ECE 327/627 Digital Hardware Systems

Lecture 3: SystemVerilog Basics II

Andrew Boutros

andrew.boutros@uwaterloo.ca

Some Logistics

- Lab group registration is open on LEARN
  - Connect > Groups > View Available Groups
  - Deadline is **January 16 @ 11:59 pm**

- Lab 1 is released
  - Worth **5%** of the total course grade for ECE 327 students
  - Worth **3%** of the total course grade for ECE 627 students
  - Deadline **January 23 @ 11:59 pm**

- Tutorial session will be held this week
  - Wednesday 12:30 - 1:20 pm @ PSE-4053
  - More SystemVerilog examples

In Previous Lecture …

- SystemVerilog basics
  - Typical module skeleton
  - Ports
  - Continuous assignment
  - Signals (`logic`)
  - Module parameterization
  - Hierarchical instantiation
  - Procedural blocks
    - `initial` – behavior happens only once at beginning of time
    - `always_comb` – behavior happens every time a RHS signal changes
    - `always_ff` – behavior happens every pos/neg clock edge
  - Blocking (=) vs. non-blocking (<=) assignments
    - Blocking → "sequential" description of the behavior
    - Non-blocking → "concurrent" description of the behavior

One More Example: Blocking vs. Non-Blocking

```
module poly1 (
     input  clk,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     input  [7:0] x,
     output [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
     y0 = a * x;
     y1 = y0 + b;
     y2 = y1 * x;
     y3 = y2 + c;
end

assign y = y3;

endmodule
```

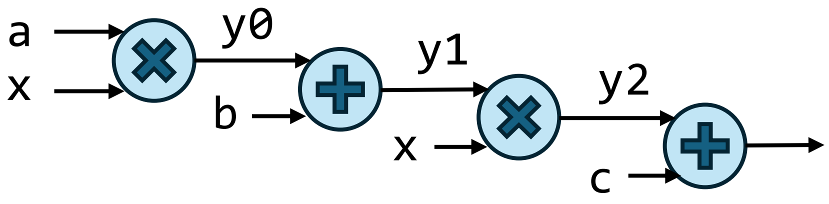
```
module poly2 (
     input  clk,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     input  [7:0] x,
     output [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
     y0 <= a * x;
     y1 <= y0 + b;
     y2 <= y1 * x;
     y3 <= y2 + c;
end

assign y = y3;

endmodule
```

One More Example: Blocking vs. Non-Blocking

```
module poly1 (
     input  clk,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     input  [7:0] x,
     output [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
     y0 = a * x;
     y1 = y0 + b;
     y2 = y1 * x;
     y3 = y2 + c;
end

assign y = y3;

endmodule
```

One More Example: Blocking vs. Non-Blocking

```
module poly1 (
     input  clk,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     input  [7:0] x,
     output [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
     y0 = a * x;
     y1 = y0 + b;
     y2 = y1 * x;
     y3 = y2 + c;
end

assign y = y3;

endmodule
```



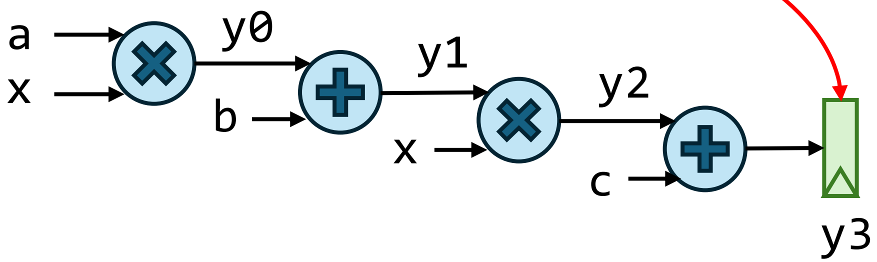
Blocking assignments mean that this behavior is described sequentially. You can substitute dependencies:

$$y3 = \underbrace{(\underbrace{(\underbrace{(a * x)}_{y0} + b)}_{y1} * x)}_{y2} + c;$$

One More Example: Blocking vs. Non-Blocking

```
module poly1 (
     input  clk,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     input  [7:0] x,
     output [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
     y0 = a * x;
     y1 = y0 + b;
     y2 = y1 * x;
     y3 = y2 + c;
end

assign y = y3;

endmodule
```

always_ff means that final results of the procedural block are registered



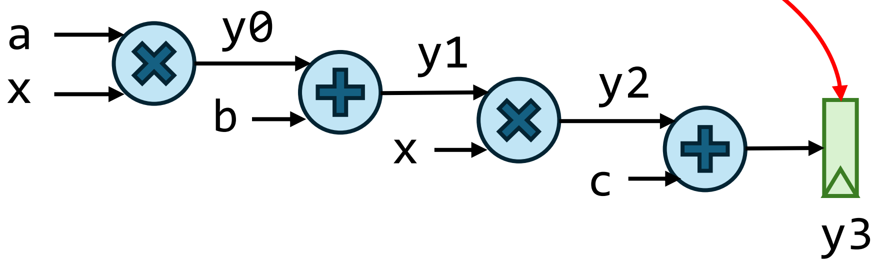
Blocking assignments mean that this behavior is described sequentially. You can substitute dependencies:

$$y3 = (\underbrace{(\underbrace{(\underbrace{a * x)}_{y0} + b)}_{y1} * x)}_{y2} + c;$$

One More Example: Blocking vs. Non-Blocking

```
module poly1 (
     input  clk,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     input  [7:0] x,
     output [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
     y0 = a * x;
     y1 = y0 + b;
     y2 = y1 * x;
     y3 = y2 + c;
end

assign y = y3;

endmodule
```

always_ff means that final results of the procedural block are registered



Blocking assignments mean that this behavior is described sequentially. You can substitute dependencies:

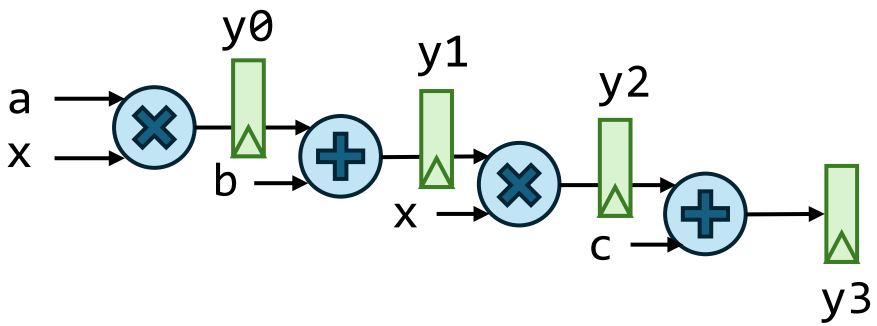
$$y3 = \underbrace{(\underbrace{(\underbrace{(a * x)}_{y0} + b)}_{y1} * x)}_{y2} + c;$$

What if we change always_ff to always_comb?

One More Example: Blocking vs. Non-Blocking

```
module poly2 (
     input  clk,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     input  [7:0] x,
     output [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
     y0 <= a * x;
     y1 <= y0 + b;
     y2 <= y1 * x;
     y3 <= y2 + c;
end

assign y = y3;

endmodule
```

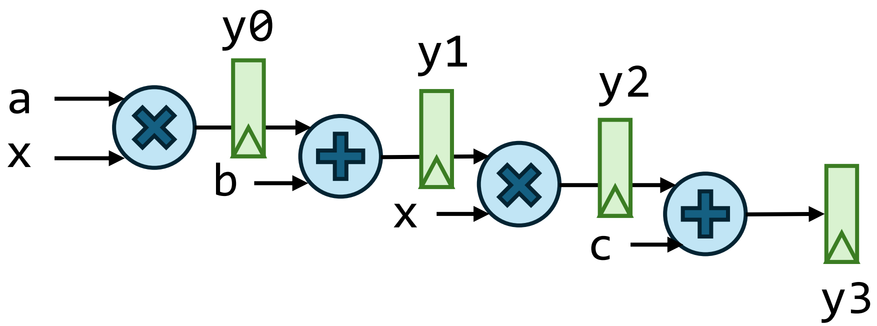
One More Example: Blocking vs. Non-Blocking



Concurrent non-blocking assignments (i.e., The LHS of all non-blocking assignments in an `always_ff` translate to registers)

```
module poly2 (
     input  clk,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     input  [7:0] x,
     output [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
     y0 <= a * x;
     y1 <= y0 + b;
     y2 <= y1 * x;
     y3 <= y2 + c;
end

assign y = y3;

endmodule
```

One More Example: Blocking vs. Non-Blocking

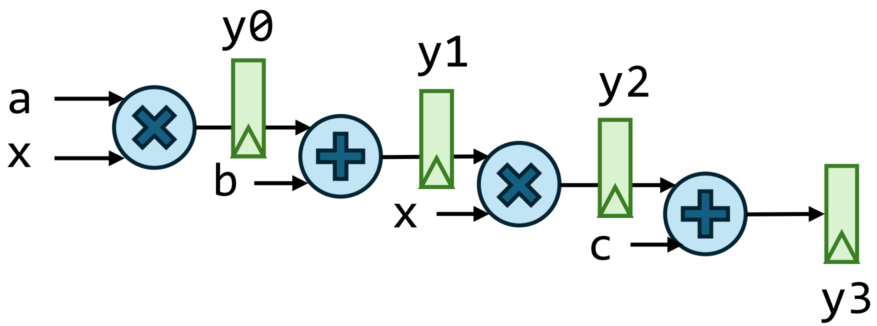


Concurrent non-blocking assignments
(i.e., The LHS of all non-blocking assignments in an always_ff translate to registers)

🍬 **Does this work correctly if the inputs change every clock cycle?**

```
module poly2 (
     input  clk,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     input  [7:0] x,
     output [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
     y0 <= a * x;
     y1 <= y0 + b;
     y2 <= y1 * x;
     y3 <= y2 + c;
end

assign y = y3;

endmodule
```

One More Example: Blocking vs. Non-Blocking



Concurrent non-blocking assignments
(i.e., The LHS of all non-blocking assignments in an `always_ff` translate to registers)

🍬 **Does this work correctly if the inputs change every clock cycle?**

More on this in the pipeline lecture later in the course

```
module poly2 (
        input   clk,
        input   [7:0] a,
        input   [7:0] b,
        input   [7:0] c,
        input   [7:0] x,
        output  [23:0] y
);

logic [15:0] y0;
logic [15:0] y1;
logic [23:0] y2;
logic [23:0] y3;

always_ff @ (posedge clk) begin
        y0 <= a * x;
        y1 <= y0 + b;
        y2 <= y1 * x;
        y3 <= y2 + c;
end

assign y = y3;

endmodule
```

Recommended Best Practices for <= vs. =

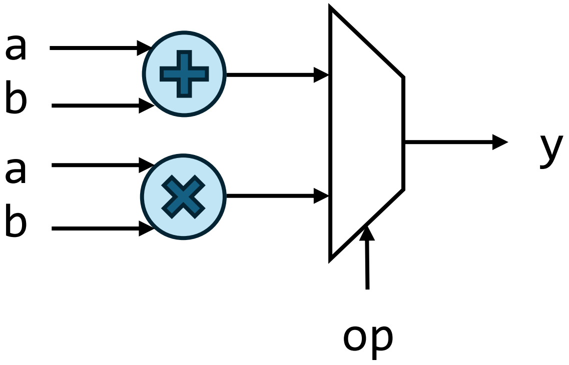
- The language allows using blocking (=) or non-blocking (<=) assignments to describe sequential or combinational logic

- Can get really confusing and cause unintentional bugs if you don't know exactly what you are doing!

- Best practices and & guidelines to minimize sources of errors
  - Use non-blocking (<=) assignments only to describe sequential (i.e., clocked) logic in `always_ff` blocks
  - Use blocking (=) assignments only to describe combinational logic in `always_comb` blocks

Conditions in Always Blocks

- In always blocks, condition branches instantiate distinct circuits + multiplexer
- The multiplexer select line is the condition logic
- Circuits corresponding to all conditions process inputs, but only one output is selected
  - Compare to software branches?



```
module add_mult (
     input  signed [7:0] a,
     input  signed [7:0] b,
     input  op,
     output signed [15:0] y
);

logic signed [15:0] res;

always_comb begin
     if (!op) begin
          res = a + b;
     end else begin
          res = a * b;
     end
end

assign y = res;

endmodule
```

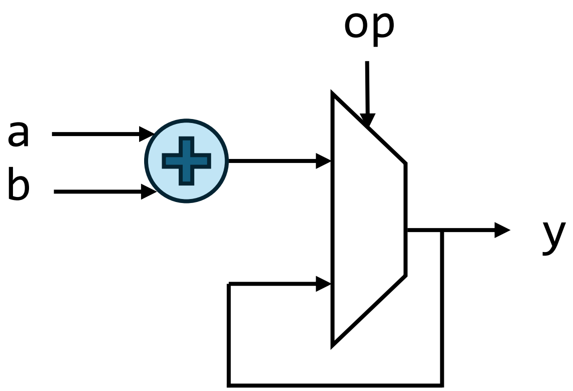
=

```
module add_mult (...);

assign y = (!op)? a+b : a*b;

endmodule
```

Incomplete Conditions

- If a conditional statement is incomplete (missing else), this generates a feedback from the mux output to its input
  - In combinational circuit → latch
    - Initial value problem
    - Need to be very carefully designed to avoid metastability (feedback delay for the value to stabilize before select line changes)

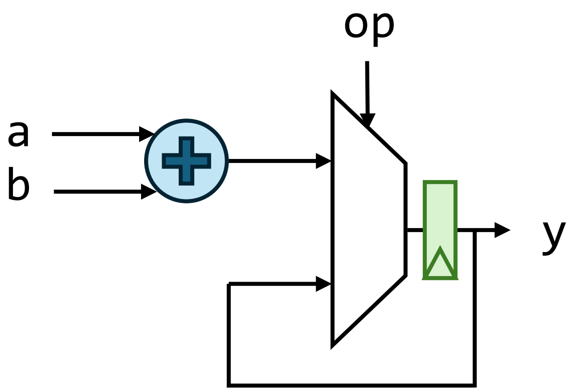
```
module add_mult (
     input  signed [7:0] a,
     input  signed [7:0] b,
     input  op,
     output signed [15:0] y
);

logic signed [15:0] res;

always_comb begin
     if (!op) begin
          res = a + b;
     end
end

assign y = res;

endmodule
```



Incomplete Conditions

- If a conditional statement is incomplete (missing else), this generates a feedback from the mux output to its input
  - In combinational circuit → latch
    - Initial value problem
    - Need to be very carefully designed to avoid metastability (feedback delay for the value to stabilize before select line changes)
  - In sequential circuit → feedback loop broken by a register stage

```
module add_mult (
     input  clk,
     input  signed [7:0] a,
     input  signed [7:0] b,
     input  op,
     output signed [15:0] y
);
logic signed [15:0] res;

always_ff @ (posedge clk) begin
     if (!op) begin
          res = a + b;
     end
end

assign y = res;

endmodule
```



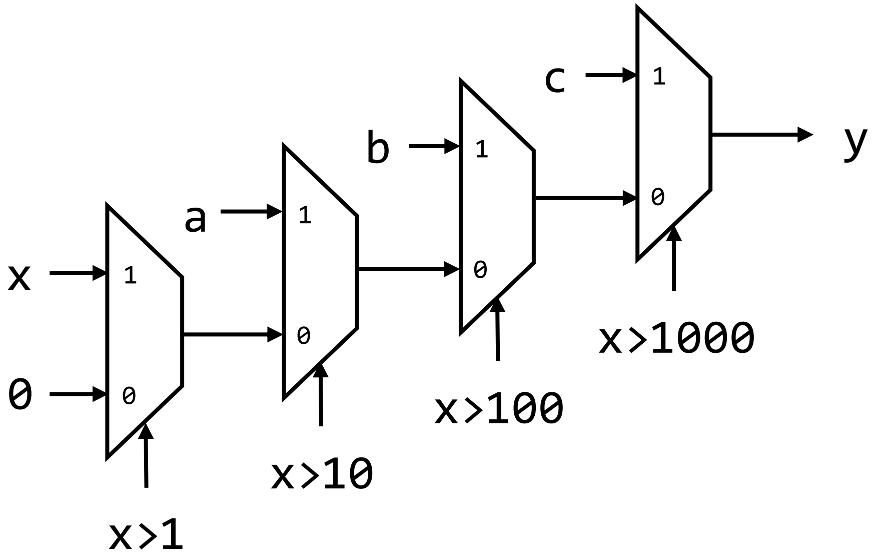
Non-mutually Exclusive Conditions

```
...

always_comb begin
     if (x > 1000) begin
          y = c;
     end else if (x > 100) begin
          y = b;
     end else if (x > 10) begin
          y = a;
     end else if (x > 1) begin
          y = x
     end else begin
          y = 0;
     end
end

...
```

Highest priority

Lowest priority

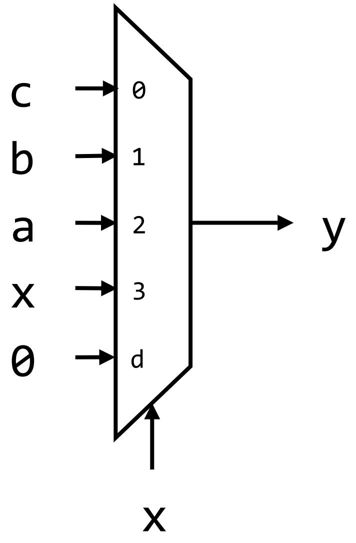
- If-else conditions can be non-mutually exclusive
  - e.g., x = 1001 satisfies all conditions)
- In this case, the order defines condition priority



Mutually Exclusive Conditions

```
...

always_comb begin
     case(x)
          8’h00: begin
               y = c;
          end
          8’h01: begin
               y = b;
          end
          8’h02: begin
               y = a;
          end
          8’h03: begin
               y = x;
          end
          default: begin
               y = 0;
          end
     endcase
end

...
```

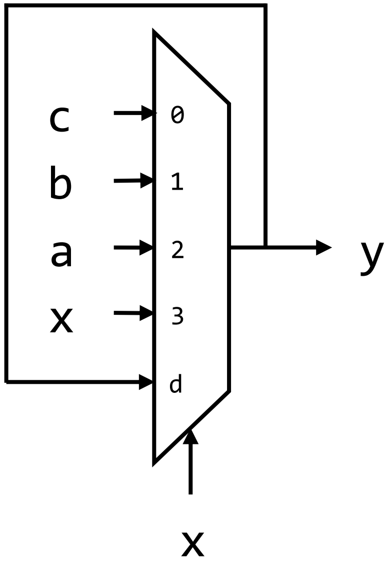
- Can use if-else conditions or a case statement
- No notion of condition priority



Mutually Exclusive Conditions

```
...

always_comb begin
     case(x)
          8’h00: begin
               y = c;
          end
          8’h01: begin
               y = b;
          end
          8’h02: begin
               y = a;
          end
          8’h03: begin
               y = x;
          end



     endcase
end

...
```

- Can use if-else conditions or a case statement
- No notion of condition priority
- Removing the `default` case infers a latch (same as missing else)

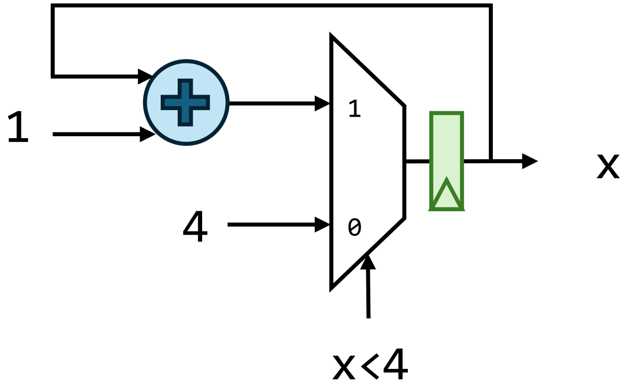


Default Condition in If-Else

```
...

always_ff @ (posedge clk) begin
     if (rst) begin
          x <= 0;
     end else begin
          x <= 4;
          if (x < 4) begin
               x <= x + 1;
          end
     end
end

...
```

- In procedural blocks, the last non-blocking assignment of a signal wins!
  - Previous assignments are ignored

- This can be used to define a default condition and avoid inferring a feedback loop if there is a no else



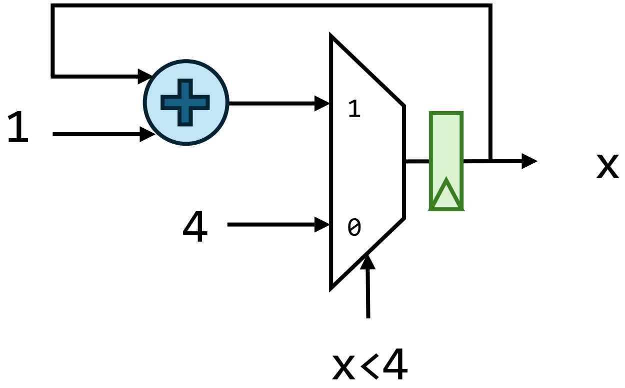
Default Condition in If-Else

```
...

always_ff @ (posedge clk) begin
     if (rst) begin
          x <= 0;
     end else begin
          if (x < 4) begin
               x <= x + 1;
          end
          x <= 4;
     end
end

...
```

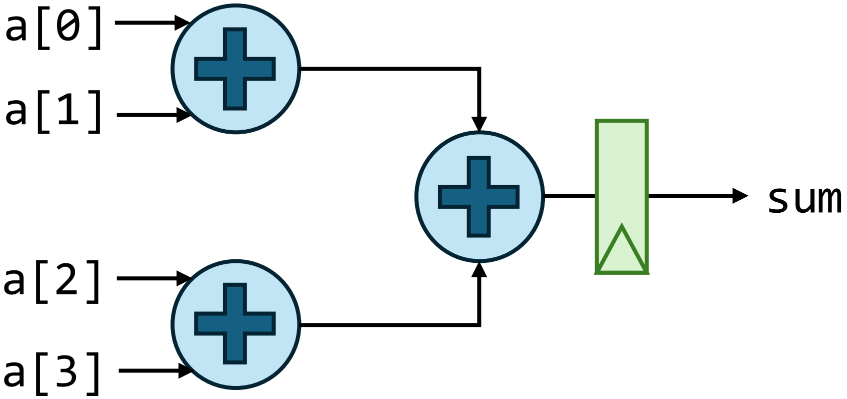
What hardware will be generated if we flip the order?

- In procedural blocks, the last non-blocking assignment of a signal wins!
    - Previous assignments are ignored

- This can be used to define a default condition and avoid inferring a feedback loop if there is a no else



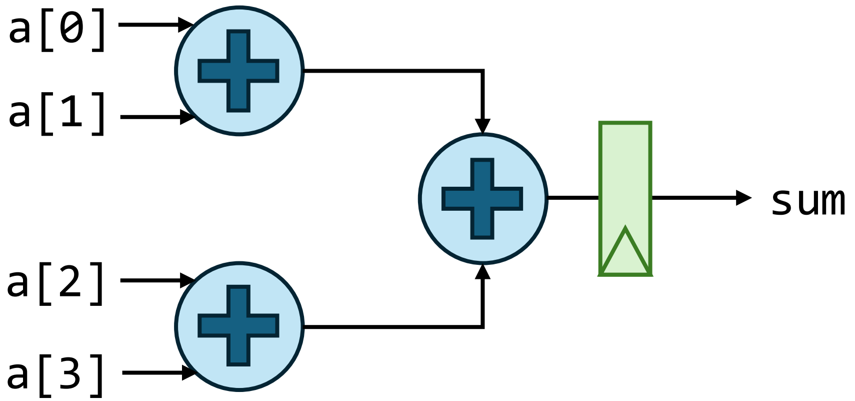
Vector Summation Example



Behavioral description that synthesizes similar hardware to our hierarchical implementation from last lecture

```
module vector_sum (
     input  clk,
     input  rst,
     input  signed [7:0] a [0:3],
     output signed [31:0] sum
);

logic signed [31:0] res;

always_ff @ (posedge clk) begin
     if (rst) begin
          res <= 0;
     end else if
          res <= a[0] + a[1] + a[2] + a[3];
     end
end

assign sum = res;

endmodule
```

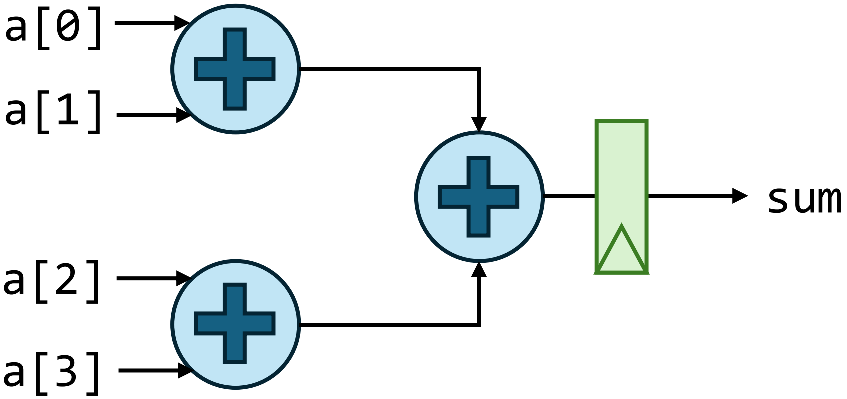
Vector Summation Example



```
module vector_sum (
     input  clk,
     input  rst,
     input  signed [7:0] a [0:99],
     output signed [31:0] sum
);

logic signed [31:0] res;

always_ff @ (posedge clk) begin
     if (rst) begin
          res <= 0;
     end else if
          res <= a[0] + a[1] + ... + a[99];
     end
end

assign sum = res;

endmodule
```

What if vector a had 100 elements?

Vector Summation Example



```
module vector_sum #(
    parameter N = 128
)(
    input  clk,
    input  rst,
    input  signed [7:0] a [0:N-1],
    output signed [31:0] sum
);

logic signed [31:0] res;

always_ff @ (posedge clk) begin
    if (rst) begin
        res <= 0;
    end else if
        res <= ???
    end
end

assign sum = res;

endmodule
```

What if vector a had 100 elements?

Or what if I want to implement a parameterizable module?

For Loops in Procedural Blocks

- For loops can be used to describe circuit behavior in procedural blocks
- Loop iterator variable is defined as an `integer`
    - This is not synthesized in hardware – just to describe the behavior of the circuit

```
module vector_sum #(
    parameter N = 128
)(
    input  clk,
    input  rst,
    input  signed [7:0] a [0:N-1],
    output signed [31:0] sum
);

logic signed [31:0] res;
integer i;

always_ff @ (posedge clk) begin
    if (rst) begin
        res = 0;
    end else begin
        res = 0;
        for (i = 0; i < N; i = i + 1) begin
            res = res + a[i];
        end
    end
end
assign sum = res;

endmodule
```

For Loops in Procedural Blocks

- For loops can be used to describe circuit behavior in procedural blocks
- Loop iterator variable is defined as an `integer`
    - This is not synthesized in hardware – just to describe the behavior of the circuit
- This synthesizes into a large adder tree to sum up the elements of the vector
    - Remember that the loop is describing the behavior – **There are no iterations in the synthesized hardware**

```
module vector_sum #(
        parameter N = 128
)(
        input  clk,
        input  rst,
        input  signed [7:0] a [0:N-1],
        output signed [31:0] sum
);

logic signed [31:0] res;
integer i;

always_ff @ (posedge clk) begin
        if (rst) begin
                res = 0;
        end else begin
                res = 0;
                for (i = 0; i < N; i = i + 1) begin
                        res = res + a[i];
                end
        end
end
assign sum = res;

endmodule
```

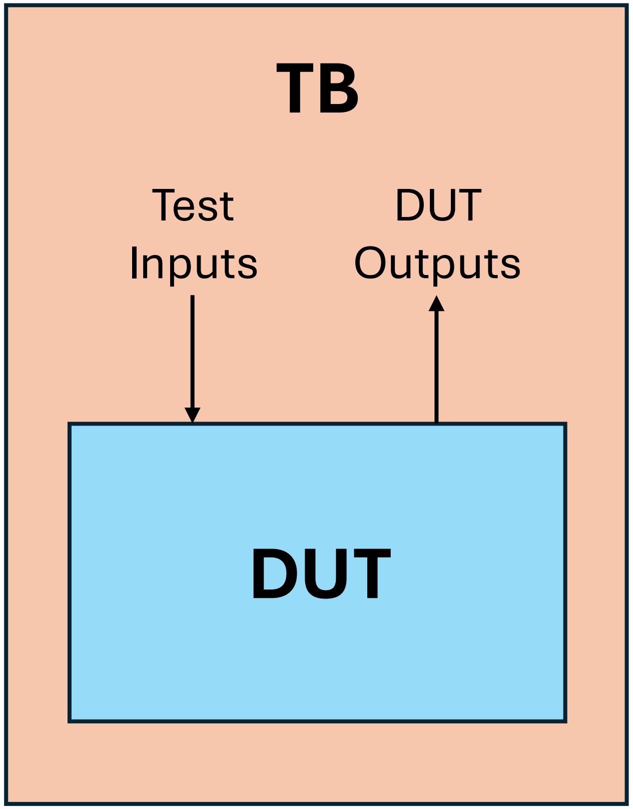
Generate Blocks

- Circuits can instantiate many replicas of subcomponents
- Generate blocks allow you to instantiate modules in a loop to avoid doing it manually
- Keep in mind …
    - For loops in procedural blocks → describe the behavior of the module
    - Generate for loops → instantiate sub-modules

```
module adder (
     input  [7:0] a,
     input  [7:0] b,
     output [8:0] out
);
assign out = a + b;
endmodule

/**********************************/

module vector_add #(
     parameter N = 128
)(
     input  [7:0] a [0:N-1],
     input  [7:0] b [0:N-1],
     output [8:0] res [0:N-1]
);

genvar i;
generate
for (i = 0; i < N; i = i + 1)
begin: gen_adders
     adder add_inst(
       .a(a[i]), .b(b[i]), .out(res[i])
     );
end
endgenerate

endmodule
```

Implementing Simulation Testbenches

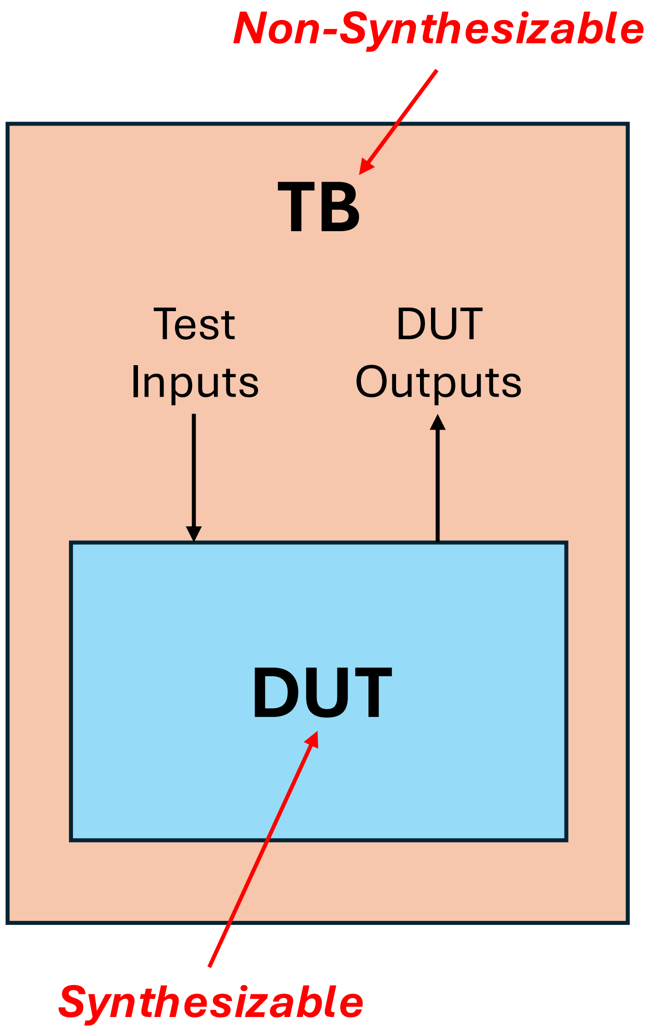
- A simulation testbench is a wrapper around the module to be tested (design under test or DUT):
    - Instantiates DUT as a subcomponent
    - Provides test inputs & monitors the DUT outputs
    - (Optional) Compares to reference “golden” outputs to verify correctness



Implementing Simulation Testbenches

- A simulation testbench is a wrapper around the module to be tested (design under test or DUT):
  - Instantiates DUT as a subcomponent
  - Provides test inputs & monitors the DUT outputs
  - (Optional) Compares to reference "golden" outputs to verify correctness

- The testbench is not synthesized to hardware – used to verify & simulate a synthesizable DUT



Simple Testbench Example

```
`timescale <unit>/<precision>
```

- Directive that specifies the time unit and precision used for simulation
  - Delays and simulation time are measured in time unit
  - Delay values are rounded based on the time precision
- Quantities can be 1, 10, or 100
- Units can range from seconds (s) to femtoseconds (fs)

```
`timescale 1ns/1ps

module adder8b_tb ();

logic [7:0] in1, in2;
logic [8:0] out;

adder8b dut (
     .in1(in1),
     .in2(in2),
     .out(out)
);

initial begin
     $monitor($time, “ns: in1=%d, in2=%d,
          out=%d”, in1, in2, out);
     in1 = 8’d3; in2 = 8’d2;
     #10 in1 = 8’d10; in2 = 8’d34;
     #10 in1 = 8’d22; in2 = 8’d17;
     #10 in1 = 8’d13; in2 = 8’d85;
     #10 in1 = 8’d74; in2 = 8’d44;
     #10 $stop;
end

endmodule
```

```
module adder8b (
  input  [7:0] in1,
  input  [7:0] in2,
  output [8:0] out
);
```

Simple Testbench Example

- A testbench typically has no external inputs or outputs (i.e., no port list)

- Self-contained testbenches
  - Generate test inputs internally and supply them to the DUT
  - Receive DUT outputs (can be investigated by “print” statements or in waveforms)
  - Can compare DUT outputs to reference “golden” solutions

```
`timescale 1ns/1ps

module adder8b_tb ();

logic [7:0] in1, in2;
logic [8:0] out;

adder8b dut (
     .in1(in1),
     .in2(in2),
     .out(out)
);

initial begin
     $monitor($time, “ns: in1=%d, in2=%d,
          out=%d”, in1, in2, out);
     in1 = 8’d3; in2 = 8’d2;
     #10 in1 = 8’d10; in2 = 8’d34;
     #10 in1 = 8’d22; in2 = 8’d17;
     #10 in1 = 8’d13; in2 = 8’d85;
     #10 in1 = 8’d74; in2 = 8’d44;
     #10 $stop;
end

endmodule
```

```
module adder8b (
  input  [7:0] in1,
  input  [7:0] in2,
  output [8:0] out
);
```

Simple Testbench Example

- Declare signals corresponding to DUT inputs and outputs

- Instantiate DUT and connect its ports to the declared signals

```
`timescale 1ns/1ps

module adder8b_tb ();

logic [7:0] in1, in2;
logic [8:0] out;

adder8b dut (
     .in1(in1),
     .in2(in2),
     .out(out)
);

initial begin
     $monitor($time, “ns: in1=%d, in2=%d,
          out=%d”, in1, in2, out);
     in1 = 8’d3; in2 = 8’d2;
     #10 in1 = 8’d10; in2 = 8’d34;
     #10 in1 = 8’d22; in2 = 8’d17;
     #10 in1 = 8’d13; in2 = 8’d85;
     #10 in1 = 8’d74; in2 = 8’d44;
     #10 $stop;
end

endmodule
```

```
module adder8b (
  input  [7:0] in1,
  input  [7:0] in2,
  output [8:0] out
);
```

Simple Testbench Example

- Set test input values in a procedural block (e.g., initial)

```
`timescale 1ns/1ps

module adder8b_tb ();

logic [7:0] in1, in2;
logic [8:0] out;

adder8b dut (
     .in1(in1),
     .in2(in2),
     .out(out)
);

initial begin
     $monitor($time, “ns: in1=%d, in2=%d,
          out=%d”, in1, in2, out);
     in1 = 8’d3; in2 = 8’d2;
     #10 in1 = 8’d10; in2 = 8’d34;
     #10 in1 = 8’d22; in2 = 8’d17;
     #10 in1 = 8’d13; in2 = 8’d85;
     #10 in1 = 8’d74; in2 = 8’d44;
     #10 $stop;
end

endmodule
```

```
module adder8b (
  input  [7:0] in1,
  input  [7:0] in2,
  output [8:0] out
);
```

Simple Testbench Example

- Set test input values in a procedural block (e.g., initial)
- Use delay statements to specify when to set input signals to certain values
  - Delay statements are in the units specified by timescale directive
  - In this example, #10 means 10 units of 1ns → 10ns
  - Delay statements are not synthesizable – only used for modeling and simulation

```
module adder8b (
  input  [7:0] in1,
  input  [7:0] in2,
  output [8:0] out
);
```

```
`timescale 1ns/1ps

module adder8b_tb ();

logic [7:0] in1, in2;
logic [8:0] out;

adder8b dut (
     .in1(in1),
     .in2(in2),
     .out(out)
);

initial begin
     $monitor($time, “ns: in1=%d, in2=%d,
          out=%d”, in1, in2, out);
     in1 = 8’d3; in2 = 8’d2;
     #10 in1 = 8’d10; in2 = 8’d34;
     #10 in1 = 8’d22; in2 = 8’d17;
     #10 in1 = 8’d13; in2 = 8’d85;
     #10 in1 = 8’d74; in2 = 8’d44;
     #10 $stop;
end

endmodule
```

Simple Testbench Example

- Some SystemVerilog system functions can be useful for simulation
  - $monitor: continuously monitors the signals in its argument list and outputs a message when any of their values changes
  - $display: outputs a message at any time (e.g., output simulation time during the execution of a procedural block)
  - $time: outputs the current simulation time
  - $stop: pauses the simulation (analogous to a breakpoint)
  - $finish: terminates simulation

```
module adder8b (
  input  [7:0] in1,
  input  [7:0] in2,
  output [8:0] out
);
```

```
`timescale 1ns/1ps

module adder8b_tb ();

logic [7:0] in1, in2;
logic [8:0] out;

adder8b dut (
     .in1(in1),
     .in2(in2),
     .out(out)
);

initial begin
     $monitor($time, “ns: in1=%d, in2=%d,
          out=%d”, in1, in2, out);
     in1 = 8’d3; in2 = 8’d2;
     #10 in1 = 8’d10; in2 = 8’d34;
     #10 in1 = 8’d22; in2 = 8’d17;
     #10 in1 = 8’d13; in2 = 8’d85;
     #10 in1 = 8’d74; in2 = 8’d44;
     #10 $stop;
end

endmodule
```

Generating a Clock Signal in the Testbench

- When simulating sequential circuits, the testbench needs to generate a periodic clock signal for the DUT
- Clock can be generated in an independent `initial` block
  - Set clock signal initially to 0
  - Use `forever` to repeat a behavior until end of time
  - The repeated behavior is to wait half a clock period & negate the clock signal → 50% duty cycle
- Remember this is generating a clock signal for simulation
  - Does not synthesize hardware that generates a clock

```
`timescale 1ns/1ps

module sequential_module_tb ();

parameter CLK_PERIOD = 2; // 2ns clock

// Declare logic signals
logic clk;
...

sequential_module dut (
     // Connect DUT ports
     .clk(clk),
     ...
);

initial begin
     clk = 1’b0;
     forever #(CLK_PERIOD/2) clk = ~clk;
end

initial begin
     // Testbench logic goes here
end

endmodule
```

How does the Simulator Work?

- SystemVerilog simulation is event-driven

How does the Simulator Work?

- SystemVerilog simulation is event-driven
- Two types of events:
  - **Evaluation** events – RHS expressions of assignments are evaluated
  - **Update** events – LHS signals of assignments are updated

How does the Simulator Work?

- SystemVerilog simulation is event-driven
- Two types of events:
  - **Evaluation** events – RHS expressions of assignments are evaluated
  - **Update** events – LHS signals of assignments are updated
- Update events can happen at different times
  - Right after evaluation events (e.g., blocking & continuous assignments)
  - At the end of a physical time step (e.g., non-blocking assignments)

How does the Simulator Work?

- SystemVerilog simulation is event-driven
- Two types of events:
  - **Evaluation** events – RHS expressions of assignments are evaluated
  - **Update** events – LHS signals of assignments are updated
- Update events can happen at different times
  - Right after evaluation events (e.g., blocking & continuous assignments)
  - At the end of a physical time step (e.g., non-blocking assignments)
- Update events → Trigger evaluation events to be queued → Trigger update events → Trigger evaluation events → …

How does the Simulator Work?

- SystemVerilog simulation is event-driven
- Two types of events:
  - **Evaluation** events – RHS expressions of assignments are evaluated
  - **Update** events – LHS signals of assignments are updated
- Update events can happen at different times
  - Right after evaluation events (e.g., blocking & continuous assignments)
  - At the end of a physical time step (e.g., non-blocking assignments)
- Update events → Trigger evaluation events to be queued → Trigger update events → Trigger evaluation events → …
- Events enter a priority queue ("stratified event queue") to be processed:
  - Earliest event first
  - Then first come first serve

How does the Simulator Work?

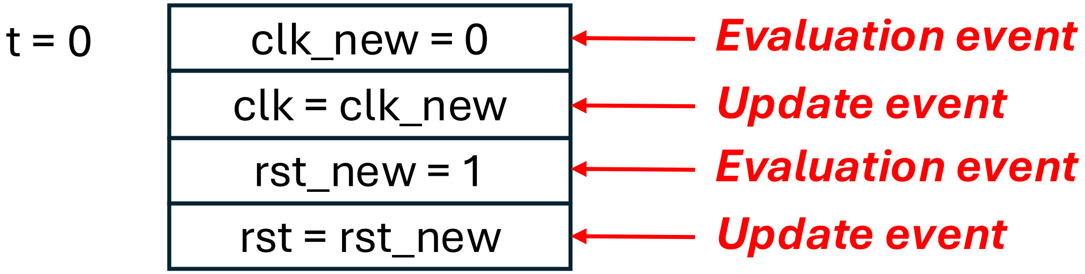
- SystemVerilog simulation is event-driven
- Two types of events:
  - **Evaluation** events – RHS expressions of assignments are evaluated
  - **Update** events – LHS signals of assignments are updated
- Update events can happen at different times
  - Right after evaluation events (e.g., blocking & continuous assignments)
  - At the end of a physical time step (e.g., non-blocking assignments)
- Update events → Trigger evaluation events to be queued → Trigger update events → Trigger evaluation events → …
- Events enter a priority queue ("stratified event queue") to be processed:
  - Earliest event first
  - Then first come first serve
- For concurrent blocks, events can be queued in any order

Simulation Example

```
module swap (
     input  clk,
     input  rst,
     output logic out1,
     output logic out2
);

always_ff @ (posedge clk)
     if (rst) out1 = 0;
     else out1 = out2;

always_ff @ (posedge clk)
     if (rst) out2 = 1;
     else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
     clk = 1’b0;
     forever #1 clk = ~clk;
end

initial begin
     rst = 1;
     #2 rst = 0;
end

endmodule
```

Simulation Example



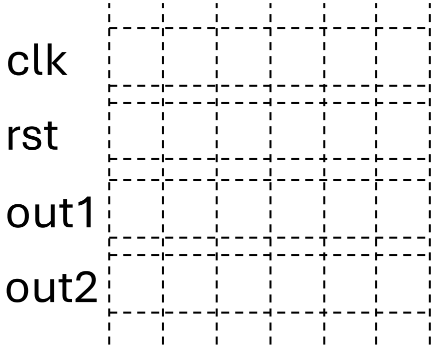
t = 0

clk_new = 0 ← *Evaluation event*

clk = clk_new ← *Update event*

rst_new = 1 ← *Evaluation event*

rst = rst_new ← *Update event*



Signal	Value	_new
clk	X	
rst	X	
out1	X	
out2	X	

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else  out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else  out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 0

clk_new = 0

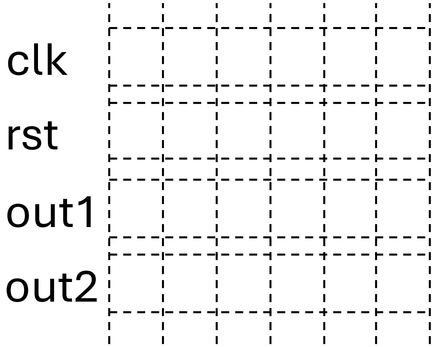
clk = clk_new

rst_new = 1

rst = rst_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	X	0
rst	X	
out1	X	
out2	X	

Simulation Example

t = 0

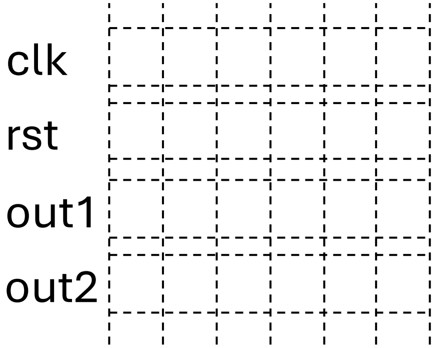
clk = clk_new

rst_new = 1

rst = rst_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	0	0
rst	X	
out1	X	
out2	X	

Simulation Example

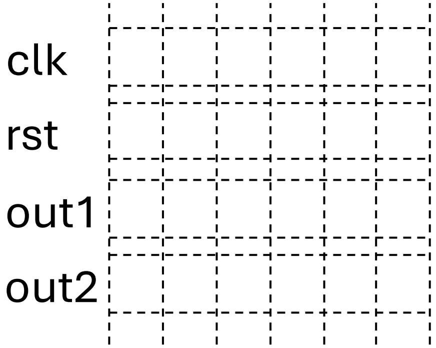
t = 0

rst_new = 1

rst = rst_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

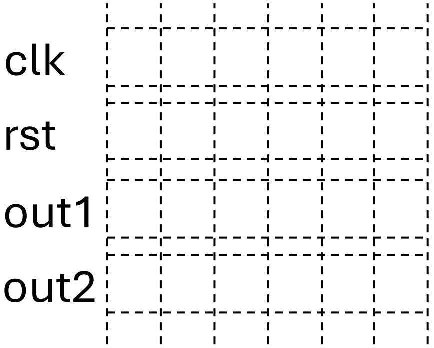
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	0	0
rst	X	1
out1	X	
out2	X	

Simulation Example

t = 0 rst = rst_new



Signal	Value	_new
clk	0	0
rst	1	1
out1	X	
out2	X	

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

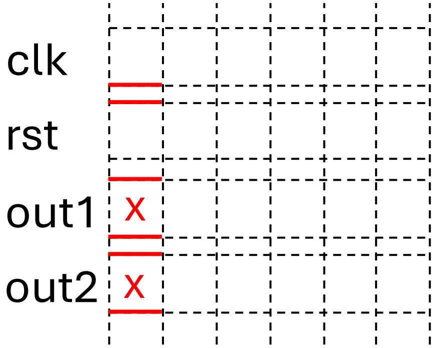
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 0

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



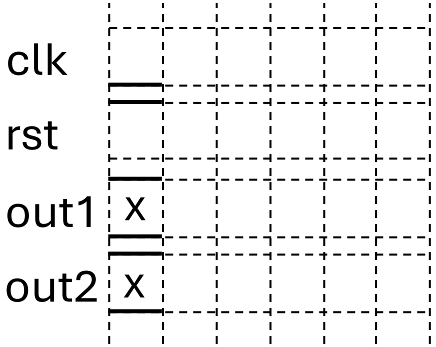
Signal	Value	_new
clk	0	0
rst	1	1
out1	X	
out2	X	

Simulation Example

t = 1

clk_new = ~clk

clk = clk_new



Signal	Value	_new
clk	0	0
rst	1	1
out1	X	
out2	X	

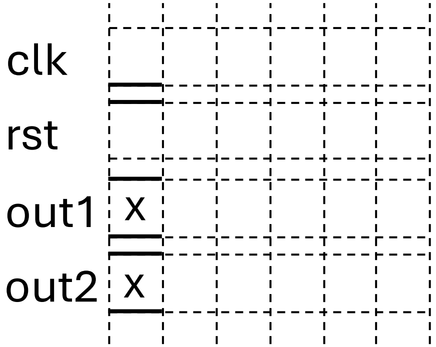
```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 1

clk_new = ~clk
clk = clk_new



Signal	Value	_new
clk	0	1
rst	1	1
out1	X	
out2	X	

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

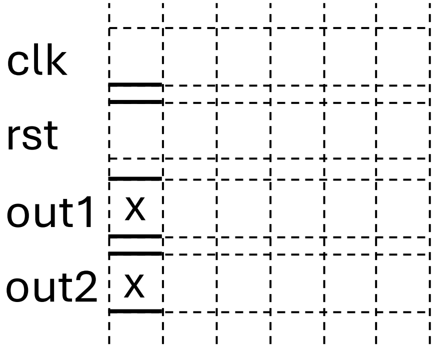
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 1

clk = clk_new

Clock posedge queues more events



Signal	Value	_new
clk	**1**	1
rst	1	1
out1	X	
out2	X	

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 1

out1_new = 0

out1 = out1_new

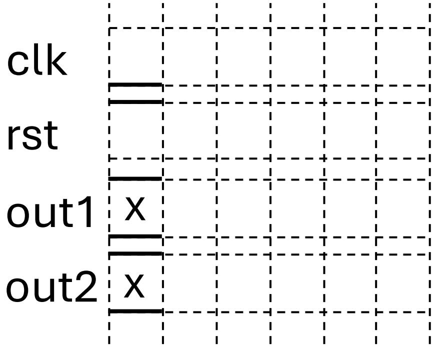
out2_new = 1

out2 = out2_new

Clock posedge queues more events

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else  out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else  out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	1	1
rst	1	1
out1	X	
out2	X	

Simulation Example

t = 1

out1_new = 0

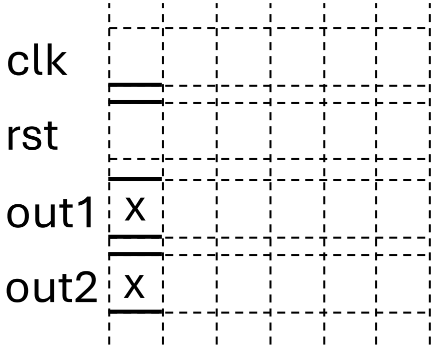
out1 = out1_new

out2_new = 1

out2 = out2_new

```
module swap (
        input  clk,
        input  rst,
        output logic out1,
        output logic out2
);

always_ff @ (posedge clk)
        if (rst) out1 = 0;
        else out1 = out2;

always_ff @ (posedge clk)
        if (rst) out2 = 1;
        else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
        rst = 1;
        #2 rst = 0;
end

endmodule
```

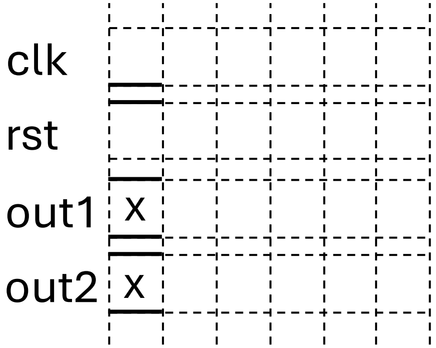


Signal	Value	_new
clk	1	1
rst	1	1
out1	X	**0**
out2	X	

Simulation Example

t = 1

- out1 = out1_new
- out2_new = 1
- out2 = out2_new



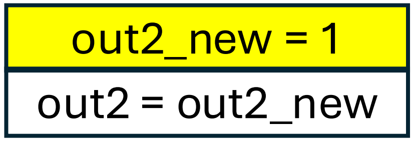
Signal	Value	_new
clk	1	1
rst	1	1
out1	**0**	0
out2	X	

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

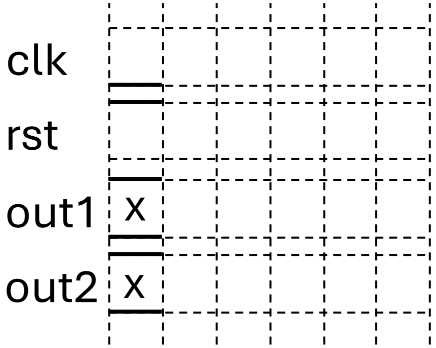
Simulation Example

t = 1


out2_new = 1

out2 = out2_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```



Signal	Value	_new
clk	1	1
rst	1	1
out1	0	0
out2	X	**1**

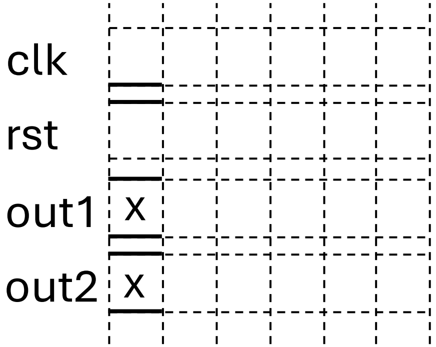
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 1   out2 = out2_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

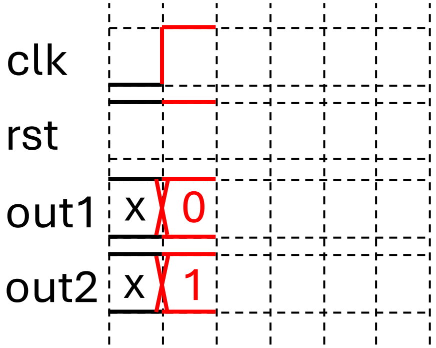
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	1	1
rst	1	1
out1	0	0
out2	1	1

Simulation Example

t = 1



Signal	Value	_new
clk	1	1
rst	1	1
out1	0	0
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

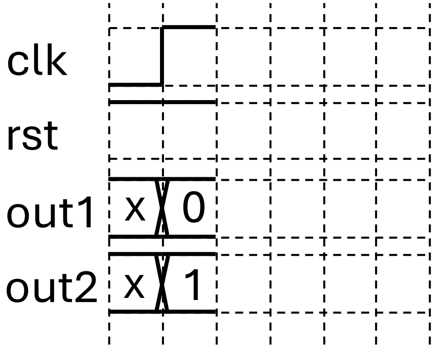
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 2

clk_new = ~clk
clk = clk_new
rst_new = 0
rst = rst_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```



Signal	Value	_new
clk	1	1
rst	1	1
out1	0	0
out2	1	1

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

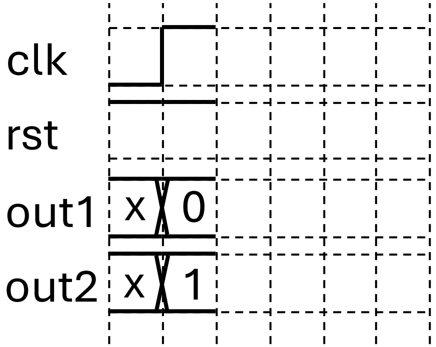
t = 2

clk_new = ~clk

clk = clk_new

rst_new = 0

rst = rst_new



Signal	Value	_new
clk	1	**0**
rst	1	1
out1	0	0
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else  out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else  out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 2

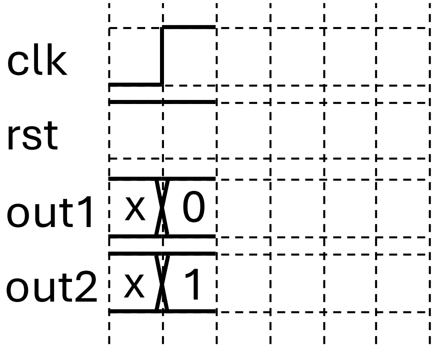
clk = clk_new

rst_new = 0

rst = rst_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



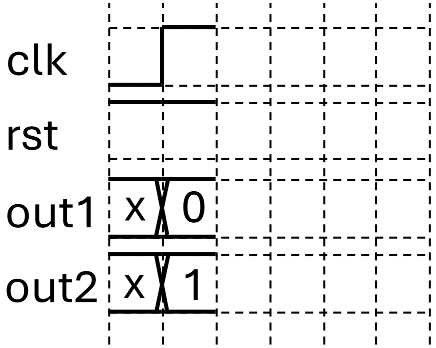
Signal	Value	_new
clk	0	0
rst	1	1
out1	0	0
out2	1	1

Simulation Example

t = 2

rst_new = 0

rst = rst_new



Signal	Value	_new
clk	0	0
rst	1	0
out1	0	0
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

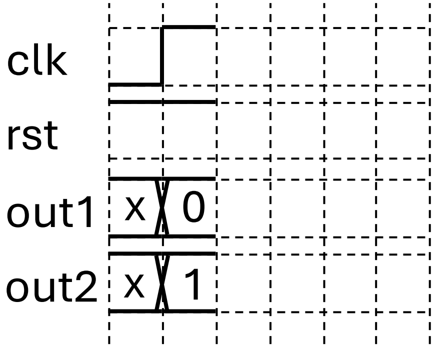
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 2   rst = rst_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else  out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else  out2 = out1;

endmodule
```

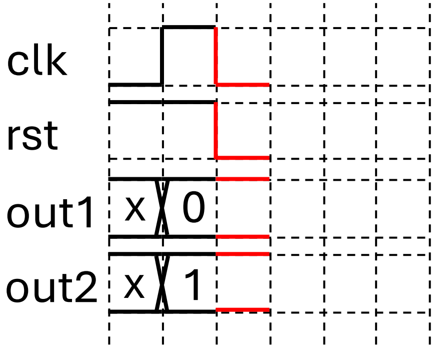
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	0	0
rst	0	0
out1	0	0
out2	1	1

Simulation Example

t = 2



Signal	Value	_new
clk	0	0
rst	0	0
out1	0	0
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

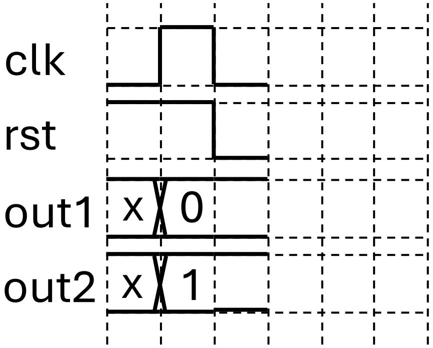
Simulation Example

t = 3

clk_new = ~clk
clk = clk_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	0	0
rst	0	0
out1	0	0
out2	1	1

Simulation Example

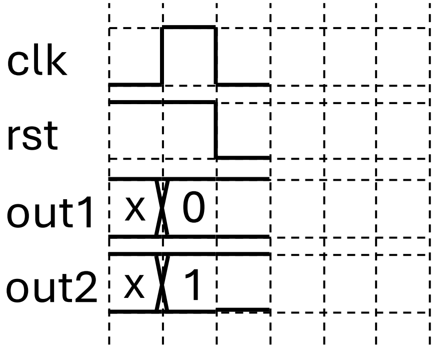
t = 3

clk_new = ~clk

clk = clk_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else  out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else  out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	0	1
rst	0	0
out1	0	0
out2	1	1

Simulation Example

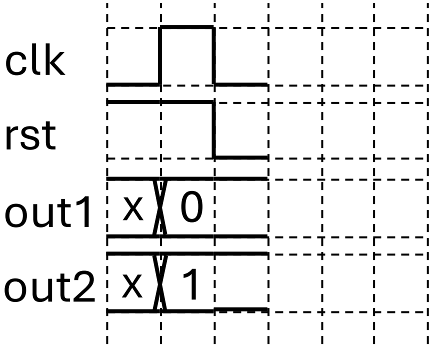
t = 3

clk = clk_new

Clock posedge queues more events

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



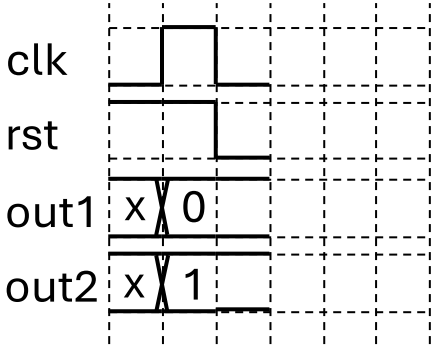
Signal	Value	_new
clk	1	1
rst	0	0
out1	0	0
out2	1	1

Simulation Example

t = 3

out1_new = out2
out1 = out1_new
out2_new = out1
out2 = out2_new

Clock posedge queues more events



Signal	Value	_new
clk	1	1
rst	0	0
out1	0	0
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else  out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else  out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 3

out1_new = out2

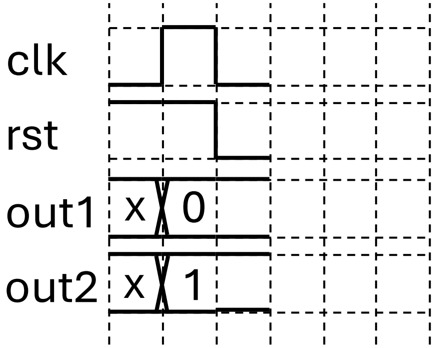
out1 = out1_new

out2_new = out1

out2 = out2_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else  out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else  out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	1	1
rst	0	0
out1	0	1
out2	1	1

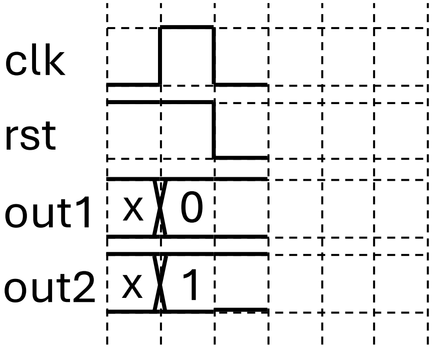
Simulation Example

t = 3

out1 = out1_new

out2_new = out1

out2 = out2_new



Signal	Value	_new
clk	1	1
rst	0	0
out1	1	1
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else  out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else  out2 = out1;

endmodule
```

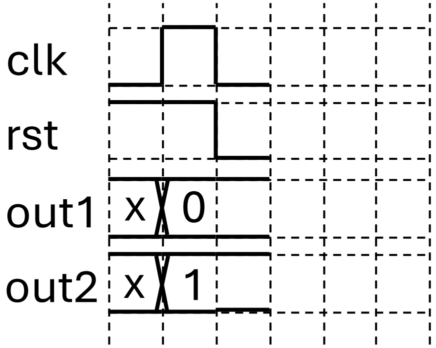
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 3

out2_new = out1

out2 = out2_new



Signal	Value	_new
clk	1	1
rst	0	0
out1	1	1
out2	1	1

```
module swap (
     input  clk,
     input  rst,
     output logic out1,
     output logic out2
);

always_ff @ (posedge clk)
     if (rst) out1 = 0;
     else  out1 = out2;

always_ff @ (posedge clk)
     if (rst) out2 = 1;
     else  out2 = out1;

endmodule
```

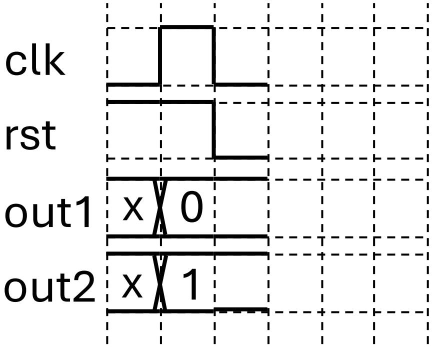
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
     clk = 1’b0;
     forever #1 clk = ~clk;
end

initial begin
     rst = 1;
     #2 rst = 0;
end

endmodule
```

Simulation Example

t = 3   out2 = out2_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else  out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else  out2 = out1;

endmodule
```

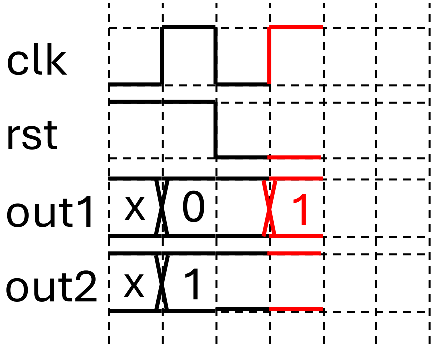
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	1	1
rst	0	0
out1	1	1
out2	1	1

Simulation Example

t = 3



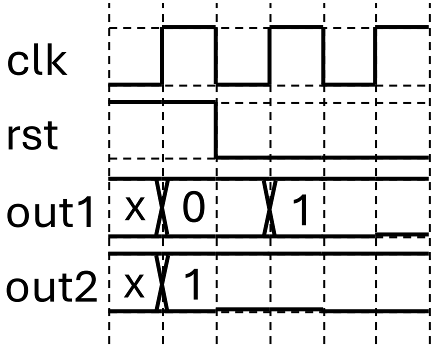
Signal	Value	_new
clk	1	1
rst	0	0
out1	1	1
out2	1	1

```
module swap (
     input  clk,
     input  rst,
     output logic out1,
     output logic out2
);

always_ff @ (posedge clk)
     if (rst) out1 = 0;
     else  out1 = out2;

always_ff @ (posedge clk)
     if (rst) out2 = 1;
     else  out2 = out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
     clk = 1’b0;
     forever #1 clk = ~clk;
end

initial begin
     rst = 1;
     #2 rst = 0;
end

endmodule
```

Simulation Example

t = 5



Signal	Value	_new
clk	1	1
rst	0	0
out1	1	1
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 = 0;
      else out1 = out2;

always_ff @ (posedge clk)
      if (rst) out2 = 1;
      else out2 = out1;

endmodule
```

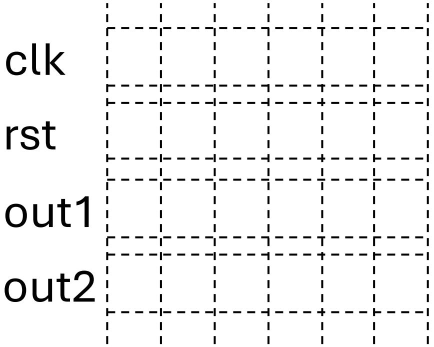
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

Now let’s see what happens when we use non-blocking assignments

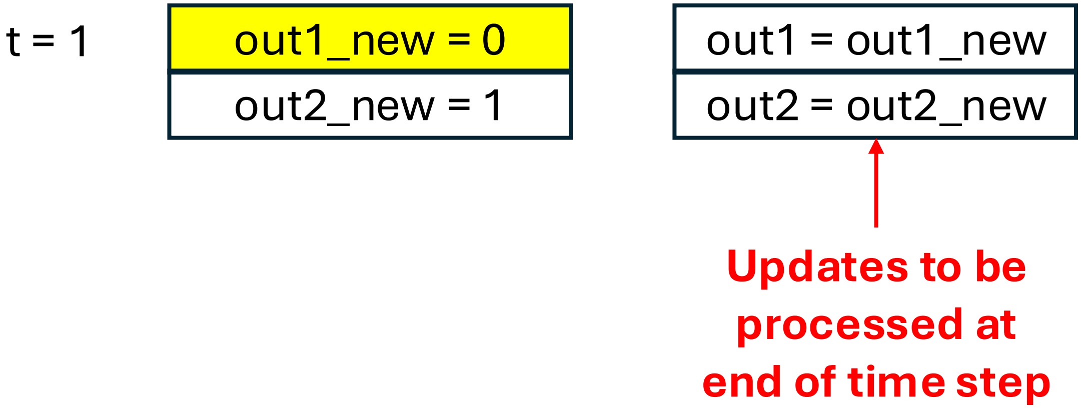
```
module swap (
    input  clk,
    input  rst,
    output logic out1,
    output logic out2
);

always_ff @ (posedge clk)
    if (rst) out1 <= 0;
    else  out1 <= out2;

always_ff @ (posedge clk)
    if (rst) out2 <= 1;
    else  out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
    clk = 1’b0;
    forever #1 clk = ~clk;
end

initial begin
    rst = 1;
    #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	X	
rst	X	
out1	X	
out2	X	

Simulation Example



t = 1

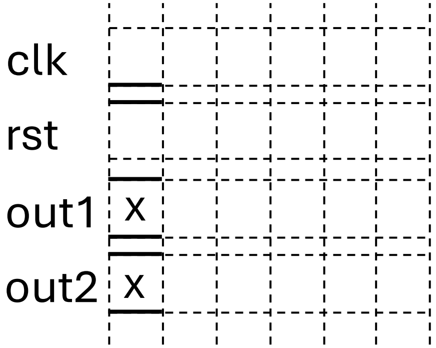
out1_new = 0

out2_new = 1

out1 = out1_new

out2 = out2_new

Updates to be processed at end of time step

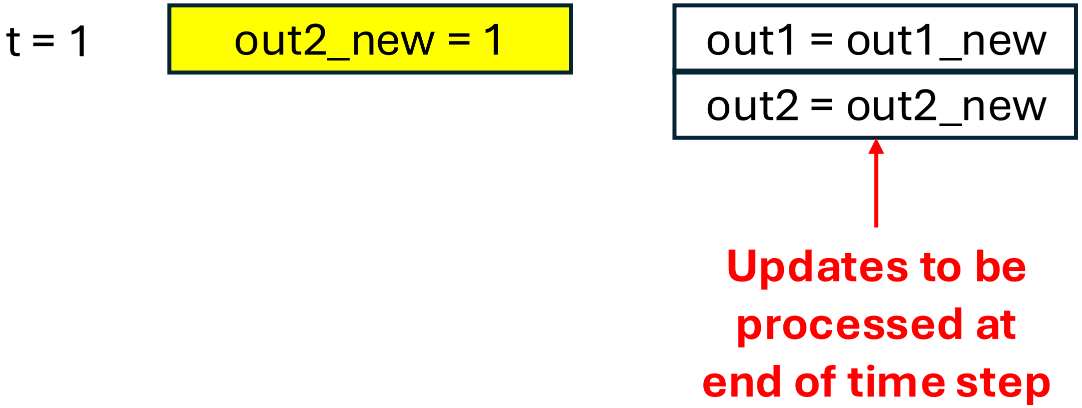


Signal	Value	_new
clk	1	1
rst	1	1
out1	X	**0**
out2	X	

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else  out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else  out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example



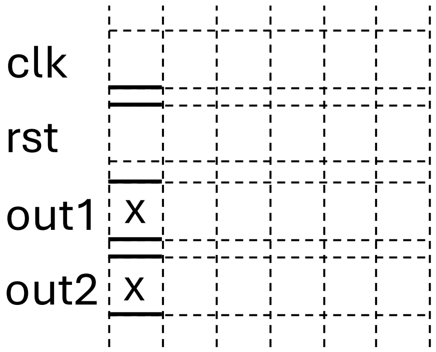
t = 1

out2_new = 1

out1 = out1_new

out2 = out2_new

Updates to be processed at end of time step



Signal	Value	_new
clk	1	1
rst	1	1
out1	X	0
out2	X	**1**

```
module swap (
      input   clk,
      input   rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else  out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else  out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 1

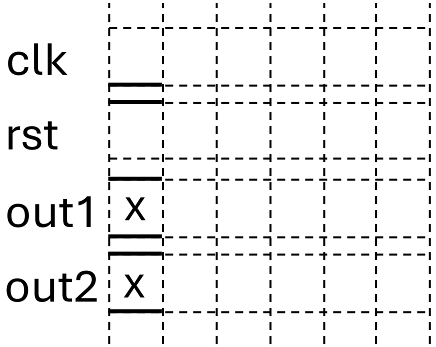
out1 = out1_new

out2 = out2_new

Updates to be processed at end of time step

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else  out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else  out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



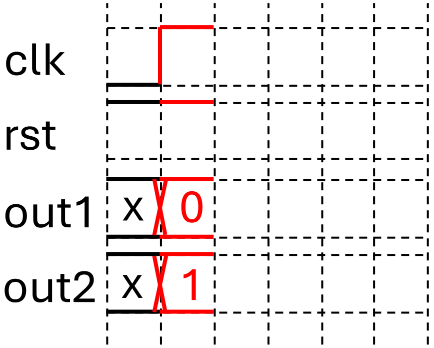
Signal	Value	_new
clk	1	1
rst	1	1
out1	**0**	0
out2	**1**	1

Simulation Example

t = 1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else out2 <= out1;

endmodule
```

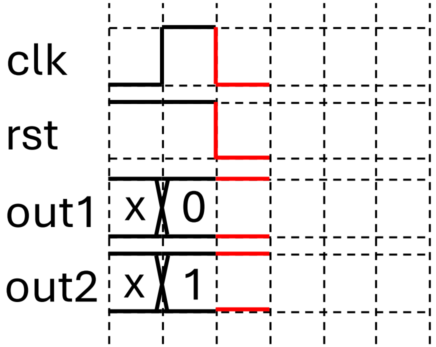
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	1	1
rst	1	1
out1	0	0
out2	1	1

Simulation Example

t = 2



Signal	Value	_new
clk	0	0
rst	0	0
out1	0	0
out2	1	1

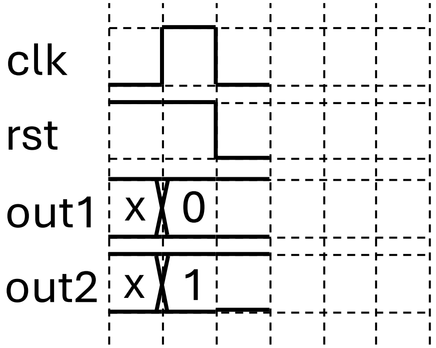
```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else  out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else  out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 3

clk_new = ~clk
clk = clk_new



Signal	Value	_new
clk	0	0
rst	0	0
out1	0	0
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

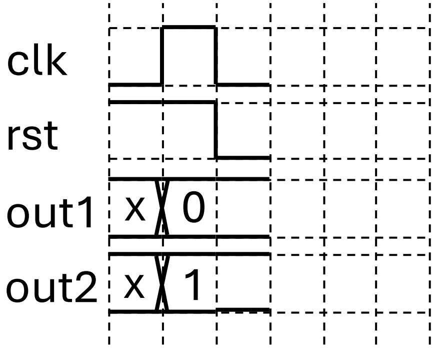
t = 3

clk_new = ~clk

clk = clk_new

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



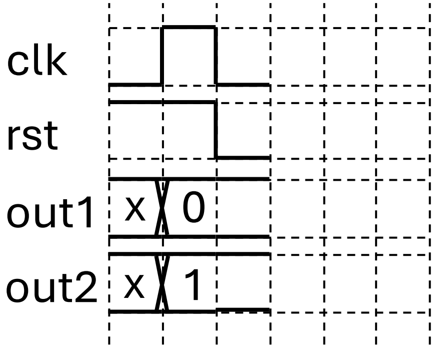
Signal	Value	_new
clk	0	1
rst	0	0
out1	0	0
out2	1	1

Simulation Example

t = 3

clk = clk_new

Clock posedge queues more events



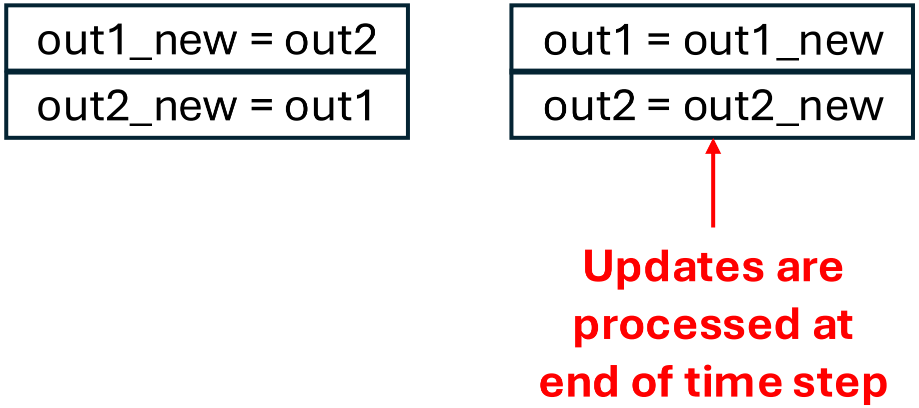
Signal	Value	_new
clk	1	1
rst	0	0
out1	0	0
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 3


out1_new = out2

out2_new = out1

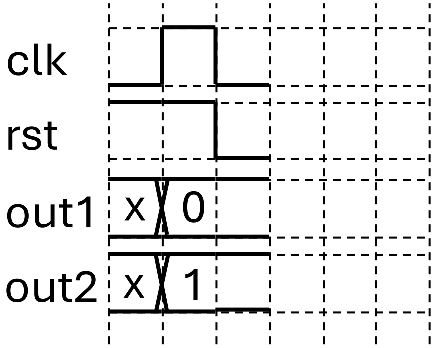
out1 = out1_new

out2 = out2_new

Updates are processed at end of time step

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else  out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else  out2 <= out1;

endmodule
```

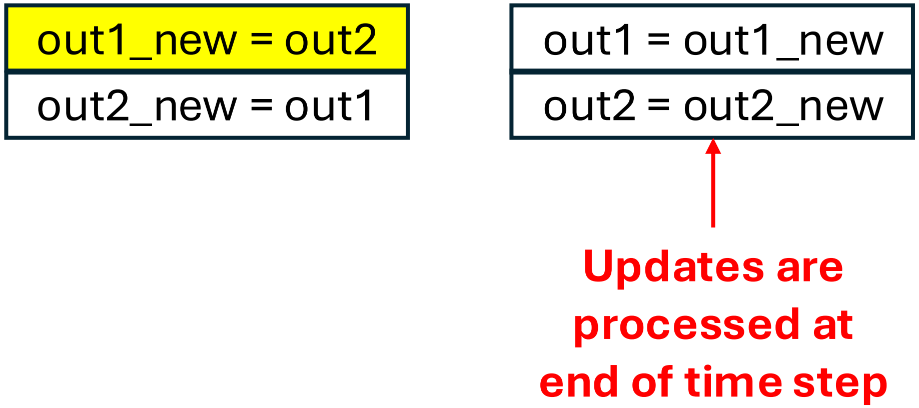
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	1	1
rst	0	0
out1	0	0
out2	1	1

Simulation Example

t = 3



out1_new = out2

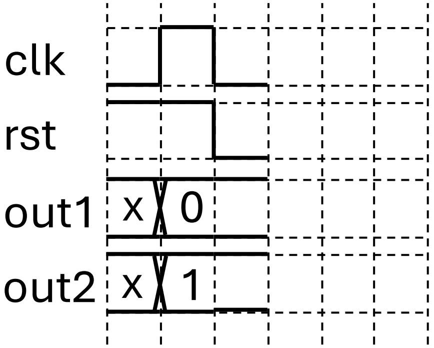
out2_new = out1

out1 = out1_new

out2 = out2_new

Updates are processed at end of time step

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else  out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else  out2 <= out1;

endmodule
```



Signal	Value	_new
clk	1	1
rst	0	0
out1	0	**1**
out2	1	1

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

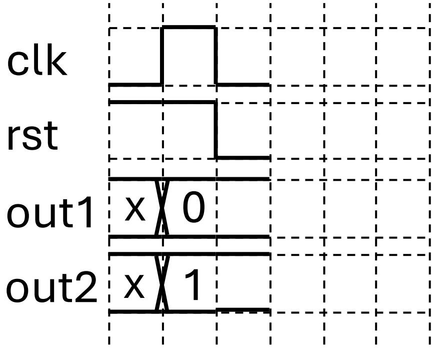
t = 3

out2_new = out1

out1 = out1_new

out2 = out2_new

Updates are processed at end of time step



Signal	Value	_new
clk	1	1
rst	0	0
out1	0	1
out2	1	**0**

```
module swap (
      input   clk,
      input   rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else  out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else  out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 3

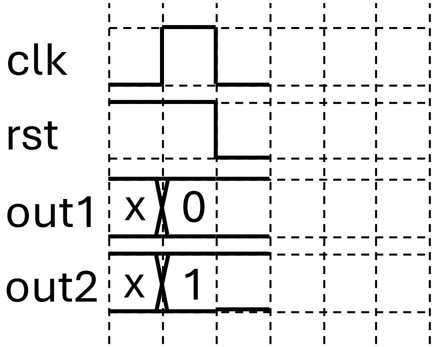
out1 = out1_new

out2 = out2_new

Updates are processed at end of time step

```
module swap (
     input  clk,
     input  rst,
     output logic out1,
     output logic out2
);

always_ff @ (posedge clk)
     if (rst) out1 <= 0;
     else  out1 <= out2;

always_ff @ (posedge clk)
     if (rst) out2 <= 1;
     else  out2 <= out1;

endmodule
```

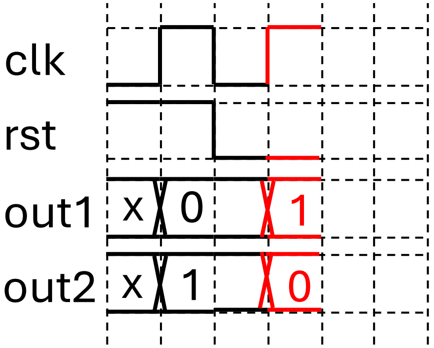
```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
     rst = 1;
     #2 rst = 0;
end

endmodule
```



Signal	Value	_new
clk	1	1
rst	0	0
out1	1	1
out2	0	0

Simulation Example

t = 3



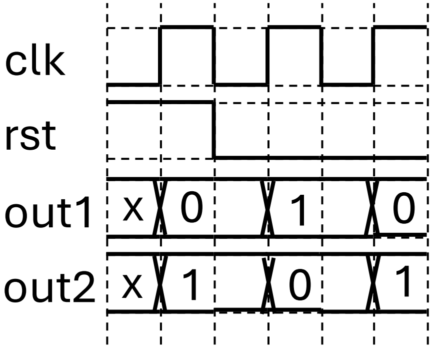
Signal	Value	_new
clk	1	1
rst	0	0
out1	1	1
out2	0	0

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else  out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else  out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

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

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```

Simulation Example

t = 5



Signal	Value	_new
clk	1	1
rst	0	0
out1	0	0
out2	1	1

```
module swap (
      input  clk,
      input  rst,
      output logic out1,
      output logic out2
);

always_ff @ (posedge clk)
      if (rst) out1 <= 0;
      else out1 <= out2;

always_ff @ (posedge clk)
      if (rst) out2 <= 1;
      else out2 <= out1;

endmodule
```

```
module swap_tb ();
logic clk, rst, out1, out2;
swap dut(clk, rst, out1, out2);

initial begin
      clk = 1’b0;
      forever #1 clk = ~clk;
end

initial begin
      rst = 1;
      #2 rst = 0;
end

endmodule
```