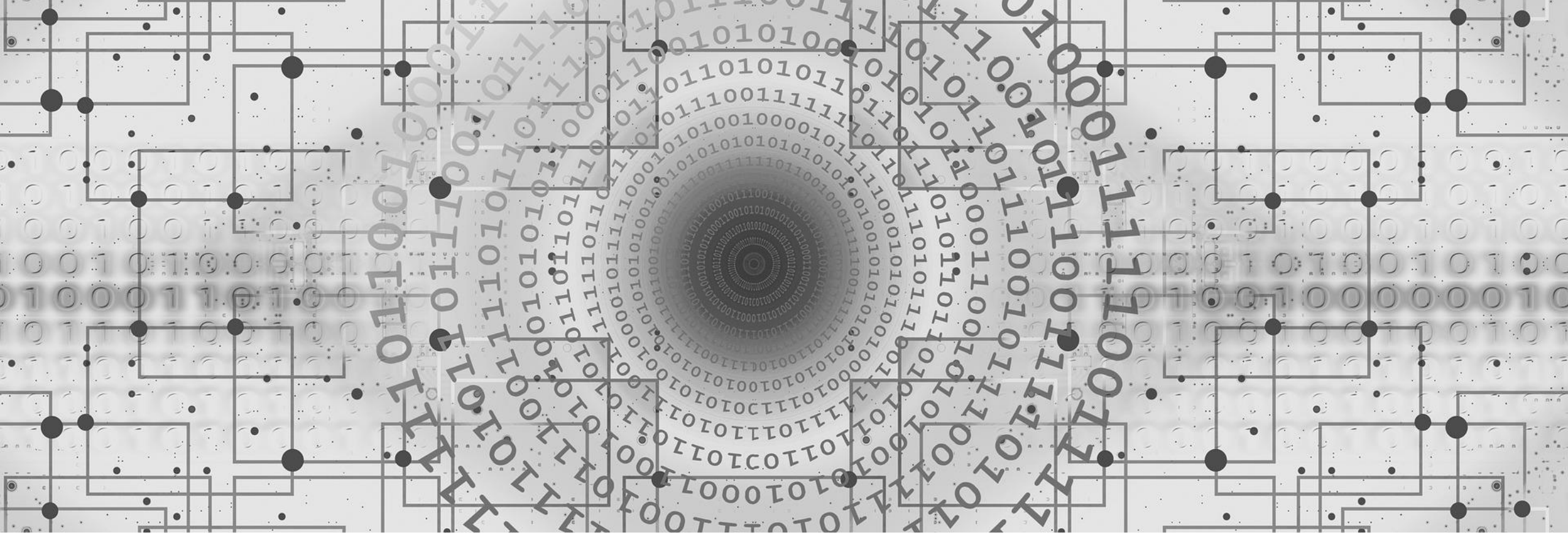


Testbench, System Tasks/Functions, Compiler directives, Simulation, EDA Tools

ECE-111

Vishal Karna

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Testbench

Why Testbench ?

- **SystemVerilog design RTL code is developed to specify a circuit, but :**
  - How do we know if the circuit works correctly as per the original intent ?
- **Hardware generation by trial and error is costly!**
- **To avoid costly hardware generation re-spins :**
  - Perform functional verification of design code prior to actual hardware fabrication
  - Functional verification is performed by developing testbench code
    - Using event driven simulator and testbench code simulate design-under-test (DUT)
- **In modern digital designs functional verification is one of the most complex and time consuming stage of ASIC design flow**
  - Quality of hardware is dependent on completeness in verification !!

What is a Testbench ?

- **Testbench is a SystemVerilog module which tests another module know as design-under- test (DUT)**

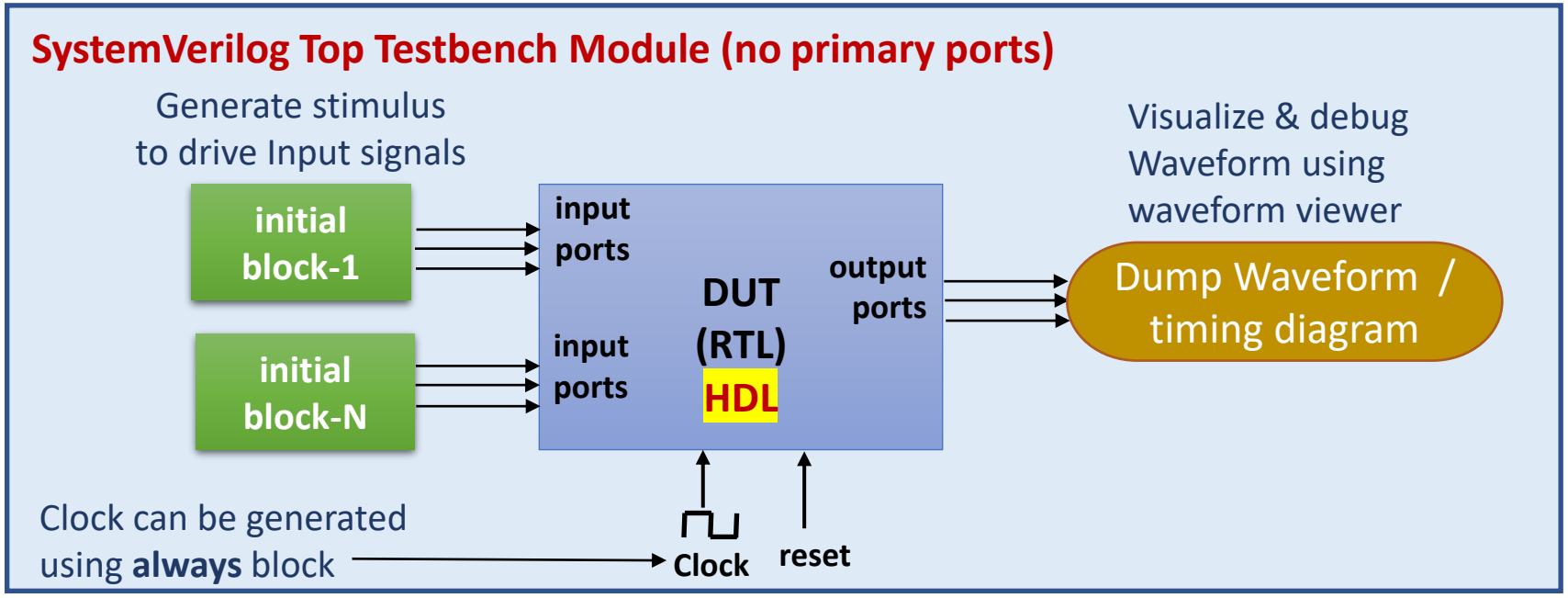
- **Non-synthesizable code and it is not part of the final hardware generated**
  - Testbench is a SystemVerilog procedural block that executes only once
  - It is used for simulation purpose only

- **Types of testbench :**
  - Basic directed testbench without using advance SystemVerilog OOP constructs :
    - Without self-checking, only stimulus generation for input signals of design
    - With self-checking and stimulus generation using code specified within procedural block
    - With self-checking and stimulus generated using input vector file
  - Advance constraint random and coverage driven testbench using SystemVerilog OOP constructs

Note : ECE-111 course will only cover basic directed testbench. Advance testbench using SystemVerilog OOP constructs is out of the scope of this course.

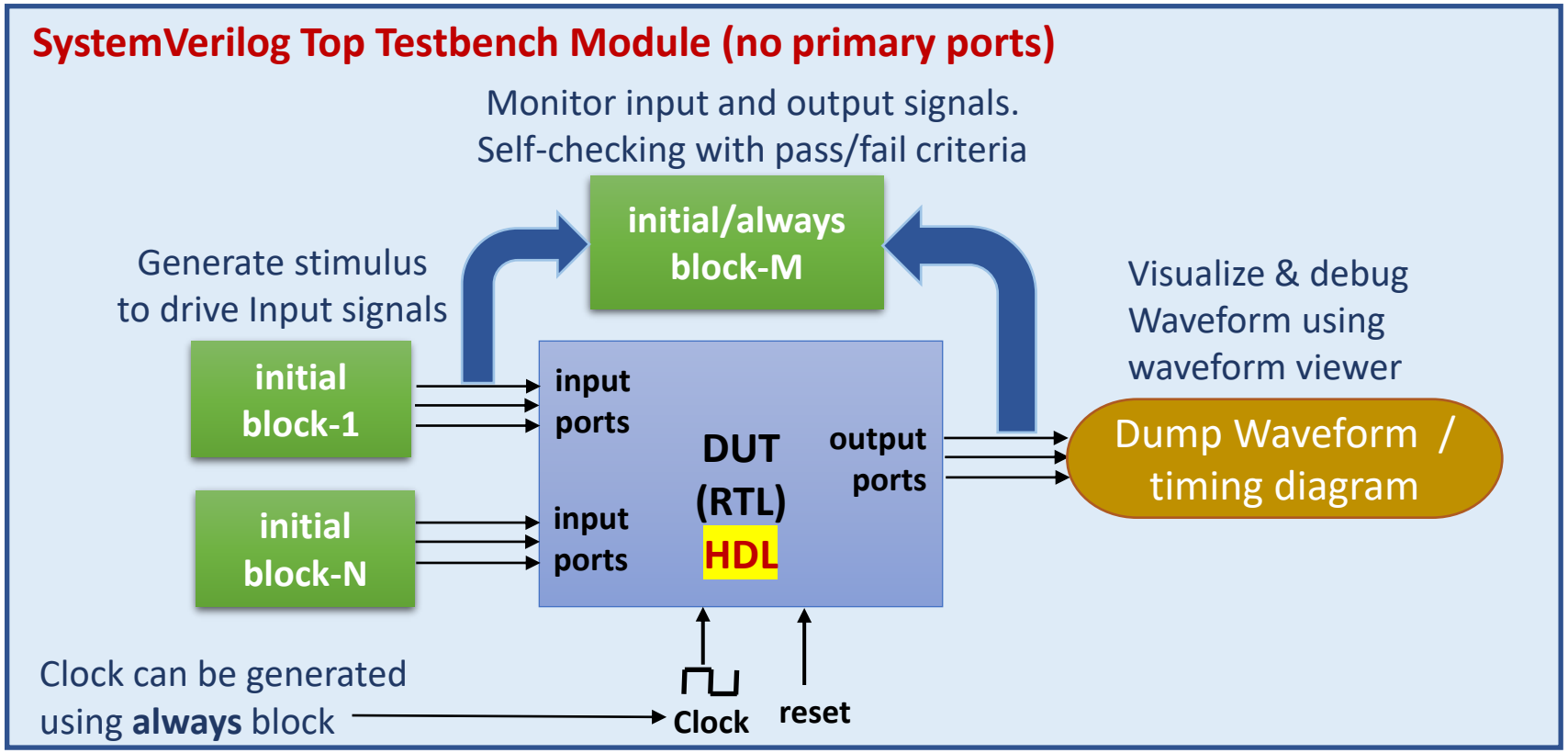
Basic Testbench Without Self-Checking

- **Testbench contains top SystemVerilog module without primary ports**
- **Instantiates the design-under-test (DUT)**
  - Inputs and outputs of the DUT are connected to the local variables in testbench
- **Generates stimulus for all input signals using initial, always procedural blocks and blocking statements (=)**
  - Clock generation logic using **always** procedural block and apply generated clock to DUT clock signal
  - Applies series of values from one or more **initial** blocks to all input and inout signals of DUT
- **Outputs are observed and compared against expected behavior by reviewing simulation waveforms –** This approach of ensuring correctness of design is cumbersome and error prone !



Basic Testbench With Self-Checking

- **Includes statements and code to perform self-checking**
  - Monitor input and output signals
  - Compare outputs with expected values at each cycle or with respect to certain state of signals
  - Alternatively, compare outputs with golden model developed using function/task/module
  - Generate pass or failure message for each occurrence using simulator system tasks such as $display, $error
  - With this approach functional failures of design are generated dynamically without relying for exhaustive cycle by cycle waveform checking – Practical approach to ensure correctness of design in a test scenario !



Testbench With Self-Checking Example

❑ Testbench using multiple initial blocks and self-checking

```
`timescale 1ns/10ps // timeunit=1ns, precision=10ps
module testbench_top; // Testbench module without ports
 logic in0, in1, carryin;
 logic[1:0] result;

 fulladder DUT(    // instantiate design under test
 .a(in0),
 .b(in1),
 .cin(carryin),
 .cout(result[1]),
 .sum(result[0]));

initial begin // initial block to drive input values
  repeat(10) begin
    in0 = $random; //apply random stimulus
    in1 = $random;
    carryin = $random;
    #10; // wait for 10 time unit period
    if( in1 + in2 + carryin != result) // self-checking logic.
      $error("ERROR : Incorrect addition value returned\n");
  end
  $finish; // Terminate simulation !
end
```

Test fail reporting !

```
module fulladder( // Design Module (DUT)
  input logic a, b, cin,
  output logic sum, cout
);
  logic p, q;
  assign p = a ^ b;
  assign q = a & b;
  assign sum = p ^ cin;
  assign cout = q | (p & cin);
endmodule
```

```
initial begin // initial block to display inputs and outputs
  $monitor("time=%0t   in0=%d   in1=%d   carryin=%d
                result=%d",   $time, in0, in1, carryin, result);
 end
endmodule: testbench_top
```

Testbench With Vectors and Self-Checking Example

❑ Testbench using multiple initial blocks and self-checking

```
`timescale 1ns/10ps // timeunit=1ns, precision=10ps
module testbench_top; // Testbench module without ports
 logic in0, in1, carryin;
 logic[1:0] result;

 fulladder DUT(    // instantiate design under test
 .a(in0),
 .b(in1),
 .cin(carryin),
 .cout(result[1]),
 .sum(result[0]));

initial begin // initial block to drive input values
  repeat(10) begin
    in0 = $random; //apply random stimulus
    in1 = $random;
    carryin = $random;
    #10; // wait for 10 time unit period
    if( in1 + in2 + carryin != result) // self-checking logic.
      $error("ERROR : Incorrect addition value returned\n");
  end
  $finish; // Terminate simulation !
end
```

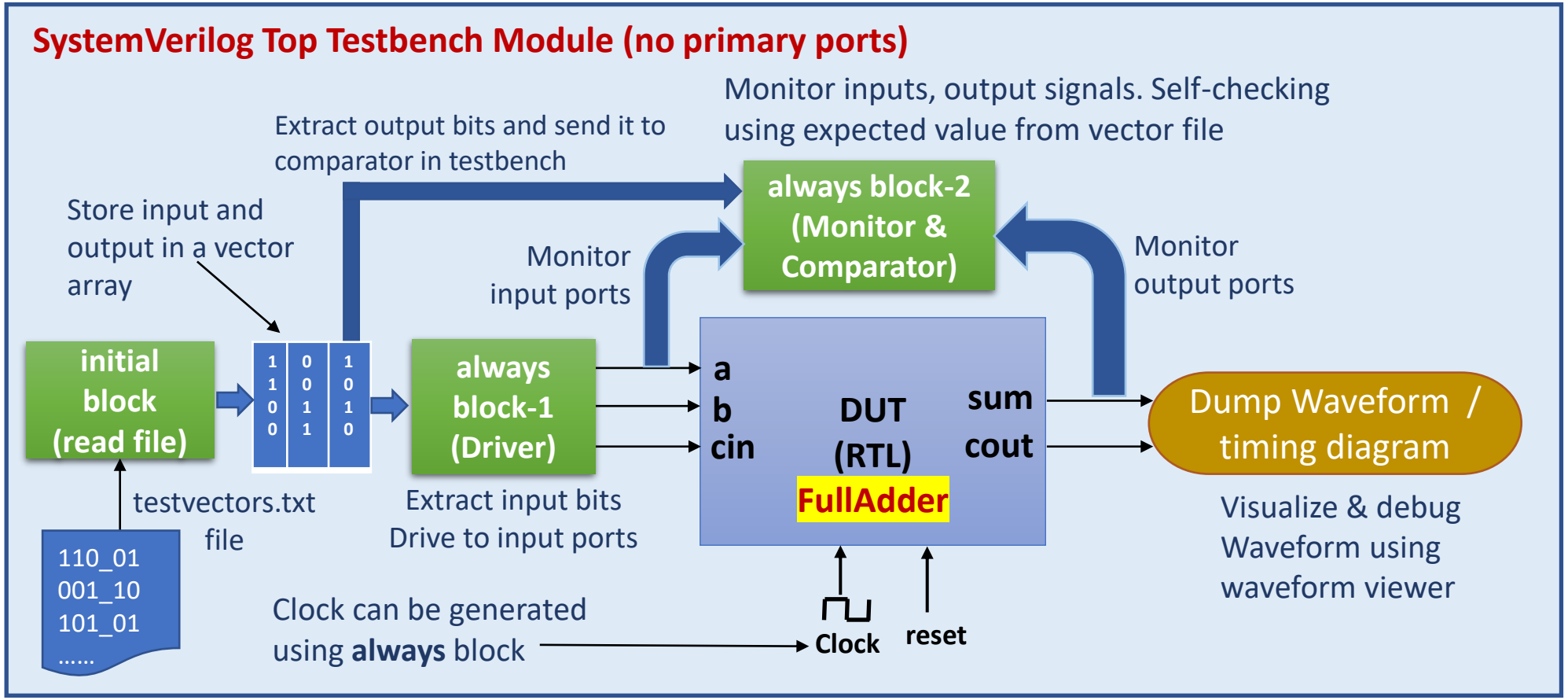
Test fail reporting !

```
module fulladder( // Design Module (DUT)
  input logic a, b, cin,
  output logic sum, cout
);
  logic p, q;
  assign p = a ^ b;
  assign q = a & b;
  assign sum = p ^ cin;
  assign cout = q | (p & cin);
endmodule
```

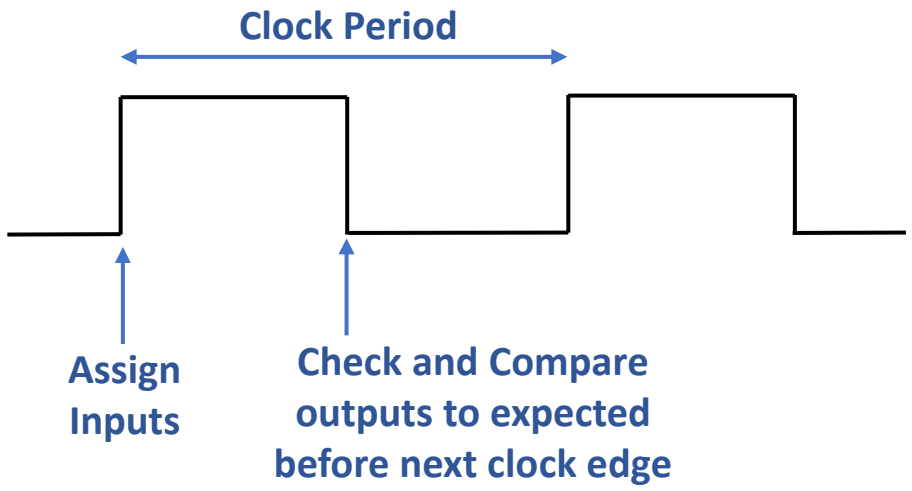
```
initial begin // initial block to display inputs and outputs
  $monitor("time=%0t   in0=%d   in1=%d   carryin=%d
                result=%d",   $time, in0, in1, carryin, result);
 end
endmodule: testbench_top
```

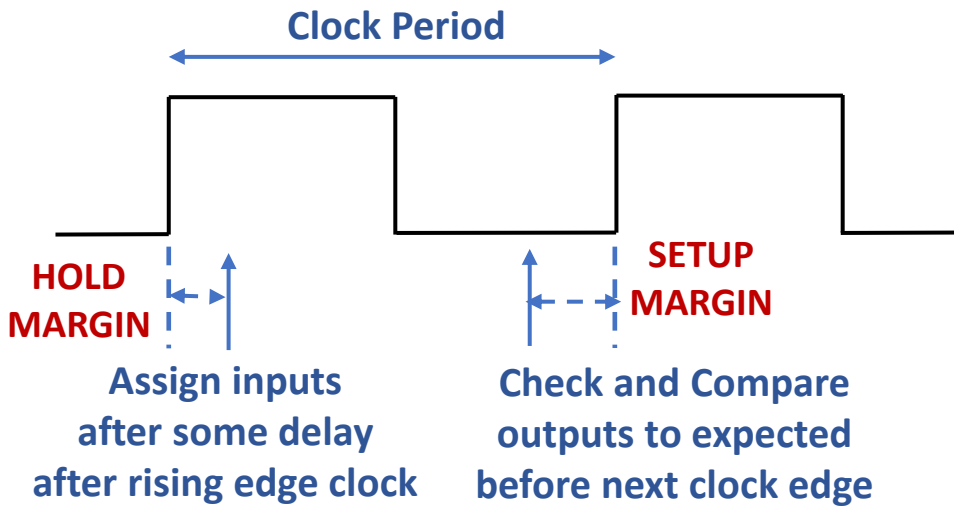
Testbench With Vectors and Self-Checking

- **Write testvectors file which includes : inputs and expected output values**
  - Generate clock for assigning inputs, reading outputs
  - Read testvectors file into array using initial procedural block
  - Assign inputs, get expected outputs from DUT using always procedural block
  - Compare outputs to expected outputs and report errors using always procedural block



Testbench With Vectors and Self-Checking





Approach A

Approach A

- **Testbench clock used to synchronize I/O**
  - Same Clock can be used for DUT clock
- **Assign inputs on rising edge**
- **Compare outputs with expected outputs on falling edge**

Approach B

- **Testbench clock used to synchronize I/O**
  - Same Clock can be used for DUT clock
- **Assign inputs after hold margin (real world usecase)**
- **Compare outputs with expected outputs on falling edge**

Testbench With Vectors and Self-Checking

- **Step 1 : Create test vectors file : test_vectors.txt**
  - **Contains input vectors and expected output (<inputs>_<expected outputs>)**
    - 110_01 → **inputs** a=1, b=1, cin=0 and **expected outputs** sum=0, cout=1
    - 100_10
    - 101_01
    - 001_10
    - 000_00
    - 111_11

Testbench With Vectors and Self-Checking

❑ Step 2 : Instantiate DUT and Generate Clock

```
`timescale 1ns/10ps // timeunit=1ns, precision=10ps
module testbench_top; // Testbench module without ports
 logic clk, reset, in0, in1, carryin;  // local variables to connect to DUT ports
 logic[1:0] result, expected_val;
 logic[4:0] test_vectors[10000:0]; // array of test vectors
 logic[31:0] test_vector_idx, errors; // index for test vector and error counter

 fulladder DUT(    // instantiate design under test
 .a(in0),
 .b(in1),
 .cin(carryin),
 .cout(result[1]),
 .sum(result[0]));

 // generate clock with 10ns period
 always // no sensitivity hence always executes
 begin
  clk = 1;
  #5; // wait for 5ns
  clk = 0;
  #5; // 10ns period end
 end
 .....
```

Testbench With Vectors and Self-Checking

❑ Step 2 : Read test vectors into array and apply reset sequence

```
// at start of test, load vectors, apply and wait for reset
// Note: $readmemh reads testvector files written in hexadecimal
initial begin
  // read input and expected outputs from testvectors.txt file
  $readmemb(“testvectors.txt", test_vectors);

 // initialize test vector index
 test_vector_idx = 0;

 // initialize error counter
 errors = 0;

 // Apply reset and wait for rest to end
 reset = 1;
 #38;
 reset = 0;
end
```

Testbench With Vectors and Self-Checking

❑ Step 3 : Assign inputs and expected outputs

```
// Apply test vectors to inputs at rising edge of clock after some delay
always@(posedge clk) begin
  #1; // wait for some delay after rising edge of clk

  // apply test vectors to input and store expected outputs to a local variable
 {a, b, cin, expected_val} = test_vectors[test_vector_idx];
end
```

- **Apply inputs with some delay (1ns) after rising edge of clock**
  - This is important Inputs should not change at the same time with clock

- **Ideal circuits (HDL code) are immune, but real circuits (netlists) may suffer from hold violations.**

Testbench With Vectors and Self-Checking

❑ **Step 4 : Compare outputs with expected outputs**

```
// Compare results on failing edge of the clock
always@(negedge clk) begin
  if (!reset) begin // skip during reset
    if (result !== expected_val) begin
      $display("Error: inputs = %b", {a, b, cin});
      $display("  outputs = %b (%b expected)", result, expected_val);
      errors = errors + 1;  // increment error count
    end
  end

  // increment array index and read next test vector
  vectornum = vectornum + 1;
  if(testvectors[vectornum] === 4'bx) begin
    $display("%d tests completed with %d errors", test_vector_idx, errors);
    $finish; // terminate simulation
  end
end
endmodule
```

Why Verification Signoff is Difficult ?

- **How long would it take to test a two input 32-bit adder?**
  - In such an adder there are 64 bit inputs = 2^{64} possible inputs
  - That makes around 1.85 10^{19} possibilities
  - If one input tested in 1ns, then 109 inputs per second can be tested
    - or 8.64 x 10^{14} inputs per day
    - or 3.15 x 10^{17} inputs per year
  - Considering above mentioned it will need **58.5** years to test all possibilities

- **Directed testing is not feasible for all circuits, we need alternatives**
  - Verify in simulation combination of directed test and corner cases
  - Complement with Assertion Based Formal verification methods
  - And also use hardware accelerator platforms such as FPGA based, Mentor Veloce, Cadence Pallidium and to accelerate simulation

SystemVerilog System Tasks and Functions

SystemVerilog System Tasks and Functions

- **SystemVerilog includes pre-defined tasks and functions for below mentioned usage:**
  - Simulation time and Simulation control
  - Standard output display and File I/O
  - Timescale
  - Data type conversions
  - Timingchecks
  - Waveform dumping

- **System tasks and function names begin with a dollar sign ($).**

- **System tasks and functions are not synthesizable**
  - Many of the system tasks and functions used in testbench for simulation purpose
  - Synthesis compiler tools ignores system functions even if included in synthesizable RTL model

- **System tasks that extract data, like $monitor needs to be in an initial or always block**

SystemVerilog System Tasks and Functions

Type	Purpose	System Task/Function List
Simulation Control	simulation control tasks allow user to either stop or reset or quit simulation	**$finish, $stop, $reset,$fatal**
Standard I/O Display	To display signal and variable values as text on the screen during simulation.	**$display, $strobe, $monitor, $write**
File I/O	To write signal and variable values in a file and read content from a file	**$fopen, $fclose, $fdisplay, $fstrobe, $fmonitor, $fwrite, $readmemb, $readmemh**
Timescale	To print timescale and timeformat for simulation	**$timeformat, $printtimescale**
Simulation Time	To return current simulation time either as a 64-bit integer, a 32-bit integer, and a real number	**$time, $stime, $realtime**
Timing Checks	Tasks for timing checks and to support timing verification during simulation, i.e. dynamic timing analysis	**$setup, $hold, $period, $skew, $width, $nochange, $recovery, $setuphold**
Data type Conversion	To convert from one data type to another	**$itor, $rtoi, $bitstoreal, $realtobits**

System Tasks and Functions

Type	Purpose	System Task/Function List
Waveform dumping	To dump all variable changes to a simulation viewer	**$dumpfile, $dumpvar, $dumpon, $dumpoff, $dumpall**
Assertion severity control	To specify messages with severity	**$error, $warning, $info, $fatal, $assertoff, $assertkill**
Random number	To generate random numbers	**$random, $urandom, $urandom_range, $srandom**

$finish and $stop

❑ $finish and $stop

- **$finish** when invoked it exits the simulator and gives control back to the operating system
- **$stop** when invoked it suspends simulation, puts execution in interactive mode where user can enter commands to further advance simulation

❑ Syntax :

- **$finish**[(N)] and **$stop**[(N)] where N can take value of 0,1,2
- argument(N) is optional
- if argument (N) is provided then diagnostic message will be printed on screen
- default value of N is 1 for **$stop** and 0 for **$finish** if not specified

N	message
0	No Message
1	Print simulation time and location
2	Print simulation time, location, memory consumption and CPU time

$finish and $stop Example

```
module clock_generator;
  logic clock;
  initial begin
    clock = 0;
    $monitor("at time=%g value of clock=%b", $time, clock);
    #41ns;
    $finish(1);
  end
  always@(clock)
    #10ns clock <= !clock;
endmodule
```

Simulation output if using $finish

```
at time=0 value of clock=0
at time=10 value of clock=1
at time=20 value of clock=0
at time=30 value of clock=1
at time=40 value of clock=0
$finish called from file "testbench.sv", line 11.
$finish at simulation time 41
Time: 41 ns
Done
```

Note: $finish quits simulation and control goes back to operating system

```
module clock_generator;
  logic clock;
  initial begin
    clock = 0;
    $monitor("at time=%g value of clock=%b", $time, clock);
    #41ns;
    $stop(1);
  end
  always@(clock)
    #10ns clock <= !clock;
endmodule
```

Simulation output if using $stop

```
at time=0 value of clock=0
at time=10 value of clock=1
at time=20 value of clock=0
at time=30 value of clock=1
at time=40 value of clock=0
$stop at time 41 Scope: test1 File: testbench.sv Line: 11
vsim>
```

Note: $stop suspended simulation and simulator puts in interactive mode by returning to command line vsim>

$display and $write

❑ $display and $write :

- **$display** and **$write** are similar to **print** function in the ANSI C language, when invoked immediately prints its arguments
- Arguments are printed in the same order it is specified.
- Both these tasks gets executed in the active region of execution
- **$write** and **$display** tasks work in the same way and the only difference is that the $display task adds a new line character at the end of the output, while the $write task does not
- Both tasks have a special character (**%**) to indicate that the information about signal value is needed. It is known as format specification.
- **Example Usage :**

```
module test_display_task;
  logic a, b;
  initial begin
    a = 1;
    b = 0;
    $display("Value of a is: %b", a) ;
    $display("Value of b is: %b", b) ;
  end
endmodule
```

%b is a binary format specifier hence display prints binary value of 'a' and 'b'

```
module test_write_task;
  logic a, b;
  initial begin
    a = 1;
    b = 0;
    $write("Value of a is: %b", a) ;
    $write("Value of b is: %b", b) ;
  end
endmodule
```

Simulation output if using $display

Value of a is: 1
Value of b is: 0

Simulation uutput if using $write

Value of a is: 1Value of b is: 0

Note: $write does not add newline character by default unlike $display

When does $monitor and $display execute ?

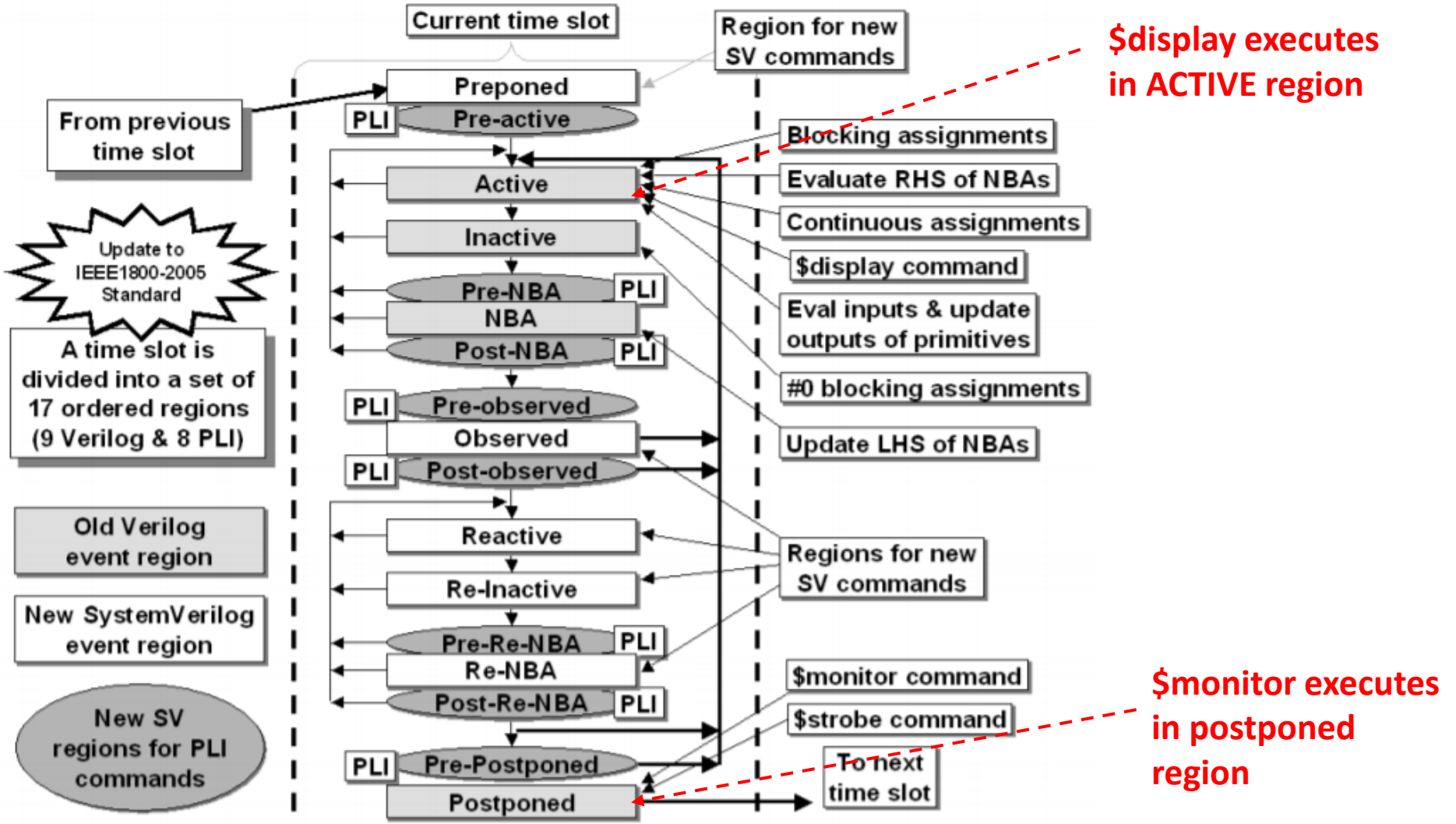


Figure 4 - SystemVerilog-2005 event regions with PLI regions shown

Credit to : Clifford E. Cummings (sunburst design), Arturo Salz (Synopsys)

$monitor and $strobe

❑ $monitor and $strobe :

- **$monitor** statement monitors the values of variables throughout the simulation
  - only displays the value of a variable or a signal whenever its value changes
  - whereas **$display** and **$write** only prints arguments once when invoked
- Only one **$monitor** process can run at a time.
  - If multiple **$monitor** statements specified, then current **$monitor** process will get canceled and get replaced by new **$monitor** process.
- **$monitor and $strobe** gets executed in the postponed region of execution !
- **$monitor** and **$strobe** tasks works in the same way and the only difference is :
  - **$strobe** displays the value of a variable or a signal at the end of the current time step and
  - **$strobe** only prints value of variables once.
- Arguments are printed in the same order it is specified.
- Has a special character (**%**) to indicate that the information about signal value is needed.
  - It is known as format specification.

❑ Example Syntax :

- **$strobe**("At time=**%g** using strobe value of sum = **%0h**",**$time**, data);
- **$monitor**("At time=**%g** using monitor value of p = **%0h**",**$time**, data);

Example for $display, $write, $strobe, $monitor using non-blocking statement

```
module test1();
  reg [7:0] p;
  initial begin
    p = 8'h16;
    #5 p <= 8'h22; // since non-blocking, LHS is updated in INACTIVE region
    $display("\t At time=%g using display value of p = %0h",$time, p);
    $write("\t At time=%g using write value of p = %0h\n",$time, p);
    $strobe("\t At time=%g using strobe value of p = %0h",$time, p);
    $monitor("\t At time=%g using monitor value of p = %0h",$time, p);
    #5 p <= 8'h44;
    #5 p <= 8'h66;
   end
endmodule
```

%h is a hexadecimal format specifier hence $monitor prints hexadecimal value for 'p'

Simulation Output

At time=5 using display value of p = 16
At time=5 using write value of p = 16
At time=5 using monitor value of p = 22
At time=5 using strobe value of p = 22
At time=10 using monitor value of p = 44
At time=15 using monitor value of p = 66

❑ Note :

- **@5, $display** and **$write** prints value of variable 'p' as 16 since both tasks executes in active region of NBA
- **$write** requires explicit newline character (\n) to be specified to have next print in next line
- **@5, $monitor** and **$strobe** prints value of variable 'p' as 22 since both tasks executes in postponed region of NBA (non blocking assignment)
- **@5, $strobe** only prints value of 'p' once which is 22
- **$monitor** prints value of 'p' each time it changed hence it has multiple prints for values 22, 44, 66 at timeunits **@5, @10** and **@15** respectively

Example for $display, $write, $strobe, $monitor using blocking statement

```
module test2();
 reg [7:0] p;
 initial begin
   p = 8'h16;
   #5 p = 8'h22; // since blocking, LHS is updated in ACTIVE region
   $display("\t At time=%g using display value of p = %0h",$time, p);
   $write("\t At time=%g using write value of p = %0h\n",$time, p);
   $strobe("\t At time=%g using strobe value of p = %0h",$time, p);
   $monitor("\t At time=%g using monitor value of p = %0h",$time, p);
   #5 p = 8'h44;
   #5 p = 8'h66;
   p <= 8'h77;
 end
endmodule
```

At same time unit, variable 'p' has two assignment. One blocking with value 66 and other is non-blocking with value 77. Since $monitor executes in non-active region, it will print overridden value of 77 for 'p'

Simulation Output

```
At time=5 using display value of p = 22
At time=5 using write value of p = 22
At time=5 using monitor value of p = 22
At time=5 using strobe value of p = 22
At time=10 using monitor value of p = 44
At time=15 using monitor value of p = 77
```

❑ Note :

- Changing non-blocking assignment to blocking assignment**, $display** and **$write** at timeunit **@5** prints value of variable 'p' as 22 instead of 16.
- **@15, $monitor** prints value of 'p' as 77 instead of 66, since in same timeunit there is a non-blocking assignment statement with value 77 assigned to 'p' and 77 will be assigned in non-active region of NBA
  - This is due $monitor executes in non-active region of NBA

Format Specifier and Escape Characters

- **Format Specifiers for $display, $write, $monitor, $strobe**
  - Required to print variables and signals in one of the format as shown in table below
  - Default format specifier is %d if not specified
- **$display, $write, $monitor, $strobe can also have escape characters**

Format Specifier	Description
%d or %D	Decimal format
%b or %B	Binary format
%h or %H	Hexadecimal format
%o or %O	Octal format
%c or %C	ASCII character format
%v or %V	Net signal strength
%m or %M	Hierarchical name
%s or %S	As a string
%t or %T	Current time format
%e or %f or %g or %G	Real format

Escape Characters	Description
\n	Newline
\t	Tab
\”	Double quote
\\	Backslash

$time, $stime and $realtime

- **$time, $stime and $realtime :**
  - When invoked, it returns current simulation time
  - It returns 64-bit unsigned value, rounded to the nearest unit
  - **$stime** returns a 32-bit unsigned value, truncating large time values.
  - **$realtime** returns a real number
  - These functions have no inputs

- **Example Syntax** :

```
integer current_time1 ;
curr_time1 = $time ;

integer curr_time2 ;
curr_time2 = $stime ;

real curr_time3 ;
curr_time3 = $realtime ;
```

$fatal, $error, $warning, $info

- **$fatal :**
  - Run-time fatal error, which terminates the simulation with an error code.
  - First argument passed to **$fatal** shall be consistent with the corresponding argument to the Verilog **$finish** system task, which sets the level of diagnostic information reported by the tool.
  - Calling **$fatal** results in an implicit call to **$finish**.

- **$error:**
  - When invoked it will generate run-time error message

- **$warning:**
  - When invoked it will generate run-time warning, which can be suppressed in a tool-specific manner.

- **$info:**
  - Similar to $display

SystemVerilog Compiler Directives

SystemVerilog Compiler Directive

- **Compiler directive may be used to control the compilation of a SystemVerilog description.**
  - The grave accent mark, `, denotes a compiler directive.
  - A directive is effective from the point at which it is declared to the point at which another directive overrides it, even across file boundaries.
  - Compiler directives may appear anywhere in the source description, but it is recommended that they appear outside a module declaration.

- **Some of the compiler directives supported in SystemVerilog are listed below :**
  - `include
  - `define
  - `ifdef, `else, `ifndef, `undef
  - `timescale
  - `import
  - `defaultnetype
  - `nounconnected_drive and `unconnected_drive

`include directive

- **`include inserts the contents of a specified file into a file in which it was called.**
  - Compilation proceeds as though the contents of the included source file appear in place of the `include command
  - File name should be given in quotation marks (") and
  - File name can be given using full or relative path

```
`include "half_adder.sv"
module full_adder(
  input logic a, b, cin,
  output logic sum, cout);
  logic w1, w2, w3;
  half_adder ha1 (a, b, w1, w2);
  half_adder ha2(w2, cin, w3, sum);
  or #1 (cout, w1, w3);
endmodule: half_adder
```

Intent to include half adder code which is defined in separate file

Filename : full_adder.sv

```
module half_adder(
  input logic a, b,
  output logic sum, cout);
  and #1 (cout, a, b);
  xor #2 (sum, a, b);
endmodule: half_adder
```

Filename : half_adder.sv

compiler will treat LHS code as RHS

=

```
module half_adder(
  input logic a, b,
  output logic sum, cout);
  and #1 (cout, a, b);
  xor #2 (sum, a, b);
endmodule: half_adder

module full_adder(
  input logic a, b, cin,
  output logic sum, cout);
  logic w1, w2, w3;
  half_adder ha1 (a, b, w1, w2);
  half_adder ha2(w2, cin, w3, sum);
  or #1 (cout, w1, w3);
endmodule: half_adder
```

Filename : full_adder.sv

`ifdef, `else, `ifndef, `endif directive

- **`ifdef, `else, `ifndef can** be used to decide which lines of Verilog code should be included for the compilation
  - If macro name after **`ifdef** is defined, then all lines between **`ifdef** and **`else** will be compiled. Otherwise, only lines between **`else** and **`endif** will be compiled.
  - **`ifndef** is same as **`ifdef** except it evaluates true if macro after it is not defined

```
`define behavioral
module full_adder(
 input logic a, b, cin,
 output logic sum, cout
);
 `ifdef behavioral
   assign {cout, sum} = a + b + cin;
 `else
   logic w0, w1, w2;
   xor x0(w0, b, a);
   and a0(w1, b, a);
   and a1(w2, w0, cin);
   or r0(cout, w2, w1);
   xor x1(sum, w0, cin);
`endif
endmodule: full_adder
```

Since behavioral macro is defined, compiler will include lines within `ifdef body

if behavioral macros was not defined, compiler will include lines within `else body

compiler will choose behavioral implementation if behavioral macro is defined

```
module full_adder(
  input logic a, b, cin,
  output logic sum, cout);
  assign {cout, sum} = a + b + cin;
endmodule: full_adder
```

compiler will choose gatelevel implementation if behavioral macro is not defined

```
module full_adder(
  input logic a, b, cin,
  output logic sum, cout);
  logic w0, w1, w2;
  xor x0(w0, b, a);
  and a0(w1, b, a);
  and a1(w2, w0, cin);
  or r0(cout, w2, w1);
  xor x1(sum, w0, cin);
endmodule: full_adder
```

`define and `undef directive

- **`define** directive is used to define the text macros and constants
  - `define <macro func name> (ARGS) is used to define a macro function that can generate RTL based on ARGS
  - `define <constant name> <optional value> is used to declare a synthesis-time constant
- **`undef** directive is used to remove the definition of text macros and constants created by `define directive

filename : memory.sv

```
`include "constants.vh"
module memory(
  input logic [`ADDR_BITS - 1:0] address,
  .....
  .....
  output logic [`LOG2(`NUM_WORDS) - 1:0] data
);
 // memory model implementation
 .....
 .....
endmodule: memory
```

Includes the content of constants.vh in memory.sv file

`LOG2 expands to the macro defined in constants.vh

filename : constants.vh

```
`ifndef CONSTANTS
  `define CONSTANTS
  `define ADDR_BITS 16
  `define NUM_WORDS 32
  `define LOG2(x) \
          (x <= 2) ? 1 : \
          (x <= 4) ? 2 : \
          (x <= 8) ? 3 : \
          (x <= 16) ? 4 : \
          (x <= 32) ? 5 : \
          (x <= 64) ? 6 : \
          -1
`endif
```

`timescale directive

- **`timescale directive tells simulator what #delay specified within a module should mean in terms of time. It has two component(s)**
  - Time unit and Time precision

- **Syntax**

  `timescale <timeunit>/<timeprecision>

- **Example :**

  `timescale 1ns/1ns

  `timescale 1ns/1ps

  `timescale 10ns/1ns

- **Note :**
  - The simulation time and delay values are measured using time unit.
  - The precision is how delay values are rounded before being used in simulation.
  - **Time precision value has to be equal to or smaller than time unit value**

`timescale directive

- **Time unit has two parts :**
  - Magnitude and Unit
  - In **`timescale 10ns**/1ns specification, **10** is the magnitude and **ns** is the unit of time

- **Legal values for magnitude are :** 1, 10, 100

- **Legal Time Unit Values :**

Legal Time Unit	Real World Time Unit
s	Seconds
ms	milliseconds
us	Microseconds
ns	Nanoseconds
ps	Picoseconds
fs	femtoseconds

- **Example Syntax**

Syntax	Legal or Illegal ?
`timescale 1ns/1ns	Legal syntax
`timescale 10ns/1ns	Legal syntax
`timescale 100ns/1ns	Legal syntax
`timescale 1000ns/1ns	illegal syntax since magnitude value 1000 is not a legal value
`timescale 1ns/10ns	illegal syntax since timeprecision > timeunit
`timescale 1us/10ns	Legal syntax

`timescale directive

- **Rule :** Multiply each #delay value in module by timeunit and then round the result to the nearest number based of time precision

Example A

```
`timescale 10ns/10ns
module t_directive();
 reg enable;
 initial begin
  enable = 0;
  #4.55;  // will result in 50ns delay
  enable = 1;
 end
endmodule
```

Example B

```
`timescale 10ns/1ns
module t_directive();
 reg enable;
 initial begin
  enable = 0;
  #4.55;  // will result in 46ns delay
  enable = 1;
 end
endmodule
```

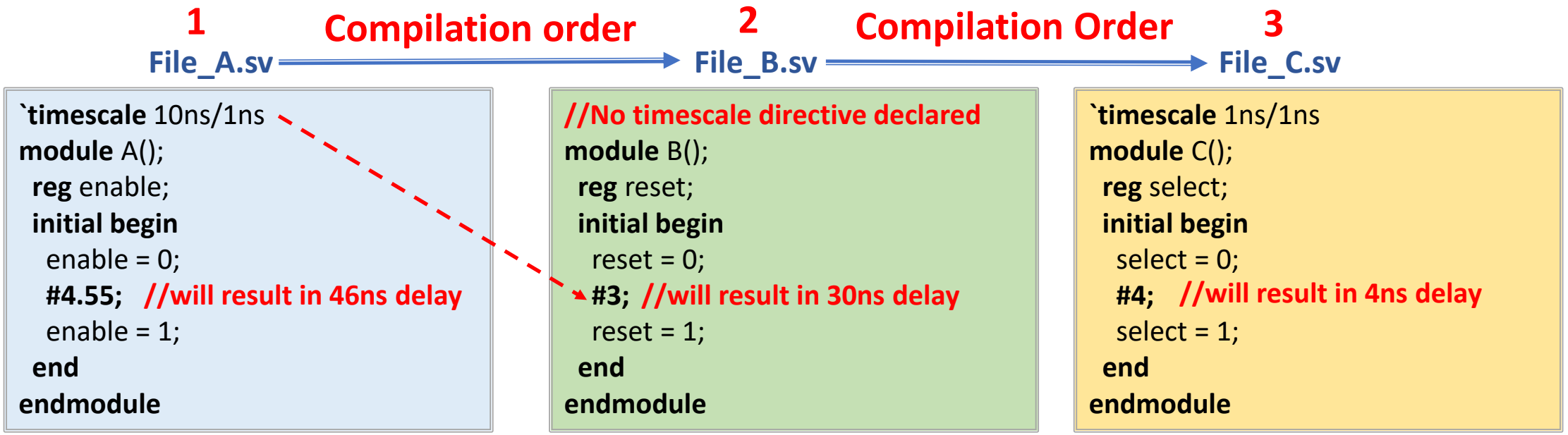
Example C

```
`timescale 1ns/1ns
module t_directive();
 reg enable;
 initial begin
  enable = 0;
  #4.55;  // will result in 5ns delay
  enable = 1;
 end
endmodule
```

- **Example A :**
  - #4.55 x 10ns timeunit = 45.5ns
  - Simulator will round 45ns to closet integer multiple of 10ns timeprecision resulting in 50ns
- **Example B :**
  - #4.55 x 10ns timeunit = 45.5ns
  - Simulator will round 45.5ns to closet integer multiple of 1ns timeprecision resulting in 46ns
- **Example C :**
  - #4.55 x 10ns timeunit = 4.55ns
  - Simulator will round 4.55ns to closet integer multiple of 1ns timeprecision resulting in 5ns

`timescale directive

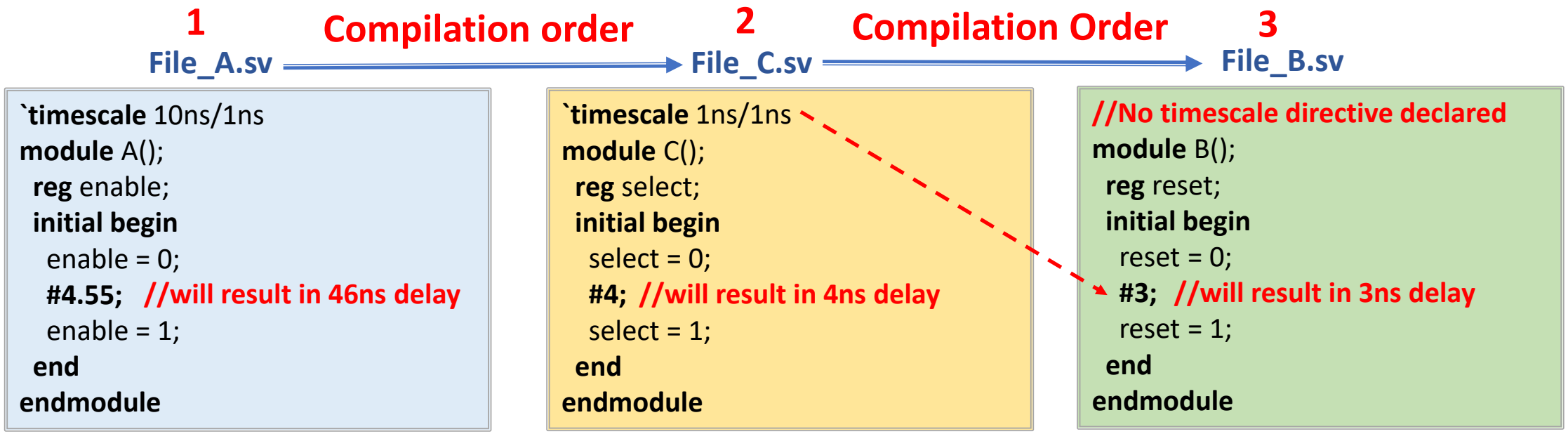
- **`timescale** directive can be defined in multiple SystemVerilog source files
- **`timescale** directive is not bound to specific module. It is effective until next **`timescale** directive is encountered
  - SystemVerilog source file without a **`timescale** directive is dependent on the order in which the file is compiled relative to previous files and will inherit timescale from last file which has timescale declared.
- **Example :** (File_A.sv compiled first, then File_B.sv and then File_C.sv)



`timescale 10ns/1ns from File_A.sv is in effect for module B since No timescale directive declared locally in File_B.sv

`timescale directive

- Changing SystemVerilog File_B.sv order of compilation below will result in different simulation behavior for module B when compared to previous compilation order



timeprecision and timeunit directive

- **SystemVerilog provides timeunit and timeprecision to specify time unit and precision within a module**
  - Binds the time unit and precision information directly to a module, interface or program block
  - Resolves the ambiguity and file order dependency that exist
  - The **timeunit** and **timeprecision** statements must be specified immediately after the module, interface, or program declaration, before any other declarations or statements.

- **Syntax**

  timeunit <value>;

  timeprecision <value>;

- **Example :**

```
module adder (input wire [63:0] a, b,
  output reg [63:0] sum,
  output reg carry);
  timeunit 1ns;
  timeprecision 10ps;
  ...
endmodule
```

Precedence in case of mixed declaration of time precision and time unit

```
// external time unit and precision
`timeunit 1ns;
`timeprecision 1ns;

module my_chip ( ... );
 timeprecision 1ps; // local precision (priority over external)

 always @(posedge data_request) begin
   #2.5 send_packet;  // uses external units & local precision
   #3.75us check_crc;  // specified inline units take precedence
 end

 task send_packet();
   ...
 endtask

 task check_crc();
  ...
 endtask
endmodule
```

Credit : Stuart Sutherland

```
// directive takes precedence over external
`timescale 1ps/1ps

module FSM ( ... );
 timeunit 1ns; // local units take priority over directive

 always @(state) begin
   #1.2 case (state) // uses local units & external precision
          WAIT: #20ps ...; // specificied units take precedence
    ....
  end
endmodule
```

Credit : Stuart Sutherland

Time unit and precision search order

- **SystemVerilog compiler/simulator will derive time unit and precision in below mentioned specific search order :**

1. if specified time unit specified as part of the time value.
2. else if local time unit and precision specified in the module
3. else if in case of nested module, use the time unit and precision in parent module
4. else if specified, use the `timescale time unit and precision in effect when the module was compiled.
5. else if specified, use the time unit and precision defined in compilation scope
6. else use the simulator's default time unit and precision

Simulator, Logic Simulation, Waveform, EDA Tools

Simulator

- **Simulator role is approximating reality. Design realized in virtual environment !**
  - Simulator creates an artificial universe that mimics the future real circuit design
  - It lets the designer interact with the design before it is manufactured, and enables designer to correct flaws and problem earlier
    - Simulator takes RTL code and simulates design using stimulus provided in testbench code
    - Generates waveforms to provide approximate behavior of design prior to manufacturing.
  - Many physical characteristics are simplified or even ignored to ease the simulation task
    - Four state digital simulator assumes only possible values of signals as 0,1,X,Z,
    - In the physical/analog world, the signal value can have infinite value since it is continuous function of voltage and current over a copper wire.
  - Simulator does not correct automatically incorrect description of the design written in HDL
  - It is the responsibility of verification engineer, not the simulator, to apply legal set of stimulus to design under test based on real world usecase.
  - Simulator is not a static tool ! It requires stimulus to mimic design behavior and provide results

Logic Simulation

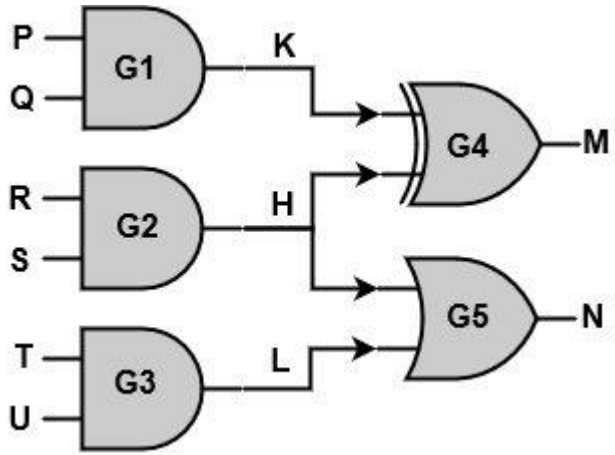
- **Two types of Logic simulation**
  - Event driven
    - RTL simulation, Verilog/VHDL, slower and more accurate
  - Cycle based simulation
    - Faster and lesser accurate then event driven simulation

Event Driven Simulation

- **Main function of an event driven simulator is to detect events, and schedule gate simulations in response to them.**
  - An event is defined to be a change in the value of a net/signal
  - If no events occur, implying that there are no net changes, then no gates will be simulated.
  - Event driven simulation is designed to remove unwanted gate simulations to achieve higher simulation performance of a circuit described using HDL

Event Driven Simulation (Combinational Circuit)

- **In case of combinational circuit :**
  - Event driven simulator evaluates gates only when there is a change to its input net/signal value
  - No evaluation of gates if there are no events



Combinational Circuit

Event Driven Simulation Results Table

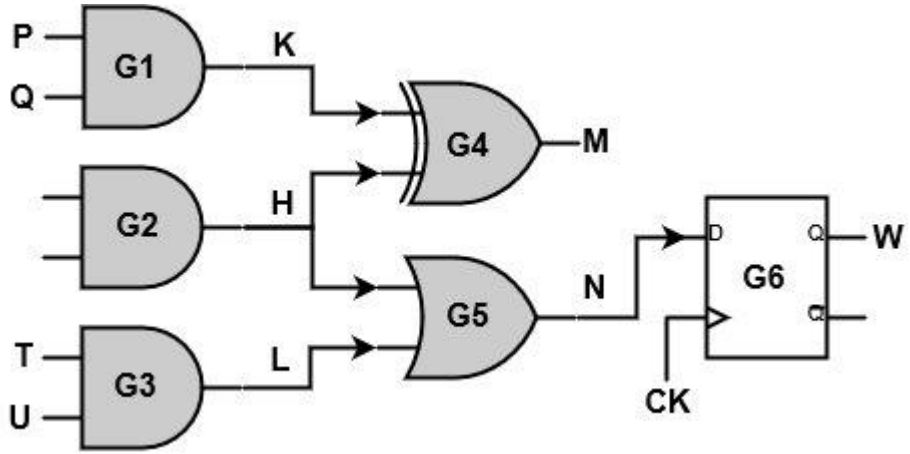
E = Gate Evaluated **N** = Gate Not Evaluated

Time	P	Q	R	S	T	U	K	H	L	M	N	G1	G2	G3	G4	G5
0 ns	0	0	0	0	0	0	0	0	0	0	0	E	E	E	E	E
2 ns	1	0	0	0	1	1	0	0	1	0	1	E	N	E	N	E
4 ns	1	0	0	0	1	1	0	0	1	0	1	N	N	N	N	N
6 ns	1	1	1	1	1	1	1	1	1	0	1	E	E	N	E	E
8 ns	0	0	1	1	1	1	0	1	1	1	1	E	N	N	E	N

Note : Even though output of XOR gate G4 did not change from time 4 ns to 6 ns, gate G4 is still evaluated by simulator since G4 gate input signal K and H value changed from 0 to 1. This is because all simulator knows that is a logic gate with 2 input and 1 output, whether it is a 2 input AND gate or XOR gate, it does know.

Event Driven Simulation (Sequential Circuit)

- **In case of sequential circuit :**
  - Event driven simulation evaluates sequential element (such as flipflop) only when there is change to its clock signal



Sequential Circuit

Input signal N propagates to output W only on clock event

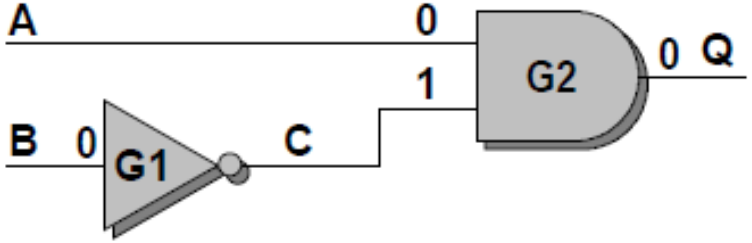
Time	P	Q	R	S	T	U	K	H	L	M	N	CK	W	G1	G2	G3	G4	G5	G6
0 ns	0	0	0	0	0	0	0	0	0	0	0	0	0	E	E	E	E	E	E
2 ns	1	0	0	0	1	1	0	0	1	0	1	0	0	E	N	E	N	E	N
4 ns	1	0	0	0	1	1	0	0	1	0	1	1	1	N	N	N	N	N	E
6 ns	1	1	1	1	1	1	1	1	1	0	1	1	1	E	E	N	E	E	N

Event Driven Simulation Results Table

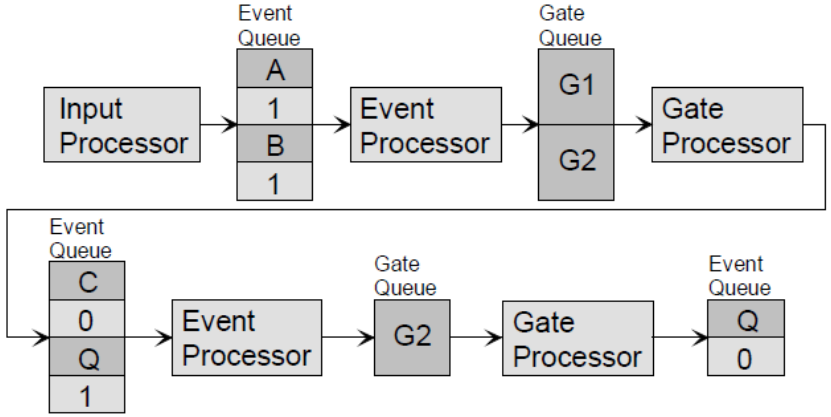
E = Gate Evaluated **N** = Gate Not Evaluated

Two Phase Event Driven Simulation Algorithm

- Inputs which has change in value, input processor adds these events to event queue
- For each event in event queue, all fan out gates are placed in gate queue by event processor
- Gate processor evaluates each gate in gate queue and propagates input signal to its output after unit delay
- Figure illustrates several important points about event driven simulation.
  - G2 is simulated twice.
  - There are two events for the net Q, one which changes the value from zero to one and the second which changes Q from one to zero. Thus the value of Q changes briefly from zero to one and back to zero.



Evaluation for change in value for (A,B) from (0,0) to (1,1)



Two Phase Event Driven Simulation Algorithm

- Each execution of the Event Processor subroutine defines one unit of simulated time.
- Intermediate outputs can be stored after each execution of gate in gate queue
- The first execution of the Event Processor is designated as simulated time zero.
- Simulated time increases by one for each subsequent execution of the Event Processor

Source and Credit : Writing testbenches : Functional Validation of HDL Models book, Janick Bergeron

Cycle-Based Simulation

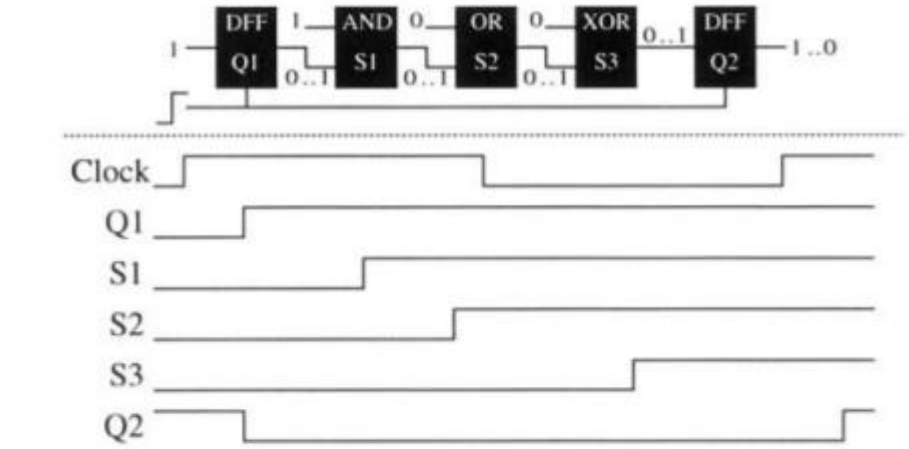
- **Cycle based simulator evaluates logic between state elements and/or ports in the single shot.**
  - Cycle-based simulators collapse combinatorial logic into equations
  - When the clock input rises, the value of all flipflops are updated using the input value returned by the pre-compiled combinatorial input functions

- **Cycle-based simulators have no notion of time within a clock cycle.**
  - All timing and delay information is lost.
  - Cycle based simulators assume that entire design meets the setup and hold requirements of all the flip flops

- **Each logic element is evaluated only once per cycle**
  - Significantly increase the speed of execution, but this can lead to simulation errors.

- **Cycle-based simulators assume that the active clock edge is the only significant event in the changing the state of the design.**
  - All other inputs are assumed to be perfectly synchronous with the active clock edge
  - Cycle-based simulators only function on synchronous logic

Cycle-Based Simulation Example



1. The event (rising edge) on the clock input causes the execution of the description of the flip-flop models, changing the output value of Q1 to '1' and of Q2 to '0', after a delay of 1 ns.
2. The event on Q1 causes the description of the AND gate to execute, changing the output S1 to '1', after a delay of 2 ns.
3. The event on S1 causes the description of the OR gate to execute, changing the output S2 to '1', after a delay of 1.5 ns.
4. The event on S2 causes the description of the XOR gate to execute, changing the output S3 to '1' after a delay of 3 ns.
5. The next rising edge on the clock causes the description of the flip-flops to execute, Q1 remaining unchanged but Q2 changing back to '1', after a delay of 1 ns.

1. When the circuit description is compiled, all combinatorial functions are collapsed into a single expression that can be used to determine all flip-flop input values based on the current state of the fan-in flip-flops.

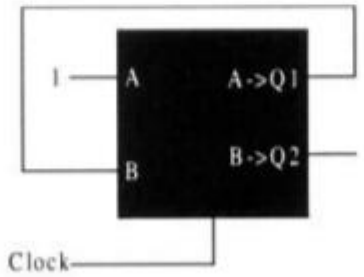
   For example, the combinatorial function between Q1 and Q2 would be compiled from the following initial description:

```
S1 = Q1 & '1'
S2 = S1 | '0'
S3 = S2 ^ '0'
```

   into this final single expression

```
S3 = Q1
```

   The cycle-based simulation view of the compiled circuit is shown in Figure 2-5.

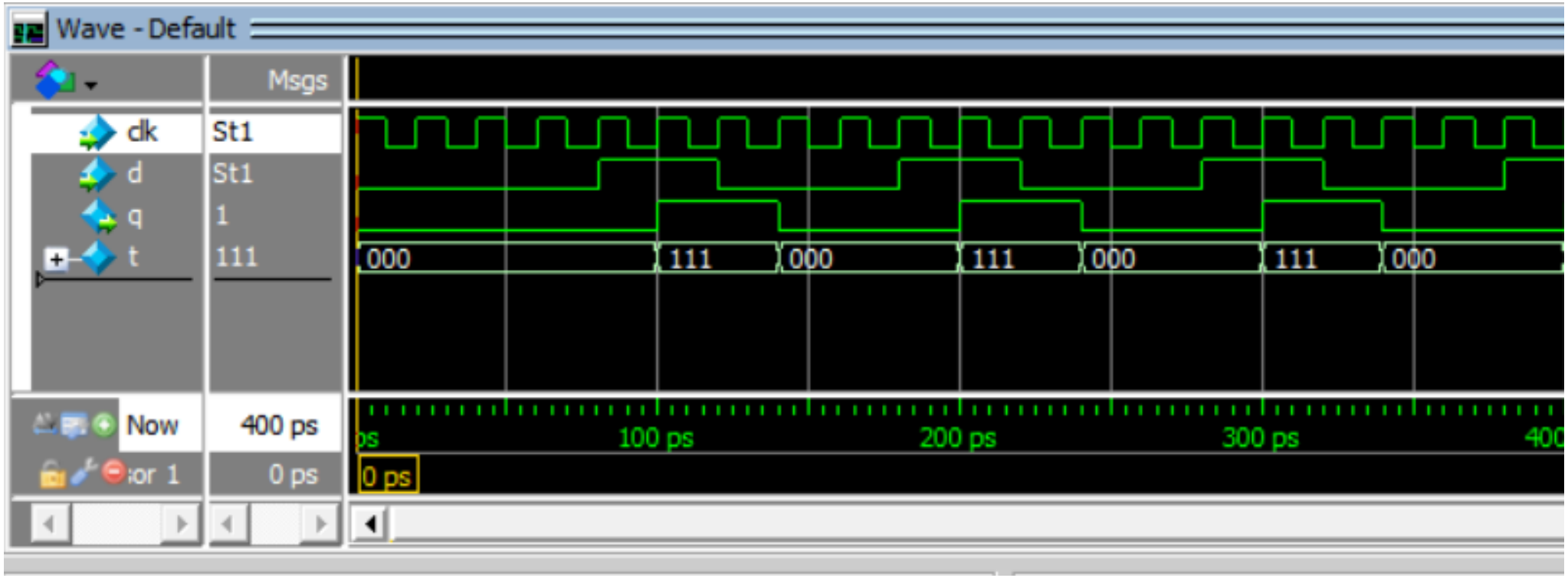


2. During simulation, whenever the clock input rises, the value of all flip-flops are updated using the input value returned by the pre-compiled combinatorial input functions.

Source and Credit : Writing testbenches : Functional Validation of HDL Models book, Janick Bergeron

What is a Waveform ?

- Waveform is a plot of signal values over period of time

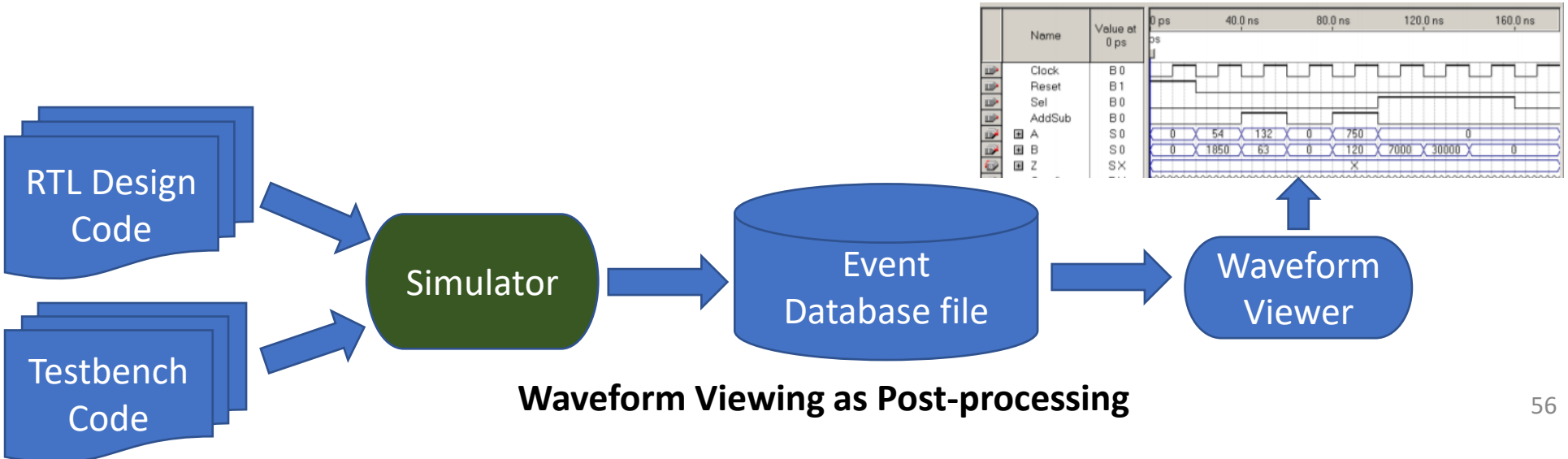


Waveform Viewer

- **Waveform viewer is a most common verification tool used in conjunction with simulator**
  - It allows users visualize the transition of multiple signal over time, and their relationship with other signal transitions.
  - Each simulator's graphical interface has native waveform viewer built into it.
  - Each simulator generates waveform / trace file in its own proprietary format
    - There is a common trace format VCD which each simulator can generated however it is a large and not very optimal compared to the format which each simulator generates.
  - Synopsys Verdi is most commonly used for waveform debugging and it works with all major simulators (VCS, VSIM, NSCIM).
    - Each simulator can interact with Verdi to eventually generate fsdb waveform file which can be viewed using Verdi nWave waveform viewer.

Waveform Viewer

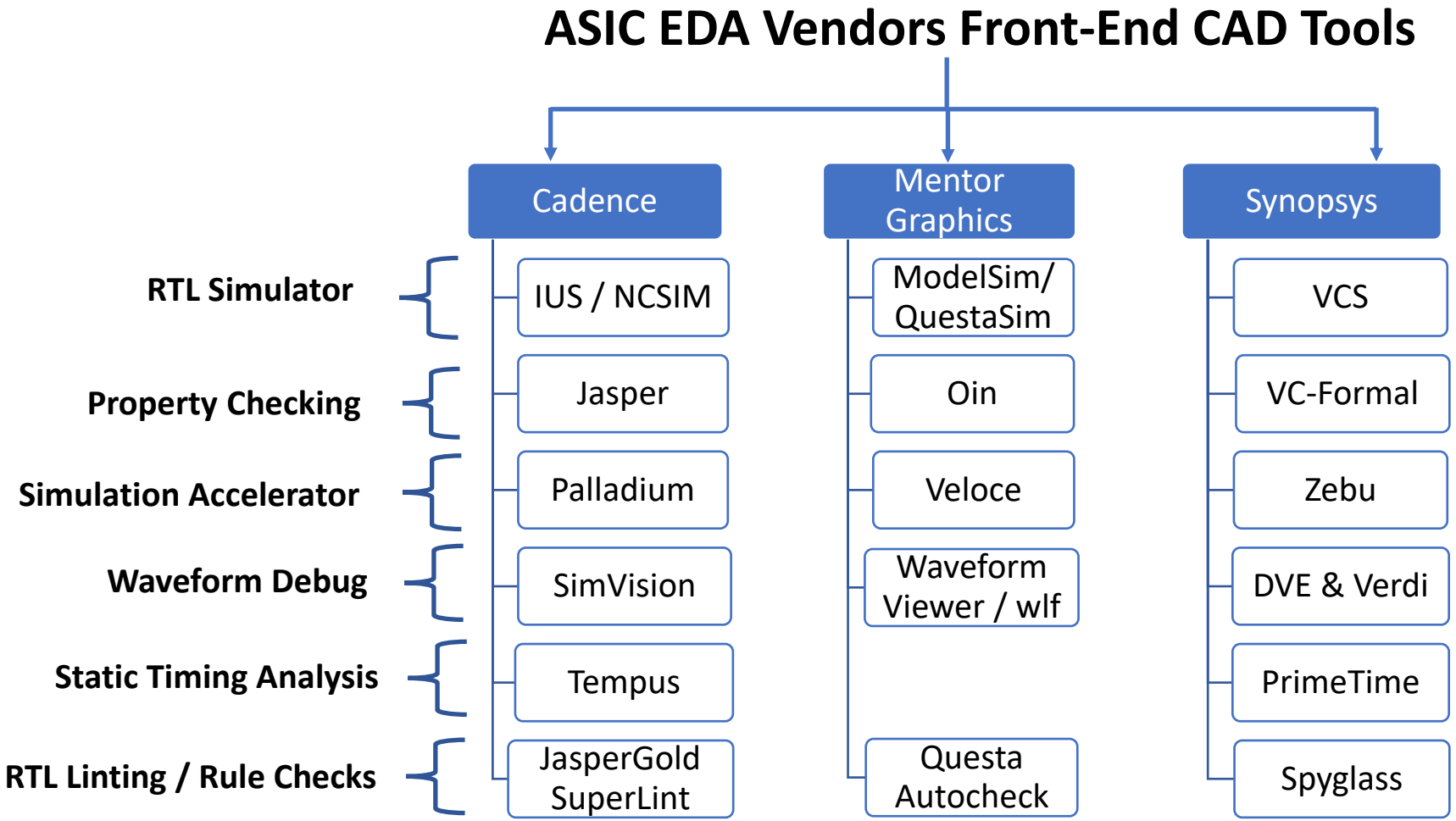
- **Waveform viewer is frequently used to debug simulation during RTL design and verification phase**
  - Waveform viewer can be used to inspect the behavior of the code is as expected
  - Can be used interactively while simulation is being run or once simulation has completed
  - Can play back the events that occurred during RTL and gate level simulation that were recorded in some trace file
  - Recording signal trace information while simulation is running, significantly reduces the performance of the simulator
  - Typically all signals are recording during RTL and testbench development phase.
    - Later, limited traces are recorded and waveforms are only generated for a failure test



Waveform Viewing as Post-processing

EDA Vendors

- **Three major EDA Vendors in field of ASIC RTL Design and Verification**
  - Cadence Design Systems
  - Mentor Graphics
  - Synopsys



ModelSim Compilation and Running Simulation Flow

Step 1 : Create Work Library	**vlib** work
Step 2 : Map work to physical directory	**vmap work** $HOME/worklib
Step 3 : Compile design files	**vlog –sv –work work** full_adder.sv half_adder.sv
Step 4 : Compile testbench files	**vlog –sv –work work** full_adder_testbench.sv
Step 5 : Elaborate and Run Simulation	**vsim work**.full_adder_testbench

Note :

- **vlog** is a compiler tool which compiles HDL source files
- **-sv** switch instructs **vlog** to compile source files as SystemVerilog code and follow SystemVerilog LRM
- If **-sv** switch is not passed to simulator, then **vlog** will treate source files as Verilog 2001 code format
- **vsim** does elaboration of code first and then runs simulation.
- End of **vsim** command waveform file with extension **.wlf** is generated
- All compile directives such as **`define, `ifdef, `timescale**, etc are considered during **vlog** stage
- SystemVerilog parameter constants are considered during **vsim** elaboration stage
- Connections between module ports is done during **vsim** elaboration stage