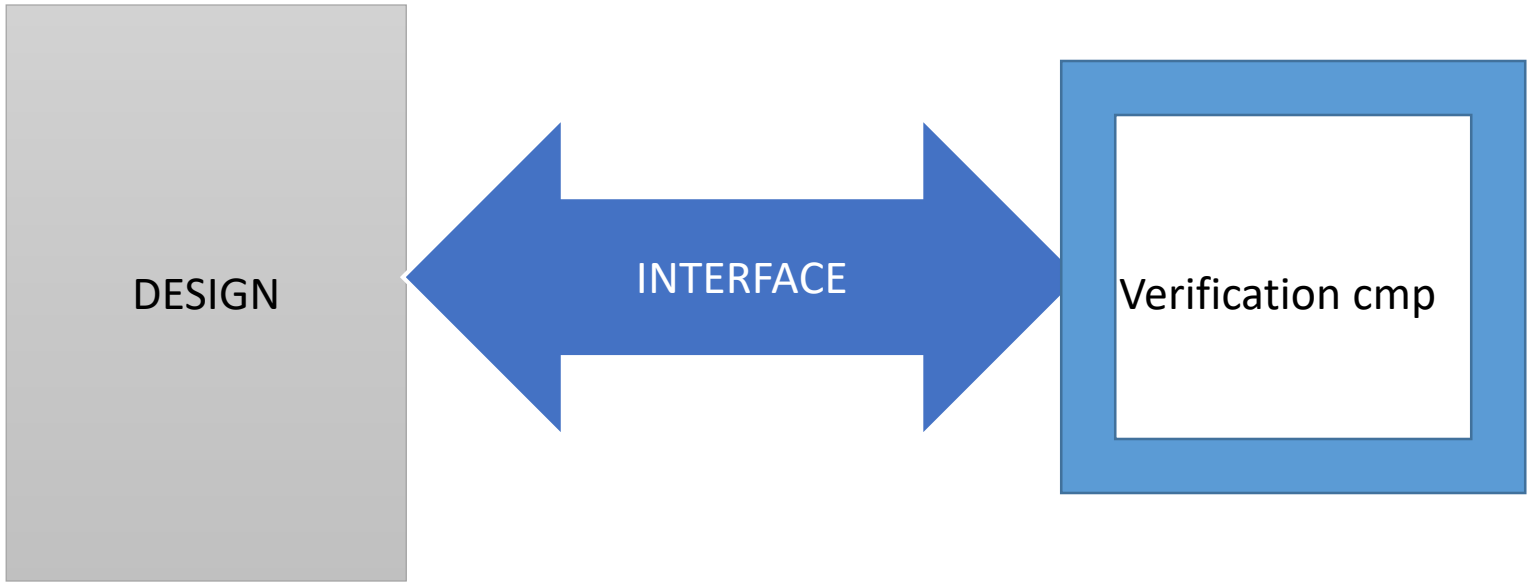
Interface

- Interface is used to encapsulate communication between design blocks, and between design and verification blocks.
- Encapsulating communication between blocks facilitates design reuse. Interfaces can be accessed through ports as a single item.
- Signals can be added to and remove easily from an interface without modifying any port list.
- Interface can contain the connectivity, synchronization, and optionally, the functionality of the communication between two or more blocks.



Interface can be used to transmit requests and receive Reponses

Design

```
module adder (input bit clk,
              input logic [3:0] a, b,
              output logic [4:0] sum);

always @ (posedge clk)
sum= a + b;

endmodule
```

Test Bench

```
program tb (input bit clk,
            input logic [4:0] sum,
            output logic [3:0] a, b);

initial
begin
$monitor ("a=%0d b=%0d sum=%0d", a, b, sum);
forever begin a=$random;  b=$random; #10; end
end

endprogram
```

Top Level

```
module top ();
bit clk=0;
logic [3:0] a, b;
logic [4:0] sum;

always #5 clk=~clk;

adder a0 (.*);        //connect variables to ports that have
tb t0 (.*);           // same name and same data type
endmodule
```

Now in case you have to add one more input c, you have to define c at three place adder, tb and top.

Defining Interface

```
interface intf (input bit clk);

logic [3:0] a, b;
logic [4:0] sum;

endinterface : inf
```

Design

```
module adder (intf inf);

always @ (posedge inf.clk)
inf.sum= inf.a + inf.b;

endmodule : adder
```

Test Bench

```
program tb (inft inf);

initial begin
$monitor (“a=%0d b=%0d sum=%0d”, inf.a, inf.b, inf.sum);
forever begin
inf.a=$random;
inf.b=$random;
#10; end
end

endprogram : tb
```

Top Level

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

always #5 clk=~clk;

intf inf(clk);              //inf is interface instance
adder a0 (inf);
tb t0 (inf);

endmodule
```

Modport

- modport construct is to used to provide direction information for module ports.

```
interface intf (input bit clk);

logic [3:0] a, b;
logic [4:0] sum;
//incase of inout port use wire
modport DUT (input clk, a, b, output sum);
modport TB (input clk, sum, output a, b);

endinterface : intf
```

Design

```
module adder (intf.DUT inf);

always @ (posedge inf.clk)
inf.sum= inf.a + inf.b;

endmodule : adder
```

Test Bench

```
program tb (intf.TB inf);

initial begin
$monitor (“a=%0d b=%0d sum=%0d”, inf.a, inf.b, inf.sum);
forever begin
inf.a=$random;
inf.b=$random;
#10; end
end

endprogram : tb
```

Top Level

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

always #5 clk=~clk;

intf i0 (.*);
adder a0 (.*);
tb t0 (.*);

endmodule
```

Clocking Block

- Clocking block construct identifies clock signals and captures the timing and synchronization requirements of the blocks being modeled.
- Clocking block assembles signals that are synchronous to a particular clock and makes their timing explicit.
- Clocking block separates the timing and synchronization details from the structural, functional, and procedural elements of a test bench.

Clocking Block

- In case of synchronous circuits, input should be sampled just before the clock and output should be driven just after the clock.
- So from test bench point of view, design outputs should be sampled just before the clock and design inputs should be driven just after the clock.
- By default design outputs are sampled at #1step (Prepone Region) and design inputs are driven at #0 (Inactive /Re-Inactive Region).

Clocking Block

```
interface intf(input bit clk);
……
modport TB (input clk,  clocking cb);
clocking cb @ (posedge clk);    //specifying active clock edge
input sum;                          //sampled in prepone region
output a, b;                       //driven in inactive region
endclocking : cb                    //Directions w.r.t Test Bench
……
endinterface
```

Clocking Block

```
……….
clocking cb @ (posedge clk);    //specifying active clock edge
default input #3ns output #2ns;
//Specifying user default for sampling and driving

input sum, reset;       //sampled 3ns before active clock edge
output a, b;            //driven 2ns after active clock edge
endclocking : cb
……….
```

Clocking Block

```
……….
clocking cb @ (posedge clk);    //specifying active clock edge
default input #3ns output #2ns;
//Specifying user default for sampling and driving

input sum;                  //sampled 3ns before active clock edge
output a, b;               //driven 2ns after active clock edge
input negedge reset;   //Overriding default sampling for reset
endclocking : cb
……….
```

Interface

```
interface intf(input bit clk);
logic [3:0] count; logic en;

modport DUT (input clk, en, output count);  //DUT
modport TB (input clk,  clocking cb);        //Test Bench

clocking cb @ (posedge clk);                 //Clocking Block
input count;
output en;
endclocking
endinterface
```

Design

```
module counter (intf.DUT inf);

always @ (posedge inf.clk)
if(inf.en)
inf.count+=1;

endmodule : counter
```

Test Bench

```
program tb (intf.TB inf);

initial begin
@inf.cb;                          //continue on active edge in cb
#3 inf.cb.en<=1;                   // use NBA for signals in cb
##8 inf.cb.en<=0;                 //wait for 8 active edges in cb
repeat(2) @inf.cb;                 //wait for 2 active edges in cb
inf.cb.en<=1;
wait (inf.cb.count==3) inf.cb.en<=0;  //wait for count to become 3
end
endprogram : tb
```

Parameterized Interface

```
interface intf #(size=3, type typ=logic) (input bit clk);
typ [size-1:0] count;
typ en;

modport DUT (input clk, en, output count);  //DUT
modport TB (input clk, count, output en);    //Test Bench

endinterface
```

Virtual Interface

- A virtual interface is a variable that represents an interface instance. This interface is not use to represent physical connections.
- Virtual interface variables can be passed as arguments to tasks, and functions.
- Tasks and functions can modify variables present in a virtual interface which has the same effect as accessing physical interface.
- A virtual interface can be declared as class property, should be initialize before usage.

Virtual Interface

```
interface intf (input bit clk);
logic req, gnt;

always @(posedge clk);        //functionality inside interface
if(req) begin
repeat(2)  @(posedge clk);
gnt<=1; end
else gnt<=0;
endinterface
```

Virtual Interface

```
module test;
bit clk=0;
always #5 clk=~clk;
intf inf(clk);
initial
fork
gen_req(inf);
rec_gnt(inf);
join
endmodule
```

```
task gen_req(virtual intf a);
@(posedge a.clk) a.req<=1;
@(posedge a.clk) a.req<=0;
repeat (5) @(posedge a.clk); a.req<=1;
@(posedge a.clk) a.req<=0; endtask
```

```
task rec_gnt(virtual intf b);
forever begin
@(posedge b.gnt)
$display($time, " grant asserted "); end
endtask
```

Virtual Interface

```
class test;
virtual intf t1 ;

function new(virtual intf t2);
t1=t2;  //initializing virtual
          //interface
endfunction

//task gen_req;
//task rec_gnt;

endclass
```

```
module top;
bit clk=0;
always #5 clk=~clk;
intf inf(clk);
initial begin
test c1= new(inf);
fork
c1.gen_req;
c1.rec_gnt;
join end
endmodule
```

- Create a design for a for 8 bit SIPO(serial in parallel out)shift register

Input din,clk,rst,load

Output dout

Create an interface for above design and try to implement the functionality using interface ports

Create testbench for above design using interface and use top module for mapping

Update your interface with modport and try to update design and testbench

- Create a class and declare the property same as your SIPO design and with the help of methods and virtual interface try provide stimulus to the design via top module