```
wire signed [3:-12] a_r, a_i, b_r, b_i;
wire signed [7:-24] p_r, p_i;

real real_a_r, real_a_i, real_b_r, real_b_i,
     real_p_r, real_p_i, err_p_r, err_p_i;

task apply_test ( input real a_r_test, a_i_test,
                           b_r_test, b_i_test );
  begin
    real_a_r = a_r_test; real_a_i = a_i_test;
    real_b_r = b_r_test; real_b_i = b_i_test;
    input_rdy = 1'b1;
    @(negedge clk) input_rdy = 1'b0;
    repeat (5) @(negedge clk);
  end
endtask

multiplier duv ( .clk(clk), .reset(reset),
                 .input_rdy(input_rdy),
                 .a_r(a_r), .a_i(a_i),
                 .b_r(b_r), .b_i(b_i),
                 .p_r(p_r), .p_i(p_i) );

always begin  // Clock generator
  #(t_c/2)       clk = 1'b1;
  #(t_c - t_c/2) clk = 1'b0;
end

initial begin  // Reset generator
  reset <= 1'b1;
  #(2*t_c) reset = 1'b0;
end

initial begin  // Apply test cases
  @(negedge reset)
  @(negedge clk)
  apply_test(0.0, 0.0, 1.0, 2.0);
  apply_test(1.0, 1.0, 1.0, 1.0);
  // further test cases ...
  $finish;
end

assign a_r = $rtoi(real_a_r * 2**12);
assign a_i = $rtoi(real_a_i * 2**12);
assign b_r = $rtoi(real_b_r * 2**12);
assign b_i = $rtoi(real_b_i * 2**12);

always @(posedge clk)  // Check outputs
  if (input_rdy) begin
    real_p_r = real_a_r * real_b_r - real_a_i * real_b_i;
```

(continued)


```
    real_p_i = real_a_r * real_b_i + real_a_i * real_b_r;
    repeat (5) @(negedge clk);
    err_p_r = $itor(p_r)/2**(-24) - real_p_r;
    err_p_i = $itor(p_i)/2**(-24) - real_p_i;
    if (!( -(2.0**(-12)) < err_p_r && err_p_r < 2.0**(-12) &&
           -(2.0**(-12)) < err_p_i && err_p_i < 2.0**(-12) ))
      $display("Result precision requirement not met");
  end

endmodule
```

Within the module, we have instantiated the multiplier module as the device under verification. The instance is connected to testbench nets and variables declared in the module.

Since the multiplier is clocked, we need to generate a clock signal to drive it. This is done by the first always block. It uses a parameter, called t_c, for the clock cycle time. Using a parameter like this allows us to change the clock cycle time without having to chase down every number that varies as a consequence of the change. The block delays for half a clock cycle time, sets the clock to 1, delays a further half a clock cycle time, then sets the clock to 0. (The expression for the duration of the second half clock cycle time is structured so as to compensate for any rounding that may occur in the expression for the first half cycle duration.) After that, the block repeats from the beginning. We also need to generate a reset pulse for the device under verification. This is done by the first initial block. The block sets reset to 1 immediately, then back to 0 after a delay of two clock cycles.

The second initial block stimulates the device under verification with input data. The block uses a task to abstract out the common operations in applying each test-case. Rather than generating fixed-point values directly, the block generates test-case operands of type real on the variables real_a_r, real_a_i, real_b_r and real_b_i. The assignments following the stimulus initial block use the $rtoi conversion function, which converts a real value to an integer value, to assign test-case values to the input inputs of the device under verification. The scaling by 2^{12} is required, since the binary point in each input is 12 places from the right.

Within the stimulus initial block, we must ensure that we generate input stimulus values that meet the timing requirements of the device under verification. The operand values and the input_rdy signal must be set up before a clock edge. The operand values must be held for four cycles while the operation proceeds. To satisfy these requirements, we wait until the first falling clock edge after reset has returned to 0. We do this using the @ notation to delay until the required events occur. The call to the apply_test task then assigns the first test-case operands to