SoC
Design Methodology

Gedeon Nyengele

SoC Design Challenges - Bird's Eye View

- Design Space Exploration
- Hardware Construction
- Software Design
- Design Verification
- Simulation / Emulation
- Documentation

1. Design Space Exploration

- Can happen at different levels of abstraction:
  - untimed functional views
  - timed/approximately-timed views
- Can happen at different levels of integration:
  - before interconnect refinement
  - after RTL generation
  - etc.
- Is based on metrics:
  - performance, area, power estimation
- Requires a comparison method
  - IP comparison
  - strategy comparison
  - etc

Most, if not all, of this information can be obtained by:

- using modern HCLs (hardware construction languages) with annotations
- manually/separately describing needed information: requires spec languages

2. Hardware Construction

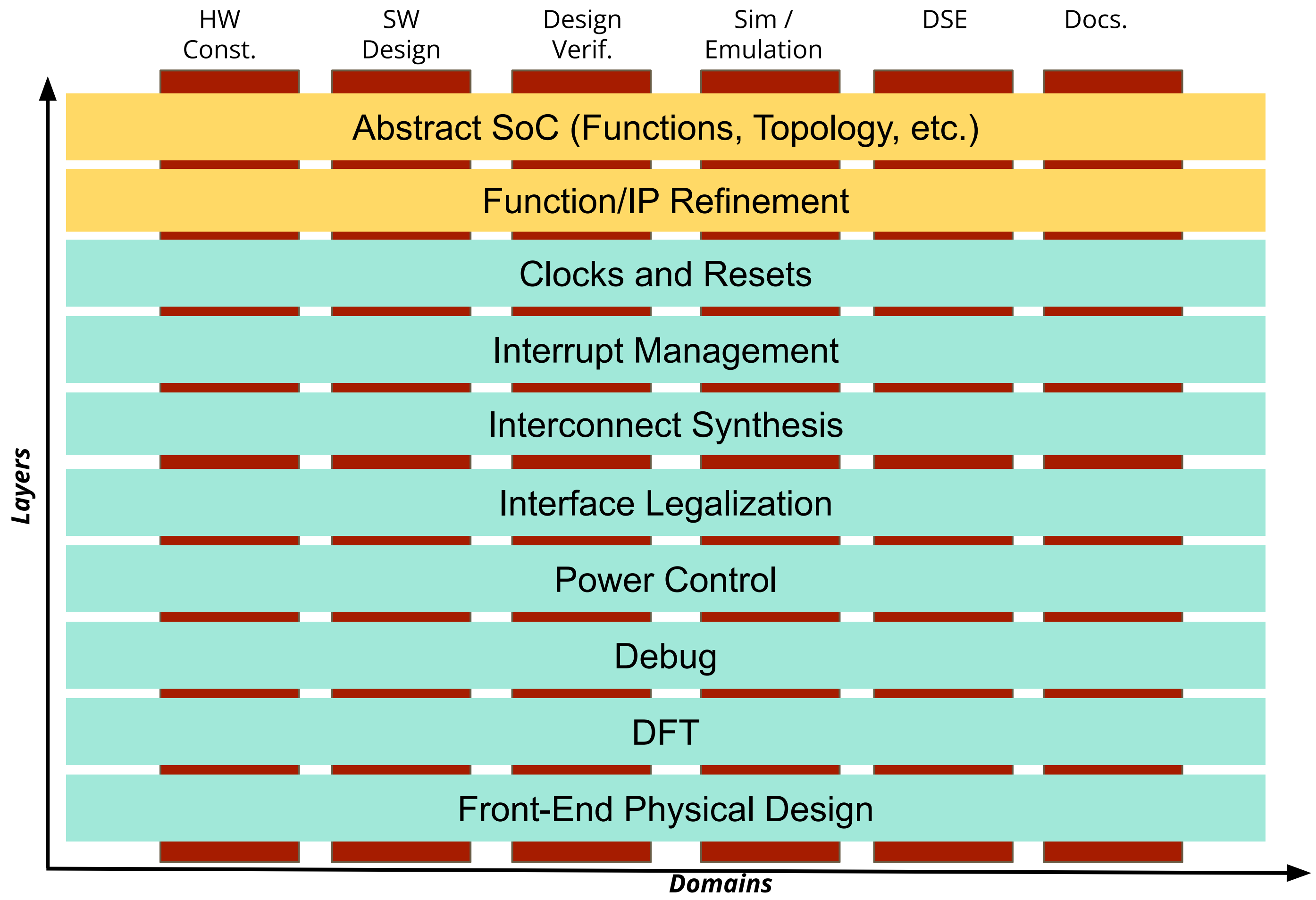
- Primarily concerned with connectivity issues
    - signal polarity
    - signal width
    - protocol/bus connections
    - protocol conversion
    - interconnect synthesis/instantiation
    - priority assignments
    - interrupts assignments
    - clock and reset management
    - power control
    - etc.

3. Software Design

- Clear component abstractions
- Generate collateral for various software methodologies
  - CMSIS
  - Xilinx (emulation)
  - Linux (device trees, configurations, etc.)
  - etc.
- Propagate changes from spec changes
- Maintain consistency with other verticals (verification, documentation, etc.)
- Reuse

Proposed SoC Design Methodology

- 2-5 policies per layer
- Want to focus most design effort in top 2 layers



Software Abstractions with Register Sequence (RegSeq) Specification

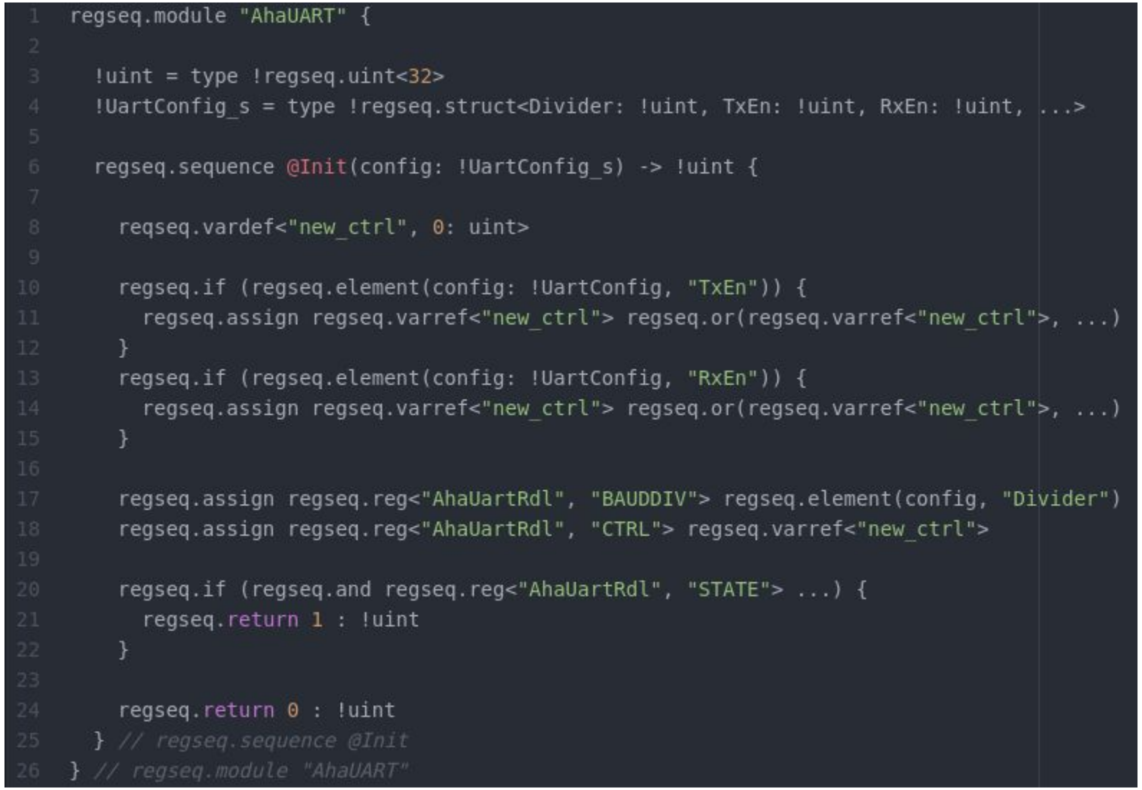
Motivation

- Register description languages (SystemRDL, IPXACT, Excel, etc.) are structural
  - Capture register properties (Read/Write, Address Offset, Fields, etc.)
  - Do not capture functional aspects of an IP (mode setting, control operations, data operations, etc.)
- Word documents are used to capture operation sequencing
  - Leads to misinterpretation
  - Makes it hard to manage and propagate changes in sequences
  - Makes it difficult to be consumed by automation tools
- Lack of open register sequence specifications
  - Difficult to share sequences and back-end tools among companies
- No reusable abstractions

RegSeq IR : Proposed Sequence Specification Language

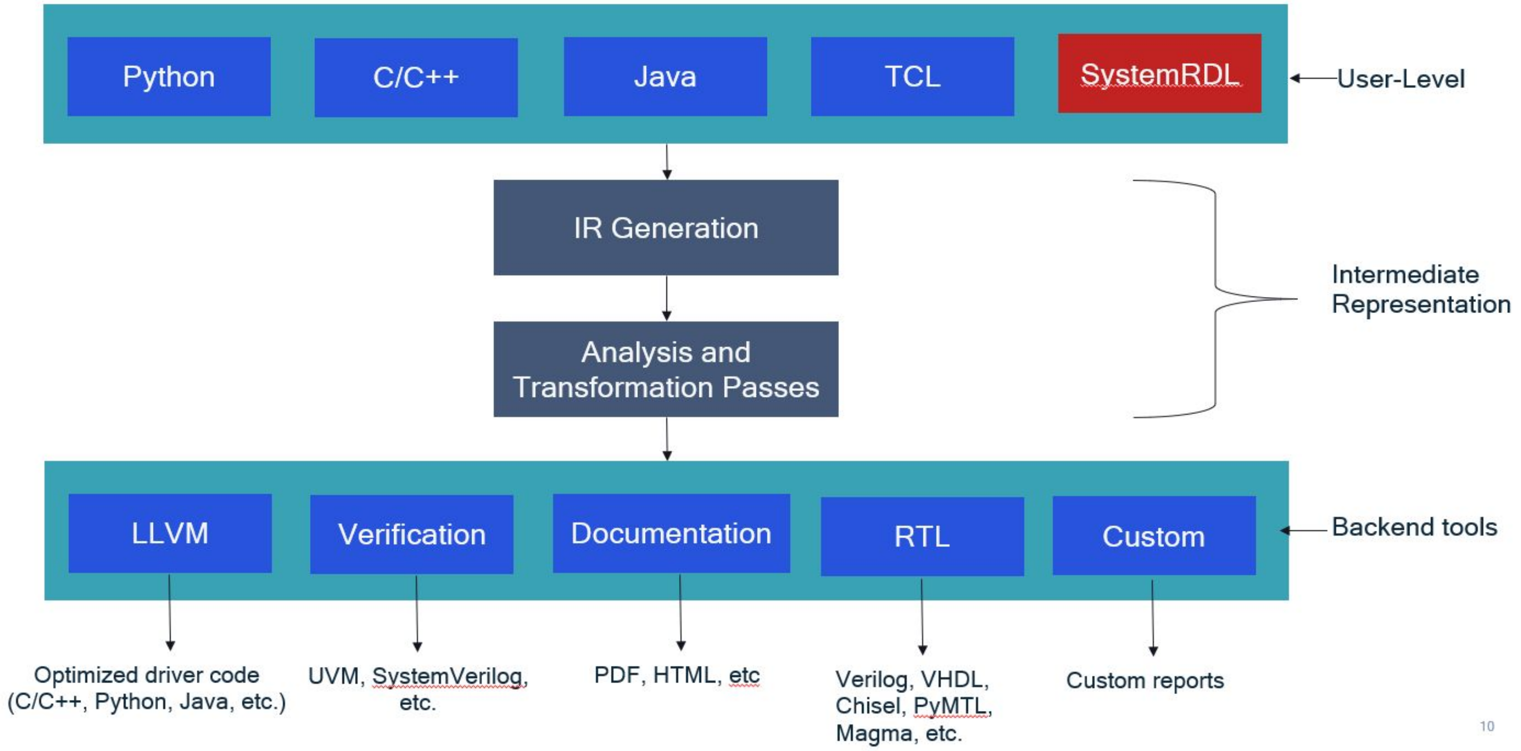
- MLIR dialect for capturing register sequences
- Supports a custom type system
  - Boolean, Integer, String, Array, Structure, Enumeration, and Pointer
- Supports a set of operations needed to capture a sequence
  - Arithmetic Ops: ADD, SUB, MUL, etc.
  - Logical Ops: AND, OR, etc.
  - Relational Ops: Less, Greater, Min, Max, etc.
  - Statements: Assignments, Loops, Conditionals, Variable Declaration, etc.
  - Ternary Ops: Select
  - Hierarchy Ops: Sequence Definition, Module Definition
  - Support some non-sequence intrinsics

RegSeq IR Example

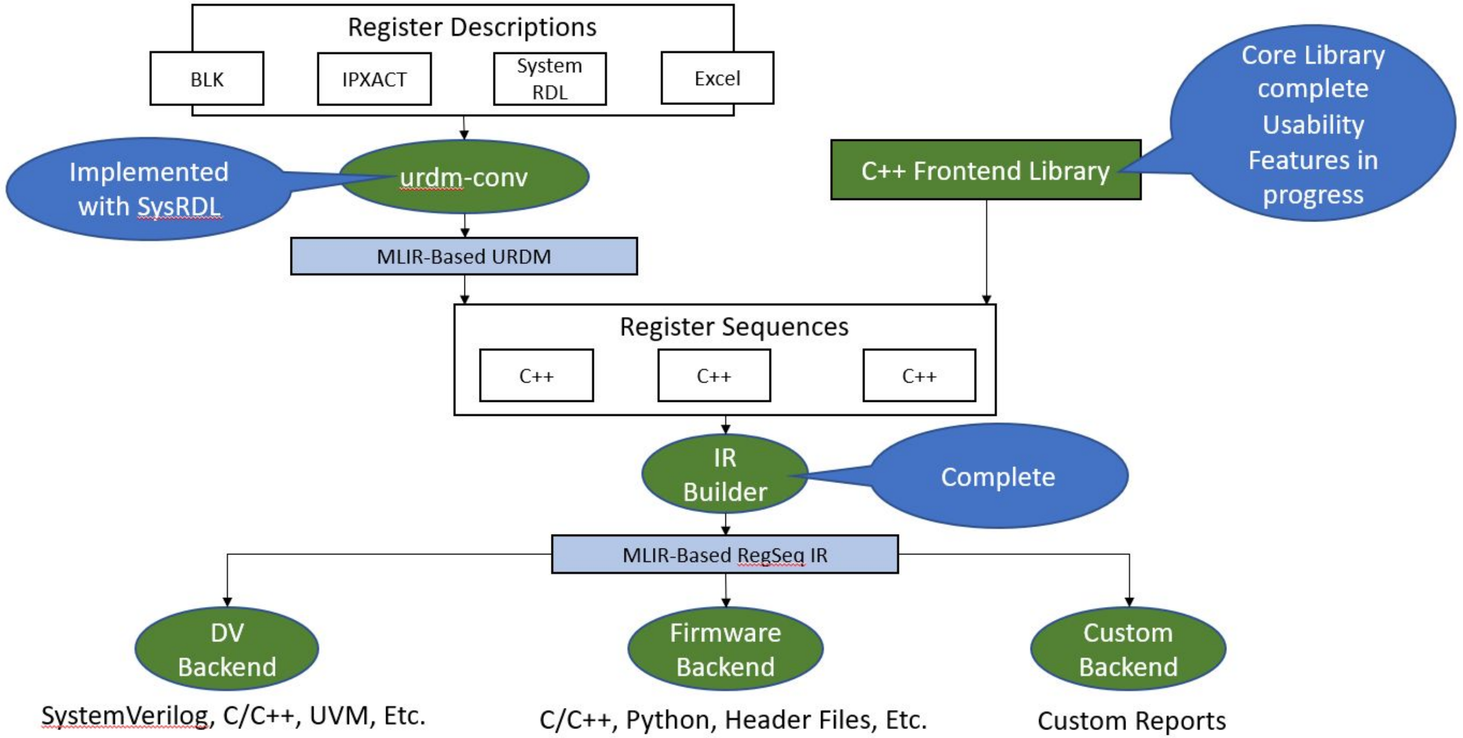


```
regseq.module "AhaUART" {

  !uint = type !regseq.uint<32>
  !UartConfig_s = type !regseq.struct<Divider: !uint, TxEn: !uint, RxEn: !uint, ...>

  regseq.sequence @Init(config: !UartConfig_s) -> !uint {

    reqseq.vardef<"new_ctrl", 0: uint>

    regseq.if (regseq.element(config: !UartConfig, "TxEn")) {
      regseq.assign regseq.varref<"new_ctrl"> regseq.or(regseq.varref<"new_ctrl">, ...)
    }
    regseq.if (regseq.element(config: !UartConfig, "RxEn")) {
      regseq.assign regseq.varref<"new_ctrl"> regseq.or(regseq.varref<"new_ctrl">, ...)
    }

    regseq.assign regseq.reg<"AhaUartRdl", "BAUDDIV"> regseq.element(config, "Divider")
    regseq.assign regseq.reg<"AhaUartRdl", "CTRL"> regseq.varref<"new_ctrl">

    regseq.if (regseq.and regseq.reg<"AhaUartRdl", "STATE"> ...) {
      regseq.return 1 : !uint
    }

    regseq.return 0 : !uint
  } // regseq.sequence @Init
} // regseq.module "AhaUART"
```

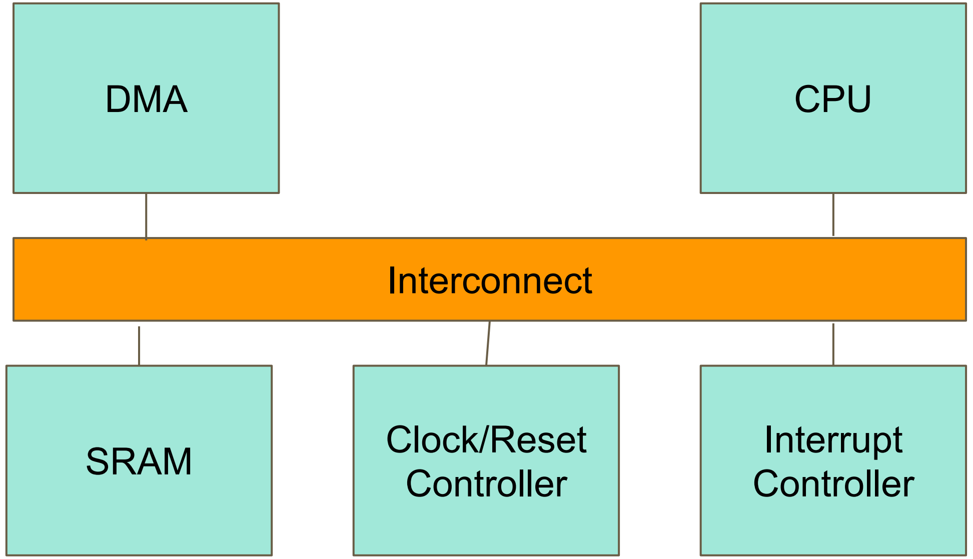
RegSeq Infrastructure Overview



RegSeq Infrastructure - Current State



Example : System Software Generation using RegSeq



https://github.com/thenextged/socgen/tree/main/admin/Presentations/AHA-01-25-2023

Next Steps

- Define higher-level component abstractions (Standardized Abstractions)
  - DMA
  - Interrupts
  - Interconnects
  - Etc.
- Capture sequences for a few representative components
- Generate software/drivers for different target environments:
  - Linux OS
  - Bare-Metal
  - RTOS