Programming Abstractions and Efficient Compilation Techniques for Modern FPGAs

Luis Vega

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2022

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Programming Abstractions and Efficient Compilation Techniques for Modern FPGAs

Luis Vega

Chair of the Supervisory Committee:
Luis Ceze
Paul G. Allen School of Computer Science & Engineering

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 *lookup tables*. Nowadays, FPGAs are highly heterogeneous architectures that support a wide variety of compute operations such as scalar, vector, fused, and floating-point arithmetic together with different kinds of programmable memories. Unfortunately, the behavioral semantics available in hardware languages today cannot effectively capture these architectural advances, resulting in inefficient programs that are missing all the benefits of specialization. An example of this abstraction gap is that vector operations cannot be described *behaviorally* for targeting vector hardware (SIMD) available in modern FPGAs.

This thesis describes 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*. The design goal of the intermediate language is to describe behavior, while the assembly language aims for layout. Furthermore, I demonstrate how to lower intermediate programs to assembly programs, using *instruction selection* which can be both faster and deterministic compared to existing *technology mapping* approaches. I use Reticle to implement compute centric benchmarks, such as 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. Additionally, I show how using Reticle's memory instructions can lead to 5× performance improvement on an existing encryption application (AES).

Contents

Acknowledgments

This thesis covers work done during the last two years of my PhD. I am fortunate to have experienced the mentorship of Luis Ceze and Dan Grossman throughout this journey, including the darkest moments. I will be forever grateful for the guidance and support. I was extremely lucky to also have the advising of Zach Tatlock and Adrian Sampson along the way, from which I have learned tremendously.

Doing a PhD truly takes a village. Thanks to all my friends from SAMPA, SAMPL, and PLSE groups. It certainly would not be the same without you. And finally, thanks to my family, Jorbelly and Alejandra, for being there for me this whole time.

Chapter 1

Introduction

Field-programmable gate arrays (FPGAs) have emerged as a relief from stagnating performance on CPUs and GPUs [PCC+14, FOP+18a, FOP+18b]. 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 lookup tables, modern FPGAs incorporate an array of heterogeneous, special-purpose “hardened” units that implement commonplace functionality: memories, arithmetic units, and complex interconnects [Xil20c, Int20a]. 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 [RF16].

On the other hand, the mainstream languages to program FPGAs is based on behavioral hardware description languages (HDLs), which are not only used today for programming FPGAs

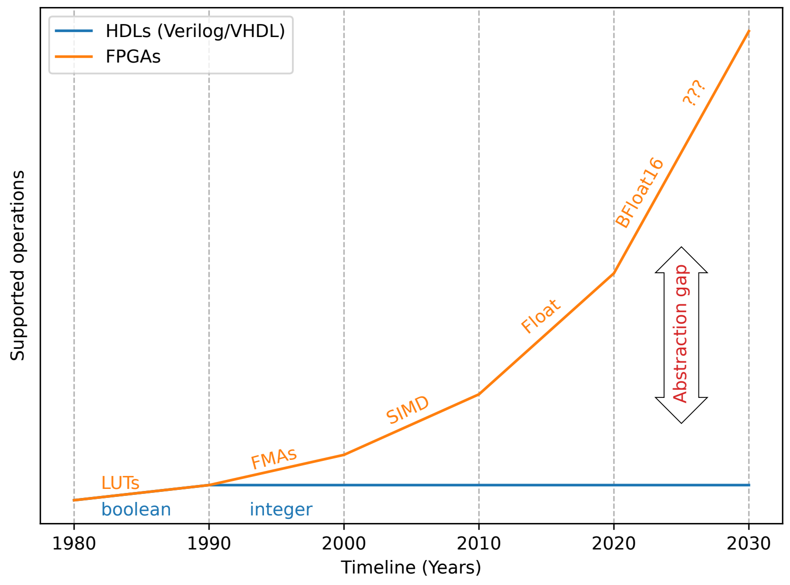


Figure 1.1: FPGA and HDLs evolution over the years. Traditional HDLs are falling behind on capturing latest architectural advances in FPGAs, making programming modern FPGAs efficiently really hard.

but also for designing ASICs. In fact, FPGAs are designed and programmed with the same programming language (e.g., Verilog). Moreover, these languages are part of an IEEE standard, designed and maintained by a committee mostly concerned with the task of modelling and describing hardware circuits efficiently, which has little to do with optimally programming FPGAs. Figure 1.1 describes how FPGA primitives have evolved over the years in terms of supported operations compared to HDLs which has stayed mostly stagnant. This in turn creates an abstraction gap between the programming language and the underlying hardware, making it really hard, and in some cases impossible, to express newer operations (e.g., bfloat16 multiplication) behaviorally.

1.1 FPGA Programming Today

Currently, FPGAs are programmed using HDLs, either by describing hardware circuits by hand or generating them from higher level languages [Xil20d, Int20b, NAT$^+$20, NTLS21, DTS20, BVR$^+$12]. These languages, however, rely on behavioral HDLs as *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.

Moreover, behavioral HDLs like Verilog and VHDL do not have a way to efficiently represent operations supported by modern FPGAs' 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 [LK18]:

> 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.

1.2 The Future of FPGA Programming

The fact that FPGAs were initially created for emulating hardware behavior, instead of a CPU alternative for efficient computing, has greatly shaped their programming foundation from the beginning. The programming challenge, since their invention, was based on how to compile an existing language used for designing and modeling hardware into an array of compute primitives. This approach has greatly limited the opportunity for unlocking the potential available in modern FPGAs, because the core language was never designed to get the most out of the architecture. Instead, the goal was to support an already existing language that has nothing to do with programming FPGAs. In fact, today FPGAs are designed using Verilog, then manufactured into a chip, and finally programmed in Verilog again.

The abstraction used to program FPGAs should, at a minimum, be different from the one used to design them. For example, other hardware architectures such as CPUs and GPUs have greatly benefited from taking this approach. Today, CPUs and GPUs are both designed in Verilog but their low-level programming language has nothing to do with Verilog. Instead, CPUs and GPUs are programmed with assembly languages based on instruction sets that reflect the primitives available in them. Therefore, higher-level languages can use these instructions to build highly productive programming stacks on top.

The work presented in this thesis addresses the concerns raised above by proposing a new programming foundation that, compared to the traditional approach, captures the advances achieved in FPGA architectures over the last several decades and supports the requirements needed by higher-level languages. These requirements include: virtual instruction set, resource binding, and fast compilation. In the following paragraphs, we cover why these requirements are crucial for future FPGA programming stacks.

1.2.1 Virtual Instruction Set

The number of specialized primitives available in multiple FPGA architectures is constantly increasing and becoming more complex. FPGAs now support primitives that are no longer based on boolean operations, such as bfloat16 multiplication, that cannot be expressed using traditional HDLs. Future high-level compilers will require an efficient instruction set with value types and operations that captures new FPGA primitives in a target independent way, allowing compilers to perform optimizations that can target multiple architectures.

1.2.2 Resource Binding

The programming complexity of new primitives keeps increasing, making it difficult to assess the optimizations available in current FPGA toolchains. There is more than one way to perform an operation in these devices, each with its own set of tradeoffs. For example, a multiplication can be performed by either DSPs or LUTs, but their availability and performance are vastly different. Future high-level compilers require an ergonomic mechanism to promote, and constrain if necessary, the mapping of primitives that can enable structured performance, power, and area optimizations.

1.2.3 Fast Compilation

The size of FPGAs continues to grow over time, lately requiring multiple silicon dies to fully accommodate them inside a chip. This is causing traditional toolchains to take considerable time, in most cases several hours, for compiling source code into binaries. The heuristics used in these toolchains are performing search on a large space to try to find a global optimal solution. This approach was reasonable back when FPGAs were small devices, but that is no longer the case. Future programming stacks require compilers that do not perform search, but rather find a solution quickly by ranking more explicit directions from the source language.

1.3 Overview

The work presented in this thesis focuses on programming abstractions and efficient compilation techniques for modern FPGAs. Furthermore, I propose a low-level abstraction for FPGA programming, called Reticle, that capture high-performance operations supported in recent architectures. Additionally, I show how this intermediate abstraction is lowered to a target-dependent representation, also known as *assembly language*, that can be used to spatially accommodate operations into the FPGA fabric. While designing languages and compilers for FPGAs is an active area of research, this thesis makes the claim that we need a new low-level programming abstraction for effectively using all the architectural advances available in modern FPGAs. Innovation in high-level FPGA programming models is accelerating [DFH+20, NAT+20, NTLS21, KFP+18], and these new compilers need a better target than current hardware languages. In this thesis, I focus on two different aspects of this problem: language design and compiler infrastructure.

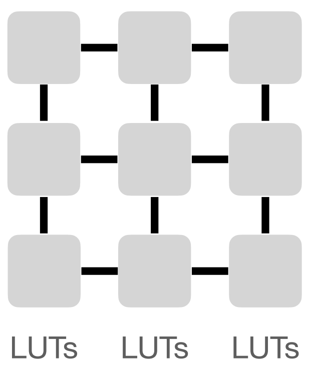
Chapter 2

Background

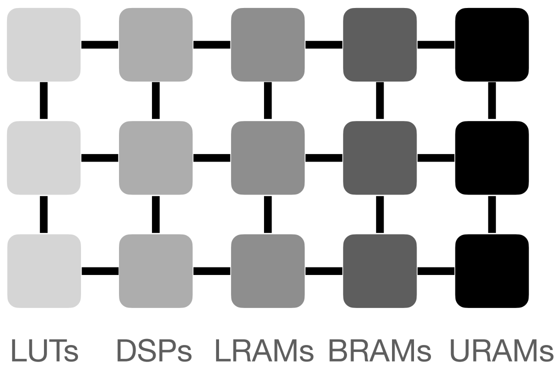
2.1 Classic vs. Modern Architectures

Although LUTs are the main building block that classically dominated FPGAs as shown in Figure 2.1a, they are not the only programmable resource available on modern FPGAs. Over the years, FPGAs have added other kinds of primitives: most prominently, digital signal processing slices (DSPs) that can execute more involved operations as described in Figure 2.1b. In modern FPGAs, DSPs are a source of heterogeneity because they support a wide variety of complex operations, such as scalar, vector, and fused integer operations and, in recent architectures, even floating-point arithmetic [CNM+18]. Although LUTs can implement arbitrary Boolean logic formulas, DSPs can perform operations faster and far more efficiently [LK18]. For example, an 8-bit *and* operation can typically be implemented using a single DSP or 8 LUTs.

In addition to compute operations, FPGA architectures have also evolved in terms of memory primitives. FPGAs now support multiple types of addressable memories with different capacity and bandwidth features [Xil20b]. Memories can be implemented using one or more primitives, similarly to how compute operations can be composed of multiple DSPs or LUTs. Moreover,



(a) Classic FPGAs.



(b) Modern FPGAs.

Figure 2.1: FPGA architecture evolution over the years.

memory bandwidth varies based on the capacity available in memory primitives [RBL+09]. For example, smaller-sized memory primitives, based on LUTs, have less memory bandwidth compared to larger-sized memories [Xil20b]. Therefore, the number of primitives used to implement a particular addressable memory description can vary significantly, impacting the performance of a design [AI19].

2.2 Hardware Synthesis

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

Moreover, hardware languages like Verilog support these two representations: *behavioral* and *structural*. For example, consider the behavioral Verilog program in Figure 2.2a. This program uses a binary expression that performs the *and* operation. One valid transformation of this program is

```
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.

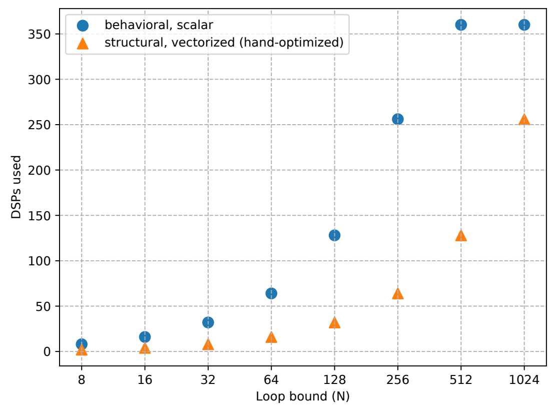
Figure 2.2: Three Verilog representations of *and* program.

shown in Figure 2.2b, 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 2.2c, 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.

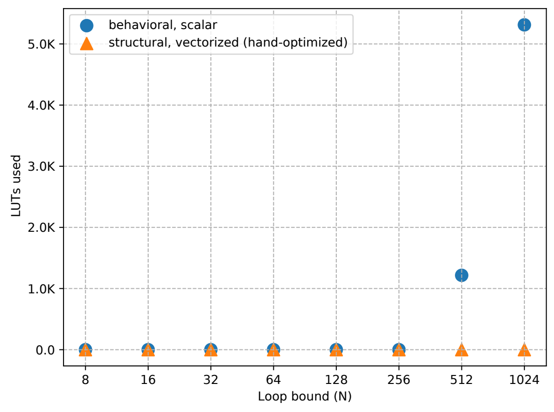
Modern FPGA hardware synthesizers heuristically map behavioral HDLs onto LUTs and DSPs based on a cost model and resource availability. The cost model is normally based on the type of the operation and integer type of the operands. For example, a synthesizer might prefer to map integer multiplications to DSPs because of the poor size and speed trade-off 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.

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

(a) Behavioral Verilog program



(b) DSP utilization.



(c) LUT utilization.

Figure 2.3: DSP **(b)** and LUT **(c)** resource utilization for multiple loop bounds (N) of the *behavioral* program described in **(a)** for adding two arrays of N elements in parallel 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).

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 2.3a, 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 2.3b and 2.3c show the number of resources consumed by this program when using the *behavioral* HDL description versus a hand-optimized *structural* implementation.

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 2.3b shows that the total number of DSPs in this particular device is already reached for size $N = 512$, although the maximum number of parallel additions 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 2.3c.

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

Chapter 3

Reticle: A Virtual Machine for Programming Modern FPGAs

Reticle [VMS+21] is an intermediate representation (IR) and compiler for FPGAs that addresses the challenges outlined in the previous chapters. Reticle aims to directly represent and optimize for the heterogeneous programmable resources available in modern FPGAs. 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 described in Chapter 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 manual and target-specific structural implementations.

The following Sections describes the Reticle language design and compiler implementation. Section 3.1 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. Subsection 3.2.1 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 [MCB07]. Subsection 3.2.4 show how our compiler implementation emits structural hardware descriptions for a specific FPGA target.

3.1 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 low-level *assembly language*, where operations correspond to physical primitives available on a specific device. Figure 3.1 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.

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

(a) The Intermediate Language

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

(b) The Assembly Language

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

$$\begin{array}{lll}
\otimes \in WireOp & \boxplus \in PrimOp & \boxtimes \in AsmOp \\
?? \in Wildcard & n \in Name & v \in Variable \\
\tau \in bool, int, \overrightarrow{int} & i \in \mathbb{Z} &
\end{array}$$

Figure 3.1: The Intermediate and Assembly Languages.

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

Figure 3.2: 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).

3.1.1 The Intermediate Language

Figure 3.1a 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) [SF92]: they consist of a flat list of instructions whose arguments are always variables v.

Wire & primitive instructions

There are two types of instructions in the language: *wire* and *primitive* instructions. While both share a common format, primitive 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 primitive instructions are based on an operation $\boxplus$ that are performed by a resource type ρ. Therefore, primitive instructions are candidate for optimizations. Figure 3.2 shows an example of wire and primitive instructions.

Primitive instructions also have an annotation @ *res* that can optionally control which kind of resource to use on the target device, from compute resources (*lut* or *dsp*) to memory resources (*lram*, *bram*, and *uram*). Additionally, 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 3.2, 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 resources. Therefore, we leverage this knowledge to define these and other operations i.e., *constants* as wire instructions.

Instruction set

Figure 3.3a lists the full set of primitive and wire instructions in the intermediate language. Most primitive instructions are pure, i.e., they have no side effects. The only exception is memory instructions. In the absence of memory instructions, programs can leverage *referential transparency*.

An `add` instruction, for example, takes two arguments and writes to an output of a given type, as shown in Figure 3.3b. On the other hand, a `reg` instruction described in Figure 3.3c looks similar to `add`, but it is stateful in its operation. Furthermore, the `add` instruction will write a new value to `y` each cycle (based on inputs `a` and `b`), whereas the `reg` instruction will hold its value until overwritten. For 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 `y` every cycle.

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 Subsection 3.1.3, a program with a cycle and without register is considered ill-formed and will be rejected.

The register instruction, however, is not the only stateful instruction. There are four more memory instructions available for accessing indexed memory locations, including `ram`, `rom`, `sram`, and `srom`. More importantly, the differences between these instructions are based on two features:

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

(a) The instruction set.

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

(b) Addition instruction.

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

(c) Register instruction.

```
y:i8 = rom(a) @??;
```

(d) ROM instruction.

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

(e) RAM instruction.

Figure 3.3: The IR instruction set **(a)** with usage examples **(b,c,d,e)**.

mutation and timing behavior. For example, the `rom` and `ram` instructions differ in mutability—`rom` being a read-only memory and `ram` being a read-write memory respectively. Another feature, based on timing behavior and denoted with the `s` prefix, describe when the instruction output is valid. For example, the `rom` and `ram` will write a new value to `y` each cycle (based on the input address `a`), whereas the `sram` and `srom` instructions will bound such value after one cycle of operation.

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 [BC13]. The *dataflow* semantics are used to describe the behavior of *pure* combinational instructions [TH19], whereas the *synchronous* model abstracts away the details about how stateful instructions are updated. For example, a *synchronous* design is defined as a hardware program in which all stateful elements i.e., 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.

3.1.2 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. The design goal of the assembly language is to provide an abstraction for laying out programs, based on the fact that *placing* operations is more efficient than using primitives directly. Figure 3.1b 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 $\boxtimes$ can be composed of one or more intermediate operations in a single instruction.) Therefore, the number of *asm* instructions is far greater than $prim$ 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 shown Figure 3.4a.

Assembly instructions also differ from primitive instructions because they support location semantics *loc*. A location includes not only a resource type (LUTs or DSPs), but also a Cartesian

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

(a) Multiply-Add instruction.

Figure 3.4: Example of an assembly instruction.

```
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 3.5: 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.

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 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.

3.1.3 Well-formedness

In hardware design, programs typically need to avoid *combinational loops:* memory-free cycles in the wiring graph that would produce undefined behavior [RT06]. 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 memory instructions are removed. The following paragraph describes how we define and check this criterion.

Figure 3.5 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 3.5b's cycle includes a `reg` instruction while Figure 3.5a's has a combinational (register-free) loop.

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 memory 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 [Tur03]. 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.

3.2 The Compiler

The Reticle 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 3.6, including instruction selection, layout optimizations, instruction placement, and code generation. Each of these program transformations progressively increases

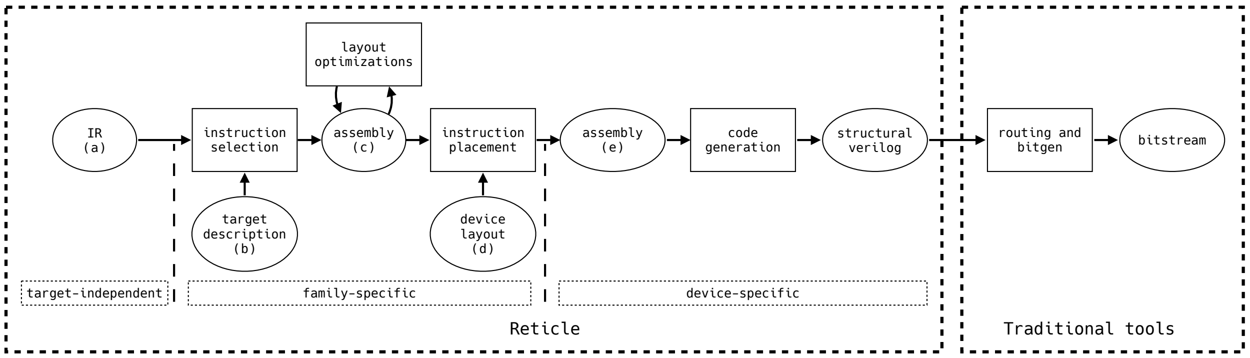


Figure 3.6: Reticle compilation pipeline. **(a)** The intermediate program (Figure 3.1a). **(b)** The target description specification (Figure 3.8). **(c)** The compiled assembly program (Figure 3.1b). **(d)** The device layout specification. **(e)** The placed assembly program with known locations (Figure 3.1b).

the level of detail of the compiler target such as: target-independent, family-specific, and device-specific transformations.

3.2.1 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 [AG85] but applied to the hardware domain. This is not the first time instruction selection has been proposed for hardware compilation [Joh83, KW88, CCDW98]. Whereas today's RTL toolchains rely on slow, unpredictable metaheuristics to do a similar logical-to-physical mapping [MCB07], the Reticle compiler can leverage the large body of work on efficient, deterministic instruction selection algorithms to achieve the same effect.

Figure 3.7a shows an example of instruction selection in Reticle. The intermediate-language program in Figure 3.7a is semantically equivalent to both assembly programs in Figures 3.7b and 3.7c, 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.

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.

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

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 3.8.

$$\begin{aligned}
des \in \mathit{Description} &::= asm^{+} \\
asm \in \mathit{Assembly} &::= n[\rho, i, i](v : \tau)^{*} \longrightarrow (v : \tau)\{ins^{+}\} \\
ins \in \mathit{Instruction} &::= v : \tau = \boxplus \mid \otimes\ [i^{*}](v^{+}) \\
\rho \in \mathit{Primitive} &::= \mathrm{lut} \mid \mathrm{dsp}
\end{aligned}$$

$$\otimes \in \mathit{WireOp} \quad \boxplus \in \mathit{CompOp}$$
$$n \in \mathit{Name} \quad v \in \mathit{Variable} \quad \tau \in \mathit{Type} \quad i \in \mathbb{Z}$$

Figure 3.8: 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 3.9: Example of an FPGA target described using the target description language (Figure 3.8). 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.

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.

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

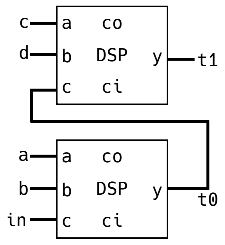
Instruction Selection

The steps for performing instruction selection include: data-flow graph (DFG) generation, tree-partitioning and selection. Initially, the intermediate program is converted to a 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 3.1.3), 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.

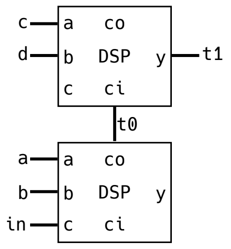
After tree-partitioning, the next step is *selection*, whose goal is to transform and optimize these trees of compute instructions into assembly instructions using the target description specification. Instruction selection is performed using a linear-time tree-covering algorithm originally developed for code generation in compilers [AG85]. 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.

```
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 3.10: 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.

3.2.2 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 [SGKK19]. Hardware support for DSP cascading is widely available in most architectures today, including FPGAs designed by Intel [Int20c], Xilinx [Xil20a], Lattice [Lat20], and Achronix [Ach19].

Figure 3.10a 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 3.10b, 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. 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.

3.2.3 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 3.6; 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 [DMB08]. 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 Subsection 3.2.2) 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.

3.2.4 Code Generation

The goal of code generation is to expand assembly and wire instructions into structural Verilog with layout annotations (Figure 2.2c). 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 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.

Chapter 4

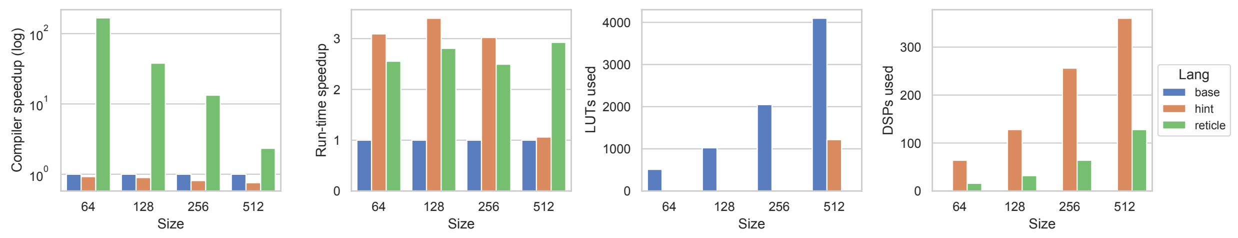
Evaluation

In order to assess the efficiency of Reticle, we performed two separate evaluations. First, we evaluate compute centric benchmarks such as linear algebra operators and coroutines. Then, we discuss the implementation results of memory descriptions, and how these can impact programs that are highly dependent on memory, specifically AES encryption.

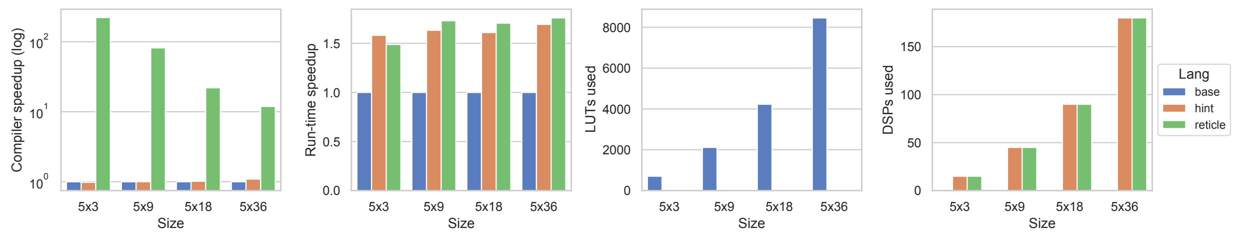
4.1 Linear algebra and coroutine benchmarks

We evaluated Reticle by generating programs for linear algebra operators and control coroutines (Section 4.1.1), and then compiled them to structural Verilog with layout annotations (Figure 2.2c) using the compilation pipeline described in Subsection 3.2.1. 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.

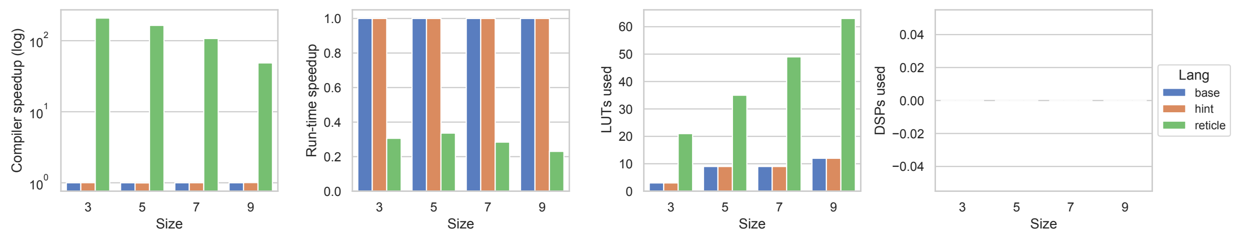
Furthermore, the two behavioral Verilog baselines include: (1) one using standard, portable Verilog, and (2) an advanced version using vendor-specific synthesis hints. The latter represents the use of *ad hoc* and vendor-specific Verilog language extensions that can tune the toolchain to



(a) `tensoradd`



(b) `tensordot`



(c) `fsm`

Figure 4.1: 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`).

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.

4.1.1 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 [Kun82] 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.

4.1.2 Results Comparison

Compiler speed

The leftmost plots in Figure 4.1 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 4.1.1. 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 [BM10] 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 4.1 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, 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 (Subsection 3.2.2). 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 4.1 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 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`.

4.2 Memory Benchmarks

In addition to compute centric benchmarks, we also evaluated the efficiency of Reticle when targeting memory primitives available in FPGAs. Furthermore, we performed two different experiments that highlight the impact of efficient memory implementations. First, we assessed two read-only memory implementations and compared them against the `srom` instruction available in Reticle. Then, we evaluated how these memories can affect a larger program such as a cryptographic encryption protocol (AES) implementation.

```
reg [15:0] ram;
always @(posedge clock) begin
  case({addr, reset})
    {1'd0, 1'b0}: ram <= 16'h7FEE;
    {1'd1, 1'b0}: ram <= 16'h7BEE;
    default: ram <= 16'd0;
  endcase
end
```

(a) Baseline.

```
reg [15:0] ram [1:0];
initial begin
  ram[0] = 16'h7FEE;
  ram[1] = 16'h7BEE;
end

reg [15:0] data;
always @(posedge clock) begin
  if (reset) begin
    data <= 0;
  end else begin
    data <= ram[addr];
  end
end
```

(b) PyRTL (generated).

```
data:i16 = srom(addr) @bram;
```

(c) Reticle.

Figure 4.2: Example of three semantically equivalent read-only memory (ROM) implementations. The values of `srom` instructions in Reticle are provided in a separate configuration file.

4.2.1 Read-only Memories

Read-only memories can be described in multiple ways, using *behavioral* Verilog. For example, Figure 4.2 shows three different implementations that are semantically equivalent, two using behavioral Verilog and another one based on Reticle. Every memory can store up to two elements of 16-bits. The *baseline* reference implementation described in Figure 4.2a was taken from the *HDL coding techniques* found in the Xilinx Vivado user guide [Xil21b]. The other Verilog implementation shown in Figure 4.2b was generated from an embedded hardware DSL called PyRTL [DTS20]. In contrast to Verilog implementations, there is only one mechanism to describe these memories in the Reticle language, and it is based on the `srom` instruction. This instruction is described in Figure 4.2c.

While Verilog allows programmers to express different memory behavior, using storage elements such as registers, the Vivado *synthesizer* does not guarantee predictable results in

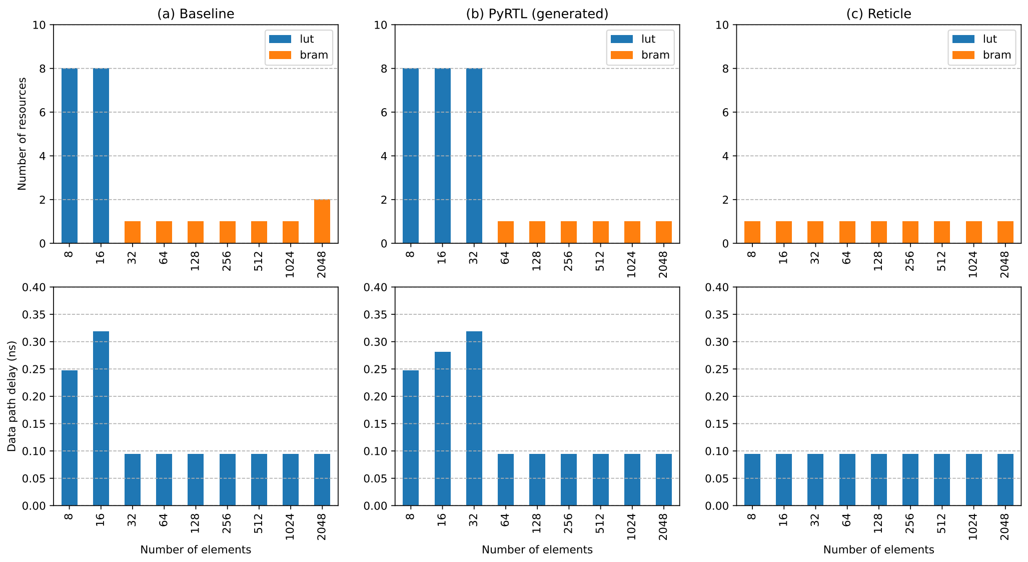


Figure 4.3: Resource utilization and data-path delay (performance) for three memory implementations described in Figure 4.2 for multiple number of elements.

terms of resources and performance. For example, Figure 4.3 shows the resource utilization and data-path delay for memories of different sizes (number of elements). First, we can see how resources are mapped to either LUTs or BRAMs depending on the number of elements for the two Verilog implementations (a, b) compared to Reticle's (c), which always maps to the same primitive. Interestingly, this primitive mapping affects directly the performance of the final program. For example, LUTs can only store few bits compared to BRAMs, requiring more primitives to store the same amount of data. Therefore, LUT-based memories consume more routing resources, which translates into longer data-path delays or worse performance. Conversely, the memory instruction in Reticle results in the same performance across multiple sizes, because it always uses the same resource (BRAMs).

Name	LUTs (%)	LRAMs (%)	BRAMs (%)	Data Path Delay (ns)
PyRTL	7.29	4.49	0	36.46
PyRTL + Reticle	7.29	0	57.64	6.93

Figure 4.4: Evaluating Reticle read-only memory instruction (`srom`) for an AES encryption implementation using the Xilinx Kria board.

4.2.2 AES Encryption

Now that we understand how different memory descriptions can change the resource utilization and performance metrics, we can discuss how Reticle's memory instruction `srom` can improve a more involved program such as AES encryption. Concretely, we evaluated two implementations based on PyRTL [DTS20]. The only difference between the two is that one implementation uses Reticle's memory instruction `srom`, while the other does not. This implementation uses read-only memories to implement the substitution box (SBOX) and multiplication tables for Galois fields. Finally, we used the Xilinx Kria board for evaluating these two implementations.

Table 4.4 shows the percentage of used resources (LUTs, LRAMs, and BRAMs) for the two implementations. Furthermore, the percentage of LUTs represents mapped compute operations, whereas LRAMs shows how many LUTs were used for memory operations. We can see how the use of Reticle's `srom` instruction results in the utilization of 36.46% (BRAMs) available in the device compared to the Verilog generated by PyRTL, which uses only 4.49% (LRAMs). The benefit of choosing BRAMs over LUTs represents a 5× speedup in the whole program for a cost of 36.46% (BRAMs) available in the device.

Chapter 5

Related Work

In this dissertation, we focus on the tasks of designing programming abstractions and evaluating efficient compilation techniques for modern FPGAs. We explore the architectural features, based on compute and memory operations, present in FPGAs today and show how these operations can be expressed more efficiently in an intermediate language called Reticle. Additionally, we demonstrate how to compile programs written in this language into FPGA primitives, including their physical locations.

Interestingly, there has been a large amount of work done in the space of languages and compilers for FPGAs. Programs written for these reconfigurable devices have been traditionally described using hardware description languages, and compiled into FPGA primitives by a process called hardware synthesis. First, we discuss work on programming languages for the task of hardware programming. Then, we talk about prior work on compiling hardware descriptions, including layout, for FPGAs.

5.1 Programming Languages for FPGAs

In the early days, the main purpose for FPGAs was emulating arbitrary hardware circuits. Hardware designers used FPGAs to prototype their architectures i.e., processors and evaluate software programs running on top it, due to the fact that simulating complex hardware designs i.e., CPUs took considerable amount of computing time [TQC+15]. Therefore, the languages used for programming FPGAs have been always related to the ones used for designing a silicon device (ASIC), regardless of all the architectural advancements in FPGAs over the last three decades [Tri15, Xil20c]. The following discussion is based on work done in the area of programming languages for FPGAs.

5.1.1 Hardware Description Languages

The Verilog language was designed back in 1984, and to this date, it is the most widely used language for programming hardware [FMG+20]. Originally, Verilog was a proprietary HDL that led to the creation of VHDL, a royalty-free hardware language that became an IEEE standard in 1987 [FMG+20]. The Verilog language, however, eventually dominated the market and became the standard source language for describing hardware that every FPGA toolchain uses today. Therefore, other languages designed over the last couple of decades had to eventually be compiled down to Verilog in order to compile a binary that can program the device. Interestingly, the design of the Verilog language was highly inspired by C [FMG+20] but adapted to capture circuits semantics. Specifically, the programming abstraction is based on data operations flowing through registers known as the register-transfer level (RTL) abstraction. Furthermore, this abstraction requires programmers to define the behavior of a given hardware circuit for every clock cycle.

A range of recent domain specific languages, based on the RTL abstraction, have emerged aiming to improve productivity compared to Verilog. For example, Chisel [BVR+12], PyRTL [DTS20], and PyMTL [JPOB20] were designed to improve the hardware programming experience by leveraging features available in the host language such as collections, decorators, and type systems. On the other hand, there is work on automating the scheduling complexity required by RTL-based languages, called Bluespec [Nik04]. This language provides a simpler concurrency model, based on rules that are scheduled by a compiler instead of a programmer. The design goal of the Bluespec language is to allow formal reasoning without compromising performance of the resulting hardware programs [BPCCA20].

All of these languages, however, target the semantics of the Verilog language, which were designed to express behavioral descriptions of low-level circuits. This approach works great when designing a silicon chip, but it falls short for efficiently programming specialized hardware primitives in a device, which it is the case for FPGAs. For example, there is no mechanism available to express vector operations *behaviorally* in the Verilog language, therefore, programs written in Verilog cannot use vector units (SIMD) available in state-of-the-art FPGAs. In this work, I focus on designing programming abstractions, specifically for FPGAs, that can be used to address the expressiveness challenges present in traditional HDLs.

5.1.2 Layout Languages

Inspired by functional geometry [Hen82], muFP was the first hardware language that proposed a different approach to conventional hardware description languages. The language was based on functional programming and let programmers describe not only behavior but also the layout of hardware programs. This idea of combining behavior and layout was later explored for FPGAs in a language called Lava [BCSS98]. The Lava language employs first order functions to describe basic operations, based on LUT primitives, that are then composed using high-order functions to efficiently express complex hardware programs. The approach demonstrated to be particularly useful for achieving great performance on highly structured designs, compared to traditional synthesis and layout approaches [Sin11].

In this work, however, my approach is based on splitting the task of expressing behavior and layout in two separate languages instead of using a single one. The reason behind is that separating the program behavior from layout allows program to be expressed independently of the target, since layout requires information about the supported operations and physical locations of resources. More importantly, compilation can be efficiently performed in finer grained steps, solving one concern at the time, instead of a single monolithic transformation task.

5.1.3 High-level Languages

A range of recent languages have aimed to improve the programmability of hardware accelerators by using a higher-level abstractions compared to RTL languages. One work in this area is called Spatial [KFP+18] which proposes the use of parallel patterns, including control, memory hierarchy, and host interfaces, to program FPGAs and CGRAs. Another area of work focused on creating novel type systems for expressing hardware-level concerns such as scheduling [DFH+20] and interfaces [dMV19]. Moreover, Dahlia [NAT+20] demonstrated how to reduce the performance unpredictability of high-level programs by modeling consumable hardware using a *time-sensitive affine type system.* This category of work is centered on productive high-level programming, whereas the programming abstractions proposed in this work are designed for compilers targeting FPGA architectures instead of programmers.

5.1.4 Intermediate Representations

The number of intermediate representations focusing on hardware programming have increased over the last couple of years, due to the ubiquity of hardware accelerators. Interestingly, there are several work on IRs operating at different level of abstraction and addressing the challenge of transforming a program specification into a hardware description. A novel approach in this area is Calyx [NTLS21] which is an intermediate language for accelerator-designs [Adr21] that is based on two languages: one for describing *control* and another for *structure*. This representation enables certain machine-driven optimizations and analysis such as resource sharing and register liveness that are critical for efficiently lowering high-level program specifications into hardware. Additionally, other lower-level IRs have also been proposed such as FIRRTL [IKL+17] and LLHD [SKGB20]. The FIRRTL language was created as a compiler target for the Chisel language. The language aims to provide a representation for performing optimizations and analysis such as constant folding and resource estimation, before producing a Verilog implementation. The LLHD IR, on the other hand, was designed to model arbitrary hardware behavior, including *timing*, that matched most of the features available in the Verilog standard aiming verification and synthesis.

Similarly to related work on hardware description languages, most of these IRs are designed around the semantics of the Verilog language, dismissing the technological advances in modern FPGA architectures [Ivo06]. Conversely, the programming abstractions described in this work aim to solely cover operations supported by FPGAs, rather than focusing on supporting every possible hardware operation required for designing arbitrary silicon chips.

5.2 Compilation Techniques for FPGAs

FPGA programs are written (or compiled to) the Verilog language, because Verilog is the common interchange format for FPGA toolchains. These tools perform the arduous task of transforming arbitrary hardware descriptions into primitives that can fit into an FPGA target. The abstraction gap between Verilog descriptions and FPGA hardware is enormous [Adr19], requiring a multi-stage compilation process [VMS+21]. In the following discussion, I cover related work on compilation techniques used throughout this process.

5.2.1 Hardware Synthesis

The process of translating a behavioral specification written in a hardware description language i.e., Verilog into a netlist of FPGA primitives is known as hardware synthesis [LSB05]. First, the hardware description is transformed into a network of boolean operations, or gates, that is then optimized using logical equivalences to minimize the number of operations—and therefore resources [Mic94]. Next, these boolean operations are mapped to FPGA primitives by a step called *technology mapping* [MCB07]. There are several ideas explored in the area of hardware synthesis for FPGAs over the last three decades [Tri15], however, my work focused on leveraging software compilation techniques i.e., instruction selection instead of logic optimizations and unpredictable metaheuristics used in technology mapping [LSB05].

5.2.2 Instruction Selection

While it is not mainstream, other compilers have used instruction selection and similar techniques to optimize hardware programs [Joh83]. For example, a compiler based on *silicon instruction sets* [KW88] powered the design of six chips at IBM, and a similar approach has been applied to FPGAs [CCDW98]. However, this line of work primarily targets compound logic functions used in silicon chips i.e., AND-OR-INVERT, and even the FPGA-based variants focus solely on programming LUTs. Previous work does not attempt to program recent specialized units i.e., DSPs, because FPGAs at that time were mainly based on LUTs. In this work, I show how to use instruction selection to optimize programs not only for DSPs and LUTs but also for memory primitives including registers and SRAMs.

5.2.3 Layout Compilation

Historically, the task of compiling a program layout for an FPGA has always been influenced by hardware synthesis. For example, a layout engine operates over low-level primitives instead of high-level operations, because this is the standard output of a conventional synthesizer [CC17]. The layout process consists of two intertwined steps: one called *placement* that finds a legal location for every primitive in the program and another one called *routing* that searches for the best possible path (wiring) between primitives [CC17]. Although there are some efforts on open-sourcing *routing* for some FPGA architectures [MEB+20], the access to this feature is closed-source and not widely supported by toolchains [Xil21a]. Therefore, the layout compilation technique proposed in this work focuses only on *placement*. Specifically, I focus on evaluating the benefits of laying out programs using a more coarse-grained abstraction based on the Reticle's assembly language.

Chapter 6

Conclusion

The work described in this thesis presents an alternative approach to program state-of-the-art FPGAs. The programming abstractions and compilation techniques covered in this work aim to provide a better foundation for FPGA programming.

In this thesis, I present a new low-level abstraction for FPGA programming, called Reticle, that captures high-performance operations i.e., SIMD available in recent architectures and provide a mechanism to bind operations to primitives if needed. Additionally, I show how this intermediate abstraction is lowered to a target-dependent representation, also known as *assembly language*, using a standard and faster compilation technique called *instruction selection*. Furthermore, Reticle's assembly language can be used to spatially accommodate operations into the FPGA fabric. Therefore, this work shows a viable alternative to traditional FPGA synthesis and placement with deterministic results and compile times in seconds instead of days.

6.1 Future work

There are several areas of exciting future work to be done in the space of Reticle-like compilation for FPGAs. One area that I am particularly interested in is related to generating *target descriptions* from architectural descriptions. This is particularly important because newer FPGA architectures can be supported without any human intervention. Today, the *instruction selector* used in Reticle is completely target independent, requiring only a target description for every FPGA backend. Unfortunately, these descriptions are implemented by hand and require an expert on the target architecture for generating optimal implementations.

One notable difference between Reticle and traditional hardware synthesis is that Reticle does not decompose arithmetic operations into boolean logic to search for a global optimum. Instead, Reticle aims to find or select the best possible instruction implementations, based on a *target description* that was derived offline or before the actual compilation happen. This means that search-based approaches based on *program synthesis* or *learning techniques* can be used to discover newer and more efficient implementations compared to the ones we have today.

Another interesting research opportunity that Reticle enables, and can be framed as a compilation problem, is related to the layout of FPGA primitives. Normally, the placement of primitives is done after the structure of the source program is completely destroyed. This means that current heuristics place primitives instead of operations, making it really hard to find a feasible solution fast on large devices. Reticle's assembly language offers an alternative to this approach by providing a language with layout information that allows operations to be placed instead of primitives, reducing considerably the search-space and making possible faster placement. Future work can explore further the idea of placing operations instead of primitives and evaluate performance and compilation time tradeoffs for a set of workloads and FPGA targets.

Bibliography

[Ach19] Achronix. Speedster7t IP Component Library User Guide. `https://www.achronix.com/sites/default/files/docs/Speedster7t_IP_Component_Library_User_Guide_UG086.pdf`, 2019.

[Adr19] Adrian Sampson. FPGAs Have the Wrong Abstraction. `https://www.cs.cornell.edu/~asampson/blog/fpgaabstraction.html`, 2019.

[Adr21] Adrian Sampson. From Hardware Description Languages to Accelerator Design Languages. `https://www.sigarch.org/hdl-to-adl/`, 2021.

[AG85] Alfred V. Aho and Mahadevan Ganapathi. 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*, POPL '85, page 334–340, New York, NY, USA, 1985. Association for Computing Machinery.

[AI19] Mikhail Asiatici and Paolo Ienne. Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 310–319, New York, NY, USA, 2019. Association for Computing Machinery.

[BC13] Thomas Braibant and Adam Chlipala. Formal verification of hardware synthesis. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 213–228, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, page 174–184, New York, NY, USA, 1998. Association for Computing Machinery.

[BM10] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BPCCA20] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 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*, PLDI 2020, page 243–257, New York, NY, USA, 2020. Association for Computing Machinery.

[BVR+12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[CC17] Shih-Chun Chen and Yao-Wen Chang. Fpga placement and routing. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 914–921, 2017.

[CCDW98] Timothy J. Callahan, Philip Chong, André DeHon, and John Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, FPGA '98, page 123–132, New York, NY, USA, 1998. Association for Computing Machinery.

[CNM+18] Philip Colangelo, Nasibeh Nasiri, Asit Mishra, Eriko Nurvitadhi, Martin Margala, and Kevin Nealis. Exploration of low numeric precision deep learning inference using intel fpgas, 2018.

[DFH+20] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 408–422, New York, NY, USA, 2020. Association for Computing Machinery.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. 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*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[dMV19] Jan de Muijnck-Hughes and Wim Vanderbauwhede. A typing discipline for hardware interfaces. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*, pages 6:1–6:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[DTS20] D. Dangwal, G. Tzimpragos, and T. Sherwood. Agile hardware development and instrumentation with pyrtl. *IEEE Micro*, 40(4):76–84, 2020.

[FMG$^+$20] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. Verilog hdl and its ancestors and descendants. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[FOP$^+$18a] 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. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[FOP$^+$18b] 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. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[Hen82] Peter Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP ’82, page 179–187, New York, NY, USA, 1982. Association for Computing Machinery.

[IKL$^+$17] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD ’17, page 209–216. IEEE Press, 2017.

[Int20a] Intel. 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`, 2020.

[Int20b] Intel. Intel HLS Compiler: Fast Design, Coding, and Hardware. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01274-intel-hls-compiler-fast-design-coding-and-hardware.pdf`, 2020.

[Int20c] Intel. Intel Stratix 10 Variable PrecisionDSP Blocks User Guide. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-dsp.pdf`, 2020.

[Ivo06] Ivo Bolsens. Programming Modern FPGAs. `http://xilinx.eetrend.com/files-eetrend-xilinx/forum/201703/11148-29110-bolsens.pdf`, 2006.

[Joh83] S. C. Johnson. Code generation for silicon. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 14–19, New York, NY, USA, 1983. Association for Computing Machinery.

[JPOB20] S. Jiang, P. Pan, Y. Ou, and C. Batten. Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, 40(4):58–66, 2020.

[KFP+18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. *SIGPLAN Not.*, 53(4):296–311, June 2018.

[Kun82] H. T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, January 1982.

[KW88] K. Keutzer and W. Wolf. Anatomy of a hardware compiler. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 95–104, New York, NY, USA, 1988. Association for Computing Machinery.

[Lat20] Lattice. sysDSP Usage Guide for Nexus Platform. `https://www.latticesemi.com/view_document?document_id=52791`, 2020.

[LK18] C. Lavin and A. Kaviani. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140, 2018.

[LSB05] A. Ling, D.P. Singh, and S.D. Brown. Fpga technology mapping: a study of optimality. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 427–432, 2005.

[MCB07] A. Mishchenko, S. Chatterjee, and R. K. Brayton. Improvements to technology mapping for lut-based fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):240–253, 2007.

[MEB+20] 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. *IEEE Micro*, 40(4):49–57, 2020.

[Mic94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Higher Education, 1st edition, 1994.

[NAT+20] 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*, PLDI 2020, page 393–407, New York, NY, USA, 2020. Association for Computing Machinery.

[Nik04] R. Nikhil. 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.*, pages 69–70, 2004.

[NTLS21] 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*, ASPLOS 2021, page 804–817, New York, NY, USA, 2021. Association for Computing Machinery.

[PCC$^+$14] 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. A reconfigurable fabric for accelerating large-scale datacenter services. *SIGARCH Comput. Archit. News*, 42(3):13–24, June 2014.

[RBL$^+$09] Mariusz Rawski, Grzegorz Borowik, Tadeusz Luba, Pawel Tomaszewicz, and Bogdan J. Falkowski. Logic synthesis strategy for fpgas with embedded memory blocks. In *2009 MIXDES-16th International Conference Mixed Design of Integrated Circuits Systems*, pages 296–301, 2009.

[RF16] B. Ronak and S. A. Fahmy. Mapping for maximum performance on fpga dsp blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(4):573–585, 2016.

[RT06] R. B. Reese and M. A. Thornton. *Introduction to Logic Synthesis using Verilog HDL.* 2006.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, page 288–298, New York, NY, USA, 1992. Association for Computing Machinery.

[SGKK19] A. Samajdar, T. Garg, T. Krishna, and N. Kapre. Scaling the cascades: Interconnect-aware fpga implementation of machine learning problems. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 342–349, 2019.

[Sin11] Satnam Singh. The rloc is dead - long live the rloc. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 185–188, New York, NY, USA, 2011. Association for Computing Machinery.

[SKGB20] 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*, PLDI 2020, page 258–271, New York, NY, USA, 2020. Association for Computing Machinery.

[TH19] Lenny Truong and Pat Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:21, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[TQC+15] Zhangxi Tan, Zhenghao Qian, Xi Chen, Krste Asanovic, and David Patterson. Diablo: A warehouse-scale computer network simulator using fpgas. *SIGARCH Comput. Archit. News*, 43(1):207–221, March 2015.

[Tri15] Stephen M. Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, 103(3):318–331, 2015.

[Tur03] Mike Turpin. The Dangers of Living with an X (bugs hidden in your Verilog). `https://developer.arm.com/documentation/arp0009/a/`, 2003.

[VMS+21] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. Reticle: A virtual machine for programming modern fpgas. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[Xil20a] Xilinx. UltraScale Architecture DSP Slice User Guide. `https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf`, 2020.

[Xil20b] Xilinx. Versal ACAP Memory Resources. `https://www.xilinx.com/support/documentation/architecture-manuals/am007-versal-memory.pdf`, 2020.

[Xil20c] Xilinx. Versal:The First Adaptive Compute Acceleration Platform (ACAP). `https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf`, 2020.

[Xil20d] Xilinx. Vitis Unified Software Development Platform 2020.1 Documentation. `https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/introductionvitishls.html`, 2020.

[Xil21a] Xilinx. Vivado Design Suite User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug904-vivado-implementation.pdf, 2021.

[Xil21b] Xilinx. Vivado Design Suite User Guide Synthesis. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug901-vivado-synthesis.pdf, 2021.