Razi University

Analysis and Modeling of Hardware using HDLs

Department of Computer Engineering
Razi University

Dr. Abdolhossein Fathi

References and Evaluation

- **Textbook:**
  - James R. Armstrong, F. Gail Gray, "*VHDL Design Representation and Synthesis*", Prentice Hall, 2000.
  - Z. Navabi, "*VHDL Analysis and Modeling of Digital Systems*", McGraw Hill, 1998.
  - VHDL and Velilog Language Reference Manual IEEE.

- **Score Details:**
  - Presentation 15%
  - Implementation Project 15%
  - Final Exam 70%

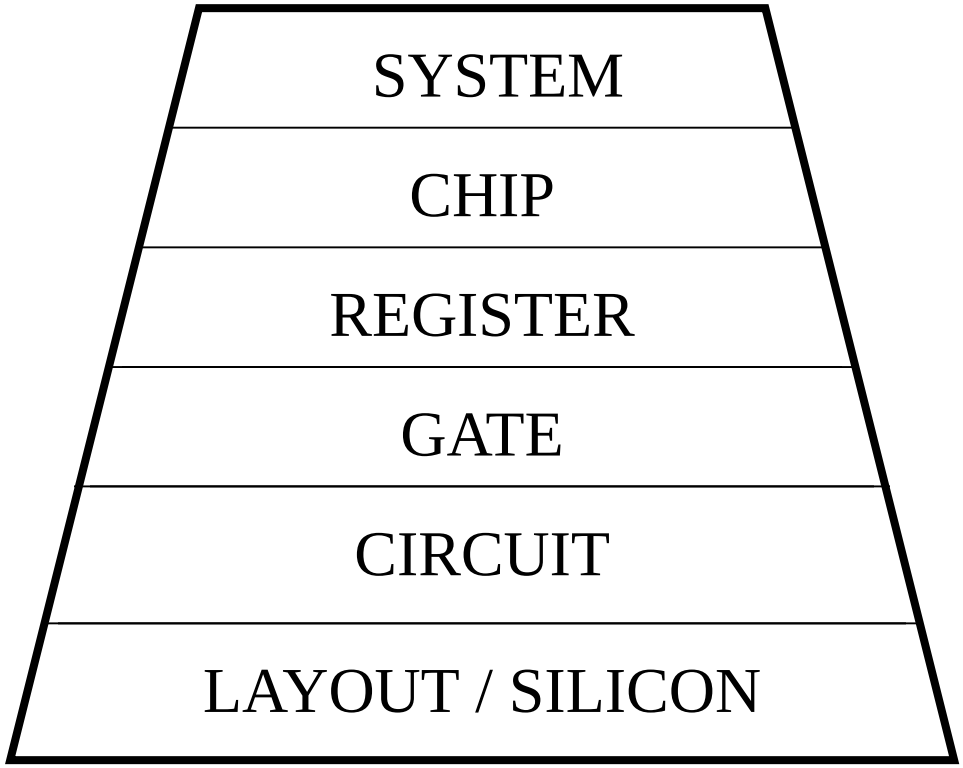
Contents

- Structured Design Concepts
- Design Tools
- Basic Features of VHDL
- VHDL Modeling and Simulation
  - Structural (Gate Level)
  - Data flow (RTL Level)
  - Behavioral (Algorithmic Level)
- Modeling for Synthesis

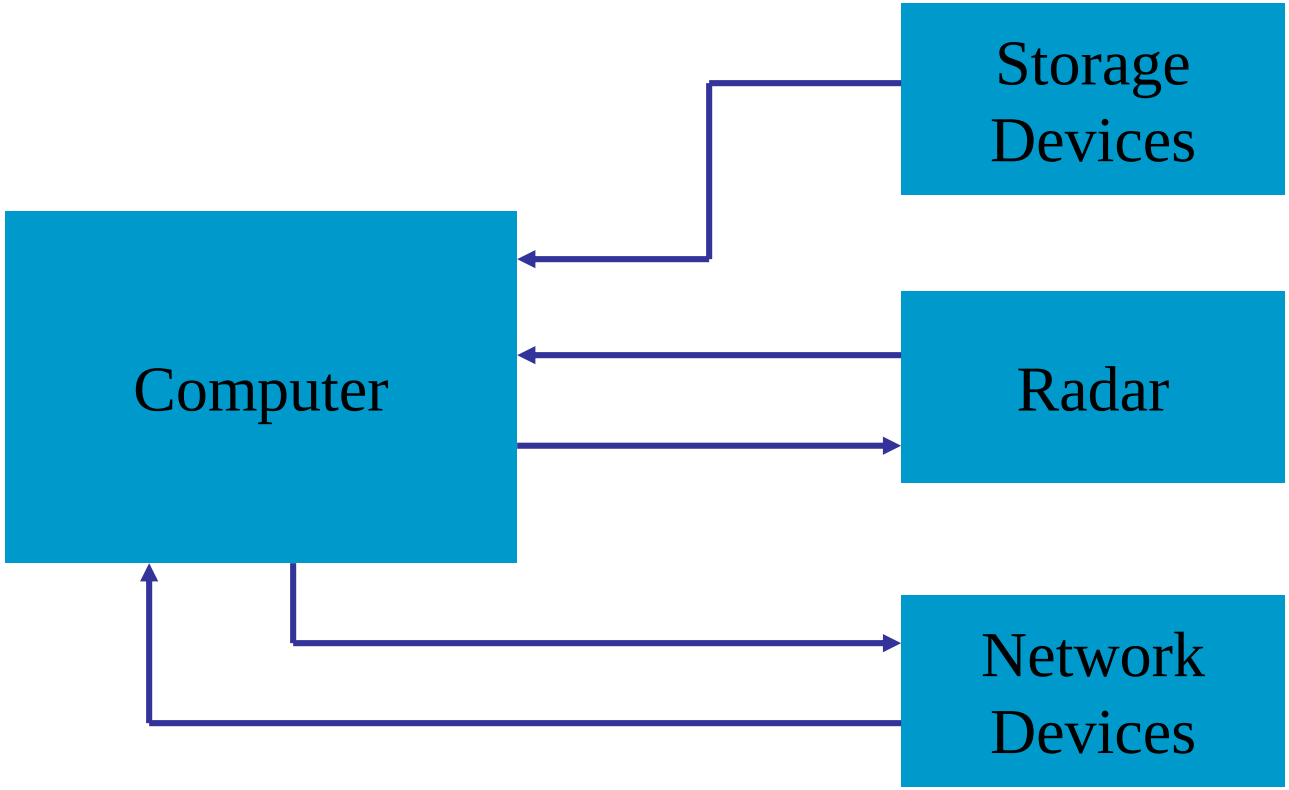
- Your Presentations are in a New Application, Model or Concept of VHDL (Each of yours one new paper after 2015).
- Your Projects are behavioral implementation and simulation of the method in your papers.

Structured Design Concepts

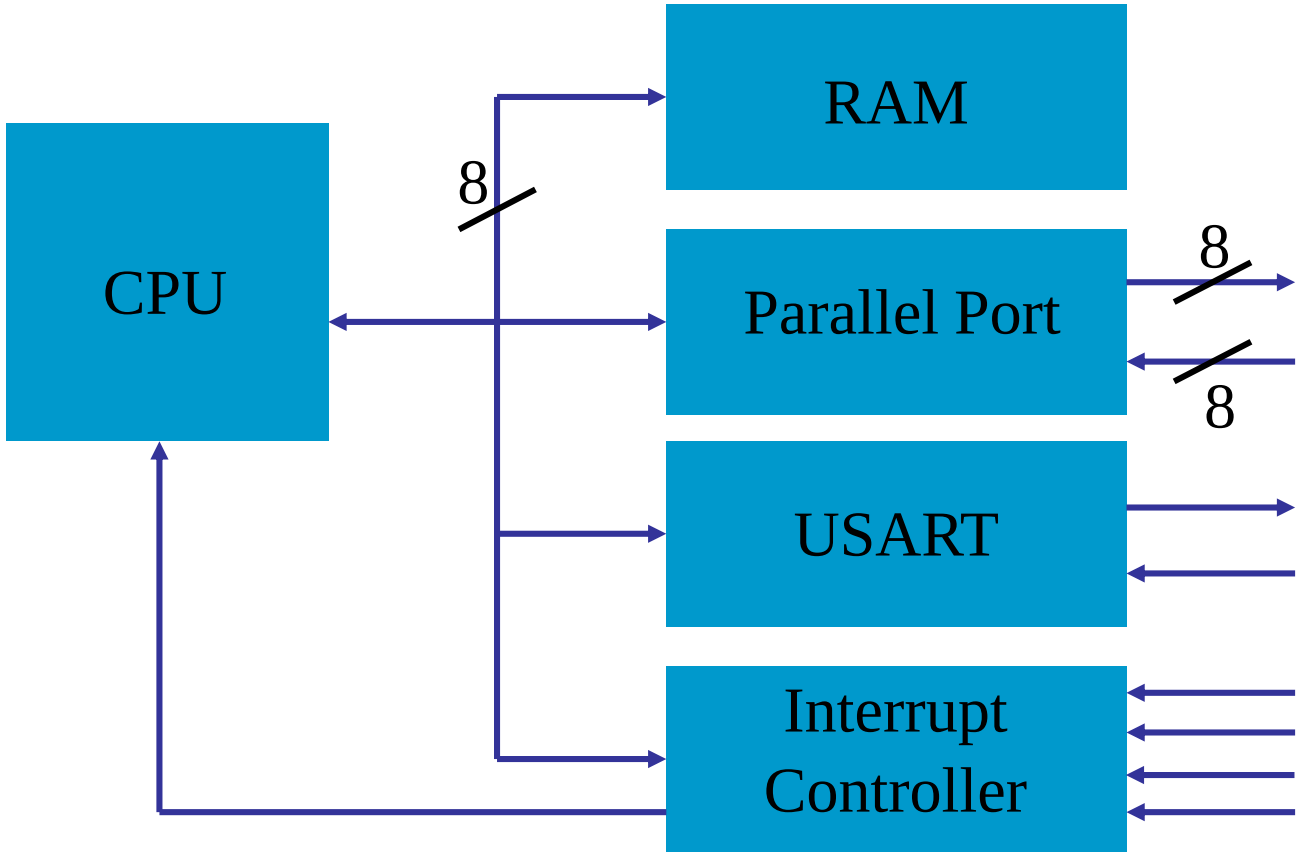
Abstraction Levels of Hardware Design



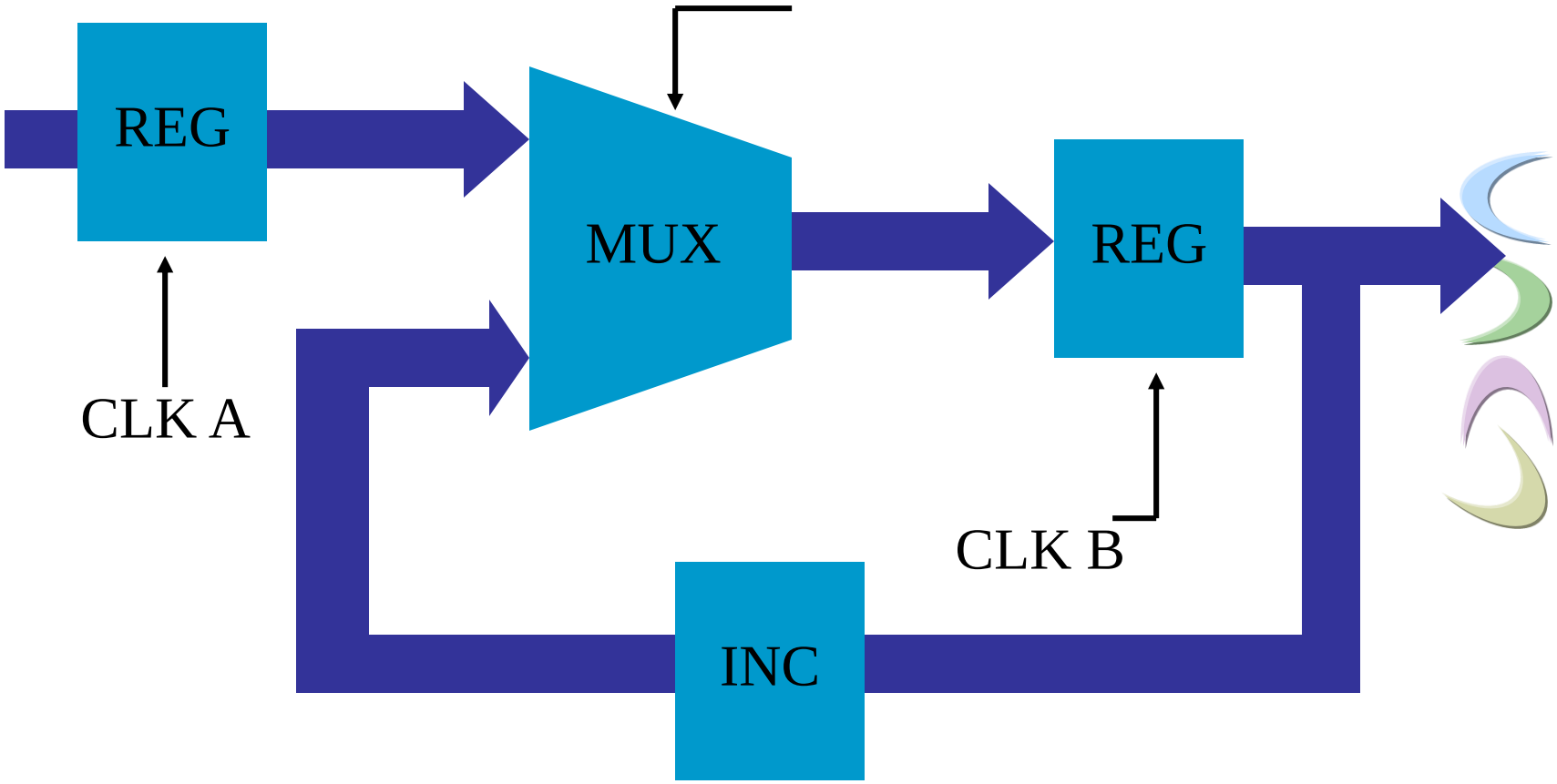
System Level



Chip Level

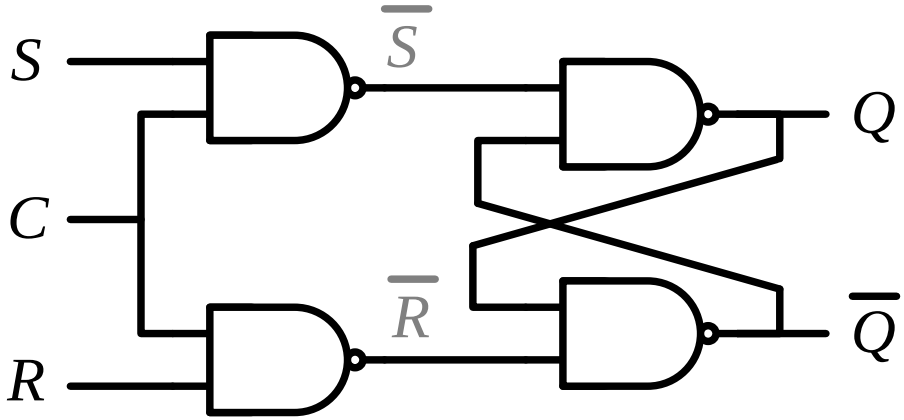


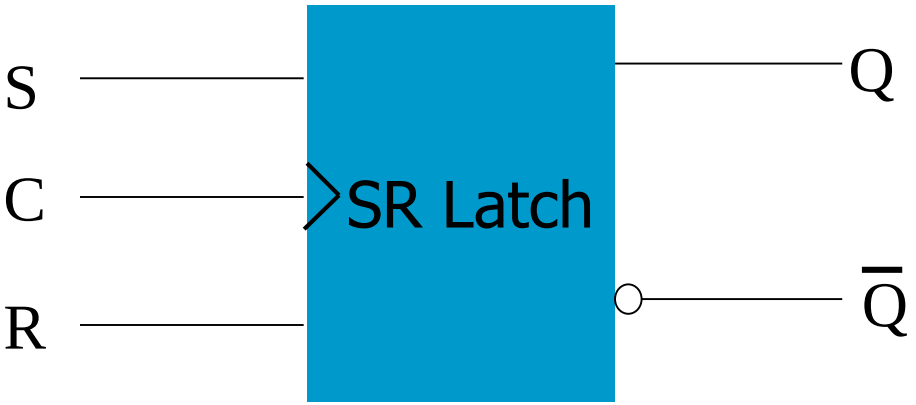
Register Level



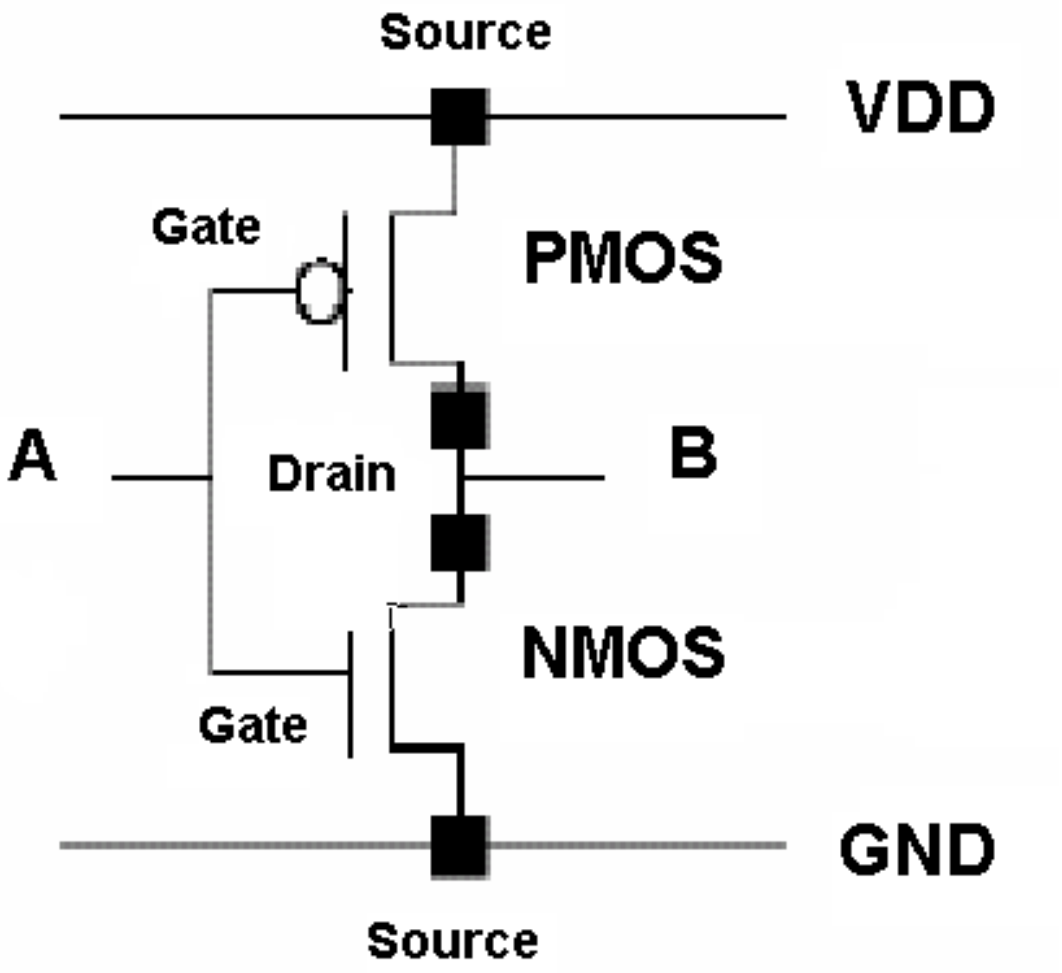
Gate Level

SR Latch

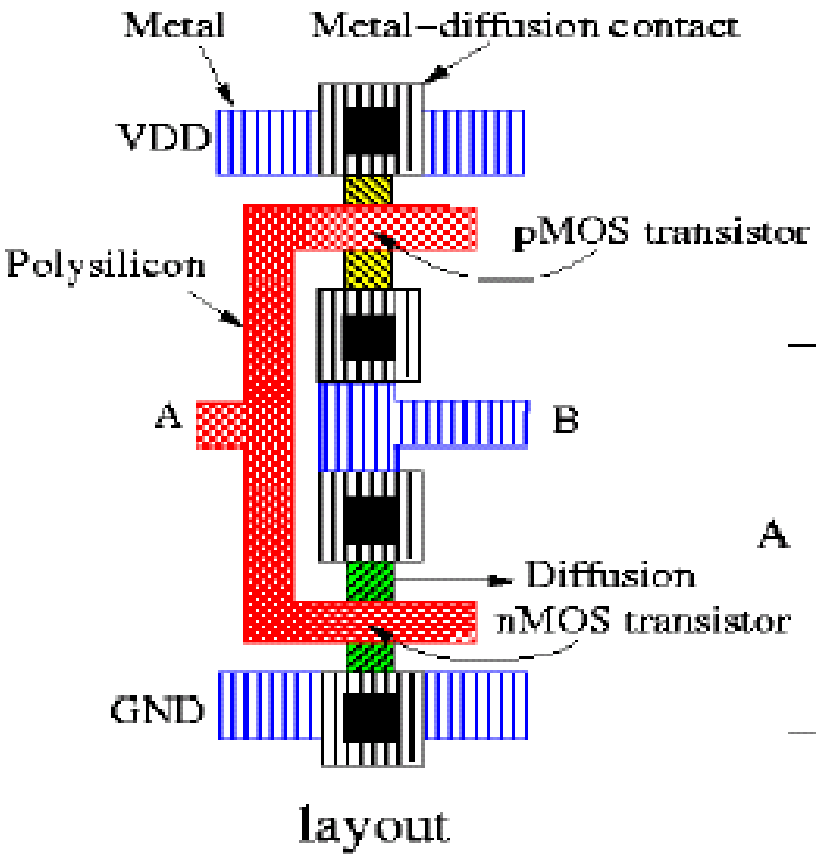


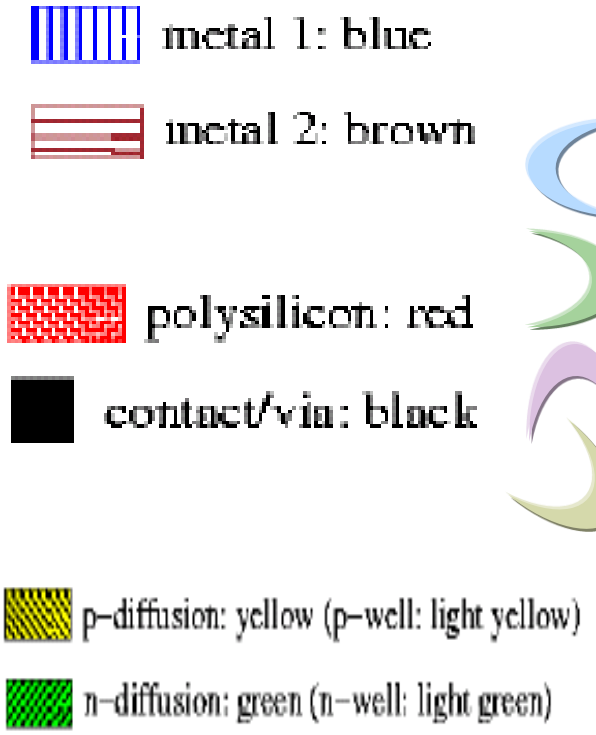


Circuit Level : CMOS Inverter



Layout/Silicon Level

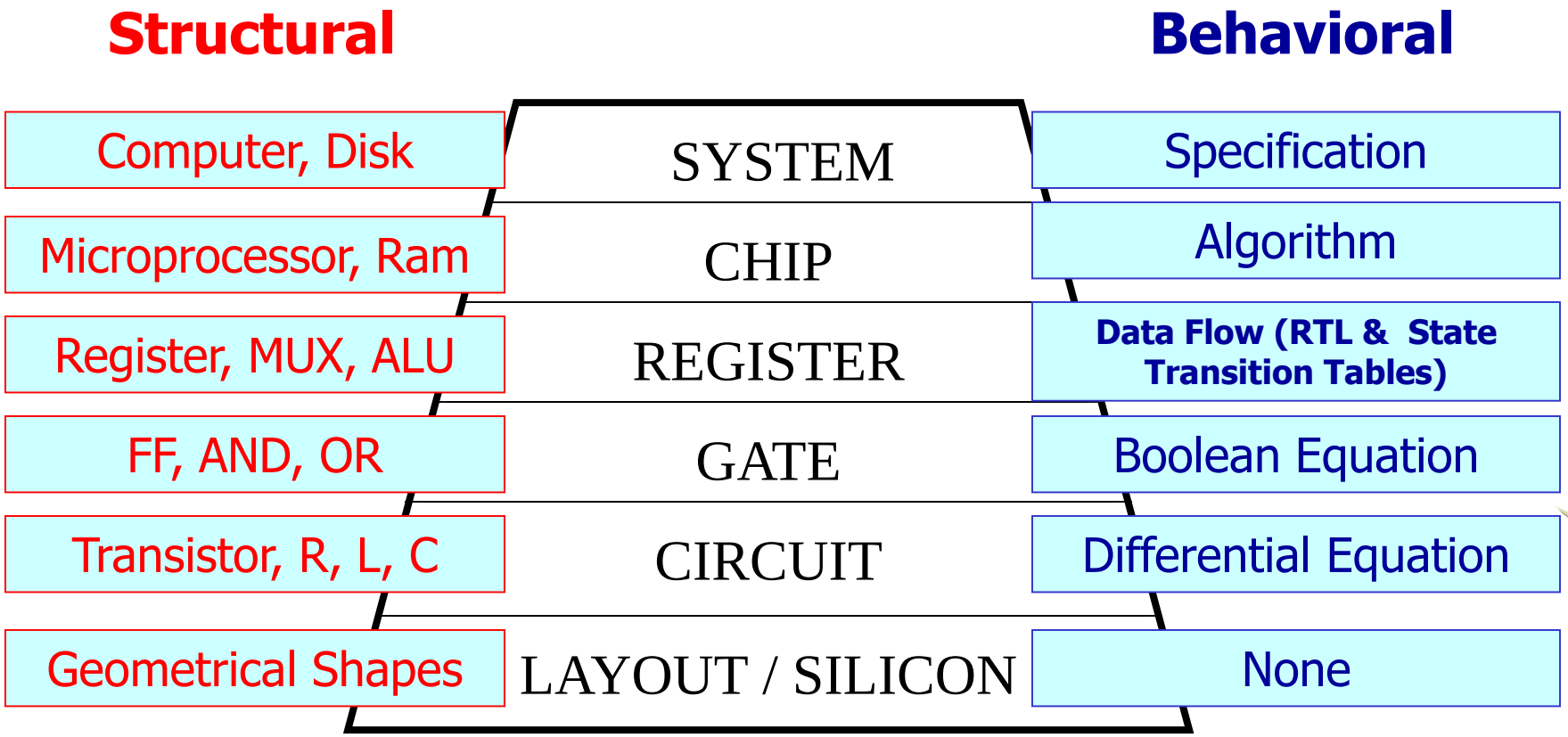




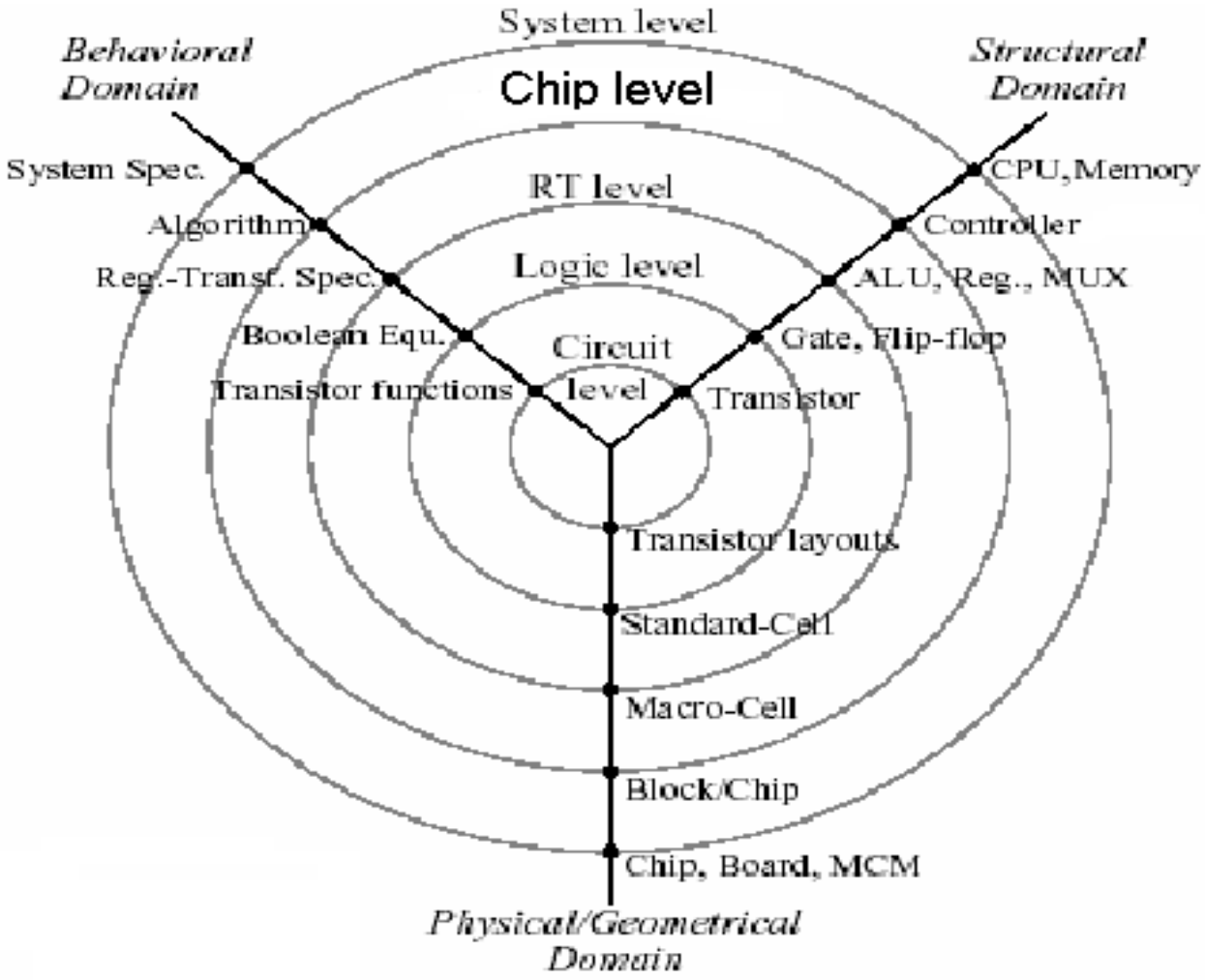
Hardware Modeling Domain

- **Behavioral Domain**
  - Components are defined with their input/output
  - Components can be implemented in different ways
- **Structural Domain**
  - Systems are defined with primitive components
  - Larger components contains Primitive components
- **Physical Domain**
  - Systems are defined with Geometrical of primitive components.
  - Primitive components are defined with their layout plans.

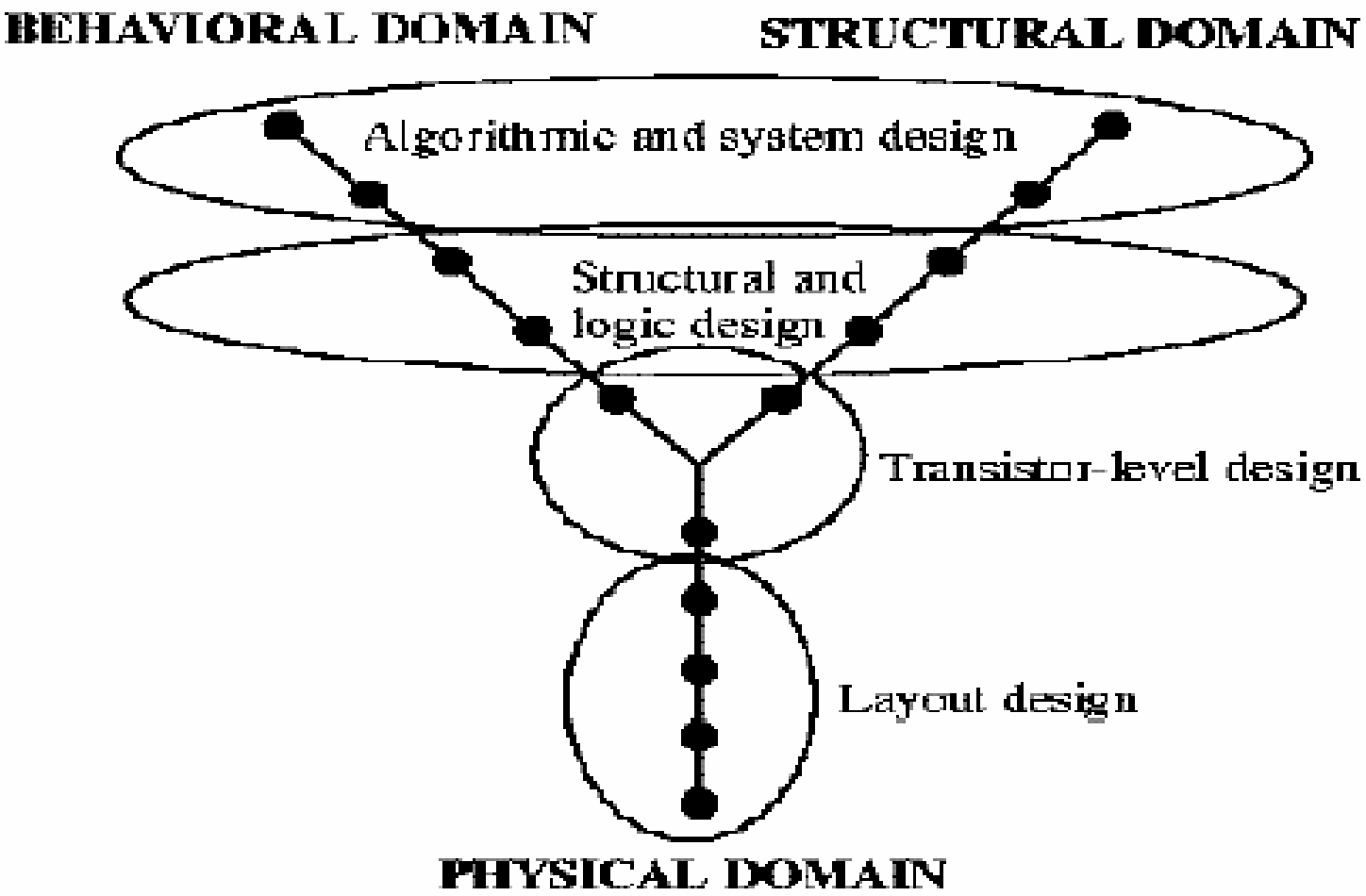
Abstraction Levels
Hardware Modeling Domain



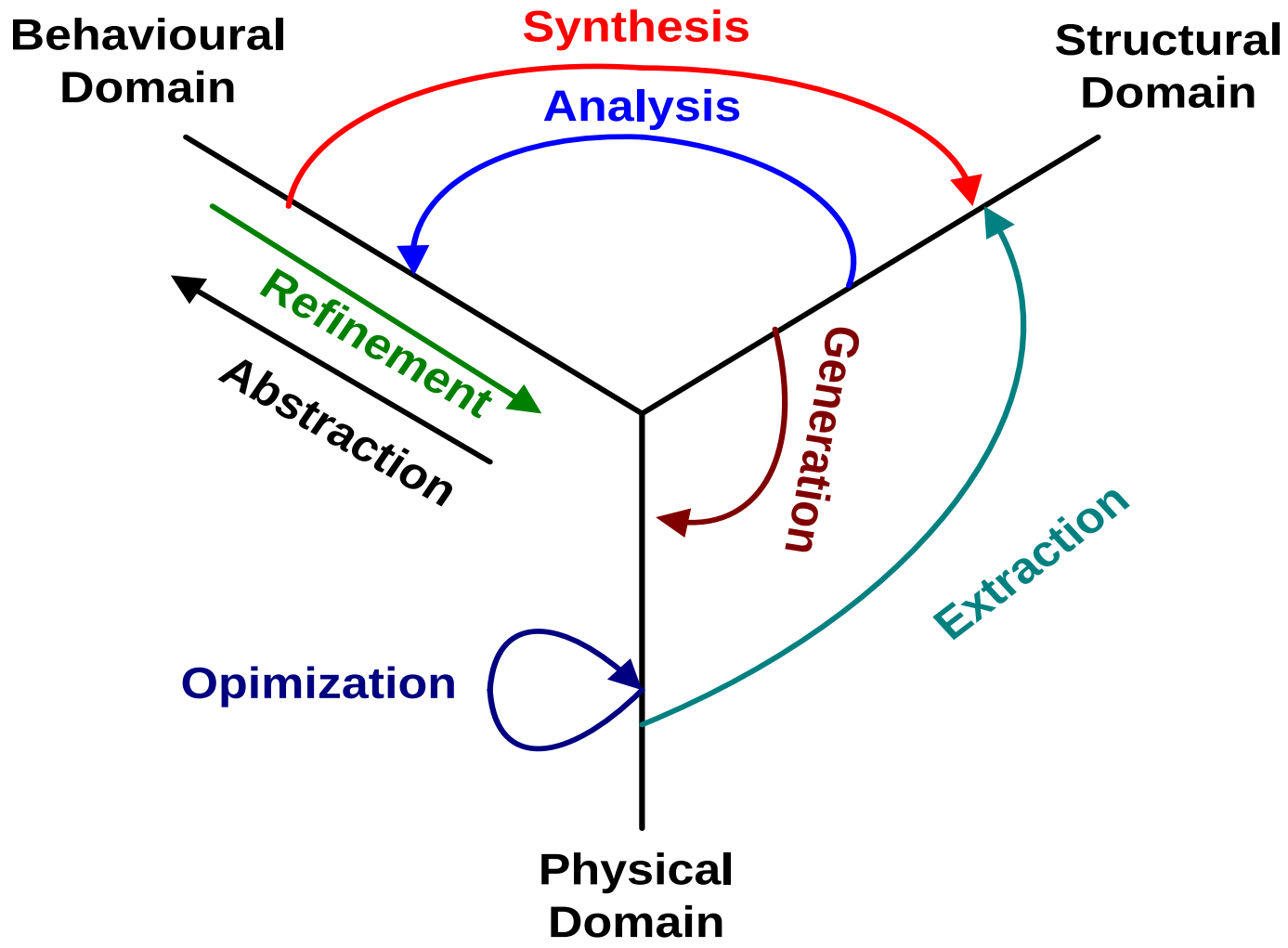
Y-Chart


System level
Behavioral Domain
Chip level
Structural Domain
System Spec.
CPU, Memory
RT level
Algorithm
Controller
Logic level
Reg.-Transf. Spec.
ALU, Reg., MUX
Boolean Equ.
Gate, Flip-flop
Circuit level
Transistor functions
Transistor
Transistor layouts
Standard-Cell
Macro-Cell
Block/Chip
Chip, Board, MCM
Physical/Geometrical Domain

Y-Char
Tool Groups



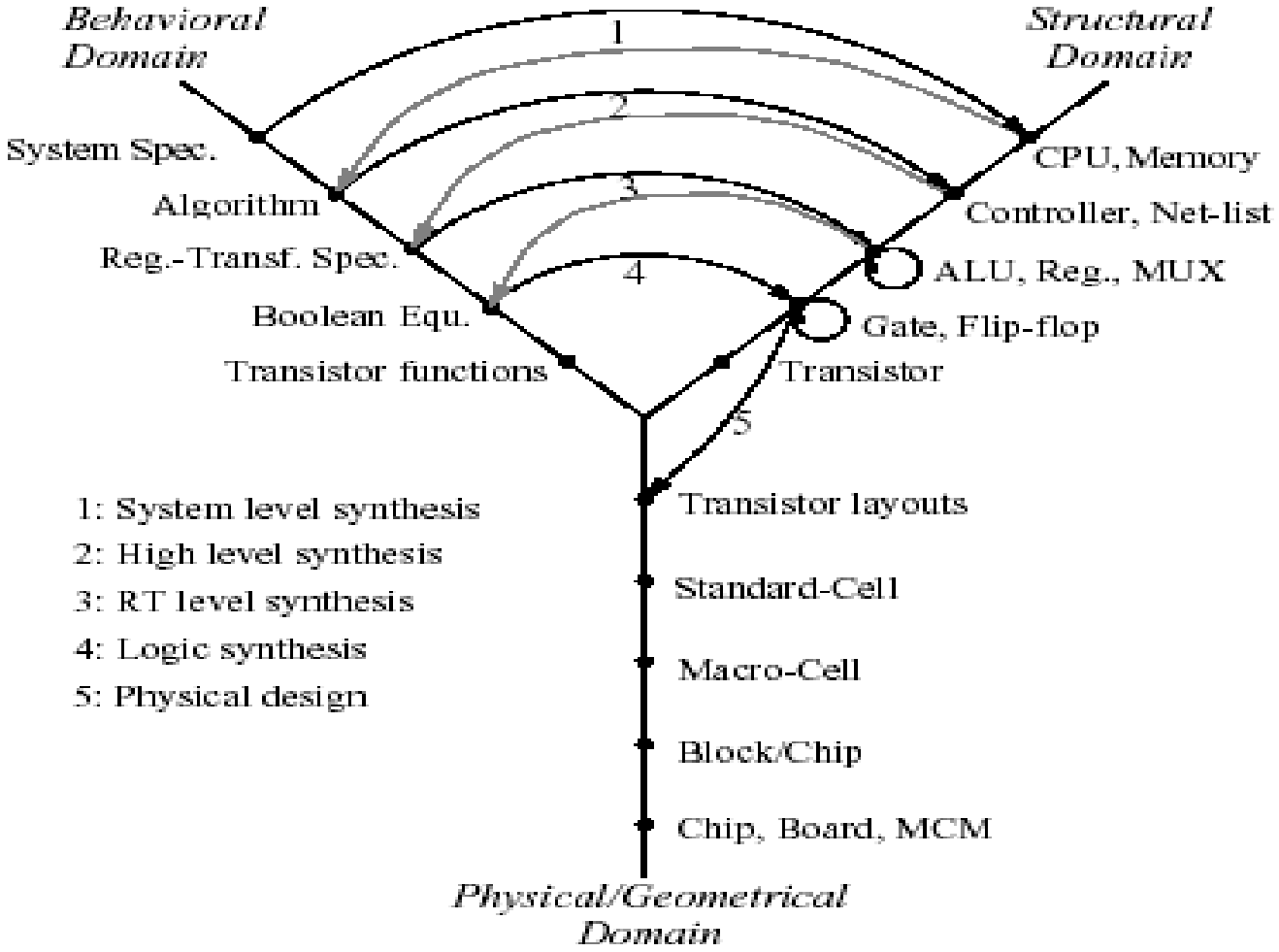
Y-Char
Design Concepts



Synthesis

- **Basic definition**
  - **Behavioral to structural domain (same abstraction level)**
- **Old definition**
  - **Generating a logic circuit from a truth table**
- **Modern definition (HDL domain)**
  - **Translating HDL code into a network of logic gates.**

Y-chart
Top-Down Synthesis



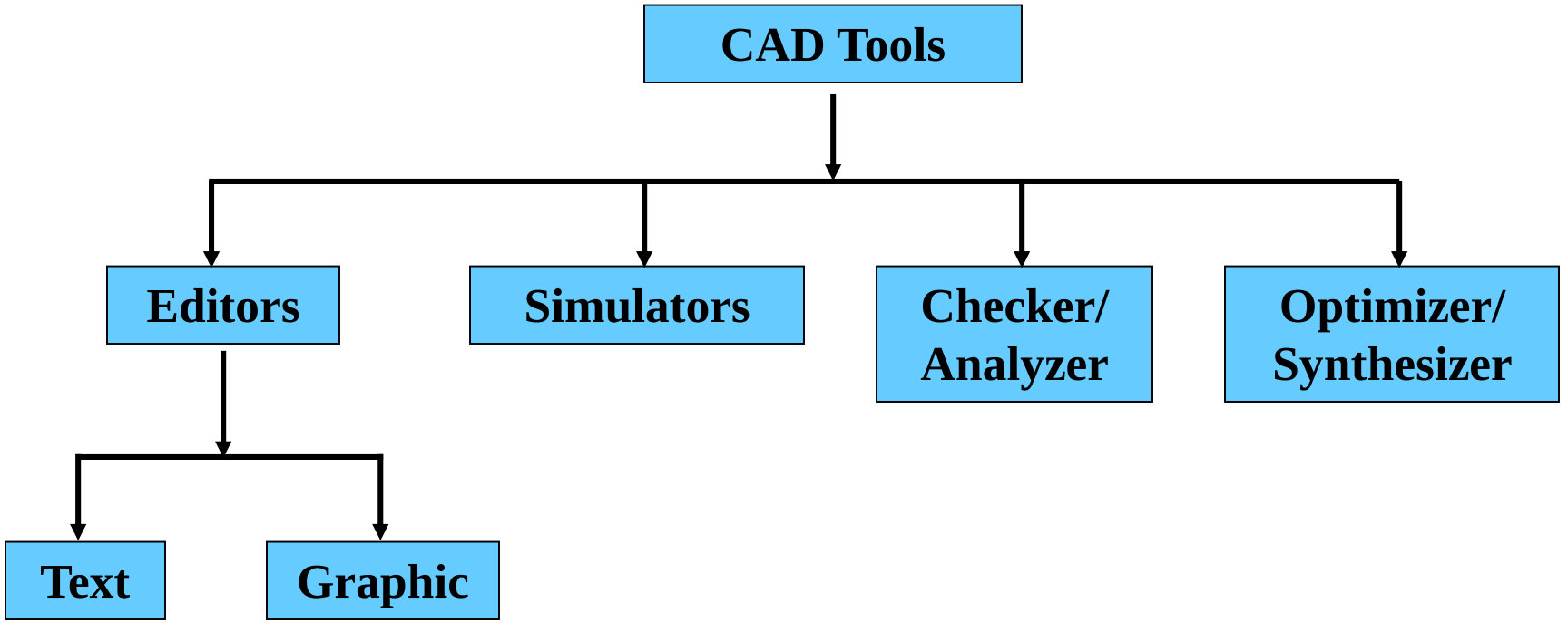
Design Methodology

- **Top-Down Design**
  - Begins at the top level
  - Partitioning the design in to lower-level primitives without any available primitives consideration.
  - The partitioning in each level is optimized for cost, speed and chip area, but it may produce not standard components.
- **Bottom-Up Design**
  - Begins at the top level
  - Partitioning the design in to lower-level primitives with available primitives consideration.
  - It is more economical, but its performance may not high as well as top-down design.
- **Most real designs use combination of both of them.**

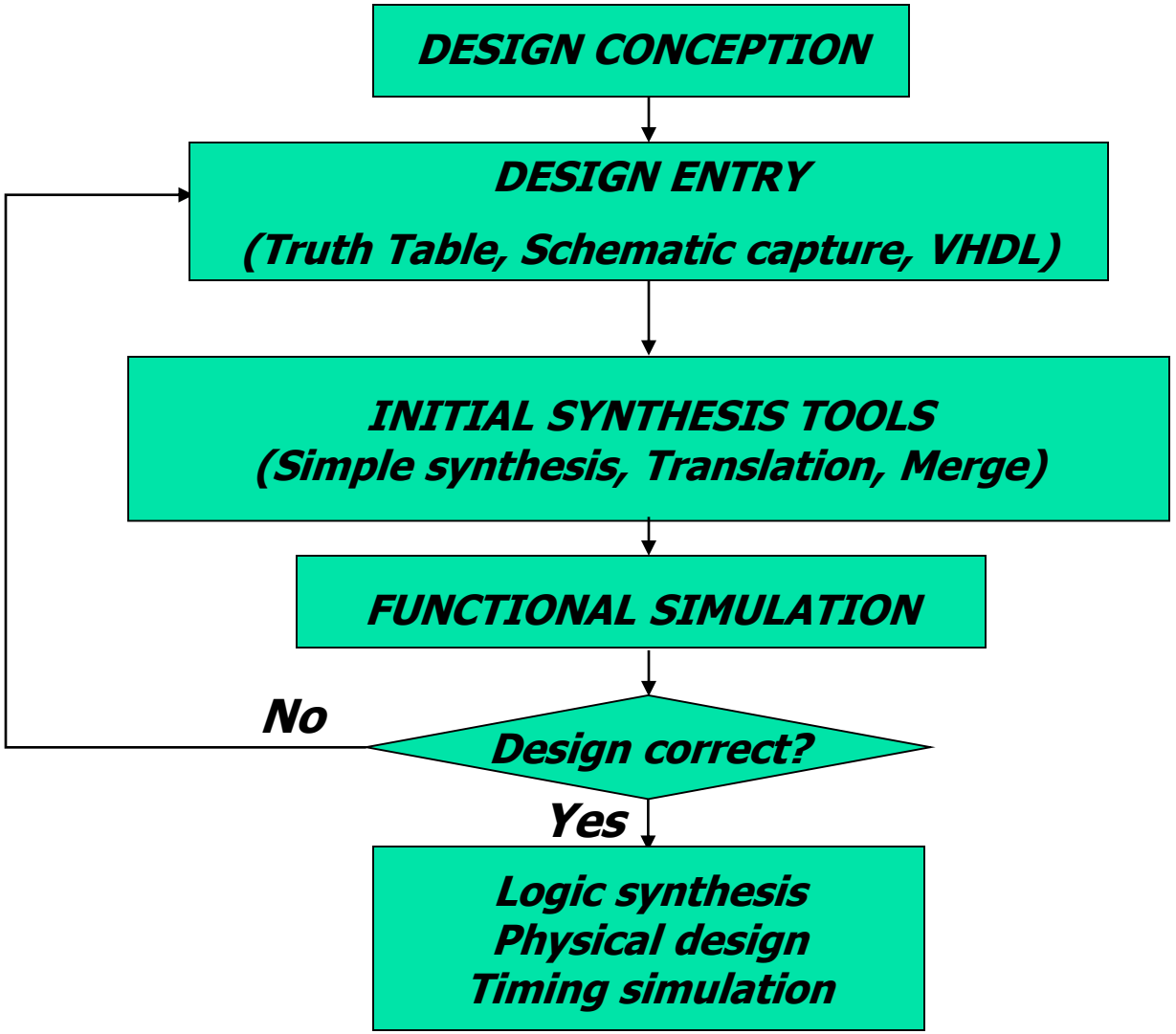
Design Tools

CAD Tool

- **CAD (Computer Aided Design) tool is a software that assists in performing or automating a particular design function.**



CAD Design Flow



Simulation

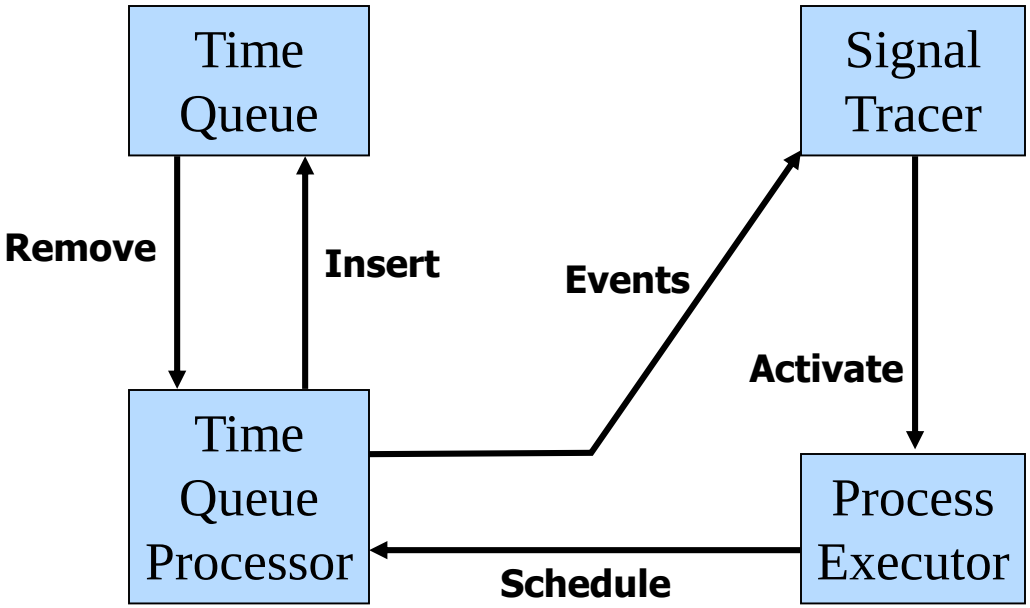
- **Functional simulation**
  - **Functionality Checking**
    - **Inputs & Outputs**
  - **Ignoring propagation delays**
- **Timing simulation**
  - **Determining the propagation delays**

Design Simulators

- **Stochastic**
  - **Stochastic simulation is carried out at the system level.**
  - **It determines the statistical parameters of system's units (like the percentage of time that a particular unit is busy).**
- **Deterministic**
  - **Deterministic simulation is carried out at all levels of the system (except the silicon level).**
  - **Based on the level of abstraction it determines different parameters of units (like voltages, bits or tokens).**

Simulator Organization

- **Simulation Cycle**
  - **Advance the simulation time to the time of next entry in the queue. If there are no new time queue entries, stop.**
  - **For all signals that have events at that time, active the processes triggered by them.**



Checker and Analyzer

- Checkers and Analyzers are employed at all levels.
- Rule Checkers
  - At the silicon level are used to insure that the layout of the circuit can be fabricated reliably.
  - At other levels are used to determine if connection rules or fan-out rules have been violated.
- Timing Analyzers
  - Can be used to check the longest path of logic circuit of system.
  - Can be used to check errors that violate in the hardware description.

Design Editors

- **Graphical**
  - **Schematics**
  - **Block diagram**
  - **State diagram and state tables**
  - **Timing diagram**
  - **Truth Tables …**
- **Text**
  - **Boolean Equation**
  - **Differential Equation**
  - **HDLs …**

HDLs:
Hardware Description Languages

- **Describing Hardware for**
  - **Design & Modeling**
  - **Simulation**
  - **Synthesis**
  - **Testing**
  - **Documentation**
  - **…**

HDLs: Hardware Description Languages

There are several language extensions built to assist in modeling:

- **VHDL: Appropriate for gate, register and chip levels.**
- **Verilog: Appropriate for circuit, gate and register levels.**
- **SystemC: Appropriate for all levels.**
- **SPICE: Appropriate for circuit level.**
- **ParaCore -** http://www.dilloneng.com/paracore.shtml
- **RubyHDL -**http://www.aracnet.com/~ptkwt/ruby_stuff/RHDL/index.shtml
- **MyHDL -** http://jandecaluwe.com/Tools/MyHDL/Overview.shtml
- **JHDL -** http://www.jhdl.org/
- **Lava -** http://www.xilinx.com/labs/lava/
- **HDLmaker -** http://www.polybus.com/hdlmaker/users_guide/
- **AHDL –** http://www.altera.com
  - **It is good for Altera-made Analog chips only, which limits its usefulness**

VHDL

- VHDL is an acronym of VHSIC Hardware Description Language
- VHSIC is an acronym of Very High Speed Integrated Circuits
- VHDL originated in the early 1980s
  - The American Department of Defense initiated the development of VHDL in the early 1980s
    - because the US military needed a standardized method of describing electronic systems
- VHDL was standardized in 1987 by the IEEE
  - IEEE Std-1076-1987
- ANSI Standard in 1988
- Added Support for RTL Design
  - VITAL: VHDL Initiative Towards ASIC Library
- Revised version in 1993,1995, 2000, 2002 and 2008.
- It is now accepted as one of the most important standard languages for
  - specifying
  - verifying
  - designing of electronics

Verilog

- Verilog is an acronym of Verifying Logic
- Phil Moorby from Gateway Design Automation in 1984 to 1987
  - Absorbed by Cadence
    - Cadence's ownership of Verilog => others support VHDL
- Verilog-XL simulator from GDA in 1986
- Synopsis Synthesis Tool in 1988
- In 1990 became open language
  - OVI: Open Verilog International
- IEEE Standard in 1995
  - IEEE Std-1364-1995
- Last revision in 2001
  - IEEE Std-1364-2001
- Ongoing work for adding
  - Mixed-signal constructs: Verilog-AMS
  - System-level constructs: SystemVerilog

VHDL vs. Verilog

VHDL	Verilog
All abstraction levels	All abstraction levels
Complex grammar	Easy language
Describe a system (everything)	Describe a digital system
Lots of data types	Few data types
User-defined package & library	No user-defined packages
Full design parameterization	Simple parameterization
Easier to handle large designs	-
Very consistent language. Code written and simulated in one simulator will behave exactly the same in another simulator. E.g. strong typing rules.	Less consistent language. If you don't follow some adhoc methodology for coding styles, you will not get it right.
-	It executes differently on different platforms unless you follow some adhoc coding rules.

VHDL vs. Verilog (Cont.)

VHDL	Verilog
In Europe the VHDL is the most popular language	-
Based on Pascal language	Based on C language
Most FPGA design in VHDL	Most ASIC design in Verilog
Easier for designing at the gate-level	Easier for designing at the gate-level
-	allows the simulator greater freedom
delta cycles of delay in propagating in each level	multiple levels of zero-delay gates
allows concurrent procedure calls	does not allow concurrent task calls

VHDL vs. Verilog: Managing Large designs

- ***VHDL:***
  - **Configuration, generate, generic and package statements all help manage large design structures**
- ***Verilog:***
  - **There are no statements in Verilog that help manage large designs**

The *generate* statement replicates a number of instances of the same design-unit or some sub part of a design, and connects it appropriately

Languages "under development"

- SystemVerilog
  - Extending Verilog to higher levels of abstraction for architectural and algorithm design and advanced verification
- VHDL 201x
  - Goal of VHDL Analysis and Standards Group (VASG):
    - Enhance/update VHDL for to improve performance, modeling capability, ease of use, simulation control, and the type system
      - e.g.: Data types and abstractions:
        - variant records
        - interfaces

Verilog Weak Spots

- **Not well suited for complex, high level modeling**
  - **No user defined type definition**
  - **No concept of libraries, packages, configurations**
  - **No ‘generate’ statement - can’t build parameterized structural models**
  - **No complex types above a two-dimensional array**

- **Based on these limitation we first try to learn VHDL programming.**

Basic Features of VHDL

Mechanisms by which to reduce complexity

- **Language abstractions** use the language to describe complex matters without having to describe small details
  - **Functions and procedures are important parts of the language in order to handle complexity**
- **Design hierarchy** uses components in order to conceal details - the black box principle
  - **The term black box means that only inputs/outputs of a component are visible at a certain level**
  - **It is the designer who decides how many different hierarchies there are to be in the design**

VHDL Component

- Components are a central concept in VHDL
- Components are used, among other things, to build up component libraries, e.g.:
  - microprocessors
  - special user circuits
  - other standard circuits
- If a "good" component has been designed, it can be saved in a component library, enabling it to be copied as many times as required, i.e. components are reusable
  - this is called creating instances, i.e. creating the component in a schematic or in the text file

Using the Black Box

- The internal structure can be concealed from the designer - the black box principle.
- In some cases there is no need to know how component is structured.
- The designer is usually only interested in
  - inputs and outputs
  - a specification function and
  - access times
- The majority of hardware designers are used to working with black boxes such as the 74LSXX circuit family, for example

Key feature: delta delay

- VHDL uses the concept of delta delay to keep track of processes that should occur at a given time step, but are actually evaluated in different machine cycles
- A delta delay is a unit of time as far as the simulator hardware is concerned, but in the simulation time has no advance

Object-Based Language

- Staying with computer science a while longer, VHDL is an object-based language, i.e. what separates VHDL from object-oriented languages is that the language does not have inheritance
- Generic components and instantiation are typical for object-based languages
- Generic components are components which can be modified before instantiation, e.g. a generic component which copies with different width for the input and output signals

VHDL Features

- Case insensitive
  - inputa, INPUTA and InputA are refer to same variable.
- Comments
  - '--' until end of line.
  - If you want to comment multiple lines, '--' need to be put at the beginning of every single line.
- Statements are terminated by ';'
- Signal assignment:
  - '<='
- Variable and Constant assignment:
  - ':='
- User defined names:
  - Start with a letter.
  - Can consist of letters, numbers, underscores ('_').

Reserved VHDL keywords

ABS ACCESS AFTER ALIAS ALL AND ARCHITECTURE ARRAY ASSERT ATTRIBUTE BEGIN BLOCK BODY BUFFER BUS CASE COMPONENT CONFIGURATION CONSTANT	DISCONNECT DOWNTO ELSE ELSIF END ENTITY EXIT FILE FOR FUNCTION GENERATE GENERIC GROUP GUARDED IF IMPURE	IN INERTIAL INOUT IS LABEL LIBRARY LINKAGE LITERAL LOOP MAP MOD NAND NEW NEXT NOR NOT NULL	OF ON OPEN OR OTHERS OUT PACKAGE PORT POSTPONED PROCEDURE PROCESS PURE RANGE RECORD REGISTER REM REPORT ROL ROR	RETURN SELECT SEVERITY SIGNAL SHARED SLA SLL SRA SRL SUBTYPE THEN TO TRANSPORT TYPE UNAFFECTED UNITS UNTIL USE	VARIABLE WAIT WHEN WHILE WITH XNOR XOR

VHDL Built-In Operators

- Logical operators:
  - NOT, AND, OR, NAND, NOR, XOR, XNOR (XNOR only in VHDL'93 and later)
- Relational operators:
  - =, /=, <, <=, >, >=
- Addition operators:
  - +, -, &
- Multiplication operators:
  - *, /, mod, rem
- Miscellaneous operators:
  - **, abs

VHDL Structure

- Library
  - Definitions, constants
- Entity
  - Interface
- Architecture
  - Implementation, function

VHDL - Library

- Include library

  library IEEE;

- Define the library package used

  Use ieee.std_logic_1164.all; -- **defines a standard for designers to use in describing interconnection data types used in VHDL modeling.**

  Use ieee.std_logic_arith.all; -- **provides a set of arithmetic, conversion, comparison functions for signed, unsigned, std_ulogic, std_logic, std_logic_vector.**

  Use ieee.std_logic_signed.all; -- **provides a set of signed arithmetic, conversion, and comparison functions for std_logic_vector.**

  Use ieee.std_logic_unsigned.all; -- **provides a set of unsigned arithmetic, conversion, and comparison functions for std_logic_vector.**

- See all available packages at http://www.cs.umbc.edu/portal/help/VHDL/stdpkg.html

IEEE Predefined Data Types

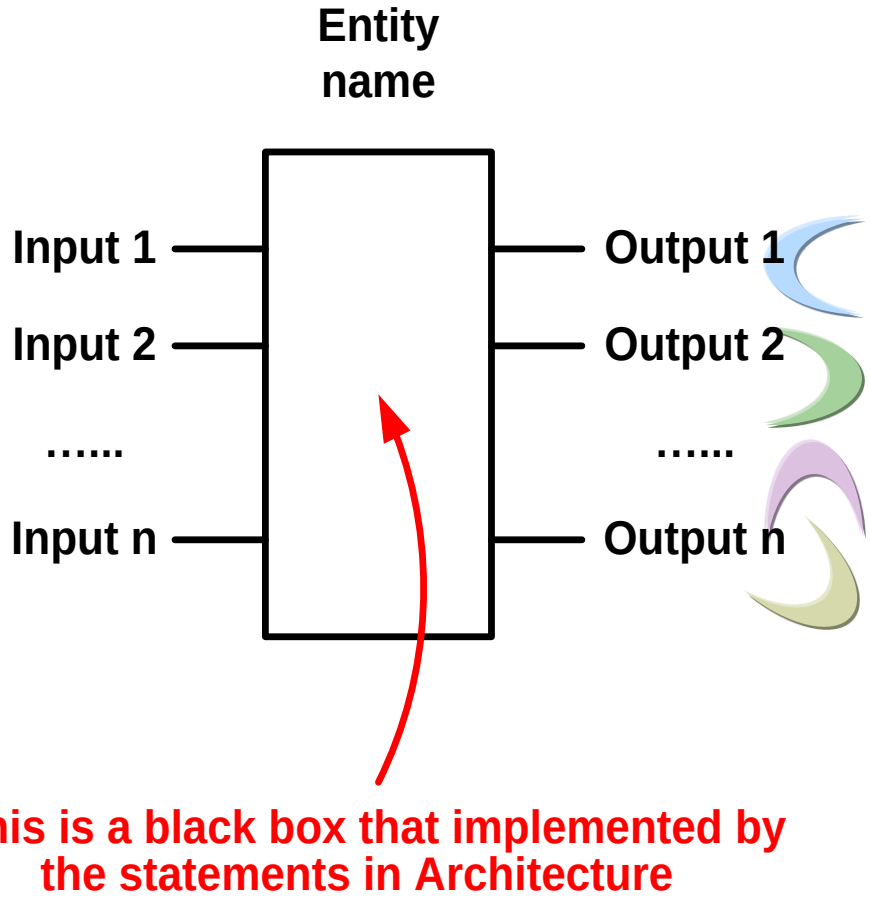
- bit values: '0', '1'
- boolean values: TRUE, FALSE
- integer values: $-(2^{31})$ to $+(2^{31} - 1)$
- std_logic values: '1','0','Z'
- std_ulogic values: 'U','X','1','0','Z','W','H','L','-'
  - 'U' = uninitialized
  - 'X' = unknown
  - 'W' = weak 'X'
  - 'Z' = floating
  - 'H' = weak '1'
  - 'L' = weak '0'
  - '-' = don't care
- Std_logic_vector (0 upto n);
- Std_logic_vector (n downto 0);

Other Existing Data Types

- real values: -1.0E38 to 1.0E38
- bit_vector (array of bits)
- time (physical data type)
- Subtype
  - Allows for user defined constraints on a data type.
  - May include entire range of base type.
- Array
  - Used to collect one or more elements of a similar type in a single construct.
- Record
  - Used to collect one or more elements of different types in a single construct.

VHDL - Entity

- It is the interface for communication among different modules/components and define the signal port modes (INPUT and OUTPUT)



Syntax of the Entity Declaration

entity **<entity_name>** is

port([signal] **<identifier1>:[<mode>] <type_indication>;**

...

[signal] **<identifierN>:[<mode>] <type_indication>)**

end [*entity*] **[<entity_name>];**

<mode> = in, out, inout, buffer, linkage

in: Component only read the signal

out: Component only write to the signal

inout: Component read or write to the signal (bidirectional signals)

buffer: Component write and read back the signal (no bidirectional signals, the signal is going out from the component)

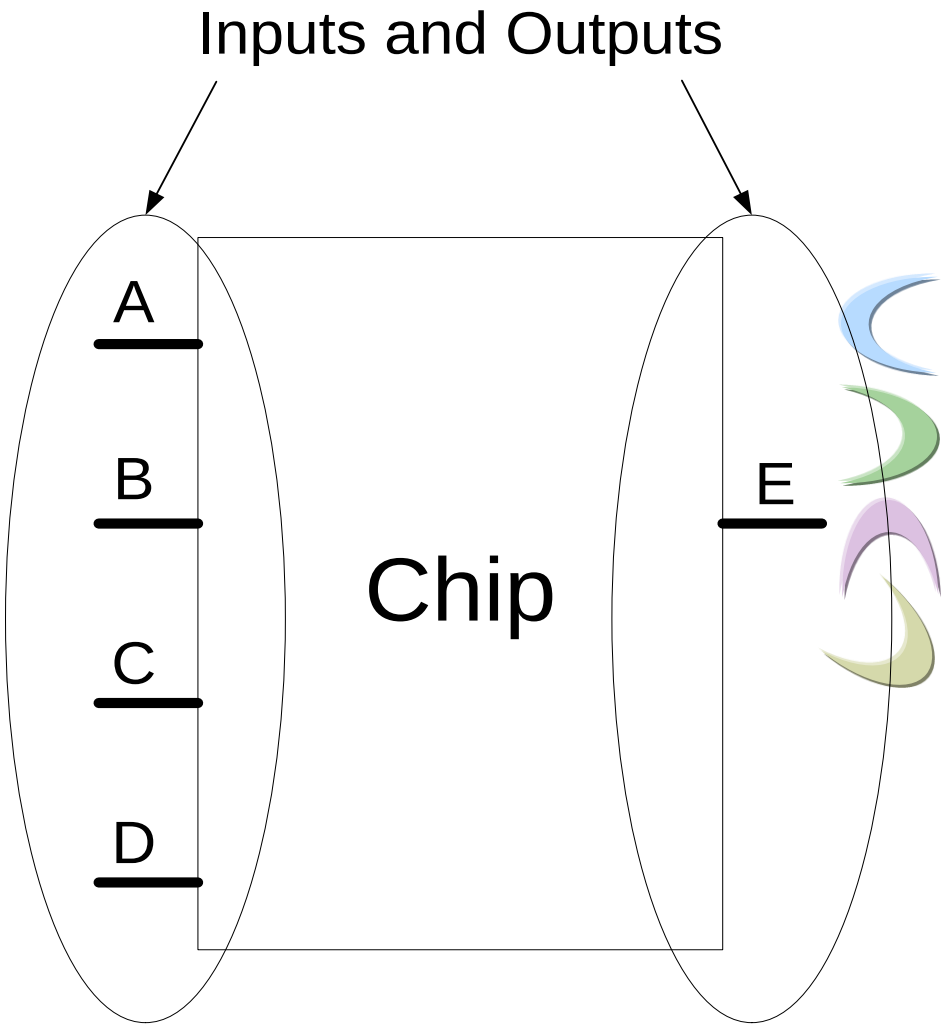
linkage: Used only in the documentation

VHDL – Entity Example

- **Define inputs and outputs**
- **Example:**

```
Entity ChipName is
Port( A,B,C,D: in std_logic;
      E: out std_logic
    );
End ChipName ;
```

DO NOT have ; here



Entity name (ChipName) should be same as the file name

Generics

- **Generics can be used to introduce information into a model, e.g. timing information.**
- **The generic must be declared in the entity before the port command.**
- **A generic can have any data type**
- **There are some restrictions with regard to what synthesis tools will accept.**
- **Example:**

```
Entity gen_ex is
    generic (del: time:=10 ns; N: Integer);
    port (a,b: in std_logic_vector(0 to N-1); c: out std_logic);
end;
```

- The component above has a **generic** delay of ***del*** ns.
- The del has been defined as 10 ns.
- This value can be changed if the component is instanced.
- Generics can also be used to design **generic** components (as define N).

Architecture

- An **architecture** defines a body for a component entity.
- An **architecture** body specifies a behavior between inputs and outputs.
- Input port can only be read inside architecture
  - input1 <= temp; -- This statement is NOT allowed
- Output port can only be written inside architecture
  - temp <= output1; -- This statement is NOT allowed

Syntax of the Architecture

architecture <architecture_name> **of** <entity_name> **is**
[<architecture_declarative_part>]
Begin

<architecture_statement_part> -- The body of the arch.

end [*architecture*] [<architecture_name>]**;**

- The architecture_name is not the same as the entity_name: instead an architecture is tied to an entity.
- The word "*architecture*" in the last line is not supported before the VHDL-93 standard

Architecture Declarative Part

- The **architecture_declarative_part** must be defined before first **begin** and can consist of, for example:
  - **constant declarations**
  - **variable declarations**
  - **signal declarations**
  - **types**
  - **subprograms**
  - **components**

Constant Declaration

- A constant can have a single value of a given type.
- A constant's value cannot be changed during the simulation.
- Constants declared at the start of an architecture can be used anywhere in the architecture.
- Constants declared in a process can only be used inside the specific process.

```
CONSTANT  constant_name  :  type_name  [ : = value];

CONSTANT  rise_fall_time  :  TIME  : =  2 ns;
CONSTANT  data_bus  :  INTEGER  : =  16;
```

Variable Declaration

- Variables are used for local storage of data.
- Variables are generally not available to multiple components or processes.
- All variable assignments take place immediately.
- Variables are more convenient than signals for the storage of (temporary) data.

```
VARIABLE variable_name : type_name [:=value];

VARIABLE opcode : BIT_VECTOR(3 DOWNTO 0) := "0000";
VARIABLE freq : INTEGER;
```

Signal Declaration

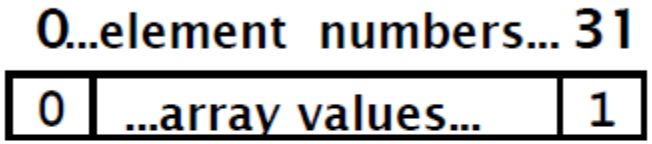
- Signals are used for communication between components.
- Signals are declared outside the process.
- Signals can be seen as real, physical signals.
- Some delay must be incurred in a signal assignment.

```
SIGNAL signal_name : type_name [:=value];

SIGNAL brdy : BIT;
SIGNAL output : INTEGER := 2;
```

Array

- Array
  - Used to collect one or more elements of a similar type in a single construct.
  - Elements can be any VHDL data type.

```
TYPE data_bus IS ARRAY (0 TO 31) OF BIT;
```




```
VARIABLE X:  data_bus;
VARIABLE Y:  BIT

Y := X(12);  -- Y gets value of 12th element
```

```
TYPE register IS ARRAY (15 DOWNTO 0) OF BIT;
```

Record

- Record
  - Used to collect one or more elements of different types in a single construct.
  - Elements can be any VHDL data type.
  - Elements are accessed through field name.

```
TYPE binary IS ( ON, OFF );
TYPE switch_info IS
    RECORD
      status : binary;
      IDnumber : integer;
END RECORD;

VARIABLE switch : switch_info;

switch.status := on;  -- status of the switch
switch.IDnumber := 30;  -- number of the switch
```

Subtype

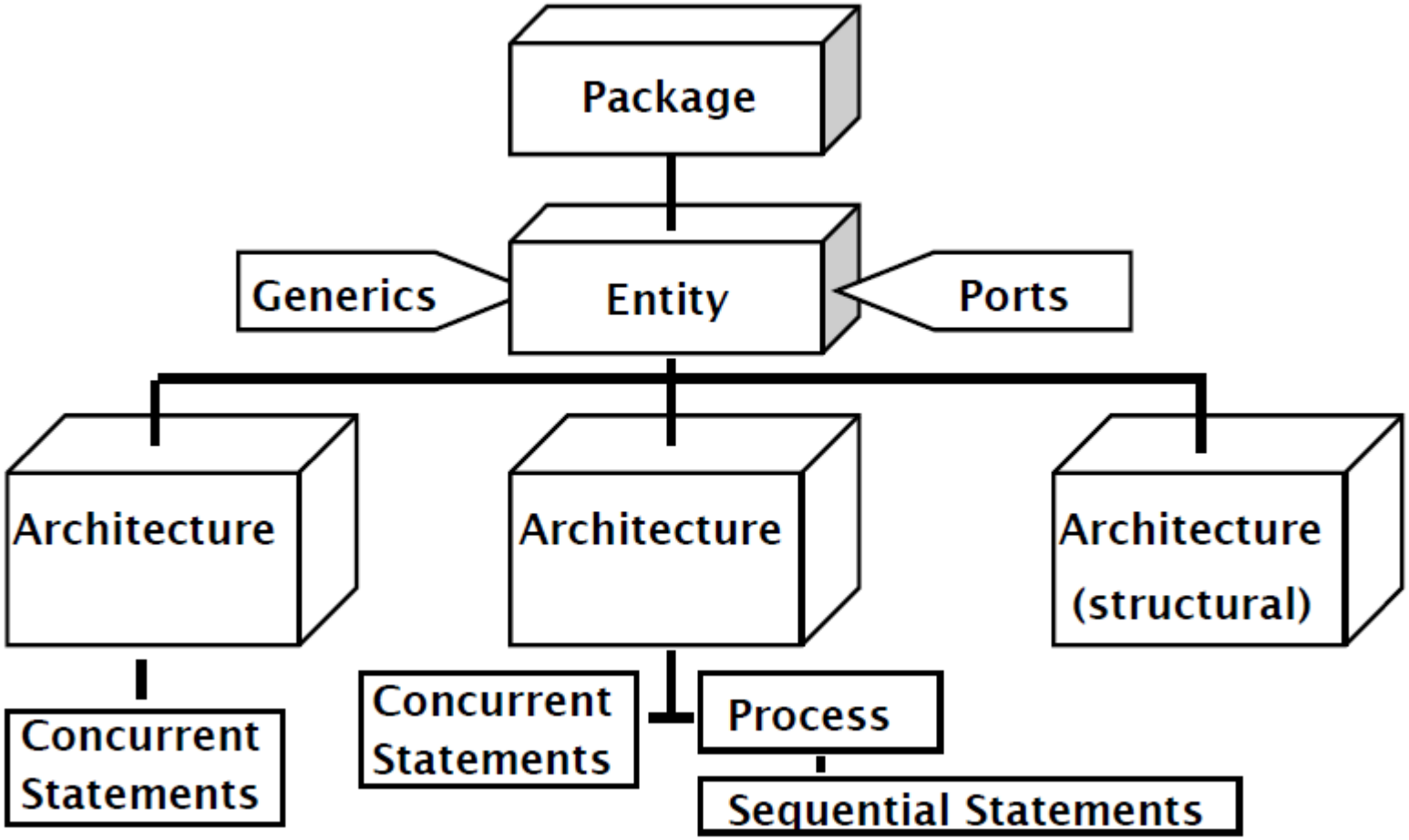
- Subtype
  - Allows for user defined constraints on a data type.
  - May include entire range of base type.
  - Assignments that are out of the subtype range result in error.
  - Subtype example

```
SUBTYPE name IS base_type RANGE <user range>;

SUBTYPE first_ten IS INTEGER RANGE 0 to 9;
```

Design Description Methods

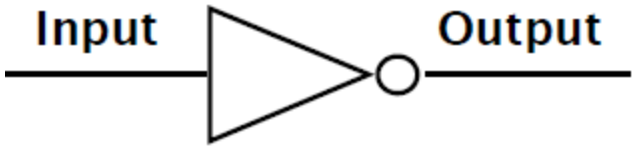
- The **architecture_statement_part** can be defined using:
  - ***Behavioral Description Method:*** *in this method we implement the algorithmic representation of circuits using conditional statements and sub-programs.*
  - ***Data-Flow Description Method:*** *in this method we define the circuits using relationship between inputs and outputs values by determining the direction of inputs and only applying original operators to them.*
  - ***Structural Description Method:*** *in this method we implement the schematic representation of circuits using fundamental gates.*
  -
- **We can define and implement several architectures for each entity, but only one may be activated at a time.**

VHDL Hierarchy



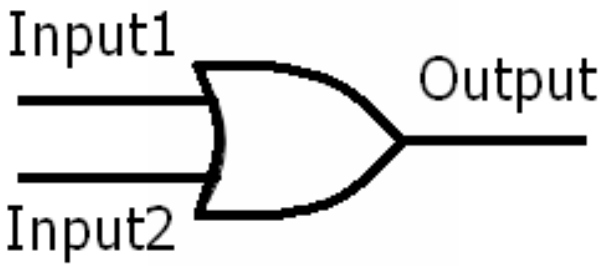
Example of Simple inverter


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Inv IS
PORT (Input :  IN STD_LOGIC;
          Output : OUT STD_LOGIC);
END Inv;
ARCHITECTURE dtf OF Inv IS
BEGIN
     Output <= not Input ;
END dtf;
```



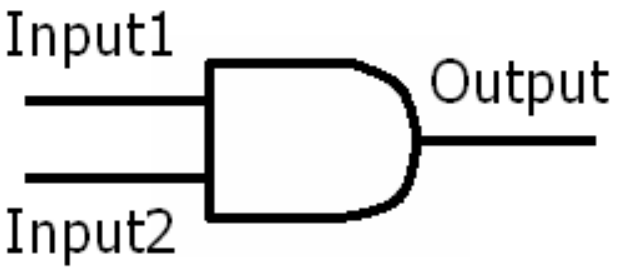
Example of Simple OR Gate


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY OR2 IS
PORT (Input1, Input2 :  IN STD_LOGIC;
          Output : OUT STD_LOGIC);
END OR2;
ARCHITECTURE dtf OF OR2 IS
BEGIN
     Output <= Input1 or Input2 ;
END dtf;
```



Example of Simple AND Gate

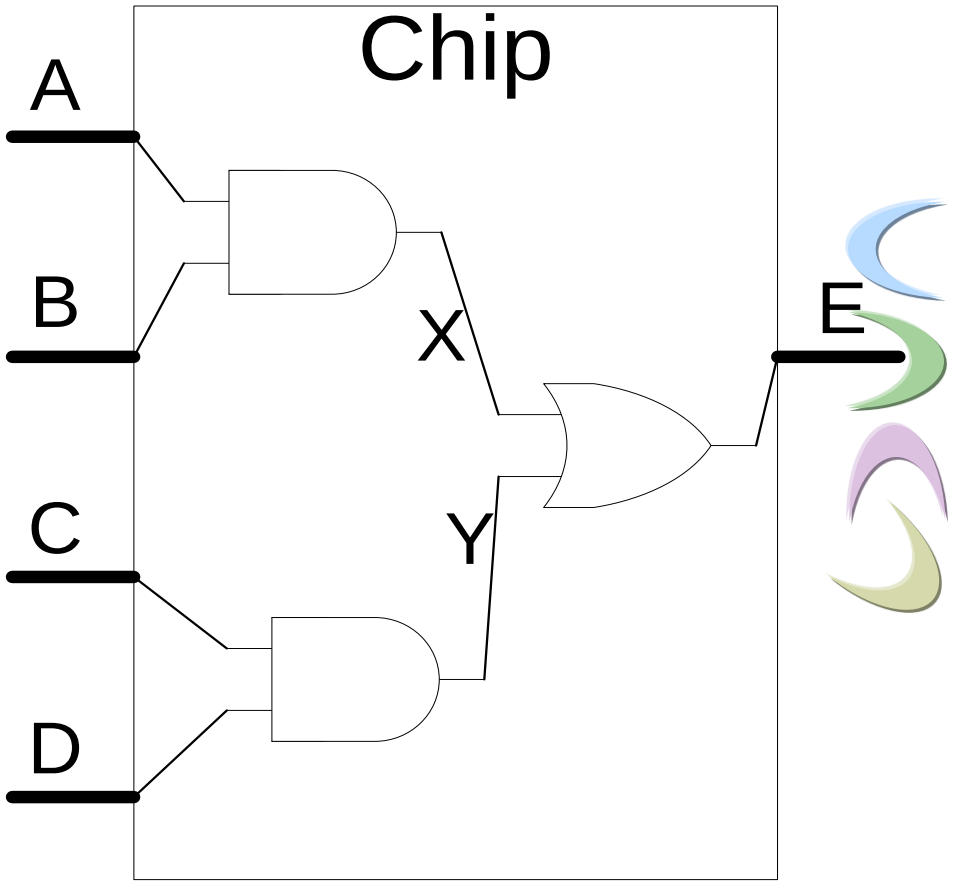

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY AND2 IS
PORT (Input1, Input2 :  IN STD_LOGIC;
          Output : OUT STD_LOGIC);
END AND2 ;
ARCHITECTURE dtf OF AND2 IS
BEGIN
     Output <= Input1 and Input2 ;
END dtf;
```



Example of Simple Chip

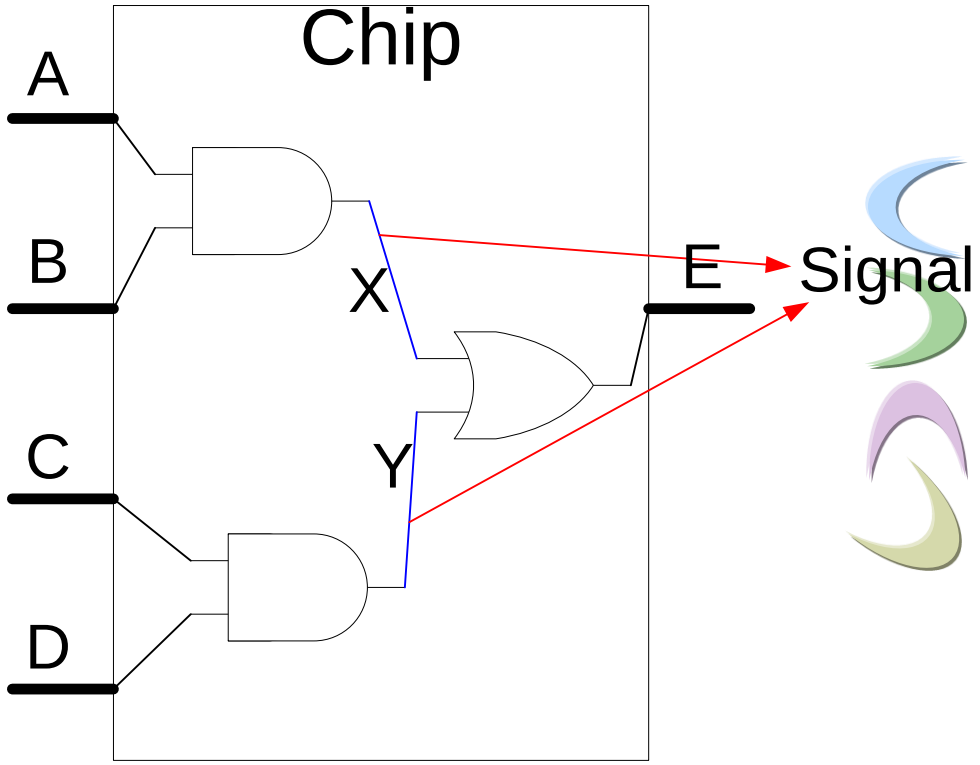
- **Define functionality of the chip**

- **X <= A AND B;**
- **Y <= C AND D;**
- **E <= X OR Y;**



Signal

- **All internal variables**
  - **Signal X,Y : std_logic;**
- Define the logic function
  - ***Output <= inputa and inputb;***
- LHS RHS
- LHS contains only 1 variable only
- RHS can be logics operations for many variables



Final code


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Chip IS
PORT (A,B,C,D :  IN STD_LOGIC;
          E           : OUT STD_LOGIC);
END Chip ;

ARCHITECTURE dtf OF Chip IS
SIGNAL X,Y : STD_LOGIC;
BEGIN

    X <= A AND B;
    Y <= C AND D;
    E <=  X OR Y;

END dtf;
```

Sequential v.s. Concurrent Statements

- VHDL provides two different types of execution: sequential and concurrent.
- Sequential statements view hardware from a "programmer" approach.
- Concurrent statements are order-independent and asynchronous.
- Different types of execution are useful for modeling of real hardware.
  - Supports various levels of abstraction.

Concurrent Signal Assignment

- The concurrent signal assignment statements are executed any time there is an event on any signal on the right hand side (RHS) of them. Each concurrent signal assignment can be conditional:

```
Output_Signal <= [Transport]
   Value1 WHEN Condition1 ELSE
          …
   ValueN WHEN ConditionN ELSE
   ValDef;
```

Example:

```
X <= A after 5 ns  when Z=0 else
       B after 10 ns when Z=1 else
       C after 10 ns when Z=2 else
       D after 15 ns;
```

WITH Statement Syntax

- The with statement is used to simplify the conditional concurrent signal assignments with same condition:

```
WITH Experssion SELECT
   Output_Signal <= [Transport]
   Value1    WHEN    Choise1,
             …
   ValueN    WHEN    ChoiseN,
   ValDef    WHEN    OTHERS;
```

Example:

```
WITH (S1+S2) SELECT
   X <= A after 5 ns when 0,
    B after 10 ns when 1,
    C after 10 ns when 2 to 19,
    D after 10 ns when others;
```

Concurrent Assert Statement

- The assert statement is used to check different statuses and conditions in the proposed model and it shows a message on the screen if its Expression is equal to FALSE:

[Label:] ASSERT Expression
REPORT "Message"
SEVERITY severity_level;

Example:
ASSERT (A or B)=C
REPORT "Warning: C is not equal to (A or B)"
SEVERITY WARNING;

- The severity level can be equal to "NOTE", "WARNING" , "ERROR" or "FAILURE", .

Concurrent BLOCK Syntax

- A basic element of VHDL is the block.
- A block is a bounded region of executable statements.
- Each architecture can be divided to several independent or nested blocks.
- There are two main reasons why this structure is employed:
  - It supports a natural form of design decomposition.
  - A guarded condition can be associated with some of block statements.

```
[Label:] BLOCK
Begin

--unguarded concurrent statements

END BLOCK [Label];
```

```
[Label:] BLOCK(Condition)
Begin
--unguarded statements
--guarded statements with Condition
END BLOCK [Label];
```

Concurrent BLOCK Example


```
A: BLOCK(C='1')
Begin
O1<=I1 --unguarded statements
O2<= guarded I2 --guarded statements with Condition (C='1')
End BLOCK A;
```

- Any guarded statement will be executed when:
  - The guard Condition (C='1') is TRUE and signal on the right hand side of guarded statement changes.
  - The guard Condition (C='1') changes from FALSE to TRUE.
- When the guard Condition (C='1') is FALSE, the left hand side signal retain its old value.
- Unguarded statements will be executed when one of its right hand side signals changes.

If Statement Syntax


```
If condition then
        sequential_statements
End if;

If condition1 then
        sequential_statements1;
Elsif condition2 then
        sequential_statements2;
        …
Elsif conditionN then
        sequential_statementsN;
Else
        default_of_statements;
End if;
```

Example

```
If  A = '0' then
    C<=B;
End if;
```

```
If  A = '0' then
    C<=B;
Elsif  A = '1' then
    C<=A;
End if;
```

CASE Statement Syntax


```
Case Experssion Is
    When value1  =>
      sequential_statements1
      …
    When valueN  =>
        sequential_statementsN;
    When others  =>
        default_statements;
End Case;
```

Example

```
Case (a&b) Is
    When "00" =>
      q <= '0';
    When "11" =>
      q <= '1';
    When others  =>
      q <= 'Z';
End Case;
```

- To combine some states use "|" operator: When 0|1|2|3 =>
- To determine a range of values use "(firstV to lastV)" statements: When (0 to 5) =>
- All possible values should be determined and if one state has no instruction use "null" statement for it.

IF v.s. CASE Statement

Example

```
If (( a = '0' ) and (b = '0' )) then
    q<='0';
Elsif (( a = '1' ) and (b = '1' )) then
    q<='1';
Else
    q<='Z';
End if;
```

Example

```
Case (a&b) Is
    When "00" =>
      q <= '0';
    When "11" =>
      q <= '1';
    When others  =>
      q <= 'Z';
End Case;
```

LOOP Statement Syntax

[looplabel:] LOOP

sequential_statements;

End LOOP;

Example

```
i:=0;
LOOP
    q(i) <= a(i)  and  b(i);
    I := I + 1;
    exit when i>9;
End LOOP;
```

- To neglect current execution of loop, use "next" statement as:
  next [looplabel] [when Condition];
- To finish the execution of loop, use "exit" statement as:
  exit [looplabel] [when Condition];

FOR Statement Syntax


```
FOR Counter in FVal  to LVal LOOP
      sequential_statements;
End LOOP;
```

Example

```
FOR i in 0  to 9 LOOP
      q(i) <= a(i)  and  b(i);
End LOOP;
```

```
FOR Counter in LVal downto FVal  LOOP
      sequential_statements;
End LOOP;
```

```
FOR i in 9  downto 0 LOOP
      q(i) <= a(i)  and  b(i);
End LOOP;
```

- To neglect current execution of for, use "next" statement as:
  next [forlabel] [when Condition];
- To finish the execution of for, use "exit" statement as:
  exit [forlabel] [when Condition];

WHILE Statement Syntax

WHILE Condition LOOP

sequential_statements;

End LOOP;

Example

```
i:=0;
WHILE i<=9 LOOP
    q(i) <= a(i) and b(i);
    i := i + 1;
End LOOP;
```

- To neglect current execution of while, use "next" statement as:
  next [whilelabel] [when Condition];
- To finish the execution of while, use "exit" statement as:
  exit [whilelabel] [when Condition];

WAIT Statement Syntax

- The wait statement causes the suspension of a process statement or a procedure.
- **wait [sensitivity_clause] [condition_clause] [timeout_clause];**
  - Sensitivity_clause ::= on signal_name
    - `wait on CLOCK;`
  - Condition_clause ::= until boolean_expression
    - `wait until Clock = '1';`
  - Timeout_clause ::= for time_expression
    - `wait for 150 ns;`
  - Complex wait statement:
    - `wait on X,Y until Z=0 for 100 ns; --wait for an --event on X or Y with Z=0 (Max wait is 100 ns)`

Process Statement Syntax

- All statements in a process occur sequentially.
- If statements are defined in a process statement.
- Processes have sensitivity list or wait statement.
- Different processes execute concurrently.

```
Process (sensitivity_list )
Begin
        sequential_statements;
End process;
```

```
Process
Begin
        sequential_statements;
        wait  statement;
End process;
```

Sensitivity-lists vs Wait-on - statement


```
Summation:
  PROCESS( A, B, Cin)
  BEGIN
    Sum <= A xor B xor Cin;
END PROCESS Summation;
```

=

```
Summation:  PROCESS
  BEGIN
    Sum <= A xor B xor Cin;
       WAIT ON A, B, Cin;
END PROCESS Summation;
```

if you put a sensitivity list in a process,
you can't have a wait statement!

if you put a wait statement in a process,
you can't have a sensitivity list!

Concurrent Process Equivalents

- All concurrent statements correspond to a process equivalent.

```
U0: q <= a xor b after 5 ns;
```

is short hand notation for

```
U0: process
begin
     q <= a xor b after 5 ns;
     wait on a, b;
end process;
```

Predefined Attributes

- Attributes allow retrieving information about types, array objects or their aliases, signals and named entities.
- VHDL standard defines a set of predefined attributes to denote values, functions, types, and ranges that characterize various VHDL entities.

Scalar type attributes

Attribute	Result type	Result
T'Left	**same as T**	**leftmost value of T**
T'Right	**same as T**	**rightmost value of T**
T'Low	**same as T**	**least value in T**
T'High	**same as T**	**greatest value in T**
T'Ascending	**boolean**	**true if T is an ascending range, false otherwise**
T'Image(x)	**string**	**a textual representation of the value x of type T**
T'Value(s)	**base type of T**	**value in T represented by the string s**

Predefined Attributes

Attributes of discrete or physical types and subtypes

Attribute	Result type	Result
T'Pos(s)	universal integer	position number of s in T
T'Val(x)	base type of T	value at position x in T (x is integer)
T'Succ(s)	base type of T	value at position one greater than s in T
T'Pred(s)	base type of T	value at position one less than s in T
T'Leftof(s)	base type of T	value at position one to the left of s in T
T'Rightof(s)	base type of T	value at position one to the right of s in T

Predefined Attributes

Attributes of the array type or objects of the array type

Attribute	Result
A'Left(n)	leftmost value in index range of dimension n
A'Right(n)	rightmost value in index range of dimension n
A'Low(n)	lower bound of index range of dimension n
A'High(n)	upper bound of index range of dimension n
A'Range(n)	index range of dimension n
A'Reverse_range(n)	reversed index range of dimension n
A'Length (n)	number of values in the n-th index range
A'Ascending(n)	True if index range of dimension n is ascending, False otherwise

Signals Attributes

Razi University

Attribute	Result
S'Delayed(t)	implicit signal, equivalent to signal S, but delayed t units of time
S'Stable(t)	implicit signal that has the value True when no event has occurred on S for t time units, False otherwise
S'Quiet(t)	implicit signal that has the value True when no transaction has occurred on S for t time units, False otherwise
S'Transaction	implicit signal of type Bit whose value is changed in each simulation cycle in which a transaction occurs on S (signal S becomes active)
S'Event	True if an event has occurred on S in the current simulation cycle, False otherwise
S'Active	True if a transaction has occurred on S in the current simulation cycle, False otherwise

Signals Attributes

Attribute	Result
S'Last_event	the amount of time since last event occurred on S, if no event has yet occurred it returns Time'High
S'Last_active	the amount of time since last transaction occurred on S, if no event has yet occurred it returns Time'High
S'Last_value	the previous value of S before last event occurred on it
S'Driving	True if the process is driving S or every element of a composite S, or False if the current value of the driver for S or any element of S in the process is determined by the null transaction
S'Driving_value	the current value of the driver for S in the process containing the assignment statement to S

Procedure Syntax

- The procedure is a form of subprograms. It contains local declarations and a sequence of statements. Procedure can be called in any place of the architecture.
- The procedure definition can be done as below:

```
procedure procedure_name (formal_parameter_list) is
 --declarations
begin
      --sequential statements;
end procedure procedure_name;
```

- Objects classes constants, variables, signals, and files can be used as formal parameters.
- There are three modes available: **in**, **out**, and **inout**
- When a procedure is called, formal parameters are substituted by actual parameters of the same class.

Function Syntax

- The function is a subprogram that either defines an algorithm for computing values or describes a behavior.
- The important feature of functions is that they are used as expressions that return values of specified type. This is the main difference from another type of subprograms: *procedures*, which are used as statements.
- The function definition can be done as below:

```
function function_name (parameters) return type is
 --declarations
begin
      --sequential statements;
      return(ret_value);
end function function_name;
```

Function Syntax

- The type of input parameters of the function can only be constant, signal or file (default is a constant).
- In case of signal parameters the attributes of the signal are passed into the function, except for 'STABLE, 'QUIET, 'TRANSACTION and 'DELAYED, which may not be accessed within the function.
- The result returned by a function can be of either scalar or complex type.
- declarations *part*, may contain nested subprograms, types, constants, variables and etc. definition.
- *function body*, which contains sequence of statements specifying the algorithm performed by the function.

Sub-Program Syntax

- Subprograms (functions and procedures) can be called recursively.
- Subprograms (functions and procedures) body may not contain a wait statement or a signal assignment.
- Subprograms (functions and procedures) can be nested.
- When a Subprogram (function and procedure) is called, formal parameters are substituted by actual parameters of the same class.

Function and Procedure Example


```
function FPOW (constant A,B,X: real) return real is
begin
return (A*X**2+B);
end FPOW;
```

The value returned by this function is the A*X**2+B calculated from the formal constant parameters A, B and X.

```
procedure PPOW (constant A,B,X: in real; variable PO: out real) is
begin
PO:= (A*X**2+B);
end PPOW;
--Example of calling Functions and Procedures
T:=FPOW(3,1,5);

PPOW(3,1,5,T);
```

Function and Procedure Example


```
function Transcod(Value: in bit_vector(0 to 7)) return bit_vector is
begin
        case Value is
                when "00000000" => return "01010101";
                when "01010101" => return "00000000";
                when others => return "11111111";
        end case;
end Transcod;
```

The case statement has been used to realize the function algorithm. The formal constant parameter Value has the Bit_vector type. This function returns a value of the same type.

```
procedure Transcoder(variable Value: inout bit_vector(0 to 7)) is
begin
        case Value is
                when "00000000" => Value:=" 01010101";
                when "01010101" => Value:=" 00000000";
                when others => Value:=" 11111111";
        end case;
end Transcoder;
```

Package Syntax

- To prevent the repetition of declarations (like sub type definition and sub-programs) that frequently used, we can implement them in a package file.

- The package definition can be done as below:

```
package package_name is
 -- sub type definition;
 -- sub-programs prototyping;
end package _name;
package body package_name is
 -- sub-programs implementation;
 end package_name;
```

- We simply can add this package to our declaration with USE statement:

```
Use work. package_name.all;
```

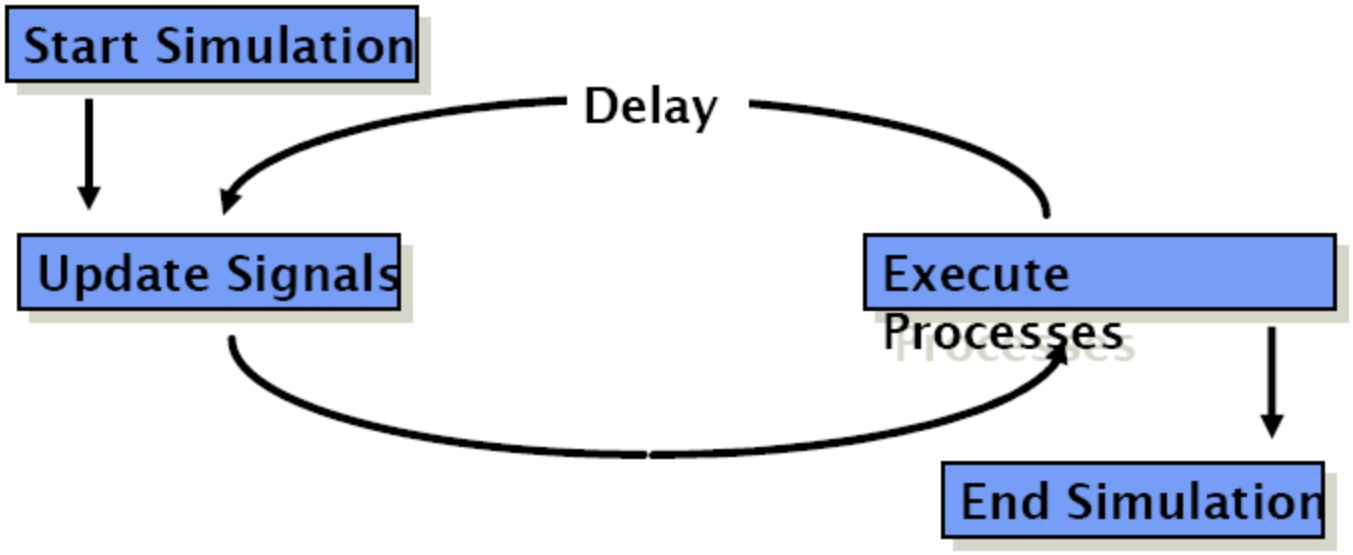
Package Example


```
package mypack is
 subtype BITVEC16 is BIT_VECTOR(o to 15);
 subtype BITVEC32 is BIT_VECTOR(o to 31);
 Function MAJ3(X:BIT_VECTOR(0 to 2)) return BIT;
 Function INTVAL(X:BIT_VECTOR) return INTEGER;
 -- Other sub-programs prototyping;
end mypack;
package body mypack is
 Function MAJ3(X:BIT_VECTOR(0 to 2)) return BIT is
 Begin
      return ( (X(0) and  X(1)) or (X(0) and X(2))  or (X(1) and X(2)) );
 End MAJ3;
Function INTVAL(X:BIT_VECTOR) return INTEGER is
Variable SUM:INTEGER:=0;
 Begin
      for I in X'LOW to X'HIGH loop
         if (X(I)='1') then
              SUM := SUM + (2**I);
         end if;
      end loop;
      return (SUM);
 End INTVAL;
 -- Other sub-programs implementation
end mypack;
```

VHDL modeling and Simulation

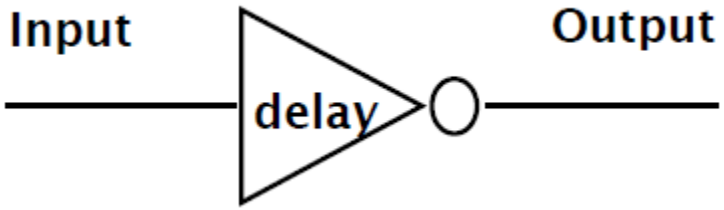
VHDL Simulation Cycle

- VHDL uses a simulation cycle to model the stimulus and response nature of digital hardware.



VHDL Delay Models

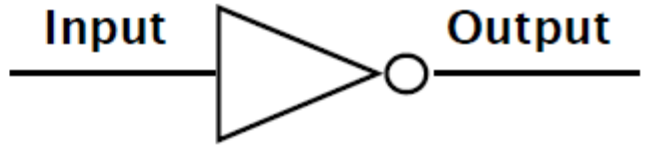
- Delay is created by scheduling a signal assignment for a future time.
- Delay in a VHDL cycle can be of several types
  - Inertial
  - Transport
  - Delta

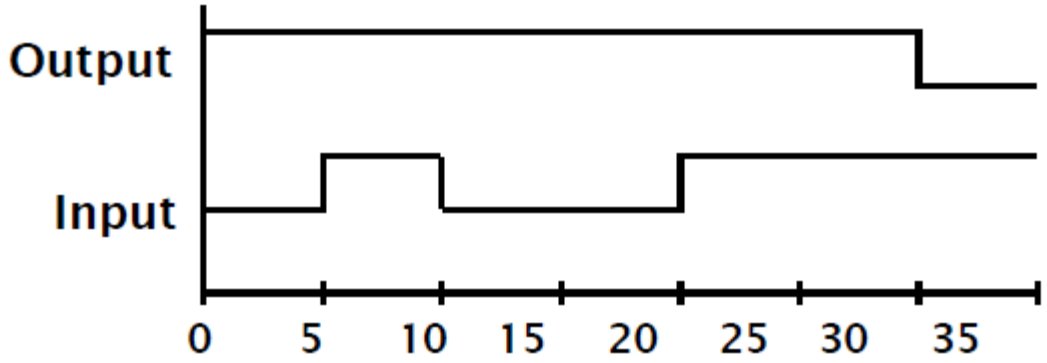


Inertial Delay

- Default delay type
- Allows for user specified delay
- Absorbs pulses of shorter duration than the specified delay

```
-- Inertial is the default
Output <= NOT Input AFTER 10 ns;
```

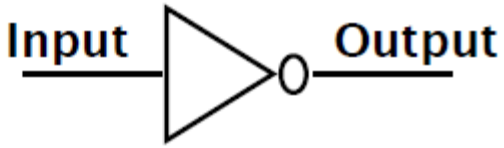


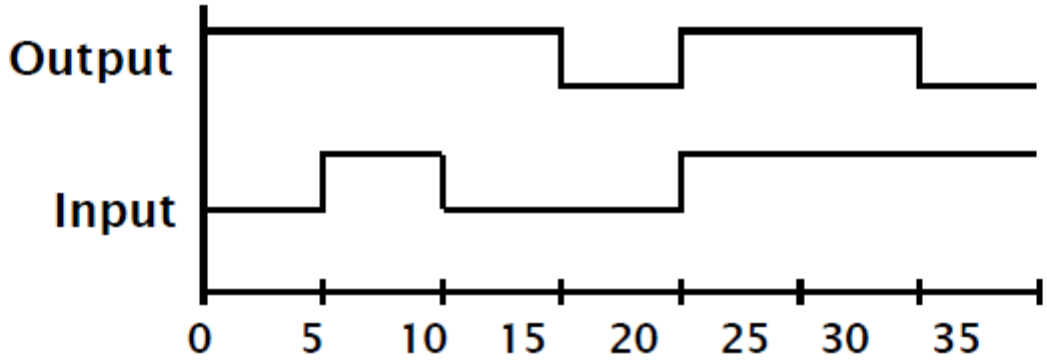


Transport Delay

- Must be explicitly specified by user
- Allows for user specified delay
- Passes all input transitions with delay

```
-- TRANSPORT must be specified
Output <= TRANSPORT NOT Input AFTER 10 ns;
```





Delta Delay

- Delta delay needed to provide support for concurrent operations with zero delay
  - The order of execution for components with zero delay is not clear
- Scheduling of zero delay devices requires the delta delay
  - A delta delay is necessary if no other delay is specified
  - A delta delay does *not advance simulator time*
  - One delta delay is an infinitesimal amount of time
  - The delta is a scheduling device to ensure repeatability

```
Process(X)
 begin
        Y <= X;
        Z <= Y;
end process;
```

The new value of Y will not be copied to Z until another event on X

```
Process(X,Y)
 begin
        Y <= X;
        Z <= Y;
end process;
```

The new value of Y will be copied to Z after delta delay

Usage of the 'after' for synthesis

- This description method is not supported by the synthesis tools
- Other rule, that the delay after the after command must be in ascending order

❖ Incorrect code:

❖ c <= '1',
'0' after 20 ns,
'1' after 10 ns; --Error

❖ Correct code:

❖ c <= '1',
'0' after 10 ns,
'1' after 20 ns;

VHDL Design Approaches

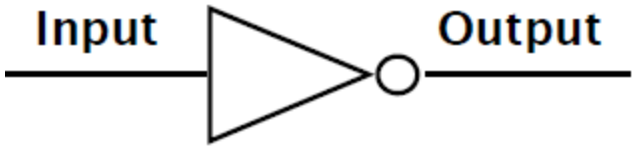
Three design approaches can be used when writing VHDL code:

- Structural: in this approach we design a circuit using concatenating of existing components (like schematic design).
- Dataflow: in this approach we only use assignment statements (conditional or unconditional signal assignment) for designing circuit by determining the relation between its inputs and outputs.
- Behavioural: in this approach we use algorithmic statements (like if-else, case, loops, sub-programs and etc.) for designing circuits.

- All design approaches can be synthesized.
- Often, however, the behavioural code is written in such a way that it can only be simulated and not synthesized.

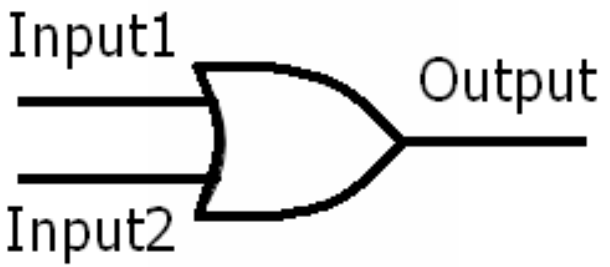
Example of Simple inverter


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Inv IS
PORT (Input :  IN STD_LOGIC;
          Output : OUT STD_LOGIC);
END Inv;
ARCHITECTURE dtf OF Inv IS
BEGIN
     Output <= not Input ;
END dtf;
```



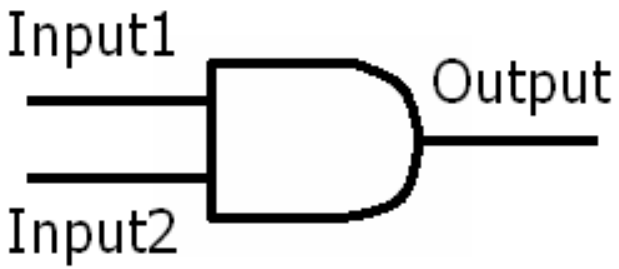
Two Inputs OR Gate


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY OR2 IS
PORT (Input1, Input2 :  IN STD_LOGIC;
          Output : OUT STD_LOGIC);
END OR2;
ARCHITECTURE dtf OF OR2 IS
BEGIN
     Output <= Input1 or Input2 ;
END dtf;
```



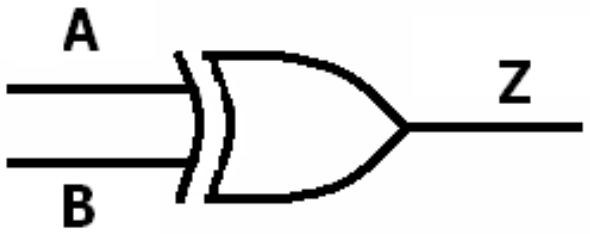
Two Inputs AND Gate


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY AND2 IS
PORT (Input1, Input2 :  IN STD_LOGIC;
          Output : OUT STD_LOGIC);
END AND2 ;
ARCHITECTURE dtf OF AND2 IS
BEGIN
     Output <= Input1 and Input2 ;
END dtf;
```



Behavioral and Data Flow of XOR Gate


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY XOR2 IS
PORT (A, B :  IN STD_LOGIC;
          Z : OUT STD_LOGIC);
END XOR2 ;
ARCHITECTURE BH OF XOR2 IS
BEGIN
    if (A=B) then
          Z<='0';
    else
          Z = '1';
    end if;
END BH;
```




```
ARCHITECTURE dtf OF XOR2 IS
BEGIN
      Z <= A xor B ;
END dtf;
```

Structural Description

Structural Description has four steps:

- Individual Components Description: in this step we describe each component using basic statements of VHDL.
- Components Declaration: in this step we prototype all needed components by determining their port descriptions.
- Components Specification: in this step we determine the entity and architecture of all prototyped components.
- Components Instantiation: in this step we create all needed components and connect their inputs and outputs to the local signals.

Component Declaration

- To use or instance one VHDL component in another structural declaration, it first has to be declared.
- This is done in the concurrent declaration part of the architecture
- Syntax:

```
component <cmp_name>
    [generic(<generic-association-list>);]
        port(<port-association-list>);
    end component;
```

- Port_list in the component declaration must be identical to that in the component's entity.
- The best way to guarantee this is copying the entity to the component declaration or vice versa.
- *Generic* parameters do not need to be declared in the component declaration.

Component Specification and Instantiation

- In the component specification statements we determine the entity and selected architecture of each declared component as below:

```
FOR  ALL:  cmp_name USE ENTITY entity_name(arc_name);
```

or

```
FOR instantiation_label :  cmp_name USE ENTITY entity_name(arc_name);
```

- Component instantiation statement are given after the begin of the main architecture body as below:

```
begin
        …
        instantiation _label :  cmp_name
                generic map([ list of generic values]) ;
                port map ([list of local signals]) ;
        …
```

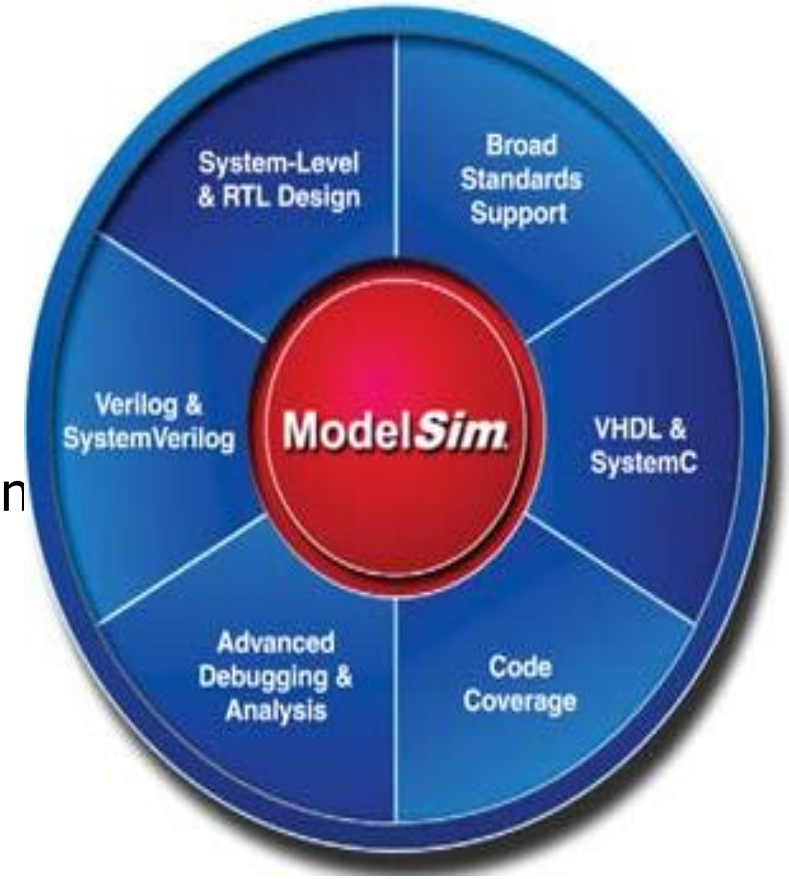
Structural XOR Gate


```
ARCHITECTURE STR OF XOR2 IS
-- Component declaration
component InvC
PORT (Input :  IN STD_LOGIC;
            Output : OUT STD_LOGIC);
End component;
component OR2C
PORT (Input1, Input2 :  IN STD_LOGIC;
            Output : OUT STD_LOGIC);
End component;
component AND2C
PORT (Input1, Input2 :  IN STD_LOGIC;
            Output : OUT STD_LOGIC);
End component;
-- Component specification
For all: InvC use entity Inv(dft);
For all: OR2C use entity OR2(dft);
For all: AND2C use entity AND2(dft);
```

```
-- Internal Signal Definition
Signal NA,NB,A1,A2:std_logic;

BEGIN
     -- Component instantiation
    N1: InvC          port map(A,NA);
    N2: InvC          port map(B,NB);
    AN1:AND2C  port map(NA,B,A1);
    AN2:AND2C  port map(NB, A,A2);
    OR1:OR2C  port map(A1,A2,Z);
END STR;
```

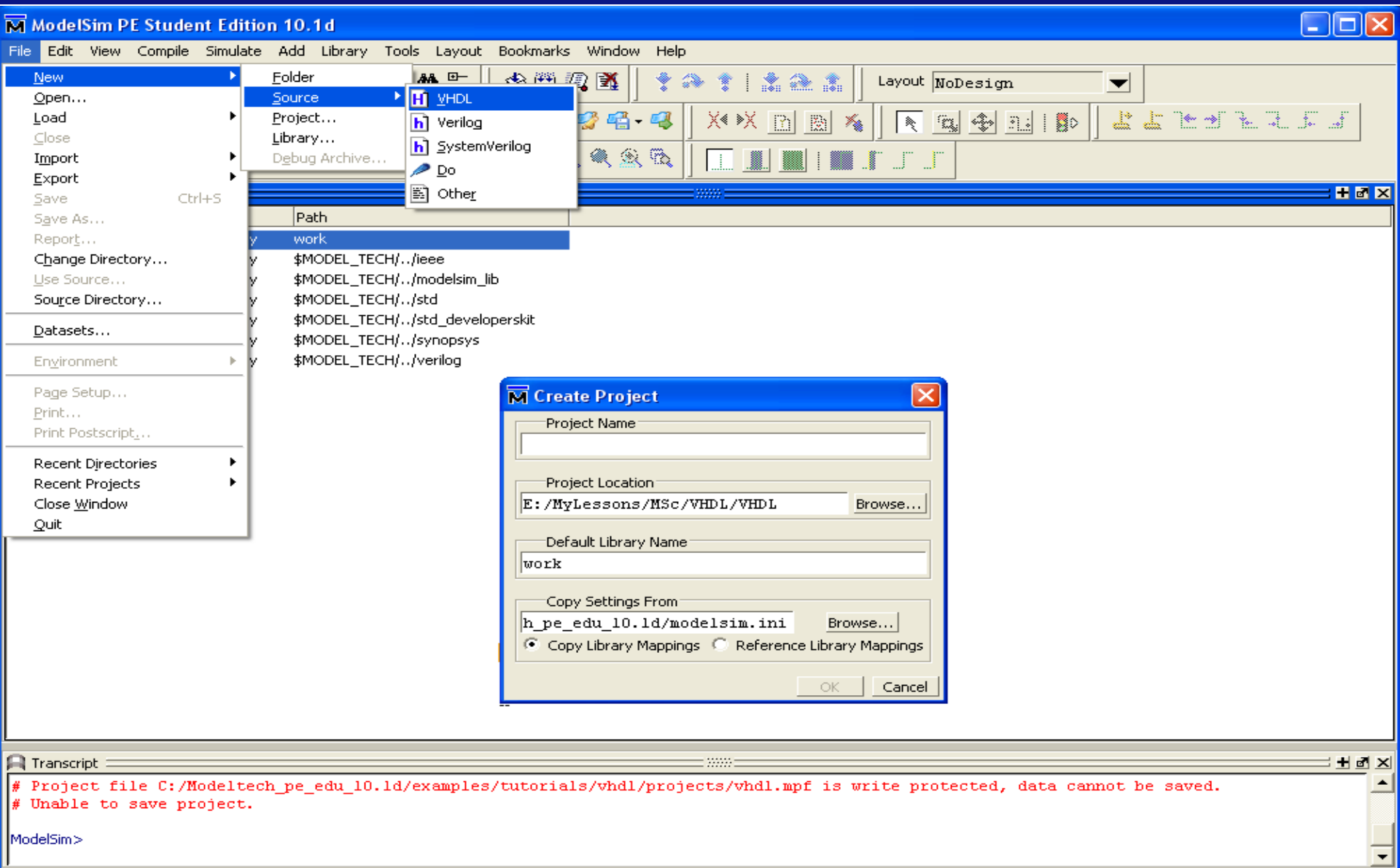
ModelSim is the industry-leading, Windows-based simulator for VHDL, Verilog, or mixed-language simulation environments. Its Features are:

- Partial VHDL 2008 support
- Windows7 Support
- SecureIP support
- SystemC option
- RTL and Gate-Level Simulation
- Integrated Debug
- Verilog, VHDL and SystemVerilog Design
- Mixed-HDL Simulation option
- Code Coverage option
- Enhanced debug option
- Windows 32-bit



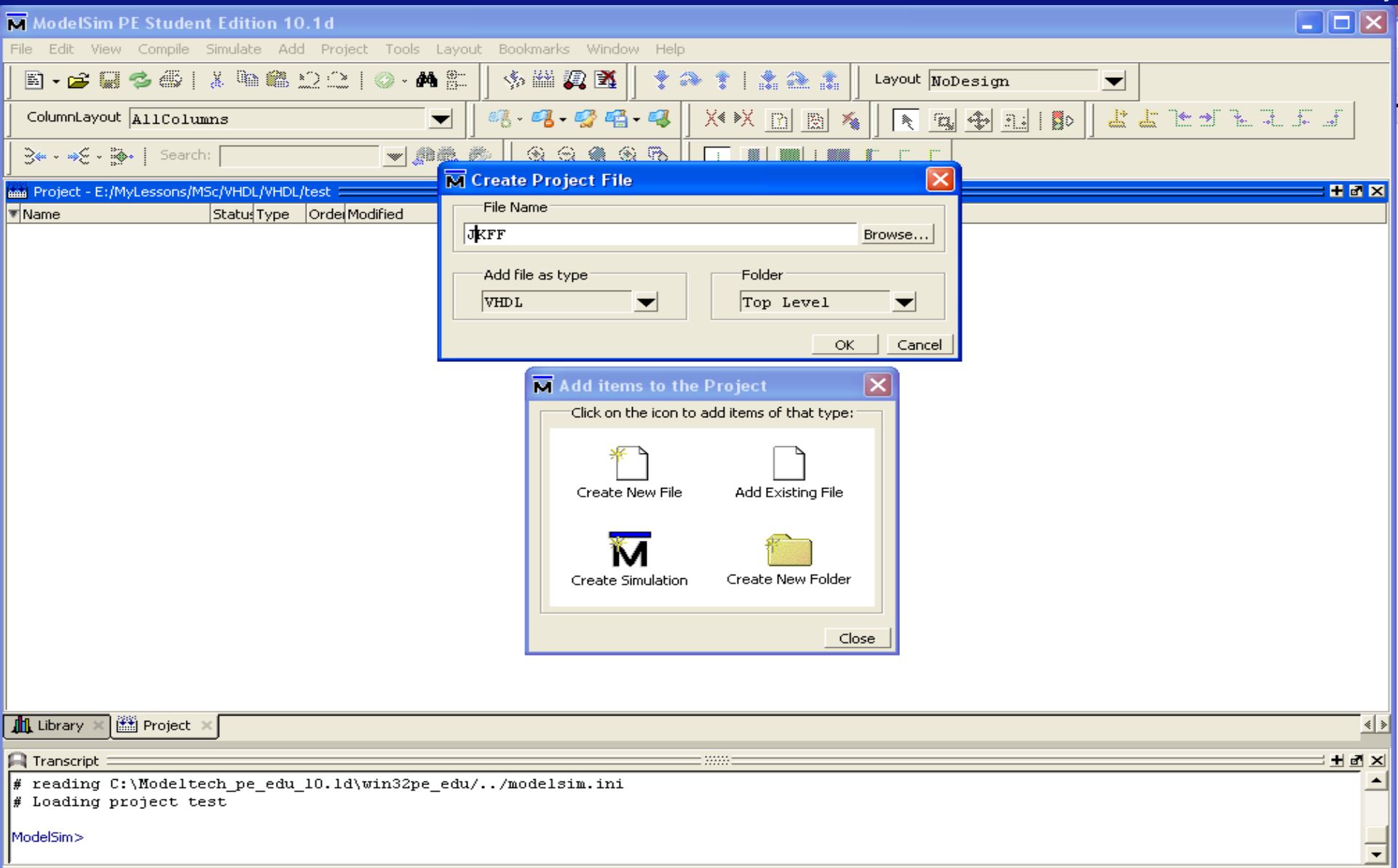
Creating Project

Razi University



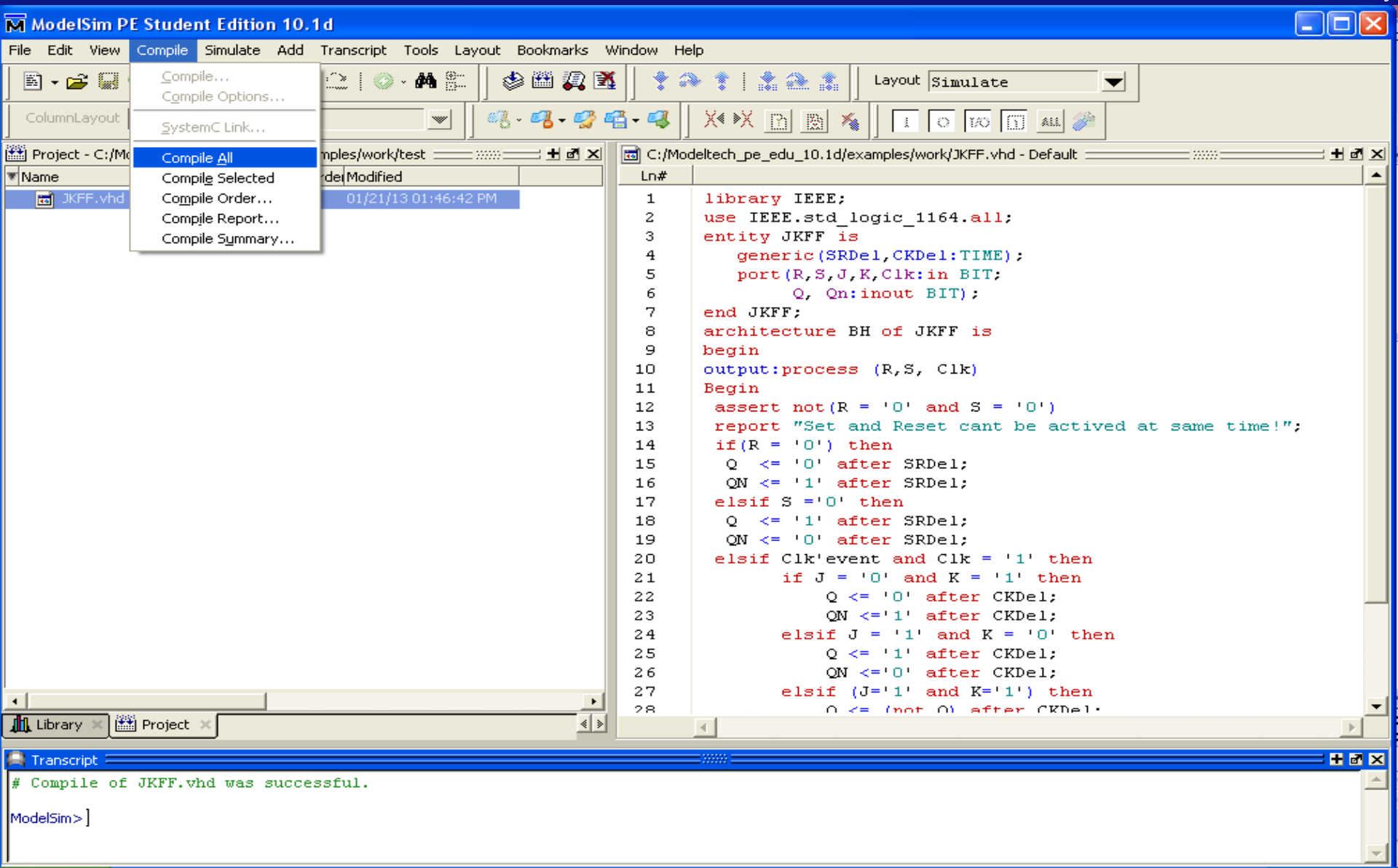
Creating VHDL Files

Razi University



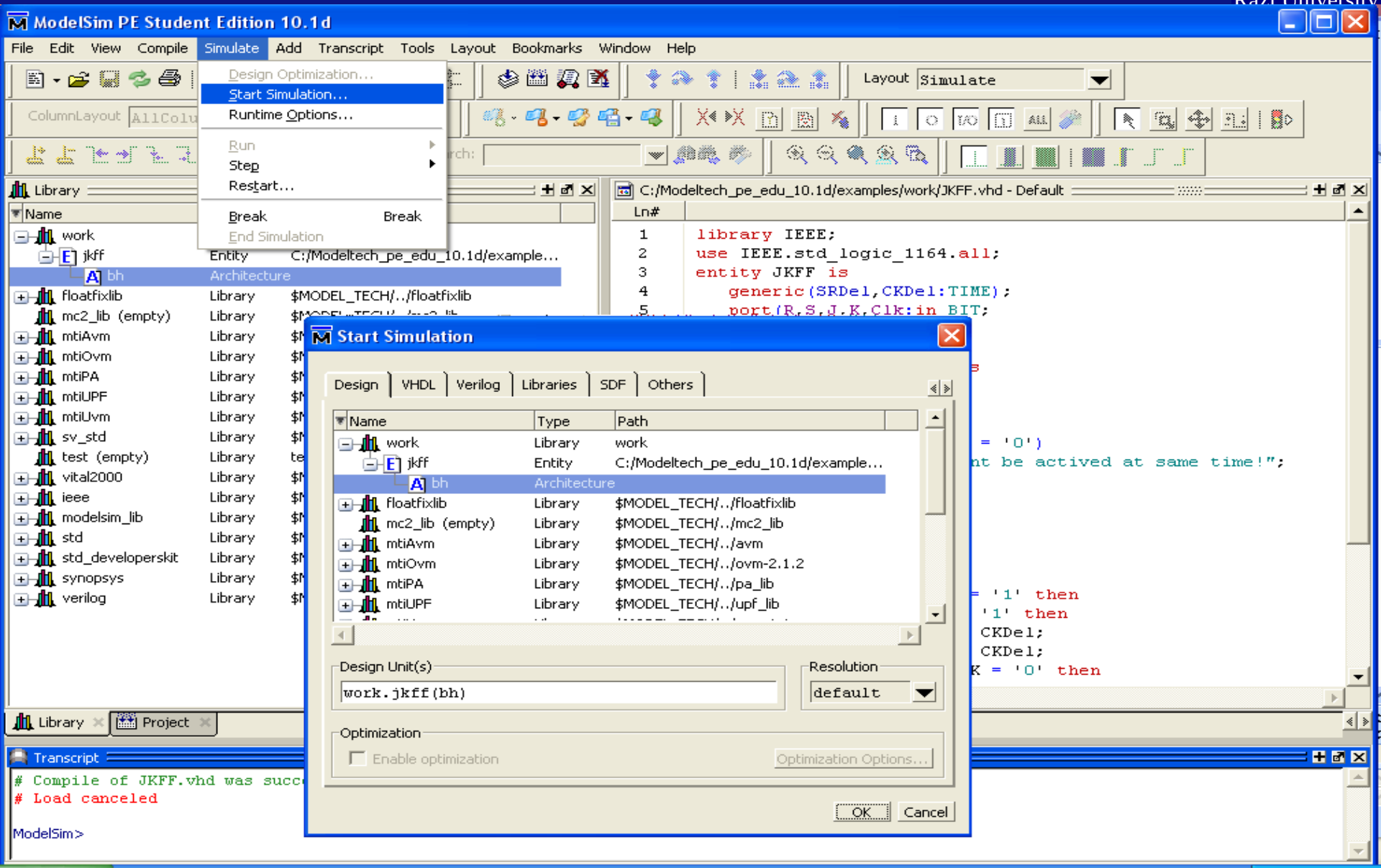
Compiling Project

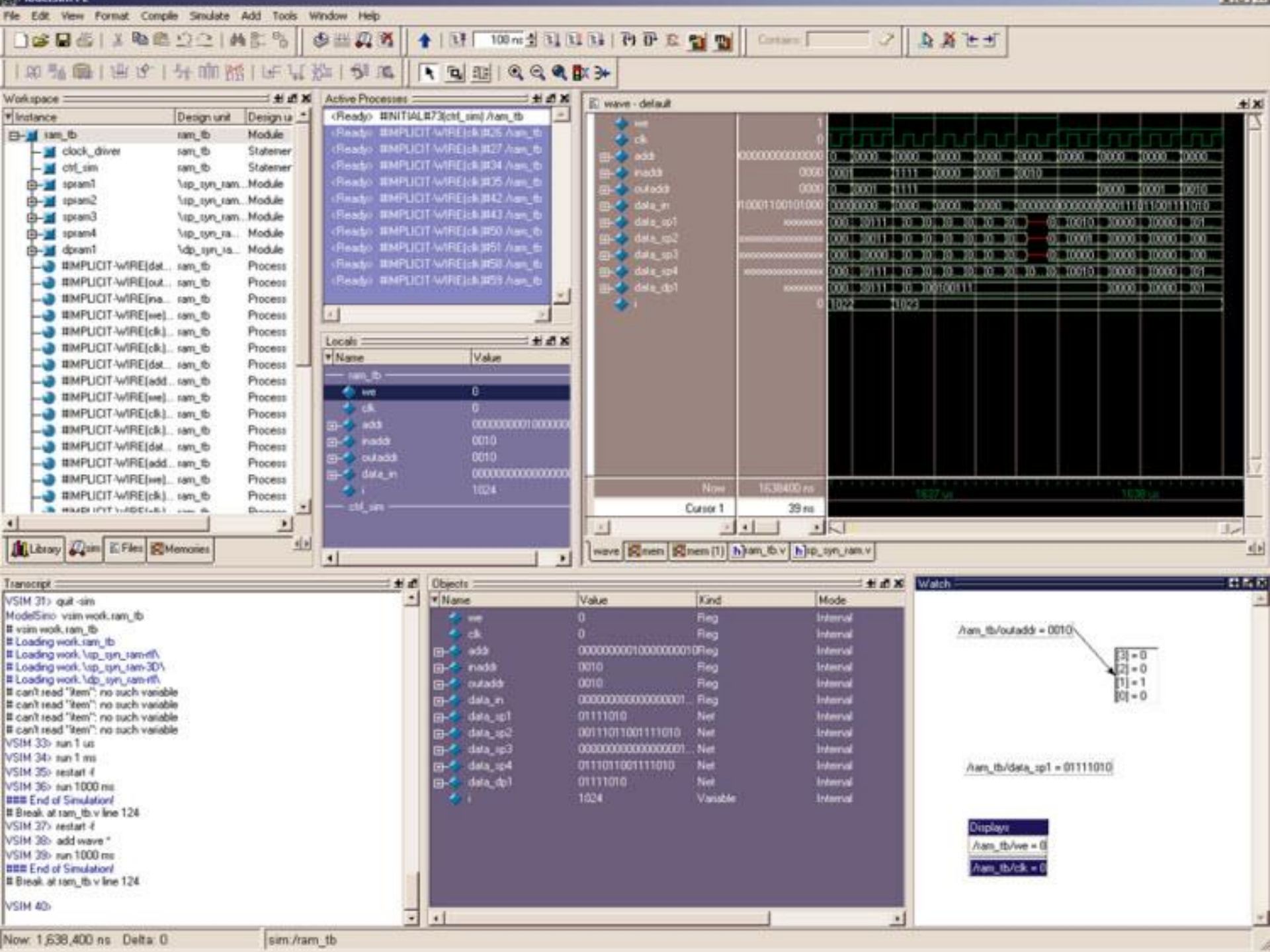
Razi University



Simulating Project

Razi University



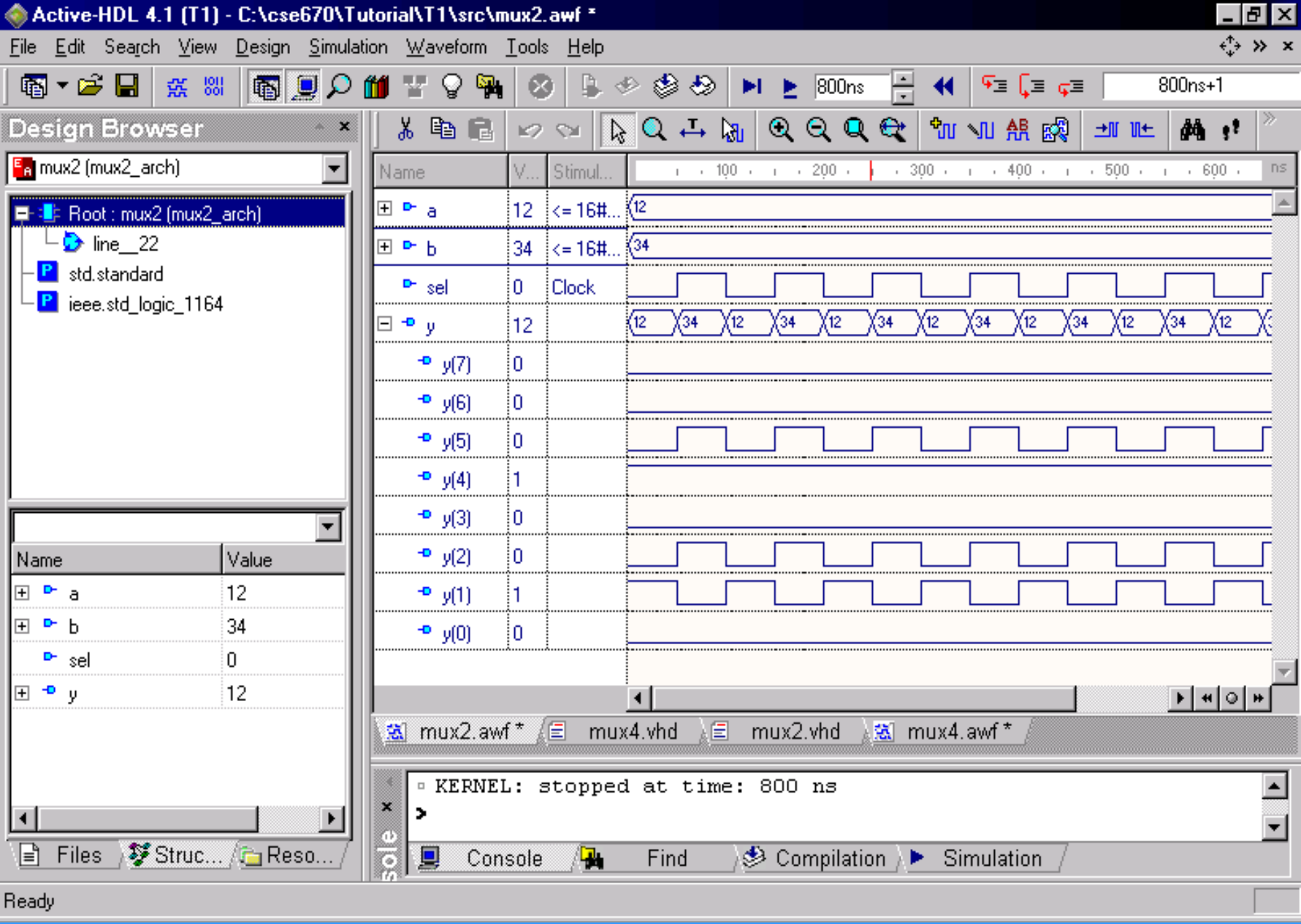

File Edit View Format Compile Simulate Add Tools Window Help
Workspace
Instance
Design unit
ram_tb
clock_driver
ctrl_sim
spram1
spram2
spram3
spram4
dpram1
Active Processes
Locals
Name
Value
wave - default
Now
Cursor 1
Transcript
VSIM 31> quit -sim
ModelSim> vsim work.ram_tb
vsim work.ram_tb
Loading work.ram_tb
Loading work.\sp_syn_ram-rtl\
Loading work.\sp_syn_ram-3D\
Loading work.\dp_syn_ram-rtl\
can't read "item": no such variable
VSIM 33> run 1 us
VSIM 34> run 1 ms
VSIM 35> restart -f
VSIM 36> run 1000 ns
End of Simulation!
Break at ram_tb.v line 124
VSIM 37> restart -f
VSIM 38> add wave *
VSIM 39> run 1000 ns
End of Simulation!
Break at ram_tb.v line 124
VSIM 40>
Objects
Kind
Mode
Watch
/ram_tb/outaddr = 0010
/ram_tb/data_sp1 = 01111010
Displays
/ram_tb/we = 0
/ram_tb/clk = 0
Now: 1,638,400 ns Delta: 0
sim:/ram_tb

mux2.vhd - HDL Editor

File Edit Search View Synthesis Project Tools Help

```
1  -- An 8-line, 2 to 1 multiplexer
2  --------------------------------
3  -- Written by Richard E. Haskell
4  -- Oakland University
5  -- email: haskell@oakland.edu
6  --------------------------------
7  library IEEE;
8  use IEEE.std_logic_1164.all;
9
10 entity mux2 is
11     port (
12         a: in STD_LOGIC_VECTOR (7 downto 0);
13         b: in STD_LOGIC_VECTOR (7 downto 0);
14         sel: in STD_LOGIC;
15         y: out STD_LOGIC_VECTOR (7 downto 0)
16     );
17 end mux2;
18
19 architecture mux2_arch of mux2 is
20 begin
21   process (sel, a, b)
22   begin
23     if sel = '0' then
24       y <= a;
25     else
26       y <= b;
27     end if;
28   end process;
29 end mux2_arch;
```

Ready Ln 6, Col 33 VHDL NUM


Active-HDL 4.1 (T1) - C:\cse670\Tutorial\T1\src\mux2.awf *
File Edit Search View Design Simulation Waveform Tools Help
800ns
800ns+1
Design Browser
mux2 (mux2_arch)
Root : mux2 (mux2_arch)
line__22
std.standard
ieee.std_logic_1164
Name Value
a 12
b 34
sel 0
y 12
Files Struc... Reso...
Name V... Stimul...
100 200 300 400 500 600 ns
a 12 <= 16#...
b 34 <= 16#...
sel 0 Clock
y 12
y(7) 0
y(6) 0
y(5) 0
y(4) 1
y(3) 0
y(2) 0
y(1) 1
y(0) 0
mux2.awf * mux4.vhd mux2.vhd mux4.awf *
KERNEL: stopped at time: 800 ns
>
Console Find Compilation Simulation
Ready

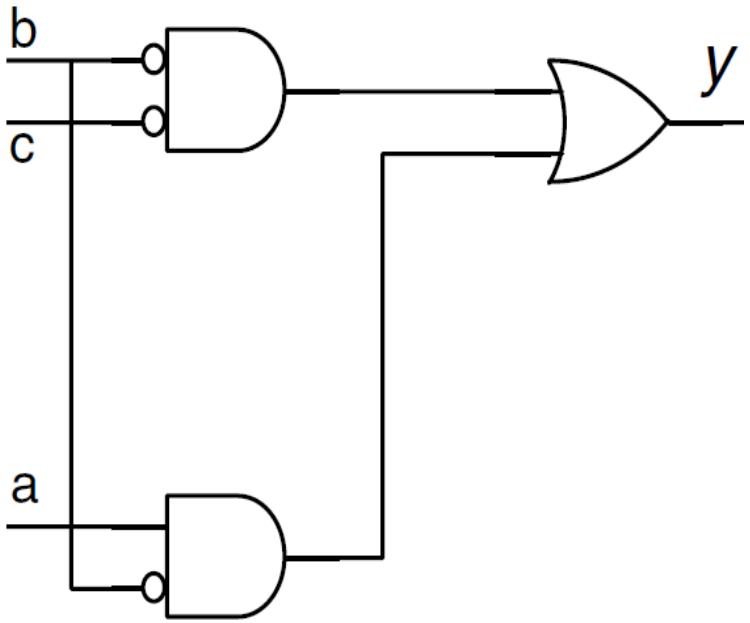
Combinational Circuit Design

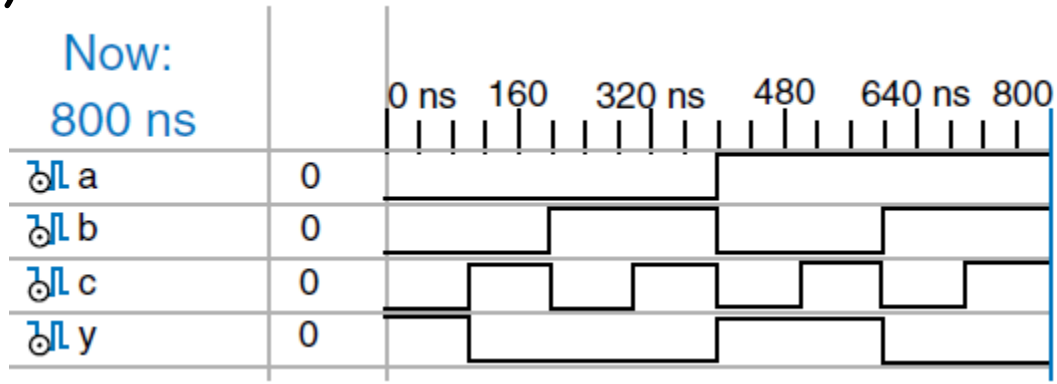
Combinational Circuit Example

- Simple Combinational circuit
- Full-Adder
- Encoders
- Decoders
- Multiplexers
- Bus Driver
- 32 Bits Adder/Subtractor
- TestBench

Simple Combinational Circuit

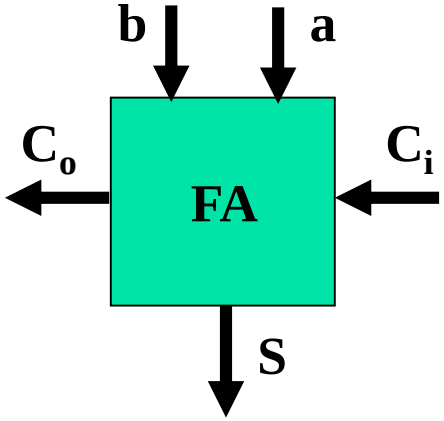

```
library IEEE;
use IEEE.std_logic_1164.all;

entity CLC is
    port (a,b,c: in STD_LOGIC;
          y: out STD_LOGIC);
end CLC;
architecture dtf of CLC is
begin
Y<=((not b) and (not c))
   or (a and (not b));
End dtf;
```





Full Adder


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

entity FA is
    generic(
        SDEL,CDEL:  Time;
    );
    port(
        a:  in STD_LOGIC;
        b:  in STD_LOGIC;
        Ci: in STD_LOGIC;
        S:  out STD_LOGIC;
        Co: out STD_LOGIC
    );
end FA ;
```



a	b	C_i	S	C_o
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

Behavioral Architecture of Full Adder

Razi University

```
architecture BH of FA is
begin
  F1: process(a, b, Ci)
      begin
         if ((b = '1')and (a='1')) then
            Co <= '1';
            S  <= Ci;
         elsif ((Ci = '1')and (a='1')) then
            Co <= '1';
            S  <= b;
         elsif ((Ci = '1')and (b='1')) then
            Co <= '1';
            S  <= a;
         else
            Co <= '0';
            S  <= a or b or Ci;
         end if;
      end process F1;
end BH;
```

Dataflow Architecture of Full Adder


```
Architecture dtf of FA is
begin
    S <= a xor b xor Ci after SDEL;
    Co<= (a and b) or (a and Ci) or
         (b and Ci) after CDEL;
end dtf;
```

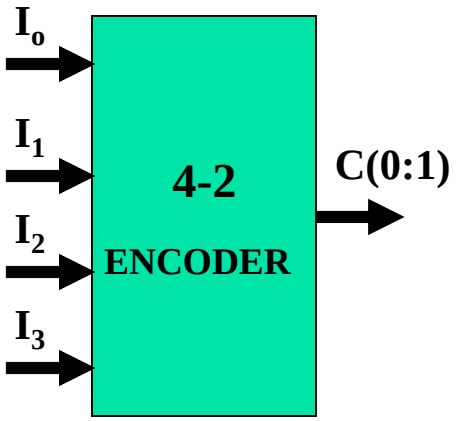
Structural Architecture of Full Adder

Razi University

```
Architecture str of FA is
component AND2C port (a,b: in std_logic;
                      c: out std_logic); end component;
component OR2C port (a,b: in std_logic;
                     c: out std_logic); end component;
component XORC port (a,b: in std_logic;
                     Z: out std_logic); end component;
signal i1,i2,i3,i4, i5:std_logic;
for all: XORC Use Entity work.XOR2(dtf);
for all: AND2C Use Entity work.AND2(dtf);
for all: OR2C Use Entity work.OR2(dtf);
begin
  U1: XORC port map(a,b,i1);
  U2: XORC port map(Ci,il,S);
  U3: AND2C port map(a,b,i2);
  U4: AND2C port map(Ci,b,i3);
  U5: AND2C port map(a,Ci,i4);
  U6: OR2C port map(i2,i3,i5);
  U7: OR2C port map(i4,i5,Co);
end;
```

4-to-2 Binary Encoder


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




```
entity Encoder4 is
    port(
        I0: in STD_LOGIC;
        I1: in STD_LOGIC;
        I2: in STD_LOGIC;
        I3: in STD_LOGIC;
        C: out STD_LOGIC_VECTOR(1 downto 0)
);
end Encoder4 ;
```

I_0	I_1	I_2	I_3	C_1	C_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

4-to-2 Binary Encoder


```
ARCHITECTURE DTF OF Encoder4 IS
BEGIN
    C <=   "00" WHEN I0='1' ELSE
           "01" WHEN I1='1' ELSE
           "10" WHEN I2='1' ELSE
           "11";
END DTF ;

ARCHITECTURE Behavior OF Encoder4 IS
BEGIN
   PROCESS(I0,I1,I2,I3)
   BEGIN
    IF (I0='1')THEN     C <= "00";
    ELSIF (I1='1')THEN C <= "01";
    ELSIF (I2='1')THEN C <= "10";
    ELSE   C <= "11";
    END IF;
   END PROCESS;
END Behavior ;
```

Another Implementation of Encoder


```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY PriorityEncoder IS
        PORT (      I       : IN        STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                    C       : OUT       STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                    z       : OUT       STD_LOGIC ) ;
END PriorityEncoder;
ARCHITECTURE Behavior OF PriorityEncoder IS
BEGIN
        PROCESS (I)
        BEGIN
                IF I(3) = '1' THEN
                        C <= "11" ;
                ELSIF I(2) = '1' THEN
                        C <= "10" ;
                ELSIF I(1) = '1' THEN
                        C<= "01" ;
                ELSE
                        C<= "00" ;
                END IF ;
        END PROCESS ;

        z <= '0' WHEN I = "0000" ELSE '1' ;
END Behavior ;
```

Process will be executed when value of "I" change

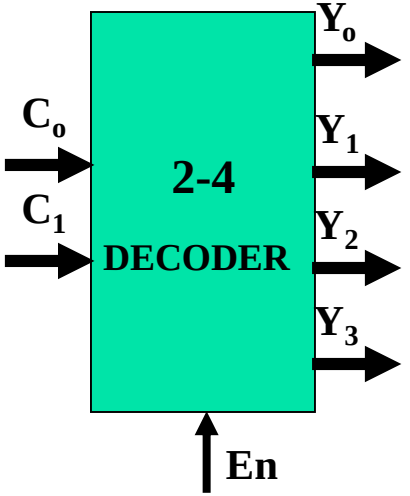
Inside process, statements executed in sequential

Executed in concurrent

2-to-4 Binary Decoder


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

entity Decoder4 is
  port(
    En: in STD_LOGIC;
    C0: in STD_LOGIC;
    C1: in STD_LOGIC;
    Y: out STD_LOGIC_VECTOR(3 downto 0)
  );
end Decoder4 ;
```

En	C_1	C_0	Y_3	Y_2	Y_1	Y_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0



2-to-4 Binary Decoder


```
ARCHITECTURE DTF OF Decoder4 IS
    SIGNAL EnC : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    EnC <= En & C1 & C0 ;
    WITH EnC SELECT
          Y <=      "0001"   WHEN "100",
                    "0010"   WHEN "101",
                    "0100"   WHEN "110",
                    "1000"   WHEN "111",
                    "0000"   WHEN OTHERS ;
END DTF ;
```

Concatenation:
EnC(2) <= En;
EnC(1) <= C(1);
EnC(0) <= C(0);

"y" will be assigned with different values based on value of "EnC"

Another Implementation of Binary Decoder

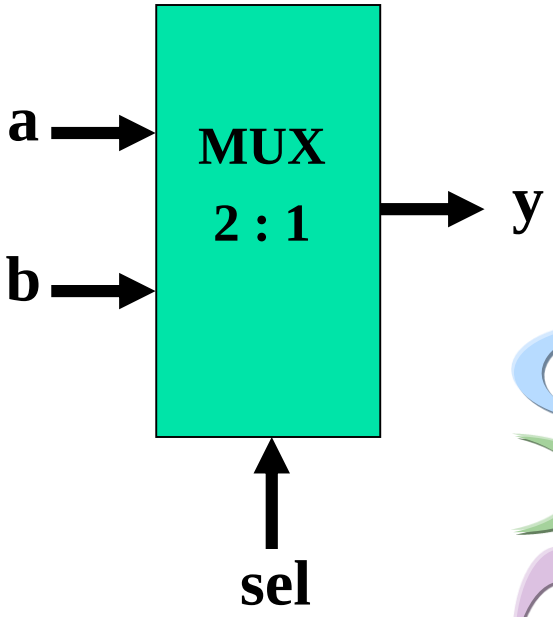

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY dec2to4 IS
        PORT (      C      : IN        STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                    En     : IN        STD_LOGIC ;
                    y      : OUT       STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
        PROCESS ( C, En )
        BEGIN
                IF En = '1' THEN
                        CASE C IS
                                WHEN "00" =>        y <= "1000" ;
                                WHEN "01" =>        y <= "0100" ;
                                WHEN "10" =>        y <= "0010" ;
                                WHEN OTHERS =>      y <= "0001" ;
                        END CASE ;
                ELSE
                        y <= "0000" ;
                END IF ;
        END PROCESS ;
END Behavior ;
```

Process will be executed when value of “C” or “En” change

“y” assigned with different value based on value of “C”

2-to-1 Multiplexer


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

entity mux2 is
    port (
        a: in STD_LOGIC;
        b: in STD_LOGIC;
        sel: in STD_LOGIC;
        y: out STD_LOGIC
    );
end mux2;
```



sel	y
0	a
1	b

Behavioral Architecture of 2:1 Mux


```
architecture BH of mux2 is
begin
  mux2_1: process(a, b, sel)
      begin
         if sel = '0' then
            y <= a;
         else
            y <= b;
         end if;
      end process mux2_1;
end BH;
```

Dataflow Architecture of 2:1 Mux

- These two dataflow models of a multiplexor are equivalent.
- The first architecture uses a when statement and the second a with statement.
- Both models can also be synthesized, ignoring the 10 ns delay.

```
Architecture dtf1 of mux2
is
begin
   y<=a when sel='0' else
      b;
end;
```

```
Architecture dtf2 of mux2
is
begin
   with sel select
        y <= a when '0',
             b when others;
end;
```

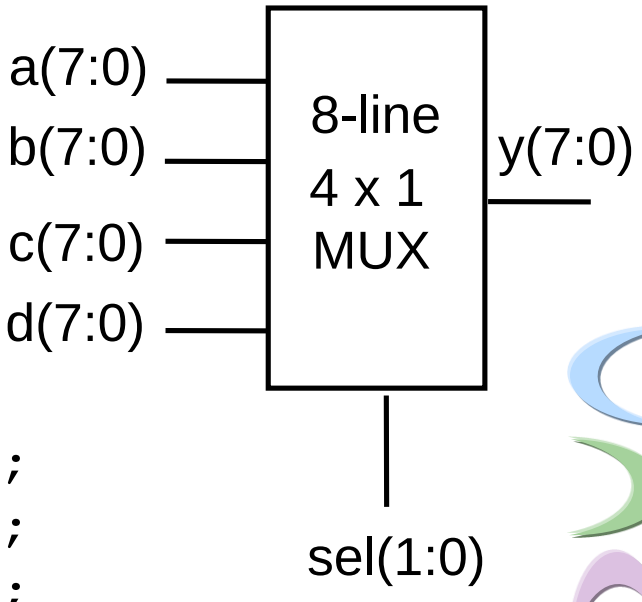
Structural Architecture of 2:1 Mux

Razi University

```
Architecture str of mux2 is
component AND2C port (Input1,Input2: in std_logic;
                      Output: out std_logic); end component;
component OR2C port (Input1,Input2: in std_logic;
                     Output: out std_logic); end component;
component INVC port (Input: in std_logic;
                     Output: out std_logic); end component;
signal i1,i2,sel_n:std_logic;
for U1: INVC Use Entity work.Inv(dtf);
for U2,U3: AND2C Use Entity work.AND2(dtf);
for U4: OR2C Use Entity work.OR2(dtf);
begin
  U1: INVC port map(sel,sel_n);
  U2: AND2C port map(a,sel_n,i1);
  U3: AND2C port map(sel,b,i2);
  U4: OR2C port map(i1,i2,y);
end;
```

An 8-line 4:1 Multiplexer


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

entity mux8L4 is
  port (
   a: in STD_LOGIC_VECTOR (7 downto 0);
   b: in STD_LOGIC_VECTOR (7 downto 0);
   c: in STD_LOGIC_VECTOR (7 downto 0);
   d: in STD_LOGIC_VECTOR (7 downto 0);
   sel: in STD_LOGIC_VECTOR (1 downto 0);
   y: out STD_LOGIC_VECTOR (7 downto 0)
  );
end mux8L4;
```



Sel	y
“00”	a
“01”	b
“10”	c
“11”	d

Behaioral Architecture of 8-line 4:1 Mux


```
architecture BH of mux8L4 is
begin
  process (sel, a, b, c, d)
  begin
    case sel is
      when "00"   => y <= a;
      when "01"   => y <= b;
      when "10"   => y <= c;
      when others => y <= d;
    end case;
  end process;
end BH;
```

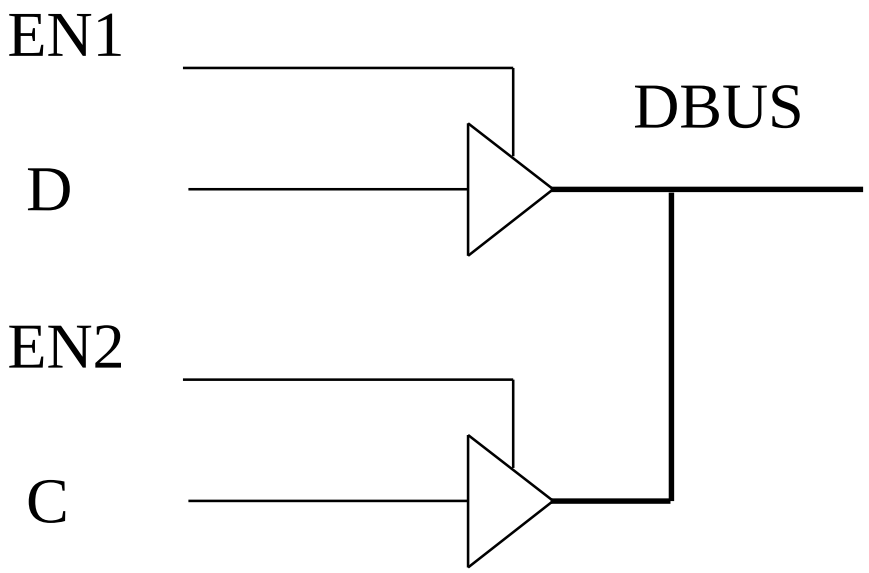
Sel	y
“00”	a
“01”	b
“10”	c
“11”	d

Dataflow Architecture of 8-line 4:1 Mux


```
architecture dtf1 of mux8L4 is
begin
  y <= a when sel="00" else
       b when sel="01" else
       c when sel="10" else
       d;
end dtf1;

architecture dtf2 of mux8L4 is
begin
     with sel select
          y <=  a when "00",
                b when "01",
                c when "10",
                d when others;
end dtf2 ;
```

Sel	y
"00"	a
"01"	b
"10"	c
"11"	d

BUS Driver



DBUS driver with std_logic


```
library IEEE;
use IEEE.std_logic_1164.all;
Entity busc is
  port (d,c,en1,en2: in std_logic;
        dbus: out std_logic);
End busc;
Architecture rtl of busc is
begin
  dbus <= d when en1 = '1' else 'Z';
  dbus <= c when en2 = '1' else 'Z';
end;
```

Generate Statement

- Some regular structure of codes could be written in a more compact form using a loop FOR statement.
- It is different from the "for-loop" statement in C language, in here, FOR loop means duplicate.
- We use For statement along with Generate statement to duplicate existing components and create new circuits by concatenating them.

Generate Statement
Create 16:1 Mux using 4:1 Muxs


```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux4 is
  port (
   a: in STD_LOGIC;
   b: in STD_LOGIC;
   c: in STD_LOGIC;
   d: in STD_LOGIC;
   sel: in STD_LOGIC_VECTOR (1 downto 0);
   y: out STD_LOGIC);
end mux4;
architecture dtf of mux4 is
begin
  y <= a when sel="00" else
       b when sel="01" else
       c when sel="10" else
       d;
end dtf;
```

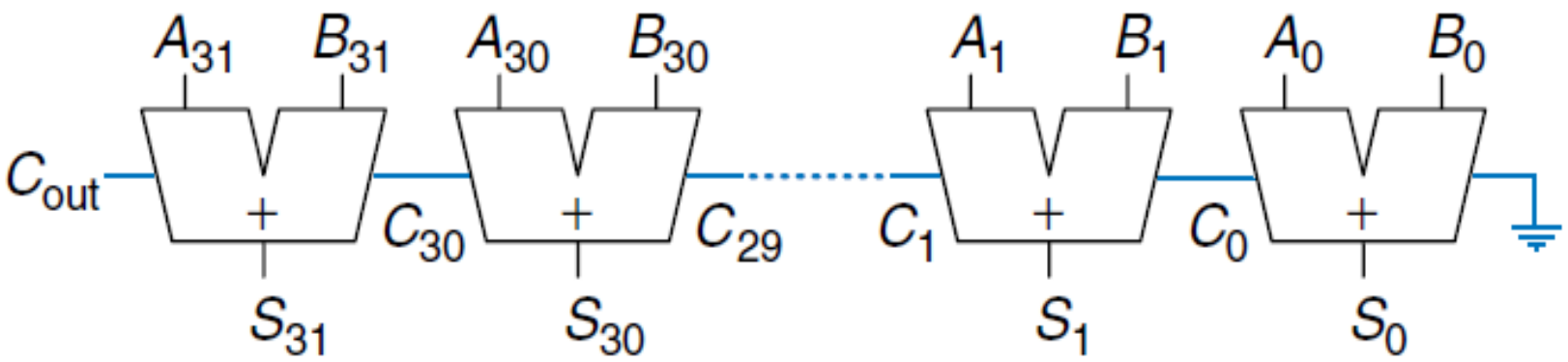
Sel	y
“00”	a
“01”	b
“10”	c
“11”	d

Generate Statement
16-to-1 Multiplexer


```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.all ;
ENTITY mux16to1 IS
      PORT (   w    : IN      STD_LOGIC_VECTOR(0 TO 15) ;
               s    : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
               f    : OUT     STD_LOGIC ) ;
END mux16to1 ;
ARCHITECTURE Structure OF mux16to1 IS
COMPONENT mux4c PORT (a: IN STD_LOGIC; b: IN STD_LOGIC;
                      c: IN STD_LOGIC; d: IN STD_LOGIC;
                      sel: IN STD_LOGIC_VECTOR (1 downto 0);
                      y: OUT STD_LOGIC); END COMPONENT;
FOR ALL: mux4c USE ENTITY work.mux4(dtf);
SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
      G1: FOR i IN 0 TO 3 GENERATE
           Muxes: mux4c PORT MAP (w(4*i), w(4*i+1), w(4*i+2), w(4*i+3),
                                  s(1 DOWNTO 0), m(i) ) ;
      END GENERATE;
      Mux5: mux4c PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
END Structure ;
```

32 Bits Adder


```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
ENTITY Add32 IS
    PORT ( A, B   : IN    STD_LOGIC_VECTOR(31 DOWNTO 0) ;
           Cout   : OUT   STD_LOGIC;
           OV     : OUT   STD_LOGIC;
           S      : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END Add32 ;
```

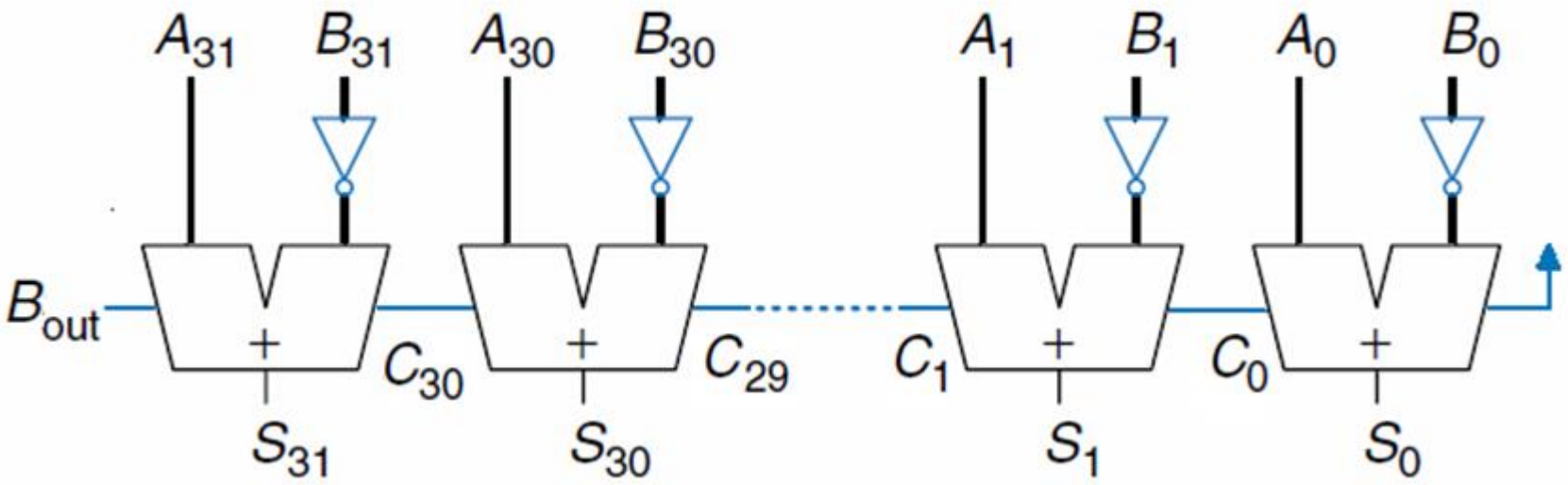


32 Bits Adder


```
ARCHITECTURE STR OF Add32 IS
COMPONENT FAC GENERIC (SDEL, CDEL:TIME);
     PORT(a:  in STD_LOGIC;  b:  in STD_LOGIC;  Ci: in STD_LOGIC;
          S:  out STD_LOGIC; Co: out STD_LOGIC);
END COMPONENT ;

FOR ALL: FAC USE ENTITY work.FA(dtf);
SIGNAL C : STD_LOGIC_VECTOR(31 DOWNTO 0) ;
BEGIN
        Fac0: FAC GENERIC MAP (10 ns, 10 ns);
         PORT MAP (A(0), B(0),'0', S(0), C(0) ) ;
    G1: FOR i IN 1 TO 31 GENERATE
            Facs: FAC GENERIC MAP (10 ns, 10 ns);
         PORT MAP (A(i), B(i), C(i-1), S(i), C(i) ) ;
    END GENERATE;
    Cout <= C(31);
    OV <= C(30) xor C(31);
END STR ;
```

32 Bits Subtractor


```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
ENTITY Sub32 IS
    PORT ( A, B   : IN    STD_LOGIC_VECTOR(31 DOWNTO 0) ;
           Bout   : OUT   STD_LOGIC;
           OV     : OUT   STD_LOGIC;
           S      : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END Sub32 ;
```

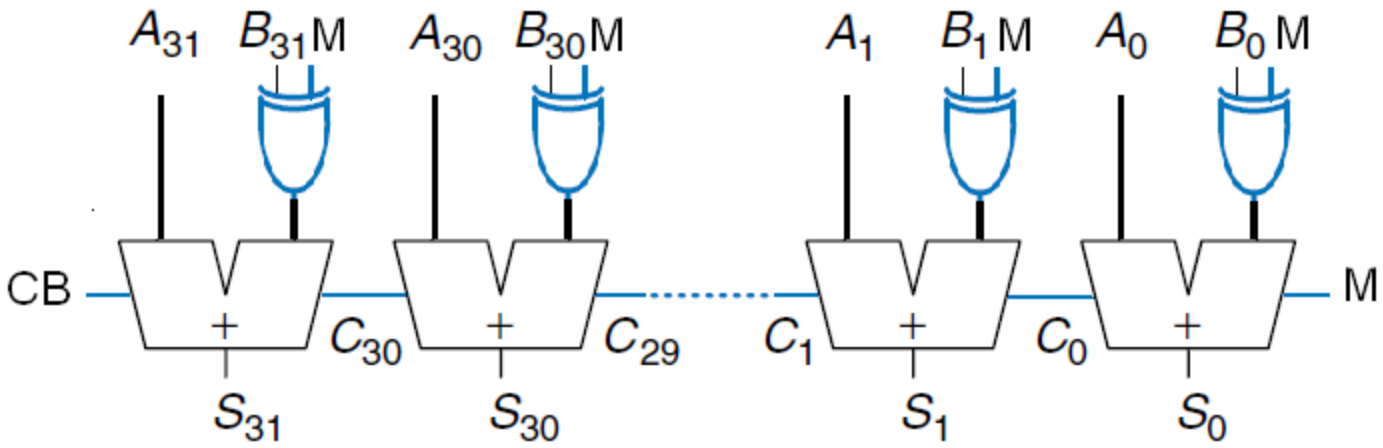


32 Bits Subtractor


```
ARCHITECTURE STR OF Sub32 IS
COMPONENT FAC GENERIC (SDEL, CDEL:TIME);
                PORT(a:  in STD_LOGIC;  b:  in STD_LOGIC;  Ci: in STD_LOGIC;
                S:  out STD_LOGIC; Co: out STD_LOGIC);
END COMPONENT ;
COMPONENT INVC  PORT(Input:  in STD_LOGIC; Output: out STD_LOGIC);
END COMPONENT ;
FOR ALL: FAC USE ENTITY work.FA(dtf);
FOR ALL: INVC USE ENTITY work.INV(dtf);
SIGNAL C : STD_LOGIC_VECTOR(31 DOWNTO 0) ;
SIGNAL BN : STD_LOGIC_VECTOR(31 DOWNTO 0) ;
BEGIN
      G1: FOR i IN 0 TO 31 GENERATE       Facs: INVC   PORT MAP (B(i), BN(i)) ;
      END GENERATE;
       Fac0: FAC GENERIC MAP (10 ns, 10 ns);
            PORT MAP (A(0), BN(0),'1', S(0), C(0) ) ;
      G1: FOR i IN 1 TO 31 GENERATE
            Facs: FAC GENERIC MAP (10 ns, 10 ns);
             PORT MAP (A(i), BN(i), C(i-1), S(i), C(i) ) ;
      END GENERATE;
       Bout <= C(31);
       OV <= C(30) xor C(31);
END STR ;
```

32 Bits Adder/Subtractor


```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
ENTITY AddSub32 IS
    PORT ( A, B   : IN    STD_LOGIC_VECTOR(31 DOWNTO 0) ;
           M      : IN    STD_LOGIC;
           CB     : OUT   STD_LOGIC;
           OV     : OUT   STD_LOGIC;
           S      : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END AddSub32 ;
```



32 Bits Adder/Subtractor


```
ARCHITECTURE STR OF AddSub32 IS
COMPONENT FAC GENERIC (SDEL, CDEL:TIME);
                PORT(a:  in STD_LOGIC;  b:  in STD_LOGIC;  Ci: in STD_LOGIC;
                  S:  out STD_LOGIC; Co: out STD_LOGIC);
END COMPONENT ;
COMPONENT XORC  PORT(Input1:  in STD_LOGIC; Input2:  in STD_LOGIC;
Output: out STD_LOGIC); END COMPONENT ;
FOR ALL: FAC USE ENTITY work.FA(dtf);
FOR ALL: XORC USE ENTITY work.XOR2(dtf);
SIGNAL C : STD_LOGIC_VECTOR(31 DOWNTO 0) ;
SIGNAL BX : STD_LOGIC_VECTOR(31 DOWNTO 0) ;
BEGIN
      G1: FOR i IN 0 TO 31 GENERATE   Facs: XORC   PORT MAP (B(i), M, BX(i)) ;
      END GENERATE;
       Fac0: FAC GENERIC MAP (10 ns, 10 ns);
            PORT MAP (A(0), BX(0),M, S(0), C(0) ) ;
      G1: FOR i IN 1 TO 31 GENERATE
            Facs: FAC GENERIC MAP (10 ns, 10 ns);
             PORT MAP (A(i), BX(i), C(i-1), S(i), C(i) ) ;
      END GENERATE;
       CB <= C(31);
       OV <= C(30) xor C(31);
END STR ;
```

Test Bench

- **Testbench is not defined by the VHDL Language Reference Manual and has no formal definition.**
- **The testbench is a specification in VHDL that plays the role of a complete simulation environment for the analyzed system (unit under test, UUT).**
- **A testbench contains both the UUT as well as stimuli for the simulation.**
- **The UUT is instantiated as a component of the testbench.**
- **The architecture of the testbench specifies stimuli for the UUT's ports, usually as waveforms assigned to all output and bidirectional ports of the UUT.**
- **The entity of a testbench does not have any ports as this serves as an environment for the UUT.**

Test Bench


```
entity testbench is
end testbench;

architecture testbench_arch of testbench is
   --signal declarations
   --component declarations
begin
   --component instantiations
   --stimuli (test vectors)
end testbench_arch;
```

Test Bench for Full Adder


```
entity FAtestbench is
end FAtestbench;
architecture testbench_arch of FAtestbench is
signal a,b,ci,S,Co:STD_LOGIC;
COMPONENT FAC GENERIC (SDEL, CDEL:TIME);
   PORT(a:  in STD_LOGIC;  b:  in STD_LOGIC;  Ci: in STD_LOGIC;
        S:  out STD_LOGIC; Co: out STD_LOGIC); END COMPONENT ;
FOR ALL: FAC USE ENTITY work.FA(dtf);
begin
 --component instantiations
 Fac0: FAC GENERIC MAP (10 ns, 10 ns);
           PORT MAP (a, b, ci, S, Co) ;
 --stimuli (test vectors)
Process
Begin
 a <= '0' , '1' after 200 ns, '0' after 800 ns, '1' after 2000 ns;
 b <= '0' , '1' after 400 ns, '0' after 1200 ns,'1' after 1800 ns;
 ci<= '0' , '1' after 650 ns, '0' after 900 ns, '1' after 1300 ns;
Wait;
End process;
end testbench_arch;
```

Direct Instantiation

- Direct instantiation of components is introduced in the VHDL-93 standard
- It means that neither component declaration nor configuration are needed to instance a component
- The next example shows the difference between the two instantiation method:

```
-- Model according to VHDL-87
architecture rtl of top_level is
  component c1
    port(a,b: in std:logic; q: out std_logic);
  end component;
for U1: c1 use entity work.c1(rtl);
begin
  U1: c1 port map(a,b,q);
end;
```

```
-- Model according to VHDL-93
architecture rtl of top_level is
begin
  U1: entity work.c1(rtl) port map(a,b,q);
end;
```

Components in packages I

- If the components are defined in packages, no component declaration is needed in the architecture when the component is instantiated
- Example:

```
package mypack is
 function minimum (a,b:in std_logic_vector) return std_logic_vector;
 component c1 port(clk,resetn,din:in std_logic; q1,q2:out std_logic); end component;
 component c2 port(a,b:in std_logic; q:out std_logic); end component;
end mypack;
package body mypack is
 function minimum (a,b:in std_logic_vector) return std_logic_vector is
 begin
   if a<b then return a;
   else return b;
   end if;
 end minimum;
end mypack;
```

Components in packages II

- To use the component the declaration of the package is needed in the beginning of the VHDL code, only:

```
library ieee;
use ieee.std_logic_1164.ALL;
use work.mypack.ALL;

entity ex is
 port(clk,resetn:in std_logic; d1,d2:out std_logic; a,b: in std_logic_vector(3 downto 0);
       q1,q2,q3: out std_logic; q4:out std_logic_vector(3 downto 0));
end;

Architecture rtl of ex is
begin
  U1: c1 port map(clk,resetn,d1,q1,q2);
  U2: c2 port map(d1,d2,q3);
  q4<=minimum(a,b);
end;
```

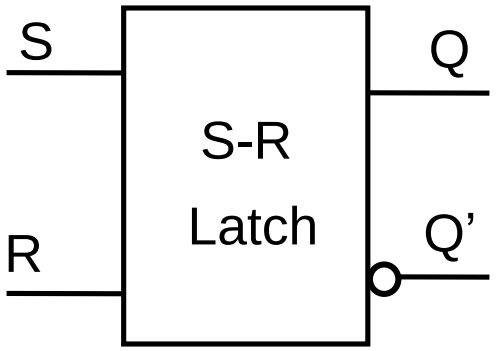
Sequential Circuit Design

Sequential Circuit Example

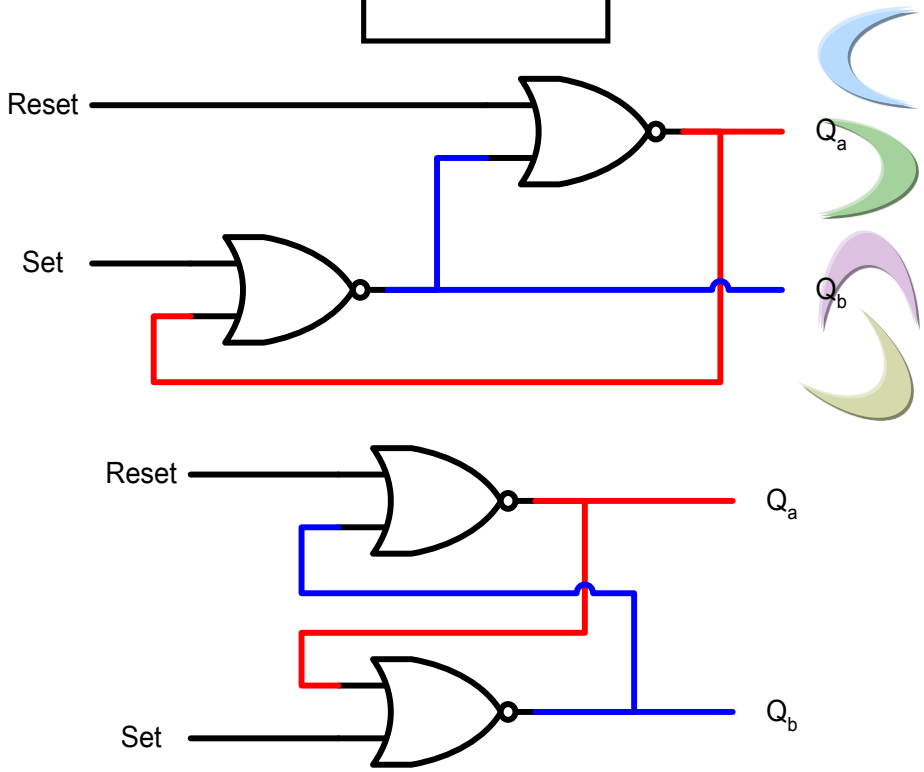
- SR-Latch
- D-Flip Flop
- T-Flip Flop
- JK-Flip Flop
- Clock Generator
- Shift Registers
- Memory RAM
- Sequence Detectors with F.F.s
- Sequence Detectors with FSMs
- Timing Checking (Setup Time ,Hold Time and Minimum Pulse Width Checker)
- Testing Models
- Algorithmic Design
  - Parallel to Serial Converter
  - Multi Value Logics
- Register Level Design

Basic S-R Latch

- **Both circuit are the same**
- **The only feedback path is the red line**




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

entity SRL is
    port (
        S: in BIT;
        R: in BIT;
        Q: inout BIT;
        QN: inout BIT
    );
end SRL;
```

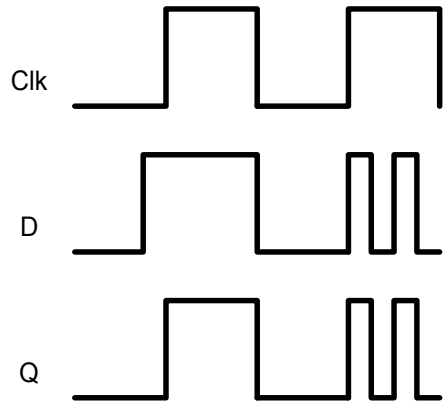


Basic S-R Latch

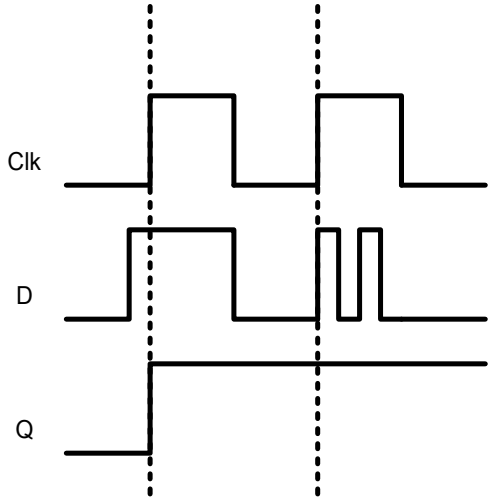

```
Architecture str of SRL is
component NOR2C port (Input1,Input2: in
  std_logic;
                      Output: out std_logic); end
  component;
for All: NOR2C Use Entity work.NOR2(dtf);
begin
  U1: NOR2C port map(S,Q,QN);
  U2: NOR2C port map(R,QN,Q);
end;
```

Level sensitive and edge sensitive

- For a latch and flip-flop (FF), it can be level sensitive or edge sensitive
  - Level sensitive means the latch / FF will copy input D to output Q when Clk = 1
  - Edge sensitive means that the latch / FF will only copy input D to output Q when Clk change from 0 -> 1 (positive edge trigger) / 1 -> 0 (negative edge trigger)

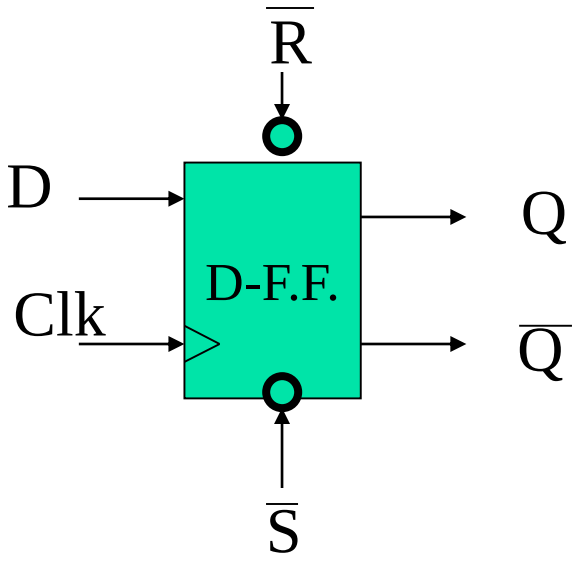


Level Sensitive



Edge Sensitive

D Flip-Flop



$\overline{S}$	$\overline{R}$	Clk	D	Q	$\overline{Q}$
0	1	X	X	1	0
1	0	X	X	0	1
1	1	R	1	1	0
1	1	R	0	0	1
0	0	X	X	?	?

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DFF is
   generic(RSDel,CKDel:TIME);
   port(R,S,D,Clk:in BIT;
        Q, Qn:out BIT);
end DFF;
```

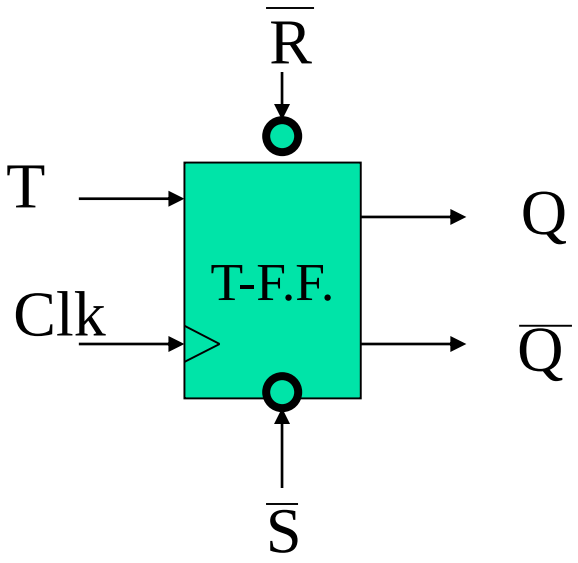
D Flip-Flop


```
architecture BH of DFF is
begin
    output:process (R,S, Clk)
    begin
          assert not(R = '0' and S = '0')
          report "Set and Reset can't be actived at same
    time!"
       if(R = '0') then
            Q  <= '0' after SRDel;
            QN <= '1' after SRDel;
       elsif S ='0' then
            Q  <= '1' after SRDel;
            QN <= '0' after SRDel;
       elsif (Clk'event and Clk = '1') then
            Q  <= D after CKDel;
            QN <= (not D) after CKDel;
       end if;
    end process output;
end BH;
```

D Flip-Flop

- Asserting the **Set** input (**S**), forces Q = 1.
- Asserting the **Reset** input (**R**), forces Q = 0.
- These signals **override** the effect of the *clock signal*, and are *active* at **any time**.
- Hence they are **asynchronous** inputs, as opposed to the ***synchronous*** nature of the clock signal.
- The **R** input **overrides** the **S** input.
- Both signals are **active low**. E.g. to set Q = 0, a zero pulse is applied to the reset input, while the set input is held to 1, and vice versa.
- During ***synchronous*** operation, both **S** and **R** must be held to **1**.
- The proposed Flip Flop is **positive edge trigger.**

T Flip-Flop



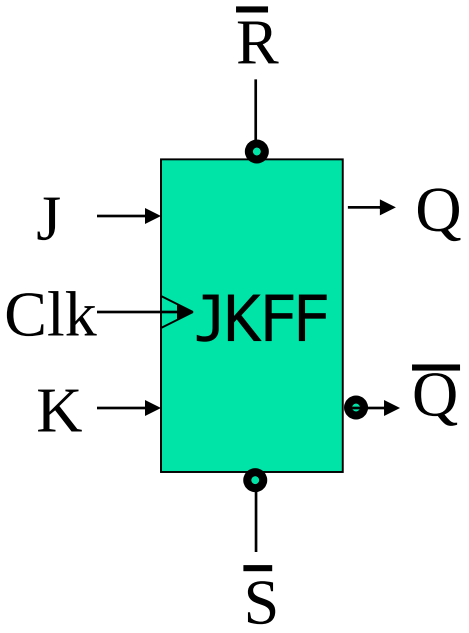
$\overline{S}$	$\overline{R}$	Clk	T	Q	$\overline{Q}$
0	1	X	X	1	0
1	0	X	X	0	1
1	1	R	1	$\overline{q}$	q
1	1	R	0	q	$\overline{q}$
0	0	X	X	?	?

```
library IEEE;
use IEEE.std_logic_1164.all;
entity TFF is
   generic(RSDel,CKDel:TIME);
   port(R,S,T,Clk:in BIT;
        Q, Qn:inout BIT);
end TFF;
```

T Flip-Flop


```
architecture BH of TFF is
begin
   output:process (R,S, Clk)
   Begin
    assert not(R = '0' and S = '0')
    report "Set and Reset can't be actived at same time!"
    if(R = '0') then
       Q  <= '0' after SRDel;
       QN <= '1' after SRDel;
     elsif S ='0' then
       Q  <= '1' after SRDel;
       QN <= '0' after SRDel;
     elsif ((Clk'event and Clk = '1')and T = '1') then
       Q  <= (not Q)  after CKDel;
       QN <= (not QN) after CKDel;
     end if;
   end process output;
end BH;
```

JK Flip-Flop



$\overline{S}$	$\overline{R}$	J	K	Q	$\overline{Q}$
0	1	X	X	1	0
1	0	X	X	0	1
1	1	0	0	q	$\overline{q}$
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	$\overline{q}$	q
0	0	X	X	?	?

```
library IEEE;
use IEEE.std_logic_1164.all;
entity JKFF is
   generic(RSDel,CKDel:TIME);
   port(R,S,J,K,Clk:in BIT;
        Q, Qn:inout BIT);
end JKFF;
```

JK Flip-Flop

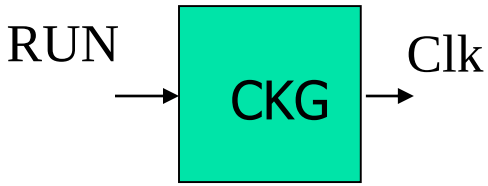

```
architecture BH of JKFF is
begin
   output:process (R,S, Clk)
   Begin
    assert not(R = '0' and S = '0')
    report "Set and Reset can't be actived at same time!"
    if(R = '0') then
       Q  <= '0' after SRDel;
       QN <= '1' after SRDel;
     elsif S ='0' then
       Q  <= '1' after SRDel;
       QN <= '0' after SRDel;
```

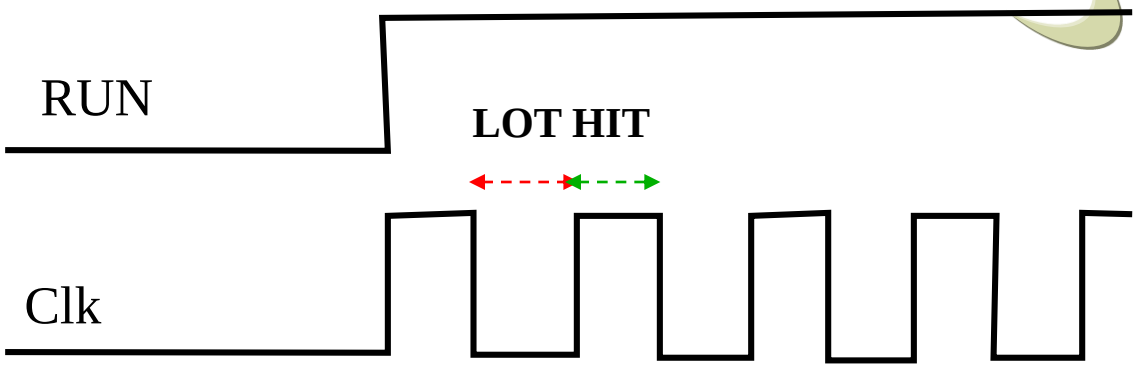
JK Flip-Flop Continue


```
    elsif (Clk'event and Clk = '1') then
     if (J = '0' and K = '1')then
            Q <= '0' after CKDel;
            QN <='1' after CKDel;
     elsif (J = '1' and K = '0')then
            Q <= '1' after CKDel;
            QN <='0' after CKDel;
     elsif (J = '1' and K = '1')then
            Q <= (not Q) after CKDel;
            QN <= (not QN) after CKDel;
     end if;
    end if;
  end process output;
end BH;
```

Clock Generator


```
library IEEE;
use IEEE.std_logic_1164.all;
entity CKG is
   generic(HIT,LOT:TIME);
   port(RUN:in BIT; Clk:out BIT:='0');
end CKG;
architecture ALG of CKG is
begin
   process
   Begin
       wait until RUN='1';
       while RUN='1' loop
           Clk <= '1';
           wait for HIT;
           Clk <= '0';
           wait for LOT;
        end loop;
   end process;
end ALG;
```



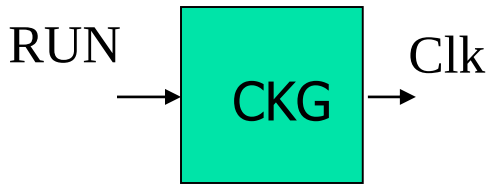


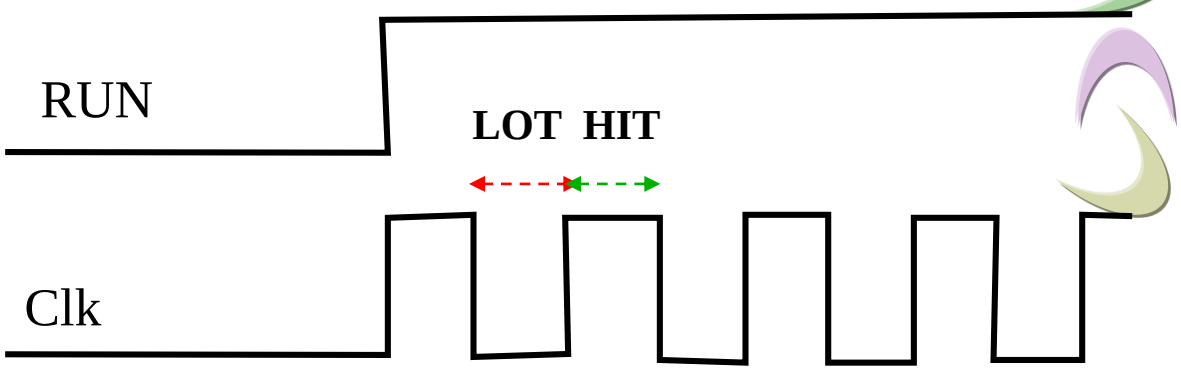
Clock Generator


```
architecture BH of CKG is
Signal CK:BIT:='0';
begin
   Process(RUN , CK)
   Variable CKE:BIT:='0';
   Begin
        if (RUN='1' and not RUN'STABLE) then
            CKE:='1';
            CK <= transport '0' after HIT;
            CK <= transport '1' after LOT;
        end if;
        if (RUN='0' and not RUN'STABLE) then
            CKE:='0';
        end if;
        if (CK='1' and not CK'STABLE and CKE='1') then
            CK <= transport '0' after HIT;
            CK <= transport '1' after LOT;
        end if;
        Clk <= CK;
   end process;
end BH;
```

Clock Generator

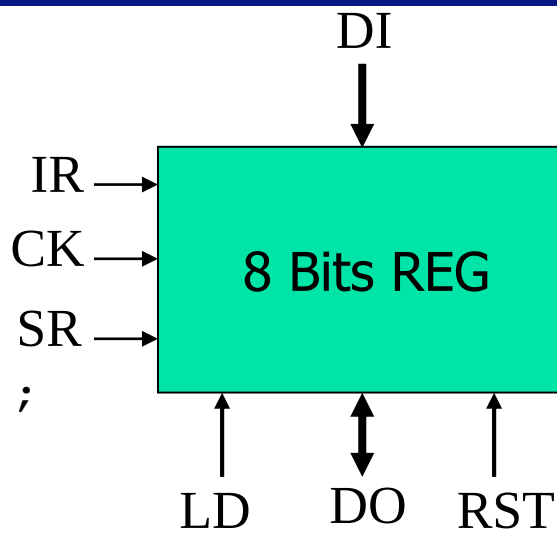

```
architecture DF of CKG is
begin
   Process(RUN , Clk)
   Begin
    Clk <= not(Clk) after HIT when (RUN='1' and Clk='1') else
           not(Clk) after LOT when (RUN='1' and Clk='0');
   end process;
end DF;
```





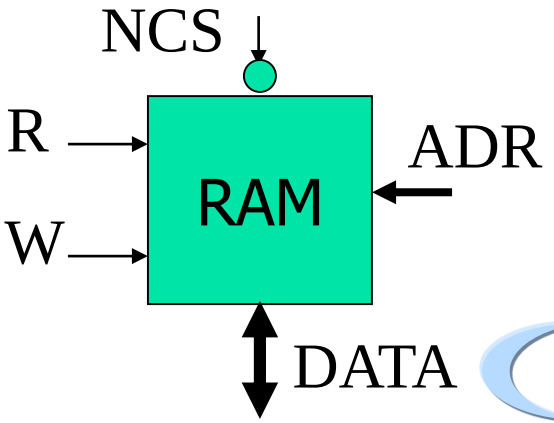
Shift Register


```
library IEEE;
use IEEE.std_logic_1164.all;
entity SREG is
   generic(DEL:TIME);
   port(DI:in BIT_VECTOR(7 downto 0);
        CK,LD,RST,SR,SL,IR,IL:in BIT;
        DO:inout BIT_VECTOR(7 downto 0));
end SREG;
architecture DF of SREG is
begin
   REG:block (not CK'STABLE and CK='1')
   Begin
       DO <= guarded "00000000" after DEL when RST='1' else
             DI after DEL when LD='1' else
             DO(6 downto 0) & IL after DEL when SL='1' else
             IR & DO(7 downto 1) after DEL when SR='1' else
             DO;
   end block REG;
end DF;
```



Memory RAM

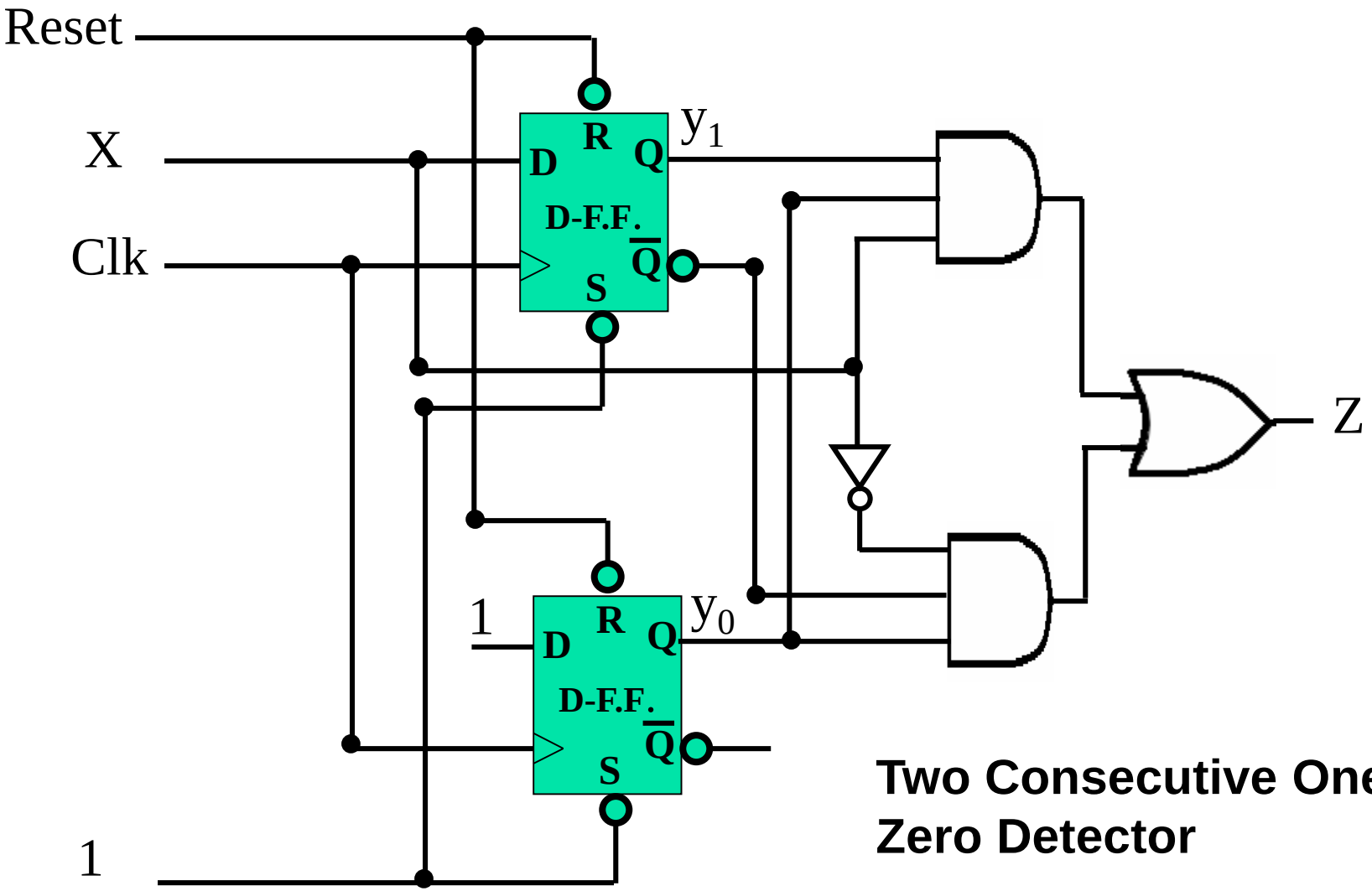

```
library IEEE;
use IEEE.std_logic_1164.all;
entity RAM is
  port(W,R,NCS: in BIT;
       ADR: in BIT_VECTOR(7 downto 0);
      DATA: inout BIT_VECTOR(31 downto 0));
end RAM;
architecture BH of RAM is
type memory is array(0 to 255) of BIT_VECTOR(31 downto 0);
Function INTVAL(V:BIT_VECTOR) return INTEGER is
  Variable V1:BIT_VECTOR(V'LENGTH-1 downto 0);
  Variable SUM:INTEGER:=0;
  begin
     V1:=V;
     for I in V1'LOW to V1'HIGH loop
        if V1(I)='1' then SUM := SUM+(2**I); end if;
     end loop;
    Return SUM;
  End INTVAL;
```



Memory RAM


```
Begin
     process(R,W,NCS,ADR,DATA)
        variable MEM:memory;
     begin
        if NCS='0' then
           if R='1' then
              DATA <= MEM(INTVAL(ADR));
           elsif W='1' then
              MEM(INTVAL(ADR)):= DATA;
           end if;
        else
           DATA <= "ZZZZZZZZ";
        end if;
     end process;
end BH;
```

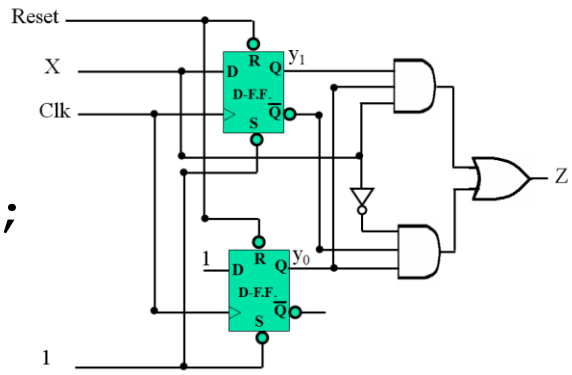
Sequence Detector with D-F.F.



Two Consecutive One or Zero Detector

Sequence Detector without F.F.


```
library IEEE;
use IEEE.std_logic_1164.all;
entity TWOCONSEC is
      port(Reset,X,Clk:in BIT; Z:out BIT);
end TWOCONSEC;
architecture DF of TWOCONSEC is
Signal y0,y1:BIT;
begin
   STATE:block((Clk='1' and not Clk'STABLE) or Reset ='0')
   Begin
       y1 <= guarded '0' when Reset ='0' else X;
       y0 <= guarded '0' when Reset ='0' else '1';
   end block STATE;
   Z <= y0 and ((not y1 and not X) or (y1 and X));
end DF;
```

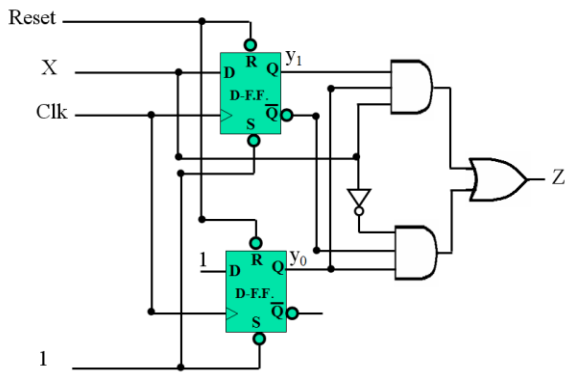


Sequence Detector with D-F.F.


```
architecture STR of TWOCONSEC is
component AND3C port (Input1,Input2,Input3: in std_logic;
          Output: out std_logic);
end component;
component OR2C port (Input1,Input2: in std_logic;
          Output: out std_logic);
end component;
component INVC port (Input: in std_logic;
                     Output: out std_logic);
end component;
component DFFC generic(RSDel,CKDel:TIME);
   port(R,S,D,Clk:in BIT; Q, Qn:out BIT);
end component;
for D0,D1: DFFC Use Entity work.DFF(BH);
for A1,A2: AND3C Use Entity work.AND3(dtf);
for O1: OR2C Use Entity work.OR2(dtf);
for N1: INVC Use Entity work.Inv(dtf);
```

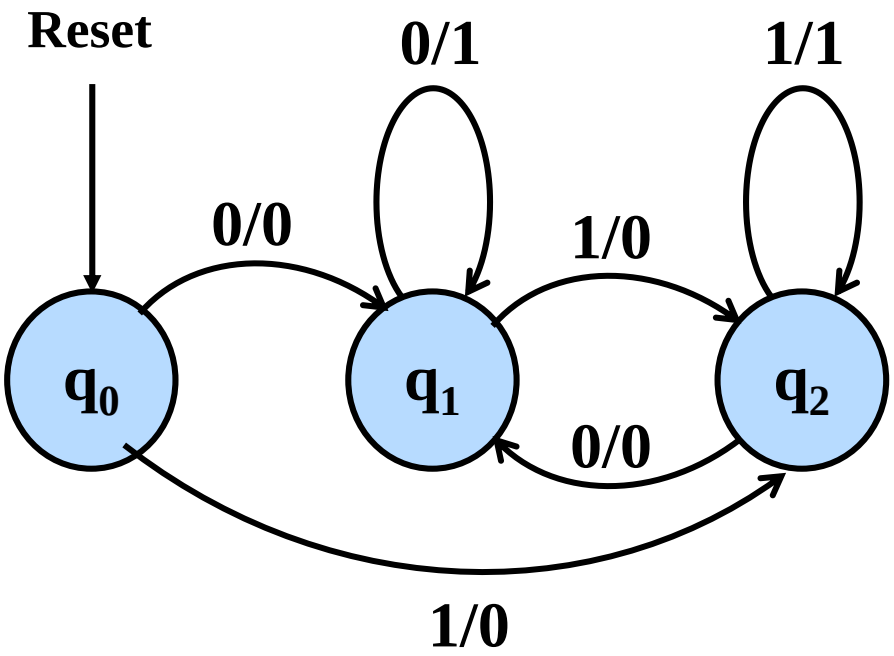
Sequence Detector with D-F.F.


```
Signal y0,ny0,y1,ny1,nx,ao1,ao2:BIT;

begin
N1: INVC PORT MAP (X, nx) ;

D0: DFFC GENERIC MAP (5 ns, 5 ns);
         PORT MAP (Reset,'1','1',Clk,y0,ny0) ;
D1: DFFC GENERIC MAP (5 ns, 5 ns);
         PORT MAP (Reset,'1',X,Clk,y1,ny1) ;
A1: AND3C PORT MAP (nx,ny1,y0,ao1) ;
A2: AND3C PORT MAP (X,y1,y0,ao2) ;
O1: OR2C PORT MAP (ao1,ao2,Z) ;

end STR;
```



Sequence Detector with FSM

Two Consecutive One or Zero Detector Mealy FSM



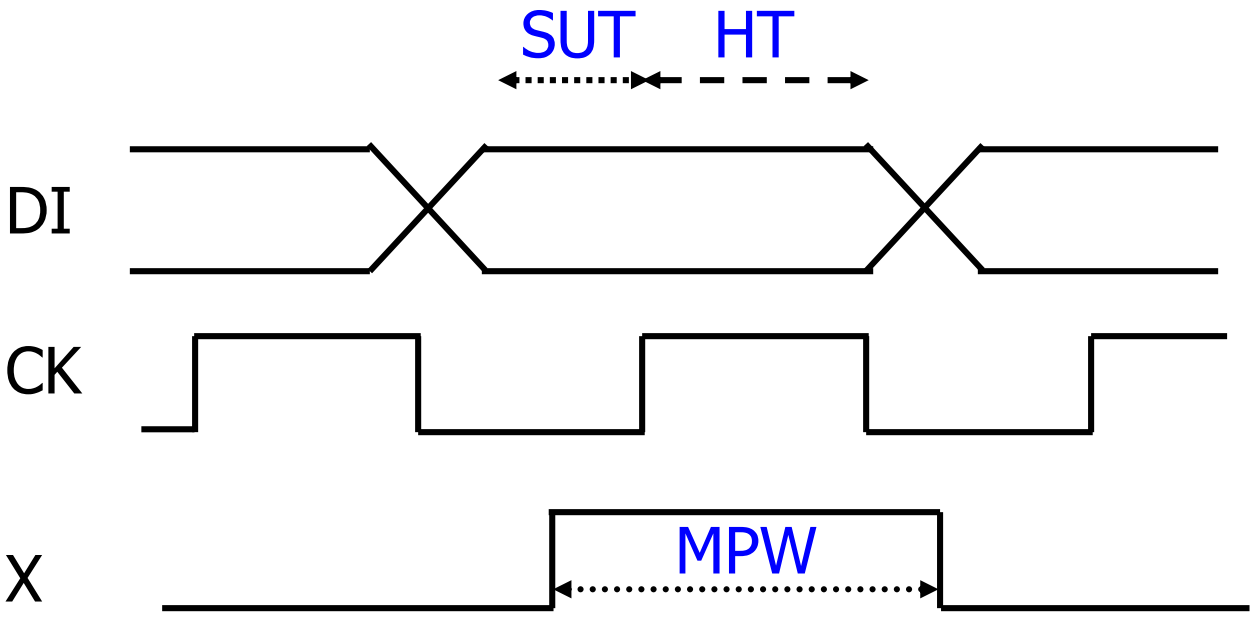

```
library IEEE;
use IEEE.std_logic_1164.all;
entity TWOCONSEC is
       port(Reset,X,Clk:in BIT; Z:out BIT);
end TWOCONSEC;
```

Sequence Detector without F.F.


```
architecture ALG of TWOCONSEC is
TYPE STATE is (q0,q1,q2);
Signal Q: STATE := q0;
begin
   process(Clk,X,Q,Reset)
   Begin
        if (Reset='0' and Reset'EVENT) then  Q <= q0;
         elsif (Clk='1' and Clk'EVENT) then
             if (X='0') then    Q <= q1;
              else   Q <= q2;
             end if;
        end if;
        if (Q'EVENT or X'EVENT) then
           if (X='0' and Q=q1) or (X='1' and Q=q2) then
               Z <= '1';
            else
               Z <= '0';
           end if;
        end if;
   end process;
end ALG;
```

Timing Checking

- **Built-in Inertial delay filters out the signal assignment that its pulse width is shorter than specific delay value in "after" clause.**
- **It is useful to be able to print a message when short pulse is detected;**
- **Also it is necessary to check other complex timing issue like "SETUP-TIME" and "HOLD-TIME" for registers and other clocked circuits.**



Timing Checking


```
entity CKEDCOMP is
   generic(MPW,SUT,HT:TIME);
   port(CK: in BIT;
        DI: in BIT_VECTOR(7 downto 0);
         X: in BIT);
Begin
   -- these assertions are used for all architectures.
   assert not(not CK'STABLE and CK='1' and not DI'STABLE(SUT))
   report "Setup Time Failure!"

   assert not(not CK'DELAYED(HT)'STABLE and CK'DELAYED(HT)='1'
          and not DI'STABLE(HT))
   report "Hold Time Failure!"

   assert not(not X'STABLE and X='0' and
          not X'DELAYED'STABLE(MPW))
   report "Minimum Pulse Width Failure!"

end CKEDCOMP;
```

Timing Checking


```
entity CKEDCOMP is
   generic(MPW,SUT,HT:TIME);
   port(CK: in BIT;
        DI: in BIT_VECTOR(7 downto 0);
         X: in BIT);
Begin
   -- these assertions are used for all architectures.
   assert (CK'STABLE or CK='0' or DI'STABLE(SUT))
   report "Setup Time Failure!"

   assert (CK'DELAYED(HT)'STABLE or CK'DELAYED(HT)='0' or
           DI'STABLE(HT))
   report "Hold Time Failure!"

   assert (X'STABLE or X='1' or X'DELAYED'STABLE(MPW))
   report "Minimum Pulse Width Failure!"

end CKEDCOMP;
```

Testing Models

- **Any VHDL model must be thoroughly simulated and tested before one can conclude that it is correct.**

- **Models are tested in a *test bench*. The test bench can be implemented in three ways:**

  - **Stimuli all inputs of model using signal assignment statements.**

  - **Stimuli all inputs of model using simulator input generator tools. (Connect each input to a clock with predefined period)**

  - **Stimuli all inputs of model using input generator primitives. (Create all possible input patterns and apply them to model's inputs)**

Testing Models (Input Generator Primitives)

Razi University

```
entity IGP is
   generic(N: INTEGER; PER:TIME);
   port(RUN: in BIT; PGOUT: out BIT_VECTOR(N-1 downto 0));
End IGP;
Architecture ALG of IGP is
 Function INT2BIN(INP:INTEGER; N:POSITIVE)return BIT_VECTOR is
      variable FOUT: BIT_VECTOR;
      variable TA: INTEGER:=0; variable TB: INTEGER:=0;
   Begin – beginning of function
      TA:=INP;
      For I in N-1 downto 0 loop
         TB:= TA/(2**I);
         TA:= TA rem (2**I);
         If (TB = 1) then  FOUT(N-1-I):= '1';
         Else  FOUT(N-1-I):= '0'; end if;
      End loop;
      return FOUT;
   end INT2BIN;
```

Testing Models (Input Generator Primitives)


```
Begin – beginning of architecture
        Process(RUN)
        begin
            For I in 0 to 2**N-1 loop
                PGOUT <= transport INT2BIN(I,N) after I*PER;
            End loop;
        End process;
end ALG;
```

Time from beginning of simulation

- **PER** is the duration of each input value.
- **N** is the Number of Inputs in the model under test (**MUT**).
- We define a component of input generator primitives (**IGP**) and apply its outputs to Inputs in the model under test (**MUT**).

Testing Models

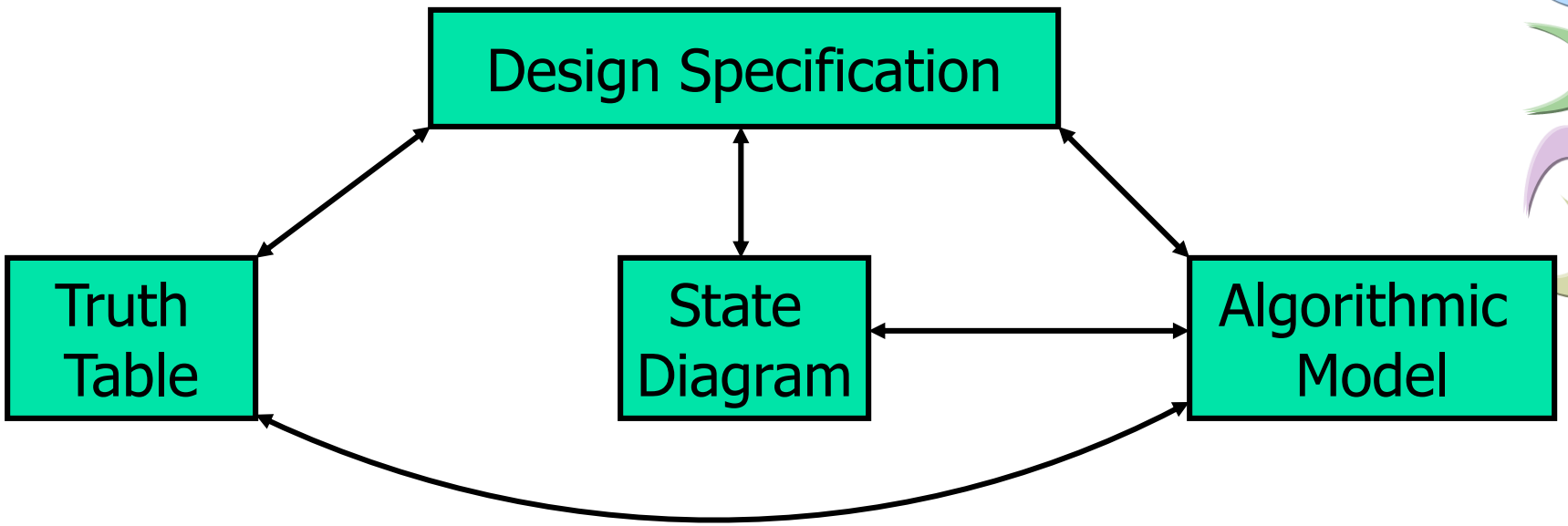

```
entity testbench is
End testbench;
Architecture ALG of testbench is
   signal RUN,S,R,J,K,CK,Q,QN:in BIT;
   signal PG: BIT_VECTOR(3 downto 0);
   component  CKGC
      generic(HIT,LOT:TIME);
      port(RUN:in BIT; Clk:out BIT:='0');
   end component;
   component JKFFC
      generic(RSDel,CKDel:TIME);
      port(R,S,J,K,Clk:in BIT;
           Q, Qn:inout BIT);
   end component ;
   component IGPC
      generic(N: INTEGER; PER:TIME);
      port(RUN: in BIT; PGOUT: out BIT_VECTOR(N-1 downto 0));
   end component ;
```

Testing Models


```
for T1:CKGC use entity CKG(ALG);
for T2:IGPC use entity IGP(ALG);
for T3:JKFFC use entity JKFF(BH);
begin
    T1:CKGC
       generic map(25 ns,25 ns);
       port map(RUN, CK);
    T2:IGPC
       generic map(4, 50 ns);
       port map(RUN, PG);
    T3:JKFFC
       generic map(12 ns,12 ns);
       port map(PG(3),PG(2),PG(1),PG(0),CK,Q,Qn);
    Process
    begin
       RUN <= '0', '1' after 50 ns, '0' after 9000 ns;
       wait;
    end process;
end ALG ;
```

Algorithmic Level Design

- The specification of each circuit is written in a natural language, such as English.
- The first step in the process of the designated description is translating it to one of three forms: 1) a truth table 2) a state diagram, 3) an algorithmic model.



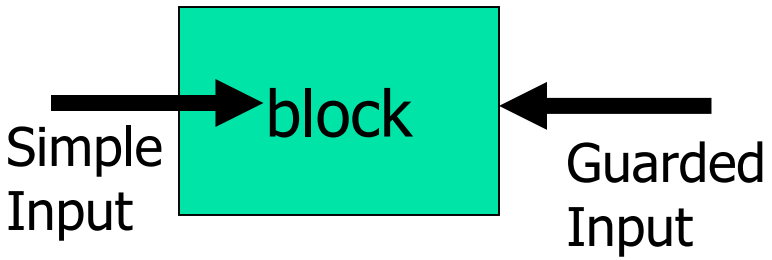
Algorithmic Level Design

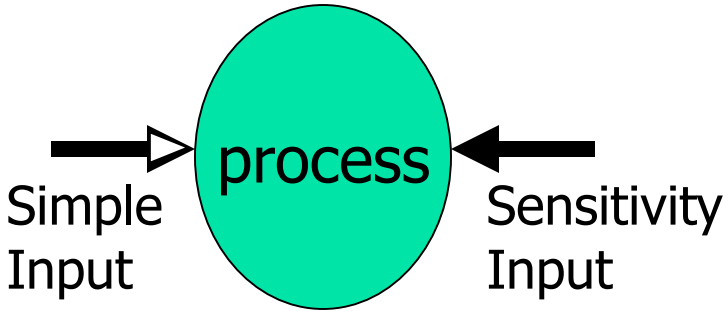
- The truth table and state diagram modeling methods are discussed previously, and algorithmic model can be formed by:
  - Map groups of sentences to the VHDL processes.
  - Assign an activity list to the each process.
  - Develop the VHDL code that implement each process (activity).
- Since in the VHDL an architecture body is defined as a set of concurrent processes, one can represent the design with a process model graph (PMG).
- In the PMG each node relates to one process of the design model and arcs denote the passages between the process nodes.
- Each node of the PMG can be implemented in the VHDL by one block statement or one process statement.

Algorithmic Level Design

- When the node is implemented with block statement ,
  - We represent each node by a rectangle.
  - For guarded inputs of block, their arcs are touched to the outer edge of blocks.
  - For simple inputs of block, their arcs points to the internal of the blocks.

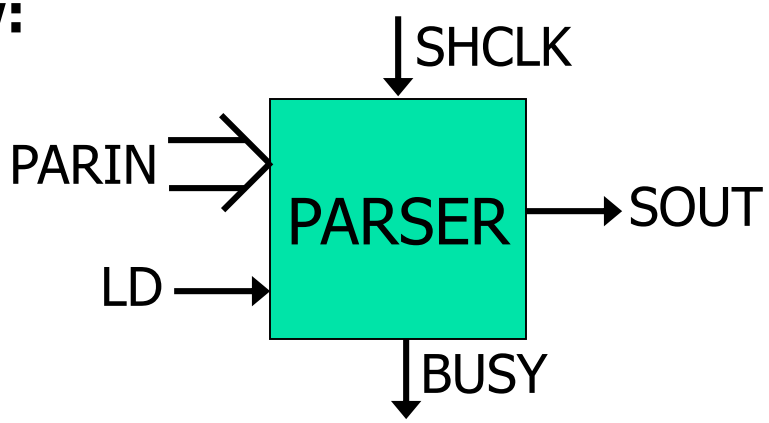
- When the node is implemented with process statement:
  - We represent each node by a circle.
  - For sensitivity inputs of process, their arcs have solid arrowheads.
  - For simple inputs of process, their arcs have open arrowheads.





Algorithmic Model for Parallel to Serial Converter (PARSER)

- **The block diagram of the parallel to serial converter (PARSER) is given in the below:**

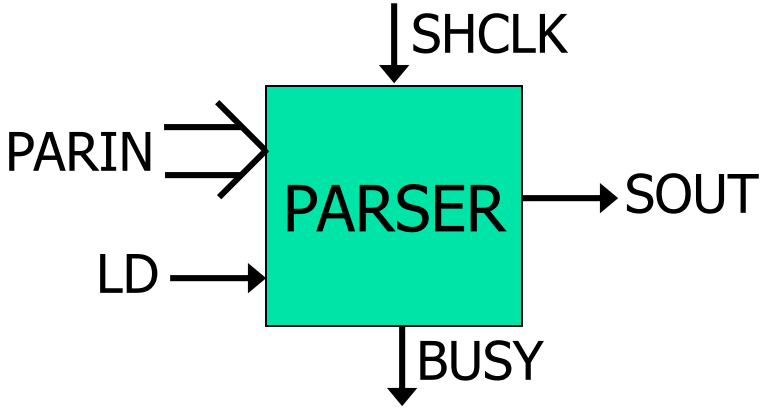


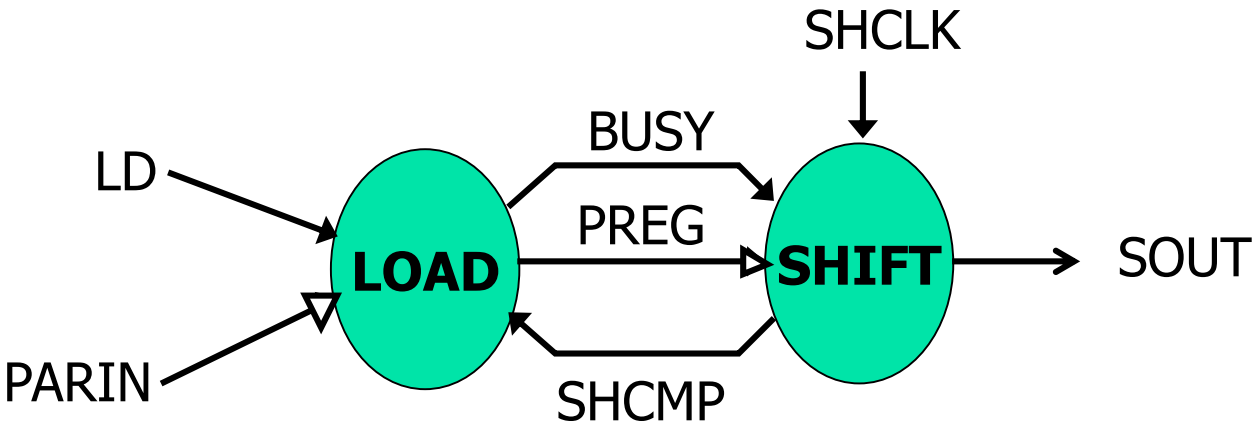
- **In this diagram the 8-bit parallel input PARIN is loaded into converter when LD has a zero to one transition.**
- **After loading input the status signal BUSY is set to high.**
- **Then this parallel input is shifted out serially at a rate controlled by the input shift clock SHCLK.**
- **Shifting occurs at the rise of clock.**
- **BUSY remain high until shifting is completed.**
- **While BUSY is high, no further load will be accepted.**

Algorithmic Model for Parallel to Serial Converter (PARSER)

- **To extract the process model graph of the parallel to serial converter (PARSER), we can simply partition it to two processes:**
  - **LOAD Process:**
    - **In this diagram the 8-bit parallel input is loaded into converter when LD has a zero to one transition.**
    - **Then BUSY signal is set to high.**
    - **BUSY remain high until shifting is completed.**
    - **While BUSY is high, no further load will be accepted.**
  - **SHIFT Process:**
    - **The parallel input data is shifted out serially at a rate controlled by the input shift clock SHCLK.**
    - **Shifting occurs at the rise of clock.**
    - **BUSY remain high until shifting is completed.**

Algorithmic Model for Parallel to Serial Converter (PARSER)





Algorithmic Model for Parallel to Serial Converter (PARSER)


```
entity PARSER is
   port(LD, SHCLK: in BIT; PARIN: in BIT_VECTOR(0 to 7);
        BUSY: inout BIT; SOUT: out BIT);
End PARSER;
Architecture ALG of PARSER is
   signal SHCMP: BIT:='0';
   signal PREG: BIT_VECTOR(0 to 7);
   Begin
        LOAD: Process(LD,SHCMP)
        Begin
        - Activities:
         -- 1) Register loading
         -- 2) BUSY Set to high (at beginning of shifting)
         -- 3) BUSY set to low (at end of shifting)
        End Process LOAD;
        SHIFT: Process(BUSY,SHCLK)
        Begin
        - Activities:
         -- 1) Shift Initialize
         -- 2) Shift
         -- 3) Shift Complete
        End Process SHIFT;
    End ALG;
```

Algorithmic Model for Parallel to Serial Converter (PARSER)

Razi University

```
entity PARSER is
   port(LD, SHCLK: in BIT; PARIN: in BIT_VECTOR(0 to 7);
        BUSY: inout BIT; SOUT: out BIT);
End PARSER;
Architecture ALG of PARSER is
   signal SHCMP: BIT:='0';
   signal PREG: BIT_VECTOR(0 to 7);
   Begin
        LOAD: Process(LD,SHCMP)
        Begin
         -- 1) Register loading
         -- 2) BUSY Set to high (at beginning of shifting)
         If LD'EVENT and LD='1' and BUSY='0' then
            PREG <= PARIN;
            BUSY <= '1';
         End If;
         -- 3) BUSY set to low (at end of shifting)
         If SHCMP'EVENT and SHCMP='1' then
            BUSY <= '0';
         End If;
        End Process LOAD;
```

Algorithmic Model for Parallel to Serial Converter (PARSER)


```
        SHIFT: Process(BUSY,SHCLK)
        variable COUNT: Integer;
        variable OREG: BIT_VECTOR(0 to 7);
        Begin
         -- 1) Shift Initialize
         If BUSY'EVENT and BUSY='1' then
            COUNT:=7;
            OREG:= PREG;
            SHCMP <='0';
         End If;
         If SHCLK'EVENT and SHCLK='1' and BUSY='1' then
            -- 2) Shift
             SOUT <= OREG(COUNT);
            COUNT := COUNT -1;
            -- 3) Shift Complete
            If COUNT<0 then
               SHCMP <= '1';
            End If;
         End If;
        End Process SHIFT;
End ALG;
```

Modeling for Synthesis

System Design Technologies

- **System designer can use three Design paradigms:**
  - **Full Custom Design:** the designer has full control on all part of circuit and layouts can be optimized each transistor for optimum performance.
  - **ASICs (Application Specific Integrated Circuits) Design:** for fixed logic where high speed is required and implemented using:
    - Standard Cells
    - Gate Array
  - **Programmable Logics Design:** for reconfigurable logic that allows a rapid change of function and can be implemented using:
    - SPLDs (Simple Programmable Logic Devices)
    - CPLDs (Complex Programmable Logic Devices)
    - FPGAs (Field Programmable Gate Array)

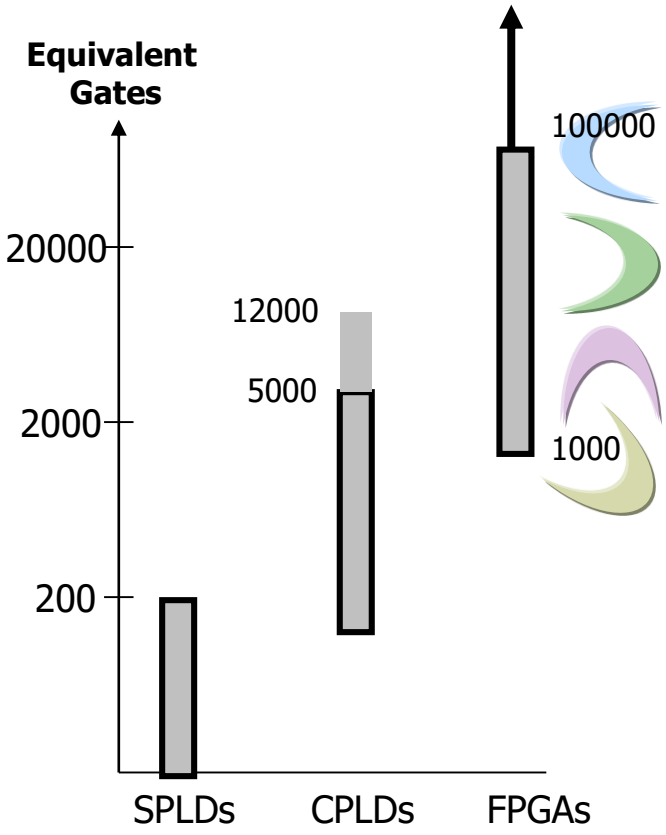
ASIC Design Process

- **ASIC is an integrated circuit produced for a specific application and produced in relatively small volumes.**
- **Although this definition is technology independent and can be implemented using full custom, PLDs, FPGAs, gate array or standard cells, the term ASIC reserved for fabricating the chip with gate array and standard cells.**
- **A Gate Array chip contains prefabricated adjacent rows of PMOS and NMOS transistors that the connection layers masks (metal layers, contacts and vias) can be created by users (it is also called Mask Programmable Gate Arrays or MPGAs).**
- **The ASICs are also referred as Cell-Based Integrated Circuits (CBICs) and can be designed using standard cells library consist of:**
  - **SSI logic: and, or, nand, nor, not, buffer and registers.**
  - **MSI logic: decoders, encoders, adders, comparators and parity checkers.**
  - **Data path: bus extractors, register files, shifters and ALUs.**
  - **Memory: RAMs and ROMs.**
  - **System level blocks: multipliers, micro-controllers, UARTs and RISC cores.**

Programmable Devices

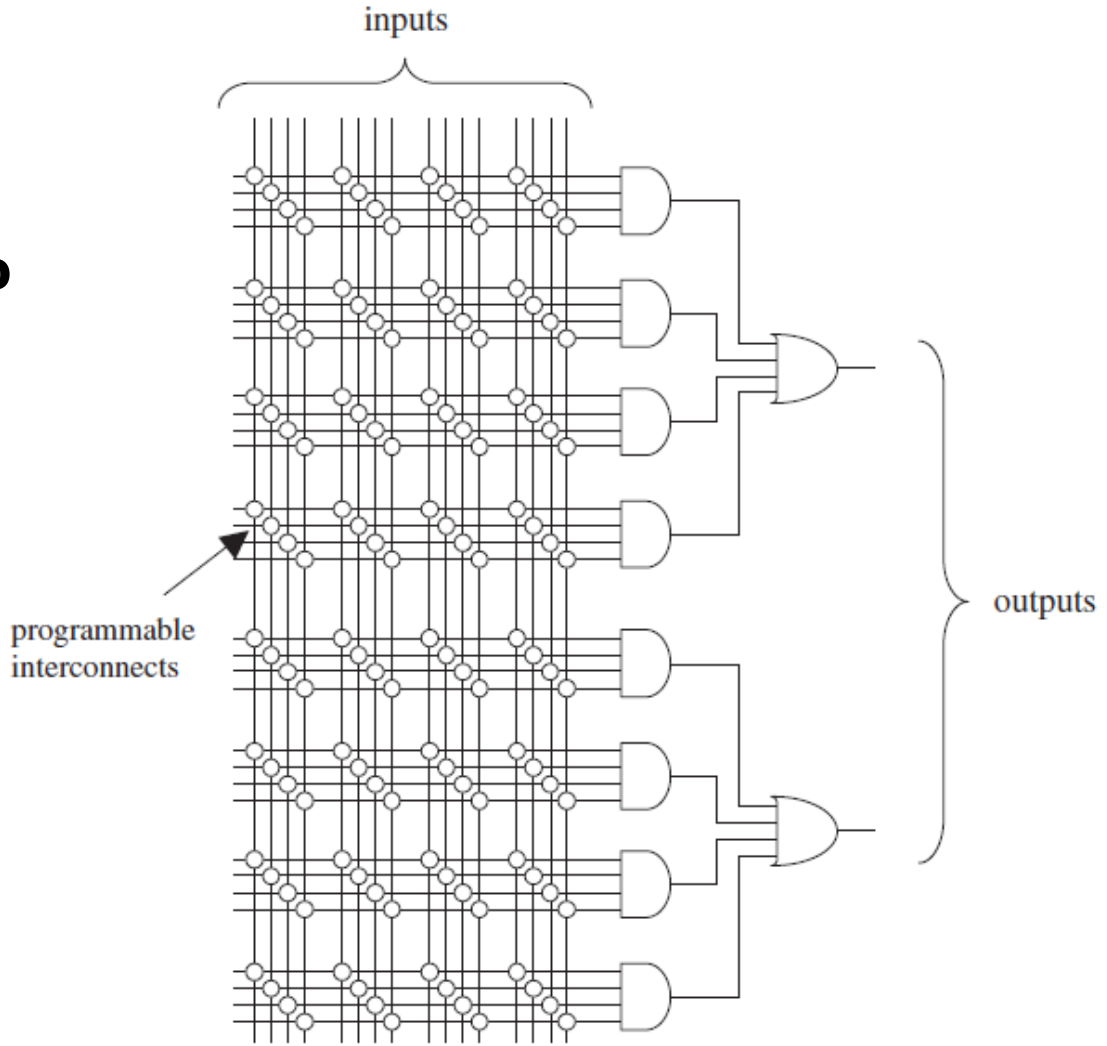
- **Programmable Logic Devices (PLDs) were introduced in the mid 1970s. The idea was to construct combinational logic circuits that were programmable. However, contrary to microprocessors, which can run a program but posses a fixed hardware, the programmability of PLDs was intended at the hardware level.**

- **In other words, a PLD is a general purpose chip whose hardware can be reconfigured to meat particular specifications.**
  - **SPLD: Simple PLD**
    - **PLA or PAL**
  - **CPLD: Complex PLD**
    - **Multiple SPLDs onto a single chip**
    - **Programmable interconnect**
  - **FPGA: Field-Programmable Gate Array**
    - **An array of logic blocks**
    - **Interconnect resources**
    - **A high ratio of flip-flops to logic resources**



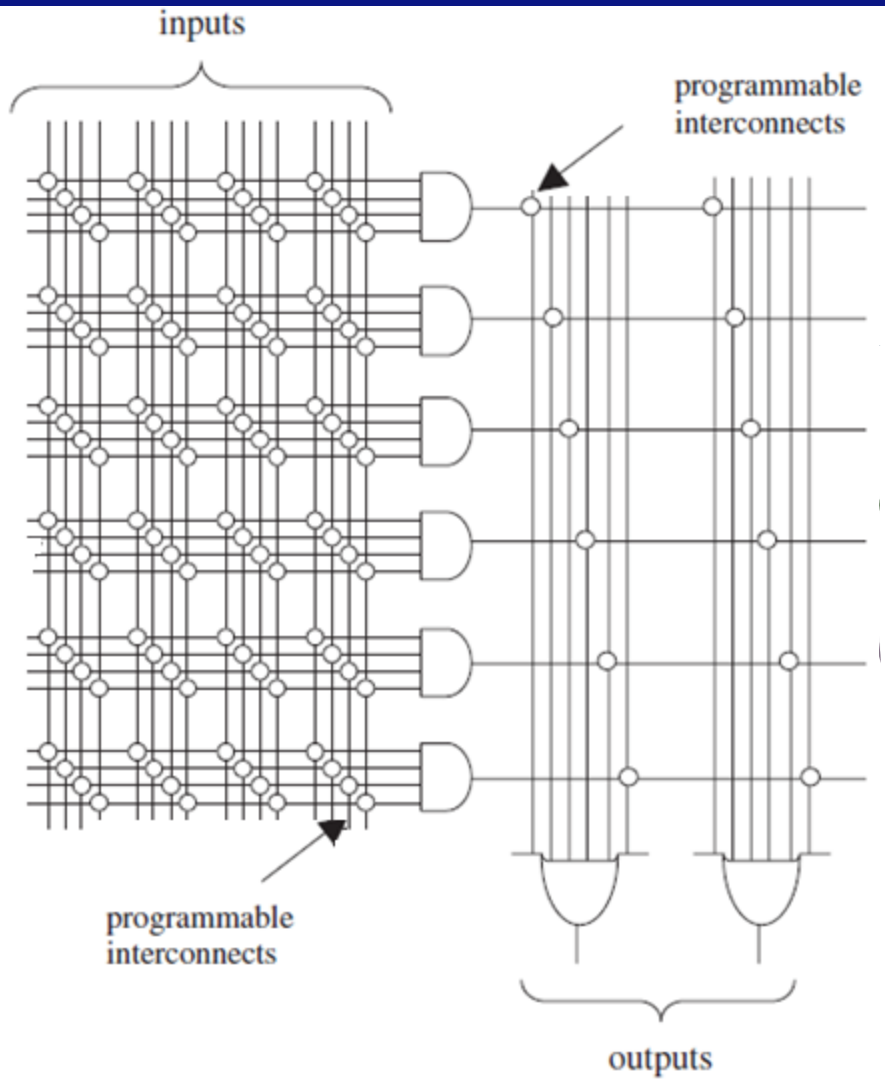
SPLD
PAL (Programmable Array Logic)

- **The PAL (Programmable Array Logic) allowing the implementation of combinational circuits (no flip-flops).**
  - **Realizing logic functions in SOP form**
  - **Only AND planes are programmable**



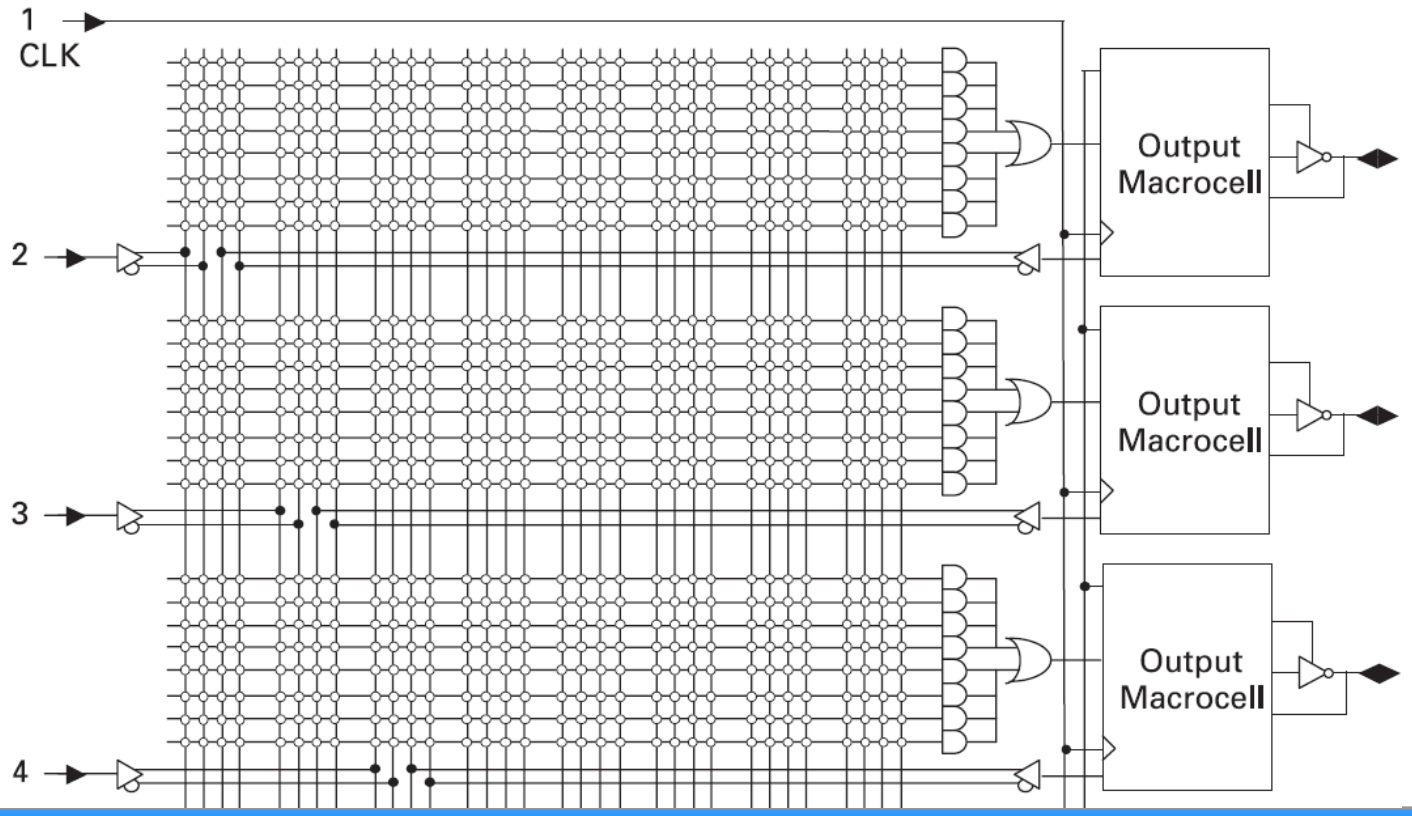
SPLD
PLA (Programmable Logic Array)

- **The PLA (Programmable Logic Array) allowing the implementation of combinational circuits (no flip-flops).**
  - **Realizing logic functions in SOP form**
  - **Both AND and OR planes are programmable**

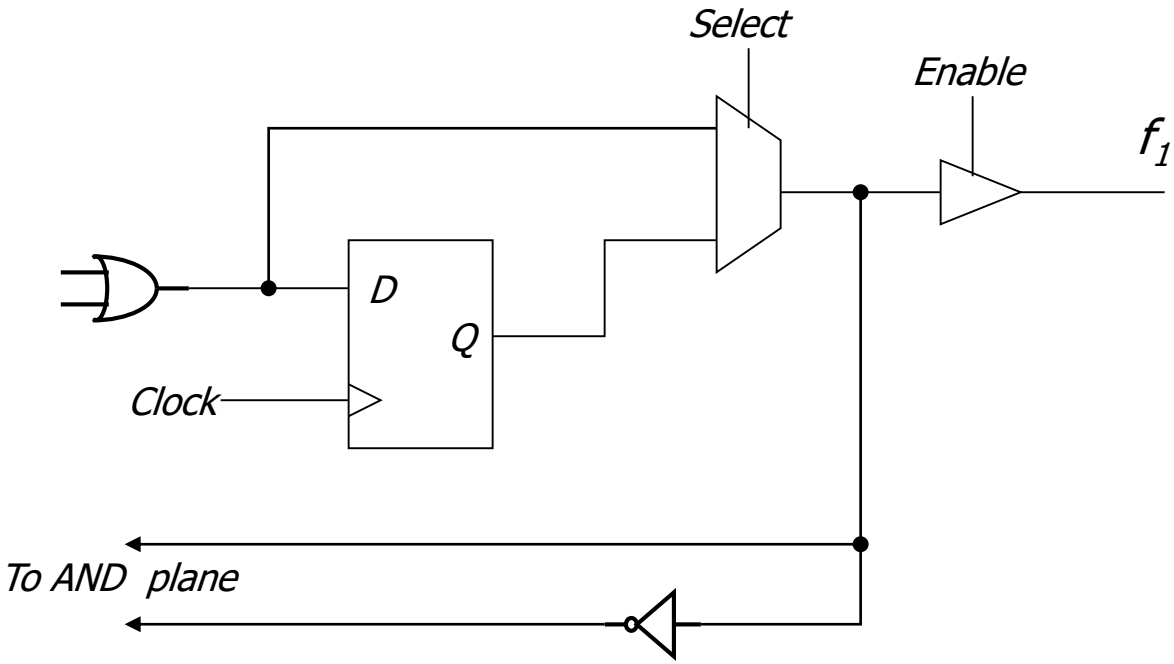


SPLD with Macrocells

- **In the beginning of the 1980s, additional logic circuitry was added to each SPLD output cell, called Macrocell, contained flip-flop and multiplexers then simple sequential functions could be implemented.**



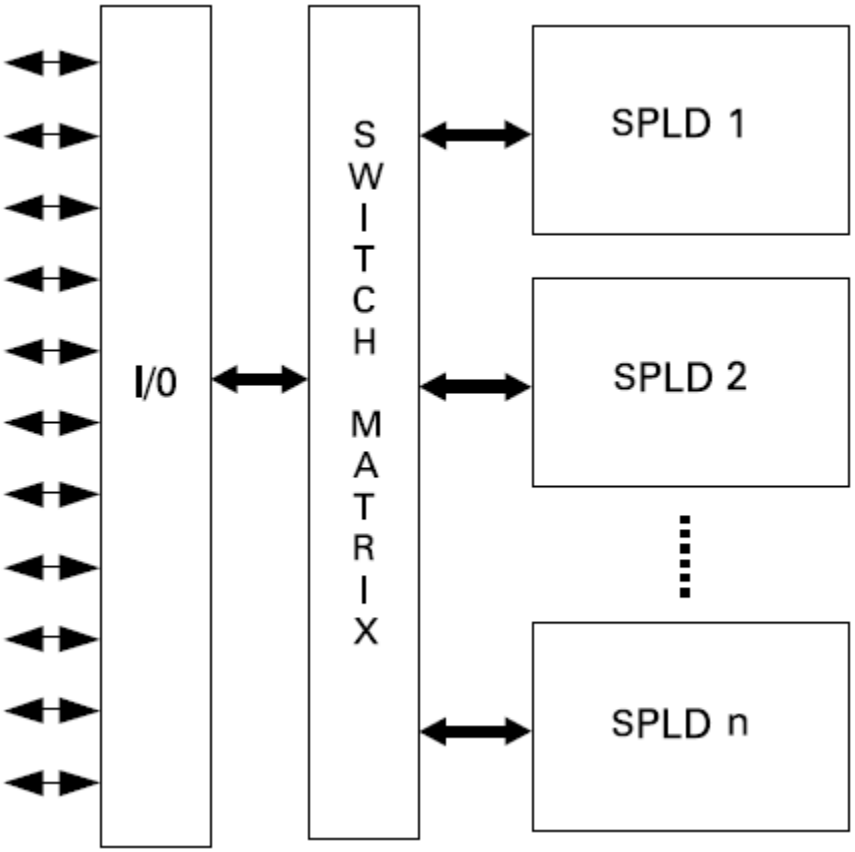
Macrocell



Select=0, combinational mode
Select=1, Registered mode

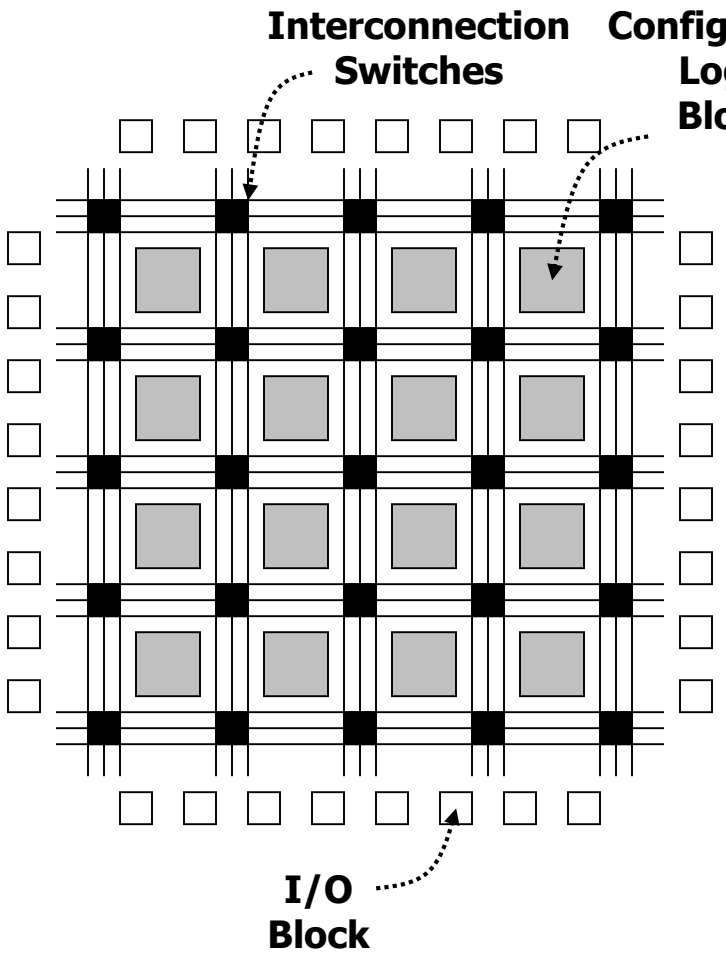
Complex PLD

- **The Complex PLD s (CPLDs) consist of Multiple SPLDs onto a single chip Programmable interconnect.**

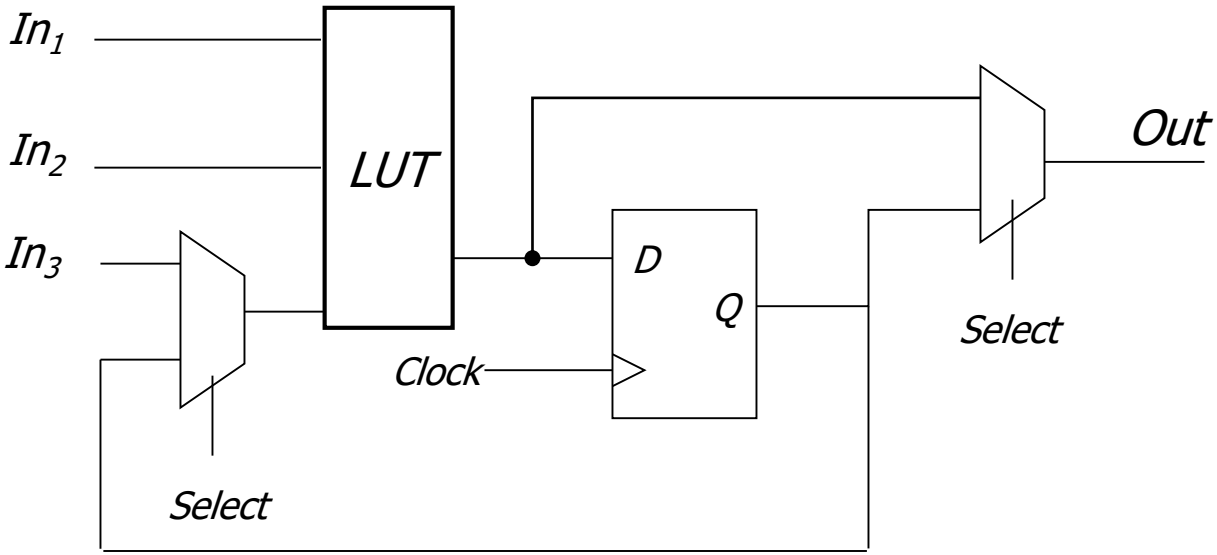


FPGA

- FPGAs do not contain AND or OR planes and use for implementation of relatively large logic circuits.
- FPGAs contain:
  - Configurable Logic blocks (CLB)
  - I/O blocks
  - Interconnection wires and switches
- All parts of FPGAs are Programmable.
- The programmable logic and routing interconnect of FPGAs makes them flexible and general purpose but at the same time it makes them larger, slower and more power consuming than standard cell ASICs.

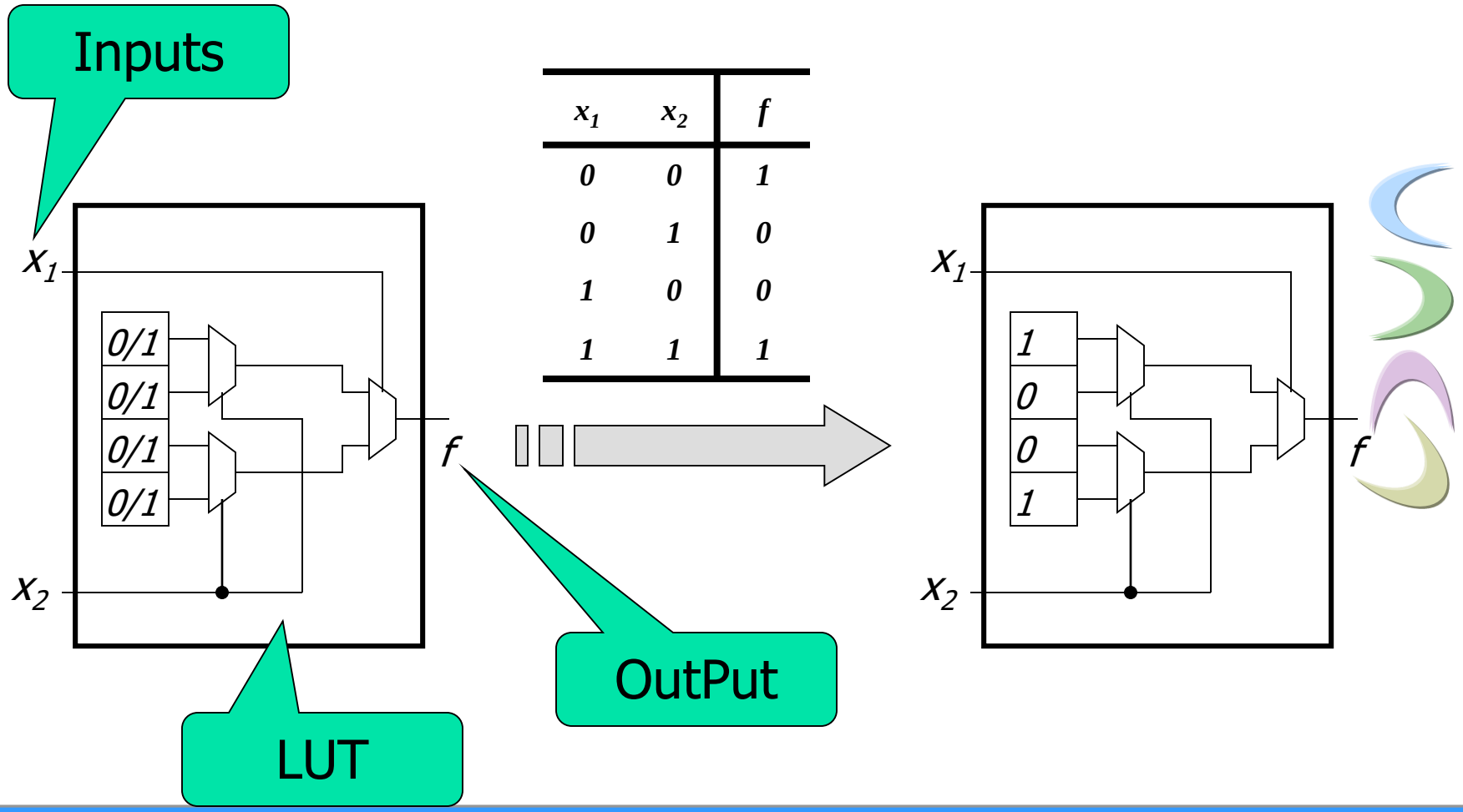


FPGA
General Form of Logic Cell (CLB)



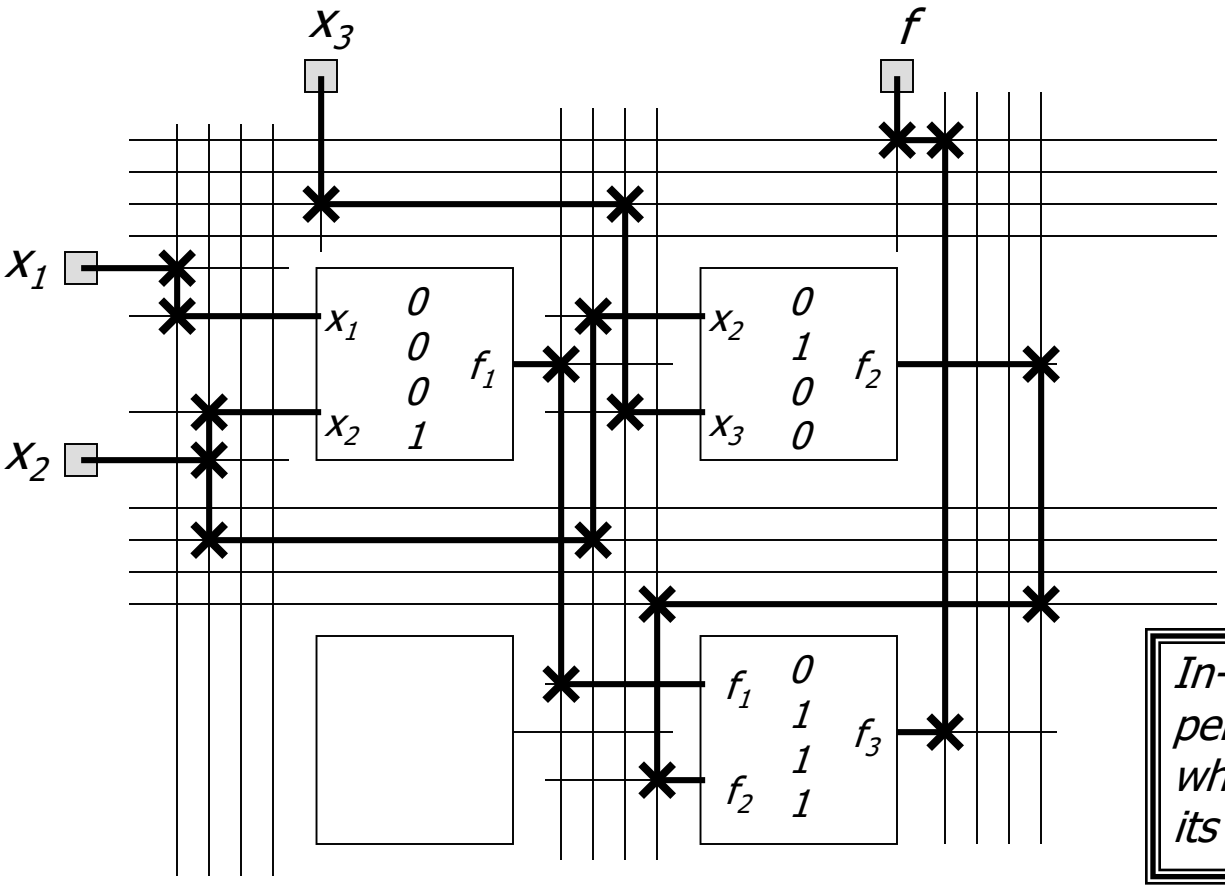
- **The LUT is a multiplexer that can implement all combinational function of its inputs.**
- **The storage cells in the LUTs in an FPGA are volatile.**
- **Volatile: losing stored contents whenever the power is off.**
- **Using external PROM to hold data permanently.**
- **The storage cells are loaded automatically from PROM when the chip is initialized.**
- **The Select pins of multiplexers can be used to implement sequential or combinational circuits.**

FPGA
LUT-Based Logic Block



x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	1

A Programmed FPGA



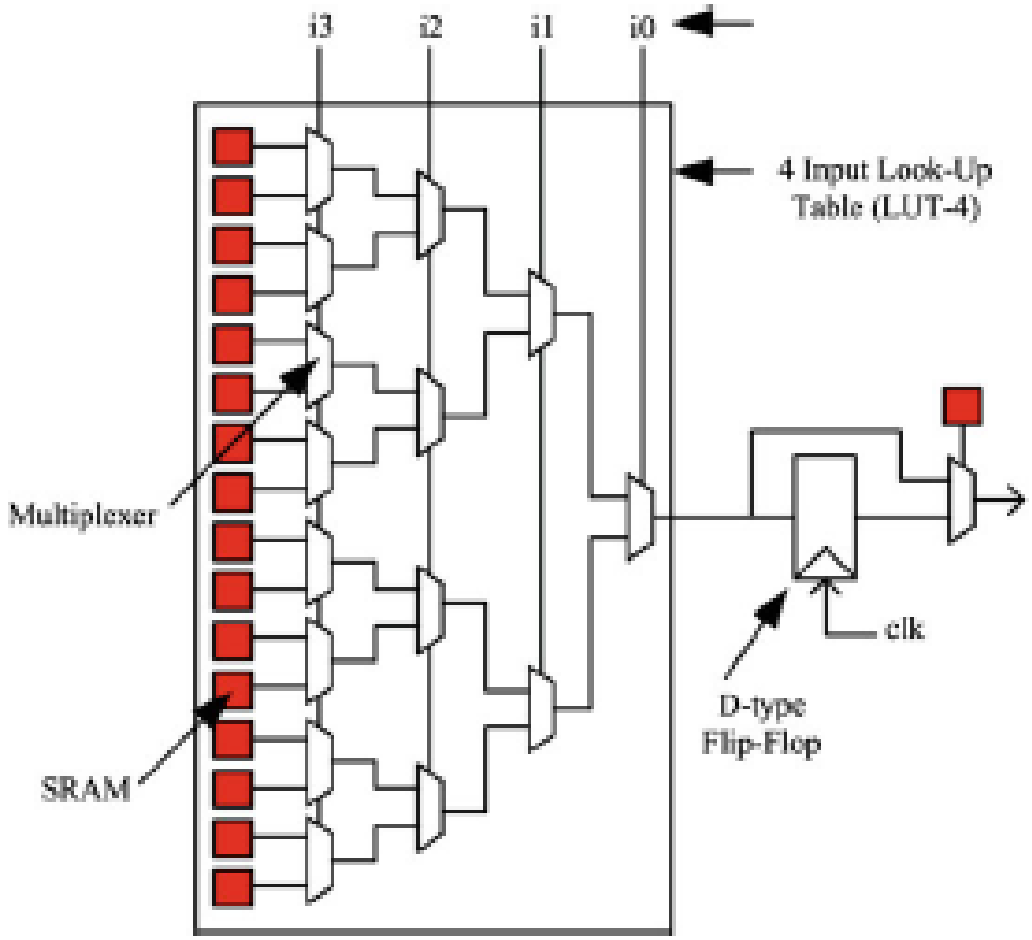
$$f_1 = x_1 x_2$$

$$f_2 = \overline{x_2} x_3$$

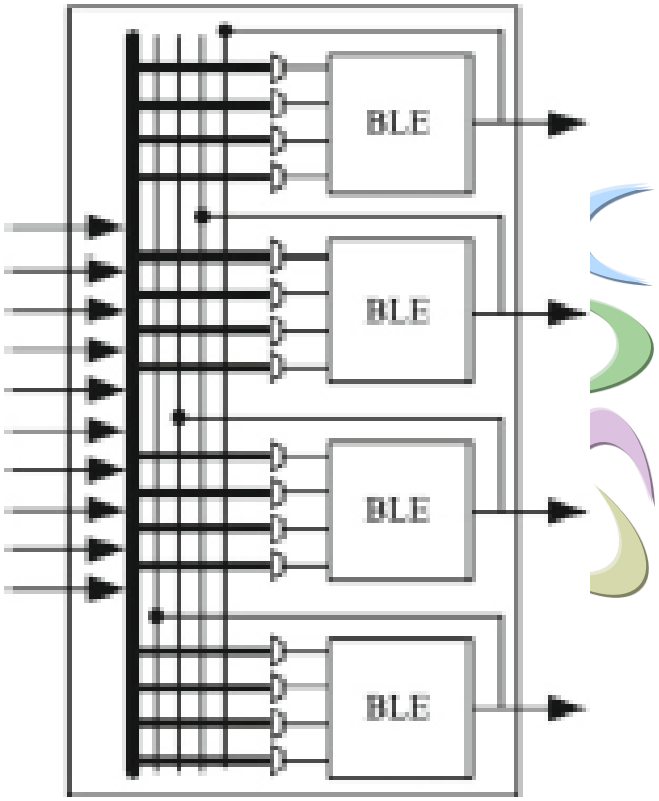
$$f_3 = x_1 x_2 + \overline{x_2} x_3$$

In-system programming (ISP) : performing the programming while the chip is still attached to its circuit board

FPGA
Configurable Logic Block (CLB)

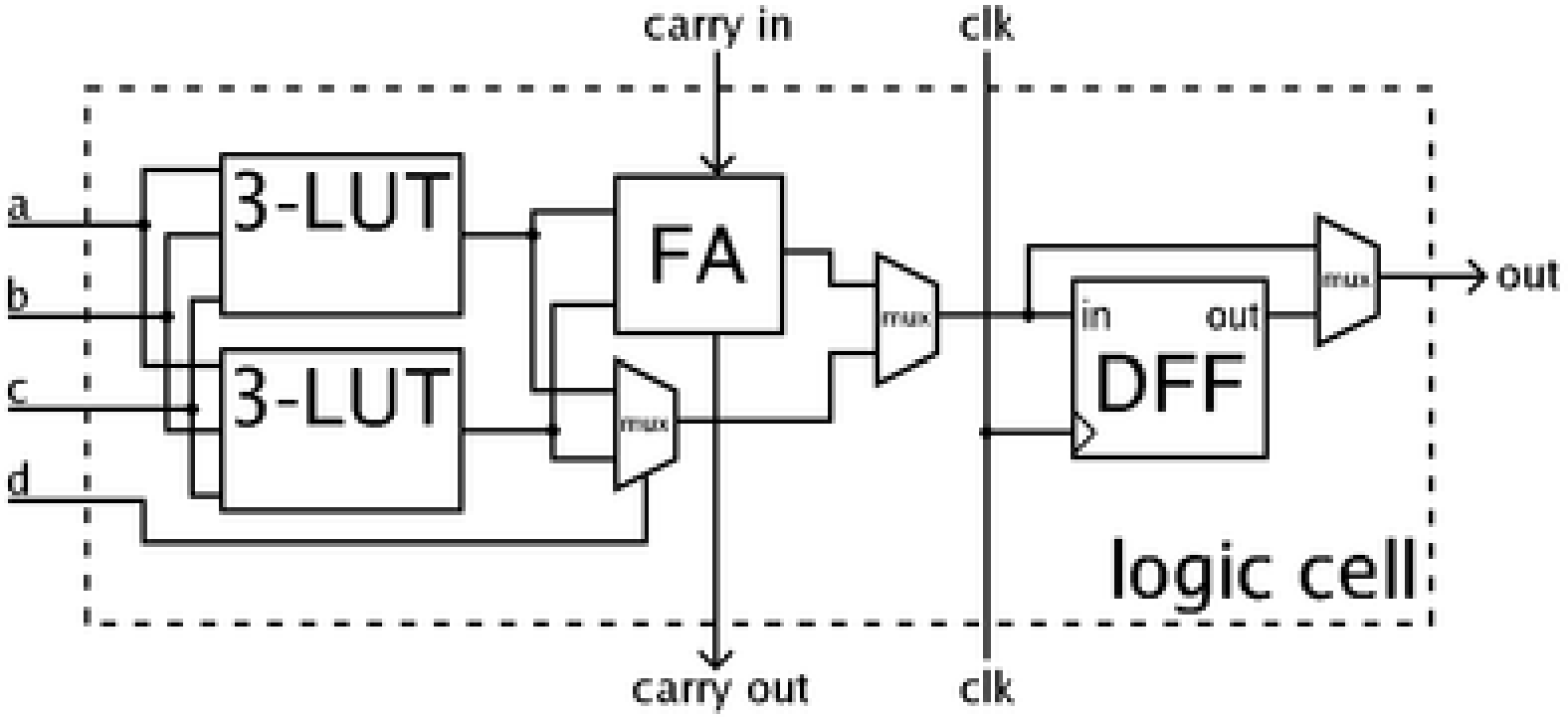


Basic Logic Element (BLE)



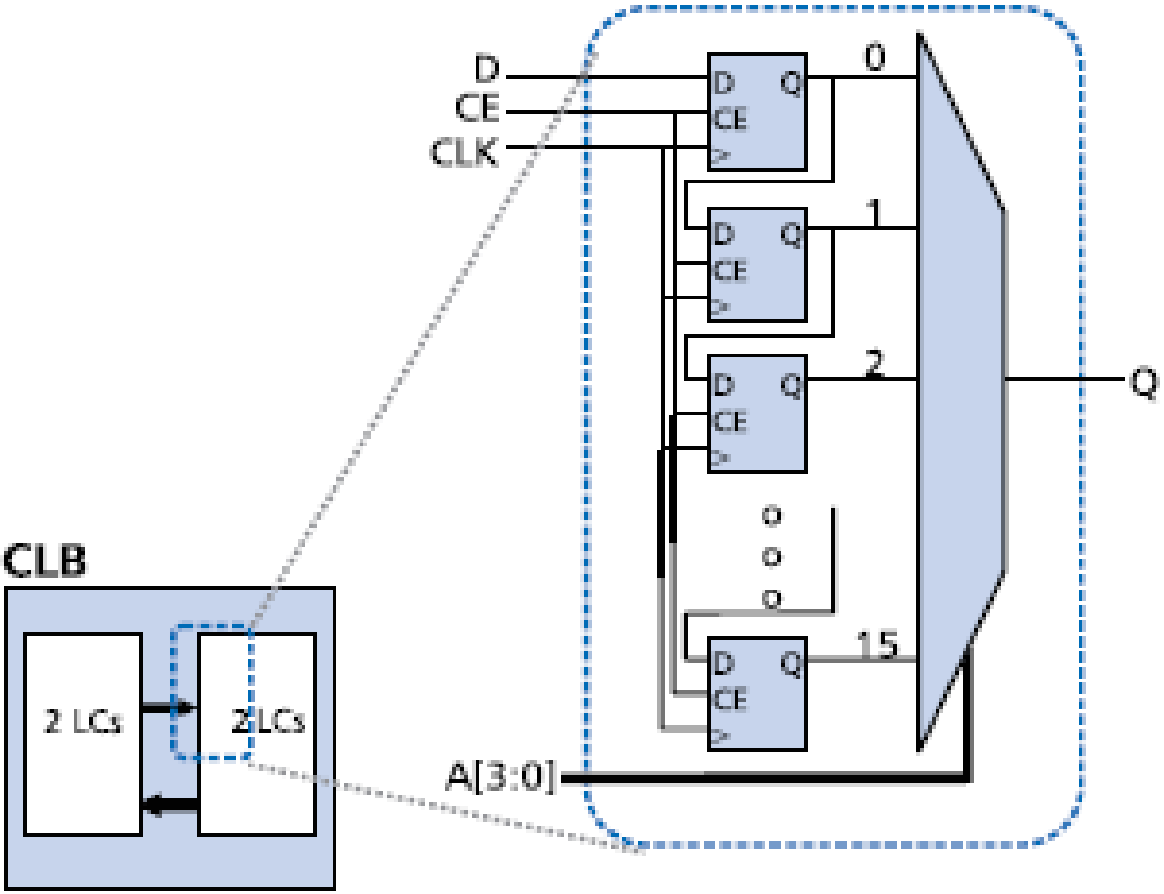
CLB with 4 BLE

FPGA
CLB with Full Adder



FPGA
Xilinx CLB (Virtex)

Virtex LUT as a Shift Register



FPGA
Configurable Logic Block (CLB)

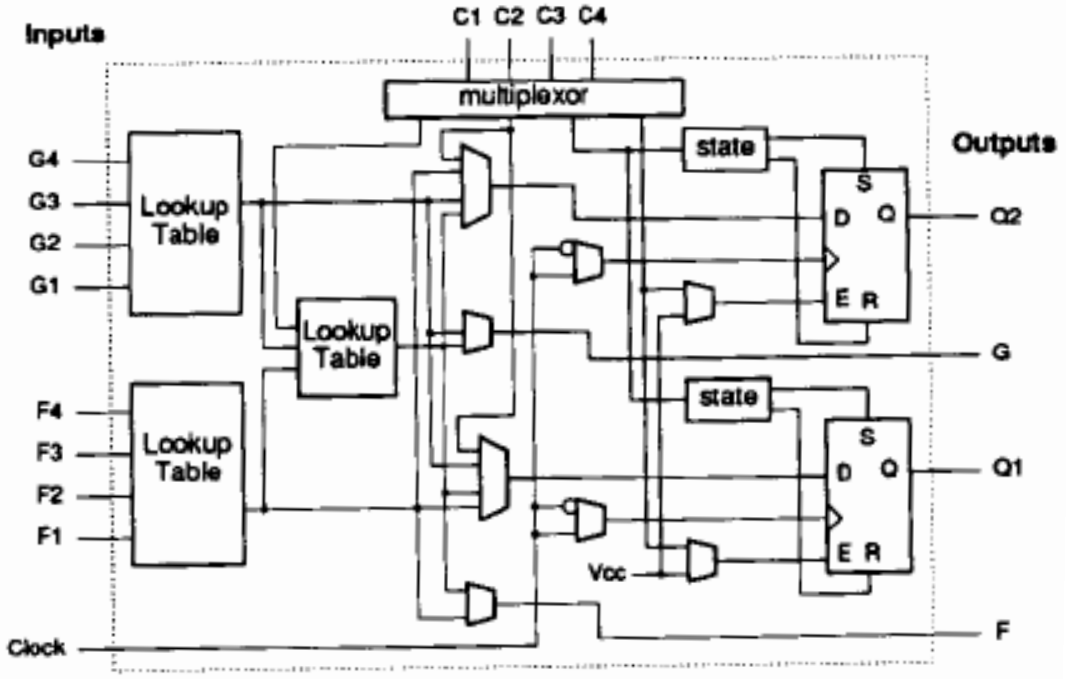
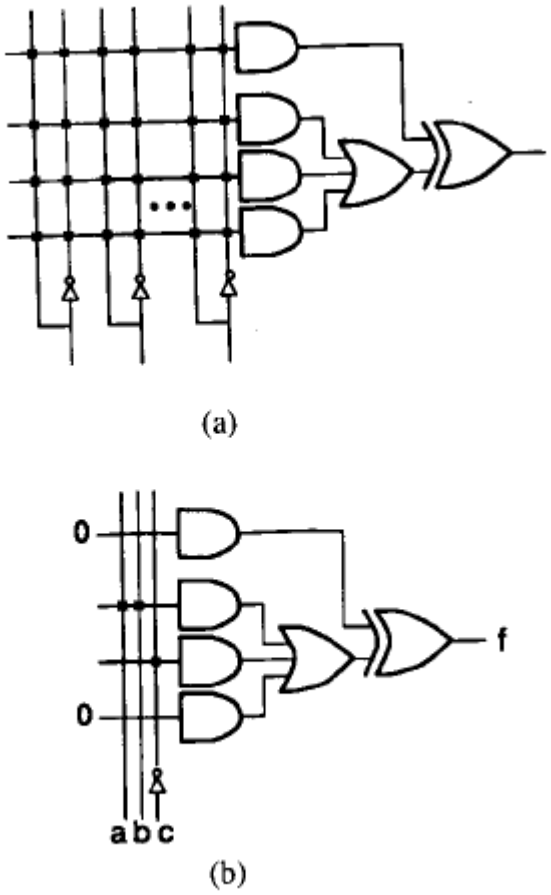


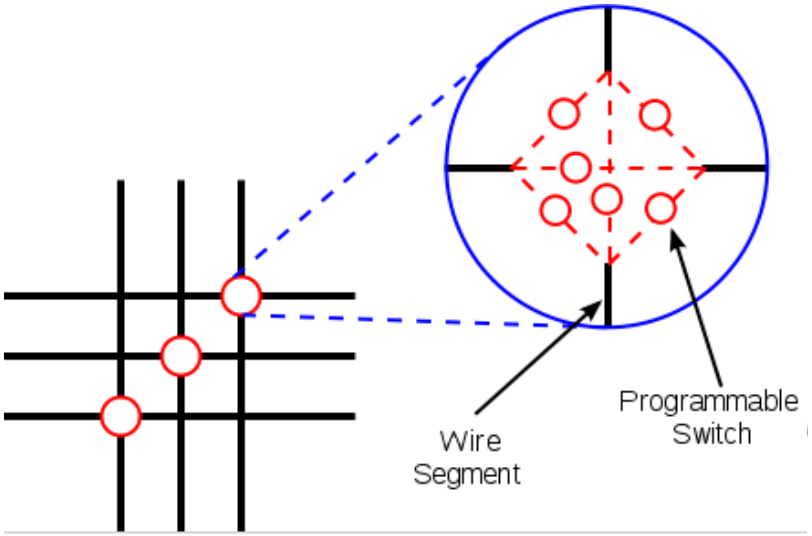
Fig. 13. The Xilinx 4000 logic block.



The Altera 5000 Series logic block.

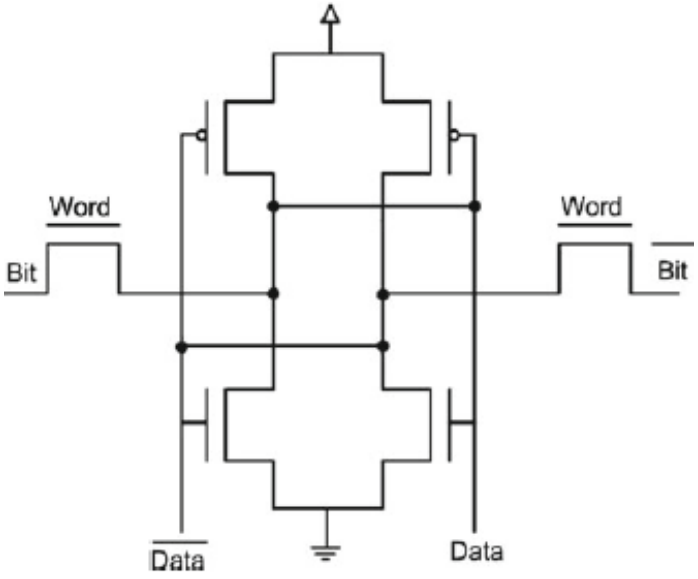
Programming Technologies

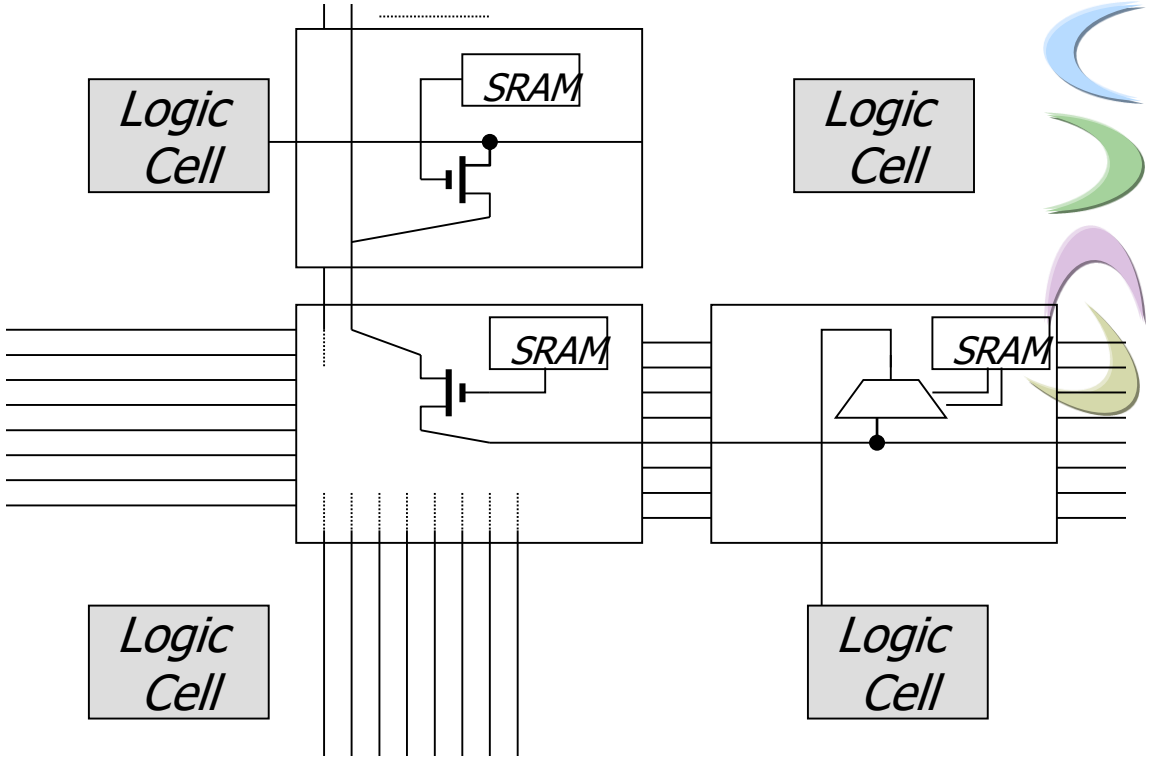
- Metal-alloy Fuses (Old)
  - SPLD
- SRAM Programming Technology
  - FPGA
  - CPLD
- EEPROM (Floating Gate Programming Technology)
  - SPLD
  - CPLD
- Antifuse Programming Technology
  - FPGA
- Programmable Switches



Programming Technology
SRAM (FPGA, CPLD)

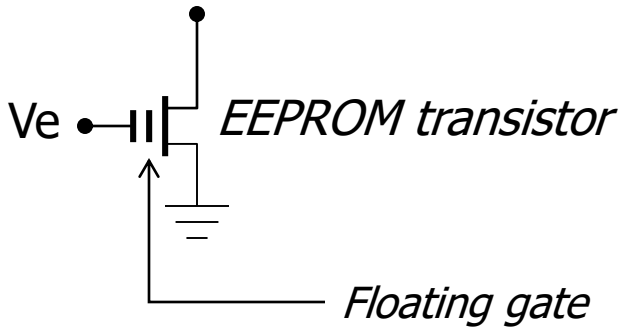
- **SRAM employment**
  - **Programmable switches (CPLD & FPGA)**
    - **An NMOS transistor which its gate controlled by a SRAM cell (Pass transistor switch)**





Programming Technology
Floating Gate (CPLD, SPLD)

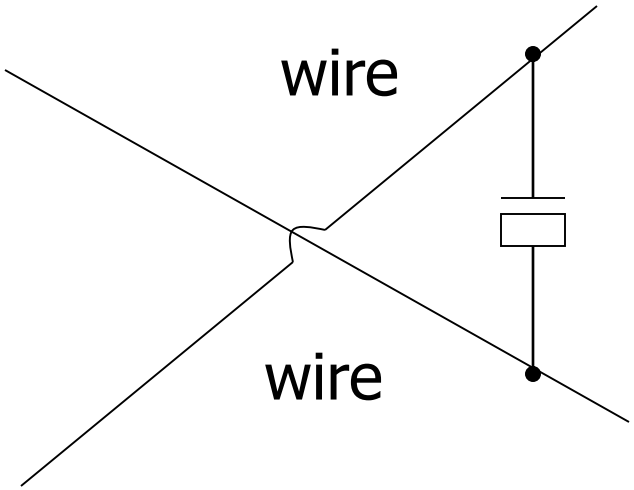
- **EPROM (OLD-not use in PLDs)**
- **EEPROM or Flash**
  - **An NMOS transistor with two gates**
    - **Programming mode:**
      - **Ve=12v → channel current=high → Float gate is charged because of tunneling → increase V_{th}**
    - **Reading mode:**
      - **Ve=5v→ transistor is off at all**

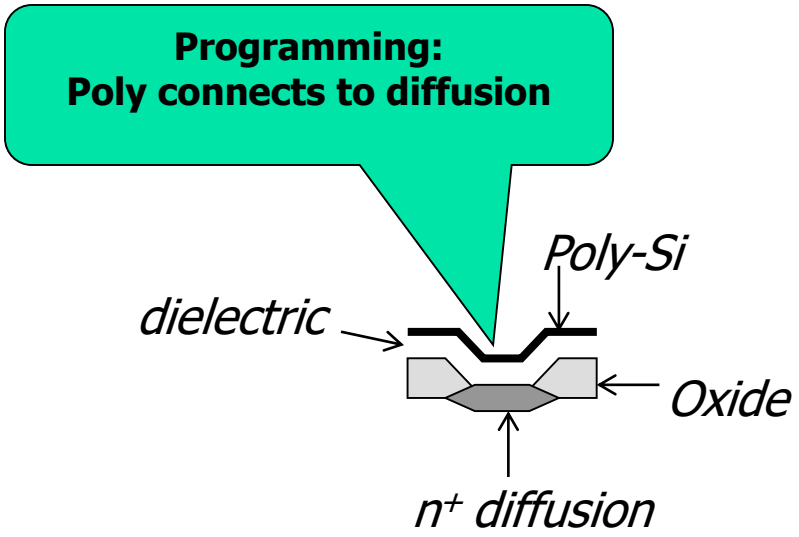


Programming Technology
Antifuse (FPGA)

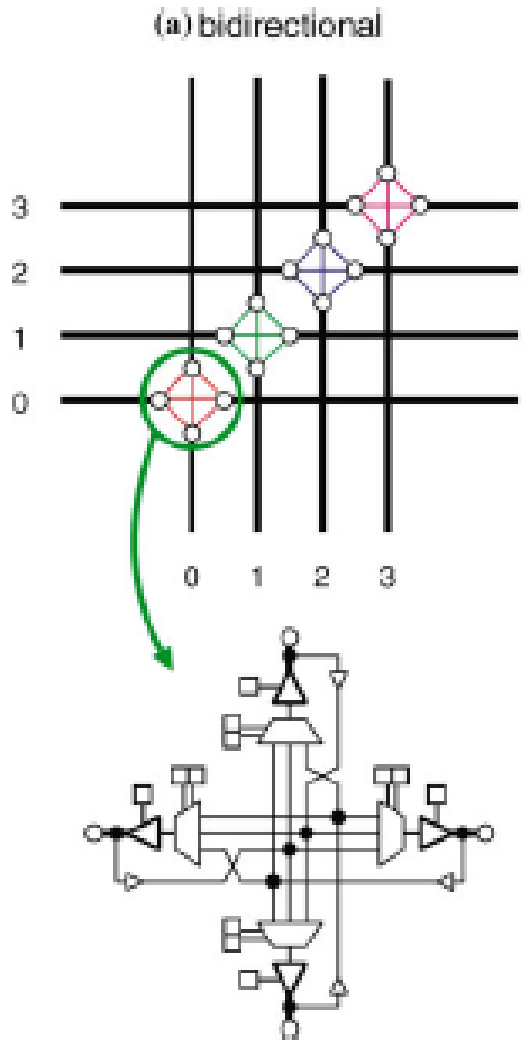
- **Antifuse device:**
  - **An electrically programmable two-terminal device**
  - **A normally "off" device**
  - **When a programming voltage is applied across the terminal, it changes from high resistance to low resistance**

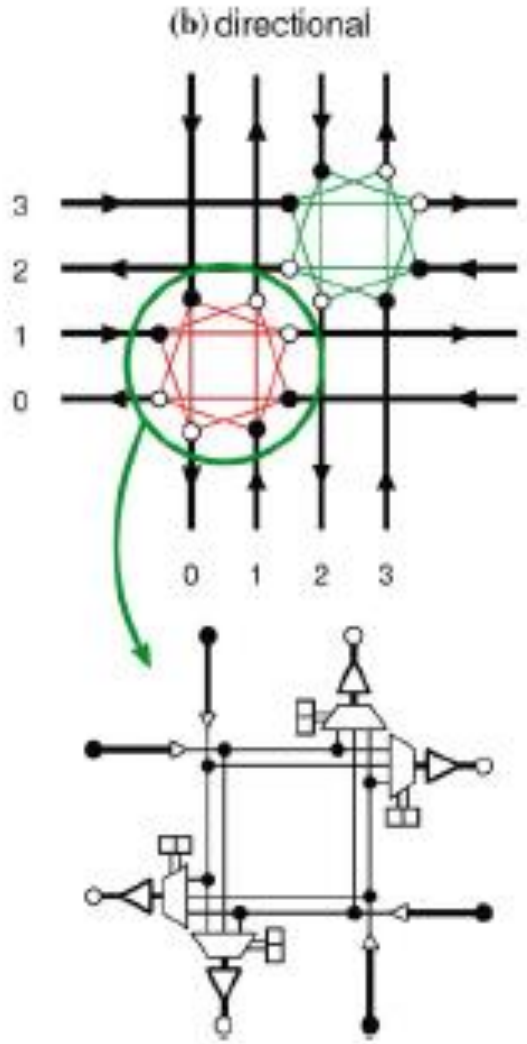
- **The parasitic capacitance of an un-programmed antifuse device is significantly lower than for other programming technologies**





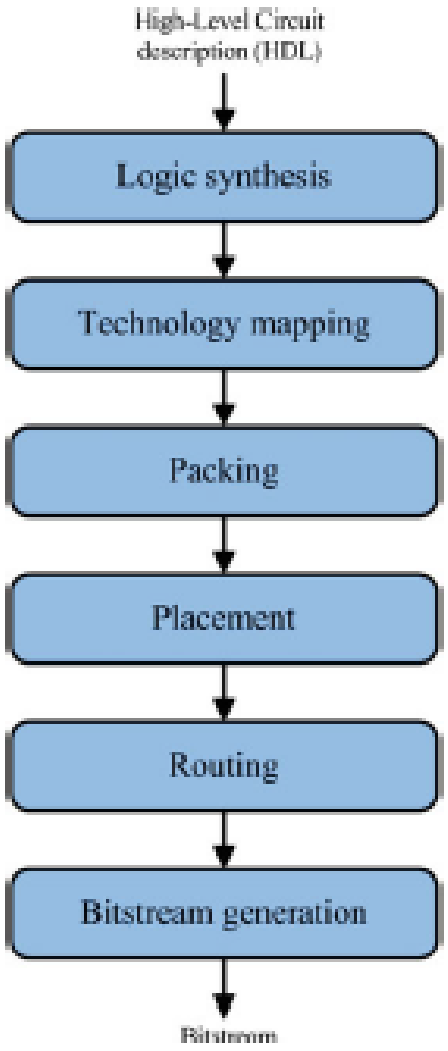
Programmable Switches





FPGA Design Process

- **The design description is written in a Hardware Description Language (HDL) and converted to a stream of bits that is eventually programmed on the FPGA. This process divided into six distinct steps, namely:**
  - **Logical synthesis**
  - **Technology mapping**
  - **Clustering /Packing**
  - **Placement**
  - **Routing**
  - **Timing Analysis**

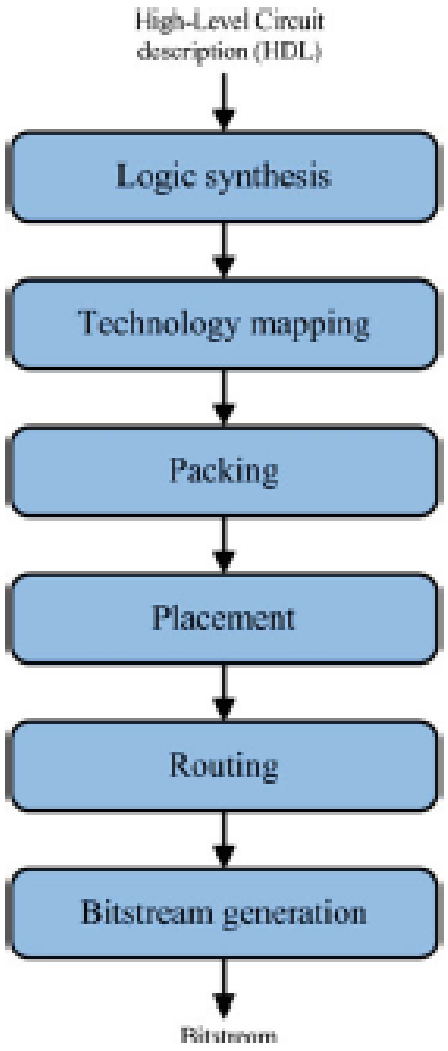


FPGA Design Process

- **Logical synthesis:** Logical synthesis transforms an HDL description (VHDL or Verilog) into a set of boolean gates and Flip-Flops.

- **Technology mapping:** The FPGA consists of the library of cells with k-input LUTs and flip-flops. Therefore, FPGA technology mapping involves transforming the Boolean network into k-bounded cells. Each cell can then be implemented as an independent k-LUT.

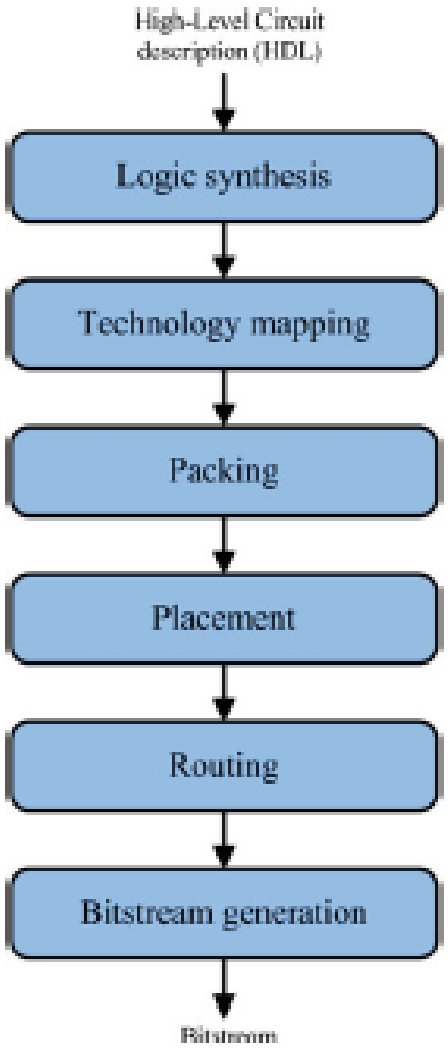
- **Clustering /Packing:** Clustering algorithms can be broadly categorized into three general approaches:
  - **Top-down** approaches partition the LBs into clusters by successively subdividing the network or by iteratively moving LBs between parts.
  - **Depth-optimal** solutions attempt to minimize delay at the expense of logic duplication.
  - **Bottom-up** approaches are generally preferred for FPGA CAD tools due to their fast run times and reasonable timing delays.



FPGA Design Process

- **Placement:** Placement algorithms determine which logic block within an FPGA should implement the corresponding logic block (instance) required by the circuit. The optimization goals consist in placing connected logic blocks close together to minimize the required wiring (wire length-driven placement), and sometimes to place blocks to balance the wiring density across the FPGA (routability-driven placement) or to maximize circuit speed (timing-driven placement).

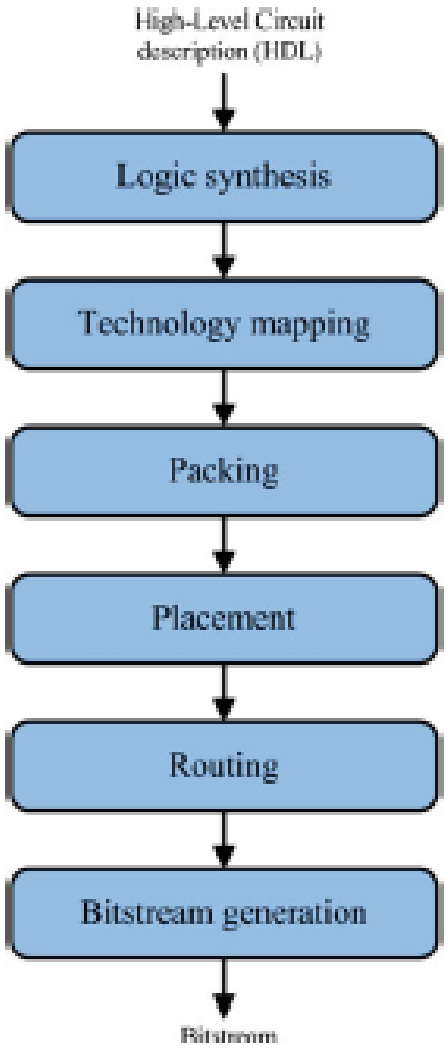
- **Routing:** The FPGA routing problem consists in assigning nets to routing resources such that no routing resource is shared by more than one net. *Path finder is the* current, state-of-the-art FPGA routing algorithm.



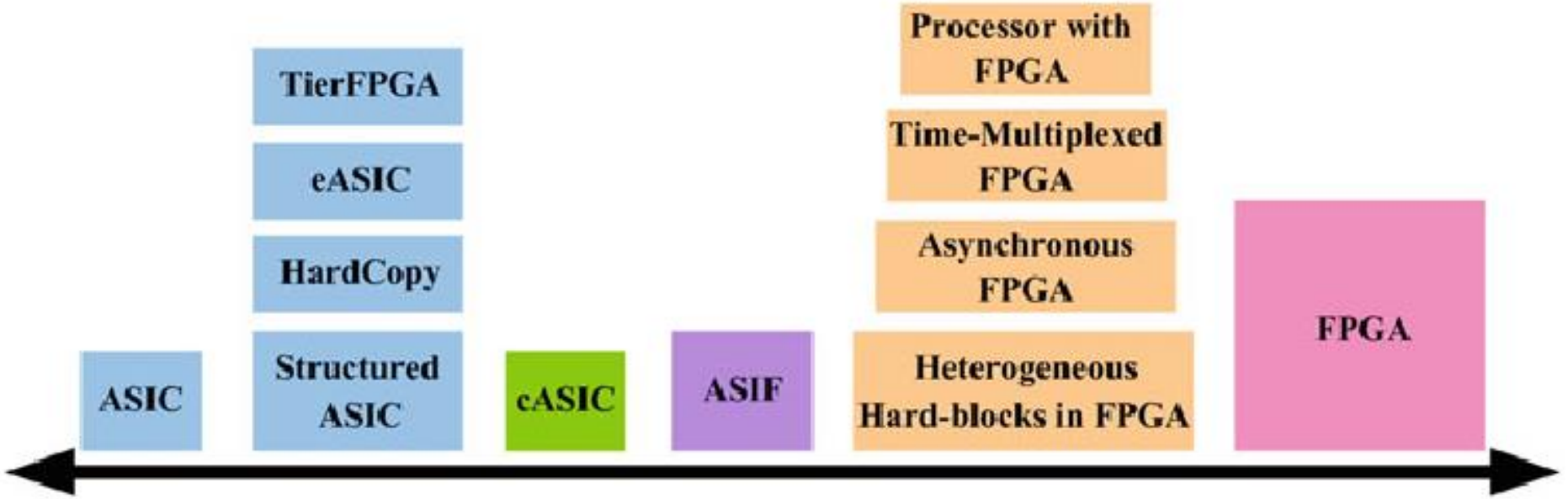
FPGA Design Process

- **Timing analysis:** Timing analysis is used for two basic purposes:
  - To determine the speed of circuits which have been completely placed and routed,
  - To estimate the slack of each source-sink connection during routing (placement and other parts of the CAD flow) in order to decide which connections must be made via fast paths to avoid slowing down the circuit.

- **Bitstream Generation:** Once a netlist is placed and routed on an FPGA, bitstream information is generated for the netlist. This bitstream is programmed on the FPGA using a bitstream loader. The bitstream of a netlist contains information as to which SRAM bit of an FPGA be programmed to 0 or to 1. The bitstream generator reads the technology mapping, packing and placement information to program the SRAM bits of Look-Up Tables. The routing information of a netlist is used to correctly program the SRAM bits of connection boxes and switch boxes.



Comparison of FPGA and ASIC



+ High Performance
+ High Volumn Production
+ Low Area
+ Low Power Consumption
– Low Reusability
– Low Flexibility
– High NRE
– High Time-to-market

High Flexibility +
High Reusability +
Low Time-to-market +
Low NRE +
High Area –
Low Performance –
High Power Consumption –
Low Volume Production –

End

Your Presentations Scheduled From Next Week