Agenda

Introduction:
SystemVerilog Motivation
Vassilios Gerousis, Infineon Technologies
Accellera Technical Committee Chair

Session 1:
SystemVerilog for Design

Language Tutorial
Johny Srouji, Intel

User Experience
Matt Maidment, Intel

Session 2:
SystemVerilog for Verification

Language Tutorial
Tom Fitzpatrick, Synopsys

User Experience
Faisal Haque, Verification Central

Lunch: 12:15 – 1:00pm

Session 3: SystemVerilog Assertions

Language Tutorial
Bassam Tabbara, Novas Software

Tecnhology and User Experience
Alon Flaisher, Intel

Using SystemVerilog Assertions and Testbench Together
Jon Michelson, Verification Central

Session 4: SystemVerilog APIs
Doug Warmke, Model Technology

Session 5: SystemVerilog Momentum

Verilog2001 to SystemVerilog
Stuart Sutherland, Sutherland HDL

SystemVerilog Industry Support
Vassilios Gerousis, Infineon

End: 5:00pm

SystemVerilog 3.1

Design Subset

Johny Srouji

Intel

Chair – SV-Basic Committee

Presentation Outline

- Data Types
- Structures & Unions
- Literals
- Enumerated Data Types
- Constants & Parameters
- Scope & Lifetime
- Interfaces

Basic SV3.1 Data Types

```
reg r;      // 4-state Verilog-2001 single-bit datatype
integer i; // 4-state Verilog-2001 >= 32-bit datatype
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
int i;      // 2-state, 32-bit signed integer
shortint s;// 2-state, 16-bit signed integer
longint l; // 2-state, 64-bit signed integer
```

Make your own types using typedef
Use typedef to get C compatibility

```
typedef     shortint        short;
typedef     longint         longlong;
typedef     real            double;
typedef     shortreal       float;
```

2 State and 4 State Data Types

Verilog
SystemVerilog

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

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

SystemVerilog

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

Equivalent to these 4-valued SystemVerilog types

SystemVerilog

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

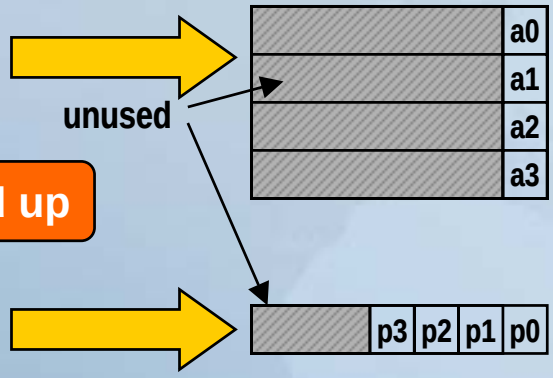
These SystemVerilog types have two-valued bits (0 and 1)

If you don't need the X and Z values then use the SystemVerilog bit and int types which MAKE EXECUTION FASTER

Packed And Unpacked Arrays

unpacked array of bits

```
bit a [3:0];
```



Don't get them mixed up

packed array of bits

```
bit [3:0] p;
```

1k 16 bit unpacked memory

```
bit [15:0] memory [1023:0];
memory[i] = ~memory[i];
memory[i] [15:8] = 0;
```

Packed indexes can be sliced

1k 16 bit packed memory

```
bit [15:0] [1023:0] Frame;
always @inv Frame = ~Frame;
```

Can operate on entire memory

Structures

```
struct { bit [7:0]     opcode;
         bit [23:0]    addr;
       } IR;           // anonymous structure
```

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

```
typedef struct { bit [7:0]    opcode;
                 bit [23:0]   addr;
} instruction;          // named structure type

instruction IR;         // define variable

IR.opcode = 1;          // set field in IR
```

Unions

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

u_type u;

initial
  begin
    u.n = 27;
    $display("n=%d", u.n);
    u.f = 3.1415;
    $display("f=%f",u.f);
    $finish(0);
  end
```

union

provide storage for either **int** or **real**

int

real

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

Packed Structures

Represents bit or part selects of vectors

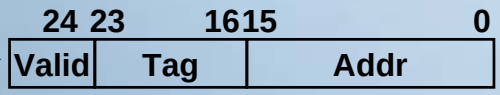
```
struct packed {
  bit        Valid;
  byte       Tag;
  bit [15:0] Addr;
} Entry;
iTag   = Entry.Tag;
iAddr  = Entry.Addr;
iValid = Entry.Valid
```

```
reg [24:0] Entry;
`define Valid 24
`define Tag 23:16
`define Addr 15:0
iTag   = Entry[`Tag];
iAddr  = Entry[`Addr];
iValid = Entry[`Valid]
```

packed struct may contain other packed structs or packed arrays

unpacked struct

packed struct



SV 3.1 Literals

- SV literal values are extensions of those for Verilog

```
reg [31:0] a,b;
reg [15:0] c,d;
...
a = 32'hf0ab;
c = 16'hFFFF
```

This works like in Verilog

Adds the ability to specify unsized literal single bit values with a preceding ‘

```
a = '0;
b = '1;
c = 'x;
d = 'z;
```

This fills the packed array with the same bit value

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

These are equivalent

SV 3.1 Literals

This works like in Verilog

Adds time literals

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

You can also specify delays with explicit units

Similar to C, but with the replication operator (**{{}}**) allowed

Adds Array literals

```
int n[1:2][1:3] = {{0,1,2},{3{4}}}
```

Enumerated Data Types

```
enum {red, yellow, green} light1, light2;
```

anonymous int type

```
enum {bronze=3, silver, gold} medal;
```

silver=4, gold=5

```
enum {a=0, b=7, c, d=8} alphabet;
```

Syntax error

```
enum {bronze=4'h3, silver, gold} medal;
```

silver=4'h4, gold=4'h5

```
typedef enum {red, green, blue, yellow, white, black} Colors;

Colors   col;
integer  a, b;

a = blue * 3;
col = yellow;
b = col + green;
```

a=2*3=6
col=3
b=3+1=4

Type Casting

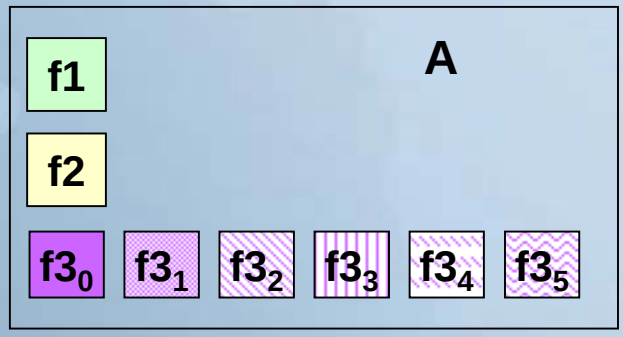
```
int'(2.0 * 3.0)
shortint'{8'hFA, 8'hCE}
17 '(x - 2)
```

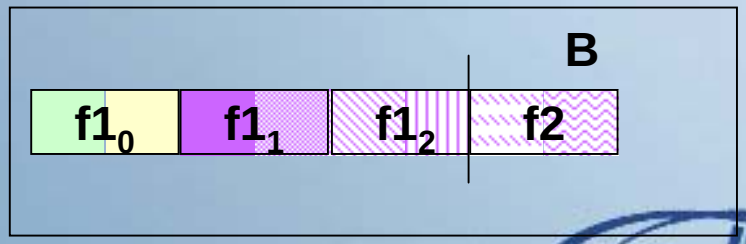
A data type can be changed by using a cast (‘) operation

- Any aggregate bit-level object can be reshaped
  - Packed ⇔ Unpacked, Array ⇔ Structure

Objects must have identical bit size

```
typedef struct {
        bit [7:0] f1;
        bit [7:0] f2;
        bit [7:0] f3[0:5];
} Unpacked_s;
typedef struct packed {
        bit [15:0][0:2] f1;
        bit [15:0] f2;
} Packed_s;
Unpacked_s A;
Packed_s B;
…
        A = Unpacked_s'(B);
        B = Packed_s'(A);
```





Constants

- Use like defines or parameters
- Global constant (const) is resolved at the END of elaboration.
- Local constant (localparam) is resolved at the BEGINNING of elaboration.
  - No order dependency problems when compiling
- specparam is used for specify blocks

global constant

Can contain an expression with any hierarchical path name

local constant

```
const bit TRUE = 1;
const logic option = a.b.c;

module top;
localparam int TRUE = -1;
localparam bit FALSE = 0;

initial
  $display("TRUE=%d", TRUE); // -1
endmodule
```

Can't be overridden with defparam

Parameters

```
module top;
logic clk;

clockgen #(.start_value(1'b1), .delay(50),
             .ctype(int)) c (clk);

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

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

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

initial clk <= start_value;

always #delay clk <= !clk;

endmodule
```

Override parameters by name

Parameter used before definition

Parameters can have explicit type

Variable Types

- **Static** variables
  - Allocated and initialized at time 0
  - Exist for the entire simulation
- **Automatic** variables
  - Enable recursive tasks and functions
  - Reallocated and initialized each time entering a block
  - May not be used to trigger an event
- **Global** variables
  - Defined outside of any module (i.e. in $root)
  - Accessible from any scope
  - Must be static
  - Tasks and functions can be global too
- **Local** variables
  - Accessible at the scope where they are defined and below
  - Default to static, can made automatic
  - Accessible from outside the scope with a hierarchical pathname

Scope and Lifetime

data declared outside of modules is static and global

i is automatic and local to that block

```
top inst;

int max = 10;
int n;
module top;
 int n;
 initial begin
    automatic int i;
    n = 1;
    for (i=2; i<=max; i++)
       n *= i;
  end
initial begin : myblock
     n = 1;
     for (int i=2; i<=max; i++)
       n *= i;
     $root.n = n;
  end
endmodule
```

data declared inside of a module is static and available to all tasks and functions in that module

global **n**

local **n**

Task and Function Arguments

- Default Arguments
  - Definition: `task foo(int j=5, int k=8);`
  - Usage: `foo(); foo(5); foo(,8); foo(5,8);`
- Pass by Name

  `foo(.k(22)); // j uses default`
- Pass by Reference
  - Declaration: `task tk([const] ref int[1000:1] ar);`
  - Usage: `tk(my_array); // note: no '&'`

Optional "read-only" qualifier

Simplifies Task/Function Usage

Familiar C Features In SystemVerilog

```
do
  begin
    if ( (n%3) == 0 ) continue;
    if (foo == 22) break;
  end
while (foo != 0);
…
```

continue starts next loop iteration

break exits the loop

works with:
for
while
forever
repeat
do while

Blocking Assignments as expressions

```
if ( (a=b) ) …
while ((a = b || c))
```

Extra parentheses required to distinguish from **if(a==b)**

Auto increment/ decrement operators

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

```
a += 3;
s &= mask;
f <<= 3;
```

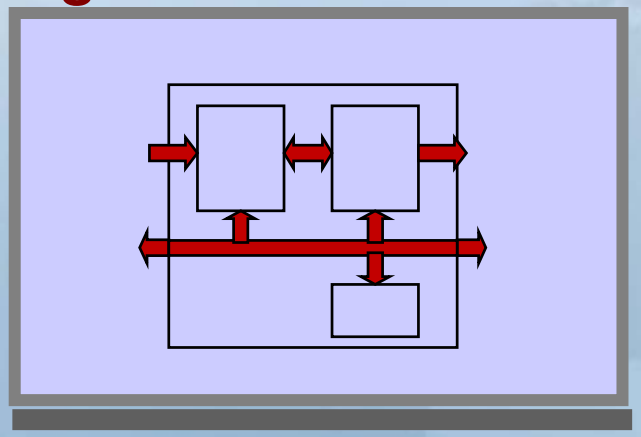
Assignment Operators Semantically equivalent to blocking assignment

Wildcard Comparisons X and Z values act as wildcards

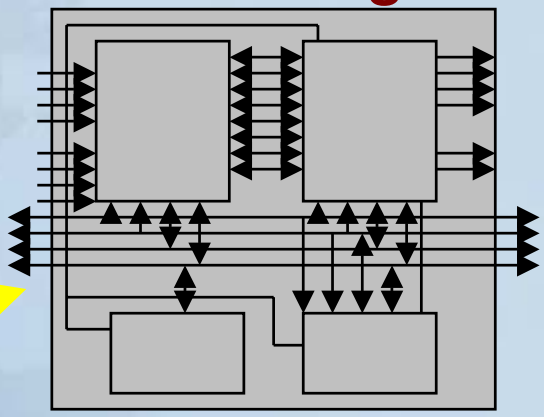
```
a =?= b
a !?= b
```

SystemVerilog Interfaces

Design On A White Board

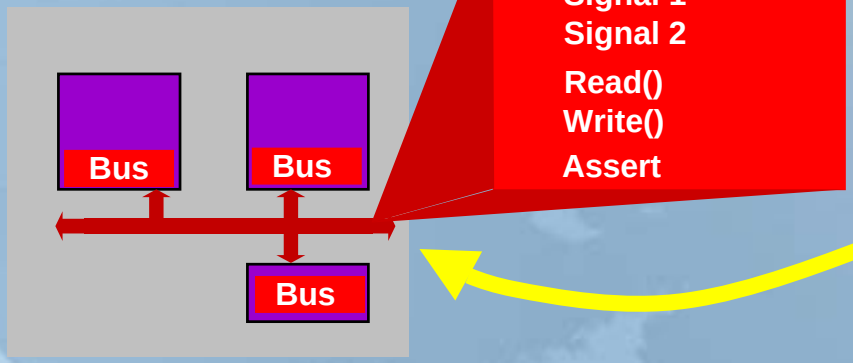


HDL Design



Complex signals
Bus protocol repeated in blocks
Hard to add signal through hierarchy

SystemVerilog Design



Communication encapsulated in interface
- Reduces errors, easier to modify
- Significant code reduction saves time
- Enables efficient transaction modeling
- Allows automated block verification

What is an Interface?

- Provides a new hierarchical structure
  - Encapsulates communication
  - Captures Interconnect and Communication
  - Separates Communication from Functionality
  - Eliminates “Wiring” Errors
  - Enables abstraction in the RTL

```
int i;
logic [7:0] a;

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

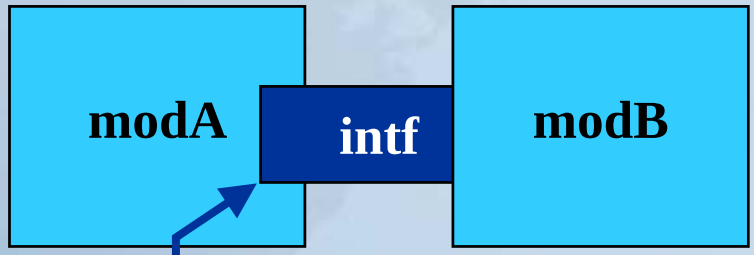
At the simplest level an interface is to a wire what a struct is to a variable

```
int i;
wire [7:0] a;

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

How Interfaces work

```
interface intf;
  bit   A,B;
  byte  C,D;
  logic E,F;
endinterface

intf w;

modA m1(w);
modB m2(w);

module modA (intf i1);
endmodule

module modB (intf i1);
endmodule
```

Instantiate Interface



An interface is similar to a module straddling two other modules

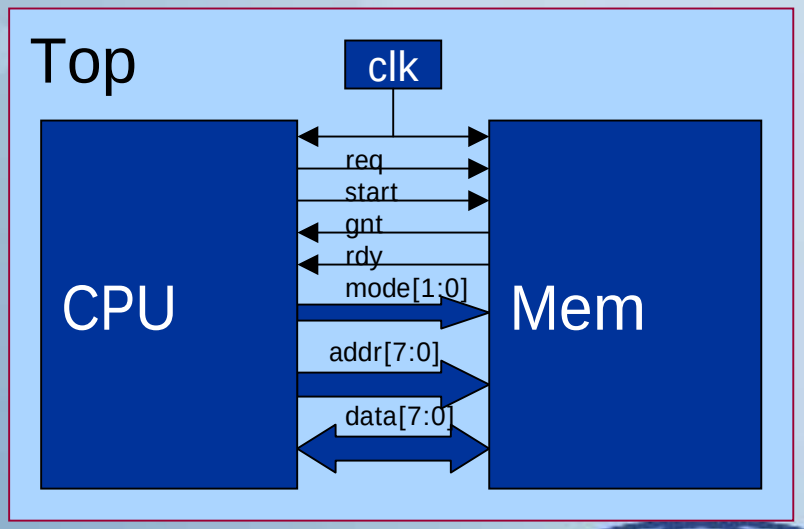
An interface can contain anything that could be in a module except other module definitions or instances

Allows structuring the information flow between blocks

Example without Interface

```
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
```



Example Using Interfaces

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

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
```

Bundle signals in interface

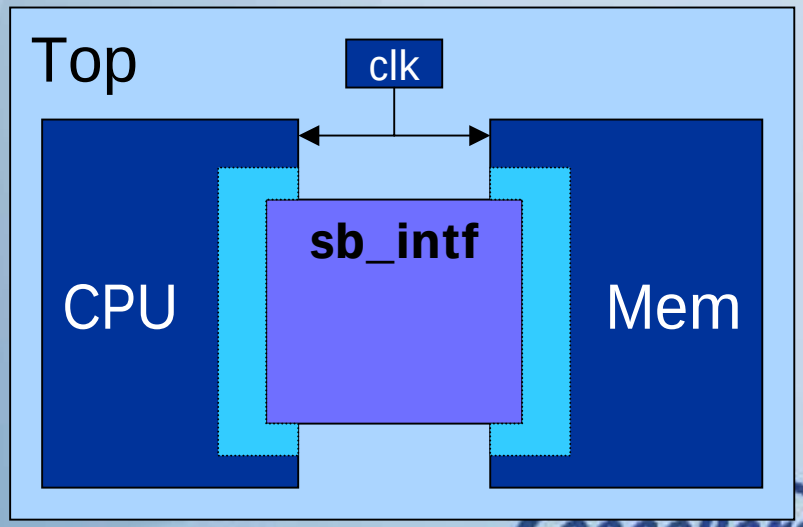
Use interface keyword in port list

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
```

interface instance

Connect interface



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
```

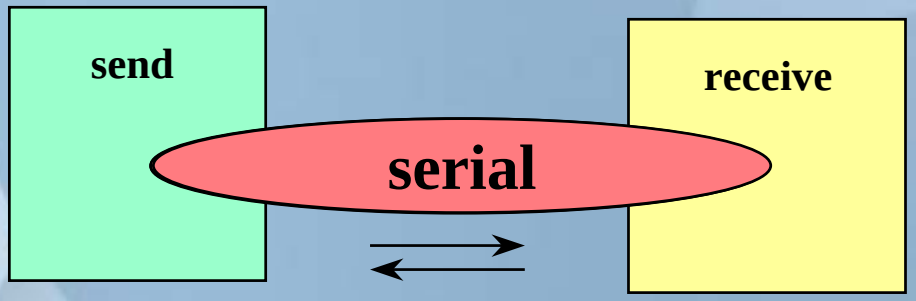
```
interface serial(input bit clk);

  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

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
```

Serial Interface

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);
```



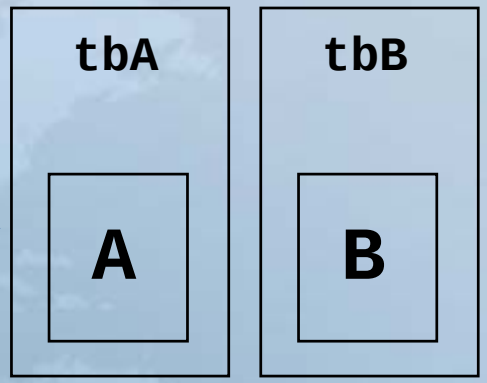

```
module send(input bit clk,
            interface i);
  data_type d;
...
  i.write(d);
endmodule
```

Module inherits communication method from interface

Conventional Verification Strategy

- Pre-Integration

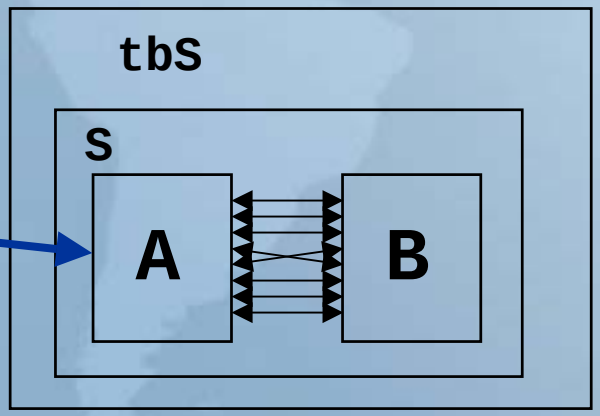
Test Subblocks in isolation



- Testbench reuse problems
- tbA and tbB separate

- Post-Integration

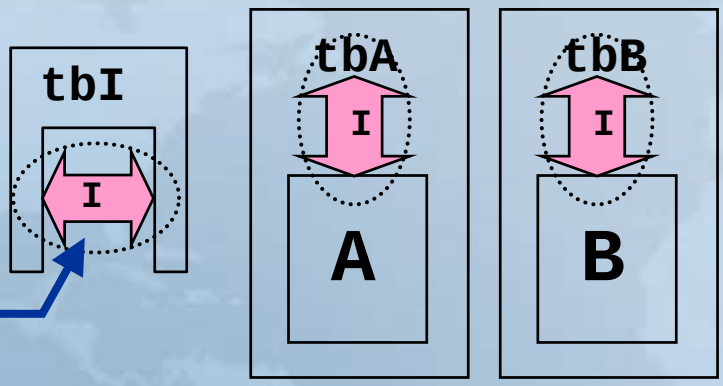
Need to check interconnect, structure (missing wires, twisted busses) as well as functionality



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

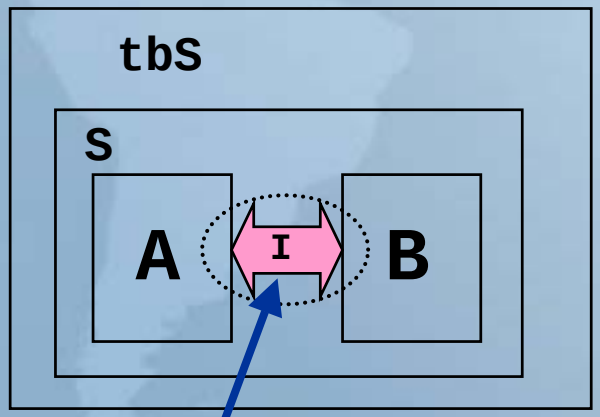
SystemVerilog Verification Strategy

- Pre-Integration



Test interface in isolation

- Post-Integration

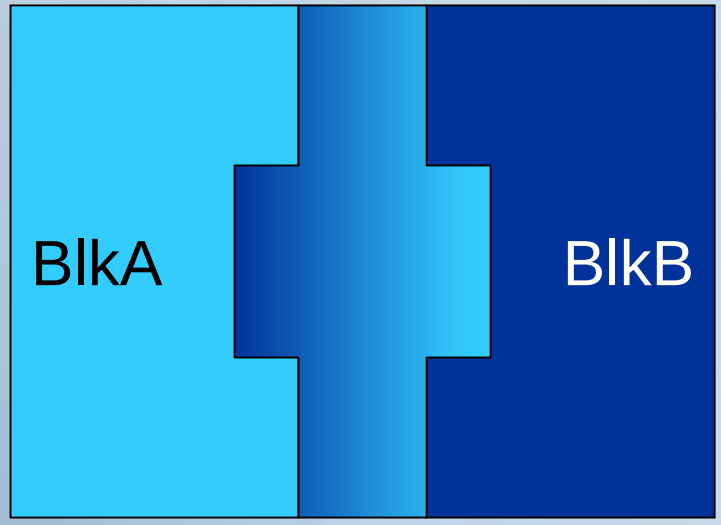


Protocol bugs already flushed out

- Interfaces provide reusable components
- tbA and tbB are 'linked'
- Interface is an executable spec
- Wiring up is simple and not error prone
- Interfaces can contain protocol checkers and coverage counters

SystemVerilog Interfaces: The Key to Design Exploration

- Interfaces Encapsulate Data and How Data Move Between Blocks
- Design Exploration is All About Looking at Alternatives



SystemVerilog Interfaces: The Key to Design Exploration

- Interfaces Encapsulate Data and How Data Move Between Blocks
- Design Exploration is All About Looking at Alternatives
- Interfaces Should Support Multiple Layers of Abstraction for both "Send" and "Receive"
  - Shield BlockA from Abstraction Changes in BlockB