Reticle: A Virtual Machine for Programming Modern FPGAs

Luis Vega
University of Washington
USA
vegaluis@cs.washington.edu

Joseph McMahan
University of Washington
USA
jmcmahan@cs.washington.edu

Adrian Sampson
Cornell University
USA
asampson@cs.cornell.edu

Dan Grossman
University of Washington
USA
djg@cs.washington.edu

Luis Ceze
University of Washington
USA
luisceze@cs.washington.edu

Abstract

Modern field-programmable gate arrays (FPGAs) have recently powered high-profile efficiency gains in systems from datacenters to embedded devices by offering ensembles of heterogeneous, reconfigurable hardware units. Programming stacks for FPGAs, however, are stuck in the past—they are based on traditional hardware languages, which were appropriate when FPGAs were simple, homogeneous fabrics of basic programmable primitives. We describe Reticle, a new low-level abstraction for FPGA programming that, unlike existing languages, explicitly represents the special-purpose units available on a particular FPGA device. Reticle has two levels: a portable *intermediate language* and a target-specific *assembly language*. We show how to use a standard *instruction selection* approach to lower intermediate programs to assembly programs, which can be both faster and more effective than the complex metaheuristics that existing FPGA toolchains use. We use Reticle to implement linear algebra operators and coroutines and find that Reticle compilation runs up to 100 times faster than current approaches while producing comparable or better run-time and utilization.

CCS Concepts: • **Software and its engineering** → *Software notations and tools*; Compilers;

Keywords: compilers, FPGAs

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454075

ACM Reference Format:
Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. 2021. Reticle: A Virtual Machine for Programming Modern FPGAs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454075

1 Introduction

Field-programmable gate arrays (FPGAs) have emerged as a relief to stagnating performance on CPUs and GPUs [16, 17, 38]. Their key advantage is their ASIC-like ability to customize data paths, control logic, and memory hierarchies for specific applications. Unlike an ASIC, however, deploying an FPGA-based accelerator merely requires buying off-the-shelf parts—and not the astronomical investment that manufacturing custom silicon entails.

Early FPGAs were simple, homogeneous fabrics mostly based on *lookup tables* (LUTs), and their toolchains could treat them as fluidly reconfigurable circuits. Modern FPGAs, however, no longer resemble those simple, homogeneous architectures. Because real, specialized hardware remains far more efficient than reconfigurable emulated circuits, modern FPGAs incorporate an array of heterogeneous, special-purpose "hardened" units that implement commonplace functionality: memories, arithmetic units, and complex interconnects [21, 48]. To make these modern FPGAs perform well, it is critical to exploit this fixed-function logic as much as possible—programs that underutilize it can consume significantly more area and power [40].

The mainstream approach to program FPGA today is by using behavioral hardware description languages (HDLs), either written by hand or emitted by higher-level languages [4, 12, 22, 34, 35, 49] as shown in Figure 1. These languages, however, rely on behavioral HDLs as an *ad hoc* IRs, because programs can be ported to multiple targets without the burden of directly programming low-level and target-specific primitives. Instead, the complex task of compiling traditional hardware languages to these primitives is normally performed by proprietary vendor toolchains.

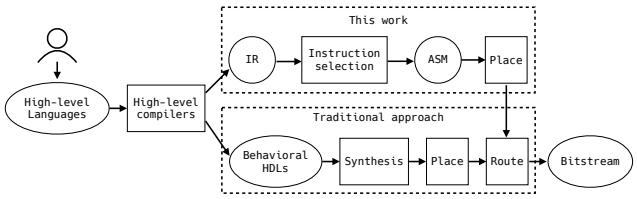


Figure 1. Overview of the traditional compilation pipeline compared to Reticle's. Independent of the source language, FPGA programs today are funneled down to the same abstraction (hardware description languages). This abstraction is not expressive enough to capture high-performance operations available in DSPs, such as SIMD.

Moreover, behavioral HDLs like Verilog and VHDL do not have a way to represent modern FPGA's primitives. Alternatively, vendor toolchains use heuristics that attempt to guess when a program's logic can efficiently map to a device's available hardened units. For example, a Verilog expression `a + b` would need to compile to an adder circuit when generating a custom ASIC; in an FPGA toolchain, it might instead map onto a specific configuration of an FPGA's *digital signal processing slice* (DSP) that includes a range of built-in integer arithmetic units.

Relying on heuristics for performing this mapping results in unpredictable and poor performance. As authors from Xilinx, a major FPGA vendor, observed [31]:

> The necessity of breadth coverage by commercial tools often leads to implementations that do not take full advantage of the underlying hardware. For example, UltraScale+ devices employ DSP blocks that are rated at 891MHz for the fastest speed grade. Nonetheless, large designs implemented on FPGAs typically achieve system frequencies lower than 400MHz.

We corroborate this finding in Section 7, which demonstrates that HDL-based FPGA programming leads to extremely slow compilation and unpredictable, suboptimal performance results. In practice, programmers must resort to using vendor-specific Verilog annotations to extract the best performance from modern FPGAs, or even building *ad hoc* Verilog post-processing scripts to insert directives for efficiency [3]. Although it is fragile and not portable, this approach has no viable alternative: vendors keep lower-level representations secret and the accompanying toolchains are closed source.

This paper's thesis is that we need a new low-level programming abstraction for modern FPGAs. Innovation in high-level FPGA programming models is accelerating [15, 18, 28, 34], and these new compilers need a better target than Verilog. Where traditional HDLs hide the complexity of modern FPGA resources, a better abstraction would explicitly represent the model-specific capabilities that make modern FPGAs fast. The representation should offer both a high-level abstraction for portability and a low-level abstraction to directly address model-specific hardware resources—and come with an infrastructure that provides analyses and optimizations to target specific devices.

We describe Reticle, a low-level language for efficiently programming modern FPGAs. The goal is to provide an intermediate language and a compilation target that can replace the use of traditional behavioral HDLs for targeting FPGAs, as shown in Figure 1. Reticle is a compilation target for higher-level languages that are substantially different from traditional HDLs.

Reticle has two layers: a portable *intermediate language* that abstracts over specific hardware units while providing high-level control over resource binding and placement, and a parameterized *assembly language* that explicitly addresses a device's hardware resources.

The contributions in this paper are:

- We identify and measure the challenges of using behavioral HDLs for programming modern FPGAs in terms of result quality and compiler speed.
- We design Reticle, an intermediate language that can describe the binding of high-performance hardware primitives in modern FPGAs (i.e., DSPs) as well as classic reconfigurable logic (i.e., LUTs).
- We implement a compiler for Reticle that optimizes and lowers hardware programs and then emits structural, device-specific Verilog to bypass the majority of traditional vendor FPGA toolchains.
- We demonstrate that the Reticle compiler runs up to 100× faster than an existing vendor FPGA toolchain while producing comparable or better run-time and utilization results on linear algebra benchmarks.

This paper describes a language design aiming to efficiently program modern FPGAs, but it leaves important avenues for future work to build on. Currently, the *intermediate language* focuses on programming DSP and LUT slices; it does not support memory primitives, such as BRAMs. Additionally, the paper focuses on designing and implementing an *assembly language* capable of capturing layout semantics that enable layout optimizations such as instruction cascading. The compiler, however, uses only a simple solver-based approach for placement, limiting the search of optimal layouts for any given program. There is plenty of exploration needed in the layout space i.e., incorporating timing information that is beyond the scope of this work. Lastly, the Reticle compiler, as of today, relies on routing and bitstream generation from traditional toolchains as shown in Figure 1.

2 Background

The first major program transformation that FPGA compilers perform today is *hardware synthesis*. This transformation rewrites a hardware program described *behaviorally* into an

```
module bit_and(input a, input b, output y);
  assign y = a & b;
endmodule
```

(a) Behavioral Verilog.

```
module bit_and(input a, input b, output y);
  LUT2 # (.INIT(4'h8))
    i0 (.I0(a), .I1(b), .O(y));
endmodule
```

(b) Structural Verilog.

```
module bit_and(input a, input b, output y);
  (* LOC = "SLICE_X0Y0", BEL = "A6LUT" *)
  LUT2 # (.INIT(4'h8))
    i0 (.I0(a), .I1(b), .O(y));
endmodule
```

(c) Structural Verilog with layout annotations.

Figure 2. Three Verilog representations of *and* program.

equivalent *structural* representation. Hardware languages use behavioral expressions to define *what* operations compute, whereas structural expressions defines concretely *how* they are implemented from primitive components.

For example, consider the behavioral Verilog program in Figure 2a. This program uses a binary expression that performs the *and* operation. One valid transformation of this program is shown in Figure 2b, where the *and* operation is lowered to a LUT, the traditional programmable logic unit of FPGAs. The behavioral program is the standard, portable way to program FPGAs today; the structural implementation addresses the specific LUT resources on a specific family of FPGA devices. Additionally, structural implementations can capture layout semantics via Verilog attributes as shown in Figure 2c, including the location `LOC` of a slice and the basic element of logic `BEL` for a primitive. In this case, the `LOC` value represents a specific slice located at the Cartesian coordinate $(0, 0)$ and the `BEL` value denotes an unique LUT (*A6LUT*) within this slice. A LUT slice can host multiple LUTs i.e., UltraScale+ devices support eight LUTs per slice.

Although LUTs are the building block that classically dominated FPGAs, they are not the only programmable resource available on modern FPGAs capable of computing *and* operations. Over the years, FPGAs have added other kinds of primitives: most prominently, digital signal processing slices (DSPs) that can execute this and other complex operations. Although LUTs can implement arbitrary Boolean logic formulas, DSPs can implement specific operations faster and far more efficiently [31]. For example, an 8-bit *and* operation can typically be implemented using a single DSP or 8 LUTs. In modern FPGAs, DSPs are a source of heterogeneity because they support a wide variety of complex operations, such as

```
(* use_dsp = "yes" *)
module dsp_add(...);
  genvar i;
  for (i=0; i<N; i++) begin
    assign y[i] = a[i] + b[i];
  end
endmodule
```

Figure 3. Behavioral Verilog program for adding two arrays of N elements in parallel using DSP annotations. Figure 4 shows how, even with the compiler hint, a hand-optimized structural program is better able to use FPGA resources.

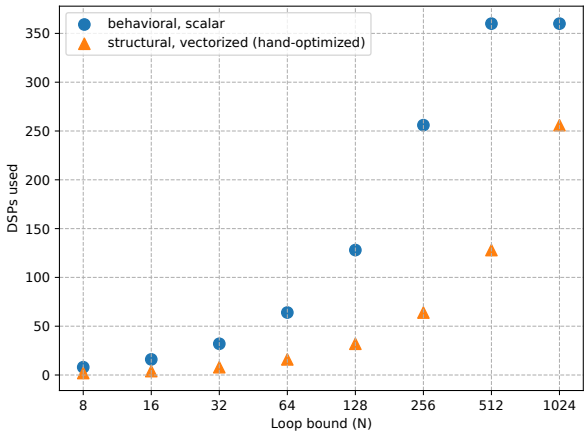
scalar, vector, and fused integer operations and, in recent products, even floating-point arithmetic [11].

Modern FPGA hardware synthesizers heuristically map behavioral HDLs onto LUTs and DSPs based on a cost model and resource availability. The cost model is normally based on the type of the operation and integer type of the operands. For example, a synthesizer might prefer to map integer multiplications to DSPs because of the poor size and speed tradeoff of a LUT-based multiplier, but a small-integer additions might map to LUTs because the speed difference is small and FPGAs typically have more LUTs than DSPs. In addition to cost models, synthesizers also support *hint* annotations in HDLs to *suggest* the use of DSPs over LUTs.

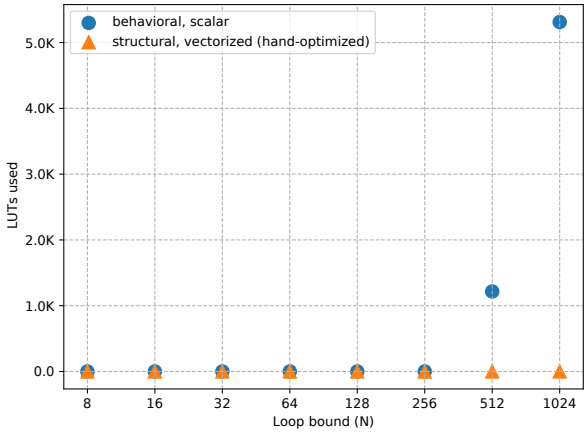
Even with cost models and hints, however, behavioral HDLs are insufficient to fully exploit resources like DSPs available in modern FPGAs. Consider the program in Figure 3, which consists of a loop that performs the summation of two arrays of N elements in parallel. We performed hardware synthesis on this program for different values of N targeting an FPGA that contains 360 DSPs. Figure 4 shows the number of resources consumed by this program when using the *behavioral* HDL description versus a hand-optimized *structural* implementation. The syntax of the optimized program is not shown due to space constraints: it requires configuring up to 96 different DSP parameters for each operation.

This experiment demonstrates three challenges of current hardware synthesizers and languages when targeting FPGAs. First, using the behavioral representation together with compiler hints to force the use of DSPs over LUTs only covers one of the many configurations available in the DSP, resulting in the underutilization of resources. For example, Figure 4a shows that the total number of DSPs in this particular device is already reached for size $N = 512$, although the maximum number of parallel addition allowed in this device is 1440 because of DSP vectorization (360 DSPs each performing 4 parallel additions). Nevertheless, the synthesizer starts rewriting *add* expression to LUTs for $N \geq 512$ as shown in Figure 4b.

A second challenge is that, in HDLs, hints are merely suggestions—not constraints. Behavioral-to-structural synthesis heuristics make it difficult to deterministically respect



(a) DSP utilization.



(b) LUT utilization.

Figure 4. Resource utilization for multiple loop bounds (N) of the *behavioral* program described in Figure 3 versus a hand-optimized and *structural* version of the same program. Even though a compiler hint in the behavioral program requests the use of DSPs, a more optimal DSP configuration exists, leading to under-utilization of the resource potential (the behavioral program runs out of DSP resources and must resort to LUTs).

constraints, so toolchains silently ignore hints that they are unable to fulfill. The consequence is that programming with hints is unpredictable in both area and performance.

The third challenge is that directly programming at the structural level, while necessary for peak efficiency, is impractical. It requires understanding the complex, device-specific semantics of DSP configuration parameters. Structurally programming the DSP in Xilinx UltraScale+ FPGAs, for example, can entail setting up to 96 parameters. This representation is verbose, brittle, and vendor-specific—no structural representation is portable across FPGA families.

3 Overview

Reticle is an intermediate representation (IR) and compiler for FPGAs that addresses these challenges. Reticle aims to directly represent and optimize for the heterogeneous programmable resources available in modern FPGAs: specifically, LUTs and DSPs. Its goal is to target the efficiency of structural FPGA implementations while adding abstraction and portability. Reticle is an instruction-based IR that decouples the low-level details of the underlying hardware from a higher-level instruction set that can generate code for different hardware targets. More importantly, Reticle presents an alternative approach for programming FPGAs and is not a drop-in replacement for any stage in the traditional compilation flow (i.e., the goal is not to support traditional HDLs like Verilog). Alternatively, higher-level languages can use Reticle as a compiler target.

Reticle addresses the challenges from Section 2 by using a more expressive type system that supports vector types, which enable programs to promote particular hardware resources over others when they are available. Additionally, the intermediate language makes primitive constraints part of the language semantics, so the Reticle compiler can reject programs with unsatisfiable constraints instead of silently ignoring them as in HDL hints. Therefore, programs are more predictable in terms of resource usage and performance. We show how to use *instruction selection* to map a portable representation onto a device-specific representation, while achieving the same optimization results as target-specific structural implementations.

The rest of the paper describes the Reticle language design and compiler implementation. Section 4 describes the two forms of Reticle: a portable, high-level intermediate language and a low-level, device-specific assembly language that can be parameterized for a specific FPGA device. Section 5 describes how the Reticle compiler lowers from the intermediate language to an assembly language. We show how to use standard *instruction selection* to efficiently and deterministically lower intermediate programs to assembly programs—a sharp departure from traditional FPGA toolchains, which must resort to expensive, often randomized metaheuristics to perform similar lowering [33]. Sections 6 and 7 show how our compiler implementation emits structural hardware descriptions for a specific FPGA target and results comparable to or better than a traditional HDL toolchain while running many times faster. Finally, Section 8 discusses the responsibilities and optimization opportunities for front-end languages when compiling to Reticle.

4 The Language

This section describes the Reticle language. Reticle has two variants: the high-level *intermediate language*, where operations are abstract and portable across FPGA devices, and a

$$
\begin{aligned}
fun \in Function &::= n(v:\tau)^* \rightarrow (v:\tau)^+\{ins^+\} \\
ins \in Instruction &::= wire \mid comp \\
wire \in Wire &::= v:\tau = \otimes[i^*](v^*) \\
comp \in Compute &::= v:\tau = \boxplus[i^*](v^+)\ @\ res
\end{aligned}
$$

(a) The Intermediate Language

$$
\begin{aligned}
fun \in Function &::= n(v:\tau)^* \rightarrow (v:\tau)^+\{ins^+\} \\
ins \in Instruction &::= wire \mid asm \\
wire \in Wire &::= v:\tau = \otimes[i^*](v^*) \\
asm \in Assembly &::= v:\tau = \boxtimes[i^*](v^+)\ @\ loc
\end{aligned}
$$

(b) The Assembly Language

$$
\begin{aligned}
res \in Resource &::= ?? \mid \rho \\
loc \in Location &::= \rho(\theta,\theta) \\
\theta \in Coordinate &::= ?? \mid e \\
\rho \in Primitive &::= \text{lut} \mid \text{dsp} \\
e \in CoordExpr &::= i \mid v \mid e + e
\end{aligned}
$$

$$
\otimes \in WireOp \quad \boxplus \in CompOp \quad \boxtimes \in AsmOp
$$

$$
?? \in Wildcard \quad n \in Name \quad v \in Variable
$$

$$
\tau \in bool, int, \overrightarrow{int} \quad i \in \mathbb{Z}
$$

Figure 5. The Intermediate and Assembly Languages.

low-level *assembly language*, where operations correspond to physical primitives available on a specific device.

Figure 5 lists the syntax for the two languages, which share a common structure and differ in the kinds of operations that are available. We first describe the intermediate language and then show how the assembly language differs.

4.1 The Intermediate Language

Figure 5a lists the Reticle intermediate language. A program is a function with a name n, a number of inputs and outputs ($v:\tau$), and a sequence of instructions *ins*. Function bodies are in A-normal form (ANF) [41]: they consist of a flat list of instructions whose arguments are always variables v.

Wire & compute instructions. There are two types of instructions in the language: *wire* and *compute* instructions. While both share a common format, compute instructions are the ones that consume device resources and therefore consume area; wire instructions are area-free and only involve wiring. Both kinds of instructions support static integer attributes i, argument variables v, and always produce a single output value ($v:\tau$).

Wire instructions consist of operations $\otimes$, while compute instructions are based on an operation $\boxplus$ that are performed

```
t0: i8 = const[5];
t1: i8 = sll[1](t0);
t2: i8 = add(t0, t1) @??;
```

Figure 6. Reticle instructions to compute the expression $5 \times 2 + 5$. The constant 5 and shift-left-logical operation consume no compute resources (wire operations), while the `add` instruction does (compute operation).

Table 1. The intermediate instruction set.

Instruction	Type	Operation
Compute	Arithmetic	add, sub, mul
	Bitwise	not, and, or, xor,
	Comparison	eq, neq, lt, gt, le, ge
	Control	mux
	Memory	reg
Wire	Shift	sll, srl, sra
	Misc	slice, cat, id, const

by a primitive ρ. Therefore, compute instructions are candidate for optimizations. Figure 6 shows an example of wire and compute instructions.

Compute instructions also have an annotation @ *res* that can optionally control which kind of resource to use on the target device: either LUTs or DSPs. The *res* annotation may be the wildcard ??, in which case the compiler has the freedom to choose which resource to use for the instruction.

Interestingly, other operations besides simple bit extraction and slicing can be implemented as wire instructions i.e., static shift instructions. Consider the implementation of the logical left shift instruction `sll` described in Figure 6, which consist of taking the lower 7-bit wires of `t0` and appending a 1-bit wire assigned to the value *zero* in order to produce `t1`. Curiously, single-bit constant values such as *zero* and *one* can be created with electrical ground and voltage available throughout the device without consuming any LUT or DSP. Therefore, we leverage this knowledge to define these and other operations i.e., *constants* as wire instructions.

Instruction set. Table 1 lists the full set of compute and wire instructions in the intermediate language. Most instructions are pure, i.e., they have no side effects. The only exception is the register instruction, reg. In the absence of register instructions, programs can leverage *referential transparency*. An `add` instruction, for example, takes two arguments and writes to an output of a given type, such as:

```
c:i8 = add(a,b) @??;
```

A `reg` instruction looks similar to `add`, but it is stateful in its operation. Furthermore, the `add` instruction will write a new value to `c` each cycle (based on inputs `a` and `b`), whereas the `reg` instruction will hold its value until overwritten. For

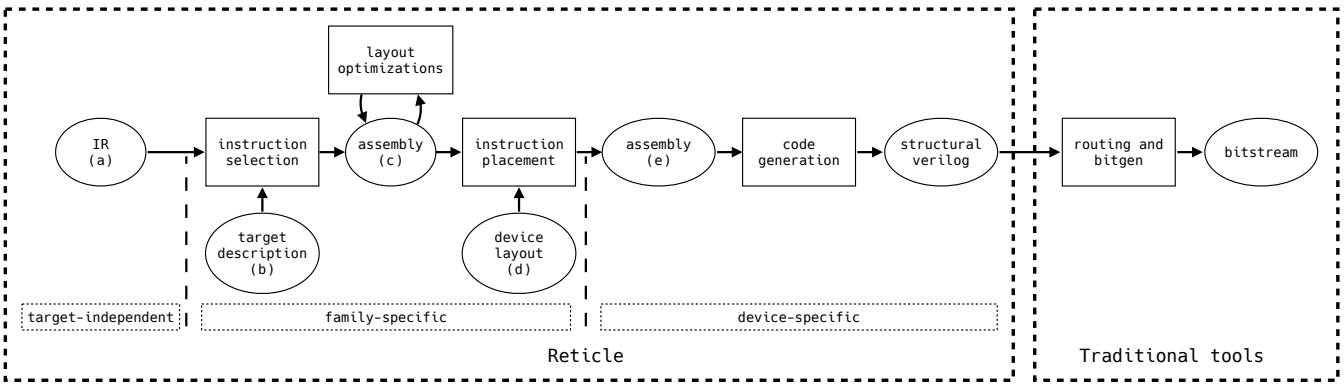


Figure 7. Reticle compilation pipeline. **(a)** The intermediate program (Figure 5a). **(b)** The target description specification (Figure 9). **(c)** The compiled assembly program (Figure 5b). **(d)** The device layout specification. **(e)** The placed assembly program with known locations (Figure 5b).

example, the following register instruction will produce a 0 as long as `b` is *False*. Similarly, once `b` is *True*, then the value of `a` will be bound to `c` every cycle.

```
c:i8 = reg[0](a,b) @??;
```

The stateful `reg` instruction is essential for allowing cycles in programs. Registers "break up" combinational cycles by stopping them from looping back within the same cycle. As we discuss in Section 6.1, a program with a cycle and without register is considered ill-formed and will be rejected.

Semantics. The primary goal behind the intermediate language is to capture the semantics of operations available in modern FPGA, while removing details of the primitives used to implement such instructions. This is accomplished by using *dataflow* and *synchronous* semantics [7]. The *dataflow* semantics are used to describe the behavior of *pure* combinational instructions [45], whereas the *synchronous* model abstracts away the details about how register instructions are updated. For example, a *synchronous* design is defined as a hardware program in which all stateful elements (registers), can only be updated on a single event trigger, i.e., a positive clock edge. Therefore, the syntax for describing such timing details is not required for programming FPGAs, resulting in a more compact representation.

4.2 The Assembly Language

The Reticle assembly language resembles the intermediate language, but it replaces high-level, abstract operators like `add` with target-specific primitives available on a particular FPGA device. Figure 5b lists the syntax for the assembly language, in which compute instructions *comp* are replaced with assembly instructions *asm*. (Reticle assembly retains the same wire instructions as the intermediate language.) As we lower to hardware targets, special-purpose hardware becomes available that can handle specialized operations, such as multiply-add, with known implementation costs (area and latency.) Although *asm* instructions are considered specialized instructions, they are still portable within an FPGA family. Devices within a family share the same primitives, varying only the total number of primitives available in them.

To capture the semantics of these varied operations, each is defined in terms of a sequence of intermediate language operations, which are then automatically composed in the compilation process. (This means that assembly operations ⊠ can be composed of one or more intermediate operations in a single instruction.) Therefore, the number of *asm* instructions is far greater than *comp* instructions, allowing different FPGA architectures to be targeted with a simpler intermediate language. For example, two intermediate operations consisting of a multiplication followed by an addition can be fused into a `muladd` (if it supported by the hardware target) and it can be expressed in assembly as:

```
y:i8 = muladd(a,b,c) @dsp(??, ??);
```

Assembly instructions also differ from compute instructions because they support location semantics *loc*. A location includes not only a primitive kind (LUTs or DSPs), but also a Cartesian x, y coordinate describing the physical placement of the operation. Each coordinate can be either a concrete expression e or a wildcard `??`, indicating that the compiler is responsible for determining the placement. While the wildcard gives the compiler the greatest flexibility, placing explicit constraints on coordinates with expressions gives front-end tools greater control over programs (and its ultimate performance). An expression can refer to variables defined in other coordinate expressions to place constraints between the placement of the two instructions. For example, an instruction location could be specified using unconstrained variables like `(x0_loc, y0_loc)`, while another instruction is constrained to always be adjacent (right after) within the same column: `(x0_loc, y0_loc+1)`. Because we

```
t0:i8 = mul(a,b) @??;
t1:i8 = add(t0,c) @??;
```

(a) Intermediate program

```
t0:i8 = mul(a,b) @dsp(??,??);
t1:i8 = add(t0,c) @dsp(??,??);
```

(b) Assembly program, cost=2

```
t0:i8 = muladd(a,b,c) @dsp(??,??);
```

(c) Assembly program, cost=1

Figure 8. Example of an intermediate program **(a)** and two equivalent assembly programs **(b,c)** with different costs.

used the same variables, the two instructions have a placement relationship: they have the same `x_loc` (column), and the second instruction is right after the first. Anecdotally, we were inspired by the Lava hardware language [5], on the benefits of incorporating layout semantics to perform layout optimizations (discussed further in related work, Section 9).

5 Compilation

Our hardware compiler performs a series of transformations to convert and optimize a source intermediate program into a target structural representation. The transformations in the compiler are described in Figure 7, including instruction selection, layout optimizations, instruction placement, and code generation. Each of these program transformations progressively increases the level of detail of the compiler target such as: target-independent, family-specific, and device-specific transformations.

5.1 Instruction Lowering

The Reticle compiler is responsible for lowering the abstract intermediate language to the concrete assembly language. The core problem is *instruction selection*, i.e., choosing a high-quality sequence of assembly instructions that have the same semantics as the original intermediate instructions. A key consequence of Reticle's design is that the problem is similar to instruction selection in a traditional software compiler [2] but applied to the hardware domain. This is not the first time instruction selection has been proposed for hardware compilation [9, 26, 27]. Whereas today's RTL toolchains rely on slow, unpredictable metaheuristics to do a similar logical-to-physical mapping [33], the Reticle compiler can leverage the large body of work on efficient, deterministic instruction selection algorithms to achieve the same effect.

Figure 8a shows an example of instruction selection in Reticle. The intermediate-language program in Figure 8a is semantically equivalent to both assembly programs in Figures 8b and 8c, assuming a target architecture that supports `mul`, `add`, and `muladd` assembly instructions. The choice of the best implementation depends on the target-specific costs of these instructions.

$$
\begin{aligned}
des \in Description &::= asm^{+} \\
asm \in Assembly &::= n[\rho, i, i](v:\tau)^{*} \rightarrow (v:\tau)\{ins^{+}\} \\
ins \in Instruction &::= v:\tau = \boxplus \mid \otimes\,[i^{*}](v^{+}) \\
\rho \in Primitive &::= \text{lut} \mid \text{dsp} \\
\otimes \in WireOp &\quad \boxplus \in CompOp \\
n \in Name &\quad v \in Variable \quad \tau \in Type \quad i \in \mathbb{Z}
\end{aligned}
$$

Figure 9. The Target Description Language.

```
reg[lut,1,2](a:i8,en:bool) -> (y:i8) {
    y:i8 = reg[0](a,en);
}

add[lut,1,2](a:i8,b:i8) -> (y:i8) {
    y:i8 = add(a,b);
}

add_reg[lut,1,2](a:i8,b:i8,en:bool) -> (y:i8) {
    t0:i8 = add(a,b);
    y:i8 = reg[0](t0,en);
}
```

Figure 10. Example of an FPGA target described using the target description language (Figure 9). This hypothetical target supports three assembly instructions (`reg`, `add`, and `add_reg`), which are implemented using LUTs and have area and latency cost of 1 and 2 respectively.

Reticle's instruction selector uses a *target definition* that describes the instructions available for a specific FPGA family. The target description gives the area and latency costs for each assembly instruction along with its semantics in terms of intermediate language instructions.

Target Description Language. Because the availability of different low-level hardware operations (and their costs) can vary across FPGA families, the Reticle compiler needs a mechanism for describing a target platform. We designed a target description language that allows succinct specification of assembly instructions supported by a given FPGA target; it is specified in Figure 9.

In FPGA terms, a target is defined as a set of devices that support the same kinds of primitives, and it is often referred as an FPGA family or series. Devices within a family can be programmed with the same set of assembly instructions, and only differ on the number of instructions that are capable to accommodate spatially. Moreover, devices only differ on the number of DSPs and LUTs supported (columns and rows) and how their columns are arranged i.e., six columns of LUTs followed by one column of DSPs.

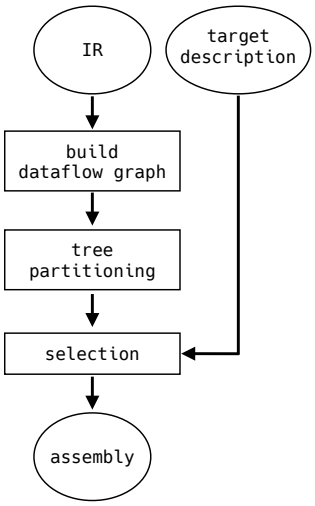


Figure 11. Instruction selection.

Concretely, a target description is defined as a list of assembly definitions *asm* that represents all the assembly instructions supported by a specific family. Each definition has an operation name n, a hardware resource ρ that the operation occupies, and area and latency costs as integers i, and the (typed) inputs and outputs to the operation. The definition also has a *body* that defines its semantics in terms of intermediate instructions. The body consists of a sequence of instructions that resemble an intermediate language program, without cycles (DAG) or *place* information. The instruction selection algorithm uses the body and costs to determine when a fragment of an intermediate language program can be replaced with an equivalent target-specific assembly instruction.

Figure 10 lists an example target in this specification language. This hypothetical target supports three assembly instructions: `reg`, `add`, and `add_reg`.

Instruction Selection. The steps for performing instruction selection are described in Figure 11. Initially, the intermediate program is converted to a *dataflow* graph (DFG), where nodes represent instructions and inputs, and edges correspond to how data flow through the program.

Once the DFG is created, the graph is partitioned into trees of intermediate instructions. The reason behind this partitioning is the fact that the DFG might contains cycles, which are not supported by tree-covering algorithms. Because the Reticle definition of well-formed programs excludes combinational cycles (see Section 6.1), we know that simply cutting on register operations is sufficient to make valid trees. The procedure for partitioning the DFG into trees consists on finding the nodes in the graph that are root candidates to make a cut. There are two conditions required to be a root node, (1) the node must be a compute instruction, and (2) its outgoing edges must be greater than one or none; compute nodes without outgoing edges represent outputs, meanwhile compute nodes with more than one outgoing edge can contain cycles and therefore they are considered as root nodes.

After tree-partitioning, the next step is *selection*, whose goal is to transform and optimize these trees of compute instructions into assembly instructions using the target description specification. Instruction selection is performed using a linear-time tree-covering algorithm originally developed for code generation in compilers [2]. The procedure is based on dynamic programming, using previous solutions to create better solutions at every node while traversing the tree in a postorder fashion. Then, the solutions (assembly instructions) from every tree are composed to produce a final assembly program. The assembly instructions in this program have unknown locations (coordinate holes) that are further optimized spatially (if necessary), and later resolved by the instruction placement stage in the compiler for a specific device.

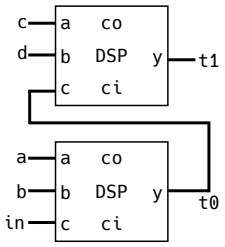
5.2 Layout Optimizations

After instruction selection, the Reticle compiler can further optimize assembly programs by placing them into high-performance spatial layouts. Layout optimizations can be expressed as constraints in the assembly language, using coordinate expressions. The relative placement of target-specific operations can have a large impact on the efficiency of a program. For example, by placing DSP-mapped operations within the same column, programs can take advantage of *DSP cascading:* leveraging high-speed routing resources available within DSP columns [42]. Hardware support for DSP cascading is widely available in most architectures today, including FPGAs designed by Intel [23], Xilinx [47], Lattice [30], and Achronix [1].

Figure 12a shows an example containing a pair of `muladd` instructions without any layout constraints. There are multiple valid layout candidates for this program; however, the version in Figure 12b, which places the operations vertically adjacent in the same DSP column, is far faster than one that scatters the operations across different columns or more distant within the same column.

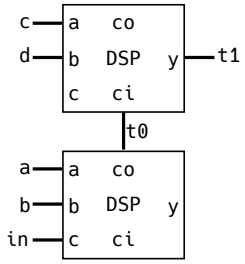
A Reticle assembly program can express this layout optimization as a layout constraint using expressions for the placement coordinates on each instruction. By using `x` as the column for both operations and `y` and `y+1` as the row expression, the assembly program describes a placement of neighboring DSPs. Moreover, the semantic of the `muladd_co` assembly instruction means that not only the DSP is configured to perform the `muladd` operation but must use the cascade port (`co`) instead of the default port (`y`) for the result. Similarly, the `muladd_ci` instruction uses the cascade input port (`ci`) instead of the default port (`c`) for the partial sum.

```
t0:i8 = muladd(a,b,in) @dsp(??,??);
t1:i8 = muladd(c,d,t0) @dsp(??,??);
```



(a) Without cascading, regular routing

```
t0:i8 = muladd_co(a,b,in) @dsp(x,y);
t1:i8 = muladd_ci(c,d,t0) @dsp(x,y+1);
```



(b) With cascading, high-speed routing

Figure 12. Example of optimizing the layout of an assembly program **(a)** using instruction cascading. In **(b)**, the unknown location specifiers ("??") have become parametric layout expressions over x and y coordinates. They imply the adjacency constraint in y, while still being place-able almost anywhere. These constraints can be solved later, during the *instruction placement* step, for a given device.

Notably, this and other parameterizable layout optimizations can be *ported* within an FPGA family using our assembly language, and later solved in the compilation pipeline i.e., *instruction placement*, for maximum portability.

5.3 Instruction Placement

After instruction selection and layout optimizations have taken place, all assembly instructions must be placed in a valid position on the target device. Therefore, the placement procedure consists of converting a family-specific program (unresolved locations) into an equivalent device-specific program (resolved locations) as described in Figure 7; finding a unique value for each coordinate variable used, as well as filling in all wildcards (??).

Deciding a physical layout consists of mapping all assembly instructions to specific FPGA resources, for a specific target FPGA. Each instruction will already have undergone selection, so the task is reduced to finding a mapping for each LUT instruction to an available LUT slice and each DSP instruction to an available DSP slice. All modern FPGAs are constructed as columns of resources; the layout engine takes as input the layout of the target FPGA — specifically, which columns are DSPs and LUTs, and how many entries or slices those columns have.

Notably, LUT column slices are different from DSP slices, due to the fact that LUT slices host more than one programmable resource. We formulate the placement problem in terms of these slices.

To solve and optimize layout, we use the Z3 SAT solver [13]. The layout problem is expressed as a series of constraints for each instruction, which are fed to the solver. Z3 quickly finds a valid coordinate assignment for each instruction, subject to the following constraints:

- The y-coordinate must match a column of the appropriate resource (DSP vs LUT);
- The x-coordinate must be between 0 and the maximum number of resources for that type of column;
- If there is a relative constraint placed (as described in Section 5.2) such that this instruction must follow another at y_1, then the y-coordinate must be at $y_1 + 1$;
- All instruction resources are unique (this instruction's coordinates cannot match any other instruction's coordinates).

If Z3 cannot find a valid placement for every instruction, placement fails.

Once a valid placement is found for each instruction, the layout engine optionally performs a series of shrinking passes as an optimization. It computes the highest x- and y-coordinate for each resource type, takes this as a maximum area, then uses a binary search to successively re-run placement with an artificially reduced area. If it succeeds, the next iteration shrinks again; if it fails, binary search is repeated in the new interval. The end result is a more compact physical layout on the FPGA.

5.4 Code Generation

The goal of code generation is to expand assembly and wire instructions into structural Verilog with layout annotations (Figure 2c). Because of the work of our prior compiler passes, this step is purely one of generation — we simply need to create valid Verilog that reflects our accumulated decisions. While this transformation is more complex than a conventional assembly to a binary format, it conceptually serves the same role — converting to a format that can be given to program a hardware target.

Instructions have been selected, optimized, and placed; now, based on the resources previously chosen for each instruction, they are expanded to a set of primitive LUTs or DSPs. DSP-based instructions are converted into a DSP primitive with a proper configuration in terms of ports and attributes to execute the desired instruction. On the other hand, LUT-based instructions require configuring a LUT for every

```
t0:i8 = const[4];
t1:i8 = add(t1,t0) @??;
```

(a) Ill-formed

```
t0:bool = const[1];
t1:i8 = const[4];
t2:i8 = add(t3,t1) @??;
t3:i8 = reg[0](t2,t0) @??;
```

(b) Well-formed

Figure 13. Example of an ill and well formed program. A well-formed program only allows cycles when stateful instructions such as `reg` are present in the path of the cycle.

bit of computation. The reason for this is that these primitives produce a single-bit output and not a full word. (For example, one 8-bit integer operation requires 8 LUTs.) Additionally, there are instructions e.g., addition or subtraction that require other primitives also present within a LUT slice such as carry chains. In any case, each primitive is annotated with the coordinate result obtained in the *instruction placement* step.

Not every instruction will result in instantiating LUTs or DSPs. As we expect, wire operations consume no area to execute (they simply require different wiring). These instructions are generated as direct structural Verilog expressions without location information.

6 Implementation

We implemented a Reticle compiler in 8662 LoC in Rust, together with a Verilog AST library (2486 LoC). We used this AST library for code generation. Additionally, a target library describing the assembly instructions supported by the Xilinx UltraScale FPGA is written in 444 lines, using our *target description language* (TDL). The instruction placement is implemented in 117 lines of Python, using the Z3 bindings.

The following two subsections explain details about the well-formedness criterion and the interpretation of programs.

6.1 Well-Formedness

In hardware design, programs typically need to avoid *combinational loops:* register-free cycles in the wiring graph that would produce undefined behavior [39]. In Reticle, this constraint manifests as a well-formedness criterion. The dependency graph for a well-formed program, in both the intermediate and assembly language variants, must be acyclic (a dag) when register instructions (`reg`) are removed. This section describes how we define and check this criterion.

Figure 13 shows examples of ill-formed and well-formed Reticle intermediate language programs. Both programs attempt to increment a stored value by a constant value 4. And both programs contain dependence cycles: in general, cycles are required for instructions to reuse their own outputs as arguments later in time. However, Figure 13b's cycle includes a `reg` instruction while Figure 13a's has a combinational (register-free) loop.

Algorithm 1: Reticle interpreter.

```
1  function Interpreter(trace, program)
2      (env, P, R) ← WellFormedCheck(program);
3      (inputs,outputs) ← GetPortNames(program);
4      T ← Trace();
5      foreach step_in ∈ trace do
6          env ← Update(env, step_in, inputs);
7          env ← Eval(env, P);
8          step_out ← Step(env, outputs);
9          T ← Push(T, step_out);
10         env ← Eval(env, R);
11     end foreach
12     return T;
13 end function
```

The Reticle implementation checks well-formedness by forming a dependence graph for a given function, where the vertices are instructions and the edges are definition–use relationships. It then sorts nodes in topological order, excluding `reg` instructions. If the sort procedure succeeds, the program is well-formed.

Reticle differs from many traditional hardware tools in rejecting programs with combinational loops. Many interpreters (a.k.a. simulators) for hardware description languages (HDLs) such as Verilog and VHDL silently produce *undefined* or *x-values* instead of producing errors [46]. Hardware engineers must therefore carefully avoid creating these cycles or risk obscuring serious bugs. We instead opt to reject these programs ahead of time to avoid the need to handle this undesired behavior during compilation and interpretation.

6.2 Interpreter

To clearly define the meaning of a Reticle program, we define an interpreter in Algorithm 1 that evaluates a function *program* by stepping through a sequence of values defined in an input *trace*[1] and producing an output trace T. This gives users a fast, convenient way to debug their programs without having to actually program an FPGA.

The first step in the interpreter consists of checking that a program is well-formed (line 2). This procedure topologically sorts the instructions in the function (see Section 6.1) and returns two sorted instruction queues and an environment *env* containing the initial value of every register instruction. The returned queues include one queue of *pure* instructions P and another queue R with register instructions.

For every step in the input *trace*, the interpreter updates all *inputs* variables in the environment with new values *step_in*

[1] A "trace" is a general term for a map of values present in a hardware circuit. Each variable (circuit element) in the trace has an associated value for every clock cycle in the domain. An *input* trace gives a complete specification for a circuit's inputs, for every cycle, while an *output* trace does so for the outputs.

(line 6). Then, pure instructions P are evaluated under the current environment (line 7). Next, a new step value is created for all *outputs* variables and pushed into the output trace T (lines 8 - 9). Finally, register instructions R are evaluated, updating the environment for the step in the *trace*.

7 Evaluation

We evaluated Reticle by generating programs for linear algebra operators and control coroutines (Section 7.1), and then compiled them to structural Verilog with layout annotations (Figure 2c) using the compilation pipeline described in Section 5. We also compiled these benchmarks to two behavioral Verilog baselines for a standard vendor toolchain, and compared their compilation time and the quality of the resulting hardware.

Furthermore, the two behavioral Verilog baselines include: (1) one using standard, portable Verilog, and (2) an advanced version using vendor-specific synthesis hints. The latter represents the use of *ad hoc* and vendor-specific Verilog language extensions that can tune the toolchain to do a better job of mapping the program to the FPGA's fixed-function resources, and it represents significant implementation effort beyond standard RTL design. We generate these baselines by transforming Reticle programs using translation backends that emit code resembling standard behavioral Verilog.

We use a Xilinx xczu3eg-sbva484-1 FPGA, with 360 DSPs and 71K LUTs, as a target device. For the baseline RTL toolchain, we use Xilinx's Vivado 2020.1.

7.1 Benchmark Description

We use three benchmarks, intended to represent three distinct facets of Reticle: a tensor addition kernel `tensoradd` demonstrates vectorization, a dot product implementation `tensordot` demonstrates fused operations and cascading, and a finite state machine `fsm` demonstrates support for control-oriented programs. Each benchmark is parameterized with four sizes.

The `tensoradd` benchmark consists of an element-wise summation over four different one-dimensional tensor sizes $(128, 256, 512, 1024)$. We *pipelined* the addition operation with register instructions to get the best possible performance available in DSP primitives.

Next, `tensordot` consists of five systolic arrays [29] performing the `dot` operation over five pairs of one-dimensional tensors of four different sizes (3, 9, 18, 36).

Finally, `fsm` is based on a coroutine, implemented as a hardware finite state machine (FSM), that ranges over some number of states $(3, 5, 7, 9)$ based on input values. The motivation is to show that Reticle programs can describe control-oriented programs normally found in hardware processor schedulers and protocols. More importantly, these programs can only be implemented on LUTs, not DSPs: conditional branching requires multiplexing (the `mux` instruction in Reticle), which it is implemented using only LUT-based logic.

7.2 Results Comparison

Compiler speed. The leftmost plots in Figure 14 compare the compilation time for Vivado (labeled `base` for standard Verilog and `hint` for directive-laden Verilog) and our compiler (`reticle`), when compiling and placing (layout) programs for every benchmark described in Section 7.1. The Reticle compiler is between 10 and 100 times faster than Vivado. By starting with programs at a lower level of abstraction, the Reticle compiler is solving a simpler problem than a traditional HDL toolchain like Vivado. The Reticle compiler focuses exclusively on selecting and configuring the FPGA's coarse-grained heterogeneous resources; an RTL toolchain also attempts to perform bit-level logic synthesis [8] to transform behavioral descriptions into structural realizations, which is important for traditional circuit generation but does not directly affect the mapping to modern units like DSPs.

The compilation performance gains in linear algebra benchmarks (`tensoradd`, `tensordot`) monotonically decreases as the sizes of the tensors grow, which translates into more DSPs to be placed by Reticle's SMT-based layout mechanism. On the other hand, the speedup obtained when compiling the `fsm` benchmark is somewhat average due to the fact the number of used LUTs are relatively low.

Run-time performance. The second plots in Figure 14 show run-time speedup for Reticle over Vivado, which is the ratio between the running times for the generated FPGA-based programs from the different compilers. Here, a running time is the *critical path* of the hardware circuit, which determines the maximum clock frequency at which hardware operates. For `tensoradd`, Reticle-generated programs are faster than the standard Vivado baseline for all tensor sizes because of the performance advantages of using the hardened units in DSPs compared to LUT-based logic. Vivado's heuristics fail to exploit DSPs at all using a pure behavioral description (`base`); Reticle, in contrast, maps the program to DSP hardware deterministically.

Surprisingly, even though there is hardware support for vectorization in every DSP of Xilinx FPGAs, Vivado fails to use this feature when using behavioral representation even in the presence of compiler hints. Vivado fails to exploit vectorization even for this simple, dependency-free parallel workload. Reticle successfully selects vectorized DSP configurations in every case. While vectorized configurations make more area-efficient use of DSP resources, they are slightly slower than scalar operations on DSPs. This phenomenon explains why the hint-laden Verilog versions can be slightly faster than Reticle for some sizes: when sufficient DSP resources exist on a target, Vivado can heuristically select scalar operations (at tensor sizes 64, 128, and 256). However,

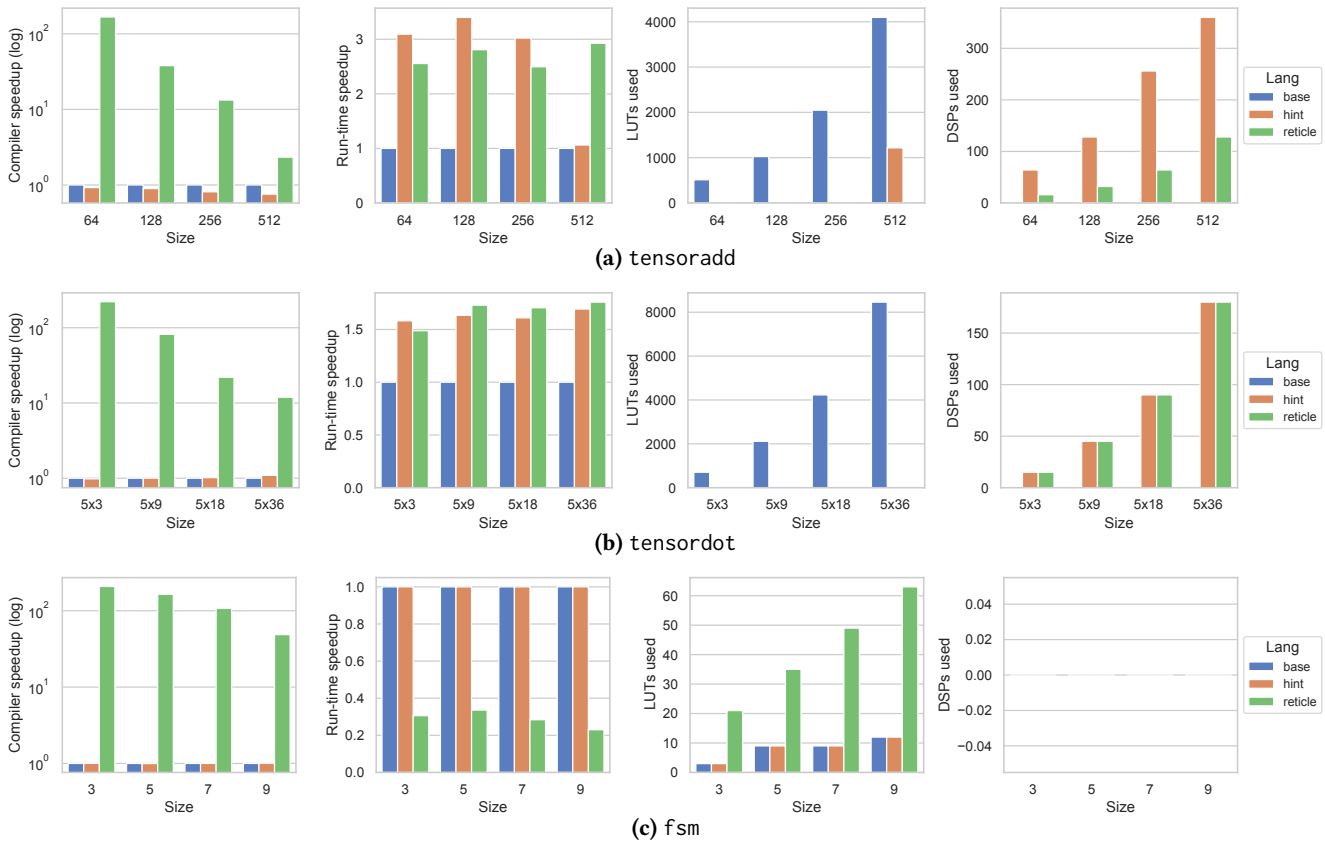


Figure 14. Compiler, run-time, and utilization results of three benchmarks, `tensoradd` **(a)**, `tensordot` **(b)**, and `fsm` **(c)**, when using behavioral Verilog (`base`), behavioral Verilog with DSP hints (`hint`), and Reticle (`reticle`).

Vivado's heuristic approach fails when the program grows larger, i.e., at a tensor size of 512: a scalar configuration exhausts all the DSPs on the target, and the toolchain silently falls back to using slower LUT-based implementations instead. At this latter configuration, the Reticle-generated vectorized program is nearly 3× faster than the Verilog program, both with and without hints. A differently annotated Reticle program could express the scalar configuration as well; we focus specifically on the vectorized version here to show the differences with a traditional HDL toolchain.

Next, the `tensordot` benchmark shows the benefits of *cascading* DSPs (Section 5.2). The latest version of Vivado (2020.1) is capable of applying this type of cascade optimization when using hints, similar to our compiler, at the expense of compilation time (up to 100 times slower in the worst case). The performance is the same for Reticle and Verilog with hints, and both outperform plain Verilog.

Lastly, the `fsm` benchmark shows the performance of control-oriented programs when mapped to LUTs. This kind of control logic is a kind of pathological case for Reticle: there is no way to use hardened logic resources like DSPs, which are Reticle's main target, and traditional HDL toolchains use complex logic synthesis optimizations to minimize the number of LUTs they require. Our aim with this benchmark is to show that Reticle can nonetheless support this kind of synthesis and that the performance is not much worse from a heavily engineered behavioral HDL toolchain. In this case, Reticle produces `fsm` programs that are slower than Verilog's results. While the Reticle compiler focuses on extracting peak performance from hardened logic units like DSPs, it nonetheless supports LUT-based compilation with much faster compilation and some performance penalty.

Utilization. The final two plots in the rows of Figure 14 compare the FPGA resources used by the generated programs. The aim here is to show how the difference in the resource binding policies for Reticle versus Verilog. With Verilog, Vivado's job is to search for *any* implementation that matches the behavioral description—any resource-binding hints are "soft" and the compiler can ignore them. In contrast, Reticle placement and resource annotations are "hard": the compiler predictably allocates exactly the kind of resource that the programmer requested.

The benchmarks' resource utilization reveals this unpredictability in Vivado as the sizes vary. In Reticle, both linear algebra benchmarks use vector instructions and chained

```
t0:i8 = mul(a,b);
t1:i8 = add(t0,c);
t2:i8 = reg[0](t1,en); // cycle 0
```

(a) Scheduled in one cycle

```
t0:i8 = reg[0](a,en);  // cycle 0
t1:i8 = reg[0](b,en);  // cycle 0
t2:i8 = mul(t0,t1);
t3:i8 = reg[0](t2,en); // cycle 1
t4:i8 = add(t3,c);
t5:i8 = reg[0](t4,en); // cycle 2
```

(b) Scheduled in three cycles

Figure 15. Example of two different *scheduling* solutions for a program that computes the expression $a * b + c$.

instructions (`mul` followed by an `add`) that the compiler deterministically maps to DSPs. Vivado performs a heuristic mapping based on the availability of resources, resulting on unpredictable behavior that, for example, silently replaces DSPs with LUTs in the largest size of `tensoradd`.

8 Discussion

This section describes the requirements and optimization opportunities for front-end tools when targeting Reticle.

8.1 Requirements

When compiling from a higher-level language or tool, the following compilation steps may be necessary depending on the features and abstractions of the source tool. Reticle is built around instructions and does not have higher-level features for control-flow, dynamic scheduling of operations, or ambiguous resource sharing; these features, if present, must be compiled down to Reticle instructions. Notably, the absence of these features in Reticle does not exclude any categories of hardware design, but rather requires that higher-level features are translated to explicit, "structural" implementations.

Control flattening. This step involves flattening any control structures available in the high-level languages to Reticle instructions. For example, if-then-else constructs or phi-nodes can be lowered to `mux` instructions as:

```
t0:i8 = mux(cond,a,b);
```

Scheduling. This task consists of choosing when abstract operations run by mapping them onto clock cycles and inserting registers. Scheduling decisions impact the performance, including throughput and run-time, of programs. For example, Figure 15 describes two schedules for a program that computes the abstract expression $a \times b + c$. The Reticle program in Figure 15a produces a result every clock cycle, while the program in Figure 15b does so every three clock cycles. Depending on the *target* primitives, certain schedules can be more profitable than others, because the register distribution within DSPs and LUTs slices varies considerably.

```
t0:i8 = add(a,b);
```

(a) Sequential

```
t0:i8 = add(a,b);
t1:i8 = add(c,d);
t2:i8 = add(e,f);
t3:i8 = add(g,h);
```

(b) Parallel

Figure 16. Example of two different *resource sharing* strategies for a program that computes four additions. **(a)** The sequential implementation, using one instruction in four units of time, and **(b)** the parallel implementation, using four instructions in one unit of time.

Resource sharing. This step includes assigning abstract operations to Reticle instructions, which are eventually lowered to physical resources by the Reticle compiler. Resource sharing strategies often involve space-time trade-offs and, similar to scheduling, affect program performance. For example, Figure 16 shows two different strategies for a program that computes four additions. The program described in Figure 16a uses one `add` instruction sequentially to compute the four additions, taking four units of time, while the program in 16b uses four instructions for computing all additions in one unit of time (in parallel). If the input tool does not distinguish between these cases, or uses the ambiguity for optimizations, the final space/time decision must be made before outputting Reticle code.

8.2 Optimizations

The following compilation steps are not required to generate a Reticle program; however, they provide important opportunities for higher-level tools to take advantage of information present in the source program and use it for optimization.

Vectorization. The goal of this optimization is to combine independent scalar instructions, that are scheduled at the same clock cycle, into vector instructions (Section 4.) Front-end tools can promote the use of vector instructions in Reticle by using vector types; alternatively, more complex optimizations can attempt to automatically combine scalar operations into vector expressions. The benefits of vectorization are twofold: faster programs, due to high-performance primitives (i.e, DSPs), and better resource utilization, because more operations are mapped to the same primitive. For example, the two programs shown in Figure 17 describe the result of this optimization. Generating individual instructions for each arithmetic operation is valid, but vectorization can provide large gains in performance and efficiency.

Resource binding. The purpose of this optimization is to control how instructions are bound to primitives. The Reticle IR supports such control via the resource annotations

```
t0:i8 = add(a,b);
t1:i8 = add(c,d);
t2:i8 = add(e,f);
t3:i8 = add(g,h);
```

(a) Scalar program

```
t0:i8<4> = add(a,b);
```

(b) Vector program

Figure 17. Example of two equivalent programs computing four additions in parallel: a scalar program (unoptimized) and a vector program (optimized).

```
t0:i8 = add(a,b) @lut;
```

(a) LUT-based instruction

```
t0:i8 = add(a,b) @dsp;
```

(b) DSP-based instruction

Figure 18. Example of two different *resource binding* options for the same instruction.

`dsp` or `lut`, as described in Figure 18a and 18b respectively. These annotations are not required; if omitted, the Reticle compiler will assign resources according to internal metrics as described in Section 5. Interestingly, these constraints can be exploited by higher-level tools to optimize programs for metrics the Reticle compiler does not, by default, accommodate. For example, rather than prioritizing performance, front-end tools can optimize the power-profile of a program by changing these annotations.

9 Related Work

Reticle builds on a body of work on applying programming languages techniques for hardware programming.

Instruction selection for hardware design. While it is not mainstream, other compilers have used instruction selection and similar techniques to optimize hardware designs [26]. A compiler based on *silicon instruction sets* [27] powered the design of six chips at IBM, for example, and a similar approach has been applied to FPGAs [9]. However, this work primarily targets synthesizing real silicon, and even the FPGA-focused variants focus solely on programming LUTs. It does not attempt to program DSP units. Reticle is the first compiler work we are aware of to target modern FPGAs and their fixed-function hardware resources.

Layout in spatial programming. Some work inspired by functional geometry [19] designs combinators that express both the behavior and the layout of hardware circuits [44] and, in Lava [5], FPGAs. Lava differs from other FPGA-based programming models by directly describing to LUT-based implementations instead of behavioral models, yielding more efficient FPGA implementations. Instead of layout combinators, Reticle assembly uses layout constraints to express and optimize placement information. Chlorophyll [37] is a different system that also uses layout constraints—in its case, to program a 2D array of simple CPUs.

High-level languages for hardware programming. A range of recent languages have aimed to improve the programmability of custom hardware accelerators, including parallel pattern-based programming in Spatial [28], novel type systems for expressing hardware-level concerns [14, 15, 34], and compiler infrastructure for accelerator generators [35]. This category of work focuses on productive high-level programming, while Reticle aims at the other end of the abstraction hierarchy: providing an efficient, low-level hardware abstraction. We see Reticle as a better compilation target for these languages compared to traditional HDLs.

Improving hardware description languages. There is a rich body of work on improving the programmability of hardware description languages (HDLs), from Bluespec [36] to more recent embedded languages for register transfer level (RTL) design [4, 12, 25]. Other work has focused on formal verification, including new Bluespec-descended HDLs [6, 10], verified interactions between software and hardware [32], and fuzz testing that has revealed frequent bugs in FPGA tools that arise because of the complexity of traditional behavioral HDLs [20]. All of these languages and tools, however, target behavioral descriptions of arbitrary circuits—none of them address the problem of describing and mapping to fixed-function FPGA resources.

There is, however, previous work on finding optimal DSP-based implementations for FPGAs [40], using standalone data flow graphs and covering only certain arithmetic expressions e.g, multiplication and addition instead of using a general-purpose complete IR.

Hardware intermediate representations. Some intermediate languages have emerged for RTL design toolchains, such as FIRRTL [24] and LLHD [43]. Like Verilog and VHDL, however, these IRs have behavioral semantics that require traditional, heavyweight digital design toolchains to optimize and lower to either silicon or FPGA targets. They generally do not have a way to capture the vendor-specific FPGA resources that Reticle targets and therefore, unlike Reticle, rely on vendor-supplied toolchains to heuristically infer mappings to modern FPGA hardware.

10 Conclusion

The observation that hardware design toolchains have much to learn from mainstream compiler technology is at least 37 years old [26]. However, we believe the opportunity is particularly ripe for modern FPGAs. If compiler advancements can make FPGA programming feel more like software programming, with deterministic results and compile times in seconds instead of days, we may unlock the potential of reconfigurable hardware specialization as part of everyday software development.

Acknowledgements

We would like to thank members of the PLSE group at the University of Washington and the CAPRA group at Cornell University for early discussions on Reticle. Many thanks to the anonymous PLDI reviewers for suggesting important additions. We would also like to thank Pedro Torruella for reviewing the paper, and Rachit Nigam, Samuel Thomas, and Chris Lavin for their invaluable help and support.

This work was supported in part by the Center for Intelligent Storage and Processing in Memory (CRISP), a Semiconductor Research Corporation (SRC) program co-sponsored by DARPA. It was also supported by the Real Time Machine Learning (RTML) NSF and DARPA program, and the NSF award #1518703.

References

[1] Achronix. 2019. Speedster7t IP Component Library User Guide. UG086. https://www.achronix.com/sites/default/files/docs/Speedster7t_IP_Component_Library_User_Guide_UG086.pdf.

[2] Alfred V. Aho and Mahadevan Ganapathi. 1985. Efficient Tree Pattern Matching (Extended Abstract): An Aid to Code Generation. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '85)*. Association for Computing Machinery, New York, NY, USA, 334–340. https://doi.org/10.1145/318593.318663

[3] Ananda Samajdar, Tushar Garg, Tushar Krishna, and Nachiket Kapre. [n.d.]. Scaling the Cascades. https://git.uwaterloo.ca/watcag-public/fpga-cascades-rtl.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. https://doi.org/10.1145/2228360.2228584

[5] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 174–184. https://doi.org/10.1145/289423.289440

[6] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[7] Thomas Braibant and Adam Chlipala. 2013. Formal Verification of Hardware Synthesis. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 213–228.

[8] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–40.

[9] Timothy J. Callahan, Philip Chong, André DeHon, and John Wawrzynek. 1998. Fast Module Mapping and Placement for Datapaths in FPGAs. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '98)*. Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/275107.275132

[10] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

[11] Philip Colangelo, Nasibeh Nasiri, Asit Mishra, Eriko Nurvitadhi, Martin Margala, and Kevin Nealis. 2018. Exploration of Low Numeric Precision Deep Learning Inference Using Intel FPGAs. arXiv:1806.11547 [cs.DC]

[12] D. Dangwal, G. Tzimpragos, and T. Sherwood. 2020. Agile Hardware Development and Instrumentation With PyRTL. *IEEE Micro* 40, 4 (2020), 76–84. https://doi.org/10.1109/MM.2020.2997704

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[14] Jan de Muijnck-Hughes and Wim Vanderbauwhede. 2019. A Typing Discipline for Hardware Interfaces. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:27. https://doi.org/10.4230/LIPIcs.ECOOP.2019.6

[15] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 408–422. https://doi.org/10.1145/3385412.3385983

[16] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA.2018.00012

[17] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA.2018.00012

[18] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.* 33, 4, Article 144 (July 2014), 11 pages. https://doi.org/10.1145/2601097.2601174

[19] Peter Henderson. 1982. Functional Geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh, Pennsylvania, USA) *(LFP '82)*. Association for Computing Machinery, New York, NY, USA, 179–187. https://doi.org/10.1145/800068.802148

[20] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 277–287. https://doi.org/10.1145/3373087.3375310

[21] Intel. 2020. Intel Agilex FPGA Architecture. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/intel-agilex-fpgas-deliver-game-changing-combination-wp.pdf.

[22] Intel. 2020. Intel HLS Compiler: Fast Design, Coding, and Hardware. WP-01274.https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01274-intel-hls-compiler-fast-design-coding-and-hardware.pdf.

[23] Intel. 2020. Intel Stratix 10 Variable PrecisionDSP Blocks User Guide. UG-S10-DSP. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-dsp.pdf.

[24] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design* (Irvine, California) *(ICCAD '17)*. IEEE Press, 209–216.

[25] S. Jiang, P. Pan, Y. Ou, and C. Batten. 2020. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro* 40, 4 (2020), 58–66. https://doi.org/10.1109/MM.2020.2997638

[26] S. C. Johnson. 1983. Code Generation for Silicon. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 14–19. https://doi.org/10.1145/567067.567070

[27] K. Keutzer and W. Wolf. 1988. Anatomy of a Hardware Compiler. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 95–104. https://doi.org/10.1145/53990.54000

[28] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. *SIGPLAN Not.* 53, 4 (June 2018), 296–311. https://doi.org/10.1145/3296979.3192379

[29] H. T. Kung. 1982. Why Systolic Architectures? *Computer* 15, 1 (Jan. 1982), 37–46. https://doi.org/10.1109/MC.1982.1653825

[30] Lattice. 2020. sysDSP Usage Guide for Nexus Platform. Technical Note FPGA-TN-02096-1.1.

[31] C. Lavin and A. Kaviani. 2018. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 133–140.

[32] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1041–1053. https://doi.org/10.1145/3314221.3314622

[33] A. Mishchenko, S. Chatterjee, and R. K. Brayton. 2007. Improvements to Technology Mapping for LUT-Based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (2007), 240–253. https://doi.org/10.1109/TCAD.2006.887925

[34] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[35] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A Compiler Infrastructure for Accelerator Generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712

[36] R. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

[37] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures. *SIGPLAN Not.* 49, 6 (June 2014), 396–407. https://doi.org/10.1145/2666356.2594339

[38] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 13–24. https://doi.org/10.1145/2678373.2665678

[39] R. B. Reese and M. A. Thornton. 2006. *Introduction to Logic Synthesis using Verilog HDL*. https://doi.org/10.2200/S00060ED1V01Y200610DCS006

[40] B. Ronak and S. A. Fahmy. 2016. Mapping for Maximum Performance on FPGA DSP Blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 4 (2016), 573–585. https://doi.org/10.1109/TCAD.2015.2474363

[41] Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style.. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, USA) *(LFP '92)*. Association for Computing Machinery, New York, NY, USA, 288–298. https://doi.org/10.1145/141471.141563

[42] A. Samajdar, T. Garg, T. Krishna, and N. Kapre. 2019. Scaling the Cascades: Interconnect-Aware FPGA Implementation of Machine Learning Problems. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 342–349.

[43] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 258–271. https://doi.org/10.1145/3385412.3386024

[44] Mary Sheeran. 1984. MuFP, a Language for VLSI Design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LFP '84)*. Association for Computing Machinery, New York, NY, USA, 104–112. https://doi.org/10.1145/800055.802026

[45] Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:21. https://doi.org/10.4230/LIPIcs.SNAPL.2019.7

[46] Mike Turpin. 2003. The Dangers of Living with an X (bugs hidden in your Verilog). https://developer.arm.com/documentation/arp0009/a/

[47] Xilinx. 2020. UltraScale Architecture DSP Slice User Guide. UG579. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[48] Xilinx. 2020. Versal:The First Adaptive Compute Acceleration Platform (ACAP). WP505. https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf.

[49] Xilinx. 2020. Vitis Unified Software Development Platform 2020.1 Documentation. https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/introductionvitishls.html.