Design Concepts with System Architect: Level 2

John Leidel

Chief Scientist, Tactical Computing Laboratories

ver 2020.09.15

Tutorial Series

- Level 0: Introduction to System Architect
- Level 1: System Architect Design Concepts and Developing a basic RISC processor
- **Level 2: Instruction-Level (StoneCutter) Implementation Concepts**
- Level 3: Advanced Design Concepts
- Level 4: System Architect Plugins and Integrating External RTL

OpenSoC System Architect

Overview

- StoneCutter Overview
- StoneCutter Tool Infrastructure
- Intro to StoneCutter Syntax
- Implementing a Basic RISC Device
- References

OpenSoC System Architect

StoneCutter Overview

Instruction Definition Concepts

OpenSoC System Architect

What is StoneCutter?

- A concise language for specifying the implementation of a **SINGLE** instruction
- StoneCutter implements instructions as “functions”
- Supports all the standard types of:
  - Arithmetic
  - Boolean logic
  - Conditionals/Flow control
- Support for pathological, optimized circuits via “intrinsics”
  - Similar in style to traditional compiler builtins/intrinsics
- Language is constructed to support very rapid design evaluation
- StoneCutter tools **compile** to Chisel HDL

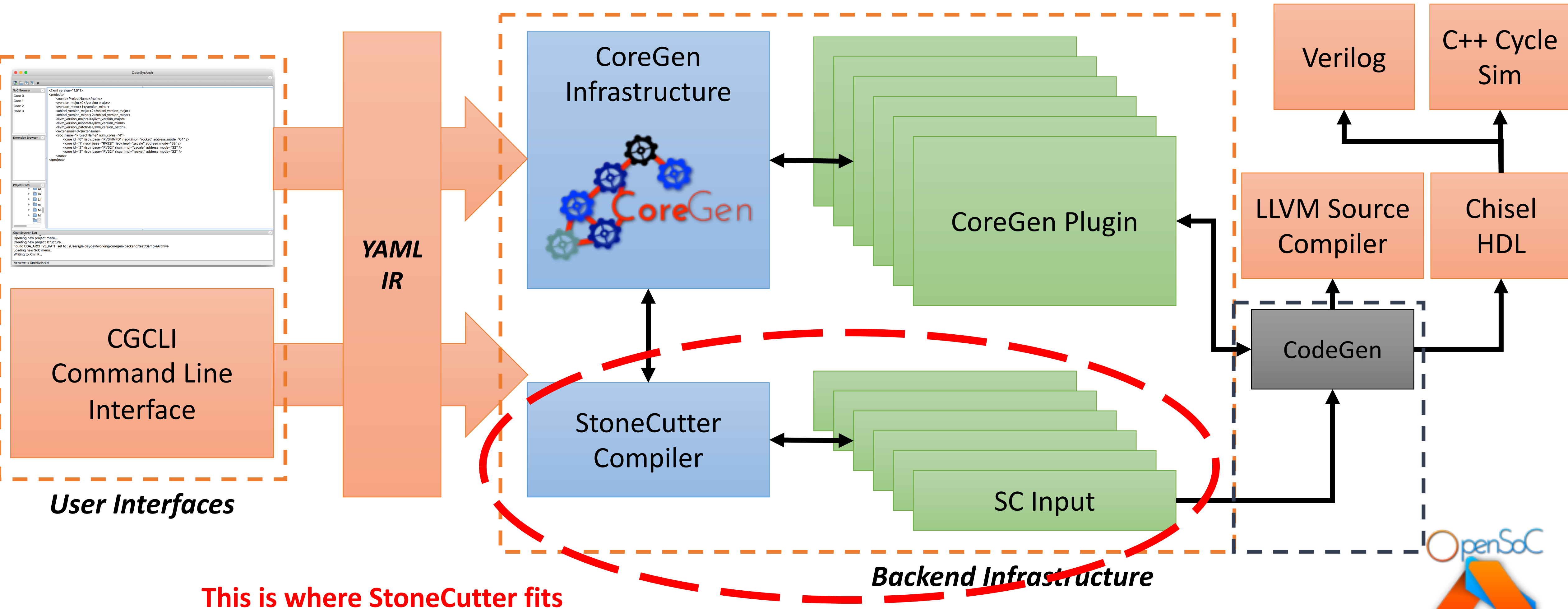
OpenSoC System Architect

What is StoneCutter NOT?

- StoneCutter is **not** the latest C-to-gates language
  - Does not compile directly to Verilog
- StoneCutter is **not** designed to implement the entire design
  - The language constructs are only designed to handle individual instructions
- StoneCutter still relies upon the user to utilize reasonable design concepts
  - StoneCutter has a number of safety passes in order to error/warn the user of erroneous or potentially slow paths
  - Is not guaranteed to produce optimized implementations from poor inputs
- StoneCutter does **not** have a notion of physical layout
  - We're compiling to Chisel HDL, not gates
- StoneCutter does **not** have explicit access to clocks
  - Clocks are exposed in the Chisel, but not StoneCutter

OpenSoC System Architect

System Architect Infrastructure

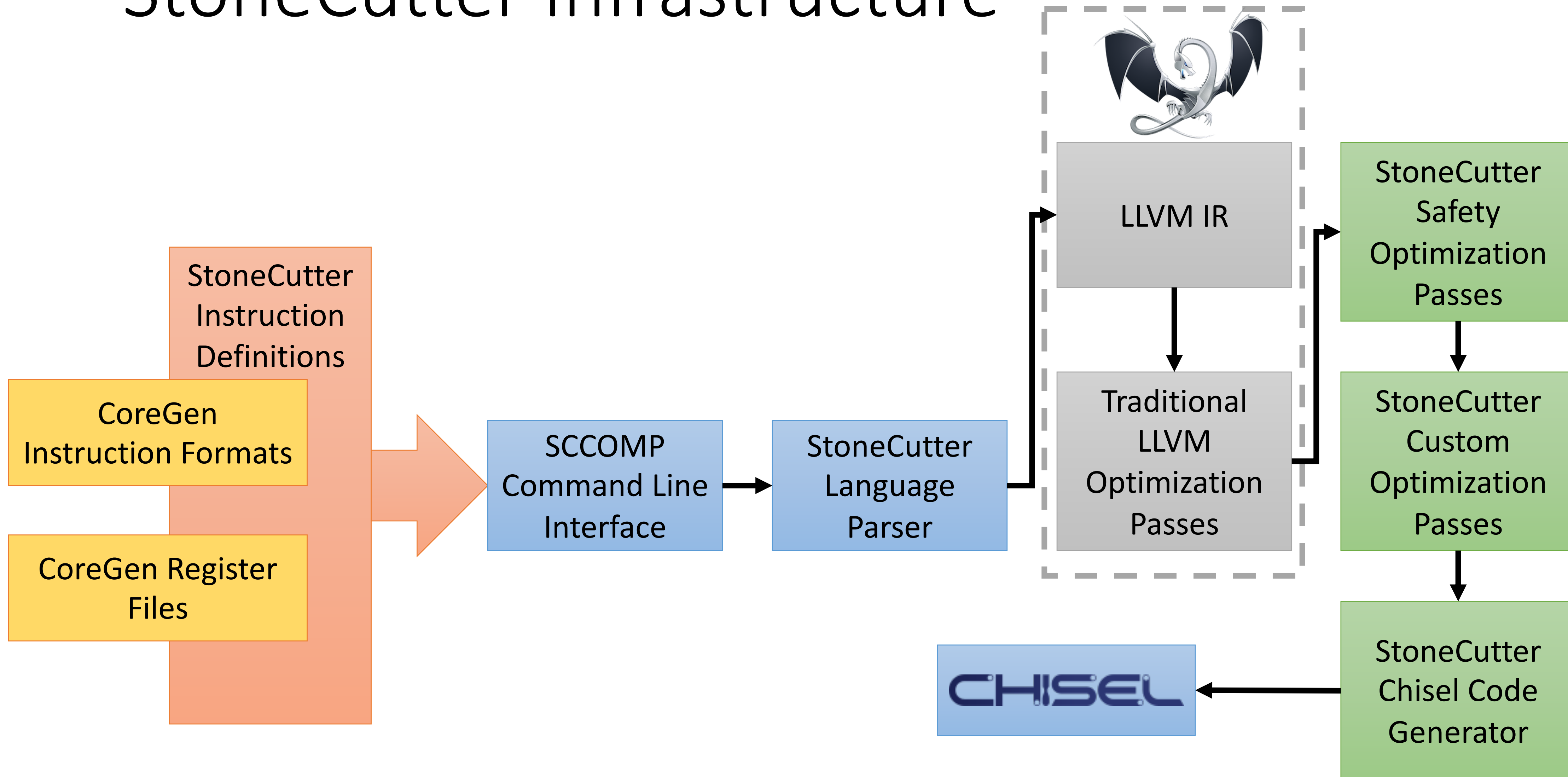


StoneCutter Infrastructure

- Users craft StoneCutter implementation files that represent their instructions
  - Each instruction implementation is a “function” with inputs and outputs
- StoneCutter implementation files are compiled via the StoneCutter compiler (*sccomp*)
- StoneCutter compiler is based upon LLVM
  - Custom language frontend & custom code generator
- Compiler utilizes mixture of traditional LLVM optimization passes and custom language passes
  - We encapsulate circuit-specific logic in our custom passes

OpenSoC System Architect

StoneCutter Infrastructure



OpenSoC System Architect

Example LLVM Optimization Passes

CFGSimplicationPass	LICMPass
• Performs dead code elimination and basic block merging • Removes basic blocks with no predecessors • Merges basic blocks with simple control flows • Eliminates PHI nodes for basic blocks with single predecessors • Eliminates basic blocks with only unconditional branches • https://llvm.org/docs/Passes.html#simplifycfg-simplify-the-cfg	• Performs "Loop Invariant Code Motion" • Attempts to remove code from loop bodies • "Hoists" or "Sinks" unnecessary code out of loops in order to minimize redundant operations • Will reduce downstream size of iterative loop circuits • https://llvm.org/docs/Passes.html#licm-loop-invariant-code-motion

OpenSoC System Architect

Example StoneCutter Safety Passes

FieldIO

- Walks all the statements that write to an output value
- Ensures that these output values are permissible output fields
- The following values/fields are ALWAYS read-only
  - Instruction encoding fields
  - Immediate value fields
- Flags the issues and halts compilation

IOWarn

- Examines the entire set of I/O statements in an instruction definition
- Determines if the user is performing "rogue" I/O's
- Rogue I/O's are reading/writing registers that aren't included in the instruction format
- The downstream effect is additional data paths in order to provide I/O capability to additional register files
- Warns user of rogue I/O's, but does not prevent code generation

OpenSoC System Architect

Example StoneCutter Optimization Passes

PipeBuilder	test

• Whole ISA pipeline optimization
• Examines all instruction implementations and records the set of operations required to implement the entire ISA
    • I/O's (sequential and parallel)
    • Arithmetic
    • Flow Control
• Constructs a corollary map across instructions in order to group like operations
• Required control signals are subsequently generated during code generation
• **Future language support for explicit pipeline construction
	• test

OpenSoC System Architect

StoneCutter Language Specification

- Language spec is governed in the same manner as source code development
  - Changes to the spec must be received in the form of pull requests on Github
  - Adjacent pull requests (that include all the necessary tests) must also exist in StoneCutter tree
  - NO changes to the spec are accepted without support in StoneCutter
- Entire language spec is documented with examples
- Latest revision:
  - http://www.systemarchitect.tech/index.php/stonecutter-language-spec/

OpenSoC System Architect

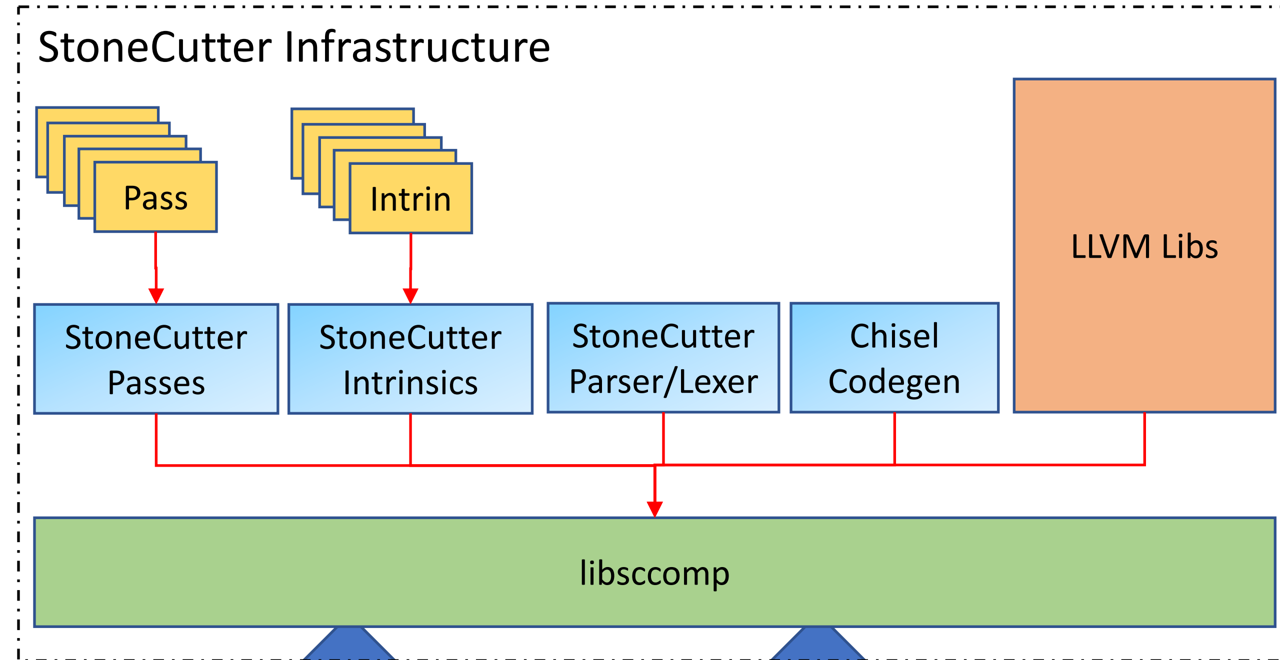
StoneCutter Tool Infrastructure

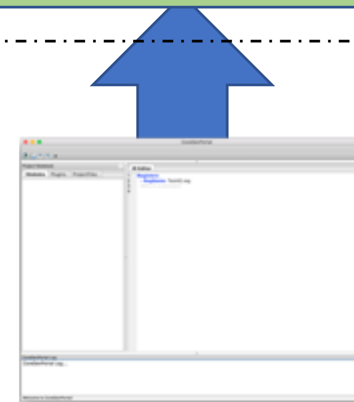
Tools/API Interfaces for StoneCutter

OpenSoC System Architect

StoneCutter Tool/API Infrastructure

- Infrastructure/compiler implemented as a compiler linked against LLVM libs
  - libSCComp
    - Language frontend
    - Custom passes
    - Intrinsics
    - Chisel code generation
- Entire API interface is documented via Doxygen:
  - https://codedocs.xyz/opensocsysarch/CoreGen/group__StoneCutter.html
- User-facing tools include a command line interface and GUI
  - Command Line: SCCOMP
  - GUI: CoreGenPortal





CoreGenPortal

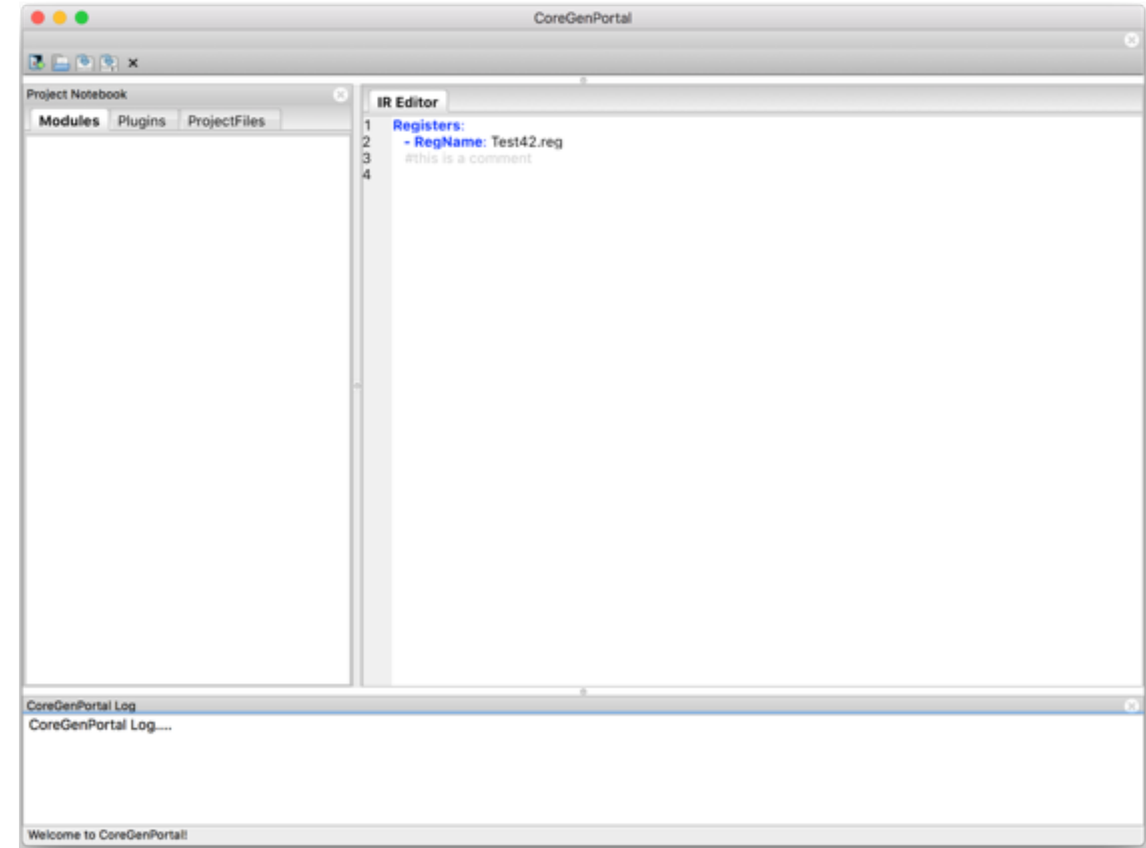
```
sccomp [Options] /path/to/input.sc
Options:
     -h|-help|--help                     : Print the help menu
     -k|-keep|--keep                     : Keep intermediate files
     -c|-chisel|--chisel                 : Generate Chisel output (default=on
     -p|-parse|--parse                   : Parse but do not compile
     -f|-outfile|--outfile /path/to/out  : Set the output file name
     -o|-object|--object                 : Generate target object file [*.o]
     -V|-version|--version               : Print the version info

Execution Options:
     -O|-optimize|--optimize             : Execute the optimizer
     -N|-no-optimize|--no-optimize       : Do not execute the optimizer
     -D|-disable-chisel|--disable-chisel : Disables Chisel output
     -v|-verbose|--verbose               : Enable verbosity

Optimization Pass Options:
     --list-passes                       : Lists all the LLVM passes
     --enable-pass "PASS1,PASS2"         : Enables individual passes
     --disable-pass "PASS1,PASS2"        : Disables individual passes
TCL-Lap:build jleidel$
```

SCCOMP

OpenSoC System Architect

Graphical Interface: *CoreGenPortal*

- CoreGenPortal is the primary graphical interface within System Architect
- Written in C++
- Graphics are handled via wxWidgets
  - Currently cross platform support for Linux (Ubuntu, CentOS) and Mac OSX
- Use cases
  - For those seeking a development environment that resembles traditional IDE's
  - For those unfamiliar/uncomfortable with command line tools



OpenSoC System Architect

Command Line Interface: *sccomp*

- Simple, concise command line interface to drive
  - Drives compilation/optimization
- Use cases:
  - For those who prefer to utilize the command line and write/modify StoneCutter using text editors
  - Run quick tests
  - Regression/CI environments to maintain designs

```
sccomp [Options] /path/to/input.sc
Options:
     -h|-help|--help                    : Print the help menu
     -k|-keep|--keep                    : Keep intermediate files
     -c|-chisel|--chisel                : Generate Chisel output (default=on
     -p|-parse|--parse                  : Parse but do not compile
     -f|-outfile|--outfile /path/to/out : Set the output file name
     -o|-object|--object                : Generate target object file [*.o]
     -V|-version|--version              : Print the version info

Execution Options:
     -O|-optimize|--optimize            : Execute the optimizer
     -N|-no-optimize|--no-optimize      : Do not execute the optimizer
     -D|-disable-chisel|--disable-chisel : Disables Chisel output
     -v|-verbose|--verbose              : Enable verbosity

Optimization Pass Options:
     --list-passes                      : Lists all the LLVM passes
     --enable-pass "PASS1,PASS2"        : Enables individual passes
     --disable-pass "PASS1,PASS2"       : Disables individual passes
TCL-Lap:build jleidel$
```


OpenSoC System Architect

SCCOMP Info Options

- --help : Prints the help menu
- --version : Prints the version info

```
$> sccomp --help
$> sccomp --version
```


OpenSoC System Architect

SCCOMP Execution Options

- Execution options require StoneCutter input input
  - sscomp /path/to/input.sc
- Four execution options:
  - --parse : parses & optimizes the input, but does not generate Chisel
  - --chisel : parses, optimizes and generates Chisel output
  - --object: parse, optimizes and generates LLVM bytecode output (utilized for debugging)
  - --keep: keep all the intermediate files (*.ll); combined with other execution options

```
$> sccomp --parse test.sc
$> sccomp --chisel test.sc
$> sccomp --object test.sc
$> sccomp --chisel --keep test.sc
```


OpenSoC System Architect

SCCOMP Execution Options cont.

- Additional options can be utilized
- --outfile /path/to/output.chisel : specifies output Chisel file
  - The default is “test.sc.chisel” in the same path as the input chisel file
- --optimize : enables the LLVM optimizer (enabled by default)
- --no-optimize : disables the LLVM optimizer
- --disable-chisel : disables the Chisel output
- --verbose : Enable verbosity during compilation

```
$> sccomp --chisel --outfile output.chisel test.sc
$> sccomp --chisel --optimize test.sc
$> sccomp --chisel --no-optimize test.sc
$> sccomp --disable-chisel --object test.sc
$> sccomp --verbose --chisel test.sc
```


OpenSoC System Architect

SCCOMP Optimization Options

- Users can list all the supported passes on the command line
- --list-passes: prints a table of all LLVM passes

```
TCL-Lap:build jleidel$ ./src/SCComp/sccomp --list-passes
 StoneCutter LLVM Optimization Passes
---------------------------------------------------------------------------------
 -      CFGSimplificationPass
 -      ConstantPropagationPass
 -      GVNPass
 -      IndVarSimplifyPass
 -      LICMPass
 -      LoopDeletionPass
 -      LoopIdiomPass
 -      LoopRerollPass
 -      LoopRotatePass
 -      LoopUnswitchPass
---------------------------------------------------------------------------------
```

```
$> sccomp --list-passes
```


OpenSoC System Architect

SCCOMP Optimization Options cont.

- Users can also enable/disable individual LLVM passes
- --enable-pass “PASS1,PASS2”
- --disable-pass “PASS1,PASS2”

```
TCL-Lap:build jleidel$ ./src/SCComp/sccomp --verbose --enable-pass "LICMPass" BasicRoot/RTL/stonecutter/BasicRISC.ISA.sc
Enabling LLVM Pass: LICMPass
Executing LLVM Pass: LICMPass
Executing Pass: InstArg
Executing Pass: InstFormat
Executing Pass: PipeBuilder
Executing Pass: IOWarn
Executing Pass: FieldIO
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=bra
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=br
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=br
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brac
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brac
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brac
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brc
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brc
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brc
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brc
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brr
```

```
$> sccomp --enable-pass “LICMPass” test.sc
```


OpenSoC System Architect

Intro to StoneCutter Syntax

Implementing Instructions in StoneCutter

OpenSoC System Architect

StoneCutter Syntax

- StoneCutter is a basic, C-like language construct that is designed to provide users a rapid development tool for individual instructions
- Designed to support compilation of very high level language to Chisel
- Designed to support users with reasonable knowledge of hardware architecture, but not Chisel (or Verilog) HDL
- Directly integrated into the remainder of the System Architect infrastructure
- See slide 13 for a reference to the official StoneCutter language specification

OpenSoC System Architect

StoneCutter Syntax cont.

- StoneCutter language syntax includes 10 distinct features

- Comments
- Datatypes
- Instruction Format Definitions
- Register Class Definitions
- Instruction Prototypes

- Variable Definitions
- Arithmetic Operations
- Conditional Operations
- Loop Operations
- Intrinsic Functions

OpenSoC System Architect

StoneCutter Syntax Notes

- **Semicolons**: In C, expressions are terminated by semicolons (;). Semicolons are not required in StoneCutter.
- **Complement Operations**: StoneCutter provides all the standard arithmetic, Boolean and logical operations provided by C. The one exception are complement operations (! or ~). StoneCutter does not have a native complement operations for binary or Boolean types. You must utilize the "NOT" intrinsic to complement either Boolean operations or binary operators

OpenSoC System Architect

StoneCutter Syntax: Comments

- Comments can be inserted into a StoneCutter implementation on new lines or inline with the source
- All comments are preceded by the ‘#’ sign

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


OpenSoC System Architect

StoneCutter Syntax: Datatypes

- StoneCutter supports a standard set of datatypes similar to C
- Unlike C, hardware designs need the ability to support types of arbitrary width
- The StoneCutter type system supports signed/unsigned types of any width {1-N bits}

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

```
u7 foo          #-- unsigned 7 bit integer

u1024 bar       #-- unsigned 1024 bit integer

s37 foobar      #-- signed 37 bit integer
```


OpenSoC System Architect

StoneCutter Syntax: Instruction Format Definitions

- StoneCutter requires that users define the instruction format utilized in the respective instructions
  - This is analogous to C-style prototypes
- Each field in the instruction format is treated as a global variable across all instruction definitions
  - These globals can be read from or written to just local any global variable
- Each field is designated as one of three types:
  - Encoding field: *enc*
    - Instruction encoding: Read-Only
  - Immediate field: *imm*
    - Immediate values: Read-Only
  - Register Index: *reg[REGCLASS]*
    - Register indices: Read-Write
- Register index fields require an additional syntactical note of defining the register class that the index is associated with
  - This allows us to performs I/O's to the correct register file at the correct index
  - See Register Class Definitions
- *Note*: The field names should map back to the same naming conventions utilized in the CoreGen IR definition

```
instformat FORMATNAME( FIELDTYPE FIELD1, FIELDTYPE FIELD2, … )
```

StoneCutter Syntax: Instruction Format Definitions cont.

- Using the *Arith.if* format from the Level 1 tutorial, we can define our instruction format in StoneCutter
- Six fields:
  - Register Fields: {ra, rb, rt}
    - Notice that each utilizes the GPR register file
  - Encoding Fields: {opc, func}
  - Immeidate Field: {imm}

```
instformat Arith.if(reg[GPR] ra, reg[GPR] rb, reg[GPR] rt, enc opc, enc func, imm imm)
```


```yaml
- InstFormatName: Arith.if
  ISA: BasicRISC.ISA
  FormatWidth: 32
  Fields:
    - FieldName: ra
      FieldType: CGInstReg
      FieldWidth: 5
      StartBit: 0
      EndBit: 4
      MandatoryField: false
      RegClass: GPR
    - FieldName: rb
      FieldType: CGInstReg
      FieldWidth: 5
      StartBit: 5
      EndBit: 9
      MandatoryField: false
      RegClass: GPR
    - FieldName: rt
      FieldType: CGInstReg
      FieldWidth: 5
      StartBit: 10
      EndBit: 14
      MandatoryField: false
      RegClass: GPR
    - FieldName: opc
      FieldType: CGInstCode
      FieldWidth: 5
      StartBit: 15
      EndBit: 19
      MandatoryField: true
    - FieldName: func
      FieldType: CGInstCode
      FieldWidth: 5
      StartBit: 20
      EndBit: 24
      MandatoryField: true
    - FieldName: imm
      FieldType: CGInstImm
      FieldWidth: 7
      StartBit: 25
      EndBit: 31
      MandatoryField: false
```


OpenSoC System Architect

StoneCutter Syntax: Register Class Definitions

- Users must also define the set of register classes contained within the design
- Each register class and register is raised to a global variable across all instruction definitions
- Each register must be encapsulated in a register class
- Each register must include a datatype
  - These are the datatypes defined in slide 28
- Users can create register files with mixed data types (but performance and/or area may suffer)
- Register class and register names must map back to the names defined in your CoreGen IR file

```
regclass RCNAME( DATATYPE RegName1, DATATYPE RegName2, … )
```


OpenSoC System Architect

StoneCutter Syntax: Register Class Definitions cont.

- Using the *GPR* register file from our Level 1 tutorial, we can define a register class in StoneCutter
- 32 general purpose registers
- All are defined as unsigned 64 bit integers

```
# ------------------------------------------------------------
# Register Class Section
# ------------------------------------------------------------
RegClasses:
  - RegisterClassName: GPR
    Registers:
      - r0
      - r1
      - r2
      - r3
      - r4
      - r5
      - r6
      - r7
      - r8
      - r9
      - r10
      - r11
      - r12
```

```
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 )
```


OpenSoC System Architect

StoneCutter Syntax: Instruction Prototypes

- Each instruction implementation requires an instruction *definition*
- The instruction definition is analogous to a C-function that defines the respective implementation of the instruction
- Each instruction implementation requires an instruction prototype
- Prototypes provide several key implementation elements:
    - Instruction naming convention that matches the decoding/instruction crack logic
    - The instruction format for the instruction that provides the compiler the ability to validate the I/O structure
    - The instruction arguments in the form of instruction fields, register classes or registers that define the pipelined I/O structure

```
def INSTNAME[:INSTFORMAT]( ARG1 ARG2 ARG3 … ){
}
```


OpenSoC System Architect

StoneCutter Syntax: Instruction Prototypes cont.

- The *INSTNAME* from each instruction definition must match the complementary instruction name defined in the CoreGen IR
- The *INSTFORMAT* is an optional syntactical addition to the instruction definition, however
  - Defining the instruction format allows StoneCutter to correctly validate the I/O pipeline
  - Will prevent downstream issues in creating erroneous I/O paths
  - The instruction format must match an existing format defined in the instruction format block and the CoreGen IR

```
def INSTNAME[:INSTFORMAT]( ARG1 ARG2 ARG3 … ){
}
```

StoneCutter Syntax: Instruction Prototypes cont.

- The instruction argument list provides the set of standard I/O paths for the target function
- The argument lists can include any global variables:
  - Register classes
  - Individual registers
  - Instruction fields
- *Note:* Most instructions will include the register and immediate fields of the respective register format
  - These are things that are commonly utilized in the instruction body
- *Note*: The instruction arguments are NOT separated by commas

```
def INSTNAME[:INSTFORMAT]( ARG1 ARG2 ARG3 … ){
}
```


OpenSoC System Architect

StoneCutter Syntax: Instruction Prototypes cont.

- Using the basic ADD instruction from the Level 1 tutorial, we can define a simple prototype
- Note how we define the instruction to utilize the *Arith.if* format and utilize the {*ra,rb,rt*} register fields as well as the *imm* field from the instruction format

```
# ------------------------------------------------------
# Instruction Section
# ------------------------------------------------------
Insts:
# -- Integer Arithmetic
  - Inst: add
    ISA: BasicRISC.ISA
    InstFormat: Arith.if
    Encodings:
      - EncodingField: opc
        EncodingWidth: 5
        EncodingValue: 0
      - EncodingField: func
        EncodingWidth: 5
        EncodingValue: 0
      - EncodingField: imm
        EncodingWidth: 7
        EncodingValue: 0
```

```
def add:Arith.if( ra rb rt imm ){
        #-- this is the instruction body
}
```


OpenSoC System Architect

StoneCutter Syntax: Variable Definitions

- Variables define permanent and temporary storage for use by the instruction implementation
- Variables can be global or local (just as in C)
- Global variables are defined within the instruction format and register classes
- Local variables are defined *first* in the instruction implementation body
  - Just like in C where variables are defined at the top of the function
  - Local variable scope is the entire instruction body
- Local variables must include an associated datatype (Slide 28)
  - Local variables can be initialized with specific values
  - Multiple variables with the same type can be defined together

```
def INST:INSTFORMAT( … ) {
        DATATYPE var1
        DATATYPE var2 = VALUE
        DATATYPE var3, var4, …
}
```


OpenSoC System Architect

StoneCutter Syntax: Variable Definitions cont.

- Variables define permanent and temporary storage for use by the instruction implementation
- Global variables reside within register file storage
- Local variables generally reside within the pipeline
- Note: Its imperative that you make good choices with respect to the datatypes of your local variables
  - StoneCutter will always attempt to do the "right thing" with respect to the values present in the operation
  - Maintaining correct datatypes for the desired operation width will ensure numerical stability
- Take the following example to perform a fused multiple add operation
  - **This is an incredibly simple example
  - Can also be done using rt = ra + (rb * rt)

```
def fadd:Arith.if( ra rb rt imm ){
        u64 tmp1 = 0
        tmp1 = rb * rt
        rt = ra + tmp1
}
```


OpenSoC System Architect

StoneCutter Syntax: Variable Definitions cont.

- There are a number of special case global variable definitions
  - Reading/writing these variables induces special backend I/O circuits
- Registers: directly reading/writing named registers will directly read/write these registers within their target register files
  - Registers must be defined within a register class object
- Register Fields: reading/writing register fields will manipulate the register at the index referenced by the register field
  - RegisterClass[Field] = Value

```
pc = 10  #-- set the value of the pc register to 10

rt = 15   #-- write the value 15 to the register at the index in the RT field
```


OpenSoC System Architect

StoneCutter Syntax: Arithmetic Operations

- The StoneCutter language supports the full complement of C arithmetic operations in standard form:
  - *Output = Input <operator> Input*
- As mentioned earlier, the only exception to this is the Boolean and bitwise complement operations (!,~)
- The target (output) of all operations must be writable entities (variables or register fields)

```
u16 tmp1 = 0
rt = ra + rb        #-- legal
tmp1 = ra * rb      #-- legal
imm = rt            #-- ILLEGAL, "imm" is a read-only instruction field
```


OpenSoC System Architect

StoneCutter Syntax: Arithmetic Operations cont.

- StoneCutter has a unique feature in that users can perform operations with inputs of any type
- StoneCutter will attempt to up/down convert based upon the type of the output operand
- Any time operations are upconverted, the extended bit space must be considered volatile. You MUST utilize the sign and zero extension intrinsics in order to maintain numerical stability
  - u64 = u32 + u32
- Any time operations are truncated, the least significant N bits are always taken
  - u32 = u64 + u64

```
u64 tmp1, tmp2, tmp3
u32 tmp4, tmp5
tmp1 = tmp2 + tmp4   #-- tmp4 is extended to 64 bits
tmp4 = tmp1  + tmp3  #-- result is truncated to 32 bits (31:0)
tmp3 = tmp1 << tmp2 #-- result is still truncated to 64 bits
```


OpenSoC System Architect

StoneCutter Syntax: Arithmetic Operations cont.

<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 nor 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>

StoneCutter Syntax: Conditional Operations

- The StoneCutter language supports the use of conditional expressions
- These conditional flow control expressions mimic traditional *if-else* statements in C
  - The else block is optional
- The expression can be any set or combination of traditional Boolean operations
- The conditional operations are contained within brackets {}

```
if( BOOLEAN OPERATION ){
        #-- Conditional Body
}

if( BOOLEAN OPERATION ){
}else{
}
```


OpenSoC System Architect

StoneCutter Syntax: Conditional Operations cont.

- The Boolean expressions can compare any combination of local variables, global variables and immediate values
  - *VAR <BOOLEAN OPERATOR> VAR*
- Complex operations can be contained within parenthesis

```
if( (RA >RB) && (RB < RC) || (RA == RD) ){
        #-- Conditional Body
}
```

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

OpenSoC System Architect

StoneCutter Syntax: Loop Operations

- StoneCutter supports a full set of *for, while* and *do-while* loop constructs
- The syntax is analogous to traditional C-like loops
  - Several nuances in the construction of for loops
- The Boolean expressions utilized to construct the loop trips can be constructed using any Boolean operator
  - See slide 44
- All loop bodies are contained within brackets (mandatory)

```
while( Boolean expression ){
}
```

```
for( looptrip = base; looptrip <BOOLEAN OPERATOR> terminator ){
}
for( looptrip = base; looptrip <BOOLEAN OPERATOR> terminator; iterator ){
}
```

```
do{
}while( Boolean expression )
```

StoneCutter Syntax: For Loop Operations

- The *for* loop expression requires two statements in the loop expression with an optional third statement
- Statement 1: loop trip base case
  - Can be an existing variable or a new variable
  - New variables will automatically be defined as new locals
- Statement 2: loop termination statement
  - Utilizes all the traditional Boolean operators (<,>,<=,>=)
- Statement 3: optional iterator
  - If iterators are not specified, then the loop is automatically assumed to add "1" to for each loop trip
  - Note that the loop iterators **MUST** be standard arithmetic operators
  - StoneCutter **DOES NOT** support "++", "--", "+="

```
#-- RA, RB, RC are existing globals
for( RA = 1; RA < RB ){
        #-- RA incremented by 1
}

for( RA = 20; RA > RC; RA-1 ){
        #-- RA decremented by 1
}
```

```
#-- 'i' becomes a new local
for( i = 1; i < RB; i+1 ){
        #-- i incremented by 1
}
```


OpenSoC System Architect

StoneCutter Syntax: While Loop Operations

- The *while* loop expression requires one statement in the loop expression
  - Loop terminator statement
- While loops require that the user perform any necessary modifications to the loop trip counter within the loop body in order to avoid infinite loops

```
while( RA < RB ){
    #-- loop body
    RA = RA + 1
}
```


OpenSoC System Architect

StoneCutter Syntax: Do-While Loop Operations

- The *do-while* loop expression requires one statement in the loop expression
  - Loop terminator statement
- Do-While loops require that the user perform any necessary modifications to the loop trip counter within the loop body in order to avoid infinite loops

```
do{
        #-- loop body
        RA = RA + 1
}while( RA < RB )
```


OpenSoC System Architect

StoneCutter Syntax: Intrinsic Functions

- Much like other C/C++ language constructs, StoneCutter provides a set of builtin *intrinsic* functions
- Intrinsic functions are utilized to implement pathological operations in a high performance manner
- The intrinsic logic is expanded inline within the remainder of the StoneCutter source
- StoneCutter intrinsics behave like function calls
  - Some intrinsics have return values, several modify data in line
  - Most intrinsics require arguments
  - StoneCutter will perform argument checking for every included intrinsic
- See Section 3 in the StoneCutter Language Spec

```
#-- intrinsic with a return value
RA = INTRINSIC( ARG1, ARG2, … )

#-- inline intrinsic
INTRINSIC( ARG1, ARG2, … )
```


OpenSoC System Architect

StoneCutter Syntax: Intrinsic Functions cont.

- Two types of StoneCutter intrinsics: *arithmetic* and *memory*
- Arithmetic Intrinsics
  - Perform some notional permutation on the input arguments
  - Returns the result of the permutation
    - A = INTRIN(B)
- Memory Intrinsics
  - Perform loads/stores to/from memory
  - Links the instruction to the memory pipeline
  - Load operations return data
    - A = LOAD(B)
  - Store operations do not return data
    - STORE(DATA, ADDR)

```
#-- LOAD intrinsic with a return value
RA = LOAD( ARG1 )

#-- STORE intrinsic with no return value
STORE( RA, RB )
```

StoneCutter Syntax: Intrinsic Arguments

- Unlike C-style intrinsics, StoneCutter intrinsics are *typeless*
- Each intrinsic can accept any argument of any type or bit width
- The compiler expands the intrinsic using the appropriate input/output data types with the correct internal logic
- Note: Intrinsic functions are very literal! Be sure to understand how specific intrinsic functions react to certain input types. Pay special attention to the expected output type.

```
u8 var1 = 1234
u64 var2 = 123456789

RA = POPCOUNT( var1 )
RB = POPCOUNT( var2 )
```


OpenSoC System Architect

Implementing a Basic RISC Device

Implementing a basic RISC device with System Architect and StoneCutter

OpenSoC System Architect

Tutorial Source

- Tutorial source is published in a Github repository
- All design source code is open source under an Apache2 license
  - Feel free to reuse it!!
- https://github.com/opensocsysarch/CoreGenTutorials
  - See the LEVEL2 subdirectory

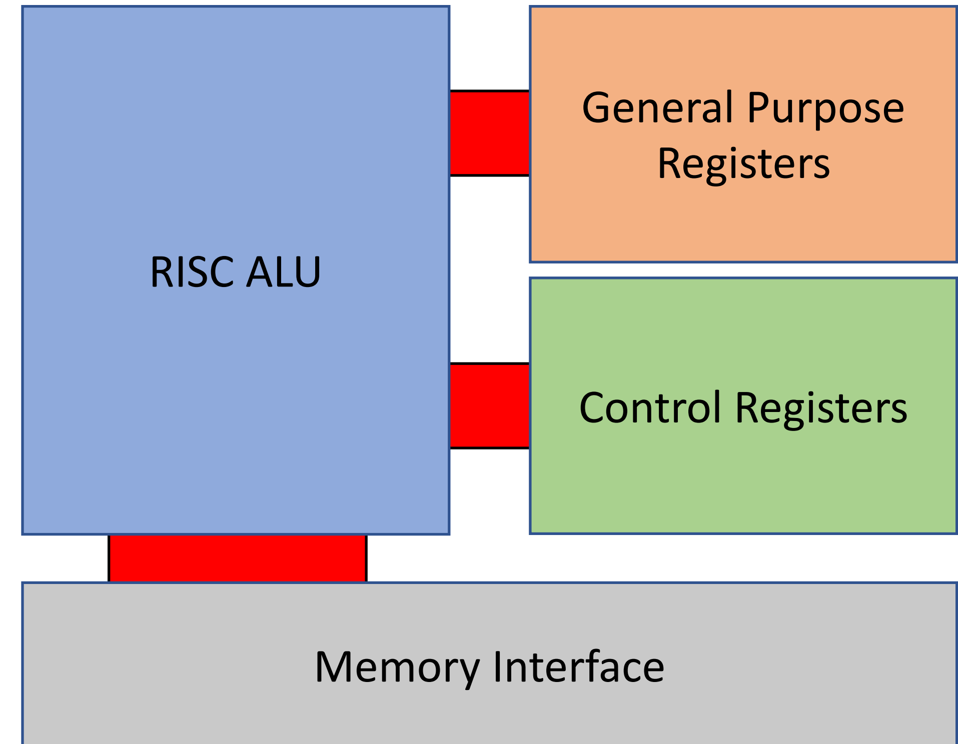
OpenSoC System Architect

Tutorial Assumptions

- Standard installation location:
  - “/opt/coregen”
  - EG, the “sccomp” binary will be located at /opt/coregen/bin/sccomp
  - We don’t explicitly reference the fully qualified path in the tutorial
- Text editing is required!
  - Emacs and Vim are most prevalent, but any standard text editor will suffice
- Basic command line knowledge is required
  - Executing commands with arguments
- Basic knowledge of Git/Github
  - Only required if you seek to download/edit the tutorial content

BasicRISC Core

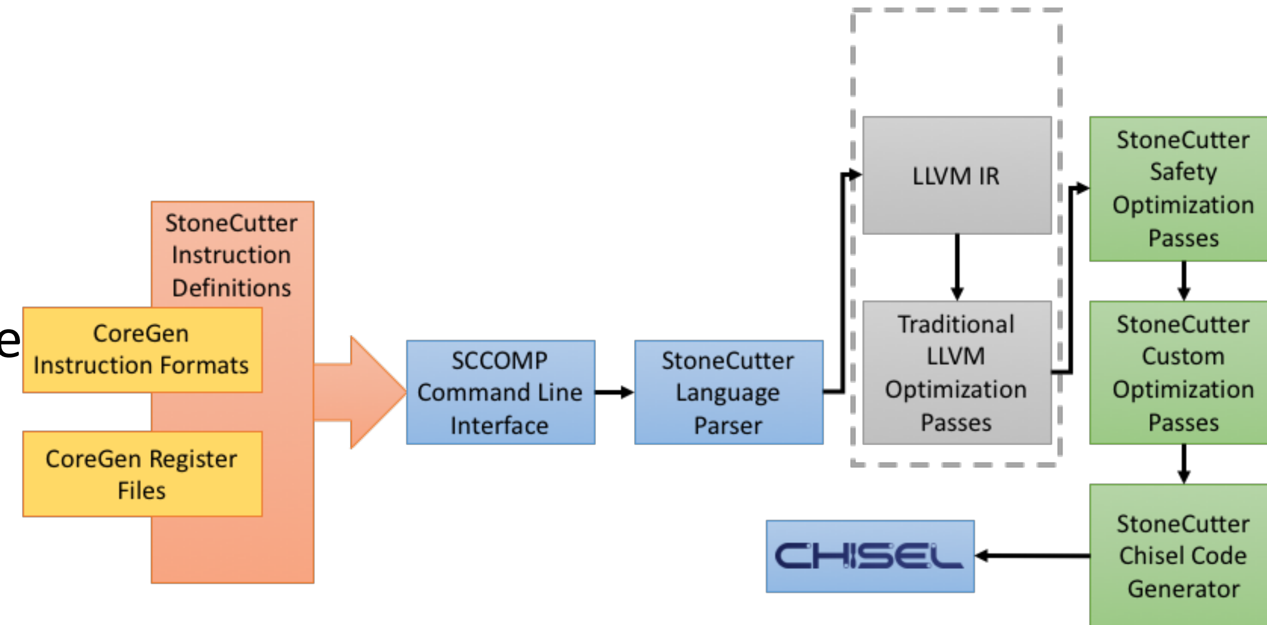
- The remainder of this Level 2 tutorial will build upon the Level 1 tutorial material
- We will extend the original Level 1 design with the necessary instruction implementations
- This will include all the instructions defined in our BasicRISC Core



OpenSoC System Architect

CoreGen and StoneCutter

- One of the nice features of CoreGen is the ability to directly integrate the StoneCutter syntax into the CoreGen IR design files
- This allows you to *inline* the instruction implementation in the design
- CoreGen autogenerates all the instruction format and register class blocks without user intervention
- Permits users to focus on the specific instruction implementations
  - Reduces the ramp to become proficient with StoneCutter
- All of our work will be done in the CoreGen IR file from the Level 1 tutorial



OpenSoC System Architect

BasicRISC ISA

- Traditional RISC ISA
  - Opcodes (opc) determine the “class” of instructions
  - Function codes (func) determine the target instruction
  - Instructions are grouped by their argument types:
    - INST GPR, GPR, GPR
    - INST GPR, CTRL, GPR
    - INST CTRL, GPR, GPR
  - Plenty of opcode/function space to expand for your own use

- Arithmetic:
  - Integer arithmetic (2’s-complement)
  - Add, Sub, Mul, Div
  - Logical/Arithmetic shifts
  - Logicals (AND, OR, NAND, NOR, XOR, NOT)
- Comparisons:
  - Compare {NE, EQ, GT, LT, GTE, LTE}
- Branches
  - Conditional and unconditional
  - Absolute and relative (jump)

See *BasicRISCInstTable* for a full instruction set listing

Directly Editing CoreGen Yaml IR

- Yaml IR is ASCII text
- Hierarchy is determined by indentions
  - Indentions are SPACES, not tabs
  - Each indentation should be two (2) spaces
- You can use any potential editor!
- A few important notes:
  - Nodes are parsed in the correct order regardless of their order in the file
    - We do this to preserve the natural hierarchy and dependence between nodes
  - Node names are case sensitive
    - "RegName" != "Regname"
  - Certain nodes have required and optional attributes
    - Refer to the IR documentation for what is optional
  - Comments are delineated with '#' characters
    - Similar to BASH shell scripts

Example CoreGen Yaml IR Formatting

```
#-- this is a comment
NODE:
 - SubNode: Name1
   Attribute1: 64
   Attribute2: false
   Attribute3: This_Is_A_String
 - SubNode: FOO
   Bars:
     - bar0
     - bar1
     - bar2
```

OpenSoC System Architect

Ten Design Steps for Level 2

- ***Step 1***: Copy the CoreGen Yaml IR file from the Level 1/Step 10 directory
- ***Step 2***: Update the project definition for our Level 2 directory structure
- ***Step 3***: Implement the arithmetic instructions
- ***Step 4***: Implement the comparison instructions
- ***Step 5***: Implement the load/store instructions
- ***Step 6***: Implement the logical NOT instruction
- ***Step 7***: Implement the branch instructions
- ***Step 8***: Implement the control instructions
- ***Step 9***: Compile the StoneCutter source

The CoreGen IR for each step is outlined in **~/CoreGenTutorials/LEVEL2/Step*N***

OpenSoC System Architect

Step 1: Copy the basic project files

- The first step in the Level 2 tutorial is to copy over the CoreGen design input from Step10 of the Level1 tutorial
- We will build upon the work done in the Level 1 tutorial material

```
$> cd ~/CoreGenTutorials/LEVEL2/Step1

$> cp ../LEVEL1/Step10/BasicRISC.yaml ./
```

~/CoreGenTutorials/LEVEL2/Step1

OpenSoC System Architect

Step 2: Update the project definition

- With the Level 2 tutorial, we need to define the project structure
- The project structure will be utilized to generate all the necessary directory structure and intermediate makefiles
- Edit the BasicRISC.yaml file and specify a new *ProjectRoot* directory
  - This will place all our generated files in the *./BasicRISC* directory

```
# ----------------------------------------------------
# ProjectInfo Section
# ----------------------------------------------------
ProjectInfo:
  - ProjectName: BasicRISC
    ProjectRoot: ./BasicRISC
    ProjectType: soc
    ChiselMajorVersion: 3
    ChiselMinorVersion: 0
```

~/CoreGenTutorials/LEVEL2/Step2

OpenSoC System Architect

Step 3: Implement the arithmetic instructions

- In this stage of the tutorial, we will begin implementing instructions
- Rather than implementing our StoneCutter source code by hand, we will utilize a unique feature of CoreGen to assist our development process
- CoreGen can directly include inline StoneCutter language syntax in the design input (YAML)
  - Users define the body of each instruction
  - CoreGen will automatically generate all the instruction formats, register class and instruction prototypes
- Each instruction block in the YAML input will require a new attribute: *Impl*

Impl: Insert StoneCutter syntax here

~/CoreGenTutorials/LEVEL2/Step3

OpenSoC System Architect

Step 3: Implement the arithmetic instructions

- In this stage of the tutorial, we will begin implementing instructions
- Rather than implementing our StoneCutter source code by hand, we will utilize a unique feature of CoreGen to assist our development process
- CoreGen can directly include inline StoneCutter language syntax in the design input (YAML)
  - Users define the body of each instruction
  - CoreGen will automatically generate all the instruction formats, register class and instruction prototypes
- Each instruction block in the YAML input will require a new attribute: *Impl*
  - *Impl* attributes may include any syntactical nuances, including newlines, brackets, etc

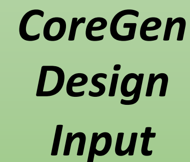
```
Insts:
- Inst: add
    ISA: BasicRISC.ISA
    InstFormat: Arith.if
    Encodings:
      - EncodingField: opc
        EncodingWidth: 5
        EncodingValue: 0
      - EncodingField: func
        EncodingWidth: 5
        EncodingValue: 0
      - EncodingField: imm
        EncodingWidth: 7
        EncodingValue: 0
    Impl: Insert StoneCutter syntax here
```

~/CoreGenTutorials/LEVEL2/Step3

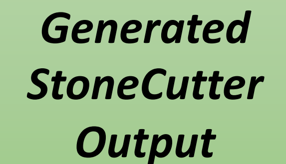
OpenSoC System Architect

Step 3: Implement the arithmetic instructions cont.

```
Insts:
- Inst: add
   ISA: BasicRISC.ISA
   InstFormat: Arith.if
   Encodings:
    - EncodingField: opc
      EncodingWidth: 5
      EncodingValue: 0
    - EncodingField: func
      EncodingWidth: 5
      EncodingValue: 0
    - EncodingField: imm
      EncodingWidth: 7
      EncodingValue: 0
   Impl: rt = ra + rb
```




```
instformat Arith.if(reg[GPR] ra,reg[GPR] 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, 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 )

def add( ra rb rt imm )
{
rt = ra + rb
}
```



~/CoreGenTutorials/LEVEL2/Step3

OpenSoC System Architect

Step 3: Implement the arithmetic instructions cont.

- Arithmetic instructions we need to implement:

• add • sub • mul • div • divu • sll	• srl • sra • and • or • nand • nor • xor

```
add: rt = ra + rb
sub: rt = ra - rb
mul: rt = ra * rb
div: rt = ra / rb
divu: rt = ra / rb
sll: rt = ra << rb
srl: rt = ra >> rb
sra: rt = ra >> rb
and: rt = ra & rb
or: rt = ra | rb

#-- note how we use intrinsics
nand:  rt = NOT(ra & rb)
nor: rt = NOT(ra | rb)

xor: rt = ra ^ rb
```

~/CoreGenTutorials/LEVEL2/Step3

Step 4: Implement the comparison instructions

- Using the same technique, we need to implement our comparison instructions that utilize if/else clauses
- Instructions:
  - cmp.ne
  - cmp.eq
  - cmp.gt
  - cmp.lt
  - cmp.gte
  - cmp.lte

```
cmp.ne: if( ra != rb ){ rt = 2 }else{ rt = 0 }

cmp.eq: if( ra == rb ){ rt = 3 }else{ rt = 0 }

cmp.gt: if( ra > rb ){ rt = 4 }else{ rt = 0 }

cmp.lt: if( ra < rb ){ rt = 5 }else{ rt = 0 }

cmp.gte: if( ra >= rb ){ rt = 6 }else{ rt = 0 }

cmp.lte: if( ra <= rb ){ rt = 7 }else{ rt = 0 }
```

~/CoreGenTutorials/LEVEL2/Step4

Step 5: Implement the load/store instructions

- Using the same techniques, we need to define our load/store instructions
  - Note that we will utilize memory intrinsics to enable the memory pipeline

• lb • lh • lw • ld • sb • sh • sw • sd	• lbu • lhu • lwu • sbu • shu • swu

```
lb: rt = SEXT(LOADELEM(ra+imm,8),7)
lh: rt = SEXT(LOADELEM(ra+imm,16),15)
lw: rt = SEXT(LOADELEM(ra+imm,32),31)
ld: rt = LOADELEM(ra+imm,64)
sb: STOREELEM(SEXT(ra,7),rt+imm,8)
sh: STOREELEM(SEXT(ra,15),rt+imm,16)
sw: STOREELEM(SEXT(ra,31),rt+imm,32)
sd: STOREELEM(ra,rt+imm,64)
lbu: rt = ZEXT(LOADELEM(ra+imm,8),7)
lhu: rt = ZEXT(LOADELEM(ra+imm,16),15)
lwu: rt = ZEXT(LOADELEM(ra+imm,32),31)
sbu: STOREELEM(ZEXT(ra,7),rt+imm,8)
shu: STOREELEM(ZEXT(ra,15),rt+imm,16)
swu: STOREELEM(ZEXT(ra,31),rt+imm,32)
```

~/CoreGenTutorials/LEVEL2/Step5

OpenSoC System Architect

Step 6: Implement the logical NOT instruction

- Using the same techniques, we can now implement the logical NOT instruction
  - Two operand instruction
  - Utilizes the NOT intrinsic to implement the only valid input argument

```
not: rt = NOT(ra)
```

~/CoreGenTutorials/LEVEL2/Step6

OpenSoC System Architect

Step 7: Implement the branch instructions

- Using the same techniques, we can now implement the branch instructions
- These are unique in that they utilize mixtures of explicit registers (pc, rp) as well as register inputs from the instruction payload
  - We also utilize if/else expressions for conditional branches

- bra
- br
- brac
- brc

```
bra: pc = rt

br: pc = pc + r)

brac: if( ra == rb ){ pc = rt }else{ pc = pc + 4 }

brc: if( ra == rb ){ pc = pc + rt }else{ pc = pc + 4 }
```

~/CoreGenTutorials/LEVEL2/Step7

OpenSoC System Architect

Step 8: Implement the control instructions

- Using the same techniques, implement our move to/from control instructions
- Note that we utilize the traditional RISC method of doing so: add pipeline
- Instructions
  - ladd
  - cadd

ladd: rt = ra + rb

cadd: rt = ra + rb

~/CoreGenTutorials/LEVEL2/Step8

OpenSoC System Architect

Step 9: Compile the StoneCutter source

- Now that we have our entire ISA implementation defined, we must perform two steps to generate the Chisel output

1. Execute the CoreGen code generator to construct the directory structure and our master StoneCutter file
2. Compile the StoneCutter master implementation file

~/CoreGenTutorials/LEVEL2/Step9

OpenSoC System Architect

Step 9: Compile the StoneCutter source cont.

- Ensure that the CGCLI binary is in your path
- Execute the CoreGen code generator in order to generate the necessary directory structure and the master StoneCutter source file

- This will generate the ./BasicRISC/ directory
- Your StoneCutter source file will reside in ./BasicRISC/RTL/stonecutter/BasicRISC.ISA.sc
  - The file name is inherited from the name of the ISA from the YAML input

```
$> cgcli --chisel --ir BasicRISC.yaml
```

~/CoreGenTutorials/LEVEL2/Step9

OpenSoC System Architect

Step 9: Compile the StoneCutter source cont.

```
#-- 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
}
```

Instruction Formats

Register Classes

Instruction Definitions

OpenSoC System Architect

Step 9: Compile the StoneCutter source cont.

- Now that we have the full StoneCutter source, we can compile it to Chisel
- Ensure that the SCCOMP binary is in your path
- Note: we will see some warnings from the StoneCutter compiler
  - But why!?

```
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=bra
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=br
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=br
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brac
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brac
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brc
WARNING : IOWarn : Detected rogue I/O operation to register file for variable=pc in instruction=brc
```

```
$> sccomp --chisel ./BasicRISC/RTL/stonecutter/BasicRISC.ISA.sc
```

~/CoreGenTutorials/LEVEL2/Step9

Step 9: Compile the StoneCutter source cont.

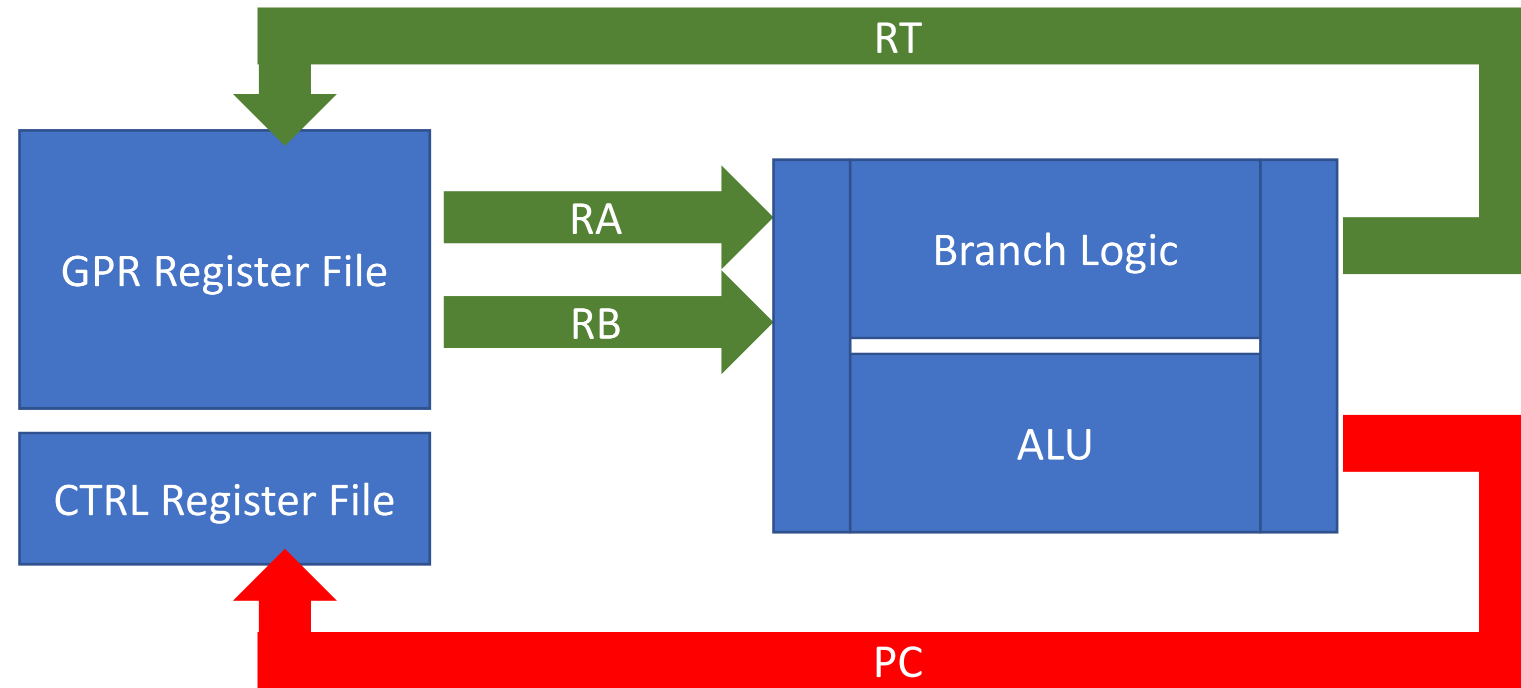
- As mentioned in the intro, StoneCutter includes a set of safety and performance optimization passes
- The safety passes analyze the structure of the instruction implementation in order to verify whether the implementation is safe and/or induces unwanted performance idiosyncrasies
- Optimization passes modify the source in a safe manner in order to improve performance
  - Both of the arithmetic pipeline and the I/O channels

- The warnings we saw when compiling our source come from the IOWarn pass
- The IOWarn pass analyzes the use of all global variables in order to ensure that the instruction format has predefined I/O paths for the appropriate variables
  - Reading/Writing registers outside of these paths will force the StoneCutter compiler to generate additional paths, which may have a negative effect on performance
- The branch instructions flagged by the warning exhibit this case
  - The branch instructions are of type Arith.if, which only include paths for GPR registers
  - The PC register is a CTRL register, thus an additional path is generated
  - Given that branches are known to be latent instructions, this is ok for our design

~/CoreGenTutorials/LEVEL2/Step9

OpenSoC System Architect

Step 9: Compile the StoneCutter source cont.



Standard I/O Path

Additional I/O Path

The IOWarn pass detects the additional required I/O path and warns the user. This is NOT an error, just a warning

~/CoreGenTutorials/LEVEL2/Step9

OpenSoC System Architect

References

Where do I find more info?

OpenSoC System Architect

Web Links

- System Architect Public Web
  - http://www.systemarchitect.tech/
- Documentation
  - Latest StoneCutter Specification:
    - http://www.systemarchitect.tech/index.php/stonecutter-language-spec/
- Tutorials
  - http://www.systemarchitect.tech/index.php/tutorials/
  - https://github.com/opensocsysarch/CoreGenTutorials

OpenSoC System Architect

Source Code

- Main source code hosted on Github:
  - https://github.com/opensocsysarch

- CoreGen Infrastructure
  - https://github.com/opensocsysarch/CoreGen
- CoreGenPortal GUI
  - https://github.com/opensocsysarch/CoreGenPortal
- CoreGen IR Spec
  - https://github.com/opensocsysarch/CoreGenIRSpec
- StoneCutter Language Spec
  - https://github.com/opensysarch/StoneCutterLanguageSpec
- System Architect Weekly Development Releases
  - https://github.com/opensocsysarch/SystemArchitectRelease

OpenSoC System Architect

Contact

- Issues should be submitted through the respective Github issues pages (see source code links)

- Mailing Lists:
  - http://www.systemarchitect.tech/index.php/lists/

- Direct developer contacts
  - John Leidel: jleidel<at>tactcomplabs<dot>com
  - Frank Conlon: fconlon<at>tactcomplabs<dot>com