

Using SystemC for high-level synthesis and integration with TLM

Presented at India SystemC User Group (ISCUG)
<http://www.iscug.in>

April 14, 2013

Mike Meredith – VP Strategic Marketing
Forte Design Systems

- **High-level synthesis**
 - A brief history of high-level synthesis
 - What is HLS?
 - HLS Basics
- **Synthesizability**
 - Introduction to the synthesizable subset standard draft
 - Recommendations and extensions
- **Keys to raising abstraction level**
 - Implicit state machine vs explicit state machine
 - Design exploration
 - HLS and Arrays
 - HLS and I/O protocols
- **Integrating synthesizable code in TLM models**
- **User Experience**

A Brief History Of High-level Synthesis

What's In A Name?

**Behavioral
Synthesis**

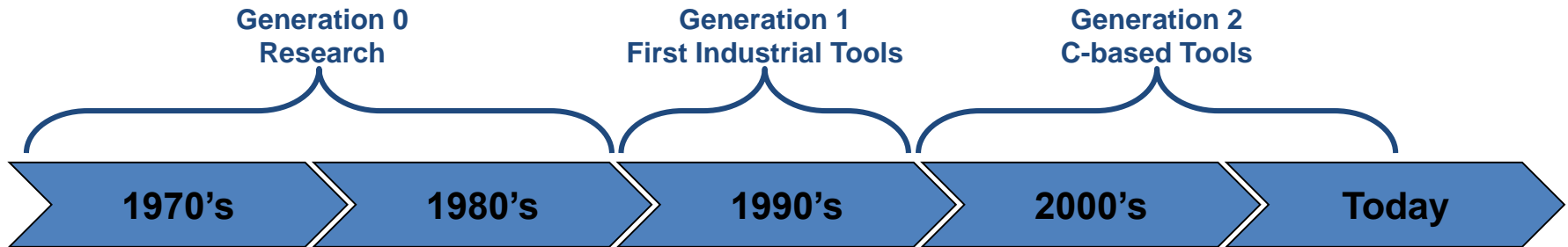


**Algorithmic
Synthesis**

**ESL
Synthesis**

**High-level
Synthesis**

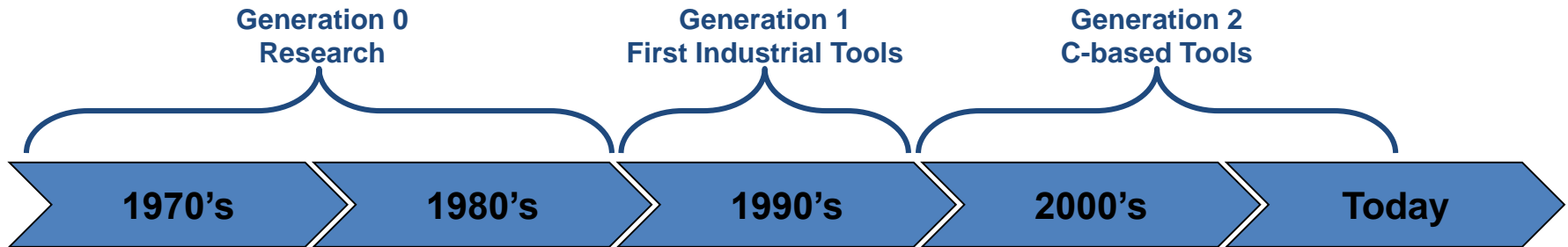
Synthesizing Hardware From A High-level Description -- Not a New Idea!



- **Generation 0 - Research**

- 1970's - 1990
- First mention (that I can find) - 1974
 - CMU: Barbacci, Siewiorek "Some aspects of the symbolic manipulation of computer descriptions"
- A sampling of key researchers:
 - Thomas, Paulin, Gajski, Wakabayashi, DeMann
- Academic implementations
 - 1979 CMU - CMU-DA
 - 1988 Stanford Hercules
- Most research focused on scheduling and allocation for datapath-dominated designs

Generation 1 - 1990's



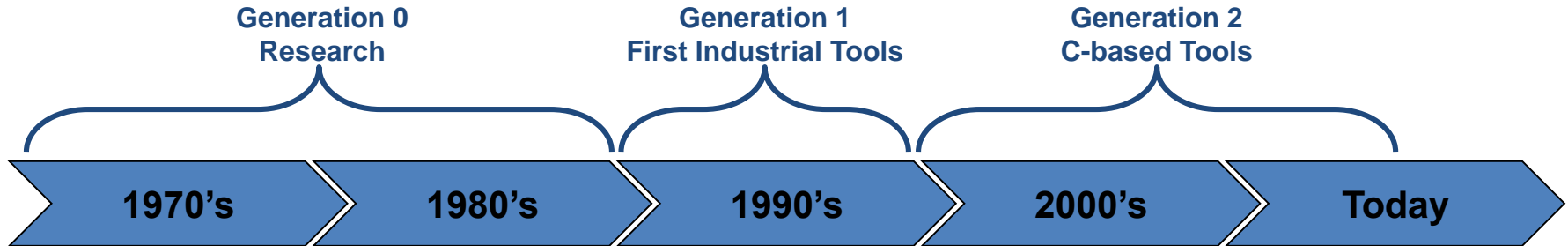
- **First Industrial Tools**

- NEC Cyber Workbench - 1993 (internal to NEC)
- Synopsys Behavioral Compiler - 1994
- Cadence Visual Architect - 1997
- Mentor Monet - 1998

- **Challenges**

- Synthesis direct to gates
- HDL's are not good behavioral algorithm languages
- Poor quality of results
 - Area and timing closure
- Verification difficulty
 - Interface synthesis not good
 - I/O cycle accuracy not guaranteed

Generation 2 - 2000's



- **“Bloom” of industrial implementations**

- Forte Cynthesizer
- Celoxica Agility Compiler
- Mentor Catapult
- Synfora Pico Express
- Cadence C to Silicon Compiler

- **Reasons for success**

- Adoption of C & C variants for input give access to algorithm code
- Adoption of flows that include trusted logic synthesis tools
- Quality of results caught up with hand-coded
- Verification considered along with design
- Designs getting too big for RTL

Why RTL Productivity Isn't Enough (Hint: Moore's Law)

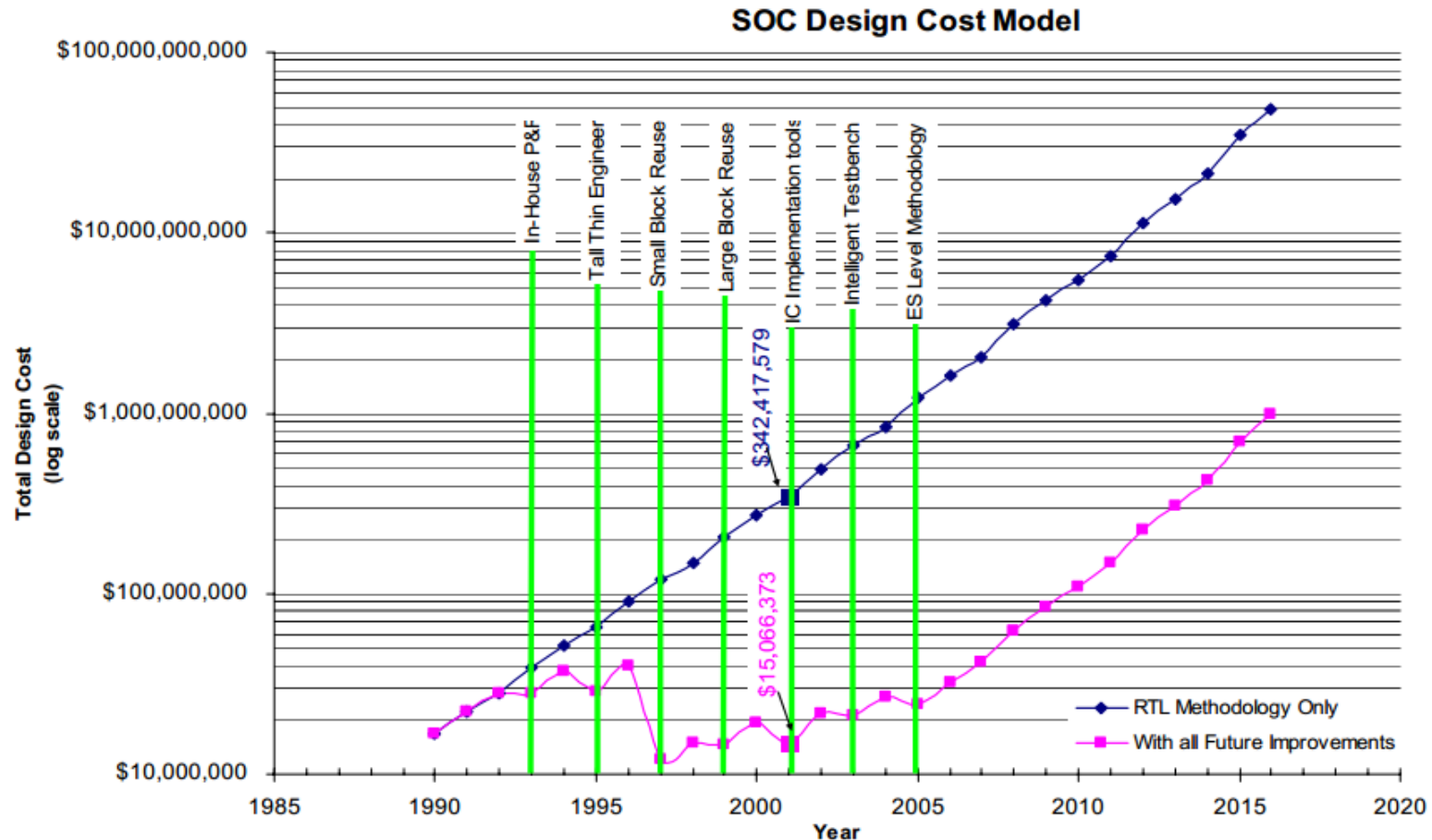


Figure 1: Rising cost of IC design and effect of CAD tools in containing these costs. [Courtesy: Andrew Kahng, UCSD and SRC]

What Is High-level Synthesis

What Is High-level Synthesis?

It's all about abstraction

- **Y-chart**

- Gajski & Kuhn
- Captures relationships between specification domains and abstraction levels

Logic Synthesis:

Register-transfer → Gates/Flipflops

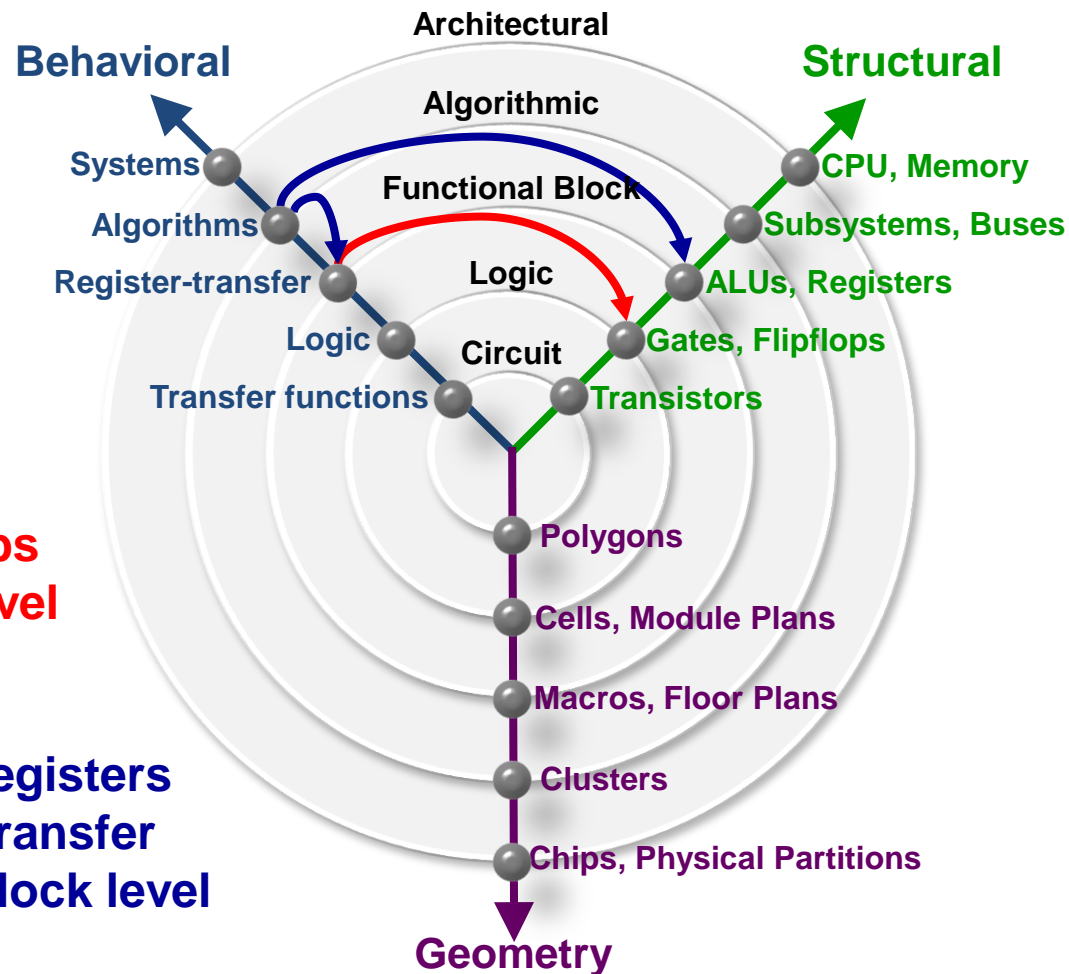
Functional Block level → Logic level

High-level Synthesis:

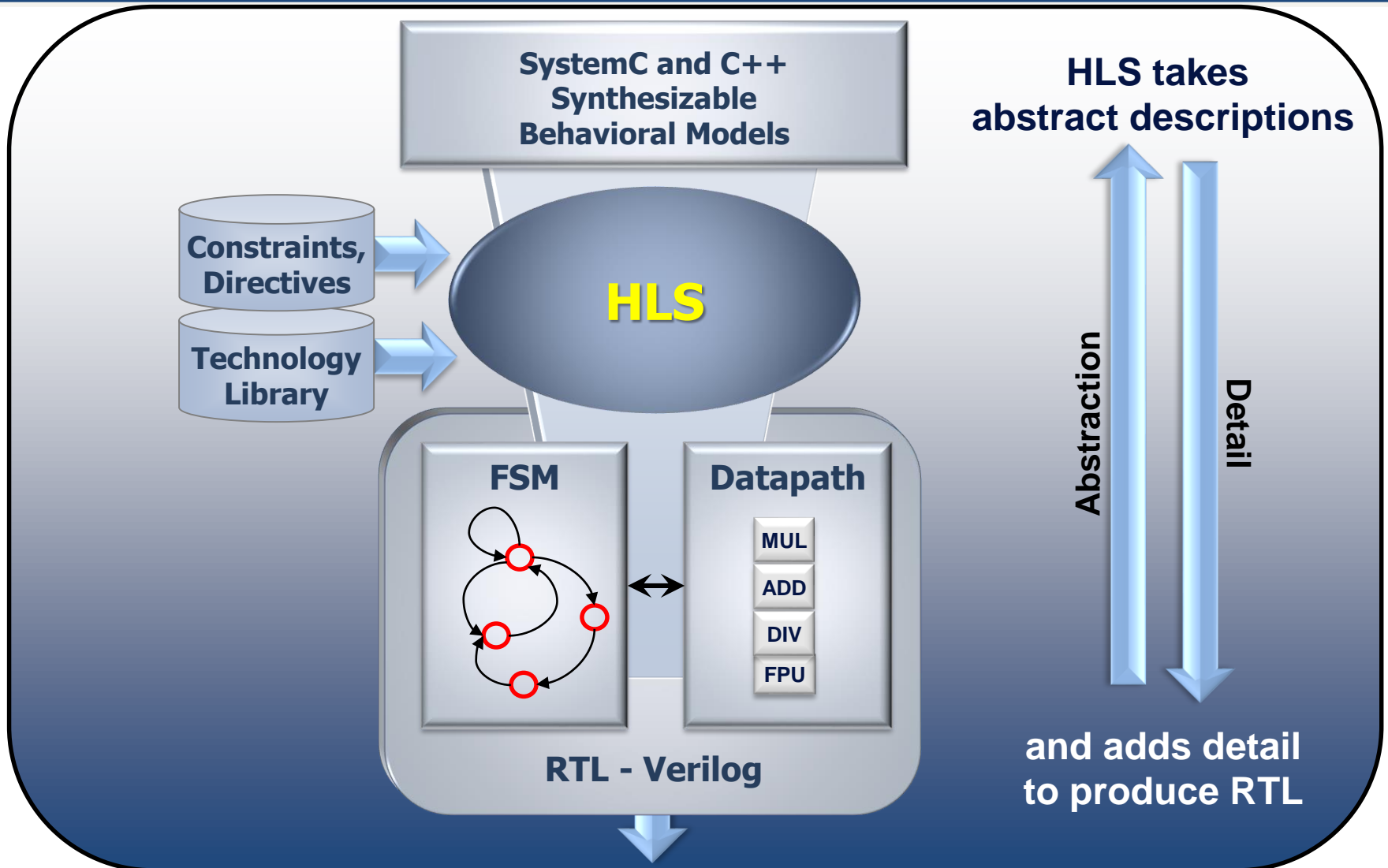
Datapath: Algorithms → ALUs/Registers

Control: Algorithms → Register-transfer

Algorithmic level → Functional Block level



What is High-Level Synthesis?



What is High-Level Synthesis?

- **HLS creates hardware from un-timed algorithm code**
 - Synthesis from a behavioral description
- **Input is much more abstract than RTL code**
 - No breakdown into clock cycles
 - No explicit state machine
 - No explicit memory management
 - No explicit register management
 - The high-level synthesis tool does all of the above

High-Level Synthesis is *not*...

- **... the same as writing software**
 - the algorithm may be the same,
but the implementation is *very* different
 - the cost/performance tradeoffs are different
 - e.g. use of large arrays
 - Software can assume essentially infinite storage with equal (quick) access time
 - Hardware implementations must tradeoff storage size vs. access time
 - these in turn affect the coding style used for synthesis
- **... the same as writing RTL**
 - the input code is much more abstract

What does the designer do?

What does the tool do?

Decisions made by designer

- **Function**
 - As implicit state machine
- **Performance**
 - Latency, throughput
- **Interfaces**
- **Storage architecture**
 - Memories, register banks etc
- **Partitioning into modules**

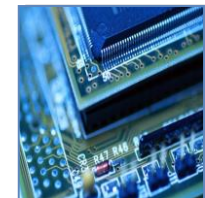
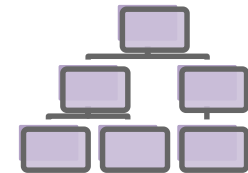
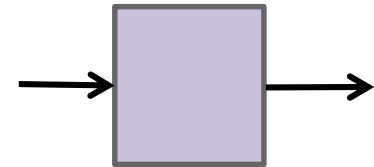
Decisions made by HLS

- **State machine**
 - Structure, encoding
- **Pipelining**
 - Pipeline registers, stalling
- **Scheduling**
 - Memory I/O
 - Interface I/O
 - Functional operations

What is High-level Synthesis Used For?

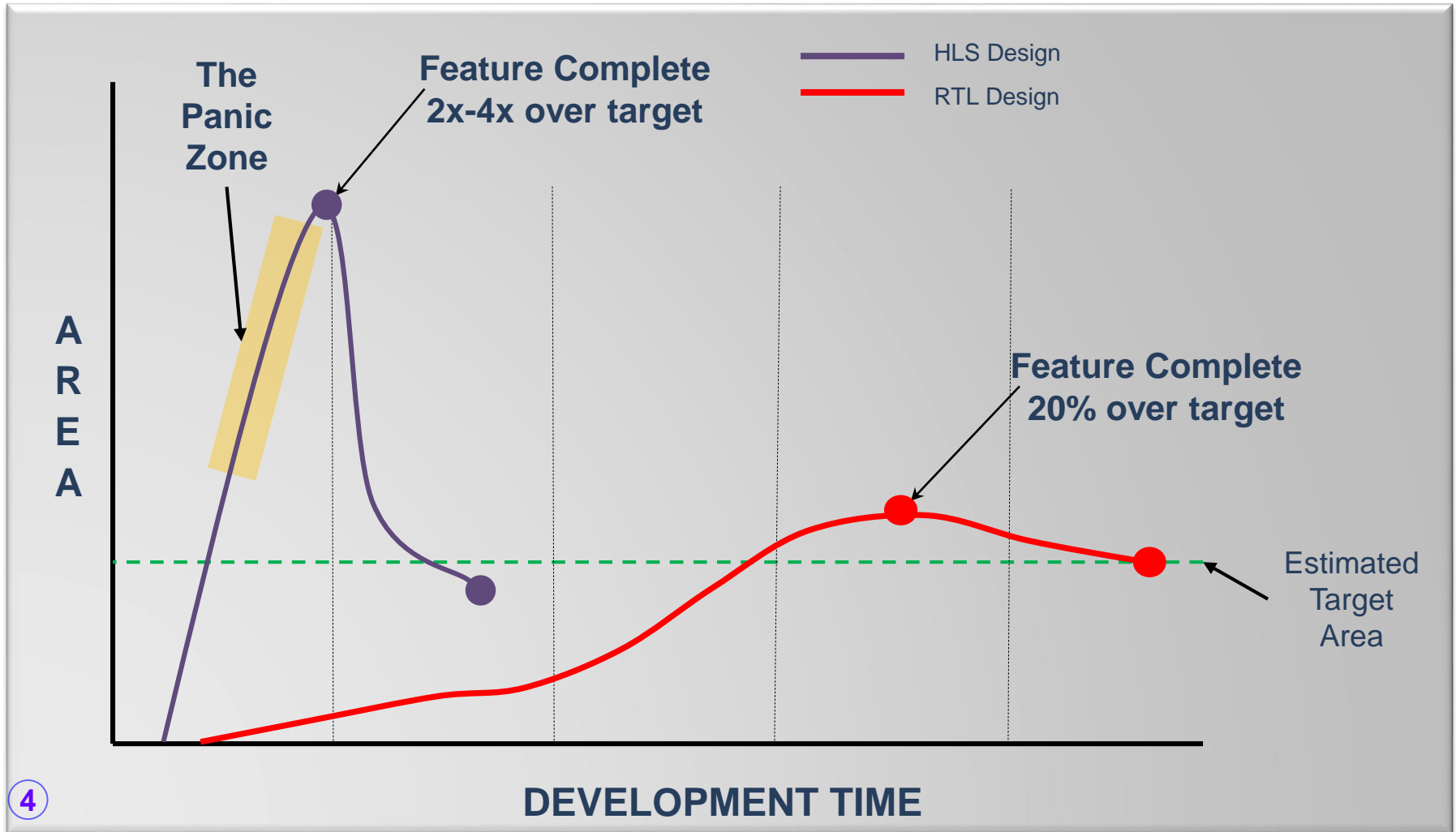
- **Datapath dominated designs**
 - Most of the design is computation oriented
 - Image processing
 - Wireless signal processing
- **Control dominated designs**
 - Most of the design is making decisions and moving data
 - SSD controllers
 - Specialized processor development
 - Network switching applications

- **Block / Module size**
 - Common = 30K → 300K gates
 - Largest we have seen = 750K gates
- **Blocks per Project**
 - Typical
 - Depends on customer stage of adoption
 - Common = 1-8
 - Largest we have seen = 200+ blocks
- **Project size**
 - (implemented with Synthesizer)
 - Largest we have seen = 15M gates
- **QOR (Area)**
 - Typically 5-20% improvement over hand-RTL



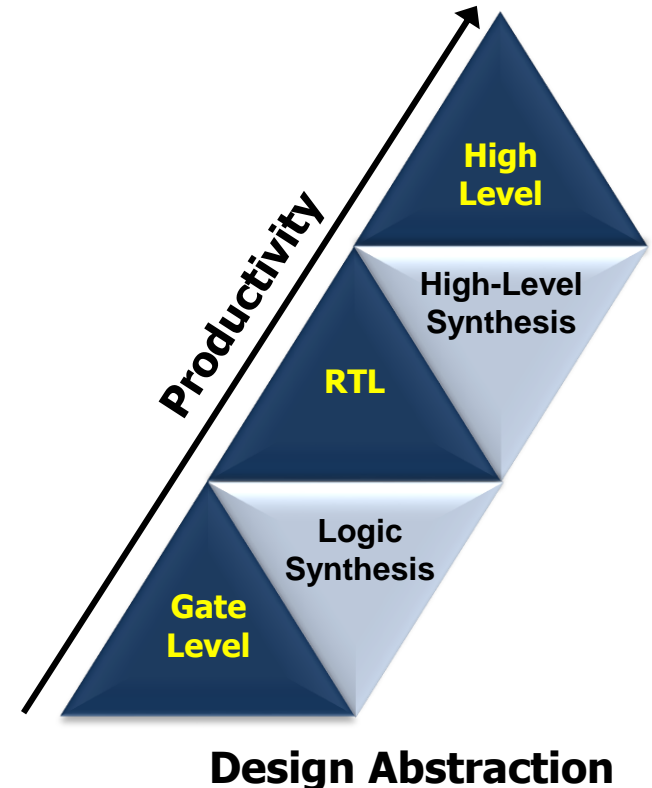
Adopting HLS

Productivity values and expectations



Keys to HLS Raising Abstraction

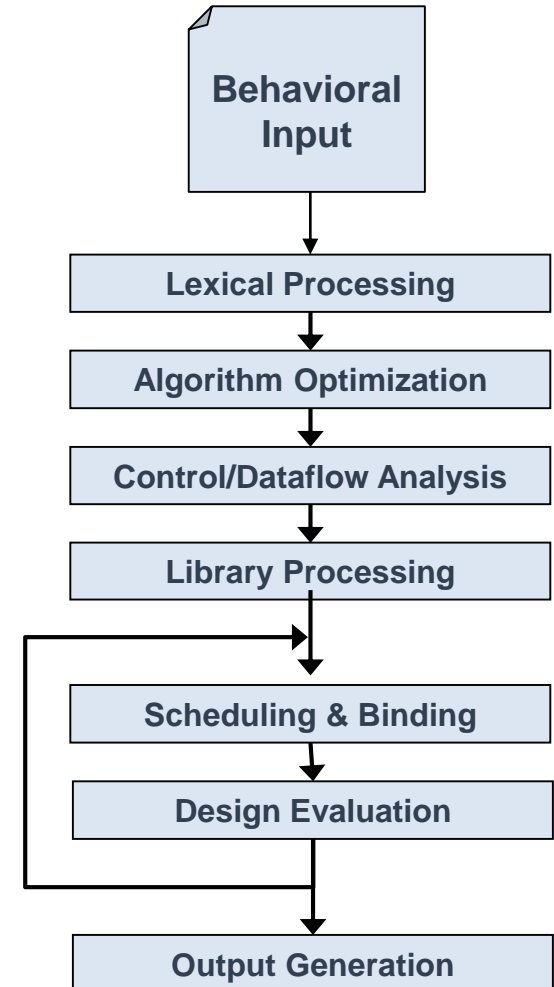
- **Implicit state machine**
 - More like algorithm code
 - Leaves room for exploration
- **Design exploration**
 - Same source can produce designs for different purposes
- **Array handling**
 - Arrays can be implemented in multiple ways
 - HLS schedules memory protocol signals w/ algorithm
- **Handling I/O protocols**
 - HLS schedules I/O protocol signals w/ algorithm
 - Modular interfaces allow protocol encapsulation & reuse



High-level Synthesis Basics

High-level Synthesis Basics

- **High-level synthesis begins with an algorithmic description of the desired behavior**
- **Behavior is expressed in a high level language**
- **SystemC is the input language for Cynthesizer**
- **The second input is the ASIC or FPGA technology library**
 - Cell functions, area and timing
- **The third input is directives for synthesis**
 - Target clock period (required)
 - Other directives: pipelining, latency constraints, memory architecture



High-level Synthesis Example

Consider the following behavioral description to be synthesized for a 100MHz clock period:

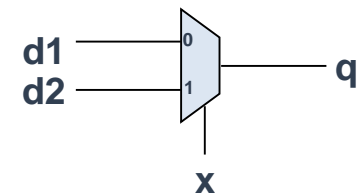
```
typedef unsigned char bit8;  
unsigned long example_func( bit8 a, bit8 b, bit8 c,  
bit8 d, bit8 e )  
{  
    unsigned long y;  
    y = ( ( a * b ) + c ) * ( d * e );  
    return y;  
}
```

- **This description is *un-timed*:** The user does not specify the clock boundaries
 - Does not specify the latency
 - Does not specify which values are registered or when to compute each value
- **Cynthesizer will map the SystemC/C++ operations ('*', '+') to datapath components, and add clock boundaries**
 - Driven by target clock period and other directives
 - Results in known latency and registers
 - Depends on technology library

High-level Synthesis Example: Control & Data Flow Graph

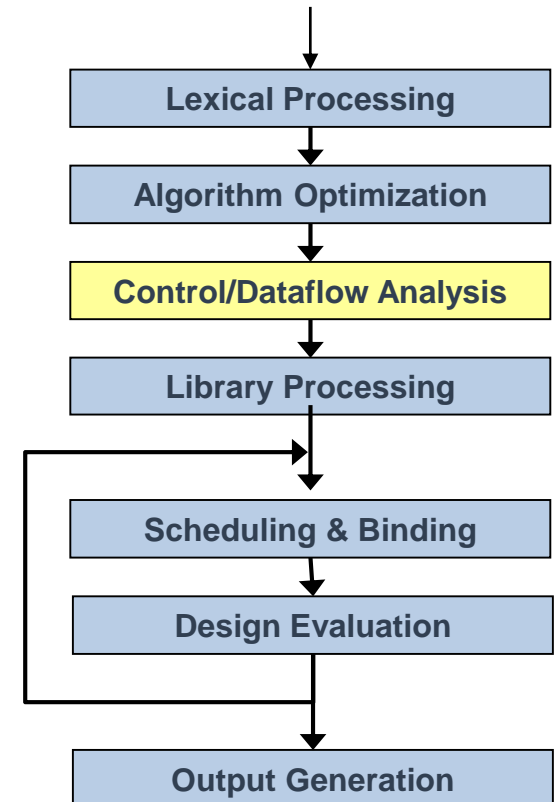
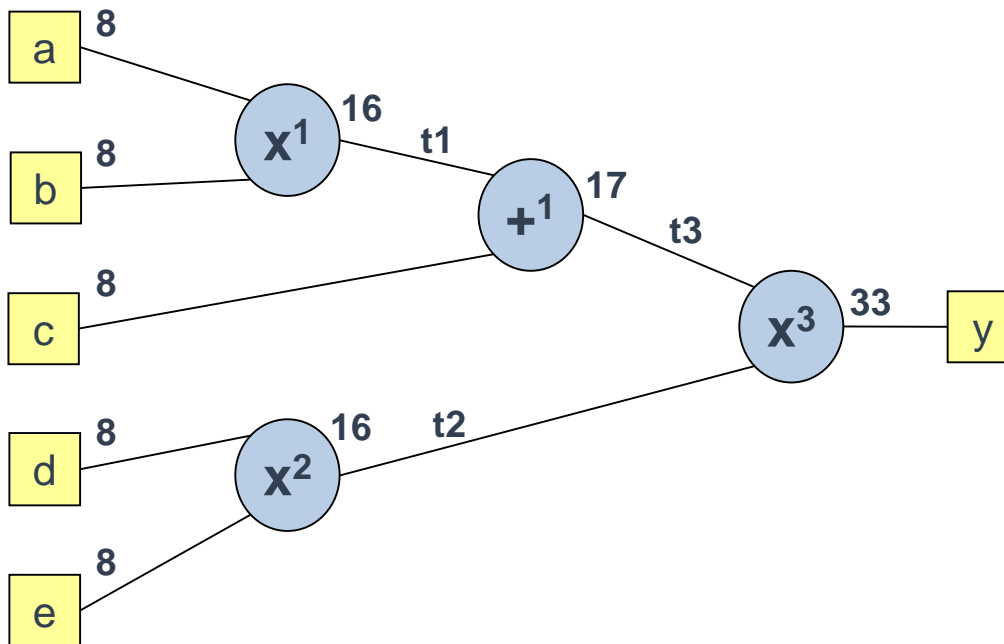
- **Cynthesizer analyzes the source and determines:**
 - Inputs & Outputs
 - Operations
 - Data dependencies
- **A control/dataflow graph (CDFG) is constructed:**
 - With no timing information
 - Represents the data flow for the functionality of the design
- **The graph shows data dependencies:**
 - Operations that must complete before others can begin
 - e.g. $(a*b)$ must complete before we can calculate $(a*b)+c$
- **The graph shows control dependencies:**
 - `if/else` constructs will turn into select operations

```
if (x == 0)
    q = d1;
else
    q = d2;
```



High-level Synthesis Example: Control & Data Flow Graph

The control/dataflow graph (CDFG) analysis would construct the following structure from `example_func()`:



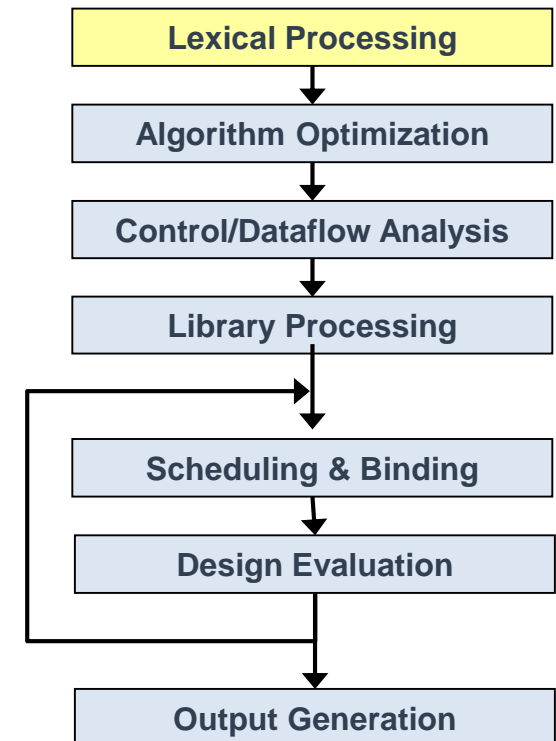
High-level Synthesis Example: Library Processing

Assume the characterized library has the following functional units available

Functional Unit	Delay	Area
8x8=16	2.78 ns	4896.5
16+16=17	1.99 ns	1440.3
20x20=40	5.88 ns	27692.6

An initial allocation might be:

- Two “8x8=16” multipliers
- One “16+16=17” adder
- One “20x20=40” multiplier

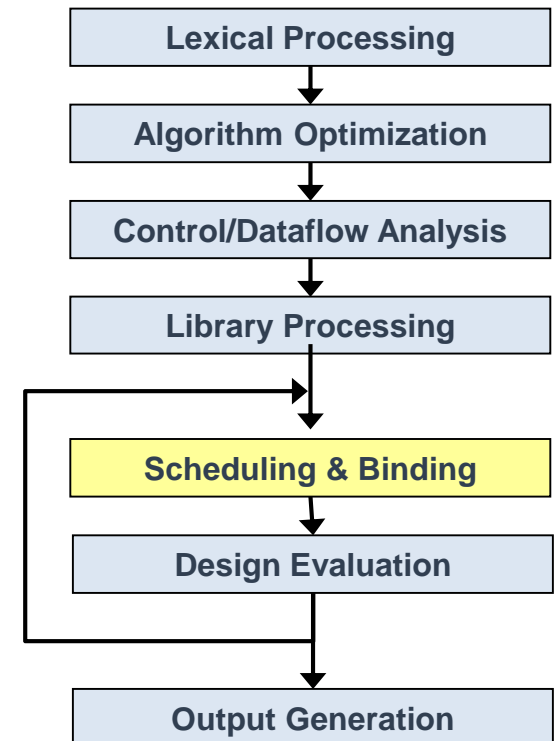


High-level Synthesis Example: Scheduling

- Using the initial allocation the initial schedule would be:

Operator	# Needed	Cycle1	Cycle2	Cycle3
8x8=16	2	t1=a*b t2=d*e		
16+16=17	1		t3=t1+c	
20x20=40	1			y=t2*t3

- Total functional unit area =**
 $(2 * 4896.5) + 1440.3 + 27692.6 = 38925.9$
- Delays fit within 100 MHz clock**

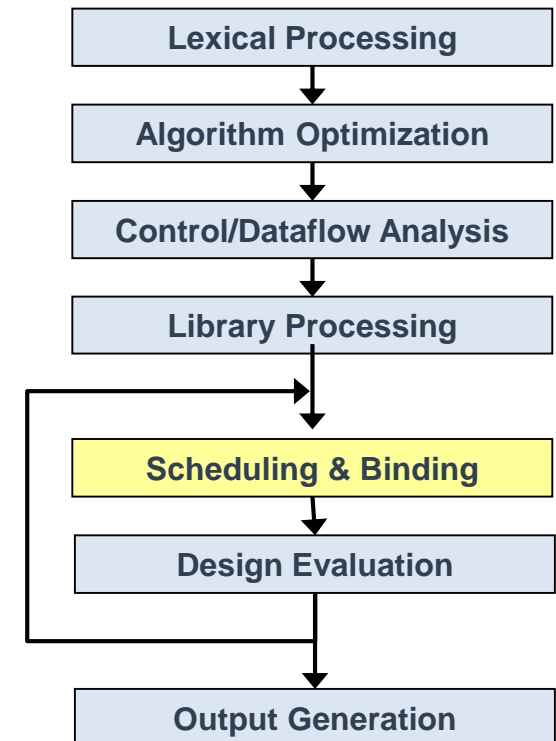


High-level Synthesis Example: Scheduling (cont.)

Scheduling notes that the $d * e$ operation can be moved to cycle 2 to eliminate one 8×8 multiply operator

Operator	# Needed	Cycle1	Cycle2	Cycle3
$8 \times 8 = 16$	1	$t1 = a * b$	$t2 = d * e$	
$16 + 16 = 17$	1		$t3 = t1 + c$	
$20 \times 20 = 40$	1			$y = t2 * t3$

- **Total functional unit area =**
 $4896.5 + 1440.3 + 27692.6 = 34029.4$
- **Delays fit within 100 MHz clock**

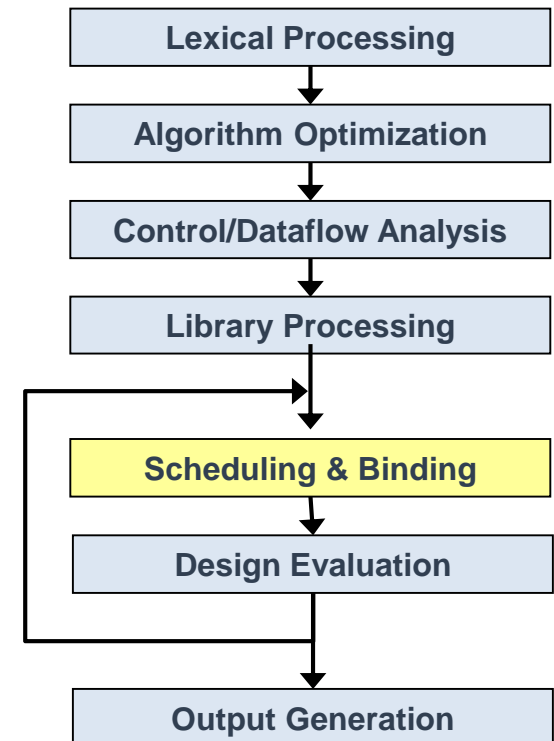


High-level Synthesis Example: Binding

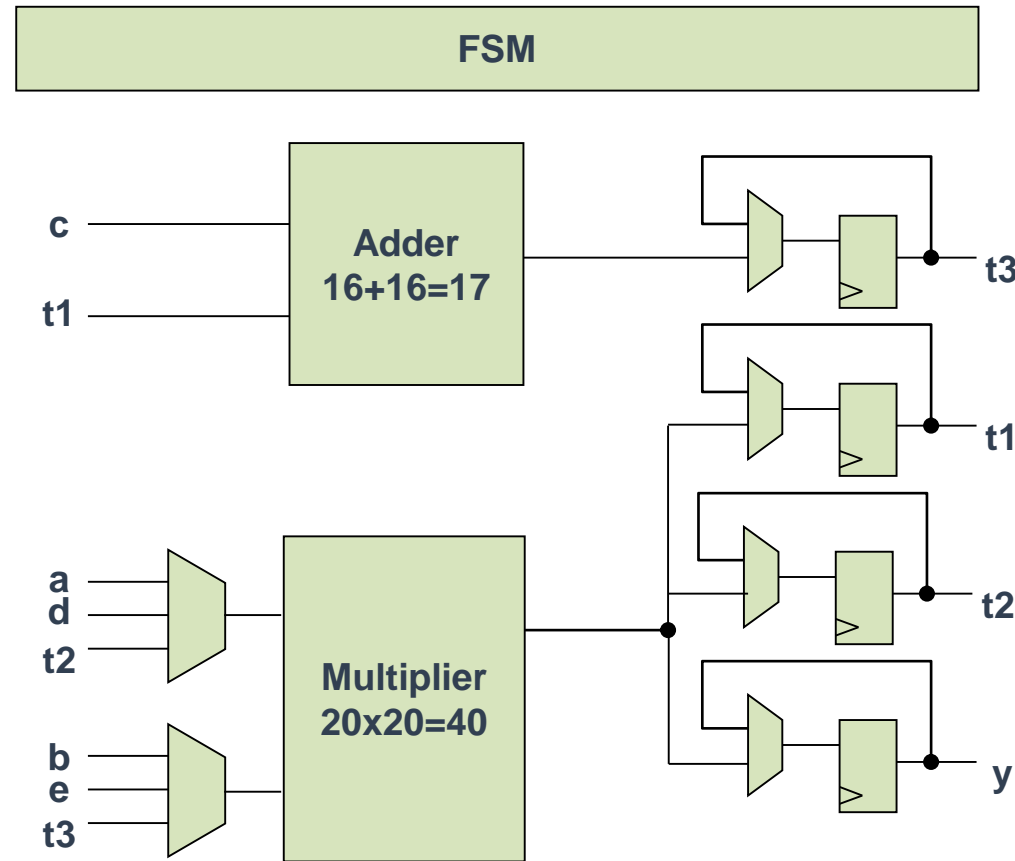
- **Binding assigns specific instances of functional units to the required operators**
- **Further optimization binds the 8x8 multiply operations to the 20x20 multiplier instance**

Operator	# Needed	Cycle1	Cycle2	Cycle3
16+16=17	1		t3=t1+c	
20x20=40	1	t1=a*b	t2=d*e	y=t2*t3

- **Total functional unit area =**
 $1440.3 + 27692.6 = 29132.9$
- **Delays fit within 100 MHz clock**

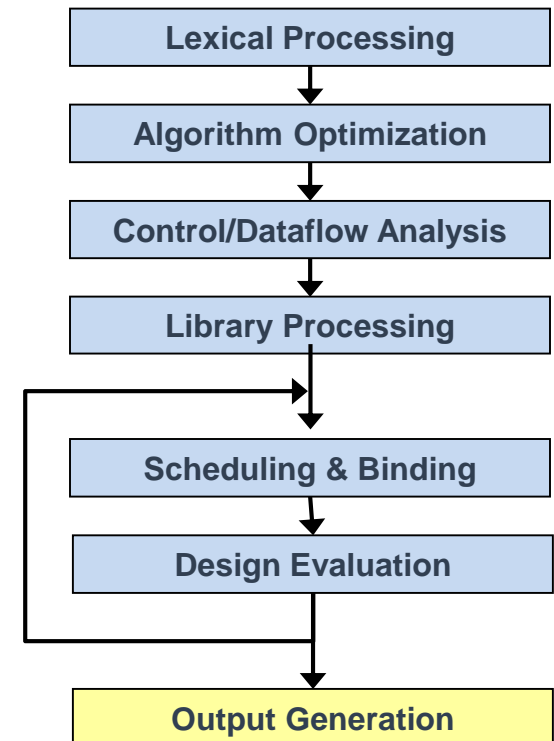


High-level Synthesis Example: Output Hardware Structure



Mux selects come from the FSM

Note: Implicit state machine input produces explicit state machine output





Introduction To The SystemC Synthesizable Subset Draft Under Development

March 2013

SystemC Synthesizable Subset Work

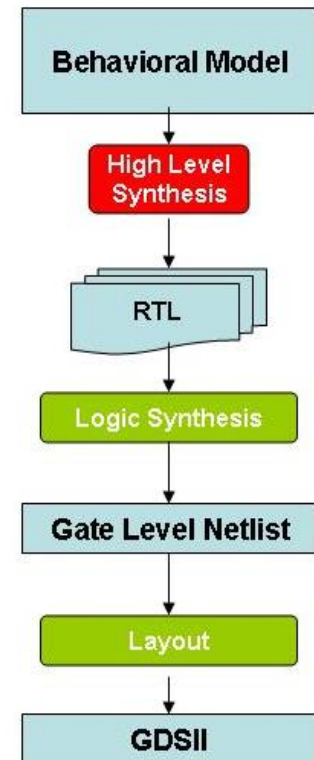
- Development of a description of a synthesizable subset of SystemC
- Started in the OSCI Synthesis Working Group
- Current work is in Accellera Systems Initiative Synthesis Working Group
- Many contributors over a number of years
- Broadcom, Cadence, Calypto, Forte, Fujitsu, Global Unichip, Intel, ITRI, Mentor, NEC, NXP, Offis, 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 Planned 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

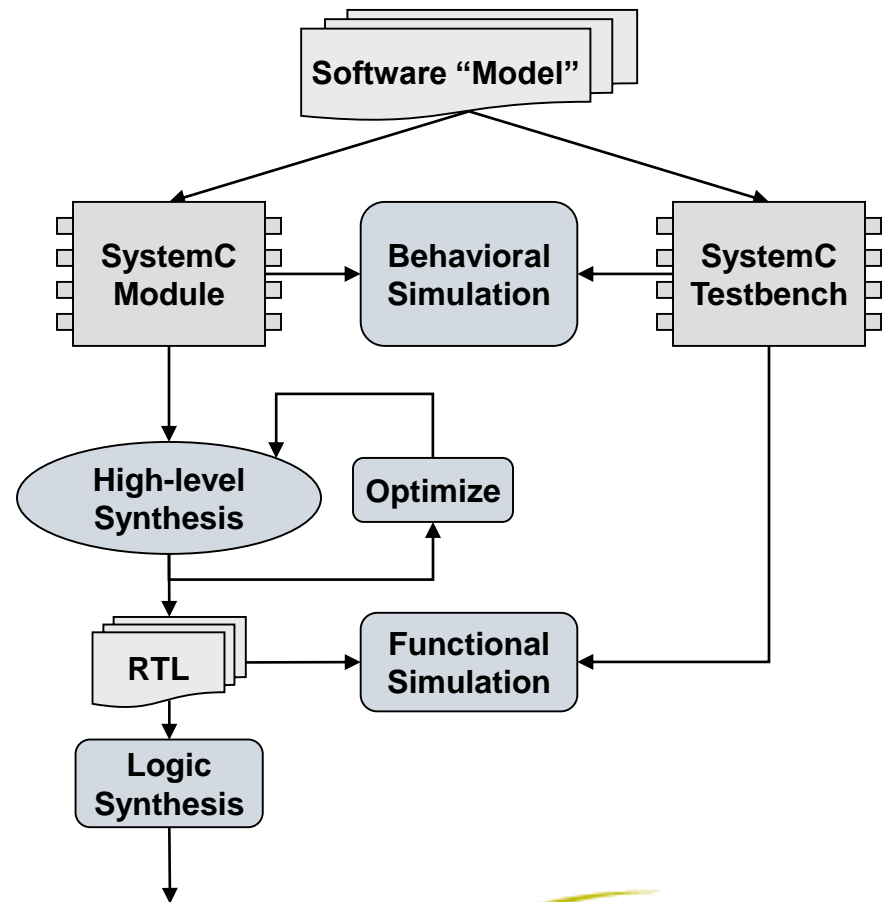
- Modules
- Processes
 - SC_CTHREAD
 - SC_THREAD
 - SC_METHOD
- Reset
- Signals, ports, exports
- SystemC datatypes

C++ Elements

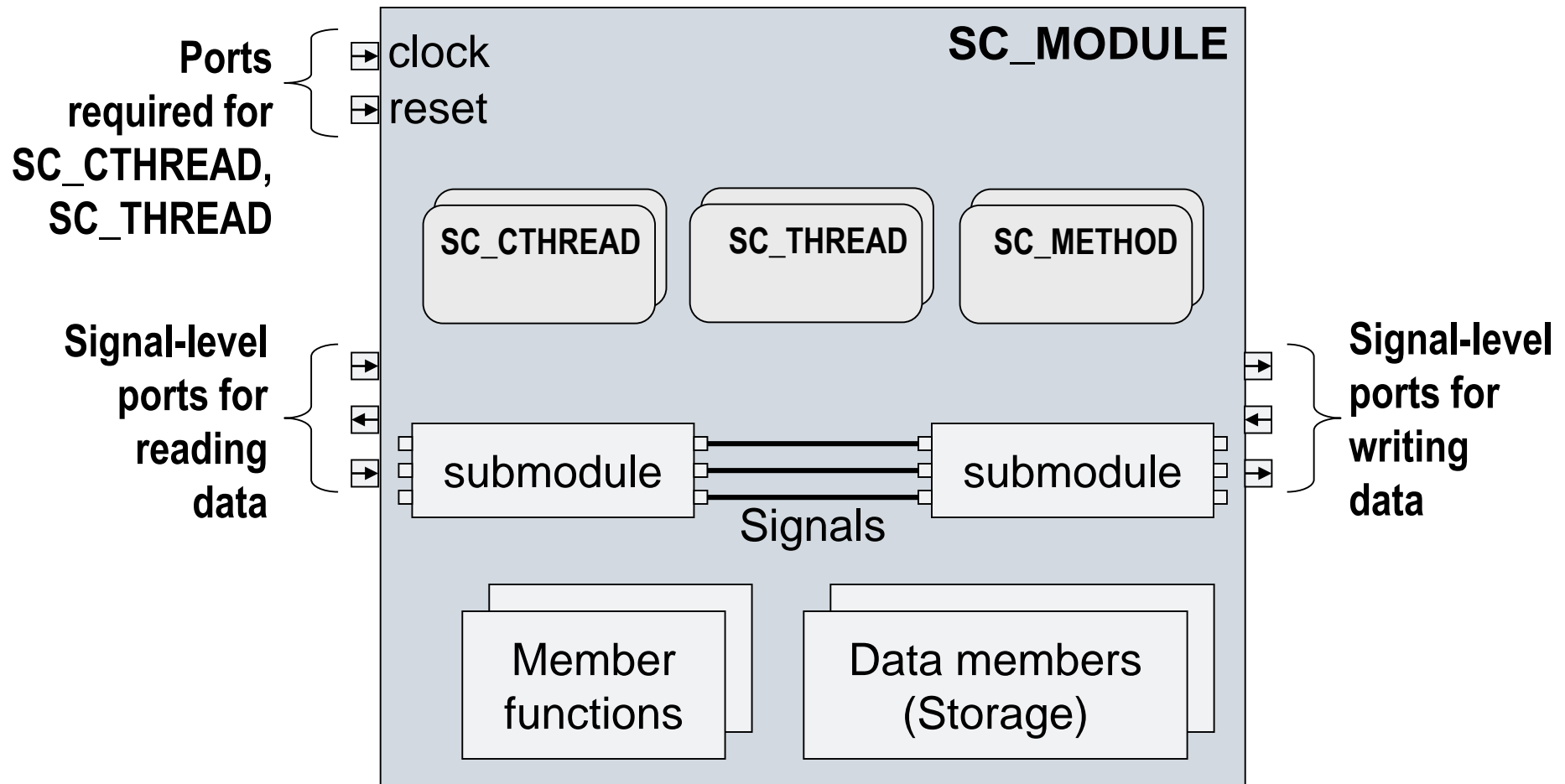
- C++ datatypes
- Expressions
- Functions
- Statements
- Namespaces
- Classes
- Overloading
- Templates

High-level Synthesis With SystemC 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

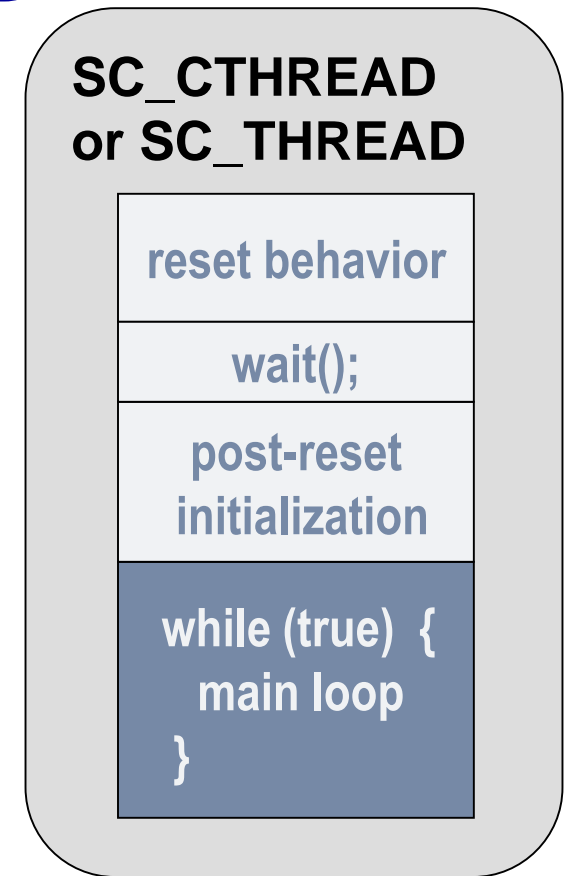
```
// A module declaration
SC_MODULE( my_module1 ) {
    sc_in< bool> X, Y, Cin;
    sc_out< bool > Cout, Sum;
    SC_CTOR( my_module1 ) {...}
};
```

- 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_HAS_PROCESS( my_module1 );
    my_module1(const sc_module_name name )
        : sc_module(name)
    {...}
};
```

SC_THREAD & SC_CTHREAD Reset Semantics

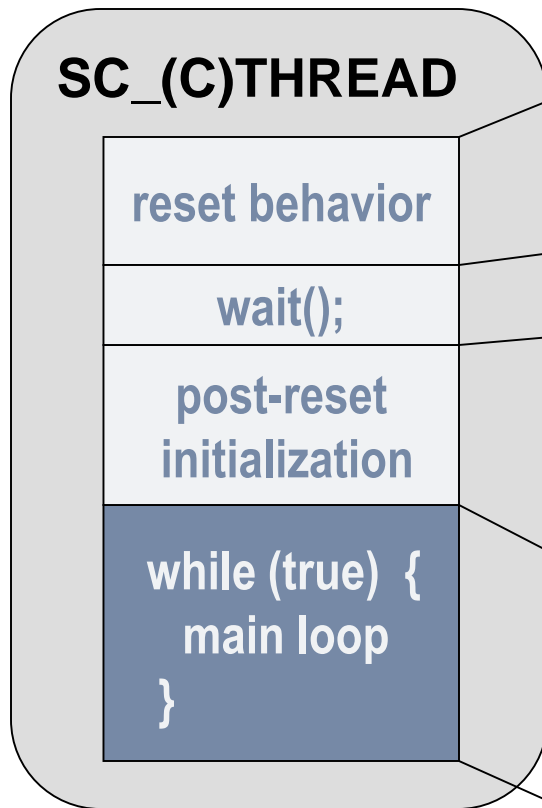
- At start_of_simulation each SC_THREAD and SC_CTHREAD function is called
 - It runs until it hits a wait()
- When an SC_THREAD or SC_CTHREAD is restarted after any wait()
 - If reset condition is false
 - ◆ execution continues
 - If reset condition is true
 - ◆ stack is torn down and function is called 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();  
    }  
}
```

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

- Simple signal/port and level

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

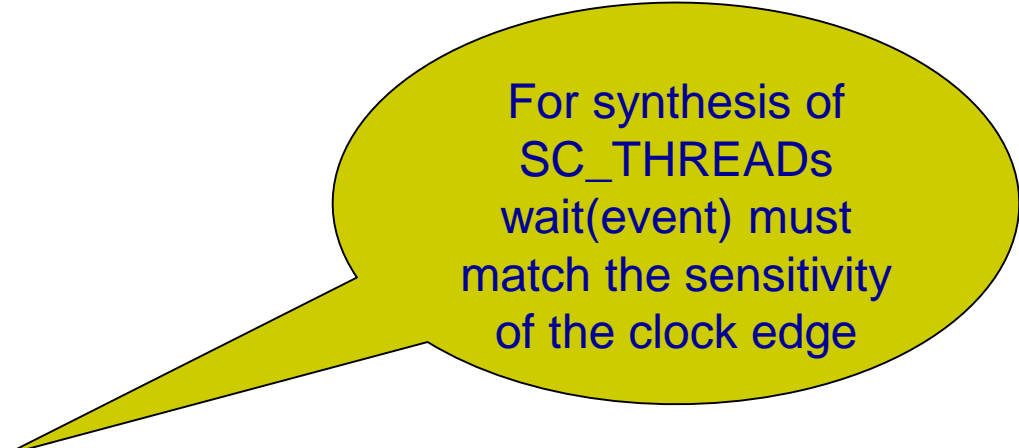
```
SC_THREAD( func );  
sensitive << clk.pos();  
reset_signal_is( reset, true );  
areset_signal_is( areset, true );
```

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

```
inline void reset_signal_is( const sc_in<bool>& port, bool level );  
inline void reset_signal_is( const sc_signal<bool>& signal, bool level );  
inline void areset_signal_is( const sc_in<bool>& port, bool level );  
inline void areset_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`
- **“X” has limited support for synthesis**
 - A tool MAY use “X” to specify an explicit don't-care condition for logic synthesis
- **“Z” has limited support for synthesis**
 - Supported only if it appears in an expression assigned directly to a port variable

Pointers

■ Supported for synthesis

- “this” pointer
- “Pointers that are statically resolvable are supported for synthesis. 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

- **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**

Join A Working Group And Contribute!

The screenshot shows the Accellera Systems Initiative website. The Accellera logo is at the top left. Below it is a navigation bar with links: ABOUT US, TECHNICAL ACTIVITIES, DOWNLOADS, COMMUNITY, NEWS & EVENTS, FORUMS, and WORKSPACE. The 'TECHNICAL ACTIVITIES' link is circled in red. A dropdown menu is visible under 'TECHNICAL ACTIVITIES' with options: Overview, Technical Committees, IEEE Activities, and Acronyms and Definitions. The 'Technical Committees' option is highlighted. To the right of the navigation bar is a Google Custom Search box with a 'SEARCH' button. Below the navigation bar, the breadcrumb trail reads: Activities » Technical Committees » SystemC Synthesis. The main content area is titled 'SystemC Synthesis Working Group (SWG)' and 'Charter'. It states: 'This group is responsible for the definition of a synthesizable subset of SystemC.' It lists the Chair as Andres Takach, Mentor Graphics, and the Vice-Chair as Michael Meredith, Forte Design Systems. The 'Background' section states: 'In August 2009, this group released the Synthesis Subset Draft 1.3 standard synthesis subset draft for public review. The draft features several technical updates. Supported language constructs are now established, and a chapter on processes, clocks, and resets has been added. The draft also includes a discussion on abstraction levels that puts the concepts of the synthesizable subset in the context of the abstraction levels defined for TLM.' It also mentions: 'Public review of the draft is now closed. The draft is available for [download here](#).' The 'Join this Working Group' section states: 'If you are an employee of a member company and would like to join this working group, [click here](#) (requires login) and click Join Group. WG participation requires right of entry by the group chair.' The 'click here' link is circled in red. On the right side, there is a 'QUICK LINKS' section with two links: 'Download SystemC Synthesizable Subset Draft 1.3' and 'Group working area'. A sidebar on the left lists various SystemC components: Overview, Technical Committees, Interface, IP Tagging, IP-XACT, Open Verification Library (OVL), SystemC Analog/Mixed-Signal, SystemC Configuration, Control & Inspection (CCI), SystemC Language, SystemC Synthesis, SystemC Transaction-level Modeling (TLM), SystemC Verification, and Unified Coverage Interoperability Standard (UCIS).

Synthesizability Recommendations & Extensions

Pointer Support Recommendations

- **HLS tools can handle pointers in the C++ code**

```
// Passing a pointer to array
void func1(int *a) {
    for (i = 0; i < N; i++)
        sum += a[i];
}
```

- **But a pointer needs to be resolved to a specific array (or variable) in the design at compile time**

**// OK to call with a determinate
// pointer**

```
ptr = arr1;
func1(ptr);
ptr = arr2;
func1(ptr);
```

**// Not OK to call with an
// indeterminate pointer**

```
if (external_input.read() == 1)
    ptr = arr1;
else
    ptr = arr2;
// array used depends on
// external input
func1(ptr); // ERROR
```

Reset Logic Recommendations

- **The reset logic code block *should*:**
 - Write the reset value to all output ports
 - e.g. `my_port.write(0)` for a standard SystemC `sc_out` port
 - Call the `reset()` function of all modular interface ports that have one
 - `cynw_p2p` ports (`din.reset()`, `dout.reset()`)
 - Memory ports for explicit memories
 - Write the reset value to all variables
 - An array cannot be initialized in one cycle unless it is flattened
- **The reset logic code block *should not*:**
 - Access a memory (e.g. an un-flattened array)
 - Call a function that calls `wait()`
(e.g. cannot call the `get()` function of a `cynw_p2p` input port)
 - Such multi-cycle code should be executed after the reset protocol block

Data Type Recommendations

- **Correct sizing of variables will have a big impact on area**
 - Particularly on ports!
 - Affects not just the variable but all operations on it
- **Cynthesizer can often *automatically* determine the optimum bit width for a variable**
 - *Only if* it can determine the maximum value
- **Replacing C++ built-in types with `sc_int` or `sc_uint` types enforces desired sizing of hardware**
 - It helps when Cynthesizer cannot determine maximum values
 - Allows selection and testing of individual bits and ranges


```
sc_uint<32> status; // 32-bit register
if ( status[7] == 1) // test individual bits
    count = status.range(3,0); // get a range of bits
```
- **C++ semantics sometimes force data values to 64 bits**
 - Casting to a narrower type can avoid this

- ◆ **Cynthesizer supports synthesis of fixed point arithmetic**

```
sc_fixed<wl, iwl, q_mode, o_mode, n_bits> Object_Name ;
```

```
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> Object_Name ;
```

wl : “word length” Total number of bits used

iwl : “integer word length.” Number of bits to left of the decimal point

q_mode : “quantization mode” Represents the process to be done when the value can not be represented precisely.

o_mode : “overflow mode” (SC_WRAP、SC_SAT, ,)

n_bits : is used in overflow processing

- ◆ **Forte provides an sc_fixed compatible class for improved simulation speed**

- ◆ **Cynthesizer supports floating point types**
 - Use `cynw_cm_float<>` an optional library if it is necessary to synthesize float/double.

```
cynw_cm_float<E,M,D,N,EX,R> object_name、 ... ;
```

E : The number of bits for exponent (For single precision:8)

M : The number of bits for Mantissa (For singleprecision:23)

D : 1 use denormal numbers for gradual underflow (default 1)

N : When set to 1 , interpret NaN, Inf according to IEEE 754 (default 1)

EX :When set to 1, generate IEEE 754 exceptions (default 0)

R : Rounding (Default : CYNW_NEAREST)

Example :

```
#include "cynw_cm_float.h"
```

```
cynw_cm_float<8,23> a, b, c; // IEEE Single Precision Format
```

```
...
```

```
a = b + c;
```

```
a = b * c;
```


- **Cynthesizer includes a synthesizable complex datatype**

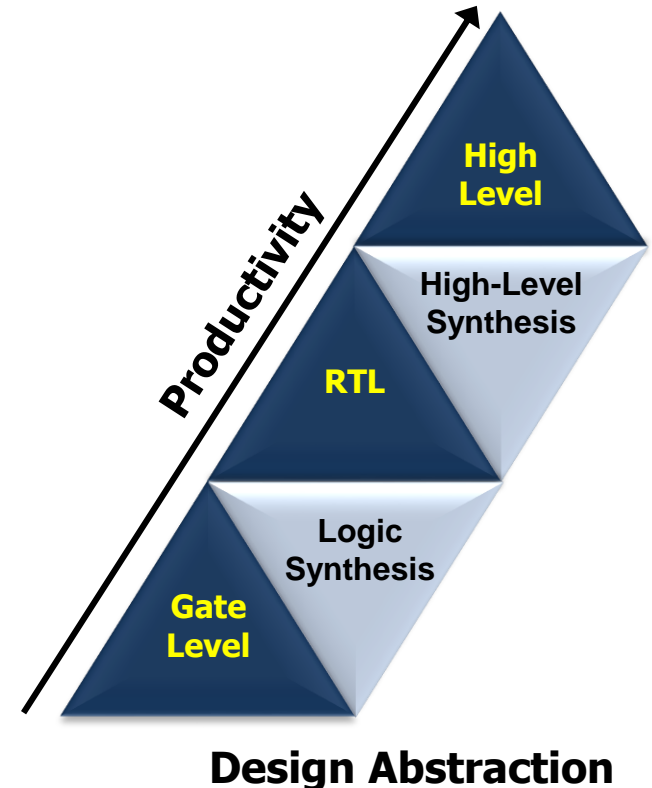
- API compatible with `std::complex`
- Just use `std::complex` in your code, the synthesizable implementation will be used

```
#include <complex>
typedef cynw_fixed< W, L, SC_RND > my_fp;
typedef std::complex< my_fp > my_cplx;
. . .
my_cplx out, a, b, c, d;
out = (a * b) + (c * d);
. . .
```

Keys To Raising Abstraction Level

Keys to Raising Abstraction

- **Implicit state machine**
 - More like algorithm code
 - Leaves room for exploration
- **Design exploration**
 - Same source can produce designs for different purposes
- **Array handling**
 - Arrays can be implemented in multiple ways
 - HLS schedules memory protocol signals w/ algorithm
- **Handling I/O protocols**
 - HLS schedules I/O protocol signals w/ algorithm
 - Modular interfaces allow protocol encapsulation & reuse

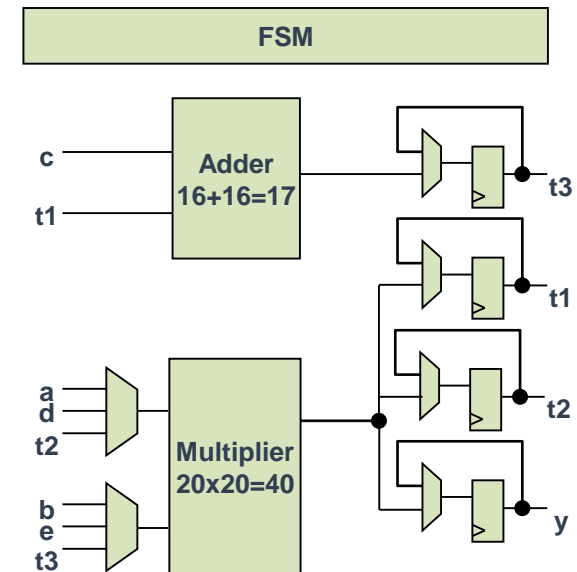
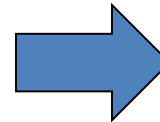


Implicit State Machine vs Explicit State Machine

- **Implicit state machine form**

- Has a higher level of abstraction
- Matches common algorithm description
- Can be mapped to multiple implementations with different performance characteristics

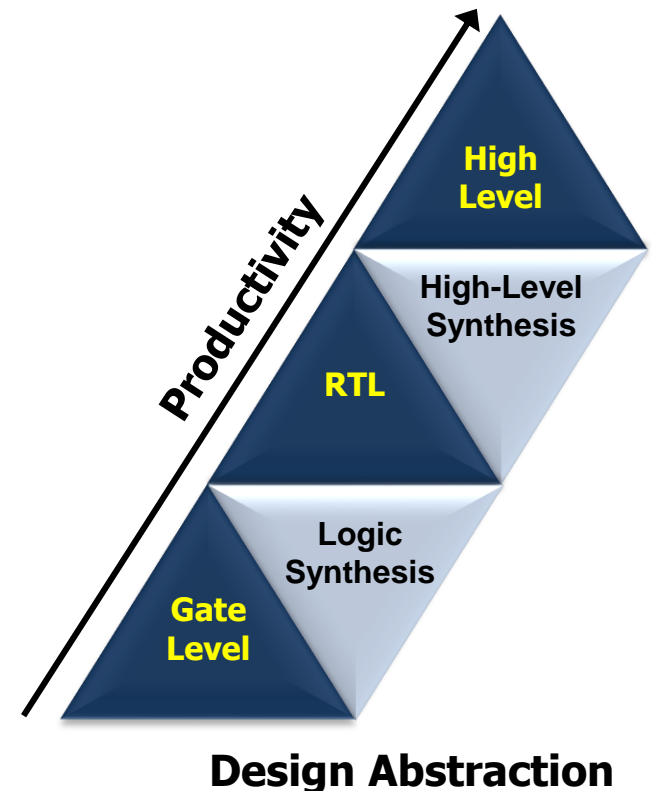
```
typedef unsigned char bit8;  
unsigned long example_func( bit8 a,  
                           bit8 b, bit8 c, bit8 d, bit8 e )  
{  
    unsigned long y;  
    y = ( ( a * b ) + c ) * ( d * e );  
    return y;  
}
```



Exploration

Keys to Raising Abstraction

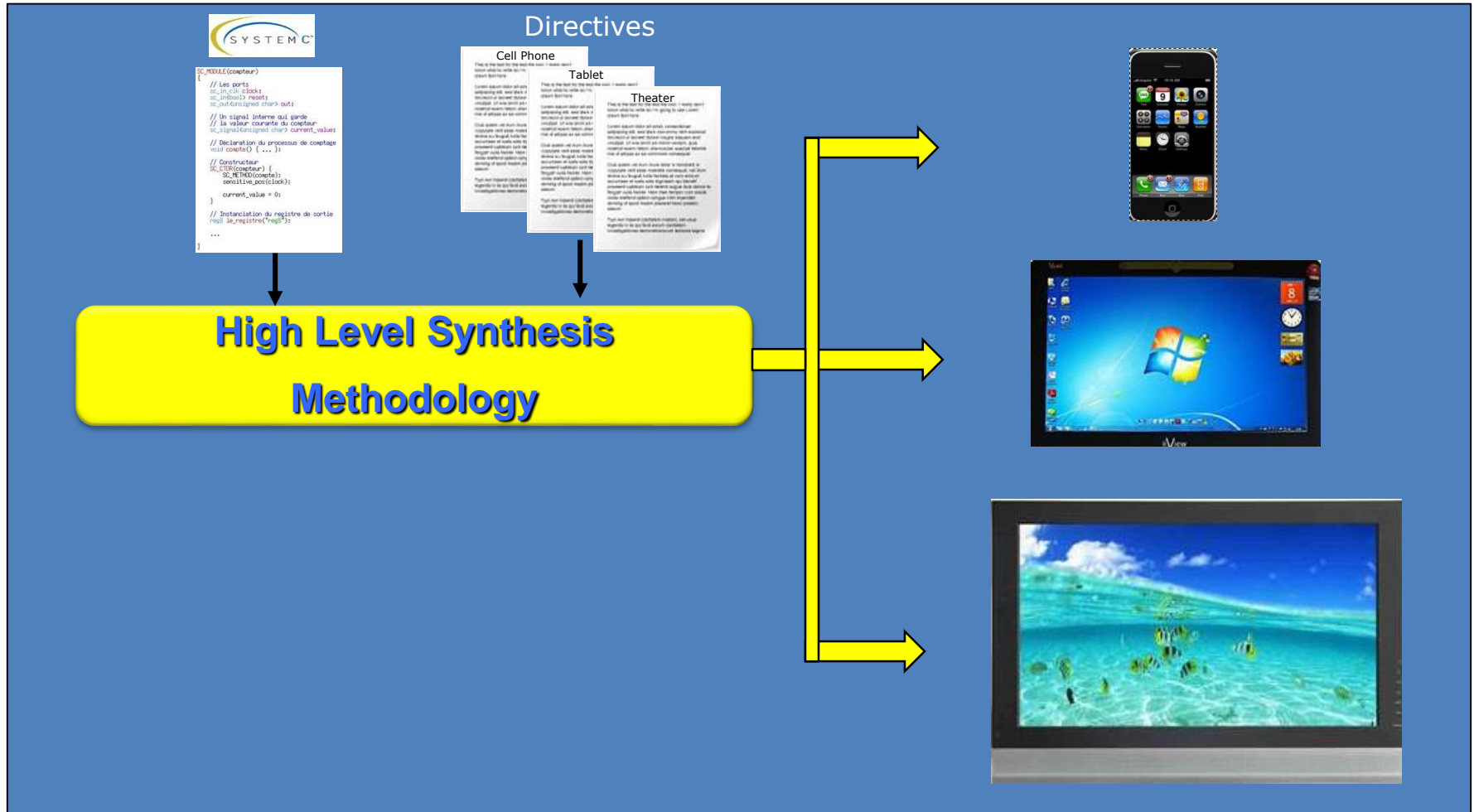
- **Implicit state machine**
 - More like algorithm code
 - Leaves room for exploration
- **Design exploration**
 - Same source can produce designs for different purposes
- **Array handling**
 - Arrays can be implemented in multiple ways
 - HLS schedules memory protocol signals w/ algorithm
- **Handling I/O protocols**
 - HLS schedules I/O protocol signals w/ algorithm
 - Modular interfaces allow protocol encapsulation & reuse



Exploration

One source can be targeted to multiple uses

FORTE
DESIGN SYSTEMS



Courtesy Michael Bohm, Intel

Exploration Example

- **Non-pipelined implementation**
VS
- **Pipelined implementation (1 calculation / 2 cycles)**
VS
- **Pipelined implementation (1 calculation / cycle)**

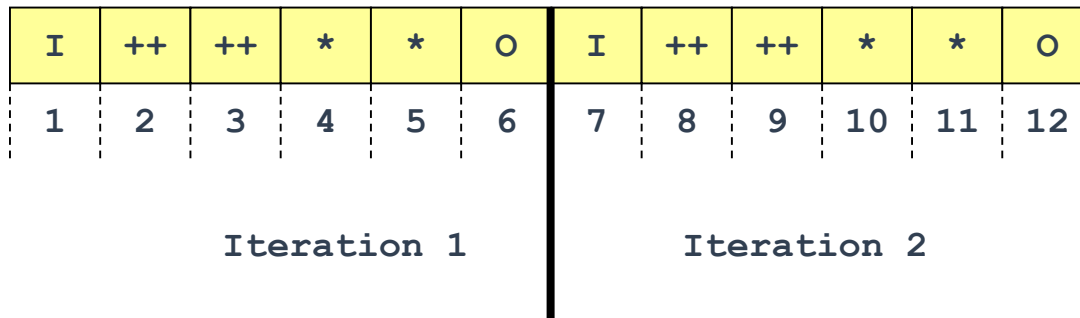
Simple Schedule

Non-pipelined implementation

```
// Read an input, do some calculations, write output
for (i = 0; i < N; i++)
{
    X = inp.get(); // Read input
    // Assume the arrays are flattened, so accesses take
    // zero time
    Y = (A[i] + B[i] + C[i]) * (D[i] + E[i] + F[i]) * X;
    outp.put(Y); // Write output
}
```

```
--- cycle 1 ---
X = inp.get();
--- cycle 2 ---
R1 = A[i] + B[i]
R2 = D[i] + E[i]
--- cycle 3 ---
R3 = R1 + C[i]
R4 = R2 + F[i]
--- cycle 4 ---
R5 = R3 * R4
--- cycle 5 ---
Y = R5 * X
--- cycle 6 ---
outp.put(Y);
```

- **An un-pipelined schedule:**
 - Each iteration takes six cycles



Resources	2 Adders, 1 Multiplier
Latency	6 clock cycles
Throughput	1 I/O per 6 clock cycles

Types of Pipelining

- **Pipelined datapath components**

- A multicycle computation implemented in a datapath component
- E.g. a wide multiplier that takes 3 cycles to complete and can start a new computation every cycle
- If a datapath component is pipelined, Synthesizer will potentially create a schedule that starts a new computation every cycle

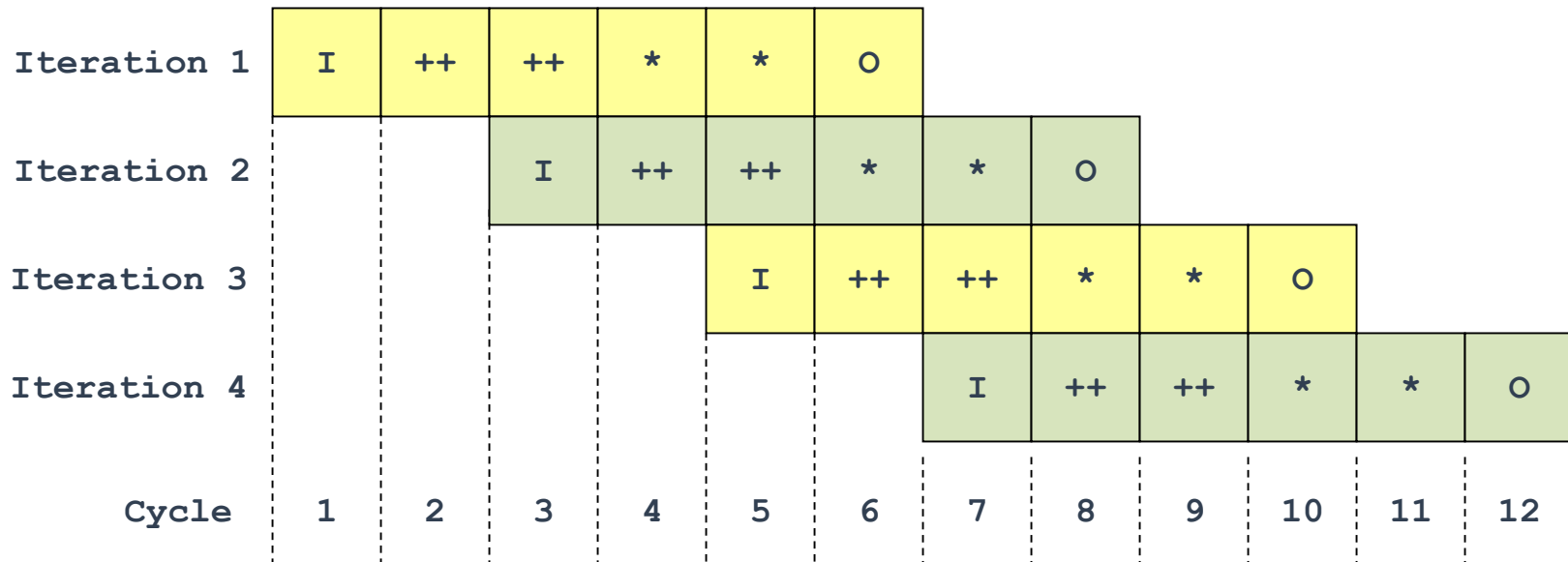
- **Pipelined loops**

- A multicycle computation implemented in the FSM
- E.g. a fir filter that takes 5 cycles to complete a computation, but can start a new computation every cycle
- Very useful technique to improve throughput
- When pipelined, a new loop iteration can start before the previous one ends
- Multiple iterations of the loop are active at the same time

Pipelining Example

- Pipeline the example with an initiation interval of "2"

```
CYN_INITIATE( CONSERVATIVE, 2, "my_pipe" );
```



Resources	2 Adders, 1 Multiplier
Latency	6 clock cycles
Throughput	1 I/O per 2 clock cycles

Throughput tripled
with no increase in
resources!

Pipelining Example

- Pipeline the example with an initiation interval of "1"

```
CYN_INITIATE( CONSERVATIVE, 1, "my_pipe" );
```

Iteration 1	I	++	++	*	*	O						
Iteration 2		I	++	++	*	*	O					
Iteration 3			I	++	++	*	*	O				
Iteration 4				I	++	++	*	*	O			
Iteration 5					I	++	++	*	*	O		
Iteration 6						I	++	++	*	*	O	
Iteration 7							I	++	++	*	*	O
Cycle	1	2	3	4	5	6	7	8	9	10	11	12

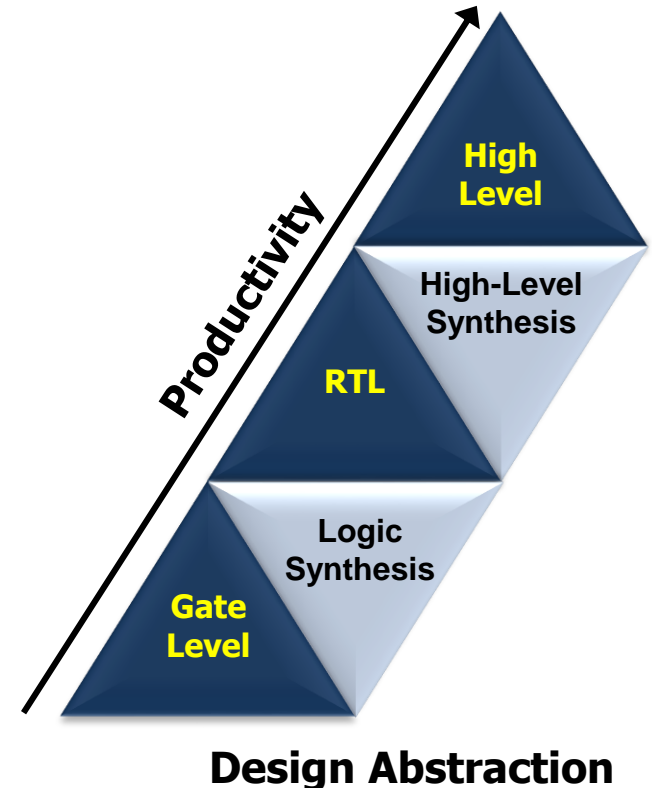
Resources	4 Adders, 2 Multiplier
Latency	6 clock cycles
Throughput	1 I/O per 1 clock cycles

Fully pipelined for a 6x improvement in throughput for only a 2x increase in resources.

HLS and Arrays

Keys to Raising Abstraction

- **Implicit state machine**
 - More like algorithm code
 - Leaves room for exploration
- **Design exploration**
 - Same source can produce designs for different purposes
- **Array handling**
 - Arrays can be implemented in multiple ways
 - HLS schedules memory protocol signals w/ algorithm
- **Handling I/O protocols**
 - HLS schedules I/O protocol signals w/ algorithm
 - Modular interfaces allow protocol encapsulation & reuse



Array Handling

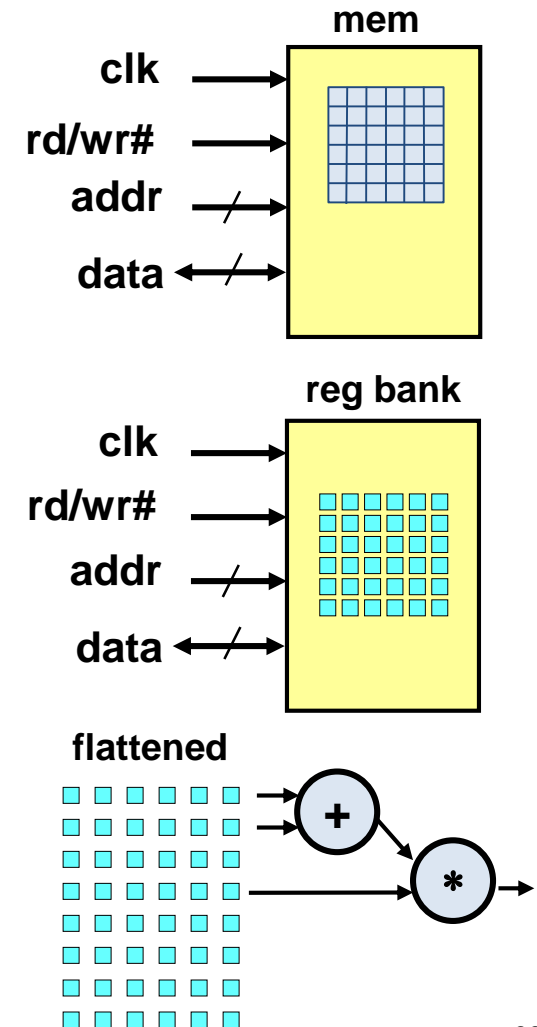
- **Array handling is a key contributor to HLS abstraction**
- **Arrays are accessed using normal C++ syntax**

```
sc_uint<8> array[6][8];  
x = (array[5][0] + array[5][1])* array[5][3];
```

- **Directives control how array is implemented**
- **If array is implemented as memory, HLS determines when to drive each protocol signal**

- **Multiple ways to implement arrays**

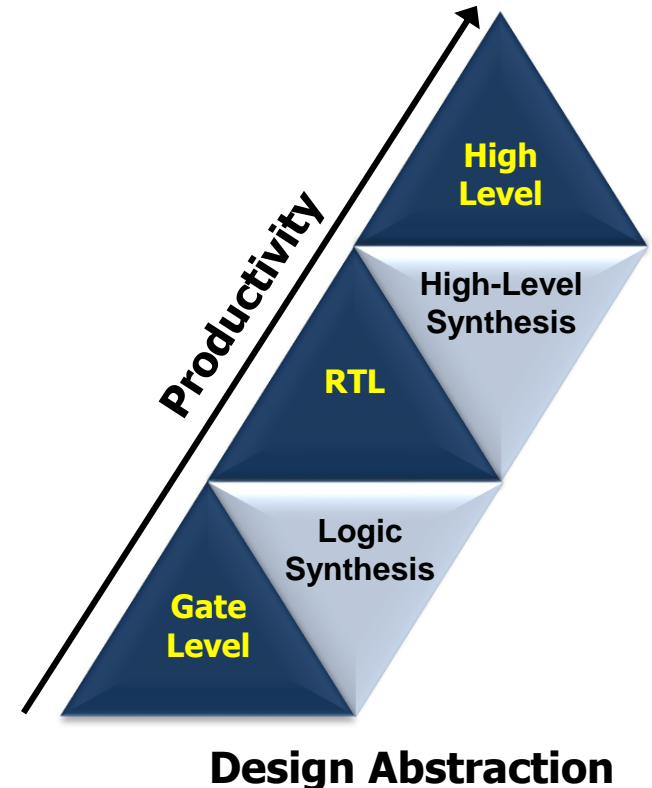
- Memory
 - Storage is RAM or ROM
 - Access through port/protocol
- Register bank
 - Storage is registers
 - Access through port/protocol
- Flattened
 - Each array element is treated as an individual variable
 - Permits unlimited accesses per cycle



HLS and I/O Protocol

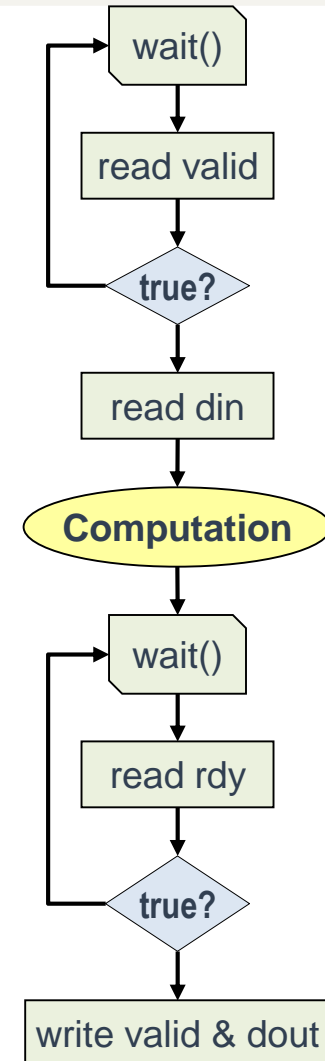
Keys to Raising Abstraction

- **Implicit state machine**
 - More like algorithm code
 - Leaves room for exploration
- **Design exploration**
 - Same source can produce designs for different purposes
- **Array handling**
 - Arrays can be implemented in multiple ways
 - HLS schedules memory protocol signals w/ algorithm
- **Handling I/O protocols**
 - HLS schedules I/O protocol signals w/ algorithm
 - Modular interfaces allow protocol encapsulation & reuse



Protocol Scheduling

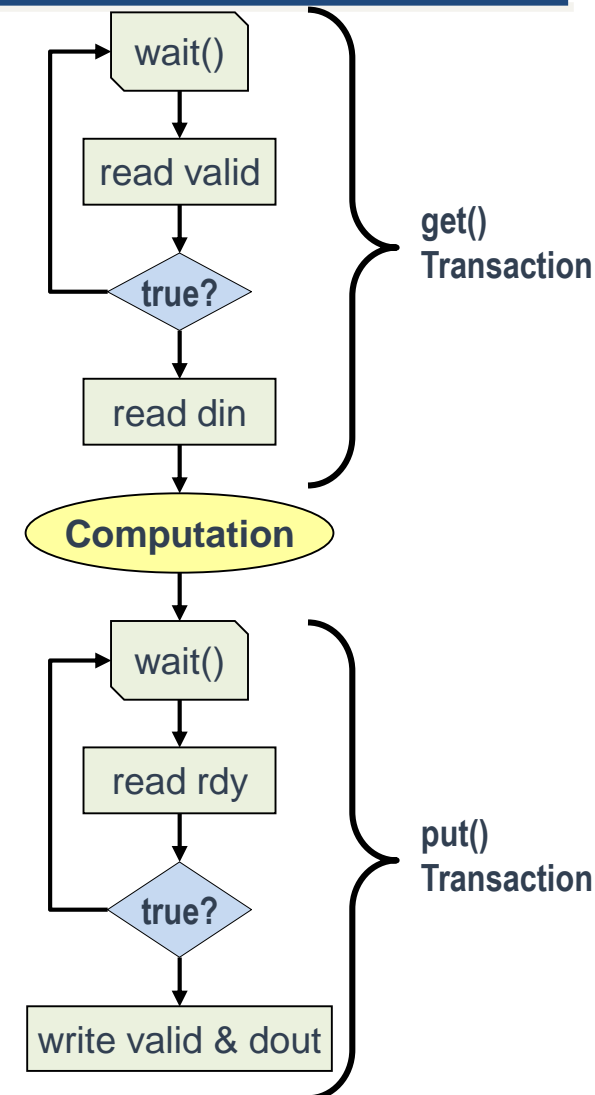
- **Scheduling determines what operations happen in each clock cycle**
- **Protocol scheduling refers to the way scheduling takes I/O operations and wait()s into account**
- **In order to produce a good result, scheduling must satisfy these conditions**
 1. Operations must be kept in order according to the dataflow graph
e.g. in " $A = B + (C * D)$ " the multiplication must be done before the addition
 2. Key relationships between I/O operations and clock edges must be maintained to preserve protocol integrity
e.g. address, data, and write-enable must be asserted in the same cycle for an SRAM write



Transaction Accurate Interface Scheduling

Cynthesizer uses a “transaction accurate” protocol scheduling policy

- **Directives are used to define groups of I/O operations and wait()s as transactions**
- **Within each transaction I/O operations and waits are kept in order as written**
- **Rules determine how transactions can move with respect to each other**
- **Computation operations are allowed to move wherever the dataflow graph will allow**



Code Example Protocol

```
void mymod::thread0( void ) {  
    {  
        CYN_PROTOCOL( "reset_protocol" );  
        out.write( 0 );  
        wait(1);  
    }  
    while( true )  
    {  
        {  
            CYN_PROTOCOL( "read_protocol" );  
            for( int i = 0; i < 8; i++ ) {  
                MEM[i] = in.read();  
                wait(1);  
            }  
        }  
        MEM[0] = MEM[3] + MEM[2] * MEM[1];  
        MEM[4] = MEM[5] / MEM[6];  
        MEM[7] = MEM[0] - MEM[4];  
        {  
            CYN_PROTOCOL( "write_protocol" );  
            for( int i = 0; i < 8; i++ ) {  
                out.write( MEM[7 - i] );  
                wait(1);  
            }  
        }  
    }  
}
```

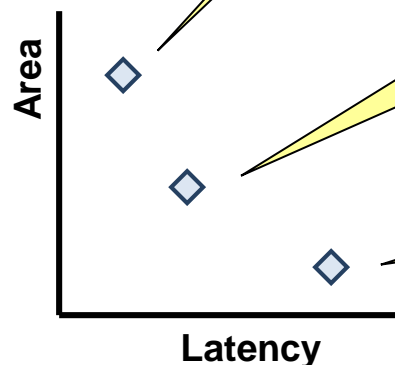
In these protocol blocks, the RTL generated by Cynthesizer maintains the order of I/O operations (fixed I/O)

The protocol accuracy is maintained by the wait() statements embedded into the protocol blocks

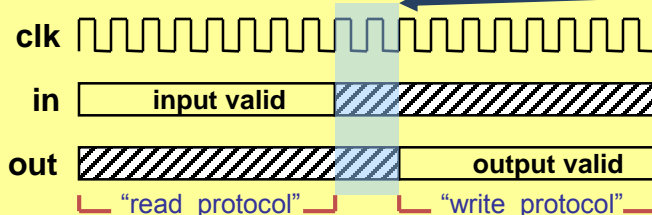
Protocol Results

For all the architectures

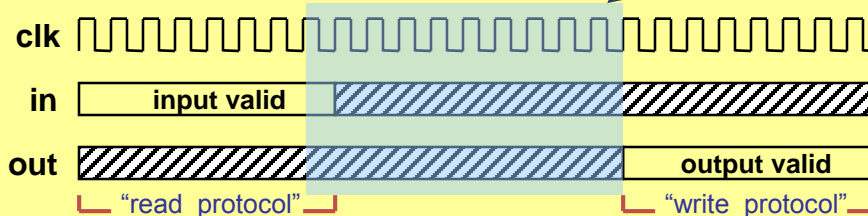
- Input: 8 cycles
- Output: 8 cycles
- These are un-pipelined architecture
- The cycles for processing code between I/O protocol blocks depend on the constraints



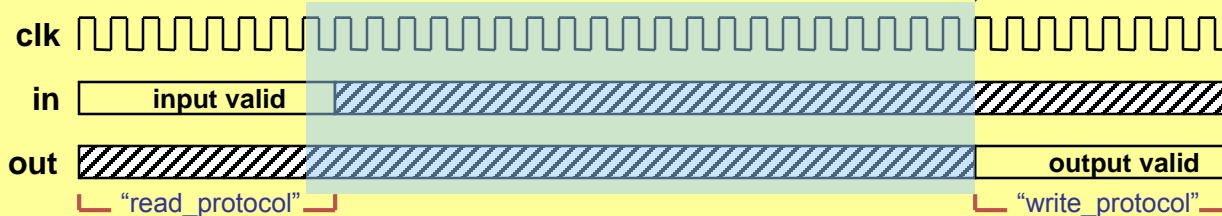
Architecture 1



Architecture 2



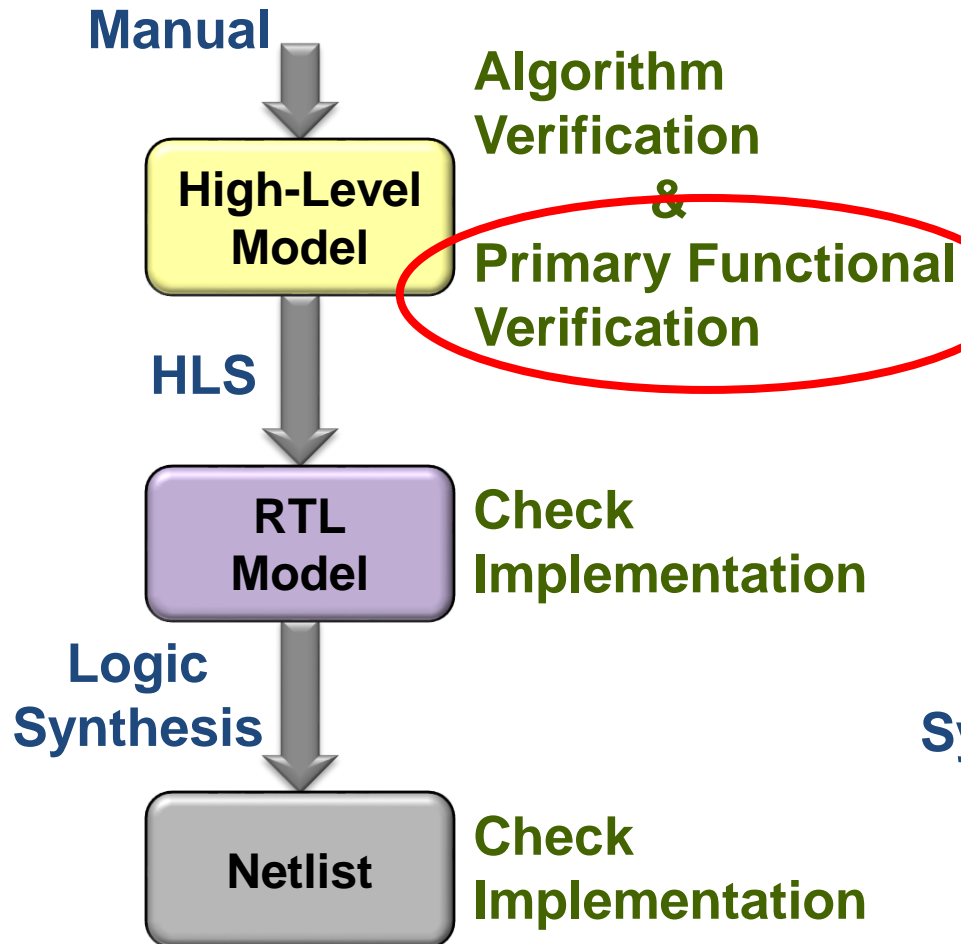
Architecture 3



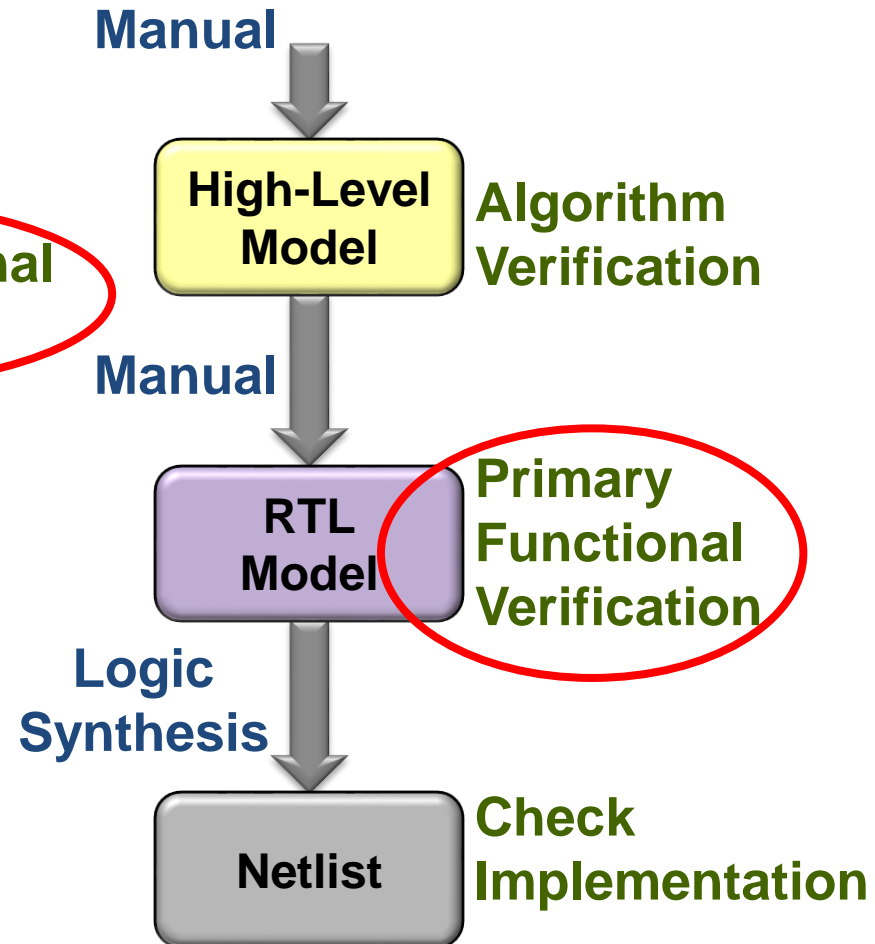
The operations involving the MEM variable gets scheduled (free scheduling) depending on the latency constraints, Library and clock frequency.

Protocol + Algorithm Enables HLS Verification Flow

HLS Flow

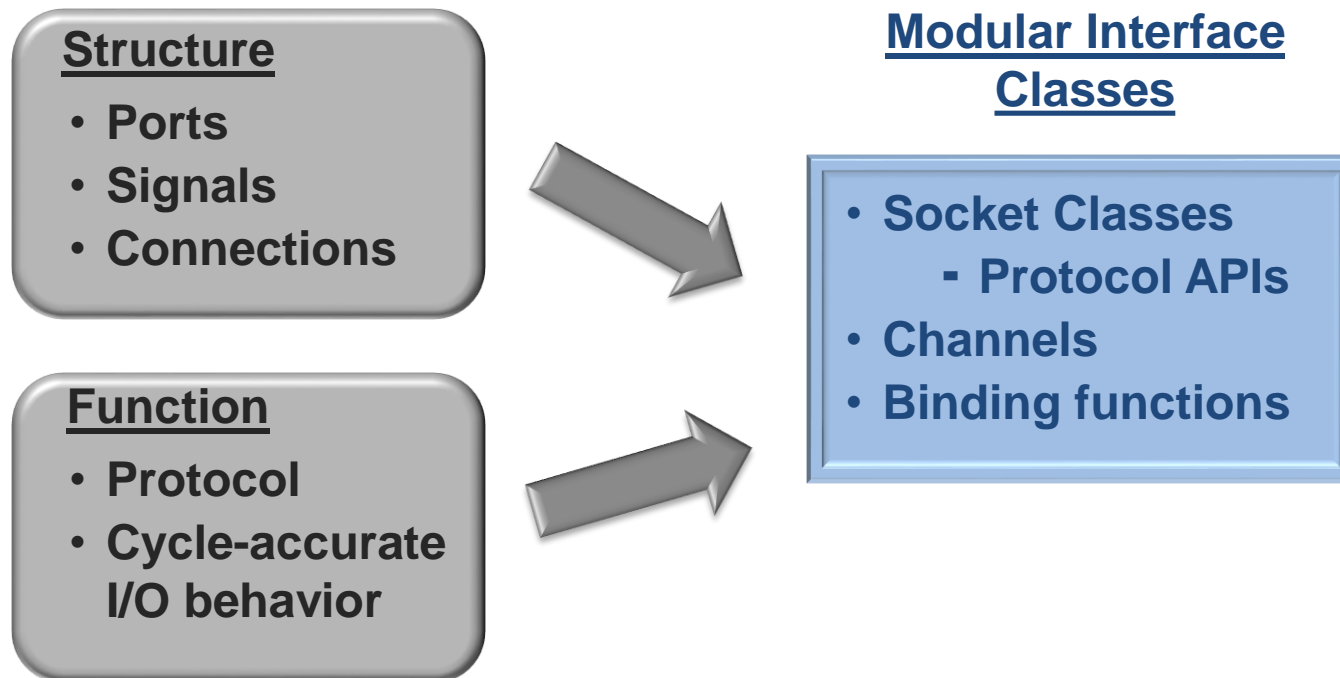


RTL Flow



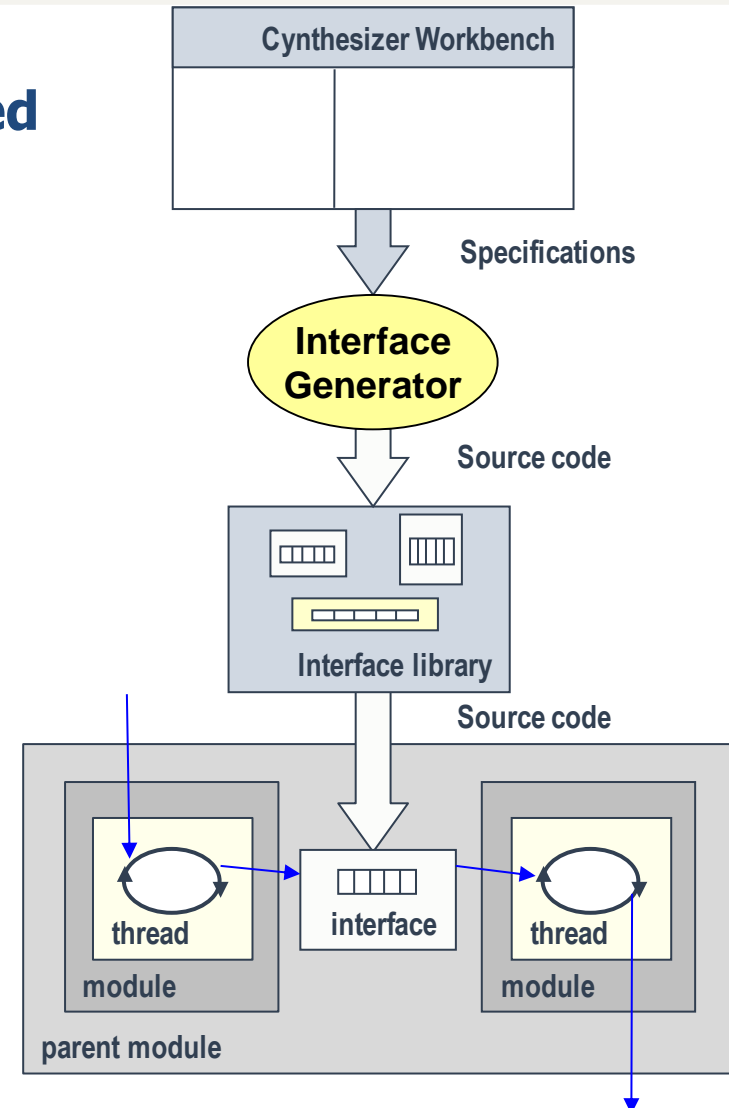
Modular Interfaces

- **Separate protocol behavior from functional behavior**
- **Improves reuse of protocol**
- **Improves reuse of function**



Cynthesizer Interface Generator

- **The Interface Generator is a specialized tool**
 - For creating modular interfaces between modules or threads
- **The user specifies the characteristics of the interface**
 - By filling in forms in the Cynthesizer Workbench
- **The interface generator produces SystemC source code**
 - Synthesizable PIN-level versions
 - High-speed TLM versions

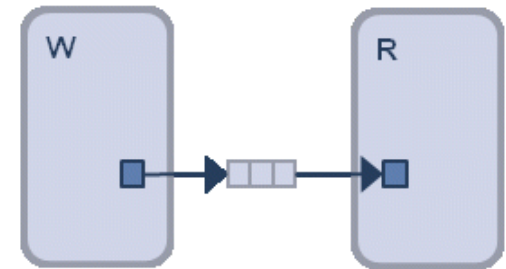


Cynthesizer Interface Classes

Streaming Interfaces

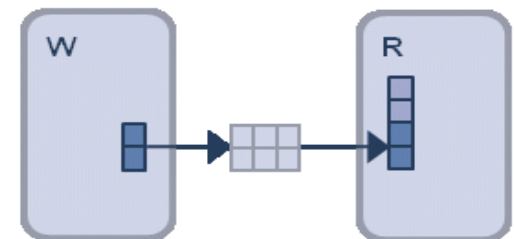
- **The p2p stream class**

- 2-wire ready/busy handshake
- 1 transfer per clock
- Synchronous stalling due to starvation
- Synchronous or asynchronous stalling due to back pressure
- Supports fifo of memory or registers
- Supports clock domain crossing



- **The stream class**

- Similar to p2p stream
- Supports writer and reader with different I/O granularities

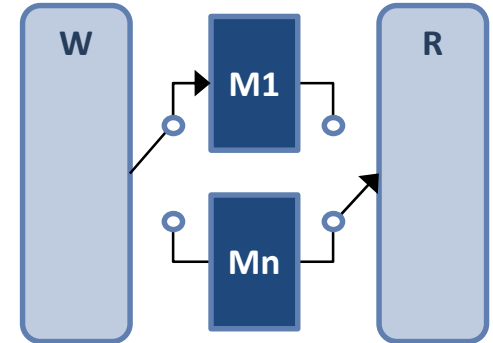


Cynthesizer Interface Classes

Buffer Interfaces

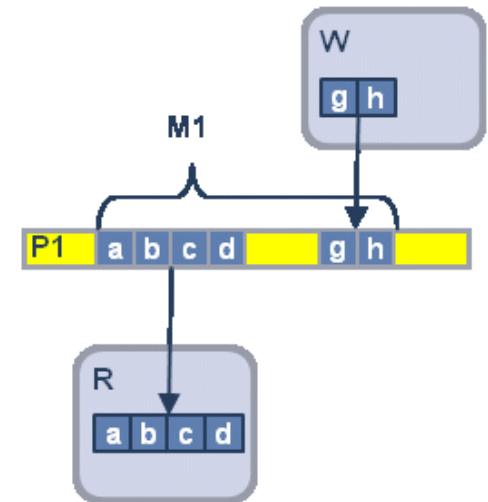
- **Buffer class**

- Supports single buffer, double buffer etc
- Memory based storage



- **Circular buffer class**

- Writer & reader share a memory
- Manages indices for wrap around accesses
- Stalls writer if needed to prevent overrun
- Stalls reader if needed for starvation

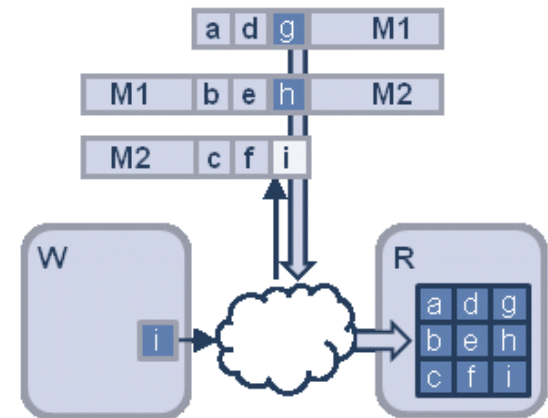


Cynthesizer Interface Classes

Windowing Interfaces

- **Line buffer class**

- For image processing applications
- Implements moving window within a frame
- Contains line memories
- Implements memory index generation
- Handles boundary conditions



Integrating Synthesizable Code In TLM Models

TLM Methodology Goals

- **Modularity**

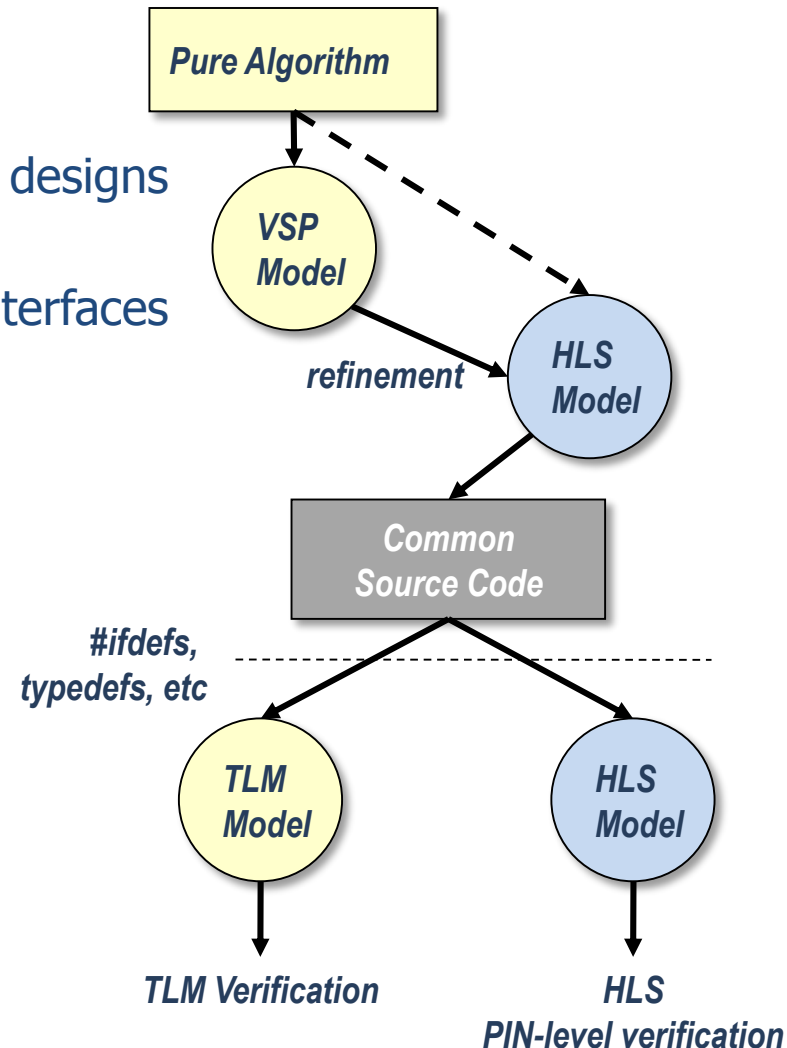
- Allow reuse of bus interface with multiple designs without modification
- Allow reuse of design with multiple bus interfaces without modification
- Maximize shared code between TLM and implementation model

- **TLM priorities**

- Interoperability with other TLM-2 models
- Maximum simulation speed

- **Implementation priorities**

- All the detail needed for implementation
- Good QoR

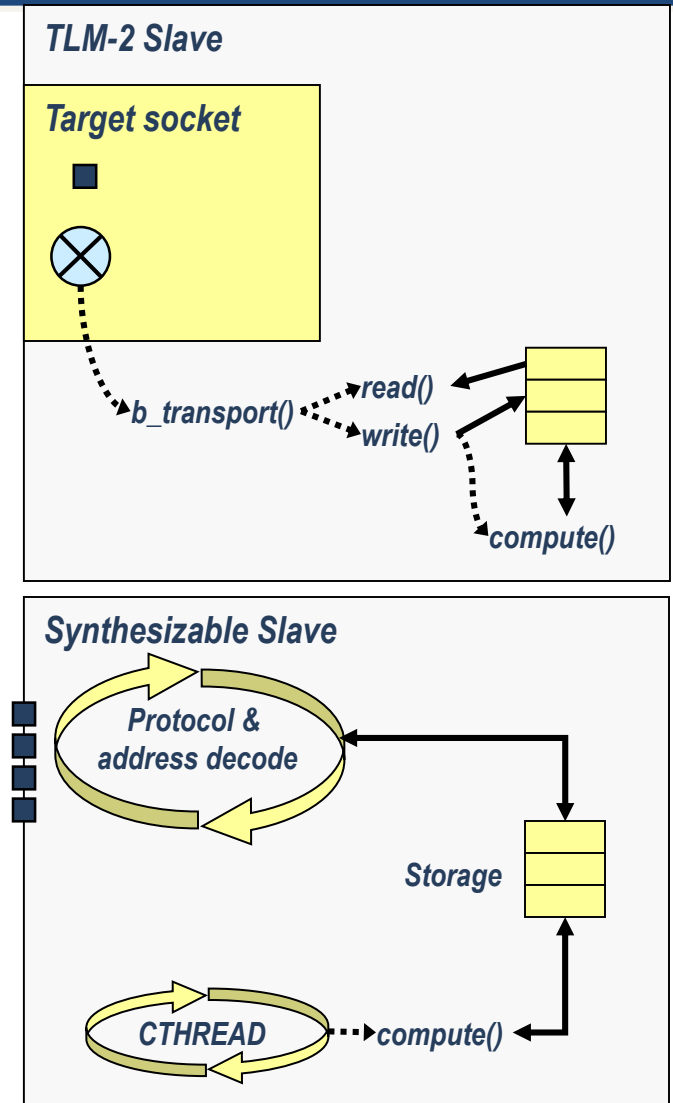


Forte TLM Approach

- **Transform synthesis input using substitution of TLM sockets and synthesizable sockets**
- **Transformation using conventional C++ techniques**
 - Macros, class substitution, template specialization, etc
- **These techniques have been in use since 2006**
 - Point-to-point, TLM-1, TLM-2

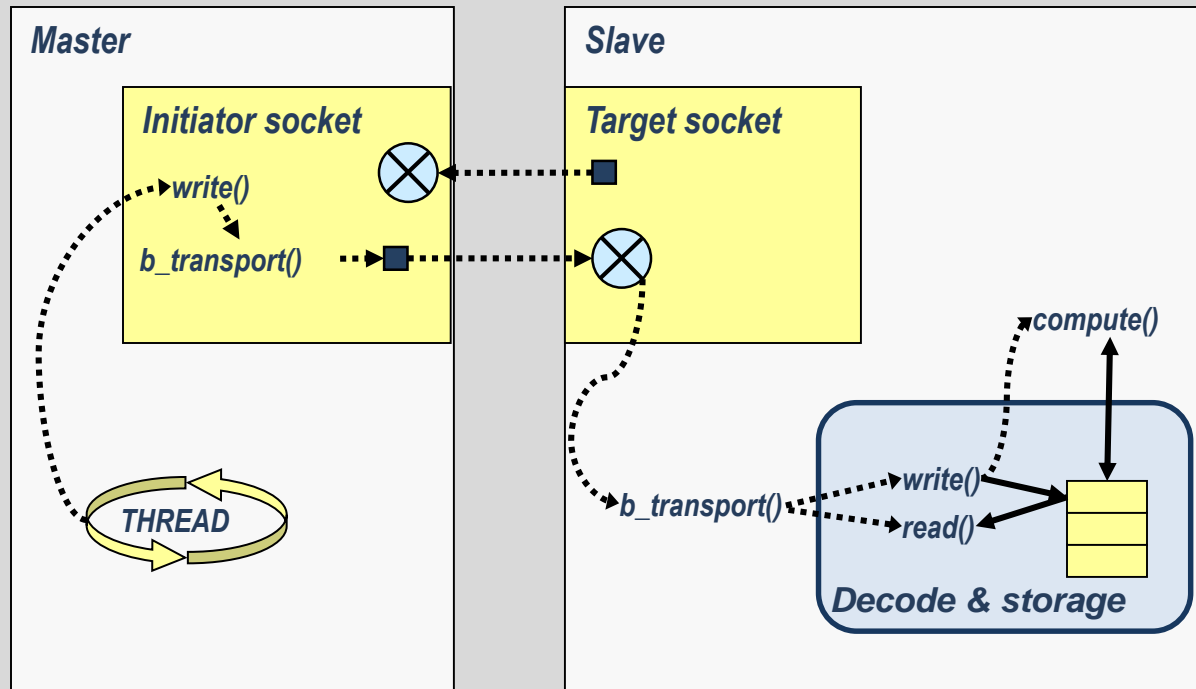
Building Modular Bus Interfaces

- **Main elements of TLM slave**
 - TLM socket
 - Address decode
 - Convenience functions `read()` and `write()`
 - Registers / Memories
 - Computation function
- **Main elements of synthesizable slave**
 - Bus ports
 - Bus protocol thread
 - Address decode
 - Registers / Memories
 - Computation function
 - Computation thread



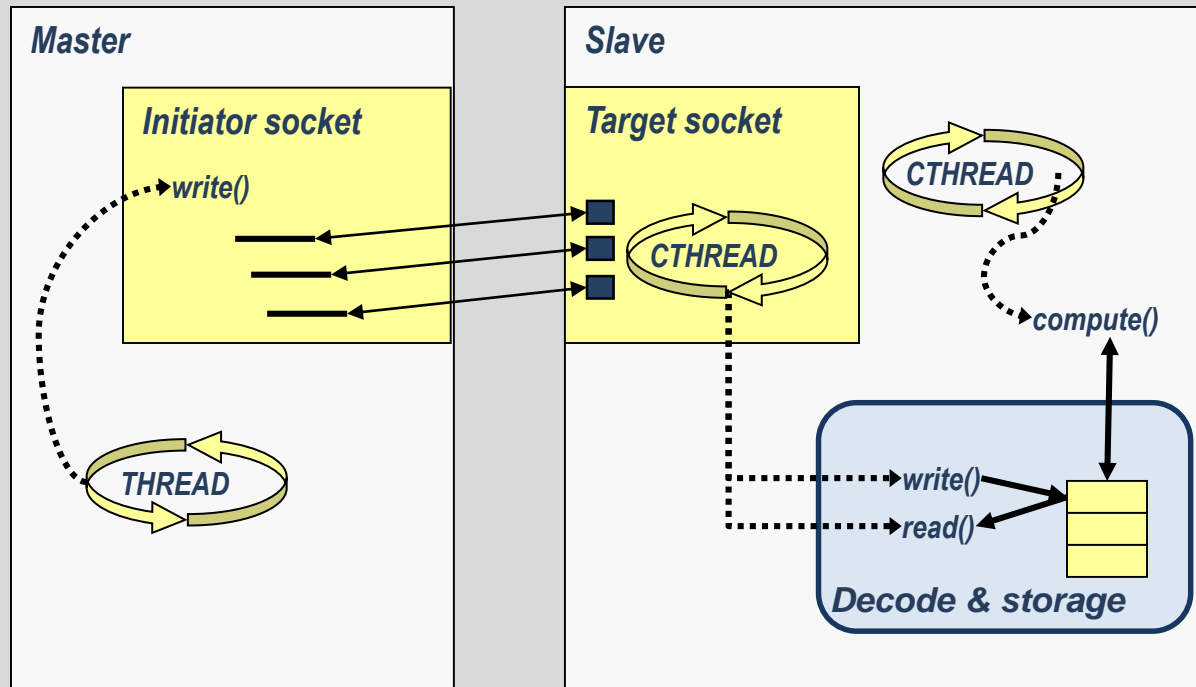
Master/Slave with TLM Sockets

TLM-2



Master/Slave with Pin-level Sockets

PIN Level



Speed Issues To Deal With By Methodology

- **Waiting for clock edges in protocols**
 - Use of modular interfaces with PIN and TLM versions
- **SC_CTHREAD vs SC_THREAD**
 - Macro or equivalent
- **wait() in reset code**
 - #ifdef or equivalent
- **Leaving clocks wired up without transitions**
 - David Black's noclock class
- **sc_uint vs int**
 - Macros/typedefs or equivalent

Forte Features Supporting TLM

- **Modular interface features**

- Synthesizability of encapsulated socket classes
 - sc_in, sc_out ports
 - Binding functions
 - SC_CTHREADs and SC_METHODs in synthesizable sockets

- **Automated simulation configuration**

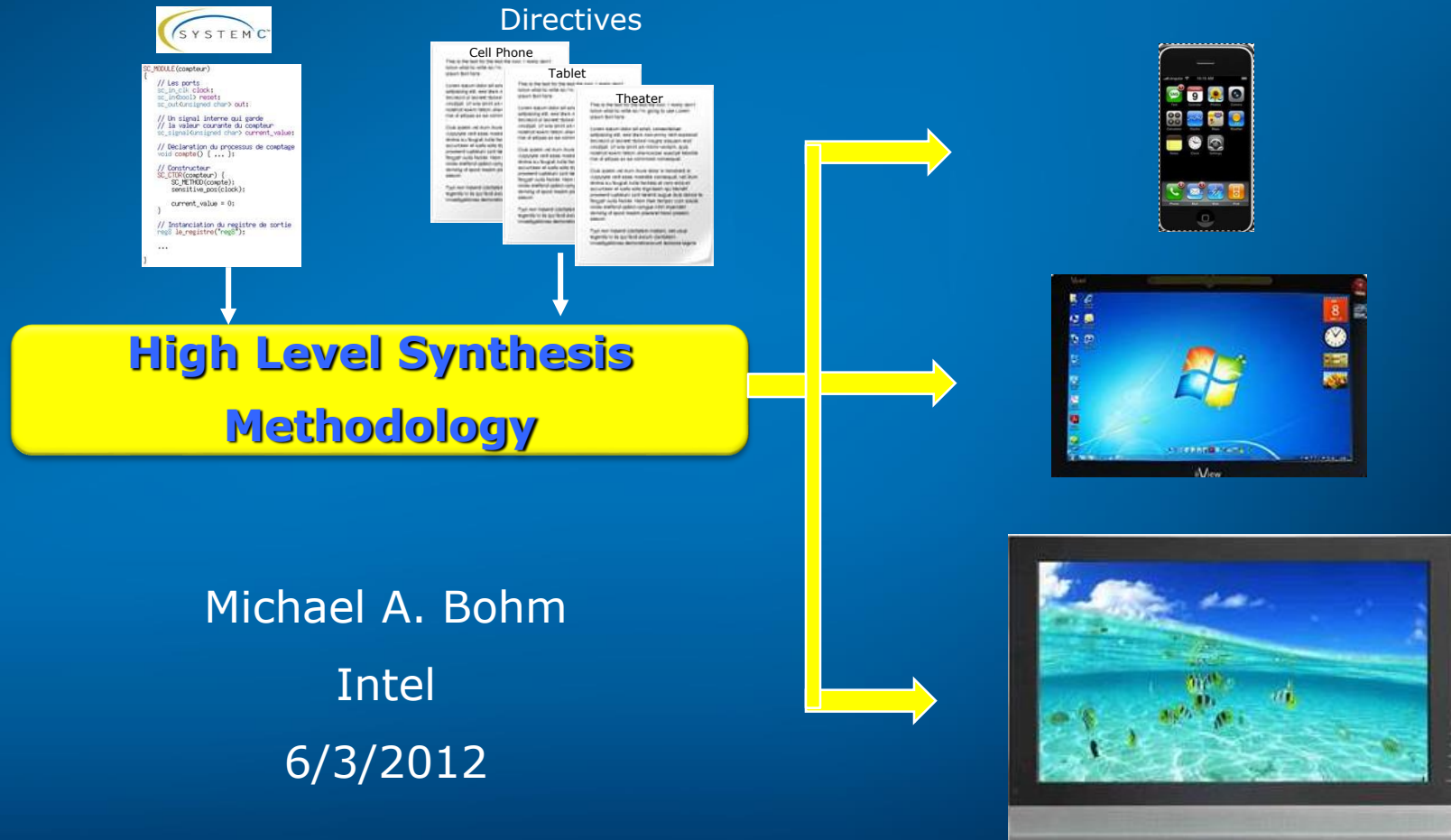
- With automated switching between TLM and PIN-level

- **IP**

- TLM support in all point-to-point and generated interface IP
- Methodology examples
 - TLM-1, TLM-2
 - APB, AHB, AXI

User Experience

Synthesizing SystemC to Layout



Why are companies moving to High Level Synthesis with SystemC?

Faster time to RTL :	64%
Faster verification time :	49%
Fewer engineering resources :	31%
Fewer bugs :	19%
RTL better than hand-coded :	14%
Faster ECO implementation :	8%
Better product differentiation :	7%
Other :	4%

Productivity

QoR

www.deepchip.com (ESNUG 479 Item 4) [02/05/09]

Typical Industry HLS experience

Presented at SystemC conference Japan2010

Design	Hand written RTL [cell area]	1 st trial HLS		Final HLS results		Number of lines		
		[cell area]	Compare to hand design	[cell area]	Compare to hand design	hand RTL	System C	ratio
DesignA	92,187	139,581	1.51	97,479	1.06	612	275	0.45
DesignB	94,248	106,812	1.13	71,982	0.76	654	273	0.42
DesignC	1,053,981	5,381,658	5.11	923,508	0.88	2206	689	0.31
DesignD	131,193	163,026	1.24	124,407	0.95	593	355	0.60
DesignE	280,809	533,115	1.90	268,812	0.96	2220	1185	0.53
DesignF	402,624	636,145	1.58	419,733	1.04	4862	3468	0.71

All Rights Reserved Copyright © 富士通アドバンステクノロジー株式会社 2010

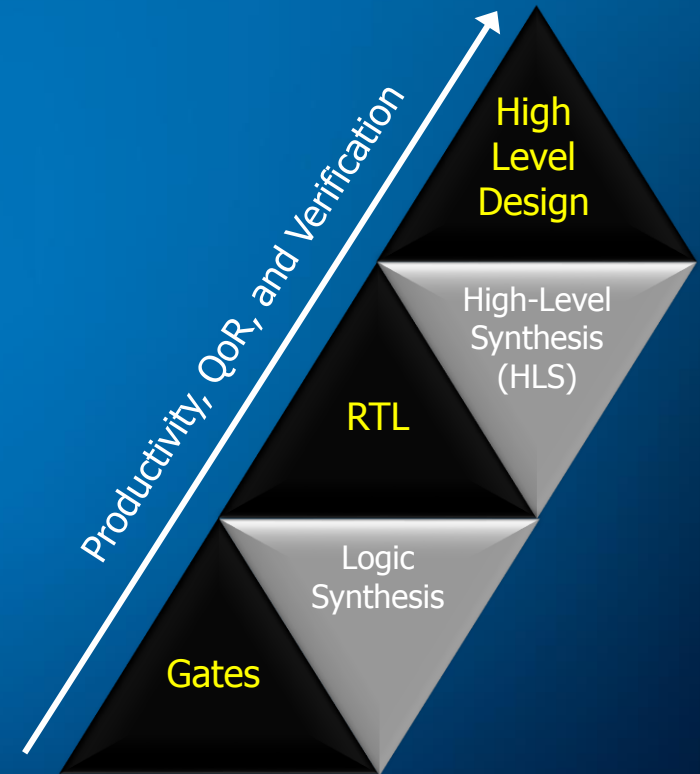
- ***HLS Synthesis has the same learning curve as RTL development.***

© 2013 Forte
Design Systems



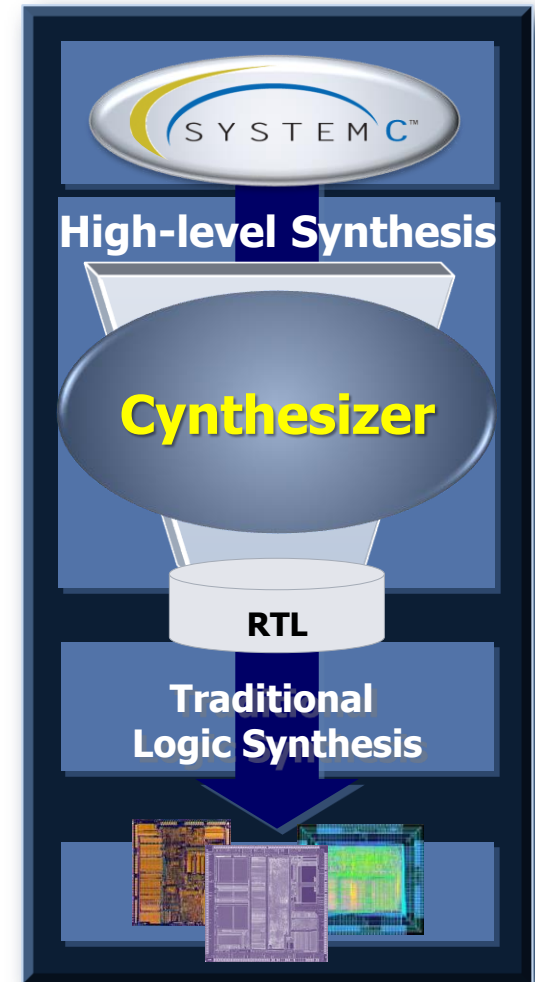
HLS benefits

- Simulation 10-1000x faster
- Hardware/Software co-dev
- High Level Model (HLM) re-use
- End Results:
 - Less engineering time
 - Max IP value, better re-use
 - Faster TTM
- Increased productivity
 - With higher abstraction
- QofR close to/better than RTL
- Faster debug and verification



Keys to Raising Abstraction

- **Implicit state machine**
 - More like algorithm code
 - Leaves room for exploration
- **Design exploration**
 - Same source can produce designs for different purposes
- **Array handling**
 - Arrays can be implemented in multiple ways
 - HLS schedules memory protocol signals w/ algorithm
- **Handling I/O protocols**
 - HLS schedules I/O protocol signals w/ algorithm
 - Modular interfaces allow protocol encapsulation & reuse



The background of the slide is a deep blue with a complex pattern of glowing, fiber-optic-like lines. These lines are in various shades of blue and white, creating a sense of movement and depth. They appear to be light trails or reflections on a dark surface, giving the overall image a high-tech, digital feel.

Thank you for your time!