BASIC VHDL FOR DIGITAL SYSTEMS DESIGN

VHDL

Introduction

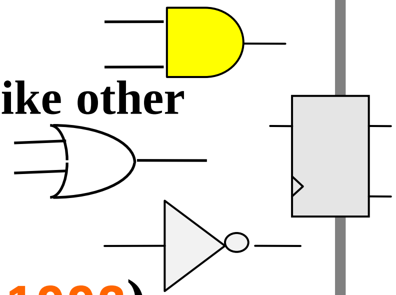
What is VHDL?

VHDL stands for ***V**HSIC **H**ardware **D**escription **L**anguage*

(VHSIC: **Very High Speed Integrated Circuits**)

VHDL is high level programming language which is optimized for describing the behaviour of digital systems.

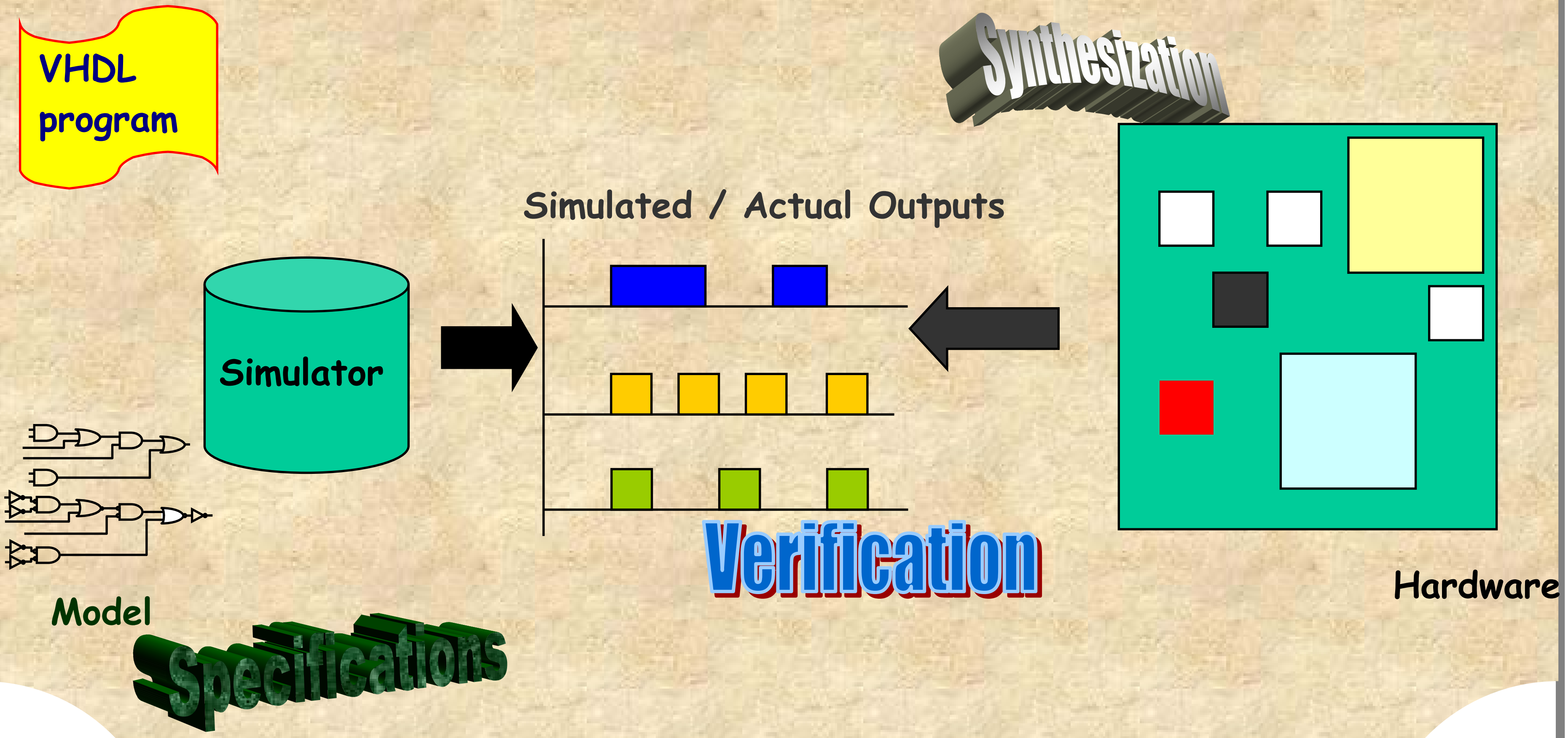
VHDL is capable of describing concurrent events (unlike other languages) which is crucial for describing real circuits.



VHDL is an IEEE Standard (Std 1076-1987, Std 1076-1993).

VHDL is one of many HDLs (see self-reading material).

Major Purposes of VHDL



VHDL program

Synthesization

Simulated / Actual Outputs

Simulator

Verification

Hardware

Model

Specifications

Major Purposes of VHDL

A simulation modeling language:

VHDL allows for describing behavior of electronic circuits (from simple logic gates to complete μPs) at high levels of detail. These simulation models can be used as building blocks in larger circuits.

A specification language:

VHDL can be used to capture the performance & interface requirements of each component in a large system including precise descriptions of functional & timing aspects (e.g. rise time, fall time etc).

A design entry language:

VHDL allows complex designs to be expressed as computer programs (some also allow for schematic entry as well).

A verification language:

VHDL can capture performance specification for a circuit in the form of test bench (descriptions of circuit stimuli & corresponding expected outputs)

A synthesization language:

VHDL's structural features enable it to be used as a netlist language (like EDIF) for targeted implementation in CPLD, FPGA or ASIC.

Why VHDL?

Advantages:

- Shorter Design Cycles
- Improved Design Quality
- Vendor and Technology Independence
- Lower Design Cost
- Easy Design Management

Disadvantages:

- Cost (incl. training)
- Debugging

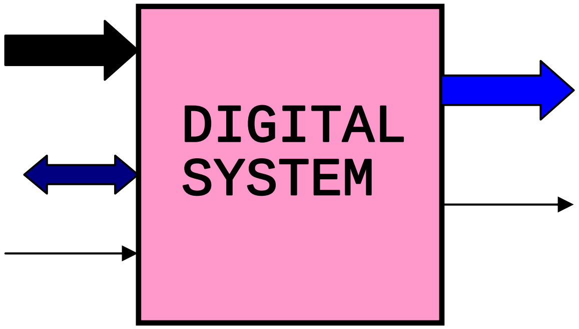
VHDL

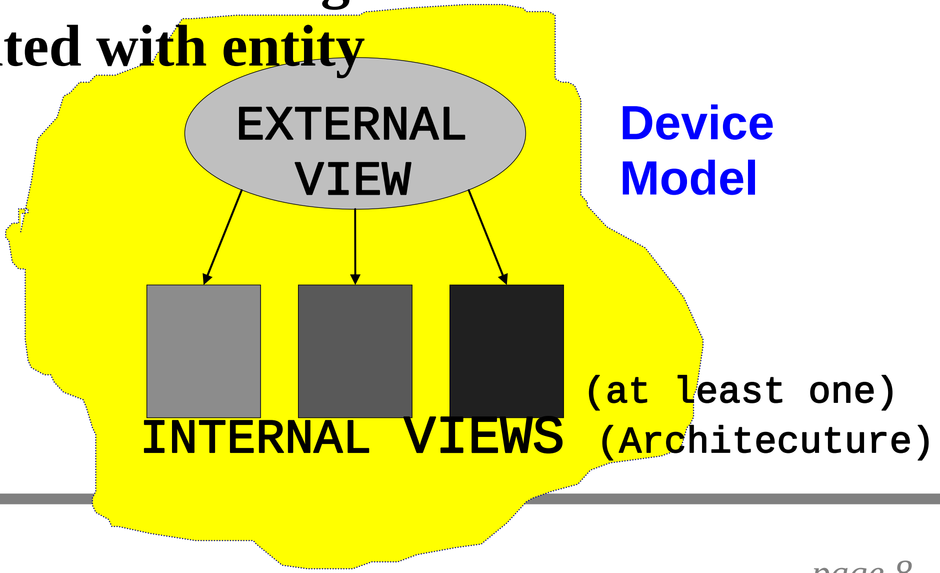
CONCEPTS

Device modeling in VHDL: Entity & Architecture

VHDL describes a model for a digital device by specifying:

- *external view of device (entity)*: interface through which it communicates with other models.
- *internal views of device (architecture)*: (one or more) specifies functionality or internal structure.
- *binding (configuration)*: specifies binding of one architecture body from others associated with entity

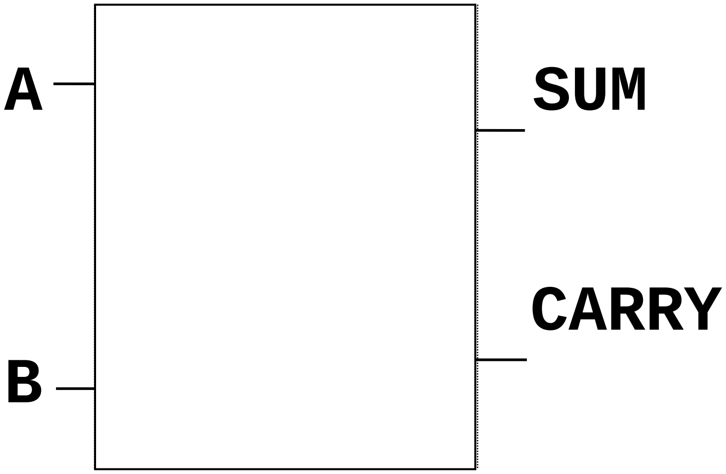




Entity Declaration

- Specifies *entity name* & *interface ports* but *not* internals
- *Example*: entity declaration for *Half Adder* :

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



```
IEEE stand logic type:
type STD_LOGIC is (
 'U',   -- uninitialized (default initial value)
 'X',   -- forcing an unknown
 '0',   -- forcing 0
 '1',   -- forcing 1
 'Z',   -- high impedance (three state)
 'W',   -- weak unknown
 'L',   -- weak O
 'H',   -- weak 1
 '-' ); -- don't care
```

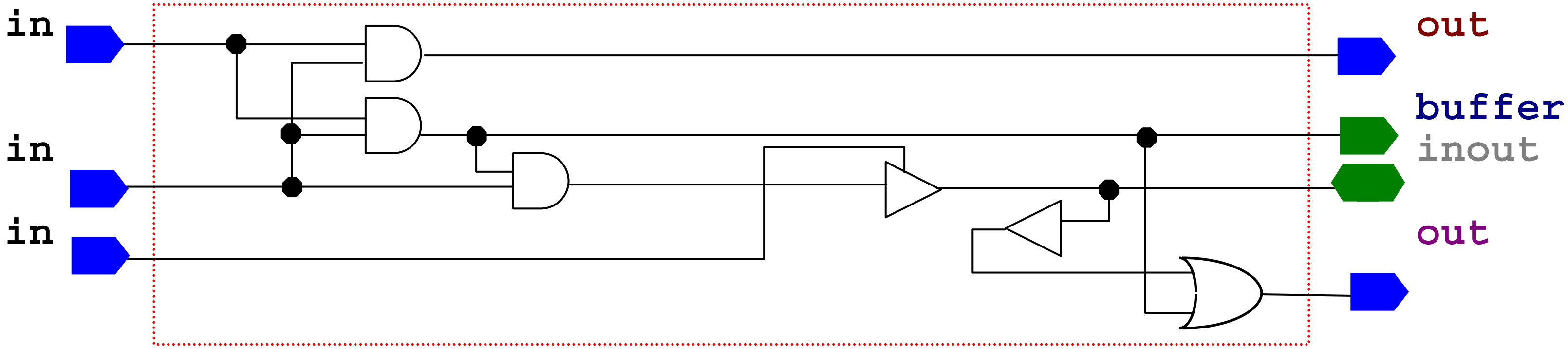
```
entity HALF_ADDER is
 port ( A ,   B  :  in std_logic;
       SUM, CARRY: out std_logic);
end HALF_ADDER;
-- this is a comment line, after – upto end of line
-- case-insensitive: CARRY,CarrY refer to same name
-- entity HALF ADDER has two input ports (A,B),
-- two output ports, (SUM, CARRY) of type std_logic
```

Entity Declaration

```
-- general form:
entity entity-name is
  [ generic ( list-of-generics-and-their-types ) ;]
  [ port(list-of-interface port-names-and-their-types) ;]
  [ entity-item-declarations]
  [ begin
    entity-statements ]
end [ entity ] [ entity-name] ;
```

lists port *names* and *modes* (i.e., direction)

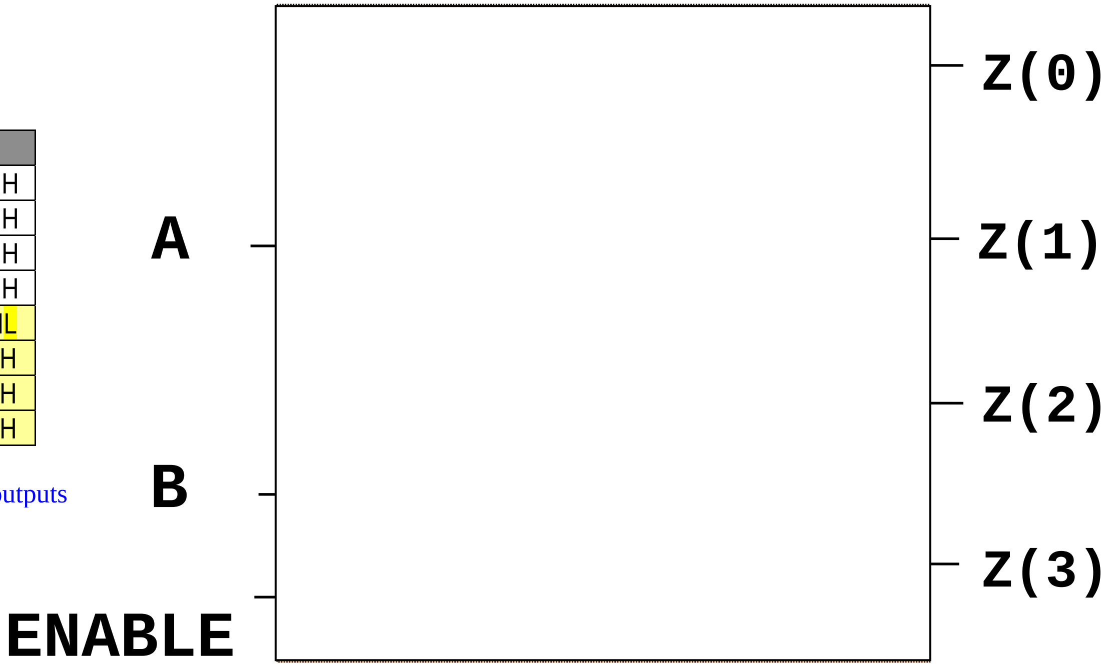
in	*input port*: can only be read and not assigned in the circuit
out	*output port*: can only be assigned and not read in the circuit
inout	*bi-directional port*: can be used only for tristate buses
buffer	*buffer port*: provides internal feedback but not bidirectional feedback



Entity Declaration for the *2-to-4 decoder* circuit

A	B	ENABLE	Z
L	L	L	HHHH
L	H	L	HHHH
H	L	L	HHHH
H	H	L	HHHH
L	L	H	HHHL
L	H	H	HHLH
H	L	H	HLHH
H	H	H	LHHH

Decoder with **Active –low** outputs



```
entity DECODER2x4 is
  port (A,B,ENABLE: in std_logic;
        Z: out std_logic_vector(0 to 3));
 end DECODER2x4;
 -- entity DECODER2x4 has 3 input and 4 output ports,
 -- port type BIT_VECTOR is a array type of std_logic
```

Architecture

Specifies underlying function/structure of an entity and is described using one of the following *modeling styles*:

- **a set of concurrent assignment statements**
- **a set of sequential assignment statements**
- **a set of interconnected components**
- **any combination of the above three**

dataflow

behavioral

structural

Each *internal view* is described using a *separate* architecture.

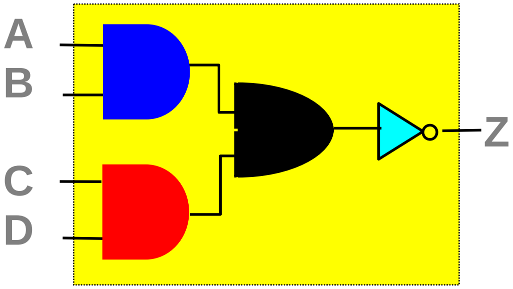
```
-- general form:
architecture architecture-name of entity-name is
   [ architecture-item-declarations ]
begin
   concurrent-statements; these are -->
       process-statement
       block-statement
       concurrent-procedure-call-statement
       concurrent-assertion-statement
       concurrent-signal-assignment-statement
       component-instantiation-statement
       generate-statement
end [ architecture ] [ architecture-name ];
```

statements execute in parallel

AOI circuit architecture (*concurrent*)

```
entity AOI is
 port (A,B,C,D  : in std_logic;
           Z     :out std_logic);
end AOI;

architecture AOI_CONCURRENT of AOI is
begin
  Z <= not ( (A and B) or (C and D));
end AOI_CONCURRENT;
```

-- Architecture body AOI_CONCURRENT consists of concurrent
-- statements that execute *in* parallel



don't specify delays for PLD realizations

Concurrent signal assignment statement assigns values to signals:

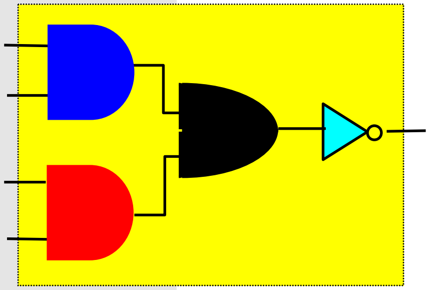
```
signal-object <= expression [after delay-value];
```

when there's an event on a RHS signal, expression is evaluated & computed value is assigned to LHS signal after specified delay. If no specified delay, a default delta delay is assumed.

For every *concurrent signal assignment statement*, there is an equivalent process with the same semantic meaning. PTO!

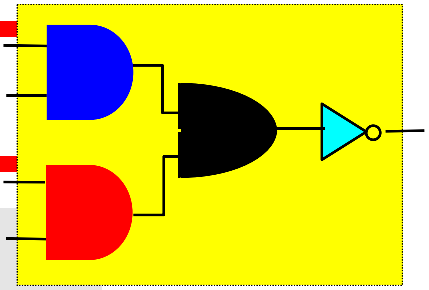
AOI circuit architecture (*sequential*)

```
entity AOI is
  port (A, B, C, D: in std_logic; Z: out std_logic);
end AOI ;
-- Architecture AOI_SEQUENTIAL uses behavioral
--                                     style of modeling
architecture AOI_SEQUENTIAL of AOI is
begin
 process (A, B, C, D)
     variable TEMP1, TEMP2: std_logic;
 begin
   TEMP1 := A and B;         -- stmt1
   TEMP2 := C and D;         -- stmt2
   TEMP1 := TEMP1 or TEMP2;  -- stmt3
   Z <= not TEMP1;           -- stmt4
end process;
end AOI_SEQUENTIAL;
```



```
-- All statements in process are executed sequentially
-- When an event occurs on A,B,C or D, AOI_SEQUENTIAL is
-- executed in the order: stmt1,stmt2,stmt3,stmt4, and
-- then the process suspends and waits for another event.
```

AOI circuit architecture (*sequential*)



```
entity AOI is
  port (A, B, C, D: in std_logic; Z: out std_logic);
end AOI ;


architecture AOI_SEQUENTIAL of AOI is
begin
 process (A, B, C, D)
     variable TEMP1, TEMP2: std_logic;
 begin
    TEMP1 := A and B;            -- stmt1
    TEMP2 := C and D;            -- stmt2
    TEMP1 := TEMP1 or TEMP2;     -- stmt3
    Z <= not TEMP1;              -- stmt4
 end process;
end AOI_SEQUENTIAL;

-- All statements in process are executed sequentially in ZERO TIME.
-- When an event occurs on A,B,C or D, AOI_SEQUENTIAL is
-- executed in the order: stmt1,stmt2,stmt3,stmt4, and
-- then the process suspends and waits for another event.
```

When an event occurs on any signal in the sensitivity list, statements within the process are executed one after another, *sequentially*.

A process is either being *executed* or *suspended*

Variables are declared within a **process** and their scope limited to that **process**.

Variables are assigned values *instantaneously* using the variable assignment statement:

```
variable-object := expression;
```

In zero time!

Sequential signal assignment statement → executed sequentially with other statements in process.

When a signal assignment statement is triggered at time T, the output signals *will be scheduled* to get their new values at a later time.

process itself is a *concurrent* statement !

process statement - for describing sequential operations

process statement contains *sequential statements* that describe behaviour of a circuit over time

```
  -- general form
[process-label:] process [(sensitivity-list)] [is]
 [ process-item-declarations ]
begin
   sequential-statements; these are -->
   variable-assignment-statement
   signal-assignment-statement
   wait-statement
   if-statement
   case-statement
   loop-statement
   null-statement
   exit-statement
   next-statement
   assertion-statement
   report-statement
   procedure-call-statement
   return-statement
end process [ process-label];
```

declared items can only be used in the process

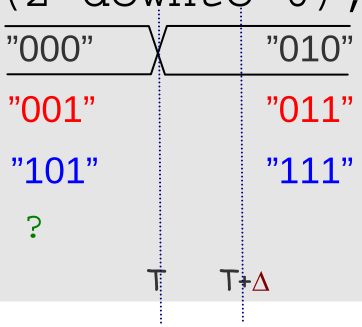
Signals cannot be declared in process!

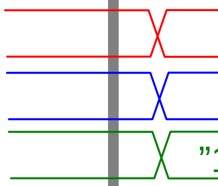
Variables only declared in process!

Variables declared outside of a **process** are called shared variables and can be read by more than one **process**.

Delta Delay: *an example*

```
signal A, Z: std_logic_vector(2 downto 0);
        :   :
  PZ: process ( A )      -- PZ is a label  (optional)
        variable V,W: std_logic_vector(2 downto 0);
begin                    --          A =  ”000”        ”010”
    V  := A OR “001”;    -- stmt1    V =  ”001”        ”011”
    Z  <= V OR “100”;    -- stmt2    Z =  ”101”        ”111”
    W  := Z;             -- stmt3    W =    ?
end process PZ;          --                      T    T+Δ
```



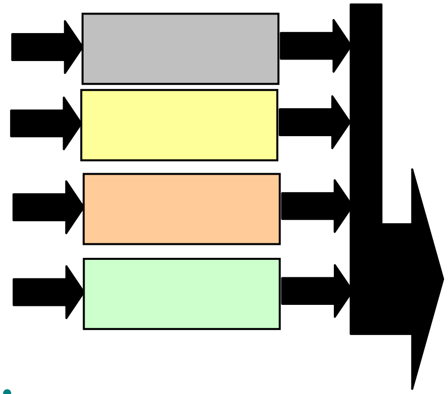


If an event occurs on signal A at simulation time T:

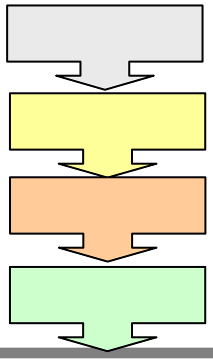
- `stmt1` **is executed and causes V to get a value,**
- **signal Z is *scheduled* to get a value at time T+Δ & finally**
- `stmt3` **is executed using *old value* of signal Z (at T) *since simulation time is still* T and *has not advanced to* T+Δ.**

Concurrent *versus* Sequential Signal Assignment

Concurrent signal assignment statements are *event-triggered*

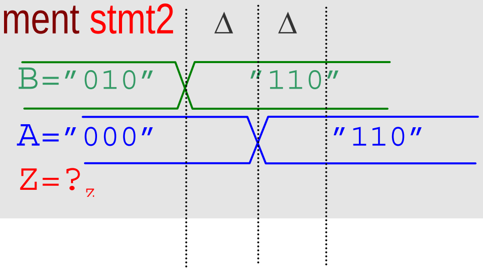


Sequential signal assignment statements are not event-triggered (by signals on the RHS) but are executed *in sequence* in relation to other sequential statements within `process`.



Concurrent *versus* Sequential Signal Assignment

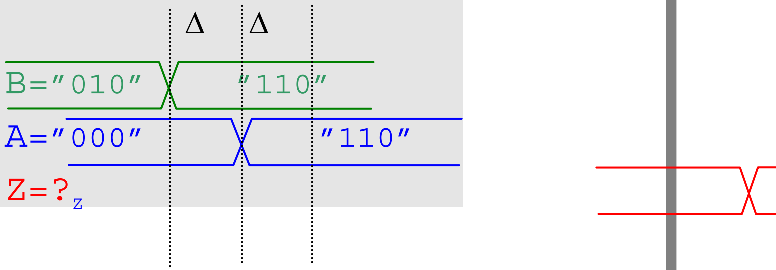
```
architecture SEQ_SIG_ASG of FRAGMENT1 is
begin
 process ( B )
   begin     -- following are sequential signal assignment stmts
   A <= B;   -- sequential signal assignment stmt1
   Z <= A;   -- sequential signal assignment stmt2
 end process;
end SEQ_SIG_ASG;
```



In architecture `SEQ_SIG_ASG`: when there's an event on signal B at time T, `stmt1` is executed then `stmt2`, both in *zero time*. Signal A is scheduled to get its new value (*of* B) at T+Δ, and Z is scheduled to be assigned the old value of A (not of B) at T+Δ.

Concurrent *versus* Sequential Signal Assignment

```
architecture CON_SIG_ASG of FRAGMENT2 is
begin
   A <= B;          -- concurrent signal assignment stmt1
   Z <= A;          -- concurrent signal assignment stmt2
end CON_SIG_ASG;
```



In architecture `CON_SIG_ASG`, when there is an event on signal **B** at time **T**, signal **A** gets the value of **B** at **T**+Δ. When simulation time advances to **T**+Δ, signal **A** will get its new value and this event on **A** (*if there's a change*) will trigger **`stmt2`** which will cause the new value of **A** to be assigned to **Z** at **T**+2Δ.

Dataflow first....

BASIC ELEMENTS OF VHDL

Basic elements of *VHDL*

VHDL **is a strongly typed language: the type of a data object defines the set of values that the object can assume and set of operations on those values.**

Basic elements of *VHDL*:

data types **: different types of data** (e.g. std_logic, integer**)**

operators **: operate on data values.**

Relational Operators: =, /= , <=, >=, >, <
Logical Operators: and or nand nor xor xnor not
Arithmetic Operators: + , - (for integer type)

New types **can be defined using type declarations:**

```
type DIGIT is (‘0‘,'1‘,'2‘,'3‘,'4‘,'5‘,'6‘,'7‘,‘8‘,'9');
type MICRO_OP is (LOAD,STORE,ADD,SUB,MUL,DIV);
type STATES is (state0,state1,state2,state3);
```

default value: left-most

Vectors..

Consider: (elements assigned by *position* not element number)

```
variable C : std_logic_vector( 3 downto 0);
variable D : std_logic_vector( 0 to 3);

                 --   D(0)   D(1)   D(2)   D(3)
D := "1010";     --    1      0      1      0
C := D;          --   C(3)   C(2)   C(1)   C(0)
```

Data assignment to vectors:

```
variable OP_CODES: std_logic_vector (1 to 5);
OP_CODES := "01001";                              -- string literal
OP_CODES := ('0', '1', '0', '0', '1');            -- positional association
OP_CODES := (2=>'1', 5=>'1', others=>'0') ;       -- named association
OP_CODES := (others=>'0');                        -- all values set to '0'
```

Overriding the default value:

```
variable SUM: INTEGER range 0 to 100 := 10;
```

Behavioral MODELING

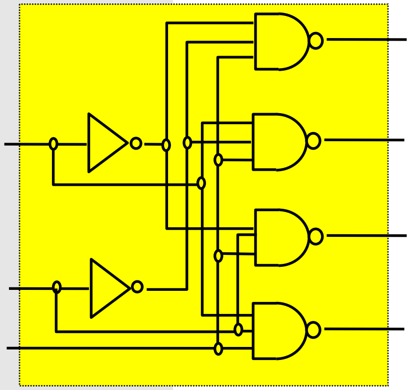
Behavioral VHDL Descriptions:

- **circuit behavior over time (i.e., sequential behavior) is described using either *state diagrams*, *timing diagrams*, or *algorithmic descriptions***
- **similar to conventional sequential computer programs**
- **use of registers are implied**
- `process` **is used for behavioral descriptions**

Behavioral Style of Modeling

```
-- Behavioural representation for DECODER2X4 entity

entity DECODER2x4 is
  port (A,B,ENABLE: in std_logic;
        Z: out std_logic_vector(0 to 3));
 end DECODER2x4;
 architecture DEC_SEQUENTIAL of DECODER2x4 is
  begin
  process (A, B, ENABLE)   -- sensitivity list,
    variable ABAR, BBAR: std_logic;
  begin
  ABAR  := not A;                       -- stmt 1
  BBAR  := not B;                       -- stmt 2
  if ENABLE = '1' then                  -- stmt 3
    Z(3)  <= not(A and B);              -- stmt 4
    Z(0)  <= not(ABAR and BBAR);        -- stmt 5
    Z(2)  <= not(A and BBAR);           -- stmt 6
    Z(1)  <= not(ABAR and B);           -- stmt 7
  else
    Z <= "1111";                        -- stmt 8
  end if;
  end process;
 end DEC_SEQUENTIAL;
```

When there's an event on a signal in *sensitivity list* ⇒ **process** is invoked and its statements are executed *sequentially* irrespective of an event on any RHS signal.



Combinational Logic Circuits can also be implemented using sequential statements without need for logic simplification.

Behavioral.. `if` **statement** (*nesting allowed*)

Use only in a process!

Each condition checked until *first* true condition. *Format:*

```
if boolean-expression then
    sequential-statements
{ elsif boolean-expression then -- elseif clause (optional)
    sequential-statements }
[ else sequential-statements ]  -- else clause (optional)
 end if;
     -- null  in if statement: does not cause any action to
     -- take place & execution continues with next statement
```

CTRL1	CTRL2	
0	0	“1000”
0	1	“0100”
1	0	“0010”
1	1	“0001”

```
-- example
process (CTRL1,CTRL2)
begin
if CTRL1 = '1' then
     if CTRL2 = '0' then          -- ctrl1 ctrl2
           data OUT <= "0010";  --   '1'   '0'
     else data OUT <= "0001";  --   '1'   '1'
     end if;
else
     if CTRL2 = '0' then          -- ctrl1 ctrl2
           data OUT <= "1000";  --   '0'   '0'
     else data OUT <= "0100";  --   '0'   '1'
     end if;
end if;
end process;
```

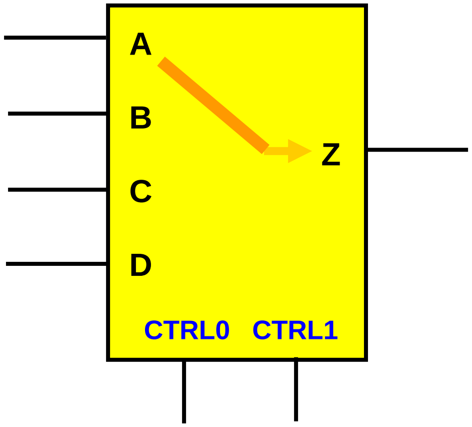
Behavioral.. `case` statement (*nesting allowed*)

use only in a process!

`case` selects one branch for execution.

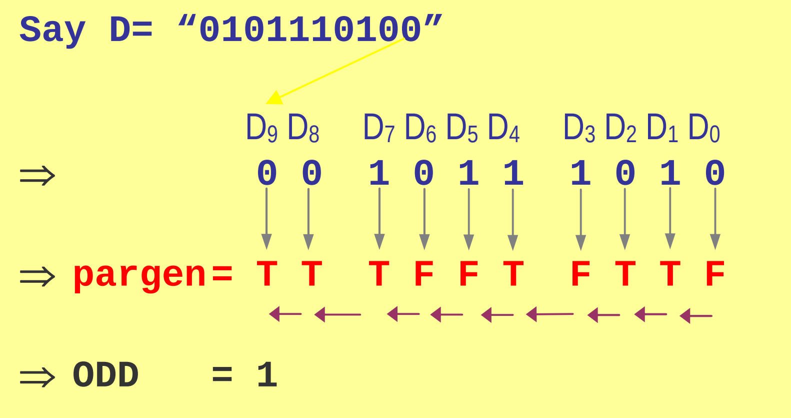
```
case expression is
   when choices => sequential-statements    -- branch #1
   when choices => sequential-statements    -- branch #2
[  when others  => sequential-statements ]  -- all options!
end case;
```

```
entity MUX is -- A 4 to 1 multiplexer using case
     port (A, B, C, D: in STD_LOGIC;
        CTRL:in  STD_LOGIC_VECTOR(0 to 1);
        Z:   out STD_LOGIC);
end MUX;
architecture MUX_BEHAVIOR of MUX is
begin
PMUX:    process (A, B, C, D, CTRL)
         variable TEMP: STD_LOGIC;
     begin
        case CTRL is
         when "00" => TEMP := A;
         when "01" => TEMP := B;
         when "10" => TEMP := C;
         when others => TEMP := D;
              -- std_logic has other possible values..
        end case;
        Z <= TEMP;
     end process PMUX;
end MUX_BEHAVIOR;
```



Behavioral.. loop scheme: for identifier in range

```
library ieee;
use ieee.std_logic_1164.all;
entity parity10 is
 port (D: in std_logic_vector(0 to 9); ODD:out std_logic);
end parity10;

architecture beh of parity10 is
begin   -- example of an 10-bit parity generator
process(D)
  variable pargen: boolean;
 begin
 pargen := false;
 for i in 0 to D'length -1 loop -- implicit "i"
    if D(I)='1' then pargen:= not pargen;
    end if;
 end loop;
 if pargen then ODD<='1';
 else           ODD<='0';
 end if;
 end process;
end beh;
```



Say D= "0101110100"

	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
⇒	0	0	1	0	1	1	1	0	1	0
⇒ pargen=	T	T	T	F	F	T	F	T	T	F

⇒ ODD = 1

Behavioral.. equivalent of previous program..

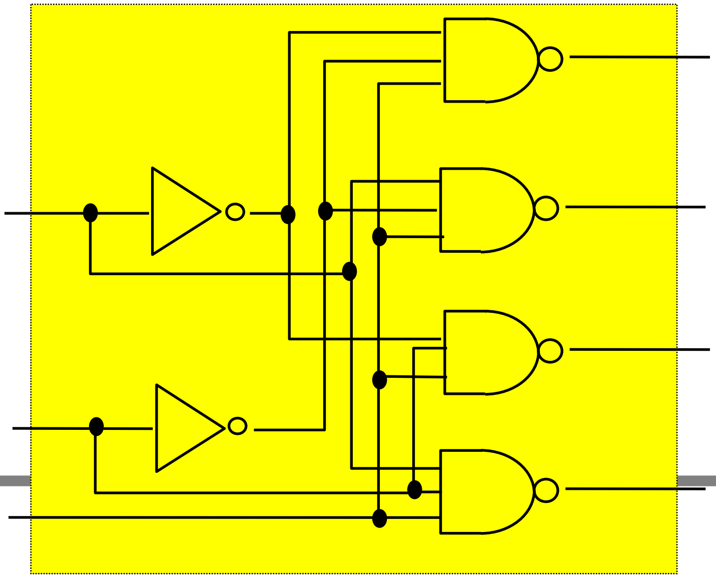
```
library ieee;
use ieee.std_logic_1164.all;
entity parity10 is
 port (D: in std_logic_vector(0 to 9); ODD:out std_logic);
end parity10;
architecture beh of parity10 is
begin   -- example of an 10-bit parity generator
process(D)
  variable pargen: boolean;
 begin
 pargen := false;
    if D(0)='1' then pargen:= not pargen; end if;
    if D(1)='1' then pargen:= not pargen; end if;
    if D(2)='1' then pargen:= not pargen; end if;
    if D(3)='1' then pargen:= not pargen; end if;
    if D(4)='1' then pargen:= not pargen; end if;
    if D(5)='1' then pargen:= not pargen; end if;
    if D(6)='1' then pargen:= not pargen; end if;
    if D(7)='1' then pargen:= not pargen; end if;
    if D(8)='1' then pargen:= not pargen; end if;
    if D(9)='1' then pargen:= not pargen; end if;
     if pargen then ODD<='1';
     else           ODD<='0';
     end if;
 end process;
end beh;
```

loop just repeats instructions!

DATAFLOW MODELING

Dataflow VHDL Descriptions:

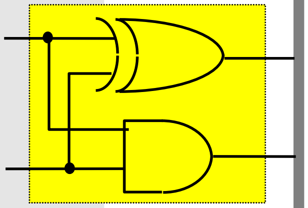
- circuit is described in terms of how *data flows* through circuit
- described using concurrent signal assignment statements
  - ordering not important
  - *executed only* ***when*** a signal in RHS expression has an *event*
- usually used to describe combinational logic which can be simplified using logic synthesis tools



Dataflow Style of Modeling

A *dataflow* representation for the HALF_ADDER **entity is described by two concurrent signal assignment stmts:**

```
entity HALF_ADDER is
port ( A , B     : in std_logic;
         SUM,CARRY:out std_logic);
end HALF_ADDER;

architecture HA_CONCURRENT of HALF_ADDER is
begin
  SUM <= A xor B;
  CARRY <= A and B;
end HA_CONCURRENT;
```



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

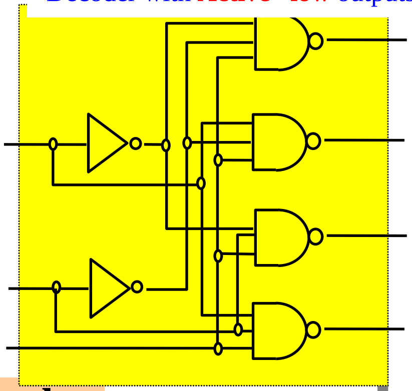
If A *or* B has an event at simulation time T, RHS expressions are evaluated and at time=T+Δns: CARRY ⇐ *new value*, and at T+Δns: SUM ⇐ *new value*

A *Dataflow* representation for DECODER2x4 entity

```
entity DECODER2x4 is
  port (A,B,ENABLE: in std_logic;
        Z: out std_logic_vector(0 to 3));
end DECODER2x4;

architecture DEC_DATAFLOW of DECODER2x4 is
  signal ABAR, BBAR: STD_LOGIC; -- internal signals
begin
  Z(3) <= not (A and B and ENABLE);      -- stmt1
  Z(0) <= not(ABAR and BBAR and ENABLE); -- stmt2
  BBAR <= not B;                         -- stmt3
  Z(2) <= not(A and BBAR and ENABLE);    -- stmt4
  ABAR <= not A;                         -- stmt5
  Z(1) <= not(ABAR and B and ENABLE);    -- stmt6
end DEC_DATAFLOW:
```

A	B	ENABLE	Z
L	L	L	HHHH
L	H	L	HHHH
H	L	L	HHHH
H	H	L	HHHH
L	L	H	HHHL
L	H	H	HHLH
H	L	H	HLHH
H	H	H	LHHH

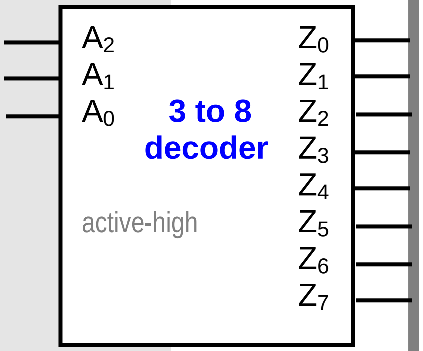
Decoder with **Active –low** outputs



if B changes at T, stmts 1,3,6 are triggered, and new values assigned to Z(3), BBAR, Z(1) at T+Δ ⇒ stmts 2,4 are triggered and new values assigned to Z(0), Z(2) at T+2Δ.

A *Dataflow* representation for 3to8 DECODER

```
entity DECODER_3TO8 is
      port (A0, A1, A2: in std_logic;
            Z0,Z1,Z2,Z3,Z4,Z5,Z6,Z7: out std_logic);
 end DECODER_3TO8;
 -- architecture based on dataflow style of modeling
 architecture DECODER_DFLOW of DECODER_3TO8 is
 begin
    Z0 <= ( not(A2) and not(A1) ) and not(A0); -- s0
    Z1 <= ( not(A2) and not(A1) ) and     A0 ; -- s1
    Z2 <= ( not(A2) and    A1   ) and not(A0); -- s2
    Z3 <= ( not(A2) and    A1   ) and     A0 ; -- s3
    Z4 <= (     A2  and not(A1) ) and not(A0); -- s4
    Z5 <= (     A2  and not(A1) ) and     A0 ; -- s5
    Z6 <= (     A2  and    A1   ) and not(A0); -- s6
    Z7 <= (     A2  and    A1   ) and     A0 ; -- s7
 end DECODER_DFLOW;
```



Concurrent statements:

- ordering in an architecture body is not important
- executed whenever ***events occur on signals*** (event-triggered) in RHS expression.

Combinational Logic Circuits can be easily & directly implemented using concurrent statements without need for logic simplification.

Dataflow **Modeling..** more concurrent statements

Two concurrent signal assignment statement forms:

- **conditional signal assignment statement**
- **selected signal assignment statement**

CANNOT use in a process!

Dataflow: Conditional signal assignment

Selects different values for target signal based on specified conditions:

```
target_signal <= [waveform_elements when condition else]
                  waveform_elements when condition else]
                                 ...
                  waveform_elements [when condition]
```

statement executed when an event occurs on a signal in *any* waveform element or *any* condition by evaluating *each* condition one at a time; value(s) of *first true condition* found, is scheduled to target signal

```
Z <=  IN0 when S0='0' and S1='0' else
      IN1 when S0='1' and S1='0' else
      IN2 when S0='0' and S1='1' else
      IN3;
```

-- executed when an event occurs on signals IN0, IN1, IN2, IN3, S0 *or* S1;
-- *1st condition* is checked; if false *2nd condition* is checked; etc
-- If S0='0' and S1='1', value of IN2 is scheduled to be assigned to signal Z

```
process (IN0, IN1, IN2, IN3, S0, S1)
begin
  if S0 = '0' and S1 = '0' then
      Z <= IN0;
  elsif S0 = '1' and S1 = '0' then
       Z<= IN1;
  elsif S0 = '0' and S1 = '1' then
      Z <= IN2;
  else Z <= IN3;
 end if;
end process;
```

Dataflow: Selected Signal assignment

Selects different values for a target signal based on value of a select expression (like a `case` statement):

```
with select_expression select
 target_signal <=   waveform_elements when choices,
                    waveform_elements when choices,
                                          . . .
                    waveform_elements when others;
```

Values not covered explicitly →

statement executed when an event occurs on a signal in select expression or *any* signal used in any waveform element; selected waveform value(s) that matches specified choice, is scheduled to be assigned to target signal.

```
type OP is (op1, op2, op3, op4);
signal OP_CODE: OP;
       .....
with OP_CODE select
 Z <=   A and B when op1,
        A  or B when op2,
        A xor B when op3,
        A nor B when op4;  -- or use "others"
-- statement executed when an event occurs on OP_CODE, A, or B
-- if OP_CODE = op2: "AorB" is scheduled to be assigned to Z.
```

```
process (OP_CODE, A, B)
begin
 case OP_CODE is
      when op1 => Z <= A and B;
      when op2 => Z <= A  or B;
      when op3 => Z <= A xor B;
      when op4 => Z <= A nor B;
  end case;
end process;
```

Dataflow:Selected Assignment.. *important notes*

- must include all possible conditions in a selected assignment
- selected expressions may include ranges and multiple values:

```
with Address select  --16-bit number eg "0000 0001 0100 1111" or 0x"014F"
  CS <= "001" when 0x"0000" to 0x"7FFF",
        "010" when 0x"8000" to 0x"82FF",
        "100" when 0x"8300" to 0x"83FF",
        "000" when others;  -- '0x' refers to hexadecimal
```

- can specify that outputs do not change under some conditions:

```
 type STATE_TYPE is
      (RESET, APPLY, WAITS, HOLD, RECEIVE);
 signal NEXT_STATE: STATE_TYPE;
   ....
 with NEXT_STATE select
 ZRX <="0001" when APPLY,
       "0010" when WAITS,
       "0100" when RESET,
       unaffected when others;
 -- when NEXT_STATE has value HOLD or RECEIVE, value unaffected
 -- is assigned to signal ZRX,  so ZRX retains its old value
```

Dataflow: Conditional vs Selected Signal Assignment

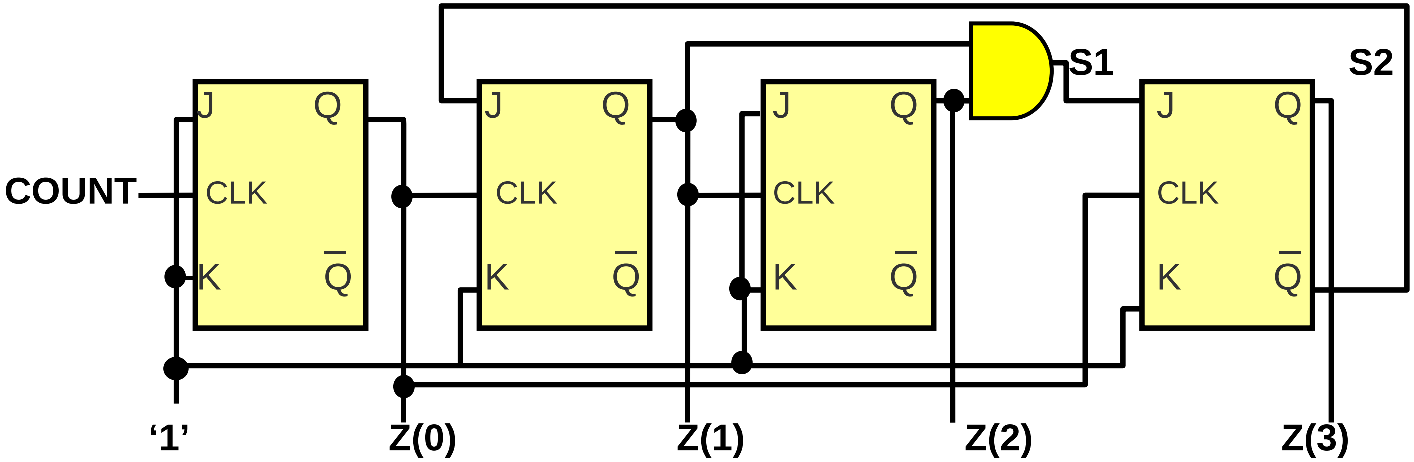
Conditional always enforces priority on the conditions:

```
 Q1 <= "01" when A='1' else
       "10" when B='1' else
       "11" when C='1' else
       "00" ;
                -- is identical to
with std_logic_vector'(A,B,C) select
  Q2 <= "01" when "100",
        "01" when "101",
        "01" when "110",
        "01" when "111",
        "10" when "010",
        "10" when "011",
        "11" when "001",
        "00" when others;

-- "A" takes priority and this is implied in conditional assignment
-- but all possible conditions must be specified for selected assignment
```

STRUCTURAL MODELING

Structural VHDL Descriptions:

- circuit described as *interconnected components* ⇒ HIERARCHY
- Structure can be very low level descriptions (such as transistors) to very high level descriptions (such as block diagrams)
- use of components enables the re-use elements of the design

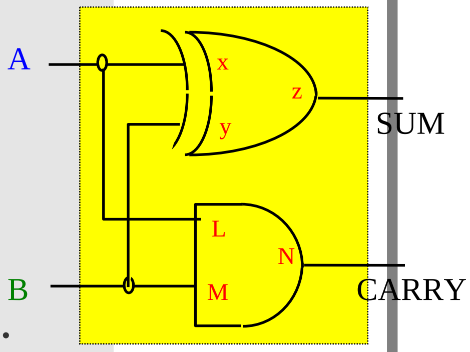


- Configuration links models to components:
  - selects one of many architecture bodies of an entity
  - binds components in a design library to entities

Structural Style of Modeling

- `component` statement is just a **template**, does not specify its behavior
- library of standard components are provided with VHDL complier
- there's an **`entity-architecture`** pair associated with each component!

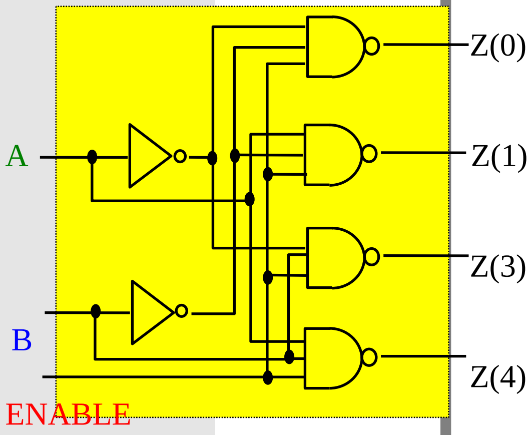
```
entity HALF_ADDER is   -- same as before
 port (A,B:in std_logic;
       SUM,CARRY: out std_logic);
end HALF_ADDER;

architecture HA_STRUCTURE of HALF_ADDER is
          -- before begin: component declarations
 component XOR2
  port (X,Y: in std_logic; Z:out std_logic);
 end component;
 component AND2
  port (L,M: in std_logic; N:out std_logic);
 end component;
  begin  -- below we have component instantiation statements; one per component..
     X1: XOR2 port map (A, B, SUM);
     A1: AND2 port map (A, B, CARRY);
  end HA_STRUCTURE;

 -- X1 (lable): ports A, B, SUM of entity HALF_ADDER connected to ports X, Y, Z
 --      respectively of XOR2 component (positional association)
 -- A1: ports A, B, CARRY connected to ports L, M, N of entity AND2
```



***component instantiation statements* (concurrent)** ⇒ **order *not* important**

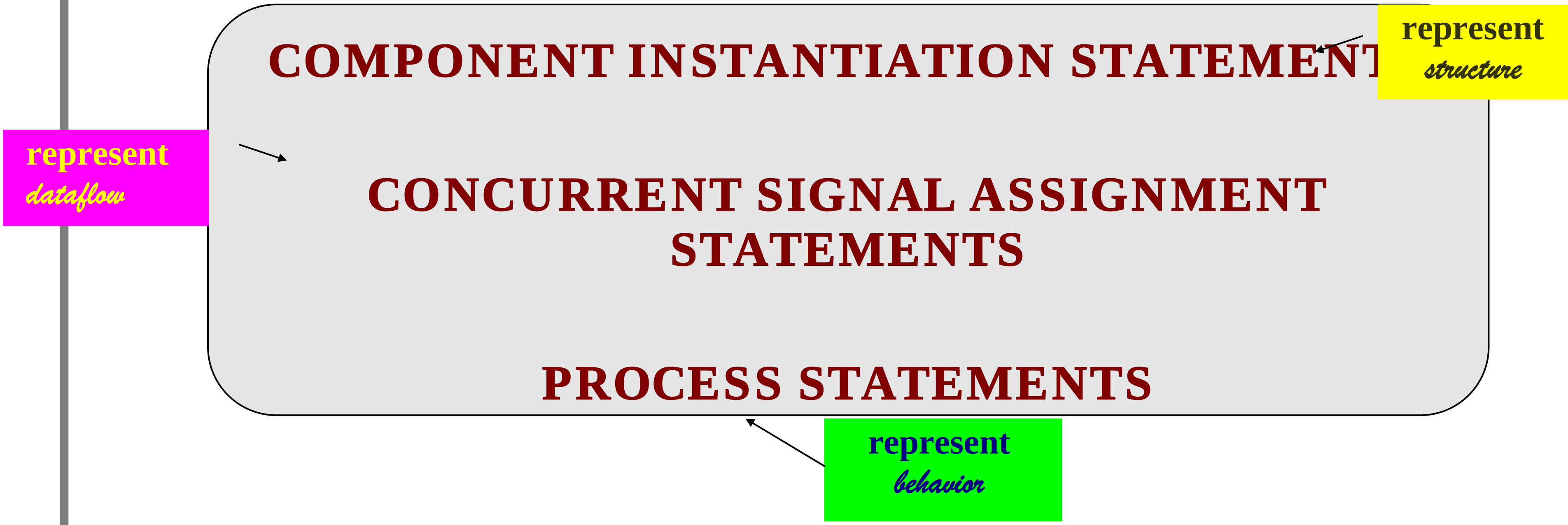
Structural representation for DECODER2x4 entity

```
entity DECODER2x4 is
 port (A,B,ENABLE:in std_logic;Z:out std_logic_vector(0to 3));
end DECODER2x4;

architecture DEC_STR of DECODER2x4 is
 component INV
   port (PIN: in std_logic; POUT: out std_logic);
 end component;
 component NAND3
   port (D0,D1,D2:in std_logic; DZ:out std_logic);
 end component;
signal ABAR, BBAR: std_logic;
begin
 V0: INV     port map (A, ABAR); -- inverter
 V1: INV     port map (B, BBAR);
 N0: NAND3   port map (ENABLE,ABAR,BBAR,Z(0)); -- NAND
 N1: NAND3   port map (ABAR, B, ENABLE, Z(1));
 N2: NAND3   port map (A, BBAR, ENABLE, Z(2));
 N3: NAND3   port map (A, B,    ENABLE, Z(3));
end DEC_STR;

-- signals ABAR, BBAR (not visible outside DEC_STR) connect components
--  within decoder
```



MIXED STYLE OF MODELING

Three modeling styles mixed in a single architecture body:



COMPONENT INSTANTIATION STATEMENT

represent *structure*

represent *dataflow*

CONCURRENT SIGNAL ASSIGNMENT STATEMENTS

PROCESS STATEMENTS

represent *behavior*

A mixed style model for one-bit full-adder

```
use ieee.std_logic_1164.ALL;
entity FULL ADDER is     - - full adders use carry bit from the previous stage of addition
  port (A, B, CIN: in std_logic; SUM, COUT: out std_logic);
end FULL_ADDER;          - - FULL_ADDER  has 3 concurrent stmt

architecture FA_MIXED of FULL_ADDER is
  component XOR2 port (P1,P2:in std_logic;PZ:out std_logic);
  end component;
  signal S1: std_logic;
begin
 X1: XOR2 port map (A, B, S1);    - - component instantiation stmt
 process (A, B, CIN)              - - process stmt
 variable T1, T2, T3: std_logic;
 begin
  T1  := A and B;
  T2  := B and CIN;
  T3  := A and CIN;
  COUT <= T1 or T2 or T3;
 end process;
 SUM <= S1 xor CIN;         -- concurrent signal assignment stmt
end FA_MIXED;
```

A	B	CIN	SUM	COUT
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Truth table of full adder.

BASIC VHDL FOR DIGITAL SYSTEMS DESIGN

(Self Reading)

How is VHDL used…..

DESIGN SPECIFICATION

VHDL allows capture performance/interface requirements of each component in a large system

↓

DESIGN ENTRY

details of design are entered as VHDL descriptions

↓

FUNCTIONAL SIMULATION

VHDL descriptions are compiled and simulated by applying the test bench (not detailed timing simulation).

↓

DETAILED TIMING SIMULATION

After synthesis of the VHDL descriptions into some low level technology specific representaion (CPLD, FPGA, ASIC), the design can be simulated again using detailed timing data generated as a result of the implementation.

↓

DESIGN DOCUMENTATION

VHDL 's structured programming features make it natural to be to document large and complex circuits (no wonder DoD requires VHDL as standard format)

What about VERILOG and PLD Languages?

VERILOG:

- **Availability of VERILOG simulation models paved the way for its early eminence in ASIC simulation libraries.**
- **IEEE Standard 1076.4 VITAL overcomes this shortcoming of VHDL**
- **Adoption of VHDL as IEEE Standard and its design management features makes VHDL ideal for large projects.**
- **VHDL & VERILOG are similar; both are easy to learn but hard to master. Once you have learned one, its easy to transition to the other.**

PLD Languages (ABEL, PALASM, etc):

- **VHDL is general purpose simulation modeling language**
- **PLD oriented languages were developed as specialized languages for capture and synthesis of relatively small digital circuits.**
- **VHDL allows for sophisticated programs for simulation (test bench)**

Major Capabilities of *VHDL*

- ***supports hierarchy*: a digital system can be modeled as a set of interconnected components where each component can be modeled as a set of interconnected sub-components.**

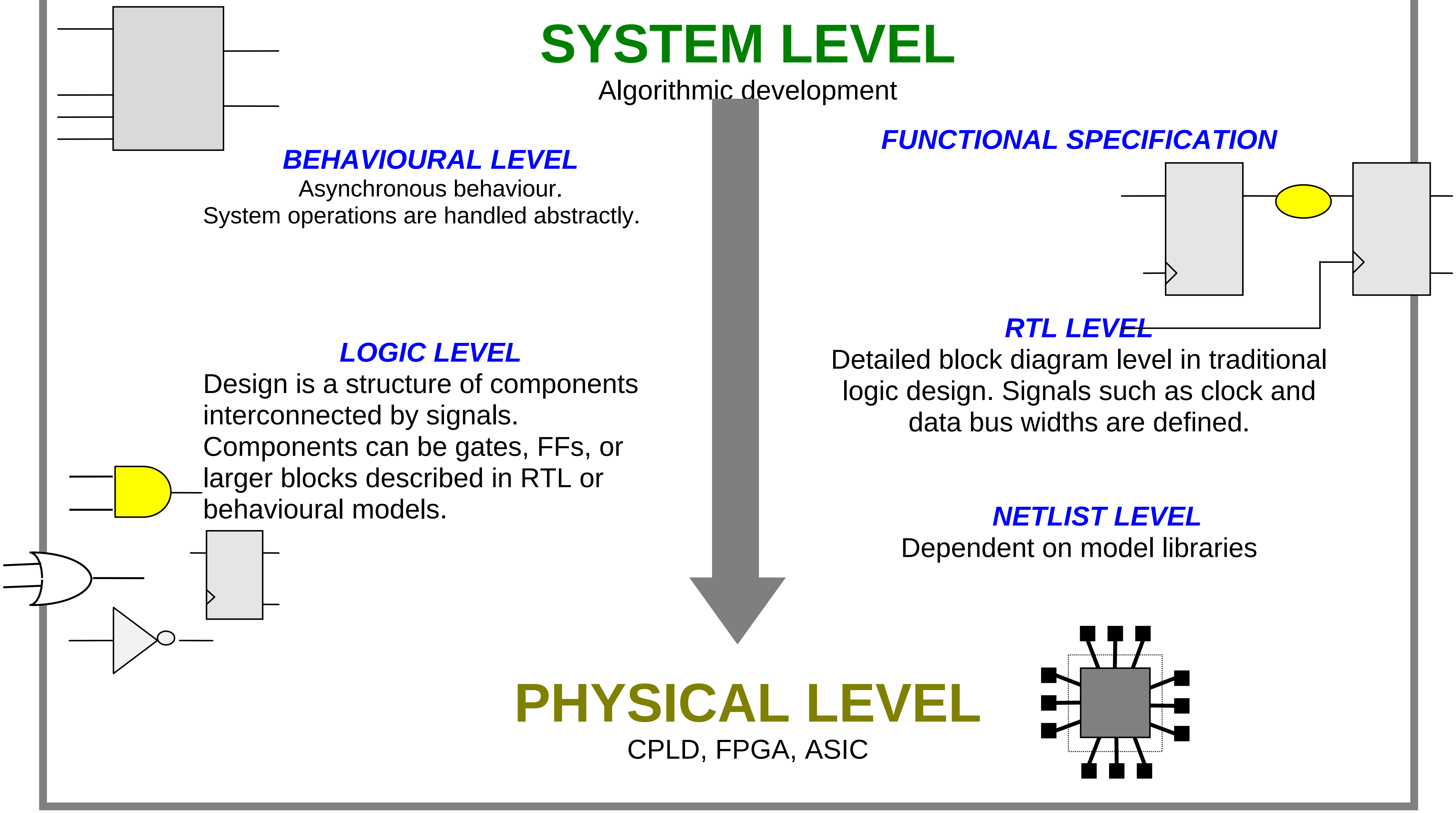
- ***supports flexible design methodologies*: top-down, bottom-up, or mixed.**

- ***not technology-specific, supports various technologies* ⇒ same model can be synthesized into different vendor libraries**

- ***supports both synchronous & asynchronous timing models***

- **models various digital modeling techniques, such as *finite-state machine descriptions*, *algorithmic descriptions*, *Boolean equations***

Levels of Abstraction



SYSTEM LEVEL

Algorithmic development

BEHAVIOURAL LEVEL

Asynchronous behaviour.
System operations are handled abstractly.

FUNCTIONAL SPECIFICATION

RTL LEVEL

Detailed block diagram level in traditional logic design. Signals such as clock and data bus widths are defined.

LOGIC LEVEL

Design is a structure of components interconnected by signals. Components can be gates, FFs, or larger blocks described in RTL or behavioural models.

NETLIST LEVEL

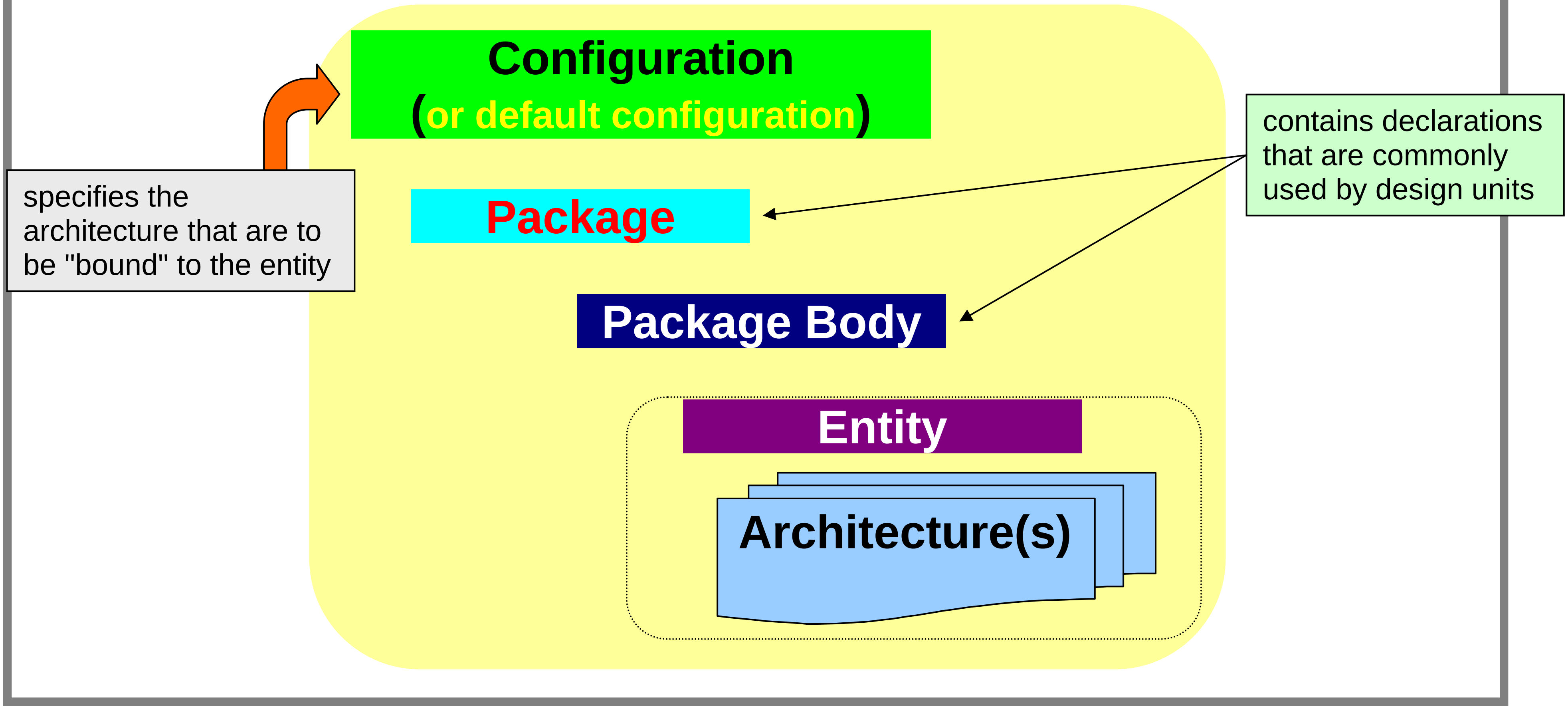
Dependent on model libraries

PHYSICAL LEVEL

CPLD, FPGA, ASIC

VHDL DESIGN UNITS

VHDL design units are segments that can be separately compiled and stored in a library:


Configuration
(or default configuration)
Package
Package Body
Entity
Architecture(s)
specifies the architecture that are to be "bound" to the entity
contains declarations that are commonly used by design units

BASIC ELEMENTS OF VHDL

(Self-Reading)

VHDL Identifiers

BASIC IDENTIFIERS:

- **letter, digit or underscore:** `A..Z, a..z, 0..9,_`
- **first must be a letter & last cannot be an underscore**
- **lower & upper-case letters are *identical* (case insensitive)**
- **two underscores cannot appear consecutively**
- **examples:** `DRIVE_BUS SelSignal SET_CK_HIGH`

EXTENDED IDENTIFIERS:

- **sequence of *any* characters between backslashes**
- **lower & uppercase letters are *distinct***
- **examples:** `\-25\  \-Q\  \.~$*****\ \74oOTTL\`

```
\process\     -- Distinct from keyword process
\----\\----\  --  two consecutive backslashes represents one backslash
```

VHDL Literals

we will use mainly standard logic & ranged integers!

LITERAL	EXPLANATION	EXAMPLES
bit	*two* discreet character literals '0' (*default*) or '1'	'0' '1' *only*
bit_vector	array of bits enclosed in double quotes; octal, hexadecimal, or binary (*default*) representations	"0011_0011" x"00FF" BIT_VECTOR' ("10")
IEEE Standard logic	*nine* signal strengths useful for logic simulation	'U' 'X' '0' '1' 'Z' 'W' 'L' 'H' '-'
boolean	*two* discreet values: *false* (*default*) or *true*	TRUE , FALSE
real	+ve or -ve numbers from -1.0E+38 to +1.0E+38 (with *decimal point*) default: *most negative number*	16.2, 5.0E+2 3_1.4_2 62.3E-2
integer	discreet values, range usually –(2^{31} - 1) to +(2^{31} - 1) implementation-dependent; should have *range constraint*	+1 862 -257 6E2 +123_469 16#00FF#

VHDL Data Objects: *Constant*, *Variable*, *Signal*

CONSTANT: value is assigned *before* simulation starts and cannot be changed during simulation. ***Declarations:***

```
constant BUS_WIDTH: INTEGER := 8;
```

VARIABLE: different values can be assigned at different times using *variable assignment statement*. ***Declarations:***

```
variable SUM: INTEGER range 0 to 100 := 10;
variable CTRL: STD_LOGIC_VECTOR(10 downto 0);
```

SIGNAL: holds *current* and *set of future values* assigned using *signal assignment statement*. ***Signal declarations:***

```
 signal CLOCK: STD_LOGIC;
 signal DATA_BUS: STD_LOGIC_VECTOR(0 to 7);
 signal A,B : STD_LOGIC_VECTOR(0 to 4);
```

VHDL Data Types

- **Each data type has a set of *values* & *operations*. E.g.** `integer`**: set of integers with predefined operators: +, –, /, ***

- ***New types* defined using type declarations and operations on these types defined by functions**

```
type DIGIT is (‘0‘,'1‘,'2‘,'3‘,'4‘,'5‘,'6‘,'7‘,‘8‘,'9');
type MICRO_OP is (LOAD,STORE,ADD,SUB,MUL,DIV);
```

- ***Sub-types* are derived from *base type* with *range constraint* and have *same* set of operations as base type:**

```
 subtype MY_INTEGER is INTEGER range 48 to 156;
 subtype MIDDLE is DIGIT range '3' to '7';
 subtype ARITH_OP is MICRO_OP range ADD to DIV;
```

VHDL … more on bit_vectors

Can be represented as octal, binary or hexadecimal

```
signal RX_BUS: STD_LOGIC_VECTOR(0 to 5) := O"37";
      -- X for hexadecimal eg X"FF0"
      -- B for binary eg B"00_0011_1101"
      -- O for octal eg O"327"
```

Assignment modes:

```
variable OP_CODES: STD_LOGIC_VECTOR(1 to 5);
OP_CODES := "01001";                           -- string literal
OP_CODES := ('0', '1', '0', '0', '1');          -- positional association
OP_CODES := (2=>'1', 5=>'1', others=>'0');     -- named association
OP_CODES :=  (others=>'0');                     -- all values set to '0'
```

```
signal Z_BUS: STD_LOGIC_VECTOR(3 downto 0);
signal A_BIT, B_BIT, C_BIT, D_BIT: bit;

Z_BUS <= A_BIT & B_BIT & C_BIT & D_BIT; -- concatenation
Z_BUS <= (A_BIT, B_BIT, C_BIT, D_BIT);  -- aggregates
(A_BIT, B_BIT, C_BIT, D_BIT) <=“0101“;   -- aggregates
```

OPERATORS

Categories of predefined operators (*in increasing precedence*):

- Logical operators: `and or nand nor xor xnor not`
- Relational operators: `=, /= , <=, >=, >, <`
- Shift operators:
- Adding operators
- Multiplying operators
- Miscellaneous operators

Operators in same category have the *same precedence* and evaluation is from *left to right*.

***Parentheses* may be used to override left to right evaluation.**

Logical operators: and or nand nor xor xnor not

- **For** STANDARD_LOGIC, BIT **&** BOOLEAN**, and their 1-D arrays**
- **bit values** '0' **&** '1' **treated as** **FALSE** **&** **TRUE** **of** BOOLEAN **type when applied with logic operators**
- not **is a** ***unary*** **logical operator**
- nand**,** nor ***not associative*****;** ⇒ **sequence of** nand ***or*** nor **is illegal** (avoided using *parentheses*). **eg.** A nand B nand C -- illegal

Relational Operators: `=, /= , <=, >=, >, <`

- For ANY operand type
- `BOOLEAN` is the result type for relational operations
- `<=, >=, >, <` operators: comparison is performed *one element at time*, left to right.

Key attributes for data types (T) or signals (S)

ATTRIBUTE	EXPLANATION	EXAMPLES

T'LEFT	Returns left value specified in type declaration	`INTEGER'LEFT is -2147483647`
		`BIT'LEFT is '0'`

T'RIGHT	Returns right value specified in type declaration	`INTEGER'RIGHT is 2147483647`
		`BIT'RIGHT is '1'`

T'HIGH	Returns largest value specified in declaration	`TYPE bit8 is 255 downto 0`
		`bit8'HIGH is 255`

T'LOW	Returns smallest value specified in declaration	`TYPE bit8 is 255 downto 0`
		`bit8'LOW is 0`

T'LENGTH	Returns the number of bits of T	`D: std logic vector(0 to 9)`
		`D'Length  is 9`

S'EVENT	Returns true when an event has occurred for **signal** S	`clock: std logic` clock'EVENT : is **TRUE** only when clock changes (low•high OR high•low)