Lecture 5 Design Simulation with Verilog HDL

Outline

Outline

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.

```
Syntax: $display(``string'', variable1, variable2, etc.);
$display(``display signals a=%b, b=%d, c=%h'', a, b, c);

Syntax: $monitor(``string'', variable1, variable2, etc.);
$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

```
Syntax: $fwrite(file_id, "string", variable1, etc.);

initial begin
  file_id = $fopen("Output.log", "w");
  if(file_id) begin
    $fwrite(file_id, "An example of fwrite System Task!");
    $fclose(file_id);
  end else begin
     $display("Error opening file!");
  end
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.

```
$display(``ERROR occurs at %d ns! a=%b, b=%b, c=%h'',
                                    $time, a, b, c);
// Printed log: ERROR is detected at 10 ns: a=1, b=0, c=1
```

5.1 System Tasks

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

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

- 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.

```
Syntax: $dumpfile("dump_file.vcd");
Syntax: $dumpvars(level, Module name);

$dumpfile("dut.vcd");
$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).

```
reg [7:0] mem[0:255];
Syntax: $readmemh("file name", mem);

$readmemh("array.txt", mem);
```

```
01
02
03
...
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'h01 into memory location ``mem[0]'', the value 8'h02 into ``mem[1]'', the value 8'h03 into ``mem[2]'', and so on, until the final value 8'hff is loaded into ``mem[255]''.

- 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.

```
initial begin
  repeat(100) begin
    $display("Random Number: %0d", $random(02142024));
    $display("Unsigned Random Number: %0d",
                        $urandom(02142024));
  end
end
```

5.1 System Tasks

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

```
module full_adder (.sum     ,
                   .c_out   ,
                   .a       ,
                   .b       ,
                   .c_in    );

assign {c_out, sum} = a + b + c_in;
endmodule
```

```
`timescale 1ns/1ns //reference time/resolution
module tb_full_adder;
reg a, b, c_in;    // in an initial block must be reg
wire sum, c_out;   // wire connection between tb and dut

// --------  Instantiate DUT ---------------
full_adder u_full_adder (.sum   (sum   ),
                         .c_out(c_out  ),
                         .a     (a     ),
                         .b     (b     ),
                         .c_in  (c_in));
```

5.1 System Tasks

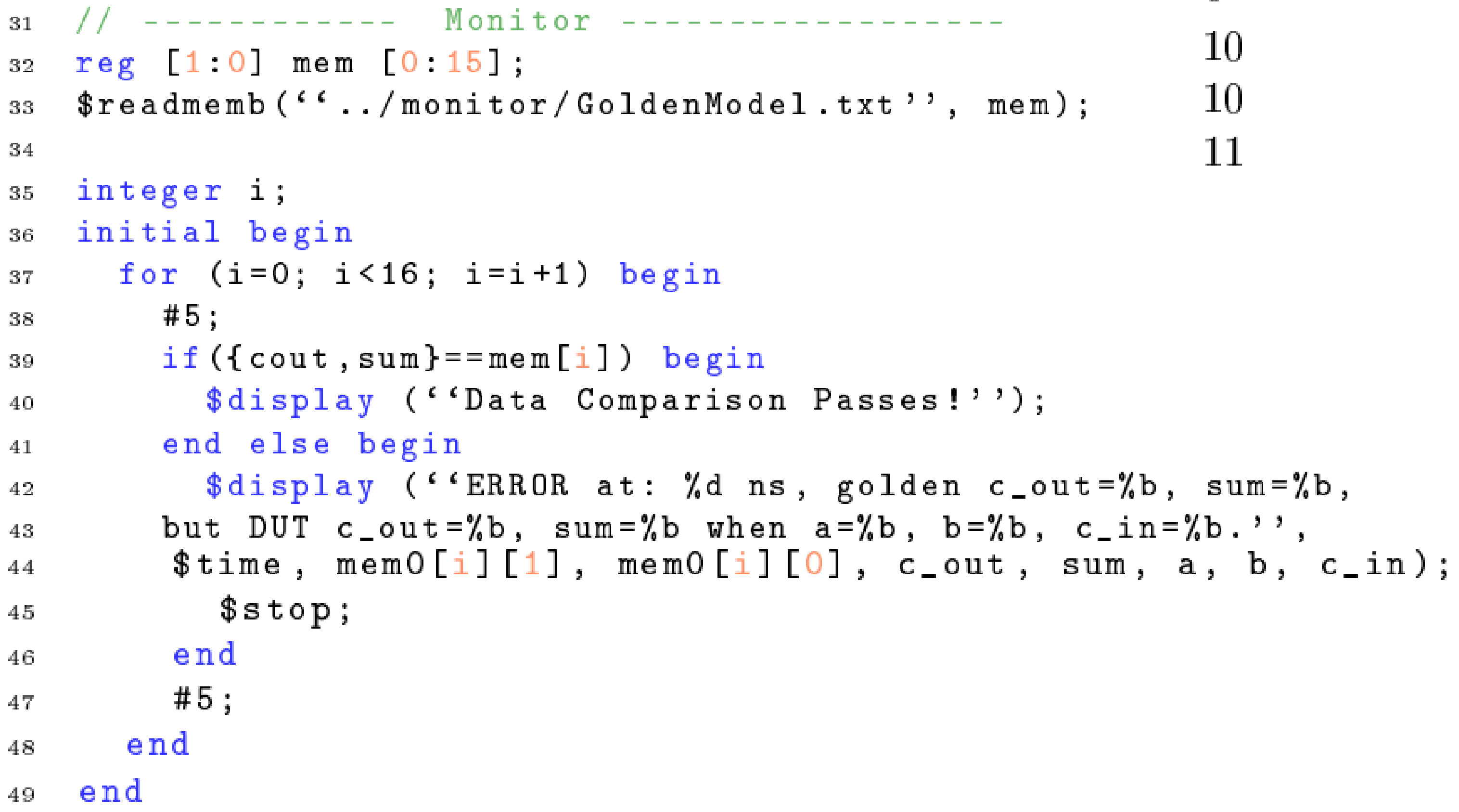
- A testbench Example Using System Tasks
  – Design instantiation
  – Waveform dumping
  – Bus functional model
  – Monitor

```
// --------  Dump VCD Waveform -------------
initial begin
  $dumpfile(``full_adder.vcd'');
  $dumpvars(0,tb_full_adder.u_full_adder);
end

// ------ Bus Function Model (BFM) ---------
initial begin
  a=1'b0;  b=1'b0; c_in=1'b0;        // golden result: 00
  #10;  a=1'b0;  b=1'b0; c_in=1'b1;  // golden result: 01
  #10;  a=1'b0;  b=1'b1; c_in=1'b0;  // golden result: 01
  #10;  a=1'b0;  b=1'b1; c_in=1'b1;  // golden result: 10
  #10;  a=1'b1;  b=1'b0; c_in=1'b0;  // golden result: 01
  #10;  a=1'b1;  b=1'b0; c_in=1'b1;  // golden result: 10
  #10;  a=1'b1;  b=1'b1; c_in=1'b0;  // golden result: 10
  #10;  a=1'b1;  b=1'b1; c_in=1'b1;  // golden result: 11
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
```



```
// ------------  Monitor ------------------
reg [1:0] mem [0:15];
$readmemb(``../monitor/GoldenModel.txt'', mem);

integer i;
initial begin
  for (i=0; i<16; i=i+1) begin
    #5;
    if({cout,sum}==mem[i]) begin
      $display (``Data Comparison Passes!'');
    end else begin
      $display (``ERROR at: %d ns, golden c_out=%b, sum=%b,
    but DUT c_out=%b, sum=%b when a=%b, b=%b, c_in=%b.'',
     $time, mem0[i][1], mem0[i][0], c_out, sum, a, b, c_in);
        $stop;
     end
     #5;
  end
end
```

Outline

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.

```
// An Example Using ‘define
`define WIDTH 8
module adder (input  [`WIDTH-1:0] a,
              input  [`WIDTH-1:0] b,
              output [`WIDTH-1:0] s);
assign s = a+b;
endmodule

// An Example Using parameter
module adder #(parameter WIDTH=8) (input  [WIDTH-1:0] a,
                                   input  [WIDTH-1:0] b,
                                   output [WIDTH-1:0] s);
assign s = a+b;
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

```
// An Example Using `ifdef-`elsif-`endif
module FP_adder (input           clk       ,
                 input           rst       ,
`ifdef PROJECT_32BIT
                 input  [31:0]   i32_data  ,
                 output [31:0]   o32_data
`elsif PROJECT_64BIT
                 input  [63:0]   i64_data  ,
                 output [63:0]   o64_data
`elsif PROJECT_128BIT
                 input  [127:0]  i128_data ,
                 output [127:0]  o128_data
`endif
                                            );
endmodule
```

```
// Select the source code with a TCL script
vlog +define+PROJECT_64BIT \
-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.

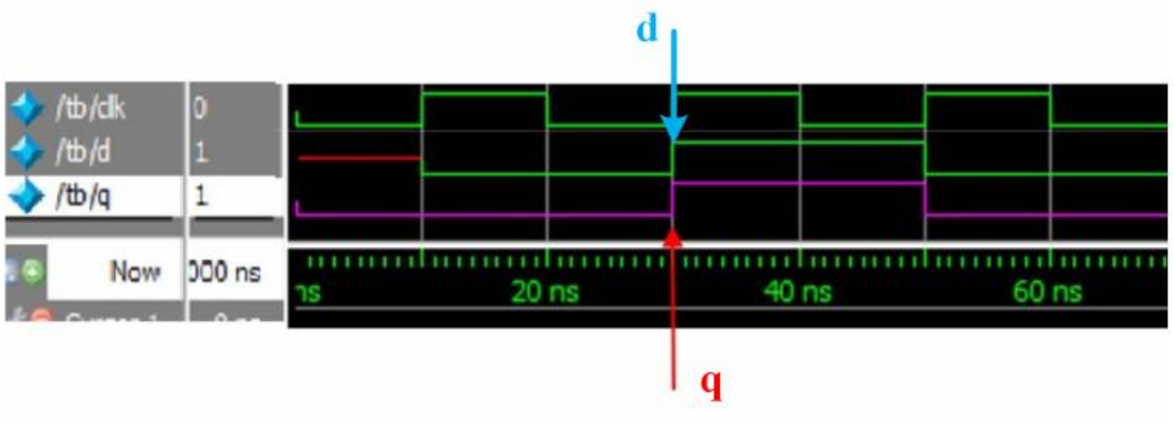
```
initial begin
     $display("%d ns, Load Matrices into Reg Files",$time);
     $readmemh("../golden/dmx_ieee754.txt", dmx_ram);
     $readmemh("../golden/tri_ieee754.txt", tri_ram);
end
```

```
module tb ();
reg [`DMX_MX_WIDTH*32-1:0] dmx_ram[0:`DMX_MX_DPTH-1];
reg [`TRI_MX_WIDTH*32-1:0] tri_ram[0:`TRI_MX_DPTH-1];

`include "load_mem.v"
endmodule
```

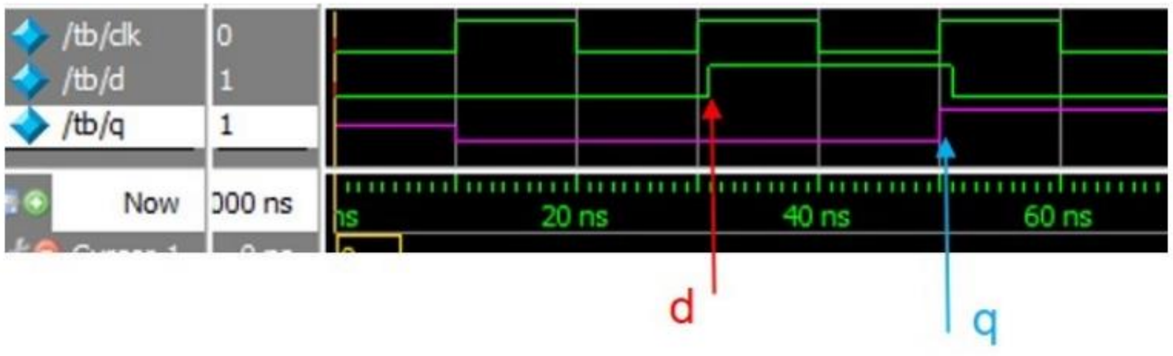
5.2 Compiler Directives

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

```
`timescale 1ns/1ps
always @(posedge clk) begin
  q <= d;
end

always #10 clk = ~clk;

initial begin
  d = 0;
  clk = 0;
  #30.1 d = 1;
  #20 d = 0;
end

initial begin
  d = 0;
  clk = 0;
  #31 d = 1;
  #20 d = 0;
end
```



(a) ModelSim Results of Timescale with 30.1 ns



(b) ModelSim Results of Timescale with 31 ns.

Outline

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.

```
function [<data_type>] <function_name> (<inputs>);
<variable declration>
<statements>
<return result>
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

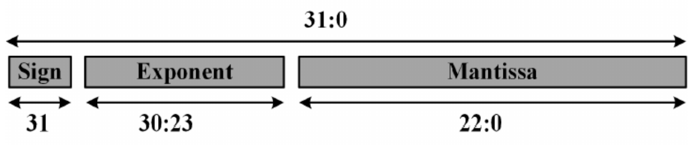


FIGURE 5.2
IEEE 754 Standard (Single Precision): $(-1)^S \times (1.0 + M) \times 2^{(E-127)}$

  - B. Function of IEEE 754 to FP Conversion

```
function real ieee754_to_fp (input [31:0] ieee754_data);
reg          sign    ;
reg [7:0]    exponent;
reg [22:0]   mantissa;

integer int_exp      ;
real    mantissa_val ; // Divide by 2^23
real    fp_output    ;

// Extracting sign, exponent, and mantissa bits
sign     = ieee754_data[31]   ;
exponent = ieee754_data[30:23];
mantissa = ieee754_data[22:0] ;

// Calculating floating-point value
int_exp      = exponent-127;
mantissa_val = 1.0+(mantissa/8388608.0); // Divide by 2^23
fp_output    = (sign?-1:1)*mantissa_val*(2.0**int_exp);

return fp_output;
endfunction
```

5.3 Functions and Tasks

- 5.3.1 Functions
  - C. Call the Function in testbench

```
module tb();
reg [31:0] golden_result, dut_result;
real         golden_real, dut_real, diff_real;
real         error_percent;
integer      log;

initial begin
  log=$fopen("./report.log", "w");
  wait(data_check_en);
  golden_real    = ieee754_to_fp(golden_result)       ;
  dut_real       = ieee754_to_fp(dut_result)          ;
  diff_real      = golden_result_real-dut_result_real;
  error_percent = diff_real/golden_real*100           ;
  if(error_percent<5) begin
    $fwrite (log, "Test Pass!");
    $fwrite (log, "Error percent: %f%%", error_percent);
  end else begin
    $fwrite (log, "Test FAIL!");
    $fwrite (log, "Error percent: %f%%", error_percent);
    $fwrite (log, "Golden real: %f", golden_real);
    $fwrite (log, "DUT real: %f\n", dut_real);
    $fwrite (log, "Golden Hex: %h", golden_result);
    $fwrite (log, "DUT Hex: %h\n", dut_result);
  end
  $fclose(report);
end
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.

```
task <task_name>(<inputs>   ,
                 <outputs>);
<variable declrations>
<statements>
endtask
```

5.3 Functions and Tasks

- 5.3.2 Task
  - A. Tasks of Bus Control Operations

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//            -----       ------
//  clk _____|     |_____|      |_____
//            ------------
//  ce  _____|           |___________
//            ------------
//  wr  _____|           |___________
//
//addr  -----|  tk_addr  |---8'h00---|
//
//wrdata-----| tk_wrdata |---8'h00---|
//
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
task task_cpu_write; // a task to mimic cpu write
input [7:0] tk_addr     ;
input [7:0] tk_wrdata   ;
begin
  $display ("%d CPU Write address: %h, data: %h",
            $time, tk_addr, tk_wrdata);
  $display ("%d -> Driving ce, wr, wrdata, addr", $time);
  @(posedge clk)        ;
  ce        = 1         ;
  wr        = 1         ;
  addr      = tk_addr   ;
  wrdata    = tk_wrdata ;
  @(posedge clk)        ;
  ce        = 0         ;
  wr        = 0         ;
  addr      = 0         ;
  wrdata    = 0         ;
  $display ("======================");
end
endtask
```

5.3 Functions and Tasks

- 5.3.2 Task
  - A. Tasks of Bus Control Operations

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//            -----       -----
//  clk _____|     |_____|     |_____
//            -----------
//  ce  _____|           |___________
//            -----------
//  rd  _____|           |___________
//
//addr  -----|  tk_addr  |---8'h00---|
//
//rddata-----------------------|  rddata  |
//
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
task task_cpu_read;
input  [7:0] tk_addr  ;
output [7:0] tk_rddata;
begin
  $display ("%d CPU Read address: %h!", $time, tk_addr)
  $display ("%d -> Driving ce, rd, addr", $time);
  @(posedge clk)      ;
  ce        = 1       ;
  rd        = 1       ;
  addr      = tk_addr ;
  @(posedge clk)      ;
  ce        = 0       ;
  rd        = 0       ;
  addr      = 0       ;
  @(negedge clk)      ;
  tk_rddata = rddata  ;
  $display ("%g CPU Read data : %h", $time, tk_rddata);
  $display ("======================");
end
endtask
```

5.3 Functions and Tasks

- B. Call the Tasks in testbench

```
// Call cpu_write and cpu_read tasks
module task_cpu_wr_rd();
reg         ce, rd, wr  ;
reg   [7:0] addr, wrdata;
wire  [7:0] rddata      ;
`include "task_cpu_write.v"
`include "task_cpu_read.v"

initial clk = 0;
always #5 clk = ~clk;

initial begin
  ce     = 1'b0;
  wr     = 1'b0;
  rd     = 1'b0;
  addr   = 8'h0;
  wrdata = 8'h0;

  task_cpu_write(8'h01, 8'h10    ); //Call the write task
  task_cpu_read (8'h01, rddata   ); //Call the read task
  #5 task_cpu_write(8'h00, 8'h85 );
     task_cpu_read (8'h00, rddata);
end
endmodule
```

```
// Printed log:
0 CPU Write address: 01, data : 10
0 -> Driving ce, wr, wdata, addr
======================
25 CPU Read address : 01
25 -> Driving ce, rd, addr
40 CPU Read data: 10
======================
45 CPU Write address: 00, data: 85
45 -> Driving ce, wr, wrdata, addr
======================
65 CPU Read address : 00
65 -> Driving ce, rd, addr
80 CPU Read data : 85
======================
```

Outline

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:
    - Event expressions are used to specify events that trigger a block of code, and they 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

```
initial begin
  rst=1'b0; clk=1'b0;
  #10; rst=1'b1;
end

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 synchro clock domain.

```verilog
initial begin
  a=1'b0;
  @(posedge clk);
  a=1'b1;
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.

```verilog
initial begin
  b=1'b0;
  wait(a);
  b=1'b1;
end
```

Outline

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.

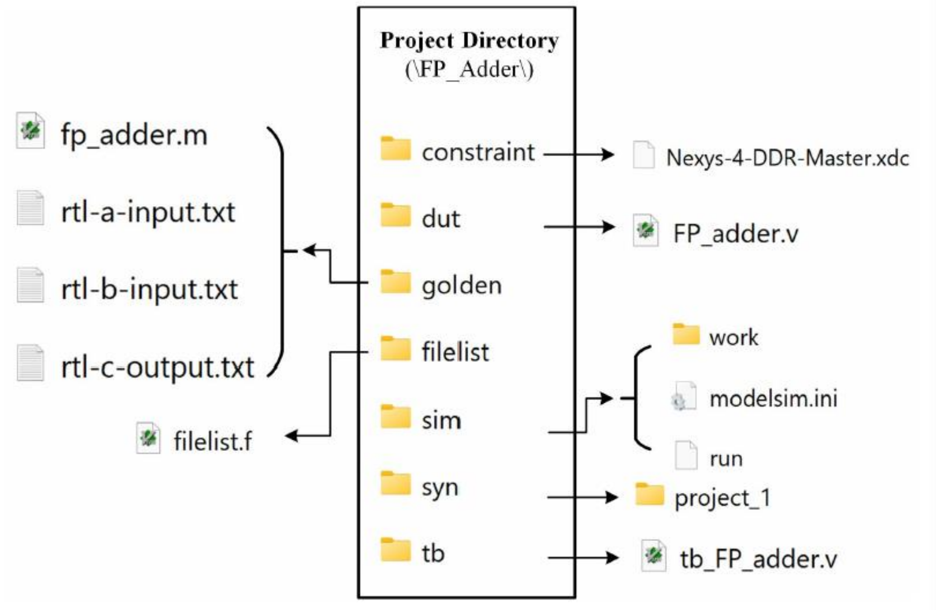
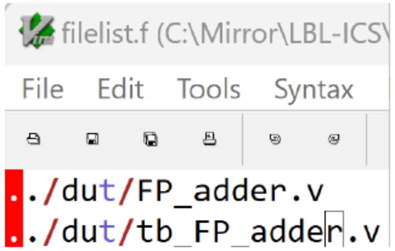


FIGURE 5.3
Structured Project Directory

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''.



```
../dut/FP_adder.v
../dut/tb_FP_adder.v
```

FIGURE 5.4
Filelist Example

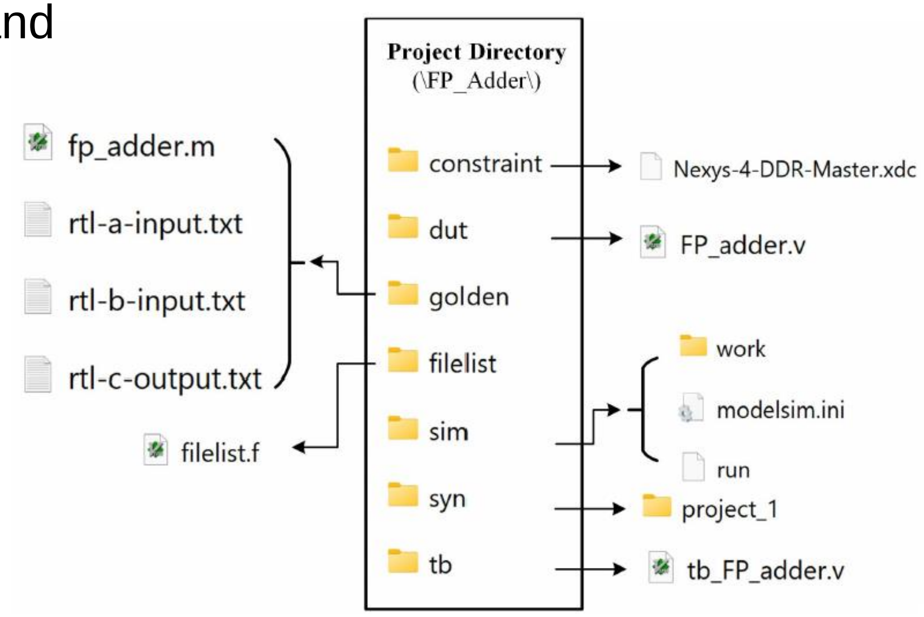


FIGURE 5.3
Structured Project Directory

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.

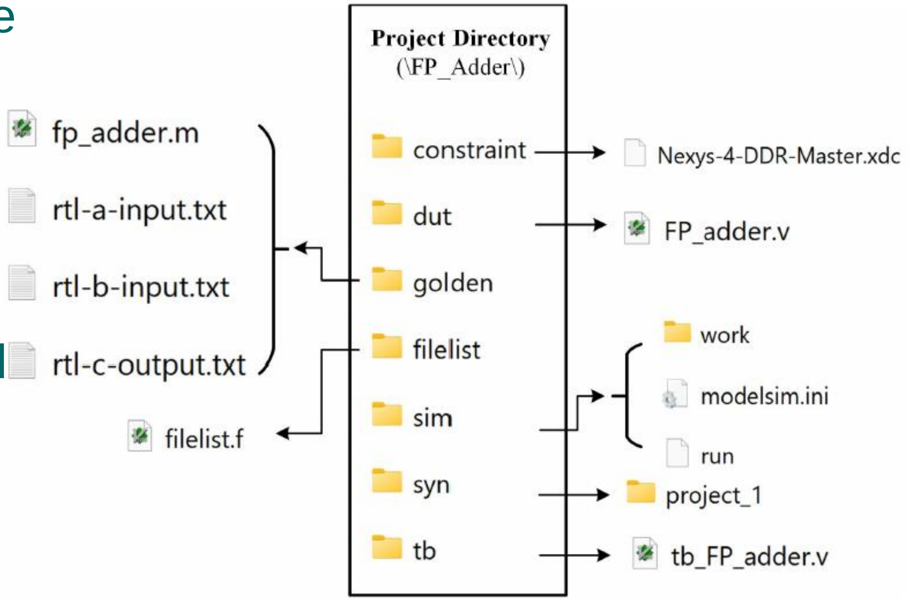
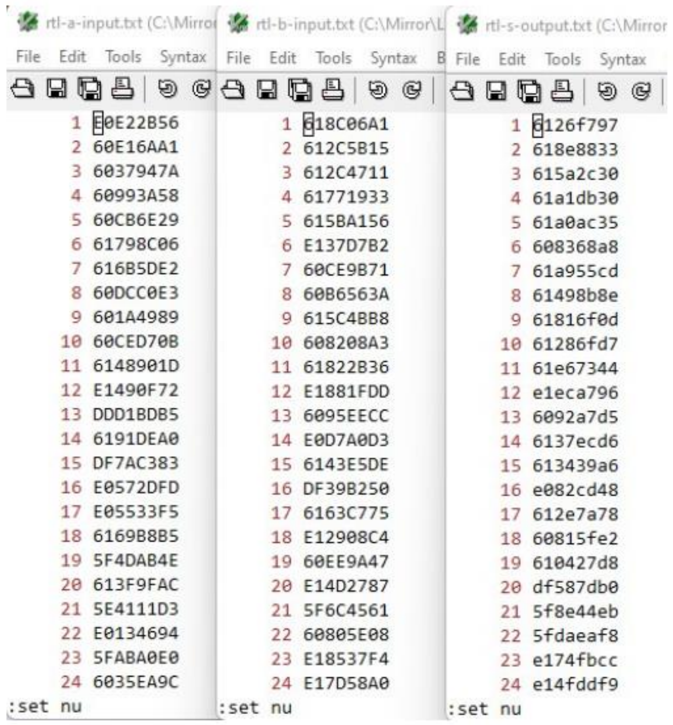


FIGURE 5.3
Structured Project Directory

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 (C:\Mirror
rtl-b-input.txt (C:\Mirror\L
rtl-s-output.txt (C:\Mirror
File Edit Tools Syntax
File Edit Tools Syntax B
File Edit Tools Syntax

```
E0E22B56
60E16AA1
6037947A
60993A58
60CB6E29
61798C06
616B5DE2
60DCC0E3
601A4989
60CED70B
6148901D
E1490F72
DDD1BDB5
6191DEA0
DF7AC383
E0572DFD
E05533F5
6169B8B5
5F4DAB4E
613F9FAC
5E4111D3
E0134694
5FABA0E0
6035EA9C
:set nu
```

```
618C06A1
612C5B15
612C4711
61771933
615BA156
E137D7B2
60CE9B71
60B6563A
615C4BB8
608208A3
61822B36
E1881FDD
6095EECC
E0D7A0D3
6143E5DE
DF39B250
6163C775
E12908C4
60EE9A47
E14D2787
5F6C4561
60805E08
E18537F4
E17D58A0
:set nu
```

```
6126f797
618e8833
615a2c30
61a1db30
61a0ac35
608368a8
61a955cd
61498b8e
61816f0d
61286fd7
61e67344
e1eca796
6092a7d5
6137ecd6
613439a6
e082cd48
612e7a78
60815fe2
610427d8
df587db0
5f8e44eb
5fdaeaf8
e174fbcc
e14fddf9
:set nu
```

FIGURE 5.5
Golden Results Files.

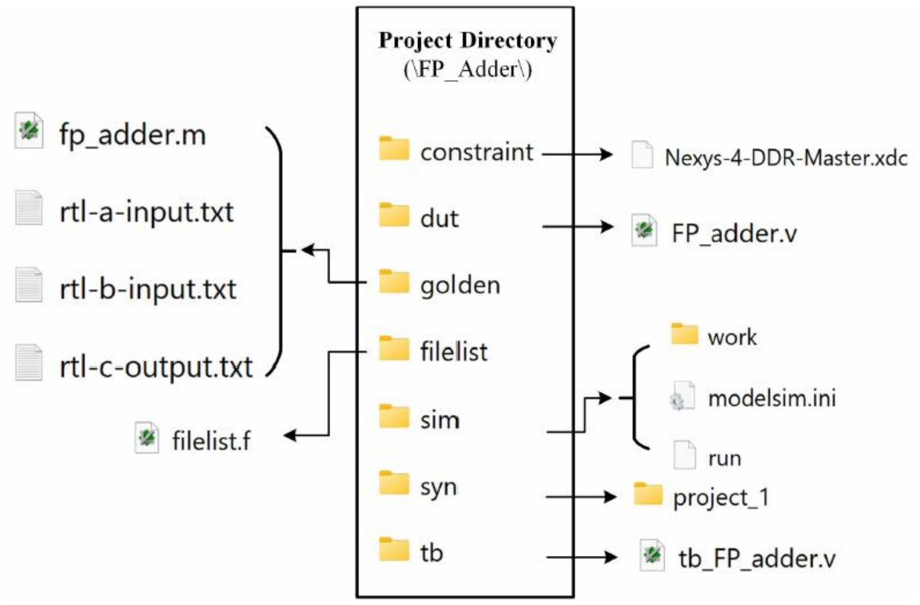
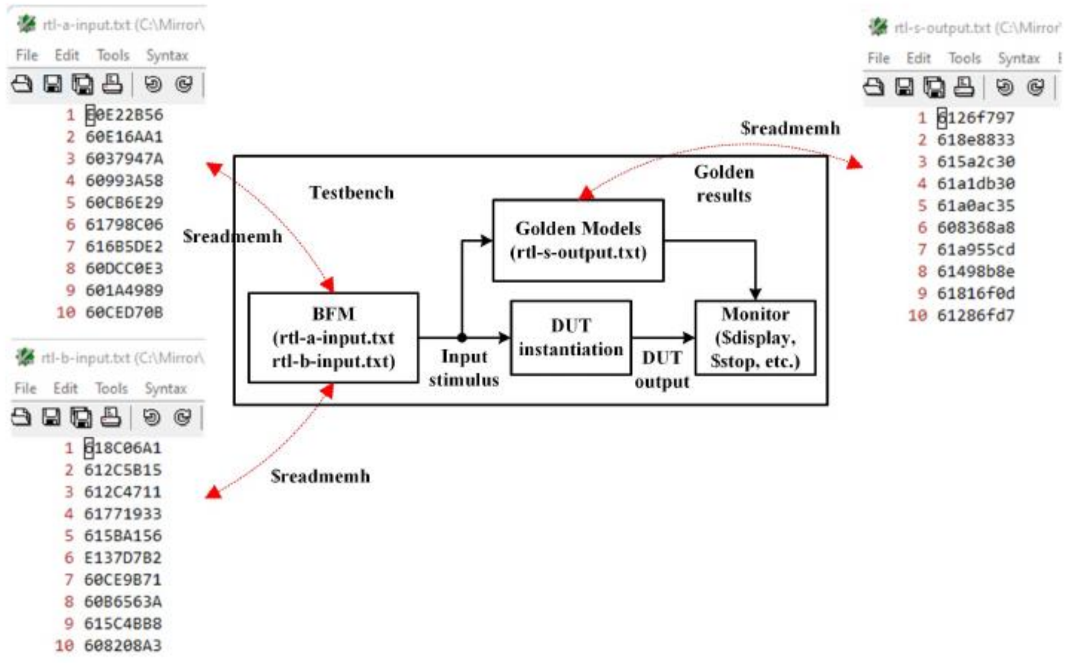


FIGURE 5.3
Structured Project Directory

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.



rtl-a-input.txt (C:\Mirror\
File Edit Tools Syntax
rtl-b-input.txt (C:\Mirror\
File Edit Tools Syntax
rtl-s-output.txt (C:\Mirror
File Edit Tools Syntax
$readmemh
$readmemh
$readmemh
Testbench
Golden results
Golden Models (rtl-s-output.txt)
BFM (rtl-a-input.txt rtl-b-input.txt)
Input stimulus
DUT instantiation
DUT output
Monitor ($display, $stop, etc.)

rtl-a-input.txt:

```
60E22B56
60E16AA1
6037947A
60993A58
60CB6E29
61798C06
616B5DE2
60DCC0E3
601A4989
60CED70B
```

rtl-b-input.txt:

```
618C06A1
612C5B15
612C4711
61771933
615BA156
E137D7B2
60CE9B71
60B6563A
615C4BB8
608208A3
```

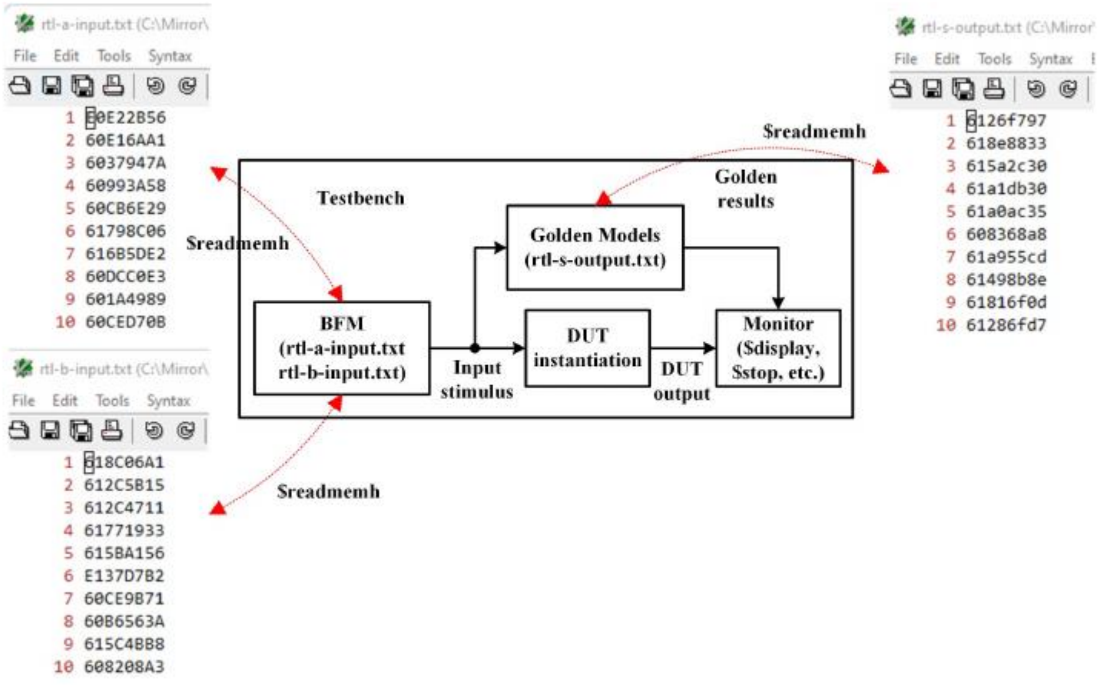
rtl-s-output.txt:

```
6126f797
618e8833
615a2c30
61a1db30
61a0ac35
608368a8
61a955cd
61498b8e
61816f0d
61286fd7
```

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.


rtl-a-input.txt (C:\Mirror\
File Edit Tools Syntax
rtl-b-input.txt (C:\Mirror\
File Edit Tools Syntax
rtl-s-output.txt (C:\Mirror
File Edit Tools Syntax
$readmemh
$readmemh
$readmemh
Testbench
Golden results
Golden Models (rtl-s-output.txt)
BFM (rtl-a-input.txt rtl-b-input.txt)
Input stimulus
DUT instantiation
DUT output
Monitor ($display, $stop, etc.)

```
60E22B56
60E16AA1
6037947A
60993A58
60CB6E29
61798C06
616B5DE2
60DCC0E3
601A4989
60CED70B
```

```
618C06A1
612C5B15
612C4711
61771933
615BA156
E137D7B2
60CE9B71
60B6563A
615C4BB8
608208A3
```

```
6126f797
618e8833
615a2c30
61a1db30
61a0ac35
608368a8
61a955cd
61498b8e
61816f0d
61286fd7
```

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

```verilog
/* -------------------------------------------------------
 * This source file contains a simulation testbench for a FP
 * Adder generated by the Chisel HCL.
 * Design-under-test: FP_adder
 * Testbench         : tb_FP_adder
 * Design IOs        : inputs - clock, reset,
 *                              io_in_a[31:0], io_in_b[31:0]
 *                     outputs- io_out_s[31:0]
 * Latency           : One cycle per FP addition
 * Precision         : Single precision
 * Resource Required: 16 Binary Adders (32 x 32 bits)
 *
 * Author            : Xiaokun Yang
 * Date              : June 2022
*///------------------------------------------------------
`timescale 1ns/1ns
module tb_FP_adder();
parameter TEST_SIZE = 1_000_000;
reg          clock    ;
reg          reset    ;
reg   [31:0] io_in_a  ;
reg   [31:0] io_in_b  ;
wire  [31:0] io_out_s ;
```

```verilog
// --------------------------------
// Stimulus Files and Golden File
// --------------------------------
reg [31:0]  input_a[TEST_SIZE-1:0] ;
reg [31:0]  input_b[TEST_SIZE-1:0] ;
reg [31:0]  output_c[TEST_SIZE-1:0];

initial begin
   $readmemh("../golden/rtl-a-input.txt",input_a)  ;
   $readmemh("../golden/rtl-b-input.txt",input_b)  ;
   $readmemh("../golden/rtl-s-output.txt",output_s);
end
```

5.5 Automated Simulation Environment and Verilog Testbench

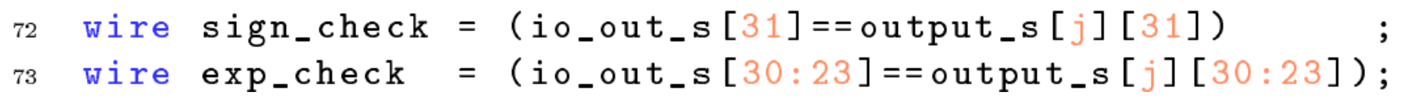
- 5.5.3 Verilog Design on Automated Simulation Testbench
  - A. Load Memory Array and Instantiate Design-under-Test
  - B. Bus Functional Model

```
// --------------------------------
// --- DUT Instantiation ---------
// --------------------------------
FP_adder u_FP_adder (
                     .clock   (clock   ),
                     .reset   (reset   ),
                     .io_in_a (io_in_a ),
                     .io_in_b (io_in_b ),
                     .io_out_s(io_out_s));
```

```
// --------------------------------
// --- Bus Functional Models -----
// --------------------------------
integer i;
initial begin
   reset = 1'b0;
   clock = 1'b0;
   #100;
   reset = 1'b1;
   @(posedge clock);

  for (i=0; i < TEST_SIZE; i = i+1) begin
    io_in_a <= input_a[i];
    io_in_b <= input_b[i];
    @(posedge clock);
  end
end

always #10 clock = ~clock;
```

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.



```
wire sign_check = (io_out_s[31]==output_s[j][31])      ;
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.

```
// ---------------------------------
// ----------Monitor -------------
// ---------------------------------
integer j, m, n, o, p, q, r, s;
wire sign_check = (io_out_s[31]==output_s[j][31])        ;
wire exp_check  = (io_out_s[30:23]==output_s[j][30:23]);
wire mant_check = (io_out_s[22:0]==output_s[j][22:0])   ;
wire mant_22_4  = (io_out_s[22:4]==output_s[j][22:4])   ;
wire mant_22_8  = (io_out_s[22:8]==output_s[j][22:8])   ;
wire mant_22_16 = (io_out_s[22:16]==output_s[j][22:16]);
wire mant_3_0   = (io_out_s[3:0]==output_s[j][3:0])     ;
wire mant_7_0   = (io_out_s[7:0]==output_s[j][7:0])     ;
wire mant_15_0  = (io_out_s[15:0]==output_s[j][15:0])   ;
```

5.5 Automated Simulation Environment and Verilog Testbench

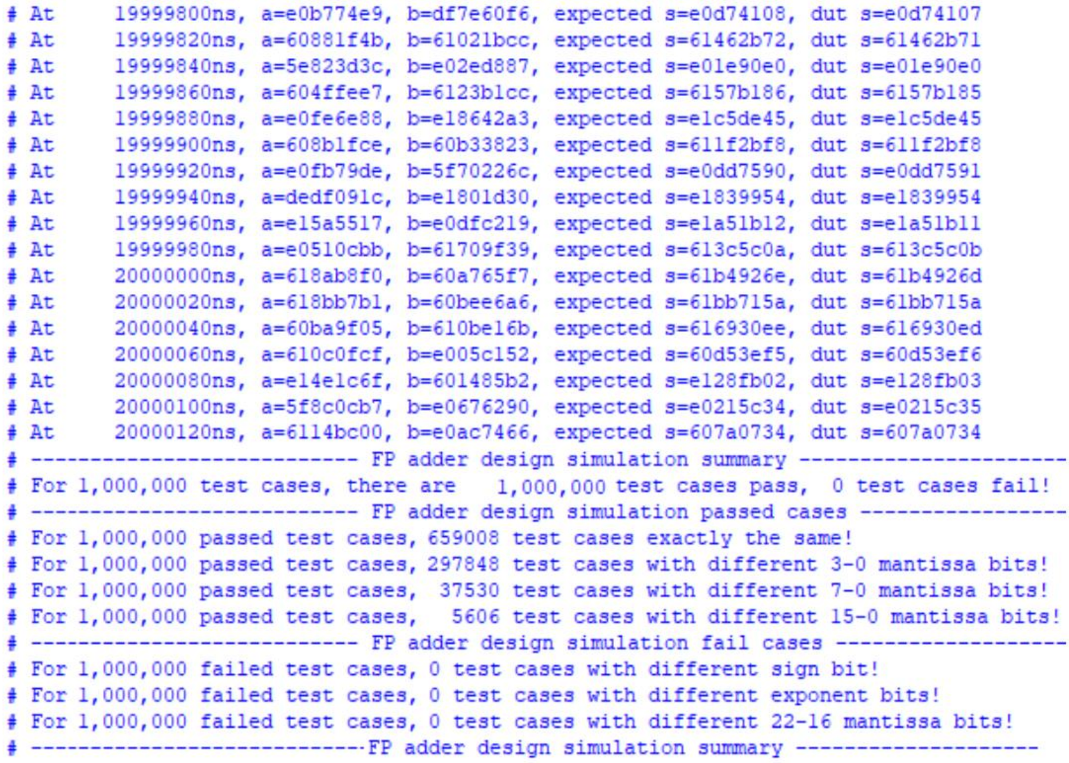
- 5.5.3 Verilog Design on Automated Simulation Testbench
  - C. Monitor and Test Plan

```
initial begin
  m = 0 ;
  n = 0 ;
  o = 0 ;
  p = 0 ;
  q = 0 ;
  r = 0 ;
  s = 0 ;
  wait (reset);
  repeat (2) @(negedge clock);
  for (j=0; j < TEST_SIZE; j = j+1) begin
    $display("%d ns, a=%h, b=%h, golden s=%h, dut s=%h",
$time, input_a[i-1], input_b[i-1], output_s[j], io_out_s);
    case({sign_check, exp_check, mant_check})
    3'b111: begin
              m=m+1;
            end
    3'b110: begin
            if(mant_22_4 & ~mant_3_0) begin
              n=n+1;
             end else if(mant_22_8 & ~mant_7_0) begin
               o=o+1;
             end else if(mant_22_16 & ~mant_15_0) begin
               p=p+1;
             end
             end
  default: begin
            if(~sign_check) begin
              $display("Sign bit different!, j=%d", j);
              q=q+1;
            end else if(~exp_30_23) begin
              $display("Exponent are different!");
              r=r+1;
            end else if(~mant_22_16) begin
              $display("Matanssa[22:16] are different!");
              s=s+1;
            end
            end
    endcase
    @(negedge clock);
  end
```

5.5 Automated Simulation Environment and Verilog Testbench

```verilog
    $display("------FP adder simulation summary-------");
    $display("%d cases pass, %d fail!", m+n+o+p, q+r+s);
    $display("----FP adder simulation passed cases---");
    $display("%d cases exactly the same!", m);
    $display("%d cases: different mantissa[3:0]!", n);
    $display("%d cases: different mantissa[7:0]!", o);
    $display("%d cases: different mantissa[15:0]!", p);
    $display("----FP adder simulation failed cases---");
    $display("%d cases: different sign bit!", q);
    $display("%d cases: different exponent bits!", r);
    $display("%d cases: different mantissa[22:16]!", s);
    $display("------FP adder simulation summary-------");
  end
  endmodule
```

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

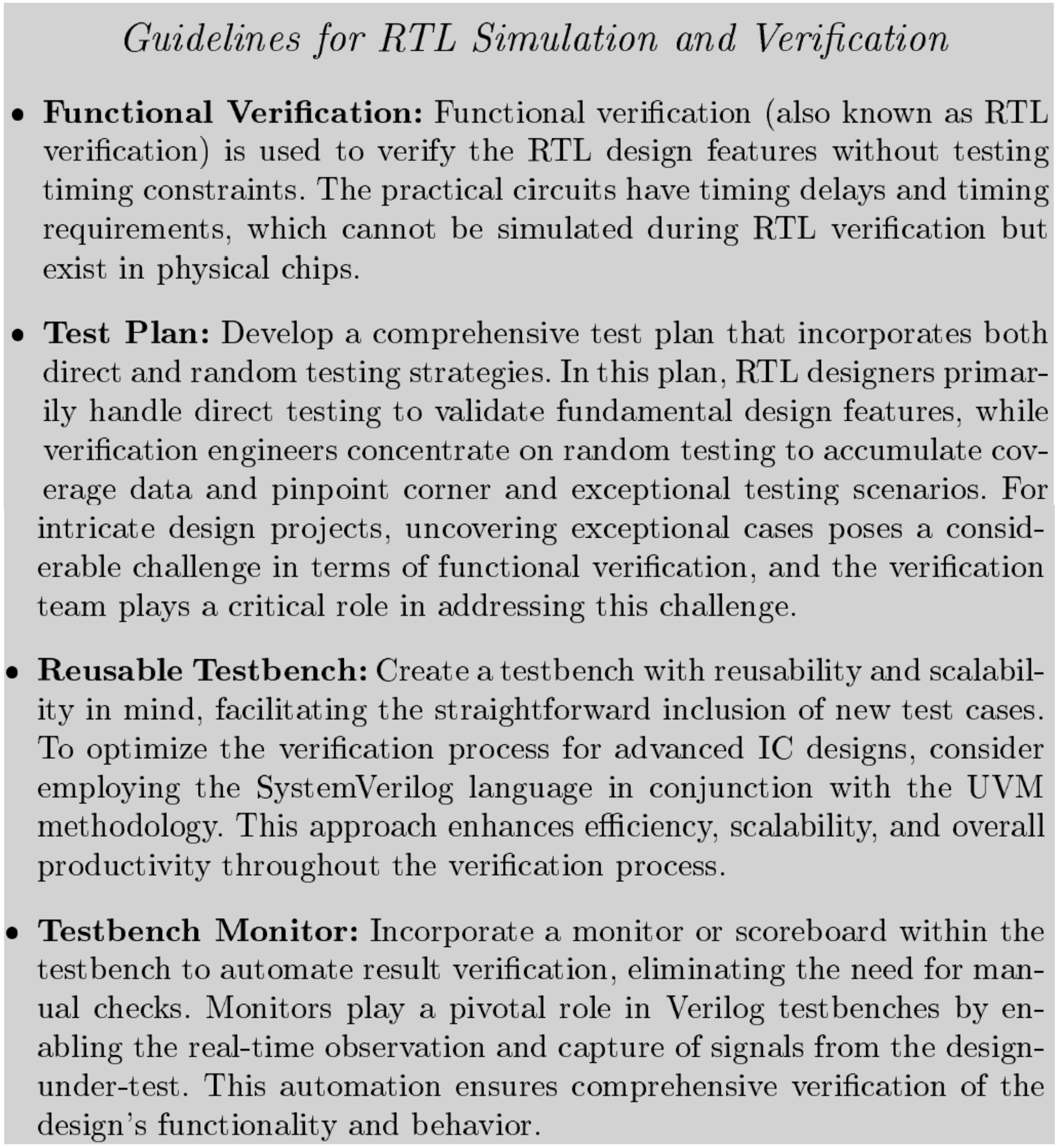


```
# At     19999800ns, a=e0b774e9, b=df7e60f6, expected s=e0d74108, dut s=e0d74107
# At     19999820ns, a=60881f4b, b=61021bcc, expected s=61462b72, dut s=61462b71
# At     19999840ns, a=5e823d3c, b=e02ed887, expected s=e01e90e0, dut s=e01e90e0
# At     19999860ns, a=604ffee7, b=6123b1cc, expected 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=e14e1c6f, 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

Outline

5.6 Guidelines for RTL Simulation and Verification



Guidelines for RTL Simulation and Verification

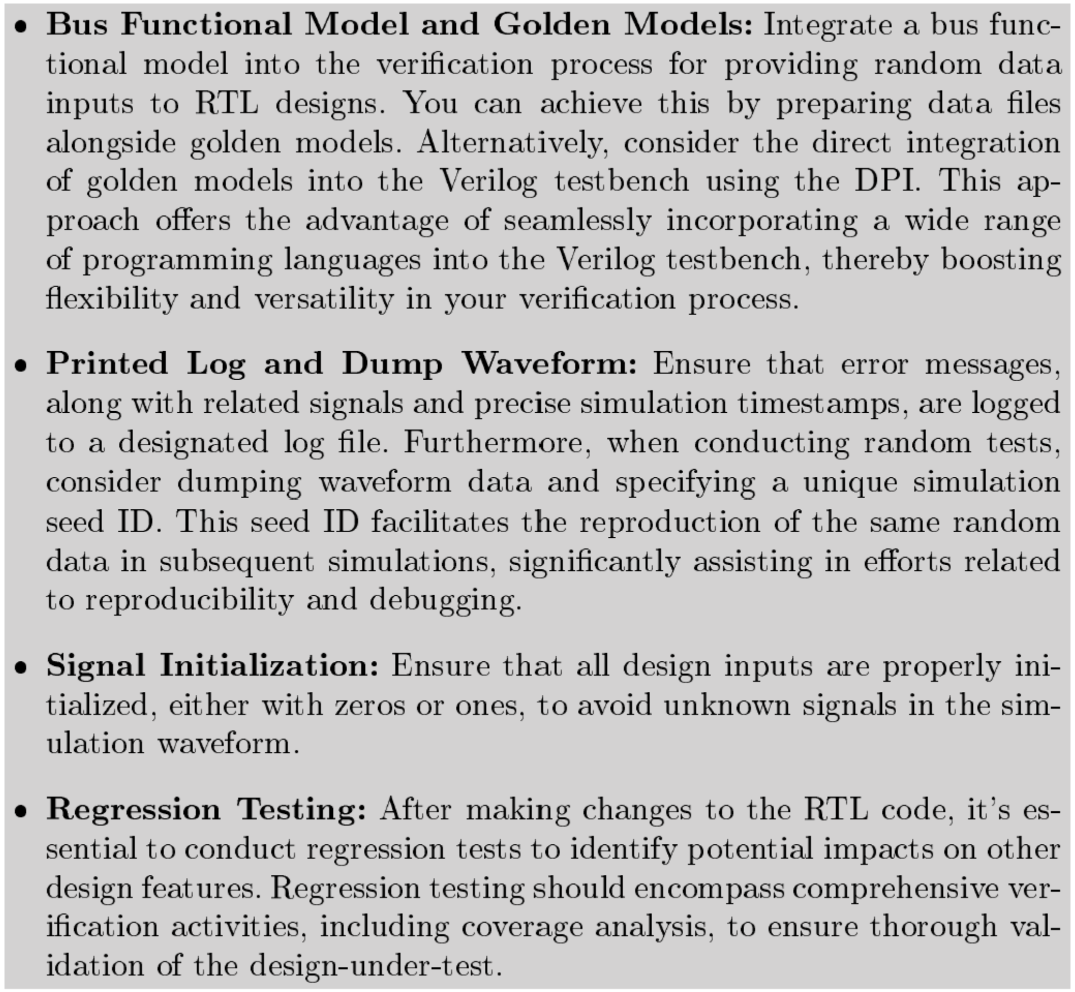
- **Functional Verification:** Functional verification (also known as RTL verification) is used to verify the RTL design features without testing timing constraints. The practical circuits have timing delays and timing requirements, which cannot be simulated during RTL verification but exist in physical 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.