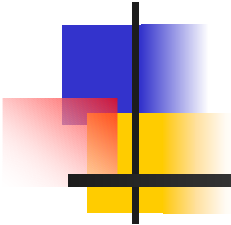


**March 11 - 12, 2002**

# A Communication-based Design Platform: The Power of SUPERLOG and SystemVerilog Interfaces



Tom Fitzpatrick

fitz@co-design.com

Co-Design Automation, Inc.





# Agenda

---

- Defining a Platform
- What is an Interface?
- SUPERLOG/SystemVerilog Language Basics
- Interface Syntax and Basics
- Encapsulating Communication Methods in Interfaces
- Communication-Based Verification
- Communication-Based Design Exploration
- Wrap-Up
- Questions

# What Is a Platform?

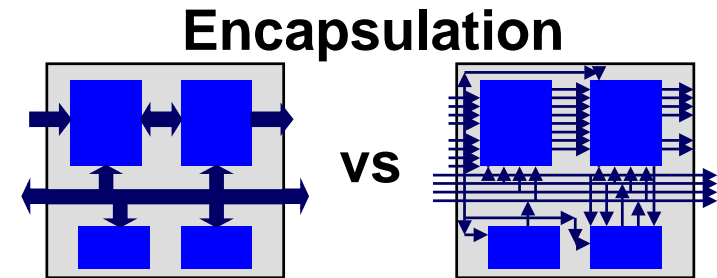
- Complex Systems Require a Divide-and-Conquer Methodology
  - Separate the “What” from the “How”
  - Identify Functions that the device must perform
  - Identify communication paths between functional blocks
- Methodology Must Promote Reuse
  - Common applications can be pieced-together from common collections of functional blocks
  - A Platform must allow exploration of tradeoffs between different implementation choices of each function

# More Platform Concerns

- Abstract Modelling Allows Exploration of Function Independent from Implementation
- Must Facilitate Simple Transition from Abstract to Implementation (RTL)
  - Allow abstract and RTL models to interact easily
  - Make it easy to swap between different implementation models for given blocks
- Methodology Must Focus on Interfaces and Communication Between Blocks
  - Functional Blocks operate on data once they get it
  - Interfaces define how data moves between blocks

# Interfaces: Communication-Based Design and 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 Methodology



- Think About the Fat Arrows on a Block Diagram as More Than Just Wire Bundles
  - Wires are an implementation choice of how to communicate information between blocks
  - Focus on the information that is being communicated

# Interface Basics

- Encapsulate Connectivity
  - Bundle of nets/wires
  - Instantiated
  - Passed through ports as single item
  - Significantly reduces port-list clutter
  - Improves maintainability

```
interface cpu_i (input bit rst);
    wire      Bus Mode = BusModeType;
    logic [11:0] Addr;
    logic      Sel;
    wire [ 7:0] Data;
    logic      Rd_DS;
    logic      Wr_RW;
    logic      Rdy_Dtack_r = 'z;
    wire      Rdy_Dtack = Rdy_Dtack_r;

    task automatic write (input logic [11:0] A, input logic [7:0] D);
endtask
endinterface

module squat_m(utopia_i rx[max:0], utopia_i tx[max:0],
                cpu_i  cpu, clock_i clock);

    initial
    begin
        rx[1].atm_start_rx_clk(); tx[1].atm_start_tx_clk();
        cpu.write('b0, 'b0);
    end
endmodule
```

# Additional Power of Interfaces

- Encapsulate Functionality
  - Parameters, variables, functions, tasks
  - Dynamic types
- Facilitates reuse
  - Members referenced locally from connected modules
  - Processes allow for protocol self-checking
- Abstract methods
  - Defined in one module
  - referenced in other modules connected via the interface

March 11 - 12, 2002

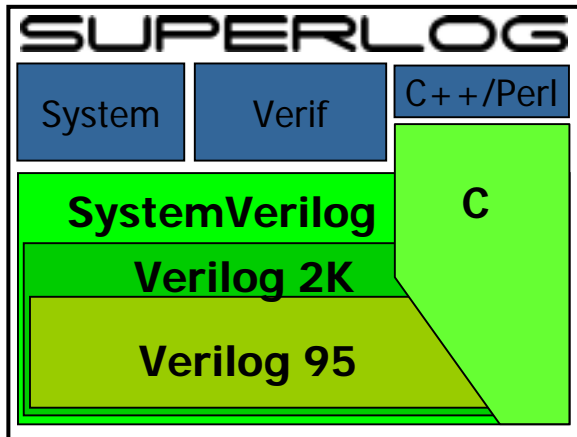
# SUPERLOG/SystemVerilog Language Basics

---



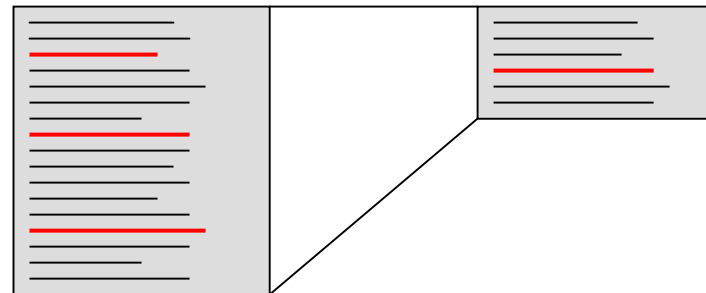


# SUPERLOG ESS => SystemVerilog



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/SystemVerilog Has 2 and 4 State Datatypes

- Verilog
- SUPERLOG/  
SystemVerilog

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

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

- SUPERLOG/  
SystemVerilog

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

**Equivalent to these  
4-valued types**

- SUPERLOG/  
SystemVerilog

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

**These types have  
two-valued bits  
(0 and 1)**

**If you don't need the x and z values then  
use the 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/SystemVerilog datatype

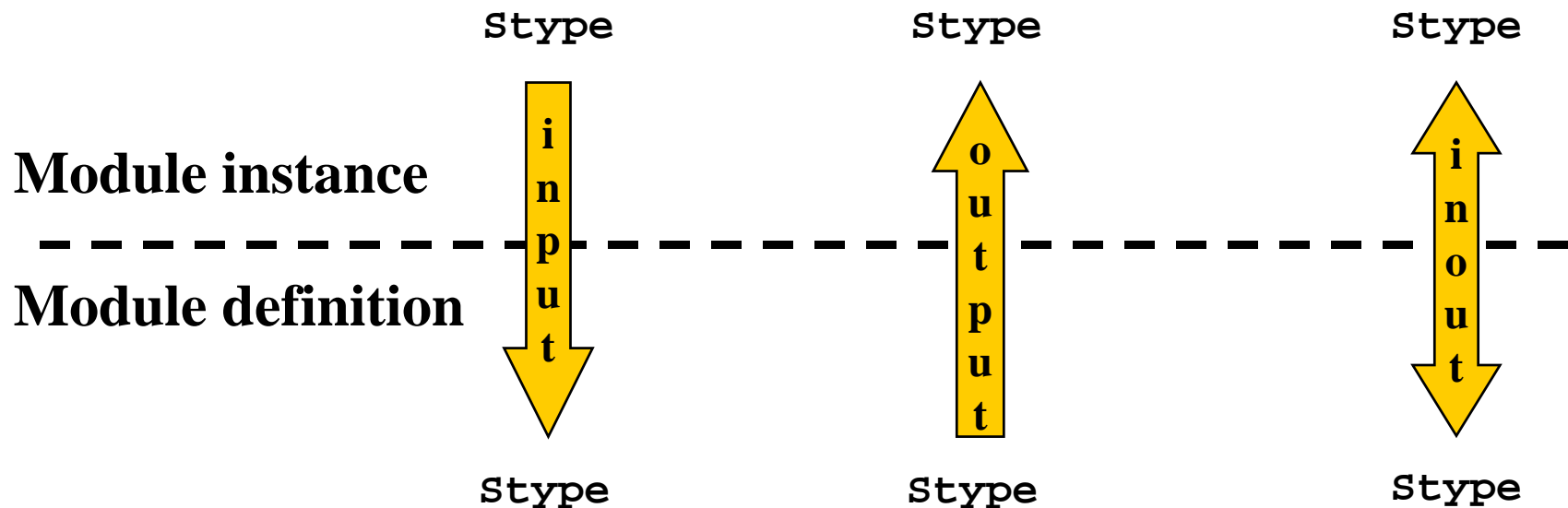
```
typedef struct {  
    real R;  
    real I;} Complex;  
Complex X,Y,Z;  
always @(negedge clk)  
    begin  
        X = Complex_F1(Z);  
        Y = Complex_F1(Z);  
    end  
always @(posedge clk)  
    X = Complex_F1(Y);  
  
assign Z = Complex_F1(X);
```

**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



# SUPERLOG/SystemVerilog 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
```

# 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
- Not part of SystemVerilog

**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.
- Not Included in SystemVerilog

# 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**

March 11 - 12, 2002



## Interface Basics

---



# 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
```

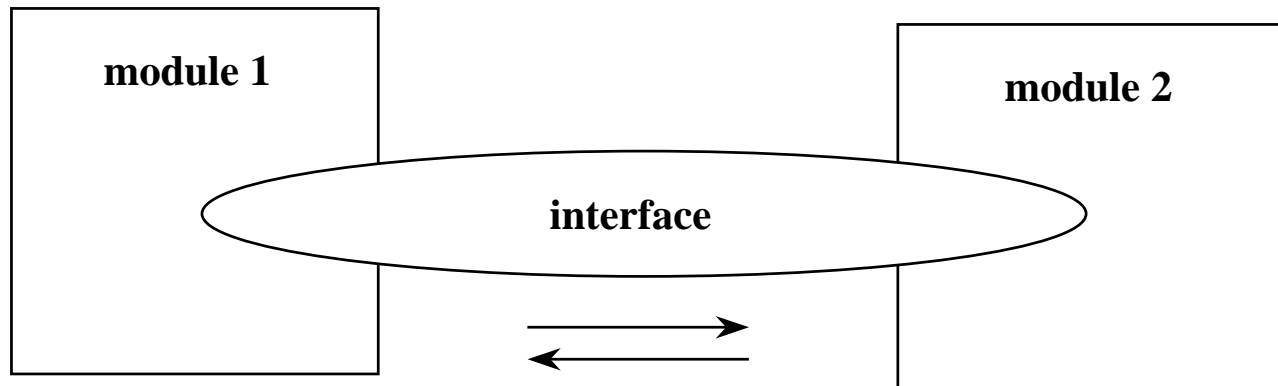
```
wire w;
intf if1;

modA a (w, if1);
```

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

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

# Connection Object



**Interfaces are not just for  
encapsulation ...**

- **An interface describes the communication between modules**

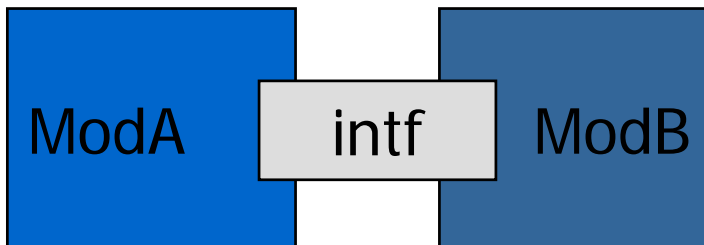
# How Does an Interface Work?

```
interface intf;
bit A,B;
endinterface
intf w;
modA m1(w);
modB m2(w);
module modA (intf i1);
endmodule
module modB (intf i1);
endmodule
```

Instantiate Interface

Connect Interface to Module Port

Declare Module Port to  
be an Interface

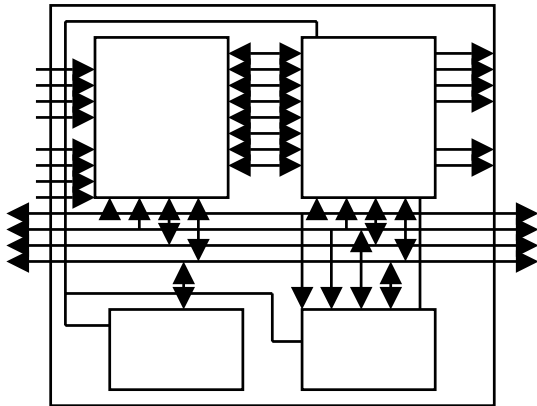


An interface is  
similar to a module  
straddling two  
other modules

- Allows structuring the information flow between blocks

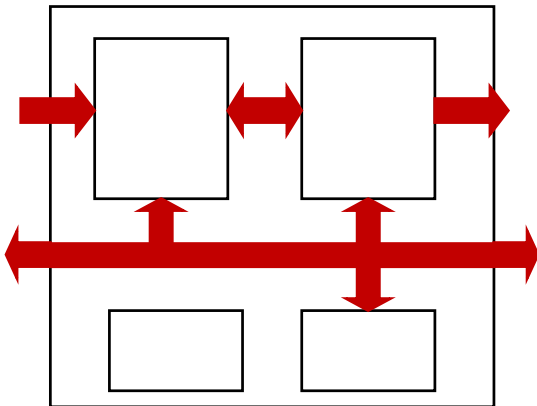


# Systems Without the Interface Construct



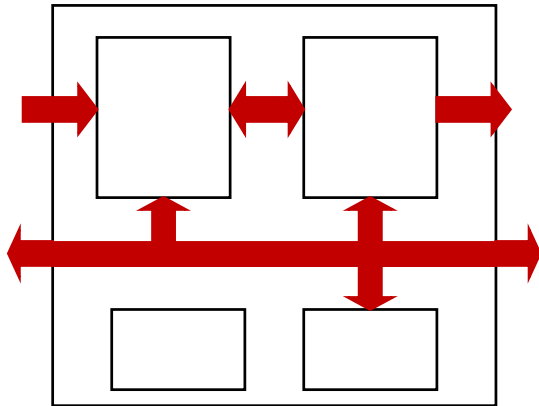
**schematics map  
to a Verilog  
netlist**

```
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,
    ....
)
```



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

# Interfaces Reduce Interconnection Text



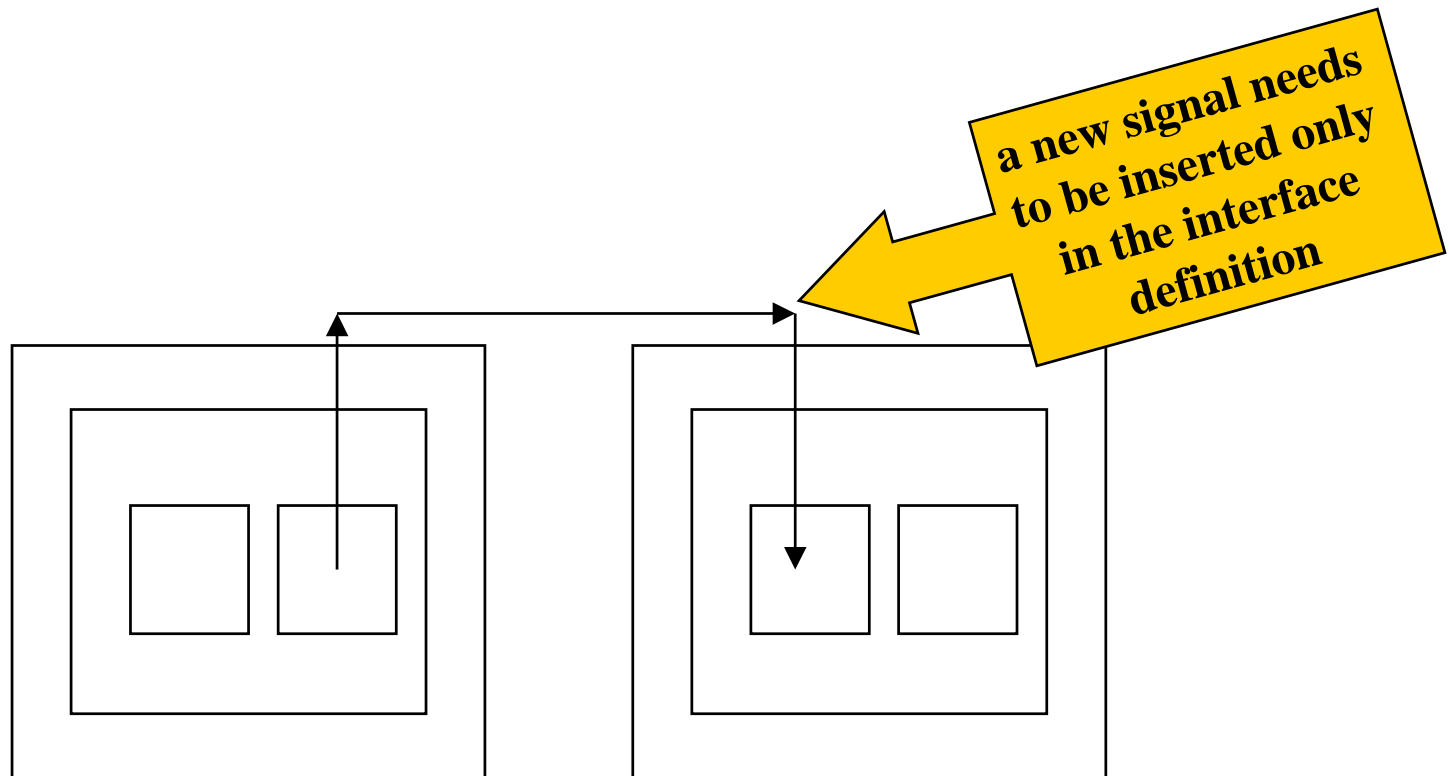
```
interface system_bus_intf;  
  wire [63:0] data_bus;  
  wire ctl1, ctl2, ctl3,  
  bit pre1, pre2, pre3,  
  ...  
endinterface
```

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

```
module mem_control(  
  system_bus_intf system_bus,  
  memory_interface mem_bus,  
  ...  
)
```

- 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

# Simple Example Without Interfaces

```
module memMod(input    logic req,
              bit      clk,
              logic start,
              logic[1:0] mode,
              logic[7:0] addr,
              inout logic[7:0] data,
              output logic gnt,
              logic rdy);

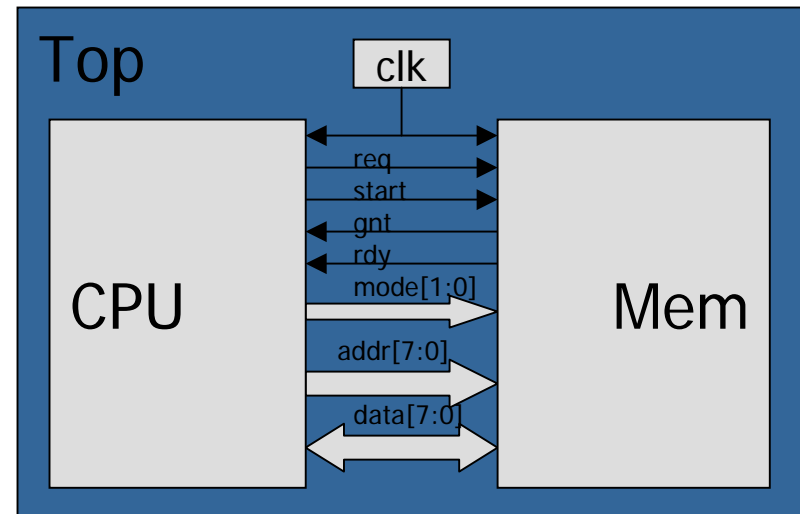
always @(posedge clk)
    gnt <= req & avail;
endmodule

module cpuMod(input bit clk,
              logic gnt,
              logic rdy,
              inout logic [7:0] data,
              output logic req,
              logic start,
              logic[7:0] addr,
              logic[1:0] mode);

endmodule
```

```
module top;
    logic req,gnt,start,rdy;
    bit    clk = 0;
    logic [1:0] mode;
    logic [7:0] addr,data;

    memMod mem(req,clk,start,mode,
               addr,data,gnt,rdy);
    cpuMod cpu(clk,gnt,rdy,data,
               req,start,addr,mode);
endmodule
```



# Simple Example Using Interfaces

```
interface simple_bus;
  logic req, gnt;
  logic [7:0] addr, data;
  logic [1:0] mode;
  logic start, rdy;
endinterface: simple_bus
```

Bundle  
signals in  
interface

```
module memMod(simple_bus a,
               input bit clk);

  logic avail;
  always @(posedge clk)
    a.gnt <= a.req & avail;
endmodule
```

Refer to  
intf signals

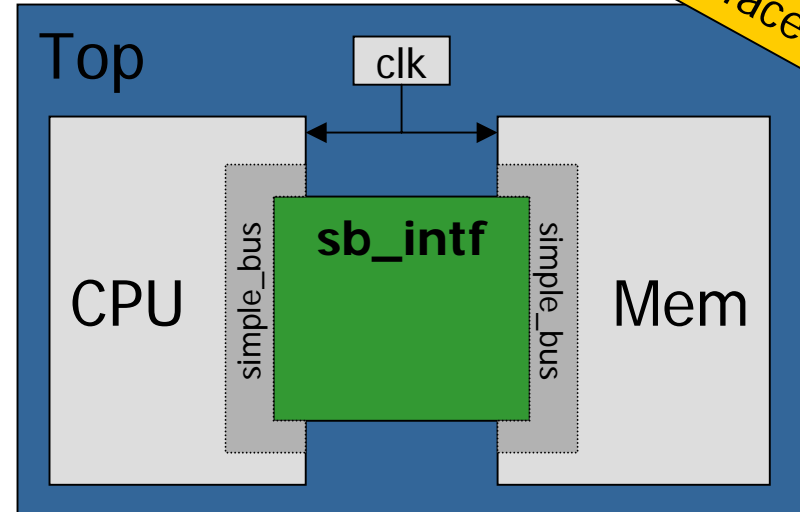
```
module cpuMod(simple_bus a,
               input bit clk);
endmodule
```

```
module top;
  bit clk = 0;
  simple_bus sb_intf;

  memMod mem(sb_intf, clk);
  cpuMod cpu(.b(sb_intf),
              .clk(clk));
endmodule
```

intf  
instance

Connect  
interface



# Reuse Issues

- Previous Example “Hard-coded” Interface Type in Module Definition
  - Modules can only be connected to a simple\_bus
  - Modules automatically inherit simple\_bus signals
  - Allows flexibility as long as changes are local to simple\_bus definition
- Proper Reuse Methodology Requires Modules to Be Flexible
  - Include generic “interface” port in module definition
  - Modules automatically inherit signals from interface of whatever type is connected
  - Encapsulate signals and protocol in interface definition

# Simple Example Using Generic Interface Ports

```
interface simple_bus;
  logic req, gnt;
  logic [7:0] addr, data;
  logic [1:0] mode;
  logic start, rdy;
endinterface: simple_bus
```

Bundle  
signals in  
interface

Use  
generic intf  
in port list

```
module memMod(interface a,
               input bit clk);

  logic avail;
  always @(posedge clk)
    a.gnt <= a.req & avail;
endmodule
```

```
module cpuMod(interface b,
               input bit clk);
endmodule
```

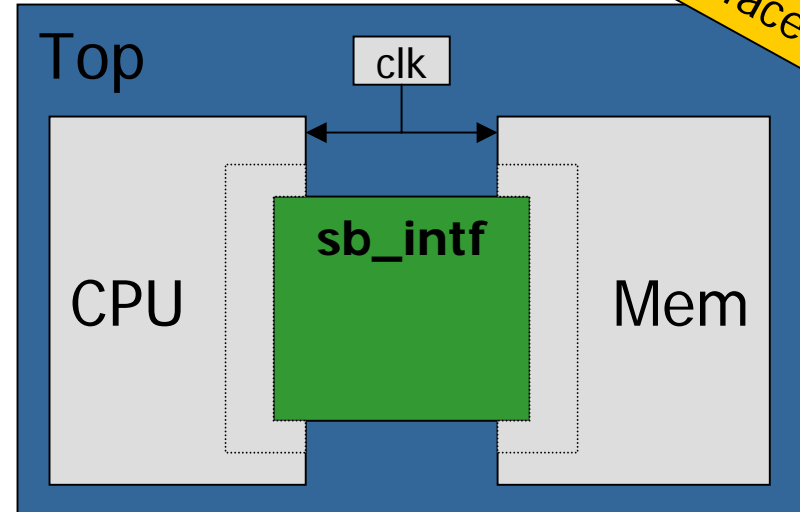
Refer to  
intf signals

```
module top;
  bit clk = 0;
  simple_bus sb_intf;

  memMod mem(sb_intf, clk);
  cpuMod cpu(.b(sb_intf),
             .clk(clk));
endmodule
```

intf  
instance

Connect  
interface



# Sharing Signals Between Interfaces

```
interface simple_bus(input bit clk);  
    logic req,gnt;  
    logic [7:0] addr,data;  
    logic [1:0] mode;  
    logic start,rdy;  
endinterface: simple_bus
```

```
module memMod(interface a);  
    logic avail;  
    always @(posedge a.clk)  
        a.gnt <= a.req & avail;  
endmodule
```

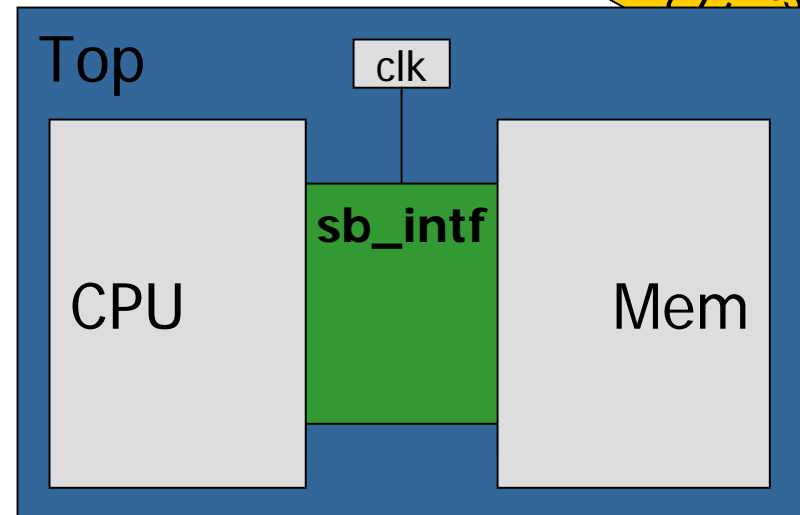
```
module cpuMod(interface b);  
    ...  
endmodule
```

clk is now in  
the interface

```
module top;  
    bit clk = 0;  
    simple_bus sb_intf(clk);  
  
    memMod mem(sb_intf);  
    cpuMod cpu(.b(sb_intf));  
endmodule
```

Connect  
clk

Simplifies  
Port List



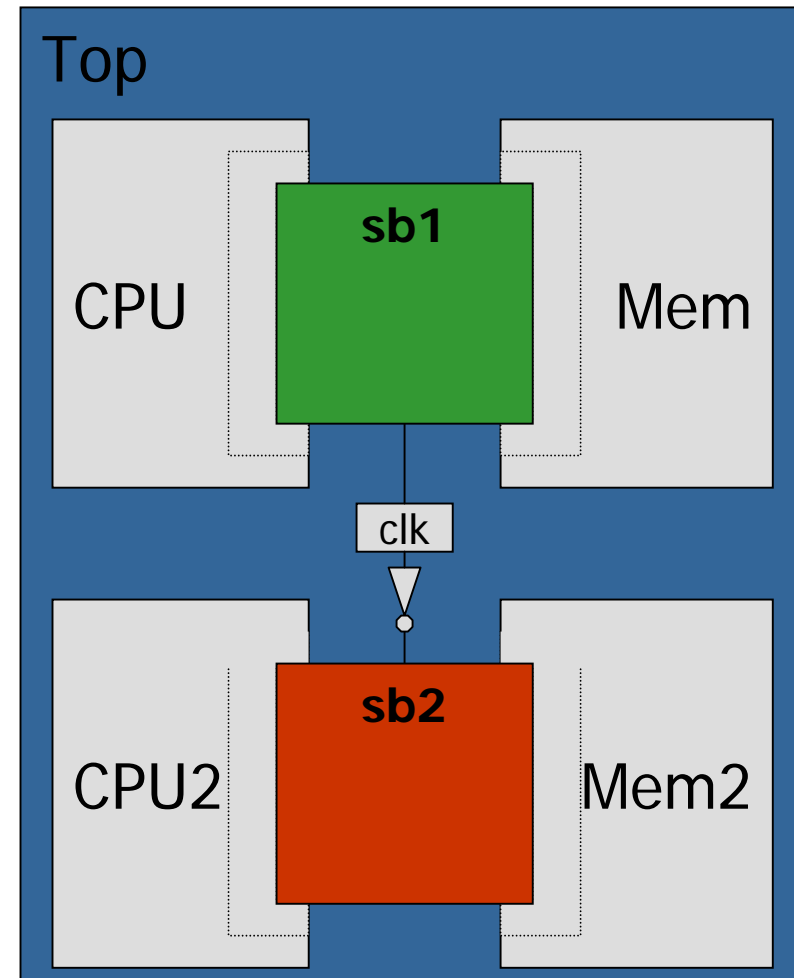
Interface Ports Let All Instances of  
an Interface Share the Same Signal



# Interface Instances: Using Different Signals

```
module top;
  bit clk = 0;
  bit clk2 = 1;
  assign clk2 = !clk;
  simple_bus sb1(clk);
  simple_bus sb2(clk2);

  memMod mem(sb1);
  cpuMod cpu(.b(sb1));
  memMod mem2(sb2);
  cpuMod cpu2(.b(sb2));
endmodule
```





# Controlling Signal/Port Direction: Modports

---

- If Direction is Not Specified, Signals in Interfaces Are Inout
- How to Specify Direction When Ports Are Different for Different Modules?
  - CPU module uses addr as an output
  - Mem module uses addr as an input
- Group Signals/Directions in Interface
  - “Modport” implies direction from point-of-view of the module using the interface
  - Can also control visibility of signals and methods

# Simple Example Using Modports in Definition

```
interface simple_bus(input bit clk);  
    logic req,gnt;  
    logic [7:0] addr,data;  
    logic [1:0] mode;  
    logic start,rdy;  
    modport slave(input req,addr,mode,  
                  start, clk,  
                  output gnt,rdy,  
                  inout data);  
    modport master(input gnt,rdy,clk,  
                  output req,addr,mode,start,  
                  inout data);  
endinterface: simple_bus  
  
module memMod(simple_bus.slave a);  
    ...  
endmodule  
  
module cpuMod(simple_bus.master b);  
endmodule
```

```
module top;  
    bit clk = 0;  
    simple_bus sb_intf(clk);  
  
    memMod mem(sb_intf);  
    cpuMod cpu(.b(sb_intf));  
endmodule
```

modport picked  
up from module  
definition

interface.modport  
hard-coded in  
module definition

# Simple Example Using Modports in Instantiation

```
interface simple_bus(input bit clk);
    logic req,gnt;
    logic [7:0] addr,data;
    logic [1:0] mode;
    logic start,rdy;
    modport slave(input req,addr,mode,
                  start, clk,
                  output gnt,rdy,
                  inout data);
    modport master(input gnt,rdy,clk,
                  output req,addr,mode,
                  start,
                  inout data);
endinterface: simple_bus

module memMod(simple_bus a);
    ...
endmodule

module cpuMod(interface b);
endmodule
```

```
module top;
    bit clk = 0;
    simple_bus sb_intf(clk);

    memMod mem(sb_intf.slave );
    cpuMod cpu(.b(sb_intf.master ));
endmodule
```

modport specified  
explicitly by  
instantiation

interface hard-coded in  
module definition...

...or use generic interface  
specification

# Modports: Controlling Visibility

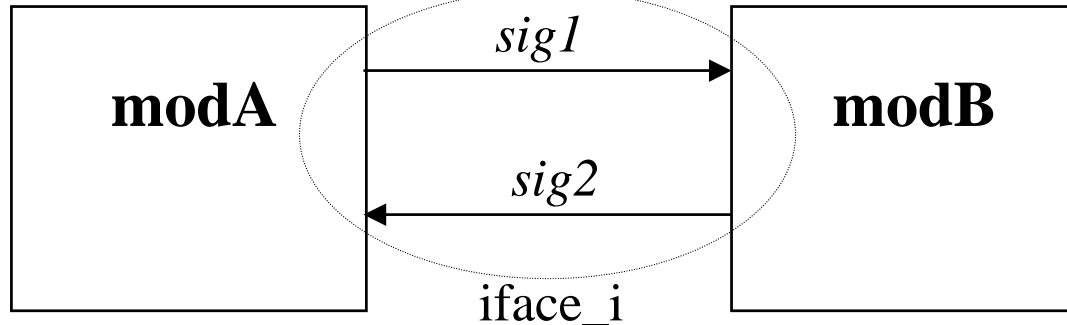
```
interface iface_i;  
  bit sig1, sig2;  
  int internal;  
  modport modeA(output sig1, input sig2),  
               modeB(input sig1, output sig2);  
endinterface
```

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
```



**March 11 - 12, 2002**



## Tasks and Functions in Interfaces: Encapsulating Communication Methods

---



# Tasks & Functions in Interfaces

- Tasks/Functions Can Be Defined In...
  - Interfaces
  - A module connected via the interface
- Allows More Abstract Modeling
  - Transaction can be executed by calling a task without referring to specific signals
  - "Master" module can just call the tasks
- Modports Control Sharing of Methods
  - Methods defined outside a module are "imported" into a module via modport
  - Methods defined inside a module are "exported" to the interface via modport
  - Effectively gives "public" and "private" methods based on whether the module uses the interface or the modport

# Using Tasks in an Interface

```
interface simple_bus(input bit clk);
    logic req,gnt;
    logic [7:0] addr,data;
    logic [1:0] mode;
    logic start,rdy;
    task masterRd(input logic[7:0] raddr);
        ...
    endtask:masterRd
```

Communication  
Task Encapsulated  
in Interface

```
    task slaveRd;
        ...
    endtask:slaveRd
endinterface: simple_bus
```

```
module memMod(interface a);
    ...
    always @(a.start)
        a.slaveRd;
endmodule
```

Interface Method  
Called From  
Instantiating Module

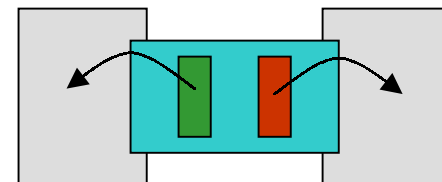
```
module cpuMod(interface b);
endmodule
```

```
module top;
    bit clk = 0;
    simple_bus sb_intf(clk);

    memMod mem(sb_intf);
    cpuMod cpu(.b(sb_intf));
endmodule
```

By default, all Interface Methods  
can be called from any module  
that instantiates the Interface

What if we want to restrict  
"slave" modules from calling the  
masterRd task?





# Modports: Importing Tasks From an Interface

```
interface simple_bus(input bit clk);  
    logic req,gnt;  
    logic [7:0] addr,data;  
    logic [1:0] mode;  
    logic start,rdy;  
  
    modport slave(input req,addr,mode,start,clk,  
                  output gnt,rdy,  
                  inout data,  
                  import task slaveRd(),  
                          task slaveWr());  
  
    modport master(input gnt,rdy,clk,  
                   output req,addr,  
                           mode,start,  
                   inout data,  
                   import task masterRd(input logic[7:0] raddr),  
                           task masterWr(input logic[7:0] waddr));  
  
    ...  
endinterface
```

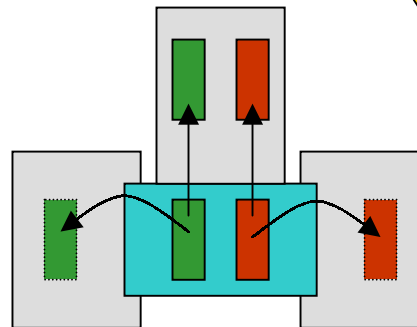
import into  
module that uses  
the modport

Modules using  
**master** modport  
can only call these  
tasks

# Modports: Using Imported Tasks

```
module memMod(interface a);  
  logic avail;  
  
  always @(posedge a.clk)  
    a.gnt <= a.req & avail;  
  
  always @(a.start)  
    if(a.mode[0] == 1'b0)  
      a.slaveRead;  
    else  
      a.slaveWrite;  
endmodule  
  
module cpuMod(interface b);  
  enum {read,write} instr;  
  logic [7:0] raddr;  
  ...  
  always @(posedge b.clk)  
    if(instr == read)  
      b.masterRead(raddr);  
      ...  
    else  
      b.masterWrite(raddr);  
endmodule
```

```
module omniMod(interface b);  
  //...  
endmodule:omniMod  
  
module top;  
  logic clk = 0;  
  
  simple_bus sb_intf(clk)  
  
  memMod mem(sb_intf.slave);  
  cpuMod cpu(sb_intf.master);  
  omniMod omni(sb_intf);  
endmodule
```



Only has access to  
slaveRd/slaveWr tasks

Only has access to  
masterRd/Wr tasks  
Has access to all  
tasks and signals



# More Reuse Issues

---

- Tasks Reside in the Interface Definition
  - Tasks are called from modules connected to interface
  - If module changes, interface may have to change accordingly
- Why Not Have Tasks Reside in Modules?
  - Need to be able to call tasks as interface methods from the other module connected
  - “Export” module tasks to the interface
- Other Modules See Exported Tasks As Regular Interface Methods
  - Changing one module connected to the interface can also change the method
  - The interface and the other module(s) connected via the interface don't have to change

# Modports: Exporting Tasks To an Interface

```
interface simple_bus(input bit clk);  
    logic req,gnt;  
    logic [7:0] addr,data;  
    logic [1:0] mode;  
    logic start,rdy;  
  
    modport slave(input req,addr,mode,start,clk,  
                  output gnt,rdy,  
                  inout data,  
                  export task Read(input logic[7:0] raddr),  
                  task Write(input logic[7:0] waddr));  
  
    modport master(input gnt,rdy,clk,  
                   output req,addr,  
                   mode,start,  
                   inout data,  
                   import task Read(input logic[7:0] raddr),  
                   task Write(input logic[7:0] waddr));  
  
    ...  
endinterface
```

export from  
module that uses  
**slave** modport

Modules using  
**master** modport  
can call exported  
tasks from **slave**

# Using Exported/Imported Tasks

```

module memMod(interface a);
logic avail;

    task a.Read(input logic raddr);
        avail = 0;
        ...
        avail = 1;
    endtask

    task a.Write(input logic[7:0] waddr);
        avail = 0;
        ...
        avail = 1;
    endtask
endmodule

module cpuMod(interface b);
    enum {read,write} instr;
    logic [7:0] raddr;
    ...
    always @(posedge b.clk)
        if(instr == read)
            b.Read(raddr);
        ...
        else
            b.Write(raddr);
    endmodule

```

Task definition  
indicates export to  
interface

Use Task as if it  
were defined in  
the Interface

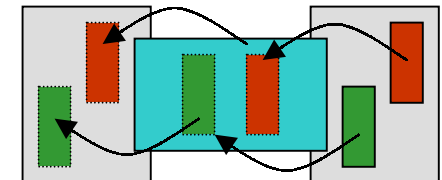
```

module top;
    logic clk = 0;

    simple_bus sb_intf(clk);

    memMod mem(sb_intf.slave);
    cpuMod cpu(sb_intf.master);
endmodule

```



# Restricting Access Using Modports

```
interface bus_i;
  bit sig1, sig2;
  logic [63:0] bus;

  task automatic singleRd(...);
  endtask
  task automatic blockRd(...);
  endtask
  task automatic singleWr(...);
  endtask
  task automatic blockWr(...);
  endtask

  modport fullslot(export task singleRd(...),
                  blockRd(...),
                  singleWr(...),
                  blockWr(...),
                  halfslot(export task singleRd(...),
                          singleWr(...)));
endinterface
```

```
bus_I PCImicro;

master M1(PCImicro);
dsp1    S1(PCImicro.fullslot);
dsp2    S2(PCImicro.halfslot);
```

**Block read/writes are  
not available to dsp2**

**No signals,  
just  
methods in  
modport**

# Encapsulating Communication

## Parallel Interface

```
interface parallel(input bit clk):

    logic [31:0] data_bus;
    logic data_valid=0;

    task write(input data_type d);
        data_bus <= d;
        data_valid <= 1;
        @(posedge clk) data_bus <= 'z;
        data_valid <= 0;
    endtask

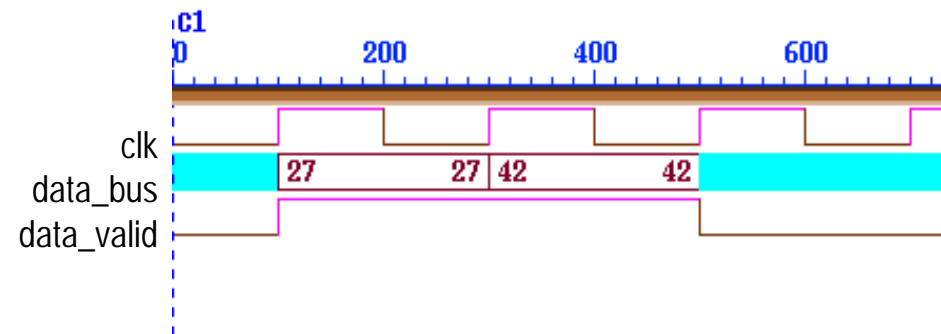
    task read(output data_type d);
        while (data_valid != 1)
            @(posedge clk);
        d = data_bus;
        @(posedge clk);
    endtask

endinterface
```

**Interface port**

**Member signals**

**read and write  
tasks/methods**



- **An interface task/function is a communication method**

# Alternate Interface

## Serial Interface

```
interface serial(input bit clk);

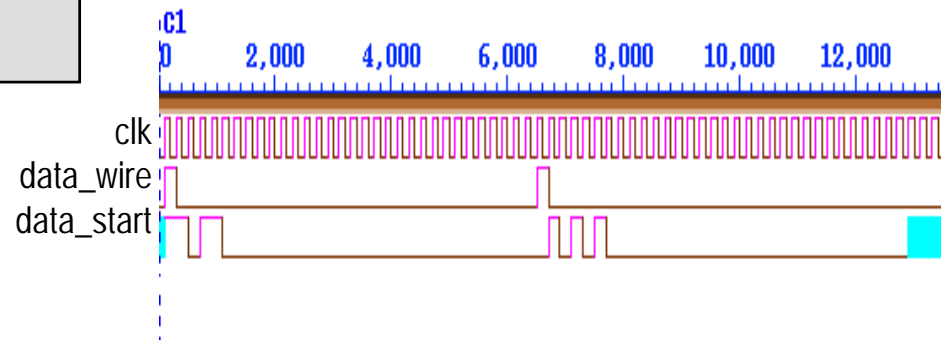
  logic data_wire;
  logic data_start=0;

  task write(input data_type d);
    for (int i = 0; i <= 31; i++)
      begin
        if (i==0) data_start <= 1;
        else data_start <= 0;
        data_wire = d[i];
        @(posedge clk) data_wire = 'x;
      end
  endtask
```

**serial signals**

```
task read(output data_type d);
  while (data_start != 1)
    @(negedge clk);
  for (int i = 0; i <= 31; i++)
    begin
      d[i] <= data_wire;
      @(negedge clk) ;
    end
endtask

endinterface
```





# Using Different Interfaces

```
typedef logic [31:0]  
data_type;  
  
bit clk;  
always #100 clk = !clk;  
  
parallel channel(clk);  
send      s(clk, channel);  
receive r(clk, channel);
```

```
typedef logic [31:0]  
data_type;  
  
bit clk;  
always #100 clk = !clk;  
  
serial channel(clk);  
send      s(clk, channel);  
receive r(clk, channel);
```

- The interface can be used to encapsulate all the communication

```
module send(input bit clk,  
            interface i);  
    data_type d;  
    ...  
    i.write(d);  
endmodule
```

**Module inherits  
communication  
method from  
interface**

# Interface Parameters and Templates

```
interface AD;
parameter size=8;
parameter type ADtype=int;

ADtype Fifo[0:$];
initial repeat (size) Fifo = {0,Fifo};

function ADtype Get();
...
endfunction
function void Put(ADtype data);
...
endfunction
endinterface

AD #(.size(24)) intAD;
AD #(.ADtype(real)) realAD;
```

Type can be overridden

Also works for  
Modules

Note explicit  
parameter  
names

- intAD size is 24
- realAD Fifo is of type real

# Interfaces Are Objects

```
class queue {  
public:  
    qtype pop();  
    int num();  
    void push(qtype *obj);  
private:  
    ....  
}
```

← C++

**An Interface fulfills a purpose similar to a class in C++.**

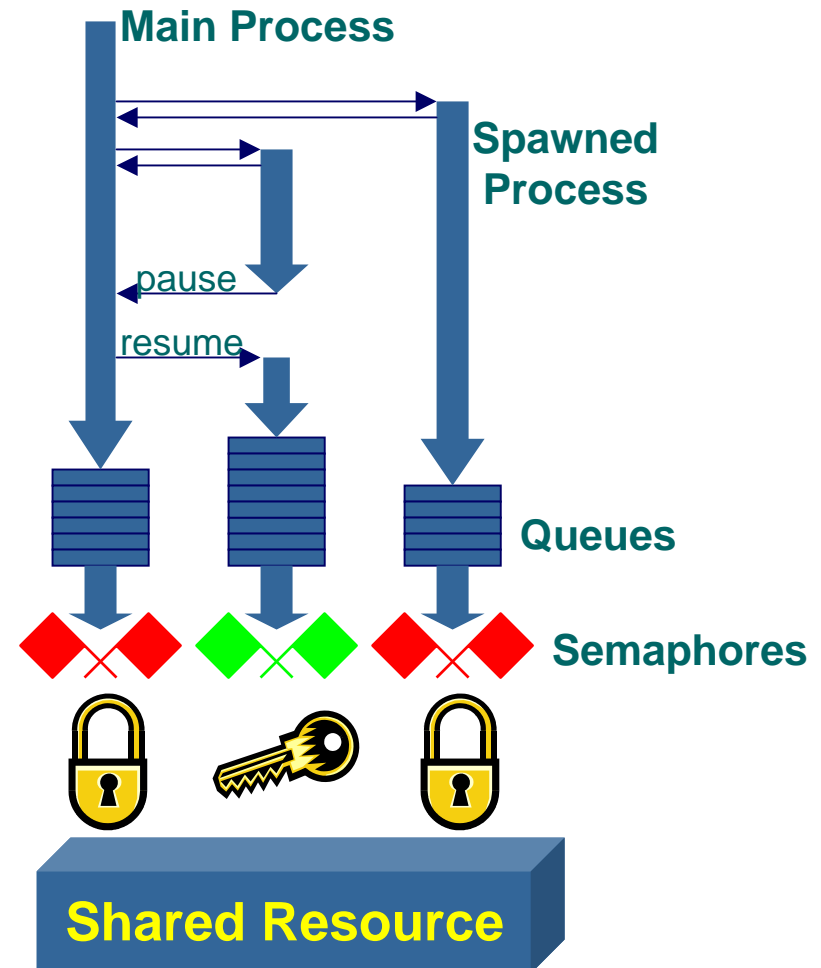
**C++ is an object oriented language that has been developed for efficient software design. SUPERLOG has been developed for systems design.**

```
interface queue;  
  
function qtype pop();  
    qtype r;  
    r = q[0]; q = q[1:$];  
    return(r)  
endfunction  
  
int function num();  
    return (q.$num);  
endfunction  
  
task push(qtype obj);  
    q = {q, obj};  
endtask  
  
qtype q[0:$];  
endinterface
```

- **Encapsulates tasks and functions like a C++ class**

# Managing Shared Resources

- Multiple Operations Use the Same Interface
- Multiple Parallel Operations Must Share
  - If Host port is doing a transaction, others must wait
  - Individual requestors should block (i.e. wait) until port is available
- Spawn Processes
  - “non-blocking” process statement
  - Pause and resume
  - Built-in Process IDs
  - **Not in SystemVerilog**
- Semaphores/Mailboxes
  - Synchronize processes
  - Coordinate shared resources





# Semaphores

---

- Interfaces between blocks represent a single physical connection
  - Only one transaction can occur at a time
    - For split transactions, multiple transactions may be “in-flight” simultaneously
    - Only one phase of the split transaction can occur at a time
  - If the Interface is busy, other requestors must wait
- Semaphores can be used to arbitrate for the communication path
- Semaphores can also be used within a block
- Interfaces Instantiated as Static Data Objects
  - Similar to SystemC “Portless Channel Access”

# Semaphore Example in SUPERLOG

```
interface semaphore;
parameter MAX = 1;
int status = 0;
$ref_process_inst q[0:$];
$ref_process_inst next;
```

Built-in SUPERLOG Process  
Management datatype

```
task lock ();
```

```
  if (status >= MAX)
```

If resource is busy

```
  begin
```

```
    q = {q, $this_process};
```

Add this process to the list

```
    $pause;
```

Pause this process

```
  end
```

```
  status++;
```

Mark resource busy and return

```
endtask
```

```
task unlock ();
```

```
  status--;
```

Free-up resource

```
  if (q.$num > 0)
```

If there are suspended processes

```
  begin
```

```
    next = q[0];
```

Pop next waiting process from list

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

```
    $resume(next);
```

Resume waiting process

```
  end
```

```
endtask
```

```
endinterface: semaphore
```

# Using Semaphores

```
semaphore sem1;
semaphore #(MAX(2)) sem2;

initial begin
    sem1.lock;
    sem2.lock;
    do_task1(...);
    sem1.unlock;
    sem2.unlock;
end

initial begin
    sem2.lock;
    sem1.lock;
    do_task2(...);
end
```

Instantiate sem1 semaphore object

sem2 supports 2 concurrent operations

Request access to sem1 semaphore Suspend if busy

Release the semaphore

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

**March 11 - 12, 2002**



## Communication-Based Verification

---

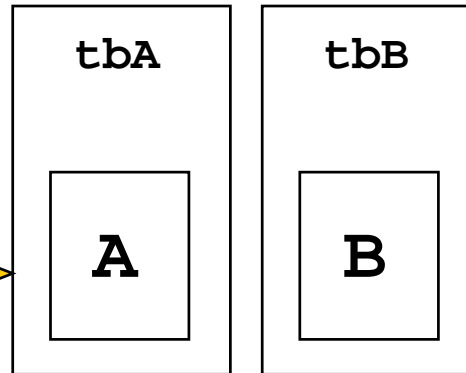




# Conventional Verification Strategy

Pre-Integration

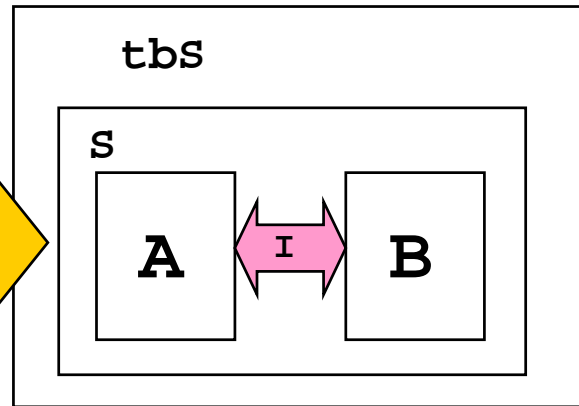
Test Subblocks  
in isolation



**Testbench reuse problems  
tbA and tbB separate**

Post-Integration

**Only want to  
test  
interconnecte  
d structure.  
Have to test  
that all signals  
are connected  
correctly**

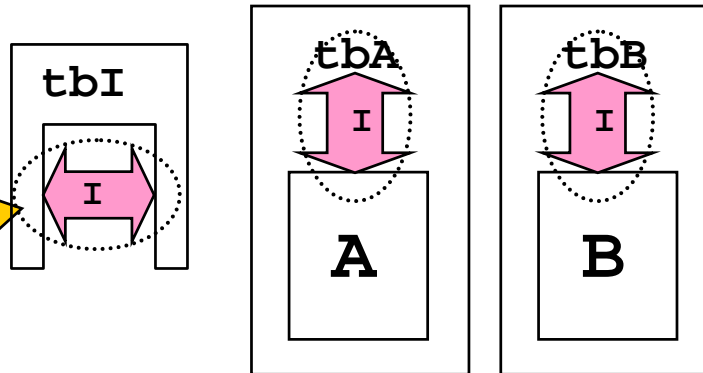


**Complex interconnect  
Hard to create tests to  
check all signals  
Slow, runs whole  
design even if only  
structure is tested**

# SUPERLOG Verification Strategy

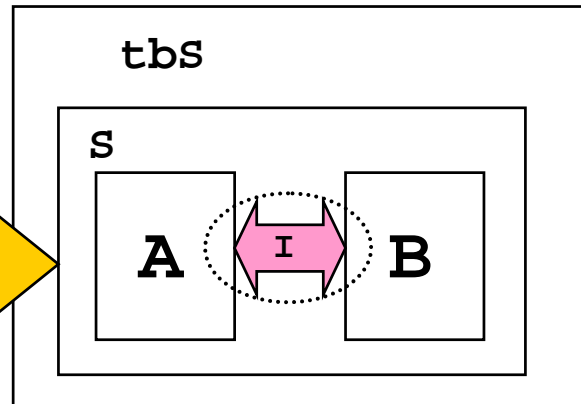
Pre-Integration

**Test  
interfaces in  
isolation**



Post-Integration

**Protocol/  
interconnect bugs  
already flushed  
out**



**Interfaces provide  
reusable components**

**tbA and tbB are  
'linked'**

**Interface is  
executable spec**

**Wiring up is simple  
and not error prone**

**Interfaces can  
contain protocol  
checkers and  
coverage counters**

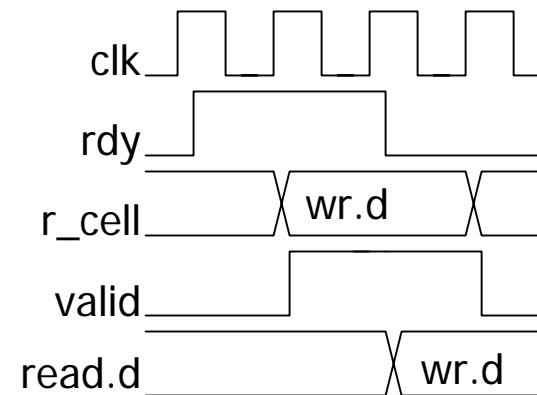
# Communication-Based Verification in Action

```
interface handshake_i(input bit clk);
    r_atm_cell_t      r_cell;
    logic             valid;
    logic             rdy;

    task write(input r_atm_cell_t d);
        @(posedge clk iff (rdy == 1)) begin
            r_cell <= d;
            valid <= 1;
        end
        @(posedge clk iff (rdy == 0)) begin
            r_cell <= 'x;
            valid <= 0;
        end
    endtask

    task read(output r_atm_cell_t d);
        ready <= 1;
        @(posedge clk iff (valid == 1)) begin
            d <= r_cell;
            rdy <= 0;
        end
        @(posedge clk iff (valid == 0)) ;
    endtask
endinterface
```

## Protocol



# Communication-Based Verification in Action: The Test

```
module top;
  bit clk = 0;
  always #100 clk = !clk;
```

```
  handshake_i hs(clk);
  test t(hs,hs);
endmodule
```

**Test connects  
interface to itself**

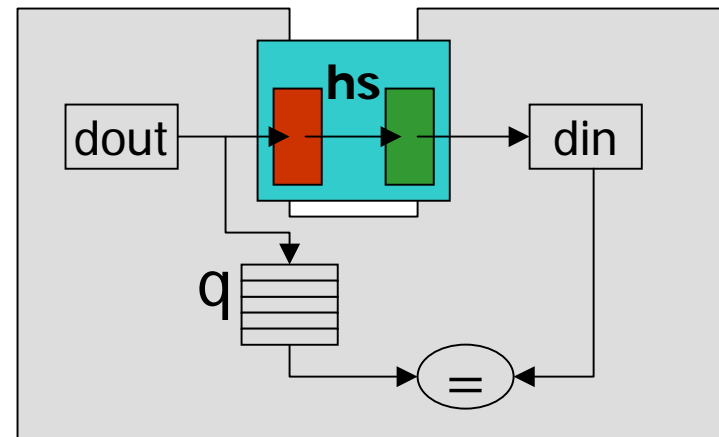
```
module test (interface in,out);
  r_atm_cell_t dout, din;
  r_atm_cell_t q[0:$];
  initial begin
    repeat(94) begin
      time t = $rand(.ranges({{1,10}}));
      #(t) dout = $rand();
      out.write(dout);
      q = {q, dout};
    end
    $display("Found %d Errors", errors);
    $finish;
  end
```

**Push expected  
results onto q**

```
always begin
  time t = $rand(.ranges({{1,10}}));
  #(t) in.read(din);
  if (q[0] != din) begin
    errors++;
    $display("Rcv=%h Exp",
            din, q[0]);
  end
  q = q[1:$];
  $display("Rcv=%h", din[31:0]);
end
endmodule
```

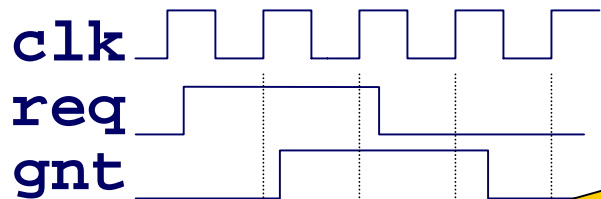
**pop q**

**Compare to  
expected value**



**We now know that the  
handshake\_i Interface is correct**

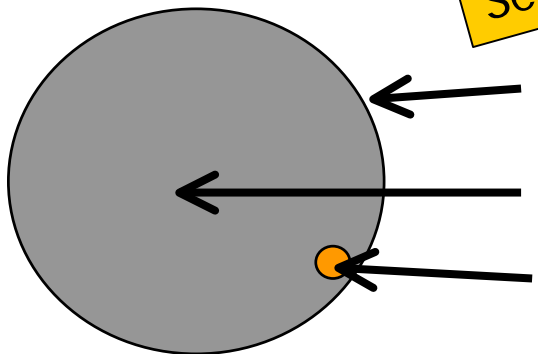
# Assertion Power



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

# Encapsulating Verification in Interfaces

- Interfaces Encapsulate the Connectivity and Protocol Methods for Inter-block Communication
- The Interface Defines Correct Behavior for Communication
- Check the Protocol Automatically in the Interface
  - Interfaces include standalone processes
  - SUPERLOG/SystemVerilog *assert* construct simplifies checking
- Automatically Track Coverage Too

```
interface foo_i(input bit clk);  
...  
task write(...);  
...  
endtask  
task read(...);  
...  
endtask  
  
always @(posedge clk iff rdy)  
    assert @(posedge clk)  
        (rdy == 0);  
  
always @(posedge clk)  
    assert (!(req1 && req2));  
  
always @(posedge clk  
        iff (as == 0))  
    count[mode]++;  
endinterface
```

When rdy goes high

check that it will be

check that only one req will be high at a time

Count number of cycles of each mode

# Using Encapsulated Coverage in a Test

```
module test;  
  bit clk = 0, done = 0;  
  bit [1:0] mode;  
  foo_i foo(clk);  
  dut dut(foo);
```

Instantiate Interface foo

Connect it to dut

```
  initial begin  
    repeat(10) begin:rp10  
      if(!done)  
        repeat(100) begin:rp100  
          mode = $rand();  
          do_test(mode);  
        end:rp100  
      for(i = 0; i<3; i++)  
        if(foo.count[i] > 10)  
          done = 1;  
        else  
          done = 0;  
      end:rp10  
      if(!done)  
        $display("ERROR");  
    end  
  endmodule
```

While coverage criteria have not been met

Randomly choose test mode

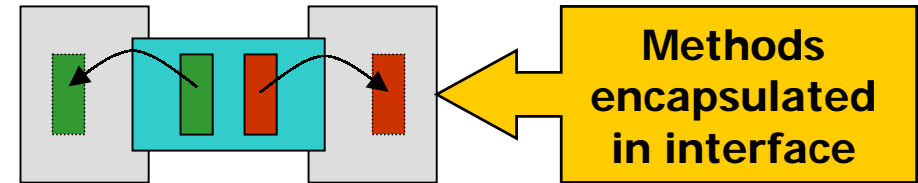
Execute test task

Check Coverage Results  
(from previous slide)

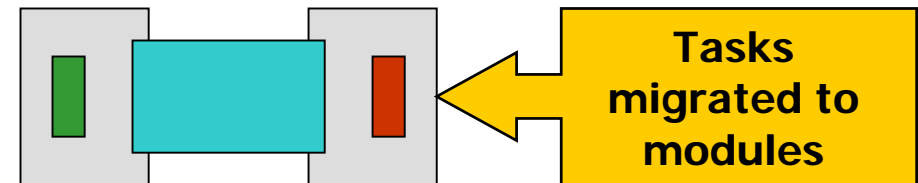
If coverage not met, keep running

# Abstract to Implementation

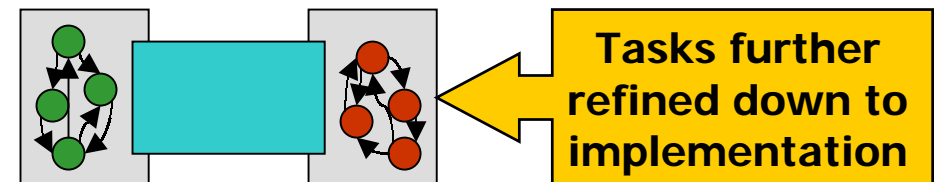
- The Interface Encapsulates the Communication
  - Why Bother to Check the Protocol?
  - Implementation-level blocks may use the interconnect but not the methods
- Implementing the Checking is Easier to do When You're Already Implementing the Protocol
  - Assertions capture the protocol more easily



Becomes



Becomes





**March 11 - 12, 2002**



# Communication-Based Design Exploration

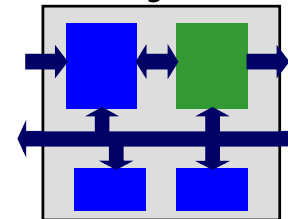
---



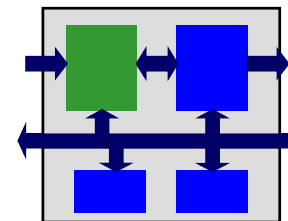
# Using Abstract Interfaces

- Interfaces Encapsulate Data and How Data Move Between Blocks
- Blocks Operate on Data Once Received
- Design Exploration is All About Looking at Alternatives
  - Focus Simulation Cycles Looking at Relevant Details
  - Keep Other Blocks at High-Level
- Interfaces Should Support Multiple Layers of Abstraction for both "Send" and "Receive"

Must be able to move easily from this:

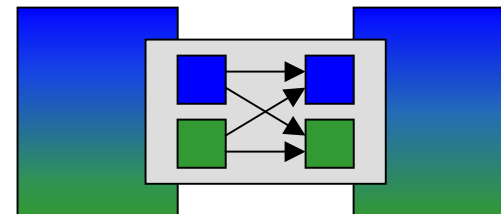


to this:



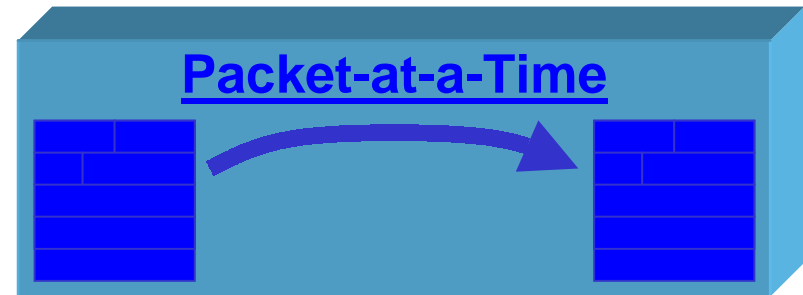
**RTL  
Block**

**Abstract  
Blocks**



# Polymorphic Interfaces

- Fast Interfaces abstract away details
- Interface still looks the same to its users
- Don't waste simulation cycles on details you don't care about



# Interface Polymorphism

```
interface abstr;
pkt_t pkt;

task send(input pkt_t ipkt);
    pkt <= ipkt;
    valid <= 1;
endtask

task rcv(output pkt_t opkt);
    @(valid) opkt <= pkt;
endtask
endinterface
```

```
module snd(interface I,
pkt_t pkt;
...
I.send(pkt);
endmodule
```

```
module rcv(interface I);
pkt_t pkt;
...
I.rcv(pkt);
endmodule
```

Use  
generic  
interface

Inherit  
method

```
interface RTL(input bit clk);
logic[7:0] dbus;

task send(input pkt_t ipkt);
    for(i=0; i<len; i++)
        @(posedge clk) begin
            valid <= 1;
            dbus <= pkt.a[i];
        end
    done <= 1;
endtask
```

```
task rcv(output pkt_t opkt);
    wait(valid);
    for(i=0; i<len; i++)
        @(posedge clk)
            opkt.a[i] <= dbus;
endtask
endinterface
```

module top;	module top;
<b>abstr I;</b>	<b>RTL I;</b>
snd s(I);	snd s(I);
rcv r(I);	rcv r(I);
endmodule	endmodule

Only  
Change

# Mixing Abstraction Levels

```
interface MixedA2R(input bit clk);  
pkt_t pkt;  
  
task send(input pkt_t ipkt);  
    pkt <= ipkt;  
    valid <= 1;  
endtask  
  
task rcv(output pkt_t opkt);  
    wait(valid);  
    for(i=0;i<len;i++)  
        @(posedge clk)  
            opkt.a[i] <= pkt.a[i];  
endtask  
endinterface
```

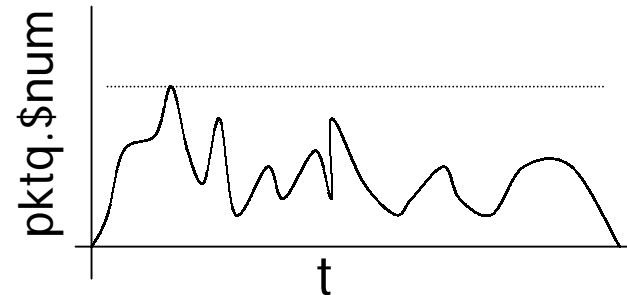
```
interface MixedR2A(input bit clk);  
pkt_t pkt;  
  
task send(input pkt_t ipkt);  
    for(i=0; i<len; i++)  
        @(posedge clk) begin  
            valid <= 1;  
            pkt.a[i] <= ipkt.a[i];  
            done <= 1;  
        endtask  
  
task rcv(output pkt_t opkt);  
    @(done) opkt <= pkt;  
endtask  
endinterface
```

- Notice that all send/rcv tasks have the same arguments
- Focus on what the rcv module does with the opkt
  - If rcv module operates on a pkt, it may not matter how it gets there
- Could insert dummy clk cycles in interface if timing accuracy is required
- Using abstract methods removes simulation detail, making simulation more efficient

# Asynchronous Interfaces

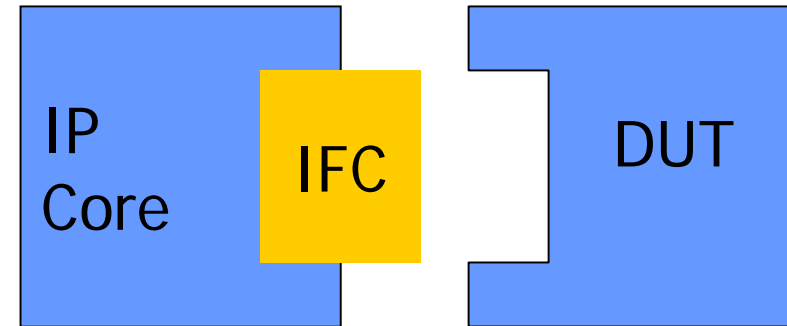
- Build Flexibility Into the Interface to Handle Different Data Rates or Abstractions
- Ideal Application for SUPERLOG Queues
  - Queue automatically grows/shrinks as data is added or removed
  - Statistics can be gathered to develop concrete FIFO details for implementation

```
interface asyn(input bit clk);  
pkt_t pktq[0:$];  
  
task send(input pkt_t ipkt);  
    pktq = {pktq,ipkt};  
endtask  
  
task rcv(output pkt_t opkt);  
    wait(pktq.$num);  
    for(i=0;i<len;i++)  
        @(posedge clk)  
            opkt.a[i] <= pktq[0].a[i];  
endtask  
endinterface
```



# IP Verification

- Using Interfaces Guarantees that IP Core is Connected Correctly
- Interface has “public” and “private” data via modports, 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;
...

always @(posedge clk)
    assert (!(rd && wr));

always @(posedge clk)
    if(rdy == 0)
        assert @(posedge clk) (rdy == 1);
...
endinterface
```

**March 11 - 12, 2002**

## Using SUPERLOG to Build a System-Level Platform

---





# System-Level Platforms

- SystemVerilog Provides Powerful Abstraction for HW Design
- Architectural and System-Level Exploration Requires Higher-Level Abstraction Available in SUPERLOG
  - Queues
  - Pointers
  - Ease of C/C++ Code Integration
- Still Requires a Smooth Migration From Architectural-Level to RTL/Implementation
  - SUPERLOG includes all of SystemVerilog
  - All of the Interface capabilities discussed extend up to higher levels of abstraction in SUPERLOG

# CBlend – Adding C/C++

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");
    }
}
```

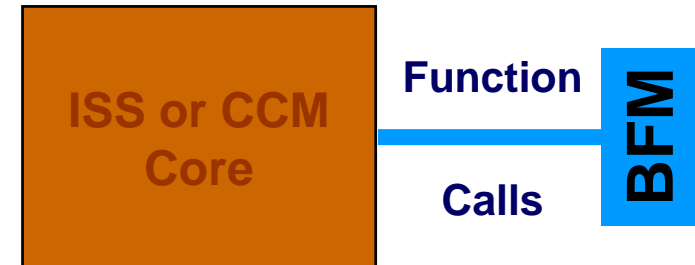
### SUPERLOG Code

```
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
```

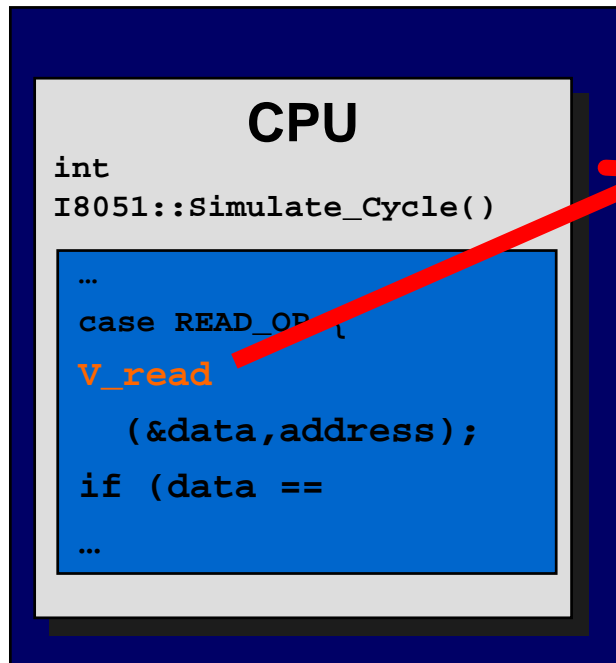
**Time Delay!**

# C Model Drives HDL BFM Directly

**Unique Capability Provides  
Simple Integration Setup**



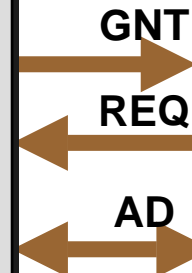
**i8051.cc**



**i8051.v BFM**

```

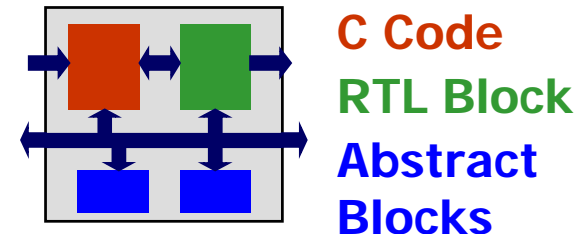
task V_read;
output data;
input address;
begin
    REQ = 0;
    wait(GNT)
    #10
    AD = address;
...
  
```



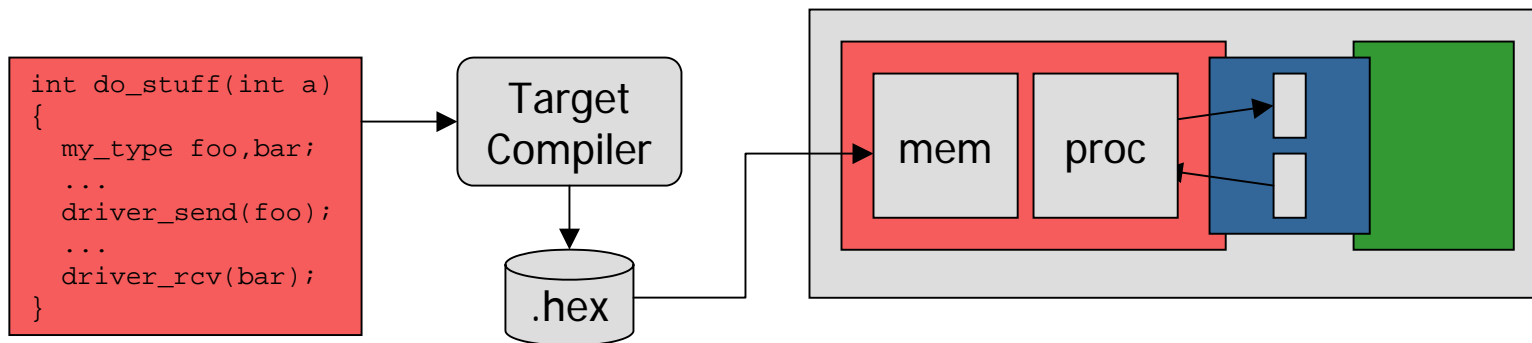
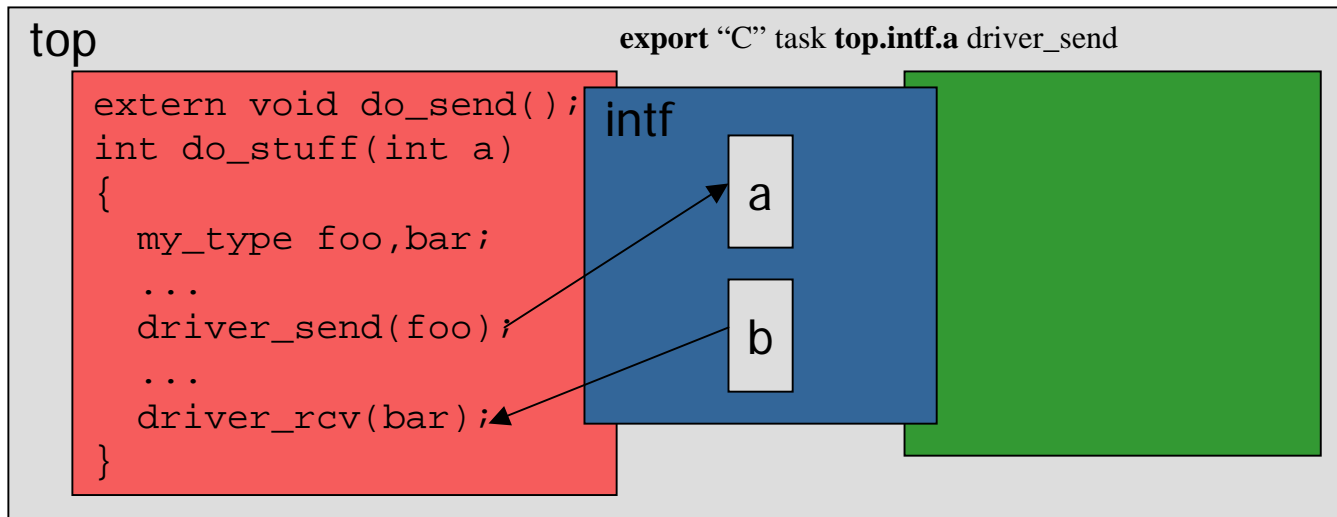
# Facilitating C/C++ Exploration

- Multiple Targets for Executing C Application Code
  - General Purpose Processor Core
  - Application-Specific Core
- Flush Out Algorithm First
  - Define Interface
  - Initially run C code directly on host platform
  - Use CBlend to call interface methods
- Compile Application Code to Target Processor
  - Drop-in core model and update interface

Must be able to do this:



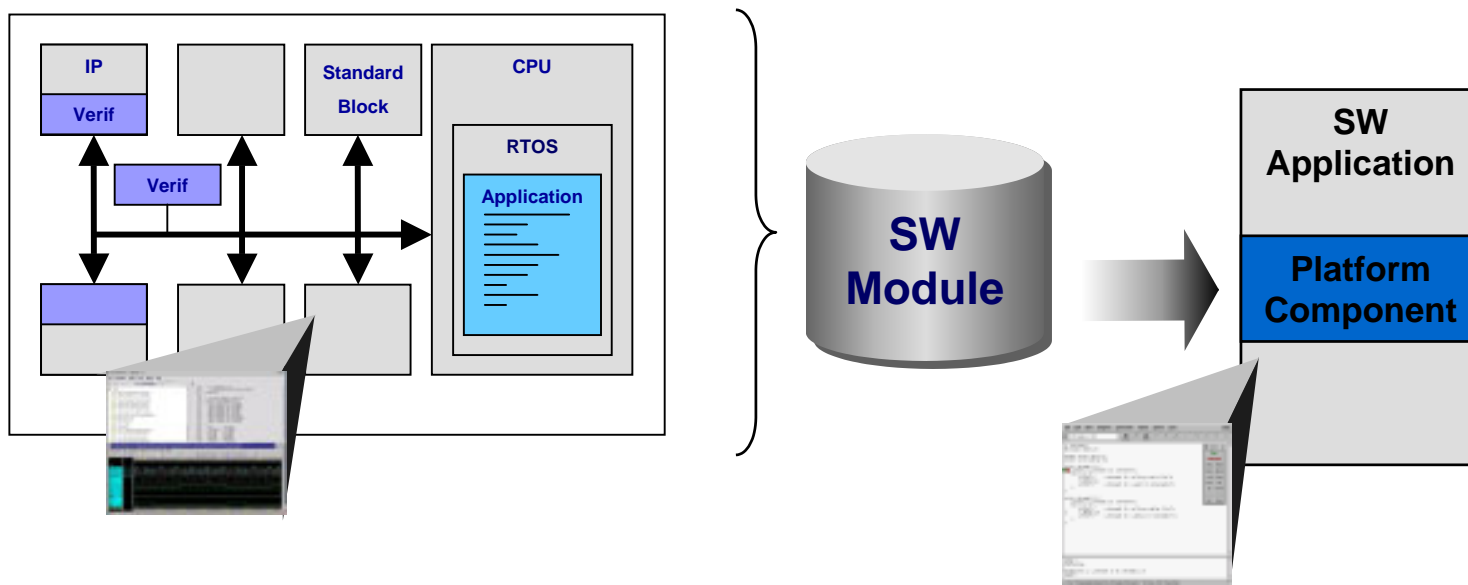
# C/C++ Application to Implementation



# Embedded SW Solution: Make Simulation Part Of SW App

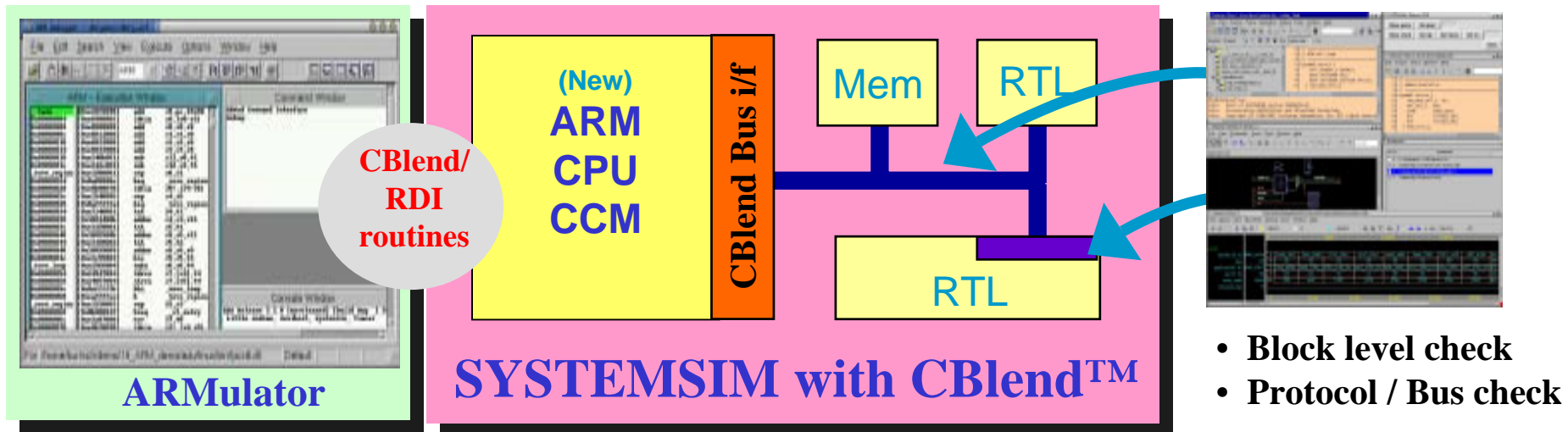
## For The First Time, Platform Can Be Verified Pre-Prototype

- 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

March 11 - 12, 2002



## Wrap-Up

---







# Interfaces and Platforms

---

- Interfaces Enable a Divide-and-Conquer Methodology
  - Separate the “What” from the “How”
  - Encapsulate all connectivity and protocol
  - Embed protocol checking and coverage
  - Model the communication paths between blocks
- Interfaces Promote Reuse
  - Library of interfaces enable users to choose the right one
  - Encapsulation in libraries minimizes changes to parent
  - Multiple levels of abstraction in interfaces allow exploration of tradeoffs between different implementation choices of each function

# SUPERLOG and Platforms

- Abstract Modelling Allows Exploration of Function Independent from Implementation
  - Abstract modelling in either SUPERLOG or C/C++ via CBlend
- Interface Encapsulation Facilitates Transition from Abstract to Implementation (RTL)
  - Allows abstract and RTL models to interact easily
  - Makes it easy to swap between different implementation models for given blocks
- Methodology Focuses on Communication Between Blocks
  - Functional Blocks operate on data once they get it
  - Interfaces define how data moves between blocks

# SystemVerilog and SUPERLOG Are REAL

- SYSTEMSIM is THE **SystemVerilog** Simulator
  - Customers using SYSTEMSIM for 2 years
  - Over 20 customers
    - One customer reduced 100 pages of Verilog RTL to 6 pages
    - Large processor company: "We could not even express our next-generation processor using Verilog RTL."
- SYSTEMSIM is THE **SUPERLOG** Simulator
  - Supports CBlend for HW/SW Co-Verification in a single process
  - Supports System-Level Modeling
  - Includes Verification features for writing complex testbenches
    - Constrained Random Data Generation
    - Functional Coverage
    - Object Orientation

# Co-Design Automation Products

*Testbench, software, IP blocks,  
high level models, algorithms, design*

*design*

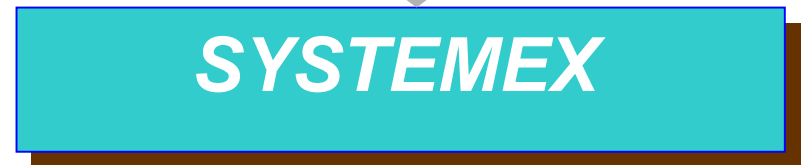
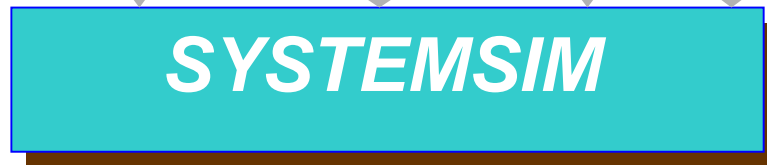
Verilog 90/95/2000

SUPERLOG

C/C++/SystemC

SystemVerilog

SystemVerilog



Verilog RTL  
**SYNTHESIS**



# SystemEx 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

# SUPERLOG Practical Design Subset

Fast, Abstract Code Can Now Be Synthesized!

**SystemVerilog**  
SUPERLOG Synthesizable Subset

