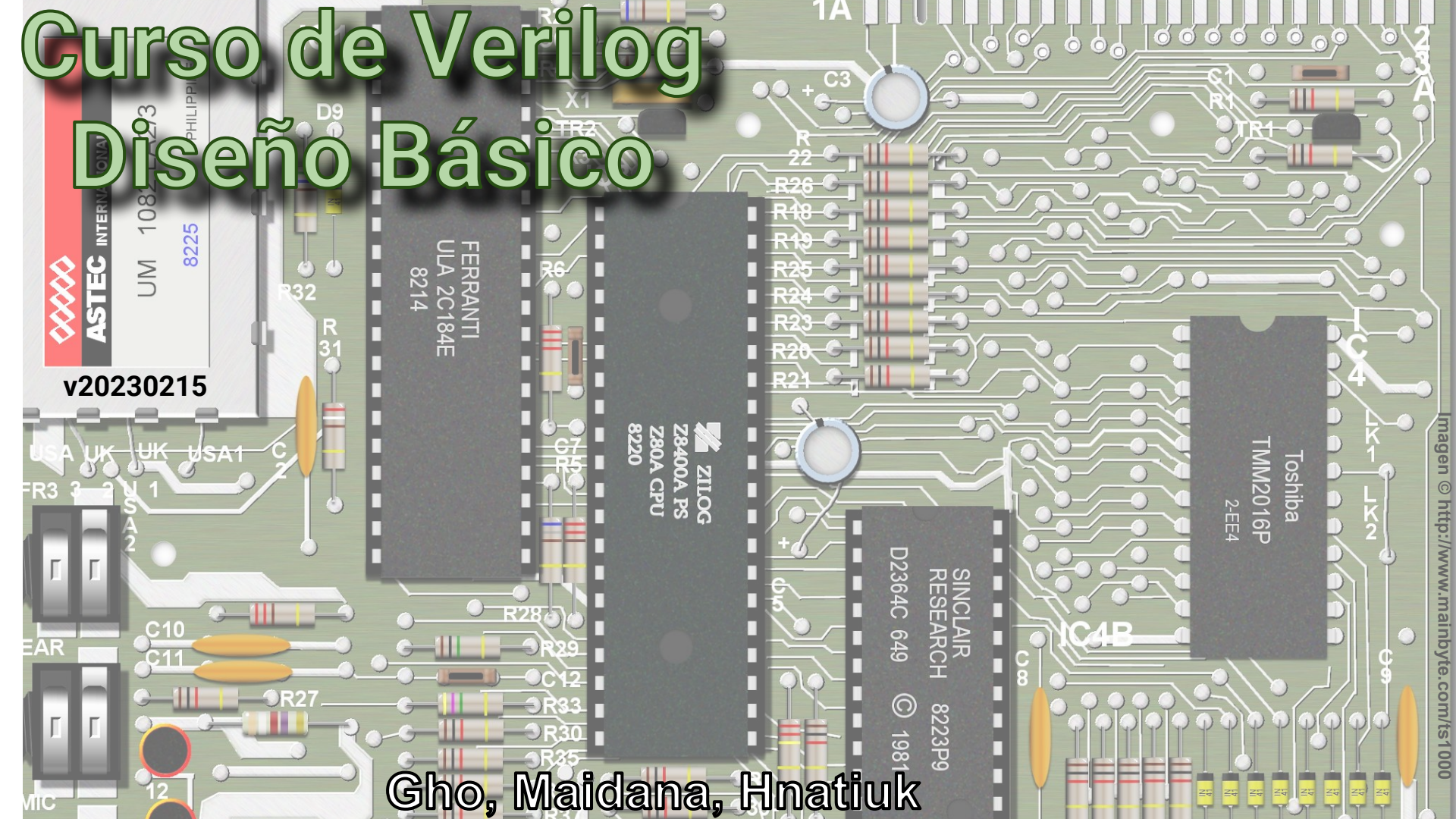

Curso de Verilog
Diseño Básico
v20230215
Gho, Maidana, Hnatiuk
Imagen © http://www.mainbyte.com/ts1000

Acerca de los autores

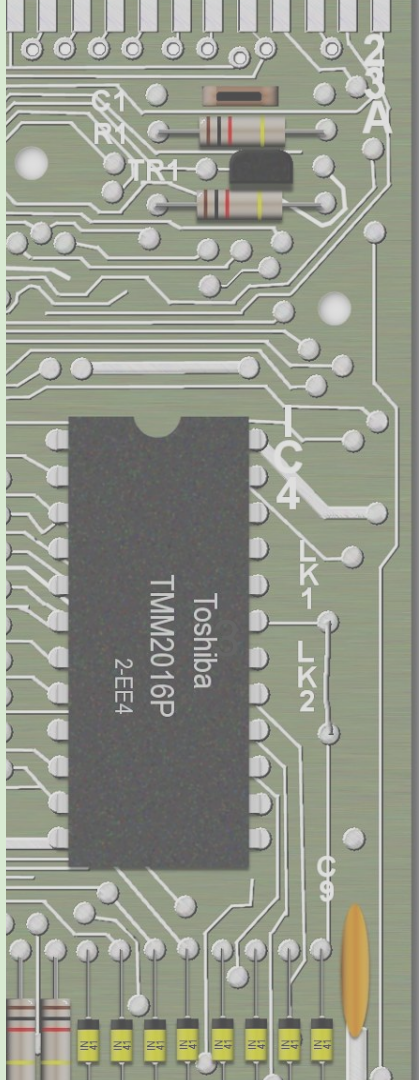
Este curso fue creado por integrantes del Grupo de Investigación en Lógica Programable (GILP-UNLaM)

Edgardo Gho

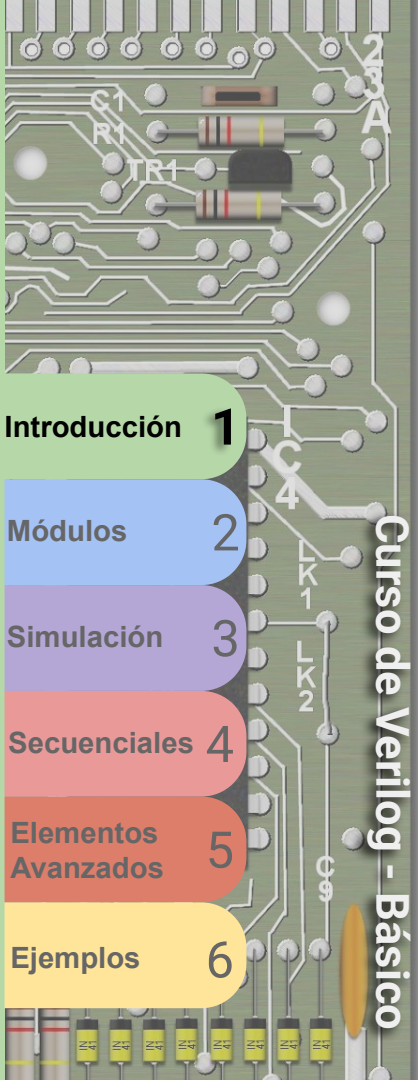
Carlos Maidana

Jair Hnatiuk

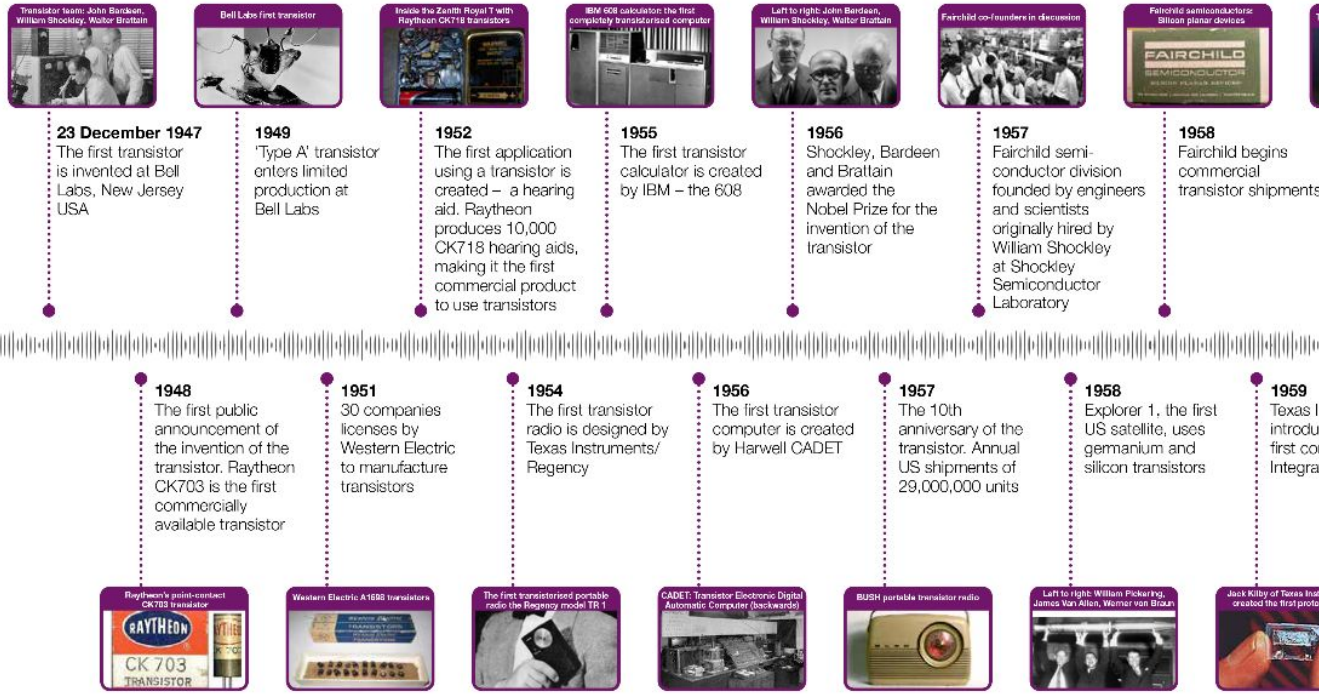
Está pensado como complemento al curso de VHDL desarrollado y dictado desde 2007. El material es un apoyo al curso presencial que se dicta en laboratorio con acceso a computadoras con el software Vivado de Xilinx y kits de desarrollo basados en la séptima familia de chips de Xilinx (Artix 7, Spartan 7, etc).



Diseño de circuitos



Celebrating 70 years of the transistor



https://www.electronicsweekly.com/news/first-transistor-created-70-years-ago-the-device-that-changed-the-world-2017-12/

La lógica programable

tor

Imagination

1958
Fairchild begins commercial transistor shipments

1960
Sony announces the TV8-301, the first portable TV containing just 23 silicon transistors

1965
Intel's Gordon Moore states that the number of transistors on a chip will double every year, a prediction that becomes known as Moore's Law. (Later revised to every 18 months)

1980s
Processors increase significantly in performance. One example is the Intel 80286 16-bit microprocessor, containing over 134,000 transistors

2000s
Graphics processing units (GPUs), employing billions of transistors bring hardware 3D acceleration to the PC desktop

2010s
Transistor counts in typical smartphones typically exceed 2 billion with mobile GPUs accounting for most of this

orer 1, the first satellite, uses anium and n transistors

1959
Texas Instruments introduces the first commercial Integrated Circuit (IC)

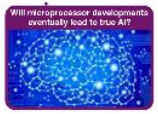
1964
The first widely-used analogue Integrated Circuit (IC) is introduced by Talbert and Widlar at Fairchild

1970s
The ability to place an increasing number of transistors on an IC leads to the widespread use of the microprocessor – a central processing unit (CPU) in a single chip

1990s
The launch of the World Wide Web in 1993 boosts the need for fast processors, making the need to increase transistor counts on CPUs even more crucial

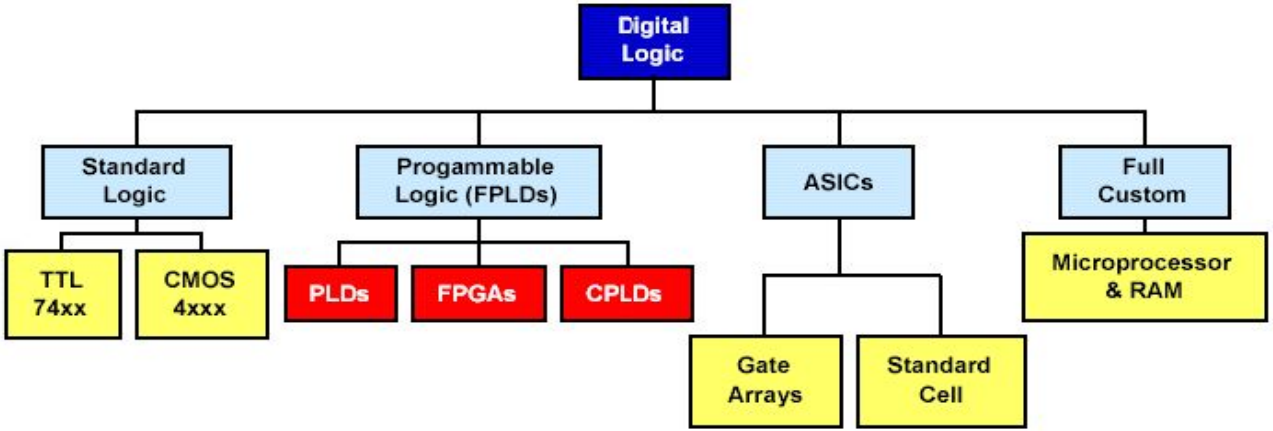
Mid-2000s
PowerVR's TBDR-based technology creates the mobile GPU market, paving the way for mass-market smartphone adoption

2020s
It is predicted that microprocessors and GPUs will have as many transistors as there are neurons in a human brain

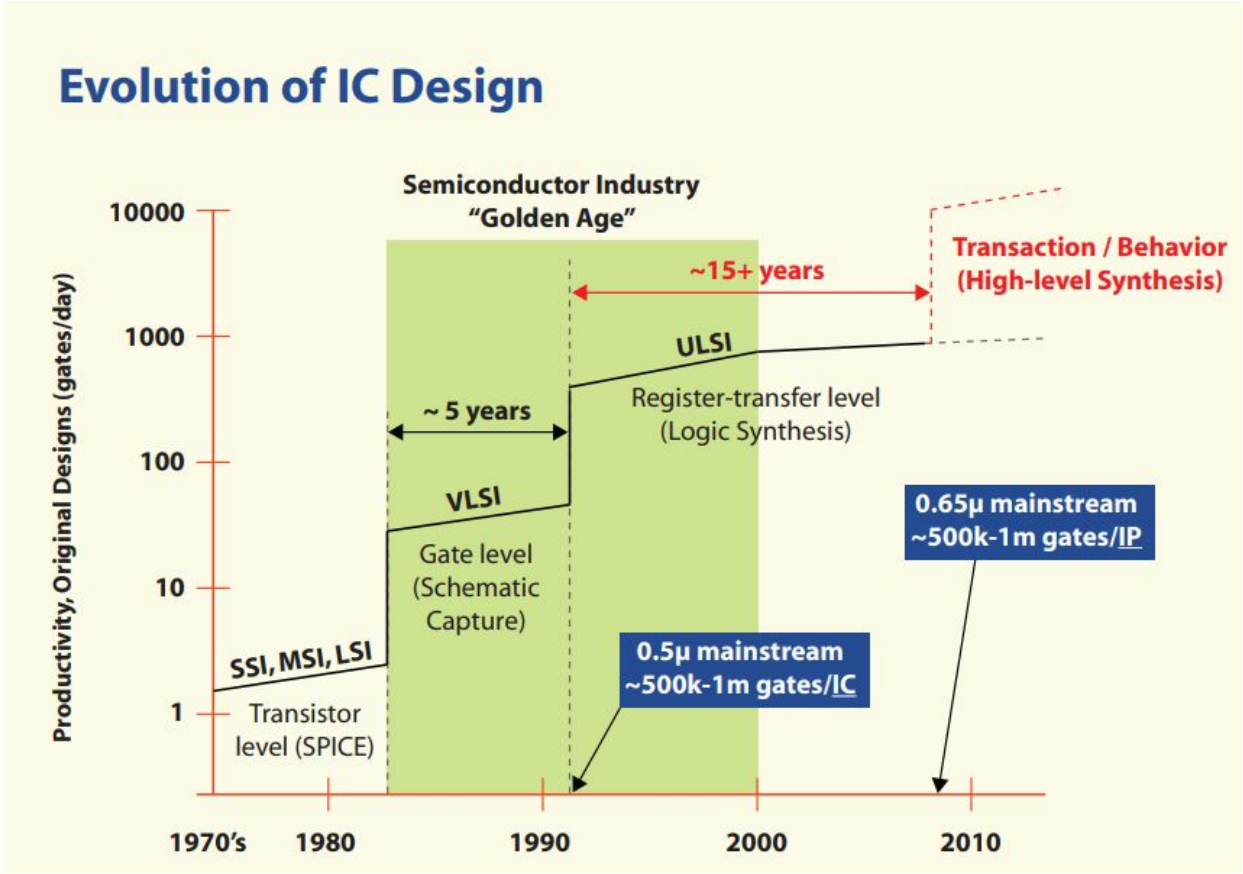


https://www.electronicsweekly.com/news/first-transistor-created-70-years-ago-the-device-that-changed-the-world-2017-12/

Diseño digital

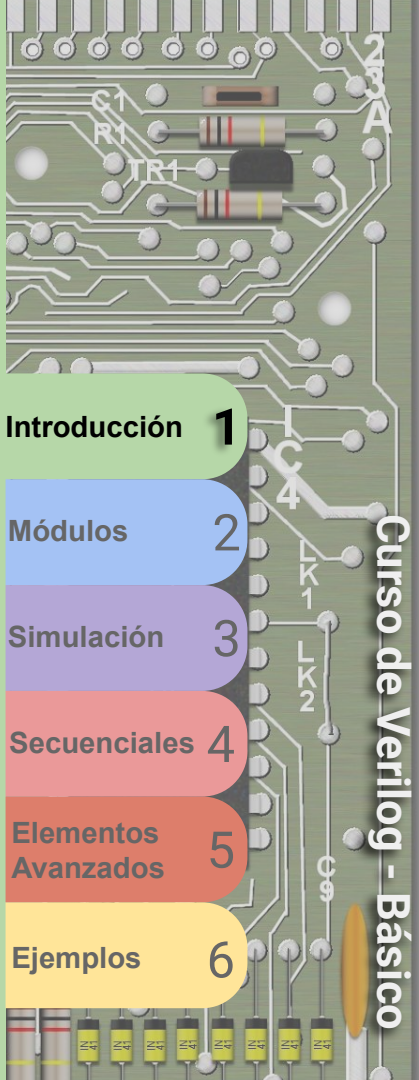


La lógica programable



https://www.accellera.org/images/resources/articles/icdesigntrans/ic_design_transition_feb2010.pdf

HDL (Lenguajes descriptivos de hardware)



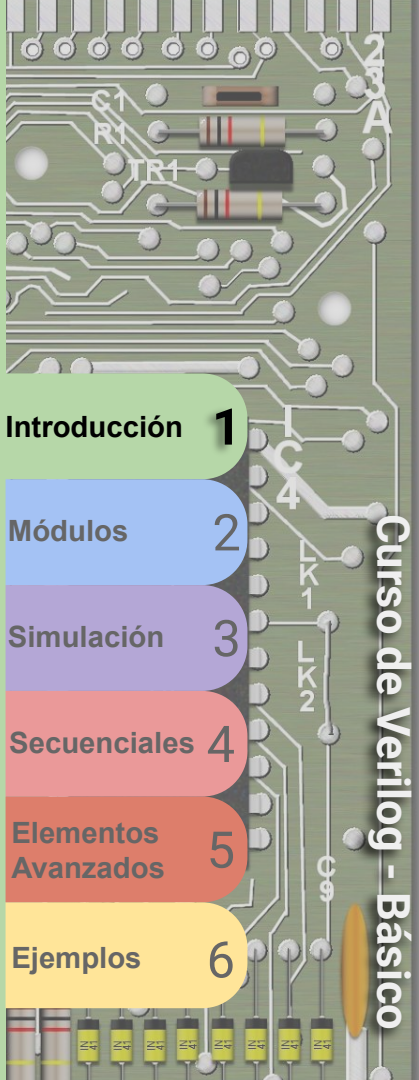
VHDL

Se desarrolla en 1983 como un lenguaje para describir el funcionamiento de circuitos. El problema era que en los 80s la documentación de cómo se “comportaban” los C.I. variaba mucho según el fabricante. El depto de defensa de USA necesitaba un estándar y entonces surge VHDL. En 1987 se estandariza como IEEE 1076-1987. A partir de este punto los fabricantes de circuitos y herramientas CAD empiezan a sugerir cambios y mejoras que terminan en IEEE 1076-1993.
A lo largo de los años se ha ido actualizando el estándar soportando cambios en el lenguaje.
Uno de los cambios principales que incorpora la versión de 1993 es la síntesis de circuitos utilizando el mismo lenguaje. Si bien existe sintaxis específica para simular y sintaxis específica para sintetizar, el lenguaje es prácticamente el mismo para ambas tareas.

Verilog (Verification-logic) IEEE 1364

Creado en 1984 como un producto comercial, fue estandarizado por IEEE en 1995. Comenzó como una herramienta de descripción de circuitos lógicos pero rápidamente obtuvo capacidad de simular los mismos. Cuando fue lo suficientemente popular obtuvo la capacidad de sintetizar circuitos.
Luego de ser estandarizado en 1995 se hicieron mejoras y cambios que resultaron en un nuevo estándar IEEE1364-2001. Esta es la versión más soportada por las diversas herramientas. Luego en el año 2009 se une con SystemVerilog. Desde ese entonces todo cambio se publica bajo el lenguaje SystemVerilog.

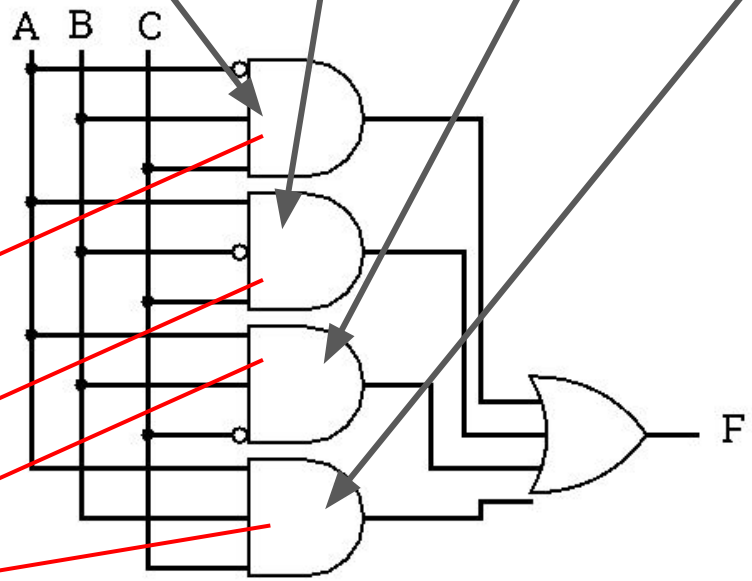
Funciones booleanas



Función booleana F (3 variables)

$$F(A,B,C)=\overline{A}.B.C+A.\overline{B}.C+A.B.\overline{C}+A.B.C$$

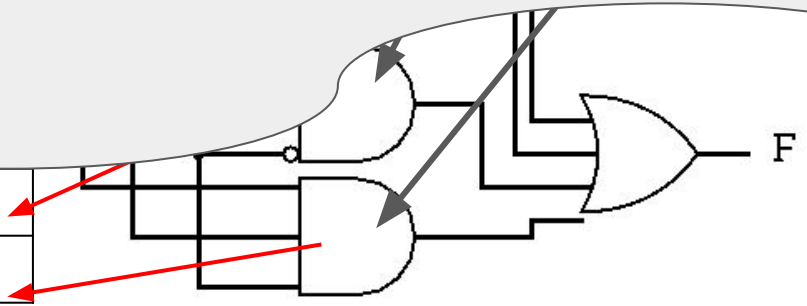
A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	**1**
1	0	0	0
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**



Función booleana F (3 variables)

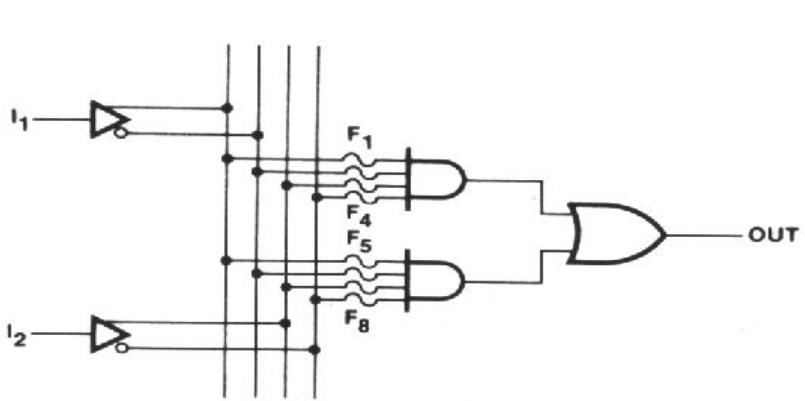
Las tres formas de representación de la función (Tabla de verdad, circuito con compuertas o por definición de operaciones booleanas) SON EQUIVALENTES!. Es decir pasar de una forma a otra es completamente trivial. Las tres representan la misma función lógica.

1			
1	1	0	**1**
1	1	1	**1**

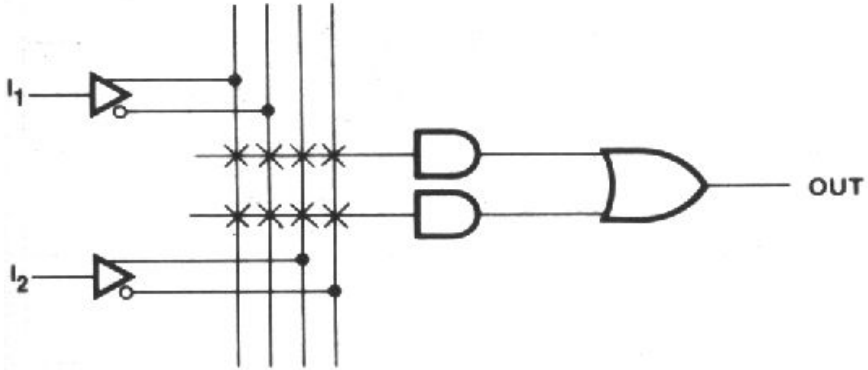


PAL

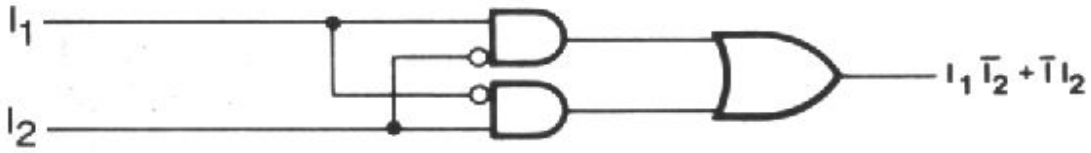
Circuito lógico real de una PAL

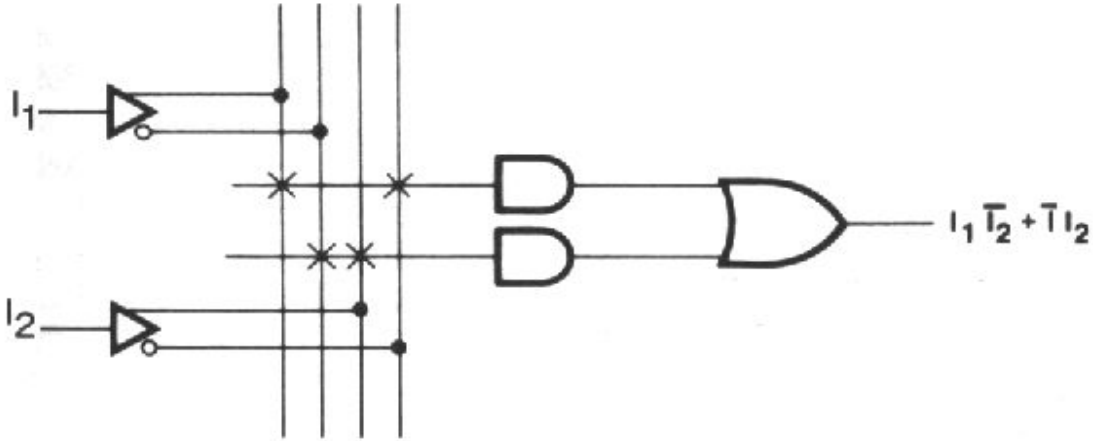


Circuito lógico simbólico de una PAL

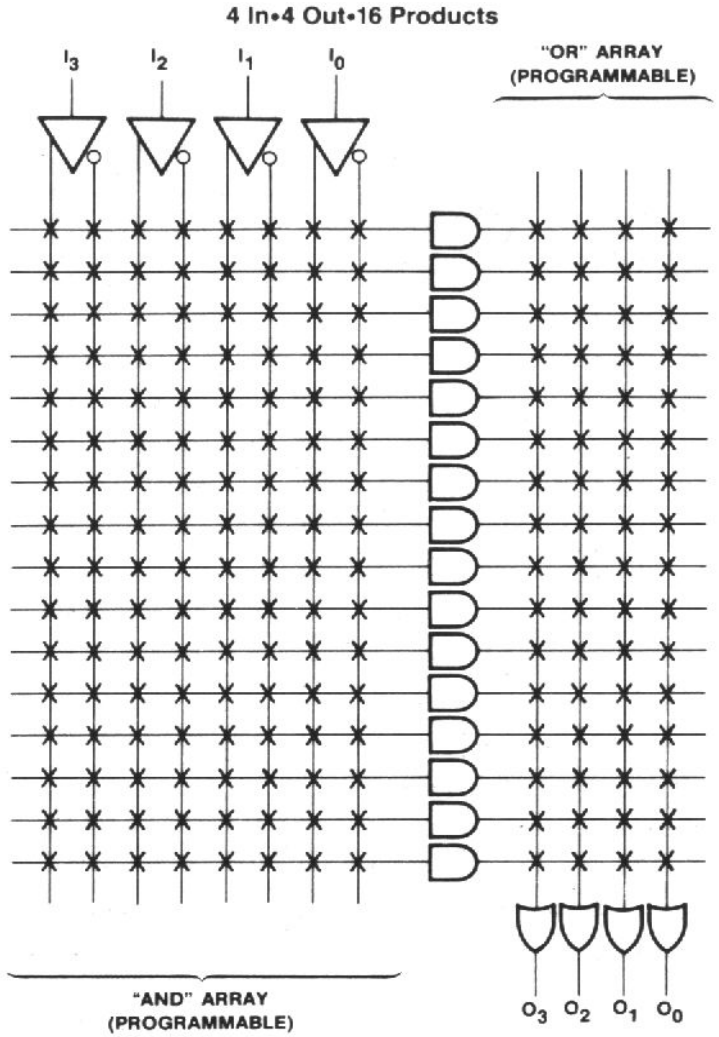


Implementación y esquema de quemado de fusibles





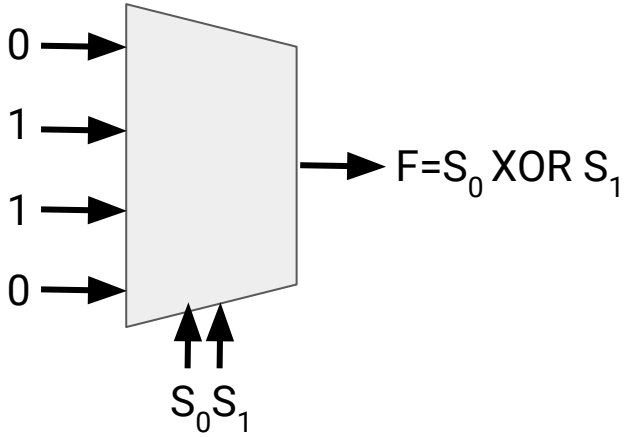
Pal de 4 entradas/salidas



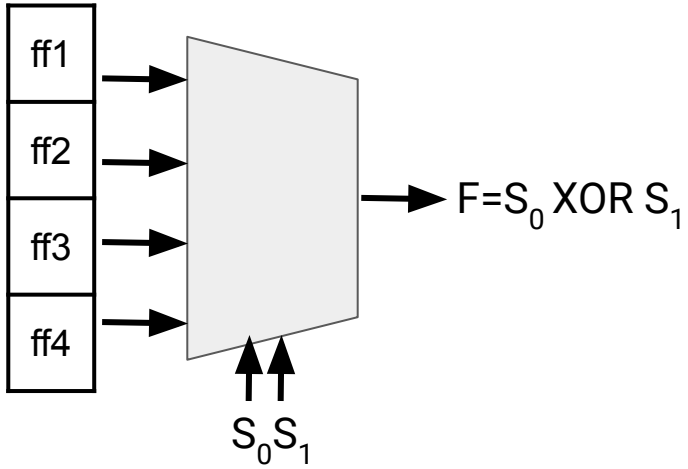
Funciones con multiplexores

Un MUX con entradas fijas

S_1	S_0	F
0	0	0
0	1	1
1	0	1
1	1	0

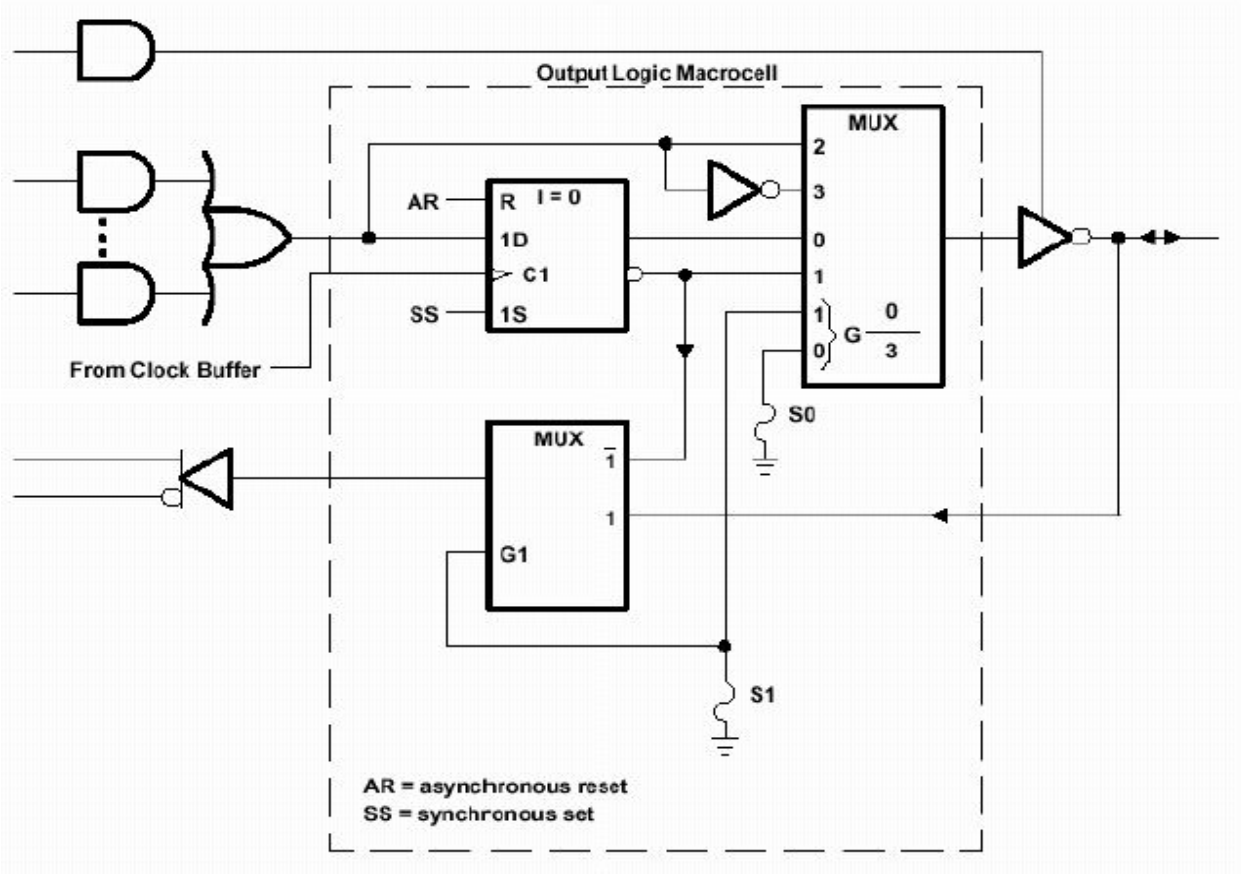


Podemos conectar un shift register compuesto de 4 FF a las entradas del MUX. Luego cambiando los valores de FFx podemos generar cualquier "compuerta" entre S0 y S1.

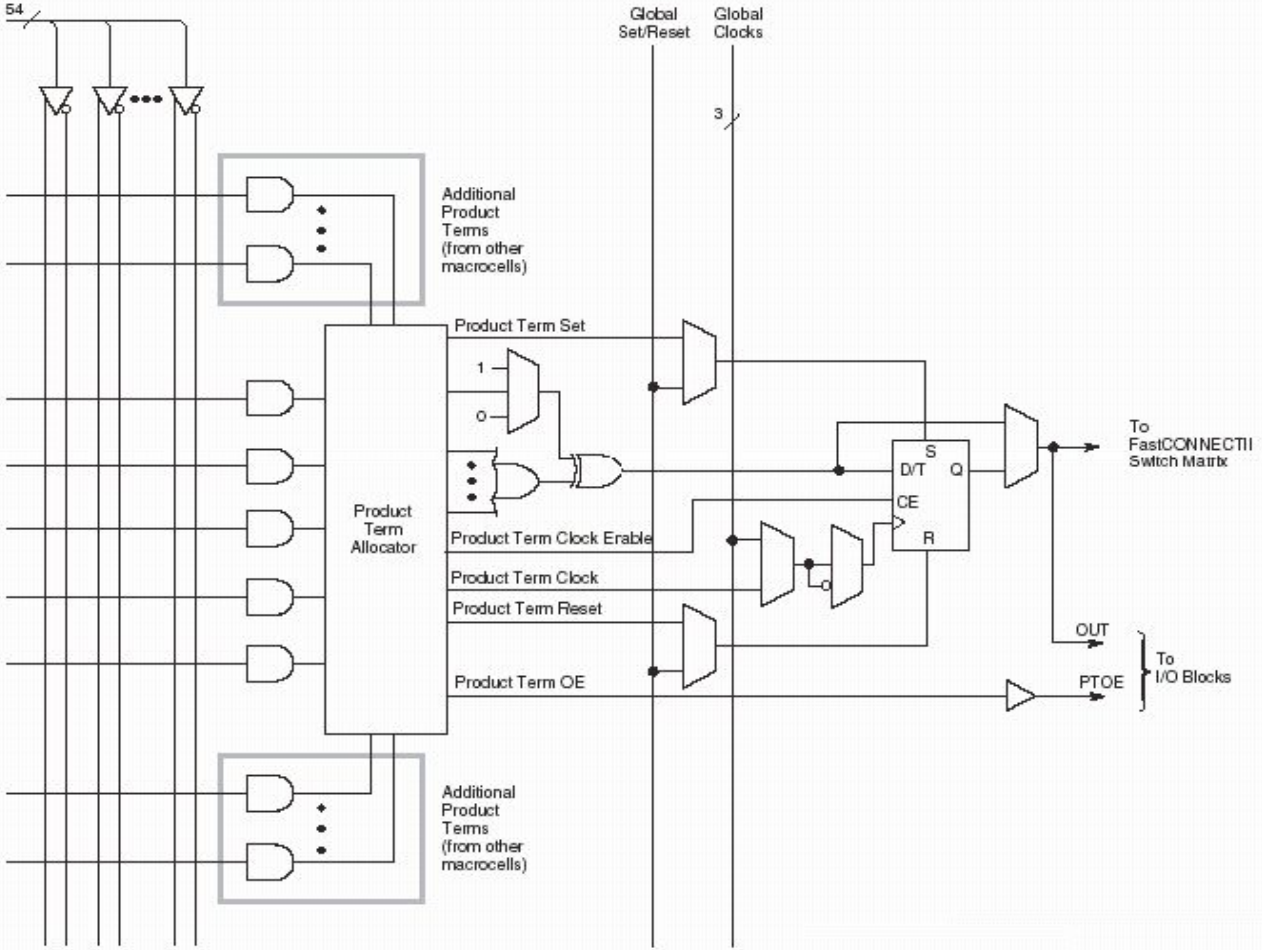


GAL y CPLD

Macrocelda de una GAL



CPLD



FPGA CLB

Configurable Logic Block (CLB)

https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

The 7 series configurable logic block (CLB) provides advanced, high-performance FPGA logic:

- Real 6-input look-up table (LUT) technology
- Dual LUT5 (5-input LUT) option
- Distributed Memory and Shift Register Logic capability
- Dedicated high-speed carry logic for arithmetic functions
- Wide multiplexers for efficient utilization

CLBs are the main logic resources for implementing sequential as well as combinatorial circuits. Each CLB element is connected to a switch matrix for access to the general routing matrix (shown in Figure 1-1). A CLB element contains a pair of slices.

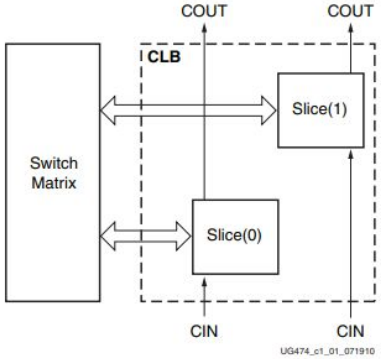
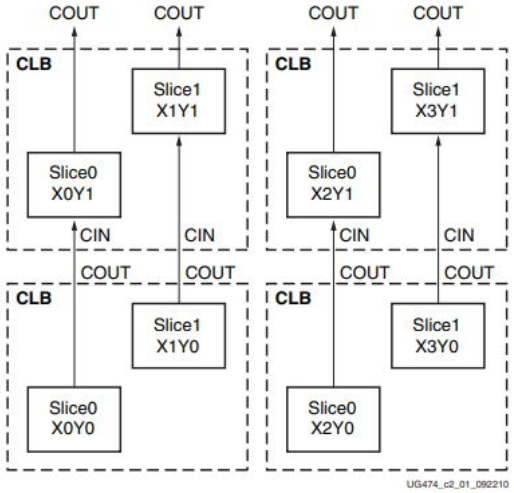


Figure 1-1: **Arrangement of Slices within the CLB**



Approximately two-thirds of the slices are SLICEL logic slices and the rest are SLICEM, which can also use their LUTs as distributed 64-bit RAM or as 32-bit shift registers (SRL32) or as two SRL16s. Modern synthesis tools take advantage of these highly efficient logic, arithmetic, and memory features. Expert designers can also instantiate them.

Configurable Logic Block (CLB)

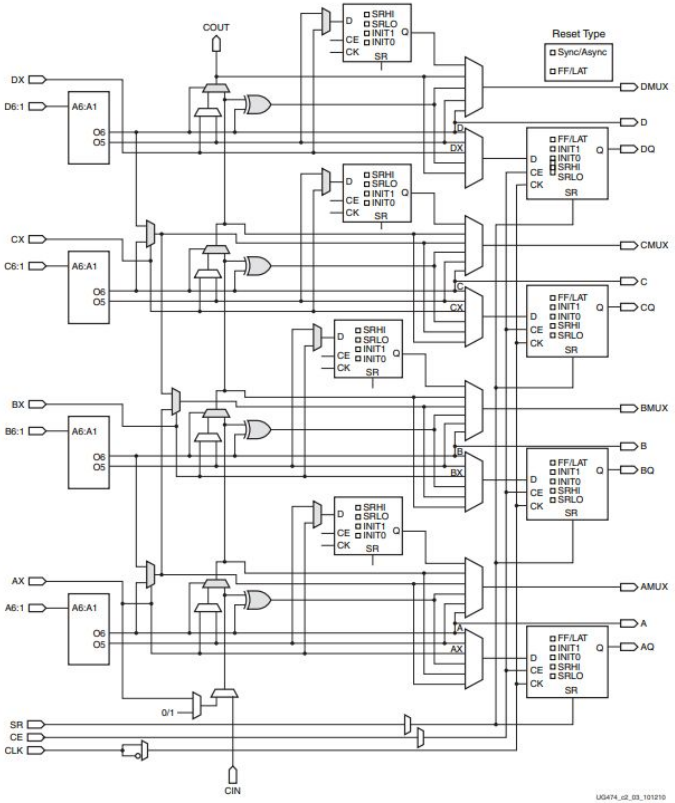


Figure 2-4: **Diagram of SLICEL**

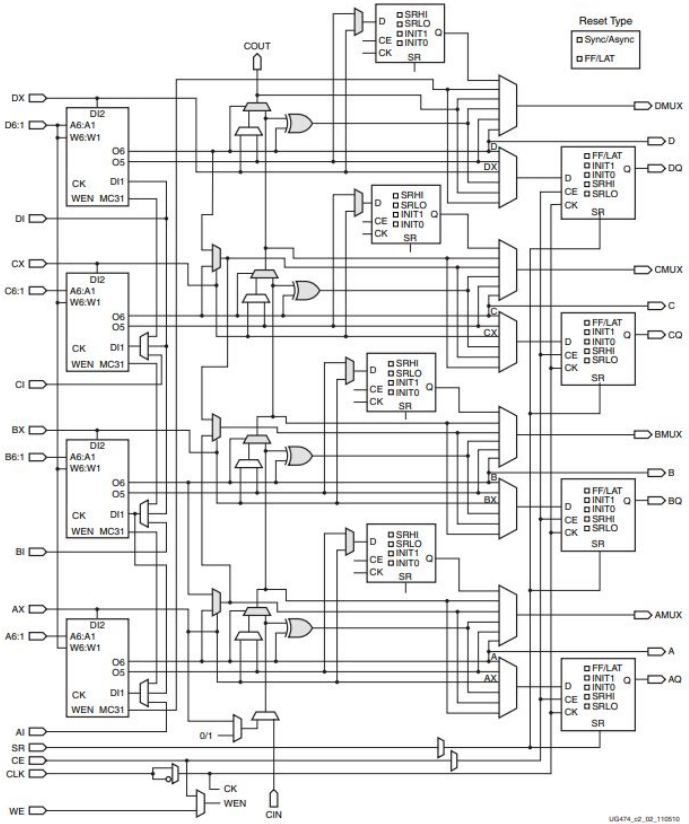
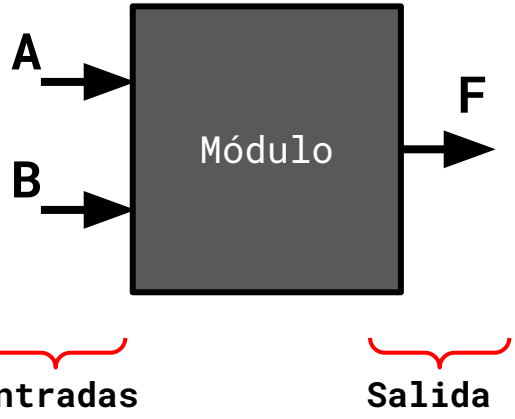


Figure 2-3: **Diagram of SLICEM**

Curso de Verilog - Básico

Módulos en Verilog

Diseño modular (Caja negra)



module *nombre* (*lista de puertos*);

- Interface
  - *Tipos de los puertos*
  - *Parámetros*
- Body
  - *Variables*
  - *Asignaciones*
  - *Instanciación de módulos*
  - *Bloques de simulación*
  - *Bloques de comportamiento*
  - *Funciones y tareas*

endmodule

Verilog

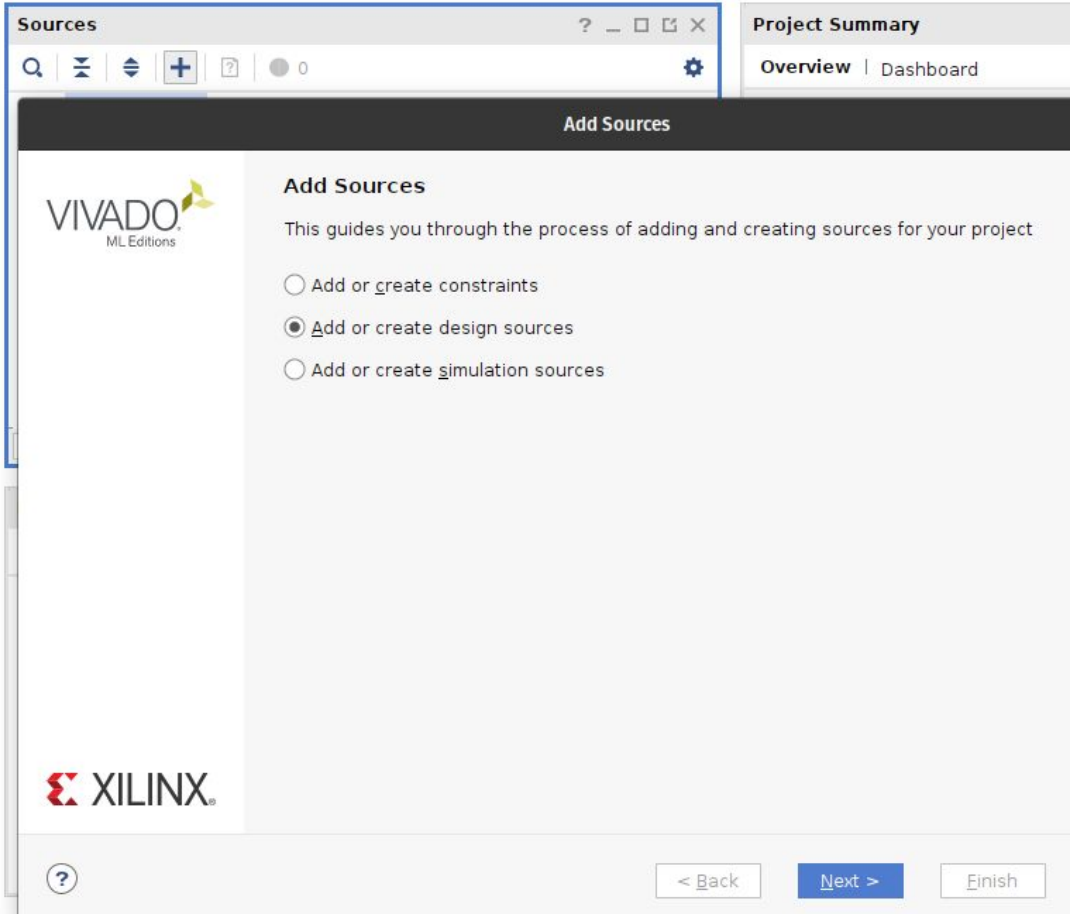
```
module Modulo( A,B,F);
   input A;
   input B;
   output F;
endmodule
```

Verilog 2001

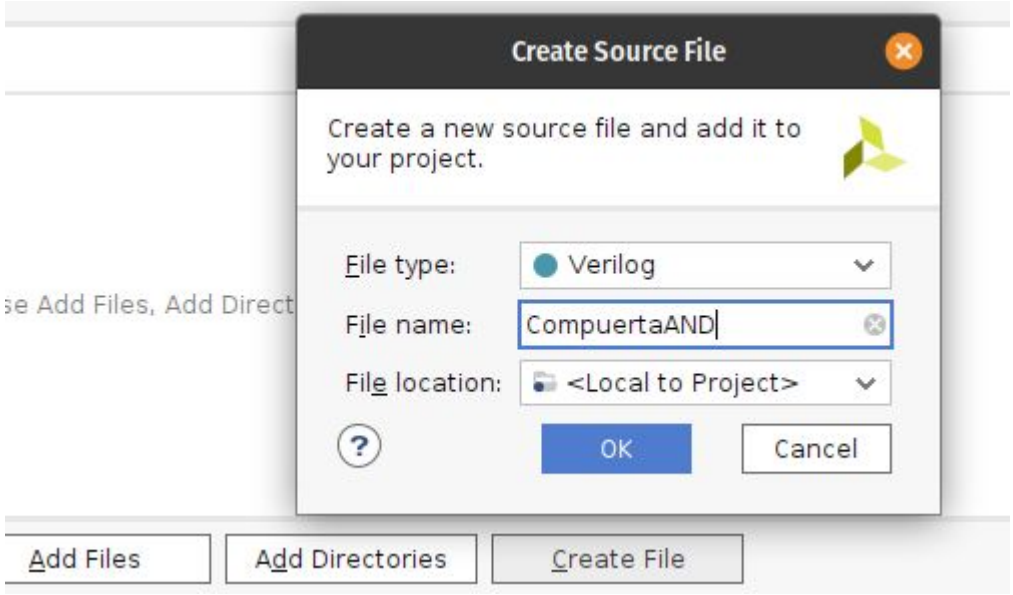
```
module Modulo(
    input A,
    input B,
    output F
    );
endmodule
```

```
module Modulo( input A,B,
               output F );
endmodule
```

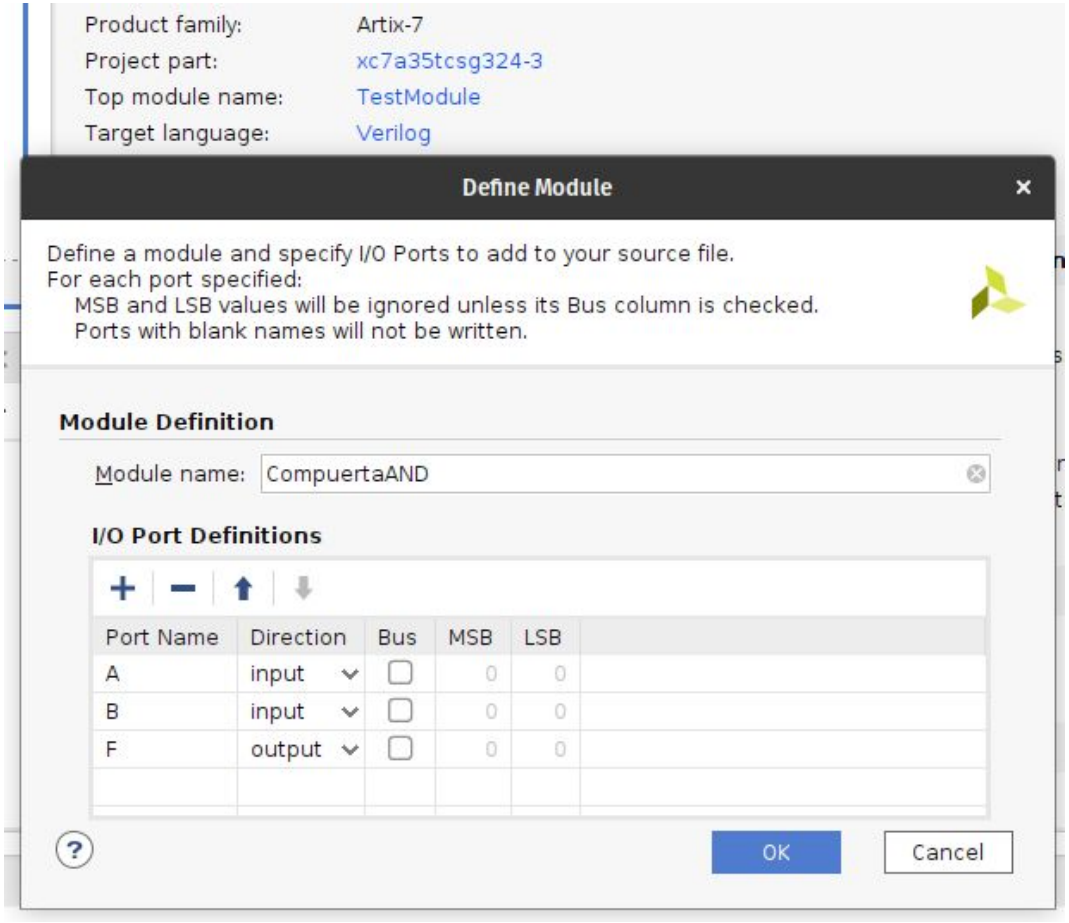
Vivado (creando un módulo)



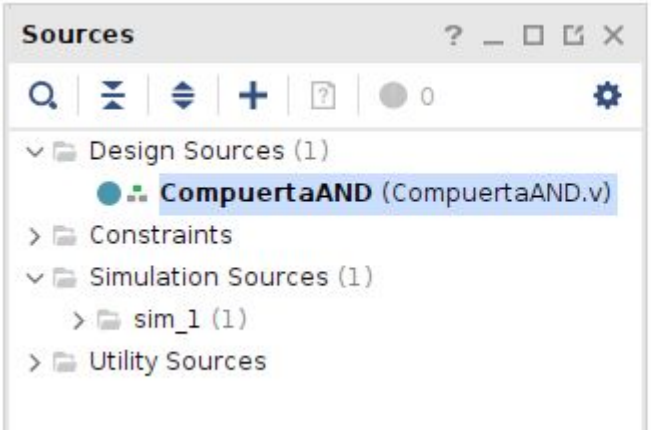
Vivado (creando un módulo)

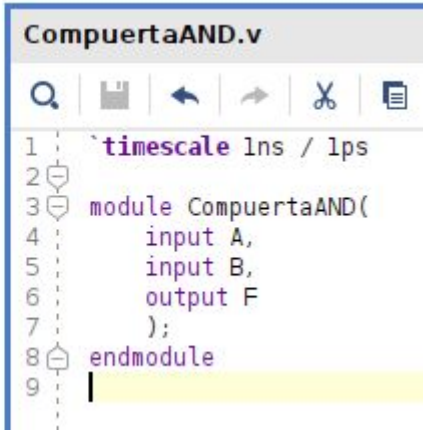


Vivado (creando un módulo)



Vivado (creando un módulo)





Puertos y Tipos de dato

Puertos y tipos de dato (verilog 2001)

Puertos

- **input**
- **output**
- **inout**

Por defecto los puertos son de tipo wire. Los output pueden definirse como '**reg**' (variable). Se puede asignar un rango **[x:y]** a un puerto para indicar un bus. Por defecto los buses se consideran como '**unsigned**' pero puede definirse como '**signed**' de ser necesario.

Valores posibles (wire/reg)

- **0**
- **1**
- **X (unknown)**
- **Z (High Impedance)**

Tipos de datos

- **wire**
- **reg**
- *integer*
- *time*
- *real*
- *string*

Algunos valores sólo tienen sentido en simulaciones.

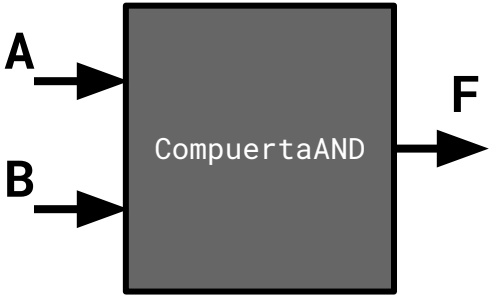
Valores literales

Se forman indicando la cantidad de bits, luego comilla simple y la base seguido del número. Ej: 4 bits (0110) se define como:

- **4'b0110**
- **4'h6**

Lógica combinatoria

Compuerta AND (assign)



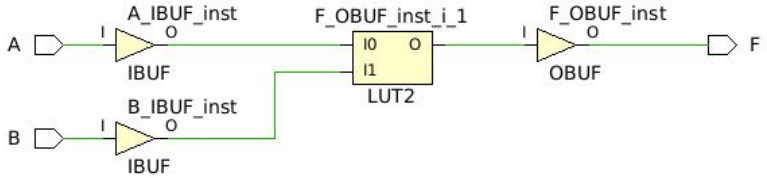

```
module CompuertaAND (
        input A,
        input B,
        output F
        );
    assign F = A & B;
endmodule
```

Entradas **Salida**

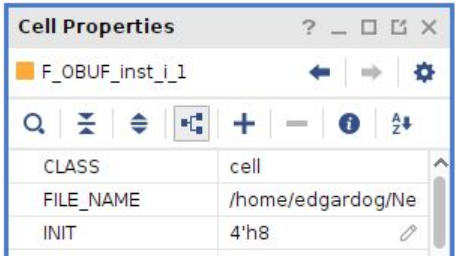
Operadores	
Símbolo	**Operación**
! ~	Negación (Bit)
&	And (Bit)
~&	Nand (Reduc)
|	Or (Bit)
~|	Nor (Reduc)
^	Xor (Bit)
~^	Xnor (Bit)

Vivado (Síntesis y esquemático)

- **SYNTHESIS**
  - Run Synthesis
  - **Open Synthesized Design**
    - Constraints Wizard
    - Edit Timing Constraints
    - Set Up Debug
    - Report Timing Summary
    - Report Clock Networks
    - Report Clock Interaction
    - Report Methodology
    - Report DRC
    - Report Noise
    - Report Utilization
    - Report Power
    - Schematic

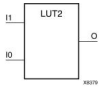


https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/7series_scm.pdf



LUT2

Primitive: 2-Bit Look-Up Table with General Output

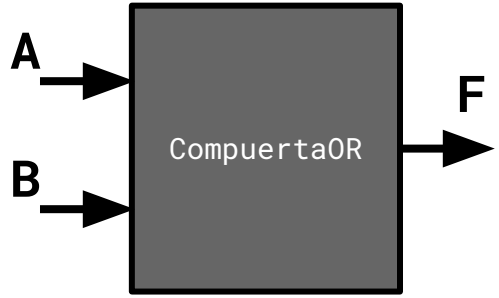


Logic Table

Inputs		Outputs
I1	**I0**	**O**
0	0	INIT[0]
0	1	INIT[1]
1	0	INIT[2]
1	1	INIT[3]

INIT = Binary equivalent of the hexadecimal number assigned to the INIT attribute

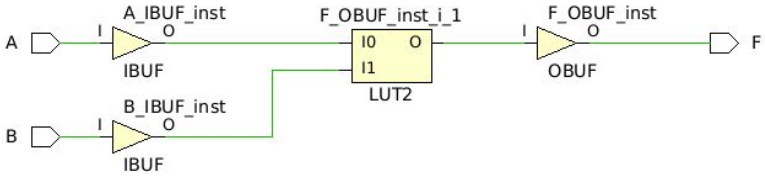
Compuerta OR (assign)



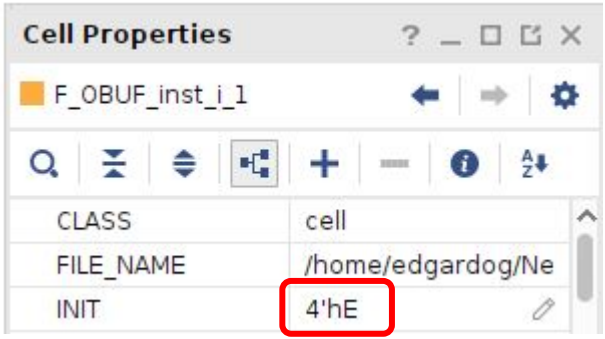
Entradas

Salida

```
module CompuertaOR(
    input A,
    input B,
    output F
    );
    assign F = A | B ;
endmodule
```

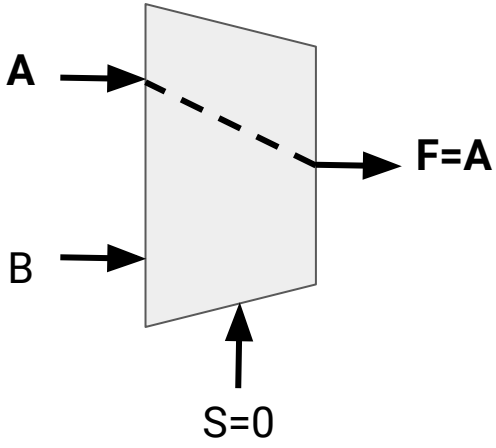


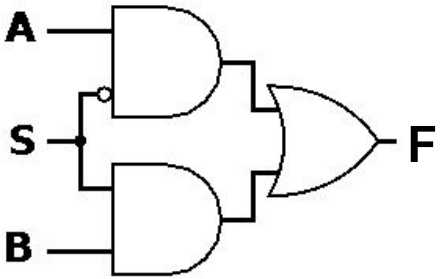
A	B	Or
0	0	0
0	1	1
1	0	1
1	1	1



Curso de Verilog - Básico

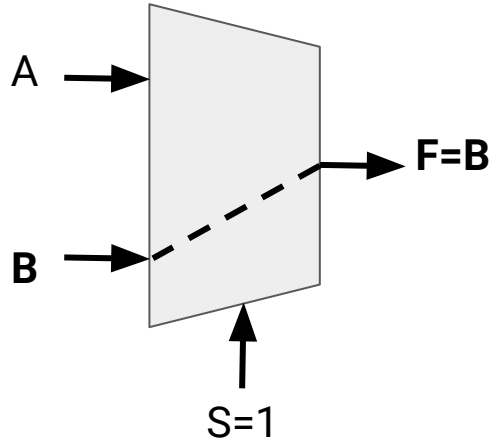
Multiplexor (assign)

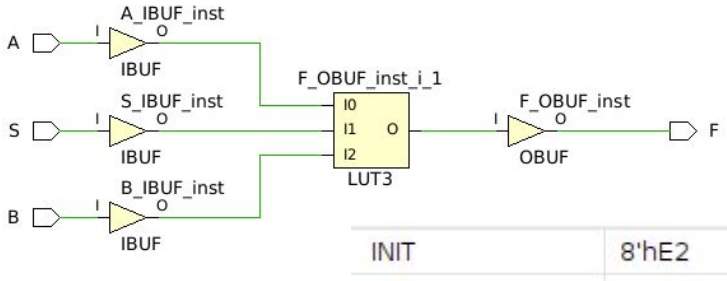




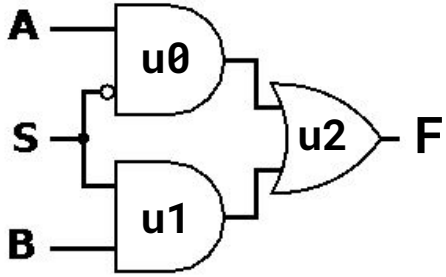
B	S	A	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

```
module Mux2(
    input A,
    input B,
    input S,
    output F
    );
    assign F = ( A & !S) | ( B & S);
endmodule
```

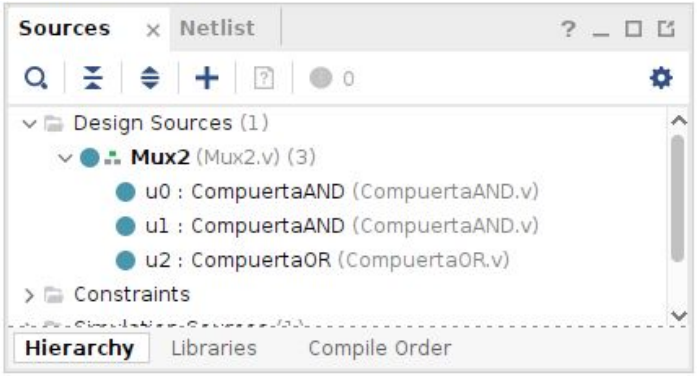


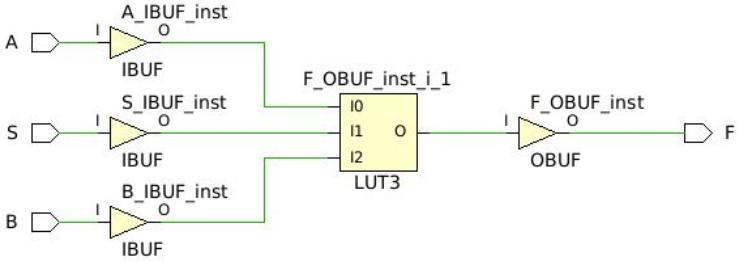


Multiplexor (instancia de módulo)

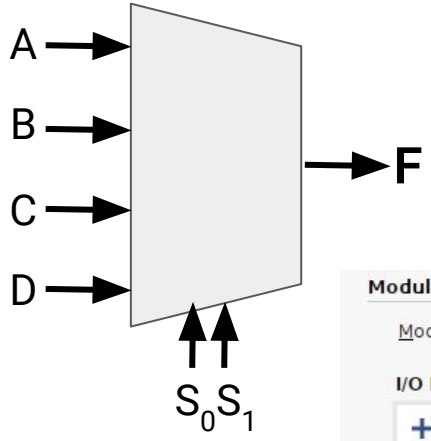


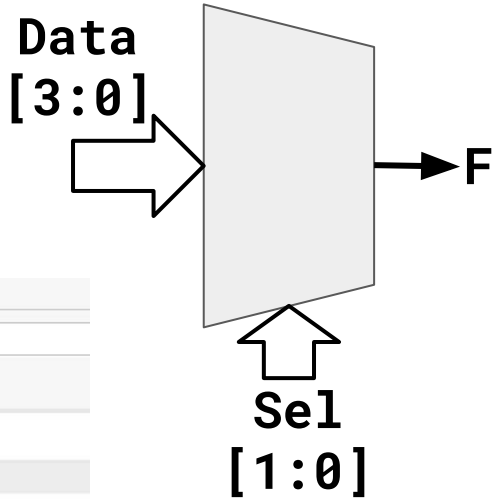

```
module Mux2(
    input A,
    input B,
    input S,
    output F
    );

    wire outAnd0;
    wire outAnd1;
    wire notS;

    assign notS = !S;
    //instancia definiendo nombres
    CompuertaAND u0 (.A(A),
                     .B(notS),
                     .F(outAnd0));
    //instancia ordenada
    CompuertaAND u1 (B,S,outAnd1);

    CompuertaOR u2 (outAnd0,outAnd1,F);
endmodule
```





Multiplexor (buses)





Module Definition

Module name: Mux4

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
Data	input	☑	3	0
Sel	input	☑	1	0
F	output	☐	0	0

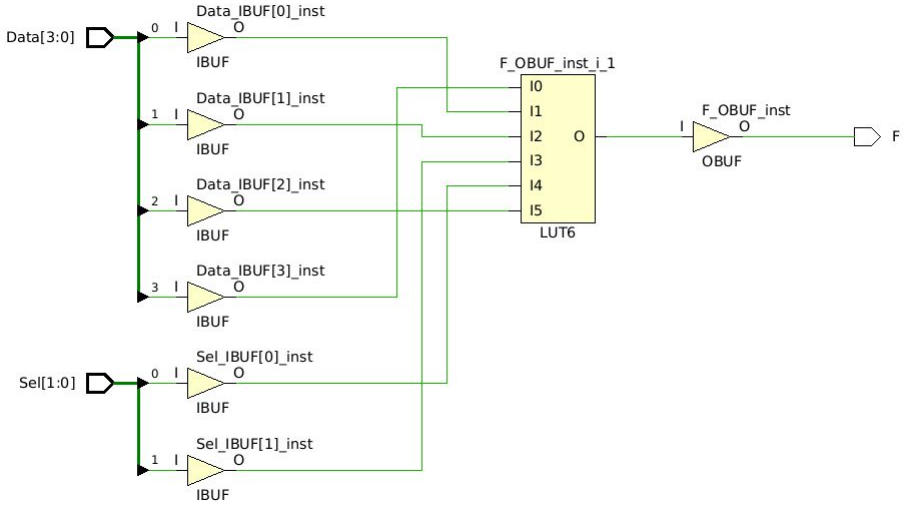
```
module Mux4(
    input [3:0] Data,
    input [1:0] Sel,
    output F
    );
endmodule
```

S_1	S_0	F
0	0	A
0	1	B
1	0	C
1	1	D

Sel	F
00	Data[0]
01	Data[1]
10	Data[2]
11	Data[3]

Multiplexor (buses)

```
module Mux4(
    input [3:0] Data,
    input [1:0] Sel,
    output F
    );

    assign F = ( Data[0] & !Sel[1] & !Sel[0] ) |
               ( Data[1] & !Sel[1] &  Sel[0] ) |
               ( Data[2] &  Sel[1] & !Sel[0] ) |
               ( Data[3] &  Sel[1] &  Sel[0] ) ;
endmodule
```

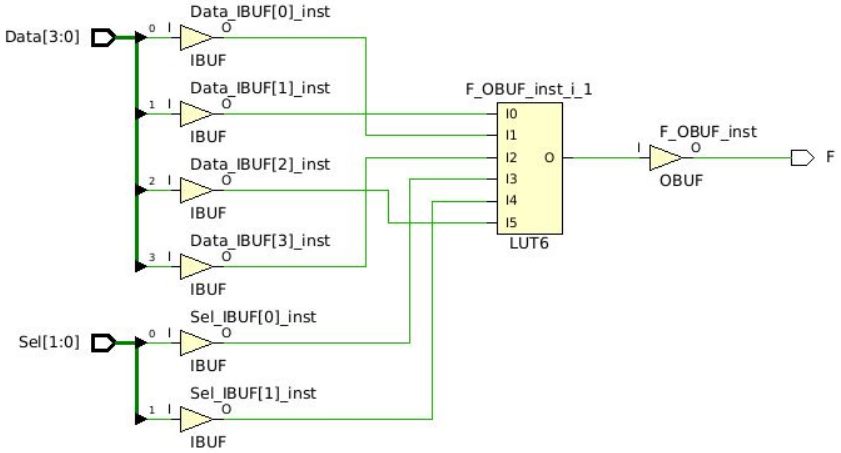


Curso de Verilog - Básico

Multiplexor (adelanto de procesos)

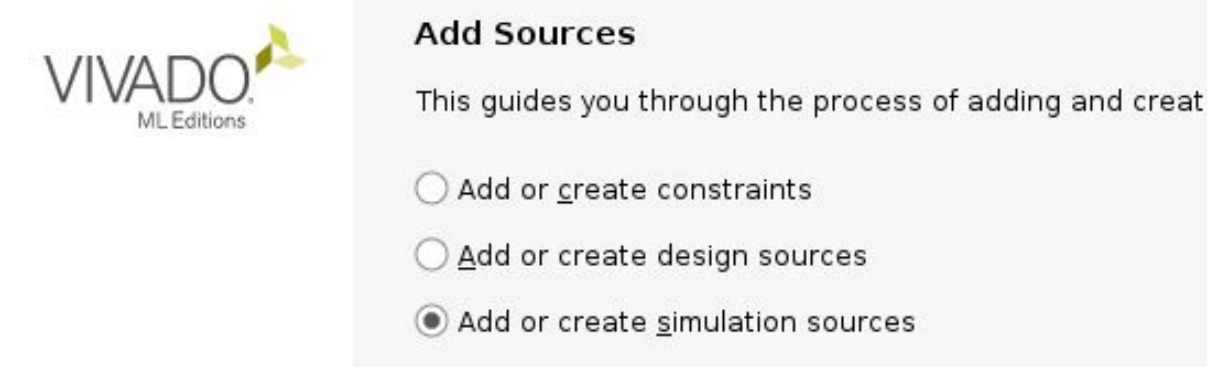
```
module Mux4(
    input [3:0] Data,
    input [1:0] Sel,
    output reg F
    );

    always @*
    begin
        case(Sel)
            2'b00  : F = Data[0];
            2'b01  : F = Data[1];
            2'b10  : F = Data[2];
            default: F = Data[3];
        endcase
    end
endmodule
```

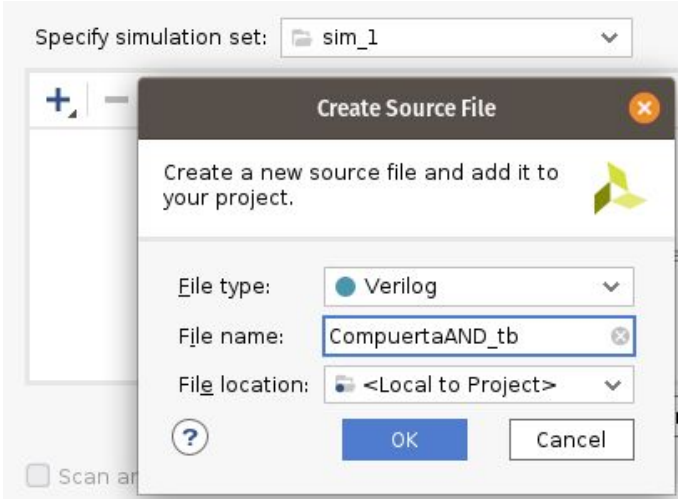
```
module Mux4(
    input [3:0] Data,
    input [1:0] Sel,
    output reg F
    );

    always @*
    begin
        if (Sel == 2'b00 )
            F = Data[0];
        else if (Sel == 2'b01)
            F = Data[1];
        else if (Sel == 2'b10)
            F = Data[2];
        else
            F = Data[3];
    end
endmodule
```

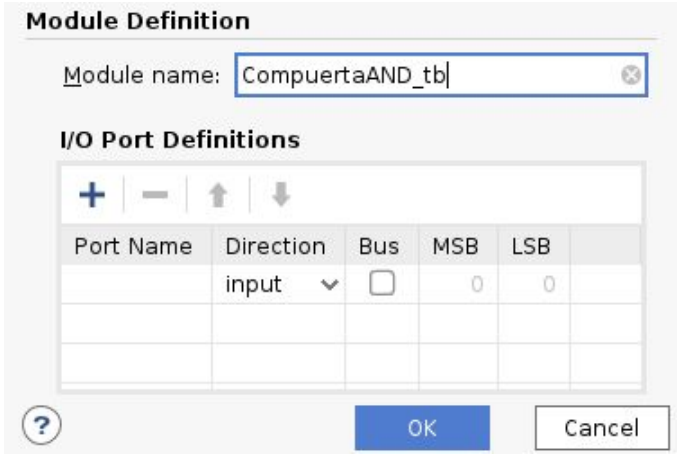


Simulando un módulo

Testbench

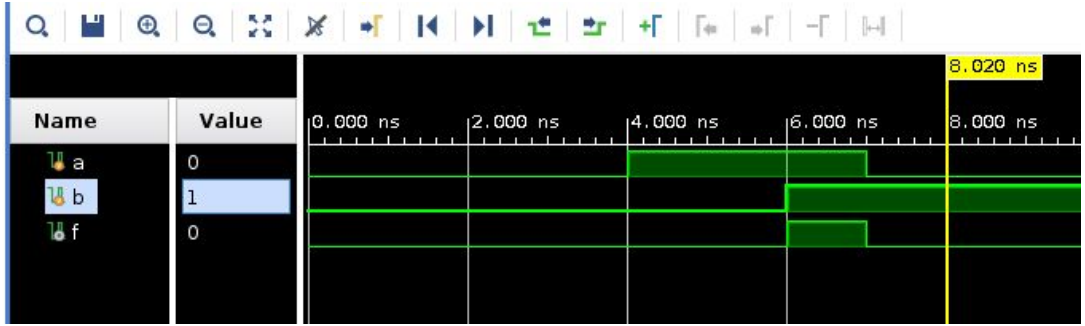






Testbench

```
`timescale 1ns / 1ps

module CompuertaAND_tb();

    reg a,b;
    wire f;

    CompuertaAND U0 ( .A(a), .B(b), .F(f) );

    initial
        begin

        a=0;
        b=0;
        #4;

        a=1;
        #2;

        b=1;
        #1;

        a=0;
        #5;

        end
endmodule
```



Funciones auxiliares

Testbench (archivo de log)

```
module CompuertaAND_tb();

    reg a,b;
    wire f;

    CompuertaAND U0 ( .A(a), .B(b), .F(f) );

    initial begin
        $monitor("Time=%0t , (A[%0d] AND B[%0d]) = F[%0d]",
                 $time,a,b,f);
    end

    initial
        begin

        a=0;
        b=0;
        #4;

        a=1;
        #2;

        b=1;
        #1;

        a=0;
        #5;

        end
endmodule
```

```
# run 1000ns
Time=0 , (A[0] AND B[0]) = F[0]
Time=4000 , (A[1] AND B[0]) = F[0]
Time=6000 , (A[1] AND B[1]) = F[1]
Time=7000 , (A[0] AND B[1]) = F[0]
```

Testbench (funciones auxiliares)

```
module CompuertaAND_tb();

    reg a,b;
    wire f;

    CompuertaAND U0 ( .A(a), .B(b), .F(f) );

    initial
        begin

        a=0;
        b=0;
        #4;

        a=1;
        #2;
        $display("Time=%0t A=%d",$time,a);
        b=1;
        #1;

        a=0;
        #5;

        end
endmodule
```

```
# run 1000ns
Time=6000 A=1
```

```
//Display inserta CRLF
$display("Time=%0t A=%d",$time,a);
//Write NO inserta CRLF
$write("Time=%0t A=%d",$time,a);
//Strobe muestra los valores ANTERIORES a $time
//mientras que $display muestra POSTERIORES
$strobe("Time=%0t A=%d",$time,a);
```

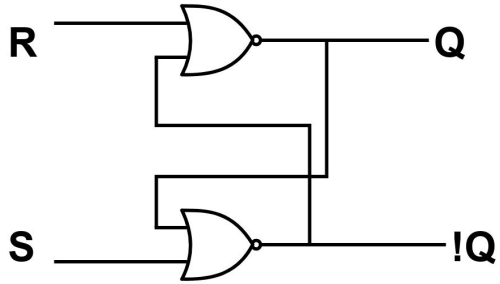
Formateadores

- **%t Tiempo**
- **%h Hexadecimal**
- **%d Decimal**
- **%b Binario**
- **%s String**
- **%f Float**
- **%e Exponencial**

Existen funciones $f{open,close,monitor,display, write,strobe,gets,eof} que interactúan con archivos. $sformat es análoga a fprintf.

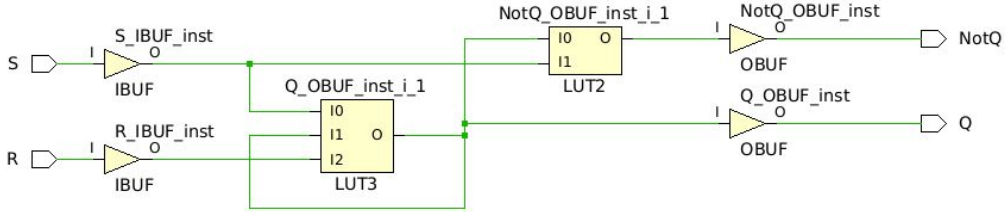
Flip Flops

Latch RS (NOR)




```
module LatchRS(
    input R,
    input S,
    output Q,
    output NotQ
    );
    wire w1,w2;

   CompuertaNOR u0 ( .A(R),.B(w2),.F(w1));
   CompuertaNOR u1 ( .A(S),.B(w1),.F(w2));

   assign Q = w1;
   assign NotQ = w2;

endmodule
```

```
module CompuertaNOR(
    input A,
    input B,
    output F
    );

    assign F = ~|{A,B};
    //La operacion NOR reduce un bus.
    //No es bitwise (entre dos bits)
    //Usando { } se concatenan entradas
    //y luego ~| reduce a un bit
endmodule
```



Latch RS (NOR)

```
module CompuertaNOR(
    input A,
    input B,
    output F
    );
```

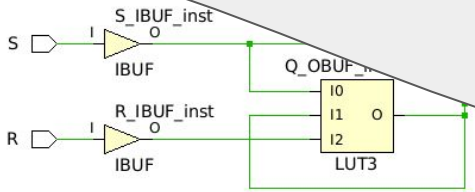
```
~|{A,B};
```

reduce un bus.

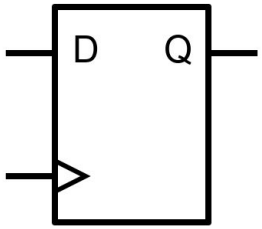
R

Implementación: **Combinatorial Loop Alert: 1 LUT cells form a combinatorial loop. This can create a race condition.** Timing analysis may not be accurate. **The preferred resolution is to modify the design to remove combinatorial logic loops.** If the loop is known and understood, this DRC can be bypassed by..
One net in the loop is Q_OBUF. Please evaluate your design.
The cells in the loop are: Q_OBUF_inst_i_1.

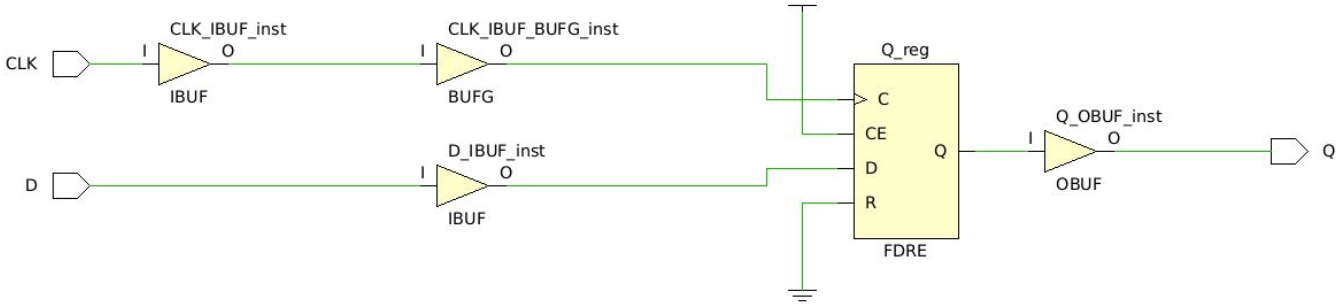
```
NotQ = w2;
ndmodule
```



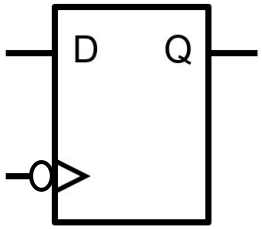
Flip Flop D (flanco ascendente sin reset)



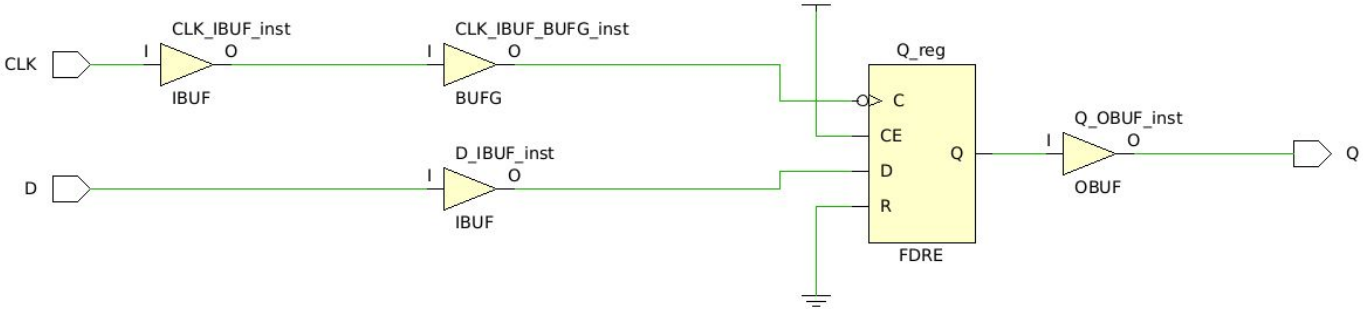

```
module FlipFlopD(
    input D,
    input CLK,
    output reg Q
    );

    always @(posedge CLK)
    begin
        Q <= D;
    end
endmodule
```



Flip Flop D (flanco descendente sin reset)

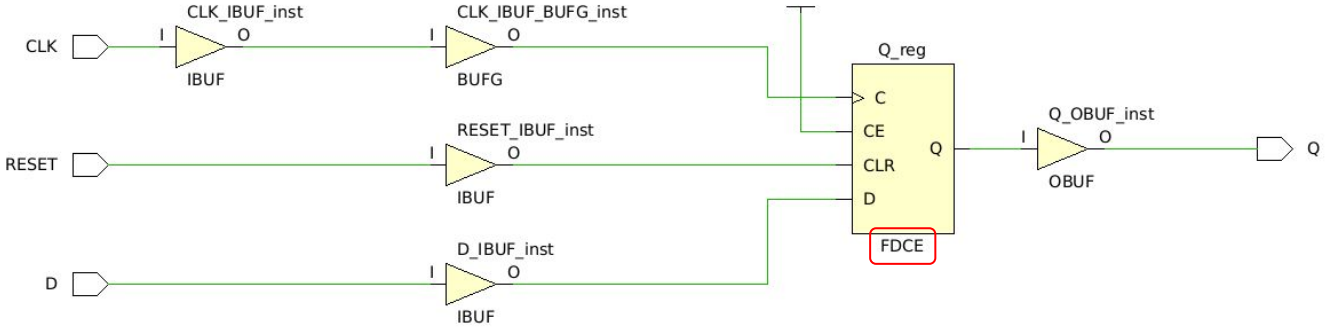



```
module FlipFlopD_desc(
    input D,
    input CLK,
    output reg Q
    );

    always @(negedge CLK)
    begin
        Q <= D;
    end
endmodule
```



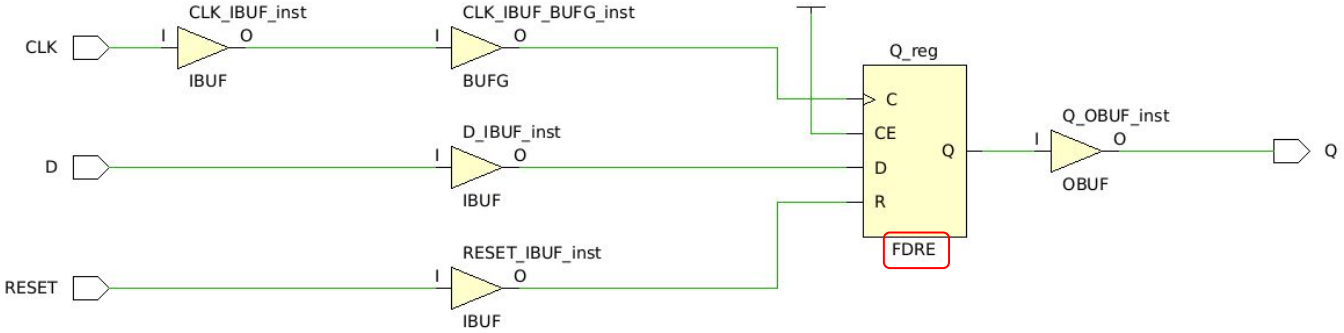
Flip Flop D (Reset Asincrónico)

```
module FlipFlopD_Rasinc(
    input D,
    input CLK,
    input RESET,
    output reg Q
    );

    always @(posedge CLK or posedge RESET)
    begin
        if ( RESET==1'b1)
            Q <= 1'b0;
            else
            Q <= D;
    end
endmodule
```



Flip Flop D (Reset Sincrónico)

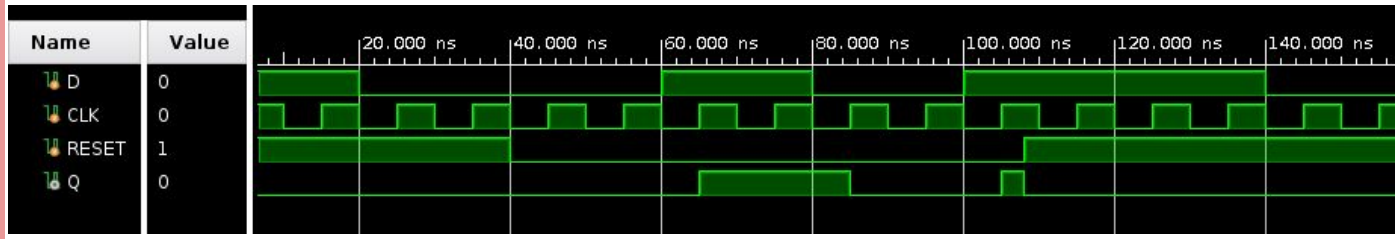
```
module FlipFlopD_Rsinc(
    input D,
    input CLK,
    input RESET,
    output reg Q
    );

    always @(posedge CLK)
    begin
        if ( RESET==1'b1)
            Q <= 1'b0;
            else
            Q <= D;
    end
endmodule
```



Flip Flop D (Simulando)

```
module FlipFlopD_tb();
reg D,CLK,RESET;
wire Q;
FlipFlopD_Rasinc U0(
        .D(D),
        .CLK(CLK),
        .RESET(RESET),
        .Q(Q) );

initial begin
    CLK=0;
    RESET=1;

    forever #5 CLK = !CLK;
end
```

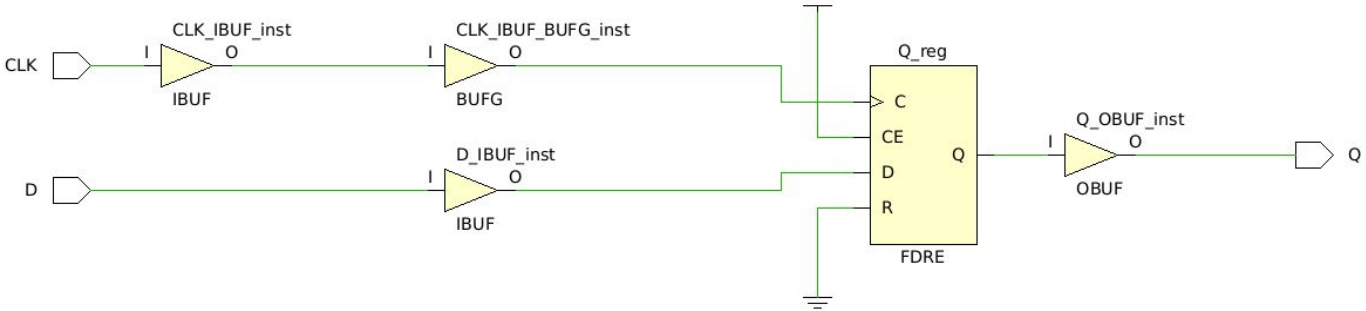
```
initial begin
    D = 1;
    #20;
    D = 0;
    #20;
    RESET=0;
    #20;
    D = 1;
    #20;
    D = 0;
    #20;
    D = 1;
    #8;
    RESET=1;
    #12;
    D = 1;
    #20;
    D = 0;
end
endmodule
```



Asignaciones

Asignación bloqueante

```
module AsigBlock(
    input D,
    input CLK,
    output reg Q
    );

 reg aux;
 always @(posedge CLK)
 begin
    aux = D;
    Q = aux;
 end

endmodule
```

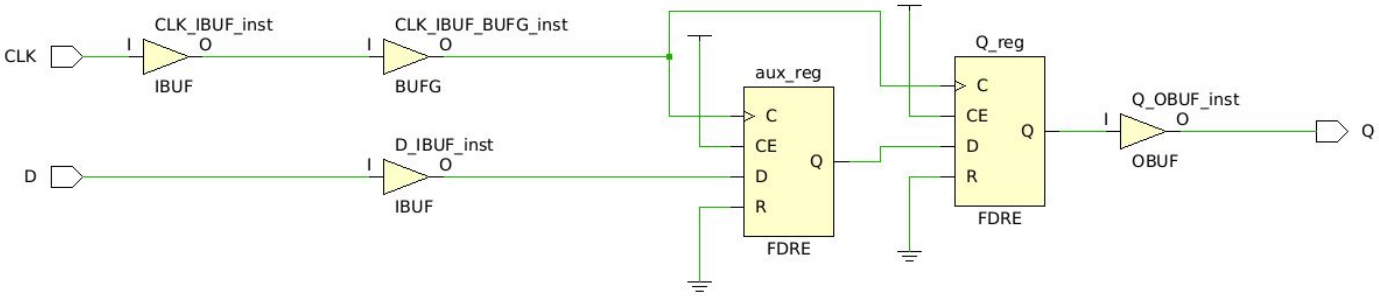


Asignación NO bloqueante

```
module AsigNoBlock(
    input D,
    input CLK,
    output reg Q
    );

 reg aux;
 always @(posedge CLK)
 begin
    aux <= D;
    Q <= aux;
 end

endmodule
```

=

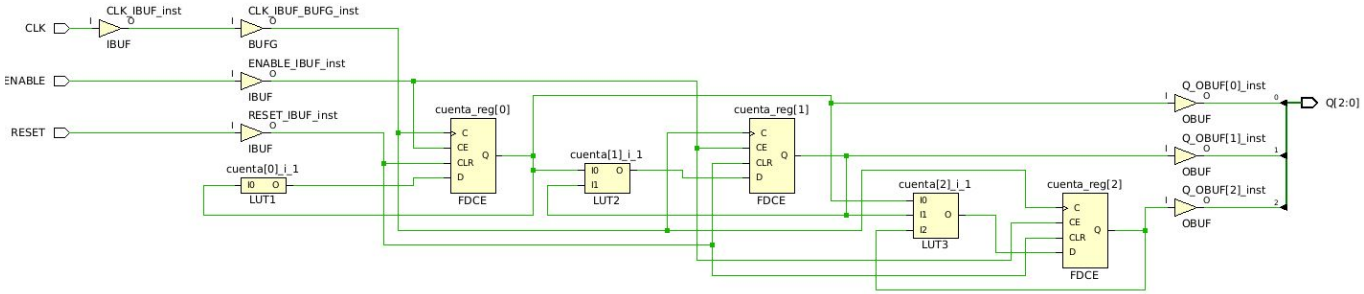
```
module AsigNoBlock(
    input D,
    input CLK,
    output reg Q
    );

 reg aux;
 always @(posedge CLK)
 begin
    Q <= aux;
    aux <= D;
 end

endmodule
```



Contadores

Contador 3 bits (Ascendente)

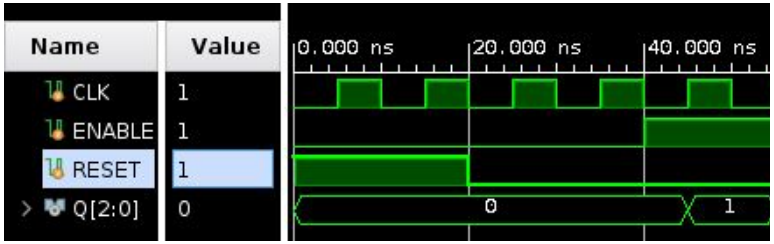
```
module ContadorASC(
    input CLK,
    input RESET,
    input ENABLE,
    output [2:0] Q
    );
reg [2:0] cuenta;
assign Q = cuenta;

always @(posedge CLK or posedge RESET)
begin
    if (RESET)
        cuenta <= 3'd0;
    else
      begin
        if (ENABLE)
            cuenta <= cuenta + 3'd1;
      end
end
endmodule
```

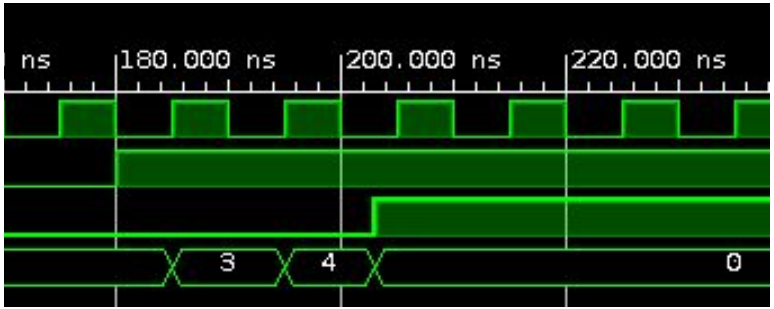


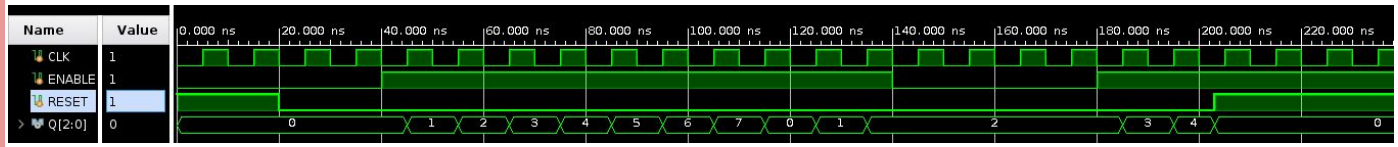
Curso de Verilog - Básico

Contador 3 bits (Simulación)

```
module ContadorASC_tb();
reg CLK,ENABLE,RESET;
wire [2:0] Q;

    ContadorASC U0 (CLK,RESET,ENABLE,Q);
initial begin
    CLK=0;
    ENABLE=0;
    RESET=1;
    forever #5 CLK=~CLK;
end

initial begin
    #20;
    RESET=0;
    #20;
    ENABLE=1;
    #100;
    ENABLE=0;
    #40;
    ENABLE=1;
    #23;
    RESET=1;
end
endmodule
```



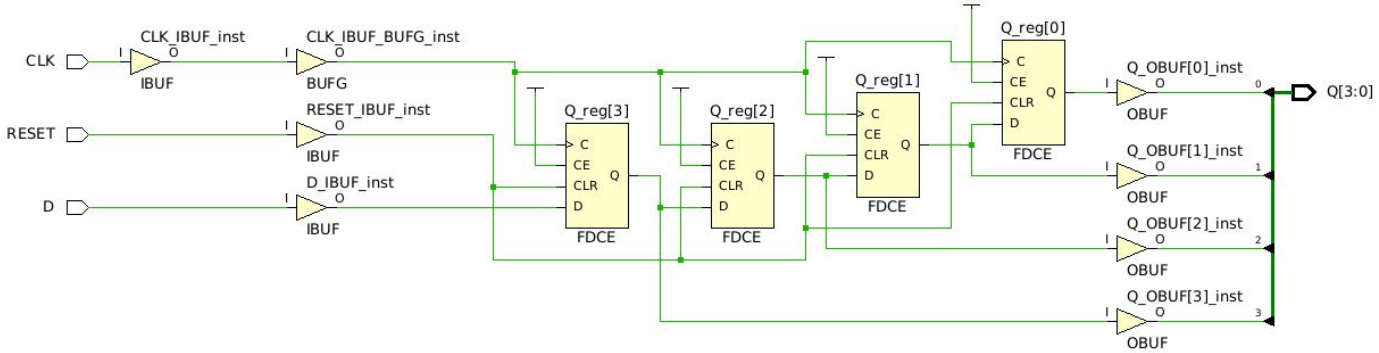




Desplazamiento

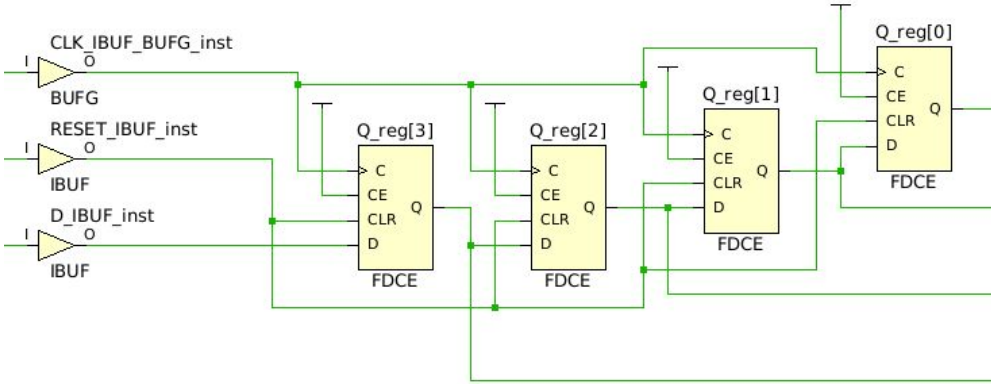
Shifter 4 bits

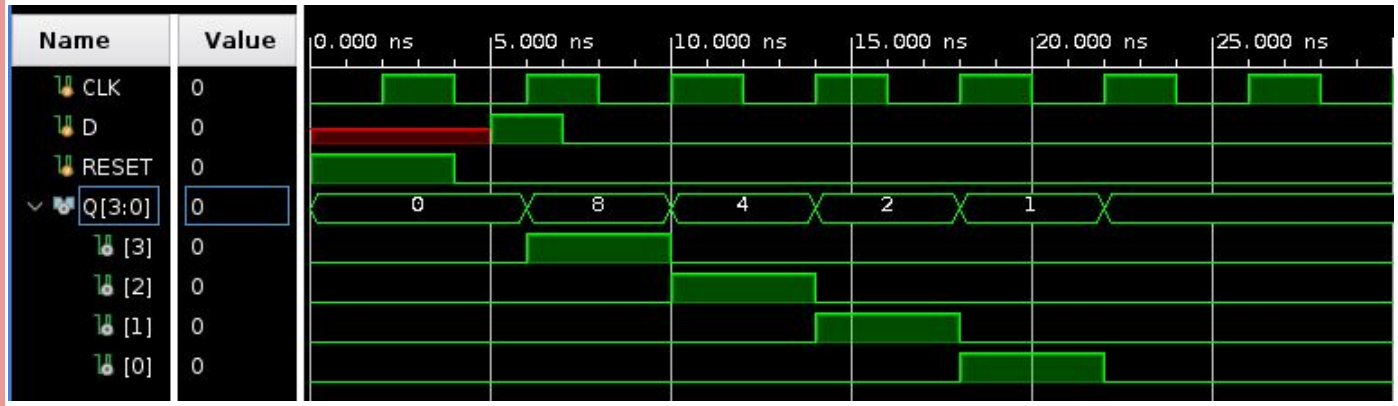
```
module ShiftRight(
    input D,
    input CLK,
    input RESET,
    output reg [3:0]  Q
    );

always @(posedge CLK or posedge RESET)
begin
     if (RESET)
        Q <= 0;
     else
      Q <= { D , Q[3:1]};

end
endmodule
```



Shifter 4 bits (testbench)

```
module ShiftRight_tb();
reg CLK,D,RESET;
wire [3:0] Q;

ShiftRight U0 (D,CLK,RESET,Q);
initial begin
    CLK=0;
    RESET=1;
    forever #2 CLK=~CLK;
end

initial begin
    #4;
    RESET=0;
    #1;
    D=1;
    #2;
    D=0;
end
endmodule
```

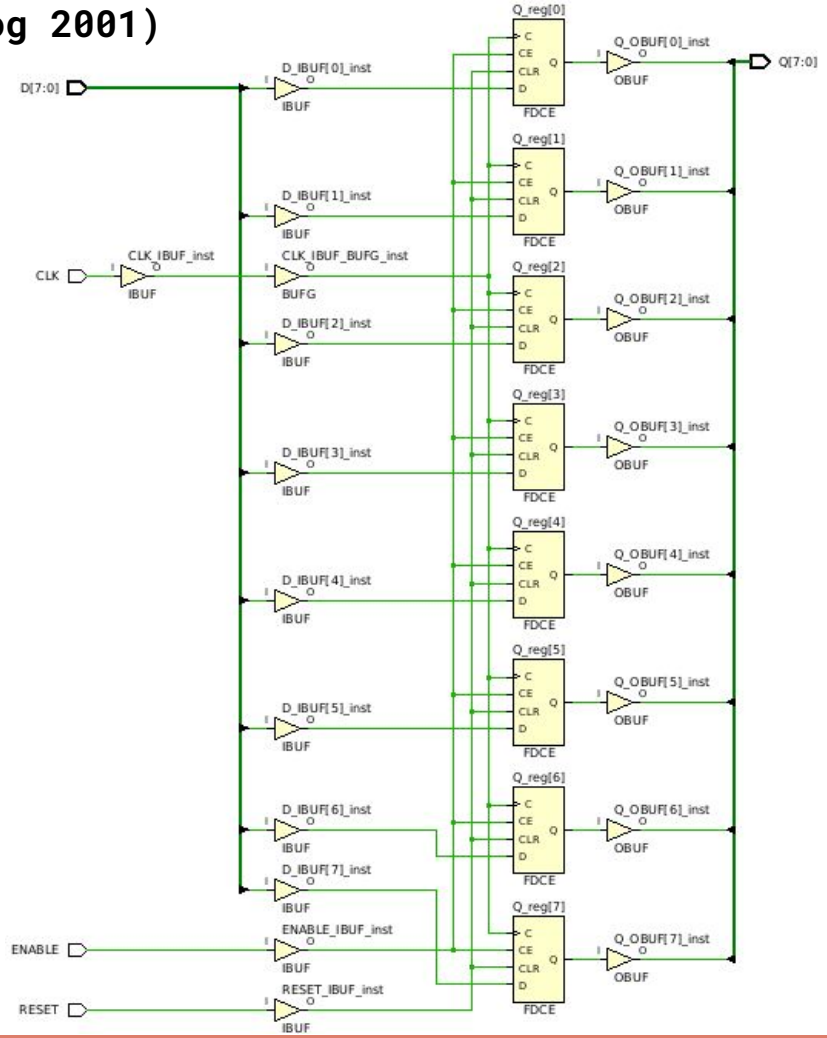




Parámetros

Definir parámetro (Verilog 2001)

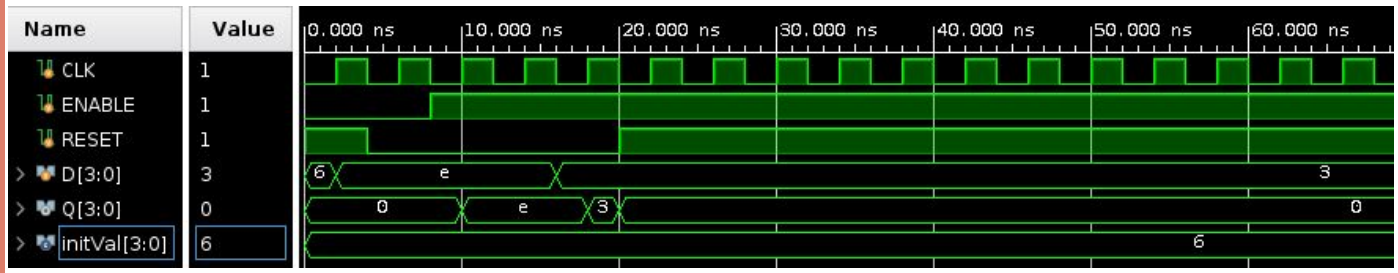
```
module LatchRegister
#(parameter SIZE=8)
(
    input [SIZE-1:0] D,
    input CLK,
    input RESET,
    input ENABLE,
    output reg [SIZE-1:0] Q
    );

always@(posedge CLK or posedge RESET)
begin
    if (RESET)
        Q <= 0;
    else
        if (ENABLE)
            Q <= D;
end

endmodule
```



Cambiar parámetro en instancia (Verilog 2001)

```
module LatchRegister_tb
#(parameter [3:0] initVal = 4'b0110)
();

reg CLK,ENABLE,RESET;
reg [3:0] D;
wire [3:0] Q;

    LatchRegister #(.SIZE(4) ) U0
        (D,CLK,RESET,ENABLE,Q);

 initial begin
    CLK=0;
    ENABLE=0;
    RESET=1;
    D=initVal;
    forever #2 CLK=~CLK;
end
```

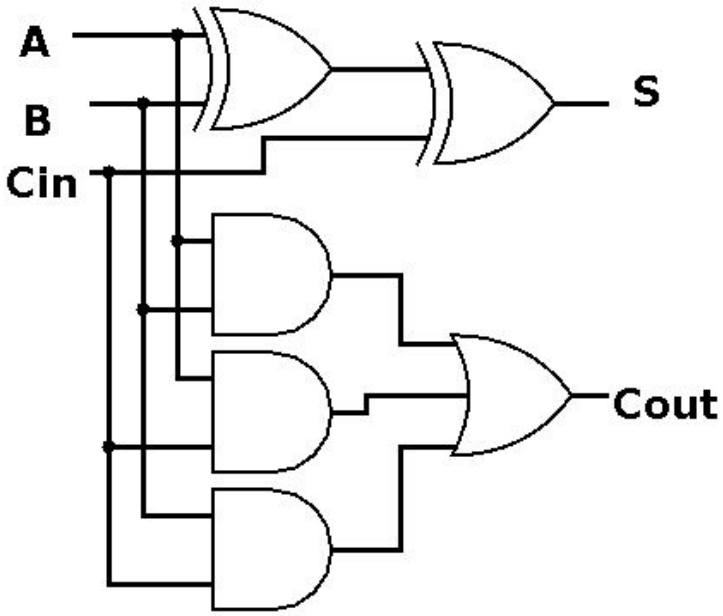
```
initial begin
    #2;
    D=4'b1110;
    #2;
    RESET=0;
    #4;
    ENABLE=1;
    #8;
    D=4'b0011;
    #4;
    RESET=1;
end
endmodule
```



Instanciar elementos internos

Full Adder (LUT)

entrada			salida	
C_{IN}	A	B	C_{OUT}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

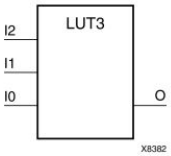


C_{OUT}=E8 S=96

https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/7series_hdl.pdf

LUT3

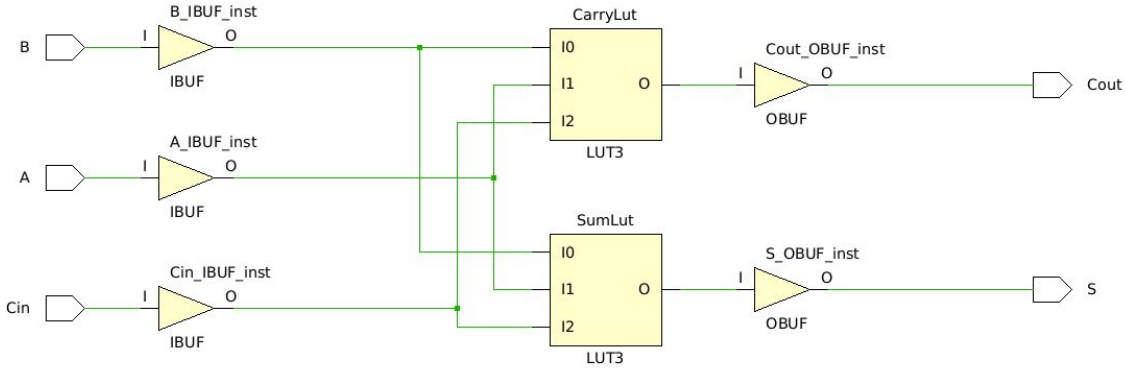
Primitive: 3-Bit Look-Up Table with General Output




```
LUT3 #(
   .INIT(8'h00)  // Specify LUT Contents
) LUT3_inst (
   .O(O),   // LUT general output
   .I0(I0), // LUT input
   .I1(I1), // LUT input
   .I2(I2)  // LUT input
);
```

Full Adder (LUT)

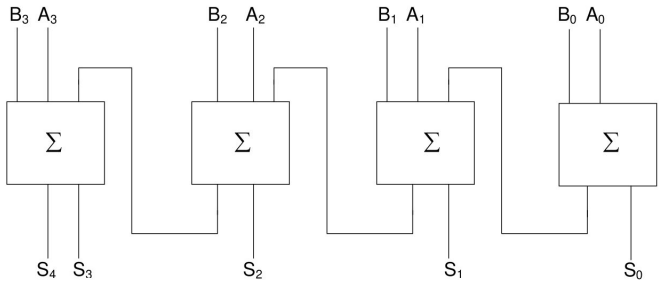
```
module FullAdderLut(
    input A,
    input B,
    input Cin,
    output S,
    output Cout
    );

LUT3  #(.INIT(8'hE8)) CarryLut (
    .O(Cout),
    .I0(B),
    .I1(A),
    .I2(Cin) );

LUT3  #(.INIT(8'h96)) SumLut (
    .O(S),
    .I0(B),
    .I1(A),
    .I2(Cin) );
endmodule
```

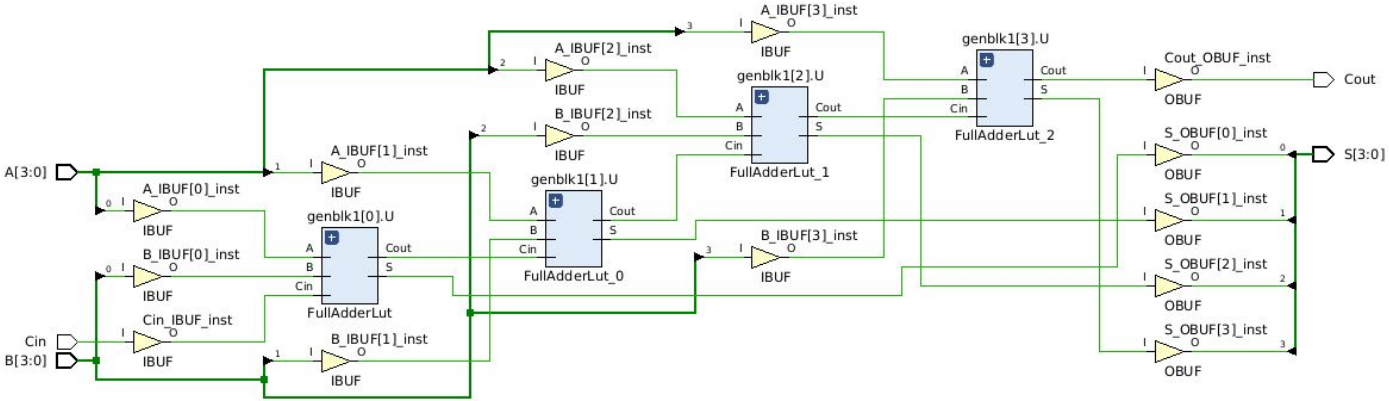


For Generate

Hex Adder (generate for)

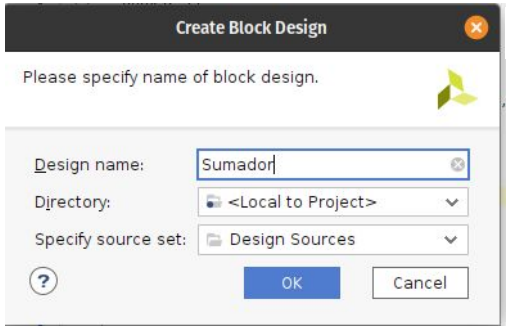
```
module HexAdder(
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output [3:0] S,
    output Cout
    );
wire [4:0] carry;
genvar i;
assign carry[0] = Cin;
assign Cout = carry[4];
generate
    for (i=0;i<4;i=i+1) begin
        FullAdderLut U (A[i],B[i],carry[i],S[i],carry[i+1]);
    end
endgenerate
endmodule
```



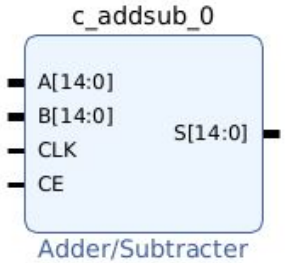


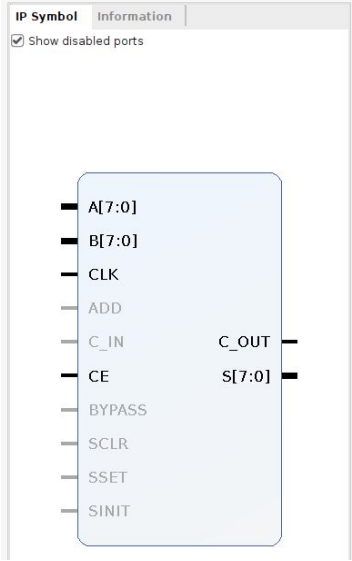
IP Cores

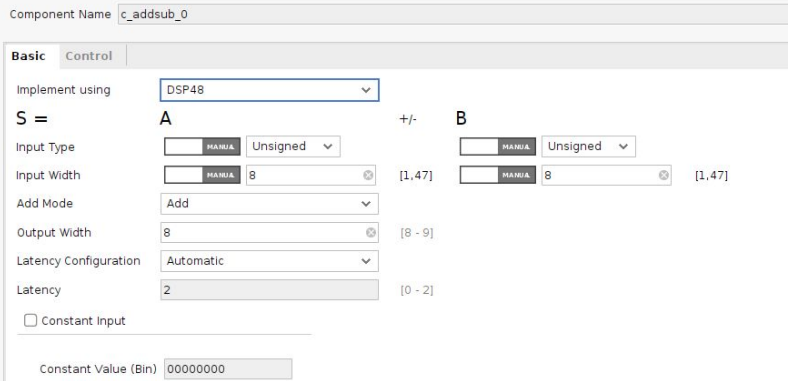
IP Cores

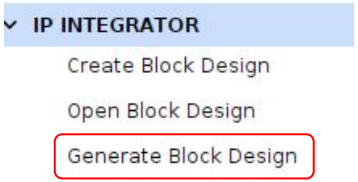


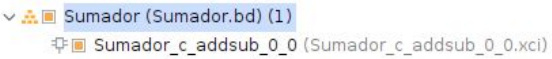
This design is empty. Press the + button to add IP.



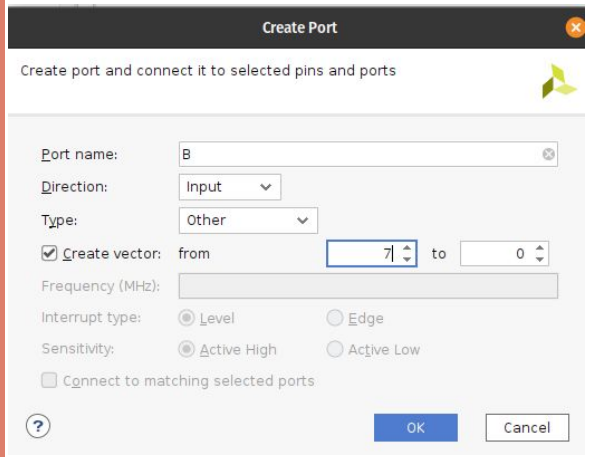


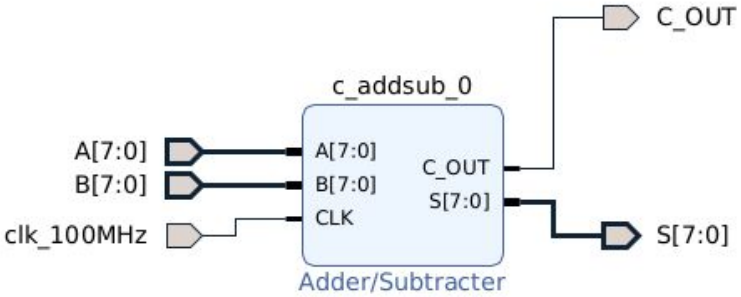






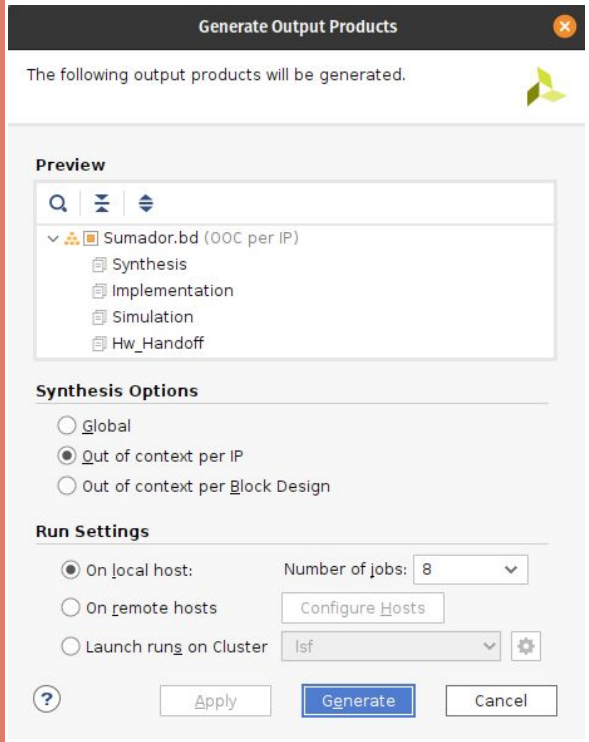
IP Cores (Create port)

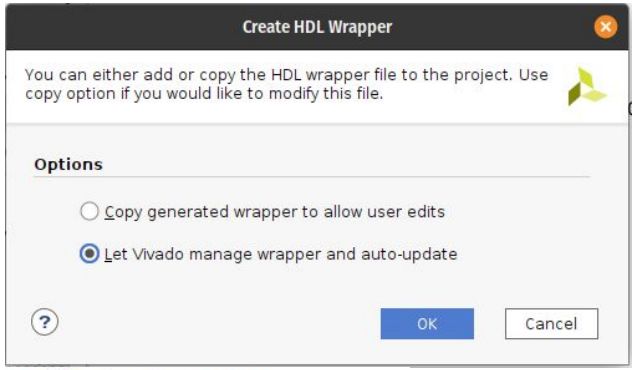




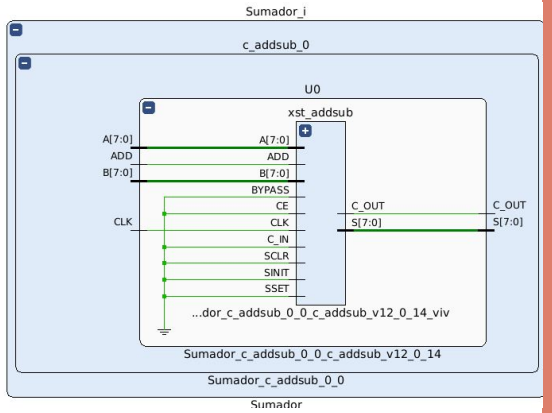
https://www.xilinx.com/support/documentation/ip_documentation/addsub/v12_0/pg120-c-addsub.pdf

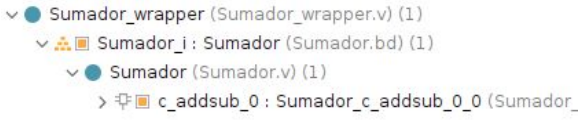
IP Cores






```
module Sumador_wrapper
   (A,
    B,
    C_OUT,
    S,
    clk_100MHz);
  input [7:0]A;
  input [7:0]B;
  output C_OUT;
  output [7:0]S;
  input clk_100MHz;

  wire [7:0]A;
  wire [7:0]B;
  wire C_OUT;
  wire [7:0]S;
  wire clk_100MHz;

  Sumador Sumador_i
       (.A(A),
        .B(B),
        .C_OUT(C_OUT),
        .S(S),
        .clk_100MHz(clk_100MHz));
endmodule
```



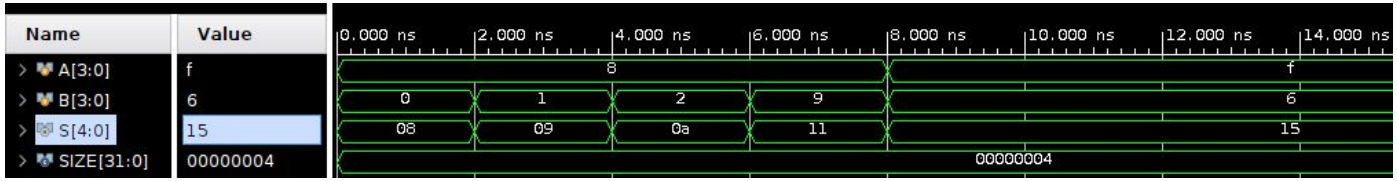


Sumador Simple

Sumador Simple

```
module SumadorSimple
#(parameter SIZE=8)
(
    input [SIZE-1:0] A,
    input [SIZE-1:0] B,
    output[SIZE  :0] S
    );
wire [SIZE:0] Amas,Bmas;
assign Amas = {0,A};
assign Bmas = {0,B};
assign S = Amas + Bmas;
endmodule
```

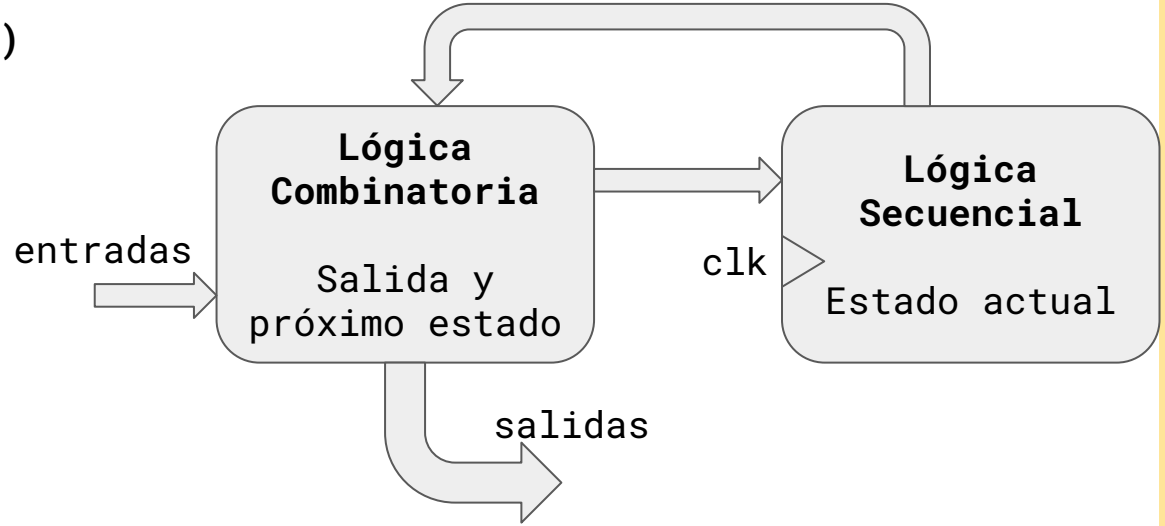
```
module SumadorSimple_tb
#(parameter SIZE=4)
();
reg [SIZE-1:0] A,B;
wire [SIZE :0] S;

SumadorSimple #(.SIZE(SIZE)) U0 (A,B,S);

initial begin
A=4'h8;
B=4'h0;
#2;
A=4'h8;
B=4'h1;
#2;
A=4'h8;
B=4'h2;
#2;
A=4'h8;
B=4'h9;
#2;
A=4'hF;
B=4'h6;
#2;
end
endmodule
```

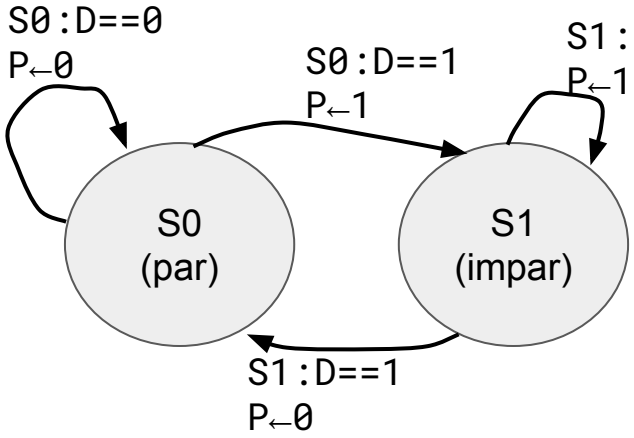


Curso de Verilog - Básico

FSM Mealy

FSM (Mealy)



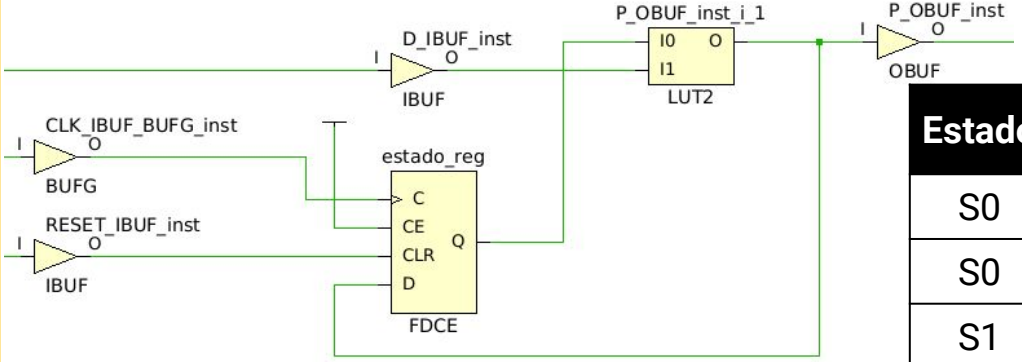


Estado	D	P	Nuevo estado
S0	0	0	S0
S0	1	1	S1
S1	0	1	S1
S1	1	0	S0

FSM (Mealy)

```
module MealyFSM(
    input CLK,
    input RESET,
    input D,
    output reg P
    );
//Definimos valor para guardar el estado
parameter S0=0, S1=1;

reg estado,nuevoEstado;

always @(posedge CLK or posedge RESET)
if (RESET)
    estado <= S0;
else
    estado <= nuevoEstado;
```

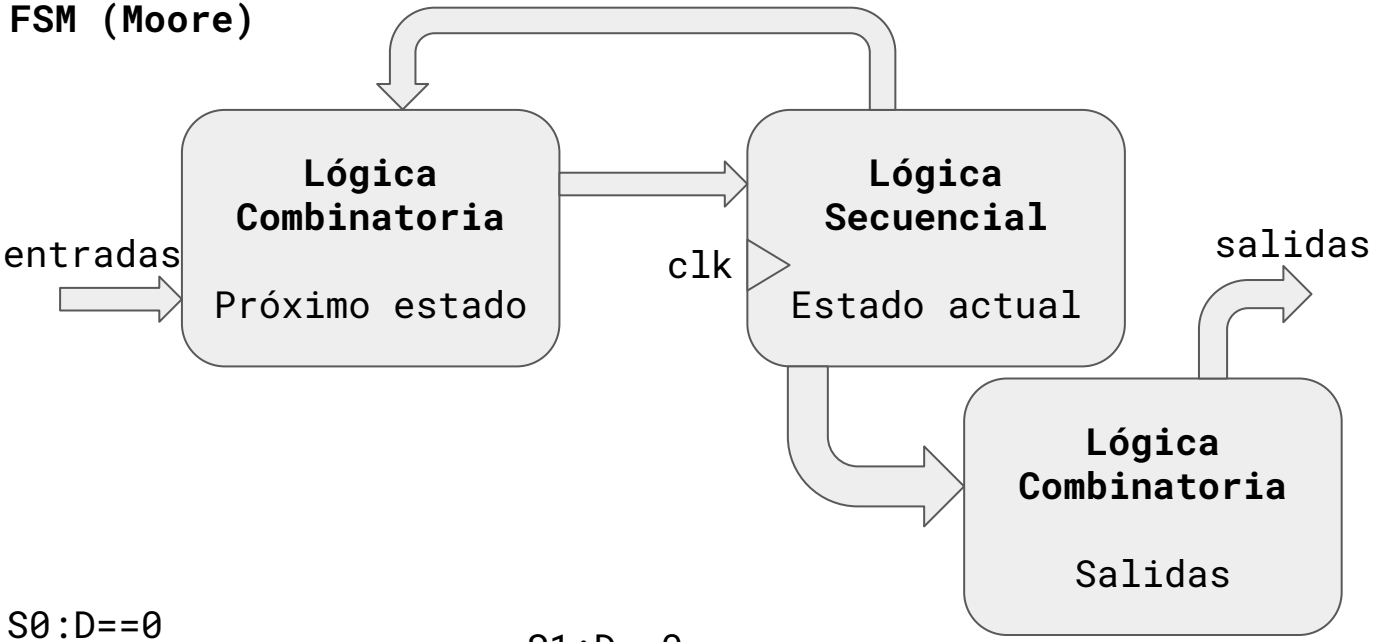
```
always @(estado or D)
begin
    P=1'b0;
    case (estado)
        S0: if (D)
            begin
                P=1; nuevoEstado=S1;
            end
            else
                //P=0 al principio del always.
                nuevoEstado=S0;
      S1: if (D)
                //P=0 al principio del always.
                nuevoEstado=S0;
            else
            begin
                P=1; nuevoEstado=S1;
            end
      default:
          nuevoEstado=S0;
      endcase
end
```

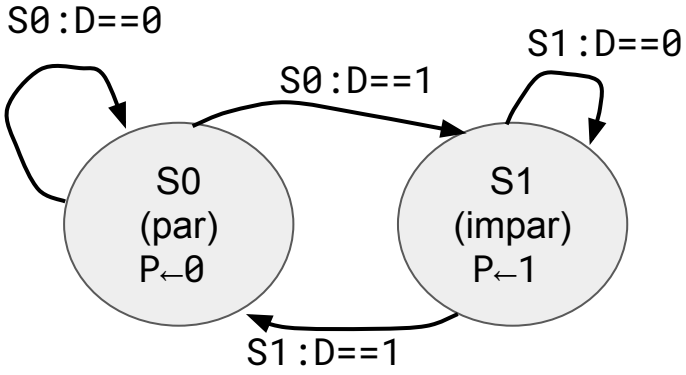


Estado	D	P_{N+1}	Nuevo estado
S0	0	0	S0
S0	1	1	S1
S1	0	1	S1
S1	1	0	S0

FSM Moore

FSM (Moore)





Estado	D	Nuevo estado
S0	0	S0
S0	1	S1
S1	0	S1
S1	1	S0

FSM (Moore)

```
module MooreFSM(
    input CLK,
    input RESET,
    input D,
    output reg P
    );
//Definimos valor para guardar el estado
parameter S0=0, S1=1;

reg estado,nuevoEstado;

always @(posedge CLK or posedge RESET)
if (RESET)
    estado <= S0;
else
    estado <= nuevoEstado;

always @(estado)
begin
    case (estado)
        S0: P=0;
        S1: P=1;
    endcase
end

always @(estado or D)
begin
    nuevoEstado=S0;
    case (estado)
        S0: if(D)
              nuevoEstado=S1;
        S1: if(!D)
              nuevoEstado=S1;
    endcase
end
endmodule
```

Estado	D	Nuevo estado
S0	0	S0
S0	1	S1
S1	0	S1
S1	1	S0