

**RAPID PROTOTYPING FRAMEWORK FOR
HARDWARE-SOFTWARE CO-DESIGN WITH
ADVANCED VECTOR ARCHITECTURES**

by

Ryan Kabrick

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical & Computer Engineering

Spring 2022

© 2022 Ryan Kabrick
All Rights Reserved

**RAPID PROTOTYPING FRAMEWORK FOR
HARDWARE-SOFTWARE CO-DESIGN WITH
ADVANCED VECTOR ARCHITECTURES**

by

Ryan Kabrick

Signed: _____
Xiaoming Li, Ph.D.
Professor in charge of thesis

Approved: _____
Jamie Philips, Ph.D.
Chair of the Department of Electrical & Computer Engineering

Approved: _____
Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____
Louis F. Rossi, Ph.D.
Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

TABLE OF CONTENTS

LIST OF FIGURES	v
ABSTRACT	vi
Chapter	
1 INTRODUCTION	1
2 PREVIOUS WORK	2
3 LANGUAGE OVERVIEW	4
3.1 Instruction Formats	5
3.2 Register Classes	6
3.3 Instruction Definitions	6
3.4 StoneCutter Operations	7
3.5 Pipelines	7
3.6 Intrinsic	9
4 STONECUTTER COMPILER	10
4.1 Signal Map Generator	11
4.2 Pipeliner	12
4.2.1 Sparse Matrix Representation	13
5 COMPLEX TYPES	15
5.1 Fixed-Width Vectors	15
5.2 StoneCutter Vector Syntax	16
5.2.1 Basic Indexing	16
5.2.2 Scalar Expansion	17

5.2.3	Ranges	18
5.3	Variable-Width Vectors	19
5.3.1	Signal Representations	21
6	RESULTS	22
6.1	Basic Tri-Diagonal Solver Implementation	22
6.2	Pipeliner Optimization	23
7	CONCLUSIONS	28
8	FUTURE WORK	29
	BIBLIOGRAPHY	30
	Appendix	
A	VECTOR VISUALIZATIONS	32
B	PERMISSIONS (IEEE)	33
C	PERMISSIONS (TCL)	34

LIST OF FIGURES

4.1	StoneCutter Compiler Architecture	11
4.2	StoneCutter Signal Map Generation	12
4.3	Signal Map Sparse Matrix Representation	13
5.1	Vector and Matrix Indexing Syntax	16
6.1	Simple Pipeline	26
6.2	Vector Pipeline (Refers to Code in Listing 6.2)	27

ABSTRACT

The 21st century has seen an unprecedented level of technological development. As we break away from the days of general-purpose, sequential computing, we have seen tremendous advancements in software infrastructure while the many nuances of hardware development have secluded innovation to the select few familiar with its complexities. Abstracting these low-level implementation details away from developers and allowing for the expeditious prototyping of novel hardware designs could greatly increase the scope of innovation inside the hardware architecture field. It becomes necessary to not only provide the tools necessary for high-level design but to also ensure the capabilities of such tools are modernized and able to model even the most contemporary of architectural innovations.

This work describes the StoneCutter infrastructure, along with its encompassing OpenSoC System Architect suite of tools; a software infrastructure aimed at providing users a frictionless design experience for rapidly prototyping new instruction set architectures. StoneCutter offers opinionated simplicity without sacrificing performance; the advancements detailed in this work include additions to the language frontend and compiler infrastructure, which give the user access to modern architectural constructs including fixed-/variable-width vector and matrix registers as well as a set of highly-optimized, modular implementations of common linear algebra operations in the form of intrinsic functions. The StoneCutter compiler then translates the design to an adjacency matrix representation of its control signals which is automatically pipelined based on its I/O, flow control, and arithmetic operations. Finally, several artifacts allowing for exploration and profiling are generated including an optimized Chisel HDL output along with a custom, LLVM-linked compiler capable of executing binary payloads on the prototyped ISA.

Chapter 1

INTRODUCTION

With the end of Dennard Scaling rapidly approaching [9] and the obsolescence of Moore's Law becoming all but realized, experts and organizations have had no choice but to consider alternate means of taking meaningful strides forward; Perhaps the most notable of these new methods surrounding hardware and software domain specificity [11]. Several novel architecture designs such as Google's TPU (Tensor Processing Unit)[1] for accelerating inference time within neural network algorithms have more than demonstrated the viability of specified hardware designs. However, high capital requirements and the necessity of FPGA prowess have both served to either hinder or entirely halt development teams from effectively prototyping their designs. Consequently, there has been a call to action for developing higher level synthesis and design flows to significantly reduce the initial phase of hardware development [7]. This call has been answered with many solutions, all of which bare the burden of being based on classic circuit analysis methodologies. Modern hardware developers remain encumbered by the latency-riddled processes of verification and synthesis within their design flow.

We have worked to develop OpenSoC System Architect (*SysArch*), a 2nd-generation design flow for hardware development focusing on rapid design, development, and evaluation. The SysArch workflow accomplishes this and more with a collection of tools and libraries enabling frictionless hardware prototyping all at the speed of an optimizing compiler. The goal of the overall infrastructure is to provide users and architects the ability to rapidly develop, prototype, and evaluate hardware design trade-offs with minimal time and effort.

Chapter 2

PREVIOUS WORK

The call to simplify the hardware design process has been far from unanswered. There are several efforts aiming to solve similar problems. One example is the PandA-bambu framework[10]. The goal of the Panda high-level synthesis flow is to permit exploration of hardware-software co-design principles much like StoneCutter and the OpenSoC SysArch ecosystem. There exists several fundamental design decisions that differentiate our two projects. First, both design flows generate optimized HDL implementations of their designs; while Panda generates Verilog (or VHDL) directly, we opted to instead generate the optimized implementation in Chisel HDL [7] as its higher-level paradigm shift away from the myriad of complexities inherent to a language like VHDL. Next, Panda uses C, C++, Fortran, or LLVM IR directly as input granting finer-grain control and consequently requiring the user to place increased consideration on the details of the circuit. Contrasting, the StoneCutter language is a specific HLS gauged at the prototyping of instruction set architectures. This specificity is the key difference between our work and Panda as it almost entirely alleviates the consideration of the underlying circuit from the user.

The Vitis Unified Software Platform by Xilinx seeks to modularize domain-specific accelerator applications through pre-built ip, tooling, and libraries for eventual deployment on their array of Xilinx platforms. This suite aims to provide a portable and reusable infrastructure for implementing accelerators via their Vivado design suite.

Another related project is Intel's (Formerly Altera's) Quartus II FPGA-based design flow for rapidly prototyping digital systems [4] [18]. This project has a strong focus on modularity like SysArch, however it is highly FPGA-centric. Quartus does not generate any artifacts aimed at software development such as a cycle-based simulator

or compiler for executing binary payloads on the design afterward. There is no direct emphasis placed on attracting those new to HW-design or offering CPU-based tooling.

Bluespec is a hardware-design suite offering its own high-level language (Bluespec System Verilog) as well as various tools for specification, synthesis, modeling, and verification [15] [5] [2]. Bluespec has since shifted towards a RISC-V-centric approach, however, for the sake of comparison we will focus on its origins as a general-purpose high-level functional HDL. Much like StoneCutter, Bluespec abstracts low-level implementation details from the architect in the form of an extension to an already existing language (Verilog for BSV & Haskell for Bluespec Classic). Additionally, Bluespec allows users to generate a cycle-based simulator: Bluesim. A key difference lies in the SysArch generated compiler is linked against LLVM while you must use the Bluespec compiler to link against Bluesim.

However, these synthesis techniques often force users to consider the entire circuit design space in order to develop a successful implementation. This lack of design specificity often results in hardware design implementations that are difficult to program, difficult to reuse in future designs and make sub-optimal use of hardware resources.

Chapter 3

LANGUAGE OVERVIEW

StoneCutter source files [14] are constructed in a similar manner as other C-based languages. Global definitions and global mutable variables are defined near the top of the file, prior to their use and function definitions encompass the body of the file. The following sections will detail several first-class citizens inside the StoneCutter language including:

1. Instruction Formats
2. Register Classes
3. Instruction Definitions
4. StoneCutter Operations
5. Pipelines
6. Intrinsic

We provide an example StoneCutter source file in Listing 3.1.

```
1 instformat Arith.if(reg[GPR] ra,  
2             reg[GPR] rb,  
3             reg[GPR] rt,  
4             enc opc, enc func,  
5             imm imm)  
6 instformat ReadCtrl.if(reg[GPR] ra,  
7                       reg[CTRL] rb,  
8                       reg[GPR] rt,  
9                       enc opc, enc func,  
10                      imm imm)  
11 regclass GPR( u64 r0, u64 r1, u64 r2,  
12              u64 r3, u64 r4, u64 r5,  
13              u64 r6, u64 r7, u64 r8 )
```

```

14 regclass CTRL( u64 pc[PC] )
15
16 def add:Arith.if( ra rb rt imm ) {
17     rt = ra + rb
18 }
19
20 def lp:Arith.if( ra rb rt imm ){
21     for( i=ra; ra < imm ){
22         rt = rb << 5
23     }
24 }
25
26 def brc:ReadCtrl.if( ra rb rt imm ){
27     if( ra == rb ){
28         pc = pc + rt
29     } else{
30         pc = pc + 4
31     }
32 }

```

Listing 3.1: Sample StoneCutter Code

3.1 Instruction Formats

In this example, we find two `instformat` definitions. Each `instformat` block defines the fields in an individual instruction format. This format maps back to the actual instruction field encoding utilized by the device during instruction crack and decode. The instruction format blocks define values in the form of globally accessible names for each field. Each field can be one of the following types:

- **Register:** Defined as `reg`
- **Encoding:** Defined as `enc`

- **Immediate:** Defined as `imm`

Each register field defined in the instruction format is also mapped back to a respective register class. In this example, the `ra` register field of the `Arith.if` instruction format maps to the GPR register file.

3.2 Register Classes

After the `instformat` definitions, we have a register class definition which are synonymous with register files. These definitions include specifications on the types of registers encapsulated in the register file. Much in the same manner as the C and C++ languages, the StoneCutter language is strongly typed. Each variable definition in global or local scope must contain an initial type definition. These types are similar to the standard language types such as boolean types (denoted `bool`) and floating point types (`float`). Due to StoneCutter effectively being a high-level HDL, it must also provide support for non byte-aligned types. To accomplish this, StoneCutter utilizes an arbitrary type system to define signed and unsigned integers. This permits users to define *n-width* signed and unsigned integer types.

In our sample StoneCutter source code, we see that each of the `r0-r8` registers are defined as unsigned 64-bit integer registers (`u64`). This example only illustrates the scalar type system inside of StoneCutter however the main scope of this work surrounds the addition of complex types which will be covered in 5.

In addition to defining basic typed registers, StoneCutter also has the ability to assign specific attributes to individual registers. Notice the definition of the `pc` register in the `CTRL` register class. By adding the `PC` attribute to the register, the StoneCutter compiler automatically recognizes that this register falls within the critical path and must be updated with each new instruction.

3.3 Instruction Definitions

Following the global state definitions, we may outline a set of instruction definitions with the `def` keyword. Each instruction definition contains the logic required to

implement an individual instruction with the associated instruction format fields that are utilized as arguments. Within each of these instruction definitions, users have the ability to utilize any global mutable state (instruction fields and registers) as well as define local variables that can be utilized as local, registered state.

The StoneCutter compiler infrastructure will perform a series of optimizations in order to minimize the area impact of local registered state in the critical path of the pipeline by combining similarly utilized local variables across instructions.

3.4 StoneCutter Operations

Given the C-like nature of StoneCutter, the language supports basic styles of arithmetic operators. This includes addition, subtraction, multiplication, modulo and shifting. Each of these operations is bit-width agnostic and can consequently be utilized in conjunction with all combinations of local, global and immediate variables/values.

StoneCutter also supports standard styles of conditional flow control and loop flow control. As shown in the *brc* instruction implementation of Listing 3.1, the syntax for conditional flow control is effectively identical to the base set of C conditional flow control syntax. The loop flow control syntax is also similar to the standard C loop control syntax. StoneCutter supports *for* loops, *while* loops, and *do-while* loops. StoneCutter *for* loop control statements require a base loop state, a conditional test and an optional modifier statement.

3.5 Pipelines

A more advanced feature of the StoneCutter language and compiler is the support for automated hardware pipelining. By defining a pipeline, the user can allocate parts of an instruction implementation to a specific pipeline stage by enclosing them in (optionally named) *pipeline blocks*. In Listing 3.2, we illustrate the most basic pipeline syntax. Similar to register definitions, pipeline definitions can have attributes which govern its behavior in the prototyped system; Currently, the supported

attributes include `in_order`, `out_of_order`, `forward`, `branch_predict`, and `stages_n` (where $n \in \mathbb{Z}$).

```
1 # Basic Pipeline
2 pipeline basic_pipeline()
3
4 # Pipeline with Attributes
5 pipeline fancy_pipeline(out_of_order, branch_predict, ...)
```

Listing 3.2: Basic Pipeline Definition

A basic fused multiply-add (`fma`) implementation is shown in Listing 3.3. Here, we define two *pipeline stages* denoted with the following syntax: `pipe stage:name` where the `name` is optional. We segregate the addition and multiplication to pipe stages `exe_0` and `exe_1`, respectively. In the event a user only specifies explicit pipeline allocations in a subset of instructions, the compiler will reason about the unspecified instructions, grouping similar operations together. It is also worth noting multiple pipelines with different attributes and stages can be used within a single design. The actual pipelining architecture will be discussed in 4.2.

```
1 def fma( RT RA RB ){
2     u64 tmp
3     # pipe stage exe_0 (addition)
4     pipe exe_0{
5         tmp = RA + RB
6     }
7
8     # pipe stage exe_1 (multiplication)
9     pipe exe_1{
10        RT = tmp * RT
11    }
```

Listing 3.3: Pipelined FMA Example**3.6 Intrinsic**s

Much in the same manner as intrinsics for items such as atomic operations and explicit vector operations, StoneCutter also supports a number of intrinsic operations that are utilized to generate optimized, pathological hardware circuits. StoneCutter contains intrinsics for items such as logical complement, sign extension, zero extension, population count, bit extraction and bit insertion. The StoneCutter intrinsics are unique in that the arguments and return values are *typeless*. The implementation of each intrinsic can be utilized to manipulate any of the supported StoneCutter types. The full list of intrinsics can be found in the StoneCutter language guide [14].

Chapter 4

STONECUTTER COMPILER

The StoneCutter compiler infrastructure utilizes major portions of the LLVM [12] compiler infrastructure as the basis for its tooling. We extended the base LLVM functionality in order to support the StoneCutter language syntax (parsing/lexing) as well as several extended passes. Figure 4.1 illustrates the four main components which comprise the StoneCutter compiler. The components are as follows:

1. Parsing and IR Decoration
2. Signal Map Generator
3. Pipeliner
4. Sparse Matrix Representation

The compilation process begins with leveraging the already existing LLVM parsing libraries to generate a standard LLVM abstract syntax tree (AST) from the language input. From this AST, we utilize the standard LLVM path to generate LLVM intermediate representation (IR) code coupled with custom metadata nodes relevant to downstream optimization and analysis passes.

This additional metadata is used to describe the target design's instruction formats, register classes and registers. We utilize a standard set of LLVM optimization passes to perform common optimizations on the generated IR. This set includes but is not limited to operations such as loop invariant code motion, instruction combining, control flow graph simplification and constant propagation. Once the StoneCutter LLVM IR has been optimized and verified to be safe for downstream synthesis, we begin the code generation process.

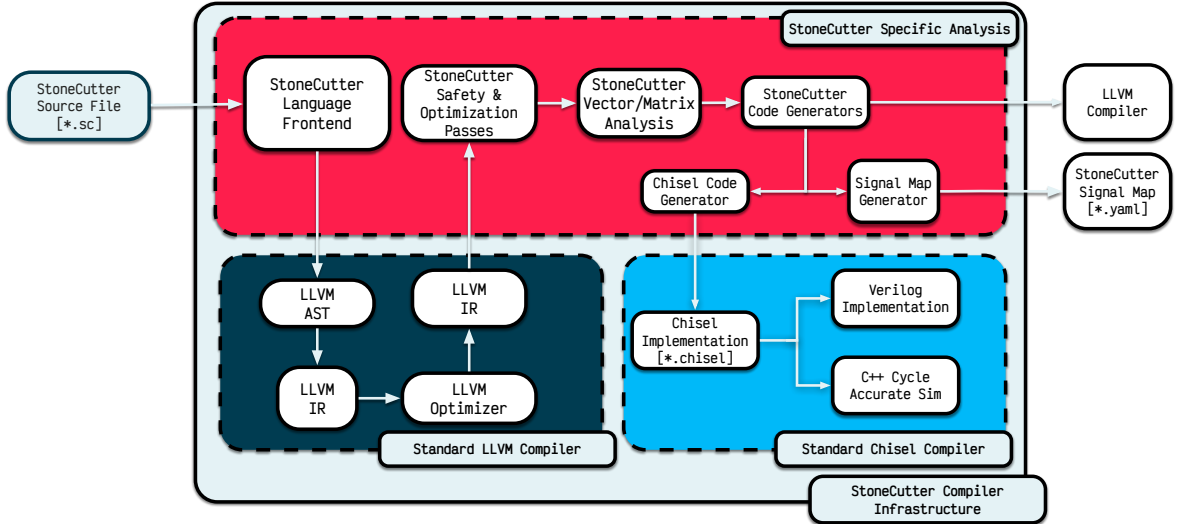


Figure 4.1: StoneCutter Compiler Architecture

4.1 Signal Map Generator

The second phase of the StoneCutter compilation process is a custom pass over the decorated LLVM IR. The signal map generation pass `SCSigMap` receives the incoming LLVM IR and generates an initial representation of traditional hardware signals. These signals include ones such as `REG_READ`, `REG_WRITE`, `MUX`, `ALU_ADD`, and other various control and data signals.

The complexity of this initial pipeline construction is significantly simplified as a consequence of LLVM IR’s standard single assignment form. For example, a load operation on a target with a known register value, simply output a read signal for the associated register file. *Store* operations whose target can be traced to a known register value (using def-use techniques) are converted to register write signals for the associated register file. Immediate operands are output as immediate read operations (from the instruction payload). All arithmetic and logic operations are output as appropriate ALU signals.

In addition to simple load, store and arithmetic operations, the signal map generator also annotates any potential modifications to the program counter as program

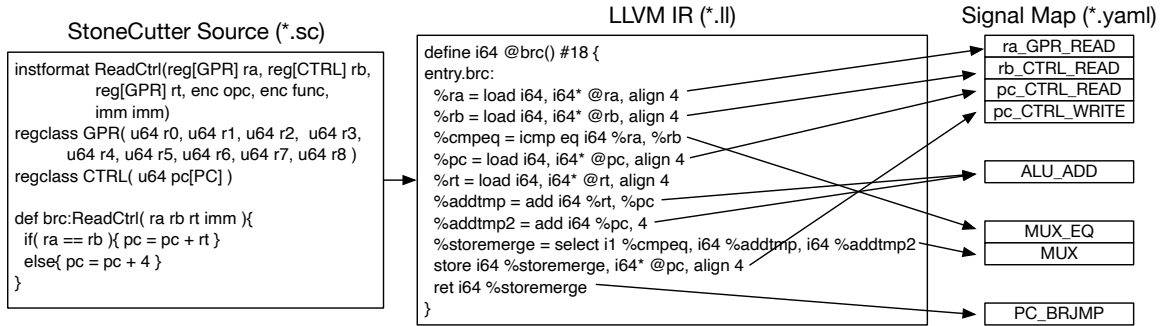


Figure 4.2: StoneCutter Signal Map Generation

counter reads and writes. This is analogous to branch operations in high level code. Further, for complex operations without physical destinations (such as select operations), we output mux or conditional mux signals. The signal map is then output to a human readable Yaml file that can be further ingested into downstream pass mechanisms such as the automated pipeliner. Finally, users can opt to edit the signal map in order to garner additional hardware optimizations downstream where the `SCSigMap` pass fails to sufficiently describe the operation.

4.2 Pipeliner

The signal map file is then fed as input to the automated pipeliner pass (`SCPipeBuilder`). From the signal map, the `SCPipeBuilder` pass constructs a sparse matrix representation of the relationship between individual pipeline stages and the total set of signals found in the design. The pipeliner operates in much the same way as the standard LLVM vectorizer in that it consists of a series of optimization operations which perform operations in a specific order. Once completed, the pipeliner will modify and augment the signal map such that each instruction implementation is properly annotated with the appropriate pipeline stages for the corresponding hardware implementation.

	ra_gpr_READ	re_ctrl_READ	MUX_EQ	pc_ctrl_READ	ALU_ADD	ALU_ADD	pc_ctrl_WRITE	ALU_ADD	pc_ctrl_WRITE	MUX	pc_ctrl_WRITE	PC_BRJUMP
FETCH	0	0	0	0	0	0	0	0	0	0	0	0
REG_READ	1	1	0	1	1	0	0	0	0	0	0	0
ARITH	0	0	1	0	0	1	0	1	0	1	0	0
WRITE_BACK	0	0	0	0	0	0	1	0	1	0	1	1
MEMORY	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.3: Signal Map Sparse Matrix Representation

4.2.1 Sparse Matrix Representation

The pipeliner utilizes an *adjacency matrix* for representing its data. Figure 4.2 shows how the compiler uses source code to generate LLVM IR and subsequently a YAML based signal map. Figure 4.3 shows how that same source code is represented in adjacency matrix form. Each entry in the matrix corresponds the state of a given stage during the corresponding signal. There were no pipeline attributes specified in the source code so a traditional 5-stage pipeline was populated. For example, the add instruction begins with the reading of a value inside a register and consequently has a 1 during the REG_READ stage indicating it is active while this signal is processed. Next, the actual addition (ALU_ADD) is processed, in turn, active during the ARITH stage. There are various operations which can be performed on the pipeline which are expanded on in our other work[13].

Finally, the last phase of the compiler involves code generation to a Chisel [6] representation. This phase utilizes the LLVM IR output and, optionally, the signal map generated from the aforementioned phase to output the target design in Chisel. Once

the design is output in Chisel, we have the ability to utilize the existing Chisel compilation tools to generate a C++ cycle accurate simulator and/or a Verilog representation of the circuit.

Chapter 5

COMPLEX TYPES

The scalar types already included inside of StoneCutter are sufficient for prototyping a vast array of architectural designs. However, many real-world problems map well to linear algebraic constructs such as vectors and matrices. The main contributions of this work surround the addition of *shaped* (ie. multi-dimensional) registers including both *fixed-length* vectors and matrices in addition to *variable-length* vectors.

5.1 Fixed-Width Vectors

Though the implementation of fixed-width vector processing is typically synonymous with SIMD processing, the implementation inside of StoneCutter is orthogonal to the traditional SIMD definition of register state and is instead more in line with *Predicated SIMD* architectures such as ARM’s SVE2[17] Intel’s AVX-512[8]. In this case, each register element is considered to be individually mutable until the user specifies how algorithms are implemented in each instruction, thus defining any notional instruction level parallelism. As we see in Listing 5.1, we define two register classes to denote fixed-width vector registers (*vec*) and matrix registers (comprised of fixed-width vectors) (*mat*), although complex and scalar types may be intermixed. Each vector or matrix register is defined to contain a set of elements, each homogeneous in their type/structure. For example, *v0* defines a vector register with 32 floating point elements. The total size of *v0* is 32×32 or 1024 bits. Similarly, *m2* defines a matrix of 16 rows and 32 columns using unsigned 64 bit elements. All types of StoneCutter datatypes are supported for use as element types inside of vector and matrix constructs.

```
1 regclass fixed_width_vec(  
2     float v0<32>, # 32 x floats
```

$$\text{VEC_NAME}[\text{START_IDX} = 0, \text{STOP_IDX} = 0, \text{STEP_SIZE} = 1]$$

$$\text{MAT_NAME}[\text{START_IDX} = 0, \text{STOP_IDX} = 0, \text{STEP_SIZE} = 1][[\text{START_IDX} = 0, \text{STOP_IDX} = 0, \text{STEP_SIZE} = 1]]$$

Figure 5.1: Vector and Matrix Indexing Syntax

```

3         u64 v1<16>,      # 64 x uint64_t
4         u64 v2<16> )    # 64 x uint16_t
5 regclass fixed_width_mat(
6         float m0<4,4>,  # 4x4 mat
7         u64 m1<16,16>,  # 16x16 mat
8         u64 m2<16,32> ) # 16x32 mat

```

Listing 5.1: Fixed-Width Vector Register Class Definitions

5.2 StoneCutter Vector Syntax

As mentioned above, unlike traditional SIMD architectures, each subset of elements inside a vector or a matrix register are individually addressable. For this reason, it is better to think about the notion of *fixed-width* inside of StoneCutter as referring to the width of the elements themselves. Figure ?? shows the new syntax and Listing 5.2 shows several examples for the new intuitive and efficient syntax.

5.2.1 Basic Indexing

```

1     def vidx( v0 v1 v2 ){
2         # Full Vector Addition
3         v2 = v0 + v1
4
5         # Element Specific Addition
6         v2[-1] = (v0[0] + v0[1]) * v0[-2]
7     }

```

```

8
9  def midx( m0 m1 m2 ){
10     # Full Matrix Addition
11     m2 = m0 + m1
12
13     # Element Specific Addition
14     for( i=0; i<16; i+1){
15         for( j=0; j<16; j+1){
16             m2[i][j] = m0[i][j] + m1[i][j]
17         }
18     }
19 }
20

```

Listing 5.2: Indexing Syntax

The added indexing syntax is meant to reinforce the high level nature of the StoneCutter language. In Listing 5.4, a basic vector addition is implemented just as a scalar addition would be in a traditional SIMD architecture. The next line illustrates the individually mutable characteristics mentioned earlier. StoneCutter now supports indexing into individual elements of vector and matrix registers. This gives users fine grain control when implementing their own instructions. Additionally, support for *Negative Indexing* was added to improve readability as well as offer a method for referencing elements at the end of variable-width constructs 5.3. When the compiler sees a negative number it then derives the current vector length and adds the negative value to get the resulting index. All four examples have accompanying visualizations provided in Appendix A.

5.2.2 Scalar Expansion

```

1  def addfive( v0 v1 ){

```

```
2     u64 tmp = 5
3     v1 = v0 + 5
4 }
5
```

Listing 5.3: Elementwise Vector Addition

The StoneCutter compiler will do its best to reason about the validity of types inside of linear algebra expressions. For example, in Listing 5.3, we are adding a scalar to a vector register. The compiler will warn the user about the mismatch however it will expand the scalar value into a vector of appropriate length to carry out the computation.

5.2.3 Ranges

```
1     # Working with Ranges
2     float SubVec<10> = v0 [4:14]      (a)
3     float SubVec<10> = v0 [:10]      (b)
4     float SubVec<10> = v0 [0:20:2]   (c)
5     float SubVec<10> = v0 [-1:0:-2]  (d)
6
```

Listing 5.4: StoneCutter Vector Ranges

Finally, support for operating on ranges of vectors was added. This support was meant to be pythonic in its implementation. The *colon* denotes the use of a range. In (a), we create a temporary vector of ten elements; namely, the 4th \rightarrow 13th elements of the vector held in `v0`. Next, in (b), we construct a ten-element vector comprised of the first ten elements held inside `v0`. Finally, we create another ten-element wide vector.

This time using the notion of a *stride* to extract only the even-numbered elements of `v0`.

5.3 Variable-Width Vectors

The key functionality found in architectures that support Variable-Width Vectors (ie. *Pure Vectors*[19] is hardware which is quasi-ambiguous to both the number of elements inside a vector register as well as the types of the elements themselves[16][3]. There are, of course, limits which must be imposed by the hardware, namely the maximum bit-width of the register file and the minimum addressable element size within it. It follows that the syntax for variable-width vectors differs slightly from 5.1 to encapsulate the information relevant to generating the proper control and data paths.

```
1 regclass variable_vec( u8 var_vec0<4...128>, u8 my_v1[VL], u8 vstr[
    STR])
2 regclass variable_mat( u8 var_mat0<32, 4...128>, u8 var_mat1<4...128,
    32>, u8 mv1[VL], u8 mstr[STR])
```

Listing 5.5: Variable Vector Register Class Definition

In Listing 5.5 we define two register classes. The first defines a vector register, `var_vec0` as a variable-width vector which can hold between 4 and 128 elements. By qualifying the definition with a `u8` type, we are telling the compiler the maximum bit width of this register is $8 \times 64 = 512$ bits and the minimum addressable element size is 8 bits. From this information, the compiler then derives the maximum possible element width to be:

$$\frac{\text{MaximumBitWidth}}{\text{MinimumNumElements}} = \frac{512}{4} = 128 \text{ bits}$$

We also define a `variable_mat` register class. It is important to note that only one dimension in the matrix can be variable, though it does not matter which one.

Also notice the two new register attributes: `VL` and `STR`. These are both global scalar registers, with their scope being the same as a register qualified with the `PC` attribute.

The `VL` (Vector Length) attribute tells the compiler how many elements, n , to operate on and update in the destination register. During signal map generation, every time a vector of variable-width is encountered, it will create an initial set of signals synonymous with those it would create for a scalar variable of the same element type. Then, it will duplicate the signals n times, adjusting any memory locations based on the width of each element and stride of the vector access if applicable.

Further, a *stride* attribute, denoted `STR`, has been added with similar nuances as the `VL` attribute. While the `VL` register says to apply an operation to VL elements, the `STR` register dictates the step size between elements to be operated on. Bits in memory as well as other vector registers will increase intuitively, according to the following formula:

$$\text{LSB}_{i+1} = \text{MSB}_n + (\text{STR} \times \text{ElemWidth})$$

The difference between a register with the `STR` attribute and the value passed as *stride* is equivalent to the difference between a local and global variable in traditional programming terms. A register with the `STR` attribute will have its value read before every vector access/operation. However, if a *stride* value is specified (ie. Scalar value after the second colon in a dimension) then the value inside of the `STR` register is ignored temporarily but only for the specific vector is used in conjunction with.

Listing 5.6 seeks to illustrate the various possible methods of accessing variable-width vectors. First, we define an 8-element temporary vector of 64-bit elements. Next, we set that temporary vector equal to `var_vec0[-1:-16:-2] + var_vec1`. The access to `var_vec0` specifies a *local stride* of -2 . In words: Starting at the last element, load every other element until you reach the 16th to last element, for a total of 8 elements. In this way the compiler will not insert register reads for the `VL` nor

the STR qualified registers. Contrasting, because `var_vec1` does not specify any additional information about the access, the compiler will see this access as equivalent to `var_vec1[:my_vl:my_str]`.

```
1 regclass variable_vec( u8 var_vec0<4...128>, u8 var_vec1<4...128>,
2   my_vl[VL], my_str[STR])
3
4 def inst1( var_vec0 var_vec1 ){
5   u64 tmpVec<8>
6   tmpVec = var_vec0[-1:-16:-2] + var_vec1
7 }
```

Listing 5.6: Stride Behavior

5.3.1 Signal Representations

In our original implementation of vector and matrix registers, we only considered fixed-width constructs. This meant it was not unreasonable to generate global variables inside the generated LLVM IR for common subsets of members inside of a vector/matrix (ie. elements, row vectors, column vectors, diagonals). With the additional consideration of variable-width constructs, the number of subsets to generate global variables for quickly became unmanageable.

Instead, the signal map infrastructure was augmented to include the concept of repetition. Each signal now has a scalar value dictating the number of times its to be repeated. This value can be an immediate specified in the design or require a read from the VL register. This information is necessary for properly generating the fetch and decode logic required by variable-width vector accesses.

Chapter 6

RESULTS

6.1 Basic Tri-Diagonal Solver Implementation

A relatively complex but common operation within modern linear algebraic workloads is a tridiagonal solver. The following will illustrate the compilation process of such a kernel utilizing all As an example of doing so, Listing 6.1 defines a tri-diagonal matrix multiplication schema commonly utilized in linear algebraic functions and Eigenvalue computations. For this example, we define a new instruction, `tdmm`, that includes two input registers (`ra,rb`) and one output register (`rt`). Using the expanded vector notation described in Section 5.3.1, we utilize the three diagonals (N , $N+1$ and $N-1$) from each of the input matrices to generate a result stored to the same diagonal vectors in the target matrix. Note that this approach requires no manual expansion of individual matrix elements and all internal register addressing logic is handled within the compiler.

```
1 def tdmm(ra rb rt){
2   for(i=0; i<VL; i+1){
3     for(j=0; j<VL; j+1){
4       if( i = j ){
5         rt[i][j] = ra[i][j] * rb[i][j]
6         if( i >= 1 ){
7           rt[i-1][j] = ra[i-1][j] * rb[i-1][j]
8         }
9         if( j >= 1 ){
10          rt[i][j-1] = ra[i][j-1] * rb[i][j-1]
11        }
12      }
}
```

```
13     }
14 }
```

Listing 6.1: Sample Tri-Diagonal Solver Algorithm

6.2 Pipeliner Optimization

```
1 // pipelined tri-diagonal
2 // matrix multiplication
3
4 //define a pipeline
5 pipeline p0(in_order, forward)
6 pipeline p1(in_order, forward)
7 pipeline p2(in_order, forward)
8
9 def tdmn(ra rb rt){
10   for(i=0; i<VL; i+1){
11     for(j=0; j<VL; j+1){
12
13       // Nth diag
14       pipe diagN:p0{
15         if( i = j ){
16           rt[i][j] = ra[i][j] * rb[i][j]
17         }
18       }
19       // Nth-1 diag
20       pipe diagNP1:p1{
21         if( i >= 1 ){
22           rt[i-1][j] = ra[i-1][j] * rb[i-1][j]
23         }
24       }
25
26       // Nth-1 diag
```

```

27     pipe diagNM1:p2{
28         if( j >= 1 ){
29             rt[i][j-1] = ra[i][j-1] * rb[i][j-1]
30         }
31     }
32 }
33 }
34 }

```

Listing 6.2: Optimized Sample Algorithm

Using our example from above, we find that the default control and data flow schedule is aligned in a single pipeline stage. As shown in Figure 6.1, each of the individual vector element operations is performed sequentially with a single stage pipeline. While this may be functionally correct, it will not perform well with sufficiently complex hardware and memory resources. The StoneCutter language specification contains an additional set of keywords and constructs that permit users to direct the output of the automated pipeliner within the compiler. As seen in Listing 6.2, we annotate our original tri-diagonal solver with explicit pipeline definitions. The *pipeline* keyword directs the compiler to create a separate data, control and ALU function for use within the instruction. In this case, we create three individual pipelines for each of the three diagonal vectors. This is analogous to creating a superscalar architecture specific for ML/AI applications. At this point, we encapsulate each of the individual vector multiply operations into unique pipeline instances. The *pipe diagNAME:pipeline* syntax informs the compiler that the operations encapsulated within the associated braces should be placed in the respective named pipeline definition. In this case, we separate the three vector operations into unique pipelines such that they can be executed concurrently.

From this high level syntax, the StoneCutter compiler sets LLVM IR attributes

for each of the contained memory, control flow and arithmetic operations. The StoneCutter pipeline optimizer then constructs an internal, sparse matrix representation of the pipeline control and data flow. Using the prescribed pipelining options, it then splits the memory, register and arithmetic operations into separate stages with a common, scalar instruction crack/decode. Figure 6.2 depicts the resulting, automatically optimized pipeline infrastructure. While the area and power requirements for the optimized pipeline are much larger than the original, simplified pipeline, the performance characteristics are much more attractive to high performance AI/ML applications. Additional details regarding the StoneCutter pipeline syntax and pipeline operations can be found in the StoneCutter specification [14][13].

```
uCode  
def tdm(ra rb rt){  
  for(i=0; i<VL; i+1){  
    for(j=0; j<VL; j+1){  
      if( i = j ){  
        rt[i][j] = ra[i][j] * rb[i][j]  
      }  
      if( i ≥ 1 ){  
        rt[i-1][j] = ra[i-1][j] * rb[i-1][j]  
      }  
      if( j ≥ 1 ){  
        rt[i][j-1] = ra[i][j-1] * rb[i][j-1]  
      }  
    }  
  }  
}
```

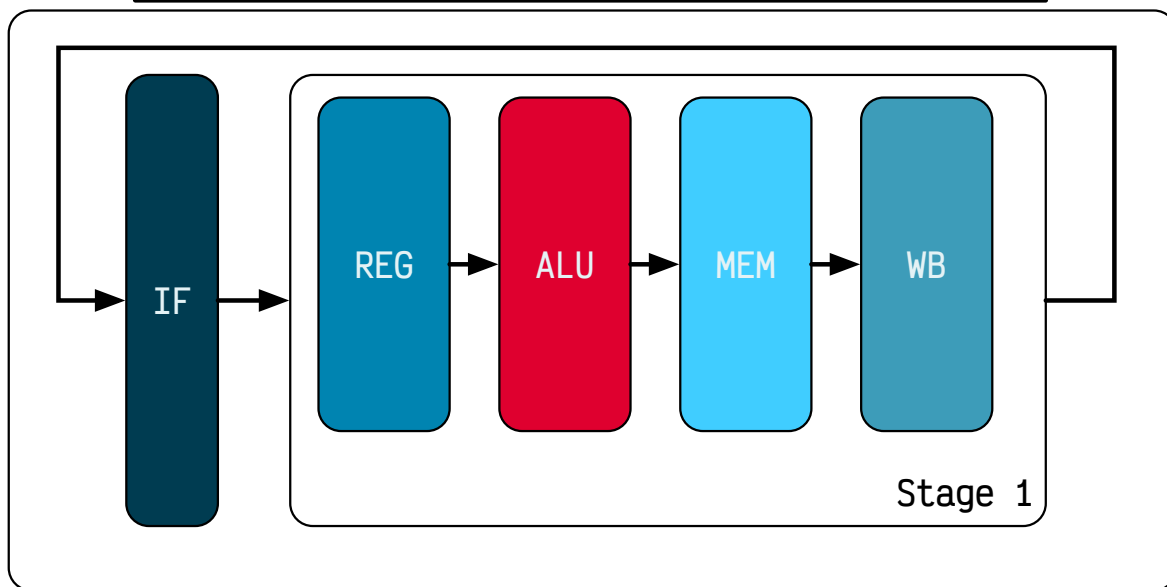


Figure 6.1: Simple Pipeline

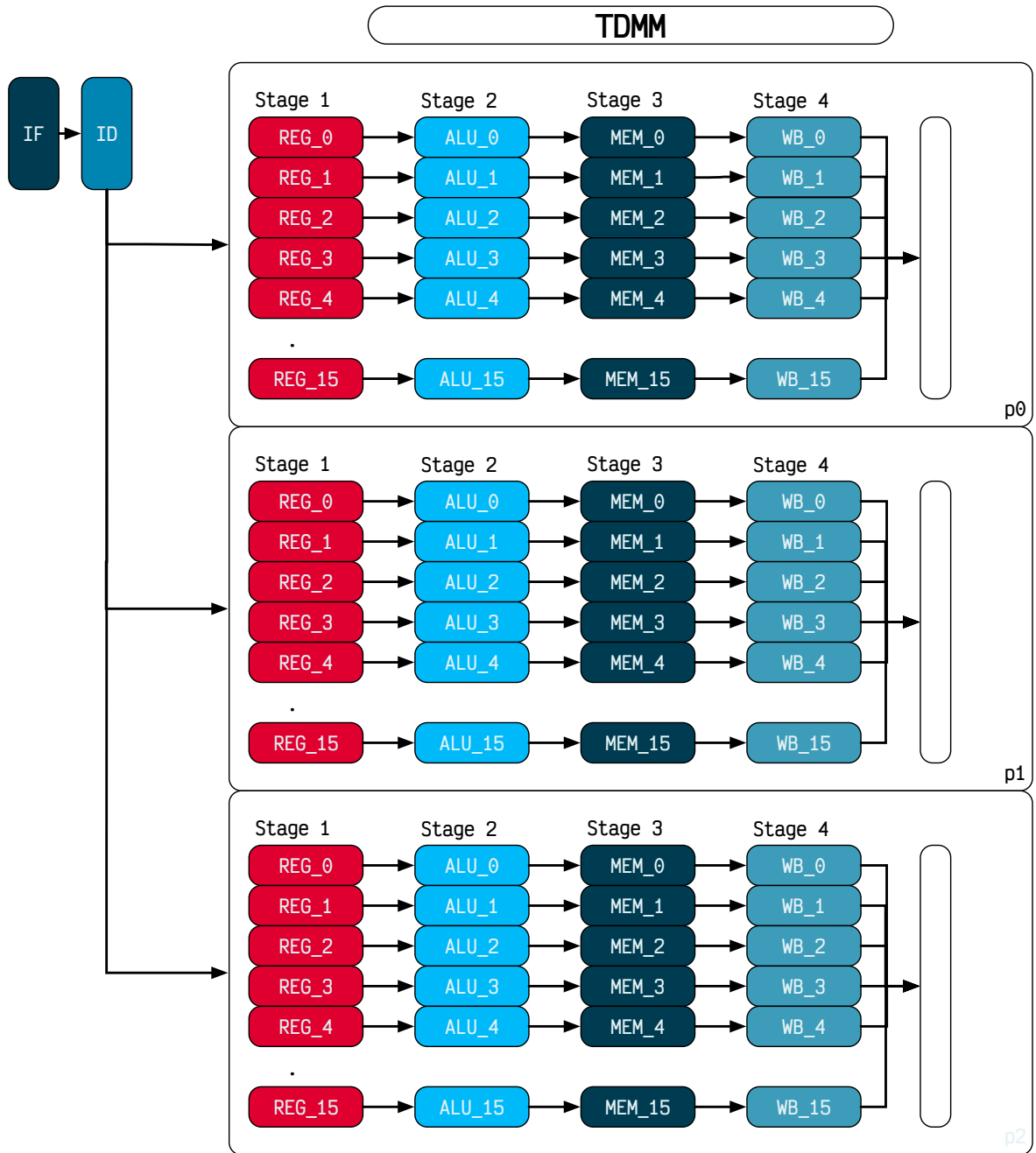


Figure 6.2: Vector Pipeline (Refers to Code in Listing 6.2)

Chapter 7

CONCLUSIONS

The field of hardware innovation has remained an understandably untapped one; Hardware performance scaled with clock frequency and there remained ample performance gains to be had within the software realm. The obselence of frequency-based scaling has catalyzed a need for novel architecture development however the capital requirements of fabrication render reconfigurable hardware as the only viable solution. The software frameworks surrounding these technologies have made leaps and bounds in efficiency but still require developers to possess an incredibly in-depth understanding of the entire circuit they are designing. StoneCutter and its encompassing OpenSOC SysArch ecosystem aim to bridge the gap between hardware and software development. Effectively bringing the luxuries afforded to software development including low-capital, rapidity, and abstraction to the hardware-space for the first time.

This work details the augmentation of the StoneCutter ecosystem such that users can design advanced vector-based architectures custom-built for their application domain. Designers can now implement novel architectures at the same cost and speed as software without sacrificing the architectural features required to solve even the most complex of linear-algebra kernels inside hardware. As a result of this work, developers can now do so in a manner analogous to the most popular programming languages today. Not only can hardware and software be co-designed with the same application in mind, the hardware and software can be created in the same way.

Chapter 8

FUTURE WORK

We have many future plans for augmenting the StoneCutter infrastructure. Within the scope of this work, we plan to augment the linear algebraic capabilities of both the language frontend and compiler itself. We plan to add the notion of front-matter to SC source files. This would allow users to easily specify the code generation behavior when certain events are detected inside of their design. As an example, a user could specify the matrix multiplication method to implement every time an expression contains matrix multiplies is encountered.

Moreover, users could also specify how their ISA handles edge behavior. For example, how to handle type-mismatches should they be deemed possible within the design. An example could be specifying how to handle down-conversions and other type casting behavior. Another example could be specifying how an architecture design should handle out-of-bounds vector element accesses (ie. how to handle index values which could potentially exceed the length of a given vector register).

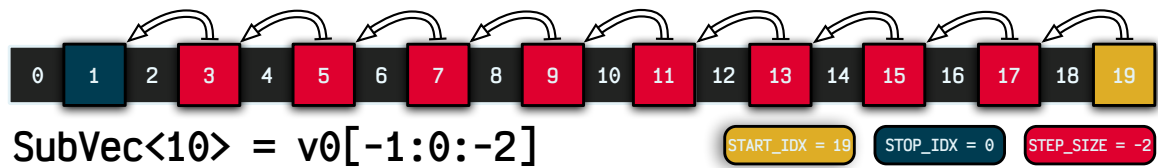
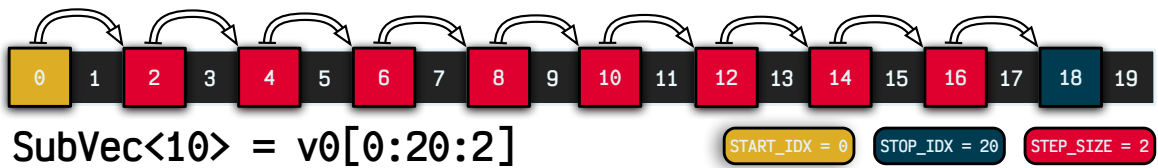
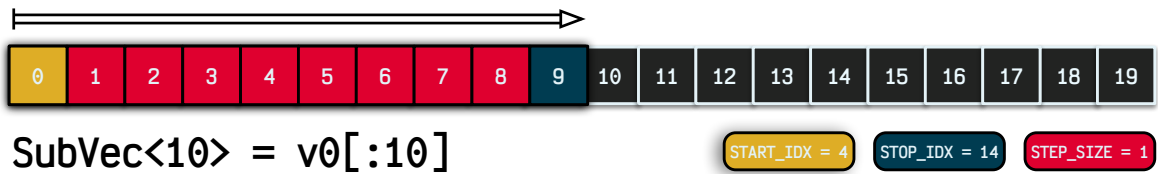
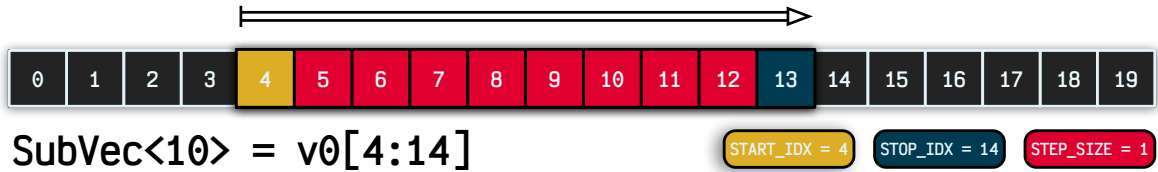
Finally, we are actively developing support for designers to implement deeply pipelined VLIW architectures. Users will be able to define instruction bundles and dictate how various VLIW pipeline stages interact with one another. Of course, the same artifacts including an optimized Chisel implementation, cycle-accurate simulator, and LLVM-based compiler for executing binary payloads will be generated for these designs.

BIBLIOGRAPHY

- [1] Cloud tensor processing units (tpus) — google cloud. <https://cloud.google.com/tpu/docs/tpus>.
- [2] Bluespec system verilog: Efficient, correct rtl from high level specifications. In *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE '04*, pages 69–70, Washington, DC, USA, 2004. IEEE Computer Society.
- [3] Risc-v vector specification (frozen for review). Technical report, 2021.
- [4] Intel Altera. Quartus prime standard edition handbook volume 1: design and synthesis, 2017.
- [5] Arvind. Bluespec and haskell. In *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-specific Languages, FPCDSL '13*, pages 1–2, New York, NY, USA, 2013. ACM.
- [6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference, DAC '12*, pages 1216–1225, New York, NY, USA, 2012. ACM.
- [7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.
- [8] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual. Technical report, 1998.
- [9] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet’s with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.
- [10] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330. IEEE, Dec 2021.

- [11] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, jan 2019.
- [12] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.
- [13] John Leidel, Ryan Kabrick, and David Donofrio. Toward an automated hardware pipelining llvm pass infrastructure. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 39–49, 2021.
- [14] John D. Leidel and Ryan Kabrick. Stonecutter language specification version 0.7. Stonecutter language specification, Tactical Computing Laboratories, 02 2022. <https://github.com/opensocsysarch/StoneCutterLanguageSpec/tree/0.7>.
- [15] Rishiyur S Nikhil. What is bluespec? *ACM-SIGDA-Newsletter*, 39(1):1–1, 2009.
- [16] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.
- [17] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The arm scalable vector extension. *IEEE micro*, 37(2):26–39, 2017.
- [18] Intel Quartus Prime Standard Edition User. Guide: Getting started.
- [19] C Xavier and Sundararaja S Iyengar. *Introduction to parallel algorithms*, volume 1. John Wiley & Sons, 1998.

Appendix A
VECTOR VISUALIZATIONS



Appendix B

PERMISSIONS (IEEE)

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

Appendix C
PERMISSIONS (TCL)

Tactical Computing Laboratories, LLC



55 CR 462 • Muenster, Texas 76252 • Phone: +1.469.712.6601 • Fax: +1.214.291.5296
E-Mail: jleidel@tactcomplabs.com Web: www.tactcomplabs.com

Distinguished Committee Members:

I am listed as first author on the following paper: J. Leidel, R. Kabrick and D. Donofrio, "Toward an Automated Hardware Pipelining LLVM Pass Infrastructure," 2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), 2021, pp. 39-49, doi: 10.1109/LLVMHPC54804.2021.00010. Ryan Kabrick is listed as a co-author on this publication due to his significant contribution to the work described therein as well as the creation of the document itself. Therefore, this correspondence serves as a supplement to the IEEE RightsLink granting Mr. Kabrick permission to utilize portions of this paper within the scope of his Master's thesis for the University of Delaware. The intellectual property and associated source code for the work described therein belongs to Tactical Computing Laboratories LLC. However, Mr. Kabrick is free to use any text, figures, and experimental data from this paper within the scope of his Master's Thesis.

Sincerely,

A handwritten signature in black ink, appearing to read 'John D. Leidel', written in a cursive style.

Dr. John D. Leidel
Chief Scientist
Tactical Computing Laboratories
+1.214.578.8510