StoneCutter Language Specification

Tactical Computing Laboratories, LLC

Date: February 21, 2019
Revision: 0.2
Authors: Tactical Computing Labs
contact@tactcomplabs.com

DRAFT

Contents

1 Overview **6**

2 StoneCutter Language **7**
2.1 Overview . . . 7
2.2 Syntactical Notes . . . 8
2.3 Comments . . . 9
2.4 Datatypes . . . 10
2.5 Instruction Format Definitions . . . 11
2.6 Register Class Definitions . . . 12
2.7 Instruction Prototypes . . . 13
2.8 Variable Definitions . . . 15
2.9 Arithmetic Operations . . . 16
2.10 Conditional Operations . . . 17
2.10.1 Boolean Operators . . . 17
2.10.2 Flow Control . . . 18
2.11 Loop Operations . . . 19
2.11.1 For Loops . . . 19
2.11.2 While Loops . . . 20
2.11.3 Do While Loops . . . 21

3 Intrinsic Functions **22**
3.1 Overview . . . 22
3.2 Arithmetic Intrinsics . . . 23
3.2.1 CLZ . . . 23
3.2.2 COMPRESS . . . 24
3.2.3 COMPRESSM . . . 25
3.2.4 CTZ . . . 26
3.2.5 DOZ . . . 27
3.2.6 EXTRACTS . . . 28
3.2.7 EXTRACTZ . . . 29
3.2.8 INSERTS . . . 30
3.2.9 INSERTZ . . . 31
3.2.10 MAJ . . . 32
3.2.11 MAX . . . 33
3.2.12 MERGE . . . 34
3.2.13 MIN . . . 35
3.2.14 NOT . . . 36
3.2.15 POPCOUNT . . . 37
3.2.16 REVERSE . . . 38
3.2.17 ROTL . . . 39
3.2.18 ROTR . . . 40
3.2.19 SEXT . . . 41
3.2.20 ZEXT . . . 42
3.3 Memory Intrinsics . . . 43
3.3.1 LOAD . . . 43
3.3.2 STORE . . . 44
3.3.3 LOADELEM . . . 45
3.3.4 STOREELEM . . . 46

4 Appendix A: Sample StoneCutter Implementation **47**

5 Appendix B: Intrinsic Function Table **53**

DRAFT

List of Figures

1 StoneCutter Architecture . . . 7

Listings

1 StoneCutter File Structure . . . 8
2 StoneCutter Comments . . . 9
3 Arbitrary Width Datatypes . . . 10
4 Instruction Format Definition Syntax . . . 11
5 Sample Instruction Format Definition . . . 11
6 Register Class Definition Syntax . . . 12
7 Sample Register Class Definition . . . 12
8 Instruction Prototype Format . . . 13
9 Instruction Prototype Format . . . 14
10 Variable Definitions . . . 15
11 Sample Variable Definitions . . . 15
12 Flow Control Syntax . . . 18
13 Sample If-Else Syntax . . . 18
14 For Loop Syntax . . . 19
15 Sample For Loop Syntax . . . 19
16 While Loop Syntax . . . 20
17 Sample While Loop Syntax . . . 20
18 Do While Loop Syntax . . . 21
19 Sample Do While Loop Syntax . . . 21
20 Intrinsic Syntax . . . 22
21 Sample StoneCutter . . . 47

List of Tables

2 StoneCutter Datatypes . . . 10
3 StoneCutter Instruction Format Field Types . . . 11
4 StoneCutter Arithmetic Operations . . . 16
5 StoneCutter Boolean Operations . . . 17
6 StoneCutter Comparison Operations . . . 19
7 CLZ Intrinsic . . . 23
8 COMPRESS Intrinsic . . . 24
9 COMPRESSM Intrinsic . . . 25
10 CTZ Intrinsic . . . 26
11 DOZ Intrinsic . . . 27
12 EXTRACTS Intrinsic . . . 28
13 EXTRACTZ Intrinsic . . . 29
14 INSERTS Intrinsic . . . 30
15 INSERTZ Intrinsic . . . 31
16 MAJ Intrinsic . . . 32
17 MAX Intrinsic . . . 33
18 MERGE Intrinsic . . . 34
19 MIN Intrinsic . . . 35
20 NOT Intrinsic . . . 36
21 POPCOUNT Intrinsic . . . 37
22 REVERSE Intrinsic . . . 38

DRAFT

23 ROTL Intrinsic 39
24 ROTR Intrinsic 40
25 SEXT Intrinsic 41
26 ZEXT Intrinsic 42
27 LOAD Intrinsic 43
28 STORE Intrinsic 44
29 LOADELEM Intrinsic 45
30 STOREELEM Intrinsic 46
31 StoneCutter Intrinsics 53

DRAFT

Revision History

Revision	Date	Author(s)	Description
0.1	02.15.2019	JLeidel	Initial public release
0.2	02.21.2019	JLeidel	Adding syntax support for instruction to instruction format association

DRAFT

1 Overview

The StoneCutter language is utilized to develop the implementation of a single instruction within the System Architect design workflow. The StoneCutter language abstracts a large portion of traditional high level design languages such that the user may focus on the implementation details of a single instruction rather than the connectivity to the remainder of the infrastructure. The core features of the StoneCutter language infrastructure are noted as follows:

- **Compiled Language**: Unlike other HDL approaches, the StoneCutter language is, in fact, compiled. The core StoneCutter compiler infrastructure makes use of the LLVM compiler infrastructure for lexing, parsing, optimization and code generation. As a result, we have the ability to initiate traditional optimizing compiler passes, syntax tests and lexical analysis as other compiled languages. Further, we have the ability to craft tooling that integrates with the StoneCutter language in the same manner as traditional compiled languages.

- **Integration with CoreGen IR**: The StoneCutter language and associated tooling is architected in a manner that permits integration with the CoreGen intermediate representation (IR) [1]. This is done in two ways. First, users have the ability to write StoneCutter instruction definitions inline within the CoreGen IR. These `Impl` definitions are directly embedded within the overarching design. Second, the CoreGen IR can be utilized to verify the I/O architecture and instruction format for each StoneCutter instruction definition. This ensures that the prescribed instruction format in the CoreGen design is *verified* to be functionally correct prior to utilizing downstream synthesis tools.

- **C-Like Syntax**: Unlike other HDL approaches, the StoneCutter language utilizes a familiar syntactical structure that is designed to mimic traditional C procedural methods. Each instruction definition is contained within an effective function body with incoming arguments (registers). Arithmetic, boolean operations, conditional operations and loop structures all mimic traditional procedural C syntax. In this manner, the learning curve required to be productive with StoneCutter is minimized.

- **Support for Intrinsics**: Much in the same manner as traditional procedural languages such as C, StoneCutter has support for inline intrinsic operations. Intrinsic operations are designed to support optimized circuits for pathological operations such as sign/zero extension, multi-input operations (majority vote, etc) and special arithmetic operations. As the StoneCutter language continues to develop, we plan to augment the list of supported intrinsics.

The remainder of this language specification is organized as follows. Section 2 introduces the StoneCutter language and the associated syntax. Section 3 provides details associated with each of the currently supported intrinsics. Section 4 provides a sample set of StoneCutter instruction definitions. Section 5 provides a consolidated list of StoneCutter intrinsics.

DRAFT

2 StoneCutter Language

2.1 Overview

As mentioned above, the StoneCutter language is designed to take instruction definitions in a procedural syntax, optimize the instruction body and output each instruction in Chisel HDL. As shown in Figure 1, input files or buffers are parsed into an appropriate AST. Instruction intrinsics are recognized and annotated in the AST. Further, register classes and registers are identified and marked as special globals. The AST is then translated to standard LLVM intermediate representation (IR). The tools utilize standard LLVM optimization passes to optimize the instruction IR. The tools then execute a series of StoneCutter-specific IR passes against the optimized LLVM IR in order to ensure that the instruction design is correct per the prescribed instruction format(s). Once verified as correct, the instruction intrinsics are expanded inline and the tools generate Chisel HDL.

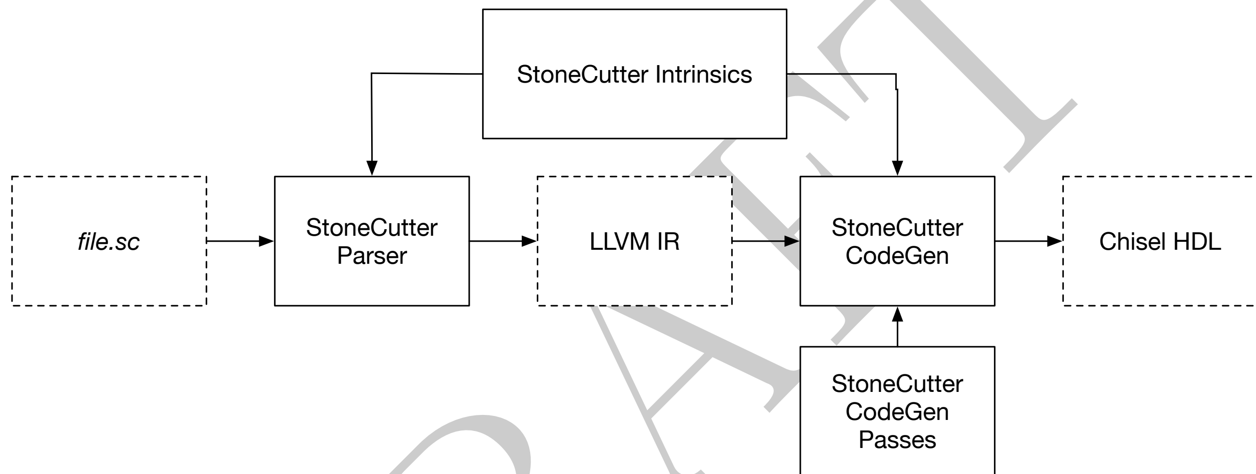

Figure 1: StoneCutter Architecture

Each StoneCutter input file is constructed is a specific, prescribed manner. The organization of the data members and the instruction definitions is done so specifically to elicit global and local hardware state within and across individual instructions. We find a sample of the StoneCutter file layout in Listing 1. StoneCutter input files begin with an optional comment block describing the contents of the file. Next, the individual register classes are defining for use by the instructions. These register classes should match the register class definitions in name and layout found in the respective CoreGen design input. Further information regarding the definition of register classes can be found in Section 2.6. The next portion of the file includes the actual instruction definitions. Each instruction definition is comprised of a *function*-like structure denoted with a `def` keyword. Each instruction takes some number of instruction arguments (I/O's) and contains private variables and arithmetic logic. Notice how the variable definitions are made near the top of the instruction definition. Much like the C language, this is required in StoneCutter in order to preemptively understand the scope of the temporary variables defined in the instruction. Sections 2.7-2.11 describe the rudimentary operations and the associated syntax that can be included in the instruction definition body.

DRAFT

```
#
# Sample StoneCutter File Layout
#

# Instruction Formats
instformat FORMAT1( FIELDTYPE FIELD1, FIELDTYPE FIELD2, ... )

# Register Class Definitions
regclass CLASS1( DATATYPE REG1, DATATYPE REG2, ... )
regclass CLASS2( DATATYPE REG3( DATATYPE SUBREG1, DATATYPE SUBREG2 ) )

# Instruction Definitions
def INST1( INPUT1 INPUT2 INPUT3 ) {
        # Variable Definitions
        DATATYPE VAR1, VAR2, VAR3
        DATATYPE VAR4

        # Instruction Body
        VAR4 = INPUT2 + INPUT1
        VAR1 = INPUT1 << 1
        VAR2 = INPUT3 << 2
        VAR3 = INPUT1 + INPUT2
        INPUT1 = VAR3
}

def INST2( REG1 REG3 SUBREG1 ) {
        # Instruction Body
        REG1 = REG3 * SUBREG1
}
```

Listing 1: StoneCutter File Structure

DRAFT

2.2 Syntactical Notes

Prior to reading the remaining sections, we highly suggest users and readers understand the following syntactical notes. Understanding these notes will significantly reduce the time required to become productive in StoneCutter.

- **Semicolons**: Unlike the C language, StoneCutter does not require utilizing semicolons (`;`) to terminate an expression. All raw expressions are in static single assignment (SSA) form. This implies that each expression will be in the form of `TARGET = INPUT <OP> OUTPUT`. The only exception to this rule is when intrinsics are utilized. A call to an intrinsic function may be

- **Complement Operations**: The StoneCutter language contains all the standard arithmetic, boolean and logical operations except complements. The bitwise complement operator from C (˜) is not supported and the boolean complement operator from C (`!`) is not supported. Complementing the result of a boolean operation and/or performing the bitwise complement of a variable or register must be performed using the `NOT` intrinsic. See Section 3.2.14 for more details.

2.3 Comments

Inline comments and comment text in StoneCutter must begin with the pound (#) sign. Comments may begin on new lines on inline with other code. Examples of using comments are shown in Listing 2.

```
1 # This is a stand alone comment
2 def inst0(RA RB RT) {    # this is an inline comment
3         RT = RA + RB
4 }
```

Listing 2: StoneCutter Comments

DRAFT

2.4 Datatypes

Much in the same manner as traditional programming models such as C and C++, StoneCutter supports a common set of datatypes for intermediate variables and registers. However, unlike traditional programming models, hardware design languages are required to support datatypes in non-byte aligned types. In order to provide more hardware-centric support, StoneCutter supports traditional data types as well as arbitrary width signed and unsigned integer types. These types are documented in Table 2.

Table 2: StoneCutter Datatypes

Type	Width (in bits)	Description
bool	1	Boolean. Analogous to unsigned 1 bit integer (u1)
u8	8	Unsigned 8 bit integer. Analogous to uint8_t
u16	16	Unsigned 16 bit integer. Analogous to uint16_t
u32	32	Unsigned 32 bit integer. Analogous to uint32_t
u64	64	Unsigned 64 bit integer. Analogous to uint64_t
s8	8	Signed 8 bit integer. Analogous to int8_t
s16	16	Signed 16 bit integer. Analogous to int16_t
s32	32	Signed 32 bit integer. Analogous to int32_t
s64	64	Signed 64 bit integer. Analogous to int64_t
float	32	Single precision floating point
double	64	Double precision floating point
uN	N bits	Arbitrary unsigned integer of N bits
sN	N bits	Arbitrary signed integer of N bits

An example of defining arbitrary width integers is shown in Listing 3.

```
1 #-- unsigned 7 bit integer "foo"
2 u7 foo
3
4 #-- unsigned 1024 integer "bar"
5 u1024 bar
6
7 #-- signed 37 integer "foobar"
8 s37 foobar
```

Listing 3: Arbitrary Width Datatypes

DRAFT

2.5 Instruction Format Definitions

Instruction formats and their associated fields are special variables in the StoneCutter language. Each defined instruction field must reside within a respective instruction format. Instruction fields are marked in the IR with attributes in order to ensure that they are correctly associated with the correct instruction format. Further, for each field that is denoted as a register field, the associated register class must also be specified in order to correctly link the register read datapath to the correct register file. Each of the associated fields is annotated as a global variable such that is can be explicitly utilized in any instruction prototype or instruction body.

Table 3: StoneCutter Instruction Format Field Types

Field	Mnemonic	Read/Write Access	Description
Instruction Encodings	`enc`	Read-Only	Instruction encodings such as opcodes and function codes
Immediate Values	`imm`	Read-Only	Immediate values encoded in the instruction payload
Register Indices	`reg[REGCLASS]`	Read-Write	Register index encodings. Must include the respective REGCLASS

For each field in the instruction format, the respective field is designated with a field type and a field name. There are three types of instruction field types. First, instruction fields denoted as encoding fields are utilized to designate individual instruction encodings such as opcodes and function codes in RISC architectures. When utilizing these fields in the implementation of an instruction, the value contained within the field is utilized. For example, if an instruction defines a field `opc` that contains the values 0x0A hex, reading from this field will provide the exact value 0x0A hex. Second, immediate fields are immediate values encoded directly within the instruction payload. Reading from immediate fields from within the instruction payload will read the literal immediate from the instruction payload. Finally, register fields denote register indices that access a single register file. Reading from this field directly will not deliver the index. Rather, accessing a register field will read or write the value at the designated index of the target register file. For fields that are denoted as register class fields (`reg`), the associated register class (register file) must also be specified. We summarize the permissible instruction field types in Table 3. Further, the syntax for the an entire instruction format block is shown in Listing 4. Encoding and immediate fields are always marked as read-only. Only register fields may be written to.

```
1 instformat FName1( FIELDTYPE FIELD1, FIELDTYPE FIELD2, ... )
2 instformat FName2( reg[REGCLASS] FIELD1, ... )
```

Listing 4: Instruction Format Definition Syntax

We find an example instruction format definition for a simple RISC ISA with encodings for an opcode and function code as well as three register fields in Listing 5.

```
1 instformat RISC( enc opc, enc func, reg[GPR] RT, reg[GPR] RA, reg[GPR] RB )
```

Listing 5: Sample Instruction Format Definition

DRAFT

2.6 Register Class Definitions

Register classes and their associated registers are special variables in the StoneCutter language. Each defined register is must reside within its respective register class. Registers are marked in the IR with attributes in order to ensure that they belong to the correct register class when being utilized in an instruction body. Each register is also annotated as a global variable such that it can be explicitly utilized in any instruction definition body.

```
1 regclass RCName1( DATATYPE RegName1, DATATYPE RegName2, ... )
2 regclass RCName2( DATATYPE RegName( DATATYPE SubReg1, DATATYPE SubReg2 ) )
```

Listing 6: Register Class Definition Syntax

We find the syntax for defining register classes and their associated registers in Listing 6. Each register class is definition is marked with a `regclass` keyword followed by the register class name. Each of the registers defined within the register class are enclosed within parenthesis. Within the parenthesis, each register definition must be preceded by its respective datatype (Section 2.4) and the register name. Registers are separated by commas. For registers that have subfields, we may also explicitly annotate these within the register definition. For each register with subfields, the subfields are defined within parenthesis attached to the register definition (Line 2 in Listing 6). Each subfield must also contain its respective datatype. The combined number of bits for all subfields within a register definition must not exceed the total number of bits in that register.

We find an example register class definition for a simple RISC ISA in Listing 7.

```
1 regclass GPR( u64 r0, u64 r1, u64 r2, u64 r3, u64 r4, u64 r5, u64 r6,
2                         u64 r7, u64 r8, u64 r9, u64 r10, u64 r11, u64 r12,
3                         u64 r13, u64 r14, u64 r15, u64 r16, u64 r17, u64 r18,
4                         u64 r19, u64 r20, u64 r21, u64 r22, u64 r23, u64 r24,
5                         u64 r25, u64 r26, u64 r27, u64 r28, u64 r29, u64 r30,
6                         r64 r31 )
7 regclass CTRL( u64 pc, u64 exc, u64 ne, u64 gt, u64 lt, u64 gte,
8                           u64 lte, u64 sp, u64 fp, u64 rp )
```

Listing 7: Sample Register Class Definition

DRAFT

2.7 Instruction Prototypes

For each instruction defined in the StoneCutter syntax, we must construct an instruction prototype. The instruction prototype is an important and powerful feature of the StoneCutter language. The instruction prototype drives two major features in the downstream Chisel HDL produced by a compiled StoneCutter input. First, the prototype describes the attachment of the respective instruction implementation to the instruction crack logic generated by the CoreGen IR. The instruction mnemonic utilized in the instruction definition must match that of the instruction mnemonic defined in the CoreGen IR. In this manner, the combined StoneCutter and CoreGen IR infrastructure logic can match the instruction crack implementation and the instruction implementation (et al. ALU). The instruction mnemonic may optionally define the instruction format utilized to decode the target instruction using the `:INSTFORMAT` syntax immediately following the instruction name. The `INSTFORMAT` name must match an instruction format as defined in Section 2.5.

Second, the instruction definition argument list defines the set of standard I/O ports utilized by the instruction. Standard I/O ports are utilized to pipeline register read and write operations within an optimized pipeline. While it is entirely permissible to directly address registers or register files from within an instruction definition that are not defined in the argument list, the eventual downstream instruction implementation may require additional register read/write operations that induce pipeline stalls. Keep in mind that standard I/O ports such as the clock, register hazarding and stall signals are automatically instantiated.

```
1 def INSTNAME[:INSTFORMAT]( ARG1 ARG2 ARG3 ... ) {
2 }
```

Listing 8: Instruction Prototype Format

As mentioned above, the instruction prototype includes two main structures: the instruction name and the instruction arguments (Listing 8). The instruction name must match the associated instruction name defined in the CoreGen IR. The instruction name is case sensitive. See the CoreGen IR specification for more information **??**. The instruction argument list contains a set of register, instruction field or register class names that define the standard I/O functions for the optimized pipeline. The instruction argument list must match the set of register, instruction format fields or register file designators in the instruction format for the target instruction.

Register arguments are interpreted literally. In this manner, a register I/O is performed to the specific register index denoted by the target register. Register class arguments are interpreted logically. The register class arguments utilize the index specified in the assembled instruction payload to load or store from the respective register index in the target register file.

We see an example of a series of instruction prototypes in Listing 9. We utilize the register class and register definitions from Section 2.6. We define four instructions, `add`, `move`, `inc` and `sub`. The `add` instruction prototype contains a single argument, `GPR`, that defines I/O ports to/from the GPR register class. The `move` instruction utilizes both the `GPR` and the `CTRL` register classes. The `inc` instruction utilizes the explicit `pc` register as well as the `GPR` register class. Finally, the `sub` Notice that the arguments within the prototype are **not** separated by commas. Further, we see that the associated instruction definition body is contained within a set of brackets ({}).

DRAFT

```
instformat RISC( enc opc, enc func, reg[GPR] RT, reg[GPR] RA, reg[GPR] RB )
regclass GPR( u64 r0, u64 r1, u64 r2, u64 r3, u64 r4, u64 r5, u64 r6,
                       u64 r7, u64 r8, u64 r9, u64 r10, u64 r11, u64 r12,
                       u64 r13, u64 r14, u64 r15, u64 r16, u64 r17, u64 r18,
                       u64 r19, u64 r20, u64 r21, u64 r22, u64 r23, u64 r24,
                       u64 r25, u64 r26, u64 r27, u64 r28, u64 r29, u64 r30,
                       r64 r31 )
regclass CTRL( u64 pc, u64 exc, u64 ne, u64 gt, u64 lt, u64 gte,
                         u64 lte, u64 sp, u64 fp, u64 rp )

# add operation
def add( GPR ){
}

# move between GPR and CTRL
def move( GPR CTRL ){
}

# increment PC by GPR
def inc( pc GPR ){
}

# subtract RT = RA - RB
def sub:RISC( RT RA RB ){
}
```

Listing 9: Instruction Prototype Format

DRAFT

2.8 Variable Definitions

In addition to utilizing register classes and register fields as variables within an instruction definition, users may also specify local, temporary variables in the instruction definition body. These values may be utilized as temporary storage for complex, multi-stage operations, loop counters and other intermediate state. The StoneCutter compiler infrastructure will make every effort to minimize the overall hardware impact of these intermediate values to the size and timing of the target design. As a result, these temporary values may be optimized out and/or reused by adjacent instructions in the generated downstream Chisel HDL.

In the same manner as individual registers, variables are required to have a defined type. These types follow the same convention as defined in Section 2.4. Variable names may not collide with existing instruction format fields or register names. Much in the same manner as the C language, StoneCutter permits users to define multiple, independent variables using the same datatype. Variables must also be defined at the **top** of an instruction definition. This permits StoneCutter to sufficiently scope temporary variables across the body of an instruction. The format of variable definitions is noted in Listing 10. We provide an example of a series of variable definitions in Listing 11.

```
def INST( ... ){
  DATATYPE name1
  DATARYPE name2 = VALUE
  DATATYPE name3, name4, ...

  <instruction body>
}
```

Listing 10: Variable Definitions

```
instformat RISC( enc opc, enc func, reg[GPR] RT, reg[GPR] RA, reg[GPR] RB )

def add( RT RA RB ){
  u64 var1, var2
  if( RA == RB ){
    var1 = 5
    var2 = 6
  }else{
    var1 = 7
    var2 = 998
  }
  RT = var1 & var2
}
```

Listing 11: Sample Variable Definitions

DRAFT

2.9 Arithmetic Operations

The StoneCutter language supports the standard set of arithmetic operations as defined by other C-like languages. However, given the nature of compiling the StoneCutter language to a target-specific circuit design, the StoneCutter compiler and tool chain enforces a number of idiosyncratic features when processing variables for arithmetic operations. First, the StoneCutter language scans the variables associated with the left hand side of a target arithmetic operation and automatically converts the operation to perform arithmetic using the width of the largest variable. Further, when storing the result of an operation to a target variable or register (right hand side of an operation), StoneCutter converts the resulting data to saturate the bit space of the target. The process of converting arithmetic operations and variables to larger bit space is performed using zero extension in order to avoid poisoning the numerical consistency of an operation. Users may override this behavior by utilizing sign and zero extended intrinsics (see Section 3.2).

We provide a summary of permissible arithmetic operations in Table 4.

Table 4: StoneCutter Arithmetic Operations

<table>
<tr><th>Operator</th><th>Example</th><th>Description</th></tr>
<tr><td>=</td><td>RT = RB</td><td>Assignment operation</td></tr>
<tr><td>+</td><td>RT = RA + RB</td><td>Add operation</td></tr>
<tr><td>−</td><td>RT = RA − RB</td><td>Subtract operation</td></tr>
<tr><td>⋆</td><td>RT = RA ⋆ RB</td><td>Multiplication operation</td></tr>
<tr><td>\</td><td>RT = RA \ RB</td><td>Division operation</td></tr>
<tr><td>%</td><td>RT = RA % RB</td><td>Modulo operation</td></tr>
<tr><td>&</td><td>RT = RA & RB</td><td>Bitwise and operation</td></tr>
<tr><td>|</td><td>RT = RA | RB</td><td>Bitwise or operation</td></tr>
<tr><td>∧</td><td>RT = RA ∧ RB</td><td>Bitwise xor operation</td></tr>
<tr><td><<</td><td>RT = RA << RB</td><td>Shift left operation</td></tr>
<tr><td>>></td><td>RT = RA >> RB</td><td>Shift right operation</td></tr>
</table>

NOTE: Please note that StoneCutter does not currently support the bitwise complement operation (˜) and the logical not operation (!).

DRAFT

2.10 Conditional Operations

2.10.1 Boolean Operators

The StoneCutter language syntax supports the standard set of boolean operators as is defined by other C-like languages. The one exception being the logical not operation (!). StoneCutter does not support the logical not operation for boolean operations. Rather, users seeking to complement the result of a boolean operation of a singular boolean value must utilize the `NOT` intrinsic (Section 3.2.14).

We provide a summary of permissible boolean operations in Table 5.

Table 5: StoneCutter Boolean Operations

<table>
<tr><th>Operator</th><th>Example</th><th>Description</th></tr>
<tr><td>==</td><td>RA == RB</td><td>Logical equivalence</td></tr>
<tr><td>!=</td><td>RA != RB</td><td>Logical in-equivalence</td></tr>
<tr><td><</td><td>RA < RB</td><td>Less than</td></tr>
<tr><td>></td><td>RA > RB</td><td>Greater than</td></tr>
<tr><td><=</td><td>RA <= RB</td><td>Less than or equal to</td></tr>
<tr><td>>=</td><td>RA >= RB</td><td>Greater than or equal to</td></tr>
<tr><td>&&</td><td>RA && RB</td><td>Logical and</td></tr>
<tr><td>||</td><td>RA || RB</td><td>Logical or</td></tr>
</table>

DRAFT

2.10.2 Flow Control

The StoneCutter language provides support for C-like conditional flow control using standard *if-else* clauses. The statements contained within these clauses must result in a boolean value that is interrogated in order to determine which body of code (circuit) to execute. Flow control operations are required to include a single *if* statement. They may optionally include a complementary *else* statement.

```
1 if( BOOLEAN OPERATION){
2   # Conditional body
3 }
4
5 if( BOOLEAN OPERATION){
6   # Conditional if body
7 }else{
8   # Conditional else body
9 }
```

Listing 12: Flow Control Syntax

We provide a set of example conditional flow control operations in Listing 13.

```
1  def add( RT RA RB ){
2    if( RA > RB ){
3      RB = RA + RB
4    }
5
6    if( RB < RT ){
7      RT = RB
8    }else{
9      RT = RT << 1
10   }
11 }
```

Listing 13: Sample If-Else Syntax

DRAFT

2.11 Loop Operations

2.11.1 For Loops

The first style of loop statement supported by the StoneCutter language is the *for* loop. The StoneCutter for loop is structured in a similar manner as the traditional C for loop. The loop is provided with a base case loop counter, a loop termination statement and an optional iterator trip step. The loop counter can be an existing variable or a newly defined temporary variable. Unlike other temporary variables, loop trip counters can be defined inline within the for loop structure. The loop termination statement includes one or more comparison statements that define the termination state of the loop using the loop trip counter. The optional loop trip counter iterator can be an immediate value or a variable. If this optional value is not specified, the iterator value is assumed to be "1". The for loop body is contained with brackets ({}) and may include any other set of StoneCutter statements. The for loop syntax is depicted in Listing 14.

```
1  for( LOOPTRIP = BASE; LOOPTRIP <COMPARATOR> TERMINATOR ){
2    # for loop body
3  }
4
5  for( LOOPTRIP = BASE; LOOPTRIP <COMPARATOR> TERMINATOR; ITERATOR ){
6    # for loop body
7  }
```

Listing 14: For Loop Syntax

The comparator operation utilized to terminate the loop is one of the standard variable comparison operations from C-like languages. We summarize these operators in Table 6.

Table 6: StoneCutter Comparison Operations

Operator	Description
<	Less Than
>	Greater Than
<=	Less Than Or Equal To
>=	Greater Than Or Equal To
==	Equal To
!=	Not Equal To

We provide a set of example for loops in Listing 15.

```
1  def add( RT RA RB ){
2    u64 var1
3    for( i = 1; i < RA ){
4      RB = RB + 1
5    }
6    for( j = 0; i< RB; 5 )
7      RT = RB | RT
8    }
9    for( var = 1024; i <= RT; RB ){
10     RT = var
11   }
12 }
```

Listing 15: Sample For Loop Syntax

DRAFT

2.11.2 While Loops

The second style of loop supported by the StoneCutter language is the *while* loop. The StoneCutter while loop is structured in a similar manner as the traditional C while loop. The loop is provided with a termination statement in the form of a comparator. It is up to the user to modify the variable utilized in the comparator statement such that a valid termination state can be reached. StoneCutter does not otherwise validate the potential for infinite while loops. The while loop body is contained within brackets ({}) and may include any other set of StoneCutter statements. The while loop syntax is depicted in Listing 16.

```
1  while( COMPARATOR STATEMENT ){
2    # while loop body
3  }
```

Listing 16: While Loop Syntax

We provide a set of example while loops in Listing 17.

```
1  def add( RT RA RB ){
2    u64 var1 = 1024
3    while( RA < RB ){
4      RT = RB << 1
5      RA = RA + 1
6    }
7    while( var1 >= RT ){
8      var1 = var1 - 5
9      RT = RT << 1
10   }
11 }
```

Listing 17: Sample While Loop Syntax

DRAFT

2.11.3 Do While Loops

The final style of loop supported by the StoneCutter language is the *do while* loop. The StoneCutter do while loop is structured in a similar manner as the traditional the traditional C do while loop. The loop is provided with an entry state provided by a *do* statement. The do statement is followed by a loop body contained within brackets ({}). The do while loop termination statement follows the last bracket and is contained in a *while* block similar in form as the while loop structure (Section 2.11.2). The while loop syntax is depicted in Listing 18.

```
1 do{
2   # do while loop body
3 }while( COMPARATOR STATEMENT )
```

Listing 18: Do While Loop Syntax

We provide a set of example do while loops in Listing 19.

```
1  def add( RT RA RB ){
2    u64 var1 = 1024
3    do{
4      RT = RB << 1
5      RA = RA + 1
6    }while( RA < RB )
7    do{
8      var1 = var1 - 5
9      RT = RT << 1
10   }while( var1 >= RT )
11 }
```

Listing 19: Sample Do While Loop Syntax

DRAFT

3 Intrinsic Functions

3.1 Overview

In the same manner as other C-like languages and compiler infrastructures, the StoneCutter language provides a notional set of builtin intrinsic functions. Much like other languages, these intrinsic functions provide pathological circuit functionality in a convenient and optimized package. However, unlike traditional strongly typed languages that provide unique intrinsic functions for disparate data types, the StoneCutter language provides single intrinsics that support all possible StoneCutter data types. In this manner, the StoneCutter compiler infrastructure interrogates the arguments of each intrinsic at compile time and constructs an optimized Chisel representation of the target operation using the user-specified types. As a result, users may utilize the forthcoming set of intrinsic operations with any target data types supported by the StoneCutter language.

StoneCutter intrinsics are classified as two disparate types: *Arithmetic* intrinsics and *Memory* intrinsics. Arithmetic intrinsics perform some notional permutation on the target input data. Arithmetic intrinsics generally require input and return the output as a unique data member (`OUTPUT = INTRINSIC( INPUT )`). Data members are not modified in place. Memory intrinsics are different in that they interact with the memory infrastructure generated by the CoreGen [1] code generation facilities such that the desired function unit may interact with external memories such as caches and off-chip memory units. Memory intrinsics are generally provide functionality such as *load* and *store* operations.

StoneCutter intrinsic operations may be utilized anywhere within the body of an instruction definition. The initial StoneCutter language parser *lowers* any encountered intrinsic operations to the equivalent of a function call. The StoneCutter code generator expands these intrinsic function calls into their full circuit descriptions using the target input and output types. In this manner, including intrinsics within the body of conditional flow control, loops and other basic blocks is both type safe and functionally sound.

Intrinsics can be utilized similar to C-style function calls. Intrinsics names are placed in the instruction body with the appropriate number of arguments. Intrinsic arguments are separated by commas. Intrinsics that return values can be utilized within other expressions and/or assigned to values (`RT = INTRIN(..)`). The syntax for utilizing intrinsics is depicted in Listing 20. A summary of the intrinsics is provided in Section 5.

```
1 def add( RT RA RB ){
2   u64 var1, var2
3   INTRIN( RB )
4   RB = INTRIN( RA )
5   RT = INTRIN( RA, RB )
6 }
```

Listing 20: Intrinsic Syntax

DRAFT

3.2 Arithmetic Intrinsics

3.2.1 CLZ

Table 7: CLZ Intrinsic

Mnemonic	`RT = CLZ(RA)`
Description	Counts the leading zeros (little endian)
Arguments	RA: Contains the value to count the leading zeros

Example	def INST(RA RB RT){
	RT = CLZ(RA)
	}

Return	The number of leading zeros in the input value

DRAFT

3.2.2 COMPRESS

Table 8: COMPRESS Intrinsic

Mnemonic	`RT = COMPRESS(RA)`
Description	Bit compression. For the number of '1' values in RA, set the least significant bits to '1'. Zero extend the result.
Arguments	RA: Contains the value to compress

Example	def INST(RA RB RT){
	RT = COMPRESS(RA)
	}

Return	The compressed version of RA

DRAFT

3.2.3 COMPRESSM

Table 9: COMPRESSM Intrinsic

Mnemonic	`RT = COMPRESSM(RA,RB)`
Description	Bit compression under mask. Perform a bitwise '&' operation of the input and bitmask. For the number of '1' values in the result, set the least significant bits to '1'. Zero extend the result.
Arguments	RA: Contains the value to compress RB: Contains the bitmask to select compression bits

Example	def INST(RA RB RT){
	RT = COMPRESSM(RA, RB)
	}

Return	The compressed version of RA via the bitmask RB

DRAFT

3.2.4 CTZ

Table 10: CTZ Intrinsic

Mnemonic	`RT = CTZ(RA)`
Description	Counts the trailing zeros (little endian)
Arguments	RA: Contains the value to count the trailing zeros

Example	def INST(RA RB RT){
	RT = CTZ(RA)
	}

Return	The number of trailing zeros in the input value

DRAFT

3.2.5 DOZ

Table 11: DOZ Intrinsic

Mnemonic	`RT = DOZ(RA,RB)`
Description	Performs "first grade subtraction." Returns $(RA - RB)$ IFF $RA >= RB$. Returns 0 otherwise
Arguments	RA: Contains the left hand input value RB: Contains the right hand input value

Example	def INST(RA RB RT){
	RT = DOZ(RA, RB)
	}

Return	If($RA >=$ RB) { return $RA - RB$ }else{ return 0 }

DRAFT

3.2.6 EXTRACTS

Table 12: EXTRACTS Intrinsic

Mnemonic	`EXTRACTS(RA,RB,RC)`
Description	Extract the target field starting a bit position RC from RB, store into RA and sign extend.
Arguments	RA: Contains the target output value RB: Input value to extract from RC: Starting bit position

Example	def INST(RA RB RC RT){
	EXTRACTS(RA, RB, RC)
	}

Return	Nothing is returned

DRAFT

3.2.7 EXTRACTZ

Table 13: EXTRACTZ Intrinsic

Mnemonic	`EXTRACTZ(RA,RB,RC)`
Description	Extract the target field starting a bit position RC from RB, store into RA and zero extend.
Arguments	RA: Contains the target input value RB: Input value to extract from RC: Starting bit position

Example	def INST(RA RB RC RT){
	EXTRACTZ(RA, RB, RC)
	}

Return	Nothing is returned

DRAFT

3.2.8 INSERTS

Table 14: INSERTS Intrinsic

Mnemonic	`INSERTS(RA,RB, RC)`
Description	Insert the field (RB) into RA starting at bit position RC and sign extend
Arguments	RA: Contains the target input value RB: Input value to insert RC: Starting bit position

Example	def INST(RA RB RC RT){
	INSERTS(RA, RB, RC)
	}

Return	Nothing is returned

DRAFT

3.2.9 INSERTZ

Table 15: INSERTZ Intrinsic

Mnemonic	`INSERTZ(RA,RB,RC)`
Description	Insert the field (RB) into RA starting at bit position RC and zero extend
Arguments	RA: Contains the target input value RB: Input value to insert RC: Starting a bit position

Example	def INST(RA RB RC RT){
	INSERTZ(RA, RB, RC)
	}

Return	Nothing is returned

DRAFT

3.2.10 MAJ

Table 16: MAJ Intrinsic

Mnemonic	`RT = MAJ(RA,RB,RC)`
Description	Performs a majority vote of the values RA,RB and RC. Returns true if at least two inputs are true; returns false otherwise
Arguments	RA: Input 1 RB: Input 2 RC: Input3

Example	def INST(RA RB RC RT){
	RT = MAJ(RA, RB, RC)
	}

Return	Returns true if at least two inputs are true; returns false otherwise

DRAFT

3.2.11 MAX

Table 17: MAX Intrinsic

Mnemonic	`RT = MAX(RA,RB)`
Description	Returns the maximum value of RA and RB
Arguments	RA: Input 1 RB: Input 2

Example	def INST(RA RB RT){
	RT = MAX(RA, RB)
	}

Return	If(RA > RB){ return RA }else{ return RB}

DRAFT

3.2.12 MERGE

Table 18: MERGE Intrinsic

Mnemonic	`MERGE(RA,RB,RC)`
Description	Selectively merge bits specified by the mask in RC from RB into RA
Arguments	RA: Contains the target data to merge into RB: Contains the value to selectively merge RC: Contains bitwise mask input

Example	def INST(RA RB RT){
	MERGE(RA, RB, RC)
	}

Return	Nothing is returned

DRAFT

3.2.13 MIN

Table 19: MIN Intrinsic

Mnemonic	`RT = MIN(RA,RB)`
Description	Returns the minimum value of RA and RB
Arguments	RA: Input1 RB: Input2

Example	def INST(RA RB RT){
	RT = MIN(RA, RB)
	}

Return	If(RA < RB){ return RA }else{ return RB}

DRAFT

3.2.14 NOT

Table 20: NOT Intrinsic

Mnemonic	`RT = NOT(RA)`
Description	Returns the bitwise complement of the input value. If the input value is a single bit boolean (`u1`), the logical complement is returned. If the input value is larger than a single bit, the bitwise complement is returned
Arguments	RA: Contains the input to be complemented

Example	def INST(RA RB RT){
	RT = NOT(RA)
	}

Return	Bitwise complement of RA

DRAFT

3.2.15 POPCOUNT

Table 21: POPCOUNT Intrinsic

Mnemonic	`RT = POPCOUNT(RA)`
Description	Returns the population count of the input variable.
Arguments	RA: Contains the input value for the popcount

Example	def INST(RA RB RT){
	RT = POPCOUNT(RA)
	}

Return	The number of positive (1) bits in the input value is returned. The output storage must contain sufficient space to storage the maximum popcount value (all positive bits)

DRAFT

3.2.16 REVERSE

Table 22: REVERSE Intrinsic

Mnemonic	`RT = REVERSE(RA)`
Description	Reverse the bit pattern of the input variable
Arguments	RA: Contains the input to be reversed

Example	def INST(RA RB RT){
	RT = REVERSE(RA)
	}

Return	The reversed bit pattern of the input variable

DRAFT

3.2.17 ROTL

Table 23: ROTL Intrinsic

Mnemonic	`RT = ROTL(RA,RB)`
Description	Rotate the value in RA by RB bits to the left.
Arguments	RA: Contains the value to be rotated RB: Contains the number of bits to rotate by

Example	def INST(RA RB RT){
	RT = ROTL(RA, RB)
	}

Return	The input value is rotated left by RB bits

DRAFT

3.2.18 ROTR

Table 24: ROTR Intrinsic

Mnemonic	`RT = ROTR(RA,RB)`
Description	Rotate the value in RA by RB bits to the right.
Arguments	RA: Contains the value to be rotated RB: Contains the number of bits to rotate by

Example	def INST(RA RB RT){
	RT = ROTR(RA, RB)
	}

Return	The input value is rotated right by RB bits

DRAFT

3.2.19 SEXT

Table 25: SEXT Intrinsic

Mnemonic	`Rt = SEXT(RA)`
Description	Sign extend the input value
Arguments	RA: Contains the input value to sign extend

Example	def INST(RA RB RT){
	RT = SEXT(RA)
	}

Return	The sign extended version of the input

DRAFT

3.2.20 ZEXT

Table 26: ZEXT Intrinsic

Mnemonic	`ZEXT(RA,RB)`
Description	Zero extend the input value
Arguments	RA: Contains the input value to zero extend

Example	def INST(RA RB RT){
	RT = ZEXT(RA)
	}

Return	The zero extended version of the input

DRAFT

3.3 Memory Intrinsics

3.3.1 LOAD

Table 27: LOAD Intrinsic

Mnemonic	`RT = LOAD(RA)`
Description	Load the value at the address in RA. Attempt to derive the datatype
Arguments	RA: Contains the target address

Example	def INST(RA RB RT){
	RT = LOAD(RA)
	}

Return	The data from the load operation

DRAFT

3.3.2 STORE

Table 28: STORE Intrinsic

Mnemonic	`STORE(RA,RB)`
Description	foo
Arguments	RA: Contains the data to be stored RB: Contains the target address

Example	def INST(RA RB RT){
	STORE(RA, RB)
	}

Return	Nothing is returned

DRAFT

3.3.3 LOADELEM

Table 29: LOADELEM Intrinsic

Mnemonic	`RT = LOADELEM(RA,RB)`
Description	Load the value at the address in RA. The element size is contained in RB.
Arguments	RA: Contains the target address RB: Contains the size of the element

Example	def INST(RA RB RT){
	RT = LOADELEM(RA, RB)
	}

Return	The data from the load operation

DRAFT

3.3.4 STOREELEM

Table 30: STOREELEM Intrinsic

Mnemonic	`STOREELEM(RA,RB,RC)`
Description	Stores the data in RA to the address at RB. The element size is contained in RC.
Arguments	RA: Contains the data to be stored RB: Contains the target address RC: Contains the size of the element

Example	def INST(RA RB RC RT){
	STOREELEM(RA, RB, RC)
	}

Return	Nothing is returned

DRAFT

4 Appendix A: Sample StoneCutter Implementation

```
#-- StoneCutter source file for ISA=BasicRISC.ISA

# Instruction Formats
instformat Arith.if(reg[GPR] ra,reg[GPR] rb,reg[GPR] rt,enc opc,
                    enc func,imm imm)
instformat ReadCtrl.if(reg[GPR] ra,reg[CTRL] rb,reg[GPR] rt,
                       enc opc,enc func,imm imm)
instformat WriteCtrl.if(reg[GPR] ra,reg[GPR] rb,reg[CTRL] rt,
                        enc opc,enc func,imm imm)

# Register Class Definitions
regclass GPR( u64 r0, u64 r1, u64 r2, u64 r3, u64 r4, u64 r5,
              u64 r6, u64 r7, u64 r8, u64 r9, u64 r10, u64 r11,
              u64 r12, u64 r13, u64 r14, u64 r15, u64 r16, u64 r17,
              u64 r18, u64 r19, u64 r20, u64 r21, u64 r22, u64 r23,
              u64 r24, u64 r25, u64 r26, u64 r27, u64 r28, u64 r29,
              u64 r30, u64 r31 )
regclass CTRL( u64 pc, u64 exc, u64 ne, u64 eq, u64 gt, u64 lt,
               u64 gte, u64 lte, u64 sp, u64 fp, u64 rp )

# Instruction Definitions
# add
def add:Arith.if( ra rb rt imm )
{
  rt = ra + rb
}

# sub
def sub:Arith.if( ra rb rt imm )
{
  rt = ra - rb
}

# mul
def mul:Arith.if( ra rb rt imm )
{
  rt = ra * rb
}

# div
def div:Arith.if( ra rb rt imm )
{
  rt = ra / rb
}

# divu
def divu:Arith.if( ra rb rt imm )
{
  rt = ra / rb
}

```

DRAFT

```
52  # sll
53  def sll:Arith.if( ra rb rt imm )
54  {
55    rt = ra << rb
56  }
57
58  # srl
59  def srl:Arith.if( ra rb rt imm )
60  {
61    rt = ra >> rb
62  }
63
64  # sra
65  def sra:Arith.if( ra rb rt imm )
66  {
67    rt = ra >> rb
68  }
69
70  # and
71  def and:Arith.if( ra rb rt imm )
72  {
73    rt = ra & rb
74  }
75
76  # or
77  def or:Arith.if( ra rb rt imm )
78  {
79    rt = ra | rb
80  }
81
82  # nand
83  def nand:Arith.if( ra rb rt imm )
84  {
85    rt = NOT(ra & rb)
86  }
87
88  # nor
89  def nor:Arith.if( ra rb rt imm )
90  {
91    rt = NOT(ra | rb)
92  }
93
94  # xor
95  def xor:Arith.if( ra rb rt imm )
96  {
97    rt = ra ˆ rb
98  }
99
100 # cmp.ne
101 def cmp.ne:Arith.if( ra rb rt imm )
102 {
103   if( ra != rb ){
104     rt = 2
105   }else{
```

```
106       rt = 0
107     }
108   }
109
110   # cmp.eq
111   def cmp.eq:Arith.if( ra rb rt imm )
112   {
113     if( ra == rb ){
114       rt = 3
115     }else{
116       rt = 0
117     }
118   }
119
120   # cmp.gt
121   def cmp.gt:Arith.if( ra rb rt imm )
122   {
123     if( ra > rb ){
124       rt = 4
125     }else{
126       rt = 0
127     }
128   }
129
130   # cmp.lt
131   def cmp.lt:Arith.if( ra rb rt imm )
132   {
133     if( ra < rb ){
134       rt = 5
135     }else{
136       rt = 0
137     }
138   }
139
140   # cmp.gte
141   def cmp.gte:Arith.if( ra rb rt imm )
142   {
143     if( ra >= rb ){
144       rt = 6
145     }else{
146       rt = 0
147     }
148   }
149
150   # cmp.lte
151   def cmp.lte:Arith.if( ra rb rt imm )
152   {
153     if( ra <= rb ){
154       rt = 7
155     }else{
156       rt = 0
157     }
158   }
159
```

DRAFT

```
160 # lb
161 def lb:Arith.if( ra rb rt imm )
162 {
163   rt = SEXT(LOADELEM(ra+imm,8))
164 }
165
166 # lh
167 def lh:Arith.if( ra rb rt imm )
168 {
169   rt = SEXT(LOADELEM(ra+imm,16))
170 }
171
172 # lw
173 def lw:Arith.if( ra rb rt imm )
174 {
175   rt = SEXT(LOADELEM(ra+imm,32))
176 }
177
178 # ld
179 def ld:Arith.if( ra rb rt imm )
180 {
181   rt = LOADELEM(ra+imm,64)
182 }
183
184 # sb
185 def sb:Arith.if( ra rb rt imm )
186 {
187   STOREELEM(ra,rt+imm,8)
188 }
189
190 # sh
191 def sh:Arith.if( ra rb rt imm )
192 {
193   STOREELEM(SEXT(ra),rt+imm,16)
194 }
195
196 # sw
197 def sw:Arith.if( ra rb rt imm )
198 {
199   STOREELEM(SEXT(ra),rt+imm,32)
200 }
201
202 # sd
203 def sd:Arith.if( ra rb rt imm )
204 {
205   STOREELEM(SEXT(ra),rt+imm,64)
206 }
207
208 # lbu
209 def lbu:Arith.if( ra rb rt imm )
210 {
211   rt = ZEXT(LOADELEM(ra+imm,8))
212 }
213
```

DRAFT

```
214 # lhu
215 def lhu:Arith.if( ra rb rt imm )
216 {
217   rt = ZEXT(LOADELEM(ra+imm,16))
218 }
219
220 # lwu
221 def lwu:Arith.if( ra rb rt imm )
222 {
223   rt = ZEXT(LOADELEM(ra+imm,32))
224 }
225
226 # sbu
227 def sbu:Arith.if( ra rb rt imm )
228 {
229   STOREELEM(ZEXT(ra),rt+imm,8)
230 }
231
232 # shu
233 def shu:Arith.if( ra rb rt imm )
234 {
235   STOREELEM(ZEXT(ra),rt+imm,16)
236 }
237
238 # swu
239 def swu:Arith.if( ra rb rt imm )
240 {
241   STOREELEM(ZEXT(ra),rt+imm,32)
242 }
243
244 # not
245 def not:Arith.if( ra rb rt imm )
246 {
247   rt = NOT(ra)
248 }
249
250 # bra
251 def bra:Arith.if( ra rb rt imm )
252 {
253   pc = rt
254 }
255
256 # br
257 def br:Arith.if( ra rb rt imm )
258 {
259   pc = SEXT(pc + rt)
260 }
261
262 # cadd
263 def cadd:ReadCtrl.if( ra rb rt imm )
264 {
265   rt = ra + rb
266 }
267
```

DRAFT

```
268 # brac
269 def brac:ReadCtrl.if( ra rb rt imm )
270 {
271   if( ra == rb ){
272     pc = rt
273   }else{
274     pc = pc + 4
275   }
276 }
277
278 # brc
279 def brc:ReadCtrl.if( ra rb rt imm )
280 {
281   if( ra == rb ){
282     pc = pc + rt
283   }else{
284     pc = pc + 4
285   }
286 }
287
288 # ladd
289 def ladd:WriteCtrl.if( ra rb rt imm )
290 {
291   rt = ra + rb
292 }
293
294 # brr
295 def brr:WriteCtrl.if( ra rb rt imm )
296 {
297   pc = rt
298 }
```

Listing 21: Sample StoneCutter

DRAFT

5 Appendix B: Intrinsic Function Table

Table 31: StoneCutter Intrinsics

Intrinsic	Inputs	Outputs	Description
CLZ	1	1	Count leading zero
COMPRESS	1	1	Bit compress
COMPRESSM	1	2	Bit compress under mask
CTZ	1	1	Count trailing zero
DOZ	1	2	First grade subtraction
EXTRACTS	0	3	Bit extract and sign extend
EXTRACTZ	0	3	Bit extract and zero extend
INSERTS	0	3	Bit insert and sign extend
INSERTZ	0	3	Bit insert and zero extend
MAJ	1	3	Majority vote
MAX	1	2	Max
MERGE	0	3	Merge under mask
MIN	1	2	Min
NOT	1	1	Bit complement
POPCOUNT	1	1	Population count
REVERSE	1	1	Bit reverse
ROTL	1	2	Rotate left
ROTR	1	2	Rotate right
SEXT	1	1	Signe extend
ZEXT	1	2	Zero extend
LOAD	1	1	Memory load
STORE	0	2	Memory store
LOADELEM	1	2	Memory element load
STOREELEM	0	3	Memory element store

DRAFT

References

[1] J. Leidel and F. Conlon, "Coregen intermediate representation language specification," 2019. [Online]. Available: https://github.com/opensocsysarch/CoreGenIRSpec

DRAFT