandon the Pass-Gate? In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8. IEEE, 2013.

[22] Mark Cummings and Shinichiro Haruyama. FPGA in the Software Radio. *IEEE communications Magazine*, 37(2):108–112, 1999.

[23] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. 2004.

[24] Carl Ebeling, Dana How, David Lewis, and Herman Schmit. Stratix™ 10 High Performance Routable Clock Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 64–73. ACM, 2016.

[25] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[26] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx Adaptive Compute Acceleration Platform: Versal™ Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 84–93. ACM, 2019.

[27] Ilya Ganusov and Benjamin Devlin. Time-borrowing Platform in Xilinx UltraScale+ Family of FPGAs and MPSoCs. In *International Conference on Field-Programmable Logic and Applications*, 2016.

[28] Ron Ho, Kenneth W Mai, and Mark A Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[29] Dana L How and Sean Atsatt. Sectors: Divide & Conquer and Softwarization in the Design and Validation of the Stratix® 10 FPGA. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 119–126. IEEE, 2016.

[30] Yutian Huan and André DeHon. FPGA Optimized Packet-switched NoC Using Split and Merge Primitives. In *FPT*, 2012.

[31] Safeen Huda, Muntasir Mallick, and Jason H Anderson. Clock Gating Architectures for FPGA Power Reduction. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 112–118. IEEE, 2009.

[32] Intel. Avalon Interface Specifications (MNL-AVABUSREF). 2018.

[33] Intel. Intel® Stratix® 10 Clocking and PLLUser Guide. *Intel UG-S10CLKPL*, 2019.

[34] Intel FPGA. Hyper-pipelining for Stratix 10 Designs (AN715). 2015.

[35] Intel FPGA. Intel Arria 10 Product Table. 2017.

[36] Intel FPGA. Intel Quartus Prime Handbook (QPS5V1). 2018.

[37] Intel FPGA. Intel Stratix 10 Product Table. 2018.

[38] Peter Jamieson, Kenneth B Kent, Farnaz Gharibian, and Lesley Shannon. Odin ii - An Open-source Verilog HDL Synthesis Tool for CAD Research. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 149–156. IEEE, 2010.

[39] Xin Jia and Ranga Vemuri. The GAPLA: A Globally Asynchronous Locally Synchronous FPGA Architecture. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 291–292. IEEE, 2005.

[40] Nachiket Kapre and Jan Gray. Hoplite: Building Austere Overlay NoCs for FPGAs. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2015.

[41] Julien Lamoureux and Steven JE Wilton. FPGA Clock Network Architecture: Flexibility vs. Area and Power. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 101–108. ACM, 2006.

[42] Julien Lamoureux and Steven JE Wilton. Clock-aware Placement for FPGAs. In *2007 International Conference on Field Programmable Logic and Applications*, pages 124–131. IEEE, 2007.

[43] Guy Lemieux, Edmund Lee, Marvin Tom, and Anthony Yu. Directional and Single-Driver Wires in FPGA Interconnect. In *Proceedings. 2004 IEEE International Conference on Field-Programmable Technology (IEEE Cat. No. 04EX921)*, pages 41–48. IEEE, 2004.

[44] Guy Lemieux and David Lewis. Using Sparse Crossbars Within LUT. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 59–68. ACM, 2001.

[45] David Lewis, Elias Ahmed, Gregg Baeckler, Vaughn Betz, Mark Bourgeault, David Cashman, David Galloway, Mike Hutton, Chris Lane, Andy Lee, et al. The Stratix II Logic and Routing Architecture. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 14–20. ACM, 2005.

[46] David Lewis, Elias Ahmed, David Cashman, Tim Vanderhoek, Chris Lane, Andy Lee, and Philip Pan. Architectural Enhancements in Stratix-III™ and Stratix-IV™. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–42. ACM, 2009.

[47] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Sandy Marquardt, Cameron McClintock, Bruce Pedersen, Giles Powell, et al. The Stratix™ Routing and Logic Architecture. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 12–20. ACM, 2003.

[48] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. Architectural Enhancements in Stratix V™. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 147–156. ACM, 2013.

[49] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, Valavan Manohararajah, Ian Milton, Tim Vanderhoek, and John Van Dyken. The Stratix™ 10 Highly Pipelined FPGA Architecture. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 159–168. ACM, 2016.

[50] Tianqi Liu, Naveen Kumar Dumpala, and Russell Tessier. Hybrid Hard NoCs for Efficient FPGA Communication. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 157–164. IEEE, 2016.

[51] Jason Luu, Jason Helge Anderson, and Jonathan Scott Rose. Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 227–236. ACM, 2011.

[52] Jason Luu, Jeff Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Norrudin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(2):6:1–6:30, June 2014.

[53] Jason Luu, Conor McCullough, Sen Wang, Safeen Huda, Bo Yan, Charles Chiasson, Kenneth B Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. On Hard Adders and Carry Chains in FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 52–59. IEEE, 2014.

[54] Jason Luu, Jonathan Rose, and Jason Anderson. Towards Interconnect-Adaptive Packing for FPGAs. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 21–30. ACM, 2014.

[55] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density. In *FPGA*, volume 99, pages 37–46, 1999.

[56] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Speed and Area Tradeoffs in Cluster-Based FPGA Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):84–93, 2000.

[57] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-Driven Placement for FPGAs. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 203–213. ACM, 2000.

[58] M Imran Masud and Steven JE Wilton. A New Switch Block for Segmented FPGAs. In *International Workshop on Field Programmable Logic and Applications*, pages 274–281. Springer, 1999.

[59] Larry McMurchie and Carl Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Reconfigurable Computing*, pages 365–381. Elsevier, 2008.

[60] Nick Mehta. Xilinx UltraScale Architecture for High-Performance, Smarter Systems. *Xilinx White Paper WP434*, 2013.

[61] Kevin E Murray and Vaughn Betz. Quantifying the Cost and Benefit of Latency Insensitive Communication on FPGAs. In *FPGA*, 2014.

[62] Kevin E Murray and Vaughn Betz. HETRIS: Adaptive Floorplanning for Heterogeneous FPGAs. In *FPT*, 2015.

[63] Kevin E Murray, Andrea Suardi, Vaughn Betz, and George Constantinides. Calculated Risks: Quantifying Timing Error Probability with Extended Static Timing Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):719–732, 2018.

[64] Kevin E Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. Titan: Enabling Large and Complex Benchmarks in Academic CAD. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8. IEEE, 2013.

[65] Hong Shan Neoh and Asher Hazanchuk. Adaptive Edge Detection for Real-Time Video Processing using FPGAs. *Global Signal Processing*, 7(3):2–3, 2004.

[66] Michael Papamichael and James Hoe. CONNECT: Re-examining Conventional Wisdom for Designing NOCs in the Context of FPGAs. In *FPGA*, 2012.

[67] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[68] Rafat Rashid, J Gregory Steffan, and Vaughn Betz. Comparing Performance, Productivity and Scalability of the TILT Overlay Processor to OpenCL HLS. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 20–27. IEEE, 2014.

[69] Alex Rodionov, David Biancolin, and Jonathan Rose. Fine-grained Interconnect Synthesis. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 9(4):31, 2016.

[70] Alex Rodionov and Jonathan Rose. Automatic FPGA System and Interconnect Construction with Multicast and Customizable Topology. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 72–79. IEEE, 2015.

[71] Alex Rodionov and Jonathan Rose. Synchronization Constraints for Interconnect Synthesis. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 95–104. ACM, 2017.

[72] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. Architecture of Field-Programmable Gate Arrays. *Proceedings of the IEEE*, 81(7):1013–1029, 1993.

[73] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson. The VTR project: Architecture and CAD for FPGAs from Verilog to Routing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 77–86. ACM, 2012.

[74] Andrew Royal and Peter YK Cheung. Globally Asynchronous Locally Synchronous FPGA Architectures. In *International Conference on Field Programmable Logic and Applications*, pages 355–364. Springer, 2003.

[75] Raphael Y Rubin and André M DeHon. Timing-Driven Pathfinder Pathology and Remediation: Quantifying and Reducing Delay Noise in VPR-Pathfinder. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 173–176. ACM, 2011.

[76] Li Shang, Alireza S Kaviani, and Kusuma Bathala. Dynamic Power Consumption in Virtex™-II FPGA Family. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 157–164. ACM, 2002.

[77] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. Network-on-Chip Programmable Platform in Versal™ ACAP Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 212–221. ACM, 2019.

[78] Stephen M Trimberger. Three ages of fpgas: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103(3):318–331, 2015.

[79] Michael Wainberg and Vaughn Betz. Robust Optimization of Multiple Timing Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(12):1942–1953, 2015.

[80] Qiang Wang, Subodh Gupta, and Jason H Anderson. Clock Power Reduction for Virtex-5 FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 13–22. ACM, 2009.

[81] Neil H.E. Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective.* Pearson Education India, 2015.

[82] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys - A Free Verilog Synthesis Suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[83] Xilinx. Virtex-II Pro and Virtex-II Pro X FPGA User Guide. *Xilinx UG012 (v4.2)*, 2007.

[84] Xilinx. AXI Interconnect Referance Guide (UG761). 2011.

[85] Xilinx. UltraScale Architecture Clocking Resources. *Xilinx ug572 (v1.8)*, 2018.

[86] Xilinx. Vivado Design Suite User Guide (UG994). 2018.

[87] Sadegh Yazdanshenas and Vaughn Betz. Interconnect solutions for virtualized field-programmable gate arrays. *IEEE Access*, 6:10497–10507, 2018.

[88] Kai Zhu and DF Wong. Clock Skew Minimization During FPGA Placement. *IEEE transactions on computer-aided design of integrated circuits and systems*, 16(4):376–385, 1997.