

# The Benefits of SystemVerilog for ASIC Design and Verification

---

Version 2.5, January 2007

**SYNOPSYS®**

Comments?

Send comments on the documentation by going to <http://solvnet.synopsys.com>, then clicking "Enter a Call to the Support Center."

---

# Copyright Notice and Proprietary Information

Copyright © 2007 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

## Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page: "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of \_\_\_\_\_ and its employees. This is copy number \_\_\_\_\_."

## Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

## Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

## Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSim, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

## Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSiM<sup>plus</sup>, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

## Service Marks (sm)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

All other product or company names may be trademarks of their respective owners.

# Introduction

---

SystemVerilog is an IEEE approved (IEEE P1800-2005) Hardware Description Language. It provides superior capabilities for system architecture, design, and verification. SystemVerilog combines many of the best features of VHDL and Verilog. VHDL users will recognize many of the SystemVerilog constructs, such as enumerated types, records, and multidimensional arrays. Verilog users can reuse existing designs: SystemVerilog is a superset of Verilog so no modification of existing Verilog code is required.

The SystemVerilog language provides three important benefits over Verilog.

1. Explicit design intent – SystemVerilog introduces several constructs that allow you to explicitly state what type of logic should be generated.
2. Conciseness of expressions – SystemVerilog includes commands that allow you to specify design behavior more concisely than previously possible.
3. A high level of abstraction for design – The SystemVerilog interface construct facilitates intermodule communication.

These benefits of SystemVerilog enable you to rapidly develop your RTL code, easily maintain your code, and minimize the occurrence of situations where the RTL code simulates differently than the synthesized netlist. SystemVerilog allows you to design at a high level of abstraction. This results in improved code readability and portability. Advanced features such as interfaces, concise port naming, explicit hardware constructs, and special data types ease verification challenges.

This white paper focuses on these three important benefits of SystemVerilog. A rudimentary knowledge of Verilog is useful to help you understand the examples provided.

---

## Using SystemVerilog With Design Compiler

Like Verilog and VHDL, SystemVerilog leverages the proven technology in Design Compiler, resulting in similar quality of results, but with the added advantages of the SystemVerilog language. The only difference between the SystemVerilog flow and the Verilog or VHDL flow is in the design read step. When synthesizing SystemVerilog code, you must direct Design Compiler to invoke the SystemVerilog reader by using either

```
analyze -f sverilog { files }
elaborate <topdesign>
or
read_sverilog { files }
```

# SystemVerilog Benefits

---

The SystemVerilog language enables explicit design intent, conciseness of design expression, and a high level of design abstraction.

---

## Explicit Design Intent

SystemVerilog provides several constructs that allow you to explicitly state what type of logic should be generated. The use of these constructs enables all of the tools in your design flow to interpret your RTL code in the same way, thus avoiding a situation where the synthesized netlist simulates differently than the RTL. The key constructs that enable explicit design intent are

- `always_comb`, `always_ff`, `always_latch`
  - `unique` and `priority` keywords
- 

### `always_comb`, `always_ff`, `always_latch`

In Verilog, processes are described through the use of the `always` keyword. This keyword is used extensively to describe combinational, sequential, or testbench behavior. The following SystemVerilog example shows how the `always` keyword is used:

```
module no_warn_always(input int0, output reg active);  
  
always@(*)  
    if(int0) active <= 1'b1;  
endmodule
```

A problem with the `always` keyword is that there is no way to explicitly specify whether the intended behavior is combinational or sequential. This problem is resolved by using the SystemVerilog `always_comb` keyword as follows:

```
module warn_always_comb(input logic int0, output logic active);  
  
always_comb  
    if(int0) active <= 1'b1;  
endmodule
```

Using the `always_comb` keyword, you can explicitly direct the simulation, synthesis, or formal verification tool to generate combinational logic. The SystemVerilog parser understands this direction and is able to inform you about the behavior of the resulting netlist, as in the following example:

Warning:. . . Netlist for always\_comb block contains a latch. (ELAB-974)

Similarly, you can use the `always_ff`, and `always_latch` keywords to specify sequential and latch logic, respectively.

---

## unique and priority Keywords

Compiler directives used with the Verilog `case` statement can be interpreted differently by synthesis and verification tools. A synthesis tool may use the compiler directive whereas a simulator may ignore it. Thus the simulation of the RTL may not match the behavior of the synthesized netlist. SystemVerilog provides the `unique` and `priority` keywords that enable you to explicitly state your design intent.

The following Verilog code example shows how you might use compiler directives to tell the synthesis tool that latches and priority logic should not be used:

```
always@(sel, a, b)
  casex (sel) /* synopsys full_case parallel_case*/
    2'b1x: y = a;
    2'bx1: y = b;
  endcase
```

However, in simulation this is not a full and parallel case, so there is potential for a simulation synthesis mismatch, which may result in a design flaw if not formally verified.

SystemVerilog introduces the capability to explicitly specify the behavior of the case statement. This is done using the `unique` and `priority` keywords. These keywords enforce identical behavior in simulation, synthesis, and verification tools, a capability that is not possible using the Verilog or VHDL languages.

The previous Verilog code example can be rewritten using the `unique` keyword as shown in the following SystemVerilog code example:

```
always_comb
  unique casex (sel)
    2'b1x: y = a;
    2'bx1: y = b;
  endcase
```

In this example, simulation will issue runtime warnings if the value of the `sel` signal is `2'b00` or `2'b11`, thus allowing potential problems to be detected early in the design flow. The simulator checks whether the case statement is a full case, parallel case, or both, or none and alert you if there is a problem. The synthesis tool will interpret the `unique` and `priority` keywords in the same manner as the simulation tool, thus greatly reducing the possibility that the netlist will behave differently than the RTL code.

The `unique` and `priority` keywords eliminate the need for compiler directives. You can use Table 1 to find the corresponding SystemVerilog keywords to use in place of existing Verilog compiler directives.

*Table 1 Verilog Compiler Directives Versus SystemVerilog unique and priority Keywords*

<b>SystemVerilog Keywords</b>	<b>Equivalent Synopsys Verilog Compiler Directive</b>	
unique case() without default	full_case	parallel_case
priority case () without default	full_case	
unique case ()		parallel_case
priority case ()	No compiler directive needed	

---

## Conciseness of Expressions

The SystemVerilog language includes commands that allow you to specify design behavior more concisely than previously possible. This results in a reduction of the number of lines of RTL code, improves readability, simplifies debugging, and reduces errors. The following SystemVerilog constructs enable you to write concise expressions:

- Data types on ports
- Multidimensional arrays on ports
- .name and .\* styles
- Assertions

---

## Data Types on Ports

SystemVerilog expands the Verilog support of data types and arrays. SystemVerilog allows user-defined types, enumerated types, structures, and packed and unpacked multidimensional arrays on ports. These improvements simplify your code, making it more readable, flexible, and compact.

The Verilog code in Figure 1 checks for the intensity of red, green, or blue in a pixel. It also checks if two pixels are identical. Each pixel consists of a 32-bit vector of red, green, and blue. R\_pixin, G\_pixin, and B\_pixin together form a single pixel.

*Figure 1 Verilog Pixel Checker*

```

module pixel_checker(input signed [31:0] R_pixin,
input signed [31:0] G_pixin,
input signed [31:0] B_pixin,
input signed [31:0] R_pixin2,
input signed [31:0] G_pixin2,
input signed [31:0] B_pixin2,
input signed [31:0] R_pixin3,
input signed [31:0] G_pixin3,
input signed [31:0] B_pixin3,

output b_pixel,
output r_pixel,
output g_pixel,
output identical_pixels);

assign r_pixel = ((R_pixin == 32'd255) && (G_pixin == 32'd0) &&
(B_pixin == 32'd0));
assign g_pixel = ((R_pixin == 32'd0) && (G_pixin == 32'd255) &&
(B_pixin == 32'd0));
assign b_pixel = ((R_pixin == 32'd0) && (G_pixin == 32'd0) &&
(B_pixin == 32'd255));
assign identical_pixels = ((R_pixin2 == R_pixin3) && (G_pixin2 ==
G_pixin3) && (B_pixin2 == B_pixin3));

endmodule

```

The Verilog code in Figure 1 declares each of the 32-bit vectors for each color of each pixel. The comparison code to produce r\_pixel, g\_pixel, and b\_pixel is lengthy, error-prone, and hard to read.

The equivalent SystemVerilog code for the pixel checker is shown in Figure 2.

*Figure 2 SystemVerilog Pixel Checker*

```

typedef int color;

typedef struct {
color R,G,B;
} pixel;

const pixel RED = {255,0,0};
const pixel BLUE = {0,0,255};
const pixel GREEN = {0,255,0};

module pixel_checker(input pixel pixin, pixin2, pixin3,
output b_pixel, r_pixel, g_pixel, identical_pixels);

assign r_pixel = (pixin == RED);
assign g_pixel = (pixin == GREEN);
assign b_pixel = (pixin == BLUE);
assign identical_pixels = (pixin2 == pixin3);

```

```
endmodule
```

In this SystemVerilog example the 32-bit signed vectors `R_pixin`, `G_pixin`, `B_pixin` are declared in a very concise, readable form using the following user-defined types and structures:

```
typedef int color;

typedef struct {
    color R,G,B;
} pixel;
```

**Note:**

SystemVerilog also makes it possible for you to port such user-defined types from C code to RTL code.

Using SystemVerilog you can create constants for RED, GREEN, and BLUE as follows:

```
const pixel RED = {255,0,0};
const pixel BLUE = {0,0,255};
const pixel GREEN = {0,255,0};
```

By declaring these constants, the comparison operations inside the pixel checker module for `r_pixel`, `g_pixel`, and `b_pixel` become very compact, as shown in the following code:

```
assign r_pixel = (pixin == RED);
assign g_pixel = (pixin == GREEN);
assign b_pixel = (pixin == BLUE);
assign identical_pixels = (pixin2 == pixin3);
```

The user-defined types and structures used in the pixel checker make the SystemVerilog code more compact, more readable, less error-prone, and portable from the C language.

---

## Multidimensional Arrays on Ports

Consider the SystemVerilog frame checker shown in Figure 3. This frame checker counts the number of equal frame fragments. Each fragment consists of RGB colors and the frame checker module has two such frames, `f1` and `f2` as inputs.

*Figure 3 Frame Checker in SystemVerilog*

```
parameter NUM_FRAGMENTS = 10;

typedef struct {
    int R, G, B;
} RGB;

typedef RGB frame_fragment [NUM_FRAGMENTS-1:0];
```

```

module frame_checker(input frame_fragment f1, f2,
    output integer count_equal_fragments);

always comb
    for(int j = 0; j < NUM_FRAGMENTS; j++)
        if(f1[j] == f2[j]) count_equal_fragments++;
        else count_equal_fragments = '0;

endmodule

```

The SystemVerilog code takes advantage of the unpacked array ports to create a parameterized frame fragment. The frame fragment consists of ten fragments that contribute to code compaction. This is shown in the in the following code:

```
typedef RGB frame_fragment [NUM_FRAGMENTS-1:0];
```

Also, notice how the input ports `frame_fragment`, `f1`, and `f2` are defined in the module declaration. In addition, the code uses the auto-increment operator `++` (for incrementing `count_equal_fragments`, syntax that is common in the C language), the `unsized '0` construct (for clearing `count_equal_fragments`), and a local loop variable `j`.

The Verilog implementation of this frame checker is shown in Figure 4.

*Figure 4 Equivalent Code for Frame Checker in Verilog*

```

module frame_checker #(parameter DATA_WIDTH=32*3, SIZE = 10) (
    input [DATA_WIDTH*SIZE-1:0] combined_frame_fragment1,
    input [DATA_WIDTH*SIZE-1:0] combined_frame_fragment2,
    output integer count_equal_frames);

wire [DATA_WIDTH-1:0] f1 [0:SIZE-1];
wire [DATA_WIDTH-1:0] f2 [0:SIZE-1];

integer                j;

genvar I;
generate for (I = 0; I < SIZE; I = I + 1) begin : GEN1
    assign f1[I] = combined_frame_fragment1[I*DATA_WIDTH +:
DATA_WIDTH];
    assign f2[I] = combined_frame_fragment2[I*DATA_WIDTH +:
DATA_WIDTH];
end
endgenerate

always@(*)
    for(j = 0; j < SIZE; j = j + 1)
        if(f1[j] == f2[j])
            count_equal_frames = count_equal_frames + 1;
        else count_equal_frames = 32'b0;
endmodule

```

Verilog allows only one-dimensional packed arrays in ports. Therefore, you need to create the parameterized packed ports called

`combined_frame_fragment1` and `combined_frame_fragment2`. These ports are of the size `DATA_WIDTH*SIZE`. `DATA_WIDTH` is `32*3` to combine R, G, and B into a single packed vector, and `SIZE` is 10, for the number of fragments. In addition, the Verilog code has an extra generate block, `GEN`, where the one-dimensional packed vector ports `combined_frame_fragment1` and `combined_frame_fragment2` are broken into individual words and assigned to `f1` and `f2` using the part-select operator. This makes the Verilog code less intuitive, less readable, and less manageable than the functionally equivalent SystemVerilog code.

---

## **.name and .\***

SystemVerilog adds two new styles of port matching: the `.name` and `.*` styles. These styles allow for a concise representation of port bindings that are not supported in Verilog or VHDL. Using these styles decreases the time you need to spend hooking up ports.

Consider the D flip-flop that is instantiated in the following Verilog example:

```
dff dff1(.clk(clk), .q(q), .d(d), .rst(reset));
```

The names of the four signals `clk`, `q`, `d`, and `reset` are listed twice when using name-based instantiation. For `rst`, the port binding does not match in syntax so you must properly connect up this port.

SystemVerilog allows name-based instantiation to be rewritten more easily as

```
dff dff1(.clk, .d, .q, .rst(reset));
```

Design Compiler will match the instance port name to the matching signal in the module declaration. In cases where the names do not precisely match, you can still explicitly match the names using the syntax `.rst(reset)`.

SystemVerilog allows port matching to be further simplified by using `.*` matching style. The `dff` instantiation becomes even more compact, as shown in the following example:

```
dff dff1(*, .rst(reset));
```

All the signals where the instance port name and the module port names are the same automatically get connected using `.*` in the port connection. Once again, you can explicitly match signals as needed, as with `rst` binding. The `.*` style is also helpful when you add a new signal because you only need to edit module declarations that use the new signal, and not the module instantiations. This makes code change more manageable.

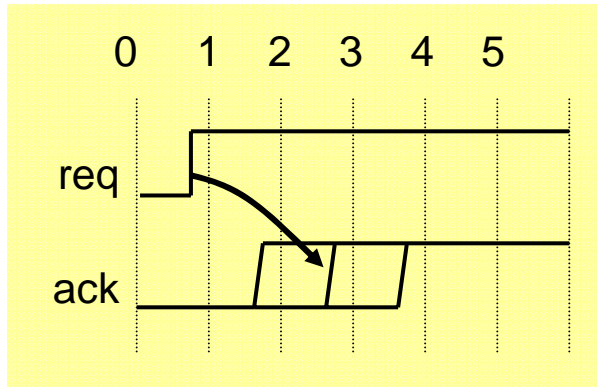
---

## **Assertions**

Assertions are fragments of verification code used to check for correct or illegal behavior of an expected handshake. Assertions add conciseness of

expression to the SystemVerilog language. VHDL users are familiar with Boolean assertions. SystemVerilog adds the power of temporal assertions, enabling assertion-based verification. For example, consider the following timing diagram in Figure 5 that describes the relationship between the request `req` and the acknowledgment `ack` signals.

*Figure 5 req-ack Handshake Protocol - Intended Behavior*



The intended behavior for the handshake is to provide an `ack` response, 1 to 3 cycles after the request occurs. The Verilog code is shown in Figure 6.

*Figure 6 req-ack Handshake Protocol in Verilog*

```

`ifndef SYNTHESIS
always @(posedge req)
begin
  repeat (1) @(posedge clk);
  fork: pos_pos
  begin
    @(posedge ack)
    $display("Assertion Success", $time);
    disable pos_pos;
  end
  begin
    repeat (3) @(posedge clk);
    $display("Assertion Failure", $time);
    disable pos_pos;
  end
  join
end // always
`endif

```

This implementation uses the Verilog `fork` and `join` block to check, on every clock cycle, that an acknowledgement occurs within 1-3 cycles after a request occurs. If the acknowledgement occurs correctly within this range, then the assertion indicates correct behavior, success. If not, a failure message is reported that can help you debug your design. This piece of code is rather lengthy.

Note:

Because this example uses simulation constructs, it is embedded in an ``ifndef SYNTHESIS` and an ``endif`, so that the synthesis tool ignores it.

On the other hand, SystemVerilog allows a compact, readable implementation as shown in Figure 7.

*Figure 7 req-ack Handshake Protocol in SystemVerilog*

```
property p_req_cycle;  
  @(posedge clk) $rose(req) |-> ##[1:3] $rose(ack);  
endproperty
```

Using Verilog, more than 15 lines of code are needed, whereas using SystemVerilog, the implementation of an assertion reduces to 3 lines. Because the assertions are defined by the language, the synthesis tool will automatically ignore them; you are not required to enclose the code in compiler directives.

---

## High Level of Design Abstraction

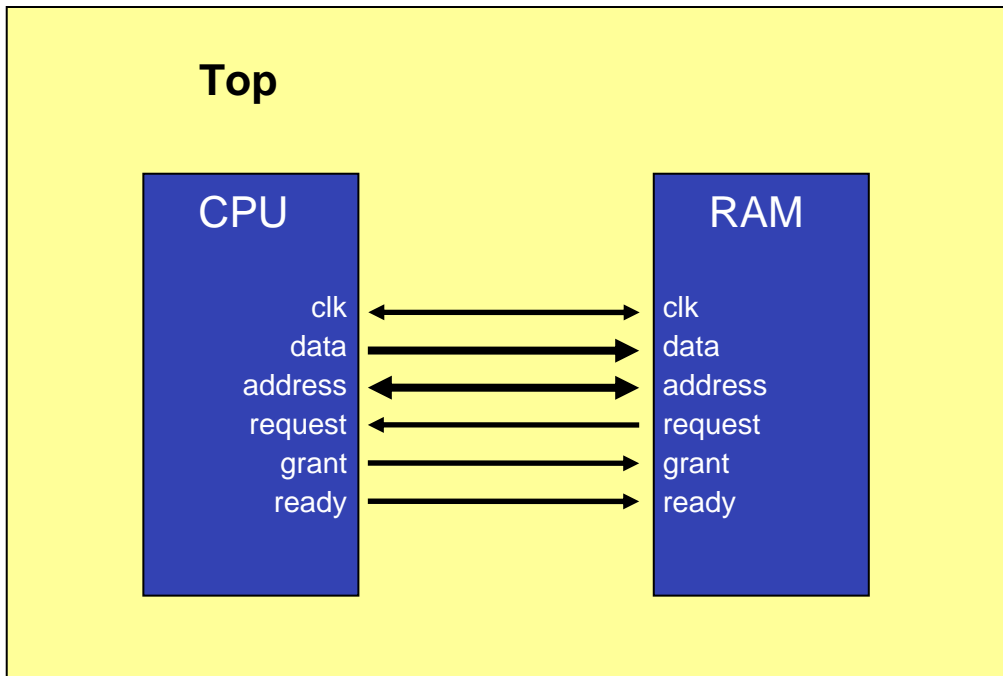
SystemVerilog provides constructs that facilitates intermodule communication using a high-level of abstraction. The advantages of high-level design abstraction using the SystemVerilog interfaces are illustrated in the examples that follow.

---

## Using Interfaces to Bundle Wires and Ports

Interfaces can have functions, combinational and sequential elements, and directional ports and wires. In a typical design interface, such as the one in Figure 8, the signals between the CPU and RAM blocks must be specified in each of the modules and declared at the top level.

Figure 8 CPU and RAM Interconnection Using Ports



The Verilog code for the design illustrated in Figure 8 is shown in Figure 9.

Figure 9 Verilog Code for CPU - RAM

```
module CPU (clk, data, address, request, grant, ready);
inout [47:0] address;
input request, clk;
output [63:0] data;
output grant, ready
...
endmodule

module RAM (clk, data, address, request, grant, ready);
inout [47:0] address;
input clk, grant, ready;
input [63:0] data;
output request;
...
endmodule

module Top;
wire clk;
wire request, grant, ready;
wire [47:0] address;
wire [63:0] data;

CPU CPU(clk, data, address, request, grant, ready);
RAM RAM(clk, data, address, request, grant, ready);
endmodule
```

The SystemVerilog `interface` construct allows a high level of design abstraction. Unidirectional and bidirectional signals, functions, and gates can be bundled into one unit. As shown in Figure 10 and Figure 11, you can define an interface called `chip_bus` to bundle all of the signals between CPU and RAM.

Figure 10 CPU and RAM Interconnection Using Interface `chip_bus`

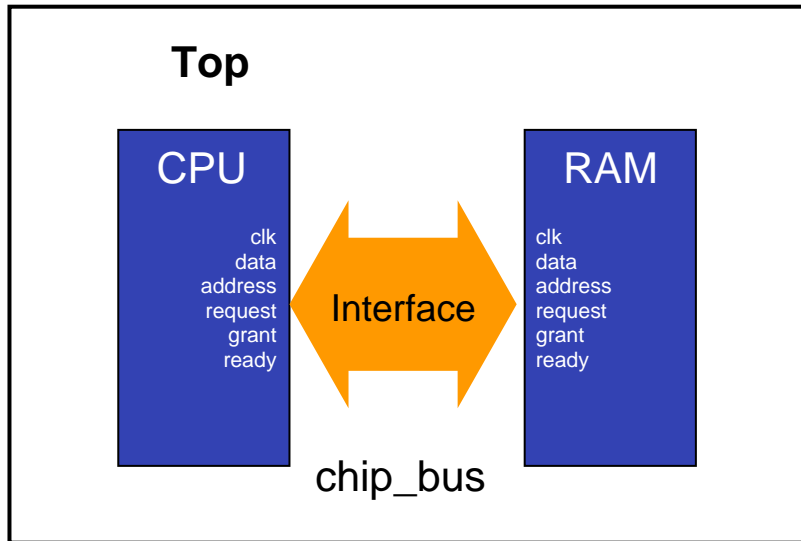


Figure 11 SystemVerilog code for CPU - RAM

```
interface chip_bus ( input wire clk );
logic      request, grant, ready;
wire [47:0] address;
wire [63:0] data;

modport cpu (input request, output grant output ready);
modport ram (output request, input grant, input ready);

endinterface

module CPU (chip_bus.cpu io);
assign cpu.grant = . . .
endmodule

module RAM (chip_bus.ram pins);
assign pins.request = . . .
endmodule

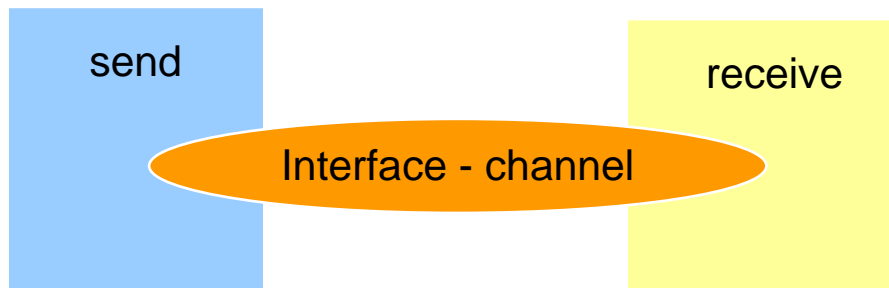
module Top;
wire clk;
chip_bus a(clk);
CPU CPU(a.cpu);
RAM RAM(a.ram);
endmodule
```

The SystemVerilog code is more readable and concise than the Verilog code because it uses an interface. If you want to add a new signal, you only need to add it to the interface; the changes will automatically be propagated to the modules using the interface, thus making it much easier for you to manage your code.

The higher level of abstraction of SystemVerilog enables ease of integration. For example, a communication protocol can be localized within a single interface, making design exploration easier. When interfaces are used, one engineer can own the interface and provide an API that other engineers can use to connect to the bus, hiding the details of the data transfer to and from the bus. This allows the design engineers to focus on designing their modules and the interface engineer to focus on the intermodule communication.

Consider a communication system consisting of send and receive modules connected by a generic interface called “channel” as shown in Figure 12.

*Figure 12 send and receive Modules Connected by a Generic Interface Called “channel”*



If there is a requirement to implement the communication channel using a parallel protocol, you could use the code shown in Figure 13.

*Figure 13 SystemVerilog Code for communication\_system Using Parallel Interface*

```
interface parallel(input bit clk);  
  
wire [31:0] data_bus;  
wire data_valid;  
  
task automatic write(input data_type d);  
    data_bus <= d;  
    data_valid <= 1;  
    @(posedge clk) data_bus <= 'z;  
    data_valid <= 0;  
endtask  
  
task automatic read(output data_type d);  
    while (data_valid != 1)  
        @(posedge clk);  
    d = data_bus;  
    @(posedge clk);  
endtask
```

```

endinterface

module send(input bit clk,interface ch);
initial
begin
  @(posedge clk) ;
  ch.write(27);
  ch.write(42);
  @(posedge clk) ;
  @(posedge clk) ;
  $finish(0);
end
. . .
endmodule

module receive(input bit clk, interface ch);
data_type d;

always
begin
  ch.read(d);
  $display("%t READ a %d",$time, d);
end

endmodule

module communication_system(input bit clk);

parallel channel(clk);

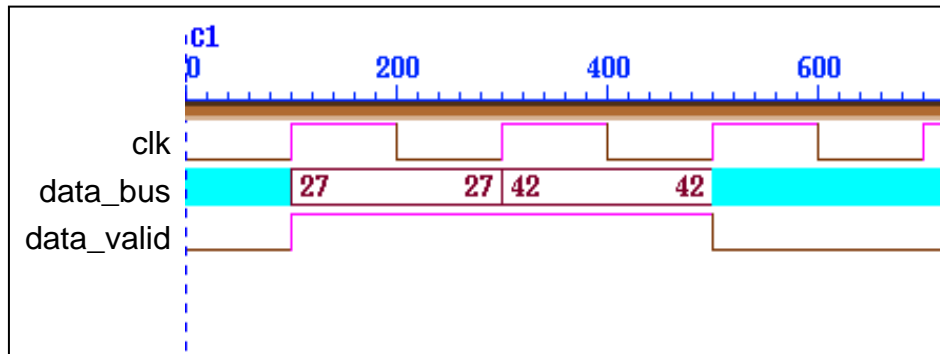
send s(clk, channel);
receive r(clk, channel);

endmodule

```

The timing diagram in Figure 14 shows the data transferred in parallel between the send and receive modules when data\_valid is asserted.

*Figure 14 Parallel Data Communication Between the send and receive Modules*



If there is a requirement to change the parallel interface to a different protocol, such as a serial interface, you can easily accomplish this with SystemVerilog. You need only to instantiate the new serial interface as shown in Figure 15.

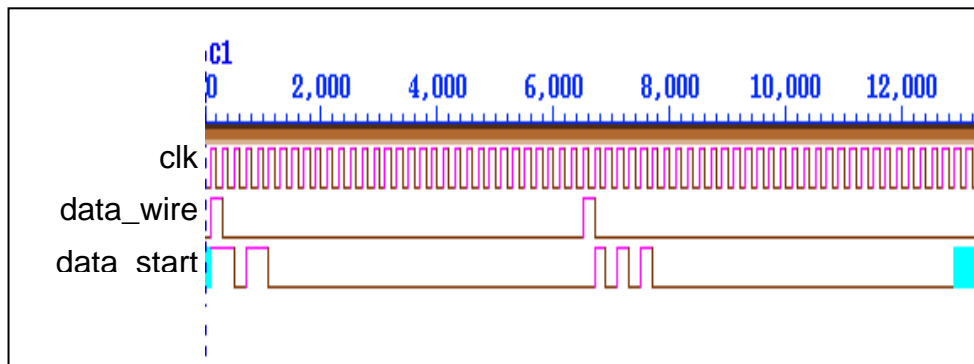
*Figure 15 SystemVerilog Code for communication\_system Using Serial Interface*

```
module communication_system(input bit clk);  
  
    serial channel(clk);  
  
    send    s(clk, channel);  
    receive r(clk, channel);  
  
endmodule
```

The data is now transferred between the `send` and `receive` modules using a serial protocol. The other engineers do not need to know of the changes and can continue with their development.

The updated timing diagram for the serial interface is shown in Figure 16. The data communication between the `send` and `receive` modules occurs serially on `data_wire` when `data_valid` is asserted.

*Figure 16 Serial Data Communication Between the send and receive Modules*



With minimal changes to the SystemVerilog RTL code, the entire communication protocol is modified. This enables easy bus replacement and rapid design exploration.

## Summary

---

SystemVerilog, the IEEE-P1800-2005 standard, bridges the gap between design and verification by providing a single language and environment for hardware design. Through new language constructs that raise the level of abstraction, additional data types, and explicit constructs to specify design intent, SystemVerilog enables you to produce code that is easier to create, debug, maintain, and integrate. Existing Verilog designs can be reused without modification.

Synopsys supports SystemVerilog throughout the design flow from simulation through implementation with support in Design Compiler, Formality, Leda, Magellan, and VCS. For more information on Synopsys SystemVerilog support, send e-mail to [systemverilogdesign@synopsys.com](mailto:systemverilogdesign@synopsys.com).