TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR FOR COMPILER CONSTRUCTION

Faculty of Computer Science

Center for Advancing Electronics Dresden
Chair for Compiler Construction

Master Thesis

A Lowering for High-Level Data Flows to Reconfigurable Hardware

Jiahong Bi

Born on: 26th August 1997 in Heilongjiang, China
Matriculation number: 4968272

to achieve the academic degree

Master of Science (M.Sc.)

First referee
Prof. Dr.-Ing. Jeronimo Castrillon

Second referee
Prof. Dr.-Ing. Diana Goehringer

Supervisors
Felix Suchert
Karl Friebel

Submitted on: 22nd February 2024

TECHNISCHE UNIVERSITÄT DRESDEN

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

TU Dresden Fakultät Informatik, Institut für Technische Informatik, Professur für Compilerbau

Task Description for Master Thesis

For:	**Jiahong Bi**
Degree program:	Informatik (Master of Science)
Matriculation number:	4968272
E-mail:	jiahong.bi@mailbox.tu-dresden.de
Topic:	**A Lowering for High-Level Data Flows to Reconfigurable Hardware**

MLIR is by now established as a versatile tool to create domain-specific IRs for various use-cases. Its main selling point is the vast extensibility which allows anyone to define their own IR dialects which can be mixed with other, existing IRs. Numerous actors in research and the private sector are incorporating it in their compilation flows and leverage the rich environment created by its users.

The DFG dialect enables the expression of high-level data flow by modelling data flow nodes and their instantiations as well as edges between the nodes. When lowered to code run on a CPU, all instantiated nodes are executed concurrently. Channels linking the nodes are unbounded and blocking, helping to facilitate the scheduling of individual nodes.

However, to this date no research has been conducted on lowering the DFG dialect to accelerator code. This thesis should address this problem by developing such a backend. It should analyze the necessary changes to the runtime semantics and adapt them for compatibility with FPGA architecture. Different kernels written in MLIR and composed using the DFG dialect should then be able to be lowered to a hardware target. To evaluate the backend developed, applications comprised of such kernels should then be executed on accelerator hardware and compared to a baseline. This thesis shall therefore include the following tasks:

1. Develop an MLIR-integrated backend for the DFG dialect that generates code that runs on hardware. This backend ideally leverages dialects developed as part of the CIRCT community if possible.
2. Analyze necessary changes to the semantics of the DFG execution model for accelerators.
3. Compare the MLIR backend to a baseline implementation to gauge its performance impact.

Given the architecture of the DFG dialect and preliminary research into this topic, we expect some of the following questions to arise during the work on this thesis and will optionally discuss them in as well:

(a) *Buffer Sizing*: The channels linking different nodes of the graph will have to have a fixed size in hardware. The appropriate sizing of the buffers could be discussed.

(b) *Dynamic Graphs*: So far, DFG has been intended for describing static graphs only. This thesis could investigate dynamic graph support and its implications.

(c) *Resource sharing*: Operators may contain implicit resource usages (adders, multipliers, . . .). Splitting an operator and adding a multiplexer node could implement resource sharing.

Start:	17.07.2023
End:	18.12.2023
1st referee:	Prof. Dr.-Ing. Jeronimo Castrillon
2nd referee:	Prof. Dr.-Ing. Diana Göhringer
Supervisor:	Felix Suchert, Karl Friebel

Prof. Dr.-Ing. Jeronimo Castrillon
(Professor in charge)

Statement of authorship

I hereby certify that I have authored this document entitled *A Lowering for High-Level Data Flows to Reconfigurable Hardware* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 22nd February 2024

Jiahong Bi

Abstract

The data flow programming paradigm has been a significant area of research in FPGA development. While FPGAs offer the potential for impressive performance gains alongside reduced power consumption. Despite these advantages, hardware design using hardware description languages presents considerable challenges, requiring expertise in both hardware and software design. This has led to the creation of several Domain-Specific Languages aimed at simplifying these complexities. Additionally, the MLIR infrastructure provides further support, enabling the creation of custom DSLs across various abstraction levels.

This thesis explores an integrated approach, utilizing these technologies to establish an MLIR dialect that encapsulates the Kahn Process Network model, along with a corresponding FPGA backend. Achieving this needs extensive work on specification of the dialect and lowering transformations within the CIRCT project, translating the DFG dialect down to a lower level hardware compatible representation. Finally, thorough evaluation validates the backend's correctness and demonstrates the significant performance benefits provided through this approach.

Contents

Symbols and Acronyms

AST	Abstract Syntax Tree
ASIC	Application-Specific Integrated Circuit
CFG	Control-Flow Graph
CPU	Central Processing Unit
iDCT	inversed Discrete Cosine Transformation
DFG	Data-Flow Graph
DFX	Dynamic Function eXchange
DIF	Dataflow Interchange Format
DMA	Direct Memory Access
DSL	Domain-Specific Language
DSP	Digital Signal Processor
FF	Flip-Flop
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
I/O	Input/Output
IP	Intellectual Property
IR	Intermediate Representation
KPN	Kahn Process Network
LLVM	Low-Level Virtual Machine
LUT	Look-Up Table
MLIR	Multi-Level Intermediate Representation
MoC	Model of Computation
MM2S	Memory to Stream
OOC	Out-Of-Context
RTL	Register Transfer Level
SoC	System on Chip
SDF	Synchronous Data Flow
SSA	Static Single Assignment
S2MM	Stream to Memory
UART	Universal Asynchronous Receiver/Transmitter
XSA	Xilinx Support Archive

1 Introduction

1.1 Motivation

Since Gordon Moore formulated Moore's Law in 1965, predicting a doubling of integrated circuit components approximately every 18 months, the electronics industry has witnessed exponential growth in computational power. However, this prediction has faced significant challenges as making the transistors smaller is reaching the physical limitations, signaling the beginning of the post-Moore era. In this new situation, the conventional strategy of boosting performance through transistor size reduction and increased transistor density is no longer viable.

This paradigm shift has led to the rise of heterogeneous computing, characterized by using different application-specific hardware designed to specific computational needs. Among these, Field-Programmable Gate Array (FPGA) has become a key player as the solution, distinguishing themselves in various high-performance computing domains, including specific computer vision algorithms. Compared to traditional Central Processing Units (CPUs) and Graphics Processing Units (GPUs), FPGAs demonstrate a better performance, particularly in peak performance, power efficiency, and sustained operational capability, as highlighted in the study [Qas+19]. Project EVEREST [Pil+21] presents a design environment tailored for processing extremely large data sets, exemplifying the growing trend of heterogeneous computing. This project effectively integrates various hardware platforms, including CPUs and FPGAs, showcasing the ability to leverage the strengths of different computing architectures to handle complex, data-intensive tasks.

Furthermore, the integration of High-Level Synthesis (HLS) tools and System on Chip (SoC) technologies has streamlined the deployment of computational kernels onto FPGAs, making the technology more accessible to developers. Leading vendors in this field, such as AMD Xilinx, offer an extensive range of reusable Intellectual Property (IP) cores, significantly simplifying the design and implementation process of FPGA-based solutions. These advancements make FPGAs more attractive to the industry, positioning them as a strategic choice for specific high-performance computing applications, ensuring both efficiency and performance optimization.

In the area of FPGA design, the data-flow acceleration kernel design appraoch is gaining traction for its enhanced memory access efficiency, which is a critical advantage in modern

computing. Data-flow applications are characterized by minimal data movement, directly translating to significant energy savings as they bypass the instruction fetching and decoding processes typically required in other programming methods. This approach's energy efficiency is particularly noteworthy, as highlighted in the paper [C P+19].

Data flow programming itself is not a new concept; its roots can be traced back to research conducted in the 1970s. Over the years, it has found applications in various domains ranging from big data processing to serving as the foundational principle behind certain visual programming languages [Sou12]. There are also projects aiming to translate imperative programs to data flow model, such as ConDRust [Suc+23]. A significant aspect of ConDRust is its focus on deterministic concurrency, which checks if the compiler preserves the semantics of the original sequential program. Today, numerous Model of Computations (MoCs) have been established, each tailored to represent multi-core stream processing activities efficiently. One such example is the Kahn Process Network (KPN).

Building on these developments, various programming languages have been developed, drawing inspiration from the data-flow approach. This includes Lucid, a groundbreaking functional programming language that implements the principles of data-flow programming. Through these advancements, FPGA design continues to evolve, adopting the data-flow models for the increasing demands for efficiency and performance in the computing world.

Taking advantage of the capabilities of FPGA hardware and innovative programming paradigms, researchers have developed a range of Domain-Specific Languages (DSLs) for specific types of application. Examples of such DSLs include OpenDF [Bha+08], OpenSpatial [Koe+18], and CAPH [SB14], each designed to make the programming for hardware-accelerated applications easier and more efficient. These DSLs work together with the well-known Hardware Description Languages (HDLs) and the C programming language, providing a diverse toolkit for developers working in the field of FPGA design and programming.

In this context, Multi-Level Intermediate Representation (MLIR) [Lat+21] stands out as a transformative technology, offering a unique approach to constructing compiler frameworks that are both reusable and extensible. MLIR is characterized by its use of Static Single Assignment (SSA)-based Intermediate Representation (IR), a structure that streamlines the compiler design process while introducing new levels of programming abstraction. A particularly noteworthy aspect of MLIR is its capacity to drive innovation, as evidenced by projects like CIRCT that uses the MLIR infrastructure for HLS, bridging the gap between high-level programming and hardware description. As a further possibility, the approach mentioned in [Sol+23] aims at efficiently moving data and fully exploiting CPU-FPGA bandwidth, which uses an MLIR-based DSL called Olympus [SP23] as well, which has multiple useful analysis and transformation passes. This project addresses the challenges in numerical simulations, particularly those that are memory-bound and massively parallel, making them suitable for FPGA acceleration.

In the scope of this thesis, the integration of MLIR with the KPN Model and FPGA hardware stands as a central objective. The aim is to develop a practical method for transforming high-level Data-Flow Graphs (DFGs) into designs that can be easily implemented on reconfigurable hardware, taking advantages of the potential of data-flow programming and FPGA technology to push forward the field hardware-accelerated computing.

1.2 Goal

The initial step in the process involves the implementation of a specific MLIR dialect named `DFG`, designed to express high-level data flow. This dialect makes it easier for the definition of nodes, instantiations, and the First-In First-Out (FIFO) channels that establish connections between each pair of nodes. A critical aspect of this implementation is the concurrent execution of all instantiated nodes, allowing for simultaneous data processing and flow throughout the network.

In the theoretical model of this system, channels are considered to be unbounded, providing unrestricted capacity for data transmission between nodes. However, when transitioning from the high-level representation to a practical FPGA implementation, it becomes necessary to introduce channel size semantics. This addition is crucial for managing the finite resources available on FPGA hardware, ensuring that the data flow can be accurately and efficiently mapped onto the physical components of it.

In the scope of this thesis, clear and precise goals has been set to guide the research and development efforts. These are crucial as they provide a road map for the study, ensuring that each step taken is aligned with the ultimate goals. The objectives are as follows:

- **Develop an MLIR Backend for `DFG` Dialect**: The primary goal is to create a backend within the MLIR framework specifically tailored for the `DFG` dialect. This backend will be responsible for generating code that is capable of running on FPGA hardware. Our goal is to incorporate this backend seamlessly into the existing CIRCT project, contributing to its extensive set of tools and capabilities. By doing so, we aim to enhance the CIRCT project with the ability to handle high-level data flow representations, smoothing their transition to FPGA-compatible formats.
- **Analyze and Adapt the `DFG` Execution Model**: A critical component of this research involves a thorough analysis of the `DFG` execution model. We intend to investigate its semantics and operational principles to identify areas that require modification or enhancement. The goal is to adapt the `DFG` execution model to ensure its compatibility with both CPU and FPGA hardware environments. This adaptation is essential to create a flexible system capable of fitting in the unique characteristics and requirements of different hardware platforms.
- **Performance and Correctness Evaluation**: Once the MLIR backend for the `DFG` dialect and the adapted execution model are in place, the next goal is to conduct a comprehensive evaluation of the system's performance and correctness. We will compare the performance of FPGA-compatible implementation with a baseline implementation to evaluate the effectiveness of our work. This comparison will provide valuable insights into the efficiency and effectiveness of `DFG` system, allowing us to quantify the performance gains achieved. Additionally, we will verify the correctness of the results generated by `DFG` system, ensuring that the transformation to FPGA hardware codes does not compromise the accuracy and reliability of the computations.

1.3 Structure of the Work

In the forthcoming sections of this thesis, a detailed and structured exploration will be provided, guiding the reader through the various stages and components of the research. The thesis is organized into distinct chapters, each focusing on specific aspects and elements of the study.

Chapter 2 serves as an introduction, where the reader will be familiarized with the basic concepts and foundational knowledge necessary for a comprehensive understanding of the thesis.

Following this, Chapter 3 delves into an analysis of related works in the field. This chapter will present case studies of similar research initiatives, highlighting their methodologies, outcomes, and the key differences that set them apart from this thesis. This comparison will provide context and demonstrate the unique contributions of the current study.

Chapter 4 represents the key part of the thesis, where the focus shifts to the implementation approaches and key methodologies employed to perform the lowering of high-level data flow representations to FPGA-compatible formats. This chapter will offer an in-depth examination of the decision-making processes and techniques that have been applied to make the transformation smoother and more efficient.

In Chapter 5, the thesis shifts from theory to practice, showcasing the experimental test kernel and presenting the results obtained from both hardware platforms. This chapter aims to validate the proposed methods and demonstrate their practical applicability.

Finally, Chapter 6 brings the thesis to a close with a summary of the key findings and discussions of the research. This concluding chapter will reflect on the study as a whole, considering the implications of the results and exploring potential future research and development.

2 Background

2.1 MLIR and CIRCT

MLIR [Lat+21] stands as a powerful and flexible tool in the field of compiler technology, showcasing its strengths in two dimensions:

- **Compiler Infrastructure**: In its first role, MLIR serves as a robust compiler infrastructure, offering developers a simple way to design, implement, and experiment with DSLs. This capability of MLIR is extremely useful for those looking to tailor programming languages to specific application domains, ensuring that the unique requirements and challenges of these domains are well addressed. MLIR's infrastructure is designed to simplify the rapid development cycles, allowing for quick iterations and modifications, which is crucial in the ever-evolving field of programming languages and compiler technologies.
- **Compiler Intermediate Representation**: Beyond its role as a compiler infrastructure, MLIR also functions as a form of compiler IR. In this character, MLIR shares a kinship with traditional SSA representations, such as the Low-Level Virtual Machine (LLVM) IR. This similarity is more than just an outlook, as MLIR embraces the fundamental principles of SSA representations, ensuring a familiar and robust environment for compiler developers. However, MLIR distinguishes itself by providing enhanced capabilities to handle multiple levels of abstraction, making it a flexible tool for a wide range of compiler optimization and code generation tasks.

At its core, MLIR is engineered to serve as a bridge between high-level Data-Flow Graph code structure and target-specific code. This ability to seamlessly transition from abstract, high-level representations to concrete, executable code is one of MLIR's standout features, and it plays a crucial role in enabling the efficient implementation of DSLs and other programming constructs. [Lat+21]

To unlock the full potential of MLIR and to design one's own dialect within this framework, it is important to understand some foundational concepts of MLIR. These concepts form the building blocks of MLIR's architecture, guiding developers through the process of creating, manipulating, and optimizing IR. By mastering these concepts, developers can leverage MLIR's

capabilities to its fullest extent, encouraging innovation and efficiency in compiler design and implementation.

Listing 2.1 MLIR High-Level Structure

```
// A function operation has one Region.
func.func @foo(%arg0: i32, %arg1: i32) -> i32 {
// Block in function region (implicit ^bb0)
    // Operation returns Value
    %0 = arith.cmpi eq, %arg0, %arg1 : i32
    // Multiple Regions in one Operation
    %1 = scf.if %0 -> (i32) {
        scf.yield %arg0 : i32
    } else {
        scf.yield %arg1 : i32
    }
    return %1 : i32
}
```

- **High-Level Structure**: uses a graph-like composition of nodes, known as *Operations*, and edges, named *Values*, to represent computations and the flow of data. Operations are the computational tasks, each capable of producing multiple results, possessing a unique set of attributes and operands, and are categorized into different functionalities across various upstream dialects. Values, serving as the edges in this graph, are generated either as the result of an operation or as a block argument, each bound to a specific *Type* defined by MLIR's extensive type system. Operations exist within Blocks, which are ordered sequences of operations representing straight-line code with the ability to branch to other blocks, all encapsulated within *Regions*, which is containers maintaining an ordered collection of blocks. The transformation and optimization of these components are managed through compiler *Passes*, routines designed to analyze, optimize, or transform the operations and their organizational structure. A visual representation of these interrelations and the overall structure of MLIR would be shown in Listing 2.1.
- **Dialect**: serves as a powerful and flexible means of interaction and expansion within the MLIR ecosystem, allowing users to define new *Operations*, *Attributes* and *Types*. Each dialect has its own unique namespace. For instance, the *Func* dialect has the namespace *func*. As shown in Listing 2.1, different dialects can coexist harmoniously within a single module. MLIR also makes it easier to switch between and within these diverse dialects through the use of *Passes*-routines. In this thesis, a variety of MLIR upstream dialects, including *Arith*, *Func*, as well as some from the CIRCT project, are used as target dialects in the process of lowering from the `DFG` dialect.
- **Attribute**: plays a crucial role in MLIR, especially in scenarios where variables are not permitted. They act as a vessel for defining constant data on operations, enhancing the expressiveness and functionality of the dialect. For example, the `arith.cmpi` operation needs a comparison predicate, which is stored within its own attribute dictionary. MLIR has a wide range of built-in attributes for common needs, such as `IntegerAttr` for operations involving integer numbers and boolean values. However, when these built-in attributes fall short of meeting the specific requirements of a dialect design, MLIR gives users the capability to define their own custom attributes.

- **Type**: is another fundamental aspect of MLIR, with every Value being assigned a Type by its extensive type system. Much like attributes, the type system is highly open-ended and customizable, enabling the definition of application-specific custom types to suit various needs. Importantly, there are no restrictions on the number of types that can be defined, nor on the level of abstraction they represent. In the context of this thesis, two custom types are introduced for the `DFG` dialect, details of which will be elaborated in Chapter 4.
- **Pass**: is crucial for implementing transformations both within the same dialect and across different dialects, similar to to the optimization passes found in LLVM. These passes can include both analysis and transformation functionalities, providing a robust infrastructure for rewriting operations that don't follow specified rules into so called *legal* ones. Chapter 4 of this thesis is dedicated to providing a detailed explanation to the various passes employed in the transformation process from `DFG` dialect to FPGA hardware code.

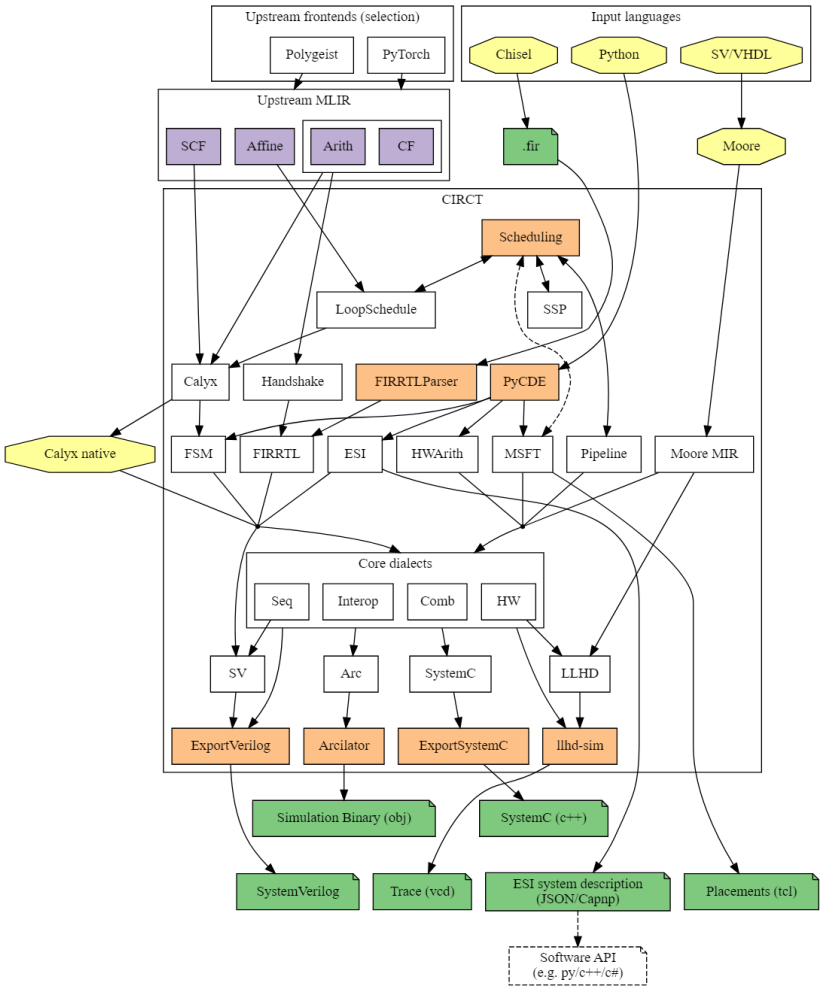


Figure 2.1: CICRT Project Overview [gro23]

The arrival of MLIR has inspired the development of numerous innovative open-source projects, each tailored to work in different scenarios and applications. A notable example of such innovation is the CIRCT project, which stands as a cornerstone for this thesis, particularly in the implementation of the FPGA backend. The acronym CIRCT encapsulates its meaning, standing for *Circuit IR Compilers and Tools*, and it is dedicated to providing various dialects

designed for representing hardware circuits across multiple levels of abstraction. These dialects play a crucial role in different stages of the hardware design process.

As shown in Figure 2.1, CIRCT offers a comprehensive top-level overview, demonstrating a selection of its most frequently used dialects. These include *Comb* for combinational logic, *FSM* for finite state machines, *Handshake* for high-level synthesis, *HW* for hardware description, *Seq* for sequential logic, and *SV* for SystemVerilog representation. This collection of dialects ensures that users have access to a flexible toolkit for hardware design, satisfying a wide range of requirements and preferences.

Beyond the dialects, CIRCT is also equipped with a variety of functional Passes, extending its utility and adaptability. These Passes are useful for users aiming to construct a complete development workflow, making the transition from high-level programming languages to the hardware circuits much easier. CIRCT supports this process through its compatibility with different front-ends and backends, ensuring that users can customize their development flows according to the specific needs.

For those deciding to use upstream MLIR as their input, CIRCT takes it a step further by providing *hlstool*: an executable binary capable of running HLS and generating split SystemVerilog files. This feature highlights CIRCT's role to bridging the gap between high-level design and hardware implementation, establishing its importance in the field of hardware design and development.

2.2 Dataflow MoC and KPN

In the domains of computer science, particularly within the area of computability theory and computational complexity theory, a MoC plays a important role. It serves as a conceptual framework that describes the process by which the output of a mathematical function is derived from its given input. A MoC outlines the organization and interaction of various fundamental elements, including units of computation, memory components, and communication mechanisms.

Communication/ organization of components	Shared Memory	Message Passing	
		Synchronous	Asynchronous
Undefined components	Plain text or graphics, use cases		
Communicating Finite State Machine	StateCharts		SDL
Data Flow	Scoreboarding	N/A	***KPN***, SDF
Petri Nets	N/A	C/E nets, P/T nets, ...	

Table 2.1: Overview of MoC [Mar21]

The book [DLa23] sheds light on this subject, providing insights and classifications of different MoCs. These classifications are based on distinct communication methods and the specific ways in which the components within the system are arranged and interact with one another. To

understand and give an easy comparison of these various MoCs, they have been systematically categorized and presented in Table 2.1.

In the context of this thesis, the data flow MoC is chosen, specifically the KPN, as the core conceptual framework for developing the `DFG` dialect. Originally proposed by G. Kahn in 1974 [Gil74], the KPN presents a simple yet robust schema for parallel computing. At its core, a KPN is visualized as a directed graph, consisting of *processes* represented as nodes, and *communication channels* shown as edges. Each process operates independently of the others, encapsulating its own data and following a singular, sequential flow. Communication between these processes is exclusively conducted through messages sent over the channels, with direct data sharing between processes being strictly prohibited. These channels function as limitless FIFO queues, capable of handling multiple input and output channels for every process. Importantly, each channel is characterized by a unique sender and receiver process at its each end, as detailed in [Vrb+09].

Figure 2.2: KPN Structure

While the theoretical model of KPN abstracts channels with infinite storage, this presents a practical challenge in real-world applications. To address this, *capacity* semantics is introduced to the channels, providing a more feasible and hardware-friendly approach for their implementation. This adaptation is crucial for the process of hardware lowering, which is explained in details in Chapter 4 of this thesis. For a clearer understanding of the KPN's structure, Figure 2.2 is helpful. In this figure, directed edges with dots on them represent the data flow, circles symbolize the processes, and the square represents a FIFO channel.

2.3 FPGA Basics and Elastic Circuit

Introduced to the world in 1985, FPGAs have become increasingly popular due to their great reusability and flexibility. When compared with CPUs, FPGAs wins with its higher performance and configurability. On the other hand, they hold a clear advantage over Application-Specific Integrated Circuits (ASICs) in terms of development time, significantly reducing both engineering costs and time to market. Additionally, when compared to GPUs, FPGAs show greater power efficiency. [GK19].

To provide a comprehensive overview of FPGA architecture, Figure 2.3 shows a high-level overview. Generally speaking, regardless of different vendors, an FPGA consists of the following critical components:

- **Programmable Logic Block**: serves as the base of logic implementation, where the primary computational and logical functions are executed, which consists normally Look-Up Tables (LUTs) and Flip-Flops (FFs).
- **Routing Interconnect**: is inclusive of Interconnect and Switch Boxes, establishes the necessary connections between various Logic Blocks, ensuring seamless communication and data flow.

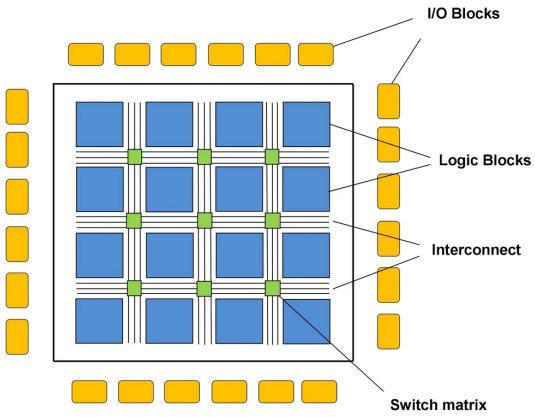


Figure 2.3: FPGA Architecture [BP21]

- **Input/Output (I/O) Block**: acts as the interface between the FPGA and external peripherals, enabling the FPGA to interact with, receive data from, and send data to external devices and systems.

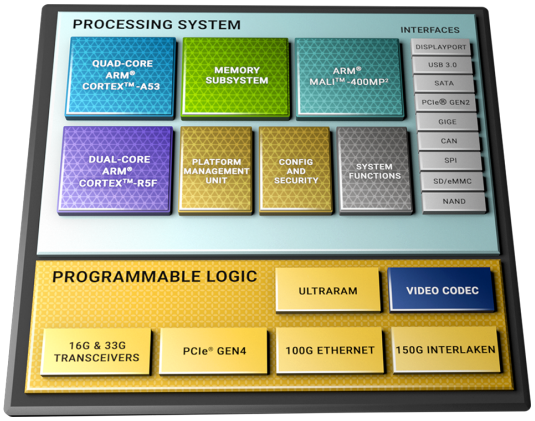


Figure 2.4: ZYNQ UltraScale+ Architecture [Xil23]

In the contemporary field of FPGAs, the resource on board extends beyond the conventional Programmable Logic. Leading vendors in the industry, such as AMD Xilinx and Intel, have revolutionized FPGA capabilities by incorporating ARM cores directly onto the chips, effectively creating SoCs. Taking the test board used in Chapter 5 as a reference, and specifically focusing on the core demonstrated in Figure 2.4, Xilinx has integrated different components for different applications. This includes a quad-core ARM Cortex-A53 applications processor, a dual-core Cortex-R5 real-time processor, a Mali-400 MP2 graphics processing unit, a 4KP60 capable H.264/H.265 video codec, and 16nm FinFET+ programmable logic, as documented in [Xil23].

This SoC architecture, when compared to traditional FPGAs that only contain programmable logic, opens the door to more possibilities. Using Xilinx's comprehensive suite of tools: Vivado, Vitis, and Vitis HLS, developers can engage in hardware-software co-design, allowing for a simple control of data on the programmable logic via data transfers to and from the processing system. This is made possible through advanced technologies such as Direct Memory Access (DMA).

Furthermore, the adoption of Dynamic Function eXchange (DFX) in this architecture greatly increases flexibility, and holds the potential to enhance energy efficiency across various applications. This integration of ARM cores and advanced design tools represents a significant step forward in the evolution of FPGA technology, encouraging both innovation and efficiency.

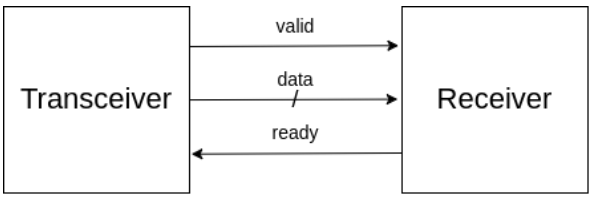


Figure 2.5: Handshake Protocol

When it comes to the implementation of data flow circuits on FPGAs, the concept of an elastic circuit interface, as shown in Figure 2.5, often serves as a foundational element [Ulm22]. This interface is characterized by the incorporation of two signals: *valid* and *ready*. These signals play a crucial role in controlling the transfer of data across the interface.

For a successful data transfer to occur, both the valid and ready signals must be simultaneously asserted. Interestingly, this assertion can transpire within the same clock cycle or span across different clock cycles, providing a degree of flexibility in the timing of data transfers.

On one end of the interface, there is the master or transceiver component. This entity is responsible for initiating the data transfer, and it is vital that it waits for the ready signal to be asserted before proceeding. On the other end, there's the slave or receiver component, which is in a position to accept the data. However, this can only happen when the valid signal is in an asserted state.

This mechanism of conditional data transfer, governed by the valid and ready signals, stands as a critical component of the implementation strategy for `DFG` dialect, as detailed in Chapter 4. By using this protocol, a synchronized and reliable exchange of data within the data flow circuit is ensured, aligning with the main goal of the FPGA-based implementation.

3 Related Work

3.1 Handshake Dialect

In the CIRCT project, the handshake dialect [NJ20] has been introduced with the goal of representing asynchronous, independent processes that communicate data through FIFO channels, drawing theoretical inspiration from the KPN model. As per ARM's specifications for the AXI-4 Stream Protocol [ARM10], the handshake dialect encapsulates all of its inputs and outputs within a handshake interface, incorporating two 1-bit signals, valid and ready, as shown in Figure 2.5.

The handshake dialect is currently implemented to define a set of data flow operators, assisting the transformation from a Control-Flow Graph (CFG) model to a data flow abstraction. Here is a list of some of the most commonly used operations within the handshake dialect:

- **BufferOp**: functions as a storage unit capable of holding a specified number of elements, serving a similar purpose to registers.
- **ForkOp**: takes input data and distributes it to all ready receivers. The fork operation will not accept new input until all receivers have consumed the current input.
- **JoinOp**: is a control-only operation that relies solely on valid and ready signals. It waits for all inputs to arrive before producing an output.
- **BranchOp**: is similar to a control flow branch, which has two outputs. It produces a token on the output port that matches the boolean condition.
- **MergeOp**: serves as a counterpart to BranchOp. It non-deterministically directs any input to its output.
- **SinkOp**: continually consumes inputs, drops them, and asserts the ready signal.

While it is possible for users to directly write MLIR code in the handshake dialect, it is important to note that not all code written this way may be executable. Alternatively, code can be written in the upstream MLIR, followed by the use of the `-lower-cf-to-handshake` pass in CIRCT once the code is transformed to the `cf` dialect, which functions as control flow. The handshake group has not only defined the dialect but also provided a suite of passes as the solution for generating a correct data flow model.

Among these passes, the `-handshake-insert-buffers` transformation is one of the most important. By default, this transformation inserts size-two buffer operations into the graphs, aiming to prevent deadlocks and enhance performance. However, this introduces a notable consideration: the number of buffers. Too many FIFOs can lead to increased area usage, potentially becoming a resource constraint.

The challenge is, when utilizing the passes that the handshake group provides, including the `-handshake-insert-buffers` transformation, developers may not have the flexibility to adjust the number of buffers. This underlines the importance of careful design and optimization, particularly when working with resource-intensive applications and aiming to create a balance between performance and resource utilization on FPGA architectures.

Listing 3.1 Handshake Example

```
handshake.func @foo(%arg0: i32, %arg1: i32, %arg2: none, ...)
    -> (i32, none)
{
  %0 = buffer [2] seq %arg2 : none
  %1 = buffer [2] seq %arg1 : i32
  %2 = buffer [2] seq %arg0 : i32
  %3:2 = fork [2] %1 : i32
  %4 = buffer [2] seq %3#1 : i32
  %5 = buffer [2] seq %3#0 : i32
  %6:2 = fork [2] %2 : i32
  %7 = buffer [2] seq %6#1 : i32
  %8 = buffer [2] seq %6#0 : i32
  %9 = arith.cmpi eq, %8, %5 : i32
  %10 = buffer [2] seq %9 : i1
  %11:3 = fork [3] %10 : i1
  %12 = buffer [2] seq %11#2 : i1
  %13 = buffer [2] seq %11#1 : i1
  %14 = buffer [2] fifo %11#0 : i1
  %trueResult, %falseResult = cond_br %12, %7 : i32
  %15 = buffer [2] seq %falseResult : i32
  %16 = buffer [2] seq %trueResult : i32
  sink %15 : i32
  %trueResult_0, %falseResult_1 = cond_br %13, %4 : i32
  %17 = buffer [2] seq %falseResult_1 : i32
  %18 = buffer [2] seq %trueResult_0 : i32
  sink %18 : i32
  %19 = mux %22 [%17, %16] : index, i32
  %20 = buffer [2] seq %19 : i32
  %21 = arith.index_cast %14 : i1 to index
  %22 = buffer [2] seq %21 : index
  return %20, %0 : i32, none
}
```

To transform a code snippet, such as the *foo* function shown in Listing 2.1, into a workable handshake dialect representation, several steps need to be followed. Initially, the code needs to be lowered into the cf dialect using the `-convert-scf-to-cf` pass. Following this, the `-lower-cf-to-handshake` and `-handshake-insert-buffers` passes are applied to transform

the code into the workable handshake dialect, with additional buffer operations included to ensure functionality and enhance performance.

When applying the handshake dialect passes, it's crucial to be careful, as simply inserting buffers might not be sufficient to ensure the correctness and optimal performance of the code. In the transformed code, listed in Listing 3.1, the handshake dialect introduces two control signals of none type for the inputs and outputs of the func operation. Additionally, a buffer is inserted for each parameter of the function and for the forked value. These buffers are crucial as they will later be lowered into handshake signals during the `-lower-handshake-to-hw` pass, ensuring the proper functioning of the code in a hardware context.

Moreover, the control flow within the transformed code is now expressed through a combination of `cond_br`, `fork` and `sink` operations. This transformation is essential for converting a general CFG into a data flow model, preserving the semantics of the original program while adapting it to the FPGA architecture.

The handshake dialect plays a key role in the CIRCT HLS workflow due to its features. However, it's important to point out that the handshake dialect implicitly uses FIFO channels, providing no direct control over channel operations such as pull and push. Despite this, the handshake dialect remains an integral part of the thesis and its methodologies, with its significance and applications further explained in Section 4.3.

3.2 IaRa Dialect

The IaRa compiler and dialect stands out as a significant data flow compiler within the field of MLIR, focusing specifically on the abstraction of Synchronous Data Flow (SDF) model. It takes programs formatted based on the Dataflow Interchange Format (DIF) and transforms them into executable code through a series of lowering processes within MLIR, followed by scheduling.

If a developer has a kernel written in C/C++ language, they can use Polygeist [Mos+21], a tool that converts C/C++ code into MLIR representation. This conversion enables the integration of the kernel into the module as a DFG, opening up possibilities for co-optimization passes. This also allows for the potential application of low-level LLVM optimizations, such as function inlining.

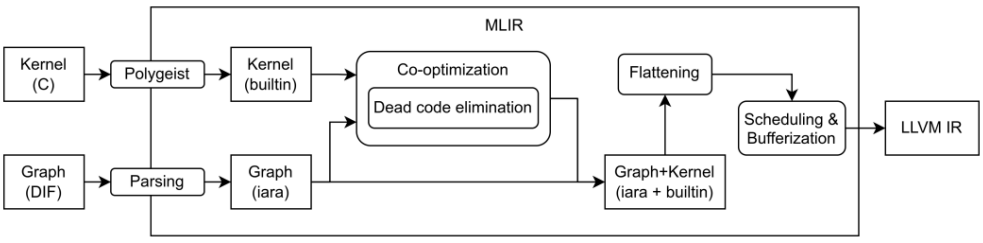


Figure 3.1: IaRa High-Level Structure [Cia22]

Furthermore, IaRa supports the external linking of kernels. This means that developers are not limited to using kernels converted from C/C++ within the MLIR environment; they can also make use of pre-compiled libraries. This flexibility is a significant advantage, as it allows for the integration of well-tested and optimized code from external sources.

To provide a comprehensive understanding of IaRa's capabilities and workflow, a high-level overview is presented in Figure 3.1. This overview shows the various stages and components involved in the IaRa compilation process, showcasing how it takes a program from its initial data flow representation all the way to an executable form, while providing options for optimization and external integration along the path.

The IaRa compiler introduces an innovative approach to DFG representation through its DIF front-end, which translates the human-readable textual format of SDFs into the IaRa MLIR dialect. This dialect has similarity to the DIF format, setting IaRa apart as a declarative implementation, different from the typical SSA IR. In the IaRa dialect, every element is referenced by name, using *SymbolName* attributes, creating a more straightforward generation process directly from the Abstract Syntax Tree (AST) of the input code. The IaRa abstraction for a SDF contains several key operations:

- **GraphOp**: represents a single hierarchical level of a SDF, encapsulating KernelOps, NodeOps, and EdgeOps within its block to detail the graph's structure. It also encodes the name and I/O ports, especially if sub-graphs are utilized. Notably, a single MLIR module can include multiple GraphOps, allowing for coexistence and interaction.
- **KernelOp**: functions as an external function, symbolizing one or more nodes within the SDF. It incorporates the name and I/O interface of the function, with the potential for parameterization to accommodate various use cases.
- **NodeOp**: serves as an abstraction for an SDF actor, encoding both a name and set of parameters. Within this operation, a reference is made to either a KernelOp, modeling a function, or a GraphOp, modeling a sub-graph, providing flexibility in representation.
- **EdgeOp**: represents a simple edge that connects two NodeOps, with the capability to carry SDF delay duration and value attributes, enhancing the expressiveness and functionality of the SDF representation.

Together, these operations can make a robust and flexible abstraction for representing and manipulating SDFs within the IaRa compiler, satisfying a wide array of data flow programming needs and scenarios. However, IaRa project offers a unique perspective on DFG modeling, giving a distinct approach compared to the KPN model utilized in the `DFG` dialect.

With this difference, the `DFG` dialect maintains the capability to represent models at a higher level than those in IaRa, as all SDFs can essentially be expressed as KPNs. Besides, IaRa's choice of a declarative format stands in contrast to the more conventional approach of using SSA for IR representations. These differences in design philosophies are crucial, especially in the context of integration with the CIRCT project. For a seamless integration and consistency across different parts of the project, it has been decided to adopt the standard SSA format within `DFG` dialect.

All these fundamental differences led to the conclusion that considering the broader context and the overarching goals of the project, the potential for extensive interactions with IaRa is limited.

3.3 Chisel and FIRRTL

FIRRTL, which stands for Flexible Intermediate Representation for Register Transfer Level (RTL), is a highly robust IR that plays a crucial role in simplifying hardware design. Developed under the [LIB16] specification, FIRRTL serves as a connecting bridge between Chisel [Bac+12], a hardware construction language, and CIRCT. Chisel itself is implemented using the Scala programming language, consisting of a set of libraries that define various hardware data types. Through Chisel, users can describe their hardware designs, which can then be converted into a hardware simulator written in C++, low-level FIRRTL IR, or Verilog HDL. FIRRTL represents the hardware circuit at an abstraction level immediately following Chisel's elaboration process, capturing the design before any simplification or optimization has been applied.

Function	**Chisel**	**FIRRTL**
Representation of module	Module	module
Register	Reg	reg
Wire for combinational logic	Wire	wire
A group of signals	Bundle	bundle
I/O ports for modules	IO	input/output
Connection of signals	:=	<=
Multiplexer	Mux	mux
Addition	+	add
Subtraction	-	sub
Multiplication	*	mul
Bitwise and	&	and
Bitwise or	|	or

Table 3.1: Comparison of Chisel and FIRRTL Semantics

Similar to Chisel, FIRRTL offers a variety of high-level data types such as `Vector` and `Bundle`, providing users with rich expressive capabilities for describing complex hardware structures. These high-level constructs can later be transformed into a more canonical form through a series of lowering transformations, enabling further optimization and analysis. Table 3.1 within the associated specification documentations highlights some of the most commonly used components of both Chisel and FIRRTL.

Listing 3.2 FIRRTL Dialect Overview

```
firrtl.circuit "bar" {
firrtl.module @bar(in %clock: !firrtl.clock, in %reset: !firrtl.uint<1>,
    in %io_i: !firrtl.uint<1>, out %io_o: !firrtl.uint<1>)
        attributes {convention = #firrtl<convention scalarized>}
{
    firrtl.strictconnect %io_o, %io_i : !firrtl.uint<1>
}
}
```

Delving deeper into the FIRRTL MLIR dialect within CIRCT requires the use of *firtool* for the conversion of FIRRTL IR to FIRRTL MLIR dialect. As listed in Listing 3.2, which presents

a straightforward function of a 1-bit stream I/O in the FIRRTL dialect, there are several significant aspects to take note of.

Firstly, firtool distinguishes between signed and unsigned integers by representing them as two distinct types. This differentiation is crucial as it ensures the generation of accurate and appropriate operations corresponding to the specific type of integer in use. This level of type specification contributes to the robustness and precision of the hardware description and subsequent synthesis.

Secondly, firtool incorporates clock and reset signals into the hardware description. This automatic insertion is a vital feature, as these signals are fundamental for the synchronous operation of digital circuits, ensuring that all components operate in a coordinated manner according to the clock cycles and can be reset to a known state when necessary.

Despite these beneficial features, the differentiation between signed and unsigned integers in FIRRTL's type system is one of the reasons for us not to use FIRRTL as a backend, a decision that is explained in Section 4.3. However, it is crucial to acknowledge the influence and inspiration drawn from Chisel and FIRRTL, particularly in the design of the FIFO channel are discussed in Section 4.6.1.

4 Methods and Implementation

4.1 Overview

In Chapter 3 several related works are presented and discussed. It was found out that none of them could fully be the target implementation for the `DFG` dialect. However, some of them does provide useful design ideas and reusable components.

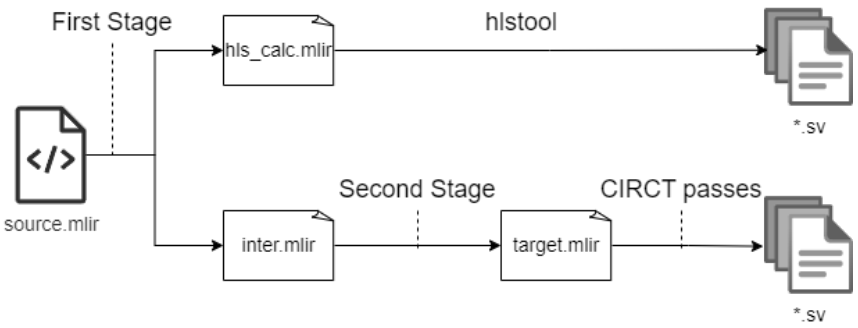


Figure 4.1: FPGA Backend Workflow

In this chapter the implementations are explained step by step, from the design of `DFG` dialect to the generation of target files. Figure 4.1 provides an overview of how the FPGA backend operates within the `DFG` dialect context. The process begins with the source code, which describes a KPN using the specific dialect in question. As a result several SystemVerilog files are generated, which can be directly used as the input of other design tools from any vendor, such as Vivado or Quartus.

4.2 DFG Rationale

The `DFG` dialect based on MLIR aims to represent a KPN, consisting of various nodes, edges, and their interconnections. The primary goal of this thesis is to develop an FPGA backend for the `DFG` dialect, as detailed in Section 1.2. To achieve this, it's really important to fully understand the fundamental components of the `DFG` dialect, including its custom operations and type system. Additionally, utilizing the FPGA backend requires a specific syntactical

structure, which may differ slightly from that needed for a CPU backend due to the constraints imposed by hardware design principles and the application of `DFG` lowering passes.

Compared to IaRa, as discussed in the previous chapter, the `DFG` dialect also provides operations to describe a DFG model in MLIR. However, unlike IaRa, it uses the standard SSA format, aligning with the conventions of most MLIR dialects. To manage the data flow direction, two custom types are defined: `!dfg.input<type>` and `!dfg.output<type>`. These types encapsulate a built-in MLIR type and are used to represent the types at both ends of an edge, as well as the I/O of any node in the graph. In the context of FPGA, these could correspond to any integer type.

Key operations within the `DFG` dialect include:

- **OperatorOp**: defines a node in a KPN, which can be linked to other operators through FIFO channels. It is marked as `IsolatedFromAbove`, creating a closed scope for its region, and it has a `NoTerminator` trait. It includes two argument lists, *inputs* and *outputs*, connected to channels, which are omitted if empty. The input types are all `!dfg.output<type>`, connecting to the output ports of channels, and vice versa for the outputs. The parser and printer assist in constructing the required types when defining this operation.
- **LoopOp**: defines a list of input or output ports to be closed later, terminating the operator and propagating to other operators for a complete shutdown. It must be the first operation within an OperatorOp if there is one and can only exist as a child of an OperatorOp. The types of ports are resolved by the parser and printer as well. Without LoopOp, the graph should shut down after the first execution; without ports as arguments of LoopOp, it monitors all ports of the parent operator. Hardware lowering only supports LoopOp for input ports for now.
- **PullOp**: pulls a data from a FIFO channel.
- **PushOp**: pushes a data into a FIFO channel.
- **ChannelOp**: creates a FIFO channel with a given type, returning an input port and an output as two results. Due to the lack of support for dynamic allocation in hardware, a *capacity* integer attribute must be added during creation. This operation is akin to the BufferOp in the handshake dialect but can be manipulated via PullOp and PushOp.
- **InstantiateOp**: instantiates an operator as a node in the graph with given inputs and outputs, which must match the operator's type and quantity. It can only be used in the top function definition.

Listing 4.1 lists the implementation of an addition function utilizing the `DFG` dialect, providing a example structure for the development of a KPN with an FPGA backend. The OperatorOp, as defined, creates operators with a variable number of inputs and outputs. Within this structure, a LoopOp is included as the first operation, containing other operator's logic. It is important to point out that the operator's logic follows the workflow of $Pull \rightarrow Calculation \rightarrow Push$ operations. The `DFG` dialect is able to co-exist various other operations defined in upstream MLIR and CIRCT dialects.

Upon defining a sufficient number of operators at the top-level, they can be instantiated and interconnected within a `func.func`, which has to be the concluding operation in the module. The data streams are implicitly represented through the inputs and outputs of FuncOp, and these are linked to ChannelOp via PullOp and PushOp. This particular code fragment describes

Listing 4.1 DFG Dialect Overview

```
dfg.operator @add
    inputs(%in0: i32, %in1: i32) outputs(%out: i32)
{
    dfg.loop inputs(%in0: i32, %in1: i32)
    {
        %0 = dfg.pull %in0 : i32
        %1 = dfg.pull %in1 : i32
        %2 = arith.addi %0, %1 : i32
        dfg.push(%2) %out : i32
    }
}
func.func @top(%in0: i32, %in1: i32) -> i32
{
    %q0_in, %q0_out = dfg.channel(4) : i32
    %q1_in, %q1_out = dfg.channel(4) : i32
    %q2_in, %q2_out = dfg.channel(4) : i32
    dfg.push(%in0) %q0_in : i32
    dfg.push(%in1) %q1_in : i32
    dfg.instantiate @add inputs(%q0_out, %q1_out)
                         outputs(%q2_in) : (i32, i32) -> i32
    %0 = dfg.pull %q2_out : i32
    return %0 : i32
}
```

a KPN comprising three FIFO channels, each with a capacity of four elements, interconnected to an addition operator. From an external perspective, the top module expresses as a black box with two input ports and a single output port.

4.3 Lowering Methodologies

This section delves into key aspects necessary for the transformation of a `DFG` program, as the example provided in Listing 4.1, into SystemVerilog files for FPGA deployment. Throughout the development process, there are three main design stages, transitioning from Handshake, progressing through FIRRTL, and resulting in the present approach that uses a variety of CIRCT dialects. Besides, several strategies have been implemented to make the transformation process more efficient, while also ensuring the integrity and correctness of the implementation during the lowering stage.

4.3.1 Target Dialects

In Section 3.1, the use of various operations in handshake, which are utilized to transform a CFG into its DFG equivalent, are detailed. Researchers who implemented the handshake dialect have assumed that any DFG abstraction within this context can be interpreted as a KPN. Additionally, the handshake dialect and `DFG` dialect share a commonality in their adoption of elastic circuit design principles. Given these similarities, handshake became the primary

candidate for the initial phase of this study, with the intention of lowering the `DFG` dialect to it. The initial hypothesis was that each operator in `DFG` program would correspondingly be lowered to a `handshake.func`, and each channel would be represented as a `handshake.buffer`. Successfully achieving this transformation would then make it easier to use various optimization and transformation passes to generate the SystemVerilog files for FPGA deployment.

Despite these initial prospects, challenges arose in relation to handshake's handling of buffers. Handshake is capable of associating function arguments with buffers and, during the lowering process, can convert these associations into a combination of data, valid, and ready signals, eventually generating a distinct hardware module for the buffer. However, a critical limitation exists in handshake's implicit expression of pull and push operations for FIFO channels, resulting in an imposed signal rate on any given port within the KPN represented in handshake. This is in contrast to `DFG` dialect, which gives users the flexibility to read from or write data to a channel. The difference of design in functionality finally led to reevaluation of this approach, resulting in the decision to move away from handshake to the search of alternative methodologies that more closely align with the unique characteristics and requirements of `DFG` dialect.

The second potential approach, a lowering from `DFG` dialect to FIRRTL dialect was explored, which serves as the IR for the Chisel hardware construction language. FIRRTL stands out in several aspects, notably due to its comprehensiveness and the availability of the firtool utility within the CIRCT ecosystem, which simplifies the direct conversion of FIRRTL programs into HDL code. Additionally, the possibility of reusing Chisel's FIFO implementation was an attractive prospect, potentially prevent the need to reinvent the wheel.

Despite these initial attractions, a deeper exploration of FIRRTL revealed certain redundancies that prompted this research to consider other options. FIRRTL, like Chisel, differentiates between signed and unsigned integers in its representations, a distinction that makes sense in the context of computation, as these two types of integers are handled differently. However, when lowering FIRRTL to core CIRCT dialects, it's true that both signed and unsigned integers are eventually converted to the *signless* integers used in both MLIR and `DFG` dialect. Utilizing FIRRTL would thus result in a loss of the flexibility provided by signless integers, and create ambiguity in distinguishing between signed and unsigned integers.

Another critical factor was the absence of Finite State Machine (FSM) abstractions in FIRRTL. While Chisel allows for the definition of `EnumType` to store FSM states and the creation of FSM logic in a manner similar to traditional HDL, FIRRTL does not provide the same level of flexibility for defining FSM logic. CIRCT, on the other hand, offers a dedicated FSM dialect designed specifically for such scenarios. The lack of FSM abstraction in FIRRTL means a significant challenge because when working directly with the FIRRTL MLIR dialect, it is unable to seamlessly mix it with other dialects. But FSMs play a crucial role in the FPGA backend for controlling the handshake interface. The FSM dialect in CIRCT not only solves this challenge but also enhances the flexibility by allowing for integration with other dialects, thereby simplifies the development of lowering passes.

A final point of conflict with FIRRTL was the needs to translate all operations into FIRRTL's built-in operations. Given that all operations would eventually be translated into core dialects, obtaining the hardware code for all operations present in the `DFG` dialect program through this intermediary step appeared both troublesome and redundant.

Despite these challenges, it is worth noting that the implementation of `Queue` class in Chisel did provide valuable insights for the custom channel implementation, as discussed in Section 4.6.1. After evaluating and deciding against the two approaches mentioned above, it is finally chose to lower the `DFG` dialect directly to the following CIRCT dialects: Comb for combinational logic, FSM for finite state machine creation, HW for hardware module definition, and SV for SystemVerilog syntax composition. Additionally, by utilizing the hlstool from CIRCT, it's possible to avoid the need for re-implementing HLS, the specifics of which are detailed in Section 4.4.1.

4.3.2 Intermediate Operations

As shown in Listing 4.1, when defining the top-level function, the input and output arguments of the FuncOp are treated as implicit data streams. These streams are represented using PullOp and PushOp, which interact with channels defined within the same scope. In the process of converting `FuncOp` to `HWModuleOp`, which is an abstraction of a hardware module in the HW dialect, the types in FuncOp are transformed into a trio of signals, establishing the specifications for the handshake interface.

However, this conversion is not straightforward. In Section 4.4.1, the lowering of OperatorOp was discussed, where the calculation part within the $Pull \rightarrow Calculation \rightarrow Push$ workflow is encapsulated into a new operation, subsequently instantiating it. Attempting to complete this entire conversion in a single pass would be impractical and goes against good design principles. This is because these steps should ideally be executed at different stages of the conversion process.

The MLIR group suggests maintaining the atomicity of each dialect and pass. In line with this guidance, a two-stage lowering process is proposed. In the first stage, the computational part within the operator is wrapped and FuncOp will be converted into HWModuleOp, altering only the types to an intermediate format using `DFG` unique type system. In the second stage, each port is unpacked to create the handshake interface, subsequently lowering the entire structure into the target dialects.

To facilitate this two-stage lowering process, two intermediate operations are introduced into `DFG` dialect, prefixed with `dfg.hw` instead of `dfg`, to distinguish them within the MLIR framework. The details of these two stages and the associated lowering processes will be explained in the subsequent sections.

- **HWInstanceOp**: represents the instantiation of the program encapsulated during the first lowering stage. It acts as a placeholder for what will eventually be transformed into an `InstanceOp` in the HW dialect.
- **HWConnectOp**: denotes the connections between the arguments of HWModuleOp and FIFO channels, ensuring type consistency across the connections.

4.3.3 Handshake Signals Transformation

Following the first stage of transformation, all child operations within ModuleOp adopt `!dfg.input` and `!dfg.output` as their input and output types. This transformation significantly

simplifies the subsequent conversion to the handshake interface, a process detailed in Section 2.3 and planned for execution during the second stage of transformation.

The top-level function is configured to receive `!dfg.input` types for its inputs and `!dfg.output` for its outputs. In contrast, as discussed in Section 4.2, OperatorOp exhibits reversed type configurations due to its connections to the channel's I/O. While this arrangement may seem bit puzzling at first, it is intentionally designed to ensure the correct direction of data flow, and it does not pose any challenges during the transformation from `DFG` dialect type to handshake signals.

The method applied for this transformation is straightforward. Regardless of the `DFG` type assigned to a port, if it serves as an input, two input ports (valid and bits) and one output port (ready) are generated. Conversely, for output ports, the original is replaced with one input port (ready) and two output ports (valid and bits). Each operation undergoing this transformation is as a result to become a HWModuleOp. To make this easier, the `hw::PortInfo` class is used to create the ports, specifying the name, direction, and data type for each. Following their creation, these ports are added to a Collection, which can subsequently be converted into an MLIR `ArrayRef`, ensuring seamless integration into the `build` functions of the HWModuleOp.

By following this approach, a consistent and efficient transformation process is ensured, setting the base of the successful implementation of the handshake interface in the second transformation stage.

4.4 Intermediate Lowering

4.4.1 Calculation Wrapping

The OperatorOp follows a specific workflow pattern: *Pull* → *Calculation* → *Push*. In the Calculation phase, `DFG` dialect is designed to be highly flexible, supporting nearly all other MLIR and CIRCT dialects, regardless of whether they possess their own regions. The primary strategy involves encapsulating everything apart from PullOp and PushOp, subsequently replacing them with HWInstanceOp of `DFG` dialect, which was introduced in the previous section. This approach is adopted to avoid the need for duplicating efforts in implementing HLS. Instead, this critical task is handed over to the hlstool tool provided by CIRCT, thereby simplifying the process. By employing this method, the concentration is the efforts on creating the FSM during the second stage of the lowering process. This stage is crucial as it involves translating the high-level abstract operations into a more concrete representation that can be directly synthesized into hardware.

Listing 4.2 shows the outcome of the first lowering stage applied to the operator listed in Listing 4.1. In this stage, all data acquired from the input channel via the PullOp operation is used as input arguments for a newly established FuncOp. This data is subsequently stored in a distinct MLIR file, named *hls_{Operator Name}_calc*, ensuring uniqueness due to the operator's unique nature.

Data derived from calculation operations, which are required to be pushed into the output channel through the PushOp operation, are generated as the return values for the new FuncOp. There is, however, a single exception to this rule: if the data to be output is acquired directly

Listing 4.2 Wrapped Operator

```
// The HWModuleOp will be produced from wrapped calculation
hw.module.extern @hls_add_calc(%in0: i32, %in0_valid: i1,
    %in1: i32, %in1_valid: i1, %in2_valid: i1,
    %clock: i1, %reset: i1, %out0_ready: i1)
        -> (in0_ready: i1, in1_ready: i1, out0: i32, out0_valid: i1)
// Wrap all the calculation operations into hls_add_calc.mlir
dfg.operator @add inputs(%arg0 : i32, %arg1 : i32)
                  outputs(%arg2 : i32)
{
    dfg.loop inputs (%arg0 : i32, %arg1 : i32)
    {
        %0 = dfg.pull %arg0 : i32
        %1 = dfg.pull %arg1 : i32
        %2 = dfg.hw.instance @hls_add_calc(%0, %1) : (i32, i32) -> i32
        dfg.push (%2) %arg2 : i32
    }
}
```

through the PullOp, it will not be included in the return values of the FuncOp. All other operations within the Operator remain unchanged during this process.

A noticeable change post-lowering is the addition of a `HWModuleExternOp` of HW dialect at the module's top. This operation serves to represent an external hardware module, not present in the current file. Given that all calculations have been encapsulated into a new file and are subject to HLS via the hlstool, this external module becomes necessary to represent the module instantiated during the operator's second stage of lowering. With the knowledge of how the input and output signals become post-HLS, creating such a module becomes a straightforward task.

As previously discussed regarding the handshake interface, all I/O ports are converted into three distinct signals. In this specific scenario, for the two inputs and one output of the encapsulated FuncOp, three sets of signals are created. Additionally, clock and reset signals are automatically included during the HLS process. Post-HLS, an extra valid signal is added to the module, standing for the module's start. This valid signal is set to high (represented as a true boolean value in code) during instantiation, as will be detailed in Section 4.5.1.

4.4.2 Top Function Connection

When focusing exclusively on the inputs and outputs of the top function, it can be entirely perceived as a black box module. This module possesses one or more FIFO input ports as its inputs, while its outputs are connected to other FIFO output ports. In the resultant SystemVerilog program later shown in Section 4.7, the top function's primary role is to instantiate various hardware modules, which in this specific context, are the channels and operators.

The program listed in Listing 4.1 demonstrates the use of PullOp and PushOp operations to establish connections between the inputs and outputs of the FuncOp and the requisite channels.

Listing 4.3 Intermediate Top Function

```
// Arguments type are FIFO channels I/O port type
hw.module @top(%in1: !dfg.input<i32>, %in2: !dfg.input<i32>)
    -> (out: !dfg.output<i32>)
{
    %in_chan, %out_chan = dfg.channel(4) : i32
    %in_chan_0, %out_chan_1 = dfg.channel(4) : i32
    %in_chan_2, %out_chan_3 = dfg.channel(4) : i32
    // Connect arguments with channels input port
    dfg.hw.connect %in1, %in_chan : i32
    dfg.hw.connect %in2, %in_chan_0 : i32
    dfg.instantiate @add inputs(%out_chan, %out_chan_1)
                         outputs(%in_chan_2) : (i32, i32) -> i32
    // Connect channel's output port to output
    hw.output %out_chan_3 : !dfg.output<i32>
}
```

These operations represents the data flows between the channels and the top function, ensuring that the inputs and outputs are correctly linked for the intended hardware interactions. This setup underscores the modular nature of the design, where the top function serves as a central hub, handling the interactions between different hardware components.

In the first stage of lowering code shown in Listing 4.3, the FuncOp is replaced with HWModuleOp, which is the target operation for the complete lowering transformation process. The PullOps are replaced with HWConnectOp, which serves to denote the actual connections of the input ports within the final top module. The output ports utilized in PushOp are directly used as operands for a newly created OutputOp, establishing a connection to the output ports of the top module.

Given that this is the top module, it is crucial to ensure the accuracy of both the type and the data stream direction post-lowering. The two ports connected via HWConnectOp must possess the type `!dfg.input`, and the OutputOp must be of the type `!dfg.output`.

It is worth pointing out that during this initial stage of lowering, no modifications are applied to either the ChannelOps or the InstantiateOps. However, with the intermediate OperatorOp and the top module operation now in place, the second stage of lowering can be proceeded, which will delve deeper into the transformation process, refining the hardware representation and moving closer to the final hardware description.

4.5 Operator Lowering

During the second stage of lowering, the operator undergoes a significant transformation, resulting in an HWModuleOp of HW dialect. This involves converting all its inputs and outputs into handshake interface signals, a process detailed in Section 4.3.3. Additionally, if there is a LoopOp within the operator that monitors the closing of certain input channels, a close signal is appended to each of those channels. Regardless of the presence of a LoopOp, a done signal is integrated into the output ports to propagate the closing behavior throughout the system.

Listing 4.4 Final Operator I/O

```
hw.module @add(%clock: i1, %reset: i1,
               %in1_valid: i1, %in1_bits: i32, %in1_close: i1,
               %in2_valid: i1, %in2_bits: i32, %in2_close: i1,
               %out_ready: i1)
    -> (in1_ready: i1, in2_ready: i1,
        out_valid: i1, out_bits: i32, out_done: i1) {}
```

Furthermore, both a clock and a reset signal are included into the I/O ports, ensuring the operator is fully equipped for synchronous operation. The operator, as shown in Listing 4.2, after this stage of transformation, resulting in a module that follows the handshake protocol and is ready for the subsequent stages of hardware synthesis and implementation as shown in Listing 4.4. This process is crucial for ensuring that the operator can properly interact with other hardware components and maintain correct data flow and synchronization across the system.

Due to the replacement of arguments during the transformation of the operator into an HWModuleOp, it is important to update the values that are used within this new module to ensure consistency and proper functionality. A crucial step in this process is the creation of a `hw::ConstantOp` that holds a boolean value of true. This operation is placed within the region of the operator module, and its purpose is to set the start control signal to a high state.

This action, as previously discussed in Section 4.4.1, is essential for initiating the module's operation, effectively signaling that the module is ready to start processing data.

4.5.1 Calculation Instantiation

The uniqueness of the calculation module wrapped during the second stage of lowering enables this backend to efficiently locate and instantiate the external module required using the `hw::InstanceOp` through a unified MLIR function, which is `walk`. The walk function navigates through the operation scope, accepting a lambda expression as its argument. For this specific scenario, a closure is crafted, aiming to identify the HWModuleExternOp that bears the designated name.

However, the instantiation process of the calculation module is complicated and requires careful handling. Initially, data necessary for calculations is retrieved via the PullOp. Furthermore, in scenarios where a LoopOp is included, it becomes important to determine the precise method to reset it, which is key to subsequent executions. It's important to point out that all these data and signals are controlled by a FSM, meaning they are essentially outputs generated by the FSM. In a scenario where the program is natively written in HDL, registers could be employed to establish connections. However, in MLIR, the entire structure is comprised of operations, with all the values being outcomes of these operations.

Given the sequential nature of the lowering process, wherein operations are transformed individually, a challenge arises when attempting to instantiate the calculation module. This is because the values needed from the FSM results are yet to be generated. To solve this issue, placeholders are introduced to serve as temporary values for these data. Following the creation of the FSM, the MLIR function `replaceAllUsesWith` is applied to revert these placeholders

back to their original intended values and then prune them in final lowering using `erase` function.

This approach ensures the accurate and timely availability of all crucial values, despite the transformations being executed in distinct stages throughout the lowering process. The placeholder strategy proves to be equally effective when applied to the FSM creation process, particularly when there is a need to use the results of the calculation module as outputs. This approach maintains the integrity of the data flow and signal control, giving a much easier way to handle the operator transformation.

4.5.2 FSM Creation

Given the handshake protocol's key role in ensuring synchronized communication between hardware components, it becomes crucial to establish a mechanism that guarantees the validity and readiness of signals before proceeding with any data read or write operations. The FSM serves as a highly efficient solution to address this requirement.

The Moore FSM model, shown in Figure 4.2, serves as a representation of the FSM that will be generated during the lowering process of a `DFG` operator. This FSM is important to the management of the state transitions and ensures the correctness of the handshake protocol. As following will delve into a detailed analysis of the states and transitions within this Moore FSM model:

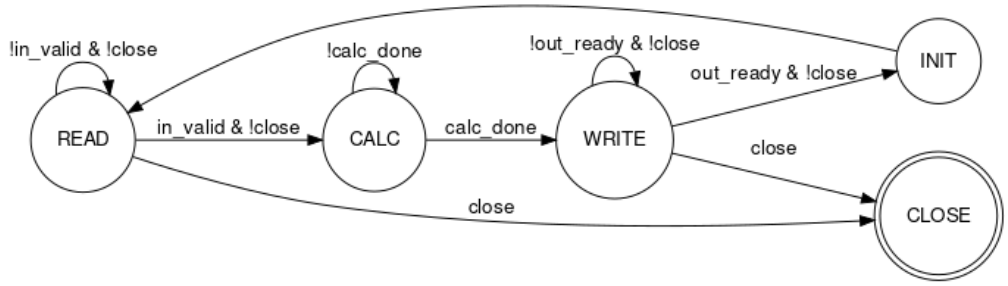


Figure 4.2: Operator FSM Model

- **INIT**: This is the initial state of the FSM, where all registers undergo an initialization process, being set to their default values. Following this initialization, there is an immediate transition to the READ state, with no conditions applied. Concurrently, a reset signal is generated and dispatched, intended for the usage by the calculation module, as discussed in last section.
- **READ**: In this state, the FSM awaits the opportunity to read from a channel. If the valid signal is not asserted, the FSM remains in this state, in a state of idleness. Conversely, if a valid signal is present and a closing signal is not, the FSM proceeds to extract the data from the channel and transition to the CALC state. If, however, a closing signal is asserted, a transition to the CLOSE state occurs, a scenario that is exclusively possible with the presence of a LoopOp monitoring this specific channel.
- **CALC**: This state is dedicated to awaiting the completion of the calculation module and the assertion of the corresponding signal indicating completion. It is crucial to note that this state operates independently of the close signal. This is because, once the input data is secured, the computation process can be executed, regardless of the state of the

closing signal, which propagates from the top. Therefore, there is always the potential to transmit the result into the channel subsequent to this state.
- **WRITE**: This state adopts a similar approach to the READ state, but with its focus shifted to awaiting the ready signal from the output channel. In the event that a LoopOp is present and no closing signal is asserted, a transition back to the INIT state is triggered, setting the stage for the next execution cycle. If this is not the case, the FSM transitions to the CLOSE state.
- **CLOSE**: Representing the terminal state of the FSM, once entered, the machine halts its operations. The FSM remains in this state until a reset signal is received, indicating a re-initialization of the machine.

For the implementation phase, the FSM dialect is used, which is a specialized dialect within CIRCT, engineered to represent abstractions tailored for an FSM. This dialect stands out due to its ability to: **1.** Offer a clear, structural description of states, transitions, and internal variables relevant to an FSM, thereby simplifying the analysis and transformation processes. **2.** Present a target-neutral representation of FSM, ensuring compatibility and seamless integration with other dialects. **3.** Collaborate efficiently with conversion passes, thereby enabling the FSM abstractions to be seamlessly lowered into hardware dialects, a crucial step for simulation and code generation activities later. [gro23]

Within the FSM dialect, several operations play key roles in the second stage of the lowering process, which are:

- **MachineOp**: represents a finite-state machine, encapsulating essential details such as the machine's name, the typology of machine states, and the input-output types. An attribute within this operation is designated for specifying the initial state. MachineOp contains a region in which internal variables and states can be created.
- **VariableOp**: serves to represent an internal variable within the state machine, complete with an initialization value. This is similar to a register in the HDLs.
- **StateOp**: describes a particular state within the state machine. It features an output region, terminated by an OutputOp of FSM dialect, to define the machine's outputs when in this state. Furthermore, it includes a transitions region, encapsulating all possible transitions from this state.
- **OutputOp**: expresses the outputs of a machine when it is in a specific state, ensuring that the operand types align with the output types specified for the state machine.
- **TransitionOp**: characterizes a state transition, including a symbol reference pointing to the subsequent state. It may contain an optional `guard` region, concluded by a ReturnOp of FSM dialect, which yields a boolean value representing the transition's guard condition. Additionally, it may also contain an optional `action` region, detailing the actions to be executed when taking this transition.
- **ReturnOp**: marks the conclusion of a region within a TransitionOp, returning values if the parent region functions as a guard region.
- **UpdateOp**: is employed to update a variable with a specific Value. The variable in question must be defined through a VariableOp of FSM dialect, and this operation is exclusively utilized within the action region of a transition.
- **HWInstanceOp**: represents a hardware-style instantiation of a state machine, including an instance name and a symbol reference to the machine. The inputs and outputs of this

operation must be the same of the instantiated machine, during which, it is important to assign the clock and reset as well.

In scenarios where an operator includes multiple PullOps and PushOps, it becomes necessary to introduce additional READ and WRITE states to fit the data fetching and output requirements. The sequencing of these states must align with the order specified in the operator's use of PullOp and PushOp. Moreover, the complexity increases when dealing with multiple close signals associated with different input channels. To address this, CLOSE transitions should be added appealing from the READ state, activated upon the assertion of a close signal for the monitored input channel this state are fetching data from. Similarly, transitions from the WRITE state should be introduced to handle scenarios where any of the monitored channels enter a closed state.

The calculation module, essential in this process, may produce multiple results. This translates to potentially having multiple `out_valid` and `out_bits` values appealing from the instance described in the preceding section. In the CALC state, it becomes important to ensure the capture of every valid signal and its corresponding data. However, a challenge arises as these values might become available at different clock cycles, and yet, once available, they persist for only a single clock cycle. To navigate this challenge, the following solution is proposed:

- **Logic OR Operation**: Execute a logic OR operation between the existing valid signal and the incoming valid signal. This ensures that, once the valid signals have arrived, they keep at a high state.
- **Multiplexer Utilization**: Employ a multiplexer to selectively route data from either the existing data or the incoming one, based on the state of the preceding valid signal. This design ensures that the data are updated exclusively when the valid signal from the instance is asserted, preventing data corruption or loss.

Listing 4.5 Simplified FSM Machine

```
fsm.machine @add_controller(input_types) -> (output_types)
    attributes {initialState = "INIT"}
{
    %valid = fsm.variable "valid" {initValue = false} : i1
    // define variables and constants
    fsm.state @READ output {
        fsm.output // the output list
    } transitions {
        fsm.transition @CALC guard {
            fsm.return // the guarding value
        } action {
            fsm.update // the variables
        }
        // other transitions
    }
    // other states
}
```

Upon applying this mechanism, the final step involves instantiating the designed machine within the operator using the HWInstanceOp of the FSM dialect. A simplified machine operation generated from Listing 4.2 is shown in Listing 4.5.

4.6 Top Function Lowering

Similar to the operator lowering process, the second stage of lowering involves a complicated transformation of the types present in the top module, as shown in Listing 4.3. Each type will undergo a conversion, resulting in the generation of three distinct handshake signals. In addition to this transformation, the creation of clock and reset ports becomes essential, alongside the introduction of close signals corresponding to each input channel, which is monitored by the LoopOp.

The creation of close ports comes from the practices of vendors such as AMD Xilinx, which provide IPs that come equipped with a *tlast* signal. Ensuring that when this signal serves as an input of Xilinx IP, it does not remain in a dangling state is of great importance, as a missing of these signals could result in unforeseen failures. It is crucial to acknowledge that in cases where an input channel isn't direct monitored by the LoopOp within the top module, a creation of close port for that channel in the argument list is unnecessary. For the top module will eventually terminate there will be always a done port appended in the outputs.

4.6.1 FIFO Queue Implementation

The second stage of lowering is important, with two of its most crucial components being: **1.** the creation of the FSM machine as detailed in Section 4.5.2, and **2.** the implementation of a lightweight and efficient FIFO queue to represent the ChannelOp at a low-level. As previously mentioned in Section 3.3, Chisel provides a high-level `Queue` class capable of generating a module in HDL. After discussions, it was decided to adopt this implementation while introducing some custom modifications into it. A queue characterized by a specific type and capacity will be generated by the lowering of ChannelOp, unless an identical queue has already been created. Listing 4.6 offers a simplified overview of the queue module derived from the ChannelOp defined in Listing 4.3, with comments indicating omitted code.

Breaking down the module, the I/O ports mirror those of the handshake interface, with the addition of close and done ports to simplify the propagation of closing behavior. Initially, constants are defined within the module, including true and false to set the `i1` value, as well as integers 0 and 1 with a bitwidth of $\lceil log_2(capacity) \rceil$ to reset and increment the index of the read and write pointer. If the capacity is not a power of 2, an additional value equal to $capacity - 1$ is defined to manage the behavior of pointers at the queue's end.

In terms of register definition, the SV dialect is utilized to create registers with `RegOp`, resulting in a `!hw.inout<built-in type>` type. Unlike HDL, these registers cannot be used directly. Instead, the `ReadInOutOp` is applied to access the stored value, a requirement in the SV dialect's specifications. For array register definition, the register is created with the given type enclosed in an `UnpackedArrayType`, offering a more flexible array representation than packed arrays and typically used to model memories. The `ArrayIndexInOutOp` enables reading

Listing 4.6 Queue Example

```
hw.module @queue_4xi32(%clock: i1, %reset: i1, %close: i1,
  %io_enq_valid: i1, %io_enq_bits: i32, %io_deq_ready: i1)
    -> (io_enq_ready: i1, io_deq_valid: i1, io_deq_bits: i32, done: i1) {
  // constants definition
  %want_close = sv.reg : !hw.inout<i1>
  %want_close_value = sv.read_inout %want_close : !hw.inout<i1>
  %ram = sv.reg : !hw.inout<uarray<4xi32>>
  %ram_read = sv.array_index_inout %ram[%ptr_read]
                    : !hw.inout<uarray<4xi32>>, i2
  %ram_data = sv.read_inout %ram_read : !hw.inout<i32>
  // Same definition for ptr_write, ptr_read, maybe_full
  %ptr_match = comb.icmp eq %ptr_write, %ptr_read : i2
  %empty_T = comb.xor %maybe_full, %true : i1
  %empty = comb.and %ptr_match, %empty_T : i1
  %full = comb.and %ptr_match, %maybe_full : i1
  %do_enq = comb.and %not_full, %io_enq_valid : i1
  %do_deq = comb.and %io_deq_ready, %not_empty : i1
  %next_write = comb.add %ptr_write, %c1_i2 : i2
  %next_read = comb.add %ptr_read, %c1_i2 : i2
  %not_same = comb.icmp ne %do_enq, %do_deq : i1
  // use xor to flip empty, full, want_close
  %enq_ready = comb.and %not_full, %not_want_close : i1
  %is_done = comb.and %want_close, %empty : i1
  sv.always posedge %clock {
    sv.if %reset { // use sv.passign to reset registers
    } else {
      sv.if %do_enq { // write io_enq_bits to ram and ptr_write increse
      } sv.if %do_deq { // ptr_read increase
      } sv.if %not_same { sv.passign %maybe_full, %do_enq : i1
      } sv.if %close { sv.passign %want_close, %true : i1 }
    }
  }
  hw.output %enq_ready, %not_empty, %ram_data, %is_done : i1, i1, i32, i1
}
```

a single value, maintaining the type of a scalar register. Using these mechanisms, the following registers are defined:

- **want_close**: stores the close signal until all data in the queue is flushed.
- **ram**: holds all enqueued data.
- **ptr_write**: keeps track of the write pointer position.
- **ptr_read**: tracks the read pointer position.
- **maybe_full**: indicates whether the queue might be full, utilized to check for fullness upon data enqueueing and prevent blocking situations when the queue is at capacity.

With the established registers, it's efficient to determine the queue's status, whether it is empty or full. This determination is crucial as it directly influences the ability to enqueue new data

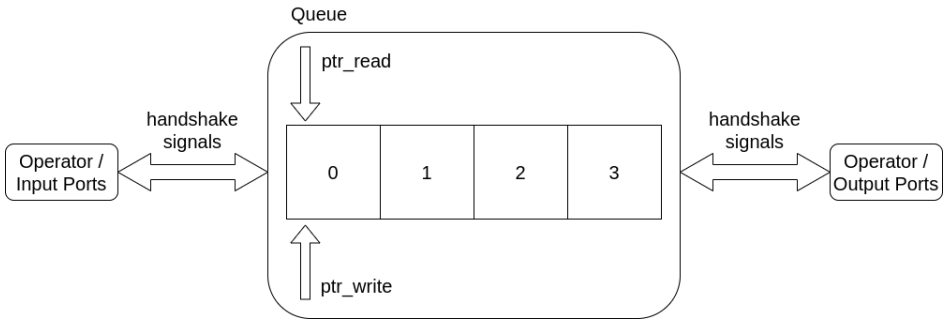


Figure 4.3: Queue Module Overview

into or dequeue existing data from the memory. Figure 4.3 shows a visual overview of the initial queue module listed in Listing 4.6 and its connections.

- **Enqueue Operation**: Whenever there is a data enqueue operation, the ptr_write (write pointer) will increment, indicating the addition of new data to the queue. However, if the want_close register has stored a close signal, the queue will stop accepting any more data from the input port, essentially locking the queue for new entries.
- **Dequeue Operation**: For data dequeue operations, the ptr_read (read pointer) will increment, indicating that data is being retrieved and removed from the queue. The queue will continue to output data until it is empty, at which point the done signal will be asserted. It is critical to note that even if a close signal has been received, the queue can still output remaining data until it becomes empty.
- **AlwaysOp and Sequential Logic**: The AlwaysOp is used to create a SystemVerilog always block, essential for implementing the clocked logic of the queue module. This block ensures that register values are updated synchronously at the rising edge of the clock signal, providing consistency and predictability throughout the circuit. The rising edge of the reset signal is also used for initializing the module, aligning with the rest of the `DFG` modules and differing from the conventional falling edge reset used in AMD Xilinx IP cores. This alignment is crucial as the hlstool used for the calculation module's HLS produces code that utilizes the rising edge for reset. Adapting to this rising edge reset mode avoids any potential issues.
- **IfOp and PAssignOp**: The IfOp creates an if-else region, similar to SystemVerilog's if-else structure, simplifying conditional logic within the hardware design. The PAssignOp is equivalent to a non-blocking procedural assignment statement (e.g. `x <= y;`), ensuring that assignments within the always block do not block the execution of subsequent statements, maintaining the integrity of the synchronous design.

To ensure the correct functionality of the created queue module, it is subjected to verification using simulation software. This process involves evaluating the SystemVerilog file that defines the queue module, checking the waveform of various signals and data relevant to this FIFO queue.

Figure 4.4 is expected to demonstrate the simulation results, showing the behavior of the I/O ports within the queue module. Before delving into the simulation, it's crucial to note that a global reset is active for the initial hundred nanoseconds, which is a safety measure to prevent any potential logic failures according to AMD Xilinx. During the simulation, all signals are applied at the falling edge of the clock cycle. This strategy is deliberately chosen to eliminate

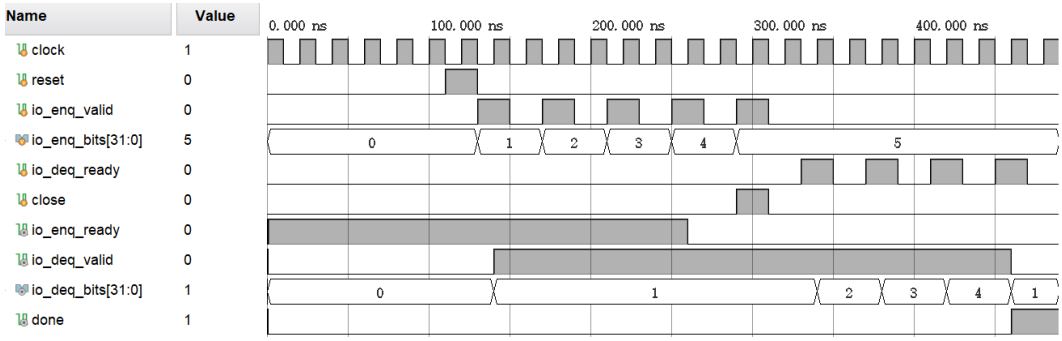


Figure 4.4: Queue Simulation Result

the risk of race conditions, as the logic within the module updates on the rising edge of the clock. The simulation proceeds as follows:

- **Reset**: Initially, a reset signal is applied to bring the queue module to its initial state, clearing any previous data or states.
- **Enqueue Operations**: Next, five valid signals are sent along with five 32-bit integers to be enqueued. Considering the queue's capacity is limited to four elements, upon enqueuing the fourth data element, the `enq_ready` signal transitions to low, preventing further data inputs.
- **Dequeue Operations**: At the start, the queue is empty, resulting in a low `deq_valid` signal until data is present in the memory. After the valid data starts populating the queue, dequeue operations start. Then in the next clock cycle the pointer will increment.
- **Close Signal**: Alongside the fifth element, a close signal is sent. Since the enqueue port is closed, the `enq_ready` signal remains low. This ensures that no more data can be enqueued, even though dequeue operations can continue until the queue is empty.
- **Done Signal**: After four successful dequeue operations, the queue is empty once again. At this point, a done signal is asserted, indicating that the queue has successfully flushed all its data, and the closing semantics are propagated.

By following this procedure, the simulation validates the functionality of the queue module, ensuring that it behaves as expected as a FIFO queue module.

4.6.2 Instantiation and Connection

In the top module shown in Listing 4.3, the main operations that need to be transformed and connected are ChannelOps, InstantiateOps, HWConnectOp, and OutputOp. These operations are vital for establishing the necessary connections between the queue module and the top module, as well as for instantiating the required components in the HW dialect.

- **Transforming ChannelOps**: Each ChannelOp is lowered into an InstanceOp, resulting in the instantiation of the queue module that was generated based on the channel. As shown in Figure 4.3, a channel is connected to either other operators or the I/O ports of the top module. If the input ports of a ChannelOp are connected directly to the top module's inputs (as indicated by HWConnectOp), these top module input ports are used in the argument list of the instantiation. If the input ports of the ChannelOp are not

connected to the top module's inputs, they are connected to the appropriate operator module within the InstantiateOp.

- **Handling Top Module Outputs**: For outputs of the top module that are connected to a channel, an OutputOp is created at the end of the scope, with the results of the output channel. Other output ports of a channel, if any, are connected to the inputs of operator modules.
- **Converting InstantiateOps**: Each InstantiateOp is also transformed into an InstanceOp. The inputs and outputs of this InstanceOp are all connected to channels, following the KPN definition.

By following these transformation and connection steps, the top module is effectively set up to interact with the queue module and other components of the system, ensuring that data flows smoothly and that all components are properly instantiated and connected.

Listing 4.7 Generated Top Module

```
hw.module @top(%clock: i1, %reset: i1, %in1_valid: i1, %in1_bits: i32,
  %in1_close: i1, %in2_valid: i1, %in2_bits: i32, %in2_close: i1,
  %out_ready: i1) -> (in1_ready: i1, in2_ready: i1,
    out_valid: i1, out_bits: i32, out_done: i1)
{
  // queue0, queue1 and queue2 instances
  %add.in1_ready, %add.in2_ready, %add.out_valid, %add.out_bits,
    %add.out_done = hw.instance "add" @add(clock: %clock: i1,
      reset: %reset: i1, in1_valid: %queue0.deq_valid: i1,
      in1_bits: %queue0.deq_bits: i32, in1_close: %queue0.done: i1,
      in2_valid: %queue1.deq_valid: i1, in2_bits: %queue1.deq_bits: i32,
      in2_close: %queue1.done: i1, out_ready: %queue2.enq_ready: i1)
        -> (in1_ready: i1, in2_ready: i1,
            out_valid: i1, out_bits: i32, out_done: i1)
  hw.output %queue0.enq_ready, %queue1.enq_ready, %queue2.deq_valid,
    %queue2.deq_bits, %queue2.done : i1, i1, i1, i32, i1
}
```

When managing the connections between various transceiver and receiver modules, the same placeholder methods described in Section 4.4.1 are applied. As shown in Figure 2.5, the receiver module generates a ready signal, which is then sent to the transceiver. This signal can subsequently be produced by the InstanceOp during the instantiation process.

To ensure accurate and reliable connections, multiple vector-like Collection objects are employed to store and manage the connection information. This approach simplifies the tracking of connections and ensures that all signals are correctly routed between different modules.

A snippet of the final top module, as listed in Listing 4.1, is shown in Listing 4.7. Within which, the instance of the add module takes signals from `queue0` and `queue1` (specifically, the valid and bits signals) as input handshake signals. Additionally, the ready signal from `queue2` is utilized as an output. The results generated by the `add` instance are then used to control the handshake interface of the queue module instances, and to propagate the closing behavior throughout the system. The outputs of this configuration consist only of the outputs from the queue instances.

4.7 Workflow

As mentioned in Section 4.1, Figure 4.1 demonstrates an overview of the FPGA backend of `DFG` dialect. The detailed workflow process is as follow:

- **First Stage of Lowering**: This initial phase takes the source code and transforms it into an intermediate form. Alongside this, it generates a code that wraps around the calculation functions of the operators present in the source code. Depending on the number of operators in the source, this process could result in multiple files.
- **Parallel Work Lines**:
  - **HLS Tool Utilization**: The wrapped calculation functions are then processed using the hlstool, which performs HLS to generate code that is optimized for hardware implementation.
  - **Second Stage of Lowering**: Simultaneously, the process undergoes a second stage of lowering, converting the entire program into a combination of CIRCT dialects. Following this, multiple built-in passes from CIRCT are utilized to generate separate SystemVerilog files.

These two sets of HDL programs collectively represent the circuit as described by the original MLIR program. The generated files can then be used in design tools from any vendor to create a RTL design. This approach ensures a seamless transition from high-level descriptions to hardware-ready code, constructing the FPGA backend for `DFG` dialect.

5 Evaluation

Upon acquiring the SystemVerilog files through the workflow introduced in Section 4.7, the next step is to encapsulate these files into a single RTL module. This encapsulation allows for a more structured, efficient approach to testing and evaluation. For the purpose of evaluation, an inversed Discrete Cosine Transformation (iDCT) kernel has been designed. The iDCT is a critical operation in signal processing and image compression, and it serves as a suitable test case to evaluate the performance and precision of the `DFG` FPGA backend. To perform the tests, a mature tool set provided by AMD Xilinx is used. These tools are specifically designed to aid in the development, simulation, and deployment of FPGA-based applications.

5.1 Experiment Setup

5.1.1 StreamIt and iDCT kernel

StreamIt, developed by the Massachusetts Institute of Technology, is a specialized programming language made to enhance productivity within the domain of stream data flow programming. The language introduces several abstraction layers and representations aimed at simplifying the complexities associated with data flow programming [TKA02]. One of the primary components of StreamIt is the filter class, which is used to describe individual nodes within a DFG. This concept has the similarity to the OperatorOp within `DFG` dialect in terms of functionality and application. Additionally, StreamIt introduces the pipeline class, which represents the connections established between different filters. This is akin to the top module of `DFG` in this scenario, where data streams flow implicitly, much like the connections created by ChannelOp in the system.

The research group behind StreamIt has also provided a variety of examples of different application areas, all programmed in StreamIt language. These examples are valuable resources for understanding the practical applications and potential of the language. Given the similarities in the conceptual models and the availability of numerous examples, it's convincing that it is feasible to reuse one of these StreamIt examples for testing `DFG` FPGA backend. This would not only validate the system but also potentially speed up the testing process, as some pre-existing and proven code bases could be utilized.

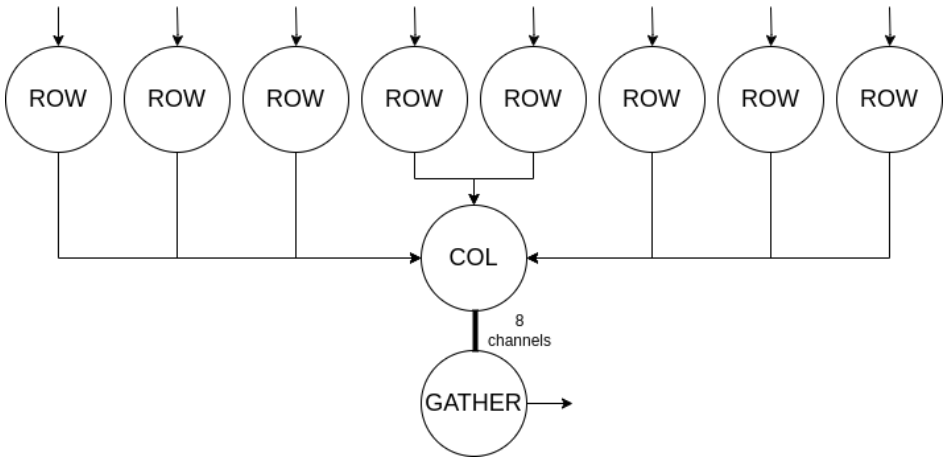


Figure 5.1: iDCT Test Kernel

Despite the constraints inherent in the work flow within an operator, following a sequence of $Pull \rightarrow Calculate \rightarrow Push$, and the needs for integer numbers in hardware design, a highly compatible algorithm is identified for test purposes. An iDCT algorithm is selected, which conforms to IEEE specifications. Figure 5.1 provides an overview of the kernel developed using `DFG` dialect. For clarity, all the channels in this figure are the arrows between nodes. The kernel is designed to process an 8×8 matrix, represented as a 1-D array, and outputs the computational results into another matrix of the same shape. The operators of the kernel is structured as follows:

- ***ROW***: takes one row from the input matrix, which is instantiated eight times in this kernel. After processing, the results are sent to the *COL* process via eight different FIFO channels.
- ***COL***: receives the results from the *ROW* processes. The output from *COL* is then distributed into eight channels, with each channel storing the results for one column of the result matrix.
- ***GATHER***: collects the results from the eight channels and assembles them back into a 1-D array, constructing the result row by row.

5.1.2 Vivado and Vitis Setup

After completing the workflow whoen in Figure 4.1, the task at hand is collecting multiple SystemVerilog programs aiming to evaluate them in terms of both precision and performance, setting the generated code from the CPU backend as the benchmark for comparison. The hardware platform selected for these tests is the ZCU104 test board, a product of AMD Xilinx, which contains various hardware resources. These resources are detailed in Figure 2.4. However, it is worth pointing out that creating the test hardware design for this specific use case isn't straightforward.

To shed light on this complicated task, Figure 5.2 provides a visual representation of the final circuit as configured in the Vivado software, showing the block design of the test kernel. In this section, there will be a step-by-step walkthrough, detailed unpacking each phase of the construction process.

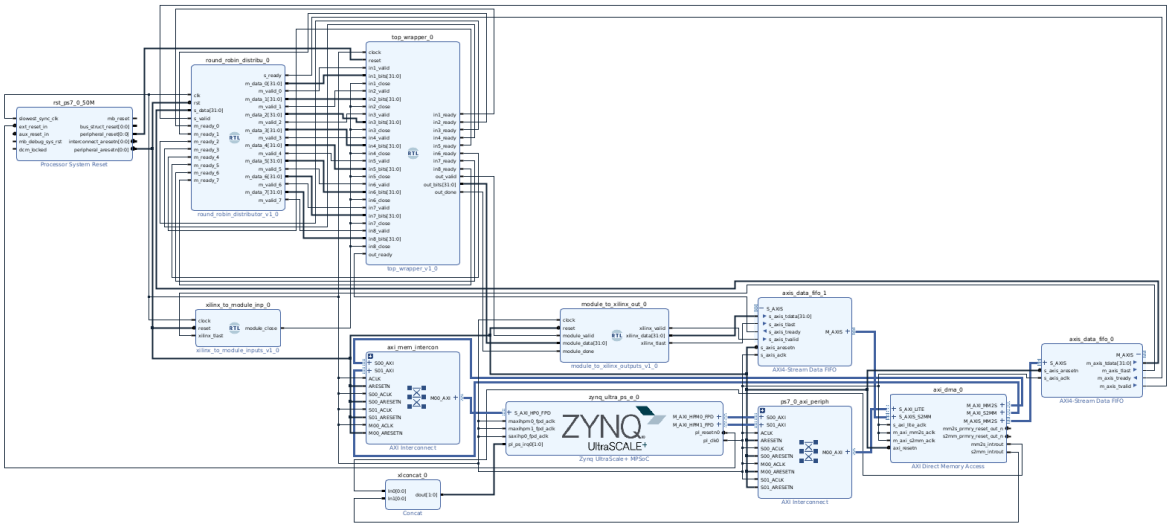


Figure 5.2: iDCT Test Circuit

In the block design, various RTL modules play critical roles, each serving a specific function in the overall system. Below is a detailed description of these modules:

- **iDCT wrapper**: serves as a wrapper, instantiating the top module generated by `DFG` backend. The creation of this wrapper is due to Vivado's lack of direct support for adding SystemVerilog modules into the block design, requiring users to re-wrap them in Verilog for integration.
- **Round-Robin distributor**: manages the distribution of incoming data into eight different channels, aligning with the *ROW* operators shown in Figure 5.1. Each channel is designed to receive eight data (which is also the capacity) from the input data stream, sourced directly from the memory on the processing system via DMA.
- **Xilinx to Module converter**: is functioning to adjust the timing of the *tlast* signal generated by Xilinx IP cores, this module delays the signal by one clock cycle. This ensures alignment with the close signal usage in the custom queue module, because the *tlast* signal is asserted with the last set of data from Xilinx IP, whereas the generated module requires the close signal asserted post the last data input.
- **Module to Xilinx converter**: operates in reverse to the *Xilinx to Module Converter*, this module ensures the synchronization of output data, valid and done signals from the queue module. It delays them by one clock cycle, ensuring that the last set of data and the done signal are asserted in the same clock cycle, ensuring the correct processing of the *tlast* signal by Xilinx IP and preventing the DMA from becoming endlessly busy.

In this scenario, the `DFG` FPGA-based setup is similar to a CUDA program, where the CPU controls data transfer, and the GPU carries out the calculations. Here, the iDCT wrapper operates on the FPGA, and the data transfer interfaces will be managed on the processing system powered by an ARM Cortex processor. To implement this, various Xilinx IP Cores are integrated into the block design:

- **ZYNQ UltraScale+ MPSoC**: acts as a software interface around the processing system, creating a logic connection between the processor and the FPGA. It allows a customization of the used resources, including Universal Asynchronous Receiver/Transmitter (UART) for debugging via a serial port and a high-performance port for DMA operations.

- **AXI Direct Memory Access**: offers high-bandwidth DMA and is suitable for AXI4-Stream type target peripherals. It supports both Stream to Memory (S2MM) and Memory to Stream (MM2S) streaming directions, and includes an optional scatter-gather mode. It allows customization of data bitwidth to align with custom design, and its initialization status and management registers are accessible through an AXI4-Lite slave interface.
- **AXI4-Stream Data FIFO**: simplifies memory-mapped access to an AXI Streaming Interface, allowing for the writing or reading of data packets to or from a device, abstracting the AXI Streaming Interface. This IP allows modifications to data bitwidth, FIFO depth, and certain I/O ports like *tlast* to ensure compatibility with the design.
- **Processor System Reset**: provides a synchronized external reset input, with the reset signal being selectable between active high or active low.
- **AXI Interconnect**: is crucial for connecting one or more AXI memory-mapped master devices to one or more slave devices. In this setup, it is used to link two DMA IPs to the processing system for both read and write directions.
- **Concat**: is added for concatenating bus signals of varying widths.

By introducing these Xilinx IP Cores, it's possible to create a smooth and efficient data flow between the ARM Cortex processor and the FPGAs, enabling the iDCTs wrapper to run effectively. After integrating all the necessary IP cores and RTL modules into the block design, the next crucial step is to manually establish connections between the custom RTL modules, as well as between the FIFO IPs. Additionally, it's required to set up the connections between the FIFOs and the DMA, linking one FIFO to the S2MM port and the other to the MM2S port port. Once these manual connections are made, Vivado's capabilities to run an automatic connection process are applied. During this phase, Vivado takes care of connecting all the reset and clock ports to the processing system. The AXI Data Stream is also connected to the ZYNQ IP through the AXI Interconnect, establishing a data pathway.

A Concat IP core is then used to concatenate the two interrupt signals from the DMA IP, transforming them into a single signal before directing it to the ZYNQ IP. This is a critical step for ensuring that interrupt signals can be properly managed and delivered in later design. The final step in this phase is to validate the design. This involves a check by Vivado to ensure that all connections are correctly established, and that there are no errors or issues in the design. Only when the design passes this validation check one can proceed to the next step. This approach to validate is crucial for ensuring the success of an FPGA-based application, as it helps to identify and resolve any potential issues early in the development process.

To generate the final bitstream, a file similar to an executable binary, it is required to sequentially execute three distinct processes in Vivado: *Synthesis*, *Implementation* and *Bitstream Generation*. The Synthesis process in Vivado adopts the Out-Of-Context (OOC) method. This approach involves synthesizing different hierarchy levels separately from the top-level design. In the context of this thesis, shown in Figure 5.2, all the components represented by blue boxes will undergo synthesis using the OOC workflow. This method significantly reduces the compilation time for subsequent synthesis runs as it only re-synthesizes the modules that are modified, leaving the rest unaffected. However, there is a crucial aspect that requires careful consideration during this process: the memory size. Prior to initiating the Synthesis process, Vivado lets the user to specify the number of cores to be used for the task. Based on this input, Vivado proceeds to run the OOC Synthesis in parallel. This parallel execution has the potential

to exhaust the available memory on the working machine, which needs cautious resource allocation to prevent system overloads.

After the successful completion of the Synthesis process, the HDL code is translated into a netlist. This netlist is a crucial representation of the design, describing the specific logic gates and their interconnections, essentially laying out the design's logical structure. The subsequent stage in the FPGA design flow is the Implementation phase. This phase is critical as it translates the logical representation provided by the netlist into a physical layout on the FPGA hardware. The *Place & Route* process, a core component of Implementation, is responsible for mapping the elements of the netlist onto the actual physical components of the FPGA, such as the logic blocks and switch box interconnections.

Resource	**Used**	**Available**	**Util%**
LUT	86439	230400	37.52
LUTRAM	850	101760	0.84
FF	211401	460800	45.88
BRAM	3	312	0.96
DSP	300	1728	17.36
BUFG	22	544	4.04

Table 5.1: Resource Utilization of iDCT Kernel

Upon the completion of the Implementation phase, Vivado provides an exhaustive report on the resource utilization of the FPGA for the given design. This report is displayed in a tabular format, referred to as Table 5.1. A noteworthy point in Table 5.1 is the considerable utilization of FFs and Digital Signal Processors (DSPs) in this iDCT design. This high utilization percentage is due to the calculation programs generated by the hlstool, which may contain too many operations in the same clock cycle. Additionally, a significant percentage of LUTs are utilized, a consequence of the data and valid updating checks taking place within the FSM, as detailed in 4.5.2.

It is important to highlight that the success of the Implementation phase heavily depends on the availability of sufficient resources on the FPGA board. If the board lacks the necessary resources to fit the design, the Implementation phase will not succeed, subsequently failing in the generation of the bitstream.

Upon the successful generation of a design encapsulated in a bitstream file, it's possible to export the design into an Xilinx Support Archive (XSA) file. This XSA file is essential because it contains all the necessary information for building a platform customized for a specific target device. Subsequent to this exportation, the workflow transitions to the Vitis software, which can be launched directly from Vivado. Within the Vitis environment, it's capable to create an application specifically for the processing system of the ZCU104 board, with the previously generated XSA file as base.

Figure 5.3 shows the workflow and interactions within the created application. The XSA package contains various header files, each serving the key role of defining macros, among which are those denoting memory addresses presently occupied by the system, and useful functions to utilize in the application design. In scenarios where data is being actively written to and read from the memory on the processing system, it becomes important to cautiously

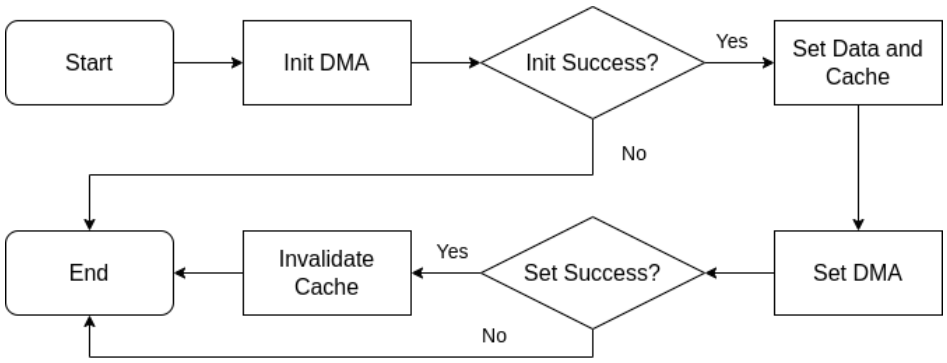


Figure 5.3: Vitis Program Control Flow

select memory addresses that are currently unoccupied by the system to avoid any potential conflicts or errors. To initialize the DMA module needs a sequential approach, as listed in the following order:

- **Look up configuration**: This initial step to use the DMA device ID, as specified in the header files, is a lookup operation. The result of this operation is a pointer to the DMA configuration, which is essential for the subsequent steps in the initialization process.
- **Initialize with configuration**: Armed with the configuration pointer acquired in the previous step, this phase involves initializing the DMA core. This core is a crucial component, as it is required for any later interactions with the DMA.
- **Disable interrupts**: In this final step of the initialization process, interrupts associated with the DMA are disabled. This is a important, as implementation of this thesis utilizes the polling mode of the DMA, as opposed to the scatter-gather mode. By disabling interrupts, a smoother operation in the polling mode is ensured, reducing the risk of interruptions that could potentially lead to errors or inefficiencies in data transfer.

In the event that the initialization process is unsuccessful, the program is designed to terminate immediately to prevent any further issues. However, if the initialization is successful, the program proceeds to write data into the memory at the predefined addresses, and it needs the caches to be flushed to eliminate any potential errors that may arise. Subsequent to these initial steps, it's proceeded to configure the DMA using the `XAxi_Dma_SimpleTransfer` function, which applies data transfer in both directions. Once the DMA settings are in place, the program enters a waiting state, monitoring the DMA device for any signs of busyness. A non-busy state of the DMA device indicates the completion of data transfers, signaling that the result data is now stored in the memory.

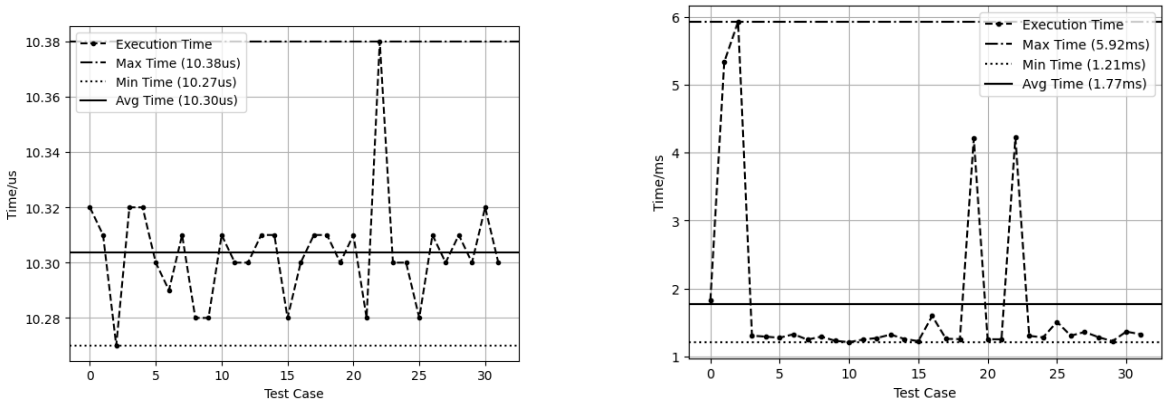
At this point, it's important to invalidate the data to make sure that the cache is not accidentally enabled, which could potentially compromise the integrity of the results. To gauge the execution time of the kernel under test, a rough estimation technique is employed, measuring the time elapsed from the moment the DMA begins the data transfer to the instant it concludes. Utilizing the `XTime_GetTime` function both before and after setting the DMA, it's convenient to capture the clock cycle counts, subsequently calculating their difference to obtain an integer value. This value, when divided by the predefined macro `COUNTS_PER_SECOND`, yields the execution time in seconds.

For the purposes of debugging and performance evaluation, the `xil_printf` function is utilized to print program status and kernel execution time to the serial debug port. It is worth noting

that `xil_printf` is a lightweight version of the standard `printf` function and does not support floating-point numbers, which is not used in this case. All the debug information printed through this function is logged for evaluation in the subsequent section of analysis. Additionally, the Vitis software provides functionality to export data stored in the memory to a binary file, a feature used to output the results generated by the iDCT kernel.

5.2 Results

The test program is executed, as discussed in the preceding section, a total of thirty-two times, recording the time counts and the divider through the serial debug port. Additionally, the resultant data are exported into binary files utilizing the Vitis software. Remarkably, the data remained consistent across all test iterations, thereby following the deterministic nature of the kernel designed using `DFG` dialect, which is also a essential characteristic of a KPN.



(a) FPGA Backend **(b)** CPU Backend

Figure 5.4: Execution Time on Hardware

Upon conducting an analysis of the results yielded by programs generated using a base C-program and the FPGA backend of `DFG` dialect, yet subjected to the same test data, different sets of completely same results are observed. This finding indicates that the integrity and correctness of the program is successfully upheld throughout the lowering transformation processes.

The ARM processor, by default, operates with a clock speed of 100MHz, resulting in a divider value of one hundred million. Through subsequent calculations, it's possible to convert the time measurements into seconds, and by multiplying by one million, the time in microseconds is obtained, as shown in Figure 5.4a. Across the thirty-two identical tests conducted independently, the kernel demonstrated an exceptional performance, clocking in at 10.30 microseconds on the FPGA. Notably, the difference between the maximum and minimum values was a mere 0.11 microseconds, which is a minor deviation in practical applications. The data shown in the figure makes clear that the execution time for the majority of the tests is around the mean value, showcasing the robust stability of the iDCT kernel.

Conversely, a evaluation of the performance of the code generated by the CPU backend is performed on a machine, which CPU is an AMD Ryzen 9 with 12 physical cores and 4.5 GHz frequency. Employing the same dataset for testing, the results are different from base program

and FPGA backend. Besides, the performance were less than optimal. As shown in Figure 5.4b, the execution time of this code is quantified at the millisecond level, showcasing a marked decrease in efficiency when compared to the FPGA backend. It is clearly observable from the figure that the execution times are distributed between a maximum of approximately 6ms and a minimum of 1ms. Considering the current erroneous results, there appear to be underlying issues in the implementation of the provided CPU backend.

Despite these, in the FPGA design, using a *Round Robin Distributor* introduces a serial part to the system. This can slow things down, especially if people are dealing with a lot of data or tasks. To try and speed things up, one might think about adding more DMA IPs, which help move data between the processing system and iDCT kernel faster. If it's possible to add as many as there needs, it could actually make the gap between the CPU and FPGA performance even bigger. However, it should be conservative considered, since this test has already utilized a lot resources on the board. As a result, comparing FPGA backend and CPU backend in this particular scenario, regardless of result consistency, an average speed up of approximately 170x is achieved, even when the frequency of FPGA is much lower than CPU.

This expected significant difference in performance can be because of the architectural differences between CPU and FPGA. The CPU, being a general-purpose processor, is designed to handle a broad variety of tasks, but it may not be as proficient in executing specific, computation-intensive operations as an FPGA. FPGAs, with their concurrent nature and the reconfigurable hardware, allow for the tailoring of their architecture to optimize the execution of particular tasks, resulting in much better performance for such applications.

6 Conclusion and Further Work

6.1 Conclusion

In this thesis, we have conducted an extensive exploration of the data flow programming model and FPGA hardware design. With MLIR, we have investigated various open-source projects and delved into different design methodologies. The main goal of this research is the development of an MLIR dialect and its associated FPGA backend, interacting with the core dialect defined in CIRCT project. This innovative backend is engineered to translate an abstract representation of a KPN into low-level HDL programs.

To evaluate the effectiveness of this backend, we designed a custom iDCT kernel, combinedly used with a set of RTL modules and various Xilinx IP cores, implemented on a ZCU104 SoC board. We conducted a comparison between the FPGA backend and the CPU backend, using the latter as the benchmark. This analysis focuses on both the computational precision and performance.

The obtained results clearly demonstrated that, when subjected to identical test kernels and input data, the FPGA outperformed the CPU in terms of computational efficiency, while upholding the integrity of the results at the same time. This great performance of the FPGA can be attributed to its hardware characteristics, which are optimally suited for such concurrent computational tasks using less power consumption.

In light of these findings, this thesis finds out the potential and viability of the `DFG` dialect in abstracting KPNs, as well as its capability to generate target code that is both efficient and reliable. This work not only showcases the advantages of FPGA in specific computational scenarios but also lays a solid foundation for future research and development in this domain, setting the base of more innovative and optimized hardware-software co-design strategies.

Despite the promising potential demonstrated by our `DFG` dialect in simplifying the construction of KPN models and its flexibility in deployment across different hardware platforms, there are several aspects for further exploration and development that needs optimization as well.

6.2 Future Work

In the pursuit of enhancing the flexibility and performance of the `DFG` dialect, we have identified several potential areas of future work. Each of these aims to extend the capabilities of the `DFG` dialect, ensuring its applicability in a broader range of scenarios and improving its integration with existing tools and hardware components.

- **Pull → Calculation → Push**: The current semantics of our dialect limit the scenarios that can be described, particularly when it comes to pulling data from channels within a specific region of an upstream operation, such as an `scf.for` loop. To overcome this limitation, it is crucial to introduce intrinsic operations into our intermediate operation set. This will simplify the analysis and transformation of control flow, similar to what is achieved through handshake dialect.
- **Compatibility with Xilinx IP**: To better integrate with Xilinx IP, we need to refine the lowering transformation process from our operations to the hardware module. This includes addressing issues such as the activation of the falling edge of the reset, modifying close and done signals of lowered modules to adjust the *tlast* signal usage, and possibly generating scripts to automate the production of circuits in Vivado and control code of processing system in Vitis.
- **Parameterizable Queue Module**: By making our queue module more flexible and parameterizable, we can enable its instantiation at the top module level with various parameters. Achieving this will require utilizing the `!hw.int<size>` type and parameter semantics of HW dialect.
- **Integration with `BASE2` dialect**: The `BASE2` dialect [FBC23] aims to provide abstraction of arbitrary precision arithmetic. When paired with our `DFG` dialect, which can be used to generate custom computation kernels, the potential for `BASE2` is greatly amplified. The integration of `DFG` dialect could lead to a robust framework capable of handling a wide variety of computational tasks. However, given that MLIR is rapidly evolving, careful attention must be paid to version control to ensure seamless integration with our CIRCT-based dialect.
- **Support for Single-Rate Operator**: Introducing support for single-rate operators could enhance flexibility and potentially improve performance. While our current design ensures integrity and correctness for multi-rate operators, single-rate operators could benefit from unordered pull or push operations, triggered as soon as the data is ready. This approach needs additional logic in the FSM and new types or attributes to the `DFG` dialect specification.

By addressing these areas, we are confident that the `DFG` dialect can go beyond its current status as a research project, potentially becoming an integral part of the CIRCT project or any others that could take advantage of our design. The journey towards these improvements promises to be both challenging and rewarding, with the potential to significantly impact the field of data flow programming and FPGA hardware design.

Acknowledgement

This work would never be done without the generous supports that I've received since the day I started working at the CCC chair. First and foremost, I'd like to thank my two supervisors, Felix Suchert and Karl Friebel, for supporting me during the research period and patiently helping me while I encountered some serious problems.

Great thanks to Professor Jeronimo Castrillon. If you didn't provide such great courses in the area of Compiler Construction and so many interesting cutting-edge research projects, I wouldn't have the chance to find out more possibilities at the very first place.

Thank you, Prof. Lana Josipović from ETH Zürich for sharing your time discussing with us about the handshake dialect and related works. And thanks to Mike Urbach and Rachit Nigam from CIRCT community for all the generous helps in the forum.

My former supervisors from ADS chair, Diana Göhringer, Lester Kalms, Matthias Nickel, Najdet Charaf, Sergio A. Pertuz, who all helped so much during my study in the Computer Science faculty. It's my honor to have the chance to take courses from you and work with you.

My current and former office-mates, Caio Vieira, Eren Yenigül, Paul Justen and Shaokai Lin, thank you for chatting with me and exchanging innovative thoughts over everything.

My Chinese fellow students, who I used to live together with. Thank you for all the supports you gave me since four years ago when I first came to Germany and for overcoming the Corona pandemic with me together even when I was badly infected.

Thanks to all the researchers in the CCC chair for showing me that research could also be fun instead of the boring stereotype and providing me an office that I could work in.

And finally a special thanks to WATZKE for providing various delicious beer. Cheers!

Bibliography

[Gil74] KAHN Gilles. “The semantics of a simple language for parallel programming”. In: *Information processing* 74.471-475 (1974), pp. 15–28.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. “StreamIt: A language for streaming applications”. In: *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings 11*. Springer. 2002, pp. 179–196.

[Bha+08] Shuvra Bhattacharyya et al. “OpenDF – A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems”. In: *Proceedings of the Swedish Workshop on Multicore Computing* 36 (Nov. 2008).

[Vrb+09] Željko Vrba et al. “Kahn Process Networks are a Flexible Alternative to MapReduce”. In: *2009 11th IEEE International Conference on High Performance Computing and Communications*. 2009, pp. 154–162. DOI: 10.1109/HPCC.2009.46.

[ARM10] ARM. *AMBA 4 AXI4-Stream Protocol Specification*. en. 2010. URL: https://documentation-service.arm.com/static/642583d7314e245d086bc8c9?token=.

[Bac+12] Jonathan Bachrach et al. “Chisel: Constructing Hardware in a Scala Embedded Language”. In: *Proceedings of the 49th Annual Design Automation Conference*. DAC ’12. San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225. ISBN: 9781450311991. DOI: 10.1145/2228360.2228584. URL: https://doi.org/10.1145/2228360.2228584.

[Sou12] Tiago Sousa. *Dataflow Programming: Concept, Languages and Applications*. Jan. 2012.

[SB14] J. Sérot and F. Berry. “High-Level Dataflow Programming for Reconfigurable Computing”. In: *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*. Oct. 2014, pp. 72–77. DOI: 10.1109/SBAC-PADW.2014.18. URL: https://ieeexplore.ieee.org/abstract/document/6972018 (visited on 10/22/2023).

[LIB16] Patrick S Li, Adam M Izraelevitz, and Jonathan Bachrach. “Specification for the FIRRTL Language”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9* (2016).

[Koe+18] David Koeplinger et al. "Spatial: A Language and Compiler for Application Accelerators". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 296–311. ISBN: 9781450356985. DOI: 10.1145/3192366.3192379. URL: https://doi.org/10.1145/3192366.3192379.

[C P+19] Jeronimo C. Penha et al. "ADD: Accelerator Design and Deploy - A tool for FPGA high-performance dataflow computing". en. In: *Concurrency and Computation: Practice and Experience* 31.18 (2019), e5096. ISSN: 1532-0634. DOI: 10.1002/cpe.5096. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5096 (visited on 10/22/2023).

[GK19] Shubham Gandhare and B. Karthikeyan. "Survey on FPGA Architecture and Recent Applications". In: *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*. 2019, pp. 1–4. DOI: 10.1109/ViTECoN.2019.8899550.

[Qas+19] Murad Qasaimeh et al. "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels". In: *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. June 2019, pp. 1–8. DOI: 10.1109/ICESS.2019.8782524. URL: https://ieeexplore.ieee.org/abstract/document/8782524?casa_token=-1U1Ftl8Fx4AAAAA:seOXtEyDeRu2NuYxHBNcBpcv6K-5J5qDWZLKOWFNp0lzj482RtsMMqhdQTi0Aabzm-VHXw_2dCc (visited on 10/22/2023).

[NJ20] Stephen Neuendorffer and Lana Josipović. *handshake_dialect*. 2020. URL: https://drive.google.com/file/d/1UYQAfHrzcsdXUZ93bHPTPNwrscwx89M-/view.

[BP21] Praveenkumar Babu and Eswaran Parthasarathy. "Reconfigurable FPGA Architectures: A Survey and Applications". en. In: *Journal of The Institution of Engineers (India): Series B* 102.1 (Feb. 2021), pp. 143–156. ISSN: 2250-2106, 2250-2114. DOI: 10.1007/s40031-020-00508-y. URL: http://link.springer.com/10.1007/s40031-020-00508-y (visited on 10/22/2023).

[Lat+21] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308. URL: https://ieeexplore.ieee.org/abstract/document/9370308 (visited on 10/22/2023).

[Mar21] Peter Marwedel. "Specifications and Modeling". en. In: *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Ed. by Peter Marwedel. Embedded Systems. Cham: Springer International Publishing, 2021, pp. 29–126. ISBN: 978-3-030-60910-8. DOI: 10.1007/978-3-030-60910-8_2. URL: https://doi.org/10.1007/978-3-030-60910-8_2 (visited on 10/25/2023).

[Mos+21] William S. Moses et al. "Polygeist: Raising C to Polyhedral MLIR". In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2021, pp. 45–59. DOI: 10.1109/PACT52795.2021.00011.

[Pil+21] Christian Pilato et al. "EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms". In: *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2021, pp. 1320–1325. DOI: 10.23919/DATE51398.2021.9473940.

[Cia22] Pedro Ferrazoli Ciambra. "IaRa: Dataflow Programming with the MLIR Framework". en. MA thesis. Universidade Estadual de Campinas, 2022. URL: https://repositorio.unicamp.br/Busca/Download?codigoArquivo=552777 (visited on 10/26/2023).

[Ulm22] Christian Ulmann. "Multi-Level Rewriting for Stream Processing to RTL compilation". en. Accepted: 2022-11-01T14:54:12Z. MA thesis. ETH Zurich, 2022. DOI: 10.3929/ethz-b-000578713. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/578713 (visited on 10/26/2023).

[DLa23] D.Lazard. *Model of computation*. Aug. 2023. URL: https://en.wikipedia.org/wiki/Model_of_computation.

[FBC23] Karl F. A. Friebel, Jiahong Bi, and Jeronimo Castrillon. "Base2: An IR for Binary Numeral Types". In: *Proceedings of the 13th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*. HEART '23. Kusatsu, Japan: Association for Computing Machinery, 2023, pp. 19–26. DOI: 10.1145/3597031.3597048. URL: https://doi.org/10.1145/3597031.3597048.

[gro23] CIRCT group. *"CIRCT" / Circuit IR Compilers and Tools*. July 2023. URL: https://circt.llvm.org/.

[SP23] Stephanie Soldavini and Christian Pilato. *Platform-Aware FPGA System Architecture Generation based on MLIR*. 2023. arXiv: 2309.12917 [cs.AR].

[Sol+23] Stephanie Soldavini et al. "Automatic Creation of High-bandwidth Memory Architectures from Domain-specific Languages: The Case of Computational Fluid Dynamics". In: *ACM Trans. Reconfigurable Technol. Syst.* 16.2 (Mar. 2023). ISSN: 1936-7406. DOI: 10.1145/3563553. URL: https://doi.org/10.1145/3563553.

[Suc+23] Felix Suchert et al. "ConDRust: Scalable Deterministic Concurrency from Verifiable Rust Programs". In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 33:1–33:39. ISBN: 978-3-95977-281-5. DOI: 10.4230/LIPIcs.ECOOP.2023.33. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.33.

[Xil23] Xilinx. *Zynq UltraScale+ MPSoC*. 2023. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

List of Listings

List of Figures

List of Tables