StoneCutter: A Very High Level Instruction Set Design Language

John D. Leidel
jleidel@tactcomplabs.com
Tactical Computing Laboratories
Muenster, Texas

David Donofrio
ddonofrio@tactcomplabs.com
Tactical Computing Laboratories
San Francisco, California

Frank Conlon
fconlon@tactcomplabs.com
Tactical Computing Laboratories
Muenster, Texas

ABSTRACT

As the density and capability of reconfigurable computing using FPGAs continues to increase and access to large scale ASIC integration continues to increase, research activities associated with high level synthesis flows have expanded at a similar rate. The goal of these research efforts is to reduce the time and effort required to construct and deploy application-specific architectures. 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.

In this work we introduce the StoneCutter instruction set design language and tool infrastructure. StoneCutter provides a familiar, C-like language construct by which to develop the implementation for individual, programmable instructions. The LLVM-based StoneCutter compiler performs individual instruction and whole-ISA optimizations in order to generate a high performance, Chisel HDL representation of the target design. Utilizing the existing Chisel tools, users can also generate C++ cycle accurate simulation models as well as Verilog representations of the target design. As a result, StoneCutter provides a very rapid design environment for development and experimentation.

CCS CONCEPTS

• **Hardware** → **Hardware description languages and compilation**; *Hardware validation*; • **Computer systems organization** → **Reconfigurable computing**; • **Computing methodologies** → *Simulation tools*.

KEYWORDS

synthesis, instruction set, LLVM, Chisel, compiler

ACM Reference Format:

John D. Leidel, David Donofrio, and Frank Conlon. 2020. StoneCutter: A Very High Level Instruction Set Design Language. In *17th ACM International Conference on Computing Frontiers (CF '20)*, May 11–13, 2020, Catania, Italy. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3387902.3394029>

1 INTRODUCTION

As the density of logic and in-situ functionality of FPGAs continue to increase, an adjacent resurgence in research associated with high level synthesis (HLS) and hardware design languages (HDL) has

also occurred. These design and language flows provide higher performance and more flexible design flows than traditional, raw RTL development. However, despite the advancements made in these tools, there still remain a number of significant gaps to rapidly developing functional, programmable reconfigurable computing payloads for the latest generation of FPGAs.

There have been a number of recent projects to construct large-scale, domain-specific language approaches to HLS. The Bluespec infrastructure [1] is an excellent example of repurposing an existing language (Haskell) into a commercially-supported design flow. However, Bluespec suffers from the same follies as many traditional HLS flows as it requires users to consider the entire target design as opposed to the implementation of individual instructions. The Panda/Bambu infrastructure [5] is a similar approach to StoneCutter as it utilizes the GCC compiler infrastructure as the basis for optimization and code generation. However, Panda also suffers a similar folly in that it requires the user to consider the entire design optimization space. Finally, the LegUp infrastructure [4] is an HLS flow that utilizes a C-syntax to construct extensions to an existing, FPGA-based MIPS soft core. This approach is likely the most similar to the StoneCutter infrastructure as it permits the user to focus on individual design elements. However, it lacks flexibility in that the approach is directly coupled to a MIPS soft core.

We assert that alleviating the need to generate any generic hardware element and instead focusing on rapid generation and exploration of instruction set architectures enables StoneCutter to produce more efficient designs than traditional HLS flows. The StoneCutter language is a C-like language construct that includes global mutable elements (global variables), local mutable elements (local variables), loop flow control, conditional flow control, intrinsic operations and all forms of standard numeric and conditional operations. The StoneCutter compiler is implemented as a series of extensions to the LLVM [6] compiler infrastructure. While the StoneCutter compiler utilizes a large portion of the standard LLVM features, such as the abstract syntax tree (AST), intermediate representation (IR) and optimization passes, we extend the base LLVM frontend and backend. This includes a custom language frontend to handle the parsing/lexing of the StoneCutter language, a number of extended optimization/safety checking passes and a series of custom code generation mechanisms for generating design signal maps and Chisel HDL from the StoneCutter input.

As the StoneCutter language and associated infrastructure is in active development, this work will explore the overall design goals, language syntax and infrastructure tooling currently utilized to implement StoneCutter.

1.1 StoneCutter Implementation Layout

StoneCutter source files [7] are constructed in a similar manner as other C-based languages. Global definitions and global mutable

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CF '20, May 11–13, 2020, Catania, Italy

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7956-4/20/05.

<https://doi.org/10.1145/3387902.3394029>

variables must be defined near the top of the file, prior to their use. We provide an example StoneCutter source file in Listing 1. In this example, we find two *instformat* definitions. Each *instformat* block defines the fields in an individual instruction format. Next, *regclass* blocks define a register class (register file) and the constituent registers contained therein. 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. We include more details on each syntactical element in the following sections.

Listing 1: Sample StoneCutter Code

```
instformat Arith(reg[GPR] ra, reg[GPR] rb,
               reg[GPR] rt, enc opc, enc func,
               imm imm)
instformat ReadCtrl(reg[GPR] ra, reg[CTRL] rb,
                  reg[GPR] rt, enc opc, enc func,
                  imm imm)
regclass GPR( u64 r0, u64 r1, u64 r2, u64 r3,
             u64 r4, u64 r5, u64 r6, u64 r7, u64 r8 )
regclass CTRL( u64 pc[PC] )

def add:Arith( ra rb rt imm ) {
    rt = ra + rb
}
def lp:Arith( ra rb rt imm ){
    for( i=ra; i < imm; i+1 ){
        rt = rb << 5
    }
}
def brc:ReadCtrl( ra rb rt imm ){
    if( ra == rb ){ pc = pc + rt }
    else{ pc = pc + 4 }
}
```

1.2 Type System

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 *bool* for boolean types and *float* for single precision floating point. However, unlike traditional software-oriented languages, StoneCutter is a very high level hardware logic description language. This implies that non byte-aligned types must also be supported. In which case, StoneCutter utilizes an arbitrary type system to define signed and unsigned integers. In this manner, users may specify any width signed and unsigned integer. StoneCutter attempts to handle any necessary type conversions where necessary. The full list of supported types is listed in Table 1.

1.3 Mutable State

The StoneCutter language contains both local and global mutable state. Local mutable state can be defined within an instruction definition using the standard *TYPE VARNAME* syntax with an optional initializer. The StoneCutter compiler optimization phases will

Table 1: StoneCutter Type System

Type	Description
uN	Unsigned integer of N bits
sN	Signed integer of N bits
bool	Boolean value (analogous to u1)
float	Single precision floating point value
double	Double precision floating point value

Table 2: StoneCutter Instruction Format Field Types

Syntax	Description
reg[CLASS] name	Register encoding field indexed in CLASS
enc name	Binary encoding field (opcode, etc)
imm name	Immediate value

make an attempt to optimize the use of these local variables such that the associated hardware state may be shared across multiple instructions as an arithmetic machine state register.

Global mutable state is, however, somewhat different in the StoneCutter language. There are two top-level constructs that generate global state. First, *instformat* construct defines the set of fields from the given instruction format. Each field entry is defined using a target type (Table 2) and a field name. Each of the fields for each instruction format definition is automatically and implicitly defined as a global variable in the LLVM IR. As a result, users may directly utilize these variables in the instruction logic implementation. When reading or writing a register field, the StoneCutter compiler automatically converts the read request into a read request for the target register file at the index provided by the instruction payload. Reading from an encoding or immediate field will provide the raw bits of the field in the instruction payload. Writing to immediate or encoding field is not permitted and will generate a compiler error. In addition to providing fundamental access to these values, the StoneCutter compiler will tag these global variables in the LLVM IR with additional attributes such that analysis and optimization passes may interrogate any access permissions and parameters.

In addition to the instruction definition global state, StoneCutter also raises the *regclass* definitions as global mutable state. As noted in Listing 1, register classes contain a set of registers, each with a specific type. Each of the registers is defined as a unique global variable in the LLVM IR per the prescribed type. As a result, each register can be read and written from within the instruction definition logic. Further, registers may contain an additional attribute block to define additional parameters that may affect downstream optimization. In our example, we define a register with a *PC* attribute. This induces a StoneCutter optimization that automatically increments the associated register as a program counter with each subsequent instruction providing that the program counter is not manually manipulated (via branch instructions).

Figure 1: StoneCutter Compiler Architecture

1.4 Instruction Definitions

StoneCutter instruction definitions are similar to C-style functions. Each instruction definition is preceded by a *def* statement. Each *def* statement is followed by the instruction name and the instruction format for which the instruction is defined. This allows the StoneCutter compiler to associate the set of permissible control and data paths for each instruction as prescribed by the instruction format.

In addition to the basic instruction name and instruction format, each instruction definition may contain a number of arguments. The instruction arguments can be any global, mutable state variables as described above. However, utilizing only the most necessary state variables will assist to minimize the data path utilized by each of the target instructions.

1.5 Supported Operations

The StoneCutter language is similar other C-like languages in that it supports all basic styles of arithmetic operators. This includes addition, subtraction, multiplication, modulo, shifting and logicals. Each of these operations can be utilized in conjunction with all combinations of local, global and immediate variables/values.

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 [7].

In addition to the fundamental arithmetic, logical and intrinsic operations, StoneCutter also supports standard styles of conditional flow control and loop flow control. As shown in the *brc* instruction implementation of Listing 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.

2 STONECUTTER TOOLING

2.1 Tool Infrastructure

The StoneCutter compiler infrastructure utilizes major portions of the LLVM [6] compiler infrastructure as the basis for its tooling. We extend the base LLVM functionality in order to support the StoneCutter language syntax (parsing/lexing), hardware-specific optimization passes and hardware-specific code generation facilities.

As we see in Figure 1, the StoneCutter compiler utilizes a new language frontend based upon the 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 library mechanisms to generate LLVM intermediate representation (IR) code. This LLVM IR is fully compliant with standard LLVM IR with no hardware-specific extensions. However, the StoneCutter frontend annotates the LLVM IR with additional metadata and variable attributes. This additional data is utilized to describe the target design’s instruction formats, register classes and registers. This allows the extended LLVM passes to utilize this information in downstream optimization and code generation. We then utilize a standard set of LLVM optimization passes to perform common optimizations on the generated IR. These passes are provided by the base LLVM infrastructure. These include operations such as loop invariant code motion, instruction combining, control flow graph simplification and constant propagation.

At this stage of the compilation, we begin executing StoneCutter-specific safety and optimization passes. We describe this extended LLVM passes further in Section 2.2. Once the StoneCutter LLVM IR has been optimized and verified to be safe for downstream synthesis, we begin the code generation process.

There are two optional phases of StoneCutter code generation. The first phase is the signal map generator. This phase utilizes the StoneCutter IR and generates a list of microcode signals and any required internal temporary state for each instruction. This information is output in a YAML [3] file for consumption by other StoneCutter code generation facilities, debugging or external tools. The goal of the signal map is to provide a holistic view of the potential operations and their constituent signals such that external

Figure 2: Sample Signal Map Flow

IP, code generation or high-level synthesis facilities can be utilized. The signal map code generation phase makes use of the standard LLVM *def-use* mechanisms in order to determine which static single assignment variables are rooted in register values or temporary storage. If the value is rooted in a register value, then a register read or write signal is generated. If the value is rooted in a temporary value, then the signal map pass generates a temporary registered storage location. These storage locations are condensed to a minimal set via an iterative process of whole program optimization. In the same manner, all arithmetic operations are translated to their equivalent signals and conditional control flow operations are translated to mux and/or branch signals. We provide an example of translating an instruction definition using conditional control flow to the appropriate signal map in Figure 2. Notice that incoming instruction arguments (register fields) are translated to equivalent register reads, the addition operations are translated to arithmetic signals for the ALU and the store operations are translated to register writes. Also note that the associated instruction manually increments the program counter (PC). As a result, the signal map mechanisms recognize this and instantiate an explicit PC_BRJMP signal to force a signal branch as opposed to an implied program counter increment.

The second code generation phase is the Chisel [2] code generator. 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 synthesizable Verilog representation of the circuit.

2.2 StoneCutter Passes

In addition to the standard set of LLVM optimization passes, we also utilize a set of extended LLVM passes that are specific to the StoneCutter infrastructure in order to ensure that the design input is both safe for downstream synthesis and optimal in terms of the prescribed data and control paths. First, the *FieldIO*, *InstArg* and *InstFormat* passes examines each instruction definition and determines whether the instruction data path correctly utilizes the prescribed fields per the associated instruction format and the associated register classes. Finally, the *IOWarn* pass examines all I/O operations within a target instruction definition and determines whether they adhere to the standard data path. For example, if an instruction is defined to utilize three general purpose registers then utilizing a register from an adjacent register file will be flagged

by the *IOWarn* pass. As we continue to enhance the StoneCutter infrastructure, we will continue to expand the set of StoneCutter micro compiler passes.

3 FUTURE WORK

The StoneCutter language and associated tooling infrastructure is still in active development. We plan to augment the existing syntax and infrastructure with the ability to make specific micro architectural optimizations such as supporting the explicit definition of individual pipeline stages and better support for vector and automated floating point operations.

ACKNOWLEDGMENTS

Research reported in this publication was supported by the U.S. Department of Defense under Contract FA8075–14–D–0002.

REFERENCES

- [1] Arvind. 2013. Bluespec and Haskell. In *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-specific Languages* (Boston, Massachusetts, USA) (FPCDSL '13). ACM, New York, NY, USA, 1–2. <https://doi.org/10.1145/2505351.2508149>
- [2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzyniek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) (DAC '12). ACM, New York, NY, USA, 1216–1225. <https://doi.org/10.1145/2228360.2228584>
- [3] Oren Ben-Kiki, Clark Evans, and Ingy. 2009. YAML Ain't a Markup Language (YAML) version 1.2. (10 2009). <https://yaml.org/spec/1.2/spec.html>
- [4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) (FPGA '11). ACM, New York, NY, USA, 33–36. <https://doi.org/10.1145/1950413.1950423>
- [5] Vito Giovanni Castellana, Marco Minutoli, Antonino Tumeo, Marco Lattuada, Pietro Fezzardi, and Fabrizio Ferrandi. 2019. Software Defined Architectures for Data Analytics. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference* (Tokyo, Japan) (ASPAC '19). ACM, New York, NY, USA, 711–718. <https://doi.org/10.1145/3287624.3288754>
- [6] Chris Lattner and Vikram Adve. 2004. 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* (Palo Alto, California) (CGO '04). IEEE Computer Society, Washington, DC, USA, 75–. <http://dl.acm.org/citation.cfm?id=977395.977673>
- [7] John D. Leidel. 2019. *StoneCutter Language Specification Version 0.5*. StoneCutter Language Specification. Tactical Computing Laboratories. <http://www.systemarchitect.tech/releases/StoneCutterLang/stonecutterspec.0.5.pdf>