Chapter 1: Introduction

Prof. Soo-Ik Chae

Objectives

After completing this chapter, you will be able to:

- Understand the features of HDLs and Verilog HDL
- Describe the HDL-based design flow
- Describe the basic features of the modules in Verilog HDL
- Describe how to model a design in structural style
- Describe how to model a design in dataflow style
- Describe how to model a design in behavioral style
- Describe how to model a design in mixed style
- Describe how to simulate/verify a design using Verilog HDL

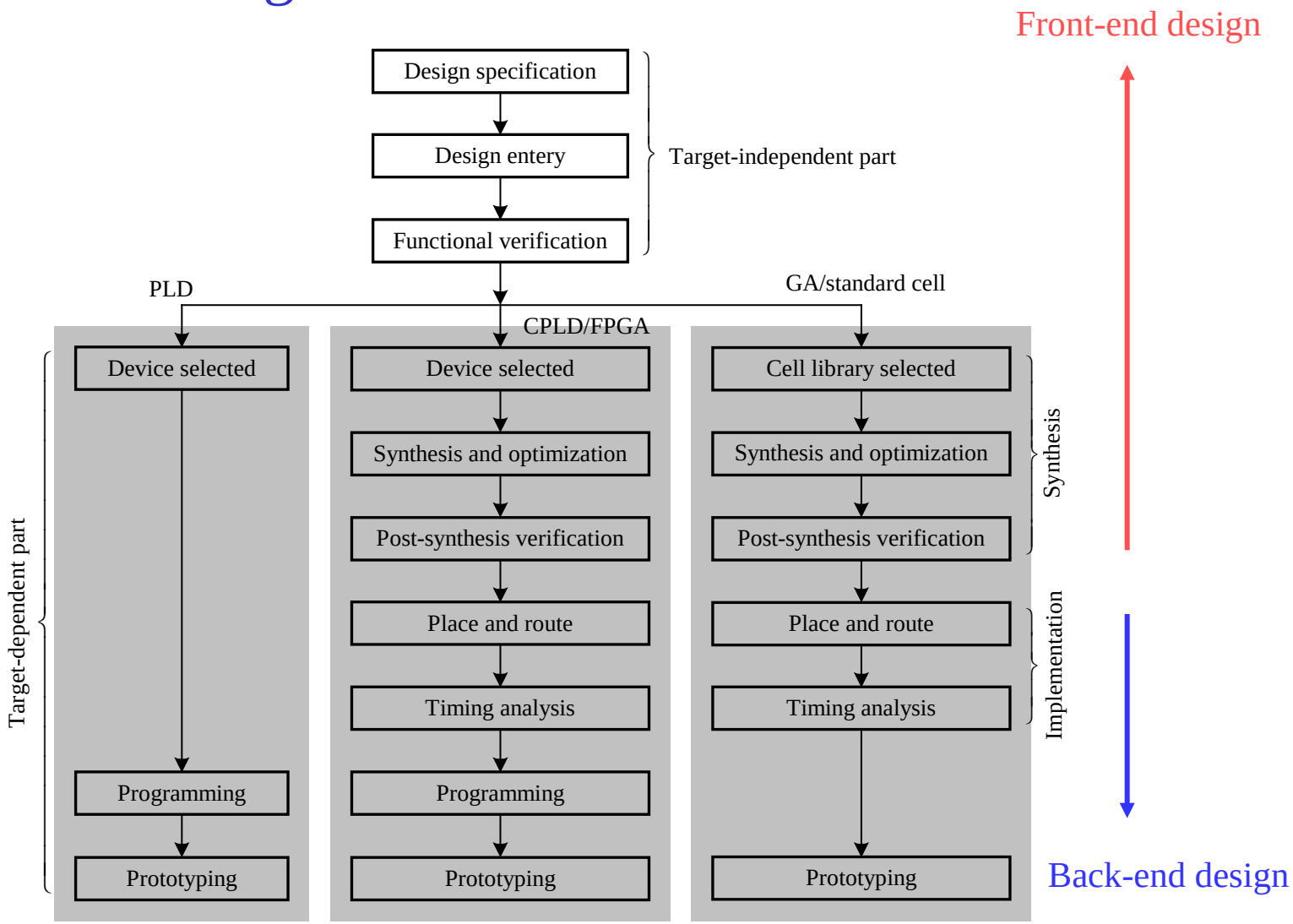
Importance of HDLs

- HDL is an acronym of Hardware Description Language.
- Two most commonly used HDLs:
  - Verilog HDL (also called Verilog for short)
  - VHDL (Very high-speed integrated circuits HDL)
- Features of HDLs:
  - Design can be described at a very abstract level.
  - Functional verification can be done early in the design cycle.
  - Designing with HDLs is analogous to computer programming.

Popularity of Verilog HDL

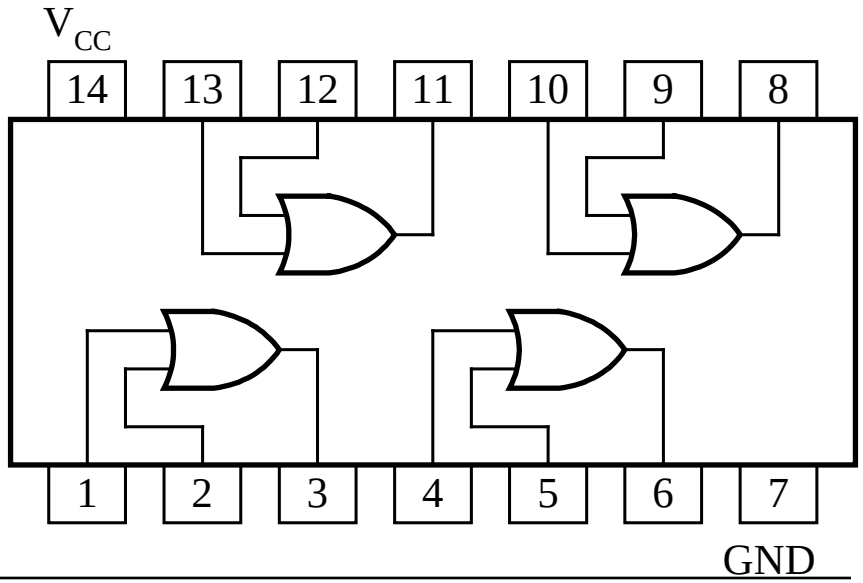
- It is a general-purpose, easy to learn, and easy to use HDL language.
- It allows different levels of abstraction to be mixed in the same model.
- It is supported by all logic synthesis tools.
- It provides a powerful Programming Language Interface (PLI).
  - Allow us to develop our own CAD tools such as delay calculator.

HDL-Based Design Flow



Modules – Hardware Module Concept

- The basic unit of a digital system is a module.
- Each module consists of:
  - a core circuit (called internal or body) --- performs the required function
  - an interface (called ports) --- carries out the required communication between the core circuit and outside.



Modules – Verilog HDL modules

- module --- The basic building block in Verilog HDL.
  - It can be an element or a collection of lower-level design blocks.

module Module name

Port List, Port Declarations (if any)

Parameters (if any)

Declarations of *wires*, *regs*, and other variables

Instantiation of lower level modules or primitives

Data flow statements (*assign*)

always and *initial* blocks. (all behavioral statements go into these blocks).

Tasks and functions.

endmodule statement

Lexical Conventions

- Verilog HDL uses almost the same lexical conventions as C language.
  - Identifiers consists of alphanumeric characters, _, and $.
    - Verilog is a case-sensitive language just like C.
  - White space: blank space (\b), tabs (\t), and new line (\n).
  - Comments:
    - // indicates that the remaining of the line is a comment.
    - /* ….*/ indicates what in between them are comments.
  - Sized number: <size>`<base format><number>
    - 4`b1001 --- a 4-bit binary number
    - 16`habcd --- a 16-bit hexadecimal number

Lexical Conventions

- Unsized number: `<base format><number>
  - 2007 --- a 32-bit decimal number by default
  - `habc --- a 32-bit hexadecimal number
- x or z values: x denotes an unknown value; z denotes a high impedance value.
- Negative number: -<size>`<base format><number>
  - -4`b1001 --- a 4-bit binary number
  - -16`habcd --- a 16-bit hexadecimal number
- "_" and "?"
  - 16`b0101_1001_1110_0000
  - 8`b01??_11?? --- equivalent to a 8`b01zz_11zz

Lexical Conventions

- String: "Back to School and Have a Nice Semester"

❖ Coding style:
- Use lowercase letters for all signal names, variable names, and port names.
- Use uppercase letters for names of constants and user-defined types.
- Use meaningful names for signals, ports, functions, and parameters.

The Value Set

- In Verilog HDL
  - 0 and 1 represent logic values low and high, respectively.
  - z indicates the high-impedance condition of a node or net.
  - x indicates an unknown value of a net or node.

Value	Meaning
0	Logic 0, false condition
1	Logic 1, true condition
x	Unknown logic value
z	High impedance

Data Types

- Verilog HDL has two classes of data types.
  - Nets mean any hardware connection points.
  - Variables represent any data storage elements.

Nets		Variables
wire	supply0	reg
tri	supply1	integer
wand	tri0	real
wor	tri1	time
triand	trireg	realtime
trior		

Data Types

- A net variable
  - can be referenced anywhere in a module.
  - must be driven by a primitive, continuous assignment, force … release, or module port.
- A variable
  - can be referenced anywhere in a module.
  - can be assigned value only within a procedural statement, task, or function.
  - cannot be an input or inout port in a module.

Module Modeling Styles

- Structural style
  - Gate level comprises a set of interconnected gate primitives.
  - Switch level consists of a set of interconnected switch primitives.
- Dataflow style
  - specifies the dataflow (i.e., data dependence) between registers.
  - is specified as a set of continuous assignment statements.

Module Modeling Styles

- Behavioral or algorithmic style
  - is described in terms of the desired design algorithm
  - is without concerning the hardware implementation details.
  - can be described in any high-level programming language.
- Mixed style
  - is the mixing use of above three modeling styles.
  - is commonly used in modeling large designs.
- In industry, RTL (register-transfer level) means
  - RTL = synthesizable behavioral + dataflow constructs

Port Declaration

- Port Declaration
  - input: input ports.
  - output: output ports.
  - inout: bidirectional ports
- Port Connection Rules
  - Named association
  - Positional association

Port Declaration

```
module half_adder (x, y, s, c);
input  x, y;
output s, c;
// -- half adder body-- //
// instantiate primitive gates
   xor xor1 (s, x, y);
   and and1 (c, x, y);
endmodule

module full_adder (x, y, cin, s, cout);
input  x, y, cin;
output s, cout;
wire   s1,c1,c2;  // outputs of both half adders
// -- full adder body-- //
// instantiate the half adder
   half_adder ha_1 (x, y, s1, c1);
   half_adder ha_2 (.x(cin), .y(s1), .s(s), .c(c2));
   or (cout, c1, c2);
endmodule
```

Can only be connected by using positional association

Instance name is optional.

Connecting by using positional association

Connecting by using named association

Instance name is necessary.

Structural modeling

```
// gate-level hierarchical description of 4-bit adder
// gate-level description of half adder
module half_adder (x, y, s, c);
input x, y;
output s, c;
// half adder body
// instantiate primitive gates
xor (s,x,y);
and (c,x,y);
endmodule
```

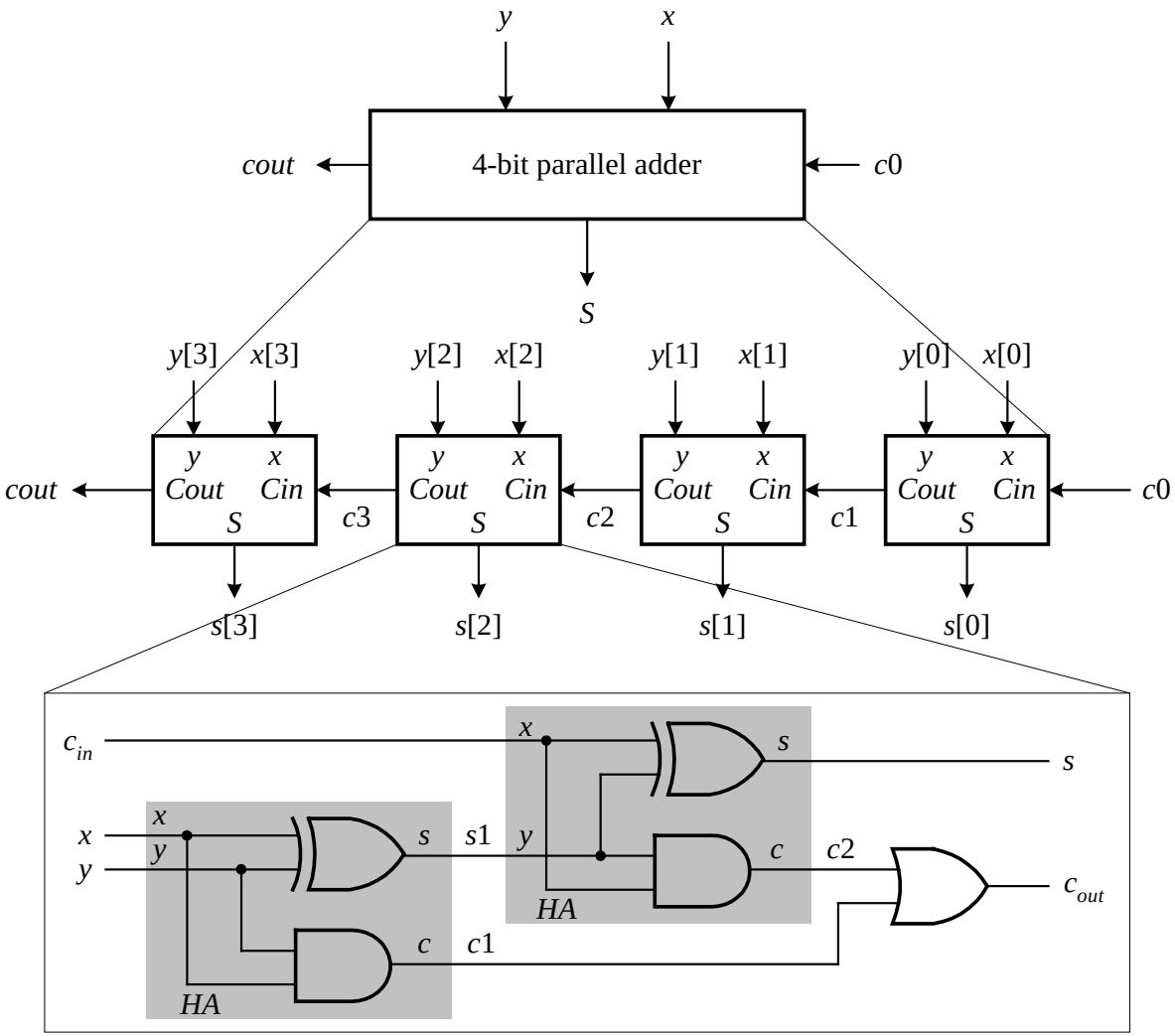
Structural modeling

```
// gate-level description of full adder
module full_adder (x, y, cin, s, cout);
input x, y, cin;
output s, cout;
wire s1, c1, c2; // outputs of both half adders
// full adder body
// instantiate the half adder
half_adder ha_1 (x, y, s1, c1);
half_adder ha_2 (cin, s1, s, c2);
or (cout, c1, c2);
endmodule
```

Structural modeling

```
// gate-level description of 4-bit adder
module four_bit_adder (x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output [3:0] sum;
output c_out;
wire c1, c2, c3; // intermediate carries
// four_bit adder body
// instantiate the full adder
full_adder fa_1 (x[0], y[0], c_in, sum[0], c1);
full_adder fa_2 (x[1], y[1], c1, sum[1], c2);
full_adder fa_3 (x[2], y[2], c2, sum[2], c3);
full_adder fa_4 (x[3], y[3], c3, sum[3], c_out);
endmodule
```

Hierarchical Design



Dataflow Modeling

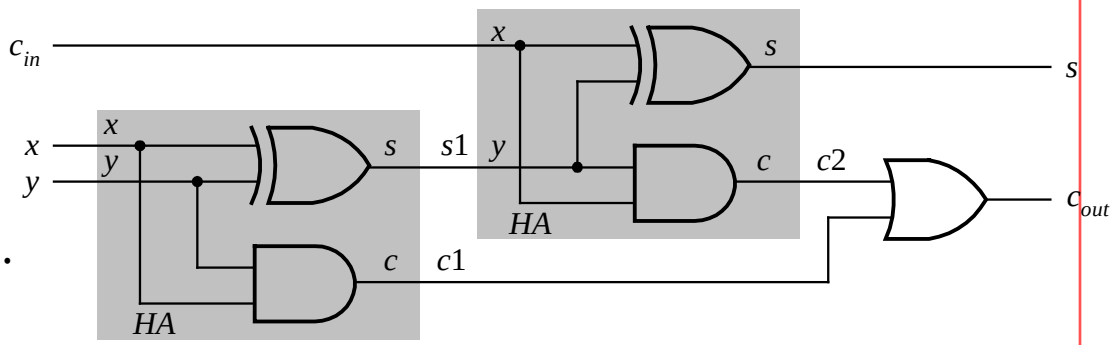
```
module full_adder_dataflow(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
// specify the function of a full adder
assign #5 {c_out, sum} = x + y + c_in;
endmodule
```

Behavioral Modeling

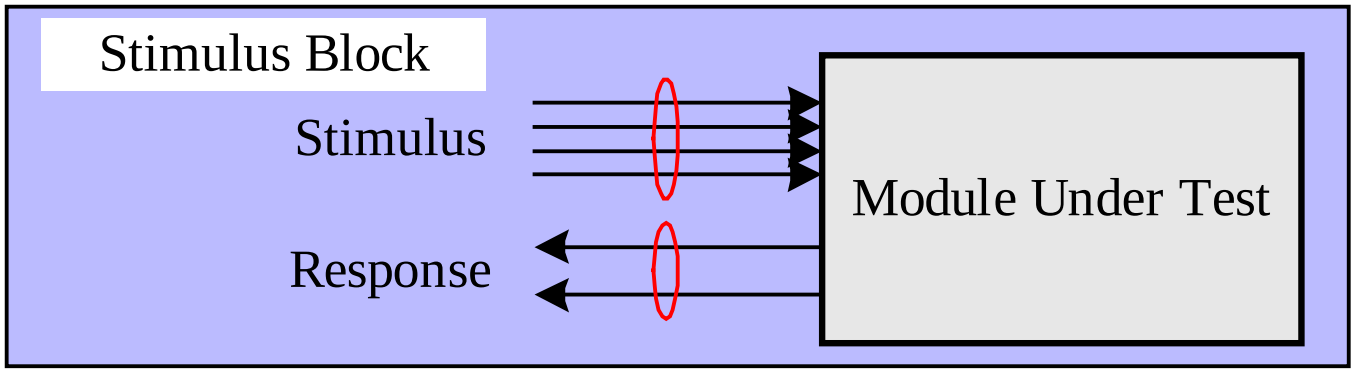
```
module full_adder_behavioral(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
reg sum, c_out; // sum and c_out need to be declared as reg types.
// specify the function of a full adder
always @(x, y, c_in)// can also use always @(*) or always@(x or y or c_in)
#5 {c_out, sum} = x + y + c_in;
endmodule
```

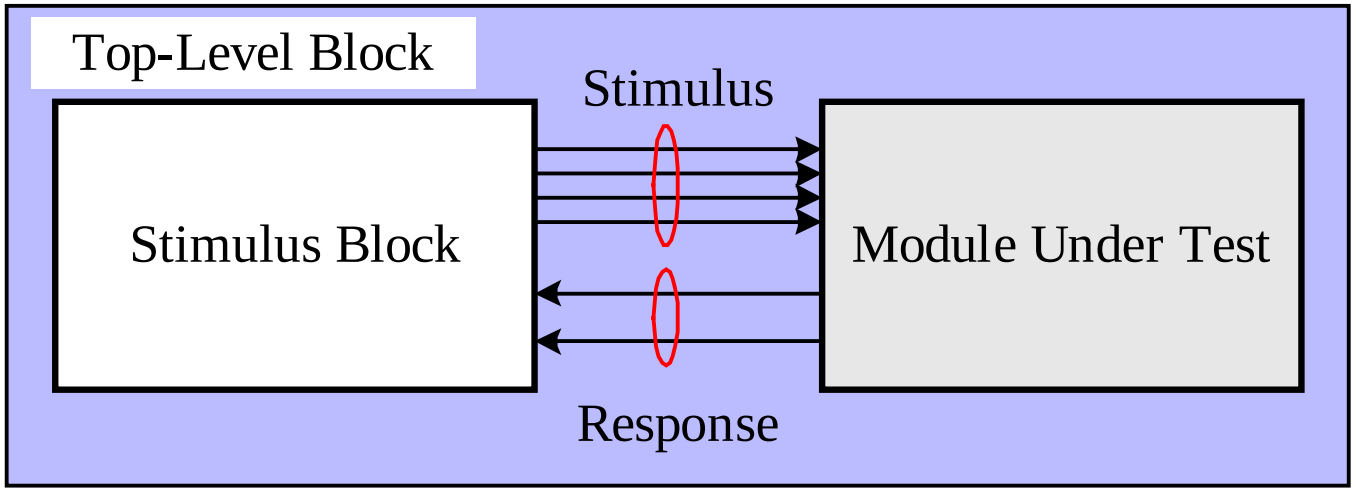
Mixed-Style Modeling

```
module full_adder_mixed_style(x, y, c_in, s, c_out);
// I/O port declarations
input x, y, c_in;
output s, c_out;
reg c_out;
wire s1, c1, c2;
// structural modeling of HA 1.
xor xor_ha1 (s1, x, y);
and and_ha1(c1, x, y);
// dataflow modeling of HA 2.
assign s = c_in ^ s1;
assign c2 = c_in & s1;
// behavioral modeling of output OR gate.
always @(c1, c2) // can also use always @(*)
c_out = c1 | c2;
endmodule
```



Basic Simulation Constructs





System Tasks for Simulation

- $display displays values of variables, string, or expressions
  - $display(*ep*1, *ep*2, …, *epn*);

    *ep*1, *ep*2, …, *epn*: quoted strings, variables, expressions.
- $monitor monitors a signal when its value changes.
  - $monitor(*ep*1, *ep*2, …, *epn*);
- $monitoton enables monitoring operation.
- $monitotoff disables monitoring operation.
- $stop suspends the simulation.
- $finish terminates the simulation.

Time Scale for Simulations

- Time scale compiler directive

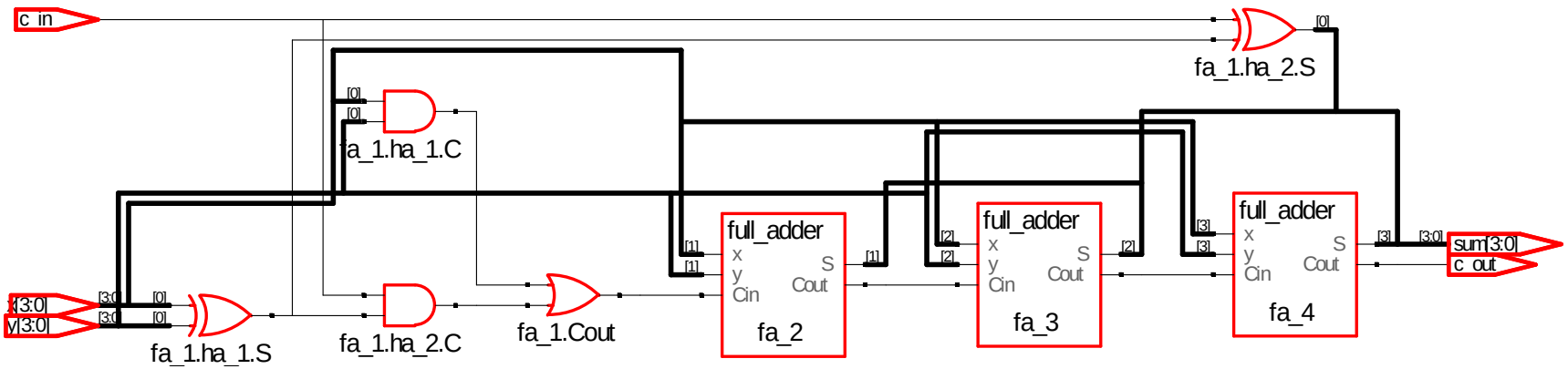
  `timescale time_unit / time_precision

  - The time_precision must not exceed the time_unit.
  - For instance, with a timescale 1 ns/1 ps, the delay specification #15 corresponds to 15 ns.
  - It uses the same time unit in both behavioral and gate-level modeling.
  - For FPGA designs, it is suggested to use ns as the time unit.

Modeling and Simulation Example --- A 4-bit adder

```
// Gate-level description of 4-bit adder
module four_bit_adder (x, y, c_in, sum, c_out);
input   [3:0] x, y;
input   c_in;
output [3:0] sum;
output c_out;
wire   C1,C2,C3;  // Intermediate carries
// -- four_bit adder body--
// Instantiate the full adder
   full_adder fa_1 (x[0],y[0],c_in,sum[0],C1);
   full_adder fa_2 (x[1],y[1],C1,sum[1],C2);
   full_adder fa_3 (x[2],y[2],C2,sum[2],C3);
   full_adder fa_4 (x[3],y[3],C3,sum[3],c_out);
endmodule
```

Modeling and Simulation Example --- A 4-bit adder



After dissolving one full adder.

Modeling and Simulation Example --- A Test Bench

```
`timescale 1 ns / 100 ps  // time unit is in ns.
module four_bit_adder_tb;
//Internal signals declarations:
reg [3:0] x;
reg [3:0] y;
reg c_in;
wire [3:0] sum;
wire c_out;
// Unit Under Test port map
   four_bit_adder UUT (.x(x), .y(y), .c_in(c_in), .sum(sum), .c_out(c_out));
reg [7:0] i;
initial begin // for use in post-map and post-par simulations.
// $sdf_annotate ("four_bit_adder_map.sdf", four_bit_adder);
// $sdf_annotate ("four_bit_adder_timesim.sdf", four_bit_adder);
end
```

Modeling and Simulation Example --- A Test Bench

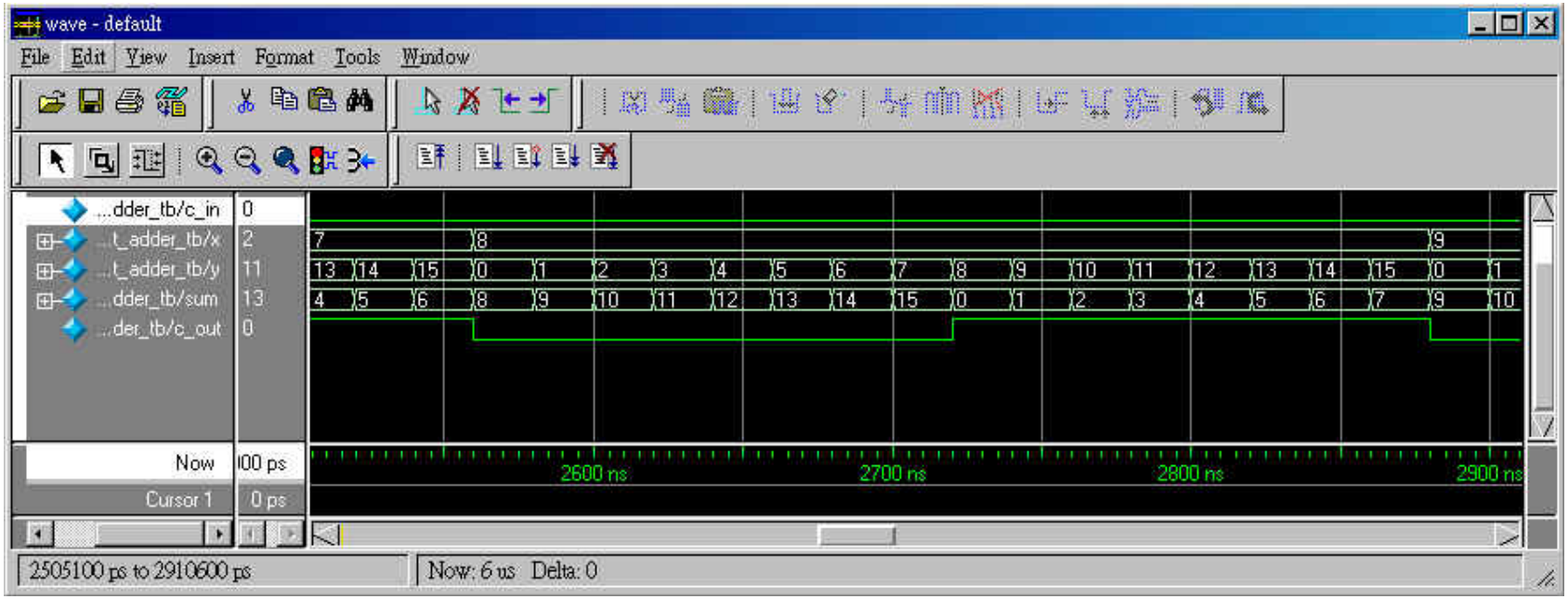
```
initial
   for (i = 0; i <= 255; i = i + 1) begin
       x[3:0] = i[7:4]; y[3:0] = i[3:0]; c_in =1'b0;
   #20 ;   end
initial #6000 $finish;
initial
   $monitor($realtime,“ns %h %h %h %h", x, y, c_in, {c_out, sum});
endmodule
```

Modeling and Simulation Example --- Simulation Results

```
     0ns 0 0 0 00
#   20ns 0 1 0 01
#   40ns 0 2 0 02
#   60ns 0 3 0 03
#   80ns 0 4 0 04
# 100ns 0 5 0 05
# 120ns 0 6 0 06
# 140ns 0 7 0 07
# 160ns 0 8 0 08
# 180ns 0 9 0 09
# 200ns 0 a 0 0a
# 220ns 0 b 0 0b
# 240ns 0 c 0 0c
# 260ns 0 d 0 0d
```

```
# 280ns 0 e 0 0e
# 300ns 0 f 0 0f
# 320ns 1 0 0 01
# 340ns 1 1 0 02
# 360ns 1 2 0 03
# 380ns 1 3 0 04
# 400ns 1 4 0 05
# 420ns 1 5 0 06
# 440ns 1 6 0 07
# 460ns 1 7 0 08
# 480ns 1 8 0 09
# 500ns 1 9 0 0a
# 520ns 1 a 0 0b
# 540ns 1 b 0 0c
```

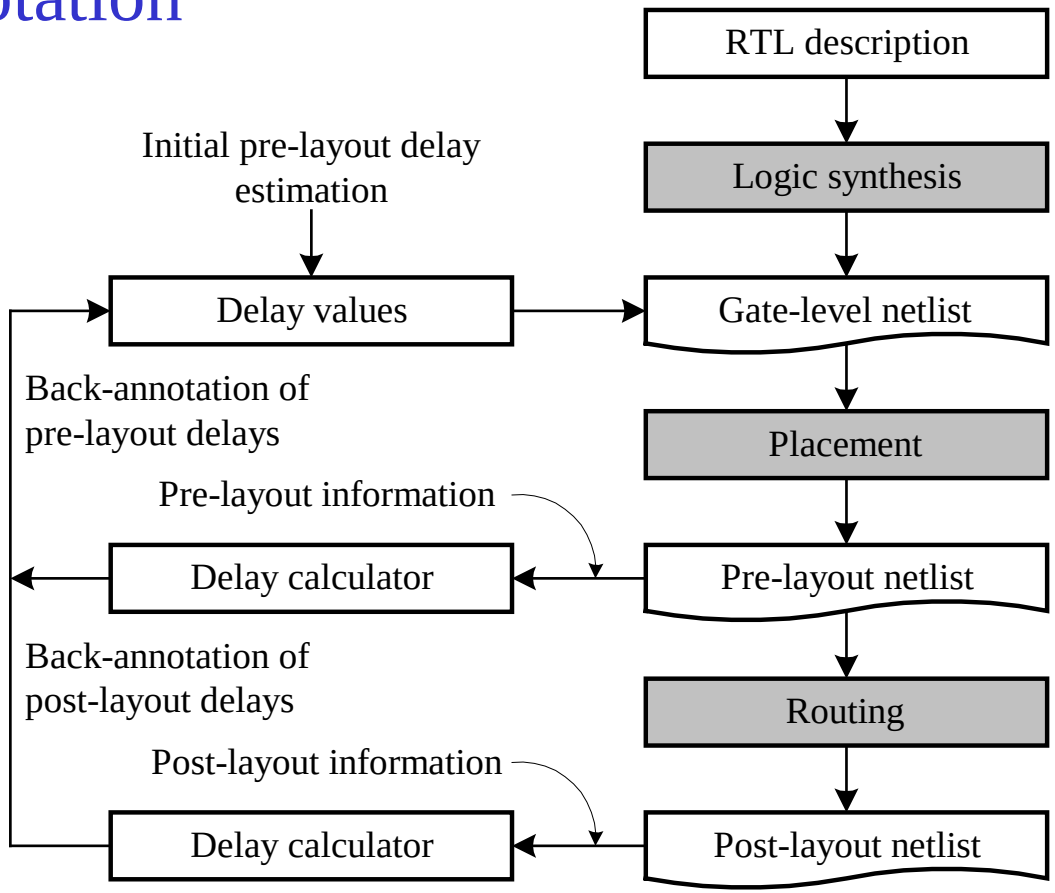
Modeling and Simulation Example --- Simulation Results



Generation of SDF Files

- Pre-layout: The pre-layout numbers contain delay values that are based on the wire-load models.
  - It uses an approximation to generate the pre-layout SDF since the pre-layout netlist does not contain the interconnect delays.
  - For example: *_map.sdf (contains gate delay only) in ISE design flow.
- Post-layout: The post-layout numbers contain delay values that are based on the actual layout, including interconnect delay information.
  - For example: *_timesim.sdf (contains both gate and interconnect delays) in ISE design flow.

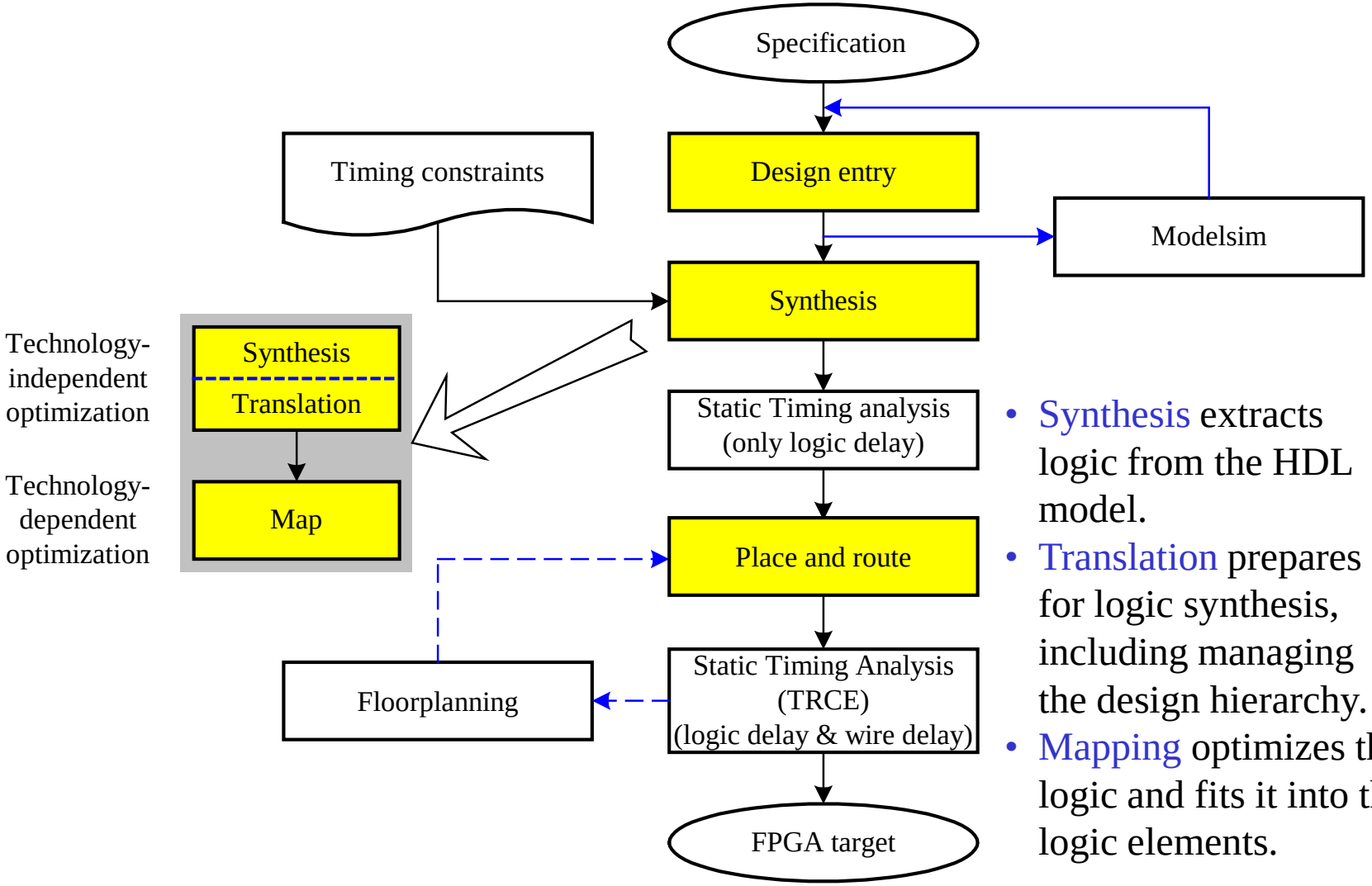
Delay Back-Annotation



In test bench, add:

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

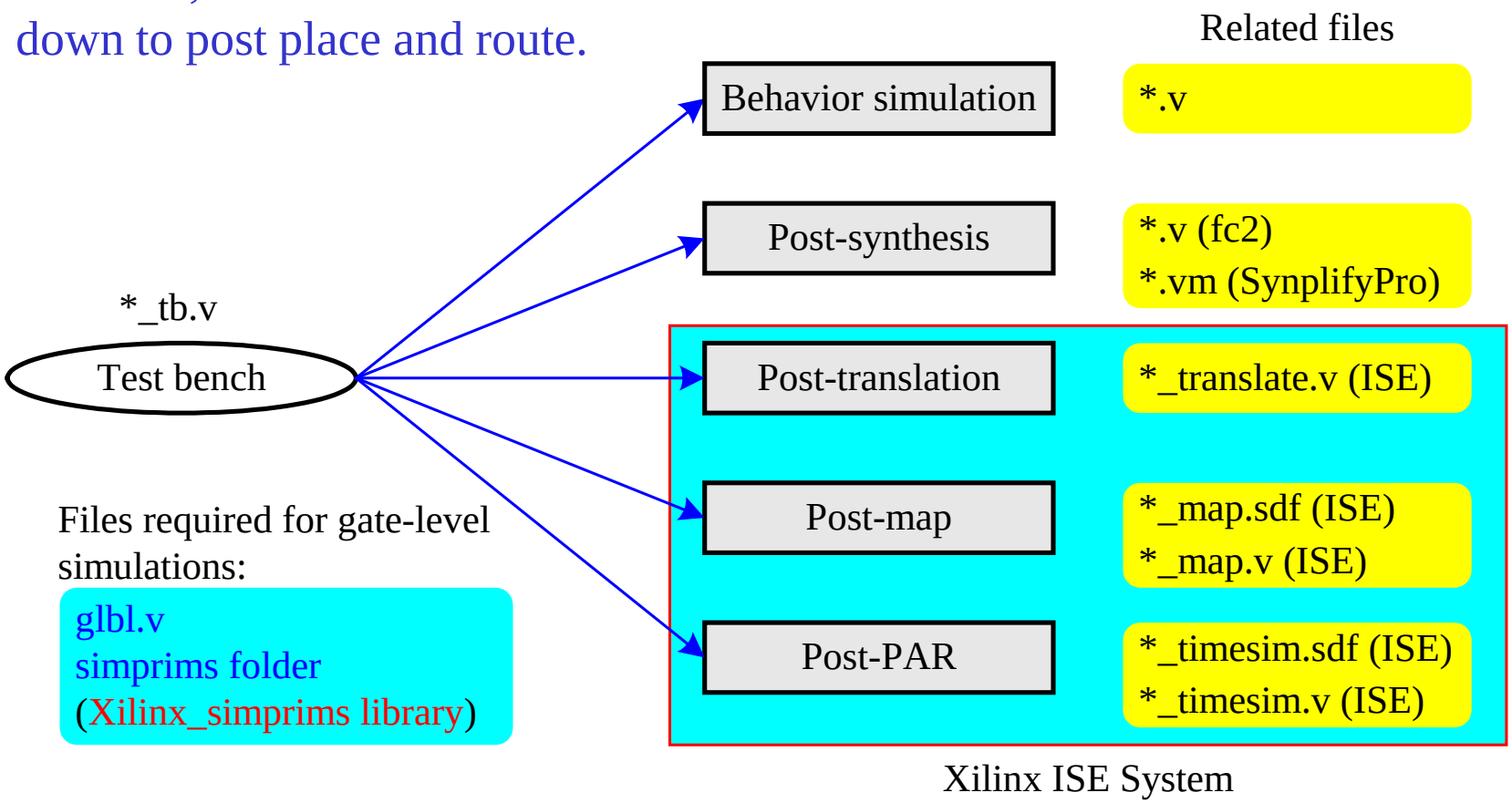
The ISE Design Flow



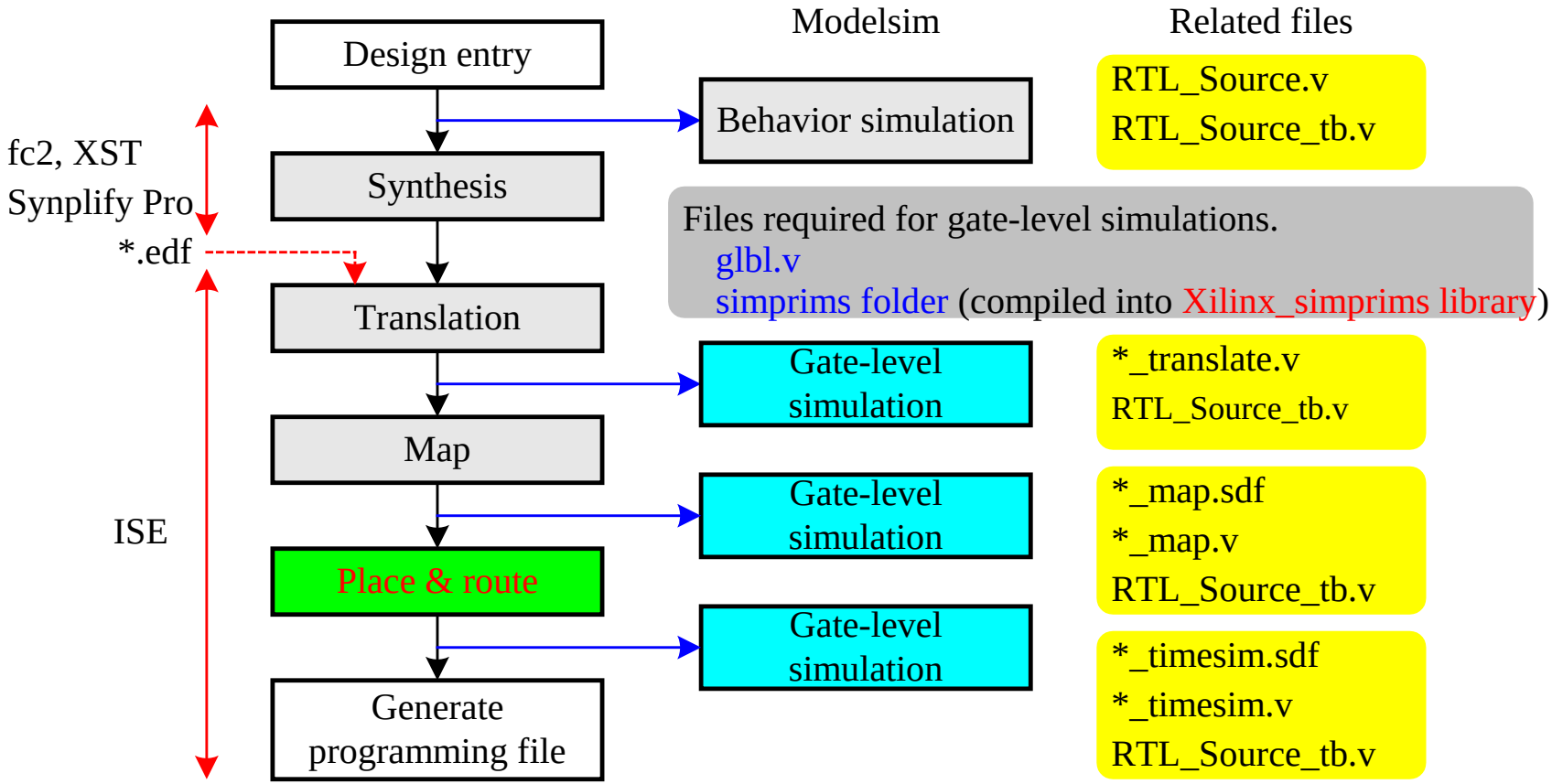
- Synthesis extracts logic from the HDL model.
- Translation prepares for logic synthesis, including managing the design hierarchy.
- Mapping optimizes the logic and fits it into the logic elements.

A Simulation Flow --- The Roles of Test benches

The same test bench is used in all levels, from behavioral down to post place and route.



A Simulation Flow --- An ISE-Based Flow



Swans

Black Swan