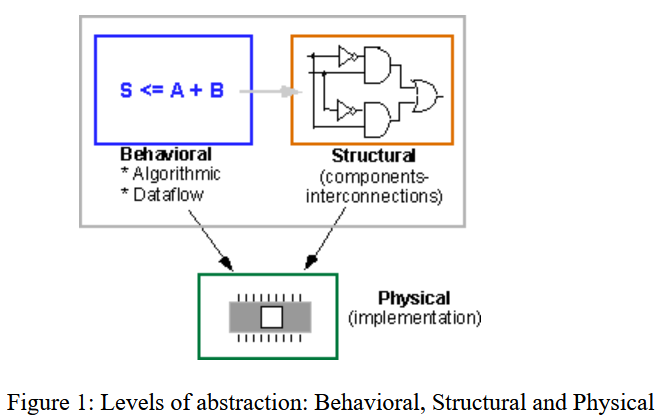
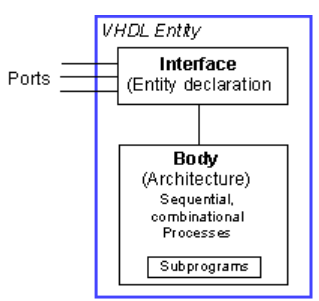
* It is being used for documentation, verification, and synthesis of large digital designs. This is actually one of the key features of VHDL, since the same VHDL code can theoretically achieve all three of these goals, thus saving a lot of effort.
* Three different approaches to describing hardware. These three different approaches are the structural, data flow, and behavioral methods of hardware description.



**Basic Structure of a VHDL File**

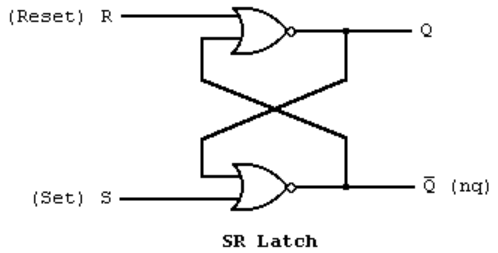
* A digital system in VHDL consists of a design **entity** that can contain other entities that are then considered components of the top-level entity
* Each entity is modeled by an *entity declaration* and an *architecture body*
* entity declaration ~ interface to the outside world that defines the input and output signals
* architecture body ~ description of the entity
  + composed of interconnected entities, processes and components, all operating concurrently
* many such entities are connected together to perform the desired function



* Keywords and user-defined identifiers are **case insensitive**
* **strongly typed** language => one has always to declare the type of every object that can have a value, such as signals, constants and variables.

**Building Blocks**

* Every portion of a VHDL design is considered a block.
* Each block in VHDL is analogous to an off-the-shelf part and is called an entity.



* Inputs and outputs to a file are defined in an **entity**
  + The *entity* describes the interface to that block and a separate part associated with the entity describes how that block operates.
  + contains a port that defines all inputs and outputs to a file.

**entity** NAME\_OF\_ENTITY **is** [ **generic** *generic\_declarations*);]

**port** (*signal\_names*: **mode** *type*;

*signal\_names*: **mode** *type*;

:

*signal\_names*: **mode** *type*);

**end** [NAME\_OF\_ENTITY] ;

* mode: in/out/buffer/inout
* type: bit, bit\_vector, Boolean, character, std\_logic, and std\_ulogic
* entity latch is  
    port (s,r: in bit; -- *port clause* with *interface declarations* all but last one followed by semicolon  
          q,nq: out bit);  
  end latch;

* An architecture is used to describe the functionality of a particular entity.
  + All signals that are used by the architecture must be defined between the “is” and the “begin” keywords
  + actual architecture logic comes between the “begin” and the “end” keywords.

**architecture** architecture\_name **of** NAME\_OF\_ENTITY **is**

**--** Declarations

-- components declarations

-- signal declarations

-- constant declarations

-- function declarations

-- procedure declarations

-- type declarations

 :

**begin**

-- Statements

:

**end** architecture\_name;

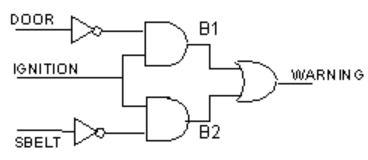
* **Structural Description:**
  + describes a system as a collection of gates and components that are interconnected to perform a desired function
  + architecture structure of latch is  
      component nor\_gate -- component declaration  
        port (a,b: in bit;  
              c: out bit);   
      end component;  
    begin  
     n1: nor\_gate -- n1 = instance name  
        port map (r,nq,q); -- connection (portmap clause)  
      n2: nor\_gate  
        port map (s,q,nq);  
    end structure
  + structural description of a design is simply a textual description of a schematic
  + one of many means of specifying netlists

* entity: describes a design interface

component: describes the interface of an entity that will be used as an instance (or a sub-block)

component instance: is a distinct copy of the component that has been connected to other parts and signals

* The *port map clause* specifies what signals of the design to connect to the interface of the component in the same order as they are listed in the component declaration.
* Eg:



**architecture** structural **of** BUZZER **is**

-- Declarations

**component** AND2

**port** (in1, in2: **in** std\_logic;

      out1: **out** std\_logic);

**end component**;

**component** OR2

**port** (in1, in2: **in** std\_logic;

      out1: **out** std\_logic);

**end component**;

**component** NOT1

**port** (in1: **in** std\_logic;

      out1: **out** std\_logic);

**end component**;

-- declaration of signals used to interconnect gates

**signal** DOOR\_NOT, SBELT\_NOT, B1, B2: std\_logic;

**begin**

-- Component instantiations statements

U0: NOT1 **port map** (DOOR, DOOR\_NOT);

U1: NOT1 **port map** (SBELT, SBELT\_NOT);

U2: AND2 **port map** (IGNITION, DOOR\_NOT, B1);

U3: AND2 **port map** (IGNITION, SBELT\_NOT, B2);

U4: OR2  **port map** (B1, B2, WARNING);

**end** structural;

* Alternate way of instantiation:

*label: component-name* **port map** (*port1*=>*signal1, port2*=> *signal2,… port3*=>*signaln*);

U0: NOT1 **port map** (in1 => DOOR, out1 => DOOR\_NOT);

U1: NOT1 **port map** (in1 => SBELT, out1 => SBELT\_NOT);

U2: AND2 **port map** (in1 => IGNITION, in2 => DOOR\_NOT, out1 => B1);

U3: AND2 **port map** (in1 => IGNITION, in2 => SBELT\_NOT, B2);

U4: OR2  **port map** (in1 => B1, in2 => B2, out1 => WARNING);

* component instantiation statement creates a new level of hierarchy
* Good for creating hierarchical designs, eg:

-- Example of a four bit adder

**library**  ieee;

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

-- definition of a full adder

**entity** FULLADDER **is**

**port** (a, b, c: **in** std\_logic;

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

**end** FULLADDER;

**architecture** fulladder\_behav **of** FULLADDER **is**

**begin**

sum <= (a **xor** b) **xor c** ;

carry <= (a **and** b) **or** (c **and** (a **xor** b));

**end** fulladder\_behav;

-- 4-bit adder

**library**  ieee;

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

**entity** FOURBITADD **is**

**port** (a, b: **in** std\_logic\_vector(3 **downto** 0);

Cin : **in** std\_logic;

sum: **out** std\_logic\_vector (3 **downto** 0);

Cout, V: **out** std\_logic);

**end** FOURBITADD;

**architecture** fouradder\_structure **of** FOURBITADD **is**

**signal** c: std\_logic\_vector (4 **downto** 0);

**component** FULLADDER

**port**(a, b, c: **in** std\_logic;

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

**end** component;

**begin**

FA0: FULLADDER

**port map** (a(0), b(0), Cin, sum(0), c(1));

FA1: FULLADDER

**port map** (a(1), b(1), C(1), sum(1), c(2));

FA2: FULLADDER

**port map** (a(2), b(2), C(2), sum(2), c(3));

FA3: FULLADDER

**port map** (a(3), b(3), C(3), sum(3), c(4));

V <= c(3) **xor** c(4);

Cout <= c(4);

**end** fouradder\_structure

* VHDL does not allow the  use of outputs as internal signals (see Cout above)
* **Data Flow Description**
  + describe how signals (data) flow through the circuit.
  + entity latch is  
      port (s,r : in bit;  
            q,nq : out bit);  
    end latch;

architecture dataflow of latch is  
begin  
  q<=r nor nq;  
  nq<=s nor q;  
end dataflow;

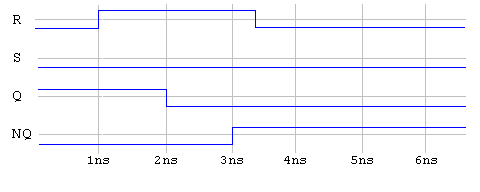
* The signal assignment operator in VHDL (<=) specifies a relationship between signals, not a transfer of data as in programming langauges
  + describes how data flows from the signals on the right side of the <= operator to the signal on the left side
  + right side of the <= operator is called an expression
* The scheme used to model a VHDL design is called *discrete event time simulation*
* When the value of a signal changes, we say an *event* has occurred on that signal
* The values of signals are only updated when certain events occur and events occur at discrete instances of time.
* Since one event causes another, simulation proceeds in rounds
* signal on the left side of the <= operator depends on all the signals appearing on the right side (*q* depends on *r* and *nq)*
* if an event occurs on *r* or *nq*, then the *nor* operator is evaluated, and if the result is different than the current value of *q*, then an event will be scheduled to update *q*.
* Functional Simulation (functional = without timing considerations):

start  : r='0',s='0',q='1',nq='0'  
round 1: r='1',s='0',q='1',nq='0', The value '0' is scheduled on q.  
round 2: r='1',s='0',q='0',nq='0', The value '1' is scheduled on nq.  
round 3: r='1',s='0',q='0',nq='1', No new events are scheduled.  
round 4: r='0',s='0',q='0',nq='1', No new events are scheduled.

* Timing simulation: models internal delays
* **Inertial delay model**: using an *after clause:*

q<=r nor nq after 1ns;  
nq<=s nor q after 1ns;

* simulator must maintain a current time value



**Data Object Classes**

1. **Signal**
   * **declared *outside* the process**
   * **Declaration:  
     signal** ***list\_of\_signal\_names*: type [ := initial value] ;**
   * **Eg:  
     signal** **SUM, CARRY: std\_logic;**

**signal** CLOCK: bit;

**signal** TRIGGER: integer :=0;

**signal** DATA\_BUS: bit\_vector (0 to 7);

**signal** VALUE: integer **range** 0 **to** 100;

* Updated when signal assignments are executed, *after a certain delay,* eg:  
  SUM <= (A **xor** B) **after** 2 ns;
  + If no delay is specified, the signal will be updated after a *delta* delay
  + Multiple events:  
    **signal** wavefrm : std\_logic;

wavefrm <= ‘0’, ‘1’ after 5ns, ‘0’ after 10ns, ‘1’ after 20 ns;

* *Attributes*
  + Used to obtain auxiliary info of signals
  + Eg:  
    x`last\_value -- previous value of signal x

1. **Variable**
   * **Has a single value**
   * **can be updated using a variable assignment statement**
     + **updated without any delay as soon as the statement is executed.**
   * **must be declared *inside* a process (and are local to the process).**
   * **Declaration:  
     variable** ***list\_of\_variable\_names*: type [ := initial value] ;**
   * **do not cause or have events**
   * **Updated using variable assignment:  
     Variable\_name := expression;  
       
     Variable assignment may only occur in processes**
   * **Eg:  
     count: process (x)  
       variable cnt : integer := -1; -- variable declaration with initialization (before begin)  
     begin  
      cnt:=cnt+1;  
     end process;**
   * **Eg:**

**variable** CNTR\_BIT: bit :=0;

**variable** VAR1: boolean :=FALSE;

**variable** SUM: integer **range** 0 **to** 256 :=16;

**variable** STS\_BIT: bit\_vector (7 **downto** 0);

1. **Constant**
   * **can have a single value of a given type and cannot be changed during the simulation**
   * **Declaration:  
     constant** ***list*\_*of*\_*name\_of\_constant*: type [ := initial value] ;**
   * **can be declared at the start of an architecture and can then be used anywhere within the architecture**
   * **Constants declared within a process can only be used inside that specific** [**process**](https://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html#Process)
   * **Eg:  
     constant**  **RISE\_FALL\_TME: time := 2 ns;**

**constant**  DELAY1: time := 4 ns;

**constant**  RISE\_TIME, FALL\_TIME: time:= 1 ns;

**constant**  DATA\_BUS: integer:= 16;

1. **File**

**Signals vs Variables**

* **Eg 1 (variables)**

**architecture** VAR **of** EXAMPLE is

**signal** TRIGGER, RESULT: integer := 0;

**begin**

**process**

**variable** variable1: integer :=1;

**variable** variable2: integer :=2;

**variable** variable3: integer :=3;

**begin**

**wait on** TRIGGER;

variable1 := variable2;

variable2 := variable1 + variable3;

variable3 := variable2;

RESULT <= variable1 + variable2 + variable3;

**end process**;

**end** VAR

* “variable1, variable2 and variable3” are computed sequentially
* values updated instantaneously after the TRIGGER signal arrives
* RESULT, which is a signal, is computed using the new values of the variables and updated a time *delta* after TRIGGER arrives
* variable1 = 2, variable2 = 5 (=2+3), variable3= 5, RESULT = 12

* **Eg 2 (signals)**

**architecture** SIGN **of** EXAMPLE is

**signal** TRIGGER, RESULT: integer := 0;

**signal** signal1: integer :=1;

**signal** signal2: integer :=2;

**signal** signal3: integer :=3;

**begin**

**process**

**begin**

**wait on** TRIGGER;

signal1 <= signal2;

signal2 <= signal1 + signal3;

signal3 <= signal2;

RESULT  <= signal1 + signal2 + signal3;

**end process**;

**end** SIGN;

* All of these signals are computed at the same time, using the old values of signal1, 2 and 3.
* All the signals will be updated at Delta time after the TRIGGER has arrived.
* signal1= 2, signal2= 4 (=1+3), signal3=2 and RESULT=6

**Data Types**

* type defines the set of values that the object can have and the set of operations that are allowed on it
* not allowed to assign a value of one type to an object of another data type
* 4 classes of data types:
  1. Scalar: single value, ordered. Relational operators can be used on them.
     1. Eg: integer, real, enumerated
  2. Composite
  3. Access
  4. File

* **Types defined in the Package *Standard* of the *std* Library**

**library** std, work;

**use** std.standard.all;

|  |  |  |
| --- | --- | --- |
| **Type** | **Range of values** | **Example** |
| **bit** | ‘0’, ‘1’ | signal A: bit :=1; |
| **bit\_vector** | an array with each element of type bit | signal INBUS: bit\_vector(7 downto 0); |
| **boolean** | FALSE, TRUE | variable TEST: Boolean :=FALSE’ |
| **character** | any legal VHDL character (see package standard); printable characters must be placed between single quotes (e.g. ‘#’) | variable VAL: character :=’$’; |
| **file\_open\_kind** | read\_mode, write\_mode, append\_mode |  |
| **file\_open\_status** | open\_ok, status\_error, name\_error, mode\_error |  |
| **integer** | range is implementation dependent but includes at least –(231 – 1) to +(231 – 1) | constant CONST1: integer :=129; |
| **natural** | integer starting with 0 up to the max specified in the implementation | variable VAR1: natural :=2; |
| **positive** | integer starting from 1 up the max specified in the implementation | variable VAR2: positive :=2; |
| **real** | floating point number in the range of –1.0 x 1038 to +1.0x 1038 (can be implementation dependent | variable VAR3: real :=+64.2E12; |
| **severity\_level** | note, warning, error, failure |  |
| **string** | array of which each element is of the type character | variable VAR4: string(1 to 12):= “@$#ABC\*()\_%Z”; |
| **time** | an integer number of which the range is implementation defined; units can be expressed in sec, ms, us, ns, ps, fs, min and hr | variable DELAY: time :=5 ns; |

* **bit\_vector** : collection of bits

entity demux is  
  port (e: in bit\_vector (3 downto 0);     -- enables for each output  
        s: in bit\_vector (1 downto 0);     -- select signals  
        d: out bit\_vector (3 downto 0));   -- four output signals  
end demux;

architecture rtl of demux is  
  signal t : bit\_vector(3 downto 0);       -- an internal signal (s*ignal declaration)*  
begin  
  t(3)<=s(1) and s(0);  
  t(2)<=s(1) and not s(0);  
  t(1)<=not s(1) and s(0);  
  t(0)<=not s(1) and not s(0);  
  d<=e and t;  
end rtl;

* std\_logic\_vector
* **time** :
  + physical type
    - has 2 parts, number and unit
* **integer**
* **Real**
* **Expressing number in different base:**

Base 2:   2#10010#  (representing the decimal number “18”)

Base 16: 16#1D#

Base 8:   8#22#  
Base 2:   2#1001\_1101\_1100\_0010#

* Bit strings:  
  Binary:  B”1100\_1001”, b”1001011”

Hexagonal: X”C9”, X”4b”

Octal: O”311”, o”113”

* Assigning values to a bit\_vector:  
  *d<="1100";  
  d<=X"C";* -- X means hexadecimal

* **User-defined Types**
  + **type** identifier **is** type\_definition;
  + Eg:
    - Integer types:  
      **type** small\_int **is range** 0 **to** 1024**;**

**type** my\_word\_length **is range** 31 **downto** 0**;**

**subtype** data\_word **is** my\_word\_length **range** 7 **downto** 0;

**subtype** int\_small **is**  integer **range** -1024 **to** +1024;

* Floating Point Types:  
  **type** cmos\_level **is range** 0.0 **to** 3.3;

**type** pmos\_level **is range** -5.0 **to** 0.0;

**type** probability **is range** 0.0 **to** 1.0;

**subtype** cmos\_low\_V **is** cmos\_level **range** 0.0 to +1.8;

* Physical Types:  
  **type** conductance **is range** 0 **to** 2E-9

units

mho;

mmho = 1E-3 mho;

umho = 1E-6 mho;

nmho = 1E-9 mho;

pmho = 1E-12 mho;

**end units** conductance;

* Object declarations using above types:  
  **variable** BUS\_WIDTH: small\_int :=24;

**signal** DATA\_BUS: my\_word\_length;

**variable** VAR1: cmos\_level **range** 0.0 **to** 2.5;

**constant** LINE\_COND: conductance:= 125 umho;

* to use our own types, we need either to include the type definition inside an architecture body or to declare the type in a package, eg:

**package** my\_types **is**

**type** small\_int **is range** 0 **to** 1024**;**

**type** my\_word\_length **is range** 31 **downto** 0**;**

**subtype** data\_word **is** my\_word\_length **is range** 7 **downto** 0;

**type** cmos\_level **is range** 0.0 **to** 3.3;

**type** conductance **is range** 0 **to** 2E-9

units

mho;

mmho = 1E-3 mho;

umho = 1E-6 mho;

nmho = 1E-9 mho;

pmho = 1E-12 mho;

**end units** conductance;

**end package** my\_types;

* **Enumerated Types**
  + consists of lists of character literals or identifiers
  + **type** *type\_name* **is** (*identifier list or character literal*);
  + Eg:  
    **type** my\_3values **is** (‘0’, ‘1’, ‘Z’);

**type** PC\_OPER  **is** (load, store, add, sub, div, mult, shiftl, shiftr);

**type** hex\_digit  **is** (‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, 8’, ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’);

**type** state\_type **is** (S0, S1, S2, S3);

Objects of above types:  
**signal** SIG1: my\_3values;

**variable** ALU\_OP: pc\_oper;

**variable** first\_digit: hex\_digit :=’0’;

**signal** STATE: state\_type :=S2;

* If one does not initialize the signal, the default initialization is the leftmost element of the list
* have to be defined in the architecture body or inside a packages
* std\_ulogic is an example of enumerated type (defined in std\_logic\_1164 package):

**type** STD\_ULOGIC **is** (

‘U’,        -- uninitialized

‘X’,        -- forcing unknown

‘0’,        -- forcing 0

‘1’,        -- forcing 1

‘Z’,        -- high impedance

‘W’,        -- weak unknown

‘L’,        --  weak 0

‘H’.        -- weak 1

‘-‘);        -- don’t care

* **Composite Types**
  + collection of related data elements in the form of an *array* or *record*
  + **Array Types**
    - Declaration:  
      **type** *array\_name* **is array** (*indexing scheme)* **of** *element\_type*;
    - Eg:

**type** MY\_WORD **is array** (15 **downto** 0) **of** std\_logic;

**type** YOUR\_WORD **is array** (0 **to** 15) **of** std\_logic;

**type** VAR **is array (**0 to 7) **of** integer**;**

**type** STD\_LOGIC\_1D **is array (**std\_ulogic) **of** std\_logic**;**

Index:        ‘U’  ‘X’  ‘0’  ‘1’  ‘Z’  ‘W’  ‘L’  ‘H’  ‘-‘

Element:

* Declaring objects of above types:  
  **signal** MEM\_ADDR: MY\_WORD; --initialized to all ‘0’s

**signal** DATA\_WORD: YOUR\_WORD :=  B“1101100101010110”;

**constant** SETTING: VAR := (2,4,6,8,10,12,14,16);

* Element access:  
  MEM\_ACCR(15) accesses the left most bit of the array, while DATA\_WORD(15) accesses the right most bit of the array with value ‘0’
* Subrange:  
  MEM\_ADDR(15 **downto** 8) or DATA\_WORD(0 **to** 7)
* Multidimensional Arrays:

**type** MY\_MATRIX3X2 **is array** (1 **to** 3, 1 **to** 2) **of** natural;

**type** YOUR\_MATRIX **is array** (1 **to** 4, 1 **to** 2) **of** integer;

**type** STD\_LOGIC\_2D **is array (**std\_ulogic, std\_ulogic) **of** std\_logic**;**

9x9 array or table with an index the elements of the std\_ulogic type.

**variable** DATA\_ARR: MY\_MATRIX :=((0,2), (1,3), (4,6), (5,7));  
  
DATA\_ARR will then be initialized to:

0  2

1  3

4  6

5  7

DATA\_ARR(3,1) returns the value 4

* Unconstrained array type (no dimensions while declaring):  
  **type** *array\_name* **is array** (*type* **range <>) of** *element\_type*;  
    
  Eg:  
  **type** MATRIX **is array** (integer **range** <>) of integer;

**type** VECTOR\_INT **is array** (natural **range** <>) of integer;

**type** VECTOR2 **is array** (natural **range** <>, natural **range** <>) of std\_logic;  
  
Range specified during object declaration:  
**variable** MATRIX8: MATRIX (2 **downto** -8) := (3, 5, 1, 4, 7, 9, 12, 14, 20, 18);

**variable** ARRAY3x2: VECTOR2 (1 **to** 4, 1 **to** 3)) := ((‘1’,’0’), (‘0’,’-‘), (1, ‘Z’));

* **Record Type**
  + consists of multiple elements that may be of different types
  + Syntax:  
    **type** *name* **is**

**record**

identifier :subtype\_indication;

:

identifier :subtype\_indication;

**end record**;

* Eg:  
  **type** MY\_MODULE  **is**

**record**

RISE\_TIME     :time;

FALL\_TIME    : time;

SIZE        : integer **range** 0 **to** 200;

DATA        : bit\_vector (15 **downto** 0);

**end record;**

**signal** A, B: MY\_MODULE;  
  
Accessing/assigning:  
A.RISE\_TIME <= 5ns;

A.SIZE <= 120;

B <= A;

* **Type Conversion**

**Conversions supported by std\_logic\_1164 package**

|  |  |
| --- | --- |
| **Conversion** | **Function** |
| std\_ulogic    to bit | to\_bit(*expression*) |
| std\_logic\_vector  to bit\_vector | to\_bitvector(*expression*) |
| std\_ulogic\_vector  to bit\_vector | to\_bitvector(*expression*) |
| bit  to std\_ulogic | To\_StdULogic(*expression)* |
| bit\_vector   to   std\_logic\_vector | To\_StdLogicVector(*expression*) |
| bit\_vector   to   std\_ulogic\_vector | To\_StdUlogicVector(*expression*) |
| std\_ulogic to std\_logic\_vector | To\_StdLogicVector(*expression*) |
| std\_logic to std\_ulogic\_vector | To\_StdUlogicVector(*expression*) |

* Eg:  
  **entity** QUAD\_NAND2 **is**

**port** (A, B: **in** bit\_vector(3 **downto** 0);

out4: **out** std\_logic\_vector (3 **downto** 0));

**end** QUAD\_NAND2;

**architecture** behavioral\_2 **of** QUAD\_NAND2 **is**

**begin**

out4 <= to\_StdLogicVector(A **and** B);

**end** behavioral\_2;

* Type conversions between integer types or between similar array types are possible
* Conversion between array types is possible if they have the same length and if they have identical element types or convertible element types.
* Enumerated types cannot be converted.

* **Attributes**
  + VHDL supports 5 types of attributes
  + used to return various types of information about a signal, variable or type
  + consist of a quote mark (‘) followed by the name of the attribute
  + Scalar Attributes:

|  |  |
| --- | --- |
| **Attribute** | **Value** |
| scalar\_type’**left** | returns the first or leftmost value of scalar-type in its defined range |
| scalar\_type’**right** | returns the last or rightmost value of scalar-type in its defined range |
| scalar\_type’**low** | returns the lowest value of scalar-type in its defined range |
| scalar\_type’**high** | returns the greatest value of scalar-type in its defined range |
| scalar\_type’**ascending** | True if T is an ascending range, otherwise False |
| scalar\_type’**value(s)** | returns the value in T that is represented by s (s stands for string value). |

Eg:

**type** conductance **is range** 1E-6 **to** 1E3

**units** mho;

**end units** conductance;

**type** my\_index **is range** 3 **to** 15;

**type** my\_levels **is** (low, high, dontcare, highZ);

conductance’right         returns:    1E3

conductance’high              1E3

conductance’low                1E-6

my\_index’left                 3

my\_index’value(5)             “5”

my\_levels’left                 low

my\_levels’low                 low

my\_levels’high                highZ

my\_levels’value(dontcare)      “dontcare”

* Array Attributes

|  |  |
| --- | --- |
| **Attribute** | **Returns** |
| MATRIX‘**left**(N)  MATRIX’**right**(N)  MATRIX’**high**(N)  MATRIX’**low**(N)  MATRIX’**length**(N)  MATRIX’**range**(N)  MATRIX’**reverse\_range**(N)  MATRIX’**ascending**(N) | left-most element index  right-most index  upper bound  lower bound  the number of elements  range  reverse range  a Boolean value TRUE if index is an ascending range, otherwise FALSE |

N between parentheses refers to the dimension. For a one-dimensional array, one can omit the number N

Eg:  
**type** MYARR8x4 **is array** (8 **downto** 1, 0 to 3) **of** boolean;

**type** MYARR1 **is array** (-2 **to** 4)  **of** integer;

MYARR1’left        returns:        -2

MYARR1’right                 4

MYARR1’high                 4

MYARR1’reverse\_range           4 downto to -2

MYARR8x4’left(1)              8

MYARR8x4’left(2)              0

MYARR8x4’right(2)             3

MYARR8x4’high(1)              8

MYARR8x4’low(1)                1

MYARR8x4’ascending(1)         False

**Operators**

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

* order of precedence is the highest for the operators of class 7, followed by class 6 with the lowest precedence for class 1
* Operators of the same class have the same precedence and are applied from left to right in an expression

* Logical Operators
  + defined for the “bit”, “boolean”, “std\_logic” and “std\_ulogic” types and their vectors
  + give a result of the same type as the operand
  + can be applied to signals, variables and constants
* Relational Operators
  + give as result a Boolean output of “TRUE” or “FALSE”

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Operand Types** | **Result Type** |
| = | Equality | any type | Boolean |
| /= | Inequality | any type | Boolean |
| < | Smaller than | [scalar](https://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html#scalarType) or discrete array types | Boolean |
| <= | Smaller than or equal | scalar or discrete array types | Boolean |
| > | Greater than | scalar or discrete array types | Boolean |
| >= | Greater than or equal | scalar or discrete array types | Boolean |

* Eg:

**variable** STS        : Boolean;

**constant** A        : integer :=24;

**constant** B\_COUNT    : integer :=32;

**constant** C        : integer :=14;

STS <= (A < B\_COUNT) ;  -- will assign the value “TRUE” to STS

STS <=  ((A >= B\_COUNT) **or** (A > C));    -- will result in “TRUE”

STS <=  (std\_logic (‘1’, ‘0’, ‘1’) < std\_logic(‘0’, ‘1’,’1’));--makes STS “FALSE”

**type** new\_std\_logic **is** (‘0’, ‘1’, ‘Z’, ‘-‘);

**variable** A1: **new\_std\_logic** :=’1’;

**variable** A2: **new\_std\_logic** :=’Z’;

STS <=  (A1 < A2); will result in “TRUE” since ‘1’ occurs to the left of ‘Z’.  
  
For discrete array types, the comparison is done on an element-per-element basis, starting from the left towards the right

* Shift Operators
  + perform a bit-wise shift or rotate operation on a one-dimensional array of elements of the type bit (or std\_logic) or Boolean

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Operand Type** | **Result Type** |
| **sll** | Shift left logical (fill right vacated bits with the 0) | Left: Any one-dimensional array type with elements of type bit or Boolean; Right: integer | Same as left type |
| **srl** | Shift right logical (fill left vacated bits with 0) | same as above | Same as left type |
| **sla** | Shift left arithmetic (fill right vacated bits with rightmost bit) | same as above | Same as left type |
| **sra** | Shift right arithmetic (fill left vacated bits with leftmost bit) | same as above | Same as left type |
| **rol** | Rotate left (circular) | same as above | Same as left type |
| **ror** | Rotate right (circular) | same as above | Same as left type |

* Eg:  
  **variable** NUM1    :bit\_vector := “10010110”;

NUM1 **srl** 2;  
will result in the number “00100101”.

* When a negative integer is given, the opposite action occurs, i.e. a shift to the left will be a shift to the right

* Addition Operators
  + perform arithmetic operation (addition and subtraction) on operands of any numeric type
  + concatenation (&) operator is used to concatenate two vectors
  + Need to specify the ieee.std\_logic\_unsigned.all or std\_logic\_arith package package in addition to the ieee.std\_logic\_1164 package

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Description** | **Left Operand Type** | **Right Operand Type** | **Result Type** |
| + | Addition | Numeric type | Same as left operand | Same type |
| - | Subtraction | Numeric type | Same as left operand | Same type |
| & | Concatenation | Array or element type | Same as left operand | Same array type |

* Eg:

**signal** MYBUS         :std\_logic\_vector (15 **downto** 0);

**signal** STATUS          :std\_logic\_vector (2 **downto** 0);

**signal** RW, CS1, CS2        :std\_logic;

**signal**     MDATA            :std\_logic\_vector ( 0 **to** 9);

MYBUS <= STATUS & RW & CS1 & CS2 & MDATA;

* Unary Operators
  + used to specify the sign of a numeric type

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Operand Type** | **Result Type** |
| + | Identity | Any numeric type | Same type |
| - | Negation | Any numeric type | Same type |

* Multiplying Operators
  + perform mathematical functions on numeric types (integer or floating point)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Description** | **Left Operand Type** | **Right Operand Type** | **Result Type** |
| \* | Multiplication | Any integer or floating point | Same type | Same type |
|  |  | Any physical type | Integer or real type | Same as left |
|  |  | Any integer or real type | Any physical type | Same as right |
| / | Division | Any integer or floating point | Any integer or floating point | Same type |
|  |  | Any physical type | Any integer or real t ype | Same as left |
|  |  | Any physical type | Same type | Integer |
| **mod** | Modulus | Any integer type |  | Same type |
| **rem** | Remainder | Any integer type |  | Same type |

* result of the **rem** operator has the sign of its first operand while the result of the **mod** operators has the sign of the second operand
* remainder (rem) and modulus (mod) are defined as follows:

A **rem** B = A –(A/B)\*B        (in which A/B in an integer)

A **mod** B = A – B \* N        (in which N is an integer)

* Eg:  
  11 **rem** 4        results in 3

(-11) **rem** 4        results in -3

9 **mod** 4        results in 1

7 **mod** (-4)        results in –1  (7 – 4\*2 = -1).

* Miscellaneous Operators

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Description** | **Left Operand Type** | **Right Operand Type** | **Result Type** |
| \*\* | Exponentiation | Integer type | Integer type | Same as left |
|  |  | Floating point | Integer type | Same as left |
| **abs** | Absolute value | Any numeric type |  | Same type |
| **not** | Logical negation | Any bit or Boolean type |  | Same type |



**Libraries and Packages**

* File containing common code can to be used by many designs.
* A library defines how certain keywords behave in your file
* Using a library, eg: textio.all (For i/p, o/p of simulations):  
  use textio.all;
  + Above statement should appear before every architecture that uses this library
  + text i/o done using variable type *line*
  + text i/o done using variables => can be done in processes
  + *write* function: append text at end of a line variable
    - used to append constant values and the value of variables and signals of the types bit, bit\_vector, time, integer, and real
  + *writeline* function: outputs the current value of a line to the monitor, and empties the line for re-use

* Eg:  
  use textio.all;  
  architecture behavior of check is  
  begin  
    process (x)  
      variable s : line;  
      variable cnt : integer:=0;  
    begin  
      if (x='1' and x'last\_value='0') then  
        cnt:=cnt+1;  
        if (cnt>MAX\_COUNT) then  
          write(s,"Counter overflow - ");  
          write(s,cnt);  
          writeline(output,s);  
        end if;  
      end if;  
    end process;  
  end behavior;

* ieee Library:
  + std\_logic\_1164 package: defines the standard datatypes
  + std\_logic\_arith package: provides arithmetic, conversion and comparison functions for the signed, unsigned, integer, std\_ulogic, std\_logic and std\_logic\_vector types
  + std\_logic\_unsigned
  + std\_logic\_misc package: defines supplemental types, subtypes, constants and functions for the std\_logic\_1164 package.
* Package Declaration:

-- Package declaration

**package** *name\_of\_package* **is**

package declarations

**end package** *name\_of\_package*;

-- Package body declarations

**package body** *name\_of\_package* **is**

package body declarations

**end package body** *name\_of\_package*;

* **Behavioral Modelling**
  + Highest abstraction level
  + describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them
  + Process
    - Main construct in behavioral modelling
    - Allows modelling of complex digital systems, in particular sequential circuits
    - Allows use of sequential statements to describe the behavior of a system over time
    - Syntax:  
      [*process\_label*:] **process** [ (*sensitivity\_list*) ] [**is**]

[ *process\_declarations*]

**begin**

*list of sequential statements such as*:

*signal assignments*

*variable assignments*

*case statement*

*exit statement*

*if statement*

*loop statement*

*next statement*

*null statement*

*procedure call*

*wait statement*

**end process** [*process\_label*];

* Eg: positive edge-triggered D flip-flop with asynchronous clear input follows.

**library** ieee;

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

**entity** DFF\_CLEAR **is**

**port** (CLK, CLEAR, D : **in** std\_logic;

Q : **out** std\_logic);

**end** DFF\_CLEAR;

**architecture** BEHAV\_DFF **of** DFF\_CLEAR **is**

**begin**

DFF\_PROCESS: **process** (CLK, CLEAR)

**begin**

**if** (CLEAR = ‘1’) **then**

Q <= ‘0’;

**elsif** (CLK’**event** **and** CLK = ‘1’) **then**

Q <= D;

**end if;**

**end process;**

**end** BEHAV\_DFF;

* can appear in the body of an architecture declaration
* Is a concurrent statement, but statements inside it are executed sequentially
* can make assignments to signals that are defined externally (e.g. interface ports) to the process, such as the Q output of the flip-flop in the above example
* CLK’**event** **and** CLK = ‘1’ checks for a positive clock edge (clock event AND clock high).
* sensitivity list: set of signals to which the process is sensitive
  + Any change in the value of the signals in the sensitivity list will cause immediate execution of the process
  + Whenever any event occurs on one of the signals in the sensitivity list, the process is re-evaluated
    - by performing each statement that it contains
    - statements (the body of the process) appear between the *begin* and *end* keywords
    - statements in the body of the process are performed (or executed) in order from first to last
    - When the last statement has been executed the process is finished and is said to be *suspended*
    - When an event occurs on a signal in the sensitivity list, the process is said to be *resumed* and the statements will be executed from top to bottom again
    - Each process is executed once during the beginning of a simulation to determine the initial values of its outputs.

* If no sensitivity list specified, must include a **wait** statement to make sure that the process will halt
* Variables and constants that are used inside a process have to be defined in the *process\_declarations* part before the keyword **begin**
* variable assignments inside a process are executed immediately and denoted by the “:=” operator (unlike <= which has a delay)
* can also describe combinational circuits with the process construct
* Signal vs Variable assignment in a process:
  + Eg:  
    ...  
    signal x,y,z : bit;  
    ...  
    process (y)  
    begin  
      x<=y;  
      z<=not x;  
    end process;
    - If the signal *y* changes then an event will be scheduled on *x* to make it the same as *y*
    - an event is scheduled on *z* to make it the opposite of *x*
    - when the second statement is executed, the event on *x* has not been processed yet
  + Eg:  
    process (y)  
    variable x,z : bit;  
    begin  
      x:=y;  
      z:=not x;  
    end process;
    - value of the variable *z* would be the opposite of the value of *y* because the value of the variable *x* is changed immediately
* If Statement
  + Syntax:

**if** *condition* **then**

sequential statements

   [**elsif** *condition* **then**

sequential statements ]

   [**else**

sequential statements ]

**end if;**

* can be used to describe combinational circuits as well

* Sequential Statements
  + Can only be used in the body of a process
  + Executed sequentially
  + if statement
    - Eg:  
      count: process (x)  
        variable cnt : integer :=0 ;  
      begin  
        if (x='1' and x'last\_value='0') then   
          cnt:=cnt+1;  -- executed for every rising edge of x  
        end if;  
      end process;
    - Eg:  
      ...  
      if (inc='1') then  
        cnt:=cnt+1;  
      else  
        cnt:=cnt-1;  
      end if;  
      ...
  + Loop statements
    - for
      * Eg:  
        signal x : bit\_vector (7 downto 0);  
        ...  
        process (x)  
          variable p : bit;  
        begin  
          p:='0'  
          for i in 7 downto 0 loop -- *parameter specification (how many times to run)*  
            p:=p xor x(i);  
          end loop;  
        end process;