VHDL

from

V. A. Pedroni, "*Circuit Design and Simulation with VHDL*", 2nd ed., MIT Press, 2010

VHDL Code Structure

outlines

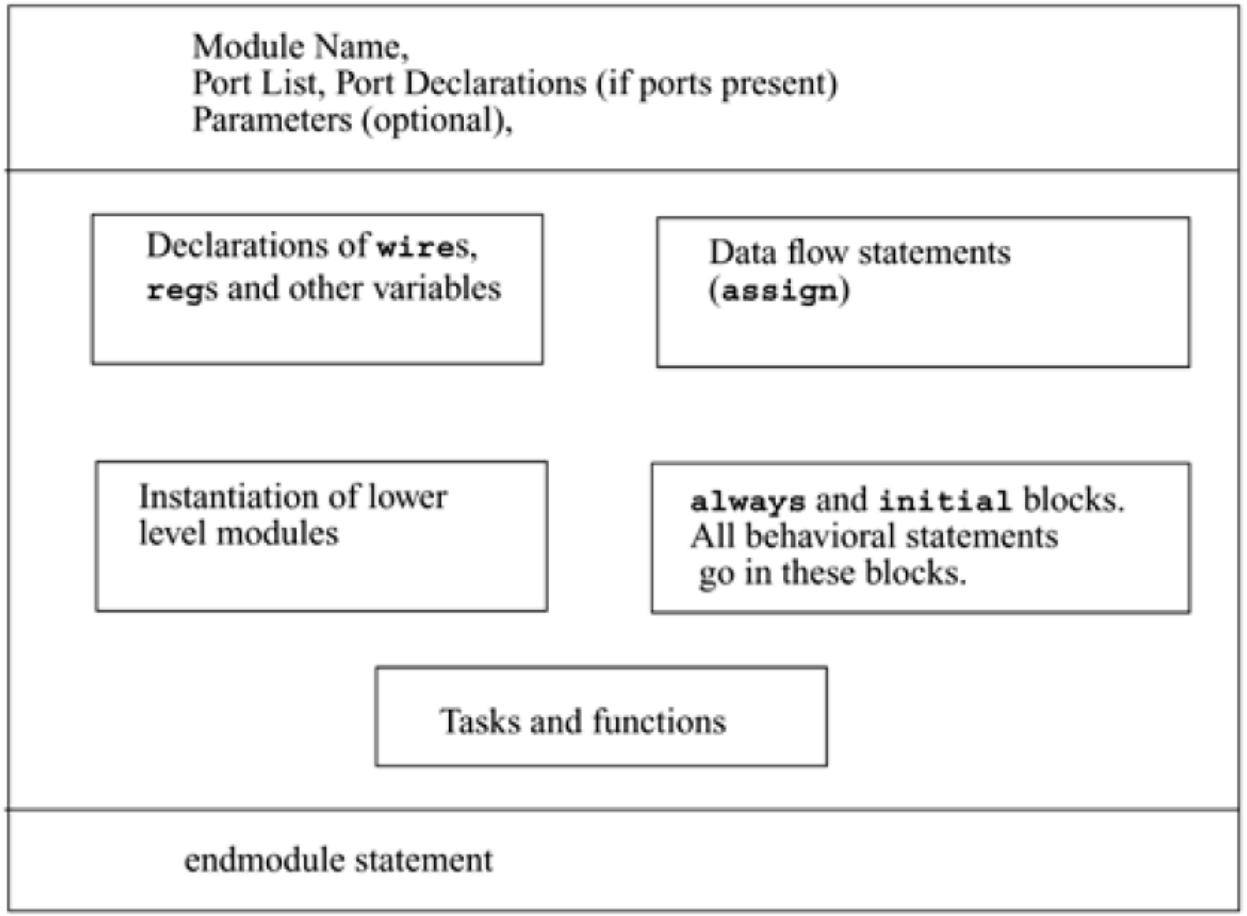
- fundamentals of VHDL codes
- data types
- operators and attributes

Hardware Description Language (HDL)

- HDL
  - originally for purpose of simulation
  - later for design (only *synthesizable* subset of HDL)
  - allows design to be described at a higher abstract level (such as behavioral level)
  - functional verification of the design can be done early in the design cycle (RTL simulation)
  - logic synthesis tools (such as Synopsys Design Compiler) automatically convert design into gate-level netlist of a selected fabrication technology
- Verilog HDL : IEEE standard 1364 in 1995
  - U.S.A. (Silicon Valley), Japan, and **Taiwan**
- VHDL (VHSIC HDL) : IEEE standard 1076 in 1987
  - U.S.A. (IBM, Intel, TI), Europe, Korea

Components of a Verilog Module

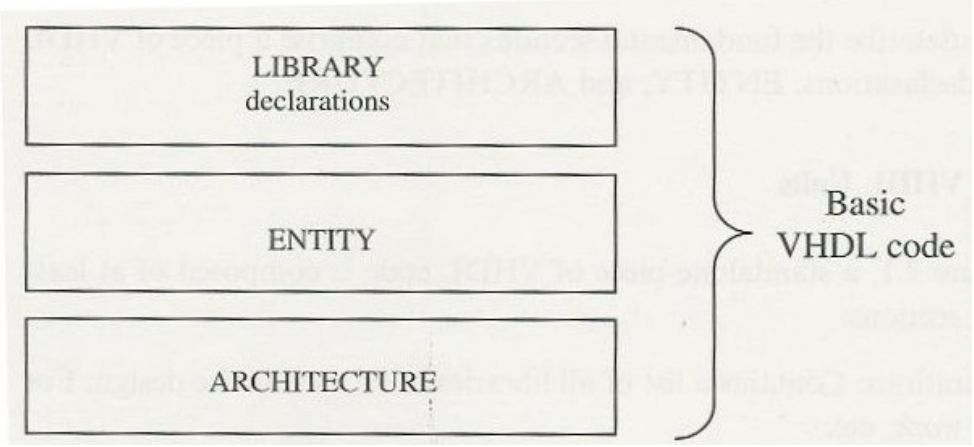
- declaration
  - inputs, outputs
  - parameters
  - data types
- ***structure*** model
- module instantiation
- ***dataflow*** model
  - continuous assignment
- ***behavior*** model
  - procedural assignment



Fundamentals of VHDL Codes

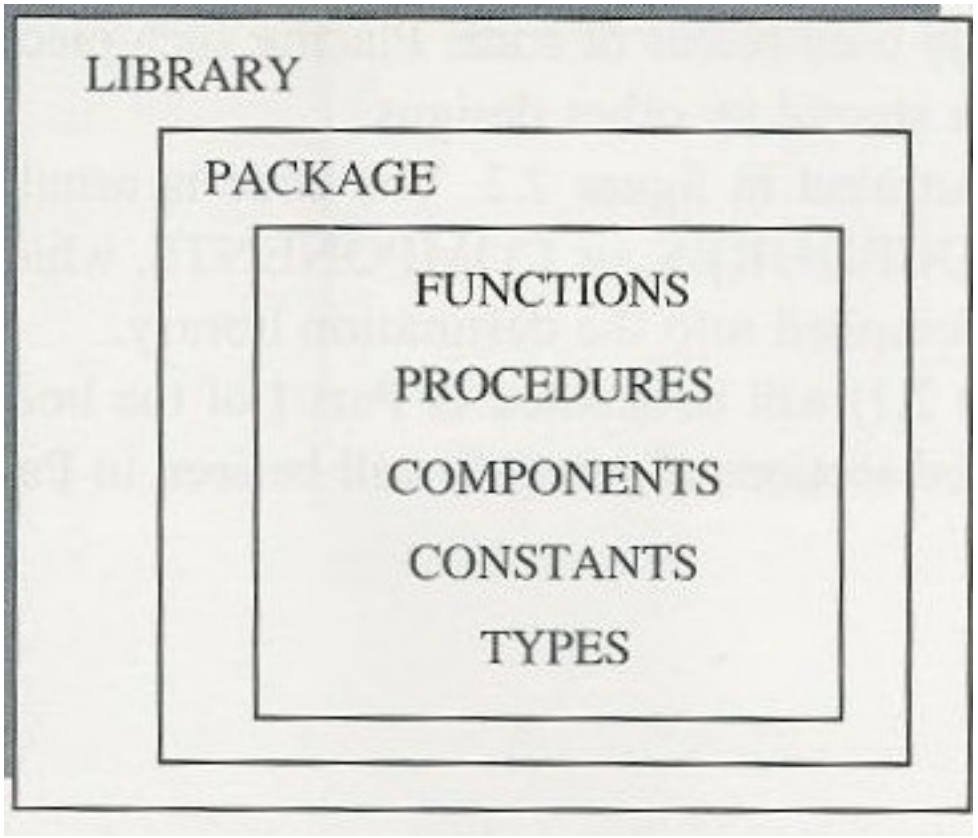
- **LIBRARY**
  - Collection of commonly used codes
- **ENTITY**
  - I/O pins of the circuit
- **ARCHITECTURE**
  - Functions of the circuit

```
-- example of VHDL code for D-flip-flop
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY dff IS
  PORT (d, clk, rst: IN STD_LOGIC;q: OUT STD_LOGIC);
END dff;
---------------------------------------
ARCHITECTURE behavior OF dff IS
BEGIN
   PROCESS (rst, clk)   -- asynchronous reset
     BEGIN
       IF (rst='1') THEN   -- active-hight reset
          q <= '0';
          ELSIF (clk'EVENT AND clk='1') THEN q <= d;
          END IF;
   END PROCESS;
END behavior;
```



Fundamental Parts of Library

- **PACKAGE**
  - **FUNCTIONS, PROCEDURES, COMPONENTS,** data types, constants, …
- Some typical libraries:
  - **std** (default)
  - **work** (default)
  - **ieee (de facto default)**



Library declaration

LIBRARY library_name;

USE *library_name.package_name.package_parts*;

```
-- contain industry standard data type STD_LOGIC
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY std;
USE std.standard.all;

LIBRARY work;
USE work.all;
```

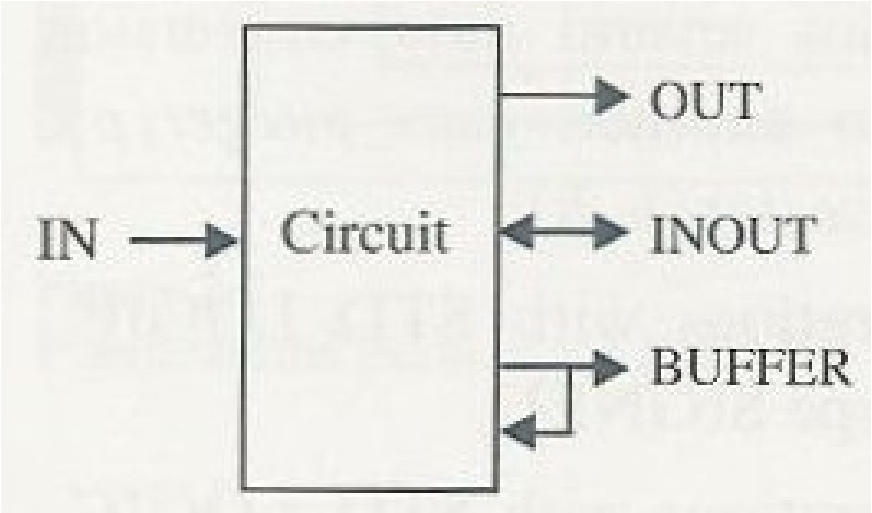
Packages in ieee library

- **std_logic_1164**
  - Define **STD_LOGIC** (8 levels), **STD_ULOGIC** (9 levels) in addition to default **BIT** (2 levels)
- **std_logic_arith**
  - Define **SIGNED** and **UNSIGNED** data types
- **std_logic_signed**
  - Allow operations on **STD_LOGIC_VECTOR** as if the data were of type **SIGNED** numbers
- **std_logic_unsigned**
  - Allow operations on **STD_LOGIC_VECTOR** as if the data were of type **UNSIGNED** numbers
- **In VHDL, bit-vectors (allowing logical operations) are different from numbers (allowing arithmetic opreations)**

ENTITY

```
-- similar to Verilog module declaration
-- VHDL is case-insensitive, i.e., a = A
ENTITY entity_name IS
PORT (
        port_name : signal_mode signal_type;
        …);
END entity_name; -- no entity name is OK
```

```
ENTITY nand_gate IS
  PORT (
      a, b : IN BIT;
      x    : OUT BIT);
END nand_gate;
```



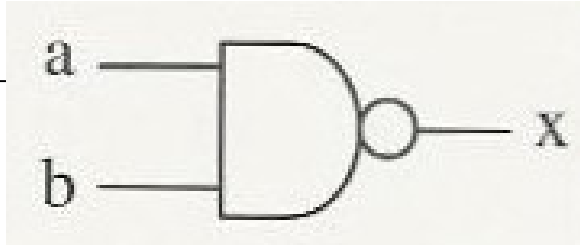
VHDL Architecture

- concurrent code
  - e.g., x <= a **NAND** b; -- NAND is logical operator
  - cp. Verilog dataflow code: **assigm** y= **~**(a **&** b);
- sequential code
  - e.g., **process** (clk) **if** (**clk'event and** clk=1) q<=d;
  - cp. Verilog procedural code: **always @ (posedge** clk) q <= d;
- components
  - e.g., FA1: FA **port map** (a, b, ci, s, co);
  - cp. Verilog structural code FA FA1(a, b, ci, s, co);

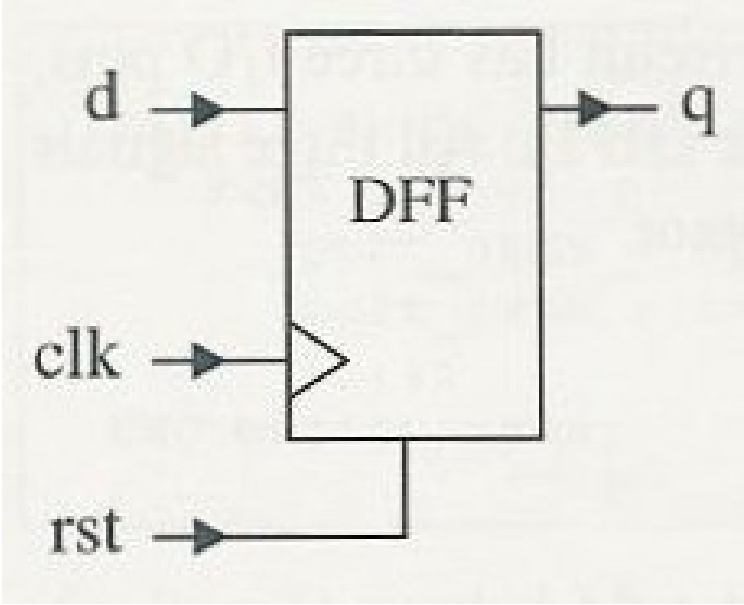
Architecture

ARCHITECTURE arch_name **OF** entity_name **IS**
[optional declaration]
BEGIN
(main code of describing hardware architecture)
END arch_name; -- no architecture name is OK

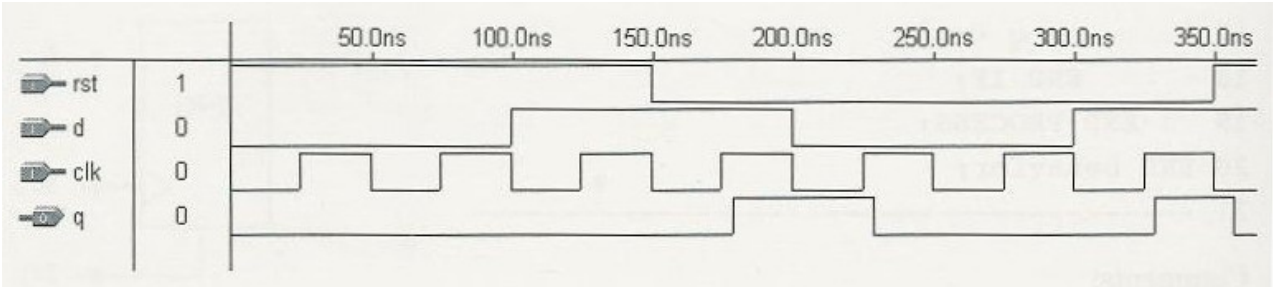
```
ARCHITECTURE myarch OF nand_gate IS
BEGIN
   x <= a NAND b;  // NAND is VHDL logical operator
END myarch;
```



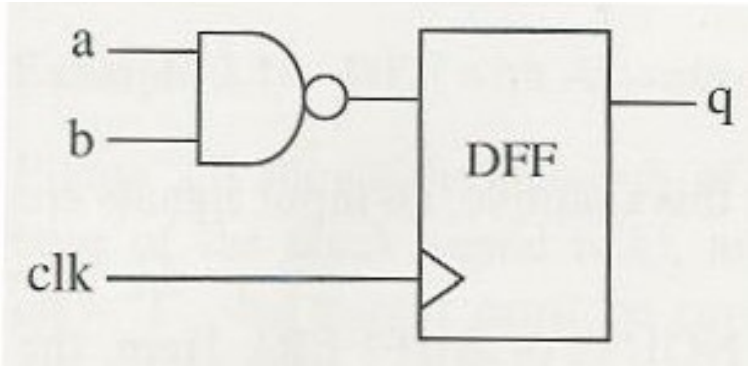
DFF (with asynchronous active-highr eset)



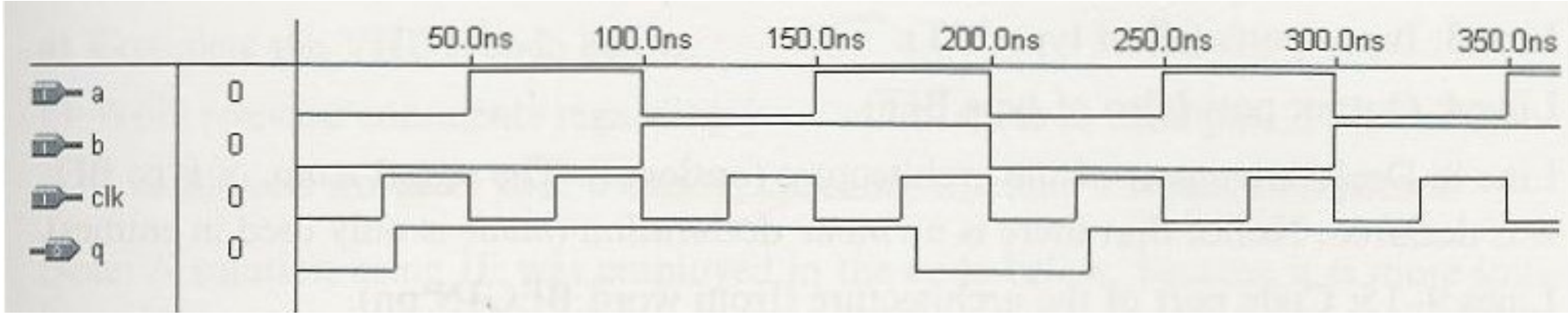

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------
ENTITY dff IS
        PORT (d, clk, rst: IN STD_LOGIC;
                q: OUT STD_LOGIC);
END dff;
---------------------------------
ARCHITECTURE behavior OF dff IS
BEGIN
        PROCESS (rst, clk)
        BEGIN
                IF (rst='1') THEN
                        q <= '0';
                ELSIF (clk'EVENT AND clk='1') THEN q <= d;
                END IF;
        END PROCESS;
END behavior;
-- END;
```



DFF with NAND




```
ENTITY example IS
        PORT (a, b, clk:     IN  BIT;
                             q: OUT BIT);
END example;
-----------------------------------
ARCHITECTURE example OF example IS
        SIGNAL temp : BIT;
BEGIN
        temp <= a NAND b;   -- assign temp = a & b;  in Verilog
        PROCESS (clk)  -- flip-flop
        BEGIN
                IF (clk'EVENT AND clk='1') THEN q<=temp; END IF;
        END PROCESS;
END example;
```



Data Types

Verilog (logic levels, data types)

- 4-logic levels
  - 0, 1, x, z
- data types
  - wire, reg, signed, unsigned, integer, real, time
  - bit vector
    - e.g., wire [7:0] x;
  - array
    - e.g., wire [7:0] x [0:1023];
- case sensitive
  - wire [7:0] x, X; // two signals x, X
- logic 1 (0) is same as true (false) and number

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

Verilog Data types: value set

- 4 values (0, 1, z, x), 8 strengths

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	▲
pull	Driving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

Verilog Four-Valued Logic System

- Signal and variable values represented using a 4-valued logic system
  - 1'b0: 1-bit binary logic 0 value
  - 1'b1: 1-bit binary logic 1 value
  - 1'bx: 1-bit binary 'x' or 'X' value
    - Unknown value, produced for uninitialized values or values driven to conflicting values by more than one signal source
  - 1'bz: 1-bit binary 'z' or 'Z' value
    - High impedance value, produced when a wire is disconnected from all signal sources driving that wire, i.e., floating node

Verilog data types (**wire** and **reg**)

- Nets (**wire**): connections between hardware elements
  - nets have values continuously by the outputs of devices that they are connected to

  eg.: **wire** a, b, c;
- Registers (**reg**): data storage element
  - a variable that can hold a value in procedural blocks starting with **always** or **initial**
  - *not necessarily* mean a flip-flop in real circuit

  eg.: **reg** reset;
- SystemVerilog use **logic** to replace 4-value **reg** to avoid confusion
  - not all **reg** data types synthesize into hardware registers

Verilog data types (vectors)

- Vectors: multiple bit widths (default is scalar)

```
wire [7:0] bus;        // 8-bit bus
wire [31:0] word;      // 32-bit word
reg [255:0] data;      // 256-bit word register, data[255] is MSB
reg [0:40] v_addr;  // v_addr[0] is MSB
```

- Vector Part Select

```
word[7:0]    // the least significant byte of the 32-bit vector word
v_addr[0:1]  // two most significant bits of the vector v_addr
```

- variable vector part select

[<starting_bit> **+:** width] : part-select *increments*

[<starting_bit> **-:** width] : part-select *decrements*

starting bit can be varied, but the width must be constant

```
eg1.: reg [255:0] data1;
      reg [0:255] data2;
      reg [7:0] byte;

      byte=data1[31-:8]  // data1[31:24]
      byte=data1[24+:8] // data1[31:24]
      byte=data2[31-:8]  // data2[24:31]
      byte=data2[24+:8] // data2[24:31]

eg2.  reg [255:0] data1;
      for (j=0; j<=31; j=j+1) byte = data1[(j*8)+:8] ;
      // sequence is [7:0], [15:8], …, [255:248]
```

Verilog register data types (integer, real, time)

- **integer**: register data type used for quantity
  - **reg** store *unsigned* quantity while **integer** store *signed* quantity

    eg., **integer** counter; counter = -1; // used as a counter
- **real**: real number constant or register data type

  eg. **real** delta;

  delta = 4e10; delta = 2.14; // delta is a real variable
- **time**: register data type to store simulation time

  eg. **time** save_sim_time;

  save_sim_time = **$time**; // define a time variable

  // system task **$time** is invoked to get current simulation time
- **real** and **time** are non-synthesizable codes for simulation only

Verilog-2001 signed vs. unsigned

- unsigned

```
wire [3:0] x;
wire [7:0] y;

assign y = {4'b0000, x};
```

```
wire [3:0] x;
wire [7:0] y;

assign y = x;   // zero padded
```

- signed

```
wire signed [3:0] x;
wire signed [7:0] y;

assign y = {4{x[3]}, x};
```

```
wire signed [3:0] x;
wire signed [7:0] y;

assign y = x; // sign-extended
```

```
wire [11:0] s1;
wire signed [11:0] s2;

assign s2 = $signed(s1);   // convert to signed number
// assign s1 = $unsigned(s2);  // convert to unsigned numer
```

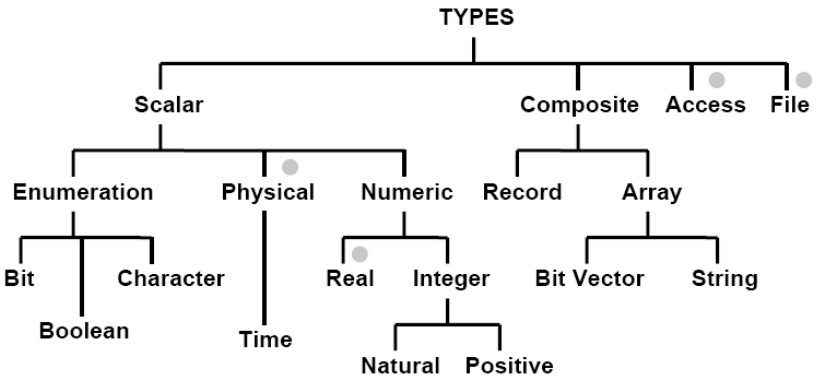
VHDL (logic levels, data types)

- logic levels
  - 2 logic levels (default)
  - 8 logic levels (ieee.std_logic)
- data object
  - signal
  - variable
  - constant
- data types
- case in-sensitive
- logic 1 (0) are different from true (false) and number

Resolved logic system (STD_LOGIC).

	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

Leaf is the default data type.



TYPE type_name IS ...

VHDL data objects

- **Signal**
  - Describe a real wire in the circuit
  - similar to **wire** in Verilog
  - assignment to signal (**<=**)
    - similar to non-blocking assignment in Verilog (parallel assignment)
- **Variable**
  - Holds any signal value from the values of the specified type.
  - Often used to hold temporary values within a **process** or **subprogram**.
  - assignment to variable (**:=**)
    - similar to blocking assignment = in Verilog (sequential assignment)
- **Constant**
  - Holds one specific value of the specified type
  - similar to **`define** compiler directive in Verilog
- **File**
  - useful for simulation

TYPE bit_file **IS FILE OF BIT**;

FILE file01: bit_file **IS** “my_file.txt”;

Signal

- represent circuit interconnections (wires)
- eg. **SIGNAL** enable: **BIT** := '0'; -- initial value is 0

  SIGNAL temp: **BIT_VECTOR** (3 **DOWNTO** 0);

  SIGNAL byte: STD_LOGIC_VECTOR (7 **DOWNTO** 0);

  SIGNAL count: **NATURAL RANGE** 0 **TO** 255;
- use "**<=**" to assign a value to a signal
  - eg. enable **<=** '1';
- when used in a sequential code (PROCESS or subprogram), update is NOT immediate (similar to non-blocking assignment **<=** in Verilog)
- multiple assignments are NOT allowed in *concurrent* code
- for multiple assignment in *sequentia*l code, only the last assignment is considered

Variable

- represent only *local* information
- only be declared, seen and modified inside sequential codes

VARIABLE flip: STD_LOGIC :='1';

VARIABLE address: STD_LOGIC_VECTOR (0 **TO** 15);

VARIABLE counter: **INTEGER RANGE** 0 **TO** 127;

- use "**:=**" to assign a value to a variable
  - eg. flip **:=** 0;
- update is immediate, similar to regular software programming
  - the new value can be used promptly in the next line of code

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

ENTITY parity_gen IS
  PORT (
      input : IN    std_logic_vector (7downto 0);
      output: OUT std_logic_vector (8 downto 0)
  );
END;
```

```
ARCHITECTURE parity OF parity_gen IS
BEGIN
      PROCESS (input)
            VARIABLE i: INTEGER;
            VARIABLE temp1: BIT;
            VARIABLE temp2: BIT_VECTOR (output'RANGE);
      BEGIN
            temp1 := '0';
            FOR i IN input'RANGE LOOP
                  temp1 := temp1 XOR input(i);
                  temp2(i) := input(i);
            END LOOP;
            temp2(output'HIGH) := temp1;
            output <= temp2;   --  assign variable to signal
      END PROCESS;
END parity;
```

Constant

```
CONSTANT bits: INTEGER := 16;
CONSTANT words: INTEGER := 2**bits;
 CONSTANT flag: BIT := '1';
CONSTANT mask: BIT_VECTOR(1 TO 8) := "00001111";
-- use ":=" to assign values to a constant
-- keyword "OTHERS" represents all unspecified index values,
CONSTANT a: BIT_VECTOR (5 DOWNTO 0) := (OTHERS=>'0');
-- constant a := "000000"
CONSTANT b: BIT_VECTOR (7 DOWNTO 0) := (7=>'0', OTHERS=>'1');
-- constant b = "01111111"

SIGNAL c: STD_LOGIC_VECTOR (1 TO 8) := (2|3=>'1', OTHERS=>'0');
-- initial value of signal c = "01100000"
VARIABLE d: BIT_VECTOR (1 TO 16) := (1 TO 8=>'1', OTHERS=>'0');
-- initial value of variable d = "1111111100000000"
```

VHDL data objects

- signal, variable, constant are synthesizable

Signal

Declaration: **signal** data **: std_logic** :='0' **;**

Assignment: data <= '1';

Constant - to enhance readability

Declaration: **constant** bitwidth **: std_logic_vector (7 downto 0)** := "01101110";

Variable - (process, procedure, function)

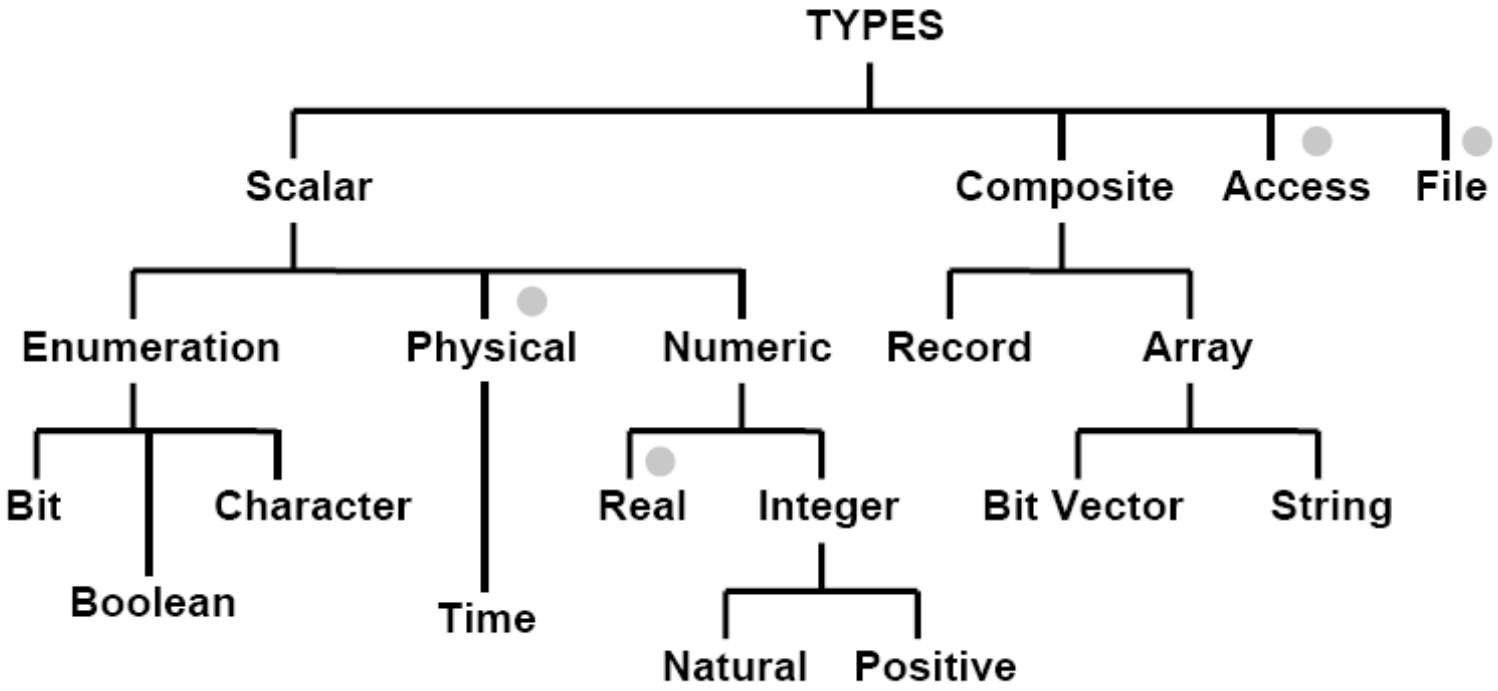
Declaration: **variable** data **: unsigned (0 to 2)** := "000";

Assignment: data := "101";

VHDL data types

- type of data objects (signal, variable, constant)

Leaf is the default data type.




```
TYPE type_name IS ...
```

Predefined VHDL data types

- Package ***standard*** in library ***std***
  - define **BIT, BOOLEAN, INTEGER, REAL** data types
- Package ***std_logic_1164*** in library ***ieee***
  - Define **STD_LOGIC, STD_ULOGIC** data types for bit
- Package ***std_logic_arith*** in library ***ieee***
  - define **SIGNED, UNSIGNED** data types for numbers
  - define data conversion functions such as **conv_integer(p), conv_unsigned (p,b), conv_signed (p,b), conv_logic_vector(p,b)**
- Package ***std_logic_signed*** and ***std_logic_unsigned*** in library ***ieee***
  - contains **functions** that allow arithmetic operations on **STD_LOGIC_VECTOR** bit-vector data to be performed as if the data were of type **SIGNED** or **UNSIGNED** numbers

Examples on BIT or BIT Vector

SIGNAL x: **BIT**;
-- 2-level logic (‘0’, ‘1’) **, cp. 8-lvel logic STD_LOGIC**
x **<=** ‘1’; -- single quotes ‘ for single bit

SIGNAL y: **BIT_VECTOR** (3 **DOWNTO** 0);
-- 4-bit vector with the **leftmost** bit y(index=3), y(3) as MSB
y **<=** “0111”; -- double quotes “ for vectors, MSB=0

SIGNAL w: **BIT_VECTOR** (0 **TO** 7);
-- 8-bit vector with the **rightmost** bit (index=7), y(7) as MSB
w **<=** “01110001”; -- MSB=1, the rightmost bit with index=7
Note: in Verilog bit vector
wire [0:7] w = 8’b 0111_0001; // MSB is w[0]=0, NOT w[7]=1

STD_LOGIC

SIGNAL x: **STD_LOGIC**; -- **wire** x; // Verilog equivalent

SIGNAL y: **STD_LOGIC_VECTOR** (3 **DOWNTO** 0) **:=** "0001";

-- **wire** [3:0] y = 4'b0001 ; // Verilog initialization

- Resolved logic system for STD_LOGIC
  - 'X': forcing unknown (synthesizable unknown)
  - '0': forcing low (synthesizable logic '1')
  - '1': forcing high (synthesizable logic '0')
  - 'Z': high impedance (synthesizable tri-state buffer)
  - 'W': weak unknown
  - 'L': weak low
  - 'H': weak high
  - '-': don't care

•most std_logic levels are intended for simulation only.
•'0', '1', 'z' are synthesizable.
•resolve conflicting logic levels in multiple-driven nodes

Resolved logic system (STD_LOGIC).

	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

STD_ULOGIC

- **STD_ULOGIC** add another logic level **'U'** meaning unsolved, in addition to the 8-valued logic of **STD_LOGIC**
  - making a 9-value logic
- contrary to **STD_LOGIC**, conflicting logic levels in **STD_ULOGIC** are NOT resolved, so the output wires should never be connected together directly
  - detect design error
- usually not used in design,

Other predefined data types

- **BOOLEAN -- different from bit '1', '0'**
  - True, False
- **INTEGER -- different from bit vector "00001111"**
  - 32-bit integers, $-(2^{31}-1) \sim +(2^{31}-1)$
- **NATURAL**
  - Non-negative integers (0, 1, 2, ….)
- **REAL**
  - Not synthesizable (only for simulation purpose)
- Character literals
  - ASCII character or string (not synthesizable)
- **SIGNED** and **UNSIGNED**
  - defined in *std_logic_arith* package of the *ieee* library
  - have appearance of **STD_LOGIC_VECTOR**, but accept arithmetic operations, which are typically used for **INTEGER** data types

More examples

X1 <= “00011111”;
X2 <= “0001_1111”; -- undescore _ for clarity
X3 <= “101111”; -- binary representation of decimal 47
X4 <= **B**”101111”; -- binary format of decimal 47
X5 <= **O**”57”; -- octal format of decimal 47
X6 <= **X**”2F”; -- hexadecimal format of decimal 47
n <= 1200; -- integer
IF ready **THEN** … -- Boolean, executed if ready=**TRUE**
Y <= 1.2**E**-5; -- real, not synthesizable
Q <= d **AFTER** 10 ns;
-- physical data type for timing, not synthesizable
-- similar to Verilog #10 assign Q = d; // `timescale 1ns/10ps

data types match/mis-match

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR (7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL e: INTEGER RANGE  0  TO  255;
…
a <= b(5);  -- legal (same scalar type: BIT)
b(0) <= a;
c <= d(5);
d(0) <= c;

a <= c; -- illegal (type mismatch: BIT vs. STD_LOGIC)
b <= d; -- illegal (type mismatch: BIT VECTOR vs.
          -- STD_LOGIC_VECTOR)
e <= b; -- illegal (type mismatch: INTEGER vs. BIT_VECTOR)
e <= d; -- illegal, (type mismatch: INTEGER vs. STD_LOGIC_VECTOR)
          -- but in Verilog, wire [7:0] d; imply 8-bit unsigned integer
```

User-defined data types

- integer

  TYPE negative **IS RANGE** **INTEGER'LOW** **TO** -1;

  TYPE my_integer **IS RANGE** -32 **TO** 32;

  -- user-defined subset of predefined data type **integer**

  TYPE student_grade **IS RANGE** 0 **TO** 100;

  -- user-defined subset of integers or naturals

- enumerate

  TYPE my_logc **IS** ('0', '1', 'Z');

  -- user-defined subset of std_logic

  TYPE bit_vector **IS** **ARRAY (NATURAL RANGE <>) OF BIT**;

  -- indeed, this is the predefined type BIT_VECTOR

  TYPE state **IS** (idle, forward, backward, white);

  -- typically used in finite state machine (FSM)

Subtypes

Resolved logic system (STD_LOGIC).

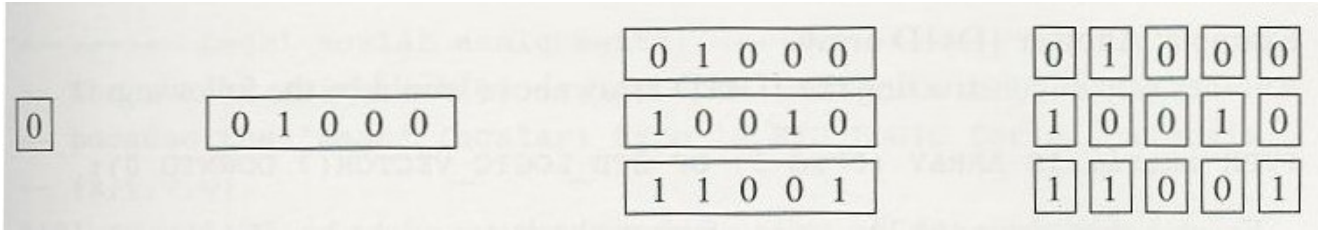
	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

- **TYPE** with a constraint
  - operations are allowed between *subtype* and the corresponding *base types*

SYBTYPE natural **IS INTEGER RANGE** 0 **TO INTEGER'HIGH**;
SUBTYPE small_integer **IS INTEGER RANGE** -32 **TO** 32;
SUBTYPE my_logic **IS** STD_LOGIC **RANGE** '0' **TO** '1';
SIGNAL a: **BIT**;
SIGNAL b: **STD_LOGIC**;
SIGNAL c: my_logic; -- base type is STD_LOGIC
b <= a; -- illegal (type mismatch: BIT vs. STD_LOGIC)
b <= c; -- legal (same "base" type: STD_LOGIC)

Arrays

- predefined *scalar* data types
  - **BIT, STD_LOGIC, STD_ULOGIC, BOOLEAN**
- predefined synthesizable *vector* (1D array) data types
  - **BIT_VECTOR**
  - **STD_LOGIC_VECTOR**
  - **STD_ULOGIC_VECTOR**
  - **INTEGER**
  - **SIGNED**
  - **UNSIGNED**



scalar **1D** **1D x 1D** **2D**

1D x 1D and 2D arrays

TYPE type_name **IS ARRAY** (specification) **OF** data_type;
SIGNAL signal_name : type_name [:= initial value]

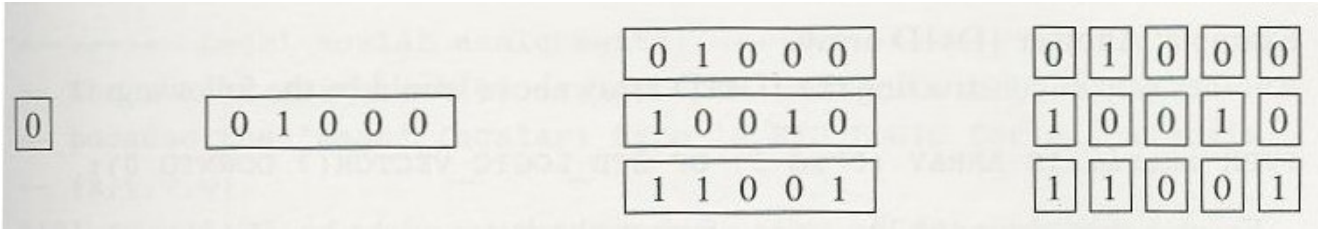
TYPE row **IS ARRAY** (7 **DOWNTO** 0) **OF** STD_LOGIC; -- 1D array **of scalars**

TYPE matrix **IS ARRAY** (0 **TO** 3) **OF** row; -- 1D x 1D array of scalars

TYPE matrix **IS ARRAY** (0 **TO** 3) **OF** STD_LOGIC_VECTOR(7 **DOWNTO** 0);
-- another 1D x 1D array, i.e., 1D array of logic vectors

SIGNAL x : matrix; -- 1D x 1D signal
-- **wire** [7:0] matrix[0:3]; // Verilog equivalent

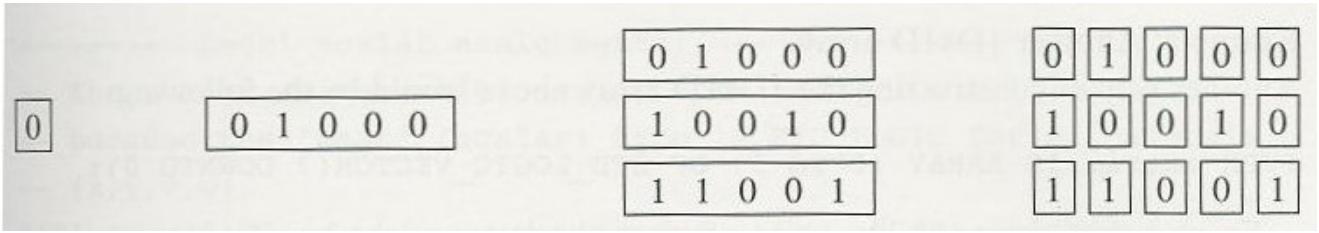
TYPE matrix2D **IS ARRAY** (0 **TO** 3, 7 **DOWNTO** 0) **OF** STD_LOGIC;
-- 2D array



Assignments of scalars and vectors

```
TYPE row     IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;
TYPE array1 IS ARRAY (0 TO 3) OF row;
TYPE array2 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
TYPE array3 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
SIGNAL x: row;      -- 1D array of scalars, which is different from a vector
SIGNAL y: array1;  -- 1D array of rows where row is 1D array of scalars
SIGNAL v: array2;  -- 1D array of vectors
SIGNAL w: array3;  -- 2D array of scalars
```

```
-- legal scalar assignment, same data type of std_logic –
x(0) <= y(1)(2);
x(1) <= v(2)(3);
x(2) <= w(2,1);
y(2)(0) <= v(0)(0);
w(3,0) <= v(0)(3);
```



row ???
array of scalars

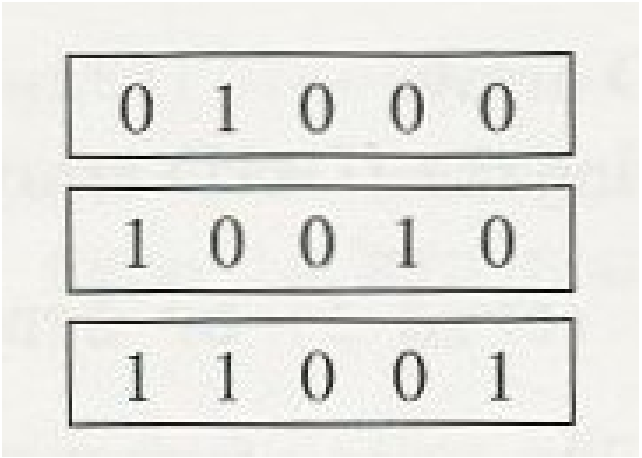
array1
arrat2

array3

```
-- vector assignments
x <= y(0);
x <= v(1) -- illegal (type mismatch: row (array of scalars) vs. STD_LOGIC_VECTOR
y(1)(7 DOWNTO 3) <= x(4 DOWNTO 0); -- legal (same type, same size)
w(1, 5 DOWNTO 1) <= v(2)(4 DOWNTO 0); -- illegal (type mismatch)
```

PORT Array (not allowed in Verilog)

```
---------- Package ----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------
PACKAGE my_data_types IS
  TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
                          STD_LOGIC_VECTOR(7 DOWNTO 0);
  END my_data_types;
…
-----------------------------------
```




```
---------- main code ----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_data_types.all;
-----------------------------------
ENTITY mux IS
  PORT (inp: IN vector_array (0 TO 3); … );  -- array of vectors
END mux;
-- But Verilog does NOT allow array in input/output port ports
```

array NOT allowed in Verilog I/O Ports

- module input/output ports could be data type of bit-vector, but not array
  - e.g., MUX4 (input [3:0] in, [1:0] s, output [3:0] out); // OK
  - e.g., MUX4 (input in [3:0], [1:0] s, output out); // NOTOK
- use long bit-vector as input/outputs and make conversion inside the module
  - e.g. MUX4 with 16-bit inputs/output

```
module MUX4 (input [4*16-1:0] in_bv, [1:0] s, output [15:0] out);
reg [15:0] in[0:3];
always@ (*) begin
  for (i=0; i<4; i=i+1)  in[i]= in_bv[16*i +:16]; // long bit-vector -> array
…
endmodule
```

Records

- contains objects of different types

```
TYPE birthday IS
RECORD
  day: INTEGER RANGE 1 TO 31;
  month: month_name;
END RECORD;
```

- record aggregations
  - named
  - positional

```
constant LEN: INTEGER := 8;

subtype BYTE_VEC is BIT_VECTOR(LEN-1 downto 0);

type BYTE_AND_IX is
  record
    BYTE: BYTE_VEC;
    IX:   INTEGER range 0 to LEN;
  end record;

signal X, Y, Z: BYTE_AND_IX;

signal DATA: BYTE_VEC;
signal NUM:  INTEGER;
. . .

X.BYTE <= "11110000";
X.IX   <= 2;

DATA <= Y.BYTE;
NUM  <= Y.IX;

Z <= X;
```

```
X <= (BYTE => "11110000", IX => 2);
```

```
X <= ("11110000", 2);
```

signed, **unsigned** (std_logic_arith)

- defined in package *numeric_std* , or in package *std_logic_arith* from ieee library
- **signed**

  type signed **is array (natural range <>) of** std_logic
- **unsigned**

  type unsigned **is array (natural range <>) of** std_logic
- both **signed** and **unsigned** are numbers but represented as 1D array of bits, similar to bit-vectors
- do not confuse with packages *std_logic_signed* and *std_logic_unsigned*
  - these two packages define signed and unsigned operators (not data types) for std_logic_vector

SIGNED, UNSIGNED vectors

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all; -- package defining signed and unsigned data types
…
SIGNAL a: IN SIGNED (7 DOWNTO 0);   -- a is a number represented as multi-bits
SIGNAL b: IN SIGNED (7 DOWNTO 0);    -- b is a number represented as multi-bits
SIGNAL c: OUT SIGNED (7 DOWNTO 0);
…
v <= a + b;        -- legal (arithmetic operations are OK for SIGNED, UNSIGNED )
w <= a AND b;  -- illegal (logic operations are not OK for signed/unsigned)
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
…
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);   -- a is a vector
SIGNAL b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);    -- b is a vector
SIGNAL c: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
…
v <= a + b;        -- illegal (arithmetic operations are not OK.)
w <= a AND b;  -- legal (logical operations are OK for STD_LOGIC_VECTOR)
```

Type Conversion

```
TYPE long IS INTEGER RANGE -100 TO 100;
TYPE short IS INTEGER RANGE -10 TO 10;
SIGNAL x: short;
SIGNAL y: long;
…
 y <= 2*x + 5;  -- error, type mismatch (type short is assigned to type long)
y <= long(2*x +5); -- OK, converted into type long
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
…
SIGNAL a: IN      UNSIGNED (7 DOWNTO 0);
SIGNAL b: IN      UNSIGNED (7 DOWNTO 0);
SIGNAL y: OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
…
 y <= CONV_STD_LOGIC_VECTOR( (a+b), 8);
-- conv_std_logic_vector () is a function defined in package std_logic_arith
```

Type Conversion (std_logic_arith)

- *std_logic_arith* in library **ieee** provides several data conversion functions
  - **conv_integer(p)**
    - UNSIGNED, SIGNED, STD_ULOGIC (p) -> INTEGER
    - STD_LOGIC is not included
  - conv_unsigned(p, b)
    - INTEGER, SIGNED, STD_ULOGIC (p) -> UNSIGNED of b bits
  - conv_signed(p, b)
    - INTEGER, UNSIGNED, STD_ULOGIC (p) -> SIGNED of b bits
  - conv_std_logic_vector(p, b)
    - INTEGER, UNSIGNED, SIGNED (p) -> STD_LOGIC_VECTOR of size b bits

std_logic_signed (std_logic_unsigned)

- *packages std_logic_signed*, *std_logic_unsigned* from ieee library
  - allow STD_LOGIC_VECTOR data to be performed as if the data were of type SIGNED or UNSIGNED

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all; -- define arithmetic operators for vectors
-- allow unsigned arithmetic operations on std_logic_vector data types
…
SIGNAL a: IN     STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN     STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL c: OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
…
v <= a + b; -- arithmetic operation OK, std_logic_vector viewed as unsigned
w <= a AND b; -- OK
```

numeric_std

- defines **signed** and **unsigned** data types for bit-vectors
  - similar to signed and unsigned types in *std_logic_arith*
- defines logical, arithmetic, comparison, and shift operators on vectors
  - similar to stod_logic_signed, or std_logic_unsigned
  - *std_logic_arith* does NOT define logical operator
- type conversion **to_integer()**, **to_unsigned()**, ..
  - similar to **conv_integer()** , **conv_unsigned()**, in *std_logic_arith* which has a wider set of data-conversion functions

summary on signal, variable, data type

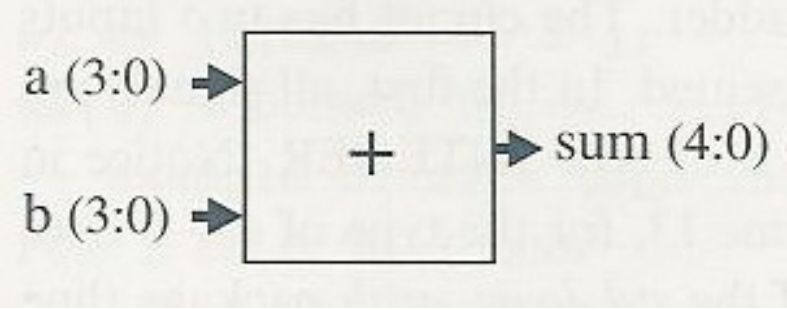
- **signal** means real hardware wires
  - value update (**<=**) is NOT immediate
- **variabl**e is local for easy programming
  - value update (**:=**) is immediate
- logic vector (e.g., std_logic_vector) allows only logic operations
- numbers (e.g., **integer**) allows only arithmetic operations
- signed or unsigned vector (e.g., **signed, unsigned**) allows only arithmetic operations
  - defined in package **ieee.std_logic.arith**
- package **eee.std_logic.signed (ieee.std_logic.unsigned)** allows both logic and arithmetic operations for std_logic vector

Synthesizable VHDL data types

Data types	Synthesizable values
BIT, BIT_VECTOR	'0', '1'
STD_LOGIC, STD_LOGIC_VECTOR	'X', '0', '1', 'Z' (resolved)
STD_ULOGIC, STD_ULOGIC_VECTOR	'X', '0', '1', 'Z' (unresolved)
BOOLEAN	True, False
NATURAL	From 0 to +2, 147, 483, 647
INTEGER	From −2,147,483,647 to +2,147,483,647
SIGNED	From −2,147,483,647 to +2,147,483,647
UNSIGNED	From 0 to +2,147,483,647
User-defined integer type	Subset of INTEGER
User-defined enumerated type	Collection enumerated by user
SUBTYPE	Subset of any type (pre- or user-defined)
ARRAY	Single-type collection of any type above
RECORD	Multiple-type collection of any types above

Example : adder

```
----- solution 1: in/out = SIGNED -----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
---------------------------------------
ENTITY adder1 IS
PORT
(a, b : IN SIGNED (3 DOWNTO 0);
sum : OUT SIGNED (4 DOWNTO 0));
END adder1;
---------------------------------------
ARCHITECTURE adder1 OF adder1 IS
BEGIN
        sum <= a + b;
END adder1;
```




```
------ solution 2: out=INTEGER ------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
---------------------------------------
ENTITY adder2 IS
PORT
(a, b : IN SIGNED (3 DOWNTO 0);
sum : OUT INTEGER RANGE -16 TO 15);
END adder2;
---------------------------------------
ARCHITECTURE adder2 OF adder2 IS
BEGIN
        sum <= CONV_INTEGER(a + b);
END adder2;
```

summary of data objects/types

- Verilog synthesizable
  - wire, signed, unsigned, reg, integer
  - bit vector, array
  - bit vectors can be considered as numbers too
    - e.g., wire [7:0] x [0:255][0:255]; // 2D array of 8-bit elements
      // MSB of element [12][34] is x[12][34][7]
- VHDL synthesizable
  - signal, variable, bit, std_logic, integer
  - bit_vector, std_logic_vector, array
  - operation and data types should match
    - e.g., signal x: array (0 to 255, 0 to 255) of std_logic vector (7 downto 0); -- 2D array of 8-bit elements
      -- MSB of element (12, 134) is leftmost bit x(12, 34)(7)

std_logic_1164, arith, unsigned/signed

- std_logic_1164
  - defined 8-level logic std_logic/std_logic_vector
  - bit-vector is NOT allowed for arithmetic operations
- std_logic_arith
  - define unsigned/signed bit-vector data type
  - unsigned/signed is NOT allowed for logical operation
- std_logic_unsigned/signed
  - define arithmetic operations for bit-vectors
  - bit-vectors are allowed forboth logical and arithmetic operations

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
…
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);   -- a is vector
SIGNAL b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);    -- b is vector
SIGNAL c: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
…
v <= a + b;        -- illegal (arithmetic operations are not OK.)
w <= a AND b;  -- legal (logical operations are OK for STD_LOGIC_VECTOR)
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all; -- package defining signed and unsigned data types
…
SIGNAL a: IN SIGNED (7 DOWNTO 0);   -- a is number, represented as multi-bits
SIGNAL b: IN SIGNED (7 DOWNTO 0);    -- b is number, represented as multi-bits
SIGNAL c: OUT SIGNED (7 DOWNTO 0);

v <= a + b;        -- legal (arithmetic operations are OK for SIGNED, UNSIGNED )
w <= a AND b;  -- illegal (logic operations are not OK for signed/unsigned)
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all; -- or USE ieee.std_logic_signed.all
-- allow unsigned (or signed) arithmetic operations on std_logic_vector data types
…
SIGNAL a: IN     STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN     STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL c: OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
…
v <= a + b; -- arithmetic operation OK, std_logic_vector viewed as unsigned
w <= a AND b; -- OK
```

Operators and Attributes

Operators

- assignment (**<=, :=, =>**)
- logical (**NOT, AND, OR, NAND, NOR, XOR, XNOR**)
  - Verilog logical operators (~, &, |, ^,…)
- arithmetic **(+, -, *, /, **, MOD, REM, ABS**)
  - Verilog (**+, -, *, /, **, %, …)**
- relational (**=, /=, <, >, <=, >=**)
  - Verilog (==, !=, …)
- shift (**sll, srl, sla, sra, rol, ror**)
  - Verilog (<<, >>, <<<, >>>, …)
- concatenation (**&, (,,,)**)
  - Verilog **{,,,}**

assignment operator

<= assign a value to a **SIGNAL**

eg. x **<=** '1'; -- x is declared as a SIGNAL

:= assign a value to **VARIABLE**, CONSTANT, GENERIC, or initialization

e.g.1. y **:=** "0000"; -- y is declared as a VARIABLE

e.g.2. GENERIC (n : INTEGER **:=** 8);

e.g.3. CONSTANT set_bit : BIT **:=** '1';

=> assign values to individual vector elements or with **OTHERS**

eg. **SIGNAL** w : STD_LOGIC_VECTOR (0 to 7);

w <= (0 **=>** '1', **OTHERS** => '0'); -- w <= "10000000"

-- w(0) is '1', the others are '0', MSB is rightmost bit w(7)

w <= (0|1 **=>** '1', **OTHERS** => '0'); -- w <= "11000000"

-- w(0)=w(1)='1', the others are '0'

examples of other operators

y <= **NOT** a **AND** b; -- in Verilog: ~a & b

y <= **NOT** (a **AND** b); -- in Verilog: ~(a & b)

x <= “01001”;

y <= x **sll** 2; -- logic shift to eft by 2, y <= “00100”

y <= x **sla** 2; -- arithmetic shift to left by 2, y <= “00111”

y <= x **srl** 3; -- logic shift to right by 3, y <= “00001”

y <= x **sra** 3; -- arithmetic shift to right by 3, y<= “00001”

y <= x **rol** 2; -- logic rotation to left by 2, y <= “00101”

z <= x **&** “1000000”; -- z <= “1_1000000” if x=‘1’;

z **<= (**‘1’**,** ‘1’**,** ‘0’**,** ‘0’, ‘0’, ‘0’, ‘0’, ‘0’**)**; -- z <= “11000000”

attributes of regular data types

TYPE negative **IS RANGE** I**NTEGER'LOW TO** -1;
SYBTYPE natural **IS INTEGER RANGE** 0 **TO INTEGER'HIGH**

-- data attribute return a value of a data type
SIGNAL d: STD_LOGIC_VECTOR (7 **DOWNTO** 0);
d**'LOW**=0; -- return lower array index,
d**'HIGH**=7; -- return upper array index
d**'LEFT**=7; -- return leftmost array index
d**'RIGHT**=0; -- return rightmost array index
d**'LENGTH**=8; -- return vector size
d**'RANGE**=(7 DOWNTO 0); -- return vector range
d**'REVERSE_RANGE**=(0 TO 7); -- return vector range in reverse order

```
SIGNAL x: STD_LOGIC_VECTOR (0 TO 7);

FOR i 0 TO 7 LOOP …
FOR i IN x'RANGE LOOP …
FOR i IN RANGE (x'LOW TO x'HIGH) LOOP …
FOR i IN RANGE (0 TO x'LENGTH-1) LOOP …
```

attributes of enumerate type

-- enumerate type

d'**VAL** (pos): return value in the position specified

d'**POS**(value): return postion of the value specified

d'**LEFTOF** (value): return value in the position to the left of the value specified

d'**VAL** (row, column): return value in the position specified

Example:

type logic_level **is** (unknown, low, undriven, high);

logic_level'**pos**(unknown) = 0; -- position number

logic_level'**val**(3) = high; -- value at a postion

logic_level'**leftof**(undriven) = low; -- value to the left

logic_level'**succ**(undriven) = high; -- value at the succeeding position

logic_level'**prec**(undriven) = low; -- value at the preceeding position

signal attributes

-- signal attribute return true or false, or specific value of a signal

s'EVENT -- return true when an event occurs on signal s (synthesizable)
s'STABLE -- return true if no event has occurred on s (synthesizable)
s'ACTIVE -- return true if s = '1'
s'QUIET <time> -- return true if no event has occurred during the time specified

s'LAST_EVENT – return the time elapsed since last event
s'LAST_ACTIVE – return the time elapsed since last s = '1'
s'LAST_VALUE – return the value of s before the last event

```
IF (clk'EVENT AND clk='1') …  -- cp. posedge(clk) in Verilog
IF (NOT clk'STABLE AND clk='1') …
WAIT UNTIL (clk'EVENT AND clk='1');
IF RISING_EDGE(clk) … -- call to a function RISING_EDGE
```

user-defined attributes

ATTRIBUTE attribute_name: attribute_type; -- declaration

ATTRIBUTE attribute_name of target_name: class **IS** value;
-- specification

example:

ATTRIBUTE number_of_inputs: ***INTEGER***; -- declaration

ATTRIBUTE number_of_inputs **OF** nand3: **SIGNAL IS** 3; --
specification

...

inputs <= nand3'number_of_inputs: -- attribute call, return 3

operator overloading

```
FUNCTION "+" (a: INTEGER, b:BIT) RETURN INTEGER IS
BEGIN
    IF (b='1') THEN RETURN a+1;
    ELSE RETURN a;
END "+";

SIGNAL inp1, outp: INTEGER RANGE 0 TO 15;
SIGNAL inp2: BIT;
…
outp <= 3 + inp1 + inp2;
-- the first "+" is predefined and the second "+" is overloaded,
-- i.e., the second "+" is defined by the above function
```

GENERIC

- specify a generic parameter for different applications
  - static parameter that can be easily modified during component instantiation, using GENERIC MAP(new_value), similar to **#**(new_value) in Verilog module instantiation)
  - more flexibility and reusability
  - similar to **parameter** in Verilog
- must be declared in the ENTITY
  - ENTITY contains declaration of port and generic
- the specified parameter is global

```
ENTITY my_entity IS
        GENERIC (n: INTEGER :=8);
        PORT (
          … : IN …. ;
          … : OUT …
        );
END my entity;
```

```
module
# (parameter n=8)
my_entity
(input … ,
output …);
…
endmodule
```

summary of operators

- non-synthesizable constructs are marked

Operators.

Operator type	Operators	Data types
Assignment	<=, :=, =>	Any
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	+, –, *, /, ** (mod, rem, abs)◆	INTEGER, SIGNED, UNSIGNED
Comparison	=, /=, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (,,,)	Same as for logical operators, plus SIGNED and UNSIGNED

summary of attributes

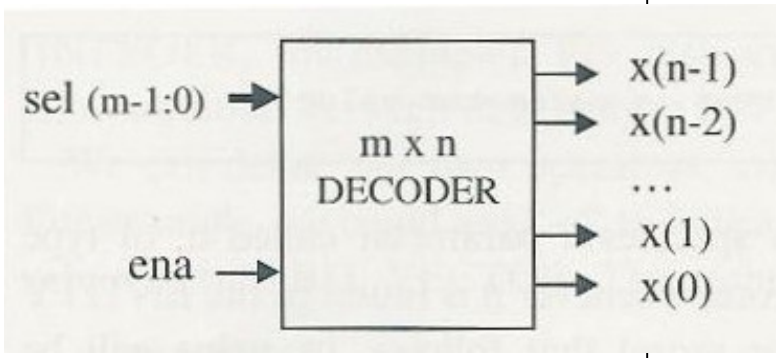
- non-synthesizable constrcuts are marked

Attributes.

Application	Attributes	Return value
For regular DATA	d'LOW	Lower array index
	d'HIGH	Upper array index
	d'LEFT	Leftmost array index
	d'RIGHT	Rightmost array index
	d'LENGTH	Vector size
	d'RANGE	Vector range
	d'REVERSE_RANGE	Reverse vector range
For enumerated DATA	d'VAL(pos)◆	Value in the position specified
	d'POS(value)◆	Position of the value specified
	d'LEFTOF(value)◆	Value in the position to the left of the value specified
	d'VAL(row, column)◆	Value in the position specified
For a SIGNAL	s'EVENT	True when an event occurs on s
	s'STABLE	True if no event has occurred on s
	s'ACTIVE◆	True if s is high

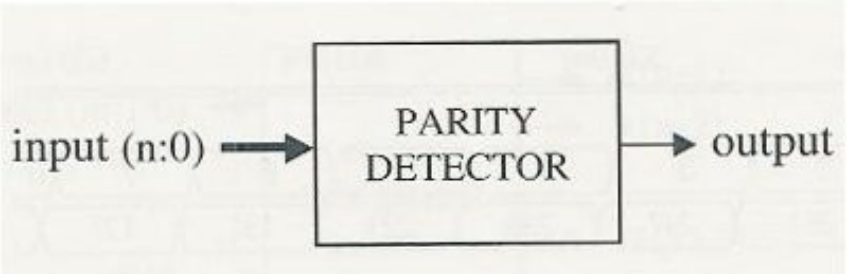
Example: 3-to-8 decoder (active low)

```
ENTITY decoder IS
GENERIC (m : INTEGER := 3);
PORT (ena : IN      STD_LOGIC; sel : IN STD_LOGIC_VECTOR (m-1 DOWNTO 0);
            x : OUT  STD_LOGIC_VECTOR (2**m -1  DOWNTO 0)  );      END decoder;
-----------------------------------------
ARCHITECTURE generic_decoder OF decoder IS BEGIN
PROCESS (ena, sel)
  VARIABLE temp1 : STD_LOGIC_VECTOR (x'HIGH DOWNTO 0);
  VARIABLE temp2 : INTEGER RANGE 0 TO x'HIGH;
BEGIN
temp1 := (OTHERS => '1');  -- initialize all output bits to be '1'
temp2 := 0;
IF (ena='1') THEN
            FOR i IN sel'RANGE LOOP    -- sel range is 2 downto 0
                        IF (sel(i)='1') THEN  -- binary-to-Integer conversion
                                    temp2:=2*temp2+1;
                        ELSE
                                    temp2 := 2*temp2;
                        END IF;
            END LOOP;
            temp1(temp2):='0'; -- active low
END IF;
x <= temp1; -- assign variable to signal
END PROCESS;   END generic_decoder;
```



Example: parity detector

```
ENTITY parity_det IS
    GENERIC (n : INTEGER := 7);
    PORT (input: IN BIT_VECTOR (n DOWNTO 0);
          output: OUT BIT);
END parity_det;
-----------------------------------------
ARCHITECTURE parity OF parity_det IS
BEGIN
    PROCESS (input)
        VARIABLE temp: BIT;
    BEGIN
        temp := '0';
        FOR i IN input'RANGE LOOP
            temp := temp XOR input(i);
        END LOOP;
        output <= temp;
    END PROCESS;
END parity;
```




```
ARCHITECTURE …
…
-- component declaration
COMPONENT parity_det IS
 GENERIC (n : INTEGER := 7);
  PORT
  (input: IN BIT_VECTOR (n DOWNTO 0);
    output: OUT BIT);
END COMPONENT;
…
-- component instantiation in VHDL
C1 : parity_det
        GENERIC MAP (15);
        PORT MAP (inp, outp);
...
```

Example: even parity generator

```
ENTITY parity_gen IS
    GENERIC (n : INTEGER := 7);
    PORT (input: IN BIT_VECTOR (n-1 DOWNTO 0);
          output: OUT BIT_VECTOR (n DOWNTO 0)); -- one more  bit
END parity_gen;
-------------------------------------------
ARCHITECTURE parity OF parity_gen IS
BEGIN
    PROCESS (input)
        VARIABLE i: INTEGER;
        VARIABLE temp1: BIT;
        VARIABLE temp2: BIT_VECTOR (output'RANGE);
    BEGIN
        temp1 := '0';
        FOR i IN input'RANGE LOOP
            temp1 := temp1 XOR input(i);
            temp2(i) := input(i);
        END LOOP;
        temp2(output'HIGH) := temp1;
        output <= temp2;
    END PROCESS;
END parity;
```