



HDLCon 2001 Tutorial 6

A Practical Approach to System Verification and
Hardware Design

The PUBLIC SUBSET Of The SUPERLOG Language

Peter Flake, Chief Technical Officer

David Rich, AE Director

This material is copyrighted; all rights are reserved by Co-Design Automation Inc. This document may not, in whole or in part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without the prior consent of Co-Design Automation Inc.

The software programs described herein are copyrighted and all rights are reserved by Co-Design Automation Inc. The programs may not be copied or duplicated except as expressly permitted in the Software License Agreement.



© 2001 Co-Design Automation, Inc.
Tutorial presented at HDLcon 2/28/2001

SUPERLOG Subset Tutorial Overview

Welcome to the SUPERLOG Public Subset Tutorial

The purpose of this tutorial is to demonstrate the nature of the SUPERLOG language, and some of the base capabilities contained therein.

It should be noted that the SUPERLOG language contains many additional capability THAT ARE NOT described in this document, particularly in the area of abstract system modeling and verification. For more information please contact:

**Co-Design Automation, Inc.
www.co-design.com**

**1 877 6 CODESIGN
info@co-design.com**



© 2001 Co-Design Automation, Inc.
Tutorial presented at HDLcon 2/28/2001 - SUPERLOG Public Subset

Co-Design Automation, Inc.

Company Goal

**Provide an order of magnitude
productivity improvement across design
and verification methodologies**



1997
Founded by
industry
leading
experts

May 1999
SUPERLOG
announced

Nov 1999
12 companies
endorse
SUPERLOG

May 2000
SYSTEMSIM
SYSTEMEX
Release

Jan 2001
Leading
companies
announce
CDA enabled
methodologies

The Co-Design Team

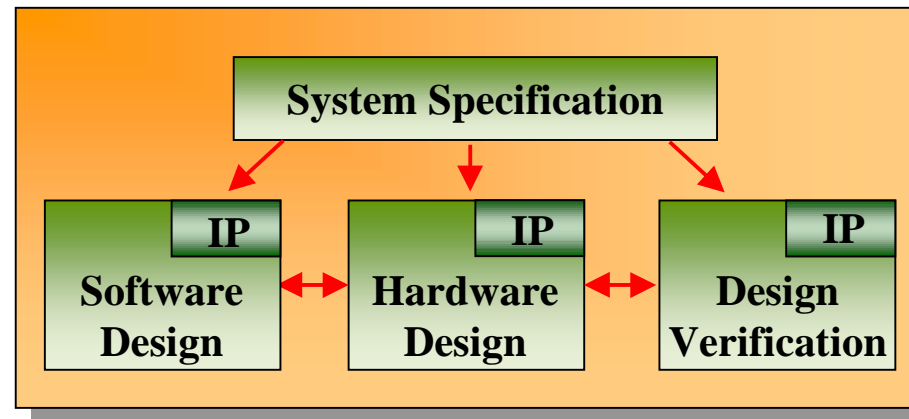
Management

- **Simon Davidmann** - CEO
 - Formerly VP and European GM Chronologic (VCS, now Synopsys), Ambit (BuildGates, now Cadence), Virtual Chips (now Pheonix)
 - **Peter Flake** - CTO
 - Formerly HILO team leader, HDL visionary
 - **Phil Moorby** – Chief Scientist
 - First Cadence fellow and architect of Verilog, Verilog-XL
 - **Dave Kelf** - VP Marketing
 - Formerly Dir Mkt Cadence simulation product line
- +14 of the top simulation & language experts worldwide**
-

Advisors

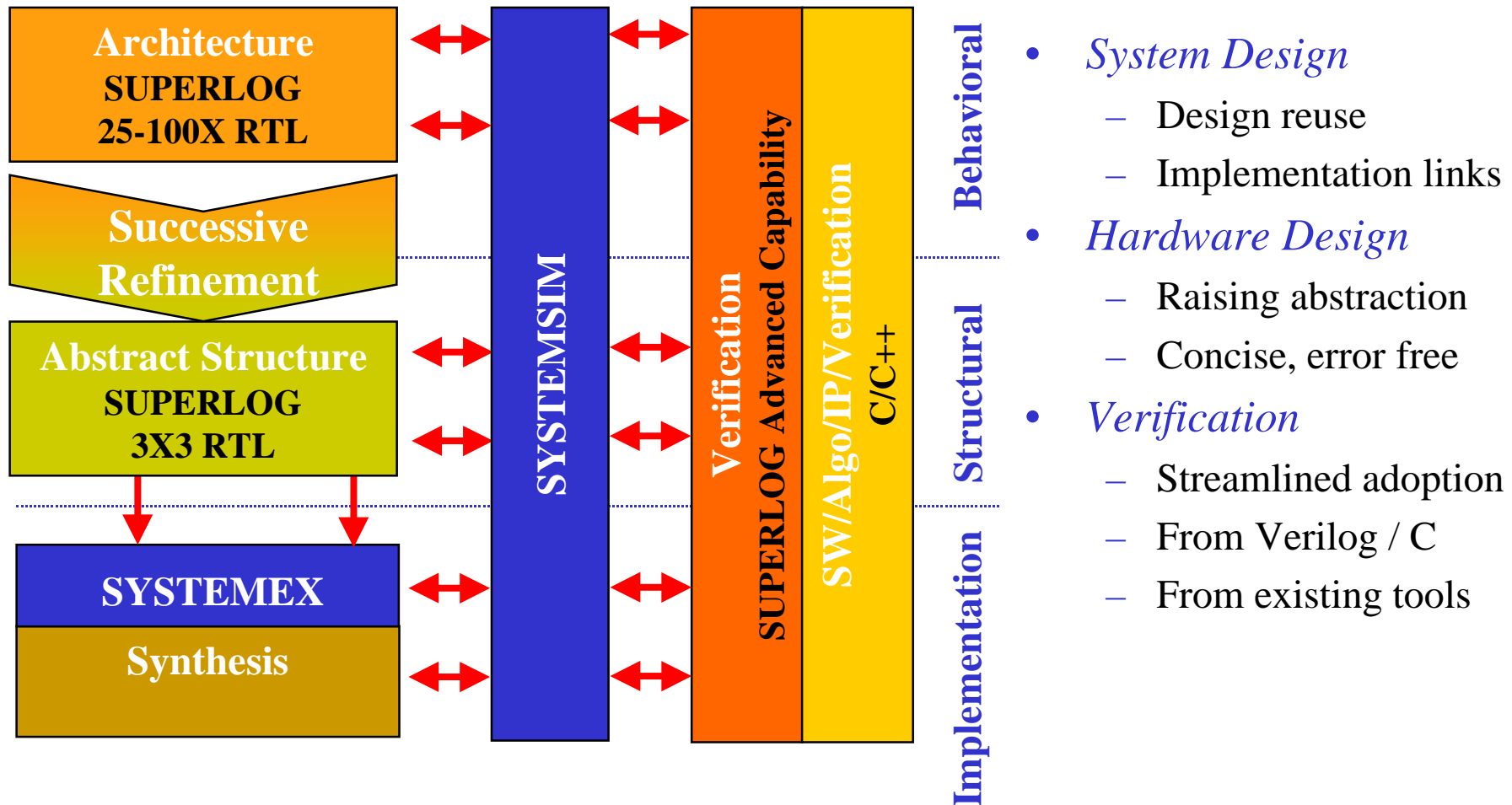
- **Professor Don Thomas**
 - Carnegie Mellon University
- **Andy Bechtolsheim**
 - Co-founder Sun Micro., VP Cisco
- **Venk Shukla**
 - Initiator of OVI, CEO EveryPath.com
- **Rajeev Madhavan**
 - Founder LogicVision, Ambit, Magma
- **Raj Singh** (Board Member)
 - Founder of Fiberlane, Cerent, Siara
- **Raj Parekh**
 - Former CTO Sun Microsystems, SGI
- **Rich Davenport** (Board Member)
 - former President & COO Summit Design

Focusing On Key Methodology Problems



- **Verification** environment speed and complexity
- **Hardware** complexity and design flow performance
- **System & SW** evaluation speed and implementation links

Streamlined Methodology Components



SUPERLOG - Targeting Design Productivity

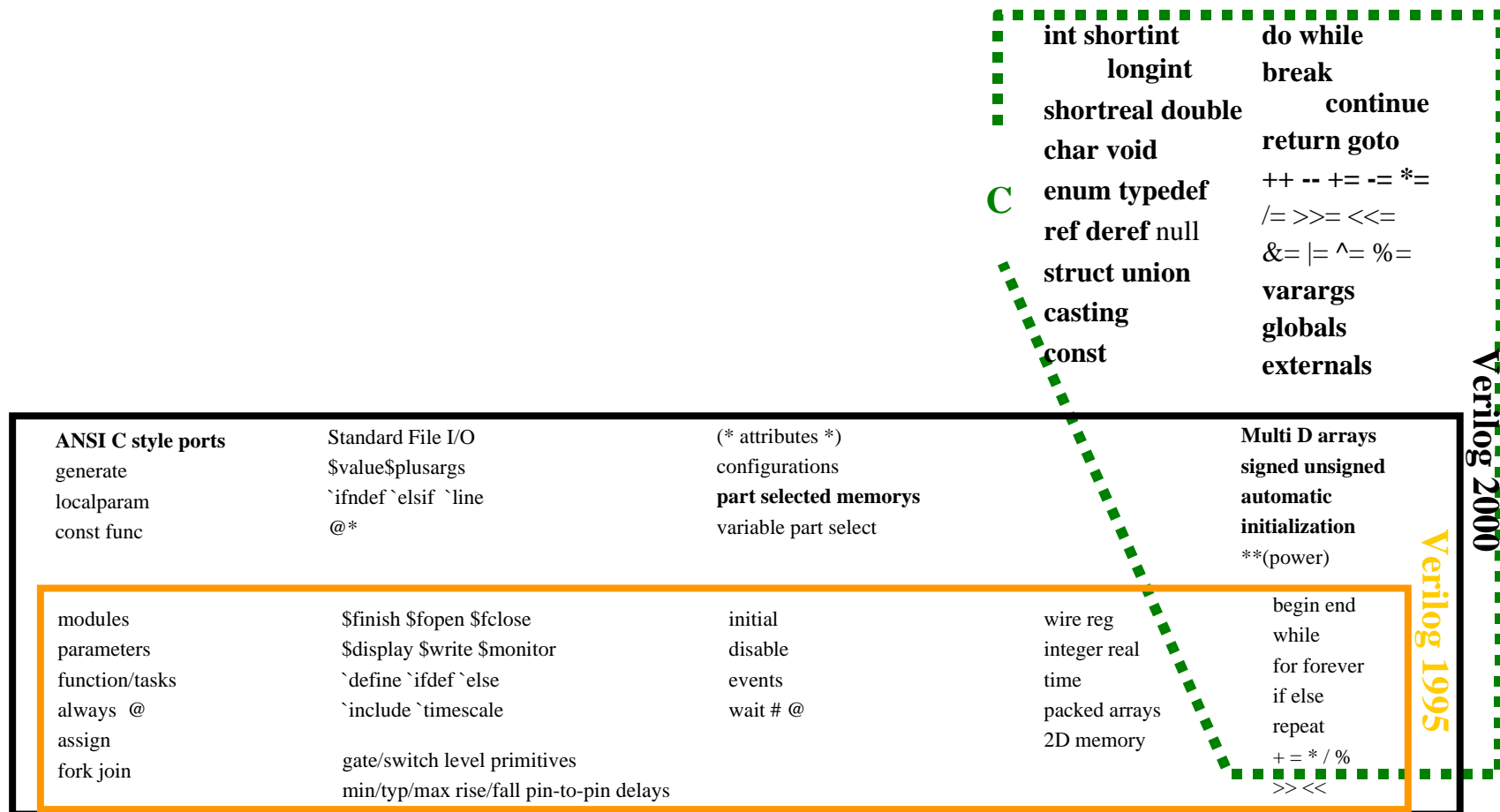


- Superset of Verilog (and Verilog2K), with the addition of C programming, system and verification capabilities
- Targeting the SoC specializations
 - architecture, hardware, verification, with links to C
- Designed for **USE**ability
 - **U**nifying the SoC design flow
 - Dramatically increasing design **S**peed
 - Providing an **E**volutionary path

SUPERLOG -Building on Verilog 1

ANSI C style ports	Standard File I/O	(* attributes *)	Multi D arrays	Verilog 2000
generate	\$value\$plusargs	configurations	signed unsigned	
localparam	`ifndef `elsif `line	part selected memorys	automatic	Verilog 1995
const func	@*	variable part select	initialization	
			**(power)	
modules	\$finish \$fopen \$fclose	initial	wire reg	begin end
parameters	\$display \$write \$monitor	disable	integer real	while
function/tasks	`define `ifdef `else	events	time	for forever
always @	`include `timescale	wait # @	packed arrays	if else
assign	gate/switch level primitives		2D memory	repeat
fork join	min/typ/max rise/fall pin-to-pin delays			+ = * / %
				>> <<

SUPERLOG -Building on Verilog 2



SUPERLOG -Building on Verilog 3

Additional Verification / System Features				
User Defined Ports	Self-Introspection	Process	int shortint	do while
State Machines	Safe Memory	Queues	longint	break
Communication Interfaces	Management	String Types	shortreal double	continue
Templates	Temporal Procedures	Dynamic Types	char void	return goto
Packed Arrays/Struct	Assertions	2/4 State Variables	C enum typedef	++ -- += -= *=
Timeunits	Protocol Checks	Semaphores	ref deref null	/= >>= <<=
Sparse/Associative Arrays	Coverage Checks	Liveness	struct union	&= = ^= %=
	Block Labeling	System Data	casting	varargs
	Conditional Events		const	globals
				externals
ANSI C style ports	Standard File I/O	(* attributes *)		Multi D arrays
generate	\$value\$plusargs	configurations		signed unsigned
localparam	`ifndef `elsif `line	part selected memorys		automatic
const func	@*	variable part select		initialization
				**(power)
modules	\$finish \$fopen \$fclose	initial	wire reg	begin end
parameters	\$display \$write \$monitor	disable	integer real	while
function/tasks	`define `ifdef `else	events	time	for forever
always @	`include `timescale	wait # @	packed arrays	if else
assign			2D memory	repeat
fork join	gate/switch level primitives			+ = * / %
	min/typ/max rise/fall pin-to-pin delays			>> <<

SUPERLOG

Verilog 2000

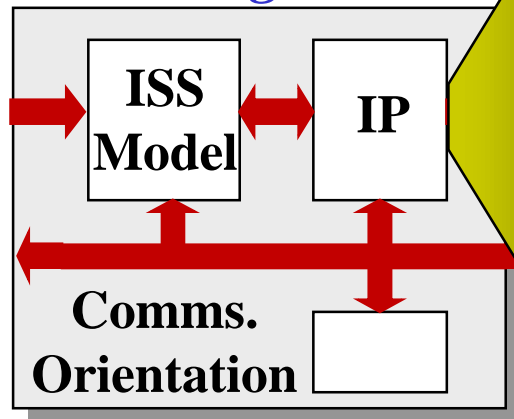
Verilog 1995

Items in **BOLD** discussed in Tutorial

SUPERLOG Enhances The Design Flow

Efficient Design Across Methodology Boundaries

Platform Design Flow



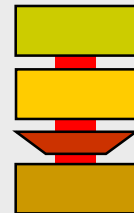
Interfaces *C Import Export* *Reuse Features*

Optimized HW Flow

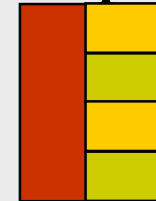
Algorithm Model

```
always @eventstr
  if (str == parent ->s)
endfunction
```

Pipeline/ Datapath



Concise Descriptions



Complex Control

Standard HW



*C/Verilog
Code Mix*

*Abstract
Refine*

*Processes
Queues*

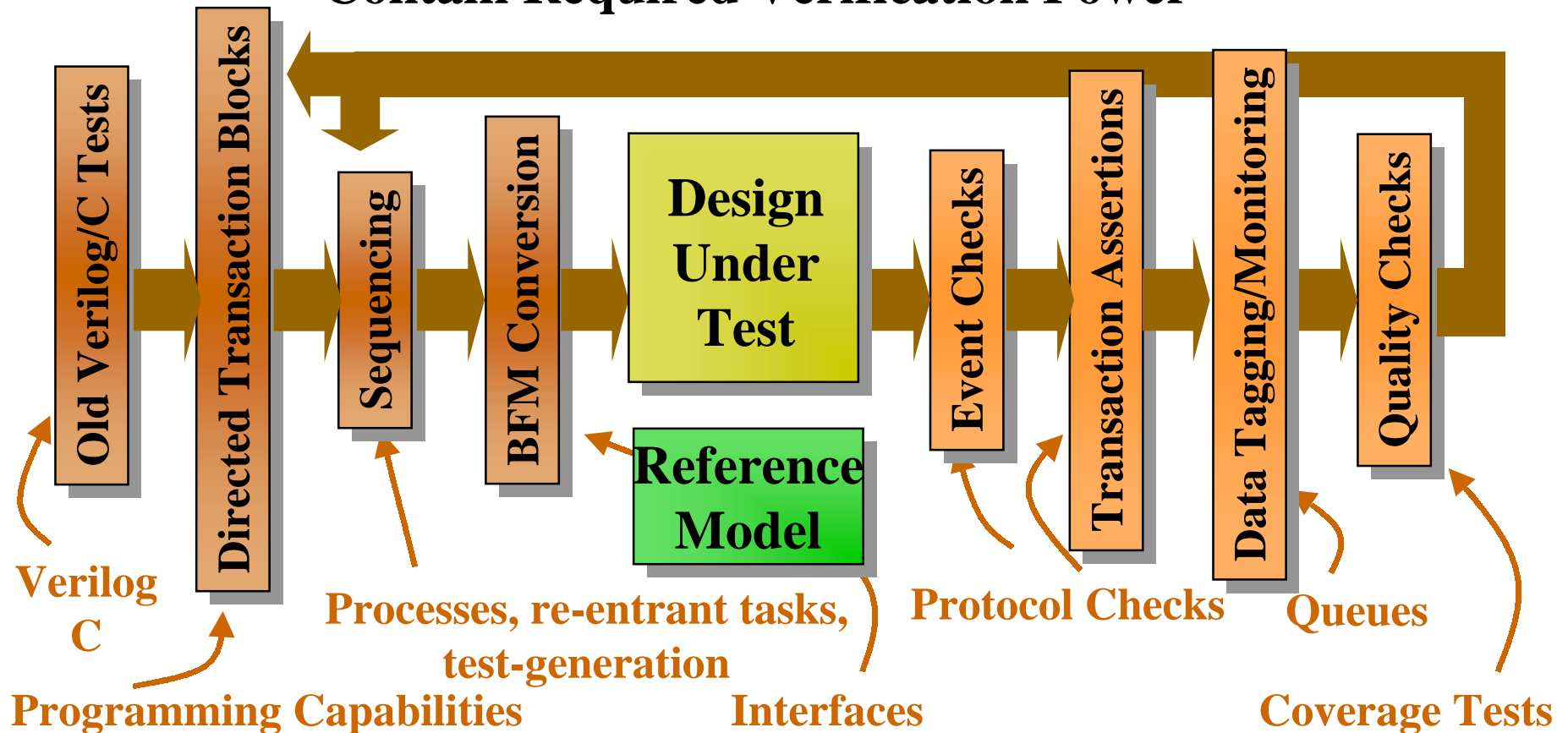
*Verilog2K
C abstracts*

*Advanced
FSMs*

*Verilog
Base*

SUPERLOG Accelerating Verification

**SUPERLOG and SYSTEMSIM
Contain Required Verification Power**

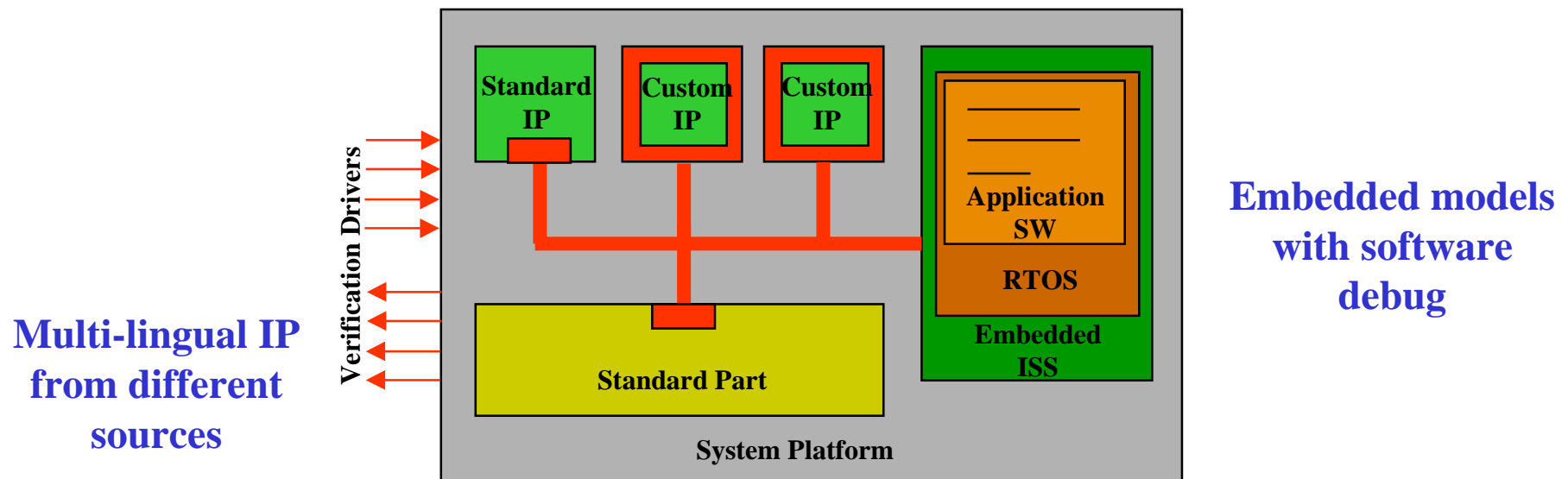


SUPERLOG Enabling Platform Infrastructure

Platform Based Design Offers Possibilities

Inter-block Communication

Verification Automated For IP



Communication, Verification, Reuse SUPERLOG Provides Transparent Infrastructure

SUPERLOG Abstraction Driving Performance

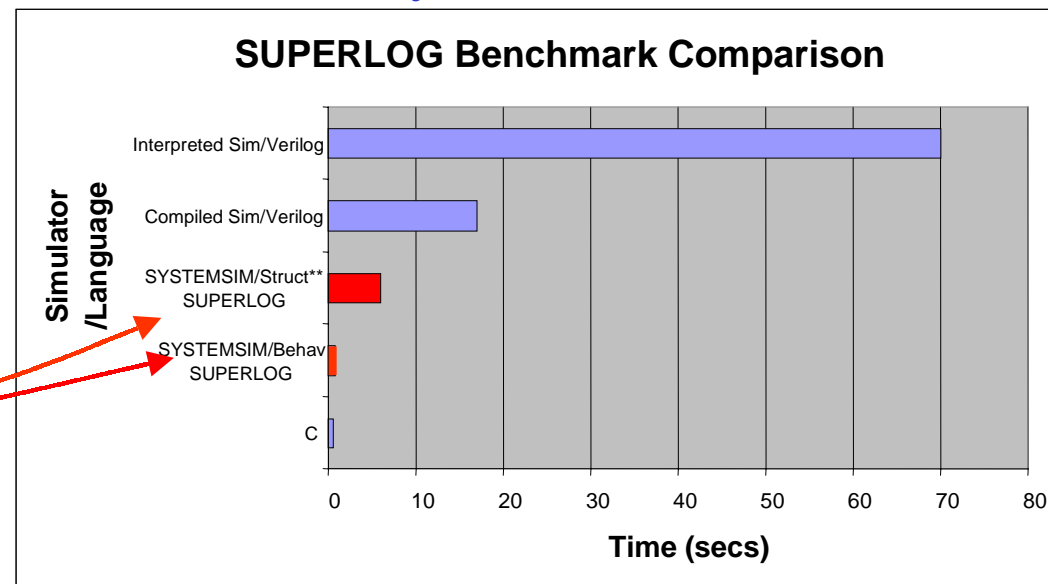
SUPERLOG enables performance abstraction trade-off choice

SUPERLOG Opportunity

Dramatic performance improvement

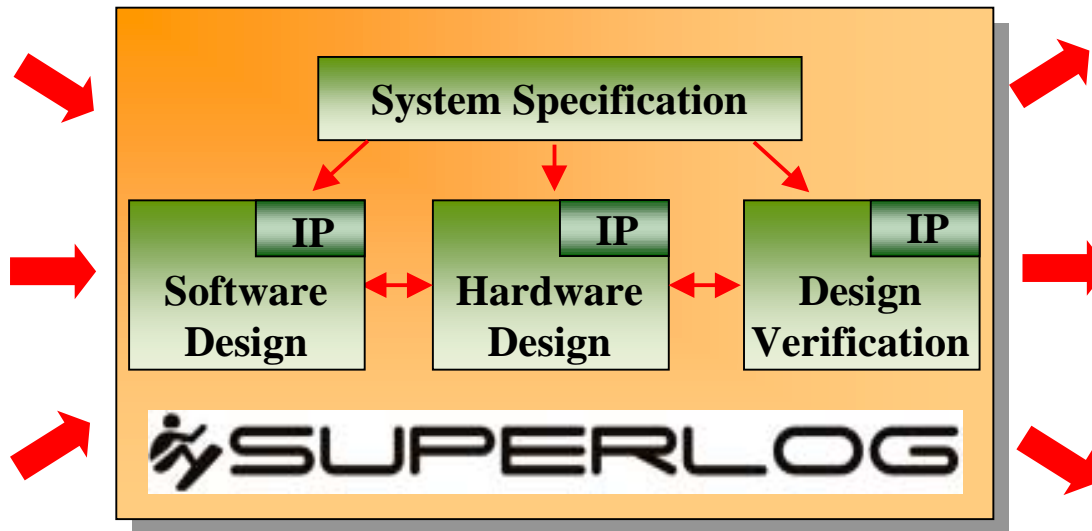
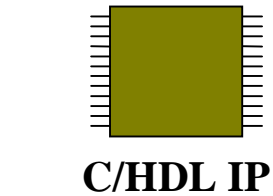
- RTL Coding
- Abstract Coding
- Integrated Verification
- Seamless C usage

Dhrystone Benchmarks On Various Code Styles and Simulators

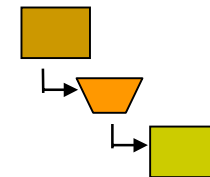


SUPERLOG - Evolution, Not Revolution!

Easy design integration



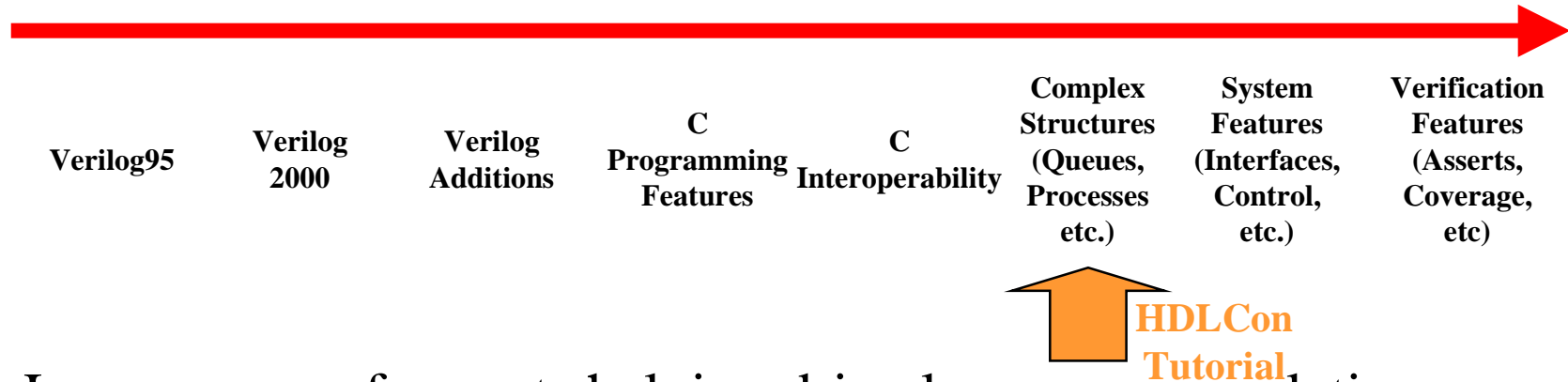
Easy adoption



SUPERLOG = Verilog +++
Easy to take existing Verilog designs
and leverage SUPERLOG where required

Progress Towards Publication

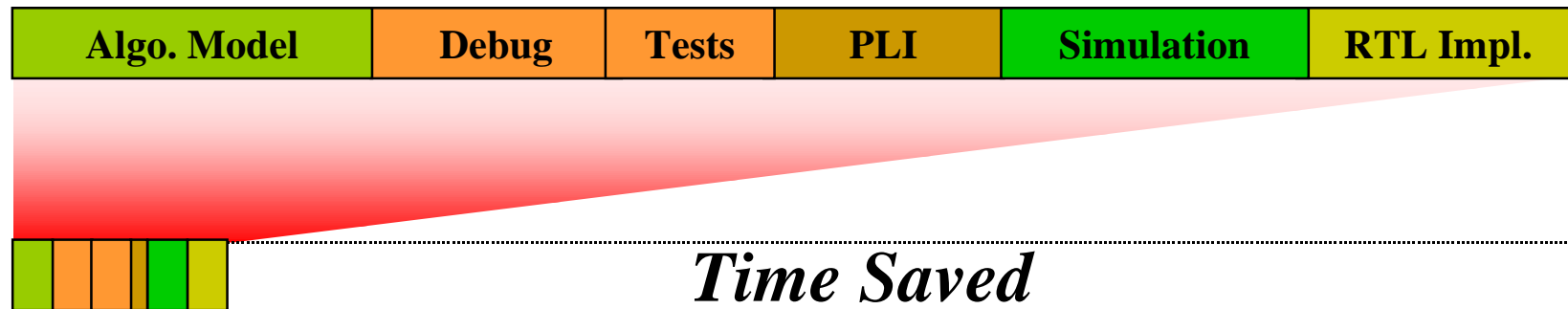
SUPERLOG Publication Stages



- Large group of experts helping drive language completion
 - Forum and other organizations actively involved
- Phased approach to publication, as sections are completed
 - Good progress to date
- SUPERLOG Extended Synthesis Subset offered to Accellera

Order Of Magnitude Productivity Enhancements

Practical Productivity



Direct Performance

- Concise RTL implementation
- PLI elimination
- Simulation speed acceleration

Overall Productivity

- Integrated algorithm modeling
- Efficient testbench creation
- Rapid debug



Language Evolution

Learning SUPERLOG Is Easy

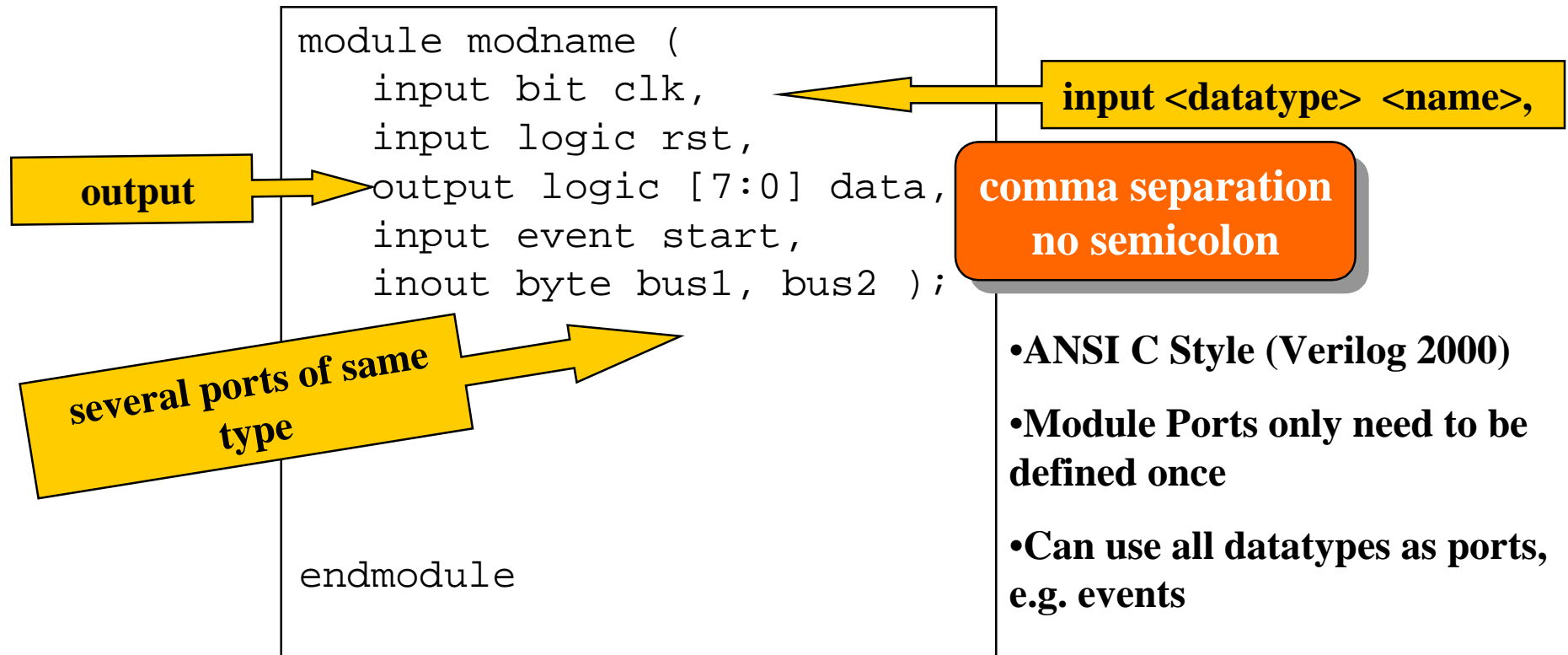
module

**always
block**

```
module shifter (  
    input  logic clk,  
    input  logic [7:0] data_in,  
    output logic [7:0] data_out,  
    input  logic load,  
    input  logic shift_left );  
  
    always @(posedge clk)  
        if (load) data_out <= data_in;  
        else  
            if (shift_left)  
                data_out <= {data_out[6:0], 1'b0};  
endmodule
```

**ANSI C Style
header**

Module Header Syntax Enhancements



- More concise format

Module Port Default Modes

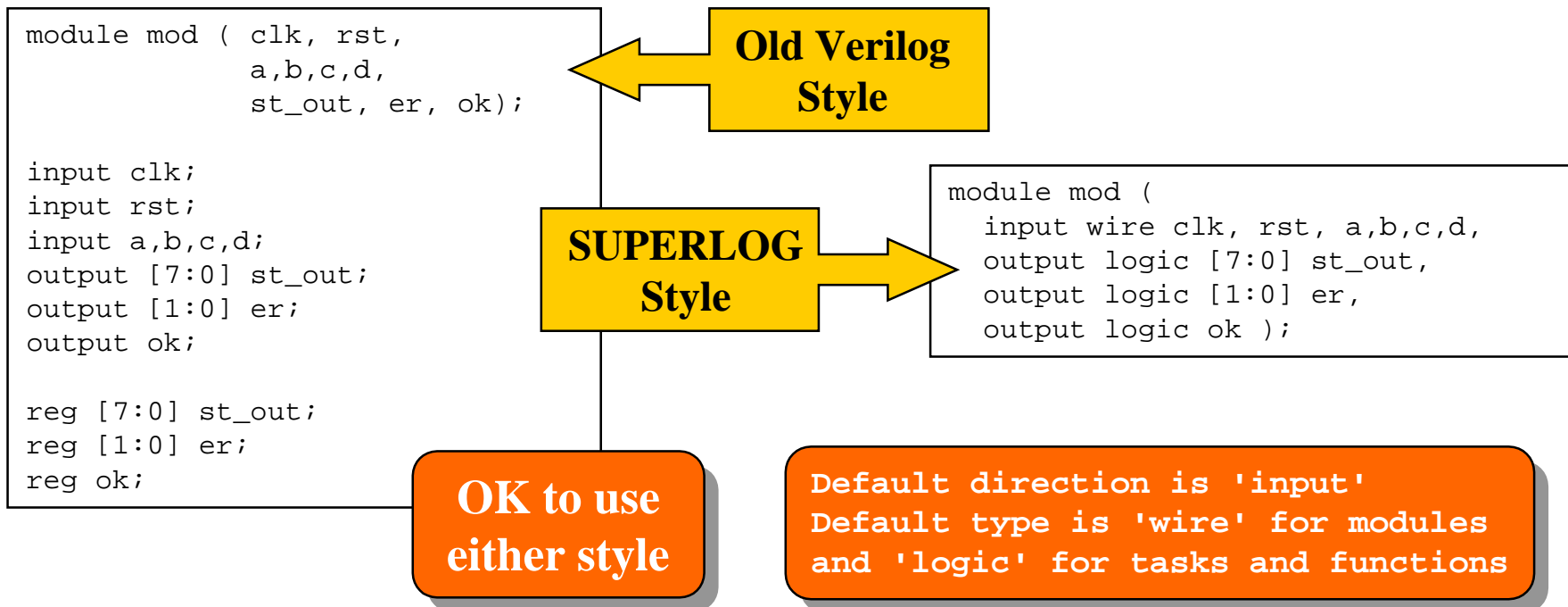
```
module modx (  
    input  logic clk,  
    input  logic rst,  
    input  logic sel_bus,  
    input  bit  ctrl,  
    input  logic [31:0] a,  
    input  logic [31:0] b,  
    input  logic [31:0] c,  
    input  logic [31:0] d,  
    input  logic [3:0] sel,  
    input  logic [7:0] addr,  
    output logic err,  
    output logic [31:0] out,  
    inout  logic [7:0] bus1,  
    inout  logic [7:0] bus2 );
```

**these work
the same**

```
module modx (  
    input  logic clk, rst, sel_bus,  
    bit    ctrl,  
    logic [31:0] a,b,c,d,  
    logic [3:0] sel,  
    logic [7:0] addr,  
    output logic err,  
    logic [31:0] out,  
    inout  logic [7:0] bus1, bus2);
```

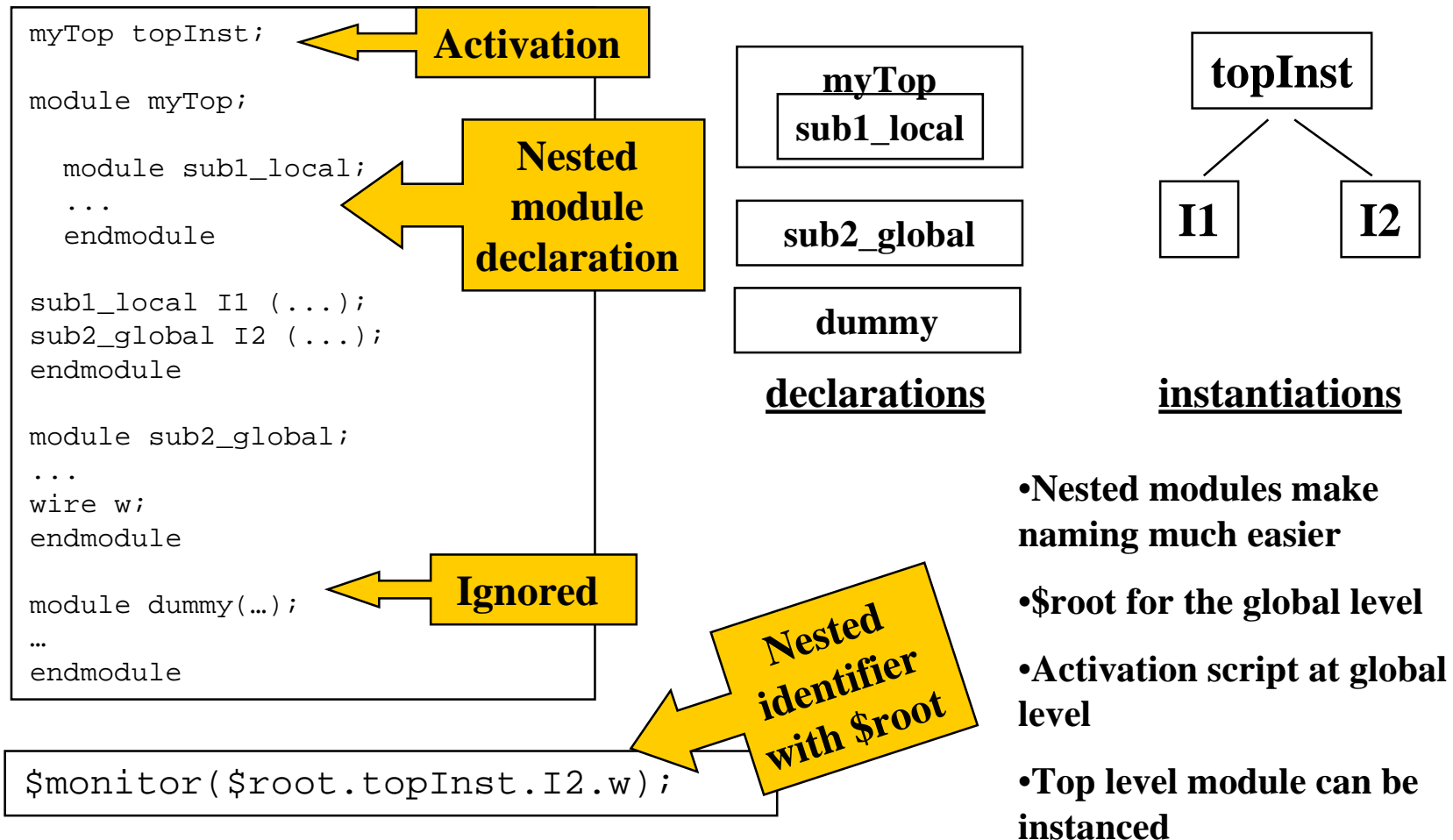
**Tasks/function headers
work the same way**

Port Comparison With Verilog 1995



- No duplication of information necessary

Explicit Hierarchy



Implicit Hierarchy

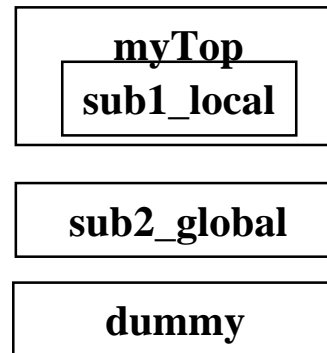
```
module myTop;

  module sub1_local;
    ...
  endmodule

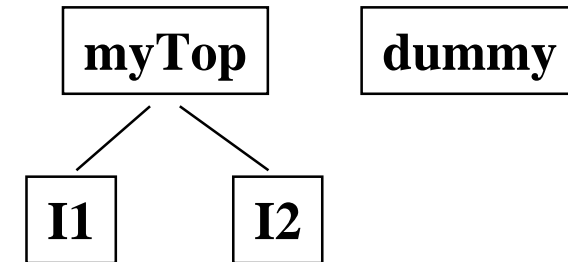
  sub1_local I1 (...);
  sub2_global I2 (...);
endmodule

module sub2_global;
  ...
  wire w;
endmodule

module dummy(...);
  ...
endmodule
```



declarations



instantiations

- **Uninstantiated modules become top level instances**
- **Same as Verilog**
- **May lead to unwanted modules in simulation**



Constants

Parameters

Macros

Timescales

Using Literals

```
logic [31:0] a;  
...  
a = 32'hf0ab;
```

**This works like
in Verilog**

```
a = 'z;  
b = '0;
```

**This fills the packed
array with the same
value**

```
logic [31:0] a;  
...  
a = 32'hffffffff;  
a = '1;
```

**These are
equivalent**

- **Convenient way to fill up a vector with a bit constant**

Parameterizable Text Macros

like in Verilog

```
`define MAX 7
`define print(var) $display("The value of var is %d", var)

...

int a[`MAX:0];
int i;

...

`print(i);
```

parameterizable
macros

no space between
macro and
open bracket

```
`define macro(arg1, arg2, arg3) ...
```

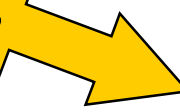
- Powerful facility to write concise code

Define Example With Text

Define is tokenized
so *y* does not
match *y* in *display*



more readable
to leave the ; on
the calling line



```
`define print(y) $display("var %s equals %d ", `"y`, y)
...
int num;
`print(num);
```

Constants

- Use like defines or parameters
- No redefining or overriding (within scope)

global constant →

```
const bit FALSE = 0;  
const bit TRUE = 1;
```

local constant →

```
const real pi = 3.1415926535;  
const int TRUE = -1;  
  
initial $display("pi=%f", pi);  
  
endmodule
```

Parameters

```
module top;

logic clk;

clockgen #(.start_value(1'b0), .delay(50)) c (clk);

always @clk $display("t=%t clk=%b", $time, clk);

initial
begin
    repeat(10) @(posedge clk) ;
    $finish(0);
end
endmodule

module clockgen( output ctype clk);
    parameter logic start_value=0;
    parameter time delay=100;
    parameter type ctype=bit;

    initial clk <= start_value;

    always #delay clk <= !clk;

endmodule
```

**override
parameter**

**parameters used
before definition.**

parameters have type

- **Parameters can be passed by name**

Time Constants

you can use
#delays with
no units, like
in Verilog

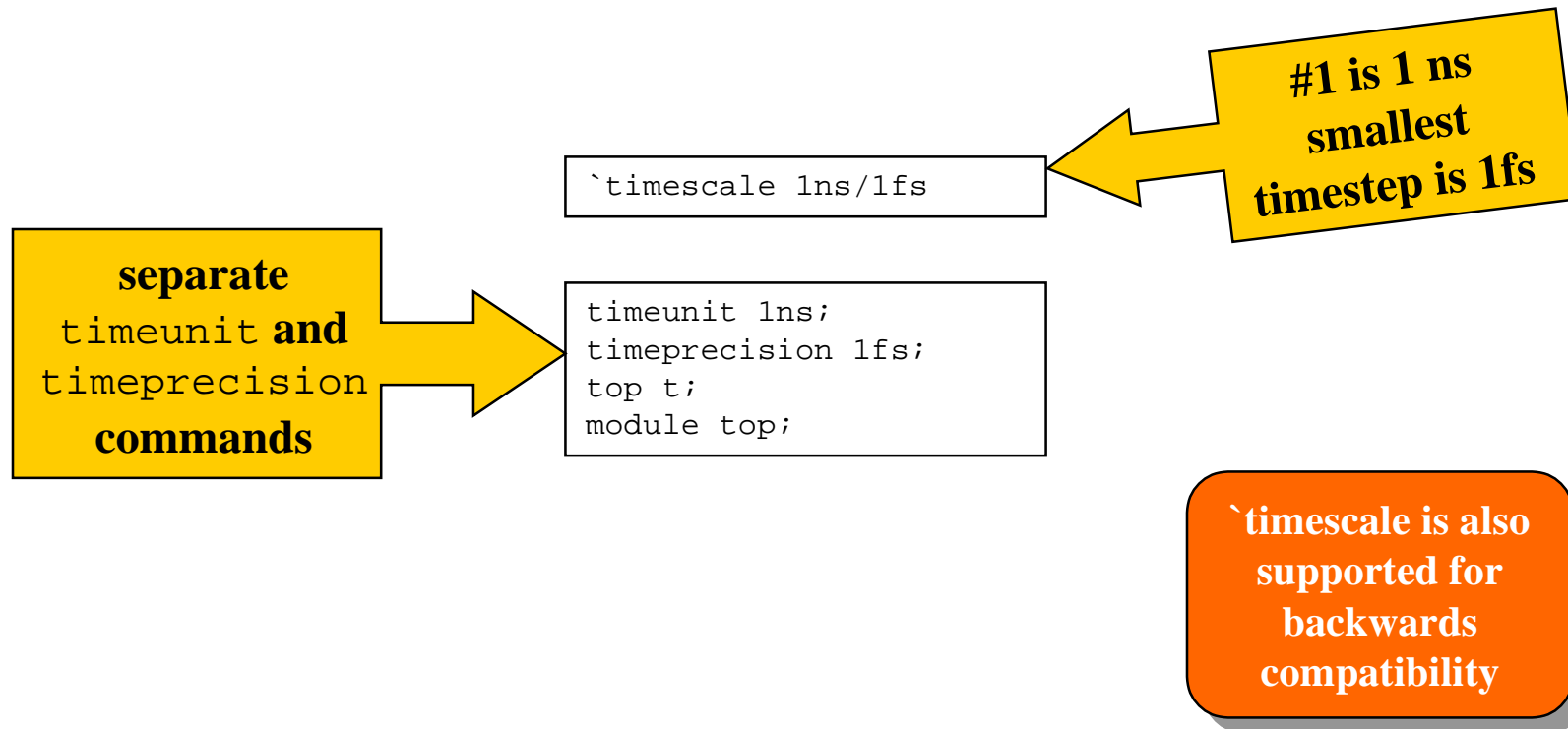
```
#10 a <= 1;  
#5ns b <= !b;  
#1fs $display("%b", b);
```

No space between
digit and letter!

you can also
specify units
with your delays

- legal time units are s, ms, us, ns, ps, fs
- The default is #1 = 1s

Timescale Rules



- Removes file order dependency
- Default is timeunit 1s; timeprecision 1s;

Timeunit Example

```
timeunit 1ns;

top t;

module top;
timeunit 1ms;

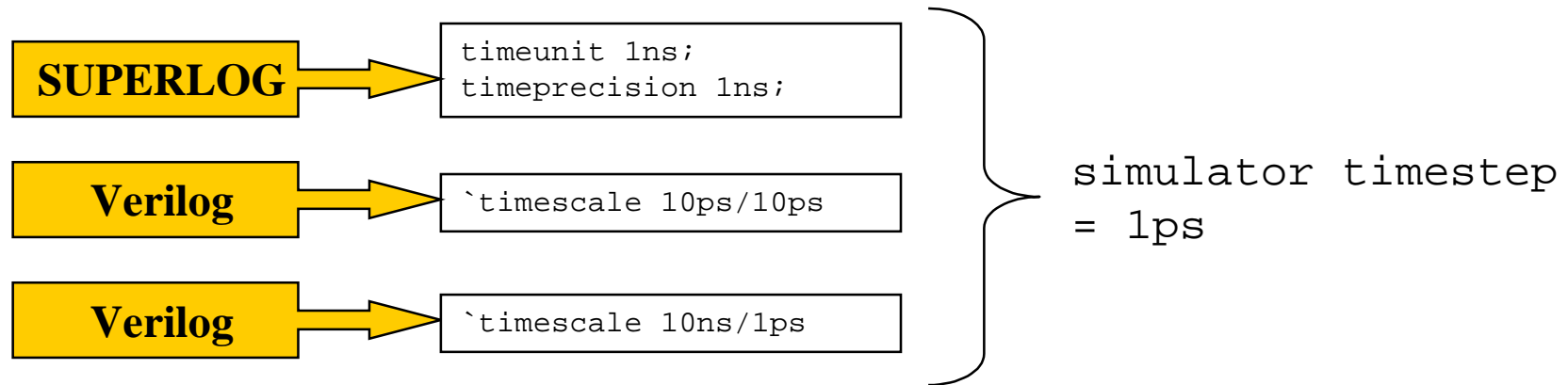
sub s;

initial
begin
    #1ns $display("top t=%t %f", $time, $realtime);
    #1fs $display("top t=%t %f", $time, $realtime);
end
endmodule
module sub;
initial
begin
    #1    $display("sub t=%t %f", $time, $realtime);
    #1fs $display("sub t=%t %f", $time, $realtime);
    #1000 $finish(0);
end
endmodule
```

**timeunit and
timeprecision must be
either at the global level
or right at the start of a
module**

**timeunit for sub is 1ns,
regardless of order of
modules**

Timescale Rules for Mixed Styles



•if SUPERLOG and Verilog timescales are mixed and there are one or more 'timescales in Verilog, the precision of the whole simulation is the smallest p in Verilog or SUPERLOG



Datatype System

Type and Variable System

- SUPERLOG is 100% backward compatible with Verilog 1995 and Verilog 2000
- It inherits the variable and type system from C
- Plus, it has additional types that are useful for system design and verification that go beyond C and Verilog

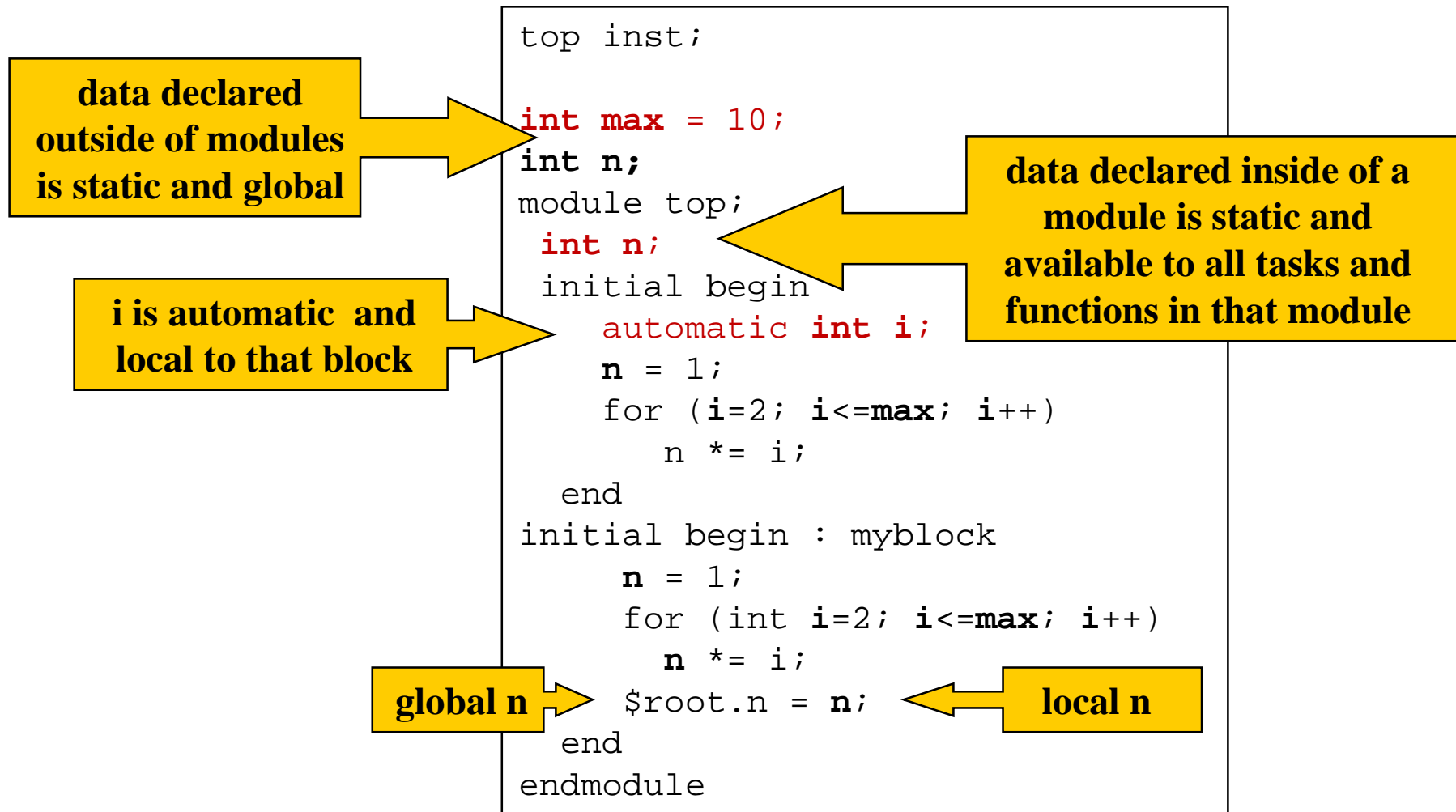
Variable Types – Static vs. Automatic

- **Static** variables
 - Allocated and initialized at time 0
 - Exist for the entire simulation
- **Automatic** variables allow recursive tasks and functions
 - Optional type for variables inside a block, task or function
 - Reallocated and initialized each time entering a block
 - May not be used to trigger an event

Variable Types – Global Vs. Local

- **Global variables**
 - Accessible from any scope
 - Must be static
 - Tasks and functions can be global too
- **Local variables**
 - Accessible at the scope where they are defined and below
 - Default to static, can made automatic
 - Static variables are accessible from outside the scope with a hierarchical pathname

Scope and Lifetime of Variables



Static and Automatic Variables in Tasks

```
task triple (input int n);  
automatic int result;  
int total;  
  
begin  
    total = total + 1;  
    result = 3 * n;  
    $display("result=%d.",result);  
    $display("Task called %0d times.",  
            total);  
  
end  
endtask
```

**static
variable**

**Default: all local
variables are static
(Verilog backwards
compatible)**

**Multiple calls of
the same task
in parallel all
access the same
static variables**

```
task automatic triple (input int n);  
int result;  
static int total;  
  
begin  
    total = total + 1;  
    result = 3 * n;  
    $display("result=%d.",result);  
    $display("Task called %0d times.",  
            total);  
  
end  
endtask
```

**automatic
variable**

**Use automatic
tasks to make all
local variables
automatic, unless
specified as static**

SUPERLOG Has the Basic C Datatypes

- Compatible to C

```
char c;           // 8 bit signed integer
int i;            // 32 bit signed integer
```

- Use typedef to get C compatibility

```
//          Superlog          C
typedef      shortint          short;
typedef      longint           longlong;
typedef      real              double;
typedef      shortreal         float;
```

Basic SUPERLOG Datatypes

```
bit b;      // single bit 0 or 1
logic w;    // 4-valued logic, x 0 1 or z as in Verilog
time t;     // 64 bit time value
byte b8;    // 8 bit signed integer
```

**Arrays of logic and bit
default to unsigned**

**Extra
SUPERLOG
datatypes**

- **Make up your own types with typedef**
- **Define arrays of bits and logic**

- **Flexibility**

SUPERLOG Has 2 and 4 State Datatypes

- Verilog
- SUPERLOG

```
reg a;  
integer i;
```

Verilog reg and integer
type bits can contain x
and z values

- SUPERLOG

```
logic a;  
logic signed [31:0] i;
```

Equivalent to these 4
valued SUPERLOG
types

- SUPERLOG

```
bit a;  
int i;
```

These SUPERLOG types
have two valued bits
(0 and 1)

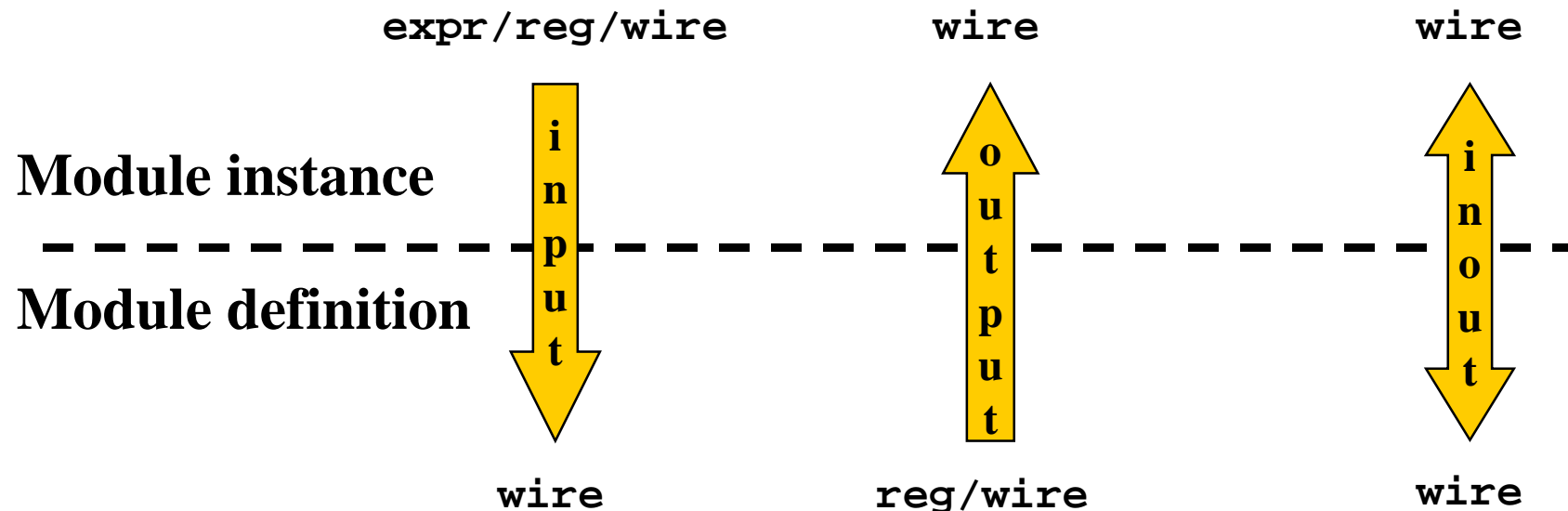
If you don't need the x and z values then
use the SUPERLOG bit and int types
which make execution much faster and
uses only half the memory

Data Types and Ports

- Verilog has 2 basic connection types
 - Nets
 - Represents a connection of one or more data drivers to a destination
 - Does not store data, just transfers it
 - Is the only type that goes through a port
 - Registers
 - Represents a place to store a value, a variable
 - Although a reg can be on either side a port, it is converted to a wire before going through it

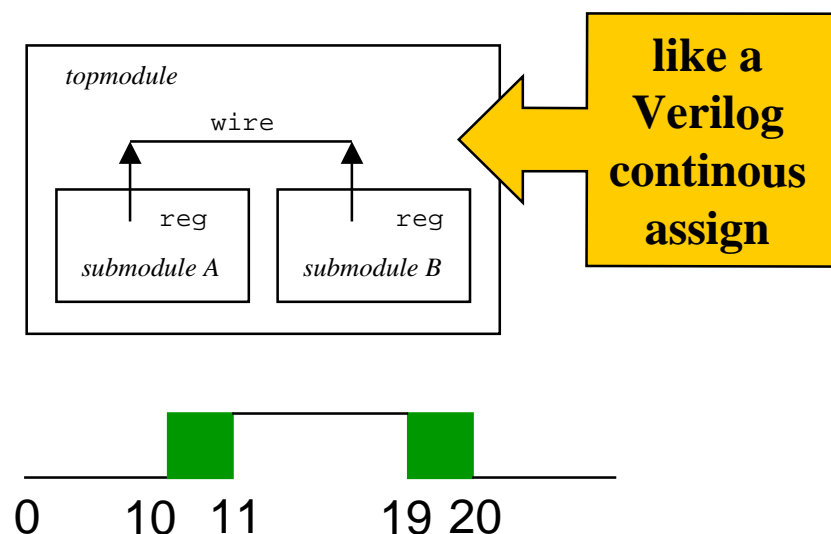
Verilog Module Ports Rules

- Expressions may contain compatible types
- When connected to a SUPERLOG module, **reg** may be replaced with **logic** or **bit**



Connecting Module Ports – Verilog Style

- Verilog wire vs. reg rules
- wires resolve all drivers
- A.r and B.r are separate



```
module topmodule;
wire w;

subA A(w);
subB B(w);

endmodule

module subA(output reg r)
initial fork
    #00 r = '0;
    #11 r = 'z;
    #19 r = '0;
join
endmodule

module subB(output reg r)
initial fork
    #00 r = 'z;
    #10 r = '1;
    #20 r = 'z;
join
endmodule
```

The SUPERLOG Type

- Works as either a register or a simple net
- Can be any SUPERLOG datatype

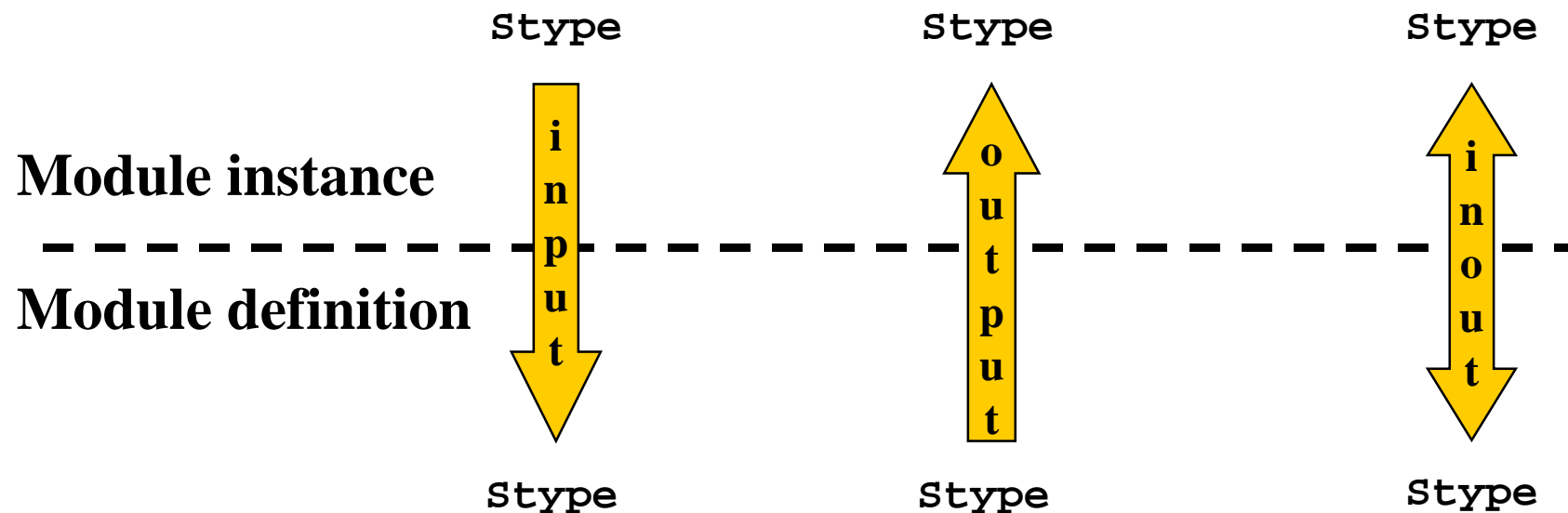
```
typedef struct {  
    real R;  
    real I;} Complex;  
Complex X,Y,Z;  
always @(negedge clk)  
begin  
    X = Complex_F1(Z);  
    Y = Complex_F1(Z);  
end  
always @(posedge clk)  
    X = Complex_F1(Y);  
assign Z = Complex_F1(X);
```

**One or more procedural
assignments to X,Y**

**Single continuous
assignment to Z**

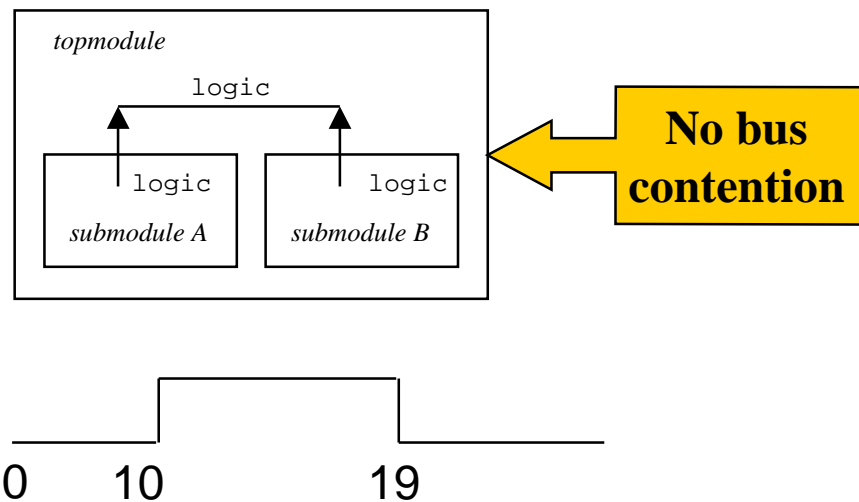
SUPERLOG Module Ports Rules

- A variable of any SUPERLOG type can pass through a port
- If the types are the same, the variable is shared
- If not, a continuous assignment is made



Connecting Module Ports - SUPERLOG

- Connected by sharing
- A.r, B.r, and w are collapsed into one variable



```
module topmodule;
  logic w;

  subA A(w);
  subB B(w);

endmodule

module subA(output logic r)
  initial fork
    #00 r = '0;
    #19 r = '0;
  join
endmodule

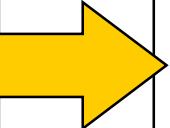
module subB(output logic r)
  initial fork
    #10 r = '1;
  join
endmodule
```

Signed Data

- Affects padding of operands in expressions
 - Unsigned – zero extended
 - Signed – sign bit extended
 - Affects Compare, shift, arithmetic
- Sign of lhs assignment does not effect rhs
- Defaults
 - All bit-level types and time are unsigned

Signed Data

**data
declaration
with 'signed'
or 'unsigned'**



**prefix format
with 's' for
signed literals**



```
int signed sint;
int unsigned uint;

logic signed [7:0] slogic;
logic unsigned [7:0] ulogic;

sint = -6;          // ffffffff
uint = -6;          // ffffffff
slogic = 4'hf;      // 0f unsigned constant
ulogic = 4'shf;     // ff signed constant

(sint + ulogic) is 1000000f9

(sint < slogic) is true
(uint < slogic) is false
```

Casts and Typedefs

```
int i;  
real f = 3.1515;  
  
...  
i = int'(f * 0.5);
```

type cast

complex type casts are possible but need a typedef

```
typedef logic [31:0] address_bus_type;  
address_bus_type address_bus;  
...  
address_bus = address_bus_type'(i);
```

**cast i as type
address_bus_type**

Declaration and Initialization

From C

```
int index = 1;
```

**one-line declaration and
initialization does not
generate an event**

```
int index;  
initial index = 1;
```

**This generates an event
at time 0**

**From
Verilog**

```
wire w = a & b;
```

**Wire can be declared
and continuously assigned
in one line**

```
wire w;  
assign w = a & b;
```

Same thing in 2 lines

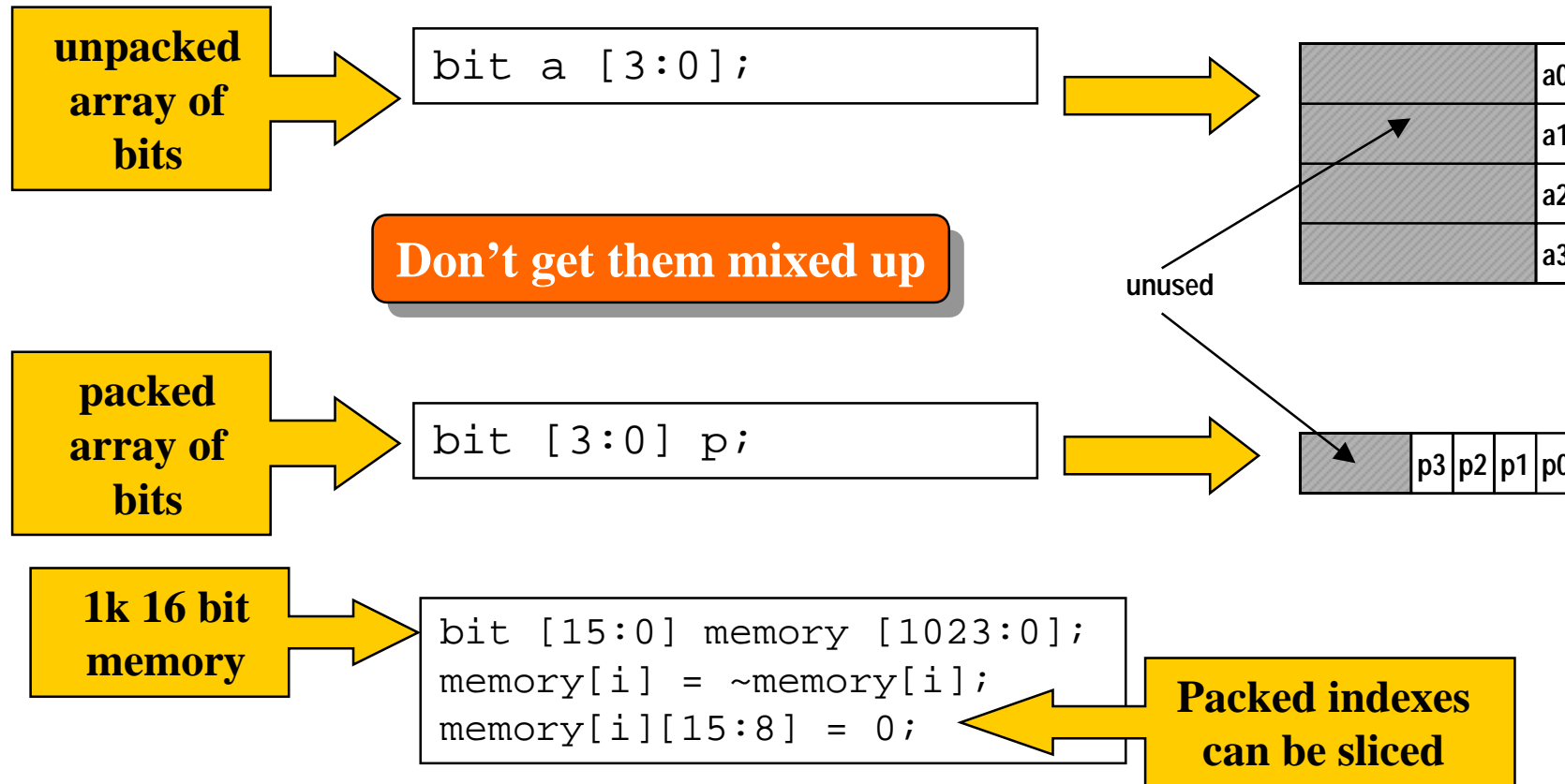


Arrays Structures

Packed and Unpacked Arrays

- Unpacked
 - Can be any datatype
 - Access only one element at a time
 - Whole arrays can be copied
 - Uses a range: `int Mem[1023:0]`
 - C uses size: `int Mem[1024]`
- Packed
 - All bit-level types: `reg`, `wire`, `logic`, `bit`
 - Access whole array or slice as a vector

Packed and Unpacked Arrays



Multidimensional Arrays

2D array

```
int a[7:0][7:0];  
...  
a[x][y] = 0;
```

**arbitrary
dimensions
possible**

3D array

```
bit ucube [maxx:0][maxy:0][maxz:0];  
bit [maxz:0] pcube[maxx:0][maxy:0];
```

```
ucube[x][y][z] = 5;  
pcube[x][y] = 0;  
pcube[x][y][1:0] = '1;  
pcube[x][y][2:0]++;
```

**Unpacked indexes
must be specified**

**Packed
dimension varies
more rapidly**

**Last packed index
may be a part select**

- **Complex data structures are easily defined**

Array Literals

Initialization

```
int A[2:0] = {0,1,2};  
int nines[1:9] = {9{9}};
```

**like
concatenation**

Assignment

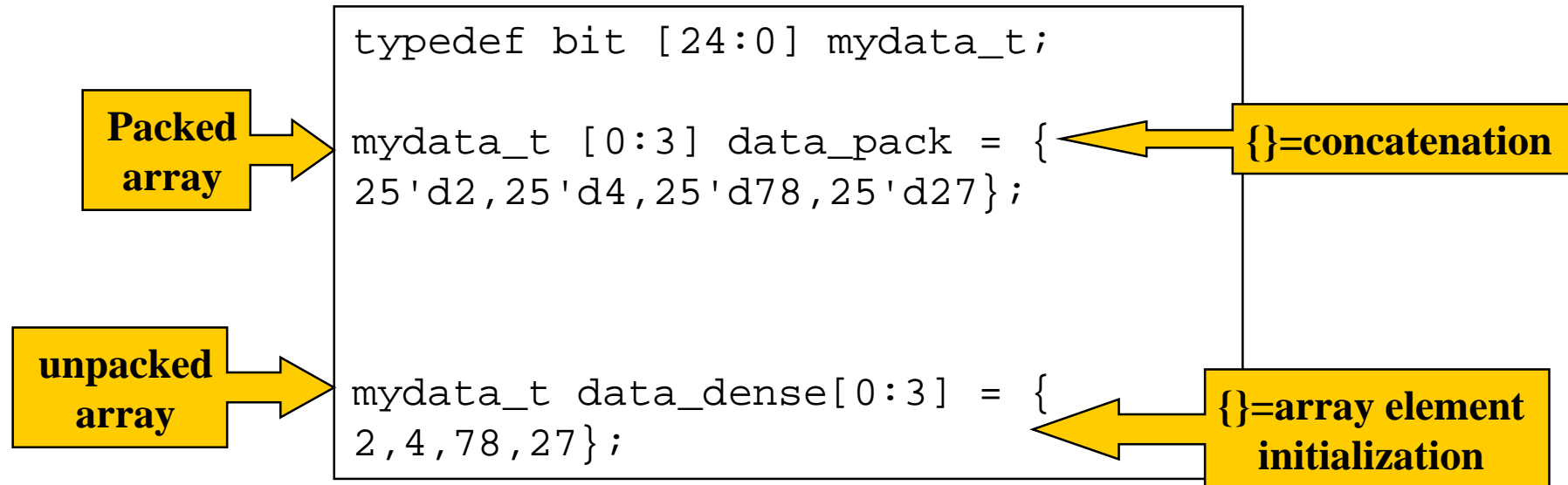
```
A = {3,4,5};
```

**Braces reflect
array layout**

```
real R[1:0][1:0]  
= {{1.5,4.5},{5.0,0.5}};
```

```
byte Frame[WIDTH:1][HEIGHT:1]  
= {WIDTH{{HEIGHT{8'hA5}}}};
```

Array Initialization



**Careful! Array
initialization with {} means
different things for packed
and unpacked arrays**

Strings

```
string s1, s2;
```

```
s1 = "";
```

← null string

```
s2 = "ab";
```

← copy string literal

```
s1 = {s1, s2};
```

← concatenation

```
$display("s1=%s, s2=%s", s1, s2);
```

The format string
of \$display must
be a literal

← display

```
if (s1 == s2) ...
```

The string data type is a variable size
packed array of characters, indexed
from 0 to \$, with the special property
that an element of the array is also
of type string

- Strings are an extension to both Verilog and C

Enumerated Data Types

```
typedef enum {bashful, doc, dopey, grumpy,  
             happy, sleepy, sneezy} dwarf_type;  
dwarf_type dwarf;  
  
enum {yes, no=0, maybe} choice;
```

Typedef

**User defined
encoding**

```
dwarf = dopey;  
choice = maybe;  
$display("dwarf=%s choice=%s", dwarf, choice );  
$display("dwarf=%d choice=%d", dwarf, choice );
```

**can print
enums as text
or as number**

```
dwarf=dopey choice=maybe  
dwarf=          2 choice=          2
```

```
dwarf=dwarf_type'(2);  
dwarf=dwarf_type'(no);  
myint=int'(happy);
```

Must use cast

Unique enum Elements

```
typedef enum {  
    r_street, r_alley, r_square  
} roads_e;  
  
typedef enum {  
    g_rectangle, g_square, g_triangle  
} geometric_e;
```

Enum elements must be unique to the current scope

```
begin: en1  
    enum { street, alley, square } roads_e;  
end  
  
begin: en2  
    enum { rectangle, square, triangle }  
    geometric_e;  
end  
  
en1.roads_e = square;  
en2.geometric_e = square;
```

If ambiguous elements are required, put the enums into separate scopes (blocks, modules, or interfaces)

Structures

```
typedef struct {  
    real F0, F1;  
    int  I0, I1;  
    Instruction IR;  
} reg_bank;
```

structure

**Like in C but without
the optional structure
tags before the {**

```
type reg_bank is  
record  
    F0, F1 : Real;  
    I0, I1 : Integer;  
    IR: Instruction;  
end record;
```

**VHDL
Record**

- **Flexible datatypes, compact**

More Structures

```
typedef struct {  
    byte R,G,B;  
} RGB;
```

```
const RGB BLUE = {0,0,255};
```

```
RGB Frame[640:0][480:0];
```

```
Frame[x][y] = BLUE;
```

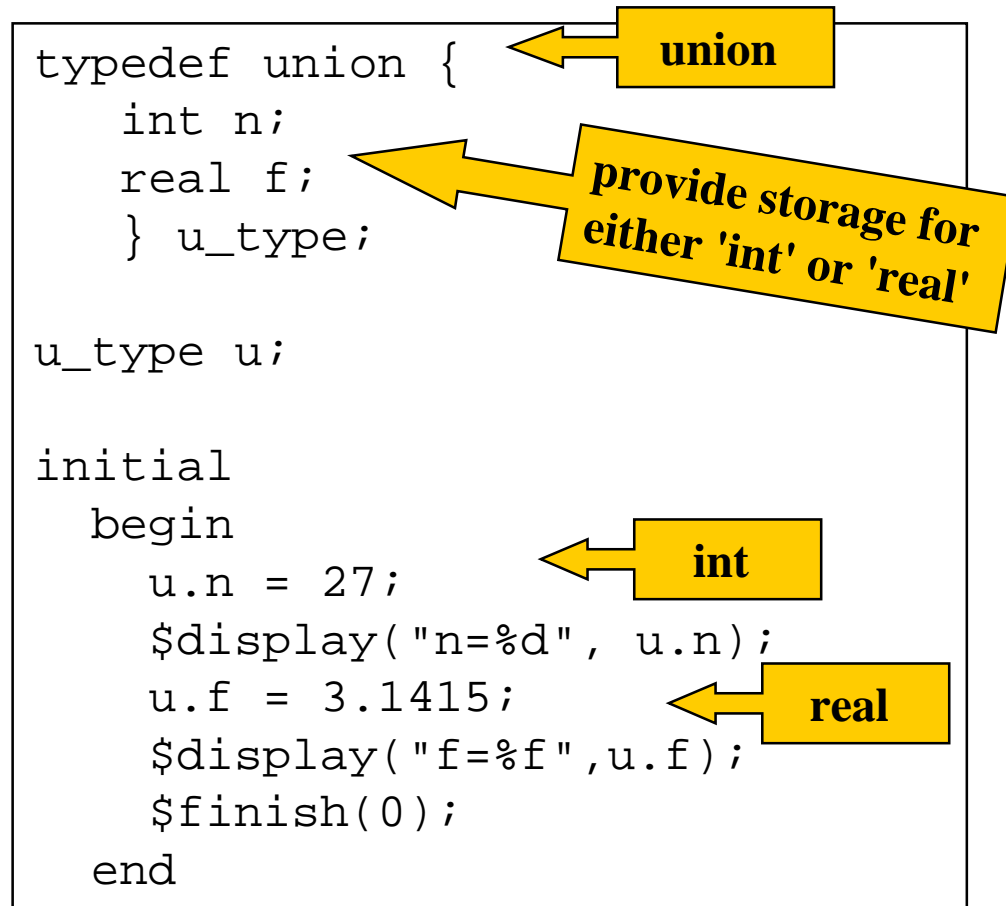
```
module Xform(input RGB pixin,  
             output RGB pixout);  
endmodule
```

constant

array of struct

struct copy

Unions



again, like in C

- structs and unions can be assigned as a whole
- Can be passed through tasks/functions/ports as a whole
- can contain fixed size packed or unpacked arrays
- pointers

Struct and Union Example

```
typedef struct {  
    bit is_real;  
    union { int i; real f;} n;  
} number_type;  
  
number_type num;  
  
initial  
begin  
    assign_real(num, -3.1415);  
    printhum(num);  
  
    assign_int(num, 1024);  
    printhum(num);  
  
    $finish(0);  
end
```

```
task printhum(input number_type num);  
    if (num.is_real)  
        $display("num=%f (real)", num.n.f);  
    else  
        $display("num=%d (int)", num.n.i);  
endtask  
  
task assign_real(output number_type num,  
                 input real f);  
  
    begin  
        num.is_real = 1;  
        num.n.f = f;  
    end  
endtask  
  
task assign_int(output number_type num,  
                input int i);  
  
    begin  
        num.is_real = 0;  
        num.n.i = i;  
    end  
endtask
```

Dynamic Types

```
typedef struct {
    bit is_valid;
    dynamic data;
} number_type;

number_type num;

initial
begin
    assignnum(num, -3.1415);
    printhnum(num);

    assignnum(num, 1024);
    printhnum(num);

    $finish;
end
```

```
task assignnum(output number_type num, input dynamic d);
if (d.$type == int || d.$type == real)
begin
    num.data = d;
    num.is_valid = 1;
end
else
    num.is_valid = 0;
endtask

task printhnum(input number_type num);
if (num.is_valid)
case (num.data.$type)
    real: $display("num=%f (real)", real'(num.data));
    int:  $display("num=%d (int)",  int'(num.data));
endcase
else
    $display("data is not valid");
endtask
```

Must cast

- Polymorphism in SUPERLOG

**Task call be called
with different
argument types**

Varargs and Structure Literals

```
typedef struct { byte header;  
                bit [31:0] payload; } cell_type;
```

```
task calc_check_sum (output unsigned byte checksum, ...);  
  cell_type cell;  
  checksum = 0;  
  for (int i = 0; i < $varargc; i++)  
    begin  
      cell = cell_type'($varargv[i]);  
      checksum += cell.header;  
    end  
endtask
```

varargs

cast vararg to cell type

```
unsigned byte checksum;  
cell_type c2;
```

```
cell_type c1 = {8'ha7, 32'hfeeble00 };  
c2 = {8'ha0, 32'hbeefeale };
```

```
calc_check_sum(checksum, c1);  
calc_check_sum(checksum, c1, c2);
```

Structure initializer

Structure literal

**Task can be called with
variable number of arguments**



Pointers Queues

Pointers

- SUPERLOG pointers point to any datatype
 - Pointers can also reference any SUPERLOG object in a design
 - tasks, functions, processes, module instances
- More efficient to pass a pointer to a large structure than copying it
- Uses safe pointers to make code easier to debug

Safe Pointers

- A SUPERLOG safe pointer always points to a valid reference or is null
- Runtime warnings are generated for
 - Dereferencing an uninitialized or null pointer
 - Deallocating memory when other pointers still have references
- Can be turned off later to improve performance

Pointer Syntax

- **ref** means "create a reference to"
 - Declaration: **ref** int iptr;
 - Note: Cannot mix pointer declarations with regular variables
 - In an expression: iptr = **ref** i;
 - Works the same as &i in C
- **deref** means "the object pointed to by"
 - In an expression: **deref** iptr = i;
 - Works the same as *iptr = in C

Pointers and Structures

- Pointers to structures need separate typedefs
- A structure can have a pointer to its own type

```
typedef struct {  
    int n;  
    ref number_type next;  
} number_type;
```

**structure for a
linked list**

```
task printlist(input ref number_type pstart);  
automatic ref number_type p = pstart;  
while(p)  
begin  
    $display("value=%d", p->n);  
    p = p->next;  
end  
endtask
```

**walk through
the linked list**

Pointer Differences to C

C

```
main () {  
  
    typedef struct tmp {  
        int value;  
        char c;  
        struct tmp *next;  
    } struct_type;  
  
    struct_type st;  
    struct_type *p;  
  
    p = &st;  
    p->value = 17;  
    printf("value=%d\n", p->value);  
    printf("value=%d\n", (*p).value);  
    exit(0);  
}
```

C structure tag

SUPERLOG

```
typedef struct {  
    int value;  
    char c;  
    ref struct_type next;  
} struct_type;  
  
struct_type st;  
ref struct_type p;  
  
p = ref st;  
p->value = 17;  
$display("value=%d", p->value);  
$display("value=%d", (dref p).value);  
$finish(0);
```

**can use
struct_type in
the definition**

Pointers and Arrays

- Pointers can reference an unpacked array element
 - The array name is not an pointer as in C
- Only pointers to an array can be incremented and decremented

```
int a[1:10]; // declares an array of integers
ref int p;   // declares that p will point to an integer
p = ref a[1]; // sets pointer to address of a[1]
p++;          // increments pointer to address of a[2]
deref p = 1; // same as a[2] = 1
```

Dynamic Memory Allocation

```
typedef struct {  
    int n;  
    bit [31:0] field;  
} data_type;
```

**a structure as
a datatype**

```
ref data_type p, p2;  
  
p = $alloc(data_type);  
  
p->n = 27;  
  
p2 = p;
```

**p is a ref to
data_type**

**\$alloc returns a
pointer to a
'data_type' item**

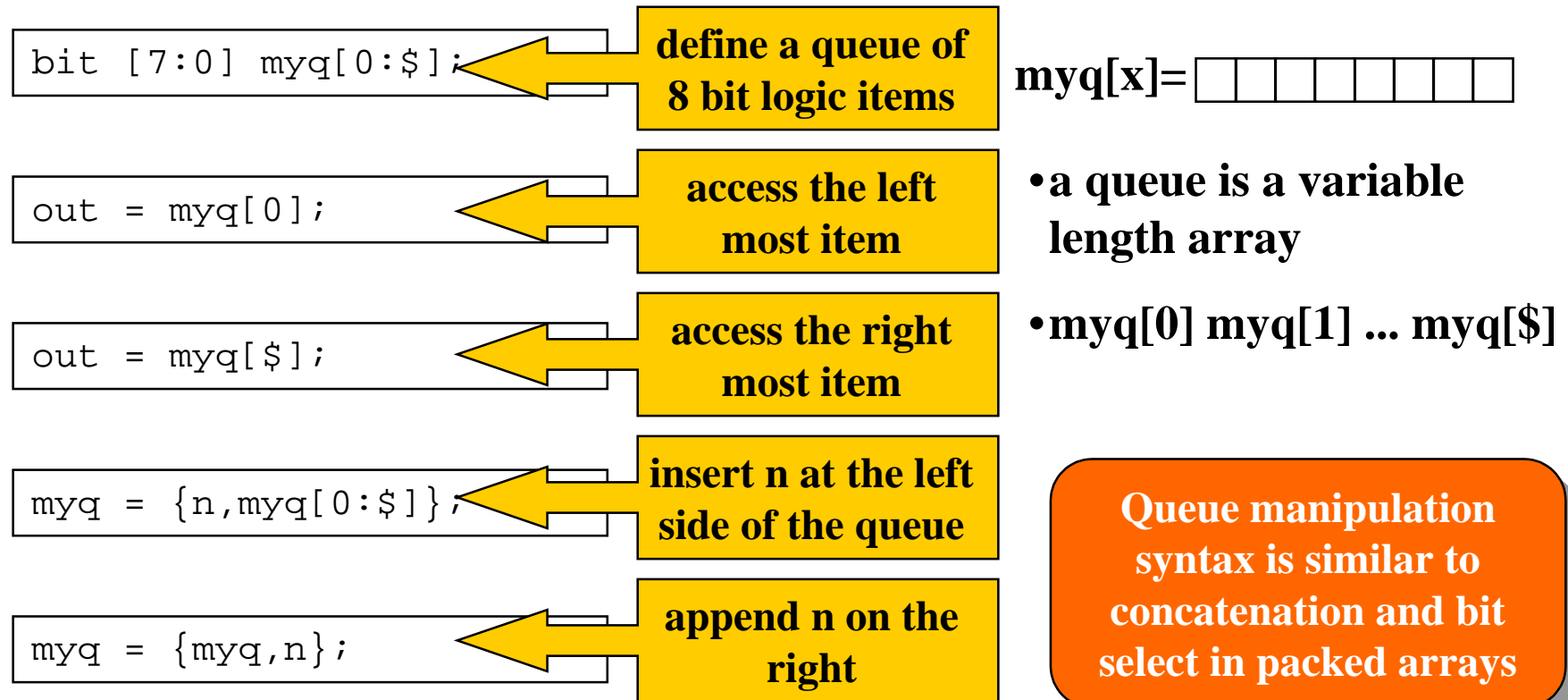
**a second ref
p2 to the same
location**

Dynamic Memory Deallocation

Statement	Safe Mode Action	Safe Mode Warning
<code>\$free(p)</code>	All pointers to memory set to null	Other pointers to memory displayed
<code>\$delete(p)</code>	All pointers to memory set to null	
<code>p = NULL;</code> <code>p2 = NULL;</code>		Memory leak

- **Safety has a performance hit**
- **Enabled/disabled by the simulator**

Queues



- **Concise, Simple, Powerful. Intuitive Syntax.**

More Queues

```
typedef struct {bit [9:0] addr;  
               bit [51:0] data;  
} packet;  
packet qp [0:$];
```

Define a queue of packets

```
qp = qp[1:$];
```

Delete the left most item

```
qp = qp[0:$-1];
```

Delete the right most item

```
n_items = qp.$num;
```

**Get the number of items in
the queue**

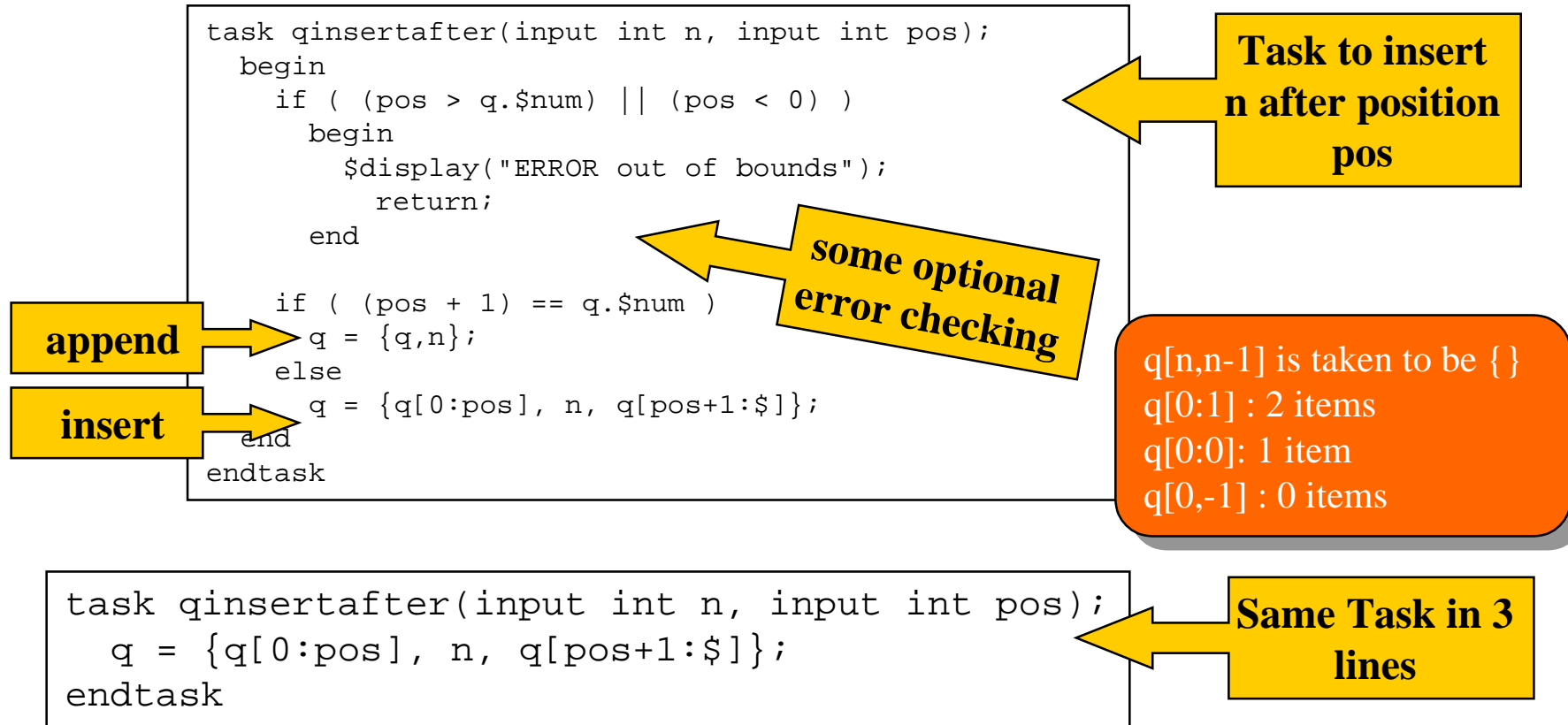
```
for (int i=0; i<qp.$num; i++)  
    qp[i] = ...
```

**To step through the queue
use an integer index**

```
qp = {};
```

Delete the whole queue

Queue As a List



- Queue operations can be encapsulated in tasks

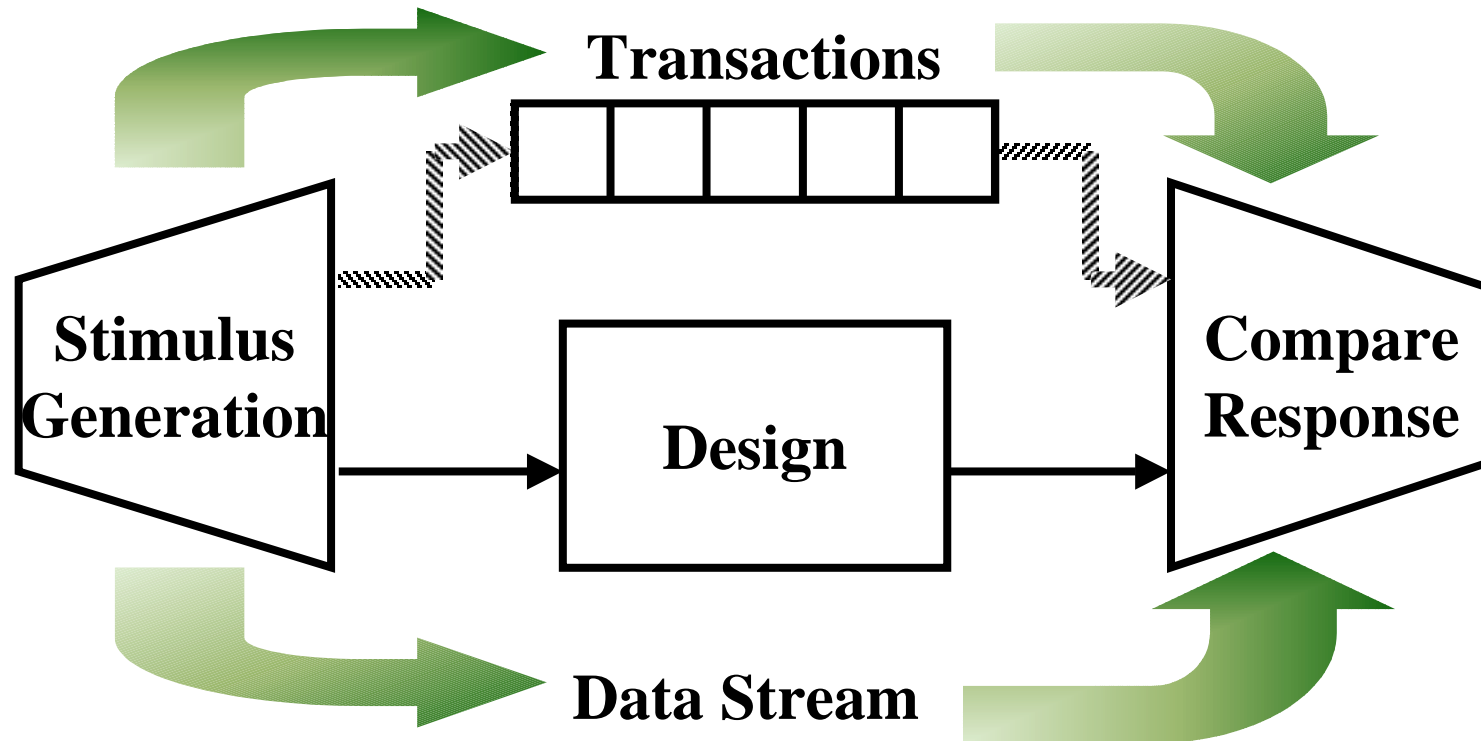
Queue As a Fixed Length Shift Register

```
module shiftreg(input bit clk,  
               input logic [7:0] in,  
               output logic [7:0] out );  
  
parameter      int delay = 24;  
logic [7:0] q [0:$];  
  
initial  
    repeat(delay) q = {8'bx, q};  
  
always @(posedge clk)  
begin  
    out = q[$];  
    q = {in, q[0:$-1]};  
end  
  
endmodule
```

**Fill up the
queue**

**Push and Pop
items**

Queues for Verification




- Transactions span many clock cycles
- Queues accommodate variable latencies from input to output


Queues and Pointers

```
typedef struct {  
    int field1;  
    int field2;  
} packet_s;  
typedef ref packet_s block_p;  
block_p bp_q[0:$]; //global
```

Stimulus

```
block_p b_p;  
  
always  
begin  
    b_p = $alloc(packet_s);  
    bp_q = {bp_q, b_p};   
    send_block(b_p);  
end
```

Response

```
block_p b_p;  
  
always  
begin  
    @(posedge clk iff bp_q.$num)  
    b_p = bp_q[0];  
    bp_q = bp_q[1:$];   
    receive_block(b_p);  
    $free(b_p);  
end
```

- Move pointers instead of data
- Performance gained from high level abstraction

Operators

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

**bump
operators**

```
x++;  
if (++c > 17) c=0;
```

```
a += 3;  
s = 1 << n;
```

```
sig_a = x && y;  
sig_o = x || y;
```

mux

```
i = (s)?b:a;
```

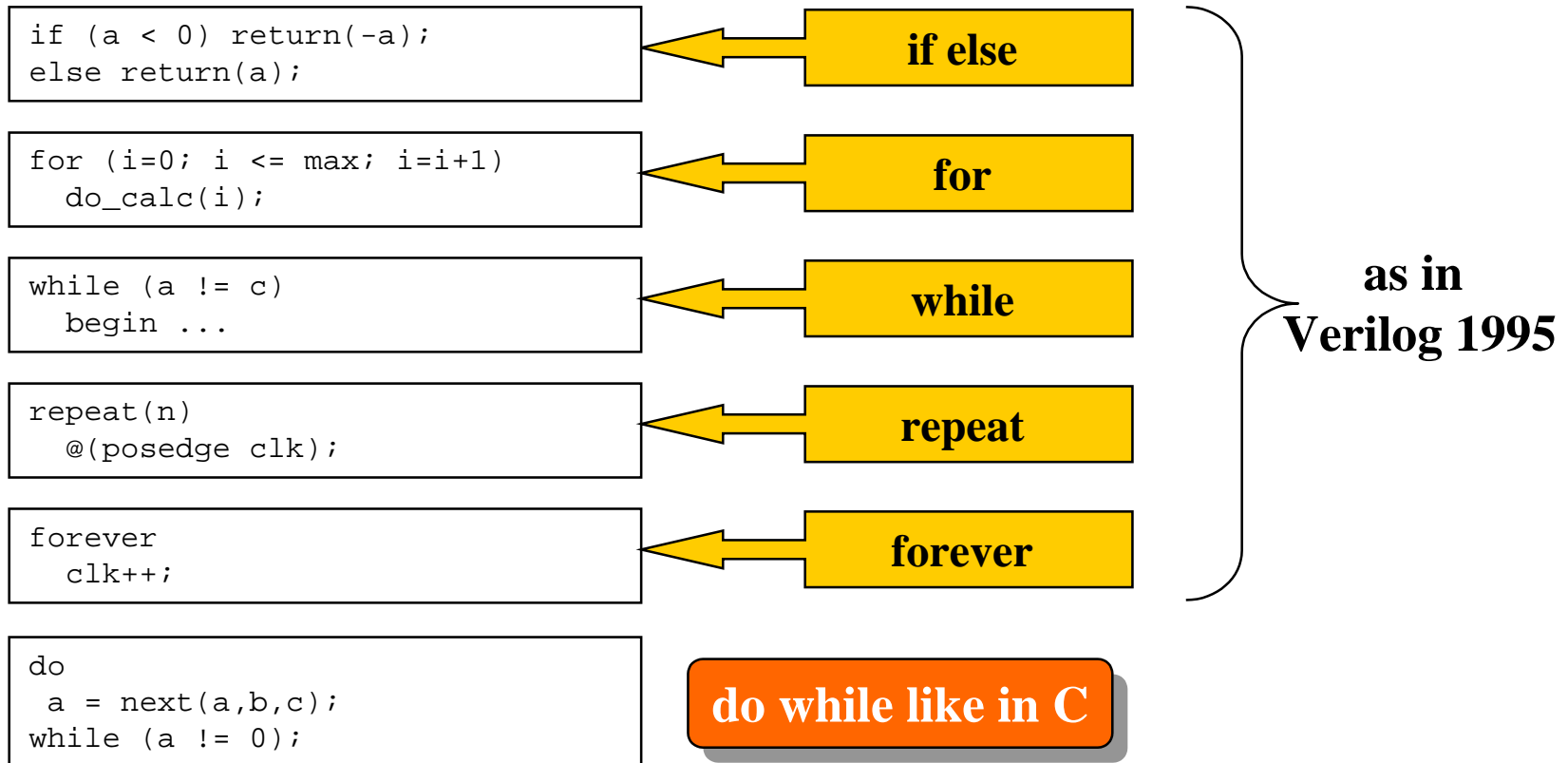
•All the operators you would expect from Verilog and C

- The power of C and Verilog



Procedural Constructs

Branch and Loop Constructs



Event Control

```
byte i;  
initial  
    repeat (256) #10 i = $random;  
  
always @(written i)  
    $display("i has been assigned to %d, i);  
  
always @(changed i)  
    $display("i changed value to %d", i);
```

**triggers whenever i is
written to, even if it does
not change value**

**triggers only when i
changes value**

**always @(i)
is the same as
always @(changed i)**

- **Useful for Transaction based designs**

Event Control

- Can be used to trigger an event
 - Static variables
- Can not be used to trigger an event
 - Automatic variables
 - Imported C variables

```
@(variable)           // event control
wait(variable)

assign 0 = variable;   // continuous assignment

mp inst(...,variable,,); // input to module or primitive
```


Always Block Enhancements

**Works like
in Verilog**

```
always @(b or c)
  a = b & c;
```

**Signal on the LHS
must not be
written to
anywhere else**

```
always_comb
  a = b & c;
```

**Sensitive to all the
signals on the RHS**

infers a latch

```
always_latch
  if(ck) q = d;
```

- **Better control of Synthesis, more security**

- **always_comb**
- **always_latch**
 - always with an implicit sensitivity list

Comparison of Sensitivity List Styles

Initially x, because the block only starts evaluating on the first change of b or c

```
logic a;  
always @(b or c)  
    a = b & c;
```

Sensitive to all the signals on the right

```
logic a;  
always @(*)  
    a = b & c;
```

Shorthand form

Proper wired logic behavior: the signal is evaluated at time 0 and whenever b or c change

```
logic a;  
always_comb  
    a = b & c;
```

SUPERLOG
always_comb also evaluates at time 0

```
wire w;  
assign  
    w = b & c;
```

Verilog style assign for wires

Simulation vs. Synthesis Results

- Synthesis pragmas cause results to diverge
- SUPERLOG adds verifiable synthesis constructs
- priority case
 - Tests each case condition in order and makes sure there is at least one branch taken
- unique case
 - Tests all case conditions and makes sure that one and only one condition matches

Priority and Unique Case

```
priority casez (b)
  4'b0000: $write("zero");
  4'b???1: $write("odd");
endcase
```

**A non-zero even
number has no match**

```
unique casez (b)
  4'b0000: $write("zero");
  4'b???1: $write("odd");
  4'b???0: $write("even");
  default: $write("other");
endcase
```

**"0" matches
zero and even**

Simulation pre and post synthesis agree

- No need for 'parallel_case' synthesis pragmas

Priority and Unique If

```
unique if (!t[0])  
    $display("even");  
else if (t[1])  
    $display("big");  
else $display("small odd");
```

**unique if means
that the else if
conditions do not
overlap**

**priority if gives warning
if there is no final else
but the else condition
happens**

**•Functions same as
priority/unique case**

- **Extra error checking**

Jumps

- Use break and continue in loops just like in C

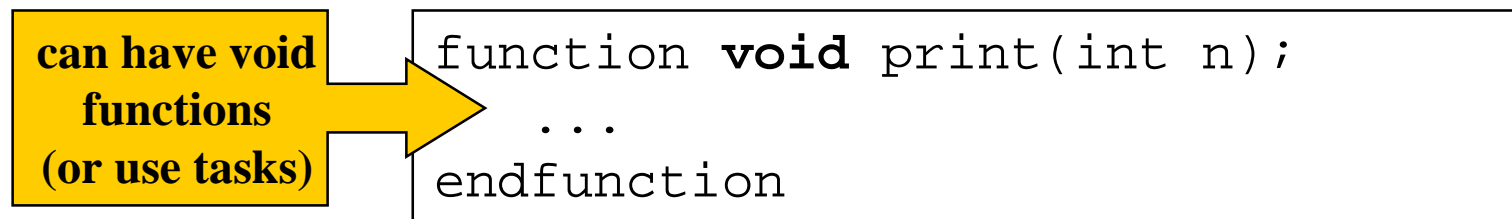
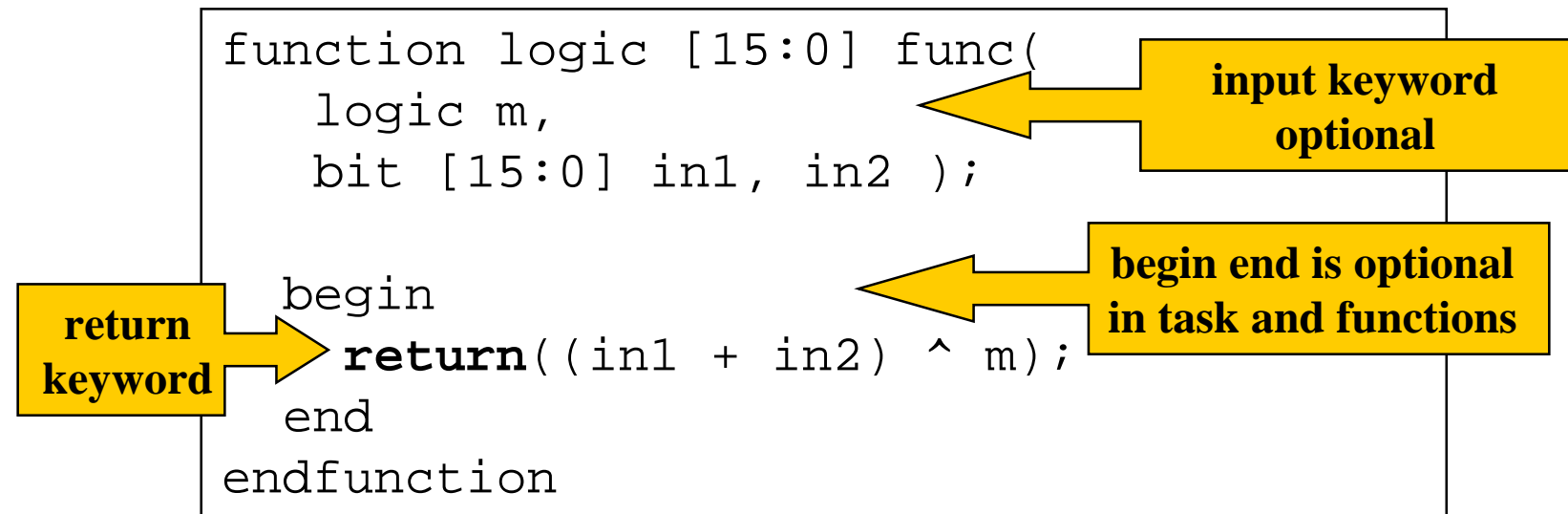
```
forever
begin
  n = n + 1;
  if ( (n%3) == 0 ) continue;
  $display("n is %d", n);
  if (n == 22) break;
end
```

**continue starts
the next loop
iteration**

**break breaks
out of the loop**

works with:
for
while
forever
repeat
do while

Functions




- **SUPERLOG functions can have output arguments**

Recursive Functions

```
module top;
  int n;
  initial
    begin
      $display("The factorials from 1 to %d", MAX);
      for (n=0; n <= MAX; n = n + 1)
        $display("%d!=%d", n, factorial(n));
    end

  function automatic int factorial (int n);
    if (n==0) return(1);
    else return(factorial(n-1) * n);
  endfunction

endmodule
```



- Recursive functions and tasks are possible

Tasks With Return

```
module primesearch;
parameter int max = 20;
int primes[max:0];
int next;
initial
begin
    int i;
    primes[0] = 2; next = 1;
    for (i=3; next<=max; i = i + 2)
        test_prime(i);
end

task test_prime (input int try_this);
int i=0;
    while ( (i < next) && ( (2 * primes[i]) <= try_this ) )
        if ( try_this % primes[i++] == 0 ) return;

    $display("%d is prime", try_this);
    primes[next++] = try_this;
endtask

endmodule
```



**ANSI C style
header like for
modules**



**return
keyword**

Process Statement

```
always @(posedge clk) $display("tick...");
```

Monitor the clock

```
initial
```

```
begin
```

```
    $display("%t We are at time zero", $time)
```

```
    process forever #10 clk = !clk;
```

```
    $display("%t We are still at time zero",  
            $time);
```

**start process to
generate the clock**

```
end
```

```
0 We are at time zero
```

```
0 We are still at time zero
```

```
tick...
```

```
tick...
```

```
tick...
```

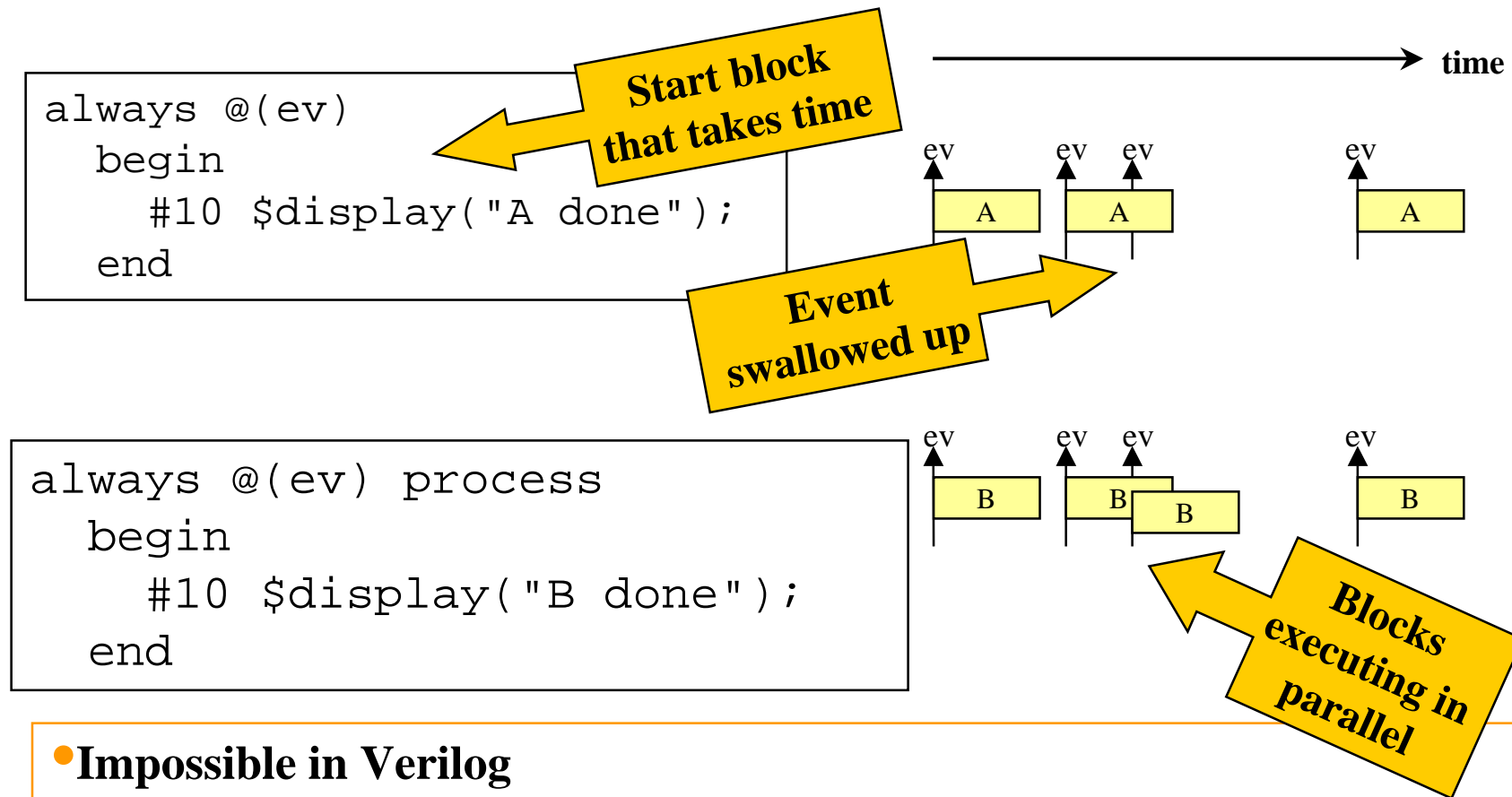
```
tick...
```

```
tick...
```

Output

**process is like a fork
without the join or the
& in Unix**

Retriggerable Always Block



- Impossible in Verilog
- Use with automatic tasks to create re-entrant code



Interfaces

Public versus Private Interface Constructs

- Only basic interface definitions presented in this tutorial
- Advanced communication capabilities and OO features, contained within full interfaces, currently available only under NDA
- Please contact Co-Design Automation for a full interface definition

What Is an Interface?

```
int i;  
logic [7:0] a;  
  
typedef struct {  
    int i;  
    logic [7:0] a;  
} s_type;
```

**At the simplest
level an interface
is to a wire
what a struct is to
a variable**

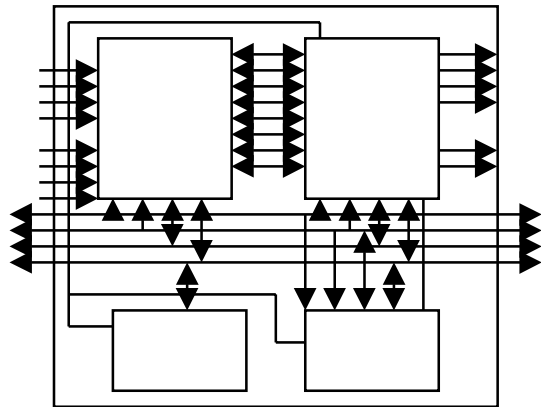
```
int i;  
wire [7:0] a;  
  
interface intf;  
    int i;  
    wire [7:0] a;  
endinterface
```

```
wire w;  
intf if1;  
  
modA a (w, if1);
```

**You can think of a
wire as a built in
interface**

- **Encapsulates communication like a struct encapsulates data**

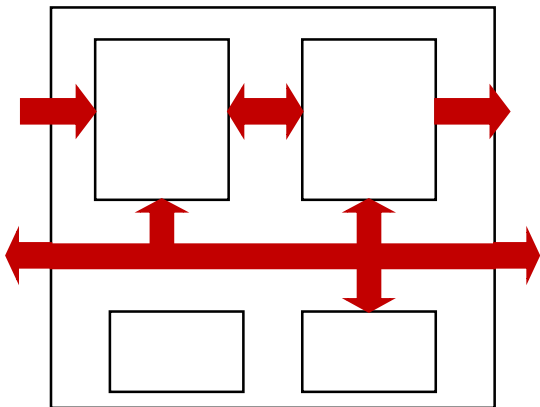
Systems Without the Interface Construct



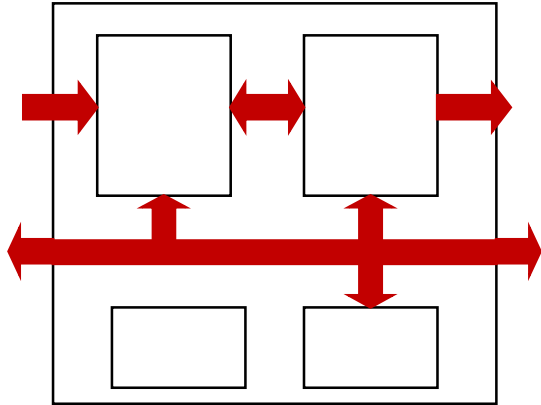
**schematics map
to a Verilog
netlist**

```
module mem_control(  
    input wire clk, global_reset,  
    inout wire [63:0] data_bus,  
    input wire ctl1, ctl2, ctl3,  
    input bit pre1, pre2, pre3,  
    input wire a0, a1, f_pdec,  
    input wire f_dsj, f_dip,  
    f_dil, g_ty0, g_ty1, g_ty2,  
    ....
```

**As in Verilog: Hierarchy is only
in the modules and not in the
interfaces which are all exploded
into separate wires**



Interfaces Reduce Interconnection Text



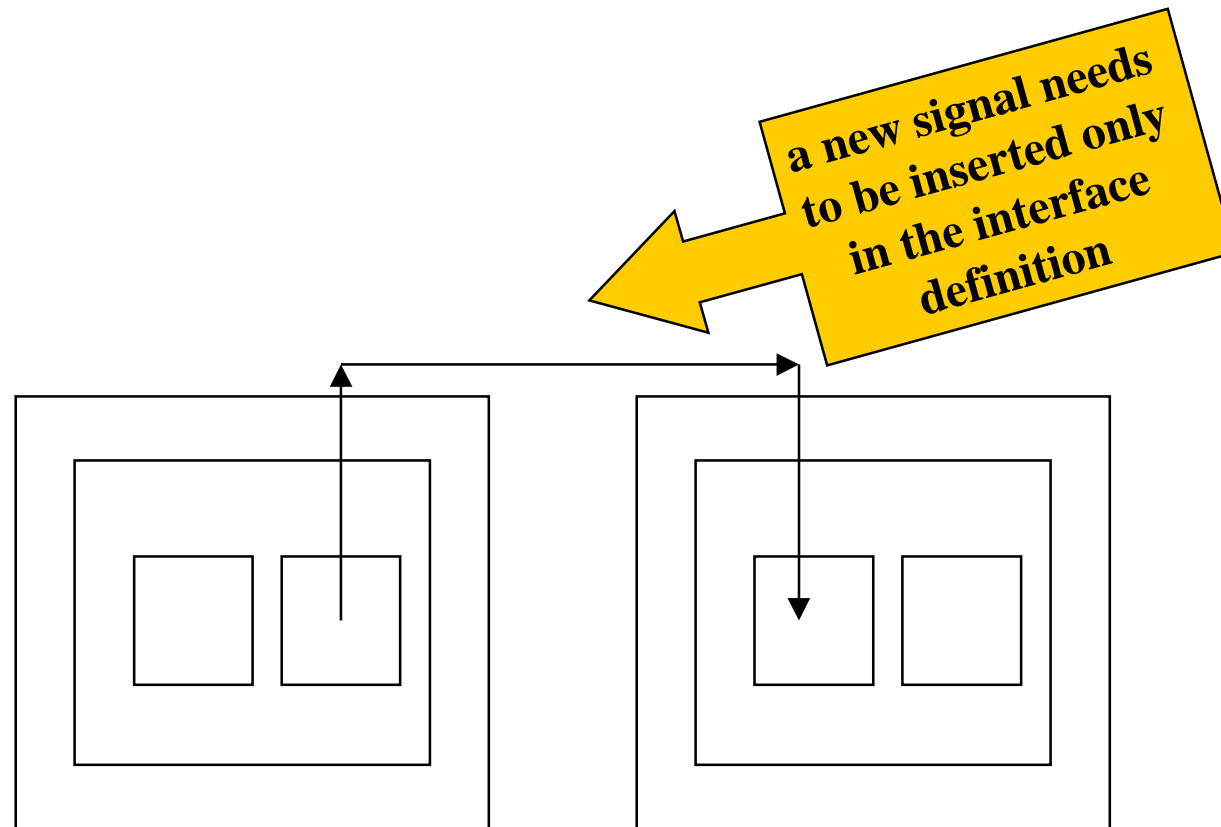
```
interface system_bus_intf;  
  wire [63:0] data_bus;  
  wire ctl1, ctl2, ctl3,  
  bit pre1, pre2, pre3,  
  
  ...  
endinterface
```

**The system block
diagram maps to the
SUPERLOG code
using interfaces**

```
module mem_control(  
  interface system_bus,  
  interface memory_interface,  
  ...  
endmodule
```

- Concise, maintainable and readable code

Interfaces Keep the Code Maintainable



- No need to edit dozens of files of intermediate levels to insert just one signal

Modports

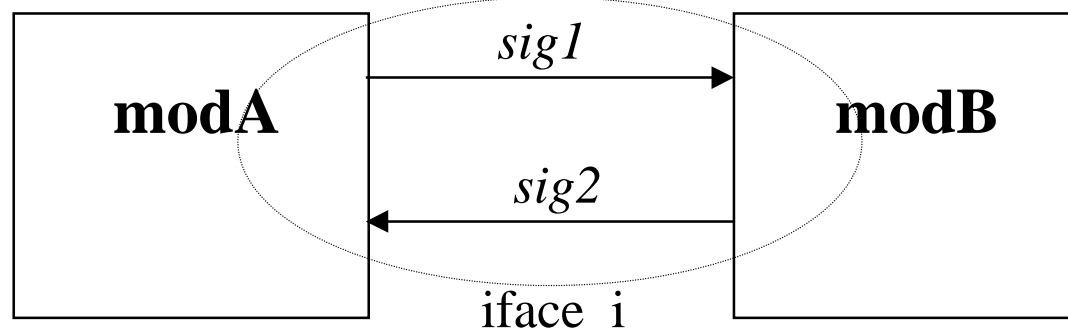
```
interface iface_i;  
  bit sig1, sig2;  
  int internal;  
  modport modeA(output sig1, input  sig2),  
               modeB(input  sig1, output sig2);  
endinterface
```

**Not visible outside
of this interface**

**Specifies the
accessibility and
direction of
interface signals**

```
module modA(iface_i.modeA iface);  
  ...  
  iface.sig1 <= 0;  
  w <= iface.sig2;  
  ...  
endmodule
```

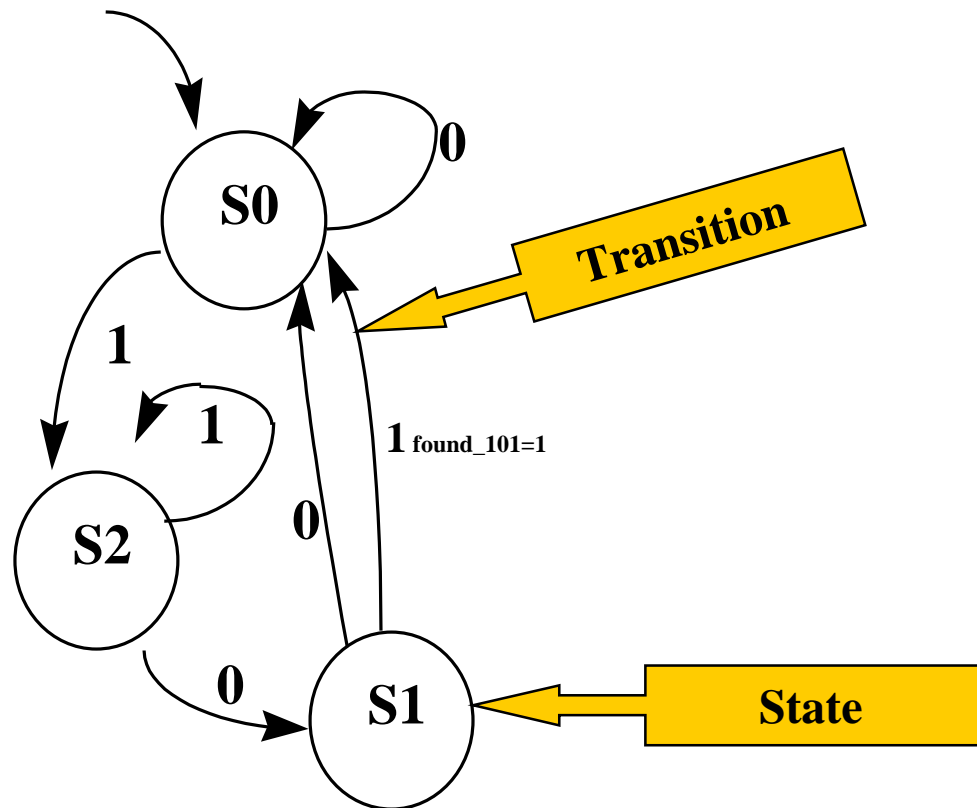
```
module modB(iface_i.modeB iface);  
  ...  
  iface.sig2 <= 1;  
  ...  
endmodule
```





Finite State Machines

Simple FSM Example



- Simple FSM example
- finds serial pattern 101

Explicit Style

```
module FSM_1( output logic found_101,  
              input logic serial, clk, reset);  
parameter bit [1:0] S0 = 0, S1 = 1, S2 = 2;  
bit [1:0] nextState;  
bit [1:0] currentState;  
  
always @(reset or serial or currentState) begin  
    found_101 = 0;  
    nextState = currentState;  
    if(reset) nextState = S0;  
    else case(currentState)  
        S0: if (serial == 1) nextState = S2;  
  
        S2: if (serial == 0) nextState = S1; else nextState = S2;  
  
        S1: begin nextState = S0;  
                if (serial == 1) found_101 = 1; end  
    endcase  
end  
  
always @(posedge clk) currentState <= nextState;  
endmodule
```

**need current and next
state variables**

**case statement for the
states**

18 lines

Transition Based Style

```
module FSM_5( output logic found_101,  
              input logic serial, clk, reset);  
state {S0, S1, S2} S;  
  
always @(posedge reset)  
    transition (S) default:->> S0;  
endtransition  
  
always @(posedge clk iff !reset)  
    transition (S)  
        S0:if (serial == 1) S0_S2 ->>S2;  
        S2:if (serial == 0) S2_S1 ->> S1;  
        S1:S1_S0 ->> S0;  
    endtransition  
  
always_comb found_101 <= S.S1 && serial;  
  
endmodule
```

13 lines

describe
transitions, rather
than states

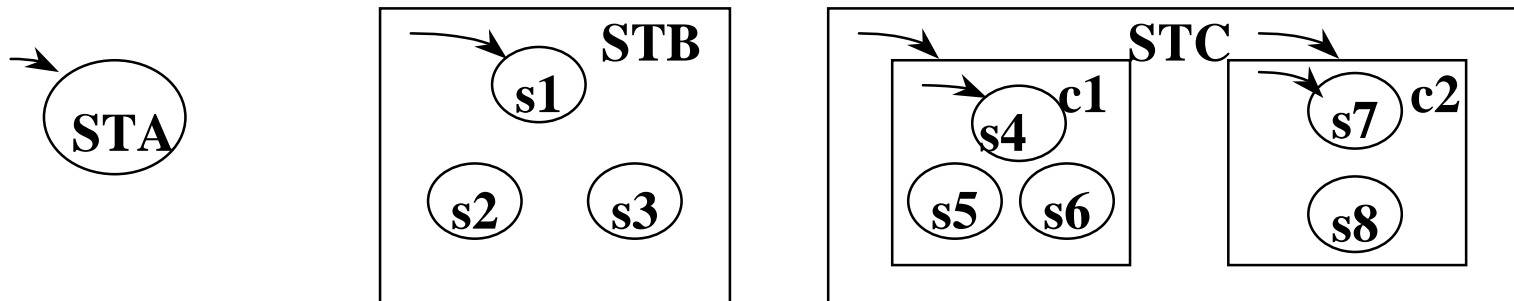
- Easily Human and machine understandable, compact

Hierarchical State Machines

- Structured state machine declaration

```
state {STA, {s1, s2, s3}STB,  
      { {s4, s5, s6}c1 and {s7, s8}c2 }STC  
}hfsm1;
```

Concurrent states



```
assign out=hfsm1.STB;  
assign out=hfsm1.s1 || hfsm1.s2 || hfsm.s3;
```

Equivalent statements

Vending Machine FSM Example

typedef

```
typedef enum { none, ten, five } coin_type;
```

states

```
state {s0, s5, s10, s15} st;
```

a state

```
always @(posedge clk)
  transition(st)
```

```
  s0:    if (coin == ten)  t1 ->> s10; fifteen <= 0;
         else if (coin == five) t2 ->> s5;  fifteen <= 0;
  s5:    if (coin == ten)  t3 ->> s15; fifteen <= 1;
         else if (coin == five) t4 ->> s10; fifteen <= 0;
  s10:   if (coin == ten)  t5 ->> s15; fifteen <= 1;
         else if (coin == five) t6 ->> s15; fifteen <= 1;
  s15:   t7 ->> s0;          fifteen <= 0;
endtransition
```

```
endmodule
```

named transition

action after a transition

Vending Machine Testbench

```
always #100 clk = !clk;  
  
vend V (coin, clk, fifteen);
```

```
initial  
begin  
    fork @V.st.t1; @V.st.t2;  
          @V.st.t3; @V.st.t4;  
          @V.st.t5; @V.st.t6;  
          @V.st.t7; join  
    $finish(0);  
end  
  
always insert_coin(tossup());
```

**finish when all
transitions
have happened**

**random
stimulus**

```
function coin_type tossup();  
    switch($random() % 3)  
        case 1: return(five);  
        case 2: return(ten);  
        default: return(none);  
    endswitch  
endfunction
```

```
task insert_coin(input coin_type c);  
begin  
    coin <= c;  
    if (c == five) sum += 5;  
    if (c == ten)  sum += 10;  
    @(posedge clk) coin <= none;  
    @(posedge clk) if (sum >= 15) begin  
        if (fifteen != 1) $display("ERROR");  
        sum = 0;  
    end  
    else if (fifteen == 1)  
        $display("ERROR");  
end  
endtask
```

• Checking the Coverage



C Interoperability

SUPERLOG C Interoperability

- SUPERLOG and SYSTEMSIM includes the capability to import and export C objects and tasks/functions, in a manner that allows easy mixing of C and SUPERLOG/Verilog code. This also eliminates the use of the PLI within the simulator, making the mechanism easy to use and fast.
- This capability is not described in the tutorial but information may be obtained from Co-Design Automation.

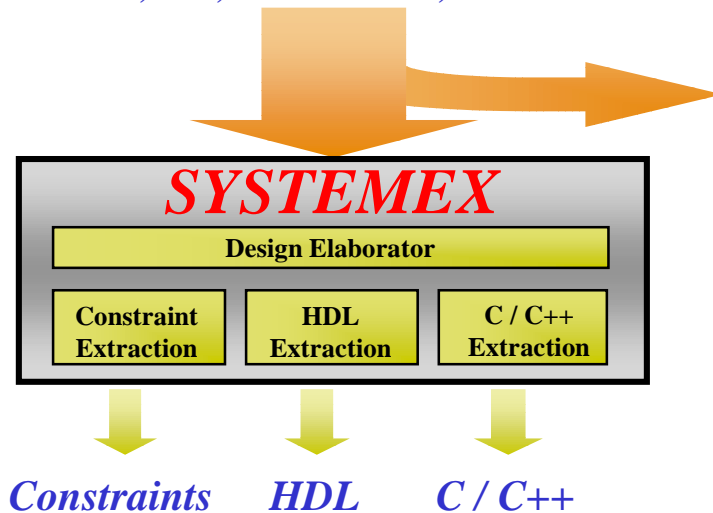


Summary

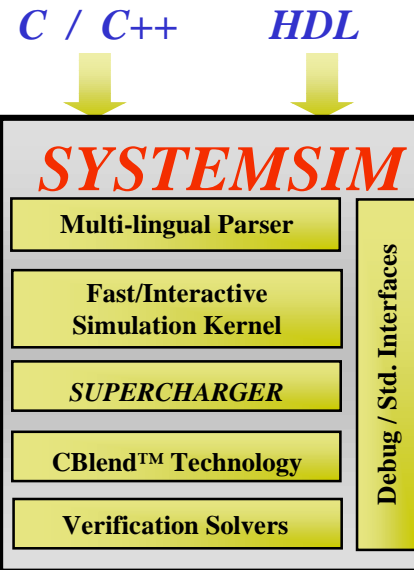
The Co-Design Automation Product Line

Streamlining Design, Accelerating Verification


HW, SW, Architecture, Verification



SYSTEMEX: Enabling SUPERLOG abstraction

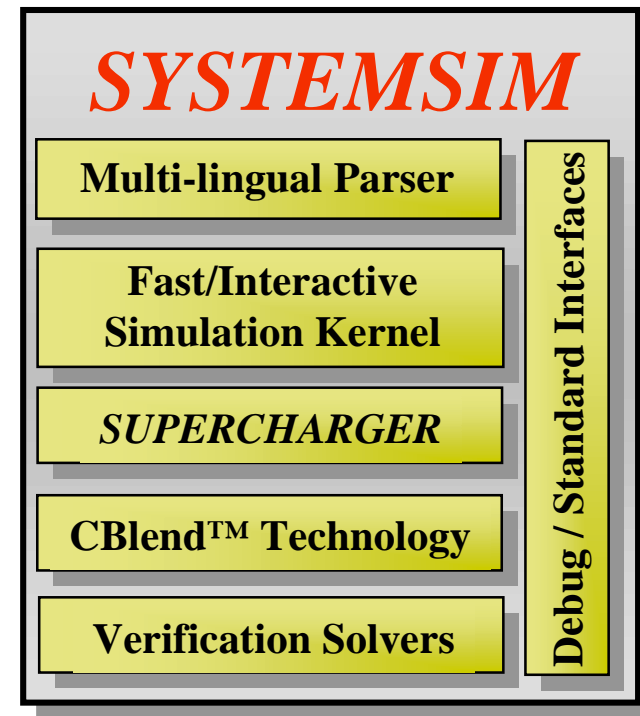


SYSTEMSIM: Performance, Flexibility, Evolution

- Methodology performance
- Multilingual SoC flexibility
- Integration of key functions

The SYSTEMSIM Unifying Simulator

- Performance AND Interactivity
 - Unique Compiled Code / Interpreted Supercharger enables fast, flexible simulation
- Integrated verification & system design
 - Allows simulation/verification/system abstraction all within single simulator
- Unique C / HDL Interleaving
 - CBlend enables no-pli C, Verilog, SUPERLOG mixing for speed and ease of use
- Fully functional Verilog drop-in capability
 - Language, PLI, debug, third party interfaces

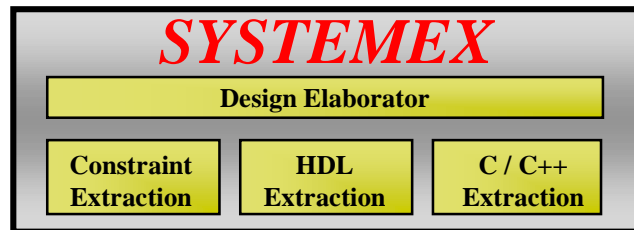


**Fast simulation with integrated verification and
seamless C/C++ interleaving, driven by SUPERLOG**

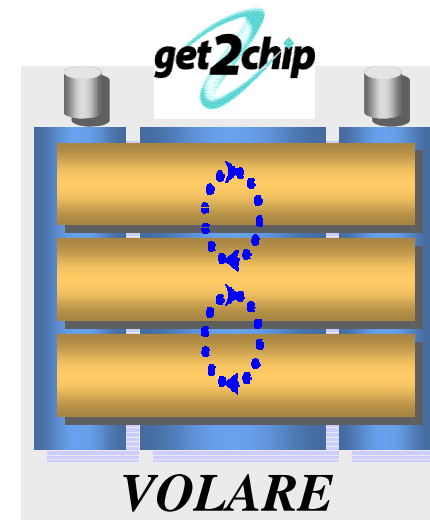
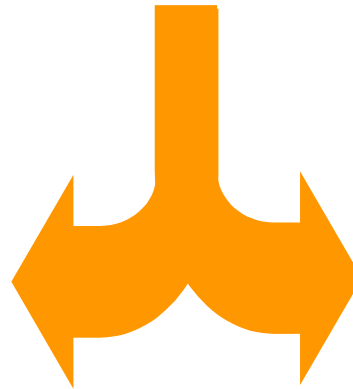
SUPERLOG Abstraction to Implementation

Fast, Abstract Code May Now Be Synthesized

 **SUPERLOG**



Synthesis Tools



**Get2Chip's Volare
Architectural Synthesizer
reads SUPERLOG directly**

Co-Design Automation At Large

Aiming to open SUPERLOG for public standardization

SUPERLOG
announced

General
industry
interest

Language
experts
interested

EDA
providers
commence
utilization

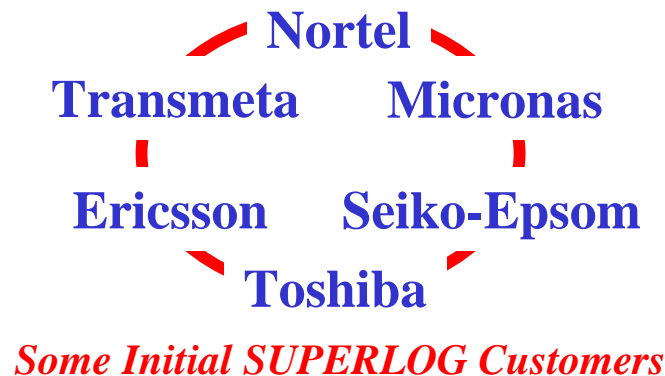
Customer
proven
flows

Industry
steering
group

Steering
group
ratification

Standards
Organization
Ratification

SUPERLOG EDA Partners

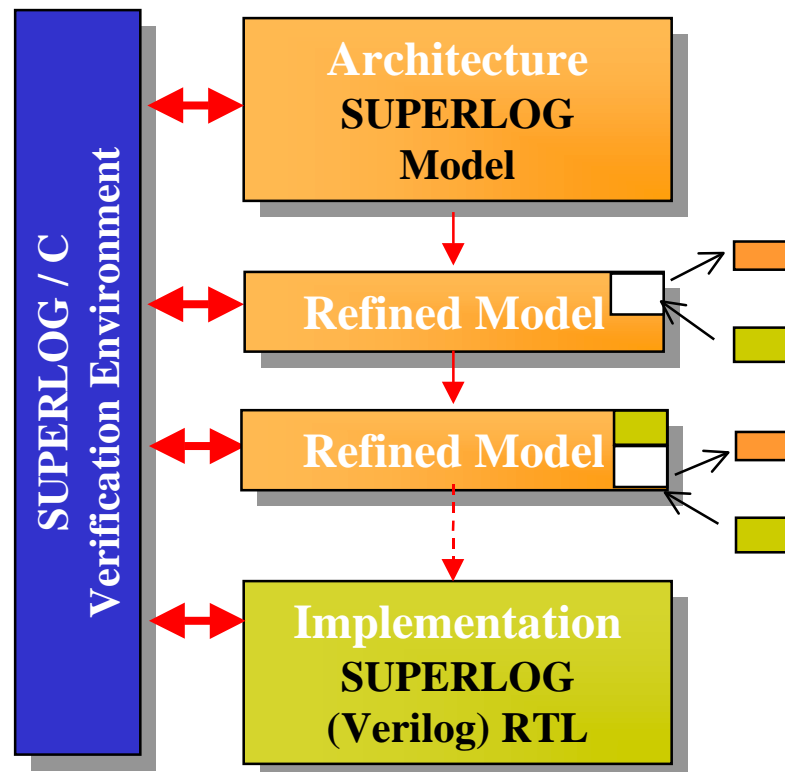


SUPERLOG Relationships



SUPERLOG In Action - Ericsson

Seamless algorithm to implementation flow + linked verification,
provides modeling accuracy and methodology productivity

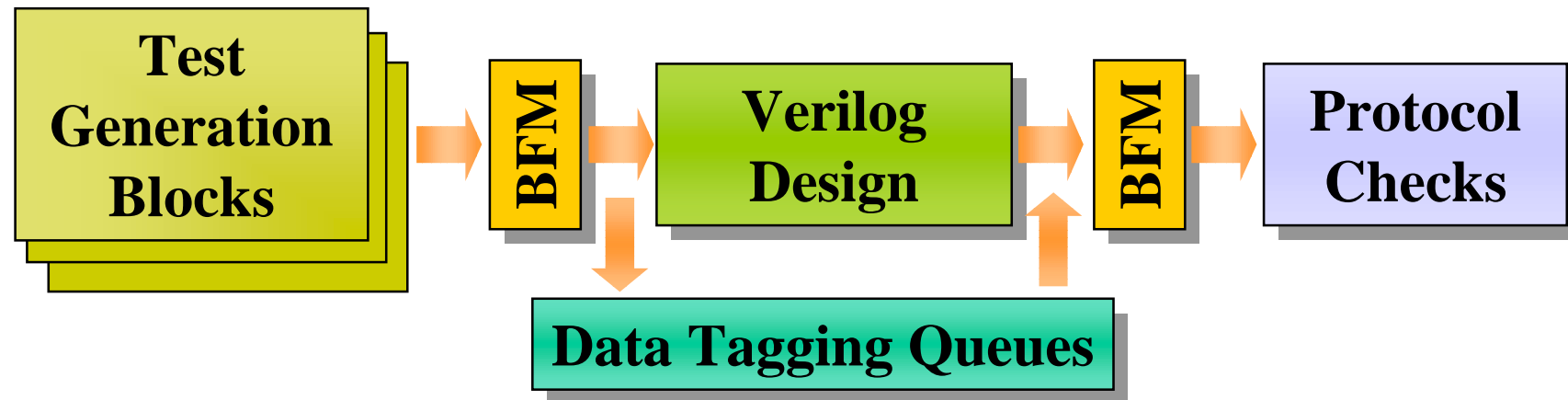


The process of Successive Refinement allowed Ericsson to develop high level algorithm models, test them quickly, and then switch in individual block implementations and ensuring minimal regressions



SUPERLOG In Action - Nortel Networks

Nortel's Networking Verification Environment
Several X faster than comparable envs., shorter learning curve



"Co-Design's SUPERLOG language enables faster design creation and effective verification, but in a practical, evolutionary manner."

**Anders Nordstrom, ASIC Development
Manager, Nortel Networks, Inc.**

**NORTEL
NETWORKS**

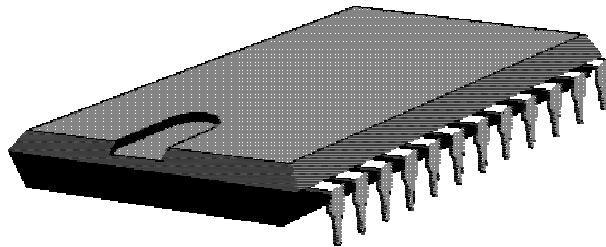
The First SUPERLOG “Tape-Out”

SUPERLOG & SYSTEMSIM proven in customer design



FreeHand Communication AB

– DSP solutions on silicon



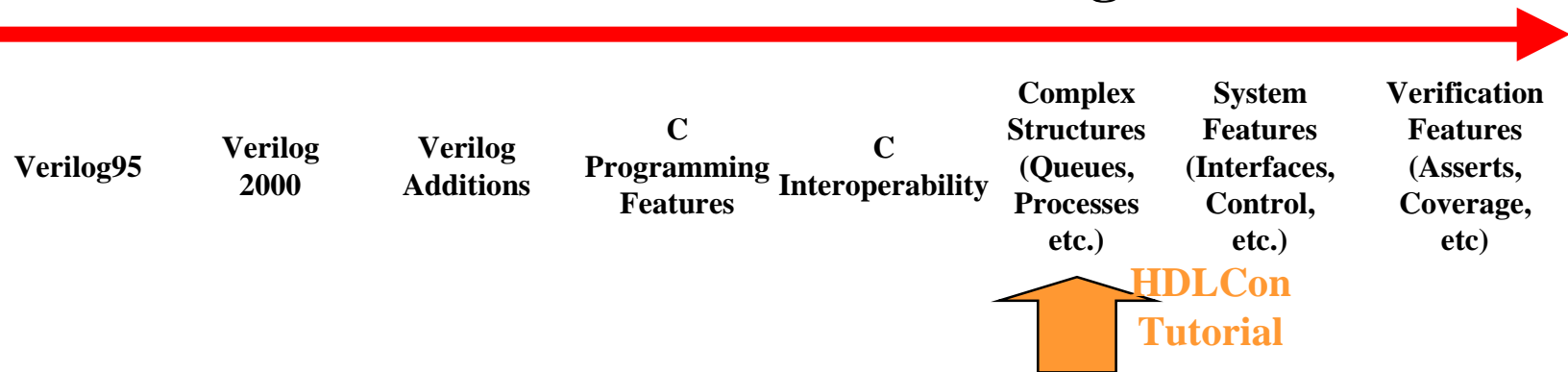
- Complex DSP device - parallel, echo cancellation algorithm
- Project used combination of C (CBlend) and SUPERLOG
- IP provided to Freehand customer who taped out product
- Device completed on time with first iteration success

“By eliminating interface overhead, SYSTEMSIM has enabled a clean and highly efficient integration of our DSP Instruction Set Simulator (ISS) model. The resulting ease of use and performance benefits will accelerate our development cycle by a significant factor.”

Harald Bergh, CEO, FreeHand Communication AB

Capabilities NOT INCLUDED In This Tutorial

SUPERLOG Publication Stages

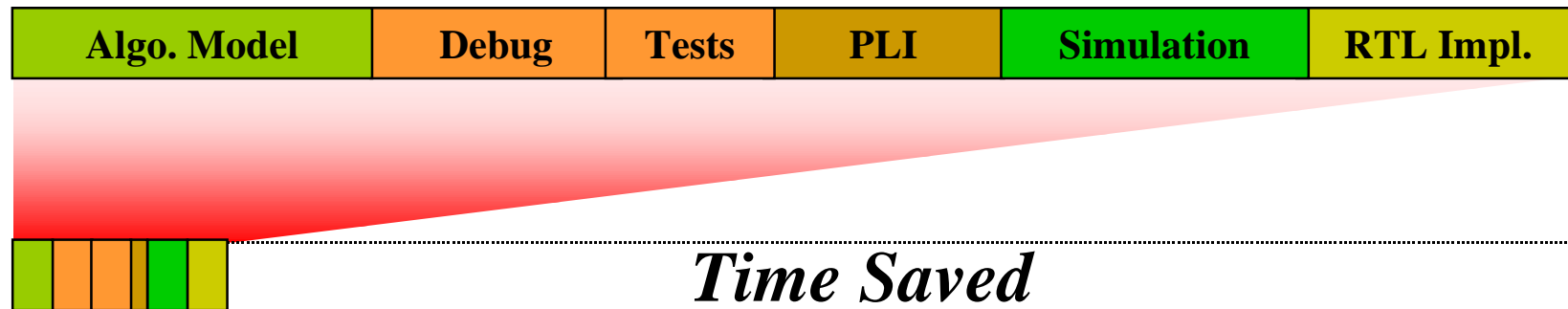


- Verification capabilities including test generation, result checking, coverage and property specification.
- Object Orientated features including the use of class
- Communication Orientated features and other interface capabilities
- Various other programming and system abstraction features
- Some Verilog 2000 features

For More Information On These Please Contact Co-Design Automation.

Order Of Magnitude Productivity Enhancements

Practical Productivity



Direct Performance

- Concise RTL implementation
- PLI elimination
- Simulation speed acceleration

Overall Productivity

- Integrated algorithm modeling
- Efficient testbench creation
- Rapid debug



More Information?

Please Contact:

Co-Design Automation, Inc.

Tel: 1 877 6 CODESIGN

Email: info@co-design.com

Web: www.co-design.com

www.superlog.org



© 2001 Co-Design Automation, Inc.
Tutorial presented at HDLcon 2/28/2001