Embedded Systems & Applications

TECHNISCHE UNIVERSITÄT DARMSTADT

Design of a modern hardware synthesis tool for Sum-Product Networks

Masterthesis by
Jörn Schöndube

January 2024

Examiner:
Prof. Dr. Andreas Koch

Supervisor:
Dr.-Ing. Julian Oppermann, Lukas Weber

Technische Universität Darmstadt
Department of Computer Science
Embedded Systems and Applications Group (ESA)

Design of a modern hardware synthesis tool for Sum-Product Networks
Entwurf eines modernen Hardware-Synthese-Werkzeugs für sog. Sum-Product-Networks
Masterthesis by Jörn Schöndube
Submitted on 08.01.2024
Examiner: Prof. Dr. Andreas Koch
Supervisor: Dr.-Ing. Julian Oppermann, Lukas Weber

Thesis Statement

Acknowledgements

I want to thank Julian Oppermann for providing me with the opportunity to work on this thesis on such an interesting topic. I want to thank him and Lukas Weber for supervising my work and being constantly available for questions and discussions. I also want to thank the whole ESA team for providing infrastructure for development and benchmarking and technical support.

Abstract

Sum-Product Networks (SPNs) are a type of probabilistic model that has emerged relatively recently. Their ability to compute different types of probabilistic inference efficiently makes them interesting for a wide variety of applications. Because of their simple architecture, they lend themselves well to be mapped to high-throughput hardware implementations as they are represented as a directed acyclic graph of arithmetic operations. This is explored in XSPN-FPGA Project (XSPN-FPGA). It is a Chisel-based compiler for mapping a text-based SPN definition to an Field-Programmable Gate Array (FPGA) bitstream.

The new emerging compiler framework Multi-Level Intermediate Representation (MLIR) shows promising results in optimizing programs at different abstraction levels. Their core approach is to gradually convert domain-specific Intermediate Representations (IRs) into other domain-specific IR such that certain optimizations are more easily expressible. Circuit IR Compilers and Tools (CIRCT) is an extension of MLIR that focuses on implementing tooling around hardware-specific IRs.

The hope is that porting the XSPN-FPGA project to MLIR+CIRCT will make future development benefit from the advantages of the MLIR+CIRCT ecosystem. Specifically, compiler tools, domain-specific optimizations and an extensive framework and ecosystem are key arguments for such an undertaking.

This work implements an end-to-end toolflow for mapping SPNs to hardware. It extends Sum-Product Network Compiler (SPNC) by offering another target architecture besides Central Processing Units (CPUs) and Graphics Processing Units (GPUs), namely FPGAs. To make the task tractable we implement a C++-based Domain Specific Language (DSL) that acts as a frontend to CIRCT's Flexible Intermediate Representation for RTL (FIRRTL) dialect. Furthermore, it briefly explores extending the Elastic Silicon Interconnect (ESI) framework with support for the AXI4-Stream protocol specifically for our use case.

This work demonstrates that MLIR+CIRCT greatly eases the design and implementation of domain-specific compile toolflows but also identifies a great set of missing functionality.

Kurzfassung

SPNs sind ein relativ neuer Typ von probabilistischen Modellen. Ihre Fähigkeit verschiedene Arten von probabilistischen Inferenzen effizient zu berechnen macht sie für eine Vielzahl von Anwendungen interessant. Aufgrund ihrer einfachen Architektur eignen sie sich gut für die Abbildung auf Hardware, da sie als gerichteter azyklischer Graph von arithmetischen Operationen dargestellt werden. Dies wird in dem XSPN-FPGA untersucht. Es ist ein Chisel-basierter Compiler, der eine textbasierte SPN-Definition auf einen FPGA-Bitstream abbildet.

Das neue aufkommende Compiler-Framework MLIR zeigt vielversprechende Ergebnisse bei der Optimierung von Programmen auf verschiedenen Abstraktionsebenen. Ihr Kernansatz besteht darin,
domänenspezifische IRs schrittweise in andere
domänenspezifische IRs umzuwandeln, so dass bestimmte Optimierungen leichter ausdrückbar sind. CIRCT ist eine Erweiterung von MLIR, die sich auf die Implementierung von Werkzeugen rund um hardware-spezifische IRs konzentriert.

Die Hoffnung ist, dass die Portierung des XSPN-FPGA-Projekts auf MLIR+CIRCT dazu führt, dass zukünftige Entwicklungen von den Vorteilen des MLIR+CIRCT-Ökosystems profitieren. Insbesondere Compiler-Tools, domänenspezifische Optimierungen und ein umfangreiches Framework und Ökosystem sind Schlüsselargumente für ein solches Unterfangen.

Diese Arbeit implementiert einen End-to-End-Toolflow für die Abbildung von SPNs auf Hardware. Sie erweitert SPNC um eine weitere Zielarchitektur neben GPUs und GPUs, nämlich FPGAs. Um die Aufgabe beherrschbar zu machen, implementieren wir eine C++-basierte DSL, die als Frontend für den CIRCT-FIRRTL-Dialekt fungiert. Darüber hinaus wird kurz untersucht, wie das ESI-Framework um die Unterstützung des AXI4-Stream-Protokolls speziell für unseren Anwendungsfall erweitert werden kann.

Diese Arbeit zeigt auf, dass MLIR+CIRCT die Gestaltung und Implementierung von domänenspezifischen Kompilierungstoolflows erheblich erleichtert, aber auch eine große Menge an Funktionalität noch fehlt.

Contents

1 Introduction **3**

2 Technical Background **5**
- 2.1 Sum-Product Networks 5
- 2.2 LLVM 6
- 2.3 MLIR 8
  - 2.3.1 IR, Dialects and Passes 8
- 2.4 CIRCT 11
  - 2.4.1 FIRRTL 11

3 Related Work **13**
- 3.1 TaPaSCo 13
- 3.2 XSPN-FPGA 13
  - 3.2.1 Technical Details 14
  - 3.2.2 The UFloat Number Format 15
- 3.3 Applications of CIRCT 15

4 Base Framework **17**
- 4.1 Architecture of The Sum-Product Network Compiler 17
  - 4.1.1 IRs and Dialects 17
  - 4.1.2 Compilation Flow 18
  - 4.1.3 Frontend and Runtime 20
- 4.2 SPFlow 21
- 4.3 CocoTb 21
- 4.4 ESI 21

5 Minimum Viable Product **23**
- 5.1 Approach 23
- 5.2 Implementation 24
  - 5.2.1 LoSPN To FPGA Conversion 25
  - 5.2.2 Embed into Controller 26
  - 5.2.3 Creating a Vivado Project 27

5.2.4 Runtime and Frontend 28
5.2.5 Testing and Simulation 29
5.3 Discussion 29

6 Refined Approach 31
6.1 Approach 33
6.2 Implementation 33
6.2.1 FIRRTL++ 33
6.2.2 ESI 43

7 Evaluation 47
7.1 Comparison to XSPN-FPGA 47
7.1.1 Disclaimer and Environment 47
7.1.2 Resource Utilization and Performance 48
7.1.3 Compilation and Synthesis Performance 52

8 Conclusion and Future Outlook 55
8.1 Conclusion 55
8.2 Future Work 56
8.2.1 Marginal Inference and MPE 56
8.2.2 Deviating from the simple Bottom-up Architecture 57
8.2.3 Using Calyx 58
8.2.4 Automatic Scheduling 58
8.2.5 Work on ESI 59

List of Figures I

List of Tables II

List of Acronyms III

Bibliography V

Abstract

SPNs are a type of probabilistic model that has emerged relatively recently. Their ability to compute different types of probabilistic inference efficiently makes them interesting for a wide variety of applications. Because of their simple architecture, they lend themselves well to be mapped to high-throughput hardware implementations as they are represented as a directed acyclic graph of arithmetic operations. This is explored in XSPN-FPGA. It is a Chisel-based compiler for mapping a text-based SPN definition to an FPGA bitstream.

The new emerging compiler framework MLIR shows promising results in optimizing programs at different abstraction levels. Their core approach is to gradually convert domain-specific IRs into other domain-specific IR such that certain optimizations are more easily expressible. CIRCT is an extension of MLIR that focuses on implementing tooling around hardware-specific IRs.

The hope is that porting the XSPN-FPGA project to MLIR+CIRCT will make future development benefit from the advantages of the MLIR+CIRCT ecosystem. Specifically, compiler tools, domain-specific optimizations and an extensive framework and ecosystem are key arguments for such an undertaking.

This work implements an end-to-end toolflow for mapping SPNs to hardware. It extends SPNC by offering another target architecture besides CPUs and GPUs, namely FPGAs. To make the task tractable we implement a C++-based DSL that acts as a frontend to CIRCT's FIRRTL dialect. Furthermore, it briefly explores extending the ESI framework with support for the AXI4-Stream protocol specifically for our use case.

This work demonstrates that MLIR+CIRCT greatly eases the design and implementation of domain-specific compile toolflows but also identifies a great set of missing functionality.

1 Introduction

Sum-Product Networks (SPNs) are a type of probabilistic model. They are represented as a Directed Acyclic Graph (DAG) with sum, product and histogram leaf nodes. They are a deep architecture and have two key advantages over other similar models: (1) In contrast to Neural Networks (NNs) they retain interpretability because of their probabilistic nature and (2) they offer tractable marginal, conditional and Maximum Probability Explanation (MPE) inference in contrast to other Probabilistic Graphical Models (PGMs) [17]. The DAG structure lends them well for mapping to hardware. This was explored by Sommer et al. in [21]. It is a Chisel-based compiler for mapping text-based SPN definitions to FPGA bitstreams. Quite some publications [24], [23], [9] have expanded on their initial work which is now known internally as XSPN-FPGA. As XSPN-FPGA is a Chisel-based project, it cannot easily benefit from established and emerging compiler frameworks which are usually implemented in C++. MLIR [11] is a compiler framework that has shown promising results in optimizing programs at different abstraction levels. MLIR implements IRs at different domain-specific abstraction levels such that optimizations can be expressed much more naturally. It has recently gained adoption in different industries. CIRCT is an extension of MLIR for hardware-specific IRs and tooling. In contrast to MLIR, CIRCT was not part of many publications exploring its capabilities in real use cases.

In this context, we want to evaluate how a compiler framework, namely CIRCT benefits the development of FPGA designs, if at all. This requires turning mapping SPNs-to-FPGAs into a compiler problem, whereas in XSPN-FPGA it is mostly considered a hardware architecture problem.

We want to answer the question if MLIR+CIRCT is a suitable framework for implementing a domain-specific compiler toolflow for mapping SPNs to FPGA bitstreams. Additionally, we want to evaluate what could be gained by using MLIR+CIRCT as a compiler framework. To answer this we build an end-to-end toolflow for FPGAs by extending SPNC [20].

This work is very explorative in nature. We first give a technical background about SPNs, Low Level Virtual Machine (LLVM), MLIR and CIRCT in chapter 2. Especially the latter two are important for understanding a lot of the implementation details and design decisions. Then we take a look at related work and take a deep dive into the technical details of XSPN-FPGA in chapter 3. This is important as this work is essentially a port of XSPN-FPGA into the CIRCT ecosystem. As we extend SPNC, we also discuss its technical details in **??**. The approach and implementation are split up into two iterations. To better manage the task of building an end-to-end toolflow to FPGA targets we set a Minimum Viable Product (MVP). It makes compromises on some key aspects of the goal but helps us guide in understanding and overcoming the encountered problems. We briefly discuss the problems and discuss possible solutions in chapter 5. We present our solution and discuss its implementation in great detail in chapter 6. Finally, we make a quantitative comparison of the quality of the generated design by benchmarking our implementation versus XSPN-FPGA before giving a conclusion and outlook for future work in chapter 7 and chapter 8.

2 Technical Background

2.1 Sum-Product Networks

A SPN is a type of probabilistic model that can be used for classification and regression tasks. They are modeled using a DAG which encodes the probability distribution $P(X_1, ..., X_n)$ over the set of univariate random variables $\mathbf{X} = \{X_1, ..., X_n\}$. For convenience we abbreviate $P(x_1, ..., x_n)$ with $P(\mathbf{x})$ for $\mathbf{x} = \{x_1, ..., x_n\}$. SPNs are closely related to Bayesian Networks (BNs) and Markov Random Fields (MRFs) in the sense that a SPN representation can be found for both of these probabilistic models. Computing the full joint probability distributions for SPNs is simple. Given a full assignment of all random variables, $\mathbf{x}$ the joint probability distribution of any node v is computed as follows:

- If v is a product node, then the joint probability of v is the product of the probabilities of all of its child nodes. Nameley, $P_v(\mathbf{x}) = \prod_{c \in Ch(v)} P_c(\mathbf{x})$.
- If v is a weighted sum node, then the joint probability of v is the weighted sum of the probabilities of all of its child nodes. Nameley, $P_v(\mathbf{x}) = \sum_{c \in Ch(v)} w_c P_c(\mathbf{x})$.
- If v is a leaf node then the joint probability of v is the probability of the random variable X_i that v represents. Nameley, $P_v(\mathbf{x}) = P_v(X_i = x_i)$.

SPNs are an interesting class of models because computing marginal distributions and MPE queries are linear in the number of nodes. This is a clear advantage over BNs and MRFs where these inference types are NP-hard [17]. Figure 2.1 shows examples of a typical BN and SPN and their respective joint probability distributions.

Marginal distribution queries are of the form

$$P(\hat{\mathbf{x}}) = \sum_{x_D \in D(X - \hat{X})} P(\hat{x}, x_D)$$

for some $\hat{X} \subseteq X$ where $D(X - \hat{X})$ is the domain of all the remaining variables. Generally, computing this is already NP-hard in BNs. Exact algorithms like variable

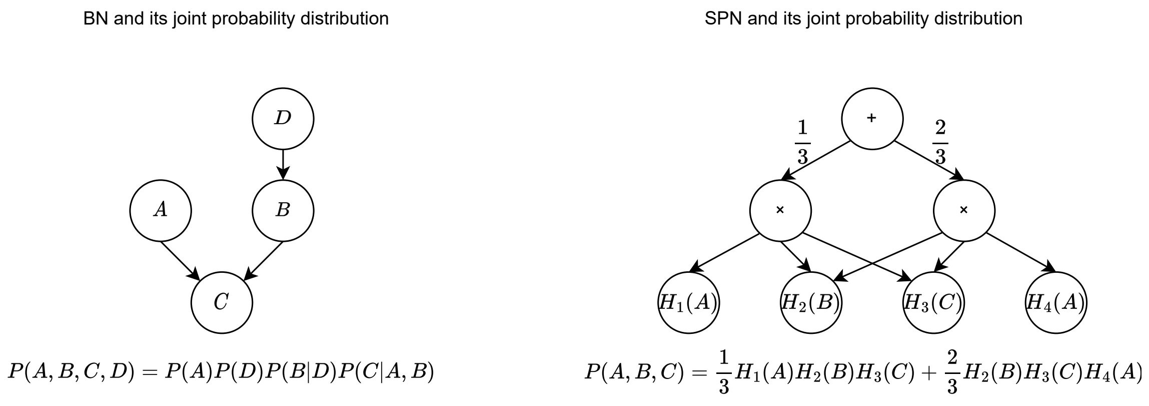
BN and its joint probability distribution
SPN and its joint probability distribution
D
A
B
C
$P(A,B,C,D) = P(A)P(D)P(B|D)P(C|A,B)$
+
$\frac{1}{3}$
$\frac{2}{3}$
×
×
$H_1(A)$
$H_2(B)$
$H_3(C)$
$H_4(A)$
$P(A,B,C) = \frac{1}{3}H_1(A)H_2(B)H_3(C) + \frac{2}{3}H_2(B)H_3(C)H_4(A)$

Figure 2.1: A BN, a SPN and their joint probability distributions.

elimination and the junction tree algorithm [7] perform much better in practice. However, SPNs only require a single bottom-up pass through the DAG where the leaf nodes which are associated $\hat{X}$ are set to 1. One has to be careful when comparing the exponential complexity in BNs and the linear complexity in SPNs because a SPN which is equivalent to a BN can be exponential in size. Nonetheless, the simplicity of the single bottom-up pass makes them very appealing.

MPE queries are of the form

$$MPE(\hat{\mathbf{x}}) = \underset{x_D \in D(X-\hat{X})}{\arg\max} P(\hat{x}, x_D)$$

with the same definitions of x_D and $D(X - \hat{X})$ as previously. MPE computes the assignment of the remaining variables that maximize the whole probability distribution. SPNs require two bottom up passes through the DAG to compute this. An interesting use-case for MPE is finding out what the state of maximum probability is given some observations because of the following relation:

$$\arg\max_A P(A|b,c) = \arg\max_A \frac{P(A,b,c)}{P(b,c)} = \arg\max_A P(A,b,c)$$

Since we are interested in building a new compiler toolflow for SPNs, we are only concerned with computing the full joint distributions. From hereon, we just view SPNs as Arithmetic Circuits (ACs).

2.2 LLVM

LLVM is a compiler framework. It provides high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs to allow for transparent program analysis and transformation of arbitrary programs [10].

The core of LLVM is an IR that captures the essence of most of today's CPU (and to some extent GPU) instruction sets. It does not model architecture-specific properties such as register count, pipelines, and low-level calling conventions. LLVM uses Static Single Assignment (SSA) form to represent computations on operands and writing the results to registers. For example, the expression `%2 = add i32 %0, %1` adds the values from registers `%0` and `%1` and writes the result to register `%2`. The registers are purely virtual and there are infinitely many of them. Each register has a concrete type attached to it, depending on the operation which assigns to it. There also exist primitive types for booleans, integers, floating point numbers, and pointers. This is in contrast to conventional CPUs where register values are untyped. In its initial release LLVM only contained 31 instructions to describe load, stores, some arithmetic instructions, and control flow.

Now, developing a programming language on top of LLVM "just" requires a parser and mapping the internal Abstract Syntax Tree (AST) to LLVM. LLVM has grown over the years into the so called LLVM umbrella project. Part of it are multiple backends that can generate machine code for different architectures, e.g. IA-32, x86-64, ARM, Qualcomm Hexagon, AMDGPU, NVPTX, and many more. So, as long as a language can be mapped to LLVM it can be compiled to a wide range of architectures. This frees up lots of development time which would otherwise have to be spent on writing analysis and optimization tools for each target architecture. Suddenly, large compiler projects become tractable for small teams [2].

Prominent examples of languages that use a LLVM-based backend are C/C++ (Clang), Swift, Rust, and Kotlin (native). NVIDIA's CUDA compiler is also based on LLVM.

LLVM has one IR on which analysis and optimizations are performed. This single abstraction layer is extremely powerful because it keeps complexity low and mapping from C/C++-like languages to LLVM is straightforward. At the same time, this is also a limitation. A lot of analysis and optimizations are better performed on higher or lower abstraction levels. Lots of modern languages (including for example Swift, Rust, and Julia) currently implement their own higher level IR on which they perform analysis and optimizations before lowering to LLVM. The result is a fragmentation of IRs and tools around them. On the other side of the abstraction spectrum, namely target architectures that don't fit the instruction-set-like-model of LLVM such as Application-Specific Integrated Circuit (ASIC)s, FPGAs and modern machine learning accelerators are not well supported. Additionally, the tooling for Verilog-to-bitstream is vendor specific and notoriously closed. [11]

2.3 MLIR

MLIR [11] is a new compiler infrastructure that is being developed by the LLVM community. It is supposed to be very extensible and easily customizable. Problems in different domains are solved using different abstraction levels. For example, the Java Virtual Machine (JVM) optimizes programs on Java byte code level, whereas the Machine Learning (ML) community optimizes ML graphs. More specifically, different programming languages have their own IRs. Ultimately, this is because the abstraction level between the high-level AST of a particular programming language is too far away from the low-level LLVM IR. MLIR's developers see this fragmentation of IRs as one major contributor to a set of problems. Some are bugs in implementations, poor performance in compilation, lacking diagnostic and debugging quality [11]. The claim is that this is because developing IRs and tools around it come at a high cost. MLIR aims to be a general framework for unifying a large set of tools around different IRs. It does this by providing a standardized SSA IR infrastructure, means to declaratively define IRs in so-called dialects and tooling and infrastructure for printing, parsing, location tracking, pass management, and much more [11].

From the start, MLIR was designed with performance and scalability in mind. That's why a lot of the AST rewriting and mapping infrastructure can execute on multiple threads.

In the following, we want to explain some core components of MLIR that are important for understanding our work.

2.3.1 IR, Dialects and Passes

MLIR core concepts are custom IRs, dialects and the pass infrastructure.

An IR is made up of sequences of operations. These sequences can be nested in blocks. Blocks can be nested in regions. And regions themselves can be nested in operations. This is how control flow in software or modules in hardware can be modeled. Operations in a given IR are in the generic SSA form `%r1, ..., %rn = my_dialect.some_op %a1, ..., %an`. This means that an operation can have any number of inputs and any number of outputs. All identifiers prefixed with `%` are so-called values. They carry the inputs to and save the results from any operation. The SSA property dictates that they can only be assigned once and not overwritten. Now, operations, more specifically the types of their results have to annotated. The expression `%0 = my_dialect.some_op %a, %b : i32` shows that the type of the `%0` is a 32 bit integer. Type annotation allows operations to be overloaded for different types and prevents the set of operations to be unnecessarily large. Non-builtin types must be prefixed with a '`!`'. Types can also be parameterized, such as in `vector<4xf32>`. Operations can be further parameterized with attributes.

Attributes are key-value pairs. An example would be `%0 = my_dialect.some_op %a, %b {key = value} : !my_dialect.my_type`. Any IR can be printed in a "pretty form" or its standardized form. The "pretty form" can be custom defined and is usually designed to be more readable. But because it depends on the validity of the IR, printing it might be undefined and crash. The standardized form can always be printed is used when debugging. All examples we have shown so far are in the "pretty form". The "pretty form" is not limited to the printing side. The user can define the syntax for parsing as well. This means that each operation and type usually have two parsers and two printers each. One for the "pretty form" and one for the standardized form.

A dialect defines a set of operations and types. This includes the aforementioned printers and parser. To verify that a given expression is valid, a dialect also provides type checking abilities. Another important concept is the canonical form of expressions. It defines a singular standard representation for expressions that have multiple equivalent representations. An example would be to reduce the expression $x + 3 + 2$ to $x + 5$. Now, the mathematical theory behind this goes a lot deeper. It is sufficient to know that a dialect can provide methods that can canonicalize expressions in that dialect. Expressions in canonical form are generally much easier to pattern match for further optimization. A common example would be constant folding. Dialects can be declaratively defined using Operation Defining Specification (ODS). ODS is a specification language from which an underlying C++ implementation is generated. ODS just exists for convenience purposes. A user could theoretically define a complete dialect in C++. But that is impractical. Usually, a mix of ODS and C++ is used to implement a dialect. C++ is often used for complex logic where ODS doesn't suffice.

Passes define match-and-rewrite patterns on operations. They are used to convert operations from one dialect into operations of another dialect. Some passes are used for lowering dialects to a more low-level representation, converting dialects to equivalent forms in other dialects, and performing specific optimizations in a certain dialect. A lot of passes are part of the dialect definition themself, but not always. Some passes are dialect agnostic and can be applied to any operation that has certain properties. For example, an operation can be defined to be associative. Now, passes that optimize by exploiting associativity can be applied to that operation.

In the following, we illustrate some of the important concepts by example.

Figure 2.2 shows a function that returns $x^2 + y^2$ for integer inputs x and y. Stepping line by line through the IR we can observe the following:

- `func.func` is an operation in the `func` dialect the defines typical functions known from programming languages. It has a name, defines input arguments and a return type. The body of the function itself is a sequence of operations that are contained in a block. The `func.func` operation contains that block.

```
module {
  func.func @square_and_add(%x: i64, %y: i64) -> i64 {
    %0 = arith.muli %x, %x : i64
    %1 = arith.muli %y, %y : i64
    %2 = arith.addi %0, %1 : i64
    func.return %2 : i64
  }
}
```

Figure 2.2: Example of MLIR code.

```
%0 = arith.constant 42 : i64
%1 = arith.addi %0, %0 : i64
```

(a) Minimal example of MLIR code that adds the constant 42 to itself.

```
%0 = arith.constant 84 : i64
```

(b) The minimal MLIR code is optimized by a canonicalization pass.

```
OpBuilder builder(mlirContext);
Type i64Type = builder.getI64Type();
IntegerAttr attr = getI64IntegerAttr(42);
Value x = builder.create<arith::ConstantOp>(i64Type, attr).getResult();
Value y = builder.create<arith::AddIOp>(x, x).getResult();
```

(c) The minimal MLIR code can also be generated using the `OpBuilder` class in C++.

- `arith.muli` is an operation in the *arith* dialect. It is a binary operation that multiplies its two signed integer operands. The result is assigned to the value `%0`.
- `func.return`, again part of the *func* dialect, returns the value of its operand.
- Everything is encapsuled in the `module` operation. For convenience, it is often omitted.

Dialects can be intermixed as long as the types can be verified and the IR is structurally sound. This is very powerful because every dialect can focus on solving a problem of a narrow scope. For example, the *arith* dialect only implements basic arithmetic operations on integers, floats and vectors. *arith* on its own would be useless without other dialects that describe control flow, functions and eventually LLVM-IR.

We briefly want to demonstrate how the effects of a canonicalization pass affect the IR and how the `OpBuilder` Application Programming Interface (API) corresponds to IR. Figure 2.3a shows an example of a very simple MLIR IR where a constant is added to itself and saved into a register. Figure 2.3b shows that IR after it was processed by a canonicalization pass that performs constant folding. Figure 2.3c shows how the original IR can be generated in C++ programmatically.

2.4 CIRCT

Similar to software, the tooling around IRs in hardware design is very fragmented. CIRCT's idea is to develop IRs and tools for hardware design. It is built on top of MLIR and uses its infrastructure. CIRCT's most important components for this work are its set of dialects and passes, a *SystemVerilog* generator and tooling for working with FIRRTL. CIRCT has dialects for describing circuit design on a very high functional level down to Register Transfer Level (RTL) logic. The passes are used for lowering high level IRs to lower level IRs. In the end, the *SystemVerilog* generator can be used to generate *SystemVerilog* code from the IR.

We will only deal with a small set of CIRCTs dialects:

- *hw* is a dialect for describing hardware modules and instances of them, their ports, wires and very basic aggregate types.
- *hwarith* defines operations for basic arithmetic operations on integers.
- *comb* defines simple logical combinational operations.
- *seq* defines operations modeling sequential logic.
- *sv* is a dialect that models constructs in the SystemVerilog language. Note, that this by no means captures the complete feature set of SystemVerilog.
- *firrtl* models the FIRRTL IR developed by Patrick S. Li et al. from the University of California, Berkeley [12].

2.4.1 FIRRTL

FIRRTL is an intermediate language outside of CIRCT that was created alongside Chisel [1]. Chisel is a hardware construction language embedded in Scala. As the name suggests, FIRRTL describes RTL logic. It was designed to be detached from Chisel such that other tools can interface with it. Additionally, its semantics are well defined and completely specified. CIRCT comes with a FIRRTL dialect and parser that can import `.fir` files generated by Chisel directly into CIRCT.

As Chisel and FIRRTL were developed alongside each other, there is a clear mapping between primitives in Chisel and FIRRTL. After all, Chisel was originally designed to model FIRRTL after all Chisel Scala code has executed. A consequence is that relatively high-level constructs such as vector types, struct (called bundle) types, memory, conditional statements and modules have first-class support in the FIRRTL language.

At the time of writing CIRCT has no *SystemVerilog* parser. This means, that the only entry point for our use case into the CIRCT ecosystem is via FIRRTL. This is necessary, as we reuse some work from XSPN-FPGA which will be introduced in the following chapter.

3 Related Work

3.1 TaPaSCo

TaPaSCo is an open-source framework developed by the Embedded Systems and Applications (ESA) group at the Technical University Darmstadt [8]. It features a frontend for Xilinx's and other synthesis tools to target multiple different FPGA architectures. The user provides a design as a packaged *IP-XACT* core. As long as the provided design contains a top module with an AXI4-Lite port for control registers, an AXI4 port for memory and an interrupt signal, *TaPaSCo* can generate a bitstream for a wide variety of FPGA target architectures. Furthermore, *TaPaSCo* provides a runtime driver and API that abstracts away the low-level management of controlling the accelerator from a host program. Accelerator execution is just a matter of calling a function from the API with the appropriate arguments and waiting for a response. As long as the design contains the required ports, such as an AXI4 port for memory, an AXI4-Lite port for the register file, an interrupt signal and ports for clock and reset, *TaPaSCo* is able to manage the design as a so-called Processing Element (PE). PEs can be instantiated multiple times on the fabric and *TaPaSCo* automatically generates all the necessary hardware for controlling and managing the different PEs. The ports of the PEs are connected to the necessary ports for communication with the host program and Random Access Memory (RAM). But, these extended features of *TaPaSCo* are not important for this work. They have already been tested with XSPN-FPGA and this work is concerned with building a replica of the XSPN-FPGA toolflow in CIRCT.

3.2 XSPN-FPGA

XSPN-FPGA is the product of multiple works by Lukas Sommer, Lukas Weber, Julian Oppermann, Hanna Kruppe et al. [21], [22], [23], [9]. It is a compiler that produces synthesizable *SystemVerilog* source code from text-based SPN descriptions. XSPN-FPGA is implemented in *Scala* and uses *Chisel* as the hardware construction language. The synthesis and deployment to a target platform is done using the in-house framework *TaPaSCo*.

3.2.1 Technical Details

The compiler stages can be crudely summed up as:

1. A parser parses a text file containing the SPN definition and produces a DAG representation in Scala.
2. Some preprocessing steps are performed on the DAG to for example:
   - Binarize product and sum nodes.
   - Perform constant folding.
   - Compute required bit widths.
   - Scheduling of the nodes according to some operator-to-latency function using a Linear Program (LP) solver.
3. The DAG nodes are then mapped to hardware operators which are implemented in Chisel. Wire connections along the edges are created. Delays are inserted to meet scheduling requirements.
4. The datapath is wrapped in a top-level module that contains the appropriate input and output ports.
5. XSPN-FPGA calls *TaPaSCo* via the Command Line Interface (CLI) to import, compose and generate a fully functional bitstream.

As XSPN-FPGA is the sum of multiple works, it also has features that we want to shortly list here [1]:

1. The number format is exchangeable.
2. The bit widths of the used number format are configurable from the CLI.
3. Modulo scheduling and resource sharing are available.
4. Inference modes for marginal inference and most probable explanation are implemented.
5. Weights and histograms can be reconfigured at runtime.
6. Seamless integration with synthesis Tapasco.
7. Seamless integration with CocoTb.

[1]At the time of writing some features are not found in any published work.

One of the key goals of our work is to implement (1), (2), (6), (7) as we deem these essential in order for our work to be comparable to a bare-bones version of XSPN-FPGA. If the implementation is sufficiently well done, the other features should be easy to implement as well.

This makes XSPN-FPGA an all-around working tool for compiling SPNs to FPGAs.

3.2.2 The UFloat Number Format

We want to shortly describe the *UFloat* (unsigned float) floating point number format, as it is an integral part of this work. The *UFloat* format is an in-house specification and implementation for floating point numbers in XSPN-FPGA. A *UFloat* number is represented using $b_e + b_m$ bits, where b_e specifies the number of exponent bits and b_m specifies the number of mantissa bits. The encoding for exponent and mantissa follow the same logic as in IEEE 754 floating point numbers [5]. The only difference is that *UFloat* numbers are unsigned as they are designed to only deal with probabilities in SPNs. The missing sign bit simplifies the implementation of arithmetic operators. XSPN-FPGA contains implementations for addition, multiplication and conversion to a traditional 32-bit or 64-bit IEEE 754 floating point format. Rounding and truncation are not supported. Also, for simplicity, infinities and NaNs are not dealt with. The operators are completely implemented in Chisel.

3.3 Applications of CIRCT

Mike Urbach and Morten B. Petersen explore applying CIRCT for mapping *PyTorch* models to synthesizable *SystemVerilog* code in their published work [13] at LATTE '22 [2].

They implement a MLIR/CIRCT-based compiler stack for gradually lowering a *PyTorch* model to *SystemVerilog* code. Some different possible lowering pipelines are sketched out in figure 3.1. Numerous different MLIR and CIRCT dialects are used so that appropriate optimization can be performed on each level. They claim that this is vastly different from common approaches, where complex designs are usually expressed in C++ High-Level Synthesis (HLS). The benefit is, that optimizations can work with more information because high-level intent is propagated downwards. As an example, C++ HLS might describe a *PyTorch* matrix multiplication as nested loops. As a result, reasoning about matrix decompositions and memory access patterns becomes very difficult.

[2]https://capra.cs.cornell.edu/latte22/

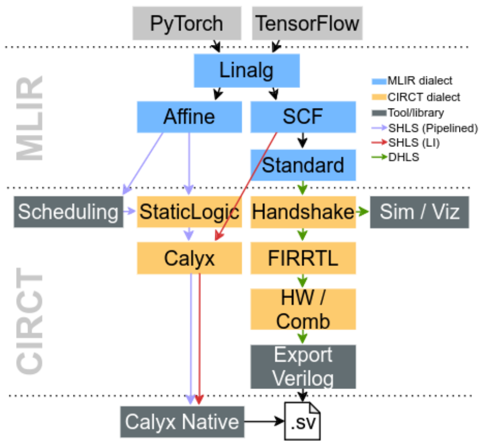

Figure 3.1: Different possible lowering pipelines for PyTorch models. Source: [13]

4 Base Framework

4.1 Architecture of The Sum-Product Network Compiler

Our work extends the SPNC project by Lukas Sommer et al. [20]. SPNC can compile text-based SPN definitions to essentially any CPU target supported by LLVM and *CUDA* GPUs. Vectorization is currently limited to x86 and *ARM Neon* targets. SPNC is written in C++ and Python and builds on MLIR. It also comes with a runtime. The frontend is implemented in Python mainly for convenience purposes. In this section, we want to break down some technical aspects of SPNC in detail as our work builds upon those.

4.1.1 IRs and Dialects

SPNC presents two new MLIR dialects. One is called *HiSPN*, and the other one *LoSPN*. As their names suggest, *HiSPN* represents more abstract high-level concepts of SPNs, while *LoSPN* has more concrete low-level aspects in mind. We will just briefly touch upon *HiSPN* and focus on *LoSPN*, as it is the dialect we are mostly interested in.

HiSPN is designed to closely mimic the representation of SPNs in *SPFlow*. An example snippet can be seen in figure 4.1b. Note, that the operations inside the body of `hi_spn.graph` model the DAG. The surrounding operations model the inference query. This allows future extensions to easily put the inner DAG into different query type operations. For example, one could imagine a `hi_spn.mpe` operation that computes MPE queries.

LoSPN is designed to model the actual computations used in the query. The DAG it encapsulates only consists of unary and binary arithmetic operations. Furthermore, weighted sum operations are decomposed into binary sum, product and constant operations. The generic probability type has been replaced with an actual floating point type. Operations surrounding the DAG model lower-level operations such as reading and writing from memory and batch size. This is of relevance in the context of execution on GPUs.

Only a subset of *LoSPN* is relevant for this work. We will only focus on the `body` operation and the contained arithmetic operations.

4.1.2 Compilation Flow

The steps from SPN description to a *LoSPN* representation are independent of the target platform and look as follows:

1. The SPN is parsed and converted to a binary format using Python and *CapnProto* [1].
2. The actual compiler parses the previously generated binary file and translates the SPN structure to a *HiSPN* representation. This step is fairly straightforward because, as previously mentioned, *HiSPN* is designed to closely mimic the SPN structure as it's defined in *SPFlow* section 4.2.
3. The *HiSPN* IR is lowered to *LoSPN*.
4. *LoSPN* is optimized using dialect-specific optimization and dialect-agnostic passes from MLIR.

An example of a text-based SPN description, conversion to *HiSPN* and lowering to *LoSPN* can be found in the figures 4.1a, 4.1b and 4.1c. Only excerpts are displayed for brevity. Notice, how for example, the `hi_spn.product` operation in *HiSPN* is lowered to a sequence of `lo_spn.mul` operations.

We left out some intermediate steps, namely graph partitioning and bufferization. Very large DAGs are partitioned to create a set of smaller DAGs contained in tasks. Because we are interested in fully pipelined architectures as in XSPN-FPGA, graph partitioning is not relevant to our work and will be left out. Bufferization is relevant when thinking about load and store operations from and to memory. *LoSPN* uses MLIRs tensors and memrefs for this. This also doesn't have any relevance to our work.

Finally, the *LoSPN* IR is lowered to a CPU or GPU target.

We briefly want to give an overview of how the compilation flow is realized in different pipeline stages. These pipeline stages essentially are C++ classes that implement small specific self-contained tasks. As an example, we assume that the compilation is performed for x86 CPUs without vectorization. It is illustrated in figure 4.2. As the names suggest, all stages from `LoSPNtoCPUConversion` onward are specific for CPU compilation.

[1]`https://capnproto.org/`

```
SumNode_0 SumNode(0.19583*ProductNode_1, 0.80416*ProductNode_7){
    ProductNode_1 ProductNode(HistogramNode_2, HistogramNode_3, HistogramNode_4, HistogramNode_5
    , HistogramNode_6){
        HistogramNode_2 Histogram(pca|[ 0., 1., 2., 3., 4., 5., 6., 8., ...],[...])
        ...
    }
    ...
}

```

(a) Excerpt from the text-based description of the *NIPS5* SPN.

```
...
"hi_spn.graph"() ({
  ^bb0(%arg0: i8, %arg1: i8, %arg2: i8, %arg3: i8, %arg4: i8):
    ...
    %11 = "hi_spn.product"(%6, %7, %8, %9, %10) : (!hi_spn.probability, ..., !hi_spn.probability)
      -> !hi_spn.probability
    %12 = "hi_spn.sum"(%5, %11) {weights = [0.195833, 0.804166]} :
      (!hi_spn.probability, !hi_spn.probability) -> !hi_spn.probability
    "hi_spn.root"(%12) : (!hi_spn.probability) -> ()
}) {numFeatures = 5 : ui32} : () -> ()
...
```

(b) Excerpt of *HiSPN* code describing for the *NIPS5* SPN.

```
...
%5 = "lo_spn.body"(%0, %1, %2, %3, %4) ({
^bb0(%arg5: i8, %arg6: i8, %arg7: i8, %arg8: i8, %arg9: i8):
  ...
  %22 = "lo_spn.mul"(%17, %18) : (f64, f64) -> f64
  %23 = "lo_spn.mul"(%22, %19) : (f64, f64) -> f64
  %24 = "lo_spn.mul"(%20, %21) : (f64, f64) -> f64
  %25 = "lo_spn.mul"(%23, %24) : (f64, f64) -> f64
  %26 = "lo_spn.mul"(%16, %7) : (f64, f64) -> f64
  %27 = "lo_spn.mul"(%25, %6) : (f64, f64) -> f64
  %28 = "lo_spn.add"(%26, %27) : (f64, f64) -> f64
  ...
}) : (i8, i8, i8, i8, i8) -> f64
...
```

(c) Excerpt of *LoSPN* code after lowering for the *NIPS5* SPN.

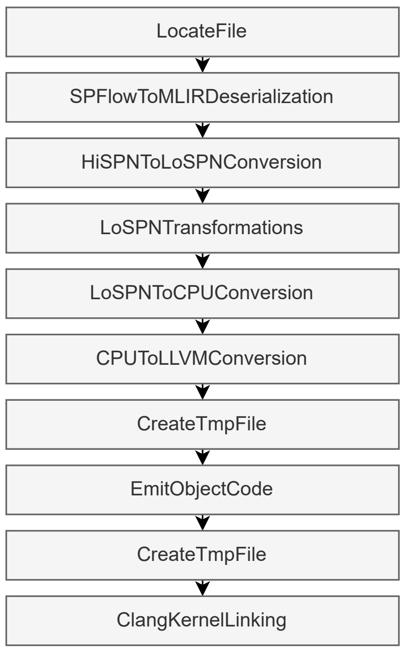

Figure 4.2: Examples of the compilation pipeline steps executed when targeting x86 CPUs without vectorization.

4.1.3 Frontend and Runtime

The *Python* frontend uses *PyBind11* [2] to interface with the actual compiler and the runtime. First of all, the SPN must be converted into a binary format with *SPFlow* section 4.2 and *CapnpProto*. This is done in *Python*. Then the different compiler methods from `CPUCompiler` or `GPUCompiler` are called. The interface with the compiler using *PyBind11*. They set the appropriate CPU arguments and point to the newly generated binary SPN file. Then the compiler takes over and, if successful, produces a shared library object. A kernel structure is returned that contains some information about the kernel and where the shared library object is located. It passes the kernel structure and query data to the runtime. The runtime loads the shared library object with `dlopen()` and calls the appropriate functions inside it. The result is returned to the frontend. Compiler and runtime are implemented in C++.

[2] `https://pybind11.readthedocs.io/en/stable/`

4.2 SPFlow

SPFlow is a Python library for SPN construction, manipulation, inference and learning. It is developed by the ML group at the Technical University Darmstadt. *SPFlow* is used by SPNC's frontend to parse SPN definitions which are subsequently converted to a binary format for SPNC itself.

In this work, *SPFlow* is used for verification. We randomly generate input samples and compute the resulting probability of the SPN using *SPFlow*. Because it is well-tested and has a high-level software implementation of bottom-up inference, it is well-suited for assessing the correctness of our hardware implementation. The additional benefit is that it integrates perfectly with *CocoTb*, a Python-based simulation suite for *SystemVerilog* designs. It is briefly introduced in the next section.

4.3 CocoTb

CocoTb [3] is a Python-based simulation framework. It is open source and functions as a frontend for a wide variety of different backends, such as *Verilator* [4] and *ModelSim* [5]. The developer builds coroutine-based test benches in a Python script. Together with the AXI extension, it allows the developer to quickly set up asynchronous test benches for complex circuits. Because it is Python-based, it integrates with *SPFlow*.

4.4 ESI

ESI is a dialect and a set of tools in CIRCT. It aims to solve the problem of developers having to manually implement wire communication protocols between different Intellectual Propertys (IPs). The problem is that connecting different IPs might require manually building protocol and bit width converters (aka. gearboxing) between them. This is painful and error-prone, as debugging happens on the wire level and requires interpretation of the blobs being sent around.

ESI wants to help solve this problem in a number of ways. First, there is the ESI dialect. It defines channel types and operations on them. A ESI channel is used to define a producer or consumer interface on *HW* modules. Instead of defining a valid signal input, a payload input, and a ready signal output, for example, the programmer can define a single ESI channel (as input) with the appropriate payload type. The ESI dialect defines operations for connecting ESI channels or unpacking channels into their underlying signals. Additionally, ESI provides means of directly

[3]`https://docs.cocotb.org/en/stable/`
[4]`https://www.veripool.org/verilator/`
[5]`https://eda.sw.siemens.com/en-US/ic/modelsim/`

```
hw.module @Sender() -> (x: !esi.channel<i1>) {
  %false = arith.constant false
  %chanOutput, %ready = esi.wrap.vr %false, %false : i1
  hw.output %chanOutput : !esi.channel<i1>
}
```

(a) Example of a simple producer module that utilizes ESI.

```
hw.module @Sender(%x_ready: i1) -> (x: i1, x_valid: i1) {
  %false = arith.constant false
  %chanOutput, %ready = esi.wrap.vr %false, %false : i1
  %rawOutput, %valid = esi.unwrap.vr %chanOutput, %x_ready : i1
  hw.output %rawOutput, %valid : i1, i1
}
```

(b) Application of the `--lower-esi-ports` pass.

```
hw.module @Sender(%x_ready: i1) -> (x: i1, x_valid: i1) {
  %false = arith.constant false
  hw.output %false, %false : i1, i1
}
```

(c) Application of the `--lower-esi-ports` and `--lower-esi-to-hw` passes.

exposing a channel to a co-simulation system. The idea is that a module can have a consumer interface, perform a function on the incoming data and send the results out via the producer interface. ESI provides operations that allow exposing these modules to a Verilator testbench.

We want to demonstrate the wrapping and unwrapping of channels using an example. Figure 4.3a shows a simple module that has a producer interface. A producer interface is exposed as the return of an ESI channel type. An ESI channel itself is not directly convertible to RTL. First, the ports facing the outside world must be lowered to a core dialect representation. Figure 4.3b shows how the IR is transformed when the appropriate pass is applied. The `esi.unwrap.vr` operation exactly describes how to unpack an abstract ESI channel into its constituent parts and allows all module ports to be represented using core dialects. Having successive wrap and unwrap operations (or vice versa) seems like a no-op. This is exploited by the pass `--lower-esi-to-hw`. A canonicalization step is applied that removes successive wrap/unwrap operations and what we are left with is a module in pure core dialect. Figure 4.3c shows the result of applying both passes. This demonstrates the very powerful concept of tying together abstract ESI channels with concrete core dialect circuitry.

5 Minimum Viable Product

The goal of this work is to extend SPNC such that it implements an end-to-end toolflow for synthesizing test-bases SPN descriptions for FPGA targets. SPNC shall offer a compilation pipeline for FPGAs parallel to the ones for CPUs and GPUs. As with XSPN-FPGA, SPNC must be able to implement surrounding controller logic and infrastructure such that the design can be synthesized with the *TaPaSCo* framework. Furthermore, the arithmetic operators for the datapath shall be customizable as this is a key feature of XSPN-FPGA.

5.1 Approach

This work is explorative in nature. We expected to face numerous challenges and therefore decided to set a MVP. It compromises on a lot of the initial goals but helps us guide in further development. It revealed lots of the problems which we addressed in the improved implementation. For the MVP we decided to borrow lots of the controller and arithmetic operator logic from XSPN-FPGA. This compromises on SPN's ability to generate its own controller and operator logic.

Our approach picks up where the pipeline step *LoSPNTransformations* leaves off. *LoSPNTransformations* returns an optimized IR in the *LoSPN* dialect. We implement a step called *LoSPNtoFPGAConversion* that is the Hardware Description Language (HDL) analog to *LoSPNtoCPUConversion*. It performs the following basic steps:

1. The inner DAG is extracted from the `lo_spn.task` operation.
2. Every *LoSPN* operation is mapped to a set of *HW* operations.
3. The *HW* operations are scheduled using CIRCTs scheduling framework by Julian Opperman et al. [16].
4. The scheduled *HW* operations are embedded in a controller that handles the communication between the host and the FPGA. In the MVP the controller logic is sourced from the old XSPN-FPGA project.

5. The controller is exported to *SystemVerilog* source files. Tool Command Language (TCL) scripts are generated that call Vivado and various *TaPaSCo* tools to produce a bitstream.

5.2 Implementation

As previously stated, the goal of the MVP was to get a prototype up and running. Overall quality and features were not a priority.

SPNC breaks down the toolflow into steps that are chained together into a pipeline. Each of these pipeline steps performs a single basic task. Our extended pipeline architecture looks like the following:

1. `LocateFile`: Locate the serialized SPN file
2. `SPFlowToMLIRDeserializer`: The serialized SPN is deserialized into MLIR IR
3. `HiSPNToLoSPNConversion`: Conversion of *HiSPN* to *LoSPN*
4. `LoSPNTransformations`: *LoSPN* transformations
5. `LoSPNToFPGAConversion`: Conversion of *LoSPN* to FPGA (*)
6. `EmbedController`: Embed the datapath into the surrounding controller (*)
7. `CreateVivadoProject`: Creates a Vivado project (*)
8. `ReturnKernelInfo`: Returns kernel and debug Information (*)

The steps marked with (*) are the new ones implemented as part of this work. It essentially replaces the steps of converting to LLVM-IR and compiling to a shared library object file. We decide to split up the tasks of the steps the following way: `LoSPNToFPGAConversion` deals with mapping *LoSPN* to MLIR IR that implements circuitry for the datapath. `EmbedController` deals with building the surrounding controller architecture to get a usable PE. `CreateVivadoProject` generates TCL scripts and performs all the CLI interfacing with Vivado and *TaPaSCo*. `ReturnKernelInfo` is more of a dummy step that returns a struct containing information such as variable count and bit widths. This struct is returned by the Python interface for the runtime. The steps `LoSPNToFPGAConversion`, `EmbedController` and `CreateVivadoProject` can be skipped to directly return the kernel information. This is useful for loading a finished bitstream without synthesis. The reason for this design should become more clear after reading the following sections.

5.2.1 LoSPN To FPGA Conversion

This step is mainly concerned with mapping the *LoSPN* IR to a *HW* IR, which in turn can be exported to *SystemVerilog*.

We traverse the SPN IR until we find a `lo_spn.body` and extract its contents. What we get is a list of operations exactly describing the bottom-up computation in a SPN using only arithmetic operations. All operations take MLIR values as inputs and produce a single MLIR value as output. Since all dialects in MLIR are in SSA form we can think of the assignees (values) as wires and the assigners (operations) as instances of hardware modules. This lets us directly map each operation to the correct `hw.instance` operation. `hw.instance`'s inputs are mapped to its input ports. Its output ports are mapped to its result values. This makes the *HW* dialect very convenient in our case. Of course, synchronous hardware modules additionally have a clock and reset port. These are added to the `hw.instance`'s input as well.

`hw.instance`'s have to refer to module implementations. We make a distinction between arithmetic operations and histograms/categorical operations.

For arithmetic operations, we make `hw.instance` operations refer to `hw.module.extern` operations that represent an implementation in an accompanying *SystemVerilog* source file. For now, only operators for 31-bit *UFloat* values are supported. Their implementation stem from the *UFloat* operator implementation in XSPN-FPGA.

Histogram and categorical nodes are implemented programmatically. A `hw.module` is generated for each histogram/categorical node with a clock, reset, input index and output probability ports. The underlying probability distribution is unpacked. *LoSPN* histogram nodes store two arrays: One for the probabilities and one of equal size for the *buckets*. These so-called *buckets* are integer tuples. A tuple (i, j) with $(i < j)$ at index k means that indices $\{i, i+1, ..., j-1\}$ get mapped to the probability residing at index k. Therefore, a flattened version is created, where probability at index k is repeated $j - i$ times. This is done because the compressed representation serves no benefit on hardware. For categorical nodes, the flattened vector can be extracted directly. Each probability, a 64-bit floating point value, is first converted to a 32-bit floating point value. This in turn is bit-cast to a 31-bit integer. Assuming non-negativity of the probability this returns a valid bit representation for a 31-bit *UFloat* format. All these 31-bit (now) integers are instantiated as `hw.constants`. All constants are collected in a `hw.array`. The correct probability is extracted from the array using the input index by the `hw.array_get` operation. The resulting 31-bit integer is returned using `hw.output`. Some intermediate buffering is done by using `seq.firregs` to improve pipelineability.

`lo_spn.constants` are just mapped to `hw.constants`. The contained floating point value contained in `lo_spn.constant` is converted to a 31-bit *UFloat* format. Again, the 64-bit float is converted to a 32-bit float and then bitcast to a 31-bit

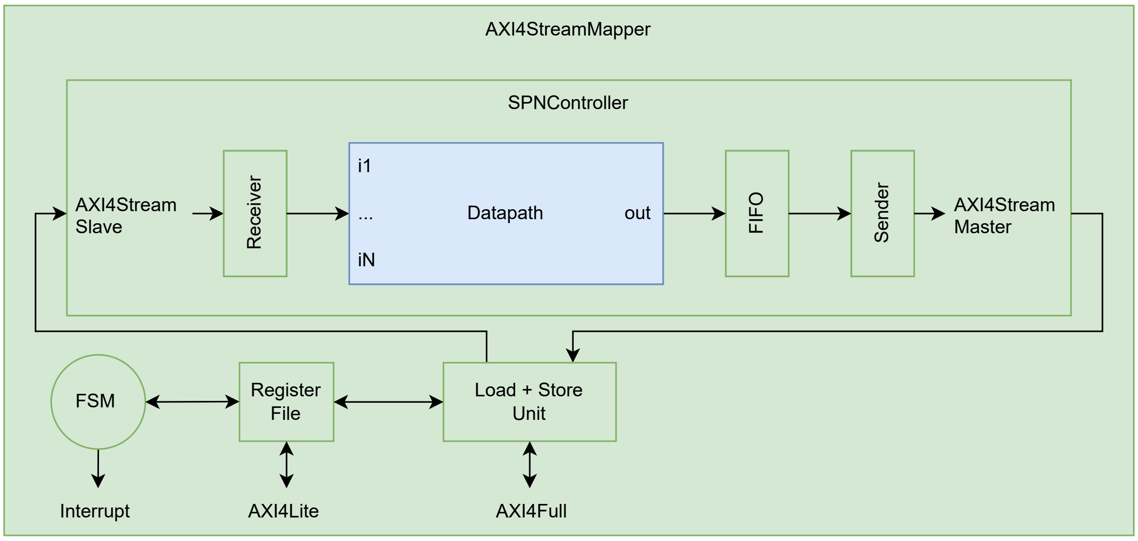

Figure 5.1: Controller architecture overview: The green parts are generated by XSPN-FPGA; The blue part is the previously generated datapath.

integer.

After all operations have been mapped or discarded, the newly described circuit must be scheduled. This is done using CIRCT's scheduling functionality. We can define a latency for each operation and call a function to schedule the operations. Then we can query the start and end time for each operation. If we have two operations A and B, where the result of A is the input of B, we query the end time of A and the start time of B. The result of A then must be delayed by this time delta Δt (which is integral). This is done by inserting Δt `seq.firreg`s.

5.2.2 Embed into Controller

To make the previously generated SPN datapath usable, some surrounding infrastructure is required. We generally refer to this as the controller. As we were only interested in building a working prototype at this point, we tried to reuse as much from the XSPN-FPGA project as possible. The controller architecture is roughly described in figure 5.1. Everything but the datapath from the previous section is imported from XSPN-FPGA. Lots of the implementation details of the controller are omitted and explained in later chapters. For now, it is important to know that most of the controller's implementation is parameterized. This means a lot of the bus widths and queue depths depend on the number of variables of the SPN, the output floating point format and the total delay of the scheduled SPN. XSPN-FPGA is a Scala CLI for which we swap out the compiler driver with a custom one. Our driver takes a few arguments from the command line that fully specify the controller. XSPN-FPGA then generates FIRRTL source files instead of *SystemVerilog*. These files can be parsed with CIRCT's `FIRRTLParser`. The parsed file is then inside CIRCT's ecosystem in the form of FIRRTL IR. This is then lowered to the *HW* dialect using numerous passes. Finally, the SPN datapath IR can be inserted into the IR tree of the previously parsed and lowered controller. In the end, the complete IR is lowered to *SystemVerilog*.

Correctly lowering from FIRRTL to *HW* is best done by drawing inspiration from CIRCT's `firtool`. The actual lowering pass, namely `LowerFIRRTLToHWPass` is preceded by passed that infer bit widths, lower FIRRTL types and expand `firrtl.whens`. Empirically, we found that these passes are necessary. Otherwise, `LowerFIRRTLToHWPass` would fail because that pass does not know how to lower every possible FIRRTL construct, for example, `firrtl.when`.

To further lower *HW* to *SystemVerilog*, we apply a pass that lowers *Seq* FIRRTL registers to *SystemVerilog* and a pass that implements *HW* memory. Finally, the `exportVerilog()` function from CIRCT can be used.

The datapath in our modified XSPN-FPGA sits inside the controller. Instead of the true datapath, we instantiate a Chisel *BlackBox* module, named "spn_body". Through FIRRTL and the numerous lowering passes it is mapped to a `hw.instance` operation that links to a `hw.module.extern` operation. We search through the whole IR tree for that `hw.instance` operation and replace it with a `hw.instance` operation that links to out *HW* datapath IR. The *HW* datapath is inserted in the controller IR such that it sits on the same level as all the other `hw.modules`. All of this is done programmatically in C++ and quite cumbersome as it requires pushing around a lot of MLIR values from arguments and results.

5.2.3 Creating a Vivado Project

This pipeline step receives the IR from the previous step in the form of *HW* and *SV* dialect. As the name suggests, *SV* is a dialect that models *SystemVerilog* constructs. First, a project directory with sub-directories for source and configuration files is set up. Then, the IR is converted and saved using CIRCT's `exportSplitVerilog()` method. A TCL script for packaging the source files into a valid *IP-XACT* zip file is generated and written to the project directory. Then Vivado, `tapasco import ...` and `tapasco compose ...` are called in this order to synthesize a bitstream. From the output log, the bitstream location is extracted using a regular expression. The bitstream is copied to the project directory. The compiler can be configured to skip any of the sub-steps in case the user just wants to export the *SystemVerilog* files for example. This is useful for simulation.

5.2.4 Runtime and Frontend

To make working with the compiler more convenient, SPNC has a Python frontend. It contains classes that essentially wrap the compiler driver and runtime. Certain functions are bound via *PyBind11* and exposed to Python. The CPU and GPU pipelines are abstracted by the classes `CPUCompiler` and `GPUCompiler` respectively. They contain methods for compilation and execution. Both interact with separate components of SPNC, namely the compiler itself and the runtime. Compilation results in a `Kernel` C struct that contains information about the compiled SPN. This struct is required for execution. For example, it stores the name of the generated shared library object. This information is then passed to the runtime so that it can load the shared library object and start executing with the provided input data.

First of all, because synthesis times and the time required for loading a bitstream are very different compared to the classical CPU/GPU workflow, we decided to split up synthesis, loading the bitstream and execution into separate steps. We create a new class `FPGACompiler` with separate methods for compilation, loading and execution. This is different to `CPUCompiler/GPUCompiler` which performs all steps in one go.

`FPGACompiler`'s compile method calls the compiler, which, with the previously described `CreateVivadoProject`, calls the *TaPaSCo* tools and generates a bitstream. The bitstream is copied from the *TaPaSCo* work directory back into the current project directory. The `Kernel` struct is extended to be a `std::variant` of a `Kernel` for classical CPU and GPU compilation and FPGA synthesis. The runtime for FPGA accelerator execution needs some information about the internal bus bit widths for packing the input data correctly. This information is also stored in the `Kernel` struct.

`FPGACompiler`'s load bitstream function loads the bitstream using *TaPaSCo*. It uses information from the `Kernel` struct to locate the bitstream.

The execution method calls the C++ runtime and passes the `Kernel` struct, input data and output buffer to its arguments. The runtime is extended for the case where a FPGA accelerator is targeted. Together with the provided kernel id in the `Kernel` struct, it creates a *TaPaSCo* device. XSPN-FPGA saves on Peripheral Component Interconnect Express (PCIe) bandwidth by tightly packing the input data. An input sample for a SPN with N input variables is usually $8N$ bits wide. This is in contrast to CPU/GPU targets where an input sample is always a $32N$ wide. The input data is packed using *bitpacker* [1].

[1]`https://github.com/CrustyAuklet/bitpacker.git`

5.2.5 Testing and Simulation

CIRCT often tests components involving dialects with regression tests. A test defines an input, usually in the form of MLIR IR, the passes that are applied to that IR and the expected output IR. These tests are for the most part handwritten. They work well when the correctness of an implementation can be reliably asserted in a small contained setting. This is not the case for our project. We need some way to test the integrity of the whole architecture. Writing a handwritten example of a correct implementation even for the smallest SPNs would be tens of thousands of lines of *HW* IR. We design *CocoTb* testbenches that fully simulate the `SPNController` module which wraps the datapath with AXI4Stream ports. Using the AXI extension for *CocoTb* we can asynchronously send and receive data from Device Under Test (DUT). Input data is randomly generated. The expected output data is computed from the input data using *SPFlow*. This whole setup is very similar to how it's done in XSPN-FPGA. Being able to completely simulate the design is necessary as synthesis times are very long and debugging on the FPGA is very difficult.

5.3 Discussion

We identified a couple of problems with this implementation.

First of all, relying on an external Scala app violates the goal of having an end-to-end solution in a standalone application. It adds a lot of friction when a developer wants to implement a new feature in SPNC: Two separate software projects have to be maintained in sync. Furthermore, the CLI is very limited because every possible setting of internal queue sizes, bus widths and so on has to be mapped to CLI arguments. This is of course also the case when passing settings in the form of a text file for example. Together with this comes the fact that importing FIRRTL code into CIRCT and lowering to *HW* works well, but connecting the different *HW* components leads to a lot of boilerplate code. This presents a large surface for introducing bugs.

In the same vein, arithmetic operators cannot be configured with arbitrary bit widths which means every implementation of a possible number format (Logarithmic number system [24], Flopoco [4], *UFloat*) must be available for a large range of possible bit-width configurations in the form of source files. The fact that CIRCT at the point of writing does not have a *SystemVerilog* or *VHDL* parser makes this even more difficult.

Another observation we made, is that a lot of the complexity arises from defining, implementing and connecting ports for communication (for example AXI4-Stream) between different IPs. We were not the first to notice this as John Demme pointed out in his work about ESI [3].

To summarize our observed problems: (1) The Scala app prevents us from implementing an end-to-end solution, (2) the *HW* dialect is the wrong abstraction level for integrating FIRRTL controller logic and datapath and (3) communication logic is difficult. In the next chapter, we will propose solutions to these problems and discuss their implementation.

6 Refined Approach

In the previous chapter, we identified some problems with our MVP implementation. First, we discuss possible approaches to solving them and then concretely describe their implementation.

The first problem, namely dynamically generating implementation logic via the CLI is not an end-to-end solution and its subsequent problems can be addressed by generating the controller and operator logic inside CIRCT. CIRCT's core dialects (*HW*, *Seq*, *Comb*, ...) can express any digital logic that *SystemVerilog* can. However, using the `OpBuilder` API would be very cumbersome as common patterns of RTL languages are not abstracted away. For example, logical expressions, such as in *SystemVerilog* are not easily expressible using the `OpBuilder` API. This boils down to the fact that the abstraction level of the core dialects is too low for our use case and that using `OpBuilder` requires lots of redundant boilerplate code. Having some kind of frontend or framework that lets us describe the controller and operator logic and seamlessly integrates with CIRCT would be ideal.

To better evaluate our options, we formulate some requirements: We need some framework/dialect/frontend that contains or lets us easily implement simple hardware components such as First-In-First-Out (FIFO) queues, shift registers, counters, logical expressions, finite state machines and maybe even communication protocols. It must be human-writable, maintainable and testable through simulation or something similar. Additionally, it must be able to seamlessly integrate with CIRCT and SPNC. We considered multiple options.

One interesting idea was to use *Calyx*. It is a compiler infrastructure for accelerator generators [15]. At the time of writing it comes with:

- An intermediate language called *Calyx* for describing hardware designs.
- A compiler that produces *Calyx* code from the general purpose HLS language Dahlia [14].
- An interface with CIRCT through the *Calyx* dialect.
- Tools for simulation, verification and *Verilog* generation.

Calyx divides hardware logic into so-called components. Every component has a list of inputs, outputs and control signals to control execution or to signal when the execution is done. Components can be nested and interconnected. The programmer then defines in a software-like fashion how these components are scheduled and how their execution interdepend. This system gives *Calyx* the ability to better optimize the design than a traditional HLS tool. However, we don't see how this model of computation fits with XSPN-FPGA's accelerator design. XSPN-FPGA's datapath has no internal handshaking between the nodes to achieve maximum throughput and enabled Vivado to dedicate as many resources as possible to actual arithmetic computation. Furthermore, *Calyx* would introduce an external compiler toolchain as a dependency. SPNC would output *Calyx* code through CIRCT and run it through *Calyx*'s compiler. This would also violate the idea of SPNC being a self-contained end-to-end solution. We decided against using *Calyx*.

Moore is a compiler for compiling *SystemVerilog VHDL* to Low-Level Hardware Description (LLHD) assembly [18]. It has a focus on usability, clear error reporting and completeness. It also features bindings for CIRCT's core dialects. For our use case, the *SystemVerilog* parser seemed the most interesting as that is still missing in CIRCT. *Moore* would allow us to translate *SystemVerilog* source files into the LLHD dialect inside CIRCT. With the appropriate lowering steps, we could arrive at the core dialects and start integration with our datapath. However, this would require implementing a lot of the Chisel logic in *SystemVerilog* which is no easy task. Additionally, this would also introduce an external compiler toolchain.

FSM is a dialect inside CIRCT for describing Finite State Machines (FSMs). It integrates with the *HW* dialect but can also describe transitions in a more imperative software-like style. We asked ourselves if this would be fit to describe the FSM inside the controller or even AXI protocols. We decided against it for several reasons. First, the meat of the problem is building the generic width operators and a sufficient subset of the AXI protocols. It would be trivial to describe the FSM inside the controller in any of the other hardware dialects.

PyCDE is a Python-based frontend for working with CIRCT constructs. It can be used to design hardware circuits in a very similar fashion to classical Python based HDL projects like *MyHDL* [6]. Its target dialect is *HW* in combination with some other core dialects. It would solve our problem by making building the controller architecture and operators feasible. However, as it is implemented in Python it would require quite some interfacing with C++-based SPNC. In order to have a more seamless and better-integrated solution we decide against using *PyCDE* and build a custom frontend to the FIRRTL dialect.

Now, FIRRTL is another available dialect that was designed to be very close to Chisel. Chisel itself is mapped to FIRRTL code. Having a frontend to FIRRTL that is on the same abstraction level as Chisel would also solve our second problem, namely that connecting different components in *HW* is cumbersome. It would lift the problem of integrating multiple IPs onto a much higher abstraction level.

Our third problem, namely that communication protocols are difficult and error-prone to implement could be solved in the future by the ESI dialect. In our implementation, this problem is circumvented by reusing AXI4 logic from XSPN-FPGA in the form of project-accompanying FIRRTL source files and by just implementing a minimal working subset of AXI4-Stream. Additionally, we explored some possibilities in how to extend the ESI dialect with support for AXI4-Stream.

We discuss our refined approach in the following sections.

6.1 Approach

Following the previous discussion we want to clearly state our refined approach that addresses the three major problems.

We reimplement the controller logic and the arithmetic operators from XSPN-FPGA using CIRCT's FIRRTL. We build a DSL inside C++ that acts as a frontend for the `OpBuilder` API. This DSL is named `FIRRTL++` and is designed to closely mimic Chisel.

We extend ESI to support the basic mechanisms of the AXI4-Stream protocol. We give the user the option to annotate an ESI with the AXI4-Stream channel type. Additionally, we provide new dialect operations that convert between high-level AXI4-Stream channels and low-level ready-valid channels.

The basic pipeline structure from the MVP implementation is not subject to change. However, the *LoSPN* to FPGA conversion and the controller embedding step are. The conversion step now maps the *LoSPN* operations to FIRRTL++ Library (FIRRTL++) modules implementing the appropriate arithmetic operators instead of *HW* instances and modules. The controller embedding step now generates the surrounding controller logic in FIRRTL++ instead of interfacing with the CLI. Additionally, the user can choose if the controller embedding is done using ESI, instead of AXI4 ports.

6.2 Implementation

6.2.1 FIRRTL++

FIRRTL++ is a DSL implemented in C++ that integrates with the FIRRTL dialect in CIRCT. It is heavily inspired by Chisel. It utilizes things like operator overloading, implicit conversions and other C++ features to make the DSL look and feel like Chisel. The goal is to make it easy to write hardware descriptions in C++.

Behind the scenes, FIRRTL++ generates FIRRTL dialect operations. It contains a lot of helper functions that encode common patterns in hardware design. Since the result is FIRRTL IR, we get all the CIRCT tooling for lowering to different dialects, rewriting and canonicalization for free.

We chose to make our DSL look and act like Chisel for a number of reasons:

- The existing infrastructure is implemented in Chisel. It is well-tested and developed over a long period. To make implementing this infrastructure in FIRRTL++ as easy as possible, we want to make the DSL be as close to Chisel as possible.
- Chisel is a well-known and widely used DSL. It is used in many research projects and is taught in many universities. This means that many people are already familiar with Chisel. This makes it easier for them to use our DSL.
- Chisel is developed around FIRRTL. FIRRTL is also implemented as a dialect in CIRCT. This means that a lot of Chisel constructs are very easily translatable to FIRRTL++.
- Finally, Chisel itself is now being developed for many years and is very mature. This means that a lot of its constructs are well thought out and tested. This means we can safely borrow a lot of ideas from Chisel.

Inner Workings

FIRRTL++ makes heave use of C++'s operator overloading and implicit conversions. To ease the creation of arithmetic and logical expressions we create a class `FValue` which simply inherits from the MLIR class `Value`. `Value`s model the results and input arguments for operations. This includes the expressions in the `FIRRTL` dialect. `firrtl.add` takes 2 `Value`s as input and produces 1 `Value` as output. Because `Value`s are copy-constructible, turning a `Value` into a `FValue` is trivial. Now this gives us the ability to define operators on `FValue`s. As an example, figure 6.1 shows how we overload the + operator. It calls the `OpBuilder`s `create` method to instantiate an `firrtl.add` operation. Now, constructing complex expressions is much more readable and convenient. Figure 6.2 highlights: It's not hard to imagine that the "manual" way can become very cluttered for non trivial expressions.

We want to replicate Chisel's behavior as closely as possible. That includes the way how actual *SystemVerilog* code is generated with respect to instances and declarations of modules. Chisel is able to map two instances of the same module with the same parameters to a single definition in the *SystemVerilog* source file. For example, instantiating `Queue(5, U(32.W))` twice causes a single implementation in *SystemVerilog*, i.e. Queue_5_type_u32, whereas `Queue(5, U(32.W))` and `Queue(6,`

```
FValue FValue::operator+(FValue other) {
    return opBuilder.create<AddPrimOp>(opBuilder.getUnknownLoc(), *this, other);
}
```

Figure 6.1: Example of implementing the + operator for `FValue`s.

```
FValue a, b, c, d;
...
FValue e = (a + b) | (c & d);
// is equivalent to
Value a, b, c, d;
...
Value tmp1 = opBuilder.create<AddPrimOp>(opBuilder.getUnknownLoc(), a, b);
Value tmp2 = opBuilder.create<AndPrimOp>(opBuilder.getUnknownLoc(), c, d);
Value e = opBuilder.create<OrPrimOp>(opBuilder.getUnknownLoc(), tmp1, tmp2).getResult();
```

Figure 6.2: Example of constructing a complex expression with and without operator overloading.

`U(32.W))` cause two separate implementations. To emulate this behavior we need some way to uniquely identify a module. We chose to let the user implement modules by inheriting from FIRRTL++'s `Module<T>` class, where `T` is the inheriting module. This pattern is called the Curiously Recurring Template Pattern (CRTP). First of all, this lets us uniquely identify each class `T` by having a `static char id` inside `Module<T>` and taking its address. That address can be interpreted as the class id. Next, the user can pass the Input/Output (IO) port list and any additional parameters to the `Module`'s constructor. The class id, the port list and any additional parameters are passed to the `FIRRTL++` context. It generates a unique id and registers it if it hasn't been seen before. If it has, then this implies that the module's body has already been defined.

Another feature of Chisel is automatic context management. Clock and reset signals are not explicitly defined in the IO ports of Chisel modules. But they are implemented in *SystemVerilog*. The usage of clock and reset signals in Chisel usually happens implicitly. The user does not have to manually connect clock and reset signals to submodules, registers and so forth. We can achieve something similar in FIRRTL++. The current clock and reset signals lie on an internal stack. This is because modules can be nested arbitrarily deep. Then `Module`'s constructor returns. The inheriting class is expected to immediately call `Module`'s `build()` method after that. `build()` first checks if the `firrt.module`'s body has already been defined. If so, it returns. Otherwise, it sets the internal `OpBuilder`'s insertion point to the start of `firrt.module`'s body. Furthermore, a new pair of clock and reset signals are pushed to the stack. They now point to the first two input arguments of the current `firrt.module`. The other IO ports have to be treated in a similar way. `Module` offers the `FValue io(std::string)` method which behaves differently if it is called from within the body or from the outside. If it is called from within the body, it returns the `FValue` of the current `firrt.module`'s port with the given name. If it is called from the outside, it returns the appropriate return value from the `firrt.instance` of that module. The FIRRTL dialect proves very convenient here because output ports are also return values from the `firrt.instance` operation.

To allow for seamlessly integrating parsed FIRRTL IR and IR generated by FIRRTL++ we implement a class named `FIRModule` which can wrap any `firrtl.module` and lift it into the FIRRTL++ context. As there is no body to implement, type registration and body implementation are skipped. IO port access also is a little simpler, since ports from FIRRTL's perspective are only accessed from the outside. Apart from that, `FIRModule` behaves exactly as `Module` which allows for seamless integration with FIRRTL++. A use case is discussed in the following sections.

After discussing the technical details we want roughly give an idea about the scope of the core library. With the aforementioned `FValue` we are able to construct complex expressions just like one would in Chisel. There are the usual logical and arithmetic unary and binary, shift operators, operators for static and dynamic indexed access for vector types and a field access type for bundle types. There is the connection operator `a <<= b` that connects any `FValue` a with `FValue` b. FIRRTL++ provides slightly higher-level functions for bit concatenation, composing vector values and folding. Their common property is that their input are generic Standard Template Library (STL) containers whose element type is compatible with `FValue`. For example, the user can concatenate `FValue`s a and b with `cat({a, b})`. FIRRTL++ provides classes that wrap `firrtl.regreset` and `firrtl.wire`. Clock and reset signals for the registers are provided by the FIRRTL++ context. Similar to Chisel there are `regReset(...)`, `regInit(...)` and `regNextWhen(...)` to provide convenience in common register use cases. We provide a class that creates a FIRRTL memory and provides easy read/write access through `FValue`s. Finally, FIRRTL++ also provides when constructs. `when(FValue condition, std::function<void()> bodyConstructor)` is a convenient way for defining when-elsewhen-otherwise blocks. Again, this is designed to be as close to Chisel as possible.

Reimplementing Basic Components and Arithmetic Operators

The power of FIRRTL++ revealed itself when implementing a basic queue. We implement it similarly to Chisel where the data buffer is an array of registers and two registers hold the enqueue and the dequeue index. The total C++ code size amounts to about 80 lines of code, including comments and spacing. The queue's size and element type are parameterizable.

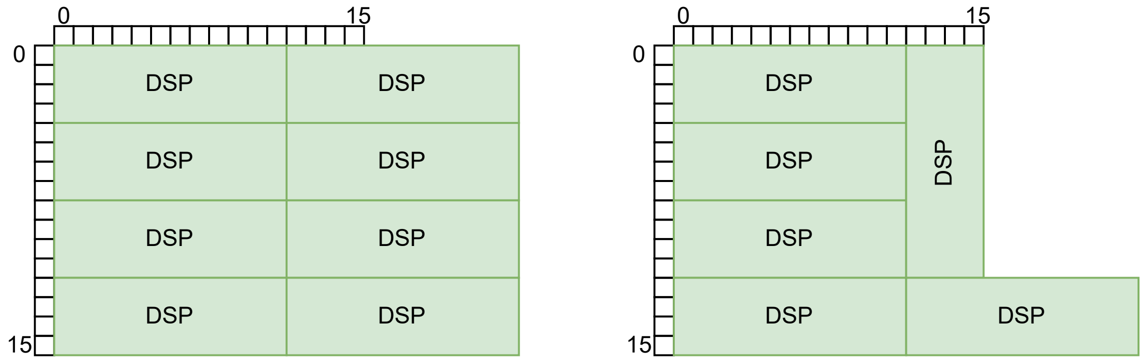

Figure 6.3: Matching sub-packages of the two input mantissas for multiplication with Digital Signal Processings (DSPs). The left one is our naive implementation; the right one is a more optimal tiling. The difference is exaggerated for demonstration purposes.

XSPN-FPGA implements so called *UFloat* operators in Chisel for addition, multiplication and *UFloat*-to-IEEE-754 conversion. They have customizable exponent and mantissa widths and largely work the same as the traditional IEEE-754 format. However, they don't implement rounding, infinity, NaN and don't support negative numbers because they are only dealing with probabilities. We reimplement these operators 1-to-1 in FIRRTL++.

The *UFloat* format is specified by providing the bit lengths of mantissa and exponent. We introduce a new C++ (`UFloatConfig`) struct that contains these in two integer fields. We implement `DenseMapInfo<UFloatConfig>` in order for it to be processable by the unique id algorithm in *FIRRTL++*. We implement a function named `BundleType ufloatType(UFloatConfig cfg)` that creates a FIRRTL bundle type with fields for the exponent and mantissa. Additionally, we provide utility functions to easily bit cast between unsigned integers and *UFloat* bundle types.

Describing the architecture of the *UFloat* operators would be outside the scope of this work. We just want to briefly touch on the multiplication operator as it has implications for our evaluation. The *UFloat* multiplication module works by multiplying the mantissas and adding the exponents. Multiplying the mantissas is done using integer arithmetic but can nonetheless be very heavy on resource usage for large bit widths. Just as the original implementation in XSPN-FPGA, we implement a pipelined integer multiplier. It works by performing multiplication between sub-packages of the two input mantissas and summing up all intermediate results. The efficiency depends on the balance between choosing the right sub-package size and the subsequent adder pipeline depth. The sub-package sizes are chosen such they can be directly mapped to the FPGA's DSP resources. For example, the *VC709* has DSPs that can multiply a 25- and 18-bit integer efficiently. If we imagine aligning the mantissas orthogonally to form a square as seen in figure 6.3, finding and multiplying sub-packages is an optimization problem, where the least number of rectangles required to completely cover the square is the optimal solution. XSPN-FPGA uses a linear programming solver to find the optimal tiling. For simplicity, we resorted to naively tiling the square from top-left to bottom-right. This is important to mention because it can cause an increase in the number of used DSPs.

The *UFloat* adder made use of a pipelined adder design for its mantissa addition. We also implemented a pipelined adder, very similar to the one in XSPN-FPGA.

In the end, FIRRTL++ code looks very similar to Chisel. This made writing the code a straightforward task. However, debugging proved to be much more challenging. The only means of debugging was to use CIRCT's *SystemVerilog* lowering and utilize a testbench with a simulator. We use *CocoTb* together with *Verilator* and *QuestaSim*. What we noticed was that a lot of errors arose from the different ways how bit widths are handled by default in Chisel and FIRRTL. Even though Chisel produces `FIRRTL` code, it does a lot of things in the background such as bit width inference and truncation. One example is that Chisel's addition operator does not perform extension, whereas `firrtl.add` does. What also made debugging difficult was that FIRRTL++ has no way to link the generated FIRRTL IR with the C++ code that generated it. This is because C++ has no straightforward way to perform source-level analysis.

Reimplementing The Datapath

With FIRRTL++ we can take a much more high-level approach. Instead of manually instantiating core dialect primitives and pushing MLIR values around, we can think about the problem in terms of mapping *LoSPN* operations to FIRRTL++ modules. All the necessary interconnecting is mostly handled by FIRRTL++. Our improved implementation with the new operators walks the *LoSPN* IR tree by using MLIR's `walk(...)` function. The `walk(...)` function takes a lambda function as its argument. That lambda function acts as a callback on every operation `walk()` encounters. We use LLVM's `TypeSwitch<...>` to type match and perform the appropriate action. For example, `lo_spn.add` is mapped to FIRRTL++'s `FPAdd` module. With the help of MLIR's `IRMapping` we associate *LoSPN*'s operation's input and output values to the new IO values of our *UFloat* operators. Delays between the FIRRTL++ modules are inserted by using FIRRTL++'s `shiftRegister`. In contrast to our MVP implementation, scheduling has to be done before IR conversion, as the exposure of the underlying FIRRTL IR is a little limited. Because the scheduling library is very generic and handles every kind of operation, scheduling is just performed on the *LoSPN* IR tree. To correctly schedule the datapath, the delays of the hardware modules must be known beforehand.

Reimplementing The Controller

The controller in XSPN-FPGA contains a few major interconnected components. The components are

- `AXI4StreamSender`
- `AXI4StreamReceiver`
- `AXI4StreamMapper`
- `IPECLoadUnit`
- `IPECStoreUnit`
- AXI4-Lite register file
- FSM

How they interplay can is roughly described in figure 5.1. Every component's implementation except for the load and store units heavily depends on the parameters of the datapath: Input variable count, bits per variable, output type (float32 or float64) and the total delay of the scheduled datapath. FIRRTL++ was extended alongside reimplementation to match our needs. This includes types and utilities for AXI4, AXI4-Lite and AXI4-Stream.

Most of the surrounding controller architecture is only necessary to form a compatible PE with *TaPaSCo*. A future alternative approach to using *TaPaSCo* altogether would be to build an integration with ESI. Additionally, our design shall be tested in simulation. In order to have these options, the controller architecture is built in layers. Each of these layers can be optionally omitted.

The innermost layer wraps the SPN datapath. It has N 2 or 8-bit integer inputs and one 32 or 64-bit floating point output. If an input is set, the output is valid a fixed amount of cycles later according to the schedule. We denote the number of cycles as Δt. The innermost layer's job is to provide a handshaking interface to the outside world. Its functionality and components are best understood if we consider the following constraints:

- If the consumer suddenly is unable to consume any more outputs, it must be guaranteed that all current samples in the datapath can be buffered.
- A new input is only accepted if it can be guaranteed that it can be buffered after it is processed by the SPN and not lost.

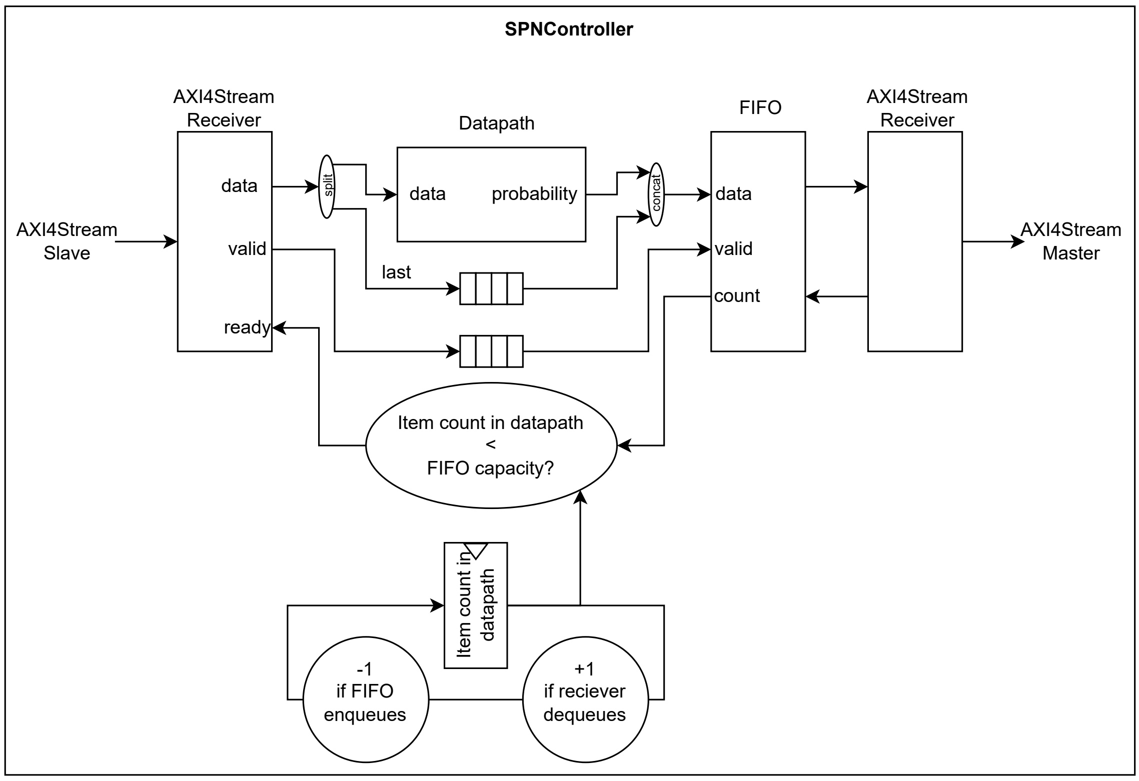

Figure 6.4: Detailed overview of the controller wrapping the datapath. Its job is to provide a handshaking interface to the outside world.

The first constraint implies a FIFO behind the datapath of at least size Δt. The second constraint implies that the input port is only allowed to accept an input if the number of samples currently in the datapath + 1 does not exceed the available slots in the FIFO. Additionally, valid outputs of the SPN are tracked using shift registers with delay Δt. They must be perfectly in sync with the data in the datapath. To combat congestion where the output is not ready for a few cycles at a time we make the FIFO have $2\Delta t$ slots. The `AXIStreamSender` and `AXIStreamReceiver` implement a subset of the AXI4Stream protocol. They mostly act like a simple ready-valid handshake but are fully compatible with Vivado tooling and IPs. The design can be viewed in more detail in figure 6.4

The *SPNController* can be optionally wrapped with the `AXI4StreamMapper`. `AXI4StreamMapper` makes the whole design compatible with *TaPaSCo*. It provides an AXI4 port for interfacing with RAM and an AXI4-Lite port that sits in front of a register file for controlling the PE. The additional AXI4Stream ports are used as endpoints for Xilinx AXI4-Stream Interconnects that perform gearboxing. They adapt the SPN's input and output widths to the data width of the memory port. These IPs are instantiated and connected using TCL scripts. We found that using Xilinx IPs here is much more convenient as they are well-tested and are better able to utilize the device's hardware resources.

`IPECLoadUnit`'s and `IPECStoreUnit`'s implementations only depend on the AXI4 address and data widths. Address widths are typically 32 or 64. Data widths are typically a power of 2 between 8 and 512 (inclusively). So for each unit, we have $2 \cdot |\{8, 16, 32, 64, 128, 256, 512\}| = 14$ possible combinations. This results in 28 implementation source files accompanying the SPNC project, which is a manageable amount. The source files are generated from Chisel in the FIRRTL format. CIRCT provides a FIRRTL code parser that outputs an IR in the FIRRTL dialect which is fully compatible with FIRRTL++. If, for example, the user specifies 32 bits for the address and 128 bits for the data, the units implementation (for example `IPECLoadUnit_32_128.fir` and `IPECStoreUnit_32_128.fir`) is loaded and parsed. The resulting IR is then inserted into the existing IR for the datapath and the surrounding controller. Finally, all the ports are connected. To make this all seamlessly happen, FIRRTL++ contains the class `FIRModule` that can wrap any given `firrtl.module` and insert it into the FIRRTL++ system.

The AXI4-Lite register file follows a very straightforward implementation. As its only job is to save the control values which are set once before the accelerator starts and once after the accelerator finished. This means that we can save on a lot of the complexity of the AXI4-Lite protocol. Our AXI4-Lite register file contains 5 queues, namely for the write/read address, write/read response and the write value. They are connected to the appropriate AXI4-Lite channels. Whenever the write address and write data queue contain an element, it is dequeued, the data is written to the

```
FValue a, b;
...
FValue c = a + b;
```

(a) c is not visible in the simulation.

```
FValue a, b;
...
FValue c = wireInit(a + b, "c");
```

(b) c is visible in the simulation as ”c”.

Figure 6.5: Example of the difference between using the `wireInit` function and not using it.

appropriate register and a write response is enqueued in the write response queue. Reading happens analogously. The user can parameterize the register file with a list of register names and the address offset between them. The names are passed to the creation of `firrtl.regreset` using `OpBuilder` and survive the lowering to *SystemVerilog*. This means that the register names given in FIRRTL++ can be seen in the Value Change Dump (VCD) dump of the simulation by *CocoTb*. This is important for debugging. In contrast to Chisel, the register names have to be given explicitly as strings as they cannot be inferred by C++ automatically.

The FSM which controls the load/store units is so simple that it is simply implemented with when-otherwise constructs. This is completely sufficient in our case.

Limitations

FIRRTL++ was developed alongside our use case. Features and functionality were added as needed. So FIRRTL++'s scope can in no way be compared to Chisel or any other comparable HDL. For example, there is no validation or simulation framework. Everything has to be lowered to *SystemVerilog* beforehand. This is tedious and sometimes difficult because some IR information is lost due to lowering. Chisel has full-fledged passes that validate and type check parts of the circuit. Helpful error messages are generated and link directly to the source location. FIRRTL++ does not have any of those. Invalid circuit constructions are usually detected by one of the type verification steps in CIRCT. However, the generated error messages of those are rarely helpful. FIRRTL++ currently requires more work from the developer to ensure that bit widths and signedness are correct. Chisel is able to infer most of the bit widths automatically. Furthermore, Chisel is very good at transferring information from the source level to the simulation environment, especially variable names. It utilizes a compiler plugin to automatically associate the declared variable name with the signal. C++ does not support source-level reflection. Consequently, if a programmer wants to link a name in the simulation with a certain signal, he or she is required to wrap that signal into a named `firrtl.wire`. An example is shown in figure 6.5.

6.2.2 ESI

The technicalities of ESI are briefly explained in section 4.4.

ESI's model shows its benefits when implementing more complex protocols. We choose AXI4-Stream because it wraps the innermost datapath as seen in figure 6.4.

First, we add another entry to the `ChannelSignaling` enum, namely `AXIStream`. This lets the programmer specify the protocol of an ESI channel like this: `esi.channel<T, AXIStream>`, `T` is the payload type. We introduce a wrapper type named `AXIStreamType<T, N>`, where `T` is the payload type and `N` is the TDATA bus width. A complete AXI4-Stream ESI channel type would look like this: `esi.channel<AXIStreamType<T, N>, AXIStream>`. We choose to introduce this wrapper type for a number of reasons:

- We think the AXI4-Stream TDATA width should be decoupled from the payload type. The reason for this is that the specification dictates that the width must be a multiple of 8, ideally a power of 2. This can lead to some awkward cases where a full 8-bit bus is utilized to carry 1-bit integers. This is a waste of resources. We want to give the user the option to specify the TDATA width explicitly.

- In order for the AXI4-Stream implementation to be compatible with the existing ESI infrastructure, we could not modify how channel protocol specification works. That is, the protocol must still be defined as a single enum value. If we would try to encode the AXI4-Stream TDATA width in the protocol enum, we would have to introduce a new enum value for every possible TDATA width. For an upper limit of 1024 bits, this would imply 127 new enum values.

Together with these two constraints, we think that having a wrapper type is the best solution. It is enforced that the `AXIStream` protocol specification only works with `AXIStreamType`s. Two AXI4-Stream ESI channels are compatible if the payload types and TDATA widths match.

Ignoring all signals of the AXI4-Stream protocol, except for TDATA, TVALID and TREADY, AXI4-Stream presents a valid-ready handshake protocol. We focus on only implementing this subset for now. This lets us utilize a lot of the already implemented operations and lowerings. We introduce 4 new operations: WrapAXIStreamOp, UnwrapAXIStreamOp, AdaptAXIStreamToValidReadyOp and AdaptValidReadyToAXIStreamOp. WrapAXIStreamOp and UnwrapAXIStreamOp work exactly like the previously described vr.wrap and vr.unwrap operations. adapt.vr2axistream and adapt.axistream2vr are used to bridge between AXIStream and valid-ready.

At the point of writing the implementation is very conceptual and far from a full integration. We want to demonstrate its current capabilities by using a

```
hw.module @SPN(%clk: i1, %rst: i1, %inChan: !esi.channel<!esi.axistream<!hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>, i40>, AXIStream>)
    -> (outChan: !esi.channel<!esi.axistream<i64, i64>, AXIStream>) {
  %inVrChan = esi.adapt.axistream2vr %clk, %rst, %inChan : !esi.axistream<!hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>, i40>
  %data, %valid = esi.unwrap.vr %inVrChan, %ready : !hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>
  %a = hw.struct_extract %data["a"] : !hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>
  %concat = hw.array_create %a, %a, %a, %a, %a, %a, %a, %a : i8
  %computed = hw.bitcast %concat : (!hw.array<8 x i8>) -> i64
  %outVrChan, %ready = esi.wrap.vr %computed, %valid : i64
  %outAxisChan = esi.adapt.vr2axistream %clk, %rst, %outVrChan : !esi.axistream<i64, i64>
  hw.output %outAxisChan : !esi.channel<!esi.axistream<i64, i64>, AXIStream>
}
```

(a) A dummy SPN datapath exposing ESI ports instead of manually implemented AXI4-Stream ports. The computation it performs is just a placeholder.

```
module {
  hw.module @SPN(%clk: i1, %rst: i1, %inChan_TVALID: i1, %inChan_TDATA: i40, %outChan_TREADY: i1)
      -> (inChan_TREADY: i1, outChan_TVALID: i1, outChan_TDATA: i64) {
    %chanOutput, %ready = esi.wrap.axistream %inChan_TVALID, %inChan_TDATA :
      i40, !esi.channel<!esi.axistream<!hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>, i40>, AXIStream>
    %0 = esi.adapt.axistream2vr %clk, %rst, %chanOutput : !esi.axistream<!hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>, i40>
    %rawOutput, %valid = esi.unwrap.vr %0, %ready_1 : !hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>
    %a = hw.struct_extract %rawOutput["a"] : !hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>
    %1 = hw.array_create %a, %a, %a, %a, %a, %a, %a, %a : i8
    %2 = hw.bitcast %1 : (!hw.array<8xi8>) -> i64
    %chanOutput_0, %ready_1 = esi.wrap.vr %2, %valid : i64
    %3 = esi.adapt.vr2axistream %clk, %rst, %chanOutput_0 : !esi.axistream<i64, i64>
    %valid_2, %data = esi.unwrap.axistream %3, %outChan_TREADY : !esi.channel<!esi.axistream<i64, i64>, AXIStream>
    hw.output %ready, %valid_2, %data : i1, i1, i64
  }
}
```

(b) The ESI channel ports have been lowered to *HW* ports.

```
module {
  hw.module.extern @ESI_AXIStreamReceiver(%clk: i1, %rst: i1, %TVALID: i1, %TDATA: i40, %data_out_ready: i1)
    -> (TREADY: i1, data_out: i40, data_out_valid: i1) attributes {verilogName = "ESI_AXIStreamReceiver"}
  hw.module.extern @ESI_AXIStreamSender(%clk: i1, %rst: i1, %TREADY: i1, %data_in: i64, %data_in_valid: i1)
    -> (TVALID: i1, TDATA: i64, data_in_ready: i1) attributes {verilogName = "ESI_AXIStreamSender"}
  hw.module @SPN(%clk: i1, %rst: i1, %inChan_TVALID: i1, %inChan_TDATA: i40, %outChan_TREADY: i1)
      -> (inChan_TREADY: i1, outChan_TVALID: i1, outChan_TDATA: i64) {
    %ESI_AXIStreamReceiver.TREADY, %ESI_AXIStreamReceiver.data_out, %ESI_AXIStreamReceiver.data_out_valid =
      hw.instance "ESI_AXIStreamReceiver" @ESI_AXIStreamReceiver(clk: %clk: i1, rst: %rst: i1, TVALID: %inChan_TVALID: i1,
        TDATA: %inChan_TDATA: i40, data_out_ready: %ESI_AXIStreamSender.data_in_ready: i1)
          -> (TREADY: i1, data_out: i40, data_out_valid: i1)
    %0 = hw.bitcast %ESI_AXIStreamReceiver.data_out : (i40) -> !hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>
    %a = hw.struct_extract %0["a"] : !hw.struct<a: i8, b: i8, c: i8, d: i8, e: i8>
    %1 = hw.array_create %a, %a, %a, %a, %a, %a, %a, %a : i8
    %2 = hw.bitcast %1 : (!hw.array<8xi8>) -> i64
    %3 = hw.bitcast %2 : (i64) -> i64
    %ESI_AXIStreamSender.TVALID, %ESI_AXIStreamSender.TDATA, %ESI_AXIStreamSender.data_in_ready =
      hw.instance "ESI_AXIStreamSender" @ESI_AXIStreamSender(clk: %clk: i1, rst: %rst: i1, TREADY: %outChan_TREADY: i1,
        data_in: %3: i64, data_in_valid: %ESI_AXIStreamReceiver. data_out_valid: i1)
          -> (TVALID: i1, TDATA: i64, data_in_ready: i1)
    hw.output %ESI_AXIStreamReceiver.TREADY, %ESI_AXIStreamSender.TVALID, %ESI_AXIStreamSender.TDATA : i1, i1, i64
  }
}
```

(c) The module is completely lowered to core dialects and is ready for *SystemVerilog* generation.

more involved example. 6.6a shows a dummy SPN datapath implementation where that manual AXI4-Stream ports have been replaced by ESI channels. The computation the module performs is just there for placeholder reasons. Notice how `esi.adapt.axistream2vr` and `esi.adapt.vr2axistream` are used to convert the complex AXI4-Stream interface to a more manageable valid-ready interface: These operations have exactly the same roles as the `AXI4StreamReceiver` and `AXI4StreamSender` modules in the classical datapath architecture. `esi.unwrap.vr` and `esi.wrap.vr` implement the handshaking before and after the datapath. In the classical architecture, the valid and ready signals would be bound to certain conditions as in the previously described FIRRTL++ implementation. Figure 6.6b shows the application of the `--lower-esi-ports` pass. It makes sure that the ESI channels facing the outside world get split up into their constituent parts. These parts have to be reassembled as the previous ESI channel type ports were direct inputs to or results from operations that operate on ESI channel types. That's the reason why another `esi.wrap.axistream` and `esi.unwrap.axistream` are introduced. The next lowering step converts the IR to core dialects. This is demonstrated in figure 6.6c. The details of lowering `esi.adapt.axistream2vr` might seem a little bit like cheating. The operation is converted to three sequential operations: `esi.unwrap.axistream`, `hw.instance` of "ESI_AXIStreamReceiver" and `esi.wrap.vr`. This enables the canonicalizer to perform the following optimization: `esi.unwrap.axistream`, `esi.wrap.axistream`, `esi.wrap.vr` and `esi.unwrap.vr` are all be eliminated. Only the "ESI_AXIStreamReceiver" module is left which performs the same function as in the classical architecture. This whole thing happens analogously with `esi.adapt.vr2axistream`. These steps are redundant right now, but come in handy when a more complex subset of the AXI4-Stream protocol is implemented: Compatibility to valid-ready is only possible through the `*adapt*` operations and implementation details are all put in there and don't require modification of any other operation. Finally, everything can be lowered to *SystemVerilog*.

7 Evaluation

7.1 Comparison to XSPN-FPGA

Ultimately, the goal is to replace XSPN-FPGA with SPNC. Naturally, it makes sense to compare our implementation to XSPN-FPGA. In this chapter, we make a quantative comparison of FPGA resource utilization and performance on real hardware. Furthermore, we will briefly discuss usability, the quality of the generated *SystemVerilog* code and synthesis/compile times.

7.1.1 Disclaimer and Environment

XSPN-FPGA by default places AXI4-Stream width converters at both ends of the datapath to convert the memory data bus width (512 on *VC709*) to N or $8N$ bits, depending on whether it is a SPN with binary inputs or not (N is the number of input variables). For some choices of N, this caused an explosion in utilization for the converter and rendered the design unschedulable. For example, *NIPS70* ($N = 70$) could not be synthesized in SPNC and XSPN-FPGA when using the Xilinx AXI4-Stream width converter. We suspect the explosion in utilization is when the greatest common divisor of input and output width is very small. To circumvent this problem we designed a custom width converter that implements just a necessary subset of the AXI4-Stream protocol to get things working. We did this for XSPN-FPGA and SPNC. With the help of FIRRTL++, it was a relatively simple task to get matching implementations. With our custom converter, we could synthesize all networks in both projects except for *NIPS70* in XSPN-FPGA. Therefore, the real-world timing benchmarks for XSPN-FPGA are taken from previous work by Sommer et al. [21]. For the comparison of resource utilization, we resynthesized all XSPN-FPGA networks.

Since the core of our work is a reimplementation of XSPN-FPGA we expect similar resource utilization and target frequencies for the SPNs. Because place-and-route involves stochastic algorithms, fluctuations are expected for identical HDL source code. To get a better understanding of the fluctuations, we synthesized the same HDL source code 10 times for the SPNs *NIPS5*, *NIPS10*, *NIPS20*, *NIPS40* and

SPN	Slice LUTs (mean)	Slice LUTs (var)	BRAM (mean)	BRAM (var)	DSP (mean)	DSP (var)
NIPS5	16.532	0.000096	3.13	0.0	0.56	0.0
NIPS10	17.278	0.000096	3.13	0.0	1.39	0.0
NIPS20	19.018	0.000016	3.16	0.0	6.22	0.0
NIPS40	21.772	0.000016	3.13	0.0	13.56	0.0
NIPS80	28.482	0.000016	3.13	0.0	29.44	0.0

Table 7.1: Resource utilization percentage of XSPN-FPGA for different SPNs.

SPN	E	M
NIPS5	7	24
NIPS10	7	24
NIPS20	8	24
NIPS30	8	26
NIPS40	9	26
NIPS50	9	26
NIPS60	9	26
NIPS70	10	26
NIPS80	10	27

SPN	E	M
ACCIDENTS4000	10	26
BAUDIO4000	10	26
BNETFLIX4000	10	26
MSNBC200	10	26
MSNBC300	10	26
NLTCS200	10	26
PLANTS4000	10	26

Table 7.2: *UFloat* exponent width and mantissa width configuration for each SPN.

NIPS80. The results can be seen in table 7.1. These numbers are only there to give a rough estimate of the expected fluctuations and should not be taken as a solid study. Specialized hardware such as Block RAM (BRAM) and DSP utilization shows no fluctuations. Lookup Tables (LUTs) on the other hand fluctuate, but only very slightly. Therefore, we closely inspect any differences in utilization between XSPN-FPGA and SPNC for a given SPN.

Next up, we accept a Worst Negative Slack (WNS) in our routing of -0.3. This number is specified by *TaPaSCo* and has proven to be acceptable in prior work [21].

Table 7.2 shows the bit widths of the internal *UFloat* used in the SPNs. They are equal for SPN and XSPN-FPGA. They were chosen such that for the given test sets, the relative error was $< 1e^{-6}$. All networks use a 64-bit IEEE floating point output format. The synthesizable networks were benchmarked using the same software interface. Performance was measured on AMD's Virtex 7 FPGA VC709 Connectivity Kit, *VC709* for short. The host CPU is an AMD Ryzen 5 1600X with 16GB of RAM.

7.1.2 Resource Utilization and Performance

A comparison of resource utilization between XSPN-FPGA and SPNC can be seen in figure 7.2. This is rather surprising as we expected resource usage of the different resource types to be more or less identical. Exemplary we took a look into *NIPS60*. What we found is that for each *UFloat* multiplier DSP utilization went from 0.11% down to 0.08%. Both underlying *SystemVerilog* implementations make use of four

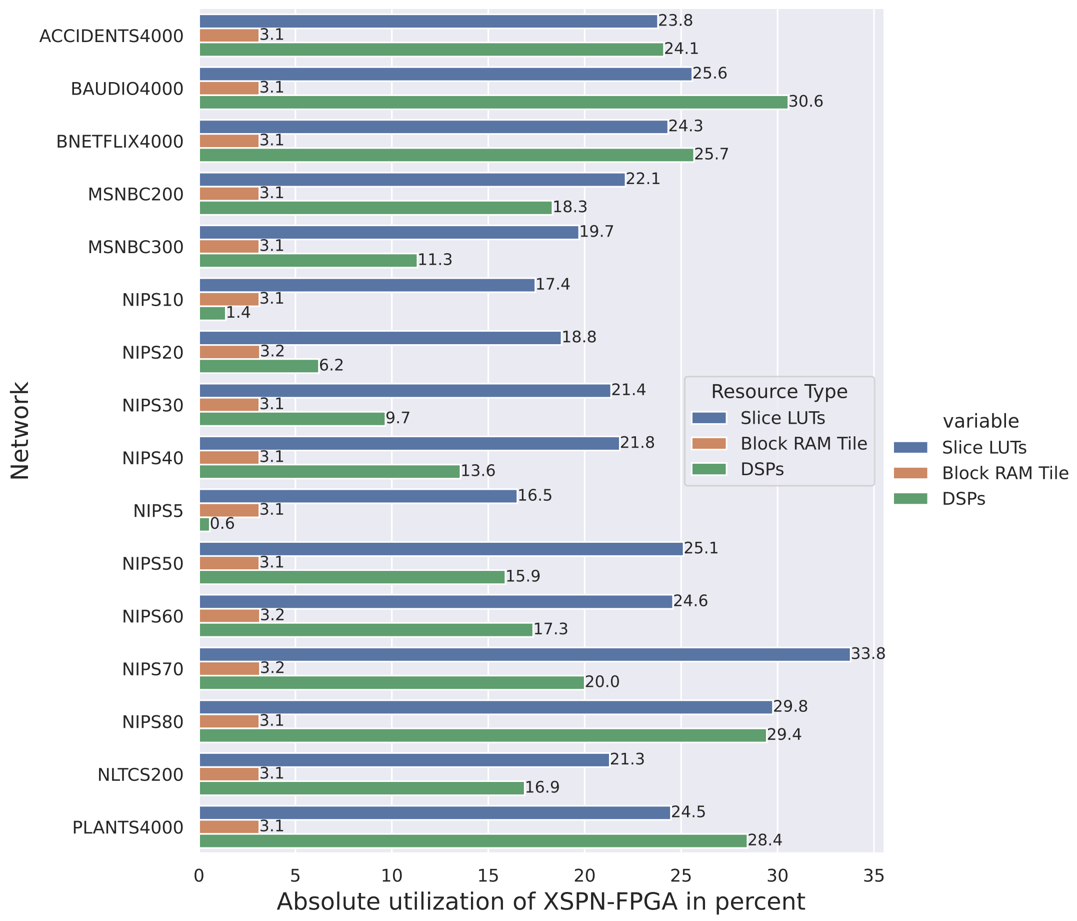

Figure 7.1: Absolute resource utilization of XSPN-FPGA in percent.

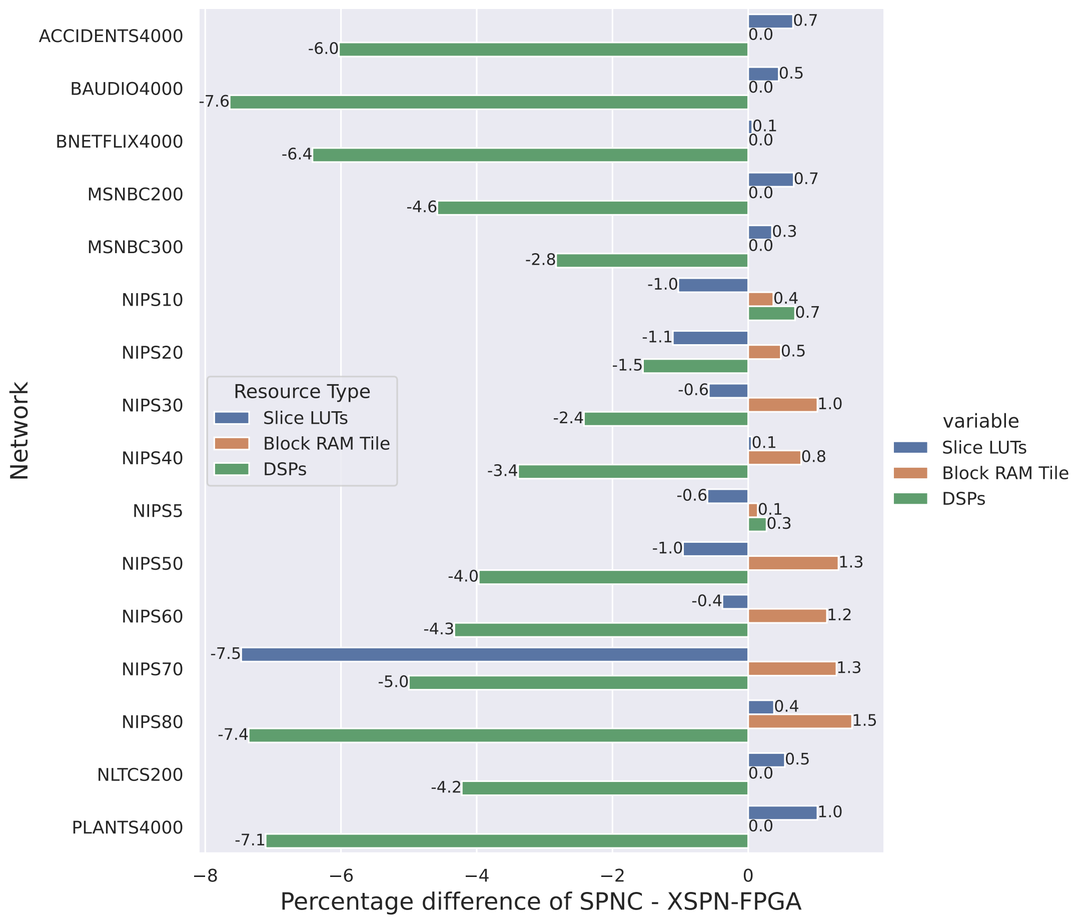

Figure 7.2: Resource utilization of XSPN-FPGA and SPNC for different SPNs.

$*$-operations with bit widths 25×18. Both implementations are functionally identical. The only difference is that XSPN-FPGA has each $*$-operation sitting in a submodule. In contrast, SPNC has the four $*$-operations side by side in a single module. We conclude that Vivado is better able to optimize the $*$-operations in the SPNC implementation as they are in close proximity to each other.

NIPS70 is also an interesting case where LUT utilization went down drastically. It turns out that the LUT utilization by the actual datapath is identical in both architectures, namely 8.23%. We find that the AXI4-Stream converter makes the difference. It utilizes 10.52% and 2.53% in XSPN-FPGA and SPNC respectively. The underlying mechanism of the converter is a ring buffer with random access. The ring buffer is divided up into a number of 8-bit registers such that the total bit width is the smallest power of 2 greater or equal to the larger of both input and output busses. This means, that a fixed set of 8-bit registers has to be able to enqueue and dequeue at the same time. Naturally, this is very heavy on logic. Both implementations map roughly to the same FIRRTL code as the SPNC implementation is an identical port from Chisel to FIRRTL++. The only difference in the *SystemVerilog* code we could spot was that CIRCT's generator produces code with dynamic array indexing. Chisel's *SystemVerilog* reduces the dynamic array indexing to MUX operations. We suspect that deferring this task to Vivado is the reason for lower LUT utilization as Vivado can reduce dynamic array indexing to a more efficient implementation.

Exemplary, we again take a look at *NIPS70* to better understand the BRAM usage. BRAM utilization went from 0.3% to 1.3%. This is quite significant. It turns out that SPNC uses BRAM in the histogram nodes to store the probability values and then uses the input index to query the probability value. XSPN-FPGA on the other hand builds a big nested MUX structure to query the probability value. The probability values are implemented as an array of constants. This trades logic for memory. XSPN-FPGA has a higher logic utilization. What's interesting is that both implementations map the probability array to a FIRRTL vector type. The lowering to *SystemVerilog* is completely dependent on Chisel and CIRCT. Again, CIRCT seems to produce more Vivado-friendly code.

Figure 7.3 shows the performance measured on real hardware. We tested on a *VC709* evaluation board. This is the same device used by prior work [22]. Due to limitations of our custom AXI4-Stream width converter, we measured with $2^5 \times 10^5$ samples. This ensured that the number of input and output frames for the converters, load and store units were integers and no partial transactions had to be made. We outperform this early version of XSPN-FPGA for every SPN. We strongly suspect that this is caused by the overhead of *TaPaSCo* job creation and job destruction. Prior work used 10^5 benchmark samples which are too little to neglect this overhead. Theoretically, there should be no difference in throughput

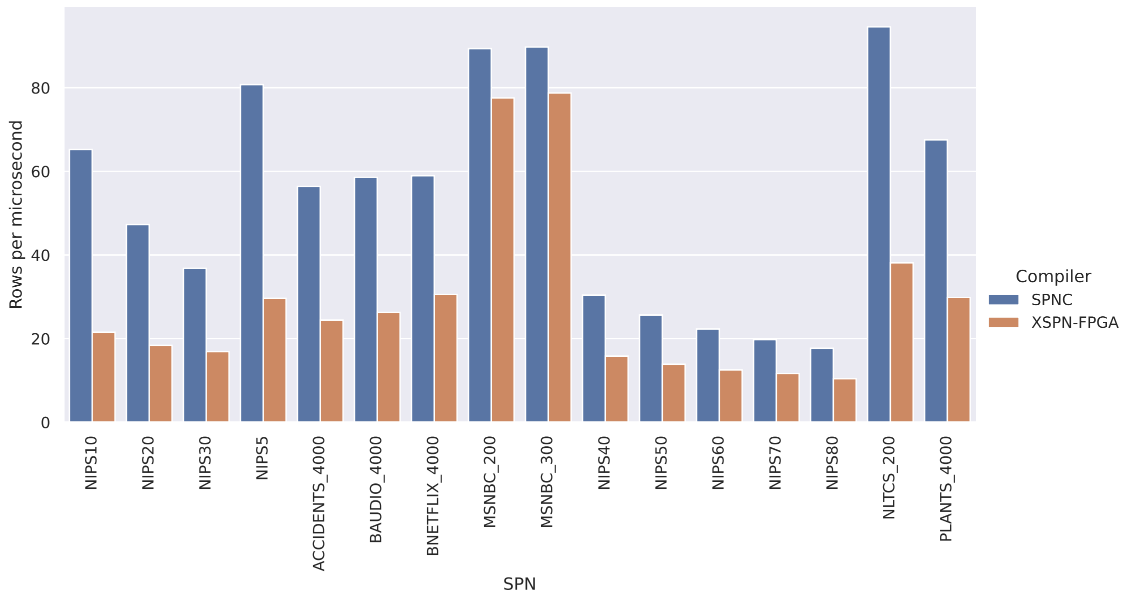

Figure 7.3: Performance of XSPN-FPGA and SPNC for different SPNs.

because both architectures are designed to process 1 sample per cycle and utilize the full 512-bit memory interface.

NIPS70 was not synthesizable at 200MHz on the *VC709* for XSPN-FPGA. We tried using Xiling IPs for bit width conversion and our custom converter implementation. Both failed to meet the timing requirements. The failing path was always part of the bit width converter. However, exactly the same converter implementation synthesized fine for SPNC. Given the previous analysis of differences in code-to-resource mapping, we strongly suspect that the generated code by XSPN-FPGA is suboptimal for Vivado's FPGA synthesis.

Initially, the analysis of the resource utilization and performance of both compilation flows was meant to confirm our thesis, that both offer a functionally equivalent toolflow and are expected to perform equally. This is clearly not the case. In certain situations the generated *SystemVerilog* code can be very different and even, if functionally equivalent, map to different technologies on the fabric. We conclude that CIRCT's *SystemVerilog* code generator is better suited for Vivado's FPGA synthesis.

7.1.3 Compilation and Synthesis Performance

In the previous section, we established that CIRCT's code generation is superior. In the worst cases, bad code renders the design unsythesizable. It's plausible that this could impact synthesis time. First, we look at the packaging times as seen in

SPN	XSPN-FPGA	SPNC
ACCIDENTS4000	47.14s	01m:08.74s
BAUDIO4000	44.91s	01m:09.95s
BNETFLIX4000	42.25s	01m:09.56s
MSNBC200	46.13s	01m:10.12s
MSNBC300	46.71s	01m:06.65s
NIPS10	51.10s	01m:01.94s
NIPS20	46.15s	01m:03.41s
NIPS30	44.35s	01m:05.88s
NIPS40	48.00s	01m:07.92s
NIPS5	48.03s	01m:01.95s
NIPS50	59.59s	01m:09.40s
NIPS60	48.30s	01m:10.08s
NIPS70	48.24s	01m:11.53s
NIPS80	55.05s	01m:16.13s
NLTCS200	48.41s	01m:07.71s
PLANTS4000	46.26s	01m:11.37s

Table 7.3: Packaging times for XSPN-FPGA and SPNC.

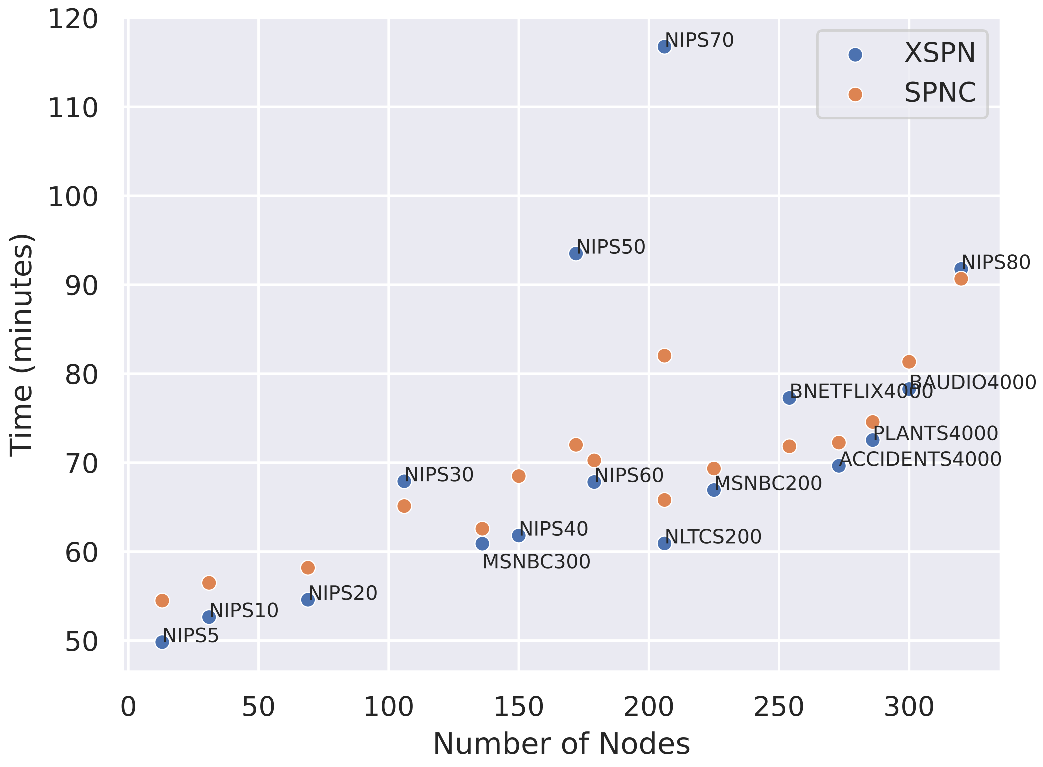

Figure 7.4: Synthesis times for XSPN-FPGA and SPNC.

figure 7.3. The packing time is the time required for producing all the necessary *SystemVerilog* files and TCL scripts for further synthesis. This is an important metric to consider when performing simulation. Long packaging times prolong the cycle of simulation and implementation or bug fixing. Packaging also includes executing a Vivado TCL script for instantiating some wrapping IPs. Unfortunately, the packaging times in SPNC are quite a lot longer. We measured the elapsed time of the command with the Unix `time` command. Real and user time seemed a bit off as SPNC *sys* time (i.e. the time spent in the kernel) was quite high. So we suspect that the timing differences are due to some IO and multithreading issues. This is further undermined by the fact that the packaging time for SPNC does only scale little with network size.

Figure 7.4 shows the synthesis times for both compilers. The synthesis times are mostly in the same ballpark of plus or minus 5 minutes. It is noteworthy that most times, SPNC has a slower synthesis process. An outlier is the *NIPS70* network. It was not synthesizable in XSPN-FPGA. Vivado had to search through a bigger solution space until it finally could determine that the design was unschedulable. We conclude that synthesis times are not significantly negatively impacted in SPNC. Without further research into how Vivado's synthesis process internally works, any deeper conclusion would be pure speculation. However, we felt that this information is worth sharing.

8 Conclusion and Future Outlook

8.1 Conclusion

We built a functioning end-to-end flow for mapping SPN descriptions to synthesizable *SystemVerilog* code. It is fully integrated with SPNC together with a user-friendly frontend, a DSL for constructing complex circuits in raw CIRCT and tight integration with *CocoTb* for simulation and testing. This was mostly an effort by one person. This would not have been possible without CIRCT and the extensive work and tooling around FIRRTL and especially the lowering steps from that dialect to the core dialects.

Even though CIRCT helped us greatly improve productivity, it was not without its caveats. The development of the FIRRTL++ DSL was a necessity to solve the problem of there not being a framework or dialect that fits our needs.

In the grand scheme of things generating MLIR code for software and hardware targets look very different. Software targets have the whole LLVM stack beneath them that seamlessly integrates with MLIR. Target-specific "primitives" such as machine instructions for memory management, stack frames, function calls, returns codes, struct layouts and so forth are all engrained in the LLVM backend. Usually, a programmer doesn't have to be concerned with implementing these things themselves. For hardware targets the story is a little different. It often does require decisions by the programmer to choose logic for implementing control registers, communication ports, FIFOs and so forth. In our experience CIRCT offers dialects/frameworks with high-level functional abstractions and dialects with low-level RTL abstractions. There is no good way to describe specific implementations of the architecture. In our case with the SPN datapath, we need tight control over how the arithmetic operators are implemented and connected as the implementation's high throughput relies on tight scheduling and no handshake protocols.

One could argue that implementing hardware primitives is not the job of CIRCT but high-level HDLs. We think otherwise. The development of *PyCDE* suggests that there is indeed a need for such frontends. Additionally, a key goal of MLIR and CIRCT is enabling a true cross-platform toolflow, including CPUs, GPUs and FPGAs. This means that domain-specific IRs, *HiSPN* and *LoSPN* for example,

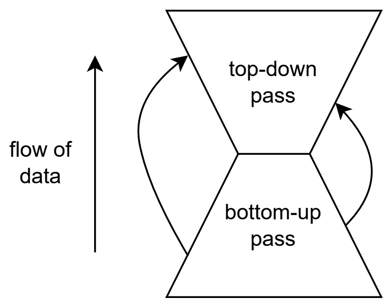

Figure 8.1: Illustration of the structure of a pipelined MPE design. The two trees are connected at the roots with skip edges.

must be mappable to a hardware implementation, which requires hardware primitives inside CIRCT.

In the end, we think that at the time of writing CIRCT is missing intermediate dialects or frontends for dialects to describe hardware in a HDL kind of way. The upcoming integrations into the CIRCT project of *Calyx* and *Moore* seem promising.

8.2 Future Work

As this work is mainly laying the foundation for future work, we want to take a deep dive into a variety of topics that we think are interesting and could be addressed in the future. Some of them are direct improvements of some of the work we did, some are sourced from work on XSPN-FPGA and some are unrelated to any published work.

8.2.1 Marginal Inference and MPE

Marginal inference and MPE are already implemented in XSPN-FPGA. Marginal inference is a trivial extension of the SPN bottom-up datapath. Every histogram must be extended by an extra bucket that contains the 1 probability. Additionally, the domain size for every variable is extended by one more possible index.

MPE is a bit more involved. The mathematical definition is found in section 2.1. For a given subset of variables $\hat{\mathbf{X}}$ and its assignment $\hat{\mathbf{x}}$, $MPE(\hat{\mathbf{x}})$ finds the assignment of variables $\mathbf{X} - \hat{\mathbf{X}}$ that maximizes the probability distribution. This is done by tracing the SPN graph from top to bottom:

- If a node is a sum node, then the subtree of the maximum child is entered.
- If a node is a product node, then all subtrees are entered.

- If a node is a histogram node and the associated variable is in $\hat{\mathbf{X}}$, then that variable is assigned the index with the maximum probability in that histogram.

Without going into too much detail about how this is implemented in XSPN-FPGA, it is important to know that this algorithm induces a reversed SPN graph. The default bottom-up pass is enhanced such that each sum node computes which input was the greater one. This single bit of information is passed to the top-down pass via skip edges. This is illustrated in figure 8.1. Now, having the reversed SPN graph on top of the original SPN graph breaks the tree property and introduces a lot of complexity because now a data structure for working with general graphs is required. XSPN-FPGA is not designed for that and getting it to work properly with scheduling and hardware instantiation was quite difficult. The benefit of CIRCT would be that it contains lots of utility for working on IRs which are general graphs that contain loops, self-references and where nodes can have multiple parents and children.

Ideally, a MPE implementation in SPN would look something like this: *HiSPN* and *LoSPN* are extended with operations that model the MPE computation. For *HiSPN* we would just add another query type operation. *LoSPN*'s job would be to explicitly model the MPE computation with a set of new operators. Speaking from experience with XSPN-FPGA, mapping the actual operators to hardware modules is trivial. The arithmetic operators are mostly the same as in the bottom-up pass without MPE support. The sum nodes in the top-down pass only propagate a single-bit signal depending on a single-bit input signal. Product nodes are no-ops.

We think, that implementing MPE in SPNC is a lot simpler than in XSPN-FPGA because generic IR modeling is an inherent part of MLIR.

8.2.2 Deviating from the simple Bottom-up Architecture

In this work SPNs get mapped to a single-pass pipeline. This allows for high throughput, but because of resource limitations, the acceptable size of a SPN is limited. This problem was explorer in prior work [19]. Their proposal is an implementation where portions of the SPN are computed in batches. The intermediate results are saved, the general SPN datapath is reprogrammed to represent a different portion and the previously saved results are inputs for the next computation. The schedule for reprogramming, loading, computing and saving is determined by a specialized compiler.

SPNC already features a very similar graph partitioning approach to circumvent resource limitations in CPU and GPU for very large networks. SPNC essentially performs the same logical steps just for software. This could be leveraged when targeting FPGA targets: Instead of mapping *LoSPN* IR to FIRRTL, a general, reprogrammable SPN circuit with memory and crossbars is implemented using FIRRTL++. The SPN is partitioned into subnetworks using the already existing functionality. From these partitioned subnetworks, a schedule can be derived just as in the software case.

8.2.3 Using Calyx

We decided against using *Calyx* because we didn't want to drastically deviate from XSPN-FPGA's datapath design to stay comparable. However, if you view the nodes in a SPN as a set of tasks with an induced dependency graph then *Calyx* execution model certainly fits. We believe that this is a promising approach for future work. It would be interesting to see how trading throughput with resource usage would affect overall performance. This could enable very large networks to be brought to the FPGA. This is a limitation of XSPN-FPGA. As *Calyx* collects synthesis resource estimates by interfacing with Xilinx tools this could enable the developer to make informed decisions about the tradeoff between throughput and resource usage.

8.2.4 Automatic Scheduling

The datapath in SPNC is scheduled using CIRCT's scheduling infrastructure [16]. It requires providing the latency for each operator. For now, this works by providing a lambda function `getDelayAndType : Operation * → std::tuple<int, std::string>` to our scheduling problem. It maps every *HiSPN* operator to its latency and its type name such as 'mult', 'add' and so on. The type name is required by the scheduler to tag the generic (`Operation *`)s with the correct latency. The latency must be computed apriori for each operator type and depends on the concrete implementation. In our case, the *UFloat* operators are implemented in FIRRTL by using FIRRTL++ in a pipelined fashion. We were required to manually reason about the latencies of the internal signals. In our experience, this work is very error-prone and we were required to fix a lot of 1-cylce-off bugs. We believe that this problem can be easily solved. FIRRTL is a RTL dialect where N cycle delays of a signal can only come from register and memory operations. We think it would be interesting to see an implementation of algorithms that can trace the propagation delay of signals through the FIRRTL design. What we have in mind for example is a function that can be called on a FIRRTL module that computes the latency between the input and output signals, or a way of settings constraints on the input signals to an operation that must reach it in the same cycle. This would greatly reduce errors and make scheduling more robust.

8.2.5 Work on ESI

We think ESI has a lot of potential not just in the context of this work. Seeing proper implementations of complex protocols would be interesting. We briefly touched on a prototype for a subset of AXI4-Stream and concluded that a lot of considerations have to be made before adding concrete types or operations. Furthermore, protocols in the AXI4 family, for example, require quite some logic and state management. Therefore, complex implementations have to be written in a HDLs such as *SystemVerilog* (or FIRRTL++) and must be integrated with ESI. This is very cumbersome work but we think the payoff is worth it.

List of Figures

2.1 A BN, a SPN and their joint probability distributions. 6
2.2 Example of MLIR code. 10

3.1 Different possible lowering pipelines for PyTorch models. Source: [13] 16

4.2 Examples of the compilation pipeline steps executed when targeting x86 CPUs without vectorization. 20

5.1 Controller architecture overview: The green parts are generated by XSPN-FPGA; The blue part is the previously generated datapath. . . 26

6.1 Example of implementing the + operator for `FValue`s. 35
6.2 Example of constructing a complex expression with and without operator overloading. 35
6.3 Matching sub-packages of the two input mantissas for multiplication with DSPs. The left one is our naive implementation; the right one is a more optimal tiling. The difference is exaggerated for demonstration purposes. 37
6.4 Detailed overview of the controller wrapping the datapath. Its job is to provide a handshaking interface to the outside world. 40
6.5 Example of the difference between using the `wireInit` function and not using it. 42

7.1 Absolute resource utilization of XSPN-FPGA in percent. 49
7.2 Resource utilization of XSPN-FPGA and SPNC for different SPNs. . 50
7.3 Performance of XSPN-FPGA and SPNC for different SPNs. 52
7.4 Synthesis times for XSPN-FPGA and SPNC. 53

8.1 Illustration of the structure of a pipelined MPE design. The two trees are connected at the roots with skip edges. 56

List of Tables

7.1 Resource utilization percentage of XSPN-FPGA for different SPNs. . 48
7.2 *UFloat* exponent width and mantissa width configuration for each SPN. 48
7.3 Packaging times for XSPN-FPGA and SPNC. 53

List of Acronyms

FPGA	Field-Programmable Gate Array
HLS	High-Level Synthesis
SPN	Sum-Product Network
BN	Bayesian Network
MRF	Markov Random Field
DAG	Directed Acyclic Graph
LLVM	Low Level Virtual Machine
MLIR	Multi-Level Intermediate Representation
CIRCT	Circuit IR Compilers and Tools
RTL	Register Transfer Level
API	Application Programming Interface
IR	Intermediate Representation
SSA	Static Single Assignment
ESI	Elastic Silicon Interconnect
JVM	Java Virtual Machine
FIRRTL	Flexible Intermediate Representation for RTL
SPNC	Sum-Product Network Compiler
FIFO	First-In-First-Out
IP	Intellectual Property
PE	Processing Element
FSM	Finite State Machine
FIRRTL++	FIRRTL++ Library
RAM	Random Access Memory
DSL	Domain Specific Language

HDL	Hardware Description Language
XSPN-FPGA	XSPN-FPGA Project
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CLI	Command Line Interface
WNS	Worst Negative Slack
RTL	Register Transfer Level
DSP	Digital Signal Processing
LUT	Lookup Table
LP	Linear Program
AST	Abstract Syntax Tree
ESA	Embedded Systems and Applications
ASIC	Application-Specific Integrated Circuit
ML	Machine Learning
AC	Arithmetic Circuit
TCL	Tool Command Language
MPE	Maximum Probability Explanation
ODS	Operation Defining Specification
MVP	Minimum Viable Product
PCIe	Peripheral Component Interconnect Express
DUT	Device Under Test
CRTP	Curiously Recurring Template Pattern
IO	Input/Output
STL	Standard Template Library
VCD	Value Change Dump
LLHD	Low-Level Hardware Description
BRAM	Block RAM
PGM	Probabilistic Graphical Model
NN	Neural Network

Bibliography

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221.

[2] T. S. Chris Lattner. *CGO 2020 Keynote: MLIR Compiler Infrastructure.*

[3] J. Demme. *Elastic Silicon Interconnects: Abstracting Communication in Accelerator Design.* 2021. arXiv: `2111.06584 [cs.AR]`.

[4] F. de Dinechin. "Reflections on 10 years of FloPoCo". In: *26th IEEE Symposium of Computer Arithmetic (ARITH-26).* June 2019.

[5] "IEEE Standard for Binary Floating-Point Arithmetic". In: *ANSI/IEEE Std 754-1985* (1985), pp. 1–20.

[6] K. Jaic and M. C. Smith. "Enhancing Hardware Design Flows with MyHDL". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, 28–31.

[7] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning.* The MIT Press, 2009.

[8] J. Korinth, J. Hofmann, C. Heinz, and A. Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems". In: *International Symposium on Applied Reconfigurable Computing (ARC).* 2019.

[9] H. Kruppe, L. Sommer, L. Weber, J. Oppermann, C. Axenie, and A. Koch. “Efficient Operator Sharing Modulo Scheduling for Sum-Product Network Inference on FPGAs”. In: *Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS).* Springer International Publishing, 2021.

[10] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization.* CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, p. 75.

[11] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. *MLIR: A Compiler Infrastructure for the End of Moore’s Law.* 2020. arXiv: `2002.11054 [cs.PL]`.

[12] P. S. Li, A. M. Izraelevitz, and J. Bachrach. *Specification for the FIRRTL Language.* Tech. rep. UCB/EECS-2016-9. EECS Department, University of California, Berkeley, 2016.

[13] M. B. P. Mike Urbach. “HLS from PyTorch to System Verilog with MLIR and CIRCT”. In.

[14] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang. *Replication Package for Article: Predictable Accelerator Design with Time-Sensitive Affine types.* 2020.

[15] R. Nigam, S. Thomas, Z. Li, and A. Sampson. “A Compiler Infrastructure for Accelerator Generators”. In: *CoRR* abs/2102.09713 (2021). arXiv: `2102.09713`.

[16] J. Oppermann, M. Urbach, and J. Demme. “How to Make Hardware with Maths: An Introduction to CIRCT’s Scheduling Infrastructure”. In: *2022 European LLVM Developers’ Meeting (EuroLLVM).* 2022.

[17] I. París, R. Sánchez-Cauce, and F. J. Díez. “Sum-product networks: A survey”. In: *CoRR* abs/2004.01167 (2020). arXiv: `2004.01167`.

[18] F. Schuiki, A. Kurth, T. Grosser, and L. Benini. “LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2020. London, UK: Association for Computing Machinery, 2020, 258–271.

[19] N. Shah, L. I. G. Olascoaga, W. Meert, and M. Verhelst. “Acceleration of probabilistic reasoning through custom processor architecture”. In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2020, pp. 322–325.

[20] L. Sommer, C. Axenie, and A. Koch. “SPNC: An Open-Source MLIR-Based Compiler for Fast Sum-Product Network Inference on CPUs and GPUs”. In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 2022.

[21] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch. “Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-based Accelerators”. In: *IEEE International Conference on Computer Design (ICCD).* IEEE. 2018.

[22] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch. “Automatic Synthesis of FPGA-based Accelerators for the Sum-Product Network Inference Problem”. In: *ICML 2018 Workshop on Tractable Probabilistic Models (TPM).* ICML. 2018.

[23] L. Sommer, L. Weber, M. Kumm, and A. Koch. “Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs”. In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 2020, pp. 1–10.

[24] L. Weber, L. Sommer, J. Oppermann, A. Molina, K. Kersting, and A. Koch. “Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs”. In: *International Conference on Field-Programmable Technology (FPT).* 2019.