2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC: Focusing on High-Level Synthesis and Functional Coverage for SystemC

Organizers: Dragos Dospinescu - *AMIQ* and Mark Glasser - *NVIDIA*

- High-Level Synthesis: An Introduction - Frederic Doucet - *Facebook*
- High Level Synthesis: Model Structure and Data Types - Mike Meredith - *Cadence*
- High Level Synthesis: Lessons Learned - Bob Condon - *Intel*
- Functional Coverage for SystemC (FC4SC) - Dragos Dospinescu - *AMIQ*
- Accellera SystemC Working Group Update - Mike Meredith - *Cadence* and Martin Barnasconi, *NXP*

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC: Focusing on High-Level Synthesis and Functional Coverage for SystemC

PART 1

High-level Synthesis: An Introduction

Frederic Doucet
Facebook
Menlo Park, CA

accellera SYSTEMS INITIATIVE

High-level Synthesis Overview

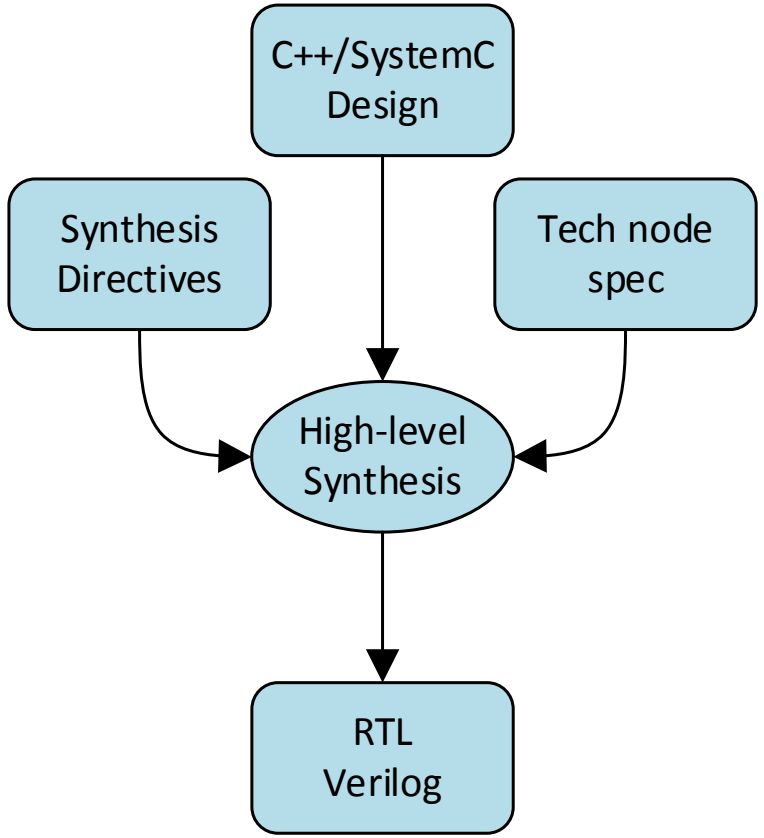
- SystemC / C++ based design with HLS
  - Higher level of abstraction than Verilog

- Thousands of tapeouts on a variety of designs
  - From very small to very large!
  - Example of sizes of synthesized SystemC processes
    - Small ~1k - 10k instances
    - Large ~100k instances
    - Very large ~500k instances
  - Large datapaths, control mixed with datapath, etc.
  - Significant productivity increases, get to the finish line faster

What does this all means?
How does it work?
How is it different than RTL?

High-level Synthesis Overview

HLS tool transforms synthesizable C++/SystemC code into RTL Verilog

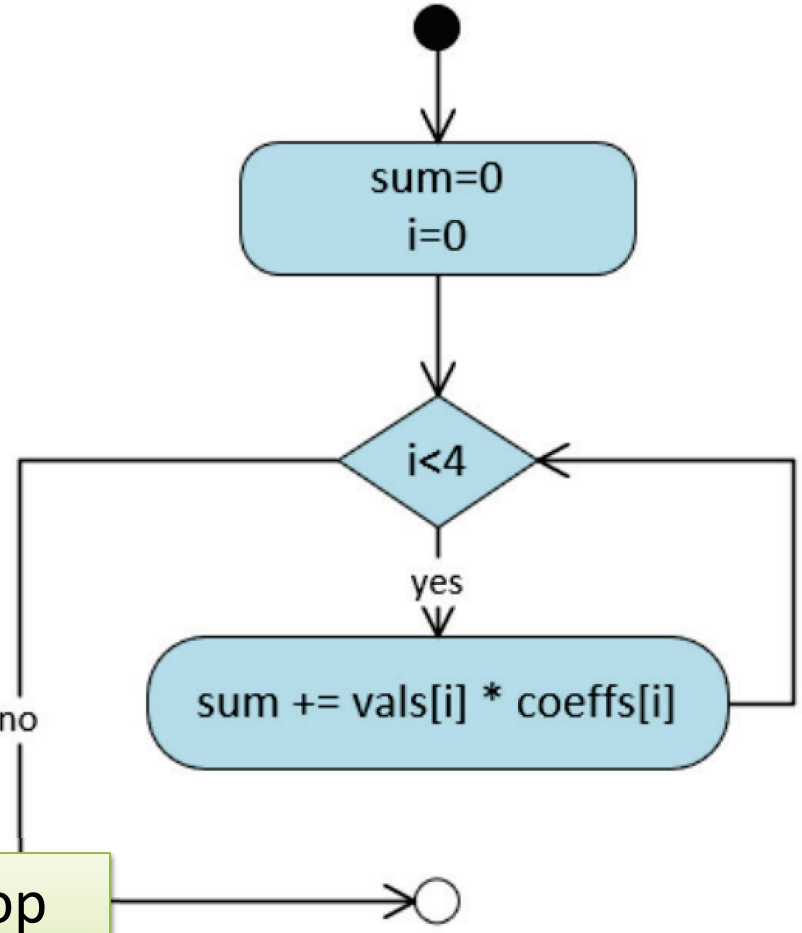


1. Elaborate C++/SystemC code describing the design
2. Apply designer-specified synthesis directives/constraints
3. Characterize resources for all operations
4. Schedule all operations onto available clock cycles
5. Generate RTL that is “equivalent” to the input

Describing Computation with C++

Datapath functions:

```
1: int compute(int val[4], int coef[4])
2: {
3:    int sum = 0;
4:    for (int i=0; i<4; i++) {
5:      sum += val[i]*coef[i];
6:    }
7:    return sum;
8: }
```



DSP processing, Image processing, etc.

HLS tool sees the body of the function as a loop

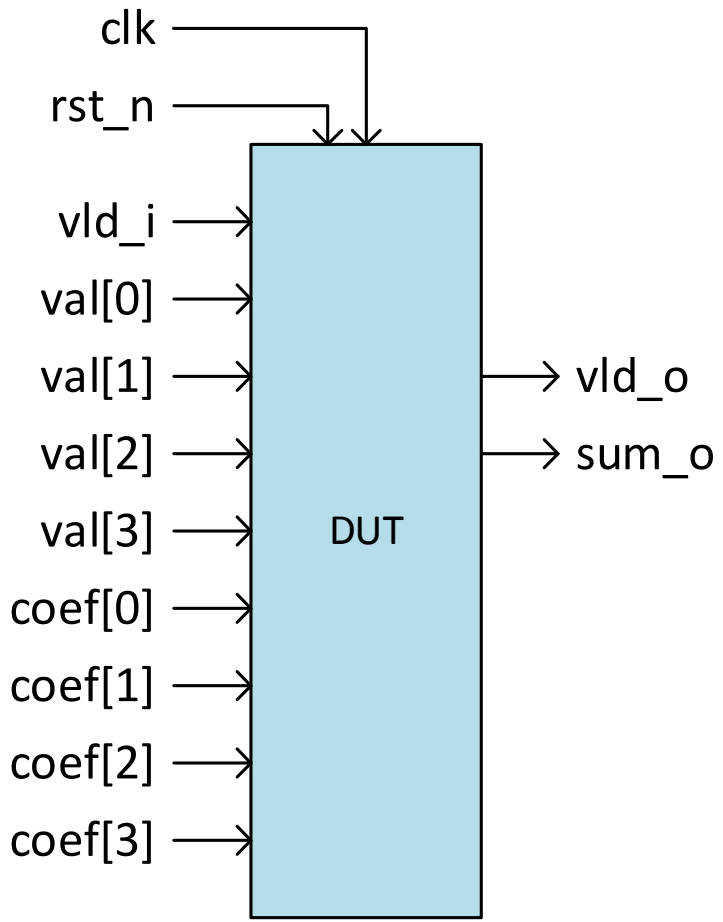
- What does it mean in hardware?

Hardware Modeling with SystemC

- SystemC: *Syntax to model hardware in C++*
  - Modules, ports, signals, processes, clocks, resets bit accurate datatypes, channels, etc.
- SystemC module:
  - Provides the I/O interface of the design, and clock and reset specifications
  - Describes structure of the design: sub-modules, connections, etc.
- SystemC process:
  - Defines I/O behavior and control *around calls to datapath functions*
  - Specifies the control flow (usually with an implicit FSM)
    - Will be “concretized” by HLS tool into FSM/datapath in the RTL

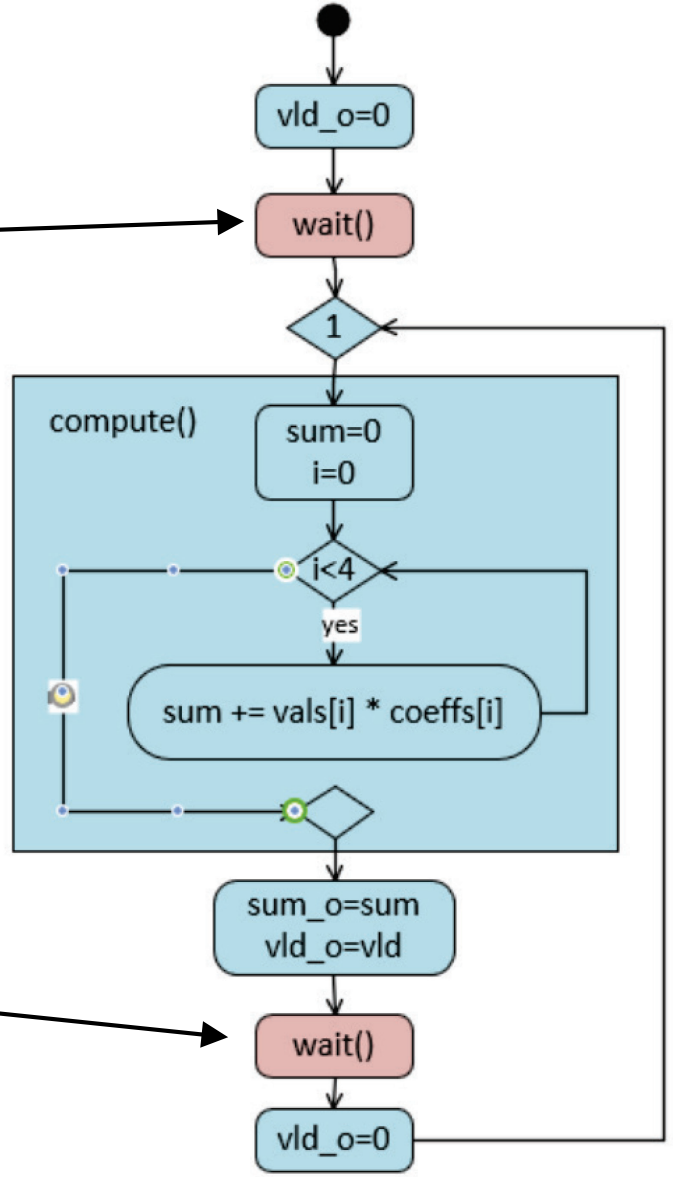
SystemC Module

```
SC_MODULE(DUT) {
  sc_in<bool>             clk;
  sc_in<bool>             rst_n;
  sc_in<bool>             vld_i;
  sc_in<sc_uint<16> >     vals_i   [N];
  sc_in<sc_uint<16> >     coeffs_i[N];
  sc_out<bool>            vld_o;
  sc_out<sc_uint<16> >    sum_o;
  ...
  SC_CTOR(DUT) {
    SC_THREAD(process);
    sensitive << clk.pos();
    reset_signal_is(rst_n,0);
  }
  ...
  void process() { ... }
};
```



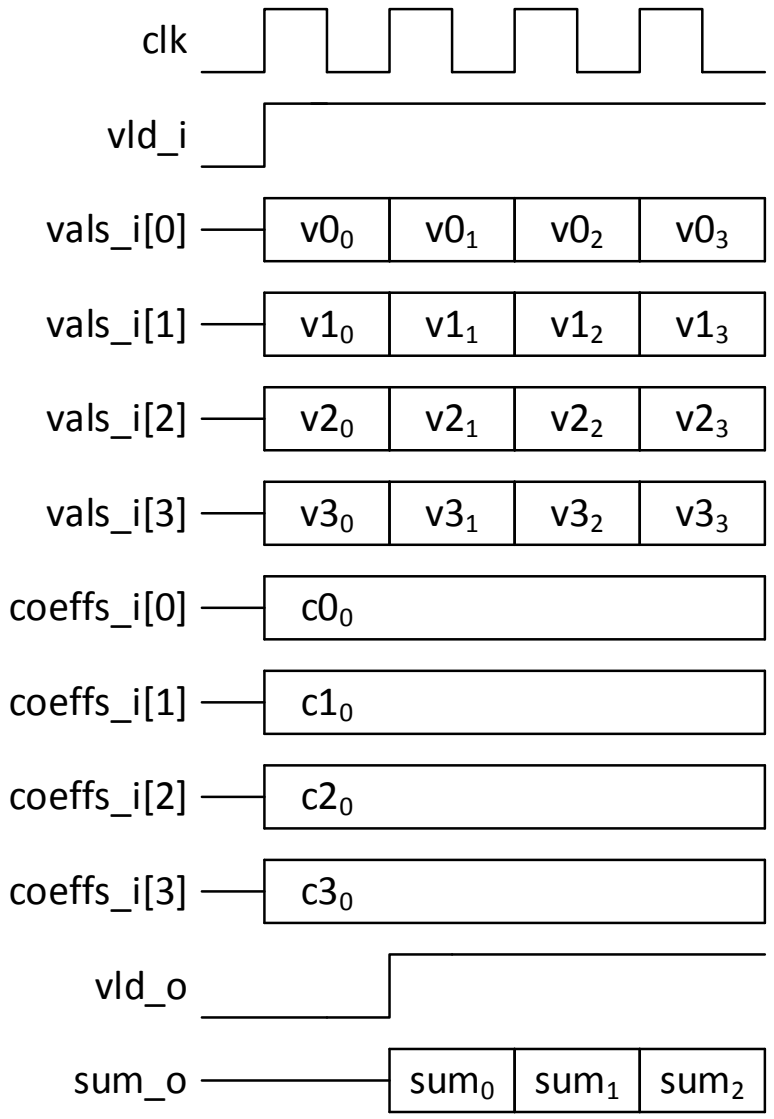
SystemC Process

```
...
void process() {
  vld_o.write(0);
  wait();
  while (1) {
    bool input_vld = vld_i.read();
    sc_uint<16> vals[4], coeffs[4];
    for (int i=0; i<4; i++) {
      vals[i]   = vals_i  [i].read();
      coeffs[i] = coeffs_i[i].read();
    }
    sc_uint<16> sum = compute(vals, coeffs);
    vld_o.write(input_vld);
    sum_o = write(sum);
    wait();
    vld_o.write(0);
  }
}
...
```



SystemC I/O Behavior

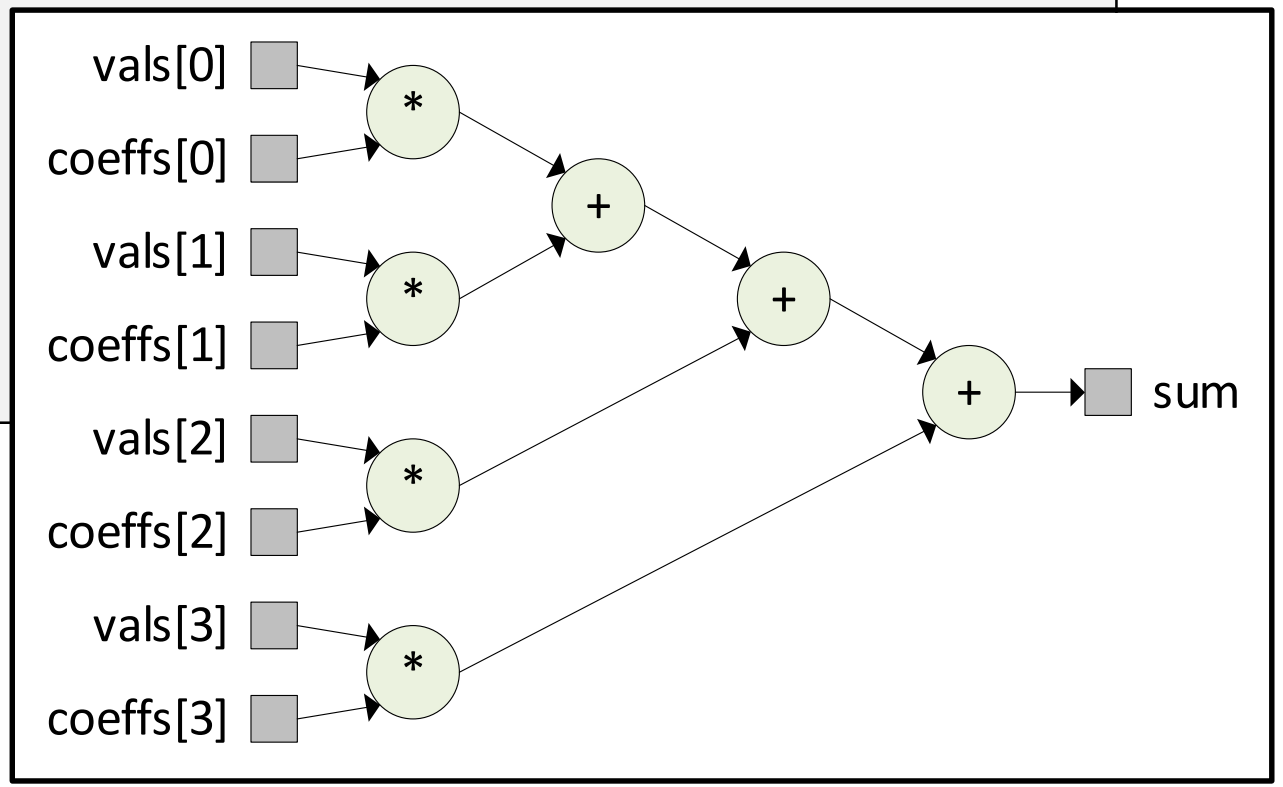
```
...
void process() {
  vld_o.write(0);
  wait();
  while (1) {
    bool input_vld = vld_i.read();
    sc_uint<16> vals[4], coeffs[4];
    for (int i=0; i<4; i++) {
      vals[i]   = vals_i  [i].read();
      coeffs[i] = coeffs_i[i].read();
    }
    sc_uint<16> sum = compute(vals, coeffs);
    vld_o.write(input_vld);
    sum_o = write(sum);
    wait();
    vld_o.write(0);
  }
}
...
```



Synthesis Directives: Provide Hardware Design Intent

Tell the HLS tool how to transform C++ structures in hardware structures

```
1: sc_uint<16> compute(sc_uint<16> val [4], sc_uint<16> coef[4])
2: {
3:    sc_uint<16> sum = 0;
4:    for (int i=0; i<4; i++) {
5:       UNROLL_LOOP;
6:       sum += val[i] * coef[i];
7:    }
8:    return sum;
9: }
```



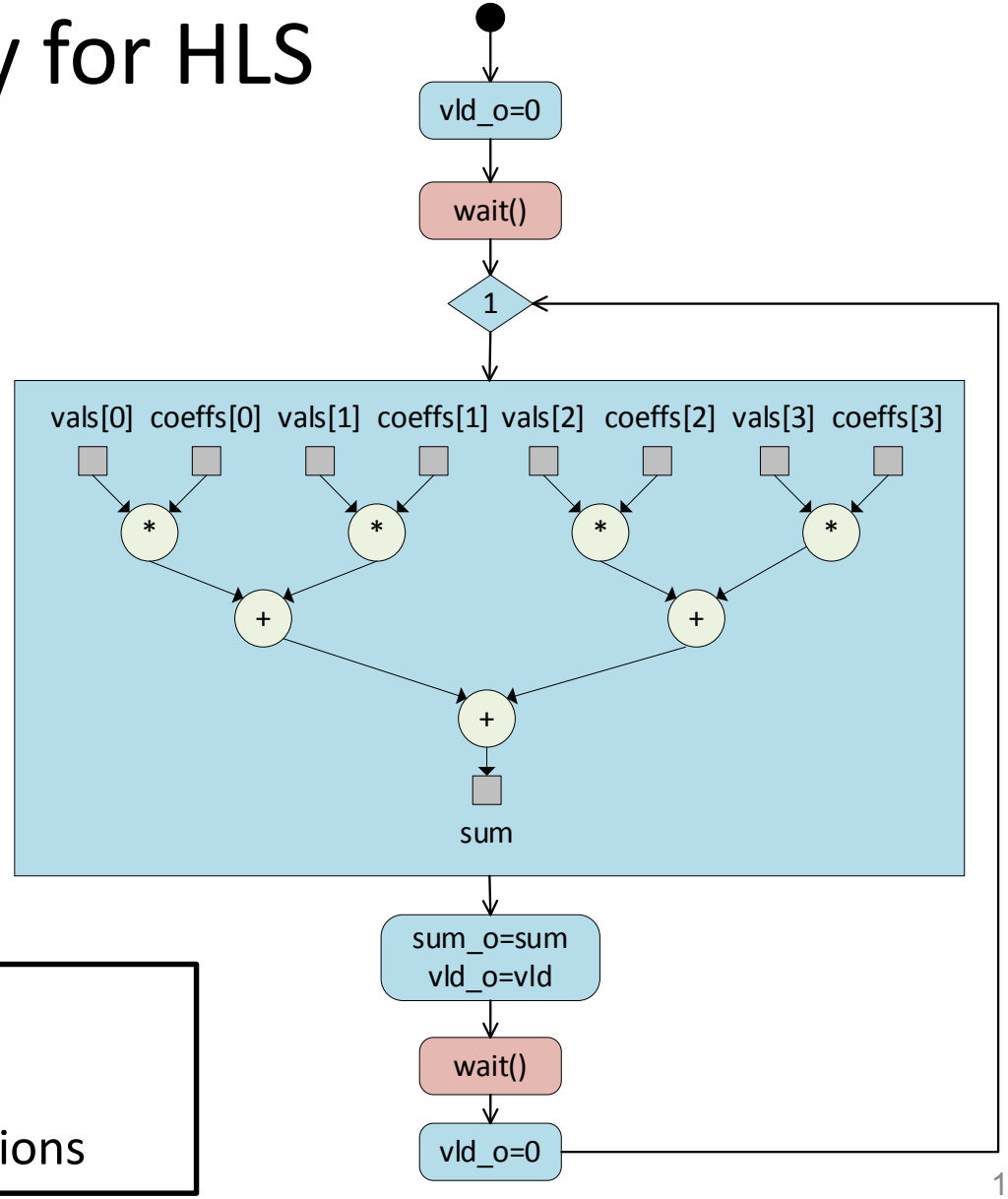
Unroll the loop:

all iterations to be executed

in parallel

SystemC + Directives = Hardware Model Ready for HLS

```
...
void process() {
  vld_o.write(0);
  wait();
  while (1) {
    bool input_vld = vld_i.read();
    sc_uint<16> vals[4], coeffs[4];
    for (int i=0; i<4; i++) {
      vals[i]   = vals_i  [i].read();
      coeffs[i] = coeffs_i[i].read();
    }
    sc_uint<16> sum = compute(vals, coeffs);
    vld_o.write(input_vld);
    sum_o = write(sum);
    wait();
    vld_o.write(0);
  }
}
...
```



With directives:

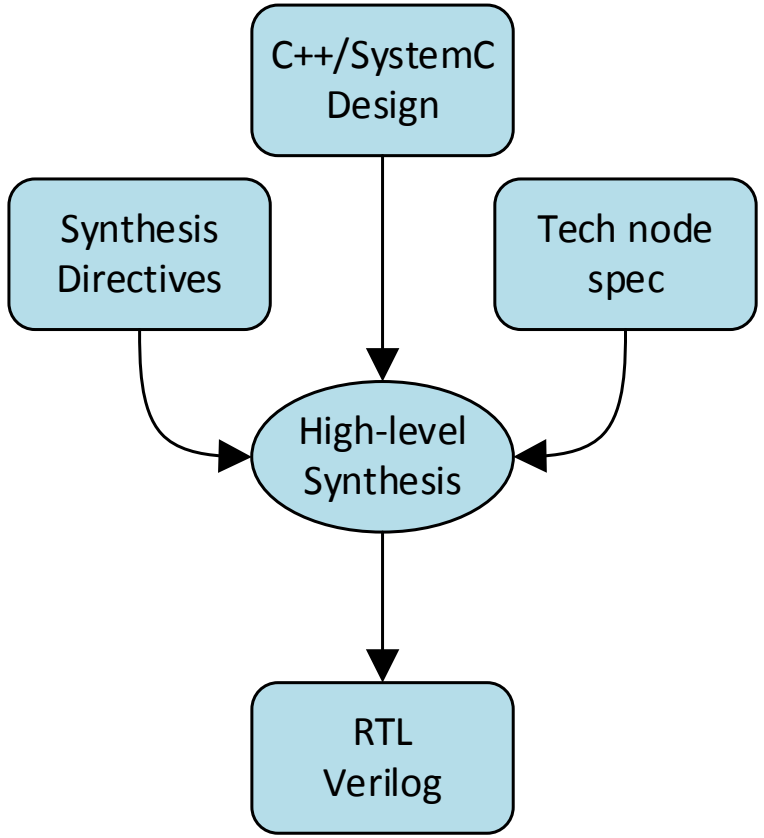
- unroll loops
- balanced expressions

HLS: Cycle-Accurate Design

- Directives / constraints:
  - Unroll loops
  - Balance expressions
  - Clock period: 0.7ns
  - Scheduling: cycle accurate

High-level Synthesis Overview

HLS tool transforms synthesizable C++/SystemC code into RTL Verilog



- ✓ Elaborate C++/SystemC code describing the design
- ✓ Apply designer-specified synthesis directives / constraints
- ➢ Characterize resources for all operations

4. Schedule all operations onto available clock cycles

5. Generate RTL that is “equivalent” to the input

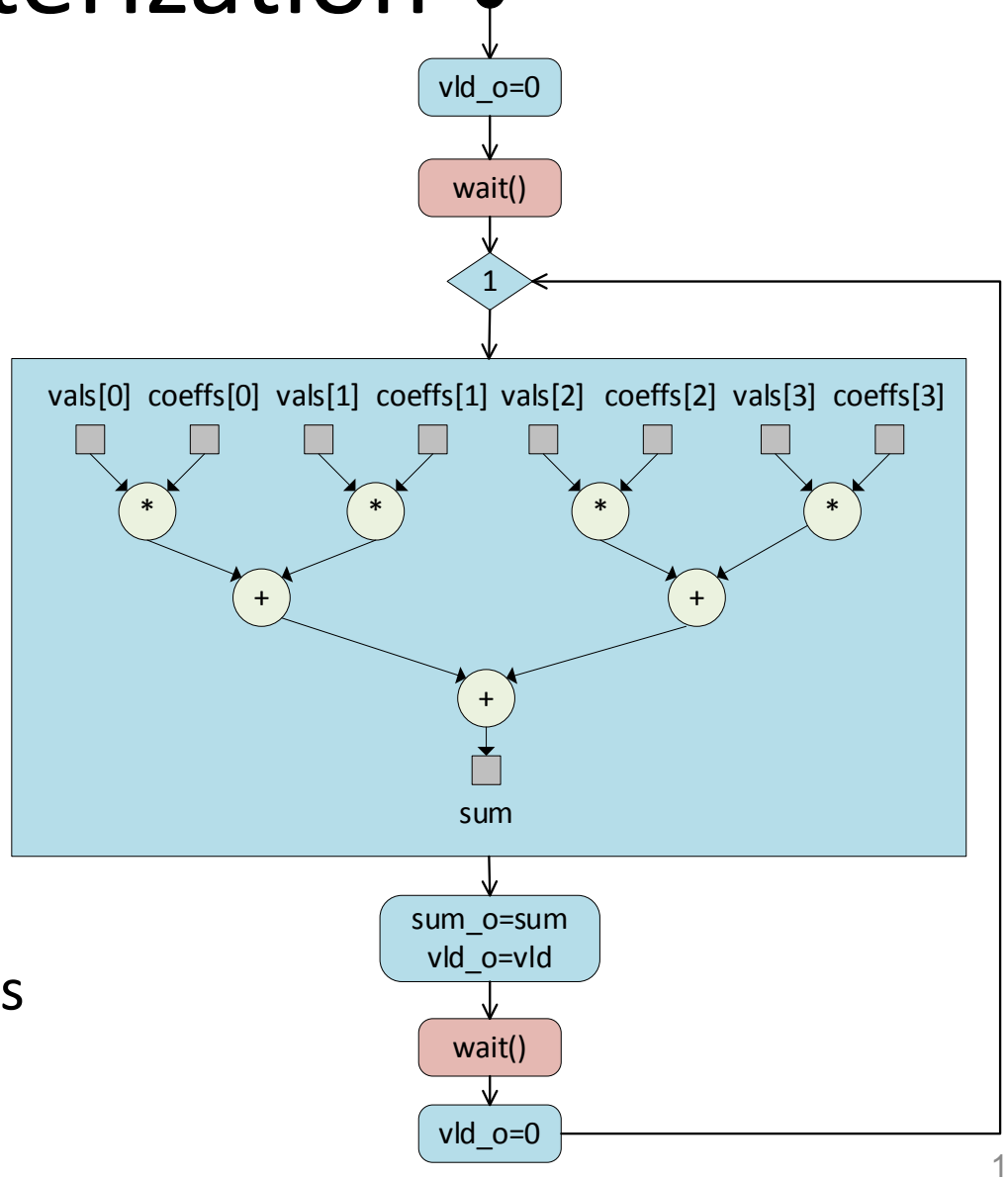
Resource Characterization

For all operations in the design,

HLS tool characterizes resources for delay and area

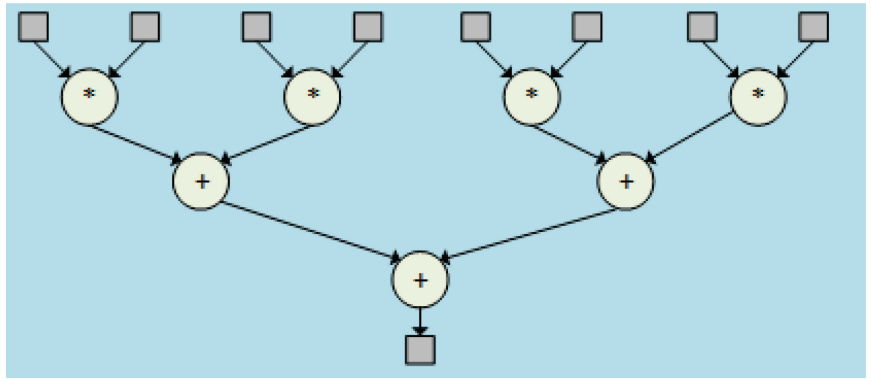
Resource	Size	Grade	Delay	Area
Multiplier	16x16x16	Fast	0.27	70
		Slow	0.5	36
Adder	16x16x16	Fast	0.1	15
		Slow	0.3	6
Mux	16x4->16	Fast	0.1	4
		Slow	0.05	3
Register	16		0.04 / 0.03	6

HLS tool will use the combination of resource grades when exploring the different schedules

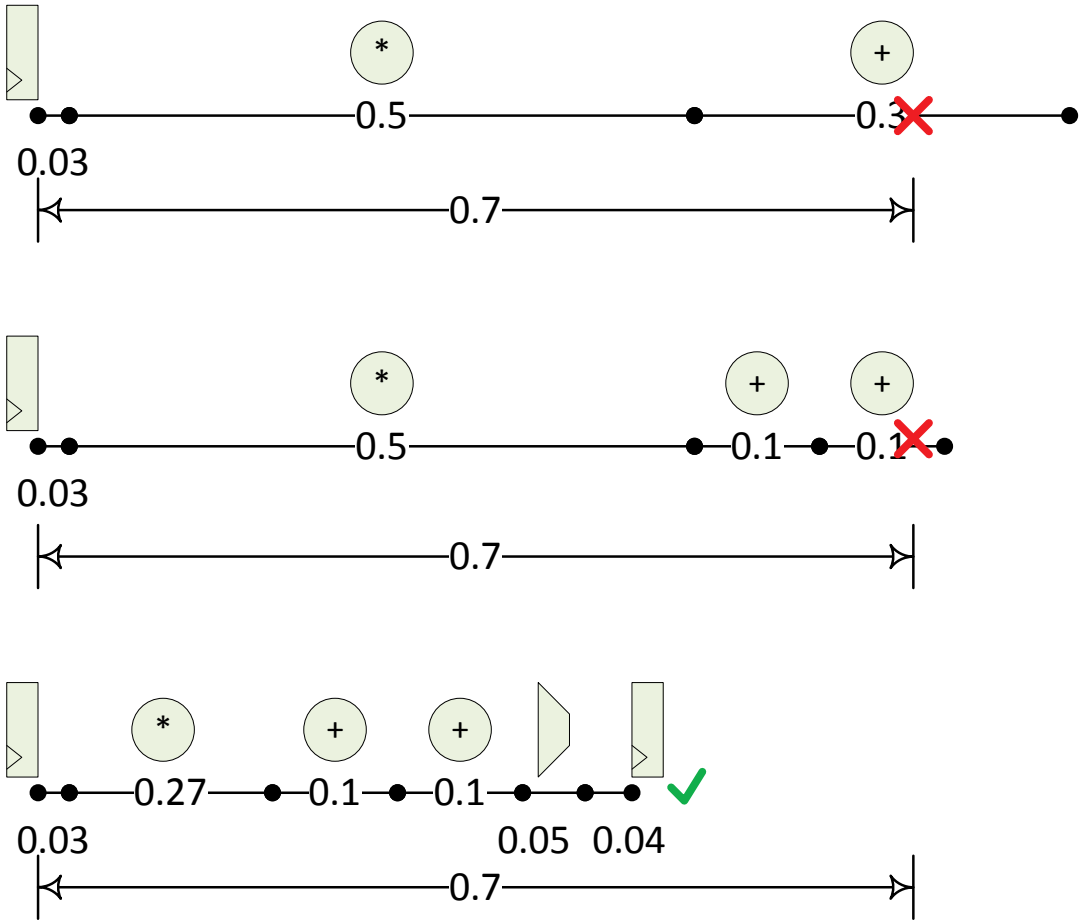


Operation Scheduling: Cycle Accurate

Resource	Size	Grade	Delay	Area
Multiplier	16x16x16	Fast	0.27	70
		Slow	0.5	36
Adder	16x16x16	Fast	0.1	15
		Slow	0.3	6
Mux	16x4->16	Fast	0.1	4
		Slow	0.05	3
Register	16		0.04 / 0.03	6



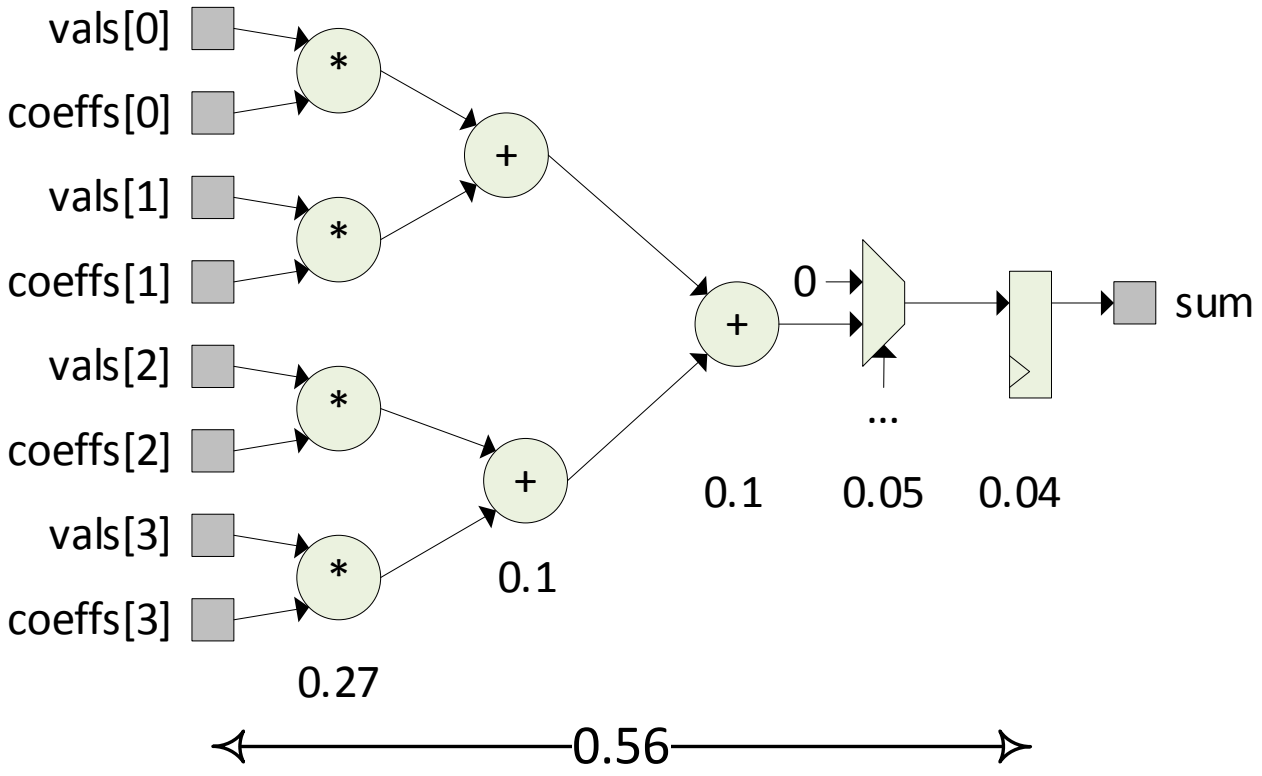
With clock period set to 0.7ns:



Cycle-Accurate Design: Generated RTL

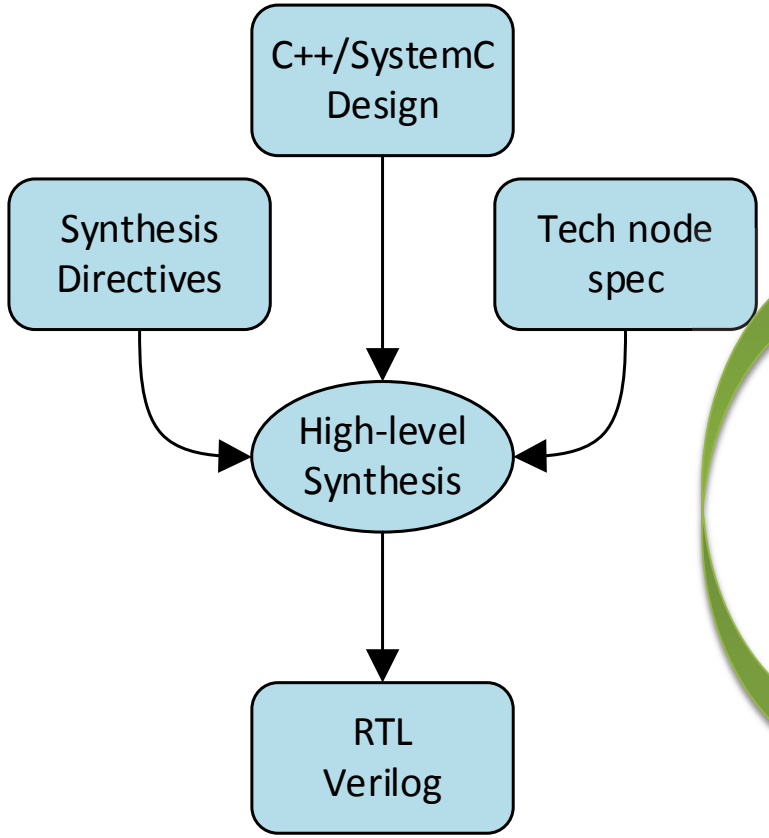
- Directives / constraints:
  - Unroll loops
  - Balance expressions
  - Clock period: 0.7ns
  - Scheduling: cycle accurate
- Area:
  - 4 fast multipliers, 3 fast adders

Micro-arch	Area	Thro.	Lat.
Cycle accurate	335	1	1



High-level Synthesis Overview

HLS tool transforms synthesizable C++/SystemC code into RTL Verilog



- ✓ Elaborate C++/SystemC code describing the design
- ✓ Apply designer-specified synthesis directives / constraints
- ✓ Characterize resources for all operations
- ✓ Sc cycles
- ✓ Generate RTL that is “equivalent” to the input

Let’s go back and try a different micro-architecture…

HLS: Minimal Area Design (1/4)

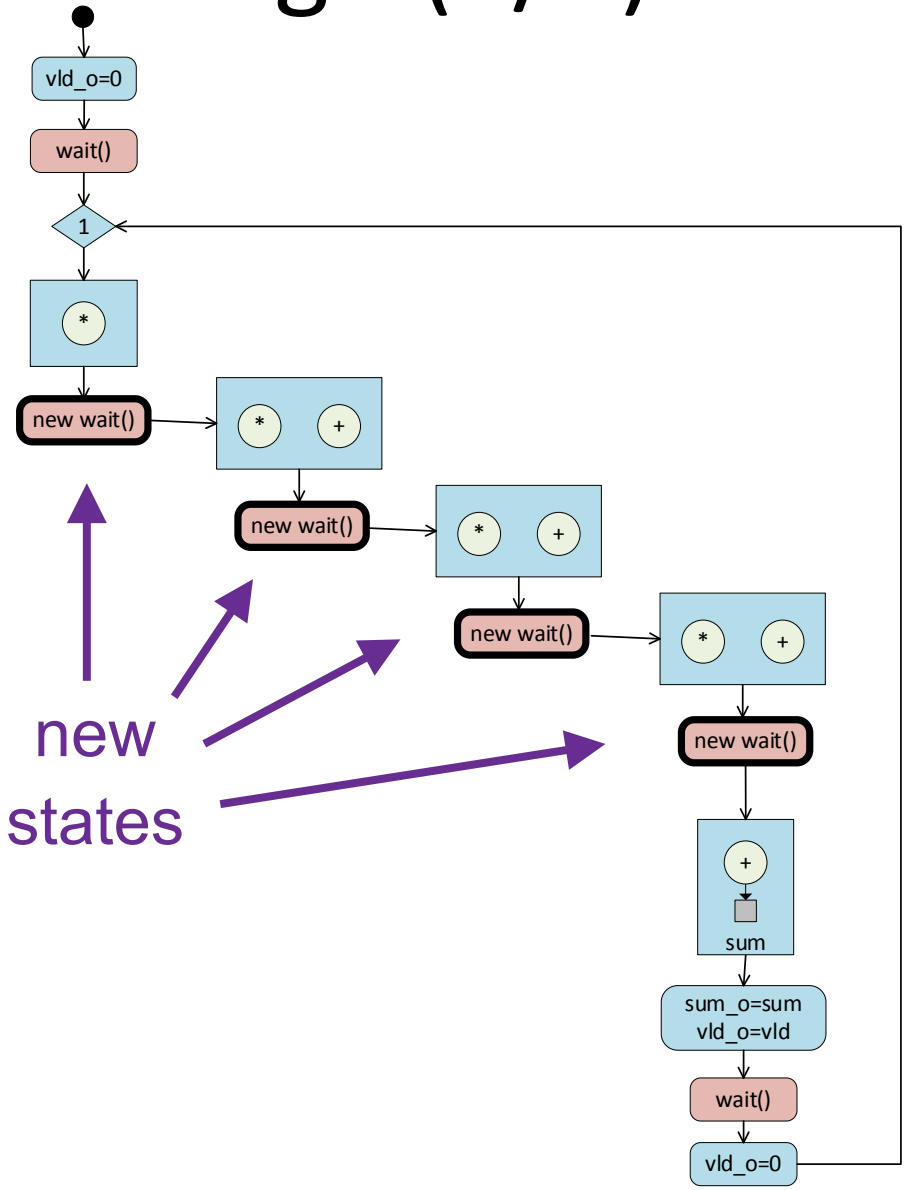
- Directives / constraints:
  - Unroll loops
  - Balance expressions
  - Clock period: 0.7ns
  - Scheduling: minimize area

Reduce area: Increase latency
to share resources and generate a new RTL

HLS: Minimal Area Design (2/4)

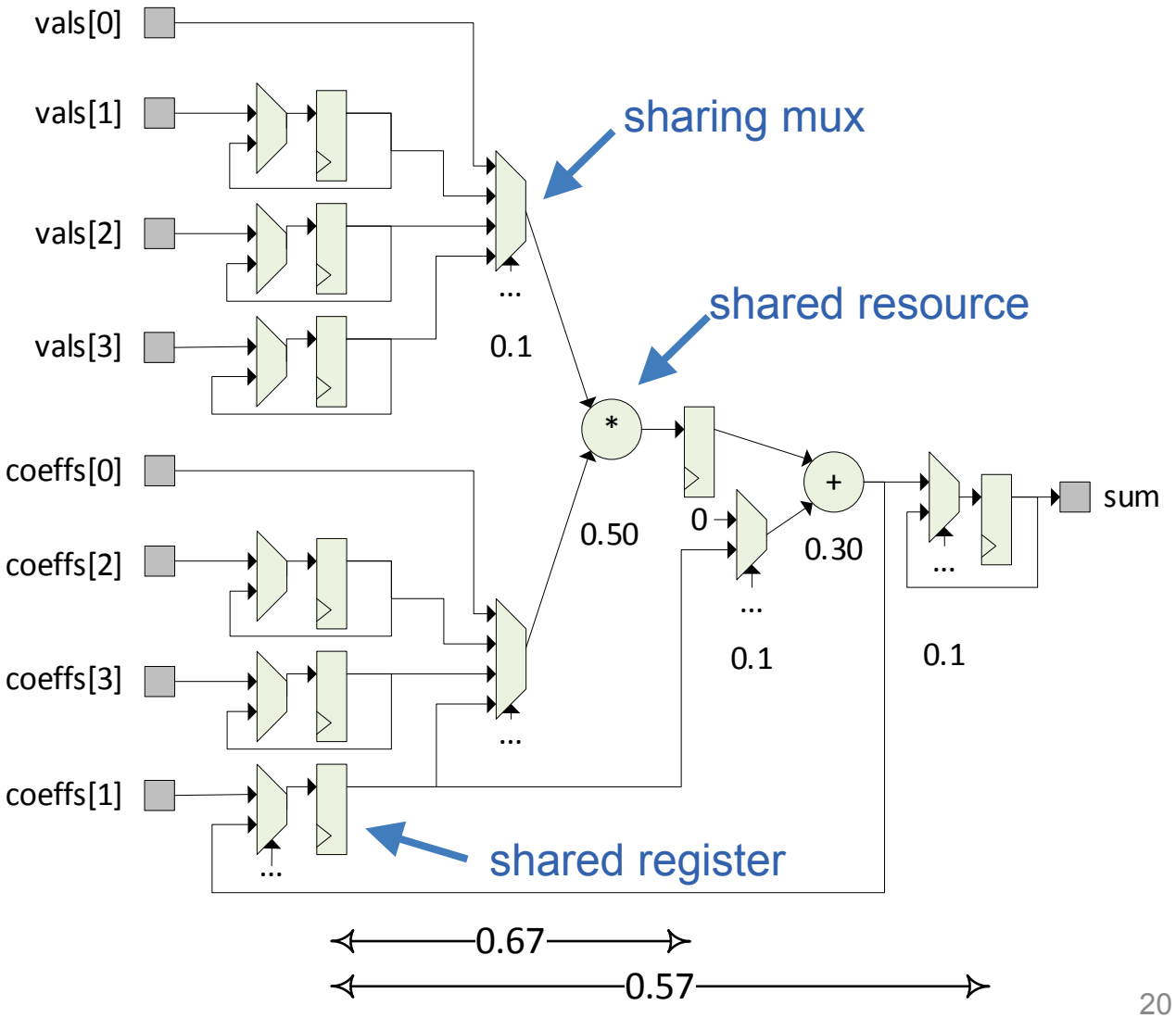
- Directives / constraints:
  - Unroll loops
  - Balance expressions
  - Clock period: 0.7ns
  - Scheduling: minimize area

- The state machine is changed
  - The scheduler adds **4 states** to share 1 multiplier for 4 multiplications
  - Adders are also shared



HLS: Minimal Area Design (3/4)

- Generated RTL will now include:
  - *shared resources*
  - *shared registers*
  - *sharing muxes*
- The generated FSM drives enables to sharing muxes and registers at the correct time

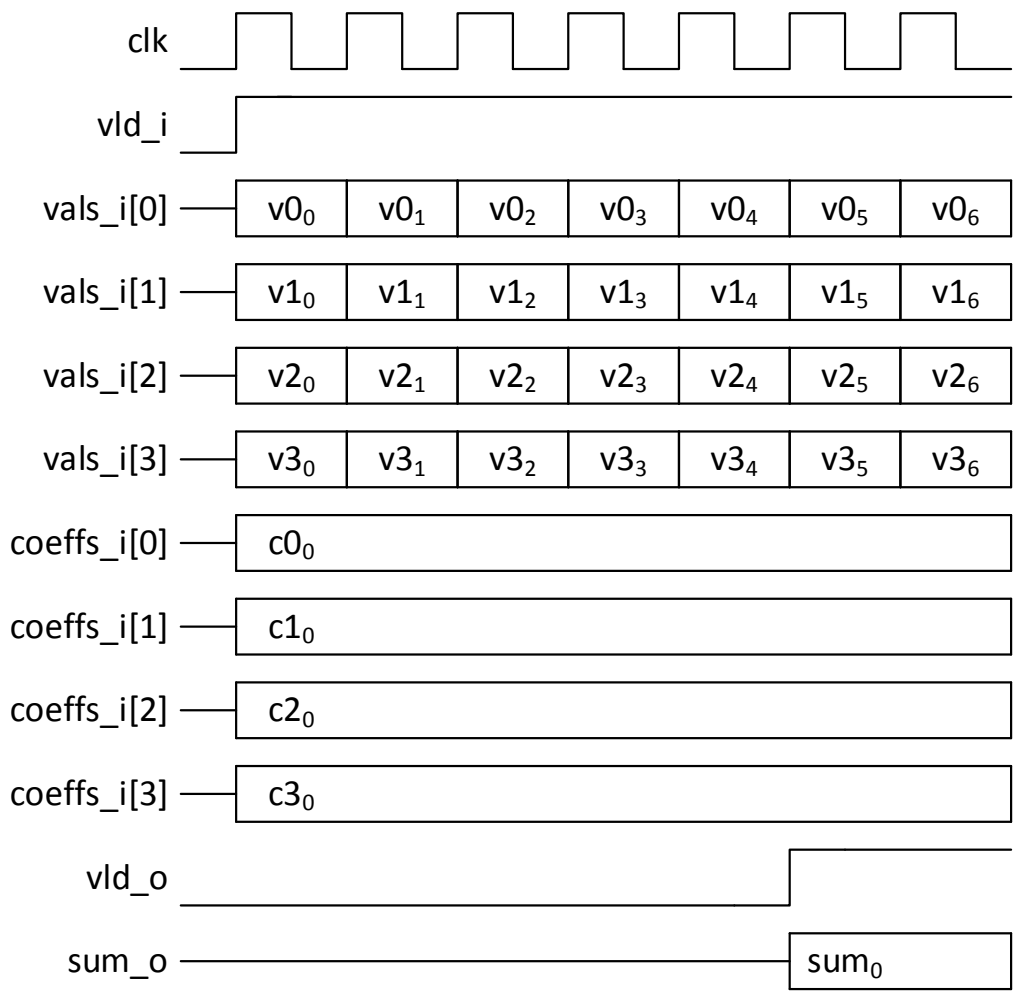


HLS: Minimal Area Design (4/4)

- Area ~1/3, but throughput 5clk

Micro-arch	Area	Thro.	Lat.
Cycle accurate	335	1	1
Min Area	139	5	5

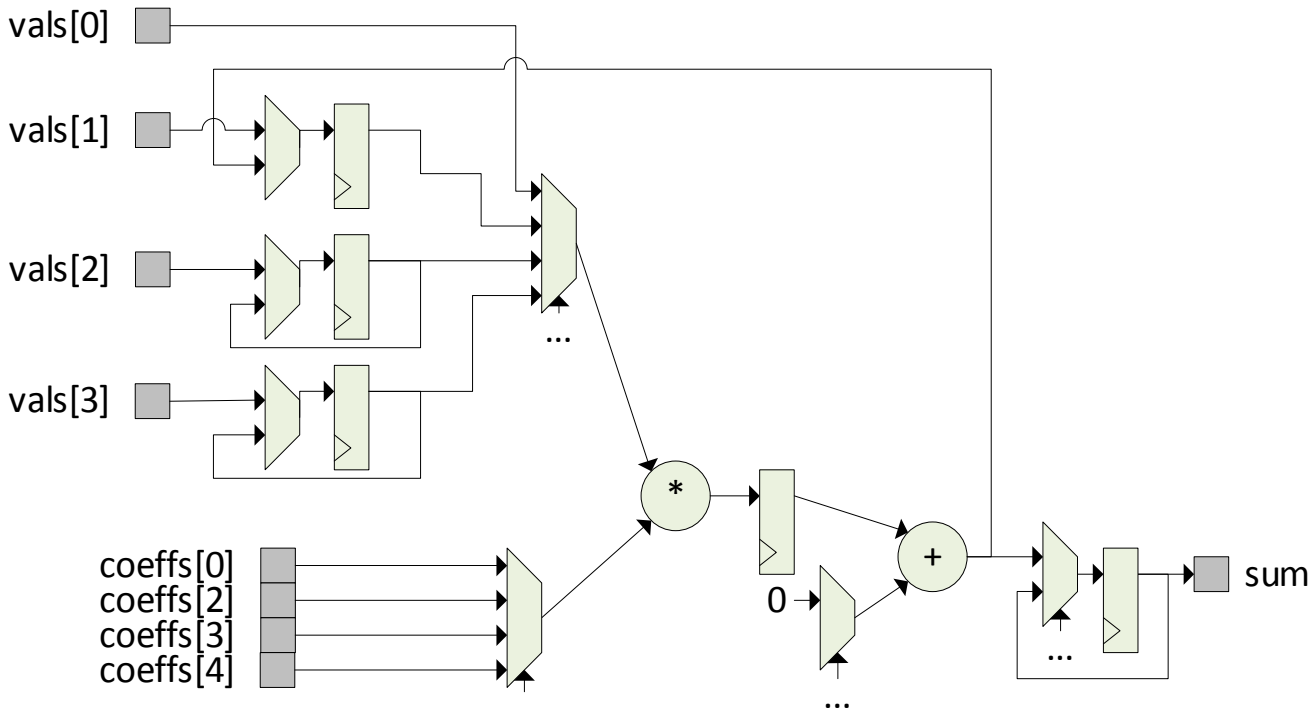
(No need to register the coeffs...)



HLS: Minimal Area Design, Stable Inputs

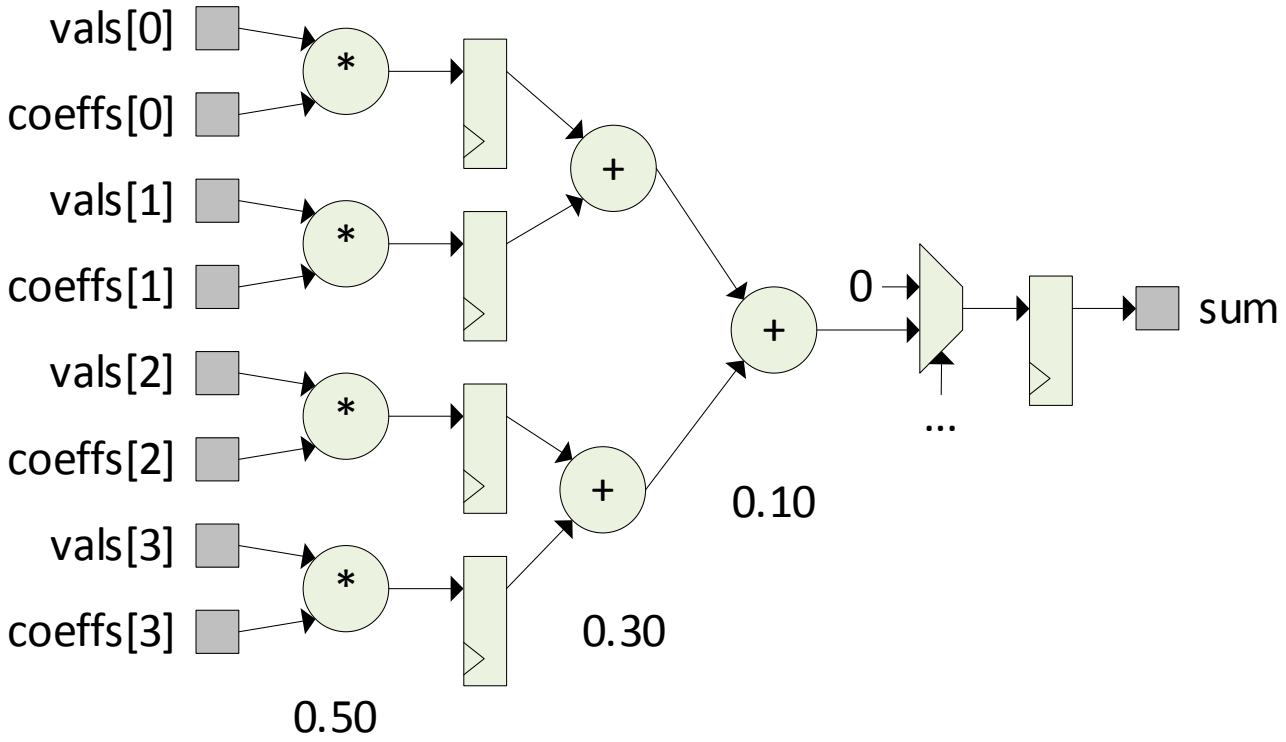
- Directives / constraints:
  - Unroll loops
  - Balance expressions
  - Minimize area
  - Coeffs inputs are stable
- 18% smaller, throughput 5clk

Micro-arch	Area	Thro.	Lat.
Cycle accurate	335	1	1
Min Area	139	5	5
Min area / stable inputs	115	5	5



HLS: Pipeline (1/2)

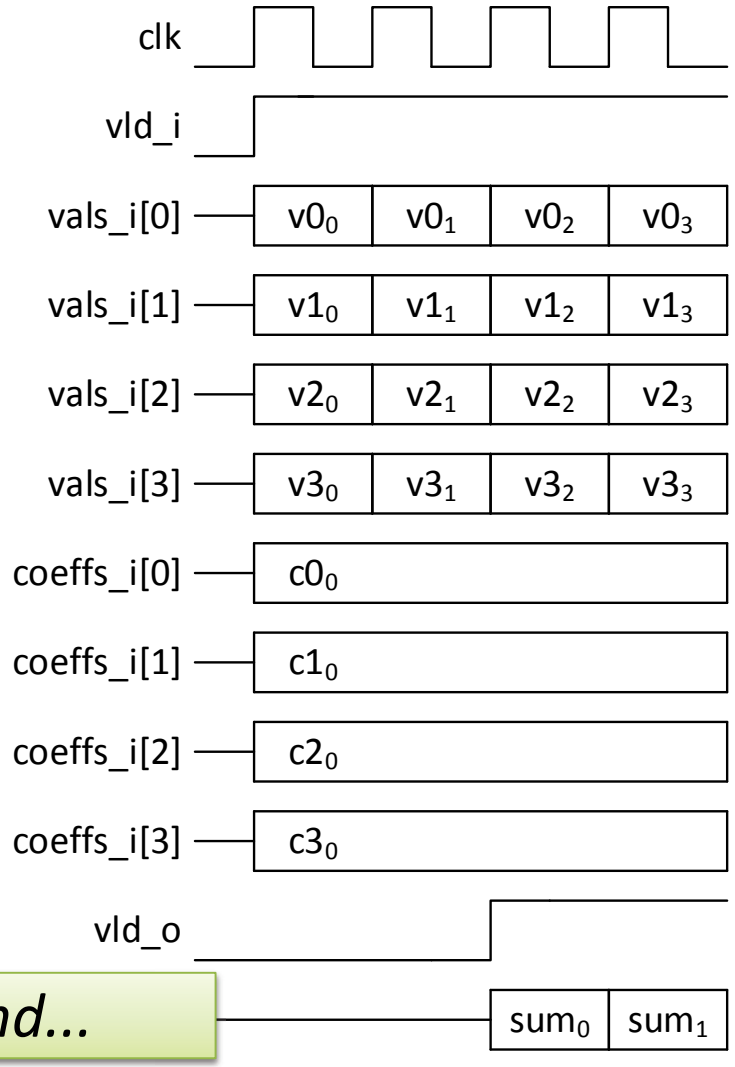
- Directives / constraints:
  - Unroll loops
  - Balance expressions
  - Pipeline
- Can use slow resources except for the last adder



HLS: Pipeline (2/2)

- Throughput is 1 per cycle
- Latency is now 2 cycles
- Significant area gain for extra cycle of latency

Micro-arch	Area	Thro.	Lat.
Cycle accurate	335	1	1
Min Area	139	5	5
Min area / stable inputs	115	5	5
Pipeline	205	1	2



Imagine making all these changes by hand...

Abstracted in SystemC, Refined by HLS

1. Operations to resource bindings and sharing muxes
   - Resource sharing depends on the synthesis directives (performance or area?)
2. Allocation and mapping of values to internal registers
   - Values in flight need to be registered
   - Depends on when the operation are mapped to the resources, which depends on the HLS directives
3. Creation of FSM states and transitions
   - wait() statements are converted to FSM states (in code, and added by tool)
   - Transitions between waits are FSM transitions
   - Current / next state logic generated by the tool

It does not turn random software into hardware!

Benefits of HLS

1. Fast design turnaround
   - Quickly implement large (micro-architecture) changes and regenerate RTL
   - Allows for fast micro-architecture exploration for design and qor optimizations
2. High-level verification
   - huge productivity benefits to verify and close coverage at SystemC level
   - Bit match datapath functions
   - Bugs are mostly in integration with other non-HLS RTL blocks
3. Get to finish line faster
   - Get a first version up and optimize it (when good enough, tape it out!)

Accellera SystemC Standardization

- Goal: Support ecosystem with multiple HLS vendors

- Further standardization work needed:
  - Channel/hierarchical port syntax
  - Channel libraries
    - fifos, point-to-point, memories, etc.
  - Standardization of Synthesis Directives
    - pipeline, loop unrolling, etc
    - syntax and interpretation
  - C++11 / C++14 support

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Thank you!

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC: Focusing on High-Level Synthesis and Functional Coverage for SystemC

PART 2
High-level Synthesis
SystemC Model Structure and Datatypes

Mike Meredith
Cadence Design Systems

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC Models

Primary purposes for use of SystemC

- Virtual platform modelling
  - Primarily for integration and validation of embedded software
  - TLM now part of IEEE 1666-2011 SystemC language standard
- High-level synthesis
  - As an alternative to traditional RTL design by hand
  - Accellera SystemC Synthesis Subset standard
- Verification
  - As glue for multiple languages and abstractions
  - Increasingly as a testbench language
  - Accellera SystemC Verification Library standard and new UVM-SystemC Library draft

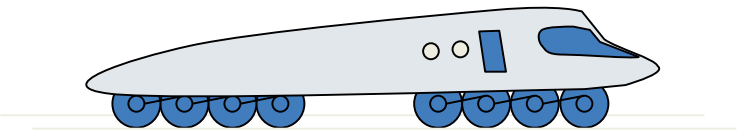
accellera SYSTEMS INITIATIVE

TLM Modeling

- TLM requirements: *SPEED!*
- Appropriate scope
  - System
- Appropriate detail
  - Memory map
  - Algorithm
  - Transaction order
- Appropriate techniques
  - Event sensitivity
  - Abstract communication with function calls through sc_port
  - Passing pointers to host memory
  - Any technique that will increase speed without losing necessary detail

Modeling for Synthesis

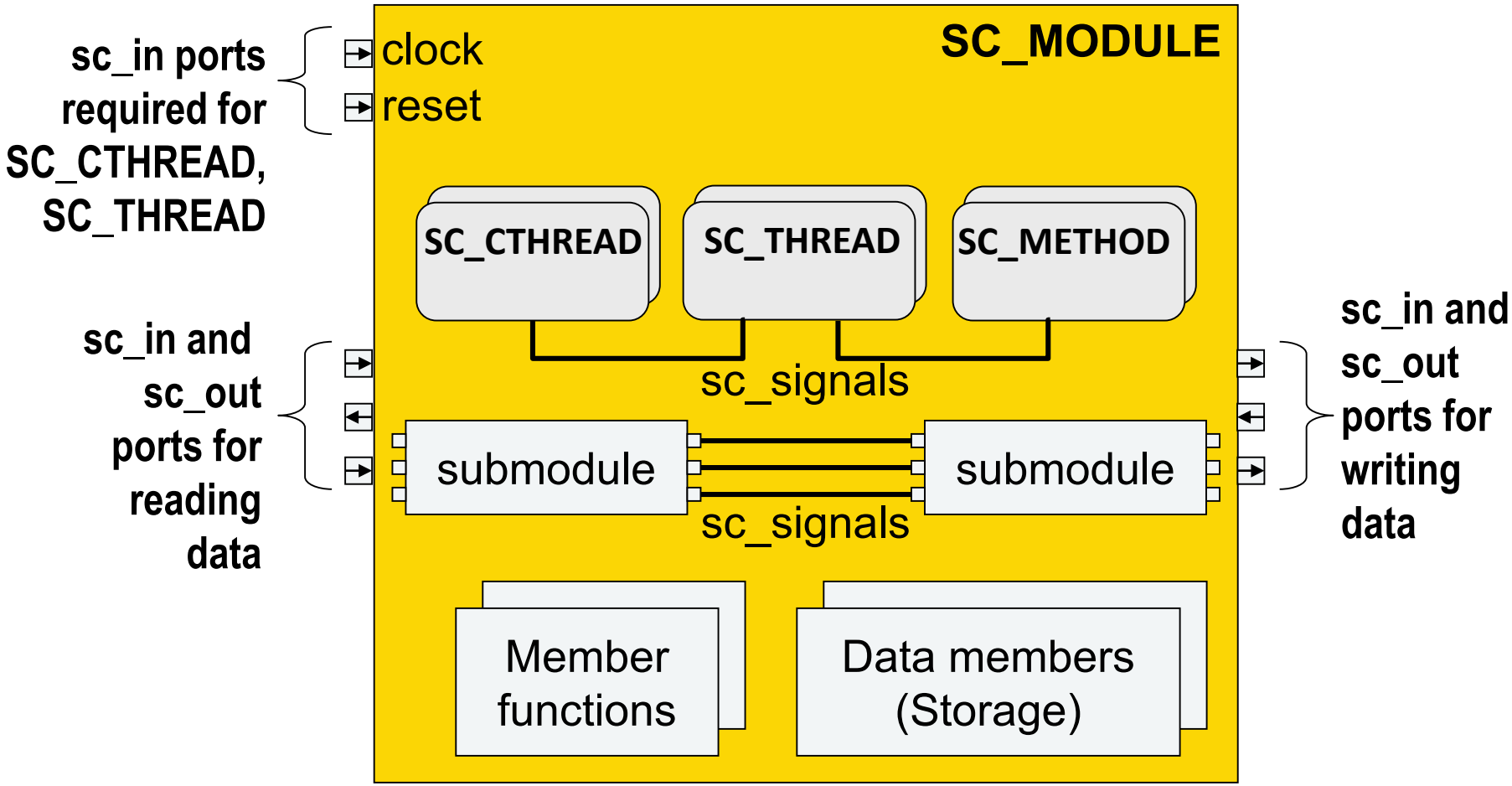
- Synthesis requirements: Bring the model down to earth
- Appropriate scope
  - Block, subsystem
- Appropriate detail
  - Cycle accuracy for protocol and control
  - Abstract algorithm for exploration
- Appropriate techniques
  - Clock sensitivity
  - Concrete communication with pin-level protocols
  - Detail modeling of reset behaviors
  - Abstract modeling of algorithm and and storage architecture



Modeling for Verification

- Appropriate scope
  - Block, subsystem, and system
  - For verifying virtual platforms and synthesizable implementations
- Appropriate techniques
  - Constrained random stimulus
  - Test sequences
  - Sequencer, driver, monitor functionality
  - Functional coverage

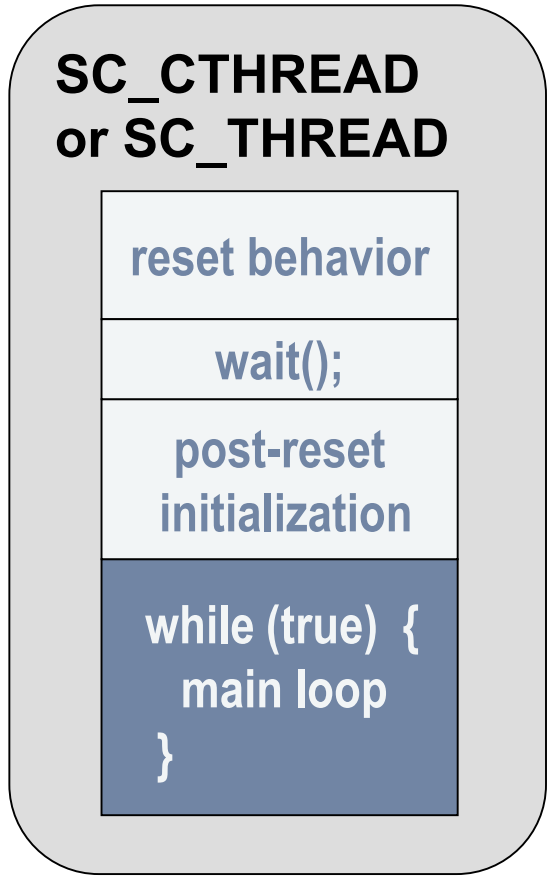
Module Structure for Synthesis



SC_CTHREAD and SC_THREAD Reset Semantics

For Simulation

- 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 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

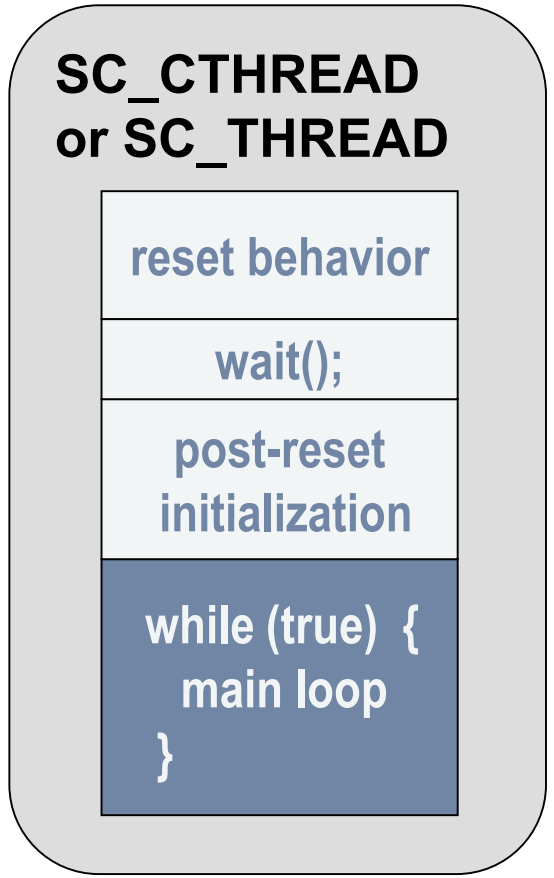


Note that every path through main loop must contain a wait() or simulation hangs with an infinite loop

SC_CTHREAD and SC_THREAD Reset Semantics

For Synthesis

- Assignments become reset initializations of registers in the hardware
  - Assignments to ports
  - Assignments to signals
  - Assignments to variables
- Initialization of data members of modules
  - Includes ports, signals, and data members
  - Should be done in reset behavior of some process
  - Should *NOT* be done in module constructor
    - This invites a mismatch between behavior and RTL reset functionality



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

SystemC Processes for Synthesis

SC_CTHREAD

- Clock-synchronous thread process
- Must have clock and reset specification
- Can have wait()s to span clock cycles
- Implemented in RTL as an FSM

SC_METHOD

- For implementing RTL constructs
- Semantics are same as Verilog always block
- Can be synchronous or asynchronous

SC_THREAD

- Equivalent in synthesis to SC_CTHREAD
- Only synthesizable if constrained like SC_CTHREAD
  - Sensitive to clock and reset
  - Only wait()s are to the sensitive clock edge

C++ Datatypes for Synthesis

- All C++ integer types are supported except wchar_t
- Synthesis standard refinements over ISOC++
  - Twos complement signed representation
  - Specific bit widths

Type	Width
(un)signed char, char	8
(un)signed short	16
(un)signed int	32
(un)signed long	32
(un)signed long long	64

Note that specification of narrower bit widths using SystemC datatypes can significantly reduce hardware cost after synthesis

SystemC Datatypes

- sc_int, sc_uint
  - Limited precision signed and unsigned integers with widths from 1 to 64
- sc_bigint, sc_biguint
  - Finite precision signed and unsigned integers with width from 1 to unlimited
- sc_fixed, sc_ufixed
  - Finite precision fixed-point data with user selectable saturation and rounding
- sc_bv
  - Finite word-length bit vector without arithmetic support
- sc_lv
  - 4-state logic, but X and Z not supported for synthesis

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Thank you!

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC: Focusing on High-Level Synthesis and Functional Coverage for SystemC

PART 3
Lessons Learned – Intel's Experience

Bob Condon

Intel

accellera SYSTEMS INITIATIVE

Introduction

- Bob Condon – past 8 years at Intel – coach new HLS teams
- At Intel we use HLS in production for both algorithm dominated designs and control dominated designs
- We have many groups who have produced multiple generations of designs and have thoroughly integrated HLS as part of their default workflow
- Key benefit – faster time to market because
  - Find bugs sooner
  - Tolerate late breaking arch changes
- What have we learned about designs which have gone through several iterations

Power

- HLS tools have some ability to consider power when pipelining
- When targeting cell libraries with low leakage cells, designer intuition of a “good design” is sketchy – and using these cells is a bit like a technology change
- Key HLS benefit – rapidly generate multiple uarchs lets us evaluate design properties which HLS doesn’t explicitly address (e.g., static and dynamic power consumption)

Reuse Tests Across Flows

- When a test fails, who is wrong? The test, the DUT, the spec?...
- Goal – find as many failures as possible with the cheapest tests
- SystemC DUT tested with MATLAB vectors
  - Keep algo and implementation in synch
  - Flushes out functional and quantization bugs
- Some HLS models are fast enough to integrate directly in a VP flow
- For designs with well established interfaces, test the pre-HLS code with OVM/UVM testbench

Designs Evolve – Refactor

- Refactor – change a design to make it easier to debug, reuse, maintain without changing the functionality
- A good one-minute C++ test will find almost all functional bugs in an HLS design. Run on every clean compile.
- Refactorings
  - Templatizing datatypes, modules …
  - Separating control from algorithm
  - Adding debugging

Evolution of a Function

```
  / Closest to the original C code
OUT_T filt_calc_vO(sc_fixed<10,2>  d[4]){
  const sc_fixed<5,1> Coef[]= { 9.0/16, -1.0/16 };
  sc_fixed<14,2> dac = (Coef[0] * (d[1]+d[2])) +
     (Coef[1] * (d[0]+d[3]));
  return dac;
}
```

```
/ Matches the spec(with explict datapath sizing)
// But the multiple RND's and SAT's are expensive
template <typename OUT_T, typename IN_T>
OUT_T filt_calc_v1(IN_T d[4]) {
{
  const sc_fixed<5,1> C[] = { 9.0/16, -1.0/16 };
  sc_fixed<10,2,SC_RND,SC_SAT> t1 = d[1]+d[2];
  sc_fixed<10,2,SC_RND,SC_SAT> t2 = d[0]+d[3];
  sc_fixed<14,2,SC_RND,SC_SAT> t3 = t1 * C[0];
  sc_fixed<14,2,SC_RND,SC_SAT> t4 = t2 * C[1];
  sc_fixed<14,2,SC_RND,SC_SAT> t5 = t3 + t4;
  OUT_T dac = t5;
  return dac;
}
```

```
// Avoids the rnd until the end
template <typename OUT_T, typename IN_T>
OUT_T filt_calc_v2(IN_T d[4]) {
   const sc_fixed<5,1> C[] = { 9.0/16, -1.0/16 };
   sc_fixed<10,2> t1 = d[1]+d[2];
   sc_fixed<10,2> t2 = d[0]+d[3];
   sc_fixed<14,2> t3 = t1 * C[0];
   sc_fixed<14,2> t4 = t2 * C[1];
   sc_fixed<15,2> t5 =  t3 + t4;
   OUT_T dac = t5;
   return dac;
}
```

Evolution of a Function (*cont*)

- Refactored to 1-off test (of a subunit)
- Kept 3 variants of the code
  - Tradeoff between maintenance and triage
- Added unit tests for individual functions
- The first three vectors found all the functional bugs.

```
// Here is a small test harness to isolate the
core of the design and allow rapid experimentation
template <int VER>
SC_MODULE(x2_experiment) {
…
  typedef x2::input_t   i_t;
  typedef x2::dac_output_t o_t;

void process() {
 wait();
 while (true) {
   input_t d[4]; d[0]=d0.read(); d[1]=d1.read();
   d[2]=d2.read();d[3]=d3.read();
   switch (VER) {
   case 0: dac.write(filt_calc_v0(d); break;
   case 1: dac.write(filt_calc_v1<o_t,i_t>(d);
break;
   case 2: dac.write(filt_calc_v2<o_t,i_t>(d);
break;
   }
   wait();
 }
}
};
```

Refactor to Extract Common Control Idioms

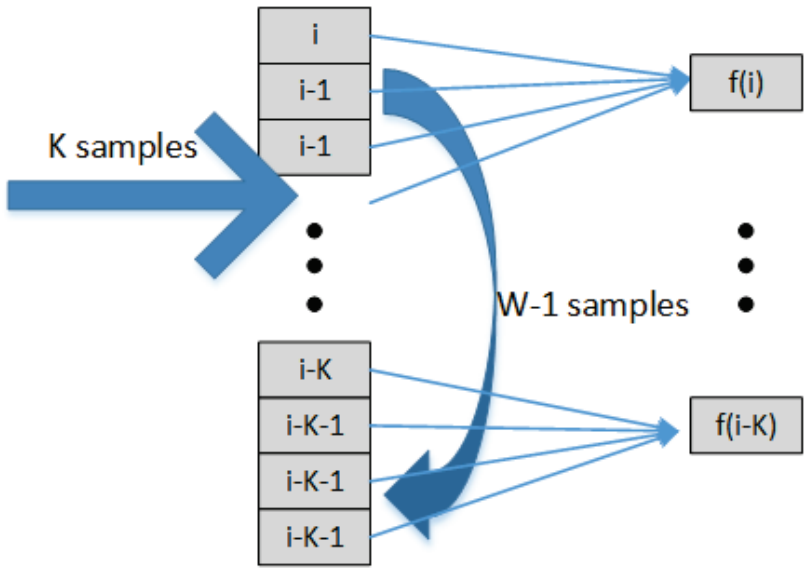
Many blocks will have the same control pattern.

A common idiom: calculate with a throughput of K outputs per clock:

result[i] = f(s[i-W], …s[i-1])

Let K = 8, W = 4

Implement with a shift register




```
template <typename T, int N, int K=1>
struct TD_Window {
  unsigned maxSample;    // debug – total samples
  T d[N];
  void reset() {
    maxSample=0;
    for (size_t i=K; i<N; ++i)   //Add HLS pragmas here
      d[i] = 0;
  }
  T operator[](int indx) const {
    sc_assert(size_t(indx) <N);
    return d[indx];
  }
  void shift_in(const T t[K]) {
    maxSample += K;
    for( size_t i=0; i<N-K; i++)
      d[i] = d[i+K]; // Shift the old
    for( size_t i=0; i<K; i++)
      d[i + (N-K)] = t[i]; // ... and read in the new
  }
};
```

Evolution of a Function (*cont*)

Algo code still uses [] – but now it is from the window class.

HLS can optimize the arrays and the functions together (so different than sharing a module).

Any debugging, logging gets shared across all users.

```
SC_MODULE(x2_experiment) {
…
  typedef x2::input_t   i_t;
  typedef x2::dac_output_t o_t;
  typedef TD_Window<i_t, 4, 8> win_t;
  win_t din;

void process() {
 din.reset();
 wait();
 while (true) {
   input_t d[K]; read_inputs(d);
   din.shift_in(d);
   dac.write(filt_calc_v2<o_t,win_t>(din));
   }
   wait();
 }
}
};
template <typename OUT_T, typename win_t>
OUT_T filt_calc_v2(win_t win) {…
    t = coef[0] * win[0]; …
}
```

Repurpose C++ Tools

- Eclipse with extensions for SystemC datatypes
  - Our code has lots of templates and the IDE helps new coders get up to speed on the codebase
- gtest for regression testing of C++ libraries
- Boost command line argument parsing
- Boost metaprogramming for iteration over the repetive parts

Recap

- Rapid generation of different RTL implementation allows power exploration
- Reuse every test you can
  - From the architectural/functional model
  - From the RTL turnin model
- Refactor to make code used in more circumstance and easier to debug
- Separate datapath from control to make each piece reusable with other models
- Keep an eye on what you can steal from the C++ software engineering world

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Thank you!

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC: Focusing on High-Level Synthesis and Functional Coverage for SystemC

PART 4

Functional Coverage For SystemC (FC4SC)

Dragoș Dospinescu
AMIQ

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Agenda

1. Motivation
2. What is functional coverage?
3. What is FC4SC?
4. FC4SC features overview
5. Coverage constructs
6. Coverage control
7. Coverage database management
8. Conclusions
9. Roadmap

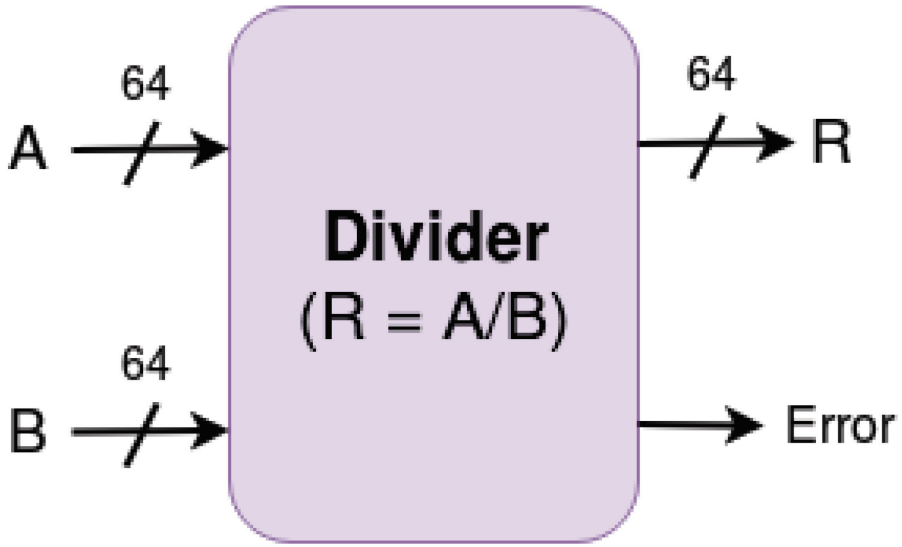
accellera SYSTEMS INITIATIVE

Motivation

- Implement constrained-random testbenches for verification
- Measure the degree of randomization in the test suite
- Define milestones based on coverage metrics
- Track verification progress during the development cycle
- Generate reports on what functionality was tested

What Is Functional Coverage? (1)

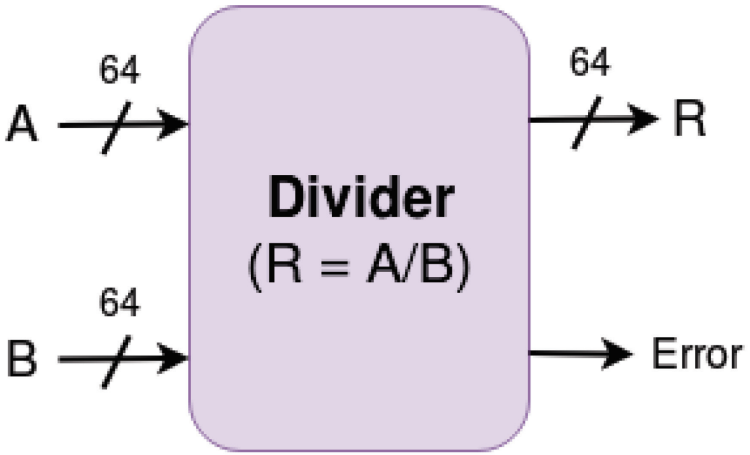
- User defined metric used in constrained-random verification
- Records what “happens” during test execution
- Qualitative metric relative to functionality aspects of the model



Two 64-bit inputs ⇒ 2^{128} possibilities.

Impossible to verify exhaustively!

What Is Functional Coverage? (2)



The functional coverage approach:

- Interesting values for A & B
  - 0, 1, MIN, MAX
  - some values in [MIN:MAX]
- Relationship between A & B
  - parity
  - sign

...

Coverage is based on the model's features!

What Is FC4SC?

- C++11 header only library
- No dependency on any 3rd party library
- Provides functional coverage capabilities
- Based on the *IEEE 1800 - 2012 SystemVerilog Standard*

- Download library: *https://github.com/amiq-consulting/fc4sc*
- Include it in your project: #include "fc4sc.hpp"
- Ready to use!

FC4SC Features Overview

- Coverage definition: bin, coverpoint, cross, covergroup
- Coverage control: options, sample disabling
- Runtime coverage interrogation
- Coverage database saving
- Coverage database management tools

Coverage Constructs: bin (1)

Bin: collection of values and intervals

FC4SC

```
bin<int>("less_than_8", // bin name
   1,                   // 1
   interval(2, 3),      // [2:3]
   interval(7, 5)       // [5:7]
);
bin_array<int>("split",
   3,                   // 3 bins
   interval(0, 255)     // [0:255]
);
illegal_bin<int>("illegal_10", 10);
ignore_bin<int>("ignore_100", 100);
```

SystemVerilog

```
bins less_than_8 = {
    1,
    [2:3],
    [5:7]
};
bins split[3] = {
    [0:255]
};
illegal_bins illegal_10 = {10};
ignore_bins ignore_100 = {100};
```

Coverage Constructs: bin (2)

```
auto fibonacci = [](size_t N) -> std::vector<int> {
  int f0 = 1, f1 = 2; // initialize starting number
  std::vector<int> result(N, f0);
  // calculate following fibonacci numbers
  for (size_t i = 1; i < N; i++) {
      std::swap(f0, f1);
      result[i] = f0;
      f1 += f0;
  }
  return result;
};
COVERPOINT(int, bin_array_cvp, value) {
      bin_array<int>("fibonacci", fibonacci(5))
};
```

```
bin<int>("fibonacci[0]", 1),
bin<int>("fibonacci[1]", 2),
bin<int>("fibonacci[2]", 3),
bin<int>("fibonacci[3]", 5),
bin<int>("fibonacci[4]", 8)
```

Coverage Constructs: coverpoint (1)

- Contains bins with data of interest
- Handles sampling
- ignore_bin → illegal_bin → bin

FC4SC

```
COVERPOINT(int, datacp, data)
{
  bin<int>("positive", interval(0, 10)),
  bin<int>("negative", interval(-10,0)),
  illegal_bin<int>("illegal_zero", 0)
};
```

SystemVerilog

```
datacp : coverpoint data
{
  bins positive = {[0:10]};
  bins negative = {[-10:0]};
  illegal_bins illegal_zero = {0};
}
```

Coverage Constructs: coverpoint (2)

FC4SC

```
COVERPOINT(int, datacp, data*2, flag!=0)
{
    // ...
};
```

SystemVerilog

```
datacp: coverpoint (data*2) iff (flag!=0)
{
    // ...
}
```

Sample expression

Sample condition

Both are evaluated at the point of sampling (dynamically)!

Coverage Constructs: cross

- Cartesian product of coverpoints' bins
- Behaves the same as a coverpoint in all regards

FC4SC

```
COVERPOINT(int, cvp1, data1) {
  bin<int>("zero", 0),
  bin<int>("positive", 1, 2, 3)
};
COVERPOINT(int, cvp2, data2) {
  bin<int>("zero", 0),
  bin<int>("negative", -1, -2, -3)
};
auto cvp1_x_cvp2 = cross<int,int>(
  "cvp1_x_cvp2", &cvp1, &cvp2);
```

SystemVerilog

```
cvp1 : coverpoint data1 {
  bins zero = {0};
  bins positive = {1, 2, 3};
}
cvp2 : coverpoint data2 {
  bins zero = {0};
  bins negative = {-1, -2, -3};
}
cvp1_x_cvp2 : cross cvp1, cvp2;
```

Coverage Constructs: covergroup

- Ties together all coverage constructs
- Dispatches sampling data to coverpoints and crosses

FC4SC

```
class cvg_ex: public covergroup {
public:
  int data;
  COVERPOINT(int, cvp1, data) {
    bin<int>("zero", 0),
    bin<int>("positive", 1, 2, 3)
  };
  CG_CONS(cvg_ex) {/*constructor*/}
};
```

SystemVerilog

```
covergroup cvg_ex {
  cvp1 : coverpoint data {
    bins zero = {0};
    bins positive = {1, 2, 3};
  }
}
```

Coverage Control (1)

- Options
  - adjusting coverage distributions: *weight*
  - setting coverage goals: *goal*, *at_least*
- Sample enable/disable
  - *starting* and *stopping* coverage collection
- Coverage interrogation (at runtime)
  - getting coverage percentage (per type/instance)
  - getting the number of hits
- Usable on: covergroup, coverpoint, cross

Coverage Control (2)

```
class cvg_ex: public covergroup
{
public:
  int data;
  CG_CONS(cvg_ex, int w = 100) {
    this->option.weight = w;
  }
  COVERPOINT(int, cp1, data) {
    bin<int>("zero", 0),
    bin<int>("positive", 1, 2, 3)
  };
};
```

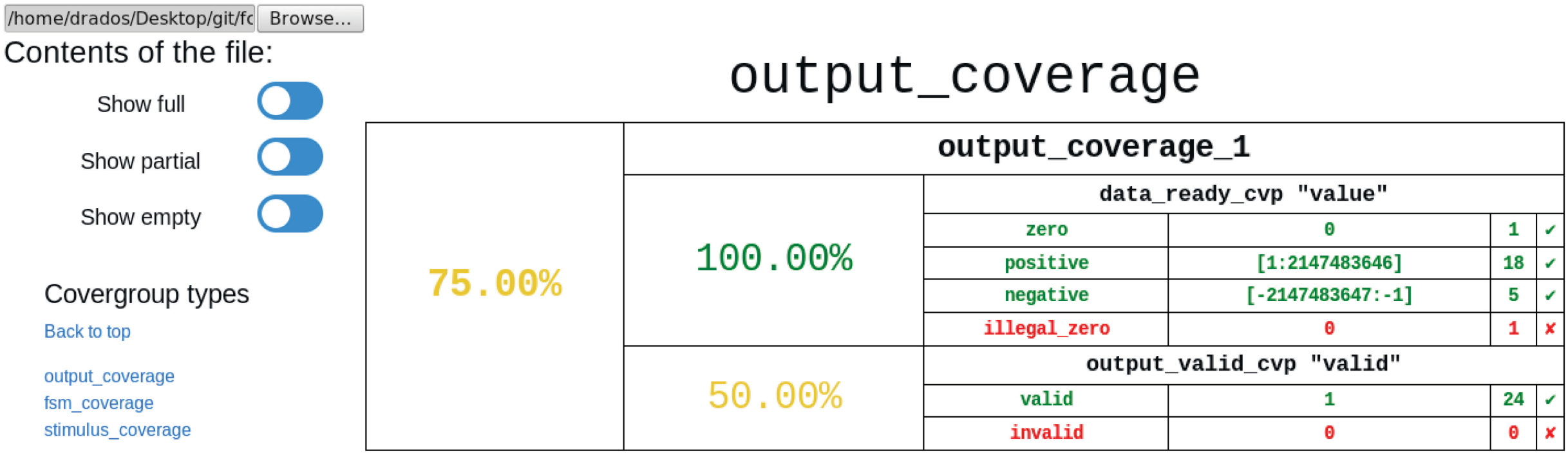
Usage Example

```
cvg_ex cvg; // instantiate covergroup
cvg.data = 0; // set data on 0
cvg.sample(); // sample
// expect 50% covered
EXPECT_EQ(cvg.get_inst_coverage(),50);


cvg.stop();     // stop sampling
cvg.data = 2;
cvg.sample();
// still 50% covered
EXPECT_EQ(cvg.get_inst_coverage(),50);
```

Coverage Database Management: Visualization

JavaScript app: *fc4sc/tools/gui/index.html*



Coverage Database Management: Creation

- Generate coverage database:

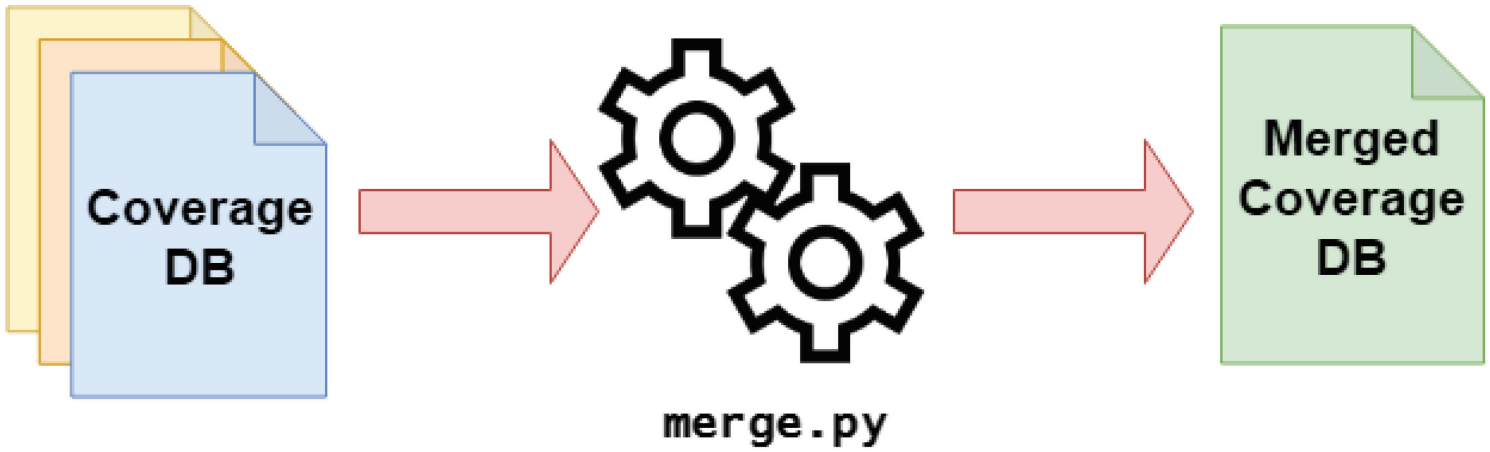
```
fc4sc::global::coverage_save("coverage_db_name.xml");
```

- Databases can be generated at any point during runtime!

- Writes to XML file:
  - Complete coverage model
  - All coverage options
  - Number of hits for each bin

Coverage Database Management: Merging

Merge = aggregate the coverage data from different executions



$> *python merge.py /path/to/top/directory merged_coverage_db.xml*

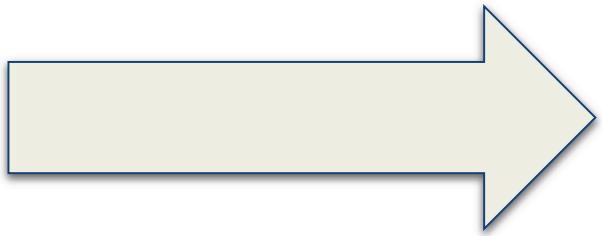
Coverage Database Management: Reporting

$> python report.py

--xml_report input_db.xml

--yaml_out report.yaml

--report_missing_bins



```yaml
modules:
  shift_coverage:
    instances:
      shift_coverage_1:
        inst_data:
          shift_cvp:
            bin_count: 2
            bin_hits: 2
            bin_misses: 0
            misses: []
            pct_cov: 100.0
            weight: 1
        pct_cov: 100.0
        weight: 1
      shift_coverage_2:
        inst_data:
          shift_cvp:
            bin_count: 2
            bin_hits: 2
            bin_misses: 0
            misses: []
            pct_cov: 100.0
            weight: 1
        pct_cov: 100.0
        weight: 1
    pct_cov: 100.0
```

Special thanks to: Armond Paiva <apaiva@tenstorrent.com>

Conclusions

FC4SC:

- Brings the functional coverage from SV domain to SystemC domain
- Provides a qualitative metric of the functionality of a SystemC model
- Introduces coverage-driven verification as an alternative to test-driven verification
- Allows an easy transition from SV syntax
- Is easy to integrate into a regression flow

Roadmap

- Default bins
- SystemC integration:
  - Support for coverage over custom data types
  - Event-based sampling
- Cross bin filtering: with keyword
- Cross definition: binsof, intersect
- Transition coverage

References

- *FC4SC github repository*
- *IEEE 1800 - 2012 SystemVerilog Standard*
- Singhal M. (2015, June 4). *What is functional coverage*
- (2013, April 20). *Why you need functional coverage*. Retrieved from SynthWorks Blog
- Marriott, P. (2006, September 1). *The 'What', 'When', and 'How Much' of functional coverage*
- INF5430 - *SystemVerilog for Verification, Ch. 9 Functional Coverage*

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Thank you!

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC: Focusing on High-Level Synthesis and Functional Coverage for SystemC

PART 5
Accellera SystemC Working Groups Update

Martin Barnasconi
Accellera Technical Committee Chair

accellera SYSTEMS INITIATIVE

SystemC Synthesis & Datatypes WG

- SystemC Synthesis Subset Language Reference Manual version 1.4.7 (2016) available on Accellera website
  - https://accellera.org/downloads/standards/systemc
- Ongoing discussion to enhance datatypes
  - Different contributions submitted to Accellera
  - Exploring standardization and implementation w.r.t. language, API and performance
- Enhancements for high-level synthesis under discussion
  - E.g., Benefit from modern language constructs in C++1

SystemC Verification Working Group

- UVM-SystemC reference implementation 1.0beta2 released for public review in November 2018
  - Current development focusing on completion of registration abstraction layer

- Next step: Introduce Constrained Randomization capabilities by using CRAVE as add-on library

Accellera SystemC Working Groups

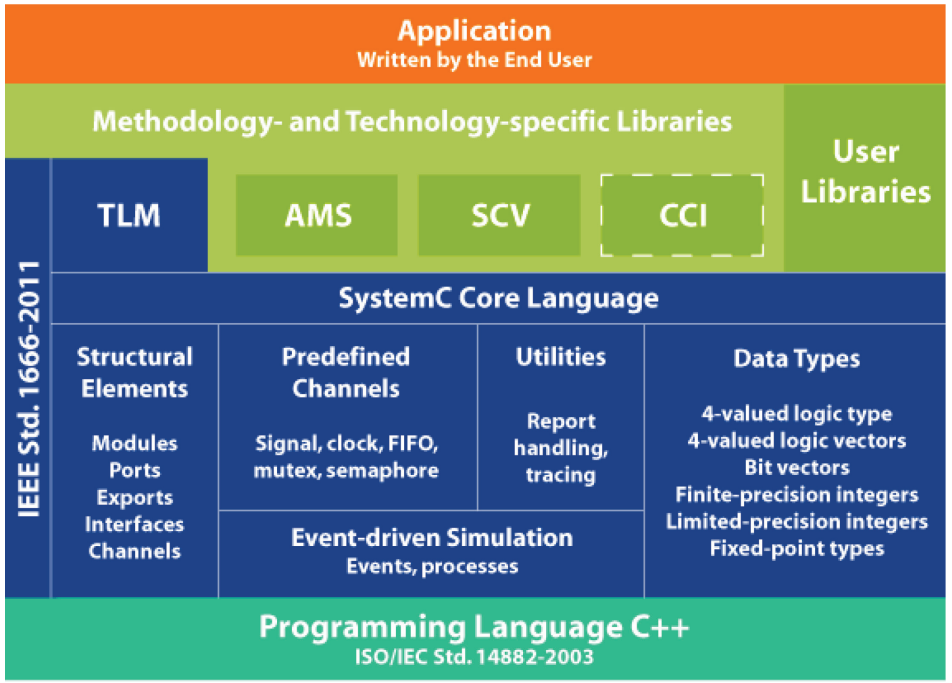
- **Language** Working Group (LWG)
- **Transaction-Level Modeling** Working Group (TLMWG)
- **Analog/Mixed-Signal** Working Group (AMSWG)
- **Configuration, Control & Inspection** Working Group (CCIWG)
- **Synthesis** Working Group (SWG)
- **Datatypes** Working Group (SDTWG)
- **Verification** Working Group (VWG)

SystemC Evolution Day

- Successful SystemC Evolution Day held at DVCon Europe October 2018
  - Interactive workshop to discuss evolution of SystemC standards to advance the SystemC eco-system
  - Topics discussed: AMS, CCI, TLM-serial, Multi-language
  - Presentation material available https://accellera.org/news/events/systemc-evolution-day-2018
- SystemC Evolution Day 2019 planned on **October 31, 2019**
  - Call for contributions will open soon, more information: https://accellera.org/news/events/systemc-evolution-day-2019

SystemC Community & Forum

- Join the vibrant SystemC Community!
- Accellera SystemC Community pages https://accellera.org/community/systemc/about-systemc
- Accellera SystemC Discussion Forums http://forums.accellera.org/forum/9-systemc/
- Or join any of the Accellera SystemC Working Groups!



SystemC Language + TLM WG

- SystemC Reference Implementation version 2.3.3 released in Nov 2018
- LWG is preparing contribution to IEEE P1666
- TLM-CAN contribution from Bosch + ST Microelectronics
  - Discussion standard to explore the need for TLM standardization for other serial protocols

SystemC Analog/Mixed Signal

- SystemC AMS User's Guide
  - Update to make it compatible with IEEE 1666.1 standard
  - Detailed documentation on dynamic TDF features
  - Release expected in Q2 2019

- Development and release of SystemC AMS regression suite
  - Containing many basic and application examples
  - Release expected 2H 2019

SystemC Configuration Control and Inspection

- CCI 1.0.0 released in June 2018, covering Configurability of SystemC models
- CCI Community forum is in place
- Language Reference Manual and supplemental material available
  - Overview tutorial, Reference implementation and 20+ examples
  - Key features
    - Portable information exchange
    - Preloading configuration info
    - Value callbacks & traceability
    - Parameters
    - Parameter callbacks
    - User-defined value types supported
    - Architected for seamless integration of existing configuration solutions
- More information and download: http://accellera.org/activities/working-groups/systemc-cci
- Next: SystemC Checkpointing

IEEE-related SystemC Working Groups

- P1666 (SystemC)
  - IEEE Standard for Standard SystemC Language Reference Manual Working Group (LWG)
  - Latest version: IEEE 1666-2011, published 2012-01-09
  - Chair: Jerome Cornet (ST Microelectronics)
  - PAR approved, P1666 WG started end of 2018
  - **Call for Participation: Please contact Jerome Cornet (chair) or Jonathan Goldberg (IEEE) how to join**

- P1666.1 (SystemC-AMS)
  - IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual
  - Latest version: IEEE 1666.1-2016, Published 2016-04-06
  - Chair: Martin Barnasconi (NXP)
  - P1666.1 WG not active at the moment

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Thank you!

accellera SYSTEMS INITIATIVE