Verilog HDL
Introduction and Reuse

P. Bakowski

bako@ieee.org

Verilog HDL

Verilog – IEEE 1364

1995

Verilog HDL

Verilog – IEEE 1364 1995

Verilog – AMS 1998

Verilog HDL

Verilog – IEEE 1364 — 1995

Verilog – AMS — 1998

Verilog'2001 – IEEE 1364 — 2001

Verilog HDL

Verilog – IEEE 1364	1995
Verilog – AMS	1998
Verilog'2001 – IEEE 1364	2001
SystemVerilog – IEEE 1800	2005

Verilog (SystemVerilog) standard

- Verilog HDL and SystemVerilog are standard hardware description languages that, in some measure, overpower VHDL language.

Verilog (SystemVerilog) standard

- Verilog HDL and SystemVerilog are standard hardware description languages that, in some measure, overpower VHDL language.

- Most CAD tools support both languages; the simulators allows to combine the VHDL components with Verilog modules (IP blocks) to build complete systems.

Verilog (SystemVerilog) standard

- The main features of Verilog are:
  - Verilog is a general standard hardware description language (IEEE standard 1364). It is similar in syntax to the C programming language, thus easy to learn and to use.

Verilog (SystemVerilog) standard

- The main features of Verilog are:
    - Verilog is a general standard hardware description language (IEEE standard 1364). It is similar in syntax to the C programming language, thus easy to learn and to use.
    - Verilog allows different levels of abstraction to be mixed in the same model. The designer can define hardware in terms of gates, RTL architecture, or behavioral code.

Verilog (SystemVerilog) standard

- Most logic synthesis tools support Verilog easier than VHDL; this makes it the language of choice for synthesis. Quite often the VHDL constructs are mapped on Verilog before the synthesis process starts.

Verilog (SystemVerilog) standard

- Most logic synthesis tools support Verilog easier than VHDL; this makes it the language of choice for synthesis. Quite often the VHDL constructs are mapped on Verilog before the synthesis process starts.

- The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with Verilog models.

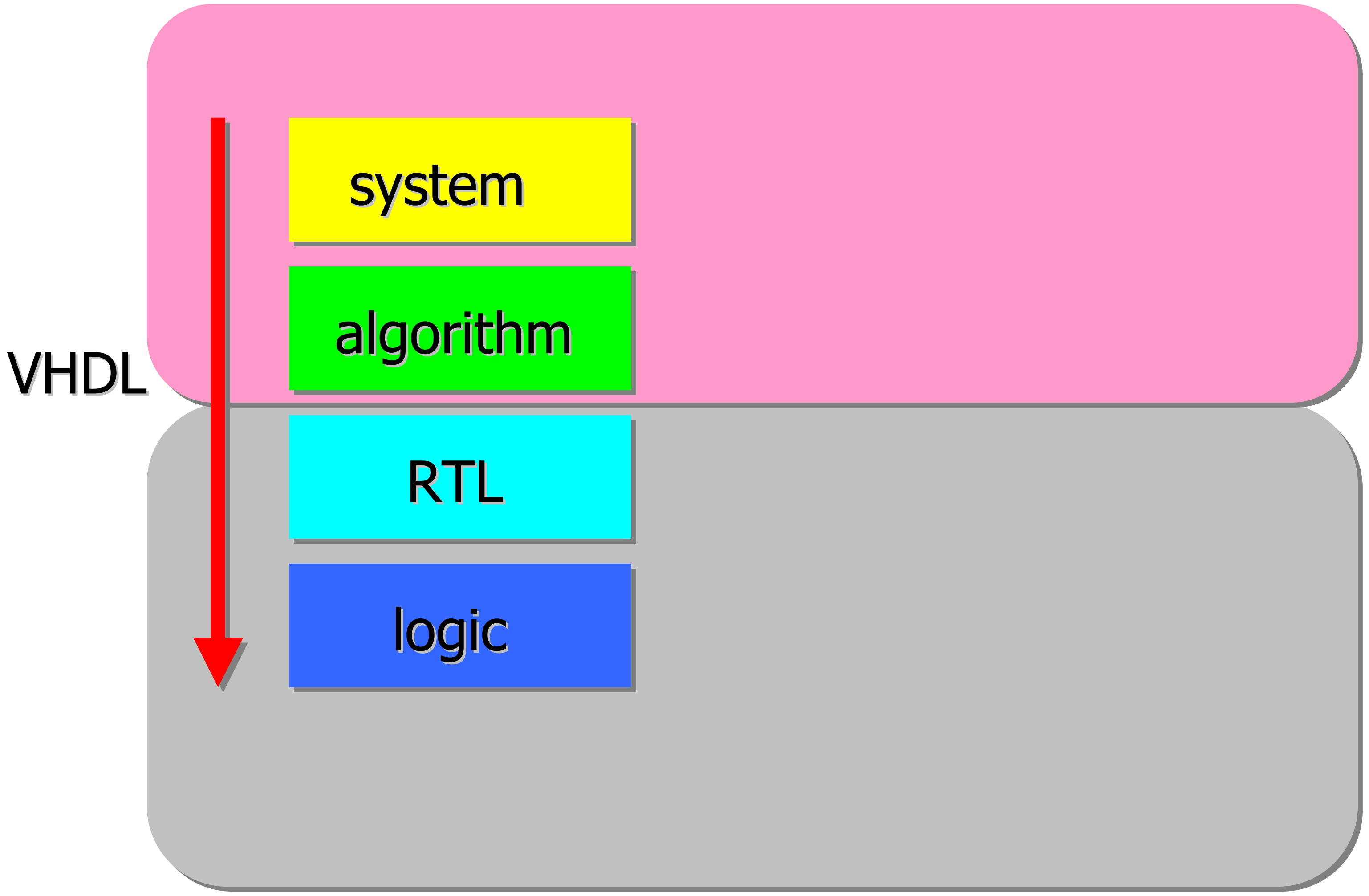
Verilog (SystemVerilog) standard

- Most logic synthesis tools support Verilog easier than VHDL; this makes it the language of choice for synthesis. Quite often the VHDL constructs are mapped on Verilog before the synthesis process starts.

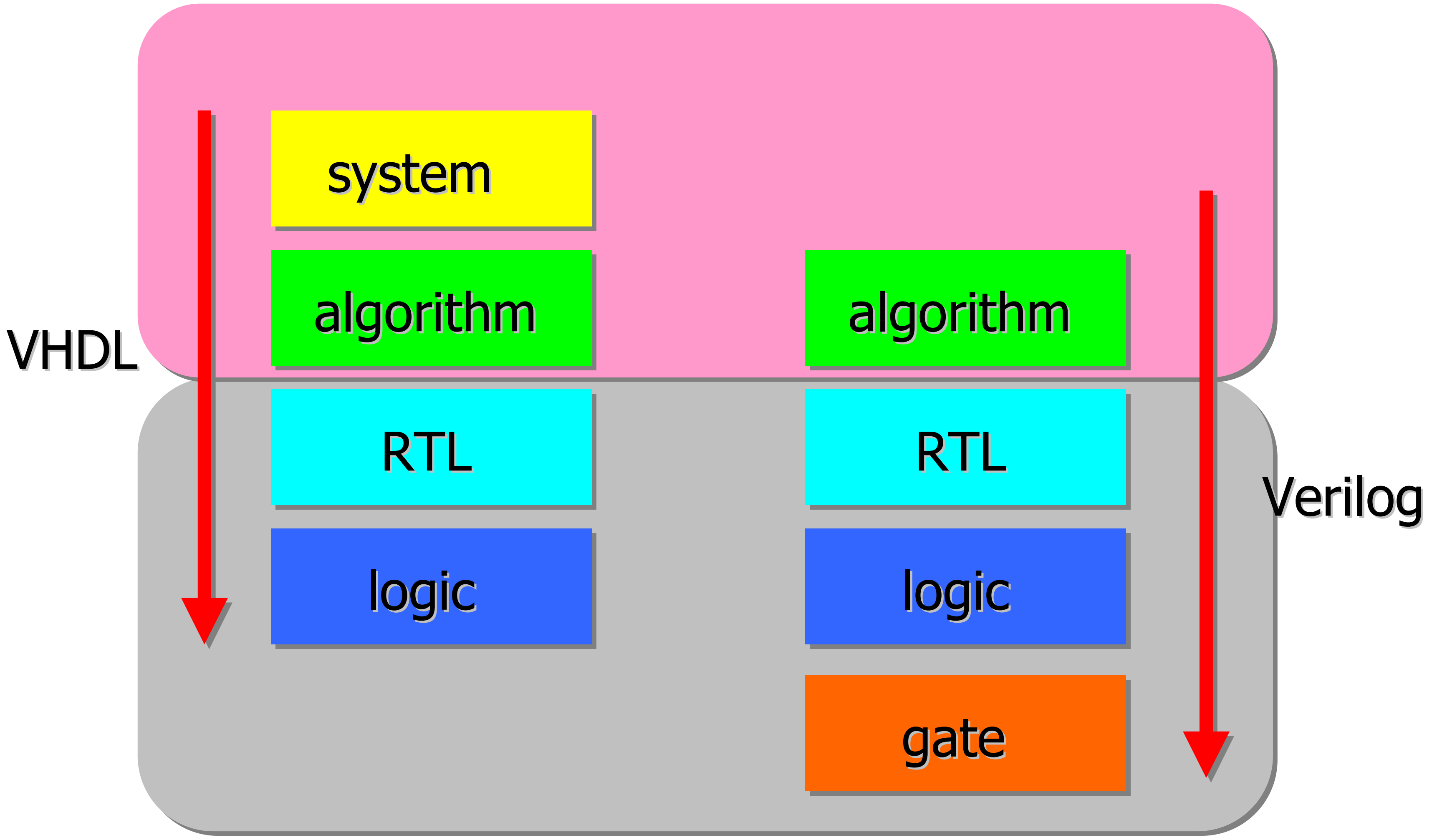
- The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with Verilog models.

- SystemVerilog adds powerful features for object oriented modeling and IP verification

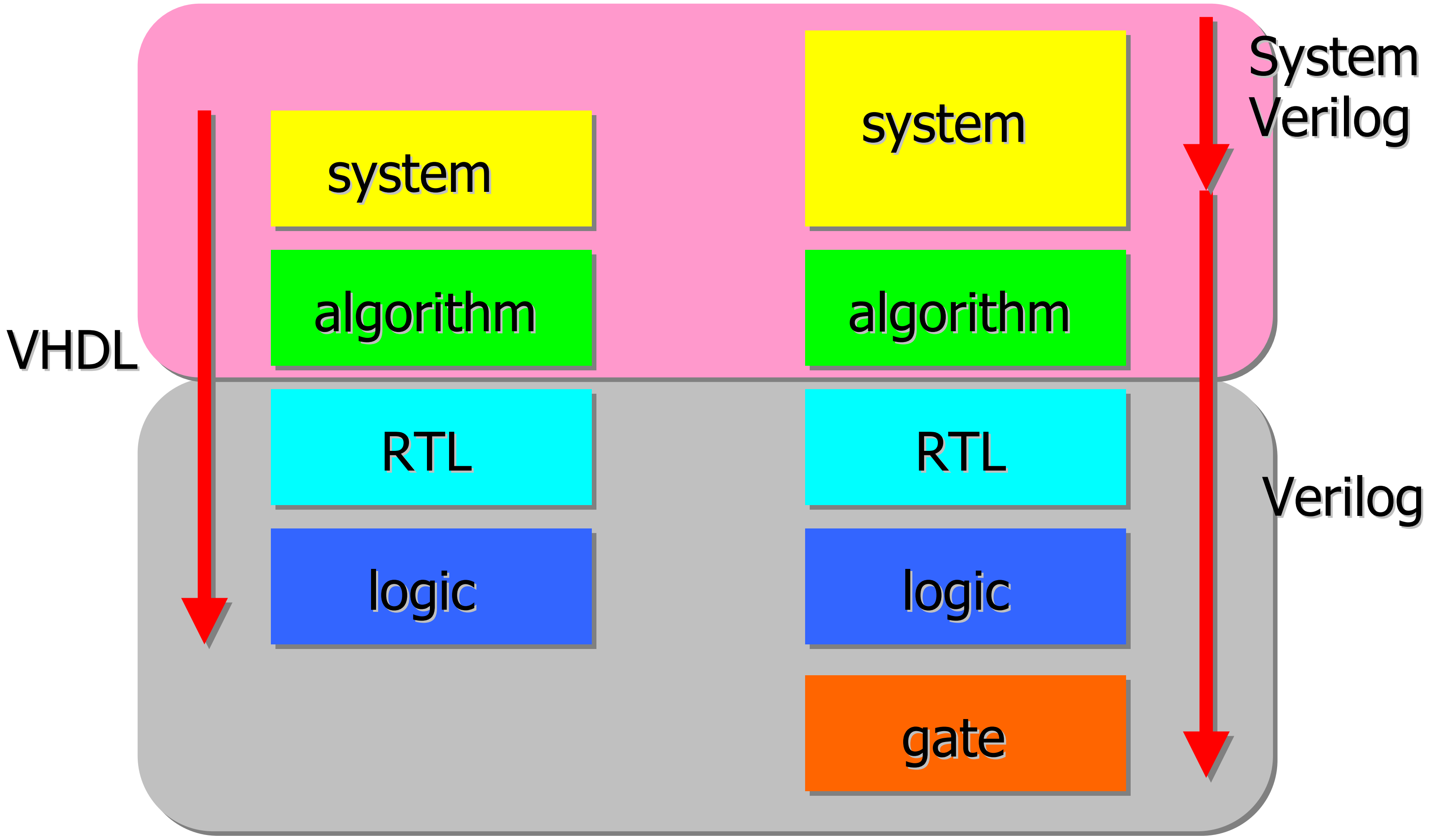
Verilog (SystemVerilog) standard



Verilog (SystemVerilog) standard



Verilog (SystemVerilog) standard



Verilog module

Modules, like VHDL entities, are the basic units in Verilog.

```
module compteur;
integer i;
initial      // initial process
for(i=1;i<=10;i=i+1)
$display(“i=%d”,i);
endmodule
```

Verilog module: inputs/outputs

```
module additionneur (a,b,sum);
input a,b;
output sum;
wire [31:0] a,b;
reg [31:0] sum;

…

endmodule
```

Verilog « hardware variables » carry always:

0, 1, x, or z values

P. Bakowski

Verilog module: inputs/outputs

```
module additionneur (a,b,sum);
input a,b;
output sum;
wire [31:0] a,b;
reg [31:0] sum;
always @(a or b)   // permanent process
// activated by events at a or b
begin
sum = a + b;
$display(“la somme est: %d”, sum);
end
endmodule
```

Verilog parameters

The modules may contain parameters; their role is similar to generics of VHDL

```
module compteur_par;
parameter MAX;    -- Verilog parameter
integer i;
initial
for(i=1;i<=MAX;i=i+1)
$display(“i=%d”,i);
endmodule
```

Verilog parameters' instantiation

```
module deux_composants
defparam
compteur1.MAX = 6,
compteur2.MAX = 12;
// initialisation of parameters
compteur_par compteur1();
// first instance of compteur_par
compteur_par compteur2();
// second instance of compteur_par
endmodule
```

Structural description in Verilog

```
module adderg(a,b,cin,sum,cout);
input a,b,cin;
output sum, cout;
wire a1,x1,a2;
xor   xor0(x1,a,b);
and   and0(a1,a,b);
xor   xor1(sum,cin,x1);
and   and1(a2,cin,x1);
or    or0(cout,a1,a2);
endmodule
```

Behavioral description in Verilog

```
module adderbeh(a,b,cin,sum,cout);
input [3:0] a,b; input cin;
output [3:0] sum; output cout;
reg  [3:0] sum; reg cout;
reg [4:0] res;
always @(a or b or cin)
begin
cout = 0;sum = 0; res = a + b + cin;
cout = res[4]; sum = res[3:0];
end
endmodule
```

Verilog primitives

```
primitive tff_clr(q,clock,clear);
output q; reg q; input clock,clear;
table
//clock clear : q : q+;
  ?        1    : ? : 0;     // clear
  ?       (10)  : ? : -;     // - ignore
  (10)     0    : 1 : 0;     // toggle to 0
  (10)     0    : 0 : 1;     // toggle to 1
  (0?)     0    : ? : -;     // ignore
endtable
endprimitive
```

A clock module (timing)

```
module horloge(clock);
output clock;
reg clock;
initial
#5 clock =1;                  // initial process
always                        // main process
#50 clock = ~clock ; // ~ logic negation
endmodule;
```

Continuous assignment

```
module adderdf(a,b,cin,sum,cout);
input a,b,cin;
output sum,cout;
  assign sum = a ^ b ^ cin;
  assign cout = a&b | a&cin | b&cin;
endmodule
```

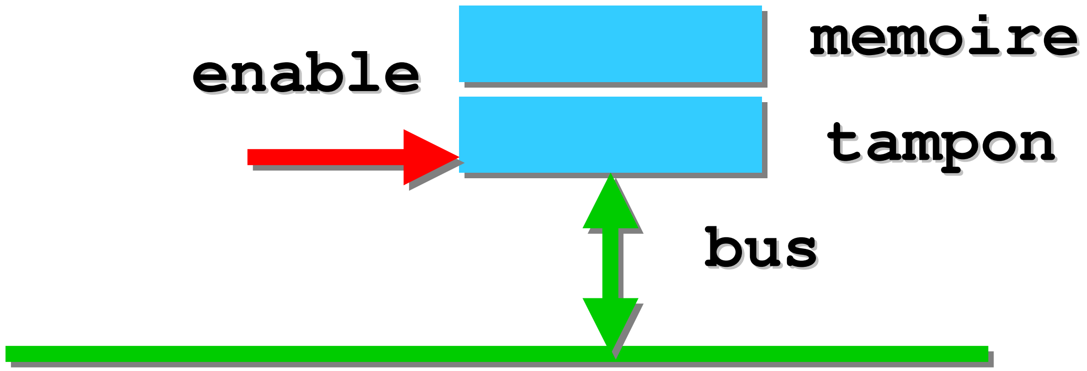
Continuous assignment

```
module adderdf(a,b,cin,sum,cout);
input a,b,cin;
output sum,cout;
  assign sum = a ^ b ^ cin;
  assign cout = a&b | a&cin | b&cin;
endmodule
```

corresponds to signal assignment in VHDL:
sum <= a xor b xor cin;

Bus assignments

```
module sousmodule (bus, enable);
inout [7:0] bus; input enable;
reg [7:0] tampon, memoire;
wire [7:0] bus = tampon; // continuous
always @(bus or enable)
..
endmodule
```



Bus assignments

```
module sousmodule (bus, enable);
inout [7:0] bus; input enable;
reg [7:0] tampon, memoire;
wire [7:0] bus = tampon; // continuous
always @(bus or enable)
if (enable == 1)  // reading from bus
begin
bus = 8'bz; // bus is free
#4; memoire = bus; // reading from bus
end
else
tampon = 255 - memoire; // writing to bus
endmodule
```

Bus assignments

```
module sousmodule (bus, enable);
inout [7:0] bus; input enable;
reg [7:0] tampon, memoire;
wire [7:0] bus = tampon; // continuous
always @(bus or enable)
if (enable == 1)  // reading from bus
begin
bus = 8’bz; // bus is free
#4; memoire = bus; // reading from bus
end
else
tampon = 255 - memoire; // writing to bus
endmodule
```

Blocking assignments

```
reg a,b,c;
reg [7:0] x,y;
integer compteur;
initial
begin
a=0;b=1;c=1;  //scalar assignments:time 0
compteur = 0; // assignment at time 0
x = 8’b0; y = x;    // at time 0
#12 x[3] = 1’b1;    // at time 12
#16 y[2:0] = {a,b,c}      // at time 28
compteur = compteur + 1; // at time 28
end
```

Non - blocking assignments

```
reg a,b,c;
reg [7:0] x,y;
integer compteur;
initial
a=0; b=1; c=1;          // at time 0
compteur = 0;           // at time 0
x = 8'b0; y = x;        // at time 0
x[3] <=  #12 1'b1;              // at time 12
y[2:0] <= #16 {a,b,c};          // at time 16
compteur <= compteur + 1;       // at time 0
end
```

Non - blocking loopbacks

```
always @(posedge clk)
begin
ter3 <= #2 ter1;  // old value of ter1
ter1 <= #2 in1;
ter2  <= @(negedge clk) in2 ^ in3;
end
```

Note that the (written) order of the assignments is not important because the internally stored right-hand-side evaluations are stored in the simulator and assigned to the left-hand-side values in the next simulation step.

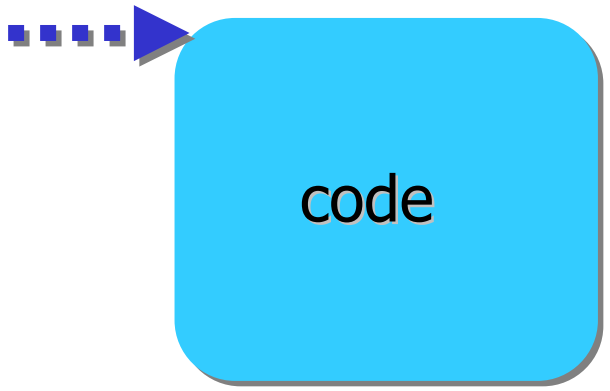
Fork and join

```
module concurrent_blocks_ABC;
initial
begin
$display ("time: %d, begin", $time);
fork    // start of 3 parallel blocks
begin   // start of sequential block A
$display("A begin");#12;$display("A end");
end
begin   // start of sequential block B
$display("B begin");#2;$display("B end");
end
begin   // start of sequential block C
$display("C begin");$display("C end");
end
join
$display ("time:%d,end",$time);
endmodule
```

Give the result of execution !

Tasks and functions

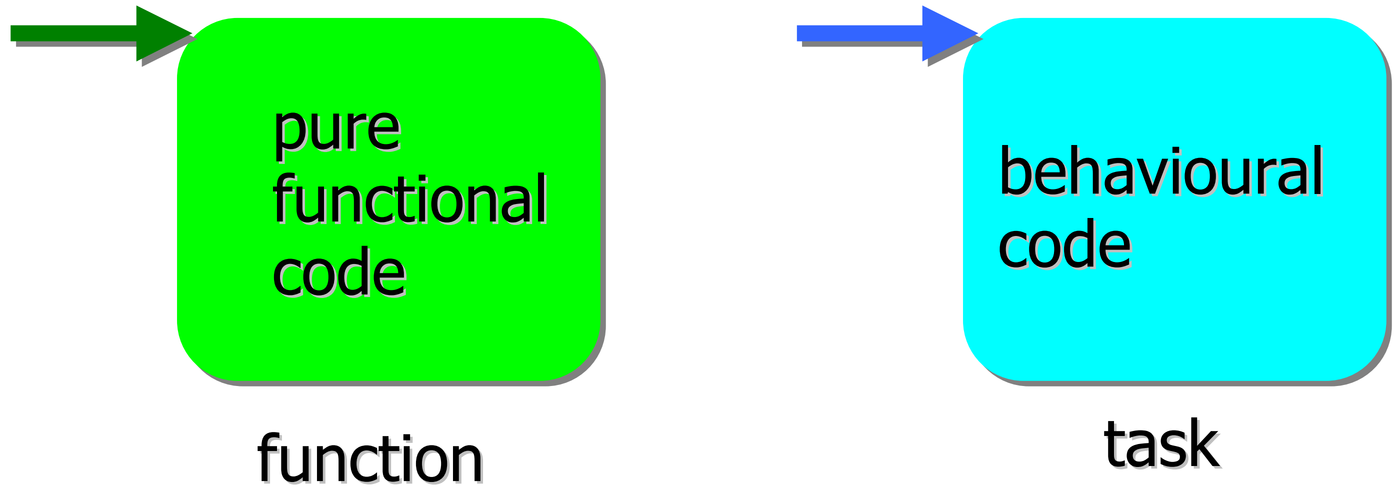
Tasks and functions are building blocks (functional blocks) of complex models using several times the same sequence of code.



Tasks and functions

Tasks and functions are building blocks (functional blocks) of complex models using several times the same sequence of code.

- the main difference between tasks and functions concerns the use of timing

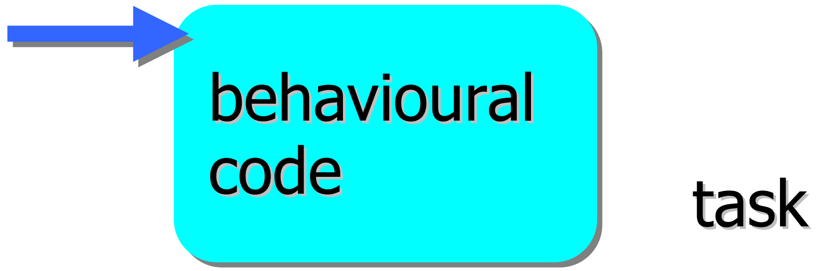


Tasks and functions

Tasks and functions are building blocks (functional blocks) of complex models using several times the same sequence of code.

- the main difference between tasks and functions concerns the use of timing

- the operations in tasks may contain delays and waits; whereas the functions are purely operational



Tasks and functions

Tasks and functions are building blocks (functional blocks) of complex models using several times the same sequence of code.

- the main difference between tasks and functions concerns the use of timing

- the operations in tasks may contain delays and waits; whereas the functions are purely operational

- the tasks resemble VHDL processes

- the Verilog functions operate like C functions

Verilog task

```
task mult; // multiplication task
inout [15:0] a;  input [15:0] b;
reg [7:0] mcnd,mpy; reg [15:0] prod;
begin
mpy = b[7:0]; mcnd = a[7:0]; prod = 0;
repeat (8) // repetition loop
begin
if(mpy[0]) prod=prod +{mcnd,8'h00);
prod = prod >>1;  mpy = mpy >> 1;
#2    // multiplication step delay
end
a=prod;
end
endtask
```

Verilog function

```
function [15:0] mulf
input [15:0] a, b;
reg [7:0] mcnd,mpy;
begin
mpy = b[7:0];mcnd = a[7:0];
mulf  = 0;
repeat (8)
begin
if(mpy[0]) mulf  = mulf  +{mcnd,8'h00);
mulf = mulf >>1; mpy = mpy >> 1;
end
end
endfunction
```

Verilog configuration blocks

- Configuration blocks are used to specify the exact version and source location of each Verilog module

Verilog configuration blocks

- Configuration blocks are used to specify the exact version and source location of each Verilog module

- Virtual model libraries are used in configuration blocks, and separate library map files associate virtual libraries with physical locations.

Verilog configuration blocks

- Configuration blocks are used to specify the exact version and source location of each Verilog module

- Virtual model libraries are used in configuration blocks, and separate library map files associate virtual libraries with physical locations.

- Configuration blocks are specified outside of module definitions.

Verilog configuration blocks

- Configuration blocks are used to specify the exact version and source location of each Verilog module

- Virtual model libraries are used in configuration blocks, and separate library map files associate virtual libraries with physical locations.

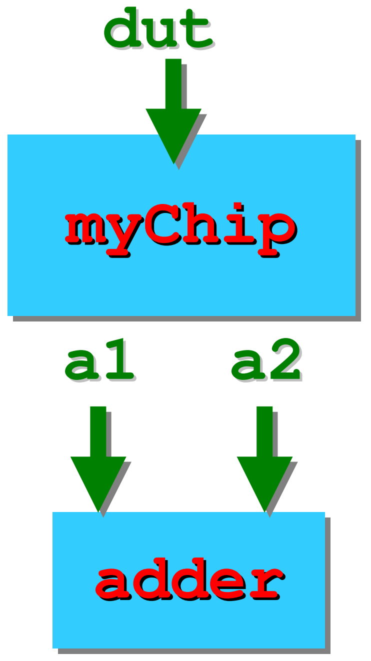
- Configuration blocks are specified outside of module definitions.

- The names of configurations exist in the same name space as module names and primitive names.

Verilog configuration blocks

```
module test;
myChip dut  (...); // myChip – module
// dut instance of design
endmodule
```

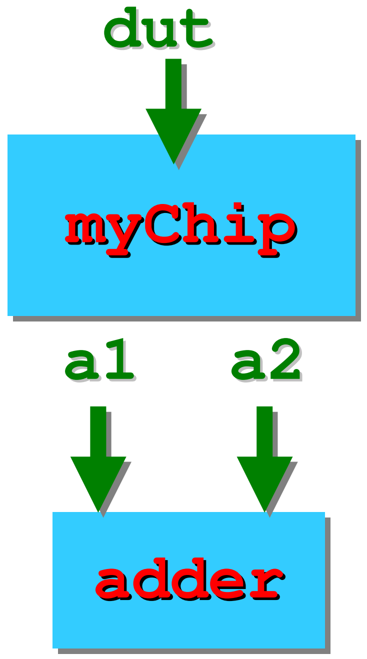
```
module myChip(...);
adder a1  (...);
adder a2  (...);
// instance a2 of adder of
// dut instance of myChip
endmodule
```



Verilog configuration blocks

```
module test;
myChip dut (...); // myChip - module
// dut instance of design
endmodule
```

```
module myChip(...);
adder a1 (...);
adder a2 (...);
// instance a2 of adder of
// dut instance of myChip
endmodule
```



Verilog configuration blocks

```
config cfg4 // name for this configuration
design rtlLib.top
//specify where to find top level modules
```

Verilog configuration blocks

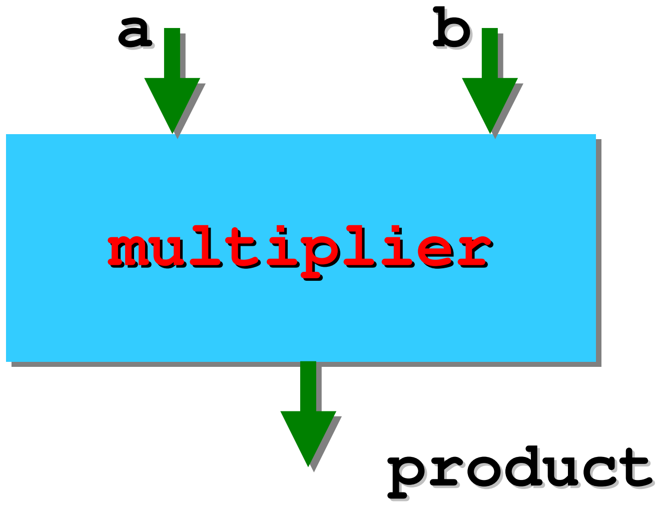
```
config cfg4 // name for this configuration
design rtlLib.top
//specify where to find top level modules
default liblist rtlLib gateLib;
//set the default search order for
//finding instantiated modules
```

Verilog configuration blocks

```
config cfg4 // name for this configuration
design rtlLib.top
//specify where to find top level modules
default liblist rtlLib gateLib;
//set the default search order for
//finding instantiated modules
instance test.dut.a2 liblist gateLib;
// explicitly specify which library to use
// for the following module instance
endconfig
```

Verilog generate blocks

```
module multiplier (a, b, product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width+b_width;
input [a_width-1:0] a;
input [b_width-1:0] b;
output [product_width-1:0] product;
```



Verilog generate blocks

```
module multiplier (a, b, product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width+b_width;
input [a_width-1:0] a;
input [b_width-1:0] b;
output [product_width-1:0] product;
generate
if((a_width < 8) || (b_width < 8))
CLA_multiplier #(a_width, b_width)
u1 (a, b, product);
else
WALLACE_multiplier #(a_width, b_width)
u1 (a, b, product);
endgenerate
```

Verilog generate blocks

```
module Nbit_adder (co, sum, a, b, ci);
parameter SIZE = 4;
output [SIZE-1:0] sum;  output co;
input [SIZE-1:0] a, b;  input ci;
wire [SIZE:0] c;
genvar i; // i - generate index variable
assign c[0] = ci;assign co = c[SIZE];
generate
for(i=0; i<SIZE; i=i+1)
begin:addbit
wire n1,n2,n3;  //internal nets (wires)
xor g1 (n1 ,a[i] ,b[i]);
xor g2 (sum[i] ,n1 ,c[i]);
and g3 (n2 ,a[i] ,b[i]);
and g4 (n3 ,n1 ,c[i]);
or g5 (c[i+1] ,n2 ,n3);
end
endgenerate
endmodule
```

Verilog constant functions

- Constant functions help to create re-usable models which can be scaled to different sizes.

Verilog constant functions

- Constant functions help to create re-usable models which can be scaled to different sizes.

- In the following example, the function called clogb2 returns an integer which has the value of the ceiling of the log base 2.

Verilog constant functions

- Constant functions help to create re-usable models which can be scaled to different sizes.
- In the following example, the function called clogb2 returns an integer which has the value of the ceiling of the log base 2.
- This constant function is used to determine how wide a RAM address bus must be, based on the number of addresses in the RAM.

Verilog constant functions

```
module ram (address,write,chip_select,data);
parameter WIDTH = 8;
parameter SIZE = 256;
localparam ADDRESS_SIZE = clogb2(SIZE);
input [ADDRESS_SIZE-1:0] address;
input write, chip_select;
inout [WIDTH-1:0] data;
reg [WIDTH-1:0] ram_data [0:SIZE-1];
......
endmodule
```

Verilog constant functions

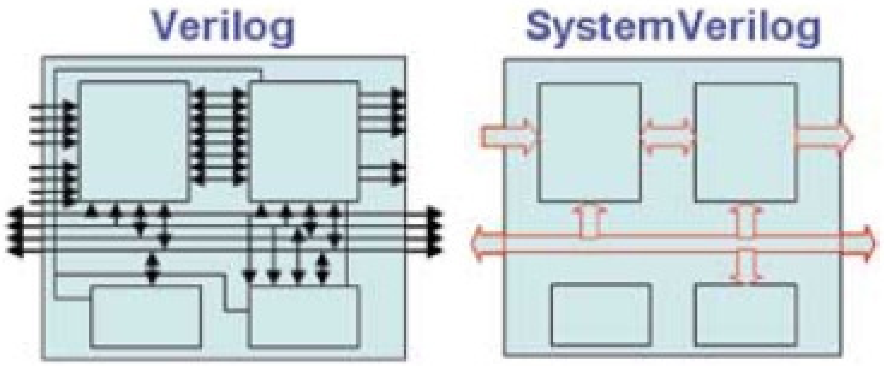
```
module ram (address,write,chip_select,data);
parameter WIDTH = 8;
parameter SIZE = 256;
localparam ADDRESS_SIZE = clogb2(SIZE);
input [ADDRESS_SIZE-1:0] address;
input write, chip_select;
inout [WIDTH-1:0] data;
reg [WIDTH-1:0] ram_data [0:SIZE-1];
.......
endmodule
```

Verilog constant functions

```
module ram (address,write,chip_select,data);
...
function integer clogb2;
//clogb2 constant function
input depth;
integer i;
begin
clogb2 = 1;
for (i = 0; 2**i < depth; i=i+1) clogb2=i+1;
end
endfunction
...
endmodule
```

SystemVerilog interfaces

- An Interface encapsulates the connectivity between two or more modules.

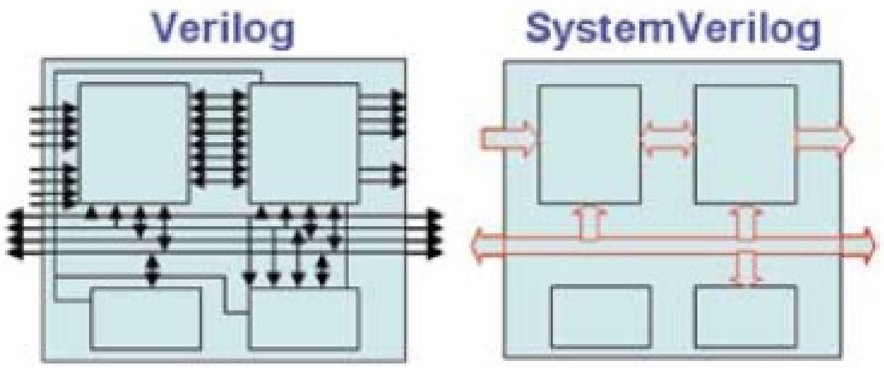


P. Bakowski

SystemVerilog interfaces

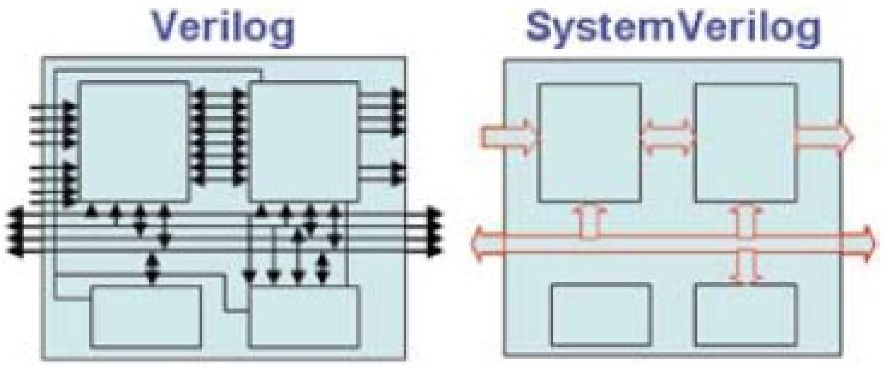
- An Interface encapsulates the connectivity between two or more modules.

- To understand interfaces, consider two modules moduleA and moduleB who talk to each other through their ports.



SystemVerilog interfaces

- An Interface encapsulates the connectivity between two or more modules.
- To understand interfaces, consider two modules moduleA and moduleB who talk to each other through their ports.
- The definitions of moduleA and moduleB are shown below along with the top-level wrapper top that encompasses them.



SystemVerilog interfaces

```
module moduleA (input bit clk ,input logic ack,
input logic ready,output logic send,
output logic [31:0] data ); ...
// actual module definition here
endmodule
```

SystemVerilog interfaces

```
module moduleA (input bit clk ,input logic ack,
input logic ready,output logic send,
output logic [31:0] data ); ...
// actual module definition here
endmodule
module moduleB ( input bit clk , input logic
send , input logic [31:0] data , output logic
ack , output logic ready ); ...
// actual module definition here
endmodule
```

SystemVerilog interfaces

```
module moduleA (input bit clk ,input logic ack,
input logic ready,output logic send,
output logic [31:0] data ); ...
// actual module definition here
endmodule
module moduleB ( input bit clk , input logic
send , input logic [31:0] data , output logic
ack , output logic ready ); ...
// actual module definition here
endmodule
module top;
... clockgen CLOCKGEN (clk);
// the clock generator
moduleA AA (clk, ack, ready, send, data);
moduleB BB (clk, send, data, ack, ready);
endmodule
```

SystemVerilog interfaces

- Configuration specification is tied intrinsically with the communicating ports of the modules.

SystemVerilog interfaces

- Configuration specification is tied intrinsically with the communicating ports of the modules.

- For example, there is no way one can test *moduleA* without building *moduleB* (or a testbench equivalent) or, each time either the communication protocol between the two modules changes, either by addition or deletion of a signal, or by change in the behavior of a signal, the two modules must change simultaneously to reflect that.

SystemVerilog interfaces

- Configuration specification is tied intrinsically with the communicating ports of the modules.

- For example, there is no way one can test *moduleA* without building *moduleB* (or a testbench equivalent) or, each time either the communication protocol between the two modules changes, either by addition or deletion of a signal, or by change in the behavior of a signal, the two modules must change simultaneously to reflect that.

- There is no way to abstract the interface between the two modules independent of the modules themselves.

SystemVerilog interfaces

- Configuration specification is tied intrinsically with the communicating ports of the modules.
- For example, there is no way one can test *moduleA* without building *moduleB* (or a testbench equivalent) or, each time either the communication protocol between the two modules changes, either by addition or deletion of a signal, or by change in the behavior of a signal, the two modules must change simultaneously to reflect that.
- There is no way to abstract the interface between the two modules independent of the modules themselves.
- Interfaces achieve this.

SystemVerilog interfaces

With interfaces, one can define the same design described above as follows.

```
interface intf_AB;
logic ack;
logic ready;
logic send;
logic [31:0] data; ...
// actual interface definition here
endinterface
```

SystemVerilog interfaces

```
module moduleA ( input bit clk , intf_AB intf1 );
...
// actual module definition here
endmodule
module moduleB ( input bit clk , intf_AB intf2 );
...
// actual module definition here
endmodule
module top; ...
intf_AB intf();    // the interface declaration
clockgen CLOCKGEN (clk); // the clock generator
moduleA AA ( .clk (clk ) ,.intf1 (intf ) );
moduleB BB ( .clk (clk ) ,.intf2 (intf ) );
endmodule
```

SystemVerilog interfaces

As the above example shows, now both the instances of moduleA and moduleB are independent of the connectivity between the two modules.

SystemVerilog interfaces

As the above example shows, now both the instances of moduleA and moduleB are independent of the connectivity between the two modules.

Here are some more information:

- Interfaces are defined just as modules but, unlike a module, an interface may contain nested definitions of other interfaces.

SystemVerilog interfaces

Here are some more information:

- Interfaces are defined just as modules but, unlike a module, an interface may contain nested definitions of other interfaces.
- An interface instantiation is similar to a module instantiation, and array of instances is permitted.

SystemVerilog interfaces

Here are some more information:

- Interfaces are defined just as modules but, unlike a module, an interface may contain nested definitions of other interfaces.
- An interface instantiation is similar to a module instantiation, and array of instances is permitted.

- The following example defines 8 instances of the same interface.
  - `intf_AB intf[7:0]`

Ports and interfaces

- Ports of an interface works similar to ports of a module. It has a direction (input, output or inout).

Ports and interfaces

- Ports of an interface works similar to ports of a module. It has a direction (input, output or inout).

- The variable that is connected to the port inside the interface is accessible outside by its hierarchical name.

Ports and interfaces

- Ports of an interface works similar to ports of a module. It has a direction (input, output or inout).

- The variable that is connected to the port inside the interface is accessible outside by its hierarchical name.

- For example, we can declare *clk* as an input port of the interface *intf_AB* and then access it from within the modules *moduleA* and *moduleB*.

Ports and interfaces

- This is shown below.

```
interface intf_AB(input bit clk);
logic ack; logic ready; logic send;
logic [31:0] data;
... // actual interface definition here
endinterface
```

Ports and interfaces

```
module moduleA (intf_AB intf1); ...
always @(posedge intf1.clk)
// clk from intf_AB ...
endmodule
module moduleB (intf_AB intf2); ...
// actual module definition here
endmodule
module top; ...
intf_AB intf(); // the interface declaration
clockgen CLOCKGEN (clk);//the clock generator
moduleA AA (.intf1 (intf));
moduleB BB (.intf2 (intf));
endmodule
```

Modports

- A legitimate question for you to ask at this point is 'if an interface is passed in the portlist of all the connected modules, how does a port behaves as an output of one module and inputs to the rest of the modules?'

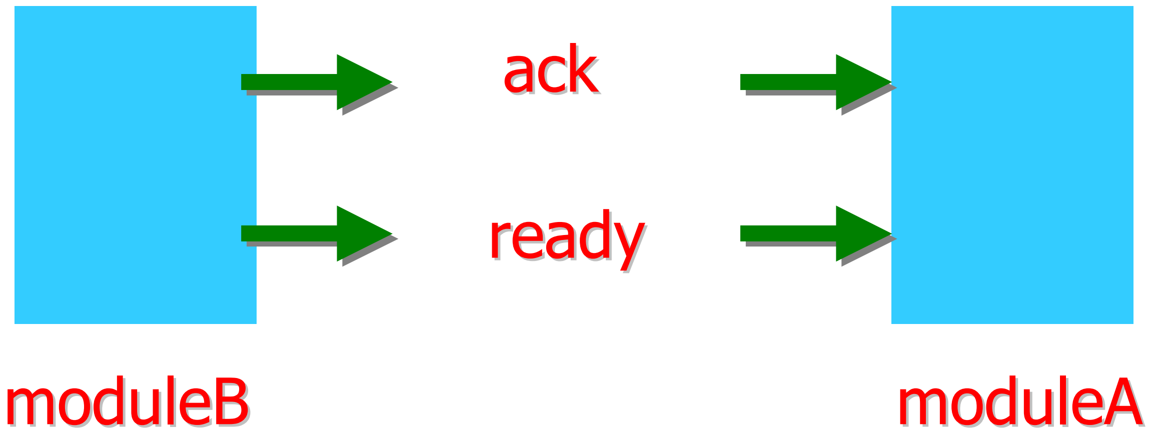
Modports

- A legitimate question for you to ask at this point is 'if an interface is passed in the portlist of all the connected modules, how does a port behaves as an output of one module and inputs to the rest of the modules?'

- This problem is solved by the use of *modports*. Modports configure the directions of various variables inside an interface based on which module is passing the interface in its port list.

Modports

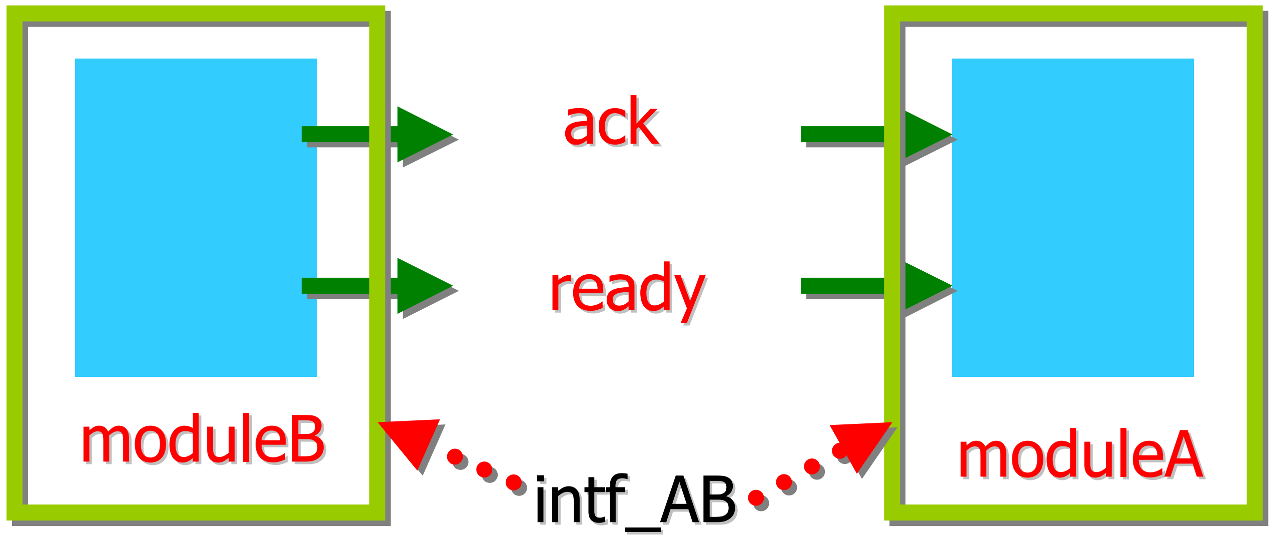
- For example, *ack* and *ready* are inputs to *moduleA*, but they are outputs from *moduleB*.



Modports

- For example, *ack* and *ready* are inputs to *moduleA*, but they are outputs from *moduleB*.

- How do we reflect that when we use *intf_AB* as an interface between them?



Modports

- For example, *ack* and *ready* are inputs to *moduleA*, but they are outputs from *moduleB*.

- How do we reflect that when we use *intf_AB* as an interface between them?

- The same goes true for ports *send* and *data*, which have a direction opposite to that of *ack* and *ready*.

Modports

To do this, we define two modports within *intf_AB*: *source* for *moduleA* and *sink* for *moduleB*.

```
interface intf_AB;
logic ack; logic ready; logic send;
logic [31:0] data;
modport source(input ack, ready, output send,
data);
modport sink(output ack, ready, input send,
data );
endinterface
```

Modports

Then we change the definitions of *moduleA* and *moduleB* as follows.

```
module moduleA ( input bit clk ,
intf_AB.source intf1 ); ...
// actual module definition here
endmodule
module moduleB ( input bit clk ,
intf_AB.sink intf2 ); ...
// actual module definition here
endmodule
```

Modports

Then we change the definitions of *moduleA* and *moduleB* as follows.

```
module moduleA ( input bit clk ,
intf_AB.source intf1 ); ...
// actual module definition here
endmodule
module moduleB ( input bit clk ,
intf_AB.sink intf2 ); ...
// actual module definition here
endmodule
```

Modports

Finally, in the top level, these two modules are instantiated as follows as usual.

```
module top; ...
intf_AB intf(); // the interface declaration
clockgen CLOCKGEN (clk);
// the clock generator
moduleA AA ( .clk (clk ) ,.intf1 (intf ) );
moduleB BB ( .clk (clk ) ,.intf2 (intf ) );
endmodule
```

Parameterized interface

- Parameterizing is one more way to generalize an interface so that it can be used with a wide variety of design objects.

Parameterized interface

- Parameterizing is one more way to generalize an interface so that it can be used with a wide variety of design objects.

- This way parameterizing an interface is no different from parameterizing a module or a function.

Parameterized interface

- Parameterizing is one more way to generalize an interface so that it can be used with a wide variety of design objects.

- This way parameterizing an interface is no different from parameterizing a module or a function.

- As a matter of fact, you will find the syntax for parameterizing an interface also closely resembles that of a module or a function; as shown in the example below.

Parameterized interface

- Parameterizing is one more way to generalize an interface so that it can be used with a wide variety of design objects.

- This way parameterizing an interface is no different from parameterizing a module or a function.

- As a matter of fact, you will find the syntax for parameterizing an interface also closely resembles that of a module or a function; as shown in the example below.

- Here, the default MSB of the bus *data* is 31.
However, when this interface is instantiated in the module *top*, the WIDTH parameter is changed to 15 and 63.

SystemVerilog classes

- SystemVerilog introduces classes as the foundation of the testbench automation language.

SystemVerilog classes

- SystemVerilog introduces classes as the foundation of the testbench automation language.
- Classes are used to model data, whose values can be created as part of the constrained random methodology.

SystemVerilog classes

- SystemVerilog introduces classes as the foundation of the testbench automation language.
- Classes are used to model data, whose values can be created as part of the constrained random methodology.
- A class is a user-defined data type.

SystemVerilog classes

- SystemVerilog introduces classes as the foundation of the testbench automation language.
- Classes are used to model data, whose values can be created as part of the constrained random methodology.
- A class is a user-defined data type.
- Classes consist of data (called *properties*) and tasks and functions to access the data (called *methods*).

SystemVerilog classes

- SystemVerilog introduces classes as the foundation of the testbench automation language.
- Classes are used to model data, whose values can be created as part of the constrained random methodology.
- A class is a user-defined data type.
- Classes consist of data (called *properties*) and tasks and functions to access the data (called *methods*).

- In SystemVerilog, classes support the following aspects of object-orientation – encapsulation, data hiding, inheritance and polymorphism.

SystemVerilog class declaration

Here is a simple class declaration:

```
class C;
  int x;
  task set (int i);
    x = i;
  endtask
  function int get;
    return x;
  endtask
endclass
```

This class has a single data member, x, and two methods.

SystemVerilog class instantiation

To use the class, an object must be created:

```
C c1;
c1 = new;
```

- The first of these statements declares c1 to be a C.
- In other words, the variable c1 can contain a *handle* to an *object* (i.e. an instance) of the class C.

SystemVerilog class instantiation

To use the class, an object must be created:

```
C c1;
c1 = new;
```

- The first of these statements declares c1 to be a C.
- In other words, the variable c1 can contain a *handle* to an *object* (i.e. an instance) of the class C.

- The second statement creates an object and assigns its handle to c1.

SystemVerilog class instantiation

To use the class, an object must be created:

```
C c1;
c1 = new;
```

The two statements could be replaced by the following statement, which declares a variable, creates a class object and initialises the variable:

```
C c1 = new;
```

SystemVerilog class instantiation

Having created a class object, we can use the class methods to assign and look at the data value, x:

```
initial
begin
  c1.set(3);
  $display("c1.x is %d", c1.get());
end
```

Data hiding

Although the task set() and the function get() were intended to be the means by which the class's member variable x was assigned and retrieved, it would be possible to do this directly:

```
initial
begin
  c1.x = 3;
  $display("c1.x is %d", c1.x);
end
```

Data hiding

```
initial
begin
  c1.x = 3;
  $display("c1.x is %d", c1.x);
end
```

This is because all class members are, by default, publicly visible.

Data hiding

```
initial
begin
  c1.x = 3;
  $display("c1.x is %d", c1.x);
end
```

This is because all class members are, by default, publicly visible.

To hide x, it must be declared local:

```
local int x;
```

It is now illegal to access c1.x outside the class, except using the class methods.

Classes and parameters

Classes may be parameterized in the same way that modules may.

```
class #(parameter int N = 1) Register;
  bit [N-1:0] data;
  ...
endclass
```

Classes and parameters

Classes may be parameterised in the same way that modules may.

```
class #(parameter int N = 1) Register;
  bit [N-1:0] data;
  ...
endclass
```

The default parameter value can be overridden when the class is instantiated.

```
Register #(4) R4; // data is bit [3:0]
Register #(.N(8)) R8; // data is bit [7:0]
Register R;         // data is bit [0:0]
```

Extending classes - Inheritance

- One of the key features of object-oriented programming is the ability to create new classes that are based on existing classes.

Extending classes - Inheritance

- One of the key features of object-oriented programming is the ability to create new classes that are based on existing classes.

- A derived class by default inherits the properties and methods of its parent or base class.

Extending classes - Inheritance

- One of the key features of object-oriented programming is the ability to create new classes that are based on existing classes.

- A derived class by default inherits the properties and methods of its parent or base class.

- However, the derived class may add new properties and methods, or modify the inherited properties and methods.

Extending classes - Inheritance

- One of the key features of object-oriented programming is the ability to create new classes that are based on existing classes.

- A derived class by default inherits the properties and methods of its parent or base class.

- However, the derived class may add new properties and methods, or modify the inherited properties and methods.

- In other words, the new class is a more specialised version of the original class.

Extending classes - Inheritance

In SystemVerilog the syntax for deriving or inheriting one class from another is this:

```
class Derived extends BaseClass;
// new and overridden property
// and method declarations.
endclass
```

Extending classes - Inheritance

Consider the example of the class Register, which was used earlier.
This could represent a general-purpose data register, which stores a value.

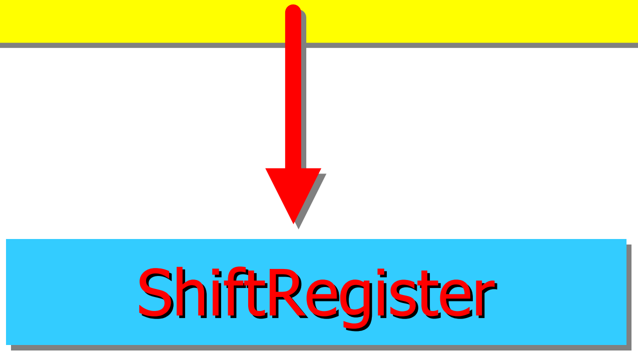
Extending classes - Inheritance

Consider the example of the class Register, which was used earlier.
This could represent a general-purpose data register, which stores a value.
We could derive a new class, ShiftRegister, from this that represents a specialised type of a register.

Extending classes - Inheritance

We could derive a new class, ShiftRegister, from this that represents a specialised type of a register.

```
class ShiftRegister extends Register;
task shiftleft;  data = data << 1;
endtask
task shiftright; data = data >> 1;
endtask
endclass
```



Extending classes - Inheritance

Objects of the class ShiftRegister can be manipulated using the original set() and get() methods, as well as the shiftleft() and shiftright() methods.

Extending classes - Inheritance

Objects of the class ShiftRegister can be manipulated using the original set() and get() methods, as well as the shiftleft() and shiftright() methods.

However, there is a problem if the data property in Register was declared as local – it would not be visible in the extended class!

Extending classes - Inheritance

Objects of the class ShiftRegister can be manipulated using the original set() and get() methods, as well as the shiftleft() and shiftright() methods.

However, there is a problem if the data property in Register was declared as local – it would not be visible in the extended class!

So instead, it would need to be declared protected:

```
class Register;
  protected int data;
  ...
endclass
```

Virtual classes and methods

- Sometimes, it is useful to create a class without intending to create any objects of the class.

Virtual classes and methods

- Sometimes, it is useful to create a class without intending to create any objects of the class.
- The class exists simply as a base class from which other classes can be derived.

Virtual classes and methods

- Sometimes, it is useful to create a class without intending to create any objects of the class.
- The class exists simply as a base class from which other classes can be derived.

- In SystemVerilog this is called an *abstract class* and is declared by using the word *virtual*:

```
virtual class Register;
   ...
endclass
```

Summary

- Verilog and SystemVerilog IEEE standards
- Introduction to Verilog
- Verilog configuration blocks
- Verilog generate blocks
- Verilog constant functions
- SystemVerilog interfaces
- SystemVerilog classes