

Design Concepts with System Architect: Level 0

John Leidel

Chief Scientist, Tactical Computing Laboratories

ver 2019.03.19

Tactical Computing Laboratories



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

System Architect Overview

Modular, High-Level Design Concepts

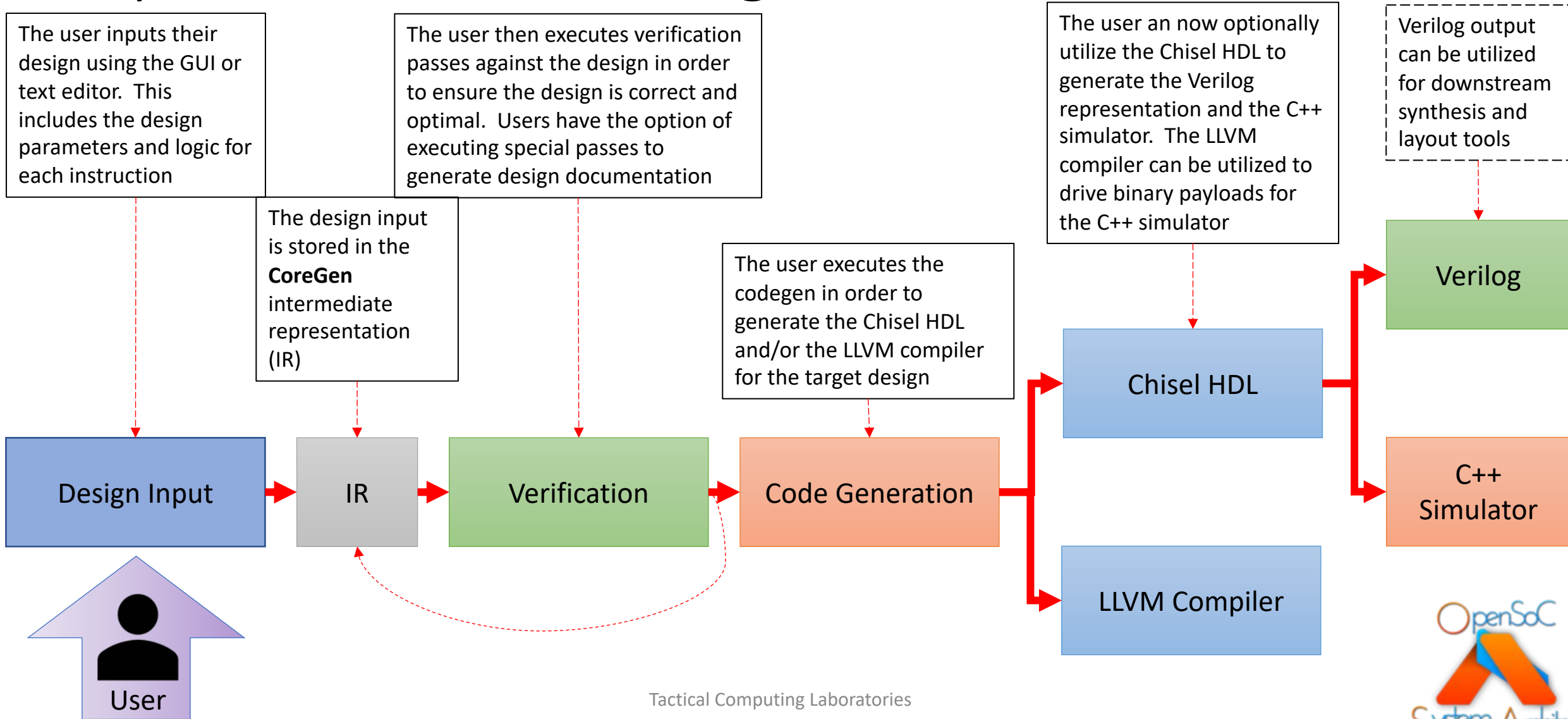
What is System Architect?

- A family of tools, APIs and associated infrastructure to permit users to rapidly develop multi-faceted hardware
- Utilizes a combination of modular hardware design reuse principles, object oriented development and dependence analysis techniques (compiler theory) to provide an infrastructure for:
 - **Design & Design Experimentation**
 - **High Level Verification**
- The artifacts generated by a System Architect design flow include:
 - **Chisel HDL and Verilog RTL**
 - **C++ cycle-based simulator**
 - **LLVM compiler**

What is System Architect NOT?

- System Architect is not the latest C-to-gates tool
 - It permits rapid design, verification and reuse
 - It does not auto-generate hardware based upon application code
- System Architect still relies upon the user to utilize reasonable design concepts
 - System Architect will **not** auto-generate optimized designs based upon unreasonable inputs
 - Users need to have a concept of the physical platform (FPGA, ASIC, etc)
- System Architect does **not** currently have a notion of physical layout
 - The generated output will not include LUT counts, physical design dimensions or power estimates
 - External FPGA/ASIC tools are required for this level of detail

System Architect Design Flow



Typical System Architect Design Flows

Rapid ISA Development

- Rapid development of ISA's with backend RTL and LLVM compiler as artifacts
- Cycle-based simulator that supports immediate experimentation
- Rapid design evaluation and prototyping
- High level verification of design before synthesis

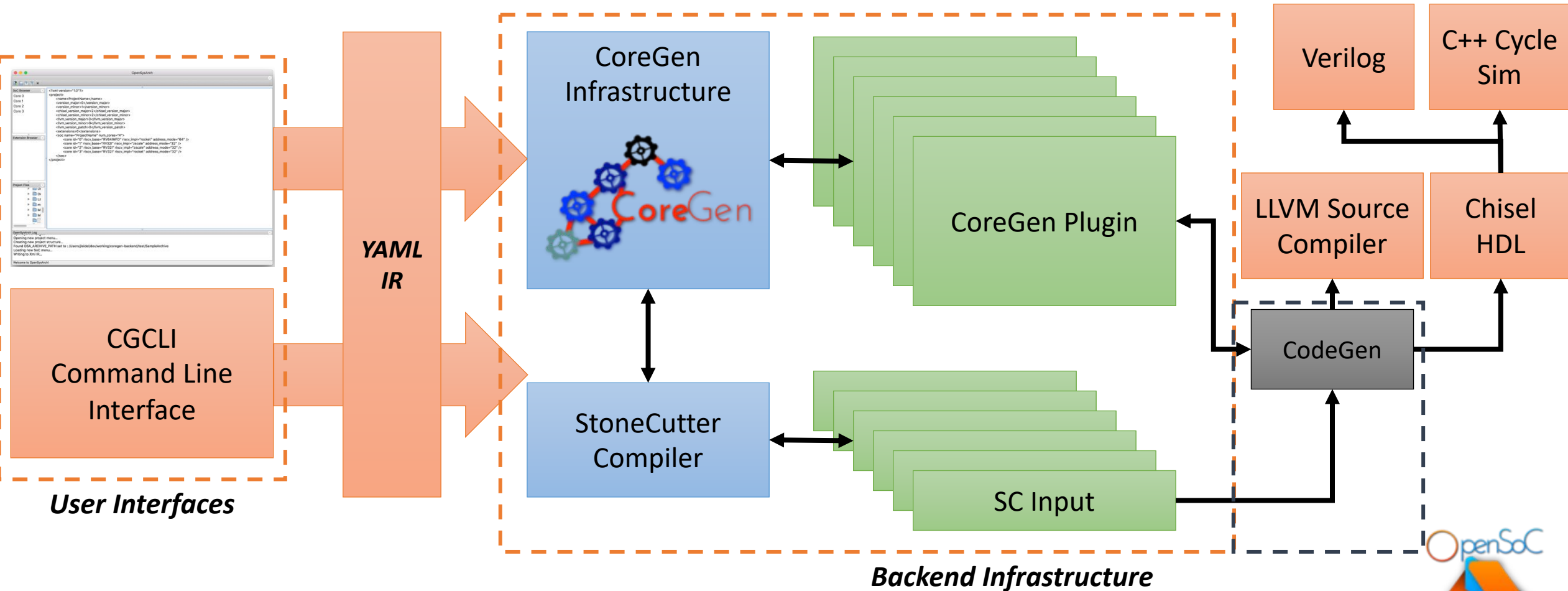
Modular Design

- Multi-module design and integration
- Integration with other System Architect designs (sub-designs)
- External module integration
- Plugin support for custom-modularity

Compiler Environment

- Auto-generated LLVM compiler infrastructure from design parameters
- Modern C/C++ frontend support
 - Other frontends can be supported as well
- Re-useable as LLVM mainline continues to develop
 - Re-spinning entire compiler based on new LLVM versions is automated

System Architect Infrastructure



How does it work?

- We input designs in the user-facing tools and generate **CoreGen Intermediate Representation (IR)**
- The IR preserves the natural dependencies within the design
 - Register classes depend upon registers
 - Instruction formats depend upon register classes
 - Instruction sets depend upon instructions
 - Cores require instruction sets
- We execute high level verification “passes” against the IR
 - Similar in design to traditional compiler passes
 - Walks the IR dependence graph and derives properties of the design
 - Reports issues in the design infrastructure, outputs interesting data or optimizes the design infrastructure
- Users implement instructions in StoneCutter instruction implementation language
 - C-like integrated with CoreGen IR to define a single instruction
 - Optimized by a traditional compiler flow to generate Chisel HDL for a single instruction
- Following the high level verification phase, we execute generate downstream code (code generation)
 - Generates Chisel HDL
 - Compiled down to Verilog & C++ cycle-based simulator
 - Generates LLVM compiler for the target design



Example dependence graph from CoreGen design input

CoreGen IR Passes

- Passes can be selectively enabled or disabled by the user (just like a normal compiler)
- Four types of passes
 - **Analysis**
 - Analyze the connectivity and the structure of the graph
 - Reports back the identified state to the user
 - DOES NOT modify the graph (IR is unchanged)
 - **Optimization**
 - Optimizes connectivity and structure of the graph
 - Much like Kennedy/Callahan dependence analysis/optimization
 - MODIFIES the dependence graph (IR is changed)
 - **Data**
 - Generates statistical data/output based upon the structure/content of the graph
 - EG, outputs a LaTeX specification document based upon the respective ISA
 - DOES NOT modify the graph (IR is unchanged)
 - **System**
 - Mixtures of each of the aforementioned pass types
 - Must be manually instantiated by the tools/users
- The internal representation of the IR is stored as a directed acyclic graph (DAG)
- Contains four levels that describe varying levels of detail
 - This is a performance optimization
 - Each subsequent level is a superset of the previous level
 - $Level^{N+1} \supseteq Level^N$

CoreGen IR Passes: Example Analysis Passes

RegSafetyPass/RegIdxPass

- Find inconsistencies between registers within the same register file
- Multiple registers with the same index
- Registers with missing indices
- Registers with prescribed values > than their bit width
- Sub-register fields with overlapping names
- Sub-register fields with overlapping bit field definitions

EncodingCollisionPass

- Identifies potential ISA encoding issues
- Utilizes all instruction formats within an ISA
- Builds a table of every known instruction within the ISA
 - Initializes a bit vector for every instruction using the encoding (eg, opcode), immediate values and register fields (bitmask)
- Examines collisions in the “loaded” encodings

CoreGen IR Passes: Example Data Passes

StatsPass

- Walks the entire dependence graph and collects data about the connectivity of the graph and incidence of the nodes
- Reports the number of adjacent dependent nodes for each node in the graph (outdegree)
- Reports final counts of the incidence of each node type

SpecDoc

- Walks the entire dependence graph and collects data about the encoding infrastructure of the ISA and portions of the micro architecture
- Outputs a specification document in LaTeX that details each of:
 - Register Classes
 - Register Encodings
 - Subregister encodings (bit fields within register)
 - Instruction Formats
 - Instruction Encodings
 - Master instruction table

CoreGen IR Passes: Example System Passes

SafeDeletePass

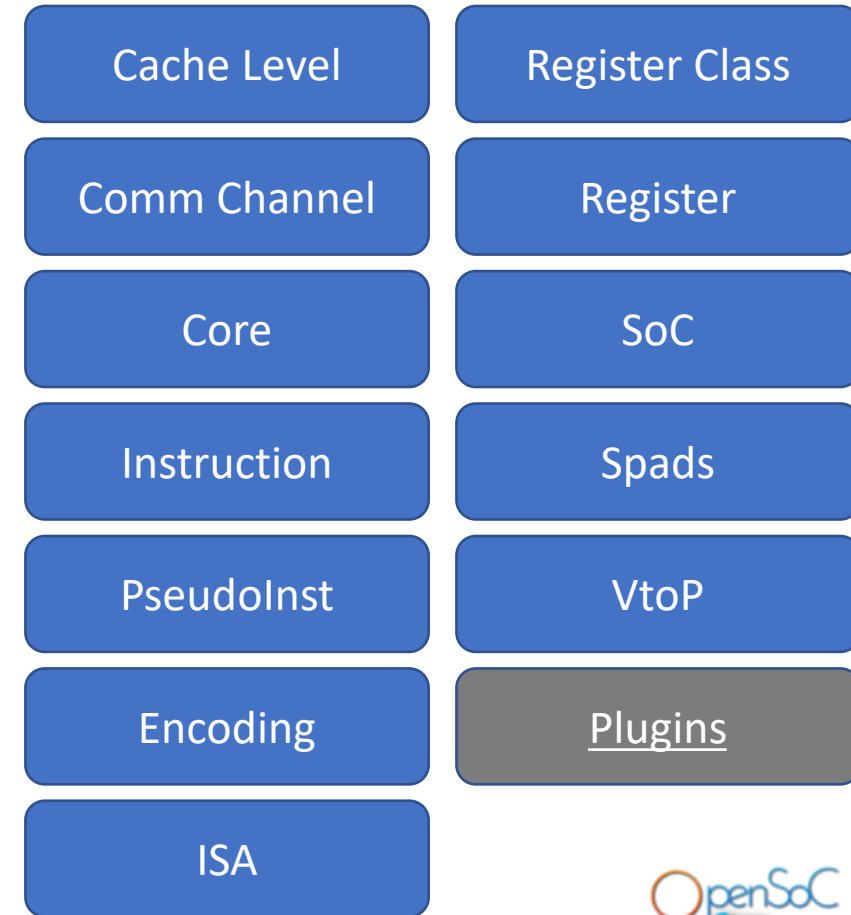
- Determines whether removing a node from the graph is safe
- For example:
 - "If I remove this register, will I break the dependence graph?"
 - "If I remove this encoding format, will I break the dependence graph?"
- Generally utilized within other tooling, but can be utilized directly by the user

ASPSolverPass

- Utilized when an existing analysis pass does not find a specific corner case
- Can be used to "programmatically" define new dependence solvers using a specific syntax
- Can be utilized as design constraints and/or regression tests to ensure specific functionality/connectivity exists in a design

CoreGen Infrastructure

- All hardware modules/units are defined as DAG nodes
- Dependence graph between nodes is “lowered” in multiple stages in order to expose increasing levels of complexity
 - Similar to Open64 notion of multi-dimensional IR
- DAG Levels:
 - Level 0: Basic node connectivity
 - Level 1: Expands “extension” nodes to contain all their children
 - Level 2: Expands all communication links
 - Level 3: Expands all instruction and register encodings



What type of nodes in the CoreGen IR?

- **SoC Nodes**: Defines a top-level system on chip (one per design)
- **Core Nodes**: Defines a single core with associated ISA and dependent nodes
- **ISA Nodes**: Defines an instruction set architecture container
- **Instruction Format Nodes**: Defines an instruction format with all of its sub-fields
 - Sub-fields have properties that define encoding fields, register fields, immediate value fields, etc
- **Instruction Nodes**: Defines a single instruction based upon a prescribed instruction format.
 - Ability to define encodings, register classes for register fields, immediate values, etc

What type of nodes in the CoreGen IR (cont)?

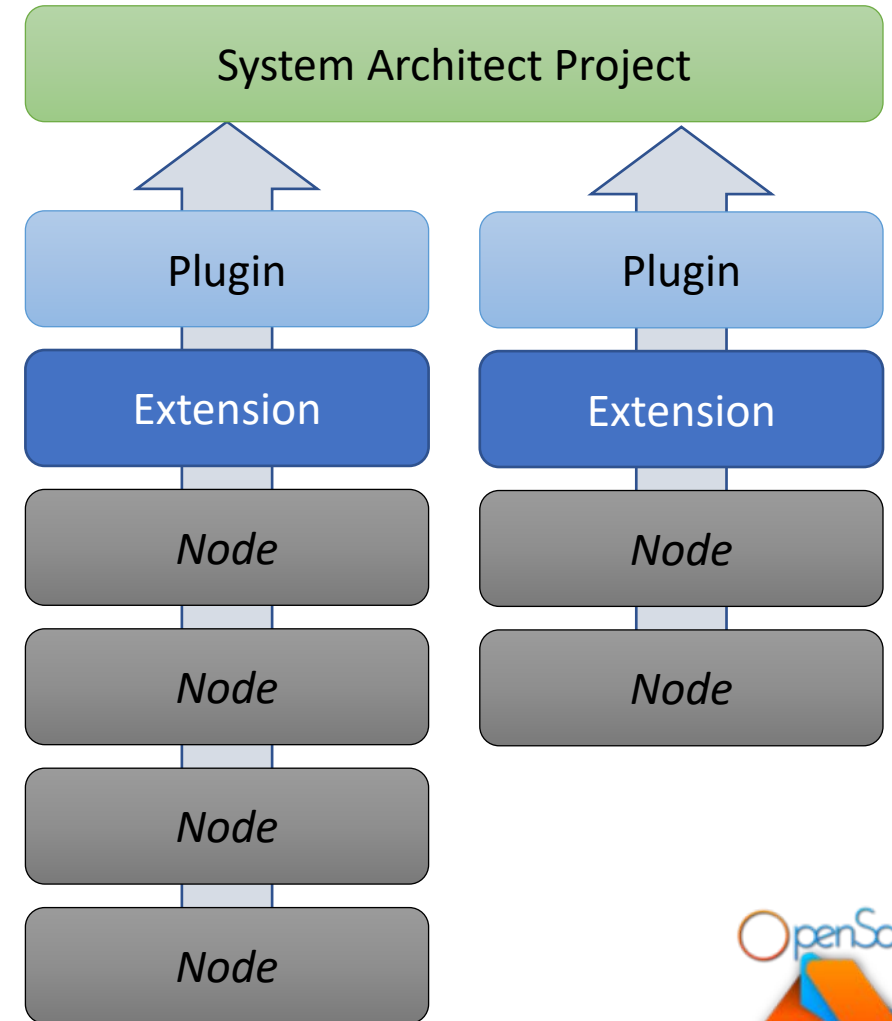
- **Pseudo Instruction Nodes**: Defines specific encodings for existing instructions
 - Ability to define prescribed encoding fields as static (eg, immediate = 0x00)
 - DOES NOT induce instruction encoding collisions
 - EG: "MOV RT, Ra" = "ADD RT, Ra, \$0"
- **Register Class Nodes**: Defines a container node with multiple dependent registers. Register indices within a register class cannot overlap
- **Register Nodes**: Defines a single register node with an index, bit width, sub-register fields and attributes (RO, RW, etc)
- **Encoding Nodes**: Define encodings for parent nodes (EG, register encodings)
- **Cache Nodes**: Defines a single cache layer
 - Can be interconnected into multi-level caches

What type of nodes in the CoreGen IR (cont)?

- **Communication Channel Nodes**: Interconnect multiple nodes via on-chip data+control paths
 - Multiple topologies supported
- **Scratchpad Nodes**: Represent addressable on-chip scratchpads
- **Memory Controller Nodes**: Represents a basic memory controller with multiple input/output ports
- **Virtual to Physical Nodes**: Handles virtual to physical memory translation
- ****Extension Nodes**: Special node type that permits users to “import” other CoreGen-developed project artifacts into other projects
 - Accelerators are excellent examples of extensions
- ****Plugin Nodes**: Special node type that represents a third-party templated design
 - Can contain unlimited number of special properties (not defined by standard CoreGen IR spec)
 - May also contain custom code generation facilities to output any style of HDL/RTL, etc

CoreGen Plugins

- CoreGen plugins are containers for self-contained extensions
 - Each plugin is effectively its own template
 - Bundled as a shared library (can be licensed and distributed outside of System Architect)
- These can be:
 - Cores, ISAs, cache modules, periphery components, etc
- Each plugin can drive unique “code generators” to modify their internal HDL state and/or generate the source compiler
- Projects can import any number of plugins
- DAG analysis works across plugins



CoreGen IR Specification

- IR 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 CoreGen library tree
 - NO changes to the spec are accepted without support in CoreGen
- Entire IR spec is documented with examples
- Latest revision:
 - <http://www.systemarchitect.tech/index.php/coregenirspec/>

What now?

- **Level 1 Tutorial:** Describes basic design concepts and walks through the initial definition of a RISC-like design
- **Level 2 Tutorial:** Implementing individual instructions using the StoneCutter language and compiler
 - Extends the design from Level 1
- **Level 3 Tutorial:** Advanced design and implementation concepts
 - Extends the work done in Level 2
- **Level 4 Tutorial:** Building external plugins and integrating external RTL
 - How do we integrate existing IP?

What do you need to continue?

- Linux/OSX system with the tools installed
 - Prebuilt packages are available:
 - <https://github.com/opensocsysarch/SystemArchitectRelease>
- Text editor
 - VIM, Emacs, Notepad, etc
- For those seeking to use the GUI
 - Graphics environment (X11, OSX, etc)
- Basic knowledge of computer architecture
- Basic knowledge of software architecture

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