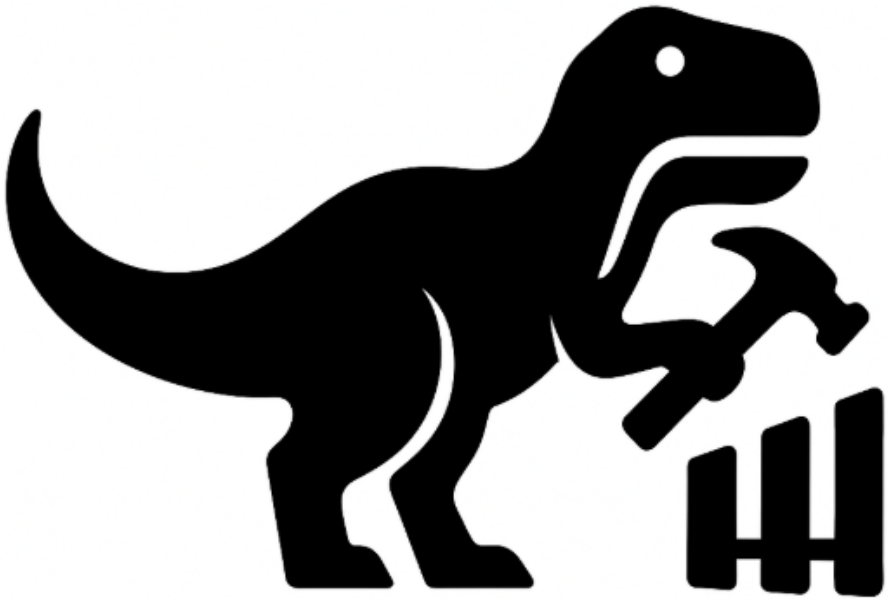
ABAX

ASIC Backend for Allo in XLS



ABAX: Allo Building a Xylephone

ECE 6775 Final Project Report

School of Electrical and Computer Engineering
Cornell University

Jisoo Kim (jk2336)
Cornell University
Ithaca, NY, USA
jk2336@cornell.edu

Zephan Sanghani (zrs29)
Cornell University
Ithaca, NY, USA
zrs29@cornell.edu

Anthony Song (abs343)
Cornell University
Ithaca, NY, USA
abs343@cornell.edu

Eric Zhou (ez255)
Cornell University
Ithaca, NY, USA
ez255@cornell.edu

Submitted: December 12, 2025

Table of Contents

Contents

ABAX: ASIC Backend for Allo in XLS

Jisoo Kim (jk2336)
Cornell University
Ithaca, NY, USA
jk2336@cornell.edu

Zephan Sanghani (zrs29)
Cornell University
Ithaca, NY, USA
zrs29@cornell.edu

Anthony Song (abs343)
Cornell University
Ithaca, NY, USA
abs343@cornell.edu

Eric Zhou (ez255)
Cornell University
Ithaca, NY, USA
ez255@cornell.edu

Abstract

With the end of Dennard scaling, there has been a significant increase in the development of special-purpose hardware accelerators designed to meet the ever-growing demand for compute. While these accelerators—typically implemented on Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs)—offer superior performance and energy efficiency, their design process is arduous and complex. High-Level Synthesis (HLS) frameworks aim to mitigate this challenge by automating the translation of algorithmic descriptions into hardware, thereby significantly reducing the barrier to entry for hardware engineers. Allo is one such composable HLS framework specifically targeting spatial accelerator design, allowing designs defined in Python to be lowered to FPGA RTL. In this work, we introduce ABAX, an extension for Allo that expands its backend support to target Google's Accelerated HW Synthesis (XLS) toolchain. ABAX acts as a bridge, transpiling Allo designs to XLS's two frontends: DSLX and XLS [CC]. To validate this flow, we implement five representative kernels—scalar add, vector-vector add (vvadd), matrix-vector multiplication (GEMV), matrix multiplication (GEMM), and systolic arrays—and successfully lower them to ASIC RTL. This work demonstrates Allo's novel capability to target ASIC flows, unifying the composable programming model across both FPGA and ASIC targets.

1 Introduction

The recent trends in technology scaling has led to a rise in development for special-purpose hardware accelerator to meet the increasing computational demands of emerging applications. Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) now dominate many data-parallel and signal-processing domains, offering substantial improvements in throughput and energy efficiency. However, the design cost, verification effort, and long turnaround times of traditional RTL-based flows have raised the barrier to accelerator deployment.

High-Level Synthesis (HLS) tools lift design entry to software-like languages (C/C++, SystemC, or DSLs) and automatically generate RTL while handling scheduling, resource sharing, and control synthesis. HLS provides engineers with: (1) efficiency by avoiding large amounts of hand-written RTL, and (2) accessibility by leveraging programming abstractions more familiar to both hardware and software engineers. Yet, practical HLS use remains challenging: performance is sensitive to pragmas and memory banking/interconnect must be carefully orchestrated. Moreover, many HLS flows are tailored to FPGAs; ASIC-oriented flows often demand tighter timing closure, explicit pipelining, and careful management of technology-specific libraries.

Commercial HLS tools [3, 4] focus primarily on C/C++ inputs and FPGA targets. Current Accelerator Design Languages (ADLs) [9, 10, 12, 13] focus on optimizing single kernels with performance being rather lackluster when it comes to multi-kernel designs. Finally, most Python-based hardware DSLs [5, 6, 7] generate RTL but offer limited automated scheduling or ASIC-oriented backend support.

Allo is a Python-embedded DSL and compiler aimed at spatial accelerator design. It proposes fixes to problems current ADLs face, being designed to handle optimizations on single and multi-kernels[1]. However, Allo's existing backend primarily targets FPGAs, leaving ASIC-oriented flows underserved. Google's XLS is an open-source hardware compiler stack with an interpreter, JIT simulation, and formal verification hooks, enabling test-driven hardware development before RTL emission. Its schedulers handle pipelining and resource sharing, targeting both FPGA and ASIC flows. [2]

A gap remains between Allo's composable, Python-based spatial design model and XLS's ASIC-capable backend. Without a bridge, Allo users cannot natively access ASIC-oriented flows, and XLS users miss higher-level spatial abstractions. In ABAX, we address this gap by translating Allo designs into code for XLS's frontend, preserving structural information (e.g., pipelines, array shapes, bit widths) needed for effective scheduling and synthesis. In contrast to prior HLS tools and Python-based DSLs, ABAX unifies Allo's expressive and reusable spatial programming model with XLS's schedulers, formal verification hooks, and ASIC-ready synthesis, establishing a practical Python-to-ASIC compilation flow.

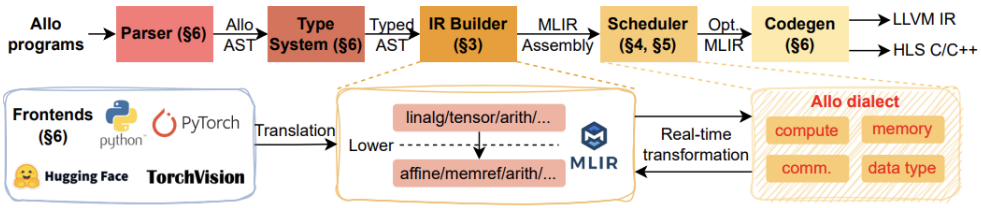


Figure 1: Base Allo Compilation Flow [1]

To test our flow, we evaluate several representative linear-algebra kernels—scalar add, vector-vector add (vvadd), matrix-vector multiplication (GEMV), matrix multiplication (GEMM), and systolic arrays—as proofs of concept. These kernels exercise key HLS features such as array handling, parallelism optimizations (e.g., unrolling and pipelining), and channel/memory interfaces. Successfully lowering them from Allo to XLS and generating ASIC-ready RTL, with synthesis and place-and-route validating feasibility, demonstrates the effectiveness of the ABAX bridge and motivates broader kernel coverage in future work.

2 Problem Description

This work explores the integration of Allo and XLS. We now present a overview of both frameworks and motivate our choice of these tools.

Allo is an ADL and compiler framework that cleanly decouples algorithm specification from hardware customization, with a strong focus on flexible memory and communication design. It supports parameterized kernels and cross-kernel composition through composable scheduling and holistic dataflow optimization, enabling efficient mapping of large, multi-kernel accelerators. This makes Allo well suited for designs that are difficult or impractical to express in prior ADLs [9, 11, 13].

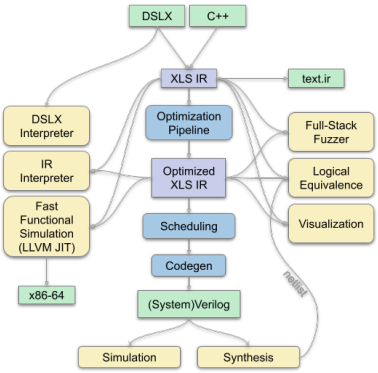


Figure 2: XLS Flow [2]

Allo is implemented as a Python-embedded ADL with an MLIR-based compilation flow. Figure 1 illustrates the process with programs first being parsed into a Abstract Syntax Tree (AST), then lowered to the Allo dialect—a dialect within the MLIR ecosystem—through a IR builder. The Allo dialect is built to separates hardware customization from computation. Customization primitives are applied at the IR level. The flow integrates with different in-tree MLIR dialects, enabling generation of designs for many backend targets [1]. This design philosophy allows Allo to extend beyond FPGA-oriented workflows and naturally support other backends, for example our ASIC one, motivating it as our choice of ADL to expand on.

XLS is an open-source high-level synthesis toolchain that generates synthesizable Verilog / SystemVerilog from its two frontends, DSLX (rust based functional language) and XLS [CC] (C++). Designs can be interpreted or JIT-compiled for fast validation, then lowered to hardware with strong correctness guarantees supported by formal equivalence checking and fuzzing (reference Figure 2 for entire flow). XLS supports both stateless pipelined functions and stateful, channel-based processes for streaming architectures. It is built on a well-defined IR with optimization and automatic Verilog code generation. The previously mentioned features as well as the explicit separation between functional specification, scheduling, and code generation makes it easier to integrate XLS with higher-level accelerator description languages such as Allo making it our choice for the tool that generates the ASIC hardware.

However, there do exist a few significant pitfalls for the conversion from Allo to XLS. First, XLS lacks built-in memory support. Furthermore, even if we manually add memory support, XLS fully unrolls loops, requiring our backend to strategically unroll/pipeline loops if a Allo function uses arrays or memory. Finally, XLS supports floating-point and integer types but not fixed-point arithmetic, requiring alternative implementations for fixed-point operations or to sacrifice support for it. These constraints as well as the solutions proposed to fix them will be further discussed and flushed out within the implementation section.

3 Implementation

3.1 DSLX Backend

DSLX is a supported frontend of XLS, and is a rust-like "dataflow-oriented functional language" [8]. DSLX supports two constructs for implementing hardware, functions and communicating sequential processes (procs). Functions are pure dataflow, lacking an explicit notion of time. Thus, for this backend, we focus on the procs, as they permit stateful representations — necessary to encode most Allo programs.

However, DSLX has a few major limitations and as such differs significantly from conventional high-level synthesis languages. First, DSLX has no notion of memory. In a typical HLS flow, users load/store to arrays, which are synthesized into BRAMs in the FPGA fabric. In DSLX, all arrays are synthesized as register files. As such, implementing common linear algebra (matrix) operations quickly uses a lot of area.

Second, DSLX has no notion of control flow. As previously mentioned, DSLX is "dataflow-oriented". All loops must be statically analyzable, and during code generation, "the XLS compiler will unroll and specialize the iterations" [8]. As such, loops over arrays are unrolled and synthesized into large datapaths.

This limitations incentivize a custom lowering strategy that resolves these issues and enables efficient ASIC RTL to be generated directly from Allo MLIR. To this end, we implemented our DSLX backend.

Resolving DSLX Limitations

In order to create strategies to resolve these limitations, we must take a closer look at the communicating sequential process (proc). See Figure 3.

The proc has four components. First, it has input/output streams called "channels". Each channel has it's own val/rdy interface. Then, we define an initial state for the proc (which is set upon reset). Lastly we have "next". This takes a state, does something with it (in the body), and produces the next state. It is syntactically similar to a DSLX function.

"next" is the key construct which enables stateful/temporal behavior. Ideally, it executes once a clock cycle. However, it is limited by the number of clock cycles needed to derive the next state from the current state (recurrence). The datapath is automatically pipelined by the compiler.

Channel operations may be blocking. As such, if any such operation (send/recv) blocks, the pipeline is stalled to handle the backpressure.

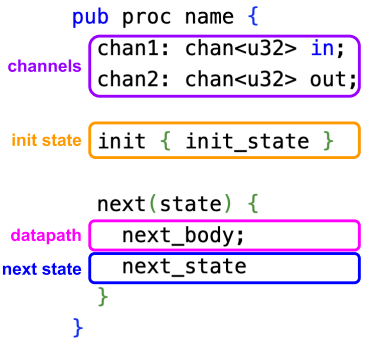


Figure 3: A "proc"

We will use these constructs, specifically channels and state, to map memory and control flow into DSLX. Specifically, we will utilize channels for memory. We construct a output channel for the memory request, and a input channel for the memory response. We use I/O constraints to enforce a specified memory latency. This pushes the responsibility for instantiating and synthesizing memory downstream, post RTL codegen.

Next, we use state to manage control flow. Specifically, we are able to keep track of loop iteration variables, and other accumulators in this state (which synthesize into registers). Then, we use the temporal-nature of a proc to execute for loops without unrolling. This provides substantially more area-efficient implementations.

In a sense, our lowerer must bind memory and schedule control flow itself. As such, it's more than a transpiler, as DSLX is at a significantly lower level of abstraction than a typical high-level synthesis language.

The Compiler

For our compiler, we lower the multi-level intermediate representation (MLIR) generated by Allo into DSLX. First, we clean the MLIR by applying various passes (i.e. dead-code elimination) before starting our lowering. Next, we analyze the MLIR to detect whether or not this program is stateful (time-multiplexed); and, we send it down either the combinational or stateful lowerer pipeline (see Figure 4). First, we detail the former.

First, the compiler analyzes the MLIR to extract the inputs/outputs (using existing Allo utilities). It then defines an input and output channel for each of this, converting MLIR types into DSLX types. Finally, it stores the mapping from input/output to channel such that the instruction emitter uses the proper channel.

Once the channels are setup, we must only emit the "next" function (there is no state for combinational). First, we emit a channel read (recv) for each of the inputs, storing them in DSLX variables. We maintain a mapping between the MLIR SSA names and DSLX variables. Then, we lower the MLIR instructions one at a time into corresponding DSLX code. Finally, we emit a channel write (send) for each of the outputs.

DSLX may not maintain the ordering of channel operations. As such, it requires annotation using tokens.

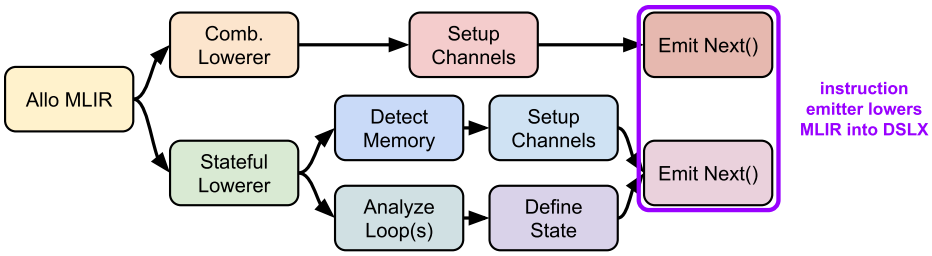


Figure 4: The Allo → DSLX Compiler Flow

A channel operation produces a token once it completes and can itself be guarded by any number of tokens. Thus, each of the output channel sends are guarded by the tokens from all of the input channel receives.

Next, we discuss the stateful lowerer, a more complicated lowerer that builds on the combinational lowerer. This lowerer handles the solutions we found earlier for the limitations of DSLX. Those being: converting memory to channels and converting control flow (and accumulators) into state. Thus, it follows that the first few steps of our lowerer include analyzing the MLIR to extract memory and loop information.

First, we detect memory. We look at each memory reference in the MLIR. If they are non-scalar, they are mapped to memory. We also analyze whether they are read/write in order to provide the correct ports. Currently we only support 1R1W SRAM. To compose modules (procs), one can write, while another reads. Additionally, if a proc uses memory, we add go/done channels (analogous to ap_start and ap_done in HLS).

Once all memory channels are determined, we analyze other scalar input/output channels. Then, we are able to emit the channel definitions in the proc. We store memory ↔ channel mappings for use in lowering.

Next, we must analyze and interpret the various loops found in the MLIR. Our lowerer supports nested loops (for/while, static/dynamic), and each can have it's own pre/post-amble. This analysis involves partitioning each of the instructions into the loops, and whether they are found in the body or pre/post-amble.

Next, we must extract the state required by these loops. First, we trivially add the loop iterators (for loops) as these are always necessary. Otherwise, we must do a more detailed search for state variables. Namely, we scan each scalar memory allocation — since Allo programs typically use these for their accumulators — and collect those which are used inside/after the loops. All of these variables are stored in the proc state. In the state, we store an additional busy bit for whether or not the accelerator is currently doing something.

At this point, we have determined both the state variables and the channels. As such, we are able to emit both the channel declarations as well as the initial state. Thus, all that remains is emitting next.

Similar to the combinational lowerer, we lower one MLIR instruction at a time into DSLX code. However, we must also consider complicated interactions with both control flow and memory when lowering.

First, if we are not busy, we do blocking reads on each of our inputs (possibly the go signal). This stalls our datapath until our inputs are received, at which point we become busy. We have a status signal for if we just became ready. Similarly, we have status signals for if (nested) for loops just started or just ended. By guarding DSLX instructions on this status signals, we are able to correctly execute loop pre/post-ambles.

At this point, we lower each MLIR instruction, guarding on the proper signals. This re-uses the same instruction emitter as the combinational lower used. However, we must now also deal with memory. If we reach a affine load/store (memory instruction), we must correctly map it to the request/response channels.

First, we construct the memory request with the appropriate address and optionally data. Then, we send it to the memory via the request channel — any multi-dimensional array is linearized such that we can index into it with a 1-dimensional address. Then, we receive it from the response channel. This is a multi-cycle operation, and as such, will naturally be placed in different pipeline stages by the XLS compiler.

After the MLIR instructions are lowered, we implement the control logic (incrementing/resetting loop variables). To properly execute nested loop iteration, we implement carry logic. When the innermost loop completes (index reaches bound), we reset it to zero and carry to the next outer loop. Thus, one innermost loop iteration runs each cycle. This continues until all loops are complete, at which point we signal done.

Lastly, not every loop needs to be time-multiplexed. To that end, we have added support for loop unrolling. Once the loop has been analyzed (namely, identifying accumulators), we utilize a DSLX for loop to implement it. Since the XLS compiler automatically unrolls this, loop unrolling is achieved. But, there are restrictions: no channel (memory) operations can occur; as, they only support one operation per cycle.

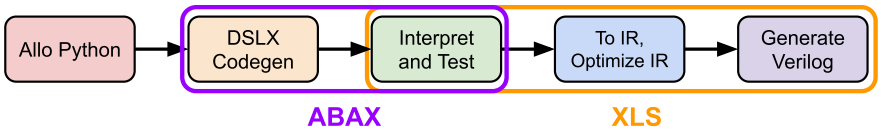


Figure 5: The DSLX flow in Allo

The Allo DSLX Flow

The DSLX flow is fully-integrated within Allo to improve user experience. Namely, users can run through the flow depicted in Figure 5 entirely within Allo. For an example, see Figure 6.

First, the user customizes the Allo module, and then generates the DSLX code (via our compiler). Then, they can interpret the code to verify it's validity. Next, they can use our framework to automatically generate and interpret DSLX tests from Python. These tests can be scalar or vector (via numpy).

Finally, they can utilize the XLS compiler to generate and optimize the XLS IR. Finally, this can be converted to verilog. The user can specify the latency of their RAMs as well as the number of pipeline stages (or their desired clock period).

However, our compiler is unable to deal with complex control (i.e. consecutive loops) and RAW hazards. Additionally, we support very few Allo optimizations.

```python
# define allo function
def vvadd(a: int32[16], b: int32[16]) -> int32[16]:
  c: int32[16] = 0
  for i in range(16):
    c[i] = a[i] + b[i]
  return c

# build the function in XLS
s = allo.customize(vvadd)
code = s.build(target='xls')

# interpret code to verify validity
code.interpret()

# test the generated DSLX
a = np.arange(16, dtype=np.int32)
code.test(a, a * 2, a * 3)

# run it through the XLS flow
code.to_ir(); code.opt()
code.to_vlog(ram_latency=1, pipeline_stages=3)
```

Figure 6: An Example DSLX Flow

3.2 DSLX Systolic Arrays

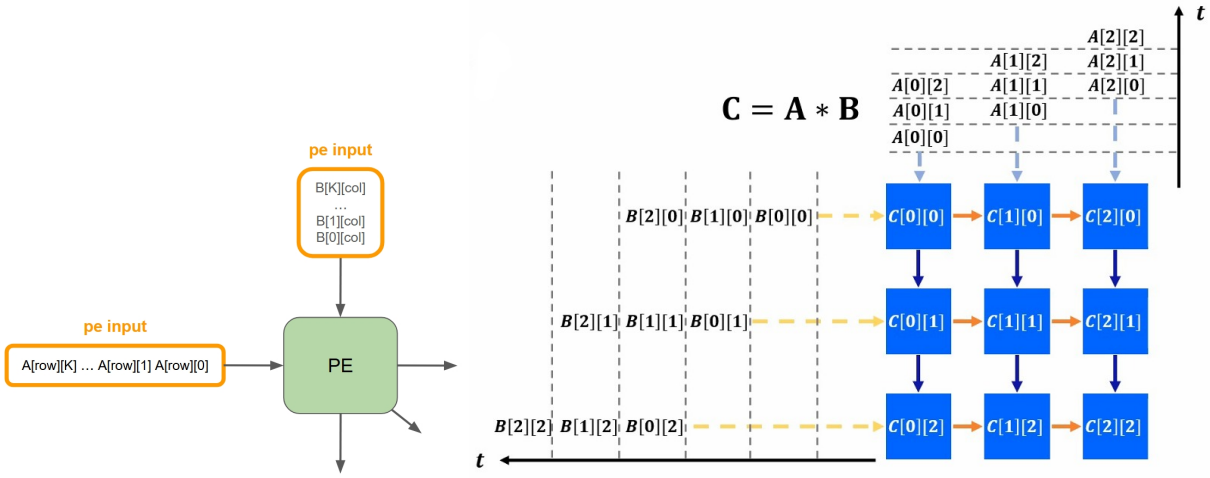


Figure 7: Output-Stationary Systolic Array Implementation of Matrix Multiplication

Since Allo is specifically aimed at spatial accelerator design,
we wanted to extend our lowering to support basic systolic array structures. Our goal was to lower a high-level Allo description of an output-stationary systolic array into correct, cycle-accurate DSLX procs while preserving the intended wavefront dataflow semantics. We decided to support the output-stationary matrix multiplication flow, as seen in Figure 7 [14], since it naturally maps to stateful processing elements (PEs), minimizes output bandwidth, and allows accumulation to remain local to each PE.

This design has two primary components: individual processing elements (PEs) and the systolic array.

PE Lowering

PEs each have two input channels from the left and top, and two output channels on the right and bottom. Each PE also has an additional output channel used to emit the final accumulated result. The key invariant

enforced during PE lowering is that exactly one element from each input channel is consumed per cycle and forwarded in the same cycle, while the accumulated output remains local to the PE until completion.

In Allo, our PE kernel implementation is almost identical to the example module, as shown in Appendix 12a. The only difference is that the reduction index is not explicitly passed and the output is returned directly.

However, to support this implementation, we encountered several issues in the existing lowering flow. Originally, our DSLX lowerer converted all array inputs into memory-backed accesses, which breaks the streaming required for systolic PEs. To address this, we introduced a PE-specific lowering path that interprets array-typed inputs as FIFO channels when their access pattern corresponds to a streaming producer–consumer relationship. This involved three steps. First, we detect PE modules using a simple naming-based check. If a PE module is identified, rather than treating every input array as memory, we interpret the first input as a channel and identify additional channels by examining the array shape and access pattern (consistent forward index increments). Finally, input channels are read using `recv` operations each cycle instead of through memory accesses.

By making these changes specific to PE modules, our lowerer continues to work for all standard combinational and stateful modules, while reusing the same lowering logic to process the internal operations of the PE and any additional arguments that are not impacted by this distinction. The final generated DSLX code for a PE is shown in Appendix 12b.

Systolic Array Lowering

For the systolic array, we wanted to support the staggered-input, output-stationary systolic array structure described earlier. The corresponding Allo implementation is shown in Appendix 13. Due to the additional structure and coordination required, we implemented a custom lowerer specifically for systolic array modules.

This lowerer begins with a scan of the Allo IR to identify the overall structure of the systolic array. This includes determining the PE grid shape based on loop bounds, and distinguishing between loops used for configuration (PE creation and wiring) and those that correspond to temporal behavior executed every cycle.

Reusing logic from the DSLX lowerer, the systolic lowerer first creates the appropriate input and output channels. Based on the inferred grid shape and PE instantiation pattern, internal channels are then created to connect neighboring PEs and to collect final results. In the `config` function, all channels are declared and the PE grid is constructed by spawning individual PE procs and connecting them using these internal channels.

In the `next` function, the temporal behavior of the systolic array is implemented. Loop state is derived from the inputs, state variables, and any necessary accumulators. Each cycle, data is injected into the left and top boundaries of the array. The blocking behavior of the PE procs ensures the desired wavefront-style staggered execution. Input matrices are read and stored during the initial cycles, while outputs are gathered from the PE result channels and emitted in the final cycle. Additional logic is generated to explicitly consume and discard values exiting the right and bottom boundaries of the array. The final generated DSLX code for the systolic array is shown in Appendix 14.

Currently Supported Features and Limitations

Currently, a wide range of PE implementations are supported, with array inputs and outputs being converted into FIFO channels as appropriate. PE-internal operations and stateful behavior are fully handled by the existing DSLX lowerer. The primary restrictions are that PE modules must have `pe_` as a prefix or `_pe` as a suffix, the PE must use the expected array input format for channel inference, and the first input to the PE must be a channel, with all other channels sharing the same shape.

For systolic arrays, the lowerer supports any grid size, and any systolic array flow that follows a left-and-top input pattern with values moving rightward and downward. All numeric types supported by the DSLX lowerer are handled correctly. However, systolic array modules must include `systolic` in their name, and only 2D arrays with this fixed flow direction are currently supported. Additionally, memory is not yet integrated into the systolic lowering path, which results in large registers being used to store matrix inputs.

Future improvements include integrating memory support, refining state representations, and generalizing channel creation and connection during PE spawning.

```python
import allo
from allo.ir.types import int32

def add(a: int32, b: int32) -> int32:
    return a + b

# Schedule construction
s = allo.customize(add)

# Codegen
mod = s.build(target="xlscc", use_memory=False)
```

(a) An example Allo program

```
module {
  func.func @add(
      %arg0: i32,
      %arg1: i32) -> i32 {
    %0 = arith.extsi %arg0 : i32 to i33
    %1 = arith.extsi %arg1 : i32 to i33
    %2 = arith.addi %0, %1 : i33
    %3 = arith.trunci %2 : i33 to i32
    return %3 : i32
  }
}
```

(b) The corresponding MLIR code

```
#include "/xls_builtin.h"
#include "xls_int.h"
template <int W, bool S = true>
using ac_int = XlsInt<W, S>;
int add(int v0, int v1) {
  ac_int<33, true> v2 = v0;
  ac_int<33, true> v3 = v1;
  ac_int<33, true> v4 = v2 + v3;
  int v5 = v4;
  return v5;
}
```

(c) The generated XLS [CC] code

Figure 8: An example Allo program and the corresponding MLIR and XLS [CC] code

3.3 XLS [CC] Backend

XLS [CC] is an alternative frontend in the XLS compiler stack. Developed by a sister team at Google, it uses C++ syntax and targets the same backend, sharing the open-source XLS flow. Like DSLX, XLS [CC] supports both functions and procs, enabling combinational and stateful, cycle-accurate hardware descriptions. It also exposes features not directly available in DSLX, including explicit on-chip memory interfaces and loop pipelining pragmas with user-specified initiation intervals, providing finer control over scheduling and microarchitectural structure.

XLS [CC] Limitations

Translating Allo to XLS [CC] is challenging because XLS [CC] is limited to a restricted subset of C++. In XLS [CC], only integer types are supported, requiring floating- and fixed-point operations to be emulated in integer arithmetic. The memory model maps arrays either to registers or to SRAM-backed memory abstractions, which require additional memory interface information on read request/response channels and write request/completion channels with valid/ready handshaking. Multi-dimensional arrays must be flattened into one-dimensional memories, losing shape information.

XLS's execution model also differs from Allo's sequential style: arrays cannot be passed through function calls and must instead be communicated via channels, shifting designs toward a streaming model. Loop transformations require explicit pipeline or full-unroll pragmas, with no support for higher-level transformations such as tiling, fusion, or dataflow. Loops that access channels cannot be unrolled and must be pipelined, while advanced scheduling features like buffer insertion and inter-kernel streaming must be implemented manually.

Consequently, the translation system must perform type lowering, static shape resolution, memory and channel mapping, loop restructuring, and scheduling simplification while preserving computation semantics.

Resolving XLS [CC] Limitations

In ABAX, we address a subset of the XLS [CC] limitations through early validation and backend transformations, enabling the end-to-end Allo-to-XLS [CC] flow shown in Figure 8. The compiler validates the MLIR for unsupported types, dynamic shapes, and invalid scheduling directives, accepting only fixed-integer and fixed-point types; fixed-point values are emitted as templated integer-based structs.

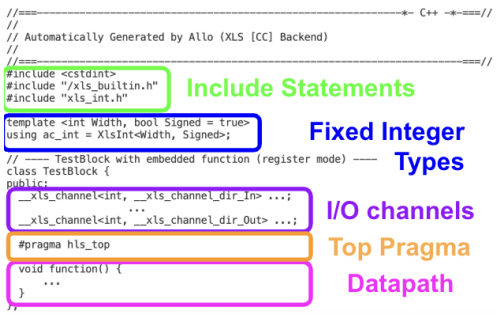


Figure 9: "TestBlock" format required by XLS [CC]

The backend supports two array modes. In register mode, arrays are emitted as multi-dimensional C arrays that lower to registers in XLS. In memory mode, memory declarations are emitted using the XLS [CC] memory interface, along with RAM configuration files mapping abstract memories to concrete implementations. Multi-dimensional arrays are flattened into one-dimensional memories with stride-based index calculations.

Array function parameters are converted to channel-based communication. Functions with array arguments are wrapped in a TestBlock class (Figure 9), where arrays become class members or channel interfaces and are streamed element-by-element through pipelined loops, transforming function calls into a streaming execution model.

Loop pragmas are generated automatically via loop analysis: loops are either pipelined or fully unrolled, with channel-accessing loops forced to pipeline to preserve correct semantics, deterring multiple channel reads per cycle.

These transformations preserve computation semantics while adapting control flow and memory access patterns to XLS's execution and compiler constraints.

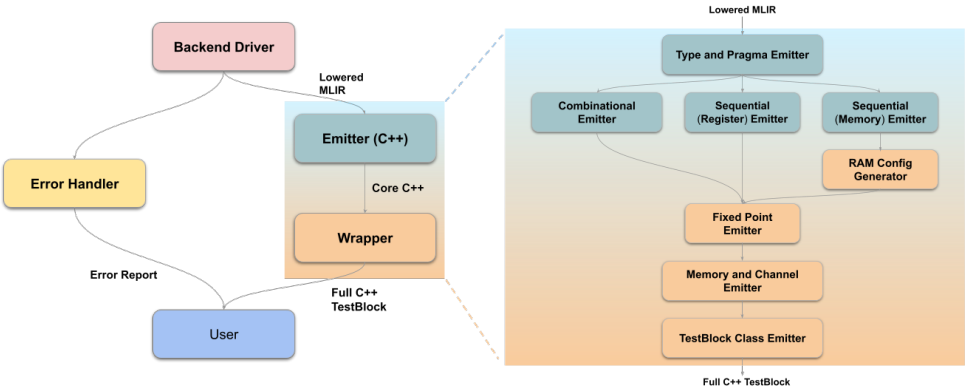


Figure 10: Overall Allo → XLS [CC] Compiler Flow (left), Detailed Emitter and Wrapper logic (right)

The Compiler

The XLS [CC] backend transforms Allo's high-level MLIR representation into XLS-compatible C++ code through a multi-stage pipeline that addresses the constraints of XLS [CC]. Figure 10 summarizes the flow. Compilation begins by lowering Allo Python to MLIR, preserving computation structure along with scheduling and memory annotations.

A backend driver manages the subsequent stages and performs early legality checks to ensure the MLIR conforms to XLS [CC] requirements. It rejects unsupported types (floating-point, dynamic shapes), invalid scheduling directives, and non-constant array dimensions. This early validation provides information to the error handler which returns a error report and prevents generation of incompatible code.

After validation, the emitter produces core C++ code by traversing the MLIR and emitting variables, types, and arithmetic operations. It selects an execution strategy based on the function interface: purely scalar kernels are emitted directly as combinational logic, while kernels with array parameters are converted to sequential logic with a streaming interface and explicit load and store phases. For arrays, the emitter generates C-style array declarations in register mode; in memory mode, it emits placeholder annotations and array indexing syntax, deferring the memory interface declarations to the wrapper. Multi-dimensional arrays are flattened to a one-dimensional layout with rewritten indices. While traversing loop nests, the emitter inserts pipeline and unroll directives to reconstruct scheduling intent from the MLIR annotations.

A wrapper stage then produces the final C++ artifact by encapsulating the core code in a TestBlock class structure. It first emits the floating point struct if necessary. The wrapper then parses the emitter's placeholder annotations to generate memory interface declarations when in memory mode, adds channel declarations for streaming I/O, and includes the required header directives. It also generates auxiliary configurations such as the RAM rewrites textproto that guides XLS in mapping abstract memories to concrete RAM implementations. This separation allows the emitter to focus on the computational core while the wrapper handles XLS-specific class structure and configuration artifacts.

Currently Supported Features and Limitations

The XLS [CC] backend currently supports integer and fixed-point types of arbitrary length, and their arithmetic. It can emit single-function lowering for both combinational logic (scalar kernels) and sequential logic with array parameters, the latter supporting both register mode (arrays as on-chip registers) and memory mode (arrays as explicit SRAM interfaces). For loop scheduling, the backend supports pipelining with user-specified initiation interval and full loop unrolling. Floating-point types are not supported, as XLS[CC] lacks native floating-point hardware primitives. Other Allo scheduling primitives are currently unsupported and will produce early validation errors with diagnostic guidance.

4 Evaluation

Since the primary objective of ABAX is to generate synthesizable, ASIC-ready RTL, we conducted a rigorous testing process to verify the functionality and synthesizability of the output.

4.1 Functionality Testing

In order to verify that ABAX generates correct XLS frontend code, we created a test suite of auto-generated tests spanning combinational and sequential logic, where generated C++ and DSLX code is executed against reference Python implementations.

4.2 ASIC Synthesis & Place and Route Results

In order to check the synthesizability, we executed a standard ASIC synthesis and physical design flow using commercial tools and the NanGate 45nm Open Source PDK[15]. Five distinct kernels (matrix vector multiplication, vector-vector add, Matrix Multiplication, and Systolic Array) were processed through this flow to evaluate Area and Power Consumption. To ensure an apples-to-apples comparison, we minimized variations by keeping physical design constraints constant, varying only the target clock period to prevent negative clock skew from happening(for XLS[CC] kernels with memory abstraction, some delay constraints were relaxed for to mitigate clock skew violations). We assert that slight adjustments in synthesis parameters do not affect the validity of our analysis regarding area and power. Note that this evaluation focuses on validating the full-flow synthesizability of ABAX-generated RTL and comparing the two frontends; therefore, aggressive area and power optimizations were not prioritized.

Kernel	**Area** (μm^2)			**Power** (mW)		
	DSLX	**XLS [CC]**	**XLS [CC] mem**	**DSLX**	**XLS [CC]**	**XLS [CC] mem**
mv	4955	32252	5182	2.78	4.93	2.33
mm	5460	78778	5309	2.95	13.00	2.35
vvadd	1393	12734	1767	1.32	13.13	1.43
systolic	5228	–	–	2.45	–	–

Table 1: Post-PnR Area and Power Comparison Between DSLX, XLS [CC], and XLS [CC] with mem

1. `mv.v`: The 32-bit integer 4×4 matrix-vector multiplication kernel serves as a representative workload for modern computing, exemplified by Fully Connected layers in Convolutional Neural Networks (CNNs). As detailed in Table 1, the baseline XLS [CC] implementation incurs significant overhead, exhibiting a 6.5× increase in area and 1.8× higher power consumption compared to DSLX. This inefficiency stems from limitations in XLS [CC]'s memory support: rather than utilizing SRAM, the generated Verilog latches all 20 input integers (16 matrix, 4 vector) into register clusters prior to computation. However, applying memory abstraction eliminated these register clusters, reducing the area to 5182 μm^2 and power to 2.33 mW. Notably, this optimization brings the area only within 4.58% of the DSLX's `mv.v`.

2. `mm.v`: The 32-bit integer 4×4 matrix multiplication kernel exemplifies modern computing workloads, particularly the attention score computation in Transformer models. The XLS [CC] post-PnR implementation exhibits a 10.7× increase in area and a 4.4× increase in power consumption. The source of this disparity mirrors the memory limitations observed in the matrix-vector kernel (`mv.v`). The magnitude of the overhead is notably larger here because the number of inputs requiring register latching increases from 20 to 32. Consequently, the additional logic and routing resources needed to accommodate these registers significantly inflate the total area and power. However, applying memory abstraction eliminated these register clusters, reducing the area to 5309 μm^2 and power to 2.35 mW. Notably, this optimization brings the area only within 2.77% of the DSLX's `mm.v`.

3. `vvadd.v`: The 32-bit integer vector-vector addition kernel (vector size 16) was a primary focus of our analysis. It models workloads requiring both memory interaction and control flow while avoiding the complexity of Read-After-Write (RAW) hazards, making it an ideal baseline for testing implementation flows. Consistent with other kernels, the XLS [CC] post-PnR implementation exhibits a 9.1× increase in

area and a 10× increase in power consumption due to identical memory limitations. However, applying memory abstraction eliminated these register clusters, reducing the area to 1767 μm^2 and power to 1.43 mW.

Overall, the substantial area and power overheads observed in the XLS[CC] modules were eliminated via memory abstraction, resulting in area utilization and power consumption comparable to those of the DSLX modules. This finding validates that the extensive register clustering in XLS[CC] was the primary driver of the excess resource utilization.

5 Project Management

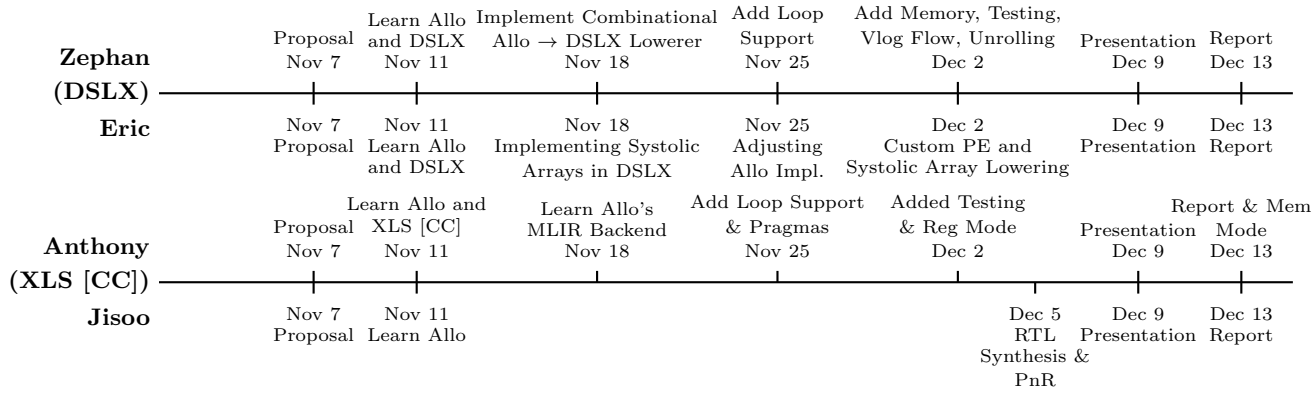


Figure 11: Project Timeline (Nov 1 – Dec 15).

We had linear project progression; we made consistent effort and progress each week. Management-wise, we did not have any challenges over the course of this project, and we achieved what we set out to do.

6 Conclusion and Acknowledgements

In this work, we presented **ABAX**, a backend for Allo that enables an ASIC-oriented compilation flow by targeting Google's XLS toolchain. ABAX translates Allo's spatial accelerator descriptions into XLS frontends, allowing Python-based kernels to be lowered to ASIC RTL. By supporting both DSLX and XLS [CC], ABAX demonstrates that Allo's programming model extends naturally beyond FPGA-centric workflows.

ABAX implements two complementary lowering strategies. The DSLX backend lowers Allo programs into stateful procs, explicitly encoding control flow, loop iteration, and memory using state and channels. The XLS [CC] backend emits structured C++ annotated with channels and scheduling pragmas, relying on XLS for pipelining and resource sharing. These backends explore tradeoffs between control and abstraction.

We evaluated ABAX on five kernels — add, vvadd, GEMV, GEMM, and systolic arrays — and successfully generated synthesizable Verilog through XLS. All designs completed a full ASIC synthesis and place-and-route flow, validating functional correctness and physical realizability. While DSLX initially demonstrated superior area and power efficiency, the integration of memory abstraction functionality removed XLS[CC]'s needs for register clusters. Consequently, XLS[CC] achieved area utilization and power consumption parity with DSLX.

This project highlighted that lowering to an ASIC-capable backend requires explicit handling of memory, control, and temporal behavior, especially when targeting low-level frontends such as DSLX. We also observed that frontend choice strongly shapes hardware outcomes and backend complexity. At the same time, this work underscored the power of Allo's composable programming model: its clean separation of algorithm and hardware customization enabled us to target fundamentally different backends without rewriting kernels.

Acknowledgements

We thank Professor Zhiru Zhang and Niansong Zhang for their guidance throughout the course, as well as Hongzheng Chen for his help with Allo. We also thank the other group working on lowering Allo into XLS (Anjelica, Cynthia, Nikil, Nikhil) for their helpful discussions and feedback during the course of our projects.

References

[1] H. Chen, N. Zhang, S. Xiang, Z. Zeng, M. Dai, and Z. Zhang, *Allo: A Programming Model for Composable Accelerator Design*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '24)*, 2024.

[2] The XLS Developers. *XLS: Accelerated HW Synthesis*. Google Open Source Project. Available at: `https://google.github.io/xls/`, Accessed: December 2025.

[3] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*. Technical documentation. Available at: `https://www.xilinx.com/support/documents/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf`, Accessed: December 2025.

[4] Intel Corporation. *Intel® High Level Synthesis Compiler Pro Edition User Guide*. Technical documentation. Available at: `https://www.intel.com/content/www/us/en/docs/programmable/683456/23-3/pro-edition-user-guide.html`, Accessed: December 2025.

[5] D. Lockhart, G. Zibrat, and C. Batten, *PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research*. In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO-47)*, 2014.

[6] K. Jaic and M. C. Smith, *Enhancing Hardware Design Flows with MyHDL*. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*, pages 28–31, 2015.

[7] The Amaranth Developers. *Amaranth HDL*. Project documentation. Available at: `https://amaranth-lang.org/`, Accessed: December 2025.

[8] The XLS Developers. *DSLX Reference*. Project documentation. Available at: `https://google.github.io/xls/dslx_reference`, Accessed: December 2025.

[9] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-M. Hwu, *PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow*. IEEE Transactions on Computers, vol. 70, no. 12, pages 2015–2028, 2021.

[10] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, *Spatial: A Language and Compiler for Application Accelerators*. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 296–311, 2018.

[11] H. Ye, H. Jun, H. Jeong, S. Neuendorffer, and D. Chen, *ScaleHLS: A Scalable High-Level Synthesis Framework with Multi-Level Transformations and Optimizations*. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, pages 1355–1358, 2022.

[12] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, et al., *T2STensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations*. In *Proceedings of the IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–189, 2019.

[13] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, *HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing*. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 242–251, 2019.

[14] Z. Zhang. *Lecture 14: More Pipelining Resource Sharing*. ECE 6775 High Level Digital Design Automation. `https://www.csl.cornell.edu/courses/ece6775/pdf/lecture14.pdf`

[15] The mflowgen Developers. *FreePDK 45nm for mflowgen*. Project repository. Available at: `https://github.com/mflowgen/freepdk-45nm`, Accessed: December 2025.

Appendix

A Processing Element Code

A.1 Allo PE and Generated DSLX PE

```
def PE_kernel[TyA, TyB, TyC, K: int32](A_in: "TyA[K]", B_in: "TyB[K]", A_out: "TyA[K]", B_out: "TyB[K]") ->
↪  TyC:
    C: TyC = 0
    for k in dsl.grid(K, name="reduction"):
        a: TyA = A_in[k]
        b: TyB = B_in[k]
        C += a * b
        A_out[k] = a
        B_out[k] = b
    return C
```

(a) Allo Implementation of a PE

```
pub proc PE_kernel {
  out0: chan<s32> out;
  mem0__chan: chan<s32> in;
  mem1__chan: chan<s32> in;
  mem2__chan: chan<s32> out;
  mem3__chan: chan<s32> out;

  config(out0: chan<s32> out, mem0__chan: chan<s32> in, mem1__chan: chan<s32> in, mem2__chan: chan<s32> out, mem3__chan:
  ↪  chan<s32> out) { (out0, mem0__chan, mem1__chan, mem2__chan, mem3__chan) }

  init { (0, 0, 0, 0, false) }

  next(state: (s32, s32, s32, s32, bool)) {
    let (acc0, acc1, acc2, index0, busy) = state;
    let (tok0, tmp0) = recv(join(), mem0__chan);
    let tmp1 = tmp0;
    let (tok1, tmp2) = recv(join(), mem1__chan);
    let tmp3 = tmp2;
    let tmp4 = (tmp1 as s64);
    let tmp5 = (tmp3 as s64);
    let tmp6 = (tmp4 * tmp5);
    let tmp7 = (tmp6 as s32);
    let tmp8 = (acc0 + tmp7);
    let tmp9 = tmp8;
    let tok2 = join(tok0, tok1);
    let tok3 = send(tok2, mem2__chan, tmp1);
    let tok4 = send(tok3, mem3__chan, tmp3);
    let tmp10 = if (index0 < s32:2) { tmp9 } else { acc0 };
    let tmp11 = if (index0 < s32:2) { tmp1 } else { acc1 };
    let tmp12 = if (index0 < s32:2) { tmp3 } else { acc2 };
    let tmp13 = if (index0 + 1 >= s32:2) { s32:0 } else { index0 + 1 };
    let tmp14 = index0 + 1 >= s32:2;
    let tok5 = send_if(tok4, out0, tmp14, tmp10);
    let tmp15 = if (tmp14) { 0 } else { tmp10 };
    let tmp16 = if (tmp14) { 0 } else { tmp11 };
    let tmp17 = if (tmp14) { 0 } else { tmp12 };
    let tmp18 = if (tmp14) { s32:0 } else { tmp13 };
    let tmp19 = !tmp14;
    (tmp15, tmp16, tmp17, tmp18, tmp19)
  }
}
```

(b) Generated DSLX Code for a PE

B Systolic Array Code

B.1 Allo Systolic Array

```
def systolic[TyA, TyB, TyC, N: int32, K: int32, M: int32](A: "TyA[N, K]", B: "TyB[K, M]") -> "TyC[N, M]":
    A_fifo: TyA[N, M + 1, K]
    B_fifo: TyB[N + 1, M, K]

    A_state: TyA[N, K] = A
    B_state: TyB[K, M] = B
    C: TyC[N, M] = 0

    for k in range(K, name="data_load"):
        for n in range(N):
            A_fifo[n, 0, k] = A_state[n, k]
        for m in range(M):
            B_fifo[0, m, k] = B_state[k, m]
    for i, j in dsl.grid(N, M, name="PE"):
          C[i, j] = PE_kernel[TyA, TyB, TyC, K](A_fifo[i, j], B_fifo[i, j], A_fifo[i, j + 1], B_fifo[i + 1,
          ↪  j])
    return C
```

Figure 13: Allo Implementation of a Systolic Array

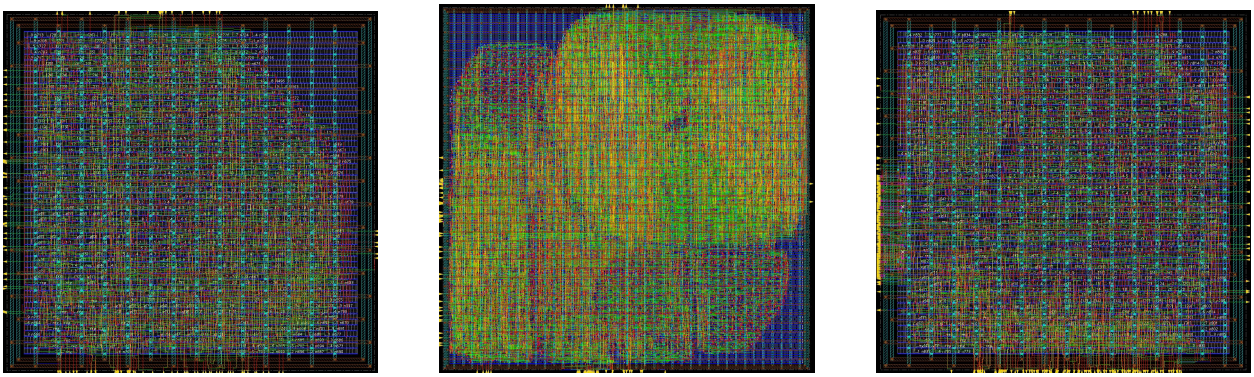
B.2 Generated DSLX Systolic Array

```
pub proc systolic {
  arg0: chan<s32[2][2]> in;
  arg1: chan<s32[2][2]> in;
  out0: chan<s32[2][2]> out;
  from_hor: chan<s32>[3][2] in;
  to_hor: chan<s32>[3][2] out;
  from_vert: chan<s32>[2][3] in;
  to_vert: chan<s32>[2][3] out;
  result_chans_in: chan<s32>[2][2] in;

  config(arg0: chan<s32[2][2]> in, arg1: chan<s32[2][2]> in, out0: chan<s32[2][2]> out) {
    let (to_hor, from_hor) = chan<s32, u32:1>[3][2]("hor_chans");
    let (to_vert, from_vert) = chan<s32, u32:1>[2][3]("vert_chans");
    let (result_chans_out, result_chans_in) = chan<s32, u32:1>[2][2]("result_chans");
    unroll_for! (row, _): (u32, ()) in u32:0..u32:2 {
      unroll_for! (col, _): (u32, ()) in u32:0..u32:2 {
        spawn PE_kernel(result_chans_out[row][col],  // result_out
        from_hor[row][col],          // a_in
        from_vert[row][col],          // b_in
        to_hor[row][col + u32:1],    // a_out
        to_vert[row + u32:1][col]     // b_out
        );
      }(());
    }(());
    (arg0, arg1, out0, from_hor, to_hor, from_vert, to_vert, result_chans_in)
  }

  init { ([[s32:0, s32:0], [s32:0, s32:0]], [[s32:0, s32:0], [s32:0, s32:0]], 0, false) }

  next(state: (s32[2][2], s32[2][2], s32, bool)) {
    let (acc0, acc1, index0, busy) = state;
    let (tok0, a_mat2, b_mat2) = if index0 == s32:0 {
      let (a_tok, a_recv) = recv(token(), arg0);
      let (b_tok, b_recv) = recv(token(), arg1);
      let tok = join(a_tok, b_tok);
      (tok, a_recv, b_recv)
    } else {
      (token(), acc0, acc1)
    };
```

```
    // Send values of A to left side
    let tok1 = {
      let t = tok0;
      if index0 < s32:2 {
        unroll_for!(row,_):(u32, ()) in u32:0..u32:2 {
          let t = send(t, to_hor[row][0], a_mat2[row][(index0 as u32)]);
        }(());
      };
      t
    };

    // Send values of B to top side
    let tok2 = {
      let t = tok0;
      if index0 < s32:2 {
        unroll_for!(col,_):(u32, ()) in u32:0..u32:2 {
          let t = send(t, to_vert[0][col], b_mat2[(index0 as u32)][col]);
        }(());
      };
      t
    };

    // Drop values from right side
    unroll_for!(row,_):(u32, ()) in u32:0..u32:2 {
      recv_non_blocking(token(), from_hor[row][2], s32:0);
    }(());

    // Drop values from bottom side
    unroll_for!(col,_):(u32, ()) in u32:0..u32:2{
      recv_non_blocking(token(), from_vert[2][col], s32:0);
    }(());

    let tok3 = join(tok1, tok2);
    if index0 == s32:3 {
      let (t00, c00) = recv(tok3, result_chans_in[0][0]);
      let (t01, c01) = recv(t00, result_chans_in[0][1]);
      let (t10, c10) = recv(t01, result_chans_in[1][0]);
      let (t11, c11) = recv(t10, result_chans_in[1][1]);
      let tok_final = t11;
      let c: s32[2][2] = [[c00, c01], [c10, c11]];
      let tok_send = send(tok_final, out0, c);

      ([[s32:0, s32:0], [s32:0, s32:0]], [[s32:0, s32:0], [s32:0, s32:0]], 0, false)
    } else {
      (a_mat2, b_mat2, index0 + s32:1, true)
    }
  }
}
```

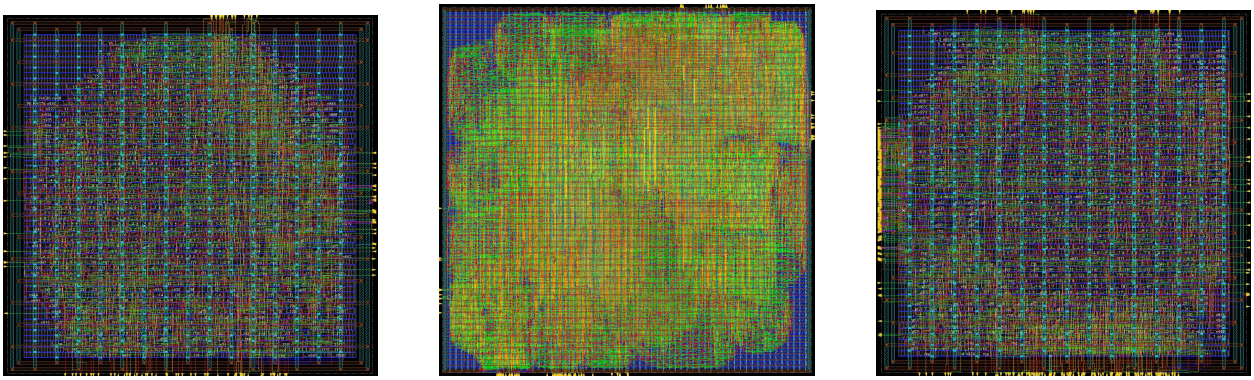
Figure 14: Generated DSLX Code for a Systolic Array

C Place & Route Images



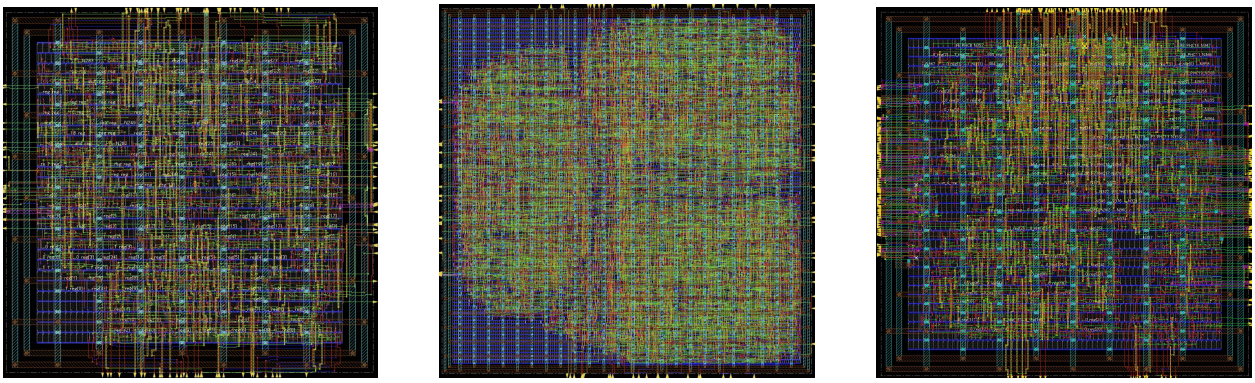
(a) DSLX (b) XLS [CC] (c) XLS [CC] with Memory

Figure 15: `mv.v` Place & Route Images



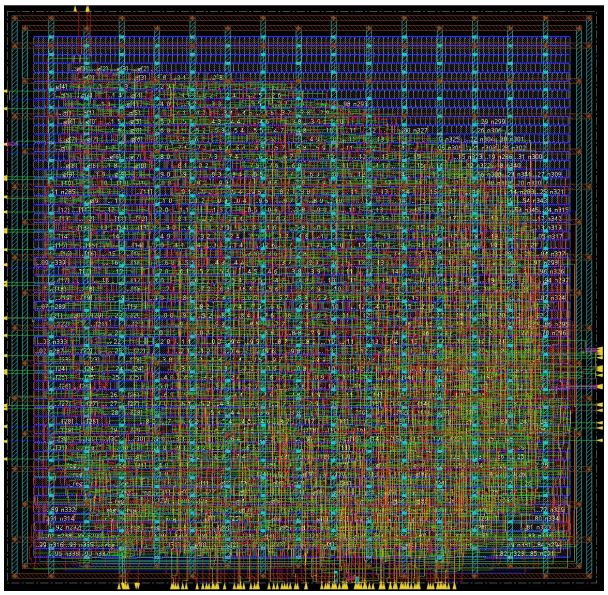
(a) DSLX (b) XLS [CC] (c) XLS [CC] with Memory

Figure 16: `mm.v` Place & Route Images



(a) DSLX (b) XLS [CC] (c) XLS [CC] with Memory

Figure 17: `vvadd.v` Place & Route Images



(a) DSLX

Figure 18: `systolic.v` Place & Route Images