CMSC838L - Final report

Arjun Vedantham
Yusuf Bham

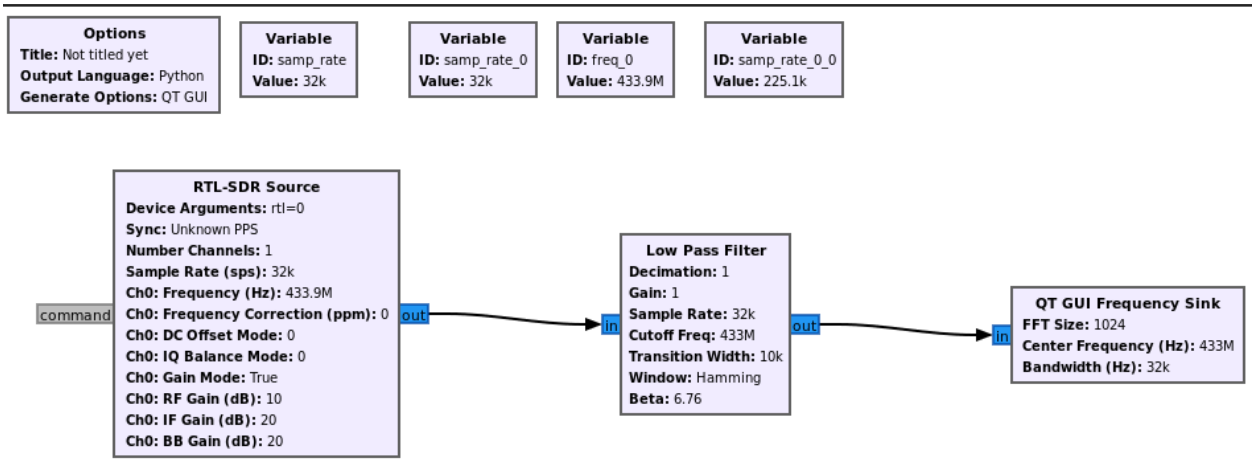
May 2024

1 Introduction and Motivation

Digital signal processing techniques are extremely important in telecommunications, computer vision, and a number of other related fields. In particular, digital signal processing techniques form a key component behind the idea of software defined radio (SDR), which refers to analyzing digital samples that represent radio signals with software, instead of discrete hardware components that operate over signals in analog formats. Software defined radio presents a notable improvement in flexibility for radio engineers, and removes the need for specialized hardware components - instead, new signal demodulation techniques or formats can be deployed just through a simple software update.

SDR users typically define signal processing pipelines using platforms like GNURadio. GNURadio presents a graphical format for creating these pipelines, with discrete blocks representing a signal source, signal sink, or an intermediate step in the processing pipeline. As an example, see the "flowgraph" (as called in GNURadio's documentation) below. This starts by instantiating a signal source from an RTL-SDR (a type of hobbyist SDR that can be used over USB with consumer PCs), and sets both a sampling rate (32000 Hz) and a listening frequency (signals at 433 MHz). From here, these samples are fed into a low pass filter block (which smooths out high frequency noise in the signal), and the resulting signal is transformed into a stream that is passed to a GUI block that graphs the signal, using a Fast Fourier Transform (FFT) to move the signal into the frequency domain.

This simple flowgraph is stored in an XML format, and is used by GNURadio's backend to generate a Python script that actually runs the defined processing pipeline. In addition, there are also C/C++ modules loaded into the runtime system for low level tasks - for instance, a USB driver for the SDR.



1.1 Problems Identified with State of the Art

There are number of problems with this current approach. First, GNURadio's practice of emitting Python scripts means that parallelism is limited on two fronts - first, because of language design choices (Python's infamous "global interpreter lock", which essentially forbids runtime concurrency), and also because we are ultimately running this script on standard PC hardware, which uses an inherently sequential von Neumann architecture. Additionally, as previously mentioned, there is also an extensive library of C/C++ modules, and the implementation of these modules are opaque to the Python-level code generated by GNURadio's flowgraph compiler.

As such, we identified two research questions that we aimed to answer in this project:

- Could we use hardware acceleration to get better performance and greater parallelism for DSP applications?
- Could we design a language that is more conducive to defining correct DSP pipelines?

2 Literature Review

We started by conducting a literature review of existing languages designed for DSP problems.

2.1 Ziria

One of the first papers we considered as Ziria [8], a domain specific language that was designed to aid development in implementations of the physical layer of wireless protocols. Ziria presented a functional language design syntax, and was specifically intended for wireless protocol implementations on IOT hardware. As such, it contained many primitives that we thought would be important to add to our language - for instance, an "FFT" primitive function. However Ziria had a key limitation - it was designed for standard CPUs, which meant that there were still parallelism limitations.

2.2 Calyx

One technique generally used for hardware acceleration for specialized applications like this is to deploy them to FPGAs. One example of this was the Catapult paper that we read in class, where Microsoft deployed FPGAs to accelerate running PageRank as part of their Bing server infrastructure. However, FPGAs are notoriously difficult to program, and generally require extremely fine-grained circuit configurations written in a hardware description language like Verilog.

Calyx [7] is an intermediate representation for compilers developed by the CAPRA research group at Cornell. It defines circuits in three distinct parts - a collection of memories (consisting of combinational memories/ flip-flops and registers), wire groups (which denote assignments between different memory components in the circuit), and a statically defined control schedule that orders wire assignments. Calyx has already been used in the Filament HDL, another domain specific language project from the CAPRA group that incorporated signal timing into the language's type system.

3 Technical Contribution

We decided to use the Calyx IR and design a more general, hardware acceleratable language for DSP tasks, called Zinnia. [1] We also considered emitting circuits using CIRCT, the LLVM framework that allows compilers to generate MLIR that is subsequently lowered to a Verilog hardware description, however we quickly found that CIRCT lacked the project maturity needed to develop even a basic language around it. This included no binaries to link against, thus requiring us to compile a large subset of the LLVM project, which was not practical. Calyx also had better documentation for its IR, and a small standard library of memory cells that could be easily integrated into circuits synthesized when the compiler lowers from the IR to a Verilog hardware description.

3.1 Parsing and Typing

Zinnia's parsing is a fairly standard approach. We used the Rust parser-combinator library `chumsky` for both the tokenization and subsequent parsing steps.

Zinnia is backed by a bidirectional type system, heavily inspired by Complete and Easy Bidirectional Typing for Higher-Rank Polymorphism and Purescript [2]. Also similarly to Purescript, while the system is based on CAEBDT, our system doesn't use the "main" takeaway of the paper – an ordered context, and stays with a more simple one. This was largely due to familiarity, and in the future we would like to rewrite it to use ordered contexts. The choice of a bidirectional type system stems from its ease of extensibility compared to HM, which is the more common choice for a language like this. This extensibility allows us to more easily add features like linear typing in the future, which is important in this space as it means we can keep the *pure* and *functional* aspect, while still preserving performance and memory constraints. Currently types are restricted to functions and primitives, just

[1] Named "Zinnia" because "calyx" refers to the petals of a flower and zinnias are a type of flower. Plus it sounds like Ziria.

due to time constraints. General universal quantification is supported, but access to it is currently limited to the `scan` primitive. While generics over types are generally supported, being generic over a *value* is currently restricted to the vector primitive.

Full semantics and syntax for our language can be found at the end of our report.

3.1.1 Ease of Use

One key consideration when designing the typechecker and parser was support for rich error reports. The parser has support for error backtracking, allowing for better reports on syntax. Similarly, the typechecker preserves as much location info as possible in order to give full-featured errors.

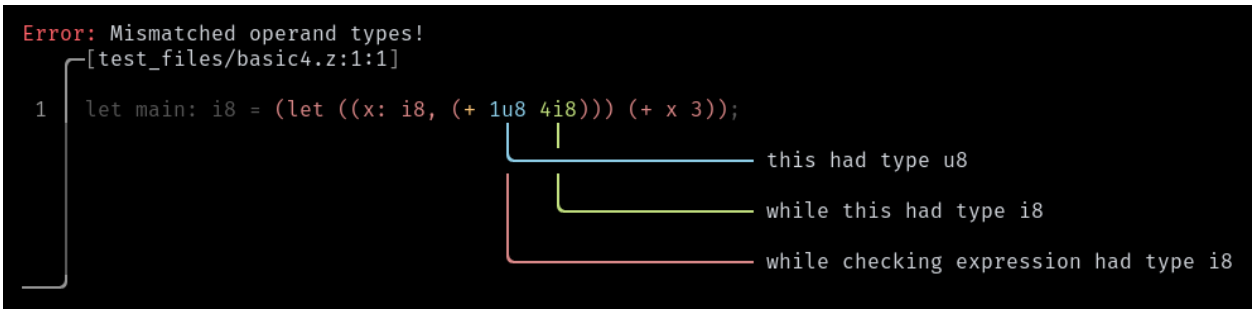


Figure 1: An error report for mismatched operands

3.2 Code Generation

The compiler that we wrote for Zinnia traverses the AST recursively. Since there is no concept of dynamic memory allocation for circuits, we focused on generating the required memories for the circuit first.

At the top level, "let" bindings are used to generate register memories (for integers) and combinational memories (for vectors). For constant values, we use Calyx's "structure" macro to instantiate a "constant" register with the requried value at compile time, which is subsequently removed in the process of optimization and lowering to Verilog. Variable bindings are stored in a hashmap and are accessed using either the component name (which is guaranteed to be unique through the Calyx frontend API) or the user-defined variable name.

For binary operations or if expressions, we need to use comparator primitives and wire them correctly - to do this, we call the memory generation function on each of the parameters of the operation, and then use a "wire" generation function which generates assignments between the inputs and outputs of the memories and logical primitives. After this is complete, we generate a small schedule of operations depending on the subexpression that we are compiling, using the Calyx "seq" block to sequentially order operations (e.g. ensuring that a register memory gets its constant value before the addition actually takes place). Finally, the top most register memory's value is copied into a specially instantiated combinational memory. This is because register memories in Calyx do not have latching/persistence guar-

antees, however, values in combinational/vector memories can be examined as part of the circuit simulation.

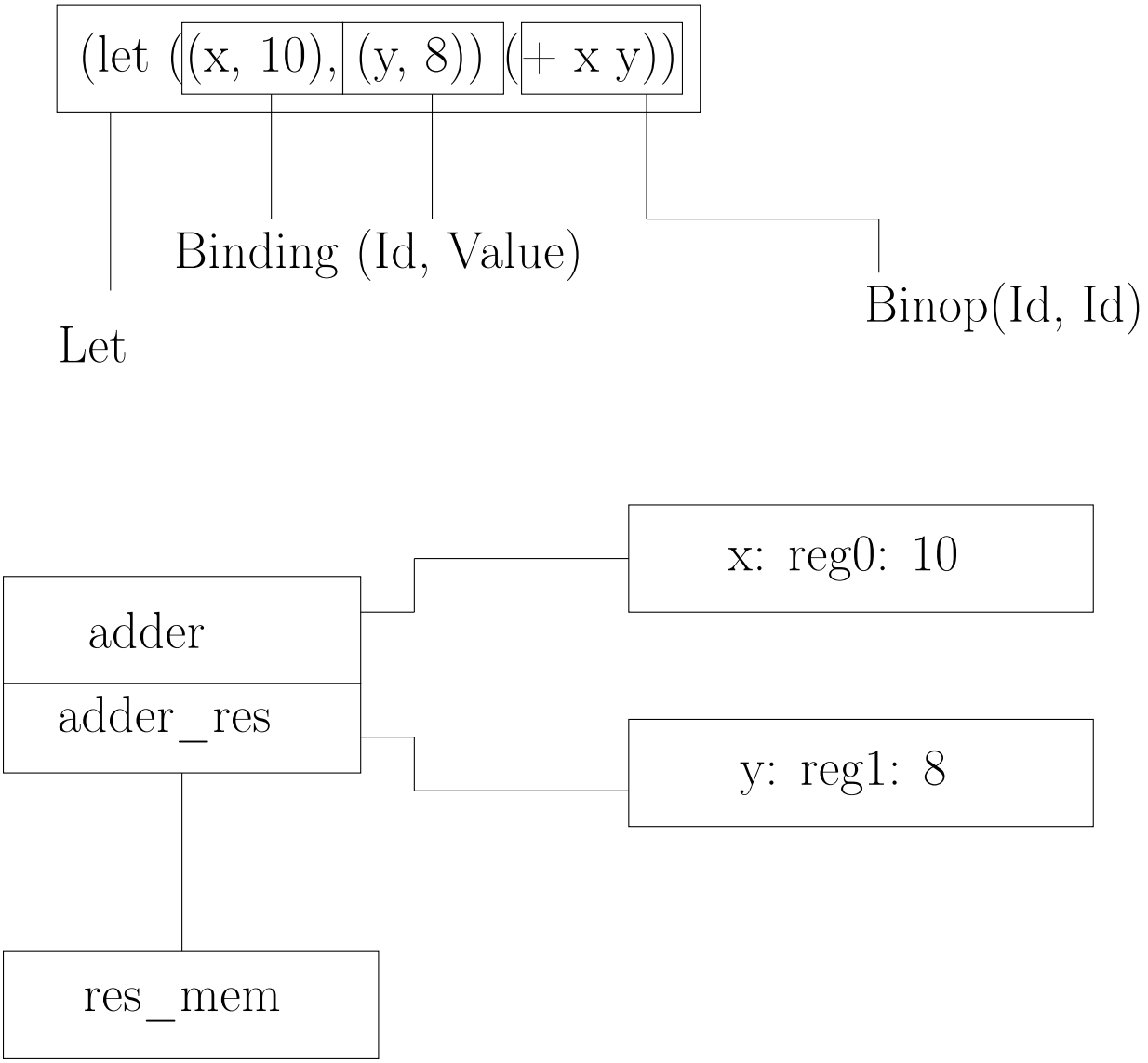


Figure 2: Simplified compilation/memory instantiation flow for a Zinnia program.

3.3 Supporting Parallelization in Compilation

One of the main goals of our DSL was to improve parallelism. While Calyx has a "par" primitive for parallelizing parts of the circuit, Zinnia currently only generates circuits using the "seq" ("sequential") control flow keyword for easier debugging and testing. Even though parallelism at the user level is limited, we still hoped to expose some of the potential parallel speedups through the use of a function primitive that could be used to get implicit parallelism.

3.3.1 Prefix Sums Scan

To expose some parallelism without explicitly supporting compiling using the "par" control flow block, we implemented a version of parallel scan using the prefix sums algorithm [1]. This primitive takes a reference to a vector with 8 integers and performs the addition operation, returning a modified array where the latter half of the array contains the scan result.

This function was supposed to be invoked as a primitive within the language, with the compiler inlining a separate Calyx component whose input memories would be populated with a reference to the vector supplied by the caller. However, while we were able to test the scan component as a separate circuit generated from our hand-written IR code, we were not able to fully resolve linking issues that would allow us to pass memories by reference into subcomponents.

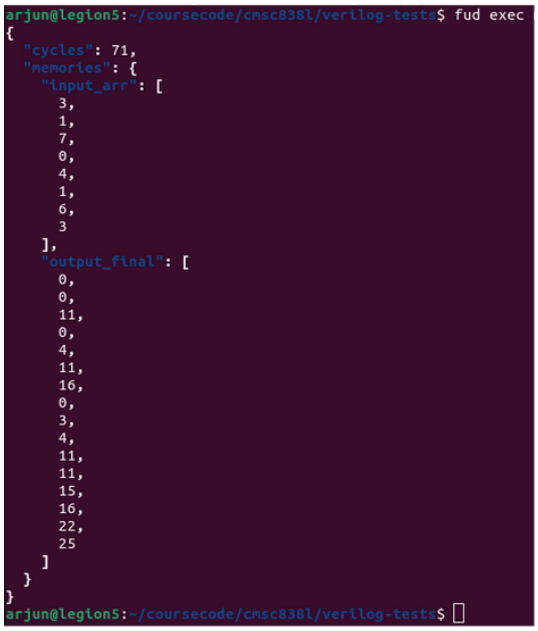


Figure 3: Screenshot of the final state of the prefix sums array - a separate vector memory is instantiated in the IR implementation to track the intermediate values as they flow up and down the prefix-sums tree.

4 Evaluation

We decided to evaluate our language in three different ways: correctness testing, scalability testing, and hardware testing.

4.1 Correctness Testing

To verify the correctness of the language, we attmepted to use the Cocotb RTL testing framework and the Icarus Verilog simulator. This allows us to run a simulated version of the circuit's clock and write unit tests around elements of the circuits that we generate. Although we did not have time to implement extensive tests, we were able to informally verify correct

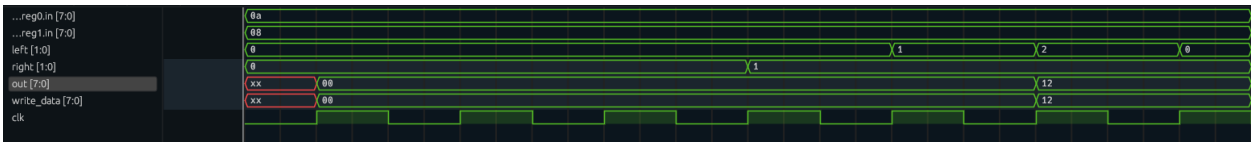


Figure 4: Example timing diagram for the program shown in Figure 1. The operands (represented in hex form here) are loaded into reg0 and reg1, before being loaded into the "left" and "right" inputs of the combinational adder. The adder register's result (5th line down) is then moved into the "output" combinational memory using its "write_data" line. Finally, the last line displays the clock signal for the whole circuit.

execution using unit tests for the binary operations supported by the language, as well as correct outputs values for the scan primitive (which was tested separately since we could not link this into the language easily) through Icarus Verilog. We were also able to generate timing diagrams that we could view with GTKWave, and this showed that values used during the circuit's evaluation were being loaded into registers correctly. However, we would have wanted to use randomized testing with cocotb to be more confident in our compiler's correctness.

4.2 Scalability Testing

Second, we wanted to test the scalability of our language. For this, we focused on evaluating the "scan" primitive, since it is the main way end-users would be able to obtain parallelism through the language (at least, as it is currently implemented). We found that in simulation, our parallel scan implementation always took 71 cycles to generate outputs, given a set of 8 integers to iterate over. While parallel scan theoretically provides work bounds of $O(n)$ and span bounds of $O(\lg(n))$, in practice the work and span bounds of our implementation are worse. This is because we were not sure if it was possible to instantiate an arbitrarily high number of registers, so we focused on using the combinational memory primitives to hold intermediate values (particularly between the "sweep-up" and "sweep-down" phases of parallel scan). Combinational memory primitives in the Calyx IR are exclusive read/write memory cells, and take at least one cycle to set the data address and read the stored value out of the memory into a register. This restriction essentially limited the parallelism that we could achieve to only a few steps of the algorithm - specifically, summing values between nodes at each level of the prefix sums tree, since the additions were purely combinational and depended only on values in registers that were ready to go at the last cycle.

4.3 Physical Testing

We hoped to eventually deploy circuits that were synthesized using Zinnia to a Lattice Icestick FPGA. This FPGA can be used with consumer devices over a USB connection, and has a large library of open source tooling. This includes the Yosys synthesis toolkit (which handles transforming high level Verilog hardware descriptions into lower level RTL circuits), the nextpnr place and route tool (which handles connections between logic units on the FPGA), and iceprog, an open source programming tool that could deploy circuits

with complete routing to the flash memory on-board the Icestick.

Circuits synthesized by Calyx include three signals by default - a "go" line, which when activated, activates the circuit, a "clk" (clock) line, which handles cycle-level signalling for the sequential logic elements of the circuit, and a "reset" line, which can be used to reset the circuit while active.

In order to actually use the synthesized circuits, the "go" line must be pulled to logical high, and while the reset lines needed to be set to logical low, and the clock line must be tied to the 12 MHz oscillator built into the Icestick. Values for the "scan" primitive could then be loaded in from the Icestick's block RAM module.

To support dynamically loading values into block RAM, we wrote a small serial communication driver for the Icestick. Specifically, this was a form of one way (half-duplex) SPI, with an Arduino serving as the primary device on the SPI bus, and the Icestick serving as the secondary device. We chose SPI because it is relatively easy to implement on the receiving side (in fact, a shift register is enough to receive data transmitted over the data line). While testing the SPI driver, we wanted to use an external clock source to control the speed of data transmission, and ensure that the Icestick was sampling the data line in phase with the transmitting device's clock (in this case an Arduino). Unfortunately, setting the clock line for the circuit loaded onto the FPGA also sets the clock line for the USB programmer chip, which requires a 12MHz signal in order to function correctly. This misconfiguration prevented us from loading new circuits onto the FPGA, and required us to use simulation as our primary evaluation method instead.

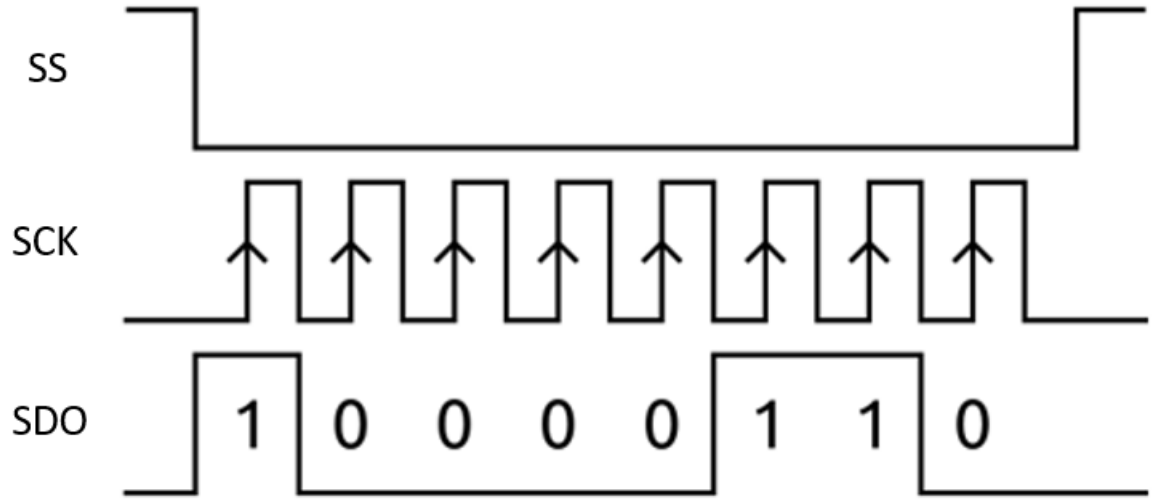


Figure 5: Half duplex SPI waveform, where an Arduino would have controlled the SCK and SDO lines and the FPGA would bit shift the received values in block RAM.
[6]

5 Conclusion

In the end, we were able to implement a version of the scan primitive in the Calyx IR, implement a bidirectional typing system, add some basic language features into our language,

and use external circuit verification and simulation tools to show that our circuits performed as the high level program specified.

However, there is still room for improvement, both in terms of feature support and performance - we do not have support for looping structures, even though the Calyx IR supports it. Additionally, our parallelism benefits are limited by the fact that we can only really exploit the parallelism present in the hardware through the scan primitive, even though Calyx supports parallelized control schedules. While part of this is due to limitations in the Calyx standard library, future libraries could be used to enable concurrent read/exclusive write accesses to these memories instead of the current exclusive read/write access pattern.

We were also not able to show our circuits running on real hardware, which was the ultimate goal of the project. Our current version of Zinnia would not be usable with real software defined radio/signal processing workloads since our hardware interfaces are not finished yet. However, we feel that there is enough expressivity in the language (e.g. customizable bit widths, a clean and functional syntax) that could eventually make it practical for creating correct signal processing pipelines.

One obstacle that we did not foresee was the difficulty of working with the Calyx IR from Rust. While LLVM's CIRCT project has a robust Rust API, it seems that the authors of the Calyx IR primarily want language authors to use their IR from a Python builder module or the CIRCT toolchain, even though the Filament HDL (which was authored by the same research group and uses Calyx as its IR) is also written in Rust. For some issues, like linking against the combinational memory primitives or inlining other components, the Rust library documentation was so poor that we ended up looking through Filament's implementation and using that as a guide for building our compiler.

Altogether, this was a very interesting exploration into building a domain specific language that interacted so closely with hardware, and was a great way to get experience with compilers, parallel algorithms, type systems, and using FPGAs. We hope to continue working on Zinnia as a hobby research project over the summer.

The codebase for Zinnia is available on GitHub: `https://github.com/javathunderman/zinnia`

6 Syntax

⟨*decl*⟩ ::= `let` ⟨*ident*⟩`:` ⟨*ty*⟩ `=` ⟨*expr*⟩`;`

⟨*ident*⟩ ::= (⟨*letter*⟩ | _) {⟨*letter*⟩ | 0..9 | _}

⟨*ty*⟩ ::= `()`
| `bool`
| ⟨*num-ty*⟩
| `Vec<`⟨*num-ty*⟩`,` {0..9}$^{+}$`>`

⟨*num-ty*⟩ ::= (`u` | `i`) {0..9}$^{+}$

⟨*expr*⟩ ::= `()`
| ⟨*bool-lit*⟩
| ⟨*int-lit*⟩
| ⟨*vec-lit*⟩
| `(`{⟨*expr*⟩}$^+$`)`
| `(`⟨*expr*⟩ ⟨*bop*⟩ ⟨*expr*⟩`)`
| `(let (`⟨*binders*⟩`)` ⟨*expr*⟩`)`
| `(if` ⟨*expr*⟩ ⟨*expr*⟩ ⟨*expr*⟩`)`

⟨*binders*⟩ ::= ⟨*binder*⟩
| ⟨*binder*⟩`,` ⟨*binders*⟩

⟨*binder*⟩ ::= `(`⟨*ident*⟩`,` ⟨*expr*⟩`)`
| `(`⟨*ident*⟩`:` ⟨*ty*⟩`,` ⟨*expr*⟩`)`

⟨*num-cmp-op*⟩ ::= `>`
| `>=`
| `<`
| `<=`

⟨*num-op*⟩ ::= `+`
| `-`
| `*`
| `/`

⟨*cmp-op*⟩ ::= `==`
| `!=`

⟨*bop*⟩ ::= ⟨*num-op*⟩
| ⟨*num-cmp-op*⟩
| ⟨*cmp-op*⟩

⟨*int-lit*⟩ ::= [`-`]{0..9}$^+$
| [`-`]{0..9}$^+$ (`u` | `i`) {0..9}$^+$

⟨*bool-lit*⟩ ::= `true`
| `false`

⟨*vec-lit*⟩ ::= `[`⟨*vec-elems*⟩`]`

$\langle$*vec-elems*$\rangle$ ::= $\langle$*int-lit*$\rangle$
| $\langle$*int-lit*$\rangle$, $\langle$*vec-elems*$\rangle$

7 Typing

$$
\begin{array}{llcll}
\text{Types} & A, B, C & ::= & () & \mid \alpha \mid \widehat{\alpha} \\
 & & & & \mid \forall \alpha.\, A \\
 & & & & \mid A_1, \ldots, A_n \to B \\
\text{Monotypes} & \tau, \sigma, \pi & ::= & () & \mid \alpha \mid \widehat{\alpha} \\
 & & & & \mid \tau_1, \ldots, \tau_n \to \sigma \\
 & & & & \mid \vartheta \mid \vartheta_{i,s} \mid \widehat{\vartheta} \mid \widehat{\vartheta}_{i,s} \\
 & & & & \mid \mathsf{Vec}(\tau, c) \mid \widehat{\mathsf{Vec}} \\
 & & & & \mid \mathsf{Bool} \\
\text{Contexts} & \Gamma, \Delta, \Theta & ::= & \cdot & \mid \Gamma, \alpha \mid \Gamma, (x : A)
\end{array}
$$

Figure 6: Syntax of types, monotypes, and contexts.

$\boxed{\Gamma \vdash A}$ Under context Γ, type A is well-formed.

$\boxed{\Gamma \overset{n}{\underset{i=1}{\vdash}} A}$ Under context Γ, types $\{A_i\}_1^n$ are well-formed.

$$\frac{}{\Gamma \vdash ()}\ \mathsf{UnitWF} \qquad \frac{}{\Gamma \vdash \mathsf{Bool}}\ \mathsf{BoolWF} \qquad \frac{n > 0}{\Gamma \vdash \vartheta_n^i}\ \mathsf{SNumWF} \qquad \frac{n > 0}{\Gamma \vdash \vartheta_n^u}\ \mathsf{UNumWF}$$

$$\frac{\Gamma \vdash \tau \qquad \mathsf{num}(\tau) \qquad c > 0}{\Gamma \vdash \mathsf{Vec}(\tau, c)}\ \mathsf{VecWF} \qquad \frac{\Gamma \overset{n}{\underset{i=1}{\vdash}} A_i \qquad \Gamma \vdash B}{\Gamma \vdash A_1, \ldots, A_n \to B}\ \mathsf{ArrowWF}$$

$$\frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \forall \alpha.\, A}\ \mathsf{ForAllWF} \qquad \frac{\begin{array}{c}\Gamma \vdash A_1 \\ \vdots \\ \Gamma \vdash A_n\end{array}}{\Gamma \overset{n}{\underset{i=1}{\vdash}} A}\ \mathsf{WFs}$$

Figure 7: Well-formedness of types

$\boxed{\Gamma \vdash A \cap B \Rrightarrow C \dashv \Delta}$ Under context Γ, types A and B unify to C with output context Δ.

$\boxed{\Gamma \vdash A_i \cap B_i \overset{n}{\underset{i=1}{\Rrightarrow}} C_i \dashv \Delta}$ Under context Γ, types A_i and B_i unify to C_i with output context Δ.

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \cap A \Rrightarrow A \dashv \Gamma}\ \textsf{EqUni}$$

$$\frac{\Gamma \vdash \mathsf{Vec}(\tau, c) \qquad \Gamma \vdash \mathsf{Vec}(\sigma, c) \qquad \Gamma \vdash \tau \cap \sigma \Rrightarrow \pi \dashv \Delta}{\Gamma \vdash \mathsf{Vec}(\tau, c) \cap \mathsf{Vec}(\sigma, c) \Rrightarrow \mathsf{Vec}(\pi, c) \dashv \Delta}\ \textsf{VecUni}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1, \ldots, A_n \to R_A \qquad \Gamma \vdash B_1, \ldots, B_N \to R_B \\ \Gamma \vdash A_i \cap B_i \overset{n}{\underset{i=1}{\Rrightarrow}} C_i \dashv \Delta \qquad \Gamma \vdash R_A \cap R_B \Rrightarrow R_C \dashv \Delta\end{array}}{\Gamma \vdash A_1, \ldots, A_n \to R_A \cap B_1, \ldots, B_n \to R_B \Rrightarrow C_1, \ldots, C_n \to R_C \dashv \Delta}\ \textsf{ArrowUni}$$

$$\frac{\Gamma \vdash \widehat{\alpha} \qquad \Gamma \vdash \widehat{\beta}}{\Gamma \vdash \widehat{\alpha} \cap \widehat{\beta} \Rrightarrow \widehat{\alpha} \dashv \Delta}\ \textsf{ExAnyUni}$$

$$\frac{\Gamma \vdash \widehat{\vartheta}_1 \qquad \Gamma \vdash \widehat{\vartheta}_2}{\Gamma \vdash \widehat{\vartheta}_1 \cap \widehat{\vartheta}_2 \Rrightarrow \widehat{\vartheta}_1 \dashv \Delta}\ \textsf{ExUNumUni}$$

$$\frac{\begin{array}{c}\Gamma \vdash \widehat{\vartheta}_{s_1,c_1} \qquad \Gamma \vdash \widehat{\vartheta}_{s_2,c_2} \\ s_3 = s_1 \wedge s_2 \qquad c_3 = \max(c_1, c_2)\end{array}}{\Gamma \vdash \widehat{\vartheta}_{s_1,c_1} \cap \widehat{\vartheta}_{s_2,c_2} \Rrightarrow \widehat{\vartheta}_{s_3,c_3} \dashv \Delta}\ \textsf{ExSSNumUni}$$

$$\frac{\Gamma \vdash \widehat{\vartheta}_{s_1,c_1} \qquad \Gamma \vdash \widehat{\vartheta}}{\Gamma \vdash \widehat{\vartheta}_{s_1,c_1} \cap \widehat{\vartheta} \Rightarrow\!\!> \widehat{\vartheta}_{s_1,c_1} \dashv \Delta}\ \text{ExSUNumUni}$$

$$\frac{\Gamma \vdash \widehat{\mathsf{Vec}_1} \qquad \Gamma \vdash \widehat{\mathsf{Vec}_2}}{\Gamma \vdash \widehat{\mathsf{Vec}_1} \cap \widehat{\mathsf{Vec}_2} \Rightarrow\!\!> \widehat{\mathsf{Vec}_1} \dashv \Delta}\ \text{ExVecUni}$$

$$\frac{\Gamma \vdash \widehat{\mathsf{Vec}_\alpha} \qquad \Gamma \vdash \mathsf{Vec}(\tau, c)}{\Gamma \vdash \widehat{\mathsf{Vec}_\alpha} \cap \mathsf{Vec}(\tau, c) \Rightarrow\!\!> \mathsf{Vec}(\tau, c) \dashv \Delta}\ \text{ExVecCUni}$$

$$\frac{\Gamma \vdash \widehat{\vartheta} \qquad \Gamma \vdash \vartheta_{s,c}}{\Gamma \vdash \widehat{\vartheta} \cap \vartheta_{s,c} \Rightarrow\!\!> \vartheta_{s,c} \dashv \Delta}\ \text{ExUNumCUni}$$

$$\frac{\Gamma \vdash \widehat{\vartheta}_{s_1,c_1} \qquad \Gamma \vdash \vartheta_{s_2,c_2} \quad \neg s_1 \vee s_2 \qquad c_1 \leq c_2}{\Gamma \vdash \widehat{\vartheta}_{s_1,c_1} \cap \vartheta_{s_2,c_2} \Rightarrow\!\!> \vartheta_{s_2,c_2} \dashv \Delta}\ \text{ExSNumCUni}$$

$$\frac{\Gamma \vdash \widehat{\alpha} \qquad \Gamma \vdash \alpha}{\Gamma \vdash \widehat{\alpha} \cap \alpha \Rightarrow\!\!> \alpha \dashv \Delta}\ \text{ExAnyCUni}$$

$$\frac{\Gamma \vdash \forall \alpha.\, A \qquad \Gamma \vdash B \qquad \Gamma, \widehat{\alpha} \vdash A[\alpha := \widehat{\alpha}] \cap B \Rightarrow\!\!> B \dashv \Delta}{\Gamma \vdash \forall \alpha.\, A \cap B \Rightarrow\!\!> B \dashv \Delta}\ \text{ForAllUni}$$

$$\frac{\Gamma \vdash A \cap B \Rightarrow\!\!> C \dashv \Delta}{\Gamma \vdash B \cap A \Rightarrow\!\!> C \dashv \Delta}\ \text{CommutUni}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \cap B_1 \Rightarrow\!\!> C_1 \dashv \Theta_1 \\ \Theta_1 \vdash A_2 \cap B_2 \Rightarrow\!\!> C_2 \dashv \Theta_2 \\ \vdots \\ \Theta_{n-1} \vdash A_n \cap B_n \Rightarrow\!\!> C_n \dashv \Theta_n\end{array}}{\Gamma \vdash A_i \cap B_i \overset{n}{\underset{i=1}{\Rightarrow\!\!>}} C_i \dashv \Theta_n}\ \text{Unis}$$

Figure 8: Unification of types

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ Under input context Γ, e checks against input type A, with output context Δ

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ Under input context Γ, e synthesizes output type A, with output context Δ

$\boxed{\Gamma \vdash e_i \underset{i=1}{\overset{n}{\Rightarrow}} A_i \dashv \Delta}$ Under input context Γ, $\{e_i\}_1^n$ synthesizes types $\{A_i\}_1^n$, with output context Δ

$$\frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash A \cap B \Rightarrow\!\!\Rightarrow C \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \text{UniToCheck}$$

$$\frac{\Gamma \vdash \Gamma[x] \Rightarrow A \dashv \Delta}{\Gamma \vdash x \Rightarrow A \dashv \Delta}\ \text{IdSyn}$$

$$\frac{x = \texttt{true} \vee x = \texttt{false}}{\Gamma \vdash x \Rightarrow \text{Bool} \dashv \Gamma}\ \text{BoolSyn}$$

$$\frac{x = \texttt{NuC} \qquad C > 0 \qquad 0 \leq N < 2^C}{\Gamma \vdash x \Rightarrow \vartheta_{u,C} \dashv \Gamma}\ \text{SNumLitAnnSyn}$$

$$\frac{x = \texttt{NiC} \qquad C > 0 \qquad -2^{C-1} \leq N \leq 2^{C-1} - 1}{\Gamma \vdash x \Rightarrow \vartheta_{i,C} \dashv \Gamma}\ \text{UNumLitAnnSyn}$$

$$\frac{x = N \qquad N \in \mathbb{Z} \qquad sz = \text{min-width}(N) \qquad sgn = \text{sign}(N)}{\Gamma \vdash x \Rightarrow \widehat{\vartheta}_{sgn,sz} \dashv \Delta}\ \text{NLitSyn}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \dashv \Delta \qquad \Delta \vdash \text{Vec}(\tau, 1)}{\Gamma \vdash [e] \Rightarrow \text{Vec}(\tau, 1) \dashv \Delta}\ \text{Vec1LitSyn}$$

$$\frac{\begin{array}{c}\Gamma \vdash [e_1, \ldots, e_n] \Rightarrow \text{Vec}(\tau, i) \dashv \Theta_1 \\ \Theta_1 \vdash e_{n+1} \Rightarrow \sigma \dashv \Theta_2 \\ \Theta_2 \vdash \sigma \cap \tau \Rightarrow\!\!\Rightarrow \pi \dashv \Delta\end{array}}{\Gamma \vdash [e_1, \ldots, e_{n+1}] \Rightarrow \text{Vec}(\pi, i+1) \dashv \Delta}\ \text{VecLitSyn}$$

$$\frac{\begin{array}{c}op \in [+, -, /, \cdot] \\ \Gamma \vdash l \Rightarrow \tau \dashv \Theta \qquad \Gamma \vdash r \Rightarrow \sigma \dashv \Theta \\ \text{num}(\tau) \qquad \text{num}(\sigma) \\ \Theta \vdash \tau \cap \sigma \Rightarrow\!\!\Rightarrow \pi \dashv \Delta\end{array}}{\Gamma \vdash l \ op \ r \Rightarrow \pi \dashv \Delta}\ \text{NumFieldOpSyn}$$

$$\frac{\begin{array}{c}\mathsf{op} \in [<, \leq, >, \geq] \\ \Gamma \vdash \mathsf{l} \Rightarrow \tau \dashv \Theta \qquad \Gamma \vdash \mathsf{r} \Rightarrow \sigma \dashv \Theta \\ \mathsf{num}(\tau) \qquad \mathsf{num}(\sigma) \\ \Theta \vdash \tau \Leftarrow \sigma \dashv \Delta\end{array}}{\Gamma \vdash \mathsf{l}\ \mathsf{op}\ \mathsf{r} \Rightarrow \mathsf{Bool} \dashv \Delta}\ \mathsf{NumCmpOpSyn}$$

$$\frac{\begin{array}{c}\mathsf{op} \in [=, !=] \\ \Gamma \vdash \mathsf{l} \Rightarrow \tau \dashv \Theta \qquad \Gamma \vdash \mathsf{r} \Rightarrow \sigma \dashv \Theta \\ \Theta \vdash \tau \Leftarrow \sigma \dashv \Delta\end{array}}{\Gamma \vdash \mathsf{l}\ \mathsf{op}\ \mathsf{r} \Rightarrow \mathsf{Bool} \dashv \Delta}\ \mathsf{EqOpSyn}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{f} \Rightarrow F \dashv \Theta_1 \\ \Theta_1 \vdash e_i \overset{n}{\underset{i=1}{\Rightarrow}} A_i \dashv \Theta_2 \\ \Theta_2 \vdash F \cap A_i, \ldots, A_n \to R_A \Rrightarrow B_i, \ldots, B_n \to R_B \dashv \Delta\end{array}}{\Gamma \vdash (\mathsf{f}\ e_1\ \ldots\ e_n) \Rightarrow R_B \dashv \Delta}\ \mathsf{CallSyn}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{c} \Leftarrow \mathsf{Bool} \dashv \Theta_1 \\ \Theta_1 \vdash \texttt{br\_t} \Rightarrow A \dashv \Theta_2 \qquad \Theta_2 \vdash \texttt{br\_f} \Rightarrow B \dashv \Theta_3 \\ \Theta_3 \vdash A \cap B \Rrightarrow C \dashv \Delta\end{array}}{\Gamma \vdash \texttt{(if c br\_t br\_f)} \Rightarrow C \dashv \Delta}\ \mathsf{IfSyn}$$

$$\frac{\Gamma \vdash \mathsf{b} \Rightarrow B \dashv \Delta}{\Gamma \vdash \texttt{(let () b)} \Rightarrow B \dashv \Delta}\ \mathsf{Let0Syn}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \\ \Theta, (x : A) \vdash \texttt{(let (...) b)} \Rightarrow B \dashv \Delta\end{array}}{\Gamma \vdash \texttt{(let ((x, e\_1) ...) b)} \Rightarrow B \dashv \Delta}\ \mathsf{LetSyn}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Leftarrow T \dashv \Theta \\ \Theta, (x : T) \vdash \texttt{(let (...) b)} \Rightarrow B \dashv \Delta\end{array}}{\Gamma \vdash \texttt{(let ((x: T, e\_1) ...) b)} \Rightarrow B \dashv \Delta}\ \mathsf{LetAnnSyn}$$

$$\frac{}{\Gamma \vdash \texttt{scan} \Rightarrow \forall \mathsf{Vec}_\alpha.\, \mathsf{Vec}_\alpha \to \mathsf{Vec}_\alpha \dashv \Delta}\ \mathsf{ScanSyn}$$

Figure 9: Inference and checking of types

8 Zinnia Examples

```
let main: u8 = (let ((x : Vec<i8, 8>, [1, 4, 5, 7, 1, 2, 3, 4])) 2);
```

Example of using vector types - "main" is defined as returning an 8 bit integer, x is defined as getting a vector of eight 8-bit integers, and the two is returned as the result.

```
let main: i8 = (let ((x: i8, (+ 1u8 4i8))) (+ x 3));
```

Example of a mismatch caught by the type checker - 1 (an unsigned int) and 4 (a signed int) cannot be added together to get a signed int as a result.

References

[1] Guy E. Blelloch. *Prefix Sums and Their Applications*. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990.

[2] Jana Dunfield and Neelakantan R. Krishnaswami. "Complete and easy bidirectional typechecking for higher-rank polymorphism". In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming* (2013). URL: https://api.semanticscholar.org/CorpusID:7586176.

[3] David Durst et al. "Type-directed scheduling of streaming accelerators". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '20: 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation. London UK: ACM, June 11, 2020, pp. 408–422. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385983. URL: https://dl.acm.org/doi/10.1145/3385412.3385983 (visited on 05/14/2024).

[4] Matthew Gordon. *µ: A Functional Programming Language for Digital Signal Processing*. Apr. 9, 2003. URL: https://www.cs.unb.ca/tech-reports/honours-theses/Matthew.Gordon-4997.pdf.

[5] Andrew Lenharth and Chris Lattner. *CIRCT: Lifting hardware development out of the 20th century*. Nov. 17, 2021. URL: https://llvm.org/devmtg/2021-11/slides/2021-CIRCT-LiftingHardwareDevOutOfThe20thCentury.pdf.

[6] Mathworks. *Support SPI Communication*. URL: https://www.mathworks.com/help/simulink/supportpkg/raspberrypi_ug/support-spi-communication.html.

[7] Rachit Nigam et al. "A compiler infrastructure for accelerator generators". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Virtual USA: ACM, Apr. 19, 2021, pp. 804–817. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446712. URL: https://dl.acm.org/doi/10.1145/3445814.3446712 (visited on 05/11/2024).

[8] Gordon Stewart et al. "Ziria: A DSL for Wireless Systems Programming". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15: Architectural Support for Programming Languages and Operating Systems. Istanbul Turkey: ACM, Mar. 14, 2015, pp. 415–428. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694368. URL: https://dl.acm.org/doi/10.1145/2694344.2694368 (visited on 05/11/2024).