
Lecture 5 Design Simulation with Verilog HDL

5.1 System Tasks

5.2 Compiler Directives

5.3 Functions and Tasks

5.4 Verilog Delay Control

5.5 Automated Simulation Environment and Verilog
Testbench

5.6 Guidelines for RTL Simulation and Verification

5.1 System Tasks

5.2 Compiler Directives

5.3 Functions and Tasks

5.4 Verilog Delay Control

5.5 Automated Simulation Environment and Verilog Testbench

5.6 Guidelines for RTL Simulation and Verification

5.1 System Tasks

- 5.1.1 \$display, \$monitor, and \$fwrite

- \$display is a simple printing task that prints out its variables wherever it is present in the code.

- \$monitor prints out its variables only when they change.

```
1 Syntax: $display("string", variable1, variable2, etc.);
2 $display("display signals a=%b, b=%d, c=%h", a, b, c);
3
4 Syntax: $monitor("string", variable1, variable2, etc.);
5 $monitor("monitor signals a=%b, b=%d, c=%h", a, b, c);
```

- Unlike \$display and \$monitor, which output data to the console, \$fwrite writes the formatted data to a log file

```
1 Syntax: $fwrite(file_id, "string", variable1, etc.);
2
3 initial begin
4     file_id = $fopen("Output.log", "w");
5     if(file_id) begin
6         $fwrite(file_id, "An example of fwrite System Task!");
7         $fclose(file_id);
8     end else begin
9         $display("Error opening file!");
10    end
11 end
```

5.1 System Tasks

- 5.1.2 \$time

- It can return the specific simulation time when used in \$display and/or \$monitor tasks.
 - When data checking fails, use a testbench to print out debug messages with specific simulation time. This helps to locate exactly when and where bugs occur in the simulation waveform.

```
1 $display("ERROR occurs at %d ns: a=%d, a=%b, a=%h",
2           $time, a, a, a);
3 //Printed log: ERROR occurs at 10 ns: a=16, a=10000, a=10
```

5.1 System Tasks

- 5.1.3 \$finish and \$stop
 - \$finish task terminates the simulation
 - \$stop task pauses the simulation.

```
1  initial begin
2      if(rtl_result != golden_result) begin
3          $display("ERROR occurs at %d ns!", $time);
4          #100; $stop;
5      end
6  end
```

5.1 System Tasks

- 5.1.4 \$dumpfile and \$dumpvars
 - \$dumpfile task is used to create a waveform file that records all signal changes during simulation.
 - Siemens ModelSim: .vcd format, Debussy/Verdi: .fsdb format.
 - \$dumpvars task specifies signals should be included in the waveform.
 - ``0" includes all signals in the current module and in any lower-level instantiated modules,
 - ``1" : only signals in the current module.

```
1 Syntax: $dumpfile("dump_file.vcd");
2 Syntax: $dumpvars(level, Module name);
3
4 $dumpfile("dut.vcd");
5 $dumpvars(0, tb.dut);
```

5.1 System Tasks

- 5.1.5 \$readmemh and \$readmemb

- A memory array can be declared as an array of registers that is as wide as the reg declaration and as deep as the number of reg arrays.

- Ex: declare a register array with the width in bytes (reg [7:0]) and the depth of 256 (mem[0:255] from indexes 0 to 255).

```
1  reg [7:0] mem[0:255];
2  Syntax: $readmemh("file name", mem);
3
4  $readmemh("array.txt", mem);
```

00
01
02
...
ff

- The memory array can be loaded from a text data file (array.txt) using the system task \$readmemh (in hexadecimal) or \$readmemb (in binary).

- The task is to load the hexadecimal value 8'h00 into memory location ``mem[0]``, the value 8'h01 into ``mem[1]``, the value 8'h02 into ``mem[2]``, and so on, until the final value 8'hff is loaded into ``mem[255]``.

5.1 System Tasks

- 5.1.6 \$random and \$urandom
 - \$random can generate both signed and unsigned random numbers
 - \$urandom specifically generates unsigned random numbers.
 - ``seed" is an optional argument that specifies the seed for the random number generator.
 - Typically, the \$random and \$urandom system tasks are utilized in combination with initial or always blocks within Verilog testbench.

```
1  initial begin
2      repeat(100) begin
3          $display("Random Number: %0d", $random(02142024));
4          $display("Unsigned Random Number: %0d",
5                  $urandom(02142024));
6      end
7  end
```

5.1 System Tasks

- A testbench Example Using System Tasks

- Design instantiation
- Waveform dumping
- Bus functional model
- Monitor

```
1  module full_adder (output sum    ,
2                      output c_out ,
3                      input  a     ,
4                      input  b     ,
5                      input  c_in  );
6  assign {c_out, sum} = a + b + c_in;
7  endmodule
```

```
1  `timescale 1ns/1ns //reference time/resolution
2  module tb_full_adder();
3  reg  a, b, c_in;    // LHS signals in an initial block
4  wire sum, c_out;    // wire connection between tb and dut
5
6  // ----- DUT Instantiation -----
7  full_adder u_full_adder (.sum    (sum    ),
8                          .c_out   (c_out),
9                          .a       (a      ),
10                         .b       (b      ),
11                         .c_in    (c_in));
```

5.1 System Tasks

- A testbench Example Using System Tasks
 - Design instantiation
 - Waveform dumping
 - Bus functional model
 - Monitor

```
13 // ----- Dump VCD Waveform -----
14 initial begin
15     $dumpfile("full_adder.vcd");
16     $dumpvars(0,tb_full_adder.u_full_adder);
17 end
18
19 // ----- Bus Function Model (BFM) -----
20 initial begin
21     a=1'b0;  b=1'b0;  c_in=1'b0;           // golden result: 00
22     #10;    a=1'b0;  b=1'b0;  c_in=1'b1;   // golden result: 01
23     #10;    a=1'b0;  b=1'b1;  c_in=1'b0;   // golden result: 01
24     #10;    a=1'b0;  b=1'b1;  c_in=1'b1;   // golden result: 10
25     #10;    a=1'b1;  b=1'b0;  c_in=1'b0;   // golden result: 01
26     #10;    a=1'b1;  b=1'b0;  c_in=1'b1;   // golden result: 10
27     #10;    a=1'b1;  b=1'b1;  c_in=1'b0;   // golden result: 10
28     #10;    a=1'b1;  b=1'b1;  c_in=1'b1;   // golden result: 11
29 end
```

5.1 System Tasks

- A testbench Example Using System Tasks

- Design instantiation
- Waveform dumping
- Bus functional model
- Monitor

00
01
01
10
01
10
10
11

```
31 // ----- Monitor -----
32 reg [1:0] mem [0:7];
33 $readmemb("../monitor/GoldenModel.txt", mem);
34
35 integer i;
36 initial begin
37     for (i=0; i<8; i=i+1) begin
38         #5;
39         if({cout,sum}==mem[i]) begin
40             $display ("Data Comparison Passes!");
41         end else begin
42             $display ("ERROR at: %d ns, golden c_out=%b, sum=%b,
43             but DUT c_out=%b, sum=%b when a=%b, b=%b, c_in=%b.",
44             $time, mem0[i][1], mem0[i][0], c_out, sum, a, b, c_in);
45             $stop;
46         end
47         #5;
48     end
49 end
50 endmodule
```

5.1 System Tasks

5.2 Compiler Directives

5.3 Functions and Tasks

5.4 Verilog Delay Control

5.5 Automated Simulation Environment and Verilog Testbench

5.6 Guidelines for RTL Simulation and Verification

5.2 Compiler Directives

- 5.2.1 ``define` vs. parameter
 - A ``define` can be used to define a ``global" text macro that applies to all modules being compiled.
 - A parameter is typically used inside a module to parameterize an attribute for that module alone.
 - Main difference:
 - ``define` is a text substitution mechanism
 - parameters are actual variables that can be assigned different values when instantiating a module.

```
1 // An Example Using `define
2 `define WIDTH 8
3 module adder (input  [`WIDTH-1:0] a,
4                 input  [`WIDTH-1:0] b,
5                 output [`WIDTH-1:0] s);
6     assign s = a+b;
7 endmodule
8
9 // An Example Using parameter
10 module adder #(parameter WIDTH=8) (input  [WIDTH-1:0] a,
11                                     input  [WIDTH-1:0] b,
12                                     output  [WIDTH-1:0] s);
13     assign s = a+b;
14 endmodule
```

5.2 Compiler Directives

- 5.2.2 `ifdef-`else-`endif
 - Conditional compiler directives in Verilog HDL allow optional inclusion of certain lines of code during compilation.
 - Others
 - `ifndef
 - `elsif

```
1  // An Example Using `ifdef-`elsif-`endif
2  module FP_adder (input      clk      ,
3                   input      rst      ,
4                   `ifdef PROJECT_32BIT
5                       input  [31:0]    i32_data ,
6                       output  [31:0]    o32_data
7                   `elsif PROJECT_64BIT
8                       input  [63:0]    i64_data ,
9                       output  [63:0]    o64_data
10                  `elsif PROJECT_128BIT
11                      input  [127:0]   i128_data ,
12                      output  [127:0]   o128_data
13                  `endif
14                                     );
15  endmodule
```

```
1  // Select the source code with a TCL script
2  vlog +define+PROJECT_64BIT \
3  -v ../dut/FP_adder.v
```

5.2 Compiler Directives

- 5.2.3 ``include`

- allows the entire contents of a source file to be inserted into another file during Verilog compilation.

```
1  initial begin
2      $display("%d ns, Load Matrices into Reg Files", $time);
3      $readmemh("../golden/dmx_ieee754.txt", dmx_ram);
4      $readmemh("../golden/tri_ieee754.txt", tri_ram);
5  end
```

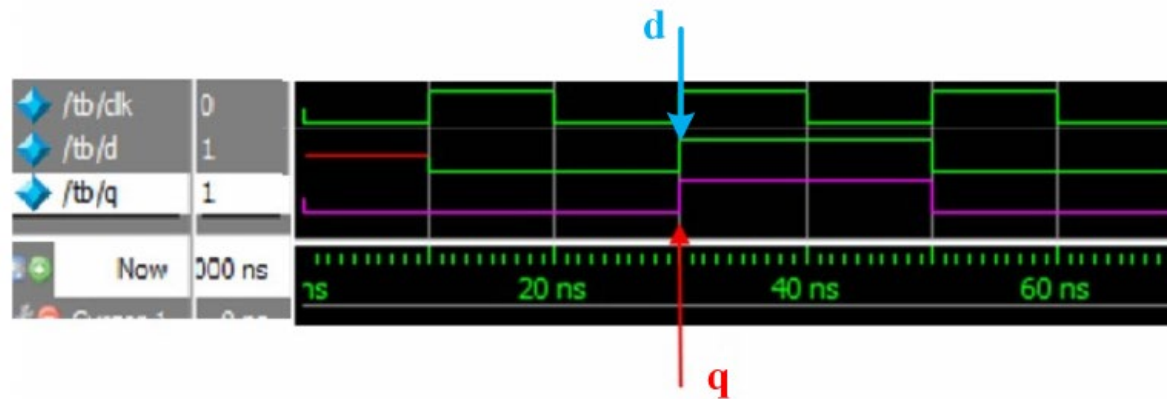
```
1  module tb ();
2  reg [`DMX_MX_WIDTH*32-1:0] dmx_ram[0:`DMX_MX_DPTH-1];
3  reg [`TRI_MX_WIDTH*32-1:0] tri_ram[0:`TRI_MX_DPTH-1];
4
5  `include "load_mem.v"
6  endmodule
```


5.2 Compiler Directives

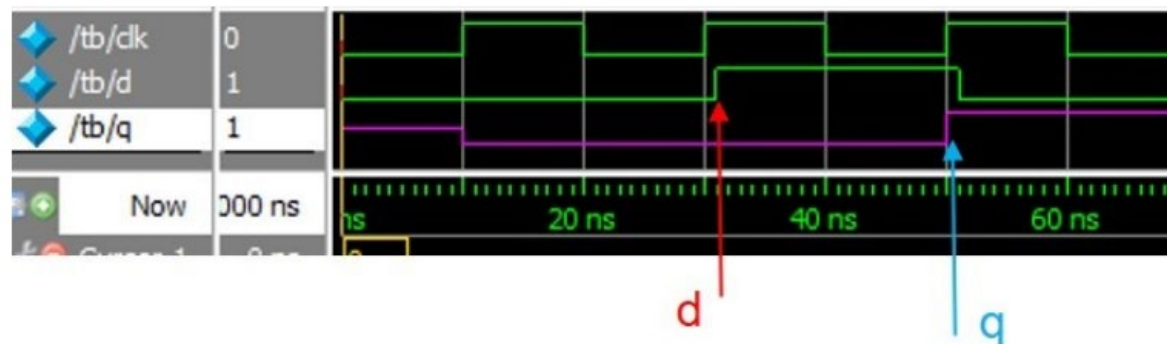
- 5.2.4 `timescale

- `timescale unit/precision
- Example, time unit is 1ns and simulation has 1ps precision

```
1  `timescale 1ns/1ps
2  always @(posedge clk) begin
3      q <= d;
4  end
5
6  always #10 clk = ~clk;
7
8  initial begin
9      d = 0;
10     clk = 0;
11     #30.1 d = 1;
12     #20 d = 0;
13 end
14
15 initial begin
16     d = 0;
17     clk = 0;
18     #31 d = 1;
19     #20 d = 0;
20 end
```



(a) ModelSim Results of Timescale with 30.1 ns



(b) ModelSim Results of Timescale with 31 ns.

5.1 System Tasks

5.2 Compiler Directives

5.3 Functions and Tasks

5.4 Verilog Delay Control

5.5 Automated Simulation Environment and Verilog Testbench

5.6 Guidelines for RTL Simulation and Verification

5.3 Functions and Tasks

- Functions vs. Tasks

- Tasks are invoked in a non-blocking manner, meaning that their execution does not hinder the progress of the rest of the code.
- Conversely, functions are blocking, and they must complete their execution before the remaining code can proceed.

```
1  function [<data_type>] <function_name> (<inputs>);  
2  <variable declration>  
3  <statements>  
4  <return result>  
5  endfunction
```

- 5.3.1 Functions

- Cannot include delay control statements, such as @posedge, @negedge, # delay, or wait(). In other words, a function must be executed with zero time delay.
- Can have one or more input arguments but cannot have any argument declared as output or inout. A function is designed to return a single value to the calling function.
- Cannot enable tasks. In other words, functions cannot be used to trigger or initiate tasks.

5.3 Functions and Tasks

- 5.3.1 Functions
 - A. IEEE 754 Format

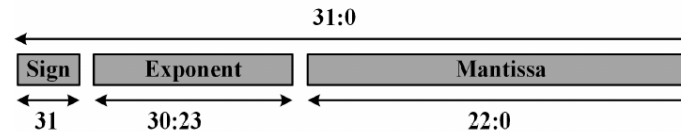


FIGURE 5.2

IEEE 754 Standard (Single Precision): $(-1)^S \times (1.0 + M) \times 2^{(E-127)}$

- B. Function of IEEE 754 to FP Conversion

```
1  function real ieee754_to_fp (input [31:0] ieee754_data);
2  reg      sign      ;
3  reg [7:0] exponent;
4  reg [22:0] mantissa;
5
6  integer int_exp      ;
7  real    mantissa_val ; // Divide by 2^23
8  real    fp_output    ;
9
10 // Extracting sign, exponent, and mantissa bits
11 sign      = ieee754_data[31]      ;
12 exponent  = ieee754_data[30:23];
13 mantissa  = ieee754_data[22:0]    ;
14
15 // Calculating floating-point value
16 int_exp    = exponent-127;
17 mantissa_val = 1.0+(mantissa/8388608.0); // Divide by 2^23
18 fp_output   = (sign?-1:1)*mantissa_val*(2.0**int_exp);
19
20 return fp_output;
21 endfunction
```

5.3 Functions and Tasks

- 5.3.1 Functions
 - C. Call the Function in testbench

```
1 module tb();
2 reg [31:0] golden_result, dut_result;
3 real      golden_real, dut_real, diff_real;
4 real      error_percent;
5 integer    log;
6
7 initial begin
8     log=$fopen("./report.log", "w");
9     wait(data_check_en);
10    golden_real = ieee754_to_fp(golden_result)      ;
11    dut_real    = ieee754_to_fp(dut_result)          ;
12    diff_real   = golden_result_real-dut_result_real;
13    error_percent = diff_real/golden_real*100        ;
14    if(error_percent<5) begin
15        $fwrite (log, "Test Pass!");
16        $fwrite (log, "Error percent: %f%%", error_percent);
17    end else begin
18        $fwrite (log, "Test FAIL!");
19        $fwrite (log, "Error percent: %f%%", error_percent);
20        $fwrite (log, "Golden real: %f", golden_real);
21        $fwrite (log, "DUT real: %f\n", dut_real);
22        $fwrite (log, "Golden Hex: %h", golden_result);
23        $fwrite (log, "DUT Hex: %h\n", dut_result);
24    end
25    $fclose(report);
26 end
27 endmodule
```

5.3 Functions and Tasks

- 5.3.2 Task
 - Can include delay control, such as @posedge, @negedge, #delay, and/or wait().
 - Can have any number of inputs and outputs.
 - Can call other tasks or functions.

```
1  task <task_name>(<inputs> ,  
2                      <outputs>);  
3  <variable declrations>  
4  <statements>  
5  endtask
```

5.3 Functions and Tasks

- 5.3.2 Task
 - A. Tasks of Bus Control Operations

```
1  //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  //
3  //  clk  _---|_---|_---|_---
4  //
5  //  ce   _---|_---|_---
6  //
7  //  wr   _---|_---|_---
8  //
9  //addr  _---|  tk_addr  |---8'h00---|
10 //
11 //wrddata----|  tk_wrddata  |---8'h00---|
12 //
13 //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
14 task task_cpu_write; // a task to mimic cpu write
15 input [7:0] tk_addr   ;
16 input [7:0] tk_wrdata ;
17 begin
18     $display ("%d CPU Write address: %h, data: %h",
19               $time, tk_addr, tk_wrdata);
20     $display ("%d -> Driving ce, wr, wrdata, addr", $time);
21     @(posedge clk)      ;
22     ce                  = 1      ;
23     wr                  = 1      ;
24     addr                = tk_addr ;
25     wrdata              = tk_wrdata ;
26     @(posedge clk)      ;
27     ce                  = 0      ;
28     wr                  = 0      ;
29     addr                = 0      ;
30     wrdata              = 0      ;
31     $display ("=====");
32 end
33 endtask
```

5.3 Functions and Tasks

- 5.3.2 Task
 - A. Tasks of Bus Control Operations

```
1  //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  //
3  //  clk  _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _
4  //
5  //  ce   _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _
6  //
7  //  rd   _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _
8  //
9  //addr  _ _ _ _ | tk_addr | ---8'h00--- |
10 //
11 //rddata-----| rddata  |
12 //
13 //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
14 task task_cpu_read;
15 input  [7:0] tk_addr  ;
16 output [7:0] tk_rddata;
17 begin
18     $display ("%d CPU Read address: %h!", $time, tk_addr)
19     $display ("%d -> Driving ce, rd, addr", $time);
20     @(posedge clk)      ;
21     ce                  = 1      ;
22     rd                  = 1      ;
23     addr                = tk_addr ;
24     @(posedge clk)      ;
25     ce                  = 0      ;
26     rd                  = 0      ;
27     addr                = 0      ;
28     @(negedge clk)      ;
29     tk_rddata = rddata  ;
30     $display ("%g CPU Read data : %h", $time, tk_rddata);
31     $display ("=====");
32 end
33 endtask
```


5.3 Functions and Tasks

- B. Call the Tasks in testbench

```
1 // Call cpu_write and cpu_read tasks
2 module task_cpu_wr_rd();
3 reg      ce, rd, wr  ;
4 reg [7:0] addr, wrdata;
5 wire [7:0] rddata    ;
6 `include "task_cpu_write.v"
7 `include "task_cpu_read.v"
8
9 initial clk = 0;
10 always #5 clk = ~clk;
11
12 initial begin
13     ce      = 1'b0;
14     wr      = 1'b0;
15     rd      = 1'b0;
16     addr    = 8'h0;
17     wrdata  = 8'h0;
18
19     task_cpu_write(8'h01, 8'h10    ); //Call the write task
20     task_cpu_read (8'h01, rddata   ); //Call the read task
21     #5 task_cpu_write(8'h00, 8'h85 );
22         task_cpu_read (8'h00, rddata);
23 end
24 endmodule
```

```
1 // Printed log:
2 0 CPU Write address: 01, data : 10
3 0 -> Driving ce, wr, wdata, addr
4 =====
5 25 CPU Read address : 01
6 25 -> Driving ce, rd, addr
7 40 CPU Read data: 10
8 =====
9 45 CPU Write address: 00, data: 85
10 45 -> Driving ce, wr, wrdata, addr
11 =====
12 65 CPU Read address : 00
13 65 -> Driving ce, rd, addr
14 80 CPU Read data : 85
15 =====
```

5.1 System Tasks

5.2 Compiler Directives

5.3 Functions and Tasks

5.4 Verilog Delay Control

5.5 Automated Simulation Environment and Verilog Testbench

5.6 Guidelines for RTL Simulation and Verification

5.4 Verilog Delay Control

- Delay control: delay and event expressions
 - Delay expressions:
 - Can be used to model the propagation delay of signals and specify timing constraints.
 - However, timing descriptions that use delay expression are not synthesizable.
 - Event expressions:
 - Can be used to specify events that trigger a block of code
 - Can be used to model combinational/sequential logic and to synchronize events between different design/simulation modules.

5.4 Verilog Delay Control

- 5.4.1 Delay Expression
 - Introduces a delay before executing the following statements in Verilog.
 - Its syntax is ``# delay value". The delay value for each statement is based on the time unit defined in the `timescale directive.
 - Nonsynthesizable

```
1  initial begin
2      rst=1'b0; clk=1'b0;
3      #10; rst=1'b1;
4  end
5
6  always #5 clk=~clk;
```

5.4 Verilog Delay Control

- 5.4.2 Event expressions:

- A. posedge and negedge events

- Synthesizable design: flip-flops/registers
 - Testbench: allows the signal ``a" to be synchronous to clock domain.

```
1  initial begin
2      a=1'b0;
3      @(posedge clk);
4      a=1'b1;
5  end
```

- B. level events:

- Synthesizable designs: combinational logics and latches
 - Testbench: In a testbench, a wait statement can be used to add a delay until a condition becomes TRUE. It is commonly used to synchronize signals across different blocks or modules.

```
1  initial begin
2      b=1'b0;
3      wait(a);
4      b=1'b1;
5  end
```

5.1 System Tasks

5.2 Compiler Directives

5.3 Functions and Tasks

5.4 Verilog Delay Control

5.5 Automated Simulation Environment and Verilog Testbench

5.6 Guidelines for RTL Simulation and Verification

5.5 Automated Simulation Environment and Verilog Testbench

- Verilog Design on Testbench
 - Verilog HDL is a versatile language that enables us not only to describe hardware modules but also to create testbenches for the purpose of verifying the functionality of those design modules.
 - The primary objective of a testbench is to simulate the behavior of the design module under specific input stimuli in order to validate its correctness
 - The simulation results can be further analyzed using waveform viewers to visualize signal behavior and ensure that the design functions as intended.
 - Simulators:
 - Synopsys VCS, Cadence NC-Verilog, Siemens EDA Modelsim

5.5 Automated Simulation Environment and Verilog Testbench

- 5.5.1 Structured Project Directory
 - The project directory can grow in complexity, encompassing a multitude of files such as design code, testbenches, constraint files, scripts, golden models, and various files generated during simulation and synthesis.
 - Simulation and debugging often involve repetitive tasks.

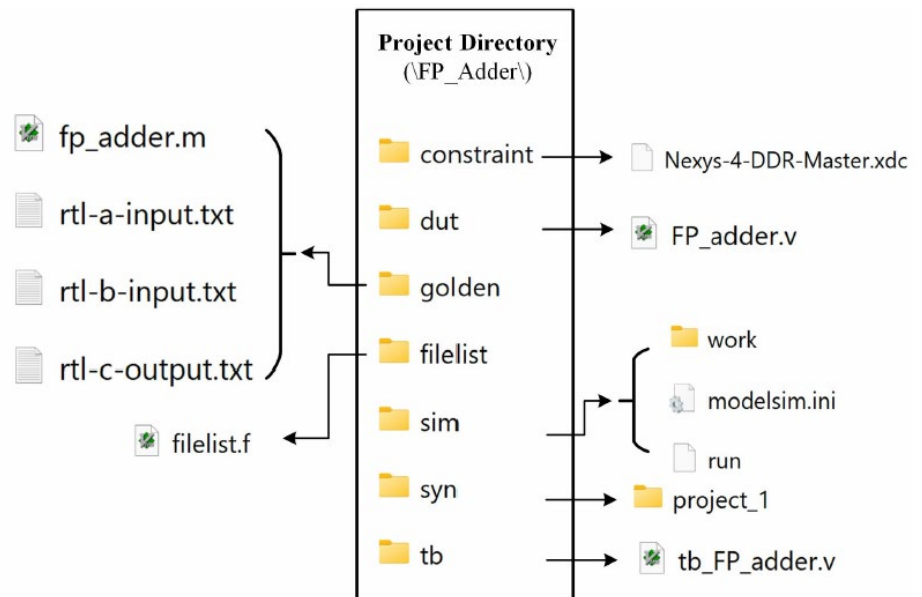


FIGURE 5.3
Structured Project Directory

5.5 Automated Simulation Environment and Verilog Testbench

- Establish A Basic Simulation Environment
 - A. dut, tb, and filelist:
 - ``dut" folder contains all the ``.v" design files.
 - ``tb" folder contains the testbench.
 - To keep track of all design files and the testbench, a ``filelist.f" file is located in the ``filelist" folder.
 - Ex: ``../dut/FP_adder.v" and ``../tb/tb_FP_adder.v".

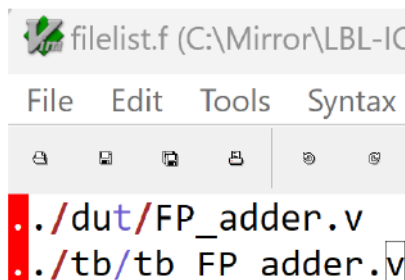


FIGURE 5.4
Filelist Example

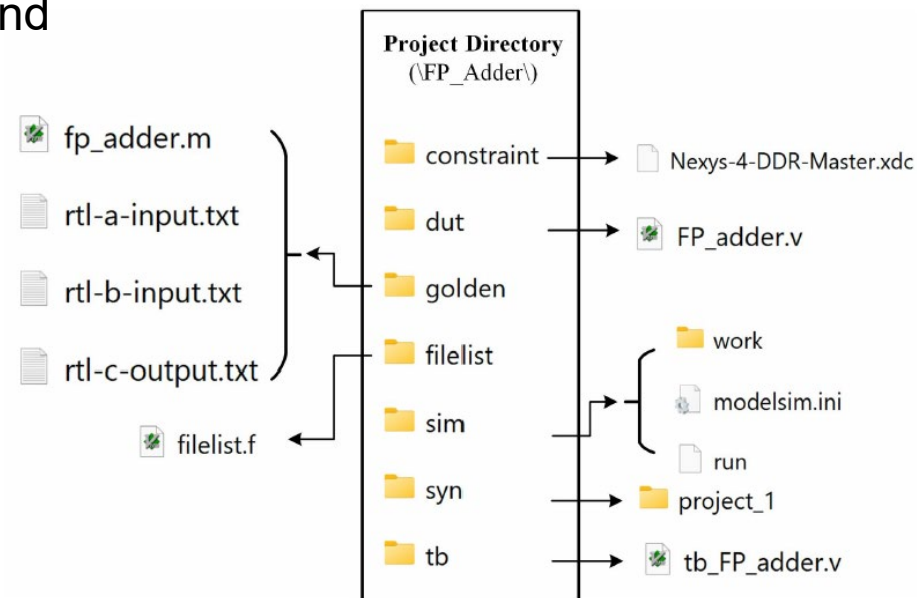


FIGURE 5.3
Structured Project Directory

5.5 Automated Simulation Environment and Verilog Testbench

- Establish A Basic Simulation Environment
 - B. sim and syn:
 - The TCL script for automating simulations is located in the ``sim" folder, while the synthesis project is located in the ``syn" folder.
 - C. constraint:
 - The constraints file, also known as the Xilinx Design Constraints file (XDC file), is used to inform the software about the resources (such as physical pins, switches, buttons, VGA interface, LEDs, etc.) that will be used or connected to the HDL design in the FPGA.
 - Ex: Nexys-4-DDR-Master.xdc for the AMD/Xilinx Nexys 4 FPGA.

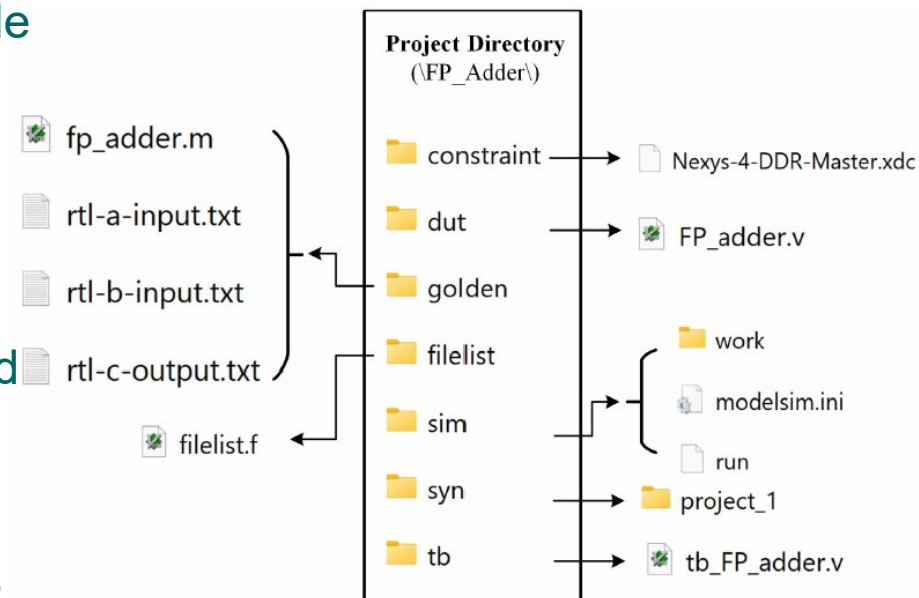
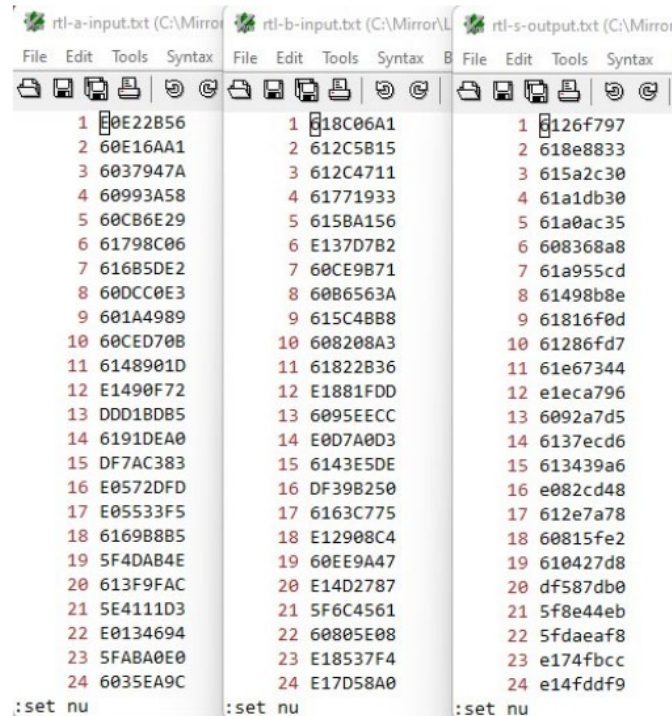


FIGURE 5.3
Structured Project Directory

5.5 Automated Simulation Environment and Verilog Testbench

- Establish A Basic Simulation Environment
 - D. golden:
 - golden models: C, Matlab, etc. can be embed with API/DPI in testbench
 - golden results: inputs -- 0xE0E22B56 and 0x618C06A1 (-1.30377713877e+20 and 3.22877729169e+20), output -- 0x6126F797 (in decimal representation is 1.92500015293e+20)



rtl-a-input.txt	rtl-b-input.txt	rtl-s-output.txt
1 0xE22B56	1 0x18C06A1	1 0x126F797
2 0xE16AA1	2 0x612C5B15	2 0x618E8833
3 0x37947A	3 0x612C4711	3 0x615a2c30
4 0x993A58	4 0x61771933	4 0x61a1db30
5 0xCB6E29	5 0x615BA156	5 0x61a0ac35
6 0x1798C06	6 0xE137D7B2	6 0x08368a8
7 0x1685DE2	7 0x0CE9B71	7 0x61a955cd
8 0x0CC0E3	8 0x0B6563A	8 0x61498b8e
9 0x01A4989	9 0x615C48B8	9 0x61816f0d
10 0x0CED708	10 0x08208A3	10 0x61286fd7
11 0x148901D	11 0x61822B36	11 0x61e67344
12 0xE1490F72	12 0xE1881FDD	12 0xe1eca796
13 0xDD1BD85	13 0x095EECC	13 0x092a7d5
14 0x191DEA0	14 0xE0D7A0D3	14 0x6137ecd6
15 0xDF7AC383	15 0x6143E5DE	15 0x613439a6
16 0xE0572DFD	16 0xDF39B250	16 0xe082cd48
17 0xE05533F5	17 0x6163C775	17 0x612e7a78
18 0x1669B8B5	18 0xE12908C4	18 0x0815fe2
19 0x5F4DAB4E	19 0x0EE9A47	19 0x610427d8
20 0x613F9FAC	20 0xE14D2787	20 0xdf587db0
21 0x5E4111D3	21 0x5F6C4561	21 0x5f8e44eb
22 0xE0134694	22 0x0805E08	22 0x5fdaeaf8
23 0x5FABA0E0	23 0xE18537F4	23 0xe174fbcc
24 0x035EA9C	24 0xE17D58A0	24 0xe14fddf9

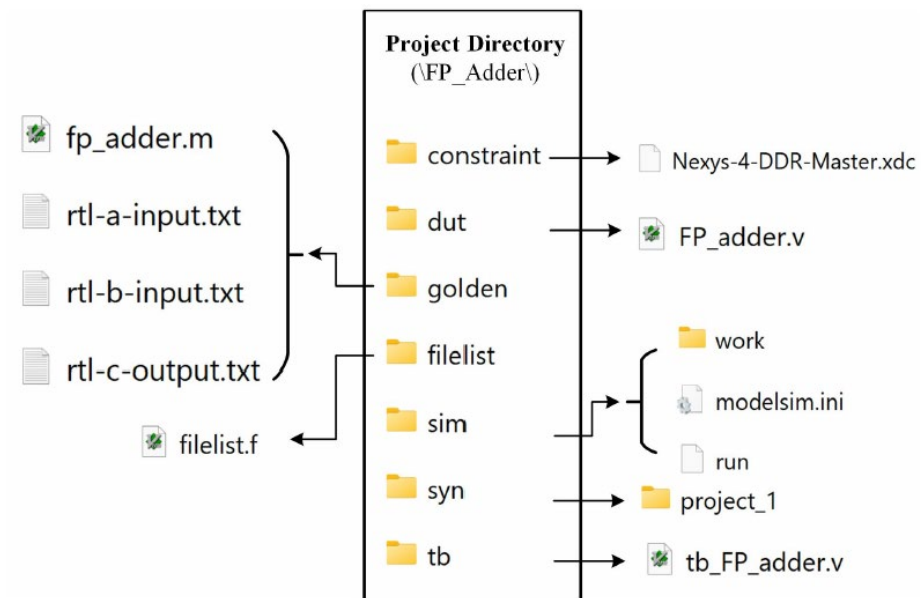


FIGURE 5.5
Golden Results Files.

FIGURE 5.3
Structured Project Directory

5.5 Automated Simulation Environment and Verilog Testbench

- 5.5.2 Automated Simulation Testbench Utilizing BFM and Monitor
 - Design-under-test instantiation (DUT)
 - Bus function models (BFM) to drive and respond to the design
 - Monitor to check the results.

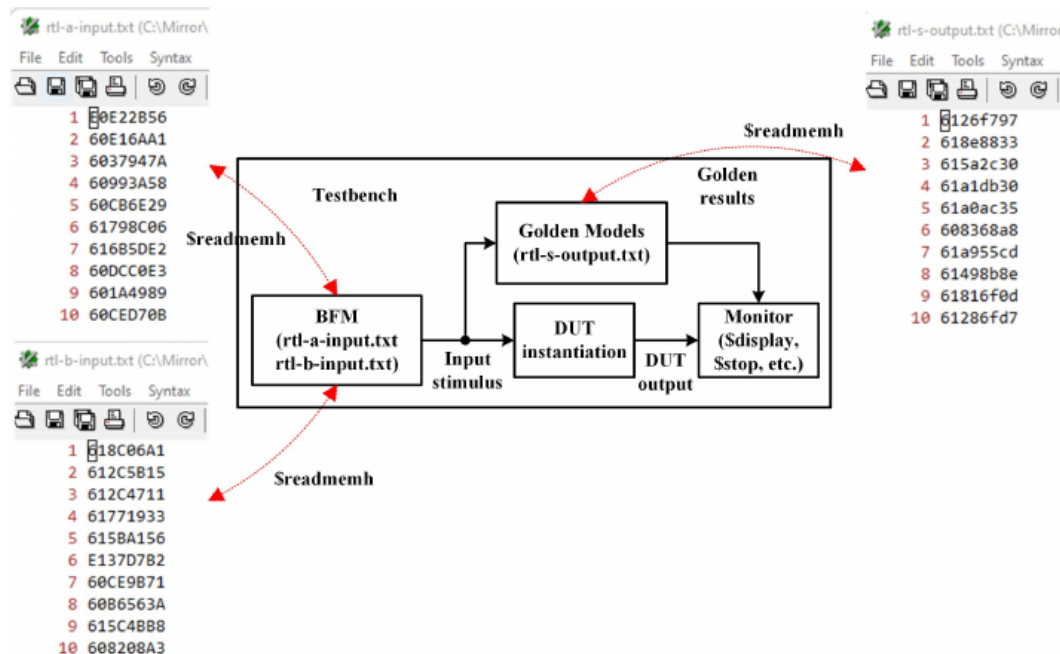


FIGURE 5.6

A Basic Testbench with BFM and Monitor.

5.5 Automated Simulation Environment and Verilog Testbench

- Another option is to embed golden models, such as C code and Matlab code, into the Verilog testbench

- BFM randomizes the input stimulus and feed them into both the DUT and the golden model simultaneously
- The results from the golden model serve as the expected output, which will be compared to the DUT results to determine testing success or failure.
- Since the golden model doesn't incorporate clock timing, the monitor must control the comparison at specific clock cycles.

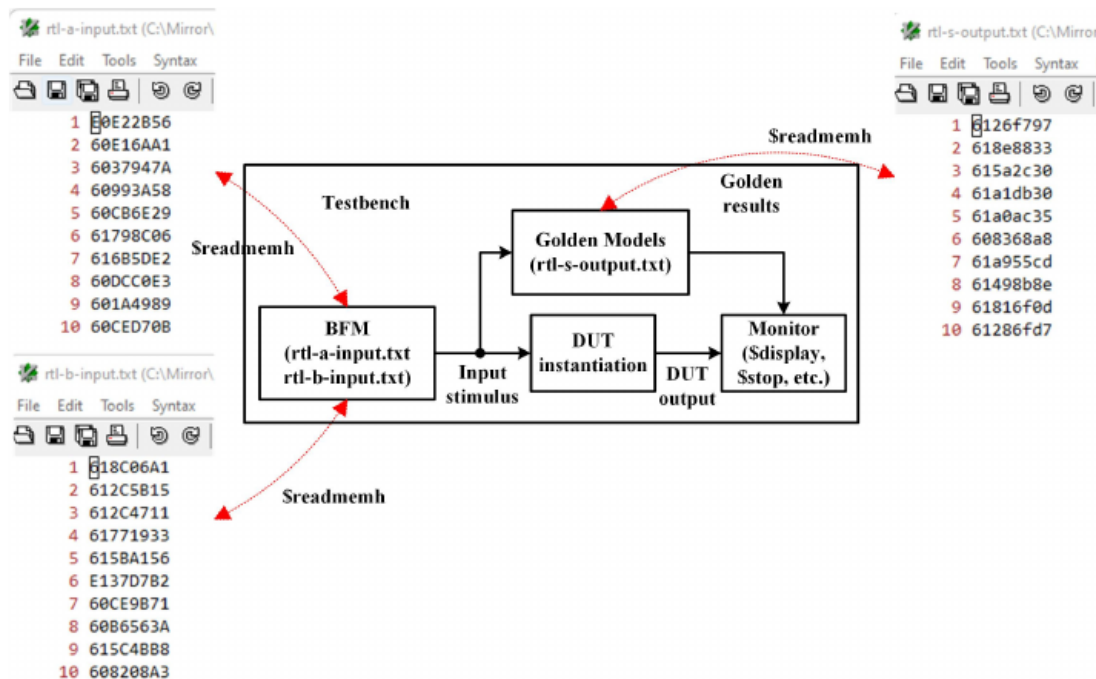


FIGURE 5.6

A Basic Testbench with BFM and Monitor.

5.5 Automated Simulation Environment and Verilog Testbench

- 5.5.3 Verilog Design on Automated Simulation Testbench
 - Ex: The design is a single-precision FP adder that takes one clock cycle for each addition
 - A. Load Memory Array and Instantiate Design-under-Test

```
1  /* -----
2  * This source file contains a simulation testbench for a FP
3  * Adder generated by the Chisel HCL.
4  * Design-under-test: FP_adder
5  * Testbench          : tb_FP_adder
6  * Design IOs         : inputs - clock, reset,
7  *                      io_in_a[31:0], io_in_b[31:0]
8  *                      outputs- io_out_s[31:0]
9  * Latency            : One cycle per FP addition
10 * Precision          : Single precision
11 * Resource Required: 16 Binary Adders (32 x 32 bits)
12 *
13 * Author             : Xiaokun Yang
14 * Date               : June 2022
15 *///-----
16 `timescale 1ns/1ns
17 module tb_FP_adder();
18 parameter TEST_SIZE = 1_000_000;
19 reg      clock      ;
20 reg      reset      ;
21 reg [31:0] io_in_a   ;
22 reg [31:0] io_in_b   ;
23 wire [31:0] io_out_s ;
24
25 // -----
26 // Stimulus Files and Golden File
27 // -----
28 reg [31:0] input_a[TEST_SIZE-1:0] ;
29 reg [31:0] input_b[TEST_SIZE-1:0] ;
30 reg [31:0] output_c[TEST_SIZE-1:0];
31
32 initial begin
33     $readmemh("../golden/rtl-a-input.txt",input_a)  ;
34     $readmemh("../golden/rtl-b-input.txt",input_b)  ;
35     $readmemh("../golden/rtl-s-output.txt",output_s);
36 end
```

5.5 Automated Simulation Environment and Verilog Testbench

- 5.5.3 Verilog Design on Automated Simulation Testbench
 - A. Load Memory Array and Instantiate Design-under-Test
 - B. Bus Functional Model

```
48 // -----
49 // --- Bus Functional Models ----
50 // -----
51 integer i;
52 initial begin
53     reset = 1'b0;
54     clock = 1'b0;
55     #100;
56     reset = 1'b1;
57     @(posedge clock);
58
59     for (i=0; i < TEST_SIZE; i = i+1) begin
60         io_in_a <= input_a[i];
61         io_in_b <= input_b[i];
62         @(posedge clock);
63     end
64 end
65
66 always #10 clock = ~clock;
```

```
38 // -----
39 // --- DUT Instantiation ----
40 // -----
41 FP_adder u_FP_adder (
42     .clock    (clock    ),
43     .reset    (reset    ),
44     .io_in_a  (io_in_a  ),
45     .io_in_b  (io_in_b  ),
46     .io_out_s(io_out_s));
```

5.5 Automated Simulation Environment and Verilog Testbench

• 5.5.3 Verilog Design on Automated Simulation Testbench

– C. Monitor and Test Plan

- IEEE-754 standard, a single-precision FP number is stored in 32 bits and can be divided into a sign bit (the MSB), an 8-bit exponent (the middle 30-23 bits), and a mantissa of 23 bits (the least significant 22-0 bits).
- The sign bit and exponent bits must be exactly the same.

```
72 wire sign_check = (io_out_s[31]==output_s[j][31])      ;  
73 wire exp_check  = (io_out_s[30:23]==output_s[j][30:23]);
```

- The comparison of mantissa bits involves four tolerance levels:
 - m: error tolerance #0 requires all the mantissa bits to be the same;
 - n: error tolerance #1 requires only the least significant 22-4 bits to be the same;
 - o: error tolerance #2 requires the least significant 22-8 bits to be the same
 - p: error tolerance #3 requires the least significant 22-16 bits to be the same.

5.5 Automated Simulation Environment and Verilog Testbench

- 5.5.3 Verilog Design on Automated Simulation Testbench
 - C. Monitor and Test Plan
 - The comparison of mantissa bits involves four tolerance levels:
 - m: error tolerance \#0 requires all the mantissa bits to be the same;
 - n: error tolerance \#1 requires only the least significant 22-4 bits to be the same;
 - o: error tolerance \#2 requires the least significant 22-8 bits to be the same
 - p: error tolerance \#3 requires the least significant 22-16 bits to be the same.

```
68 // -----
69 // -----Monitor -----
70 // -----
71 integer j, m, n, o, p, q, r, s;
72 wire sign_check = (io_out_s[31]==output_s[j][31]) ;
73 wire exp_check  = (io_out_s[30:23]==output_s[j][30:23]);
74 wire mant_check = (io_out_s[22:0]==output_s[j][22:0]) ;
75 wire mant_22_4  = (io_out_s[22:4]==output_s[j][22:4]) ;
76 wire mant_22_8  = (io_out_s[22:8]==output_s[j][22:8]) ;
77 wire mant_22_16 = (io_out_s[22:16]==output_s[j][22:16]);
78 wire mant_3_0   = (io_out_s[3:0]==output_s[j][3:0]) ;
79 wire mant_7_0   = (io_out_s[7:0]==output_s[j][7:0]) ;
80 wire mant_15_0  = (io_out_s[15:0]==output_s[j][15:0]) ;
```

5.5 Automated Simulation Environment and Verilog Testbench

• 5.5.3 Verilog Design on Automated Simulation Testbench

– C. Monitor and Test Plan

```
82 initial begin
83     m = 0 ;
84     n = 0 ;
85     o = 0 ;
86     p = 0 ;
87     q = 0 ;
88     r = 0 ;
89     s = 0 ;
90     wait (reset);
91     repeat (2) @(negedge clock);
92     for (j=0; j < TEST_SIZE; j = j+1) begin
93         $display("%d ns, a=%h, b=%h, golden s=%h, dut s=%h",
94 $time, input_a[i-1], input_b[i-1], output_s[j], io_out_s);
95         case({sign_check, exp_check, mant_check})
96             3'b111: begin
97                 m=m+1;
98             end
99             3'b110: begin
100                 if(mant_22_4 & ~mant_3_0) begin
101
102                     end else if(mant_22_8 & ~mant_7_0) begin
103                         o=o+1;
104                     end else if(mant_22_16 & ~mant_15_0) begin
105                         p=p+1;
106                     end
107                 end
108                 default: begin
109                     if(~sign_check) begin
110                         $display("Sign bit different!, j=%d", j);
111                         q=q+1;
112                     end else if(~exp_30_23) begin
113                         $display("Exponent are different!");
114                         r=r+1;
115                     end else if(~mant_22_16) begin
116                         $display("Matanssa[22:16] are different!");
117                         s=s+1;
118                     end
119                 end
120             endcase
121             @(negedge clock);
122         end
```

5.5 Automated Simulation Environment and Verilog Testbench

- 5.5.3 Verilog Design on Automated Simulation Testbench
 - D. Simulation Log:

```

124 $display("-----FP adder simulation summary-----");
125 $display("%d cases pass, %d fail!", m+n+o+p, q+r+s);
126 $display("----FP adder simulation passed cases---");
127 $display("%d cases exactly the same!", m);
128 $display("%d cases: different mantissa[3:0]!", n);
129 $display("%d cases: different mantissa[7:0]!", o);
130 $display("%d cases: different mantissa[15:0]!", p);
131 $display("----FP adder simulation failed cases---");
132 $display("%d cases: different sign bit!", q);
133 $display("%d cases: different exponent bits!", r);
134 $display("%d cases: different mantissa[22:16]!", s);
135 $display("-----FP adder simulation summary-----");
136 end
137 endmodule

```

```

# At 19999800ns, a=e0b774e9, b=df7e60f6, expected s=e0d74108, dut s=e0d74107
# At 19999820ns, a=60881f4b, b=61021bcc, expected s=61462b72, dut s=61462b71
# At 19999840ns, a=5e823d3c, b=e02ed887, expected s=e01e90e0, dut s=e01e90e0
# At 19999860ns, a=604ffee7, b=6123blcc, expected s=6157b186, dut s=6157b185
# At 19999880ns, a=e0fe6e88, b=e18642a3, expected s=e1c5de45, dut s=e1c5de45
# At 19999900ns, a=608b1fce, b=60b33823, expected s=611f2bf8, dut s=611f2bf8
# At 19999920ns, a=e0fb79de, b=5f70226c, expected s=e0dd7590, dut s=e0dd7591
# At 19999940ns, a=dedf091c, b=e1801d30, expected s=e1839954, dut s=e1839954
# At 19999960ns, a=e15a5517, b=e0dfc219, expected s=e1a51b12, dut s=e1a51b11
# At 19999980ns, a=e0510cbb, b=61709f39, expected s=613c5c0a, dut s=613c5c0b
# At 20000000ns, a=618ab8f0, b=60a765f7, expected s=61b4926e, dut s=61b4926d
# At 20000020ns, a=618bb7b1, b=60bee6a6, expected s=61bb715a, dut s=61bb715a
# At 20000040ns, a=60ba9f05, b=610be16b, expected s=616930ee, dut s=616930ed
# At 20000060ns, a=610c0fcf, b=e005c152, expected s=60d53ef5, dut s=60d53ef6
# At 20000080ns, a=e14elc6f, b=601485b2, expected s=e128fb02, dut s=e128fb03
# At 20000100ns, a=5f8c0cb7, b=e0676290, expected s=e0215c34, dut s=e0215c35
# At 20000120ns, a=6114bc00, b=e0ac7466, expected s=607a0734, dut s=607a0734
# ----- FP adder design simulation summary -----
# For 1,000,000 test cases, there are 1,000,000 test cases pass, 0 test cases fail!
# ----- FP adder design simulation passed cases -----
# For 1,000,000 passed test cases, 659008 test cases exactly the same!
# For 1,000,000 passed test cases, 297848 test cases with different 3-0 mantissa bits!
# For 1,000,000 passed test cases, 37530 test cases with different 7-0 mantissa bits!
# For 1,000,000 passed test cases, 5606 test cases with different 15-0 mantissa bits!
# ----- FP adder design simulation fail cases -----
# For 1,000,000 failed test cases, 0 test cases with different sign bit!
# For 1,000,000 failed test cases, 0 test cases with different exponent bits!
# For 1,000,000 failed test cases, 0 test cases with different 22-16 mantissa bits!
# -----FP adder design simulation summary -----

```

FIGURE 5.7
Simulation Log

5.1 System Tasks

5.2 Compiler Directives

5.3 Functions and Tasks

5.4 Verilog Delay Control

5.5 Automated Simulation Environment and Verilog Testbench

5.6 Guidelines for RTL Simulation and Verification

5.6 Guidelines for RTL Simulation and Verification

Guidelines for RTL Simulation and Verification

- **Functional Verification:** Functional verification (also known as RTL verification) is used to verify the RTL design features without testing timing constraints. The practical circuits have timing delays and timing requirements, which cannot be simulated during RTL verification but exist in physical chips.
- **Test Plan:** Develop a comprehensive test plan that incorporates both direct and random testing strategies. In this plan, RTL designers primarily handle direct testing to validate fundamental design features, while verification engineers concentrate on random testing to accumulate coverage data and pinpoint corner and exceptional testing scenarios. For intricate design projects, uncovering exceptional cases poses a considerable challenge in terms of functional verification, and the verification team plays a critical role in addressing this challenge.
- **Reusable Testbench:** Create a testbench with reusability and scalability in mind, facilitating the straightforward inclusion of new test cases. To optimize the verification process for advanced IC designs, consider employing the SystemVerilog language in conjunction with the UVM methodology. This approach enhances efficiency, scalability, and overall productivity throughout the verification process.
- **Testbench Monitor:** Incorporate a monitor or scoreboard within the testbench to automate result verification, eliminating the need for manual checks. Monitors play a pivotal role in Verilog testbenches by enabling the real-time observation and capture of signals from the design-under-test. This automation ensures comprehensive verification of the design's functionality and behavior.

5.6 Guidelines for RTL Simulation and Verification

- **Bus Functional Model and Golden Models:** Integrate a bus functional model into the verification process for providing random data inputs to RTL designs. You can achieve this by preparing data files alongside golden models. Alternatively, consider the direct integration of golden models into the Verilog testbench using the DPI. This approach offers the advantage of seamlessly incorporating a wide range of programming languages into the Verilog testbench, thereby boosting flexibility and versatility in your verification process.
- **Printed Log and Dump Waveform:** Ensure that error messages, along with related signals and precise simulation timestamps, are logged to a designated log file. Furthermore, when conducting random tests, consider dumping waveform data and specifying a unique simulation seed ID. This seed ID facilitates the reproduction of the same random data in subsequent simulations, significantly assisting in efforts related to reproducibility and debugging.
- **Signal Initialization:** Ensure that all design inputs are properly initialized, either with zeros or ones, to avoid unknown signals in the simulation waveform.
- **Regression Testing:** After making changes to the RTL code, it's essential to conduct regression tests to identify potential impacts on other design features. Regression testing should encompass comprehensive verification activities, including coverage analysis, to ensure thorough validation of the design-under-test.