Syncopation: An Adaptive Clock Management Technique for High-Level Synthesis Generated Circuits on FPGA

by

Kahlan Gibson

B.A.Sc., The University of British Columbia, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical & Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Syncopation: An Adaptive Clock Management Technique for High-Level Synthesis Generated Circuits on FPGA

submitted by Kahlan Gibson in partial fulfillment of the requirements for the degree of Master of Applied Science

in Electrical & Computer Engineering

Examining Committee:

Steve Wilton, Computer Engineering

Supervisor

Guy Lemieux, Computer Engineering

Supervisory Committee Member

André Ivanov, Computer Engineering

Supervisory Committee Member

Abstract

High-Level Synthesis (HLS) tools improve hardware designer productivity by enabling software design techniques to be used during hardware development. While HLS tools are effective at abstracting the complexity of hardware design away from the designer, producing high-performance HLS-generated circuits still generally requires awareness of hardware design principles. Designers must often understand and employ pragma statements at the software level or have the capability to make adjustments to the design in Register-Transfer Level (RTL) code.

Even with designer hardware expertise, the HLS-generated circuits can be limited by the algorithms themselves. For example, during the HLS flow the delay of paths can only be estimated, meaning the resulting circuit may suffer from unbalanced computational distribution across clock cycles. Since the maximum operating frequency of synchronous circuits is determined statically using the worst-case timing path, this may lead to circuits with reduced performance compared to circuits designed at a lower level of abstraction.

In this thesis, we address this limitation using *Syncopation*, a performance-boosting fine-grained timing analysis and adaptive clock management technique for HLS-generated circuits. Syncopation instrumentation is imple-

mented entirely in soft logic without requiring alterations to the HLS-synthesis toolchain or changes to the FPGA, and has been validated on real hardware. The key idea is to use the HLS scheduling information along with the placement and routing results to determine the worst-case timing path for individual clock cycles. By adjusting the clock period on a cycle-by-cycle basis, we can increase performance of an HLS-generated circuit. Our experiments show that Syncopation improve performance by 3.2% (geomean) across all benchmarks (up to 47%). In addition, by employing targeted synthesis techniques called Enhanced Synthesis along with Syncopation we can achieve 10.3% performance improvement (geomean) across all benchmarks (up to 50%).

Lay Summary

Specialized tools can reduce the time and effort to design circuits. However, these tools may produce slow circuits because some design decisions are based on estimates of path timing in the final design, which cannot be determined in advance. Typically, low circuit performance is due to a slow clock frequency, as the clock must be slow enough for the longest paths in the design. To improve the performance of circuits developed with these tools, we propose a method of recovering performance by adjusting the clock frequency in a fine-grained manner. This allows any design to operate faster without requiring additional compilation time for circuit designers.

Preface

The work presented in this thesis was originally published in part as a short paper in the 2020 International Conference on Field-Programmable Logic and Applications (FPL)[1]. Section 4.5 elaborates upon the Enhanced Synthesis technique, which was initially proposed in experiments conducted by E. Roorda. The implementation of Enhanced Synthesis presented in this thesis was extended, implemented, and evaluated by Kahlan Gibson.

All the research, implementation, and experimentation presented in this thesis was primarily conducted by Kahlan Gibson. This research was conducted under the supervision of Dr. S. Wilton, who provided editorial support for the publication. Additional inspiration, guidance, and editorial support was provided by D. H. Noronha, who co-authored the publication.

Syncopation is available as an open-source tool online at `https://github.com/kahlangibson/Syncopation`.

[1] K. Gibson, E. Roorda, D. H. Noronha, and S. Wilton. Syncopation: An Adaptive Clock Management Technique for HLS-Generated Circuits on FPGA. In *2020 30th International Conference on Field Programmable Logic and Applications (FPL)*, 2020.

Table of Contents

List of Tables

List of Figures

List of Programs

Acronyms

ACM Adaptive Clock Management.

ALM Adaptive Logic Module.

ASIC Application-Specific Integrated Circuit.

CAD Computer-Aided Design.

CGRA Course-Grained Reconfigurable Architecture.

CPU Central Processing Unit.

CSS Clock Skew Scheduling.

DM Divisor Memory.

ES Enhanced Synthesis.

FPGA Field-Programmable Gate Array.

FSM Finite State Machine.

GPP General Purpose Processor.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

HLS High-Level Synthesis.

LUT Lookup Table.

MCD Multiple Clock Domain.

PLL Phase-Locked Loop.

RTL Register-Transfer Level.

SDC Systems of Difference Constraints.

SDC Synopsis Design Constraints.

SM State Machine.

STA Static Timing Analysis.

Acknowledgements

First and foremost, thank you to my supervisor Steve Wilton, whose immeasurable support, patience, and teaching was inspirational throughout my tenure at UBC.

Special thanks to the members of my research group for their ideas, feedback, and comradery: Daniel Holanda Noronha, Jose Pinilla, Bahar Salehpour, Esther Roorda, Eddie Hung, and Nikhil Ganathe; and to the many others who were my colleagues, friends, mentors, and idols over the years: Dave Evans, Amin Azar, Khalid Essam, Xiaowei Ren, John Deppe, Mohamed Omran, William Yang, Deval Shah, Max Golub, Hossein Omidian, and Al-Shahna Jamal. Your advice, feedback, high-spirited chats, and lunch breaks have made these years a delight.

I would also like to thank the broader faculty of Electrical & Computer Engineering, who provided seven years of challenging learning and allowed me to discover my passion for teaching and my career.

A final round of thank-yous to my colleagues at Microsoft, who bolstered my professional growth by fostering an innovative and engaging culture over two fantastic summers.

Funding for this research was provided by Intel and the Natural Sciences and Engineering Research Council of Canada.

My deepest gratitude to my friends who have remained my greatest fans, critics, and fellow adventurers through it all, whether they were an editor for a day, a lab partner for months, or a confidante and companion for years.

To my parents, Allison and Randall, and Kyle: your foundation is an unwavering source of energy. I am lucky to have three of the most resilient people I know on my team, forever.

Chapter 1

Introduction

As Field-Programmable Gate Array (FPGA) architecture and Computer-Aided Design (CAD) tools evolve, designers are using FPGAs to implement larger and more complex circuits than ever before. Many of these circuits have tight timing requirements and achieving timing closure is time consuming and often requires extensive optimization. For expert hardware designers with familiarity with Register-Transfer Level (RTL)-based design flows, techniques such as pipelining and strategically imposing timing constraints can help achieve timing closure. Increasingly, non-domain expert designers are using High-Level Synthesis (HLS) tools such as LegUp, which automatically transform a software-oriented language (e.g. C Language) to hardware [10]. For these designers, optimization methods requiring hardware knowledge may not be feasible, either because the designer does not have the expertise or because the effort to effectively use these techniques defeats the purpose of accelerating the design flow using HLS. For this reason, recent work has focused on improving tools to automatically optimize circuit performance, power, and area in an attempt to achieve custom-implementation results without requiring additional designer expertise.

1.1 Motivation

In the HLS toolflow, RTL code is generated using *estimated* delays for individual paths. The RTL code is then synthesized by place and route tools which attempt to balance path lengths, often by giving preference to timing-critical nets. The degree of path balancing achievable by synthesis tools is limited without changing the scheduling decisions made by the HLS tool, leading to a longer-than-necessary clock period.

To demonstrate this and to motivate our technique, we synthesize several CHStone benchmark circuits [51] using the LegUp open source HLS tool [10] and use Intel Quartus Pro to perform placement, routing, and timing analysis. For each implementation, we profiled the design to determine the minimum clock period broken down by circuit state, and then determined the number of cycles across execution spent in each state. Fig. 1.1 shows the results of this experiment for the *adpcm* benchmark.

For the *adpcm* benchmark, the critical path delay is close to 15 ns. However, only a small proportion of execution time is spent in states which exercise this path. The vast majority of execution time is spent in states in which the longest exercised path is at least 30% faster. This means that although the operating frequency of this circuit would normally be estimated as 64 MHz, this circuit could operate with much lower latency if we could tune the clock on a cycle-by-cycle basis according to the requirements of the active state. This would effectively increase the operating frequency to 98 MHz.

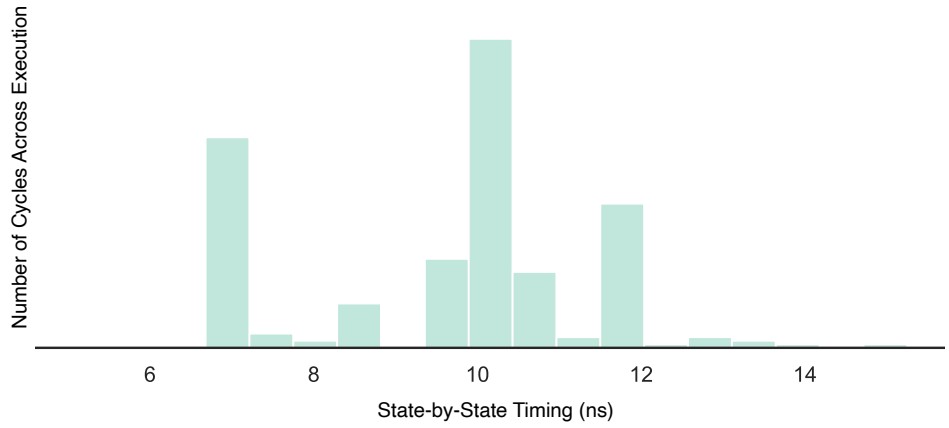

Figure 1.1: Number of Cycles Across Execution Spent in States with Various State Delays for the *adpcm* CHStone Benchmark

1.2 Key Idea

In this thesis, we present *Syncopation*, a technique to recover lost execution time due to path length variation after placement and routing[2]. The key idea is to use Adaptive Clock Management (ACM), in which the clock frequency is adjusted on a cycle-by-cycle basis based on the needs of the current circuit state. This is made possible by leveraging three observations. First, computational path delay variations exist across state machine states. Second, in any circuit, the output of all paths are not required in every cycle and only paths that compute scheduled instructions are required to meet timing in each circuit state. The clock period does not need to accommodate timing for paths which are not active in a given circuit state. Finally, HLS-generated circuits are particularly amenable to ACM

[2]In music, syncopation is the temporary displacement of a beat that disrupts the expected rhythm

techniques as HLS tools automatically generate instruction schedules which identify paths that are important for each state, therefore requiring no additional time to identify the relevant paths for each state. By combining the information from the detailed instruction schedules with Static Timing Analysis (STA) results for each path, we can identify the minimum clock period required on a *state-by-state* basis.

1.3 Contributions

This paper describes and evaluates our technique in combination with LegUp HLS and Intel Quartus Pro software. Specifically, we make the following contributions:

1. We implement a fine-grained adaptive clock using a dynamic clock generator based on [40] and a memory-based lookup controller system. The clock generator is capable of producing a glitch-free global clock with a different clock period in every cycle.

2. We propose a method for fine-grained static timing analysis. This method uses the schedule generated by the HLS tool to determine which values are computed in each circuit state, at the cost of using synthesis directives to prevent register optimization during place and route. The adaptive clock controller is programmed with path-length data and achieves a 3.2% increase in performance on average (up to 47%).

3. We investigate integrating the results of fine-grained STA with com-

mercial timing-driven place-and-route tools by automatically generating SDC files of timing constraints used in an additional synthesis pass to further enhance performance. We obtain a 10.3% increase in performance on average (up to 50%) without altering the place-and-route algorithms.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents relevant background on FPGA tools, including static timing analysis and high-level synthesis. Chapter 3 summarizes common techniques to improve performance in circuits, both in general-purpose processors and custom applications. Chapter 4 details our technique and describes the additional hardware required to implement the Syncopation clock management strategy. Chapter 5 presents experimental results which show the impact of applying Syncopation to a set of benchmark circuits.

Chapter 2

Background

Designers selecting a computing platform consider tradeoffs between programmability, performance, and cost. On one end of the spectrum, Central Processing Units (CPUs) offer a highly programmable platform capable of executing any software, but may suffer in performance due to (1) the lack of customized logic optimized for a specific application and (2) the assumption of a sequential compute paradigm. On the other end of the spectrum, fully-custom Application-Specific Integrated Circuits (ASICs) are not generally programmable, and instead are designed to maximize performance (minimize latency, maximize throughput) to complete a specific task. For low-to-medium volume applications, the cost of using a device fabricated at scale, such as a CPU, is significantly lower than the costs of developing a custom-designed ASIC. At these lower volumes, the non-recurring ASIC development costs (including engineering development, fabrication, and validation) make fully-custom solutions economically unfavourable. In high-volume applications, the cost of ASIC development is offset by the volume produced and sold, making fully-custom applications more appealing.

In applications which require higher performance at volumes not sufficient to offset the initial design costs, FPGAs provide a platform to develop

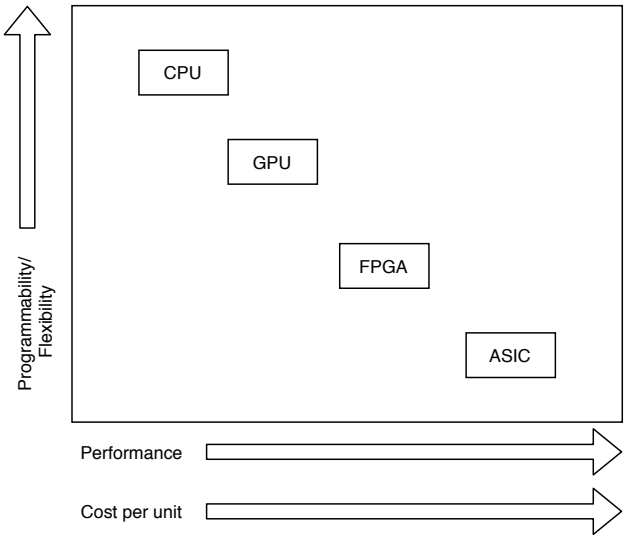

Figure 2.1: The Performance-Programmability Computing Landscape at Low Volumes.

custom solutions at a lower price-point. FPGAs are highly programmable arrays of gates which are mass fabricated, eliminating one-time engineering costs. Since FPGAs are programmed to implement circuits at the gate level and can be reprogrammed for testing without waiting for fabrication, FPGAs offer hardware designers a unique opportunity to quickly design high performance accelerators at a lower cost compared to ASICs. Due to these advantages, FPGAs have become a common technology in consumer devices and high performance applications in big data, networking, and artificial intelligence. In recent years, FPGA instances have become a standard feature in major cloud computing infrastructures (including Amazon AWS [6] and Microsoft Azure [41]), such as the latest 10nm process Intel Agilex FPGAs designed specifically for datacenter use [26]. Widespread utilization of FPGAs has been bolstered by the improvement of development environments

and tools such as HLS, which strives to make programming FPGAs accessible to designers without hardware design experience.

Fig. 2.1 describes the computational platform landscape, including Graphics Processing Units (GPUs), which are not discussed further here. CPUs occupy the low-cost, low-performance, high-programmability corner of the landscape. ASICs occupy the high-cost, high-performance, low-programmability corner. FPGAs combine these characteristics into a package which provides specialized compute desired for high performance applications at a lower cost than ASIC development and in a platform that can be reprogrammed when needed.

FPGAs development and implementation quality are highly dependent on the tools and methodology provided by commercial vendors. Intel [2] and Xilinx [5] have the largest market share, followed by a small number of other companies including Achronix Semiconductor [1], Lattice Semiconductor [3], and Microsemi Corporation [4]. Each company's FPGAs are programmed using a tightly coupled closed-source toolflow. The toolflows use intimate knowledge of the FPGA architecture as well as circuit timing information to synthesize a designer's description of a circuit into a gate-level implementation to generate a bitstream which is used to populate memories, implement logic, and connect specified wires on the FPGA device. In addition to realizing the designer's circuit description using device resources, synthesis tools are responsible for fine-tuning the circuit implementation. Given a behavioural description of a circuit, the synthesis tool should implement the circuit on the FPGA device to minimize area, latency, and power utilization. Since there are many ways to implement a given circuit on FPGA resources,

determining the best implementation is a complicated problem.

The complexity of determining a high-quality implementation is further complicated when designers choose to use HLS tools. HLS tools enable designers to describe the operation of their circuit using a high-level software language, enabling faster development and abstracting away the hardware-level details. However, the high-level description does not explicitly define the circuit operation and leaves even more implementation details to the tools to determine. The algorithms which transform high-level software languages into circuits may be heuristic or stochastic in nature to encourage design space exploration, and significant research effort has been expended to improve the quality of HLS-generated circuits while minimizing the hardware knowledge required by designers.

This chapter presents the background information relevant to this thesis. It begins with an overview of static timing analysis and how timing information is used to guide circuit implementation in synthesis tools. Then, it discusses high-level synthesis tools and how they utilize how timing information to generate hardware descriptions from software algorithms.

2.1 Static Timing Analysis

FPGA devices consist of an array of programmable "blocks" which can be wired together to form an arbitrary circuit. The wiring between blocks consists of a routing "fabric" of switch blocks to connect required gates together. The blocks on an FPGA include hardened arithmetic units, I/O blocks, on-chip memories, routing switch blocks, and generic programmable

blocks (Adaptive Logic Modules (ALMs) in the Intel vernacular [24]).

To design circuits using an FPGA, hardware designers write Hardware Description Language (HDL) code (such as Verilog) which describes circuit behaviour. Then, specialized vendor tools (such as Intel's Quartus) synthesize the code into a circuit implementation which maps onto the FPGA resources. Synthesis consists of multiple stages, including netlist generation, technology mapping, clustering, placement, and routing, as shown in Fig. 2.2. During netlist generation, the Verilog is transformed into a gate-level description of the circuit. Then, technology mapping transforms groups of gates in the netlist into components available on the FPGA, such as N-input Lookup Tables (LUTs) (N-LUTs). Then, the N-LUTs are clustered to encourage highly connected sections of the circuit to be within the same logic block cluster on the chip (to reduce latency and routing congestion). Finally, the N-LUT contents are assigned to specific ALMs on the FPGA, and routing is performed to make connections between the ALMs.

To evaluate the performance of a circuit implementation, STA measures each path in the circuit to determine whether it meets timing requirements. Additionally, the longest path (critical path) is recorded to determine the maximum operating frequency, since the clock period must be long enough to accommodate the timing of the longest path. The clock frequency determined during STA determines the performance of the design, as higher frequency circuits have lower latency. STA must be performed after synthesis for an accurate result, as the specific implementation produced by the tool directly impacts the critical path delay.

Although STA identifies the critical path of the post-synthesis circuit, it

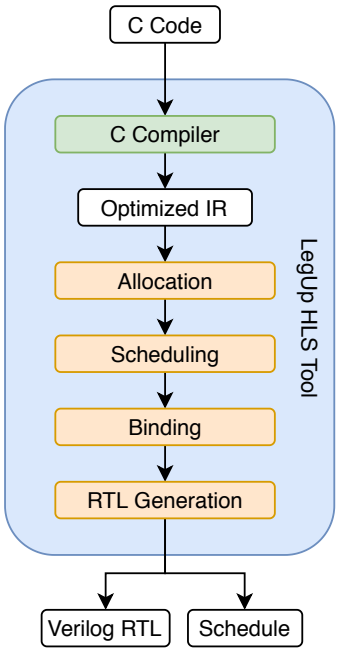

Figure 2.2: The LegUp high-level synthesis flow.

is desirable to use timing information throughout synthesis. This technique is referred to as Timing-Driven Synthesis, which incorporates timing information into the algorithms to continuously perform optimizations aimed at reducing the critical path of the circuit. A simple example of a non-timing-driven clustering algorithm is VPack [7], which attempts to efficiently cluster the design to have high utilization of physical resources while minimizing signal latency. The VPack clustering algorithm attempts to cluster ALMs to meet the following goals:

1. Minimize the total number of ALM clusters used on chip, to encourage high cluster utilization.

2. Minimize the number of inter-cluster connections used in the design,

since these wires have longer latency.

In VPack, an unclustered ALM in the design with the most inputs is selected as the "seed", or first ALM, in a cluster. Then, ALMs with the most shared inputs to the current cluster population are iteratively added to the cluster until no further packing into the cluster is possible.

On the other hand, a timing-driven clustering algorithm, such as T-VPack [37], prioritizes clustering of the critical path by selecting an unclustered ALM on the critical path with the most inputs as the seed for the cluster. Additionally, T-VPack uses more complex selection criteria when adding to the cluster. In particular, ALMs will prioritized for cluster addition if they (1) share many inputs with the current population, (2) are close in connectivity to the critical path, and (3) are involved in multiple timing-critical paths. Adding timing awareness to clustering incentivizes the implementation of solutions for which the critical path spans the fewest number of clusters.

Another target for timing-driven algorithms is to attempt to balance path-lengths such that the amount of compute taking place in each clock period fully utilizes the time available in each cycle. However, there are limits on how even the path lengths may be due to inherent limitations of the tools and FPGA architectures. First, the circuit description is explicitly defined at the RTL level, and significant changes to when computations are being performed are not desirable. For example, the synthesis tools are not given the ability to broadly reschedule large numbers of operations in order to improve circuit performance. For this reason, the first performance

optimizations are typically performed by hardware designers at the RTL level. Second, the granularity of synchronization elements and delay differences in paths through elements such as routing switch blocks on FPGAs makes producing precise path lengths difficult. While ASIC designers may manually lay out gates and wires to achieve very similar path delays, the synthesis tools for FPGAs are limited by the availability of the remaining resources to achieve matching path lengths. Finally, synthesis, place and route, and STA are time-consuming, and calculating timing for all paths would cause intractable increases in compilation time. To reduce synthesis run-time, tools use timing estimates and focus optimizations primarily on the few most-critical paths, unless specifically incentivized to reduce timing for all paths.

2.2 High-Level Synthesis

HLS is the process of converting high-level software language code into a circuit which implements the described algorithm. HLS tools reduce the hardware knowledge required to use FPGAs as general acceleration platforms, increases designer productivity, and boosts debug productivity. However, the level of abstraction that makes HLS so attractive also removes much of the control designers have over the quality of the final circuit implementation and may lead to longer critical paths compared to what may be possible with RTL-oriented design techniques. Details such as the schedule of instructions and computations in the datapath are removed from the designer's control and are left to the discretion of the HLS tool, unless the designer has suffi-

cient experience to change the code at the Verilog level. There are multiple stages to the HLS toolflow, all of which have unique opportunities to influence the final outcome of the synthesis flow.

At the input to the HLS flow, depicted in Fig. 2.2, the tool takes the high-level behavioural description of the algorithm and information about available hardware resources to generate RTL which describes the software. To generate the RTL, the tool generally completes the following steps:

1. Compilation of the high-level code to a formal representation, including basic optimizations
2. Hardware resource allocation
3. Instruction Scheduling
4. Binding computations to functional units (multipliers, dividers)
5. Binding variables to storage elements (registers or memories)
6. Binding data transfers to buses
7. Generate RTL description

In each of these steps, there are opportunities to improve circuit performance. In particular, the scheduling of instructions can directly influence the critical path length. The challenge is that, instruction scheduling at the HLS level can only use rough timing estimates to inform decisions, as no information about the final implementation is available yet.

Consider the instructions provided in Program 2.1. An approximate delay for each computation is provided as an example. There are multiple

Program 2.1 A simple example program written in an assembly-like language.

```
main:
1   add r3, r1, r2   # add: delay 2
2   sub r4, r3, r1   # sub: delay 2
3   add r5, r4, r2   # add: delay 2
4   sub r6, r1, r5   # sub: delay 2
5   mul r7, r3, r6   # mul: delay 10
6   mul r8, r5, r6   # mul: delay 10
7   add r9, r7, r8   # add: delay 2
8   add r0, r9, r1   # add: delay 2
```

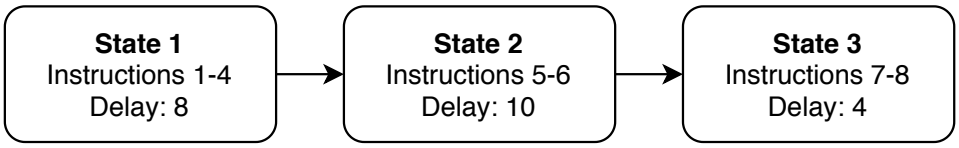

Figure 2.3: Program 2.1, scheduled into states.

ways to schedule these computations into states. We assume this code segment is part of a larger algorithm, and minimizing the critical path length of this code segment is critical for circuit performance. There are two goals to consider scheduling these instructions. First, we would like to minimize the longest path in a single state. Since multiplication is typically slower than addition, we are limited by the delay of the multiply instructions on lines 5-6, which can be parallelized. No other instructions can be added to this state without increasing the path length. Second, we would like to minimize the total number of states. We can combine instructions 1-4 and instructions 7-8 into two other states without increasing the critical path length. The proposed schedule is shown in Fig. 2.3.

Although Fig. 2.3 meets our scheduling goals, we still have path length

variation between our states, resulting in 27% of the program's execution time being unutilized due to short computational paths. Although this example is simple, it is clear how the scheduling problem is complicated for HLS tools to solve without significant designer effort either at the RTL level or through the use of compiler directives. ACM techniques forgo the need for this extra effort and can reduce the underutilized compute time.

Chapter 3

Related Work

At the RTL level, hardware designers can carefully tune circuits to maximize circuit operating frequency without introducing erroneous computation. However, modern synthesis tools have been designed to automatically perform circuit tuning to make hardware design faster and more accessible. In this section, we introduce multiple approaches in recent work to automate improved circuit performance through multiple stages of the synthesis toolflow, including circuit-level optimizations, HLS pass optimizations, and clocking schemes.

3.1 Pipelining and Retiming

An RTL description of a circuit describes operation on a state-by-state basis. Operations occurring in each state are scheduled such that data dependencies and timing requirements are satisfied. When these scheduling decisions are made by a hardware designer writing RTL, the designer can carefully optimize the code to achieve a target performance. Once the RTL is written, there are few opportunities for automated tools to further optimize the circuit implementation since they lack knowledge of the overall operation.

However, pipelining and retiming can be automatically employed in syn-

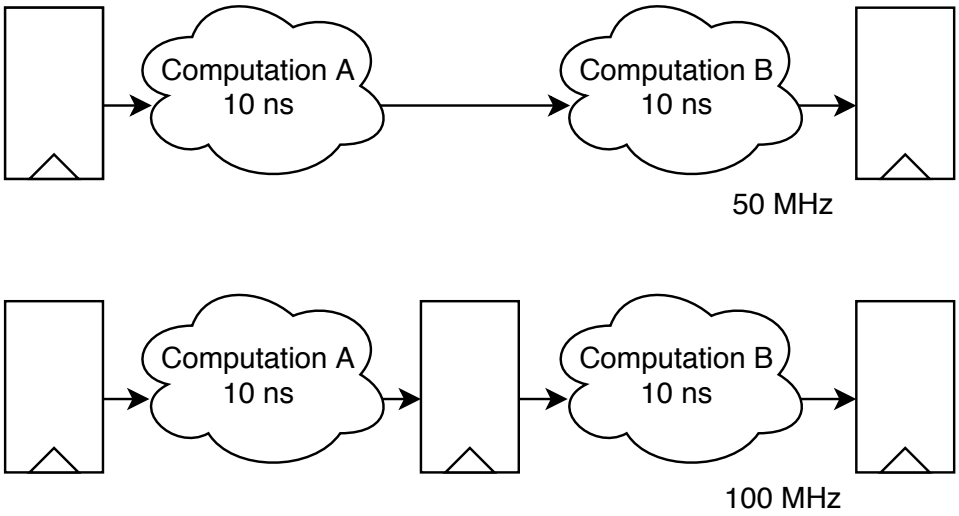

Figure 3.1: Pipelining some computations to balance path delays.

thesis tools as they often do not require significant changes to the schedule of operations as described in the RTL. Fig. 3.1 and Fig. 3.2 depict examples of pipelining and retiming simple circuits. Propagation delays have been arbitrarily assigned between the registered stages indicated as flip-flops, which could be due to scheduled computations or data transfers over wires. In Fig. 3.1, there are two combinational components (labelled A and B) which are connected sequentially. Each component takes some amount of computational time to produce a result. In this case, the compute times are 10 ns each, for a total path delay of 20 ns. If this path is the critical path of the circuit, the maximum frequency the circuit can operate at is 50 MHz. Pipelining can increase the maximum frequency by adding an additional register stage to break up this path, as shown in the second path in Fig. 3.1. In this circuit, the path has been broken up by an additional register into two

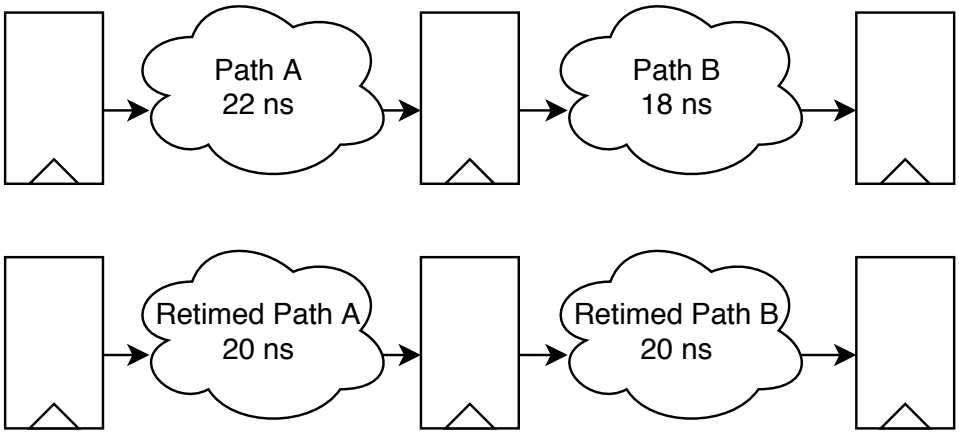

Figure 3.2: Retiming some computations to balance path delays.

paths with 10 ns delays, increasing the maximum frequency of the circuit to 100 MHz.

In Fig. 3.2, retiming is performed to balance out paths and reduce the critical path length without adding additional registers. In this circuit, there are two paths consisting of combinational logic seperated by a register. Path A is the critical path of the circuit with a delay of 22 ns, and Path B has a shorter delay of 18 ns. The maximum operating frequency due to Path A is 45 MHz. It is possible that some of the computation that is performed in Path A could be moved to Path B to improve circuit performance. In Fig. 3.2, retiming is performed to balance Path A and Path B. The new paths now have delays of 20 ns each, and the new maximum circuit frequency is 50 MHz.

Modern FPGA tools support automated pipelining and retiming in synthesis. For example, Intel's Hyperflex Architecture in the top-of-the-line Stratix 10 and Agilex devices and Quartus Pro allow users to enable au-

tomated pipelining and retiming [24]. Although leveraging the fine-grained registered architecture in these devices can improve circuit performance, finding appropriate paths can increase synthesis time due to added complexity. Furthermore, not all circuits are amenable to these optimizations at the synthesis stage. For HLS-generated circuits in particular, academic research has proposed leveraging these device architectures to improve timing. In particular, [12] demonstrated improved retiming techniques in Intel Hyperflex devices. Additionally, changes to the loop pipelining algorithms in HLS tools have been shown to increase circuit throughput [15]. These methods require changes to HLS algorithms, and may increase complexity of the tools or increase compilation time. Unlike previous work, we do not propose changes to the HLS or commercial synthesis algorithms, and Syncopation does not rely on a specific FPGA architecture. Instead, we exploit the path imbalances which persist in HLS-generated circuits despite these algorithmic improvements. In fact, Syncopation could be used to further improve previous work by exposing slack that was not recovered through pipelining and retiming.

3.1.1 Clock Skew Scheduling

Clock Skew Scheduling (CSS) is a technique for achieving performance-boosting path balancing comparable to retiming, but is implemented by intentionally inserting clock skew instead of physically moving logic between flip-flops [19]. CSS has also specifically been applied to FPGAs to improve performance [47]. FPGA architecture changes have also been proposed to benefit performance of circuits [16].

3.2 HLS Scheduling

In high-level synthesis, scheduling is the process of mapping the software-level operations into clock steps. Scheduling is a complicated optimization problem typically solved with heuristic algorithms, which may not result in the highest-performance result. Furthermore, since the detailed path-timing information used to inform scheduling decisions is not known until after place-and-route is complete, HLS tools use timing estimates to assign software instructions to computational states. When determining the instruction schedule, the HLS tool attempts to maximize performance by balancing path lengths across cycles and minimize power utilization and circuit area by efficiently utilizing on-chip resources.

The open source HLS tool LegUp 4.0 uses a Systems of Difference Constraints (SDC)-based scheduling algorithm [34]. SDC-based algorithms find solutions to problems such as scheduling by solving a mathematical formulation of the constrained system. Multiple scheduling algorithms have been proposed in academic research which improve upon the LegUp SDC-based scheduler. In [22], instruction-level data dependency graphs were utilized to improve scheduling across basic blocks to reduce circuit latency. Other scheduling algorithms have attempted to optimize for circuit area and congestion, such as in [17], by constraining the scheduler by both latency and resource usage. Stochastic optimization methods which use randomisation to perform broad design space exploration have also been proposed. These algorithms, such as simulated annealing (proposed in [39]) and genetic algorithms (proposed in [44]) explore the multi-dimensional design spaces to

determine schedules which meet area, delay, and power requirements.

3.2.1 Multi-Cycle Operation

Another technique that has been applied to improve the performance of HLS circuits is multi-cycling. Multi-cycling is a technique which allows some paths within a circuit to have longer path delays than the clock period and allowing those paths more than one clock cycle to complete execution. HLS tools are a particularly suitable for multi-cycling as the high-level algorithm and scheduling analysis occurring during HLS reveals which computational results are not required in the next clock cycle, and therefore may benefit from multi-cycling techniques. Furthermore, the HLS scheduling algorithms can be altered to automatically employ multi-cycling for improved performance [23, 52].

3.2.2 Dynamic Scheduling

Taken to an extreme, multi-cycling leads to a dynamic synchronous dataflow-like architecture incorporating handshake protocols to signal the completion of individual operations. Dynamic scheduling algorithms have been explored for HLS circuits to gain additional performance. Typically, HLS tools schedule instructions statically, meaning that even if the time required for an operation to complete is data-dependent, the worst-case timing estimate of that operation is assumed. Static scheduling techniques suffer performance degradation as the worst-case timing estimate is conservative in the typical case. To overcome this limitation, dynamic HLS scheduling has been proposed [13, 31, 32]. Dynamic scheduling prevents variable-latency operations

from blocking future computations at the overhead of additional circuit area for handshaking protocols (2.48x LUTs [13]). To mitigate the area cost, the authors proposed a hybrid static-dynamic scheme which dynamically schedules sub-circuits with the most improvement due to dynamic scheduling (1.52x LUTs [13]). Another technique for improving software execution is speculative execution, a performance-boosting technique in which a processor will predict what to execute when a data-dependent branch is reached. In [33], data-dependent branch operations are speculatively computed in parallel ahead of the branch resolution to achieve additional performance. Utilizing speculative execution reduces latency up to 22.4%, but the additional resources hardware to perform the predictions and computations increases circuit area up to 19.7% [33].

Syncopation does not require changes to the HLS scheduling algorithms, and could be implemented in conjunction with existing scheduling algorithms to further improve performance. For example, a hybrid dynamic-static scheduling paradigm such as in [13] could employ Syncopation in statically-scheduled sub-circuits which would otherwise not see performance improvement from dynamic scheduling. Additionally, Syncopation can be made compatible with speculative execution techniques, data-dependent dynamic scheduling, and strategic multi-cycling for more aggressive performance enhancements.

3.3 Overclocking

Overclocking is a method of speculatively increasing the clock frequency of a circuit to achieve higher performance. Overclocking is a speculative technique because increasing the clock frequency beyond the maximum frequency computed during static timing analysis may produce erroneous results. Overclocking is widely employed to increase the operating speed of CPUs and GPUs by increasing the base clock multiplier or front side bus overclocking [49]. In online dynamic overclocking, the clock frequency is dynamically tuned while the circuit is operating while monitoring for errors to either ensure none have occurred or to keep the number of timing faults below an acceptable threshold for the application. In [48], online dynamic overclocking is combined with an error detection mechanism to prevent faults in superscalar CPUs. Razor [18] is another technique which enables overclocking by using Razor flip-flops, which automatically identify timing faults by double-sampling computational paths.

Overclocking is also employed to improve performance of FPGA circuits, both with and without automated error detection mechanisms. In [45], the authors propose using overclocking as an alternative performance-boosting method to reduced precision computing, as both methods increase performance at the cost of some acceptable threshold of accuracy loss. In applications which require no accuracy loss, automated techniques to perform overclocking have been proposed for arbitrary circuits [8][42] and algorithm-level error detection in convolutional neural networks [38]. Others have proposed changes to datapath architecture to reduce the severity of timing-

induced failures to enable aggressive overclocking [46]. Additionally, work has been conducted to adapt the Razor timing speculation technique for paths with bidirectional communication, including computational arrays, Course-Grained Reconfigurable Architectures (CGRAs), and FPGAs [9].

Syncopation is not a dynamic overclocking technique, and does not require error detection hardware. We perform fine-grained STA for each state in the HLS schedule, and select a clock frequency based on the result. Since the clock frequency is not set higher than the provided value, there is no risk of corrupting the data. However, online adaptive overclocking techniques could be used in combination with Syncopation to further increase performance.

3.4 Multiple Clock Domains

Using Multiple Clock Domains (MCDs) is a method to improve performance by operating portions of a circuit at a higher frequency, dictated by the sub-circuit critical path. MCDs require the addition of clock domain crossing hardware to prevent corruption of data sampled during a period of signal metastability. A circuit that would benefit the most from using MCDs is one which spends a significant proportion of run-time performing computations with paths much shorter than the critical path. Multiple clock domains for HLS-generated circuits has been proposed by partitioning the clock domains on boundaries of hardware modules [43]. This minimizes the complexity of the boundary crossing hardware (due to fewer signals passing data between software functions than within them) and improved wall-clock time by 33%

at an additional hardware cost (7-9%) [43]. To improve the performance of this technique, designers need to partition the software-level code intelligently. For example, the code could be partitioned so that operations with long computational paths are separated from functions which dominate program execution time, so that the amount of time operating at a higher frequency is maximized. However, since operation latency is not a typical concern for software developers, the authors of [36] proposed an automated framework for optimizing MCD boundaries based on dataflow graph representations of the source code. In addition to using multiple clock domains to achieve higher performance in some applications, MCDs can be used to slow down some domains in power-sensitive applications. In [35], an automated multi-clock domain scheme is used to slow down some parts of the design to reduce clock network complexity.

Syncopation does not use multiple clock frequencies distributed spatially across the design, as is the case with MCD. Instead, Syncopation uses the global clock routing resources to distribute a single clock signal which varies in frequency temporally. Syncopation could be used in conjunction with MCD techniques to further boost HLS-generated circuit performance. While using multiple clock domains would be very effective if one module was limiting the performance of all other modules, the performance benefit would be limited in cases where all the modules are limited by a similar critical path. In these cases, Syncopation could be used to control the clock in domains modules which did not benefit from a large increase in operating frequency.

3.5 Adaptive Clock Management

ACM is the dynamic adjustment of processor clock period on a per-cycle granularity based on in-flight instructions. ACM has primarily been targeted at General Purpose Processors (GPPs) such as CPUs and GPUs [14][30] [29]. In general, ACM works by adjusting the clock period dynamically based on the operation code (op-code) of the instructions in-flight, as the op-code indicates which functional units will be operating. Operations with longer critical paths through the functional units will be given longer clock periods to complete operation. In this application, ACM must be applied dynamically due to out-of-order execution of instructions employed in GPPs.

ACM has also been proposed for FPGA-based soft-processors. To achieve adjustable clock period, previous work ([20]) has proposed a hybrid clock multiplexing and clock stretching technique called Hybrid Adaptive Clock Management (HACM). HACM amortizes the long switching delays associated with clock multiplexing by incorporating clock stretching to recover additional slack.

Syncopation is different than previous ACM techniques as it is applicable to arbitrary circuits synthesized with HLS tools, whereas previous ACM techniques relied upon on known delays through GPP functional units and knowledge of in-flight instructions. Applying Syncopation ACM to arbitrary circuits is made possible by using fine-grained timing analysis of each state's path in the state machine at synthesis time. While Syncopation could be extended to arbitrary FPGA circuits not generated by HLS (expanded upon in Sect. 5.4.4), our technique currently relies upon the static schedule of

instructions already produced by HLS tools. Furthermore, the Syncopation clock generator avoids long clock switching times of clock multiplexing and provides greater dynamic range than clock stretching techniques by using a high-frequency clock divider.

Chapter 4

Syncopation

4.1 Overview

Syncopation is an adaptive clock management strategy which improves the performance of FPGA circuits. The key observations which make Syncopation possible are (1) not all circuit paths are active in every clock cycle, (2) the clock period only needs to be long enough to accommodate the computational delay of the active paths, and (3) HLS-generated circuits are particularly amenable to ACM techniques as HLS tools automatically generate instruction schedules which reveal the active paths in each clock cycle of the hardware state machine. By combining these observations with post-place-and-route static timing analysis and using a small amount of additional instrumentation, Syncopation tunes the clock period on a cycle-by-cycle basis to reduce latency and boost performance.

To demonstrate Syncopation, we use the Finite State Machine (FSM) in Fig. 4.1. The Syncopation clock generator is a clock divider which receives an integer divisor every cycle which corresponds to the desired clock period. Assuming a 500 MHz reference clock (selection discussed in Sect. 4.3), we can use the HLS information to determine the minimum integer divisor based on the active path delays within each state; each state in the Fig. 4.1

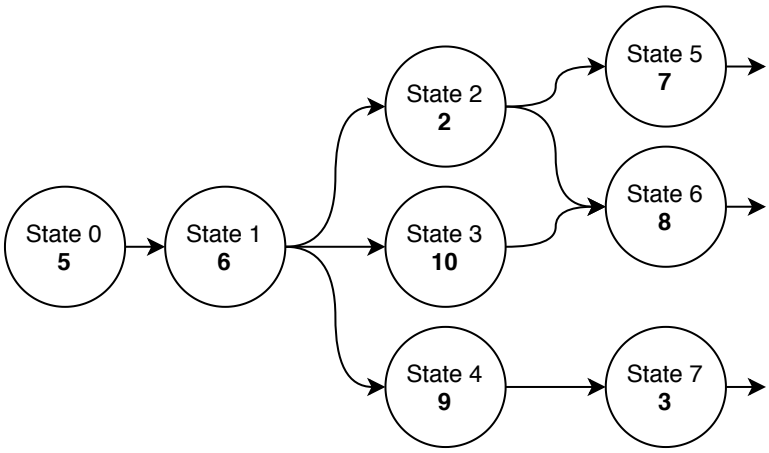

Figure 4.1: A simple Finite State Machine.

is annotated with this divisor. The fastest state in this circuit has a divisor of ten and the slowest has a divisor of three; these states can operate at 167 MHz and 50 MHz. Using a single-frequency clock this circuit would operate at 50 MHz for the entirety of its operation. However, Syncopation effectively increases the clock frequency by enabling fine-grained adjustment of the clock period to accommodate only the active computational paths.

While Syncopation does not require changes to the HLS or place-and-route algorithms, the per-state timing information can be used to target synthesis tool effort in timing-driven place-and-route to further boost performance.

4.2 Overall Flow

The Syncopation flow is shown in Fig. 4.2. First, LegUp [10] transforms each function in the software description into an FSM. An algorithm consisting

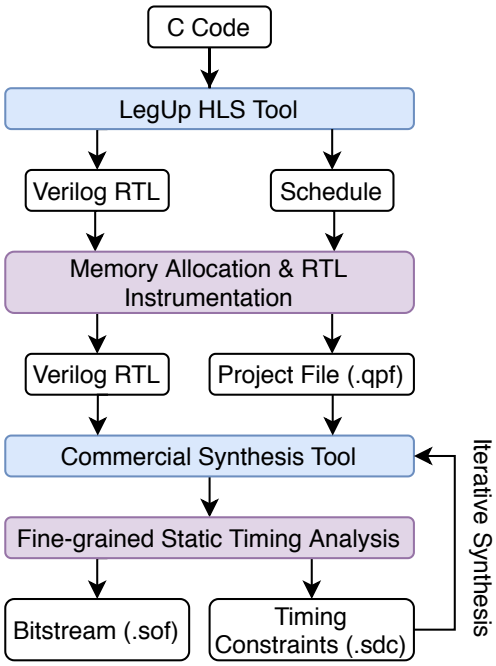

Figure 4.2: Syncopation overall flow.

of multiple software functions will result in a circuit of multiple FSMs. The instruction schedule describing the computations performed in each state is also generated in this step.

Next, Syncopation clock generator circuitry (described in Sect. 4.3) is inserted into the LegUp-generated RTL and a Divisor Memory (DM) is allocated for each FSM. DM allocation and operation is described in Sect. 4.3.2. In addition, synthesis directives are inserted into the RTL to enable *fine-grained static timing analysis* (details in Sect. 4.4).

Once the Syncopation hardware has been inserted, the design is synthesized using commercial tools (we use Quartus Pro 15.0). Fine-grained STA is performed using the post-place-and-route implementation to determine

circuit timing on a per-path basis. The measured path lengths are then used in conjunction with the HLS schedule to populate the DMs, described in Sect. 4.4. The DM contents can be updated without re-synthesizing the design.

Finally, and optionally, a fined-grained timing constraints file is created and used to improve performance in successive synthesis passes, which we call Enhanced Synthesis (ES). ES is described in Sect. 4.5.

4.3 Adaptive Clock Management Circuitry

The Syncopation ACM circuitry is inserted into the user circuit RTL before place-and-route. Consisting of the DMs, divisor selection logic, and the clock generator, the Syncopation circuitry monitors the user circuit and tunes the clock period according to the circuit state. In this section, the Syncopation circuitry is described in detail.

4.3.1 Circuit Operation

A simple Syncopation circuit is depicted in Fig. 4.3. The LegUp-generated state machine is the user circuit, and the Divisor Memory, Divisor Selection Logic, and Clock Generator comprise the Syncopation circuitry.

During circuit operation, the state signals from each state machine are used to address the corresponding DM. The DMs are populated with divisor values. The values obtained from the memory reads are propagated to the Divisor Selection Logic which determines the longest active path in the design across all the state machines and selects the corresponding divisor to

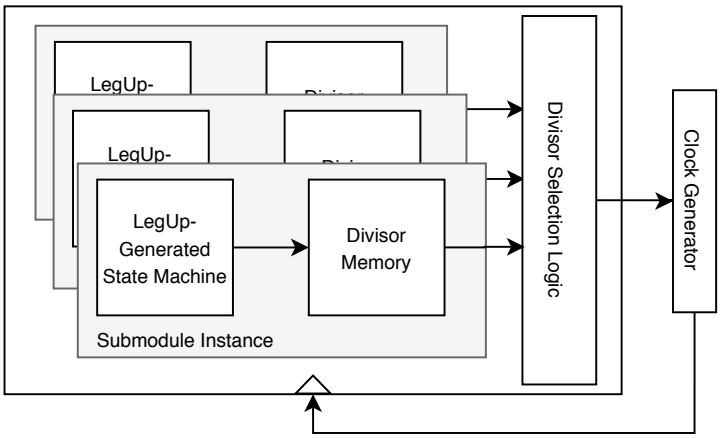

Figure 4.3: A Syncopation Circuit.

tune the Clock Generator.

To ensure stability of our clock signal, we require that the clock divisor is set before the beginning of the next clock cycle. Since the divisor memories are registered, the entire clock tuning procedure requires three clock cycles: one to initiate the DM read, one to select the divisor value, and a third in which the clock is tuned using that value. This procedure implies that the circuit state is known two cycles in advance, which is not possible in designs with branch instructions. To successfully implement the divisor lookup system, we propose two architectures: one which performs this lookup by monitoring the *Current State* of the user circuit, and another which monitors the *Next State* of the circuit. In our studies, we implemented and tested both of these configurations.

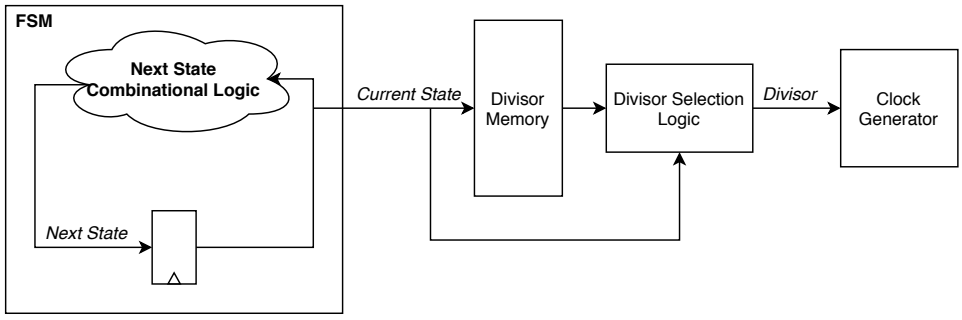

Figure 4.4: Current State based divisor selection logic.

Current State Syncopation

In the Current State implementation of Syncopation, the *current state* signal of each state machine is connected to the DM address port and the Divisor Selection Logic, as shown in Fig. 4.4. Electing to use the current state signal to monitor the user circuit means the exact circuit state cannot be determined in advance. To ensure that the clock is tuned to accommodate the longest path, the *divisor* values packed into the DM correspond to the slowest possible state that the state machine may be in, given the current state. This is determined by inspecting the HLS-generated state machine. The DM in Fig. 4.5 depicts the DM contents for the example state machine in Fig. 4.1. Each address corresponds to a state, and the corresponding line contains the *tags* and *divisors* used to tune the clock generator. Note that this implementation of Syncopation does not require storing all the state divisors in the DMs due to the selection of the maximum divisor value. The maximum divisor is determined offline through inspection of the state machine and the results from static timing analysis. In this example, the divisor values for States 1, 2, 4, and 5 are not present in the DM.

Address	Current State Tags			Next State Divisors		
0			1			10
1	2	3	4	8	8	3
2		5	6			
3			6			
4			7			
...	...			...		

Figure 4.5: Current State divisor memory packing based on the Finite State Machine in Fig. 4.1.

The maximum divisor selection results in slower clock period (on average) compared to the ideal operation. To demonstrate operation of Current State Syncopation, consider the example state machine in Fig. 4.1. In this example, we will walk through how Current State Syncopation uses the state *tags* and *divisors* from the DM to tune the clock period for Cycle 3 and Cycle 4.

To determine the divisor value for Cycle 3, the memory read to line 0 is initiated in Cycle 1 and the divisor is selected and sent to the clock generator in Cycle 2. In Cycle 2, the current state is State 1, and the next state divisors are 2, 9, and 10. Since we must accommodate for the longest computational path for Cycle 3, the divisor value selected to send to the clock generator is 10. Finally, in Cycle 3, the clock period is tuned to have a clock period corresponding to a divisor value of 10. However, the computational path would not utilize the entire clock period unless the circuit is in State 3 in Cycle 3. This corresponds to a longer-than required clock period compared to an implementation capable of determining exactly the required divisor

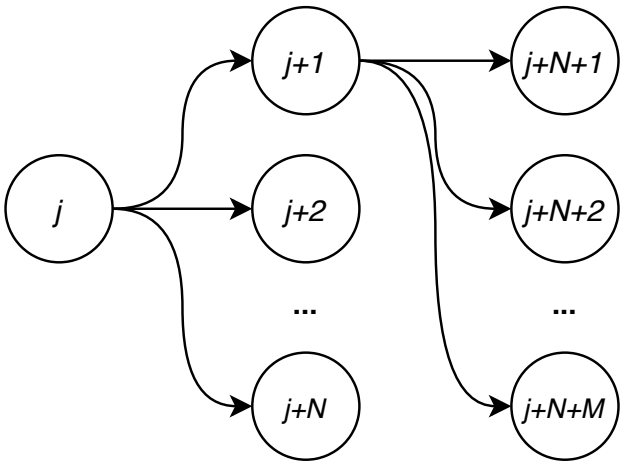

Figure 4.6: The general form of a Finite State Machine.

value. In particular, State 2 can operate 5× faster than State 3, resulting in 80% of the clock period being unutilized if the state machine transitions to State 2.

Determining the clock divisor for Cycle 4 follows a similar procedure. In Cycle 2, the DM read to line 1 is initiated. In Cycle 2, the set of tags and divisors corresponding to the possible states State 2, State 3, and State 4 are obtained. For each of these states, the corresponding divisor is worst-case divisor for the next cycle. For State 2, this is a divisor of 8; for State 3, a divisor of 8; and for State 4, a divisor of 3. Similarly to Cycle 3, the divisor selection is conservative and the worst-case divisor corresponds to a longer clock period than the active path if the state machine transitions from State 2 to State 5.

The three-cycle clock tuning procedure using the current state signal as the selector operates as follows, assuming the general form of the state machine depicted in Fig. 4.6.

1. In Cycle i, the current state of the state machine, which we will denote as $state_j$, initiates a read to line j of the DM. The memory line j contains a set of N *tags* and N *divisors*, where N is the possible number of next states for the state machine during Cycle $i+1$ and the set of *tag* values is the set *forall* a $\{$ $state_{j+a}$ $\}$, where $1 \leq a \leq N$.

2. In the next Cycle $i+1$, the circuit is in some state $state_{j+n}$ and the DM line is obtained from memory. The value $state_{j+n}$ is equal to one of the *tag* values returned from the DM, and is compared to the tags to select the divisor value to propagate to the clock generator. In this case, the selected divisor will correspond to the longest path delay among the possible states for Cycle $i+2$. For example, given M possible states for Cycle $i+2$ corresponding to M possible divisor values, the *max divisor* across the set of states *forall* b $\{$ $state_{j+N+b}$ $\}$ where $1 \leq b \leq M$ corresponds to the longest path. For simplicity of the Syncopation hardware, this selection is determined offline before the tags and divisor values are packed into the DMs.

3. Finally, in Cycle $i+2$, the circuit is in some state $state_{j+N+m}$ and the clock is tuned to the period determined by the *max divisor*, which corresponds to the longest path among all M possible states. Non-ideal performance will be achieved in cases where the required divisor for $state_{j+N+m}$ is less than the *max divisor*.

Address	Next States			Next State Divisors		
0			1			6
1	2	3	4	2	10	9
2		5	6		7	8
3			6			8
4			7			3
...	...			...		

Figure 4.7: Next State divisor memory packing based on the Finite State Machine in Fig. 4.1.

Next State Syncopation

In the Next State implementation of Syncopation, the *next_state* signal of each state machine is connected to the DM address port and the Divisor Selection Logic, as shown in Fig. 4.8. Unlike the Current State implementation of Syncopation, the Next State implementation allows the clock period to be tuned to exactly the required clock period.

The packed tags and divisor values for the state machine in Fig. 4.1 are shown in Fig. 4.7. Like the current state DMs, each line corresponds to a state and contains both tags and divisors. However, this implementation of Syncopation captures all the divisor values and corresponding tags.

The three-cycle tuning procedure using the next state signal as the selector operates as follows, again referencing the generic state machine depicted in Fig. 4.6.

1. In Cycle i, the next state signal of the state machine, which has a value we will denote as $state_{j+n}$ where $1 \leq n \leq N$, initiates a read to

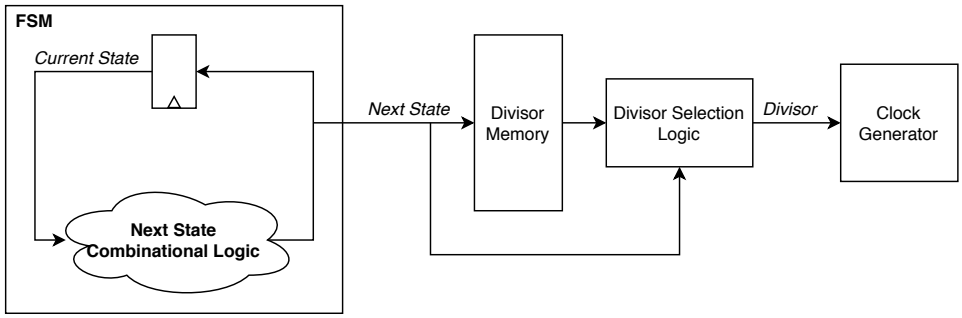

Figure 4.8: Next State based divisor selection logic.

line $j+n$ of the DM. The memory line $j+n$ contains a set of M *tags* and M *divisors*, where M is the number of possible states the circuit can be in Cycle $i+2$ and the set of *tag* values is the set *forall* b $\{$ $state_{j+N+b}$ $\}$ where $1 \leq b \leq M$.

2. In Cycle $i+1$, the DM line is obtained. During this cycle, the next state signal will resolve to a state value for Cycle $i+2$, $state_{j+N+m}$ where $1 \leq m \leq M$. By comparing this value to the state tags obtained from the DM, the divisor corresponding to the active path length can be determined and sent to the clock generator.

3. In Cycle $i+2$, the clock period is tuned to the minimum clock period required to accommodate the active paths for the current state $state_{j+N+m}$.

By using the next state signal of the state machine, we can determine exactly the minimum clock divisor for the state, even though the memory read is initiated two cycles in advance.

While Next State Syncopation enables additional performance over Cur-

rent State Syncopation by enabling selection of exactly the required divisor instead of the *max divisor*, there is also some possibility that some performance degradation can occur. Consider Fig. 4.4 and Fig. 4.8. In these circuit diagrams, we can clearly see the connection of the Current State and Next State signals from the user circuit leading to the Syncopation instrumentation. However, there is one key difference: in Current State Syncopation, the signal is connected directly from the state register, and the propagation delay of the Current State signal to the instrumentation is only wiring. This is not the case in Next State Syncopation, where the Next State computational logic lies on the path between the state register and the Syncopation instrumentation. In cases where the user state machine may have relatively simple computations in a state, this state path may be long. By adding additional instrumentation to this path, some designs which are already performance-limited by the complexity of the control logic may see reduced performance using Next State Syncopation.

4.3.2 Divisor Memories

Clock generator divisor values are determined through post-synthesis fine-grained timing analysis. In order to utilize these values during circuit operation, they are packed into the DMs to enable single-read access to the values corresponding to all possible circuit states. The DMs are sized statically before place-and-route for each FSM, and the memory contents are updated with divisor values after timing analysis without re-synthesizing the design.

The values used in each DM depend on whether the Current State or Next State implementation of Syncopation is being used. The size of the

DMs are the same across both Syncopation variants. In both cases, the DMs are sized statically before place-and-route by inspection of the state machine. For each FSM, the DM size in bits is:

$$bits = N_{states} \times max(\mathbf{N}_{next\_states}) \times (log_2(N_{states}) + D)$$

where N_{states} is the number of states in the state machine, corresponding to the number of lines in the memory; $\mathbf{N}_{next\_states}$ is a set containing the number of next states reachable for each state in the state machine; $max(\mathbf{N}_{next\_states})$ is the maximum number of next states from the set $\mathbf{N}_{next\_states}$, corresponding to the number of elements contained in each memory line; $log_2(N_{states})$ is the number of bits required to represent the state tag; and D is the divisor width in bits.

One limitation of this approach is that it may not size the memories efficiently using other common state encoding techniques, such as 1-hot encoding. This could be rectified using an encoder on the memory address port. However, the additional logic required to implement the encoder may reduce the benefit of other encoding techniques altogether. Since LegUp HLS encodes states with integer values, we did not perform studies to evaluate the performance of other encoding methods or evaluate other DM addressing techniques. This would be an interesting area for future work.

4.3.3 Divisor Selection Logic

The Syncopation clock period is directly related to the state of the user circuit, so the DMs are addressed by the state bits. During each cycle, a

divisor is read from the memory address corresponding to the state of each state machine. This lookup method generates multiple candidate divisors: one for each module instantiated on chip. The role of the divisor selection logic is to determine which of these candidate divisors will be propagated to the clock generator for clock tuning.

The simplest method to select the appropriate candidate divisor is to use a *max()* function to select the largest candidate divisor. Using this technique, the generated clock will always have a period long enough to accommodate the active path, ensuring that circuit operation will be correct, even with instantiated modules performing operations in parallel. However, the *max()* divisor selection technique assumes that all instantiated modules are performing computations in every clock cycle. This assumption is overly conservative in our implementation, since LegUp-generated circuits use a *start-finish* protocol to perform function calls. Using the *start-finish* protocol, only one module instantiation is performing computations in any given cycle, and all other modules are waiting to be called or are waiting for computed results to be returned.

In order to improve performance over the generic *max()* selection logic, we implement a mechanism to identify function calls and precisely determine which module instantiation is currently performing operations. To do this, the HLS schedule is inspected pre-synthesis to identify function call wait-states. These wait-states indicate when a "caller" module is waiting on the result of a computation being performed in a "callee" function. Then, during circuit operation, the state of each module is monitored to dynamically evaluate whether the "caller" or one of the "callees" is currently operating.

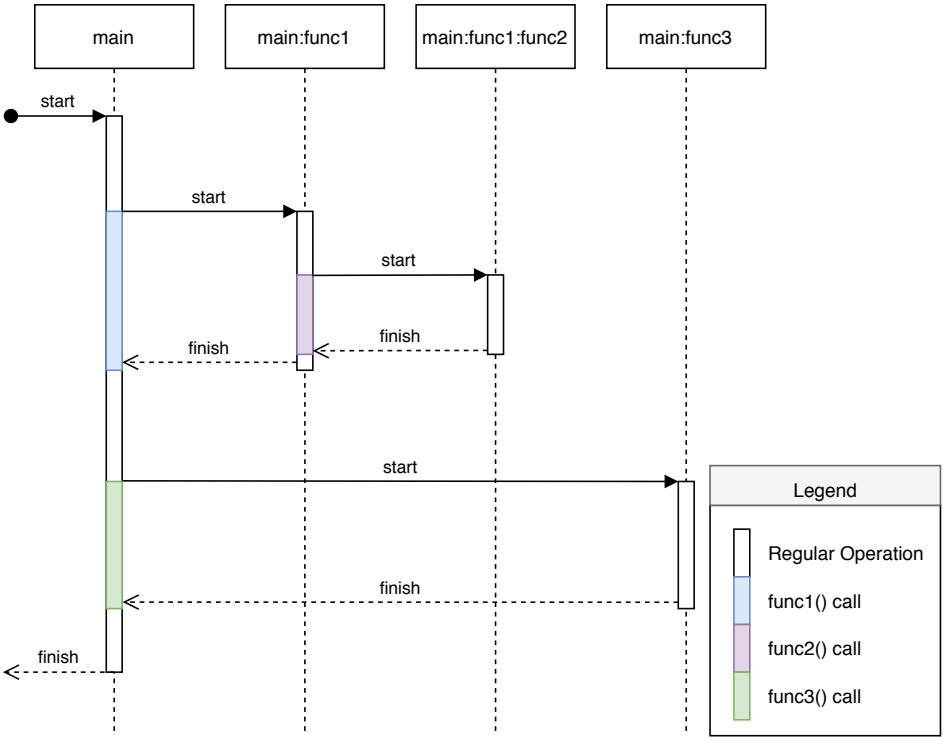

Figure 4.9: A sequence diagram describing the operation of a High-Level Synthesis-generated circuit.

As an example, consider the sequence diagram in Fig. 4.9. The main function of the circuit is the top level C function, and every other module is indicated hierarchically under the top-level design. In this case, the main function instantiates two modules (func1 and func3) and the func1 module instantiates another module (func2). During circuit operation, each callee module may eventually issue a function call to one of the instantiated submodules. In this case, the first function call is issued by main to func1. While the submodule func1 is operating, the main state machine remains in a wait-state until the finish signal is obtained. Similarly, func1 issues a function call to func2 at some point during circuit operation. While this occurs, func1 remains in a wait-state. Then, once both func2 and func1 finish operating, the main function continues operation until a function call is issued to func3. Finally, func3 returns and main finishes operation.

To implement divisor selection logic for this circuit, a small amount of instrumentation is inserted in every caller function; in this case, the caller functions are main and func1. This instrumentation monitors the *next_state* signal of the caller function state machine to determine whether the divisor for the callee or the caller function should be propagated to the clock generator.

4.3.4 Clock Generator Circuitry and Clock Distribution

The Syncopation clock is produced by a custom soft-logic clock generator. This custom clock generator is capable of producing clock signals at a wide range of frequencies (50–250 MHz) and is capable of changing the clock frequency, glitch-free, on a per-cycle basis. Glitches are transient logic errors

which can cause corruption of circuit data when a clock signal glitch causes flip-flops to prematurely register signals which have not yet settled to their final computed value.

While there are other clock generating techniques (such as Phase-Locked Loops (PLLs), clock stretching, and clock multiplexing) available for FPGAs, these options do not meet the specifications for our implementation either due to limited dynamic range or long switching times. For example, dynamically reconfiguring PLLs requires tens of cycles to change the configuration and lock the clock output. Clock multiplexing also requires multiple cycles to switch between clock sources to match clock phases and prevent glitches. Clock stretching techniques, such as in [11], do not incur a long switching overhead, but can only change the clock period by 25%. In previous work proposing ACM, hybrid clocking schemes were adapted using both clock stretching and clock multiplexing techniques to overcome the limitations of both [20].

Our clock generator is the 50% duty cycle clock generator from [40], shown in Fig. 4.10. We choose the 50% duty cycle implementation to accommodate dual-edge logic in some memories, although this requirement is likely conservative for the majority of applications. Our implementation uses a 4-bit integer divisor and a 500 MHz reference clock. The reference clock drives a counter which is provided along with a divisor value to two comparators. The first comparator determines the clock period by resetting the counter once the divisor value is reached by the counter. The second comparator determines the clock phase by comparing the counter to half of the divisor value. At the output of the clock generator, labelled "clock",

the signal is promoted to the global clock distribution network resources available on the FPGA. This promotion is automatically performed during place-and-route and is confirmed by analysing the synthesis messages for each benchmark.

The clock generator produces a clock signal which can vary in period on a per-cycle basis. However, there are some timing requirements which must be satisfied to prevent glitches. First, to guarantee that an update to the divisor value does not cause a glitch on the output of the clock phase comparator, we register the divisor so it can not be updated mid-cycle. Second, we require a minimum divisor value of 2 to prevent a race condition between the outputs of the period-reset comparator and the phase comparator.

It is possible to implement a more aggressive form of Syncopation which would forgo our requirement to have the clock divisor arrive before the rising edge of the clock period, therefore relaxing the three-cycle clock tuning procedure presented in Sect. 4.3.1. In fact, the divisor only needs to arrive at the clock generator in time to properly tune the falling edge of the 50% duty cycle phase, although insuring that the Syncopation control logic always satisfies this timing is more complex than our implementation. Additionally, it is likely that the 50% duty cycle requirement could be relaxed entirely, given that no dual-edge triggered logic is used in the design. This would relax the divisor selection procedure and timing even further.

This clock generator produces frequencies between 33 MHz and 250 MHz, which is sufficient for the observed frequencies in the CHStone [51] benchmark circuits (63–197 MHz). We did not evaluate clock generators with counter bits > 4, as the additional range provides no further benefit for the

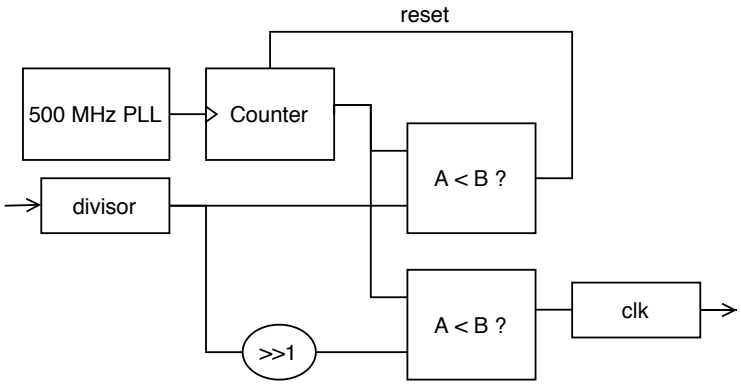

Figure 4.10: Integer divisor clock generator circuit.

circuits we surveyed.

4.4 Fine-Grained Static Timing Analysis

Once the HLS-generated RTL and instruction schedules have been analyzed and modified to include the DMs, Divisor Selection Logic, and clock generator, the design is synthesized using vendor place-and-route tools. No changes are made to the place-and-route algorithms. Once place-and-route finishes, timing analysis is performed to evaluate design performance. For a statically-clocked design, STA involves identifying the longest path in the design and determining the path delay to calculate the maximum operating frequency. For a Syncopation design, timing analysis also includes what we refer to as fine-grained static timing analysis, which evaluates cycle-specific path timing to generate data for the Syncopation DMs.

Fine-grained STA is a three-stage process. First, timing between registered instruction nodes in the design is measured for each state in each state

machine. Once the timing for these paths has been measured, we update the project timing constraints such that the per-state path delays satisfy the design timing requirements. Once these constraints are in place, we can request the maximum operating frequency of the unconstrained paths in the design on a per-module granularity. Typically, the longest unconstrained path consists of State Machine (SM) control logic which cannot be attributed to specific cycles in circuit operation. This maximum frequency is then used to limit all frequencies in the design to ensure that timing requirements for the control logic will not be violated. Finally, these results are combined and translated into per-state divisor values and packed into the DMs.

4.4.1 Per-State Timing and Synthesis Directives

Per-state timing is estimated using a custom Tcl script which is automatically generated by analyzing the LegUp-generated instruction schedule and the RTL. In the instruction schedule, each instruction is analyzed to identify operands that are produced or consumed. These operands are then matched to the corresponding signal names in the Verilog RTL, which are used to identify signals during post-place-and-route timing analysis. To determine the path delay between these operands for each state, Syncopation automatically generates a Tcl script which requests timing data between all operands in a state, and labels the resulting data with the corresponding state. We perform our requests using the Quartus Tcl command `report_timing`, which provides slack data between two named registered in the post-synthesis netlist. Since all registered nodes in the design are clocked

using our promoted Syncopation clock signal, this measurement automatically includes conservative timing estimates to account for process variation and clock skew.

Ideally, it would be possible to request timing data from or two any named register in the RTL using Quartus Tcl commands. However, this is not always the case. During synthesis, extensive register optimizations take place which merge, duplicate, and rename registers in the RTL [27]. The post-synthesis netlist does not retain information about optimized registers, and therefore it is not possible to obtain timing information about optimized nodes. However, if synthesis directives are inserted into the LegUp-generated RTL to prevent register optimizations it is possible to obtain this timing data at the cost of performance and resource utilization. For this reason, we have included *synthesis preserve* and *synthesis noprune* directives to retain information about registered nodes used to compute results of HLS instructions in the RTL. The *preserve* attribute prevents removal or minimization of a register without preventing other optimizations such as duplication; and the *noprune* attribute prevents removal of fanout-free nodes, which we require to estimate timing for intermediate computational steps [25]. The *noprune* directive is only included for a subset of the registers; in particular, the intermediate computational steps used for retiming of multi-cycle operations such as multiply operations are retained.

Table 4.1 shows the overhead of inserting these synthesis directives into the baseline CHStone benchmarks.

Inserting the synthesis directives into the baseline benchmarks reduces their performance by 10.8% (geomean), and more than doubles ALM utiliza-

Table 4.1: The Area and Performance Impact of Synthesis Directives

	Baseline		**SD Baseline**		
	FMax (MHz)	**ALMs** (%)	**FMax** (MHz)	(%)	**ALMs** (%)
adpcm	87.3	16.6	64.3	-26	32.1
aes	90.6	15.8	86.5	-5	27.9
bf	124.2	7.0	117.7	-5	22.2
dfadd	135.0	4.5	112.3	-17	18.9
dfdiv	102.9	9.9	103.4	+0	30.3
dfmul	98.8	9.9	93.0	-6	12.8
dfsin	96.4	26.2	83.4	-13	62.1
gsm	94.4	10.2	78.5	-17	25.0
jpeg	72.6	52.2	70.2	-3	75.1
mips	89.3	3.3	81.5	-9	9.9
motion	101.9	17.7	81.5	-20	26.0
sha	151.4	4.2	142.7	-6	10.1
Geomean	**101.7**	9.9	**90.7**	-10.9	24.4

tion. However, the performance of the statically clocked baseline circuits is determined solely by the critical path delay. Since the performance of Syncopation circuits is not solely determined by the critical path delay, it is still possible to regain and exceed the performance of the baseline circuits without synthesis directives using our technique. In Chapter 5, we will quantify to what extent this is true.

4.4.2 Control Logic Constraints

Once the per-state timing is determined, it is necessary to determine the maximum operating frequency of the control logic in each module to ensure timing requirements are not violated. As an example, consider Program 4.1.

In Program 4.1, per-state path delays have been determined by fine-

Program 4.1 A Syncopation schedule with various path delays and a function-call wait state.

```
Top Design:
    Module A:
        State 1: Delay 6.67 ns
        State 2: Delay 5.00 ns
        State 3: Function call B()
        State 4: Delay 11.11 ns
    Module B:
        State 1: Delay 8.33 ns
        State 2: Delay 8.33 ns
        State 3: Delay 5.00 ns
        State 4: Delay 5.00 ns
```

grained static timing analysis. However, these delays were determined based on the timing for instructions in each state which produce and consume operands, not the control logic of the circuit state machines. To ensure that the individual state timing doesn't violate the minimum delay for the control logic, we separately measure the longest control logic delay of each module. To do this, we generate an Synopsis Design Constraints (SDC) file containing `set_max_delay` constraints for the paths already measured during fine-grained static timing analysis to ensure that these paths won't generate exceptions. Then, we generate timing reports requesting the longest delays within each module that weren't measured during fine-grained timing analysis. Then, the delays obtained in the reports are used to constrain the timing of individual states according to the following rule:

$$d_{updated} = max(d_{sta}, d_{control})$$

where d_{sta} is the delay measured during fine-grained static timing analysis,

$d_{control}$ is the longest delay in the state machine control logic, and $d_{updated}$ is the updated clock period for the state which will not violate timing of the state machine. Program 4.2 shows how the example program in Program 4.1 would be updated after the control logic delays are measured.

Program 4.2 A Syncopation schedule including control logic constraints.

```
Top Design:
    Module A: Longest Control Logic Delay 3.33 ns
        State 1: Delay 6.67 ns
        State 2: Delay 5.00 ns
        State 3: Function call B()
        State 4: Delay 11.11 ns
    Module B: Longest Control Logic Delay 5.55 ns
        State 1: Delay 8.33 ns
        State 2: Delay 8.33 ns
        State 3: Delay 5.55 ns *
        State 4: Delay 5.55 ns *
```

To constrain timing for each state and avoid control logic timing violations, the first step is to measure the longest control logic delay for each module. In this circuit there are two modules instantiated: Module A and Module B. The longest control logic path for each of these modules as been measured as 3.33 ns and 5.55 ns, respectfully. Now, we can update the state delays accordingly. As indicated in Program 4.1 with asterisks, the state delay for State 3 and State 4 in Module B have been lengthened, since using the previously measured delays of 5.00 ns violated control logic timing requirements. Instead, these states will be executed with a clock period of 5.55 ns.

After the timing constraints are updated for all states in the design, the new per-state delays are converted into divisor values and packed into the

DMs.

4.4.3 Divisor Calculation

Given a list of maximum frequencies per-state constrained to both the maximum per-module frequency and the maximum per-design frequency, the divisor values for the clock generator can be calculated. The divisor value corresponding to a state is calculated using the state frequency and the reference clock frequency for the clock generator. The divisor value D is defined as:

$$D = \lceil \frac{F_{ref}}{F_{state}} \rceil$$

where F_{ref} is the reference clock frequency for the clock generator and F_{state} is the maximum clock frequency for a given state. A non-integer result is "snapped" to the larger integer which corresponds to a longer clock period. Once the divisor values for all states have been computed and packed into the DMs, the circuit is operational.

4.5 Enhanced Synthesis

Syncopation circuits are able to gain performance by adjusting the clock frequency according to the individual clock cycle. However, the integer clock divider does result in non-ideal performance compared to a clock divider with infinite frequency tuning granularity. Ideally, the place-and-route tool would be aware of the integer clock divider and attempt to optimize each path to meet timing for a higher frequency bucket. It would be possible to

implement this custom place-and-route algorithm using an open-source tool chain, however we opted to leave a custom tool chain to future work. Instead, we show that we can achieve better performance without implementing a custom algorithm by using vendor tools, fine-grained timing targets, and an additional synthesis pass using a technique we call *Enhanced Synthesis*.

Generally, timing-driven place-and-route tools target effort to optimize the longest paths in the circuit in order to shorten the critical path length. The critical path optimization target improves the performance of statically-clocked designs because the clock frequency is only dependent on the timing of the longest paths in the design. Since the performance of Syncopation designs is not entirely dependent on the critical path delay, the additional effort to optimize the most-critical paths may be disproportionate to the possible gain. Instead, synthesis effort would be better targeted to optimize the delay of every state's computational path to meet timing for the higher frequency produced by the clock generator.

As an example, consider a path (which we will denote as Path A, with delay j) which is in a design with critical path J. In this example, $j < J$ so Path A is not a critical path. In a statically clocked design, Path A would be executed at the frequency $F_{max} = 1/J$, as no instrumentation is in place for clock tuning. By adding the Syncopation instrumentation to the design and assuming an ideal (infinite granularity, for now) clock generator, we can calculate the minimum clock period required to meet timing for Path A and tune the clock to the corresponding frequency $F_A = 1/j$ in the corresponding states. In this case, since $j < J$, we will achieve lower delay (higher performance) by using the Syncopation design.

Now consider the same circuit, but assume an integer divider clock generator. In the best case, Path A meets the following requirements: (1) the path length corresponds to the minimum-delay implementation achievable by the synthesis tool (i.e. cannot be significantly more optimized by increasing synthesis effort on this path) and (2) the maximum operating frequency $F_A = 1/j$ corresponds exactly to a frequency generated by the clock generator. This ideal case makes additional synthesis effort to optimize this path redundant.

In reality, a path in the circuit implementation is unlikely to meet these two requirements. Paths are likely to fail the first requirement because there may be many ways to implement a path, and it is unlikely to always select the lowest-delay implementation. Furthermore, the lowest-delay implementation may only be determined by the synthesis tool given sufficient motivation to optimize a path with a delay lower than the critical path delay. Paths are also likely to fail the second requirement for similar reasons; without sufficient incentivization to achieve a target delay, paths delays are equally likely to be just longer than a clock period generated using our technique, resulting in performance loss due to "snapping" of the clock period to a longer value to prevent timing violations. In the same context of the previous example, consider another path (Path B) with delay $k < J$. Since the synthesis tool is not incentivized to optimize this path, the implementation may not be optimized to minimize the path delay. Furthermore, since the synthesis tool has no awareness of the integer clock divider, the worst-case for Path B is that the path delay k is slightly longer than a clock period produced by the clock generator, and therefore incurs the maximum overhead.

For example, our 500 MHz, 4-bit clock generator produces clocks with 4 ns and 6 ns periods, but in the case where $k = 4.1$ ns, the clock period will "snap" to 6 ns, a 46% increase in delay.

For Syncopation circuits, a better synthesis tool would target resources and computational effort to optimize paths close to the boundary between generated frequencies, not just paths with the longest delays in the design. To achieve targeted performance improvement, we automatically generate per-path synthesis timing constraints using the results of the fine-grained static timing analysis in an additional synthesis pass. The timing constraints improve the place-and-route implementation by setting additional constraints for all paths, such that the synthesis tool has target timing constraints for all paths in the design, instead of just for paths which violate the critical path frequency. In our scripts, the individual path constraints are set using the Tcl command `set_max_delay`. Since Syncopation uses an integer clock divider, we set the maximum delay for each path to the next-fastest clock period, relative to the previously calculated per-state timing. Using the previous example, this would mean Path B with delay $k = 4.1$ ns would have a target maximum delay of 4 ns.

It is possible additional performance could be obtained using more aggressive timing constraints. In our existing procedure, we set timing constraints for paths to match the next highest frequency bucket for each path. To achieve further improvement, it would be necessary to optimize paths such that they meet an even higher frequency bucket. We investigated this experimentally, and found that this led to $< 1\%$ additional performance over Enhanced Synthesis.

Once the timing constraints files are generated, a second synthesis pass is performed to improve the circuit implementation, and fine-grained synthesis is performed again to update timing analysis results and the DM contents.

4.6 Summary

In this chapter, we presented Syncopation: an ACM technique which boosts performance of HLS-generated circuits by tuning the clock period on a cycle-by-cycle basis according to the operations currently being performed. At compile time, the RTL and instruction schedules produced by the HLS tool (LegUp) is profiled and instrumented with the Syncopation hardware, including the divisor memories, divisor selection logic, clock generator, and synthesis directives. We presented two methods of instrumenting the user circuit to select the appropriate clock divisor: Next State and Current State Syncopation. After synthesis, fine-grained static timing analysis is performed to determine circuit timing requirements on a per-state, per-module, and per-design basis. These timing measurements are then used to calculate divisor values which are packed into the divisor memories to tune the clock generator at run-time. Additionally and optionally, the per-state timing constraints are also used to target resources during an additional synthesis pass to minimize delay for paths in the design in a technique we call Enhanced Synthesis. The techniques described in this chapter do not require changes to the FPGA architecture, HLS tools, or place and route algorithms. In the next chapter, we evaluate the performance and overhead of Syncopation by measuring the frequency, area, and memory requirements across a set of

benchmarks.

Chapter 5

Adaptive Clock Management Using Syncopation

In Chapter 4, we introduced *Syncopation*, an adaptive clock management strategy for HLS-generated circuits. Syncopation uses a memory-based divisor reference system and dynamic selection logic to select a pre-calculated divisor value, and tunes the clock period according to the divisor value to accommodate the critical path of each circuit state on a cycle-by-cycle basis. In this chapter, evaluate the performance benefit achieved using Syncopation for general HLS circuits.

We begin with an introduction to the methodology, evaluation metrics, and the studied configurations of Syncopation. Then we present the experimental set-up used to validate circuit functionality in hardware and simulation. Finally, we present the results achieved using Syncopation and summarize our performance improvement on generic HLS circuits.

5.1 Introduction

To evaluate Syncopation, we use the CHStone [51] HLS benchmarks compiled with LegUp 4.0 and synthesized with Quartus Pro 15.0 Web Edition,

targeting Intel Cyclone V FPGAs (5CSEMA5F31C6N). To ensure validity of the HLS instruction schedule, we turned off retiming to implement Syncopation circuits. The baseline circuit performance is measured with retiming enabled.

5.1.1 Evaluation Metrics

Typically, the performance of digital circuits is quantified by F_{max}, which is the maximum frequency at which the circuit can operate. In normal STA, the F_{max} is determined from the critical path delay and is calculated as:

$$F_{max} = \frac{1}{t_{delay} + t_{setup} + t_{skew}}$$

where t_{delay} is the combinational delay of the critical path logic, t_{setup} is the minimum time before the rising edge of the clock that the data must be stable at the input of a flip-flop in order to be latched correctly, and t_{skew} is the difference in time that a clock edge arrives at a destination component compared to the source.

F_{max} is not sufficient to describe the performance of Syncopation circuits, where the clock period changes every cycle according to the delay of the active paths. Instead of using F_{max}, we we propose describing performance of Syncopation circuits using *effective* F_{max} which we denote F_{eff}. F_{eff} is defined as:

$$F_{eff} = \frac{t_{sta}}{t_{sync}} F_{max}$$

where the wall-clock time for a statically clocked circuit is denoted as t_{sta} and the wall-clock time for a Syncopation circuit is denoted as t_{sync}. Since

the wall-clock time required for Syncopation circuits to complete operation depends on *which* states are active, calculating F_{eff} requires simulating the designs using sample inputs. This is different from evaluating the performance of statically clocked circuits, which only require STA to determine F_{max}, since the clock frequency is independent of which states are activated.

In terms of performance to the end-user, an improvement in effective frequency is proportional to the improvement in operation latency for a set of inputs applied to a benchmark circuit. Specifically, a 10% increase in effective frequency corresponds to a 10% decrease in operation latency for the same benchmark and input vector.

5.2 Validation

To validate the operation of the Syncopation circuits we programmed the target FPGA with all the benchmark circuits presented in this dissertation. Additionally, we confirmed that timing was properly constrained for all paths in simulation using vendor static timing analysis tools.

In this section, we describe how we confirmed functionality of the Syncopation circuits was confirmed both in hardware and using simulation.

5.2.1 Validation in Hardware

In hardware, circuit operation was validated by executing all circuits on a known set of inputs and confirming receipt of the expected result. Fig. 5.1 depicts the block diagram of our validation setup. At the beginning of operation, the LegUp-generated circuit reads the input data from an on-

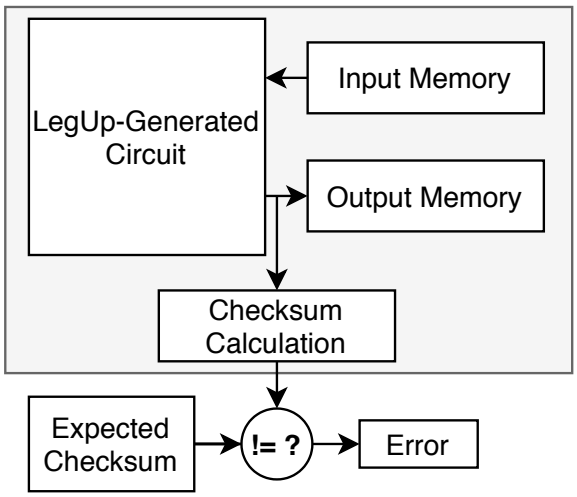

Figure 5.1: Validation circuit for Syncopation.

chip memory. Then, circuit operation is performed as usual, with the clock controlled by Syncopation. Finally, the result is written back into an output memory and a checksum is generated. The checksum is then compared to the expected value determine using simulation. If the produced checksum does not match the expected value, an error signal is generated and recorded.

To ensure that circuit functionality is reliable, our validation setup also automatically reset and repeated circuit operation $1,024$ times. A counter was used to record the total number of errors observed over all executions for each benchmark. The number of clock cycles each benchmark was observed for (across all Syncopation implementations) as well as the number of errors observed for each benchmark is included in Table 5.1.

By evaluating the performance of all our benchmark circuits over this many clock cycles, we seek to confirm three hypotheses: first, that the fine-grained STA technique used to determine the minimum Syncopation clock

Table 5.1: Validation Results in Hardware

	Total Clock Cycles Observed	**Error Count**
adpcm	40157184	0
aes	27801600	0
bf	503580672	0
dfadd	1978368	0
dfdiv	5790720	0
dfmul	666624	0
dfsin	174305280	0
gsm	14635008	0
jpeg	3998300160	0
mips	15470592	0
motion	25377792	0
sha	512305152	0

period is accurate; second, that the control logic for the Syncopation logic and the LegUp-generated state machines can be properly reset; and third, that glitches and jitter on our clock signal do not cause failures during typical circuit operation.

In our validation studies, we did not perform rigorous investigation to determine test vectors which would activate the longest data-dependent paths of each circuit in the design to attempt to cause failures of each circuit. Instead, we chose to use simulation techniques to evaluate whether the paths associated with each state had positive slack relative to their Syncopation clock period. Since the STA techniques employed by Quartus use conservative timing estimates for both the circuit implementation and the clock distribution networks, passing simulation is equivalent to ensuring that we are not overclocking any paths within our design.

5.2.2 Validation in Simulation

To validate circuit operation using simulation, SDC files were generated which used `set_max_delay` commands to describe the timing requirements for all paths in the Syncopation circuit. The command `set_max_delay` generates an exception for paths which are longer than the maximum permitted delay for that specific path. This enables us to ensure that vendor static timing analysis will only generate timing errors if a path's delay is longer than the clock period selected by Syncopation.

Using this technique, the Syncopation-determined constraints were confirmed for all benchmarks for both the Next State and the Enhanced Synthesis implementations of Syncopation.

5.3 Performance Results

In Chapter 4, multiple methods of implementing the Syncopation instrumentation were introduced. In our experiments, we evaluate Syncopation using both the Current State and Next State configurations (described in Sect. 4.3.1), active module divisor selection logic (described in Sect. 4.3.3), and per-module timing constraints (described in Sect. 4.4.2). Enhanced Synthesis (described in Sect. 4.5) was also presented as a technique to target synthesis to overcome some of the limitations of using an integer-divisor clock generator.

In this section, first we compare the Current State and Next State implementations of Syncopation, and discuss performance tradeoffs. Next, we present our performance using Enhanced Synthesis and evaluate the bene-

fits of a targeted synthesis pass by comparing our results to the maximum theoretical effective frequency achievable assuming a clock generator with the ability to tune to exactly the required clock period, as opposed to relying on the integer-based mechanism. Finally, we compare our results to our previous version of Syncopation[3] which did not contain the active module divisor selection logic.

5.3.1 Current State Syncopation

Using the current state (CS) to select the divisor, there is inherent loss as the state is not known in advance, and the *max divisor* among the possible next states is selected for clock tuning, as described in Sect. 4.3.1. However, it is possible that using the current state instead of the next state implementation avoids additional performance-limiting delay incurred by inserting the Divisor Selection Logic after the Next State Computational Logic, described in Sect. 4.3.1. The results using this configuration are shown in Table 5.2.

Using the current state of the circuit does provide additional performance over the baseline with synthesis directives (9.9% on average (geomean)). However, this benefit is not significant enough to overcome the performance overhead of the synthesis directives themselves. Compared to the baseline, the performance improvement of individual benchmarks range in performance from -25 to +25%, with an average performance change of -2% (geomean).

Given the previous description of Current State and Next State Syncopa-

[3]Published as a short paper in the 2020 International Conference on Field-Programmable Logic and Applications.

Table 5.2: Performance of Current State Syncopation (Sync-CS)

	Baseline	**SD Baseline**	**Sync-CS**	
	FMax (MHz)	**FMax** (MHz)	**FEff** (MHz)	(%)
adpcm	87.3	64.3	84.4	-3
aes	90.6	86.5	95.1	5
bf	124.2	117.7	110.5	-11
dfadd	135.0	112.3	101.1	-25
dfdiv	102.9	103.4	99.8	-3
dfmul	98.8	93.0	112.1	13
dfsin	96.4	83.4	89.0	-8
gsm	94.4	78.5	94.2	0
jpeg	72.6	70.2	77.8	7
mips	89.3	81.5	111.3	25
motion	101.9	81.5	105.3	3
sha	151.4	142.7	125.1	-17
Geomean	**101.7**	**90.7**	**99.6**	-2.0

tion presented in Chapter 4, it is possible that additional performance can be achieved using Next State Syncopation, which does not require conservative *max divisor* selection.

5.3.2 Next State Syncopation

To improve performance over the Current State implementation of Syncopation, we can use the Next State implementation of Syncopation to ensure that the clock is tuned to exactly the clock period required by the active state. Table 5.3 presents the performance of the CHStone benchmarks using this technique.

As expected, tuning the clock period exactly to the required clock period according to the per-state timing measurements prevents unnecessarily in-

Table 5.3: Performance of Next State Syncopation (Sync-NS)

	Baseline	**SD Baseline**	**Sync-NS**	
	FMax (MHz)	**FMax** (MHz)	**FEff** (MHz)	(%)
adpcm	87.3	64.3	97.5	12
aes	90.6	86.5	95.9	6
bf	124.2	117.7	126.7	2
dfadd	135.0	112.3	111.3	-18
dfdiv	102.9	103.4	99.5	-3
dfmul	98.8	93.0	101.9	3
dfsin	96.4	83.4	80.9	-16
gsm	94.4	78.5	97.8	4
jpeg	72.6	70.2	80.2	10
mips	89.3	81.5	85.6	-4
motion	101.9	81.5	111.2	9
sha	151.4	142.7	221.7	47
Geomean	**101.7**	**90.7**	**105.0**	+3.2

creasing design latency and improves performance. In our experiments, we see a 15.9% average (geomean) performance improvement over the baseline with synthesis directives, and a 3.2% average (geomean) performance improvement over the baseline. Overall, using the Next State implementation of Syncopation improves performance by approximately 5% over Current State Syncopation on average. The performance improvement for individual benchmarks ranges from -18% to +47%.

While the performance improved on average across the benchmark designs, some benchmarks saw reduced performance using the Next State version of Syncopation. As elaborated in Sect. 4.3.1, Next State Syncopation adds additional logic between the Next State logic and the Syncopation hardware, which may increase the longest computational path length in

designs which are performance limited by control logic instead of computational logic. In particular, *dfmul*, *dfsin*, and *mips* lose performance (-10%, -10%, and -23%, respectfully). However, other designs do gain significant performance (up to 77% for *sha*), suggesting that Next State Syncopation is generally a favourable implementation over the Current State version.

5.3.3 Enhanced Synthesis

As described in Sect. 4.5, Enhanced Synthesis is a technique to regain some of the performance lost due to the integer clock generator by targeting timing-driven synthesis to optimize each path in the design to meet timing for a higher clock frequency produced by the clock generator. To further motivate the need for this technique, Table 5.4 compares the performance of Syncopation to the theoretical performance achievable by Syncopation if the clock period was tuned to exactly the delay required by the active state. This theoretical performance quantifies the performance we could gain with a clock generator capable of tuning to any arbitrary frequency.

On average, the theoretically achievable performance is 12.6% over the baseline circuit implementations, which is 9.4% higher than the performance achieved using Syncopation. However, it is possible to recover some of this unrealized potential by revealing details about the clock generator to the place-and-route tool. Currently, the place-and-route algorithms are not attempting to optimize paths to meet the frequencies generated by our clock generator, and in fact are not incentivized to optimize the non-critical circuit paths at all. Enhanced Synthesis uses SDC file constraints to target path optimization efforts across all paths in the design, instead of only the

Table 5.4: Performance of Syncopation Compared to the Theoretical Performance Achievable without an Integer Divider Clock Generator

	Baseline	**SD Baseline**	**Sync-NS**		**Theoretical**	
	FMax (MHz)	**FMax** (MHz)	**FEff** (MHz)	(%)	**FEff** (MHz)	(%)
adpcm	87.3	64.3	97.5	12	108.0	24
aes	90.6	86.5	95.9	6	102.5	13
bf	124.2	117.7	126.7	2	147.8	19
dfadd	135.0	112.3	111.3	-19	121.8	-10
dfdiv	102.9	103.4	99.5	-4	101.6	-1
dfmul	98.8	93.0	101.9	3	118.7	20
dfsin	96.4	83.4	80.9	-16	88.6	-8
gsm	94.4	78.5	97.8	4	103.9	10
jpeg	72.6	70.2	80.2	10	85.1	17
mips	89.3	81.5	85.6	-4	94.4	6
motion	101.9	81.5	111.2	9	124.5	22
sha	151.4	142.7	221.7	47	230.9	53
Geomean	**101.7**	**90.7**	**105.0**	+3.2	**114.6**	+12.6

most-critical paths. Table 5.5 shows the performance of a single Enhanced Synthesis pass.

As expected, Enhanced Synthesis typically improves the performance of Syncopation by targeting synthesis effort to optimize paths near clock generator frequency boundaries, with few exceptions. On average, the performance across all benchmarks improves 23.8% compared to the baseline with synthesis directives, and 10.3% compared to the baseline. The performance of individual benchmarks improve between -20 and +50%, with between -4 to +21% performance improvement over Syncopation alone. Additional iterations of Enhanced Synthesis do not significantly improve design performance (<1%), so we have not included these results in this thesis.

Table 5.5: Performance of Enhanced Synthesis

	Baseline	**SD Baseline**	**Sync-NS**		**Sync-NS+ES**	
	FMax (MHz)	**FMax** (MHz)	**FEff** (MHz)	(%)	**FEff** (MHz)	(%)
adpcm	87.3	64.3	97.5	12	119.4	37
aes	90.6	86.5	95.9	6	97.7	8
bf	124.2	117.7	126.7	2	124.2	0
dfadd	135.0	112.3	111.3	-18	108.5	-20
dfdiv	102.9	103.4	99.5	-3	99.0	-4
dfmul	98.8	93.0	101.9	3	117.6	19
dfsin	96.4	83.4	80.9	-16	93.1	-4
gsm	94.4	78.5	97.8	4	102.0	8
jpeg	72.6	70.2	80.2	10	76.9	6
mips	89.3	81.5	85.6	-4	104.7	17
motion	101.9	81.5	111.2	9	125.0	23
sha	151.4	142.7	221.7	47	226.9	50
Geomean	**101.7**	**90.7**	**105.0**	+3.2	**112.2**	+10.3

5.3.4 Instrumentation Overhead

The bulk of the additional area utilization for Syncopation is due to the synthesis directives. Tab. 5.6 shows the ALM utilization across the designs we studied.

Without Syncopation instrumentation or synthesis directives, the benchmark circuits utilize 9.9% of our device resources. With synthesis directives, this utilization increases to 24.4%. This overhead is not insignificant and degrades circuit performance. Attempts to minimize this cost included employing these directives only for datapath nodes and studying the Quartus synthesis reports for evidence of optimizations. Unfortunately, sufficient detail was not found in the reports to forgo the need for the directives and thus we were not able to remove them for this work.

Table 5.6: Syncopation Instrumentation Overhead for device 5CSEMA5F31C6N, 32070 ALMs.

	Baseline (ALM %)	**SD Baseline** (ALM %)	**Syncopation** (ALM %)	(%)
adpcm	16.6	32.2	32.3	16
aes	15.8	27.9	28.0	12
bf	7.0	22.2	22.0	15
dfadd	4.2	18.9	19.1	15
dfdiv	9.9	30.3	30.4	21
dfmul	3.4	12.8	12.8	9
dfsin	26.2	62.1	62.3	36
gsm	10.2	25.0	25.3	15
jpeg	52.2	75.1	75.3	23
mips	3.3	9.9	10.2	7
motion	17.7	26.0	26.4	9
sha	4.2	10.1	10.1	6
Geomean	9.9	24.4	24.5	+14.6

Adding in the Syncopation hardware, the utilization increases to 24.5% of the device, which is an additional 15% of the device compared to the baseline. However, only 0.1% of this device utilization is due to the Syncopation instrumentation, and as previously mentioned, a vendor or custom-tool implementation of Syncopation would not require the insertion of Synthesis Directives.

5.3.5 Memory Overhead

The number and size of the Divisor Memories may vary significantly depending on the design. Table 5.7 shows the memory utilization of the baseline and Syncopation circuits. For simplicity, the change in memory utilization is also indicated.

Table 5.7: Syncopation Memory Utilization.

	Baseline (kB)	**Syncopation** (kB)	
adpcm	1.1	1.8	+0.7
aes	4.3	5.0	+0.7
bf	18	19	+0.5
dfadd	1.0	1.5	+0.5
dfdiv	0.2	1.9	+1.7
dfmul	1.0	1.1	+0.1
dfsin	1.1	3.6	+2.4
gsm	1.2	1.7	+0.5
jpeg	56	68	+12
mips	0.4	3.3	+2.9
motion	4.0	4.7	+0.7
sha	16.4	16.6	+0.2
Geomean	2.4	4.4	+2.0

Syncopation DMs are sized to contain the state tags and divisors for each state machine. For example, the number of lines required in each DM is determined by the number of states in the corresponding state machine and the width of each memory line is determined by the state with the most next states in the state machine. Additionally, since a State Machine is generated for each function in the software level description, the number of DMs is equal to the number of functions in the software level description.

On average, the Syncopation DMs require an additional 2 kB (geomean) of on-chip memory in the benchmarks we studied. However, the DM size is sensitive to features of the RTL implementation which designers may not have the expertise to alter. Additionally, since State Machines are automatically generated by the LegUp HLS tool, the designer does not directly control these parameters; a future extension of Syncopation may include

alterations to the LegUp source code to take designer pragma directives to constrain DM size. Without altering the LegUp tool, a designer could indirectly influence the DM size by partitioning code into multiple functions, instead of a large *main* function.

5.3.6 Comparison to Previous Version of Syncopation

A previous publication that described Syncopation used a Next State implementation with a simple *max()* function to select the longest active paths, instead of the active module identification logic. The active module identification logic works with LegUp-generated circuits because only one module is performing computations in each clock cycle, which may not be true in general HLS-generated circuits which incorporate parallelization. Table 5.8 includes the results from this original publication.

Without the active module identification Logic, Syncopation does improve performance on average over the baseline with synthesis directives, but does not overcome the performance loss due to insertion of the directives, resulting in a 2.7% loss in performance compared to the baseline.

5.4 Discussion

We have shown that the Syncopation Adaptive Clock Management techniques can improve the performance of High-Level Synthesis-generated circuits, and using Enhanced Synthesis along with Syncopation can further improve performance.

Fig. 5.2 summarizes the baseline, theoretical performance, Next State

Table 5.8: Performance of Syncopation without Active Module Identification Logic (Previous Work)

	Baseline	**SD Baseline**	**Sync-NS**		**Sync-1.0 [21]**	
	FMax (MHz)	**FMax** (MHz)	**FEff** (MHz)	(%)	**FEff** (MHz)	(%)
adpcm	87.3	64.3	97.5	12	95.4	9
aes	90.6	86.5	95.9	6	99.6	10
bf	124.2	117.7	126.7	2	111.9	-10
dfadd	135.0	112.3	111.3	-18	100.0	-26
dfdiv	102.9	103.4	99.5	-3	103.4	0
dfmul	98.8	93.0	101.9	3	109.6	11
dfsin	96.4	83.4	80.9	-16	84.8	-12
gsm	94.4	78.5	97.8	4	97.6	3
jpeg	72.6	70.2	80.2	10	73.3	1
mips	89.3	81.5	85.6	-4	95.1	6
motion	101.9	81.5	111.2	9	100.8	-1
sha	151.4	142.7	221.7	47	126.5	-16
Geomean	**101.7**	**90.7**	**105.0**	3.2	**99.0**	-2.7

Syncopation, and Enhanced Synthesis performance across the CHStone benchmark circuits. While insertion of the synthesis directives reduces the performance of the benchmarks, Syncopation typically recovers this loss and gains additional performance over the baseline design (3.2% on average), and using Syncopation with Enhanced Synthesis achieves even better performance (10.3% on average and up to 50%).

Our implementation of Syncopation achieved this performance boost with minimal instrumentation and on-chip memory utilization and without changing the HLS or place-and-route toolchains. However, additional benefits could be achieved by customizing these tools. In this section, we will discuss some of these extensions. Additionally, we will discuss extended

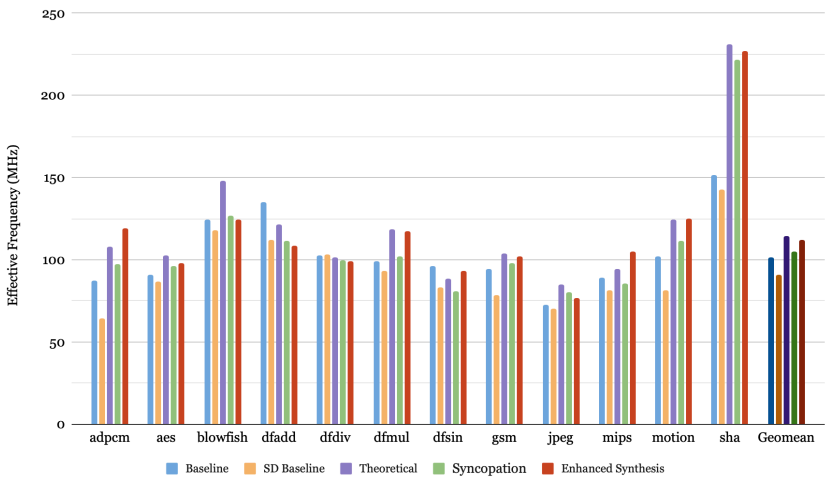

Figure 5.2: Syncopation Performance on CHStone Benchmark Circuits.

applications which could benefit from Syncopation.

5.4.1 Custom Place-and-Route Toolchain

Our implementation of Syncopation requires synthesis directives to enable the use of our fine-grained static timing analysis scripts. Without synthesis directives, many named nodes in the RTL cannot be located in the post-synthesis netlist, and therefore path timing cannot be mapped back to the original state machine state. As stated previously, this could be rectified if Syncopation was implemented as a vendor tool, in which case the node optimizations or transformations which occurred in intermediate netlist optimizations could be tracked and recorded for timing reports. An-

other method to eliminate the need for these synthesis directives is to implement a custom place-and-route tool for Syncopation which automatically performs fine-grained timing analysis.

In addition to automatic fine-grained timing analysis, a custom place-and-route tool would forgo the necessity of an additional synthesis pass for Enhanced Synthesis. Currently, Enhanced Synthesis is only possible after computing the clock generator frequency for each path in the design and automatically generating a timing constraint file to use in a second, full place-and-route pass through a vendor synthesis tool. While the second synthesis pass is shown to improve design performance, it doubles the synthesis time, which is not desirable for large or complex designs. Implementing a custom place-and-route tool would enable targeted timing driven synthesis to automatically adapt path timing targets according to current path timing, eliminating the need for additional synthesis time.

5.4.2 Custom HLS Toolchain

A custom HLS toolchain, or alterations to the LegUp HLS tool, could also improve the performance of Syncopation and enable higher levels of control for designers with hardware expertise. By altering the HLS toolchain, we can include options for pragmas to control the size and shape of the state machines, divisor memories, and enable additional transformations to improve performance such as Syncopation-aware pre-synthesis pipelining and retiming. Additionally, a custom HLS tool could be designed to identify modules in the user circuit which benefit from Syncopation and modules which suffer in performance, and automatically generate a multi-clock domain scheme to

achieve additional performance.

Multiple Clock Domain Optimization

As presented in this thesis, Syncopation does not utilize multiple clock domains and instead uses a single clock for the entire design. Using multiple clock domains is a common technique to improve performance of circuits. Using multiple clock domains with Syncopation could improve design performance further by enabling the option to employ static clock and Syncopation clock domains according to which would achieve better performance. In Table 5.9, we determined the maximum operating frequency of each module within the benchmark circuits we surveyed to quantify whether a multiple clock domain scheme would improve performance.

Table 5.9: Per-Module Maximum Operating Frequency

	Per-Module Frequencies (MHz)
adpcm	87.3, 355.6
aes	131.3, 147.23, 199.3, 231.2
bf	146.4, 184.5
dfadd	135.0, 144.7
dfdiv	102.9
dfmul	98.8
dfsin	96.4, 97.2, 120.2
gsm	94.4
jpeg	80.1, 83.8, 84.62, 116.1, 121.27, 129.2, 132.36
mips	89.3
motion	125.5, 142.7
sha	162.3, 168.1

As an example, consider the module frequencies for the *adpcm* bench-

mark: 87.3 and 355.6 MHz. The maximum clock frequency produced by our clock generator is 250 MHz, so it is guaranteed that the 355.6 MHz module would have reduced performance using Syncopation. By separating these modules into multiple clock domains, the 87.3 MHz module could remain in a Syncopation domain while the 355.6 MHz module could be in a statically-clocked domain. Clock domain partitions could be specified manually by the designer using pragma statements or automatically determined during HLS using timing techniques similar to those already performed during instruction scheduling.

5.4.3 Security Applications

A final extended application of Syncopation which we will discuss is the security of multi-tenancy FPGAs in modern cloud computing. The multi-tenancy paradigm has recently become more attractive as it enables fine-grained allocation of FPGA resources among end users. However, multi-tenancy opens user applications, which are performing operations on potentially sensitive or secret data, to side-channel power analysis attacks. Previous work [28, 50] has shown that secret data in victim circuits can be obfuscated from these attack methods by using randomized or time-fractured clock frequencies.

Extending Syncopation to perform data-protecting clock randomization would require minimal additional processing and instrumentation. For example, a simple clock randomization scheme could be implemented using a random number generator connected to the integer divisor clock generator. More sophisticated methods of data obfuscation could time-multiplex or

partition the user circuit into Syncopation (performance-oriented) or Obfuscation (data security-oriented) clock schemes. This technique could balance the performance of a data processing design as a whole with the need to protect sensitive, unencrypted data processing.

5.4.4 Syncopation for Arbitrary Circuits

In this thesis, we investigated applying adaptive clock management techniques to HLS-generated circuits. Although HLS circuits are particularly well-suited for this type of optimization due to the instruction schedules available from the HLS tools, Syncopation could hypothetically be applied to any arbitrary state machine circuit. In particular, Automated Test and Formal Analysis tools exist which can determine which regions of a circuit are exercised during specific execution states. By incorporating an Automated Test Tool into the Syncopation flow, the paths activated by each state in an arbitrary FPGA circuit could be identified and measured to determine the minimum clock period, enabling Syncopation.

Chapter 6

Conclusions

This thesis presents Syncopation, a novel adaptive clock management strategy to improve the performance of High-Level Synthesis-generated user circuits by dynamically adjusting the clock period. Syncopation is made possible by leveraging three key observations. First, computational path delay variation exists in HLS circuits due to complexities in instruction scheduling algorithms. Second, not every path in a design is active in every clock cycle, and therefore the clock period only needs to be long enough to accommodate the active paths. Finally, Syncopation leverages detailed instruction schedules already automatically generated by HLS tools to determine exactly which paths are active in each state of the circuit's operation.

By combining these observations with a customized post-place-and-route timing analysis techniques, our results show that it is possible to tune the clock period on a cycle-by-cycle basis using only a small amount of additional instrumentation. Additionally, we show that our timing analysis technique can reveal detailed path timing information that can be used to target iterative synthesis passes to further improve performance in a method we call Enhanced Synthesis.

In this thesis, we detailed the key design elements which enable the Syn-

copation adaptive clock management technique. In particular, we presented the instrumentation, mechanism and performance of both Current State and Next State Syncopation, which monitor the user State Machines using different metrics to determine the per-cycle clock tuning. Additionally, we described the Divisor Memory, Divisor Selection Logic, and Clock Generator Instrumentation in detail.

Overall, we show that Syncopation can be used to improve performance of our benchmark circuits between -18 to +47% (by 3.2% on average). Furthermore, the Enhanced Synthesis technique improves design performance between -10 to +50% (10.3% on average) over Syncopation alone.

Over time, making high-performance circuit development easier and more accessible to designers without low-level hardware experience will require better tools to exploit limitations of the existing heuristic algorithms. Syncopation is an automated tool which enables additional performance with no changes to standard HLS and place-and-route tool algorithms, and does not require additional synthesis time unless Enhanced Synthesis is performed. Even higher performance is possible by developing custom toolflows to automatically employ Syncopation without the need for synthesis directives. Additionally, the cycle-by-cycle adjustment of the design clock period could easily be extended to other applications, such as security. Detailed experiments implementing these tools were out of scope for this thesis, but will be interesting areas for future work.

Bibliography

[1] Achronix Semiconductor Corporation. https://www.achronix.com. Accessed: 2020-05-01.

[2] Intel. https://www.intel.ca/. Accessed: 2020-05-01.

[3] Lattice Semiconductor. https://www.latticesemi.com. Accessed: 2020-05-01.

[4] Microsemi. https://www.microsemi.com. Accessed: 2020-05-01.

[5] Xilinx. https://www.xilinx.com/. Accessed: 2020-05-01.

[6] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/, 2020.

[7] V. Betz and J. Rose. Cluster-based logic blocks for FPGAs: Area-efficiency vs. Input sharing and size. In *Proceedings of CICC 97 - Custom Integrated Circuits Conference*, pages 551–554, 1997.

[8] Jacob A. Bower, Wayne Luk, Oskar Mencer, Michael J. Flynn, and Martin Morf. Dynamic clock-frequencies for FPGAs. *Microprocessors and Microsystems*, 30:388–397, 2006.

[9] A. Brant, A. Abdelhadi, D. H. H. Sim, S. L. Tang, M. X. Yue, and G. G. F. Lemieux. Safe Overclocking of Tightly Coupled CGRAs and Processor Arrays using Razor. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 37–44, 2013.

[10] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13, 09 2013.

[11] K. Chae, S. Mukhopadhyay, Chang-Ho Lee, and J. Laskar. A dynamic timing control technique utilizing time borrowing and clock stretching. In *IEEE Custom Integrated Circuits Conference 2010*, pages 1–4, 2010.

[12] Y. T. Chen, J. H. Kim, K. Li, G. Hoyes, and J. H. Anderson. high-level synthesis techniques to generate deeply pipelined circuits for fpgas with registered routing. In *2019 International Conference on Field-Programmable Technology (ICFPT)*.

[13] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. Combining Dynamic & Static Scheduling in High-Level Synthesis. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 288–298, New York, NY, USA, 2020. Association for Computing Machinery.

[14] J. Constantin, A. Bonetti, A. Teman, C. Müller, L. Schmid, and

A. Burg. DynOR: A 32-bit microprocessor in 28 nm FD-SOI with cycle-by-cycle dynamic clock adjustment. In *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, pages 261–264, Sep. 2016.

[15] L. de Souza Rosa, V. Bonato, and C. Bouganis. Scaling Up Loop Pipelining for High-Level Synthesis: A Non-iterative Approach. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 62–69, Dec 2018.

[16] X. Dong and G. G. F. Lemieux. PGR: Period and glitch reduction via clock skew scheduling, delay padding and GlitchLess. In *2009 International Conference on Field-Programmable Technology*, pages 88–95, 2009.

[17] S. Dutt and O. Shi. A Fast and Effective Lookahead and Fractional Search Based Scheduling Algorithm for High-level Synthesis. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 31–36, 2018.

[18] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 7–18, 2003.

[19] J. P. Fishburn. Clock Skew Optimization. *IEEE Transactions on Computers*, 39(7):945–951, 1990.

[20] A. Gheolbănoiu, L. Petrică, and S. Coţofană. Hybrid adaptive clock management for FPGA processor acceleration. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1359–1364, March 2015.

[21] K. Gibson, E. Roorda, D. H. Noronha, and S. Wilton. Syncopation: Adaptive Clock Management for HLS-Generated Circuits on FPGAs. In *2020 30th International Conference on Field Programmable Logic and Applications (FPL)*, 2020.

[22] Z. Gu, W. Wan, and C. Wu. Latency Minimal Scheduling with Maximum Instruction Parallelism. In *2019 IEEE 13th International Conference on ASIC (ASICON)*, pages 1–4, 2019.

[23] S. Hadjis, A. Canis, R. Sobue, Y. Hara-Azumi, H. Tomiyama, and J. Anderson. Profiling-driven multi-cycling in FPGA high-level synthesis. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 31–36, 2015.

[24] Mike Hutton. Understanding How the New Intel HyperFlex FPGA Architecture Enables Next- Generation High-Performance Systems. 2017.

[25] Intel. Verilog HDL Synthesis Attributes and Directives. https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/vlog/vlog_file_dir.htm, 2017.

[26] Intel. Intel Ships First 10nm Agilex FPGAs. https://newsroom.intel.com/news/intel-ships-first-10nm-agilex-fpgas/, Aug. 2019.

[27] Intel. Intel® Quartus® Prime Pro Edition User Guide: Design Optimization. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-design-optimization.pdf, 2020.

[28] D. Jayasinghe, A. Ignjatovic, and S. Parameswaran. RFTC: Runtime Frequency Tuning Countermeasure Using FPGA Dynamic Reconfiguration to Mitigate Power Analysis Attacks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2019.

[29] T. Jia, R. Joseph, and J. Gu. 19.4 An Adaptive Clock Management Scheme Exploiting Instruction-Based Dynamic Timing Slack for a General-Purpose Graphics Processor Unit with Deep Pipeline and Out-of-Order Execution. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 318–320, Feb 2019.

[30] T. Jia, R. Joseph, and J. Gu. An Instruction-Driven Adaptive Clock Management Through Dynamic Phase Scaling and Compiler Assistance for a Low Power Microprocessor. *IEEE Journal of Solid-State Circuits*, 54(8):2327–2338, Aug 2019.

[31] L. Josipovic, P. Brisk, and P. Ienne. From C to elastic circuits. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 121–125, 2017.

[32] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically Scheduled High-Level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA

'18, page 127–136, New York, NY, USA, 2018. Association for Computing Machinery.

[33] V. Lapotre, P. Coussy, C. Chavet, H. Wouafo, and R. Danilo. Dynamic branch prediction for high-level synthesis. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–6, Sep. 2013.

[34] Legup. LegUp 4.0 Programmer's Manual. http://legup.eecg.utoronto.ca/docs/4.0/programmermanual.html.

[35] G. Lhairech-Lebreton, P. Coussy, and E. Martin. Hierarchical and Multiple-Clock Domain High-Level Synthesis for Low-Power Design on FPGA. In *2010 International Conference on Field Programmable Logic and Applications*, pages 464–468, 2010.

[36] T. Liang, J. Zhao, L. Feng, S. Sinha, and W. Zhang. Hi-ClockFlow: Multi-Clock Dataflow Automation and Throughput Optimization in High-Level Synthesis. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2019.

[37] Alexander (Sandy) Marquardt, Vaughn Betz, and Jonathan Rose. Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, page 37–46, New York, NY, USA, 1999. Association for Computing Machinery.

[38] T. Marty, T. Yuki, and S. Derrien. Enabling Overclocking Through Algorithm-Level Error Detection. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 174–181, 2018.

[39] J. A. Nestor and G. Krishnamoorthy. SALSA: A New Approach to Scheduling with Timing Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1107–1122, 1993.

[40] B. Pontikakis, H. T. Bui, F. Boyer, and Y. Savaria. A Low-Complexity High-Speed Clock Generator for Dynamic Frequency Scaling of FPGA and Standard-Cell Based Designs. In *2007 IEEE International Symposium on Circuits and Systems*, pages 633–636, May 2007.

[41] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 13–24. IEEE Press, June 2014. Selected as an IEEE Micro TopPick.

[42] R. Ragavan, C. Killian, and O. Sentieys. Adaptive Overclocking and Error Correction Based on Dynamic Speculation Window. In *2016*

IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pages 325–330, 2016.

[43] O. Ragheb and J. H. Anderson. High-Level Synthesis of FPGA Circuits with Multiple Clock Domains. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 109–116, April 2018.

[44] D. S. H. Ram, M. C. Bhuvaneswari, and S. M. Logesh. A Novel Evolutionary Technique for Multi-objective Power, Area and Delay Optimization in High Level Synthesis of Datapaths. In *2011 IEEE Computer Society Annual Symposium on VLSI*, pages 290–295, 2011.

[45] K. Shi, D. Boland, and G. A. Constantinides. Accuracy-Performance Tradeoffs on an FPGA through Overclocking. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 29–36, 2013.

[46] K. Shi, D. Boland, E. Stott, S. Bayliss, and G. A. Constantinides. Datapath Synthesis for Overclocking: Online Arithmetic for Latency-Accuracy Trade-offs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[47] Deshanand P. Singh and Stephen D. Brown. Constrained Clock Shifting for Field Programmable Gate Arrays. New York, NY, USA, 2002. Association for Computing Machinery.

[48] V. Subramanian, M. Bezdek, N. D. Avirneni, and A. Somani. Superscalar Processor Performance Enhancement through Reliable Dynamic

Clock Frequency Tuning. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 196–205, June 2007.

[49] D. Thomas and M. Shanmugasundaram. A Survey on Different Overclocking Methods. In *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, pages 1588–1592, 2018.

[50] Stephen M. Williams and Mingjie Line. Reactive Signal Obfuscation with Time-Fracturing to Counter Information Leakage in FPGAs. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 322, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*, pages 1192–1195, May 2008.

[52] H. Zheng, S. T. Gurumani, L. Yang, D. Chen, and K. Rupnow. High-level synthesis with behavioral level multi-cycle path analysis. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, 2013.