Chapter 7: Advanced Modeling Techniques

Prof. Soo-Ik Chae

Objectives

After completing this chapter, you will be able to:

- ❖ Describe the features of sequential blocks
- ❖ Describe the features of parallel blocks
- ❖ Describe the features of nested blocks
- ❖ Describe the features of procedural continuous assignments
- ❖ Describe how to model a module delay
- ❖ Describe the features of specify blocks
- ❖ Describe the features of timing checks

Sequential and Parallel Blocks

- Two block types:
  - Sequential blocks use keywords begin and end to group statements.
  - Parallel blocks use keywords fork and join to group statements.

Sequential Blocks

- The statements are processed in the order specified.
- Delay control can be applied to schedule the executions of procedural statements.

```
initial  begin
          x = 1’b1;    // execute at time 0.
   #12 y = 1’b1;    // execute at time 12.
   #20 z = 1’b0;    // execute at time 32.
end
```

```
initial begin
          x  = 1'b0;    // execute at time 0.
  #20 w  = 1'b1;    // execute at time 20.
  #12 y <= 1'b1;    // execute at time 32.
  #10 z <= 1'b0;    // execute at time 42.
  #25 x   = 1'b1; w = 1'b0;   // execute at time 67.
end
```

Parallel Blocks

- The statements are processed in parallel.
- They are executed relative to the time the blocks are entered.

```
initial  fork
         x = 1'b0;    // execute at time 0.
   #12 y = 1'b1;    // execute at time 12.
   #20 z = 1'b1;    // execute at time 20.
join
```

```
initial  fork
         x <= 1'b0;    // execute at time 0.
   #12 y <= 1'b1;    // execute at time 12.
   #20 z <= 1'b1;    // execute at time 20.
join
```

Special Features of Blocks

- Two special blocks:
  - Nested blocks
  - Named blocks

Nested Blocks

- Nested blocks
  - Blocks can be nested.
  - Both sequential and parallel blocks can be mixed.

```
initial begin
   x = 1'b0;              // execute at time 0.
   fork      // parallel block -- enter at time 0 and leave at time 20.
       #12 y <= 1'b1;  // execute at time 12.
       #20 z <= 1'b0;  // execute at time 20.
   join
   #25 x = 1'b1;       // execute at time 45.
end
```

Named Blocks

- Named blocks
  - Blocks can be given names.
  - Local variables can be declared for the named blocks.
  - Variables in a named block can be accessed by using hierarchical name referencing scheme.
  - Named blocks can be disabled.

```
initial  begin: test    // test is the block name.
reg x, y, z;            // local variables
         x = 1'b0;
   #12 y = 1'b1;        // execute at time 12.
   #10 z = 1'b1;        // execute at time 22.
end
```

Disabling Named Blocks

- The keyword disable can be used
  - to terminate the execution of a named block,
  - to get out of a loop,
  - to handle error conditions, or
  - to control execution of a pieces of code.

```
initial  begin: test              // test is the block name.
   while (i < 10) begin
         if (flag) disable test; // block test is disable if flag is true.
         i = i + 1;
   end
end
```

Procedural Continuous Assignments

- Two kinds of procedural continuous assignments:
  - assign and deassign procedural continuous assignments assign values to variables.
    - They are usually not supported by logic synthesizer.
    - They are now considered as a bad coding style.
  - force and release procedural continuous assignments assign values to nets or variables.
    - They are usually not supported by logic synthesizer.
    - They should appear only in stimulus or as debug statements.

Procedural Continuous Assignments

- assign and deassign constructs
  - Their LHS can be only a variable or a concatenation of variables.
  - They override the effect of regular procedural assignments.
  - They are normally used for controlling periods of time.
- force and release constructs
  - Their LHS can be a variable or a net.
  - They override the effect of regular procedural assignments.
  - They override the effect of assign and deassign construct.
  - They are normally used for controlled periods of time.

Procedural Continuous Assignments

```
// negative edge triggered D flip flop with asynchronous reset.
module edge_dff(input clk, reset, d, output reg q, qbar);
always @(negedge clk) begin
     q <= d; qbar <= ~d;
end
always @(reset)      // override the regular assignments to q and qbar.
     if (reset) begin
          assign q = 1'b0;
          assign qbar = 1'b1;
    end else begin     // remove the overriding values
          deassign q;
          deassign qbar;
    end
endmodule
```

Modes of Assignments

Data type	Primitive output	Continuous assignment	Procedural assignment	assign deassign	force release
Net	Yes	Yes	No	No	Yes
Variable	Yes (Seq.UDP)	No	Yes	Yes	Yes

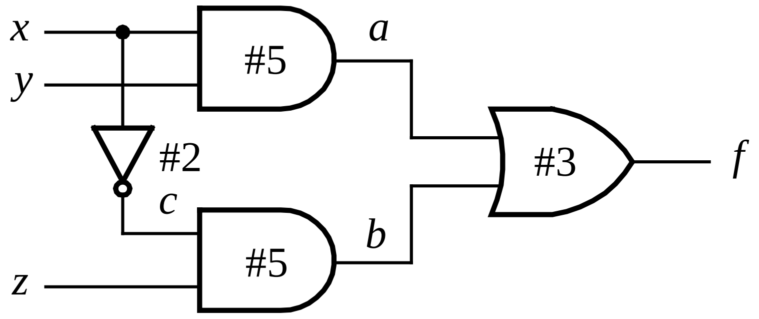
Type of Delays

- Three types of delay models:
  - Distributed delays
  - Lumped delays
  - Module path (pin-to-pin) delays
- Both distributed and module path (pin-to-pin) delays are often used to describe the delays for structural modules such as ASIC cells.

Distributed Delay Model

- Distributed delay model
  - Delays are considered to be associated with individual element, gate, cell, or interconnect.
  - Distributed delays are specified on a per element basis.
  - It specifies the time it takes the events to propagate through gates and nets inside the module.

Distributed Delay Model



```
module M (input  x, y, z, output f);
wire a, b, c;

and  #5 a1 (a, x, y);
not   #2 n1 (c, x);
and  #5 a2 (b, c, z);
or    #3 o1 (f, a, b);
endmodule
```

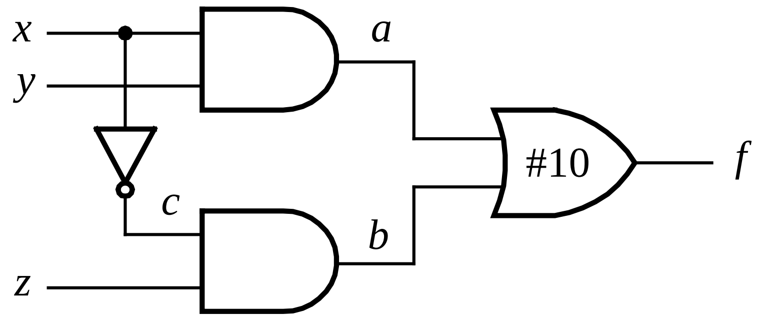
```
module M (input  x, y, z , output f);
wire a, b, c;

assign  #5  a = x & y;
assign  #2  c = ~x
assign  #5  b = c & z
assign  #3  f = a | b;
endmodule
```

Lumped Delay Model

- Lumped delay model
  - Delays are associated with the entire module.
  - The cumulative delay of all paths are lumped at the single output.
  - A lumped delay is specified on a per output basis.
  - It specifies the time it takes the events to propagate at the last gate along the path.

Lumped Delay Model



```
module M (input  x, y, z , output f);
wire a, b, c;

and      a1 (a, x, y);
not       n1 (c, x);
and      a2 (b, c, z);
or        #10 o1 (f, a, b);
endmodule
```

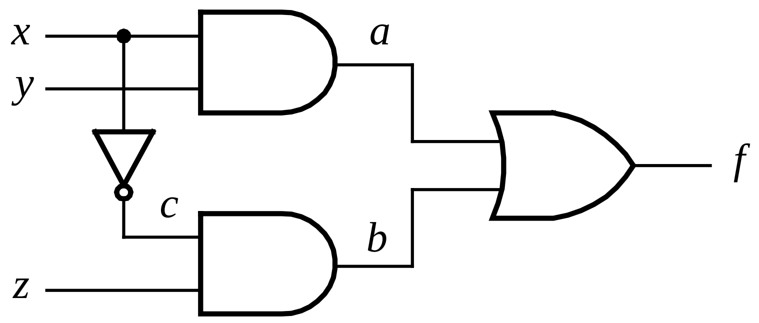
```
module M (input  x, y, z , output f);
wire a, b, c;

assign      a = x & y;
assign      c = ~x
assign      b = c & z
assign      #10 f = a | b;
endmodule
```

Module Path Delay Model

- Module path (pin-to-pin) delay model
  - Delays are individually assigned to each module path.
  - The input may be an input port or an inout (bidirectional) port.
  - The output may be an output port or an inout port.
  - A path delay is specified on a pin-to-pin (port-to-port) basis. This kind of paths is called module path.
  - It specifies the time it takes an event at a source (input port or inout port) to propagate to a destination (output port or inout port).

Module Path Delay Model



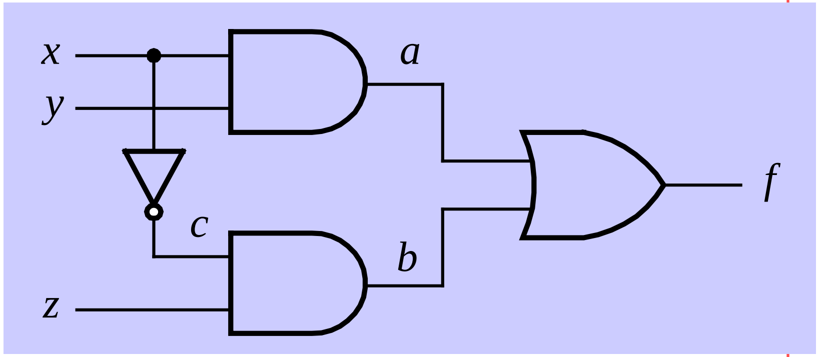
Path x-a-f,	delay = 8
Path x-c-b-f,	delay = 10
Path y-a-f,	delay = 8
Path z-b-f,	delay = 8

Specify Blocks

- A specify block is declared within a module by keywords specify and endspecify.
- The specify block is used to
  - describe various paths across the module
  - assign delays to these paths
  - perform necessary timing checks

Path Delay Modeling – The specify Block

```
module M (input  x, y, z , output f);
wire a, b, c;
// specify block with path delay statements
specify
     (x => f) = 10;
     (y => f) = 8;
     (z => f) = 8;
endspecify
// gate instantiations
    and  a1 (a, x, y);
    not   n1 (c, x);
    and  a2 (b, c, z);
    or    o1 (f, a, b);
endmodule
```



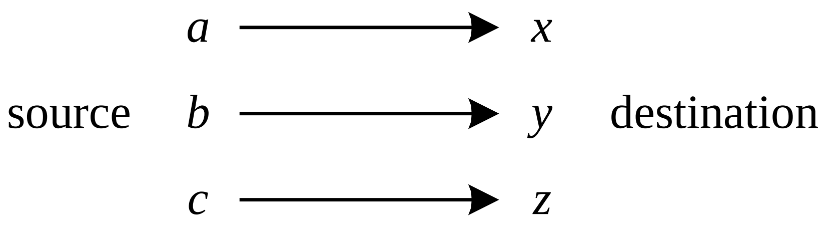
Path x-a-f,   delay = 8
Path x-c-b-f, delay = 10
Path y-a-f,   delay = 8
Path z-b-f,   delay = 8

Path Declarations

- Path declarations:
  - Single-path
  - Edge-sensitive path
  - State-dependent path

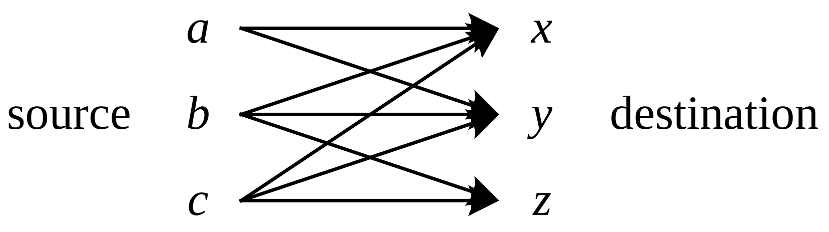
Single Path

- Single path connection methods:
  - Parallel connection (source => destination)
  - Full connection (source *> destination)



```
(a => x) = 10;
(b => y) = 8;
(c => z) = 8;
```

(a) Parallel connection



```
(a, b, c *> x) = 8;
(a, b, c *> y) = 10;
(b, c *> z)    = 12;
```

(b) Full connection

Edge-Sensitive Path

- Edge-sensitive path:
  - posedge clock => (out +: in) = (8, 6);
  - At the positive edge of clock, a module path extends from clock to out using a rise time of 8 and a fall delay of 6. The data path is from in to out.
  - negedge clock => (out -: in) = (8, 6);
  - At the negative edge of clock, a module path extends from clock to out using a rise time of 8 and a fall delay of 6. The data path is from in to out.
- Level-sensitive path:
  - clock => (out : in) = (8, 6);
  - At any change in clock, a module path extends from clock to out.

State-Dependent Path

- The module path delay is assigned conditionally based on the value of the signals in the circuit.

```
if (cond_expr) simple_path_declaration
if (cond_expr) edge_sensitive_path_declaration
ifnone simple_path_declaration
```

```
specify
   if (x)   (x => f) = 10;
   if (~x) (y => f) = 8;
endspecify
```

```
specify
   if (!reset && !clear)
     (positive clock => (out +: in) = (8, 6);
endspecify
```

The specparam Statement

- specparam statements are
  - usually used to define specify parameters.
  - used only inside their own specify block.

```
specify
// define parameters inside the specify block
    specparam d_to_q    = (10, 12);  // two value delay specification
    specparam clk_to_q = (15, 18);
    ( d => q)   = d_to_q;
    (clk => q) = clk_to_q;
endspecify
```

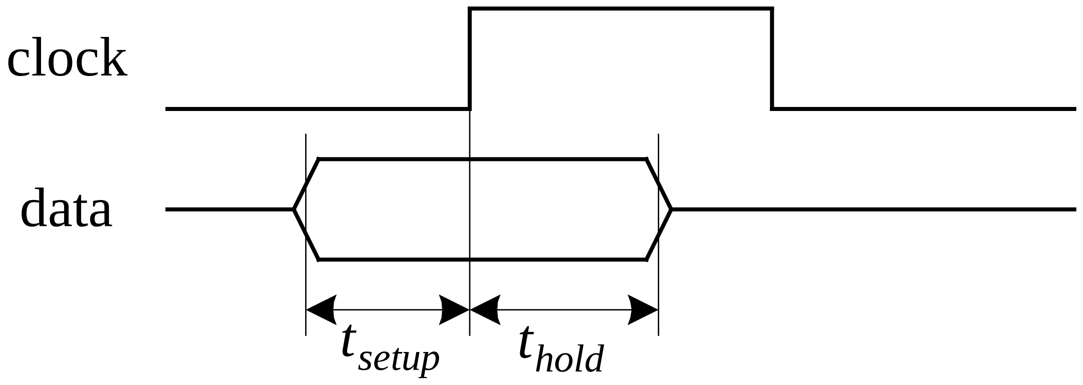
An Example --- An NOR Gate

```
module my_nor (a, b, out);
input a, b:
output out;
    nor nor1 (out, a, b);
    specify
        specparam trise = 1, tfall = 2
        specparam trise_n = 2, tfall_n = 3;
        if (a) (b => out)  = (trise, tfall);
        if (b) (a => out)  = (trise, tfall);
        if (~a)(b => out) = (trise_n, tfall_n);
        if (~b)(a => out) = (trise_n, tfall_n);
      endspecify
endmodule
```

Timing Checks

❖ Timing Checks:
- Tasks are provided to do timing checks.
- Although they begin with $, timing checks are not system tasks
- All timing checks must be inside the specify blocks.
- The following tasks are most commonly used:
  - $setup
  - $hold
  - $setuphold
  - $width
  - $skew
  - $period
  - $recovery

Timing Checks -- $setup Task

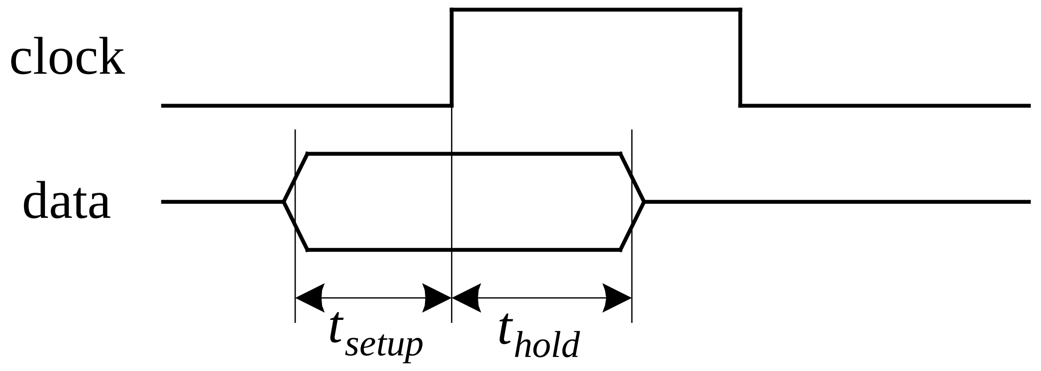


$setup (data_event, reference_event, limit);

Violation is reported when $t_{\text{reference\_event}} - t_{\text{data\_event}} < \text{limit}$.

```
specify
   $setup (data, posedge clock, 15);
endspecify
```

Timing Checks -- $hold Task

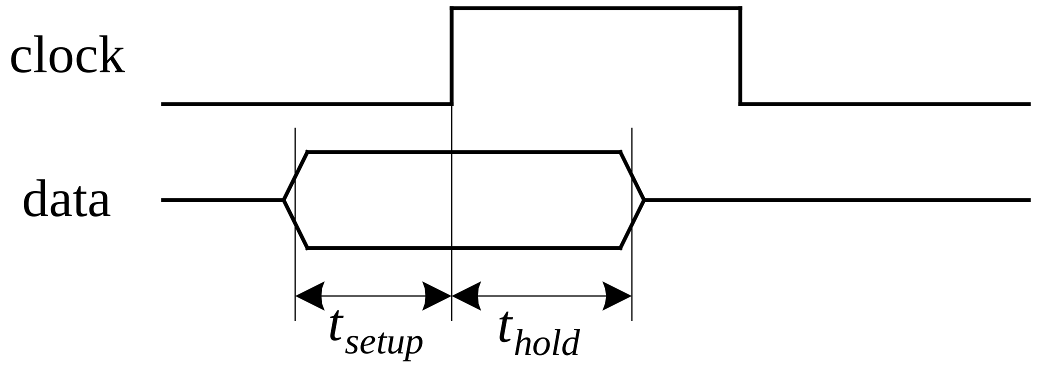


$hold (reference_event, data_event, limit);

Violation is reported when $t_{\text{data\_event}} - t_{\text{reference\_event}} <$ limit.

```
specify
   $hold (posedge clock, data, 8);
endspecify
```

Timing Checks -- $setuphold Task



$setuphold (reference_event, data_event, setup_limit, hold_limit);

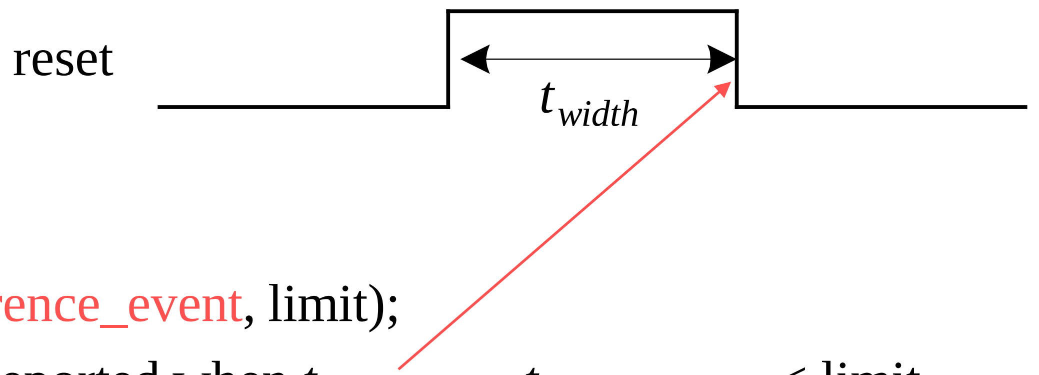
Violation is reported when:

$t_{\text{reference\_event}} - t_{\text{data\_event}} <$ setup_limit.

$t_{\text{data\_event}} - t_{\text{reference\_event}} <$ hold_limit.

```
specify
   $setuphold (posedge clock, data, 10, -3);
endspecify
```

Timing Checks -- $width Task



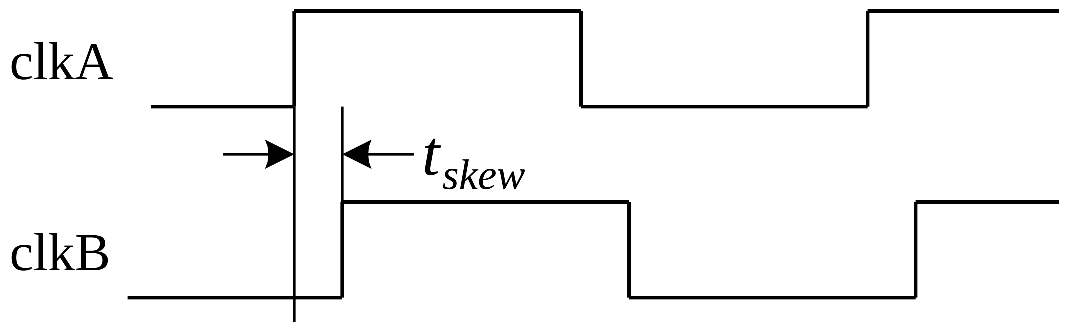
$width (reference_event, limit);

Violation is reported when $t_{data\_event} - t_{reference\_event} <$ limit.

Data_event is the next opposite edge of the reference_event signal.

```
specify
   $width (posedge reset, 6);
endspecify
```

Timing Checks -- $skew Task

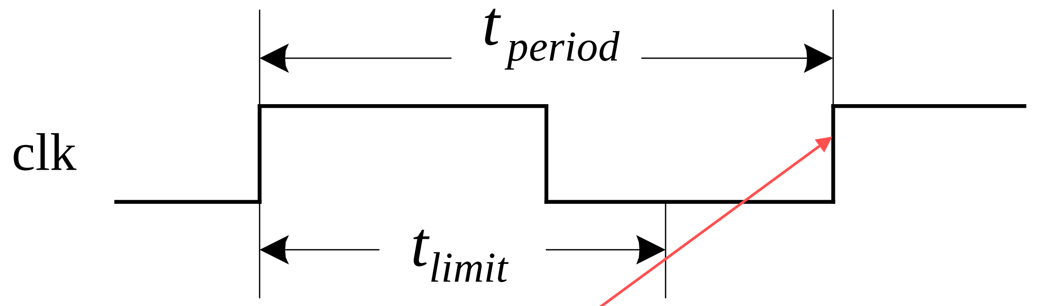


$skew (reference_event, data_event, limit);

Violation is reported when $t_{\text{data\_event}} - t_{\text{reference\_event}} > \text{limit}$.

```
specify
   $skew (posedge clkA posedge clkA, 5);
endspecify
```

Timing Checks -- $period Task



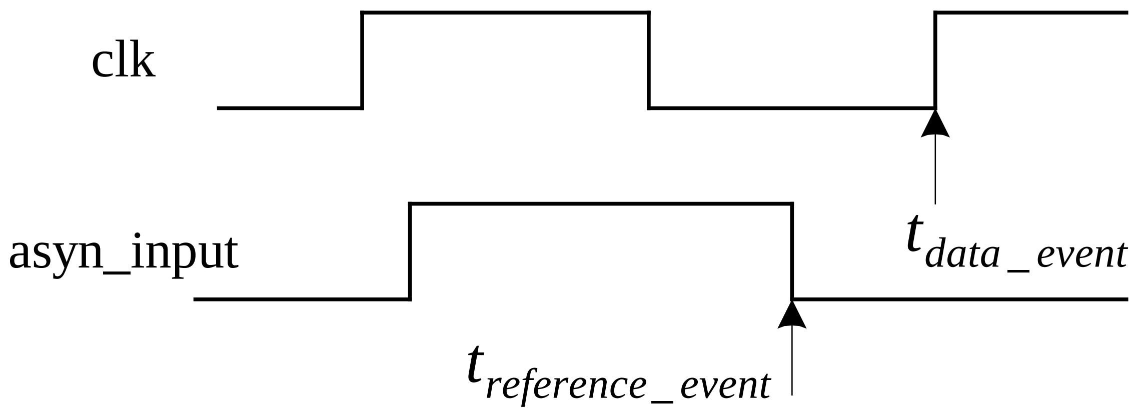
$period (reference_event, limit);

Violation is reported when $t_{\text{data\_event}} - t_{\text{reference\_event}} < \text{limit}$.

Data_event is the next same edge of the reference_event signal.

```
specify
   $period (posedge clk, 15);
endspecify
```

Timing Checks -- $recovery Task



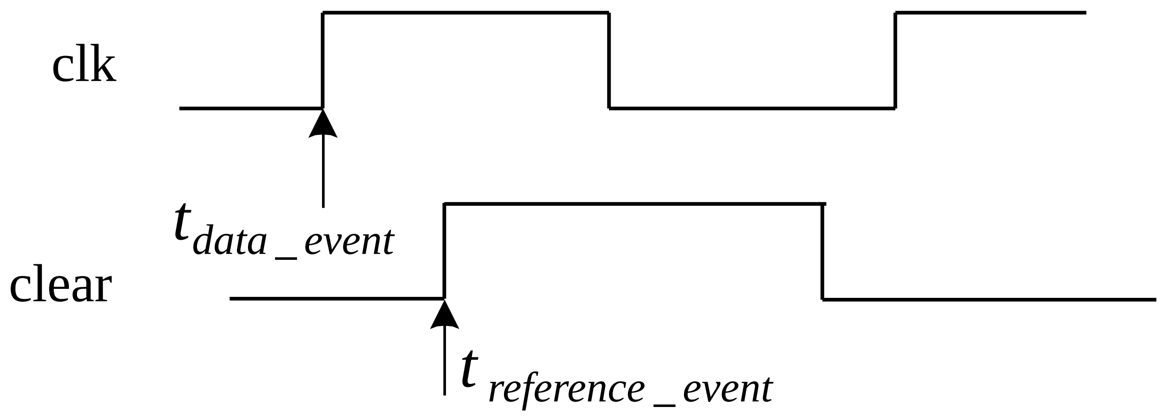
$recovery (reference_event, data_event, limit);

Violation is reported when $t_{\text{reference\_event}} <= t_{\text{data\_event}} < t_{\text{reference\_event}} + \text{limit}$.

It specifies the minimum time that an asynchronous input must be stable before the active edge of the clock. (compare to $hold task)

```
specify
   $recovery (negedge aysyn_input, posedge clk, 5);
endspecify
```

Timing Checks -- $removal Task



$removal (reference_event, data_event, limit);

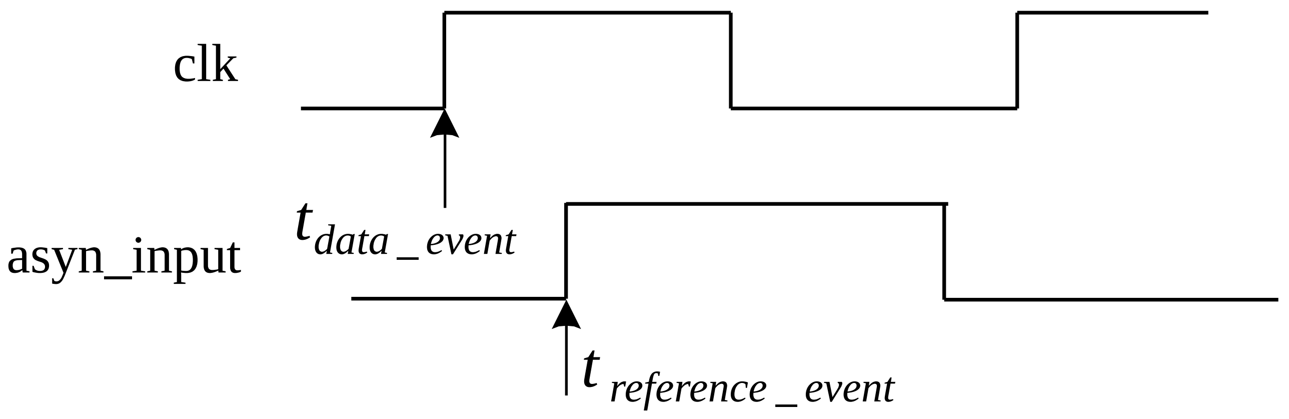
Violation is reported when $t_{\text{reference\_event}} - \text{limit} <= t_{\text{data\_event}} < t_{\text{reference\_event}}$

t.

It specifies the minimum time that an asynchronous input must be stable after the active edge of the clock. (compare to $setup task)

```
specify
   $removal (posedge clear, posedge clk, 5);
endspecify
```

Timing Checks -- $recrem Task



$recrem (reference_event, data_event, t_rec, t_rem);

It specifies the minimum time that an asynchronous input must be stable before and after the active edge of the clock. (compare to $setuphold task)

```
specify
   $recrem (posedge clear, posedge clk, 5, -2);
endspecify
```