



Digital IC Design

Session 6: Behavioral Modeling(1)

2024 | Nineplus Infotech X Anseong Polytechnic University
Sr Research Engineer, William Woo

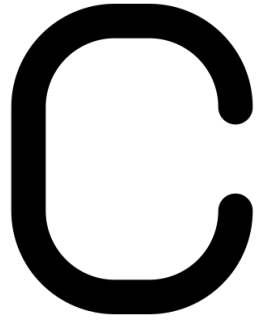


1. Hardware Description

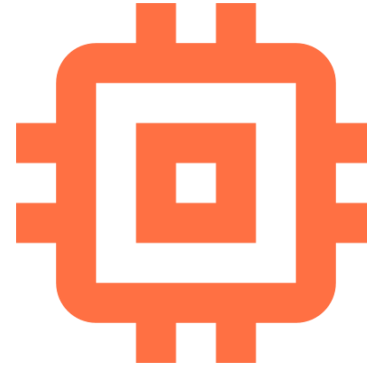
THINK

① SW개발 언어와 HW개발 언어는 어떤 차이점이 있을까?

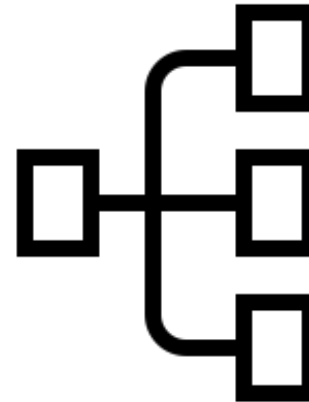
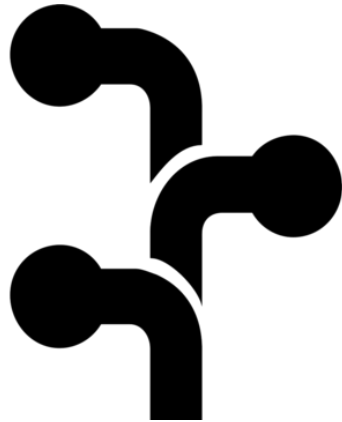
Software Language VS Hardware Descriptive Language



Sequential programming language



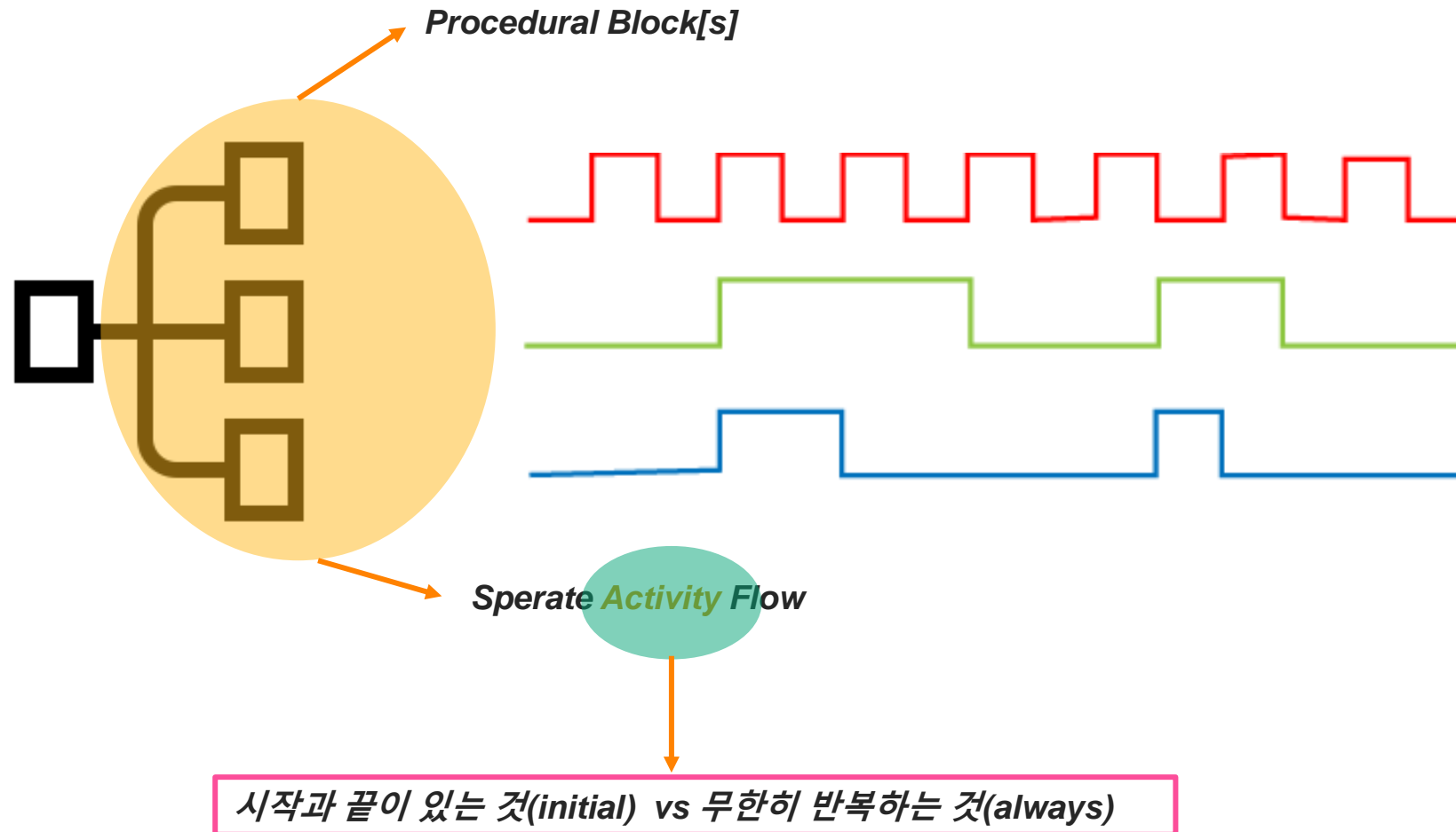
Concurrent programming language



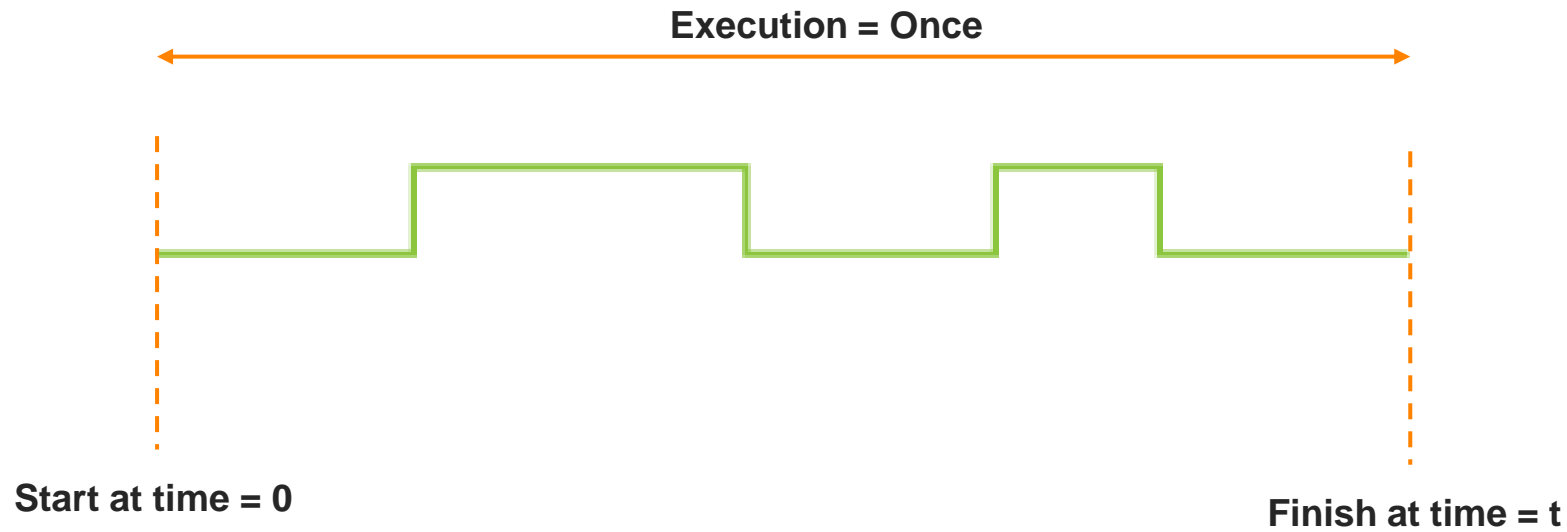
THINK

① How can we describe HW's concurrent behavior?

Procedural Block



Initial Statement: overview



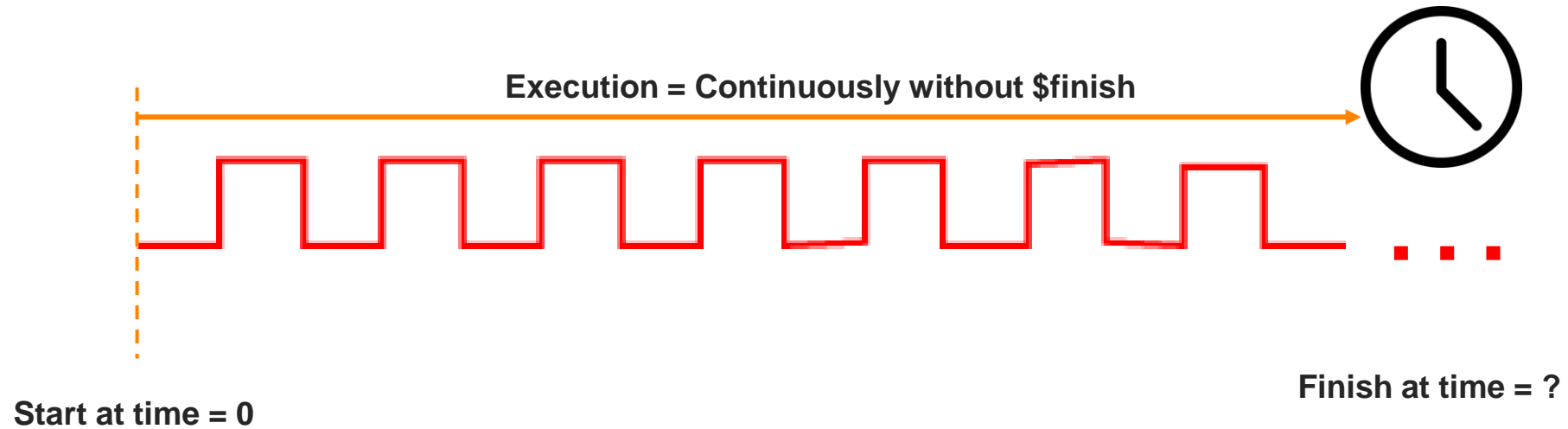
Initial Statement: Example

```
1  module tb_initial;
2
3      reg x, y, a, b, m;
4
5      // 1st block
6      initial
7      |   m = 1'b0;
8
9      // 2ed block
10     initial
11     begin
12         #5 a = 1'b1;
13         #25 b = 1'b0;
14     end
15
16     // 3rd block
17     initial
18     begin
19         #10 x = 1'b0;
20         #25 y = 1'b1;
21     end
22
23     // 4th block
24     initial
25     |   #50 $finish;
26
27 endmodule
```



Time at	Statement executed	Block executed
0	m = 1'b0	1
5	a = 1'b1	2
10	x = 1'b0	3
30	b = 1'b0	2
35	y = 1'b1	3
50	\$finish	4

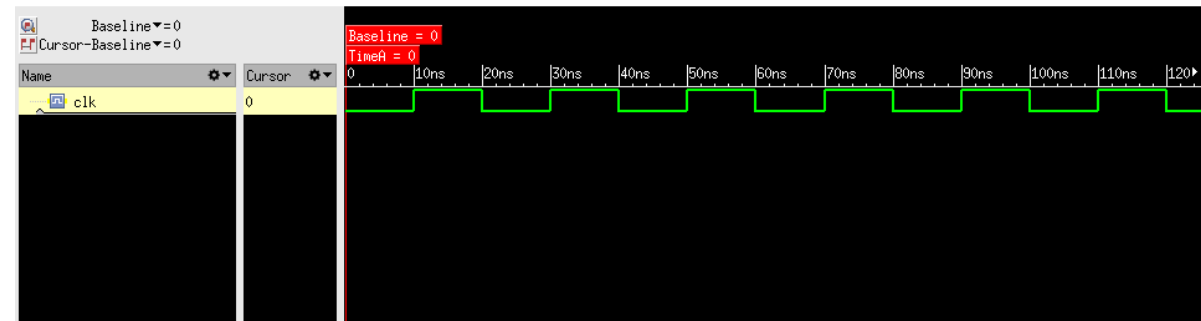
Always Statement: overview



... “always” block is similar to an infinite loop in C language ...

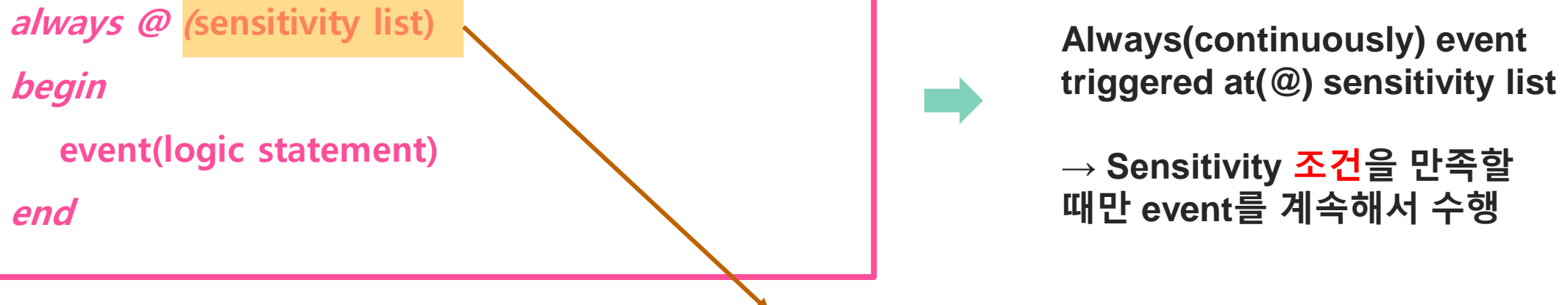
Always Statement: Example

```
1  module clock_gen;
2
3  reg clk;
4
5  // 1. Initialize clk at time = 0
6  initial begin
7      clk = 1'b0;
8  end
9
10 // 2. Toggle clk every half-cycle to produce time period = 20
11 always
12     #10 clk = ~clk;
13
14 // 3. Finish system tick
15 initial
16     #1000 $finish;
17
18 endmodule
```



Always Statement: Event control

```
always @ (sensitivity list)  
begin  
    event(logic statement)  
end
```



The diagram illustrates the components and usage of the 'always' statement. A pink box on the left contains the Verilog code snippet. A green arrow points from the '(sensitivity list)' part of the code to a text box on the right. A brown arrow points from the entire code snippet to a box at the bottom containing examples.

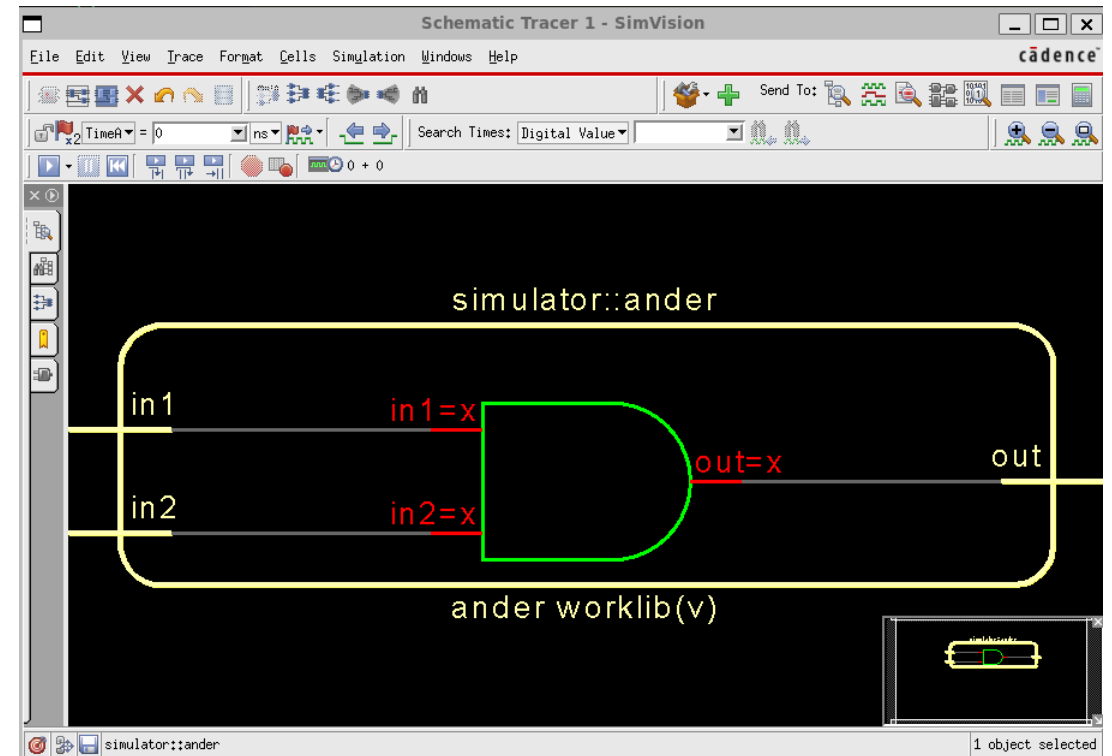
Always(continuously) event triggered at(@) sensitivity list

→ Sensitivity 조건을 만족할 때만 event를 계속해서 수행

- Multiple List
→ `always @(a, b, ..., n) / always @(a or b or ... n)`
- Edge Trigger
→ `always @(posedge/negedge sig)`
- Wild Card
→ `always @* / always @(*)`

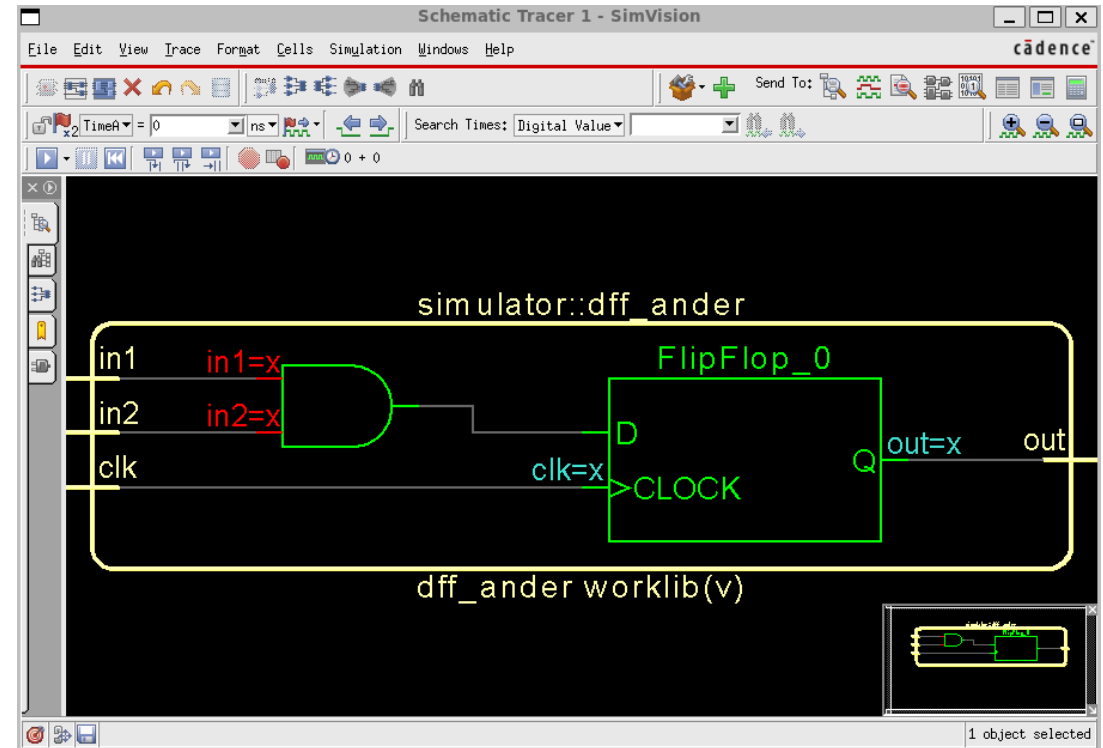
Always Statement: AND gate modeling1

```
1  module ander (  
2      in1, in2, out  
3  );  
4  
5      input wire in1, in2;  
6      output reg out;  
7  
8      always @(in1 or in2)  
9  begin  
10         out = in1 & in2;  
11     end  
12  
13 endmodule
```



Always Statement: AND gate modeling2

```
1  module dff_ander (  
2      in1, in2, out, clk  
3  );  
4  
5      input wire in1, in2, clk;  
6      output reg out;  
7  
8  
9      always @(posedge clk)  
10     begin  
11         out = in1 & in2;  
12     end  
13  
14  
15  
16     endmodule
```



Procedural Assignment

initial / always @ (sensitivity list)

begin

Procedural block

event(logic statement)

end

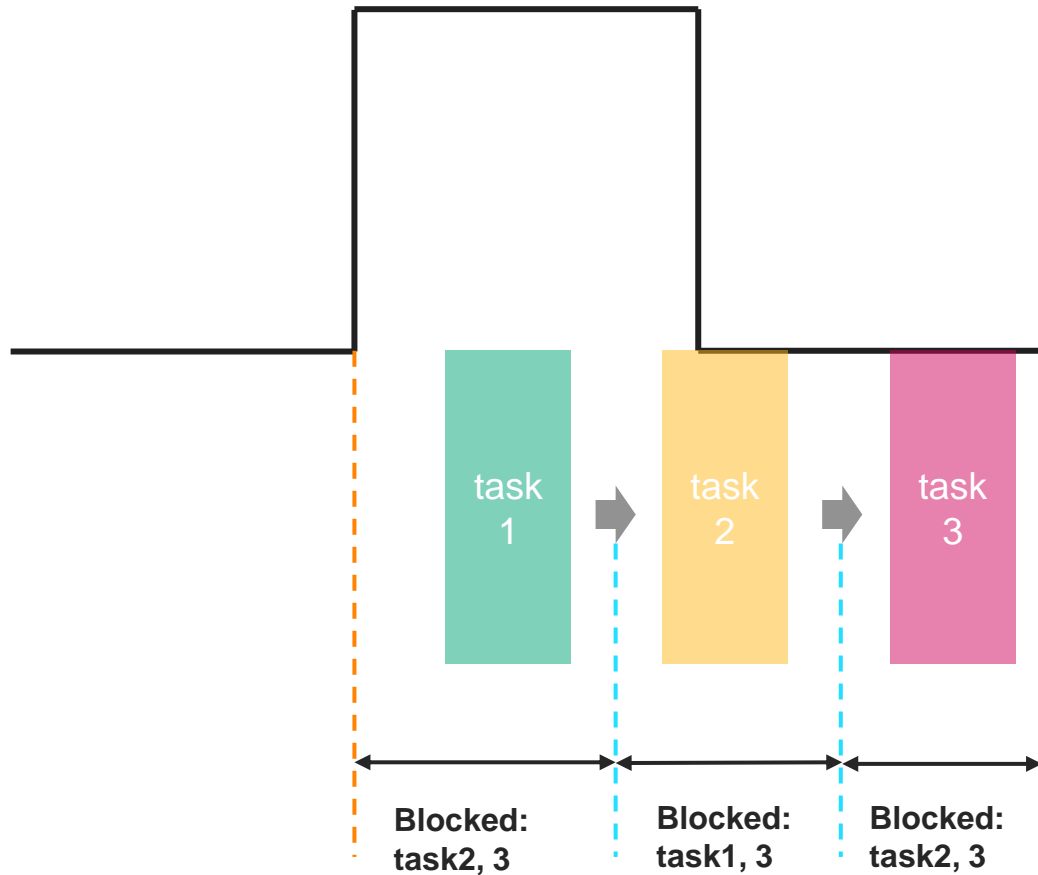
Blocking vs Non-blocking

=

←=

- Type of LHS = reg
- 또 다른 procedural assignment가 변수의 값을 업데이트 하기 전까지는 값을 유지(∴ reg)

Blocking assignment

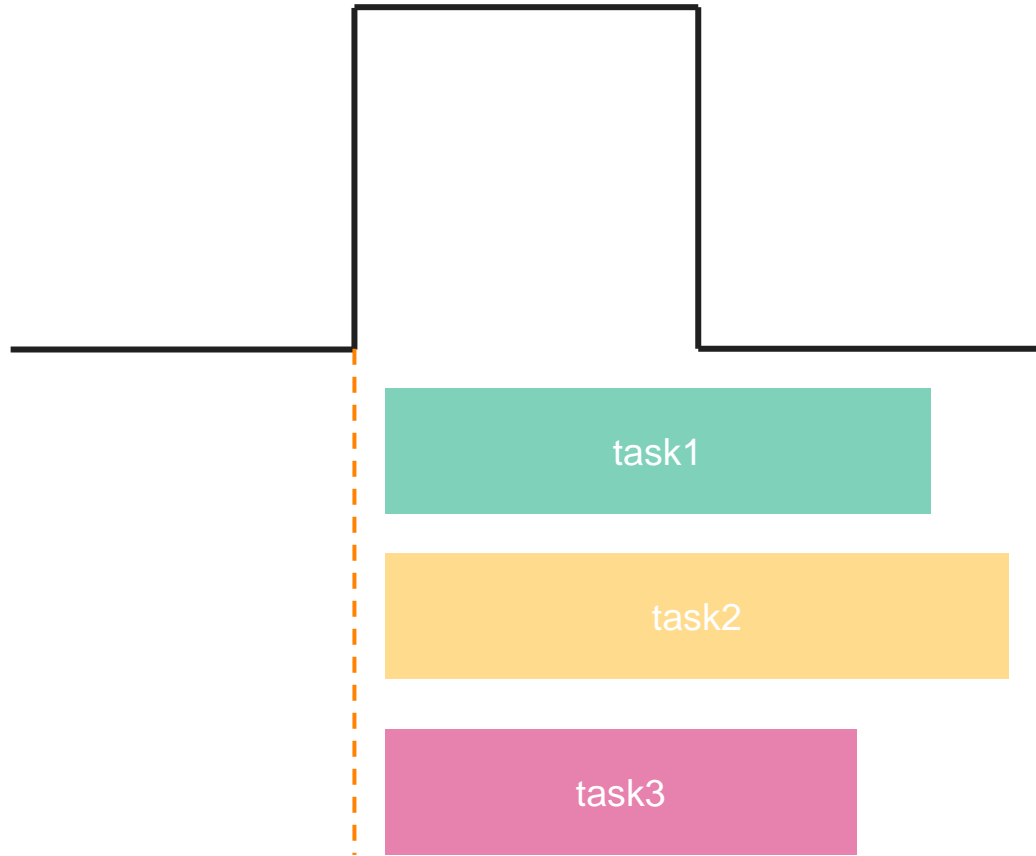


```
always @(posedge clk, negedge n_rst)
begin
    b = a; // task #1
    c = b; // task #2
    a = c; // task #3
end
```

*... executed in the order tasks
are specified in a sequential block*

...

Non-Blocking assignment



Each task doesn't wait for the others to complete execution!

```
always @(posedge clk, negedge n_rst)
begin
    // each task simultaneously occurs at positive clk edge
    b <= a; // task #1
    c <= b; // task #2
    a <= c; // task #3
end
```

Non-blocking is used to deliver/assign multiple value at the same time

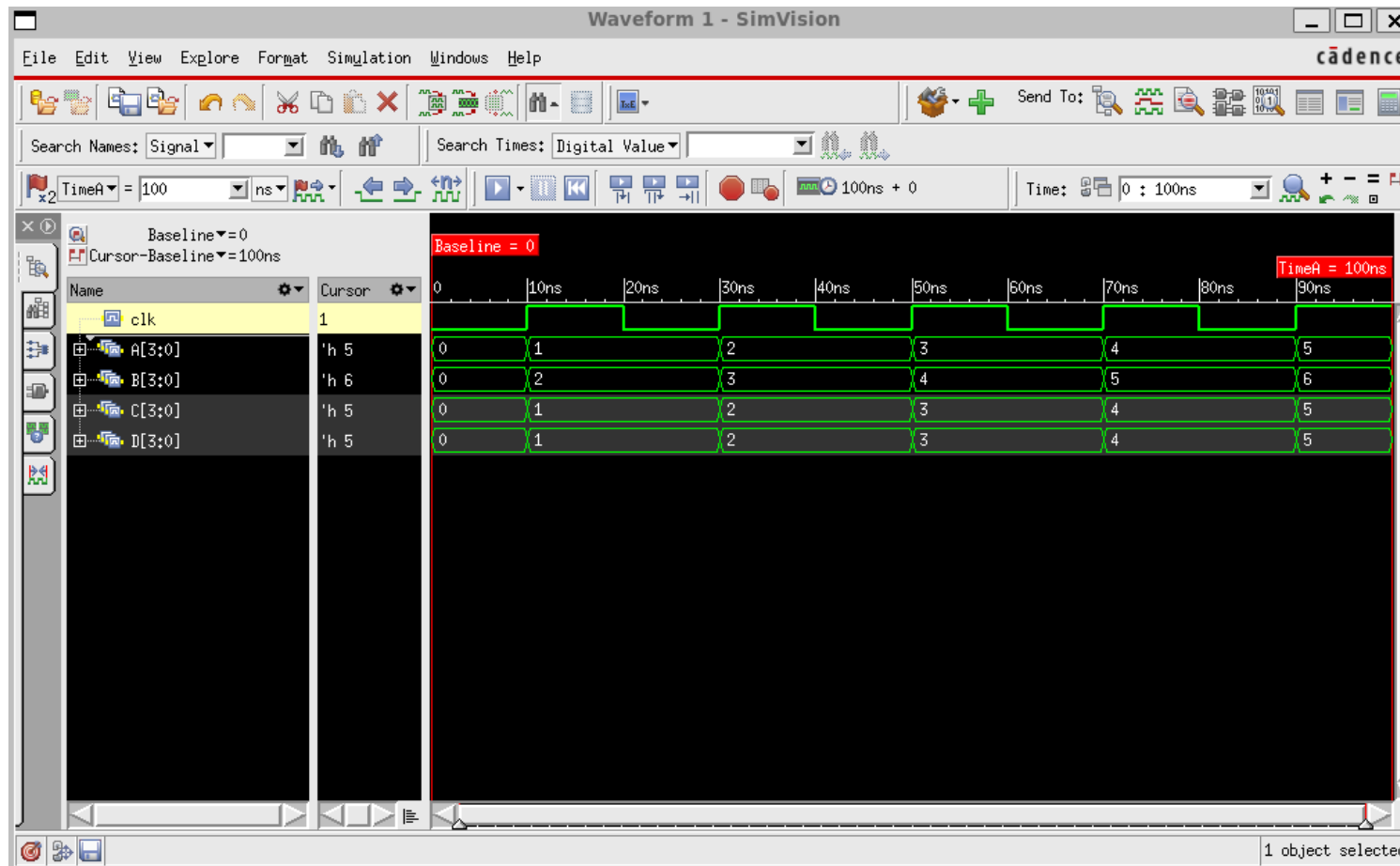
Blocking vs Non-Blocking

```
1  module non_blocking;
2
3  reg [3:0] A;
4  reg [3:0] B;
5  reg [3:0] C;
6  reg [3:0] D;
7
8  reg clk;
9
10 // time = 0
11 initial
12 |   clk = 1'b0;
13
14 initial begin
15 |   A = 4'd0; B = 4'd0; C = 4'd0; D = 4'd0;
16 end
17
18 // 20ns clock
19 always
20 |   #10 clk = ~clk;
21
22
23 always @(posedge clk)
24 begin
25 |   A = A + 1; $display("[%0t] A = 0x%4h", $time, A);
26 |   B = A + 1; $display("[%0t] B = 0x%4h", $time, B);
27 end
28
29
30 always @(posedge clk)
31 begin
32 |   C <= C + 1; $display("[%0t] C = 0x%4h", $time, C);
33 |   D <= C + 1; $display("[%0t] D = 0x%4h", $time, D);
34 end
35
36 initial // 5 tick
37 |   #100 $finish;
38
39 endmodule
```



```
xcelium> run
[10] A = 0x0001
[10] B = 0x0002
[10] C = 0x0000
[10] D = 0x0000
[30] A = 0x0002
[30] B = 0x0003
[30] C = 0x0001
[30] D = 0x0001
[50] A = 0x0003
[50] B = 0x0004
[50] C = 0x0002
[50] D = 0x0002
[70] A = 0x0004
[70] B = 0x0005
[70] C = 0x0003
[70] D = 0x0003
[90] A = 0x0005
[90] B = 0x0006
[90] C = 0x0004
[90] D = 0x0004
Simulation complete via $finish(1) at time 100 NS + 0
```

Blocking vs Non-Blocking



LAB(1)

- Make 1-bit register referring to the description below:
 1. Use positive edge clock
 2. Use active low reset
- Verifying DUT using testbench

LAB(2)

- Make 4-bit register referring to the description below:
 1. Use positive edge clock
 2. Use active low reset
- Verifying DUT using testbench