Insert as Section 19.8 – At the end of interfaces

19.8 Virtual interfaces

Virtual interfaces provide a mechanism for separating abstract models and test programs from the actual signals that make up the design. A virtual interface allows the same subprogram to operate on different portions of a design, and to dynamically control the set of signals associated with the subprogram. Instead of referring to the actual set of signals directly, users are able to manipulate a set of *virtual* signals. Changes to the underlying design do not require the code using virtual interfaces to be re-written. By abstracting the connectivity and functionality of a set of blocks, virtual interfaces promote code-reuse.

A virtual interface is a variable that represents an interface instance. The syntax to declare a virtual interface variable is given below.

```
virtual_interface_declaration ::=                                                // from Annex A.2.9
      virtual  [interface] interface_identifier  list_of_virtual_interface_decl ;

list_of_virtual_interface_decl ::=                                               // from Annex A.2.3
                        variable_identifier [ = interface_instance_identifier ]
                      { , variable_identifier [ = interface_instance_identifier ] }

data_type ::=                                                                    // from Annex A.2.2.1
                      …
                      | virtual [interface] interface_identifier
```

Syntax 19-2—Virtual interface declaration.

Virtual interface variables may be passed as arguments to tasks, functions, or methods. A single virtual interface variable may thus represent different interface instances at different times throughout the simulation. A virtual interface must be initialized before it can be used; it has the value **`null`** value before it is initialized. Attempting to use an uninitialized virtual interface shall result in a fatal run-time error.

Only the following operations are directly allowed on virtual interface variables:

- Assignment (=) to:
  - another virtual interface of the same type
  - an interface instance of the same type
  - the special constant **null**
- Equality (==) and inequality (!=) with:
  - another virtual interface of the same type
  - an interface instance of the same type
  - the special constant **null**

Virtual interfaces shall not be used as ports, interface items, or as members of unions.

Once a virtual interface has been initialized, all the components of the underlying interface instance are directly available to the virtual interface via the dot notation. These components may only be used in procedural statements; they may not be used in continuous assignments or sensitivity lists. In order for a wire to be driven via a virtual interface, the interface itself must provide a procedural means to do so. This can be accomplished either via a clocking domain or by including a driver that is updated by a continuous assignment from a register within the interface.

Virtual interfaces can be declared as class properties, which may be initialized procedurally or by an argument to new(). This allows the same virtual interface to be used in different classes. The following example shows how the same transactor class can be used to interact with various different devices.

```
interface SBus;                     // A Simple bus interface
    logic req, grant;
    logic [7:0] addr, data;
endinterface


class SBusTransctor;                // SBus transactor class
    virtual SBus bus;               // virtual interface of type Sbus

    function new( virtual SBus s );
        bus = s;                    // initialize the virtual interface
    endfunction

    task request();                 // request the bus
        bus.req <= 1'b1;
    endtask

    task wait_for_bus();            // wait for the bus to be granted
        @(posedge bus.grant);
    endtask
endclass

module devA( Sbus s ) ...  endmodule     // devices that use SBus
module devB( Sbus s ) ...  endmodule

module top;

    SBus s[1:4] ();                     // instantiate 4 interfaces

    devA a1( s[1] );                    // instantiate 4 devices
    devB b1( s[2] );
    devA a2( s[3] );
    devB b2( s[4] );

    initial begin
        SbusTransactor t[1:4];          // create 4 bus-transactors and bind

        t[1] = new( s[1] );
        t[2] = new( s[2] );
        t[3] = new( s[3] );
        t[4] = new( s[4] );
        // test t[1:4]
    end
endmodule
```

In the preceding example, the transaction class *SbusTransctor* is a simple reusable component. It is written without any global or hierarchical references, and is unaware of the particular device with which it will interact. Nevertheless, the class can interact with any number of devices (4 in the example) that adhere to the interface's protocol.

19.8.1 Virtual interfaces and clocking domains

Clocking domains and interfaces can be combined to represent the interconnect between synchronous blocks. Moreover, because clocking domains provide a procedural mechanism to assign values to both wires and registers, they are ideally suited to be used by virtual interfaces. For example:

```
interface SyncBus( input bit clk );
    wire a, b, c;

    clocking sb @(posedge clk);
        input a;
        output b;
        inout c;
    endclocking

endinterface

typedef virtual SyncBus VI;         // A virtual interface type

task do_it( VI v );                 // handles any SyncBus via clocking sb
    if( v.sb.a == 1 )
        v.sb.b <= 0;
    else
        v.sb.c <= ##1 1;
endtask
```

In the preceding example, interface SyncBus includes a clocking domain, which is used by task do_it to ensure synchronous access to the interface's signals: a, b, and c. Note that changes to the storage type of the interface signals (from wire to register and vice-versa) requires no changes to the task. The interfaces can be instantiated as shown below.

```
module top;
    bit clk;

    SyncBus b1( clk );
    SyncBus b2( clk );

    initial begin
        VI v[2] = { b1, b2 };

        repeat( 20 )
            do_it( v[ $urandom_range( 0, 1 ) ] );
    end
endmodule
```

The top module above shows how a virtual interface can be used to randomly select among a set of interfaces to be manipulated, in this case by the do_it task.

19.8.2 Virtual interfaces modports and clocking domains

The **modport** construct provides direction information for module ports as well as control the use of tasks and functions within particular modules. When using a **modport**, the directions are those seen from the module in which the interface becomes a port.

As shown in the example above, once a virtual interface is declared, its clocking-domain can be referenced using dot-notation. However, this only works for interfaces with no modports. Typically, a device under test and its testbench exhibit **modport** direction. This common case can be handled by including the **clocking** in the corresponding **modport**. The syntax for this is shown below.

modport_declaration ::= **modport** modport_item { **,** modport_item } **;** *// from Annex A.2.9*

modport_item ::= modport_identifier **(** modport_ports_declaration { **,** modport_ports_declaration } **)**

modport_ports_declaration ::=
  modport_simple_ports_declaration
  | modport_hierarchical_ports_declaration
  | modport_tf_ports_declaration
  | modport_clocking_declaration   Note: this is new

modport_clocking_declaration ::= **clocking** clocking_identifier   Note: this is new

Syntax 19-3—Interface modport declaration.

All of the **clocking** constructs used in a **modport** declaration shall be declared by the same interface as is the **modport** itself. Like all **modport** declarations, the direction of the clocking signals are those seen from the module in which the interface becomes a port. The example below shows how **modports** can be used to create both synchronous as well as asynchronous ports. When used in conjunction with virtual interfaces, these constructs facilitate the creation of abstract synchronous models.

```
interface A_Bus( input bit clk );
    wire req, gnt;
    wire [7:0] addr, data;

    clocking sb @(posedge clk);
        input gnt;
        output req, addr;
        inout data;

        property p1; req ##[1:3] gnt; endproperty
    endclocking

    modport DUT ( input clk, req, addr,     // Device under test modport
                  output gnt,
                  inout data );

    modport STB ( clocking sb );            // synchronous testbench modport

    modport TB ( input gnt,                 // asynchronous testbench modport
                 output req, addr,
                 inout data );
endinterface
```

The above interface A_Bus can then be instantiated as shown below:

```
module dev1(A_Bus.DUT b);               // Some device: Part of the design
    ...
endmodule

module dev2(A_Bus.DUT b);               // Some device: Part of the design
    ...
endmodule

program T (A_Bus.STB b1, A_Bus.STB b2 );    // Testbench: 2 synchronous ports
    ...
endprogram

module top;
    bit clk;

    A_Bus b1( clk );
    A_Bus b2( clk );

    dev1 d1( b1 );
    dev2 d2( b2 );

    T tb( b1, b2 );
endmodule
```

And, within the testbench program, the virtual interface can refer directly to the clocking domain.

```
program T (A_Bus.STB b1, A_Bus.STB b2 );    // Testbench: 2 synchronous ports

    typedef virtual A_Bus.STB SYNCTB;

    task request( SYNCTB s );
        s.sb.req <= 1;
    endtask

    task wait_grant( SYNCTB s );
        wait( s.sb.gnt == 1 );
    endtask

    task drive(SYNCTB s, logic [7:0] adr, data );
        if( s.sb.gnt == 0 ) begin
            request(s);                 // acquire bus if needed
            wait_grant(s);
        end
        s.sb.addr = adr;
        s.sb.data = data;
        repeat(2) @s.sb;
        s.sb.req = 0;                   //release bus
    endtask

    assert property (b1.p1);            // assert property from within program

    initial begin
        drive( b1, $random, $random );
        drive( b2, $random, $random );
    end
endprogram
```

The example above shows how the clocking-domain is referenced via the virtual interface by the tasks within the program block.