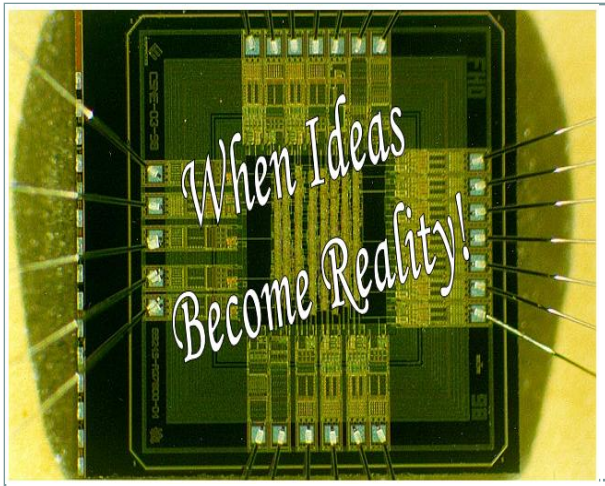
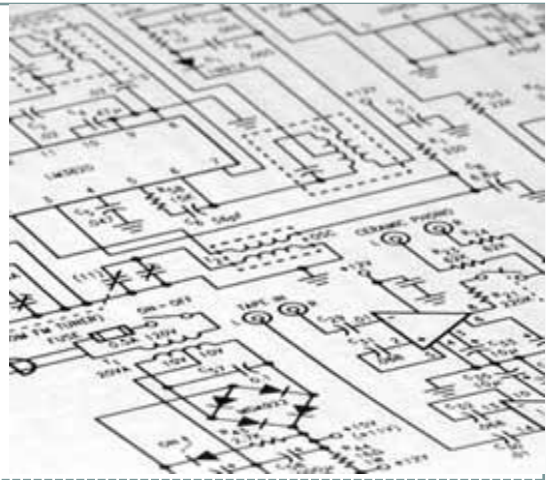


VHDL



Very High Speed Integrated Circuit
Hardware
Description
Language

CHAPTER TEN

DESIGN PROCESS

```
 1 LIBRARY IEEE;
 2 USE IEEE.std_logic_1164.ALL;
 3 ENTITY  comparison IS
 4 PORT(   a,b      :IN     std_logic;
 5         c        :OUT    std_logic);
 6 END comparison;
 7
 8 ARCHITECTURE behaviour OF comparison IS
 9 BEGIN
10 c<='1' WHEN a>b ELSE '0';
11
12 END behaviour;
```

Fig a Behavioural structure

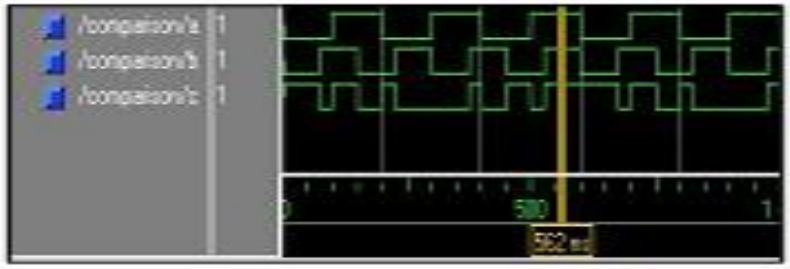


Fig b Simulation Result

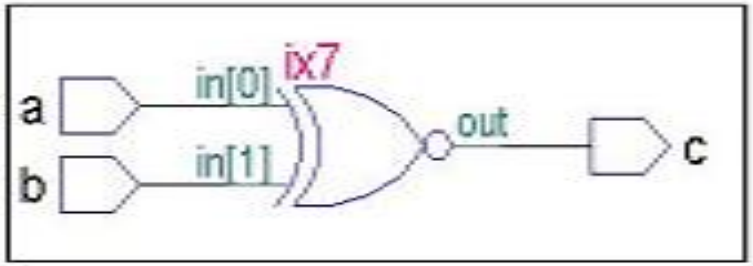


Fig c Synthesis Result

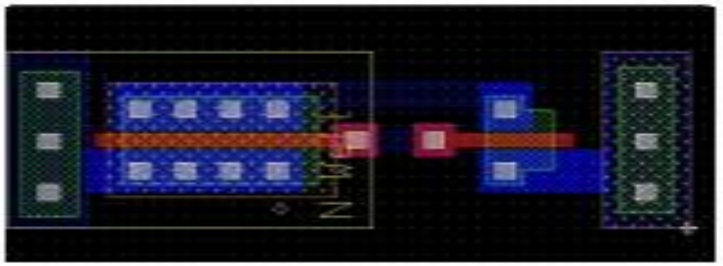


Fig d Layout Result

Figure 2-3 Four phases Process: a-Behaviour, b-Simulation, c-Synthesis and d-Layout

Introduction

- VHDL is the acronym of VHSIC Hardware Description Language. VHSIC is an abbreviation for Very High Speed Integrated Circuit. It can describe the behavior and structure of electronic systems, but is particularly suited as a language to describe the structure and behavior of digital electronic hardware designs, such as ASICs and FPGAs as well as conventional digital circuits.

- Developed by Department of Defense (DOD) between 1970s and 80s, it was officially standardized as IEEE 1076 in 1987.

Levels of representation and abstraction

- A digital system can be represented at different levels of abstraction. This keeps the description and design of complex systems manageable. Figure below shows different levels of abstraction.

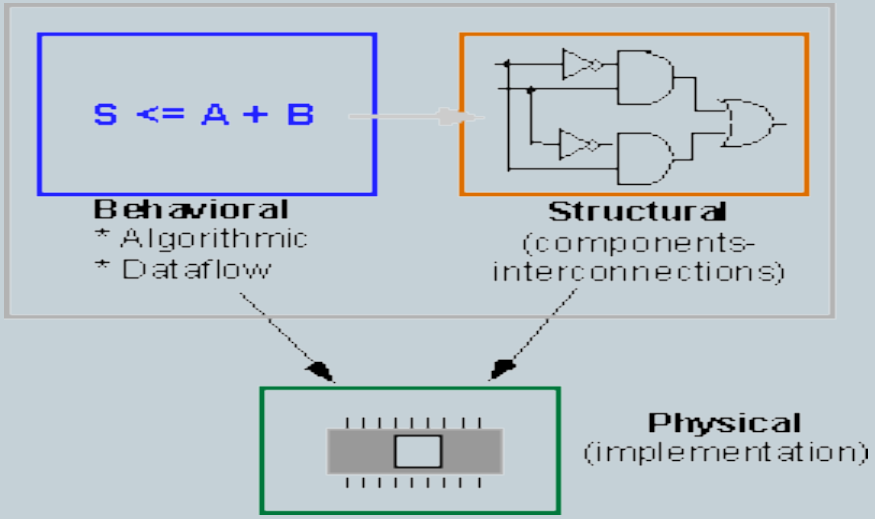


Figure 1 : Levels of abstraction: Behavioral, Structural and Physical

Contd…

- **Behavioral:** The highest level of abstraction that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. This could be a Boolean expression or a more abstract description such as the Register Transfer or Algorithmic level.
- Example, let us consider a simple circuit that warns car passengers when the door is open or the seatbelt is not used whenever the car key is inserted in the ignition lock. At the behavioral level this could be expressed as,

Warning = Ignition_on AND (Door_open OR Seatbelt_off)

Contd…

- **Structural**: The structural level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is usually closer to the physical realization of a system. For the example above, the structural representation is shown in Figure below.

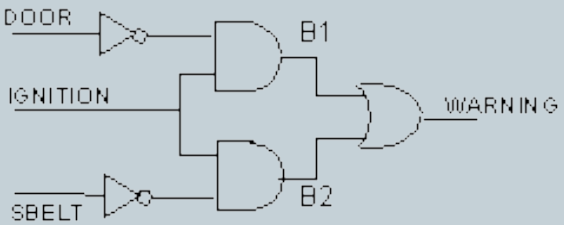


Figure 2: Structural representation of a “buzzer” circuit.

Basic Structure of a VHDL file

- A digital system in VHDL consists of a design **entity** that can contain other entities that are then considered components of the top-level entity. Each entity is modeled by an *entity declaration* and an *architecture body*.

- One can consider the entity declaration as the interface to the outside world that defines the input and output signals, while the architecture body contains the description of the entity and is composed of interconnected entities, processes and components, all operating concurrently, as schematically shown in Figure below. In a typical design there will be many such entities connected together to perform the desired function.

Contd...

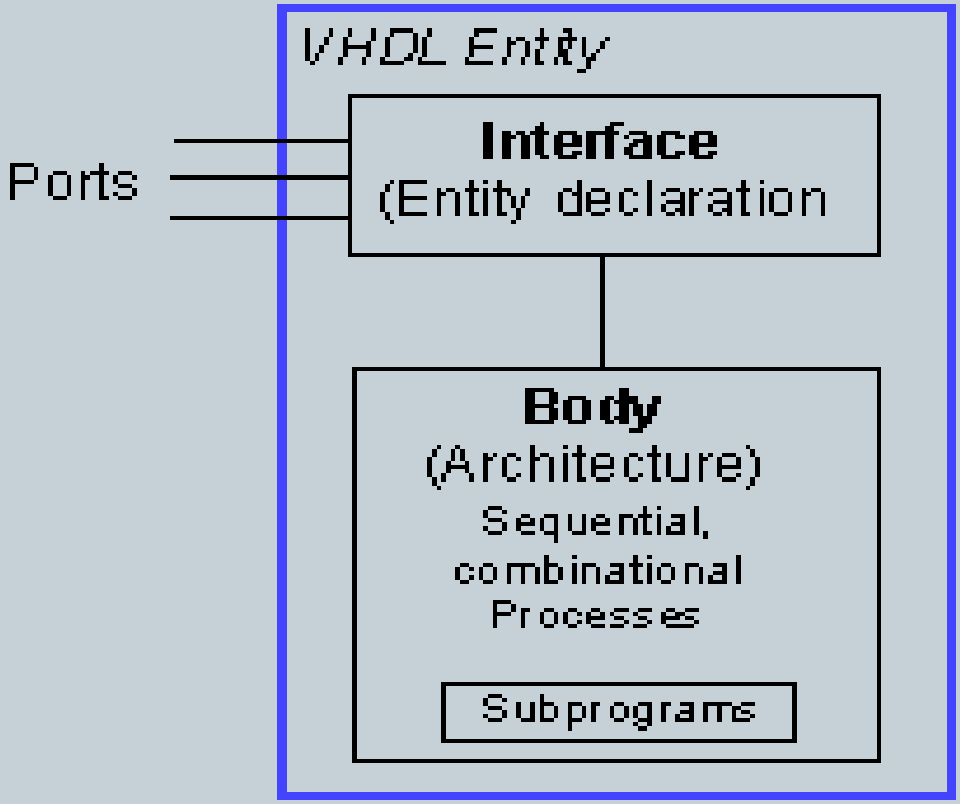


Figure 3: A VHDL entity consisting of an interface (entity declaration) and a body (architectural description).

Contd…

- VHDL uses reserved **keyword**s that cannot be used as signal names or identifiers.
- Keywords and user-defined identifiers are **case insensitive**.
- Lines with comments start with two adjacent hyphens (--) and will be ignored by the compiler.
- VHDL also ignores line breaks and extra spaces.
- VHDL is **a strongly typed** language which implies that one has always to declare the type of every object that can have a value, such as signals, constants and variables.

Contd…

Entity Declaration:

- The entity declaration defines the NAME of the entity and lists the input and output ports. The general form is as follows,

entity NAME_OF_ENTITY **is**
port (*signal_names*: **mode** *type*;
signal_names: **mode** *type*;
:
signal_names: **mode** *type*);
end [NAME_OF_ENTITY] ;

Contd…

- An entity always starts with the keyword **entity**, followed by its name and the keyword **is**. Next are the port declarations using the keyword **port**. An entity declaration always ends with the keyword **end**, optionally [] followed by the name of the entity.
- The NAME_OF_ENTITY is a user-selected identifier.
- signal_names consists of a comma separated list of one or more user-selected identifiers that specify external interface signals.

Contd…

mode: is one of the reserved words to indicate the signal direction:

- **in** – indicates that the signal is an input
- **out** – indicates that the signal is an output of the entity whose value can only be read by other entities that use it.
- **buffer** – indicates that the signal is an output of the entity whose value can be read inside the entity's architecture
- **inout** – the signal can be an input or an output.

Contd…

type: a built-in or user-defined signal type. Examples of types are bit, bit_vector, Boolean, character, std_logic, and std_ulogic.

- *bit* – can have the value 0 and 1
- *bit_vector* – is a vector of bit values (e.g. bit_vector (0 to 7)
- *std_logic, std_ulogic, std_logic_vector, std_ulogic_vector*: can have 9 values to indicate the value and strength of a signal. Std_ulogic and std_logic are preferred over the bit or bit_vector types.
- *boolean* – can have the value TRUE and FALSE
- *integer* – can have a range of integer values
- *real* – can have a range of real values
- *character* – any printing character
- *time* – to indicate time

Contd…

For the example of Figure 2 above, the entity declaration looks as follows.

```
-- comments: example of the buzzer circuit of fig. 2
   entity BUZZER is
         port(DOOR,IGNITION,SBELT: in std_logic;
                  WARNING: out std_logic);
    end BUZZER;
```

Note: try out for Mux 4 to 1, D-flipflop

Contd...

Architecture body

- The architecture body specifies how the circuit operates and how it is implemented.

  The architecture body looks as follows,

```
architecture architecture_name of NAME_OF_ENTITY is
        -- Declarations
        -- components declarations
        -- signal declarations
        -- constant declarations
        -- type declarations
begin
 -- Statements
:
end architecture_name;
```

Contd…

Behavioral model

- The architecture body for the example of Figure 2, described at the behavioral level, is given below,

architecture behavioral **of** BUZZER **is**

begin

WARNING <= (not DOOR and IGNITION) or (not SBELT and IGNITION);

end behavioral;

Contd…

- The header line of the architecture body defines the architecture name, e.g. behavioral, and associates it with the entity, BUZZER.
- The architecture name can be any legal identifier.
- The main body of the architecture starts with the keyword **begin**.
- The "<= " symbol represents an assignment operator and assigns the value of the expression on the right to the signal on the left.
- The architecture body ends with an **end** keyword followed by the architecture name.

Note: Try out for Basic Gates Like AND, OR, NOT

Complete program looks like


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

entity BUZZER is
        port(DOOR,IGNITION,SBELT: in std_logic;
            WARNING: out std_logic);
end BUZZER;

architecture behavioral of BUZZER is
    begin

   WARNING <= (not DOOR and IGNITION) or (not SBELT and IGNITION);

end behavioral;
```

Concurrency

Contd…

Structural description

- The circuit of Figure 2 can also be described using a structural model that specifies what gates are used and how they are interconnected. The following example illustrates it.

```
architecture structural of BUZZER is
    -- Declarations
    component AND2
        port (in1, in2: in std_logic;
              out1: out std_logic);
    end component;
```

Contd…


```
component OR2
    port (in1, in2: in std_logic;
          out1: out std_logic);
end component;

component NOT1
    port (in1: in std_logic;
          out1: out std_logic);
end component;

-- declaration of signals used to interconnect gates
signal DOOR_NOT, SBELT_NOT, B1, B2: std_logic;
```

Contd…


```
Begin
    -- Component instantiations statements
    U0: NOT1 port map (DOOR -> DOOR_NOT);
    U1: NOT1 port map (SBELT -> SBELT_NOT);
    U2: AND2 port map (IGNITION, DOOR_NOT, B1);
    U3: AND2 port map (IGNITION, SBELT_NOT, B2);
    U4: OR2  port map (B1, B2, WARNING);

end structural;
```

Contd…

- Following the header is the *declarative* part that gives the **components** (gates) that are going to be used in the description of the circuits.
- In our example, we use a two- input AND gate, two-input OR gate and an inverter. These gates have to be defined first.
- The *statements* after the **begin** keyword gives the instantiations of the components and describes how these are interconnected.

LABEL: COMPONENT-NAME **PORT MAP** (*PORT1=>SIGNAL1, PORT2=> SIGNAL2,… PORT3=>SIGNALN*);

Library and Package

- A library can be considered as a place where the compiler stores information about a design project. A VHDL package is a file or module that contains declarations of commonly used objects, data type, component declarations, signal, procedures and functions that can be shared among different VHDL models.

- std_logic is defined in the package ieee.std_logic_1164 in the ieee library. In order to use the std_logic one needs to specify the library and package.

Contd…

- This is done at the beginning of the VHDL file using the **library** and the **use** keywords as follows:

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

- The **.all** extension indicates to use all of the ieee.std_logic_1164 package.

Contd…

- The Xilinx Foundation Express comes with several packages.

  ieee Library:

  - std_logic_1164 package: defines the standard data types
  - std_logic_arith package: provides arithmetic, conversion and comparison functions for the signed, unsigned, integer, std_ulogic, std_logic and std_logic_vector types
  - std_logic_misc package: defines supplemental types, subtypes, constants and functions for the std_logic_1164 package.

To use any of these one must include the library and use clause:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

Lexical Elements of VHDL

Identifiers

- *Identifiers* are user-defined words used to name objects in VHDL modules. We have seen examples of identifiers for input and output signals as well as the name of a design entity and architecture body.

- When choosing an identifier one needs to follow these basic rules:
  - May contain only alpha-numeric characters (A to Z, a to z, 0-9) and the underscore (_) character.
  - The first character must be a letter and the last one cannot be an underscore.
  - An identifier cannot include two consecutive underscores.
  - An identifier is case insensitive (ex. And2 and AND2 or and2 refer to the same object)
  - An identifier can be of any length.

Contd...

Keywords (Reserved words)

- Certain identifiers are used by the system as keywords for special use such as specific constructs. These keywords cannot be used as identifiers for signals or objects we define.

access	exit	mod	return	while
after	file	new	signal	with
alias	for	next	shared	
all	function	null	then	
attribute	generic	of	to	
block	group	on	type	
body	in	open	until	
buffer	is	out	use	
bus	label	range	variable	
constant	loop	rem	wait	

Contd…

Numbers

- The default number representation is the decimal system. VHDL allows integer literals and real literals. Integer literals consist of whole numbers without a decimal point, while real literals always include a decimal point. Exponential notation is allowed using the letter “E” or “e”. For integer literals the exponent must always be positive. Examples are:

Integer literals: 12 10 -100

Real literals: 1.2 256.24 3.14E-2

Contd…

Characters, Strings and Bit Strings

- Single quotation mark is used to represent character literal in VHDL, as shown in the examples below:

  ‘a’, ‘B’, ‘,’

- A string of characters are placed in double quotation marks as shown:

  “This is a string”,

- A bit-string represents a sequence of bit values. In order to indicate a bit string, one places the ‘B’ in front of the string: B”1001”. One can also use strings in the hexagonal or octal base by using the X or O specifiers, respectively. Some examples are:

  Binary: B”1100_1001”, b”1001011”

  Hexagonal: X”C9”, X”4b”

  Octal: O”311”, o”113”

Data Objects: Signals, Variables, Constants

Constant

- A constant can have a single value of a given type and cannot be changed during the simulation. A constant is declared as follows,

  constant *list_of_name_of_constant*: type [:= initial value] ;

- where the initial value is optional. Constants can be declared at the start of an architecture and can then be used anywhere within the architecture. Constants declared within a process can only be used inside that specific process.

  constant RISE_FALL_TME: time := 2 ns;
  constant DELAY1: time := 4 ns;
  constant RISE_TIME, FALL_TIME: time:= 1 ns;
  constant DATA_BUS: integer:= 16;

Contd…

Variable

- A variable may be changed during program execution. Variable value is updated using a variable assignment statement. The variable is updated without any delay as soon as the statement is executed. Variables must be declared *inside* a process (and are local to the process). The variable declaration is as follows:

  variable *list_of_variable_names*: type [:= initial value] ;

  A few examples follow:

  variable CNTR_BIT: bit :=0;
  variable VAR1: boolean :=FALSE;
  variable SUM: integer **range** 0 **to** 256 :=16;
  variable STS_BIT: std_logic_vector (7 **downto** 0);

Contd…

Signal

- Signals are similar to wires on a schematic, and can be used to interconnect concurrent elements of the design.

signal *list_of_signal_names*: type [:= initial value] ;

signal SUM, CARRY: std_logic;
signal CLOCK: bit;
signal TRIGGER: integer :=0;
signal DATA_BUS: std_logic_vector (7 downto 0);
signal VALUE: integer **range** 0 **to** 100;

- Signals are updated when their signal assignment statement is executed, *after a certain delay*.

Data Types

- Each data object has a type associated with it. The type defines the set of values that the object can have and the set of operations that are allowed on it.

- It is not allowed to assign a value of one type to an object of another data type (e.g. assigning an integer to a bit type is not allowed).

- Data Types defined in the Standard Package

  VHDL has several predefined types in the *standard* package as shown in the table below:

Contd…

Types defined in the Package *Standard* of the *std* Library		
Type	**Range of values**	**Example**
Bit	'0', '1'	signal A: bit :=1;
bit_vector	an array with each element of type bit	signal INBUS: bit_vector(7 downto 0);
boolean	FALSE, TRUE	variable TEST: Boolean :=FALSE'
character	any legal VHDL character (see package standard); printable characters must be placed between single quotes (e.g. '#')	variable VAL: character :='$';
file_open_kind*	read_mode, write_mode, append_mode	
file_open_status*	open_ok, status_error, name_error, mode_error	
integer	range is implementation dependent but includes at least $-(2^{31}-1)$ to $+(2^{31}-1)$	constant CONST1: integer :=129;
natural	integer starting with 0 up to the max specified in the implementation	variable VAR1: natural :=2;
positive	integer starting from 1 up the max specified in the implementation	variable VAR2: positive :=2;
real*	floating point number in the range of -1.0×10^{38} to $+1.0 \times 10^{38}$ (can be implementation dependent. *Not supported by the Foundation synthesis program.*	variable VAR3: real :=+64.2E12;
severity_level	note, warning, error, failure	
String	array of which each element is of the type character	variable VAR4: string(1 to 12):= "@$#ABC*()_%2";
time*	an integer number of which the range is implementation defined; units can be expressed in sec, ms, us, ns, ps, fs, min and hr. . *Not supported by the Foundation synthesis program*	variable DELAY: time :=5 ns;

* Not supported by the Foundation synthesis program

Type Conversion

- Since VHDL is a strongly typed language one cannot assign a value of one data type to a signal of a different data type.

Conversions supported by std_logic_1164 package	
Conversion	**Function**
std_ulogic to bit	to_bit(*expression*)
std_logic_vector to bit_vector	to_bitvector(*expression*)
std_ulogic_vector to bit_vector	to_bitvector(*expression*)
bit to std_ulogic	To_StdULogic(*expression*)
bit_vector to std_logic_vector	To_StdLogicVector(*expression*)
bit_vector to std_ulogic_vector	To_StdUlogicVector(*expression*)
std_ulogic to std_logic_vector	To_StdLogicVector(*expression*)
std_logic to std_ulogic_vector	To_StdUlogicVector(*expression*)

Operators

- VHDL supports different classes of operators that operate on signals, variables and constants. The different classes of operators are summarized below.

Class						
1. Logical operators	and	or	nand	nor	xor	xnor
2. Relational operators	=	/=	<	<=	>	>=
3. Shift operators	sll	srl	sla	sra	rol	ror
4.Addition operators	+	=	&			
5. Unary operators	+	-				
6. Multiplying op.	*	/	mod	rem		
7. Miscellaneous op.	**	abs	not			

Sequential Statements

Process

- A PROCESS is a sequential section of VHDL code. It is characterized by the presence of IF, WAIT, CASE, LOOP and a sensitivity list(except when WAIT is used). Process is executed every time a signal in the sensitivity list changes(or the condition related to WAIT is fulfilled). Its syntax is shown below:

[label:] **PROCESS** (sensitivity list)
  [VARIABLE name type [range] [:=initial_value]]
BEGIN
    (sequential statements)
END PROCESS

An example of a positive edge-triggered D flip-flop


```
library ieee;
use ieee.std_logic_1164.all;
entity DFF_CLEAR is
    port (CLK, CLEAR, D : in std_logic;
        Q : out std_logic);
end DFF_CLEAR;

architecture BEHAV_DFF of DFF_CLEAR is
begin
DFF_PROCESS: process (CLK, CLEAR)
    begin
        if (CLEAR = '1') then
            Q <= '0';
        elsif (CLK'event and CLK = '1') then
            Q <= D;
        end if;
    end process;
end BEHAV_DFF;
```

If- statements

The if statement executes a sequence of statements whose sequence depends on one or more conditions. The syntax is as follows:

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

Example:

```
if S1='0' and  S0='0' then
   Z <= A;
elsif S1='0' and  S0='1' then
   Z <= B;
Else  Z <= C;
end if;
```

Case Statements

- The case statement executes one of several sequences of statements, based on the value of a single expression. The syntax is as follows,

case *expression* **is**
when *choices* =>
sequential statements
when *choices* =>
sequential statements
-- branches are allowed
when others => *sequential statements*]
end case;

Contd…

Example:

```
case VALUE is
        when 51 to 60 =>
           D <= '1';
        when 61 to 70 | 71 to 75 =>
           C <= '1';
        when 76 to 85 =>
           B <= '1';
         when 86 to 100 =>
           A <= '1';
         when others  =>
           F <= '1';
end case;
```

Finite State Machine (FSM)

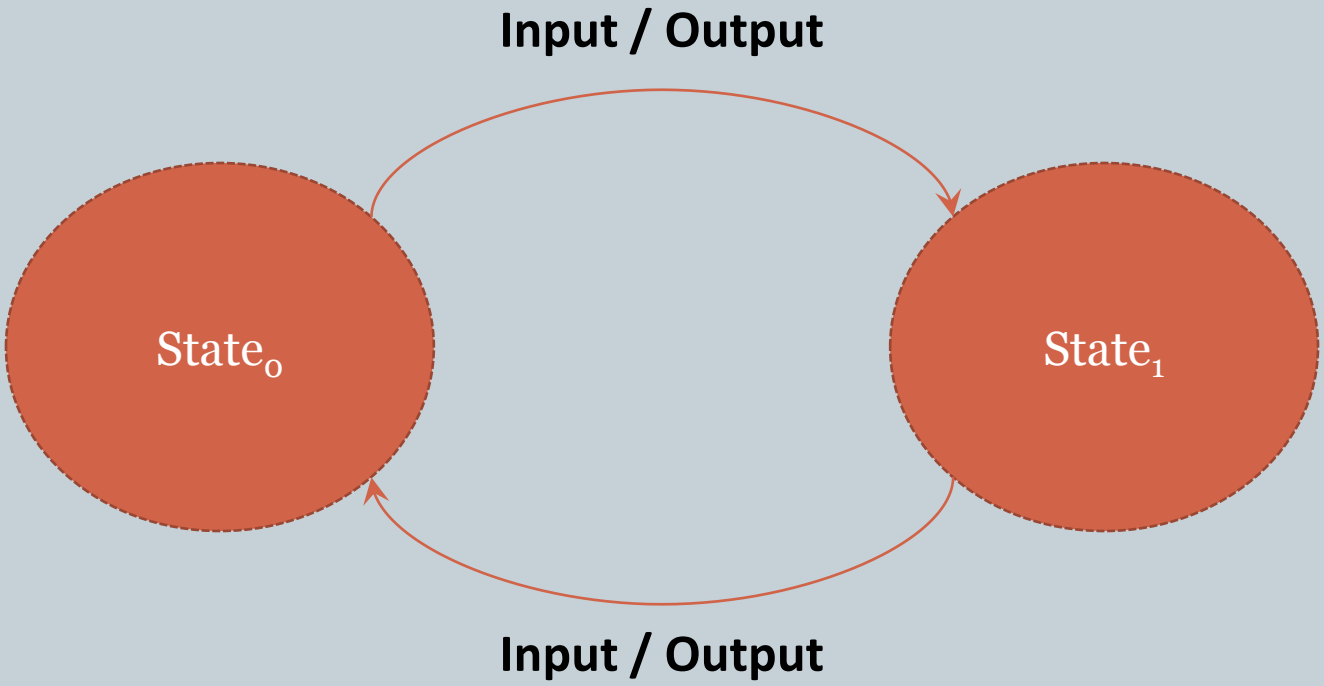
- A sequential logic unit which
  - Takes an input and a current state
  - Produces an output and a new state

- It is called a Finite State Machine because it can have, at most, a finite number of states.

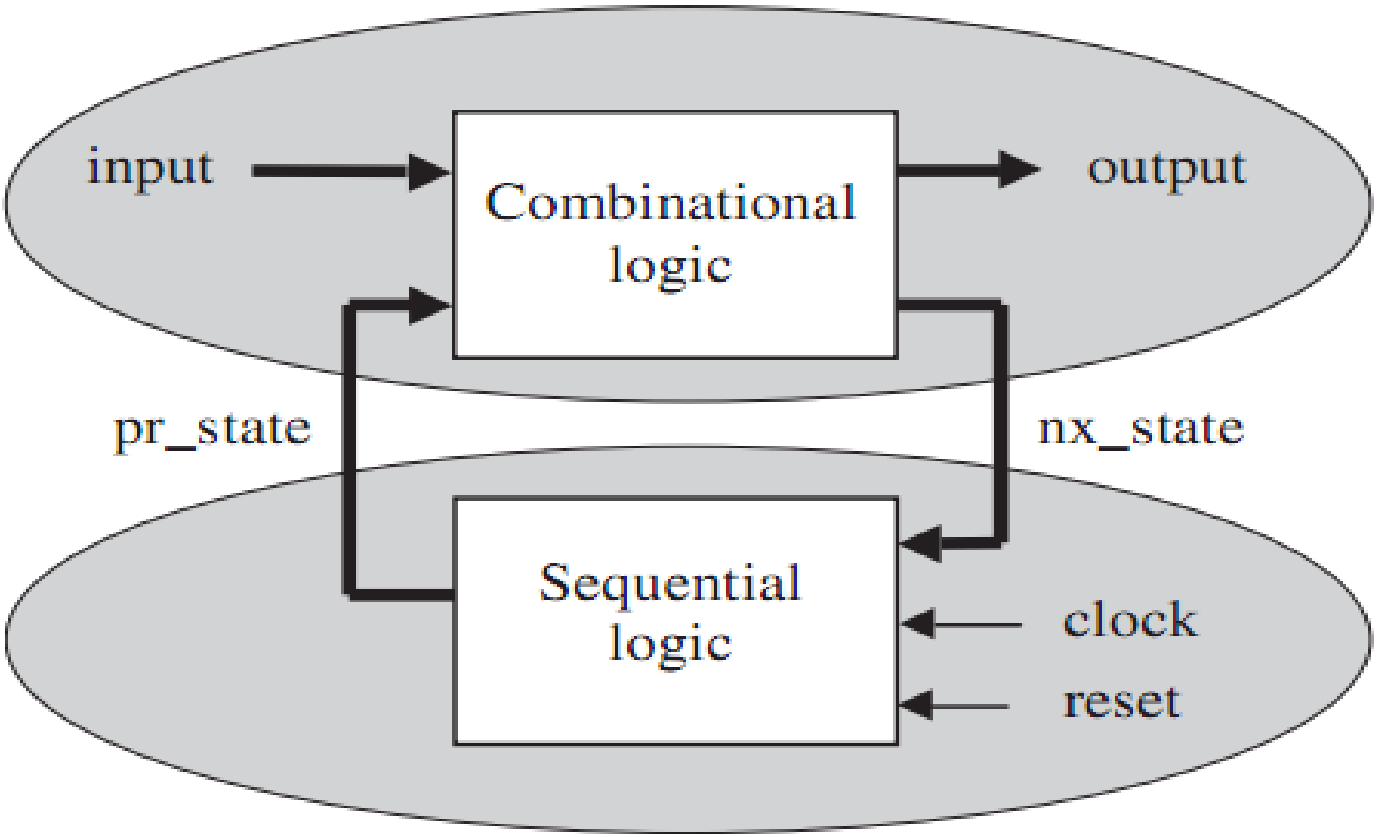
- It is composed of a combinational logic unit and flip-flops placed in such a way as to maintain state information.

Contd...

- It can also be represented using a **state diagram** as below.



FSM Diagram


input
Combinational logic
output
pr_state
nx_state
Sequential logic
clock
reset

Finite State Machine Design

- Design a custom processor that calculates Greatest Common Divisor (GCD) using FSM.

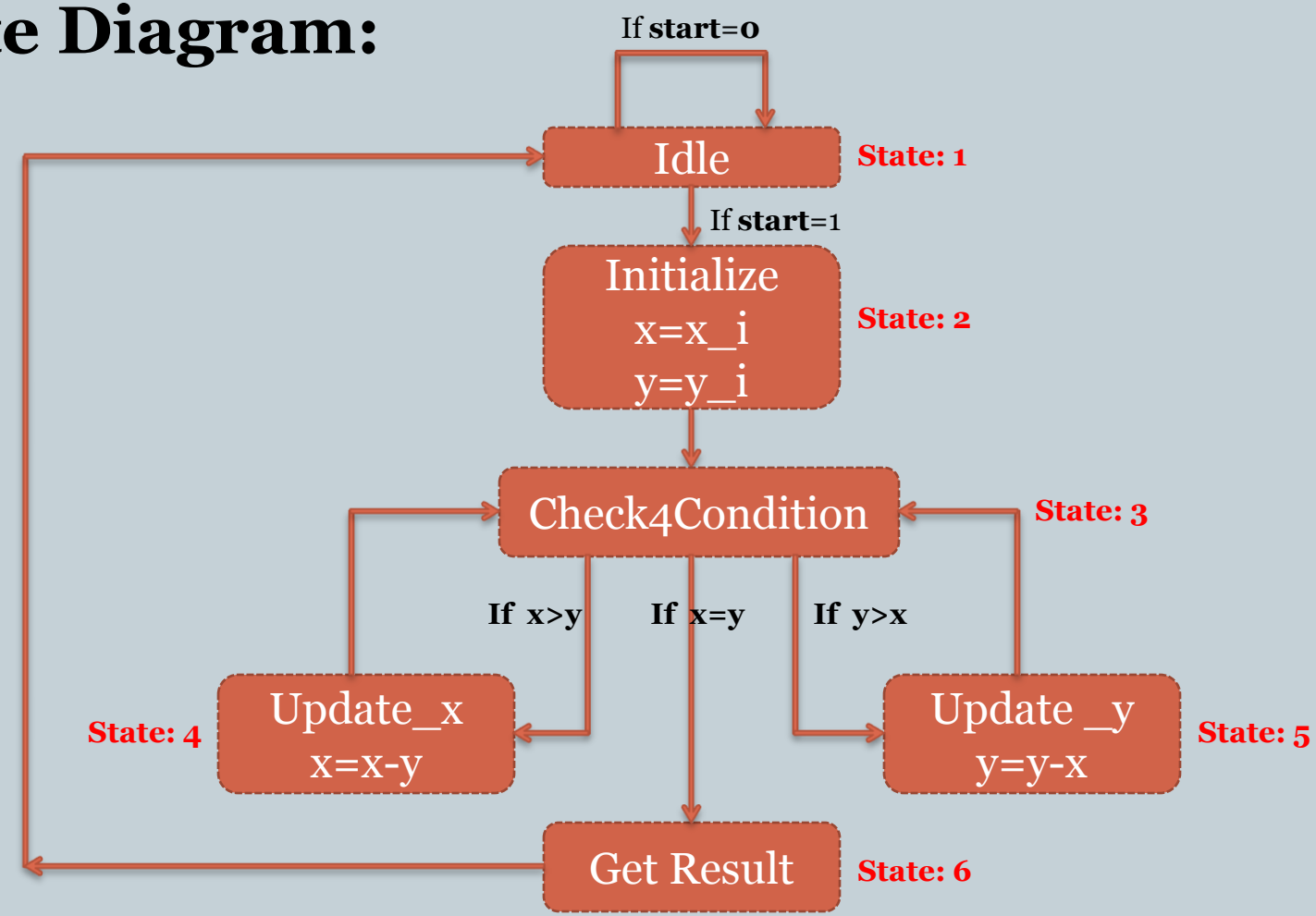
- **STEPS:**

Algorithm:

```
0:  int  x, y;
1:  while (1) {
2:        while (!go_i);
3:              x = x_i;
4:              y = y_i;
5:              while  (x != y)  {
6:                    if  (x < y)
7:                          y = y - x;
                      else
8:                          x = x - y;
                    }
9:    d_o = x;
      }
```

Contd...

State Diagram:



Contd…

VHDL Coding:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;


entity gcd is
   port( clk :      in  std_logic;
         reset:     in  std_logic;
         num_1:     in  unsigned(3 downto 0);
         num_2:     in  unsigned(3 downto 0);
         gcd_num:   out unsigned(3 downto 0)
   );
end entity;
```

Contd...


```
architecture behav of gcd is
   type state is (idle, init, check, update_x, update_y, get_result );
   signal pr_state:        state:=idle;
   signal nx_state:        state;

   signal flag:      std_logic:='0';
   signal start:     std_logic:='1';

   begin

      --Sequential Section
       sequential:process(clk,reset) is
       begin
          if(reset='1') then
             pr_state<=idle;
          elsif(clk'event and clk='1') then
             pr_state<=nx_state;
          end if;
       end process sequential;
```

Contd...


```
--Combinational Section
        combinational:process(pr_state,num_1,num_2) is

        variable temp_x:    unsigned(3 downto 0):=(others=>'0');
        variable temp_y:    unsigned(3 downto 0):=(others=>'0');
        begin
            case pr_state is
                when idle=>
                        if(start='1') then
                            nx_state<=init;
                            start<='0';
                         else
                            nx_state<=idle;
                        end if;

                when init=>
                        temp_x:=num_1;
                        temp_y:=num_2;
                        nx_state<=check;
```

Contd…


```
when check=>
      if(temp_x=temp_y) then
            nx_state<=get_result;
      elsif(temp_x>temp_y) then
            nx_state<=update_x;
      else
            nx_state<=update_y;
      end if;

 when update_x=>
      temp_x:=temp_x-temp_y;
      nx_state<=check;

 when update_y=>
      temp_y:=temp_y-temp_x;
      nx_state<=check;
```

Contd…


```
        when get_result=>
            gcd_num<=temp_x;
            nx_state<=idle;
             start<='1';
    end case;
    end process combinational;
end architecture;
```

Contd…

Testbench:

```
LIBRARY ieee  ;
USE ieee.std_logic_1164.all  ;
USE ieee.std_logic_arith.all  ;
USE ieee.std_logic_unsigned.all  ;

ENTITY gcd_tb  IS
END ;

ARCHITECTURE gcd_tb_arch OF gcd_tb IS
  SIGNAL num_1   :  unsigned (3 downto 0)  ;
  SIGNAL gcd_num   :  unsigned (3 downto 0)  ;
  SIGNAL num_2   :  unsigned (3 downto 0)  ;
  SIGNAL clk   :  std_logic  ;
  SIGNAL reset   :  std_logic  ;
  constant clk_period : time := 10 ns;
```

Contd…


```
COMPONENT gcd
   PORT (
     num_1  : in unsigned (3 downto 0) ;
     gcd_num  : out unsigned (3 downto 0) ;
     num_2  : in unsigned (3 downto 0) ;
     clk  : in std_logic ;
     reset  : in std_logic );
END COMPONENT ;

BEGIN
  DUT  : gcd
    PORT MAP (
              num_1   => num_1  ,
             gcd_num   => gcd_num  ,
             num_2   => num_2  ,
             clk   => clk  ,
             reset   => reset   ) ;
```

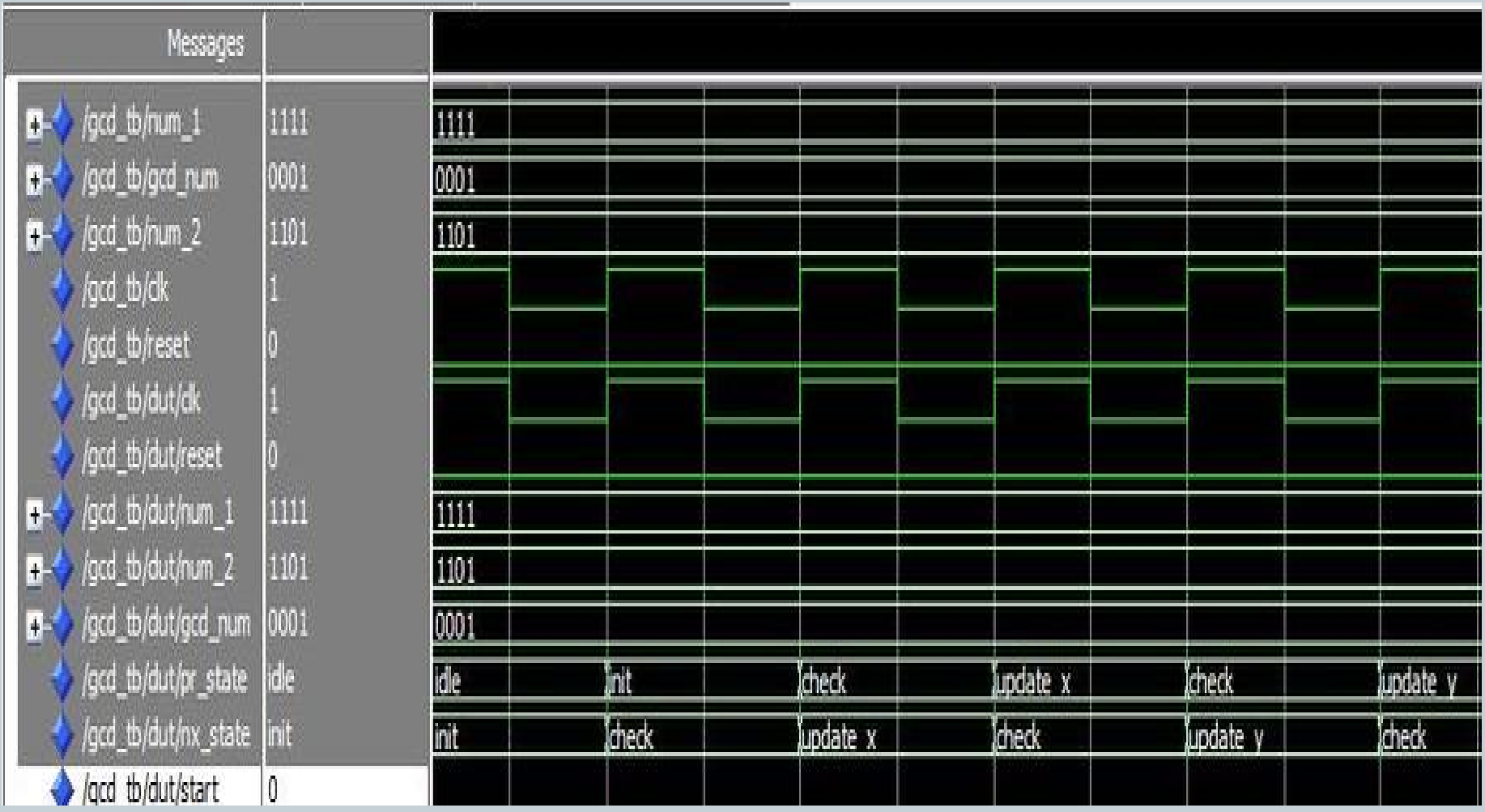
Contd…


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

 stim_proc: process
 begin
    -- hold reset state for 100 ns.
    wait for 5 ns;
```

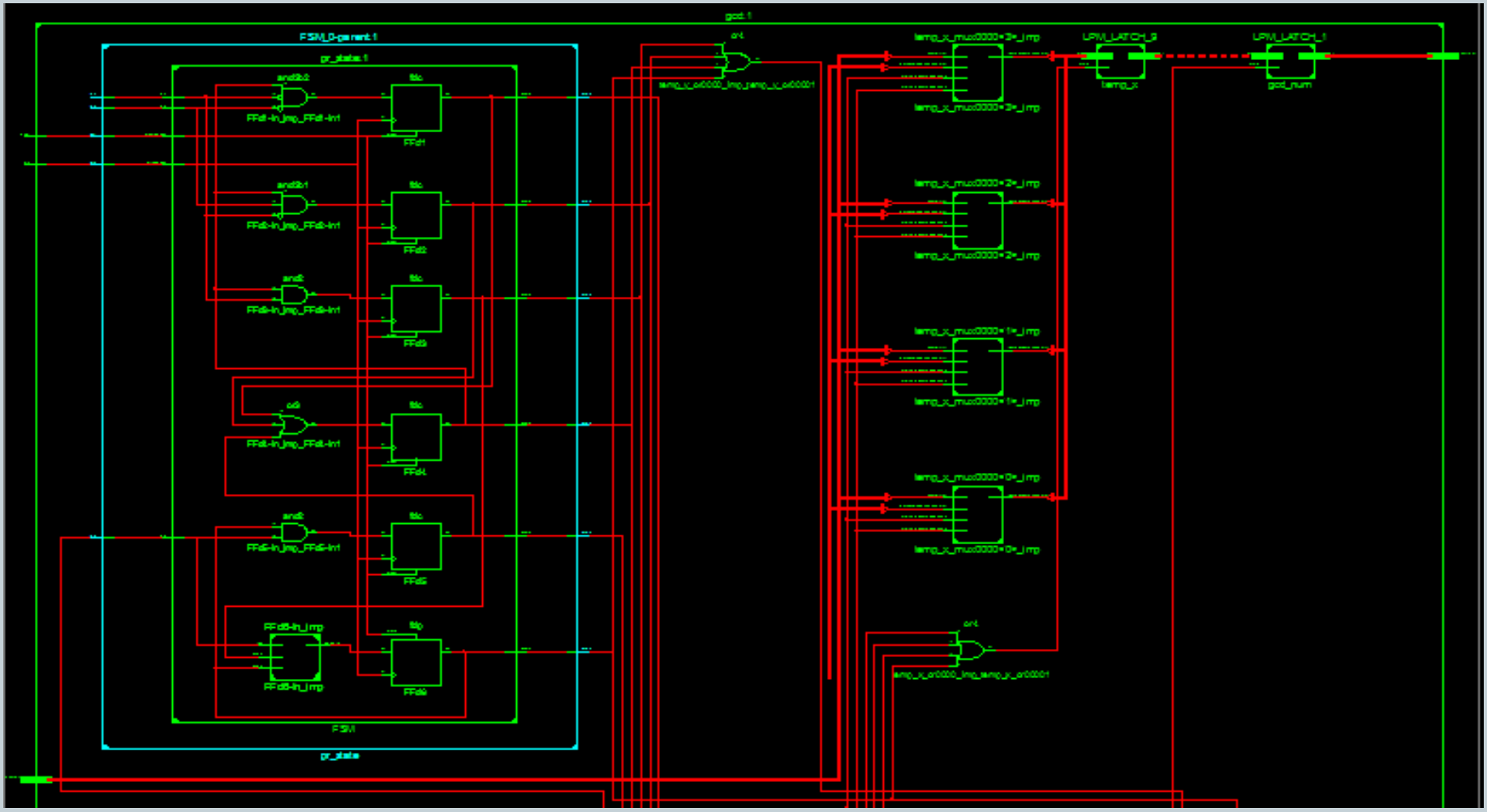
Contd...


```
reset<='0';
num_1<="1010";
num_2<="0101";
wait for 10 ns;
num_1<="1100";
num_2<="1001";
wait for 10 ns;
num_1<="1111";
num_2<="1101";
--wait for 10 ms;

  wait;
end process;

END ;
```

Simulation Result



Synthesis Result



THANK YOU