

École Polytechnique Fédérale de Lausanne

A Dynamically Scheduled HLS Flow in MLIR

by Morten Borup Petersen

Master Thesis

Prof. Paolo Ienne - EPFL LAP
Thesis Advisor

Stephen Neuendorffer - Xilinx Research
Thesis Supervisor

EPFL IC LAP
INF 136 (Bâtiment INF)
CH-1015 Lausanne

January 28, 2022

Acknowledgments

Thank you to Paolo Ienne, Lana Josipović, and Mirjana Stojilović for allowing me to be both a student as well as a collaborator of yours during my time at EPFL.

Thank you to Mike Urbach for the countless code reviews and discussions that we have shared to get HLS in CIRCT where it is today.

And finally, thank you to Stephen Neuendorffer for being a great teacher and mentor, not only in the context of this thesis, but also with respect to the many conversations that we have shared, on work, life, career, and how to find ones' path through it all.

Ringsted, Denmark, January 28, 2022 Morten Borup Petersen

Abstract

In *High-Level Synthesis* (HLS), we consider abstractions that span from software to hardware and target heterogeneous architectures. Therefore, managing the complexity introduced by this is key to implementing good, maintainable, and extendible HLS compilers. Traditionally, HLS flows have been built on top of software compilation infrastructure such as LLVM, with hardware aspects of the flow existing peripherally to the core of the compiler. Through this work, we aim to show that MLIR, a compiler infrastructure with a focus on domain-specific *intermediate representations* (IR), is a better infrastructure for HLS compilers. Using MLIR, we define HLS and hardware abstractions as first-class citizens of the compiler, simplifying analysis, transformations, and optimization. To demonstrate this, we present a C-to-RTL, dynamically scheduled HLS flow. We find that our flow generates circuits comparable to those of an equivalent LLVM-based HLS compiler. Notably, we achieve this while lacking key optimization passes typically found in HLS compilers and through the use of an experimental front-end. To this end, we show that significant improvements in the generated RTL are but low-hanging fruit, requiring engineering effort to attain. We believe that our flow is more modular and more extendible than comparable open-source HLS compilers and is thus a good candidate as a basis for future research. Apart from the core HLS flow, we provide MLIR-based tooling for C-to-RTL cosimulation and visual debugging, with the ultimate goal of building an MLIR-based HLS infrastructure that will drive innovation in the field.

Contents

Chapter 1

Introduction

As Moore's law is tapering off, and with the breakdown of Dennard scaling, the computing landscape has transitioned towards ever more heterogeneity over the past decade. GPGPU and domain-specific accelerators in the AI space have shown that specialization is the most promising path to satisfy the increasing demands for computational power. However, increased specialization inevitably implies a divergence from general-purpose programming models—the programming models for which traditional CPU-targeting compilers have been designed. From the compiler's view, the modern computing landscape is broad, both in terms of the inputs we consider and the targets to which we compile. A notion that made general purpose CPU-targeting compiler infrastructures great was the idea of a shared intermediate representation, wherein different input languages and target architectures could all benefit from a shared set of target-independent optimizations. However, a one-representation-fits-all approach has become a limiting factor for compilation in the modern heterogeneous computational landscape.

Hardware is hard - to create high-quality digital designs requires intimate knowledge of both the fundamentals of the field as well as device-specific capabilities. Furthermore, with a need to consider many design points in design exploration, manually describing our accelerators in RTL-level *hardware description languages* (HDLs) can quickly become prohibitively expensive. As a solution to this, *High-Level Synthesis* (HLS) seeks to enable the generation of hardware designs directly from a software description, and by doing so, reduce development time, ensure correctness, and allow software-native designers to perform hardware design.

HLS tools will often be based on software compiler infrastructure. While this design has allowed tools to leverage decades of work into high-performance software compilers capabilities, using such infrastructure for hardware construction has inherent challenges in managing the complexity and changes in levels of abstraction involved in an HLS flow. So as with hardware, it is now time to specialize the compiler itself—without losing what made traditional CPU compiler infrastructure great. Recently, the MLIR (*Multi-Level Intermediate Representation*) project has become a cornerstone in the heterogeneous compilation space. MLIR is an LLVM project that builds on the lessons learned from the past decades of research and development into the LLVM compiler infrastructure. The goal of MLIR is to provide an infrastructure that makes it cheap to define new domain-specific *intermediate representations* (IRs),

transform them, and mix and match them alongside other IRs. The use of domain-specific IRs is observed in many existing HLS flows [31, 37, 52]. Naturally, the motivation for introducing an IR in HLS is to manage the complexity of going from sequential software to parallel hardware, while having a representation that facilitates transformation and optimization. At a high level, HLS can be considered not just as the process of taking an arbitrary sequential program and turning it into a hardware description. Instead, we can consider it a flow that maps software onto hardware *units*, whether being programmable logic—which itself can use many different models of computation—, hardened IPs, or CPU execution. To facilitate such mappings, program transformations are needed at both high levels of abstraction, such as loop tiling, or low levels, when converting from branching control flow to a *finite state machine* (FSM) model. Many such transformations care heavily about the structure of the code. However, in traditional IRs such as LLVM IRs, the high-level structure is quickly lost, making complex HLS transformations difficult at best, impossible at worst. By using multiple levels of abstractions through MLIR, we can clearly define *where* and *how* we perform HLS transformations and optimizations. All this is under the consideration of design constraints such as power, area, throughput, and latency requirements. Due to the vast range of abstractions and constraints to be considered, HLS tools must be built using infrastructure that lessens the burden of working with, and lowering through, the different levels of abstraction.

That is the main theory of this work—we believe that approaching HLS with a primary focus on the abstractions used is a better way of both researching and engineering HLS tools. It is then these abstractions that guide our transformations—transformations make an abstraction powerful and meaningful, and an abstraction is only as good as the transformations it enables.

This work presents an end-to-end MLIR-based HLS flow. The core of which is based on the CIRCT project, an LLVM incubator project that seeks to use MLIR as the basis for a hardware compilation infrastructure. Through CIRCT, we can leverage existing infrastructure to lower a high-level hardware IR into synthesizable HDL. Since HLS is a vast field, we do not seek to solve "HLS in MLIR" but rather show one possible path from software to hardware. In doing so, this work will mainly focus on dynamically scheduled HLS instead of statically scheduled HLS.

The preliminary building blocks for supporting a dynamically scheduled HLS flow in CIRCT existed prior to this project, which has been leveraged extensively through this work. These include definitions of the core abstractions to be used, as well as work on lowering from software to our domain-specific IR, and from the latter down to hardware. In this project, we face the challenge of composing and integrating existing tooling both within CIRCT and outside, as well as the development of new tooling to allow for the synthesis and simulation of C programs to a SystemVerilog representation.

To show that the proposed flow is viable for both future research and engineering efforts, significant work has gone into the creation of an ecosystem supporting the HLS compiler itself. We provide tools for cosimulating C testbenches with RTL simulations of kernels generated by the compiler, a system for visually debugging RTL simulations at the level of a domain-specific IR, and a driver for composing all tools in the flow. Careful consideration has gone into the design of these tools to ensure reproducibility and general applicability for this project and future projects to come.

In evaluating the quality of our flow, we compare both qualitatively and quantitatively against a

comparable compiler, Dynamatic [31]. In this, we compare at the front-end, dataflow IR and hardware level to identify the effects of either flow at these levels of abstraction. To evaluate the quality of the emitted hardware, we compare resource consumption between designs generated by the two systems.

Finally, we present a new model for task pipelining in dataflow circuits. The compiler which this flow is modeled after, Dynamatic, assumes no task pipelining. This significantly limits performance in real world programs that do not have a structure of a single nested loop. Through our model, we show that a significant increase in kernel initiation interval and temporal utilization can be achieved. Furthermore, this solution is compatible with our proposed cosimulation infrastructure such that a sequentially described high-level testbench can exercise the decoupled interface of a task-pipelineable accelerator.

This thesis is structured as follows:
In chapter 2, an introduction to the topic of HLS and an overview of the current state-of-the-art is given. Here, we also introduce MLIR and the CIRCT project, the building blocks of the proposed flow. In chapter 3 we go through the end-to-end dynamically scheduled HLS flow, presenting the choice of front-end, description of the dataflow IR, dataflow lowering, and finally how dataflow circuits are lowered to hardware. In chapter 4 we describe the importance of cosimulation in HLS and our solution to an MLIR-based approach for cosimulating high-level testbenches with RTL simulations of generated hardware. Furthermore, we present HSdbg, a tool for visually debugging RTL simulations of handshake circuits. In chapter 5 we compare our flow with Dynamatic, as well as a qualitative analysis on the applicability of MLIR to HLS compilers. In chapter 6 we present a new model for the generation of dataflow circuits safe for task pipelining. In chapter 7 we conclude on the work, as well as provide possible directions for future work. In Appendix, we provide a comprehensive guide to our driver tool, `hlstool`, as well as a guide to reproducing the experiments performed through this work.

The output of this work is available in the CIRCT repository[1] as well as the CIRCT-HLS repository[2]. CIRCT-HLS is a separate repository not managed by the LLVM project, which has been used to capture this work's integration and testing aspects. We expect that a substantial amount, if not all, of CIRCT-HLS will eventually merge into CIRCT.

[1] `https://github.com/llvm/circt`
[2] `https://github.com/circt-hls/circt-hls`

Chapter 2

Background

High-level synthesis is an interdisciplinary field, considering everything from language design and compiler construction to hardware design—basic knowledge within these fields is assumed. To introduce the reader to the background relevant for this thesis, we will first provide a general introduction to the field of high-level synthesis. Here, we introduce the motivations for HLS, the concepts of statically and dynamically scheduled HLS, and a survey of existing HLS infrastructure. Then we describe MLIR, the compilation infrastructure upon which this work builds. Here, we introduce its motivations as well as an overview of its concept of *dialects*. Finally, we describe CIRCT, an MLIR-based hardware compilation infrastructure, which this work is part of.

2.1 High-Level Synthesis

The need for high-level design methods has been seen in both software and hardware development since the inception of the fields. In software, the use of programming languages at ever-higher levels of abstraction has been a natural evolution to both increase productivity and the capabilities available to the programmer. Likewise, when considering digital hardware, transistor-level design quickly became unfeasible as Moore's law started to scale. As a result, hardware description languages were developed to raise the level of abstraction for a hardware designer to the RTL level. However, contrary to software, the hardware world has been slow to move beyond this step. This can be attributed to the difficulties of needing cycle-accurate behaviour. While software can rely on operating systems and *application binary interfaces* (ABIs) to handle resources and inter-procedural control flow, hardware must have intimate knowledge of the often latency-sensitive interfaces and device-specific behavior which digital circuits are designed under.

This poses a problem in our current age; efficient RTL level design requires skills orthogonal to those of software engineering. With the ever-growing need for specialization and domain-specific accelerators, we seek to close the gap between the behavioral specification of an accelerator and its *structural* definition. Thus, through the proliferation of reconfigurable hardware, allowing skilled software designers to target hardware devices, without intimate knowledge of RTL level design.

High-level synthesis (HLS) is one solution to this problem. In short, HLS transforms a behavioral software program into parallel hardware with equivalent semantics, while leveraging hard- and soft IP, *system-on-chip* (SoC) peripherals, and programmable logic. HLS is a vast field, spanning digital design, compiler engineering, chip design, and everything in between. While the concepts of HLS have been researched since the 1970s [19] and the core concepts of the field well established since the 1990s [23], it was not until the proliferation of FPGAs that use of the technology started to take off beyond research environments [15, 34]. Today, HLS tools have become vital given their enablement of rapid development and deployment of new algorithms on reprogrammable hardware.

The classical approach to HLS is through *statically scheduled HLS*. In this model, a program is initially partitioned into substructures, which are transformed into a hardware representation through scheduling and allocation. Scheduling partitions the operations of a design with respect to time, and allocation with respect to hardware resources. In this process, allocation achieves spatial parallelism by deciding the amount of functional units in a design, which enables the scheduling of operations in parallel - these problems are thus intertwined. HLS algorithms are typically performed under latency and area constraints, with the tradeoff usually being higher performance (latency, f_{max}, throughput) implying greater resource usage. Any given design will be assigned a *schedule*, guiding the activation and interconnection of the allocated functional units during kernel invocation. In *statically* scheduled HLS, this schedule is fixed at compile time, and under the consideration of the allocated hardware and input program, accounts for the resolution of possible structural and data hazards during execution. A static schedule and a set of allocated hardware units can then be transformed into a *finite state machine + datapath* (FSMD) model. In this model, the functional and storage units as well as their interconnections comprise the datapath, whereas the schedule drives the definition of the FSM controller. This FSM

controller will interact with components in the datapath to drive runtime routing and component enablement. Considering kernel performance, speedup through parallelism can be achieved in two ways - spatial parallelism, as described above, and temporal parallelism. A key factor to designing high-performance circuits is to maximize temporal use of the allocated hardware resources, which is done through the process of pipelining. For a given kernel, pipelining can be achieved both for structures within the kernel, such as loop pipelining, and across kernel calls, task pipelining, meaning that multiple invocations of a kernel can be live concurrently.

The main barrier to efficient pipelining is the HLS tools' ability to determine dependencies across initiations of the pipeline. Most significantly, a `read-after-write` (RAW) hazard may occur when later iterations of the pipeline require the use of a value written in earlier iterations of the pipeline. In such cases, later iterations must wait for the earlier pipeline iteration to compute a value before the later iteration can proceed. Such dependencies may also occur across memory accesses, wherein each loop iteration reads from and writes to the same memory. In such cases, the HLS tool often has to conservatively assuming that a RAW hazard can occur on *any* iteration, thus substantially reducing the `initiation interval` (II) of the pipeline, being the number of cycles between subsequent initiations of a pipeline. To illustrate, consider the `histogram` kernel in Figure 2.1.

```
void histogram(int f[N], int hist[N]) {
  for (int i = 0; i < N; ++i)
    hist[f[i]]++;
}
```

Figure 2.1: Histogram kernel.

The `hist` array is accessed by a value that is read through a pointer, `f[i]`. Due to this, it is impossible to statically determine whether subsequent loop iterations indexing into `hist` may alias. As such, a statically scheduled flow will, in general, resort to conservatively assume that *all* ambiguous indexes into `hist` may alias. This is visualized in Figure 2.2, assuming a memory access latency of one cycle and an adder latency of three cycles. Due to this, subsequent iterations of the loop body may only start once every five cycles, i.e., II=5. This conflicts with our desire to maximize throughput by minimizing the II of the kernel.

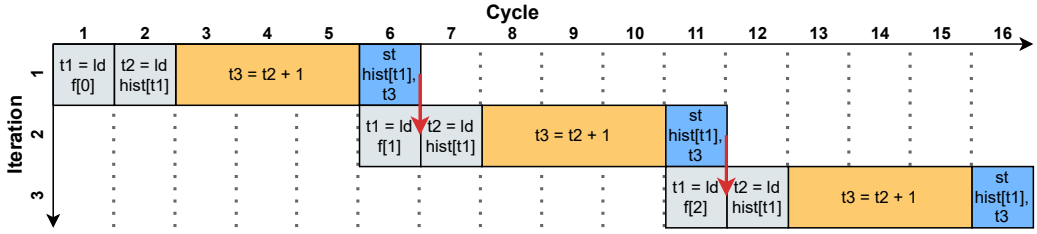


Figure 2.2: Static schedule of the `hist` kernel. Red arrows denote possible inter-iteration dependencies.

2.1.1 Dynamically Scheduled HLS

Statically scheduled HLS fails to generate high-performance circuits in the presence of dependencies that are ambiguous at compile time. This may occur with the existence of variable-latency functional units, memory dependencies, and control dependencies. This observation does not solely impact HLS, it is rather a central issue in the field of computer architecture, and similar challenges are met in the design of (statically scheduled) VLIW processors [27] versus their dynamically scheduled counterparts.

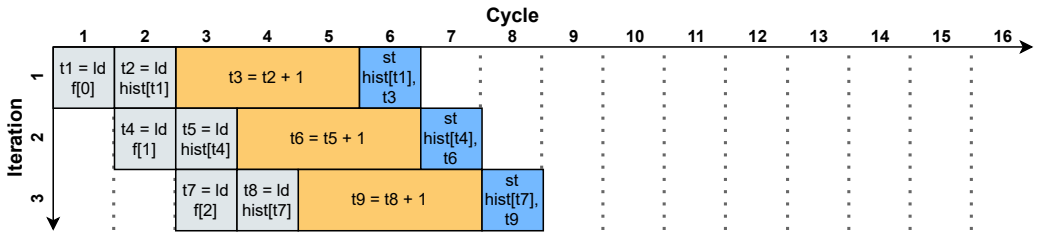


Figure 2.3: Dynamic schedule of the `hist` kernel, assuming no runtime dependencies.

Consider again the histogram kernel in Figure 2.1. While a statically scheduled approach has to conservatively schedule based on worst-case assumptions, in practice, it may be that *actual* conflicting accesses into `hist` occur rarely. We want to optimize for the common case, which is where a dynamically scheduled approach is beneficial. Instead of relying on static worst-case assumptions, dynamically scheduled circuits insert runtime dependency checks which guide the scheduling of the hardware units employed throughout the circuit. In this model, we can achieve the best-case dynamic schedule, shown in Figure 2.3 with an $II = 1$ and much greater parallelism[1].

Recently, methods for dynamically scheduled HLS have been developed, allowing for a structured approach to convert a *control and dataflow graph* (CDFG) program into a dataflow circuit [30, 31]. This method relies on the use of *elastic circuits* [11], a latency insensitive, composable dataflow model. An elastic interface consists of handshake control signals (ready/valid wires) alongside a data signal. Elastic circuits lend themselves to both synchronous and asynchronous design, wherein request-acknowledge handshaking can be used to connect FSMs of variable or unknown latencies in sequence. As compared to a statically scheduled approach, control in elastic circuits is fully distributed, implying that each component defines its outputs (readiness of input handshake ports, validity of output handshake ports) based solely on its inputs (validity of input handshake ports, readiness of output handshake ports) as well as its current state, if synchronous.

[1] Assuming no constraints on the number of ports in the `hist` memory, allocation of three distinct adders or alternatively a single pipelined adder.

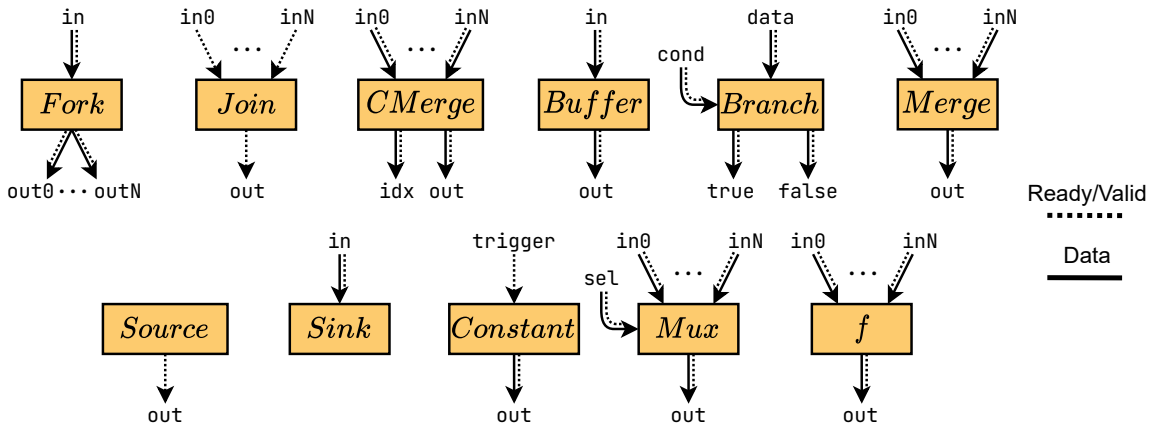


Figure 2.4: Dataflow operators considered in this work.

Figure 2.4 shows the operators used in the dataflow model of this work. These dataflow components have the capabilities necessary to translate a CDFG into a semantically equivalent dataflow circuit. The semantics of these operations are as follows:
A **fork** operation takes a single input and replicates it to N outputs.
A **join** operation is a control-only operator which assigns `out` to valid once all inputs are ready.
A **control merge** operation non-deterministically transacts any valid input on `out` and the index of the transacted input on `idx`.
A **buffer** operation can buffer a transaction through decoupling its input readiness from its output readiness. The capacity of the buffer defines the maximum number of transactions stored within. Implementation style can be sequential to break combinational paths or as a transparent *first-in, first-out* (FIFO) buffer, used to manage backpressure. In dataflow circuits, a buffer can be inserted on any path without modifying the semantics of the circuit [7].
A **branch** operation will, upon both its `cond` and `data` inputs being valid and the selected output (`true/false`) being ready, emit its `data` operand to the selected output.
A **merge** operation functions like the control merge operation, but does not provide an index output.
A **source** operation always has its output valid and can be used as a continuous source of control tokens.
A **sink** operation is always ready to accept a transaction on its input and can be used to tie off unused signals.
A **constant** operation will, when provided with an input control value, output a compile-time fixed value on its output.
A **mux** operation will, upon its `sel` and selected input signal being valid, and `out` being ready, output the value of the selected input on its output.
Any arbitrary function can be placed within a handshaking component, represented by the f operation; such components will have join semantics for synchronizing the input and outputs - all inputs must be valid, and the output must be ready before the function may provide a valid output. This is also what is known as a *unit-rate actor* [20], and is used for wrapping unary and binary operators ($+$, $\wedge$, $\neg$, ...).

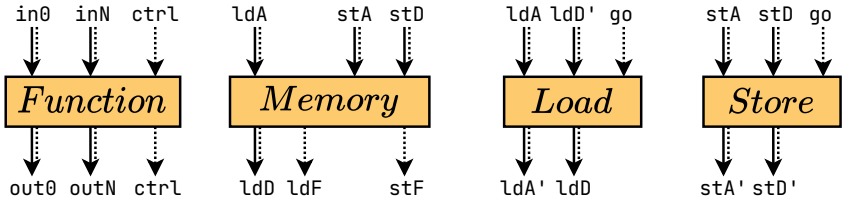


Figure 2.5: Dataflow function and memory operations.

Figure 2.5 shows how the concepts of functions and memories are handled in the dataflow model. A function is a component with an arbitrary number of input and outputs, and mandatory input and output control signals. These control signals indicate when the function should start execution and when the function is finished. The memory component provides (in our model) an arbitrary number of input and output ports. Each load and store port has an associated output control signal to indicate that the memory operation was completed. For a store operation, `stA`, `stD` represents address and data signals coming from a predecessor operation. These are sent to a memory operation through `stA'`,`stD'`, subject to synchronization with the `go` signal. Similar logic applies for the load operation, wherein `ldD'` represents data coming from a memory, and `ldD` data sent to a successor operation. The `go` signal can be used as a synchronization mechanism to stall the execution of a memory operation, in cases of runtime RAW conflicts.

Having a dataflow representation of a CDFG program, operators must be lowered further into their hardware representation. Edwards et al. [20] describes the implementation of *compositional* dataflow components. These are hardware implementations of the dataflow operators mentioned earlier, that have the property of being composable, while breaking combinational cycles in the control network of a circuit. This style of implementation will be revisited in section 3.3.

Finally, we note that while dynamically scheduled circuits show promising performance, there is no such thing as a free lunch. Statically scheduled circuits will theoretically always remain superior to their dynamically scheduled counterpart in cases where all dependencies and latencies are known at compile time—thus removing the need for design elasticity and the hardware overhead which comes with such an implementation. DASS [12] explores how a design may be partitioned into its statically and dynamically scheduled parts, striking a balance between possible performance and resource usage, showing promising results in cases where such partitioning is done.

2.1.2 HLS Infrastructure

Traditionally, open-source HLS tools have leveraged software compilation infrastructures, with LLVM being the most commonly used [48]. Some examples of LLVM based HLS tools are LegUp [9], able to generate a design using a mixture of RTL and code running on a soft MIPS processor and AutoPilot [15, 63], able to synthesize a mixture of C, C++, and SystemC, which also provides methods for design exploration and cosimulation. The use of a software compilation infrastructure is with good reason; HLS and

software compilation share many optimization goals, a few examples being *common subexpression elimination* (CSE), constant folding, and loop transformations [15, 22]. In general, these optimizations are performed on unstructured IR—an IR which expresses control flow through the use of basic blocks and branches. Structured IRs, and abstract syntax trees (AST) in general, define control flow through operations like `if-`, `for-`, and `while` constructs.

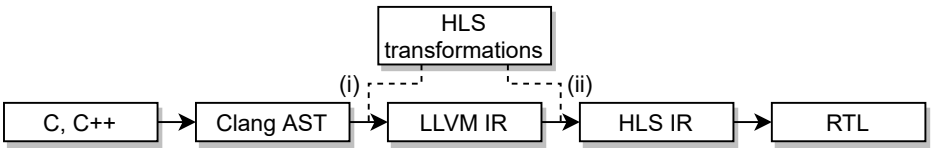


Figure 2.6: Example LLVM-based HLS flow, using C/C++ as its source language. HLS transformations are performed both at the structured AST level (i) as well as the unstructured LLVM IR level (ii). Often, some form of domain-specific HLS IR will be used to bridge the gap between LLVM IR and RTL emission.

Figure 2.6 shows how an HLS compiler might be based on an LLVM pipeline. HLS transformations may be performed both before and after LLVM IR. A large and important class of HLS optimizations, loop nest optimizations, is most easily performed before conversion into unstructured IR. In an HLS context, loop optimizations may aid in e.g. guiding loop splitting to improve pipelining performance [38], to guide loop tiling and unrolling, as well as memory access analysis and partitioning, to exploit data and loop level parallelism in hardware [13, 14]. Many of such optimizations can only apply when program structure is maintained or recoverable. Once unstructured, memory access patterns and control flow become at best hard to analyze, and at worst, ambiguous, to the point where transformations cannot apply without danger of modifying program semantics. As a result, many HLS optimizations need to be performed at the AST level (Clang AST, for C/C++ based HLS flows, Figure 2.6(i)) where structure is maintained. This then presents the issue that ASTs in LLVM are source language specific, thus restricting the HLS transformations to a single source language. A better solution would be to anchor transformations to the concepts that the transformation cares about, i.e. the concept of a loop, and make that source independent. This is the approach taken by MLIR, as we shall see in section 2.2. While not necessarily a barrier for research, it is a significant limitation if we want open-source HLS tools to learn from staged compiler design, decoupling source and target-level transformations, as well as avoiding reimplementing that which is shared between all HLS flows.

Due to the recent open-sourcing of the Xilinx Vitis [61] LLVM front-end, we can gain insight into how commercial tools handle some of the complexities discussed above. The Vitis front-end ingests, and transforms, LLVM IR and annotates source-level pragmas at points in the IR where the pragma semantics most closely relate to the operations of the source program, e.g. loop-level pragmas being attached as attributes to backedges in the *control flow graph* (CFG). These attributes—scattered across the unstructured IR—are used in the (closed-source) Vitis backend to guide HLS transformations and hardware emission. We may also imagine that more complex semantics are inserted as function calls at the LLVM IR level, which later map to intrinsics[2] implemented by the HLS backend. While we can only speculate about the challenges and restrictions faced by such a commercial tool, such an approach

[2]Compiler intrinsics, also known as built-in functions, are functions implemented by the compiler itself. These may map to a sequence of instructions or aid in guiding program transformations.

still suffers from the removal of structured information about memory access patterns and control flow, which may be vital in informing the backend about high-level program behaviour.

Programming models for driving HLS flows can be divided into two groups; general-purpose and domain-specific. Considering general-purpose programming models, low-level languages, most notably C/C++, have long been the de-facto language for commercial tools such as Vivado HLS [28] or Intel HLS [54]. Being imperative in nature, such languages allow for static analysis of program behavior and memory accesses. Contrary to this, domain-specific HLS flows consider a more restricted programming model. Due to this, such flows can place more assumptions on, e.g. program control flow, data liveness, and memory accesses. By doing so, such flows allow for complex program transformations only possible due to the added information provided by the programming model itself. Domain-specific HLS has become increasingly prevalent during the past decade due to the rise of domain-specific accelerators. Tools such as FINN [56], a framework for building FPGA accelerators of binarized neural networks, Dahlia [42], a programming language for designing FPGA accelerators, and StreamBlocks [5], a compiler for actor-based dataflow programs, all act as front-end tools that produce C++ code for further synthesis in Vivado HLS. This is reasoned by the fact that these high-level programming models perform program transformations based on domain-specific knowledge. C++ is then the semantically closest point at which such front-ends can offload into existing HLS infrastructure for further synthesis.

This observation serves as a motivation for this work; while it is acknowledged that C++ is a highly capable programming language, using it as a de-facto intermediate representation forces high-level programming models to lower their basic units of computations (tensors, actors, ...) into structures that are identifiable by a C++ compiler—all while trying to coerce the tool into understanding the existence of any inherent parallelism clearly identified by the front-end tool. Such information is communicated through pragma insertion at places in the C++ code where a front-end can hope that the HLS tool will understand the intentions, but no guarantees are provided that the knowledge of the front-end is properly communicated to the HLS tool. While this work focuses on, as we shall see, implementing a C/C++ based HLS flow, building end-to-end HLS flows based on domain-specific IRs is ongoing research and highly related to this work [57].

2.2 MLIR

The growing importance of domain-specific accelerators requires a new approach to the design of compiler infrastructure. In this new space, with mixtures of high-level and low-level program representations, operations, target-specific and target-independent optimizations, different models of parallelism, and different levels of abstraction, the need for compilers that allow for code and concept reuse across different domains is evident. One such approach is the MLIR compiler infrastructure [36].

To understand MLIR, let us first consider its historical predecessor, LLVM. The LLVM project [35] is an open-source compiler infrastructure that has, over the past two decades, grown into not only one of the most popular infrastructures for implementing programming languages but also an essential resource for compiler research. Traditionally, LLVM has been used in conjunction with the C-family languages such as C and C++, targeting all commonly used CPU architectures. Over time, however, the computing landscape itself has seen a shift. At the time of the projects' inception (2004), computing was still CPU-dominated, albeit with the slowdown of Moore's law looming around the corner. In the following years, we saw increasing reliance on parallelization and specialization through the emergence of multi-core CPU and GPU architectures—a shift that the LLVM project has been able to accommodate. Within the past decade, however, the slowdown in the capabilities of traditional compute has been noticeable, and the need for highly specialized architectures has become increasingly crucial through the introduction of new workloads, most notably the increasing importance of AI. Specialized architectures, such as AI accelerators, are motivated by the need to accelerate a specific programming model [33, 44], and the instruction set of such accelerators is typically expressed at a higher level of abstraction than the unstructured TAC[3]-based IR typical of many compiler infrastructures, LLVM including.

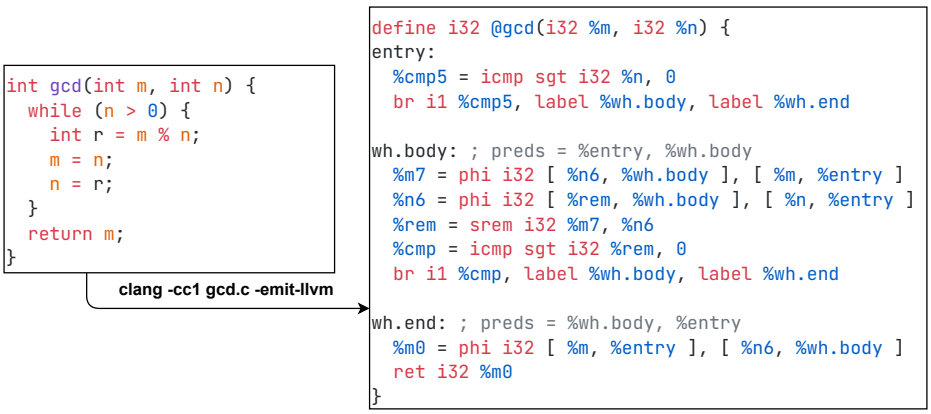


Figure 2.7: gcd function lowered to LLVM IR through Clang.

Figure 2.7 shows how a example C function is represented in LLVM IR. LLVM IR expresses control flow through a *control-flow graph* (CFG) of labeled basic blocks and branches as CFG edges. ϕ-functions are used to select control-flow dependent values, defined in predecessor basic blocks. Internally within each basic block, a dataflow graph (DFG) is realized by the operations and values defined and referenced.

[3]Three Address Code, a RISC-like IR representation with operators taking two operands and returning a single result.

LLVM IR is a *static single assignment* IR, requiring that each value (SSA operand) defined is assigned exactly once. Information regarding the structure of the source code has largely been lost at the point of lowering to LLVM IR[4].

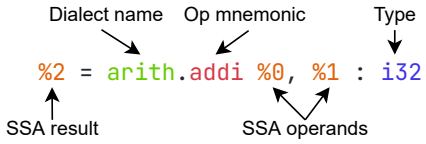
An issue arises when considering how novel, domain-specific programming models target LLVM. Due to the need for domain-specific optimizations and analysis, many LLVM-targeting projects implement their own AST or IR, which later lowers to LLVM IR. Each comes with a considerable cost to implement, especially when considering that identical concepts may be reimplemented for each source representation. This also means that an opportunity to benefit from the gains made by other projects on shared infrastructure is missed. This is one of the primary motivations of MLIR - to provide a compiler infrastructure for facilitating the creation of domain-specific IRs, while abstracting away that which is needed for all IRs, such as parsing, printing, pass management, SSA construction, and optimization.

2.2.1 Dialects and Operations

In compiler and IR design, we seek to define explicit abstractions for capturing concepts of software systems. When designed correctly, such abstractions can be complementary, mutually exclusive in their scope, and composable, ensuring maximal reuse and modularity.

In MLIR, a domain-specific IR can be captured by a *dialect*. At the syntactical level, dialects are a collection of operations, attributes, and types that describe a particular domain, which together make up the domain-specific IR of the dialect. At the semantical level, dialects and operations can specify things such as *verifiers*—determining the legality of an operation based on its input operands and types, and *canonicalizations*—specifying rules for transforming an operation to a canonical form (peephole optimizations). By adding such capabilities to the IR itself, MLIR ensures that it is not just a ”JSON of compiler IRs” but also able to define IRs that can reason about themselves, their structure, and validity.

Let us now consider how textual MLIR is represented.

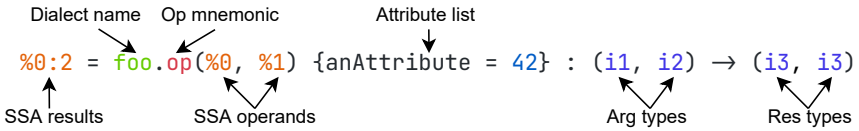


The above shows an operation from the `arith` dialect, a dialect intended to capture the semantics of integer and floating-point arithmetic. On the left-hand side of the operation, any results defined by the operation are written (`%2`) - this is the point of definition for SSA values. Next, a dialect operation is prefixed with the dialect name (`arith`), followed by the mnemonic of the operation (`addi`). In the case of integer addition, two SSA values are required as operands, and the operation takes a single type parameter. For any operation, all types of its operands and results must be resolvable based solely on the

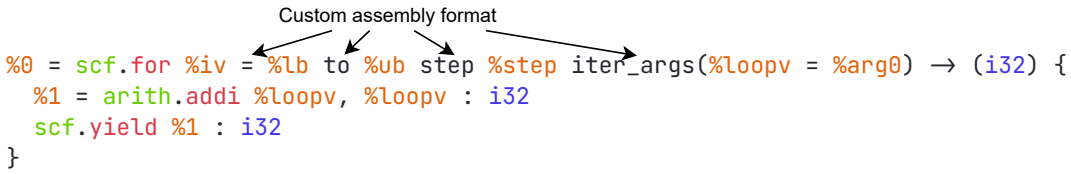
[4]It is possible for the front-end that generates LLVM IR to annotate the IR with metadata such as annotations on CFG back edges to indicate loops, and in other cases, IR may be raised to a higher level of abstraction if specific patterns can be inferred from the CDFG. However, in general, information will inevitably be lost when lowering from a high-level AST into LLVM IR.

types made available when parsing the operation in isolation. In other words, types cannot be deduced contextually. In the case of `arith.addi`, all operands and results of the operation have an equivalent type (`i32`).

The `arith` dialect operations are easily understood due to the nature of the concepts which are represented. However, it is important to realize that MLIR operations and dialects are at heart, constructs that capture a *concept* which may have *structure*, and it exists within a *typed* and *SSA* context. To emphasize this, consider the following operation:



Here we have a made-up operation `op` from a made-up dialect `foo`. In this case, the operation defines multiple results (`%0:2`). This is a shorthand notation for a list of SSA values - other operations can reference the individual values of this list through references to `%0#0` and `%0#1`. For this operation, we have also specified an *attribute list*. Inside an attribute list, any combination of named (key-value) or unnamed (unary) attributes can be specified, providing a mechanism for attaching statically known information to an operation. Finally, the type of this operation has been specified using a *function-like* type declaration `() → ()`. While this operation does not have any semantics attached, it is a fully valid operation in MLIR.



The final example that we will consider is the `scf.for` operation. The SCF (structured control flow) dialect captures concepts such as `for`, `while` and `if` statements, which can be further lowered to unstructured control flow–a CFG. The `scf.for` operation makes heavy use of the ability to specify custom assembly formats for an operation. In this case, `%lb`,`%ub` reference SSA values dominating the `scf.for` operation, specifying the lower and upper bound of the loop. `%iv` is the induction variable of the loop, which may be referenced inside the body of the loop. `%step` defines the incrementation of `%iv` on each iteration. `iter_args` specifies a list of loop-carried values (`%loopv`), each of which must have an initial value (`%arg0`) and a type to guide type resolution (`i32`). Types are not specified for `%lb`, `%ub`, `%step`, `%iv` since these are all defined as being of type `index`, an integer-like type with a platform-dependent bit-width, typically used in cases such as iteration or memory indexing.

This operation also gives insight into how operation verifiers work. `scf.for` implements a verifier requiring that its body is terminated by an `scf.yield` operation, an operation that specifies the values to be forwarded to the next loop iteration (or, upon exiting the loop, to be written as the loop result `%0`).

This operation must have an identical type signature to the `iter_args`, and if not, the operation verifier will fail.

2.2.2 IR Semantics

Semantics shared by all dialect operations, i.e. those defined by the infrastructure itself, are: all values are SSA. Values can only be defined as either the result of an operation or as a block argument. Structure is added to the IR in two ways: through blocks (grouping structure), defined as sets of operations nested within the scope of a parent region, and through regions (hierarchical structure), which can be defined by an operation, and themselves contain blocks. Each block may have a list of values as arguments. Values may be referenced whenever defined in a block dominating the parent block of the use location or those defined in a parent region if the operation that defines the region does not have the property of being *isolated from above*. This property also allows the infrastructure to schedule transformations in parallel.

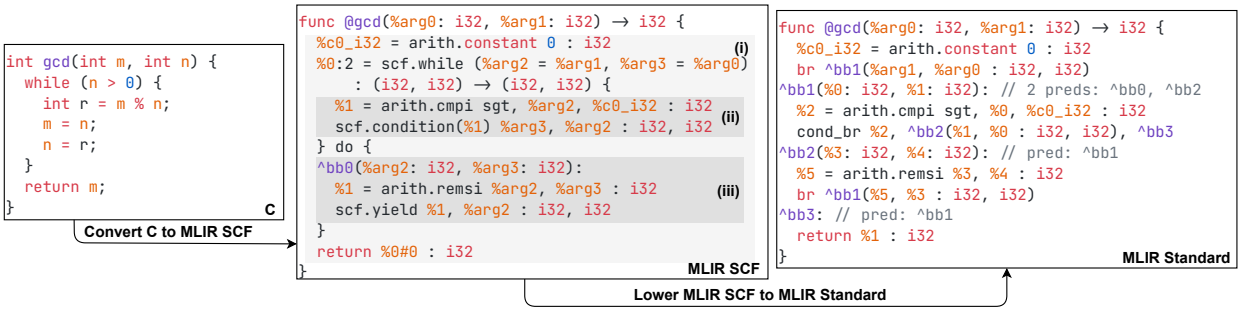


Figure 2.8: A `gcd` function at C, `scf` (structured control flow), and `standard` (CDFG) levels of abstraction.

Consider Figure 2.8. We here see two different representations of the `gcd` function in MLIR. The first, MLIR SCF, represents an IR where structured control flow is maintained. This structure is defined through the `scf.while` operation. The `scf.while` operation defines two regions, a conditional region (ii) and a body region (iii). In the conditional region, a continuation condition is computed as a boolean value (`%1`) and passed to the `scf.condition` operation. The variables passed after the condition value (`%arg3`, `%arg2`) represent arguments to the *body* region of the for loop. In region (ii), we see that the `arith.cmpi` operation is able to reference an SSA value defined in the parent region (`%c0_i32`).

The second, MLIR Standard, represents an IR where control flow is unstructured. Such IR abstraction closely resembles LLVM IR, with blocks being used identically as basic blocks in LLVM IR, and branches are used for control flow.

In both representations, we see a `%c0_i32 = arith.constant 0 : i32` operation. This contrasts with the LLVM IR, wherein constant literals can be provided as operands to an operation. In MLIR, we represent integer comparison through the `arith.cmpi` operation. This operation is defined as taking two integer-type SSA values as operands and, since the operands are values due to the semantics of the IR, they must be defined by either an operation or a block argument—hence, the inclusion of the `arith.constant` operation.

Having multiple representations of the same program provides multiple benefits:

- The SCF operations maintain structured information that can be analyzed and inform transformations. Compared to LLVM, such structure would exist internally in the AST of a front-end that produces LLVM IR, and most likely `only` exists as a data structure (in contrast to a textual representation, allowing for storage, manual modification, and interchange between tools).
- Furthermore, we have the option of lowering the structured control flow to unstructured control flow and continue our compilation flow from there, possibly into LLVM IR to leverage an existing CPU-targeting flow.

This means that we are free to perform transformations at a level of abstraction which suits our needs—if compiling to a GPU target, it might be relevant to parallelize and desynchronize a loop body. Alternatively, in the case of HLS, we may want to transform the structure of nested loops or transform the loop body into a pipelined datapath. This illustrates a cornerstone of MLIR, i.e., clearly defined concepts are abstracted away into separate dialects. The power of the compilation flow that one defines is then based on the transformations of target dialect operations.

Until now, we have only considered SSACFG regions—regions where use-def chains are defined by the dominance relationship between blocks and their operations. While such representation applies to most sequential software, in hardware, we often work with abstractions that represent cyclical graphs, something which is cumbersome to represent in an SSACFG. To address this, a region can be defined as a *graph region*. In a graph region, SSA dominance now considers all operations within a block (as opposed to only the sequential predecessors in SSACFG), allowing operations to reference values defined at a point succeeding the point of reference.

2.2.3 On Executability

When considering MLIR IR such as that shown in Figure 2.8, due to the resemblance to programming languages, a common misconception is to assume that IR operations are CPU executable. In practice, the constraints here are identical to those of regular programming languages; there needs to exist either an interpreter capable of executing the operations of a language or a compilation path capable of lowering operations to something executable. In MLIR, the main goal for dialects and operations is to facilitate transformation—this does not necessarily imply that CPU executability is required or prioritized. This is especially true for operations in dialects that start to diverge from being CPU-facing, such as the hardware-related dialects we will consider in this work.

In practice, for many operations in MLIR that model mathematical and software concepts, lowering paths *do* exist to make such IR executable. Operations in the `linalg` and `vector` dialects lowers to loops in the `scf` or `affine` dialects. These operations may themselves lower to the `standard` dialect (Figure 2.9(i)). From here, operations can lower to the MLIR LLVM dialect (Figure 2.9(iii)). Note that this is is not LLVM IR but a separate dialect in MLIR which models LLVM IR. The MLIR LLVM dialect can then be exported to LLVM IR (Figure 2.9(iv)), which the LLVM infrastructure ingests and can be either compiled

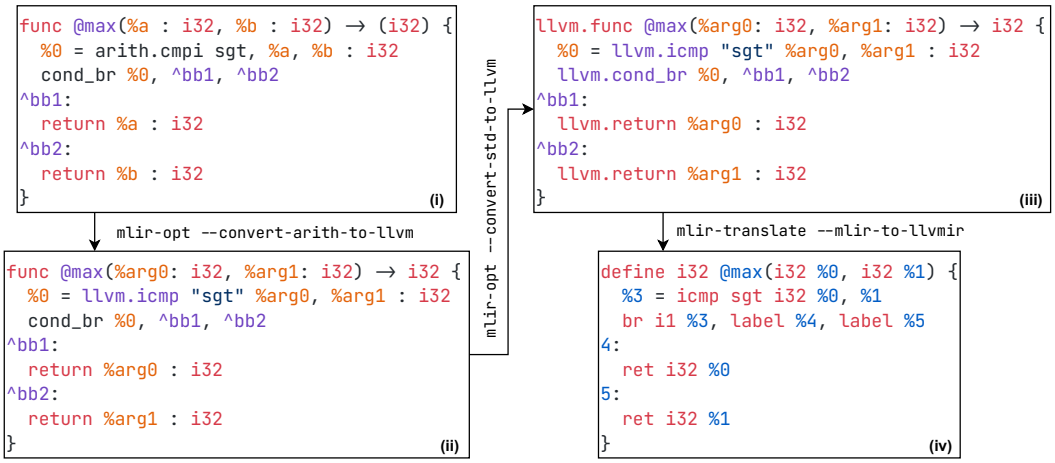


Figure 2.9: Operations from different dialects (`arith` (i) and `std` (ii)) are progressively lowered to MLIR LLVM (iii), which eventually gets exported as LLVM IR (iv).

or JIT executed. The `mlir-cpu-runner` tool is provided to facilitate execution of MLIR LLVM programs. This tool is a wrapper around the LLVM just-in-time (JIT) compiler. The use of the `mlir-cpu-runner` tool is revisited in chapter 4. Finally, we note that building IR interpreters is also a popular option for providing execution semantics to dialect operations.

2.3 CIRCT

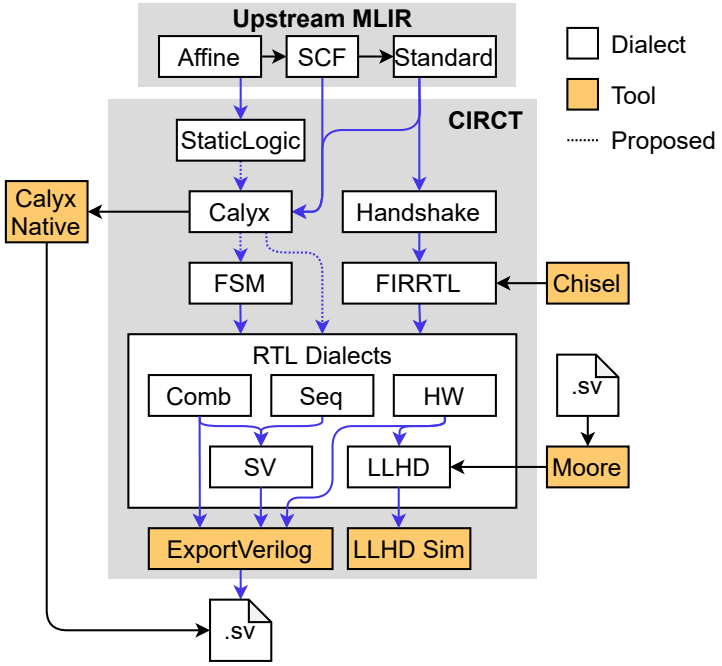


Figure 2.10: Overview of the dialects in CIRCT, how they interconnect, as well as relevant tools for producing inputs to and transforming outputs from CIRCT.

CIRCT is an LLVM incubator project for developing open-source MLIR-based electronic design automation (EDA) tools. The open-source EDA tool space has become increasingly capable in recent years, allowing for an end-to-end synthesis and programming flow without the use of vendor tools. An example of a Xilinx-targeting flow is that of Symbiflow [40] wherein Yosys [60] is used for hardware synthesis, NextPNR [51] for place-and-route, and Project X-Ray for bitstream generation [46]. However, an issue with the current space is much like that which faced LLVM regarding fragmentation across front-end tooling. Open-source hardware tooling generally does not use a library-based approach for infrastructure shared across all tools, thus suffering from having to reimplement logic in projects. The design of hardware systems is inherently hierarchical, and existing tools already leverage a variety of intermediate representations to represent such hierarchy. However, these are not necessarily explicit, and different abstractions may be highly coupled within the tools. Finally, shared IRs, allowing for interchangeability between the different tools, do not exist. Verilog/SystemVerilog is generally used as the de-facto netlist standard in the open-source EDA community, for pragmatic reasons. Verilog and SystemVerilog were never intended to be IRs and as such have broad-ranging specs that make them hard to parse, analyze, and generate. In contrast, having hardware abstractions as real IRs, which are designed to be analyzed and transformed, enables us to build hardware tools in a cheap and scalable manner. It is in this context that CIRCT exists. Using MLIR, CIRCT seeks to define explicit abstractions for hardware synthesis, transformations on these, and lowerings to synthesizable HDL. At the time of writing, CIRCT concerns itself mainly with the behavioral and structural aspects of hardware synthesis. External tools are still expected to be leveraged for the physical domain, with Verilog as the output format. This may

then be piped to some of the above-mentioned EDA tools or proprietary vendor tools.

2.3.1 A Tour of CIRCT

Figure 2.10 shows an overview of the dialects in CIRCT, how they interconnect, as well as various tools for producing inputs to and transforming outputs from CIRCT. The core of CIRCT is the collection of RTL dialects. These dialects express structural and behavioral aspects of synchronous hardware, each with different passes to transform, optimize, and canonicalize the IR of the respective dialect.

The `hw` dialect represents hierarchical concepts such as hardware modules, module ports, and module instantiation. The remainder of the RTL dialects implement operations that can exist within `hw` modules. A `hw` module implements a graph region instead of an SSACFG region. Below, we see an empty `hw` that connects its input directly to its output:

```
hw.module @foo(%in: i32, %clk: i1, %rst: i1) → (out: i32) {
  hw.output %in : i32
}
```

Lowers to →

```
module foo(
  input [31:0] in,
  input clk, rst,
  output [31:0] out);
  assign out = in;
endmodule
```

The `comb` dialect represents combinational concepts such as addition, subtraction, and multiplexers. Below, we now place an adder that adds the input with itself and outputs the sum:

```
hw.module @foo(%in: i32, %clk: i1, %rst: i1) → (out: i32) {
  %0 = comb.add %in, %in : i32
  hw.output %0 : i32
}
```

Lowers to →

```
module foo(
  input [31:0] in,
  input clk, rst,
  output [31:0] out);
  assign out = in + in;
endmodule
```

The `seq` dialect represents sequential concepts such as registers. Below, we now have a register after the adder which breaks a combinational path between the input and output ports of the module:

```
hw.module @foo(%in: i32, %clk: i1, %rst: i1)
    → (out: i32) {
  %c0 = hw.constant 0 : i32
  %reg.out = seq.compreg %in, %clk, %rst, %c0 : i32
  hw.output %reg.out : i32
}
```

→

```
hw.module @foo(%in: i32, %clk: i1, %rst: i1)
    → (out: i32) {
  %c0_i32 = hw.constant 0 : i32
  %reg.out = sv.reg
    {hw.verilogName = "reg_out"} : !hw.inout<i32>
  %0 = sv.read_inout %reg.out : !hw.inout<i32>
  sv.alwaysff(posedge %clk) {
    sv.passign %reg.out, %in : i32
  }(syncreset : posedge %rst) {
    sv.passign %reg.out, %c0_i32 : i32
  }
  hw.output %0 : i32
}
```

→

```
module foo(
  input [31:0] in,
  input clk, rst,
  output [31:0] out);
  reg [31:0] reg_out;
  localparam [31:0] _T = 32'h0;
  always_ff @(posedge clk) begin
    if (rst)
      reg_out <= _T;
    else
      reg_out <= in;
  end
  assign out = reg_out;
endmodule
```

Figure 2.11: Lowering registers to SystemVerilog through the `sv` dialect.

The `SV` dialect is responsible for managing SystemVerilog specific constructs, many of which target

either behavioral, verification, or textual related aspects of the language—SystemVerilog is used as a tool interchange output format for CIRCT. Some example operations are `sv.alwaysff`, used to enforce inference of registers (used in Figure 2.11 when lowering `seq.compreg`). `sv.bind`, allowing for indirect instantiation of hardware modules—often used in verification contexts to insert verification modules into existing RTL designs. And `sv.ifdef`, inserting conditionally compiled regions, giving existing HDL tooling control over various synthesis parameters of the generated HDL.

LLHD [50] is a low-level hardware dialect that, when used in conjunction with the remainder of the RTL dialects, can model and simulate hardware systems. As an example, LLHD extends the capabilities of the RTL dialects by being able to capture timing information from SystemVerilog, allowing for timing-accurate simulation using the `llhd-sim` tool. LLHD is also the target of the Moore compiler [50], a compiler front-end for HDLs which can parse Verilog/SystemVerilog. Further developments on the Moore compiler and LLHD will ensure that CIRCT applies to a wide range of existing HDL codebases.

FIRRTL [29] is a hardware IR that was created based on the observation that a one-shot lowering from the Chisel [2] HDL all the way to Verilog was needlessly complicated, and did not facilitate circuit transformations. The development of FIRRTL in CIRCT has been one of the main driving forces for the maturation of the CIRCT project. Currently, FIRRTL is actively developed by SiFive, and the CIRCT FIRRTL based compiler can be used as a drop-in replacement for the native FIRRTL compiler. The FIRRTL dialect models many existing concepts of the RTL dialects, with the addition of handling Chisel-specifics and oddities.

Given an input program described in a mixture of RTL dialects, the ExportVerilog tool can emit a synthesizable SystemVerilog description. The tool allows for the specification of lowering options to ensure that the generated HDL is compatible with the tool that one is targeting—an example being indexing into multidimensional arrays, something which may be trivially described in an IR, but which is not necessarily supported by all tools. In such cases, lowering options can guide Verilog emission to either emit multidimensional array indexing or emission of temporary wires representing subarray indexing, thus factoring away the complexities of Verilog emission from the RTL dialects themselves.

2.3.2 HLS in CIRCT

Next, we consider the current state of HLS in CIRCT. It is important to note that prior to this work, significant work has already gone into developing critical parts of an HLS flow in CIRCT. In this section, we shed light on the current state of HLS in CIRCT, both in terms of statically and dynamically scheduled HLS, as well as supporting libraries.

The Handshake dialect represents dataflow programs where producers and consumers transact values through a handshaking mechanism. The dialect is designed based on the operators described in subsection 2.1.1. Dynamically scheduled HLS is achieved through the `std-to-handshake` pass, a pass that converts Standard dialect programs into Handshake IR. Handshake programs can then be lowered to a FIRRTL representation through `handshake-to-firrtl` - which can then be lowered to HDL through the existing FIRRTL path. The operations of the dialect and related lowerings are explained in detail in

chapter 3.

The Calyx dialect models the Calyx *intermediate language* (IL) [43], a hardware IL for accelerator design. Calyx works by splitting the representation of an accelerator into three parts: instantiations of structural components, the definition of interconnect groups, defining connectivity between components, and a control schedule for defining the order in which interconnect groups are activated. With this separation of concerns, each part of the design can be transformed and optimized separately before being lowered to an FSMD representation. In CIRCT, Calyx is being developed as a core abstraction for statically scheduled HLS designs. Support for generating Calyx programs in CIRCT exists for both `standard` (branch-based control flow) and `SCF` (structured control flow) operations through the `scf-to-calyx` pass. At the time of writing, Calyx cannot be lowered to the RTL dialects, but support is expected in the future. Instead, a Calyx MLIR program can be exported to native Calyx IR, which then may be compiled to SystemVerilog through the native Calyx compiler.

The `StaticLogic` dialect represents ongoing research into the development of abstractions and lowerings for representing statically scheduled pipelines. The generation of a `StaticLogic` pipeline expects loops with complete information on inter-loop and memory dependencies. In the MLIR case, this style of loops is represented by the `affine.for` operation, describing loops adhering to the polyhedral model [21]. By using this, the `StaticLogic` dialect can leverage existing dependency analysis tooling available in MLIR. To assist pipeline scheduling, a scheduling library is being developed within CIRCT. The scheduling library is based on modulo scheduling [47] and can solve both resource-free and resource-constrained scheduling problems [18]. `affine.for` loops, and their accompanying dependency information, are used to build scheduling problems which are solved for optimal initiation intervals under the given constraints. Scheduling results are then used to build a `staticlogic.pipeline.while` operation, which describes a set of stages, loop registers, and a loop continuation criteria. Such representation can then be further lowered to Calyx, which turns the resulting pipeline into an FSMD representation.

Chapter 3

An End-To-End Dynamically Scheduled HLS Flow in MLIR

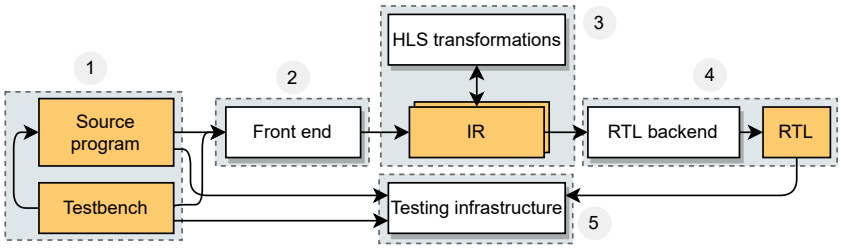


Figure 3.1: Core components of an HLS flow.

In subsection 2.1.2 we provided an overview of the components needed for an LLVM-based HLS flow. To build an MLIR-based flow, we need to consider which components and transformations we need, and the ordering thereof, to have an end-to-end HLS flow. The components and interconnections which we need to resolve, shown in Figure 3.1, are as follows:

1. At what level of abstraction do we describe an accelerator? Alongside that, we will also want the ability to describe testbenches at the same level of abstraction.
2. What front-end will we use to convert the source representation into an MLIR representation?
3. Once in MLIR, how do we lower from a sequential software program into something with hardware semantics? What representations (IRs) and transformations do we need to do so?
4. How do we lower our HLS IRs to RTL?
5. How do we verify and validate the generated RTL-level accelerators using the source-level testbenches?

This chapter concerns itself with points 1 through 4. In chapter 4, we will cover point 5 by introducing tools for cosimulation and debugging.

3.1 Source Abstraction and Front-End

When designing a high-level synthesis flow, one of the first things that must be considered is the level of abstraction at which we expect inputs to the flow to be at. As explained in subsection 2.1.2, both domain-specific and general-purpose HLS flows are viable source abstractions for a modern HLS flow. However, due to the prevalence of domain-specific tools that generate C/C++ code, as well as when considering existing code-bases that target vendor HLS tools, we deem it prudent to design a flow able to ingest C/C++.

At the time of writing, the number of front-ends available for generating high-level MLIR are limited. We desire to use C/C++, given their prevalence in both existing vendor HLS tools and open-source alternatives. With a C/C++-based flow, we can compare and interchange with existing tooling while establishing the basic capabilities of our CIRCT HLS flow. In this respect, a C/C++ front-end is needed, capable of generating MLIR from C/C++ source code. We consider ScaleHLS and Polygeist, two Clang-based projects that seek to ingest C/C++ code to perform MLIR-based transformations. ScaleHLS [62] is a tool for exploring HLS concepts in MLIR, mainly focusing on loop and graph-level analysis and transformations. To facilitate this, ScaleHLS implements a C front-end to extract affine loops from C code. The focus of ScaleHLS is to perform source-level transformations which can be emitted to existing HLS tools through a C/C++ emitter.

Polygeist [39] is a tool designed to connect existing polyhedral compilation infrastructure with MLIR. As with ScaleHLS, this project uses a Clang-based front-end and can extract affine loops from C/C++ code. Both ScaleHLS and Polygeist convert C code into a combination of Standard, Affine, and SCF dialect operations, which are suitable sources for further HLS lowerings.

While the transformations in ScaleHLS are specifically targeting HLS flows, Polygeist is chosen as the front-end for this project. We seek to ingest and correctly synthesize a large set of programs, and less so to investigate the effects of high-level program transformations, such as those found in ScaleHLS. Polygeist is seen as more applicable to our case due to its maturity and given its focus on the ability to parse as large a subset of C/C++ programs as possible into an MLIR representation. Furthermore, affine transformations for HLS such as those performed by ScaleHLS have equally been demonstrated for a Polygeist-based flow in the Phism project [64].

Memory Flattening

With Polygeist, we can ingest a broad range of C/C++ programs into our flow. However, various software-level transformations must be performed to meet preconditions of the dataflow lowering, which will be presented in the following section.

In MLIR, a value of `memref` type represents a reference to an N-dimensional memory (`memref<2x3x4xi32>`) with optionally dynamic dimensions (`memref<2x?x4xi32>`). For the hardware lowerings that are presented in later sections, only unidimensional memories are supported. While there are no technical limitations to support multidimensional memories in hardware abstractions, supporting

only unidimensional memories reduces the complexity of the abstraction. Furthermore, performing such conversion is trivial at the software level, where use-def relations on memories are easily analyzed via `memref.load/store` operations. We implement a pass `-flatten-memref`, which, given a program will transform any `memref` with multiple static dimensions into a `memref` with a single dimension. The pass inserts a combination of multiplication and addition operations, combining the original dimension indexing arguments with the static dimensions of the memory to calculate a flattened index. An example is shown below, where a `memref<10x10xi32>` is flattened into a `memref<100xi32>`.

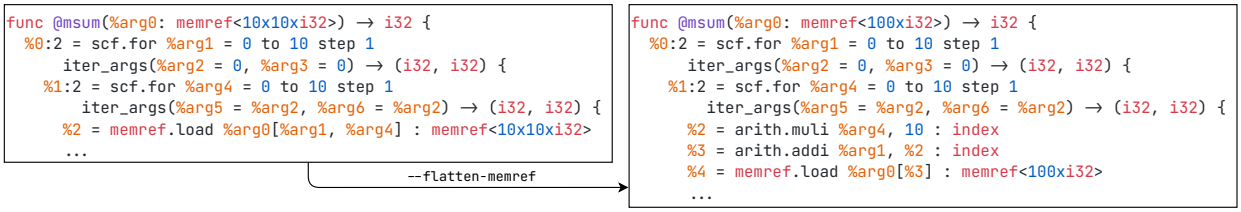


Figure 3.2: A `memref<10x10xi32>` and related accesses are flatten to a unidimensional `memref<100xi32>`.

Note that the function signature has also been converted. This poses a problem to callers who expect an API of the kernel similar to that of the C source function. To handle this, a companion pass `-flatten-memref-calls` has been implemented, which modifies `builtin.call` operations to match the converted function signature. We make this a separate pass from `-flatten-memref`, such that we can transform only the `builtin.call` to the modified function, while still allowing for multidimensional `memrefs` in the remainder of the caller IR.

The dataflow lowering presented in the following section requires an input program to be specified by a CDFG. At our current level of abstraction, control flow may be structured, defined through `affine/scf` dialect operations. We achieve a CDFG by applying existing lowering passes for these dialects. The loop lowering performed follows the well-known technique [16] of inserting basic blocks for the loop header, body, latch code, and branch operations to jump between these.

3.2 Dataflow IR & Dataflow Lowering

3.2.1 The Handshake Dialect

The concepts of dataflow circuits have been captured in a distinct dialect, the `handshake` dialect. The operations of the dialect are based on those presented in subsection 2.1.1. We seek to make the Handshake IR applicable to dataflow compilation in general, and not just for HLS flows targeting FPGAs. An example alternative use-case is the mapping of dataflow graphs onto coarse-grained reconfigurable arrays (CGRAs)[1]. These are reconfigurable devices that trade the single bit flip-flop, LUT, or wire flexibility

[1]Academic work on CGRAs is often centered around an FPGA implementation combined with a coarse-grained overlay, due to practical reasons. However, we here consider the potential for hardened CGRAs.

of FPGAs for blocks hardened at a higher level of abstraction. The use of dataflow CGRAs is an active area of research [41, 59]. An example of a design choice made to keep such generality has been to i.e., not define IR validity on SSA value-use-counts, or whether the graph contains unbuffered (combinational) loops. While such conditions must hold for the lowering we present in section 3.3, we see it as better design to have a less restrictive IR and instead provide passes to transform an IR to meet any target-specific requirements.

```
% = buffer [#] % {sequential = (true|false)} : T
% = fork [#] % : T
%:# = lazy_fork [#] % : T
% = merge %... : T
% = mux %select [%...] : T
%data, %idx = control_merge %... : T
%... = instance @sym(%...) : (T..., none)→(T..., none)
handshake.func @sym(%..., %ctrl) : (T..., none)→(T..., none)
```

```
% = br % : T
%true, %false = cond_br %cond, %data : T
sink % : T
% = source : T
% = constant %ctrl {value = # : T} : T
%dOut, %addrToMem = load [%addrIn] %dataFromMem, %ctrl : Td, Ta
%dToMem, %addrToMem = store [%addrIn] %dIn, %ctrl : Td, Ta
% = join %... : T
```

Figure 3.3: Syntax of the operations in the Handshake dialect.

The operations defined within the Handshake dialect are shown in Figure 3.3. '%' represents a single SSA value, '%...' a variable number of SSA values, 'T' a type parameter, 'T...' a variable number of Type parameters, and '#' an integer literal. For fork/`lazy_fork`, # denotes the number of replications (outputs) of the input variable. For buffer, # denotes the number of buffer slots. `[#]` fields are implemented as custom assembly formats for the operations, wherein the # value is stored as an attribute of the operation. The `handshake.func` operation represents handshake modules. Apart from regular arguments, they always have a `none`-typed argument and result. These represent the data-less input and output control signals used to signal function initiation and function completion. Hierarchy is achieved through the `instance` operation which instantiates a referenced `handshake.function`.

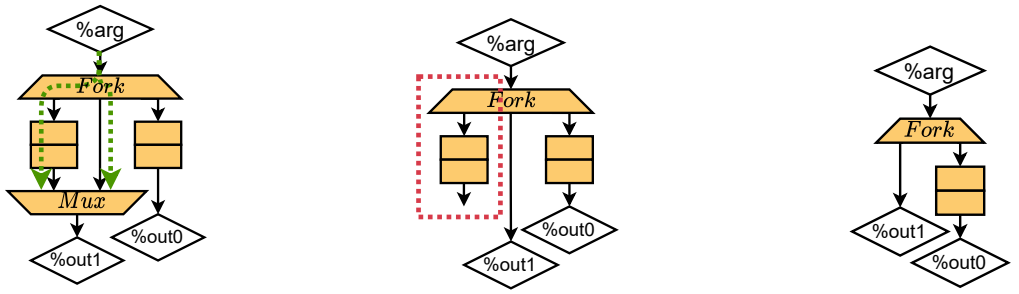
As explained in subsection 2.2.1, MLIR uses the `index` type for integer-like platform-dependent bit-width operations. Examples where `index`-typed values occur are whenever a `memref` value is indexed, or for the induction variable in `scf` loops. On software platforms, `index` values will typically be interpreted as being of CPU register width. At the time of writing, neither Handshake, nor CIRCT/MLIR in general, implement any general form of bit-width inference which could be used to assign a fixed bit width to `index` signals. As such, we keep the notion of `index` values in the Handshake dialect for things such as memory address signals.

Many of the core properties of Handshake IR are described either declaratively (through TableGen) or explicitly. This may be canonicalization rules defining a canonical IR form, or operation verifiers that define the legality of the IR. MLIR provides an infrastructure for defining pattern-based graph rewrites to drive IR canonicalization. Using these, canonicalization is performed by folding the set of canonicalization patterns on the IR until a fixed-point is reached, similar to how peephole optimizations are applied. At the time of writing, a canonical form of the handshake dialect is derived from a set of rewrite rules which:

- Eliminate simple merges/control merges/branches/muxes—operations with a single input.
- Eliminate simple forks—fork operations with a single output.

- Reduce forks—modify fork operations with a subset of its results that are unused.
- Eliminate simple muxes—mux operations with identical data inputs.

In defining peephole optimizations we use a notion of *dematerialized values*. A dematerialized value in Handshake IR is the SSA value that was the input for a sequence of fork or buffer operations—this is in line with the passes that materialize or dematerialize forks, buffers and sinks. Through this, we are able to apply peephole optimizations on both materialized and dematerialized code, and let either *dead code elimination* (DCE) or other canonicalization patterns fold to bring IR into a legal state.



(a) Mux canonicalization opportunity; green arrows represent dematerialized SSA values.

(b) With the mux removed, both the buffer and its fork input are canonicalized away.

(c) Resulting circuit after canonicalization.

Figure 3.4: Mux canonicalization example.

An example of a canonicalization using dematerialized values are mux canonicalizations, shown in Figure 3.4. This canonicalization tries to eliminate multiplexers where both values originate from the same source.

3.2.2 From Standard to Handshake IR

In this section, we will cover how a `standard`-dialect program is converted into Handshake IR. Since `standard` models a CDFG, this conversion is follows closely to the approach taken in Dynamatic [30].

First, we will consider `standard`-level transformations that will fulfill preconditions of dataflow lowering. Afterwards, we will consider the actual dataflow conversion.

Constant Pushing

In MLIR, canonicalization of constant operations will move any operation in a nested block to the entry block. This makes sense in a software context since it allows operations within all blocks of a region to reference the same constant definition, i.e. CSE. However, in a dataflow context, this presents a potential performance problem. As we shall see later, each live-in value to a block must be propagated along

with a control signal, when control is transferred to a block. This therefore requires that each live-in and live-out in a block is made explicit, instead of implicit (through SSA domination). Hence, placing constant operations at the entry block and referencing these deep within the CFG is undesired since unnecessary circuitry will have to be emitted to carry constant values along control flow edges. Instead, we seek to push constant operations into the blocks where they are referenced.

To do this, we implement a pass `-push-constants` which, through a dataflow analysis, creates constant operations within the blocks wherein constants are referenced. An example pass execution is shown in Figure 3.5.

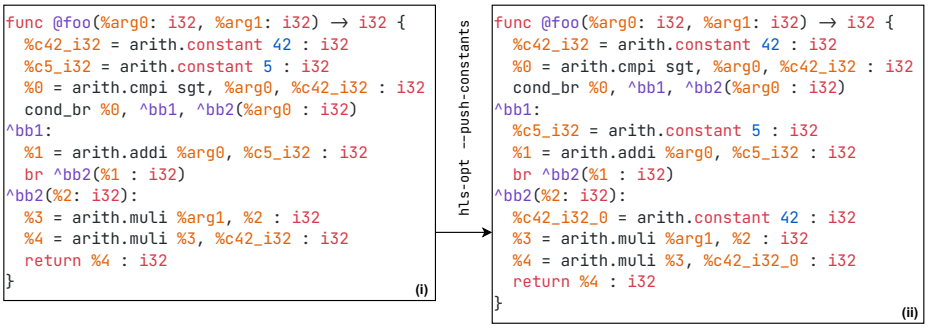


Figure 3.5: Run of the constant pushing pass. Since `%5_i32` is referenced in `^bb1`, its definition will be moved from `^bb0` to `^bb1`.

SSA Maximization

As explained above, every live-in and -out of a block has to be made explicit when converting a sequential program to a dataflow program. An example where this is not the case is shown in Figure 3.6(i); `%arg1` is not referenced in `^bb1` but referenced in `^bb2` through SSA dominance. So, when control flows from `^bb0`→`^bb1`→`^bb2` we must ensure that a handshake signal for `%arg1` is carried along this control flow path. A solution to this is to convert the program into *maximal* SSA form. Contrary to minimal SSA form, wherein SSA value definitions are moved to dominating basic blocks in order to minimize the number of ϕ-functions (or, in MLIR, the number of block arguments), maximal SSA form ensures that any value used within a block is also defined within the block, through the addition of new block arguments. In essence, this ensures that data flow is made explicit, through block arguments, rather than implicit, through block dominance.

In Figure 3.6(ii), we transformed the program into a state where any value referenced within a block is also defined within that block.

Having `standard`-dialect code in maximal SSA form is made a precondition for further dataflow conversion. The motivation for this is to make dataflow conversion less monolithic—if we can transform source programs to adhere to a precondition, then, in most cases, performing transformations on higher-level IR will be easier than at a lower level, allowing the dataflow conversion pass to be simplified in scope.

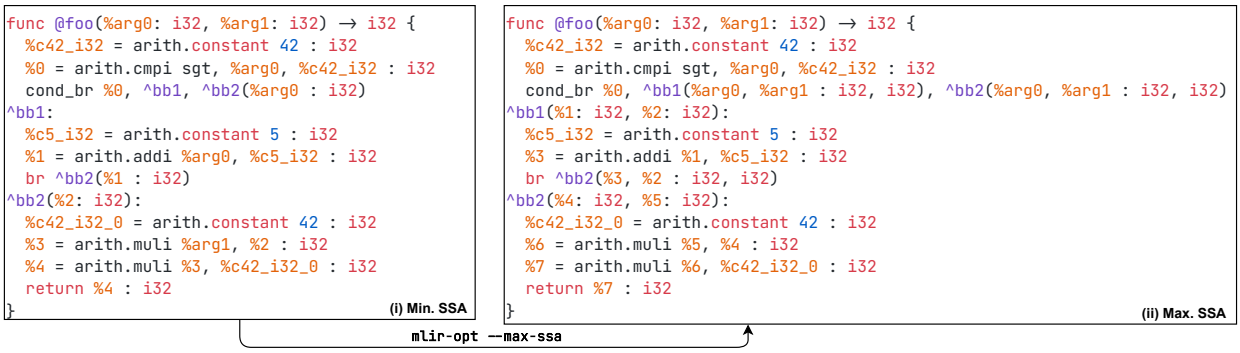


Figure 3.6: Run of the SSA maximization pass. Post execution, any value referenced within a block will also be defined within that block, through either operations or block arguments.

Our algorithm for transforming a CDFG into maximal SSA form is as follows; for each produced value v in the program, gather the set of blocks B wherein the variable is referenced. Then, $\forall b \in B$, add a block argument v' to b of type(v) and within b replace all uses of v with v'. Then, $\forall b_{pred} \in \text{pred}(B)$, recurse to b_{pred}, performing an equivalent block argument transformation as what occurred for b, having the added block argument in b_{pred} being v'_{pred}. Rewrite control flow from $b_{pred} \rightarrow b$, passing v'_{pred} mapping to v'. Each time a block is visited and its block arguments are rewritten, store this in a mapping; this mapping is used to ensure that the v replacement argument is only added once to each block for each live-in. This is relevant due to a block potentially being visited multiple times in cases of branching control flow. Recursion stops upon visiting the defining block of v.

Dataflow Conversion

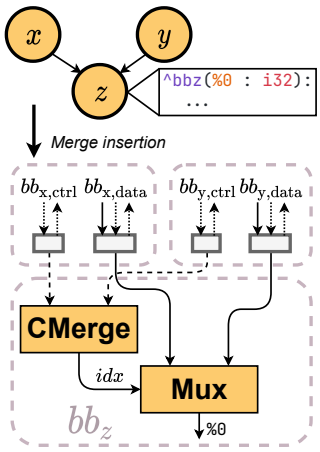
```
func @foo(%arg0: i32, %arg1: i32) → i32 {
  %c42_i32 = arith.constant 42 : i32
  %0 = arith.cmpi sgt, %arg0, %c42_i32 : i32
  cond_br %0, ^bb1(%arg0, %arg1 : i32, i32),
    ^bb2(%arg0, %arg1 : i32, i32)
^bb1(%1: i32, %2: i32):
  %c5_i32 = arith.constant 5 : i32
  %3 = arith.addi %1, %c5_i32 : i32
  br ^bb2(%3, %2 : i32, i32)
^bb2(%4: i32, %5: i32):
  %c42_i32_0 = arith.constant 42 : i32
  %6 = arith.muli %5, %4 : i32
  %7 = arith.muli %6, %c42_i32_0 : i32
  return %7 : i32
}
```

circt-opt --lower-std-to-handshake

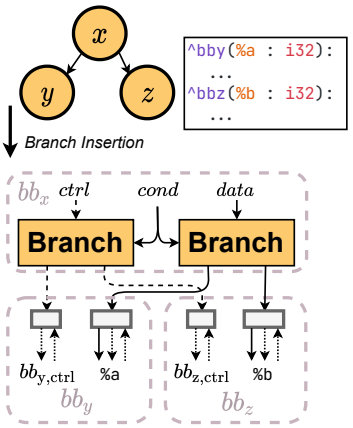
```
handshake.func @foo(%arg0: i32, %arg1: i32, %arg2: none, ...)
    → (i32, none) {1
  %0 = merge %arg0 : i32 2
  %1 = merge %arg1 : i32 2
  %2 = constant %arg2 {value = 42 : i32} : i32
  %3 = arith.cmpi sgt, %0, %2 : i32
  %trueResult, %falseResult = cond_br %3, %0 : i32 3
  %trueResult_0, %falseResult_1 = cond_br %3, %1 : i32 3
  %trueResult_2, %falseResult_3 = cond_br %3, %arg2 : none 3
// ^bb1:
  %result, %index = control_merge %trueResult_2 : none 2
  %4 = merge %trueResult : i32 2
  %5 = merge %trueResult_0 : i32 2
  %6 = constant %result {value = 5 : i32} : i32
  %7 = arith.addi %4, %6 : i32
  %8 = br %result : none 3
  %9 = br %5 : i32 3
  %10 = br %7 : i32 3
// ^bb2:
  %result_4, %index_5 = control_merge %8, %falseResult_3 : none 2
  %11 = mux %index_5 [%10, %falseResult] : index, i32 2
  %12 = mux %index_5 [%9, %falseResult_1] : index, i32 2
  %13 = constant %result_4 {value = 42 : i32} : i32
  %14 = arith.muli %12, %11 : i32
  %15 = arith.muli %14, %13 : i32
  return %15, %result_4 : i32, none
}
```

Figure 3.7: Dataflow conversion, from a `standard`-level CDFG to Handshake IR.

At this point, the `standard` level code has been optimized for minimizing live in/outs, and dataflow has been made explicit through SSA maximization. We now consider dataflow conversion. Dataflow conversion is performed through the following steps. Numbers in Figure 3.7 highlights the transformations applied by each step.



(a) Merge insertion; data inputs to a block are selected by the control merge of a block.



(b) Branch insertion; data outputs are sent to predecessor blocks based on the result of a conditional value.

Figure 3.8: Merge and branch insertion.

1. **Function conversion: (1)** The `builtin.func` operation is converted to a `handshake.func` operation. In doing so, we add a `none`-typed argument and result, representing the control input and output signals of dataflow functions.

2. **Merge insertion: (2)** Each incoming value to a block is fed through a merge-like operation (merge, mux, or control merge, respectively). For block arguments, we insert a merge operation if a block has only a single predecessor and a mux operation in case of multiple predecessors. For each block, apart from the entry block, we add a control merge operation. Merge-like operation inputs are connected to values defined in the predecessor blocks of a given block. The select value for a mux operation is connected to the control merge operation within the block. This is illustrated in Figure 3.8a. In short, control will transform from a predecessor block, the control merge will accept this transfer and output an index signal, indicating the block that transferred control. This index signal is then used in the mux operators to select operands coming from the same block. Merge insertion is made simple by the maximum-SSA precondition - any live-in to a block is ensured to be defined in the predecessor of the block.

3. **Branch insertion: (3)** Branch insertion is complimentary to merge insertion. In the `standard` dialect, block-based control flow is either through the `br` (unconditional branch) or `cond_br` operation (conditional branch). In either case, we insert a (`handshake` dialect) branch/conditional branch operation for each live-out variable and control transfer. These are the values which will be referenced by the merge-like operations mentioned above. This is illustrated in Figure 3.8b.

Constant-defining operations (`arith.constant`) will be emitted as a pair of (`handshake` dialect) constant and source operations. Optionally, constants can be emitted as connected to the control network (i.e., the entry control value of a block), ensuring that they only generate a single value for any control entry into the block. An argument for this method is that, if source operations are *not* used in a circuit, we know, that for any correct execution of a kernel, all buffers within the design are expected to be empty post-execution. This property can therefore help in debugging. In our flow, a flag can be set to enable this form of constant emission. Emitting constants as source-connected is motivated by the possibility of improving the resulting hardware size. This is due to the reduction in fan-out of the input-control fork of the block and by the fact that the structure of a source-connected constant operation will reduce to a tiny circuit in hardware.

At this point, only arithmetic operations remain to be considered. In Handshake IR, we use the `arith` dialect as a representation for dataflow arithmetic operators. This is possible since the Handshake IR assumes that any SSA use-def relation implicitly has Handshake semantics. Therefore, for an operation `%0 = arith.addi %arg0, %arg1 : i32`, it is implicit that values `%0`,`%arg0`,`%arg1` all represent handshake bundles of data, ready, and valid signals.

At this point, we have now described dataflow conversion in the absence of memory operations. A promise of dynamically scheduled dataflow circuits is the ability to, at runtime, resolve data dependencies through memory which might occur when e.g. executing a loop body that accesses memory. In Dynamatic, this is resolved by the use of a *load-store queue* (LSQ). This LSQ connects to the memory operations and block input controls of the lowered dataflow circuit. Through this, the LSQ will stall the

execution of memory operations in the circuit in order to resolve memory hazards. Through such an LSQ, we are then able to pipeline such loops without having subsequent loop iterations dependent on the finishing of prior iterations.

In this flow, we do not implement support for such LSQs, but instead implement a simpler scheme that ensures correctness of execution of memory operations, but prevents pipelining.

Let us now see how memory operations are converted; consider Figure 3.9:

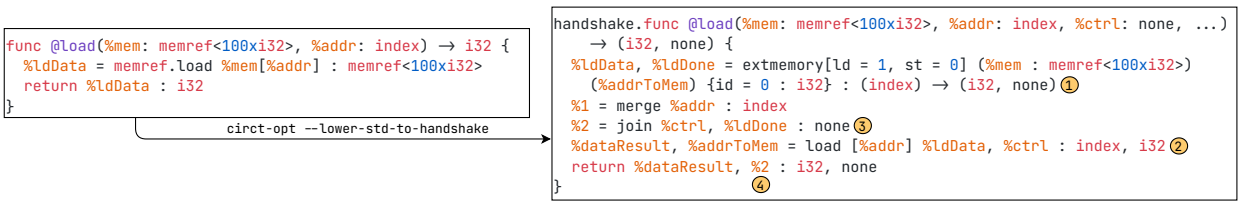


Figure 3.9: Dataflow conversion with memory operations.

This program takes a `memref` argument `%mem`, an `index` `%addr`, loads a value from `%mem` and returns this value. In the memory-accessing case, two additional steps are performed:

1. **Memory insertion (1):** For each memory accessing operation in the IR, we count up the number of load and store operations to the `memref` which they reference. Then, we insert a memory operation with the given number of load and store ports. In the case of referencing a `memref` value defined as a function argument (i.e. a pointer argument in C), we emit an `extmemory` operation. In the case of referencing a `memref` value created by a `memref.alloc`, we insert a `memory` operation (internal memory). Both of these operations have a similar interface regarding the order of input and output operands.

2. **Memory-accessing operation conversion (2):** For any memory accessing operation, we connect it to the `memory` operation that was created from the `memref` value that it referenced. Given the absence of an LSQ, ensuring correct execution requires the assumption that all memory accessing operations may alias. A simple method of ensuring correct execution is therefore to guarantee that any memory accessing operation has to complete before control can be transferred. Each load/store port of a memory will have a *done* signal, which indicates completion of the memory operation. In the order which they appear in the source program, we connect such `done` signals to the `go` inputs of subsequent memory operations. For the first memory operation in a block, we connect it to the entry control value of the block (`%ctrl` in Figure 3.9).

3. **Join insertion (3):** To coordinate control exit from a block, we insert a join operation with operands being the incoming control value of the block (`%ctrl`)—checking that control flow has entered the block—and the `done` signal of the final memory operation—checking that all memory operations finished. Note that this implicitly prevents loop pipelining when memory operations exist in the body of a loop, since control transfer to subsequent loop iterations is dependent on memory operation completion.

4. **Block exit control: (4)**: The above join insertion modified which control value that defines the finishing condition of a block. As seen in Figure 3.9(4), we then rewrite the program to use %2 to signal kernel completion. Block output control values are in Figure 3.7 defined as e.g. the `%8 = br %result : none` operation.

Fork/Sink Materialization

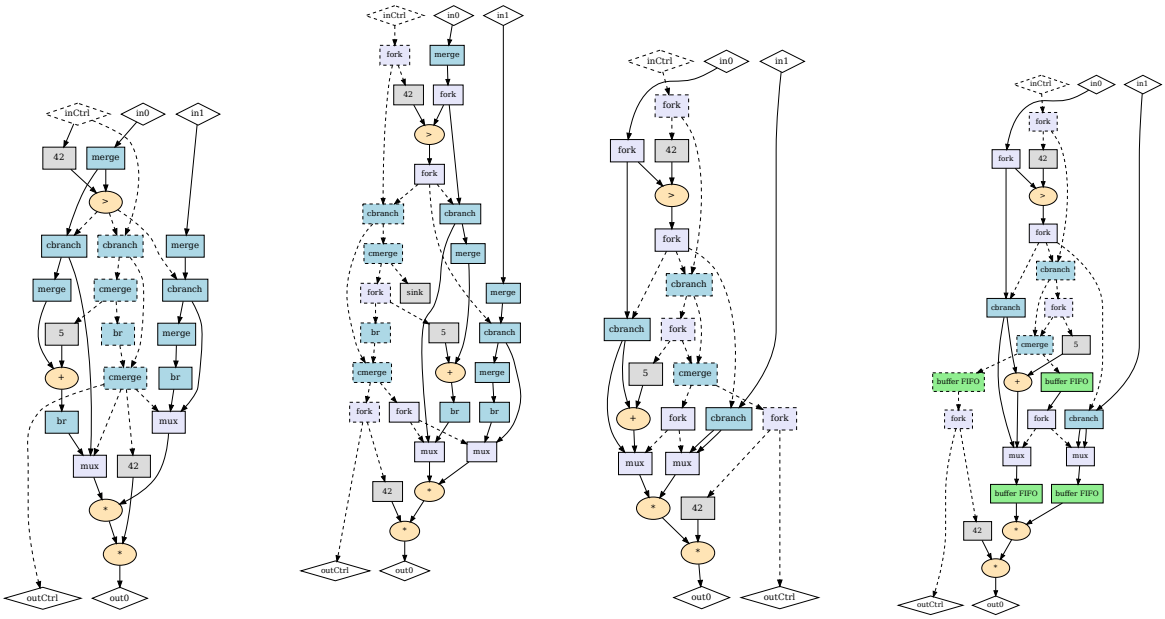
In preparing Handshake IR for hardware lowering, we must make the IR adhere to a constraint of every SSA value being used *exactly* once. By doing so, the hardware lowering does not need to consider, in the never-used case, how to tie off unused signals (which, if not done, could result in undriven wires), and, in the used-more-than-once case, the possibility for wires with multiple drivers. To fulfill this constraint, we implement a fork and sink materialization pass `-handshake-materialize-forks-sinks`. For every value defined in the IR, if it is used exactly 0 times, a sink operation is inserted, referencing the value. For every value referenced N times, $N > 1$, a fork operation is inserted with N outputs, and each use is replaced with a different fork output.

Buffer Insertion

In general, the dataflow components used in this flow are implemented as combinational logic. For correctness, it is therefore required that loops in the dataflow network are broken by sequential buffers, to avoid combinational loops in the generated hardware [20].

The `-handshake-insert-buffers` pass will place a buffer on every merge-like operation in the dataflow graph. If the operation is within an unbuffered loop, a sequential buffer is inserted, breaking combinational loops. Else, the operation will be buffered by a transparent FIFO. Thus, we still ensure buffering, but avoid a possible latency increase due to the non-transparency of sequential buffers.

Our buffer placement strategy only concerns itself with correctness of execution. For increased performance, advanced buffer placement techniques such as ILP (*integer linear programming*)-based methods, for placement and sizing of buffers [32], can be used. In this, sequential buffers are placed to cut critical paths to both increase f_{max} as well as pipeline the circuit, and transparent FIFO buffers can be used for managing backpressure.



(a) IR of Figure 3.7 (b) Fork/sink materialization (c) Canonicalization (d) Buffer insertion

Figure 3.10: Progressive dataflow lowering. Dotted edges represent control-only signals.

Figure 3.10 shows how the handshake IR of Figure 3.7 is progressively lowered. Figure 3.10a is a visualization of the source IR. Figure 3.10b materializes forks and sinks. Figure 3.10c runs handshake canonicalization, removing redundant merge and branch operations. Figure 3.10d inserts buffers.

3.3 Progressive Hardware Lowering

Having a dataflow representation of a program, we now turn to converting it into a hardware representation. Prior to this work, significant progress has been made on converting a large portion of the Handshake operations into FIRRTL. However, these conversions remained largely untested at the RTL level. This work has contributed to the Handshake-to-FIRRTL hardware conversion through rewriting of various component lowerings, guided by RTL simulations, adding support for new components, and reworking name generation for components, ports, and internal signals to facilitate debugging. The choice of the Handshake dialect lowering to FIRRTL—as opposed to the CIRCT RTL dialects—is partially historic. During the inception of the Handshake dialect, the RTL dialects of CIRCT were still in their infancy and did not provide a level of abstraction equal to that of FIRRTL. Specifically, when lowering the Handshake representation into hardware, having features such as bundle and flippable types greatly simplifies how to specify component interfaces. Furthermore, FIRRTL provided a memory abstraction suitable for lowering Handshake memories, which can have an unbounded amount of load and store ports. We lower Handshake operations to FIRRTL by programmatically building a FIRRTL module for an operation, instantiating it in a top-level module, and connecting these based on the use-def chains of

the source Handshake IR. Every such FIRRTL module is specialized based on operand and result widths of the source Handshake operation, and whether the operation is control-only. In CIRCT, FIRRTL is a monomorphic IR, meaning that we do not have the option of describing a generic FIRRTL component in textual IR and then instantiating a specialization of it based on the data width(s) of its input and output signals. This is a possible weak point of our implementation and possibly the current capabilities of MLIR. We wish to do a form of "template" lowering, meaning that we do not have a 1:1 mapping from one operation in dialect A to one operation in dialect B, but instead require emitting a large, but well-defined, set of operators from dialect B. What is needed is a method for writing textual IR with generic values, which can be referenced inside a lowering pass to specialize the templated IR. We will return to this point in section 7.1.

An alternative to our current approach is to map Handshake operations directly to instantiations of existing RTL designs—this is the approach taken in Dynamatic. A benefit of this is the ability to write generic implementations of the dataflow components, which most HDLs support, and specialize at the point of instantiation. However, by doing so, we create a dependency on an RTL design external to the CIRCT flow, making us unable to benefit from optimizations within the FIRRTL dialect and the RTL dialects that FIRRTL lowers to.

Lowering `index` Types

As explained in subsection 3.2.1 the Handshake dialect uses `index` typed values in places such as memory operation addresses and multiplexer select signals. While FIRRTL provides bitwidth inference, initial experiments with using this for deducing the width of `index` typed values in lowered Handshake components have proven unsuccessful. Instead, we conservatively emit all `index`-typed values with a fixed-width. We find that synthesis tools can properly identify and narrow `index`-typed signals when connected to internal memories—memories that are contained within the design and described as a FIRRTL memory. However, when these `index`-typed signals are involved in arithmetic or to external memories, width-narrowing is less successful. In evaluating our flow, we investigate the influence of the width of these `index`-typed signals, see chapter 5.

Example: A Dataflow Multiplexer

To provide insight into the implementation of dataflow components, below we present how a dataflow mux is implemented in FIRRTL. For any other component, our implementation follows directly from those of Edwards et al. [10, 20] and Dynamatic [31]. Any remaining operations (e.g., `arith` operations), are implemented as unit-rate actors.

Figure 3.11 shows how a kernel `df_mux` in Handshake IR is lowered into a FIRRTL module. First, we consider the FIRRTL module that encapsulated the mux component. Figure 3.11(ii) shows the signature of a two-input 32-bit mux. Note that each input and output port are defined as FIRRTL bundles. In FIRRTL IR, this allows us to emit `firrtl.connect` operations on flip-

Figure 3.11: Handshake to FIRRTL lowering of a kernel containing a 2-input multiplexer.

compatible types, instead of having to connect each sub-signal within each bundle. The module name (`handshake_mux_in_ui1_ui32_ui32_out_ui32`) serves as a de-facto type specifier for the component. During lowering, we create module names for each dataflow component that includes the operator name, input and output port widths, and if the component is control-only. Given this unique module name, in cases where other operators require an equivalently specialized mux component, we perform a symbol lookup in the symbol table of the FIRRTL program for the component name, and in the matching case, instantiate the module. Figure 3.11(iii) shows how the top-level module `firrtl.module @df_mux` is defined. Each input and output argument is specified as a bundle of `ready`, `valid` and `data` signals, with the `none`-typed arguments being only a `ready`,`valid` signal pair. Additionally, `clock` and `reset` input signals are added. The body of the module consists of dataflow component instantiations and connectivity. Here, we see how bundle types allow us to directly connect the input and output arguments of the top-level module and the instantiated `@df_mux` module.

Consider Figure 3.12, showing the internal structure of the mux component - the FIRRTL IR of this has been omitted for brevity. For a multiplexer, the output is valid whenever the select signal is valid and the selected input is valid:

$$\text{out}_v = \text{sel}_v \wedge \text{in}[\text{sel}_d]_v \tag{3.1}$$

The readiness of the select signal and selected input signal is asserted whenever the output is transacting:

$$\text{sel}_r, \text{in}[\text{sel}_d]_r = \text{out}_v \wedge \text{out}_r \tag{3.2}$$

This therefore synchronizes the validity state of the multiplexer inputs with the readiness of its output.

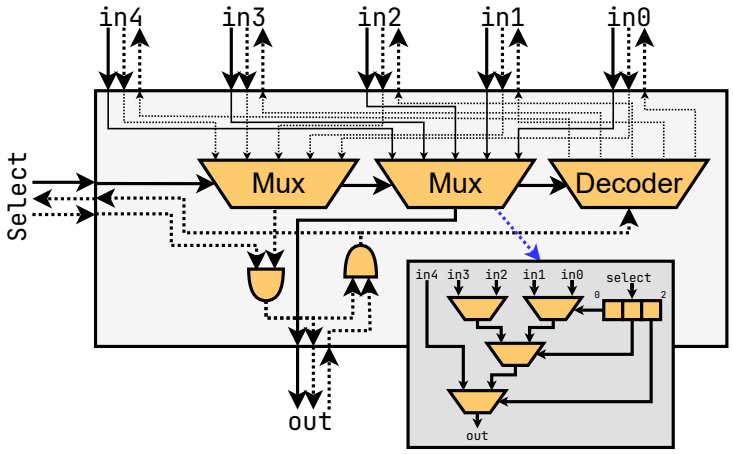


Figure 3.12: Implementation of a 5-input dataflow mux.

HDL Emission

```
module handshake_mux_in_ui1_ui32_ui32_out_ui32(
  input select_valid, select_data, in0_valid,
  input [31:0] in0_data,
  input in1_valid,
  input [31:0] in1_data,
  input out0_ready,
  output select_ready, in0_ready, in1_ready, out0_valid,
  output [31:0] out0_data);

  wire _T = (select_data ? in1_valid : in0_valid)
        & select_valid;
  wire _T_0 = _T & out0_ready;
  assign select_ready = _T_0;
  assign in0_ready = ~select_data & _T_0;
  assign in1_ready = select_data & _T_0;
  assign out0_valid = _T;
  assign out0_data = select_data ? in1_data : in0_data;
endmodule
```

```
module df_mux(
  input arg0_valid, arg0_data, arg1_valid,
  input [31:0] arg1_data,
  input arg2_valid,
  input [31:0] arg2_data,
  input arg3_valid, out0_ready, outCtrl_ready, clock, reset,
  output arg0_ready, arg1_ready, arg2_ready, arg3_ready,
  output out0_valid,
  output [31:0] out0_data,
  output outCtrl_valid);

  handshake_mux_in_ui1_ui32_ui32_out_ui32 handshake_mux0 (
    .select_valid(arg0_valid), .select_data(arg0_data),
    .in0_data(arg1_data), .in1_valid(arg2_valid),
    .in1_data(arg2_data), .out0_ready(out0_ready),
    .select_ready (arg0_ready), .in0_ready(arg1_ready),
    .in1_ready(arg2_ready), .out0_valid(out0_valid),
    .out0_data(out0_data)
  );
  assign arg3_ready = outCtrl_ready;
  assign outCtrl_valid = arg3_valid;
endmodule
```

Figure 3.13: SystemVerilog lowering of a kernel containing a 2-input multiplexer.

After FIRRTL lowering, we rely exclusively on existing hardware lowering infrastructure within CIRCT, and no modifications to this have been performed through this work. Progressively lowering from FIRRTL to SystemVerilog takes the following steps:

1. **Type lowering**: Complex types, such as bundle and flipped types, are lowered to ground types. In the case of module ports, flipped subtypes will be added to the module outputs and non-flipped to the module inputs. The inverse is done in the case of output bundles.

2. **Module inlining**: Modules annotated for inlining are inlined at the point of instantiation, and dead module elimination is performed. This pass can greatly increase the number of peephole optimizations that may apply.

3. **Inter-module constant propagation**: Constant propagation into module instantiations, as well as DCE cleanup.

4. **HW lowering**: Lowering to a combination of the RTL dialects; `comb` is used for combinational logic, `SV` for sequential logic, and `HW` for module definitions and instantiations.

CSE and canonicalizations are run in between all the above steps, given that new opportunities for optimization may have been exposed after an IR transformation. At this point, the program is in a state suitable for ingestion by the ExportVerilog tool (see subsection 2.3.1), which translates the program into SystemVerilog.

Considering the `df_mux` FIRRTL module shown in Figure 3.11, Figure 3.13 shows this module lowered to its SystemVerilog representation.

Chapter 4

A Testable and Debuggable HLS Infrastructure

With the ability to generate Handshake circuits, we now turn to the question of how to test and debug these circuits.

The goal of HLS is to raise the level of abstraction at which hardware is described, increase the productivity of a designer, and transfer the burden of RTL-level correctness onto the HLS tool. However, just as it is desired to describe behavior at a high level of abstraction, it is equally desired to verify that behavior through high-level tests. Writing RTL testbenches can be cumbersome and requires specialized hardware knowledge. As such, what we are looking for is a method of *cosimulation*—running a high-level test against an RTL-level simulation. Work on cosimulation generally takes the approach of using transactors [3, 58]. A transactor is a component that facilitates domain crossing. In our case, between a testbench, with sequential software semantics and an RTL simulation, with hardware and handshake semantics. Such a transactor will translate the concept of a function call (in the software world) into exercising the RTL interface of a kernel. This includes things such as asserting input signals (passing arguments), enable signals (calling a function), as well as accepting kernel outputs (return values).

When designing a cosimulation infrastructure, we need to consider how an RTL simulation will be called from the software world and how the software itself will perform validation of the RTL simulation.

A cosimulation testing infrastructure is a convenient tool for figuring out `when` things go wrong. However, since such a tool primarily considers interactions at the interface between the software and the RTL simulation, we rarely gain insight into `what` went wrong. This is where debugging tools come in handy. In section 4.4 we describe HSdbg, a tool for debugging RTL simulations of Handshake accelerators, which has been used extensively throughout this project.

4.1 Transaction-based RTL Simulation

To use a software testbench with a hardware implementation of a kernel, we must consider how to translate between the software domain of the testbench, and the hardware domain of the kernel. Furthermore, just as an application binary interface (ABI) defines a calling convention in software, in hardware we use interfaces and interface properties to define how to interact with a hardware module. In our case, a handshake interface. A transactor will therefore provide compatability between any given calling interface. The use of transactors and *transaction level modelling* [8] is well explored in both HLS and SoC co-design.

Semantically speaking, when progressively lowering to hardware, function call and return semantics will be made increasingly more explicit in the IR and kernel interfaces. In this, we also consider the fact that a kernel may exist at multiple levels of abstraction, and not only at the hardware level. For instance, for the developer who only cares about a subset of an HLS flow, having the ability to simulate at a higher level abstraction can significantly reduce overall time spent on simulation and debugging. For this, we provide the `handshake-runner` tool which is an interpreter for Handshake IR. This concept is shown in Figure 4.1. Each level of abstraction can have a different simulation style which influences how a call to that kernel is performed. In our work, we also implement a transactor for the `handshake-runner`, allowing for *C-to-Handshake interpreter* cosimulation. However, we will here focus on the `C-to-RTL` case.

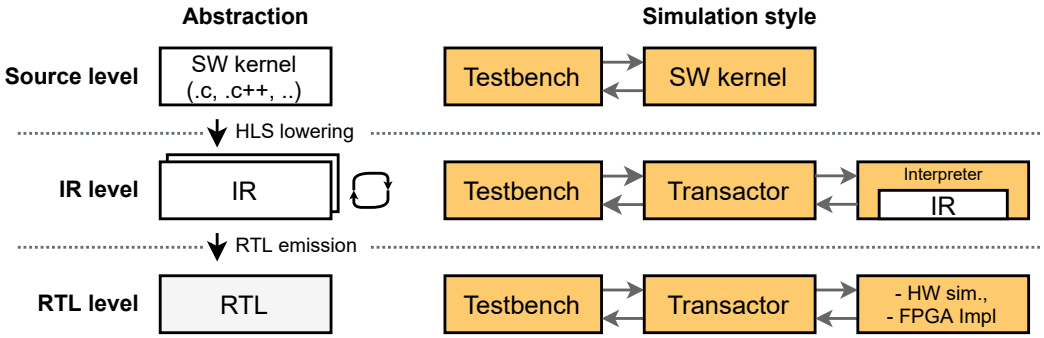


Figure 4.1: Transactors enable simulation using a single high-level testbench against kernels at different levels of abstraction.

Transactor Style

Below, we consider three design options for implementing transactors.

The first option is to encapsulate transactors in a class construct, with logic and state as member functions and values. This method is relatively simple to implement in cases where the target simulator is defined in the same language as the transactor, simplifying compilation and execution. In our use case, we perform RTL simulations through Verilator [53]. Verilator generates the RTL simulator as a C++ class, which is natively interfaced with when implementing transactors in C++.

A second option is to implicitly generate transactors through HLS compiling the testbench itself and instantiating the kernel as a submodule of the testbench. An advantage of this method is that no additional components need to perform the domain translation. The translation is implicit, in part by the lowering process, in part by the simulator itself. However, some drawbacks are that the programming constructs we would like to use in a testbench program do not necessarily make sense in HLS, and therefore make it difficult or impossible to HLS lower the testbench–an example could be dynamic memory allocation. Secondly, this style may be unusable for a tool developer in cases where the HLS flow itself may be faulty, implying that bugs can manifest themselves in both the kernel and the testbench. Finally, transforming the testbench to an RTL abstraction will inevitably slow down the execution of that testbench by possibly orders of magnitude since we are trading a software execution for an RTL simulation.

A third option is to describe the transactional domain as an MLIR dialect. A transactional dialect would allow for a scalable method of composing all kinds of execution styles, whether being software execution, hardware simulators, or interfacing with real hardware. We return to this point in section 7.1.

We implemented the first option because it is the most suitable given the scope of this work—we will only consider the case of *C-to-Verilator*. However, we believe that this work can provide insights into how a transactional dialect may be defined in the future.

4.1.1 Generating Transactors

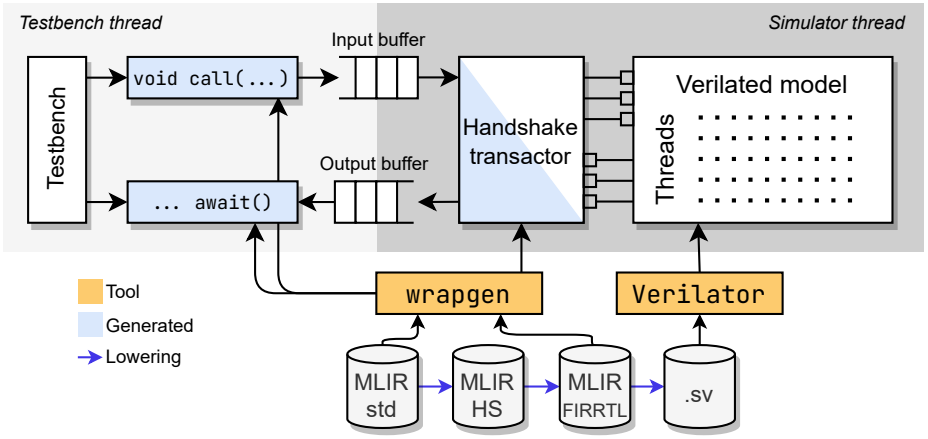


Figure 4.2: HLT overview. Call/await and a transactor specialization is generated from different abstractions of the kernel. This constitutes a wrapper around a verilated model.

Our class-based transactor approach consists of two main parts: the `wrapgen` tool and the simulator library.

The simulator library contains all logic for driving the RTL simulator. This includes concepts such as transactor interfaces, how ports are modeled and transacted, and interactions with the input and output buffers shown in Figure 4.2. This library is heavily templated, allowing for the library logic to

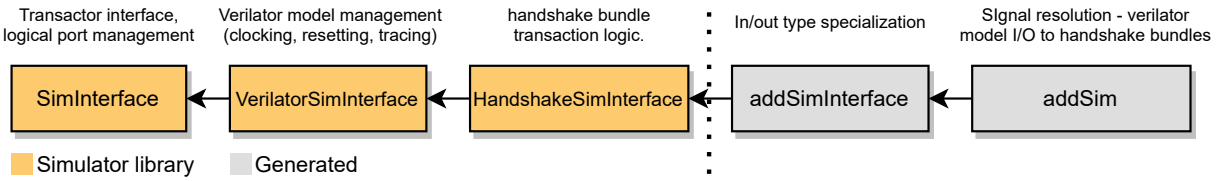


Figure 4.3: Class hierarchy of a transactor.

be specialized solely by a set of types defining the software interface of a simulator. The input and output buffers used to communicate between the transactor and testbench are thread-safe queues, and the transactor lives in a separate thread to the testbench. By doing so, we allow for concurrency between the testbench and simulator execution, for speeding up overall execution time. Transactors are implemented in a hierarchical fashion, as shown in Figure 4.3. Starting from the `SimInterface`, we incrementally specialize the transactor for the domain that it is targeting. To define a Handshake transactor implementation, first the `VerilatorSimInterface` will handle instantiation, clocking, and resetting of an RTL model. Then, the `HandshakeSimInterface` implements all logic for pushing and popping values to and from the I/O buffers to the verilated model ports by performing handshake transactions.

`wrapgen` is used for generating a kernel-specific specialization of the generic transactors exposed by the simulator library. In our case, in order to generate the transactor, `wrapgen` requires an MLIR standard representation of a kernel as well as its FIRRTL representation. The standard representation determines the software interface of the transactor, and the FIRRTL representation determines the port names and handshake bundles of the verilated model. From this, we can infer the variable names of the top-level verilator model, and hook into these from the transactor.

Let us consider how such a generated transactor may look. All code snippets shown are generated. Given function with signature `add(i32, i32) -> i32`, we run our HLS flow to generate the MLIR files required for `wrapgen`, shown in Figure 4.2. Figure 4.3 shows the transactor class hierarchy.

```
// add_wrapgen.cpp
using TArg0 = IData; // IData is a Verilator-compatible i32
using TArg1 = IData;
using TInput = std::tuple<TArg0, TArg1>; // Defines an input transaction

using TRes0 = IData;
using TOutput = std::tuple<TRes0>; // Defines an output transaction

using TModel = Vadd; // Vadd is the name of the Verilated C++ class
using addSimInterface = HandshakeSimInterface<TInput, TOutput, TModel>;
```

Figure 4.4: Input and output types are specified corresponding to that of the source (software) representation of a kernel. Using these, we define types for input (`TInput`) and output (`TOutput`) transactions.

Figure 4.4 shows how the input and output transactions (`TInput`, `TOutput`) are defined. By

defining these in a `std::tuple` we are able to use template metaprogramming to iterate across the subtypes of the tuple from within the simulator library. `addSimInterface` defines a specialization of the `HandshakeSimInterface` based on the input types (`i32, i32`) and output types (`i32`) of the source function. We also provide the type of the verilated model class (`Vadd`), required by the `VerilatorSimInterface`.

```
// add_wrapgen.cpp
class addSim : public addSimInterface {
public:
  // 'dut' is a member variable instance of the verilated class TModel
  addSim() : addSimInterface() {
    // --- Generic Verilator interface
    interface.clock = &dut->clock; interface.reset = &dut->reset;(1)
    // --- Handshake interface
    inCtrl->readySig = &dut->inCtrl_ready; inCtrl->validSig = &dut->inCtrl_valid;
    outCtrl->readySig = &dut->outCtrl_ready; outCtrl->validSig = &dut->outCtrl_valid;(2)
    // --- Software interface
    // - Input ports
    addInputPort<HandshakeDataInPort<TArg0>>("in0", &dut->in0_ready, &dut->in0_valid, &dut->in0_data);
    addInputPort<HandshakeDataInPort<TArg1>>("in1", &dut->in1_ready, &dut->in1_valid, &dut->in1_data);(3)
    // - Output ports
    addOutputPort<HandshakeDataOutPort<TRes0>>("out0", &dut->out0_ready, &dut->out0_valid, &dut->out0_data);(4)
  };
};
```

Figure 4.5: In the transactor constructor, interface ports (such as clock, reset, and handshake control signals) are resolved to those of the verilated model. Furthermore, we create the logical input and output ports that will be used for transactions.

Next, we need to create a coupling between the top-level I/O of the verilated model and the transactor, as shown in Figure 4.5. This coupling is performed within the constructor body of a specialization of the `addSimInterface` just defined, called `addSim`. For the handshake transactor, two levels of abstraction must be set up. First, `VerilatorSimInterface` needs to be informed of the `clock` and `reset` variables of the verilated model (Figure 4.5(1)). Next, the `HandshakeSimInterface` needs access to the input and output control ports of a Handshake kernel (Figure 4.5(2)). Finally, we must provide logical port definitions to fulfill the software interface of the kernel. For each input and output argument of the source function, an input or output logical port must exist in the transactor. In the handshake transactor case, for non-pointer arguments, this constitutes a bundle of data, ready, and valid variables. For pointers, this constitutes all load, store, and control ports of a memory, as seen in Figure 2.5. An input port is defined for each of the `i32` arguments to the kernel in Figure 4.5(3) and a single output port is defined for the return `i32` value in Figure 4.5(4).

```
using TSim = addSim;
using TSimDriver = SimDriver<TInput, TOutput, TSim>;
static TSimDriver *driver = nullptr
void init_sim() { driver = new TSimDriver(); }

extern "C" int32_t add_await(){
  TOutput output = driver->pop(); // blocking
  return std::get<0>(output);
}
```

```
extern "C" void add_call(int32_t in0, int32_t in1) {
  if (driver == nullptr)
    init_sim();
  TInput input;
  std::get<0>(input) = static_cast<TArg0>(in0);
  std::get<1>(input) = static_cast<TArg1>(in1);
  driver->push(input); // non-blocking
}
```

Figure 4.6: The simulator is allocated in `static` scope, and definitions of the `call` and `await` function are emitted, comprising the asynchronous kernel interface.

Finally, we need to specify a software interface for the transactor. This interface can be called from anything that links against the transactor's compilation unit. Since our transactor infrastructure decouples input and output values to a transactor via the input and output buffers, the software interface of a transactor is conveniently specified as a decoupled interface. Figure 4.6 shows the software interface of the simulator. A class `SimDriver` is introduced to handle interactions with the input and output buffers and the transactor (`TSim = addSim`). The `add_call` and `add_await` functions presents the decoupled interface to the simulator, which when combined has an equivalent interface as the source `add(i32, i32) -> i32` function. Inside these functions, input and output transactions will be assembled or disassembled and forwarded to the `SimDriver`. Calls to `driver->pop()` are blocking, meaning that `add_await` will only return once the transactor has pushed a value into the output queue. Note, therefore, that this infrastructure assumes an in-order constraint in terms of calls to the `call/await` functions and the ordering of values written by the kernel to the input and output buffers. An instance of the `SimDriver` is allocated in static scope, ensuring that the transactor (and therefore the verilated model) remains live between calls to the call/await functions.

Having now specialized the `HandshakeSimInterface` with respect to the types of the source function and hooked it up to the top-level I/O of the verilated model, we now describe how the Handshake transactor works. For each simulation step:

1. The clock is asserted (rising edge).
2. If a value is present in the input buffer and if the logical ports of the transactor are ready to accept an input, the input transaction is unwrapped, and each subvalue is pushed onto its corresponding logical input port. Input ports define validity and readiness based on the transactor. In our handshake case, this means that each logical port will inspect the ready/valid state of its associated handshake control signals.
3. The transactor then enters an evaluation loop, where the verilated model and the logical ports are continuously evaluated until a steady-state is reached. Each logical port contains propagation functions that indicate whether changes to the verilated model interface were made. If such change occurs, looping continues. This behavior is necessary because handshake hardware modules represent a Mealy machine [6], meaning that its output depends on both its current state as well as its inputs. These propagation functions are also responsible for pushing inputs to and popping

outputs from the I/O buffers.

4. We perform a check on whether an output transaction can occur. This is possible when the output control port has transacted and all logical output ports have transacted a value—these are stored in an intermediate buffer. If this is the case, we write the intermediate buffer to the output buffer, thus performing an output transaction.

5. Once a fixed point is reached, the clock is de-asserted (falling edge).

Upon a call to `driver->pop()`, the `SimDriver` will continuously request the transactor to step its simulation until an output is available in the output buffer. If no meaningful state change has occurred in the simulator within a fixed number of steps, an assert is thrown to indicate deadlock.

4.2 Source-Level Testbench Transformations

In the following sections, we consider the implementation of MLIR transformations to support the transactor interface defined in the previous section and abstractions for covalidating our RTL simulation with a software reference model.

4.2.1 Testbench Cosimulation

For any given testing infrastructure, we seek to verify the functionality of some function or device under test (DUT). This can be done either by the test writer explicitly defining test and verification vectors and comparing expected to actual behavior or through generating test vectors automatically by cosimulating against a golden model. In the case of HLS, the high-level source code from which an accelerator was generated can be such a golden model.

The proposed testing infrastructure has been implemented to support automatic cosimulation. In doing so, we seek to execute a testbench against both the golden model and the RTL simulation and verify that whatever side effects the golden model had, the RTL simulation will exhibit equivalent side effects. We define side effects of any given function call as the modifications made to mutable input arguments and the output arguments.

To facilitate program cosimulation-related program transformation, we capture this in a new MLIR dialect called `cosim`. In `cosim` we define two operations, `cosim.call` and `cosim.compare`.
`cosim.call` may replace any normal function call but with the ability of targeting multiple implementations of the same function. This is done to perform calls to a reference function, and an arbitrary number of target functions, at an exact point in the source program.[1]. The `cosim.call` operation is used in Figure 4.7(i), where a function `foo` will be used as a reference function and `foo2` as a target function.

[1] Or, more generally, any other function with an identical signature.

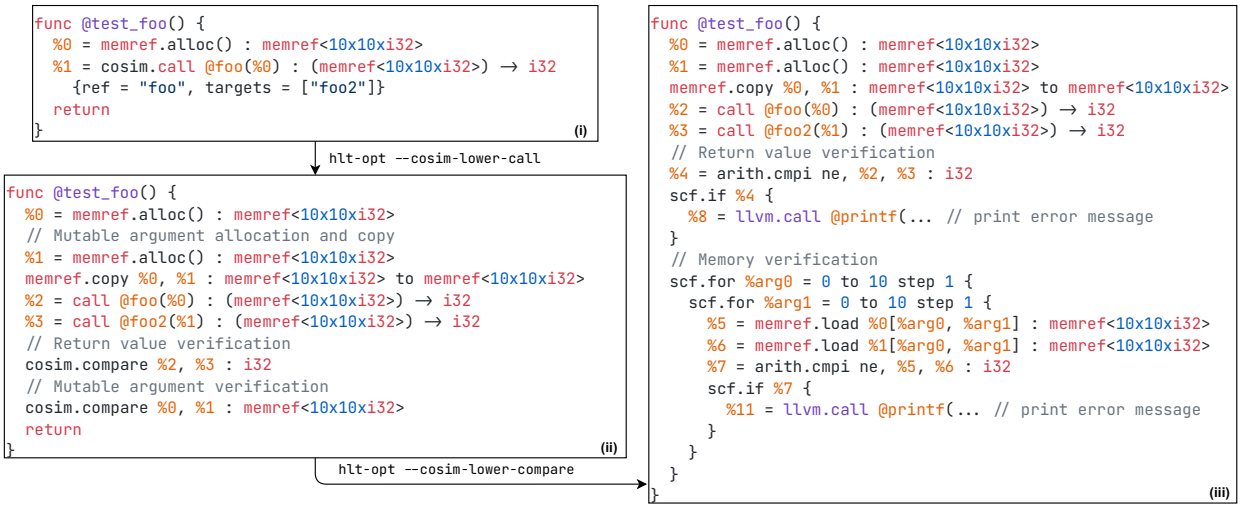


Figure 4.7: Lowering of `cosim` dialect operations. A `cosim.call` operation (i) is lowered into two function calls (ii), mutable argument copying and side-effect comparison (iii) using `cosim.compare` operations. `cosim.compare` operations are lowered into verification code in (iii), which can be further lowered to executable LLVM IR.

Having this, we then implement a lowering pass to do the following:

1. Any mutable input operand is copied for each target. Here, we only consider the case of `memref`-typed values.
2. `builtin.call` operations are inserted for each target and the reference function.
3. `cosim.compare` operations are inserted to compare the side effects of the reference function call with the side effects of the target function calls.

To illustrate this, consider Figure 4.7(ii). The `cosim.call` operation has been lowered to calls to both `foo` and `foo2`, as well as an allocation and copy of the memory that was an input argument to the original function. Furthermore, two `cosim.compare` operations have been emitted; one for comparing the mutable input argument as well as one for the output results.

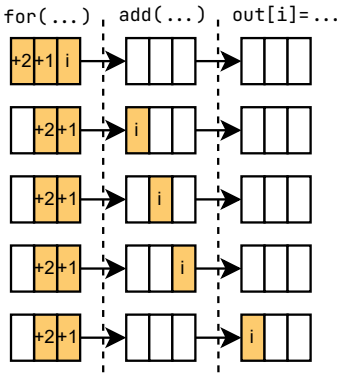
Each `cosim.compare` operation will be based on the type of its input argument and will be lowered to a set of operations comparing the argument values. We choose to implement these comparisons directly as MLIR operations, but a viable alternative is a runtime library for performing the comparisons and error reporting.

4.2.2 Desynchronizing RTL Simulator Invocations

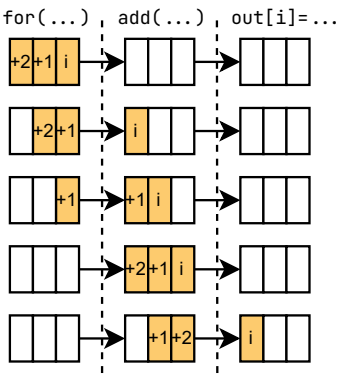
A desirable property of a cosimulation infrastructure is the ability to stimulate the interface of an RTL simulation in such a way that mimics how the kernel would be used in hardware.

```
int out[3];
for(int i = 0; i < 3; ++i)
  out[i] = add(i, i);
check(out);
```

(a) Testbench calling a kernel `add`.



(b) Synchronous value flow through `add`.



(c) Asynchronous value flow through `add`.

Figure 4.8: Synchronous versus asynchronous value flow through a kernel `add`.

As a motivating example, consider the testbench shown in Figure 4.8a, wherein a function `add` is called within a loop. If the testbench is executed *as-is*, invocations of `add` in subsequent loop iterations will be blocking, due to the dependency between the call of `add` and the write to `out[i]`. Next, imagine that `add` is an RTL model containing three pipeline stages. Given a sequentially executing testbench, this results in simulation semantics shown in Figure 4.8b. For each loop iteration, `add` must ingest the set of input parameters and execute until a value is available on its output. This presents multiple problems: First, by having only a single stage active within the hardware model at any given time, inter-stage semantics such as forwarding or stalling remain untested. Secondly, total simulation time will increase due to not exploiting pipelining within the simulated model.

If we instead consider breaking the dependency between returning from `add` and issuing subsequent loop iterations, `add` calls within the loop are *asynchronous*. In such case, invocations of `add` are now only dependent on the simulation model being ready to accept inputs, and multiple invocations of `add` may be active at once (Figure 4.8c). Not only does this mimic how the pipelined kernel would be used in hardware, but it may also significantly speed up overall test execution due to the overlapping of testbench and simulation execution.

The testing infrastructure proposed in this work is designed under the assumption that a designer, from their testbench, expects to call a kernel synchronously. An alternative to this could be to require the designer to write a testbench with asynchronous behavior. In such a model, separate function calls for invoking an RTL model and waiting for its results would have to be made. We see this style as unfavorable due to it introducing an API and behavioral mismatch between the kernel's source code, which we expect to be a normal synchronous C function. As a practical consideration, this would also require the rewriting of existing designs, benchmarks, and test suites with which we might want to use our HLS infrastructure. This, therefore, poses the problem of wanting to call an RTL simulation asynchronously but simultaneously wanting to write synchronous testbench code.

The process of desynchronization—converting synchronous programs into asynchronous—is a complex problem [4] but highly relevant in the world of modern compute, since efficient use of (task-

pipelined) hardware accelerators relies on communicating through decoupled interfaces. Maintaining program semantics while decoupling a function invocation from the use of function results is difficult given the complexities of disambiguating runtime memory dependencies and tracking side effects between the caller and callee. In practice, we observe that many modern programming languages implement asynchronous features through concepts such as subroutines, futures, or channels, but still leaving it up to the user to apply these rather than the compiler.

Due to the problem presented above, we do not intend to solve the general problem of desynchronization within MLIR. Instead, we seek to implement a restricted form of desynchronization that can be used for most testbench-like programs. We observe that most testbenches follow simple programming patterns, with computation split into the generation of test stimulus data, calling a kernel, and verifying the side-effects of a kernel.

We implemented a desynchronization pass based on the above assumptions. These properties are assumed true for the testbench program, and we only implement non-exhaustive checks to verify them, catching the most blatant violations. If necessary, these assumptions could be verified through more elaborate program analysis.

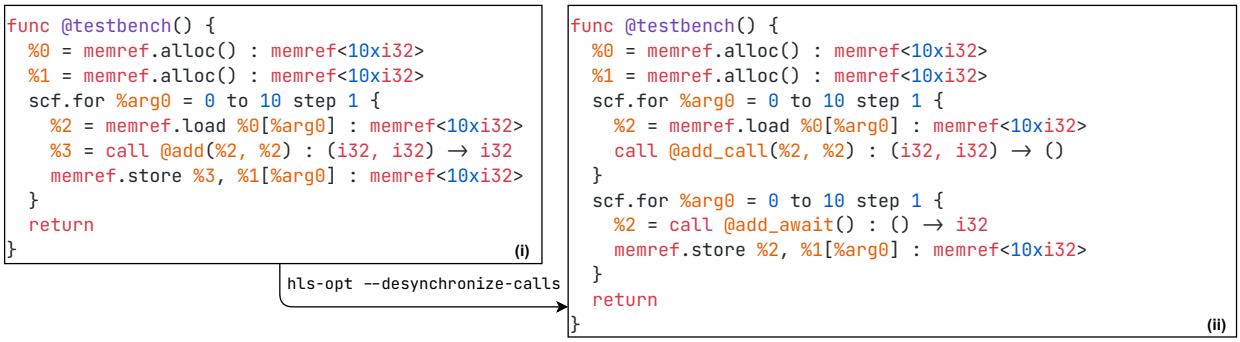


Figure 4.9: (i) shows a function with a call to `add`. (ii) shows the desynchronized version, with loops emitted for the up- and downstream operations of the function call. Note that `add_call` returns no results and `add_await` takes no arguments.

Figure 4.9 shows how the `call @add` operation has been desynchronized. The pass supports function calls being either nested directly within the top-level function or within a loop, like the one shown in Figure 4.9(i).

To desynchronize an operation o_f we first generate two new function declarations o_{call}, o_{await} within the current module. Given a function signature `func @add(i32, i32)→i32`, `func @add_call(i32, i32)→()` declares the asynchronous invocation, and `func @add_await()→(i32)` the (blocking) await. These operations define symbols that are referenced by the asynchronous function calls. Importantly, these are only *declarations*, given that the actual function implementations are external to the MLIR module, being defined through the C-functions generated by `wrapgen`. When desynchronizing an operation o_f within a loop l_{src}, we first create two new loop operations, l_{call} and l_{await}. Then, we perform the following dataflow analysis to copy any upstream dependencies of o_f to l_{call} and any downstream dependencies (along with their upstream dependencies) to l_{await}:

- `copydeps(o)`: for each operand σ of o, if σ is defined by an operation o_σ within l_{src} then copy o_σ to l_{call} and run `copyDeps` on o_σ.
- For each result σ of o_{f}, for each user o_{user} of σ, if o_{user} is defined within l_{src} then copy o_{user} to l_{await} and call `copyDeps` on o_{user}, with operations being copied to l_{await}.

Having these, we then add o_{call} to the end of l_{call}. For the await loop, we add o_{await} to l_{await}, at the point after all operations which produces one of its operands.

The result of this transformation is shown in Figure 4.9(ii). Relative to the original `call @add` function, the upstream `memref.load` operation has moved with the call to `@add_call`, and the downstream `memref.store` has moved with the call to `@add_await`.

4.3 An End-to-End Example

In section 4.1 we saw how transactors could be implemented, creating a software interface to an RTL simulation. Section 4.2 then showed how to do source-level rewriting of a testbench in order to perform cosimulation, validation, and asynchronous execution. This section presents an end-to-end example of how these flows compose and execute.

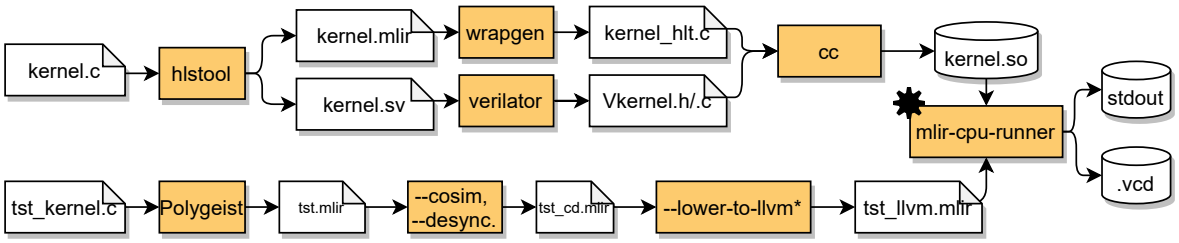


Figure 4.10: End-to-end flow for the creation of an executable, testbench driven simulation. The upper row shows kernel lowering, the bottom row the testbench lowering.

Figure 4.10 illustrates the overall flow starting from a C kernel and testbench to an executable simulation. The upper half illustrates the lowering flow for the kernel. The `hlstool` box represents the flow presented in chapter 3. From the SystemVerilog representation of the kernel, `verilator` creates the core of the RTL simulator, and `wrapgen` generates the transactor. This is compiled into a shared library, containing definitions for the `kernel_call/_await` functions.

The bottom half of Figure 4.10 illustrates the testbench lowering flow. We first convert the testbench into MLIR through Polygeist. Then, we run the `cosim` conversion pass—this pass must be run before desynchronization, since we only want to desynchronize the call to the transactor-implemented function, and not the reference function. After `cosim` conversion, we finally run desynchronization. As covered in subsection 2.2.3, many of the upstream MLIR dialects can lower to an MLIR LLVM dialect representation. The operations of the `cosim` dialect all lower to a combination of `arith` operations for comparison logic

and `scf` operations for control flow. These dialects both have existing lowering paths to LLVM, so at this point, the entire testbench can lower to LLVM.

Finally, we pass the lowered testbench (now in MLIR LLVM form) and the kernel `.so` library to `mlir-cpu-runner`. `mlir-cpu-runner` will JIT compile and execute the testbench while resolving the kernel call and await symbols to those provided in the `.so` library. Execution is then verified through a combination of the `stdout` of the run, the return code, and VCD files.

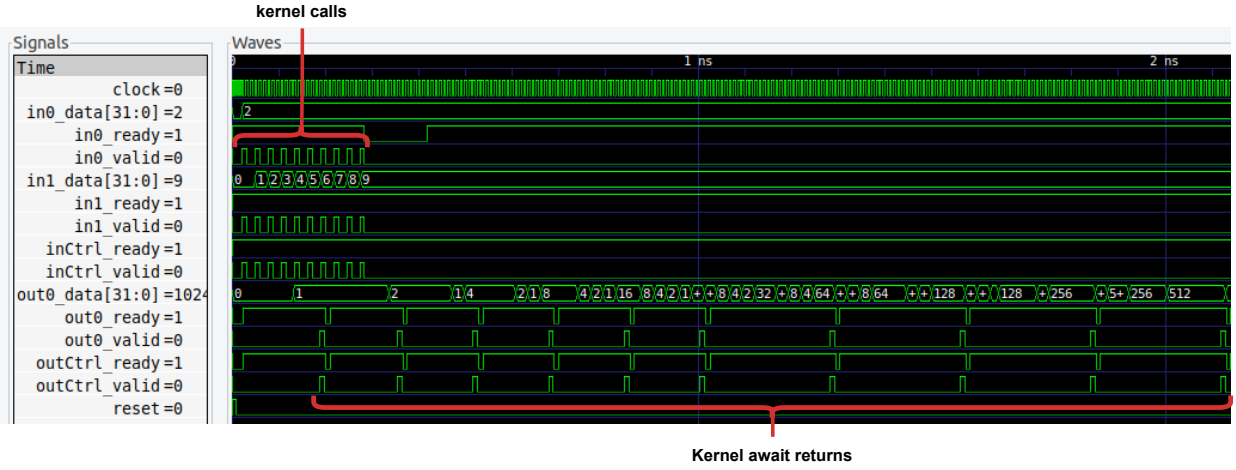


Figure 4.11: Example test execution of a kernel performing integer exponentiation. Note that kernel calls are decoupled from kernel returns.

Figure 4.11 shows one example execution of a kernel taking two 32-bit inputs and returning a single 32-bit output. The kernel is called ten times. Note the decoupling of calling the kernel from the kernel returning values. Each output value will generate a token in the output buffer, which in turn wakes up a testbench currently blocking on an `await` call.

4.4 HSdbg

Hardware debugging can be a difficult task even for designers writing their own RTL code—and debugging generated circuits maybe even more difficult. One reason is the highly hierarchical structure typically found in generated circuits. This makes sense for a tool but may make it difficult to understand what a circuit is doing due to the use of intermediate signals and components. This generally applies when generating design patterns that diverge from humanly-written designs. Secondly, generated circuits often struggle to carry meaningful names that are traceable back to the source code. This, coupled with the fact that different levels of abstraction of the circuit may have inferred signals other than one would immediately expect when providing the source program. In an HLS case, structured control flow, such as loops, is first reduced to a CDFG. Then, at the handshake level, control components will be inserted to transfer control between blocks in this CFG, and finally, at the FIRRTL level, all handshake signals are

elaborated into bundles of ready, valid, and data signals. These transformations add new information to the kernel that does not necessarily trace back to the source program.

A result is that even C programs of just a few lines of code containing control flow may elaborate to thousands of signals in a simulated VCD trace. Moreover, if VCD trace inspection is the only available debugging method, debugging becomes a tedious and challenging task.

To aid in debugging the circuits generated by our HLS flow, we have developed a tool that can be used to debug handshake circuits visually. The premise of this tool builds on the idea that debugging representations should be provided at the highest level of abstraction where a problem can be detected. In software, this is analogous to source-level debugging with the ability to step through a program line-by-line, even though the underlying system is executing machine code[2].

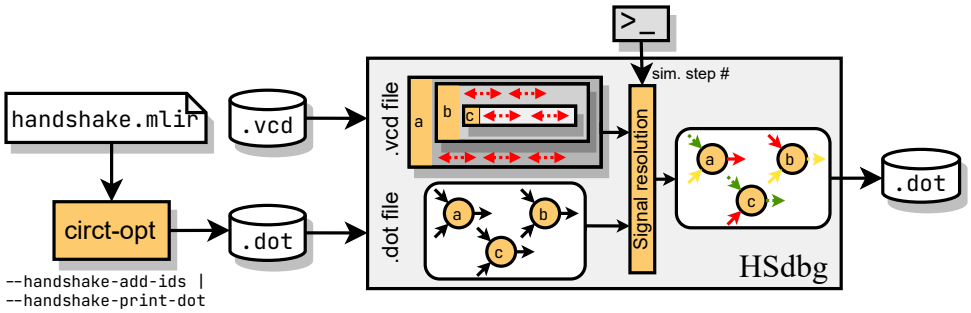


Figure 4.12: HSdbg flow.

HSdbg is a tool that can provide a visual representation of an execution trace of an RTL simulation, the overall flow of which is shown in Figure 4.12. HSdbg uses GraphViz `.dot` files to visualize a Handshake IR dataflow graph. This graph is then modified based on values read from a VCD trace to reflect the underlying state of the RTL simulation.

To do this, we first add unique IDs to the Handshake IR through a CIRCT pass. Having unique IDs allows for dataflow hardware instances to be given a unique and deterministic name. These names are then reflected in both the `.dot` file as well as the module names present in the VCD trace. The `.dot` file and VCD trace are then provided to HSdbg, wherein nodes in the graph are resolved to modules and signals in the trace. To visualize handshake state, we resolve the `${signal}_(valid/ready/data)` signals in the VCD file to the nodes and edges of our GraphViz graph. Once resolved, for any simulation step, we can then read signal values from the trace and adjust attributes of the `.dot` graph to reflect the state of the simulation.

The visual semantics of the representation are shown in Figure 4.13. Each square node represents a handshake component, circular nodes represent any other operation, and rhombus-shaped nodes represent input and output variables. Each edge in the diagram represents a handshake bundle (ready, valid, and an optional data signal for non-control connections). An edge terminates in a symbol (arrowhead, empty or full circle). The source of the edge is the component providing the `valid` and `data`

[2]Going further, stepping through assembly code is but a high-level debugging abstraction for a hardware designer who implements a processor.

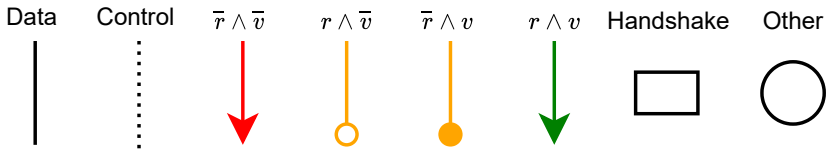
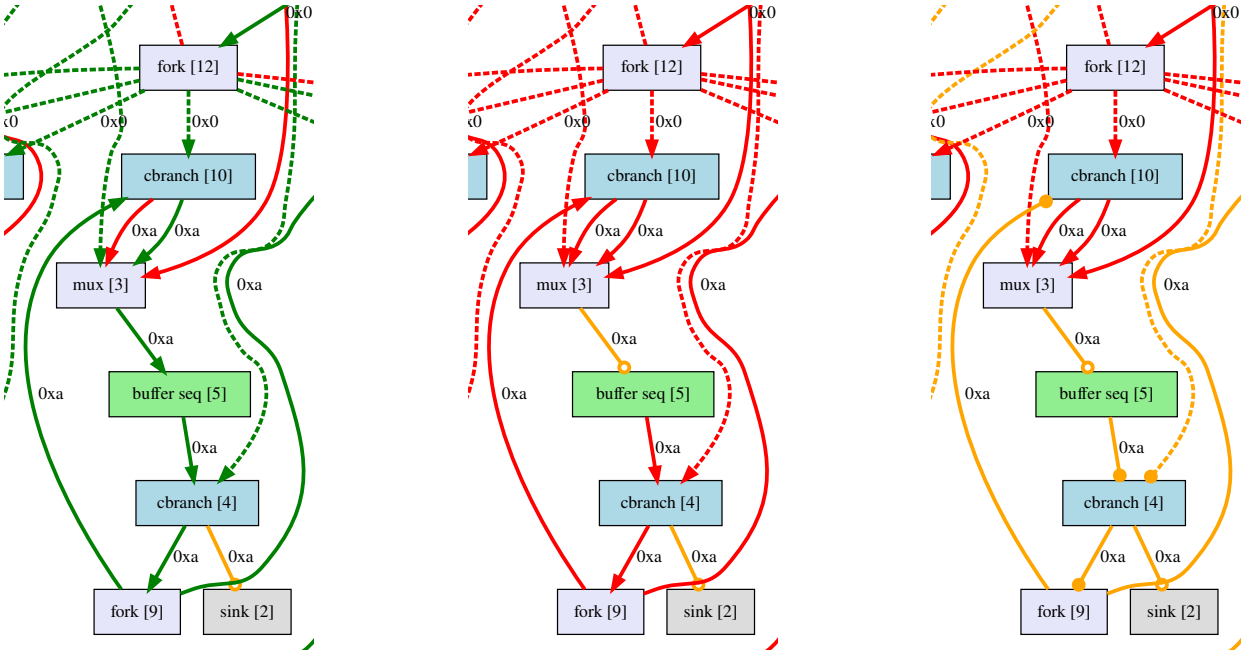


Figure 4.13: Visual semantics of an HSdbg visualization. Lines represent bundles of valid (v), ready (r) and optionally a data component.

signals. Dotted edges represent control connections. Red edges indicate that a signal is neither ready nor valid, yellow indicates ready or valid, and green indicates ready and valid. The terminating symbol of the edge indicates, on an empty circle, that the bundle is ready, and full circle, that the bundle is valid. Data signals will display the current value near the edge. Values can be shown in either hex, decimal, or binary.

The debugger is controlled via a command-line interface, with right and left arrow presses, stepping back and forward in simulation time in the VCD, as well as a `goto` option, jumping to a specific simulation time. Figure 4.14 shows how HSdbg visualizes subsequent clock cycles in a handshake circuit.

Throughout this work, experience with the tool has been highly positive, and it has been a go-to for identifying issues related to the circuits themselves, such as deadlocks and issues with the implementation of dataflow components. This has, in large part, made VCD inspection obsolete. Such a tool may also be applicable in an educational context. Prior work has shown that employing visualization at the level of abstraction of design is a valuable tool for teaching computer architecture concepts [45]—we believe that visualizing handshake circuits at the handshake level of abstraction will help people new to dataflow circuits to get a sense for the execution model.



(a) Cycle 1 (b) Cycle 2 (c) Cycle 3

Figure 4.14: Example HSdbg visualization, showing token flow across three subsequent cycles. In cycle 1, the sequential buffer both accepts and emits a token. In cycle 2, the circuit remains idle while the value flows throught the buffer. In cycle 3, the value written to the buffer in cycle 1 is now valid on the output of the buffer.

Chapter 5

Evaluating the Dynamically Scheduled Flow

We evaluate our proposed flow, CIRCT-HLS, against Dynamatic at three levels:
First, at the front-end level, comparing Polygeist to Clang. Through this, we seek to determine whether one front-end can generate input programs more suitable for HLS than the other.
Then, at the dataflow IR level by comparing the Handshake IR generated by our flow against Dynamatics intermediate representation. Through this, we seek to gain insight into the efficacy of our dataflow lowering, as well as the impact of canonicalization.
Finally, we compare at the hardware level by comparing resource usage between the two flows.

All tests shown in this chapter have been verified to work through the proposed cosimulation infrastructure.

5.1 Front-end Evaluation

To evaluate the Polygeist front-end, we compare against Clang (LLVM 6.0-based), used by Dynamatic. The `scf/affine/standard` IR generated by Polygeist will be optimized by MLIR through the canonicalizations available for each dialect. The LLVM IR generated by Clang will be optimized by LLVM passes. We run each flow on our set of benchmarks and gather arithmetic operation and basic block counts, shown in Figure 5.1.

Figure 5.1 (top) shows the relative amount of operations emitted by Polygeist+MLIR compared to that of Clang+LLVM. Bars hitting the horizontal line indicate that both front-ends emit the same number of operations (i.e., relative amount of operations are equal), whereas bars below this line indicate that Polygeist+MLIR emitted fewer operations.

In general, we see that the two front-ends are comparable when considering arithmetic operations. Outliers here are `gaussian` and `pivot`, where Polygeist performs poorly. In the case of `gaussian`, Clang generates three `sub` operations with Polygeist generating one. For `pivot`, Clang generates one `add` and one `sub` instruction with Polygeist generating four and two, respectively. Upon inspection, we find

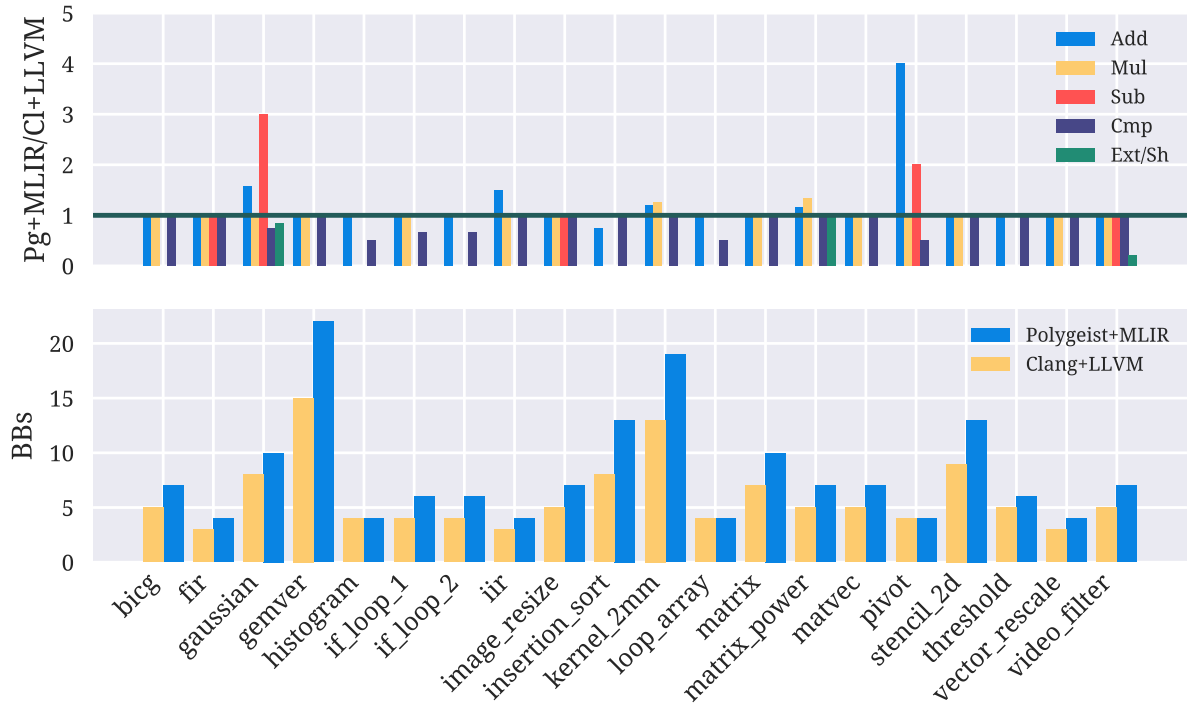


Figure 5.1: Front-end comparison. Upper graph shows arithmetic operations generated by Polygeist+MLIR relative to that generated by Clang+LLVM. Lower graph shows number of basic blocks generated by Polygeist+MLIR and Clang+LLVM.

that Clang+LLVM can reduce the number of arithmetic operations in conjunction with control flow simplifications; for instance, `gaussian` in clang uses eight basic blocks, whereas the Polygeist generated code uses ten.

Figure 5.1 (bottom) shows the number of basic blocks generated by either front-end after optimizations have been applied. Here we see that Clang+LLVM consistently performs better. Results in favor of Clang are expected due to the many years of work going into optimizing both control and dataflow in LLVM IR. For our CDFG-level code in MLIR, we have comparatively few optimizations available. In general, we seek to reduce the size of the CFG of a program since that reduces the size of the resulting control network of the dataflow circuit.

We find that each front-end produces an identical amount of memory operations (allocation, load, and store) for each program. This is important for HLS since memories are both expensive and hard to analyze.

5.2 Handshake IR Evaluation

Given a program in CDFG form, we now turn to the dataflow representation of the kernel. In evaluating this, we will consider how the dataflow lowerings and IR optimizations for our flow compares to Dynamatic. Since we do not use the same front-end for both flows, and by the fact that each front-end produces different programs as identified in the previous section, this analysis should only be seen as a first-order approximation for determining the efficacy of the flows' dataflow lowering and IR optimization.

Dynamatic circuits have been generated using its simple buffer strategy (`-simple-buffers=true`) and without LSQs (`-use-lsq=false`). By doing so, the dataflow lowering style mimics that of our flow.

Dynamatic uses GraphViz `.dot` files as an IR, to specify dataflow graphs. The Dynamatic IR is in large part identical to that of Handshake IR. A major exception is the use of `getelementptr` as an IR instruction in Dynamatic, used to model the LLVM `getelementptr` operation. This operation will be emitted in cases of access to multidimensional memories. In hardware,Dynamatic will instantiate such an operation as a flattened memory access. I.e., a three dimensional access `array[dimX][dimY][dimZ]` expands to `[i*dimY*dimZ+j*dimZ+k]` thus contributing three multiplications and two additions. Once encountered, we therefore count `getelementptr` operations as the sequence of operators they eventually expand to.

First, we compare the number of dataflow operations used in either flow. Initially, we disable canonicalizations (peephole optimizations) of CIRCT-HLS. Neither Dynamatic nor CIRCT-HLS reduces the number of arithmetic operations during optimization; thus, these remain as shown in Figure 5.1.

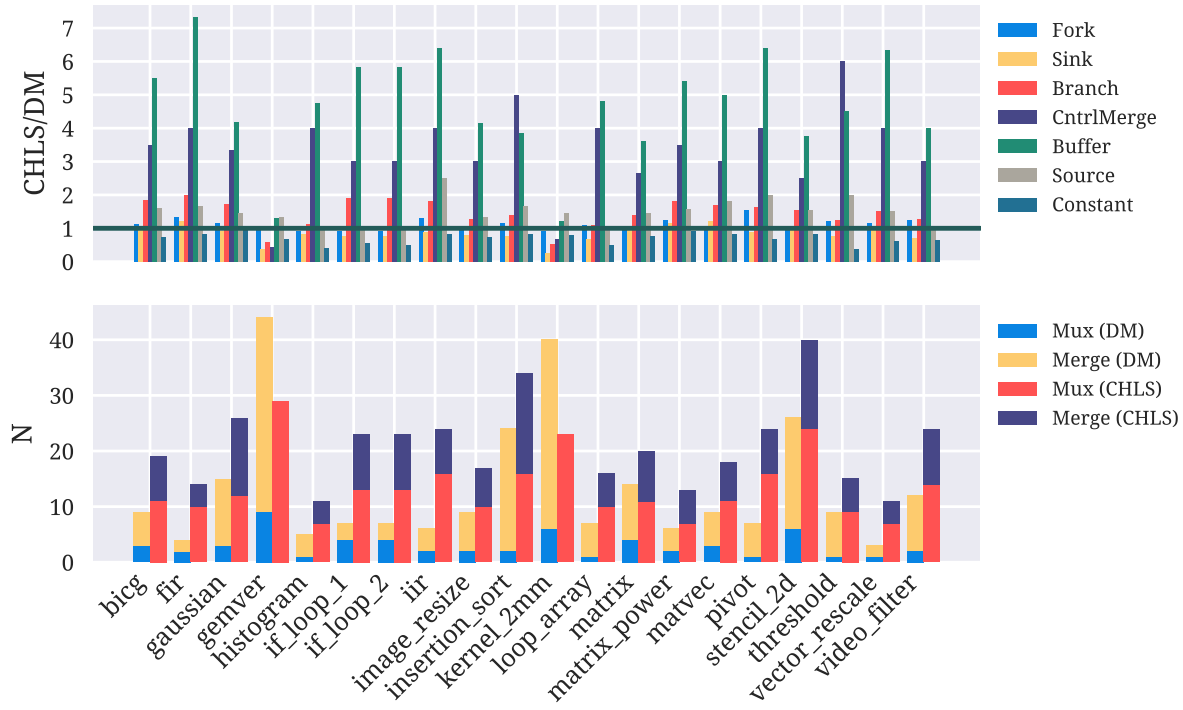


Figure 5.2: Dataflow operation usage in CIRCT-HLS and Dynamatic, with canonicalizations **disabled** in CIRCT-HLS. Top shows CIRCT-HLS usage relative to Dynamatic. Bottom shows total number of mux/merge operations for either flow.

Figure 5.2 (top) shows relative dataflow operation usage between CIRCT-HLS and Dynamatic. Each flow emits a comparable amount of fork and sink operations. The CIRCT-HLS flow is seen to emit on average close to 2x more branch operations than the Dynamatic flow. Note that this number contains both conditional and unconditional branch operations. Unconditional branch operations are used in CIRCT-HLS to indicate value transfer across an unconditional control flow boundary. As we shall see, these can be canonicalized away—we expect the Dynamatic flow to have omitted such branch operations. This is also true for ”simple” control merge operations, i.e. control merges with only a single predecessor. We see significantly more buffering in CIRCT-HLS as compared to Dynamatic. As described in subsection 3.2.2, our buffer placement method places buffers on any merge-like output. Hence, the large number of merge-like operations will lead to an equally large number of buffer operations. In general, the CIRCT-HLS flow emits fewer constants than the Dynamatic flow. We assume this is due to constants being specified as literal values in LLVM IR, which makes Dynamatic emit a constant operation for each constant literal. In MLIR, constants are SSA values and must be defined by operations. Moreover, due to our constant pushing pass, constants are CSE’d within each block, with fork operations used to distribute constant values. Note that Figure 5.3 does not capture the fan-in/fan-out of the operations. For instance, CIRCT-HLS programs emit constant operations that are triggered exclusively by source operations. In Dynamatic, constants are triggered by either source operations or through the control network. By the fact that both methods use an equivalent amount of forks, and that CIRCT-HLS on

average uses fewer constants, we can deduce that a substantial amount of fork fan-out in Dynamatic must be used to trigger constants.

Figure 5.2 (bottom) shows mux and merge operation usage between the two flows. Due to SSA maximization, CIRCT-HLS mainly relies on mux operations for data transfer across control boundaries. In cases of unconditional branches in the CFG, a simple merge will be emitted for the input values of a block. Such simple merge operations may then later be canonicalized away. In Dynamatic, we see a mixture of mux and merge operations used for the prior case.

We now enable canonicalization of the Handshake IR, resulting in the operation usage shown in Figure 5.3.

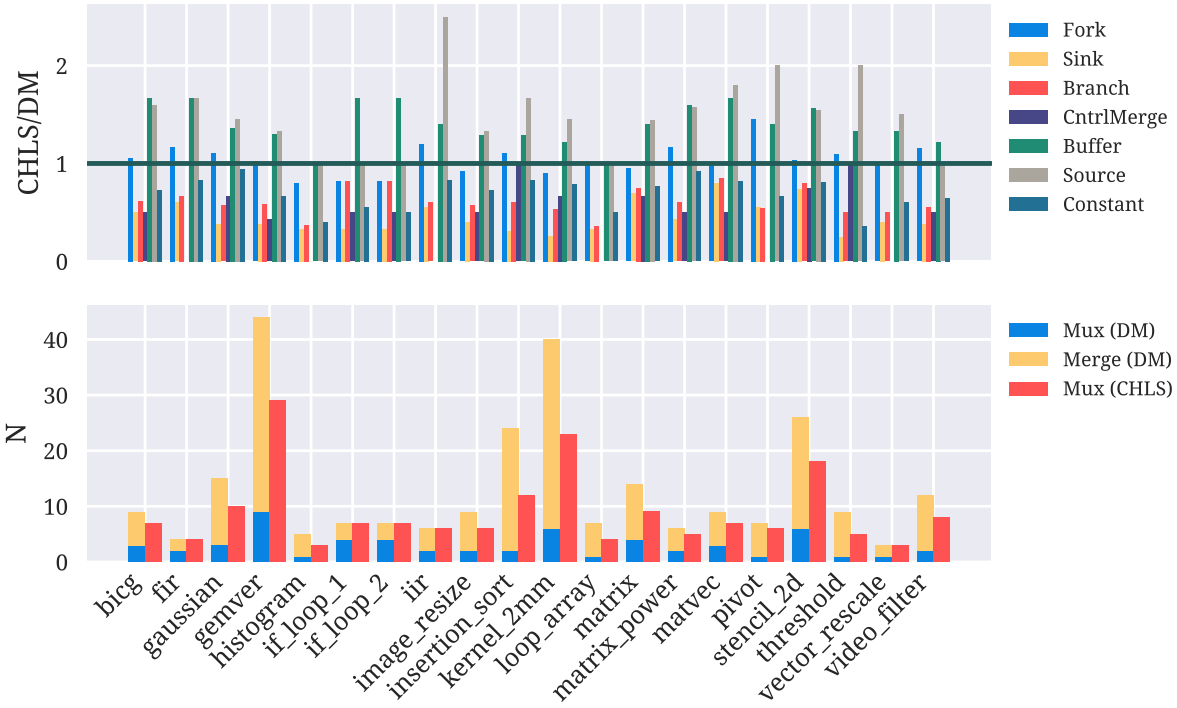


Figure 5.3: Dataflow operation usage in CIRCT-HLS and Dynamatic, with canonicalizations **enabled** in CIRCT-HLS. Top shows CIRCT-HLS usage relative to Dynamatic. Bottom shows total number of mux/merge operations for either flow.

Compared to the uncanonicalized case, Figure 5.3 (top) shows a drastic reduction in sink, branch, control merge, and merge operations. These optimization opportunities most frequently occur due to unconditional branches in the CFG—that is, blocks with a single predecessor. The reduction in merge-like operations also leads to a reduction in buffering. The uncanonicalized case showed an average of ≈5x more buffers, and we now see an average of ≈1.5x buffers compared to Dynamatic. In case of loops in the source program, which are due to the use of loop networks in CIRCT-HLS, our tool will at least replace a control merge with a mux+buffer operation in such circuits. This may also contribute to the increased

amount of buffering and reduced control merge usage that we observe.

In Figure 5.3 (bottom), we see, as expected, see that all merge operations in CIRCT-HLS have been canonicalized away. We also see that some circuits have a reduced mux usage, such as `stencil_2d`. From this, we can conclude that CIRCT-HLS consistently uses fewer merge-like operations for handling dataflow across control boundaries, than Dynamatic.

Note that join operations have been left out of the analysis. Join operations are not seen in any Dynamatic circuit. However, join logic *is* used internally for implementing operations such as unit-rate actors.

Considering Figure 5.3, we conclude that even with an inferior front-end, we can produce dataflow IR with substantially fewer operations. We again emphasize that these plots do not capture the fan-in/fan-out of the operations. This means that optimizations such as fork size reduction may have been applied, which can have a significant impact on the final circuit size.

5.3 Hardware Evaluation

Given a kernel in a dataflow IR, we now evaluate its hardware representation. RTL generated by either project is synthesized by the same `tcl` script, using the Xilinx device `xczu3eg`, and with `-flatten-hierarchy=full`, through Xilinx Vivado 2020.1. CIRCT-HLS will, at the FIRRTL level, inline all of its dataflow operator instances into the top-level module. By doing so, we allow FIRRTL to use its canonicalization passes to their full extent.

By using the simple buffering strategy in Dynamatic, neither of the flows performs any performance optimizations (latency, throughput, f_{max}). Since no technical restrictions hinder the implementation of such optimizations in CIRCT-HLS, like the advanced buffer placement techniques found in Dynamatic, we instead focus on comparing resource utilization. Resource utilization values are gathered from post-routing reports.

In section 3.3, we described how `index`-typed values will lower to a fixed width. For the sake of fair comparison, we set this fixed width to 32 bits. This is comparable to Dynamatic, which defines a maximum bit-width of 32 in its bit-width minimization pass. This, therefore, still leaves Dynamatic at a slight advantage given the *existence* of a bit-width minimization pass—something which is not yet available in CIRCT-HLS.

Running hardware lowering and synthesizing the generated RTL, we produce the results of Table 5.1.

Clock period: We generally observe a shorter clock period for Dynamatic circuits. However, due to using the simple buffering strategy—that does not consider *where* a combinational path is broken nor pipelining of combinational paths—we do not consider one flow to be superior to the other in this aspect.

DSPs: The number of DSP slices used are, identical. Considering that both flows showed comparable use of arithmetic operations—many of which map to DSP slices—at the CDFG and dataflow level, we

Benchmark	CP (ns)		DSPs		LUTs		FFs		CLBs	
	CHLS	DM	CHLS	DM	CHLS	DM	CHLS	DM	CHLS	DM
bicg	4.43	6.29	6	6	819	822	740	894	171	181
fir	4	3.69	3	3	348	460	297	548	71	93
gaussian	4.62	3.33	3	3	949	599	1287	564	205	142
gemver	8.94	4.06	18	18	2727	2916	3535	3533	639	660
histogram	4.29	4.08	0	0	265	588	295	696	52	128
if_loop_1	5.48	4.35	0	3	526	572	453	707	100	132
if_loop_2	5.69	5.28	0	0	489	579	453	669	102	127
iir	6.41	3.93	6	6	485	726	575	920	110	153
image_resize	4.34	3.51	0	0	649	490	716	555	150	111
insertion_sort	5.75	5.04	0	0	1056	1530	918	1788	214	318
kernel_2mm	5.17	4.44	12	12	2407	2156	2975	2449	543	488
loop_array	3.96	4.49	0	0	348	542	431	701	86	115
matrix	4.93	3.86	3	3	1061	667	1019	596	223	138
matrix_power	4.25	3.51	3	3	620	598	593	630	125	125
matvec	5.43	3.16	3	3	799	562	601	566	153	118
pivot	4.97	4.43	3	3	584	882	713	945	141	173
stencil_2d	6.50	3.83	3	4	1760	1039	1875	972	364	203
threshold	5.66	5.99	0	0	417	719	325	851	86	150
vector_rescale	3.93	2.87	3	3	213	330	291	409	51	71
video_filter	5.04	3.99	9	9	900	1165	1010	1250	187	253

Table 5.1: CIRCT-HLS results compared to Dynamatic. CIRCT-HLS results are with 32-bit `index`-type values.

expect to see comparable DSP slice usage. Due to the existence of a bit-width minimization pass in Dynamatic, Dynamatic should theoretically be able to narrow signals in the datapath around functional units, which may then map better to DSP slices. However, due to the similar results observed, we conclude that the bit-width minimization passes in Dynamatic are either not very effective or that the minimization achieved is not enough to impact DSP usage. We here note that the exceptions in `if_loop_1` and `stencil_2d` wherein Dynamatic uses more DSPs that CIRCT-HLS are due to the bit-width minimized operations of Dynamatic fitting into the DSPs, wherein the unminimized operations of CIRCT-HLS do not.

In comparing LUT and FF usage, we plot the CIRCT-HLS usage relative to Dynamatic, shown in Figure 5.4. Here we also plot resource usage with `index` types having a fixed width of 16 instead of 32 bits.

On average, we see that CIRCT-HLS (32-bit mode) is comparable with Dynamatic in terms of FF and LUT usage, and in many cases generate smaller circuits. However, a few outliers exist, such as `matrix`, `gaussian`, and `stencil_2d`.

For these, CIRCT-HLS performs worse due to the width of `index` signals. This can be attributed to the following: each program contains multiple nested loops (3 for `gaussian`, `matrix`, 4 in `stencil_2d`) with every loop induction variable being of `index` type. Each of these kernels perform multidimensional memory accesses using the induction variables. As such, the final datapath will use multiply-and-add

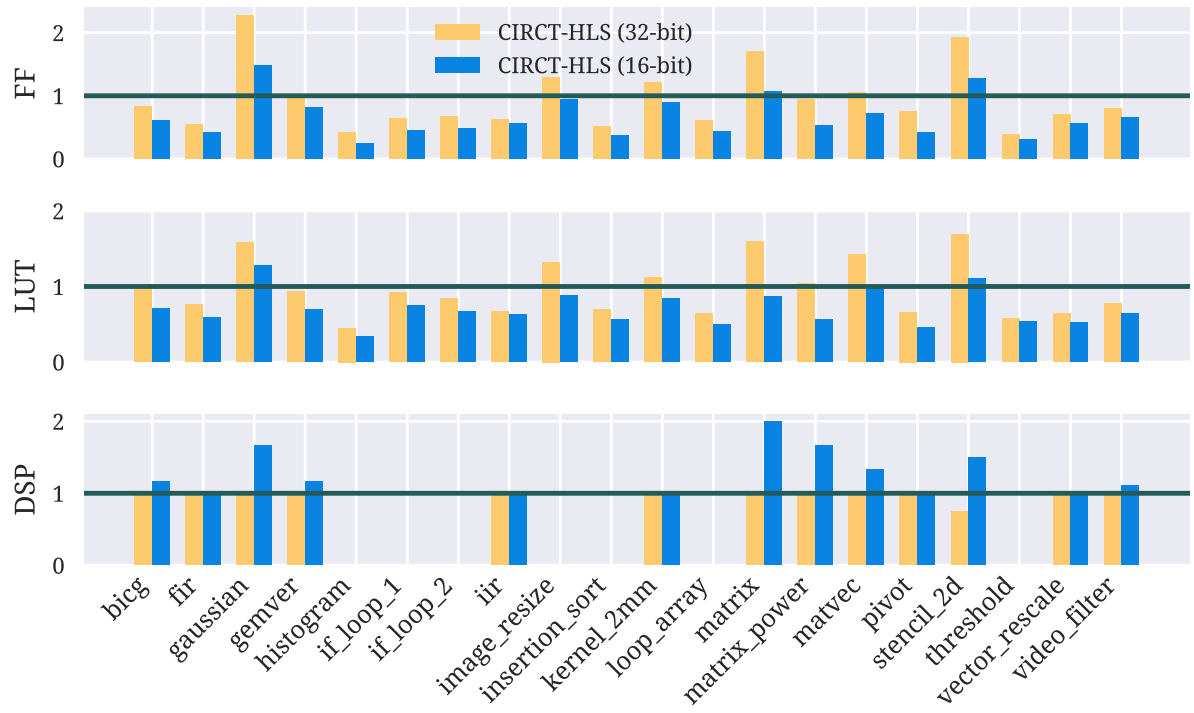


Figure 5.4: CIRCT-HLS resource usage relative to Dynamatic, with 32- and 16-bit `index`-typed signals.

arithmetic (to calculate the memory index) in the width of the `index` type. Naturally, having a significant part of the datapath being 32 bits wide will substantially contribute to overall resource utilization.

To verify that this indeed is the cause for the discrepancy, we run our tests using 16-bit `index` types, the results of which are shown in Figure 5.4 (blue). Indeed, we see a significant reduction in both FF and LUT usage compared to the 32-bit case. Furthermore, we observe an increase in DSP usage, presumably due to the now narrower `index`-based arithmetic being a better fit for the DSP slices of the target FPGA.

These results would mimic those expected if we were to have a proper bit-width minimization pass in CIRCT-HLS. Note also that the chosen width of 16-bits is more than sufficient to represent the maximum bit-width required by all of the tests. Across all tests, the largest value that must be represented by an `index`-typed signal is 900, thus only requiring $\lceil \log_2(900) \rceil = 10$ bits to represent. Since all memory sizes and loop bounds are statically known, we expect a bit-minimization pass to be able to come close to this optimal width. If so, it is reasonable to expect that the results of Figure 5.4 could improve even further.

5.4 On MLIR as an Infrastructure for HLS

While the results presented in the previous section are promising, we stress that the primary goal of this work is not strictly to provide a tool that—in its presented state—is competitive with other HLS

tools. We expect our infrastructure to evolve naturally and to eventually support many well-known HLS transformations. This will inevitably improve the performance and resource usage of the generated circuits.

Instead, we hope that a lasting impact of this work will be to provide insight into whether MLIR applies to the creation of HLS compilers. The following section attempts an opinionated review of this.

We believe the primary reason that MLIR is a good fit for HLS is the fact that MLIR IRs are *not* restricted to following the properties of a software IR. In broad terms, this can be defined as the requirement to represent the stored-program concept of Von Neumann architectures. Instead, MLIR diverges from this by providing features such as graph-based SSA representations and arbitrary structuring and hierarchy of domain-specific IRs. In this, we get a compiler infrastructure that is able to embrace stored-program representations *when needed*, but also one that fully embraces *and trusts* the IR to define its own structure, validity, and semantics.

The implications of this are first felt by the compiler engineer. By our IRs now being considered first-class citizens of the infrastructure, the ergonomics of defining analysis and transformation passes are significantly improved, compared to similar experiences with an LLVM based approach.

We also stress the ease at which our work has seamlessly integrated with the upstream MLIR dialects—for software concepts—and dialects in CIRCT—for hardware concepts. One can fear that in the process of HLS, where domain crossing is necessary, we will inevitably have to shoehorn the semantics of such domain crossing into the representations understood by the compiler. Again, this is what we see for LLVM based approaches—we either stick to LLVM IR, and thus be restricted by a software-like IR, or we diverge, write a custom IR, and in the process of doing so, lose the capabilities of the infrastructure. In our work, no tricks or hacks are needed to describe abstractions and transformations that perform this domain crossing. Contrary to when doing HLS as part of a software compiler infrastructure, MLIR allows us to focus strictly on solving HLS problems without worrying about the details of how to get inputs into the flow and how to generate HDL as an output.

Chapter 6

Task Pipelining in Dataflow Circuits

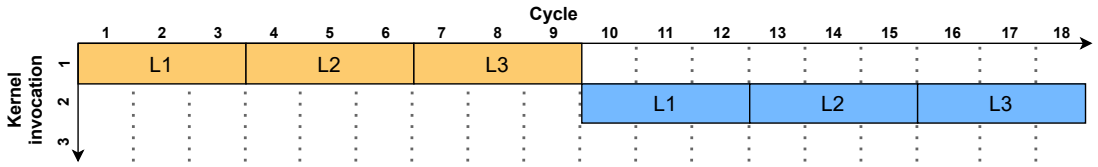
An important class of pipelining is that of *task pipelining*. In a task-pipelined kernel, multiple invocations of the kernel can be live at once, which can significantly increase the temporal use of kernel resource as well as increase throughput through a reduction of the kernel's initiation interval. Efficient task pipelining of an accelerator is important for a broad class of programs, such as filters, feedforward neural networks, and cryptographic functions.

As an example, consider the code in Figure 6.1.

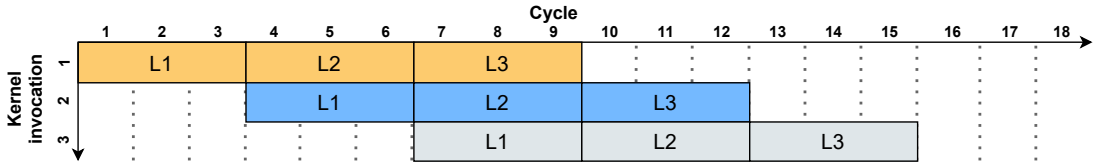
```
unsigned serial_loop(unsigned v) {
L1:for (int i = 0; i < 3; ++i)
     v = (v ^ 0x1234) ^ (v >> 16);
L2:for (int i = 0; i < 3; ++i)
     v = (v ^ 0x1234) ^ (v >> 16);
L3:for (int i = 0; i < 3; ++i)
     v = (v ^ 0x1234) ^ (v >> 16);
   return v;
}
```

Figure 6.1: The `serial_loop` function contains three loops with feed-foward dependencies (each loop and loop iteration depends on `v`).

`serial_loop` represents three loops connected in series. A feedforward dependency exists between each loop, as well as internally within each cycle of every loop. The body of the loop may be emitted as combinational logic. Therefore, given appropriate buffer placement, loops L1, L2, L3 will have an II=1 and take 3 cycles to execute, thus giving `serial_loop` a latency of 9 cycles.



(a) Non task-pipelined schedule.



(b) Task-pipelined schedule.

Figure 6.2: `serial_loop` execution schedules.

Figure 6.2 shows possible execution schedules for `serial_loop`. If a kernel does not support task pipelining, only a single control token may be live within the kernel, i.e. a single kernel invocation. This equates to the schedule shown in Figure 6.2a, with $II_{kernel} = 9$. However, if task pipelining is supported, the loops may execute in parallel, as shown in Figure 6.2b. For this we observe $II_{kernel} = 3$, thus greatly improving throughput and temporal utilization of the kernel.

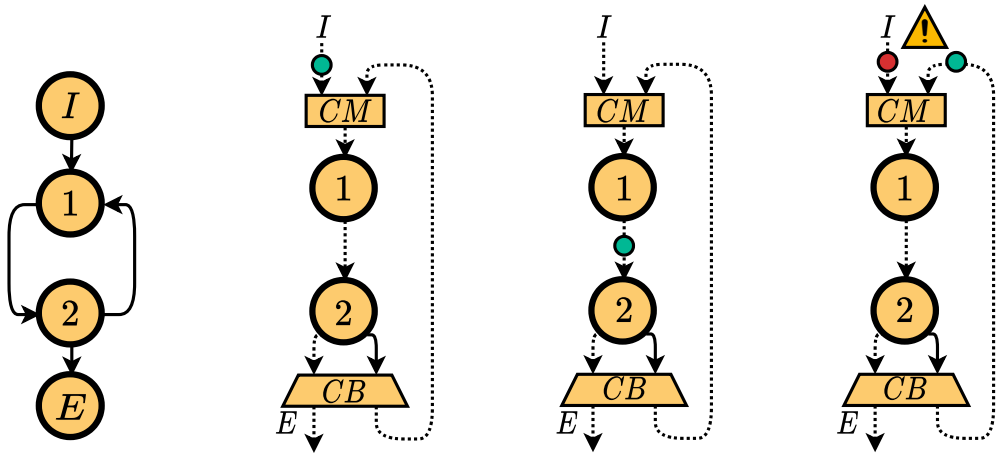
However, the dataflow conversion shown thus far, as well as that of Dynamatic, cannot guarantee correct execution under task pipelining. This is due to control merge operations being non-deterministic with respect to the input that they selected. As such, if multiple kernel invocations are live concurrently—implying that multiple control tokens are flowing in the circuit—control merge operations may select control tokens out-of-order with respect to the order of kernel invocations.

In this chapter, we show an improved handshake architecture that can support task pipelining by replacing control merges with deterministic structures. First, we will present an example wherein our current lowering method fails to ensure correctness under task pipelining. Then, we present a model for making looping (feedback) and branching (feedforward) circuits correct under task pipelining. Finally, we qualitatively evaluate our proposed model by comparing the performance and hardware of our task-pipeline safe model with its unsafe counterpart.

6.1 A Case for Task Pipelining

We now demonstrate a case where the use of control merges renders a circuit unsafe for task pipelining. Consider the CFG in Figure 6.3a. With the conversion method presented thus far, a dataflow control network will be created as that of Figure 6.5a. The loop is initiated by control passing from a block I to a control merge that will accept and forward this control token into the loop header block 1—this is represented by the green control token (Figure 6.5a). The token flows through the loop (Figure 6.3c) and eventually traverses block 2, which will either transfer control back to the loop header (looping case) or block E (exiting case) through the control branch operation.

A case like Figure 6.3d may then occur; here, the red control token represents a second kernel initiation. At this point, due to the non-determinism of control merge operations, *either* of the two control tokens may be executed. Continuing from here, neither deterministic control flow nor the ordering of which the kernel will return results, can be guaranteed.



(a) A CFG with a loop.

(b) Loop is initiated from I.

(c) Control flows through the loop body.

(d) The loop backedge is traversed. At the same time, another loop initiation occurs from I.

Figure 6.3: Example CFG and control token sequence, representing an unsafe execution. Figures *b*-*d* represent the control network of the corresponding dataflow program. In Figure 6.3d, the control merge will non-deterministically select between either of the incoming control tokens, stemming from two separate kernel invocations.

The core of the issue stems from control merge operations and their non-deterministic behavior. To solve this, control flow determinism must be restored. As a solution for the simple case shown in Figure 6.3, consider the dataflow circuit of Figure 6.4.

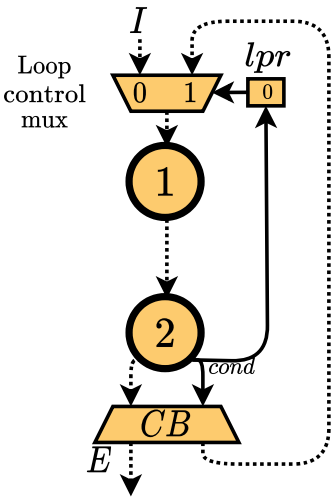
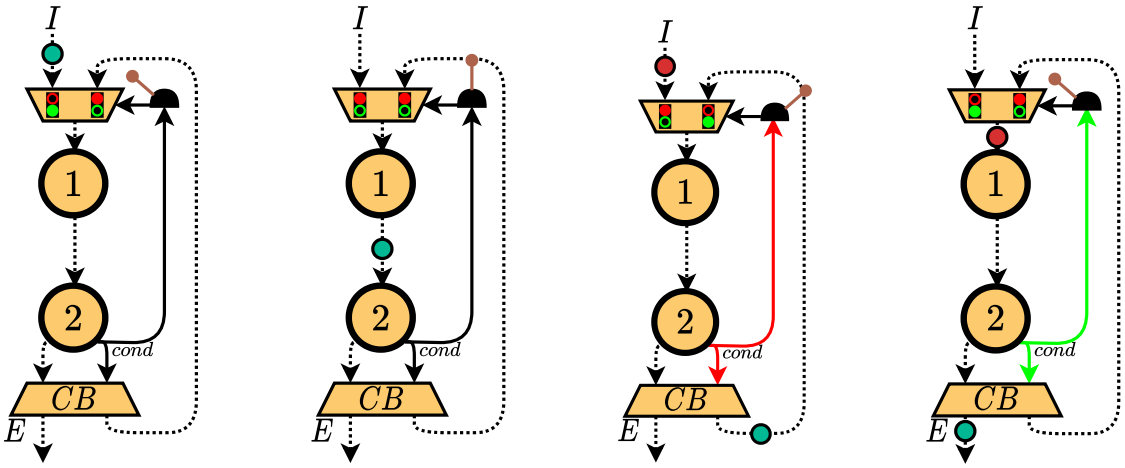


Figure 6.4: Loop control network safe for task-pipelining.

Here, the control merge has been replaced by a *loop control mux* (*lcm*) and a *loop priming register* (*lpr*). The mux has two inputs—one for external initiation (*I*) and one for the loop backedge initiation. The job of the *lpr* is to add determinism to loop initiation, allowing us to control from where loop iterations can be started. The *lpr* is a buffer that is initialized such that the external control input will be accepted upon circuit reset. The loop exit condition of block 2 will drive the next-state value of the *lpr*. Therefore, any time the condition value is transacting, the *lpr* will be re-primed.



(a) lpr is initialized to allow external control.

(b) lpr is empty, no control allowed.

(c) $cond$ branches back to loop header; lpr now only allows internal control. External control cannot transact, and must wait.

(d) Eventually, the current iteration finishes; $cond$ sets lpr to allow external control.

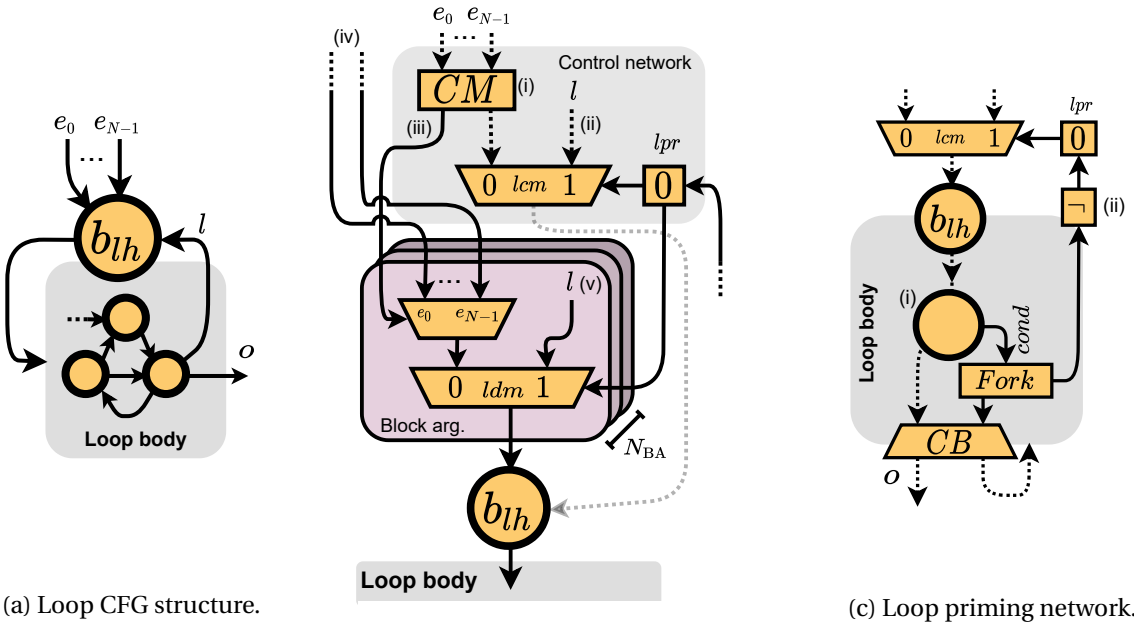
Figure 6.5: Task-pipelining safe execution of the token sequence shown in Figure 6.3.

To demonstrate how this structure enables deterministic loop execution, we consider the same sequence of control tokens as the ones shown in Figure 6.3, wherein we use our loop protection mechanism.

In Figure 6.5a, the *lpr* is initialized to only allow for external input. Once this control input transacts, neither of the mux inputs can accept a new input due to the *lpr* being empty (Figure 6.5b). In Figure 6.5c, the green control token traverses the conditional branch, which reprimes the *lpr*. Now, the next control input of the mux *must* arrive from the loop backedge. Eventually, the first control token will exit the loop, thus repriming the *lpr* to again allow for an external control transaction (Figure 6.5d).

6.2 Making Loops Safe Under Task Pipelining

Figure 6.4 showed the simplest form of a loop, wherein the loop header had only a single backedge and a single external invocation. We now consider the general case and how block arguments (data inputs) to the loop header are handled.



(a) Loop CFG structure.

(b) Loop input network.

(c) Loop priming network.

Figure 6.6: Task pipelining of outer loops. $e_0, \ldots, e_{N-1}$ represent external predecessor blocks, l the loop latch block, and o the exit block..

The general structure that we consider is shown in Figure 6.6a. In identifying loops, we state that an outer loop is any set of *maximal strongly connected components* (SCC) S within a CFG G, that is, a set of blocks $S \subseteq G$ s.t. $\forall q, r \in S \mid \exists \text{path}(q, r) \wedge \text{path}(r, q)$ and $|S| > 1$[1]. S is maximal if no more blocks can be included while remaining an SCC. Furthermore, we only consider *reducible* loops [25], restricting our class of loops to those with only a single loop header block b_{lh}, that is $\forall b \in S \mid b_{lh}$ dom b.

We detect loops as follows: Starting from the entry block, we traverse the CFG in a breadth-first manner, and for each block visited, we check if it is a loop header. A block b is a loop header b_{lh} if $\exists p, p \in \text{predecessors}(b) \mid b \text{ dom } p$ (p is a backedge to b and p can only be reached through b). In general, programming languages with structured control flow (if, for, while, ...) will generate reducible CFGs. Irreducible CFGs can be the result of either `goto` statements (unstructured control flow) or compiler optimizations.
Then, to build S, we run a strongly-connected components analysis starting from the loop header b_{lh}, $b_{lh} \in S$. From this, we can determine a set of blocks $\{E = e_0 \dots e_{N-1}\}, E \cap S = \emptyset$ which are the external predecessor blocks of b_{lh}, and $l, l \in S$ which is the b_{lh} predecessors originating from within the loop (the loop backedge).

Given this description, to perform safe task pipelining, we seek to identify an outermost loop and "protect" this loop from having the control tokens of multiple kernel invocations live concurrently. Nested loops may exist within the outer loop, which still allows for loop-pipelining based on the control token of a single invocation.

We assume that an outer loop has only a single backedge, i.e., only a single loop latch block. This restriction may be met by transforming the CFG to unify loop latches.

Next, we consider the construction of the *loop input network*—responsible for deterministically selecting control and data inputs to a block—and the *loop priming network*—responsible for resetting the loop priming register.

Loop Input Network

Figure 6.6b shows the general dataflow structure of a task-pipelining safe loop header. Two muxes are added, the *lcm* (*loop control mux*) and *ldm* (*loop data mux*), both driven by the *lpr*. The *lcm* is responsible for deterministically selecting control flow tokens, and the *ldm* for selecting a data input originating from the predecessor block where control originated from. A control merge is added with the inputs of E (i). We use a control merge to determine which external predecessor block transferred control to the loop header, thereby providing an index signal (iii) that can drive a mux to select a data input from the corresponding block in E. In a later section, we will see that this control merge will eventually also be replaced.

The control merge (collating external input triggers) and the control output of l (the loop latch block) are connected to the *lcm*.

Next, we consider incoming dataflow to the loop header (Figure 6.6b (block arg.)). The *ldm* will select between a data input coming from either the external predecessor blocks (Figure 6.6b(iv)) or the loop latch block l (Figure 6.6b(v)). Note that this structure is repeated N_{BA} times, where N_{BA} equals the number of block arguments of the loop header.

Loop Priming Network

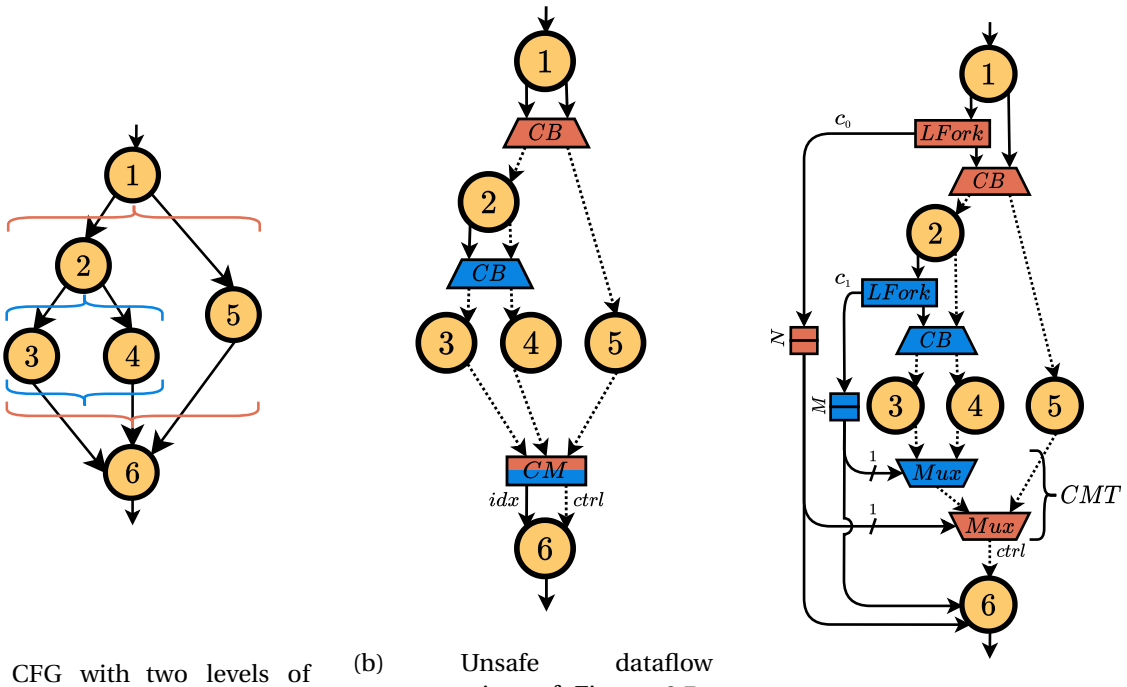
Finally, we again consider the assignment of the loop priming register (Figure 6.6c). We detect the exit block of the loop (Figure 6.6c(ii)), and within this, the conditional branch operation. The argument provided to the conditional branch operation will thus be our $cond$ value to drive the lpr. If there is a mismatch between the taken/not-taken parity of the $cond$ value and the convention used in the loop priming register, an inverter is inserted to correct this (Figure 6.6c(iv)).

Finally, we note that for both the input and loop priming network, in cases where $|E| = 1$ or $|L| = 1$, the generated hardware will be simplified due to the canonical form of the IR. In such cases, both the input control merge and input data muxes will be redundant and removed automatically.

6.3 Feedforward and Feedback Task Pipelining

For the conversion presented in subsection 3.2.2, a control merge will be anywhere where a block has multiple predecessors. The looping case discussed so far can capture situations where control merges are used in the existence of backedges in the CFG—we define the kind of graphs captured by this as *feedback* graphs. What remains to be accounted for is the use of control merge operations in subgraphs of the CFG that are *directed acyclical graphs* (DAGs).

Consider the CFG in Figure 6.7a. Similar to the looping case presented in Figure 6.3, control tokens from different kernel initiations may be live within this circuit at once, e.g. tokens in block 3 and 5. If the circuit of Figure 6.7b is used, the control merge preceding block 6 will non-deterministically select between either of the two tokens, thus making task pipelining unsafe.



(a) CFG with two levels of divergent control flow. Bracket pairs represent control flow divergence and convergence points.

(b) Unsafe dataflow representation of Figure 6.7a. The control merge may non-deterministically mix control tokens from the two branch scopes.

(c) Safe dataflow representation of Figure 6.7a. In-order control flow is now guaranteed for each control scope.

Figure 6.7: A CFG with divergent control flow, and corresponding dataflow representations.

Figure 6.7c shows our mechanism for deterministic execution of *feedforward* graphs. This mechanism revolves around the detection of control flow divergence and convergence points (bracket pairs in Figure 6.7a)—we denote this as a *control scope*. We seek to ensure that the order in which control flow passes through a divergence point will be maintained at the convergence point.

At every convergence point, we insert a mux. This mux will be driven by a buffered value of the *cond* signal of the control branch at the divergence point. By doing so, the convergence mux will deterministically accept control flow in the order of transactions made by the control branch. Due to the feedforward nature of the circuit, this buffer can be implemented as a transparent FIFO. The buffer size N determines the number of tasks that may be live, concurrently, in between the convergence and divergence points. The lazy fork may be swapped for a regular fork, in which case $N+1$ tasks may be live within the kernel at once. The size of this buffer should always be less than or equal to the size of the buffer of an enclosing control scope, i.e., $M \leq N$. Figure 6.7c also shows how the control merge has been broken up into a mux tree, called *CMT* (*control mux tree*). The output of the *CMT* will feed into block 6, wherein execution continues from.

This accounts for the control input to block 6, but we must also consider the *idx* signal provided by the control mux, as seen in Figure 6.7b. This must be resolved to be able to select data inputs to block 6, based on predecessor control flow.

To demonstrate how the index signal is defined, we create a modified version of the loop input network that was shown in Figure 6.6b. In that case, we demonstrated that a loop header may have $e_0, \ldots, e_{N-1}$ predecessors—these predecessors must originate from a situation such as that shown in Figure 6.7.

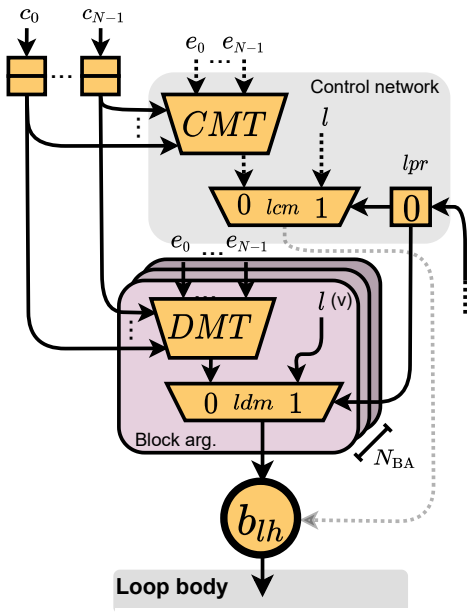


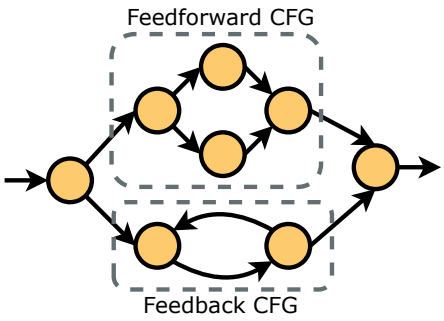
Figure 6.8: Feedback graph entry structure with control merges removed. The *CMT* (*control mux tree*) and *DMT* (*data mux tree*) are driven by the buffered conditionals ($c_0, \ldots, c_{N-1}$) of the preceding feedforward network.

The modified structure is shown in Figure 6.8. Now, the control merge has been replaced by the *CMT* of Figure 6.7c, and the mux for selecting predecessor data inputs have been replaced by a *DMT* (*data mux tree*). Both the *CMT* and *DMT* will be driven by the buffered conditional values of the predecessor control scopes.

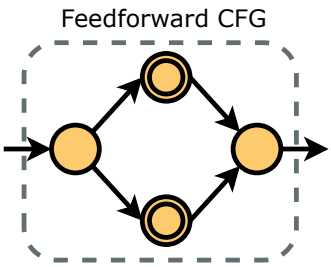
6.3.1 CFG Structure for Task Pipelineable Circuits

We have presented two schemes that allow for deterministic execution under task pipelining of feedforward and feedback subgraphs of a CFG. Therefore, this implies that if we can partition a graph into a set of feedforward and feedback subgraphs, we may convert it to a dataflow circuit safe for task pipelining.

Formally, G must be partitionable into two collections of sets $\mathbf{F}$ and $\mathbf{B}$, with $\mathbf{F}$ containing the feedforward subgraphs of G and $\mathbf{B}$ the feedback regions of G, wherein all sets in $\mathbf{F}, \mathbf{B}$ are disjoint. Any $B \in \mathbf{B}$ will be a maximal SSC and any $F \in \mathbf{F}$ will be *polar*, meaning that there exists exactly one *source* vertex ($\deg^-(source) = 0$) and exactly one *sink* vertex ($\deg^+(sink) = 0$) in the graph.



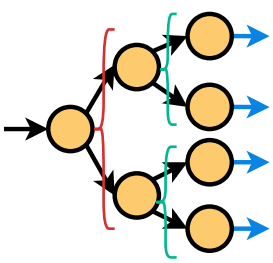
(a) Source CFG with sub-CFGs classified as feedforward and feedback.



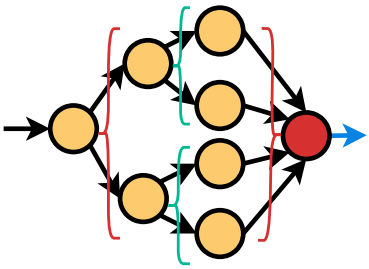
(b) Reduced version of Figure 6.9a, which is now a DAG.

Figure 6.9: Recursive reduction of a CFG based on classifying sub-CFGs as feedforward or feedback (a). Matching sub-CFGs are considered as a single vertex when recursion returns (b).

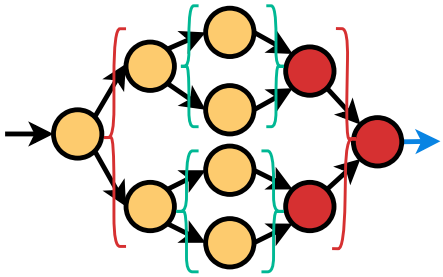
If directly considering G, we do not define feedforward graphs as DAGs. Consider the case of Figure 6.9a. Here, a CFG has two sub-CFGs that adhere to the properties of being a feedforward and a feedback graph. In detecting feedforward regions, we may recursively reduce a CFG based on whether feedforward or feedback subgraphs exist within it. If so, these may be considered a single node, in the parent node, due to the polar nature of the parent graph. After such recursive reduction, we say that a feedforward graph is a DAG.



(a) CFG with unclosed control scopes.



(b) Return unification closes outer control scope.



(c) The remaining control scopes are closed through intermediate blocks.

Figure 6.10: Control scope unification in a tree-like CFG. Blue edges represent function returns, red vertices represent convergence nodes.

Finally, we consider how to maximize task pipelining in feedforward graphs. To apply our lowering scheme, we must detect control flow divergence and convergence points, i.e. the control scopes shown in Figure 6.7a. This can be reduced to a problem of rewriting the CFG itself such that all divergence points (blocks with a conditional branch) eventually have an explicit convergence block, thus forming the polar structure required for a feedforward graph. In general, we can say that in a (reduced, Figure 6.9b) DAG, two control flow paths are created wherever a conditional branch is used. Paths leading from these must eventually either converge or return.

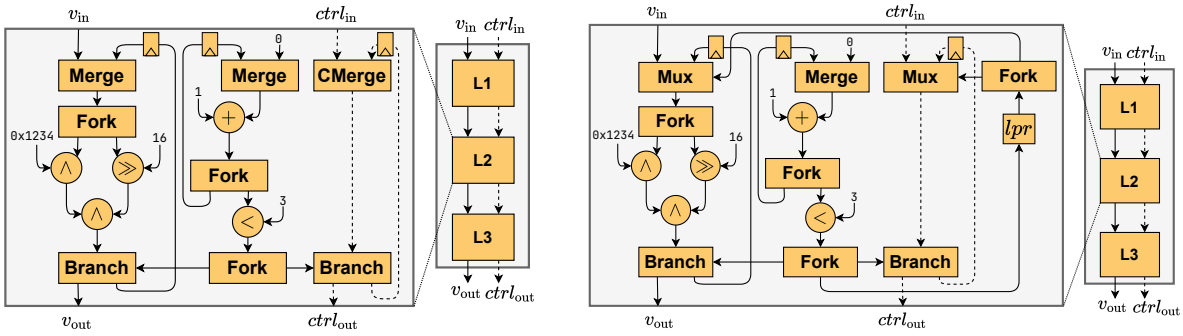
Consider Figure 6.10a. This is a CFG of tree-like structure with fully unclosed control scopes. Initially, we may perform return unification, thus having only a single return block and closing the outer control

scope (Figure 6.10b). Then, convergence points are identified, such as to close the remaining inner control scopes (Figure 6.10c). At this point, all control scopes have been made explicit, facilitating dataflow conversion.

This form of scope closing follows the same motivation as that of the SSA maximization pass presented earlier; we can define properties that must be true for an input CFG to our dataflow lowering pass, which in turn simplifies dataflow lowering. Furthermore, such properties may be easily fulfilled by CFG-level transformations. And by doing so, we avoid having a monolithic dataflow conversion pass.

6.4 Task Pipelining Example

Consider again the `serial_loop` code shown in Figure 6.1. We run Dynamatic and our flow on this kernel. In both, we ensure that buffers are placed optimally, such that loops L1, L2, L3 all have an II of 1. Through simulation, we indeed find that our solution is able to achieve a kernel II=3, whereas Dynamatic shows a kernel II=9.



(a) Unsafe for task pipelining.

(b) Safe for task pipelining.

Figure 6.11: Dataflow representations of `serial_loop`. L1, L2, and L3 are identical.

Next, we consider the hardware implications of our conversion. Figure 6.11a shows the unsafe dataflow representation of `serial_loop`. Through applying the loop transformation, we build the circuit of Figure 6.11b. Since only modifying the control network, no additional buffering is needed.

We have for all programs evaluated in chapter 5 verified correct execution using our task pipelining conversion method. This has been done through our proposed cosimulation infrastructure. Note also, that this makes significant use of the testbench desynchronization methods developed in subsection 4.2.2, wherein the decoupled interface of a task-pipelineable kernel is being exercised through the kernel `call` and `await` functions.

6.4.1 Limitations

Task pipelining is still subject to the constraints of memory data hazards, and would require an LSQ to connect to every memory operation of the circuit. The proposed method for protecting feedback graphs has been tested and verified in CIRCT-HLS. Implementing the feedforward mechanism is considered future work. No program of those evaluated in chapter 5 contains any feedforward graphs; instead they all result in a CFG containing an entry block plus a loop structure.

Chapter 7

Conclusions and Future Work

This work described an end-to-end HLS flow in MLIR and evaluated it against a similar LLVM-based flow. Through this, we have first shown that HLS is possible in MLIR. We find that MLIR, a compiler infrastructure centered around domain-specific IRs, is highly applicable to the creation of HLS compilers. Using MLIR, HLS flows are no longer tied to traditional software compilation infrastructure, whose utility decreases once a program representation diverges from the sequential semantics expected by the infrastructure. Instead, through MLIR, our HLS abstractions are treated as first-class citizens, simplifying aspects such as static analysis, optimization, verification, and transformation. Ultimately, we believe that this will lead to more capable compilers, faster compilation times, and better use of target resources.

This work focused on implementing a dynamically scheduled HLS flow in MLIR. Even with an experimental front-end and a limited set of CDFG optimizations, we can generate hardware with a smaller footprint than a comparable LLVM-based flow. This is primarily due to the ecosystem of IRs that our HLS flow exists within—the RTL dialects of CIRCT. Whereas Dynamatic generates RTL directly from its dataflow IR, in CIRCT, our Handshake IR is but a step in a chain of abstractions. Each with its own set of optimizations and canonicalizations.

Furthermore, we also presented a model for task pipelining of dynamically scheduled circuits, a critical feature for having high-performance kernels in real-world code. Initial experiments show that we, through this model, indeed achieve improved kernel II and better spatial utilization of kernel resources.

While much remains to be done to make MLIR-based HLS competitive against commercial tools, we believe that the path forward is clear and promising, both in terms of research and engineering efforts. We believe that our flow is highly applicable as a basis of future research, in part due to the existence of cosimulation and debugging tools, allowing the researcher to focus on extending the capabilities of the compiler instead of how to drive, test, and debug it, and in part through MLIR itself, allowing for the development of new HLS abstractions and transformation. We also see a clear path forward for improving the generated circuits based on the results observed. Techniques such as advanced buffer placement and bit-width optimization are well explored, and it is now only a matter of engineering effort to unlock their benefits.

7.1 Future Work

The following list of future work is extensive and in no particular order. It serves as a collection of ideas that relate to shortcomings of this work, future research directions, as well as work which we think is relevant in order to create a truly useful, scalable, and open HLS infrastructure.

Handshake IR Canonicalizations

In subsection 3.2.1 we saw the definition of various canonicalization patterns. These are generally simple patterns, and most likely only scratches the surface of what is possible for a canonical form of dataflow IR. Having a well defined canonical form may not only result in better circuits, but can also be a strong argument for why other dataflow projects should be based on Handshake IR.

One example optimization could be the detection of redundant conditional branch-to-mux constructs, such as the one seen in Figure 3.10. Here, a mux operation will always receive the same input, regardless of control flow, and as such it should be valid to remove the mux and `cbranch` operation generating the mux inputs.

Handling Runtime Data Hazards in Handshake IR

While this work lays the foundations for a dynamically scheduled HLS flow, there are still some key issues that must be addressed for the tool to produce performant circuits. One such issue is the detection and handling of run-time memory dependencies. In Dynamatic, this task is delegated to an LSQ. But, LSQ are complex pieces of hardware that may result in excess resource consumption. Possible avenues for handling runtime memory dependencies without relying on an explicit LSQ can be e.g. insertion of alias-checking circuitry that stalls memory operators upon a potential data hazard, or the insertion of shift registers and forwarding paths around a memory to handle the aliasing case.

Structured Handshake IR

Currently, operations in Handshake IR are all placed at the same level of hierarchy. In other words, there is no hierarchy in Handshake IR. However, when we take a CDFG and lower it to Handshake IR, the hierarchy of the source program manifests itself by having clearly denominated input and output control networks based on the source basic blocks. This could be reflected in, e.g. a `handshake.region` operation, wherein all operations originating from a basic block would be nested within it. Such operation could also formalize the use of input and output control signals representing the activation of a set of dataflow operations. At a pragmatic level, this will be beneficial for the programmer that is trying to figure out the semantics of the Handshake program. But we also believe that such a structured representation might open the door for new transformations and program analysis.

Advanced Buffer Placement

As demonstrated in [32], buffer placement and sizing in dynamically scheduled circuits is key to ensure that the generated hardware is performant. Through this work, we implemented a simple buffer placement strategy with the goal of generating correct circuits. This has now been proven to be possible, and we therefore believe that implementing such advanced buffer placement techniques would be a promising avenue for generating more capable hardware.

Coarse-grained Pipelines and Streams

Latency insensitive interfaces allow us to both scale our hardware systems more easily as well as interface with other tools and programming models, one such model being the Handshake kernels generated through this flow. Having a well-defined notion of *streams*, latency insensitive interfaces with buffers, will be crucial in coupling together e.g. statically and dynamically scheduled HLS circuits generated in CIRCT, in a performant way. Currently in CIRCT, the *elastic silicon interconnect* (ESI) [17] dialect provides operations and types for streams. We believe that Handshake could easily fit into the view of the world provided by ESI, and ESI could be a point of convergence for mixing dynamically and statically scheduled HLS in CIRCT.

Better Source Tracing

While MLIR provides location tracking, this is a fairly rudimentary style of tracking wherein each location is tied to an input line/character in a source program. In other words, this location does not track how operations may recursively depend on the location of other operations when being built during lowering. As a result, when an error is thrown deep inside a long compilation pipeline, the error will be associated with a location in the source file, and not a location relating to some intermediate step in the compilation pipeline. We believe that a great contribution to MLIR will be to improve location tracking in such a way that information is added each time a new operation references the location of another. Such tracing would be helpful not only for improving compiler debugging, but could also be the foundations for a Compiler Explorer [24]-like visualization of MLIR pipelines.

Bit Width Minimization for Dataflow Circuits

Currently, `index` types in Handshake IR will be lowered to signals of a fixed width. In section 5.3, we saw how resource utilization of the generated circuits reduced dramatically in response to reducing this fixed width. One solution to this could be to implement a general infrastructure for path-sensitive dataflow analysis [55] in MLIR, on top of which a bit width inference pass could be built upon. This would undoubtedly be a significant contribution that is useful not only to CIRCT, but MLIR in general.

Lowering Through Metaprogramming

In lowering Handshake IR to FIRRTL, we rely on programmatically describing modules that correspond to Handshake IR operations. However, this is fairly cumbersome, since we do not need the full power of MLIR but rather just the ability to specialize a known circuit that implements an operator. In Verilog, we would do this through modules with generic parameters. Having a method of writing template-able MLIR programs and using these in lowering could greatly simplify the specification of a lowering pass.

A Transactional Dialect

In section 4.1, we saw how transactors and interactions between them could be based on a C++-centric flow. For a more scalable approach, a *transactional dialect* could allow for the composition of all kinds of execution styles. This would be a significant step towards cosimulation and heterogeneous execution in MLIR. A transactional dialect could be the glue that binds together the runtime aspects of heterogeneous compute, being software execution, hardware simulators, or interfacing with hardware accelerators.

Call-like Operations in Handshake IR

In our current HLS flow, we lower function calls in a source program to `handshake.instance` functions. It is then expected that the callee is implemented as a Handshake IR function, such that the `handshake.instance` function can instantiate the other function as a hardware module. In practice, an HLS system is heterogeneous, and we can easily imagine designs where a function can be implemented in a number of ways; whether being system peripherals, hard IP, or perhaps a processor. In such cases, we would like to have an operation in Handshake IR which can represent the concept of function calls, allowing for calling hardware that does not necessarily have handshake semantics. We could also imagine that such an operation could lower into those of the transactional dialect mentioned above, which will capture the concept of how to compose heterogeneous execution styles.

Front-end Improvements

Currently, an issue with leveraging a tool such as Polygeist is that its main purpose is not just to be a C/C++ front-end for MLIR, but rather a tool facilitating polyhedral research in MLIR. As seen during evaluation, this frontend still has a lot of room for improvement when compared to Clang. We therefore believe that what is needed is a community-managed C/C++ front-end for MLIR, much like the Clang project. We also find it pertinent to consider how specific dialects can be targeted directly from C/C++ code. An example could be to allow for the mapping of C++ attributes/intrinsics directly to MLIR operations and attributes. With this, users of the front-end can design domain-specific C++ libraries mapping to a domain-specific dialect. An example could be a programming model such as SYCL [49].

Scalable HLS Infrastructure

One of the key properties of a good HLS compiler is the ability to scale to very large and complex input programs, and through this work we have so far only considered relatively small programs. To avoid an explosion in runtime when HLS'ing large programs, the HLS infrastructure itself must therefore also be scalable. An example avenue to improve scalability could be to focus on module reuse across a large HLS project. By doing so, methods must be developed for module identification, partitioning, and how such can be managed through a module library that acts as a cache for the HLS flow.

Concurrent Function and Loop Pipelining

In section 6.2 we saw how a protection mechanism was implemented to ensure the correctness of loop execution in cases of function pipelining. For any loop with an II> 1, a subset of the functional units used in the loop body will inevitably remain idle during execution, making for poor temporal utilization of the generated hardware. Looking to the CPU world, a similar issue is faced in superscalar processors. In that domain, to increase spatial utilization of pipeline stages, *simultaneous multithreading* [26] is used to allow for instructions from different threads to be live concurrently within a pipelined functional unit. In our model, a thread could be similar to a kernel invocation, and an instruction could represent a control transfer token, such as that occurring on a loop backedge. We theorize a link between the amount of buffering available within a loop body and the number of separate invocations that may be live at once in the loop. In the future, we may investigate a modification to our proposed method for safe function pipelining, wherein we allow for out-of-order execution within pipelined loops.

Appendix

A.1 Tooling Overview

The following sections provides a brief overview of the set of tools used throughout this project. Since most of these are actively being developed, APIs are subject to change and the contents of this guide may be outdated. For up-to-date information, please refer to the web page and source code of each tool, to get the most up-to-date information on usage.

The set of tools used in this project consists of `circt-opt`, `hls-opt`, `mlir-opt`, `mlir-clang` and `hlstool`. `hlstool` is described separately in appendix A.2.

mlir-opt

MLIR `opt` tools are *optimization driver* tools. These are used to drive individual passes available within the parent project, and to drive canonicalizations. In CIRCT-HLS, `mlir-opt` is mainly used during testbench lowering to convert high level dialects such as `affine/scf` to `standard` and `LLVM`. The tool is available in the `build/bin` directory of an LLVM build, where the MLIR subproject has been enabled. Example uses:

```
$ mlir-opt --canonicalize        // Runs canonicalizations (peephole opts.)
$ mlir-opt --convert-scf-to-std  // Converts scf dialect to standard dialect
$ mlir-opt --convert-std-to-llvm // Converts standard dialect to LLVM dialect
```

circt-opt

The CIRCT optimization driver. Drives passes and canonicalizations on dialects defined within CIRCT (`handshake,comb,firrtl,...`). The tool is available in the `build/bin` directory of a CIRCT build. Example uses:

```
$ circt-opt --canonicalize           // Runs canonicalizations (peephole opts.)
$ circt-opt --lower-std-to-handshake // Converts std dialect to handshake dialect
```

```
$ circt-opt --handshake-insert-buffers // Inserts buffers in handshake IR
$ circt-opt --handshake-add-ids      // Adds unique IDs to handshake IR operations
$ circt-opt --handshake-print-dot    // Prints a .dot file of the handshake IR
$ circt-opt --flatten-memref-calls   // Flattens multidim. memrefs to unidim. memrefs
```

hls-opt

The CIRCT-HLS optimization driver. This tool is used to apply the testbench conversion passes. The tool is available in the `build/bin` directory of a CIRCT-HLS build.

Example uses:

```
$ hls-opt --cosim-convert-call  // Converts builtin.call ops to cosim.call ops
$ hls-opt --cosim-lower-call    // Lowers cosim.call operations
$ hls-opt --cosim-lower-compare // Lowers cosim.compare operations
$ hls-opt --asyncify-calls      // desynchronizes builtin.call ops
```

mlir-clang

`mlir-clang` is the front end tool exposed by Polygeist used to convert C/C++ code to MLIR. The tool is available in the `build/bin` directory of a Polygeist build. In our usecase, all calls have the format:

```
$ mlir-clang --function=* --memref-fullrank -S ${input file}
```

`-function=*` instructs the tool to convert all functions within the source file. `-memref-fullrank` instructs the tool to emit `memref` operations with static indexes (e.g. `memref<10xi32>`) instead of with dynamic indexes (e.g. `memref<?xi32>`). Statically sized `memref`s are required for the Handshake lowering to work. `-S` instructs the tool to emit assembly code - in this case, assembly code is MLIR code.

Polygeist contains a number of canonicalization passes to both optimize and convert the Polygeist dialect operations. These must be run before passing the MLIR code to CIRCT tools, since any Polygeist dialect operations must be converted fully to a combination of `standard/scf/affine/memref/arith` operations. Polygeist canonicalizations can be driven from the `polygeist-opt` tool:

```
$ polygeist-opt --canonicalize // Runs canonicalizations (peephole opts.)
```

A.2 HLSTool Tutorial

The following document details `hlstool`. `hlstool` is the main driver of CIRCT-HLS, used to compose the various internal and external tools which encompasses the HLS flow.

While the main goal of `hlstool` is to expose commands that allows the synthesis of C programs to HDL in a single, simple call, a secondary goal is to create a tool that helps the development of the toolchain itself.

Some features of the tool are:

- Each command executed is printed verbatum to the executing terminal. Given this, a tool user can inspect the sequence of commands that was executed and easily copy/paste from the terminal, to reproduce any of the commands. All this, without a cryptic shell script in sight!
- Automatically generates `CMakeLists.txt` files for compiling simulator libraries.
- Iteratively `CMake`s a simulator library to achieve the maximum possible Verilator model parallelism.

`hlstool` is also used to drive the CIRCT-HLS regression test suite. As developers, we want to be able to inspect, adjust, and rerun regression tests either when debugging failures or implementing new features. To facilitate this, the tool implements a simple checkpointing mechanism. This tutorial is structured as a sequence of common use-cases of `hlstool`, each building on top of the prior.

In general, tool arguments are provided at two levels; the general level and the mode level:

```
hlstool [general arguments] {mode} [mode arguments]
```

The mode is intended to adjust `hlstool` behaviour to a specific HLS flow. At time of writing, only the `dynamic-polygeist` mode has been implemented. Since the tool is ever evolving, please reference the `hlstool -help` output for a complete and up-to-date description on the full set of capabilities of the tool. The following tutorial assumes that you have the `hlstool` available in your path. The tool will be located in `circt-hls/build/bin`.

Disclaimer: Since CIRCT, and by extension CIRCT-HLS, is an evolving and actively developed project, tool APIs are highly volatile and subject to change. While it is hoped that the following tutorial will be a useful reference for the foreseeable future, the most surefire way of seeing how the `hlstool` is currently being used is to reference the CIRCT-HLS regression tests.

A.2.1 Setup

Clone the CIRCT[1] and CIRCT-HLS[2] repositories, and follow the most up-to-date setup guide provided for each repository. After this, add the binary output directory of CIRCT-HLS to your `$PATH` variable:

[1] <https://github.com/llvm/circt>
[2] <https://github.com/circt-hls/circt-hls>

```
$ export PATH=some/path/to/circt-hls/build/bin:$PATH
```

Optionally to your `.bash_rc/.profile` file to make the change persistent.

A.2.2 Usecase 1: An example kernel

Create a new file `triangle.c` containing the following C program:

```
int triangle(int n) {
  int sum = 0;
  for (int i = 1; i <= n; i++)
    sum += i;
  return sum;
}
```

as well as a directory for `hlstool` to output to:

```
$ mkdir triangle_out
$ cd triangle_out
```

The `hlstool` may also be provided an optional `-outdir` argument to specify the output (working) directory of the tool.

To convert our `triangle` function to a DHLS Verilog program, we will need to specify:

- A path to the C file to convert
- The name of the function to convert within this kernel. The provided function name is interpreted as the top-level function of the kernel
- The HLS mode of the tool

```
$ hlstool --kernel_file ../triangle.c --kernel_name triangle dynamic-polygeist
```

Executing this command, you should now see a `triangle.sv` file containing a SystemVerilog implementation of the kernel. Alongside this, a number of intermediate files is available in the output directory. If you are a developer of `hlstool` or CIRCT-HLS, it is recommended to familiarize yourself with the output of `hlstool` which indicates both the order of as well as the commands that resulted in the generation of each of the intermediate files.

A.2.3 Usecase 2: Testbenches and cosimulation

Create a testbench file `tst_triangle.c` in the same directory where you created the `triangle.c` file:

```
int triangle(int);
int main(void) {
  printf("Triangle(%d) = %d\n", 42, triangle(42));
  return 0;
}
```

In testbench mode, `hlstool` is able to infer the kernel name and kernel file based on the assumption that the testbench file is named `tst_{kernel_name}.c` with the kernel file being `{kernel_name}.c` and the kernel name within the kernel file being `{kernel_name}`.

Next, we will build the testbench and simulator library. We also pass `-rebuild` to ensure that all steps in the HLS flow are repeated. Internally, `hlstool` has most of its commands guarded by a check on the existence of the output file, which it generates. This is used to avoid recompilation in cases where we haven't made any significant changes to the input program, such as when running tests.

```
$ hlstool --rebuild --tb_file ../tst_triangle.c dynamic-polygeist
```

On the testbench side, this will convert the testbench to MLIR using Polygeist (`triangle_affine.mlir`), desynchronize any invocations of the RTL model to `_call`/`_await` functions (`triangle_tb.mlir`) and lower the testbench to MLIR LLVM (`triangle_tb_llvm.mlir`).

On the kernel side, it is at this point where the simulator library is built. A file `triangle.cpp` should now be present, which is the `hlt` wrapper around the verilated model. A `CMakeLists.txt` file is copied into the output directory. This file contains a call to Verilator, which will verilate the model upon executing CMake. Then, the `hlt` wrapper and the verilated model is compiled and linked together to produces `libhlt_triangle.so`—a shared library which implements the `triangle_call`/`triangle_await` functions used by the desynchronized.
Next, we will use `hlstool` to run the simulation:

```
$ hlstool --tb_file ../tst_triangle.c dynamic-polygeist --run_sim
```

Inspecting the output of `hlstool`, we see that `mlir-cpu-runner` is invoked to execute the testbench (paths minimized for brevity):

```
$ mlir-cpu-runner -e main -entry-point-result=i32 -O3                          \
  -shared-libs=libmlir_c_runner_utils.so -shared-libs=libmlir_runner_utils.so \
  -shared-libs=libhlt_triangle.so triangle_tb_llvm.mlir                        \
  > triangle_tb_output.txt
```

Here we see that `triangle_tb_llvm.mlir` is used as the main MLIR file to execute, and `-shared-libs=libmlir_runner_utils.so` ensures that the simulator library functions are linkable.
After simulation finishes, three additional files will be available in the output directory:

- `triangle_tb_output.txt`: `stdout` output generated during execution will be streamed to this file. Within this file, you should be able to see `0`, indicating the return code of the execution, as well as `Triangle(42) = 903`.
- `logs/vlt_dump.vcd`: VCD output of the verilated model. You can inspect this using tools such as `gtkwave`.
- `sim.log`: A log printed by the `hlt` infrastructure. This can be used to debug at which steps inputs were pushed and popped to the `hlt` queues that communicates with the transactor interface of the RTL model.

Note: In case the Handshake model deadlocks during simulation, an assert will be triggered in the `hlt` infrastructure that is triggered after a fixed number of steps has been performed without any noticeable change in simulator state.
Going one step further, we may want to cosimulate the RTL simulation with a software implementation of the kernel. Passing `-cosim` will enable cosimulation transformation of the testbench.

```
$ hlstool --rebuild --cosim --tb_file ../tst_triangle.c dynamic-polygeist
```

Inspecting `triangle_tb.mlir`, we now see calls to the `triangle_call/await` functions and a call to `triangle_ref`. The implementation of `triangle_ref` should have been inlined within the module in `triangle_tb.mlir`.
The testbench can then be executed as before:

```
$ hlstool --cosim --tb_file ../tst_triangle.c dynamic-polygeist --run_sim
```

Currently, cosimulation failure is indicated through `printf` calls. Given this, failing cases can be identified in the testbench output file `triangle_tb_output.txt`.

A.2.4 Usecase 3: HSdbg Visualization and Checkpointing

In cases where VCD inspection make be inconclusive in determining the behaviour of a Handshake circuit, HSdbg can be used to provide a visualization of a testbench run. HSdbg can be run within a directory after a simulation has been executed. By default, `hlstool` will look for a VCD file `logs/vlt_dump.vcd`. To point `hlstool` to a custom VCD file, use the `-vcd` argument. Starting `hsdbg` through `hlstool` is just a small convenience wrapper to ensure that the necessary files are available (Handshake MLIR version of the kernel and a Grapviz `.dot` file of this handshake kernel), and to pass them to `hsdbg`. `hsdbg` is available in `circt-hls/build/bin` and can be invoked separately, if needed.
In the `triangle_out` directory, run:

```
$ hlstool --checkpoint dynamic-polygeist --hsdbg
```

In this commandline, we used the `-checkpoint` option. When the `hlstool` is run, a `.hlstool_checkpoint` file is generated in the working directory of the run. This contains information that can be used to restore the tool arguments at a later point in time. The files loaded from a checkpoint are printed at the start of a run. Executing the above command, `hlstool` will output (path names shortened):

```
INFO:     Using input files from .hlstool_checkpoint
INFO:     Using kernel file: ../triangle.c
INFO:     Using kernel name: triangle
INFO:     Using testbench file: ../tst_triangle.c
INFO:     Stored checkpoint to .hlstool_checkpoint. You can rerun HLSTool in
          this directory and pass the --checkpoint flag instead of providing
          paths and kernel names.
```

On executing the command, a new terminal window will be opened with `hsdbg`, executing and ready to accept commands. `hsdbg` will start a server, defaulting to `localhost:8080` where the visualization will be hosted. Navigate to this page in a browser.
The webpage will show the current step in the simulation. `hsdbg` does not (yet) have a notion of cycles—the simulation grannularity is identical to that of the timesteps used in the VCD file. In the terminal window executing `hsdbg`, simulation time can be increase by pressing right-arrow, and decreased by pressing left-arrow. A specific step in the simulation can be navigated to by pressing g followed by the step number. It may be helpful to use this tool in conjunction with viewing the VCD trace, wherein the VCD trace is used to find situations of interest, and `hsdbg` to get a sense for what is going on at the handshake-level.

A.2.5 Usecase 4: Modifying kernels at the MLIR level

When developing the HLS flow, we oftentimes want to make precise changes to the MLIR representation of a kernel—changes, which may not be directly possible when writing the kernel in C. `hlstool` has support for running only a subset of its lowering based on type of the input kernel. In these case, we can provide the `-mlir_kernel` flag.
For instance, to run the tool starting from the Standard dialect representation of the `triangle` kernel we can run:

```
$ cd triangle_out
$ hlstool --rebuild --mlir_kernel --kernel_name triangle \
  --kernel_file triangle_std.mlir dynamic-polygeist
```

`hlstool` should now inform you that the Polygeist step was skipped:

```
INFO:     Skipping Polygeist lowering due to using an MLIR kernel
```

Or with a Handshake IR kernel; in this case we pass the mode argument `-hs_kernel` to direct the Handshake-specific part of `hlstool` to skip lowering:

```
$ hlstool --rebuild --mlir_kernel --kernel_name triangle \
  --kernel_file triangle_handshake.mlir dynamic-polygeist --hs_kernel
```

`hlstool` should now inform you that both the Polygeist and handshake steps were skipped:

```
INFO:     Skipping Polygeist lowering due to using an MLIR kernel
INFO:     Skipping handshake lowering due to using a handshake kernel
```

Tip: After building CIRCT-HLS and running the cosim test suite:

```
$ cd build
$ ninja check-circt-hls-cosim
```

You can easily rerun the `hlstool` invocation by navigating to the output directory of the tests (e.g. `circt-hls/build/cosim_test/suites/Dynamatic/simple_example_1`). This is useful when you want to make slight modifications to the test runs or to reproduce a failing test.

A.2.6 Usecase 5: Creating a Binary Executable Testbench

It may happen that we want to write a C/C++ program to directly drive the `call/await` functions of the `hlt` wrapper - this is often the case when developing and debugging the simulation infrastructure itself. When compiled into a regular executable (as opposed to executing testbenches through `mlir-cpu-runner`) This also gives us the ability to easily step through the code while debugging the executable in an IDE.

To illustrate this, we write a testbench `tst_triangle_manual.c` similar to the one shown previously. However, we explicitly reference the call and await functions exposed by the `hlt` transactor interface. Note that we must have predeclarations available for the `call/await` functions. These are available in the `triangle.h` file in the output directory, and generated alongside the `triangle.cpp` file.

```
#include "triangle.h"
int main() {
  triangle_call(42);
  int res = triangle_await();
```

```
  printf("Triangle(%d) = %d\n", 42, res);
  return 0;
}
```

The `CMakeLists.txt` file in the output directory contains variables that can be used to create a standalone executable instead of a shared library. To build the above testbench alongside our simulator, we run `cmake` with:

```
$ cmake -DHLT_EXEC=1 -DHLT_TESTNAME=triangle -DHLT_EXEC_TB=tst_triangle_manual.c
```

`HLT_EXEC` will trigger an executable build instead of a shared library. By default, a file `main.cpp` is expected to be present in the directory. If this is not the case, you can provide a path to a testbench file with the `-DHLT_EXEC_TB=${path}` variable. The testbench file must contain a `main` function - if not, compilation will fail.
Then, run `ninja`, and you should have an exectuable `hlt_triangle` in the output directory which you can execute/debug like any other CMake project:

```
$ ninja
$ ./hlt_triangle
Triangle(42) = 903
```

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. “Compilers, principles, techniques”. In: *Addison wesley* 7.8 (1986), p. 9.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. “Chisel: constructing hardware in a scala embedded language”. In: *DAC Design Automation Conference 2012*. IEEE. 2012, pp. 1212–1221.

[3] B Bailey, F Balarin, M McNamara, G Mosenson, M Stellfox, and Y Watanabe. *TLM-Driven Design and Verification Methodology*. Cadence Design Systems, June 2010.

[4] Albert Benveniste, Benoit Caillaud, and Paul Le Guernic. “From synchrony to asynchrony”. In: *International Conference on Concurrency Theory*. Springer. 1999, pp. 162–177.

[5] Endri Bezati, Mahyar Emami, Jörn Janneck, and James Larus. “StreamBlocks: A compiler for heterogeneous dataflow computing (technical report)”. In: *arXiv preprint arXiv:2107.09333* (2021).

[6] Stephen D Brown. *Fundamentals of digital logic with Verilog design*. Tata McGraw-Hill Education, 2007.

[7] Dmitry Bufistov, Jordi Cortadella, Mike Kishinevsky, and Sachin Sapatnekar. “A general model for performance optimization of sequential systems”. In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE. 2007, pp. 362–369.

[8] Lukai Cai and Daniel Gajski. “Transaction level modeling: an overview”. In: *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (IEEE Cat. No. 03TH8721)*. IEEE. 2003, pp. 19–24.

[9] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. “LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.2 (2013), pp. 1–27.

[10] Bingyi Cao, Kenneth A Ross, Martha A Kim, and Stephen A Edwards. “Implementing latency-insensitive dataflow blocks”. In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE. 2015, pp. 179–187.

[11] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. “Elastic circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.10 (2009), pp. 1437–1455.

[12] Jianyi Cheng, Lana Josipović, George A Constantinides, Paolo Ienne, and John Wickerson. “DASS: Combining Dynamic and Static Scheduling in High-level Synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[13] Alessandro Cilardo and Luca Gallo. “Interplay of loop unrolling and multidimensional memory partitioning in HLS”. In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 163–168.

[14] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. “Source-to-source optimization for HLS”. In: *FPGAs for Software Programmers*. Springer, 2016, pp. 137–163.

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. “High-level synthesis for FPGAs: From prototyping to deployment”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), pp. 473–491.

[16] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[17] John Demme. “Elastic Silicon Interconnects”. In: *Latte’21* (2021).

[18] Benoit Dupont de Dinechin. “Simplex scheduling: More than lifetime-sensitive instruction scheduling”. In: *Proceedings of the International Conference on Parallel Architecture and Compiler Techniques*. Citeseer. 1994.

[19] Stephen Director, A Parker, D Siewiorek, and D Thomas. “A design methodology and computer aids for digital VLSI systems”. In: *IEEE Transactions on Circuits and Systems* 28.7 (1981), pp. 634–645.

[20] Stephen A Edwards, Richard Townsend, Martha Barker, and Martha A Kim. “Compositional dataflow circuits”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.1 (2019), pp. 1–27.

[21] Paul Feautrier and Christian Lengauer. “Polyhedron Model.” In: *Encyclopedia of parallel computing* 1 (2011), pp. 1581–1592.

[22] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. “Transformations of high-level synthesis codes for high-performance computing”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.5 (2020), pp. 1014–1029.

[23] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 1992.

[24] Godbolt, M. *Compiler Explorer [Online]*. Available: `https://godbolt.org/`. 2012.

[25] Paul Havlak. “Nesting of reducible and irreducible loops”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.4 (1997), pp. 557–567.

[26] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[27] MA Ilgamov. “Static problems of hydroelasticity”. In: (1998).

[28] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis. UG902*. Version v2018.3. Dec. 2018. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf`.

[29] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. “Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 209–216.

[30] Lana Josipovic, Philip Brisk, and Paolo Ienne. “From C to elastic circuits”. In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. IEEE. 2017, pp. 121–125.

[31] Lana Josipović, Radhika Ghosal, and Paolo Ienne. “Dynamically scheduled high-level synthesis”. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2018, pp. 127–136.

[32] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. “Buffer placement and sizing for high-performance dataflow circuits”. In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 186–196.

[33] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. “In-datacenter performance analysis of a tensor processing unit”. In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[34] R. Kastner, J. Matai, and S. Neuendorffer. “Parallel Programming for FPGAs”. In: *ArXiv e-prints* (May 2018). arXiv: 1805.03648.

[35] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[36] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. “MLIR: A compiler infrastructure for the end of Moore’s law”. In: *arXiv preprint arXiv:2002.11054* (2020).

[37] Maysam Lavasani. “Generating irregular data-stream accelerators: methodology and applications”. PhD thesis. 2015.

[38] Junyi Liu, John Wickerson, and George A Constantinides. “Loop splitting for efficient pipelining in high-level synthesis”. In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2016, pp. 72–79.

[39] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. “Polygeist: Raising C to Polyhedral MLIR”. In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2021, pp. 45–59.

[40] Kevin E Murray, Mohamed A Elgammal, Vaughn Betz, Tim Ansell, Keith Rothman, and Alessandro Comodi. “Symbiflow and vpr: An open-source design flow for commercial and novel fpgas”. In: *IEEE Micro* 40.4 (2020), pp. 49–57.

[41] Chris Nicol. “A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing”. In: *Wave Computing White Paper* (2017).

[42] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. "Predictable accelerator design with time-sensitive affine types". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 393–407.

[43] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. "A compiler infrastructure for accelerator generators". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 804–817.

[44] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. "Accelerating deep convolutional neural networks using specialized hardware". In: *Microsoft Research Whitepaper* 2.11 (2015), pp. 1–4.

[45] Morten Borup Petersen. *Ripes*. `https://github.com/mortbopet/Ripes`.

[46] *Project X-ray [Online]*. Available: `https://github.com/SymbiFlow/prjxray/`. 2020.

[47] B Ramakrishna Rau. "Iterative modulo scheduling: An algorithm for software pipelining loops". In: *Proceedings of the 27th annual international symposium on Microarchitecture*. 1994, pp. 63–74.

[48] S Ravi and M Joseph. "Open source HLS tools: A stepping stone for modern electronic CAD". In: *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*. IEEE. 2016, pp. 1–8.

[49] Ruyman Reyes and Victor Lomüller. "SYCL: Single-source C++ accelerator programming". In: *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.

[50] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca 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*. 2020, pp. 258–271.

[51] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. "Yosys+ nextpnr: an open source framework from verilog to bitstream for commercial fpgas". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 1–4.

[52] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. "μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 940–953.

[53] Wilson Snyder. "Verilator and systemperl". In: *North American SystemC Users' Group, Design Automation Conference*. 2004.

[54] M Sussmann and T Hill. "Intel HLS Compiler: Fast Design, Coding, and Hardware". In: *White paper*. 2017.

[55] Aditya Thakur and R Govindarajan. "Comprehensive path-sensitive data-flow analysis". In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 2008, pp. 55–63.

[56] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. “Finn: A framework for fast, scalable binarized neural network inference”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 65–74.

[57] Mike Urbach and Morten Borup Petersen. “HLS from PyTorch to System Verilog with MLIR and CIRCT”. In: *Latte'22* (2022).

[58] Devadas Varma, Duncan Mackay, and Pradeep Thiruchelvam. “Easing the verification bottleneck using high level synthesis”. In: *2010 28th VLSI Test Symposium (VTS)*. IEEE. 2010, pp. 253–254.

[59] Maria Vieira, Michael Canesche, Lucas Bragança, Josué Campos, Mateus Silva, Ricardo Ferreira, and Jose A Nacif. “RESHAPE: A Run-time Dataflow Hardware-based Mapping for CGRA Overlays”. In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2021, pp. 1–5.

[60] Clifford Wolf. *Yosys open synthesis suite*. 2016.

[61] Xilinx. *Vitis High-Level Synthesis User Guide UG1399*. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf`.

[62] Hanchen Ye, Cong Hao, Hyunmin Jeong, Jack Huang, and Deming Chen. “ScaleHLS: Achieving Scalable High-Level Synthesis through MLIR”. In: (2021).

[63] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. “AutoPilot: A platform-based ESL synthesis system”. In: *High-Level Synthesis*. Springer, 2008, pp. 99–112.

[64] Ruizhe Zhao and Jianyi Cheng. “Phism: Polyhedral High-Level Synthesis in MLIR”. In: *arXiv preprint arXiv:2103.15103* (2021).