

Hierarchical Design

Divide and Conquer:

It is not practical to write a truth table when the number of inputs is large.

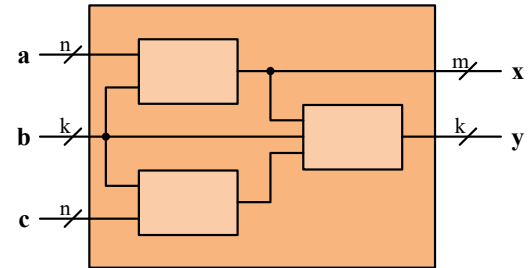
In practice a designer must **subdivide the system** into smaller sub-systems. Each sub-system will have a simpler function and also will have less inputs and outputs.

When we specify the internal structure of a system we have provided a concrete definition of the system. If the structure is defined in terms of sub-systems, then the definition is only concrete once the **internal structures of all the sub-systems** have been clearly defined.

Entity **SubCircuit** is

```
port( a, b, c : in std_logic;
      x, y, z : out std_logic );
```

End Entity **SubCircuit**;



Architecture structural of **MainCircuit** is

```
signal nodeA, nodeB, nodeC, nodeX, nodeY, nodeZ : std_logic;
begin
```

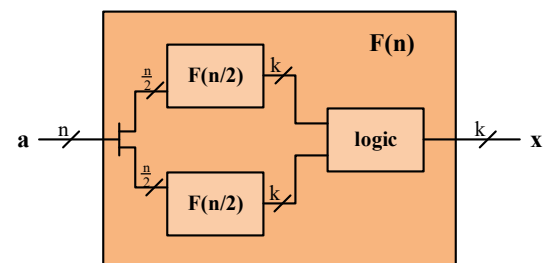
```
il: ENTITY work.SubCircuit
```

```
End Architecture structural;
```

Recursive Definitions:

Suppose the system that we are defining (designing) has **some parameter** that describes the size of the system, **called N**. Furthermore, suppose that the system performs some **function, F**.

In some cases we are able to define the system function $F(N)$ in terms of smaller sub-systems that perform the **same function**. ie. **large additions** can be decomposed into a collection of **smaller additions**.



Breaking the Recursion:

It is clear that to define a function F in terms of the function F is **circular reasoning**.

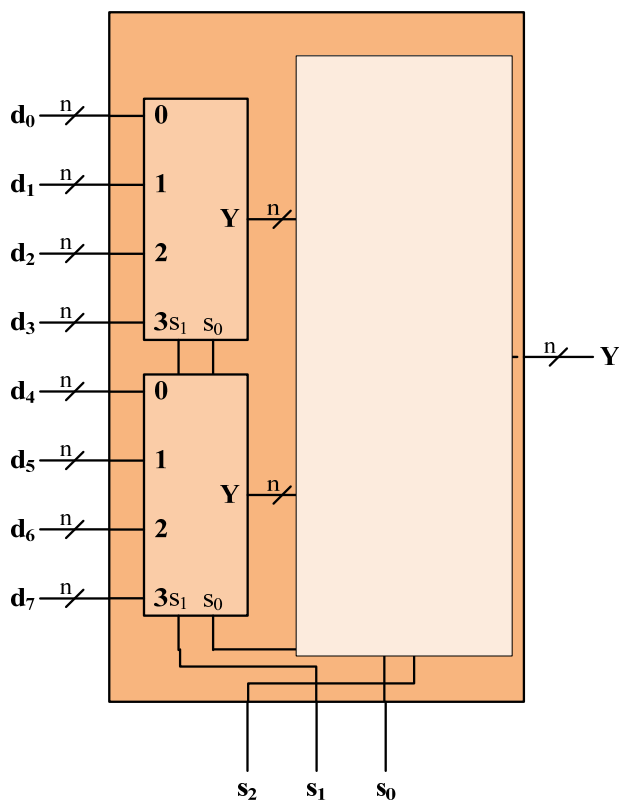
This is not as bad as it seems because we were careful to define $F(n)$ in terms of $F(k)$ where $k < n$.

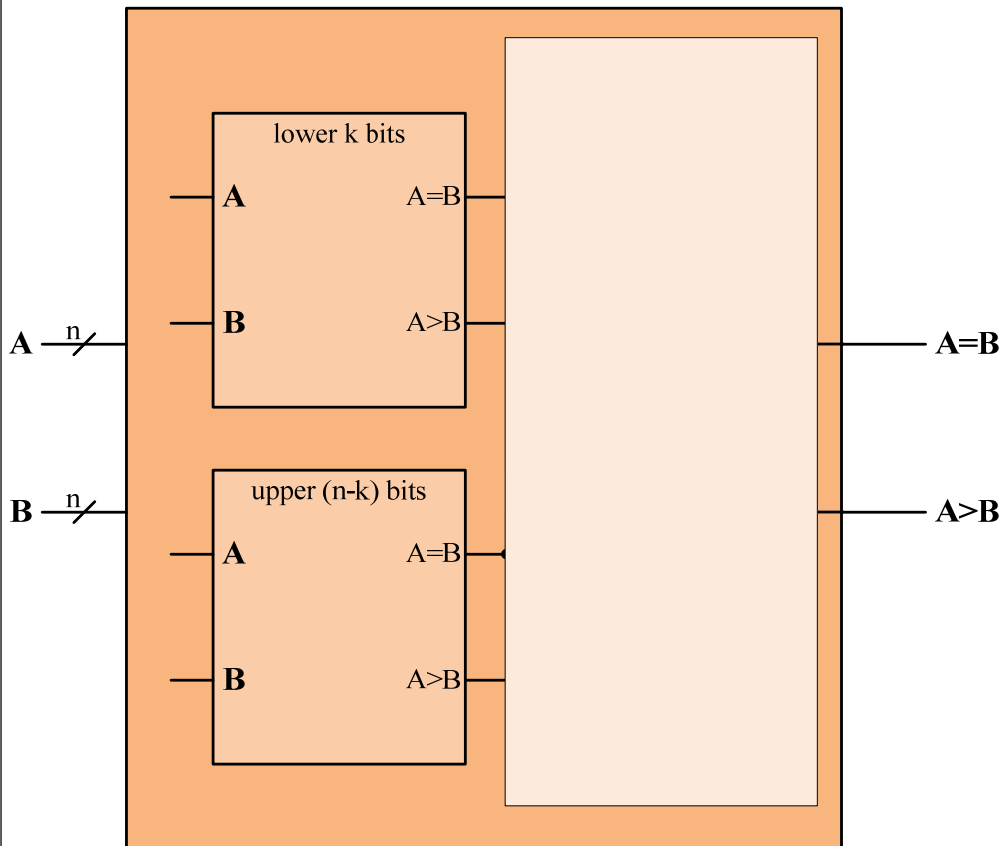
We avoid the circular definition by providing **TWO descriptions**.

- The first is a recursive description (**structural**) that is valid for sufficiently large n .
- The second is a specific, non-recursive description (**behavioural**) for some fixed small n .

Example:

If n is sufficiently small, we can define the function, $F(n)$, using a truth table.

Example - MUX:

Example – Magnitude Comparator:Calculating Circuit Cost:

Suppose that $n = 2^p$. Furthermore, suppose we define the magnitude comparator by sub-dividing into two equal halves. (This is possible because n is a power of 2) We will stop the recursion when $n = 1$. The truth table will be simple.

$$A=B \Leftrightarrow A \cdot B + \bar{A} \cdot \bar{B}$$

$$A>B \Leftrightarrow A \cdot \bar{B}$$

Let C_k denote the cost of a circuit when $n = 2^k$.

The stopping condition gives us, $C_0 = 12$ (using AND/OR instead of XOR)

We can determine the general cost inductively by considering,

$$C_1 = 2 (C_0) + 9$$

$$C_2 = 2 (C_1) + 9 = 2^2(C_0) + 2(9) + 9 = 2^2(C_0) + 3(9)$$

$$C_3 =$$

$$C_4 =$$

$$C_p =$$

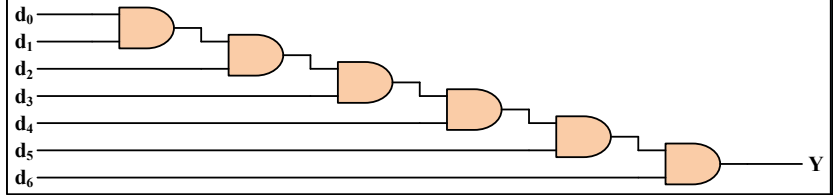
Hierarchical Design

Chain of AND gates:

Entity And2 is

```
port (a, b : in std_logic;
      y : out std_logic );
```

End entity And2;



Entity Chain is

```
port ( d : in std_logic_vector( 6 downto 0 );
      Y : out std_logic );
```

end Entity Chain;

Architecture simple of Chain is

```
signal iq : std_logic_vector( 6 downto 0 );
```

begin

```
iq(0) <= d(0);
```

```
Y <= iq(6);
```

```
A0: entity work.And2 port map ( iq(0), d(1), iq(1) );
```

```
A1: entity work.And2 port map ( iq(1), d(2), iq(2) );
```

```
A2: entity work.And2 port map ( iq(2), d(3), iq(3) );
```

```
A3: entity work.And2 port map ( iq(3), d(4), iq(4) );
```

```
A4: entity work.And2 port map ( iq(4), d(5), iq(5) );
```

```
A5: entity work.And2 port map ( iq(5), d(6), iq(6) );
```

End Architecture simple;

VHDL- Generating Repetitive Statements:

Architecture simple of Chain is

```
signal iq : std_logic_vector( 6 downto 0 );
```

begin

```
iq(0) <= d(0);
```

```
Y <= iq(6);
```


gg:

Az:

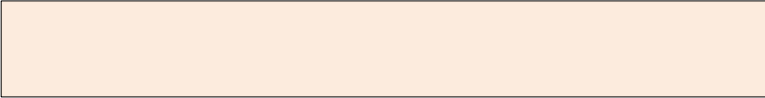
End Architecture simple;

VHDL – Generic Parameters - Example – Ripple Adder:

```
Entity Adder is
generic( N : integer := 4 );
port( x, y : in std_logic_vector( N-1 downto 0 );
      Cin : in std_logic;
      s : out std_logic_vector( N downto 0 ) );
End Entity Adder;
```

```
Architecture Ripple1 of Adder is
    signal c : std_logic_vector( N downto 0 );
    signal g, p : std_logic_vector( N-1 downto 0 );
begin
    columns: for i in 0 to N-1 generate
        
    end generate columns;
    c(0) <= Cin;
    S(N) <= c(N);
End Architecture Ripple1;
```

Example – Ripple Adder without Generate:

```
Architecture Ripple2 of Adder is
    signal c: std_logic_vector( N downto 0 );
    signal g, p : std_logic_vector( N-1 downto 0 );
begin
    g <= x and y;
    p <= x xor y;
    
    S(N) <= c(N);
End Architecture Ripple2;
```

VHDL - Instantiating different Architectures:

```

Entity TestAdder is
End Entity TestAdder;

Architecture Stuff of TestAdder is
    signal x, y : std_logic_vector( 7 downto 0 );
    signal cin  : std_logic;
    signal s1, s2 : std_logic_vector( 8 downto 0 );
Begin
    i1: Entity work.Adder(Ripple1) generic map ( 3 )
        port map ( x(2 downto 0), y(2 downto 0), cin, s1(3 downto 0) );

    i2: Entity work.Adder(Ripple2) generic map ( 5 )
        port map ( x(4 downto 0), y(4 downto 0), cin, s2(5 downto 0) );
End Architecture Stuff;

```

Parity Generation/Checking:

- An ODD Parity Generator is a circuit that measures a group of N-bits at it's input and produces a single-bit output.
- The single-bit output is combined with the N-bits to produce a group of N+1 bits.
- The output bit is chosen to be either '1' or '0' so that the

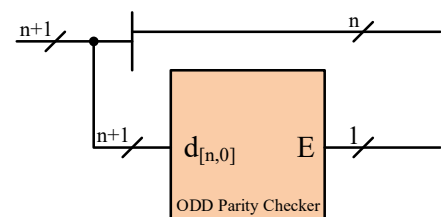
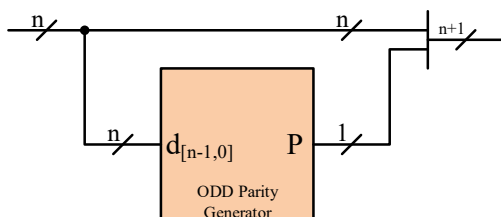
An ODD Parity Checker is a circuit that measures a group of bits at it's input and produces a single-bit output.

- The single-bit output is considered to be
- The output bit is chosen to '1' if there is

The error output will be true (= '1') if the input does NOT have ODD parity.

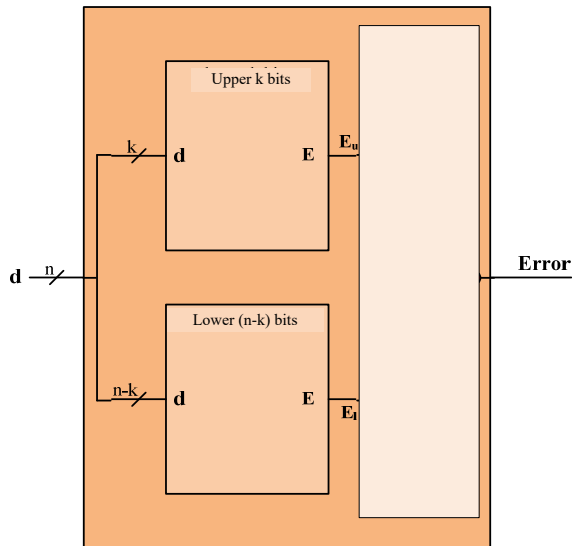
(Notice that this is the same circuit as an ODD parity generator)

The output bit is chosen to '0' if there is NO error. This will be true if the input has ODD parity.



Hierarchical Design

Example – Designing an ODD parity Checker:



E_u	E_l	upper parity	lower parity	total parity	E
0	0				
0	1				
1	0				
1	1				

We can choose either $n=2$ or $n=1$ as the stopping condition.

for $n=1$

for $n=2$

Splitting the wires:

When we split the wires into two parts there will be a larger part and a smaller part when N is ODD.

Suppose that **we choose** to split the wires so that the **larger part** contains the **upper wires**.

- The **size** of the smaller part will always be $N/2$
- The **size** of the larger part will always be $(N+1)/2$

N	size of upper	size of lower	upper	lower	$N/2$
8	4	4	[7,4]	[3,0]	4
7					
6					
5					
4					
3					
2	1	1	[1,1]	[0,0]	1

The indices of the upper wires will always range
The indices of the lower wires will always range

Hierarchical Design

VHDL – Entity / Architecture:

Entity ParityCheck is

```
port ( d : in std_logic_vector( N-1 downto 0 );  
      E : out std_logic );  
End Entity ParityCheck;
```

Architecture Tree of ParityCheck is

```
signal Eup, Elo : std_logic;  
Begin
```

Recur:

StopRecur:

```
End Architecture Tree;
```

Additional Notes: