**module arbiter (**

**2 // Two slashes make a comment line.**

**3 clock , // clock**

**4 reset , // Active high, syn reset**

**5 req\_0 , // Request 0**

**6 req\_1 , // Request 1**

**7 gnt\_0 , // Grant 0**

**8 gnt\_1 // Grant 1**

**9 );**

**10 //-------------Input Ports-----------------------------**

**11 // Note : all commands are semicolon-delimited**

**12 input clock ;**

**13 input reset ;**

**14 input req\_0 ;**

**15 input req\_1 ;**

**16 //-------------Output Ports----------------------------**

**17 output gnt\_0 ;**

**18 output gnt\_1**

**;**

|  |  |
| --- | --- |
| **Bi-Directional Ports Example** | inout read\_enable; // port named read\_enable is bi-directional |

**Vector Signals Example** inout [0:7] address; //port "address" is bidirectional

**Data Type**

Driver that can store a value (example: flip-flop).

Driver that can not store value, but connects two points (example: wire).

The first type of driver is called a reg in Verilog (short for "register"). The second data type is called a wire (for... well, "wire"). You can refer to tidbits section to understand it better.

Wire and\_gate\_output; // "and\_gate\_output" is a wire that only outputs

reg d\_flip\_flop\_output; // "d\_flip\_flop\_output" is a register; it stores and outputs a value

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Operator Symbol** | **Operation Performed** |
| **Arithmetic** | \* | Multiply |
|  | / | Division |
|  | + | Add |
|  | - | Subtract |
|  | % | Modulus |
|  | + | Unary plus |
|  | - | Unary minus |
| **Logical** | ! | Logical negation |
|  | && | Logical and |
|  | || | Logical or |
| **Relational** | > | Greater than |
|  | < | Less than |
|  | >= | Greater than or equal |
|  | <= | Less than or equal |
| **Equality** | == | Equality |
|  | != | inequality |
| **Reduction** | ~ | Bitwise negation |
|  | ~& | nand |
|  | | | or |
|  | ~| | nor |
|  | ^ | xor |
|  | ^~ | xnor |
|  | ~^ | xnor |
| **Shift** | >> | Right shift |
|  | << | Left shift |
| **Concatenation** | { } | Concatenation |
| **Conditional** | ? | conditional |

**// begin and end act like curly braces in C/C++.**

**2 if (enable == 1'b1) begin**

**3 data = 10; // Decimal assigned**

**4 address = 16'hDEAD; // Hexadecimal**

**5 wr\_enable = 1'b1; // Binary**

**6 end else begin**

**7 data = 32'b0;**

**8 wr\_enable = 1'b0;**

**9 address = address + 1;**

**10 end**

**case(address)**

**2 0 : $display ("It is 11:40PM");**

**3 1 : $display ("I am feeling sleepy");**

**4 2 : $display ("Let me skip this tutorial");**

**5 default : $display ("Need to complete");**

**6 endcase**

**while (free\_time) begin**

**2 $display ("Continue with webpage development");**

**3 end**

**module counter (clk,rst,enable,count);**

**2 input clk, rst, enable;**

**3 output [3:0] count;**

**4 reg [3:0] count;**

**5**

**6 always @ (posedge clk or posedge rst)**

**7 if (rst) begin**

**8 count <= 0;**

**9 end else begin : COUNT**

**10 while (enable) begin**

**11 count <= count + 1;**

**12 disable COUNT;**

**13 end**

**14 end**

**15**

**16 endmodule**

**for (i = 0; i < 16; i = i +1) begin**

**2 $display ("Current value of i is %d", i);**

**3 end**

**repeat (16) begin**

**2 $display ("Current value of i is %d", i);**

**3 i = i + 1;**

**4 end**

**assign out = (enable) ? data : 1'bz;**

**Test Benches**

|  |  |  |
| --- | --- | --- |
|  |  | Ok, we have code written according to the design document, now what? |
|  |  | space.gif |
|  |  | Well we need to test it to see if it works according to specs. Most of the time, it's the same we use to do in digital labs in college days: drive the inputs, match the outputs with expected values. Let's look at the arbiter testbench. |
|  |  | space.gif |
|  |  | **1 module arbiter (**  **2 clock,**  **3 reset,**  **4 req\_0,**  **5 req\_1,**  **6 gnt\_0,**  **7 gnt\_1**  **8 );**  **9**  **10 input clock, reset, req\_0, req\_1;**  **11 output gnt\_0, gnt\_1;**  **12**  **13 reg gnt\_0, gnt\_1;**  **14**  **15 always @ (posedge clock or posedge reset)**  **16 if (reset) begin**  **17 gnt\_0 <= 0;**  **18 gnt\_1 <= 0;**  **19 end else if (req\_0) begin**  **20 gnt\_0 <= 1;**  **21 gnt\_1 <= 0;**  **22 end else if (req\_1) begin**  **23 gnt\_0 <= 0;**  **24 gnt\_1 <= 1;**  **25 end**  **26**  **27 endmodule**  **28 // Testbench Code Goes here**  **29 module arbiter\_tb;**  **30**  **31 reg clock, reset, req0,req1;**  **32 wire gnt0,gnt1;**  **33**  **34 initial begin**  **35 $monitor ("req0=%b,req1=%b,gnt0=%b,gnt1=%b", req0,req1,gnt0,gnt1);**  **36 clock = 0;**  **37 reset = 0;**  **38 req0 = 0;**  **39 req1 = 0;**  **40 #5 reset = 1;**  **41 #15 reset = 0;**  **42 #10 req0 = 1;**  **43 #10 req0 = 0;**  **44 #10 req1 = 1;**  **45 #10 req1 = 0;**  **46 #10 {req0,req1} = 2'b11;**  **47 #10 {req0,req1} = 2'b00;**  **48 #10 $finish;**  **49 end**  **50**  **51 always begin**  **52 #5 clock = ! clock;**  **53 end**  **54**  **55 arbiter U0 (**  **56 .clock (clock),**  **57 .reset (reset),**  **58 .req\_0 (req0),**  **59 .req\_1 (req1),**  **60 .gnt\_0 (gnt0),**  **61 .gnt\_1 (gnt1)**  **62 );**  **63**  **64 endmodule**  You could download file arbiter.v [here](http://www.asic-world.com/code/verilog_tutorial/arbiter.v) |
|  |  | space.gif |
|  |  | It looks like we have declared all the arbiter inputs as reg and outputs as wire; well, that's true. We are doing this as test bench needs to drive inputs and needs to monitor outputs. |
|  |  | space.gif |
|  |  | After we have declared all needed variables, we initialize all the inputs to known state: we do that in the initial block. After initialization, we assert/de-assert reset, req0, req1 in the sequence we want to test the arbiter. Clock is generated with an always block. |
|  |  | space.gif |
|  |  | After we are done with the testing, we need to stop the simulator. Well, we use $finish to terminate simulation. $monitor is used to monitor the changes in the signal list and print them in the format we want. |
|  |  | space.gif |
|  |  | req0=0,req1=0,gnt0=x,gnt1=x  req0=0,req1=0,gnt0=0,gnt1=0  req0=1,req1=0,gnt0=0,gnt1=0  req0=1,req1=0,gnt0=1,gnt1=0  req0=0,req1=0,gnt0=1,gnt1=0  req0=0,req1=1,gnt0=1,gnt1=0  req0=0,req1=1,gnt0=0,gnt1=1  req0=0,req1=0,gnt0=0,gnt1=1  req0=1,req1=1,gnt0=0,gnt1=1  req0=1,req1=1,gnt0=1,gnt1=0  req0=0,req1=0,gnt0=1,gnt1=0 |
|  |  | space.gif |
|  |  | I have used Icarus Verilog simulator to generate the above output. |

|  |
| --- |
| **Example - Simple Function** |
| **1 module simple\_function();**  **2**  **3 function myfunction;**  **4 input a, b, c, d;**  **5 begin**  **6 myfunction = ((a+b) + (c-d));**  **7 end**  **8 endfunction**  **9**  **10 endmodule**  file simple\_function.v |
| **Example - Calling a Function** |
| **1 module function\_calling(a, b, c, d, e, f);**  **2**  **3 input a, b, c, d, e ;**  **4 output f;**  **5 wire f;**  **6 `include "myfunction.v"**  **7**  **8 assign f = (myfunction (a,b,c,d)) ? e :0;**  **9**  **10 endmodule**  file function\_calling.v |