

# Digital Electronics

Diego Trapero

## Table of contents

<b>1</b>	<b>Digital Electronics</b>	<b>3</b>
<b>2</b>	<b>Binary</b>	<b>3</b>
2.1	Binary - Decimal Conversion . . . . .	4
2.1.1	Binary to Decimal Conversion . . . . .	4
2.1.2	Decimal to Binary Conversion . . . . .	4
2.2	Binary Arithmetic . . . . .	4
2.2.1	Binary Addition . . . . .	4
2.2.2	Binary Substraction . . . . .	5
2.2.3	Binary Multiplication . . . . .	5
2.2.4	Binary Division . . . . .	6
2.2.5	Complements . . . . .	6
2.3	Signed numbers . . . . .	7
2.4	Boolean Algebra . . . . .	7
<b>3</b>	<b>Digital components</b>	<b>7</b>
3.1	Logic Gates . . . . .	8
3.1.1	NOT Gate . . . . .	8
3.1.2	AND Gate . . . . .	8
3.1.3	OR Gate . . . . .	8
3.1.4	NAND Gate . . . . .	9
3.1.5	NOR Gate . . . . .	9
3.1.6	XOR Gate . . . . .	9
3.1.7	XNOR Gate . . . . .	10
3.2	Adders . . . . .	10
3.2.1	Half adder . . . . .	10
3.2.2	Full adder . . . . .	11
3.3	Comparators . . . . .	12
3.3.1	Equality Comparators . . . . .	12
3.4	Multiplexers, Demultiplexers, Encoders, Decoders . . . . .	12
3.4.1	Multiplexer . . . . .	12
3.4.2	Demultiplexer . . . . .	13
3.4.3	Encoder . . . . .	13
3.4.4	Decoder . . . . .	13

3.5	Bistables . . . . .	13
3.5.1	Latches . . . . .	14
3.5.2	Flip-Flops . . . . .	16
3.6	Finite State Machines . . . . .	20
3.7	Lookup tables . . . . .	21
<b>4</b>	<b>Reference tables</b>	<b>22</b>
4.1	Logic Gates . . . . .	23
4.2	Bistables . . . . .	24
<b>5</b>	<b>Bibliography</b>	<b>25</b>
<b>6</b>	<b>More</b>	<b>25</b>

# 1 Digital Electronics

## 2 Binary

The binary numeral system, or base-2 numeral system, represents numeric values using two symbols: typically 0 and 1. More specifically, the usual base-2 system is a positional notation with a radix of 2. #

$[x]_{10}$	$[x]_2$	$[x]_{16}$
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

### Powers of 2

$n$	$2^n$
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	264
9	512
10	1024

### Numbers you can represent with $n$ bits

System	Range
--------	-------

Natural binary	0 to $2^n - 1$
One's complement	$-2^{n-1} - 1$ to $2^{n-1} - 1$
Two's complement	$-2^{n-1}$ to $2^{n-1} - 1$

## 2.1 Binary - Decimal Conversion

### 2.1.1 Binary to Decimal Conversion

Digit weighting

### 2.1.2 Decimal to Binary Conversion

Sum of weights

Repeated division by 2

## 2.2 Binary Arithmetic

### 2.2.1 Binary Addition

Addition rules

- $0 + 0 = 0$

- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 10$  (1 and carry 1)

$$\begin{array}{r}
 1\ 1\ 1\ 1\ 1 \quad \text{(carried digits)} \\
 0\ 1\ 1\ 0\ 1 \\
 +\ 1\ 0\ 1\ 1\ 1 \\
 \hline
 = 1\ 0\ 0\ 1\ 0\ 0 = 36
 \end{array}$$

### 2.2.2 Binary Subtraction

#### Subtraction rules

- $0 - 0 = 0$
- $0 - 1 = 0$
- $1 - 0 = 1$
- $10 - 1 = 1$

$$\begin{array}{r}
 \quad \quad * \quad * \quad * \quad * \quad \text{(starred columns are borrowed from)} \\
 1\ 1\ 0\ 1\ 1\ 1\ 0 \\
 - \quad 1\ 0\ 1\ 1\ 1 \\
 \hline
 = 1\ 0\ 1\ 0\ 1\ 1\ 1
 \end{array}$$

### 2.2.3 Binary Multiplication

#### Multiplication rules

- $0 \cdot 0 = 0$
- $0 \cdot 1 = 0$
- $1 \cdot 0 = 0$
- $1 \cdot 1 = 1$

$$\begin{array}{r}
\begin{array}{r}
1\ 0\ 1\ 1\quad (A) \\
\times\ 1\ 0\ 1\ 0\quad (B) \\
\hline
0\ 0\ 0\ 0\quad \leftarrow \text{Corresponds to the rightmost 'zero' in B} \\
+ \quad 1\ 0\ 1\ 1\quad \leftarrow \text{Corresponds to the next 'one' in B} \\
+ \quad 0\ 0\ 0\ 0 \\
+ \quad 1\ 0\ 1\ 1 \\
\hline
= 1\ 1\ 0\ 1\ 1\ 1\ 0
\end{array}
\end{array}$$

$$\begin{array}{r}
\begin{array}{r}
1\ 0\ 1\ .\ 1\ 0\ 1\quad A\ (5.625\ \text{in decimal}) \\
\times\ 1\ 1\ 0\ .\ 0\ 1\quad B\ (6.25\ \text{in decimal}) \\
\hline
1\ .\ 0\ 1\ 1\ 0\ 1\quad \leftarrow \text{Corresponds to a 'one' in B} \\
+ \quad 0\ 0\ .\ 0\ 0\ 0\ 0\quad \leftarrow \text{Corresponds to a 'zero' in B} \\
+ \quad 0\ 0\ 0\ .\ 0\ 0\ 0\ 0 \\
+ \quad 1\ 0\ 1\ 1\ .\ 0\ 1 \\
+ \quad 1\ 0\ 1\ 1\ 0\ .\ 1 \\
\hline
= 1\ 0\ 0\ 0\ 1\ 1\ .\ 0\ 0\ 1\ 0\ 1\quad (35.15625\ \text{in decimal})
\end{array}
\end{array}$$

## 2.2.4 Binary Division

## 2.2.5 Complements

In mathematics and computing, the method of complements is a technique used to subtract one number from another using only addition of positive numbers. This method was commonly used in mechanical calculators and is still used in modern computers. To subtract a number  $y$  (the subtrahend) from another number  $x$  (the minuend), the radix complement of  $y$  is added to  $x$  and the initial '1' of the result is discarded. Discarding the initial '1' is especially convenient on calculators or computers that use a fixed number of digits: there is nowhere for it to go so it is simply lost during the calculation. #

### 1's Complement

The One's complement of a binary number is defined as the value obtained by inverting all the bits in the binary representation of the number (swapping 0s for 1s and vice-versa). #

The ones' complement of the number then behaves like the negative of the original number in some arithmetic operations, although it presents some problems that are solved by the two's complement, like the offset by -1.

### Examples

$$1C[11010] = 00101$$

## 2's Complement

Obtain the 2's complement

- From the 1's complement

$$2C[x] = 1c[x] + 1$$

- Working from LSB to MSB

## 2.3 Signed numbers

Sign-magnitude form

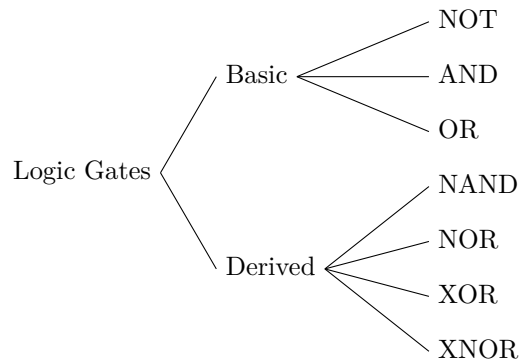
1's complement form

2's complement form

## 2.4 Boolean Algebra

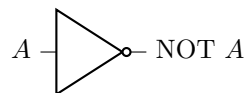
## 3 Digital components

## 3.1 Logic Gates



- **Basic gates** perform the three basic boolean operations: conjunction (AND), disjunction (OR) and negation (NOT). Only one the two other gates is necessary in conjunction with the NOT gate to implement every boolean function. The other operation can be expressed in terms of the other two.
- **Derived gates** perform derived boolean operations, operations that can be composed from basic operations. The most common are NAND, NOR, XOR, and XNOR gates.

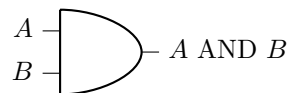
### 3.1.1 NOT Gate



**Rule:** The NOT gate output is the complementary of the input

A	NOT A
0	1
1	0

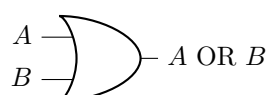
### 3.1.2 AND Gate



**Rule:** The AND gate output is 1 only when all the inputs are 1.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

### 3.1.3 OR Gate

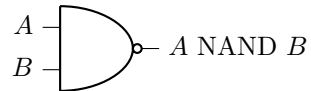




**Rule:** The OR gate output is 1 when any of the inputs is 1.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

### 3.1.4 NAND Gate



**Rule:** The NAND gate output is 0 when all the inputs are 1

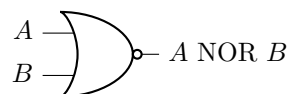
- The NAND gate is a NOT-AND gate
- The NAND gate can be viewed as a negative-OR whose output is 1 when any inputs is 0

**VHDL Code:**

```
entity NANDGate is
    port (
        a, b : in bit;
        x : out bit
    );
end NANDGate;

architecture NANDArch of NANDGate is
begin
    x <= a NAND b;
end NANDArch;
```

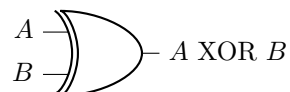
### 3.1.5 NOR Gate



**Rule:** The NOR gate output is 0 when any of the inputs is 1.

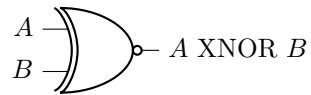
- The NOR gate is a NOT-OR gate
- The NOR gate can be viewed as a negative-AND whose output is 1 only when all the inputs are 0.

### 3.1.6 XOR Gate



**Rule:** The XOR gate output is 1 when all the inputs are not the same

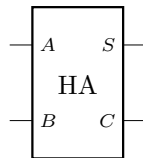
### 3.1.7 XNOR Gate



**Rule:** The XNOR gate output when the inputs are not the same

## 3.2 Adders

### 3.2.1 Half adder



A half adder is a two input, two output circuit, that takes two input bits,  $A$  and  $B$ , and outputs the two-digit sum,  $A + B$ . The MSB of the sum is referred as  $C$  (carry) and the LSB digit is  $S$  (sum). If performing addition digit by digit, the  $S$  output is the result digit and the  $C$  output is the carry resulting of the operation. This adder doesn't take into account the carry of the previous digit addition, this is why it is called half adder.

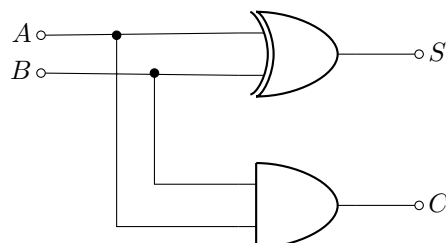
**Truth table**

$A$	$B$	$C$	$S$
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

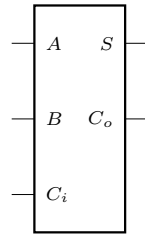
**Logic operations**

- $S = A \oplus B$
- $C = AB$

**Gate implementation**



### 3.2.2 Full adder



A full adder is a three input, two output circuit, that takes three input bits,  $A$ ,  $B$  and  $C_i$ , and outputs the two-digit sum,  $A + B + C_i$ . If performing addition digit by digit,  $A$  and  $B$  are the operands,  $C_i$  the previous carry, the  $S$  output it's the result digit and the  $C$  output is the carry resulting of the operation.

#### Truth table

$A$	$B$	$C_i$	$C_o$	$S$
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

#### Logic operations

- $S = A \oplus B \oplus C_i$
- $C_o = (A \cdot B) + (C_i \cdot (A \oplus B))$

#### Gate implementation

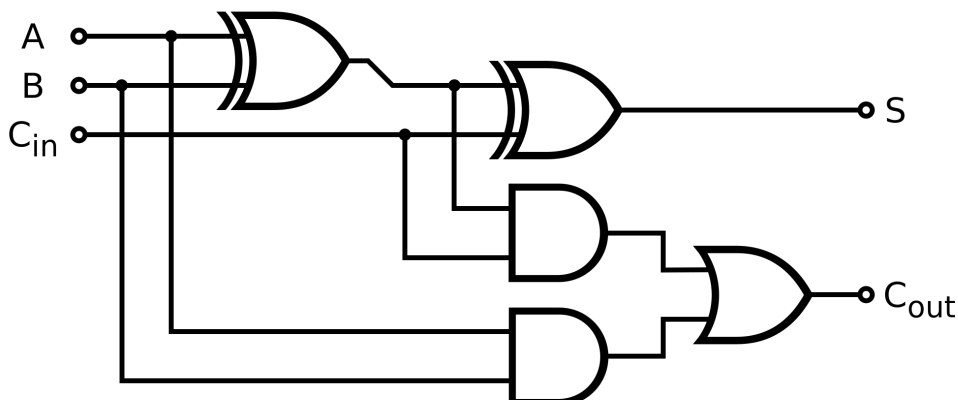
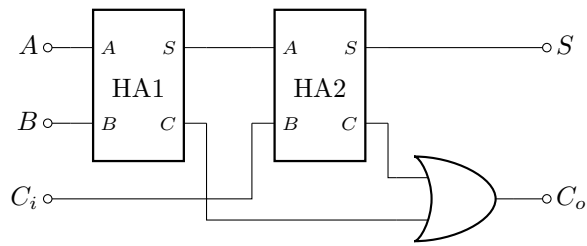


Figure 1: Source: [http://commons.wikimedia.org/wiki/File:Full\\_Adder.svg](http://commons.wikimedia.org/wiki/File:Full_Adder.svg)

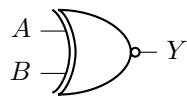
**Full adder from two half adders** A full adder can be implemented from two half adders and an OR gate:



### 3.3 Comparators

#### 3.3.1 Equality Comparators

**1-bit equality comparator** A 1 bit equality comparator can be implemented with a XNOR gate:



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

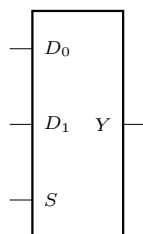
**$n$ -bit equality comparator** A  $n$ -bit comparator can be implemented by comparing two numbers digit by digit with  $n$  XNOR gates and then processing the outputs through an AND gate to check if every bit is the same in the two numbers

### 3.4 Multiplexers, Demultiplexers, Encoders, Decoders

#### 3.4.1 Multiplexer

A Multiplexer is...

**2 to 1 Multiplexer**



$$Y = (D_1 \cdot \bar{S}) + (D_0 \cdot S)$$

S	D <sub>0</sub>	D <sub>1</sub>	Y
0	0	0	0
0	0	1	0
0	1	0	1

0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

S	Y
0	$D_0$
1	$D_1$

A straightforward realization of this 2-to-1 multiplexer would need 2 AND gates, an OR gate, and a NOT gate.

**MUX implementation of truth table** A mux based circuit can be used to implement a boolean function.

A logic function with  $n$  input bits and 1 output bit can be implemented in a MUX with  $n$  selection bits and  $2^n$  data inputs, i.e. a  $2^n$ -to-1 MUX.

[\*\*\*\*\* poner ejemplo]

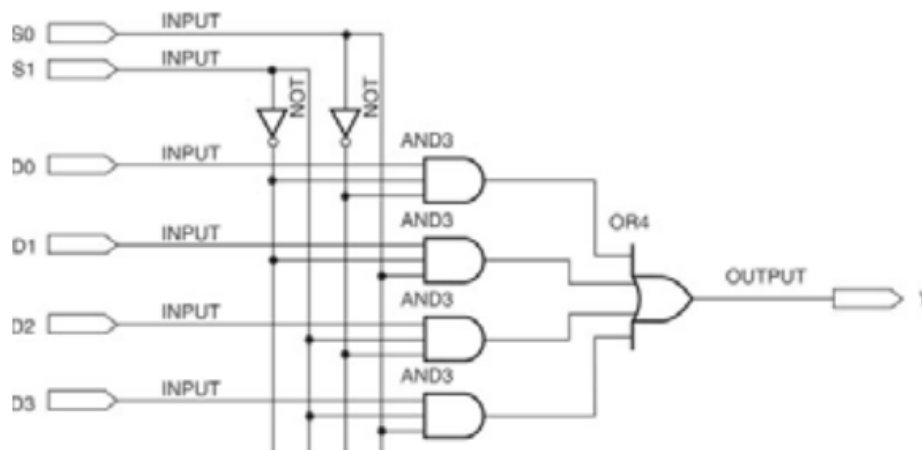


Figure 2: 4 to 1 multiplexer. Source: <http://www.ecgf.uakron.edu/grover/web/ee263/slides/Chapter%2006B.pdf>

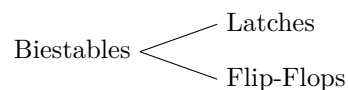
### 3.4.2 Demultiplexer

### 3.4.3 Encoder

### 3.4.4 Decoder

## 3.5 Biestables

**Biestables** are devices that have two stable states (SET and RESET); they can retain either of these states indefinitely, making them useful as storage devices.



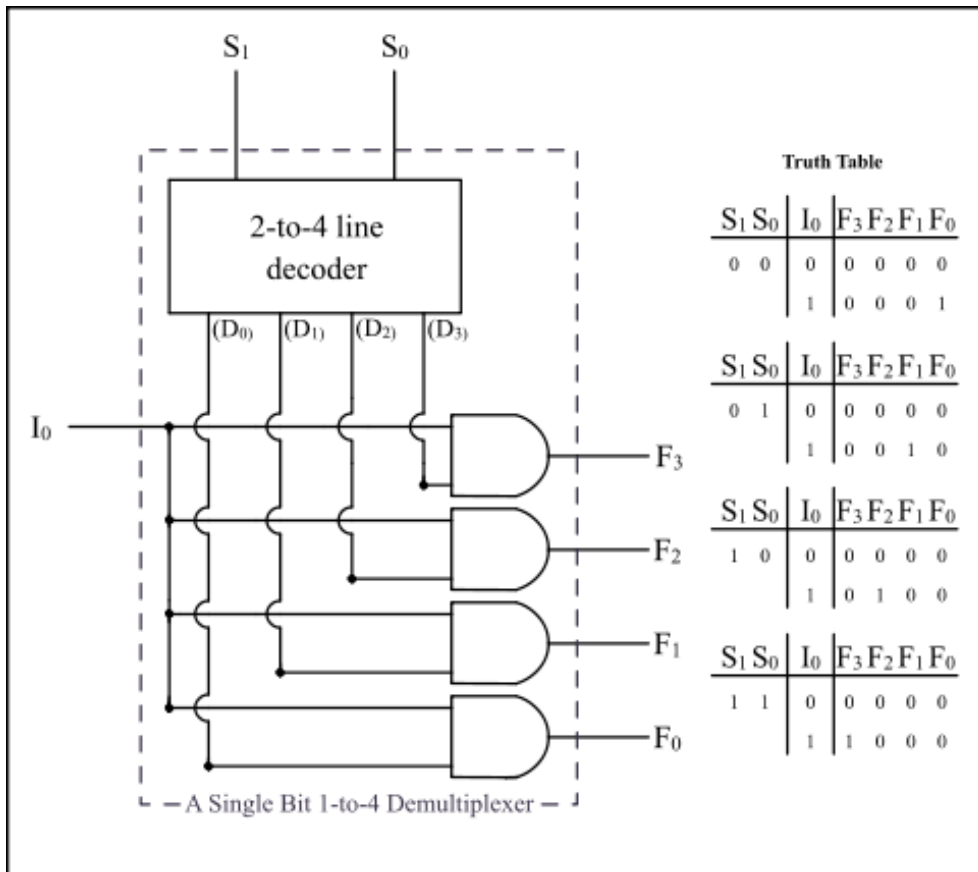
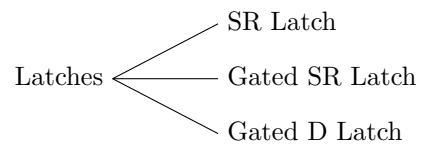


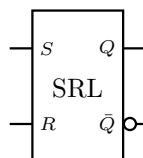
Figure 3: 4 to 1 demultiplexer. Source: [http://en.wikipedia.org/wiki/File:Demultiplexer\\_Example01.svg](http://en.wikipedia.org/wiki/File:Demultiplexer_Example01.svg)

### 3.5.1 Latches

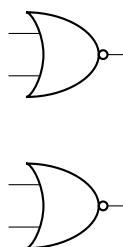
Latches are...



#### SR (SET-RESET) Latch



#### Gate implementations



#### Internal signal implementation

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity S_R_latch_top is
    Port ( S : in    STD_LOGIC;
          R : in    STD_LOGIC;
          Q : out   STD_LOGIC);
end S_R_latch_top;

architecture Behavioral of S_R_latch_top is
    signal Q2    : STD_LOGIC;
    signal notQ  : STD_LOGIC;
begin

    Q    <= Q2;
    Q2   <= R nor notQ;
    notQ <= S nor Q2;

end Behavioral;

```

### Inout ports implementation

```

entity SR_Latch is
    Port ( S,R : in STD_LOGIC;
          Q : inout STD_LOGIC;
          Q_n : inout STD_LOGIC);
end SR_Latch;

architecture SR_Latch_arch of SR_Latch is
begin
    process (S,R,Q,Q_n)
    begin
        Q <= R NOR Q_n;
        Q_n <= S NOR Q;
    end process;

end SR_Latch_arch;

```

*-- <http://vhdlbynaresh.blogspot.com.es/2013/07/design-of-sr-latch-using-behavior.html>*

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SR_Latch is
    port(
        enable : in STD_LOGIC;
        s : in STD_LOGIC;
        r : in STD_LOGIC;
        reset : in STD_LOGIC;
        q : out STD_LOGIC;
        qb : out STD_LOGIC
    );
end SR_Latch;

architecture SR_Latch_arc of SR_Latch is
begin

    latch : process (s,r,enable,reset) is
    begin
        if (reset='1') then
            q <= '0';
            qb <= '1';
        elsif (enable='1') then

```

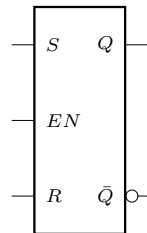
```

    if (s/=r) then
        q <= s;
        qb <= r;
    elsif (s='1' and r='1') then
        q <= 'Z';
        qb <= 'Z';
    end if;
end if;
end process latch;

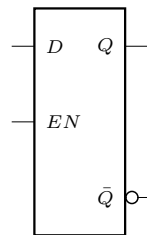
end SR_Latch_arc;

```

### Gated SR Latch

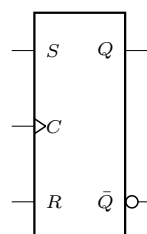
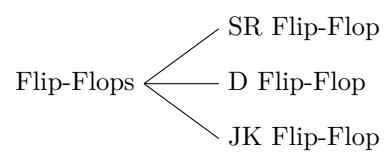


### Gated D Latch

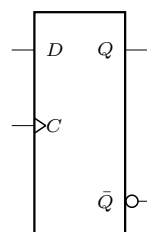


## 3.5.2 Flip-Flops

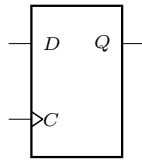
**Flip-flops** are...



### D FlipFlop







$CLK$	$D$	$Q_{next}$
$\wedge$	0	0
$\wedge$	1	1
0	X	Q
1	X	Q

## VHDL Implementations

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DFF is
    port(
        D : in std_logic; --Data input
        C : in std_logic; --Clock signal
        Q : out std_logic  --Data output
    );
end DFF;

architecture Behavioral of DFF is

begin
    process(C)
    begin
        if rising_edge(C) then
            Q <= D;
        end if;
    end process;
end Behavioral;

```

## D Flip Flop with Enable and Reset

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity DFF is
    port(
        D   : in  std_logic; --Data input
        C   : in  std_logic; --Clock signal
        EN  : in  std_logic; --Enable signal
    );
end DFF;

```

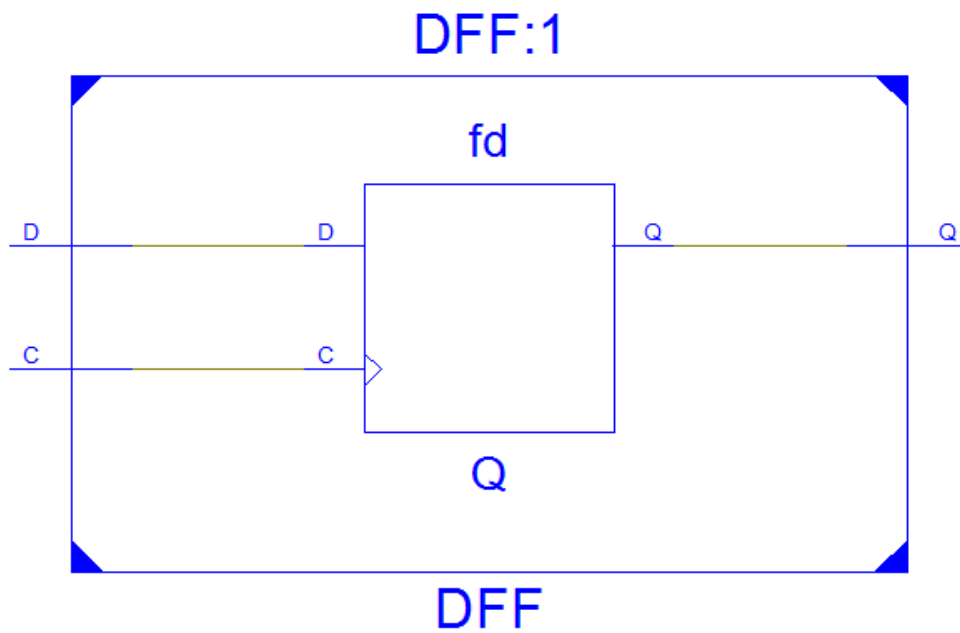


Figure 4: Synthesis Result

```

    R : in std_logic; -- Asynchronous reset
    Q  : out std_logic --Data output
);
end DFF;

architecture Behavioral of DFF is
begin
    process(C, R)
    begin
        if (R = '1') then
            Q <= '0';
        else if rising_edge(C) AND EN='1' then
            Q <= D;
        end if;
    end if;
    end process;

end Behavioral;

```

### D Flip Flop with Enable, Reset and Preset

```

-- Code from Wikibooks:
-- https://en.wikibooks.org/wiki/VHDL_for_FPGA_Design/D_Flip_Flop

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DFF is
    port
    (
        clk : in std_logic;

        rst : in std_logic;
        pre : in std_logic;
        ce  : in std_logic;

```

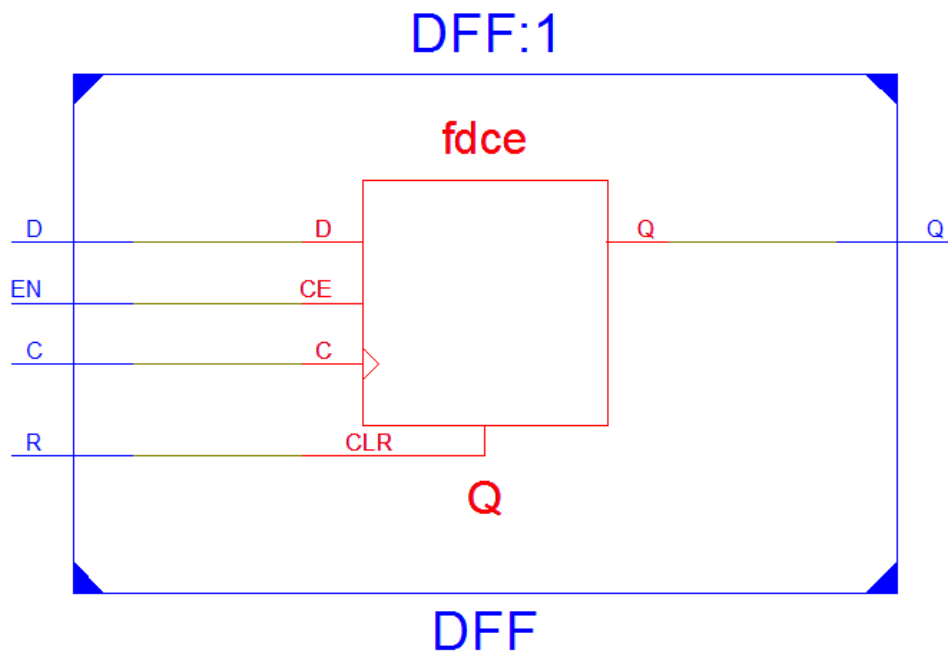


Figure 5: Synthesis Result

```

    d : in std_logic;
    q : out std_logic
);
end entity DFF;

architecture Behavioral of DFF is
begin
    process (clk) is
    begin
        if rising_edge(clk) then
            if (rst='1') then
                q <= '0';
            elsif (pre='1') then
                q <= '1';
            elsif (ce='1') then
                q <= d;
            end if;
        end if;
    end process;
end architecture Behavioral;

```

### With Guarded Block

*-- D Flip Flop with Guarded Block, from Circuit Design with VHDL*

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff IS
    PORT (
        d, clk, rst: IN STD_LOGIC;
        q : out std_logic
    );
END dff;

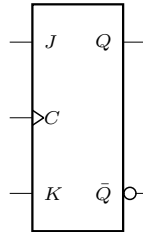
ARCHITECTURE dff OF dff IS
BEGIN
    b1: BLOCK (clk'EVENT AND clk='1')

```

```

BEGIN
    q <= GUARDED '0' WHEN rst='1' ELSE d;
END BLOCK b1;
END dff;

```



## 3.6 Finite State Machines

Finite State Machines (FSM) are...

- Moore machine: The FSM uses only entry actions, i.e., output depends only on the state.
- Mealy machine: The FSM uses only input actions, i.e., output depends on input and state.

### VHDL Template for FSMs

```

-- FSM template: from Circuit Design with VHDL

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-----
ENTITY <entity_name> IS
    PORT ( input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <entity_name>;
-----

ARCHITECTURE <arch_name> OF <entity_name> IS
    TYPE state IS (state0, state1, state2, state3, ...);
    SIGNAL pr_state, nx_state: state;
BEGIN
    ----- Lower section: -----
    PROCESS (reset, clock)
    BEGIN
        IF (reset='1') THEN
            pr_state <= state0;
        ELSIF (clock'EVENT AND clock='1') THEN
            pr_state <= nx_state;
        END IF;
    END PROCESS;
    ----- Upper section: -----
    PROCESS (input, pr_state)
    BEGIN
        CASE pr_state IS
            WHEN state0 =>
                IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state1;
                ELSE ...
            END IF;
            WHEN state1 =>
                IF (input = ...) THEN

```

```

        output <= <value>;
        nx_state <= state2;
ELSE ...
    END IF;
    WHEN state2 =>
        IF (input = ...) THEN
            output <= <value>;
            nx_state <= state3;
ELSE ...
END IF; ...
    END CASE;
    END PROCESS;
END <arch_name>;

```

### 3.7 Lookup tables

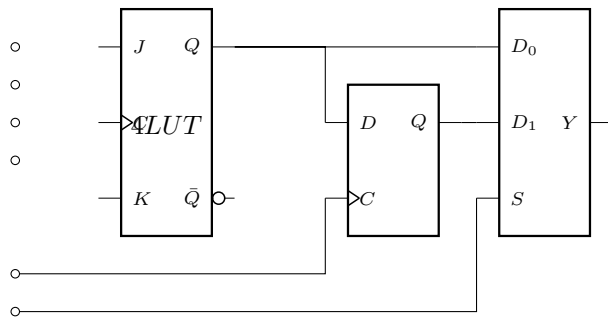
A Lookup Table (LUT) can encode a  $n$ -bit boolean function, with 1 bit output.

A  $n$ -bit LUT can be implemented with a mux whose select lines are the  $n$  inputs of the LUT and whose  $2^n$  inputs are constants.

#### 2-bit LUT

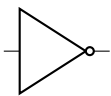
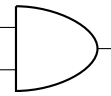
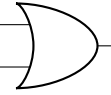
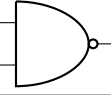
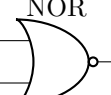
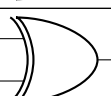
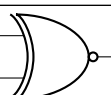
$X_0$	$X_1$	$Y$
0	0	$y_0$
0	1	$y_1$
1	0	$y_2$
1	1	$y_3$

**FPGA logic cell** FPGAs are built of many interconnected logic cells. A basic logic cell could be composed of a 4 LUT, a flipflop and a 2to1 mux to bypass the register.

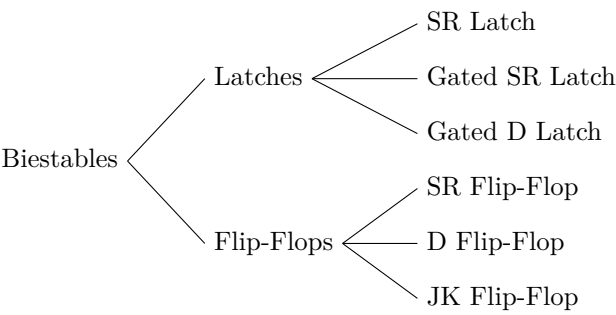


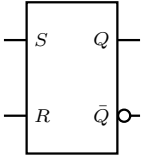
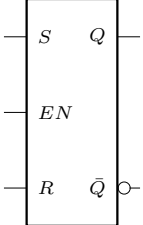
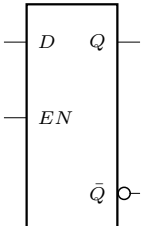
4    Reference tables

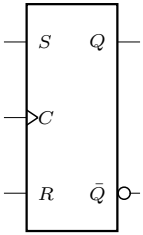
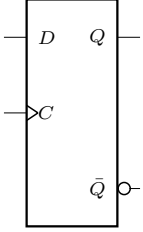
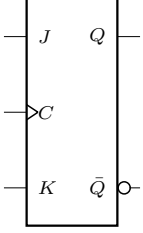
## 4.1 Logic Gates

Gate	Symbol	Rule	Truth table
NOT		The NOT gate output is the complementary of the input	not
AND		The AND gate output is 1 only when all the inputs are 1	and
OR		The OR gate output is 1 when any of the inputs is 1	or
NAND		The NAND gate output is 0 when all the inputs are 1	or
NOR		The NOR gate output is 0 when any of the inputs is 1	or
XOR		The XOR gate output is 1 when all the inputs are not the same	or
XNOR		The XNOR gate output when the inputs are not the same	or

4.2 Biestables



Latch	asd
	q
	q
	q

FlipFlop	asd
	q
	q
	q



## 5 Bibliography

- Digital Fundamentals, Thomas L. Floyd, 10th ed. Pearson
- [https://en.wikipedia.org/wiki/Binary\\_number](https://en.wikipedia.org/wiki/Binary_number)
- [https://en.wikipedia.org/wiki/Ones'\\_complement](https://en.wikipedia.org/wiki/Ones'_complement)

## 6 More