accellera SYSTEMS INITIATIVE™

Cut Your Design Time in Half with Higher Abstraction

Speakers: Mike Meredith – Cadence Design Systems, Inc.
Peter Frey – Mentor Graphics Corp.

Agenda

- Introduction
  - How High-Level Synthesis(HLS) works targeted for hardware designers
- Accellera SystemC Synthesizable Subset
- High-Level Synthesis Verification
- HLS in the Wild
  - Intel Experience

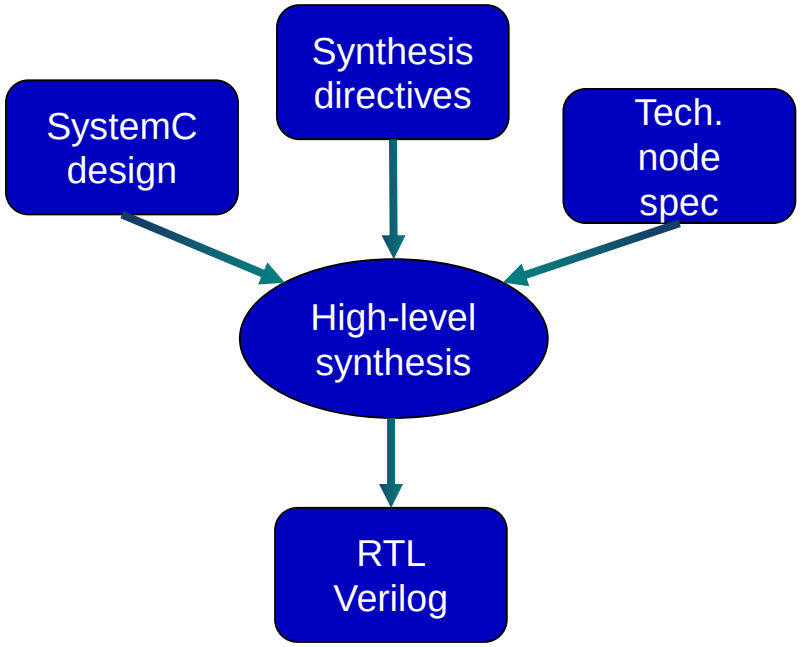
How High-level Synthesis Works:
An Intro for hardware designers

Frederic Doucet

Qualcomm Atheros, Inc

High-level Synthesis

- HLS tool transforms synthesizable SystemC code into RTL Verilog
  1. Precisely characterizes delay / area of all operations in a design
  2. Schedules all the operation over the available clock cycles
  3. Can optionally increase latency (clock cycles) to get positive slack and increase resource sharing (reduces area)
  4. Generate RTL that is equivalent to input SystemC
     - Pipe depths / latencies decided by HLS scheduler



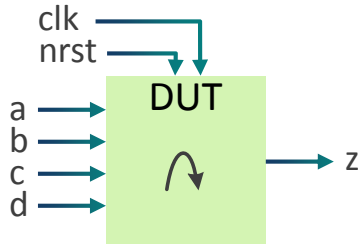
High-level Synthesis

- SystemC HLS has been used in many large semiconductors companies for years, on both control/datapath heavy designs

- Main SystemC HLS usage:
  - Encode and verify all high-level control-flow and datapath functions in SystemC
  - Use HLS tool automatically generate all pipelines and decide latencies resulting in RTL is optimized for specified clk period / tech node.

SystemC: Hardware Model in C++

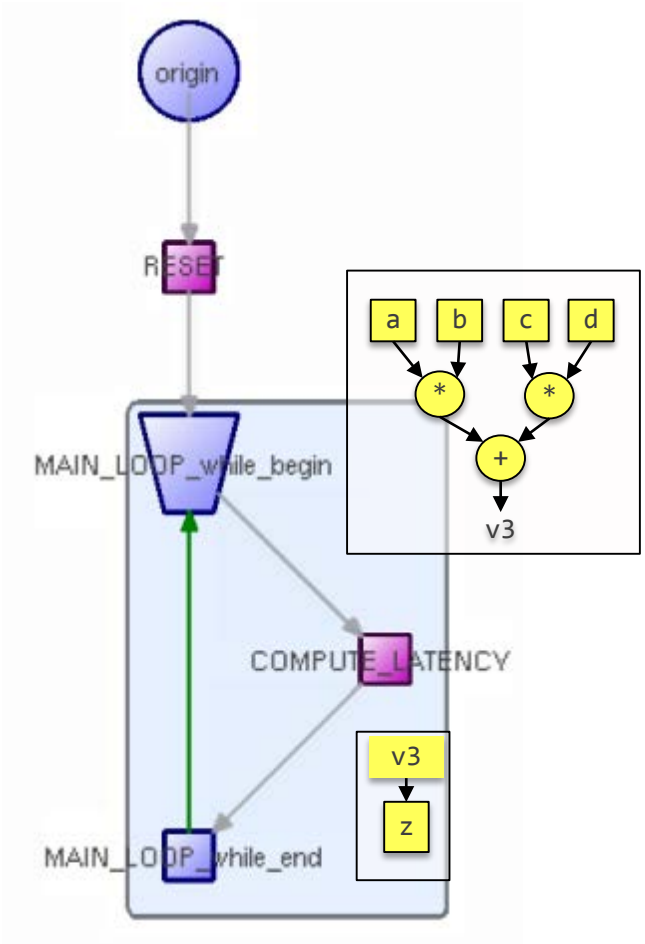
- SystemC: syntax for hardware modeling framework in C++
  - Modules
  - Ports
  - Connections
  - Processes



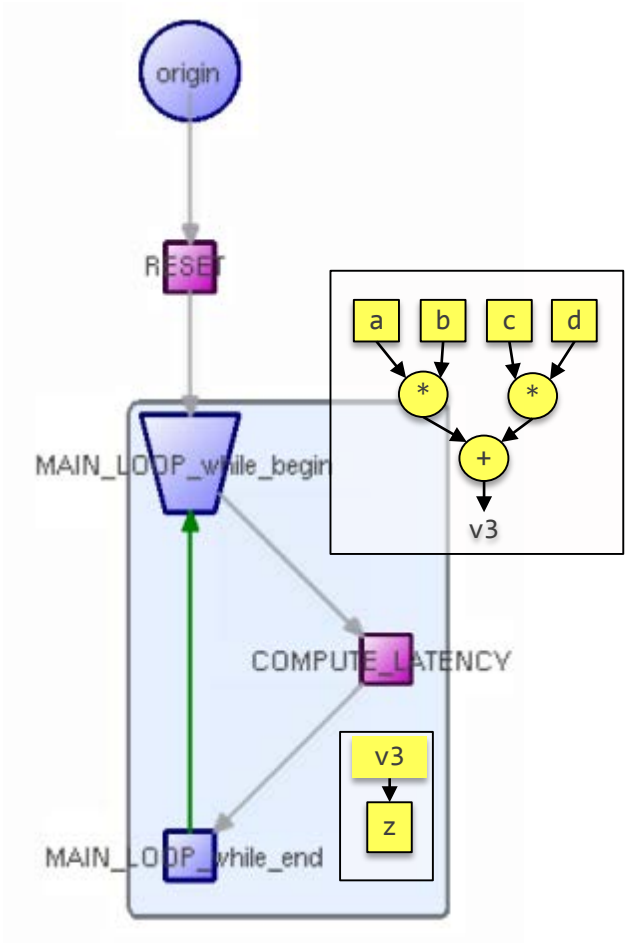
- Inside a process is C++ code describing the functionality
  - DSP processing
  - Control logic
  - Etc.

Example : Synthesizable SystemC

```
SC_MODULE(DUT)
{
  sc_in <bool> clk;
  sc_in <bool> nrst;
  sc_in <int>  a;
  sc_in <int>  b;
  sc_in <int>  c;
  sc_in <int>  d;
  sc_out<int>  z;
  ...
  void process() {
    z = 0;
    RESET:
    wait();

    MAIN_LOOP:
    while (true) {
      int v1 = a * b;
      int v2 = c * d;
      int v3 = v1 + v2;

      COMPUTE_LATENCY:
      wait();

      z = v3;
    }
  }
};
```



Example: High-level synthesis



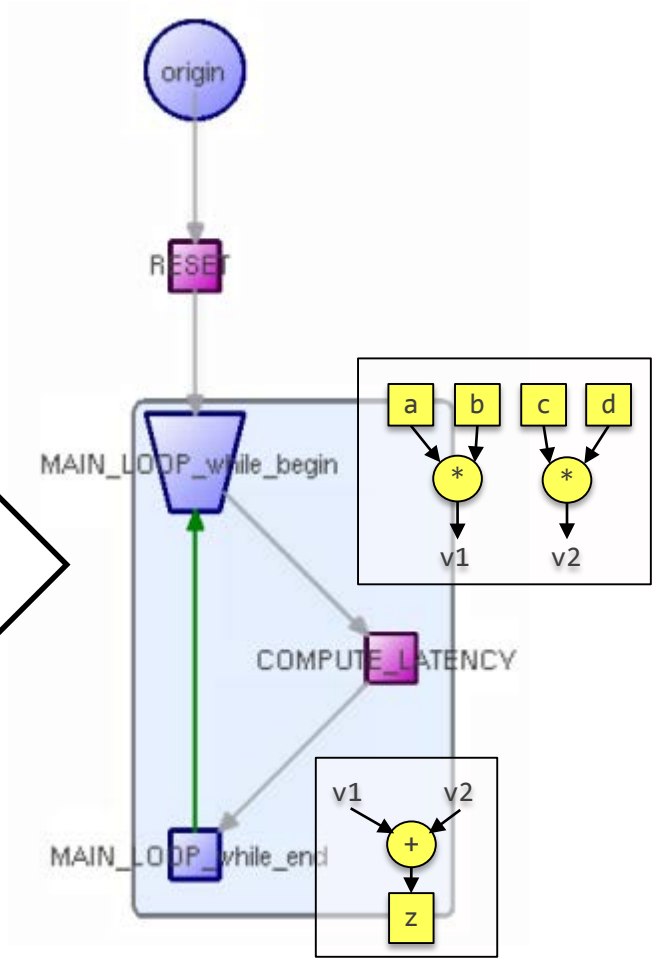
Synthesis directives:

- clk period: 5ns
- tech node: 65lp
- *no micro-arch directive*

Scheduling/resource allocation/binding

Op delays:

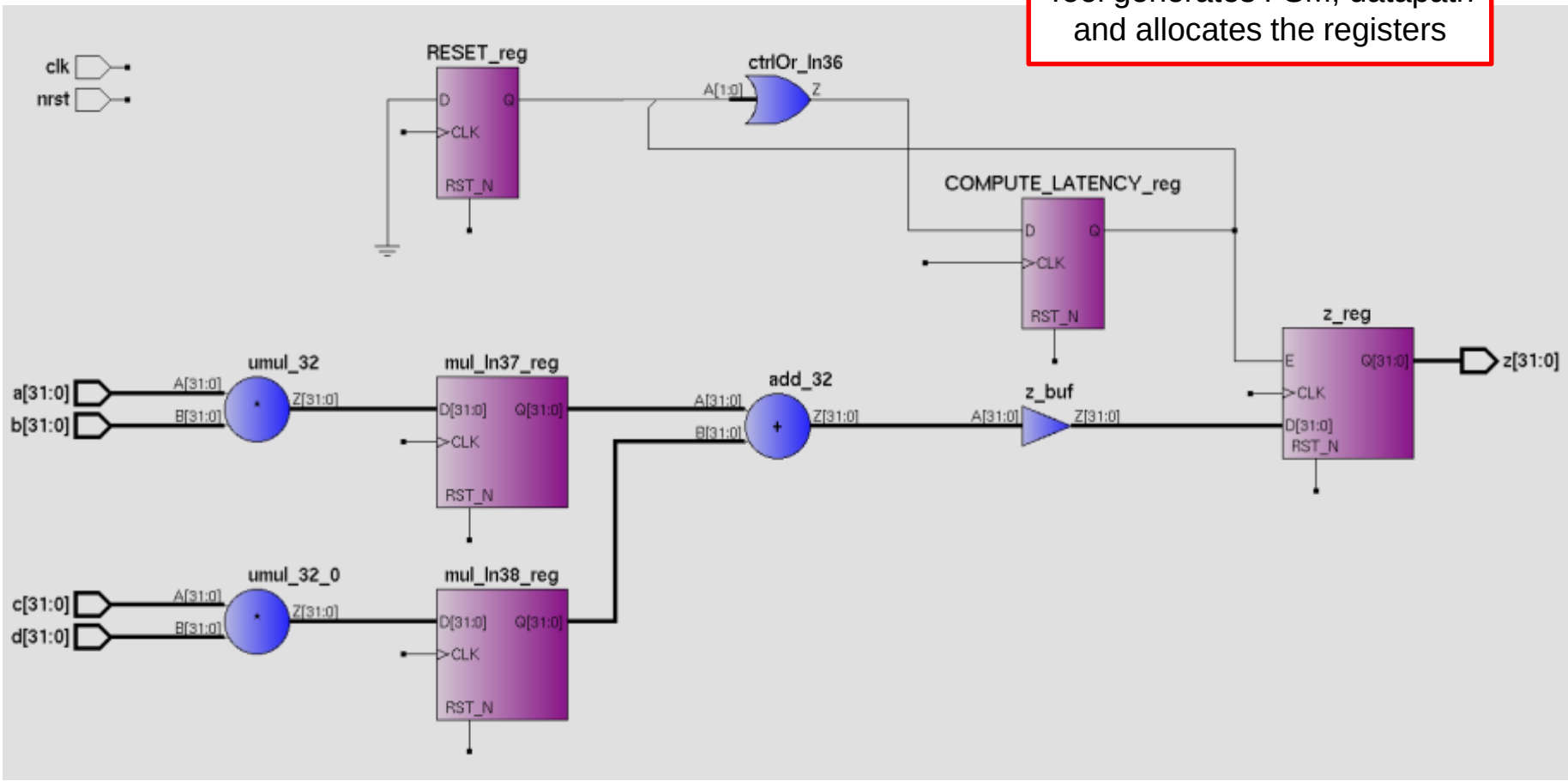
- mul: 4ns
- add: 2ns



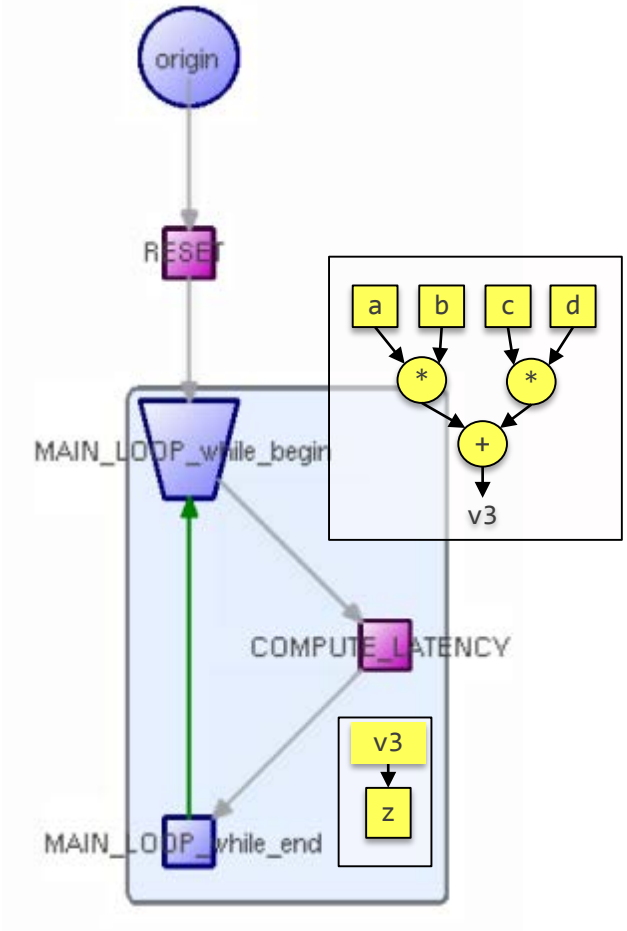
Scheduler moved the addition across the state to get positive slack

Example: High-level synthesis

Tool generates FSM, datapath and allocates the registers



Example: High-level synthesis, second run



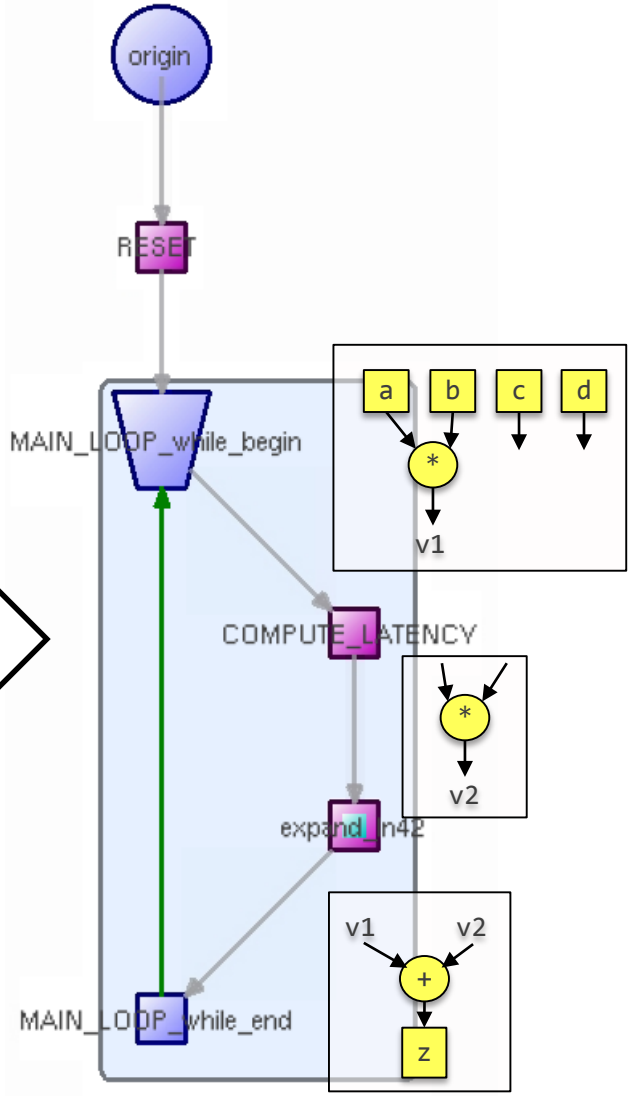
Synthesis directives:

- clk period: 5ns
- tech node: 65lp
- *minimize resources*

Scheduling/resource allocation/binding

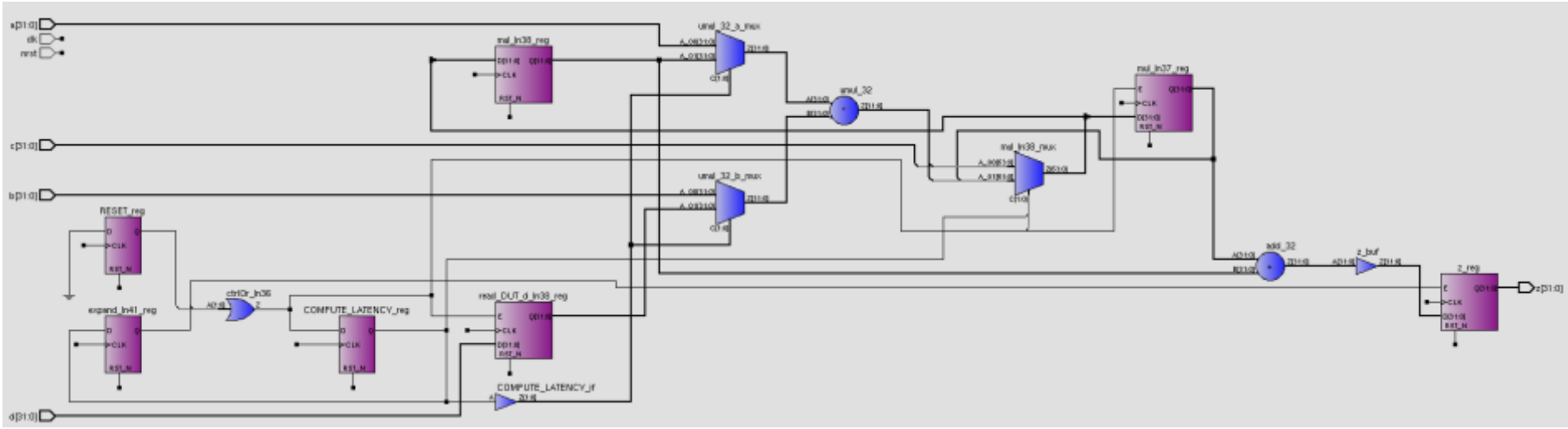
Op delays:

- mul: 4ns
- add: 2ns



Scheduler added a state to share the multiplier

Example: High-level synthesis, second run



- Notice that there is only one multiplier
- Sharing mux/registers are automatically allocated and bound to the generated FSM

HLS and Abstraction

- The tool automatically generates the micro-architecture details
  - latencies, muxes, registers, FSMs

  → *this is what can be abstracted out in the SystemC code*

- Starting from SystemC code, HLS tool does:
  1. Map arithmetic/logical operations to resources
  2. Allocate resources and try to share them as much as possible
  3. Automatically generate FSM and sharing logic
  4. Allocate registers and try to share them as much as possible
  5. Optionally add clock cycles to get positive slack and maximize sharing
  6. Generate RTL

SystemC to Describe Hardware

- Input SystemC code still needs to capture hardware architecture
  - what is the high-level control, data flow and I/O protocols
  - what are the necessary concurrent processes
  - which are the abstract datapath functions for the tool to refine

→ *Best done by hardware designer*

- Fast turnaround is a big benefit
  - Small changes in the SystemC / synthesis directives can quickly generate new RTL with new and very different micro-architecture
  - Impossible to do with RTL design

SystemC Language

- Designers can use many of the nice C++ features to help write the code
  - Structs/classes, templates, arrays/pointers, functions, fixed/complex classes, etc.
  - Coding patterns/guidelines to separate signal processing code from I/O, etc.

- A standard interpretation of SystemC will help energize the SystemC HLS marketplace and accelerate adoption

The Accellera SystemC Synthesizable Subset

Mike Meredith

Vice Chair – Accellera Synthesis Working Group

Cadence Design Systems

SystemC Synthesizable Subset Work

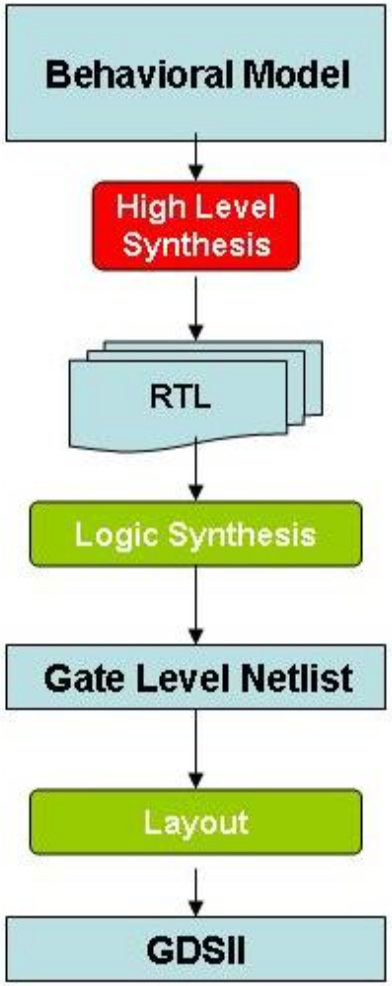
- Development of a description of a synthesizable subset of SystemC
- Started in the OSCI Synthesis Working Group
- Current work is in Accellera Systems Initiative Synthesis Working Group
- Standard approved 3/11/2016
- Many contributors over a number of years
- Broadcom, Cadence, Calypto, Forte, Fujitsu, Freescale, Global Unichip, Intel, ITRI, Mentor, NEC, NXP, Offis, Qualcomm, Sanyo, Synopsys

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

Scope of The Standard

- Synthesizable SystemC
- Defined within IEEE 1666-2011
- Covers behavioral model in SystemC for synthesis
  - SC_MODULE, SC_CTHREAD, SC_THREAD
- Covers RTL model in SystemC for synthesis
  - SC_MODULE, SC_METHOD
- Main emphasis of the document is on behavioral model synthesizable subset for high-level synthesis



Scope Of The Standard

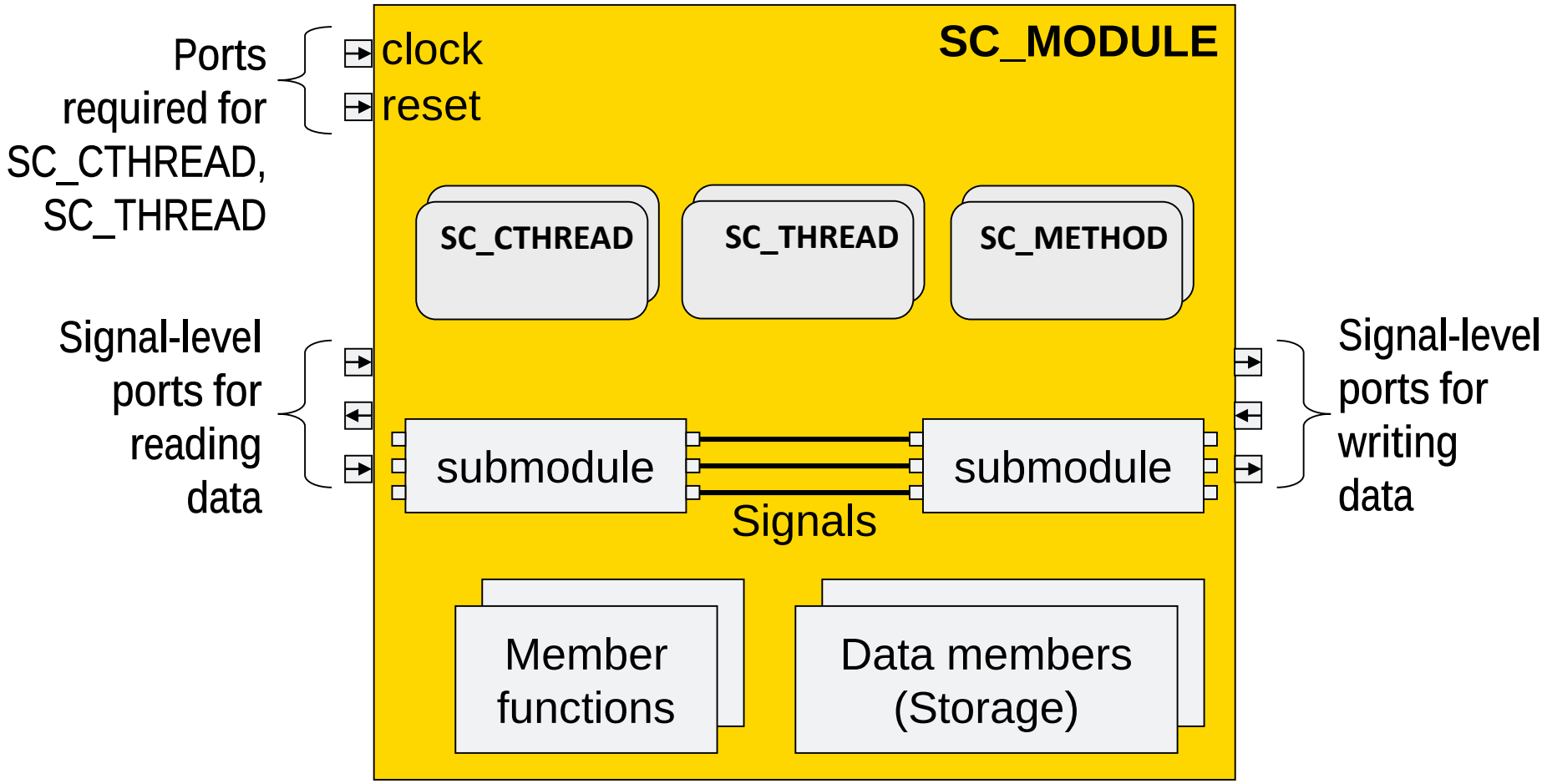
SystemC Elements

- 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



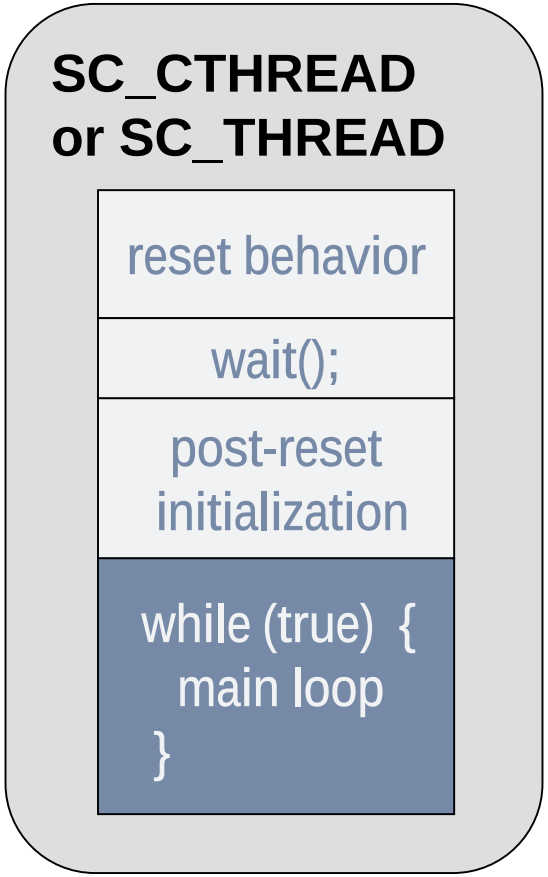
Module Declaration

- Module definition
  - SC_MODULE macro
    or
  - Derived from sc_module
    - class or struct
  - 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)
 {…}
};
```

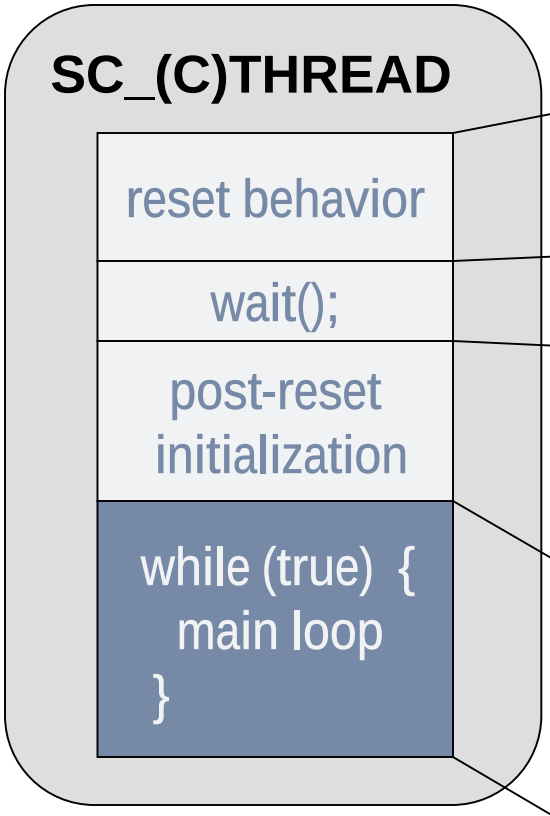
SC_THREAD & SC_CTHREAD Reset Semantics

- At start_of_simulation each SC_THREAD and SC_CTHREAD function is called
  - It runs until it hits a wait()
- When an SC_THREAD or SC_CTHREAD is restarted after any wait()
  - If reset condition is false
    - execution continues
  - If reset condition is true
    - stack is torn down and function is called again from the beginning
- This means
  - Everything before the first wait will be executed while reset is asserted



Note that every path through main loop must contain a wait() or simulation hangs with an infinite loop

SC_THREAD & SC_CTHREAD Process Structure




```
void process() {
 // reset behavior must be
 // executable in a single cycle
 reset_behavior();

 wait();

 // initialization may contain
 // any number of wait()s.
 // This part is only executed
 // once after a reset.
 initialization();

 // infinite loop
 while (true) {
   rest_of_behavior();
 }
}
```

Specifying Clock and Reset

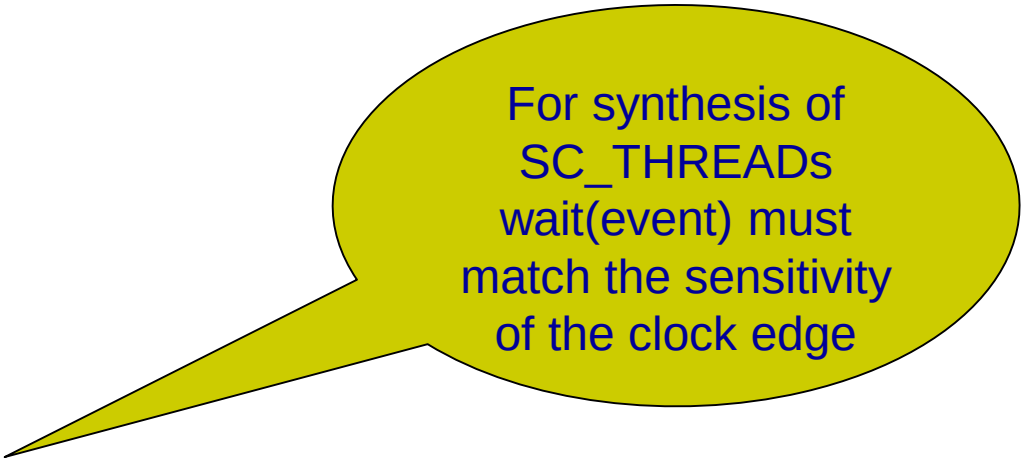
For synthesis, SC_THREAD can only have a single sensitivity to a clock edge

```
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 )
```

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())



Types and Operators

- C++ types
- sc_int, sc_uint
- sc_bv, sc_lv
- sc_bigint, sc_biguint
- sc_logic
- sc_fixed, sc_ufixed

- All SystemC arithmetic, bitwise, and comparison operators supported
- Note that shift operand should be unsigned to allow minimization of hardware

Data Types

- C++ integral types
  - All C++ integral types except wchar_t
  - char is signed (undefined in C++)
- C++ operators
  - a>>b
    Sign bit shifted in if a is signed
  - ++ and -- not supported for bool
- For sc_lv
  - “X” is not supported
  - “Z” is not supported

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

Other C++ Constructs

- Supported
  - Templates
  - const
  - volatile
  - namespace
  - enum
  - class and struct
    - private, protected, public
  - Arrays
  - Overloaded operators
- Not supported
  - sizeof()
  - new()
    - Except for instantiating modules
  - delete()
  - typeid()
  - extern
  - asm
  - Non-const global variables
  - Non-const static data members
  - unions

What to standardize next…

- Benefit of current standard:
  - Provides clear guidelines for synthesizability for C++/SystemC
  - Set clear subset for synthesis tools
- We are currently discussing the options for the next standard
- A big list of topics…
  - What is important to us designers?
  - What is valuable to EDA vendors?
  - What are the priorities?
  - Did we think of everything?

Join the discussion!
Join the SWG calls!

Current standardization discussions

- C++ / C++11
  - Unions
  - Constructor arguments
  - Automatic port naming VCD tracing for all ports for all ports
  - Safe array class
  - Type handling advances (auto, decl)
  - Many other features of interest …
- Datatype enhancements
  - Fixed bit-width and fixed point type enhancements
  - sc_complex
  - sc_float
- Channel libraries
- Microarchitecture directives

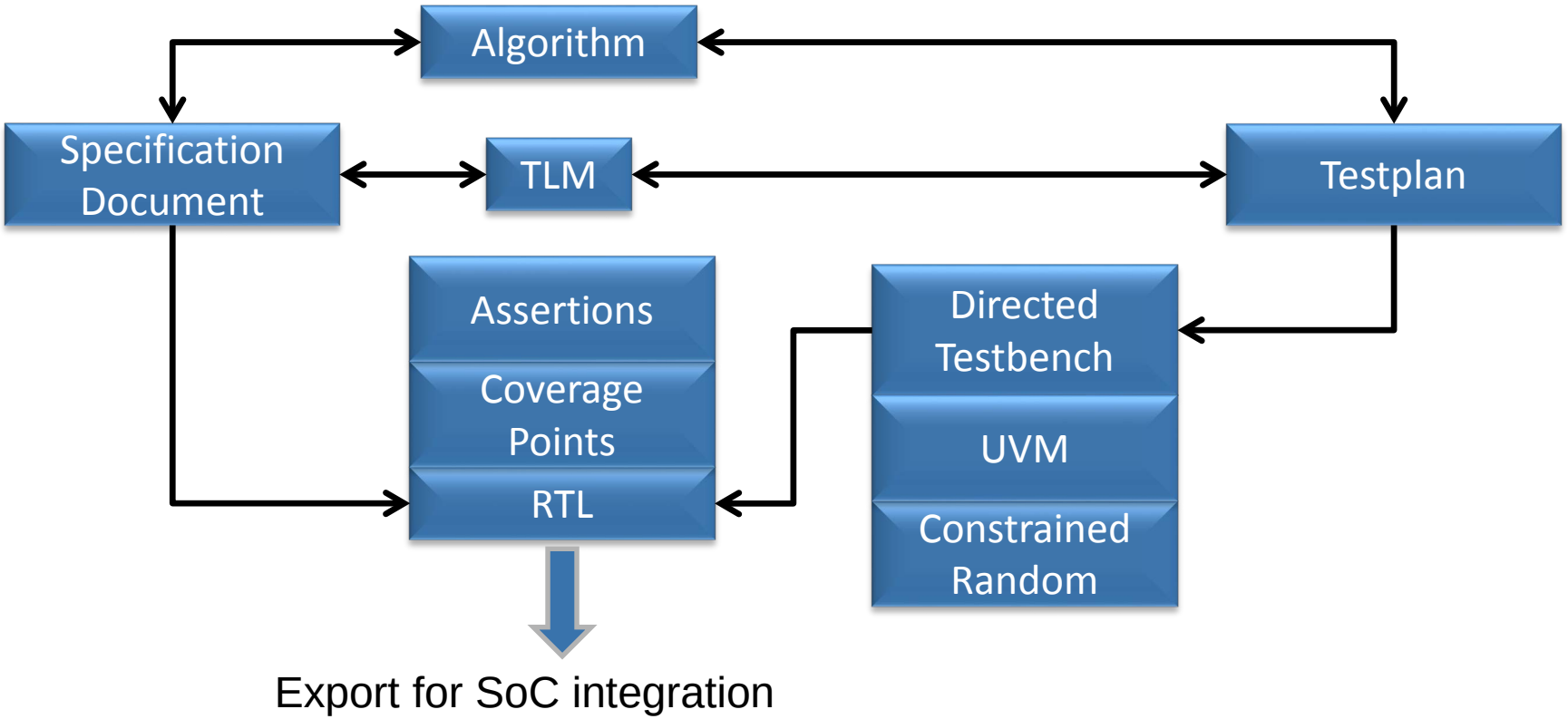
High-Level Synthesis Verification

Peter Frey, HLS Technologist

Problem Statement

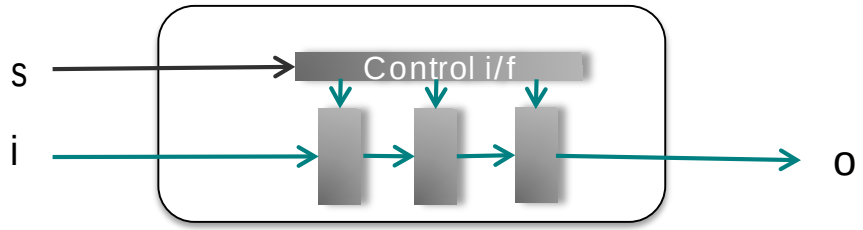
- Designing your RTL is hard
  - Complex architectures
  - Specifications open to interpretation
  - Many constraints (Power, Linting, DFT, Synthesis)
- Fully debugging your RTL is impossible
  - Massive vector sets for HW and SW
  - Massive integrated SoCs
  - Design cycles under pressure
- Each Year
  - Major advances in verification technology, but…
  - The problems still get worse

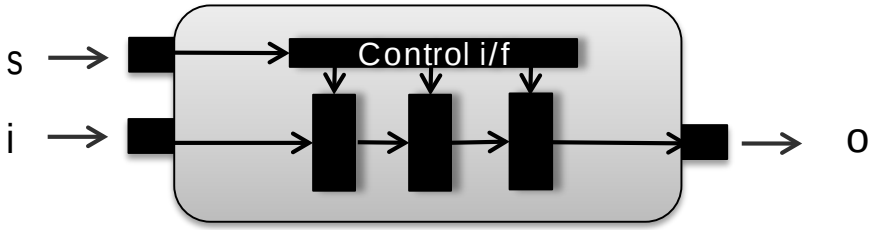
Advances In Verification Technology



Review of Hardware Abstractions

- Algorithmic Model
  - No timing or architecture
- Transaction Level Model
  - Partitioned for hardware architecture
- RTL Implementation
  - Synthesizable to gates

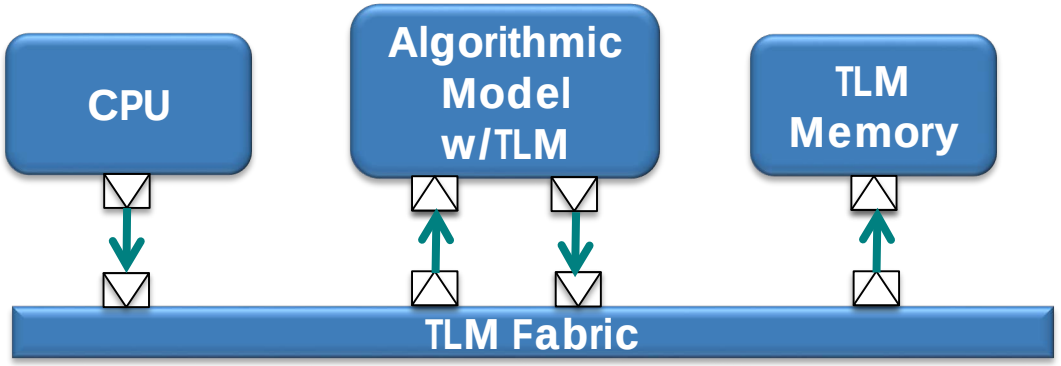




Verification in ESL Platform

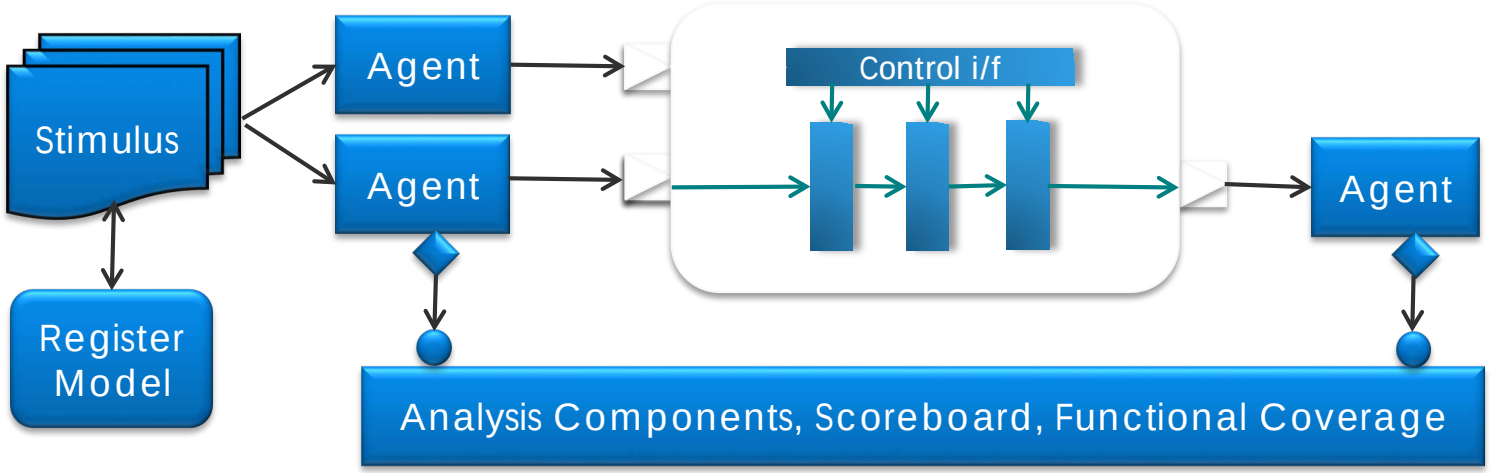
- Algorithmic Model can be used as a reference model
  - Can be embedded in SV/UVM environment
- Enables early software development
  - Software driven testing
- <10 minutes simulation vs. 1 month simulation in RTL

ESL Platform



Synthesizable TLM Verification

- Can be simulated effectively with UVM
  - Early start on UVM environment
- Leverage functional testing
- Based on Algorithmic Model, but partitioned for hardware
- Additional testing for internal control
- Limited performance testing
- Simulation ~100x faster than RTL



Coverage-Driven TLM Verification

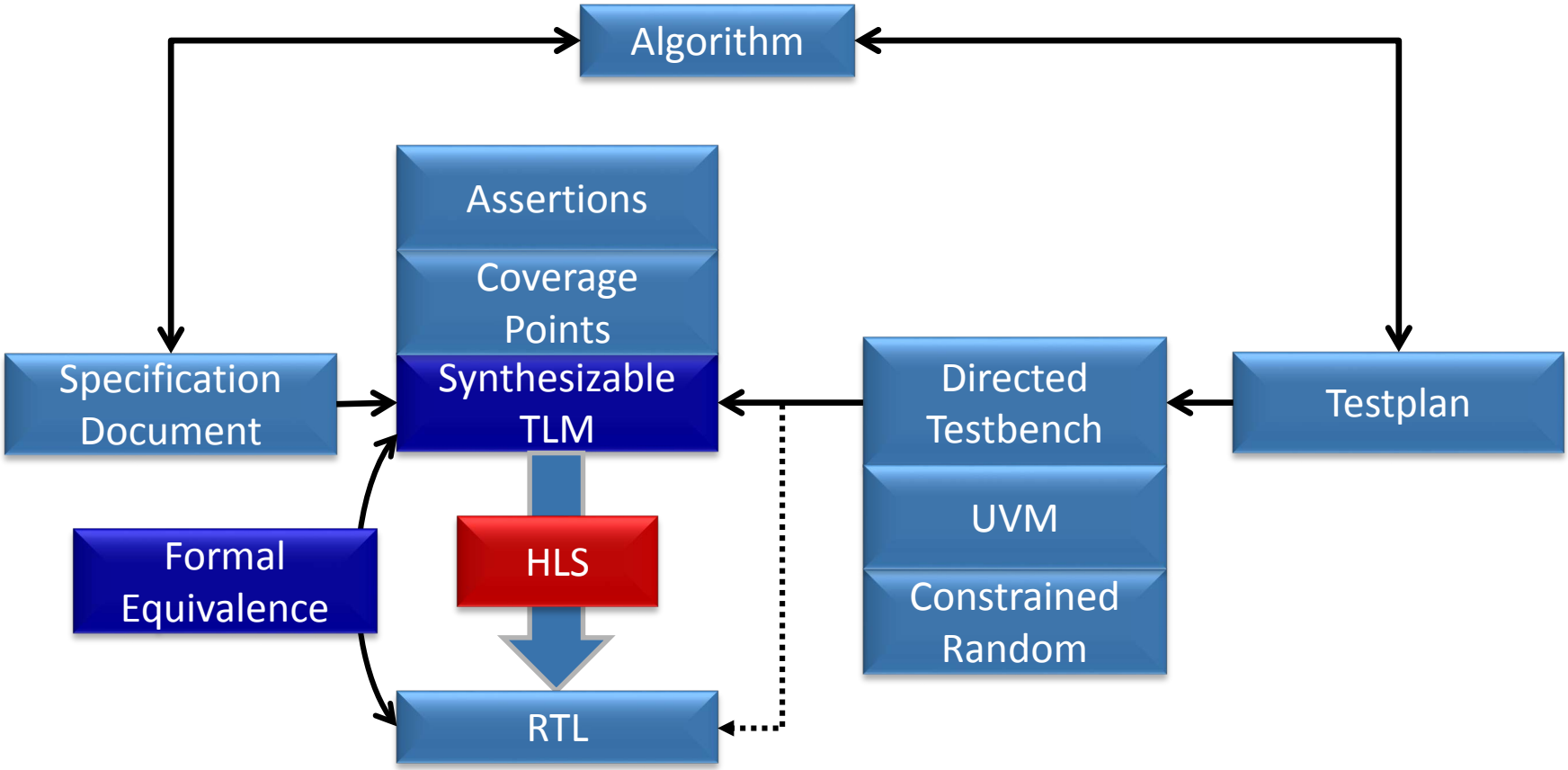
- Assertions and Cover Points
  - Functional
  - SystemC
- Testplan Coverage
  - Based on cover assertions
  - Some tests require RTL
- Code Coverage
  - Function, Line, Condition/Decision
  - Many C++ based tools
  - Nothing specialized for hardware

```
int18 alu(uint16 a, uint16 b, uint3 opcode)
{
  int18 r;

  switch(opcode) {
    case ADD:
      r = a+b;  break;
    case SUB:
      r = a-b;  break;
    case MUL:
      r = (0x00ff & a)*(0x00ff & b);  break;
    case DIV:
      r = a/b;  break;
    case MOD:
      r = a%b;  break;
    default:
      r = 0;    break;
  }

  assert(opcode<5);
  cover((opcode==ADD));
  cover((opcode==SUB));
  cover((opcode==MUL));
  cover((opcode==DIV));
  cover((opcode==MOD));

  return r;
}
```

RTL Coverage

- RTL Generated from TLM model by HLS
- Re-use SystemC Vectors
  - Will give functional coverage
  - Some gaps in branch/FSM
- Add RTL tests to cover RTL
  - FSM reset transitions
  - Stall tests
- Gives nearly 100% coverage
  - Line, branch, condition

HLS Verification



Export for SoC integration

Summary

- Increasing design complexity & shorter design cycles
  - RTL simulation based debug & verification is the bottleneck
  - Faster simulation (or emulation) is not enough on its own
- Moving to higher levels of abstraction for design & debug
  - Focus on verifying functionality, not implementation details
  - Significant simulation performance & debug improvement
- Requiring automated generation of RTL from TLMs
  - Technology targeting
  - Power Performance Area analysis & optimization
  - Verifiably correct by construction
- Adopting HLS methodology shortens verification timescales
  - Majority of functional verification at algorithmic/TLM levels
  - Minimal RTL simulation and/or formal equivalence checks to prove RTL is correct

HLS in the Wild
-- Intel's experience

Bob Condon, Intel DTS

Hi…

- Bob Condon - past 5 years at Intel
  - (past life HLS, FV, Logic Synthesis at Mentor and Exemplar)
  - Coach new teams adopting HLS adoption
  - HLS-specific tools and libraries
- Disclaimers
  - I won’t talk about specific vendor tools.
  - I won’t talk about specific Intel products.
  - “Customers” are internal Intel product groups designing RTL IP which will get integrated into a full SOC.

Spoiler Alert…

- Many production teams at Intel are using SystemC- based High-Level Synthesis to produce the RTL we ship in product.
- These designs include both algorithm dominated designs and control dominated designs.
- The groups who are happiest report :

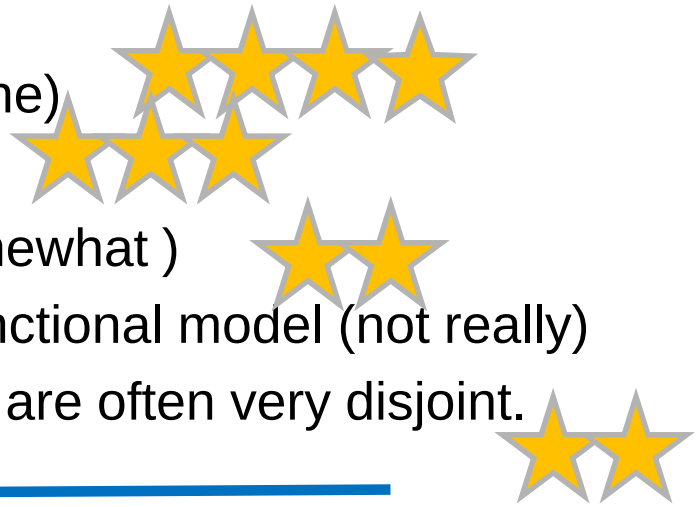
  “The HLS flow got us to meet the ___ RTL readiness milestone ___ weeks faster than we estimate with our hand-written RTL approach”

Why adopt HLS?

Marketing pitch gives lots of reasons:

- Retarget new process technology
- Automatic (or rapid) design exploration
- Free simulation
- Faster time to validated RTL
- Code is easier to modify
- Eliminates the need for hardware designers
- Provides single source with the VP/Functional model
- Design is “correct by construction”

Reality Check



- Faster time to validated RTL (the big one)
- Code is easier to modify (pretty big)
- Retarget new process technology (somewhat)
- Provides single source with the VP/Functional model (not really)
  - You can share code but these teams are often very disjoint.

(Not worth it….)

- Automatically do design exploration (not much)
- Free simulation (nope)
- Eliminates the need for hardware designers (nope)
- Design is “correct by construction” (myth)

Plan for success…

- Project
  - Under time pressure
  - Has a significant amount of new code
  - Has line of sight to a derivative
  - A C/C++ model of some flavor exists
  - The project size corresponds to the “testability” size
- Team
  - >= 4 people with skin in the game
  - at least one of them has decent C++ skills
  - lined up HLS support
  - Verification and Product build team involved
- The first deliverable is a Smoke Test testbench
- Verification team and Build team is involved early

Who does the work?

- 3 Pools of people
  - Verilog coders moving up a level of abstraction
    - ask them to anticipate a “dreaded” change
    - C++ is often a hurdle
    - Symptom – they write an SC_METHOD in their first design.
  - Architects – Our sweet spot
    - “Is overall design better if we tradeoff bus traffic for a bigger RAM?”
  - Algorithm specialists (we don’t really see them doing much HLS)
    - Hardware knowledge is still critical
    - Some software techniques work against HLS

DataPath vs Control

We do both and HLS is a win for both

- DataPath designs rely a lot on the HLS tools –
  - automatic pipelining,
  - common subexpression extraction.
- Control based designs rely on lots of use of C++ idioms.
- Things that are hard get implemented as library components.
  - Start to think of re-use (IP?) differently
  - DataPath: A FIR filter with three taps (traditional “algorithm” IP)
  - Control: A unknown block with Streaming Input, Streaming output, reading coefficients from a RAM and the ability to flush FIFOS on an interrupt.

How do I integrate to my backend flow?

- HLS output is “generated” RTL (gRTL).
  - Use the same flows as for your h(and)RTL. (We relax some lint rules)
- May need a RTL wrapper to leave exactly the same pins as before including things like scan.
- The gRTL is uglier
  - Minimize the amount of debugging there.
- Add monitors if you need them
- What about ECOs?
  - We see very few. ECO modes of the tools are satisfactory

How do I verify?

- Same as today
  - really – same way you validated the architectural model against your current RTL
  - RTL still needed for final verification.
  - The source is (usually) multi-threaded and not cycle accurate.
    - Formal only works in restricted domains (and with formal expertise).
  - Still need a full testplan to release quality silicon.
- Find bugs with “cheapest” test possible
  - HLS designs ready before full SV test ready
  - Some flavor of model (vectors, c++ code, matlab exists) – use it
  - Find (as many) algorithm bugs as possible in the fast SystemC simulation.
  - Mixed language sim to find final communication bugs (and spec changes)
- HLS lets you find and fix your bugs faster

déjà vu all over again…

- Many production teams at Intel are using SystemC- based High-Level Synthesis to produce the RTL we ship in product.
- These designs include both algorithm dominated designs and control dominated designs.
- The groups who are happiest report :

  “The HLS flow got us to meet the ___ RTL readiness milestone ___ weeks faster than we estimate with our hand-written RTL approach”

accellera SYSTEMS INITIATIVE™

Questions/Comments?