

# **EE332**

## **Digital System Design**

Dr. Yu Yajun  
Associate Professor,  
Department of Electrical and Electronic Engineering  
Office: S236, South Tower of COE  
Email: yuyj@sustech.edu.cn

## **Text Book**

- P.P. Chu, RTL hardware design using VHDL, John Wiley & Sons, 2006.

## **Reference Book**

- S. Yalamanchili, VHDL: A Starter's Guide, Prentice Hall, 2005
- Charles H. Roth, Jr. and Lizy Kurian John: Digital Systems Design with VHDL, Publishing house of electronics Industry

# Lecture Plan

- Lectures – 32 hours
- Lab exercises – 18 hours
- Two-person group projects – 14 hours. Sample group projects:
  - remote control electronic piano
  - smart phone control electronic motor
  - intelligent chess robot
  - intelligent unmanned grounded vehicle
  - wearable sports device
  - etc.

## Pre-requisites:

- Digital Electronics

## Examinations:

- Final Exam: 40% (open book, 2 hours)
- Continuous Assessment: 60%
  - Lab exercises and report: 30%
  - Group project and report: 30%
- **No plagiarism is tolerated. Zero marks are given to the part involving plagiarizing.**

# Outline

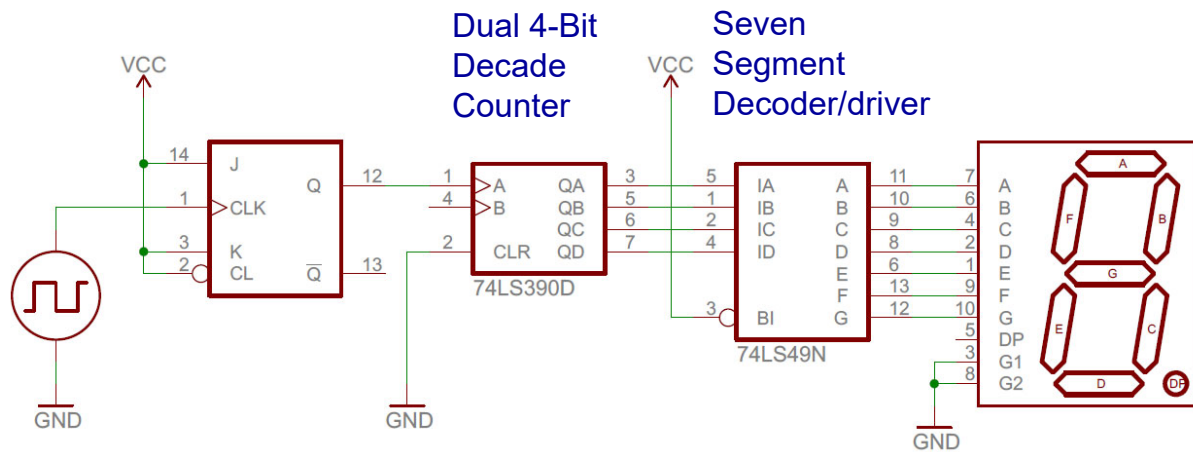
- Introduction
- Data Objects and Operands
- Concurrent statement
- Sequential statement
- Modeling Structure
- Modeling at the RT level
- Modeling at the FSMD level
- Parameterized Design
- Pipeline Design
- Subprograms, package and library
- Synthesis Of VHDL Code
- Clock and Synchronization

## Chapter 1 Introduction

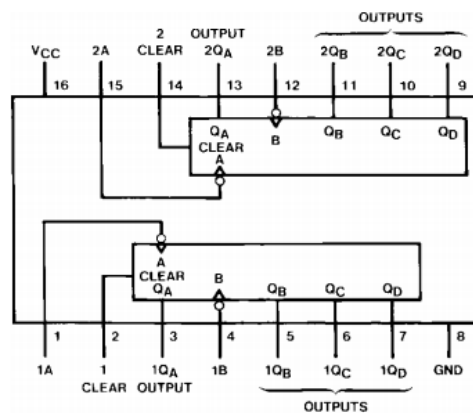
- Digital system design
  - System representation
  - Level of abstraction
  - Device technologies
- What is VHDL
- Basic VHDL concept via an example

## 1.1 Introduction to digital system design

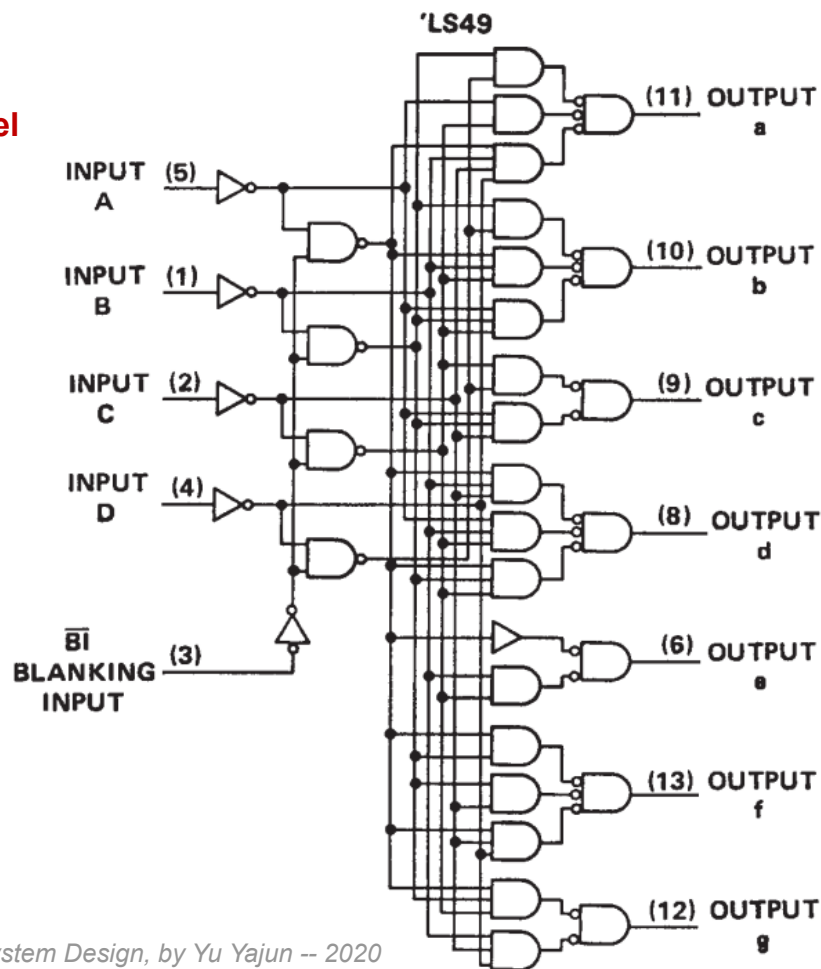
- Examples of Digital System: system level, or more accurately, register transfer level (RTL).



### 74LS390D Dual 4-Bit Decade Counter

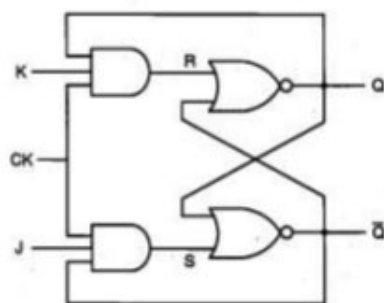


## 74LS49N Gate Level

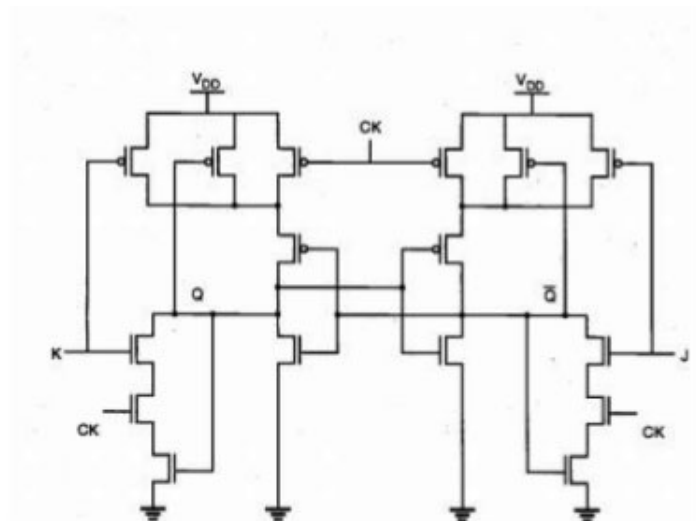


EE332 Digital System Design, by Yu Yajun -- 2020

9



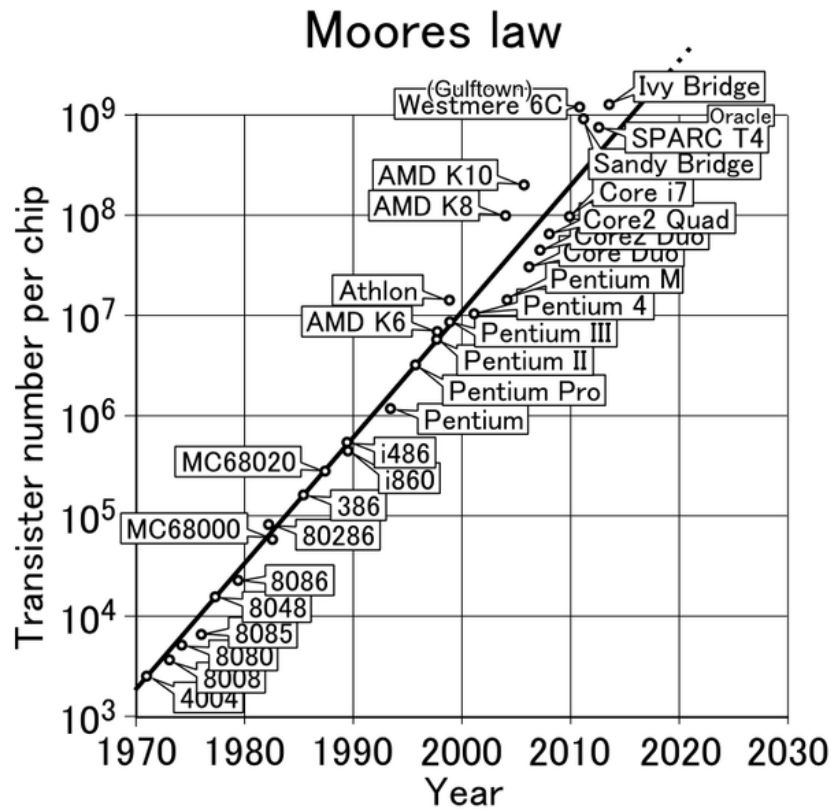
a) Gate level schematic of the clocked NOR based JK Latch



b) CMOS realization of the JK Latch

EE332 Digital System Design, by Yu Yajun -- 2019

10



EE332 Digital System Design, by Yu Yajun -- 2019

11



**Lose weight with a smart fork**



## Digital Systems

ELECTRONIC

V S

MECHANICAL



EE332 Digital System Design, by Yu Yajun -- 2019

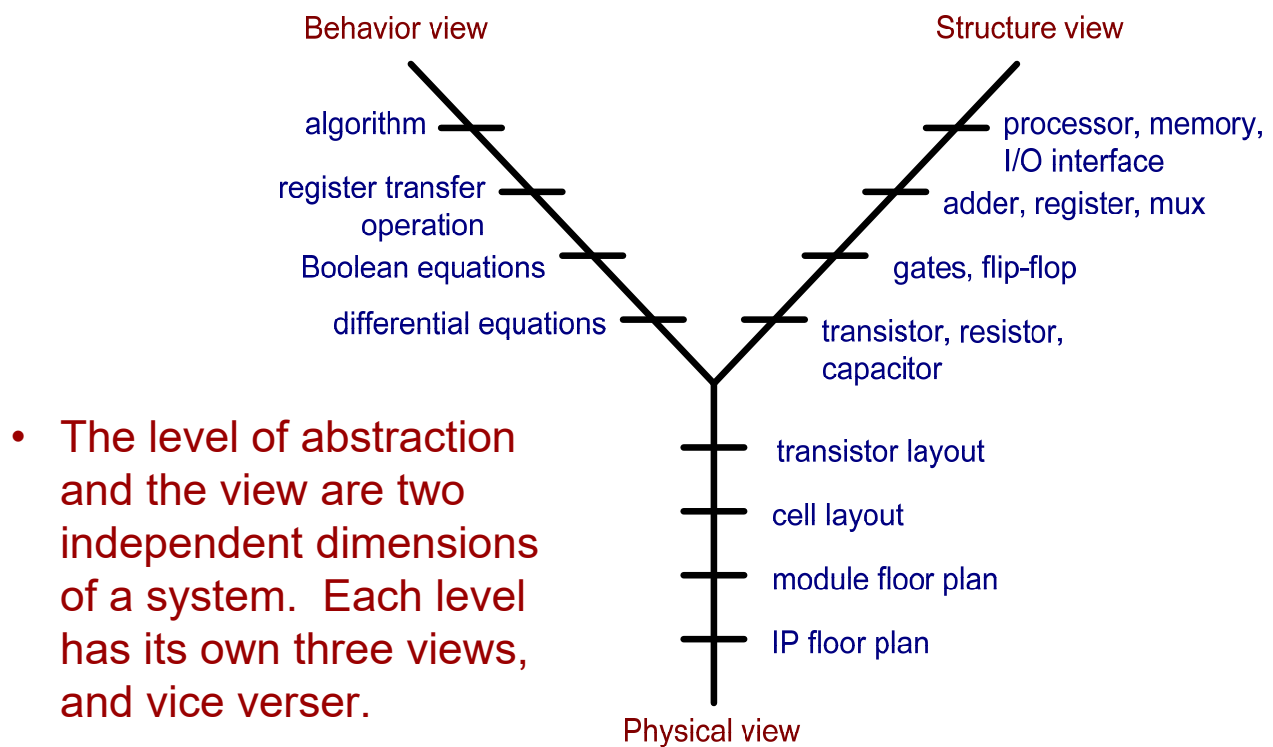
12

## Abstraction

- A key method of managing complexity is to describe a system in several levels of abstraction.
- An **abstraction** is a simplified model of the system, showing only the selected features and ignoring the associated details.
  - Transistor level
  - Gate level
  - Register transfer (RT) level
  - Processor level

## System Representation (View)

- **View:** different perspectives of a system.
  - Behavior view
    - describe the functionalities and i/o behavior
    - treat the system as a black box
  - Structural view
    - describe the internal implementation (components and interconnections)
    - essentially block diagram
  - Physical view
    - add more info to structural view: component size, component location, routing wires
    - e.g. layout of a printed circuit board



### Characteristics of each abstraction level

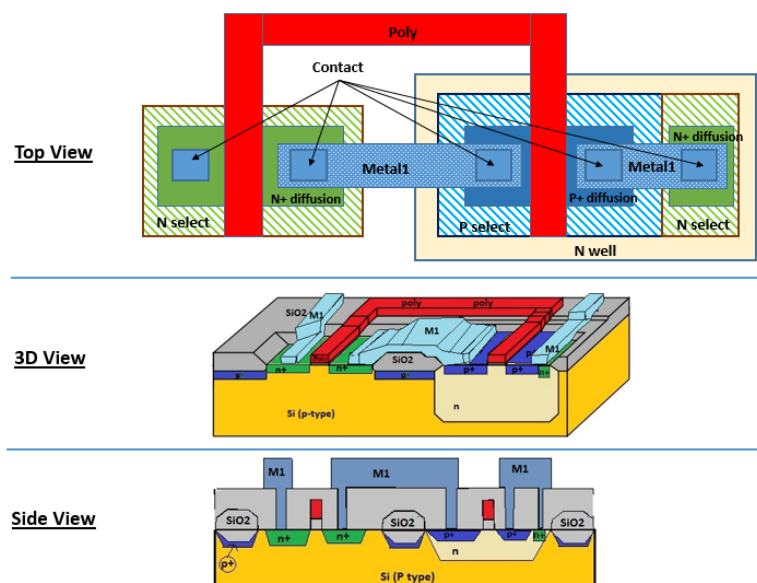
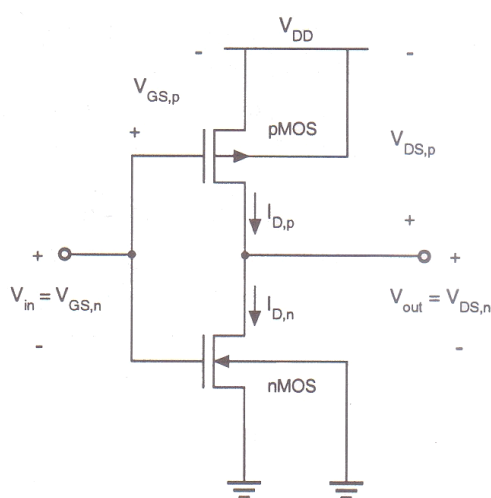
|            | Typical blocks       | Signal representation | Time representation | Behavioral description | Physical description |
|------------|----------------------|-----------------------|---------------------|------------------------|----------------------|
| Transistor | transistor, resistor | voltage               | continuous function | differential equation  | transistor layout    |
| Gate       | and, or, xor,        | logic 0 or 1          | propagation delay   | Boolean equation       | cell layout          |
| RT         | adder, mux, register | integer, system state | clock tick          | generalized FSM        | RT-level floor plan  |
| Processor  | processor, memory    | abstract data type    | event sequence      | natural language       | IP-level floor plan  |



## Device Technology

- **Fabrication of an IC**

- Transistors and connection are made from many layers (typical 10 to 15 in CMOS) built on top of one another
- Each layer has a special pattern defined by a mask
- An important aspect of IC is the length of a smallest transistor that can be fabricated
  - It is measured in micron ( $\mu\text{m}$ ,  $10^{-6}$  meter) in early time, but now in nano (nm,  $10^{-9}$  meter), and known as the **minimum feature size**.
  - e.g. we may say an IC is built with  $0.6\ \mu\text{m}$  process
  - The process continues to improve, as witnessed by Moore's law
  - The state-of-art process approaches a single digit of nm, for example 3nm.



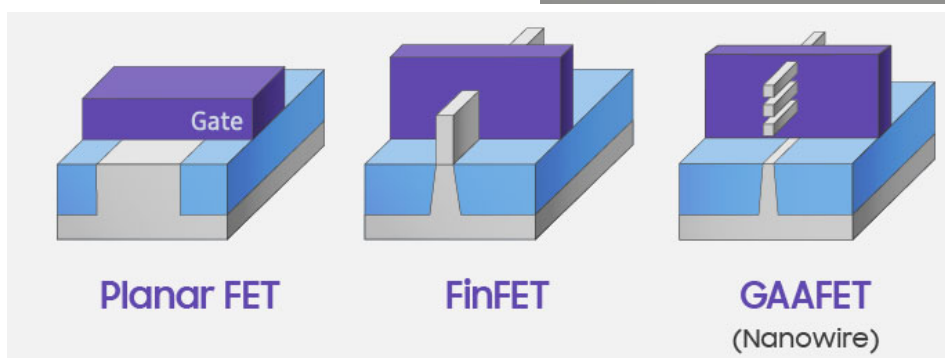
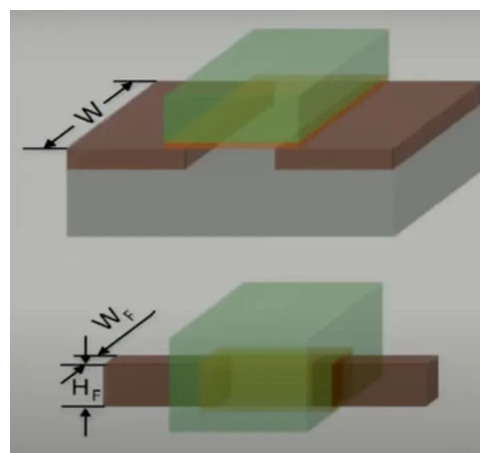
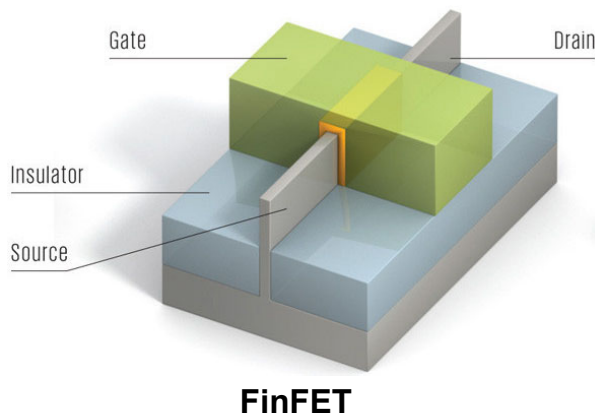
## Device Technology Development Trend

- **More Moore**

- Shrinking in the size (the channel length) of transistors
- For example, TSMC (Taiwan Semiconductor Manufacturing Company) owns all the product lines appeared since 1980's, including  $0.5\mu\text{m}$ ,  $0.35\mu\text{m}$ ,  $0.25\mu\text{m}$ ,  $0.18\mu\text{m}$ ,  $0.13\mu\text{m}$ ,  $90\text{nm}$ ,  $65\text{nm}$ ,  $40\text{nm}$ ,  $28\text{nm}$ ,  $22\text{nm}$ ,  $20\text{nm}$ ,  $16\text{nm}$ ,  $10\text{nm}$ ,  $7\text{nm}$ ,  $5\text{nm}$ ,  $3\text{nm}$ .

- **More Than Moore**

- Structure innovation, for example 3D integrate, including FinFET, GAA (Gate-All-Around) FET, etc.

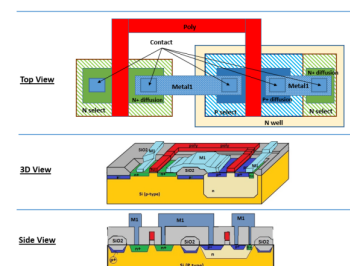


- **Classification of Device technologies:**

- Where customization is done:
  - in a fab (fabrication facility): ASIC (Application Specific IC)
  - in the “field”: Non-ASIC
- Classification
  - Full custom ASIC
  - Standard-cell ASIC
  - Gate array ASIC
  - Reprogrammable logic device: FPGA (field programmable gate array)

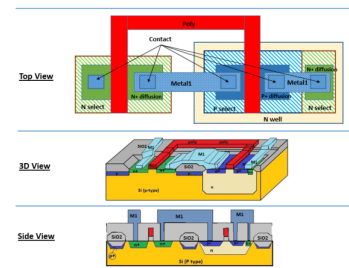
- **Full-custom ASIC**

- All aspects (e.g. size of a transistor) of a circuit are tailored for a particular application.
- Circuit fully optimized
- Design extremely complex and involved.
- Only feasible for small components
- Masks needed for all layers



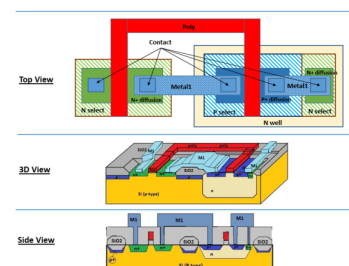
- **Standard-cell ASIC**

- Circuit made of a set of pre-defined logic, known as standard cells.
- E.g. basic logic gates, 1-bit adder, DFF etc.
- Layout of a cell is pre-determined, but layout of the complete circuit is customized
- Masks needed for all layers.



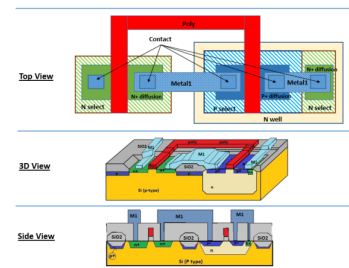
- **Gate array ASIC**

- Circuit is built from an array of a single type of cell (known as base cell)
- Base cell are pre-arranged and placed in fixed positions, aligned as one- or two- dimensional array
- More sophisticated components (macro cells) can be constructed from base cells
- Masks needed only for metal layers (connective wires)



- **Field programmable gate array**

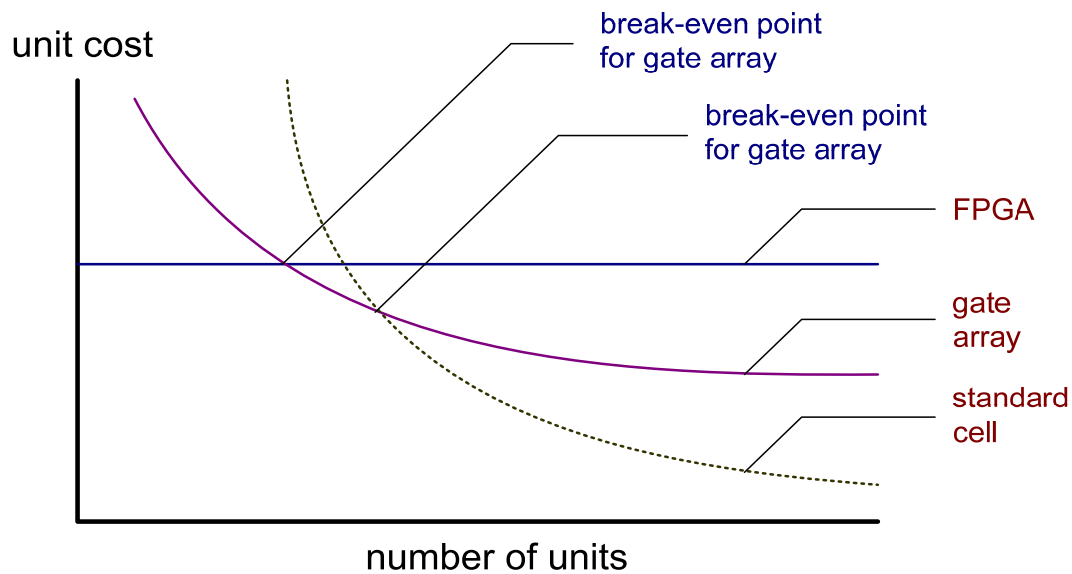
- Device consists of an array of generic logic cells and general interconnect structure
- Logic cells and interconnect can be “programmed” by utilizing semiconductor fuses or “switches”
- Customization is done in the “field”
- No custom mask needed



## Comparison of device technologies

|                         | Custom<br>(Full Custom) | Cell-based<br>(Standard Cell) | Pre-diffused<br>(Gate Array) | Prewired<br>(CPLD, FPGA) |
|-------------------------|-------------------------|-------------------------------|------------------------------|--------------------------|
| Density                 | Very High               | High                          | High                         | Medium - Low             |
| Performance             | Very High               | High                          | High                         | Medium - Low             |
| Flexibility             | Very High               | High                          | Medium                       | Low                      |
| Design Time             | Very Long               | Short                         | Short                        | Very Short               |
| Manufacturing time      | Medium                  | Medium                        | Short                        | Very Short               |
| Cost – Low /High Volume | Very High/Low           | High/Low                      | High/Low                     | Low/High                 |

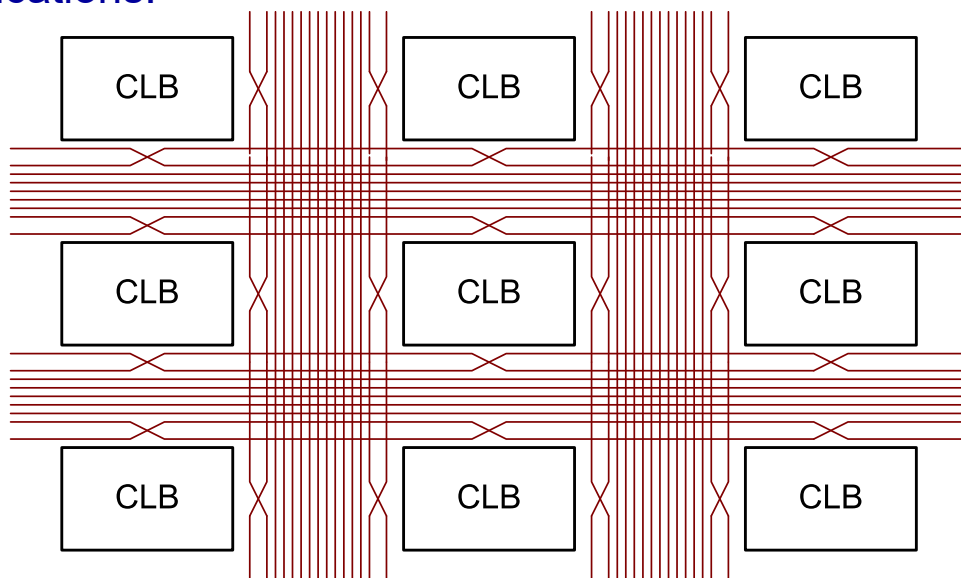
**Cost:**  $C_{\text{unit}} = C_{\text{per\_part}} + C_{\text{one\_time}} / \text{units\_produced}$



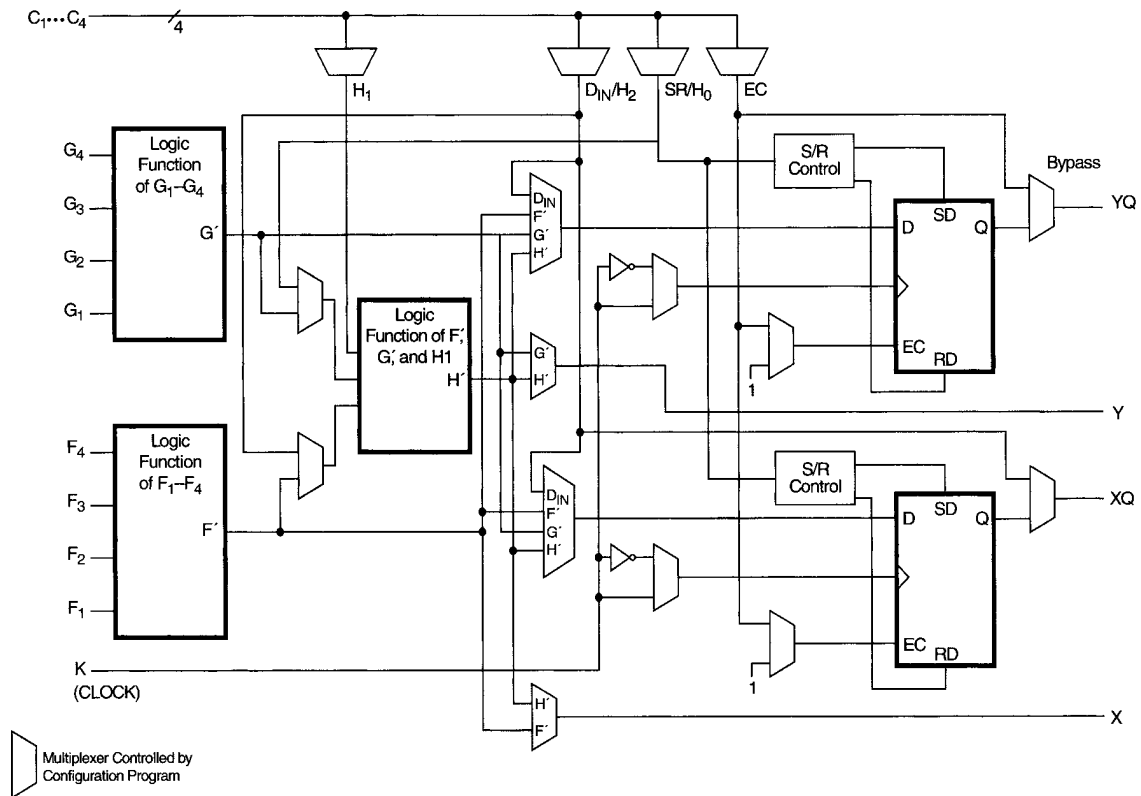
### Comparison of per unit cost

## Field programmable gate array (FPGA)

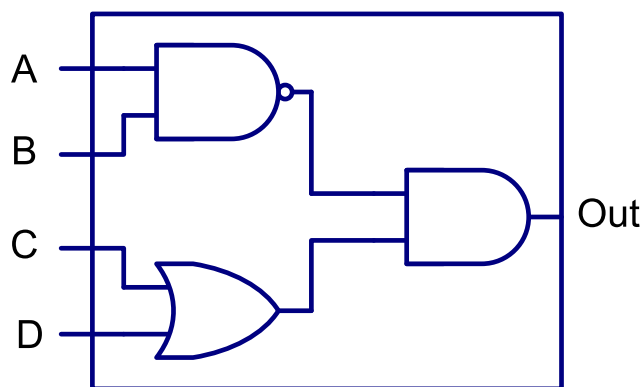
- Ideally, the interconnection between the millions of transistors can be reconfigured to realize different applications.



## CLB - Configurable logic block



29



Implementation of combinational functions using LUTs

| A | B | C | D | Out |
|---|---|---|---|-----|
| 0 | 0 | 0 | 0 | 0   |
| 0 | 0 | 0 | 1 | 1   |
| 0 | 0 | 1 | 0 | 1   |
| 0 | 0 | 1 | 1 | 1   |
| 0 | 1 | 0 | 0 | 0   |
| 0 | 1 | 0 | 1 | 1   |
| 0 | 1 | 1 | 0 | 1   |
| 0 | 1 | 1 | 1 | 1   |
| 1 | 0 | 0 | 0 | 0   |
| 1 | 0 | 0 | 1 | 1   |
| 1 | 0 | 1 | 0 | 1   |
| 1 | 0 | 1 | 1 | 1   |
| 1 | 1 | 0 | 0 | 0   |
| 1 | 1 | 0 | 1 | 0   |
| 1 | 1 | 1 | 0 | 0   |
| 1 | 1 | 1 | 1 | 0   |

## 1.2 What is VHDL?

- **VHSIC Hardware Description Language (VHDL)**
  - **V**ery **H**igh-**S**peed Integrated **C**ircuit program (VHSIC)
  - A computer language for **documenting** and **simulating** circuits, and describing circuits for **synthesis**.
  - A high level programming language with specialized constructs for **modeling hardware**.

## History of VHDL

- Intermetrics, TI and IBM under US DoD contract 1983-1985: VHDL 7.2
- IEEE standardization: VHDL 1076-1987
- First synthesized chip, IBM 1988
- IEEE Restandardization: VHDL 1076-1993
- Minor change in standard 2000 and 2002
- VHDL standard IEEE 1076-2008 published in Jan 2009

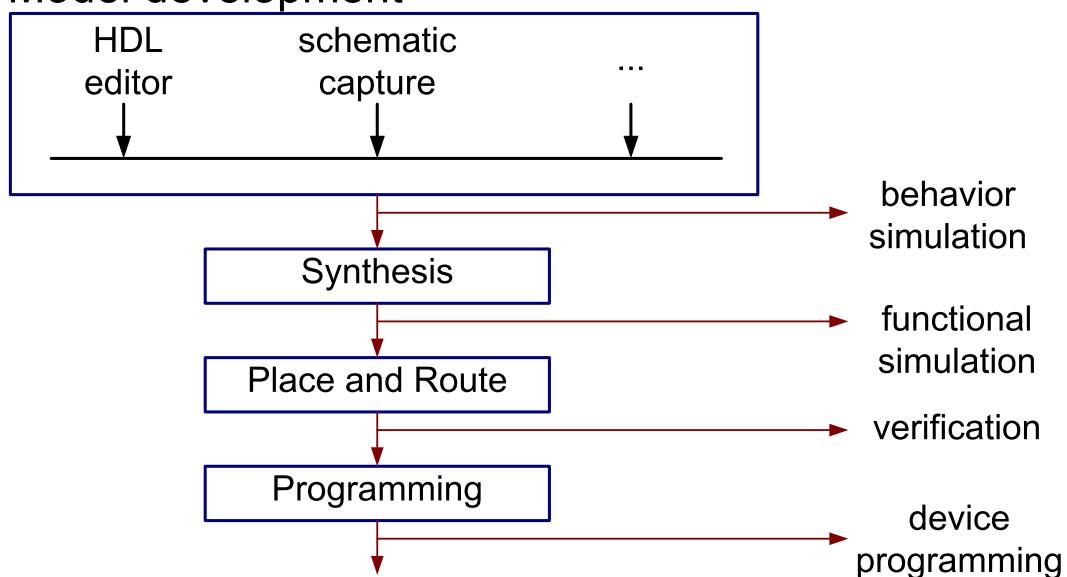


# The role of HDL

- Formal documentation
- Input to a simulator
- Input to a synthesizer

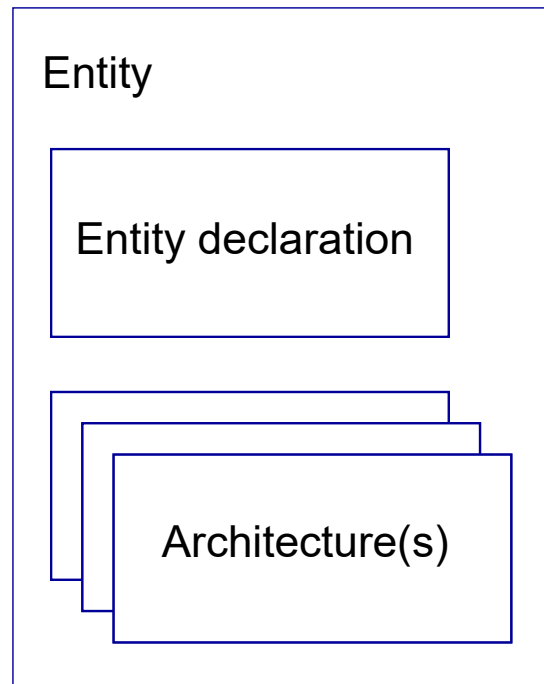
## FPGA design flow

### Model development

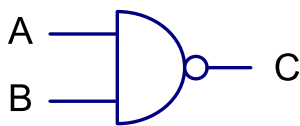


## 1.3: Basic VHDL Concept

- **Entity**
- **Architecture**
- Configuration
- Package
- Library

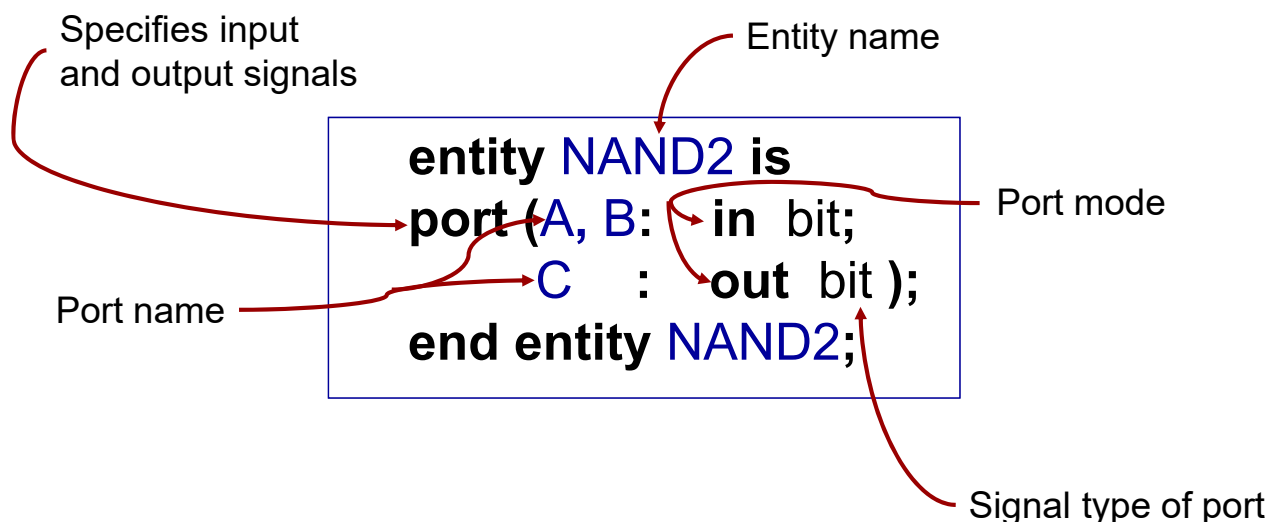


## NAND2 Gate



### Entity declaration

It provides an “external view” of a component.



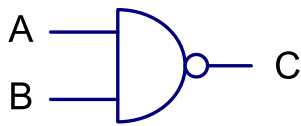
# Entity Declarations

- The entity declaration defines an interface to a component
  - It names the entity, and
  - It describes the input and output ports that can be seen from the outside,
    - Mode of signals (i.e. **in** and **out**)
    - Type of signals (i.e. bit)

## Possible modes for signals of entity ports

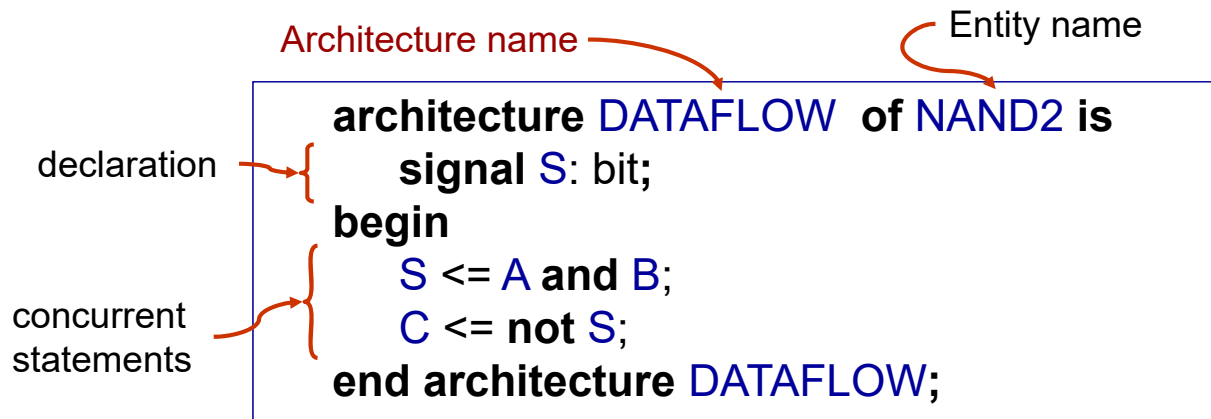
| Mode          | Purpose   |
|---------------|---|
| <b>in</b>     | Used for a signal that is an input to an entity.  |
| <b>out</b>    | Used for a signal that is an output from an entity. The value of the signal can not be used inside the entity. This means that in an assignment statement, the signal can appear only to the left of the <= operator.               |
| <b>inout</b>  | Used for a signal that is both an input to an entity and an output from the entity.   |
| <b>buffer</b> | Used for a signal that is an output from an entity. The value of the signal can be used inside the entity, which means that in an assignment statement, the signal can appear both on the left and right sides of the <= operators. |

# NAND2 Gate



## Architecture

It provides an “internal view” of a component.



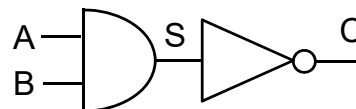
## Statements in the architecture are concurrent statements

`S <= A and B;`  
`C <= not S;`

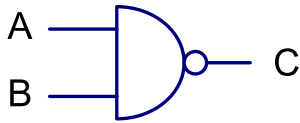
and

`C <= not S;`  
`S <= A and B;`

will produce the same result



## NAND2 Gate



Architecture:  
Behavior style

```
architecture BEHAVIOR of NAND2 is
begin
  process (A,B) is
  begin
    if (A='1' and B='1') then
      C <= '0';
    else
      if (A='0' and B='0') or (A='0' and B='1')
        or (A='1' and B='0') then
        C <= '1';
      end if;
    end if;
  end process;
end architecture BEHAVIOR;
```

## Architecture

- It defines the relationships between the inputs and outputs of a design entity.
  - It consists of a declaration section followed by a collection of *concurrent statements*.
  - It may be expressed in terms of behavior, data flow, or structure.

# Chapter 2. Data Objects and Operators

- Basic Language Elements
  - Data Objects
  - Data Type
  - Operators
  - Identifier

## 2.1 Data objects

- An object in VHDL is a named item that holds the value of a specific data type.
- Four kinds of data objects
  - Signals
  - Constants
  - Variables
  - File
- For describing logic circuits, the most important data objects are signals. They represent the logic signals (wires) in the circuit.

## 2.1.1 Signal data objects

- A signal is an object that holds the current and possible future values of the object.
- They occur as inputs and outputs in port descriptions, as signals in architecture, etc.

### Where can signal data objects be declared?

- Entity declaration
- Declarative section of an architecture
- Cannot be in a process

```
entity nand2 is
port (A, B: in bit;
      C : out bit);
end entity nand2;

architecture dataflow of
nand2 is
  signal S: bit;
begin
    S <= A and B;
    C <= not S;
end architecture dataflow;
```

### How to declare a signal?

**signal** *signal\_name* : *signal\_type* [:= *initial\_value*];

### Examples:

**signal** *status* : std\_logic := '0';

**signal** *data* : std\_logic\_vector (31 **downto** 0);

## Signal assignment

A signal assignment schedules a new value to occur at some future time. **The current value of the signal is never changed.** If no specific value of time is specified the default value is infinitesimally small value of time into the future called delta time.

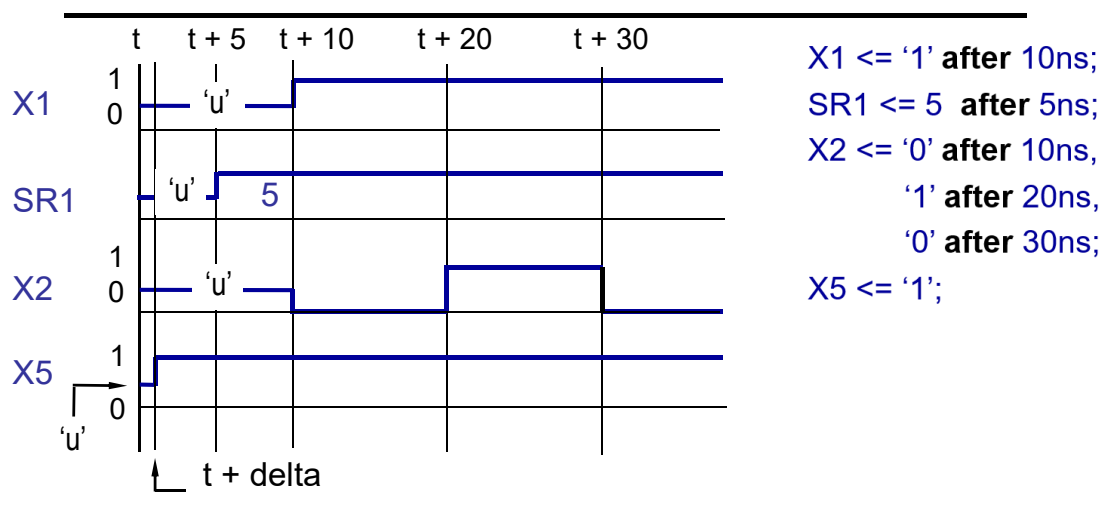
**Signals are assigned using the “<=“ operator. e.g.**

**X1 <= '1' after 10ns;**

**SR1 <= 5 after 5ns;**

**X2 <= '0' after 10ns, '1' after 20ns, '0' after 30ns;**

**X5 <= '1';**



The figure above shows the timing implied by the statements shown in the previous slides assuming that all statements executed at time  $t$ .



## 2.1.2 Variable data objects

- Variables are used to hold temporary data.

### Where to declare a variable?

- within the processes, functions and procedures in which they are used

### How to declare a variable?

**variable** `variable_name` : `variable_type` [`:= initial_value`];

### Examples:

**variable** `address` : `bit_vector` (15 **downto** 0) `:=` x"0000";

**variable** `index`: `integer range` 0 to 10 `:=` 0;

```
entity nand2 is
port (A, B: in bit;
      C : out bit);
end entity nand2;
```

```
architecture dataflow of
nand2 is
  signal S: bit;
begin
  S <= A and B;
  C <= not S;
end architecture dataflow;
```

## Variable assignment

In contrast to signal assignment, a variable assignment takes effect immediately.

Variables are assigned using the `:=` operator. e.g.

`A := '1';`

`ROM_A(5) := ROM_A(0);`

`STAR_COLOR := GREEN;`

## 2.1.3 Constant data objects

- A constant is an object which is initialized to a specific value when it is created, and which cannot be subsequently modified.

### Where can constants be declared?

- Declarative section of an architecture
- Declarative section of a process

### How to declare a constant?

**constant** `constant_name` : `constant_type` [`:= initial_value`];

### Examples:

**constant** `yes` : `boolean` := TRUE;  
**constant** `msb` : `integer` := 5;

```
entity nand2 is
port (A, B: in bit;
      C : out bit);
end entity nand2;
```

```
architecture dataflow of
nand2 is
  signal S: bit;
begin
  S <= A and B;
  C <= not S;
end architecture dataflow;
```

## 2.2 Data types

- The type of a signal, variable, or constant object specifies:
  - the range of values it may take
  - the set of operations that can be performed on it.

The VHDL language supports a predefined standard set of type definitions as well as enables the definition of new types by users.

## 8 types commonly used:

- bit
- bit\_vector
- integer
- boolean
- array
- enumeration
- std\_logic
- std\_logic\_vector

## Bit and Bit\_vector

Bit type has two values, '0' and '1'.

### Example:

```
signal a : bit := '0';  
variable b : bit ;
```

Bit\_vector is an array where each element is of type bit.

### Example:

```
signal c : bit_vector (3 downto 0) := "1000"; -- recommended  
signal d : bit_vector (0 to 3) := "1000";
```

## INTEGER type

INTEGER type represents positive, negative numbers and 0.

By default, an INTEGER signal has 32 bits and can represent numbers from  $-2^{31}$  to  $2^{31}-1$ . The code does not specifically give the number of bits in the signal.

Integers with fewer bits than 32 can be declared, using the RANGE keyword.

### Example:

```
signal x : integer range -128 to 127;
```

This defines x as an eight-bit signed number.

## BOOLEAN type

An object of type BOOLEAN can have the values TRUE or FALSE, where TRUE is equivalent to 1 and FALSE to 0.

### Example:

```
signal flag : boolean;  
constant correct : boolean := TRUE;
```

## ENUMERATION type

An ENUMERATION type is defined by listing all possible values of that type. All of the values of an enumeration type are user-defined.

```
type enumerated_type_name is (name {, name});
```

The most common example of using the ENUMERATION type is for specifying the states for a finite-state machine.

### Example:

```
type State_type is (stateA, stateB, stateC);  
signal y : State_type := stateB ;
```

When the code is translated by the VHDL compiler, it automatically assigns bit patterns (codes) to represent stateA, stateB and stateC.

# ARRAY type

ARRAY types group one or more elements of the same type together as a single object.

**type** array\_type\_name **is** array (index\_range) **of** element\_type;

**Example:**

```
type byte is array (7 downto 0) of bit;
type word is array (15 downto 0) of bit;
type memory is array ( 0 to 4095 ) of word;

signal program_counter: word := "01010101010101";
variable data_memory: memory;
```

To refer individual elements of array:

program\_counter(5 **downto** 0) accesses the 6 LSBs of program\_counter.  
data\_memory(0) accesses the first record in memory.

## std\_logic and std\_logic\_vector

std\_logic provides more flexibility than the **bit**.

To use, you must include the two statements:

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

std\_logic\_vector type represents an array of std\_logic objects.

```
type std_logic is
(
    'U',  -- uninitialized
    'X',  -- unknown
    '0',  -- forcing 0
    '1',  -- forcing 1
    'Z',  -- high impedance
    'W',  -- weak unknown
    'L',  -- weak 0
    'H',  -- weak 1
    '-',  -- don't care
);
```

## Example

```
signal x1,x2,Cin,Cout,Sel : std_logic;  
signal C                  : std_logic_vector (1 to 4);  
signal X,Y,S              : std_logic_vector (3 downto 0);
```

`std_logic` objects are often used in logic expressions.

`std_logic_vector` objects can be used as binary numbers in arithmetic circuits by including in the code the following statement

```
    use ieee.std_logic_signed.all;  
or  
    use ieee.std_logic_unsigned.all;
```

## VHDL is strongly typed

- VHDL is a strongly type-checked language. Even for objects that intuitively seem compatible, like **bit** and *std\_logic*, one cannot be assigned to another.

Recommendation:

- use *std\_logic* and *std\_logic\_vector* types

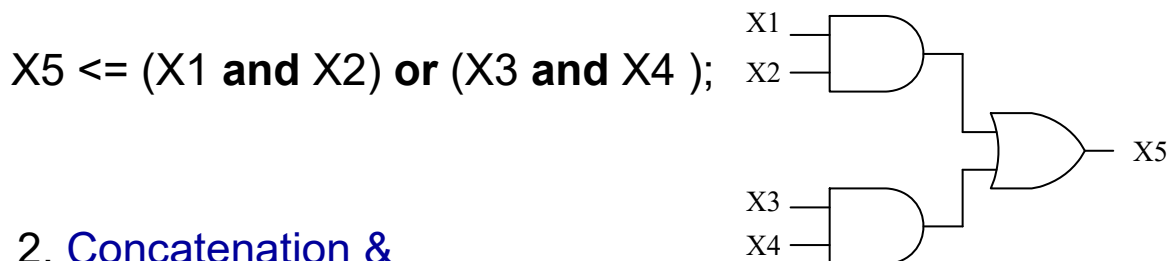
## 2.3 Operators

|                    | Operator Class | Operator                      |
|--------------------|----------------|-------------------------------|
| Lowest precedence  | Logical        | and, or, nand, nor, xor, xnor |
|                    | Relational     | =, /=, <, <=, >, >=           |
|                    | shift          | sll, srl, sla, sra, rol, ror  |
|                    | Adding         | +, −, &                       |
|                    | Sign           | +, −                          |
|                    | Multiplying    | *, /, mod, rem                |
| Highest precedence | Miscellaneous  | **, abs, not                  |

### 1. Logic operators

E.g. sig <= “11001” and “10011”;  
sig gets value “10001”

Operators on the same line have equal precedence and must be **parenthesized** when necessary.



### 2. Concatenation &

E.g. sig2 <= “001” & sig (3 **downto** 1);  
sig2 gets value “001000”



| Operator   | Description  | Data type of a           | Data type of b | Data type of result |
|--|--|--------------------------|----------------|---------------------|
| <b>a ** b</b>  | exponentiation   | integer                  |                |                     |
| <b>abs a</b>   | absolute value   | integer                  |                |                     |
| <b>not a</b>   | negation   | boolean, bit, bit_vector |                |                     |
| <b>a * b, a / b, a mod b, a rem b</b>                          | multiplication, division, modulo, remainder                                    | integer                  |                |                     |
| <b>+a, -a</b>  | identity, negation   | integer                  |                | integer             |
| <b>a + b, a - b<br/>a &amp; b</b>                              | addition, subtraction, concatenation   | integer                  |                |                     |
|  |  | 1D array, element        |                |                     |
| <b>a sll b, a srl b, a sla b, a sra b, a rol b, a ror b</b>    | shift-left (right) logical, shift-left (right) arithmetic, rotate left (right) | bit_vector               | integer        | bit_vector          |
| <b>a = b, a /= b, a &lt; b, a &lt;= b, a &gt; b, a &gt;= b</b> |  | any                      | same as a      | boolean             |
|  |  | scalar or 1D array       | same as a      | boolean             |
| <b>a and b, a or b, a xor b, a nand b, a nor b, a xnor b</b>   |  | boolean, bit, bit_vector | same as a      | same as a           |

EE332 Digital System Design, by Yu Yajun -- 2019

65

**Table: Overloaded operators in the IEEE std\_logic\_1164 package**

| Overloaded operator  | Data type of a                | Data type of b | Data type of result |
|--|-------------------------------|----------------|---------------------|
| <b>not a</b>   | std_logic_vector<br>std_logic |                | same as a           |
| <b>a and b, a or b, a xor b, a nand b, a nor b, a xnor b</b> | std_logic_vector<br>std_logic | same as a      | same as a           |

**Table: Functions in the IEEE std\_logic\_1164 package**

| Function             | Data type of a   | Data type of result |
|----------------------|------------------|---------------------|
| to_bit(a)            | std_logic        | bit                 |
| to_stdulogic(a)      | bit              | std_logic           |
| to_bitvector(a)      | std_logic_vector | bit_vector          |
| to_stdlogicvector(a) | bit_vector       | std_logic_vector    |

EE332 Digital System Design, by Yu Yajun -- 2019

66

## Use of conversion functions

**signal** *s1*, *s2*, *s3*: std\_logic\_vector(7 **downto** 0);

**Signal** *b1*, *b2*: bit\_vector(7 **downto** 0);

The following statements are **wrong** because of data type mismatch

```
s1 <= b1; -- bit_vector assigned to std_logic_vector  
b2 <= s1 and s2; -- std_logic_vector assigned to bit_vector  
s3 <= b1 or s2; -- or is undefined between bit_vector and  
--std_logic_vector
```

We can use the conversion function to correct these problems

```
s1 <= to_stdlogicvector(b1);  
b2 <= to_bitvector(s1 and s2);  
s3 <= to_stdlogicvector(b1) or s2; -- or  
s3 <= to_stdlogicvector(b1 or to_bitvector(s2));
```

## 2.4 Lexical elements

- The lexical elements are the basic syntactical units in a VHDL program.
  - comments,
  - identifiers,
  - reserved words,
  - number, characters and strings.

## 2.4.1 Comments

- A comment starts with two dashes, --, followed by the comment text.
- The comments are for documentation purpose only.

```
-- *****
```

```
-- example of entity
```

```
-- *****
```

**entity** *example* **is**

```
-- .....
```

**end entity** *example*;

## 2.4.2 Identifier

- Identifiers are used as names for signals, variables, constants, as well as entities, architectures and so on.
- A basic identifier is a sequence of characters that may be
  - upper or lower case letters and digits 0 - 9
  - underscore (“\_”) character
- VHDL language is **NOT** case sensitive.
- The first character must be a letter and the last character must **NOT** be “\_”
- Two successive underscores “\_\_” are **NOT** allowed.
  - Select, ALU\_in, Mem\_data, Two\_dash\_ok ✓
  - 12Select, \_start, out\_, Not\_\_Allow, dot#3 ✗

## 2.4.3 Preserved words

- Some words are reserved in VHDL to form the basic language constructs.

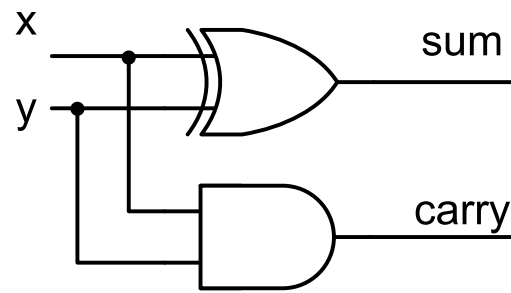
abs, access, after, alias, all, and, architecture, array, assert, attribute, begin, block, body, buffer, bus, case, component, configuration, constant, disconnect, downto, else, elsif, end, entity, exit, file, for, function, generate, generic, guarded, if, impure, in, inertial, inout, is, label, library, 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, reject, rem, report, return, rol, ror, select, severity, shared, signal, sla, sll, sra, srl, subtype, then, to, transport, type, unaffected units, until, use, variable, wait, when, while, with, xnor, xor

## 2.4.4 Numbers, characters and strings

- A Number in VHDL can be
  - integer, such as 0, 1234, and 98E7
  - real, such as 0., 1.23456, or 9.87E6
  - represented in other number bases:
    - $45 = 2\#101101\# = 16\#2D\#$
- A character in VHDL is enclosed in single quotation marks,
  - such as 'A', 'Z', '3'.
  - 1 and '1' are different.
- A string in VHDL is a sequence of characters enclosed in double quotation marks:
  - such as "Hello", "10000111".
  - $2\#10110010\#$  and "10110010" are different

## Chapter 3. Concurrent Statement

- The operation of digital system is inherently concurrent.



- Within VHDL signals are assigned values using **signal assignment statements**.

`sum <= (x xor y) after 5 ns`

- Multiple signal assignment statements are executed concurrently in simulated time and are referred to as **concurrent signal assignment statements (CSAs)**.
- There are several forms of CSA statements:
  - Simple CSA
  - Conditional signal assignment
  - Selected Signal assignment

## 3.1 Simple CSA

**architecture concurrent\_behavior of half\_adder is**  
**begin**

    sum <= ( x xor y) **after** 5 ns;

    carry <= (x and y) **after** 5 ns;

**end architecture concurrent\_behavior;**

- General form:

target <= expression;

expression is logical, comparative or arithmetic operations.

- The execution of the statements is determined by the flow of signal, rather than textual order.

## Half\_adder

**library** IEEE;

**use** IEEE.std\_logic\_1164.all;

**entity** half\_adder **is**

**port** (x, y : **in** std\_logic;

        sum, carry : **out** std\_logic);

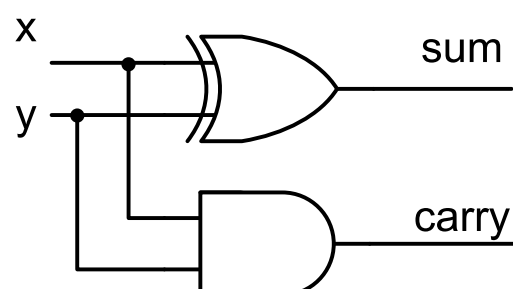
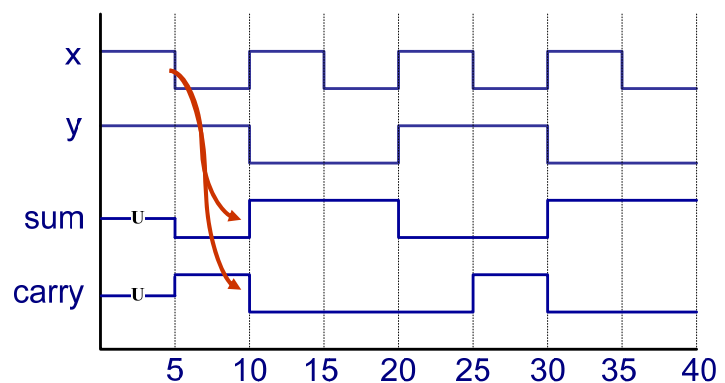
**end entity** half\_adder;

**architecture concurrent\_behavior of half\_adder is**  
**begin**

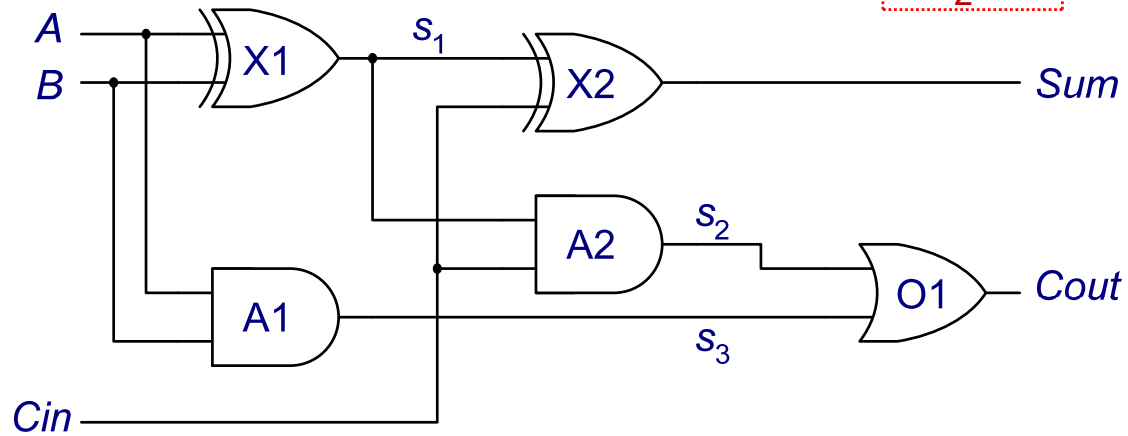
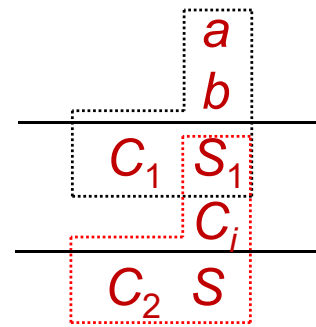
    sum <= ( x xor y) **after** 5 ns;

    carry <= (x and y) **after** 5 ns;

**end architecture concurrent\_behavior;**

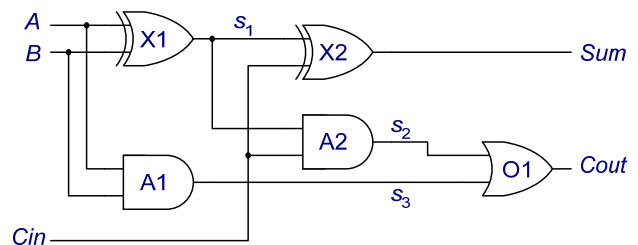


# full\_adder



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

entity full_adder is
port (A, B, Cin : in std_logic;
      Sum, Cout: out std_logic);
end entity full_adder;
```



type time

```
architecture dataflow of full_adder is
  signal s1, s2, s3 : std_logic;
  constant gate_delay : time := 10 ns;
begin
  L1: s1 <= (A xor B) after gate_delay;
  L2: s2 <= (Cin and s1) after gate_delay;
  L3: s3 <= (A and B) after gate_delay;
  L4: sum <= (s1 xor Cin) after gate_delay;
  L5: cout <= (s2 or s3) after gate_delay;
end architecture dataflow;
```

} Architecture declarative segment

} Architecture body

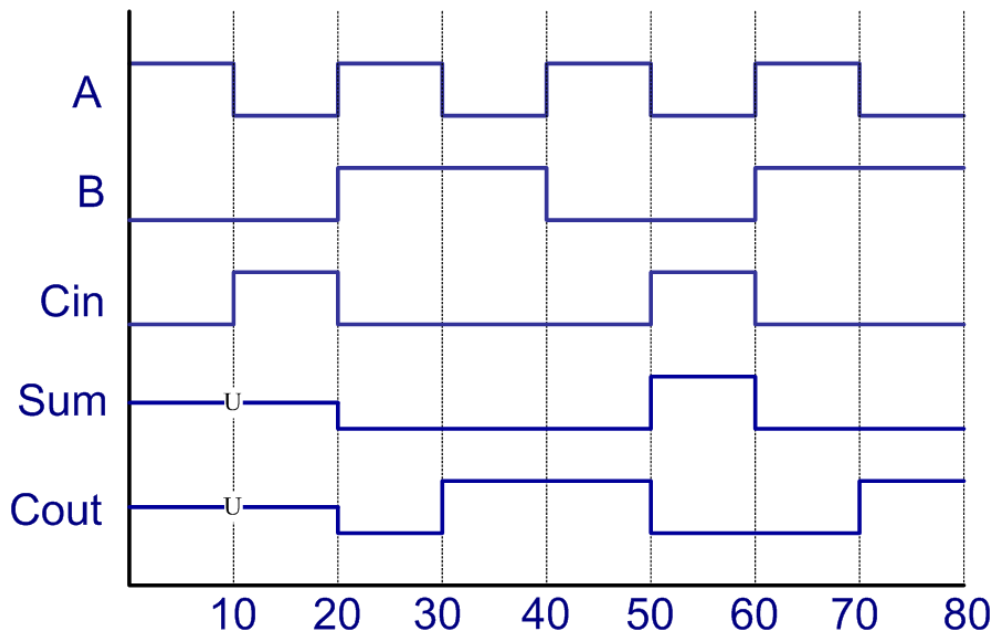
constant object



```

L1: s1 <= (A xor B) after gate_delay;
L2: s2 <= (Cin and s1) after gate_delay;
L3: s3 <= (A and B) after gate_delay;
L4: sum <= (s1 xor Cin) after gate_delay;
L5: cout <= (s2 or s3) after gate_delay;

```

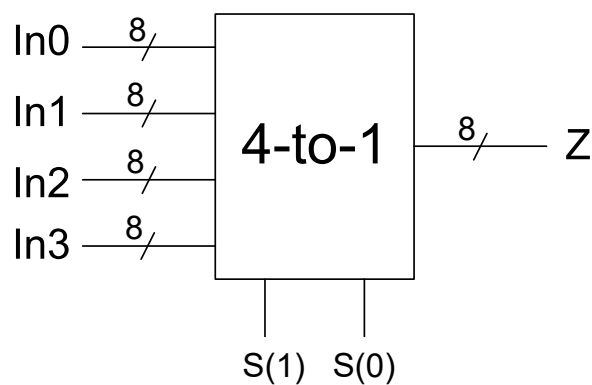


## 3.2 Conditional signal assignment

- Simple CSAs is convenient for describing gate-level circuits whose behavior can be expressed with Boolean equations.
- It is useful to model circuits at higher levels of abstraction such as multiplexors, decoders, priority encoders, etc.

### 4-to-1, 8-bit multiplexor

| Input S | Output Z |
|---------|----------|
| 00      | In0      |
| 01      | In1      |
| 10      | In2      |
| 11      | In3      |





```

library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
port (In0, In1, In2, In3 : in std_logic_vector (7 downto 0);
      S : in std_logic_vector(1 downto 0);
      Z : out std_logic_vector (7 downto 0) );
end entity mux4;

```

```

architecture con_arch1 of mux4 is
begin
  Z <= In0 when S = "00" else
    In1 when S = "01" else
    In2 when S = "10" else
    In3;
end architecture con_arch1;

```

```

architecture con_arch2 of mux4 is
begin
  Z <= In0 when S = "00" else
    In1 when S = "01" else
    In2 when S = "10" else
    In3 when S = "11" else
    "XXXXXXXX";
end architecture con_arch2;

```

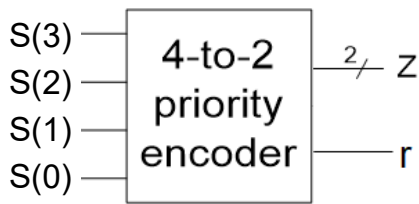
- The conditional signal assignment itself is a concurrent signal assignment.
- It has one target, but can have more than one expression.  
General form:  

```

target <= {expression} when {condition} else
          {expression} when {condition} else
          .....
          {expression};

```
- When this CSA is executed, the expressions in the right hand side are evaluated in the order that they appear.
  - The order of the expressions with their respective conditions inside the statement is important.
- Even there are several lines of text, this corresponds to only one signal assignment statement.

## 4-to-2 priority encoder



Assume that S(3) has the highest priority

| Input S | Output Z, r |
|---------|-------------|
| 1 ---   | 11, 1       |
| 0 1 --  | 10, 1       |
| 0 0 1 - | 01, 1       |
| 0 0 0 1 | 00, 1       |
| 0 0 0 0 | 00, 0       |

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

entity pr_encoder is
port (S: in std_logic_vector(3 downto 0);
      Z : out std_logic_vector (1 downto 0);
      r : out std_logic);
end entity pr_encoder;
```

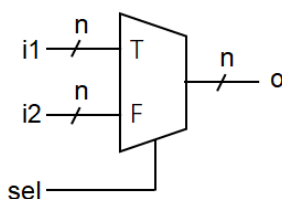
```
architecture con_behavioral of pr_encoder is
begin
  Z <= "11" when S(3) = '1' else
        "10" when S(2) = '1' else
        "01" when S(1) = '1' else
        "00" when S(0) = '1' else
        "00";
  r <= S(3) or S(2) or S(1) or S(0);
end architecture con_behavioral;
```

## Conceptual Implementation

```
signal_name <= value_expr_1 when Boolean_expr_1 else
               value_expr_2 when Boolean_expr_2 else
               .....
               value_expr_n;
```

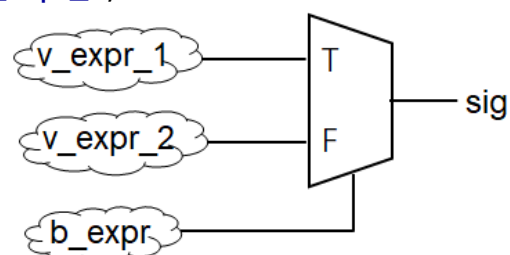
Constructing the CSA statement requires 3 groups of circuits.

- ✓ Value expression circuits
- ✓ Boolean expression circuits
- ✓ Priority routing network
  - can be implemented by a sequence of 2-to-1 multiplexer
  - An abstract multiplexer

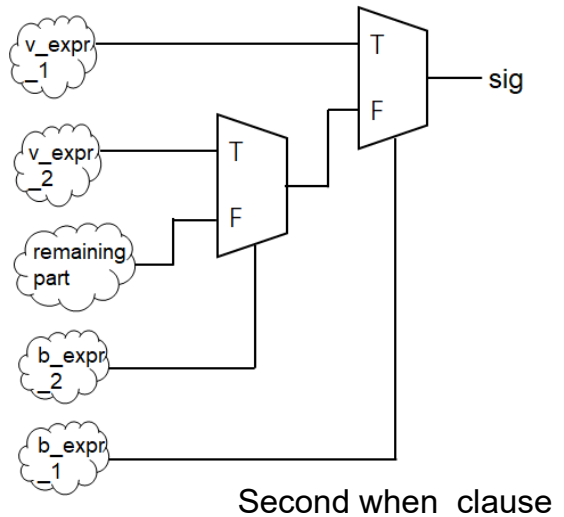
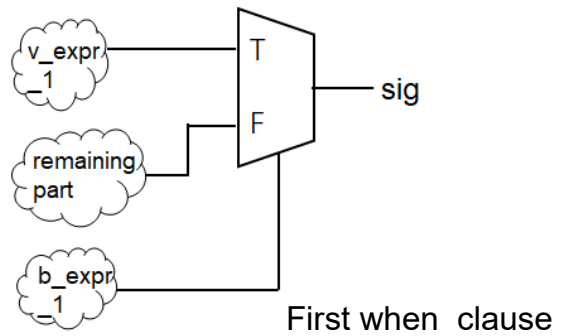
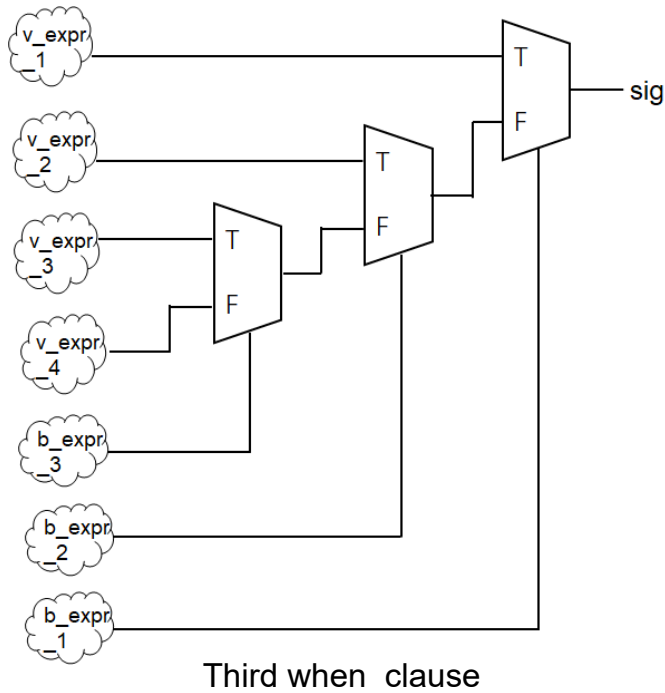


A simple CSA statement

```
sig <= v_expr_1 when b_expr else
      v_expr_2;
```

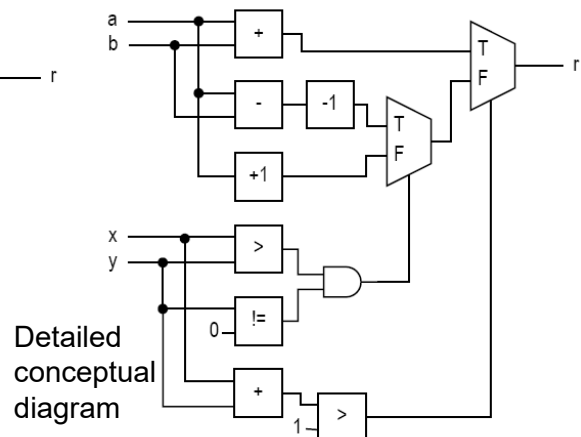
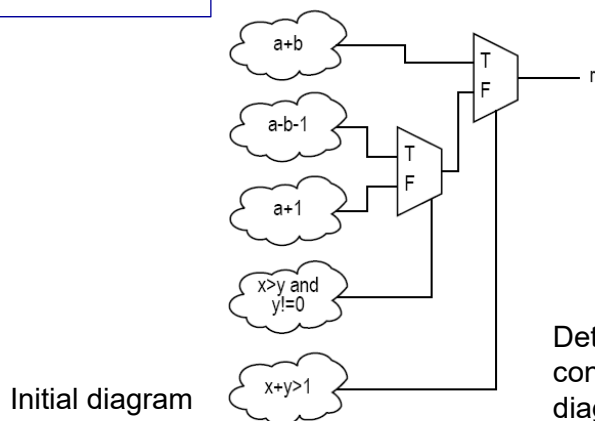
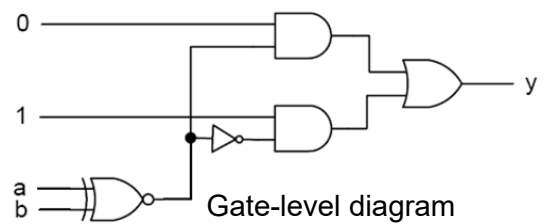
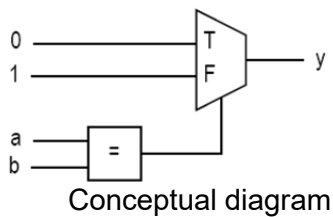


```
sig <= v_expr_1 when b_expr_1 else
v_expr_2 when b_expr_2 else
v_expr_3 when b_expr_3 else
v_expr_4;
```



## Detailed Implementation Examples

```
...
Signal a,b,y : std_logic;
...
y <= '0' when a=b else
    '1';
...
```

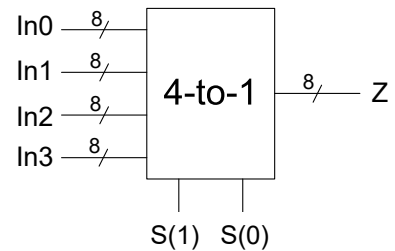


```
...
signal a,b,r : unsigned(7 downto 0);
signal x,y : unsigned(3 downto 0);
r <= a + b when x + y > 1 else
    a - b - 1 when x > y and y != 0 else
    a+1;
```

## 3.3 Selected signal assignment

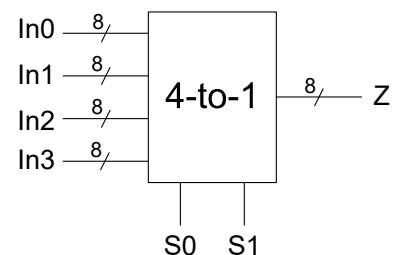
- It is similar to the conditional signal assignment statement.
- The value of a target signal is determined by the value of a select expression.
- Example: the same 4-1 multiplexer.

```
architecture sel_behavioral of mux4 is
begin
    with S select
        Z <= In0 when "00",
            In1 when "01",
            In2 when "10",
            In3 when others;
end architecture sel_behavioral;
```



```
with S select
    Z <= In0 when "00",
        In1 when "01",
        In2 when "10",
        In3 when "11";
```

Why?



```
with S select
    Z <= In0 when "00",
        In1 when "01",
        In2 when "10",
        In3 when "11",
        "XXXXXXXX" when others;
```

## 4-to-2 priority encoder based on a selected signal assignment statement

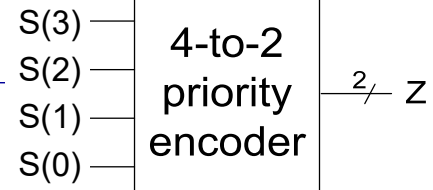
```
architecture sel_arch of pr_encoder is
begin
```

```
  with S select
```

```
    Z <= "11" when "1000" | "1001" | "1010" | "1011" |
              "1100" | "1101" | "1110" | "1111",
      "10" when "0100" | "0101" | "0110" | "0111",
      "01" when "0010" | "0011",
      "00" when others;
```



```
end architecture behavioral
```



Why?

```
architecture sel_arch of pr_encoder is
begin
```

```
  with S select
```

```
    Z <= "11" when "1--",
      "10" when "01--",
      "01" when "001-",
      "00" when others;
```



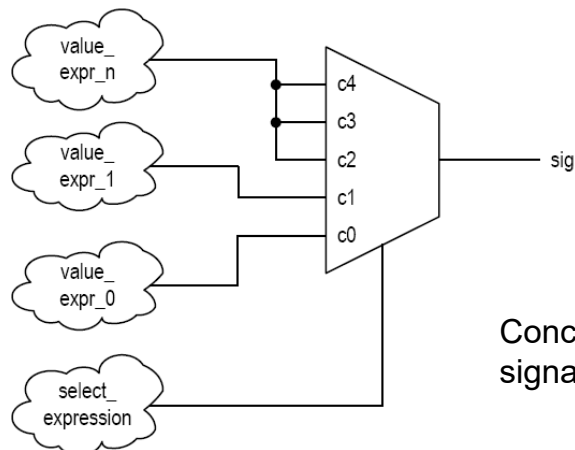
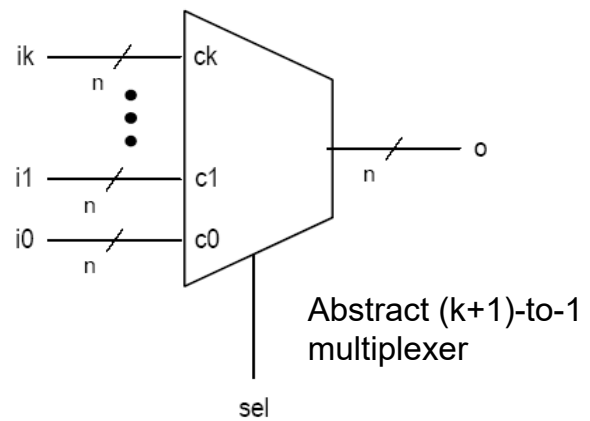
```
end architecture behavioral
```

- The choices for the select expression are not evaluated in sequence.
- All choices are evaluated, but one and only one must be true.
- All of the choices that the programmer specifies must cover all the possible values of the select expression.
- When an event occurs on a signal used in the select expression, or any of the signals used in one of the choices, the statement is executed.

## Conceptual Implementation

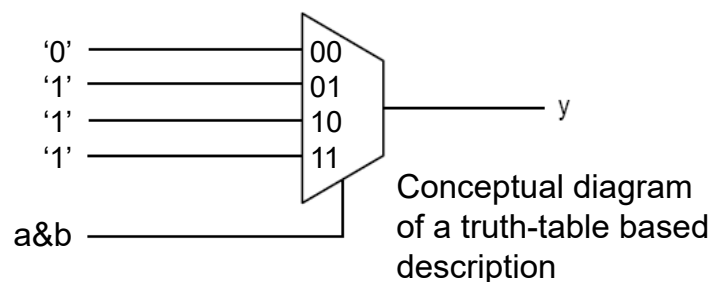
with **s\_expr** select

```
sig <= v_expr_1 when choice_1,
      v_expr_2 when choice_2,
      v_expr_3 when choice_3,
      ...
      v_expr_n when choice_n;
```

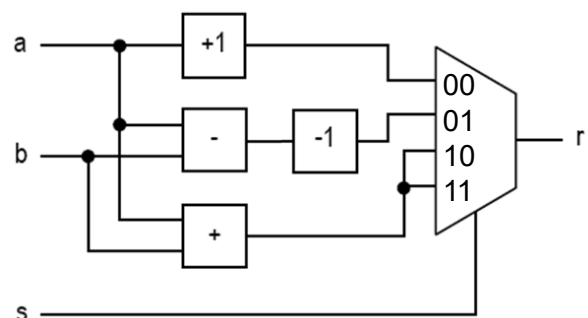


## Detailed Implementation Examples

```
tmp <= a and b;
with tmp select
  y <= '0' when "00",
      '1' when "01",
      '1' when "10",
      '1' when others;
```



```
...
signal a,b,r : unsigned(7 downto 0);
signal s : unsigned(1 downto 0);
...
with s select
  z <= a+1 when "00",
      a - b - 1 when "01",
      a + b when others;
```



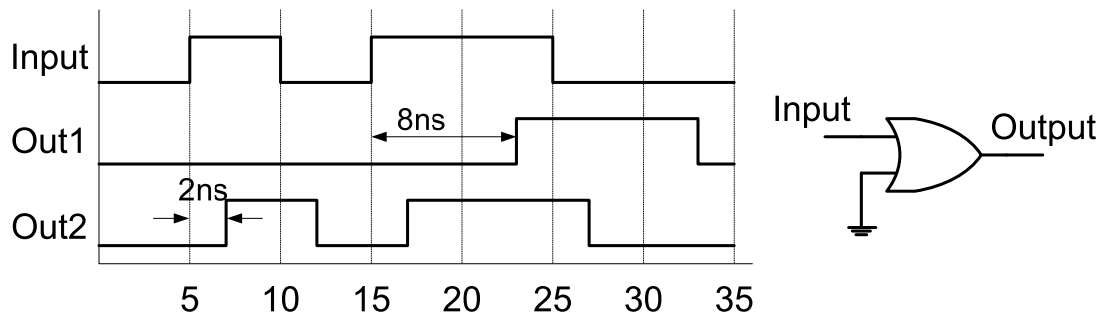
```
with s select
  x <= (a and b) when "11",
      (a or b) when "01" | "10",
      '0' when others;
```

## 3.4 Understanding delays

- Accurate representation of the behavior of digital circuits requires accurate modeling of delays through the various components.
- Three types of delay models
  - Inertial delay model
  - Transport delay model
  - Delta delay

### 3.4.1 The inertial delay model

- Digital circuits takes a finite amount of time for the output of a gate to respond to a change on the input.
- This implies that the change on the input has to persist for a certain period of time to ensure that the output will response.
- If it does not persist long enough the input events will not be propagated to the output.
- This propagation delay model is referred to as the inertial delay model.



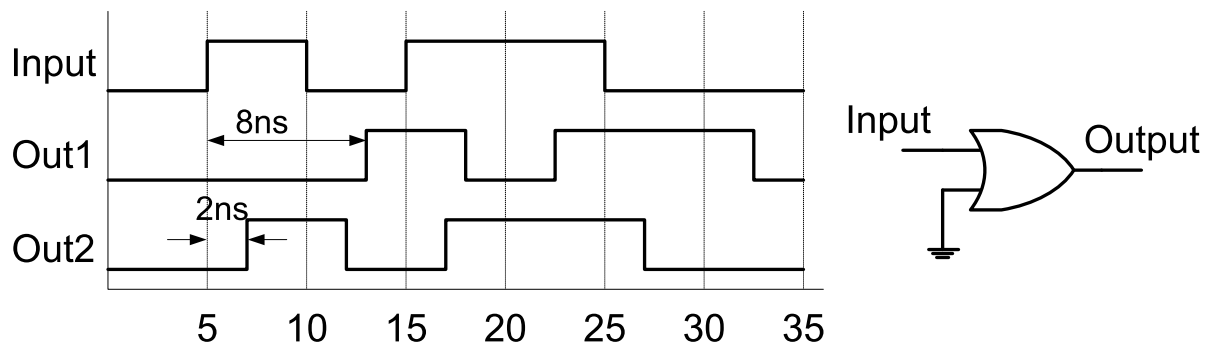
Out1 <= Input **or** '0' **after** 8 ns;  
 Out2 <= Input **or** '0' **after** 2 ns;

- Any pulse in the input with a width of less than the propagation delay through the gate is said to be rejected.
- often used for component delays
- Default in VHDL program

### 3.4.2 The transport delay model

- Transport delay models the delays in hardware that do not exhibit any inertial delay.
- This delay represents pure propagation delay; that is, any changes on an input are transported to the output, no matter how small of the width, after the specified delay.
- Keyword **transport** is used in a signal assignment statement for transport delay model.





```
Out1 <= transport (Input or '0') after 8 ns;
Out2 <= transport (Input or '0') after 2 ns;
```

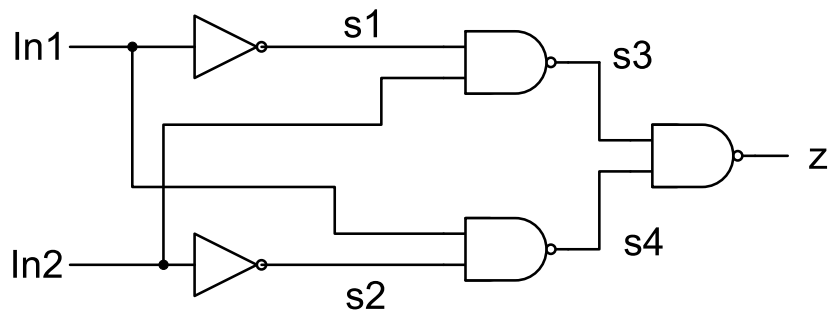
- Pulses are propagated, irrespective of width
- good for interconnect delays.

### 3.4.3 Delta Delays

- If we do not specify a delay for the occurrence of an event on a signal, for example  

```
sum <= (x xor y);
```

a **delta delay** is assumed by the simulator
- Delta delay is an infinitesimally small delay,  $\Delta$ .
- In a signal assignment, the value is not assigned to the signal directly but after a delta delay at the earliest.
- Delta delays are simply used to enforce dependencies between events and thereby ensure correct simulation.



```

library IEEE;
use IEEE.std_logic_1164.all;

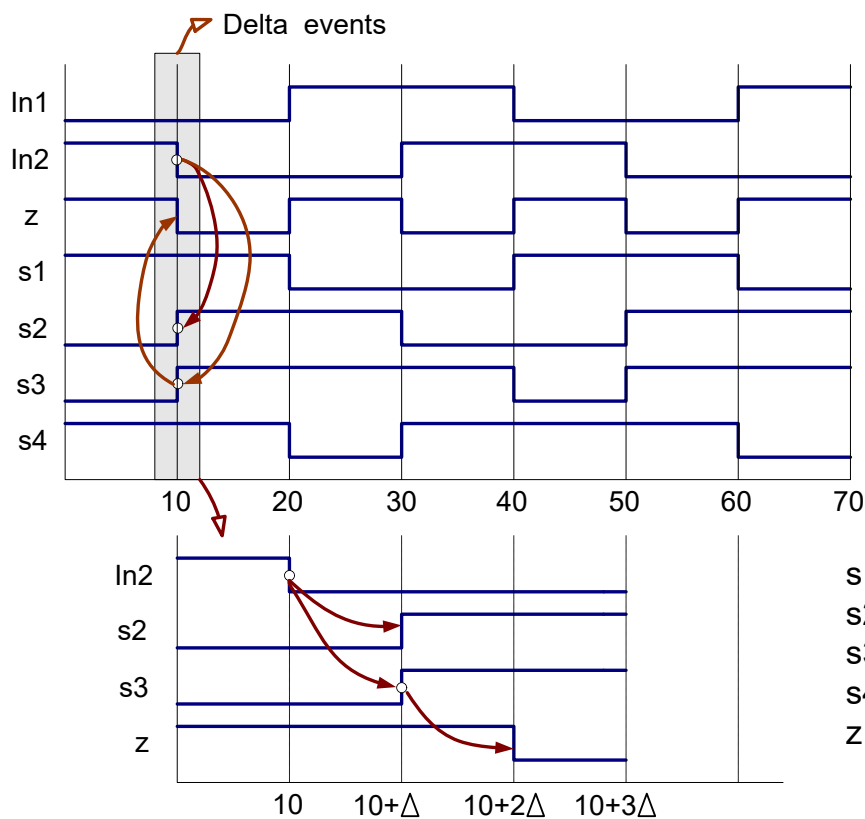
entity combinational is
port (In1, In2 : in std_logic;
      z : out std_logic);
end entity combinational;

```

```

architecture behavior of combinational is
  signal s1, s2, s3, s4 : std_logic := 0;
begin
  s1 <= not In1;
  s2 <= not In2;
  s3 <= not (s1 and In2);
  s4 <= not (s2 and In1);
  z <= not (s3 and s4);
end architecture behavior;

```



```

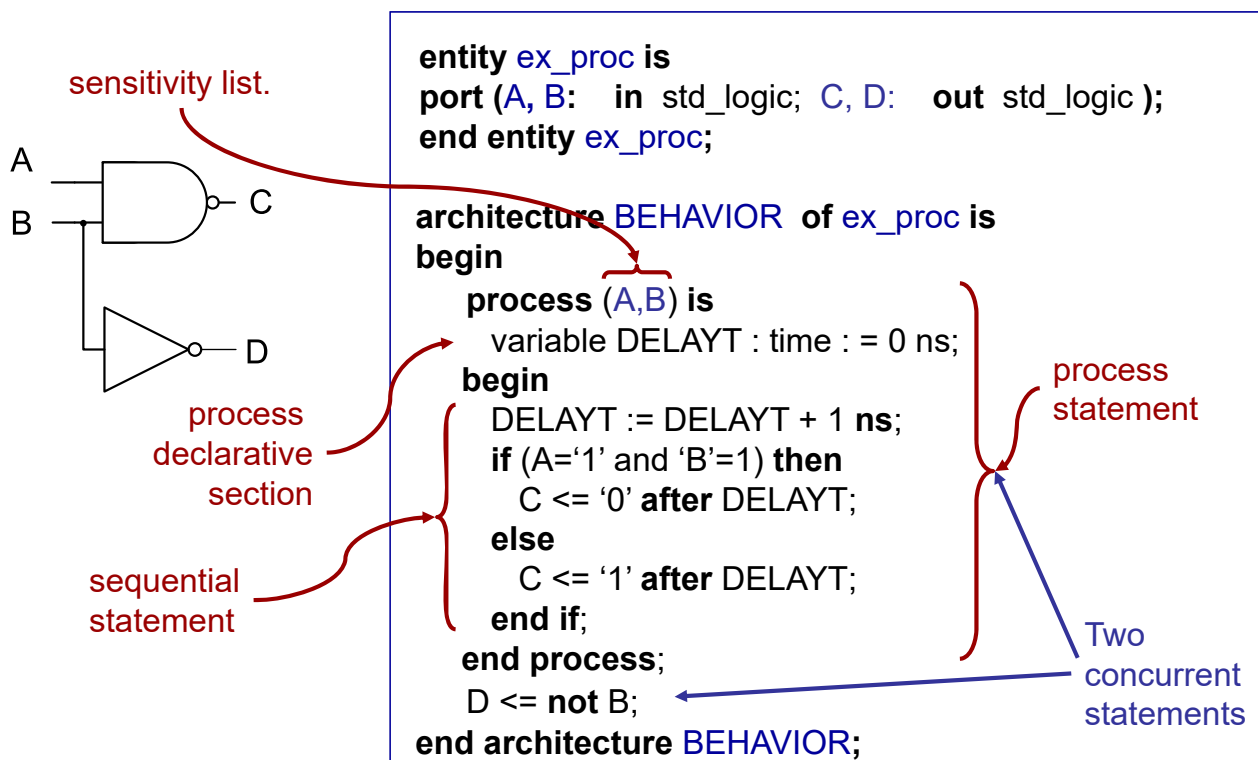
s1 <= not In1;
s2 <= not In2;
s3 <= not (s1 and In2);
s4 <= not (s2 and In1);
z <= not (s3 and s4);

```

# Chapter 4. Sequential VHDL

- Sequential statements are specified inside a process.
- Processes represent fundamental method by which concurrent activities are modeled.

## 4.1. The process construct



- A process is a large concurrent statement that itself contains a series of sequential statements.
- The body of the process is treated by VHDL as a single concurrent statement and is executed at the same time as all other concurrent statements in the simulation.

## Process with a sensitivity list

- The sensitivity list is a list of signal to which the process responds.

```

signal a, b, c, y: std_logic; -- in architecture declaration
...
process (a, b, c) -- in architecture body
begin
    y <= a and b and c; -- infer a combinational circuit
end process;

```

```

signal a, b, c, y: std_logic;
...
process (a) -- a process with incomplete sensitivity list
           -- not infer a combinational circuit
begin
    y <= a and b and c;
end process;

```

# Process with a wait statement

- A process with wait statements has one or more wait statement but no sensitivity list.

**wait on** signals;

**wait until** boolean\_expression;

**wait for** time\_expression;

```
process
begin
  y <= a and b and c;
  wait on a, b, c;
end process;
```

In synthesis, only few well defined forms of wait statement can be used, and normally only one wait statement is allowed in a process.  
**Not recommended.**

```
process (a, b) is
begin
  c <= a and b;
end process;
```

sensitivity list. The process will be activated if either a or b (or both) changes states

```
Process is
begin
  c <= a and b;
  wait on a, b;
end process;
```

A process can have wait statement or sensitivity list, but not both

```
process is
begin
  c <= a and b;
end process;
```

} This is an infinite loop.

## 4.2. Sequential statements

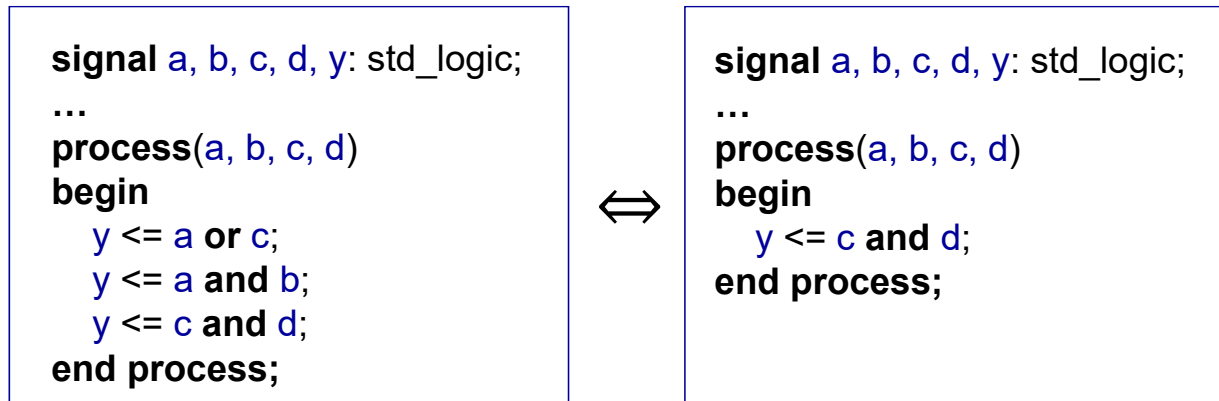
- sequential signal assignment statement
- variable assignment statement
- if statement
- case statement
- null statement
- wait statement
- loop statement
- exit statement
- next statement

### 4.2.1 Signal assignment statement

*Signal-object* <= expression [**after** delay-value];

- Outside a process – concurrent signal assignment statement
- Within a process – sequential signal assignment, executed in sequence with respect to the other sequential statement which appear within that process.
- When a signal assignment statement is executed, the value of expression is computed immediately, and this computed value is scheduled to be assigned to the signal after the specified delay.

- Inside a process, a signal can be assigned multiple times. If all assignments are with  $\delta$ -delay, only the last assignment takes effect.



- Although this segment is easy to understand, multiple assignment may introduce subtle mistakes in a more complex code and make synthesis very difficult.

- Unless there is a compelling reason, it is a good idea to avoid assign a signal multiple times in a process. The **only exception** is the assignment of a default value in the if and case statement.
- The result will be very different if the multiple assignments are the concurrent signal assignment statements

```

signal a, b, c, d, y: std_logic;
...
-- the statements are not inside a process
y <= a or c;
y <= a and b;
y <= c and d;

```

- The above code is syntactically correct.
- However, the design is incorrect because of the potential output conflict.

## 4.2.2 Variable assignment statement

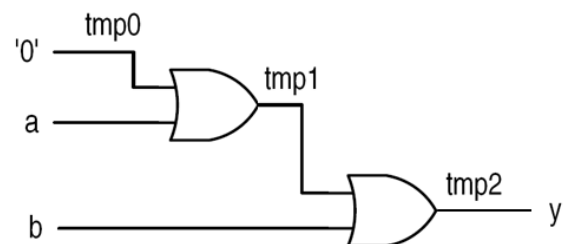
- Variables can be declared and used inside a process statement.
- Variable assignment statement:  
*variable-object := expression;*
- Variable assignment is immediate.

```
Signal a, b, y : std_logic;
.....
process (a, b) is
  variable temp : std_logic;
begin
    temp := '0';
    temp := temp or a;
    temp := temp or b;
    y <= temp;
end process;
```

Although the behavior of a variable is easy to understand, mapping it to hardware is difficult. **Not recommended.**

□ To realize the previous process in hardware, we have to rename the variables.

```
process (a, b) is
  variable tmp0, tmp1, tmp2 : std_logic;
begin
    tmp0 := '0';
    tmp1 := tmp0 or a;
    tmp2 := tmp1 or b;
    y <= tmp2;
end process;
```



For the lack of clear hardware mapping, we should try to use signal in code in general and resort to variable only for the characteristic that cannot be described by signals.

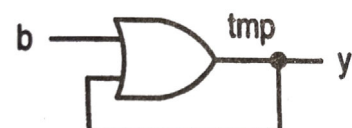
□ For comparison purpose

```
Signal a,b,y,tmp : std_logic;
process (a, b, tmp) is
begin
    tmp <= '0';
    tmp <= tmp or a;
    tmp <= tmp or b;
    y <= tmp;
end process;
```



```
Signal a,b,y,tmp : std_logic;
process (a, b, tmp) is
begin
    tmp <= tmp or b;
    y <= tmp;
end process;
```

**Not recommended.**





## 4.2.3 If statement

- An if statement selects a sequence of statements for execution based on the value of a condition.

- General form of an if statement

```

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

boolean type

```

if SUM <= 100 then
    SUM := SUM + 100;
end if;

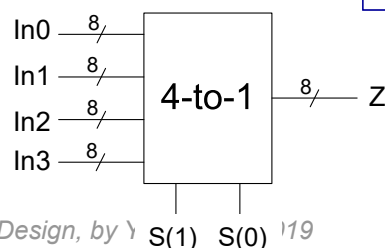
if NICKLE_IN then
    DEPOSITED <= T5;
elsif DIME_IN then
    DEPOSITED <= T10;
elsif QUARTER_IN then
    DEPOSITED <= T25;
else DEPOSITED <= Terror;
end if;
  
```

Note the absence of the letter 'e' in elsif !

### 4-to-1 multiplexer based on an if statement

```

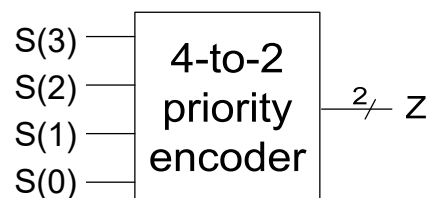
architecture if_arch of mux4 is
begin
    process(In0, In1, In2, In3, S)
    begin
        if (S = "00") then
            Z <= In0;
        elsif (S = "01") then
            Z <= In1;
        elsif (S = "10") then
            Z <= In2;
        else Z <= In3;
        end if;
    end process;
end if_arch;
  
```



### 4-to-2 priority encoder based on an if statement

```

architecture if_arch of pr_encoder is
begin
    process(S)
    begin
        if (S(3) = '1') then
            Z <= "11";
        elsif (S(2) = '1') then
            Z <= "10";
        elsif (S(1) = '1') then
            Z <= "01";
        else Z <= "00";
        end if;
    end process;
end if_arch;
  
```



## Comparison to a conditional assignment statement

- An if statement is somewhat like a concurrent conditional signal assignment statement.

```
sig <= value_exp1 when boolean_exp1 else  
      value_exp2 when boolean_exp2 else  
      value_exp3 when boolean_exp3 else  
      ...  
      value_expn;
```

### Process(...)

**begin**

**if** boolean\_exp1 **then** sig <= value\_exp1;

**elsif** boolean\_exp2 **then** sig <= value\_exp2;

**elsif** boolean\_exp3 **then** sig <= value\_exp3;

...

**else** sig <= value\_expn;

**end if;**

**end process;**

```
sig <= value_exp1 when boolean_exp1 else  
      value_exp2 when boolean_exp2 else  
      value_exp3 when boolean_exp3 else  
      ...  
      value_expn;
```

- The above two statements are equivalent.
- In general, equivalent only if each branch of the if statement consists of a single assignment of the same single signal.

- An if statement allows for arbitrary nesting of if statement.

```

process (CTRL1, CTRL2)
begin
  if CTRL1 = '1' then
    if CTRL2 = '0' then
      MUX_OUT <= "0010";
    else
      MUX_OUT <= "0001";
    end if;
  else
    if CTRL2 = '0' then
      MUX_OUT <= "1000";
    else
      MUX_OUT <= "0100";
    end if;
  end if;
end process;

```

**Write the  
equivalent of  
conditional signal  
assignment**

## Incomplete if branch

- In VHDL, only the then branch is mandatory and the other branches can be omitted.

```

process (a, b)
begin
  if (a = b) then
    eq <= '1';
  else
    eq <= eq;
  end if;
end process;

```

=

```

process (a, b)
begin
  if (a = b) then
    eq <= '1';
  end if;
end process;

```

≠

```

process (a, b)
begin
  if (a = b) then
    eq <= '1';
  else
    eq <= '0';
  end if;
end process;

```

ImPLY a circuit with a closed feedback loop,  
which constitutes memory

ImPLY a combinational  
circuit

# Incomplete signal assignment

```
process (a, b)
begin
  if (a > b) then
    gt <= '1';
  elsif (a=b) then
    eq <= '1';
  else
    lt <= '1';
  end if;
end process;
```

ImPLY a circuit with  
memories

≠

```
process (a, b)
begin
  if (a > b) then
    gt <= '1';
    eq <= '0';
    lt <= '0';
  elsif (a=b) then
    gt <= '0';
    eq <= '1';
    lt <= '0';
  else
    gt <= '0';
    eq <= '0';
    lt <= '1';
  end if;
end process;
```

=

```
process (a, b)
begin
  gt <= '0';
  eq <= '0';
  lt <= '0';
  if (a > b) then
    gt <= '1';
  elsif (a=b) then
    eq <= '1';
  else
    lt <= '1';
  end if;
end process;
```

ImPLY a combinational  
circuit

## Conceptual Implementation

```
if b_expr_1 then sig <= v_expr_1;
elsif b_expr_2 then sig <= v_expr_2;
elsif b_expr_3 then sig <= v_expr_3;
else sig <= v_expr_4;
end if;
```

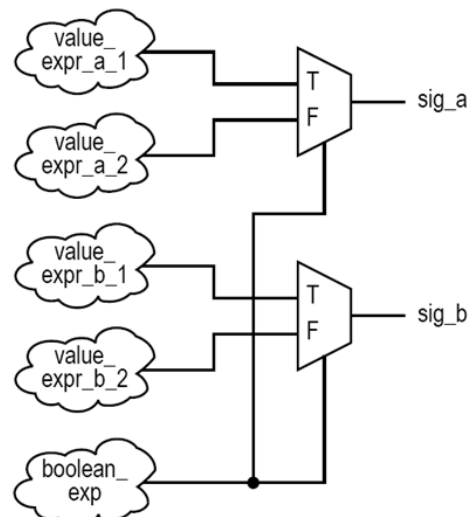
=

```
sig <= v_expr_1 when b_expr_1 else
v_expr_2 when b_expr_2 else
v_expr_3 when b_expr_3 else
v_expr_4;
```

Single output signal : Refer to Page 83

```
if boolean_expr then
  sig_a <= value_expr_a_1;
  sig_b <= value_expr_b_1;
else
  sig_a <= value_expr_a_2;
  sig_b <= value_expr_b_2;
end if;
```

Multiple output signals

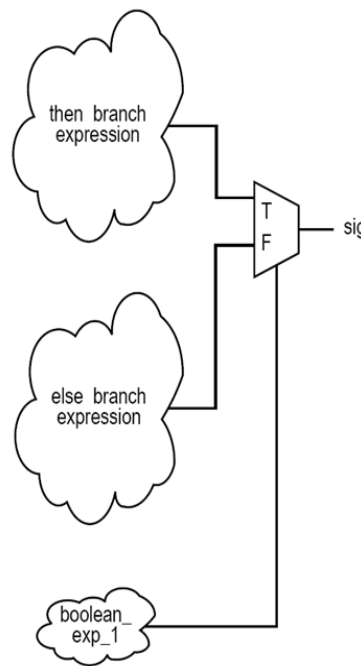


```

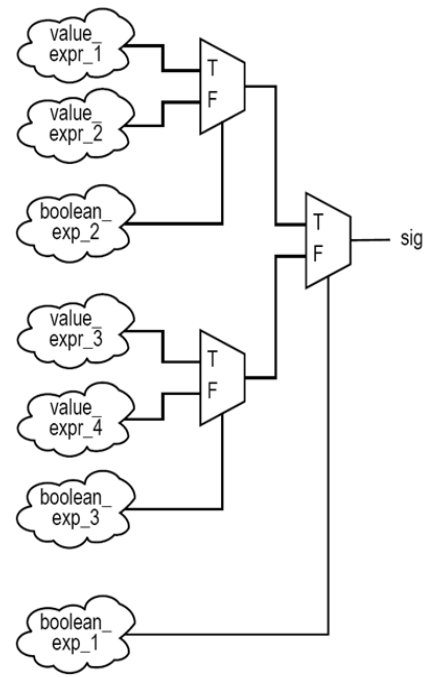
if boolean_expr_1 then
  if boolean_expr_2 then
    sig <= value_expr_1;
  else
    sig <= value_expr_1;
  end if;
else
  if boolean_expr_3 then
    sig <= value_expr_3;
  else
    sig <= value_expr_4;
  end if;
end if;

```

Nested if statement



Outer if statement



With two inner if statements

## 4.2.4 Case statement

- A case statement is a multiway branch based on the value of a control expression
- General form of a case statement

```

case expression is
  when choices =>
    {sequential-statements}
  { when choices =>
    { sequential-statements } }
end case;

```

integer type, enumerated type,  
one dimensional character array type  
such as BIT\_VECTOR.

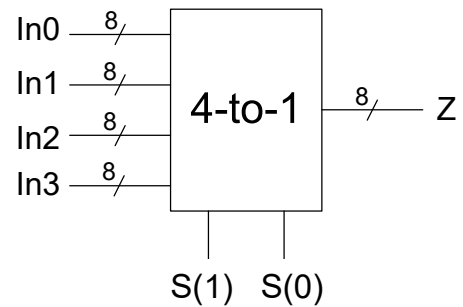
Must have the same type as the expression.  
either a static expression or a static range.

All possible value in the range of *expression's* type must be covered by one and only one choice.

The final choice can be others, which match all remaining choices in the range of *expression's* type.

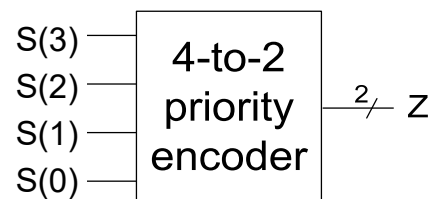
## 4-to-1 multiplexer based on a case statement

```
architecture case_arch of mux4 is
begin
  process(In0, In1, In2, In3, S)
  begin
    case S is
      when "00" =>
        Z <= In0;
      when "01" =>
        Z <= In1;
      when "10" =>
        Z <= In2;
      when others =>
        Z <= In3;
    end case;
  end process;
end case_arch;
```



## 4-to-2 priority encoder based on a case statement

```
architecture case_arch of pr_encoder is
begin
  process(S)
  begin
    case S is
      when "1000" | "1001" | "1010" | "1011" |
           "1100" | "1101" | "1110" | "1111" =>
        Z <= "11";
      when "0100" | "0101" | "0110" | "0111" =>
        Z <= "10";
      when "0010" | "0011" =>
        Z <= "01";
      when others =>
        Z <= "00";
    end case;
  end process;
end case_arch;
```



```

signal S1 : integer range 0 to 7;
Signal OU : bit;
signal I1, I2, I3 : bit;
.....
select_process: process(S1, I1, I2, I3) is
begin
    case S1 is
        when 0 | 2 => OU <= '0' ;
        when 1 => OU <= I1;
        when 3 to 5 => OU <= I2;
        when others => OU <= I3;
    end case ;
end process select_process;

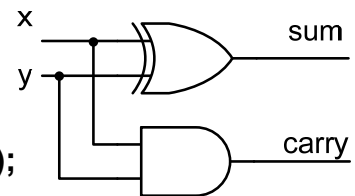
```

**Example:**  
A two-  
process  
half-adder  
model

```

library IEEE;
use IEEE.std_logic_1164.all;
entity half_adder is
port (x, y : in std_logic; sum, carry : out std_logic);
end entity half_adder;
architecture behavior of half_adder is
begin
    sum_proc: process (x, y) is -- this process computes the value of sum
    begin
        if (x=y) then sum <= '0' after 5 ns;
        else sum <= (x or y) after 5 ns;
        end if;
    end process sum_proc;
    carry_proc: process (x,y) is -- this process computes the value of carry
    begin
        case x is
            when '0' => carry <= x after 5 ns;
            when '1' => carry <= y after 5 ns;
            when others => carry <= 'X' after 5 ns;
        end case;
    end process carry_proc;
end architecture behavior;

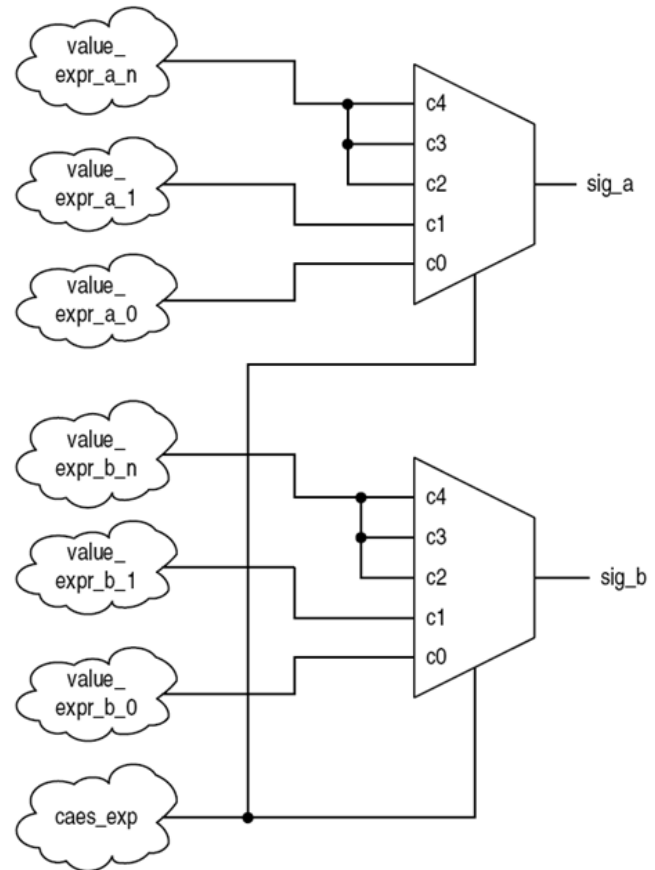
```



## Conceptual Implementation

```

case case_exp is
  when c0 =>
    sig_a <= value_expr_a_0;
    sig_b <= value_expr_b_0;
  when c1 =>
    sig_a <= value_expr_a_1;
    sig_b <= value_expr_b_1;
  when others =>
    sig_a <= value_expr_a_n;
    sig_b <= value_expr_b_n;
end case;
  
```



## 4.2.5 NULL statement

- The statement

null;

is a sequential statement that does not cause any action to take place;

**variable** SEL : integer range 0 to 31;

**variable** V : integer range 0 to 31;

**case** SEL **is**

**when** 0 to 15 => V := SEL;

**when** others => null;

**end case**;



## 4.3 More on process

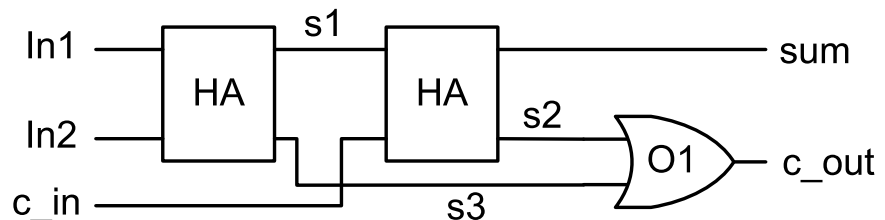
- Upon initialization all processes are executed once, or suspended on some form of the wait statement reached.
- Thereafter, processes are executed in a data-driven manner: activated
  - by events on signals in the sensitivity list of the process or
  - by waiting for the occurrence of specific event using the wait statement.

**Example:**  
Signal  
assignment  
with  
process

```
library IEEE;
use IEEE.std_logic_1164.all;
entity sig_var is
port (x, y, z : in std_logic; res1, res2 : out std_logic);
end entity sig_var;
architecture behavior of sig_var is
signal sig_s1, sig_s2 : std_logic;
begin
  proc1: process (x, y, z) is
    variable var_s1, var_s2 : std_logic;
    begin
      L1: var_s1 := x and y;
      L2: var_s2 := var_s1 xor z;
      L3: res1 <= var_s1 nand var_s2;
    end process proc1;
  proc2: process (x, y, z) is
    begin
      L1: sig_s1 <= x and y;
      L2: sig_s2 <= sig_s1 xor z;
      L3: res2 <= sig_s1 nand sig_s2;
    end process proc2;
  end architecture behavior;
```

- All of the ports of the entity and the signals declared within an architecture are visible within a process.
- These port and signals can be read or assigned values from within a process; this is how processes can communicate among themselves.

**Example:**  
Communicating  
processes

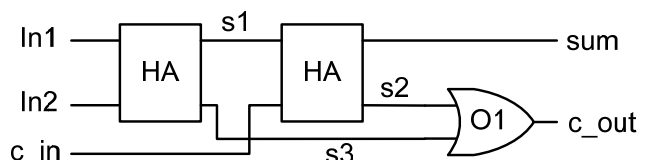


```
library IEEE
use IEEE.std_logic_1164.all
entity full_adder is
port (In1, In2, c_in : in std_logic;
      sum, c_out : out std_logic);
end entity full_adder;
```

**Example:** architecture behavioral of full\_adder is  
Communicating  
processes  
(continued)

```

signal s1, s2, s3 : std_logic;
constant delay : time := 5 ns;
begin
  HA1: process (In1, In2) is -- process describing the first half adder
  begin
    s1 <= (In1 xor In2) after delay;
    s3 <= (In1 and In2) after delay;
  end process HA1;
  HA2: process (s1, c_in) is -- process describing the second half adder
  begin
    sum <= (s1 xor c_in) after delay;
    s2 <= (s1 and c_in) after delay;
  end process HA2;
  OR1: process (s2, s3) is -- process describing the two-input OR gate
  begin
    c_out <= (s2 or s3) after delay;
  end process OR1;
end architecture behavioral;
```

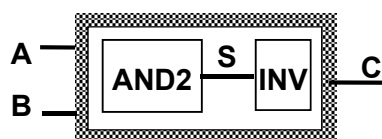
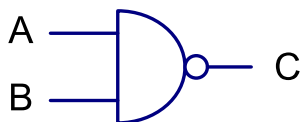


# Chapter 5: Modeling Structure

- Describing a system in terms of the interconnections of its components.
- Motivation
  - represent a digital system in hierarchical structure.
  - share components between developers.

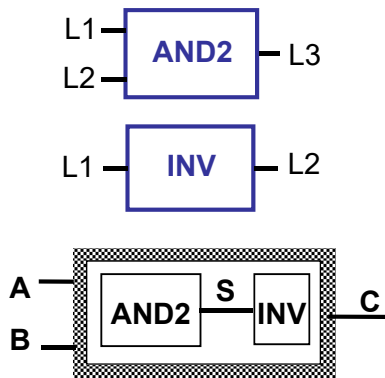
## 5.1: Component

### NAND2 Gate



## Architecture: Structure style

Component declaration:  
a component instance  
needs to be bound to an  
entity



Component  
instantiation

```
entity NAND2 is
    port (A, B: in bit;  C  : out bit );
end entity NAND2;
```

```
architecture STRUCTURE of NAND2 is
```

```
    component AND2 is
        port (L1, L2 : in bit;
              L3     : out bit);
    end component AND2;
```

```
    component INV is
        port (L1 : in bit;
              L2  : out bit);
    end component INV;
    signal S : BIT;
```

```
begin
```

```
    A1: AND2 port map (L1 => A,
                       L2 => B, L3 => S);
```

```
    A2: INV port map (L1=>S, L2=> C);
```

```
end architecture STRUCTURE;
```

## Component instantiation

port\_name → Actual signal name

```
A1 : AND2 port map (L1 => A, L2 => B, L3 => S);
```

**Name association** - the syntax “port\_name =>” is provided so that the order of the signals listed after PORT MAP keywords need **NOT** follow the order of the ports in the corresponding COMPONENT declaration.

```
A1 : AND2 port map (A, B, S);
```

**Positional association** – if the signal names following the PORT MAP keywords are given in the same order in the COMPONENT declaration, then “port\_name =>” is not needed.

## Component Entity: **AND2** and **INV**

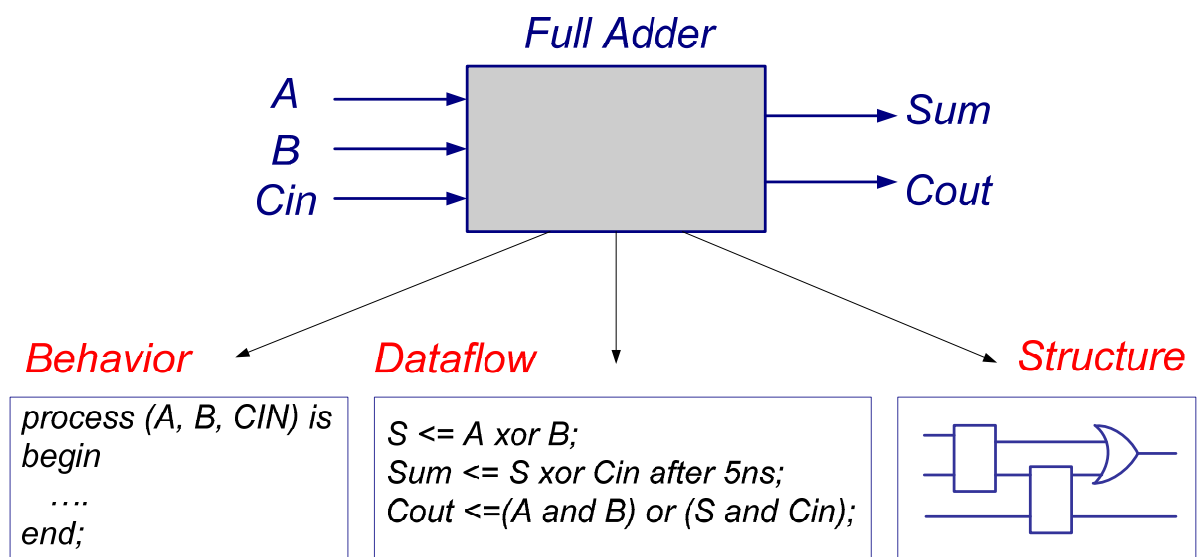
```
entity AND2 is
port (L1, L2 : in bit;
      L3 : out bit );
end entity AND2;
```

```
architecture dataflow of AND2 is
begin
  L3 <= L1 and L2;
end architecture dataflow;
```

```
entity INV is
port (L1 : in bit;
      L2 : out bit );
end entity INV;
```

```
architecture dataflow of INV is
begin
  L2 <= not L1;
end architecture dataflow;
```

## Full Adder: Entity and architecture



## Full Adder: Data flow style architecture

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

architecture DATAFLOW of Full_Adder is
  signal S: bit;
begin
  S <= A xor B;
  Sum <= S xor Cin after 5 ns;
  Cout <= (A and B) or (S and Cin);
end architecture DATAFLOW;
```

Time period between cause and effect.  
Signal pulse shorter than this time period is ignored.

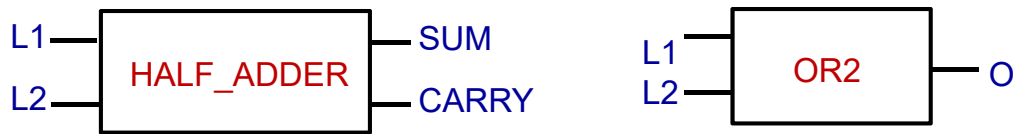
## Full Adder: Behavioral style architecture

```
architecture behavioral of full_adder is
begin
  HA1: process (In1, In2, c_in) is
  begin
    case In1&In2&c_in is
      when "000" =>
        sum <= '0'; c_out <= '0';
      when "111" =>
        sum <= '1'; c_out <= '1';
      when "001" | "010" | "100" =>
        sum <= '1'; c_out <= '0';
      when others =>
        sum <= '0'; c_out <= '1';
    end case;
  end process HA1;
end architecture behavioral;
```

| In1 | In2 | c_in | sum | c_out |
|-----|-----|------|-----|-------|
| 0   | 0   | 0    | 0   | 0     |
| 0   | 0   | 1    | 1   | 0     |
| 0   | 1   | 0    | 1   | 0     |
| 0   | 1   | 1    | 0   | 1     |
| 1   | 0   | 0    | 1   | 0     |
| 1   | 0   | 1    | 0   | 1     |
| 1   | 1   | 0    | 0   | 1     |
| 1   | 1   | 1    | 1   | 1     |

# Full Adder:

structural style architecture



architecture STRUCTURE of Full\_Adder is

component HALF\_ADDER is

port (L1, L2 : in bit; SUM, CARRY : out bit);

end component HALF\_ADDER;

component OR2 is

port (L1, L2 : in bit; O: out bit);

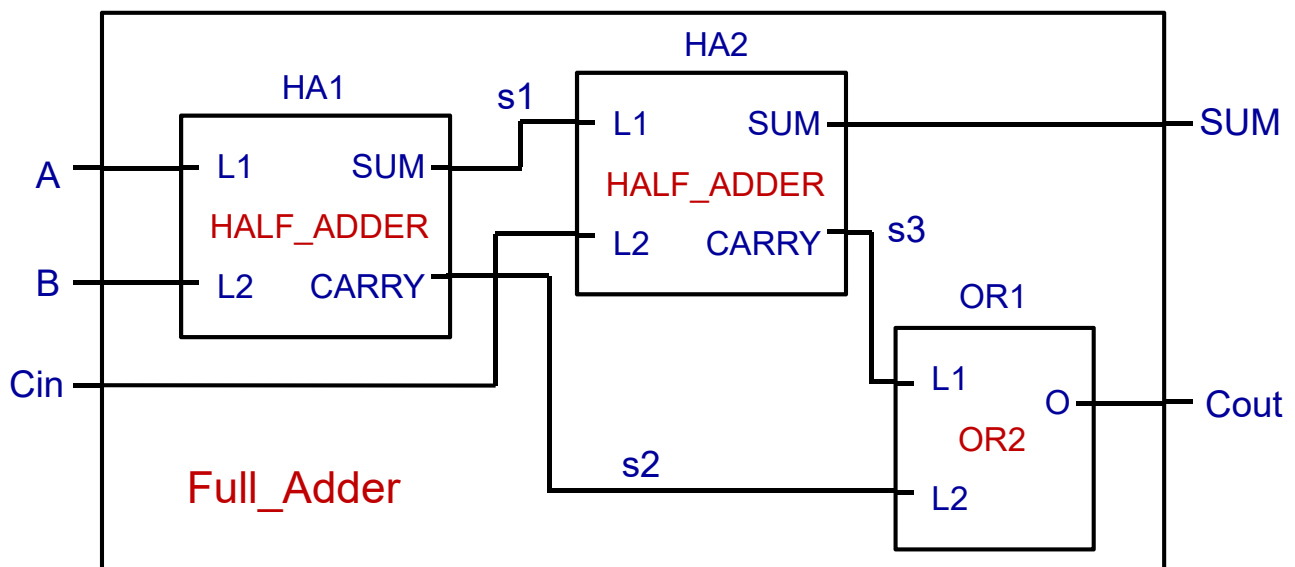
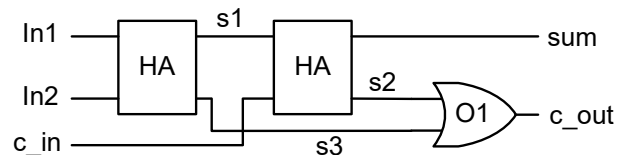
end component OR2;

signal s1, s2, s3 : BIT

begin

-- see next page;

end architecture STRUCTURE;



architecture STRUCTURE of Full\_Adder is

-- Declare components and intermediate signal here. See last page;

begin

HA1: HALF\_ADDER port map (L1=>A, L2=>B, SUM=>s1, CARRY=>s2);

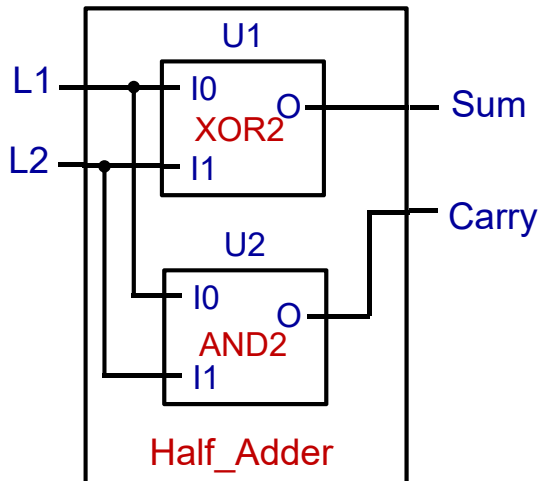
HA2: HALF\_ADDER port map (L1=>s1, L2=>CIN, SUM=>SUM, CARRY=>s3);

OR1: OR2 port map (L1=>s3, L2=>s2, O=>COUT);

end architecture STRUCTURE;

## Component Entity: Half\_Adder

```
entity Half_Adder is
  port (L1, L2 : in bit;
        Sum: out bit;
        Carry: out bit );
end entity Half_Adder;
```



```
architecture STRUCTURE of Half_Adder is
  component XOR2 is
    port (I0, I1: in BIT; O: out bit);
  end component XOR2;
  component AND2 is
    port (I0, I1 : in BIT; O: out bit);
  end component AND2;
begin
  U1: XOR2 port map (I0 => L1,
                    I1 => L2, O => Sum);
  U2: AND2 port map (I0 => L1, I1 => L2,
                    O => Carry);
end architecture STRUCTURE;
```

## Component Entity: OR2, AND2, XOR2

```
entity OR2 is
  port (L1, L2: in bit; O: out bit );
end entity OR2;
```

```
architecture BHV of OR2 is
begin
  O <= L1 or L2 after 10 ns;
end architecture BHV;
```

```
entity AND2 is
  port (I0, I1: in bit; O: out bit );
end entity AND2;
```

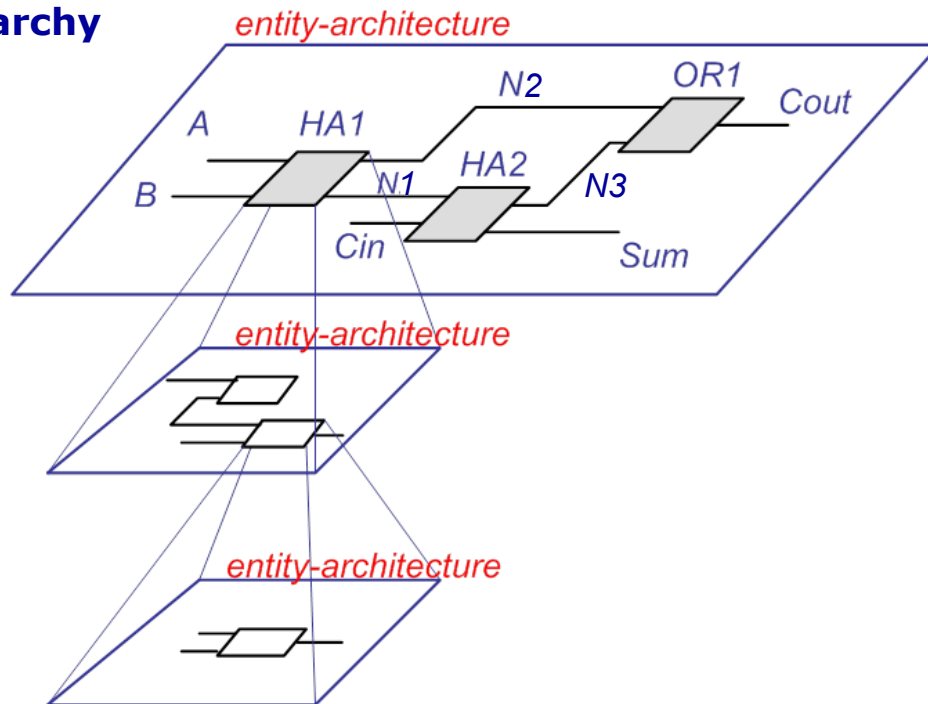
```
architecture dataflow of AND2 is
begin
  O <= I0 and I1;
end architecture dataflow;
```

```
entity XOR2 is
  port (I0, I1: in bit; O: out bit );
end entity XOR2;
```

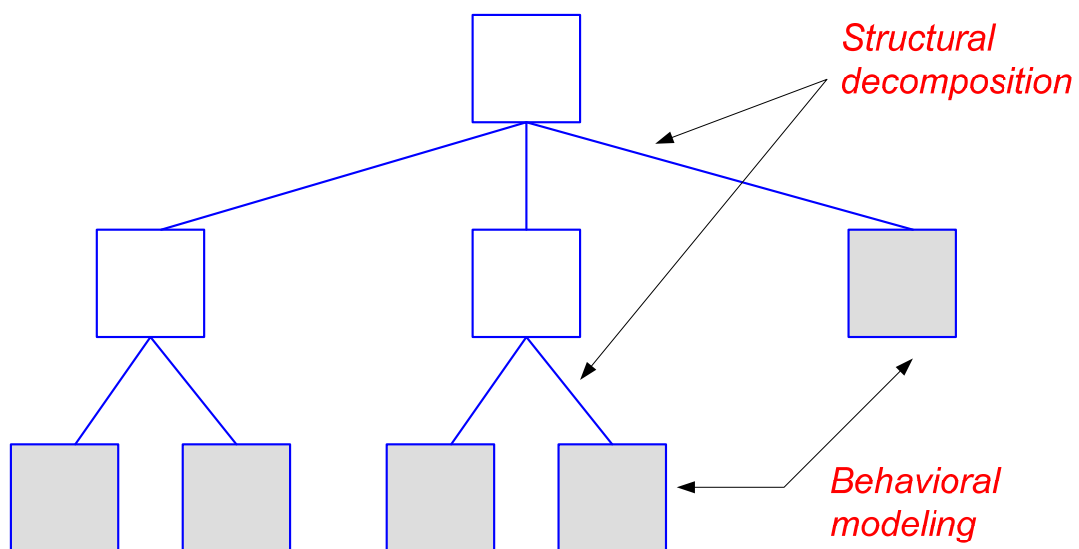
```
architecture BHV of XOR2 is
begin
  O <= I0 xor I1 after 10 ns;
end architecture BHV;
```



# Structural Decomposition: A design Hierarchy

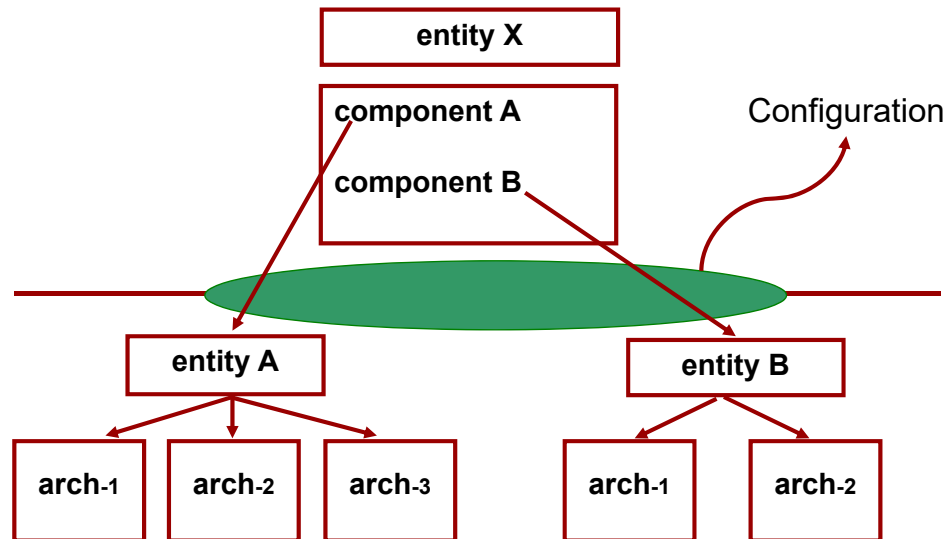


# Structural Decomposition: Design Tree



## 5.2. Configuration

- When there is more than one architecture for an entity, configuration explicitly specifies which architecture is to be used for the entity during component instantiation.



- The process of association of an architecture description with a component in a structure model is referred to as *binding* an architecture to a component.
- Default binding rules:
  - The entity with the same name as the component is bound to the component.
  - If there are multiple architectures for the entity, the last compiled architecture for the entity is used.
  - The entity-architecture description may locate in the same file as that of instantiating the component, or in some other files in the working directory.

```
entity HALF_Adder is
port (L1, L2 : in bit;
      SUM, CARRY : out bit );
end entity Half_Adder;
```

Two architectures

```
architecture STRUCTURE of HALF_Adder is
  component XOR_GATE is
    port (I0, I1: in BIT; O : out BIT);
  end component XOR_GATE;
  component AND2 is
    port (I0, I1 : in BIT; O : out BIT);
  end component AND2;
begin
  U1: XOR_GATE port map (L1, L2, SUM);
  U2: AND2 port map (L1, L2, CARRY);
end architecture STRUCTURE;
```

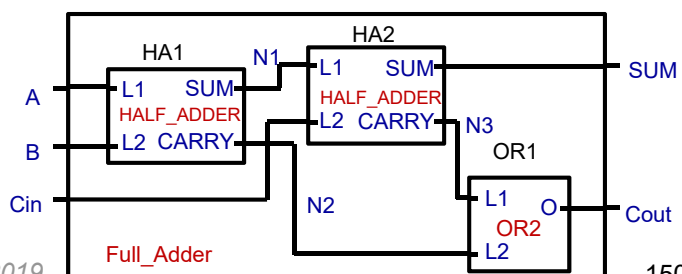
One

```
architecture BEHAVIOR of HALF_ADDER is
begin
  SUM <= L1 xor L2;
  CARRY <= L1 and L2;
end architecture BEHAVIOR;
```

The other one

```
architecture STRUCTURE of Full_Adder is
  component HALF_ADDER is
    port (L1, L2 : in bit; SUM, CARRY : out bit);
  end component HALF_ADDER;
  component OR_GATE is
    port (L1, L2 : in bit; O : out bit);
  end component OR_GATE;
  for HA1: HALF_ADDER use entity HALF_ADDER(BEHAVIOR);
  for HA2: HALF_ADDER use entity HALF_ADDER(STRUCTURE);
  signal N1, N2, N3 : BIT;
begin
  OR1: OR2 port map (L1=>N3, L2=>N2, O=>COUT);
  HA1: HALF_ADDER port map (L1=>A, L2=>B, SUM=>N1,CARRY=>N2);
  HA2: HALF_ADDER port map (L1=>N1, L2=>CIN, SUM=>SUM,
    CARRY=>N3);
end architecture STRUCTURE;
```

Configuration specifications



- If the entity-architecture descriptions locate in some other directories, the configuration specification have to includes the library name.

```
for HA1: HALF_ADDER use entity mydesign.HALF_ADDER(arch1);
for HA2: HALF_ADDER use entity mydeisgn.HALF_ADDER(arch2);
```

- If there is only one architecture in an entity, the architecture name can be omitted.

```
for OR1: OR_GATE use entity work.OR_GATE;
```

- Keywords: **all**, **others**

```
for all: HALF_ADDER use entity work.HALF_ADDER(arch1);
```

```
for HA1: HALF_ADDER use entity work.HALF_ADDER(arch1);
```

```
for others: HALF_ADDER use entity work.HALF_ADDER(arch2);
```

Library name

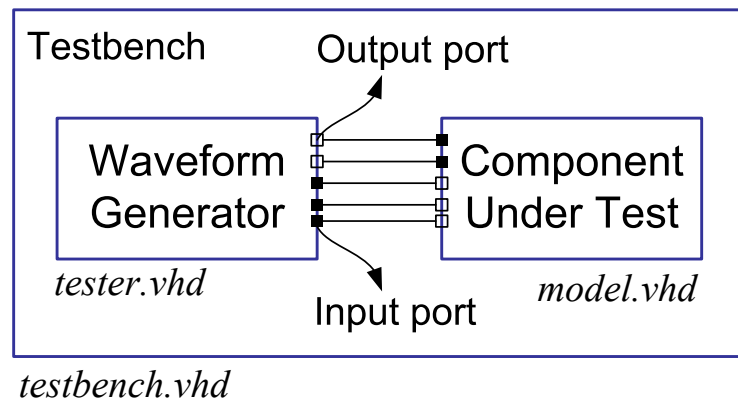
Entity name

Architecture name

## 5.3. Modeling a test bench

- Motivation
  - Test designs prior to construction and use of the circuit.
- To test a compiled VHDL design, we can either
  - provide stimuli interactively through the command window of a simulator, or
  - write a VHDL program (a VHDL test bench).

- A test bench
  - does not have external ports.
  - contains two part:
    - a component representing the circuit under test
    - waveform generators which produce waveforms to the input of the component under test.



### 5.3.1 Functional verification of combinational designs

- Determine if a design meets a system's functional specifications.
- Not concern with any timing delays that results from mapping synthesized logic to a target programmable logic device.
- Allow to find logic errors early in the design flow and prevent wasting time performing synthesis, place and route, and time simulation.

## Language and approach to be used

- Full range of VHDL language constructs and features can be used for test bench.
- Exhaustive verification
  - Counting approach treating all inputs as a single vector starting from 0 and subsequently incremented through all its possible binary combinations.
  - Functionality approach taking into account the functionality of the design being tested when determining the order for applying input combination

### 5.3.2 A simple test bench

```
library ieee;      -- load the ieee 1164 library
use ieee.std_logic_1164.all;  -- make the package 'visible'

entity test_half_adder is  -- the top level entity of the test bench has no
                           -- external ports.
end entity test_half_adder;

architecture test of test_half_adder is
  component half_adder is -- declare the component under test
    port (x, y: in std_logic; sum, carry: out std_logic);
  end component half_adder;

  for UUT: half_adder use entity -- configuration specifications
  work.half_adder(concurrent_behavior);
```

*-- Stimulus signals – to connect test bench to UUT input ports*

**signal** x\_tb, y\_tb: std\_logic;

*-- Observed signals – to connect test bench to UUT output ports*

**signal** sum\_tb, carry\_tb: std\_logic;

**begin**

*-- Create an instance of the half\_adder circuit.*

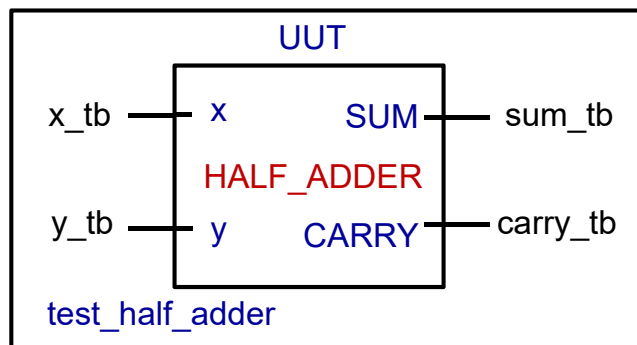
**UUT: half\_adder port map** (x => x\_tb, y => y\_tb, sum =>  
sum\_tb, carry => carry\_tb);

*-- Signal assignment statements generating stimulus values*

x\_tb <= '1', '0' **after** 5 ns, '1' **after** 10 ns, '0' **after** 15 ns;

y\_tb <= '1', '0' **after** 10 ns;

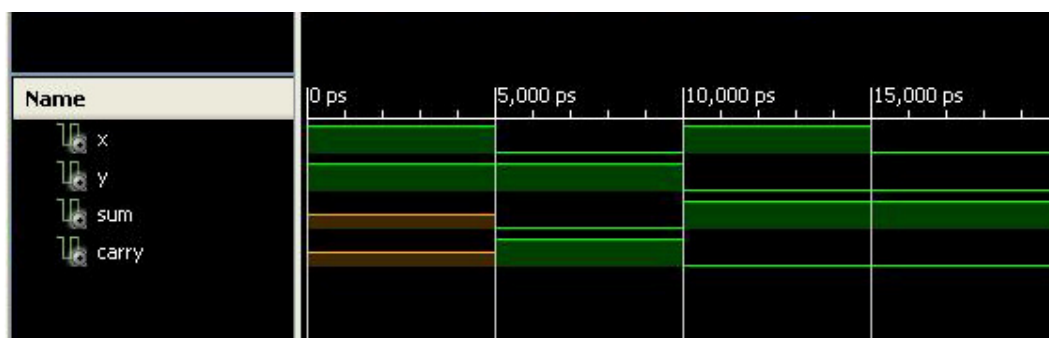
**end architecture** test;



EE332 Digital System Design, by Yu Yajun -- 2019

157

## Waveforms from simulation of half-adder test bench



### 5.3.3 Single process test bench

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

entity test_half_adder is
end entity test_half_adder;

architecture behavior of test_half_adder is
    component half_adder is
        port (x, y: in std_logic; sum, carry: out std_logic);
    end component half_adder;
    for UUT: half_adder use entity
        work.half_adder(concurrent_behavior);

    signal x_tb, y_tb: std_logic;
    signal sum_tb, carry_tb: std_logic;
```

**begin**

```
    UUT: half_adder port map (x => x_tb, y => y_tb, sum => sum_tb,
        carry => carry_tb);
```

```
    -- define a process to apply input stimulus and verify outputs.
```

```
    tb: process is
```

```
        constant PERIOD: time := 20 ns;
```

```
    begin -- apply every possible input combination
```

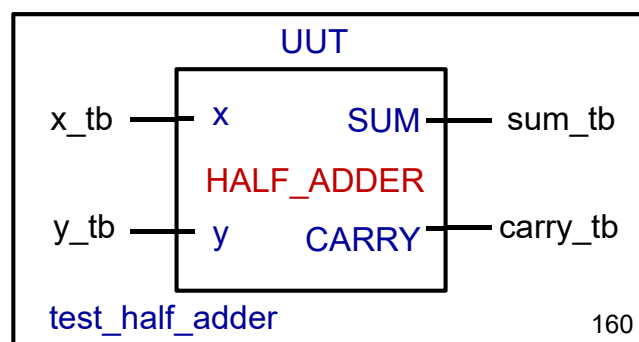
```
        x_tb <= '0'; -- apply input combination 00 and check outputs
```

```
        y_tb <= '0';
```

```
        wait for PERIOD; -- Wait for outputs to be available after
                        -- applying this stimulus
```

```
        assert ((sum_tb = '0') and (carry_tb = '0'))
```

```
        report "Test failed for input combination 00" severity ERROR;
```





```

x_tb <= '1'; -- apply input combination 10 and check outputs
y_tb <= '0';
wait for PERIOD;
assert ((sum_tb = '1') and (carry_tb = '0'))
report "Test failed for input combination 01" severity ERROR;
x_tb <= '0';
y_tb <= '1';
wait for PERIOD;
assert ((sum_tb = '1') and (carry_tb = '0'))
report "Test failed for input combination 10" severity ERROR;
x_tb <= '1';
y_tb <= '1';
wait for PERIOD;
assert ((sum_tb = '0') and (carry_tb = '1'))
report "Test failed for input combination 11" severity ERROR;
wait; -- indefinitely suspend process
end process;
end architecture behavior;

```

## Wait statement (Sequential statement)

- explicitly specify the conditions under which a process may resume execution after being suspended.
- Basic forms of the wait statement
 

|  |                                    |
|--|------------------------------------|
| <b>wait for</b> <i>time-expression</i> ; | <b>wait for</b> 20 ns;             |
| <b>wait on</b> <i>signal</i> ;           | <b>wait on</b> clk, reset, status; |
| <b>wait until</b> <i>condition</i> ;     | <b>wait until</b> A > B;           |
| <b>wait</b> ;                            |                                    |

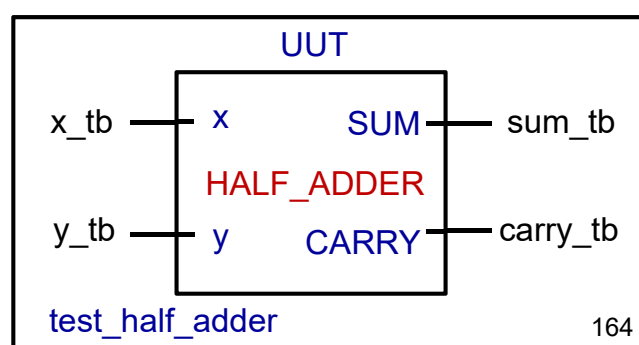
## Assert statement

- VHDL's assert statement provides a quick and easy way to check expected values and display messages from your test bench. An assert statement has the following general format:

```
assert condition_expression -- a Boolean value
report text_string          -- the text is displayed if the Boolean
                             -- value is false
severity severity_level; -- severity level can be one of
                             -- predefined levels: NOTE,
                             -- WARNING, ERROR, or FAILURE.
```

### 5.3.4 Test benches that compute stimulus and expected results

```
tb: process is -- define a process to apply input stimulus and verify outputs.
    constant PERIOD: time := 20 ns;
    constant n : integer := 2;
begin -- apply every possible input combination
    for i in 0 to 2**n - 1 loop
        (x_tb, y_tb) <= to_unsigned(i, n);
        wait for PERIOD;
        assert ( (sum_tb = (x_tb xor y_tb)) and (carry_tb = (x_tb and y_tb)) )
        report "Test failed" severity ERROR;
    end loop;
    wait;
end process;
```



## Loop statement

- A loop statement is used to iterate through a set of sequential statement.
- General form of a loop statement

```
[ loop-label ] iteration-scheme loop  
    sequential-statements  
end loop [ loop-label ]
```

- Three types of iteration schemes:

(A)

```
for i in 0 to 2**n - 1 loop  
    (x_tb, y_tb) <= to_unsigned(i, n);  
    wait for PERIOD;  
    ...  
end loop;
```

Diagram annotations:

- A red arrow points from the text "loop identifier" to the variable `i` in the `for` statement.
- A red bracket above the range `0 to 2**n - 1` is connected by a red line to the text "range".

- The object `i` is implicitly declared within the for loop to belong to the integer type whose value are in the range  $0$  to  $2^n-1$ .
- The loop identifier cannot be assigned any value inside the for loop

```
type COLOR is (RED, GREEN, BLUE);  
...  
for PAPER in COLOR loop  
    -- PAPER will take all value in type COLOR  
    -- from RED to BLUE.  
end loop;
```

```
(B)  J := 0; SUM := 10;
      WHILE_LOOP: while J < 20 loop
          SUM := SUM * 2;
          J := J + 3;
      end loop WHILE_LOOP;
```

```
(C)  J := 0; SUM := 1;
      L2: loop
          J := J+21;
          SUM := SUM * 10;
          exit when SUM > 100;
      end loop L2;
```

## Exit statement

- The exit statement can be used only inside a loop.
- It causes execution to jump out of the innermost loop or the loop whose label is specified.
- General form of an exit statement

```
exit [ loop-label ] [ when
    condition ]
```

```
SUM := 1;
L2: loop
    J := 0;
    L3: loop
        J := J+21;
        exit when J > 40;
        SUM := SUM * 10;
        if SUM > 100 then exit L2;
        end if;
    end loop L3;
end loop L2;
```

```
loop
    wait on A, B;
    exit when A=B;
end loop;
```

## Next statement

- The next statement can be used only inside a loop.
- It results in skipping the remaining statements in the current iteration; execution resumes with the first statement in the next iteration of this loop, if one exists.

- General form of a next statement

**next** [ *loop-label* ] [ **when** *condition* ]

```
for J in 10 downto 5 loop
  if SUM < TOTAL_SUM then
    SUM := SUM + 2;
  elsif SUM = TOTAL_SUM then
    next;
  else null;
  end if;
  K := K + 1;
end loop;
```

## signed and unsigned data type

- In VHDL and the std\_logic\_1164 package, the arithmetic operations are defined only over the integer data type.

```
signal a, b, sum: integer;
```

```
...
```

```
sum <= a+b;
```

- Data types signed and unsigned are defined in IEEE numeric\_std package. Both data types are an array of element with the std\_logic data type. They are interpreted as a signed number or unsigned number.
- std\_logic\_vector, signed, and unsigned are all defined as an array of elements with the std\_logic data type, but they are three independent data types.

- To use the signed and unsigned data type, we must include:

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

signal x, y: signed(15 downto 0);
```

## to\_unsigned(i, n)

- convert the integer *i* value to an unsigned vector of length *n*
- (x\_tb, y\_tb) <= to\_unsigned(i, n);
  - the unsigned vector value returned in this example is assigned to an aggregate made up of the scalar input signals.
  - each of these scalar is type std\_logic.
  - each element of an unsigned vector is also type std\_logic.

Table: Overloaded operators in the IEEE numeric\_std package

| Operator   | Description          | Data type of a   | Data type of b  | Data type of result                      |
|--|----------------------|--|---|--|
| <b>abs a</b>   | absolute value       | signed   |   |  |
| <b>-a</b>  | negation             | signed   |   |  |
| <b>a * b, a / b,<br/>a mod b,<br/>a rem b,<br/>a + b, a - b</b>        | arithmetic operation | unsigned<br>unsigned, natural<br>signed<br>signed, integer | unsigned, nature<br>unsigned<br>signed, integer<br>signed | unsigned<br>unsigned<br>signed<br>signed |
| <b>a = b, a /= b,<br/>a &lt; b, a &lt;= b,<br/>a &gt; b, a &gt;= b</b> | relational operation | unsigned<br>unsigned, natural<br>signed<br>signed, integer | unsigned, nature<br>unsigned<br>signed, integer<br>signed | boolean<br>boolean<br>boolean<br>boolean |

```
signal a, b, c, d, e: unsigned (7 downto 0);
```

```
...
a <= b + c;      d <= b + 1;      e <= (5 + a + b) - c;
```

```
"011" >= "1000"; -- return FALSE if type is std_logic_vector or unsigned
                  -- return TRUE if type is signed
```

**Table: Functions in the IEEE numeric\_std package**

| Function  | description  | Data type of a                                      | Data type of b | Data type of result |
|---|--|---|----------------|---------------------|
| shift_left(a, b)<br>shif_right(a, b)<br>rotate_left(a, b)<br>rotate_right(a, b) | shift left<br>shift right<br>rotate left<br>rotate right | unsigned, signed                                    | natural        | same as a           |
| resize(a,b)   | resize array   | unsigned, signed                                    | natural        | same as a           |
| std_match(a,b)  | compare '-'  | unsigned, signed,<br>std_logic_vector,<br>std_logic | same as a      | boolean             |
| to_integer(a)   | data type<br>conversion                                  | unsigned, signed                                    |                | integer             |
| to_unsigned(a,b)  |  | natural   | natural        | unsigned            |
| to_signed(a,b)  |  | integer   | natural        | signed              |

- std\_logic\_vector, unsigned and signed are known as *closely related data types*. Conversion between these types is done by a procedure known as *type casting*.

**Table: Type conversion of numeric data types**

| Data type of a             | To data type     | Conversion function/type casting |
|----------------------------|------------------|----------------------------------|
| unsigned, signed           | std_logic_vector | std_logic_vector(a)              |
| signed, std_logic_vector   | unsigned         | unsigned(a)                      |
| unsigned, std_logic_vector | signed           | signed(a)                        |
| unsigned, signed           | integer          | to_integer(a)                    |
| natural                    | unsigned         | to_unsigned(a, size)             |
| integer                    | signed           | to_signed(a, size)               |

```

signal u1, u2: unsigned (7 downto 0);
signal v1, v2: std_logic_vector(7 downto 0);
...
u1 <= unsigned(v1);
v2 <= std_logic_vector(u2);

```

```

library ieee;
use ieee.std_logic_1164.all
use ieee.numeric_std.all
...
signal s1, s2, s3, s4, s5, s6: std_logic_vector (3 downto 0);
signal u1, u2, u3, u4, u5, u6: unsigned(3 downto 0);
signal sg: signed(3 downto 0);
...
u3 <= u2 + u1; -- ok
u4 <= u2 + 1; -- ok
u5 <= sg; -- not ok
u6 <= 5; -- not ok

s3 <= u3; -- not ok
s4 <= 5; -- not ok

s3 <= std_logic_vector(u3); -- ok
s4 <= std_logic_vector(to_unsigned(5,4));
-- ok

u5 <= unsigned(sg); -- ok
u6 <= to_unsigned(5,4); -- ok

s5 <= s2 + s1; -- not ok
s6 <= s2 + 1; -- not ok

u7 <= sg + u1; -- not ok
u7 <= unsigned(sg) + u1; -- ok

s5 <= std_logic_vector(unsigned(s2) +
unsigned(s1)); -- ok
s6 <= std_logic_vector(unsigned(s2) +
1); -- ok

```

### 5.3.5 Post-synthesis and timing verifications for combinational designs

- **Post-synthesis verification**
  - to verify that the synthesizer has successfully translated a design description to gate-level logic.
  - the same test bench used for functional verification could be used.
- **Timing verification**
  - to verify gate delays and propagation delays of signal paths of the logic mapped to the target programmable logic device.
  - If the delay between application of each stimulus and verification of the corresponding UUT outputs was appropriately chosen in the original functional verification test bench, the same test bench could be used for timing verification.