Embedded Systems (digital design)

Verilog and SystemVerilog

- basics on sequential design and testbenches -

Prof. Davide Zoni (davide.zoni@polimi.it)

2022

Non-blocking assignment (<=)

- **(1)** model sequential logic and **(2)** traditional Verilog testbenches -

Non-blocking VS Blocking Assignments

Beginning with a=2, b=3

```
...
logic a[3:0];
logic b[3:0];
logic c[3:0];
always@(posedge clk)
begin
  a <= b;
  c <= a;
end
...
```

Afterwards a=3, c=2;
[non-blocking (<=)]

```
...
logic a[3:0];
logic b[3:0];
logic c[3:0];
always@(posedge clk)
begin
  a=b;
  c=a;
end
...
```

Afterwards a=3, c=3
[blocking (=)]

Non-blocking VS Blocking Assignments

Blocking (=) and non-blocking (<=) assignments
can ONLY be used in `always` blocks

Differences:

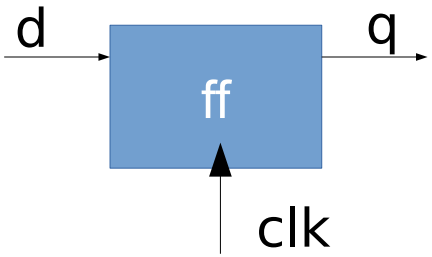
- **Blocking Assignment:** evaluated (auto-ordered) one after the other
- **Non-blocking Assignment:** within the same always block the right side of each statement is frozen and it is used to update all the left side of the corresponding statement. Then, all the variables/signals are updated in parallel at once

Practical Usage Rules:

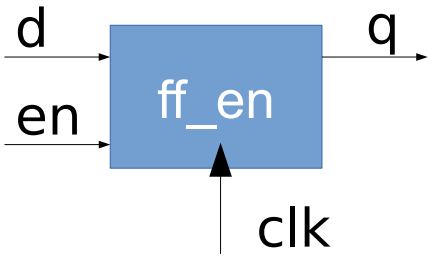
- The standard allows to mix **blocking** and **non-blocking** assignments in the same **always block**
- In this course:
  - **Blocking assignments (=)**: Combinatorial logic (design), programs (verification)
  - **Non-blocking assignments (<=)**: sequential logic (design), traditional Verilog testbenches (verification)

The flip-flop

```
module ff(input clk, input d, output reg q);
  always@(posedge clk)
  begin
     q<=d;
  end
endmodule
```

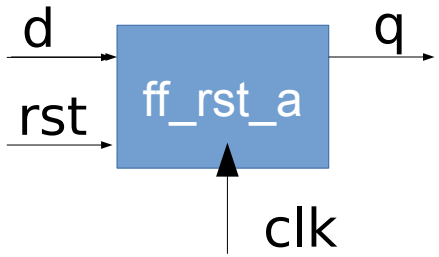



```
module ff_en(input clk, input en,
                  input d, output reg q);
  always@(posedge clk)
  begin
      if(en)
        q<=d;
  end
endmodule
```

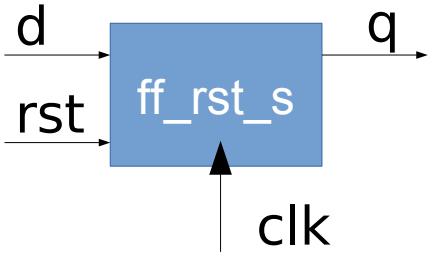


The flip-flop with (asynch/synch) reset

```
module ff_rst_a(input clk, input rst,
                input d, output reg q);
  always @(posedge clk, posedge rst) begin
    if(rst)
       q<=1'b0;
    else
       q<=d;
  end
endmodule
```

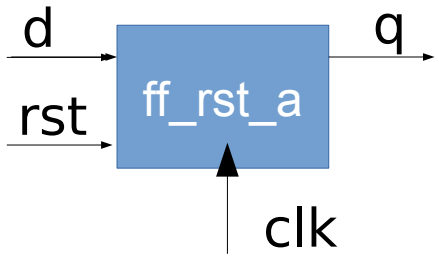



```
module ff_rst_s(input clk, input rst,
                input d, output reg q);
  always @(posedge clk) begin
      if(rst)
        q<=1'b0;
      else
        q<=d;
  end
endmodule
```

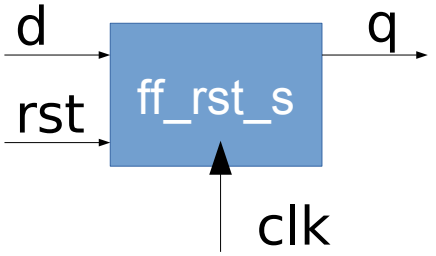


The flip-flop with (asynch/synch) reset using always_ff

```
module ff_rst_a(input clk, input rst,
                input d, output reg q);
  always_ff@(posedge clk, posedge rst) begin
    if(rst)
       q<=1'b0;
    else
       q<=d;
  end
endmodule
```




```
module ff_rst_s(input clk, input rst,
                input d, output reg q);
  always_ff@(posedge clk) begin
      if(rst)
        q<=1'b0;
      else
        q<=d;
  end
endmodule
```



The Register

(A) Standalone Component

```
module register #(parameter W=2)
(
 input clk,
 input       [W-1:0] d,
 output reg [W-1:0] q
);
  always@(posedge clk)
  begin
       q <= d;
  end
endmodule
```

(B) As made of ff modules

```
module register #(parameter W=2)
(
 input clk,
 input       [W-1:0] d1,
 output reg [W-1:0] q1
);

  ff ff0(.clk(clk), .d(d1[0]), .q(q1[0]));
  ff ff1(.clk(clk), .d(d1[1]), .q(q1[1]));
endmodule
```

Same semantic... but **(B)** is less flexible

Verification with Verilog

– basics –

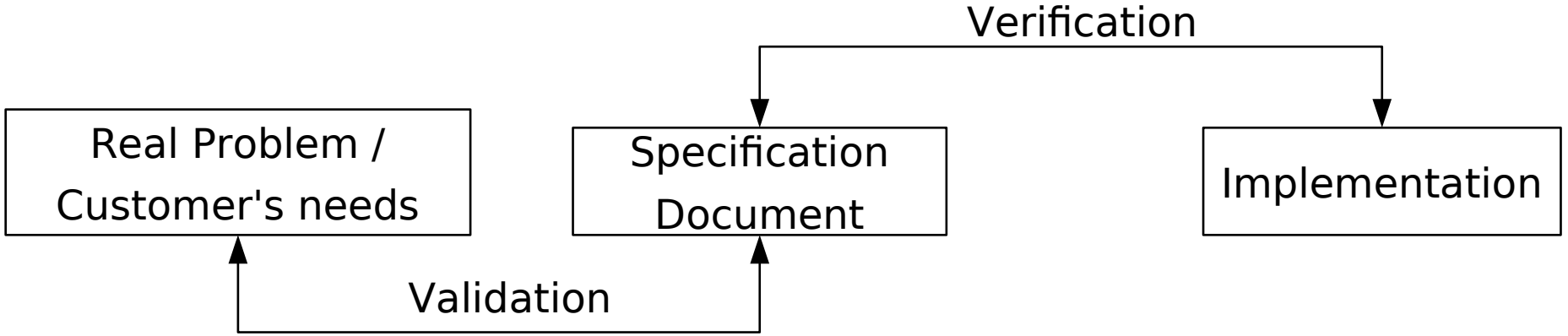
Recap: Essential Verilog for design

- **Module is the basic building block for the hardware description**
  - Input output interface (how to connect with the rest of the hierarchy)
  - Body (description of the semantic)
- **Considering the hardware design scenario, everything is parallel**
  - Always blocks and continuous assignments (their order is irrelevant)
    - NOTE: within an always block order of the statements matters!
- **Three assignments**
    - Non-procedural assignments (Outside the always block)
      - Continuous assignments (keyword **assign**)
    - Procedural assignments (Only within an always block)
      - Non-blocking assignments (**<=**)
      - Blocking assignments (**=**)
- **Information/data representation**
  - Wire (net type), reg (variable type) in the form of bit vectors
  - 4-value logic (0,1,X,Z)

Verification VS Validation

- **Verification:** check if the design satisfies the specification. Are we correctly building the system? Are we building the system right?
  - Internal process: from the early design stages till post-implementation
  - It has to comply with a set of specifications that can eventually be ambiguous or incomplete, then leading to errors/bugs

- **Validation:** it answers the question: is the design fulfill with the customer's expectations? Are we building the right system?
  - It is an external process
  - The fact that the design is doing something does not mean it is satisfactory



Functional Verification

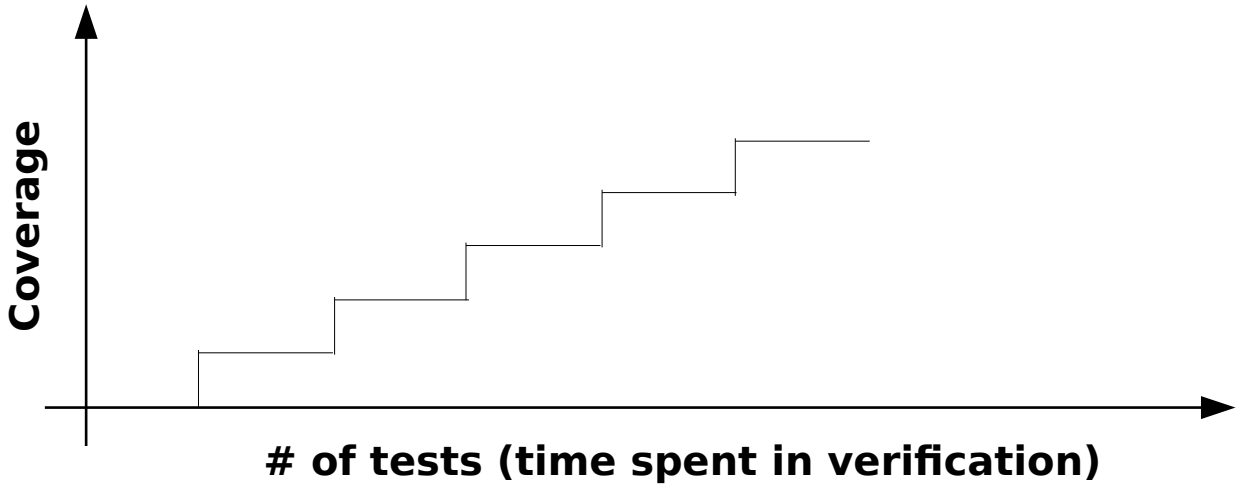
- **The goal of verification**
  - Ensure correct timing and functional behavior for the Design Under Test (DUT)
  - Usually 50% / 70% of the whole design project
  - Verification team is usually (/ should be) twice as big as the design team

- **Functional Verification**
  - Direct test
  - Constrained random verification
  - Coverage-driven constrained random verification
  - Assertion-based verification

- **Timing Verification (NOT COVERED IN THIS COURSE)**
  - Dynamic Timing Simulation (DTS)
  - Static Timing Analysis (STA)

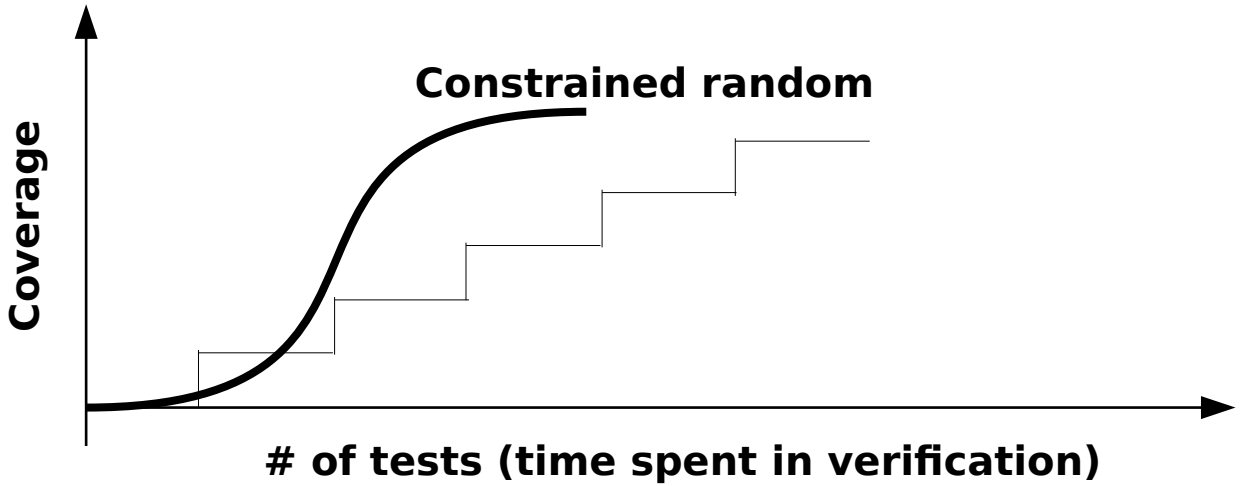
Direct testing

- **Traditional way**
  - Tests are executed one by one
  - Relatively non-flexible
  - Directly generate and pass the input vector and check results
  - Most of the time combined with monolithic verification infrastructure



Constrained random testing

- **Pros**
  - Random tests are always good (but we need to steer/constrain the test generation)
  - Reproducible (thanks to pseudo-randomness and random stability)
  - May reach points never thought by verification engineers (all corner cases)
- **Cons (of simple random)**
  - Verification engineers can lose control over the tests (use constrained-random)
  - Worst case, we never reach the verification goals (use constrained-random)



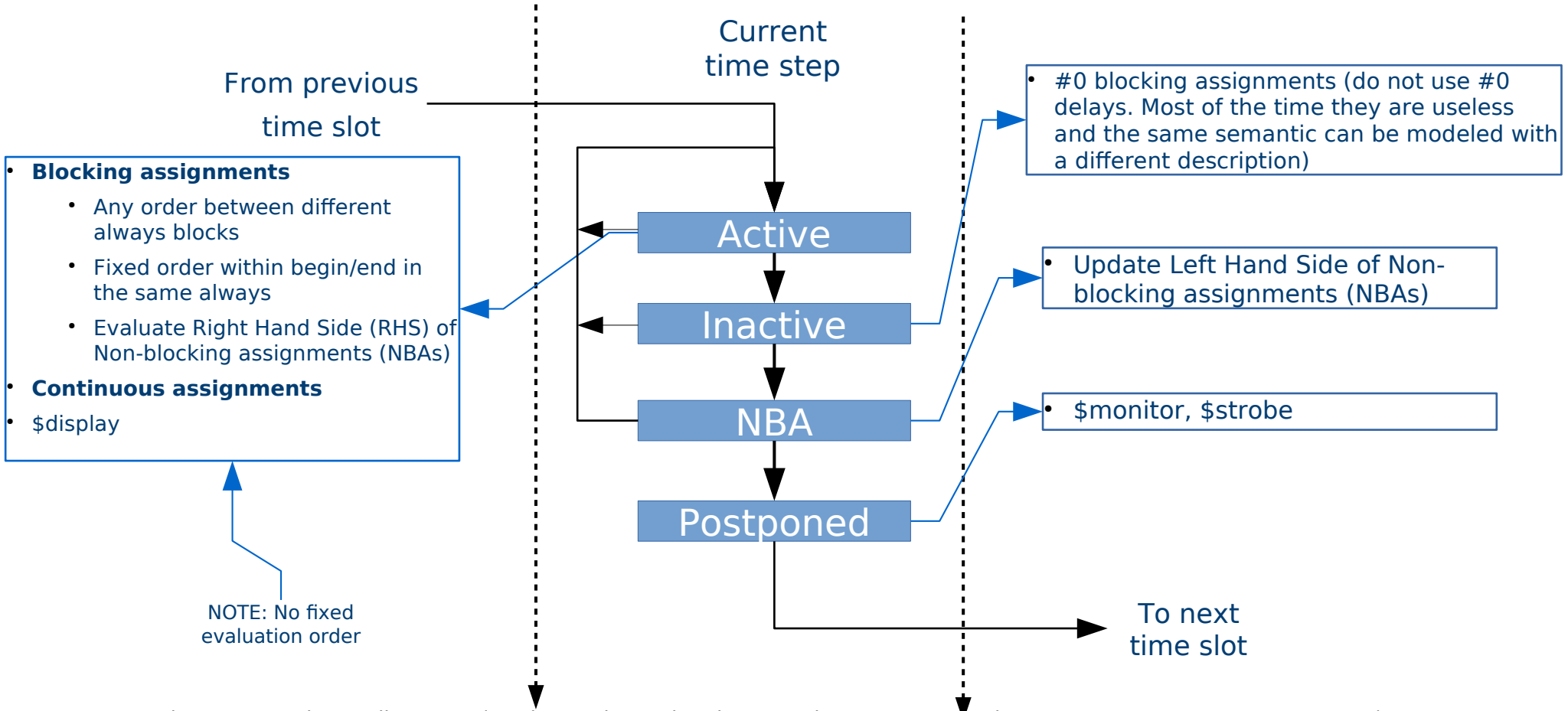
Coverage Analysis

Traditionally, **the quality of the verification** was measured with the help of code coverage tools. The code coverage reflects the “executed/visited” HDL code.

- **Structural/Code Coverage:**
  - This will give information about how many lines/statements of the HDL code are stressed, in terms of how many times they are “visited”.
  - This coverage is collected by the simulation tools
  - Almost useless to verify the semantic of the HDL description

- **Functional coverage (data and control coverage):**
  - This coverage is user defined (we need a verification plan)
  - User will define the coverage points for the functions to be covered in DUT
  - This is under user control
  - It is made of two parts: what to visit and how to generate test vectors to reach the modules/functionalities of the design to be tested.

The Simulation Flow (Verilog 2001)



Source: IEEE Std 1364-2001, i.e. Verilog-2001 (see https://ieeexplore.ieee.org/document/954909l, DOI: 10.1109/IEEESTD.2001.93352)

The time of the simulation

Simulation time is stored in specific system variables

- The time value is always scaled to the timeunit
- The time value can be stored in a variable of proper size if not directly printed

Examples

```
real time=$realtime;
integer=$stime;
integer=$time;

$display("%t",$time);
```

- **$realtime** returns a 64-bit floating point value scaled to the time precision
- **$stime** return a 32-bit unsigned integer value
  - If current value is bigger than 32-bit, the lower 32-bit only are returned
- **$time** return 64-bit integer value scaled to the timeunit, i.e., truncation of fractional timeunit delays
- The **$timeformat**(<units>, <precision>, <suffix>,<minimum field width>) allows to specify the $time, $realtime, $stime return value
  - If not specified the default values:
    - <units>:
    - <precision>: 0
    - <suffix>: null string
    - <minimum field width>:

The minimal/classic testbench module

```
`timescale [timeunit]/[timeprecision]

module minimal_tb_template();

   logic clk, in1,in2; wire out1; //in/out to the dut

   always #5 clk=~clk;  //clock generation

   initial //initial block (even more than one)
      begin
       clk <= 0; //init clk
       // init var + procedures to
       // generate input signals to dut
      end

    //dut instance
   dut #(parameter .P1(...), ...)
      dut0(.in1(in1), .in2(in2), .out1(out1));

    //tb_monitor/scoreboard to check the dut output
endmodule
```

- **timescale** directive specifies the unit of time and the precision to be used during the simulation
  - Example: `timescale 1ns/10ps
  - NOTE: Inherited from the last processed module if not specified in the .v file
- The *testbench* module has no primary inputs and outputs, but can have parameters
- Verilog comments have the same C/C++ syntax
  - // comment until the end of the line
  - /* multi-line comment */
- Input to the **dut** are **reg**
- Output from the **dut** are **wire**
- The **initial block** is a procedural block that is executed once at the simulation startup
  - Not synthesizable

Behavioral delays

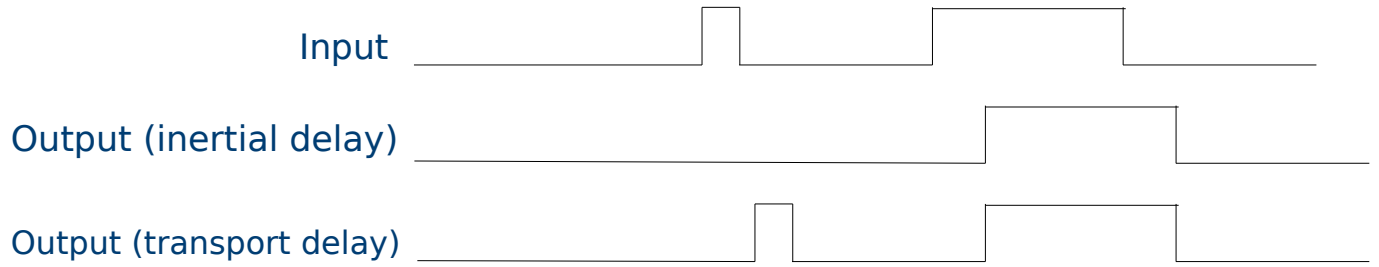
Behavioral delays

Verilog allows to model delays in the hardware description

as well as in the testbenches for behavioral simulations

- **Inertial delay model**: the inertia of a circuit node to change value. It abstractly models the RC circuit seen by the node
  - ALTERNATIVE DEFINITION: measure of the input “hold time” to get a change in the output.
  - NOTE: A pulse of duration less than the inertial delay does not contain enough energy to cause the device to switch

- **Transport delay model**: the propagation time of signals from module inputs to its outputs
  - NOTE: Time taken by the signal to propagate through the net, also known as time of flight



Behavioral Delay Models: Blocking Assignments (BA)

Adding RHS or LHS delays to model combinatorial logic is common while flawed:

- The behavioral delay is not synthesizable: it does not have a physical counterpart
- Depends on the timescale directive, functional Implications

`always@(a)` `y=~a`	No delay	• the delay is imposed by the gate from the technology library used to map the NB statement (post synthesis) • **Correct Implementation for any synthesizable Verilog description**

`always@(a)`
`#5 y=~a`
	Left-hand-side delay	• Evaluate Right-Hand-Side (RHS) then update the Left-Hand-Side (LHS) after 5 timeunit. The time of the LHS update is independent from any further change in the RHS
		  • What if an update happens at 2.5timeunits?
		  • The first value is lost **(single LHS update event is created)**

`always@(a)`
`y= #5 ~a`
	Right-hand-side delay	• Immediately evaluate the RHS, then wait 5 timeunit before updating the LHS of the statement
		  • What if an update happens at 2.5 timeunits?
		  • The first value is kept, since RHS is immediately evaluated **(single LHS update event is created)**

- **Do not use LHS or RHS delays in syntehsizable combinatorial logic**
- **Use LHS only for testbench statements to separate series of stimuli**

Behavioral Delay Models: Non-Blocking Assignments

Adding RHS or LHS delays is also possible for NBAs. While the same implications discussed for BAs are still valid, the RHS delay can be used to model the transport delay in combinatorial logic. Still not synthesizable!

`always@(a)` `  y<=~a`	No delay (**correct**)	• the delay is imposed by the gate from the technology library used to map the NB statement (post synthesis) • **Correct Implementation for any synthesizable Verilog description**
`always@(a)` `  #5 y<=~a`	Left-hand-side delay (HW nonsense)	• Evaluate Right-Hand-Side (RHS) then update the Left-Hand-Side (LHS) after 5 timeunit • What if an update happens at 2.5timeunits? • The first value is lost!!! • NBA are less efficient to simulate, thus LHS delay for NBA is also discouraged for testbenches.
`always@(a)` `  y <= #5 ~a`	Right-hand-side delay (**correct for transport delay modeling**)	• Immediately evaluate the RHS, then wait 5 timeunit before updating the LHS of the statement • What if an update happens at 2.5timeunits? • **The first value is kept and the following are queued. (multiple LHS update events are generated)**

- **Do not use LHS delays to model combinatorial logic using NBA (HW nonsense)**
- **Use RHS delays to model the transport delay in combinatorial logic using NBA (new output is queued)**

Behavioral Delay Models: Continuous Assignments

Adding RHS or LHS delays to continuous assignments is also possible.

- It makes them not synthesizable
- Used to model the inertial delay in combinatorial logic to prevent undesired glitches

`assign y = ~a`	No delay **(correct)**	• the delay is imposed by the gate from the technology library used to map the statement (post synthesis)

`assign #5 y = ~a`	Left-hand-side delay (**correct for inertial delay modeling**)	• LHS delay in continuous assignment models the inertial delay of the logic
		  • If multiple event updates *a* within the delay period, only the generated update event is kept
		  • NOTE: continuous assignments do not queue new values, while keep the last within the #5 delay to model the inertial delay
		  • **Question: why don't BAs model the inertial delay?**
		  ➢ Hint: LHS delay with continuous assignments reschedule the event in the future, if any, while LHS delay with BA just update the LHS value

~~`assign y = #5 ~a`~~	Right-hand-side delay **(illegal syntax)**	

Structure of the testbench

Behavioral Verilog: Clock Signal

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

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

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

- The clock signal is made of non-synthesizable Verilog statements, i.e. delays
- **[RULE of THUMB]** Use blocking assignments to update the clock [in a synchronous design]:
  - Avoid races with other signals that are updated in the same time-step
  - Eventually use non-blocking assignments for all the other signals
  - The use of non-blocking clock update assignment is possible even if it is considered “improper”

Behavioral Verilog: Synchro Reset Signal

```
always #5 clk=~clk


initial begin //can race!!!
 clk<=0; rst=1;
 #5 rst=0;
end
```

IMPROPER DESIGN

- The clock [clk] and the reset [rst] are updated during the same time-step
  - We can observe simulation misbehaviors
  - NOTE: the description can be used ONLY if an asynchronous reset signal is used

```
always #5 clk=~clk


initial begin //no races!!!
 clk<=0; rst=1;
 @(posedge clk);
 rst<=0;
end
```

CORRECT DESIGN

- The clock [clk] and the reset [rst] are updated during the same time-step
  - However, we assume clock has precedence to avoid races in the simulation.
  - NOTE: all the signals in the TB that are synchronous to the clock must be updated using NBAs.

Behavioral Verilog: Data Signals

```
reg clk,rst,data;
always #5 clk=~clk

initial begin //can race!!!
 clk<=0; rst=1; data=0;
 @(posedge);
 rst<=0;
 #10 data=0;
end
```

Improper design

- The cause-effect property is lost if data is a synchronous signal.
- Data is updated with the clock…
  - Wait to see the hands-on

```
reg clk,rst,data;
always #5 clk=~clk

initial begin //can race!!!
 clk<=0; rst=1; data=0;
 @(posedge);
 rst<=0;
 #10 data<=0;
end
```

Correct design

- The clock is updated first and data is updated in the NBA region of the same clock cycle.
- Hint: Data shouldn’t appear in the sensitivity list of the always block that models sequential logic of the dut…
- **The use of NBAs or Bas for TB signals is subject to the actual semantic you want**

The `timescale directive: rounding and truncation errors

Truncation

```
`timescale 1ns/1ns
reg clk;
parameter clk_period=25;
always #(clk_period/2) clk=~clk;
```

Rounding

```
`timescale 1ns/1ns
reg clk;
parameter clk_period=25;
always #(clk_period/2.0) clk=~clk;
```

Correct Implementation

```
`timescale 1ns/100ps
reg clk;
parameter clk_period=25;
always #(clk_period/2.0) clk=~clk;
```

- The design is simulated using an event driven simulation approach
  - timeunit: the unit of time for any kind of delay
  - timeprecision: how many events the simulator can generate at most between two timeunits. Also named **time steps**
- **timeunit** always bigger than **timeprecision**
  - They can be equal, while rounding and truncation errors can arise
- Different simulators allow you to specify both **timeunit** and **timeprecision** directives without cluttering the hardware source files
  - Rounding errors: due to the precision of the used computing machine
  - Truncation errors: due to the used arithmetic

Behavioral Verilog: Maintainability

```
always #5 clk=~clk

initial begin //can race!!!
 clk<=0; rst=1;
 #5 rst<=0;
end
```

```
parameter clk_period = 10;
parameter rst_delay=10;
...
always #(clk_period/2) clk=~clk

initial begin //no races!!!
 clk<=0; rst=1;
 #(rst_delay) rst<=0;
end
```

IMPROPER DESIGN

- Use **parameter** and **`define** to increase the maintainability and maintainability of your testbenches
  - parameter clk_period
  - `define CLK_PERIOD

CORRECT DESIGN

- Parameters are better than defines:
  - parameters are properties of the module
  - defines are just placeholders

Design Under Test (DUT) Observability: System tasks

$monitor, $display, $write, $strobe

Unsynthesizable system tasks to print out useful information on the simulated design.

- **$strobe :** print the values at the end of the current timestep.
  - Executed in the postponed region
- `Example: $strobe("%t %b", time,signal);`

- **$display :** print the immediate values
  - Executed in the active region
- `Example: $display("%t %b", time,signal);`

- **$monitor :** print the values at the end of the current timestep if any values changed.
  - $monitor can only be called once; sequential calls will override the previous
  - Executed in the postponed region
- `Example: $monitor("%t %b",$time,signal);`

- **$write :** same as $display but doesn't terminate with a newline (\n)
  - Executed in the active region
- `Example: $write("%t %b\n", time,signal);`

Display/write tasks: $display

```
1  module tb;
2    logic [31:0]  x;
3    logic [15:0]  y;
4
5    initial
6    begin
7      x <=  32'h1234FEDC;
8      y =   16'h12DC;
9      $display ("x: %h", x);
10     $display ("y: %h", y);
11     #10;
12     $display ("x: %h", x);
13     $display ("y: %h", y);
14   end
15 endmodule
16
17 // OUTPUT:
18 // x: xxxxxxxx
19 // y: 12dc
20 // x: 1234fedc
21 // y: 12dc
```

The **$display** task displays the arguments in the order in which they appear in the arguments list

Similar to C **printf()**, **$display** allow the use of format specifier strings

- characters are displayed literally (except special characters)
- format specifiers (starting with a `%` character) indicate the location and method to translate a piece of data to characters

The newline character **\n** is implicitly appended at the end of the displayed string

Notes

- `Line 9` – x is xxxxxxxx since the **$display** executes in the active region
  - i.e., before the NBA at `line 7`
- `Line 10` – $display prints the **y** value assigned at `line 8`
  - i.e., **$display** executed after (in program order) the blocking assignment at `line 8`

Display/write tasks: $write

```
1  module tb;
2    logic [31:0]  x;
3    logic [15:0]  y;
4
5    initial
6    begin
7      x <=  32'h1234FEDC;
8      y =   16'h12DC;
9      $write ("x: %h", x);
10     $write ("y: %h\n", y);
11     #10;
12     $write ("x: %h\n", x);
13     $write ("y: %h\n", y);
14   end
15 endmodule
16
17 // OUTPUT:
18 // x: xxxxxxxy: 12dc
19 // x: 1234fedc
20 // y: 12dc
```

The **$write** system task is equivalent to the **$display** task

However, **$write** does not IMPLICITLY appends the newline character **\n** at the end of the displayed string

Notes

- `Lines 9-10`: the two strings are displayed on the same line since there is no `\n` at the end of the string at `line 9` or at the beginning of the string at `line 10`
- `Line 10,12`: the `\n` newline character at the end of `line 10` displays the two strings on two different lines

Strobe tasks: $strobe

```
1  module tb;
2    logic [31:0]  x;
3    logic [15:0]  y;
4
5    initial
6    begin
7      $strobe ("x: %h", x);
8      $strobe ("y: %h", y);
9      x <=  32'h1234FEDC;
10     y =   16'h12DC;
11     $strobe ("x: %h", x);
12     $strobe ("y: %h", y);
13     #10;
14     $strobe ("x: %h", x);
15     $strobe ("y: %h", y);
16   end
17 endmodule
18
19 // OUTPUT:
20 // x: 1234fedc
21 // y: 12dc
22 // x: 1234fedc
23 // y: 12dc
24 // x: 1234fedc
25 // y: 12dc
```

The **$strobe** system task prints the values of variables at the end of the time-step

It displays the arguments in a way that is equivalent to the **$display** task

Notes

- Line 7 – **x** is **1234fedc**
  - **$strobe** executes after the NBA at line 9
- Line 8 – **y** is **12dc**
  - **$strobe** executes after the blocking assignment at line 10

Monitor tasks: $monitor

```
1  module tb;
2    logic [31:0]  x;
3    logic [15:0]  y;
4
5    initial
6    begin
7      x <=  32'h1234FEDC;
8      y =   16'h12DC;
9      #5
10     x <=  32'h01234567;
11     y =   16'hABCD;
12     $monitor ("x: %h", x);
13     $monitor ("y: %h", y);
14     #10;
15     y =   16'h8172;
16     #5;
17     x <=  32'hA8B7C6D5;
18   end
19 endmodule
20
21 // OUTPUT:
22 // x: 01234567
23 // y: abcd
24 // y: 8172
25 // x: a8b7c6d5
```

The **$monitor** automatically prints whenever a variable in its argument list changes

It works in the postponed region

Behavior

$monitor is like a task spawned to run in the background of the main thread, that monitors and displays value changes of its argument variables

A new `$monitor` task can be issued any number of times during simulation.

Notes

- `Time 0`: there are no active **$monitor** tasks, hence the values of x and y are not displayed
- `Time 5`: **$monitor** tasks display the updated values of x and y
- `Time 15`: **$monitor** task displays the updated value of y
- `Time 20`: **$monitor** task displays the updated value of x

Format specifiers - numeric

```
1  module tb;
2    int           x         = 32'd1647;
3    real          y         = 5.5e-1;
4    string        s         = "Testo";
5
6    initial
7    begin
8      $display ("x: 0x%h", x);
9      $display ("x:   %d", x);
10     $display ("x: 0o%o", x);
11     $display ("x: 0b%b", x);
12     $display ("x[7:0]: %c", x[7:0]);
13     $display ("y: %f", y);
14     $display ("y: %e", y);
15   end
16 endmodule
17
18 // OUTPUT:
19 // x: 0x0000066f
20 // x:          1647
21 // x: 0o00000003157
22 // x: 0b00000000000000000000011001101111
23 // x[7:0]: o
24 // y: 0.550000
25 // y: 5.500000e-01
```

Take an argument

- `%h, %H` display a value in hexadecimal format
- `%d, %D` display a value in decimal format
- `%o, %0` display a value in octal format
- `%b, %B` display a value in binary format
- `%c, %C` display a value in ASCII encoding
- `%f, %F` display a real value in decimal format
- `%e, %E` display a real value in exponential format

Notes

- `Line 12` displays the ASCII encoding of the 8 least significant bits of the `x int` variable
  - ASCII encoding: `o`
  - Hexadecimal encoding: `6f` (see two least significant digits displayed at `line 8`)

Format specifiers – string, time, %p and %m

```
1  module tb;
2    logic [15:0]  z [0:7];
3    string        s         = "Testo";
4    time          t;
5
6    initial
7    begin
8      for (int i=0; i<8; i++)
9        z[i] = $random ();
10     $display ("s: %s", s);
11     #10;
12     t = $time;
13     $display ("Time: %t", t);
14     #15;
15     t = $time;
16     $display ("Time: %t", t);
17     $display ("Module: %m");
18     $display ("%p", z);
19   end
20 endmodule
21
22 // OUTPUT:
23 // s: Testo
24 // Time:                10000
25 // Time:                25000
26 // Module: tb
27 // '{13604,24193,54793,22115,31501,39309,33893,21010}
```

Take an argument

- `%s`, `%S` display a string
- `%t`, `%T` display a time value
- `%p`, `%P` pretty-print simple and complex data types (e.g., arrays and structures)

Do not take an argument

- `%m`, `%M` display the hierarchical name of the module, task, function, or named block which invoked the system task

Notes

- Line 12,15: `$time` is a system task that returns the current simulation time as a 64-bit integer

Format specifiers – examples using left-padding

```
1  module tb;
2    int   x = 32'd1647;
3    real  f = 3.6e-1;
4
5    initial
6    begin
7      $display ("x: %d", x);
8      $display ("x: %0d", x);
9      $display ("x: %8d", x);
10     $display ("x: %08d", x);
11
12     $display ("f: %f", f);
13     $display ("f: %1.1f", f);
14     $display ("f: %2.2f", f);
15   end
16 endmodule
17
18 // OUTPUT:
19 // x:         1647
20 // x: 1647
21 // x:     1647
22 // x: 00001647
23 // f: 0.360000
24 // f: 0.4
25 // f: 0.36
```

Notes

- Line 7: leftmost 0s of the 32-bit integer value are substituted by whitespace char
- Line 8: leftmost 0s of the 32-bit integer value are not displayed
- Line 9: leftmost 0s of the 8-digit integer value are substituted by whitespace char
- Line 10: leftmost 0s of the 8-digit integer value are displayed

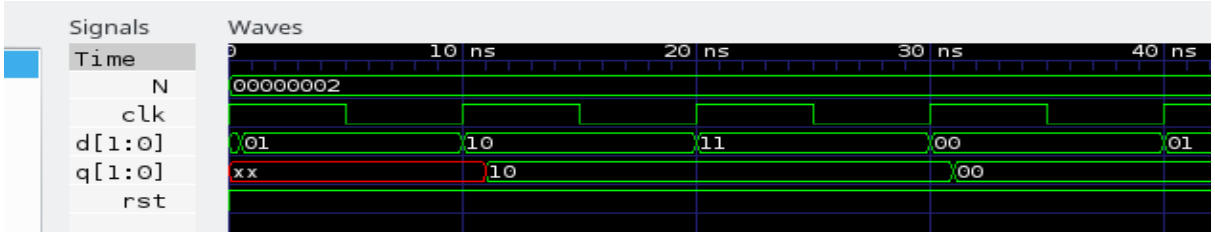
- Line 13: the displayed value is rounded so there is only one fractional digit
- Line 14: two fractional digits are displayed

Design Under Test (DUT) Observability with VCD

Value Change Dump (VCD) File Extraction

The Value Change Dump file can store all the transitions for all the signals in the design during the entire simulation

- Verilog offers system tasks to instruct the simulator on the vcd dump procedure.
- Several simulators allow a specific, tool-dependant solution for the VCD file dump without affecting the .v files
- The VCD can be read using a waveform visualizer:
  - Gtkwave [opensource]
  - Simvision [cadence]
  - Xsim waveform [xilinx, free*]



From within the testbench .v file

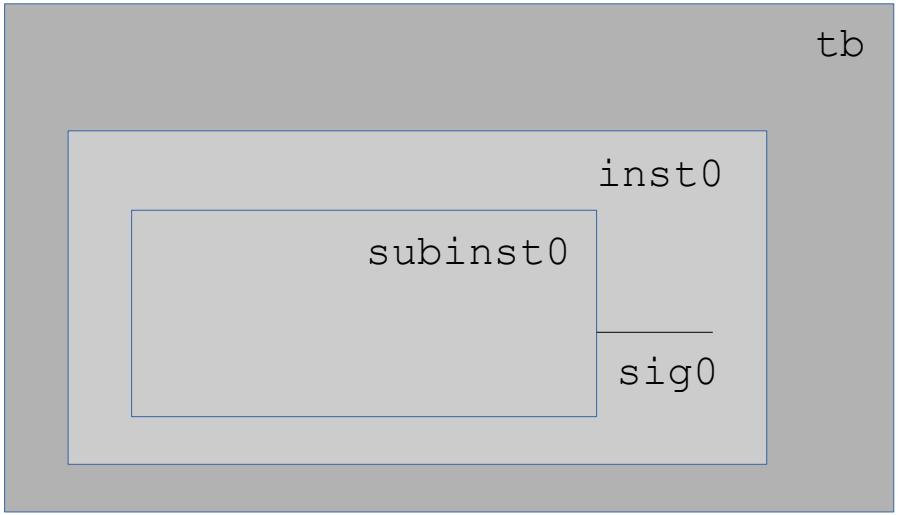
```
$dumpfile("file.vcd");
$dumpvars(level,<module_instance/variable>);
```

- 0 dump all variable in the module and all of its submodules
- 1 dump all variables in the module
- <level> dump all variables in the module and all the variables in the children up to level = <level>-1

Hierarchical names

Hierarchical path names

- Are based on the top module identifier followed by the list of module instance names in the path to the signal



Two useful scenarios:

1) verification/simulation

- Probe inter-module signals in the design from the tb or monitor modules

2) synthesis/implementation

- Connect module/signal instances from generate blocks

- To access signal sig0

```
tb.inst0.subinst0.sig0
```

- **NOTE**: sig0 can be either an input/output or an internal signal of subinst0

Hands-on session (using Cadence Xcelium)

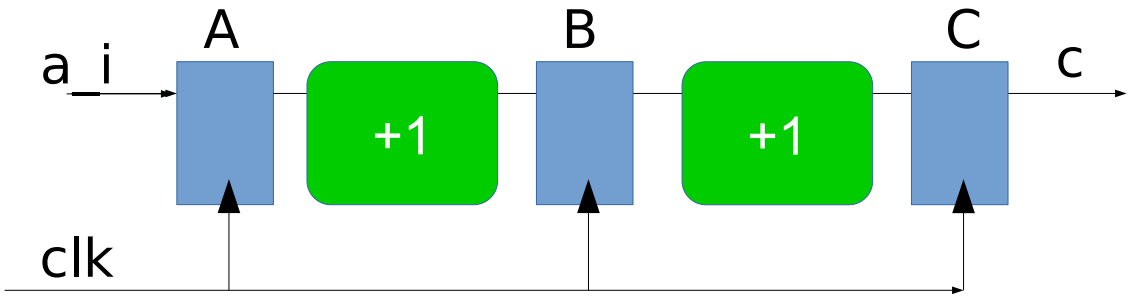
– the verification of the **flip-flop** and the chain of **flip-flops** -

Goals: (1) check the simulation model; **(2)** pre- and post-synthesis functional verification: use the same testbench; **(3)** mix SystemVerilog and VHDL

Verilog for Design: An Example

```
...
wire [2:0] a_i, b_i, c_i;
reg [2:0] a, b, c;

always@(posedge clk)
begin
  a <= a_i;
  b <= a+1;
  c <= b+1;
end
...
```

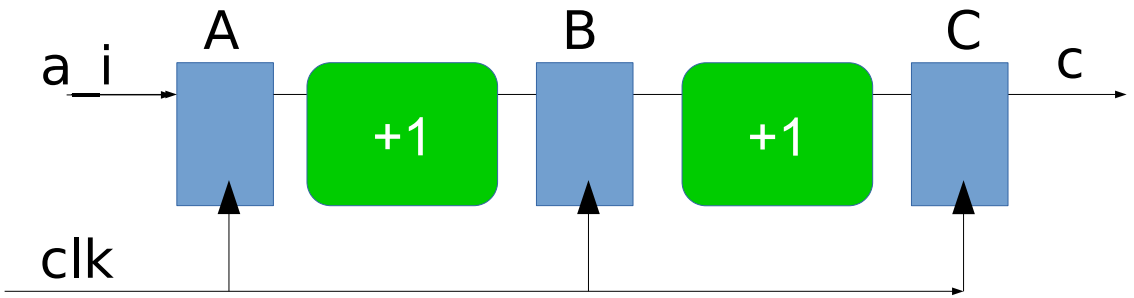


- The order of the non-blocking assignments within the same always block does not matter
- The structure of the always makes the synthesizer to infer flip-flops

Verilog for Design: A second coding style

```
...
wire [2:0] a_i, b_i, c_i;
reg [2:0] a, b, c;

always@(posedge clk)
begin
  a <= a_i;
  b <= b_i;
  c <= c_i;
end
assign b_i = a + 1;
assign c_i = b + 1;
...
```



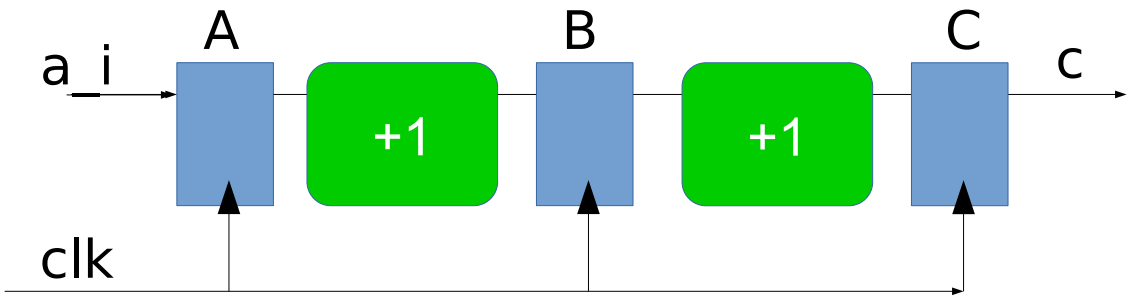
- Same semantic; combinatorial logic is outside the always block
- The order of the non blocking assignment does not matter

Verilog for Design: A WRONG solution

```
...
wire [2:0] a_i, b_i, c_i;
reg [2:0] a, b, c;

always@(posedge clk)
begin
  a <= a_i;
  b <= b_i;
  c <= c_i;
  assign b_i = a + 1;
  assign c_i = b + 1;
end

...
```



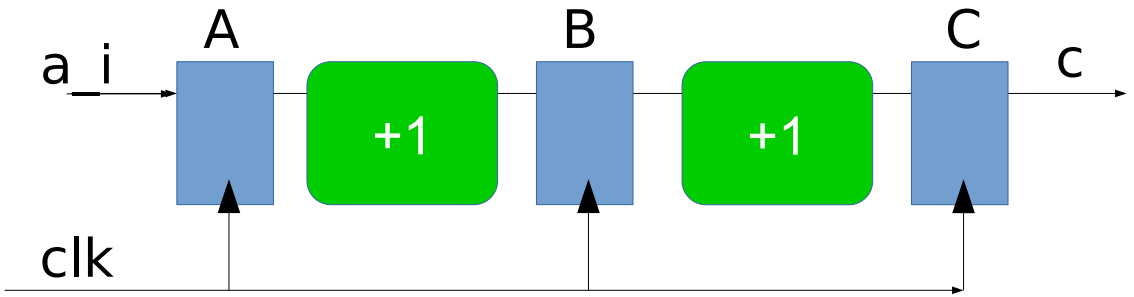
- **Syntactically NOT correct**
  - Always blocks cannot contain continuous assignment statements

Verilog for Design: A third coding style

```
...
wire [2:0] a_i, b_i, c_i;
reg [2:0] a, b, c;

always@(posedge clk)
  a <= a_i;
always@(posedge clk)
  b <= b_i;
always@(posedge clk)
  c <= c_i;

assign b_i = a + 1;
assign c_i = b + 1;
...
```

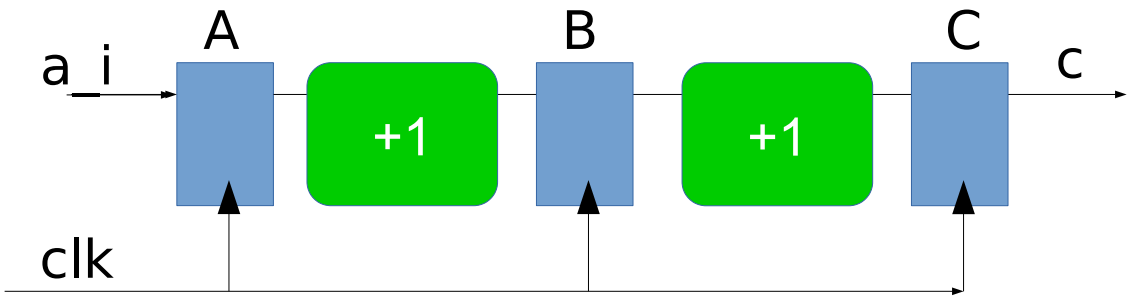


- **Same (CORRECT) behavior**
  - Concurrent (sequential) always blocks, the same circuit is inferred

Verilog for Design: Another WRONG solution

```
...
wire [2:0] a_i, b_i, c_i;
reg [2:0] a, b, c;

always@(posedge clk)
begin
  a = a_i;
  b = b_i;
  c = c_i;
end
assign b_i = a + 1;
assign c_i = b + 1;
...
```



- **Syntactically correct** (it can be synthesized)
- **Semantically broken**
  - Blocking assignments do not reflect the intrinsic behavior of the sequential logic
    - The order of the statements matters!
  - Don't use blocking assignments in sequential logic

Verilog for Design: Simulation Hints

The simulator generates and processes the events following a strict policy

- **Given a hardware description in Verilog/SystemVerilog**

  1) update the variables/wire in the combinational constructs:
     - Continuous assignments
     - **always@(*)**
     - NOTE: Recursive actions can happen

  2) evaluate all the right hand of each non-blocking assignment (unordered), and defer the update of the left side of the statement

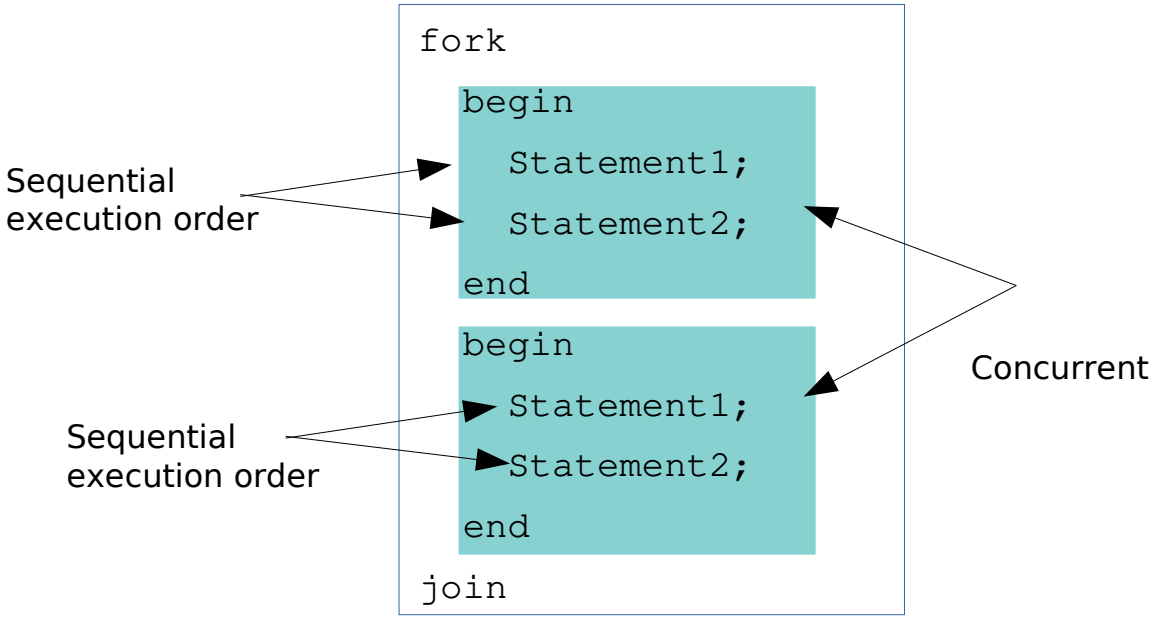
  3) update the left side of the statement

Fork-Join construct

Fork Join to create concurrent processes

fork-join construct

- Defined in Verilog 2001
- Truly parallel threads of execution into the simulation
- All threads start at the same time
- Use within procedural blocks, i.e., always or initial



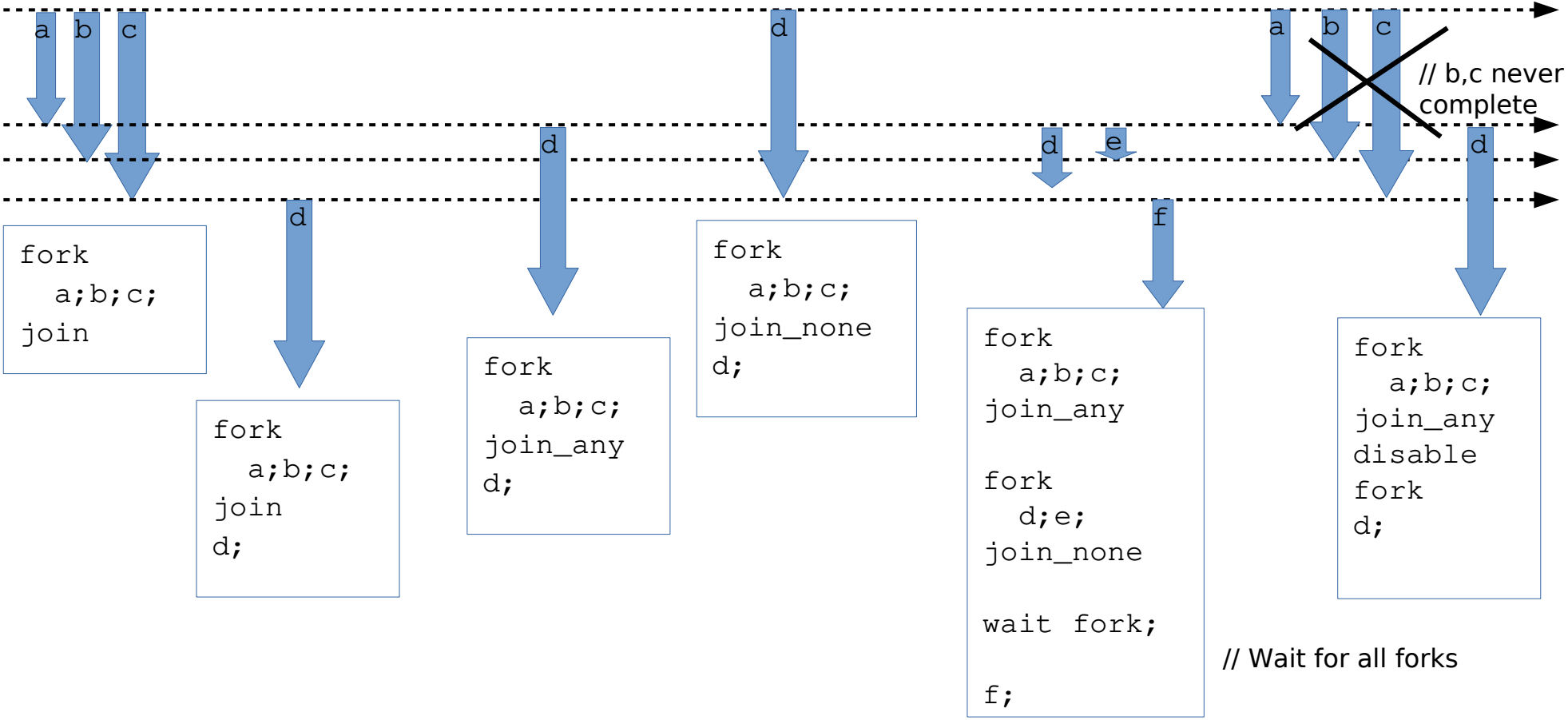
SystemVerilog added 2 fork-join variants

- fork-join_any
- fork_join_none

Procedural blocks vs Fork-join:

- Any form of fork-join implements concurrency between the statements within the fork join keywords
- Always blocks are concurrent to each others but once the simulator starts executing a procedural block must complete all the statements within it.

The fork-join constructs



Fork Join to create concurrent processes

What time unit does “enable” get the value 1?

```
initial
begin
  enable=0;
  #1
  begin
    #1 a=0;
    #2 b=0;
  end
  enable=1
  $display($time);
end
```

```
initial
begin
  enable=0;
  #1
  fork
    #1 a=0;
    #2 b=0;
  join
  enable=1
  $display($time);
end
```

```
initial
begin
  enable=0;
  #1
  fork
    #1 a=0;
    #2 b=0;
  join_any
  enable=1
  $display($time);
end
```

```
initial
begin
  enable=0;
  #1
  fork
    #1 a=0;
    #2 b=0;
  join_none
  enable=1
  $display($time);
end
```

Fork Join to create concurrent processes

What time unit does “enable” get the value 1?

```
initial
begin
  enable=0;
  #1
  begin
    #1 a=0;
    #2 b=0;
  end
  enable=1
  $display($time);
end
```

4

```
initial
begin
  enable=0;
  #1
  fork
    #1 a=0;
    #2 b=0;
  join
  enable=1
  $display($time);
end
```

3

```
initial
begin
  enable=0;
  #1
  fork
    #1 a=0;
    #2 b=0;
  join_any
  enable=1
  $display($time);
end
```

2

```
initial
begin
  enable=0;
  #1
  fork
    #1 a=0;
    #2 b=0;
  join_none
  enable=1
  $display($time);
end
```

1