Design Concepts with System Architect: Level 1

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

Overview

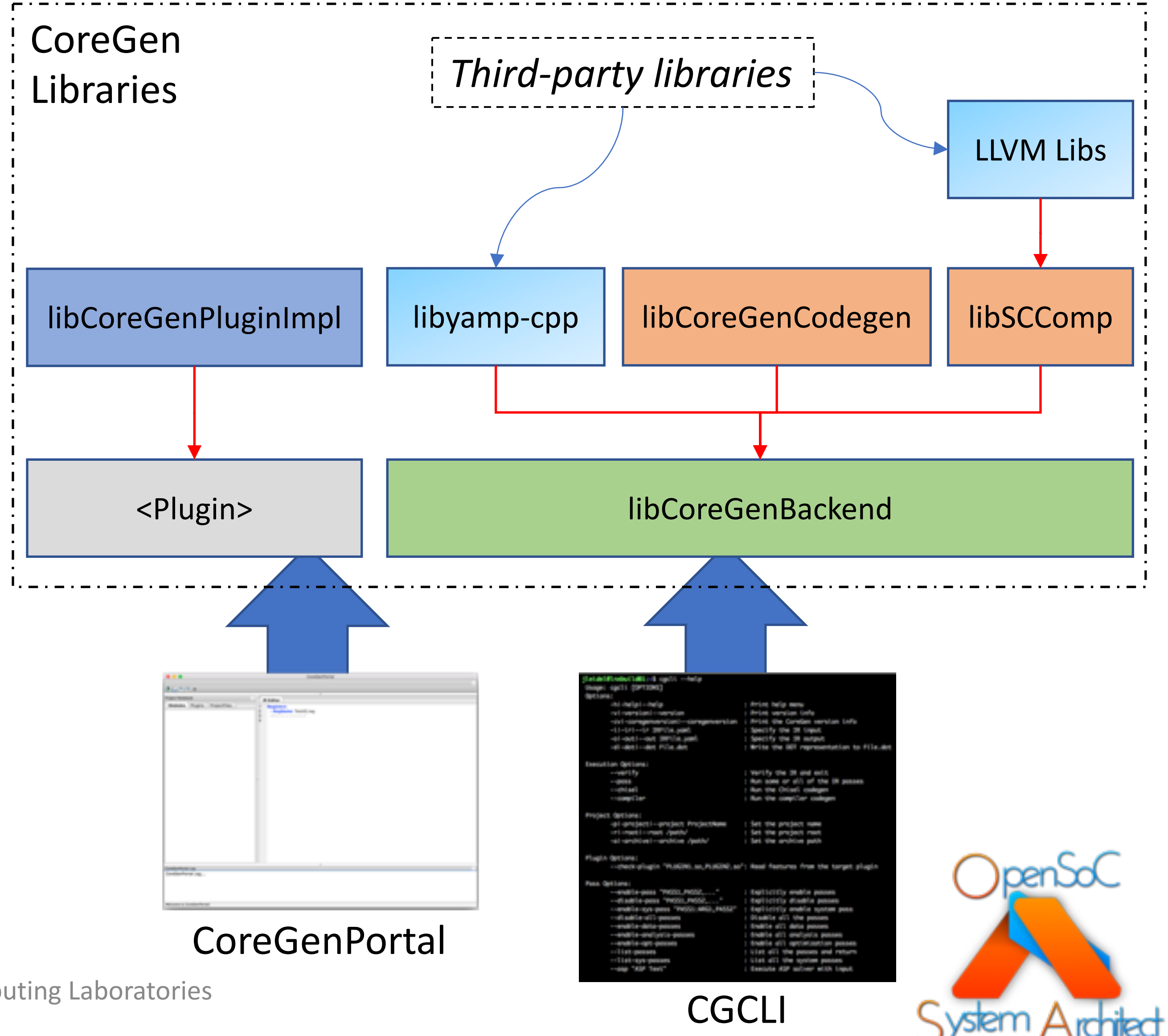
- System Architect Tool Infrastructure
- Designing a Basic RISC Device
- References

System Architect Tool Infrastructure

Tools/API Interfaces for System Architect

System Architect Tool/API Infrastructure

- Infrastructure consists of multiple libraries/APIs
  - libCoreGenBackend
  - libCoreGenCodegen
  - libCoreGenPluginImpl
  - libSCComp
  - libyamp-cpp
- Entire API interface is documented via Doxygen:
  - https://codedocs.xyz/opensocsysarch/CoreGen/
- User-facing tools include a command line interface and GUI
  - Command Line: CGCLI
  - GUI: CoreGenPortal



System Architect/CoreGen Libraries

libCoreGenBackend

- C++ library that serves as the primary library interface to CoreGen
- Handles are CoreGen IR reading/writing
- Handles the pass infrastructure
- Utilizes libyaml-cpp for reading/writing IR in *Yaml* form

libCoreGenCodegen

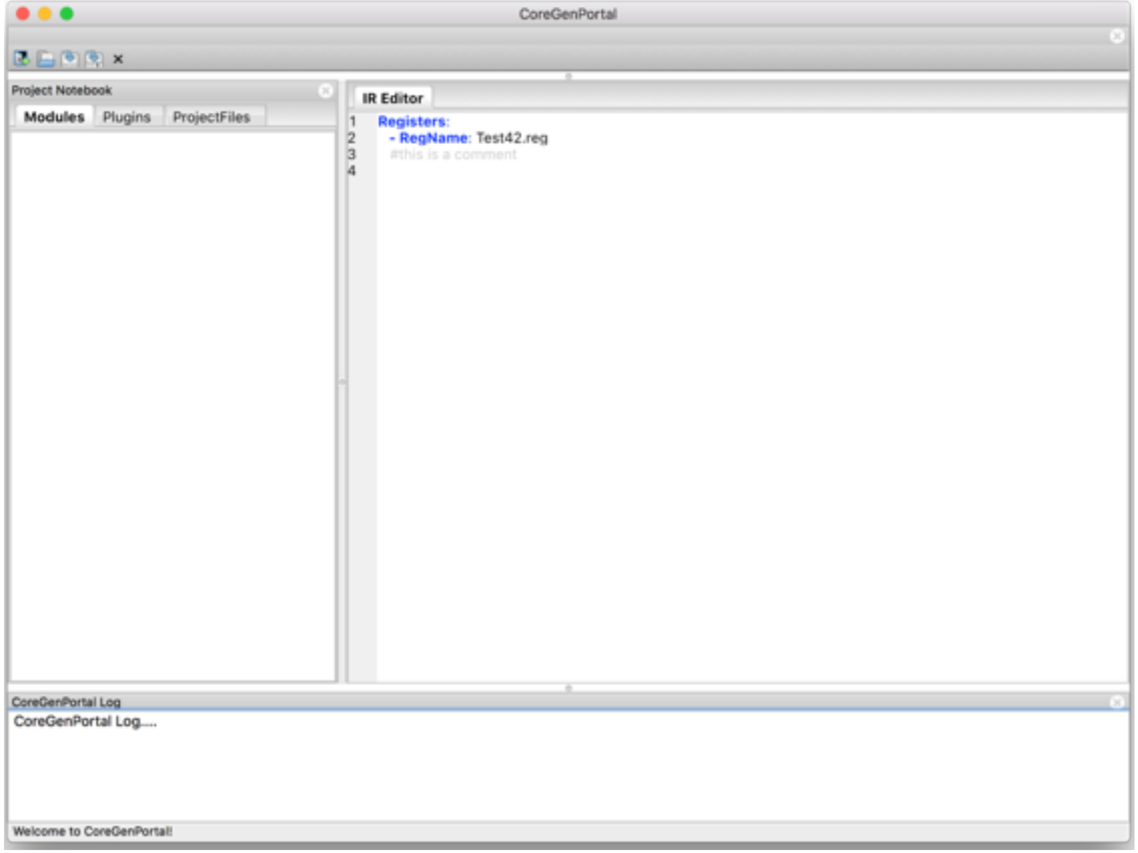
- Handles code generation for IR to Chisel translation
- Handles code generation for IR to LLVM compiler backend

libSCComp

- Implements the StoneCutter language spec
- Parser/Lexer/IR all utilize LLVM library interfaces
- Code optimization performed via LLVM optimizer
- Requires LLVM libraries
- **we also have a command interface for the StoneCutter compiler

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



Command Line Interface: *cgcli*

- Simple, concise command line interface to drive
  - Drives all verification passes
  - Drives all code generation facilities
- Use cases:
  - For those who prefer to utilize the command line and write/modify IR using text editors
  - To run quick tests against IR
  - Regression/CI environments to maintain designs

```
jleidel@lnxbuild01:~$ cgcli --help
 Usage: cgcli [OPTIONS]
 Options:
        -h|-help|--help                        : Print help menu
        -v|-version|--version                  : Print version info
        -cv|-coregenversion|--coregenversion   : Print the CoreGen version info
        -i|-ir|--ir IRFile.yaml                : Specify the IR input
        -o|-out|--out IRFile.yaml              : Specify the IR output
        -d|-dot|--dot File.dot                 : Write the DOT representation to File.dot

 Execution Options:
        --verify                               : Verify the IR and exit
        --pass                                 : Run some or all of the IR passes
        --chisel                               : Run the Chisel codegen
        --compiler                             : Run the compiler codegen

 Project Options:
        -p|-project|--project ProjectName      : Set the project name
        -r|-root|--root /path/                 : Set the project root
        -a|-archive|--archive /path/           : Set the archive path

 Plugin Options:
        --check-plugin "PLUGIN1.so,PLUGIN2.so": Read features from the target plugin

 Pass Options:
        --enable-pass "PASS1,PASS2,..."        : Explicitly enable passes
        --disable-pass "PASS1,PASS2,..."       : Explicitly disable passes
        --enable-sys-pass "PASS1:ARG1,PASS2"   : Explicitly enable system pass
        --disable-all-passes                   : Disable all the passes
        --enable-data-passes                   : Enable all data passes
        --enable-analysis-passes               : Enable all analysis passes
        --enable-opt-passes                    : Enable all optimization passes
        --list-passes                          : List all the passes and return
        --list-sys-passes                      : List all the system passes
        --asp "ASP Text"                       : Execute ASP solver with input
```

CGCLI Info Options

- --help : Prints the help menu
- --version : Prints the version info
- --coregenversion : Prints the CoreGen library version

```
$> cgcli --help
$> cgcli --version
$> cgcli --coregenversion
```

CGCLI Execution Options

- Execution options require IR input
  - --ir /path/to/IR.yaml
- Four execution options:
  - --verify : verifies that the CoreGen IR (Yaml) is syntactically correct. Returns a nonzero exit code from CGCLI upon failure
  - --pass : executes one or more verification, data or optimization passes
  - --chisel: executes the Chisel code generator to output Chisel HDL
  - --compiler: executes the LLVM code generator to output LLVM compiler for target design
- Options can be combined!

```
$> cgcli --ir TEST.yaml --verify
$> cgcli --ir TEST.yaml --pass
$> cgcli --ir TEST.yaml --chisel
$> cgcli --ir TEST.yaml --compiler
$> cgcli --ir TEST.yaml --pass --chisel --compiler
```

CGCLI Execution Options cont.

- Additional options can be utilized
- --out /path/to/output.yaml : specifies output yaml file (in the event that you want to preserve existing yaml file)
- --dot /path/to/File.dot : generates a Dot (Graphviz) output that represents the dependence graph of the IR in graphical form
- --project NAME : overrides the project name for the target IR
- --root /path/to/project/root/ : overrides the default project root (current working dir)
- --archive /path/to/archive/ : overrides the default archive path that contains plugins, LLVM compiler templates, etc

```
$> cgcli --ir TEST.yaml --dot TEST.dot
$> dot -Tpng TEST.dot > TEST.png
```

CGCLI Pass Options

- Users can list all the supported passes on the command line
- --list-passes: prints a table of all passes, their types and descriptions of what they do
- --list-sys-passes: prints a similar table of all the special-purpose system passes

```
jleidel@lnxbuild01:~$ cgcli --list-passes
PASSNAME                      PASSTYPE         DESCRIPTION
-------------------------------------------------------------------------------
StatsPass                     Data      Prints statistics for all DAG nodes
MultSoCPass                   Analysis  Examines IR and warns if multiple SoC nodes are present
ICacheCheckerPass             Analysis  Identifies any Cores that potentially lack an ICache
L1SharedPass                  Analysis  Identifies any potential L1 Caches that are potentially shared across Cores
RegIdxPass                    Analysis  Identifies potential register index collisions within a Register Class
RegFieldPass                  Analysis  Identifies instruction encodings that require registers without a Register Class
RegSafetyPass                 Analysis  Checks various safety parameters of the register files
CoreSafetyPass                Analysis  Checks various safety parameters for each defined core
CommSafetyPass                Analysis  Identifies issues in using Comm module connectivity
RegClassSafetyPass            Analysis  Examines IR and warns on issues with RegClasses
CacheLevelPass                Analysis  Tests cache hierarchy to ensure child levels are correctly connected
DanglingNodePass              Analysis  Identifies any potential dangling or unconnected nodes
DanglingRegionPass            Analysis  Identifies any potential dangling regions or modules
EncodingCollisionPass         Analysis  Identifies any potential ISA encoding collisions
MandatoryFieldPass            Analysis  Identifies discrepancies in the number of mandatory fields defined and the number of defined fields
EncodingGapPass               Analysis  Identifies any potential unused gaps in ISA encoding formats
PInstSafetyPass               Analysis  Identifies issues in using Pseudo Instructions
CommSafetyPass                Analysis  Identifies issues in using Comm module connectivity
jleidel@lnxbuild01:~$ cgcli --list-sys-passes
PASSNAME                      PASSTYPE         DESCRIPTION
-------------------------------------------------------------------------------
SafeDeletePass                Analysis  Determines whether a node can be safely deleted
ASPSolverPass                 Analysis  Executes ASP solver input against L3 IR
InstTable                     Data      Prints instruction and register tables
SpecDoc                       Data      Generates a LaTeX Spec Document
```

```
$> cgcli --list-passes
$> cgcli --list-sys-passes
```

CGCLI Pass Options cont.

- Executing passes is done using the '--pass' option
  - By default, all passes noted in '--list-passes' are executed with the --pass option
- Each pass runs sequentially and prints any potential faulty state as well as the time required to run each pass
  - The larger the graph, the more difficult some passes will be to run
- A summary is printed with PASS/FAIL state
  - All passes that FAIL will have some notional text describing where the dependence failure occurred in the graph
- **NOTE**: Failed passes don't necessarily imply the design is broken. Further analysis is required by the user

```
...Executing EncodingCollisionPass


...Executing MandatoryFieldPass


...Executing EncodingGapPass


...Executing PInstSafetyPass


...Executing CommSafetyPass
================================================================
 CoreGen Pass Summary
================================================================
PASS                          TIME (secs)                   PASS/FAIL
StatsPass....................0.002074......................PASSED
MultSoCPass..................0.000613928...................PASSED
ICacheCheckerPass............0.00103784....................PASSED
L1SharedPass.................0.0012939.....................PASSED
RegIdxPass...................0.000612974...................PASSED
RegFieldPass.................0.000607967...................PASSED
RegSafetyPass................0.00087595....................PASSED
CoreSafetyPass...............0.000684023...................PASSED
CommSafetyPass...............0.000593901...................PASSED
RegClassSafetyPass...........0.000660181...................PASSED
CacheLevelPass...............0.00075984....................PASSED
DanglingNodePass.............0.000319958...................PASSED
DanglingRegionPass...........0.000352144...................PASSED
EncodingCollisionPass........0.0014081.....................PASSED
MandatoryFieldPass...........0.00090909....................PASSED
EncodingGapPass..............0.000613928...................PASSED
PInstSafetyPass..............0.000607014...................PASSED
CommSafetyPass...............0.000605106...................PASSED
```

```
$> cgcli --ir TEST.yaml --pass
```

CGCLI Pass Failures

- Any failures will be noted in the pass summary
- Look above to the individual pass outputs for the errors reported
- Most passes will indicate the exact node name that tripped the failure(s)

```
==========================================================
 CoreGen Pass Summary
==========================================================
PASS                          TIME (secs)                  PASS/FAIL
StatsPass......................0.00336099....................PASSED
MultSoCPass....................0.00103903....................PASSED
ICacheCheckerPass..............0.0014751.....................PASSED
L1SharedPass...................0.00171709....................PASSED
RegIdxPass.....................0.00102878....................PASSED
RegFieldPass...................0.00101495....................PASSED
RegSafetyPass..................0.001297......................PASSED
CoreSafetyPass.................0.00115085....................PASSED
CommSafetyPass.................0.00102019....................PASSED
RegClassSafetyPass.............0.00107694....................PASSED
CacheLevelPass.................0.001019......................PASSED
DanglingNodePass...............0.00176311....................FAILED
DanglingRegionPass.............0.000591993...................FAILED
EncodingCollisionPass..........0.0017972.....................PASSED
MandatoryFieldPass.............0.00134087....................PASSED
EncodingGapPass................0.00102401....................PASSED
PInstSafetyPass................0.00101495....................PASSED
CommSafetyPass.................0.00101805....................PASSED
```

Failed Analysis Passes

```
...Executing DanglingNodePass
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {198:TEST47.ext0}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {199:TEST47.ext1}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {200:TEST47.ext2}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {201:TEST47.ext3}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {202:TEST47.ext4}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {203:TEST47.ext5}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {204:TEST47.ext6}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {205:TEST47.ext7}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {206:TEST47.ext8}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {207:TEST47.ext9}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {208:TEST47.ext10}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {209:TEST47.ext11}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {210:TEST47.ext12}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {211:TEST47.ext13}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {212:TEST47.ext14}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {213:TEST47.ext15}
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {214:TEST47.ext16}
```

This indicates that these nodes are not connected and will be optimized out by downstream synthesis tools

```
$> cgcli --ir TEST.yaml --pass
```

CGCLI Pass Options cont.

- You can execute the different types of passes with special options
- --enable-analysis-passes: Enables only the analysis passes
- --enable-data-passes: Enables only the data passes
- --enable-opt-passes: Enables only the optimization passes

```
$> cgcli --ir TEST.yaml --pass --enable-analysis-passes
$> cgcli --ir TEST.yaml --pass --enable-data-passes
$> cgcli --ir TEST.yaml --pass --enable-opt-passes
```

CGCLI Pass Options cont.

- You can also enable/disable individual passes using their names from the --list-passes option
- --enable-pass “PASS1, PASS2”
- --disable-pass “PASS1, PASS2”

```
#-- Only run the DanglingNodePass
$> cgcli --ir TEST.yaml --pass --enable-pass “DanglingNodePass”
#-- Run all passes, but disable the EncodingGapPass
$> cgcli --ir TEST.yaml --pass --disable-pass “EncodingGapPass”
```

CGCLI Pass Options cont.

- You can execute individual system passes with:
  - --enable-sys-pass “PASS1:ARG1,PASS2”
- NOTE
  - System passes often require arguments in order to execute correctly
  - The ”--pass” argument is not required for system passes

```
#-- Generate the specification doc for the target design
$> cgcli --ir TEST.yaml --enable-sys-pass “SpecDoc:/path/to/output/dir”
```

CGCLI Plugin Query

- Plugins are external CoreGen modules that are packaged as a shared library object
- They utilize a plugin API to drive custom hardware, software and code generation features
- These plugins can be licensed and distributed separate from CoreGen
- CGCLI provides a command line option to query the special features found in a plugin payload
  - --check-plugin “/path/to/plugin”

```
PLUGIN NAME                         VERSION                        NUM FEATURES
------------------------------------------------------------------------------------
SamplePlugin                         1.0.0                         3

 -----> Features <-----
  Idx      Feature      Type
  ------------------------
  [0]      CORES      Unsigned
  [1]      FEATURE2      Float
  [2]      FEATURE3      String

 -----> Codegen Options <-----
  HDL Codegen: no
  LLVM Codegen: no
------------------------------------------------------------------------------------
```

```
$> cgcli --check-plugin ”./libSamplePlugin.so"
```

Designing a Basic RISC Device

A first conceptual design in System Architect

Tutorial Source

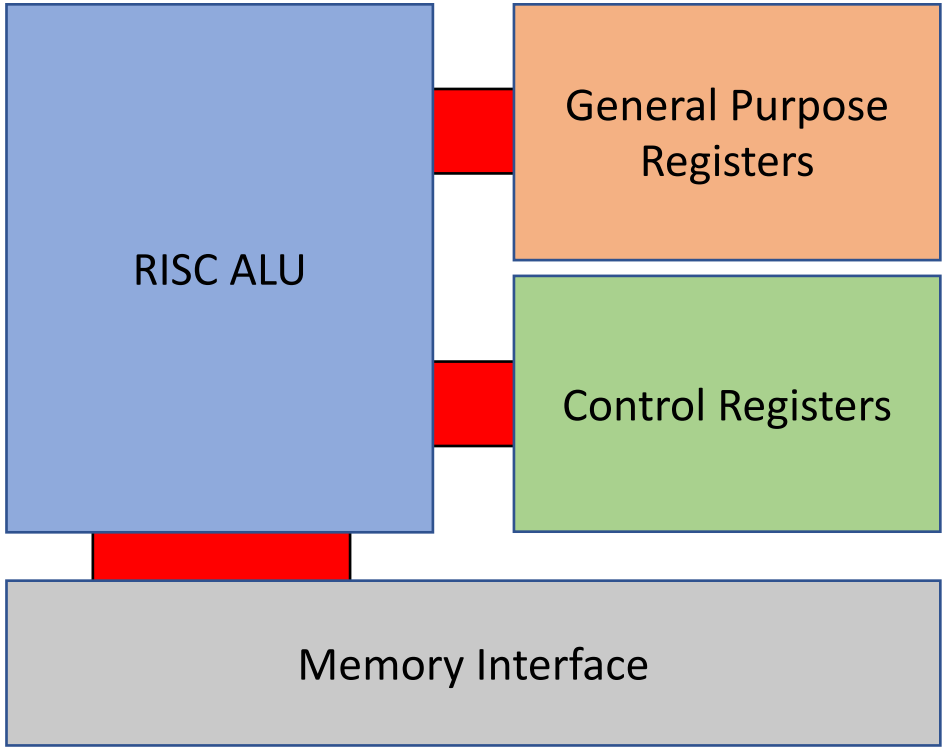
- Tutorial source is published in a Github repository
- All design source code is open source under an Apache2 license
  - Feel free to reuse it!!
- https://github.com/opensocsysarch/CoreGenTutorials
  - See the LEVEL1 subdirectory

Tutorial Assumptions

- Standard installation location:
  - "/opt/coregen"
  - EG, the "cgcli" binary will be located at /opt/coregen/bin/cgcli
  - 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 1 tutorial will focus on constructing a simple RISC core
  - Not designed to include every potential bit of functionality
  - Simple for a reason!
  - Design is open source (Apache license)
- We will utilize a text editor to construct the IR by hand
  - We will describe the each of the node attributes by hand as we go along
- Our basic core will support:
  - General purpose registers
  - Control registers
  - Simple instruction format
  - Memory I/O, basic arithmetic, control flow instructions



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
```

Ten Design Steps for Level 1

- ***Step 1***: Build our project skeleton
- ***Step 2***: Begin editing Yaml IR file by defining project infrastructure
- ***Step 3***: Define the register infrastructure
- ***Step 4***: Define the ISA
- ***Step 5***: Define the instruction format
- ***Step 6***: Define the instructions
- ***Step 7***: Define the pseudo instructions
- ***Step 8***: Define a cache
- ***Step 9***: Define a core
- ***Step 10***: Define an SoC

The CoreGen IR for each step is outlined in **~/CoreGenTutorials/LEVEL1/Step*N***

Step 1: Build the Project Skeleton

- Create a new directory to hold all your project-related files
  - All the generated artifacts will reside within this top-level directory
- Create a basic Yaml IR file using your preferred text editor
  - BasicRISC.yaml
- Add some header information to describe the file
  - Remember: comments are denoted using ‘#’
  - Multi-line comments need a new ‘#’ for every new line

```
#-----------------------------------------------------
#--- BasicRISC.yaml
#
# My first CoreGen Design Experiment
#-----------------------------------------------------
```

~/CoreGenTutorials/LEVEL1/Step1

Step 2: Define the project information

- Now we need to create a new project
  - Describes the name and type of the project
  - Helps optimize code generation for different project types
  - Future support for different Chisel or LLVM backends
- Create a new "ProjectInfo" node block as shown here

```
#-------------------------------------------------
#--- BasicRISC.yaml
#
# My first CoreGen Design Experiment
#-------------------------------------------------

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

~/CoreGenTutorials/LEVEL1/Step2

Step 2: Define the project information cont.

- Parameters:
  - *ProjectName*: Name of the project. Utilized downstream by the generated LLVM compiler to define the architecture target
  - *ProjectRoot*: The source directory of the project. Usually "./". Can point at other directories
  - *ProjectType*: Defines the "style" of the project
  - *ChiselMajorVersion*: Defines the major version for the generated Chisel
  - *ChiselMinorVersion*: Defines the minor version for the generated Chisel

- Project Types
  - *soc*: System on chip designs with connectivity between heterogeneous modules/cores as well as internal and/or external memories
  - *module*: Hardware entities that are effectively self contained. Will contain a small number of disparate node types
  - *extension*: Hardware entities that include hierarchies of modules. EG, an accelerator with its respective ISA
  - *unknown*: Projects that don't fit any of the aforementioned models
- Chisel Versions
  - Currently only supports "3" & "0" for Chisel 3.0

~/CoreGenTutorials/LEVEL1/Step2

Step 3: Define the register infrastructure

- Now we begin adding registers to our design
- All the registers are added in a single *Registers* node block
  - We will separate them into register classes in Step 4
- Registers must have unique names
  - Each register will have a multitude of parameters

- Lets start by adding an ***r0*** register as shown here
- A few interesting things to note:
  - As with other RISC ISAs, *r0* is hardwired to "0"
  - Read-Only register

```
Registers:
  - RegName: r0
    Width: 64
    Index: 0
    PseudoName: zero
    IsFixedValue: true
    FixedValue: 0
    IsSIMD: false
    RWReg: false
    ROReg: true
    CSRReg: false
    AMSReg: false
    TUSReg: false
    Shared: false
    PCReg: false
```

~/CoreGenTutorials/LEVEL1/Step3

Step 3: Define the register infrastructure cont.

- Parameters
  - *RegName*: unique name of the register used in assembly code
  - *Width*: width of the register in bits
    - All of our registers here will be 64 bits
  - *Index*: the index within the respective register class, must be unique within the register class
  - *PseudoName*:
    - [Optional] Pseudo name that be also be utilized in assembly

- Parameters
  - *IsFixedValue*: Is this register hardwired to a fixed value? If 'true', then we must also have a "FixedValue" parameter
  - *FixedValue*: the decimal value of the hardwired register

```
Registers:
  - RegName: r0
    Width: 64
    Index: 0
    PseudoName: zero
    IsFixedValue: true
    FixedValue: 0
    IsSIMD: false
    RWReg: false
    ROReg: true
    CSRReg: false
    AMSReg: false
    TUSReg: false
    Shared: false
    PCReg: false
```

~/CoreGenTutorials/LEVEL1/Step3

Step 3: Define the register infrastructure cont.

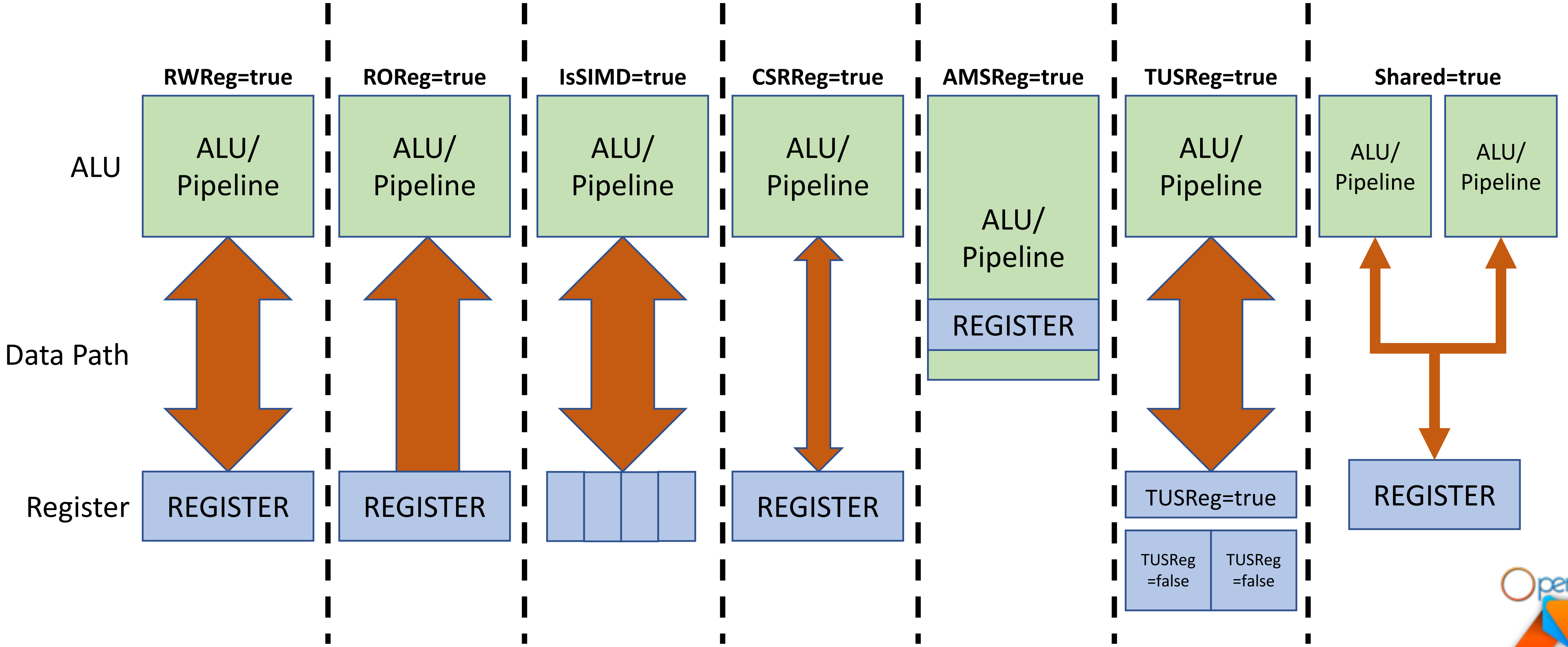
- Parameters
  - *IsSIMD:* Is the register a SIMD register?
  - *RWReg*: Can we read and write this register?
  - *ROReg*: Is this register Read-Only (from assembly)?
  - *CSRReg*: Is this a configuration/status register? Generally considered to be on slow data path

- Parameters
  - *AMSReg*: Is this an arithmetic machine state register? This forces the registers to reside within the ALU pipeline
  - *TUSReg*: Is this register shared across thread units? Thread units are duplicated register files for symmetric multi-threading within a core. If set to false, then each thread unit will get a unique copy of this register
  - *Shared*: Is this register shared across multiple cores? Configuration registers are generally shared
  - *PCReg*: Is this register a program counter? PC registers are automatically incremented if they are not modified in the instruction body

```
Registers:
  - RegName: r0
    Width: 64
    Index: 0
    PseudoName: zero
    IsFixedValue: true
    FixedValue: 0
    IsSIMD: false
    RWReg: false
    ROReg: true
    CSRReg: false
    AMSReg: false
    TUSReg: false
    Shared: false
    PCReg: false
```

~/CoreGenTutorials/LEVEL1/Step3

Step 3: Register Parameters



Step 3: Define the register infrastructure cont

- Now add the remainder of the general purpose registers
  - Add these within the existed *Registers* block
- The naming convention used here is:
  - *r1 - r31*

- Remember:
  - Each needs a unique Index value
  - Make the remainder of the registers ReadWrite
  - Don’t require PseudoNames or FixedValues
    - Feel free to add PseudoName values at your leisure

```
- RegName: r1
  Width: 64
  Index: 1
  IsFixedValue: false
  FixedValue: 0
  IsSIMD: false
  RWReg: true
  ROReg: false
  CSRReg: false
  AMSReg: false
  TUSReg: false
  Shared: false
  PCReg: false
```

~/CoreGenTutorials/LEVEL1/Step3

Step 3: Define the register infrastructure cont.

- Now we need to add our control registers
  - Notice that we restart the indexing at 0
  - We will compose two separate register files
- Read-Only Control Regs:
  - PC: program counter
  - EXC: exception flags
    - More on this in future tutorials
  - NE: *Not equal*
  - EQ: *Equal*
  - GT: *Greater Than*
  - LT: *Less Than*
  - GTE: *Greater than or equal to*
  - LTE: *Less than or equal to*

- Read-Write Control Regs:
  - SP: stack pointer
  - FP: frame pointer
  - RP: return pointer

- Comparison registers (NE, EQ, etc) are utilized for flow control and branches

```
#-- compare two registers; place result in r7
LABEL:
        #-- compare two registers;
        #-- place result in r7
        cmp.eq r7,r5,r6
        #-- if the result was eq, take the branch
        brac LABEL, r7, eq
```

~/CoreGenTutorials/LEVEL1/Step3

Step 3: Define the register infrastructure cont.

- Comparison registers
  - Each comparison register has a fixed value associated with it
  - Fixed values map to the corresponding function code for respective compare instruction
    - More on this in Step 5 & 6

- Fixed Values
  - NE = 2
  - EQ = 3
  - GT = 4
  - LT = 5
  - GTE = 6
  - LTE = 7

~/CoreGenTutorials/LEVEL1/Step3

Step 3: Define the register infrastructure cont.

- Now that we have our registers defined, we need to create two register classes:
  - GPR: {r0-r31}
  - CTRL: {pc, exc, ne, etc}
- Create a new node block called *RegClasses*

- Parameters
  - *RegisterClassName*: Unique name associated with each register class. Utilized to specify permissible registers in the instruction format register fields.
  - *Registers*: Defines the set of registers in our register class. Using the names from the *Registers* block, add each of the registers to the register class

```
RegClasses:
  - RegisterClassName: GPR
    Registers:
      - r0
      - r1
      …..
      - r31
    - RegisterClassName: CTRL
      Registers:
        - pc
        - exc
        - ne
        - eq
        ….
        - rp
```

~/CoreGenTutorials/LEVEL1/Step3

Step 3: Define the register infrastructure cont.

- Now that we have all our registers defined, lets utilize System Architect to verify the design so far
- Utilize the “cgcli” tool to run all of our passes
- FAILED!?

```
$> cgcli --ir ./BasicRISC.yaml --verify
$> cgcli --ir ./BasicRISC.yaml --pass
```

```
==================================================================
 CoreGen Pass Summary
==================================================================
PASS                          TIME (secs)                 PASS/FAIL
StatsPass....................0.000405788..................PASSED
MultSoCPass..................3.19481e-05..................PASSED
ICacheCheckerPass............3.19481e-05..................PASSED
L1SharedPass.................3.09944e-05..................PASSED
RegIdxPass...................4.29153e-05..................PASSED
RegFieldPass.................3.09944e-05..................PASSED
RegSafetyPass................3.79086e-05..................PASSED
CoreSafetyPass...............3.09944e-05..................PASSED
CommSafetyPass...............3.19481e-05..................PASSED
RegClassSafetyPass...........0.00018692...................PASSED
CacheLevelPass...............3.21865e-05..................PASSED
DanglingNodePass.............2.7895e-05...................PASSED
DanglingRegionPass...........1.90735e-05..................FAILED
EncodingCollisionPass........3.09944e-05..................PASSED
MandatoryFieldPass...........3.09944e-05..................PASSED
EncodingGapPass..............3.19481e-05..................PASSED
PInstSafetyPass..............3.19481e-05..................PASSED
CommSafetyPass...............3.09944e-05..................PASSED
```

Failed Analysis Passes

~/CoreGenTutorials/LEVEL1/Step3

Step 3: Define the register infrastructure cont.

- Lets analyze why this failed:
- *DanglingRegionPass*
  - Finds the number of unique, unconnected regions of nodes
  - Graph theory: community detection
  - We had two unconnected regions
  - Why??
- Lets utilize the graphing function to examine the dependence graph

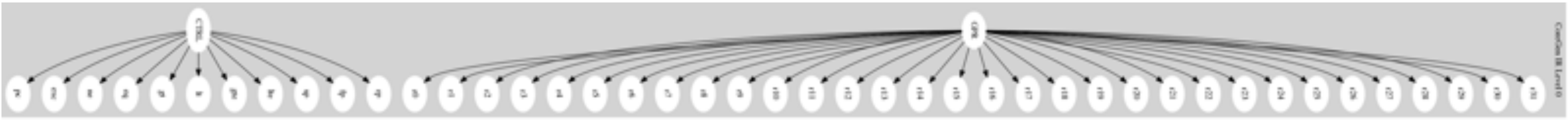
```
...Executing DanglingRegionPass
DanglingRegionPass : Identified dangling component regions: 2 Regions
```

```
$> cgcli --ir ./BasicRISC.yaml --dot Step3.dot
$> dot -Tpng Step3.dot > Step3.png
```

CTRL Register File

We have two, unique register files that aren't yet connected to anything useful. Hence, two unconnected regions of hardware!

GPR Register File



~/CoreGenTutorials/LEVEL1/Step3

Step 4: Define an instruction set

- Now that we have registers and register classes, we need an instruction set container
- ISA nodes have a single parameter that identifies a unique ISA name
- You can include an unlimited number of unique ISAs in a design
- ISA nodes are ”containers”
  - We will utilize them in future steps to organize instructions and register classes into an instruction set implementation
- Define an *ISAs* node block

- Parameters
  - *ISAName*: Unique name for an instruction set

```
ISAs:
  - ISAName: BasicRISC.ISA
```

~/CoreGenTutorials/LEVEL1/Step4

Step 4: Define an instruction set cont.

```
$> cgcli --ir ./BasicRISC.yaml --verify
$> cgcli --ir ./BasicRISC.yaml --pass
```

- Now that we have all our instruction set defined, lets rerun our analysis passes
- Utilize the “cgcli” tool to run all of our passes
- FAILED AGAIN!?

```
=================================================================
 CoreGen Pass Summary
=================================================================
PASS                          TIME (secs)                 PASS/FAIL
StatsPass.....................0.000409842.................PASSED
MultSoCPass...................3.29018e-05.................PASSED
ICacheCheckerPass.............3.31402e-05.................PASSED
L1SharedPass..................3.19481e-05.................PASSED
RegIdxPass....................4.60148e-05.................PASSED
RegFieldPass..................3.31402e-05.................PASSED
RegSafetyPass.................3.98159e-05.................PASSED
CoreSafetyPass................0.000193834.................PASSED
CommSafetyPass................0.000162125.................PASSED
RegClassSafetyPass............0.00019598..................PASSED
CacheLevelPass................3.29018e-05.................PASSED
DanglingNodePass..............3.69549e-05.................FAILED
DanglingRegionPass............1.81198e-05.................FAILED
EncodingCollisionPass.........3.40939e-05.................PASSED
MandatoryFieldPass............3.19481e-05.................PASSED
EncodingGapPass...............3.19481e-05.................PASSED
PInstSafetyPass...............3.38554e-05.................PASSED
CommSafetyPass................3.21865e-05.................PASSED
```

Failed Analysis Passes

~/CoreGenTutorials/LEVEL1/Step4

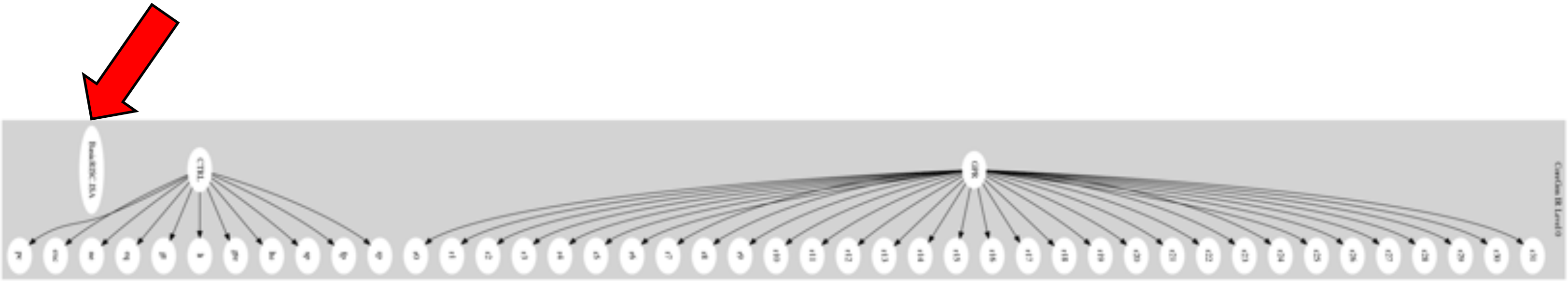
Step 4: Define an instruction set cont.

- The *DanglingRegionPass* fails for the same reason as in Step 3
  - Multiple unconnected regions
- The *DanglingNodePass* fails because our new ISA isn’t utilized in any cores yet
- Lets utilize CGCLI to rerun our passes and ignore these two analysis passes
  - Success!

```
...Executing DanglingNodePass
DanglingNodePass : Identified a dangling node at DAG node Index:Name = {45:BasicRISC.ISA}
```

```
$> cgcli --ir ./BasicRISC.yaml --pass --disable-pass "DanglingRegionPass,DanglingNodePass"
```

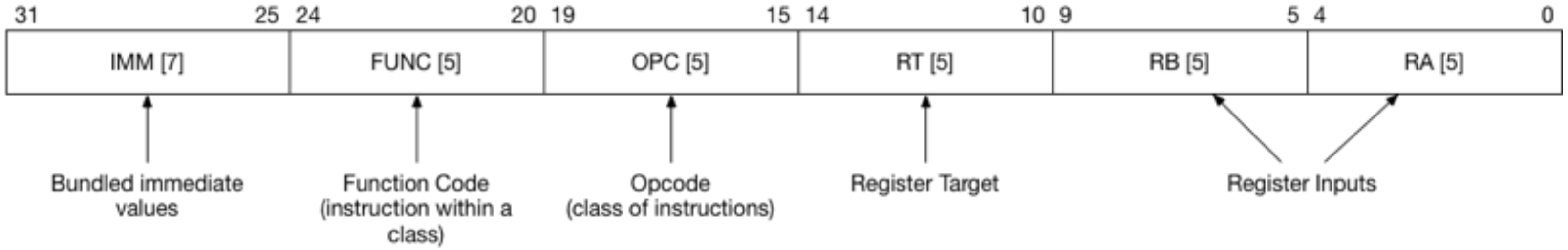
Dangling
ISA Node



~/CoreGenTutorials/LEVEL1/Step4

Step 5: Define the instruction format(s)

- Now that we have our ISA ready to receive instructions and our register classes ready to be utilized, we can begin defining instruction formats
- Instruction formats can contain an unlimited number of fields in any bit width
  - RISC is not required
- Three types of fields are supported
  - *Encoding Fields*: Hardwired encodings for individual instructions
  - *Register Fields*: Inserted by the assembler and utilized as indices into hardware state
  - *Immediate Value Fields*: Inserted by the assembler and utilized as literal values by the instruction



BasicRISC Includes 6 Fields across 32 bits of encoding

- RA: register input
- RB: register input
- RT: register output (target)
- OPC: opcode encoding
- FUNC: function encoding
- IMM: immediate value

~/CoreGenTutorials/LEVEL1/Step5

Step 5: Define the instruction format(s) cont.

- For each ***Register Field***, we assign a single register class
  - Enables the correct data path for the respective register index into the correct register file
- BasicRISC will include three instruction formats to support our three variants of register arguments

INST Rt, Ra, Rb	
Instruction Format Mnemonic	*Register Class Arguments*
Arith.if	INST GPR, GPR, GPR
ReadCtrl.if	INST GPR, GPR, CTRL
WriteCtrl.if	INST CTRL, GPR, GPR

~/CoreGenTutorials/LEVEL1/Step5

Step 5: Define the instruction format(s) cont.

- Create a new top-level node *InstFormats*
- Each format must have a unique name, instruction set container and width
- Users can define any number of subfields
    - Subfields must have at least 1 bit
    - Maximum is the bit width of the encoding
- ***Note***: You are NOT required to specify all the bits in the encoding
    - The *EncodingGapPass* will tell you if you have unused space

- Parameters
    - *InstFormatName:* Contains a unique instruction format name (utilized downstream by instruction definitions)
    - *ISA:* Specifies the instruction set container (Step 4)
    - *FormatWidth*: Defines the total width (in bits) for the encoding format
    - *Fields*: Contains the definitions of all the subfields

```
InstFormats:
  - InstFormatName: Arith.if
    ISA: BasicRISC.ISA
    FormatWidth: 32
    Fields:
```

~/CoreGenTutorials/LEVEL1/Step5

Step 5: Define the instruction format(s) cont.

- Field Parameters
  - *FieldName:* unique name (within the respective format) to describe a field
  - *FieldType:* The *type* of encoding:
    - *CGInstReg:* Register class fields (must include a *RegClass* attribute
    - *CGInstCode:* Instruction encoding field
    - *CGInstImm:* Immediate values
  - *FieldWidth:* The total width (in bits) for the encoding

- Field Parameters
  - *StartBit:* The starting bit of the respective field
  - *EndBit:* The ending bit of the respective field
  - *MandatoryField:* Determines whether any downstream instruction encodings are REQUIRED to specify a unique value for this field
    - EG, opcodes and function codes MUST be specified!

- Field Parameters
  - *RegClass:* Defines the register class for the associated CGInstReg field. The names of the register classes much match those defined in *RegClasses* section from Step 3

~/CoreGenTutorials/LEVEL1/Step5

Step 5: Define the instruction format(s) (Arith.if)

```
InstFormats:
  - 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
```

~/CoreGenTutorials/LEVEL1/Step5

Step 5: Define the instruction format(s) (ReadCtrl.if)

```
InstFormats:
  - InstFormatName: Read.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: CTRL
      - 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
```

~/CoreGenTutorials/LEVEL1/Step5

Step 5: Define the instruction format(s) (WriteCtrl.if)

```
InstFormats:
  - InstFormatName: Read.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: CTRL
      - 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
```

~/CoreGenTutorials/LEVEL1/Step5

Step 5: Define the instruction format(s) cont.

- Notes on defining instruction fields:
  - The tools support an unlimited number of formats
  - Use consistent naming conventions for your fields
    - You will utilize these downstream when defining each instruction
  - Single bit control fields are permitted
    - The *EncodingGapPass* will flag any formats that do not utilize every bit in the total encoding space
    - Even if bits are unused, its wise to mark them as fields in the format
  - Encoding fields are not bound to nibble/byte boundaries
    - The tools support arbitrarily complex encodings in non-byte aligned fields
    - Instruction fetches will always be on byte boundaries
    - Encoding orthogonal instruction formats in the final ELF binaries is more difficult and will waster space
      - EG, 129 bit instructions

~/CoreGenTutorials/LEVEL1/Step5

Step 5: Define the instruction format(s) cont.

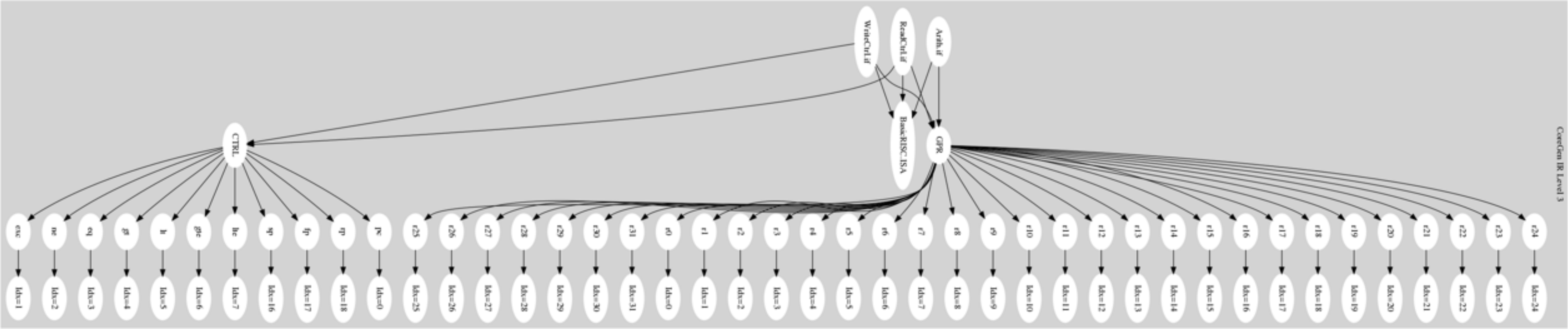
```
$> cgcli --ir ./BasicRISC.yaml --verify
$> cgcli --ir ./BasicRISC.yaml --pass
```

- Now that we have all our instruction set formats defined, lets rerun our analysis passes
- Utilize the “cgcli” tool to run all of our passes
- Everything passes!
  - Our register files are now dependencies within the instruction encodings!

```
=========================================================
 CoreGen Pass Summary
=========================================================
PASS                         TIME (secs)                PASS/FAIL
StatsPass....................0.000446081..................PASSED
MultSoCPass..................3.79086e-05..................PASSED
ICacheCheckerPass............4.00543e-05..................PASSED
L1SharedPass.................3.50475e-05..................PASSED
RegIdxPass...................5.19753e-05..................PASSED
RegFieldPass.................4.81606e-05..................PASSED
RegSafetyPass................4.31538e-05..................PASSED
CoreSafetyPass...............3.69549e-05..................PASSED
CommSafetyPass...............4.00543e-05..................PASSED
RegClassSafetyPass...........0.00022912...................PASSED
CacheLevelPass...............3.60012e-05..................PASSED
DanglingNodePass.............3.09944e-05..................PASSED
DanglingRegionPass...........1.71661e-05..................PASSED
EncodingCollisionPass........4.1008e-05...................PASSED
MandatoryFieldPass...........3.60012e-05..................PASSED
EncodingGapPass..............6.60419e-05..................PASSED
PInstSafetyPass..............3.69549e-05..................PASSED
CommSafetyPass...............3.69549e-05..................PASSED
```

~/CoreGenTutorials/LEVEL1/Step5

Step 5: Define the instruction format(s) cont.



~/CoreGenTutorials/LEVEL1/Step5

Step 6: Define the instructions

- Now that we have our instruction formats completed, we need to define our instructions
- Refer to the *BasicRISCInstTable* documents in the LEVEL1 tutorial directory for a full listing of the instructions and their associated encoding values
- Start by adding a top-level node block for *Insts*

- Parameters:
  - *ISA*: Defines the instruction set that this instruction resides within (Step 4)
  - *InstFormat*: Defines the instruction format utilized to encode this instruction (Step 5)
  - *Encodings*: Defines the set of encodings uniquely identifying this instruction

```
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
```

~/CoreGenTutorials/LEVEL1/Step6

Step 6: Define the instructions cont.

- Each of the encodings utilizes a field from our instruction format (Step 5)
- The names MUST match
  - They are case sensitive
- Each field must have a value and a width
- The field width must be <= the width of the field in the instruction format
  - We check the encoding value against the field width
  - This can be utilized for advanced designs in the future

- Encodings:
  - *EncodingField*: The instruction format field this encoding applies to
  - *EncodingWidth*: The width of the encoding (in bits)
  - *EncodingValue*: The decimal value you are encoding

```
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
```

~/CoreGenTutorials/LEVEL1/Step6

Step 6: Define the instructions cont.

- The tutorial slides do not introduce each instruction
- Its up to you to complete the remainder of the instructions using the documented encodings
  - Optionally utilize the design files provided for you
- There are 41 instructions defined in BasicRISC

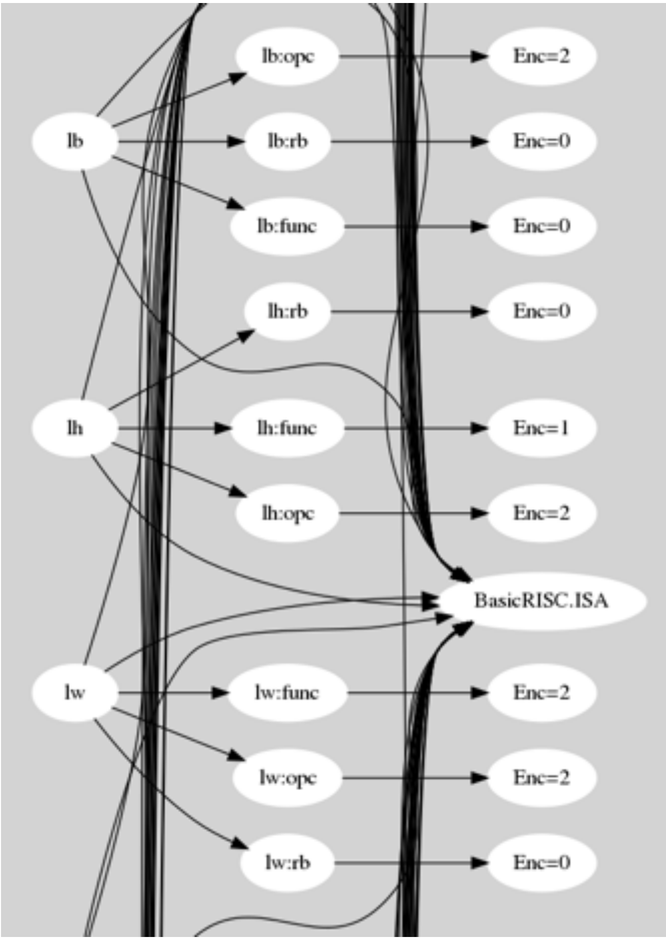
```
$> cgcli --ir ./BasicRISC.yaml --verify
$> cgcli --ir ./BasicRISC.yaml --pass --enable-pass "EncodingCollisionPass"
```

```
...Executing EncodingCollisionPass
EncodingCollisionPass.........0.00110006...................PASSED
```

- For instructions that don’t utilize the immediate field, we hardwire its value to zero
  - Isn’t necessary, but good practice
- Use the tools to check your encodings!
- The *EncodingCollisionPass* will find all potential collisions in the encoded space
  - Will report which instructions contain the collisions

~/CoreGenTutorials/LEVEL1/Step6

Step 6: Define the instructions cont.




```
$> cgcli --ir ./BasicRISC.yaml --dot Step6.dot
$> dot -Tpng Step3.dot > Step6.png
```

- This is a snapshot of our instruction set definition in the *dot* graph output with all the encodings expressed
- Notice how the encodings for each field are mapped in the dependence graph:
  - *INSTRUCTION_NAME:FIELD_NAME*
  - Enables easier visual debugging of dependence/encoding issues

~/CoreGenTutorials/LEVEL1/Step6

Step 7: Define the pseudo instructions

- Now that we have our instructions defined, we can optionally define some pseudo instructions
- Pseudo instructions are instruction aliases to make for clear, concise assembly language
- For example
  - mov Rt, Ra = add Rt, Ra, 0
- The pseudo instructions don't cost any additional hardware
  - They're only expressed in the compiler/assembler
- Our BasicRISC design has three pseudo instructions
  - Move GPR -> GPR
  - Move CTRL -> GPR
  - Move GPR -> CTRL
- Refer to the *BasicRISCInstTable* documents

- Add a top-level *PseudoInst* node block
- Parameters:
  - *PseudoInst*: The unique name of the pseudo instruction
  - *ISA*: The containing ISA
  - *Inst*: The base instruction that is aliased
  - *Encodings*: Contains a set of special encodings for this pseudo instruction

```
PseudoInsts:
  - PseudoInst: mov
    ISA: BasicRISC.ISA
    Inst: add
    Encodings:
      - EncodingField: ra
        EncodingWidth: 5
        EncodingValue: 0
```

~/CoreGenTutorials/LEVEL1/Step7

Step 7: Define the pseudo instructions cont.

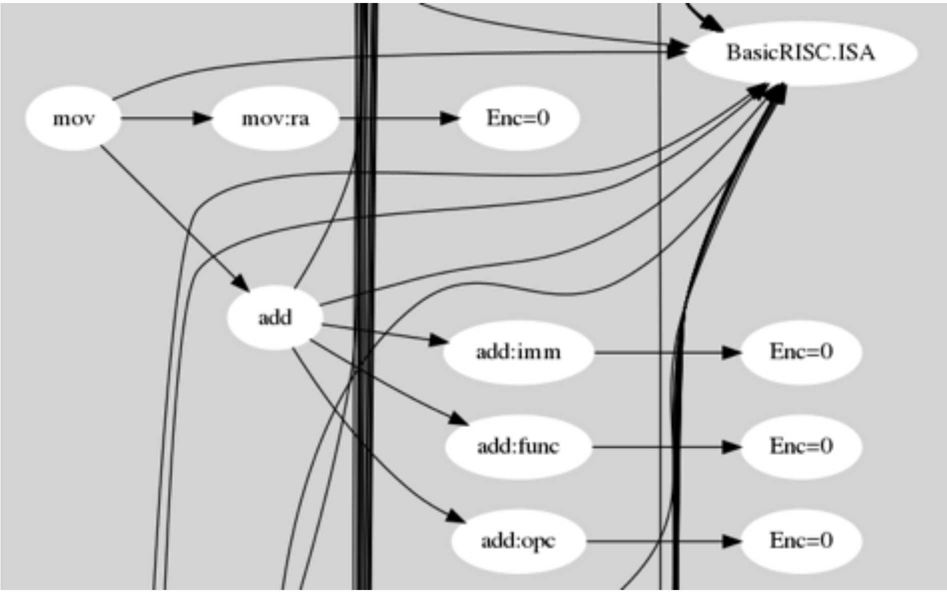
- Encodings:
  - *EncodingField*: The instruction format field this encoding applies to
  - *EncodingWidth*: The width of the encoding (in bits)
  - *EncodingValue*: The decimal value you are encoding
- In this example, we are forcing the Ra field to always be "0"
  - This forces Ra = register R0 (which is hardwired to zero
- Notes on pseudo instructions
  - You can define multiple pseudo instructions that alias a single instruction
  - You can override any of the register field and immediate encodings
  - You CANNOT override the encoding fields

```
PseudoInsts:
  - PseudoInst: mov
    ISA: BasicRISC.ISA
    Inst: add
    Encodings:
      - EncodingField: ra
        EncodingWidth: 5
        EncodingValue: 0
```

~/CoreGenTutorials/LEVEL1/Step7

Step 7: Define the pseudo instructions cont.

- Now our pseudo instructions are depicted in the dependence graph
- Note how they depend upon the original instruction definitions
- Removing the original instructions will break the dependence graph and the tools will warn the user of the issue




```
PseudoInsts:
  - PseudoInst: mov
    ISA: BasicRISC.ISA
    Inst: add
    Encodings:
      - EncodingField: ra
        EncodingWidth: 5
        EncodingValue: 0
```

~/CoreGenTutorials/LEVEL1/Step7

Step 8: Define a cache

- Now we begin adding memory units to our design: *Caches*
- Caches are optional components in your designs
  - Not required, but advantageous for performance
  - Absence of caches will force instruction fetches from memory
- Caches can also be hierarchical
  - We have a single cache layer in BasicRISC, but it is trivial to add hierarchical caches
- Create a *Caches* node block

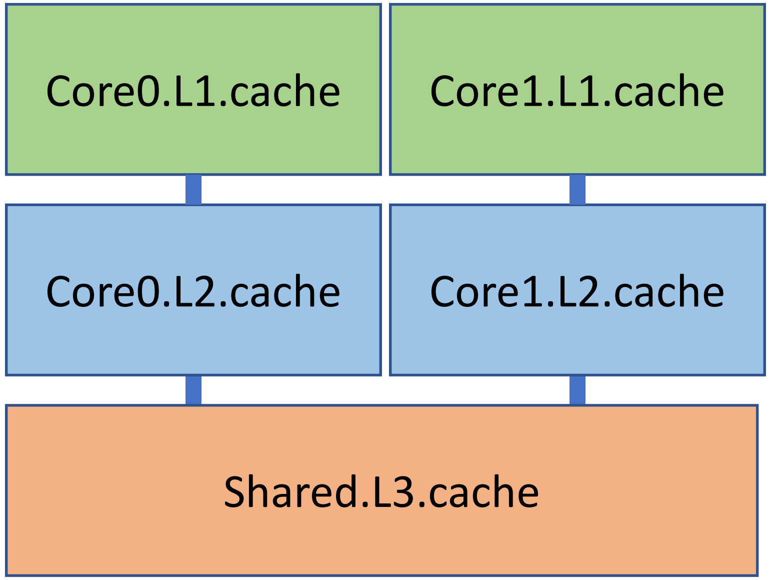
- Parameters:
  - *Cache*: The unique name of the cache level. These names must be unique across **ALL** cores within a SoC
  - *Sets*: The number of *sets* in the cache blocking configuration
  - *Ways*: The number of *ways* in the cache tagging configuration
  - *SubLevel*: Optional parameter that describes a connected sub-cache
    - The L2 would be a sublevel from L1

```
Caches:
  - Cache: Core0.L1.cache
    Sets: 2
    Ways: 8
```

~/CoreGenTutorials/LEVEL1/Step8

Step 8: Define a cache cont.

- Notes on cache configuration
  - The dependence infrastructure in CoreGen derives local versus shared caches
  - Caches are not required to be homogeneous between layers
  - Designs can contain an unlimited number of caching layers




```
Caches:
  - Cache: Core0.L1.cache
    Sets: 2
    Ways: 8
    SubLevel: Core0.L2.cache
  - Cache: Core0.L2.cache
     Sets: 4
     Ways: 8
     SubLevel: Shared.L3.cache
  - Cache: Shared.L3.cache
     Sets: 2
     Ways: 8
```

~/CoreGenTutorials/LEVEL1/Step8

Step 8: Define a cache cont.

```
$> cgcli --ir ./BasicRISC.yaml --verify
$> cgcli --ir ./BasicRISC.yaml --pass
```

- Executing the cgcli passes against our design in Step 8 will uncover dangling nodes and dangling regions
- Our caches are not yet connected to cores!

```
================================================================
 CoreGen Pass Summary
================================================================
PASS                           TIME (secs)                PASS/FAIL
StatsPass......................0.00223899...................PASSED
MultSoCPass....................0.000132084..................PASSED
ICacheCheckerPass..............0.00013113...................PASSED
L1SharedPass...................0.000131845..................PASSED
RegIdxPass.....................0.000142097..................PASSED
RegFieldPass...................0.000139952..................PASSED
RegSafetyPass..................0.000766039..................PASSED
CoreSafetyPass.................0.000130892..................PASSED
CommSafetyPass.................0.00013113...................PASSED
RegClassSafetyPass.............0.000181913..................PASSED
CacheLevelPass.................0.000129223..................PASSED
DanglingNodePass...............8.4877e-05...................FAILED
DanglingRegionPass.............0.000338078..................FAILED
EncodingCollisionPass..........0.001158.....................PASSED
MandatoryFieldPass.............0.00101089...................PASSED
EncodingGapPass................0.000163078..................PASSED
PInstSafetyPass................0.000128984..................PASSED
CommSafetyPass.................0.000128984..................PASSED
```

Failed Analysis Passes

~/CoreGenTutorials/LEVEL1/Step8

Step 9: Define a core

- Now that we have a cache (or cache hierarchy), lets define a core
- Cores are containers that include lower-level modules and data/control paths
- Cores implement an instruction set with a set of register classes
  - The register sharing attributes are defined by the individual registers
- Cores can also enable symmetric multi-threading (SMT) by specifying multiple "ThreadUnits"
  - The default is '1' if unspecified
- Define a *Cores* node block

- Parameters:
  - *Core*: Name of the respective core. Must be unique
  - *ThreadUnits*: The number of symmetric thread units within the core
  - *Cache*: The connected cache hierarchy (usually L1)
    - This is technically optional, but definitely useful for at least instruction cache space
  - *ISA*: Links the respective instruction set to be utilized by the core
  - *RegisterClasses*: List of register classes required to implement the ISA

```
Cores:
  - Core: core0
    ThreadUnits: 1
    Cache: Core0.L1.cache
    ISA: BasicRISC.ISA
    RegisterClasses:
      - RegClass: GPR
      - RegClass: CTRL
```

~/CoreGenTutorials/LEVEL1/Step9

Step 9: Define a core cont.

- Lets verify our work thus far
- Utilize the “cgcli” tool to run all of our passes
- One failure
  - *CoreSafetyPass*
    - Identified our core that is not connected to an SoC!
    - This is fine for extension and module projects
    - For SoC projects, we need a top-level SoC container

```
$> cgcli --ir ./BasicRISC.yaml --verify
$> cgcli --ir ./BasicRISC.yaml --pass
```

```
PASS                          TIME (secs)                 PASS/FAIL
StatsPass.....................0.00226903....................PASSED
MultSoCPass...................0.000133991...................PASSED
ICacheCheckerPass.............0.000138044...................PASSED
L1SharedPass..................0.000139952...................PASSED
RegIdxPass....................0.000144005...................PASSED
RegFieldPass..................0.000140905...................PASSED
RegSafetyPass.................0.00075388....................PASSED
CoreSafetyPass................0.000136137...................FAILED
CommSafetyPass................0.000130892...................PASSED
RegClassSafetyPass............0.000183105...................PASSED
CacheLevelPass................0.000131845...................PASSED
DanglingNodePass..............7.70092e-05...................PASSED
DanglingRegionPass............0.000337124...................PASSED
EncodingCollisionPass.........0.00116086....................PASSED
MandatoryFieldPass............0.00100088....................PASSED
EncodingGapPass...............0.000159025...................PASSED
PInstSafetyPass...............0.000133038...................PASSED
CommSafetyPass................0.00013113....................PASSED
```

```
...Executing CoreSafetyPass
CoreSafetyPass : Identified a core not connected to the SoC; Core=core0
```

~/CoreGenTutorials/LEVEL1/Step9

Step 10: Define an SoC

- The final step is to define a top-level system-on-chip (SoC) container
  - Generally only utilized in "soc" designs
  - Generally only one SoC per design
- Define a *Socs* node block
- Parameters:
  - *Soc*: Defines the name of the top-level SoC container
  - *Cores*: Defines the cores that are contained within the SoC (from Step 9)

```
Socs:
  - Soc: BasicRISC.soc
    Cores:
      - Core: core0
```

~/CoreGenTutorials/LEVEL1/Step10

Step 10: Define an SoC cont.

```
$> cgcli --ir ./BasicRISC.yaml --verify
$> cgcli --ir ./BasicRISC.yaml --pass
```

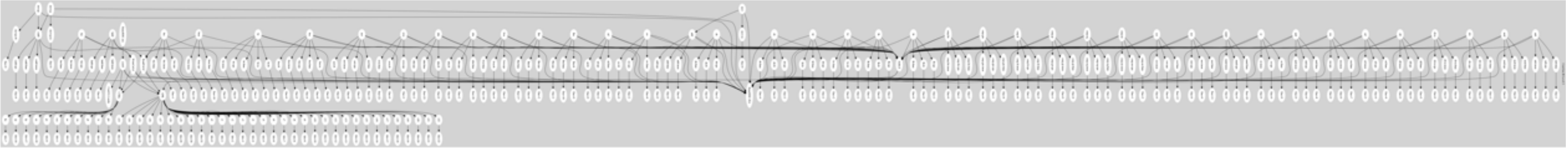
- Now that everything is defined, lets run the tools and check for issues
- Utilize the “cgcli” tool to run all of our passes
- Everything passes!

```
==========================================================
 CoreGen Pass Summary
==========================================================
PASS                          TIME (secs)                  PASS/FAIL
StatsPass...................0.00357604...................PASSED
MultSoCPass.................0.000149012..................PASSED
ICacheCheckerPass...........0.000146866..................PASSED
L1SharedPass................0.000151157..................PASSED
RegIdxPass..................0.000159979..................PASSED
RegFieldPass................0.000154018..................PASSED
RegSafetyPass...............0.000773907..................PASSED
CoreSafetyPass..............0.000148058..................PASSED
CommSafetyPass..............0.000145912..................PASSED
RegClassSafetyPass..........0.000196934..................PASSED
CacheLevelPass..............0.000143051..................PASSED
DanglingNodePass............0.000147104..................PASSED
DanglingRegionPass..........0.000355005..................PASSED
EncodingCollisionPass.......0.00117993...................PASSED
MandatoryFieldPass..........0.00103498...................PASSED
EncodingGapPass.............0.000172853..................PASSED
PInstSafetyPass.............0.000143051..................PASSED
CommSafetyPass..............0.000144005..................PASSED
```

~/CoreGenTutorials/LEVEL1/Step10

Step 10: Define an SoC cont.

Our fully encapsulated design with all the encodings expanded in the dependence graph!



~/CoreGenTutorials/LEVEL1/Step10

References

Where do I find more info?

Web Links

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

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
- System Architect Weekly Development Releases
  - https://github.com/opensocsysarch/SystemArchitectRelease

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