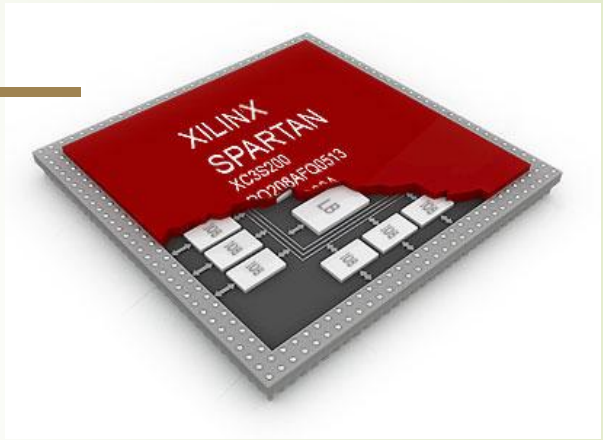
Introduction to VHDL



Embedded Systems
Università degli Studi dell'Aquila

Outline

- Hardware Description Languages
- Concurrent and sequential area of VHDL
- Simulation
- GCD Calculator
- Testbench
- FPGA Design Tools
- References

Domains and abstraction levels

- A digital circuit can be described in the:
  - Physical domain: the system is described through physical devices.
  - Structural domain: the system is described by means of sub-blocks.
  - Behavioural domain: it is described only the functionality of the system, i.e., the relation between inputs and outputs.

Y-chart

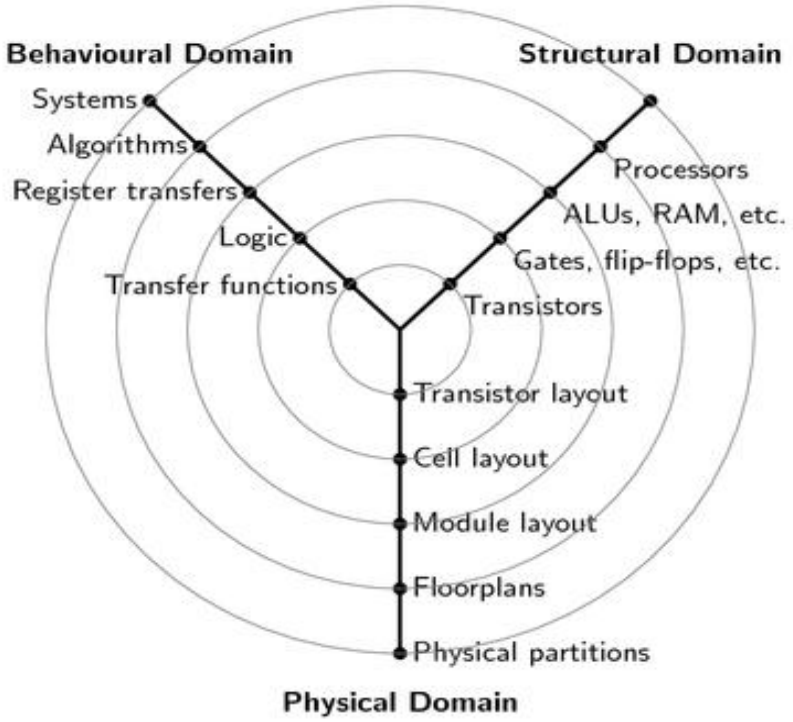


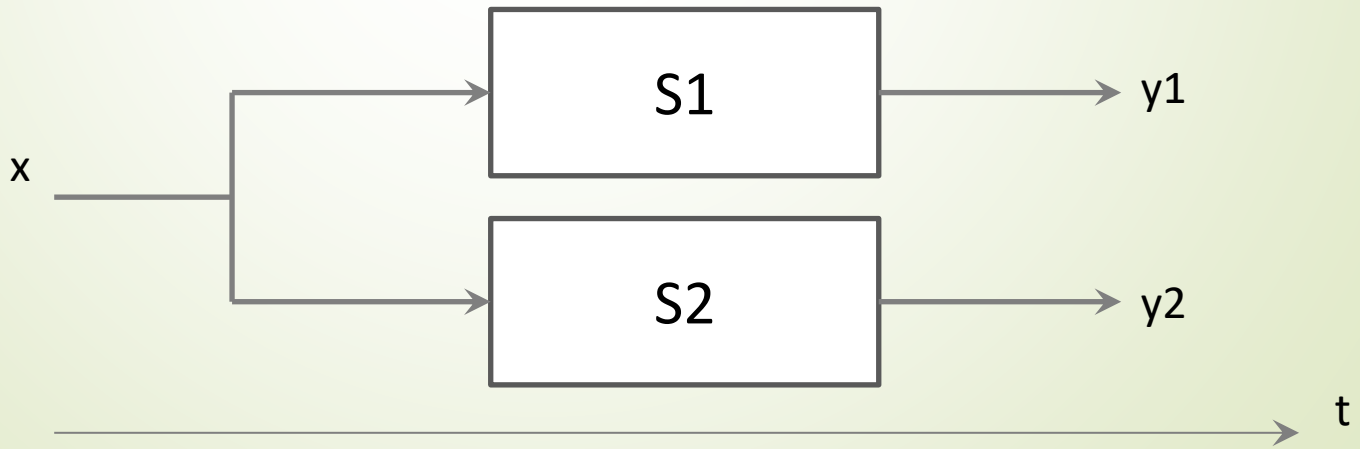
Figure 1: Gajski-Kuhn Y-chart

Hardware description languages

- An HDL (Hardware Description Language) is a language for the description of digital electronic circuits
- Hardware functionalities are described independently from the implementation
- How many types of implementation do you know?
  - Recall the Hardware Technologies chapter

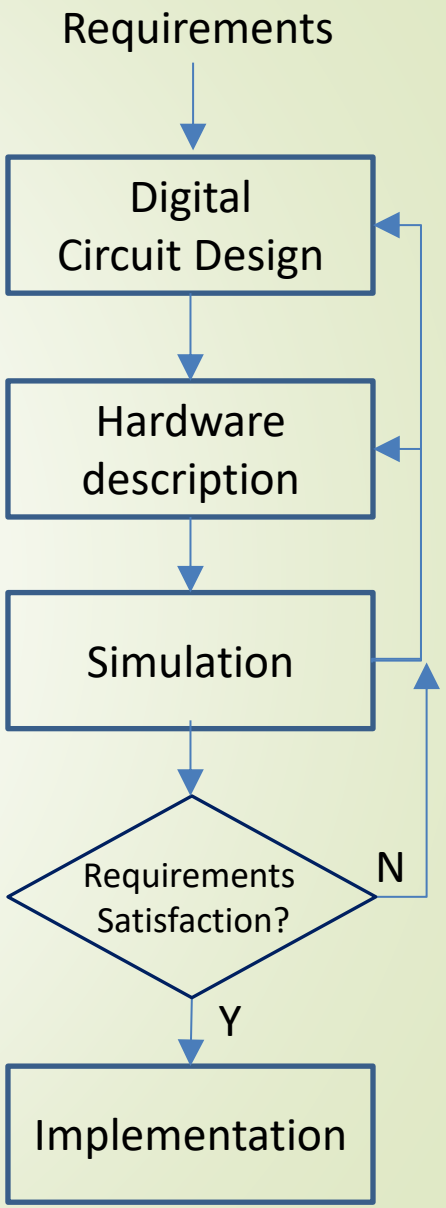
Key distinctive elements of HDL

- With respect to a programming language, HDL languages allow to describe:
  - The timing characteristics: within hardware systems, there is the propagation delay of signals.
  - The parallelism: blocks that compose a particular system evolve at the same time during a particular time interval.

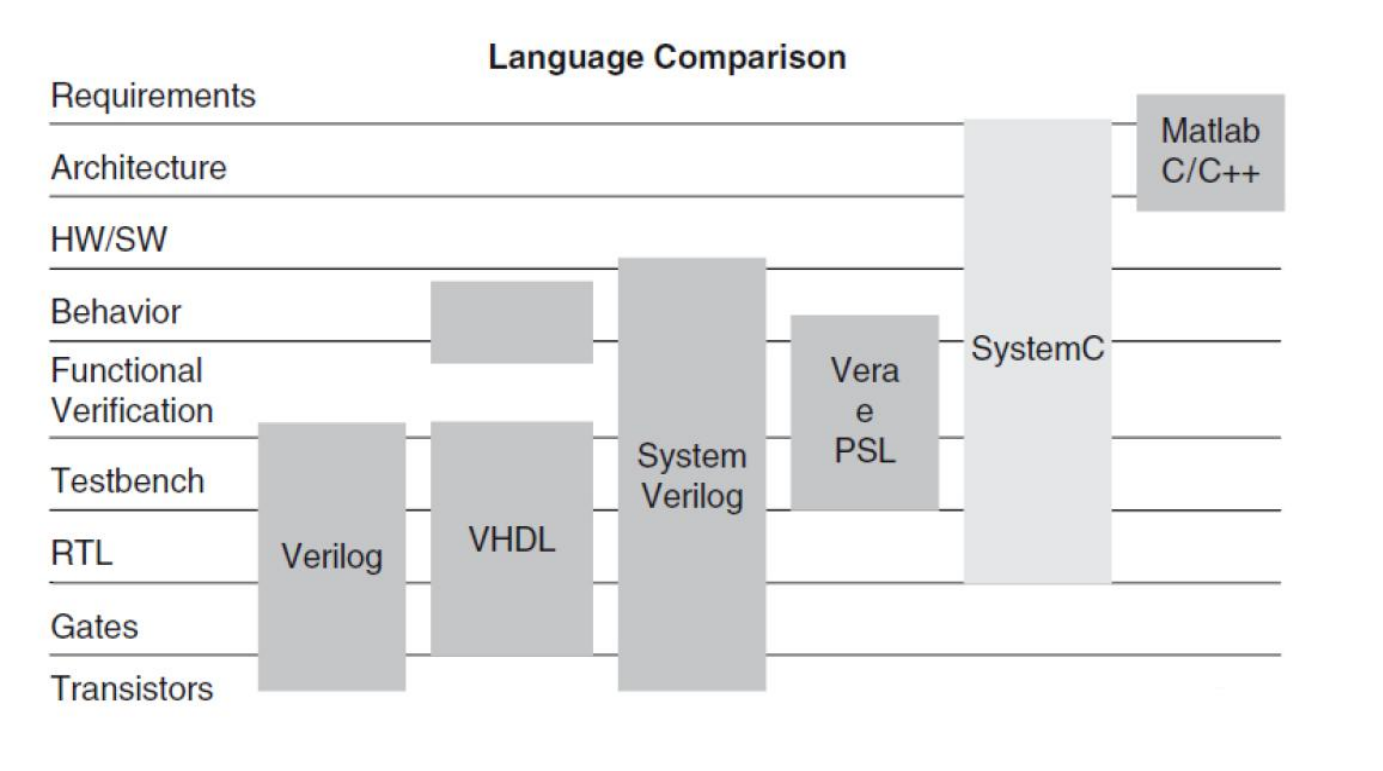


Description goals

- A description language is not executable
- Digital circuit description has the following goals:
  - Simulation: it is checked if the circuit behaviour has the expected trend.
  - Synthesis: a final implementation is produced starting from the HDL.
- Not all the constructs that can be simulated are synthesizable.



Main description languages



* Reference: SystemC: from the ground up, Wiley, 2010

VHDL language

- VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language
- It has been developed in the early ‘80 years from Defense Department of USA and publicly released in 1985.
- Standard IEEE 1076-1987, reviewed in 1993 and 2008

VHDL vs Verilog

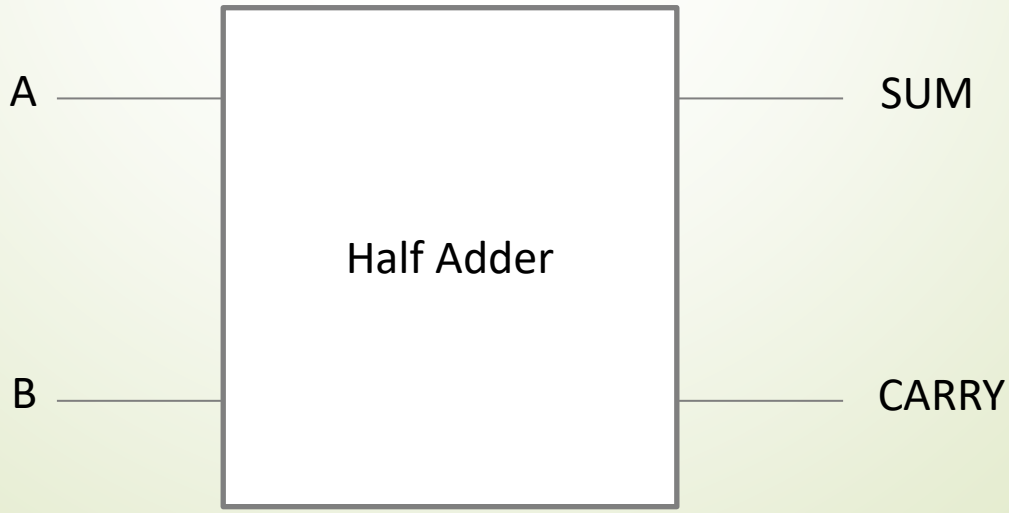
	VHDL	Verilog
Reusability	Packages	-
Easy to understand	Strongly typed	More intuitive
Back annotation	-	Verilog uses the Standard Delay Format (SDF)
High level constructs	Many available constructs	-
Large designs	Package and Generate	verbose descriptions
Readability	Similar to ADA	Similar to C
Structural replication	Generate	-

Design units

- Design Units are VHDL code segments that can be separately processed and included in a library.
- A VHDL description of a digital system must contain at least one entity and one architecture.
  - An entity specifies univocally the identifying name of a system and the characteristics of its inputs and outputs.
  - In the architecture, instead, the circuit functionality is described.

A simple example: half adder

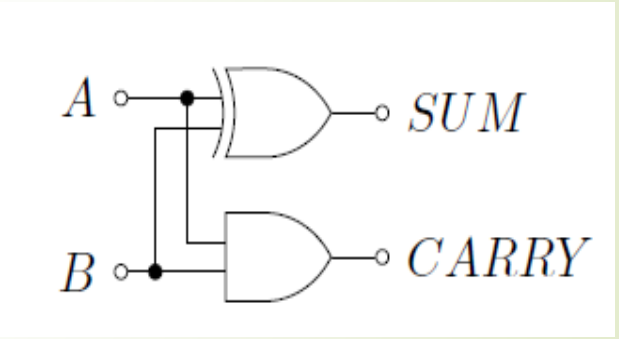
- Half adder:
  - Receives 2 bits as input (A,B);
  - Generates an output bit equal to the sum of 2 input bits (SUM);
  - Generates an output bit equal to carry (CARRY);



Half adder: functional specifications

- From functional description it is possible to fill the truth table and obtain the circuit structure

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



VHDL Data types

- In VHDL, *bit* and *bitvector* types are available for digital data representation
- It is usually preferred to use *std_logic* and *std_logic_vector* defined in the *std_logic_1164* library
- The library must always be declared in the following manner:

```
library IEEE ;
use IEEE.std_logic_1164.ALL;
```

Half adder: entity

- The following VHDL entity corresponds to the top level of the circuit:

```
-- Entity declaration of half adder

entity half_adder is
port
(
   A: in bit;
   B: in bit;
   SUM: out bit;
   CARRY: out bit
);
end half_adder ;
```

Half adder: architecture (1)

- The functionality is described in the architecture associated to the entity

```
architecture ha_arch of half_adder is

begin

    Concurrent area

end ha_arch ;
```

- Instructions contained in the concurrent area are treated as evolving in parallel.

Concurrent area: constructs (1)

- Concurrent assignment:
  - The **<=** operator is used to assign values to a signal

```
Z <= A;
Z <= '1';
Z_vector <= "10001";
```

- Conditional assignment:
  - When a certain condition is verified, a value is assigned to a signal

```
Z <= A when SEL = "00" else
     B when SEL = "11" else
     C;
```

Concurrent area: constructs (2)

- Assignment with selection:
  - Similar to the conditional assignment, it allows to quickly enumerate a lot of different cases
  - Similar to if and switch-case instructions of C language

```
with SEL select
Z <= A when "000",
     B when "001",
     C when "010",
     D when "011"
     E when others ;
```

Half adder: architecture (2)

- Boolean operators are used to describe computations, while the assignment operator is used to describe the functionality

```
architecture ha_arch of half_adder is
begin
    SUM <= A xor B;
    CARRY <= A and B;
end ha_arch ;
```

- Each architecture makes reference at only one entity, but multiple architectures can refer to the same entity (set by means of an opportune configuration file, not detailed here).

IEEE std_logic_1164 types

- The defined types in std_logic_1164 library use a 9 values logic system:
  - '0': zero value;
  - '1': one value;
  - 'Z': high impedance;
  - 'U': uninitialized;
  - 'X': undefined;
  - '-': don't care;

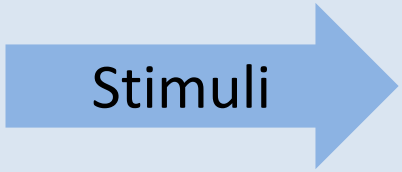
```
library IEEE ;
use IEEE.STD_LOGIC_1164.ALL;
  ...
signal X: std_logic;
signal Y: std_logic_vector (7 downto 0);
```

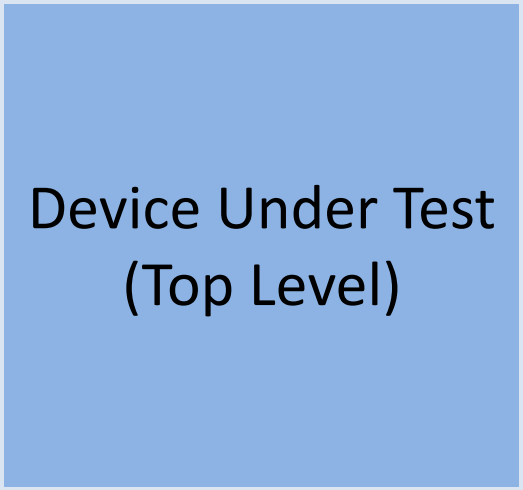
Half-Adder in Vivado

LIVE SHOW

Testbench

```
entity testbench
```





Testbench

- The simulation of a circuit requires a model of the circuit and a description of the stimuli to be applied
- In VHDL this is accomplished using a testbench, i.e. a module:
  - With no inputs and no outputs
  - Including the system to be simulated
  - Whose only functionality is to apply a proper set of stimuli with the desired timing
- The testbenches are usually the only part of the VHDL code including temporal (non-synthesizable) statements

```
y <= "00" after 5s;
```

Simulation: general concepts

- The simulation process allows the verification of the described circuit.

- The simulation can be:
  - A functional simulation, used to verify that the system effectively implements the desired functionality. This type of simulation requires the definition of an appropriate set of *stimuli*.

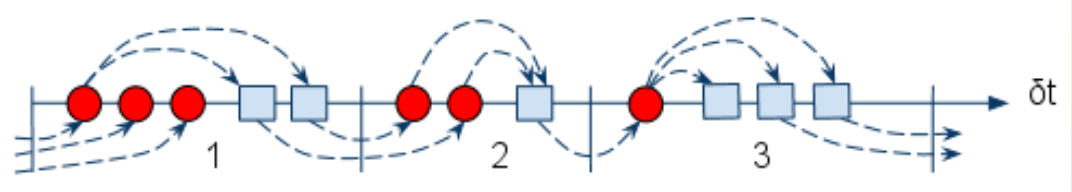
  - A timing simulation, used to verify the (temporal) performance of the circuit. This type of simulation requires an opportune model of a platform, closely related to the selected hardware technology.

HDL simulation

- HDL simulators are *Event Driven* simulators
- HDL represents hardware, that are composed of nodes that change logic values during the circuit work
- Two concepts of time
  - *Delta Time*
    - Advances through multiple *delta cycles*
  - *Actual Time*
    - Advances through multiple *time steps*
- The *delta cycle* is an HDL concept used to order events that occur in zero physical time

HDL simulation

- General HDL simulation: relation between delta cycles and time steps
  - Whenever a signal changes, that signal output fans out to the input of other logic that may cause other signals to change. This happens repeatedly, and each iteration could be considered a delta cycle. The iteration stops when there are no more signals changing and then the simulator is ready to advance time with a time step.



- Circles: signals updates (see next slides)
- Squares: process evaluation (see next slides)

Simulation tools

- FPGAs Integrated Development Environments (IDE) usually include a simulation tool or support the use of external ones.
- One of the most known simulation tools is ModelSim from Mentor Graphics.
  - ModelSim versions are provided by Mentor Graphics and often integrated within manufacturers Design Suites.
- An open source tool to consider: GHDL (Gnu HDL)
- Open source suite to simulate HDL: Integrated Infrastructure for Collaborative Open Source IC Tools (IIC-OSIC-TOOLS)



Simulation tools

- A simulation tool allows the verification (functional or timing) of a system given the set of input signals.
- Besides testbenches, many simulators allow to manually specify the set of stimuli.
- Waveform Generators are usually graphical interface tools that allow users to define with relative ease the timing diagrams of the input signals.

HOW CAN WE DEVELOP A TESTBENCH?

WE NEED TO DIG INTO THE DESCRIPTION STYLES OF VHDL

Description styles

- VHDL language allows the description of the circuit behavior following three different **styles**:
  - Behavioural: the circuit functionality in algorithmic terms is described, or it is specified how data move internally to the system;
  - Structural: the system is described as interconnection of base components;
  - Dataflow: the flow of data internally to the system is described, using concurrent assignments;
- Different types of descriptions can be combined depending on the necessity.

Example: full adder

- A full adder is a digital circuit with three inputs and two outputs.
- The sum of the two input bits is calculated, and it can account for a carry coming from previous sum



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full adder: entity

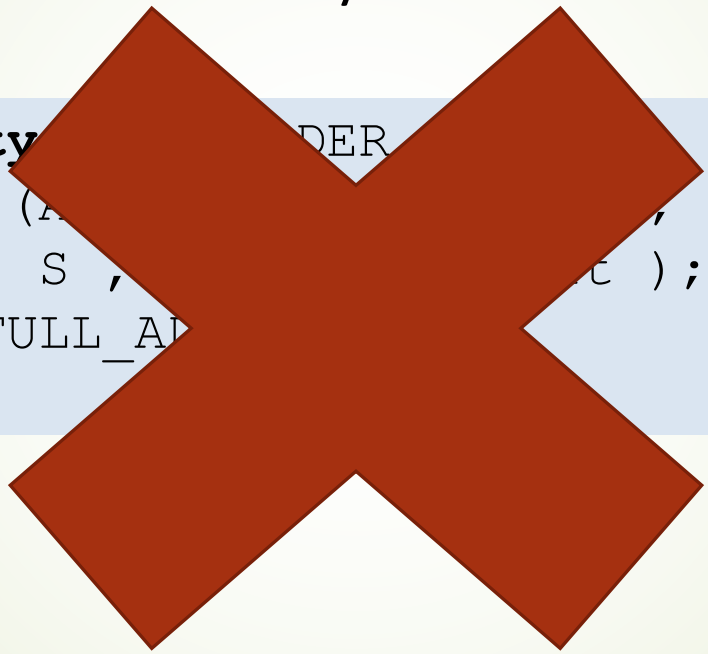
- A full adder entity:

```
entity FULL_ADDER is
port (A, B, Cin : in bit ;
      S , Cout : out bit );
end FULL_ADDER ;
```

Full adder: entity

- A full adder entity:

```
entity      DER
port (A                 ,
      S ,               t );
end FULL_A
```



BAD DESCRIPTION STYLE !

Full adder: entity

- A full adder entity:

```
entity FULL_ADDER is
port (A : in bit;
      B : in bit;
      Cin : in bit ;
      S : out bit;
      Cout : out bit
      );
end FULL_ADDER ;
```

GOOD DESCRIPTION STYLE !

Full Adder: architecture

```
architecture Behavioural of full_adder is
begin

    Concurrent Area

end Behavioural;
```

Full Adder architecture (Behavioural)

- The behavioural level is the highest-level description available in VHDL language.
- The base construct is the process: a process allows to open a "sequential evolution area" internally to a concurrent area.

```
architecture Behavioural of full_adder is
begin

    Concurrent Area

end Behavioural;
```

Full Adder architecture (Behavioural)

- The behavioural level is the highest-level description available in VHDL language.
- The base construct is the process: a process allows to open a "sequential evolution area" internally to a concurrent area.

```
architecture Behavioural of full_adder is
begin
```

Concurrent Area
Sequential Area
Concurrent Area

```
end Behavioural;
```

Full Adder architecture (Behavioural)

- The behavioural level is the highest-level description available in VHDL language.
- The base construct is the process: a process allows to open a "sequential evolution area" internally to a concurrent area.

```
architecture Behavioural of full_adder is
begin
                    Concurrent Area
sum : process (A,B,Cin)
begin

end process;
                    Concurrent Area
end Behavioural;
```

- Description of the Truth Table of the Full Adder using a Process

Process

- A process can be declared everywhere in the architecture body, because what is inside that evolves concurrently with other statements in concurrent area.
- A process opens a sequential area internally to the concurrent area: statements contained in a process evolve serially.
- The evolution happens in correspondence of a specified event in the sensitivity list.

Sensitivity list

- The sensitivity list contains all the signals to which the process must be sensible.
- Each time that a signal in the sensitivity list changes value, the process is activated.

```
architecture Behavioural of full_adder is
begin
sum : process (A,B,Cin)
begin

end process;
end Behavioural;
```

Sensitivity List

Sequential Area

Sequential area constructs (1)

- Internally to the process, VHDL statements evolve sequentially "as in a classical programming language".
- Conditional choice, for example, is available:

```
if condition then
   sequential statements
[ elsif condition then
   sequential statements ]
[ else
   sequential statements ]
end if;
```

Sequential area constructs (2)

- Multiple choice: it is the same of switch-case construct:

```
case expression is
   when choice =>
    sequential statements
   when choice =>
    sequential statements
   [when others =>
    sequential statements]
end case ;
```

- Other constructs are available in sequential area.

Full Adder architecture (Behavioural) (1)

```
architecture Behavioural of full_adder is
begin
 process(A,B,Cin)
 begin
  if(A='0' and B='0' and Cin='0')then
   S<='0';
   Cout<='0';
  elsif( A='0' and B='0' and C='1')then
   S<='1';
   Cout<='0';
  elsif( A='0' and B='1' and C='0')then
   S<='1';
   Cout<='0';
  elsif( A='0' and B='1' and C='1')then
   S<='0';
   Cout<='1';
```

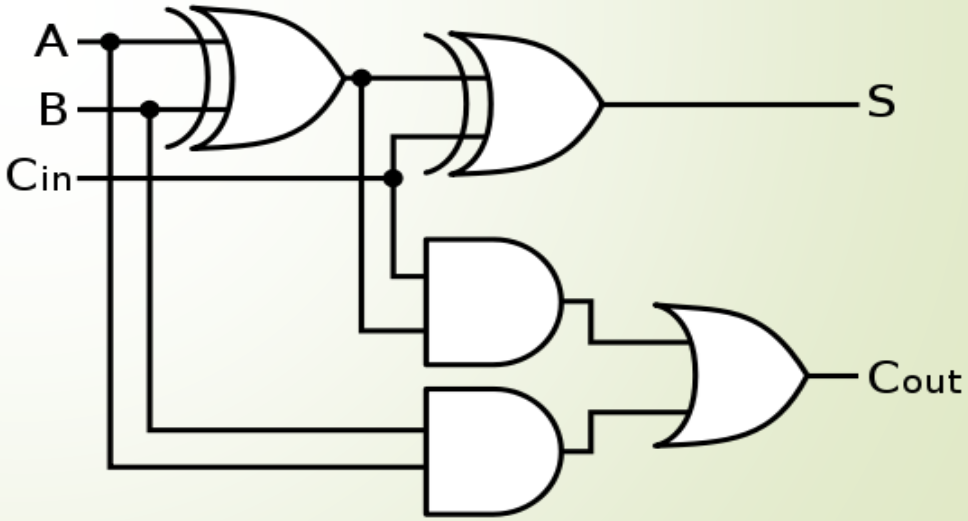
Full Adder architecture (Behavioural) (2)

```
elsif( A='1' and B='0' and C='0')then
   S<='1';
   Cout<='0';
  elsif( A='1' and B='0' and C='1')then
   S<='0';
   Cout<='1';
  elsif( A='1' and B='1' and C='0')then
   S<='0';
   Cout<='1';
  else
   S<='1';
   Cout<='1';
  end if;
 end process;
end Behavioural ;
```

Full Adder description

- We can synthesize the Full Adder with a "direct synthesis", obtaining a logic gates network.

A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Full Adder architecture (Dataflow)

- The propagation of flow of data internally to the circuit is described
- For example, in the Full-Adder:

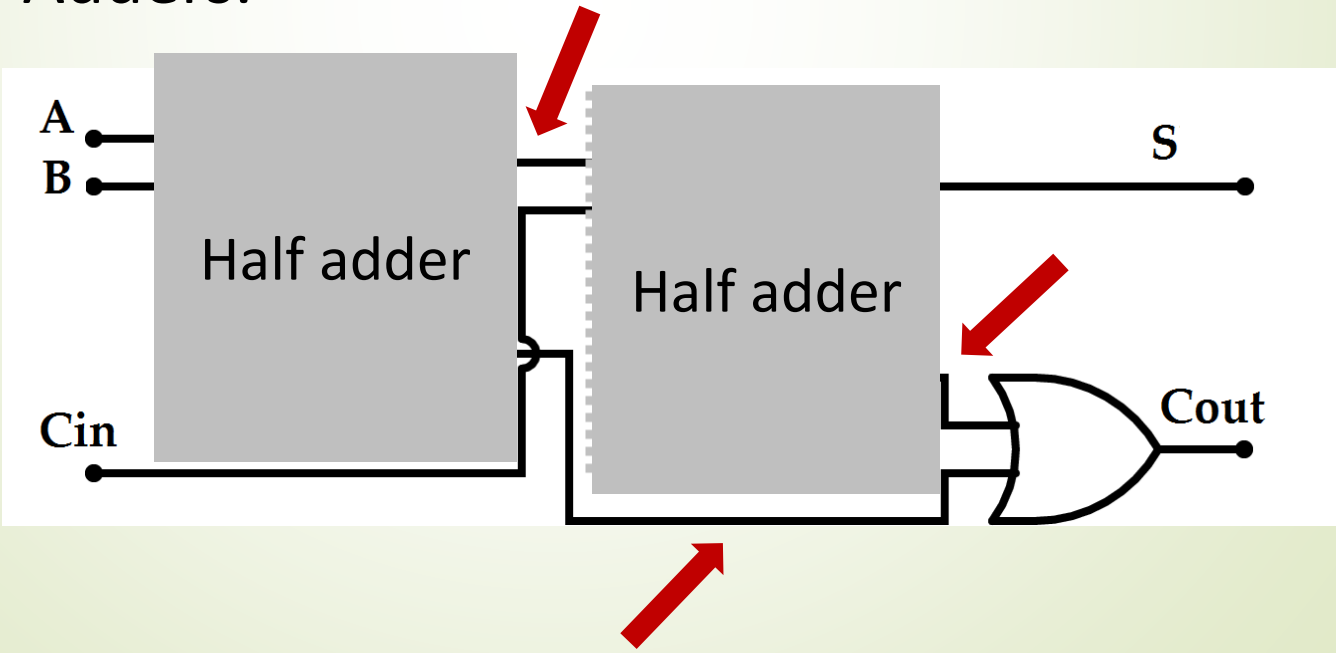
```
architecture dataflow of full_adder is

S <= A xor B xor C_in ;
Cout <= (A and B) or (C_in and (A or B));

end dataflow ;
```

Full Adder as multiple Half Adders

It can be noted, from the Truth Table, that the Full Adder functionality can be obtained by appropriately combining two Half-Adders.



Signals and Components

- The architecture describes the functionality of the instantiated circuit.
- Over the architecture body itself it is present also a declarative area in which is possible to declare signals and components:

```
architecture name of entity_name is

component INV
    port ( ... );
end component ;

signal s1,c1,c2 : bit;

begin
```

Declaration Area

- Signals are not visible out of the module itself.

Structural view

- By using components, it is possible to model system as interconnection of previous defined subsystems:
  - The **port map** construct allows to instantiate the components and to define the interconnections.
  - The signals are used to represent connections among components.

Full Adder architecture (Structural) (1)

```
architecture struct of full_adder is

signal s1 ,c1 ,c2: bit ;

component half_adder
port (A: in bit;
      B: in bit ;
      S: out bit;
      Cout: out bit
);
end component ;
```

Full Adder architecture (Structural) (2)

```
begin
    HA1 : half_adder
        port map( A => a,,
                  B => b,
                  S => s1,
                  Cout => c1
                );
    HA2 : half_adder
        port map( A => s1,
                  B => Cin,
                  S => S,
                  Cout => c2
                );
    Cout <= c1 or c2;

end struct ;
```

Full-Adder in Vivado

LIVE SHOW

GOING BACK TO TESTBENCH

Developing a testbench

- A testbench is usually the entity of higher level in a project

```
entity testbench1 is
end testbench1;
```

- It will include the device to be tested as a component and as many signals as the inputs to be excited.
- Implementation of a testbench:
  - Connect declared signals to the inputs of the system under test using a *port map*
  - Specify the temporal behaviour of the signals to obtain the sequences of desired inputs

Half-Adder simulation in Vivado

LIVE SHOW

Full-Adder simulation in Vivado

LIVE SHOW

Ready to test by yourself?



Repository of *Homelab FPGA*

Link:

https://github.com/alkalir/EmbeddedSystemsUnivaq_VHDL

WHAT CAN WE IMPLEMENT AS DIGITAL CIRCUIT?

Combinatorial circuits (1)

- A combinatorial circuit can be described by a logic expression, combination of Boolean operators *and*, *or*, *not*
  - The VHDL makes available these operators

```
Y <= not(A) and B;
```

- Alternatively, it is possible to use a truth table
  - Into VHDL a concurrent conditional assignment is used
  - The table can be also not completely specified

```
with X select
Y <= "01" when "00",
     "10" when "11",
     "--" when others;
```

Combinatorial circuits (2)

- A combinatorial circuit can be described also using a process
  - Recall the truth table description for the full adder, that is a combinatorial block
  - Combinatorial means that outputs are evaluated whenever there is an input variation. Outputs are produced with a delay depending on the circuit complexity.
  - From the point of view of hardware description, this means that we need to enter into a VHDL-process every time there is an input variation.
  - Recalling that theory of VHDL-processes expects sensitivity lists, we can conclude that, in order to describe a combinatorial circuit with a process, all the inputs must be in the sensitivity list.

Sequential circuits

- They can be synchronous and asynchronous. In digital design it is preferred to use synchronous systems.
- In VHDL the description of sequential circuits can be done by means of process statement
  - Example: Counter

```
-- Commutation on clock rising edge:
counter : process(CLOCK)
begin
if(CLOCK ='1' and CLOCK'event)
    COUNT <= COUNT +1;
end if;
end process;
```

Attributes

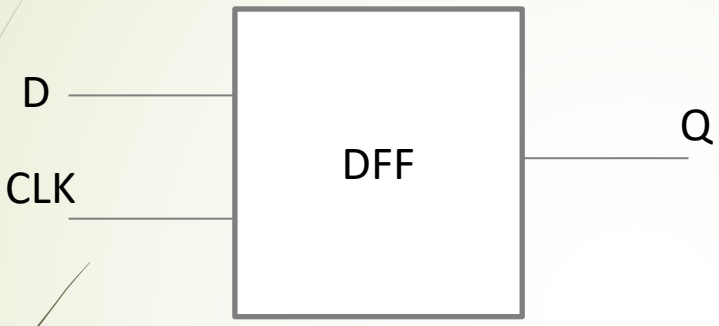
- Signals change value while system is running
- Understanding the signal information is possible using attributes

- Example - Event attribute:

  Signal'event

  - Returns a Boolean value, that is True if a signal has events

Example: 1-bit Register (D-type flip-flop)



- The input value (D) is transferred to the output (Q) only in the correspondence of an edge (for example rising edge) of the clock signal (CLK)

```
-- D type Flip Flop

entity dff is
port (D: in std_logic;
      CLK: in std_logic;
      Q: out std_logic);
end dff;
```

Example: D-type flip-flop

```
architecture arch1 of dff is
begin

process (CLK)
   begin
   if (CLK ='1' and CLK'event) then
    Q <= D;
   end if;
   end process;
end arch1;
```

Generic construct

- VHDL supports the construction of parametric components through the *generic* construct

```
entity half_adder is

generic (N: integer);

port (A, B: in std_logic_vector (N-1 downto 0);
```

- In addition to the *port map*, the instantiation of a parametric component will include also a *generic map*

```
generic map (N => 10)
```

Combinatorial and Sequential circuits simulation in Vivado

LIVE SHOW

WHAT CAN WE DO WITH A DIGITAL SYSTEM THAT WORKS CORRECTLY IN SIMULATION?

Synthesis

- *Synthesis* is the process of translating a description of a hardware system at higher abstraction level into an (optimized) implementation on a lower abstraction level.
  - From HDL description to an implementation in terms of logic gates
- FPGA manufacturers usually provide appropriate synthesis tool optimized for the target devices.
- Not necessarily a simulated HDL model will be also synthesizable

Synthesis tools

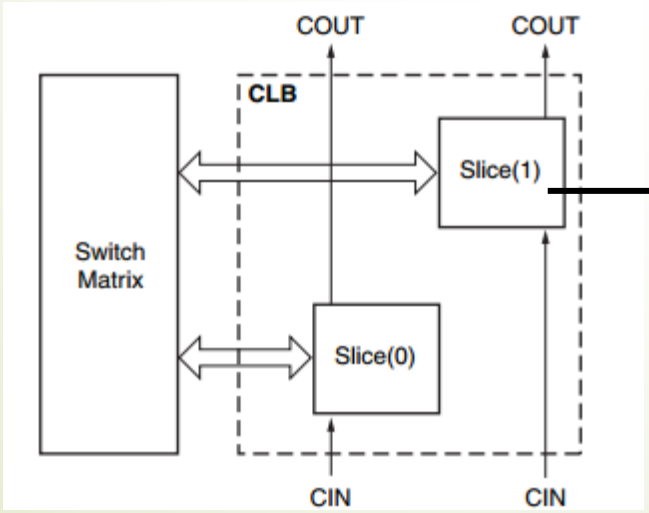
- The synthesis tool must interpret VHDL constructs and translate them into the corresponding circuit implementation
- The fundamental problem is to make sure that the translation process is correct (i.e. obtain the desired circuit)
- A frequent problem example: undesired latches

Constraints

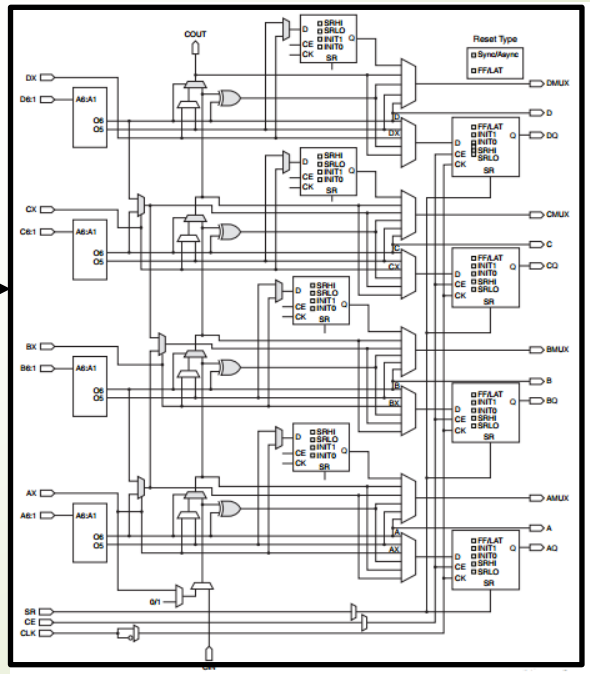
- Besides the model, the synthesis process is based also on *constraints* fixed by the user
- Using the constraints, it is possible to try to obtain a certain behaviour of the synthesis tool, in order to optimize the final result
- Most of the development environments supports the definition of the constraints by using dedicated files
  - Example: Xilinx ISE User Constraints Files (.ucf)

FPGA Internals

- In a 7-series Xilinx FPGA
  - 2 slices per CLB (Configuration Logic Block)
  - 6-input LUTs, eight flip-flops and various multiplexers and arithmetic carry logic form a slice



CLB block diagram



Slice internal components

Tools for FPGA development

- Xilinx ISE
- Intel Altera
- Xilinx Vivado (Homelab FPGA)





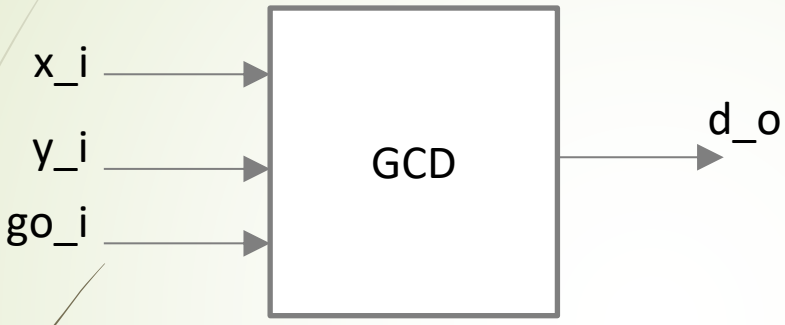








Example: Greatest Common Divider



- Calculus of the greatest common divisor between 2 numbers:
  - The starting point is the algorithm description.

```
int x, y;
while (1)
{
    while (!go_i);
    x = x_i;
    y = y_i;
    while (x != y)
    {
        if (x < y)
            y = y - x;
        else
            x = x - y;
    }
    d_o = x;
}
```

GCD Calculator (Behavioural)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

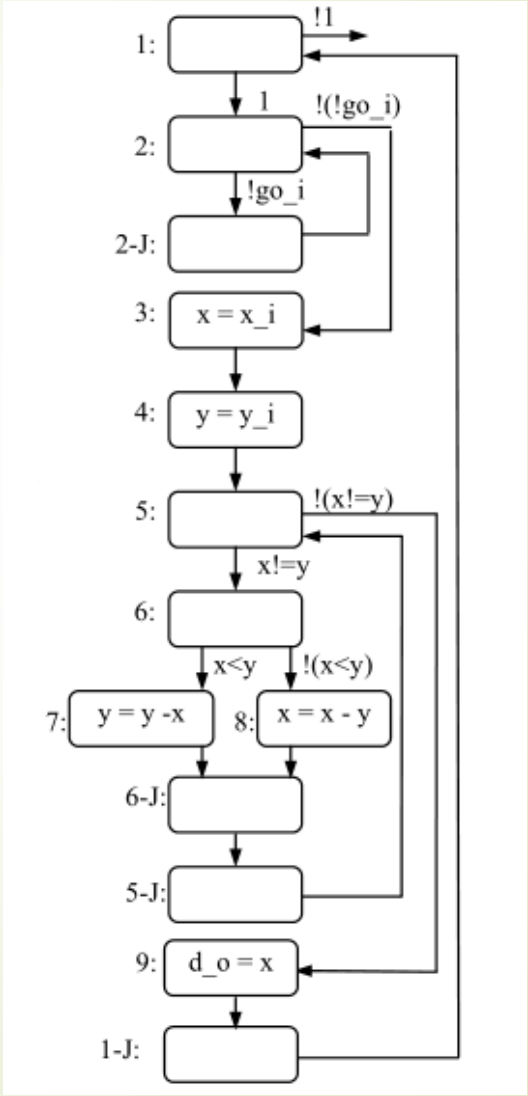
entity gcd1 is
port(   Data_X:     in unsigned(3 downto 0);
        Data_Y:     in unsigned(3 downto 0);
        Data_out:   out unsigned(3 downto 0)
);
end gcd1;
```

GCD Calculator (Behavioural)

```
architecture behav of gcd1 is
begin
    process(Data_X, Data_Y)
    variable tmp_X, tmp_Y: unsigned(3 downto 0);
    begin
    tmp_X := Data_X;
    tmp_Y := Data_Y;
    for i in 0 to 15 loop
        if (tmp_X /= tmp_Y) then
         if (tmp_X < tmp_Y) then
                  tmp_Y := tmp_Y - tmp_X;
         else
              tmp_X := tmp_X - tmp_Y;
         end if;
        else
         Data_out <= tmp_X;
        end if;
    end loop;
   end process;
end behav;
```

GCD Calculator (FSMD)

- The algorithm is converted in a complex “finite state machine”
- FSMD: finite-state machine with data



GCD Calculator (FSMD)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gcd is

port(   clk:    in std_logic;
    rst:    in std_logic;
    go_i:   in std_logic;
    x_i:    in unsigned(3 downto 0);
    y_i:    in unsigned(3 downto 0);
    d_o:    out unsigned(3 downto 0)
);
end gcd;
```

GCD Calculator (FSMD)

```
architecture FSMD of gcd is
begin

    process(rst, clk)

    -- define states using variable
    type S_Type is (ST0, ST1, ST2);
    variable State: S_Type := ST0 ;
    variable Data_X, Data_Y: unsigned(3 downto 0);

    begin


    end process;

end FSMD;
```

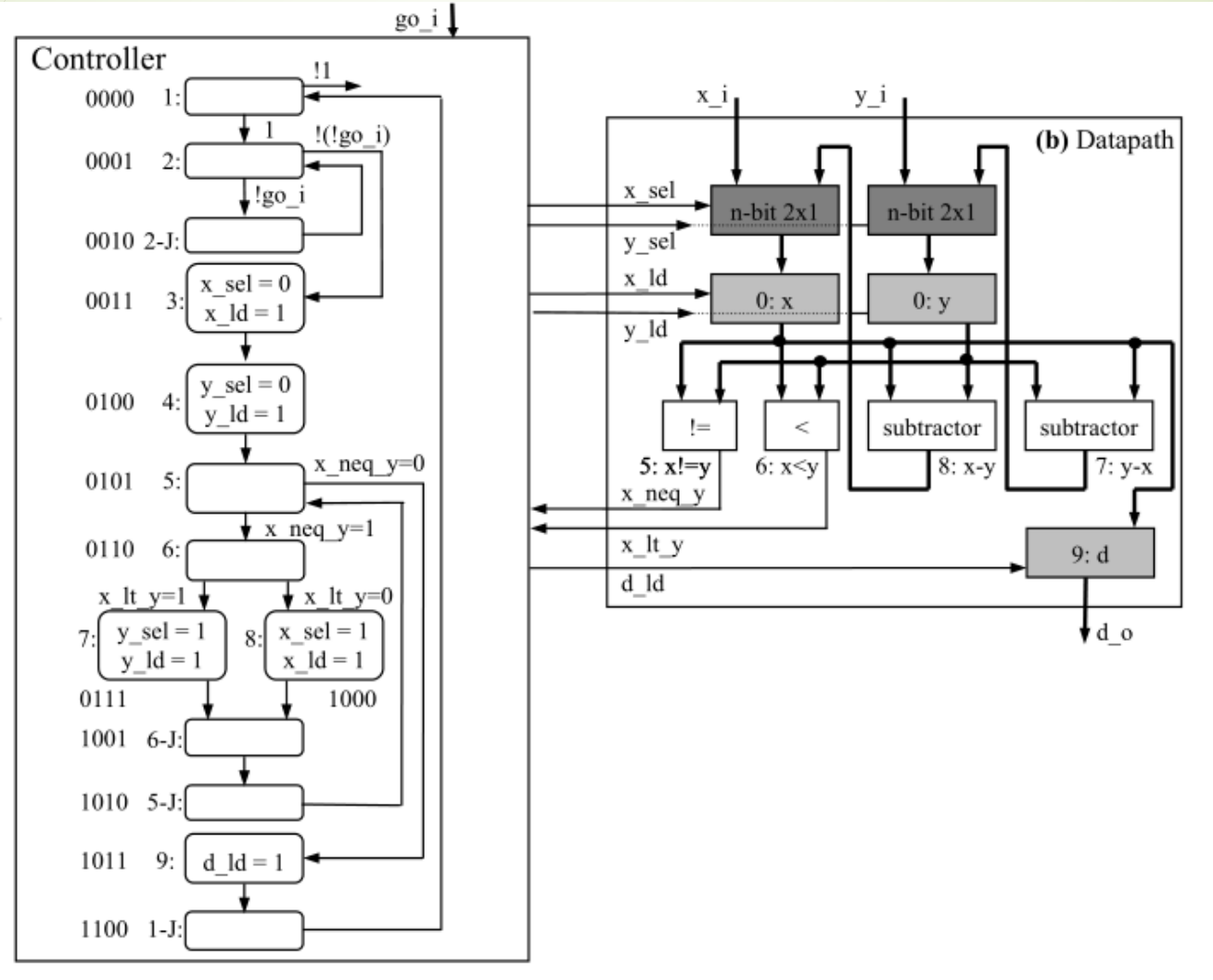
GCD Calculator (FSMD)

```
if (rst='1') then           -- initialization
        d_o <= "0000";
        State := ST0;
    elsif (clk'event and clk='1') then
        case State is
            when ST0 =>              -- starting
            if (go_i='1') then
            Data_X := x_i;
            Data_Y := y_i;
            State := ST1;
            else
                State := ST0;
            end if;
        when ST1 =>          -- idle state
            State := ST2;
```

Example: GCD Calculator (FSMD)

```
        when ST2 =>          -- computation
            if (Data_X/=Data_Y) then
                if (Data_X<Data_Y) then
                Data_Y := Data_Y - Data_X;
            else
                Data_X := Data_X - Data_Y;
            end if;
            State := ST1;
            else
            d_o <=Data_X;          -- done
                State := ST0;
            end if;
        when others =>           -- go back
            d_o <= "ZZZZ";
            State := ST0;
             end case;
    end if;
```

GCD Calculator (FSM + DP)



GCD Calculator (FSM + DP)

MULTIPLEXER (ENTITY)

```
entity mux is
    port(    rst: in std_logic;
             sLine: in std_logic;
             load : in std_logic_vector( 3 downto 0 );
             result: in std_logic_vector( 3 downto 0 );
             output: out std_logic_vector( 3 downto 0 )
         );
    end mux;
```

GCD Calculator (FSM + DP)

MULTIPLEXER (ARCHITECTURE)

```
architecture mux_arc of mux is
begin
    process( rst, sLine, load, result )
    begin
    if( rst = '1' ) then
        output <= "0000";     -- do nothing
    elsif sLine = '0' then
        output <= load;       -- load inputs
    else
        output <= result;     -- load results
    end if;
    end process;
end mux_arc;
```

GCD Calculator (FSM + DP)

COMPARATOR (ENTITY)

```
entity comparator is
    port(    rst: in std_logic;
         x : in std_logic_vector( 3 downto 0 );
         y: in std_logic_vector( 3 downto 0 );
         output: out std_logic_vector( 1 downto 0 )
    );
end comparator;
```

GCD Calculator (FSM + DP)

COMPARATOR (ARCHITECTURE)

```
architecture comparator_arc of comparator is
begin
    process( x, y, rst )
    begin
     if( rst = '1' ) then
         output <= "00";      -- do nothing
     elsif( x > y ) then
         output <= "10";      -- if x greater
     elsif( x < y ) then
         output <= "01";      -- if y greater
     else
         output <= "11";      -- if equivalence.
     end if;
    end process;
end comparator_arc;
```

GCD Calculator (FSM + DP)

SUBTRACTOR (ENTITY)

```
entity subtractor is
    port(  rst: in std_logic;
        cmd: in std_logic_vector( 1 downto 0 );
        x : in std_logic_vector( 3 downto 0 );
        y: in std_logic_vector( 3 downto 0 );
        xout : out std_logic_vector( 3 downto 0 )
        yout: out std_logic_vector( 3 downto 0 )
    );
end subtractor;
```

GCD Calculator (FSM + DP)

SUBTRACTOR (ARCHITECTURE)

```
architecture subtractor_arc of subtractor is
begin
    process( rst, cmd, x, y )
    begin
    if( rst = '1' or cmd = "00" ) then     -- not active.
        xout <= "0000";
        yout <= "0000";
    elsif( cmd = "10" ) then       -- x is greater
        xout <= ( x - y );
        yout <= y;
    elsif( cmd = "01" ) then       -- y is greater
        xout <= x;
        yout <= ( y - x );
    else
        xout <= x;             -- x and y are equal
        yout <= y;
    end if;
    end process;
end subtractor_arc;
```

GCD Calculator (FSM + DP)

REGISTER

```
entity regis is
    port(    rst, clk, load: in std_logic;
             input: in std_logic_vector( 3 downto 0 );
             output: out std_logic_vector( 3 downto 0 )
    );
end regis;

architecture regis_arc of regis is
begin
    process( rst, clk, load, input )
    begin
    if( rst = '1' ) then
        output <= "0000";
    elsif( clk'event and clk = '1') then
        if( load = '1' ) then
            output <= input;
        end if;
        end if;
    end process;
end regis_arc;
```

GCD Calculator (FSM + DP)

GCD (ARCHITECTURE)

```
component fsm is
    port(    rst, clk, proceed: in std_logic;
         comparison: in std_logic_vector( 1 downto 0 );
         enable, xsel, ysel, xld, yld: out std_logic
    );
end component;

component mux is
    port(    rst, sLine: in std_logic;
         load, result: in std_logic_vector( 3 downto 0 );
         output: out std_logic_vector( 3 downto 0 )
    );
end component;

component comparator is
    port(    rst: in std_logic;
         x, y: in std_logic_vector( 3 downto 0 );
         output: out std_logic_vector( 1 downto 0 )
    );
end component;
```

GCD Calculator (FSM + DP)

GCD (ARCHITECTURE)

```
component subtractor is
    port(    rst: in std_logic;
         cmd: in std_logic_vector( 1 downto 0 );
         x, y: in std_logic_vector( 3 downto 0 );
         xout, yout: out std_logic_vector( 3 downto 0 )
    );
end component;

component regis is
    port(    rst, clk, load: in std_logic;
         input: in std_logic_vector( 3 downto 0 );
         output: out std_logic_vector( 3 downto 0 )
    );
end component;

signal xld, yld, xsel, ysel, enable: std_logic;
signal comparison: std_logic_vector( 1 downto 0 );
signal result: std_logic_vector( 3 downto 0 );
signal xsub, ysub, xmux, ymux, xreg, yreg: std_logic_vector( 3 downto 0 );
```

GCD Calculator (FSM + DP)

GCD (ARCHITECTURE)

```
-- doing structure modelling here

-- FSM controller
TOFSM: fsm port map(   rst, clk, go_i, comparison,
                enable, xsel, ysel, xld, yld );
-- Datapath
X_MUX: mux port map( rst, xsel, x_i, xsub, xmux );
Y_MUX: mux port map( rst, ysel, y_i, ysub, ymux );
X_REG: regis port map( rst, clk, xld, xmux, xreg );
Y_REG: regis port map( rst, clk, yld, ymux, yreg );
U_COMP: comparator port map( rst, xreg, yreg, comparison );
X_SUB: subtractor port map( rst, comparison, xreg, yreg, xsub, ysub );
OUT_REG: regis port map( rst, clk, enable, xsub, result );

d_o <= result;
```

GCD Calculator (FSM + DP)

GCD (ARCHITECTURE)

```
-- doing structure m      ling here

-- FSM controll
TOFSM: fsm po                    go

-- Datapath
X_MUX: mux port ma                                    );
Y_MUX: mux port map(                                  ux );
X_REG: regis port map(                                xreg );
Y_REG: regis port ma                                  eg );
U_COMP: comparator                                    mparison );
X_SUB: subtracto                                      yreg, xsub, ysub );
OUT_REG: regi                                         lt );

d_o <= result;
```

BAD DESCRIPTION STYLE !

GCD Calculator (FSM + DP)

GCD (ARCHITECTURE)

```
-- doing structure modelling here

-- FSM controller
TOFSM: fsm port map(rst => rst,
                    clk => clk,
                    proceed => go_i,
                    comparison => comparison,
                    enable => enable,
                    xsel => xsel,
                    ysel => ysel,
                    xld => xld,
                    yld => yld
                    );
```

GCD Calculator (FSM + DP)

GCD (ARCHITECTURE)

```
-- Datapath
X_MUX: mux port map( rst, xsel, x_i, xsub, xmux );
Y_MUX: mux port map( rst, ysel, y_i, ysub, ymux );
X_REG: regis port map( rst, clk, xld, xmux, xreg );
Y_REG: regis port map( rst, clk, yld, ymux, yreg );
U_COMP: comparator port map( rst, xreg, yreg, comparison );
X_SUB: subtractor port map( rst, comparison, xreg, yreg, xsub, ysub );
OUT_REG: regis port map( rst, clk, enable, xsub, result );

d_o <= result;
```

DO THE SCHEMATIC

GCD Calculator Testbench (FSMD)

```
entity test_GCD is
end test_GCD;

architecture Bench of test_GCD is

component gcd
port(   clk:    in std_logic;
    rst:    in std_logic;
    go_i:   in std_logic;
    x_i:    in unsigned(3 downto 0);
    y_i:    in unsigned(3 downto 0);
    d_o:    out unsigned(3 downto 0)
);
end component;

signal T_clk,T_rst,T_go_i: std_logic;
signal T_x_i, T_y_i, T_d_o: unsigned(3 downto 0);
```

GCD Calculator Testbench (FSMD)

```
begin

    U1: GCD port map(T_clk,T_rst,T_go_i,T_x_i,T_y_i,T_d_o);

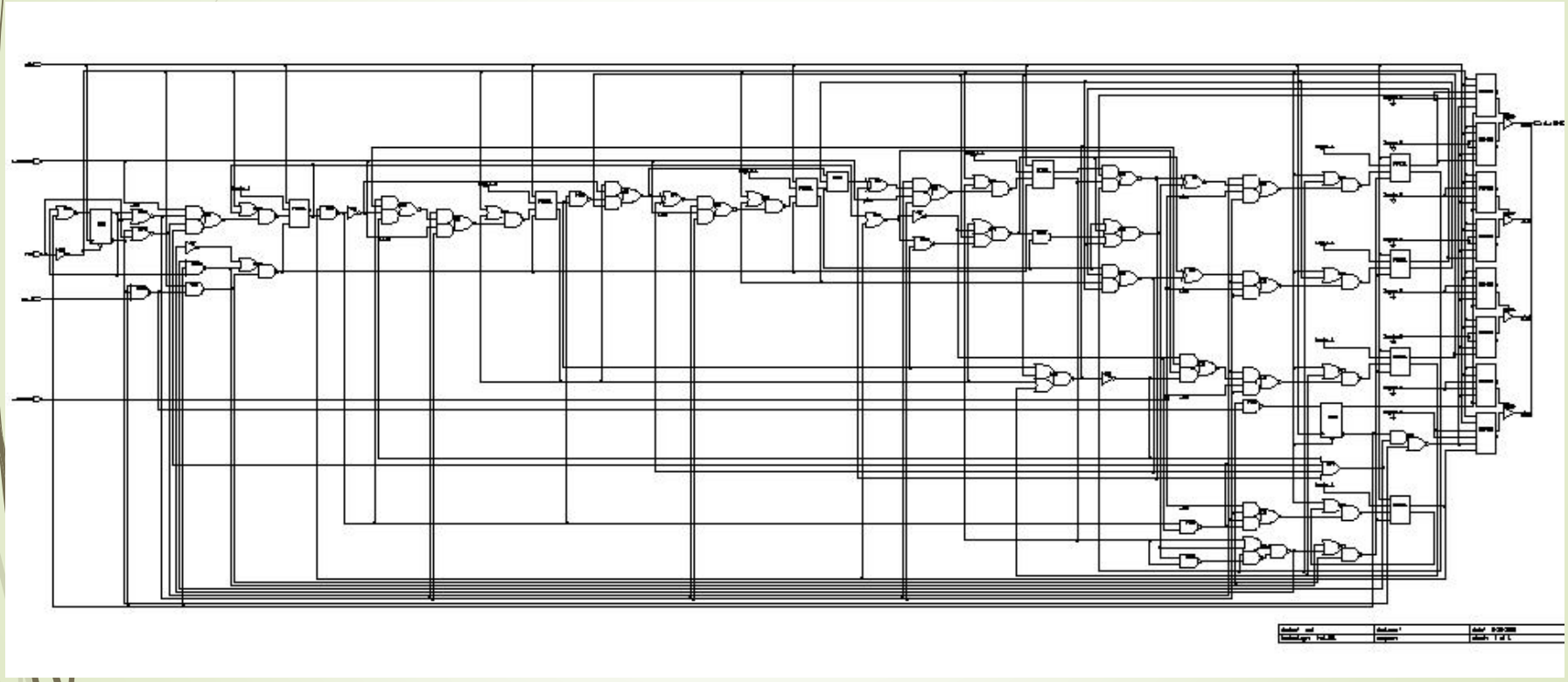
    Clk_sig: process
    begin
        T_clk<='1';
        wait for 5 ns;
        T_clk<='0';
        wait for 5 ns;
    end process;
```

GCD Calculator Testbench – good description style

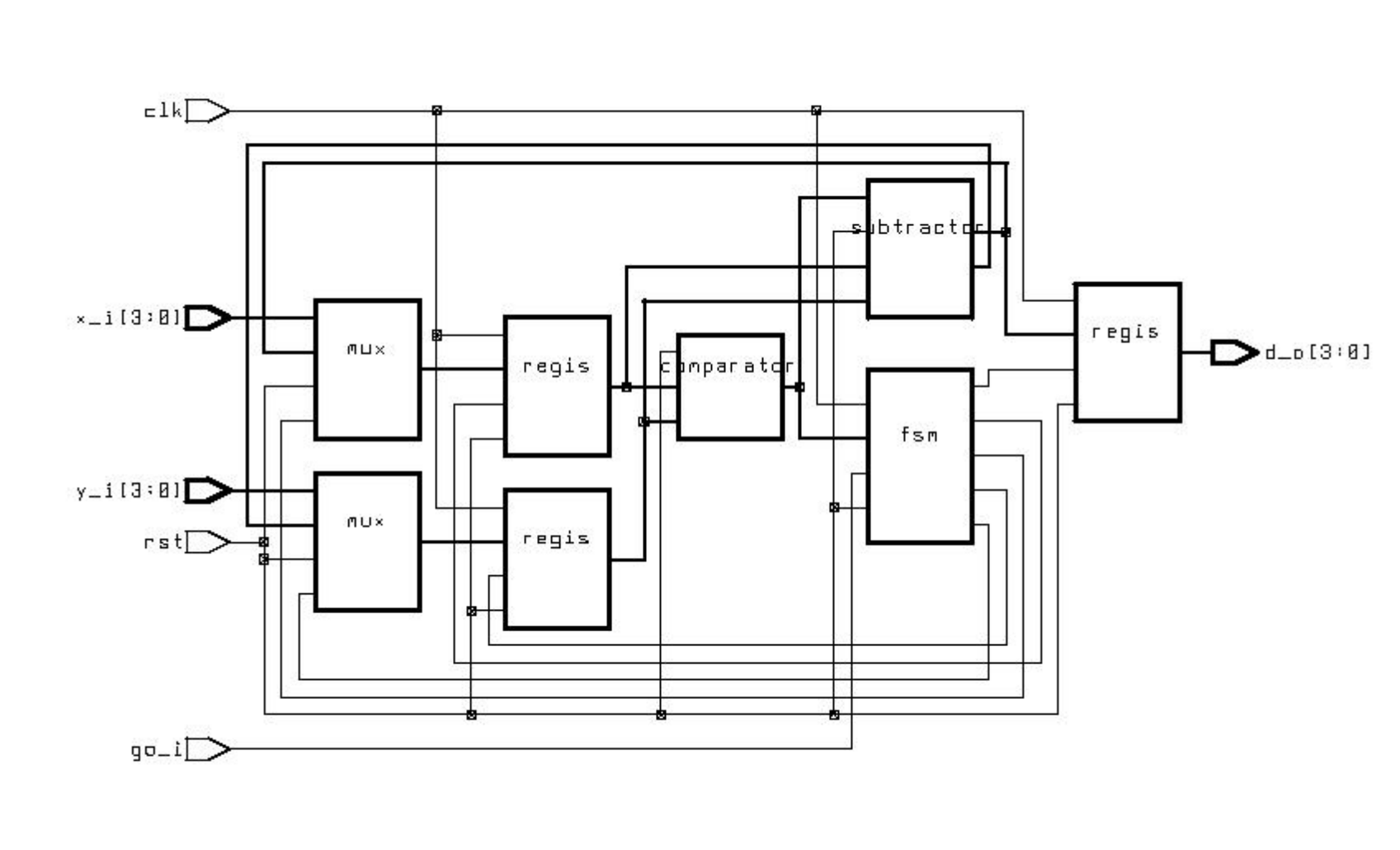
```
begin
    U1: GCD port map
    (clk => T_clk,
     rst => T_rst,
     go_i => T_go_i,
     x_i => T_x_i,
     y_i => T_y_i,
     d_o => T_d_o
     );

    Clk_sig: process
    begin
        T_clk<='1';
        wait for 5 ns;
        T_clk<='0';
        wait for 5 ns;
    end process;
```

Synthesis of GCD (FSMD)

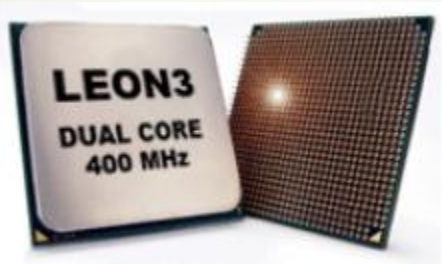


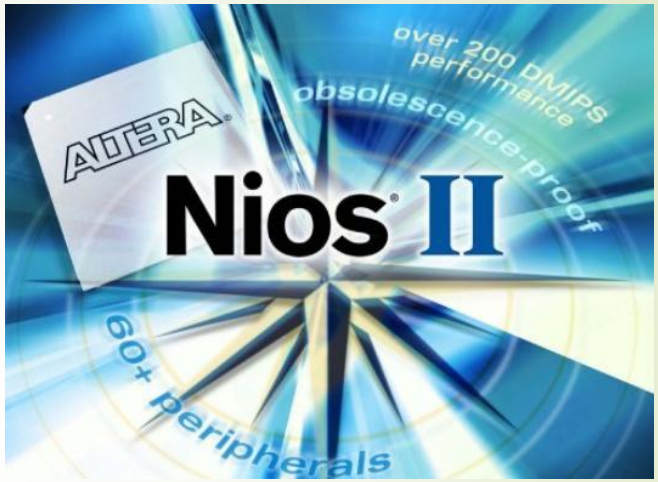
Synthesis of GCD (FSM + DP)



Example: Embedded Processors

LEON3
Synthesizable
processor







Embedded Processors

- Xilinx provides two types of processors for the development of embedded systems:
  - Hard-processor: PowerPC (available in various models of Xilinx FPGAs) and ARM based processors, available on the Zynq-7000 devices
  - Soft-processors: MicroBlaze, PicoBlaze
- PowerPC and MicroBlaze are directly supported within the Embedded Development Kit and Vivado

System Generator

- System Generator is a development environment for FPGA-based digital signal processing architectures
  - Simulink-based tool, supports the use of all Simulink features
  - Large number of DSP blocks immediately available
  - Possibility of exploiting Matlab to generate vectors of stimuli
- System Generator also supports hardware co-simulation
- Intel also provides a similar tool (DSP Builder)

References

- ANSI/IEEE Std 1076
- Frank Vahid and Tony Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, 2002
- Peter J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers, 2008
- Volnei A. Pedroni, *Circuit Design and Simulation with VHDL - 2nd Edition*, MIT Press, 2010
- Pong P. Chu, *RTL Hardware Design using VHDL*, Wiley, 2006
- Pong P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*, Wiley, 2008

Free & open resources

- vhdl.org
- OpenCores: open source hardware resource
- Vendors websites: Xilinx (forum recommended), Altera
- GHDL: GPL VHDL Simulator
- ActiveHDL: simulator and various tutorials
- Cobham Gaisler: Leon 3 processor
- SIGASI Insights: VHDL's crown jewel
- SIGASI Insights: Verilog's major flaw

Thank you !

Contact: giacomo.valente@univaq.it