WORKSHOP

Introducción al Diseño Digital con EDU-CIAA-FPGA

Aquí es donde todo comienza ...

Circuitos secuenciales

Contenido

- Definición de circuito secuencial.
- Componentes del sistema secuencial (entradas, estado interno, lógica de estado siguiente, lógica de salida).
- Ejemplos con bloques comunmente utilizados: FFD, registros, contadores.
- Finite States Machines: definición, tipos de FSM y ejemplo **arbiter**.
- Actividades propuestas:
  - Actividad 1: diseño del contador universal.
  - Actividad 2: Arbiter Game
- Anexo 1: asignaciones bloqueantes vs no bloqueantes.

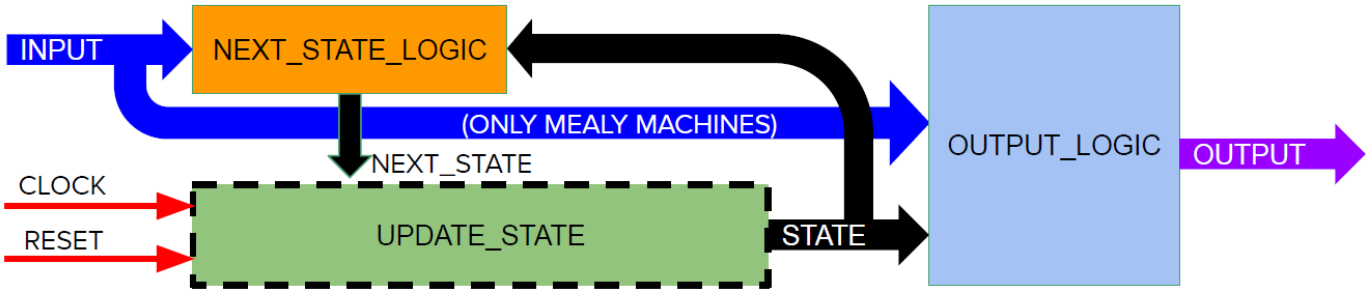
Definición

Definición

Un circuito secuencial es aquel circuito que **utiliza elementos de memoria** para almacenar un **estado interno**. A diferencia de los circuitos combinacionales, donde la salida depende únicamente de sus entradas; las **salidas de los circuitos secuenciales** dependerán de su **estado interno** y (en algunos casos) de las **entradas**.

Para diseñar circuitos secuenciales es común emplear una **metodología sincrónica**, donde existe una **señal de clock** que es utilizada para **actualizar los elementos de memoria** de forma sincronizada. Los elementos de memoria serán entonces actualizados con el flanco ascendente o descendente del clock.

Con lo presentado hasta aquí, se pueden listar los elementos que componen un circuito secuencial:
- Entradas
- Memoria del estado interno
- Lógica del estado siguiente
- Lógica de salida



Definición

Según las características de la lógica del estado siguiente, los circuitos secuenciales se pueden caracterizar como:

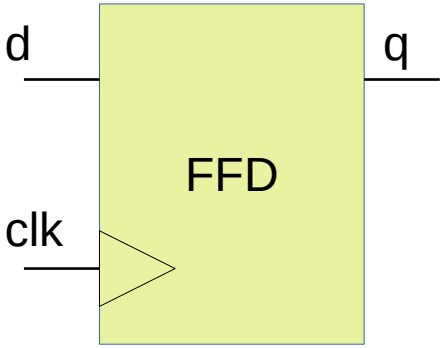
- **Regulares**: las transiciones entre estados exhiben un patrón regular, por ejemplo: contadores, registros de desplazamiento, etc.
- **FSM** (*Finite States Machine*): no hay un patrón regular para las transiciones entre estados. Ejemplo: detector de secuencia o trama, antirebote, árbitro, entre otros tantos.
- **FSMD** (*Finite States Machine with Datapath*): formada por un circuito regular (Data Path) y una FSM (Control Path).

Flip Flop D

Componente elemental: Flip-Flop D

```
module ffd
(
   input  wire clk,
   input  wire   d,
   output reg    q
);

always @ (posedge clk)
   q <= d;

endmodule
```

d	clk	q*
1	_/--	1
0	_/--	0



Componente elemental: Flip-Flop D

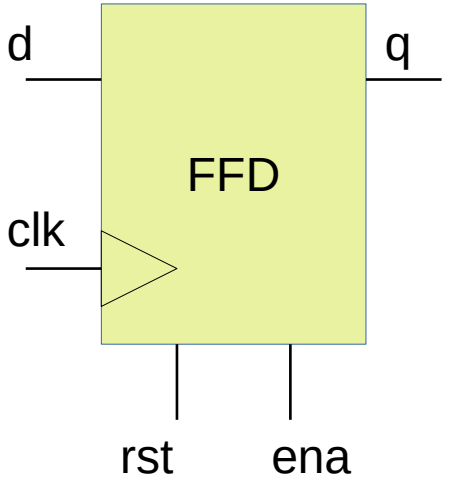
```
module ffd
(
   input  wire clk,
   input  wire   d,
   output reg    q
);

always @ (posedge clk)
   q <= d;

endmodule
```

Importante: notar el uso de asignaciones **no bloqueantes (q <= d)** para **inferir** correctamente un elemento de **memoria**.

Componente elemental:
Flip-Flop D con reset asincrónico y enable

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

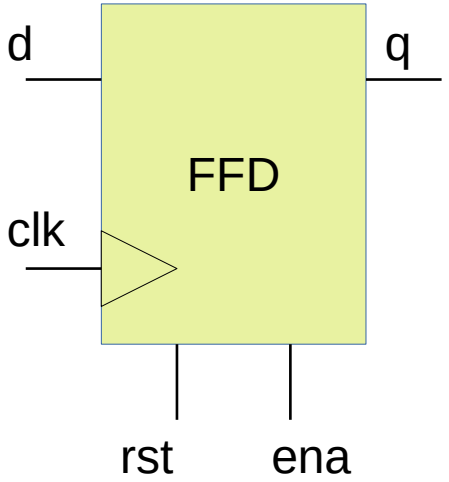
rst	d	ena	clk	q*
_/--	X	X	X	0
0	X	0	X	q
0	1	1	_/--	1
0	0	1	_/--	0



Componente elemental:
Flip-Flop D con reset sincrónico y enable

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

rst	d	ena	clk	q*
1	X	X	_/--	0
0	X	0	X	q
0	1	1	_/--	1
0	0	1	_/--	0



FFD - Síntesis

Reportes de síntesis tomados de **vscode_stdout.log**:

ffd

```
=== ffd ===

   Number of wires:                 3
   Number of wire bits:             3
   Number of public wires:          3
   Number of public wire bits:      3
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 1
     SB_DFF                         1
```

ffd_arst_en

```
=== ffd_arst_en ===

   Number of wires:                 5
   Number of wire bits:             5
   Number of public wires:          5
   Number of public wire bits:      5
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 1
     SB_DFFER                       1
```

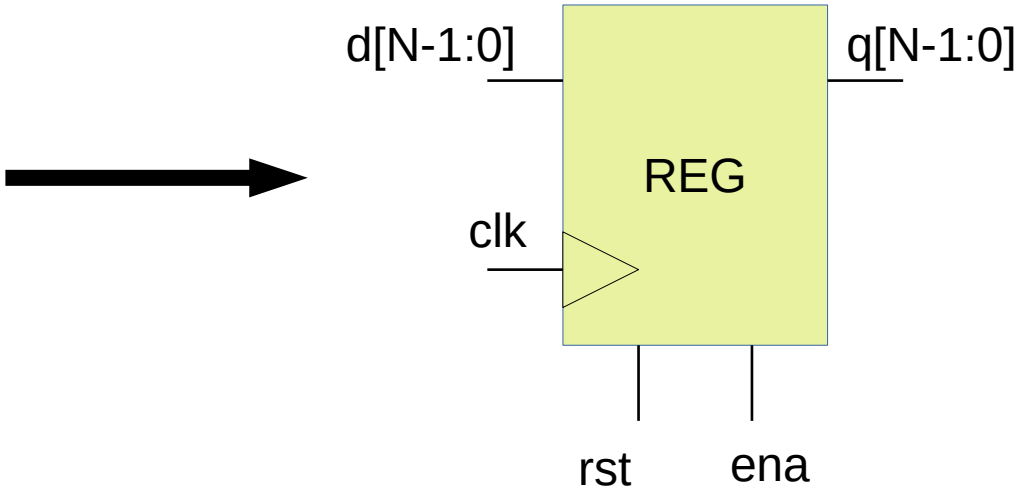
ffd_srst_en

```
=== ffd_srst_en ===

   Number of wires:                 6
   Number of wire bits:             6
   Number of public wires:          6
   Number of public wire bits:      6
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 2
     SB_DFFESR                      1
     SB_LUT4                        1
```

Registros

Registros

```
module register
#(
   parameter N = 8
)
(
   input  wire              clk,
   input  wire [N-1:0]      d,
   input  wire              rst,
   input  wire              ena,
   output reg  [N-1:0]      q
);
always @ (posedge clk, posedge rst)
begin
   if(rst)
       q <= {N{1'b0}};
   else if(ena)
       q <= d;
end
endmodule
```



Registros: síntesis

```
module register
#(
   parameter N = 8
)
(
   input  wire              clk,
   input  wire [N-1:0]      d,
   input  wire              rst,
   input  wire              ena,
   output reg  [N-1:0]      q
);
always @ (posedge clk, posedge rst)
begin
   if(rst)
       q <= {N{1'b0}};
   else if(ena)
       q <= d;
end
endmodule
```

N=8

```
=== register ===

   Number of wires:                 5
   Number of wire bits:            19
   Number of public wires:          5
   Number of public wire bits:     19
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 8
     SB_DFFER                       8
```

N=64

```
=== register ===

   Number of wires:                 5
   Number of wire bits:           131
   Number of public wires:          5
   Number of public wire bits:    131
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                64
     SB_DFFER                      64
```

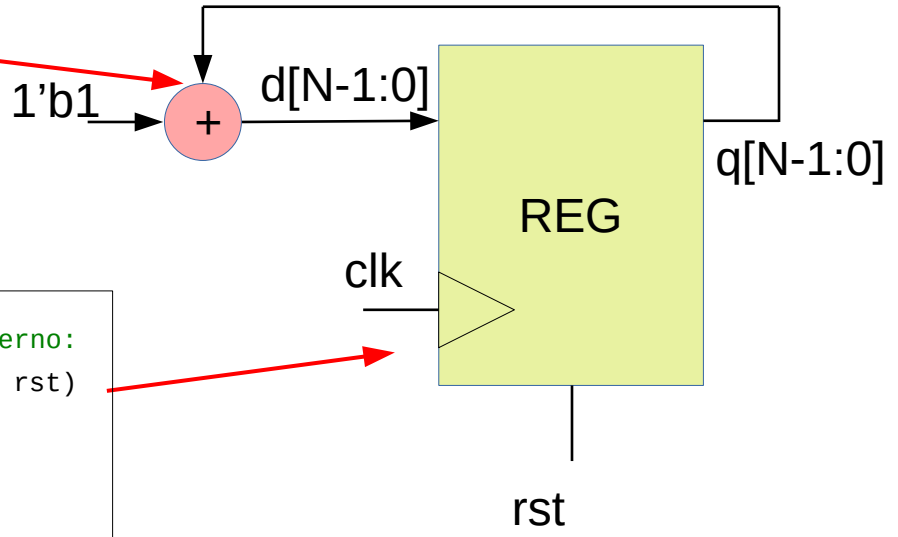
Contadores

Contadores

```
module counter
#(
   parameter N = 8
)
(
   input wire clk,
   input wire rst,
   output max_tick,
   output [N-1:0] q
);

//Declaración de señales internas
reg  [N-1:0] r_reg;
wire [N-1:0] r_next;
//Actualización del estado interno:
always @ (posedge clk, posedge rst)
begin
   if(rst)
       r_reg <= 0;
   else
       r_reg <= r_next;
end
//Lógica del estado siguiente:
assign r_next = r_reg + 1;
//Lógica de salida:
assign q = r_reg;
assign max_tick = (r_reg == (2**N)-1) ? (1'b1) : (1'b0);
endmodule
```

Contadores

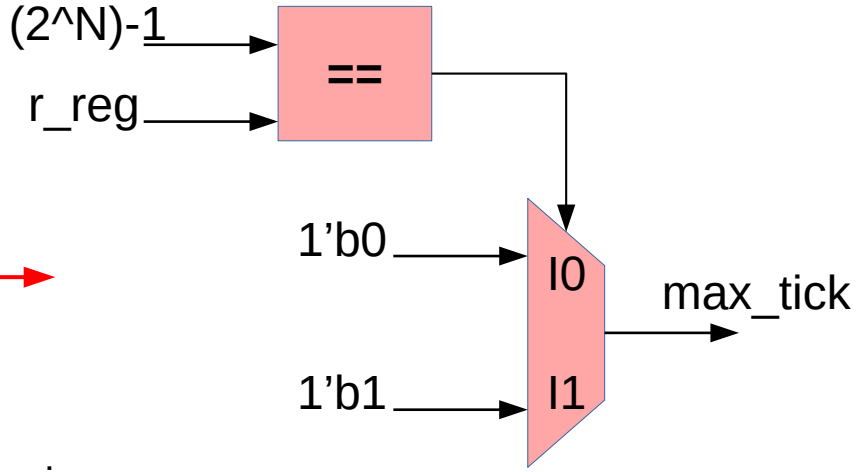
```
//Lógica del estado siguiente:
assign r_next = r_reg + 1;
//Lógica de salida:
assign q = r_reg;
assign max_tick = (r_reg == (2**N)-1) ? (1'b1) : (1'b0);
```




```
//Actualización del estado interno:
always @ (posedge clk, posedge rst)
begin
   if(rst)
       r_reg <= 0;
   else
       r_reg <= r_next;
end
```

Contadores

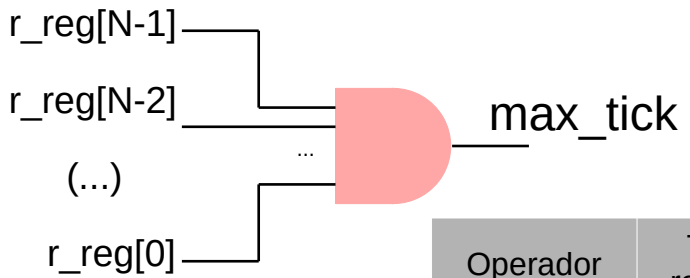
```
//Lógica del estado siguiente:
assign r_next = r_reg + 1;
//Lógica de salida:
assign q = r_reg;
assign max_tick = (r_reg == (2**N)-1) ? (1'b1) : (1'b0);
```



Operador ternario: **(condición) ? (val1) : (val2)**
Si la condición se cumple, asigna **val1**, de lo contrario, asigna **val2**.

Contadores

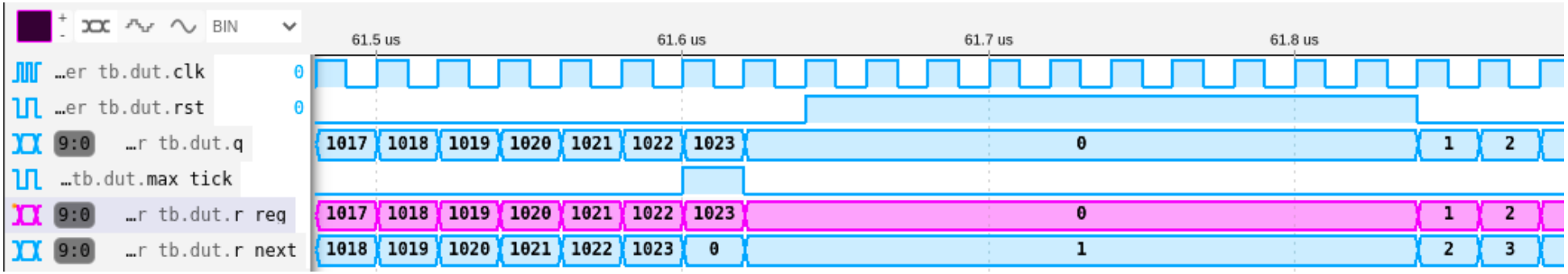
```
//Lógica de salida:
assign q = r_reg;
assign max_tick = &r_reg;
```



Operador de reducción: infiere la operación indicada por el operador entre todos los bits del operando. En este caso, se trata de una *reduction AND*, la cual aplica la operación **AND** a todos los bits de **r_reg**. El resultado es un único bit, de ahí el nombre de "reducción".

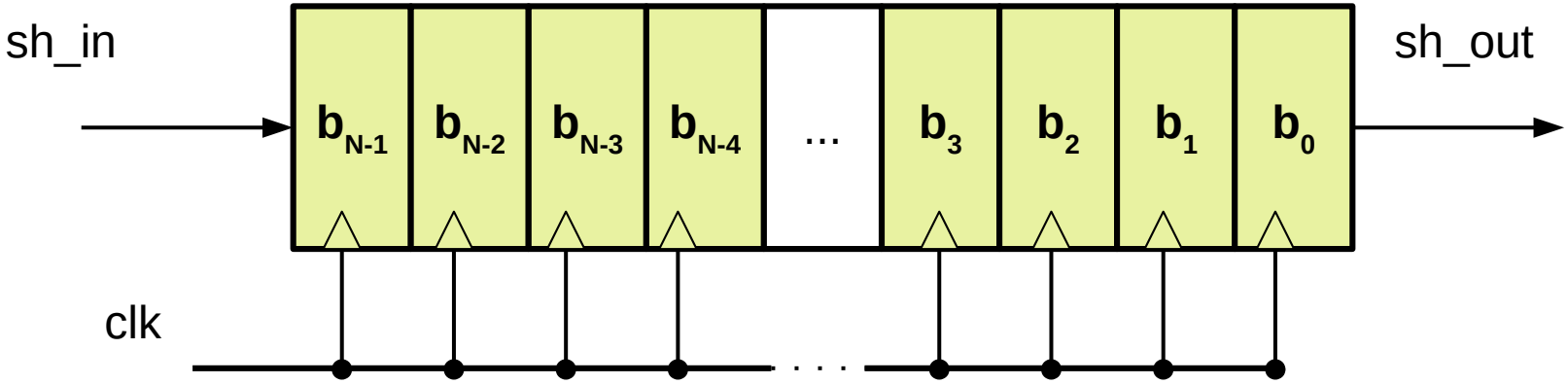
Operador	Tipo de reducción
&	AND
~&	NAND
|	OR
~|	NOR
^	XOR
~^	XNOR

Contadores


BIN
61.5 us
61.6 us
61.7 us
61.8 us
…er tb.dut.clk
0
…er tb.dut.rst
0
9:0
…r tb.dut.q
1017 1018 1019 1020 1021 1022 1023 0 1 2
…tb.dut.max tick
9:0
…r tb.dut.r_reg
1017 1018 1019 1020 1021 1022 1023 0 1 2
9:0
…r tb.dut.r_next
1018 1019 1020 1021 1022 1023 0 1 2 3

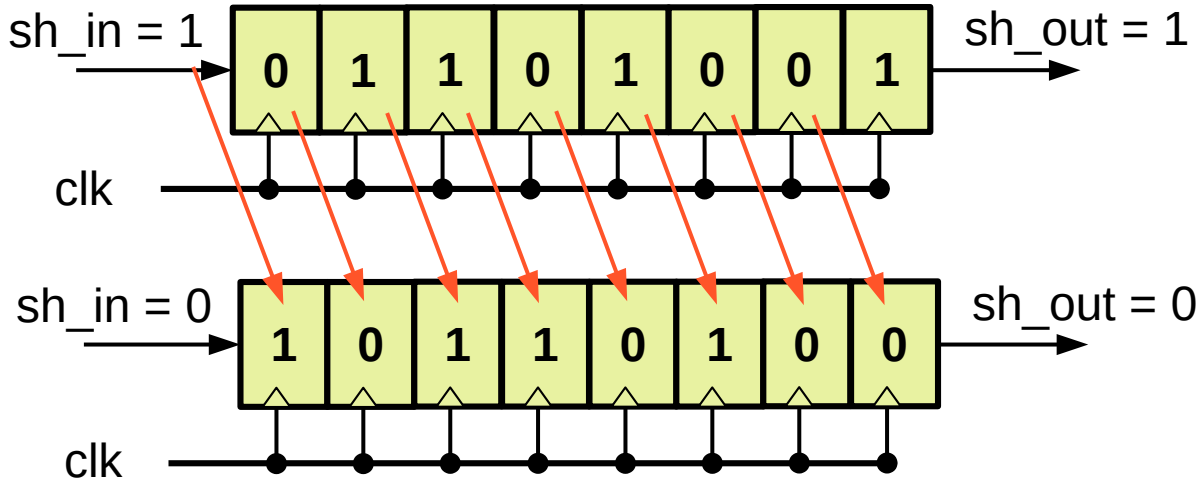
Registros de desplazamiento

Registro de desplazamiento



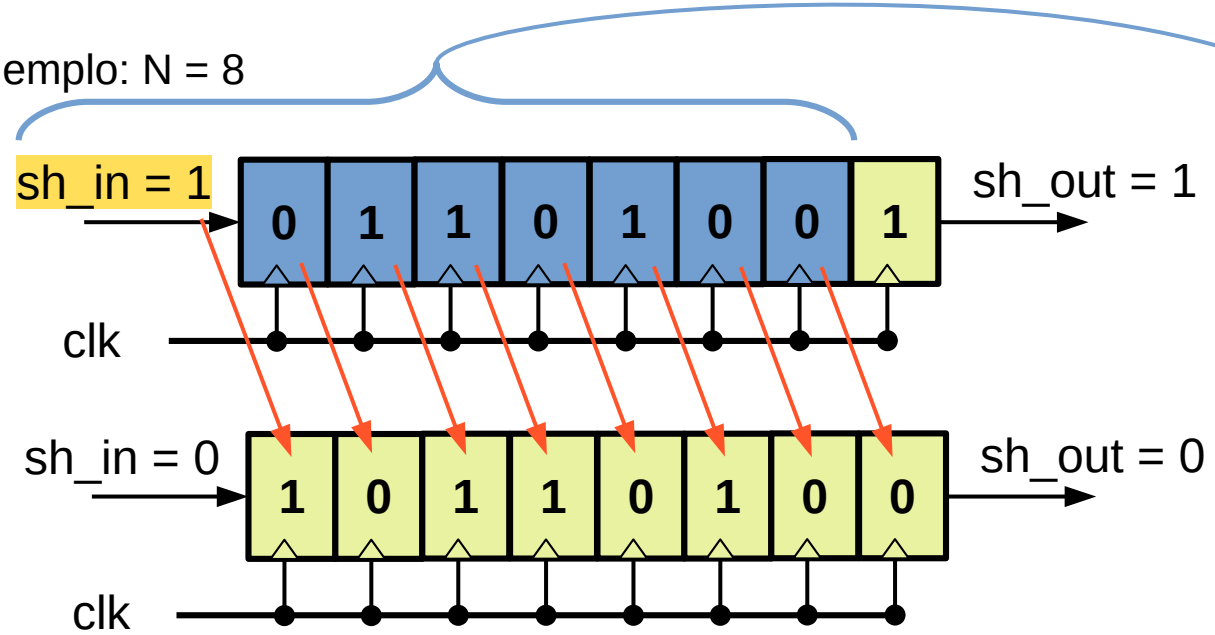
Registro de desplazamiento

Ejemplo: N = 8



Registro de desplazamiento

Ejemplo: N = 8



Definición del estado siguiente

Registro de desplazamiento

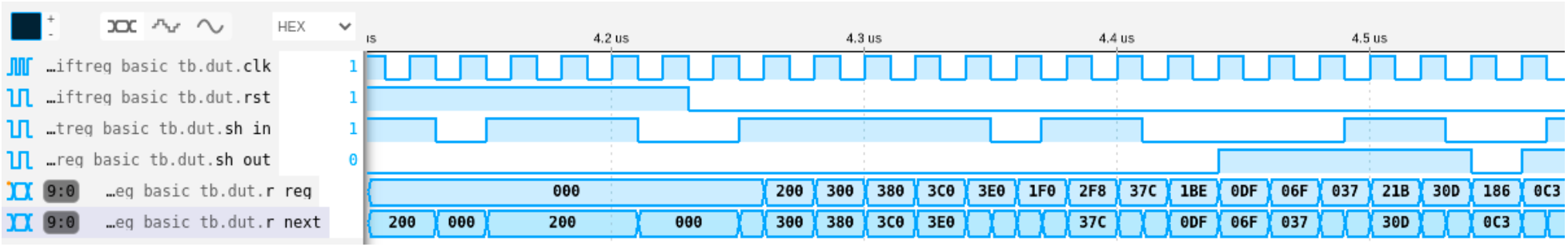
```
module shift_register
#(
    parameter N=8
)
(
    input wire clk,
    input wire rst,
    input wire sh_in,
    output wire sh_out
);
```

```
//Declaración de señales:
reg  [N-1:0] r_reg;
wire [N-1:0] r_next;
```

```
//Lógica del estado siguiente:
assign r_next = {sh_in,r_reg[N-1:1]};
//Lógica de salida:
assign sh_out = r_reg[0];
endmodule
```

```
//Actualización del estado interno:
always @ (posedge clk, posedge rst)
begin
   if(rst)
       r_reg <= {N{1'b0}};
   else
       r_reg <= r_next;
end
```

Registro de desplazamiento



Finite States Machines

Definición

Las FSM (del inglés Finite States Machines) son circuitos secuenciales con un **número finito de estados internos**, de forma tal que la **transición entre estos estados depende del estado actual y de las entradas** actuales. En general se trata de circuitos sincrónicos (con clock).

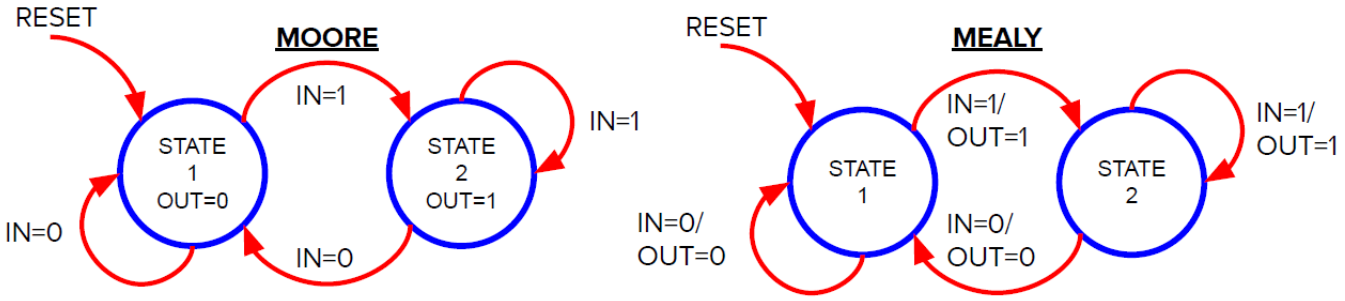
El enfoque de FSM es muy práctico y **utilizado para modelizar sistemas secuenciales complejos.** Todos los compiladores y sintetizadores de HDL pueden reconocer estas estructuras y optimizar su implementación.

FSM – Tipos de máquinas de estados

Podemos distinguir dos tipos de máquinas de estado:

- **Máquinas de Moore**: La salida solo depende del estado interno.
- **Máquinas de Mealy**: La salida depende del estado interno y de las entradas.

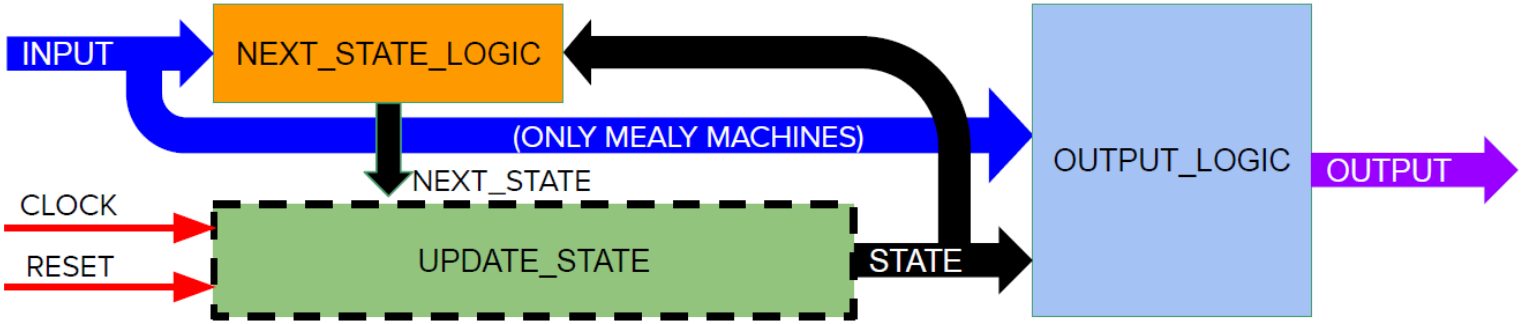
Al momento de diseñar e implementar una máquina de estados, conviene empezar desde un diagrama que facilite su interpretación. Un ejemplo muy común es el uso de “diagramas de estados y transiciones”:



FSM – Implementación RTL

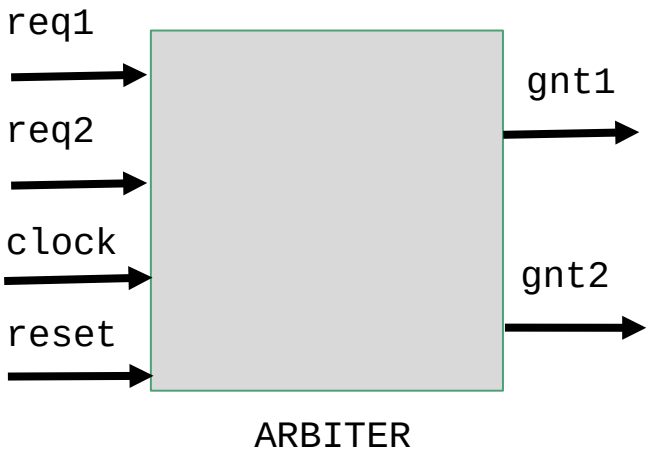
A nivel RTL, una forma sistemática y **prolija** de implementar FSM es dividiendo su comportamiento en **tres bloques always** (en Verilog):

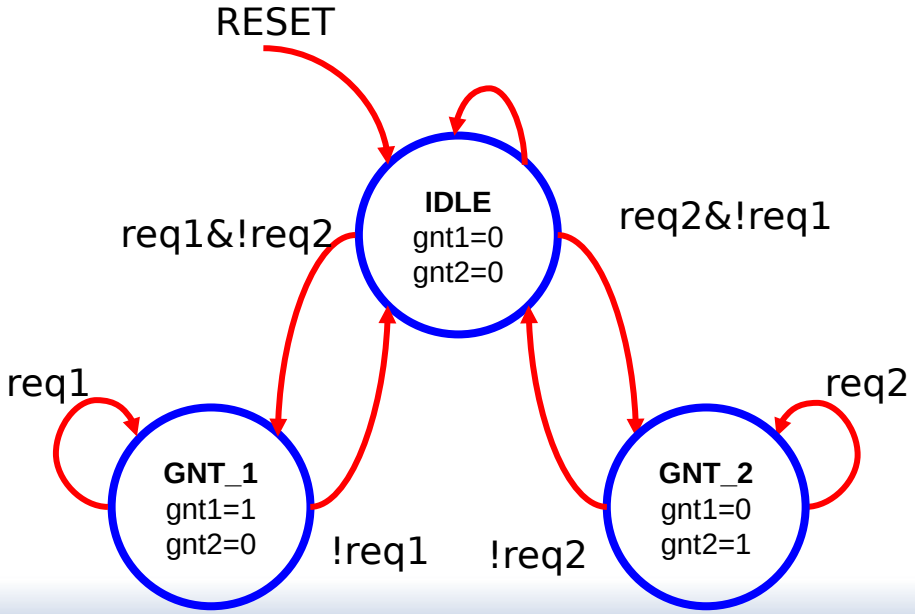
- **Un bloque combinacional (case-based)** que determina el próximo estado (**next_state_logic**)
- **Un bloque combinacional (case-based)** que determina las salidas (**output_logic**)
- **Un bloque secuencial (clock & reset sensitive)** que actualiza el estado interno (**update_state**)



Ejemplo FSM: Arbiter

Un *arbiter* es un bloque cuya función es **restringir el acceso de dos o más sistemas o procesos a un recurso compartido** (por ejemplo, un puerto E/S):





Ejemplo FSM: Arbiter – I/O and parameters

```
module ARBITER
(
 //Entradas
 input wire req1,
 input wire req2,
 //Reset
 input wire rst,
 //Clock
 input wire clk,
 //Salidas
 output reg gnt1,
 output reg gnt2
);
```

```
//Bits de representacion de estado
parameter N_BITS_STATE = 2;

//Estado actual
reg[N_BITS_STATE-1:0] state;

//Proximo estado
reg[N_BITS_STATE-1:0] next_state;

//Estados del sistema
localparam IDLE = 2'b00 ;
localparam GNT_1 = 2'b01 ;
localparam GNT_2 = 2'b10 ;
```

Ejemplo FSM: Arbiter – Next state logic

```
always @(*) begin
  //Asignación de próximo estado
  case (state)

     IDLE : begin
        if (req1==1'b1 && req2==1'b0) begin
           next_state = GNT_1;
        end else if (req2==1'b1 && req1==1'b0) begin
           next_state = GNT_2;
        end else begin
           next_state = IDLE;
        end
     end
     GNT_1 :begin
        if (req1==1'b1) begin
           next_state = GNT_1;
        end else begin
           next_state = IDLE;
        end
     end
     GNT_2 : begin
        if (req2==1'b1) begin
           next_state = GNT_2;
        end else begin
           next_state = IDLE;
        end
     end
     //Caso por default
     default : next_state = IDLE;
  endcase
end
```

Ejemplo FSM: Arbiter – Update logic

```
//Actualización sincróica del estado (bloque constante para todas las FSM)
always @(posedge rst or posedge clk) begin
  if (rst==1'b1) begin
     state <= IDLE;
  end else begin
     state <= next_state;
  end
end
```

Ejemplo FSM: Arbiter – Output logic

```
//Definición combinacional de las
salidas segun el estado actual
always @(*) begin
   case (state)
     IDLE : begin
        gnt1 = 0;
        gnt2 = 0;
     end
     GNT_1 : begin
        gnt1 = 1;
        gnt2 = 0;
     end
     GNT_2 : begin
        gnt1 = 0;
        gnt2 = 1;
     end
     default: begin
        gnt1 = 0;
        gnt2 = 0;
     end
  endcase
end
```

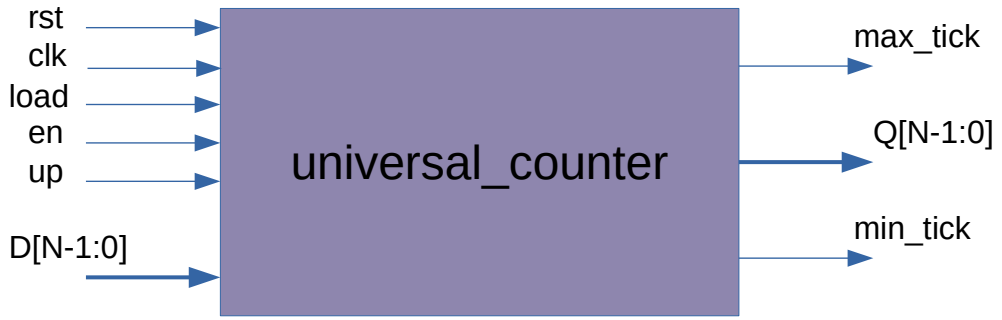
*Observar que las salidas dependen solo del estado interno. La FSM codificada es entonces del tipo Moore.

Actividades propuestas

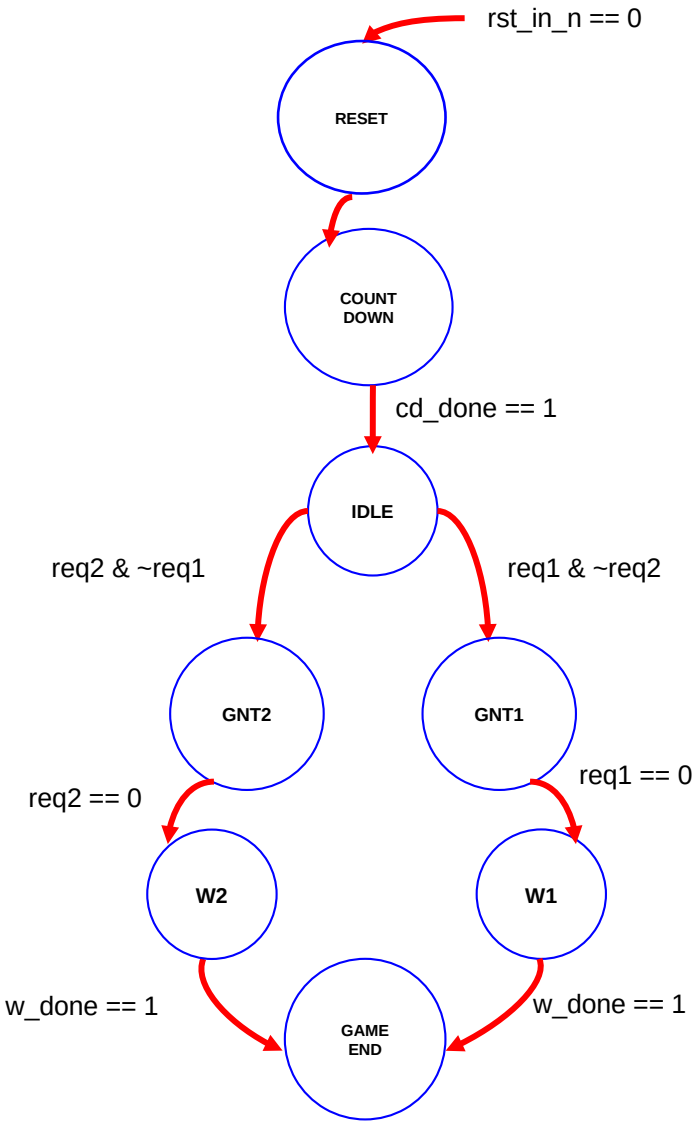
Ejercicio 1: Contador Universal

Ejercicio 1: Contador Universal

rst	load	en	up	Q*	Operación
1	--	--	--	0	Reset asincrónico
0	1	--	--	D	Carga paralela
0	0	1	1	Q+1	Cuenta ascendente
0	0	1	0	Q-1	Cuenta descendente
0	0	0	--	Q	Pausa



Ejercicio 2: Arbiter Game


rst_in_n == 0
RESET
COUNT DOWN
cd_done == 1
IDLE
req2 & ~req1
req1 & ~req2
GNT2
GNT1
req2 == 0
req1 == 0
W2
W1
w_done == 1
w_done == 1
GAME END

Arbiter Game - salidas

	RESET	COUNTDOWN	IDLE	GNT_1	GNT_2	W1	W2	GAME END
gnt1	0	0	0	1	0	1	0	0
gnt2	0	0	0	0	1	0	1	0
cd_rst	1	0	1	1	1	1	1	1
w_rst	1	1	1	0	0	0	0	1
leds_rst	1	0	1	0	0	0	0	1
leds_sel	0	0	0	1	1	1	1	0

Anexo 1: asignaciones bloqueantes y no bloqueantes

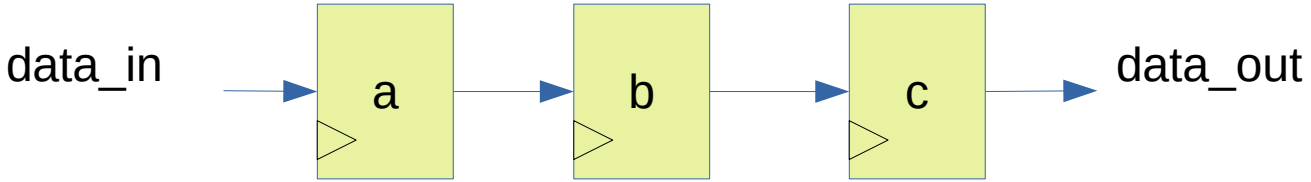
Asignaciones bloqueantes y no bloqueantes

- **Asignaciones bloqueantes (=)**: se ejecutan secuencialmente, una luego de la otra en los bloques always o initial. Se bloquea la ejecución de la sentencia siguiente hasta que la sentencia actual sea ejecutada, por tal motivo se las denomina **bloqueantes. Se utiliza para inferir lógica combinacional o asignar valores a variables en bucles (for, while).**

- **Asignaciones no bloqueantes (<=)**: se ejecutan sin bloquear la ejecución de las sentencias siguientes. El valor en el operador derecho (RHS) se lee inmediatamente, pero se asigna al operador izquierdo (LHS) al final del *timestamp*. Cuando se utilizan asignaciones no bloqueantes, todos las variables del LHS se actualizan una vez que se leyeron los valores de todos los operandos en RHS. De esta forma, se evitan problemas cuando una misma variable aparece tanto en LHS como en RHS en diferentes asignaciones (por ejemplo, dos Flip-Flops en cascada). **Se utiliza para inferir lógica secuencial.**

Asignaciones bloqueantes y no bloqueantes

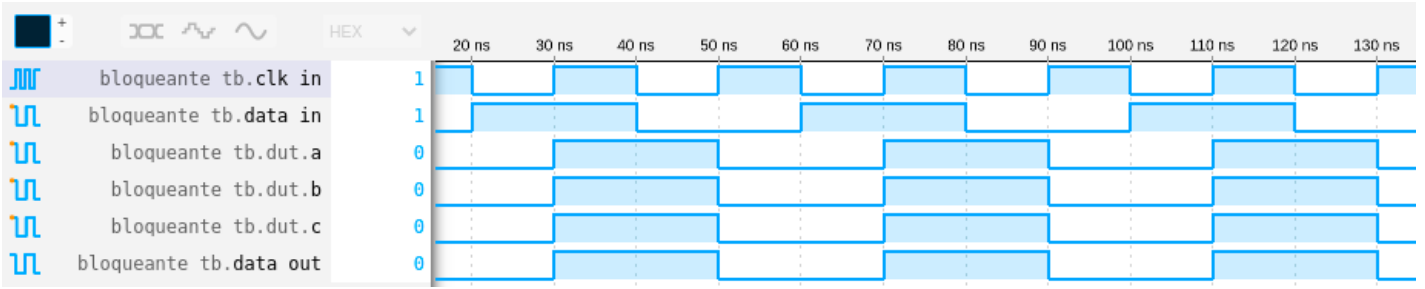
Ejemplo: queremos sintetizar 3 FFD en cascada:



Asignaciones bloqueantes y no bloqueantes

```
1   module bloqueante
2   (
3       input data_in,
4       input clk_in,
5       output data_out
6   );
7
8   reg a;
9   reg b;
10  reg c;
11
12  always @ (posedge clk_in)
13  begin
14    a = data_in;
15    b = a;
16    c = b;
17  end
18
19  assign data_out = c;
20
21  endmodule
```

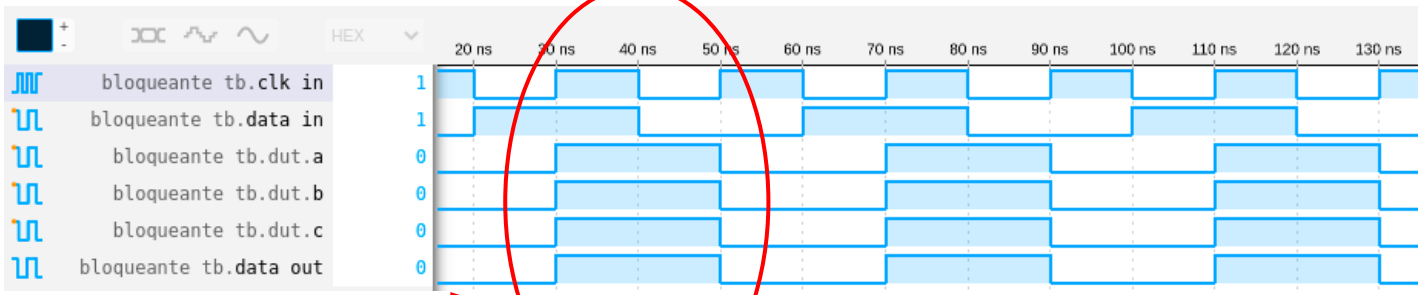
Ejemplo: mal uso de asignaciones bloqueantes para inferir lógica secuencial.



Asignaciones bloqueantes y no bloqueantes

```
1   module bloqueante
2   (
3       input data_in,
4       input clk_in,
5       output data_out
6   );
7
8   reg a;
9   reg b;
10  reg c;
11
12  always @ (posedge clk_in)
13  begin
14    a = data_in;
15    b = a;
16    c = b;
17  end
18
19  assign data_out = c;
20
21  endmodule
```

Ejemplo: mal uso de asignaciones bloqueantes para inferir lógica secuencial.

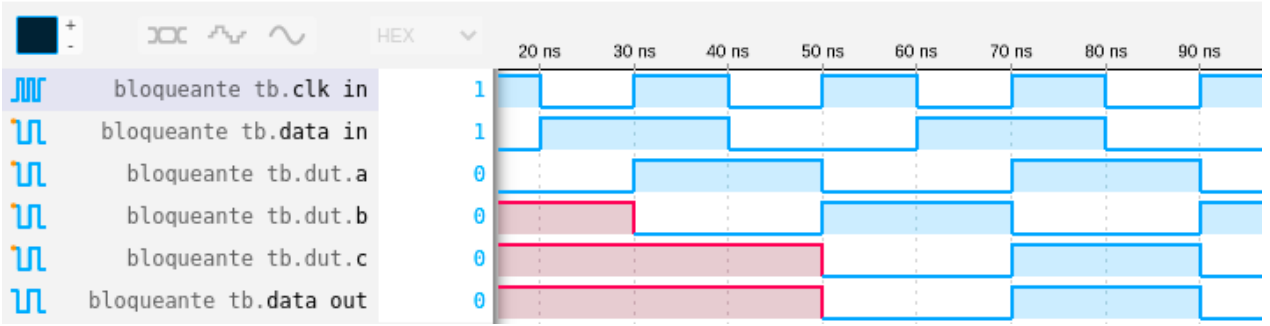


El valor de **data_in** se transfiere directamente a la salida **data_out**

Asignaciones bloqueantes y no bloqueantes

```
1   module no_bloqueante
2   (
3       input data_in,
4       input clk_in,
5       output data_out
6   );
7
8   reg a;
9   reg b;
10  reg c;
11
12  always @ (posedge clk_in)
13  begin
14    a <= data_in;
15    b <= a;
16    c <= b;
17  end
18
19  assign data_out = c;
20
21  endmodule
```

Ejemplo: uso de asignaciones no bloqueantes para inferir lógica secuencial.
Aquí puede verse que el comportamiento corresponde a 3 FFD en cascada:



¡Gracias por su atención!