SystemVerilog Interfaces

Gi-Yong Song
Chungbuk National University, Korea
SystemVerilog

SystemVerilog extends the Verilog language with a powerful interface construct. Interfaces offer a new paradigm for modeling abstraction. The use of interfaces can simplify the task of modeling and verifying large, complex designs.
Interface concepts

The Verilog language connects modules together through module ports. This is a detailed method of representing the connections between blocks of a design that maps directly to the physical connections that will be in the actual hardware.
For large designs, however, using module ports to connect blocks of a design together can become tedious and redundant
Block diagram of a simple design
Verilog module interconnections for a simple design:
module processor (  
    // main_bus ports  
    inout wire [15:0] data,  
    output reg [15:0] address,  
    output reg [3:0] slave_instruction,  
    output reg [3:0] slave_request,  
    output reg bus_grant,  
    output wire mem_read,  
    output wire mem_write,  
    input wire [3:0] slave_instruction,  
    input wire slave_request,  
    input wire bus_grant,  
    input wire data_ready,  
    // other ports  
    input wire [15:0] jump_address,  
    input wire [7:0] instruction,  
    input wire clock,  
    input wire resetN,  
    input wire test_mode  
);  
... // module functionality code  
endmodule

module dual_port_ram (  
    // main_bus ports  
    inout wire [15:0] data,  
    output wire data_ready,  
    input wire [15:0] address,  
    input tri0 mem_read,  
    input tri0 mem_write,  
    // other ports  
    input wire [15:0] program_address,  
    output reg [7:0] data_b  
);  
... // module functionality code  
endmodule

module instruction_reg (  
    output reg [15:0] program_address,  
    output reg [7:0] instruction,  
    input wire [15:0] jump_address,  
    input wire [7:0] next_instruction,  
    input wire clock,  
    input wire resetN  
);  
... // module functionality code  
endmodule

module test_generator (  
    // main_bus ports  
    output wire [15:0] data,  
    output reg [15:0] address,  
    output reg mem_read,  
    output reg mem_write,  
    // other ports  
    input wire clock,  
    input wire resetN,  
    input wire test_mode  
);  
... // module functionality code  
endmodule
Disadvantages of Verilog’s module ports

- Declarations must be duplicated in multiple modules.
- Communication protocols must be duplicated in several modules.
- There is a risk of mismatched declarations in different modules.
- A change in the design specification can require modifications in multiple modules.
Advantages of SystemVerilog interfaces

SystemVerilog adds a powerful new port type to Verilog, called an interface. An interface allows a number of signals to be grouped together and represented as a single port. The declarations of the signals that make up the interface are contained in a single location.
Each module that uses these signals then has a single port of the interface type, instead of many ports with the discrete signals.
/** Interface Definitions ***/

interface main_bus;
  wire [15:0] data;
  wire [15:0] address;
  logic [7:0] slave_instruction;
  logic slave_request;
  logic bus_grant;
  logic bus_request;
  logic slave_ready;
  logic data_ready;
  logic mem_read;
  logic mem_write;
endinterface

/** Top-level Netlist */

module top (input logic clock, resetN, tc
  logic [15:0] program_address, jump_addr
  logic [7:0] instruction, next_instruction
  main_bus bus ( ); // instance of an ir
                                 // (instance name is

processor procl ( // main_bus ports
    .bus(bus), // interface connection
    // other ports
    .jump_address(jump_address),
    .instruction(instruction),
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
    );

slave slave1 ( // main_bus ports
    .bus(bus), // interface connection
    // other ports
    .clock(clock).

// signals for main_bus are defined in just one place

program_address(program_address),
.data_b(next_instruction)
);

test_generator test_gen( // main_bus ports
    .bus(bus), // interface connection
    // other ports
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
    );

instruction_reg ir ( // main_bus ports
    .program_address(program_address),
    .instruction(instruction),
    .jump_address(jump_address),
    .next_instruction(next_instruction),
    .clock(clock),
    .resetN(resetN)
    );

endmodule

SystemVerilog module interconnections using interfaces
module processor (
  // main_bus interface port
  main_bus bus, // interface port
  // other ports
  output logic [15:0] jump_address,
  input logic [7:0] instruction,
  input logic clock,
  input logic resetN,
  input logic test_mode
);

... // module functionality code
endmodule

module slave ( // main_bus interface port
  main_bus bus, // interface port
  // other ports
  output logic clock,
  input logic resetN
);

... // module functionality code
endmodule

module dual_port_ram ( // main_bus interface port
  main_bus bus, // interface port
  // other ports
  input logic [15:0] program_address,
  output logic [7:0] data_b
);

... // module functionality code
endmodule

module test_generator ( // main_bus interface port
  main_bus bus, // interface port
  // other ports
  input logic clock,
  input logic resetN,
  input logic test_mode
);

... // module functionality code
endmodule

module instruction_reg ( // main_bus interface port
  output logic [15:0] program_address,
  output logic [7:0] instruction,
  input logic [15:0] jump_address,
  input logic [7:0] next_instruction,
  input logic clock,
  input logic resetN
);

... // module functionality code
endmodule
In example above, all the signals that are in common between the major blocks of the design have been encapsulated into a single location—the interface declaration called main_bus
The top-level module and all modules that make up these blocks do not repetitively declare these common signals. Instead, these modules simply use the interface as the connection between them.
SystemVerilog interface contents

- The discrete signal and ports for communication can be defined in one location, the interface.

- Communication protocols can be defined in the interface.

- Protocol checking and other verification routines can be built directly into the interface.
Differences between modules and interface(1)

Unlike a module, an interface cannot contain instances of modules or primitives that would create a new level of implementation hierarchy.
Differences between modules and interface(2)

An interface can be used as a module port, which is what allows interfaces to represent communication channels between modules. It is illegal to use a module in a port list.
Differences between modules and interface(3)

An interface can contain modports, which allow each module connected to the interface to see the interface differently.
The interface definition for `main_bus`, with external inputs
interface main_bus (input logic clock, resetN, test_mode);
wire [15:0] data;
wire [15:0] address;
logic [7:0] slave_instruction;
logic slave_request;
logic bus_grant;
logic bus_request;
logic slave_ready;
logic data_ready;
logic mem_read;
logic mem_write;
endinterface

module top (input logic clock, resetN, test_mode);
logic [15:0] program_address, jump_address;
logic [7:0] instruction, next_instruction, data_b;

main_bus bus (*);
processor procl (*);
slave slave1 (*);
instruction_reg ir (*);
test_generator test_gen (*);
dual_port_ram ram (*, data_b(next_instruction));
endmodule

Using interfaces with .* connections to simplify complex netlists
Using interface as module ports

With SystemVerilog, a port of a module can be declared as an interface type, instead of the Verilog input, output or inout port directions.
Explicitly named interface ports

```verilog
module <module_name> (<interface_name> <port_name>);

For example:

```verilog
interface chip_bus;

... endinterface

module CACHE (chip_bus pins, // interface port
  input clock);

... endmodule
```
Generic interface ports

module <module_name> (interface <port_name>);
module RAM (interface pins,
            input   clock);
...
endmodule
module slave (
    // main_bus interface port
    main_bus bus
    // other ports
);
// internal signals
logic [15:0] slave_data, slave_address;
logic [15:0] operand_A, operand_B;
logic mem_select, read, write;
assign bus.address = mem_select? slave_address: 'z;
assign bus.data = bus.slave_ready? slave_data: 'z;
enum logic [4:0] {RESET = 5'b00001,
                 START  = 5'b00010,
                 REQ_DATA = 5'b00100,
                 EXECUTE = 5'b01000,
                 DONE    = 5'b10000} State, NextState;
always_ff @(posedge bus.clock, negedge bus.resetN) begin: FSM
    if (!bus.resetN) State <= START;
    else             State <= NextState;
end
always_comb begin: FSM_decode
    case (State)
        START:
            if (!bus.slave_request) begin
                bus.bus_request = 0;
                nextState = State;
            end
        else begin
            operand_A     = bus.data;
            slave_address = bus.address;
            bus.bus_request = 1;
            nextState     = REQ_DATA;
        end
    ... // decode other states
endcase
end: FSM_decode
endmodule

Referencing signals within an interface
Interface modports

SystemVerilog interfaces provide a means to define different views of the interface signals that each module sees on its interface port.

The definition is made within the interface, using the `modport` keyword. `Modport` is an abbreviation for `module port`.
interface chip_bus (input logic clock, resetN);
logic interrupt_request, grant, ready;
logic [31:0] address;
wire [63:0] data;

modport master (input interrupt_request, 
               input address, 
               output grant, ready, 
               inout data, 
               input clock, resetN);

modport slave (output interrupt_request, 
               output address, 
               input grant, ready, 
               inout data, 
               input clock, resetN);

endinterface
Specifying which modport view to use

- As part of the interface connection to a module instance
- As part of the module port declaration in the module definition
interface chip_bus (input logic clock, resetN);
  modport master (...);
  modport slave (...);
endinterface

module primary (interface pins); // generic interface port
  ...
endmodule

module secondary (chip_bus pins); // specific interface port
  ...
endmodule

module chip (input logic clock, resetN);
  chip_bus bus (clock, resetN); // instance of an interface
  primary i1 (bus.master); // use the master modport view
  secondary i2 (bus.slave); // use the slave modport view
endmodule

Selecting which modport to use at the module instance
interface chip_bus (input logic clock, resetN);
    modport master (...);
    modport slave (...);
endinterface

module primary (chip_bus.master pins); // use master modport ...
endmodule

module secondary (chip_bus.slave pins); // use slave modport ...
endmodule

module chip (input logic clock, resetN);
    chip_bus bus (clock, resetN); // instance of an interface
    primary i1 (bus); // will use the master modport view
    secondary i2 (bus); // will use the slave modport view
endmodule

Selecting which modport to use at the module definition
Using modports to different sets of connections

In a more complex interface between several different modules, it may be that not every module needs to see the same set of signals within the interface.

Modports make it possible to create a customized view of the interface for each module connected.
interface main_bus (input logic clock, resetN, test_mode);
  wire [15:0] data;
  wire [15:0] address;
  logic [7:0] slave_instruction;
  logic     slave_request;
  logic     bus_grant;
  logic     bus_request;
  logic     slave_ready;
  logic     data_ready;
  logic     mem_read;
  logic     mem_write;

modport master (inout data,
                output address,
                output slave_instruction,
                output slave_request,
                output bus_grant,
                output mem_read,
                output mem_write,
                input bus_request,
                input slave_ready,
                input data_ready,
                input clock,
                input resetN,
                input test_mode);

modport slave  (inout data,
                inout address,
                output mem_read,
                output mem_write,
                output bus_request,
                output slave_ready,
                input slave_instruction,
                input slave_request,
                input bus_grant,
                input data_ready,
input clock,
input resetN
);

modport mem (inout data,
output data_ready,
input address,
input mem_read,
input mem_write
);
endinterface

/************************************************************* Top-level Netlist *****************************/
module top (input logic clock, resetN, test_mode);
logic [15:0] program_address, jump_address;
logic [7:0] instruction, next_instruction, data_b;
main_bus     bus  ( .* ); // instance of an interface
processor    proc  ( .bus(bus.master), .* );
slave        slave ( .bus(bus.slave), .* );
instruction_reg ir    ( .* );
test_generator test_gen ( .bus(bus), .* );
dual_port_ram ram    ( .bus(bus.mem), .* ,
                      .data_b(next_instruction) );
endmodule

/*** remainder of netlist and module definitions are not listed — they are similar to example 10-2, but
  clock and resetN do not need to be passed to each module instance as discrete ports. /***/
Interface methods

SystemVerilog allows tasks and functions to be declared within an interface. These tasks and functions are referred to as **interface methods**.
The code for communication between modules is only written once, as interface methods, and shared by each module connected using the interface.
Within each module, the interface methods are called, instead of implementing the communication protocol functionality within the module.
Importing interface methods

If the interface is connected via a modport, the method must be specified using the `import` keyword. The import definition is specified within the interface, as part of a modport definition.
- Import using just the task or function name
- Import using a full prototype of the task or function
Import using a task or function name

```verilog
modport ( import <task_function_name> );
```

An example of using this style is:

```verilog
modport in ( import Read,
             import parity_gen,
             input  clock, resetN );
```
Import using a task or function prototype

\[
\text{modport (import task <task_name> (<task_formal_arguments>) );}
\]

\[
\text{modport (import function <function_name> (<formal_args>) );}
\]

For example:

\[
\text{modport in (import task Read}
\]
\[
\text{(input [63:0] data,}
\]
\[
\text{output [31:0] address),}
\]
\[
\text{import function parity_gen}
\]
\[
\text{(input [63:0] data),}
\]
\[
\text{input clock, resetN);}\]
\]
interface math_bus (input logic clock, resetN);
  int  a_int, b_int, result_int;
  real a_real, b_real, result_real;
  ...
task IntegerRead (output int  a_int, b_int);
  ...
endtask

task FloatingPointRead (output real a_real, b_real);
  ...
endtask

modport int_io (import IntegerRead,
  input  clock, resetN,
  output result_int);

modport fp_io (import FloatingPointRead,
  input  clock, resetN,
  output result_real);
endinterface

/*********************** Top-level Netlist ***********************/
module dual_mmu (input logic clock, resetN);
  math_bus bus_a (clock, resetN); // 1st instance of interface
  math_bus bus_b (clock, resetN); // 2nd instance of interface

  integer_math_unit  i1 (bus_a.int_io);
  // connect to interface using integer types
  floating_point_unit i2 (bus_b.fp_io);
  // connect to interface using real types
endmodule

/*********************** Module Definitions ***********************/
module integer_math_unit (interface io);
  int a_reg, b_reg;
  always @(posedge io.clock)
    begin
      io.IntegerRead(a_reg, b_reg); // call method in
      ...
    end
endmodule

module floating_point_unit (interface io);
  real a_reg, b_reg;
  always @(posedge io.clock)
    begin
      io.FloatingPointRead(a_reg, b_reg); // call method in
      ...
    end
endmodule

Using modports to select alternate methods within an interface
Exporting tasks and functions

SystemVerilog interfaces and modports provide a mechanism to define a task or function in one module, and then export the task or function through an interface to other modules.
Exporting a task or function to the entire interface

A task or function can also be exported to an interface without using a modport.
This is done by declaring an `extern` prototype of the task or function within the interface.
interface chip_bus (input logic clock, resetN);
  logic request, grant, ready;
  logic [63:0] address, data;

  modport master (output request, ...)
    export check);

  modport slave (input request, ...)
    import check);
endinterface

module CPU (chip_bus.master io);

  function check (input parity, input [63:0] data);
    ...
  endfunction
    ...
endmodule

Exporting a function from a module through an interface modport
interface chip_bus (input logic clock, resetN);
  logic request, grant, ready;
  logic [63:0] address, data;

extern function check(input logic parity,
                      input logic [63:0] data);

modport master (output request, ...);

modport slave (input request, ...)
import function check
               (input logic parity,
               input logic [63:0] data));
endinterface

module CPU (chip_bus.master io);

  function check (input logic parity, input logic [63:0] data);
  ... endfunction
  ...
endmodule

Exporting a function from a module into an interface
Restrictions on exporting tasks and functions

It is illegal to export the same task name from two different modules, or two instances of the same module, into the same interface, unless an `extern forkjoin` declaration is used. The multiple export of a task corresponds to a multiple response to a broadcast.
Parameterized interface

Parameters can be used in interfaces to make vector sizes and other declarations within the interface reconfigurable using Verilog’s parameter redefinition constructs.
interface math_bus #(parameter type DTYPE = int)
    (input logic clock);
    DTYPE a, b, result; // parameterized types
...
    task Read (output DTYPE a, b);
        ... // do handshaking to fetch a and b values
    endtask

    modport int_io (import Read,
                        input clock,
                        output result);

    modport fp_io (import Read,
                        input clock,
                        output result);
endinterface

module top (input logic clock, resetN);
    math_bus bus_a(clock); // use int data
    math_bus (#.DTYPE(real)) bus_b(clock); // use real data
    integer_math_unit 11 (bus_a.int_io);
    // connect to interface using integer types
    floating_point_unit 12 (bus_b.fp_io);
    // connect to interface using real types
endmodule // end of module top
Behavioral and Transaction Level Modeling
Transaction level modeling in SystemVerilog

Whereas behavior level modeling raises the abstraction of the block functionality, transaction level modeling raises the abstraction level of communication between blocks and subsystems, by hiding the details of both control and data flow across interfaces.
In SystemVerilog, a key use of the interface construct is to be able to separate the descriptions of the functionality of modules and the communication between them.
Transaction level modeling is a concept, and not a feature of a specific language,
A fundamental capability that is needed for TLMs is to be able to encapsulate the lower level details of information exchange into function and task calls across an interface. The caller only needs to know what data is sent and returned, with the details of the transmission being hidden in the function/task call.
module TopTasks;

logic [20:0] A;
logic [15:0] D;
logic E;
parameter LOWER = 20'h00000;
parameter UPPER = 20'h7fff;
logic [15:0] Mem[LOWER:UPPER];

// Task for reading memory

task ReadMem(input logic [19:0] Address,
             output logic [15:0] Data,
             output logic Error);
    if (Address >= LOWER && Address <= UPPE) begin
        Data = Mem[Address];
        Error = 0;
    end
    else Error = 1;
endtask

// Task for writing memory

task WriteMem(input logic [19:0] Address,
              input logic [15:0] Data,
              output logic Error);
    if (Address >= LOWER && Address <= UPPER) begin
        Mem[Address] = Data;
        Error = 0;
    end
    else Error = 1;
endtask

initial begin
    for (A = 0; A < 21'h100000; A = A + 21'h40000) begin
        fork
            #1000;
            ReadMem(A[19:0], 0, E);
            Join
            if (E) $display("%t bus error on read %h", $time, A);
            else $display("%t read OK %h", $time, A);
        end
        fork
            #1000;
            WriteMem(A[19:0], D, E);
            Join
            if (E) $display("%t bus error on write %h", $time, A);
            else $display("%t write OK %h", $time, A);
        end
    end
endmodule : TopTasks

Simple memory subsystem with read and write tasks
This example gives the following display output:

```
1000 write OK 000000
2000 read OK 000000
3000 write OK 040000
4000 read OK 040000
5000 bus error on write 080000
6000 bus error on read 080000
7000 bus error on write 0c0000
8000 bus error on read 0c0000
```
Transaction level models via interface

This broadcast request with single response can be conveniently modeled with the `extern forkjoin` task construct in SystemVerilog interfaces.
module TopTLM;

Membus Mbus();
Tester T(Mbus);
Memory #(Lo(20'h00000), Hi(20'h3ffff))
    M1(Mbus); // lower addr
Memory #(Lo(20'h40000), Hi(20'h7ffff))
    M2(Mbus); // higher addr

endmodule : TopTLM

// Interface header
interface Membus;

extern forkjoin task ReadMem (input logic [19:0] Address,
                               output logic [15:0] Data,
                               bit Error);

extern forkjoin task WriteMem (input logic [19:0] Address,
                                input logic [15:0] Data,
                                output bit Error);

extern task Request();
extern task Relinquish();

endinterface

Two memory subsystems connected by an interface
module Tester (interface Bus);

logic [15:0] D;
logic E;
int A;

initial begin
for (A = 0; A < 21'h100000; A = A + 21'h40000) begin
fork
    #1000;
    Bus.WriteMem(A[19:0], 0, E);
join
    if (E) $display("%t bus error on write %h", Stime, A);
    else $display("%t write OK %h", Stime, A);

fork
    #1000;
    Bus.ReadMem(A[19:0], D, E);
join
    if (E) $display("%t bus error on read %h", Stime, A);
    else $display("%t read OK %h", Stime, A);
end
endmodule

// Memory Modules
// forkjoin task model delays if OK (last wins)
module Memory (interface Bus);

parameter Lo = 20'h00000;
parameter Hi = 20'h3ffff;
logic [15:0] Mem[Lo:Hi];

task Bus.ReadMem (input logic [19:0] Address, output logic [15:0] Data, output logic Error);
    if (Address >= Lo && Address <= Hi) begin
        #100 Data = Mem[Address];
        Error = 0;
    end
    else Error = 1;
endtask

task Bus.WriteMem (input logic [19:0] Address, input logic [15:0] Data, output logic Error);
    if (Address >= Lo && Address <= Hi) begin
        #100 Mem[Address] = Data;
        Error = 0;
    end
    else Error = 1;
endtask
endmodule

Two memory subsystems connected by an interface
This example gives the following display output:

1000 write OK 000000
2000 read OK 000000
3000 write OK 040000
4000 read OK 040000
5000 bus error on write 080000
6000 bus error on read 080000
7000 bus error on write 0c0000
8000 bus error on read 0c0000
module TopArbTLM;

Membus Mbus();
Tester T1(Mbus);
Tester T2(Mbus);
Arbiter A(Mbus);
Memory #(.Lo(20'h00000), .Hi(20'hffffff)) M1(Mbus);
Memory #(.Lo(20'h40000), .Hi(20'h7ffffff)) M2(Mbus);

endmodule : TopArbTLM

interface Membus; // repeated from previous example

extern forkjoin task ReadMem (input logic [19:0] Address,
output logic [15:0] Data,
        bit Error);

extern forkjoin task WriteMem (input logic [19:0] Address,
input logic [15:0] Data,
output bit Error);

extern task Request();
extern task Relinquish();

endinterface

interface Semaphore #(parameter int unsigned initial_keys = 1);
    int unsigned keys = initial_keys;

    task get(int unsigned n = 1);
        wait (n <= keys);
        keys -= n;
    endtask

    task put (int unsigned n = 1);
        keys += n;
    endtask

endinterface

module Arbiter (interface Bus);
    Semaphore s (); // built-in type would use semaphore s = new;

endmodule

module Tester (interface Bus);

logic [15:0] D;
logic E;
int A;

initial begin : test_block
    for (A = 0; A < 21'h100000; A = A + 21'h40000)
    begin : loop
        fork
            #1000;
            begin
                Bus.Request;
                Bus.WriteMem(A[19:0], 0, E);
                if (E) $display("%t bus error on write %h", $time, A);
                else $display("%t write OK %h", $time, A);
                Bus.Relinquish;
            end
            join
        fork
            #1000;
            begin
                Bus.Request;
                Bus.ReadMem(A[19:0], D, E);
                if (E) $display("%t bus error on read %h", $time, A);
                else $display("%t read OK %h", $time, A);
                Bus.Relinquish;
            end
            join
        end : loop
    end : test_block

endmodule

TLM model with bus arbitration using semaphores
// Memory Modules
// forkjoin task model delays if OK (last wins)
module Memory (interface Bus); // repeated from previous example

parameter Lo = 20'h00000;
parameter Hi = 20'h3ffff;
logic [15:0] Mem[Lo:Hi];

task Bus.ReadMem(input logic [19:0] Address, 
 output logic [15:0] Data, 
 output logic Error);

if (Address >= Lo && Address <= Hi) begin
  #100 Data = Mem[Address];
  Error = 0;
end
else Error = 1;
endtask

task Bus.WriteMem(input logic [19:0] Address, 
 input logic [15:0] Data, 
 output logic Error);

if (Address >= Lo && Address <= Hi) begin
  #100 Mem[Address] = Data;
  Error = 0;
end
else Error = 1;
endtask
endmodule

TLM model with bus arbitration using semaphores
This example gives the following output:

100 write OK 00000000
200 write OK 00000000
1100 read OK 00000000
1200 read OK 00000000
2100 write OK 00040000
2200 write OK 00040000
3100 read OK 00040000
3200 read OK 00040000
4000 bus error on write 00080000
4000 bus error on write 00080000
5000 bus error on read 00080000
References