

March 11 - 12, 2002



# Advanced Verification Utilizing the SUPERLOG Language

---

David Rich

davidr@co-design.com

Co-Design Automation, Inc.

Tom Fitzpatrick

fitz@co-design.com

Co-Design Automation, Inc.





# Agenda

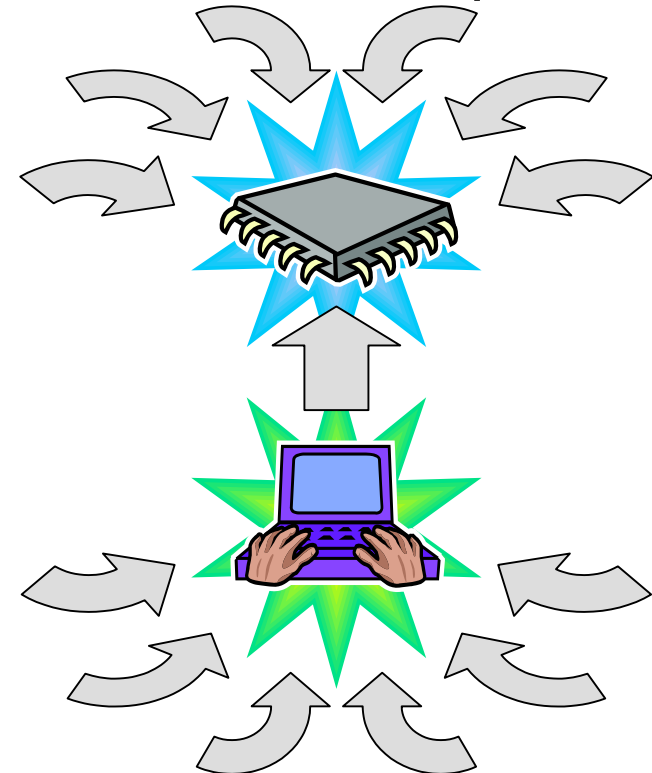
---

- Verification overview
- SUPERLOG introduction
- Useful SUPERLOG language features
- Verification example
- Functional coverage
- Using C code
- Hardware-software co-verification
- Wrap-up
- Questions

# The Reality of SoC Design

- Lots of hardware
  - Multi-million gates
  - Many different IP blocks
  - Can't possibly predict all interactions
- Embedded processor
  - Increasing percentage of design starts use embedded processors
- Performance
  - Eliminate bottlenecks
  - Be smart!

More & more complex HW

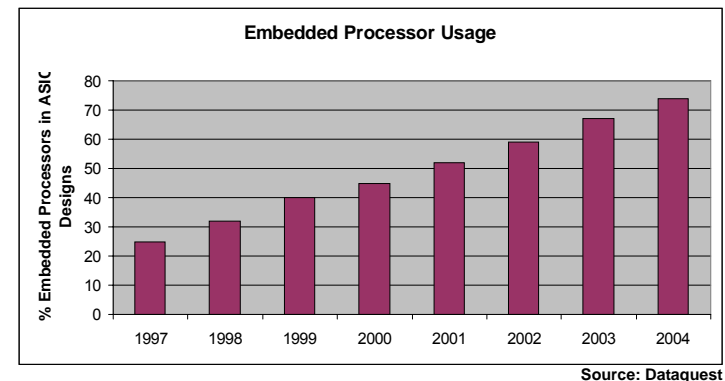
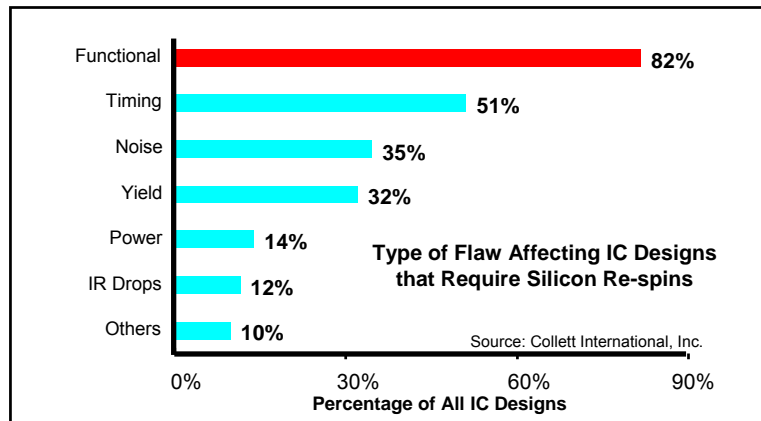


More complex Embedded  
SW Applications

# The Importance of System Verification

## Embedded System Verification Becoming Key Problem

**Over half of today's ASICs  
contain embedded processors**



**Functional bugs not found  
during verification is the  
biggest cause of chip re-spins**

**Average number of chip iterations today = 3 per project**

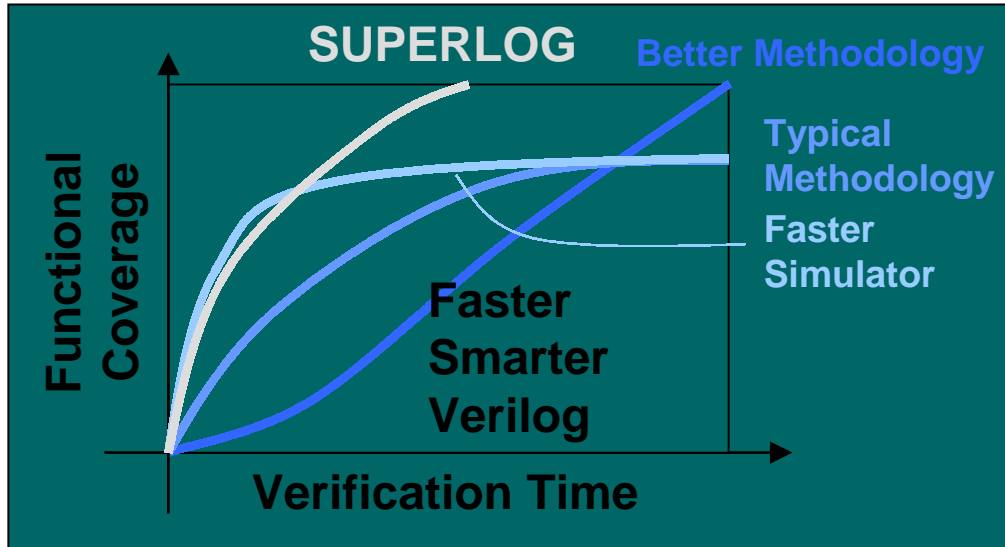


# Enhancing Verification Productivity

- Verification is all about finding bugs
- You can't find bugs if you're looking in the wrong place
  - Always start with a verification plan
  - Know how to quantify results
- The fastest simulator won't help if your test doesn't do the right thing
  - Don't waste simulation cycles on functionality already verified
- Avoid coding bugs in the first place
  - Give designers tools to write better code
  - Use designers' knowledge to enhance verification
- Be smart!
  - Think about what the real problem is
  - You may have to change the way you've been doing things

# Verification Productivity

## SUPERLOG and SYSTEMSIM Give the Functional Coverage You Need in a Faster Simulation Environment



✓ **Productivity is more than just Verilog simulation speed**

- Productivity means better coverage in less time
- Faster simulation = same coverage in less time
- Need better methodology to find more bugs
- Need a simulation environment that supports and encourages a better methodology
- Don't waste time

March 11 - 12, 2002



# SUPERLOG Overview

---



# SUPERLOG History

## The Team

Phil Moorby, **creator** Verilog; Peter Flake, **creator** HILO;  
**+ 15 simulation/language experts across EDA industry**

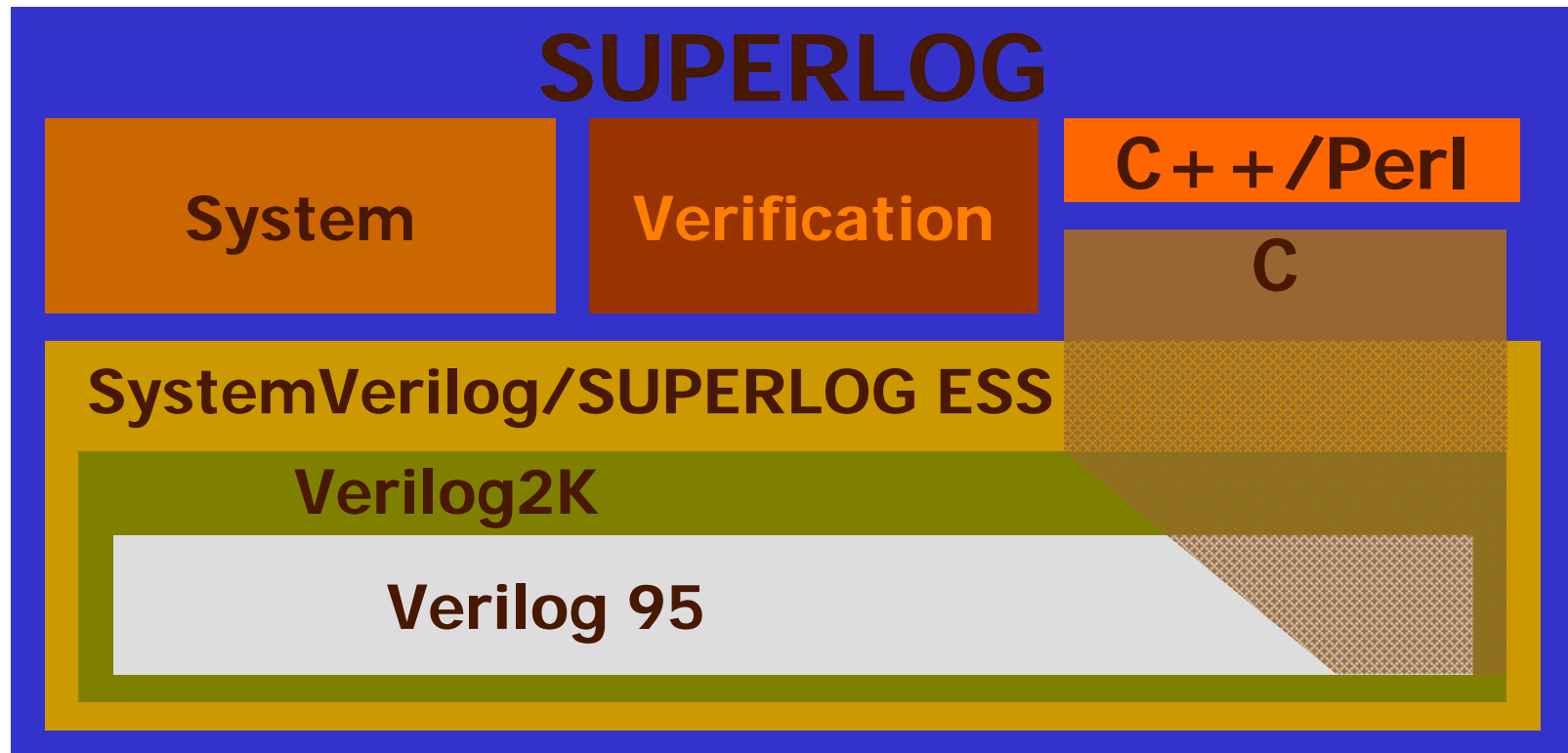
## Co-Design Automation History



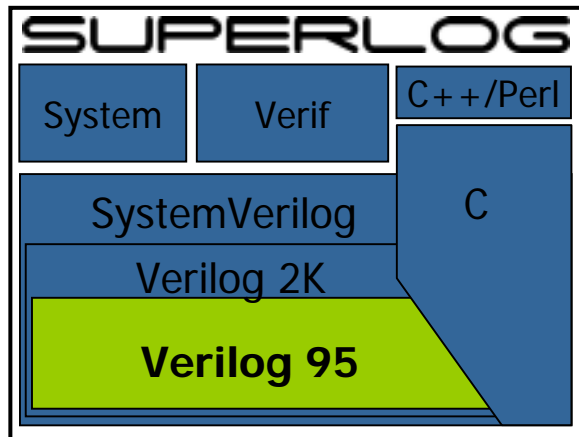


# SUPERLOG in a Nutshell

- SUPERLOG = Verilog + C + ...

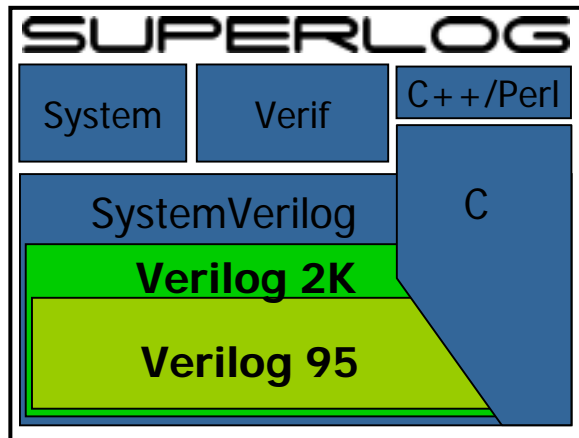


# Verilog 95



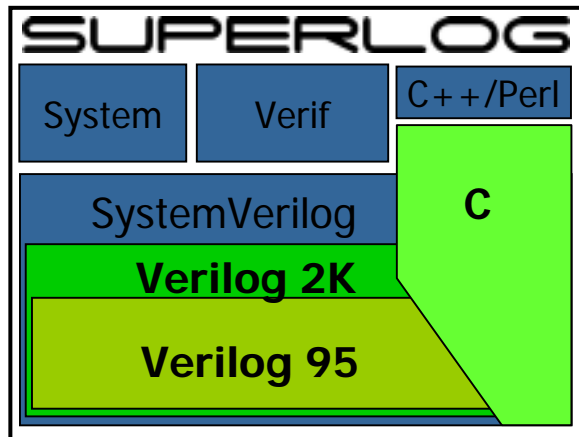
- Verilog gave us:
  - Hardware concurrency
  - Timing
  - Mixed RTL/gate/switch
  - State/strength
  - Event ordering

# Verilog 2001 Additions



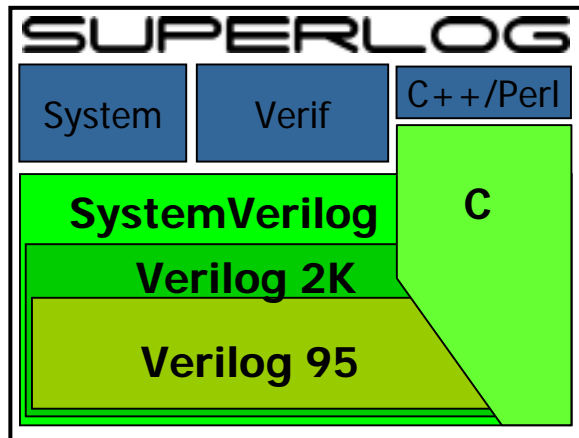
- Verilog 2001 gives us:
  - ANSI c-style port/argument lists
  - Automatic tasks/functions
  - Generate
  - Signed arithmetic
  - Configurations
  - Constant functions

# SUPERLOG Includes C Constructs



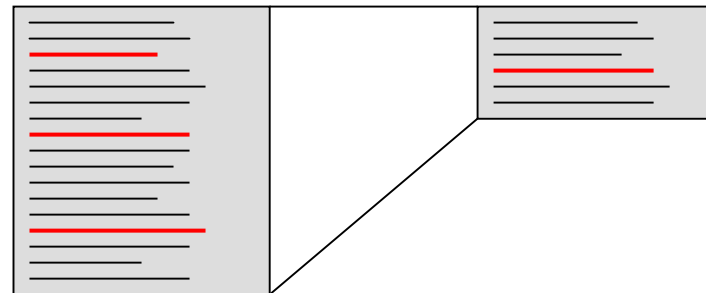
- C provides programming features and data structures
  - Structs
  - Unions
  - Pointers
  - Dynamic memory
  - Varargs

# SUPERLOG ESS → SystemVerilog

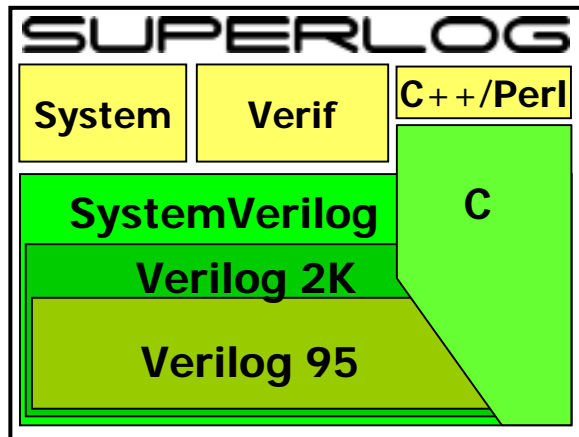


Higher Level of RTL  
Abstraction Typically  
Yields 3x Reduction  
in Code Size

- The SUPERLOG Extended Synthesizable Subset (ESS):
  - Extra Datatypes
  - Structs
  - Interfaces
  - Synthesizable Case
  - Finite State Machines
- Donated to Accellera as SystemVerilog
- More Compact Code Means Fewer Bugs



# SUPERLOG System-level and Verification Features



- More High-Level Features:
  - Multi-dimensional Arrays
  - Queues
  - Lists
  - Strings
  - Pointers
  - Design Assertions
  - Temporal Assertions
  - Random Stream Generation
  - Coverage

March 11 - 12, 2002



# Overview of Some SUPERLOG Language Features

---



# Basic SUPERLOG Datatypes

```
bit b;           // single bit 0 or 1
logic w;         // 4-valued logic, x 0 1 or z as in Verilog
byte b8;         // 8 bit signed integer
type t;          // enumeration of available types
```

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

**Extra  
SUPERLOG  
datatypes**

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



# 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 use 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

# The SUPERLOG Type

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

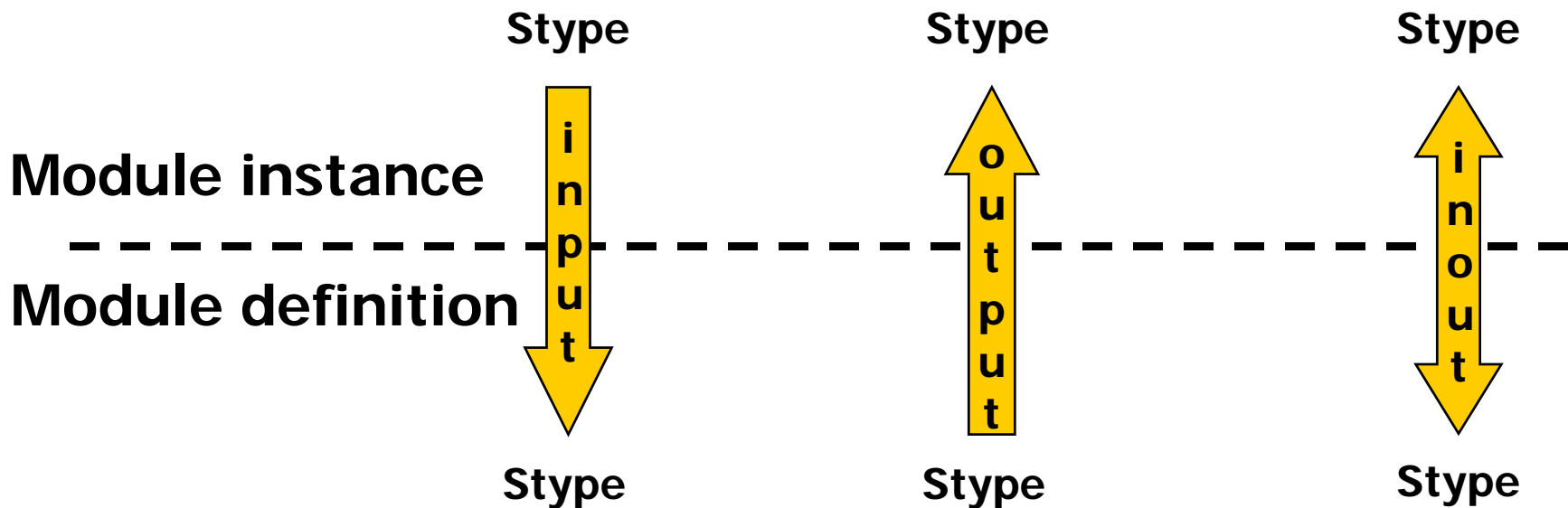
```
typedef struct {  
    real R;  
    real I;} Complex;  
Complex X,Y,Z;  
always @(negedge clk)  
    begin  
        X = Complex_F1(Z);  
        Y = Complex_F1(Z);  
    end  
always @(posedge clk)  
    X = Complex_F1(Y);  
  
assign Z = Complex_F1(X);
```

One or more  
procedural  
assignments to X,Y

Single continuous  
assignment to Z

# SUPERLOG Module Ports Connection 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



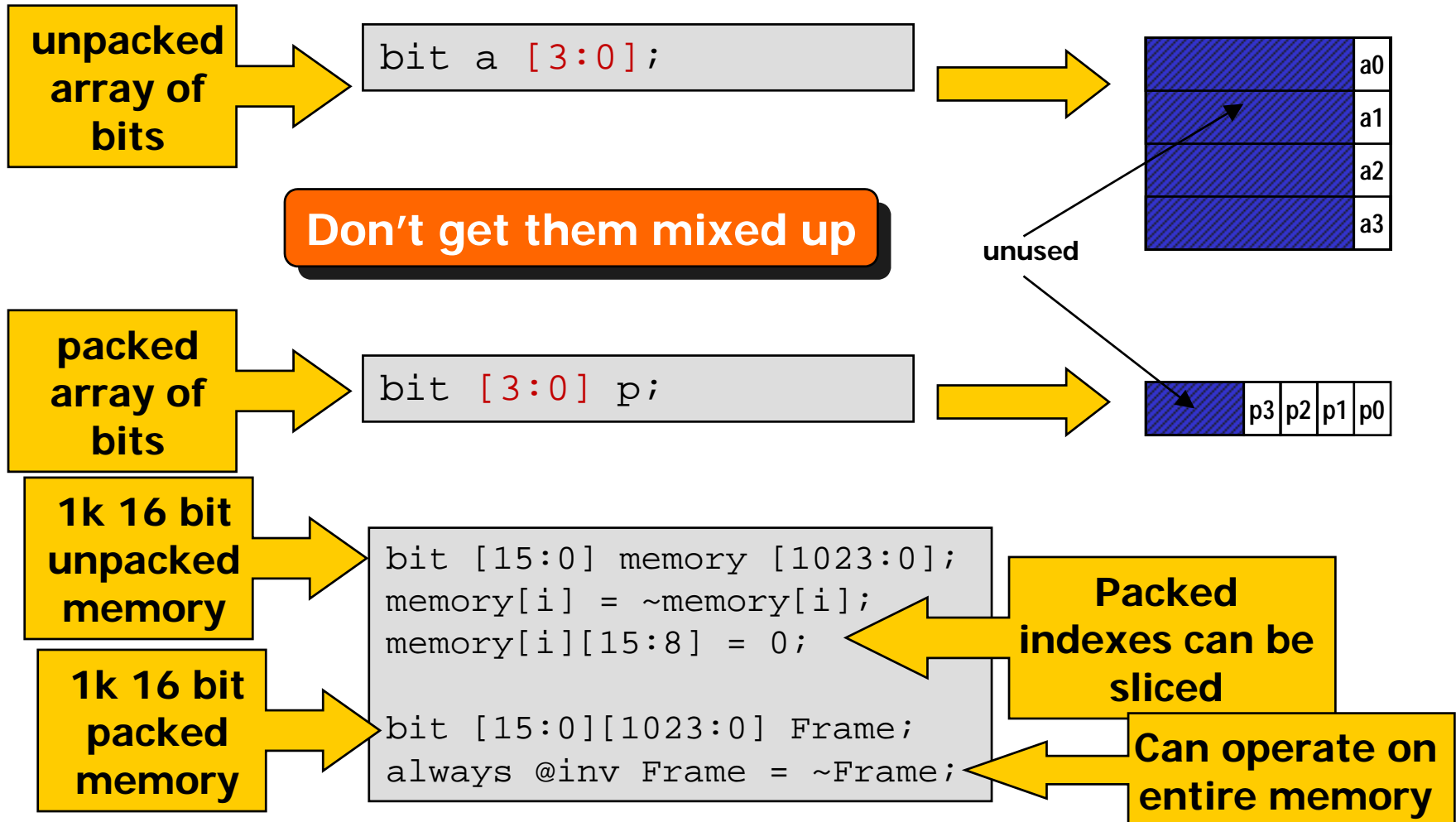


# Packed and Unpacked Arrays

---

- 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



# Packed Array Part Selects

- A part select or slice is a concatenation of bits

```
logic [4:0][3:0] var;
```

**This is a  
20 bit  
vector**

	3	2	1	0
4				
3				
2				
1				
0				

```
var[0][3:1] = 3'b000;  
{var[0][3],var[0][2],var[0][1]} = 3'b000;
```

**This is a 3 bit vector**

```
var[2:1] = 8'hFF;  
{var[2][3:0],var[1][3:0]} = 8'hFF;  
{var[2][3],var[2][2],var[2][1],var[2][0],  
var[1][3],var[1][2],var[1][1],var[1][0]} = 8'FF;
```

**This is an 8 bit vector**

# 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 [maxy:0][maxz:0] pcube[maxx:0];
```

**Unpacked indexes  
must be specified**

**Packed  
dimension  
varies more  
rapidly**

```
ucube[x][y][z] = 1'b1;
pcube[x] = 0;
pcube[x][1:0] = '1;
pcube[x][y][2:0]++;
```

**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][2:0]
= {{1.5,4.5,4.3},{5.0,0.5,2.1}};
```

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

# SUPERLOG 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 With Whole Structures

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

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

constant  
literal

```
RGB Frame[639:0][479:0];
```

array

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

copy

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

port

```
assign b = (pixin == BLUE);
```

compare

```
endmodule
```

# Unions

```
typedef union {
  int n;
  real f;
} u_type;
```

union

provide storage for  
either 'int' or 'real'

```
u_type u;
```

```
initial
```

```
begin
```

```
u.n = 27;
```

int

```
$display("n=%d", u.n);
```

```
u.f = 3.1415;
```

real

```
$display("f=%f", u.f);
```

```
$finish(0);
```

```
end
```

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);  
    printnum(num);  
  
    assign_int(num, 1024);  
    printnum(num);  
  
    $finish(0);  
end
```

```
task printnum(input number_type nm);  
    if (nm.is_real)  
        $display("num=%f (real)", nm.n.f);  
    else  
        $display("num=%d (int)", nm.n.i);  
endtask
```

```
task assign_real(output number_type nm,  
                 input real f);  
  
    begin  
        nm.is_real = 1;  
        nm.n.f = f;  
    end  
endtask
```

```
task assign_int(output number_type nm,  
                input int i);  
  
    begin  
        nm.is_real = 0;  
        nm.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
```

## Output:

```
num= -3.141500 (real)  
num=      1024 (int)
```

```
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 can be  
called with  
different  
argument types

# Queues & Lists

```
bit [7:0] myq[0:$];
```

define a list of  
8-bit logic items

```
out = myq[0];
```

access the left most  
item

```
out = myq[$];
```

access the right  
most item

```
myq = {n, myq};
```

insert n at the left  
side of the list

```
myq = {myq, n};
```

append n on the  
right

myq[x] = 

--	--	--	--	--	--	--	--

• a list is a variable  
length array

• myq[0] myq[1] ...  
myq[\$]

List manipulation  
syntax is similar to  
concatenation and  
bit select in packed  
arrays

- Concise, Simple, Powerful. Intuitive Syntax.

# More List Operations

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

Define a list 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 list

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

To step through the list use an integer index

```
qp = {};
```

Delete the whole list



# List Example

```
task qinsertafter(input int n, input int pos);
begin
  if ( (pos > q.$num) || (pos < 0) )
  begin
    $display("ERROR out of bounds");
    return;
  end

  if ( (pos + 1) == q.$num )
    q = {q,n};
  else
    q = {q[0:pos], n, q[pos+1:$]};
  end
endtask
```

append

insert

Task to insert  
n after  
position pos

some optional  
error checking

q[n,n-1] is {}  
q[0:1] : 2 items  
q[0:0]: 1 item  
q[0,-1] : 0 items

```
task qinsertafter(input int n, input int pos);
  q = {q[0:pos], n, q[pos+1:$]};
endtask
```

Same Task in  
3 lines

- List operations can be encapsulated in tasks

# Queues for Abstract FIFO Modeling

```
module fifo(
  input wire clk,
  input T in,
  output T out);
```

```
parameter type T = byte;
T q[0:$];
```

```
always @(posedge clk)
  q = {q, in};
```

```
always begin
  wait(q.$num > 0);
  repeat(2) @(posedge clk);
  out = q[0];
  q = q[1:$];
end
endmodule
```

queue of  
abstract type

push

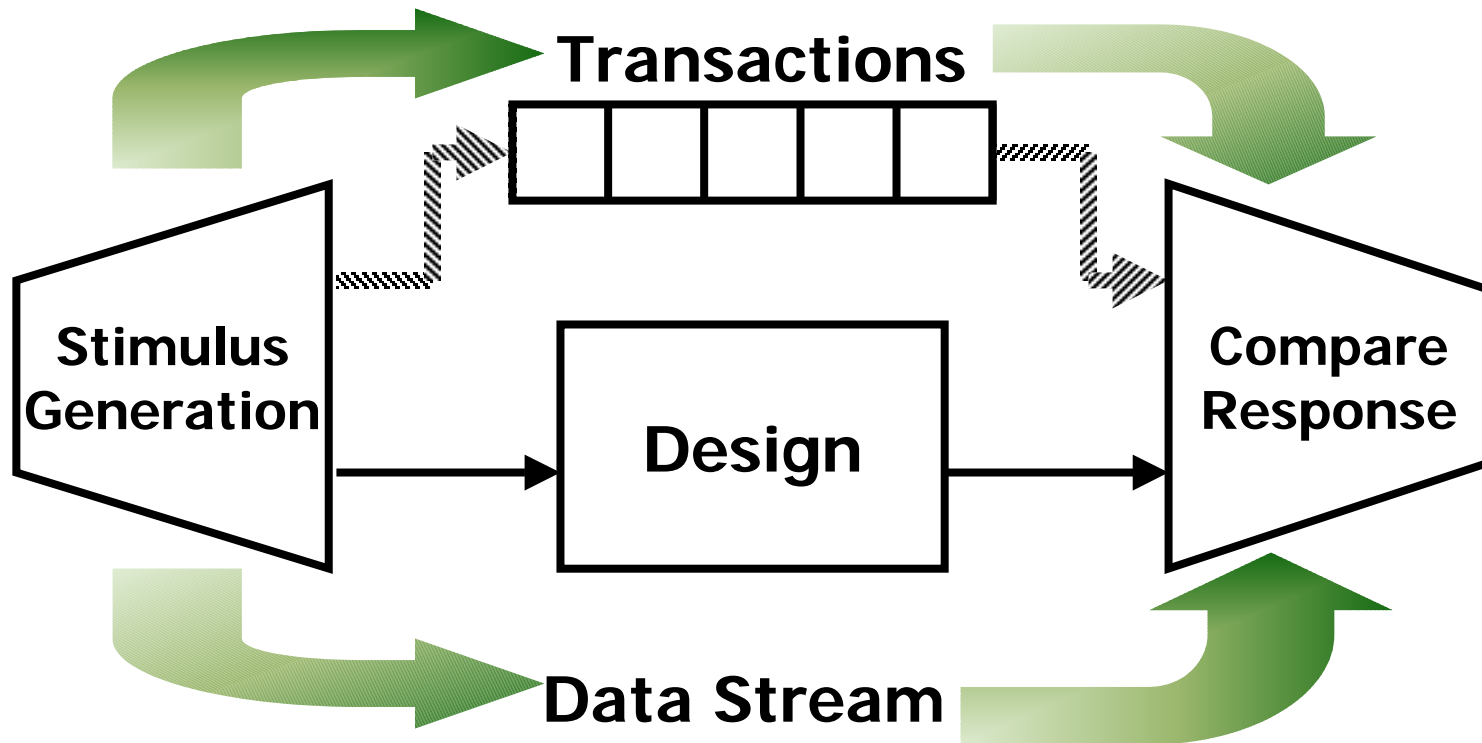
model  
delay

pop



- **Extremely common in telecoms and data processing designs**

# Queues for Verification



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

# Associative Arrays

- User-defined index type
- Memory allocated as elements are written

```
typedef enum {Red, Green, Blue} RGB;  
byte Gamma[RGB]; // Three possible bytes  
int Color[byte]; // same as int Color sparse [0:255]  
Color[(Gamma[Red]=12)] = -1;  
Gamma.$delete(Red);
```

Index specifier  
is a datatype

```
int Color[string] = {default:-1};
```

**Sparse/Associative array initialization using default value to be returned. It does NOT explicitly allocate storage.**



# 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



# 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 = i 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**

# SUPERLOG and C Pointer Differences

## 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", (deref p).value);  
$finish(0);
```

can use  
struct\_type in  
the definition



# 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; // p2->n = 27
```

p is a ref to  
data\_type

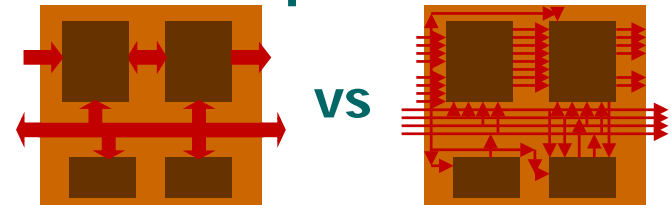
\$alloc returns a  
pointer to a  
'data\_type' item

a second ref  
p2 to the  
same location

# Interfaces: Communication-Based Verification

- Most bugs occur between blocks
- Encapsulation is key
- Capture interconnect and communication
- Separate communication from functionality
- Eliminates “wiring” errors
- Reuse interface objects
- Facilitates divide-and-conquer verification

## Encapsulation



## Interface-based Verification

Verify  
Interface  
Standalone



Focus Test on  
Functionality,  
not Interconnect



# Interface Example

```
interface utopia_i;
  wire soc;           // start of cell
  wire en;            // enable
  wire [7:0] data;     // data
  wire clav;          // cell available
  wire clk;           // ATM layer clock
endinterface
```

```
interface cpu_i(input bit rst);
  wire BusMode;
  logic [11:0] Addr;
  logic Sel;
  wire [7:0] Data;
  logic Rd_DS;
  logic Wr_RW;
  wire Rdy_Dtack;
endinterface
```

```
module squat_m(utopia_i ux, cpu_i cpu,
               input bit clk);
endmodule
```

SystemVerilog

```
module squat_m (SX_ux_soc, SX_ux_en, SX_ux_data,
SX_ux_clav, SX_ux_clk, SX_cpu_BusMode, SX_cpu_Addr,
SX_cpu_Sel, SX_cpu_Data, SX_cpu_Rd_DS, SX_cpu_Wr_RW,
SX_cpu_Rdy_Dtack, rst, clk);
  inout SX_ux_soc;
  inout SX_ux_en;
  inout [7:0] SX_ux_data;
  inout SX_ux_clav;
  inout SX_ux_clk;
  inout SX_cpu_BusMode;
  inout [11:0] SX_cpu_Addr;
  inout SX_cpu_Sel;
  inout [7:0] SX_cpu_Data;
  inout SX_cpu_Rd_DS;
  inout SX_cpu_Wr_RW;
  inout SX_cpu_Rdy_Dtack;
  input rst;
  input clk;
  wire SX_ux_soc;
  wire SX_ux_en;
  wire [7:0] SX_ux_data;
  wire SX_ux_clav;
  wire SX_ux_clk;
  wire SX_cpu_BusMode;
  wire [11:0] SX_cpu_Addr;
  wire SX_cpu_Sel;
  wire [7:0] SX_cpu_Data;
  wire SX_cpu_Rd_DS;
  wire SX_cpu_Wr_RW;
  wire SX_cpu_Rdy_Dtack;
  wire rst;
  wire clk;
endmodule
```

Verilog95

3x or more compaction  
Less chance of wiring  
mistakes

March 11 - 12, 2002

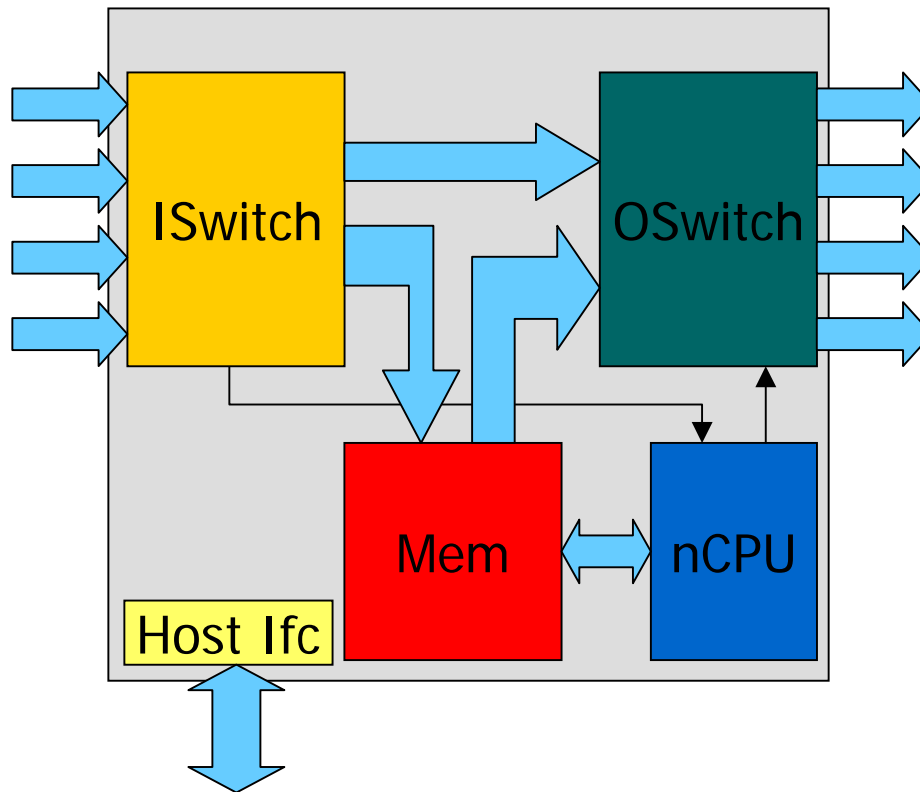


# Methodology Example

---



# Example Circuit



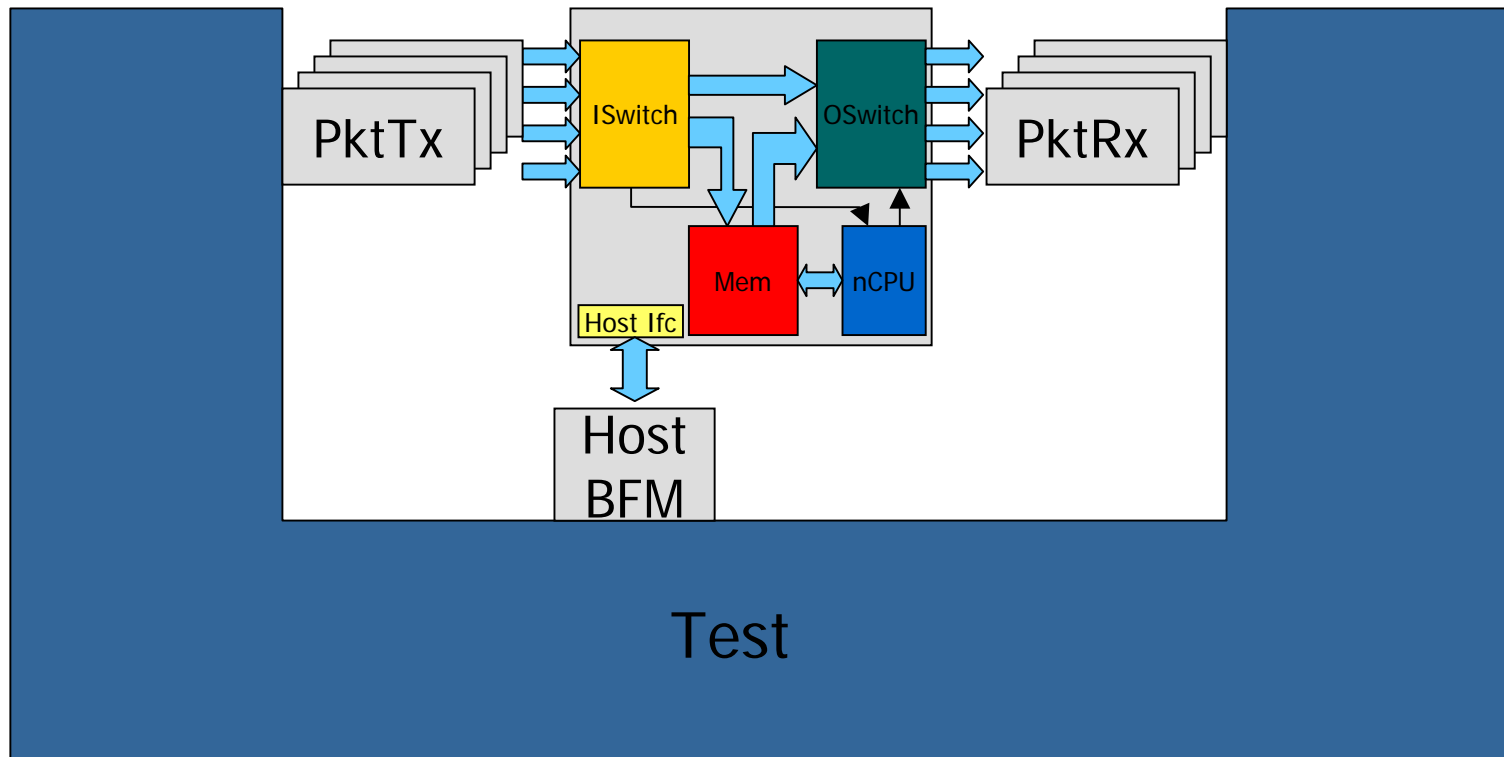
- Network switch/processor
- Packets come in on 4 input channels
- Packets route through to OSwitch or to Mem as CPU programs/data
- nCPU executes programs
  - Modifies packets for output
  - Executes algorithms
- OSwitch outputs packets on one of 4 channels
- Host interface to configure
  - Write registers
  - Write program directly into memory

# Verification Plan

- Identify interfaces
  - Packet input channel (4)
  - Packet output channel (4)
  - Host interface
- Host interface
  - Read/write registers
  - Read/write memory
  - DMA burst to/from memory
- Packet input
  - Routing packet format
  - Program packet format
  - Packet receive protocol
  - Source/Dest port tagging
- Packet output
  - Packet transmit protocol
  - Verify expected contents
- nCPU
  - Verify all instructions
  - Verify all addressing modes
  - Track combinations

# Building the Infrastructure

## TestBench Setup





# HDL Skeleton

```
module tb;
bit clk, rst;
host_i host(clk,rst);
PktTx_i tx[0:3](clk,rst);
PktRx_i rx[0:3](clk,rst);

dut dut(tx, rx, host);
...
endmodule

interface PktTx_i
    (input bit clk, rst);
<signal_declarations>;
<tasks/function methods>;
<processes>;
endinterface
```

```
interface PktRx_i
    (input bit clk, rst);
...
endinterface

interface host_i
    (input bit clk, rst);
...
endinterface

module dut(PktTx_i rx[0:3],
          PktRx_i tx[0:3],
          cpu_i host);
...
endmodule
```





# Verifying the Host Interface

- What are the operational modes?
  - Mode0: single word read/ write to device registers
    - Iswitch
    - Oswitch
    - Cpu
  - Mode1: read/write to memory
    - Single word access
    - Multi-word DMA bursts
- Gives at least 10 scenarios to test
- What are the tasks to be written?
  - Single word read/write
  - DMA burst read/write
- Test implementation choices
  - Explicit tasks in the TestBench
    - Difficult to share between tests
    - Separate from BFM
  - Encapsulate tasks in bus-functional model
    - Tasks and signals live together
    - Easily re-used on other projects
    - Interface lets you encapsulate other useful test features
      - Protocol checking
      - Coverage



# Host BFM

- Responsible for reading/writing device registers and memory
  - Mode bit 0 = reg, 1=mem
  - DMA read/write to/from mem
- Read/write methods called from test
  - Strung together for more complex tests
- Protocol checking to ensure that DUT adheres to protocol
- Coverage checking to report that all modes are covered

```
interface host_i(input bit clk);  
    logic[11:0] addr;  
    logic[31:0] data;  
    bit rd,wr,mode;  
    bit rdy;  
  
    // Methods  
    task read(...);  
    endtask  
    task write(...);  
    endtask  
    ...  
    // Protocol Checks  
    ...  
    // Coverage Monitors  
    ...  
endinterface
```

# Host BFM Interface Methods

```
interface host_i(input bit rst,clk);
    logic[11:0] addr;
    wire [31:0] data;
    logic rd, wr, mode;
    wire rdy;

    always @(posedge clk iff !rst) begin
        data = 'bz; rd = 1; wr = 1; end

    task write(input logic [11:0] A, bit m,
              input logic [7:0] D);
        @(posedge clk)
        ...
    endtask

    task read(input logic[11:0] A, bit m,
             output logic [7:0] D,);
        ...
    endtask
endinterface
```

Interfaces Let You  
Include Self Activating  
Functionality As Well  
As Methods

```
module tb;
    reg rst,clk;
    logic[7:0]rdata;

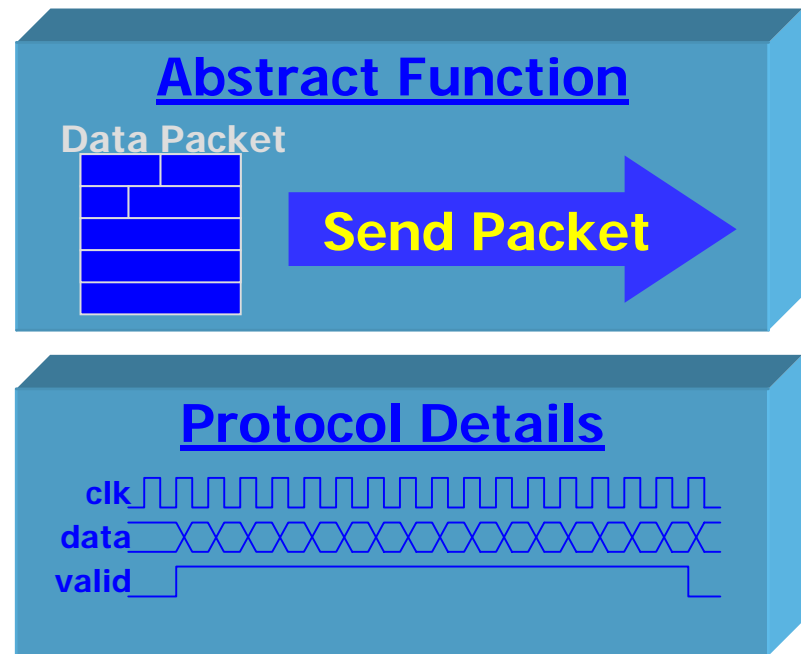
    host_i host(rst,clk);
    dut dut(rst,clk,host);

    initial begin
        do_reset;
        host.write(12'hA5E,32'h12);
        host.read(12'hA5E,rdata);
    end
endmodule
```

Methods  
Called from  
Test

# Abstract Functional Tests

- Abstraction → reuse
  - Test decoupled from design details
  - Same test can be used on similar designs
  - Keeping methods in interface helps maintain consistency
- Complex data types encapsulate information
  - Simplifies test writing
- Tests focus on *what* the DUT does
- Encapsulate test as a task that calls other tasks



# Directed Test for Host Interface

- Make Sure Each Device Can Be Written to and Read Back From
  - Call host.write followed by host.read
  - Build self-checking into the read method
    - Simplifies test since checking happens automatically
- Create Additional Tests to Access Devices in Different Order
  - What if write to OSwitch overwrites ISwitch
  - Need to interweave reads and writes

```
module tb;
  reg rst,clk;
  logic[7:0]rdata;

  host_i host(rst,clk);
  dut dut(rst,clk,host);

  initial begin
    do_reset;
    host.write(12'hA5E,32'h12);
    host.exp_read(12'hA5E,
                  rdata,
                  32'h12);

    end
  endmodule
```

# Directed vs. Random Testing

- Directed tests exercise a specific scenario
  - You direct the test
  - You explicitly orchestrate the interactions
  - It's a random world. What if you miss something?
- Injecting randomness exposes corner cases
  - Basic step: call directed tests in random order
    - Can't assume everything happens out of reset
  - Constraining random data ensures valid stimulus
  - Monitoring coverage avoids wasted simulation cycles

✓ Let the Tool Find Cases  
You Haven't Thought Of

# Managing Randomness in SUPERLOG

```
bit mode;
int a,b;
logic[11:0] addr;
logic[31:0] data,xdata;

function logic[11:0] genaddr(input bit
logic[11:0] a;
if(mode == 0)
    a = $rand(.ranges({{12'h0,12'h0FF},
                        {12'h200,12'h2FF},{12'h400,12'h4FF}}));
else
    a = $rand(.ranges({{12'h0,12'hFFF}}));
return(a);
endfunction

initial begin
    a = 1; b = 2;
    repeat(100)
        casew()
            a: mode = 0; // register access
            b: mode = 1; // memory access
        endcase
        host.write(addr = genaddr(mode),xdata = $rand());
        host.exp_read(addr,data,xdata);
    end
end
```

Each Block  
Has its own  
addr range

Branch b twice  
as likely as a

- Constraining \$rand
  - Choose from values or ranges
  - Specify weights for each
  - All parameters can be variables
  - Can choose from an array
- Constraining casew
  - Branch tags can be variables
  - Number of branches fixed at compile-time



# Mixing Reads and Writes

---

- Must make sure that address has been written before reading it
  - Build up list of possible read addresses as they are written
    - Keep track of expected values
  - Randomly choose new transaction
    - Write to random address
    - Read from previously written addr
      - Compare read data to expected data
- Must do some number of writes before doing reads



# Sample Random Test

```
typedef logic [11:0] addr_w
addr_w wraddr;
logic[31:0] rdval sparse[0:'hFFFF];
logic [31:0] rdata,expdata;
```

```
int w = 10, r = 0, cnt = 0;
while(!done)
  casew()
    w: begin:writes
      wraddr = genaddr(0);
      rdata = $rand();
      host.write(wraddr,rdata);
      rdval[wraddr] = rdata;
      if(cnt == 50)
        r = 10;
      else if(cnt == 100)
        w = 0;
      cnt++;
    end:writes
```

Initially, only  
do writes

Keep Track of  
expected value

```
r: begin:reads
  addr_w addr_l[0:$] = rdval.$keys;
  addr_w addr=$rand(.values(addr_l));
  host.exp_read(addr,rdata,
                rdval[addr]);
  rdval.$delete(addr);
  if(rdval.$num == 0)
    done = 1;
end:reads
endcase
```

List of addresses  
already written

Pick One

End Test when all  
addresses read

# Random Test Issues

- Keeping track of expected values
  - rdval[wraddr] array correlates expected value/addr pairs
  - Sparse array works like associative array
- Choosing read address
  - Could put addresses in list to read in same order as they were written
    - `addr_w rdaddr[0:$]`  
`rdaddr= {raddr,wraddr};`
  - Could put addr/data in a structure list
- Add “knobs” to the test
  - Randomize address ranges
    - Change number of ranges
    - Change weights of each range
    - Pass list as input to method
- Expand casew statement to include burst read/write when memory address range is selected

# Building the Infrastructure

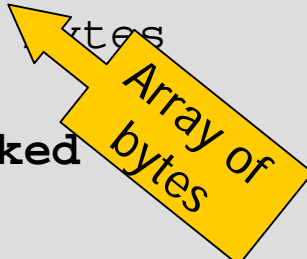
## Packet Datatypes

```
typedef struct packed {  
    logic [7:0] src;  
    logic [7:0] dest;  
    bit    rt_prg;  
    logic [6:0] length;  
    logic [7:0] aux;  
    logic [0:127][7:0] data;  
} router_pkt; // 132 bytes
```

```
typedef struct packed {  
    logic [7:0] src;  
    logic [7:0] addr;  
    bit    rt_prg;  
    logic [6:0] length;  
    logic [7:0] aux;  
    iword [0:31] instr;  
} program_pkt; // 132 bytes
```

```
typedef union packed {  
    router_pkt rpkt;  
    program_pkt ppkt;  
    logic[0:131][7:0] apkt;  
} packet_t; // 132 bytes
```

```
typedef struct packed  
...  
} iword;
```



Array of  
bytes

# Building the Infrastructure

## Packet Datatypes – Alternate Layout

```
typedef struct packed {  
    logic [7:0] src;  
    logic [7:0] dest;  
    bit    rt_prg;  
    logic [6:0] length;  
    logic [7:0] aux;  
} pkt_header; // 4 bytes
```

```
typedef logic[127:0][7:0] dfield;
```

```
typedef struct packed {  
    ...  
} word; // 32-bit instruction
```

```
typedef struct packed {  
    pkt_header head;  
    union packed {  
        dfield data;  
        iword [0:31] instr;  
    };  
} packet;
```

Anonymous


```
typedef union packed {  
    packet pkt;  
    logic[0:131][7:0] apkt;  
} packet_t; // 132 bytes
```

Array of  
bytes

# PktTx Bus-Functional Model

- Signals
  - Channel request
  - Channel available
  - Data (8 bits)
  - Data clock
  - Start of cell
- What does it do?
  - Send packet
- Add to your own library
- Reuse

```
interface pktTx_i(input bit clk);  
  bit soc, ch_avail, ch_req;  
  logic[7:0] data;  
  int i = 0;  
  
  task send_pkt(packet_t pkt);  
    @(posedge clk) ch_req <= 1;  
    @(posedge clk iff ch_avail)  
      soc <= 1;  
    for(i=0; i<pkt.length; i++)  
      @(posedge clk)  
        data <= pkt.apkt[i];  
  endtask  
  
  task rcv_pkt(...);  
    ...  
  endtask  
  
endinterface
```



Shift packet  
byte-by-byte

# Verifying the ISwitch

- Generate the packet
  - Randomize packet fields
  - Randomize length field
  - Fill packet
- Configure ISwitch via host port
  - Gain access to host port
  - Write data
- Transmit packet via interface protocol
  - Give packet to BFM
  - BFM to transmit to DUT
- Queue expected output
  - OSwitch to expect packet
  - Host to read packet in memory
- Monitor packet going in and coming out
- Compare results against expected

# Generating a Packet

```
typedef struct packed {
  logic [7:0] src = $rand(...);
  logic [7:0] dest = $rand(...);
  bit      rt_prg = $rand(...);
  logic [6:0] length = $rand();
  logic [7:0] aux = func(src,dest);
  logic [0:127][7:0] data = $rand();
} router_pkt; // max 132 bytes
```

Automatically  
randomize  
individual fields

Initialization can  
call functions

```
router_pkt mypkt;
```

Automatically initialized  
when declared

```
mypkt.$renew();
```

Explicitly re-initialized  
when called

# Transmitting a Packet

```
router_pkt mypkt;
task xmt_pkt
    (input router_pkt mypkt = router_pkt.$value);
```

Default value for  
optional argument

```
host.config(mypkt.rpkt.rt_prg);
```

Configure Host Port based on  
packet type

```
if(mypkt.rpkt.rt_prg)
```

```
    rx.expect(mypkt);
```

If router packet, expect packet on  
Rx interface

```
else
```

```
    host.expect(mypkt.ppkt.addr);
```

Else, expect packet in memory  
via host interface

```
tx.send(mypkt);
```

Send the packet

```
endtask
```

```
xmt_pkt();
```

```
xmt_pkt(this_pkt);
```



# Managing Shared Resources and Concurrency

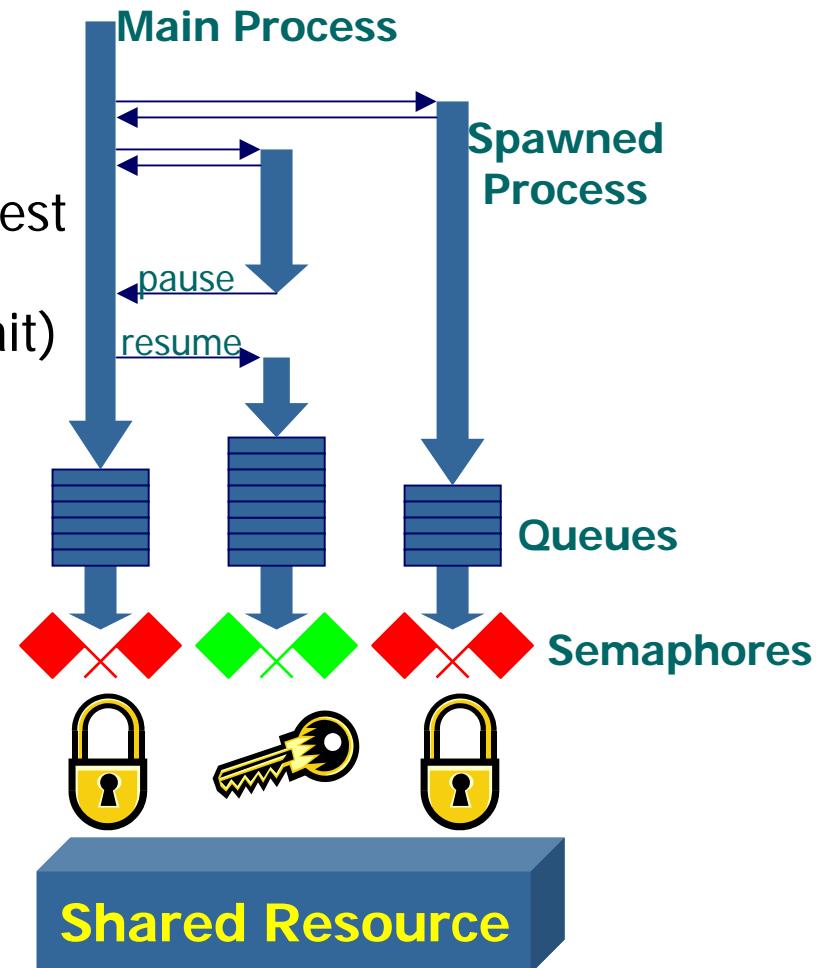
- All Config operations use the host port
- Multiple parallel tests must share
  - If host port is doing a transaction, test must wait
  - Individual test should block (i.E. Wait) until port is available

- **Spawn processes**

- "Non-blocking" *process* statement
- Pause and resume
- Built-in process ids

- **Semaphores/mailboxes**

- Synchronize processes
- Coordinate shared resources



# Using Semaphores

```
task config(input bit mode,
            ...);
host.lock;
host.write(...);
host.write(...);
...
host.unlock;
endtask

initial begin
    process config(...);
    process config(...);
end
```

Request access to Host port  
Suspend if busy

Release the host port

Spawn multiple concurrent threads  
*process* is nonblocking

- Semaphore example can be expanded to include mailboxes and regions
  - Can be provided as standard source code library
- Atomic execution is guaranteed

# Semaphore Example

```
enum {available, inuse}
  resource_status = available;
$ref_process_inst q[0:$];
$ref_process_inst next;
```

Built-in Process Management datatype

```
task lock ();
  if (resource_status == inuse)
  begin
    q = {q, $this_process};
    $pause;
  end
  resource_status = inuse;
endtask
```

If resource is busy

Add this process to the list

Pause this process

Mark resource busy and return

```
task unlock ();
  resource_status = available;
  if (q.$num > 0)
  begin
    next = q[0];
    q = q[1:$];
    $resume(next);
  end
endtask
```

Free-up resource

If there are suspended processes

Pop next waiting process from list

Resume waiting process

# Encapsulation: Dealing With Packets

- Protocol-specific encapsulation
  - Methods depend on interface protocol
    - Don't necessarily care about what's in the packet
  - Packet communication
    - Check that protocol is being adhered to
    - Ensure graceful handling of protocol errors
    - Gather statistics about what happened
  - An **interface** encapsulates inter-block communication protocols
- Data-specific encapsulation
  - Methods depend on packet type
  - Packet definition
  - Packet data generation and manipulation
    - Inserting errors
    - Generating bogus packets
  - This is what a **class** does

variable → class  
wire → interface

# Interfaces versus Classes

## ■ Interfaces

- Static objects in the hierarchy
  - Simplify interconnect
  - Model BFM's
  - Static data objects
    - Semaphores, etc.
  - Ports allow signals to be shared
  - Cannot be copied, allocated, etc.
- Can include:
  - Data types, including classes
  - Wires
  - Task/function methods
  - Processes
    - Protocol monitors, etc.

## ■ Classes

- Dynamic data objects
  - Encapsulate content
  - Cannot share signals
    - Not suitable for BFM's
  - Can be copied, allocated, etc.
  - Can inherit types/methods
- Can include
  - Data types
  - Task/function methods
- Cannot include:
  - Self activating processes
  - Wires



# Use Classes for Packets

```
class packet_c;  
    packet_t pkt;  
  
    task gen_pkt;  
        pkt = ...;  
    endtask  
  
    task insert_error;  
        pkt = ...;  
    endtask  
endclass  
  
module tb;  
    packet_c mypkt;  
  
    initial begin  
        mypkt.gen_pkt;  
        mypkt.insert_error;  
    end  
endmodule
```

# Encapsulate Tests in Classes

```
class test1;
task setup;
    ...
endtask
task run;
    host.config(a);
    host.write(...);
    tx.send(...);
    ...
endtask
endclass

module tb;
test1 test;

initial begin
    test.setup;
    test.run;
end
endmodule
```

```
class test2 extends test1;

task run;
    host.config(b);
    host.write(...);
    host.dma(...);
    ...
endtask
endclass

module tb;
test2 test;

initial begin
    test.setup;
    test.run;
end
endmodule
```

test2 class inherits  
setup method  
from test1

This is all that needs  
to change to get a  
new test

# Pointers Improve Efficiency

```
module tb;
  packet_c mypkt;
  cpu_i host(clk,rst);
  PktTx_i tx(clk,rst);
  PktRx_i rx(clk,rst);

  initial begin
    mypkt.gen_pkt;
    host.config(...);
    rx.expect(mypkt);
    tx.send(mypkt);
  end
endmodule
```

Create the  
packet

Init the  
packet

one copy

second copy

```
module tb;
  packet_c mypkt;
  ref packet_c myptr
    = ref mypkt;

  cpu_i host(clk,rst);
  PktTx_i tx(clk,rst);
  PktRx_i rx(clk,rst);

  initial begin
    mypkt.gen_pkt;
    host.config(...);
    rx.expect(myptr);
    tx.send(myptr);
  end
endmodule
```

Pointer to  
packet

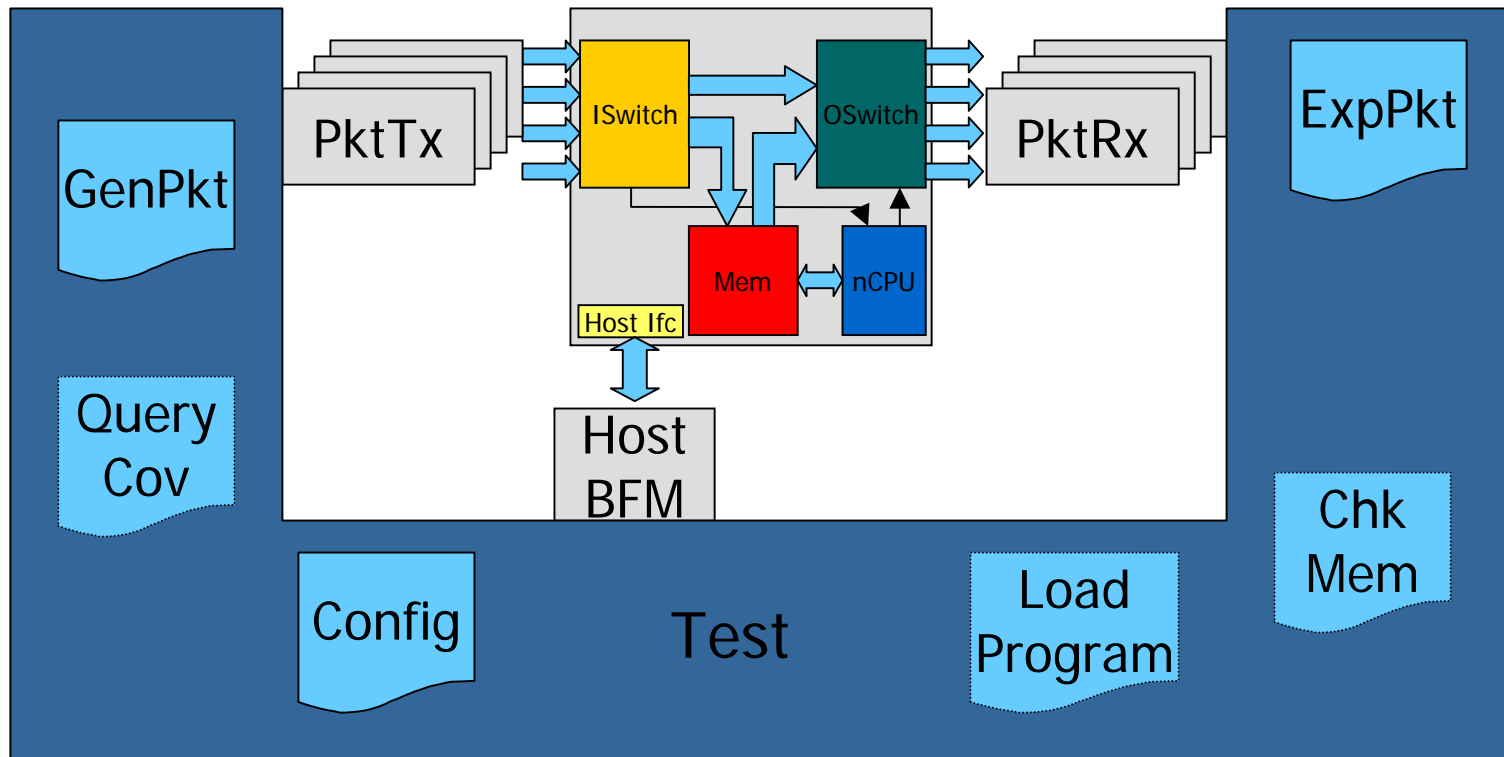
Pass the  
Pointer

No Copying  
of Packets



# Building the Infrastructure

## TestBench Setup



# Polymorphic Interfaces

- The interface is just transmitting data
- It doesn't care what the data is
- Use dynamic type for
  - Datatype flexibility
  - Automatic memory management

```
interface PktTx_i
    (input bit clk,rst);
    task send
        (input dynamic pkt);
        bytesend(pkt);
    endtask
    ...
endinterface
```

```
module tb;
    bit clk,rst;
    PktTx_i tx1,tx2;
    router_pkt pkt1;
    prog_pkt pkt2;
```

```
    initial begin
        tx1.send(pkt1);
        tx2.send(pkt2);
    end
endmodule
```

Call same  
interface method

Send protocol  
independent of  
pkt type

Send Router  
Packet

Send Program  
Packet

# Interfaces Handle Abstract Models

```
interface PktTx_i
  (input bit clk,rst);

  task send(input dynamic p);
    <xmit pkt byte-at-a-time>;
  endtask
endinterface
```

Use Detailed  
Interface

```
module tb;
  PktTx_i tx;
  dut dut(tx,...);
```

```
  initial begin
    ...
    tx.send(pkt);
  end
endmodule
```

```
interface PktTx_Beh_i
  (input bit clk,rst);

  task send(input dynamic p);
    <xmit pkt all-at-once>;
  endtask
endinterface
```

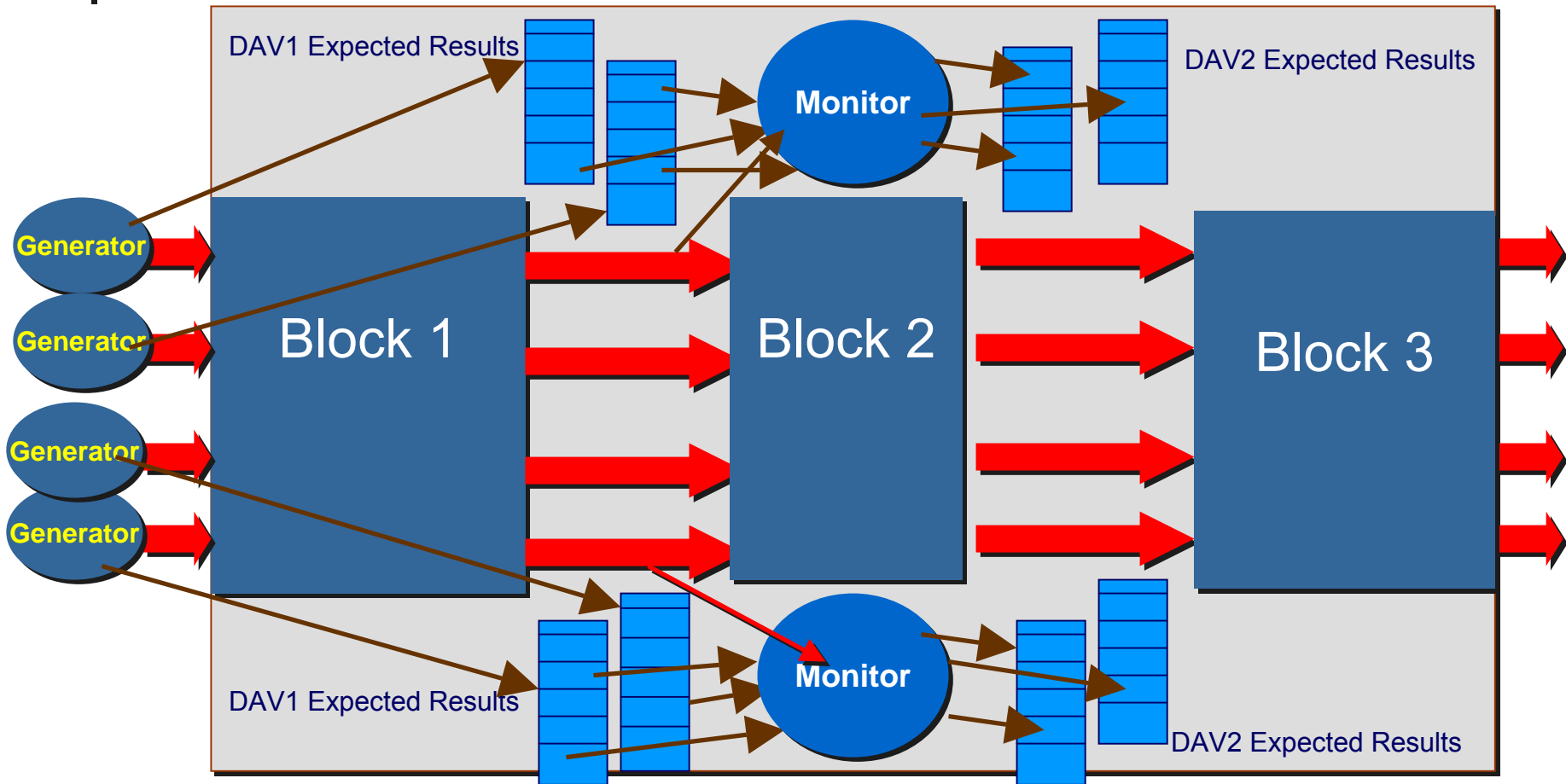
Use Abstract  
Interface

```
module tb;
  PktTx_Beh_i tx;
  dut dut(tx,...);
```

```
  initial begin
    ...
    tx.send(pkt);
  end
endmodule
```

- Don't waste simulation cycles on detail you don't care about

# Case Study: Verifying a Packet Switch



- Not just looking at inputs and outputs

March 11 - 12, 2002

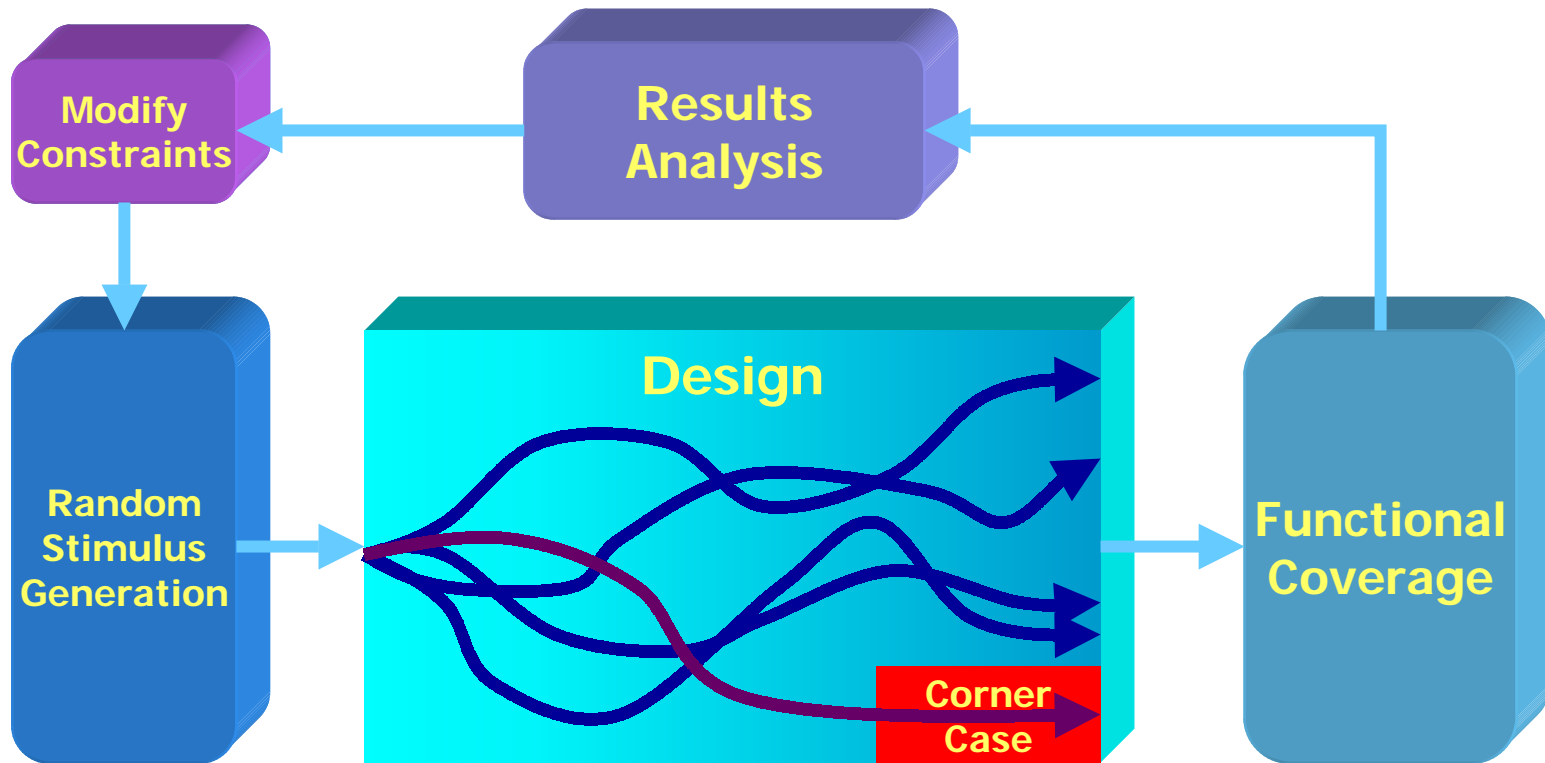


# Functional Coverage

---



# Random Stimulus and Functional Coverage



✓ Analyzing Results in the Test Lets You Modify Constraints On-The-Fly

# Functional Coverage Methodology

- With a Random Test, You Have to Know What You're Looking For
- Scenario coverage
  - Do I have a test for every "macro function" of the design?
  - Did I check all the "gozintas" and "gozoutas"?
  - Data accumulation in the test
- Protocol/transaction coverage
  - Have I accessed the DUT in every way possible?
  - Tweak the "mode" knobs when generating traffic to the DUT
  - Data accumulation in the interface
- Corner-case coverage
  - Have I exposed all of the design issues in the DUT?
  - Designer is concerned about condition 'X'
  - Have all internal blocks talked to each other?
  - Designer embeds monitors/checkers in DUT
  - Corner-case monitors checked from test

# Scenario Coverage

- Remember our sample random test
  - Write to random addrs
  - Put expected read values in array
  - After N writes, randomly choose between reads and writes
  - After M writes, do only reads until all written addres have been read
- How do we know when we're done?
  - Check that all critical addresses have been written/read
    - Use "buckets" to count how many writes occur in each addr range
    - Track when addr generated
    - Scan through array at end
  - Go back and tweak addr range knobs to favor lightly covered ranges

**\$rand function has built-in coverage recording to count individual calls, values returned, or values returned within range "buckets"**



# Tweaking Knobs for Addr

Probabilities  
are variables

```
class hostaddr;
  logic[11:0] addr;
  rangeStruct r[0:2] = {{12'h0,12'h0FF},
                        {12'h200,12'h2FF},{12'h400,12'h4FF}};
  int prob[0:2] = {10,10,10}, cnt[0:2]={0,0,0};

  function logic[11:0] genaddr;
  casew()
  prob[0]: begin
    genaddr = ($rand(.ranges({r[0]}));
    cnt[0]++;
  end
  prob[1]: return($rand(.ranges({r[1]}));
  prob[2]: return($rand(.ranges({r[2]}));
  endcase
  endfunction
endclass

hostaddr a;
logic [11:0] addr = a.genaddr;
...
if(a.cnt[0] > 10)
  a.prob[0] = 2;
addr = a.genaddr;
```

range[0] is less likely now

# Typical Coverage Points

- Input packet switch
  - Multiple simultaneous packets
  - Min/max size packets
  - Router packet collisions
    - Same priority
    - Different priority
  - Program packet collisions
  - Host R/W during packet
- Memory
  - Program packet from ISwitch
  - **Host R/W during packet write**
  - Packet read from OSwitch
  - CPU read/write
- CPU
  - Program packet execution
  - Program suspended
  - Instruction types
  - Address modes
  - Instruction pairs
- Output packet switch
  - Router/program packet collision
  - All output channels used
  - New packet before Xmit complete

# Protocol/Transaction Coverage

- Identify relevant attributes of transactions
  - Address region
  - Cycle type
- Associative array of counters tracks how many of each unique transaction type
- Associative array of queues records each transaction of a specific type
  - Can be used for post-process analysis

```
typedef enum {rd,wr,bRd,bWr} cycT;  
  
typedef enum {reg1,reg2,reg3} regionT;  
  
typedef struct {  
    bit mode;  
    cycT cyc;  
    regionT region;  
} transT;  
  
typedef struct {  
    time start,end;  
    logic[11:0] addr;  
    logic[31:0] data;  
} transInfo;  
  
transT trans;  
int transCnt[transT];  
transInfo transList[transT][0:$];  
transInfo this_trans;  
  
... begin  
transCnt[trans]++;  
transList[trans] = {transList[trans],  
                    this_trans};  
end
```

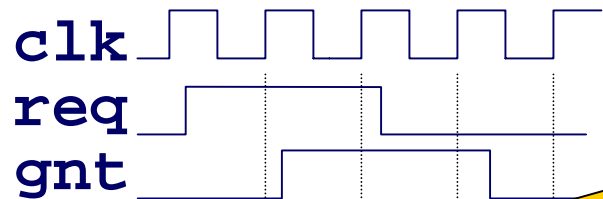
# Design Corner-Case Coverage: Assertions

- Designers make assumptions
  - About the environment
    - "The way I read it, the spec says..."
    - "The handshake protocol is..."
    - A never happens before B
    - C & D are mutually exclusive
  - About the implementation
    - This FSM is one-hot
    - Signal E will only be asserted for one cycle
    - Wr will never assert when the FIFO is full
- Need to validate the assumptions
  - Execute them in simulation
    - Specify assertions as part of the design
    - Must be executable code
    - Easy Verilog-style syntax
  - Use them in formal verification
    - Single-point-of-specification in the design eliminates ambiguity and misinterpretation
    - Exhaustively verify blocks individually
  - Executing assertions in system simulation ensures other blocks conform to assumptions

# Assertions

- See paper 4.5
  - Tuesday 11am, track 4
  - Adding SUPERLOG design assertions extensions to SystemVerilog
- Boolean assertions
- Sequential assertions
- Interval assertions

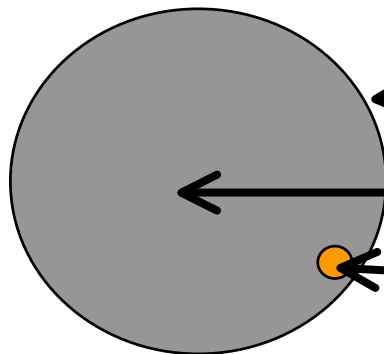
# Assertion Power



Sequence

```
assert @(posedge clk)
  (req && !gnt; req && gnt;
   !req && gnt; !req && !gnt);
```

Sample  
Clock



All possible sequences - hard to code checks

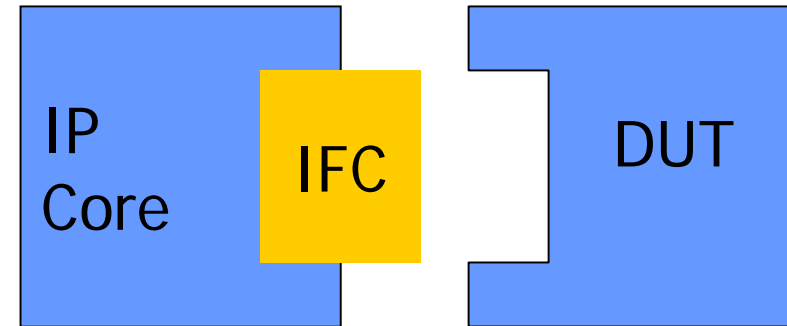
Illegal sequences

Legal sequences - easy to code checks

- Concise mechanism to test design intent
- Easily applied to simulation and model checking
- Takes guesswork out of specifying possible faults

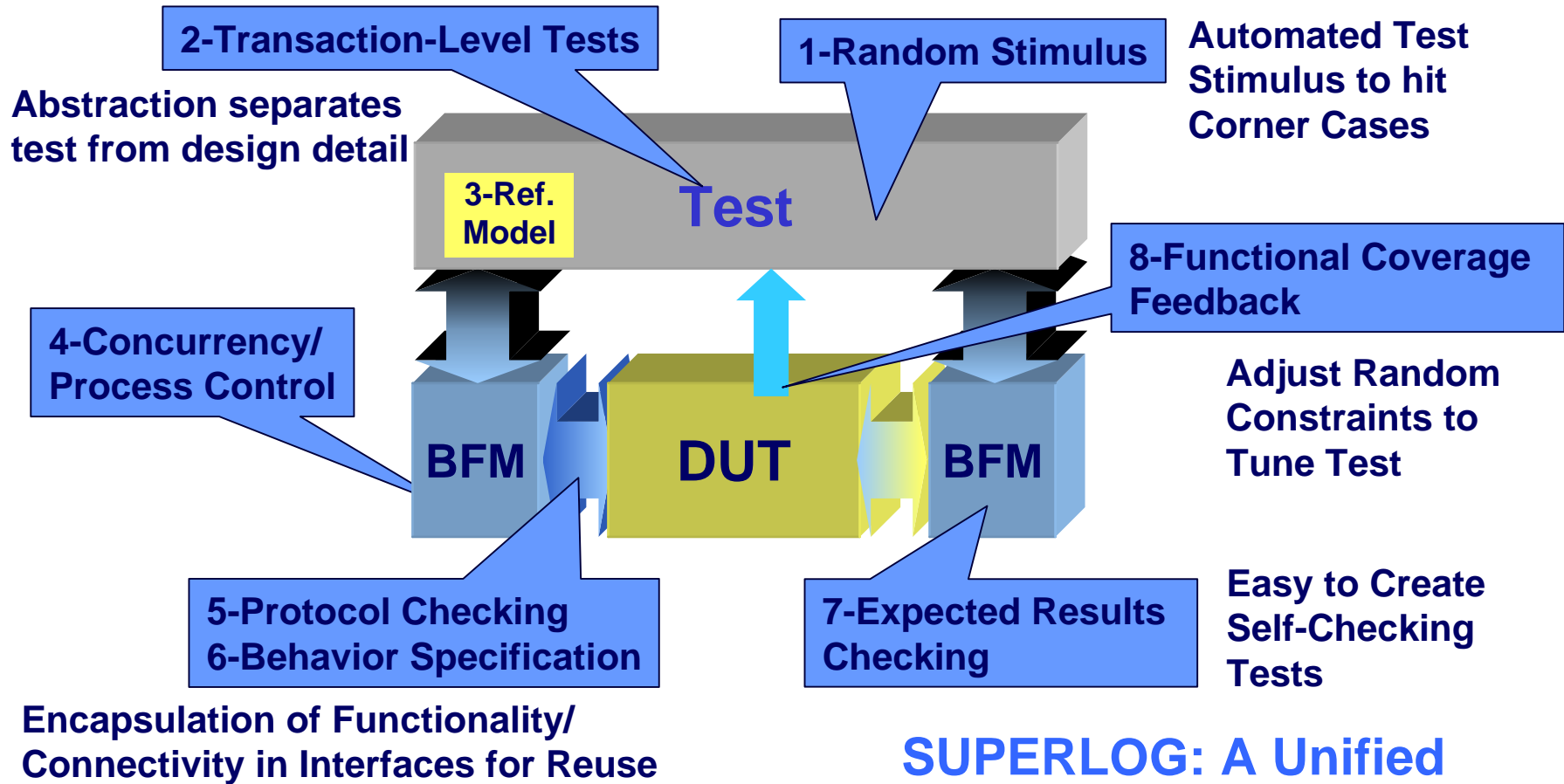
# IP Verification

- Using interfaces guarantees that IP core is connected correctly
- Interface has “public” and “private” data, so visibility can be provided to some internals
  - Eases debug if there is a problem
- Processes in interfaces allow built-in protocol checking
  - Built into IP
  - Automatically alerts user if they violate the protocol



```
interface ifc;  
...  
assert @(posedge clk) (!(rd && wr));  
  
always @(posedge clk)  
    if(rdy == 0)  
        assert @(posedge clk) (rdy == 1);  
...  
endinterface
```

# SUPERLOG Includes Key Verification Components



**SUPERLOG: A Unified  
Language for Verification**



March 11 - 12, 2002



# Using C/C++ Code with SUPERLOG

---



# Uses of C/C++ Code in Verification

- As reference models
  - Algorithmic model provides “the right answer”
  - Need an easy way to call C/C++ functions
- As test/TestBench
  - C/C++ code drives DUT signals via PLI
  - Complex PLI layer
  - Difficult to include timing
- As embedded software application

# CBlend – Adding C/C++ to Verification

No tf's, no PLI, no special scheduling, etc. Fast and Simple

## SUPERLOG calling C

### SUPERLOG Code

```
import "C" typedef cell_header_t;
import "C" function char hec(cell_header_t);

always @(posedge go) begin
    atm_cell.header = $rand();
    atm_cell.hec = hec(atm_cell.header);
end
```

### C Code

```
typedef char cell_header_t [4];
unsigned char hec(cell_header)
unsigned char cell_header[4];
{
    register unsigned char hec_accum = 0;
    register int i;
    for ( i = 0; i < 4; i++ ) {
        hec_accum = syn_tbl [hec_accum^cell_header[i] ];
    }
    return hec_accum ^ 0x055;
}
```

## C calling Verilog/SUPERLOG

### C Code

```
extern void v_delay ();
void c_thread1 () {
    for (;;) {
        v_delay ();
        printf ("c_thread1/v_delay return\n");
    }
}
```

### SUPERLOG Code

```
import "C" task c_thread1();
export "C" task top.v_delay;
module top;
    ...
    task v_delay;
    begin
        $display ("%d v_delay called", $time);
        #55 $display ("%d v_delay returns", $time);
    end
endtask
endmodule
```

Time  
Delay!

# CBlend – Pointers Improve Efficiency

## SUPERLOG Code

```
import "C" typedef cell_header_t;  
import "C" function char hec(cell_header_t);  
  
always @(posedge go) begin  
    atm_cell.header = $rand();  
    atm_cell.hec = hec(ref atm_cell.header[0]);  
end
```

Pass pointer

## C Code

```
typedef char cell_header_t [4];  
unsigned char hec(unsigned char cell_header[])  
{  
    register unsigned char hec_accum = 0;  
    register int i;  
    for ( i = 0; i < 4; i++ ) {  
        hec_accum = syn_tbl [hec_accum^cell_header[i];  
    }  
    return hec_accum ^ 0x055;  
}
```

Don't have to  
copy data,  
just use the  
pointer

March 11 - 12, 2002

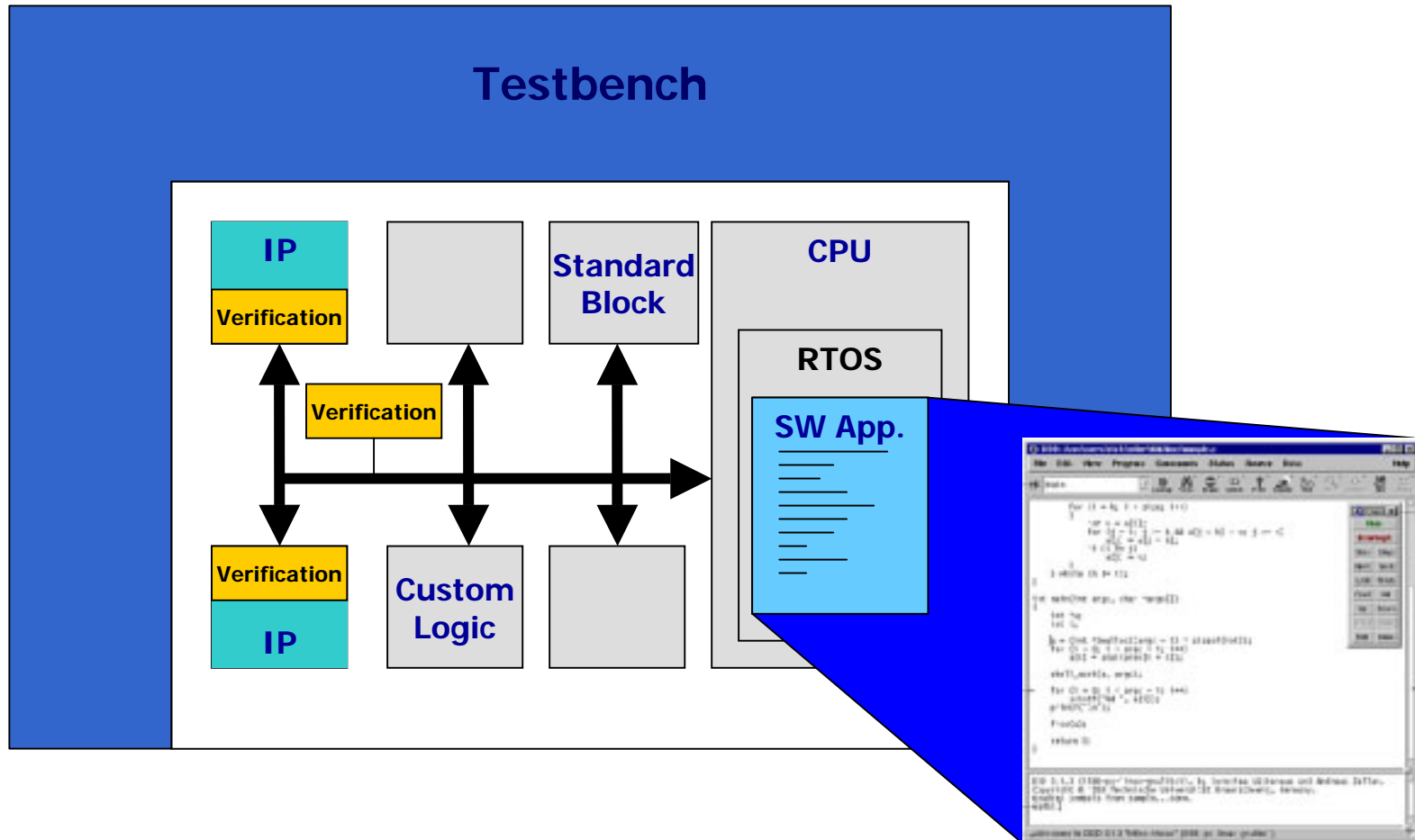


# Hardware/Software Co-Verification

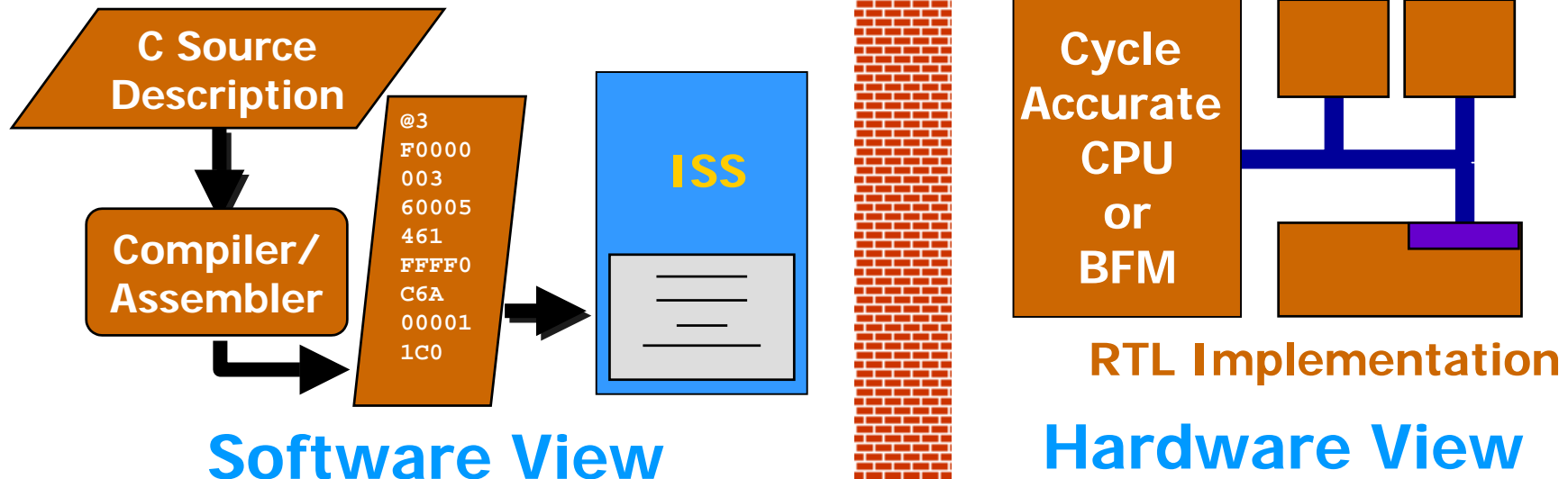
---



# The SoC Verification Platform



# Existing Platform World



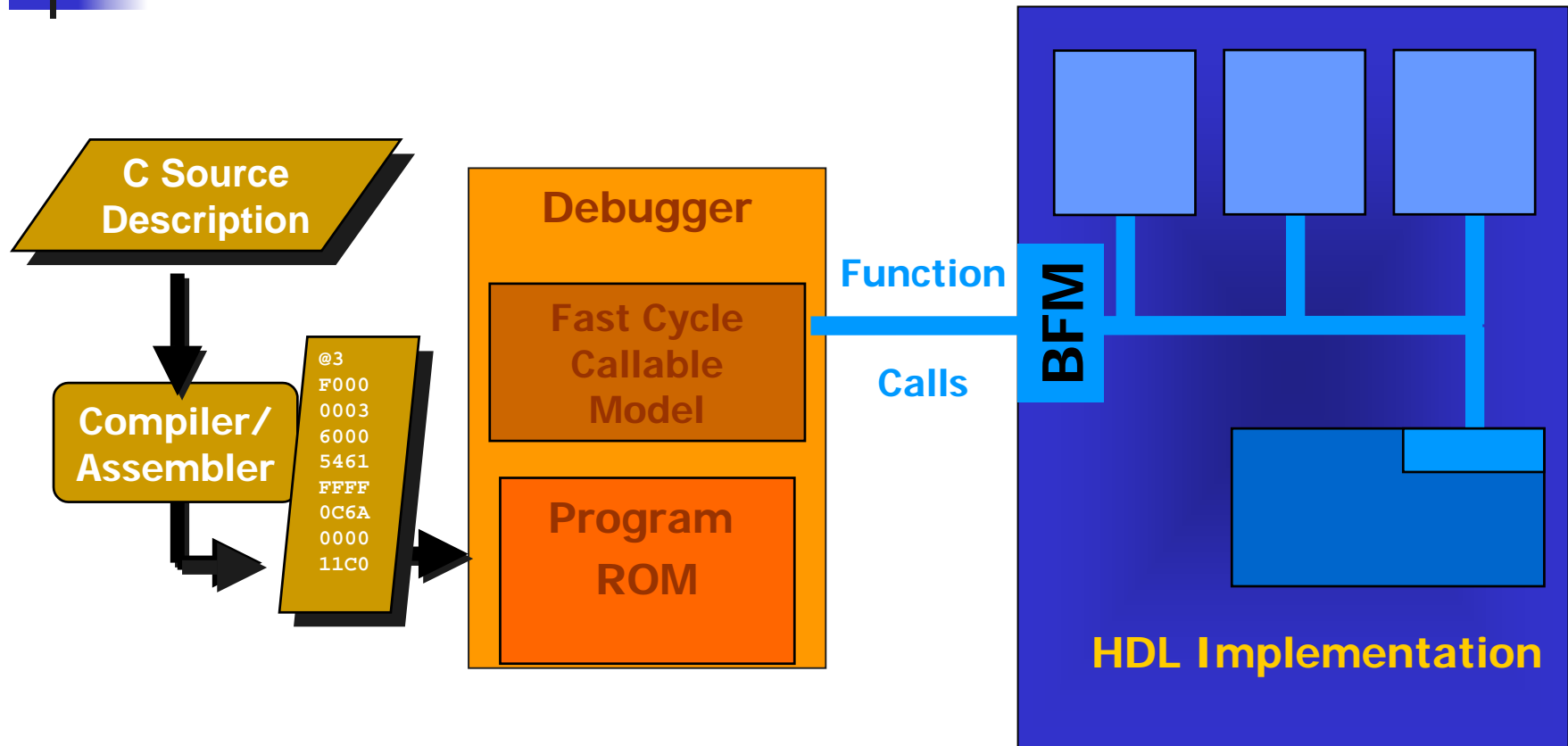
**Embedded SW developers require:** **HW developers require:**

SW debugger w/ HW access  
Many instruction cycles per sec.  
Low cost 'execute-only' platform  
Concurrent engineering with HW

Reliable IP reuse/development  
Extensive, high speed verification  
Varied model language/styles  
Concurrent engineering with SW

**Current Solutions Use Slow, Complicated Interfaces**

# What Would Be Ideal?

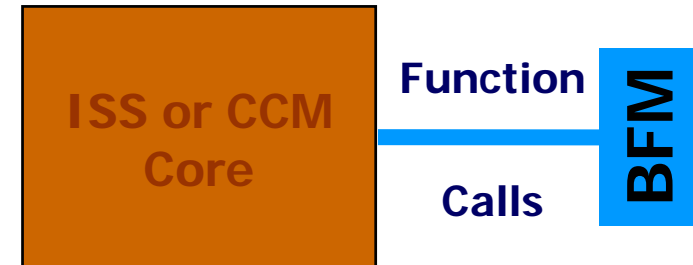


**Need Complete System Available - With  
Fast, Simple Access and Single Debugger**

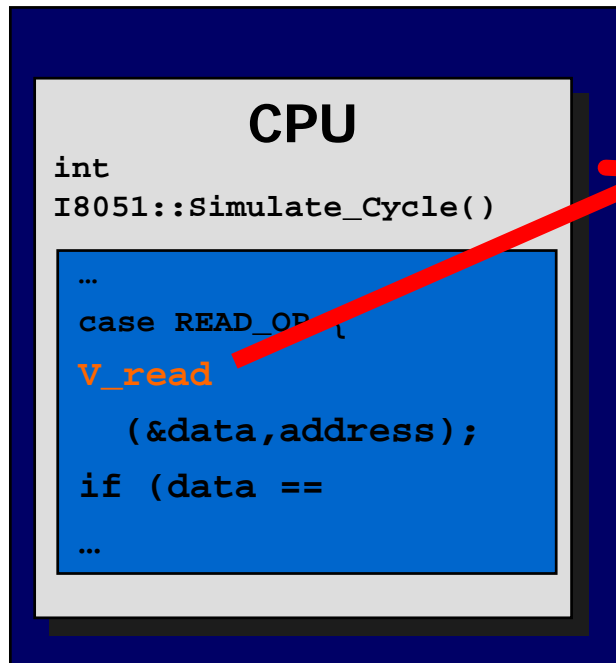


# C Model Drives HDL BFM Directly

Unique Capability Provides  
Simple Integration Setup



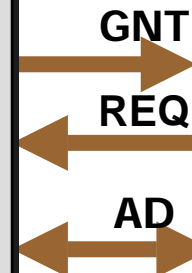
**i8051.cc**



**i8051.v BFM**

```

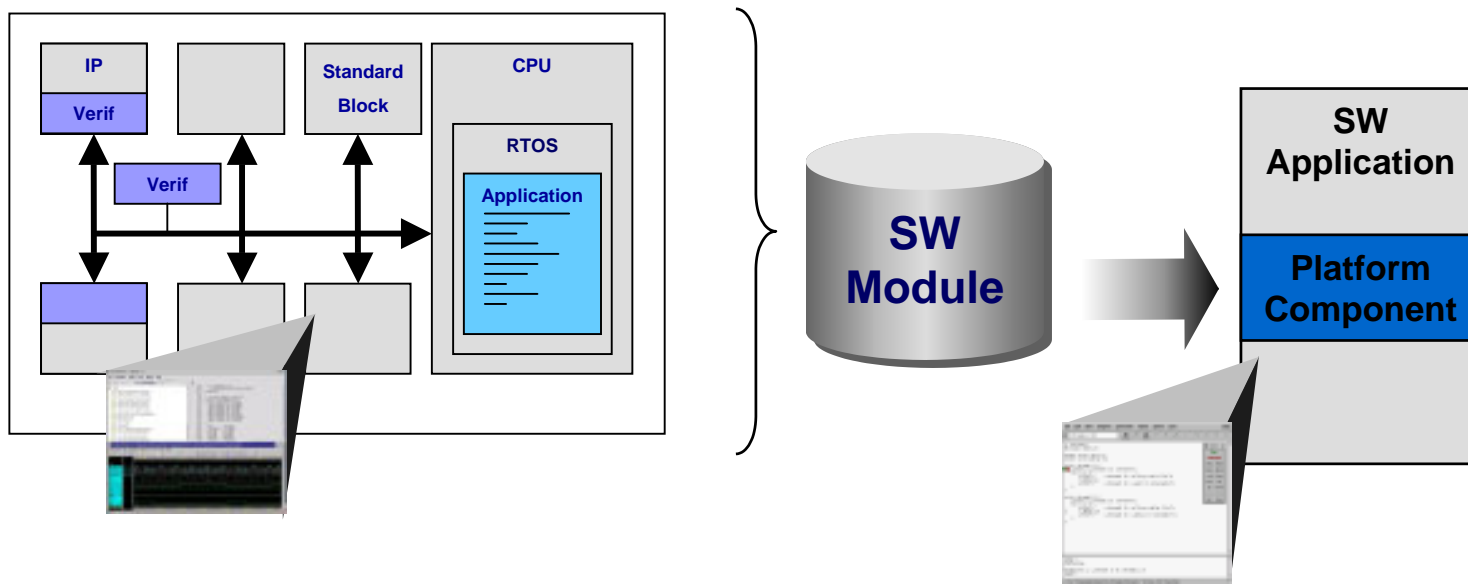
task V_read;
output data;
input address;
begin
  REQ = 0;
  wait(GNT)
  #10
  AD = address;
...
end
  
```



# Embedded SW Solution: Make Simulation Part Of SW App

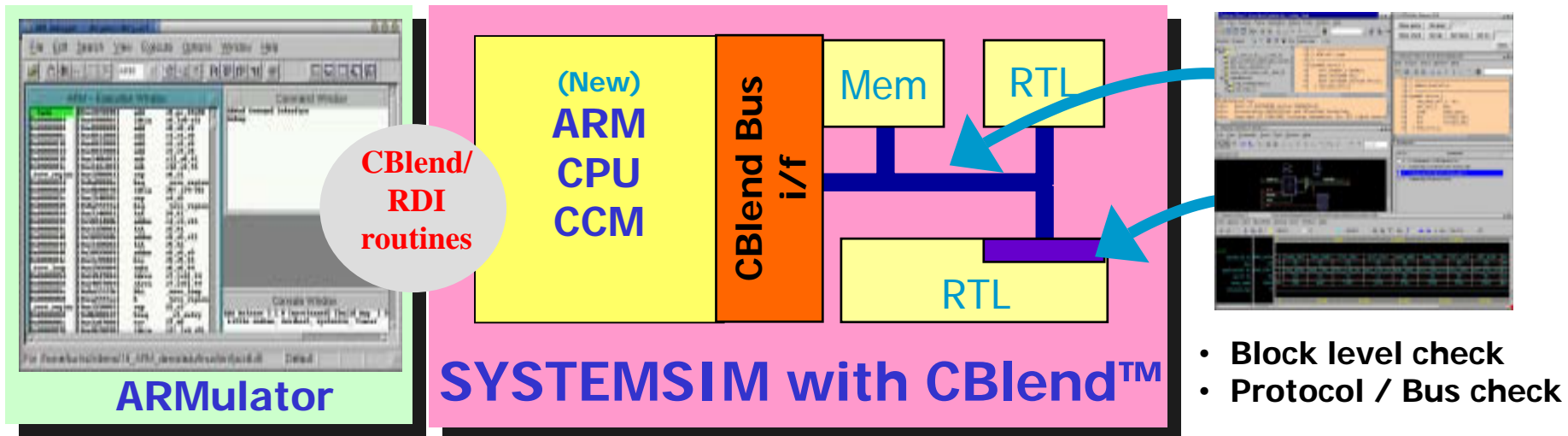
## For The First Time, Platform Can Be Verified Pre-Prototype

- Easy, fast C/HDL mixing for efficient model usage
- Real concurrent engineering
- HW & SW components designed in standard environments



# ARM CCM and Systemsim Environment

**HW and SW Debug Maximizes Flexibility To Find Problems Quickly**

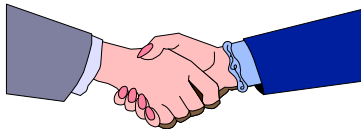
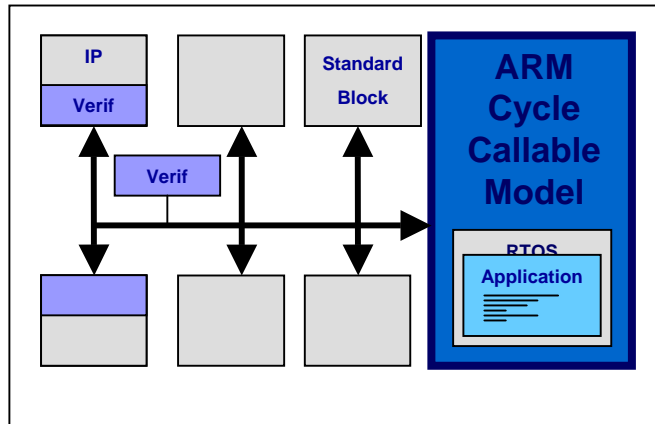


**Platform = design based on pre-verified blocks of IP**

- For efficient re-use & re-configuration the IP blocks have to work together
- Verification features in SYSTEMSIM allow communication protocol and block checking to be embedded in the design – errors trapped immediately
- New ARM CCM model can augment DSM for hardware verification (FAST)
- Other debuggers could be used through RDI

# ARM / Co-Design Automation Partnership

## Teaming Up To Accelerate Platform Design



- ARM CCM model operates many X faster with Systemsim than with other solutions
- Full SW debug and cycle functionality included
- Allows real testing of SW and hardware together  
FOR THE FIRST TIME

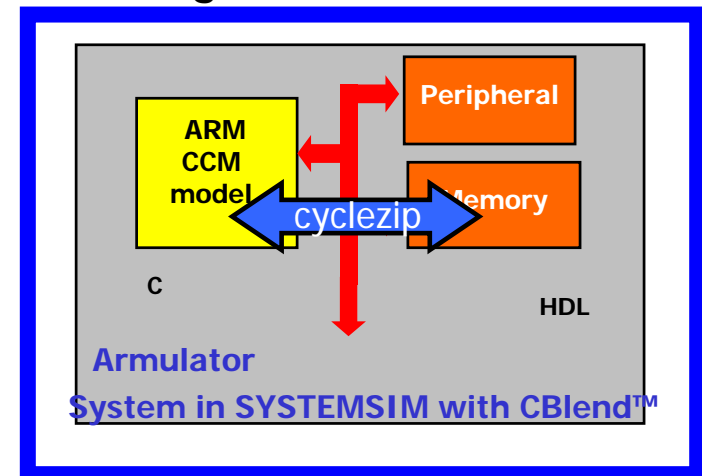
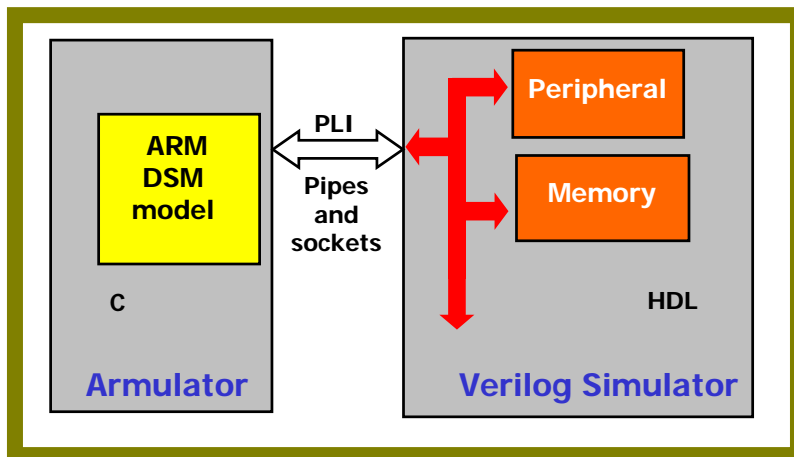
**ARM SW Can Be Tested Before Prototype Available**

# ARM/SYSTEMSIM

## Performance Mode Options

Model	Connection	Mode	Run Time (Solaris)	Performance wrt DSM
ARM7 CCM	CBlend	Normal	3.68 sec	165 x faster
ARM7 CCM	CBlend	CycleZip 1	2.26 sec	269 x faster
ARM7 DSM	PLI	Normal	607.5 sec	1 x

Source: ARM "Easy" Benchmark, 18,000 lines Verilog RTL + ARM Model



- On very large customer designs 3-6X improvement common

March 11 - 12, 2002



## Wrap-Up

---



# SUPERLOG Extends Verilog

- SUPERLOG extended Synthesizable subset donated to  
Accellera - *SystemVerilog*
  - All of Verilog-95 and Verilog-2001
  - Enhanced datatypes and user-defined types
  - Interfaces
- Design assertions subset also donated
  - Powerful tool for designers to contribute to verification

"SUPERLOG is setting the right direction for Verilog language development, providing enabling capabilities across design and verification.

**SUPERLOG is on track to either replace Verilog or be the next Verilog."**

Cliff Cummings,  
Senior Consultant,  
Sunburst Design, Inc;  
Member of Verilog IEEE-1364  
& Accellera Verilog++ Committees

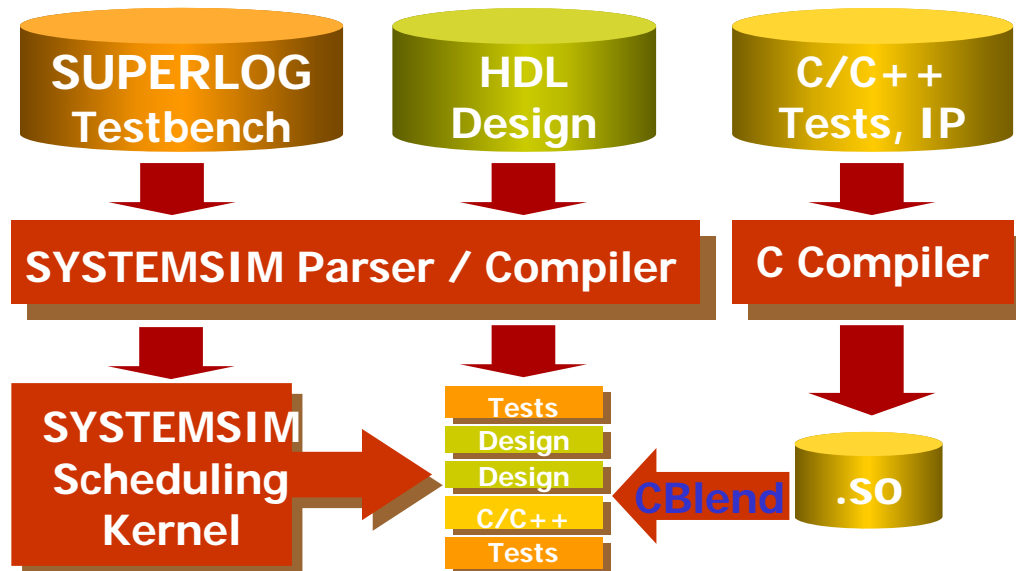
# SUPERLOG Is the Platform

- System-level language features allow top-down, architectural modeling through implementation in a single language
- Interfaces separate communication from functionality
  - Easy to change abstraction of different components
  - Simplifies IP integration and delivery
- Advanced verification features included
  - Directed random stimulus
  - Functional coverage analysis
  - Single kernel: no bottleneck
- C/C++ interoperability
  - Easy to incorporate reference models
  - Unique Systemsim architecture lets C/C++ call HDL
- Powerful, flexible platform for embedded software verification



# Systemsim Architecture Enables Speed

## UNIfied VERification Simulation ALgorithm

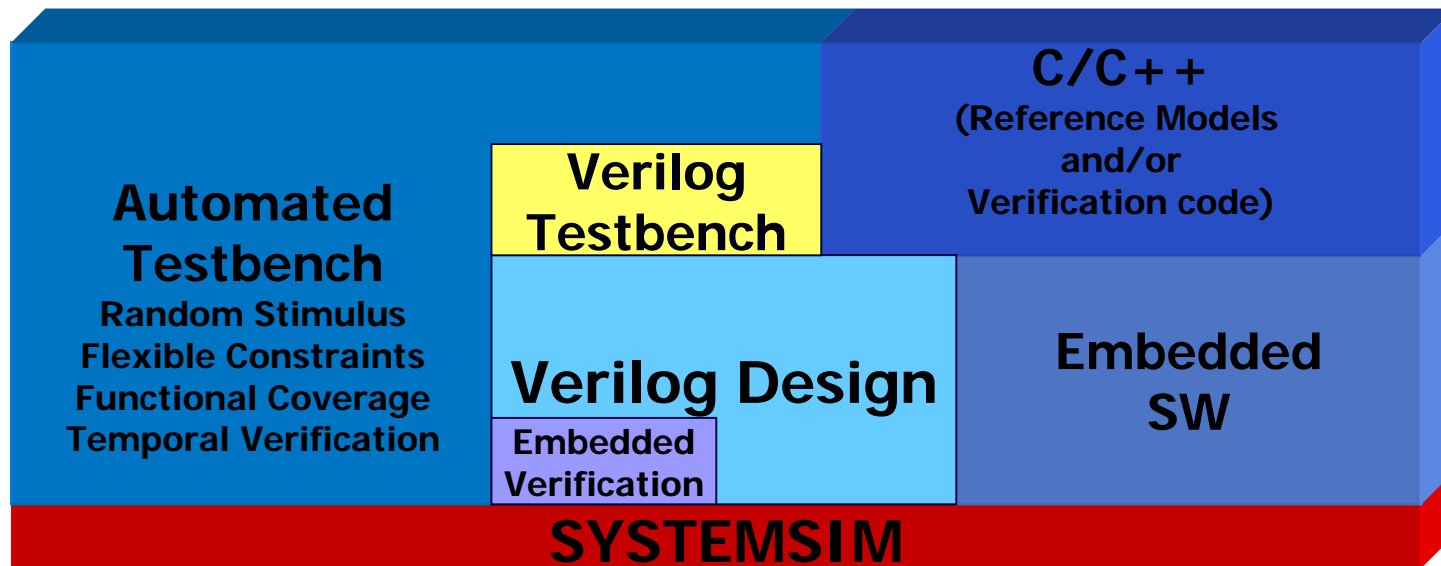


- Tests, design, and C compiled together
- High performance improvement
- Single debug environment, licensing, etc

**Substantial simulation environment  
productivity improvement  
Higher quality and reduced design cycles**

# Evolutionary Methodology

## Building on Existing Infrastructure To Save Time



"By enabling the efficient representation of complex structures, SUPERLOG allows intelligible models to be produced quickly and easily. I would very strongly recommend SUPERLOG and SYSTEMSIM as the language and simulator of first choice."

**Jon Beecroft, Principal Consultant, Quadrics Supercomputers World, Ltd.**

# CDA Customers, Investors, & Partners

## Some Partners



## Some Customers



## Some Investors

