HARDWARE DESCRIPTION LANGUAGE FOR DIGITAL DESIGN

數位設計硬體描述語言

Verification

Materials partly adapted from "Digital System Designs and Practices Using Verilog HDL and FPGAs," M.B. Lin.

OUTLINE

- Function verification
- Simulation
- Test bench design
- Dynamic timing analysis
- Static timing analysis

FUNCTION VERIFICATION

VERIFICATION

- The goal of verification
  - To ensure 100% correct in functionality and timing
  - Spend 50 ~ 70% of time to verify a design
- Functional verification
  - Simulation
  - Formal proof
- Timing verification
  - Dynamic timing simulation (DTS)
  - Static timing analysis (STA)

FUNCTIONAL VERIFICATION

- Simulated-based functional verification
  - A test bench
  - Input stimuli
  - Output analysis
- Formal verification
  - A protocol
  - An assertion
  - A property
  - A design rule

MODELS OF DESIGN UNDER TEST

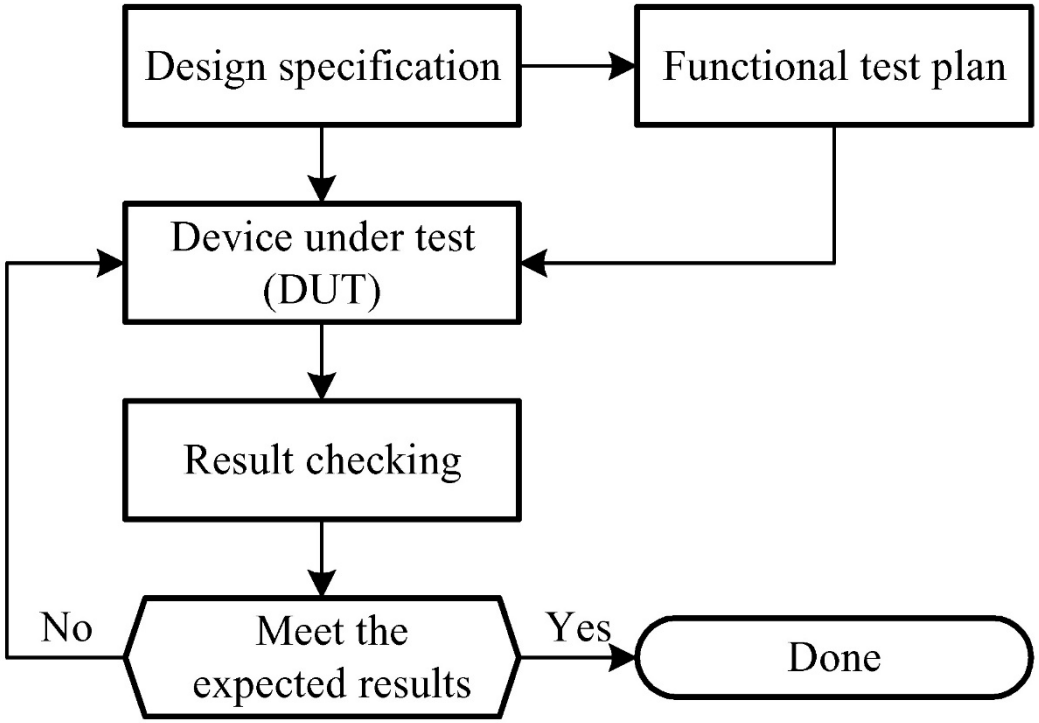
- Black box model
  - Only the external interfaces are known.
  - Most simulation-based verification environments begin here.
- White box model
  - Both external interfaces and internal structures are known.
  - Most formal verification environments
- Gray box model
  - A combination of both black box and white box models
  - Most simulation-based verification environments use this one.

TYPES OF ASSERTION

- Static assertion
  - Atomic and simple checks

- Temporal assertion
  - A sequence of events in time checks
  - Complex but useful

SIMULATION-BASED VERIFICATION



HIERARCHY OF FUNCTIONAL VERIFICATION

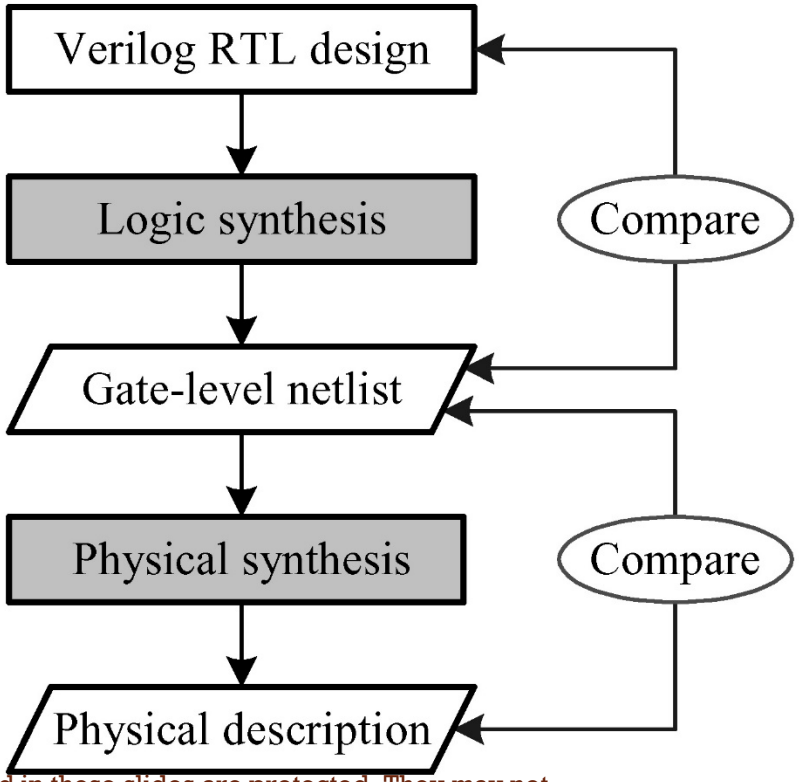
- Designer level (or block-level)
- Unit level
- Core level
- Chip level

A VERIFICATION TEST SET

- Verification test set includes at least
  - Compliance tests
  - Corner case tests
  - Random tests
  - Real code tests
  - Regression tests
  - Property check

FORMAL VERIFICATION

- Uses mathematical techniques
- Proves a design property



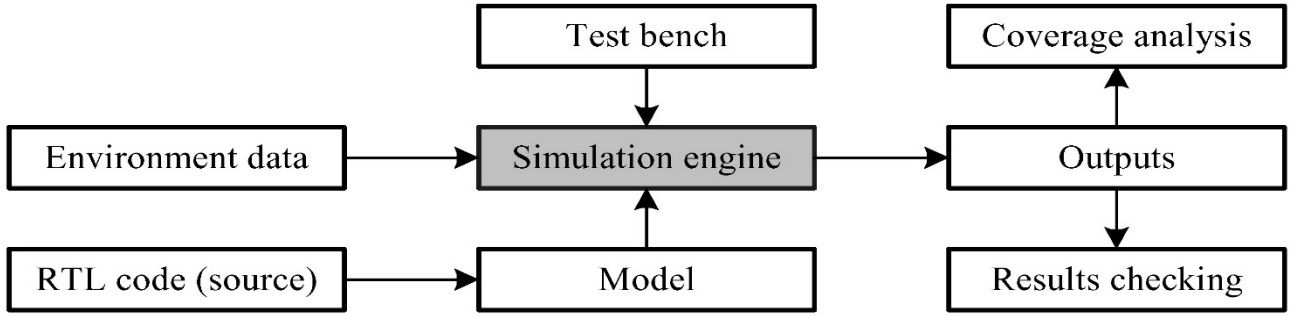
SIMULATION

TYPES OF SIMULATIONS

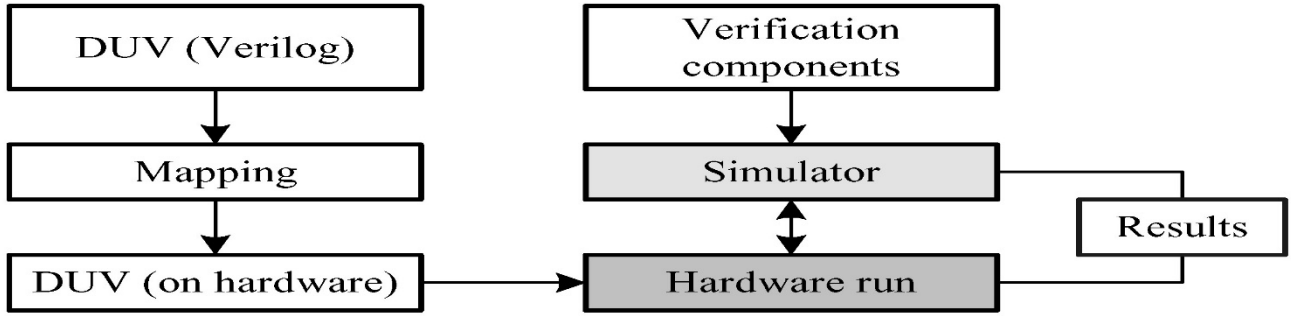
- Behavioral simulation
- Functional simulation
- Gate-level (logic) simulation
- Switch-level simulation
- Circuit-level (transistor-level) simulation

VARIATIONS OF SIMULATIONS

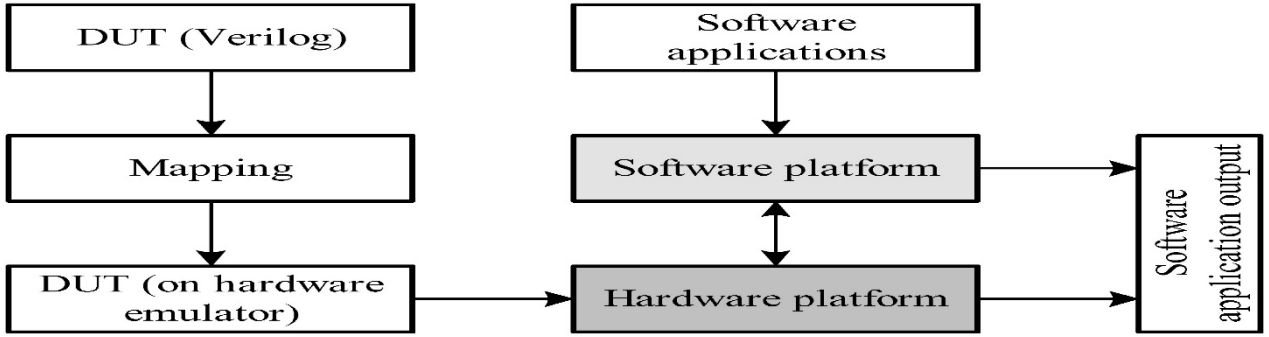
- Software simulation



- Hardware acceleration



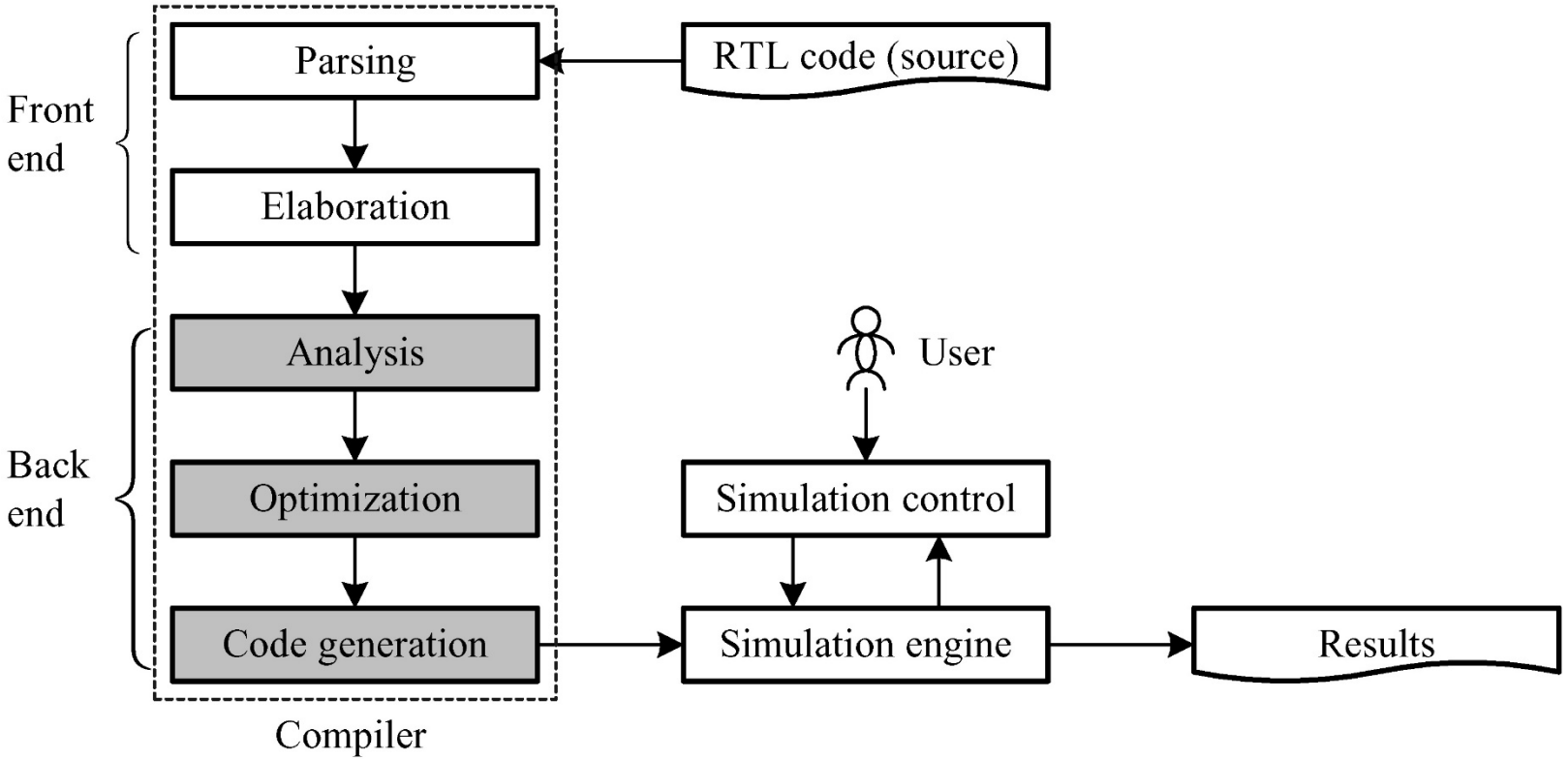
- Hardware emulation



SYLLABUS

- Objectives
- Verification
- Simulation
  - Types of simulations and simulators
  - Architecture of HDL simulators
- Test bench design
- Dynamic timing analysis
- Static timing analysis

AN ARCHITECTURE OF HDL SIMULATORS



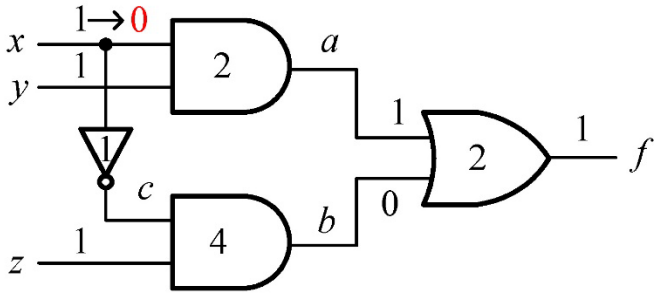
VERILOG HDL SIMULATORS

- Interpreted simulators
  - Cadence Verilog-XL simulator
- Compiled code simulators
  - Synopsys VCS simulator
- Native code simulators
  - Cadence Verilog-NC simulator

EVENT-DRIVEN/CYCLE-BASED SIMULATORS

- Event-driven simulators
  - Triggered by events
- Cycle-based simulators
  - On a cycle-by-cycle basis

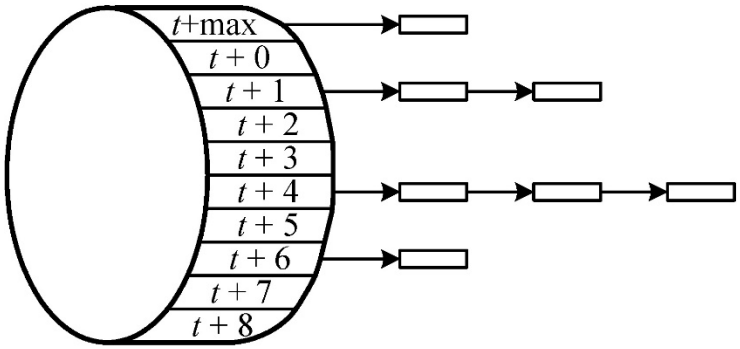
AN EVENT-DRIVEN SIMULATION



(a) A circuit example

	Scheduled events	Activity list
$t = 0$	$x = 0$	a, c
1	$c = 1$	b
2	$a = 0$	f
3		
4	$f = 0$	
5	$b = 1$	f
6		
7	$f = 1$	

(c) Scheduled events and the activity list



(b) Timing wheel

TEST BENCH DESIGN

TEST BENCH DESIGN PRINCIPLES

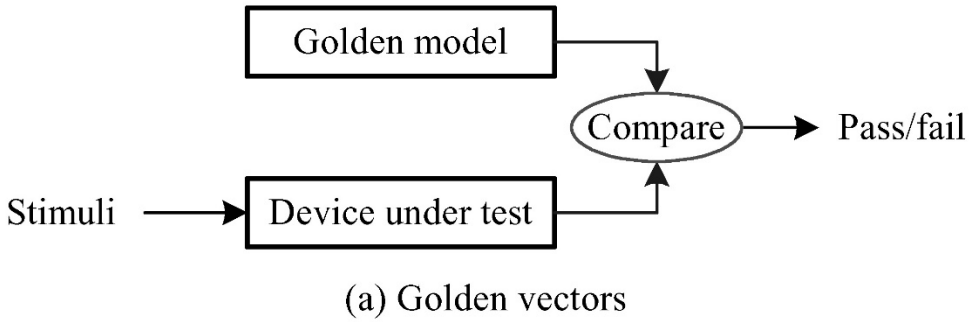
- Functions of a test bench
  - generates stimuli
  - checks responses in terms of test cases
  - employs reusable verification components
- Two types of test benches
  - deterministic
  - self-checking
- Options of choosing test vectors
  - Exhaustive test
  - Random test
  - Verification vector files

TEST BENCH DESIGN PRINCIPLES

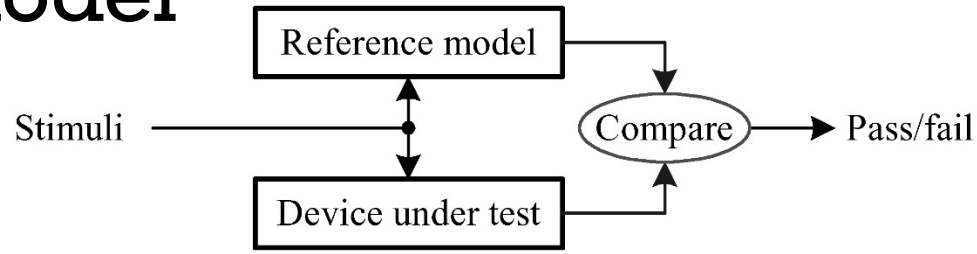
- Two basic choices of stimulus generation
  - Deterministic versus random stimulus generation
  - Pregenerated test case versus on-the-fly test case generation
- Types of result checking
  - on-the-fly checking
  - end-of-test checking
- Result analysis
  - Waveform viewers
  - Log files

TYPES OF AUTOMATED RESPONSE CHECKING

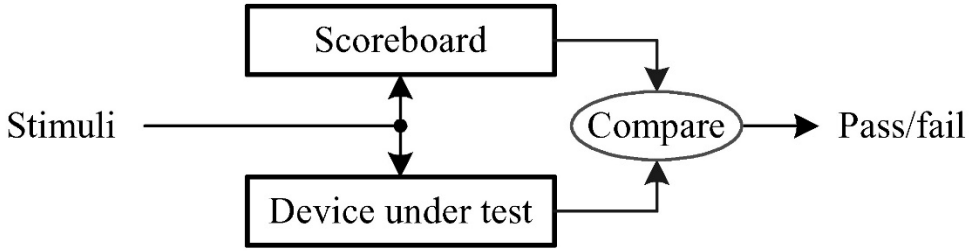
- Golden vectors
- Reference model
- Transaction-based model



(a) Golden vectors



(b) Reference model



(c) Transaction-based model

TEST BENCH DESIGNS --- A TRIVIAL EXAMPLE

```
// test bench design example 1: exhaustive test.
`timescale 1 ns / 100 ps
…
   nbit_adder_for UUT
          (.x(x), .y(y), .c_in(c_in), .sum(sum), .c_out(c_out));
reg [2*n-1:0] i;
initial  for (i = 0; i <= 2**(2*n)-1; i = i + 1) begin
              x[n-1:0] = i[2*n-1:n]; y[n-1:0] = i[n-1:0]; c_in =1'b0;
#20;
end
…
```

TEST BENCH DESIGNS --- A TRIVIAL EXAMPLE

```
// test bench design example 2: Random test.
`timescale 1 ns / 100 ps
…
nbit_adder_for UUT
        (.x(x), .y(y), .c_in(c_in), .sum(sum), .c_out(c_out));
integer i;
reg [n:0] test_sum;
initial  for (i = 0; i <= 2*n ; i = i + 1)  begin
          x = $random % 2**n;   y = $random % 2**n;
          c_in =1'b0;                  test_sum = x + y;
#15;      if (test_sum != {c_out, sum})
                        $display("Error iteration %h\n", i);
#5;       end
…
```

TEST BENCH DESIGNS --- A TRIVIAL EXAMPLE

```
// test bench design example 3: Using Verification vector files.
`timescale 1 ns / 100 ps
parameter n = 4;
parameter m = 8;
…
// Unit Under Test port map
    nbit_adder_for UUT
                (.x(x), .y(y), .c_in(c_in), .sum(sum), .c_out(c_out));
integer i;
reg [n-1:0] x_array [m-1:0];
reg [n-1:0] y_array [m-1:0];
reg [n:0] expected_sum_array [m-1:0];
```

TEST BENCH DESIGNS --- A TRIVIAL EXAMPLE

```
initial begin // reading verification vector files
    $readmemh("inputx.txt", x_array);
    $readmemh("inputy.txt", y_array);
    $readmemh("sum.txt", expected_sum_array);
end
initial
    for  (i = 0; i <= m - 1 ; i = i + 1) begin
            x = x_array[i];   y = y_array[i];
            c_in =1'b0;
#15;  if  (expected_sum_array[i] !=  {c_out, sum})
                        $display("Error iteration %h\n", i);
#5;    end
initial  #200 $finish;
```

inputx.txt	inputy.txt	sum.txt
4	1	05
9	3	0c
d	d	1a
5	2	07
1	d	0e
6	d	13
d	c	19
9	6	0f

TYPES OF CLOCK SIGNALS

- Types of clock signals
  - A general clock signal
  - Aligned derived clock signals
  - Clock multipliers
  - Asynchronous clock signals

A GENERAL CLOCK SIGNAL

- Examples

```
initial clk <=1'b0;
always #10  clk <= ~clk;
```

```
initial begin
    clk <= 1'b0;
    forever  #10 clk <= ~ clk;
end
```

```
reg  clk;
always begin
     #5 clk <= 1'b0;
     #5 clk <= 1'b1; end
```

A GENERAL CLOCK SIGNAL

- Truncation error

```
`timescale 1 ns / 1 ns
reg  clk;
parameter clk_period = 25;
```

```
always begin
    #(clk_period/2) clk <= 1'b0;
    #(clk_period/2) clk <= 1'b1;
end
```

- Rounding error

```
`timescale 1 ns / 1 ns
reg  clk;
parameter clk_period = 25;
```

```
always begin
    #(clk_period/2.0) clk <= 1'b0;
    #(clk_period/2.0) clk <= 1'b1;
end
```

- Proper precision

```
`timescale 1 ns / 100 ps
reg  clk;
parameter clk_period = 25;
```

```
always begin
    #(clk_period/2) clk <= 1'b0;
    #(clk_period/2) clk <= 1'b1;
end
```

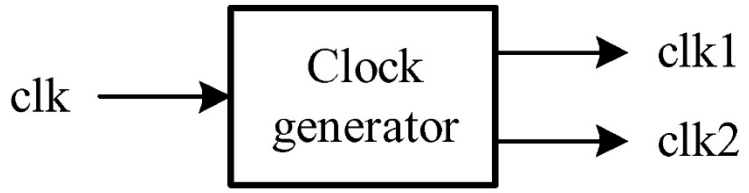
ALIGNED DERIVED CLOCK SIGNALS

- An improper approach

```
always begin
    if (clk == 1'b1) clk2 <= ~ clk2;
end
```

- A proper approach

```
// both clk1 and clk2 are derived from clk.
always begin
    clk1 <= clk;
    if (clk == 1'b1) clk2 <= ~ clk2;
end
```



CLOCK MULTIPLIERS

- An example

```
initial begin
    clk1 <= 1'b0;
    clk4 <= 1'b0;
    forever begin
        repeat (4) begin
            #10 clk4 <= ~ clk4; end
        clk1 <= ~ clk1;
    end
end
```

ASYNCHRONOUS CLOCK SIGNALS

- "Asynchronous" means random

```
initial begin
    clk100 <= 1'b0;
    #2;
    forever begin
        #5 clk100 <= ~ clk100;
    end
end
```

```
initial begin
    clk33 <= 1'b0;
    #5;
    forever begin
        #15 clk33 <= ~ clk33;
    end
end
```

RESET SIGNAL GENERATIONS

- Race condition
  - Using nonblocking assignments

```
always begin
      #5 clk = 1'b0;
      #5 clk = 1'b1;
end
initial begin // has race condition.
     reset = 1'b0;
     #20 reset = 1'b1;
     #40 reset = 1'b0;
end
```

```
always begin
      #5 clk <= 1'b0;
      #5 clk <= 1'b1;
end
initial begin // no race condition.
     reset <= 1'b0;
     #20 reset <= 1'b1;
     #40 reset <= 1'b0;
end
```

RESET SIGNAL GENERATIONS

- Increase of maintainability

```
always begin
     #(clk_period/2) clk <= 1'b0;
     #(clk_period/2) clk <= 1'b1;
end
initial begin
    reset = 1'b0;
    wait (clk !== 1'bx);
    repeat (3) @(negedge clk) reset <= 1'b1;
    reset <= 1'b0;
end
```

RESET SIGNAL GENERATIONS

- The use of a task

```
always begin
      #(clk_period/2) clk <= 1'b0;
      #(clk_period/2) clk <= 1'b1;
end
// using a task
task  hardware_reset;
begin
     reset = 1'b0;
     wait (clk !== 1'bx);
    // set reset to 1 for two clock cycles
     repeat (3) @(negedge clk) reset <= 1'b1;
     reset <= 1'b0;
end endtask
```

COVERAGE ANALYSIS

- Two major types
  - Structural coverage
  - Functional coverage
- Q: What does 100% functional coverage mean?
  - You have covered all the coverage points you included in the simulation
  - By no means the job is done

STRUCTURAL COVERAGE

- Statement coverage
- Branch or conditional coverage
- Toggle coverage
- Trigger coverage
- Expression coverage
- Path coverage
- Finite-state machine coverage

FUNCTIONAL COVERAGE

- Functional coverage
  - Item coverage
  - Cross coverage
  - Transition coverage

Module	Stmt count	Stmt hits	Stmt miss	Stmt %
Three-step Booth	35	35	0	100%
Controller	21	21	0	100%
Datapath	5	5	0	100%

DYNAMIC TIMING ANALYSIS

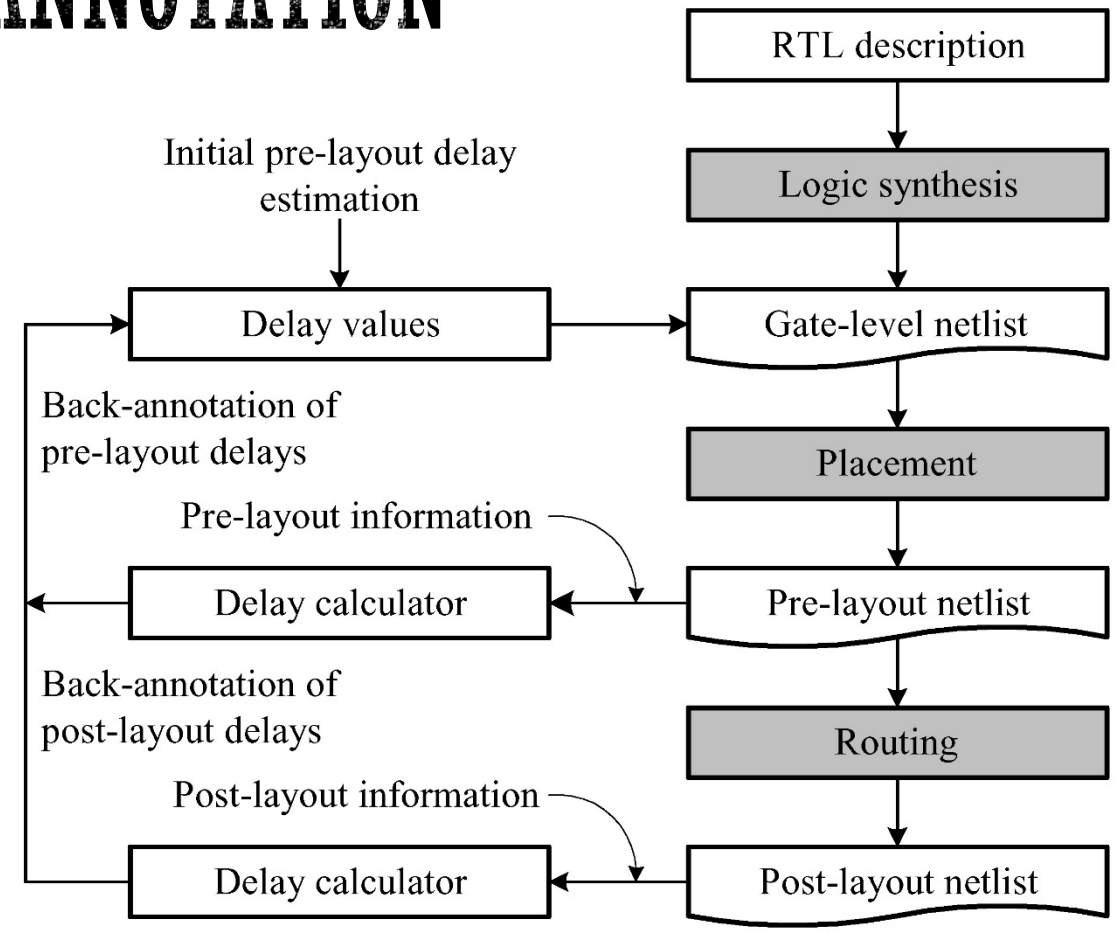
SDF AND GENERATION GUIDELINES

- The standard delay format (SDF) file
- The SDF specifies
  - IOPATH delay
  - INTERCONNECT delay
  - Timing check (SETUP, HOLD, etc)

GENERATION OF SDF FILES

- Pre-layout
  - Gate delay information
  - *_map.sdf (contains gate delay only) in ISE (Integrated Synthesis Environment) design flow
- Post-layout
  - Both gate and interconnect delay information
  - *_timesim.sdf in ISE design flow

DELAY BACK-ANNOTATION

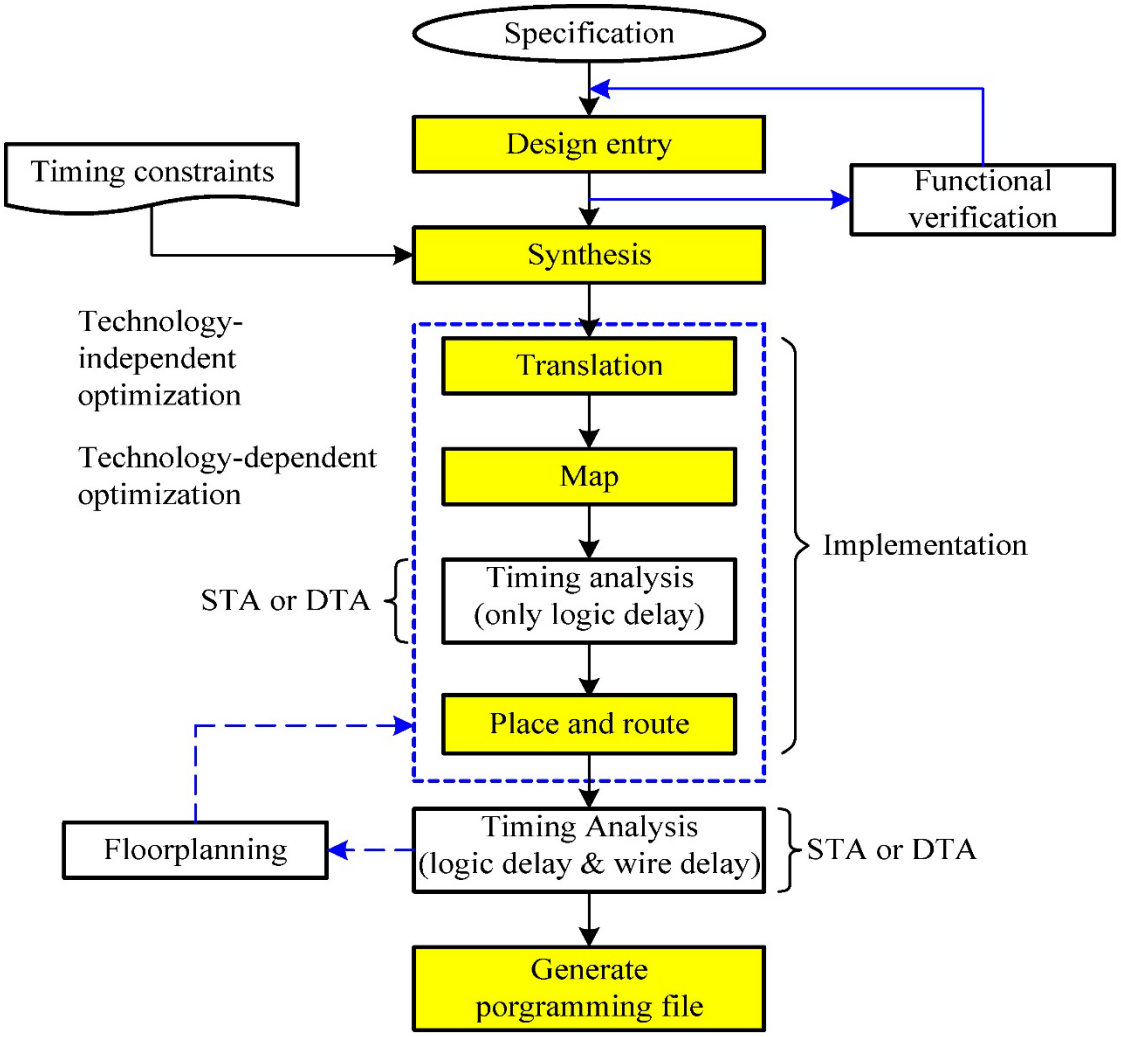



```
$sdf_annotate (design_file_name _map.sdf", design_file_name);
 $sdf_annotate (design_file_name _timesim.sdf", design_file_name);
```

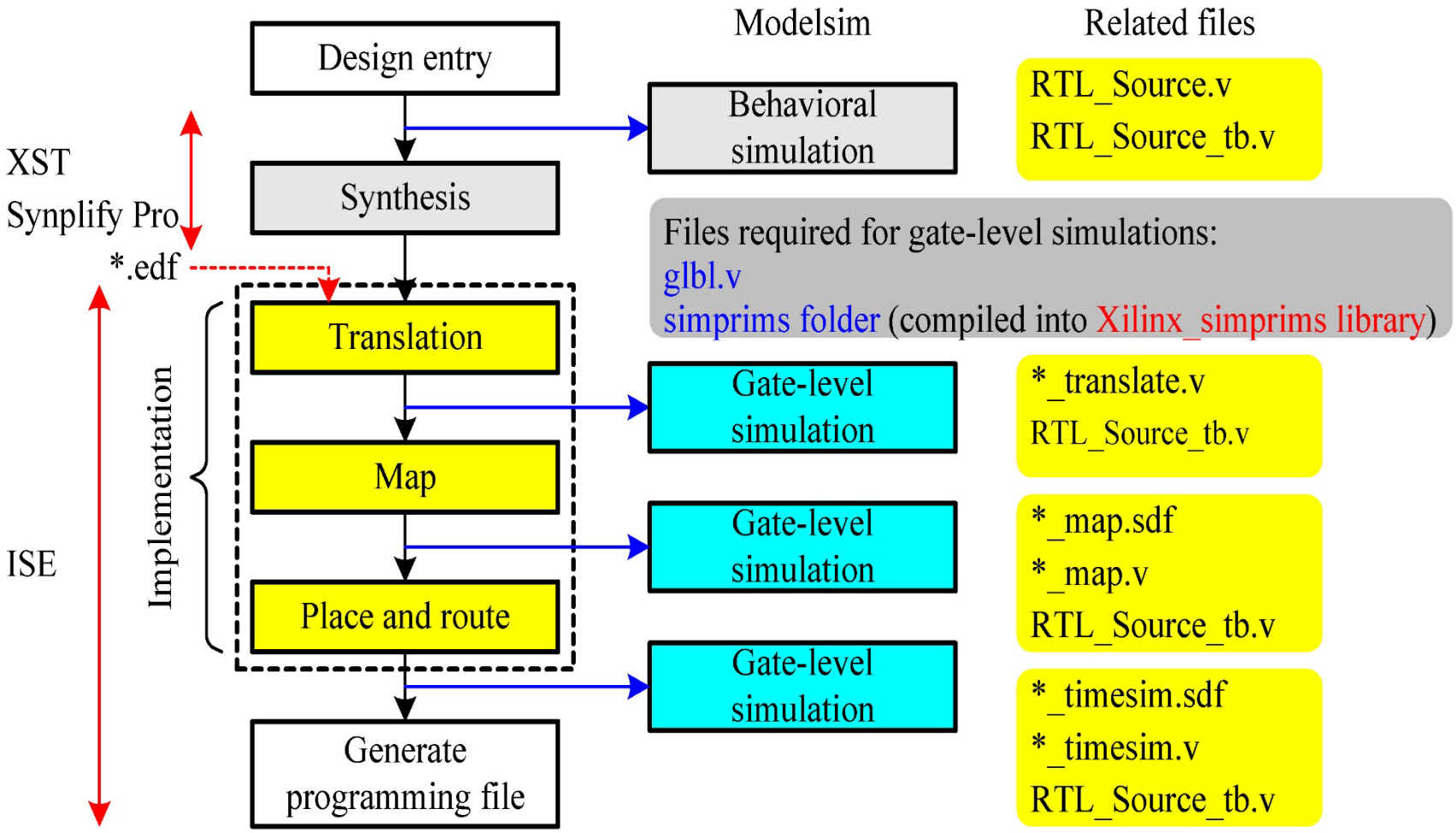
THE ISE DESIGN FLOW

- Design entry
- Synthesis to create a gate netlist
- Implementation
  - Translation
  - Map
  - Place and route
- Configure FPGA

THE ISE DESIGN FLOW



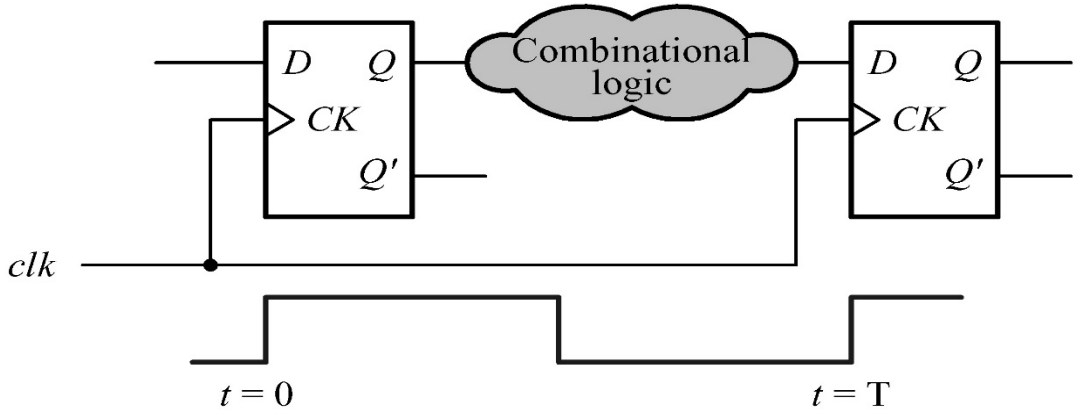
A SIMULATION FLOW --- AN ISE-BASED FLOW



STATIC TIMING ANALYSIS

TIMING ANALYSIS

- Q: The output needs to be stable by $t = T$ for the correct functionality. But how to make sure of it?
- Two approaches
  - Dynamic timing simulation
  - Static timing analysis



PURPOSES OF TIMING ANALYSIS

- Timing verification
  - if a design meets a given timing constraint?
  - Example: cycle-time constraint
- Timing optimization
  - Optimizes the critical portion of a design
  - Identifies critical paths

WHY STATIC TIMING ANALYSIS?

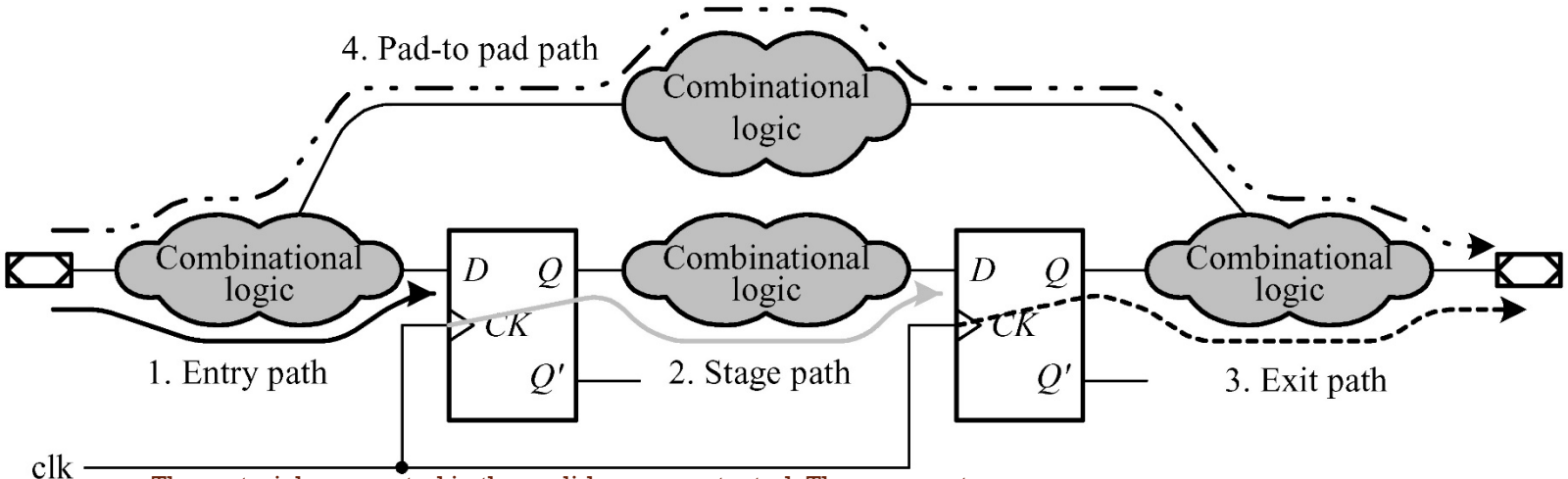
- Drawbacks of DTS
  - has posed a bottleneck for large complex designs
  - relies on the quality and coverage of the test bench
- Basic assumptions of STA
  - No combinational feedback loops
  - All register feedback broken by the clock boundary

STATIC TIMING ANALYSIS

- In STA
  - Designs are broken into sets of signal paths
  - Each path has a start point and an endpoint
- Start points
  - Input ports
  - Clock pins of storage elements
- Endpoints
  - Output ports
  - Data input pins of storage elements

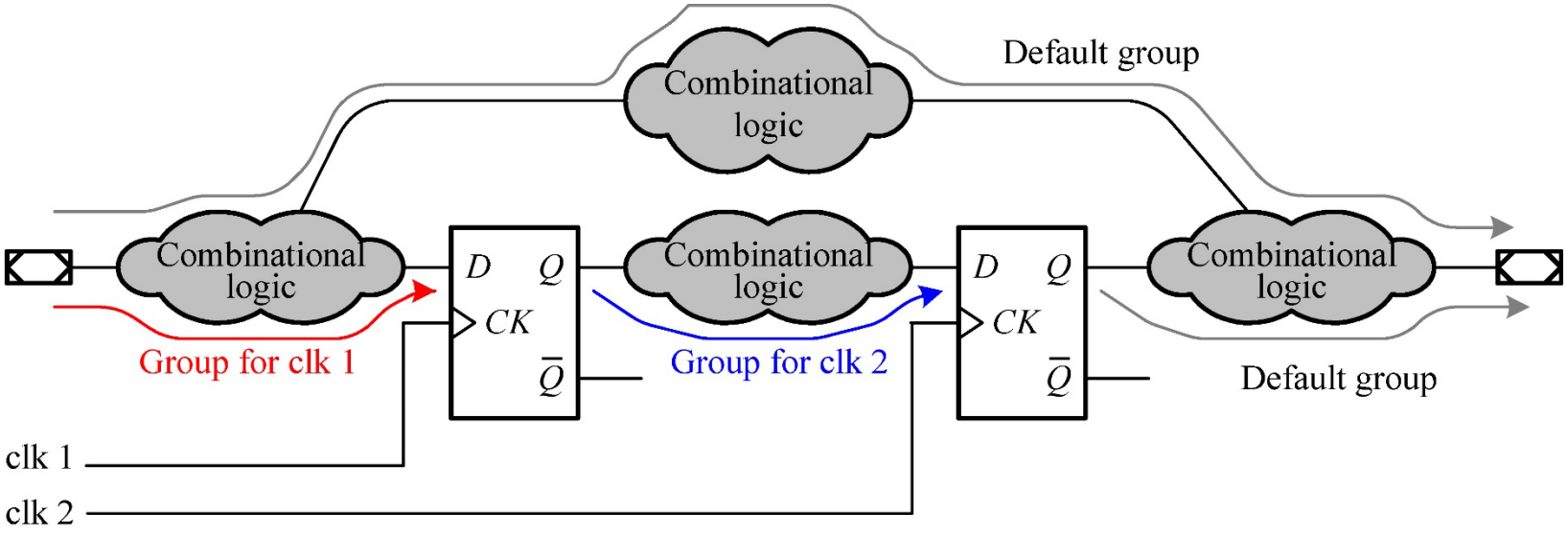
FOUR TYPES OF PATH ANALYSIS

- Entry path (input-to-D path)
- Stage path (register-to-register path or clock-to-D path)
- Exit path (clock-to-output path)
- Pad-to-pad path (port-to-port path)



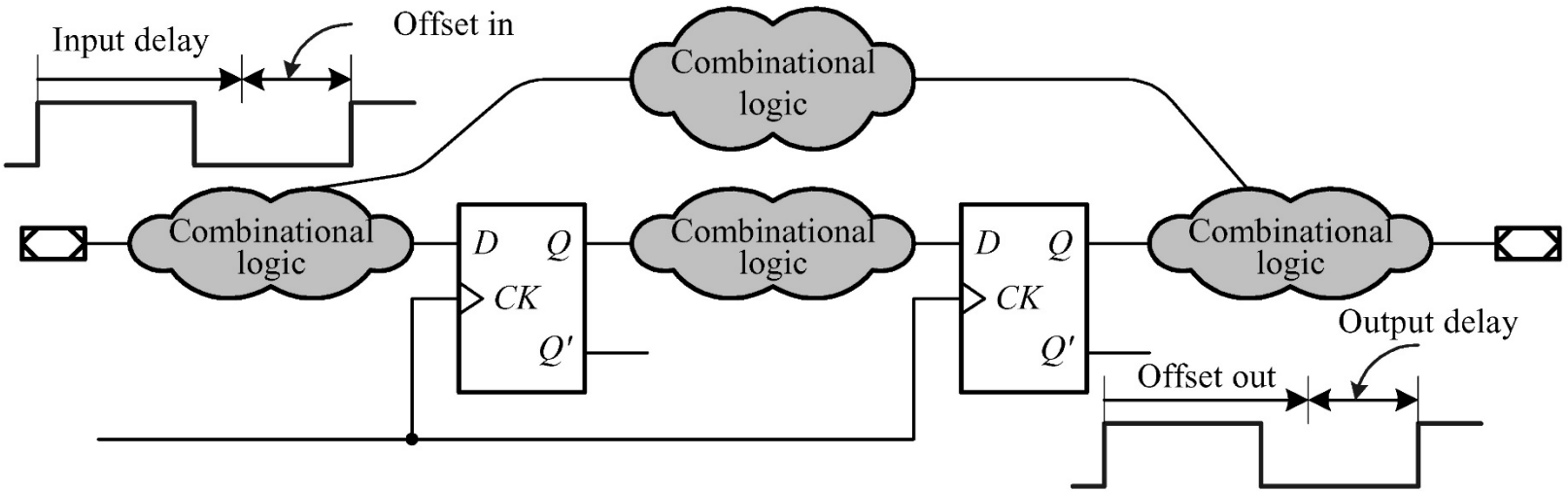
PATH GROUPS

- Types of path groups
  - Path group
  - Default path group



TIMING SPECIFICATIONS

- Port-related constraints
  - Input delay (offset-in)
  - Output delay (offset-out)
  - Input-output (pad to pad)
  - Cycle time (period)



SETUP TIME AND HOLD TIME CHECKS

- Clock-related constraints
  - Clock period
  - Setup time
  - Hold time

TIMING ANALYSIS

- A critical path
  - The path of longest propagation delay
  - A combinational logic path that has negative or smallest slack time

    slack = required time – arrival time

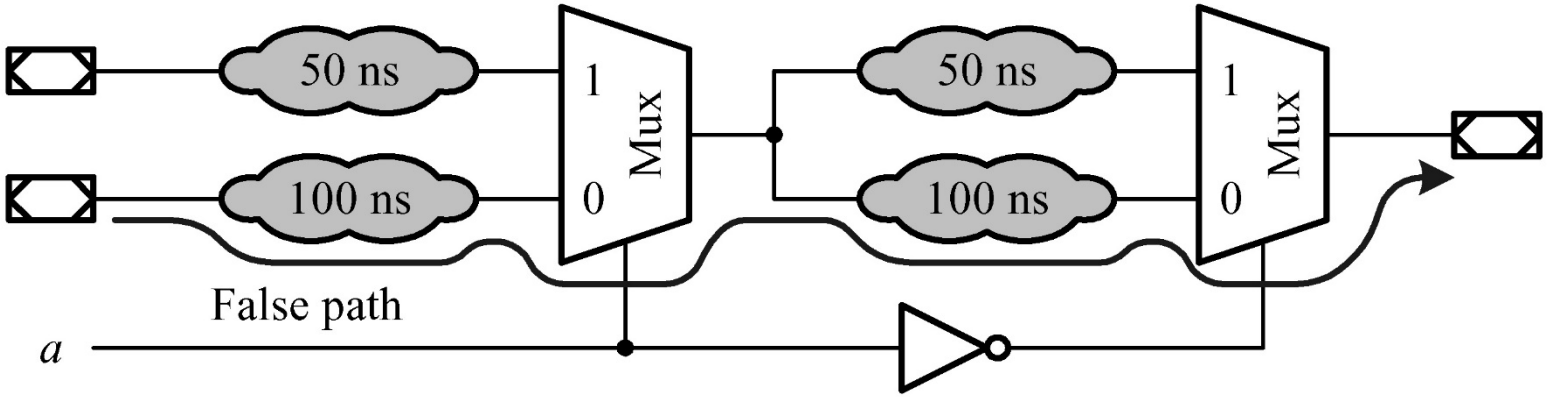
    = requirement – datapath (in ISE)

TIMING EXCEPTIONS

- Two timing exceptions
  - False paths
  - Multi-cycle paths

FALSE PATHS

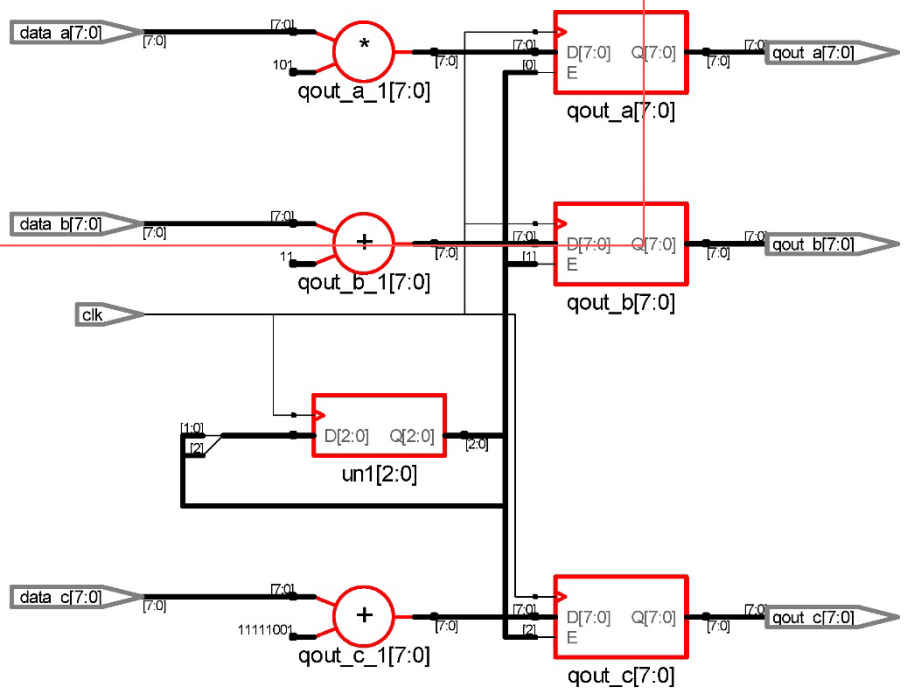
- A false path
  - A timing path does not propagate a signal
  - STA identifies as a failing timing path



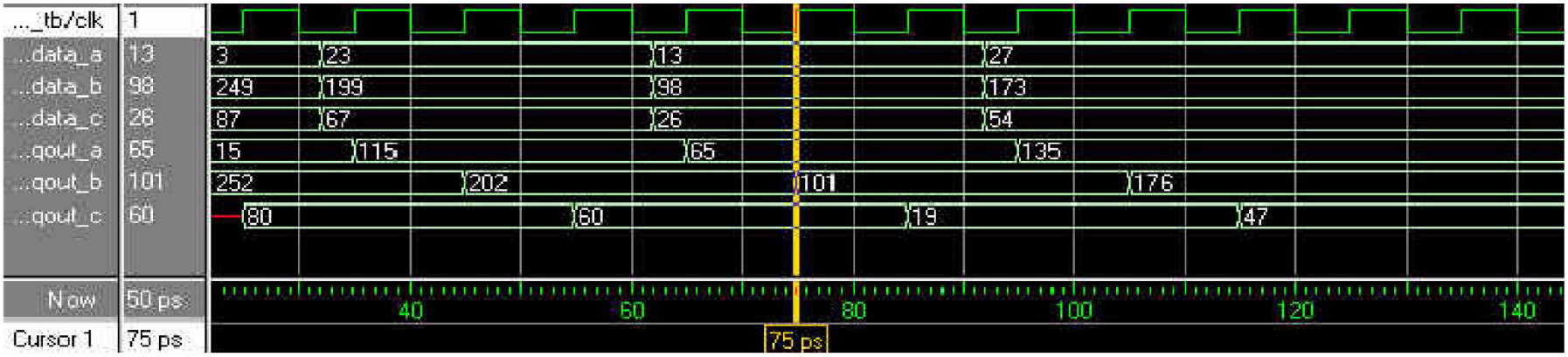
MULTI-CYCLE PATHS --- A TRIVIAL EXAMPLE

```
// a multiple cycle example
module multiple_cycle_example(clk, data_a, data_b, …);
…
// trivial multiple-cycle operations
always @(posedge clk) begin
    qout_a <= data_a * 5;
    @(posedge clk)  qout_b <= data_b + 3;
    @(posedge clk)  qout_c <= data_c - 7;
end
```

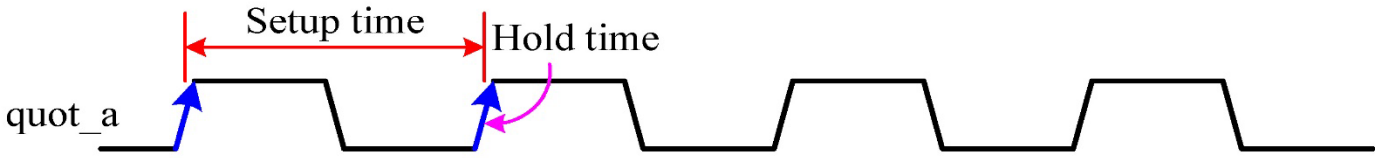
Q: Explain the operation of the above code



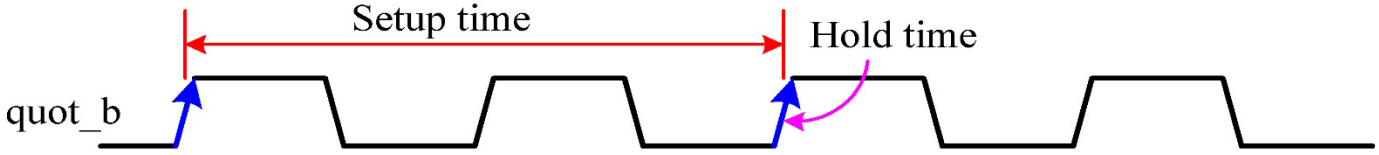
MULTI-CYCLE PATHS --- A TRIVIAL EXAMPLE



(a) Single-cycle timing relationship



(b) Two-cycle timing relationship



(c) Three-cycle timing relationship