



Lesson 1:

Hardware Modeling and Design with VHDL.

Introduction to FPGAs

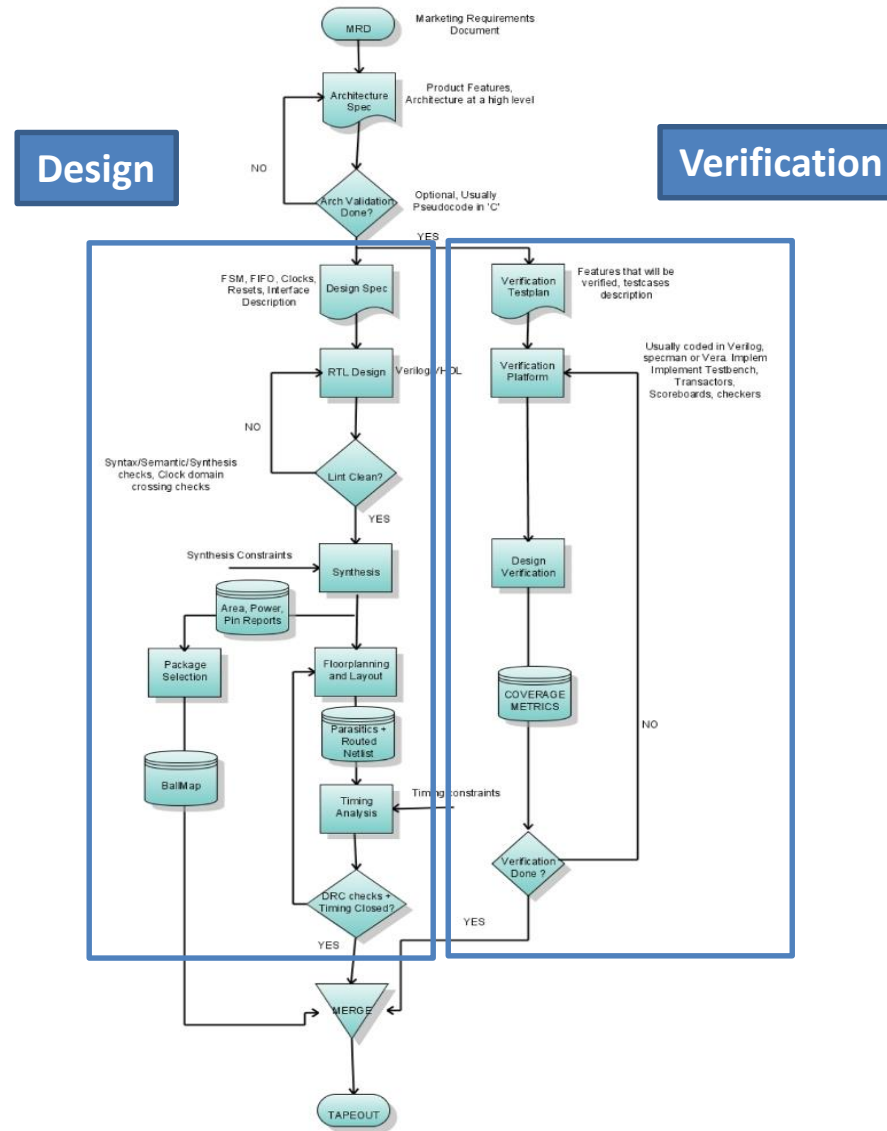


Outline

1. Design flow
2. Hardware description language
3. Simulation with VHDL
4. Structure of a VHDL code
5. VHDL basic elements
6. Finite state machines
7. Structural definitions in VHDL
8. Simulation testbenches
9. Introduction to FPGAs



Design flow





Outline

1. Design flow
2. Hardware description language
3. Simulation with VHDL
4. Structure of a VHDL code
5. VHDL basic elements
6. Finite state machines
7. Structural definitions in VHDL
8. Simulation testbenches
9. Introduction to FPGAs

Hardware Description Language



- ¿What is HDL?
 - It is a language especially created for hardware design.
 - The structure and syntax of the language suggest hardware design.
- ¿Why to use HDL?
 - To discover problems and faults in the design before actually implementing it in hardware.
 - The complexity of an electronic system grows exponentially. Thus, it is very convenient to build **a prototype of the circuit** previously to its manufacturing process.
 - It makes easy for a team of developers to work together.

Hardware description languages and software languages are completely different

Hardware Description Language



- VHDL (VHSIC Hardware Description Language)
 - VHSIC (Very High Speed Integrated Circuit): US government, 1980.
 - IEEE VHDL'87.
 - www.vhdl.org
- Verilog
 - Developed by CADENCE.
 - IEEE 1364.
 - www.eda.org
- SystemVerilog (superset of Verilog-2005)
 - Verification and hardware description.
 - IEEE 1364 extension.
 - www.systemverilog.org

Hardware Description Language



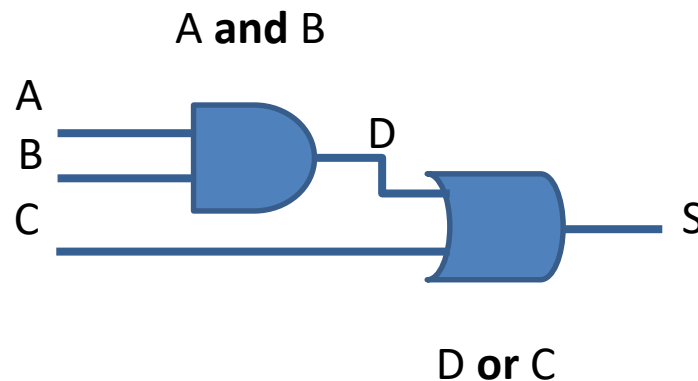
■ VHDL

- Hardware description of the circuit:
 - Behavioral: describes the functionality of the system (more like an algorithm).
 - Structural: describes the implementation specifying the HW components (registers, gates, counters) and their inter-connexions.
- It allows simulating the behavior of a circuit by means of a description language that has many similarities with software description languages before its implementation:
 - Testing and comparing alternatives without needing to build HW prototypes.

Hardware Description Language



- HDL has to be able to simulate the behavior of the HW so the programmer does not have to impose any restriction.



Hardware Description Language



t = 5ns	t = 10ns
A = 0	A = 1
B = 1	B = 1
C = 0	C = 0

Description 1

D = A and B;

S = D or C;

Description 2

S = D or C;

D = A and B;

Will we get the same result?



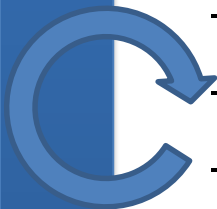
Outline

1. Design flow
2. Hardware description language
3. Simulation with VHDL
4. Structure of a VHDL code
5. VHDL basic elements
6. Finite state machines
7. Structural definitions in VHDL
8. Simulation testbenches
9. Introduction to FPGAs

Simulation with VHDL



- Typically, tools follow a **discrete event time model** for simulating circuits described in VHDL:
 - Events occur each time any signal changes its value.
 - This marks a potential change of state in the circuit. Between consecutive events, no change in the system is assumed to occur.
 - The simulation can directly jump in time from one event to the next one, independently of the time elapsed between them.
- VHDL simulations comprise three steps:
 - Step 0: All the signals are initialized and the time count is set to 0.
 - Step 1: All the transitions scheduled for that time are carried out.
 - Step 2: Signals that are modified as a consequence of transitions occurring at instant = t are written down in the list of events and scheduled for instant = $t + \delta$, where δ is infinitesimal.





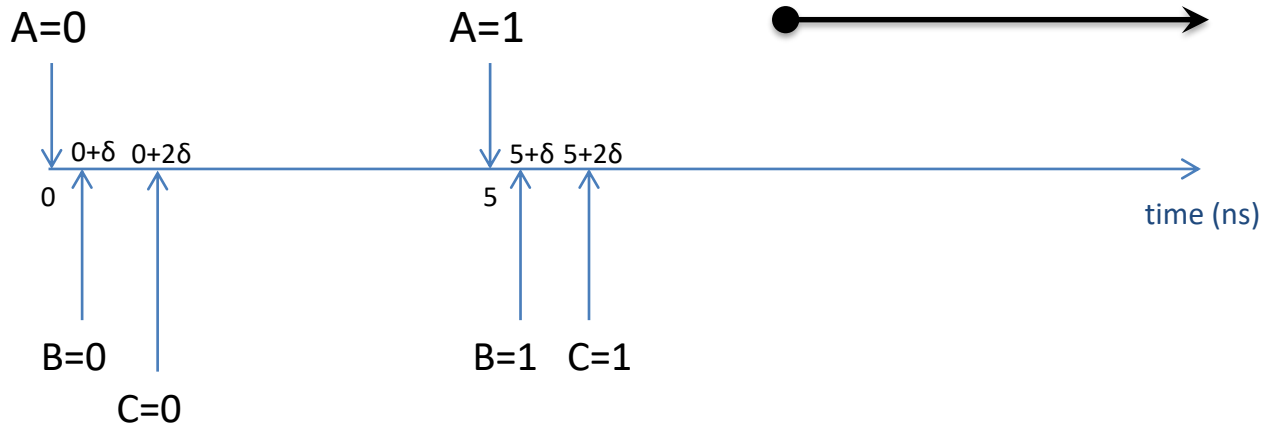
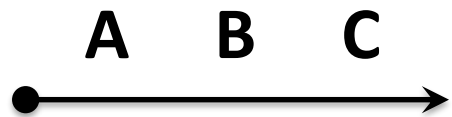
Simulation with VHDL

■ Example 1 ➡

```
C <= B;  
B <= A;
```

Initial values
A=U
B=U
C=U

Equivalent Circuit



t(ns)	0	0 + δ	0 + 2δ	No more changes	5	5 + δ	5 + 2δ	No more changes
A	U	0	0		1	1	1	
B	U	0	0		0	1	1	
C	U	U	0		0	0	1	

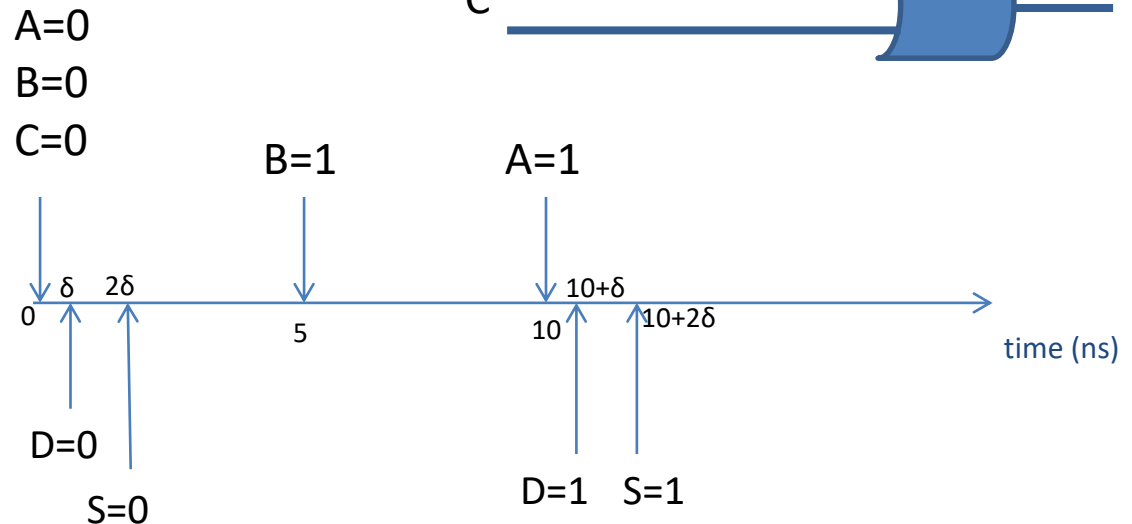
Simulation with VHDL



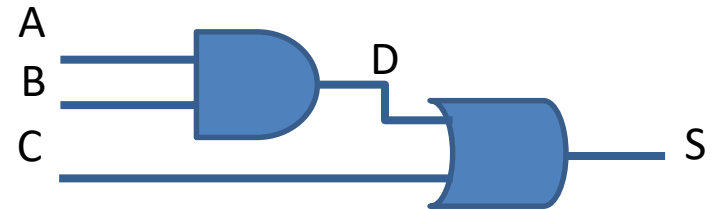
■ Example 2 \rightarrow $D \leq A \text{ and } B;$
 $S \leq D \text{ or } C;$

Initial values

A=U
 B=U
 C=U
 D=U
 S=U



Equivalent Circuit





Simulation with VHDL

■ Example 3 \rightarrow $S \leq D \text{ or } C;$
 $D \leq A \text{ and } B;$

Equivalent Circuit

Initial values

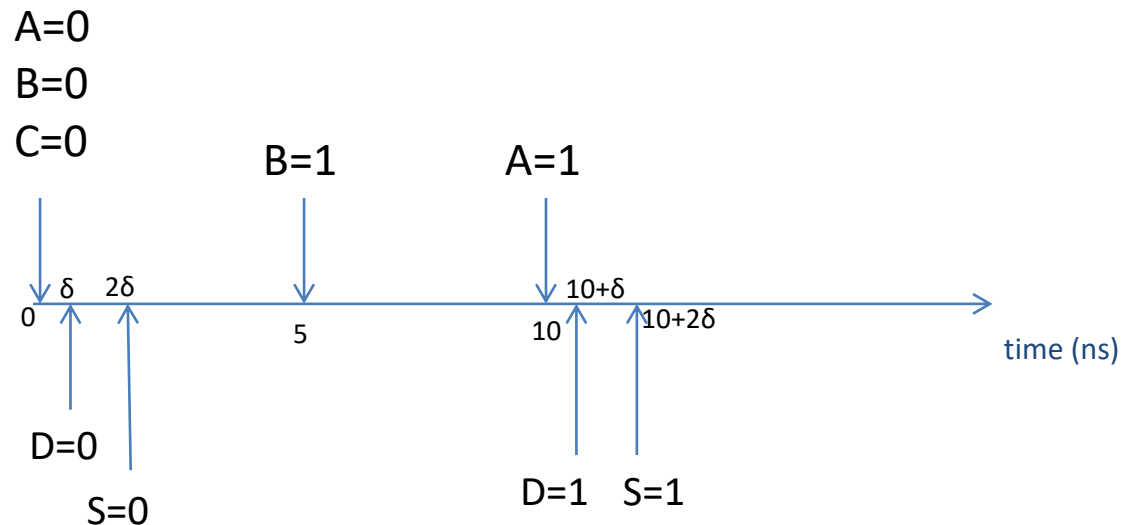
A=U

B=U

C=U

D=U

S=U

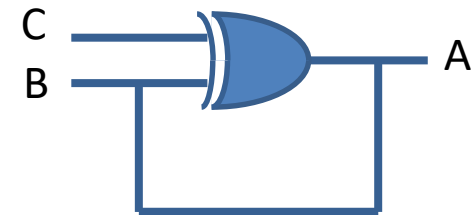




Simulation with VHDL

■ Example 4 \rightarrow $A \leq C \text{ xor } B;$
 $B \leq A;$

Equivalent Circuit

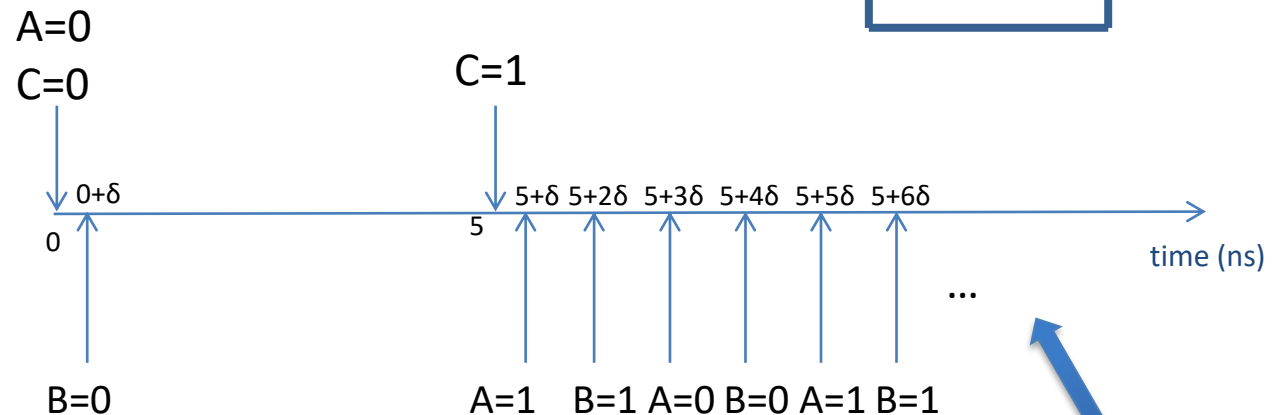


Initial values

$A=U$

$B=U$

$C=U$



It never converges...



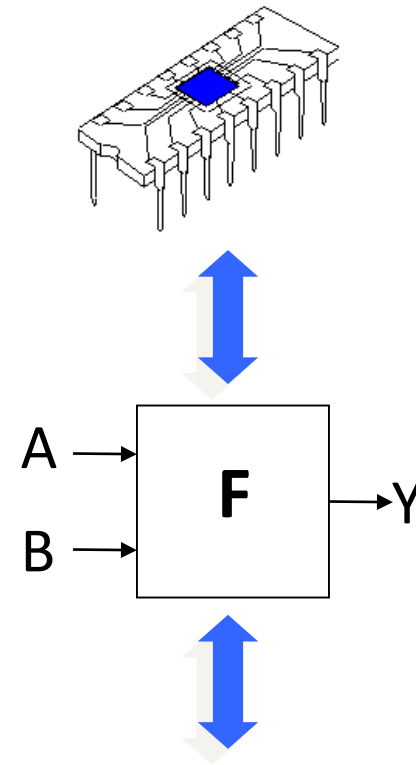
Outline

1. Design flow
2. Hardware description language
3. Simulation with VHDL
4. Structure of a VHDL code
5. VHDL basic elements
6. Finite state machines
7. Structural definitions in VHDL
8. Simulation testbenches
9. Introduction to FPGAs



Structure of a VHDL code

- Description of a digital circuit:
 - A digital circuit is described in VHDL by specifying its inputs and outputs, as well as how these outputs are obtained from the inputs



```
entity F is
    port( A: in  std_logic;
          B: in  std_logic;
          Y: out std_logic );
end F;
```



Structure of a VHDL code

- VHDL codes comprise 2 parts:

```
entity interface_name is  
    generic (parameters list);  
    port (input/output ports list);  
end interface_name;
```

```
architecture circuit_implementation of  
interface_name is  
    -- Declarations  
begin  
    -- Code  
end architecture circuit_implementation;
```



Structure of a VHDL code

- VHDL codes comprise 2 parts:
 - Entity
 - Defines the circuit or subcircuit externally.
 - Name and number of I/O ports, types of I/O data.
 - This includes all the information that is needed to connect that circuit to other modules.
 - Architecture
 - Defines the circuit or subcircuit internally.
 - Internal signals, functions, procedures, constants, ...



Outline

1. Design flow
2. Hardware description language
3. Simulation with VHDL
4. Structure of a VHDL code
5. VHDL basic elements
6. Finite state machines
7. Structural definitions in VHDL
8. Simulation testbenches
9. Introduction to FPGAs



VHDL basic elements

```
entity name is  
    port (list of Input/Output ports);  
end name;
```

list of Input-Output ports

name_port: type_of_port type_of_signal;

inputA: in std_logic_vector (7 downto 0);

std_logic and **std_logic_vector** require the following library:
library IEEE;
use IEEE.std_logic_1164.all;



VHDL basic elements

■ Data types

- Integer and real types:

INTEGER (range)

NATURAL (range)

REAL (range)

- Physical types:

TIME

RESISTENCE

range:

n_min **TO** n_max

n_max **DOWNTO** n_min



VHDL basic elements

– Enumerated types (set of possible values):

BOOLEAN : {TRUE, FALSE}

BIT/BIT_VECTOR (range) : {'0', '1'}

CHARACTER/STRING (range) : {ASCII characters}

STD_LOGIC/STD_LOGIC_VECTOR (range) :

{'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'}

'U': uninitialized. This signal hasn't been set yet.

'X': unknown. Impossible to determine this value/result.

'0': logic 0

'1': logic 1

'Z': High Impedance

'W': Weak signal, can't tell if it should be 0 or 1.

'L': Weak signal that should probably go to 0

'H': Weak signal that should probably go to 1

'-': Don't care



VHDL basic elements

■ 8:1 multiplexer:

```
entity multiplexer is
    port (I: in std_logic_vector(7 downto 0);
          S: in std_logic_vector(2 downto 0);
          O: out std_logic);
end multiplexer;
```




VHDL basic elements

```
architecture arch_name of entity_name is  
    -- Declarations in the architecture:  
    -- Signals  
    -- Types  
    -- Components  
begin  
    -- Code:  
    -- Concurrent statements  
    -- Components (interconnexions)  
    label: process (sensitivity list)  
        -- Variables  
    begin  
        -- Code  
    end process;  
end arch_name;
```

The sensitivity list is optional
(processes are discussed later)



VHDL basic elements

- An `architecture` defines how the circuit operates.
- Each `architecture` is associated to an entity. This is indicated in the first sentence.
 - `architecture circuit_arch of circuit_entity is`
- The following objects are defined before the `begin` of the architecture:
 - Signals
 - Types
 - Components: other circuits already defined and compiled, whose interface in VHDL is well known (i.e., their entity).
- The `begin` and the `end` reserved words mark the boundaries of the VHDL code that will actually describe the operation of the circuit.



VHDL basic elements: signals

- They represent memory elements or connections. They are used to interconnect the concurrent statements in the code.
- They can be used anywhere in the architecture code. The ports of an entity are signals.
- The operator for assigning signals is: `<=` `signal <= value`
- They follow the timing model based in deltas.
- Signals can be used at the right side of other signals or variable assignments.
- Reserved word `signal`
 - It is not advisable to initialize their values



VHDL basic elements: variables

- They do not represent memory elements or connections.
- They can be used as indexes, for loop instructions or to model components
- They can only exist in processes
- The operator for assigning signals is := `variable := value`
- They cannot be shared among processes
 - There exist so-called “shared” variables, but we will not use them in this course
- They do not follow the timing model based on deltas
- Signals can be used at the right side of other signals or variable assignments
- Reserved word `variable`
 - It is recommendable to initialize them.
- In this course, we will ONLY use them in the testbenches



VHDL basic elements

```
entity F is  
    port (A, B: in std_logic; Y out std_logic);  
end F;
```

```
architecture circuit of F is  
    signal D, E: std_logic_vector(1 downto 0);  
    signal H: std_logic;
```

} ← signals

```
begin
```

```
(...)
```

```
end architecture circuit;
```



VHDL basic elements

- Customized types can also be defined.
 - New enumerated and range-based types:

```
type new_type1 is (v1, v2, v3, v4);  
type new_type2 is range 0 to 17;  
signal A: new_type1;  
signal B, C: new_type2;
```

```
--assignments to signals, inside the architecture  
A <= v1;  
B <= 17;    --CORRECT assignment  
C <= 18;    --INCORRECT assignment
```



VHDL basic elements

■ Records

- Like C or C++ “structures”

```
type my_record is
    field_1: std_logic_vector (3 downto 0);
    field_2: std_logic;
end record;
signal A, C: my_record;
signal B: std_logic_vector (3 downto 0);
```

--assignments to “A”, inside the
architecture

```
A.field_1 <= B;
A.field_2 <= B(2);
C <= A;
```

You can either assign field by field, or make a direct assignment between 2 records.



VHDL basic elements

■ Arrays:

```
type my_array is array(3 downto 0)
  of std_logic_vector(3 downto 0);
```

```
signal A: my_array;
signal B: std_logic_vector(7 downto 0);
```

```
--assignments to "A", inside the architecture
A(1) <= B(3 downto 0);
A(2) <= B(7 downto 4);
```

■ Constants can also be declared:

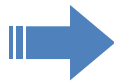
```
constant M: integer := 32;
signal    A: std_logic_vector (M-1 downto 0);
```




VHDL basic elements

■ Basic operators:

abs
*, /, mod, rem
+, -
&
and, or, nand,
nor ...



```
y,d: std_logic_vector(1 downto 0);  
x1, x2: std_logic  
...  
y(0) <= (x1 and x2) or d(0);  
y(1) <= x1 and not x2;  
...  
y <= x1 & x2;
```



```
architecture arch_name of entity_name is
  -- Declarations in the architecture:
    -- Signals
    -- Types
    -- Components
begin
  -- Code:
    -- Concurrent statements
    -- Components (interconnexions)
    label: process (sensitivity list)
      -- Variables
      begin
        -- Code
      end process;
end arch_name;
```

VHDL basic elements



- Concurrent statements:
 - ALWAYS outside of the `process`.
 - They may appear anywhere in the code (between the `begin` and the `end` of the `architecture`).
 - They are purely combinatorial logic.
 - They always end with `else` or `when others`.



VHDL basic elements

■ Concurrent statements – **WITH-SELECT-WHEN:**

```
with identifier select  
  signal_name <= value_1 when value_identifier1,  
                  value_2 when value_identifier2,  
                  ...  
                  value_i when value_identifieri,  
                  other_value when others;
```



MANDATORY

```
with input select  
  output <= "00" when "0001",  
            "01" when "0010",  
            "10" when "0100",  
            "11" when others;
```



VHDL basic elements

- `architecture` of an 8:1 multiplexer with a **WITH-SELECT-WHEN** statement

```
entity MUX_8_1 is
    port (I: in std_logic_vector(7 downto 0);
          S: in std_logic_vector(2 downto 0);
          O: out std_logic);
architecture B of MUX_8_1 is
begin
    with S select
        O <= I(0) when "000",
              I(1) when "001",
              I(2) when "010",
              I(3) when "011",
              I(4) when "100",
              I(5) when "101",
              I(6) when "110",
              I(7) when others;
end architecture;
```



VHDL basic elements

■ Concurrent statements – **WHEN-ELSE:**

```
signal_name <= value_1 when condition_1 else  
                value_2 when condition_2 else  
                ...  
                value_i when condition_i else  
                other_value;
```



MANDATORY

Highest prio.

```
Output <= "00" when input= "0001" else  
          "01" when input= "0010" else  
          "10" when input= "0100" else  
          "11";
```



```
architecture arch_name of entity_name is
    -- Declarations in the architecture:
    -- Signals
    -- Types
    -- Components
begin
    -- Code:
    -- Concurrent statements
    -- Components (interconnexions)
    label: process (sensitivity list)
        -- Variables
        begin
            -- Code
        end process;
end arch_name;
```



VHDL basic elements

■ Processes: powerful concurrent statements

```
process (sensitivity list)
    -- Variables
begin
    -- Sequential statements
    -- Conditional statements
end process
```

- The statements existing inside a process are executed only if any of the signals in the sensitivity list has changed its value.
- Sequential code (kind of...).
- Signal updates are scheduled for $t + \delta$



VHDL basic elements

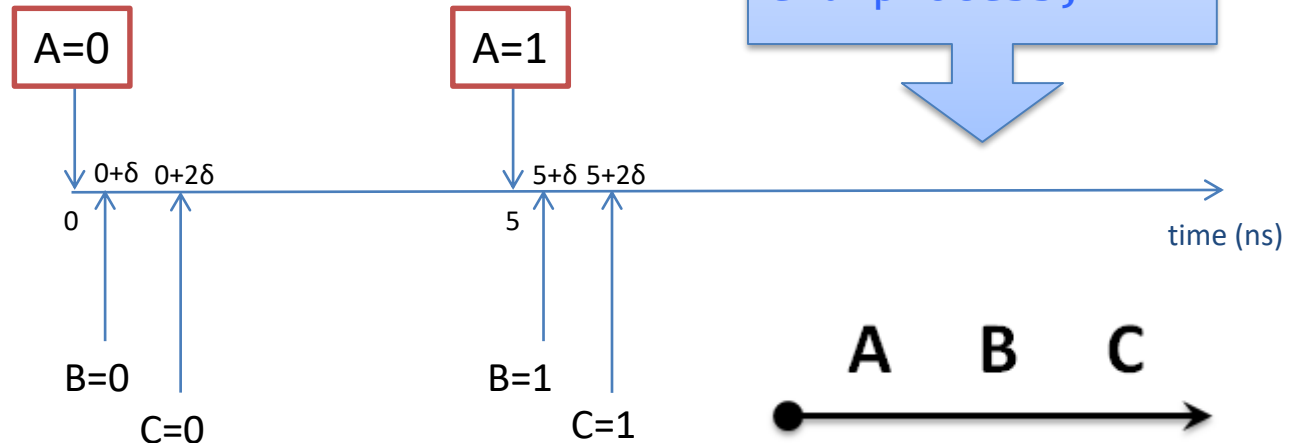
Simulation of a process

Initial values

A=U

B=U

C=U



t(ns)	0	0 + δ	0 + 2δ	No more changes	5	5 + δ	5 + 2δ	No more changes
A	0	0	0		1	1	1	
B	U	0	0		0	1	1	
C	U	U	0		0	0	1	



VHDL basic elements

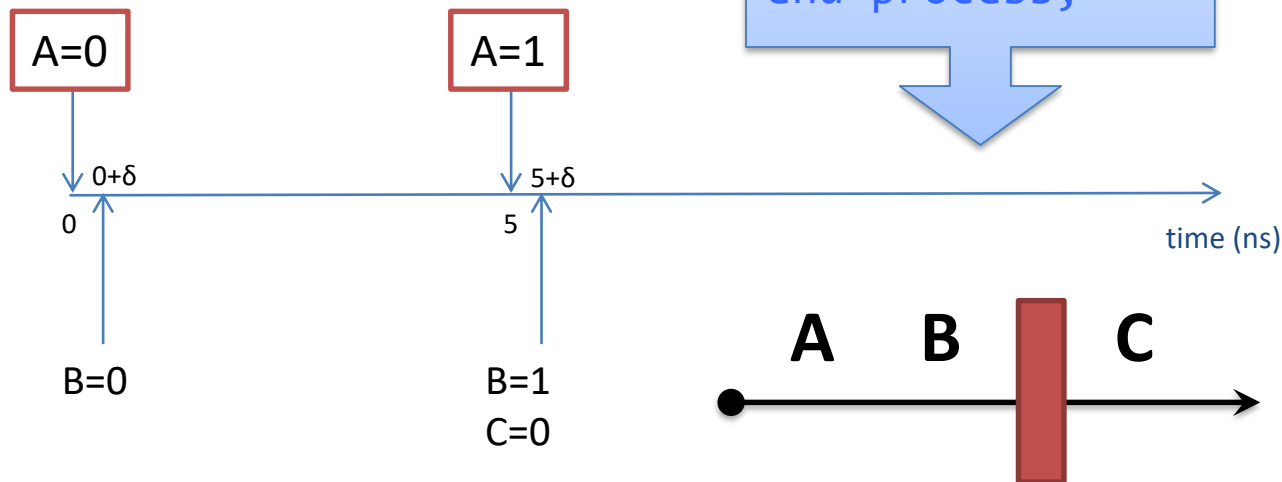
Simulation of a process

Initial values

A=U

B=U

C=U



t (ns)	0	$0 + \delta$	No more changes	5	$5 + \delta$	No more changes
A	0	0		1	1	
B	U	0		0	1	
C	U	U	No more changes	U	0	



VHDL basic elements

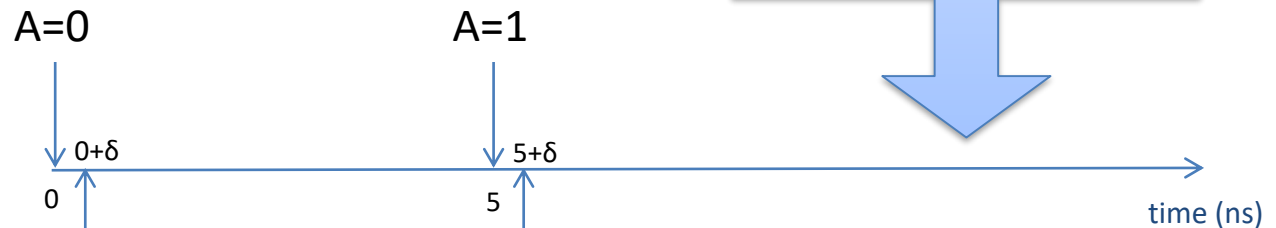
■ Simulation of a process

Initial values

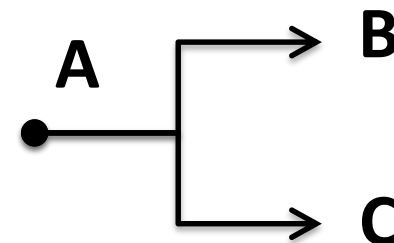
A=U

B=U

C=U



```
process (A)
  --vars d,e;
begin
  d := A;
  e := d;
  C <= d;
  B <= e;
end process;
```





VHDL basic elements

- Conditional statements
 - VHDL allows using another type of conditional statements, which have many similarities with software description languages.
 - They are always used inside a `process`.
 - As we will discuss in the next slides, incomplete conditional statements lead to sequential hardware.

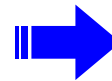


VHDL basic elements

■ Conditional statements

```
if cond-1 then  
    -- statements  
elsif cond-n then  
    -- statements  
else  
    -- statements  
end if;
```

Multiple statements



```
if a = b then  
    c <= a or b;  
elsif a < b then  
    c <= b;  
else  
    c <= "0000";  
end if;
```

The 'else' is not mandatory
(however, if we don't use it, the
code leads to sequential HW)



VHDL basic elements

- All conditional statements are placed inside a process.
- All the process have a sensitivity list.

```

process (a, b)
begin
    if a = b then
        c <= a or b;
    elsif a < b then
        c <= b;
    else
        c <= "0000";
    end if;
end process;
  
```

t = 5 ns	t = 10 ns	t = 15 ns
a <= "0010"	a <= "0010"	a <= "0010"
b <= "0010"	b <= "0110"	b <= "0001"
c <= "0010"	c <= "0110"	c <= "0000"



VHDL basic elements

- All conditional statements are placed inside a process.
- All the process have a sensitivity list.

```
process (a, b)
begin
  if a = b then
    c <= a or b;
  elsif a < b then
    c <= b;
  else
    c <= "0000";
  end if;
end process;
```

t = 5 ns	t = 10 ns	t = 15 ns
a <= "0010"	a <= "0010"	a <= "0010"
b <= "0010"	b <= "0110"	b <= "0001"
c <= "0010"	c <= "0110"	c <= "0110"

PROBLEM: The code doesn't include any final else branch



VHDL basic elements

- All conditional statements are placed inside a process.
- All the process have a sensitivity list.

```

process (a, b)
begin
    if a = b then
        c <= a or b;
    elsif a < b then
        c <= b;
    else
        c <= "0000";
    end if;
end process;

```

t = 5 ns	t = 10 ns	t = 15 ns
a <= "0010"	a <= "0010"	a <= "0010"
b <= "0010"	b <= "0110"	b <= "0001"
c <= "0010"	c <= "0010"	c <= "0010"

PROBLEM: Although there is a final *else* branch,
b is missing in the sensitivity list of the process



VHDL basic elements

- *architecture* of an 8:1 multiplexer with an **IF-THEN-ELSE** statement.

```
entity MUX_8_1 is
    port (I: in std_logic_vector(7 downto 0);
          S: in std_logic_vector(2 downto 0);
          O: out std_logic);
architecture A of MUX_8_1 is
begin
    process (I,S)
    begin
        if S = "000" then
            O <= I(0);
        elsif S = "001" then
            O <= I(1);
        elsif S = "010" then
            O <= I(2);
        --etc...
        else
            O <= I(7);
        end if;
    end process;
end architecture;
```



VHDL basic elements

■ Conditional statements:

Multiple statements

```
case expression is  
    when choice_1 =>    ... -- statements;  
    when choice_n =>    ... -- statements;  
    when others =>      ... -- statements;  
end case;
```

'when others' is not mandatory
(leads to sequential HW)

```
case RGB is  
    when "111"  => r <= '0';  
    when "100"  => r <= '1';  
    when "110"  => r <= '1';  
    when others => r <= '0';  
end case;
```



VHDL basic elements

- *architecture* of an 8:1 multiplexer with a **CASE** statement

```
entity MUX_8_1 is
  port (I: in std_logic_vector(7 downto 0);
        S: in std_logic_vector(2 downto 0);
        O: out std_logic);
architecture B of MUX_8_1 is
begin
  process (I,S)
  begin
    case S is
      when "000" => O <= I(0);
      when "001" => O <= I(1);
      when "010" => O <= I(2);
      --etc...
    end case;
  end process;
end architecture;
```

Problem with these
codes...?



VHDL basic elements

■ Loops (for testbenches)

```
[label:] loop  
    sequence-of-statements  
    -- use exit statement to get out  
end loop [label];
```

```
[label:] for var in range loop  
    sequence-of-statements  
end loop [label];
```

```
[label:] while cond loop  
    sequence-of-statements  
end loop [label];
```

VHDL basic rules



- Do not use **variables** in VHDL code if you want to synthesize it. You may use it in testbenches.
- For **combinational** processes:
 - if-then-else statements should always have an else
 - Likewise, case-when statements should have when others
 - If you assign a value to a signal in one branch of the code it should be assigned in all of them
 - We will avoid for and while loops
 - The sensitivity list must include all the signals that are read inside the process.
- A signal will not be modified in two different processes



VHDL basic elements

■ wait statements

`[label:] wait <for, until, on> [expression, signal(s)];`

– Used in processes, procedures and functions.

– Three types:

- `wait for` <time expression>
- `wait until` <Boolean expression>
- `wait on` <signal(s)>

– Examples:

- `wait for 10 ns;`
- `wait until clk='1';`
- `wait on in1;`



VHDL basic elements

■ Attributes

- S'**DELAYED**(t) is the signal value of S at time now - t.
- S'**STABLE** is true if no event is occurring on signal S.
- S'**STABLE**(t) is true if no even has occurred on signal S for t units of time.
- S'**QUIET** is true if signal S is quiet (no event this simulation cycle).
- S'**QUIET**(t) is true if signal S has been quiet for t units of time.
- S'**TRANSACTION** is a bit signal, the inverse of previous value each cycle S is active.
- S'**EVENT** is true if signal S has had an event this simulation cycle.
- S'**ACTIVE** is true if signal S is active during current simulation cycle.
- S'**LAST_EVENT** is the time since the last event on signal S.
- S'**LAST_ACTIVE** is the time since signal S was last active.
- S'**LAST_VALUE** is the previous value of signal S.



VHDL basic elements

■ Sequential statements:

They must appear inside a process (clk, ...)

```
signal'event  
signal'last_event  
signal'last_value
```

```
if clk'event and clk = '1';
```




VHDL basic elements


- Flip-flops, registers ...

`clk'event and clk='1'`
`clk'event and clk='0'`

rising edge
falling edge

- Edge-triggered D-flip flop:

```
entity D_FF is
    port(d, clk: in std_logic; q: out std_logic);
end D_FF;
```

```
architecture ARCH_D_FF of D_FF is
    process (clk, )
    begin
        if clk'event and clk = '1' then
            q <= d;
        end if;
    end process;
end ARCH_D_FF ;
```



VHDL basic elements

- D flip-flop with synchronous reset:

```
entity D_FF_SReset is
    port( d, clk, reset : in std_logic;
          q              : out std_logic );
end D_FF_SReset;

architecture ARCH_SYN of D_FF_SReset is
    process (clk)
    begin
        if clk'event and clk = '1' then
            if reset = '1' then    Q <= '0';
            else                    Q <= D;
            end if;
        end if;
    end process;
end ARCH_SYN;
```



VHDL basic elements

- D flip-flop with asynchronous reset:

```
entity D_FF_ASReset is
    port( d, clk, reset: in  std_logic;
          q          : out std_logic );
end D_FF_ASReset;

architecture ARCH_ASYNC of D_FF_ASReset is
begin
    process (clk, reset)
    begin
        if (reset = '1') then          q <= '0';
        elsif clk = '1' and clk'event then q <= d;
        end if;
    end process;
end ARCH_ASYNC;
```



VHDL basic elements

- **Packages**: Collections of definitions of customized types, functions and procedures.
 - They can be used in one or several designs.
 - Usually, they are written in separate .vhd files.
- **Syntax:**

```
package name is
    -- customized types
    -- constants
    -- component declarations
    -- functions and procedures (declarations)
end name;
package body name is
    -- functions and procedures (body)
end name;
```

You can find more details about procedures and functions in the VHDL manual available at the Campus Virtual, used in some testbenches



VHDL basic elements

- How to use packages. Example (customized types):

```
package my_package is my_package.vhd  
    type new_type is (S0, S1, S2, S3, S4, S5);  
end package my_package;
```

This is needed in order to include *my_package* in *my_entity.vhd*

```
use work.my_package.all; my_entity.vhd  
entity my_entity is  
    generic( n: natural := 32 );  
    port( a,b,c : in std_logic;  
          d      : out new_type );  
end my_entity;
```

The inputs and outputs of an entity can be something different from *std_logic* or *std_logic_vector*!!

We can even implement designs whose I/O includes signals of types different from *std_logic* or *std_logic_vector* (we will NOT do this in the labs)



VHDL basic elements

- How to use packages. Example (component declarations):

```
package my_package is                                     my_package.vhd
  component divider
    port( rst      : in  std_logic;
          clk_in   : in  std_logic;
          clk_out  : out std_logic );
  end component;
end package my_package;
```

```
use work.my_package.all;                                  counter.vhd
entity counter is
  generic (...);
  port (...);
end counter;

architecture ARCH of counter is
begin
  divider_1: divider (...);
end ARCH;
```

Now, the component declaration in `counter.vhd` is no longer necessary because it was made in `my_package.vhd`

VHDL basic elements



- **Procedures**: A procedure provides the ability to execute common pieces of code from several different places in a model.
- A procedure can contain timing controls, and it can call other procedures and functions



VHDL basic elements

- **Functions**: Functions primarily differ from procedures in that a single value is returned from executing the code.
- Functions are used if all of the following conditions are true:
 - There are no delay, timing, or event control constructs that are present
 - It returns a single value
 - There is at least one input argument
 - There are no `output` or `inout` arguments
 - There are no non-blocking assignments



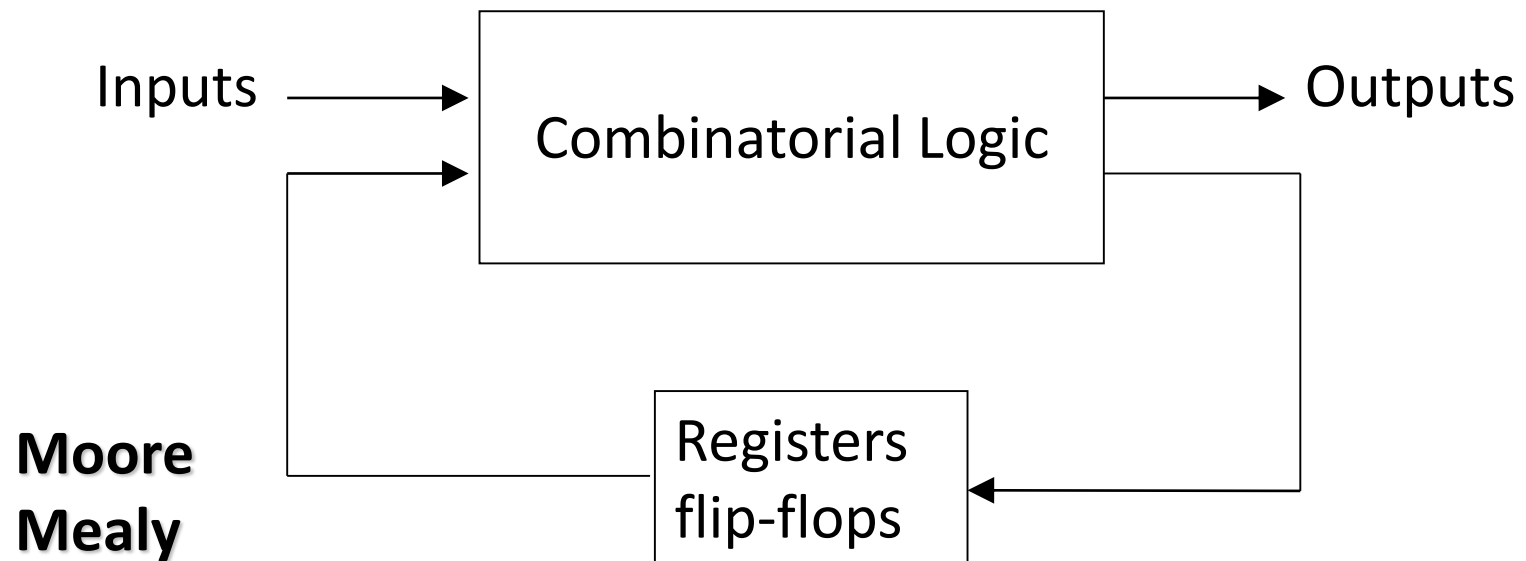
Outline

1. Design flow
2. Hardware description language
3. Simulation with VHDL
4. Structure of a VHDL code
5. VHDL basic elements
- 6. Finite state machines**
- 7. Structural definitions in VHDL**
- 8. Simulation testbenches**
- 9. Introduction to FPGAs**



Finite state machines (FSMs)

- Any sequential circuit is divided into:
 - A combinatorial logic implementing the output of the circuit and the transition function to the next state.
 - A number of storage elements (registers, flip-flops...).





Finite state machines (FSMs)

architecture beh of FSM is

--internal signal declarations;

begin

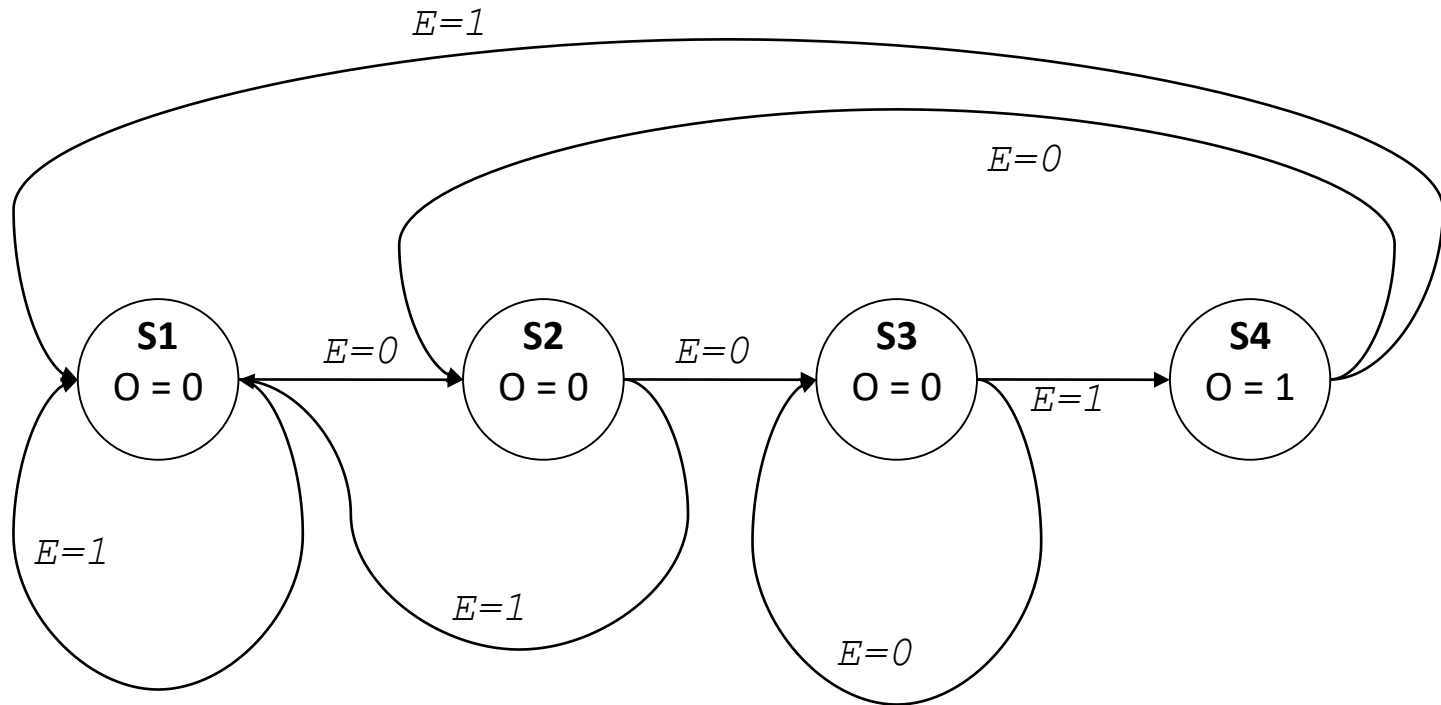
```
SYNC: process (clock, reset)
begin
  --code for flip-flops
end process SYNC;
```

```
COMB: process (sensitivity list)
begin
  --code for combinatorial logic
end process COMB;
```

end beh;



Finite state machines (FSMs)





Finite state machines (FSMs)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity FSM is
    port( reset, E, clk : in  std_logic;
          O               : out std_logic );
end FSM;

architecture ARCH of FSM is
    type STATES is (S1, S2, S3, S4);
    signal STATE, NEXT_STATE: STATES;

begin
```

```
    SYNC: process (clk)
    begin
        if clk'event and clk = '1' then
            if reset = '1' then
                STATE <= S1;
            else
                STATE <= NEXT_STATE;
            end if;
        end if;
    end process SYNC;
```

```
    COMB: process (STATE, E)
    begin
        case STATE is
            when S1 =>
                O <= '0';
                if (E='0') then NEXT_STATE <= S2;
                else NEXT_STATE <= S1;
                end if;
            when S2 =>
                O <= '0';
                if (E='0') then NEXT_STATE <= S3;
                else NEXT_STATE <= S1;
                end if;
            when S3 =>
                O <= '0';
                if (E='0') then NEXT_STATE <= S3;
                else NEXT_STATE <= S4;
                end if;
            when S4 =>
                O <= '1';
                if (E='0') then NEXT_STATE <= S2;
                else NEXT_STATE <= S1;
                end if;
        end case;
    end process COMB;
```

end ARCH;



Outline

1. Design flow
2. Hardware description language
3. Simulation with VHDL
4. Structure of a VHDL code
5. VHDL basic elements
6. Finite state machines
7. Structural definitions in VHDL
8. Simulation testbenches
9. Introduction to FPGAs



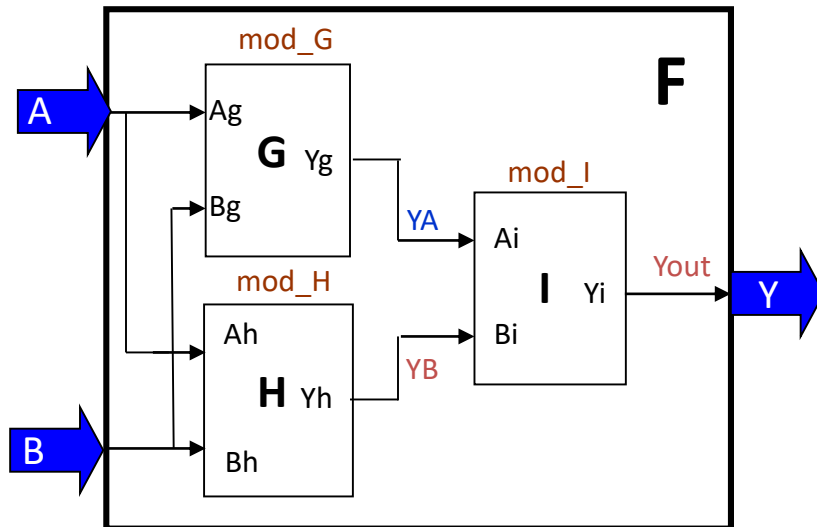
Structural description in VHDL

```
architecture arch_name of entity_name is
    component component_name
        generic (...);
        port (...);
    end component;
    -- signal declarations;
begin
    component_i: component_name
        generic map (parameter_value)
        port map (io_name)
end arch_name;
```



Structural description in VHDL

```
entity F is
  port( A, B: in  std_logic;
        Y  : out std_logic);
end F;
```



Entities G, H and I are defined in somewhere else (i.e., in separate files)

```
architecture structure of F is
  component G
    port( Ag,Bg: in  std_logic;
          Yg  : out std_logic );
  end component;
  component H
    port( Ah,Bh: in  std_logic;
          Yh  : out std_logic );
  end component;
  component I
    port( Ai,Bi: in  std_logic;
          Yi  : out std_logic );
  end component;
  signal YA,YB,Yout: std_logic;
begin
  mod_G: G port map (A, B, YA);
  mod_H: H port map (A, B, YB);
  mod_I: I port map (YA, YB, Yout);

  Y <= Yout;
end structure;
```




Structural description in VHDL

- **GENERATE:** These statements are used to automatically create an array of instances of the same component and/or other concurrent statements.

```
for index in range generate
  -- concurrent instructions
end generate;
```

```
component comp
  port (x: in std_logic;
        y: out std_logic);
end comp;
...
signal a,b: std_logic_vector(0 to 7);
...
gen: for i in 0 to 7 generate
  u: comp port map (a(i), b(i));
end generate gen;
```



Generic parameters in VHDL

- Entities can also have a list of parameters (which is OPTIONAL)

```
entity name is  
    generic (list of parameters);  
    port (list of input/output ports);  
end name;
```

list of parameters

name_parameter: type_of_parameter {:= <DEFAULT_VALUE>}; ...

n: integer := 32;

The default value for each parameter is also OPTIONAL



Generic parameters in VHDL

- The default value of the generic parameters is also OPTIONAL.
- However, **a design with parameters that have not been given any value cannot be synthesized.**
 - Example:

```
entity adder is                                     adder.vhd
    generic( n: natural );
    port( a,b : in  std_logic_vector (n-1 downto 0);
          c   : out std_logic_vector (n-1 downto 0) );
end adder;

architecture ARCH of adder is
begin
    ...
end ARCH;
```

This compiles and it is correct,
but it CANNOT be synthesized



Generic parameters in VHDL

- The default value of the generic parameters is also OPTIONAL.
- However, **a design with parameters that have not been given any value cannot be synthesized.**
 - Example:

```
entity adder is adder.vhd
    generic( n: natural := 4 );
    port( a,b : in  std_logic_vector (n-1 downto 0);
          c   : out std_logic_vector (n-1 downto 0) );
end adder;

architecture ARCH of adder is
begin
    ...
end ARCH;
```

Now, it can be synthesized
and implemented



Generic parameters in VHDL

- When instantiating that component in another file, it is necessary to assign a value to n if and only if n does not have a default value.

```
entity counter is                                counter.vhd
  generic( n: natural := 4 );
  port( rst, clk, count : in  std_logic;
        output          : out std_logic_vector (n-1 downto 0) );
end counter;
```

```
architecture ARCH of counter is
  --Declaration of component adder...
```

```
begin
```

```
...
```

```
my_adder1: adder generic map (n) port map (...);
```

```
my_adder2: adder generic map (32) port map (...);
```

```
...
```

```
end ARCH;
```

COMPULSORY if
adder.vhd DOES NOT
assign a default value to n



Generic parameters in VHDL

- When instantiating that component in another file, it is necessary to assign a value to n if and only if n does not have a default value.

```
entity counter is
    generic( n: natural := 4 );
    port( rst, clk, count : in  std_logic;
          output          : out std_logic_vector (n-1 downto 0));
end counter;

architecture ARCH of counter is
    --Declaration of component adder...

begin
    ...
    my_adder1: adder port map (...);
    my_adder2: adder port map (...);
    ...
end ARCH;
```

counter.vhd

If adder.vhd assigns a default value to n (i.e., 4), it's not necessary to assign another value. HOWEVER, here, all the adders will be 4-bits wide



Generic parameters in VHDL

- When instantiating that component in another file, it is necessary to assign a value to n if and only if n does not have a default value.

```
entity counter is                                     counter.vhd
    generic( n: natural := 4 );
    port( rst, clk, count : in  std_logic;
          output          : out std_logic_vector (n-1 downto 0));
end counter;

architecture ARCH of counter is
    --Declaration of component adder...

begin
    ...
    my_adder1: adder generic map (n) port map (...);
    my_adder2: adder generic map (8) port map (...);
    ...
end ARCH;
```

If adder.vhd assigned a default value to n (i.e., 32), it will be overridden by the new value given here



Generic parameters in VHDL

- The assignments (`others => '0'`) or (`others => '1'`) can be used to assign the values "0...0" or "1...1" to a signal of parametrizable length:

```
entity example is
    generic (n: natural := 4);
    port (...);
end example ;

architecture ARCH of example is
    signal s1, s2, s3: std_logic_vector (n-1 downto 0);

begin
    ...
    s1 <= "0000";                -- Problem if n!=4
    s2 <= (others => '0');
    s3 <= (others => '1');
    ...
end ARCH;
```



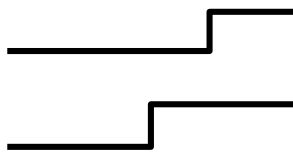

Outline

1. Design flow
2. Hardware description language
3. Simulation with VHDL
4. Structure of a VHDL code
5. VHDL basic elements
6. Finite state machines
7. Structural definitions in VHDL
8. Simulation testbenches
9. Introduction to FPGAs



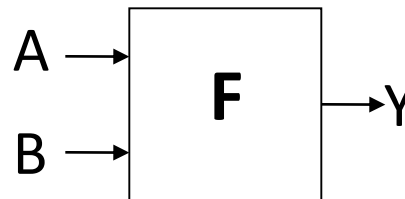
Simulation testbenches

- To know if a design works correctly, we must specify a set of input stimuli to the input ports and verify if the output values are correct



Testbench that generates the input stimuli

```
entity test_bench  
end test_bench;  
...
```



Design to verify

```
entity F is  
    port (A, ...)  
end F;  
...
```



Simulation output,
as provided by the tool



Simulation testbenches

■ Basic structure

```
entity F is
  port (A, B: in std_logic; Y out std_logic);
end F;
```

Design to verify

1) The simulation entity has no input nor output ports

```
entity test-bench
end test-bench;
```

2) The *entity* of the design to verify is added here as a *component*

```
architecture behaviour of test_bench
  component F is
    port(A, B: in std_logic; Y out std_logic);
  end component F;
  signal A, B, Y: std_logic;
begin
  ...
end architecture;
```

3) The testbench architecture must include as many signals as input/output ports the design to verify has



Simulation testbenches

Basic structure

```
...  
begin  
    uut: F port map(  
        A => A,  
        B => B,  
        Y => Y );  
  
    tb : process  
begin  
    A<='0';  
    B<='0';  
    wait for 100 ns;  
    B<= '1';  
    wait for 100 ns; -- at 200 ns  
    A<='1';  
    wait for 100 ns; -- at 300 ns  
    ...  
    wait;           -- waits forever  
end process tb;  
end;
```

4) The *component* is instantiated here, assigning its inputs and outputs to the internal signals

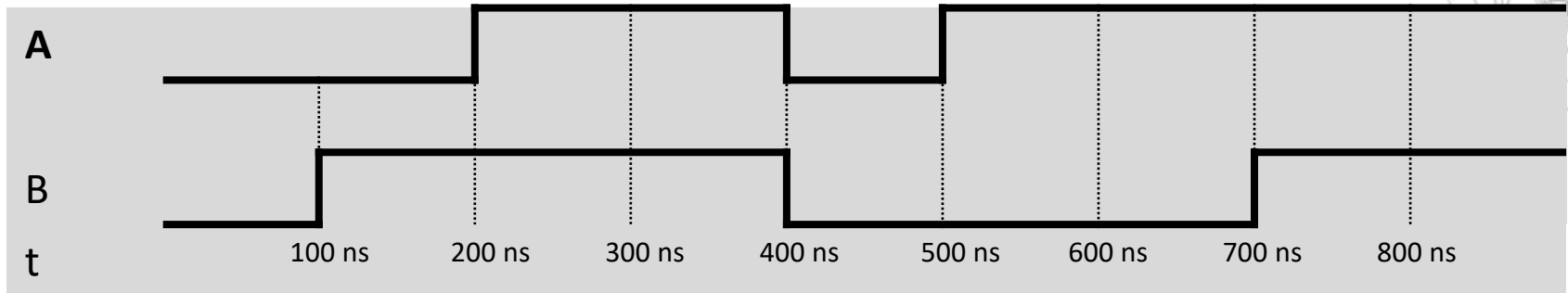
5) The simulation *process* DOES NOT have any sensitivity list

6) The initial values for the inputs are defined here

7) The desired values for the inputs are defined in the next instants of time



Simulation testbenches



```

tb : process
    begin
        A<='0';
        B<='0';
        wait for 100 ns;
        B<= '1';
        wait for 100 ns;           -- at 200 ns
        A<='1';
        wait for 200 ns;          -- at 400 ns
        A<='0';
        B<='0';
        wait for 100 ns;          -- at 500 ns
        A<='1';
        wait for 200 ns;          -- at 700 ns
        B<='1';
        wait for 100 ns;          -- at 800 ns
        wait;
    end process tb;

```



Simulation testbenches

- How to implement a periodic input: clk

```
clk_process: process
    constant clk_period : time := 10 ns;
begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
end process;
```

- Since this *process* does not end with a *wait* statement (without *for*), it will be executed all over again indefinitely.



Outline

1. Design flow
2. Hardware description language
3. Simulation with VHDL
4. Structure of a VHDL code
5. VHDL basic elements
6. Finite state machines
7. Structural definitions in VHDL
8. Simulation testbenches
9. Introduction to FPGAs



What is an FPGA?

- An FPGA (Field-Programmable Gate Array) is a hardware device for automatic prototyping.
 - They were initially conceived to serve as electronic trainers that implemented digital circuits automatically.
 - Due to their huge processing capacity, versatility and the variety of design tools available, today many commercial circuits feature FPGAs.

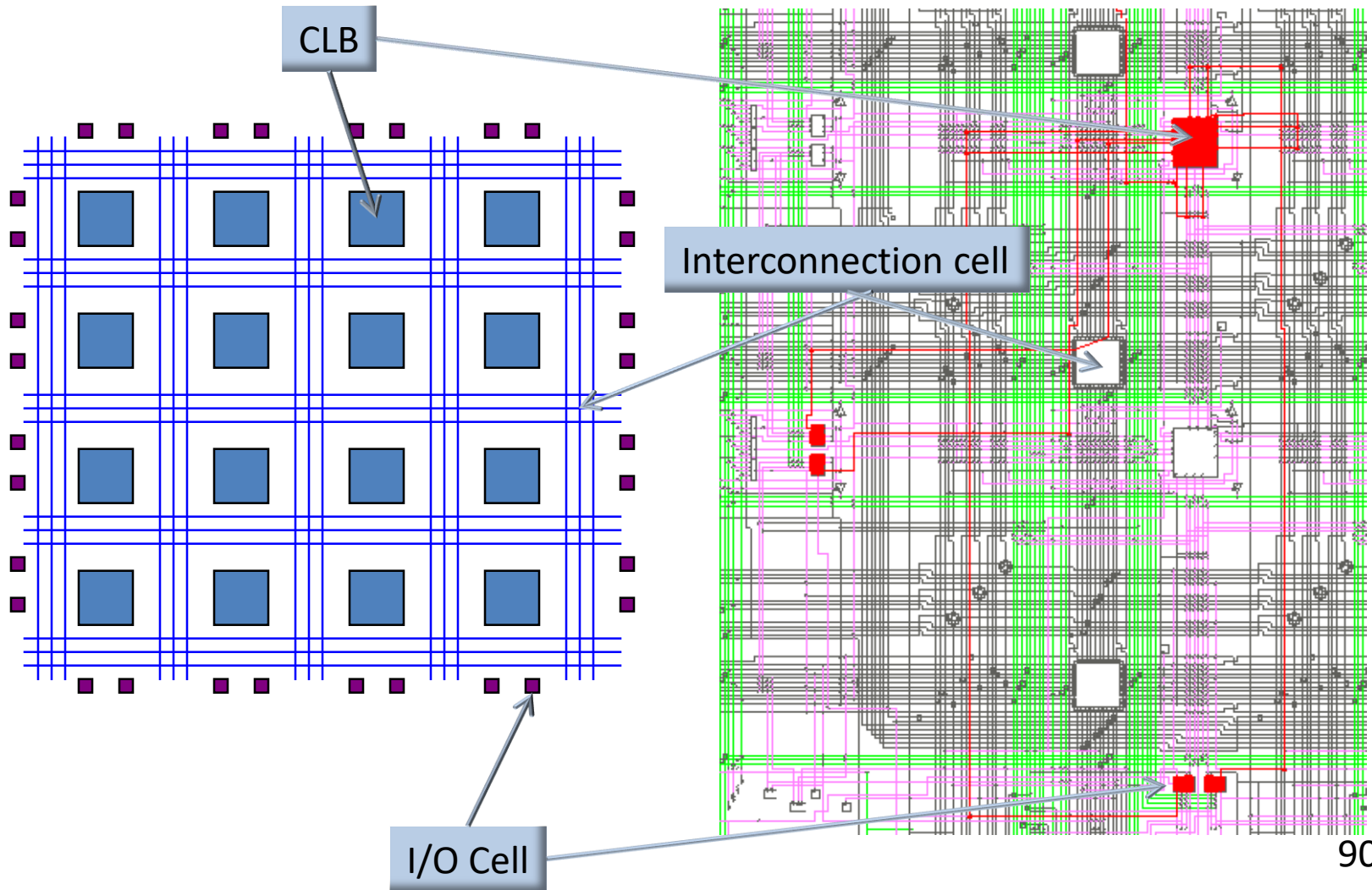


What is an FPGA?

- Basic components of FPGAs:
 - Input/output cells.
 - Configurable Logic Blocks (CLBs).
 - Interconnection programmable cells.
- Components that may be found in some FPGAs:
 - Memories.
 - Multipliers.
 - Microcontrollers.



What is an FPGA?





What is an FPGA?

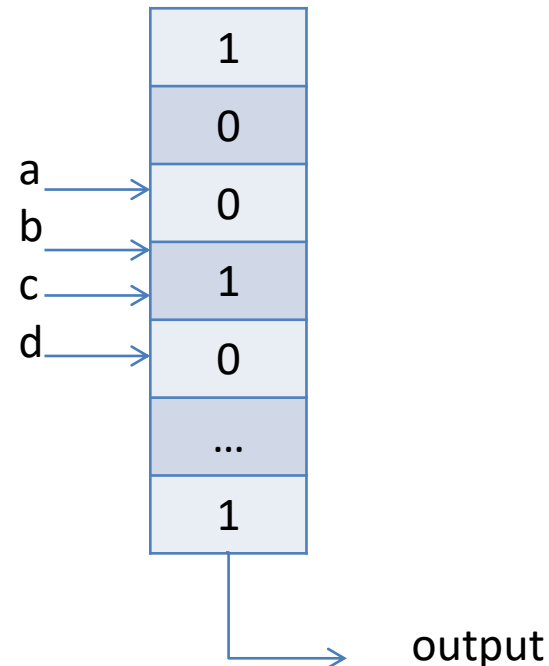
- Configurable Logic Blocks (CLBs):
 - Look-Up Tables (LUTs):
 - ROM memories with 1x16 bits of storage capacity.
 - They can implement ANY logic function with 4 inputs.
 - Flip-flops.
 - Used in case the cell must implement sequential hardware.
 - Configured as edge-triggered or level-triggered flip-flops.
 - Multiplexers.
 - Used to connect inputs to modules or modules among them.



What is an FPGA?

- LUTs as an universal set.
 - A ROM memory can implement any logic function.

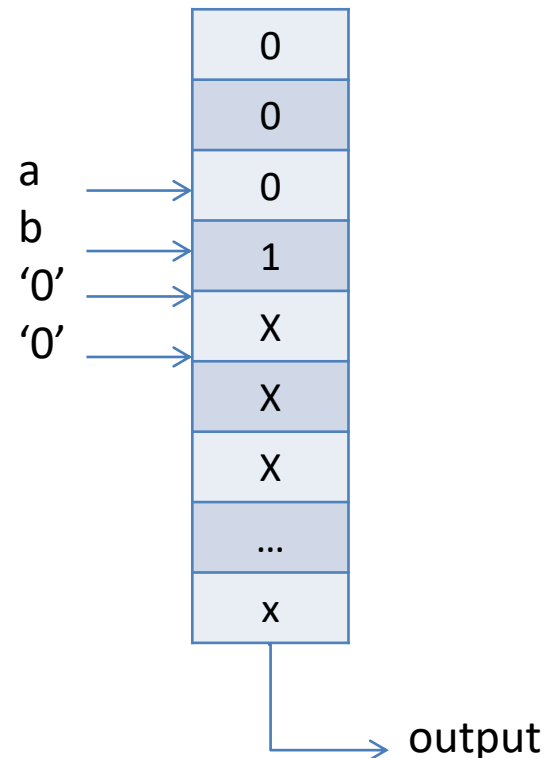
a	b	c	d		output
0	0	0	0		1
0	0	0	1		0
0	0	1	0		0
0	0	1	1		1
0	1	0	0		0
...					
1	1	1	1		1





What is an FPGA?

- LUTs as an universal set.
 - A ROM memory can implement any logic function.
 - For instance, an AND gate with 2 inputs:





Xilinx FPGAs

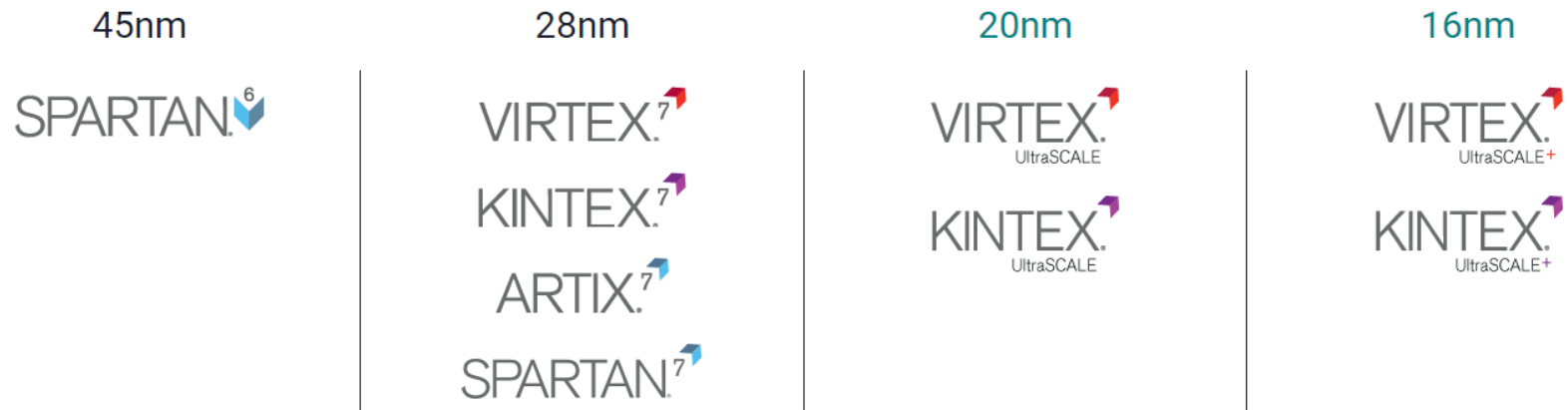
- Xilinx offers several families of devices, targeting different sectors of the market (low cost, high performance, or something in between, among others).

Xilinx - Adaptable. Intelligent. > Devices > FPGAs & 3D ICs

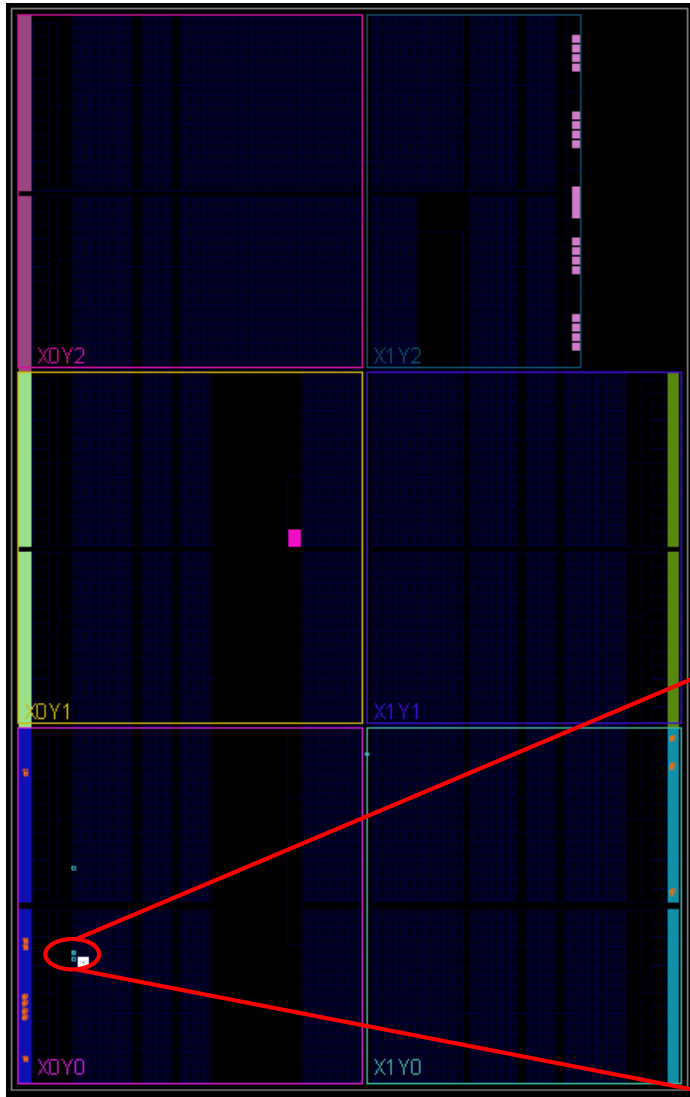
FPGA Leadership across Multiple Process Nodes

Xilinx offers a comprehensive multi-node portfolio to address requirements across a wide set of applications. Whether you are designing a state-of-the art, high-performance networking application requiring the highest capacity, bandwidth, and performance, or looking for a low-cost, small footprint FPGA to take your software-defined technology to the next level, Xilinx FPGAs and 3D ICs provide you with system integration while optimizing for performance/watt.

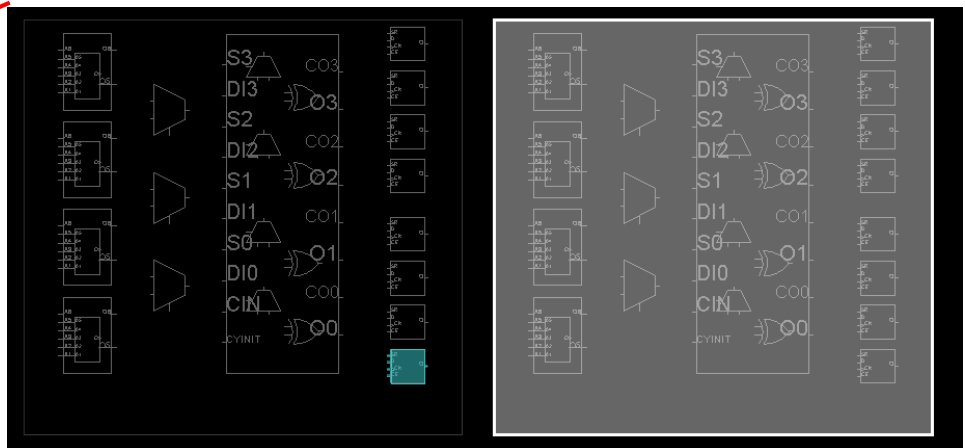
Xilinx Multi-Node Product Portfolio Offering



What is an FPGA?



- Digilent Basys 3 board
 - It features an Artix-7 FPGA (XC7A35T-1CPG236C)
 - 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops)
 - 1,800 Kbits of fast block RAM
 - 4-digit 7-segment display
 - 5 user pushbuttons
 - 90 DSP slices
 - 12-bit VGA output
 - 16 user LEDs
 - 16 user switches
- Lab1b implemented in this FPGA:





What is an FPGA?

- Synthesis tools automatically manage all the complexity of FPGAs (there may be more than one million of logic cells).
 - These tools choose the logic cells that are going to be used for implementation of the circuit, as well as the interconnections among them and the initial values of the LUTs.
 - Everything is done automatically.
 - The starting point is a synthesizable HDL description of a circuit.