



Diseño Lógico y Arquitectura de Sistemas Digitales

2024 – 1er Cuatrimestre

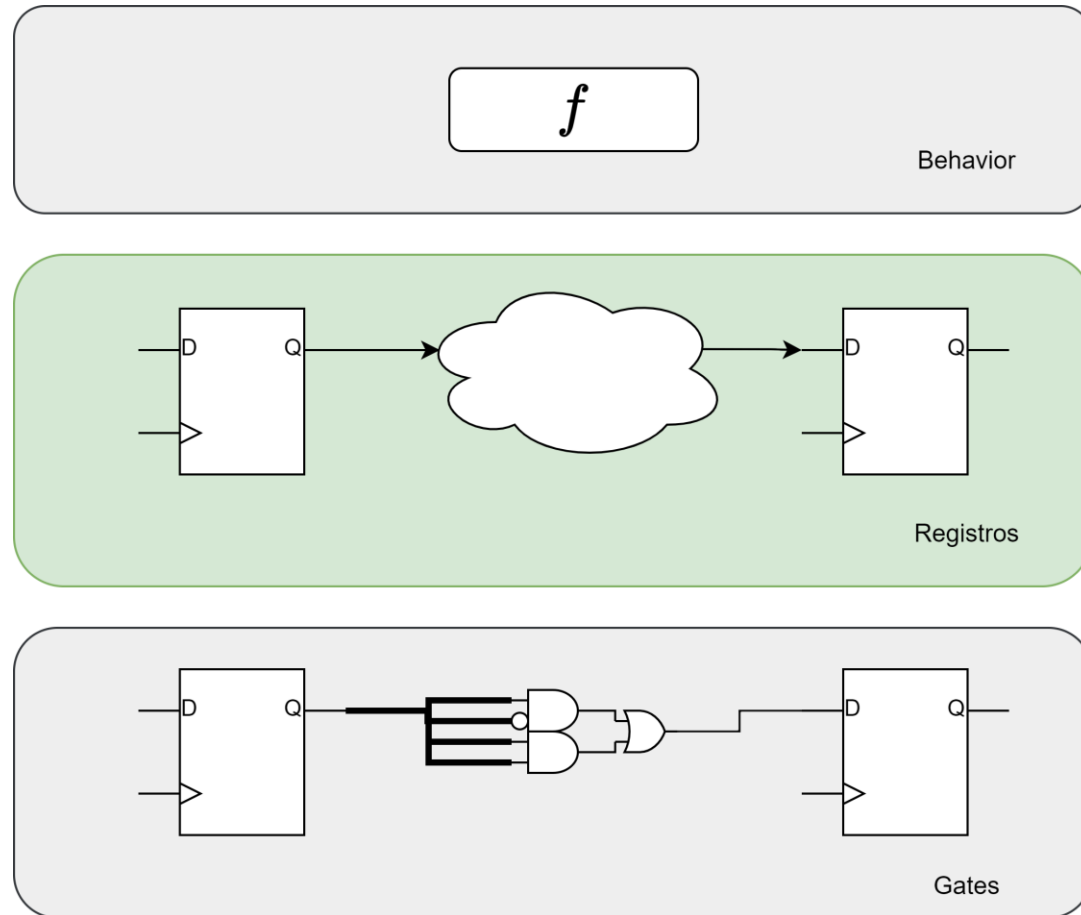
Clase 3: Verilog

Temas Seleccionados



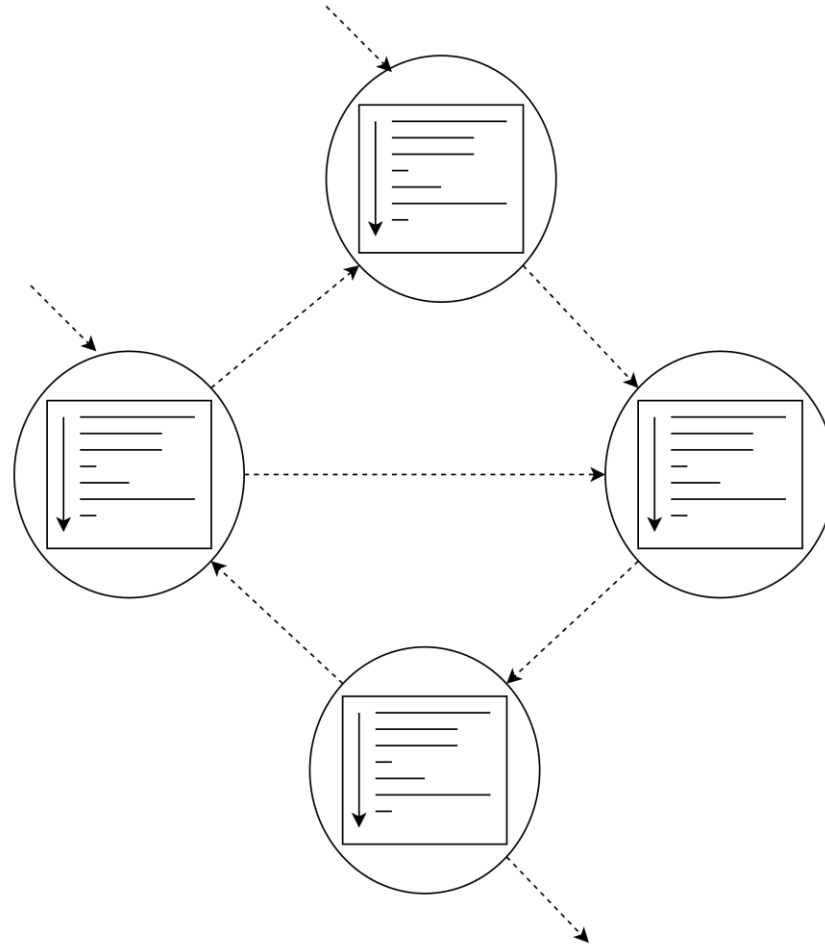
Hardware Description Language (HDL)

- Behavioral:
 - Algoritmos
- Register Transfer level (RTL)
 - Registros y conexiones
- Gate Level
 - Estructural



Interacción entre los módulos

```
module design (  
    list_of_port_declarations  
);  
    net_declaration(s)  
    reg_declaration(s)  
  
    always @*  
        begin  
            statement(s)  
        end  
  
    always @(posedge clk)  
        begin  
            statement(s)  
        end  
  
    assign net_assignment(s)  
  
endmodule
```



Blocking/Non Blocking Assignments

```
reg [7:0] byte=8'b00001111;
...
// Ejercicio: Intercambiar los nibbles
byte[3:0] = byte[7:4];
byte[7:4] = byte[3:0];
// Cual es el resultado?
...
```

```
reg [7:0] byte=8'b00001111;
...
// Ejercicio: Intercambiar los nibbles
byte[3:0] <= byte[7:4];
byte[7:4] <= byte[3:0];
// Cual es el resultado?
...
```



```
always @(*) begin
    a = 2;
    b = a + 1;
end
```

```
always @(*) begin
    a <= 2;
    b <= a + 1;
end
```

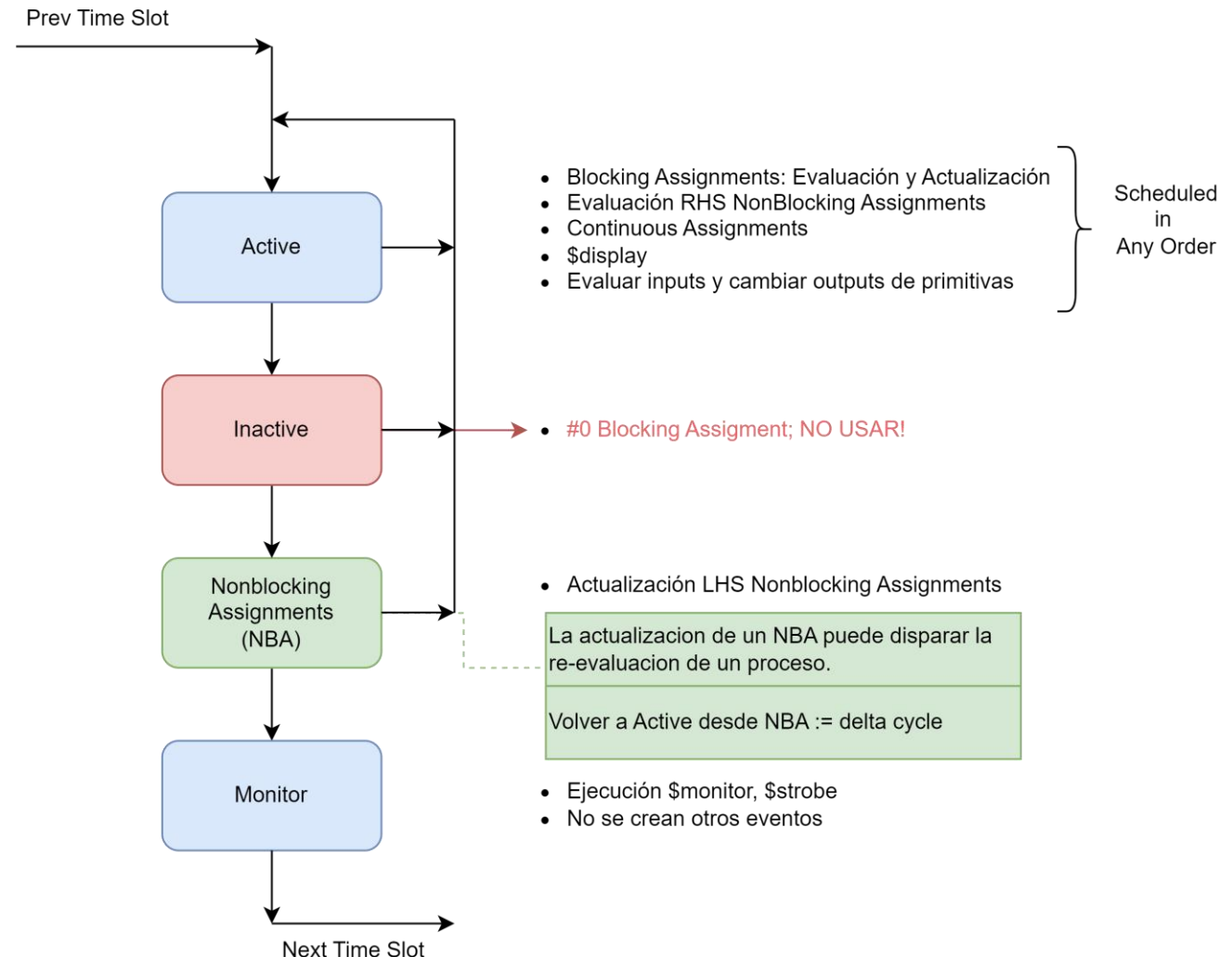


- Blocking (=) : bloquea la ejecución hasta que completa la asignación.

- NonBlocking (<=) :
 - Se calcula la expresión de RHS
 - Se agenda la actualización de la variable LHS

Verilog Event Scheduling

- **Evento:** Actualizar un objeto (signal/variable)
- **Scheduling:** Determinar cuando un evento debe ejecutarse
- **Event Queue:** ToDo list del simulador. Se divide en time-slots



Ciclo de Simulación 1/n

```
module delta_tb;

  reg a = 0, b = 0, c = 0, d = 0;

  initial begin : process_0
    #5;
    a = 1;
  end

  always @(a or b) begin : process_1
    b <= a;
    c <= b;
  end

  always @(b) begin : process_2
    d <= b;
  end

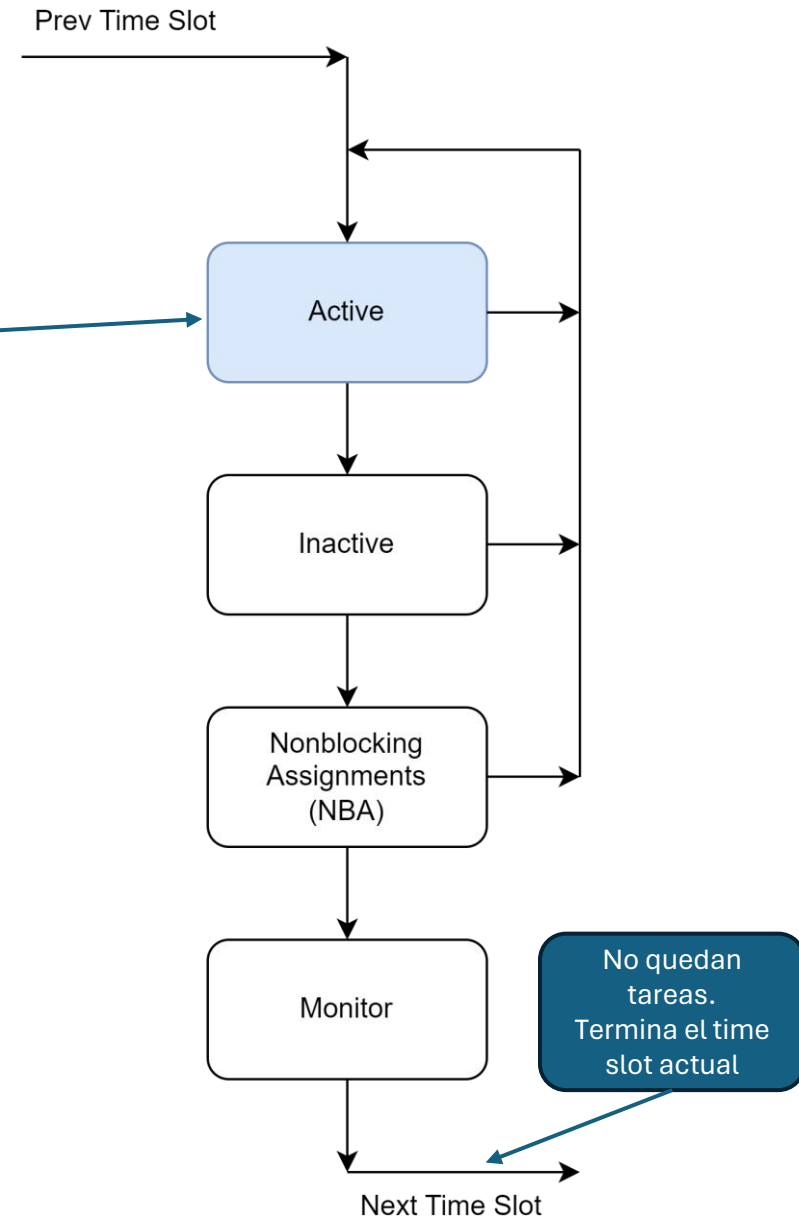
  initial begin
    #10;
    $display("\033[32mTEST PASSED\033[0m");
    $finish;
  end

endmodule
```

Ejecución
process_0
Delay 5 units

Suspendidos
Nada que
ejecutar

Name	C
a	0
b	0
c	0
d	0



Ciclo de Simulación 2/n

```
module delta_tb;

  reg a = 0, b = 0, c = 0, d = 0;

  initial begin : process_0
    #5;
    a = 1;
  end

  always @(a or b) begin : process_1
    b <= a;
    c <= b;
  end

  always @(b) begin : process_2
    d <= b;
  end

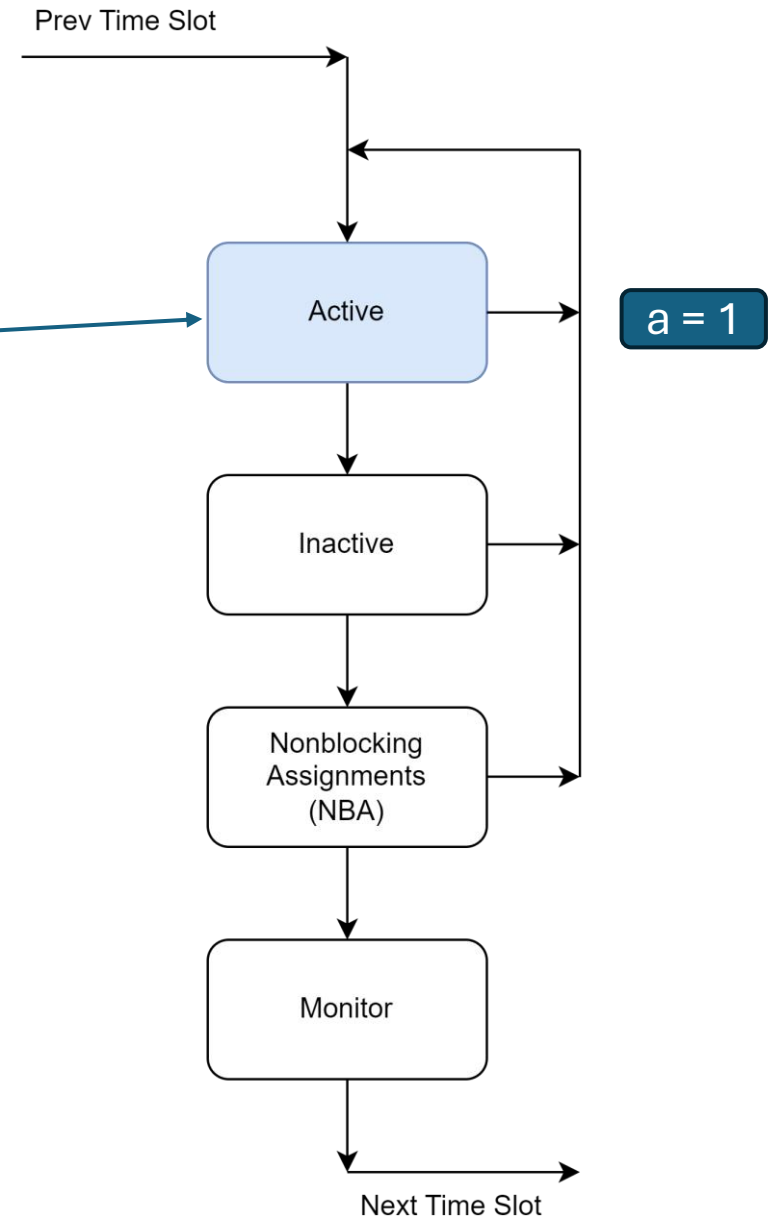
  initial begin
    #10;
    $display("\033[32mTEST PASSED\033[0m");
    $finish;
  end

endmodule
```

Blocking
Assignment

Dispara lista
de eventos de
process_1

a = 1
b = 0
c = 0
d = 0



Ciclo de Simulación 3/n

```
module delta_tb;

    reg a = 0, b = 0, c = 0, d = 0;

    initial begin : process_0
        #5;
        a = 1;
    end

    always @(a or b) begin : process_1
        b <= a;
        c <= b;
    end

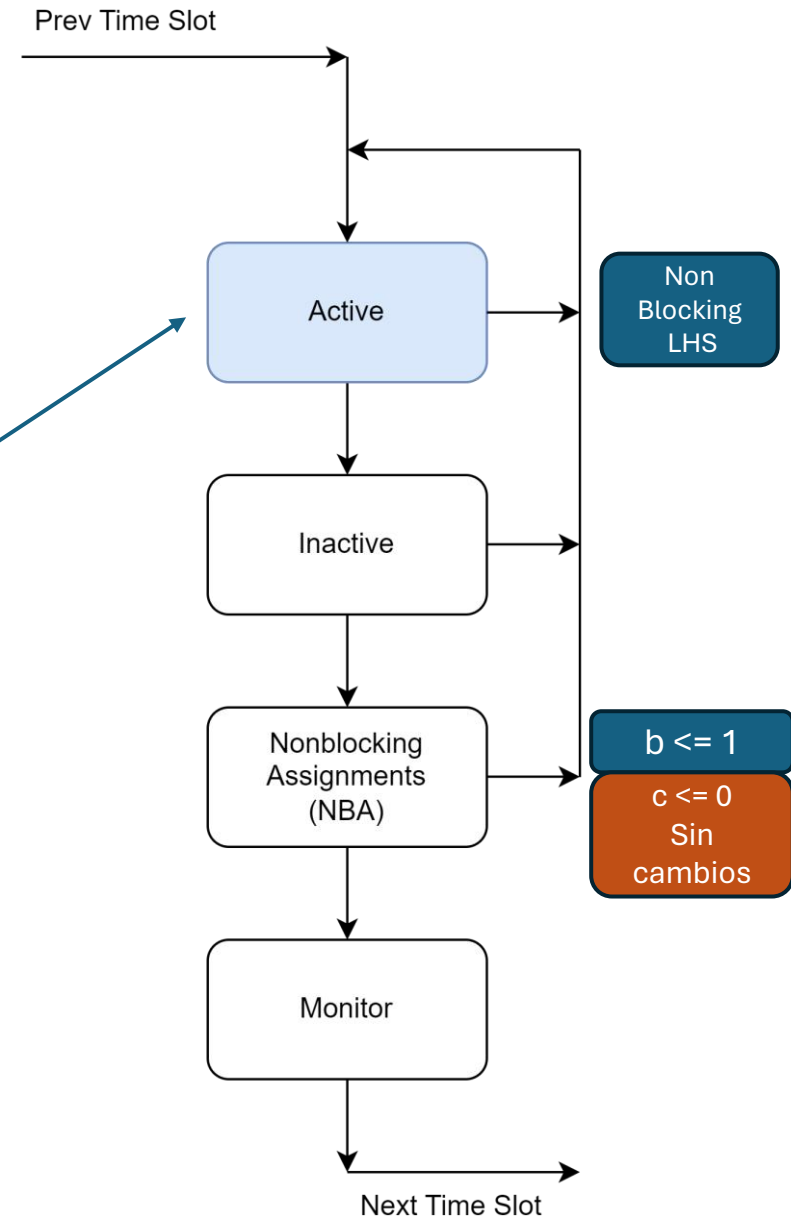
    always @(b) begin : process_2
        d <= b;
    end

    initial begin
        #10;
        $display("\033[32mTEST PASSED\033[0m");
        $finish;
    end

endmodule
```

Reentra a
Active para
ejecutar
process_1
Non Blocking
assignments
b/c scheduled

a = 1
b = 0
c = 0
d = 0



Ciclo de Simulación 4/n

```
module delta_tb;

    reg a = 0, b = 0, c = 0, d = 0;

    initial begin : process_0
        #5;
        a = 1;
    end

    always @(a or b) begin : process_1
        b <= a;
        c <= b;
    end

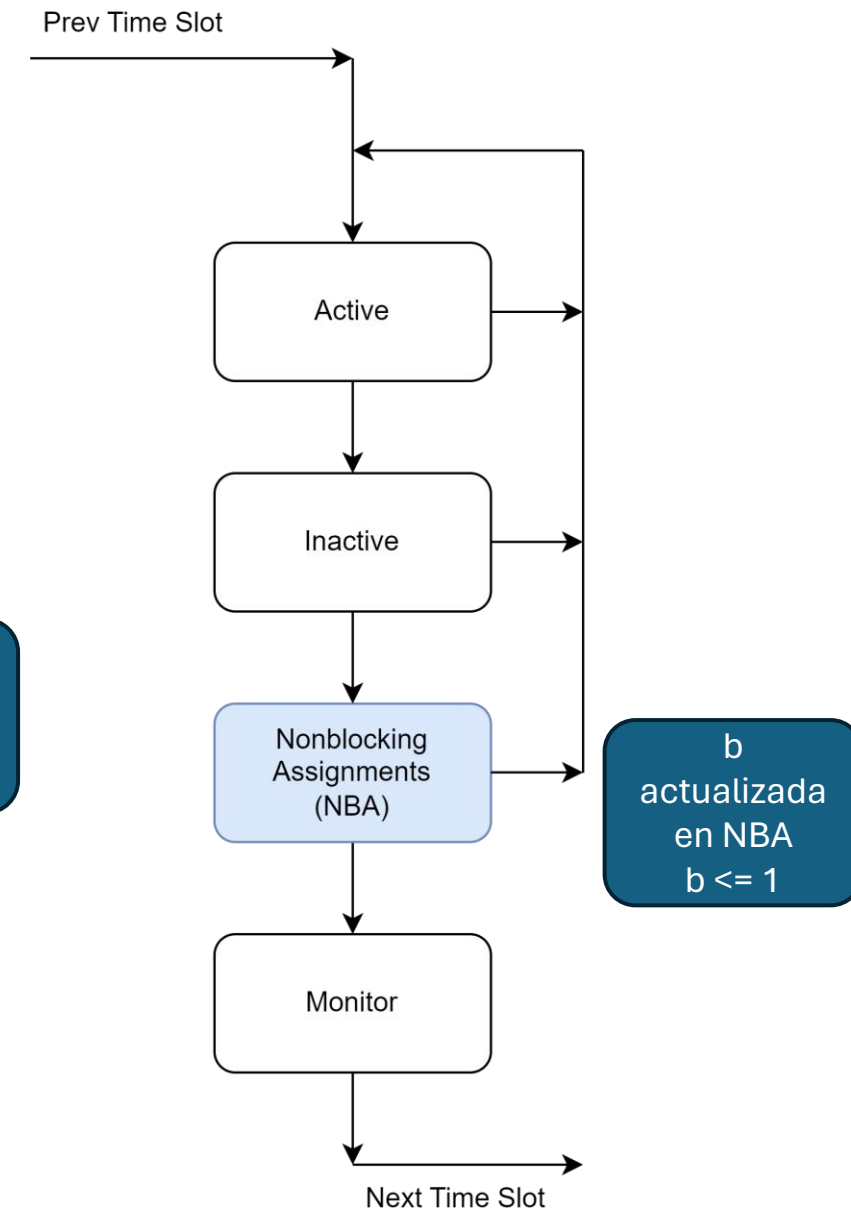
    always @(b) begin : process_2
        d <= b;
    end

    initial begin
        #10;
        $display("\033[32mTEST PASSED\033[0m");
        $finish;
    end

endmodule
```

Cambio en b
dispara
ejecución

a = 1
b = 1
c = 0
d = 0



Ciclo de Simulación 5/n

```
module delta_tb;

    reg a = 0, b = 0, c = 0, d = 0;

    initial begin : process_0
        #5;
        a = 1;
    end

    always @(a or b) begin : process_1
        b <= a;
        c <= b;
    end

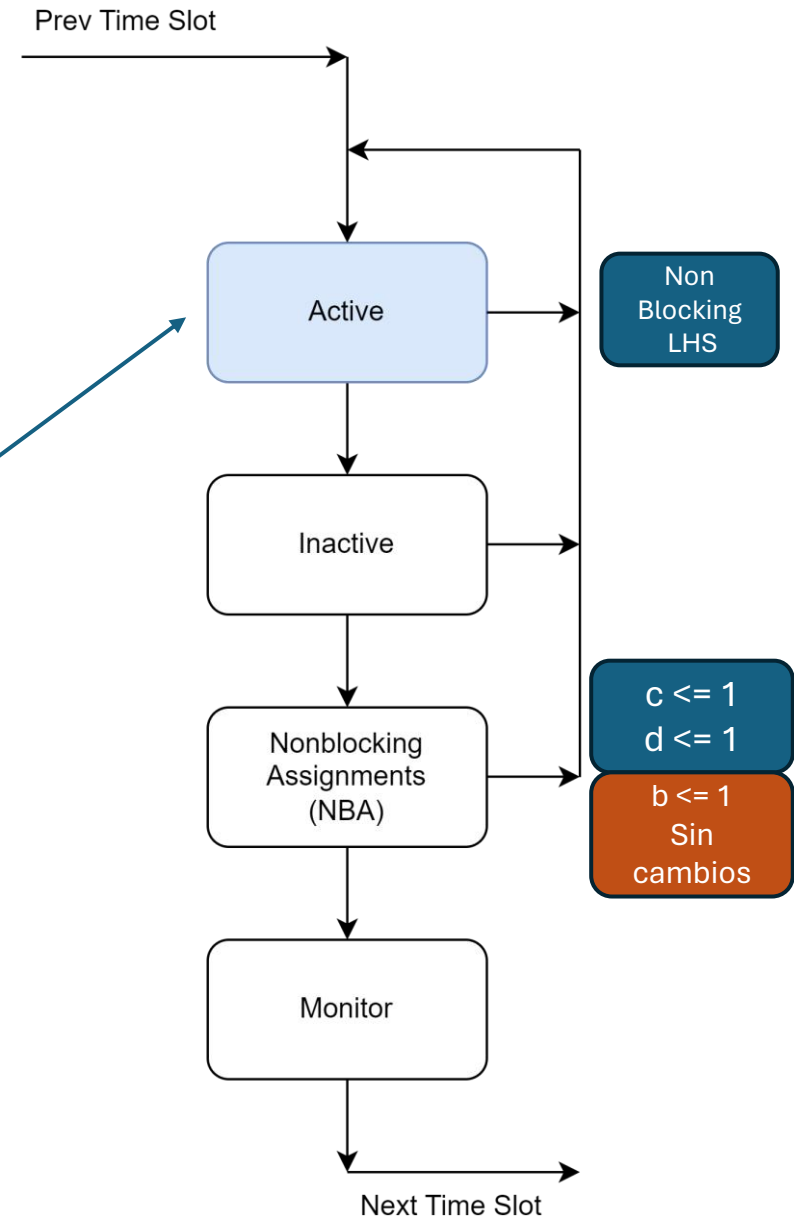
    always @(b) begin : process_2
        d <= b;
    end

    initial begin
        #10;
        $display("\033[32mTEST PASSED\033[0m");
        $finish;
    end

endmodule
```

Reentra a
Active para
ejecutar
process_1/2
Non Blocking
assignments
b/c/d
scheduled

a = 1
b = 1
c = 0
d = 0



Ciclo de Simulación 6/n

```
module delta_tb;

    reg a = 0, b = 0, c = 0, d = 0;

    initial begin : process_0
        #5;
        a = 1;
    end

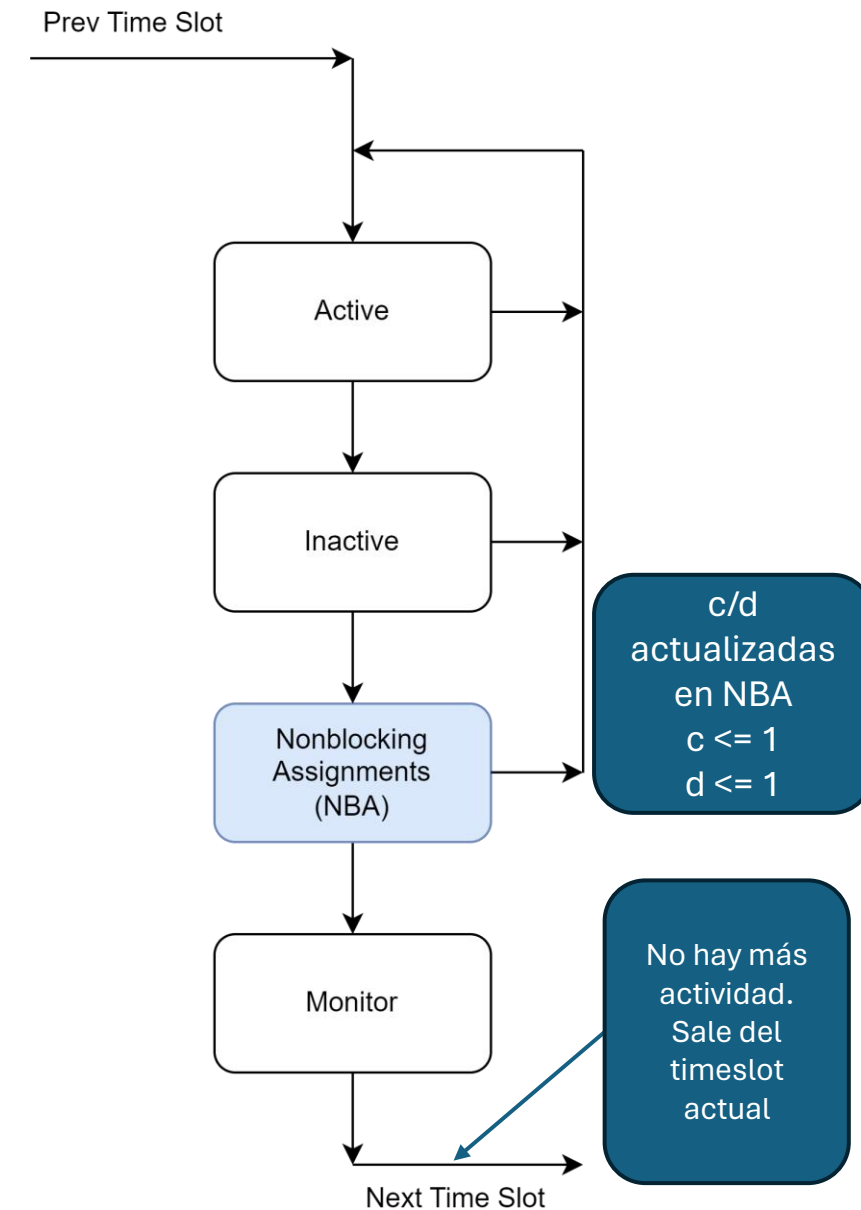
    always @(a or b) begin : process_1
        b <= a;
        c <= b;
    end

    always @(b) begin : process_2
        d <= b;
    end

    initial begin
        #10;
        $display("\033[32mTEST PASSED\033[0m");
        $finish;
    end

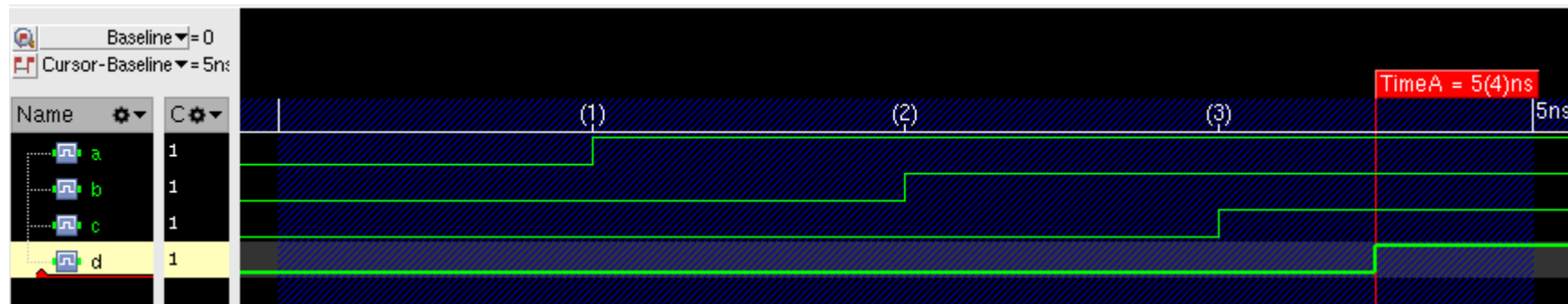
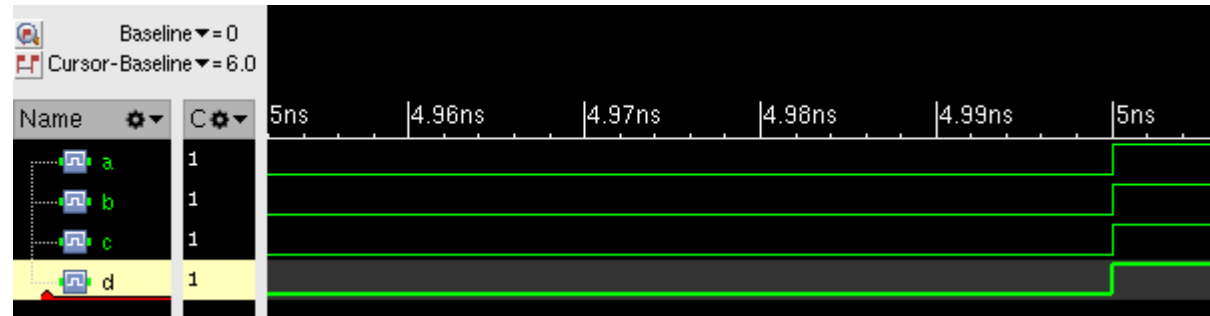
endmodule
```

a = 1
b = 1
c = 1
d = 1



Ciclo de Simulación 7/n

- View -> “Expand Sequence Time”



```
# Probing design section
database -open waves.shm -event
probe -create testbench -depth all -tasks -waveform -functions -all -dynamic -memories -database waves.shm
run
```

Ciclo de Simulación 8/n

```
module delta_tb;

    reg a = 0, b = 0, c = 0, d = 0;

    initial begin : process_0
        #5;
        a = 1;
    end

    always @(a or b) begin : process_1
        b <= a;
        c <= b;
    end

    always @(b) begin : process_2
        d <= b;
    end

    initial begin
        #10;
        $display("\033[32mTEST PASSED\033[0m");
        $finish;
    end

endmodule
```

Ciclo	Evaluación	Evento	Actualiza
N	process_0	a=1	
	process_1		b=1
N+1	process_1		c=1
	process_2		D=1

- process_1 se ejecuta 2 veces para converger
 - **Lógica combinacional -> blocking assignments**

```
reg a = 0, b = 0, c = 0, d = 0;

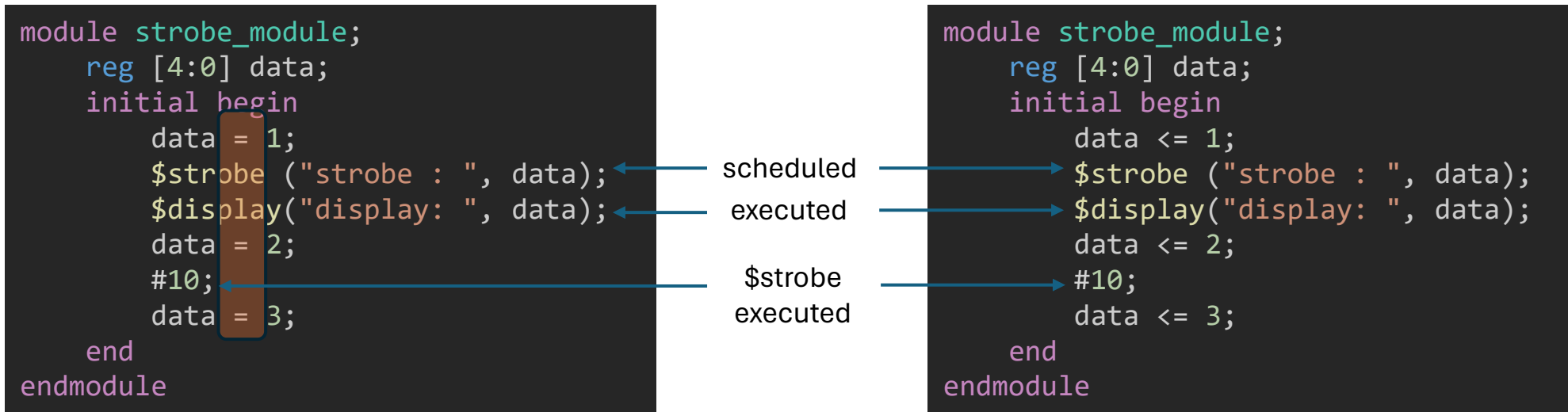
initial begin : process_0
    #5;
    a = 1;
end

always @(a or b) begin : process_1
    b = a;
    c = b;
end

always @(b) begin : process_2
    d = b;
end
```

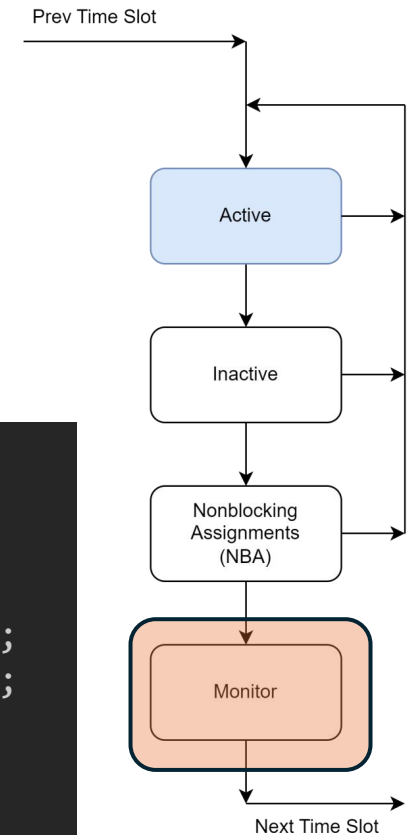
Ciclo de Simulación 9/n

- \$strobe es similar a \$display pero se ejecuta al final del time slice actual



display : 1
strobe : 2

display : x
strobe : 20



Guidelines para evitar race conditions

1. Lógica Secuencial: Usar Non Blocking Assignments \leq
2. Lógica Combinacional en bloque always – Blocking Assignments ($=$)
3. No mezclar lógica secuencial y combinacional en el mismo always block
4. No hacer asignaciones a una misma variable desde mas de un always block
5. Usar \$strobe para mostrar valores que han sido asignados usando nonblocking assignments
6. No usar #0 assignments.

Generate 1/n

- Verilog 2001 agrega nuevas keywords:
 - **generate, endgenerate genvar**
 - Se utilizan dentro de módulos
- Utilizadas para generar código en un módulo de forma dinámica
- Generate_Block_name (opcional) se usa para crear un nombre único para la instancia generada
- Genvar es un valor entero positivo, utilizado únicamente dentro del bloque generate
- Generación Condicional (case, if)
- Generación Iterativa (for)

Generate 2/n

- Generación Condicional IF:

- Instancias, funciones, tasks, variables y procedural blocks
- Label opcional

```
generate
  if ((a_width < 8) || (b_width < 8)) begin: mult
    CLA_mult #(a_width, b_width) u1(a,b,product);
  end
  else begin: mult
    WALLACE_mult #(a_width, b_width) u1(a,b,product);
  end
endgenerate
```

mult.u1

- Generación Condicional CASE:

- Instancias, funciones, tasks, variables y procedural blocks
- Label opcional

```
generate
  case (WIDTH)
    1: begin: adder // 1-bit adder
        adder_1b x1(co, sum, a, b, ci);
      end
    2: begin: adder // 2-bit adder
        adder_2b x1(co, sum, a, b, ci);
      end
    default:
      begin: adder // Otro - CLA
        adder_cla #(WIDTH) x1(co, sum, a, b, ci);
      end
  endcase
endgenerate
```

adder.x1

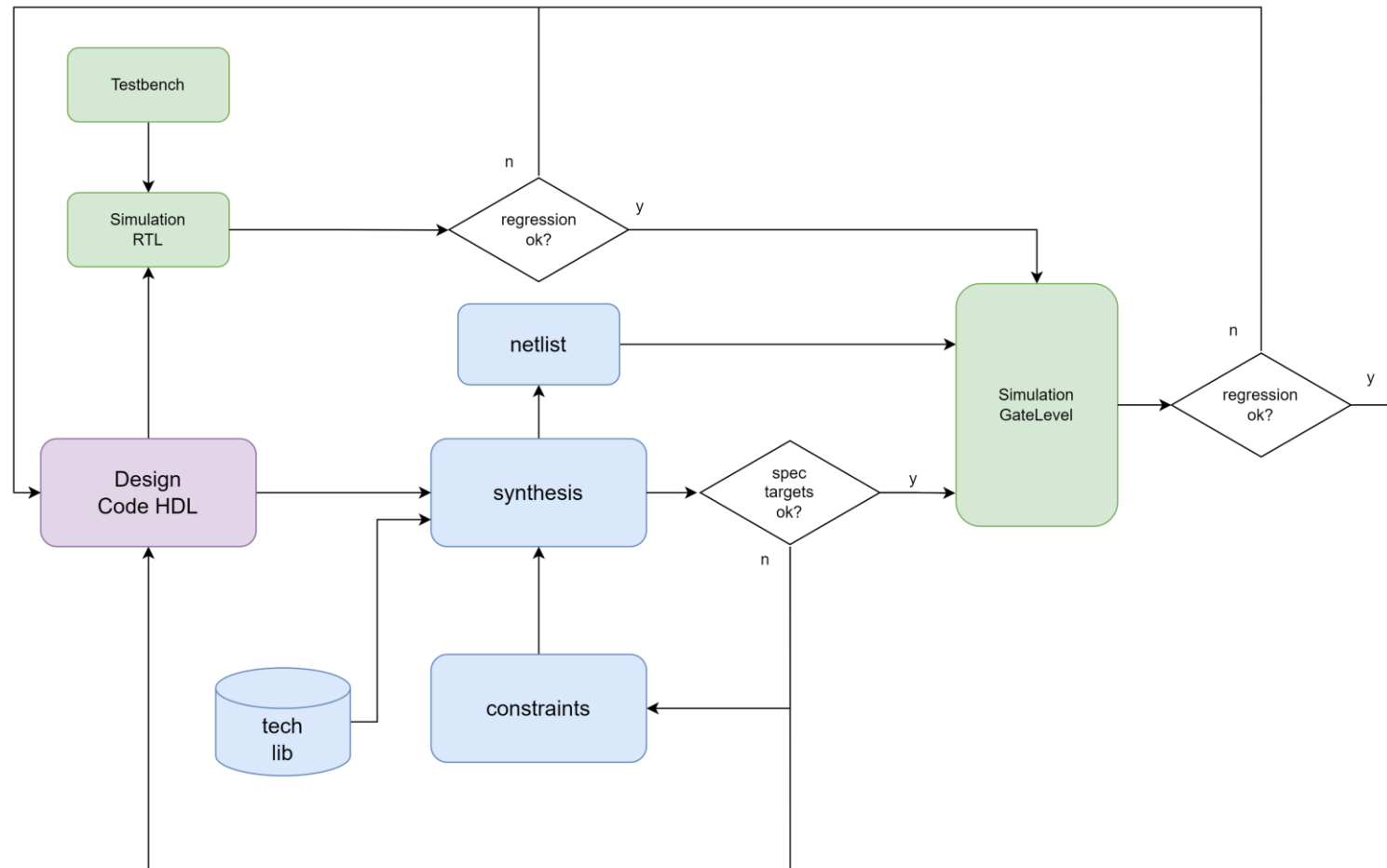
Generate 3/n

- Generación Iterativa FOR:
 - Instancias, variables y procedural blocks
 - Label requerida

```
module gray2bin1 #(
    parameter SIZE=2
) (
    input  wire [SIZE-1:0] gray,
    output wire [SIZE-1:0] bin
);

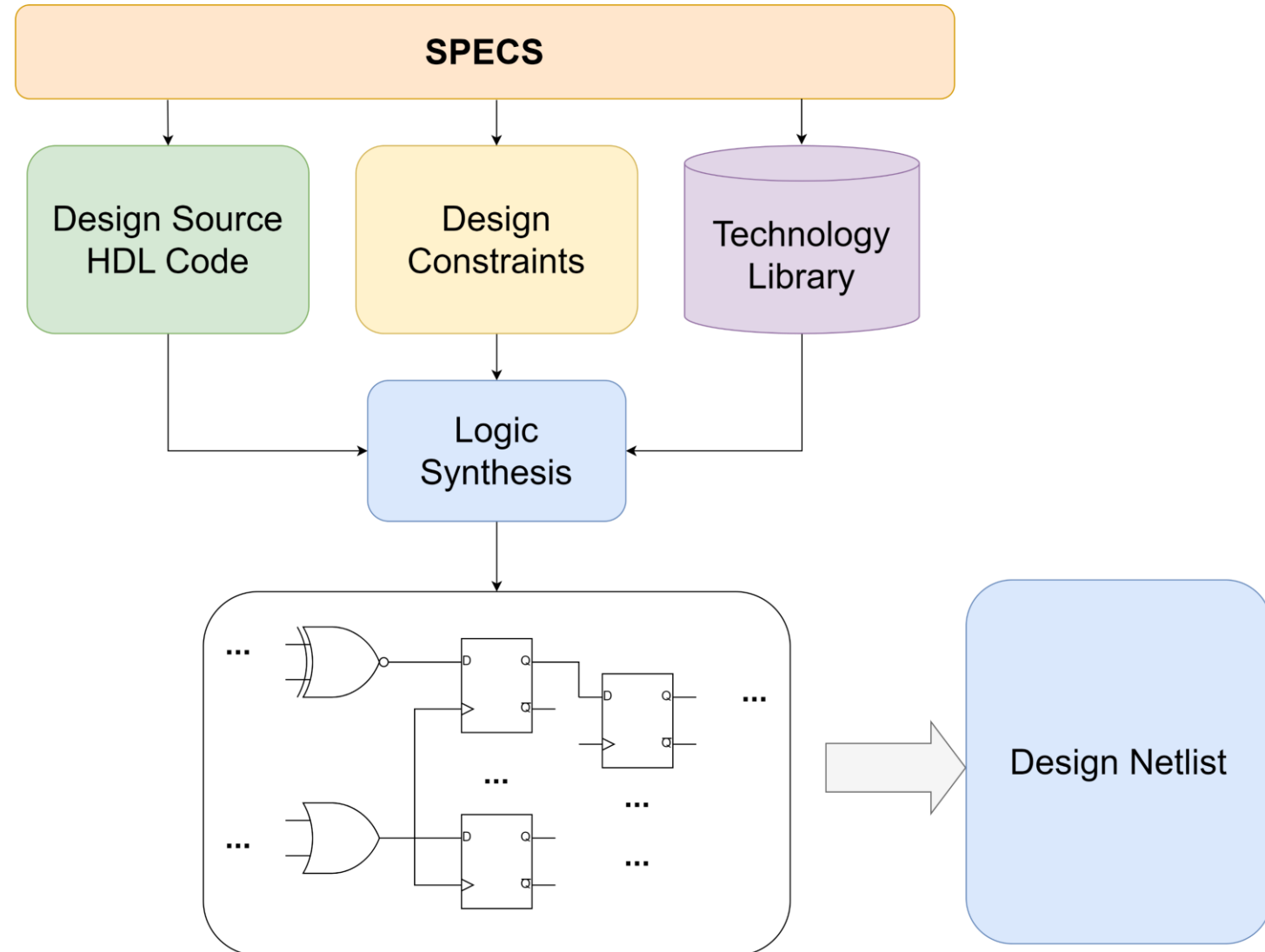
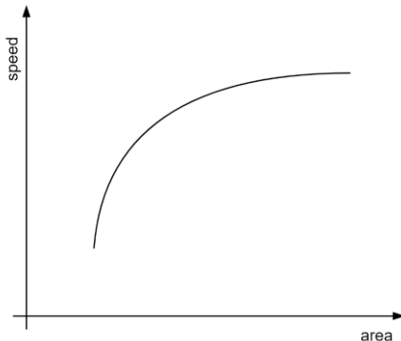
    genvar i;
    generate
        for (i=0; i<SIZE; i=i+1) begin: bitnum
            // Label requerida -> naming
            assign bin[i] = ^gray[SIZE-1:i];
        end
    endgenerate
endmodule
```

Flujo de Trabajo

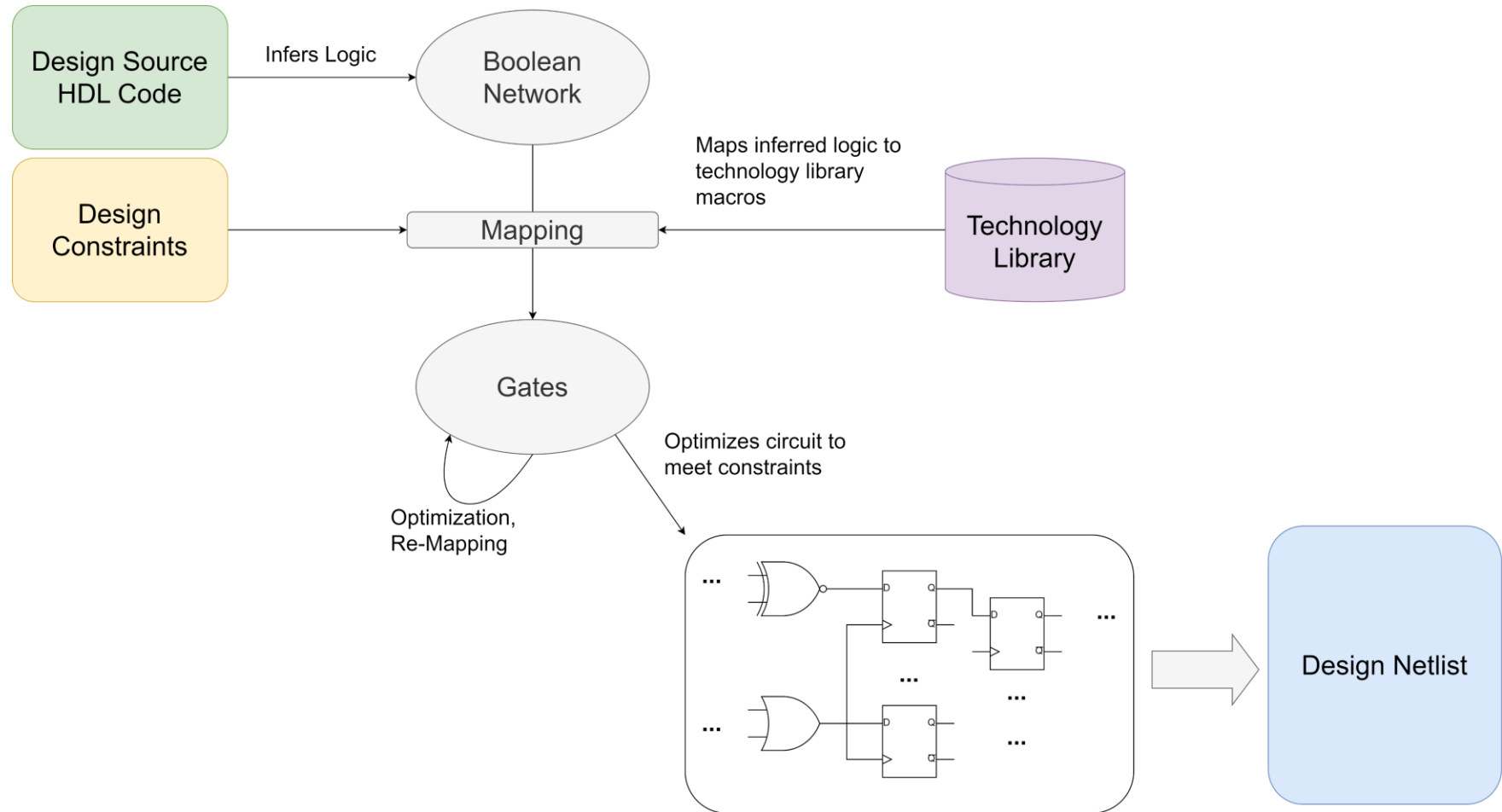


Síntesis Lógica

- La herramienta de síntesis:
 - infiere la lógica del código HDL
 - Mapea la lógica a una tech lib
 - Optimiza el circuito para cumplir con los objetivos de diseño.



Síntesis Lógica



Modelado Lógica Combinacional

- Lógica Combinacional: La salida es, en todo momento, una función combinacional únicamente de las entradas

- Net declaration assignment
- Continuous Assignment
- Always Statement

```
wire w = expressions;  
wire w; assign w = expression;  
reg r; always @* r = expression;
```

- La lista de eventos no debe contener eventos posedge o negedge
 - Incluir TODAS las entradas (o *) a fin de evitar mismatch entre pre y post síntesis
- Usar blocking assignments (=): Son suficientes y simulan de manera más eficiente

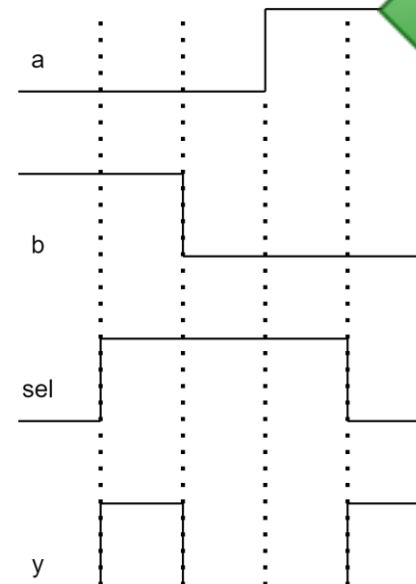
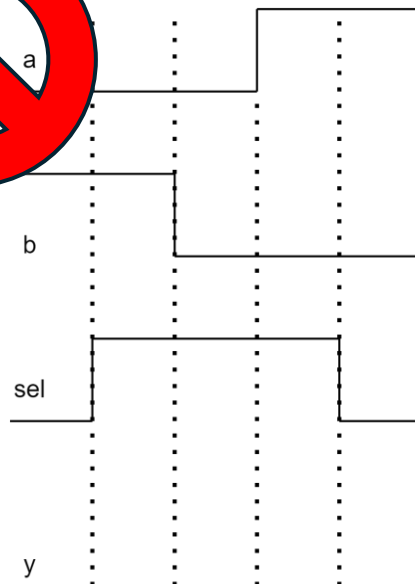
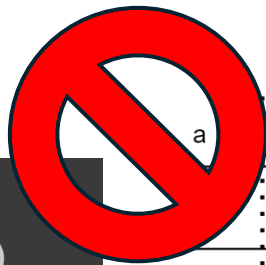
```
always @( TODAS LAS ENTRADAS ) begin  
    blocking assignments  
end
```



Lista de Sensibilidad incompleta

- Es necesario incluir en la lista de sensibilidad (o de eventos) todas las señales que son entradas a la lógica
 - La lista de sensibilidad no afecta la síntesis, pero debemos evitar valores incorrectos en la simulación
 - “The event list does not affect the synthesized netlist.” – IEEE Std. 1364.1-2002 5.1 Modeling combinational logic
- Si utilizamos *, automáticamente se incluyen todas las entradas necesarias (Verilog-2001)

```
// Incompleta
always @(a or b)
begin
    y = a;
    if (sel)
        y = b;
end
```



```
// Completa
always @*
begin
    y = a;
    if (sel)
        y = b;
end
```


Assignments incompletos

- **Lógica Combinacional:** La salida combinacional es, en todo momento, una función combinacional únicamente de las entradas.

```
// Assignment incompletos
always @(a or b)
    if (b)
        y = a;

always @(a or b)
    case (b)
        1: y = a;
    endcase
```

Fix para
evitar que
la síntesis
infera
latches

```
// Else Explicito
always @(a or b)
    if (b)
        y = a;
    else
        y = 1'b0;
```

```
// Default
always @(a or b)
    y = 1'b0;
    if (b)
        y = a;
```

```
// Valor Explicito
always @(a or b)
    case (b)
        1: y = a;
        default: y = 1'b0;
    endcase
```

```
// Valor Default
always @(a or b)
    y = 1'b0;
    case (b)
        1: y = a;
    endcase
```

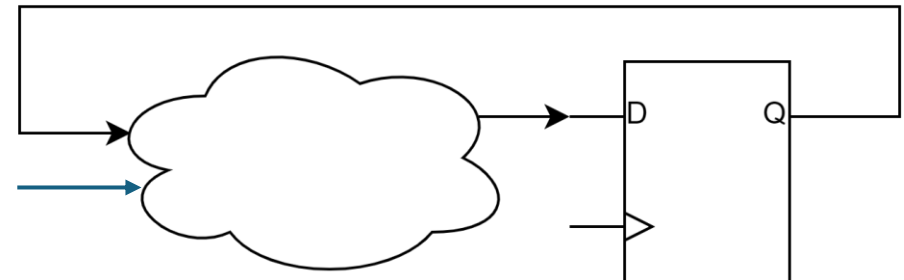
Modelado Lógica resumen

- Continuous assignments siempre sintetizan lógica combinacional.
- Si se utiliza un always block, la lista de sensibilidad tiene que tener todas las entradas o bien usar *. Always @*
- Agrupar múltiples sentencias entre begin/end .
- Usar blocking assignments. =
- Agregar assignments por default para evitar la inferencia de latches.
- Evitar loops combinacionales
- Asignar una variable en solo un procedimiento.

Modelado Lógica Secuencial

- Las salidas de los registros se muestrean en un flanco de reloj
- Se modelan con un bloque always:
 - La lista de sensibilidad (eventos) debe tener solo eventos de posedge/negedge
 - Uno representa el flanco activo del clock y otros representan set/reset async
 - Utilizar NON Blocking assignment <=

```
always @( clock edges )  
begin  
  nonblocking assignments <=  
end
```



Modelado Lógica Secuencial

- Reset/Set
 - Utilizar **if..else** para agregar set/reset al bloque always.
 - Para async reset/set, agregar los eventos correspondientes a la lista de sensibilidad.

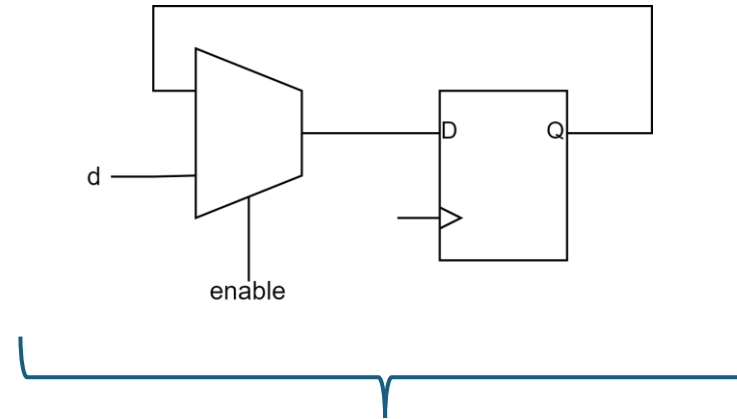
```
// Asynchronous reset
always @(posedge clk or posedge rst)
    if (rst)
        count <= 4'd0;
    else
        if (count == 9)
            count <= 4'd0;
        else
            count <= count + 4'd1;
```

```
// Synchronous reset
always @(posedge clk)
    if (rst)
        count <= 4'd0;
    else
        if (count == 9)
            count <= 4'd0;
        else
            count <= count + 4'd1;
```

Modelado Lógica Secuencial

- En lógica secuencial, la salida no es en todo momento, una función combinacional de sus entradas
 - Esto implica “memoria”
- Assignments incompletos en este caso, no infieren latches.
 - El almacenamiento ya está presente!

```
always @(posedge clk)
  if (enable)
    q <= d;
```

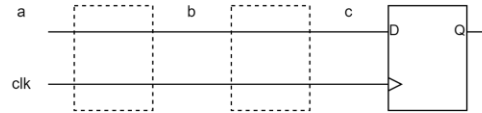


Posible implementación

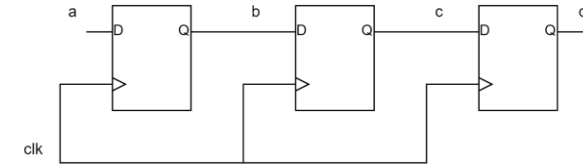
Modelado Lógica Secuencial

- Si se usa blocking assignment, el orden importa.
 - Con NonBlocking No.
- Usar siempre NonBlocking en lógica secuencial
- “Nonblocking procedural assignments should be used for variables that model edge-sensitive storage devices.” – IEEE Std. 1364.-2002 Section 5.2.2 Modeling edge-sensitive storage devices

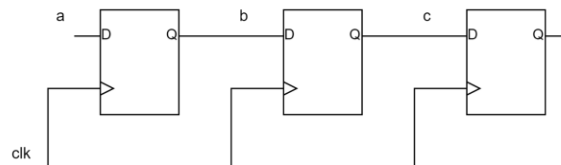
```
always @(posedge clk)
begin
    b = a;
    c = b;
    d = c;
end
```



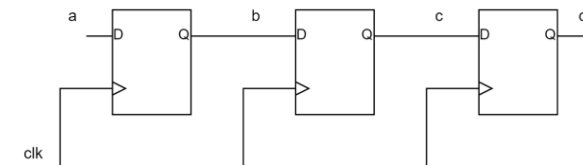
```
always @(posedge clk)
begin
    d = c;
    c = b;
    b = a;
end
```



```
always @(posedge clk)
begin
    b <= a;
    c <= b;
    d <= c;
end
```



```
always @(posedge clk)
begin
    d <= c;
    c <= b;
    b <= a;
end
```




Modelado Lógica Secuencial Resumen

- Utilizar un bloque always cuya lista de sensibilidad:
 - Use únicamente eventos posedge/negedge
 - Incluir un único clock y async reset/set en caso de ser necesarios.
- Agrupar las sentencias mediante begin/end
- Utilizar NON Blocking Assignments <=
- Asignar una variable en solo un procedimiento.


Simulation Mismatch

- Evitar asignación en la declaración: Síntesis las ignora
- Generalmente, el HW no enciende mágicamente en un estado conocido. Hay que proporcionar un camino explícito de inicialización.

```
reg [3:0] counter = 'd0;  
always @(posedge clk) begin  
    if (counter == 4'd10) begin  
        counter <= 4'd0;  
    end else begin  
        counter <= counter + 4'd1;  
    end  
end
```



```
reg [3:0] counter;  
always @(posedge clk or negedge rst_n)  
begin  
    if (!rst_n) begin  
        counter <= 4'd0;  
    end else begin  
        if (counter == 4'd10) begin  
            counter <= 4'd0;  
        end else begin  
            counter <= counter + 4'd1;  
        end  
    end  
end
```



Evitar Comportamiento Indeterminado

- El estándar de Verilog permite que el simulador:
 - Ejecute los procesos en cualquier orden
 - Ejecute sentencias contiguas como múltiples eventos, lo que potencialmente permite entrelazar la ejecución de sentencias de múltiples bloques
- “At any time while evaluating a behavioral statement, the simulator may suspend execution and place the partially completed event as a pending active event on the event queue. The effect of this is to allow the interleaving of process execution. Note that the order of interleaved execution is nondeterministic and not under control of the user.” – IEEE Std. 1364-2001 Section 5.4.2 Nondeterminism

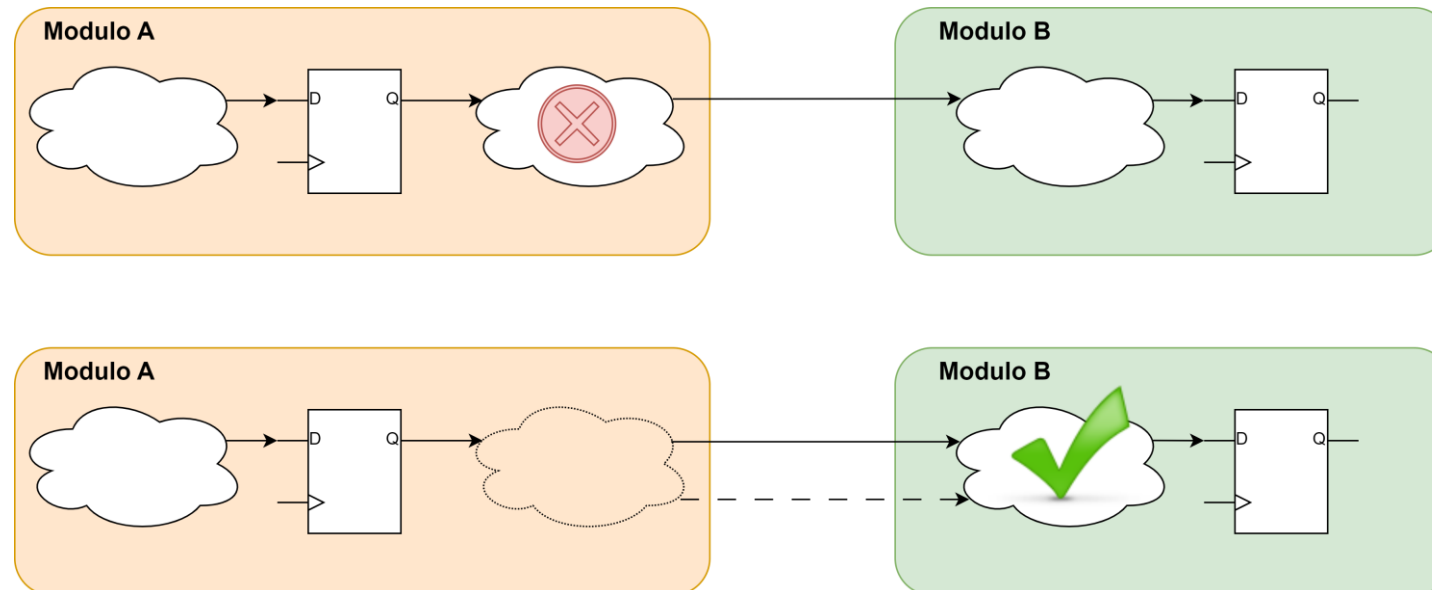
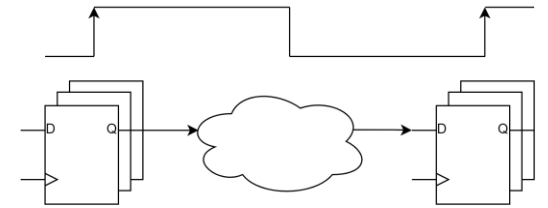
```
always @(posedge clk) begin
    b = a;
    c = b;
end
always @(posedge clk) begin
    d = b ^ c;
end
```

b = a;	d = b ^ c;	b = a;
c = b;	b = a;	d = b ^ c;
d = b ^ c;	c = b;	c = b;

Secuencias de ejecución potenciales

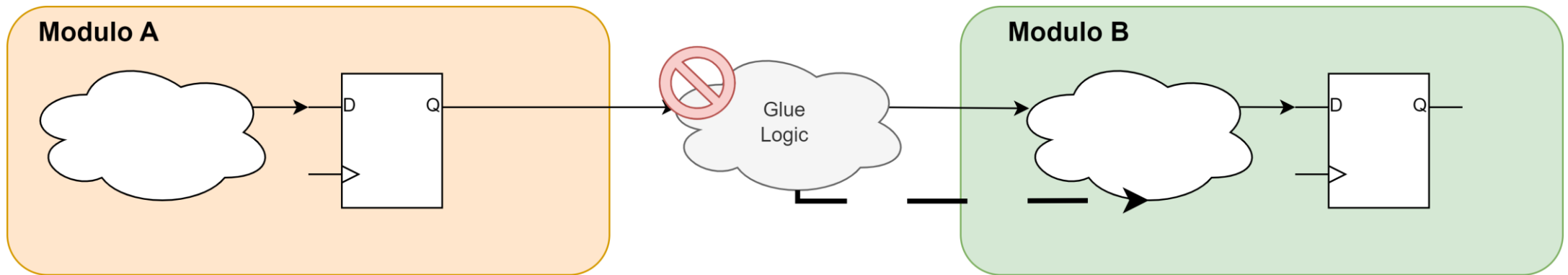
Particionado para síntesis 1/n

- Registrar las salidas del módulo
 - Las constraints son simples e idénticas para cada módulo
 - Arrival time de input/outputs se definen fácil:
 - Input drive strength -> FF anterior
 - Input arrival time -> Es el delay del path a través del FF anterior



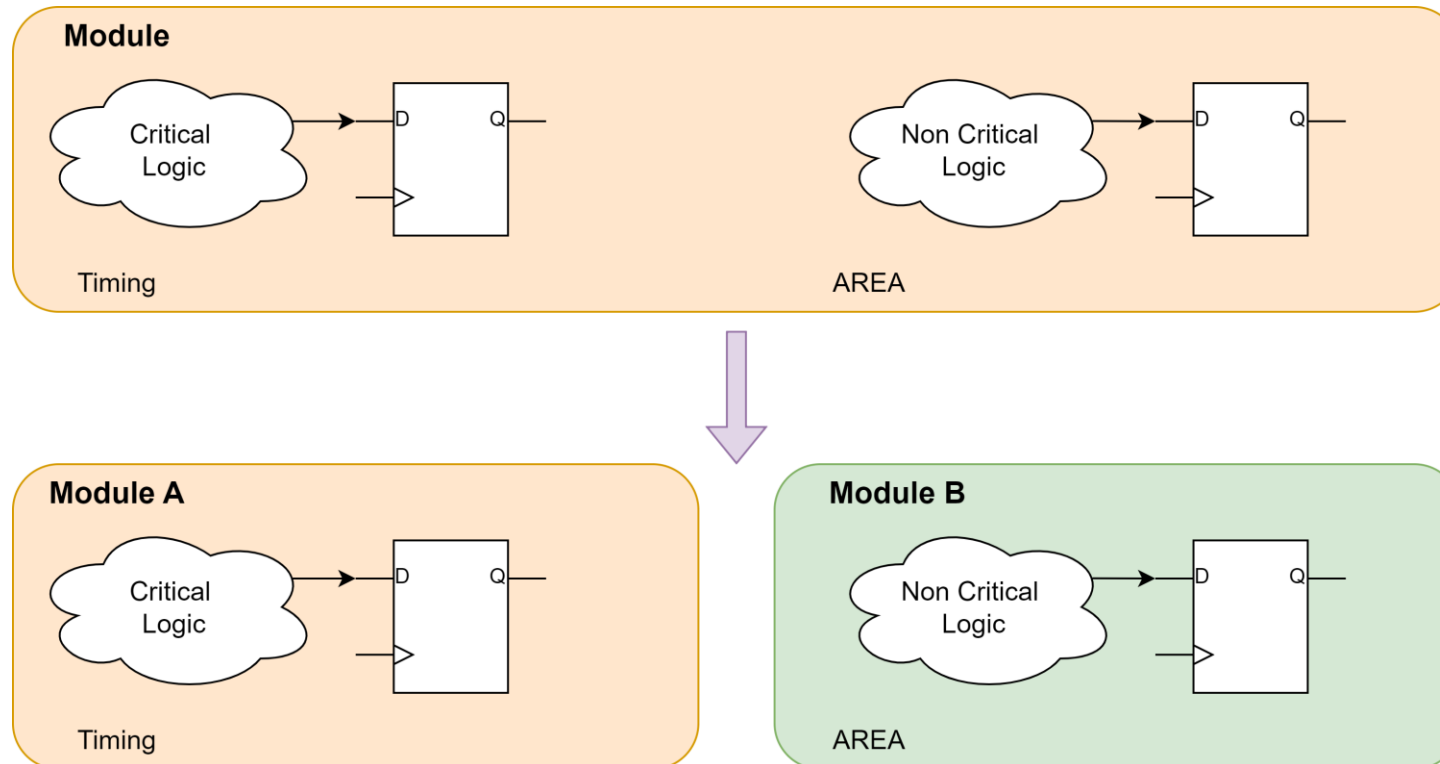
Particionado para síntesis 2/n

- Evitar “Glue Logic” on top.
 - Las herramientas de síntesis generalmente (excepto inversores) no mueven lógica entre jerarquías.
 - Si hay “glue logic” no pueden optimizar correctamente



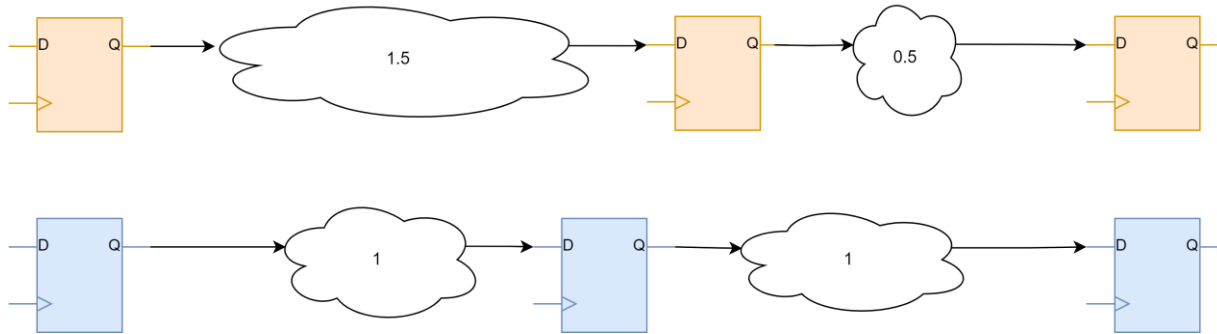
Particionado para síntesis 2/n

- Separar los bloques que requieran diferentes estrategias de síntesis.
 - Separar módulos de área crítica respecto de módulos críticos en timing

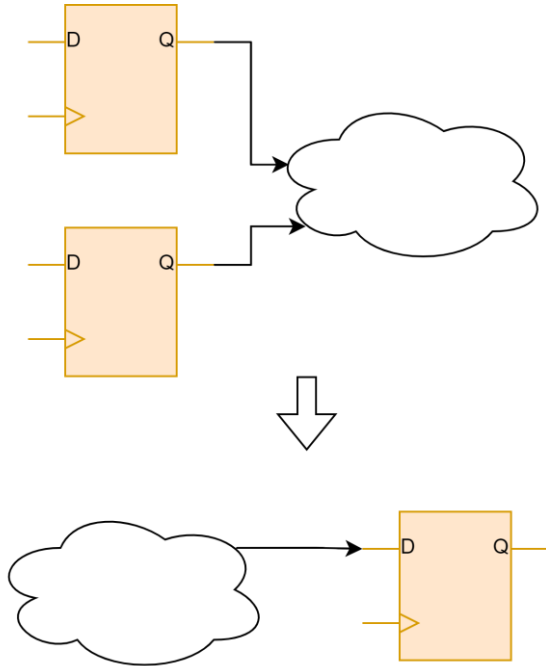


Register Retiming

- Desplaza los registros para mejorar los resultados
 - Reducir cycle time o área sin cambiar input-to-output latency
 - Mejora el “timing slack”



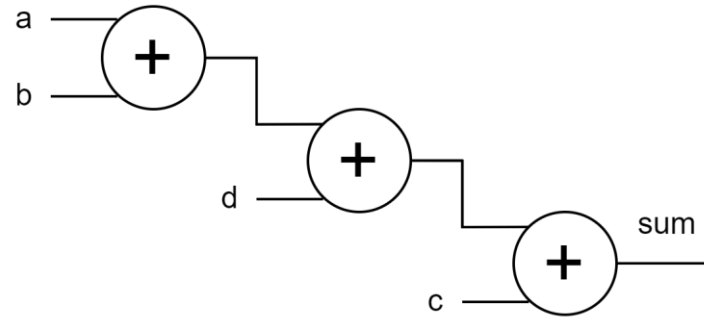
Minimizar Delay



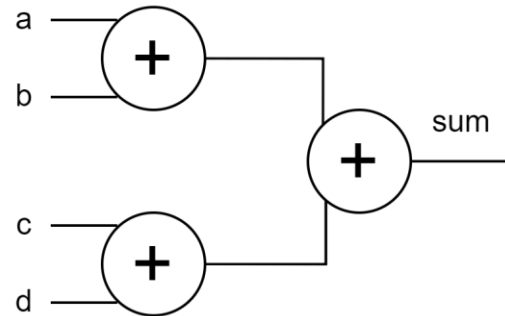
Minimizar Area

Optimización Expresiones Aritméticas

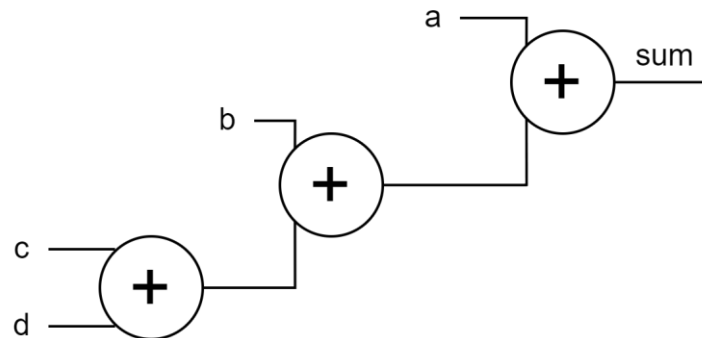
Implementación inicial



Speed Optimized: si todas las inputs llegan igual

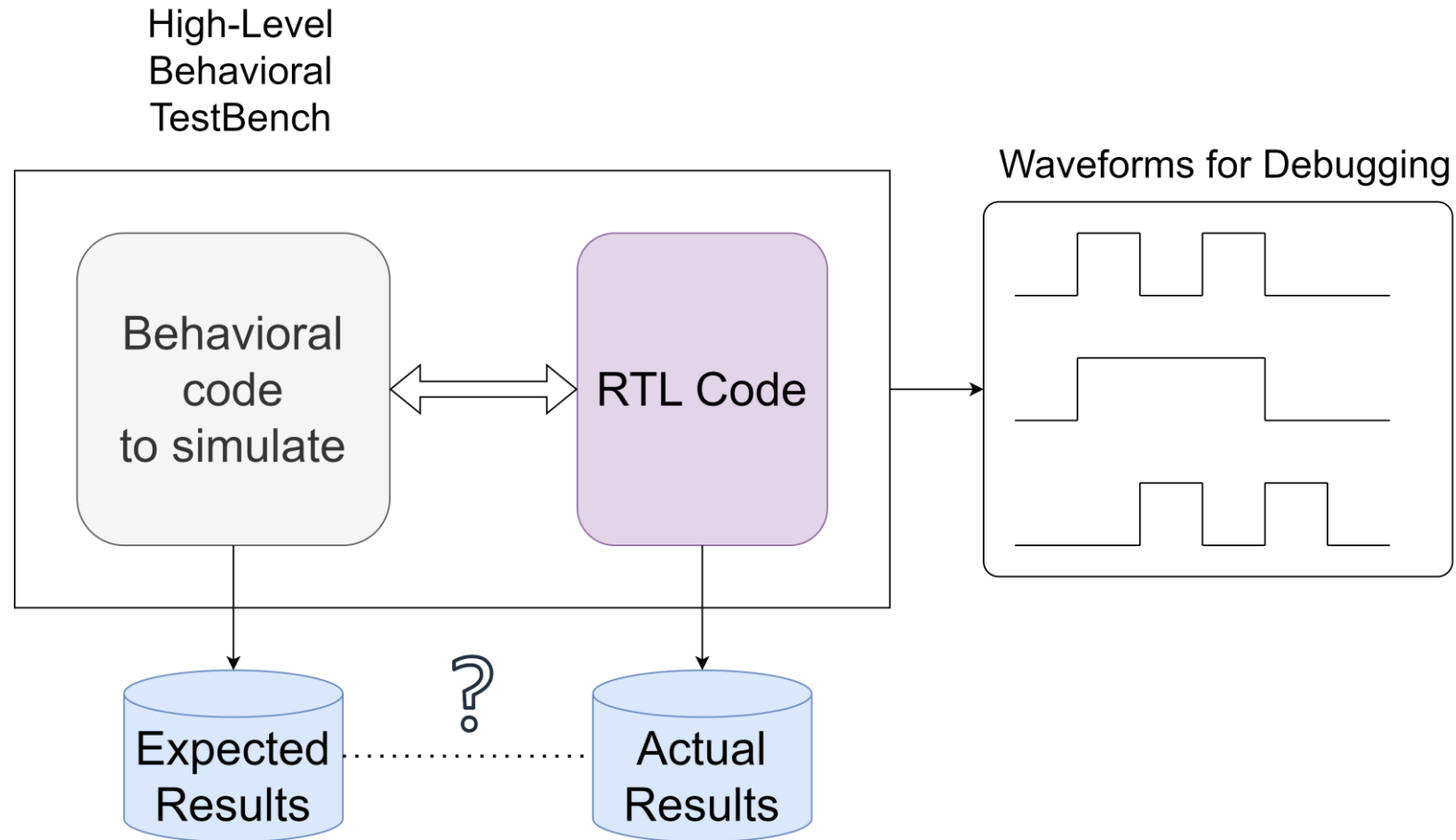


Speed Optimized: si la entrada "a" llega mas tarde



```
sum = a + b + c + d;
```

HDL-Based Simulation



TB: Generar Clock

- Behavioral Clocks

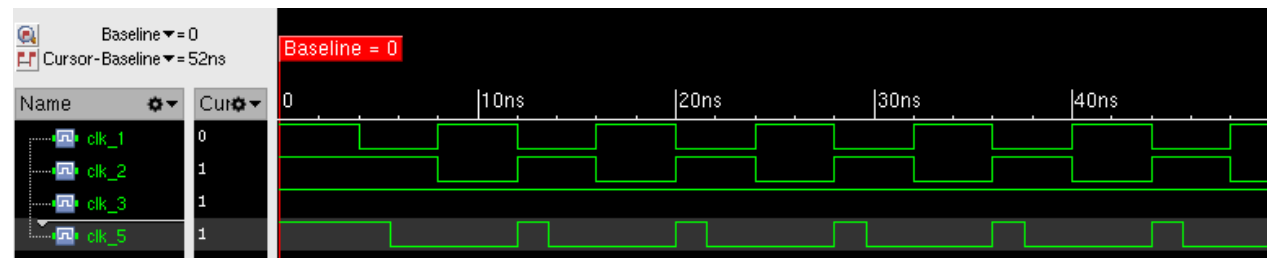
```
always
    #(PERIOD/2) clk_1 = ~clk_1;

initial begin
    #(DELAY) forever
        #(PERIOD/2) clk_2 = ~clk_2;
end

// IRREGULAR
initial begin
    # (DELAY) forever begin
        # (( DUTY) * PERIOD) clk_5 = 0;
        # ((1-DUTY) * PERIOD) clk_5 = 1;
    end
end
```

```
// NO USAR!
always @(clk_3)
    #5 clk_3 = ~clk_3; // No oscila!

// DANGER!
always
    clk_4 <= #5 !clk_4; // Zero Delay Loop
```



Conclusiones

