Sharing Silicon: Improving FPGA Accessibility

by

Matheus de Carvalho

Professor Duane Bailey, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

May 16, 2018

Abstract

Field Programmable Gate Arrays (FPGAs) are hardware circuits that can be programmed to perform any digital circuit functionality. This flexibility makes them a useful tool in improving system and application throughput and energy efficiency. In this way, FPGAs can be part of the solution in rethinking our current model of computation to better support high performance computing and avoid Dark Silicon. In this work, we attempt to improve FPGA accessibility and offer motivation for a more widespread adoption of these circuits by building on previous efforts in this area, with special emphasis to the Xilinx PYNQ FPGA+CPU development board. We explain in detail how to generate a working FPGA circuit for the PYNQ board from a software-specified function using the Xilinx Vivado Design Suite. We offer advice on efficient FPGA design, and demonstrate the acceleration potential of FPGAs: we find that an FPGA-implemented function runs up to 120× faster than its Python counterpart. We also demonstrate the potential of FPGAs to implement conservation cores by explaining the design of an FPGA circuit that supports five simultaneous cores that can be switched at runtime.

Acknowledgements

This work was only possible thanks to the continuous support and encouragement of my advisor, Prof. Duane Bailey. His guidance has helped me not only in the research and writing of this work, but in shaping by undergraduate experience and career.

I would also like to thank my second reader, Prof. Jeannie Albrecht, for her suggestions and insight, and Mary Bailey, our Systems Support Specialist, for all the expertise and support which allowed me to seamlessly run my experiments.

Contents

List of Figures

List of Tables

1 Introduction

In the current data-driven landscape of computer technology, high performance computing has become increasingly relevant. Throughput, speed and energy efficiency become a concern when large amounts of data need to be constantly processed in real-time within large-scale, computationally intensive systems. These concerns are magnified when processor capacity is undercut by the transistor utilization wall [14]. Computer Scientists are then compelled to think of new computational models, that promote higher throughput and optimize processor use. To this end, hardware supported computation offers great promise, with field programmable gate array (FPGA) circuits standing out among available solutions for acceleration and energy efficient computation.

FPGA circuits are general purpose hardware that can be programmed to implement any digital circuit functionality. An FPGA is a collection of lookup tables (LUTs)—which allow implementation of any kind of digital logic—and flip-flops (FFs)—which, as registers, maintain the result of LUT computations through multiple clock cycles. The FPGA is programmed by a user-defined bitstream that specifies the contents of and inputs to each LUT and the wiring of FFs on the chip's fabric. This allows for great flexibility since FPGAs can be configured to perform low-level functions (e.g. matrix multiplication), to implement entire algorithms (e.g. Monte-Carlo simulation), or to virtually emulate the functionality of another, more expensive, dedicated chip. Software-implemented functionality can often be accelerated—or at least gain energy efficiency—if implemented on hardware. FPGAs make this possible for most functions without the need for dedicated circuit design. When used to implement dedicated functionality, FPGAs effectively become conservation cores that free-up the processor and push back on the utilization wall [17].

An obviously novel feature of FPGAs is the possibility of reconfiguration. A single circuit can be set up during its lifetime to serve many different purposes. Reconfiguration can take place both statically and dynamically. These chips can then "shape-shift" millions of times a second during runtime to execute just as many different functions. Runtime reconfiguration provides the illusion of a larger fabric than is actually available. When coupled with sharing of state between reconfigurations, FPGAs can emulate three-dimensional stacked chips—a theorized technology that has yet to be implemented [4]. Overall, an FPGA circuit lends itself to applications that require high throughput and low-level bit manipulation: examples include real-time pattern matching, signal processing, NP-hard optimization, networking, scientific computing, etc.

Despite all their computational potential, FPGA use was only firmly established around 2010, nearly three decades after their introduction in mid-1980s [15]. There are many reasons for this, including the fact that until recently processor capacity could reasonably keep up with computational intensity, reducing the comparative advantage of FPGA use. However, we believe that mainstream use of FPGA circuits have been significantly inhibited due to the complexity of bitstream and FPGA design.

Designing FPGA circuits involves a sound understanding of complex hardware description languages (HDL) such as Verilog and VHDL, or of vendor-specific high level synthesis (HLS) tools. This restricts FPGA development to a few hardware programmers, and neglects software developers who perhaps could benefit the most from the technology. Xilinx has attempted to make FPGAs more accessible by introducing PYNQ, an FPGA + CPU development board interfaced by Python. PYNQ allows Python applications to interface the ZYNQ FPGA for accelerated computation. Although PYNQ has been shown to reduce the design time of applications that use FPGAs [12], it is far from making actual FPGA design accessible. PYNQ and its community provide a library of bitstream overlays that are ready for use in software applications. Software programmers are advised not to create their own bitstreams, but to use the ones created by hardware programmers in a way that suits their applications [18]. However, trying to make do with general-purpose bitstream overlays can arguably be counterproductive, since allowing software engineers to create hardware-accelerated functionality specific to their application would ultimately allow them to fully harness the power of FPGAs.

We believe FPGAs can be more widely used provided FPGA design is made clearer, and its benefits well demonstrated. In an attempt to further the efforts of Xilinx and others in increasing FPGA accessibility, we set out to explain how to generate a bitstream and PYNQ Overlay from scratch and to demonstrate the potential of FPGAs as function accelerators and conservation cores. To this end, we explore the idea of implementing multiple user-defined functions—with runtime switching—in a single FPGA circuit. These functions, performed by the FPGA instead of the processor, serve as "conservation cores", and hopefully allow us to establish FPGAs as a simple, low-energy and flexible alternative to dedicated c-cores [17].

The following chapters are organized as follows: Chapter 2 outlines previous work that served as inspiration for this paper; Chapter 3 describes the PYNQ workflow by walking the reader through the steps of generating a bitstream and PYNQ overlay from scratch; Chapter 4 offers motivation for FPGA adoption by demonstrating the benefits of FPGAs both as accelerators and conservation cores through the implementation of a multi-function bitstream; Chapter 5 evaluates the strengths and weaknesses of our work; Chapter 6 draws conclusions and suggests future work.

2 Related Work

In thinking about how to design a bitstream and use it with PYNQ, it is fundamental that we understand how the platform works. Many of the ideas concerning system virtualization also provide useful insight into how to use reconfiguration to extend FPGA functionality. Finally, it is important that we know what some of the most useful and simple applications for FPGAs are, so we can understand how and when to compose them together. In this section we describe previous literature that in this areas.

2.1 Successful abstraction in systems

UNIX [11] is a good example of an evolved system that is simple and accessible. The success of the UNIX operating system is largely due to its open source code base, which allows for straightforward development and distribution. This comes in stark contrast with FPGA development, where architectures are secretive and vendor-specific, stifling development. The `shell` is a really interesting component of the system. It is used to execute small, single-task programs composed together to accomplish complex tasks. This idea very much relates to our goal of providing and composing together multiple FPGA functionalities that can be switched between by a single user. We believe that the development of simple, composable tools is fundamental to the development of a robust and successful system.

While UNIX provides useful insight into module composition, Mach [10] demonstrates the power of abstraction. It is a multiprocessor kernel designed for compatibility across multiple operating systems (OS). Abstractly, Mach is a layer compatible with multiple OS implementations overlaid on top of a single kernel aimed at solving

OS-kernel compatibility issues. It introduces the notion of a memory object to be created and managed by application programs. The applications—in this case the various OS implementations—can then control the resource allocation and scheduling for its own memory objects without worrying about kernel specifics. Mach interfaces are, for the most part, independent of the target environments, allowing for multiple operating systems to run simultaneously in the same machine, provided robust synchronization between servers. Mach is an important example of virtualization used to improve interface robustness. Mach abstracts away kernel specifics from the client OSes, allowing for a single machine to run multiple operating systems. In a similar manner, we believe that a successful hardware design environment will support runtime reconfiguration and coordination between circuits separated by time and/or space.

2.2 PYNQ board and interface design

In effectively using PYNQ, we must understand the benefits of its Python interface. The Python "shell" allows users to immediately interact with the language without going through the edit and compile cycle. The IPython project [8] aims to enhance the Python shell and also provide facilities for interactive distributed and parallel computing, with a comprehensive set of tools for building special-purpose interactive environments for scientific computing. It allows access to all session state, a control system via UNIX-like commands, OS access by allowing commands to be executed directly by the underlying system, dynamic inspection and help, and the ability to run code from files. Beyond these basic features, IPython includes robust error handling, input syntax processing, and tab completion, which makes interaction more straightforward. IPython also offers startup flags that let users choose graphical interface toolkits from startup, and configure the environment to run in a non-blocking manner. IPython's basic components make no assumption about communication models, data distribution or network protocols. By separating core functionality, networking and asynchronous control, distributed and parallel systems can be easily implemented in IPython. We can easily see how IPython's features enhance the Python "shell" to provide even more development power. IPython is the predecessor of Jupyter, which is, in turn, interfaced with PYNQ, providing a simpler, faster development environ-

ment as well as allowing for detailed web-based Jupyter notebook contributions to PYNQ by its community.

Schmidt et al. [12] demonstrate well the benefits introduced by the interfacing of Python with the ZYNQ FPGA on the PYNQ board. Even though PYNQ is an attempt to make FPGAs more accessible, it does not provide high-level synthesis or porting of Python applications directly into the FPGA fabric. What PYNQ does provide is an overlay framework to support interfacing with the board's IO, where any custom logic must be created and integrated by the developer via the design of a bitstream suited to their application. Once a desired bitstream is created, PYNQ makes it easy to download it onto the chip and to interface it with the Python application. PYNQ supports many Python functionalities such as interactive debuggers, profiling and measurement tools, libraries and packages. PYNQ also supports MMIO to configure the connectivity of different kernels and DMA cores. DMA can be performed efficiently between memory and the programmable fabric, where the DMA engine is initialized as a buffer that can be interfaced by the Python application as needed. The authors find that the Python interface eliminates the portability complexity. System complexity is also significantly reduced due to the APIs for programming the bitstream and reading and writing data through MMIO and DMA. Development time is minimized due to the Python interface. Python developers are able to take advantage of the extensive set of libraries and packages available for Python to obtain substantial performance gains. In the Edge Detection application studied, the Python OpenCV version obtained an 11.43x speedup over the C version. The hardware accelerated core combined with Python obtained a 30.2x speedup compared to single-threaded C. The development efficiency introduced by Python is what drew us to choose PYNQ for the purposes of this research. The board naturally serves our goal of lowering the bar for FPGA circuit development.

Central to this paper is the work by Yang Tavares [13]. Tavares provides a very useful tutorial on creating a simple overlay for PYNQ from scratch, starting with a function defined in C/C++ down to a bitstream file. His work, coupled with Overlay documentation by Xilinx [18], allow us to explain the PYNQ workflow step by step. Building on this, we are able to define a bitstream to support multiple functions that can be switched between by the user and ultimately reconfigured at runtime.

2.3 FPGA reconfiguration

Tessier et al. [15] provide a comprehensive survey of reconfigurable computing. The authors recognize the potential of reconfigurable architectures to enable the real-time performance of tasks without resorting to static ASICs. FPGAs were introduced in the mid-1980s, but their use for computation and communication was only firmly established recently, as we now face an increasing demand for reconfigurable, general purpose hardware, capable of accelerating real-time computation and algorithms. One of the most appealing features of reconfigurable architectures such as FPGAs is the possibility of time-multiplexing. Analogous to system virtualization, reconfiguration allows a small FPGA fabric to be time-multiplexed into an apparently much larger computing device. Reconfiguration also allows for greater fault tolerance. Run-time reconfiguration is an important feature of FPGAs that enables persistence and run-time context switching, allowing for one chip to perform many different functions within the lifespan of a single application. The authors outline many important applications that take advantage of reconfigurable computing, some of which are: signal and image processing, finance, security, NP-hard optimization, pattern matching, networking, numerical and scientific computing, and molecular dynamics. These are all useful applications we might expect to implement in our bitstream overlay.

In composing such applications onto a single chip for adaptable run-time use, we need to take advantage of FPGA reconfiguration efficiently. Liu and Wong's work [7] is as an important reference in thinking about efficient FPGA reconfiguration. They show that a single physical FPGA device supports large logic designs partitioned into different time-multiplexed configurations. To ensure the correctness of execution and reduce reconfiguration overhead, it is crucial to have a good strategy for partitioning the logic design. The authors present a network-flow based approach for multiway partitioning of the FPGA chip. Their model of time-multiplexed communicating logic consists of two parts: a finite set of combinatorial logic units—the FPGA LUTs—and a finite set of memory elements. Basically, the memory elements are used to store the state of computation between reconfigurations of the logic units. The authors then devise a way of partitioning the combinatorial logic units and memory elements as to minimize reconfigurations of a logic design. They do so by employing the network-flow technique for finding the min-cut, max-flow bipartitioning between

the portions of the design on and off chip. They show that this technique can be extended to multiway partitioning of the logic design.

2.4 Specifying reconfigurable computing

IBM's Liquid Metal Project [2] recognizes that FPGAs represent great computing potential that is widely unused in light of the complexity of design and platform dependency. FPGA programming still lacks sufficient abstraction. Currently the circuits are programmed in HDL, which requires robust reasoning about low-level building blocks of logic. This excludes all but hardware programmers. It is clear that a higher level of abstraction for FPGA programming is necessary to increase adoption and reduce time-to-market for FPGA programs. The lack of abstraction coupled with the complexity of designing, synthesizing, compiling and debugging HDL prevents the wider computing community from contributing to and developing in the promising space of reconfigurable hardware. FPGA vendors have attempted to lower the barrier to entry by developing high level synthesis (HLS) tools, which convert an application into a digital circuit compiled into HDL. However, HLS frameworks vary from vendor to vendor and even from chip to chip. Therefore, programming for cross-platform compatibility using HLS requires a great deal of conflict-resolution and code adaptation, which is perhaps even more complex than writing HDL directly. IBM's Liquid Metal Project aims at providing a single and unified language for programming heterogeneous architectures.

The Lime language [1] is the main contribution of IBM's Liquid Metal project, and served as fundamental inspiration for this thesis. Lime is an object-oriented language designed to compile to heterogeneous architectures, including reconfigurable hardware. Its creators believe that achieving high performance computing while minimizing power consumption will require the use of heterogeneous architectures with specialization of hardware resources, exploring the GPU and reconfigurable hardware in addition to multiple CPU cores. Such heterogeneous systems are very difficult to program since, while CPUs are programmed at a high-level, with a diversity of languages, reconfigurable hardware, like FPGAs, must still be programmed in HDL (Verilog or VHDL). The main contribution of Lime is then the provisioning of a single,

fairly familiar language that can be compiled to all these platforms and is capable of dynamically dividing programs across different computational elements. Lime runs on top of the JVM, which allows for adaptive run-time recompilation and cross-platform compatibility. Lime also integrates functional programming—well-suited to hardware—with object-oriented programming for ease of use. Lime takes application code that includes software, graphical and circuit specifications, and compiles it to Java bytecode, C and Verilog, the latter of which is then translated into FPGA bitstreams. This approach then allows software developers, familiar with object-oriented Java programming, to specify bitstreams.

Although Lime is promising and perhaps the most comprehensive attempt to bring FPGA programming into the software engineering world, it is only able to take a high-level description and compile it to HDL. Thereafter, developers are still subject to excessively long bitstream synthesis. The authors of [2] claim the only way to ensure mainstream use of FPGAs is for vendors to open up their architecture details so languages like Lime can compile all the way down to bitstreams and enhance productivity. While FPGA architecture details remain secret, our approach attempts to explain in more detail how to generate efficient bitstream overlays from HDL in hopes to make it clearer to software programmers how to to design FPGA circuits from scratch, starting from functions defined in familiar software programming languages.

In then thinking about the integration of different functions into a single system by FPGA dynamic reconfiguration, the ReCoBus communication architecture [5] provides useful insight. First implemented on Spartan-3 devices, the ReCoBus enables the linking of configuration bitstreams in a way analogous to the plugging of cards into backplane bus sockets. In the FPGA case, however, modules are placed in one or more resource slots, which are the smallest atomic pieces of the FPGA fabric. ReCoBus is designed to provide direct interfacing to the on-chip bus and to other modules or I/O pins, with low logic overhead and high performance. Its specification is designed to guarantee glitch-less runtime reconfiguration, where reconfiguring one module won't affect the other loaded modules. We hope to draw from the ReCoBus design to compose together and exchange bitstreams safely at runtime.

The Abax 3PLD devices developed by Tabula [4] are another interesting example of safe dynamic FPGA reconfiguration and context switching. The Abax 3PLDs are impressive programmable logic devices that emulate three-dimensional stacked FPGA chips. This emulation is achieved by supporting rapid reconfiguration of two-dimensional fabrics, with up to 1.6 billion complete fabric reconfigurations per second. Rapid reconfiguration enables the devices to use a single gate for up to 8 different functions, so smaller and cheaper Tabula chips can match the capacity of larger and more expensive FPGAs. Each Abax 3PLD chip consists of 8 folds of a single fabric put together as a torus, each with separate configurations preloaded into SRAM upon initialization. These folds share "time vias", which pass the state of each wire to the next fold in a round robin fashion by using a transparent latch. As an analogy, Abax 3PLD works like an 8-terminal mainframe. Each user has the illusion that they have a dedicated machine, while processes from the 8 different terminals each get a time slice of the clock scheduled by the server. Overall the Abax 3PLD project by Tabula gives us good insight into how time-sharing can be done on chip.

2.5 Applications

Work by Glette et al. [3] is a good example of the use of FPGA reconfiguration to implement pattern recognition, a popular application for FPGAs. The authors propose an implementation for a run-time evolvable hardware classifier system. They use the FPGA lookup tables as shift registers to reconfigure the FPGA for different classifications, instead of reconfiguring the FPGA virtually at a high-level. This strategy effectively saves FPGA resources, reducing the circuit size by a factor of 3. This allows for the implementation of a larger, more accurate classifier, with 97.5% recognition accuracy for face images. This suggests that accurate pattern recognition is a compact and useful application for FPGA circuits, and perhaps components of that computation should be natively supported by our abstraction module.

Microsoft's Catapult Project [9] makes use of FPGAs and reconfiguration to provide high computational capabilities to datacenters. FPGAs allow for flexibility, power efficiency and persistence at a low cost. Using the circuits they designed a composable, reconfigurable fabric to accelerate portions of large-scale page ranking

services. Each instantiation of the fabric consists of a 6x8 2-D torus of high-end Stratix V FPGAs embedded into a half-rack of 48 machines. One FPGA is placed into each server, accessible through PCIe, and wired directly to other FPGAs with pairs of 10 Gb SAS cables. This design allows for one FPGA to be disabled to save power or to be statically reconfigured during system maintenance. Since the FPGAs are interconnected, if a given server's FPGA is disabled its neighboring server's FPGAs can easily absorb the server's workload and guarantee persistence. This way the reconfigurable circuits provide persistence to the servers, which can be reconfigured without denying service. They have determined that under high load, the large scale reconfigurable fabric improves the ranking throughput of each server by a factor of 95% for a fixed latency distribution— or, while maintaining equivalent throughput, reduces the tail latency by 29%. The Catapult project provides useful insight into the potential of FPGA reconfiguration and on the design of systems that support it.

The Binarized Neural Network (BNN) introduced by Umuroglu et al. Work from [16] is a great example of an application tailored to take full advantage of the FPGA circuit. Given the redundancy of convolutional neural networks, high classification accuracy can be obtained even when weights and activations are reduced from floating point to binary values. The combination of low-precision arithmetic and a small memory footprint presents a unique opportunity for fast and energy efficient image classification using FPGAs, which have much higher peak performance for binary operations compared to floating point. Umuroglu et al. find that their hardware implementation of BNN on FPGA is able to classify 12.3 million images per second with negligible power and latency overheads and high accuracy: 95.8%, 80.1% and 94.9% on the benchmark datasets. The use popcount-accumulation instead of simple addition spares the chip from signed arithmetic which requires twice as many LUT and FF resources. The use of thresholding allows for the computation of output activation using unsigned comparison, which requires considerably fewer LUTs and resources as compared to its signed counterpart. The authors also choose to lower convolutions to matrix multiplications, which can be easily programmed into the FPGA. The matrix design allows for easy folding of the network. This is important since large BNNs will not fit on a single FPGA fabric, so it must be time-multiplexed to take advantage of dynamic reconfiguration. The BNN implementation serves to show how to implement applications efficiently on the FPGA fabric.

2.6 Summary

As we now approach the boundaries of computing at an increasing rate, hardware support for efficient computation becomes crucial. To this end, FPGAs offer great potential. These circuits are currently being programmed to offer specific, hardware-accelerated, functionality to large scale applications, as exemplified in the works of Glette et al. [3], Tabula [4], Microsoft Catapult [9] and Umuroglu et al.[16]. These applications largely rely on the runtime reconfiguration of FPGA circuits, which—as discussed by Tessier et al.[15] and Tabula [4]—extend the functionality of FPGAs to the point that time-sharing can be implemented, with a single chip being used for many different purposes at once. The use of FPGA circuits then leads to significant performance gains and/or power-savings in large, complex systems. Despite their demonstrated cost-efficiency and performance gains, the widespread adoption of FPGAs has been prevented by the complexity of FPGA development, as discussed by Bacon et al. [2]. Recognizing this to be a problem, IBM's Liquid Metal project [2], has developed Lime [1], a single high-level programming language that is able to compile applications to CPU, GPU and hardware language components. Lime then makes it easier for software developers to specify hardware programs that can then be translated into FPGA bitstreams.

Inspired by Liquid Metal [2], this paper also attempts to make FPGA development more accessible. We draw from the concepts of system abstraction from UNIX [11] and Mach [10] to specify simple, composable functions on a single FPGA bitstream. These can be used at a high level and reconfigured at runtime by developers in Python, provided the use of PYNQ and its Overlay driver libraries. As seen in the IPython project [8] and in work by Schmidt et al. [12], PYNQ naturally lends itself to greater adoption by enabling developers to leverage Python's extensive, simple functionality to interface the ZYNQ FPGA. Liu and Wong's network-flow specification [7], Koch et al.'s ReCoBus architecture [5] and Tabula's Abax 3PLDs [4] outline how we may support efficient FPGA reconfiguration and time-sharing to indefinitely extend the functionality of the single FPGA chip. Literature on available and useful applications, such as Glette et al.'s classifier [3] and Umuroglu et al.'s BNN [16], will give us ideas of complementary functionality that can be implemented together in a single bitstream. The process of creating this multi-function bitstream overlay from

functions specified in software programming languages will be carefully documented by our paper in hopes to offer software programmers a good idea of how to incorporate FPGA design into their applications and systems using PYNQ.

3 The PYNQ Workflow

As seen in the works discussed in Chapter 2, the use of FPGAs can offer accelerated, cheap computing and energy efficiency to a wide variety of systems and applications. When these are implemented mainly in software, it is important that software engineers are able to understand the trade-offs of incorporating FPGAs into their applications, and to design efficient FPGA implementations that suit their needs. In this section, we elaborate on the PYNQ workflow and explain in detail how to create a working FPGA bitstream from software-specified functions by using Xilinx's Vivado HLx Design Suite. This section should serve as as stepping stone for those who are unfamiliar with the PYNQ FPGA design flow and integration, and is largely based on work by Tavares [13], complemented by the Vivado and PYNQ Overlay documentations [18, 19].

The PYNQ board is divided into three main components: the processing system (PS), the programmable logic (PL), and the board peripherals. The PS is a dual core ARM processor that is in charge of scheduling and performing computation. The PL is a Zynq XC7Z020 FPGA circuit. The PYNQ user is able to program the PL to perform any digital circuit functionality. The PL must be interfaced with the PS, which will enable the user to schedule and control computation in the PL. The peripherals provide support for many useful functionalities such as network access, audio and video. Note that the PYNQ package provides two ready-to-use PL bitstream overlays: `base` and `logictools`. `Base` is the default board bitstream and serves mainly to integrate and manage all the board peripherals. `Logictools` provides a pattern generator, a finite state machine generator, a boolean generator, and a trace analyzer, all of which can be composed by the user to create custom board logic. By using the libraries and drivers associated with these two bitstreams, the user can take advantage of accelerated FPGA computation without the need of independent design.

Xilinx makes the Vivado HLx projects for both overlays available as a basis for the creation of new bitstreams with different functionality. Other bitstream designs are also available in the PYNQ community.

3.1 High Level Synthesis

```
void hailstone(long n, int* r){
  long long cur = n;
  int count = 0;

  if(n < 1){
          *r = -1;
    return;
  }

  while(1) {
    if (cur % 2L) {
          if (cur == 1L){
                  *r = count;
                  return;
          } else {
                  cur = ((cur * 3L) +1L)/2L;
                  count += 2;
          }
        } else {
          cur /= 2L;
          count++;
    }
  }
}
```

Figure 1: Hailstone sequence implemented in C

The first step in defining a working bitstream overlay for PYNQ is to create a target function. For the purposes of this section, we decided to recreate Isaiah Leonard's C implementation of the hailstone sequence, as seen in Figure 1 [6]. This function computes the number of compositions of the function

$$f(x) = \begin{cases} x/2, \text{ if } x \text{ is even} \\ 3x+1, \text{ if } x \text{ is odd} \end{cases}$$

necessary to reach 1 given n. Along with the target function, we must also write what Xilinx calls a "test bench", a function that calls and tests our target function by comparing its output to the expected output, and returning an error code—in the case of C, returning 1—if the outputs do not match.

Once the target function and its test bench are created, they are imported into Vivado HLS—Xilinx's High Level Synthesis (HLS) tool which can be obtained with the WebPack license included with the PYNQ board. We set the hailstone function as our target function and run the C simulation. This step will make sure the function works as expected. Once the C simulation passes, we are ready to set HLS "directives".

HLS directives serve many purposes, including specifying the type of hardware ports—if any—that should be associated with the target function's input and output, and optimizing loops or limiting their latency. Setting appropriate HLS directives is essential for an efficient FPGA implementation. As seen in Figure 2, to set directives within the Vivado HLS GUI, we select the C source code of our target hailstone function and open the Directives tab. We choose to specify the type "s_axilite" for the inputs and output of the target function. These are AXI4-Lite ports, which allow the circuit design—which in this case will be implemented in the PYNQ PL—to be controlled by the CPU (PYNQ PS) [20]. The ports in the AXI4-Lite interface have memory-mapped locations which allow for straightforward I/O management from a software library—in this case, the `pynq` Python module. Since the loop within hailstone has no static end condition, we cannot set optimization directives, such as loop unrolling. The Vivado HLS User Guide offers more detail on directive setting [20].

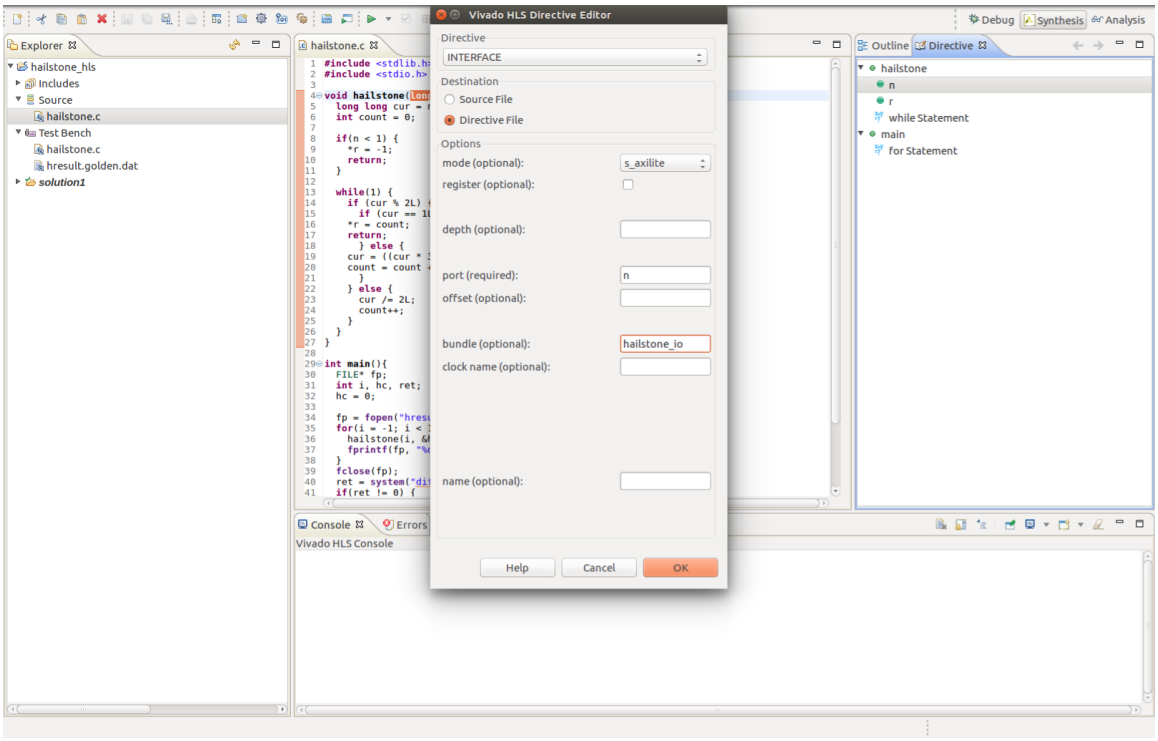

Figure 2: Setting port directives

After directives are set, the target function must be synthesized into a Register Transfer Level (RTL) description. This is a human-readable, text-based language that describes the function's implementation as a digital circuit. This is done within the Vivado HLS GUI by clicking Solution → C Synthesis → Active Solution. Once the function is synthesized, we are ready to export the RTL description of our hailstone function. We do that by clicking Solution → Export RTL. This will create Hardware Description Language (HDL) code for the hailstone function in both Verilog and VHDL.[1] These will be saved to the <solution>/impl/ip directory within our project, in what is referred to as Intellectual Property (IP) block.[2] We are now ready to move forward to generating the bitstream for our IP block.

[1]Verilog and VHDL are competing hardware definition languages. Vivado supports development with either HDL.

[2]Intellectual Property Blocks are the industry's approach to the encapsulation of logical models that may be distributed or incorporated into third party systems.

3.2 Bitstream Generation

Once we obtain an IP block described in HDL from Vivado HLS, we can use Xilinx's Vivado to integrate the ZYNQ processing system (PS) with the hailstone IP block in the programmable logic (PL) of the FPGA, and in this way effectively create a bitstream that programs the ZYNQ FPGA as a hailstone accelerator. A bitstream is the setting of Lookup Table (LUT) values and routing unit configurations. A single bitstream contains settings for thousands of individual units. In order to create a bitstream that works correctly on PYNQ, we must first certify that the PYNQ board files are downloaded and exported into Vivado as instructed in the Overlay Design Methodology documentation [19].

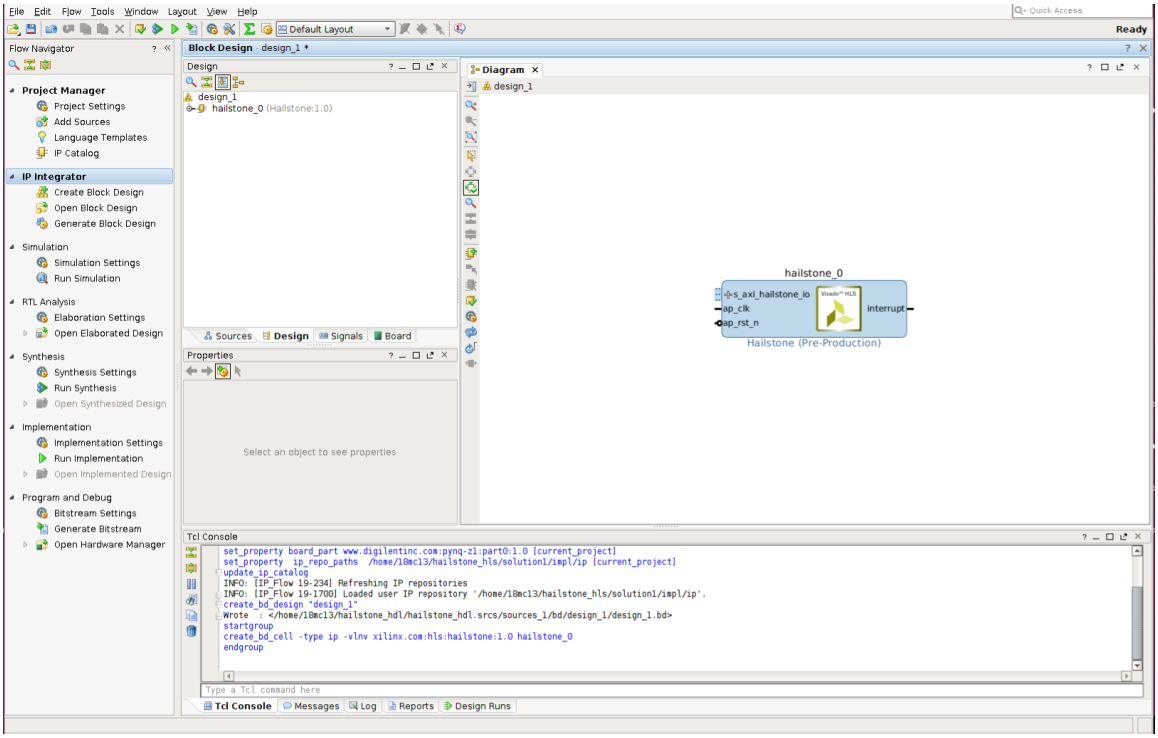

Figure 3: Hailstone IP block in block design

After the PYNQ board files are correctly installed, we can start a new Vivado project by selecting the PYNQ board. Once the project is created, under "Project Manager" we import our hailstone block by going into the IP Catalog and adding a new IP repository, which is the "ip" folder of our HLS project that contains HDL descriptions of our IP block. We then move on to creating the block design for the

FPGA. In the case of hailstone, the block design is simple: we just need to wire the ZYNQ processing system together with the hailstone block. As seen in Figure 3, we are able to find the hailstone block we just added to our project. The ZYNQ PS block is available to all Vivado projects for the PYNQ board. We include both blocks in our design, select "Run Block Automaton" and "Run Connection Automaton". These will automatically integrate both blocks and create the appropriate interface I/O ports. The resulting block design should look like Figure 4.

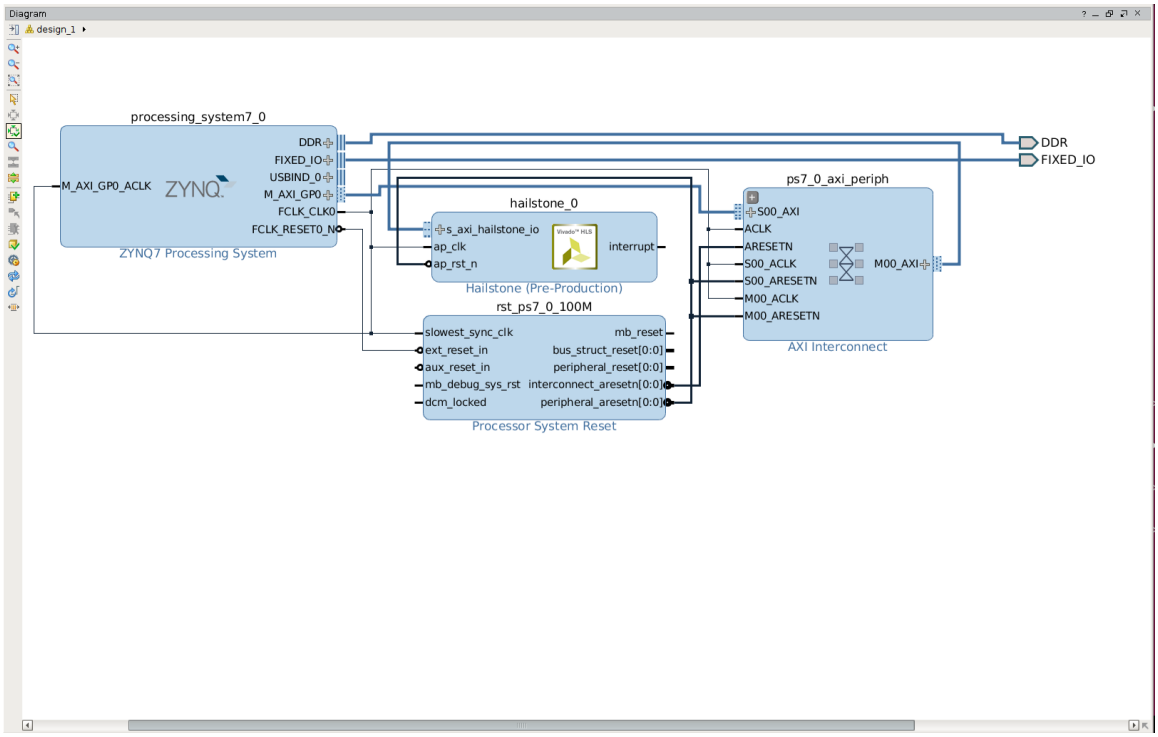

Figure 4: Completed block design

Now that we have integrated the HDL descriptions of the hailstone IP block and the ZYNQ processing system, we must wrap them into a single circuit description that can later be converted into a single bitstream. This is also done automatically by Vivado by right-clicking the source of the block design and selecting "Create HDL Wrapper." After wrapping our block design into a single description, we generate the bitstream for it by clicking on "Generate Bitstream" under "Program and debug." This will synthesize an initial block design. This is then optimized based on feedback from several runs of the system. Finally, a bitstream that describes the most efficient circuit is generated.

The final two steps are to export the tool command language (Tcl) and bitstream files for our circuit—these are both needed by PYNQ to overlay the bitstream onto the Zynq fabric using Python. Tcl is a scripting language that manages the technically complex process of communicating timing and place-and-route constraints of FPGA design. We write the Tcl file for our hailstone accelerator from the Vivado GUI by typing the command "write_bd_tcl hailstone.tcl" in the Tcl console. Once the Tcl file for our design is exported, it can be used as a script to recreate our FPGA design in Vivado, which is useful for batch processing. The bitstream file is exported as "hailstone.bit" in File → Export → Bitstream file. Note that the Tcl and the bitstream files must have the same name.

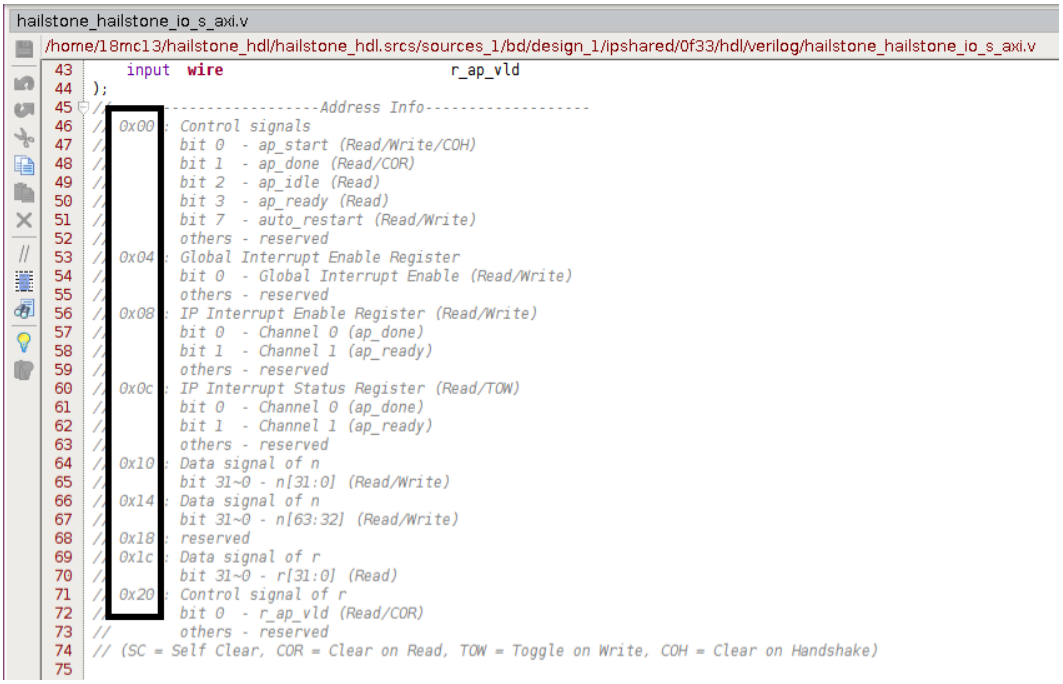

Figure 5: Hailstone signal addresses

After the bitstream is generated and the Tcl file is written, the "Address Editor" of our block design will show the physical address associated with each placed block—the Zynq PS and the hailstone IP. These are needed if you wish to communicate with the FPGA from the user application once the bitstream is downloaded. Equally as important are the addresses for the control, input and output signals of our hailstone IP block. These are found as comments in the HDL description of the hailstone IP IO which is in turn found under the source HDL wrapper (see Figure 5).

3.3 Interfacing the Bitstream

Now that we have the Tcl and bitstream files for the hailstone overlay, we are ready to leverage the power of PYNQ by using Python to download our bitstream onto the ZYNQ FPGA and manage it. We do so by using the Overlay library [18]. To create an Overlay object in Python we must give it the path to the bitstream file, which must be in the same directory as the corresponding Tcl file. The bitstream will then be downloaded by default. In Figure 6, we use a Jupyter notebook to create and download the hailstone overlay onto the FPGA.

```
In [1]: from pynq import Overlay
        from pynq import DefaultIP

        class HSDriver(DefaultIP):
            def __init__(self, description):
                super().__init__(description=description)

            bindto = ['xilinx.com:hls:hailstone:1.0']

            def hailstone(self, n):
                self.write(0x10, n) # write input
                self.write(0x00, 1) # start accelerator

                while self.read(0x00)&2 != 2: # wait for computation to finish
                    pass
                return self.read(0x1c) # return output

In [2]: bit_file = '/home/xilinx/jupyter_notebooks/hailstone_overlay/hailstone.bit'
        hs_ol = Overlay(bit_file, download=True) # download bitstream
        hs_ol.is_loaded()

Out[2]: True
```

Figure 6: Downloading and managing hailstone bitstream

After the overlay is successfully downloaded we can use Python to communicate with it in two ways: we can directly read from and write to the IP signal ports, or we can create a Python Driver that handles writing signals for us. In the former, we use the memory-mapped IO (MMIO) module provided by PYNQ. We initialize the MMIO object with the physical address and size of the hailstone IP block obtained from Vivado. We then use the object to write the desired hailstone input to the input signal address, start the IP block by writing to the start signal, wait for the done signal, and read the hailstone output from the output signal address. Although this is the most basic way of interfacing the IP block, it is inconvenient and prone to er-

rors when used multiple times throughout an application. The better alternative will often be to define a Python driver specific to the IP block. In the case of hailstone, as seen in Figure 6, this is done by extending the DefaultIP driver class of the `pynq` Python module and binding the driver to the hailstone IP block. A function hailstone is then defined to handle the MMIO interfacing. This hailstone function can then be called as a regular Python function anywhere in the software application to provide hardware accelerated hailstone computation (see Figure 7) . This clearly shows how PYNQ makes it considerably easier for programmers to use the FPGA.

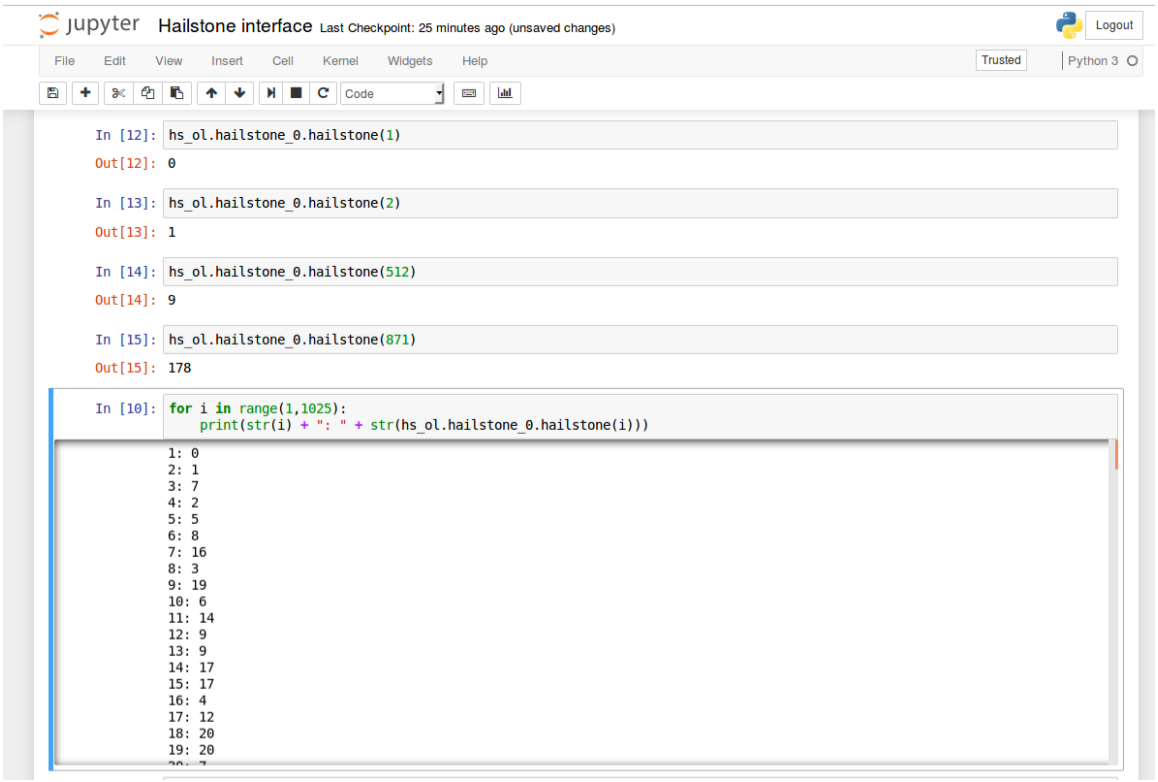

Figure 7: Interfacing the hailstone accelerator

3.4 Bitstream performance

In theory, the hardware implementation of the hailstone sequence length on PYNQ has the potential to far outperform its software implementation in Python. This is especially true considering that Isaiah Leonard found an approximate 50% speed-up of the same function between the C software implementation and the ZYNQ FPGA implementation [6]. However, we find that the PYNQ implementation of the hailstone count in this section is in fact far slower than the Python implementation.

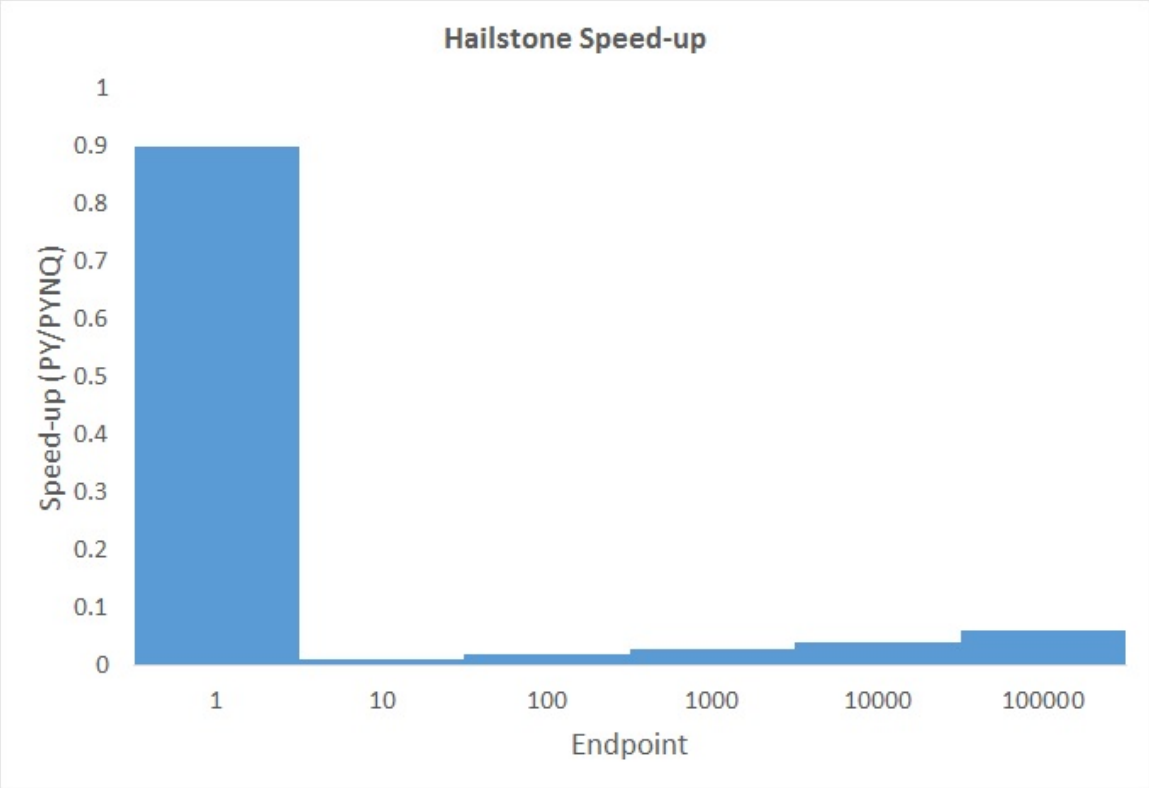

Figure 8: Hailstone slowdown on PYNQ

We measure the performance of the Python and PYNQ implementations of the hailstone count function described in this section using the `timeit` Python module. The timing script used is similar to the one transcribed in Appendix A, using the driver of Section 3.3 and a Python implementation of hailstone equivalent to the C implementation in Section 3.1. Similar to Leonard, we ran the function on both implementations to compute hailstone counts for ranges 1–1, 1–1000, 1–10000 and 1–100000 [6]. It is important to note that in this case the Python function is called, and the PYNQ accelerator is restarted, repeatedly for every number in the range. As seen in Figure 8, the Python implementation consistently performs at 10% of the speed of the PYNQ FPGA implementation.[3] Although counterintuitive at first, the slowdown of PYNQ in relation to Python is to be expected in this case. The reason for it is that the FPGA accelerator itself is idle for most of the time, waiting to be restarted for every loop iteration. Additionally, the "s_axilite" interface ports specified in HLS directives are AXI4-Lite ports that are tied to registers, which must be flushed and

[3]In range 1–1, no meaningful computation is actually performed. The slight PYNQ slowdown accounts for the bitstream download and Jupyter Notebook interfacing overheads of PYNQ.

rewritten for every new input and output. Transmitting the start signal between the PS and PL and rewriting registers turns into a considerable overhead when timing tens of thousands of operations [20]. These results demonstrate that to fully harness the power of FPGA acceleration through PYNQ, one must incorporate FPGA design in a way that does not require constant interfacing with the accelerator. We should aim for most of the computation time of our accelerated function to be spent on the accelerator itself. Chapter 4 elaborates on this design strategy and suggests ways to modify our hailstone count implementation to better benefit from FPGA acceleration.

4 Benefits of FPGAs

As seen in Chapter 3, PYNQ introduces a way for software programmers unfamiliar with hardware design to translate software functions down to an FPGA circuit. Moreover, PYNQ offers straightforward interfacing with FPGA-implemented functions through 4AXI I/O ports mapped to the memory map. These ports are accessed via the `pynq` Python library, which allows for the creation of drivers specific to a user-designed hardware block that can then be used as any other software function. As appealing as they are, these novel PYNQ features are all for naught if there is no demonstrated real benefit of using FPGAs as part of a larger system or application. Although in Chapter 3 we failed to demonstrate the function acceleration potential of FPGAs due to a short-running circuit implementation, previous work discussed in Chapter 2 suggests that there are considerable performance gains from employing FPGAs in large systems, as seen in Microsoft Catapult and the Binary Neural Network [9, 16]. In the context of dark silicon, Venkatesh et. al introduce the idea of using conservation cores ("c-cores")—functions implemented in dedicated hardware—to unload the processor, push back on the processor "utilization wall", thus increasing throughput [17].

In this chapter we demonstrate the benefits of FPGA use, both to accelerate functionality and as a flexible method for implementing conservation cores (or "c-cores"). To this end we fix the Chapter 3 implementation of hailstone count to make more efficient use of the FPGA and harness its full acceleration power. Extending on the works of Venkatesh et al. and Leonard, we show that FPGAs are a cheap way to implement multiple c-cores by creating and evaluating a multi-function FPGA circuit on PYNQ [6, 17].

4.1 FPGAs as accelerators

The hailstone count FPGA implementation described in Chapter 3 is inefficient for two reasons: (1) its I/O ports are part of the AXI4-Lite interface and memory-mapped to registers that must be rewritten for every new input; (2) the FPGA accelerator must be restarted every time it receives new input. We can see that the problem resides in the way inputs are consumed by the accelerator: they can only be consumed once for each run of the circuit, and the I/O channels are slow. This results in the accelerator being idle for most of the computation time, which is spent largely on transmitting control signals between processing system (PS) and the programmable logic (PL)—i.e. the FPGA circuit. In order to efficiently use the FPGA we must fix our hailstone count implementation so as to reduce idle accelerator time as much as possible, and this requires rethinking the way our accelerator uses or consumes inputs. We can then change the accelerator such that it only needs to consume inputs once at the beginning of computation and return the output once at the end, or change the way inputs and outputs are fed to and retrieved from the accelerator to reduce transmission overhead. The former case is more straightforward. It allows us to keep the AXI4-Lite interface, which provides direct CPU control of the board. This interface is slow, but since the input is consumed only once, it does not affect the accelerator's performance. In this case, the input given to the AXI4-Lite interface serves just to configure the FPGA, which will continually execute until all the desired computation is finished. This is, in fact, the design strategy recommended by the Vivado Design Suite User Guide – High-Level Synthesis [20]. The latter case involves the use of the AXI4-Stream interface and is more complicated. The use of streams is discussed in Subsection 4.1.2.

4.1.1 Iterative Hailstone

```
void hailstone(long start, long end, int* r){
  long s = start;
  long e = end;
  long n;
  long long cur;
  int count, max_count;

  max_count = 0;
  for(n = s; n < e; ++n){
    cur = n;
    count = 0;

    while(1) {
      if (cur % 2L) {
        if (cur == 1L){
          if(count > max_count) max_count = count;
          break;
        } else {
          cur = ((cur * 3L) +1L)/2L;
          count += 2;
        }
      } else {
        cur /= 2L;
        count++;
      }
    }
  }
  *r = max_count;
  return;
}
```

Figure 9: Iterative hailstone–C implementation

To fix our hailstone count FPGA implementation, instead of restarting our accelerator to compute the hailstone count for every number in the range of a Python-defined loop, we push this loop into our FPGA hardware. In this way, the inputs to the circuit determine the loop bounds. The C code for this iterative hailstone function is seen in Figure 9. The inputs to the function—loop-start, loop-end, and output reference—are all assigned AXI4-Lite (s-axilite) directives during High Level

Synthesis, just as described in Section 3.1. We synthesize the fixed hailstone count function into an FPGA circuit by following the simple PYNQ Workflow described in Chapter 3, with the exception that the Python driver for the fixed bitstream now accounts for the new input format (see Figure 10). Note that for testing purposes the accelerator does not output the hailstone count for every number in the range start–end. This can only be done efficiently by using more complicated I/O interfaces, such AXI4-Stream, and circuit designs involving DMA blocks, which we investigate in Subsection 4.1.2. Instead, the function returns the maximum count found within the range.

```
class HSDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:hailstone:1.0']

    def hailstone(self, s, e):
        self.write(0x10, s) # write input
        self.write(0x1c, e)

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x28) # return output
```

Figure 10: Iterative hailstone driver. Maximum hailstone count is computed over the input s–e range

```
def py_drive_hs(n):
    max_count = 0
    for x in range(1, n):
        count = 0
        cur = x

        while cur > 1:
            if (cur % 2) == 0:
                cur = cur//2
                count += 1
            else:
                cur = (cur*3+1)//2
                count +=2

        max_count = max(count, max_count)

    return max_count
```

Figure 11: Equivalent Python implementation

Using the Python script transcribed in Appendix A, we measure the performance of iterative hailstone implemented on PYNQ and compare it with the performance of equivalent Python implementation (Figure 11) via `timeit`. We time both implementations on ranges 1–1[1], 1–10, 1–100, 1–1000, 1–10000, 1–100000, 1–1000000 and 1–10000000. Table 1 shows these results, which are plotted in Figure 12. As expected, we find that PYNQ far outperforms Python. In the low ranges, 1–1 and 1–10, Python still has the edge as the bitstream loading and interfacing overheads of PYNQ dominate. However, PYNQ is faster for all other ranges measured, culminating with a 120-fold speed-up for range 1–10000000, for which PYNQ takes approximately 21s as compared to Python's 2,520s. The Python-to-PYNQ speed-up ratios shown in Figure 13 are far greater that the C-to-ZYNQ figures of Leonard [6]. Considering the many more layers of abstraction and translation involved in Python as compared to C, these results are expected.

[1] As noted before, no meaningful computation is performed here. Timing for this range reflects interfacing overhead.

Range	Python time(s)	PYNQ time(s)	Speedup
1–1	0.00002	0.00361	0.0055
1–10	0.00016	0.00317	0.0505
1–100	0.00474	0.00349	1.3582
1–1000	0.08731	0.00397	21.9924
1–10000	1.250	0.015	83.3333
1–100000	16.40	0.15	109.333
1–1000000	207.88	1.78	116.787
1–10000000	2519.99	20.95	120.286

Table 1: Performance of PYNQ hailstone and Python hailstone

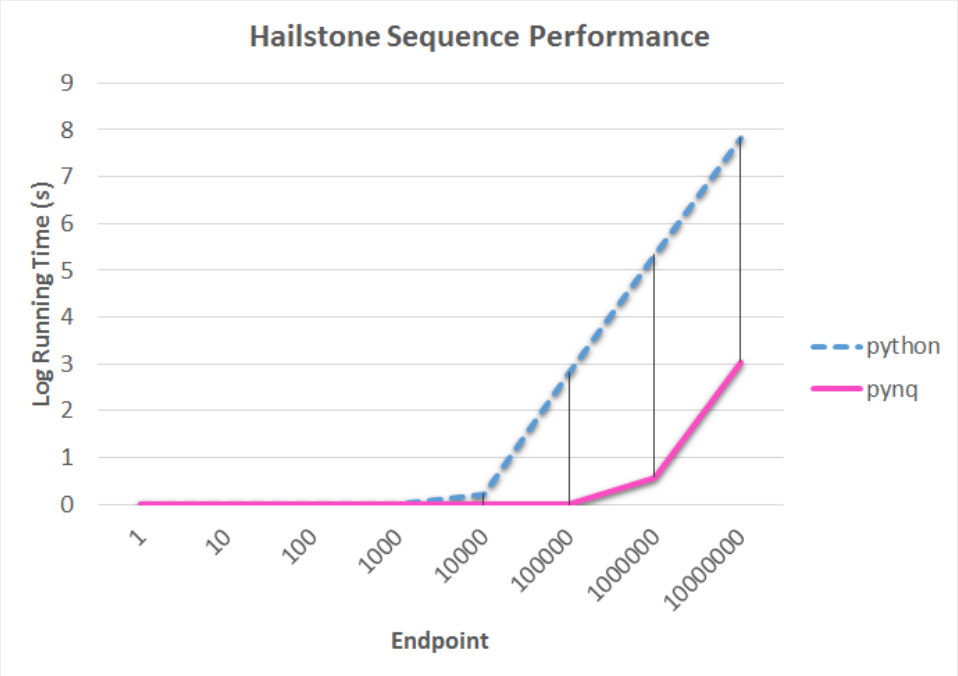

Figure 12: Performance of PYNQ hailstone and Python hailstone

Albeit simple, this experiment is effective in demonstrating the acceleration potential of FPGA circuits when designed appropriately. After seeing that this simple FPGA circuit—synthesized from software specification with little to no optimization—is capable of computing the hailstone count for approximately 0.5 million numbers per second, it should come with no surprise that employing FPGA circuits in large scale servers like Microsoft's Catapult improves throughput by 95%, or FPGA-implemented convolutional neural networks such as BNN are capable of classifying with high ac-

curacy 12.3 million images per second. When one realizes the untapped parallelism available by unrolling our hailstone range computation, the potential benefits seem well worth the added complexity of the hardware implementation.

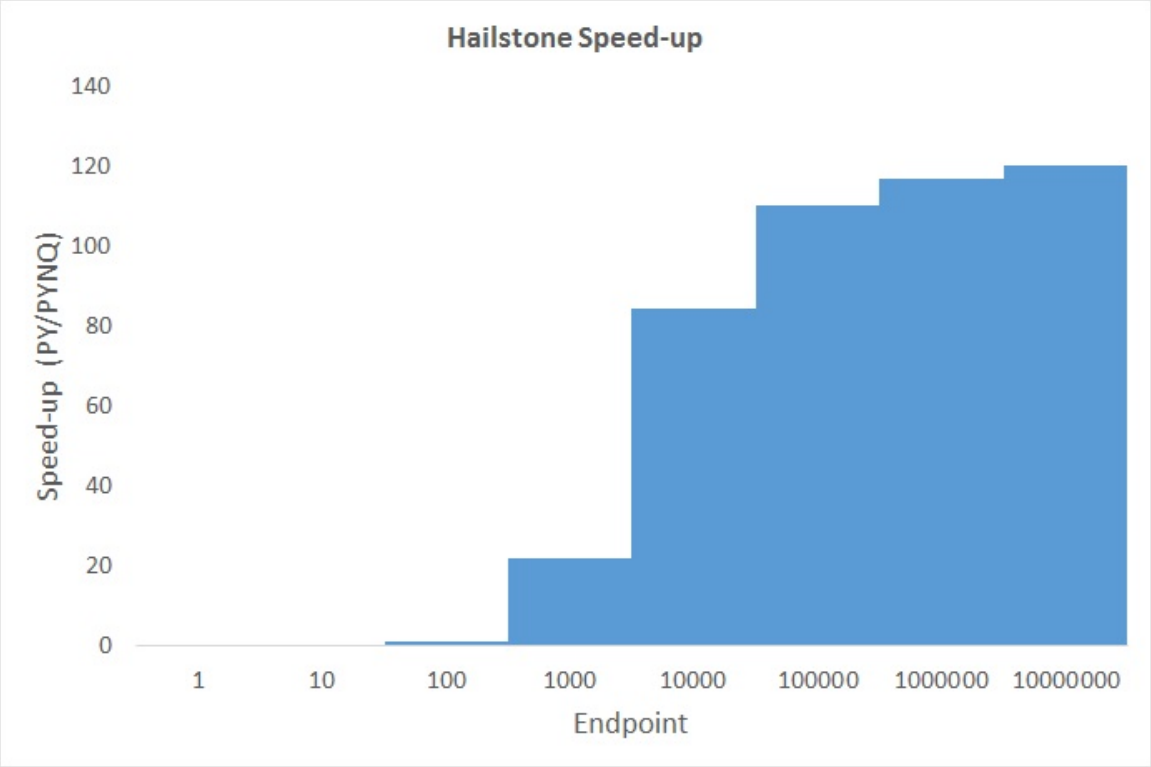

Figure 13: Hailstone speed-up on PYNQ

4.1.2 Stream Hailstone

In Subsection 4.1.1 we presented and evaluated the iterative hailstone design, where the FPGA accelerator is programmed to iterate through a continuous range of numbers—set via memory-mapped AX4-Lite input ports—to compute the maximum hailstone sequence length. Although we have shown iterative hailstone to make efficient use of the FPGA circuit, it was designed specifically to showcase the performance of PYNQ as a function accelerator as compared to Python. Iterative hailstone has the drawback that it can only be used to find the maximum hailstone count for a range of numbers, and for short ranges the software implementation is more desirable. Ideally, we would like to arbitrarily choose a potentially large set of numbers for which to

compute the hailstone count, feed it to the FPGA accelerator and get output for all of them. In software we can easily do this by passing an array of input numbers to hailstone count, which outputs a corresponding array of results. However, the AXI4-Lite input ports are static in size and do not support dynamically allocated input and output arrays [20]. Therefore, this ideal version of hailstone count cannot be efficiently implemented in the PYNQ PL by using the AXI4-Lite interface. Instead, we must employ an I/O interface that allows dynamically allocated input and output arrays—the AXI4-Stream.

The AXI4-Stream I/O interface allows the user to specify input and output streams. These can be of arbitrary size, and allow memory access by both the PS and the PL. In this way, the PS and PL communicate input and output via memory instead of signals. This eliminates the signal transmission overhead incurred by AXI4-Lite interfaced ports and allows the PL to keep reading and writing multiple inputs and outputs within a single accelerator run, eliminating the restart overhead. However, one major drawback of using the AXI4-Stream interface is that, unlike AXI4-Lite, it is not memory-mapped so it cannot be directly controlled by the PS. In PYNQ, to integrate a AXI4-Stream interfaced PL design with the PS and specify usable input and output buffers, we must use the the PYNQ DMA engine—`dma` module—and `Xlnk` module of the `pynq` Python library, as explained in the Overlay Design Methodology [18]. The `dma` module allows software control of the AXI Direct Memory Access hardware block. As shown in [18], this block must be incorporated to the circuit design in addition to the ZYNQ processing system block and the stream hailstone IP block, as described in Section 3.2. The AXI DMA block has both AXI4-Lite-interfaced slave ports and, AXI4-Stream-interfaced master and slave ports. When the blocks are connected automatically by Vivado (see Section 3.2), the AXI4-Lite ports of the AXI DMA block enable it to be directly controlled by the PYNQ PS (via the `dma` module), while its AXI4-Stream ports allow for direct access to the stream hailstone IP block's input and output streams. The AXI DMA block is then directly responsible for I/O between the PS and PL.

The most straightforward way to design the stream hailstone FPGA circuit is to start with software function specification in C++ using the `ap_axi_sdata.h` library that is included with Vivado HLS [20]. This library contains templates for AXI stream structs. We can then use these structs to define the input and output streams of our

stream hailstone function. Details on how to use the `ap_axi_sdata.h` stream structs can be found on pages 107–112 of the Vivado Design Suite User Guide–High-Level Synthesis, with additional examples available in the Overlay Design Methodology [18, 20]. We specify directives for the arguments to our function as described in Section 3.1, with the exception that we now choose the "axis" (AXI4-Stream) interface directive for our input and output streams instead of "s_axilite" (AXI4-Lite). The remainder of the workflow is the same as described in Chapter 3, except that we must also include the AXI DMA block—available in the default Vivado IP repository for PYNQ-Z1 board projects—in our circuit design.

Once the stream hailstone bitstream is loaded to the PL, we create input and output buffers on the software level with the `Xlnk` module, feed them to the DMA engine and control the direct input writes and output reads with the `dma` module, as described in the Overlay Design Methodology [18].

4.2 FPGAs as a medium for implementing conservation cores

In the current computation landscape, we have powerful processors available, with transistor counts theoretically capable of handling large and complex computations. However, these processors cannot be fully utilized at once since heat dissipated by transistors would cause them to fail. In fact it has been shown that the rate of utilization of processor potential drops exponentially 2× per processor generation, with as little as 7% of the processor fabric being utilized [14]. This is what Venkatesh et al. call the "utilization wall" of processors [17]. In [17], the authors introduce the concept of conservation cores (c-cores) as a solution to increase processor energy efficiency and allow for greater throughput. They present a toolchain for generating dedicated hardware circuits from functions specified in C. These dedicated circuits then trade area for energy efficiency. This model frees up the processor, which can delegate specific tasks to these circuits and use its power to process other tasks. The main purpose of c-cores, then, is not to serve as accelerators, but to improve energy efficiency of computation [17]. With this in mind, Venkatesh et al. conjecture that c-cores are good for implementing computationally intensive functions such as irregular integer applications, which may not be necessarily good candidates for acceleration

but will most likely be more energy efficient when implemented in dedicated silicon [17].

The similarities between c-cores and Xilinx FPGA design are clear. Chapter 3 shows that Xilinx Vivado Design Suite, like Venkatesh et al.'s c-core design toolchain, allow hardware (FPGA) circuits to be synthesized from software-specified functions. FPGAs, like c-cores, have been shown to improve energy efficiency of computation [9] without negatively affecting throughput. Moreover, within the PYNQ framework it is easy to specify what functions run on the processor or the programmable logic and when. The only difference between c-cores and FPGA circuits is that the former are dedicated and the latter can be reconfigured. While dedicated circuits can be better optimized to guarantee energy efficiency, FPGAs can be configured to perform any dedicated function. FPGAs are then an energetically inexpensive alternative to implementing any desired c-core at any given time. In this chapter we explore the versatility of FPGAs as c-cores by using the Vivado Design Suite and PYNQ to create an FPGA circuit that simultaneously implements five irregular integer application c-cores: iterative hailstone, factorial function, prime finder, Fibonacci finder and iterative multiplicative persistence. We call it the multi-sequence circuit.

We choose these five functions because they are interesting sequences that are computationally intensive and, similar to iterative hailstone, they use their inputs to configure inner iteration ranges. Therefore, their FPGA circuit completes the desired computation in a single long run, reading inputs and writing outputs once from and to AXI4-Lite-interfaced ports. These PL circuits, like iterative hailstone are then efficient, straightforward to implement and easy to control from the PS. The factorial function just computes $n!$ for input n; prime finder and Fibonacci finder respectively find the i^{th} prime and Fibonacci numbers given index i; iterative multiplicative persistence finds the maximum multiplicative persistence relative to an input base for numbers within a range whose bounds are also given as inputs.[2] The C-specification for each of these functions is found in Appendix B.

[2]Multiplicative persistence is the number of transformations by recursive digit multiplication it takes for a given number to become a single digit number given its base.

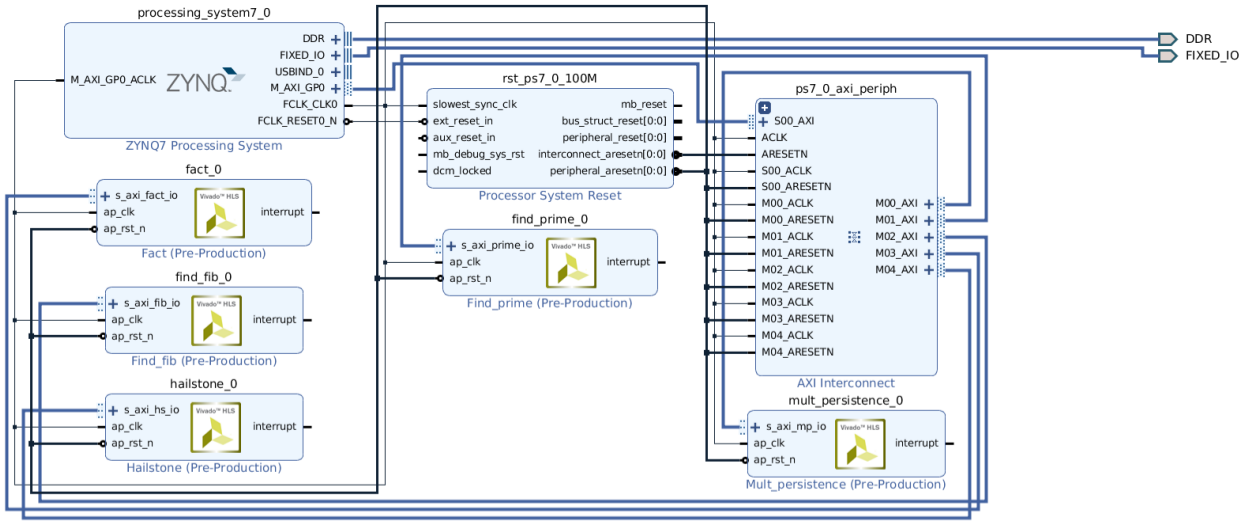

Figure 14: Block design with 5 user-defined IP blocks

To create the multi-sequence bitstream we must follow the HLS steps in Section 3.1 for each function's C implementation. Note that we set AXI4-Lite (s_axilite) interface directives for all input and return ports of all functions before synthesis. After successfully completing the HLS step for each function, we have five corresponding Vivado HLS IP blocks to be imported into Vivado as described in Section 3.2 to create our circuit design. In Vivado we then include all five IP blocks together with the ZYNQ processing system block in our block design. After running the block and connection automatons, the circuit should look like Figure 14. We then generate the bitstream for the multi-sequence circuit and export the bitstream and Tcl files by following the process described in Section 3.2.

We then download the bitstream onto the board and interface with it using the `pynq` package by creating five different drivers, each bound to one of the five user-defined functions. The Python specification for each driver is found in Appendix B. Thanks to PYNQ, switching between the five different FPGA-implemented functions is as simple as calling the associated driver function from our application!

We find that all functions work as expected when they are loaded onto the PL simultaneously within the multi-sequence bitstream. Each of these functions represents an irregular integer application, which Venkatesh et al. suggest as a good kind of candidate applications for c-cores [17]. Our result then demonstrates that a single FPGA chip, within the PYNQ framework, can be programmed to implement

many individual c-cores that can be selectively incorporated at runtime to offload from the processor just as many tasks. FPGAs and PYNQ therefore can be a great aid in cutting back the utilization wall and improving energy efficiency in systems.

However, in terms of acceleration performance, our multi-sequence FPGA circuit is generally slower than its Python counterpart. Using the script outlined in Appendix A.2, We timed our FPGA implementation of the multi-sequence circuit and compared it to the Python implementation for ranges 1–1, 1–10, 1–100, 1–1000, 1–10000 and 1–100000. This was done both by timing each function individually and by timing the entire circuit, serially selecting each function for each range.[3]

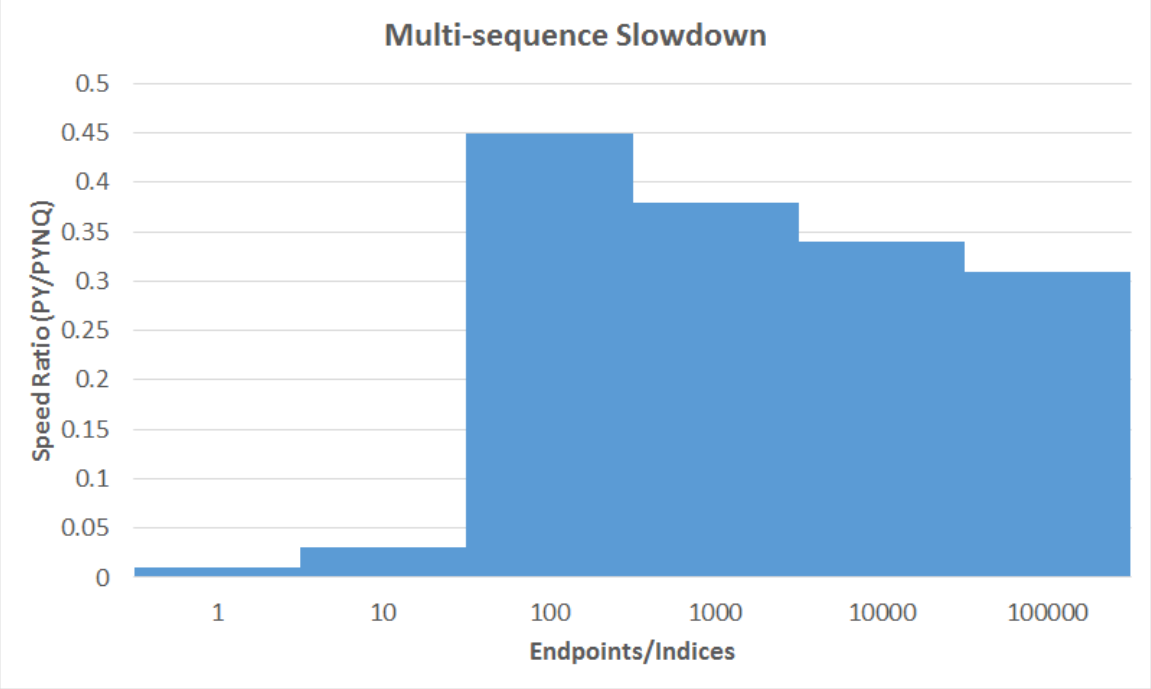

Figure 15: General slowdown of the multi-sequence circuit

In Figure 15 we see that for entire circuit timing, where we run each c-core serially for each range, the Python-implemented functions together perform at between 1–45% of the speed of their hardware implementation in FPGA. In this test, the PYNQ performance results represent an aggregation of the individual c-core performance results described below (except factorial) plus overhead from downloading the bitstream overlay and switching between c-cores, which are amortized for the bigger ranges.

[3]The factorial function is omitted from the full circuit timing and only timed individually for $n = 1, 10, 20$ due to its limited computational range.

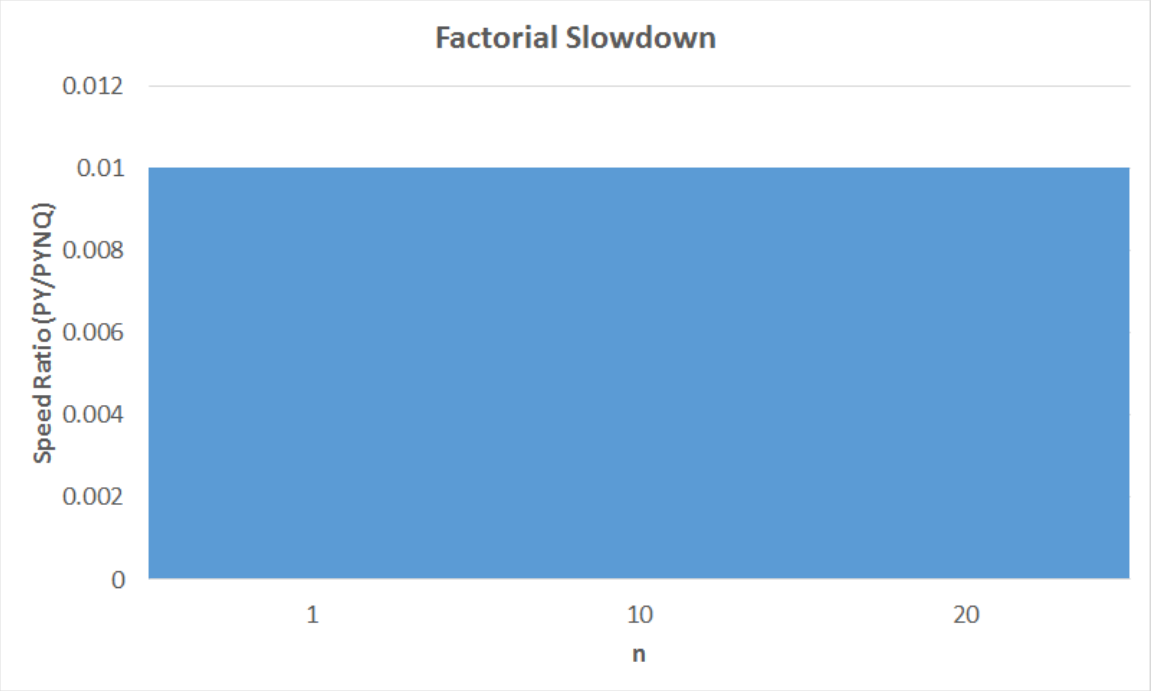

Figure 16: Factorial slowdown

As seen in Figure 16, our factorial FPGA c-core is considerably slower than the Python default `math.factorial` function. The latter consistently performs 100 times as fast than the former. This is expected given that this c-core naturally cannot make efficient use of the FPGA circuit. Factorial quickly overflows for large numbers, which prevents the FPGA from computing numbers bigger than 20!–30!. This results in short-lived computations in the FPGA accelerator, which as seen in Section 4.1 fail to amortize the interfacing overheads of the PYNQ circuit leading to an inefficient implementation. Furthermore, our factorial c-core is implemented prioritizing simplicity over performance, as opposed to the Python default.

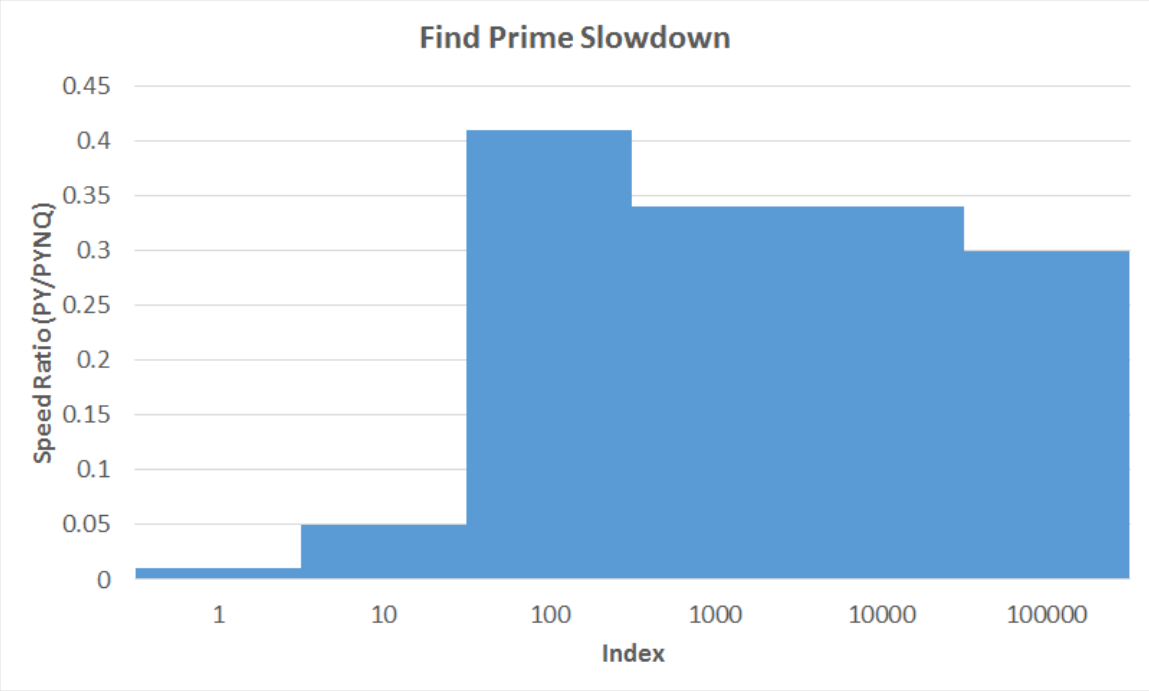

Figure 17: Prime finder slowdown

Given index i, prime finder returns the ith prime number. Figure 17 shows that the Python implementation of prime finder runs in 5–40% of the time of the hardware FPGA implementation. This is surprising because for large indices, the function is long-running and makes good use of the FPGA circuit. This can be explained, however, in the difference between the C function used to synthesize the FPGA circuit and the Python implementation of prime finder. In the latter, we are able to leverage the Fundamental Theorem of Arithmetic (FTA) that states that every number greater than 1 has a prime factorization. In this way, the function finds the prime at i by dividing each candidate by the prime numbers that precede it, which are stored in a list. This optimization, however, cannot be used in the FPGA implementation by following the PYNQ workflow described in Chapter 3. This is because the C array that would store the preceding prime numbers within the C-defined prime finder function cannot be easily synthesized into a hardware circuit for its size is dynamically allocated. See Appendix B for details on these function definitions.

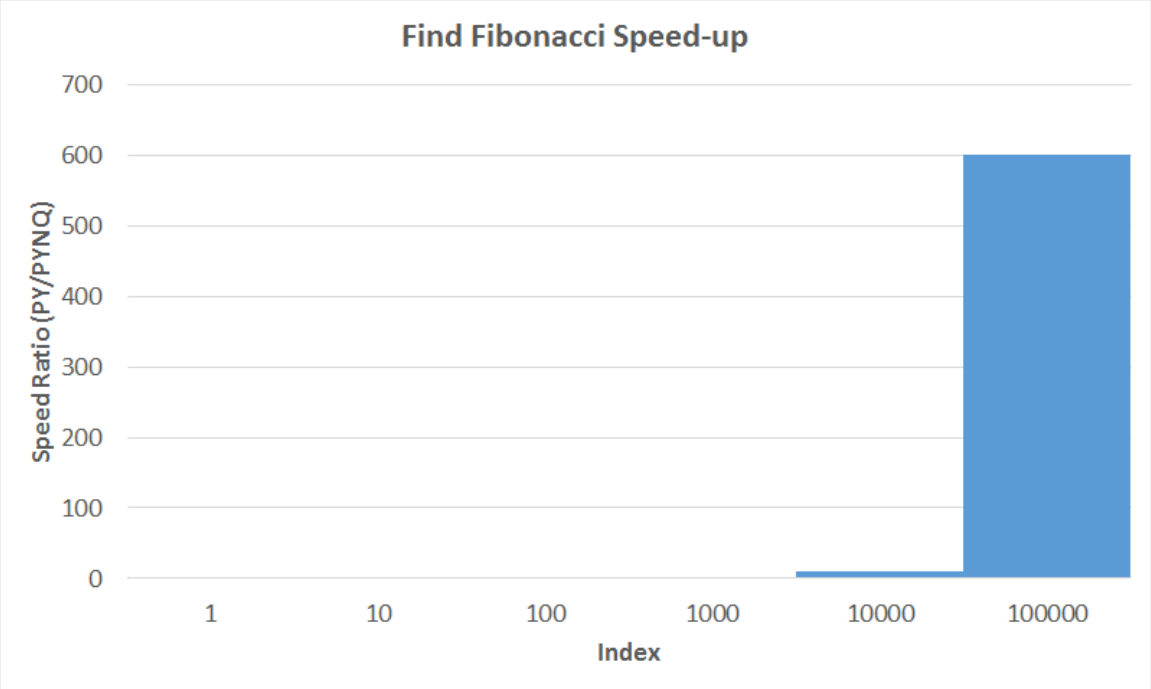

Figure 18: Fibonacci finder speed-up

Similarly to prime finder, the Fibonacci finder c-core returns the ith Fibonacci number given index i. In this case, however, the C function from which the c-core is synthesized and the Python implementation are similar. We can see in Figure 18 that for large indices, where computation becomes more intensive, the FPGA c-core performs 600× as fast as the Python implementation.

Finally, the maximum multiplicative persistence function return the largest multiplicative persistence found within the given range. In this way, it works in a way very similar to iterative hailstone, and is designed to make efficient use of the FPGA circuit. Figure 19 shows that the iterative multiplicative persistence c-core performs 1.5× to 2.6× as fast as the Python implementation of the function for larger ranges.

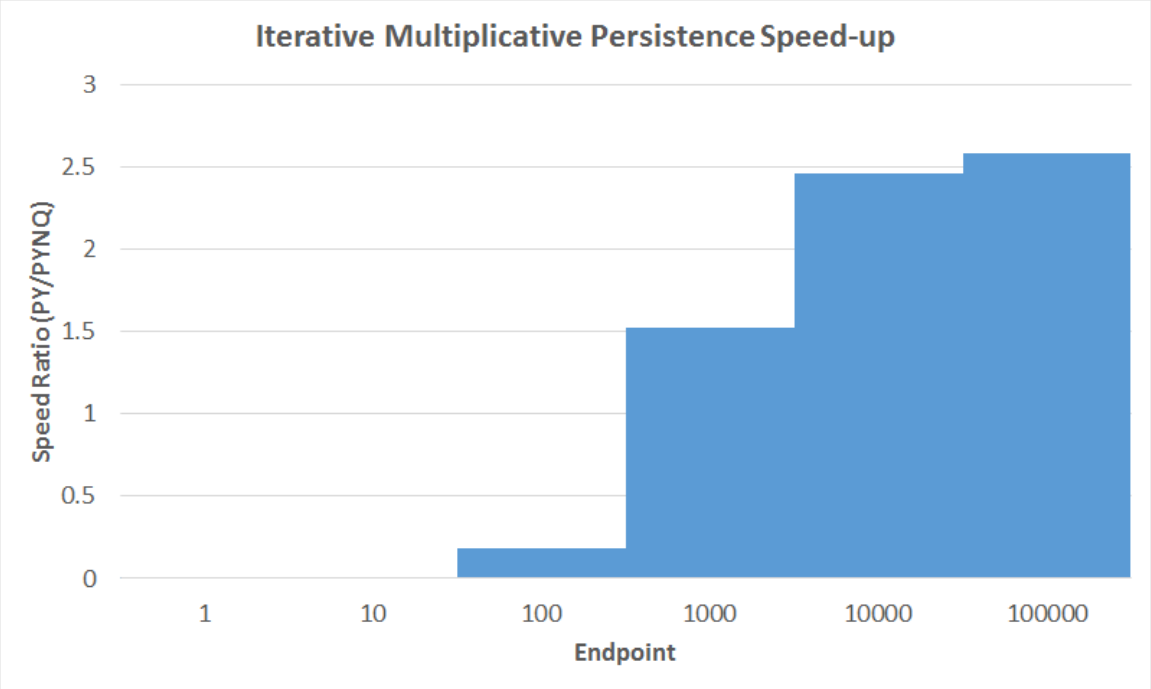

Figure 19: Iterative multiplicative persistence speed-up

These results are in line with Venkatesh et al.'s conjecture that these kinds of functions are not necessarily good candidates for hardware acceleration, yet may offer gains in energy efficiency—which is the main goal of c-cores [17]. In terms of energy, we should expect FPGA-implemented c-cores to be less efficient than the ASIC-based c-cores described in [17]. Nevertheless, FPGA circuits have been shown to still be more energy efficient than their software counterpart, as they do not pay for instruction translation. In the context of large, complex systems the benefits provided by FPGA c-core implementation—the cost-effectiveness, the ability of define and switch between multiple c-cores at once and easily change and reconfigure them—can outweigh the loss in energy efficiency as compared to dedicated c-cores.

5 Evaluation

This work sets out to improve FPGA accessibility by offering guidance in incorporating FPGA design in applications and projects using the PYNQ framework. To this end, we are successful in collating previous work and documentation into a comprehensive tutorial on the simple PYNQ workflow of Chapter 3. This allows developers unfamiliar with FPGA design to specify a target software function and synthesize it into a working FPGA circuit that can be easily interfaced with PYNQ. This work is also successful in offering some recommendations for appropriate, efficient FPGA design and showing the performance implications of both inefficient and efficient designs. In Chapter 4, we demonstrate the considerable gain in function performance when implemented in hardware via FPGA instead of software in Python. This effectively offers motivation for the still complex incorporation of FPGAs into larger application designs. We then extend on the PYNQ workflow introduced in Chapter 3 by presenting a more complex C++-to-FPGA circuit design involving I/O streams, which give us a more efficient way of communicating with the FPGA circuit as it executes, and could translate into better performance. Finally, we successfully show the potential for FPGAs as a flexible way of implementing c-cores, as defined by [17], by describing the implementation and usage of five c-cores in a single FPGA circuit.

This research can be made more complete by presenting energy efficiency analysis of FPGA in addition to the acceleration analysis. This is particularly relevant in firmly establishing FPGAs as effective, flexible and scalable conservation cores in potential. We choose instead to rely on previous work supporting the energy efficiency of FPGA circuits relative to equivalent software-implemented functionality and accept that FPGA-implemented conservation cores can well be less energy efficient than ASIC-based c-cores, yet still appealing due to their programmability, scalability and cost-effectiveness. Moreover, we do not explore the runtime reconfiguration of FPGA

circuits. Doing so can offer the reader both greater motivation, and more understanding of how and when to incorporate FPGA design into their projects, and therefore can be instrumental in furthering the success of our work.

In working with PYNQ, we are able to realize how much of an important step towards wider FPGA adoption this platform is. It brings together the power of hardware accelerated functions and the flexibility of Python programming onto a single board. It allows programmers to communicate with their FPGA chip remotely via a web browser running a Jupyter Notebook. This has major implications for productivity and ease of incorporation of FPGAs in software designs. We learn that once the FPGA circuit is properly configured and running, working with PYNQ is really straightforward. However, getting the FPGA to work as desired is still a big challenge, especially for those unfamiliar with hardware design. Creating a working FPGA bitstream from a software-implemented function using the Vivado toolchain is still a very complicated task involving a wide variety of technical skills. Getting all details right often requires specific hardware design knowledge, which arguably defeats the purpose of a hardware design toolchain that starts with software-specified functions. Although PYNQ does currently recommend that software designers use and compose the bitstreams already created and refrain from specifying entirely new circuits, this ultimately takes away from the appeal, flexibility and potential of FPGA use [19]. Documentation on the different parts of the process is decentralized and sometimes incomplete, which makes it more challenging to piece together a working PYNQ circuit using the Vivado toolchain. Ultimately, we believe that streamlining the Vivado toolchain and offering clearer, more centralized documentation can be a valuable aid in improving the accessibility of the platform. We do recognize that the PYNQ project is still in its early stages, and commend the Xilinx team for constantly updating the `pynq` package and documentation, supporting the PYNQ community by publishing relevant PYNQ Notebooks and offering constant assistance in PYNQ forums. It is clear that Xilinx is committed to the PYNQ project, which shows great potential.

6 Conclusion and Future Work

As support for high performance computing becomes increasingly necessary and the computational demands of large-scale, data driven systems begin to outpace the capacity of current-generation processors bound by the utilization wall, we begin to rethink our current model of computation. Hardware-supported computation now represents one promising solution in supporting high throughput and increasing processor energy efficiency. In this context, FPGAs are appealing as they can be programmed to act as any digital circuit. They can be used for simple function acceleration, to implement large algorithms or virtually emulate other dedicated hardware chips. Additionally, FPGAs offer the possibility of runtime reconfiguration which has major implications for system maintenance and persistence [5, 9, 17].

Despite their potential, FPGA use is not widespread. Among many reasons for low FPGA adoption, the difficulty of FPGA design stands out. This research set out to improve FPGA accessibility and encourage adoption by building on previous efforts in this area. We explained in detail how a software engineer can create a simple working FPGA circuit from scratch using the Vivado Design Suite and easily manage and interface with the circuit and incorporate it into an application using the PYNQ platform. We showed how to extend upon this simple workflow to create a slightly more complex FPGA circuit using I/O streams to offer the reader more in-depth understanding of FPGA design. We also demonstrated the benefits of FPGA use both as accelerators and conservation cores. In the former, we instructed the user on how to create a function that makes efficient use of the FPGA circuit and demonstrated its acceleration power by timing and comparing both the FPGA and equivalent Python implementations of the function. In the latter, we showed how to create a bitstream that supports multiple functions that work correctly and can be selectively used at runtime, in this way offloading tasks from the processor to

effectively work as conservation cores and improve energy efficiency, cutting back on the utilization wall. This work ultimately offered guidance and motivation for FPGA design, which we hope may make FPGAs more accessible to the reader.

Natural extensions of our work include the design of FPGA-implemented c-cores and their comparison with equivalent, ASIC-based c-cores in terms of energy efficiency, ease of design, flexibility and scalability. Exploring the runtime reconfiguration of FPGAs and offering a comprehensive tutorial on how to implement runtime-reconfigurable circuits on PYNQ can offer both more understanding of the PYNQ workflow and greater motivation for FPGA adoption, which fully aligns with the goals of this research. By combining the possibility of runtime bitstream reconfiguration with the ability to define a multi-function bitstream, we can begin to think of a system and would allow for timesharing and context-switching of a single FPGA chip between different users or capabilities. We can then begin to think about abstracting away explicit bitstream management and perhaps streamlining the entire process of FPGA design for use in general computation.

Bibliography

[1] Auerbach, J., Bacon, D. F., Cheng, P., and Rabbah, R. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 89–108.

[2] Bacon, D. F., Rabbah, R., and Shukla, S. FPGA programming for the masses. *Communications of the ACM 56*, 4 (2013), 56–63.

[3] Glette, K., Torresen, J., and Hovin, M. Intermediate level FPGA reconfiguration for an online EHW pattern recognition system. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on* (2009), IEEE, pp. 19–26.

[4] Halfhill, T. R. Tabula's time machine. *Microprocessor report 131* (2010).

[5] Koch, D., Beckhoff, C., and Teich, J. A communication architecture for complex runtime reconfigurable systems and its implementation on spartan-3 FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays* (2009), ACM, pp. 253–256.

[6] Leonard, I. Using reconfigurable hardware to fight dark silicon. Williams College.

[7] Liu, H., and Wong, D. Circuit partitioning for dynamically reconfigurable FPGAs. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays* (1999), ACM, pp. 187–194.

[8] Pérez, F., and Granger, B. E. IPython: a system for interactive scientific computing. *Computing in Science & Engineering 9*, 3 (2007).

[9] Putnam, A., Caulfield, A., Chung, E., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Peterson, E., Smith, A., Thong, J., Xiao, P. Y., Burger, D., Larus, J., Gopal, G. P., and Pope, S. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)* (June 2014), IEEE Press, pp. 13–24.

[10] Rashid, R., Baron, R., Forin, A., Golub, D., Jones, M., Orr, D., and Sanzi, R. MACH: a foundation for open systems (operating systems). In *Workstation Operating Systems, 1989., Proceedings of the Second Workshop on* (1989), IEEE, pp. 109–113.

[11] Ritchie, O., and Thompson, K. The UNIX time-sharing system. *The Bell System Technical Journal 57*, 6 (1978), 1905–1929.

[12] Schmidt, A. G., Weisz, G., and French, M. Evaluating rapid application development with python for heterogeneous processor-based fpgas. *arXiv preprint arXiv:1705.05209* (2017).

[13] Tavares, Y. Creating a simple overlay for PYNQ-Z1 board from vivado hlx. `https://yangtavares.com/2017/07/31/creating-a-simple-overlay-for-pynq-z1-board-from-vivado-hlx/`, 2017. Accessed: 2018-03-27.

[14] Taylor, M. B. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE* (2012), IEEE, pp. 1131–1136.

[15] Tessier, R., Pocek, K., and DeHon, A. Reconfigurable computing architectures. *Proceedings of the IEEE 103*, 3 (2015), 332–354.

[16] Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M., and Vissers, K. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2017), FPGA '17, ACM, pp. 65–74.

[17] Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., and Taylor, M. B. Conservation cores: reducing the energy of mature computations. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 205–218.

[18] Xilinx. Overlay design methodology. `http://pynq.readthedocs.io/en/latest/overlay_design_methodology.html`, 2017. Accessed: 2018-04-16.

[19] Xilinx. PYNQ overlays. `http://pynq.readthedocs.io/en/latest/pynq_overlays.html`, 2017. Accessed: 2018-04-16.

[20] Xilinx. Vivado design suite user guide. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirectives`, 2018. Accessed: 2018-04-16.

A Timing scripts

This appendix outlines the timing scripts used to test and compare the hailstone overlay and the multi-sequence overlay with their Python-defined counterparts. Note that these scripts are run in our PYNQ board's Jupyter Python3 Notebook environment during testing.

A.1 Hailstone timing script

```
import timeit
import time

pynqsetup = '''
from pynq import Overlay
from pynq import DefaultIP


class HSDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:hailstone:1.0']

    def hailstone(self, s, e):
        self.write(0x10, s) # write input
        self.write(0x1c, e)

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
```

```
        return self.read(0x28) # return output



hs_ol = Overlay('/home/xilinx/jupyter_notebooks/hsmax_overlay/' +
'hailstone_max.bit', download=False) # download bitstream

if not hs_ol.is_loaded():
    hs_ol.download()

def pynq_drive_hs(n):
    return hs_ol.hailstone_0.hailstone(1, n)
'''


pysetup = '''
def py_drive_hs(n):
    max_count = 0
    for x in range(1, n):
        count = 0
        cur = x

        while cur > 1:
            if (cur % 2) == 0:
                cur = cur//2
                count += 1
            else:
                cur = (cur*3+1)//2
                count +=2

        max_count = max(count, max_count)

    return max_count
'''

def pystmt(n):
    return "py_drive_hs("+str(n)+")"

def pynqstmt(n):
    return "pynq_drive_hs("+str(n)+")"


def do_timeit(sup, st):
    time.sleep(5)
    return timeit.timeit(setup=sup, stmt=st, number=1)
```

```
if __name__=="__main__":
    base = 1000
    f1 = open('iter_platform.csv', 'w')
    f2 = open('speed_iter.csv', 'w')

    print("Timing PYNQ...")

    pynq1 = do_timeit(pynqsetup, pynqstmt(base//base))
    pynq10 = do_timeit(pynqsetup, pynqstmt(base//100))
    pynq100 = do_timeit(pynqsetup, pynqstmt(base//10))
    pynq1000 = do_timeit(pynqsetup, pynqstmt(base))
    pynq10000 = do_timeit(pynqsetup, pynqstmt(10*base))
    pynq100000 = do_timeit(pynqsetup, pynqstmt(100*base))
    pynq1000000 = do_timeit(pynqsetup, pynqstmt(1000*base))
    pynq10000000 = do_timeit(pynqsetup, pynqstmt(10000*base))

    time.sleep(10)

    print("Timing Python...")

    py1 = do_timeit(pysetup, pystmt(base//base))
    py10 = do_timeit(pysetup, pystmt(base//100))
    py100 = do_timeit(pysetup, pystmt(base//10))
    py1000 = do_timeit(pysetup, pystmt(base))
    py10000 = do_timeit(pysetup, pystmt(10*base))
    py100000 = do_timeit(pysetup, pystmt(100*base))
    py1000000 = do_timeit(pysetup, pystmt(1000*base))
    py10000000 = do_timeit(pysetup, pystmt(10000*base))

    time.sleep(10)

    text1 = "iter,python,pynq\n"

    text1 += ("%d,%.5f,%.5f\n" % (base//base, py1, pynq1))
    text1 += ("%d,%.5f,%.5f\n" % (base//100,py10,pynq10))
    text1 += ("%d,%.5f,%.5f\n" % (base//10,py100,pynq100))
    text1 += ("%d,%.5f,%.5f\n" % (base,py1000,pynq1000))
    text1 += ("%d,%.5f,%.5f\n" % (base*10,py10000,pynq10000))
    text1 += ("%d,%.5f,%.5f\n" % (base*100,py100000,pynq100000))
    text1 += ("%d,%.5f,%.5f\n" % (base*1000,py1000000,pynq1000000))
    text1 += ("%d,%.5f,%.5f\n" % (base*10000,py10000000,pynq10000000))
```

```
    text2 = "speedup,iter\n"

    text2 += ("%.2f,%d\n" % (float(py1/pynq1),base//base))
    text2 += ("%.2f,%d\n" % (float(py10/pynq10),base//100))
    text2 += ("%.2f,%d\n" % (float(py100/pynq100),base//10))
    text2 += ("%.2f,%d\n" % (float(py1000/pynq1000),base))
    text2 += ("%.2f,%d\n" % (float(py10000/pynq10000),base*10))
    text2 += ("%.2f,%d\n" % (float(py100000/pynq100000),base*100))
    text2 += ("%.2f,%d\n" % (float(py1000000/pynq1000000),base*1000))
    text2 += ("%.2f,%d\n" % (float(py10000000/pynq10000000),base*10000))

    f1.write(text1)
    f2.write(text2)

    f1.close()
    f2.close()
```

A.2 Multi-sequence timing script

```
import timeit
import time

pynqsetup = '''
from pynq import Overlay
from pynq import DefaultIP

class MPDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:mult_persistence:1.0']

    def mult_persistence(self, end, base):
        self.write(0x10, end) # write input
        self.write(0x1c, base)
```

```
        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x24) # return output


class HSDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:hailstone:1.0']

    def hailstone(self, s, e):
        self.write(0x10, s) # write input
        self.write(0x1c, e)

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x28) # return output


class PrimeDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:find_prime:1.0']

    def find_prime(self, ind):
        self.write(0x10, ind) # write input

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x18) # return output


class FibDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:find_fib:1.0']

    def find_fib(self, ind):
```

```
        self.write(0x10, ind) # write input

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x18) # return output


class FactDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:fact:1.0']

    def fact(self, n):
        self.write(0x10, n) # write input

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x18) # return output


seq_ol = Overlay('/home/xilinx/jupyter_notebooks/multiseq_overlay/' +
'multisequence.bit', download=False) # Create Overlay

if not seq_ol.is_loaded():
    seq_ol.download()


def pynq_drive_fact(n):
    return seq_ol.fact_0.fact(n)


def pynq_drive_hs(n):
    return seq_ol.hailstone_0.hailstone(1, n)


def pynq_drive_prime(n):
    return seq_ol.find_prime_0.find_prime(n)


def pynq_drive_fib(n):
    return seq_ol.find_fib_0.find_fib(n)
```

```
def pynq_drive_mp(n):
    return seq_ol.mult_persistence_0.mult_persistence(n, 16)

'''

pysetup = '''
import math

def py_drive_fact(n):
    return math.factorial(n)

def py_drive_hs(n):
    max_count = 0
    for x in range(1, n):
        count = 0
        cur = x

        while cur > 1:
            if (cur % 2) == 0:
                cur = cur//2
                count += 1
            else:
                cur = (cur*3+1)//2
                count +=2

        max_count = max(count, max_count)

    return max_count


def py_drive_prime(n):
    if n < 1:
        return None

    prime_list = [2 for i in range(n)]

    cur = 3
    for i in range(1, len(prime_list)):
        j = 0
        while j < i:
            if cur%prime_list[j] == 0:
                cur += 1
                j = 0
            else:
```

```
                j += 1
        prime_list[i] = cur
        cur += 2

    return prime_list[-1]


def py_drive_fib(n):
    if n < 1:
        return None

    x_1 = 1
    x_2 = 1
    x = 1

    for i in range(2,n):
        x_2 = x
        x += x_1
        x_1 = x_2

    return x


def mult_persistence(end, base):
    max_count = 0
    for n in range(end):
        cur_n = n
        count = 0

        while cur_n >= base:
            count += 1
            tmp_n = cur_n
            prod = 1

            while tmp_n > 0:
                prod *= tmp_n%base
                tmp_n //= base
            cur_n = prod

        if count > max_count:
            max_count = count
    return max_count


def py_drive_mp(n):
```

```
    return mult_persistence(n, 16)

'''

FACT = 0
HS = 1
PRIME = 2
FIB = 3
MP = 4

def pynqstmt(n, func=None):
    s = "pynq_drive_"
    if func == FACT:
        s += "fact("+str(n)+")"
    elif func == HS:
        s += "hs("+str(n)+")"
    elif func == PRIME:
        s += "prime("+str(n)+")"
    elif func == FIB:
        s += "fib("+str(n)+")"
    elif func == MP:
        s += "mp("+str(n)+")"
    else:
        s = pynqstmt_all(n)
    return s


def pynqstmt_all(n):
    s = ""
    #s += pynqstmt(n, FACT) + "\n"
    s += pynqstmt(n, HS) + "\n"
    s += pynqstmt(n, PRIME) + "\n"
    s += pynqstmt(n, FIB) + "\n"
    s += pynqstmt(n, MP)
    return s


def pystmt(n, func=None):
    s = "py_drive_"
    if func == FACT:
        s += "fact("+str(n)+")"
    elif func == HS:
        s += "hs("+str(n)+")"
    elif func == PRIME:
        s += "prime("+str(n)+")"
```

```
    elif func == FIB:
        s += "fib("+str(n)+")"
    elif func == MP:
        s += "mp("+str(n)+")"
    else:
        s = pystmt_all(n)
    return s


def pystmt_all(n):
    s = ""
    #s += pystmt(n, FACT)+ "\n"
    s += pystmt(n, HS)+ "\n"
    s += pystmt(n, PRIME)+ "\n"
    s += pystmt(n, FIB)+ "\n"
    s += pystmt(n, MP)
    return s


def do_timeit(sup, st):
    time.sleep(5)
    return timeit.timeit(setup=sup, stmt=st, number=1)


def do_mult_timeit(t_list, setup, stmt, func=None):
    base = 1
    for i in range(len(t_list)):
        m = pow(10, i)
        t_list[i] = do_timeit(setup, stmt(base*m, func))


def print_speed_ratio(fp, pynq_list, py_list):
    text = "speedup,iter\n"
    print("Speed ratio (PY/PYNQ):")

    for i in range(len(pynq_list)):
        itr = pow(10,i)
        ratio = float(py_list[i]/pynq_list[i])
        text += ("%.2f,%d\n" % (ratio,itr))
        print("%d: %.2f\n"%(itr,ratio))

    fp.write(text)


def print_results(fp, pynq_list, py_list):
```

```
    text = "iter,python,pynq\n"

    for i in range(len(pynq_list)):
        itr = pow(10,i)
        text += ("%d,%.5f,%.5f\n" % (itr, py_list[i], pynq_list[i]))
        print("py%d: %f\n"%(itr,py_list[i]))
        print("pynq%d: %f\n"%(itr,pynq_list[i]))

    fp.write(text)


base = 1000
f1 = open('iter_platform.csv', 'w')
f2 = open('speed_iter.csv', 'w')

f_fact = open('fact.csv', 'w')
f_prime = open('prime.csv', 'w')
f_fib = open('fib.csv', 'w')
f_mp = open('mp.csv', 'w')

print("Timing PYNQ All...")

pynq_all_list = [0.0 for x in range(6)]
do_mult_timeit(pynq_all_list, pynqsetup, pynqstmt)

time.sleep(10)

print("Timing Python All...")

py_all_list = [0.0 for x in range(6)]
do_mult_timeit(py_all_list, pysetup, pystmt)

time.sleep(10)

print()
print("Results All:")
print()

print_results(f1, pynq_all_list, py_all_list)

print()
print_speed_ratio(f2, pynq_all_list, py_all_list)

f1.close()
f2.close()
```

```
print()
####### FACT #######

print("Timing PYNQ FACT...")

pynq1_fact = do_timeit(pynqsetup, pynqstmt(base//base, FACT))
pynq10_fact = do_timeit(pynqsetup, pynqstmt(base//100, FACT))
pynq20_fact = do_timeit(pynqsetup, pynqstmt(base//50, FACT))

time.sleep(10)

print("Timing Python FACT...")

py1_fact = do_timeit(pysetup, pystmt(base//base, FACT))
py10_fact = do_timeit(pysetup, pystmt(base//100, FACT))
py20_fact = do_timeit(pysetup, pystmt(base//50, FACT))

time.sleep(10)

print()
print("Results FACT:")
print()

print("Speed ratio (PY/PYNQ):")
print("1: " + str(float(py1_fact)/float(pynq1_fact)))
print("10: " + str(float(py10_fact)/float(pynq10_fact)))
print("20: " + str(float(py20_fact)/float(pynq20_fact)))


text2 = "speedup,iter\n"

text2 += ("%.2f,%d\n" % (float(py1_fact/pynq1_fact),base//base))
text2 += ("%.2f,%d\n" % (float(py10_fact/pynq10_fact),base//100))
text2 += ("%.2f,%d\n" % (float(py20_fact/pynq20_fact),base//50))

f_fact.write(text2)
f_fact.close()

print()

####### PRIME #######
print("Timing PYNQ PRIME...")

pynq_prime_list = [0.0 for x in range(6)]
```

```
do_mult_timeit(pynq_prime_list, pynqsetup, pynqstmt, PRIME)

time.sleep(10)


print("Timing Python PRIME...")

py_prime_list = [0.0 for x in range(6)]
do_mult_timeit(py_prime_list, pysetup, pystmt, PRIME)


time.sleep(10)

print()
print("Results PRIME:")
print()

print_speed_ratio(f_prime, pynq_prime_list, py_prime_list)
f_prime.close()

print()

####### FIB #######
print("Timing PYNQ FIB...")

pynq_fib_list = [0.0 for x in range(6)]
do_mult_timeit(pynq_fib_list, pynqsetup, pynqstmt, FIB)

time.sleep(10)


print("Timing Python FIB...")

py_fib_list = [0.0 for x in range(6)]
do_mult_timeit(py_fib_list, pysetup, pystmt, FIB)


time.sleep(10)

print()
print("Results FIB:")
print()

print_speed_ratio(f_fib, pynq_fib_list, py_fib_list)
f_fib.close()
```

```
print()

####### MP #######
print("Timing PYNQ MP...")

pynq_mp_list = [0.0 for x in range(6)]
do_mult_timeit(pynq_mp_list, pynqsetup, pynqstmt, MP)

time.sleep(10)


print("Timing Python MP...")

py_mp_list = [0.0 for x in range(6)]
do_mult_timeit(py_mp_list, pysetup, pystmt, MP)


time.sleep(10)

print()
print("Results MP:")
print()

print_speed_ratio(f_mp, pynq_mp_list, py_mp_list)
f_mp.close()
```

B Multi-sequence C functions and drivers

```
void fact(int* n){
  int i, m, r;
  m = *n;
  r = *n;
  fact_label0:for( i = 1; i < m; ++i) r *= i;
  *n = r;
}
```

Figure 20: Factorial C function

```
class FactDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:fact:1.0']

    def fact(self, n):
        self.write(0x10, n) # write input

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x18) # return output
```

Figure 21: Factorial driver

```
import math
math.factorial(n)
```

Figure 22: Factorial Python function

```
void find_prime(int ind, long* r){
  int index = ind;

  if(index < 1) {
    *r = -1;
    return;
  }

  long x;
  long i;

  x = 1;

  while(index){
    x++;
    i = 2;
    while(i < x){
      if(!(x%i)){
        x++;
        i = 2;
      } else {
        i++;
      }
    }
    --index;
  }

  *r = x;
  return;
}
```

Figure 23: Find Prime C function

```
class PrimeDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:find_prime:1.0']

    def find_prime(self, ind):
        self.write(0x10, ind) # write input

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x18) # return output
```

Figure 24: Find prime driver

```
def find_prime(n):
    if n < 1:
        return None

    prime_list = [2 for i in range(n)]

    cur = 3
    for i in range(1, len(prime_list)):
        j = 0
        while j < i:
            if cur%prime_list[j] == 0:
                cur += 1
                j = 0
            else:
                j += 1
        prime_list[i] = cur
        cur += 2

    return prime_list[-1]
```

Figure 25: Find Prime Python function

```
void find_fib(int ind, long* r){
  int index = ind;
  if(index < 1) {
    *r = -1;
    return;
  }

  long x;
  long x_1 = 1;
  long x_2 = 1;
  int i;

  for(i = 0; i < index; ++i){
    if(i < 2 ){
      x = 1;
    } else {
      x_2 = x;
      x += x_1;
      x_1 = x_2;
    }
  }
  *r = x;
  return;
}
```

Figure 26: Find Fibonacci C function

```
class FibDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:find_fib:1.0']

    def find_fib(self, ind):
        self.write(0x10, ind) # write input

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x18) # return output
```

Figure 27: Find Fibonacci driver

```
def find_fib(n):
    if n < 1:
        return None

    x_1 = 1
    x_2 = 1
    x = 1

    for i in range(2,n):
        x_2 = x
        x += x_1
        x_1 = x_2

    return x
```

Figure 28: Find Fibonacci Python function

```
void mult_persistence(long end, int base, long* r){
  long e, n, cur_n, tmp_n, prod;
  int count, max_count;
  e = end;
  max_count = 0;
  for(n = 0; n < e; ++n){
    cur_n = n;
    count = 0;

    while(cur_n >= base) {
      tmp_n = cur_n;
      prod = 1;
      while(tmp_n){
        prod *= (tmp_n%base);
        tmp_n /= base;
      }
      count++;
      cur_n = prod;
    }

    if(count > max_count) {
      max_count = count;
    }
  }

  *r = max_count;
  return;
}
```

Figure 29: Iterative multiplicative persistence C function

```
class MPDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:mult_persistence:1.0']

    def mult_persistence(self, end, base):
        self.write(0x10, end) # write input
        self.write(0x1c, base)

        self.write(0x00, 1) # start accelerator
        while self.read(0x00)&2 != 2: # wait for computation to finish
            pass
        return self.read(0x24) # return output
```

Figure 30: Iterative multiplicative persistence driver

```
def mult_persistence(end, base):
    max_count = 0
    for n in range(end):
        cur_n = n
        count = 0

        while cur_n >= base:
            count += 1
            tmp_n = cur_n
            prod = 1

            while tmp_n > 0:
                prod *= tmp_n%base
                tmp_n //= base
            cur_n = prod

        if count > max_count:
            max_count = count
    return max_count
```

Figure 31: Iterative multiplicative persistence Python function

C Technical Specifications

The replication of this work relies on the use of the correct versions of the tools we discussed. These are specified as follows:

- PYNQ: all bitstreams described were first downloaded, run and tested on PYNQ version 2.0, which runs Ubuntu version 15.10. The bitstream timing and performance tests we describe are done on PYNQ version 2.0. Nevertheless, we verify the bitstreams and workflows described to be compatible with PYNQ version 2.1—the most recent version to date. All interfacing and communication with the PYNQ board was done through the network, either via the board's Jupyter Notebook environment or via `ssh`.

- Python: the PYNQ 2.0/2.1 libraries require Python 3.6. Note that PYNQ 2.1 runs Ubuntu 16.04, and Python 3.6 is installed by default. In PYNQ 2.0, however, Python 3.6 must be installed from source.

- Vivado Design Suite: the bitstreams we describe are generated with Vivado Design Suite 2016.4, made available by Xilinx through the free WebPack license.

- Overleaf: thanks to Overleaf LaTex we were able to neatly organize and efficiently write up this research, and always keep it backed up. The final version of this work in LaTex took approximately 10s to compile to PDF.

- Coffee: a medium house coffee with whole milk cost us $2.75 at Tunnel City

Coffee in Williamstown, and kept us awake and motivated when we needed it most.