Cut Your Design Time in Half with Higher Abstraction

Organizer: Adam Sherer – Accellera Systems Initiative

Speakers: Frederic Doucet - Qualcomm

Mike Meredith – Cadence Design Systems, Inc.

Peter Frey – Mentor Graphics Corp.

Bob Condon – Intel Corp.

Dirk Seynhaeve – Intel Corp.

Agenda

- Introduction – How High-Level Synthesis (HLS) works targeted for hardware designers
- The Proposed Accellera SystemC Synthesizable Subset
- High-Level Synthesis and Verification
- HLS in the Wild – Intel Experience
- HLS for the FPGA/Programmable Market
- SystemC Synthesis Standard: Which Topics for the Next Round?

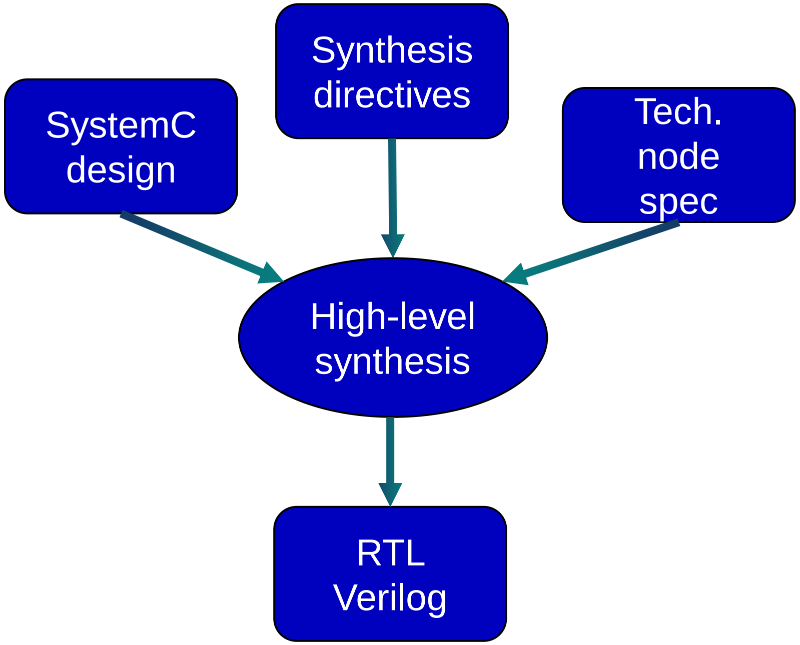
How High-level Synthesis Works:
An Intro for Hardware Designers

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



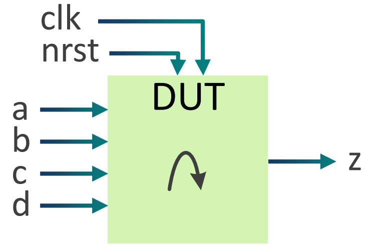
High-level Synthesis

- 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++

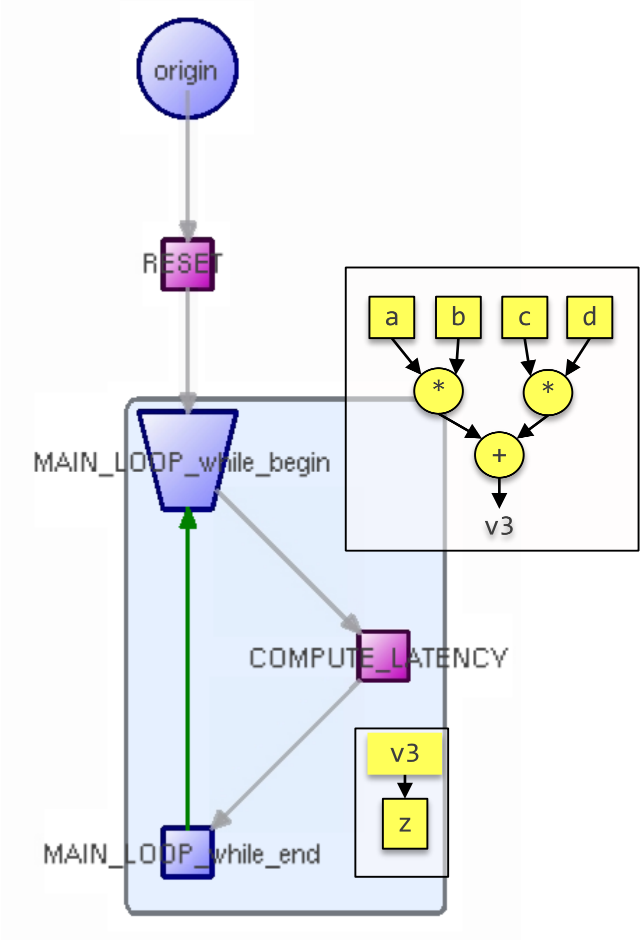
- SystemC: syntax for hardware modeling framework in C++
  - Modules
  - Ports
  - Connections
  - Processes



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

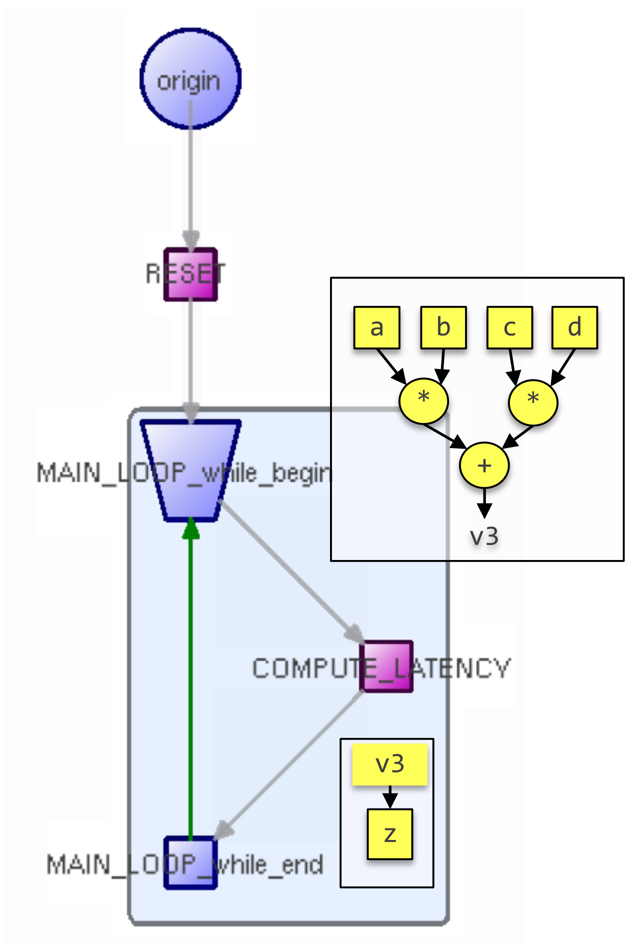
Example: Synthesizable SystemC

```
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;
    }
  }
};
```



DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES 2016

Example: High-level synthesis



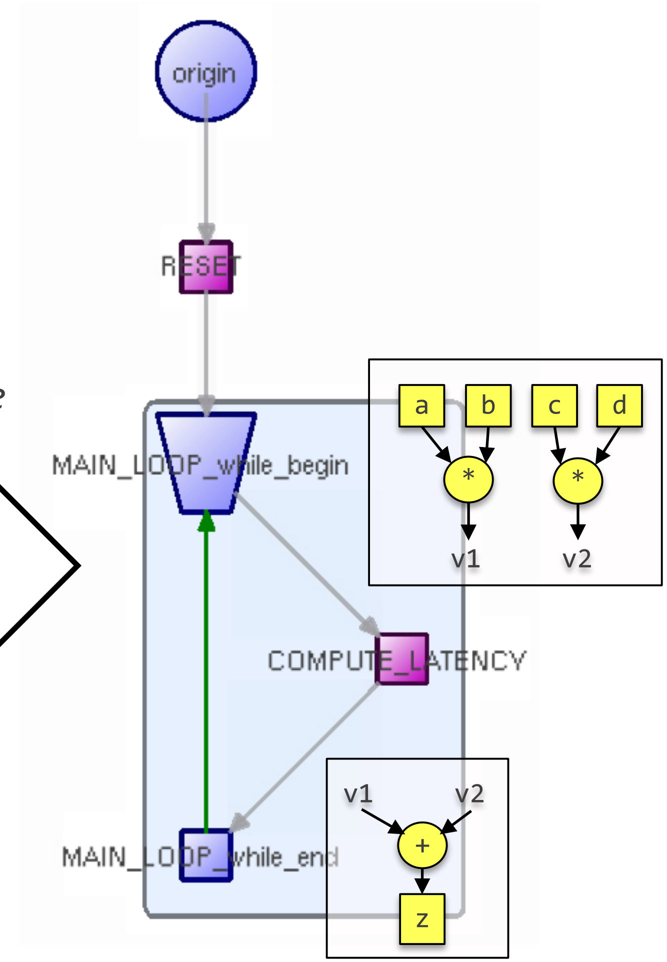
Synthesis directives:

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

Scheduling/resource allocation/binding

Op delays:

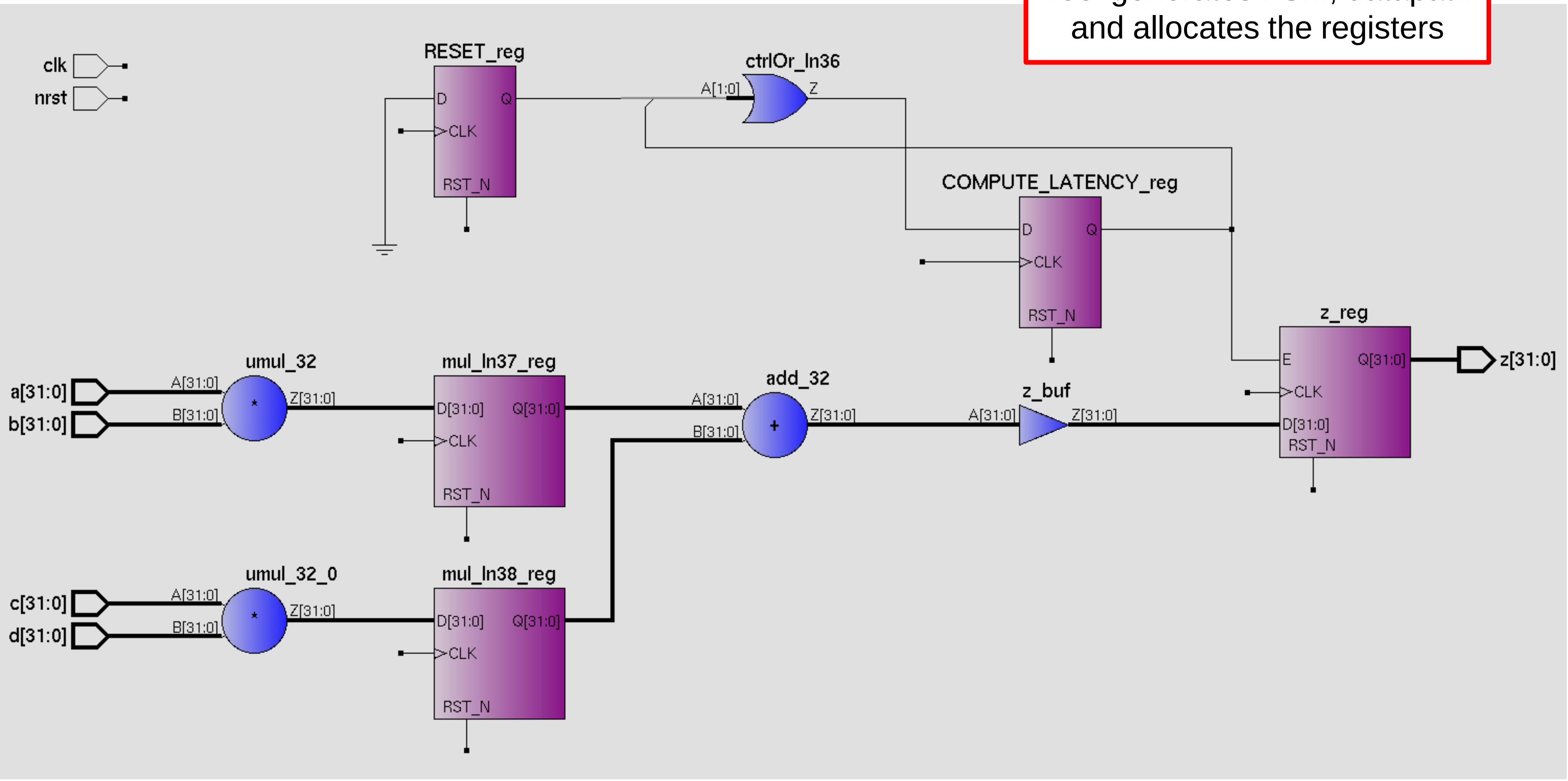
- mul: 4ns
- add: 2ns



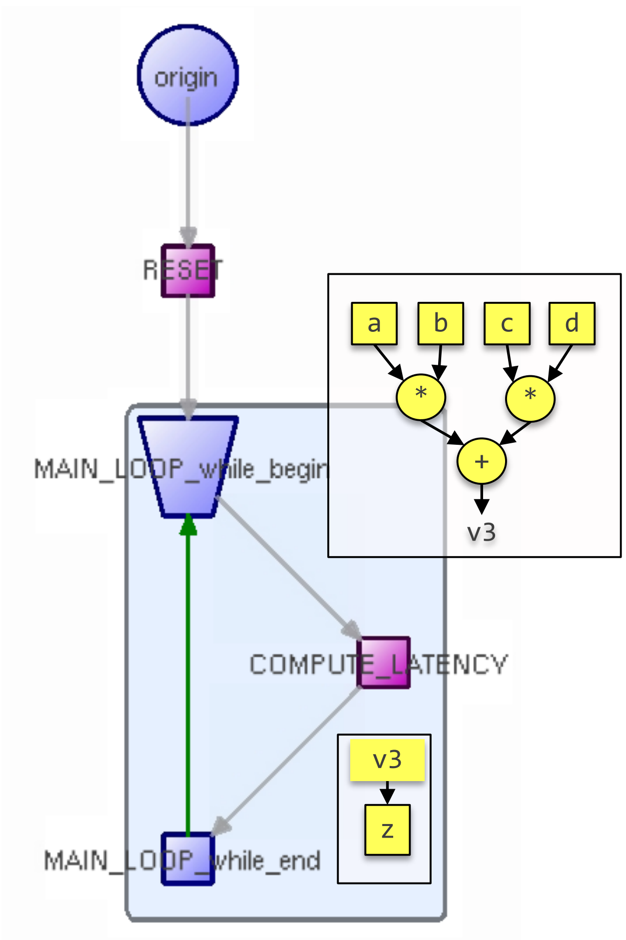
Scheduler moved the addition across the state to get positive slack

Example: High-level synthesis

Tool generates FSM, datapath and allocates the registers



Example: High-level synthesis, second run



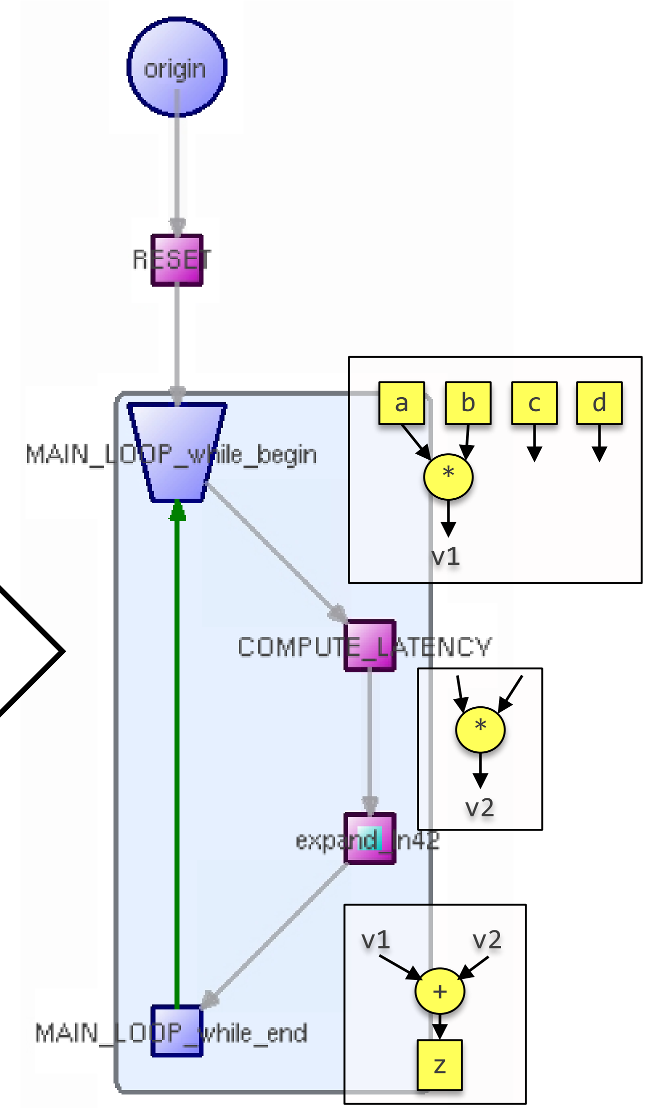
Synthesis directives:

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

Scheduling/resource allocation/binding

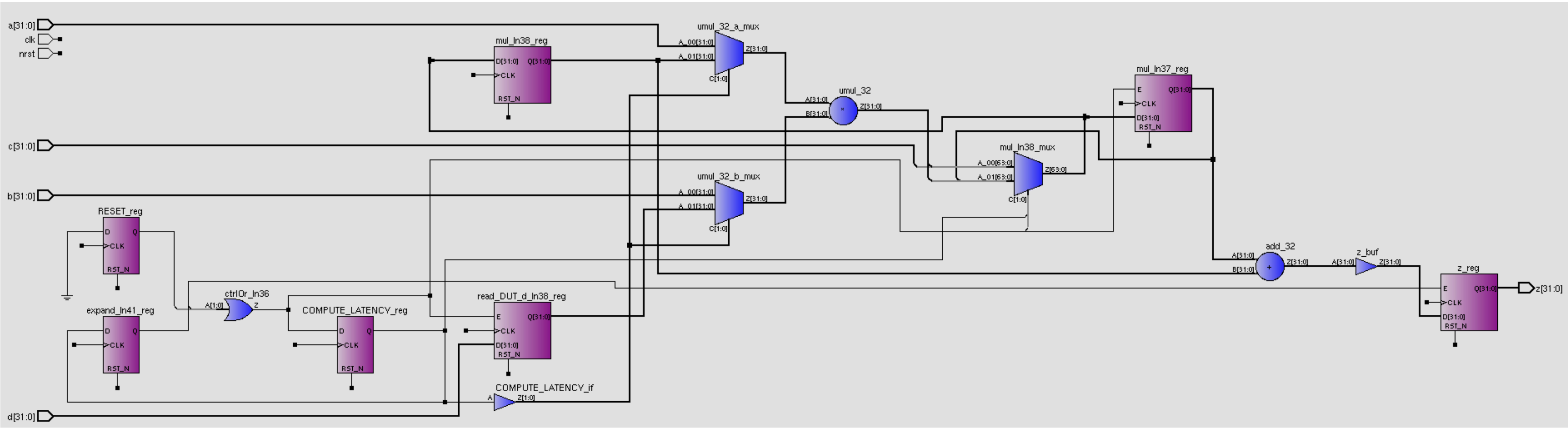
Op delays:

- 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

Thank You!

The Proposed Accellera SystemC Synthesizable Subset

Mike Meredith

Vice Chair – Accellera Synthesis Working Group

Cadence Design Systems

SystemC Synthesizable Subset Work

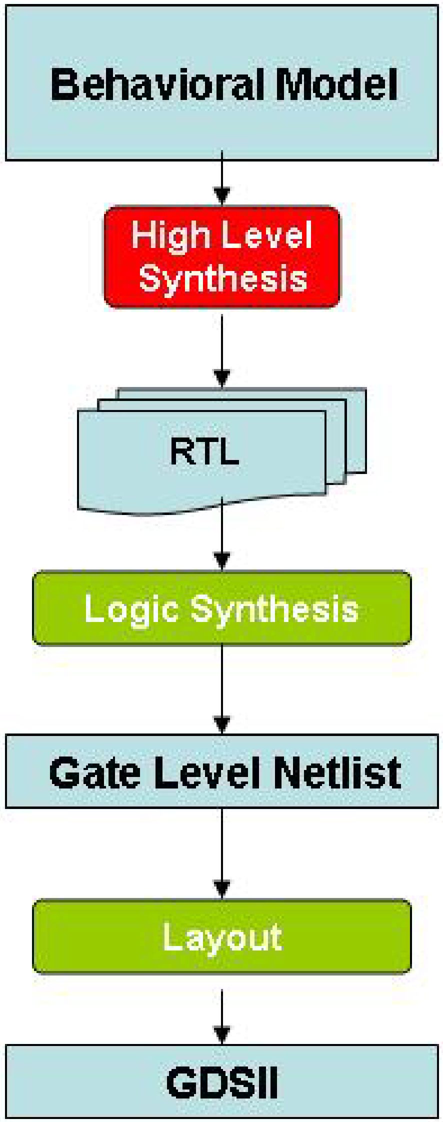
- Development of a description of a synthesizable subset of SystemC
- Started in the OSCI Synthesis Working Group
- Current work is in Accellera Systems Initiative Synthesis Working Group
- Draft has been proposed for approval as a new standard
- Many contributors over a number of years
- Broadcom, Cadence, 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 – e.g., templates

Scope of the Proposed 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 Planned Standard

SystemC Elements

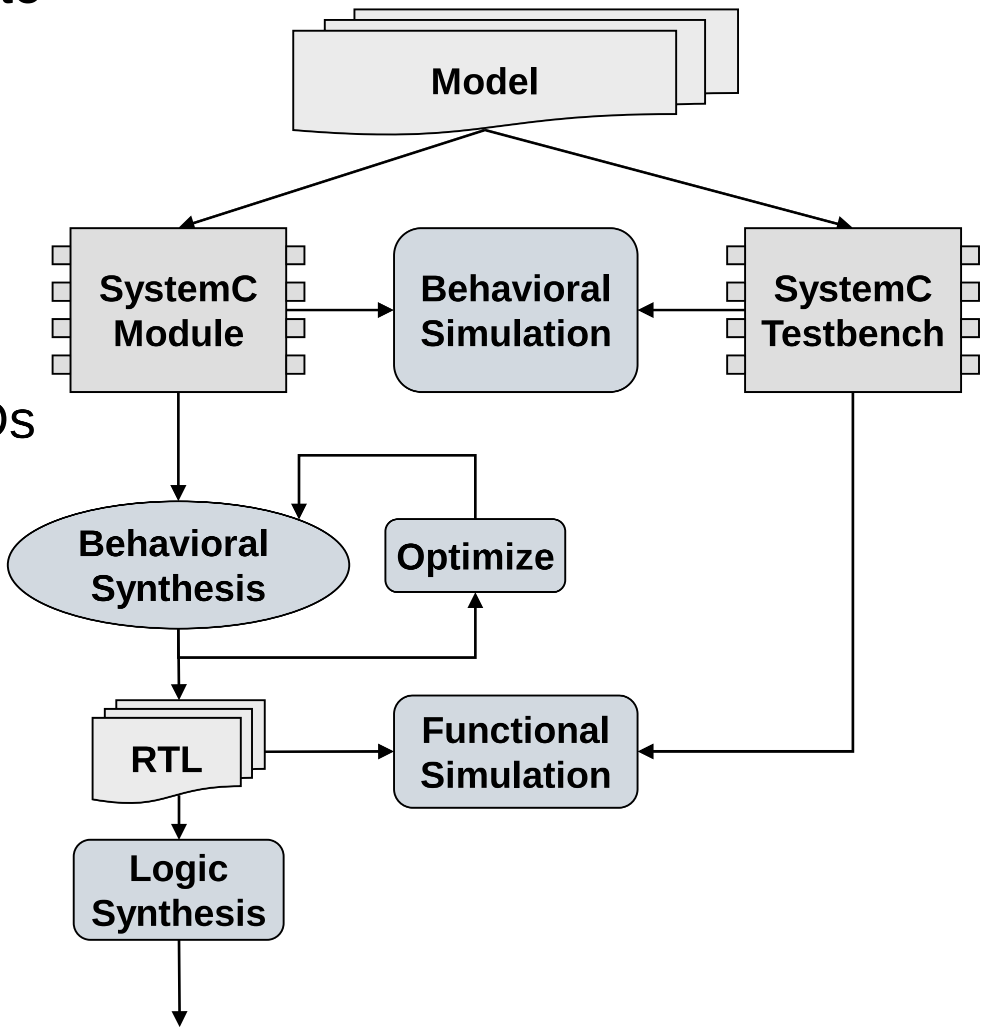
- Modules
- Processes
  - SC_CTHREAD
  - SC_THREAD
  - SC_METHOD
- Reset
- Signals, ports, exports
- SystemC datatypes

C++ Elements

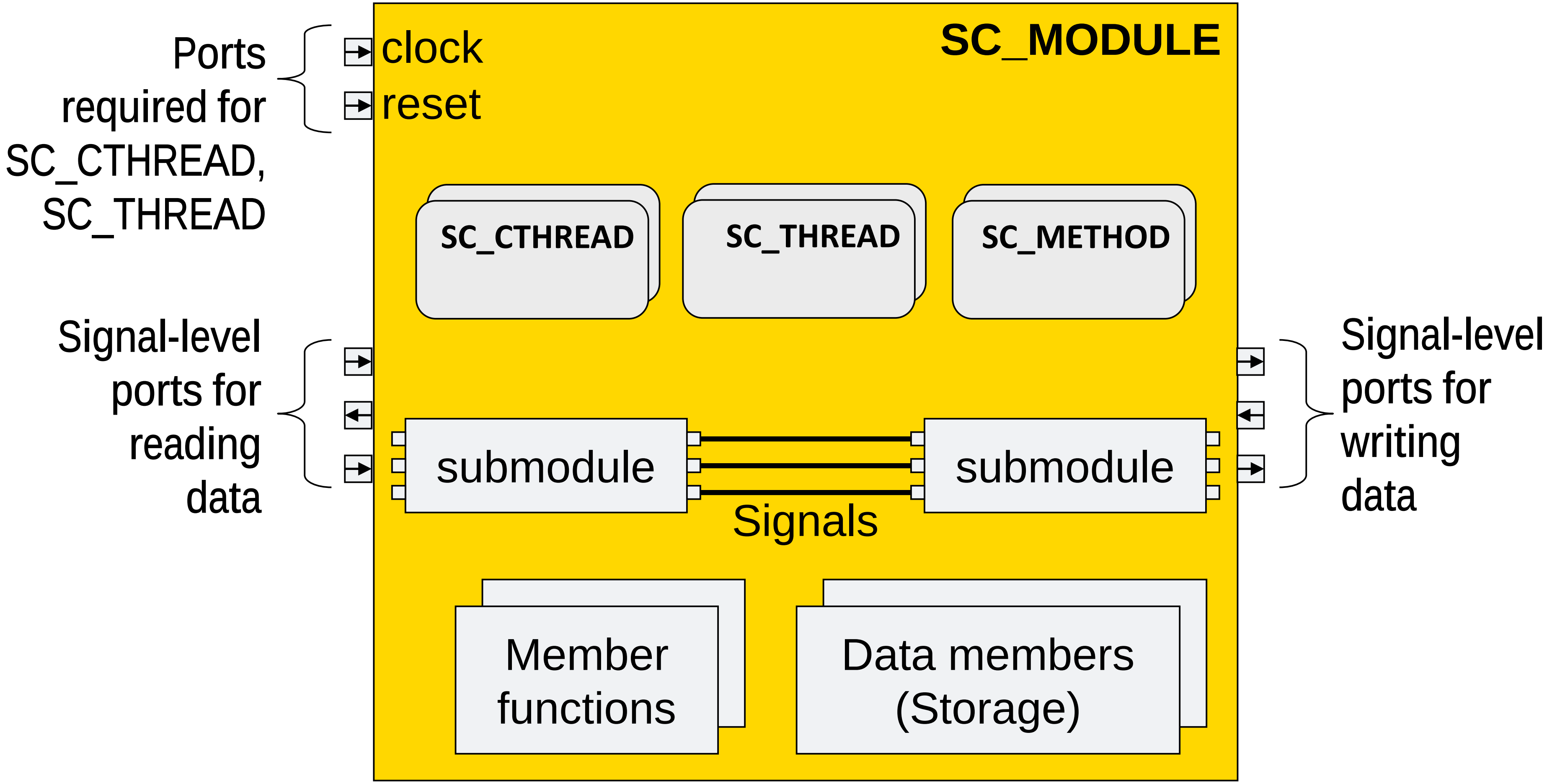
- C++ datatypes
- Expressions
- Functions
- Statements
- Namespaces
- Classes
- Overloading
- Templates

Behavioral Synthesis in the Design Flow

- Design and testbench converted to SystemC modules or threads
- Design
  - Insertion of signal-level interfaces
  - Insertion of reset behavior
  - Conversion to SC_CTHREADs
- Testbench
  - Insertion of signal-level interfaces
  - Reused at each abstraction level
    - Behavioral
    - RTL
    - Gate



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

```
// 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)
 {…}
};
```

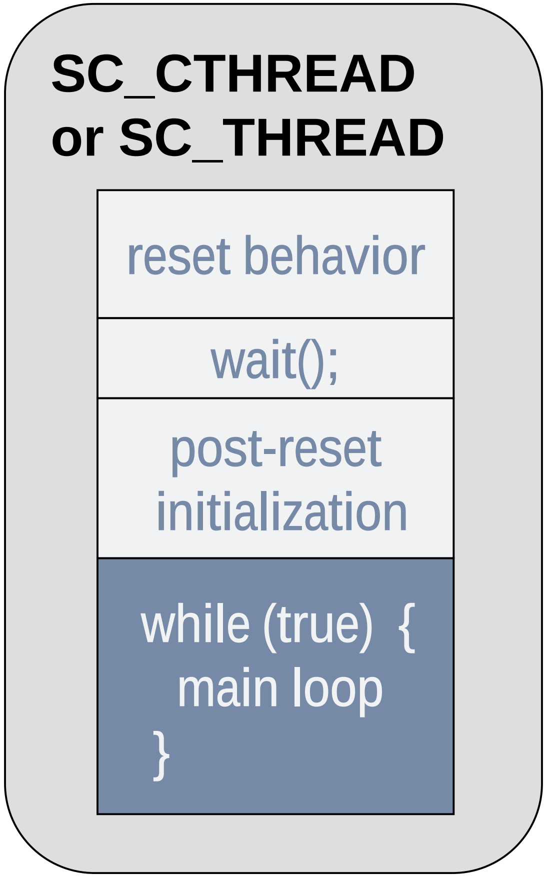
Derived Modules

- Derived modules OK

```
SC_MODULE( BaseModule ) {
  sc_in< bool > reset;
  sc_in_clk clock;
  BaseModule ( const sc_module_name name )
   : sc_module( name_ ) {
  }
};

class DerivedModule : public BaseModule {
  void newProcess();
  SC_HAS_PROCESS( DerivedModule );
  DerivedModule( sc_module_name name_ )
   : BaseModule( name_ ) {
   SC_CTHREAD( newProcess, clock.pos() );
   reset_signal_is( reset, true );
  }
};
```

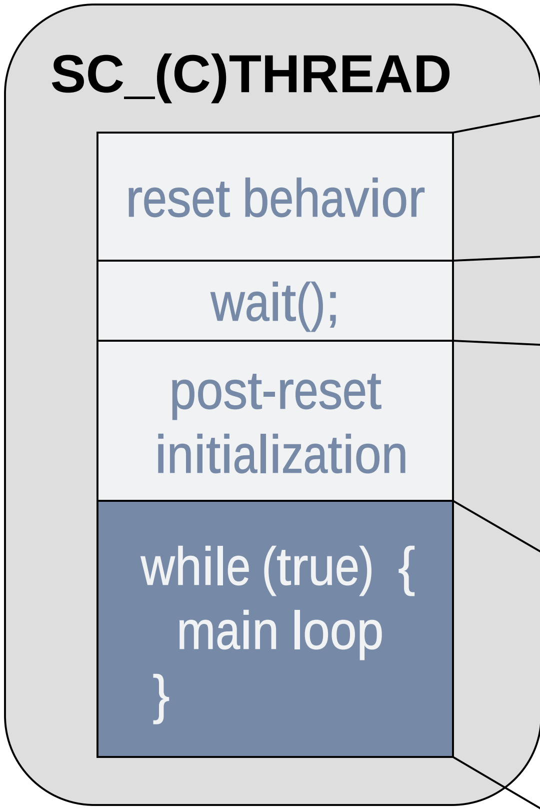
SC_THREAD & SC_CTHREAD Reset Semantics

- 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();
 }
}
```

Process Structure Options

- SC_THREAD and SC_CTHREAD processes must follow one of the forms shown
- Note that there must be a wait() in every path of the infinite loops to avoid simulator hangup

```
while( 1 )
{ }

while( true )
{ }

do { }
while ( 1 );

do { }
while ( true );

for ( ; ; )
{ }
```

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())

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

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

Supported SystemC integer functions					
bit select []	part select (i,j)	concatenate (,)			
to_int()	to_long()	to_int64()	to_uint()	to_uint64()	to_ulong()
iszero()	sign()	bit()	range()	length()	
reverse()	test()	set()	clear()	invert()	

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
    - e.g., 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

Thank You!

High-Level Synthesis and 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

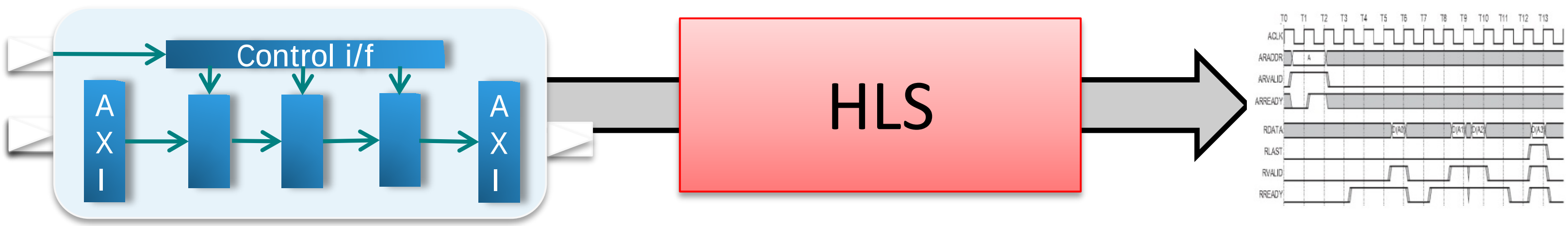
High-Level Synthesis

High-Level Synthesis

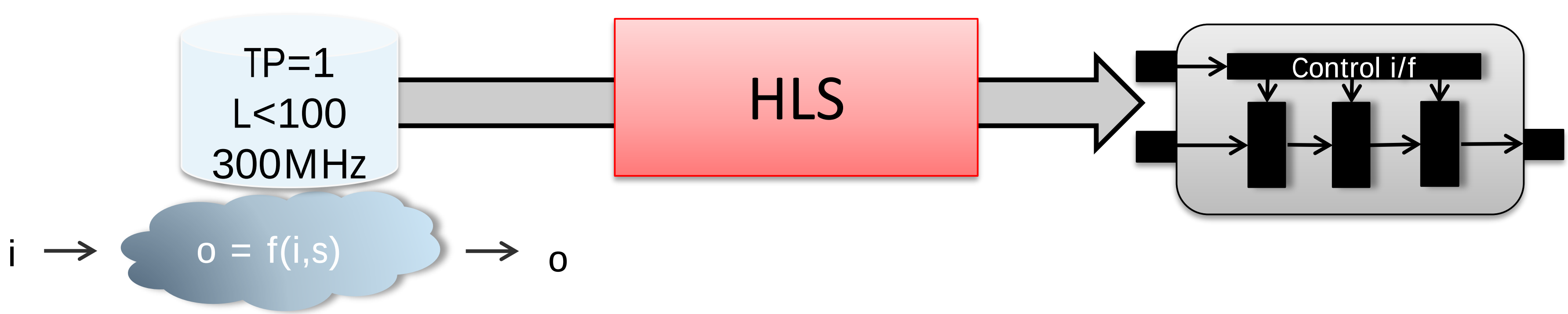
- Synthesizes “Accellera SystemC Synthesizable Subset” to production-quality RTL
- Arithmetic optimizations and bit-width trimming
- User control over the micro-architecture implementation
  - Parallelism, Throughput, Area, Latency (loop unrolling & pipelining)
  - Memories (DPRAM/SPRAM/split/bank) vs. Registers (Resource allocation)
- Multi-objective scheduling
  - Power, Performance, Area
- Hardware exploration is accomplished by applying different constraints

Properties of High-Level Synthesis?

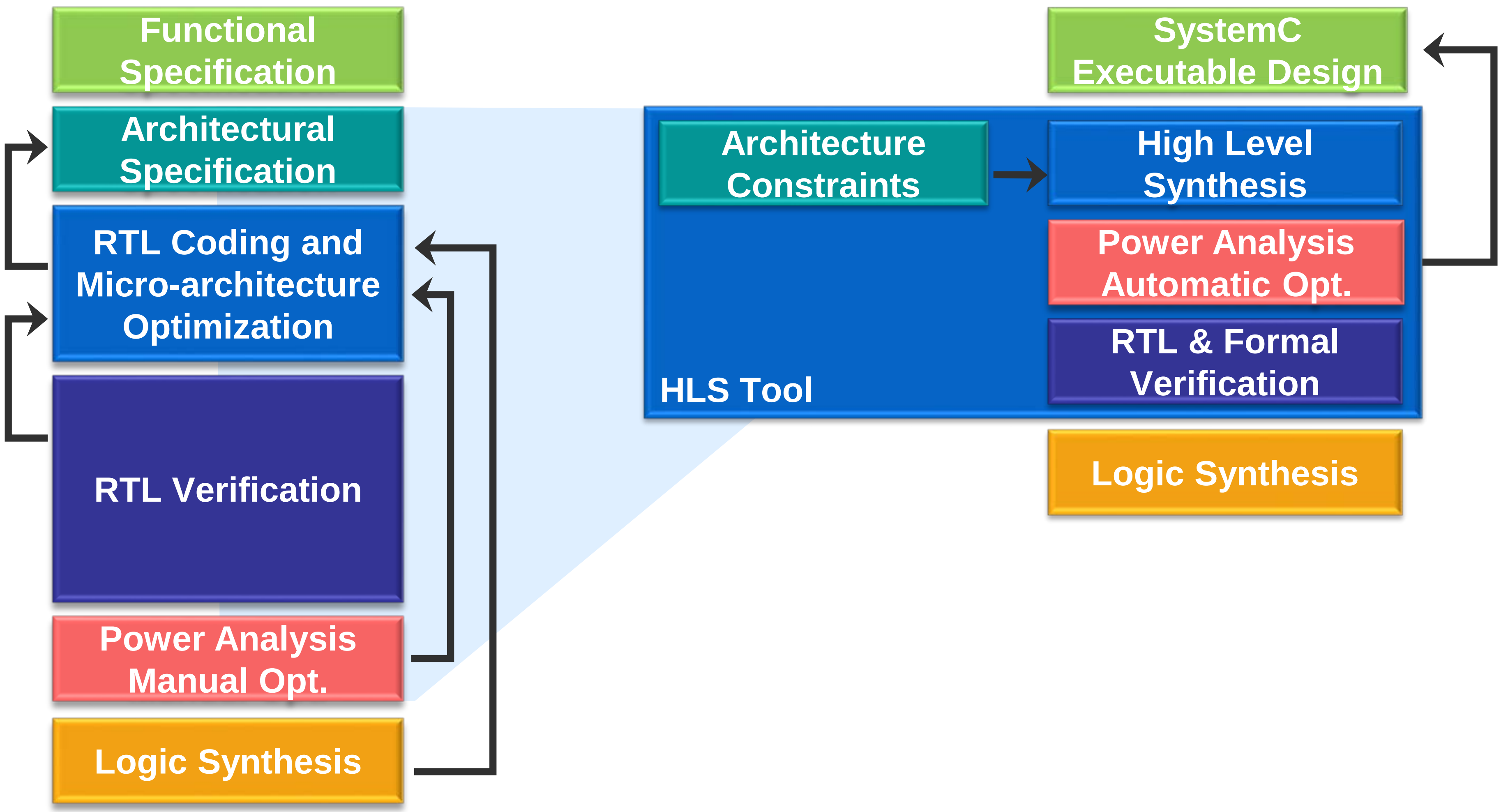
1. Mapping from abstract transactions to pin-accurate protocols



2. Optimizing for performance & area in the target technology

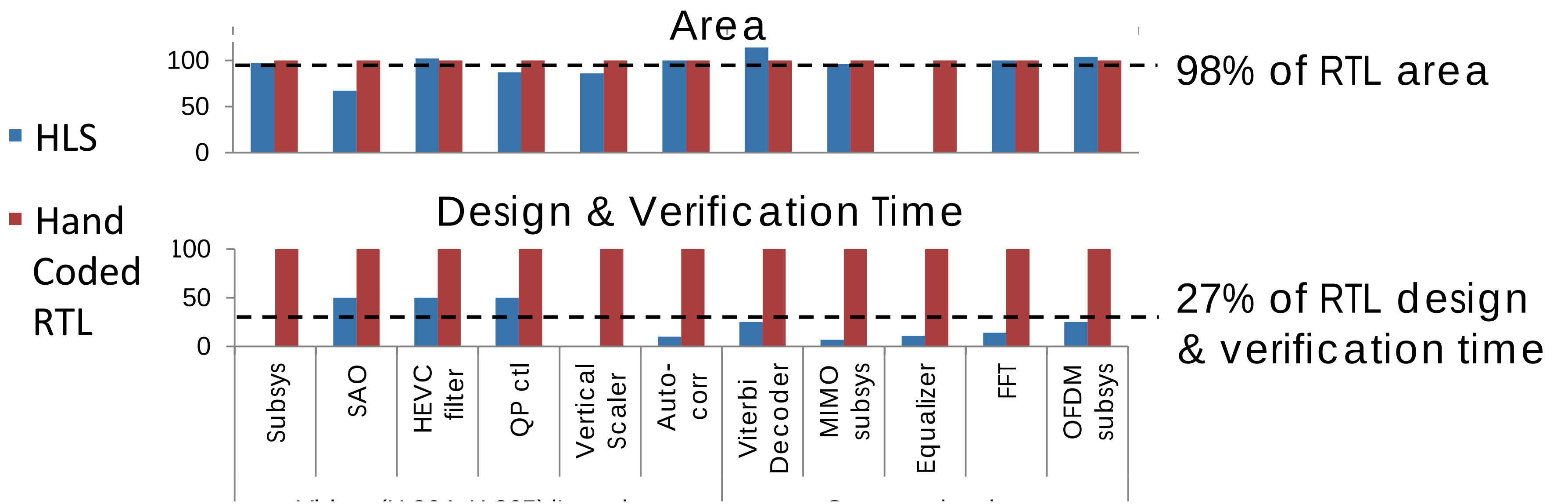


Traditional Design Flow vs. HLS Flow



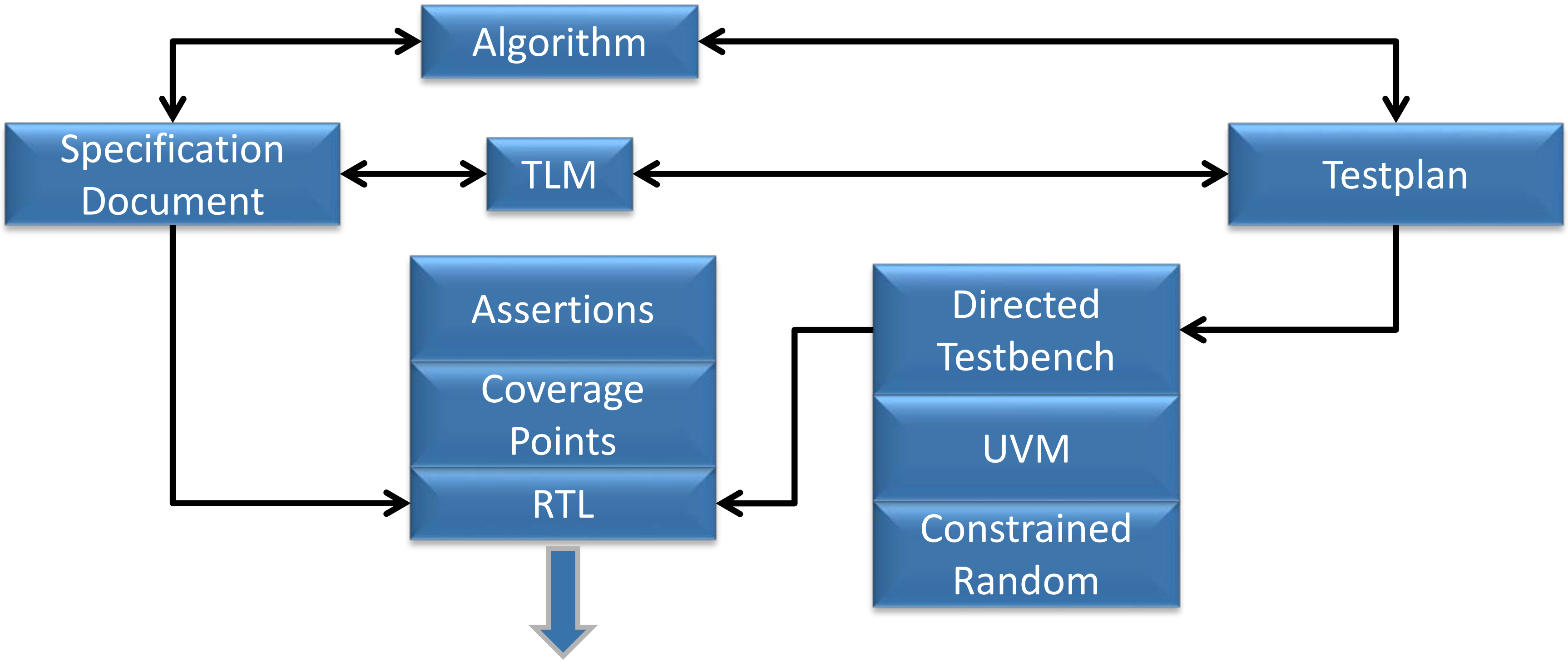
HLS Delivers QofR & Crushes RTL Design Time

- Examples of video, imaging and communication projects
- Generated RTL matches power, performance and area
- Projects complete in 10% to 50% of time needed for RTL



HLS-enabled Verification

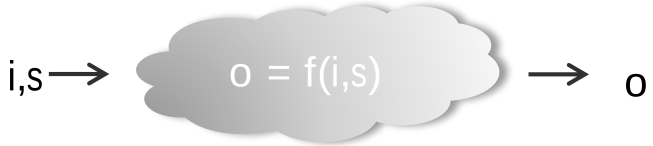
Advances in Verification Technology



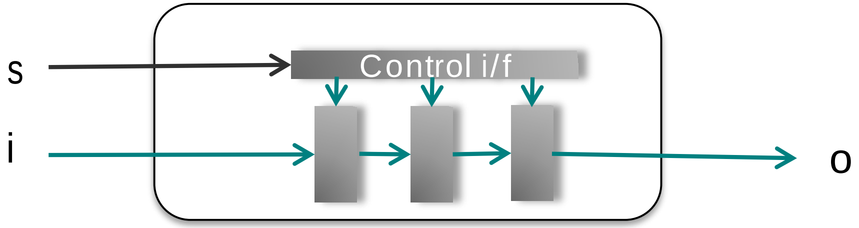
Export for SoC integration

Review of Hardware Abstractions

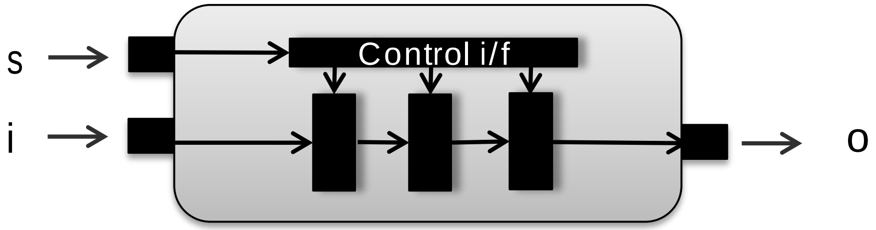
- Algorithmic Model
  - No timing or architecture



- Transaction-Level Model
  - Partitioned for hardware architecture



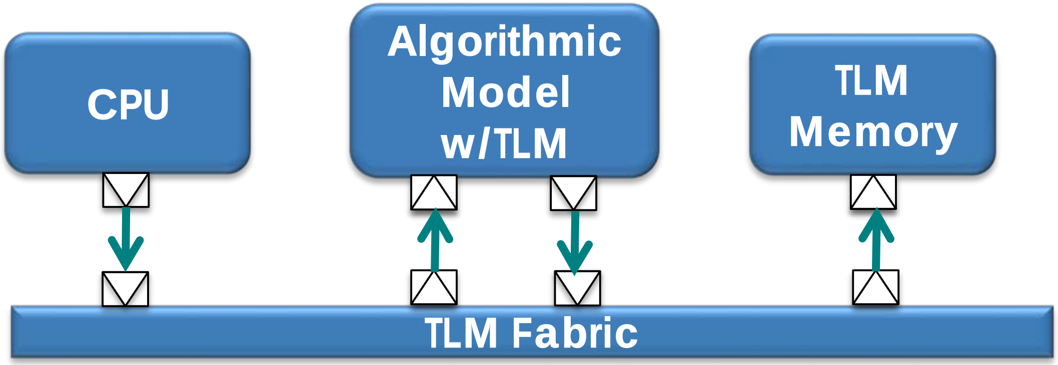
- RTL Implementation
  - Synthesizable to gates



Verification in ESL Platform

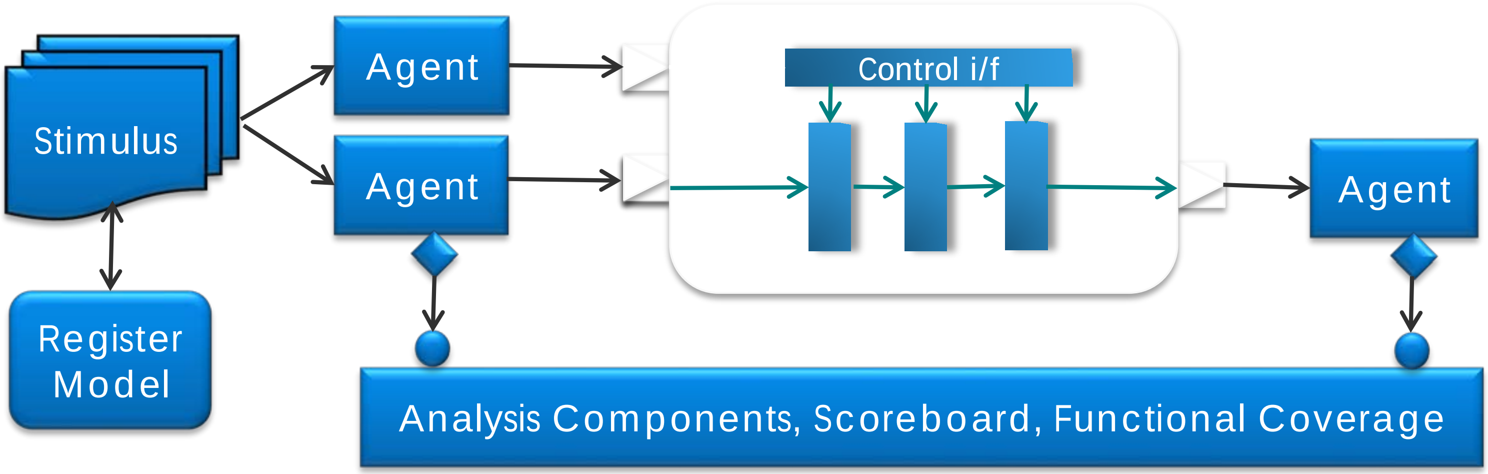
- Algorithmic Model can be used as a reference model
  - Can be embedded in SV/UVM environment
- Enables early software development
  - Software-driven testing
- <10 minutes simulation vs. 1 month simulation in RTL

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

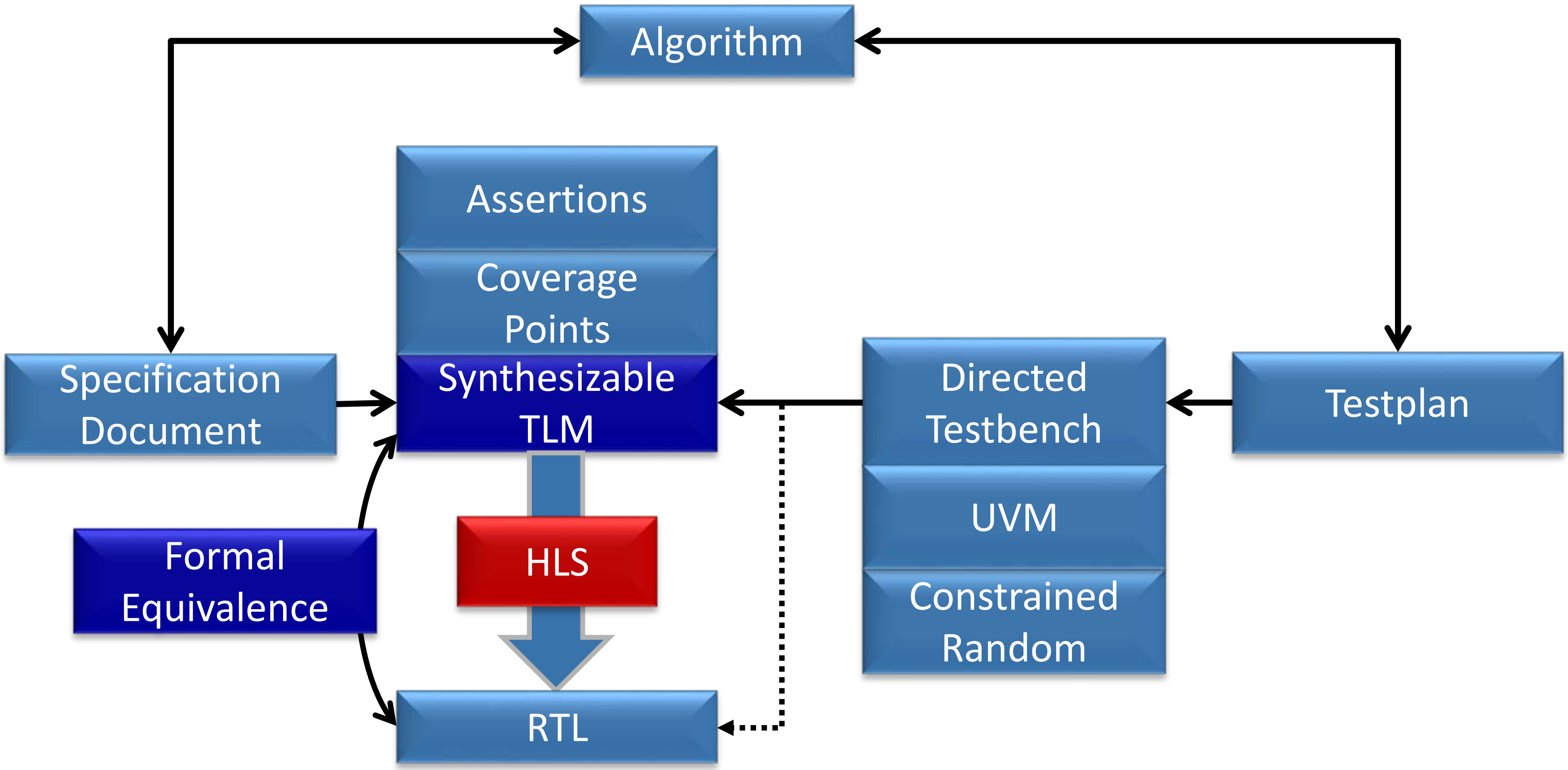
- Assertions and Cover Points
  - Functional
  - SystemC
- Testplan Coverage
  - Based on cover assertions
  - Some tests require RTL
- Code Coverage
  - Function, Line, Condition/Decision
  - Many C++ based tools
  - Nothing specialized for hardware

```
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
- Reuse 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

Thank You!

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

---

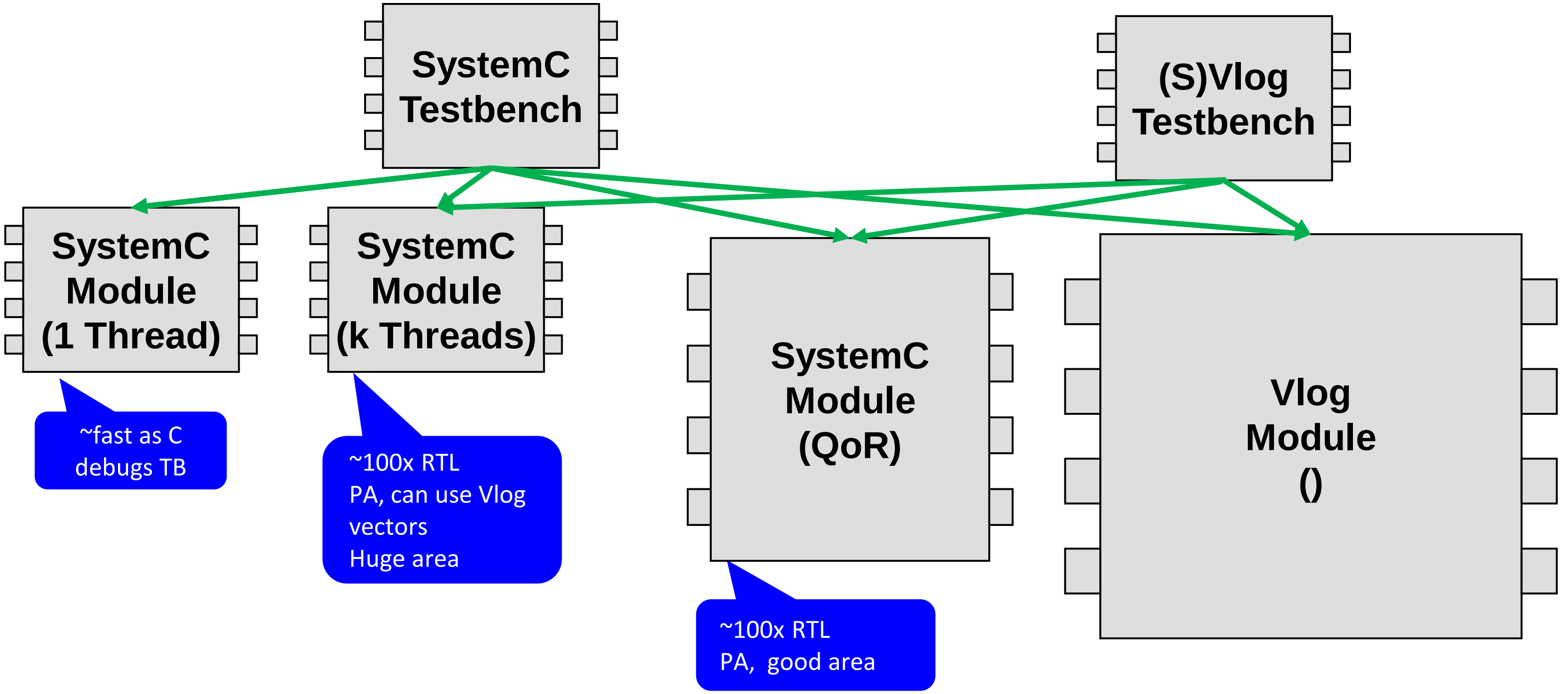
(Not worth it….)

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

HLS Increases Test Velocity

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) algo bugs as possible in the fast SystemC simulation
- Mixed language sim to find final communication bugs (and spec changes)



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 DOA test 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
  - operator[], Template,
  - Use language to make sure each decision is represented exactly once
- Things that are hard get implemented as library components
  - Start to think of reuse (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
  - You do get a waveform and all your vendor tools support mixed language
  - GDB augmented with SC viewers
  - Keep your SystemC test complete on algo-functionality
- 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, the 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)

HLS lets you find and fix your bugs faster but you still need a full testplan to release quality silicon.

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"

Thank You!

HLS for the FPGA/Programmable Market

Dirk Seynhaeve (Altera – now part of Intel)
Product Planning

now part of Intel

Agenda

- What are FPGAs
- Who wants to use them
- Why is HLS important
- How does this affect standardization

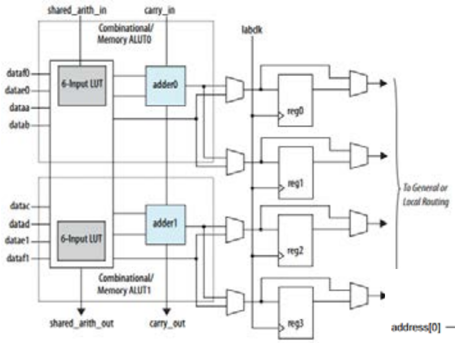
- **What are FPGAs**
- Who wants to use them
- Why is HLS important
- How does this affect standardization

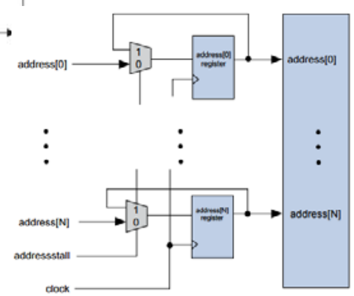
Compromise

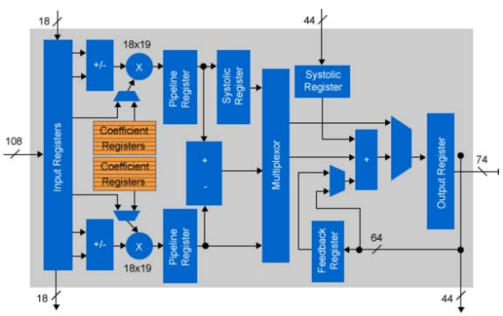
CPU/GPU/DSP	ASIC
High power consumption	Low power consumption
Low performance, high latency	High performance, low latency
Low cost	Low cost at high volume
Many low cost spins	Spins to be avoided
In field updates (remote)	No flexibility
Easy functionality (program)	Specialized functionality (design)

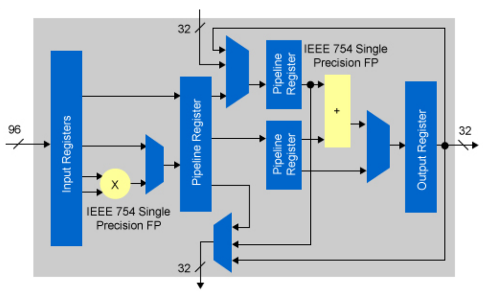
FPGA

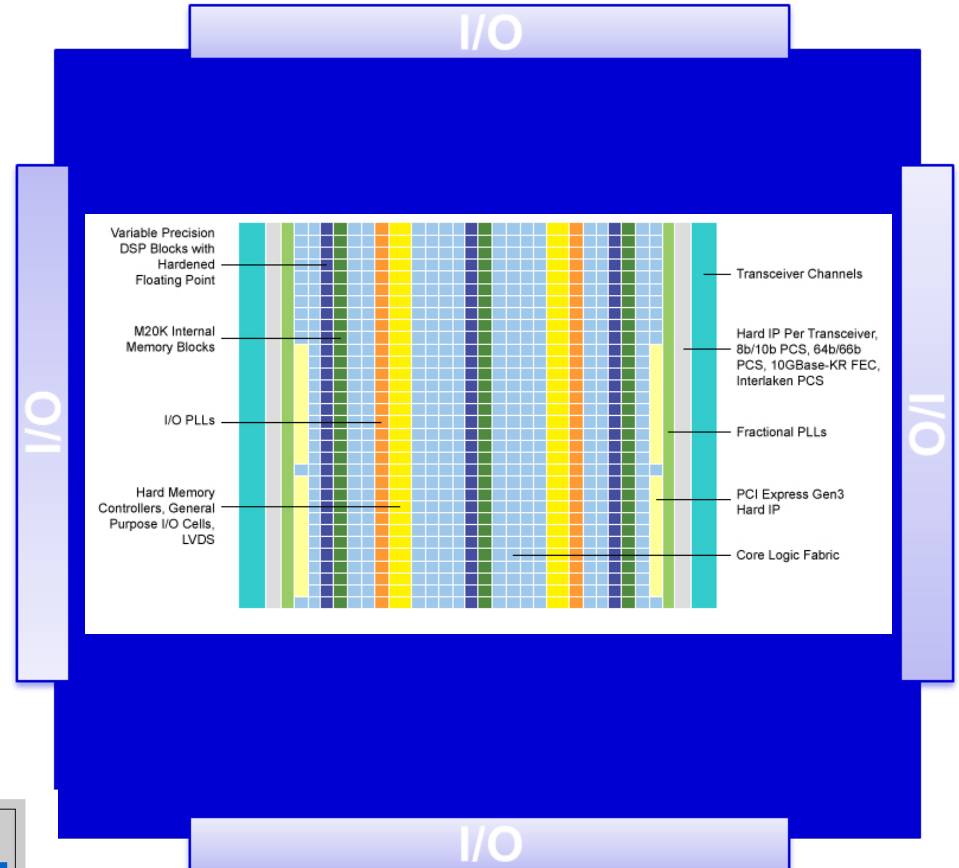
Parallel Everything









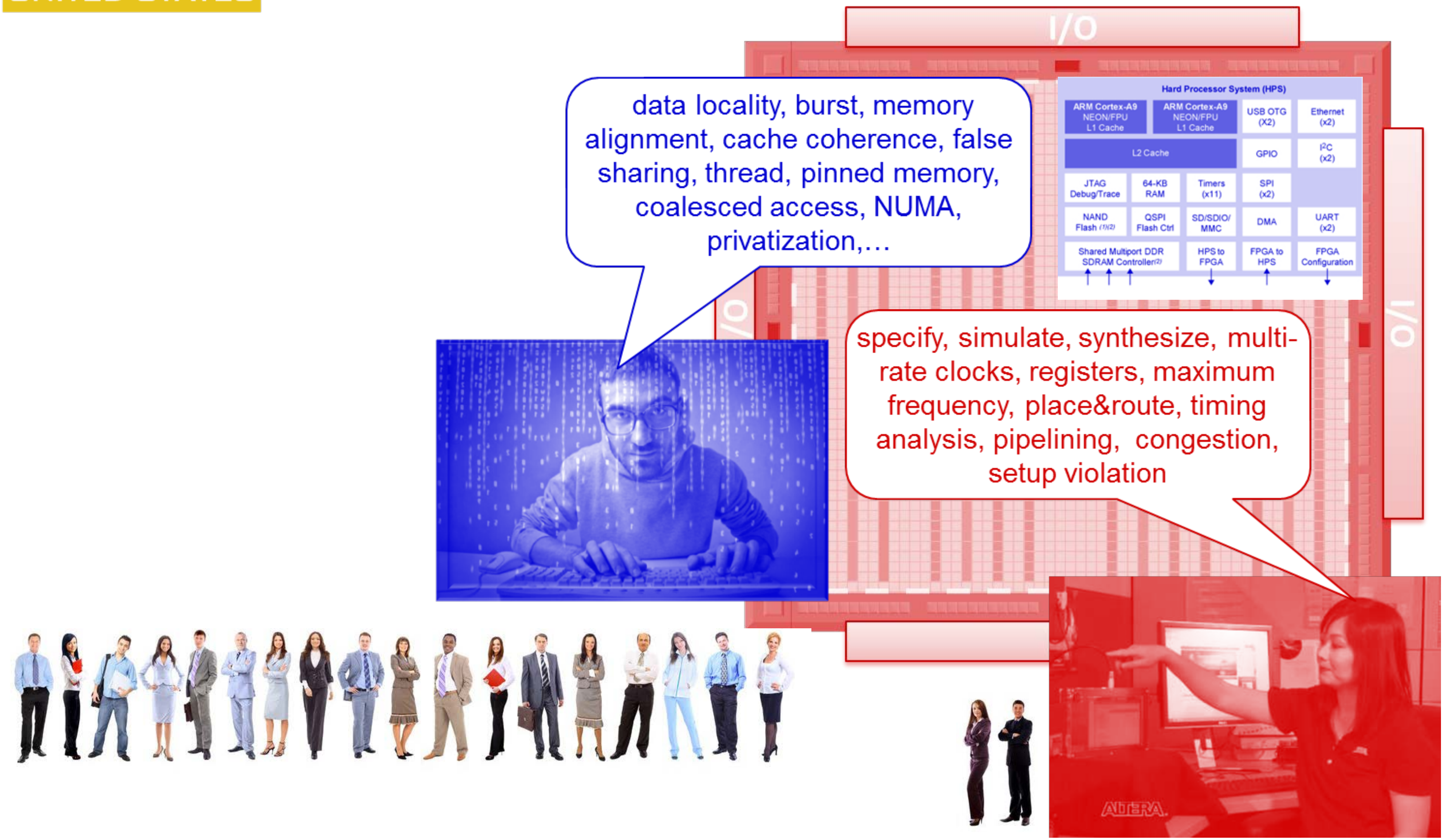


Massive…

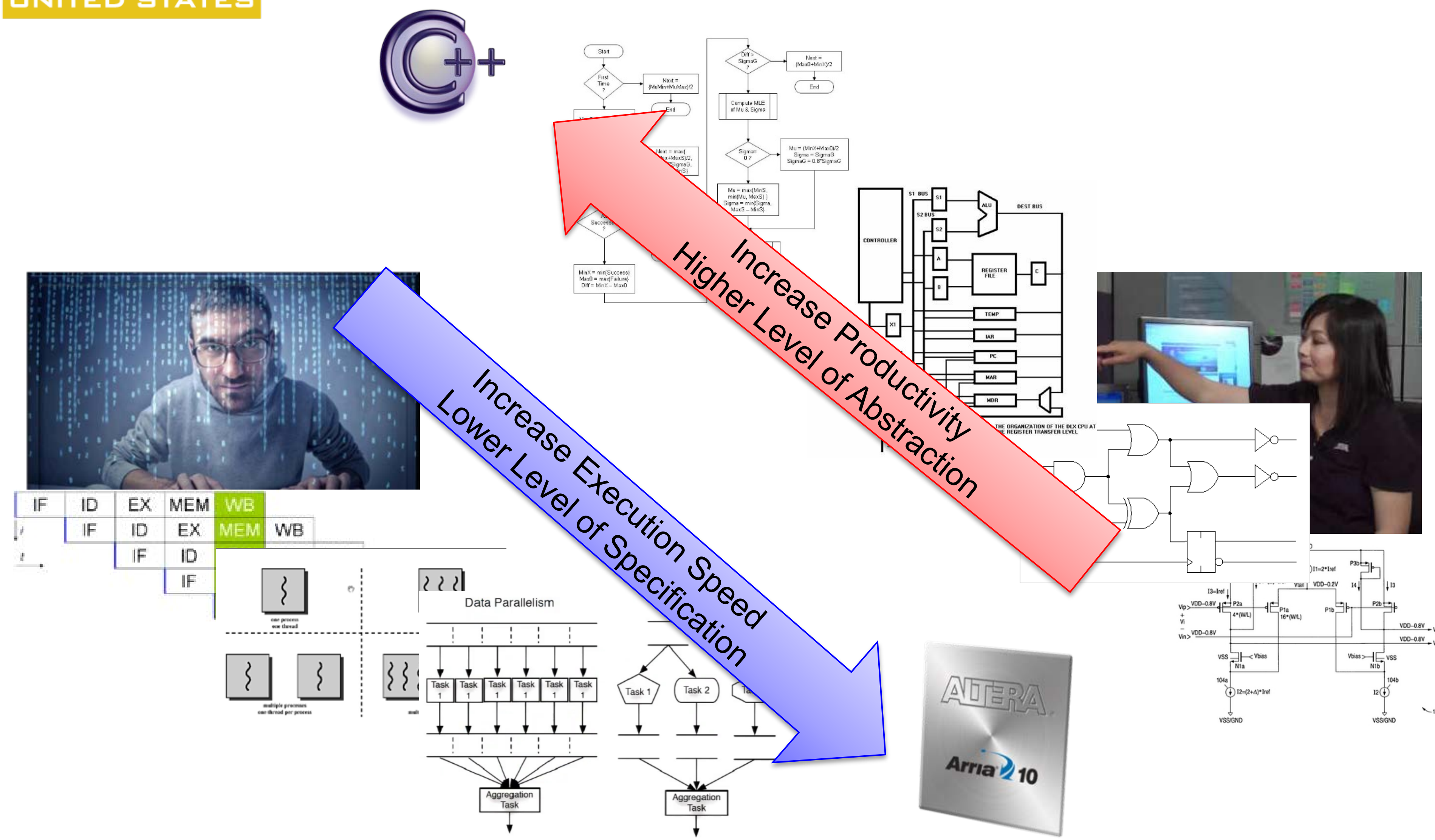
	Product Line	GX 570	GX 660	GX 900	GX 1150
Resources	Part number reference	10AX057	10AX066	10AX090	10AX115
	LEs (K)	570	660	900	1,150
	Adaptive logic modules (ALMs)	217,080	250,540	339,620	427,700
	Registers	868,320	1,002,160	1,358,480	1,708,800
	M20K memory blocks	1,800	2,133	2,423	2,713
	M20K memory (Mb)	35	42	47	53
	MLAB memory (Mb)	5.0	5.7	9.2	12.7
	Hardened single-precision floating-point multiplers/adders	1,523/1,523	1,688/1,688	1,518/1,518	1,518/1,518
	18 x 19 multipliers	3,046	3,376	3,036	3,036
	Peak GMACS	3,351	3,714	3,340	3,340
	GFLOPS	1,371	1,519	1,366	1,366
Clocks, Maximum I/O Pins, and Architectural Features	Global clock networks	32	32	32	32
	Regional clocks	8	16	16	16
	I/O voltage levels supported (V)	5, 1.35, 1.8, 2.5, 3.0²			
	I/O standards supported	nly: 3 V LVTTL, 2.5 V CMOS ntial POD12, Differential POD10, LVDS, RSDS, mini-LVDS, LVPECL -15 (I and II), HSTL-12 (I and II), HSUL-12, Differential SSTL-135, Differential SSTL-125, Differential SST fferential HSTL-15 (I and II), Differential HSTL-12 (I and II), Differential HSUL-12			
	Maximum LVDS channels (1.6 G)	270	270	384	384
	Maximum user I/O pins	624	624	768	768
	Transceiver count (17.4 Gbps)	48	48	96	96
	Transceiver count (28.3 Gbps)	–	–	–	–
	PCIe hard IP blocks (Gen3)	2	2	4	4
	Maximum 3 V I/O pins	48	48	–	–
	Memory devices supported	Xtreme, LPDDR3, LPDDR2, RLDRAM 3, RLDRAM II, LLDRAM II, HMC			

- What are FPGAs
- **Who wants to use them**
- Why is HLS important
- How does this affect standardization

Problem


data locality, burst, memory alignment, cache coherence, false sharing, thread, pinned memory, coalesced access, NUMA, privatization,…
specify, simulate, synthesize, multi-rate clocks, registers, maximum frequency, place&route, timing analysis, pipelining, congestion, setup violation
I/O
Hard Processor System (HPS)
ARM Cortex-A9 NEON/FPU L1 Cache
ARM Cortex-A9 NEON/FPU L1 Cache
USB OTG (X2)
Ethernet (x2)
L2 Cache
GPIO
I²C (x2)
JTAG Debug/Trace
64-KB RAM
Timers (x11)
SPI (x2)
NAND Flash
QSPI Flash Ctrl
SD/SDIO/ MMC
DMA
UART (x2)
Shared Multiport DDR SDRAM Controller
HPS to FPGA
FPGA to HPS
FPGA Configuration
ALTERA

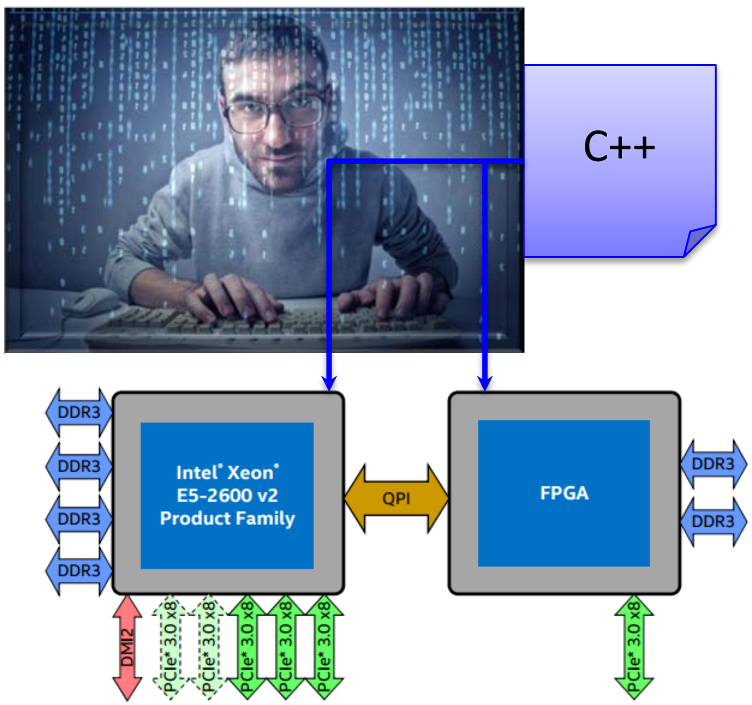
Observation



Scenarios

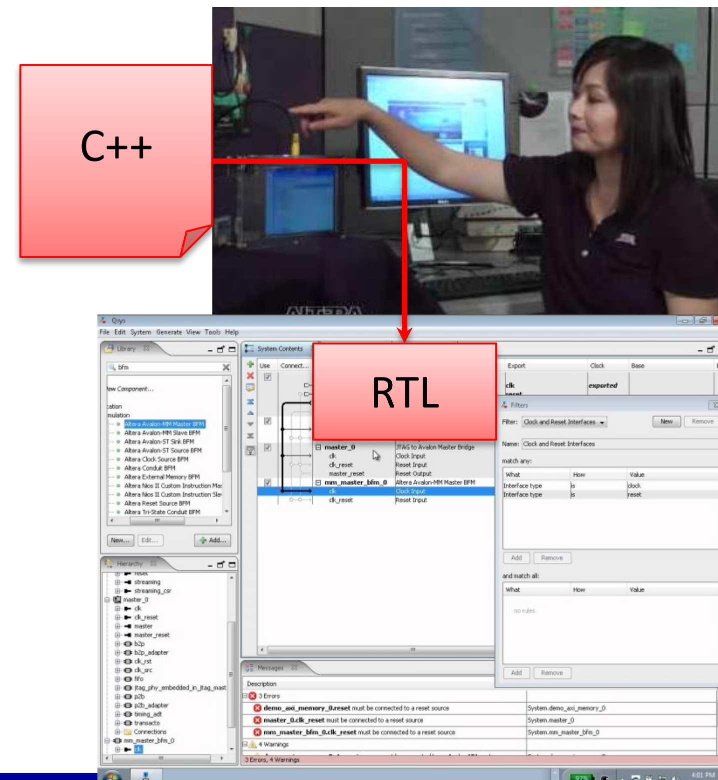
Path to acceleration

- Enablement



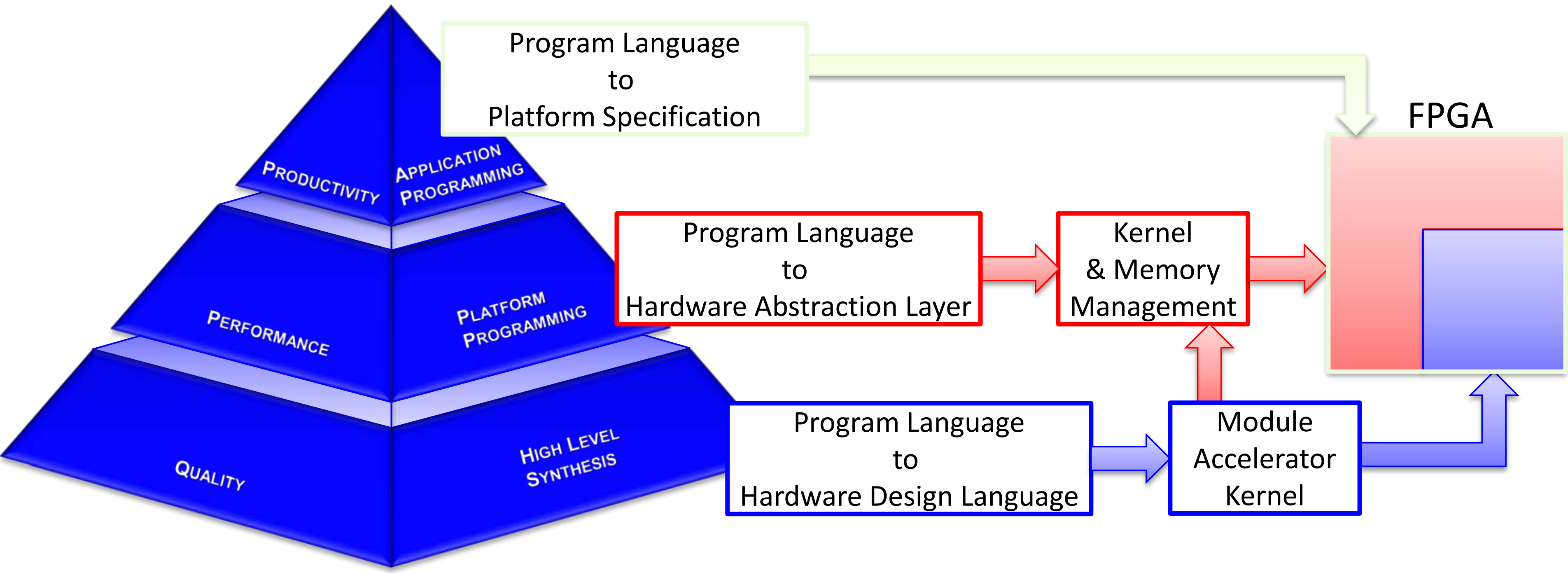
Faster path to verified RTL

- Productivity

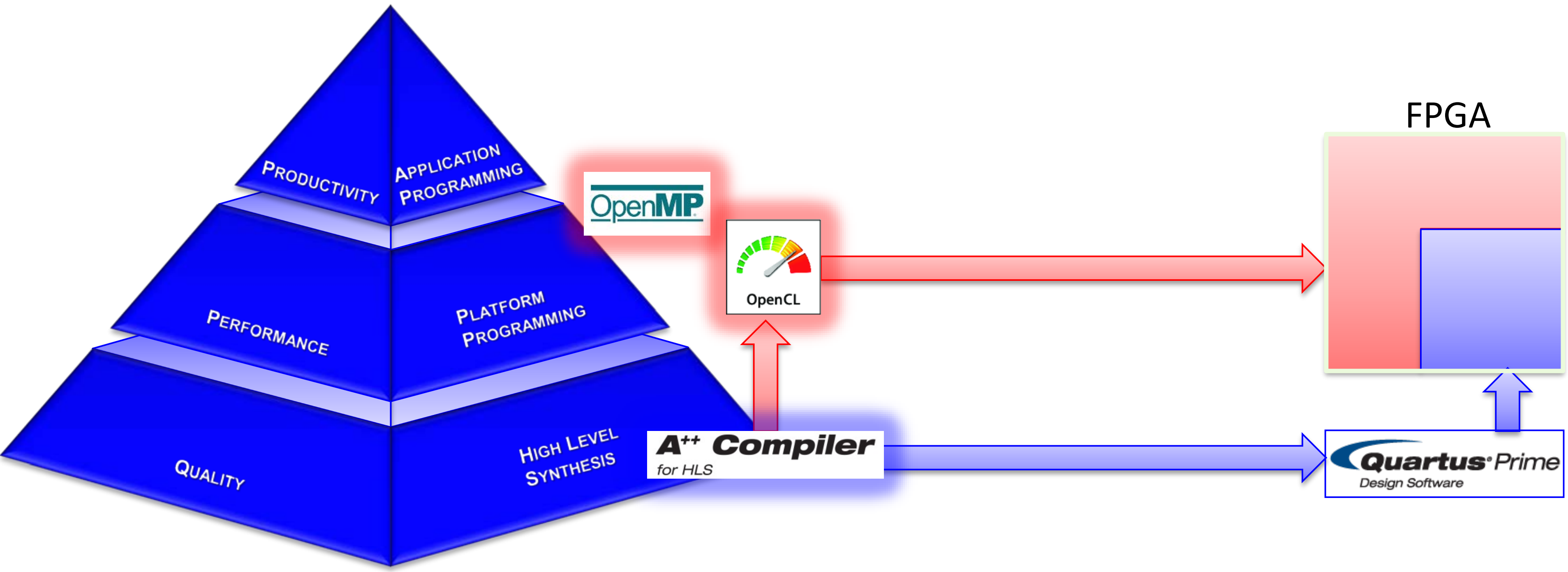


- What are FPGAs
- Who wants to use them
- **Why is HLS important**
- How does this affect standardization

Bridging the Gap



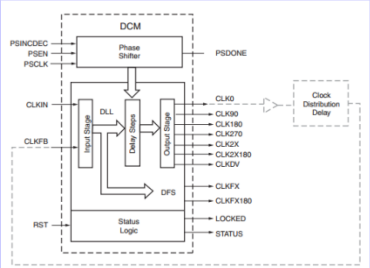
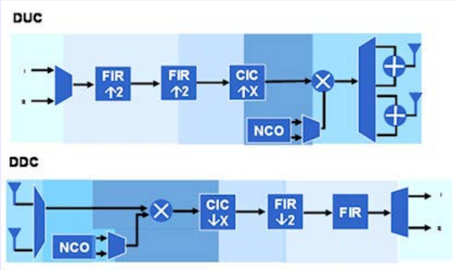
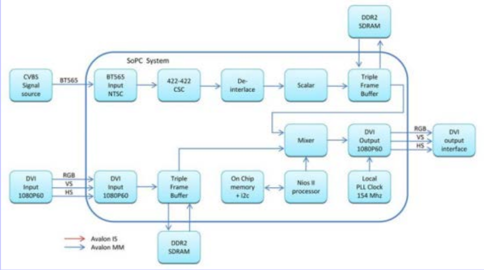
Current Approach



Agenda

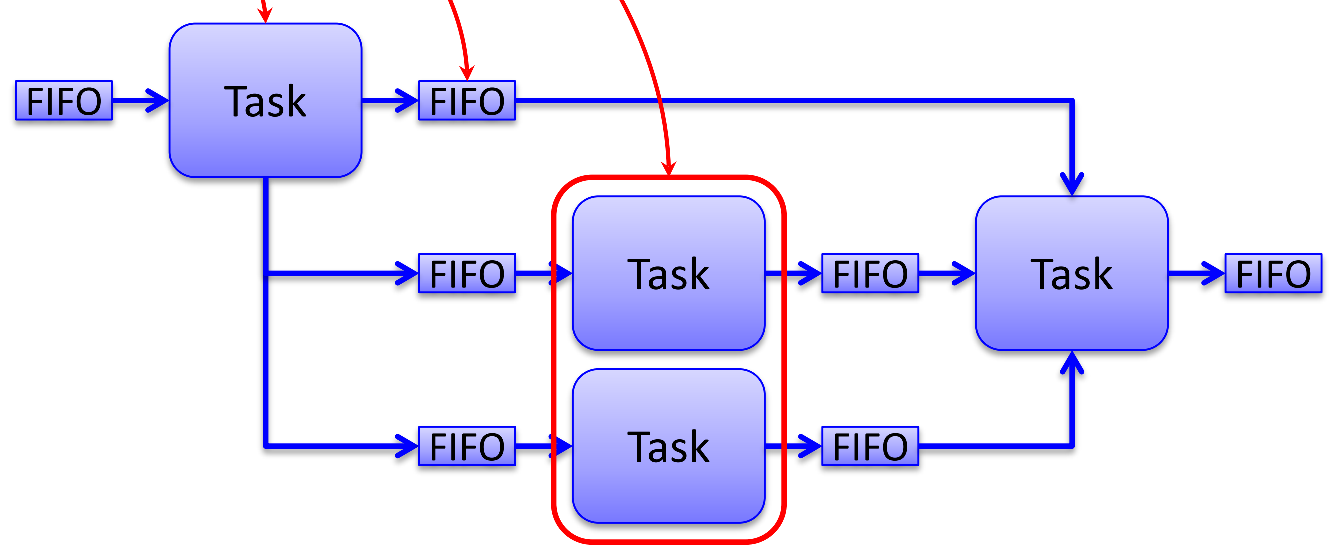
- What are FPGAs
- Who wants to use them
- Why is HLS important
- **How does this affect standardization**

Evolution

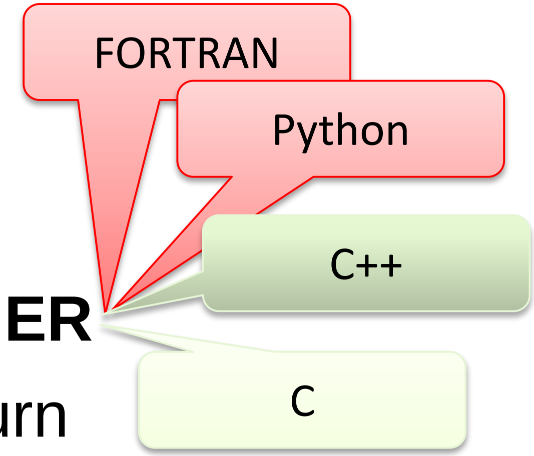
Era		FPGA deployment	
	1980	Glue logic (flexible IO management, protocol bridges,…)	
	1990	Customizable functions (telecommunication filters)	
	2010	Data processing systems (video processing, cloud computing,…)	

Requirements

FPGA Hardware	HLS Standard
Parallel	Directives to introduce parallelism in sequential code
Streaming	Self-synchronizing Channels
Low Power	Arbitrary Precision



Requirements

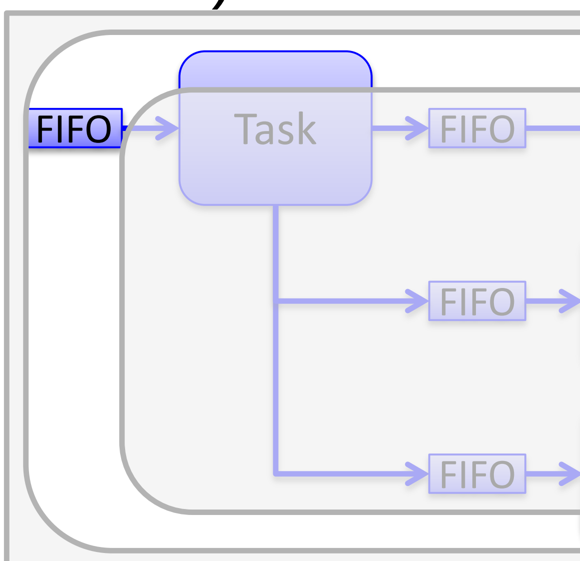


FPGA

- Register/Wire
- FIFO
- Memory Mapped (MEMIF/CSR)

PROGRAMMER

- Scalar/Return
- Pointer/Reference
- Array




```
int16_t foo (
  uint8_t control,
  int16_t *data_in,
  int16_t scratch_pad[128]) {
  unsigned char c;
  if (control)
    for(i=0;i<8;i++)
      scratch_pad[i] = *data_in++;
…
  return(scratch_pad[128]);
}
```

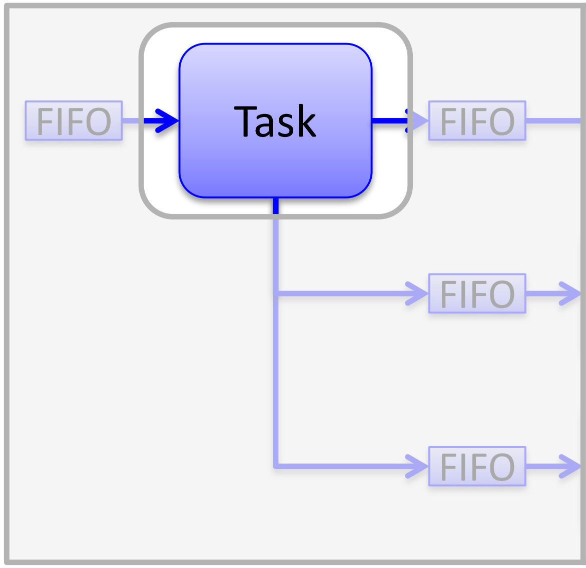
Requirements

FPGA

- Minimize bits
  - Faster
  - Lower Power
  - Smaller (more functionality)

PROGRAMMER

- Fast execution
- Comprehensive (signed/unsigned)
- Flexible (slice/range)




```
hls_int<129> message = “0x10000”;
hls_fixed<7,3,true,HLS_RND> one_way;
hls_fixed<7,3,HLS_RND>      another_way;
hls_int<13>                 x,y,z;
…
message(129,127)=one_way(7,5);
message.set_slc(124,another_way.slc<3>(5));
…
z = x<<k + y;
```

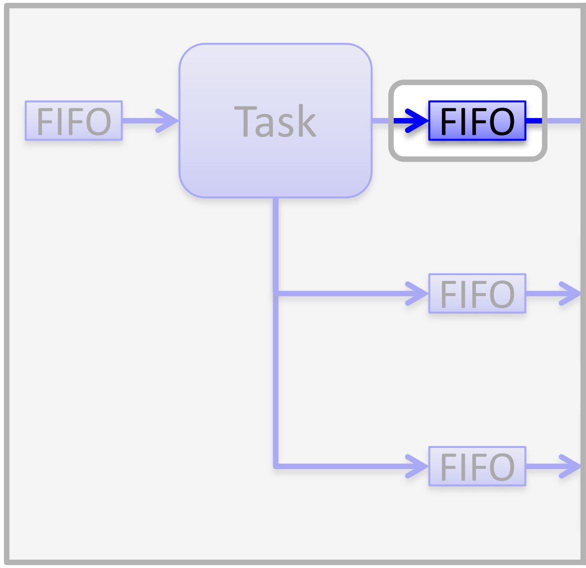
Requirements

FPGA

- FIFO (finite, point-to-point)
- Empty/Full

PROGRAMMER

- STL deque
- Blocking/non-blocking




```
hls_channel<hls_int<17>> channel;
Task1: {
…
while (!channel.write(value));
…
}
Task2: {
…
while (channel.read(value));
…
}
```

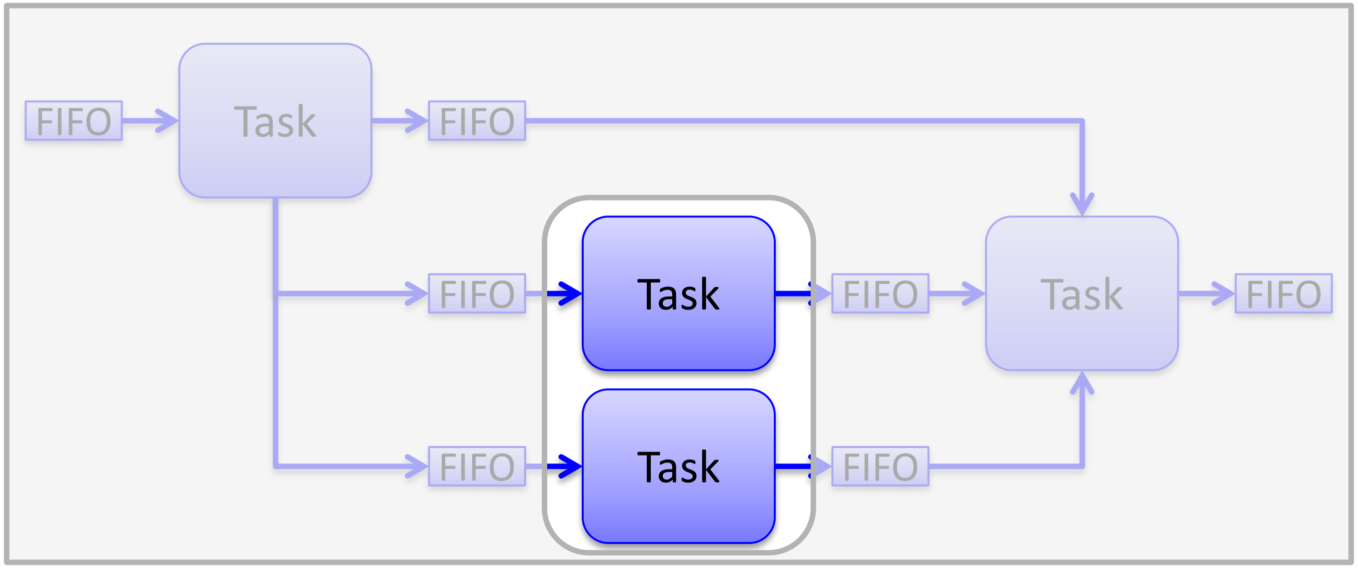
Requirements

FPGA

- Parallel Compute Units

PROGRAMMER

- Threads
- SIMD (vectorization)



Summary

- WHAT
  - Hardware-centric platform with software-centric benefits
- WHO
  - Programmers for execution speed
  - Designers for productivity
- WHY
  - HLS bridge from functionality to hardware specification
- HOW
  - Standards that let the FPGA be an FPGA, and yet respect programmer paradigms

Thank You!

SystemC Synthesis Standard: Which Topics for Next Round?

Frederic Doucet

Qualcomm Atheros, Inc

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!

C++ Language and Math Libraries

- 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 …

- Math libraries
  - AC datatypes and SystemC datatypes
  - sc_complex
  - sc_float

Channel Libraries

- Which elements :
  - FIFOs
  - point-to-point
  - pulse
  - ring buffer
  - line buffers
  - CDC
  - etc.
- Standard interpretation of the TLM interface in synthesis
  - Must blocking vs. may-block vs. non-blocking
  - Use TLM 1.0 as reference or not (need add reset)

Micro-architecture Directives

- Standard list of directives :
  - Loop handling:
    - unroll, partial unroll, pipeline, sequential
  - Function handling
    - Sequential function, pipelined, parallel, map to custom resource, etc.
  - Array handling:
    - flatten, map-to-memory, map-to-reg-file, split, combine, resize, etc.
  - Custom resource:
    - pipelined, combinational
  - Inputs:
    - stable, delay
  - Latencies:
    - Min latency, max-latency
  - Etc.

Micro-architecture Directives

- How to specify the directives:
  - Pragma in the code
  - Tcl commands in synthesis directive file
  - Directive in code (empty functions or variables with specific meaning)

- How to apply the directives
  - How to “label” and “find” structures in the code
    - *“The loop filter_kernel, unroll it”*

Synthesis Structures

- How to interpret the SystemC CDFG and synthesis directive
  - The generated RTL behaves equivalently in all tools
  - Consistent interpretation across tools
- How to write a pipeline
  - Where to freeze, where to free the I/O
  - Where to expand the pipeline
- Cycle-accurate, cycle close and super-cycle modes
  - Clearly define and implement the scheduling mode
- How to specify and create custom resources
  - Specified as C++ functions or C++ scopes
  - What interfaces to they implement
  - Specify to characterize the custom resource or not with logic synthesis

Memories

- Where are the memories in the SystemC code:
  - Mapping of C++ array into memories (implicit)
  - Using memory channel (explicit)

- How to describe the memory macro to the HLS tool
  - Memory ports, timing, simulation model file, lib file, etc.
  - Standard format

- Using the memory macro in the design (architecture model)
  - Memory port sharing by more than one process in a module
  - Memory port sharing by sub-modules
  - Multi-clock memories
  - Memories inside or outside the module

Tools and Flows

- Standard interpretation of module hierarchy
  - How to set up project with submodules
    - Many modules and processes to synthesize, process them one by one or all at once
  - Where are the memories instantiated

- Standard minimal wrapper generation
  - Tool to provide wrapper for input SystemC in SystemVerilog context
  - Tool to provide wrapper for generated Verilog in SystemC

  - Mostly about datatype conversions
  - Make the wrapper lightweight enough so it can be used with various HDL simulators
  - Help ease flow migration

Summary

- HLS is rapidly growing in adoption and proving its value for multiple users (design, verification, accelerated software…)
- Accellera SystemC synthesis subset standardization helps focus so the ecosystem can grow around it
- There are great areas for “what’s next” to standardize to complete the ecosystem for HLS

Join the discussion!
Join the SWG calls!
Drive what you need in the standard!

Thank You!