How HLS and SystemC is Delivering on its Promise of Design and Verification Productivity

Stuart Swan – HLS Technologist at Mentor, A Siemens Business
Mike Meredith – Product Engineering Group Director for Stratus HLS at Cadence Design Systems, Inc.
Matthew Bone – HLS and Design Methodology Expert at Intel Corp.
Rangharajan Venkatesan – Senior Research Scientist at NVIDIA Corp.

Speakers

Stuart Swan
HLS Technologist
Mentor, A Siemens Business

Mike Meredith
Product Engineering Group Director for Stratus HLS
Cadence Design Systems, Inc.

Matthew Bone
HLS & Design Methodology Expert
Intel Corp.

Rangharajan Venkatesan
Senior Research Scientist
NVIDIA Corp.

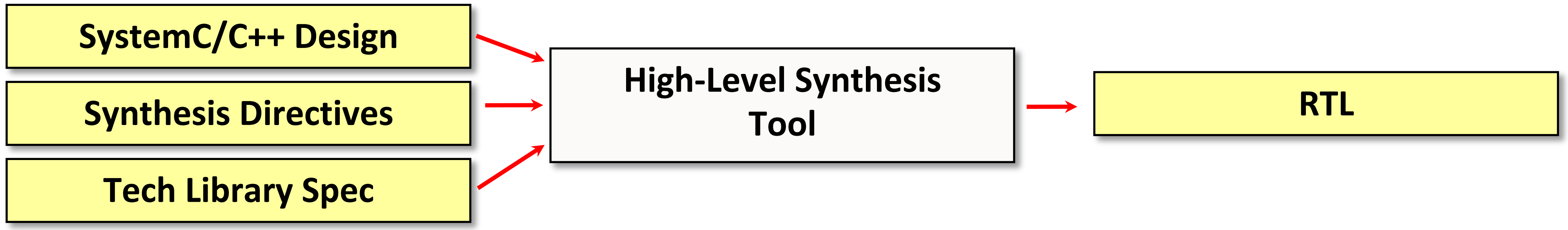
Agenda

- **Stuart Swan**: A Brief Introduction to High-Level Synthesis
- **Mike Meredith**: Accellera SystemC Synthesizable Subset Standard
- **Matthew Bone**: Techniques for Optimization of Power and Performance using HLS
- **Rangharajan Venkatesan**: MatchLib-based Object-Oriented HLS Methodology

Stuart Swan, Mentor, A Siemens Business

A BRIEF INTRODUCTION TO HIGH-LEVEL SYNTHESIS

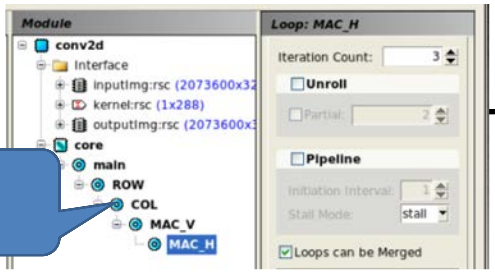
What does HLS do?



- The HLS tool:
  - precisely characterizes delay/area of all operations in design
  - schedules all the operations over the available clock cycles
  - can optionally increase latency
    - to share resources and reduce area
    - to enable positive slack at gate level
  - generates RTL that is functionally equivalent to input SystemC/C++
    - HLS automatically creates FSMs, muxes, etc for pipelining and resource sharing

Core HLS Optimization Concepts

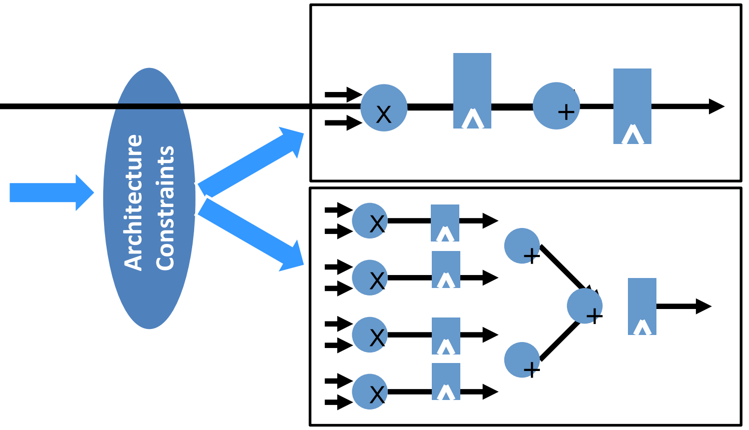
- Loop optimizations
  - Unrolling
  - Pipelining
  - Automatic merging



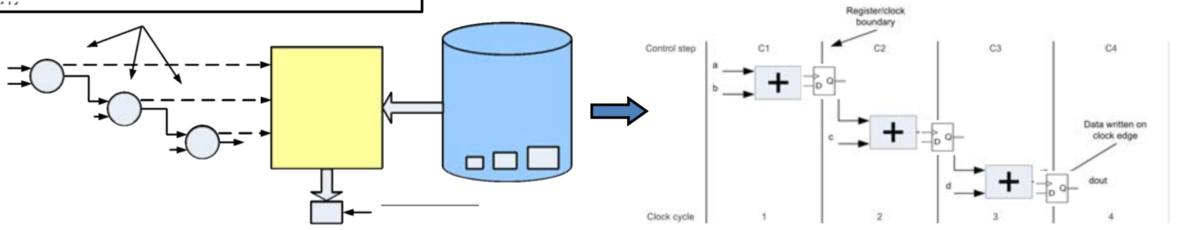
Module
- conv2d
  - Interface
    - inputImg:rsc (2073600x32
    - kernel:rsc (1x288)
    - outputImg:rsc (2073600x
  - core
    - main
      - ROW
        - COL
          - MAC_V
            - MAC_H

Loop: MAC_H
- Iteration Count: 3
- ☐ Unroll
- ☐ Partial: 2
- ☐ Pipeline
- Initiation Interval: 1
- Stall Mode: stall
- ☑ Loops can be Merged

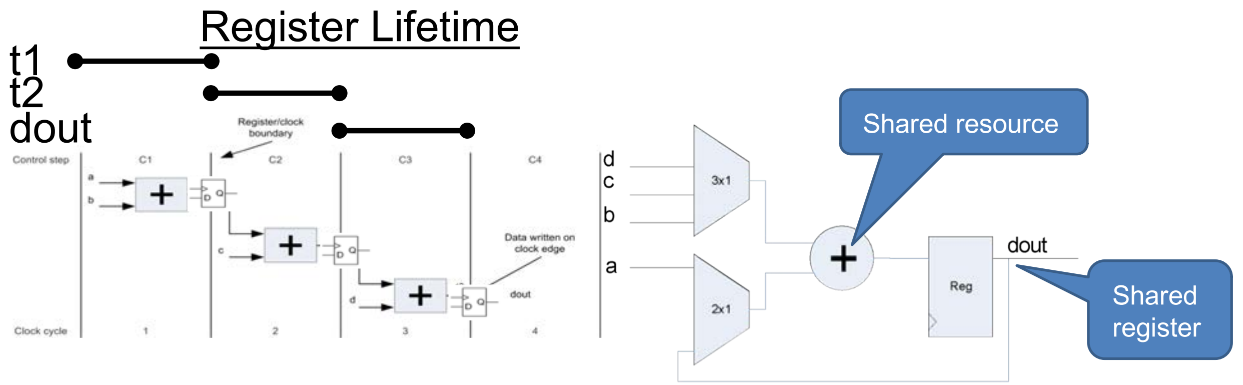
```
for (int i=0;i<4;i++) {
            acc += data_in[i] *
coef_in[i] ;
}
```



- Scheduling
  - Automatic timing closure based on target technology

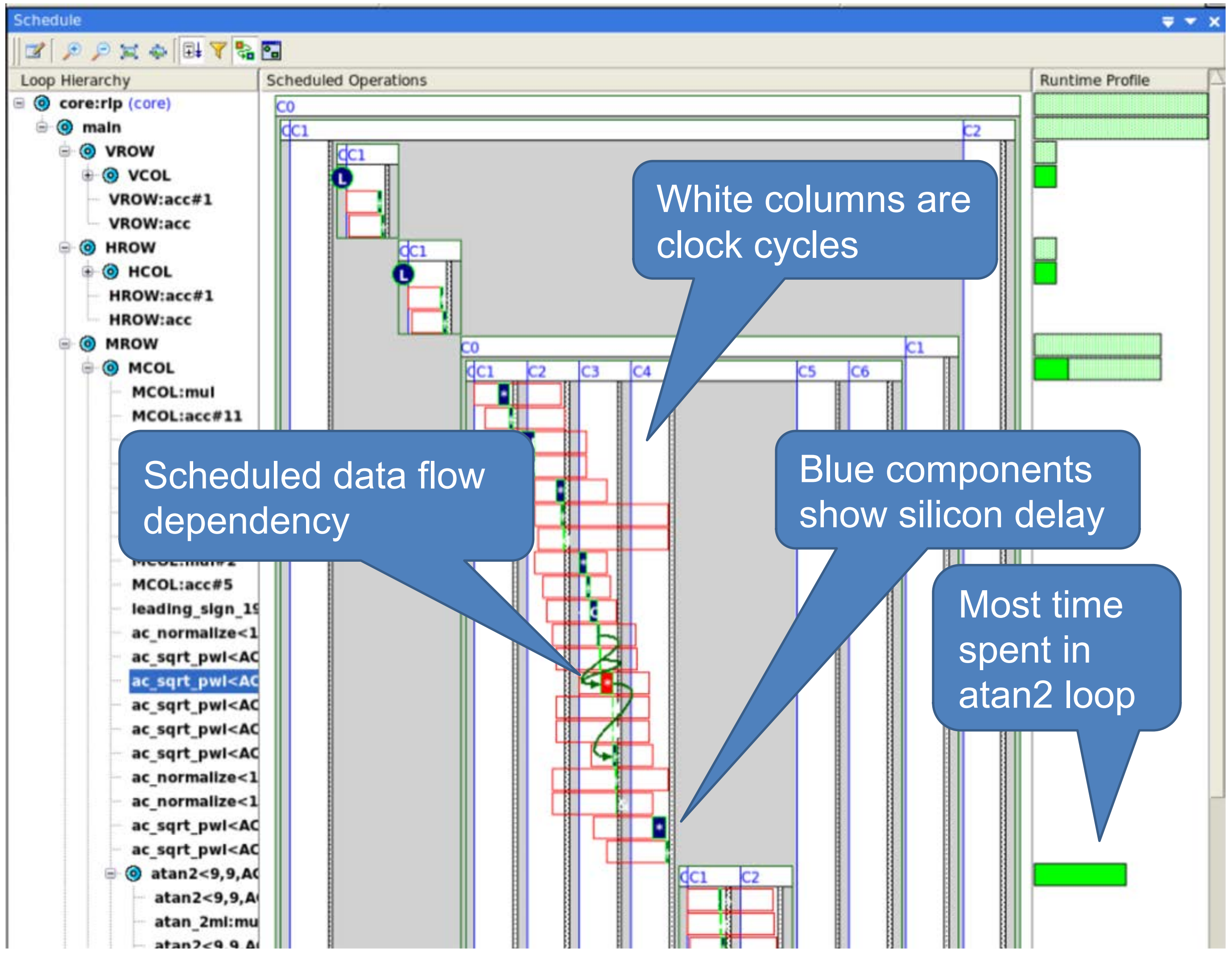


- Register and Resource sharing
  - Automatic lifetime and mutual exclusivity analysis and optimization



Graphical Analysis of HW Performance

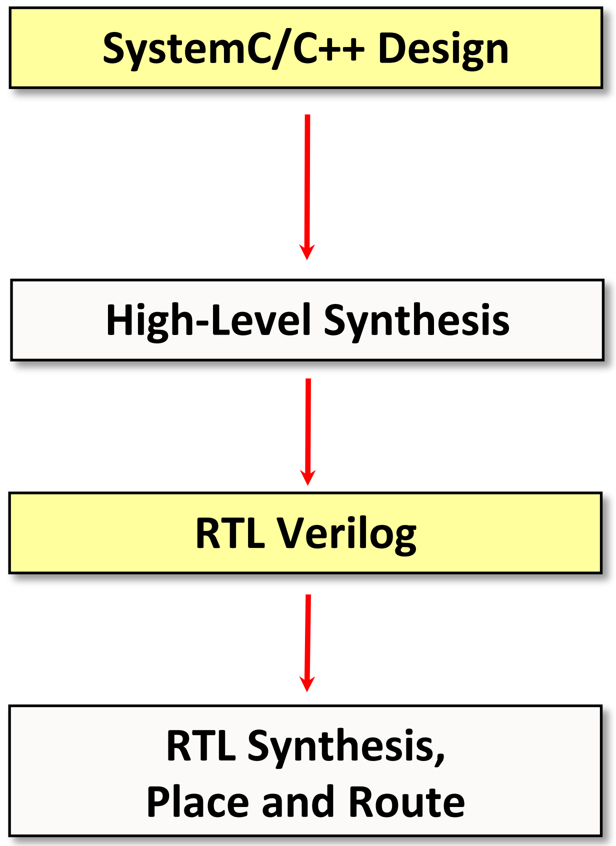
- Designers need to know what is happening during HLS
- HLS tools provide a view of the timed design
- Cross-probe back to source
- Understand dataflow, timing and dependencies



Coding Style is Key to Good QOR

- Need to properly capture HW architecture when you write your code!
  - I/O and memory architecture
  - Process structure
  - Useful to have rough idea what RTL will be generated and how expensive it will be.
    - E.g. still need to constrain bitwidths of variables (unlike SW programming)
  - Leave the details of loop pipelining, microarchitecture optimization, memory access scheduling, etc to the HLS tool to optimize

Typical HLS Flow



- Write, verify, refine C++ code
- Typical testbench environments: SystemC/C++, Matlab, SV UVM, …
- Perform code/functional coverage closure on SystemC/C++ model

- Explore and refine microarchitecture (using HLS directives)

- Achieve RTL code/functional coverage closure on RTL by leveraging existing SystemC/C++ testbench, SV UVM testbench, …

Why is HLS Beneficial?

- Model is higher abstraction:
  - Smaller, fewer mistakes
  - More easily reusable (parameterizable, etc)
  - Easier to retarget to different silicon technologies
- Verification is faster and easier
- Enables ability to quickly explore PPA tradeoffs
- QOR very competitive to hand-written RTL

Mike Meredith, Cadence Design Systems

ACCELLERA SYSTEMC SYNTHESIZABLE SUBSET STANDARD

General Principles

- Define a meaningful minimum subset
  - Establish a baseline for transportability of code between HSL tools
  - Leave open the option for vendors to implement larger subsets and still be compliant
- Include useful C++ semantics if they can be known statically – eg templates
- Covers behavioral model in SystemC for synthesis
- Covers RTL model in SystemC for synthesis
- Main emphasis of the document is on behavioral model synthesizable subset for High-Level Synthesis

Scope Of The Standard

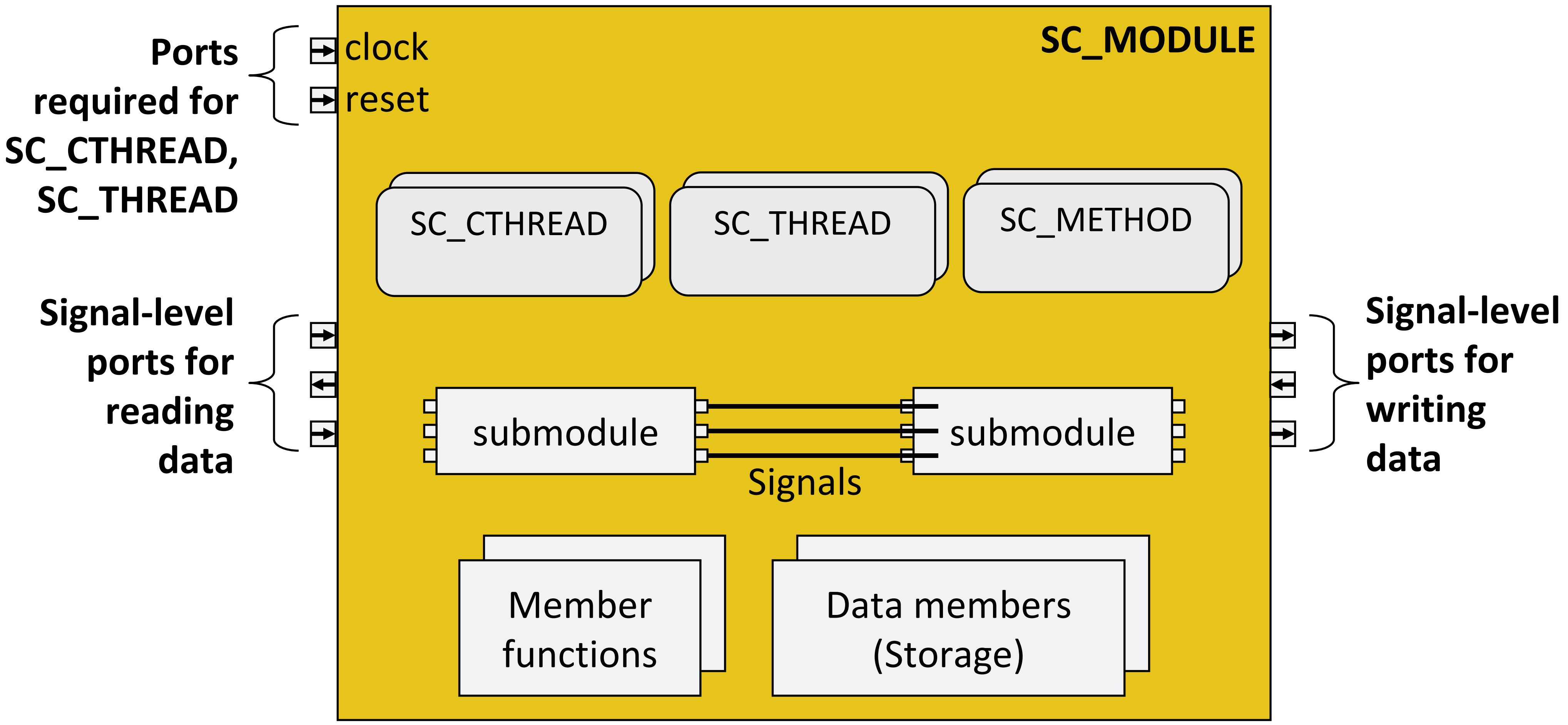
SystemC Elements

- Modules
- Processes
  - SC_CTHREAD
  - SC_THREAD
  - SC_METHOD
- Reset
- Signals, ports, exports
- SystemC datatypes

C++ Elements

- C++ datatypes
- Expressions
- Functions
- Statements
- Namespaces
- Classes
- Overloading
- Templates

Module Structure for Synthesis


Ports required for SC_CTHREAD, SC_THREAD
clock
reset
SC_MODULE
SC_CTHREAD
SC_THREAD
SC_METHOD
Signal-level ports for reading data
submodule
submodule
Signals
Signal-level ports for writing data
Member functions
Data members (Storage)

Specifying Clock and Reset

For synthesis, SC_THREAD can only have a single sensitivity to a clock edge

```
    Simple signal/port and level

        SC_CTHREAD( func, clock.pos()
);
        reset_signal_is( reset, true
);
        areset_signal_is( areset,
true );

        SC_THREAD( func );
        sensitive << clk.pos();
        reset_signal_is( reset, true
);
        areset_signal_is( areset,
true );
```

```
reset_signal_is( const sc_in<bool> &port, bool level )
reset_signal_is( const sc_signal<bool> &signal, bool level )
async_reset_signal_is( const sc_in<bool> &port, bool level )
async_reset_signal_is( const sc_signal<bool> &signal, bool level )
```

Module Declaration

- Module definition
  - SC_MODULE macro
    or
  - Derived from sc_module

  - SC_CTOR
    or
  - SC_HAS_PROCESS

- Classes that derive from modules are supported

```
// A module declaration
SC_MODULE( my_module1 ) {
        sc_in< bool> X, Y, Cin;
        sc_out< bool > Cout, Sum;
        SC_CTOR( my_module1 ) {…}
};

// A module declaration
SC_MODULE( my_module1 ) {
        sc_in< bool> X, Y, Cin;
        sc_out< bool > Cout, Sum;
  SC_HAS_PROCESS( my_module1 );
        my_module1(const sc_module_name
name )
                : sc_module(name)
 {…}
};
```

Use Of wait()

- For synthesis, wait(...) can only reference the clock edge to which the process is sensitive
- For SC_CTHREADs
  - wait()
  - wait(int)
- For SC_THREADs
  - wait()
  - wait(int)
  - wait(clk.posedge_event())
  - wait(clk.negedge_event())

For synthesis of SC_THREADs wait(event) must match the sensitivity of the clock edge

Data Types

- Primary SystemC data types
  - sc_int, sc_uint
    - Fixed bit widths <=64
  - sc_bigint, sc_biguint
    - Arbitrary fixed bit widths
  - sc_fixed, sc_ufixed
    - Fixed point with configurable rounding and saturation
- Additional SystemC data types
  - sc_bv
    - Bit vector, no arithmetic
  - sc_logic
    - Single-bit 4 state logic, but “X” and “Z” are not supported
  - sc_lv
    - Vectors of sc_logic
- C++ integral types
  - All C++ integral types except wchar_t
  - char is signed (undefined in C++)

Pointers

- Supported for synthesis
  - “this” pointer
  - “Pointers that are statically determinable are supported. Otherwise, they are not supported.”
  - If a pointer points to an array, the size of the array must also be statically determinable.

- Not Supported
  - Pointer arithmetic
  - Testing that a pointer is zero
  - The use of the pointer value as data
    - eg hashing on a pointer is not supported for synthesis

Matthew Bone, Intel Corp.

TECHNIQUES FOR OPTIMIZATION OF POWER AND PERFORMANCE USING HLS

Intro

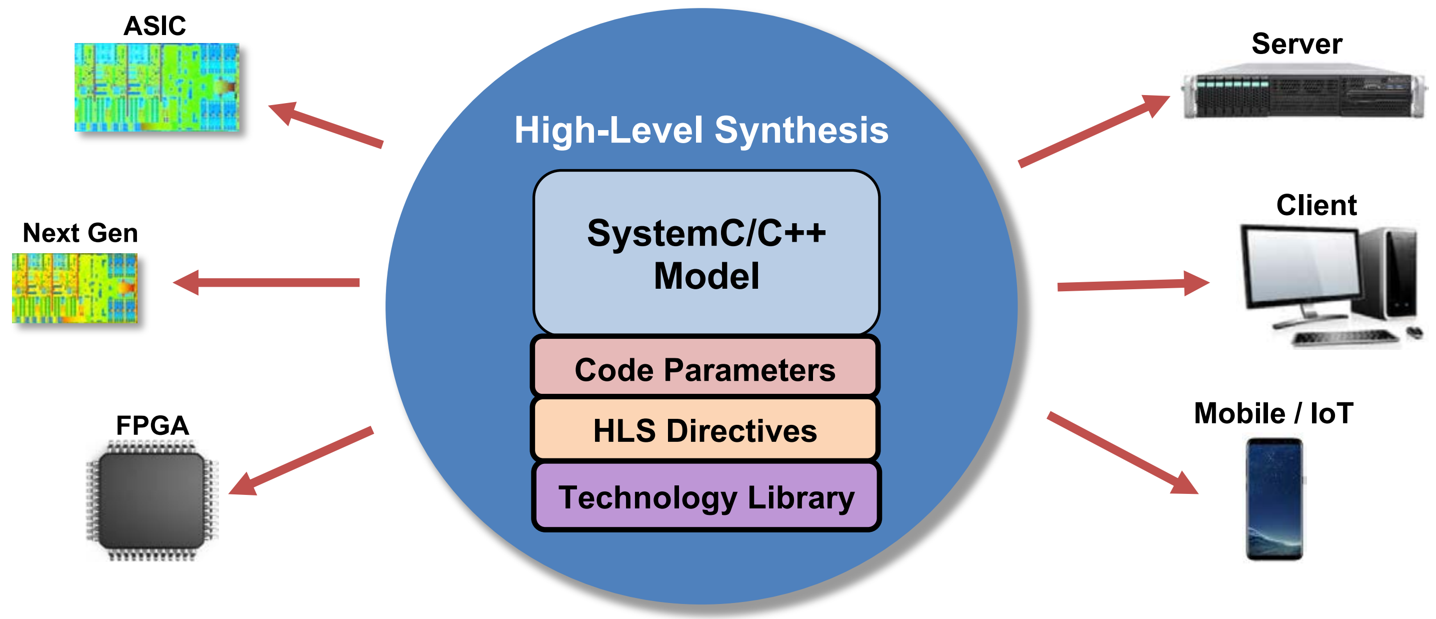
- Growth of HLS usage seen in many design domains at Intel
  - ***Powerful tools*** enable ***new design methodology***
  - Benefits for datapath and control dominated designs
  - Benefits break down roughly into three categories:
    - Schedule/TTM, power/performance, SW quality and code sharing
  - Case studies have been hand-picked to focus on power/performance and design space exploration

The Challenge

- Massive scale of designs and shortened schedules
  - Finishing functionality for many pieces is a large effort
  - Limited time available to explore power and performance
- Power is often the important metric
  - Mobile devices have passed desktop devices for web access
  - Datacenter costs (power, cooling)
- Using Traditional (RTL) Methodology
  - Power/performance estimates available later in the design phase
  - High-effort to make changes to address power/performance
  - Language improves (System Verilog), but still 35 year old abstractions

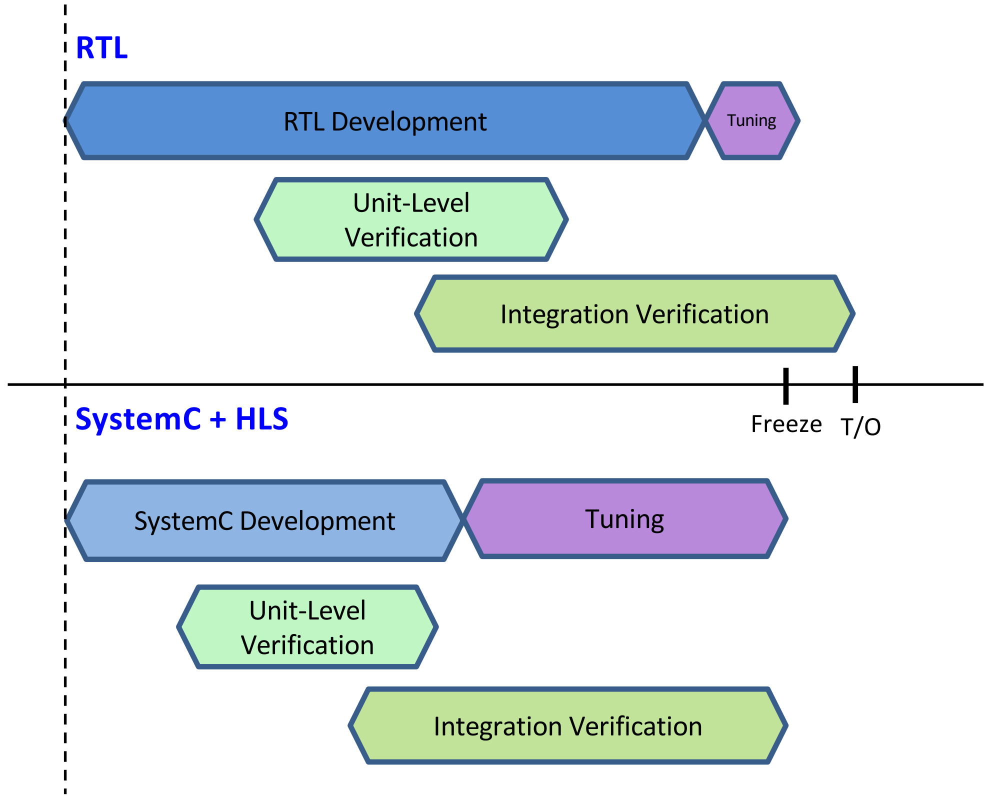
High-Level Synthesis

- SystemC source – more concise expression than RTL
  - Tool directives select parallelism, throughput, latency, storage types
  - Technology library provides parts information
    - e.g. the area and delay of a 16x16 multiplier



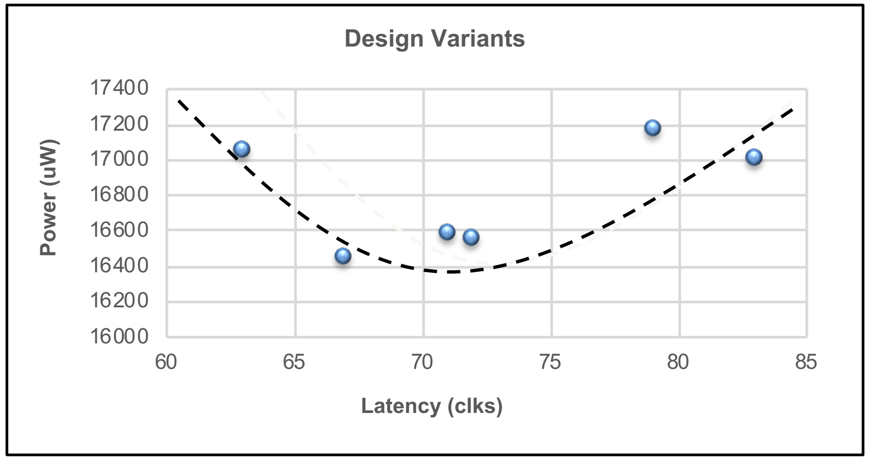
Why HLS? #1: Faster TTM

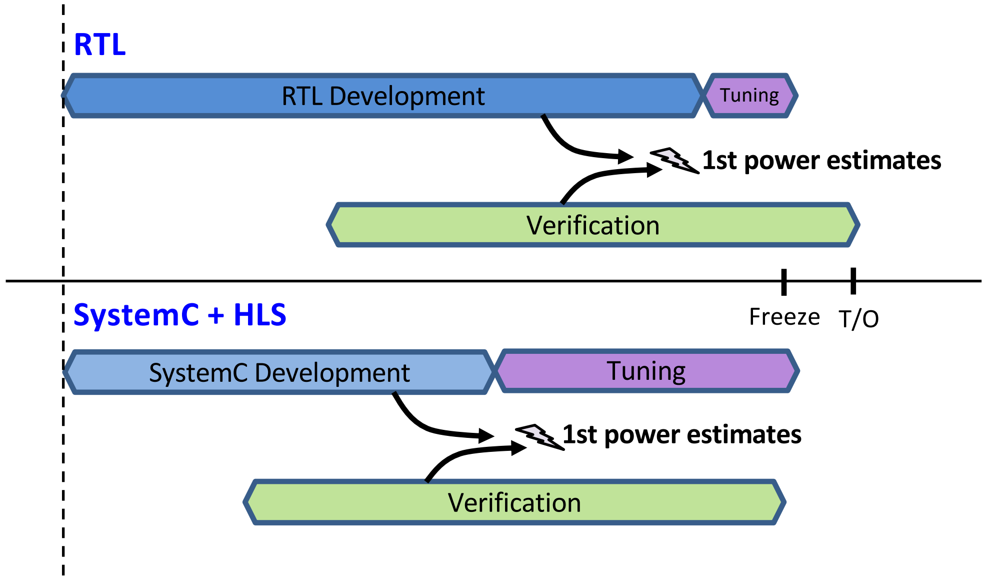
- Designer benefits:
  - Concise code (3-10X fewer lines than RTL)
  - Powerful directives for uarch decisions
  - Effort savings in tracking spec changes
- Verification benefits:
  - Begin earlier
  - Simulation acceleration (100X faster than RTL)
- Physical design benefits:
  - High quality RTL with good timing/routing characteristics



Why HLS? #2: Better PnP

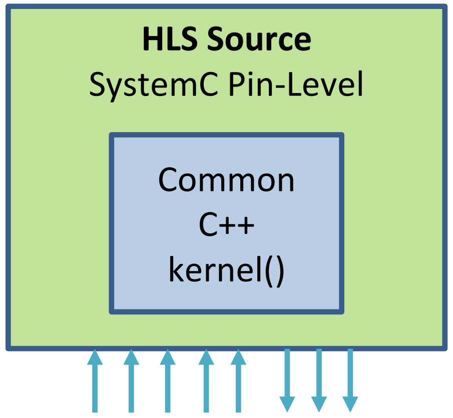
- Exploration
  - Low effort for arch/uarch variants
- Earlier power estimates
  - Time to react with architectural changes, or "tuning" HLS directives
- Custom RTL
  - Targets specific process node library (or FPGA)



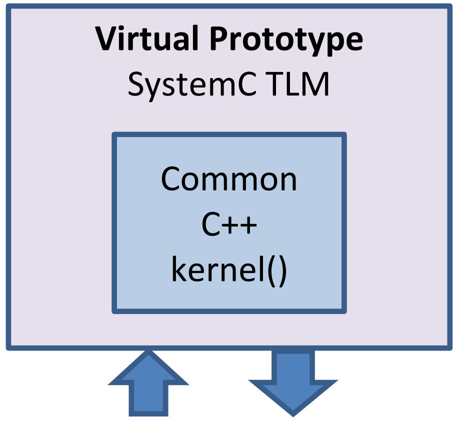


Why HLS? #3: SW/FW Quality

- SystemC language:
  - Opportunity for re-use of SC/HLS model as SW Virtual Prototype (or arch/perf model)
- Two styles of re-use seen:
  - Use SC/HLS model *as-is* for VP
  - Re-use the "core" portions of the SC/HLS model, with changes for VP speed-up



Use HLS model as-is:
Fast enough for SW/FW verification in many cases



Use "core" of HLS model:
Fastest simulation may require using TLM interfaces, and disabling some HW details

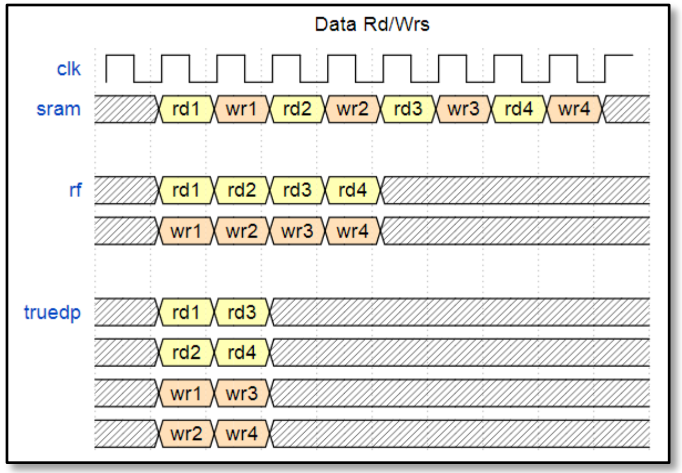
Exploration: Power/Performance

- Seeking best power/perf will be referred to as **design space exploration**, or just **tuning** later in the design phase
- Many RTL variants can be created by changing only HLS tool directives
  - "Micro-architectural" changes
  - Goal frequency, throughput, latency
  - Storage scheme (registers, latch arrays, RFs, SRAMs)
- Some exploration requires edits to the SystemC source
  - "Architectural" changes
  - Usually more significant changes to partitioning or ordering of an algorithm
  - Edits in SystemC are less effort than RTL

Pipelining II vs. Frequency

uarch	Area	Latency
PIPE_II4	16490	71
PIPE_II1	31707	55
PIPE_II1_192MHz	18847	12

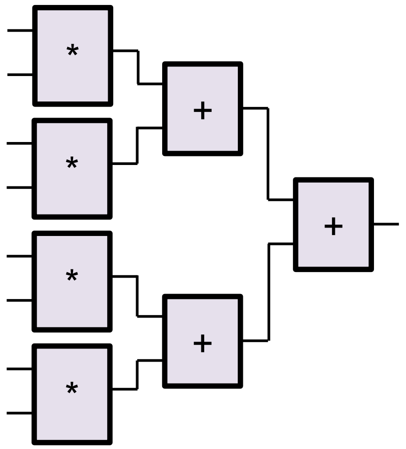
RF vs. SRAM vs. Multi-Port RAM



Exploration: Pipelining

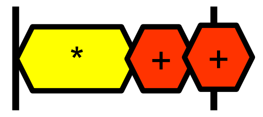
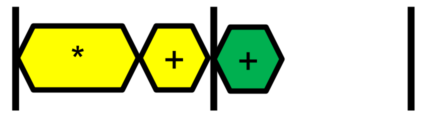
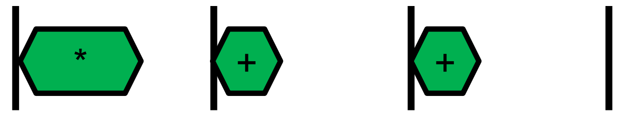
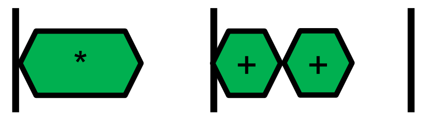
- Pipelining example
  - FIR filter

```
// Sum of taps*coefs
sum = t0*coef0 + t1*coef1 +
      t2*coef2 + t3*coef3
```



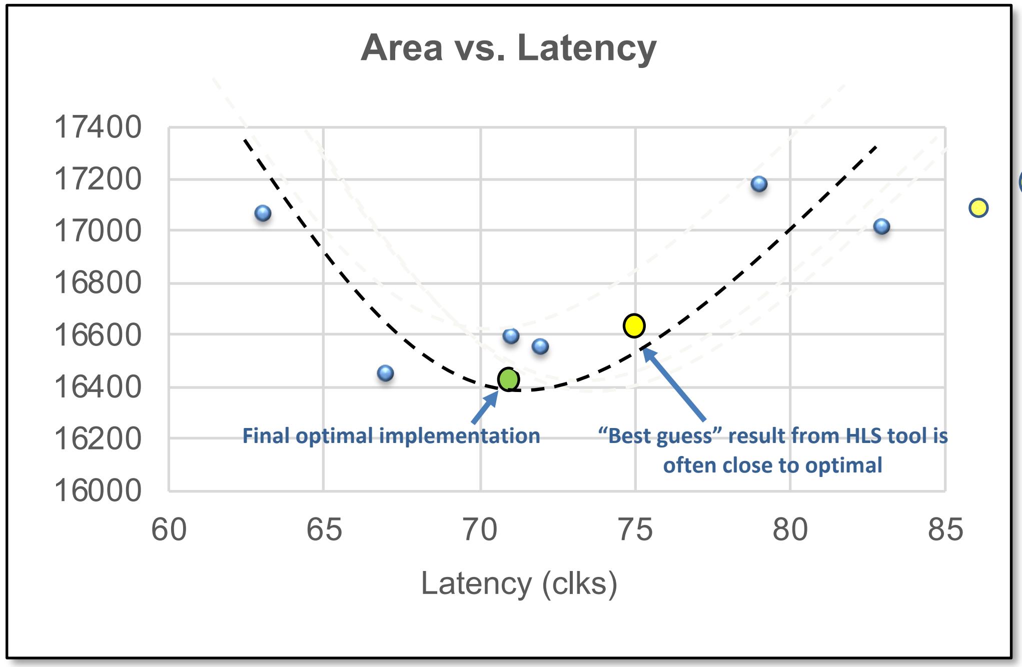
NOTE: Similar multiply and accumulate trees are common in AI/NN domain

Pipelined Variants

1 stage		Cannot meet timing
2 stage		Tight timing in 1st stage
3 stage		Good for timing, but wasted sequentials?
2 stage		Well balanced pipeline

Exploration: Pipelining 2

- **Case Study:** Signal Processing, 5G MMSE (datapath)



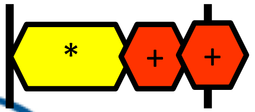
Change latency directive, 30 minute HLS run per RTL variant

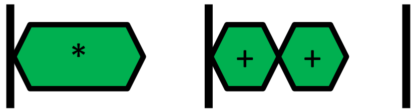
Cell upsizing or replication to reduce delays

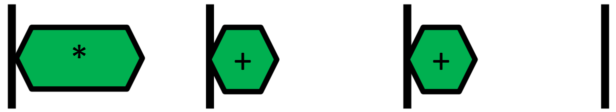
Can't meet timing

More guardband for timing delays

Wasted sequentials

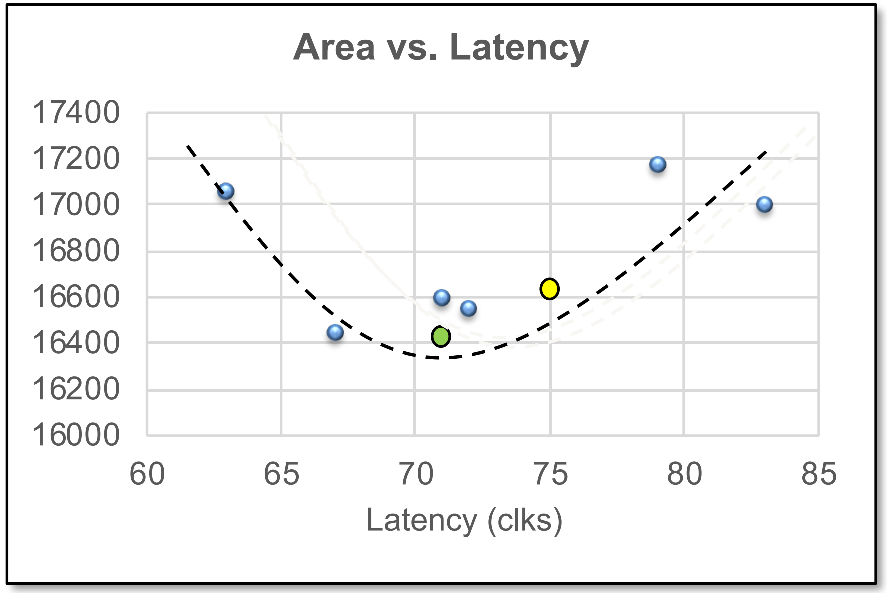






Exploration: Pipelining 3

- Existing RTL vs. HLS Results
  - Area:
    - 21.9k -> 16.4k (**34%** savings)
    - Multipliers shared efficiently by HLS
  - Latency:
    - 102 -> 71 clocks (**29%** savings)
    - Existing RTL developed on previous tech node
  - Throughput:
    - 1 per 4 clocks -> 1 per clock (**4X**)
    - HLS gives options:
      - 4X the throughput for 2X the area
      - Same throughput at 1/4th the clock freq (power)



Pipelining II vs. Frequency

uarch	Area	Latency
PIPE_II4	16490	71
PIPE_II1	31707	55
PIPE_II1_192MHz	18847	12

Exploration: Max Frequency

- **Case Study:** High-Frequency Memory Fabric (control logic)
  - Maximum frequency of the fabric primarily limited by arbitration algorithms
- Challenges with traditional RTL design:
  - Timing analysis done after synthesis of entire fabric (long turnaround)
  - Timing path analysis through arbitration and fabric modules (high effort)
- With HLS design:
  - Tool does “characterization” logic synthesis for a region of code performing arbitration
  - Early information; delay for entire arbitration operation is grouped

Exploration: Max Frequency 2

Procedural Arbitration Code

```
unsigned winner;    // Which requestor is picked
req_ingress_arb: {
  if (req_wr_vec && !pri_rd)
    winner = arb_wr[rspid].arbitrate(req_wr_vec);
  else if (req_rd_vec)
    winner = arb_rd[rspid].arbitrate(req_rd_vec);
  . . . etc . . .
}
```

Multiple HLS runs

Vary number of requestors

fabric_**4**x3_req_ingress_arb
Area = 0.47 Delay = 139.1 ps

fabric_**5**x3_req_ingress_arb
Area = 0.59 Delay = 164.2 ps

fabric_**6**x3_req_ingress_arb
Area = 0.70 Delay = 189.0 ps

Meets timing @ 2GHz

Marginal @ 2GHz

Cannot meet @ 2GHz

Fast feedback gives time to consider options

Change arb algo?

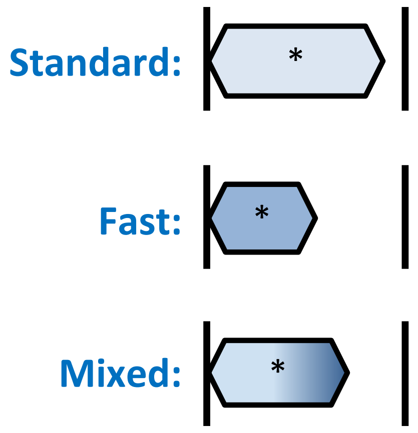
Remove pri chan?

Multi-level fabric?

Exploration: Static Power

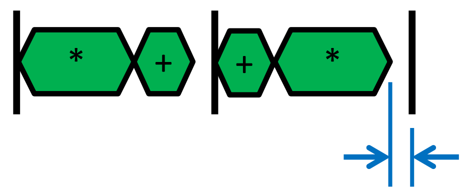
- Static (leakage) power is managed through Multi-Vt cells
  - SVT: Standard Vt - "Slow" - Standard delay, Low leakage
  - LVT: Low Vt - "Fast" - Shorter delay, High leakage

- Goals differ depending on design:
  - Mobile: Prioritize low leakage (battery life, always active blocks)
    - Design team may target a split like 90/10 for SVT/LVT cells
  - Server: Tolerant of higher leakage (power-down idle silicon)
    - Design team may target a split with majority LVT cells (and some ultra-fast ULVT)



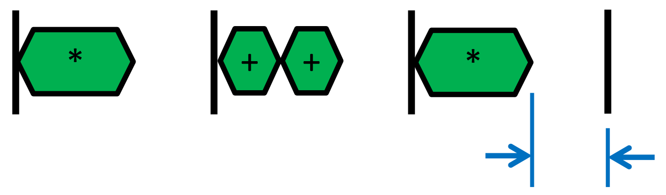
Exploration: Static Power 2

- Targeting a split like 90/10 for SVT/LVT cells:
  - "Slow" (average) cell delays, very low leakage for 90% of cells
  - Small portion of fast, high-leakage cells to resolve tight timing issues
  - In HLS tool terms:
    - "Fit as much logic as possible in each stage, reserve a very small timing guardband"



Small guardband for routing/unpredicted delays

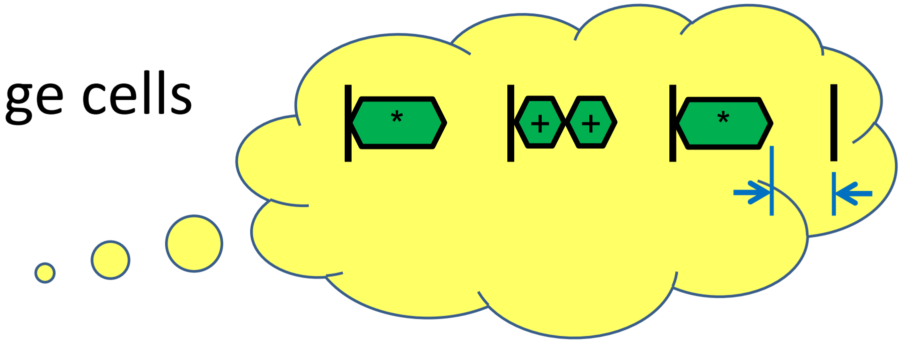
- What if low leakage is the primary goal?
  - Trade-off is latency for power
  - Slower cells require more time/stages to compute

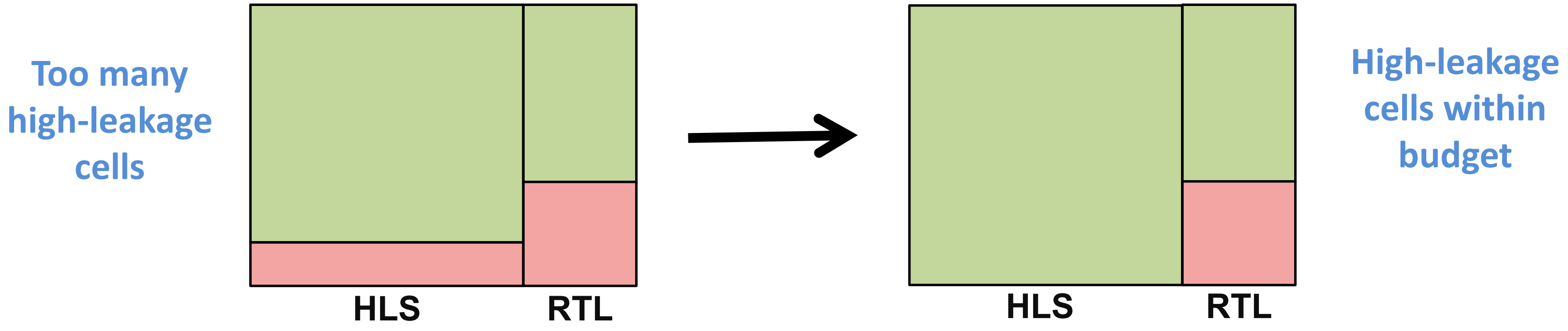


Large guardband for routing/unpredicted delays, no need for fastest cells

Exploration: Static Power 3

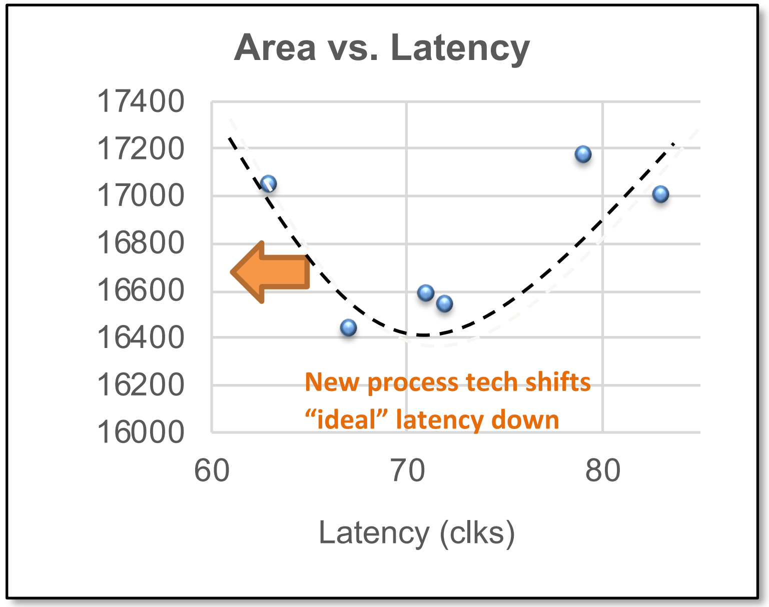
- **Case study:** 5G Transceiver (mostly datapath)
  - Always on (or quick wake-up); power-down is not an option
  - Must have low leakage
- Target was 90% of the total design using low-leakage cells
  - Legacy RTL and cores were using 40% high-leakage cells
  - Re-pipeline all blocks (20+) with large timing guardband
    - **1 day's effort**





Efficient Reuse with HLS

- Next-gen technology
  - Change from XXnm to Ynm tech node
  - Re-generate RTL that is timing-clean and has well-balanced (power efficient) pipelines on the new tech node
- HLS Parameterization
  - Same filter code can be used with different latency, throughput, parallelism directives
  - Re-use same SystemC IP across product spectrum:
    - Server: Fastest cells, high frequency/parallelism/throughput
    - Mobile: Low-leakage cells, latency/throughput trade-offs



Wrap Up

- HLS finds low power and efficient solutions
  - Design space exploration
  - Late “tuning” changes
  - Efficient designs when moving to next gen silicon tech
- SystemC-based designs
  - Less code, less effort to make design changes
  - Simulates fast – quick debug iterations
  - Portions of code usable in other domains (VP, arch/perf model)
- The Investment
  - Ramping teams into new language and toolset required real investment

Rangharajan Venkatesan, NVIDIA Corp.

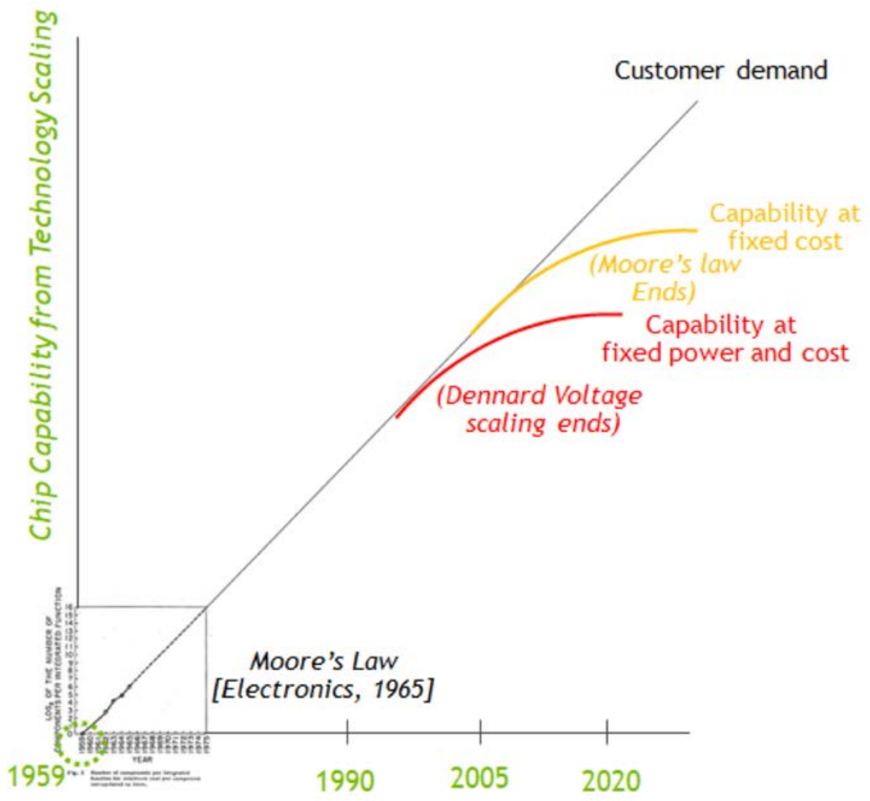
MATCHLIB-BASED OBJECT-ORIENTED HLS METHODOLOGY

Agenda

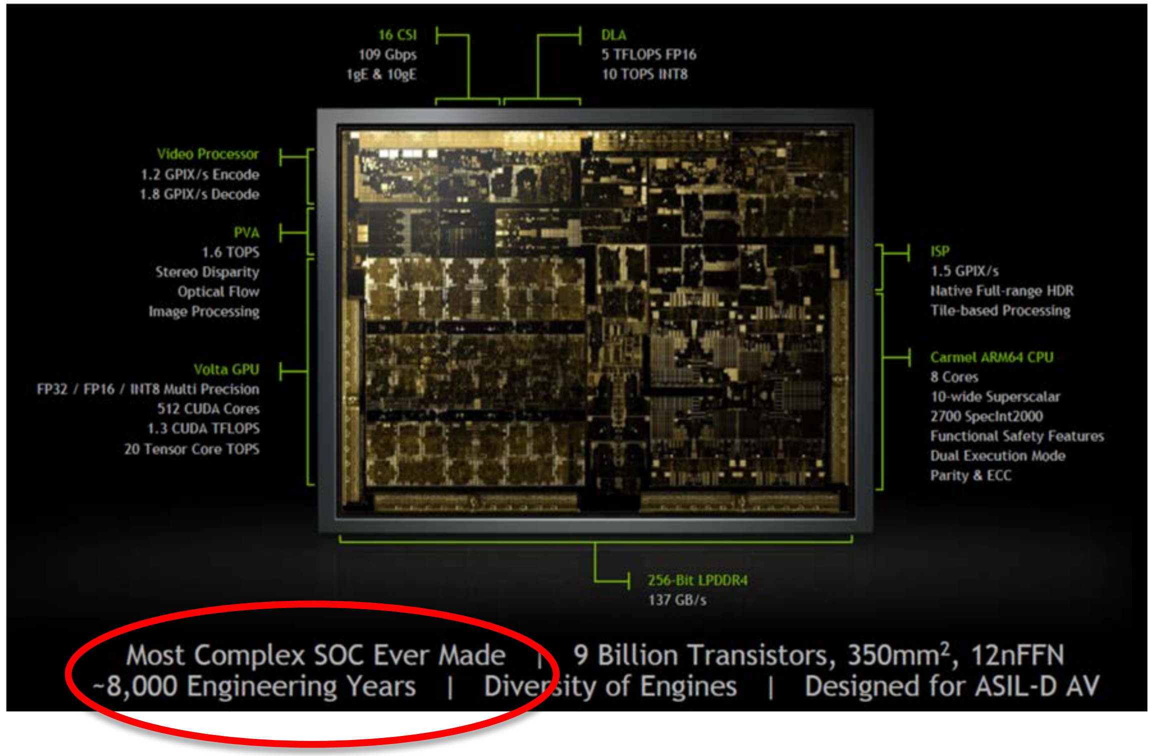
- **Motivation - Chip Design Complexity**
- **Object-oriented HLS Approach**
  - A “C++/SystemC-to-Layout” Flow
  - MatchLib Opensource HLS library
  - Latency-Insensitive Channels: Connections
  - Verification Methodology
- **SoC TestChip Demonstrations**
- **RC18: Scalable Deep Learning Accelerator**
  - 128 TOPS, ~10 TOPS/W
  - Spec-to-Tapeout in 6 months with <10 researchers

Chip Design Complexity

Customer demand for more transistors & capability, need lower design costs

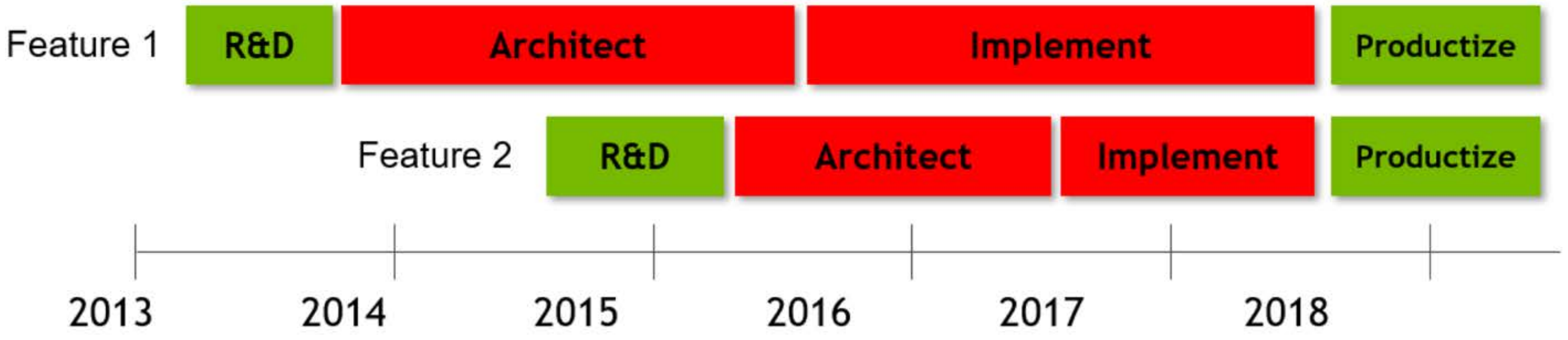


NVIDIA Xavier SoC [CES 2018]



Motivation: Shorten Development Time & Cost

Typical development timeline: 3-5 years from R&D to product



- RTL design and verification dominates: >70% of IC design effort at NVIDIA
  - Prohibits which features make it into each SoC
- Need 10x lower design and verification effort:
  - Overlap architect & implement phases for faster time-to-market, more features

High-productivity Design Approach

Enables faster time-to-market and more features to each SoC

RAISE HARDWARE DESIGN LEVEL OF ABSTRACTION

Use High-level languages

e.g. C++/SystemC instead of Verilog

Use Automation

e.g. High-Level Synthesis (HLS)

Use libraries/generators

MatchLib

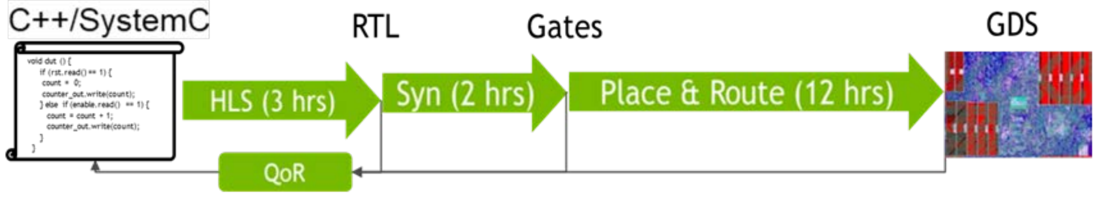
AGILE VLSI DESIGN

Small teams, jointly working on architecture, implementation, VLSI

Continuous integration with automated tool flows

Agile project management techniques

24-hour spins from SystemC-to-layout

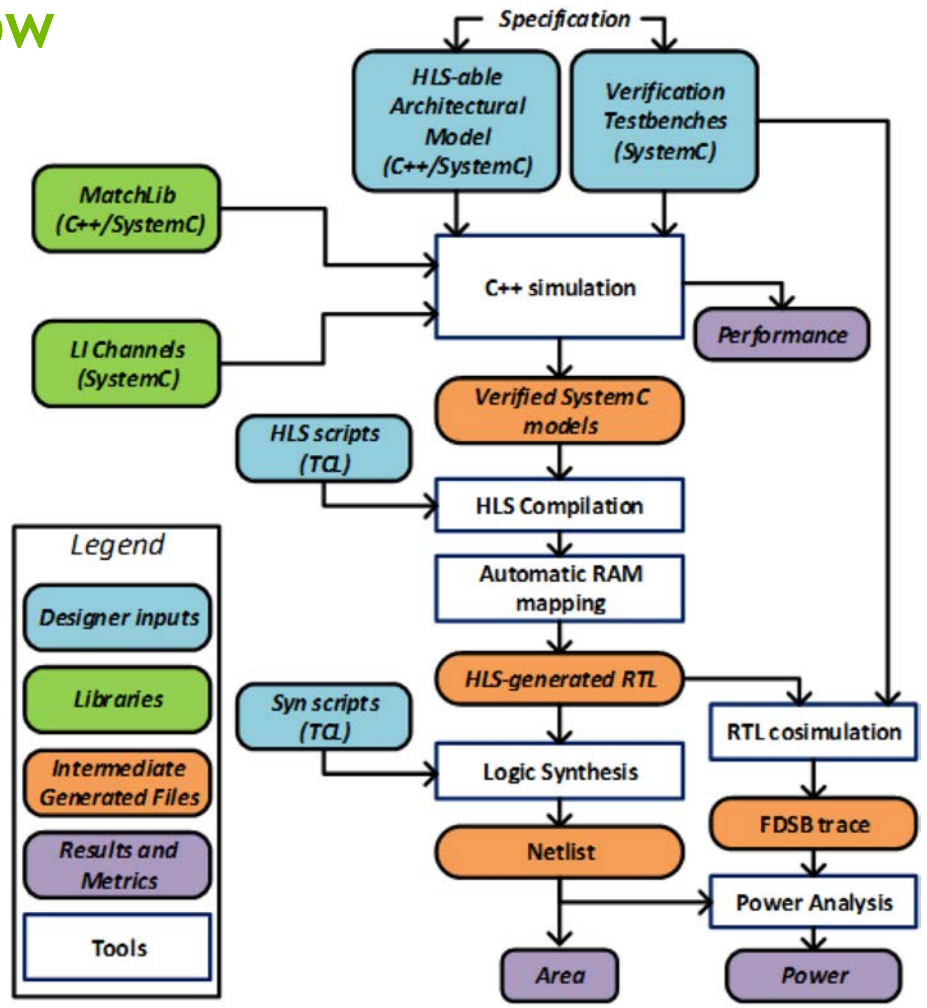


```
void dut () {
    if (rst.read() == 1) {
     count = 0;
     counter_out.write(count);
    } else if (enable.read() == 1) {
      count = count + 1;
      counter_out.write(count);
    }
  }
```

Object-Oriented High-Level Synthesis (OOHLS)

"Push-button" SystemC-to-gates flow

- Leverage HLS tools to design with C++ and SystemC models
- MatchLib: **M**odular **A**pproach **T**o **C**ircuits and **H**ardware **Lib**rary
  - "STL/Boost" for Hardware Design
  - Synthesizable hardware library developed by NVIDIA research
  - Highly-parameterized, high QoR implementation
  - Available open-source: https://github.com/NVlabs/matchlib
- Latency-Insensitive (LI) Channels
  - Enable modularity in design process
  - Decouple computation & communication architectures

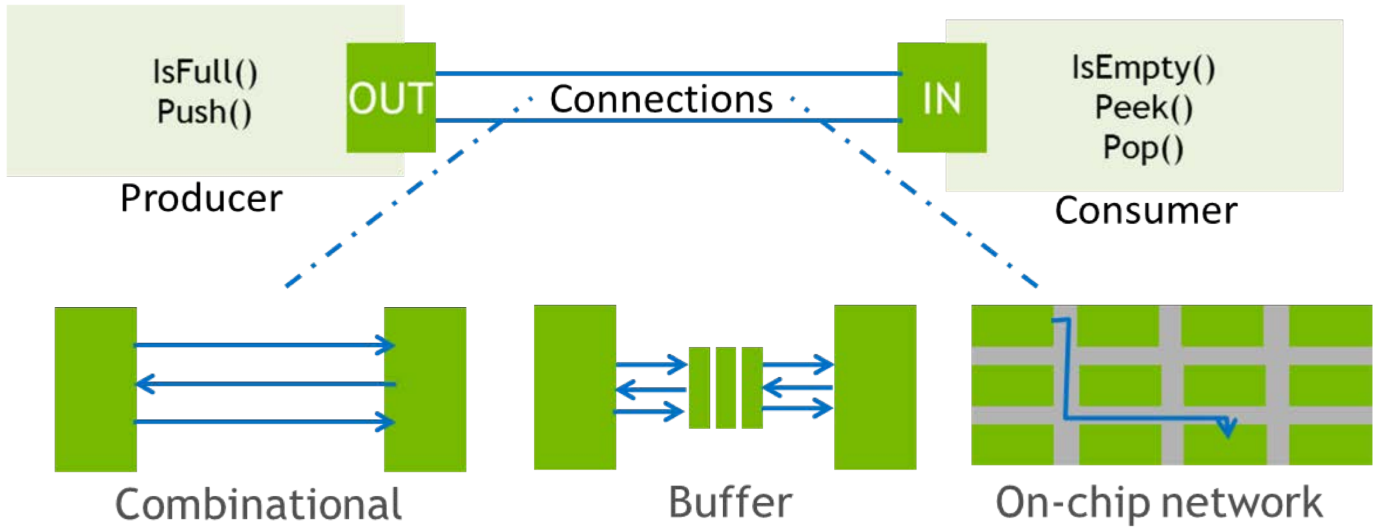


Ref: *Khailany et al., DAC 2018*

Latency-Insensitive (LI) Channels

Our approach: Connections

- Connections is a LI channel library in MatchLib



- Producer & consumer implementations are channel-agnostic, with arbitrary pipelining and scheduling
- Channels mapped to ready/valid interfaces when synthesized to RTL

Ref: L.P.Carloni et al., DAC'99, K.E.Fleming et al., FPGA 2012

Object-Oriented High-Level Synthesis (OOHLS)

Verification Methodology

All functional verification and performance verification run natively in C++ simulation of SystemC models

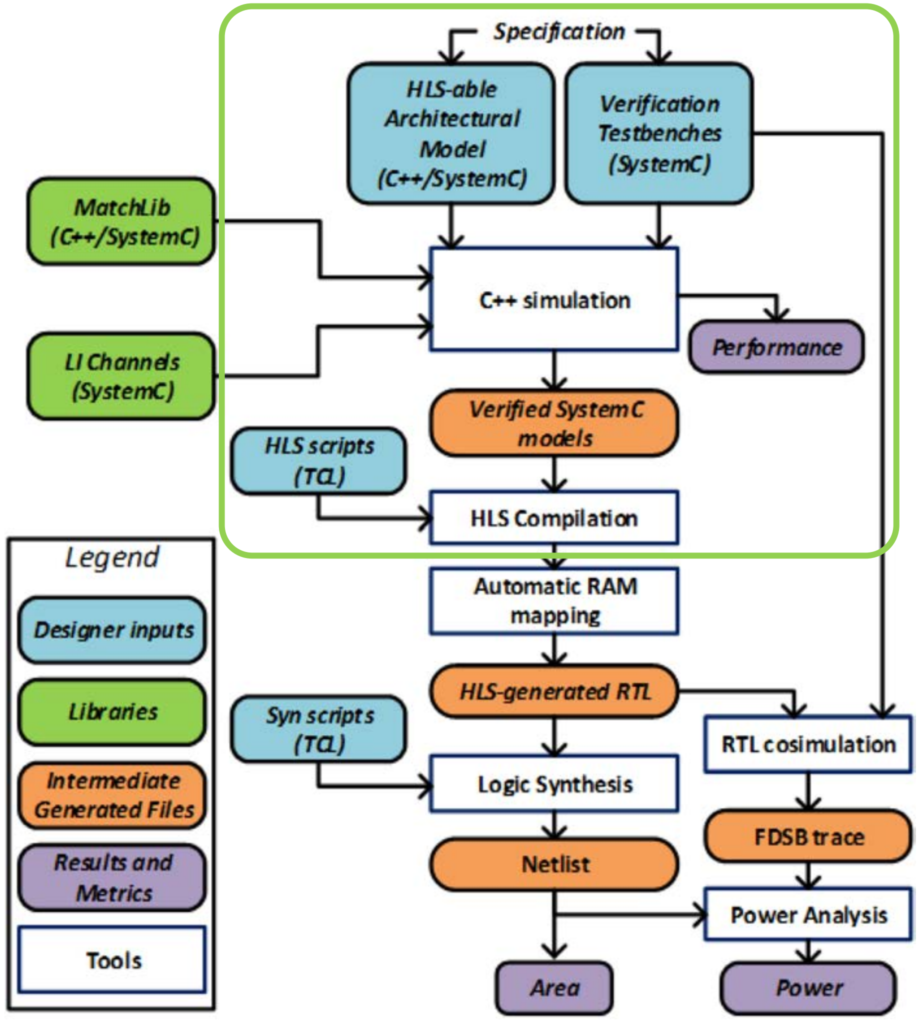
- ~50x speedup over RTL
- ~3% error in performance

Leverage libraries for productivity

Use std verification approaches but in SystemC

- Constrained random generators and checkers
- Line and expression coverage
- Assertions

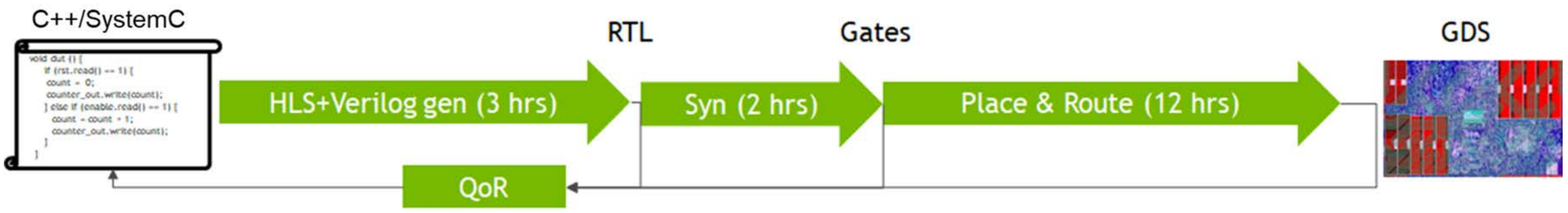
- All SystemC testbenches reusable with generated RTL for final verification at full-chip and unit-level
- No Formal verification Support between SystemC and RTL
  - A major challenge for widespread adoption of HLS in industry



Ref: Khailany et al., DAC 2018

Agile VLSI design Techniques

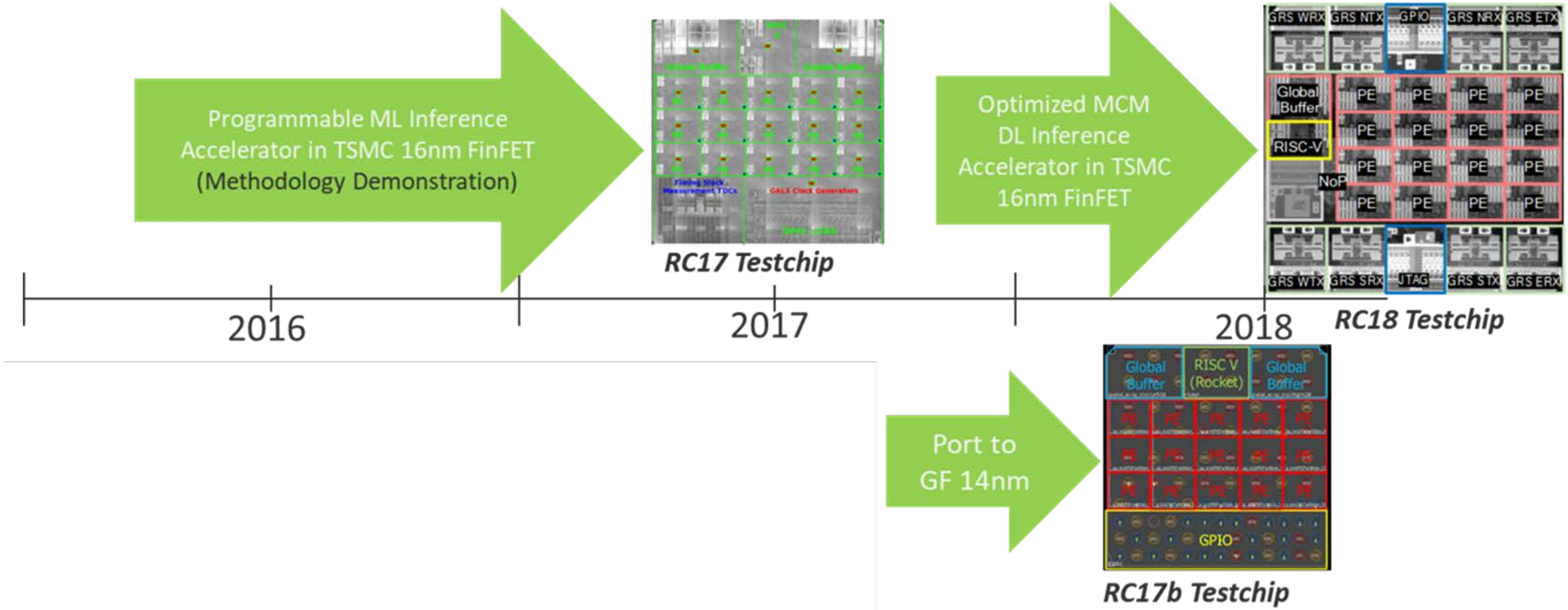
Daily "SystemC to Layout" Spins



```
void dut () {
   if (rst.read() == 1) {
    count = 0;
    counter_out.write(count);
   } else if (enable.read() == 1) {
     count = count + 1;
     counter_out.write(count);
   }
 }
```

- Agile, incremental approach to design closure during march-to-tapeout phase
- Small, abutting partitions for fast place and route iterations
- RTL bugs, performance, and VLSI constraints converge together

SoC Testchip Demonstrations



RC18: SCALABLE DEEP LEARNING INFERENCE ACCELERATOR

RC18: Scalable DL Inference Accelerator

Project Goals

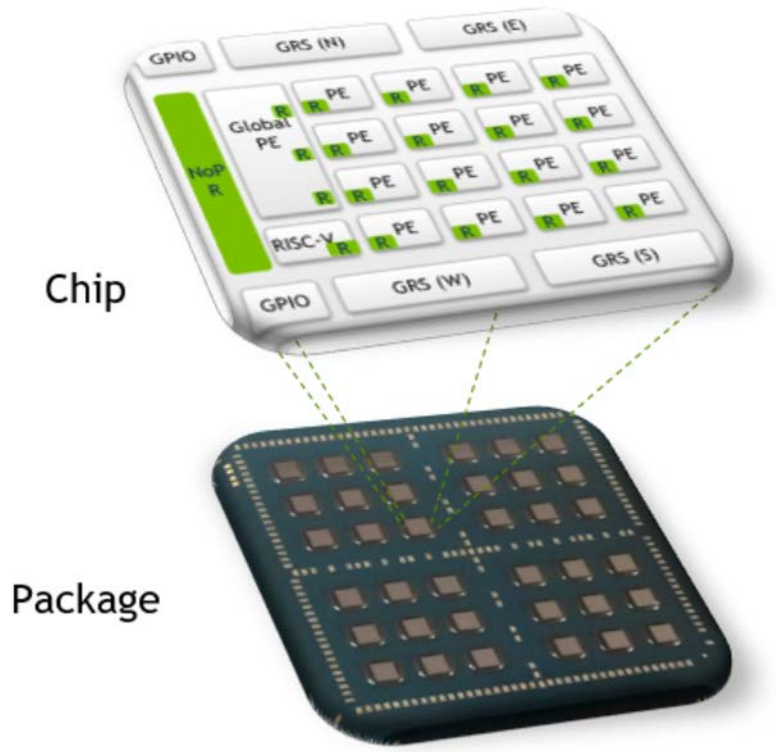
- Design a state-of-the-art scalable high-performance deep learning inference accelerator
  - >100 TOPS, ~10 TOPS/W

- Demonstrate this is possible with a small team in a high-productivity VLSI flow

RC18: Scalable DL Inference Accelerator

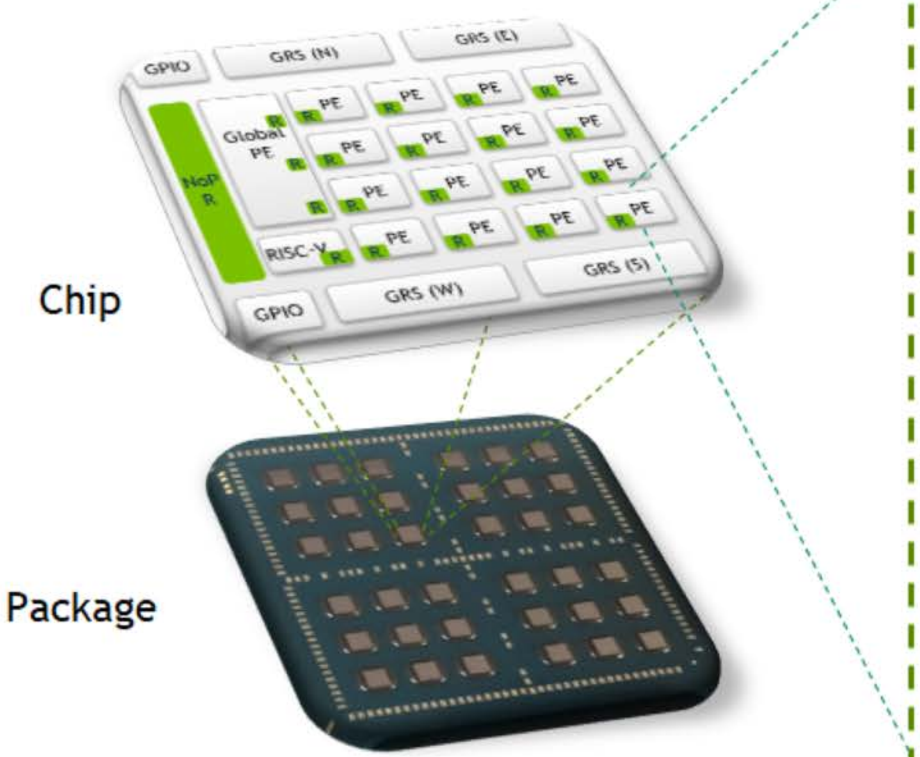
Tiled Architecture with Distributed Memory

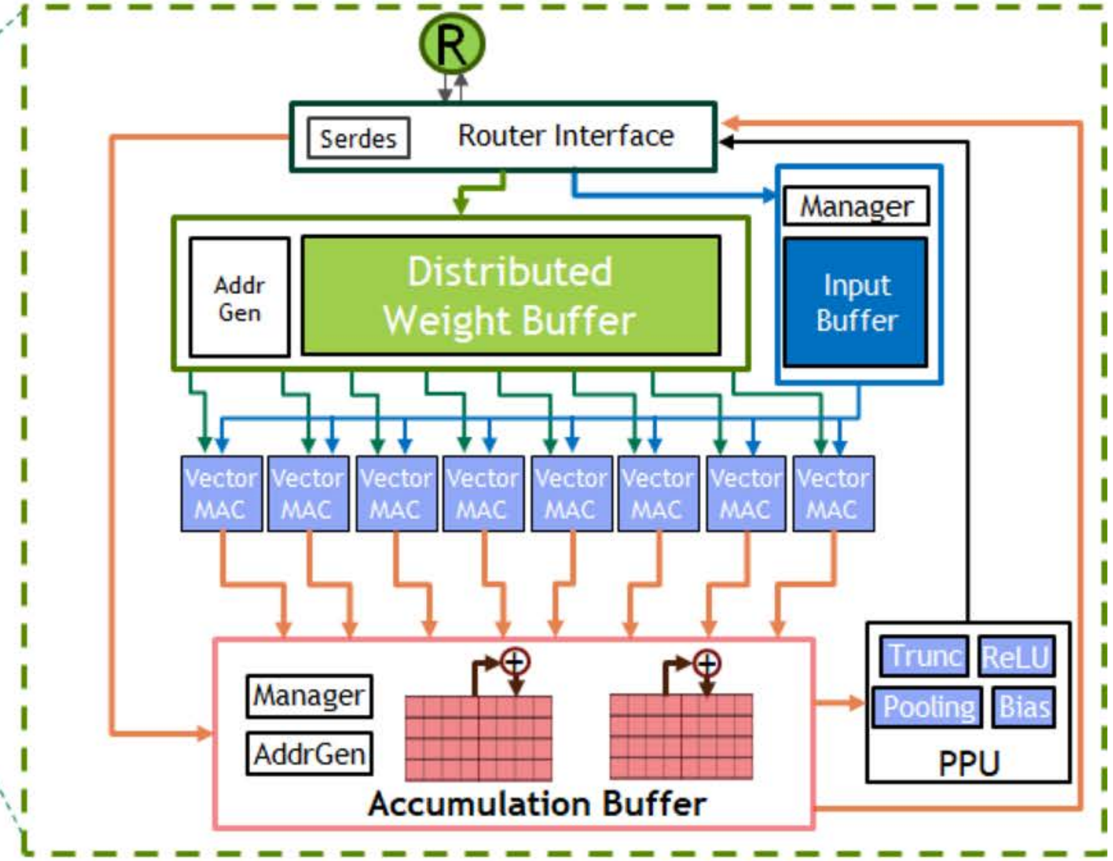
- Package
  - 6x6 Multi-Chip Modules (MCM) in a package
  - Ground-reference signaling based inter-chip communication
- Chip
  - 4x4 array of Processing Elements (PEs) per chip
  - Global Buffer/PE for 2nd-level inter-layer storage
  - RISC-V controller
  - 5x4 Mesh Network-on-Chip
  - Network-on-Package and GPIO Interface



RC18: Scalable DL Inference Accelerator

Processing Element



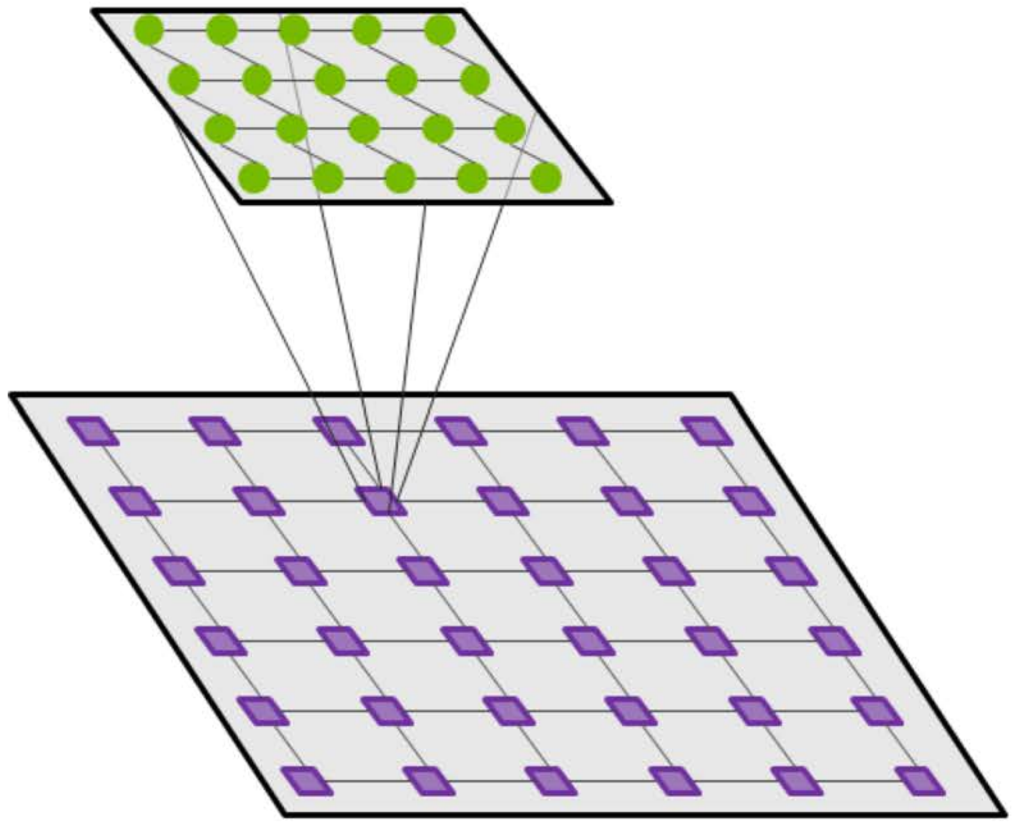


Zimmer et al., VLSI 2019,
Venkatesan et al. HotChips 2019

Ref: Sijstermans et al., HotChips 2018

Hierarchical Communication Architecture

Network-on-Package (NoP) and Network-on-Chip (NoC)



NETWORK-ON-CHIP (NoC)

4x5 mesh topology connects 16 PEs, one Global PE, and one RISC-V

Cut-through routing with Multicast support

10ns per hop, ~70Gbps per link (at 0.72V)

NETWORK-ON-PACKAGE (NoP)

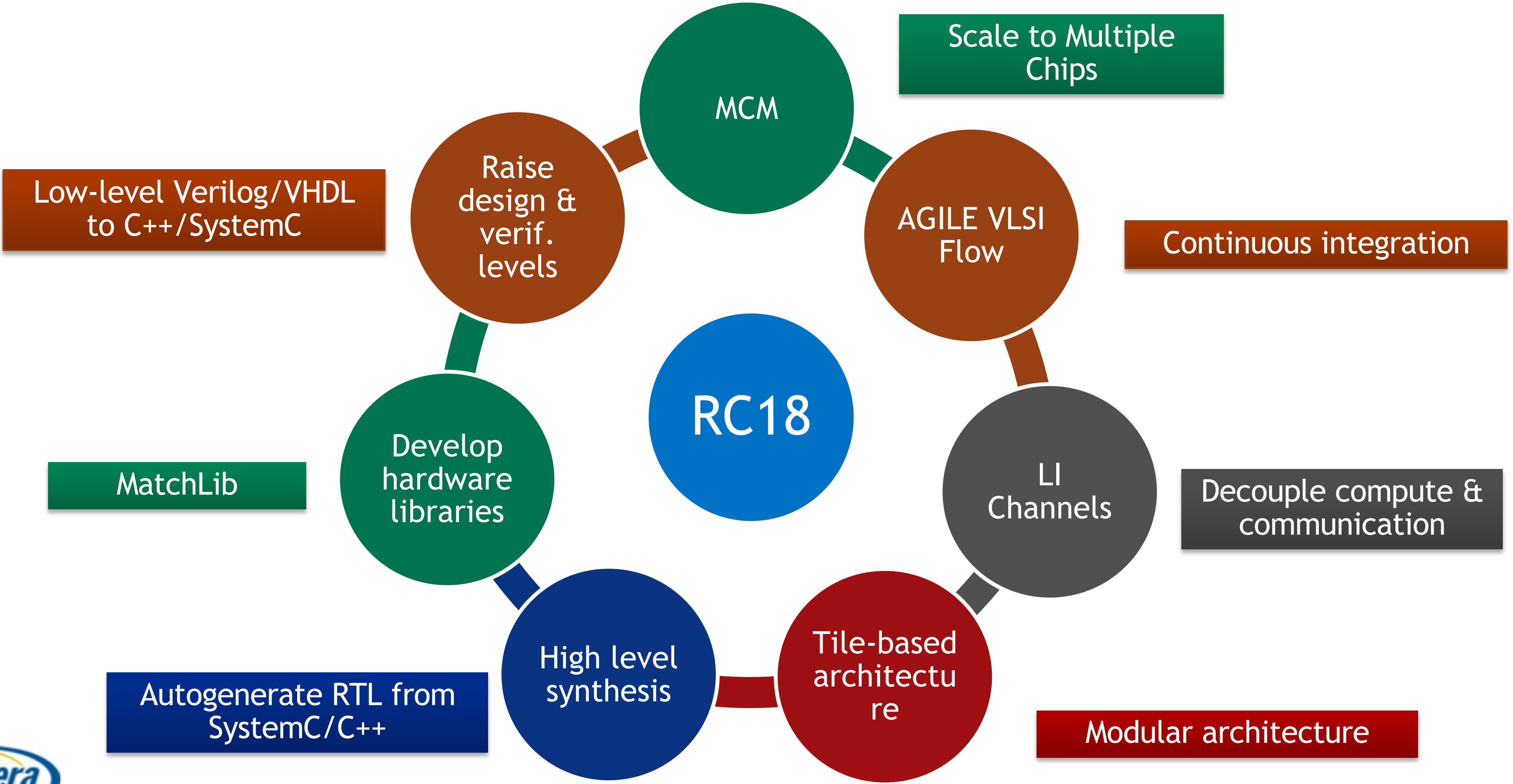
6x6 mesh topology connects 36 chips in package.

A single NoP router per chip with 4 interface ports to NoC

Configurable routing to avoid bad links/chip

~20ns per hop, 100 Gbps per link (at max)

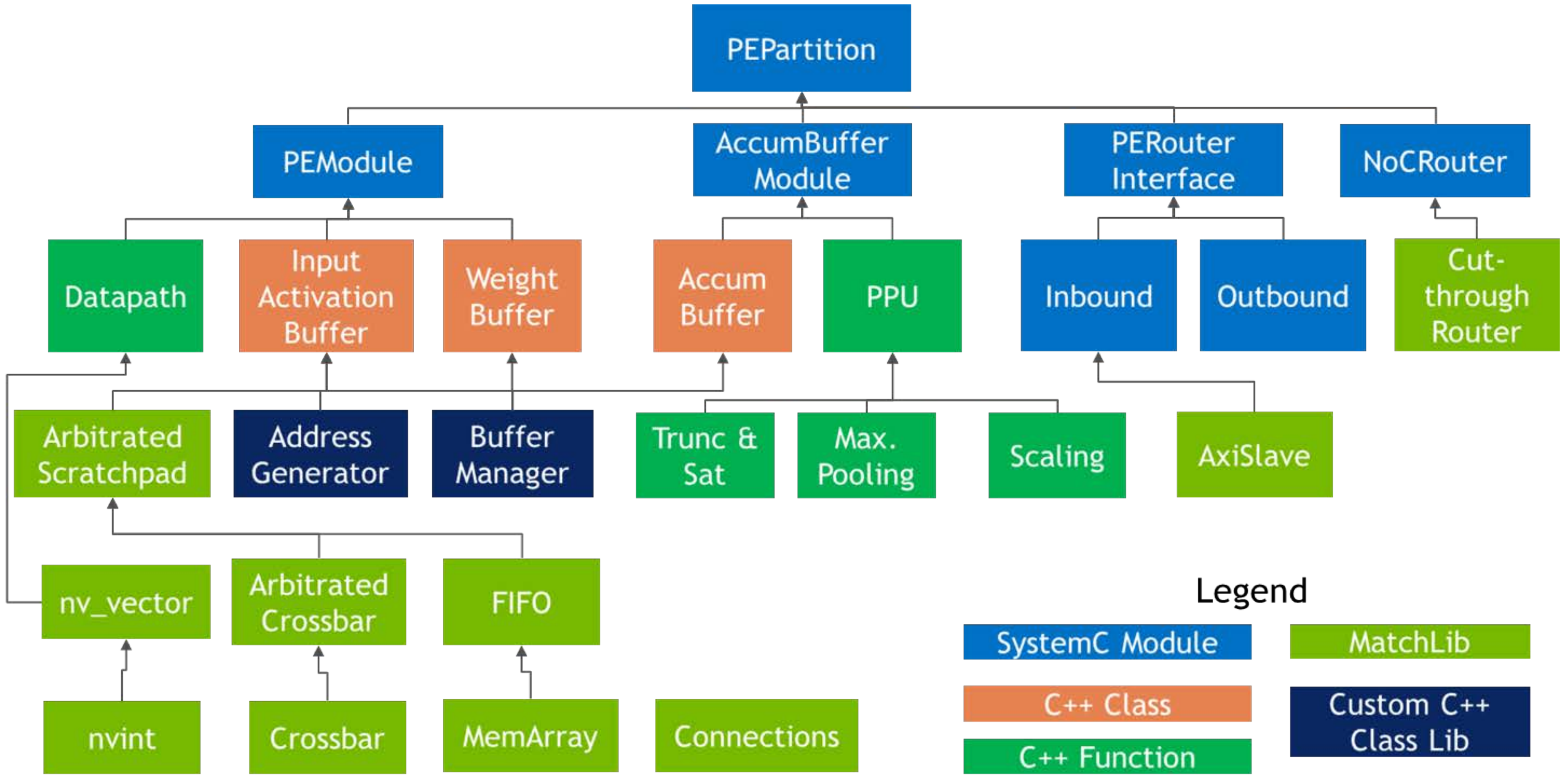
RC18 Design Methodology



Scale to Multiple Chips
MCM
Raise design & verif. levels
Low-level Verilog/VHDL to C++/SystemC
AGILE VLSI Flow
Continuous integration
RC18
Develop hardware libraries
MatchLib
LI Channels
Decouple compute & communication
High level synthesis
Autogenerate RTL from SystemC/C++
Tile-based architectu re
Modular architecture

Processing Element Implementation

Reuse, Modularity, Hierarchical Design



Fabricated MCM-based Accelerator

NVResearch Prototype: 36 Chips on Package in TSMC 16nm Technology

High speed interconnects using Ground Reference Signaling (GRS)

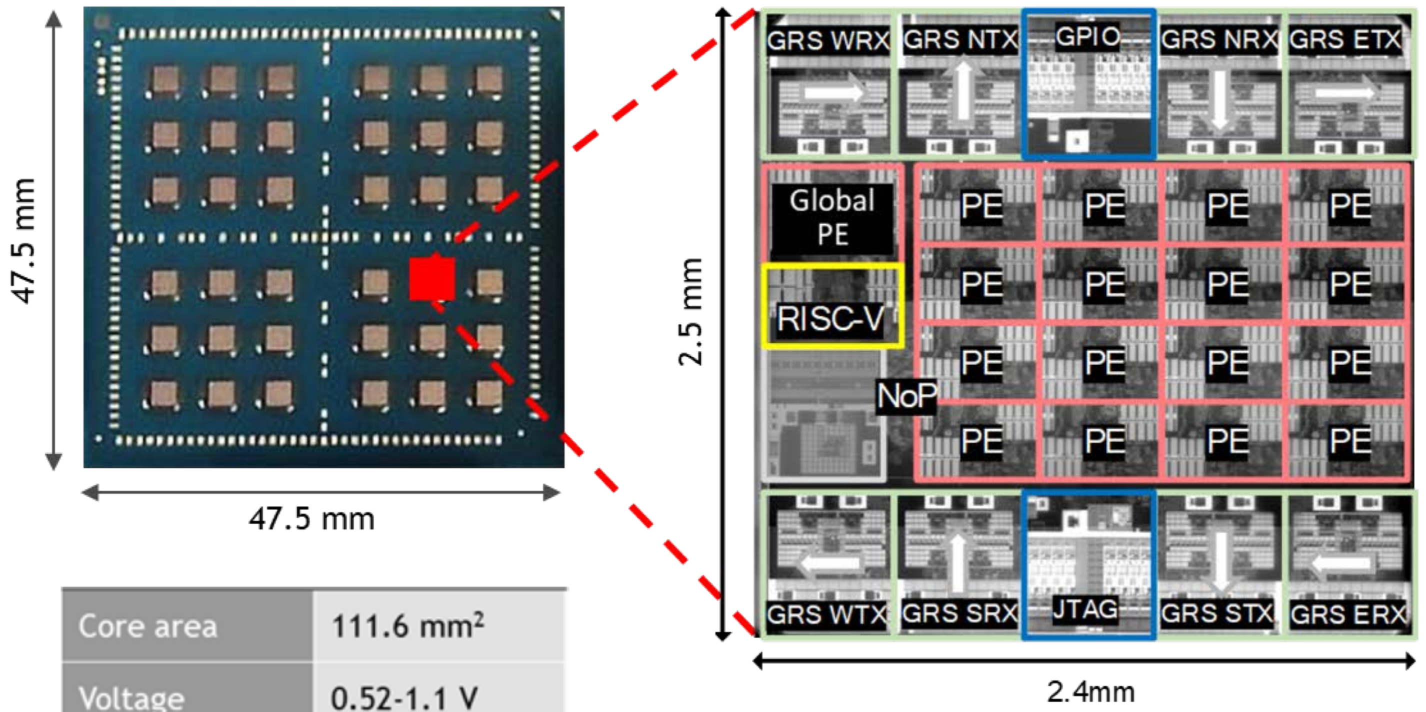
- 100 Gbps per link

Efficient Compute tiles

- ~10 TOPS/W, 128 TOPS

Low Design Effort

- Spec-to-Tapeout in 6 months with <10 researchers



Core area	111.6 mm²
Voltage	0.52-1.1 V
Frequency	0.48-1.8 GHz

Summary

MATCHLib-Based Object-Oriented HLS Methodology

- Design effort reduction from raising abstraction levels, tools
- Object-Oriented HLS Flow
  - Latency Insensitive Channels abstractions for all communication
  - MatchLib for capturing reuse of commonly-used hardware components
- Enables faster time-to-market and more features to each SoC
  - ~10X reduction in ASIC design and verification efforts

RC18: Scalable DL Inference Accelerator

Acknowledgments

Collaborative Effort Across Architecture and Design Methodology

- Research sponsored by DARPA under the CRAFT program (PM: Linton Salmon).
- NVIDIA Collaborators: Evgeni Krimer, Jason Clemons, Ben Keller, Matthew Fojtik, Alicia Klinefelter, Angshuman Parashar, Michael Pellauer, Nathaniel Pinckney, Mark Ren, Yakun Sophia Shao, Stephen Tell, Yanqing Zhang, Brian Zimmer, Bill Dally, Joel S. Emer, Tom Gray, Stephen Keckler.
- Harvard Collaborators: David Brooks, Gu-Yeon Wei, and team
- Former NVIDIA interns/postdocs: Christopher Fletcher, Ziyun Li, Antonio Puglielli, Shreesha Srinath, Gopal Srinivasan, Chris Torng, Sam (Likun) Xi
- Thanks to the Mentor Graphics Catapult HLS team for discussions and support: Bryan Bowyer, Stuart Clubb, Moises Garcia, Khalid Islam, and Stuart Swan.

THANK YOU!