VHDL – Hierarchy

- **Structuring the design**
  - **models are easier to read**
  - **sub-models can be reused**
  - **design and verification are more manageable**

structural granularity	structural modelling unit	VHDL construct
coarse	entity / architecture pairing	configuration
coarse	primary design unit	entity / architecture
coarse/medium	replication of concurrent statements	for / if - generate
coarse/medium	grouping of concurrent statements	block
medium	grouping of sequential statements	process
fine	subprogram	procedure / function

  - **Modularity features – functions & procedures**
  - **Partitioning features – libraries, packages, components, blocks, configurations, ...**

Libraries & Packages

- **A design library is an implementation-dependent storage facility for previously analyzed design units**

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

- **Packages**
  - **Package declaration & body (necessary in the case of subprograms)**
  - **Deferred constant (declared in package, assigned in package body)**
  - **The “use” clause**
  - **Signals in packages (global signals)**
  - **Resolution function in packages**
  - **Subprograms in packages**

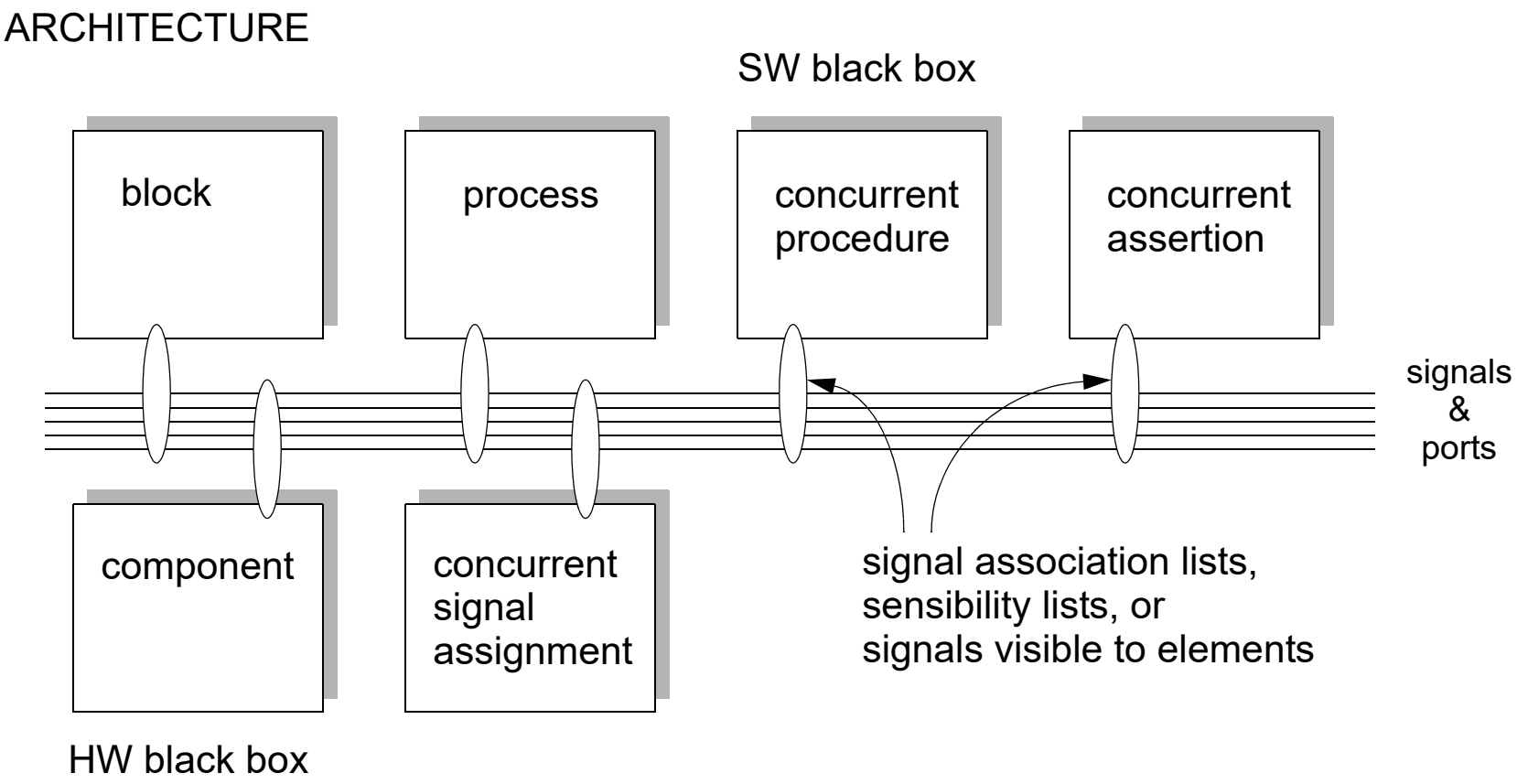
Architecture

- Architecture – declarations, concurrent statements
  - Process statement
  - Concurrent signal assignment
  - Component instantiation statement
  - Concurrent procedure call
  - Generate statement
  - Concurrent assertion statement
  - Block statement

Process

- Process – declarations, sequential statements
  - Behavior of the model
  - Contains timing control
    - either wait statement(s) or sensitivity list must exist in a process
  - Concurrent statement (data-flow statement) == process with sensitivity list

Concurrent elements in architecture



Blocks

- The blocks are used for enhancing readability
- Blocks can be nested & can define local declarations
  - hiding same names declared outside of block

```
BLOCK1: block
    signal a,b: std_logic;
begin
    ...
end block BLOCK1;
```

- Guarded blocks – additional expression known as *guard expression*
  - Used for describing latches and output enables in dataflow style

```
architecture mylatch of latch is
begin
    L1: block (LE='1')
    begin
        Q    <= guarded D after 5 ns;
        QBar <= guarded not(D) after 7 ns;
    end block L1;
end mylatch;
```

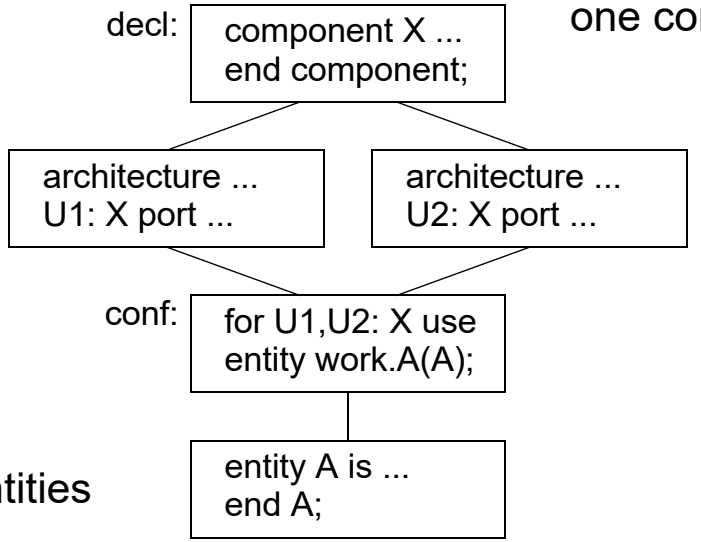
  - Remark: Guarded blocks are not supported by all synthesis tools

Configurations

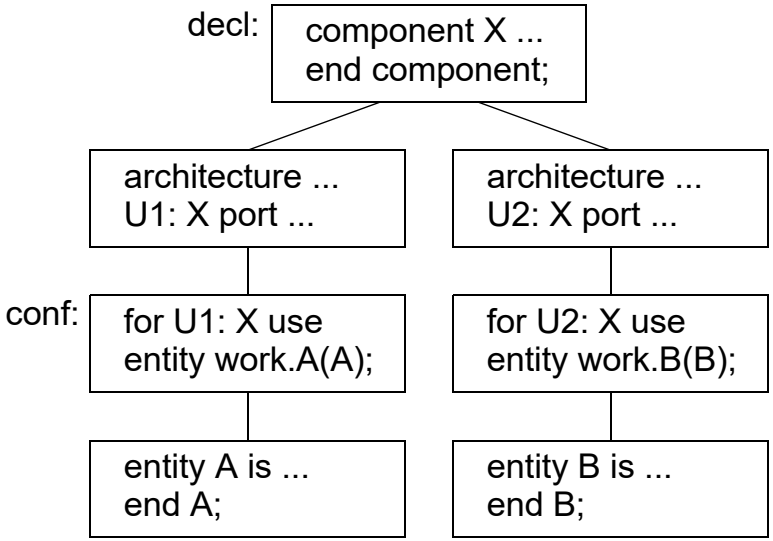
- Configuration is a separate *design unit* which allows different architecture and component bindings to be specified after a model has been analyzed and compiled
- Two types
  - Configuration declaration
    - binds the *entity* to particular *architecture body*
    - binds components, used in the specified *architecture* to a particular *entity*
    - binds component statements, used in the specified *architecture*, to particular *configuration* statements
  - Configuration specification
    - used to specify the binding of component instances to a particular *entity-architecture* pair

Using configuration – examples

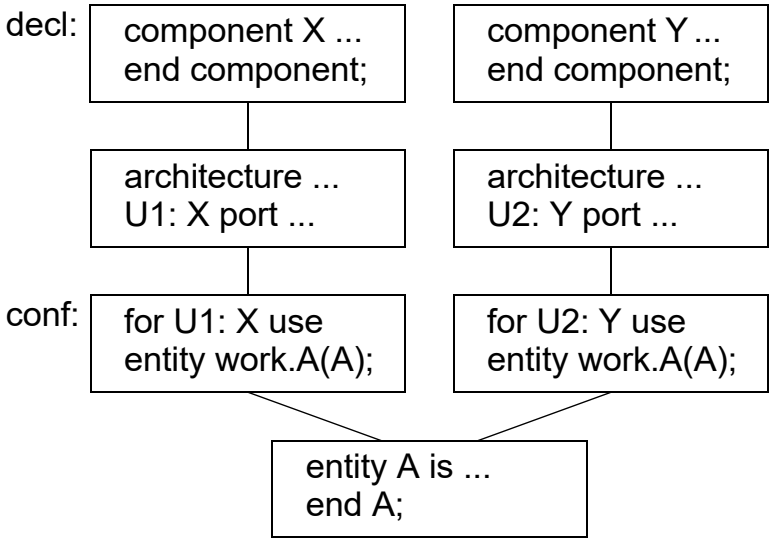
one component -- one entity



one component -- two entities



two components -- one entity



Example #2 (http://www10.edacafe.com/book/ASIC/ASICs.php)

- components

```
entity AD2 is port (A1, A2: in BIT; Y: out BIT); end;
architecture B of AD2 is begin Y <= A1 and A2; end;
entity XR2 is port (X1, X2: in BIT; Y: out BIT); end;
architecture B of XR2 is begin Y <= X1 xor X2; end;
```

- component declaration & configuration specification

```
entity Half_Adder is port (X, Y: BIT; Sum, Cout: out BIT); end;
architecture Netlist of Half_Adder is use work.all;
  component MX port (A, B: BIT; Z:out BIT); end component;
  component MA port (A, B: BIT; Z:out BIT); end component;
  for G1:MX use entity XR2(B) port map(X1 => A,X2 => B,Y => Z);
begin
  G1:MX port map (X, Y, Sum); G2:MA port map (X, Y, Cout);
end;
```

- configuration declaration, block configuration, component configuration

```
configuration C1 of Half_Adder is
  use work.all;
  for Netlist
    for G2:MA
      use entity AD2(B) port map(A1 => A,A2 => B,Y => Z);
    end for;
  end for;
end;
```

Behavioral hierarchy – functions & procedures

- Function
  - used as an expression – can not contain timing control statements
  - input parameters only (as constants)
  - operator overloading
  - resolution functions – multiple drivers of a signal
- Procedure
  - used as a statement (sequential or concurrent) – can contain timing control statements
  - input parameters (constants)
  - output parameters (variables/signals)
- Declaration (prototype)
  - package or declarative part of architecture, process, function, procedure, etc.
- Content (body)
  - package body
  - declarative part of architecture, process, function, etc. (together with declaration)

Functions

```
-- Conversion version example
function conv_boolean (a: signed) return boolean is begin
  if to_bit(a(a'low))='1' then return TRUE; else return FALSE; end if;
end conv_boolean;

-- Operator overloading example
function "and" (l,r: signed) return signed is begin
  return signed(std_logic_vector(l) and std_logic_vector(r));
end;


-- Architecture - declarative part
  signal a, b, x: signed (7 downto 0);
  signal y: boolean;

-- Architecture/process/... - body
  X <= a and b;
  -- ...
  y <= conv_boolean(a);
```

Procedures

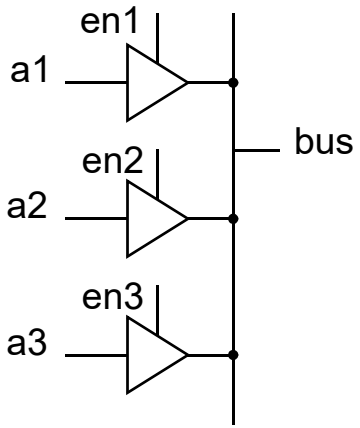
```
PACKAGE adder_elements IS
--  full_adder :  1-bit full adder  (declaration)
PROCEDURE full_adder (CONSTANT a0, b0, c0: IN bit; VARIABLE o0, c1: OUT bit);
END adder_elements;

PACKAGE BODY adder_elements IS
PROCEDURE half_adder (CONSTANT a0, b0: IN bit; VARIABLE o0, c1: OUT bit) IS
BEGIN
  o0 := a0 XOR b0;    c1 := a0 AND b0;
END half_adder;

PROCEDURE full_adder (CONSTANT a0, b0, c0: IN bit; VARIABLE o0, c1: OUT bit) IS
  VARIABLE c_1, c_2, o_1: bit;
BEGIN
  half_adder ( a0, b0, o_1, c_1 );
  half_adder ( o_1, c0, o0, c_2 );
  c1 := c_1 or c_2;
END full_adder;
END adder_elements;
```

Resolution functions

- **Multiple drivers of a signal**

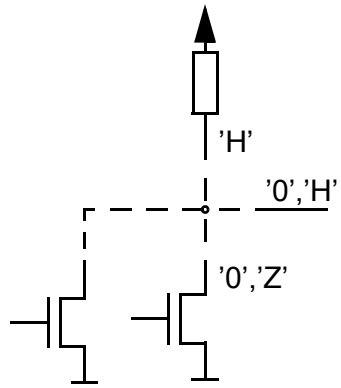



```
type tri_value is ('0','1','Z');
type tri_val_vector is array (natural range <>) of tri_value;
function resolve_tri (b:tri_val_vector) return tri_value is ...
    ...
signal bus: resolve_tri tri_value;
```

- **Driving values – a sequence of pairs: (value,time)**
- **Ports**

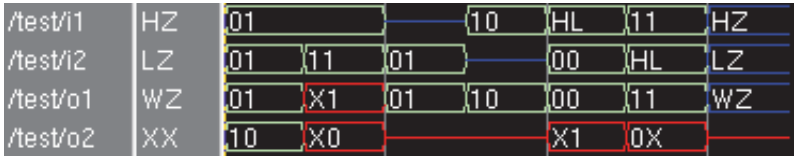
Resolution function – I²C

```
package I2C_defs is
  type I2C_bit is ( '0', 'Z', 'H' );
  type I2C_bit_vector is array (integer range <>) of I2C_bit;
  function resolved ( v: I2C_bit_vector ) return I2C_bit;
  -- ...
end I2C_defs;
package body I2C_defs is
  function resolved ( v: I2C_bit_vector ) return I2C_bit is
    variable r: I2C_bit := 'Z';
    type I2C_1d is array ( I2C_bit ) of I2C_bit;
    type I2C_2d is array ( I2C_bit ) of I2C_1d;
    constant resolution_table: I2C_2d := (
    ------------------------------
    -- '0'  'Z'  'H'
    ------------------------------
     ( '0', '0', '0' ),   -- '0'
     ( '0', 'Z', 'H' ),   -- 'Z'
     ( '0', 'H', 'H' ) ); -- 'H'
  begin
    for i in v'range loop    r := resolution_table ( r ) ( v(i) );    end loop;
    return r;
  end resolved;
  -- ...
end I2C_defs;
```



Resolution function – how to use

```
library IEEE;    use IEEE.std_logic_1164.all;
entity driver is
  port ( i1: in std_logic_vector(1 downto 0);
         o1, o2: inout std_logic_vector(1 downto 0) );
end entity driver;
library IEEE;    use IEEE.std_logic_1164.all;
architecture driving of driver is begin
  o1 <= i1;    o2 <= not i1;
end architecture driving;
-------------------------------------------------------------------
entity test is    end entity test;
library IEEE;    use IEEE.std_logic_1164.all;
architecture bench of test is
  signal i1, i2, o1, o2: std_logic_vector(1 downto 0);
  component driver
    port ( i1: in std_logic_vector(1 downto 0);
           o1, o2: inout std_logic_vector(1 downto 0) );
  end component;
begin
  process begin
    i1 <= "01";    i2 <= "01";    wait for 10 ns;
    i1 <= "01";    i2 <= "11";    wait for 10 ns;
    i1 <= "ZZ";    i2 <= "01";    wait for 10 ns;
    i1 <= "10";    i2 <= "ZZ";    wait for 10 ns;
    i1 <= "HL";    i2 <= "00";    wait for 10 ns;
    i1 <= "11";    i2 <= "HL";    wait for 10 ns;
    i1 <= "HZ";    i2 <= "LZ";    wait for 10 ns; wait;
  end process;
  u1: driver port map (i1, o1, o2);
  u2: driver port map (i2, o1, o2);
end architecture bench;
```



Composite types – array & record

- *Array* – all elements are of the same (sub)type, multiple dimension possible

```
subtype X01_Typ is STD_Logic range 'X' to '1';
type Vect8_Typ is array (7 downto 0) of X01_Typ;
type IntArr is array (integer range <>) of integer;
```

- *Record* – consist of named elements with arbitrary types

```
type Task_typ is record
  Task_Numb   : integer;
  Time_tag    : time;
  Task_mode   : Task_mode_Typ;
end record;
```

Alias – part of a type/variable/signal

```
variable Nmbr   : BIT_VECTOR (31 downto 0);
alias Sign      : BIT is Nmbr(31);
alias Mantissa : BIT_VECTOR (23 downto 0) is Nmbr (30 downto 7);
alias Exponent : BIT_VECTOR ( 6 downto 0) is Nmbr ( 6 downto 0);
```

Composite type – examples

- **Defining 3D array (hyper-cube)**

```
type look_up_1 is array (bit) of bit;
type look_up_2 is array (bit) of look_up_1;
type look_up_3 is array (bit) of look_up_2;
constant output_a: look_up_3 :=
  ( ( ( '0', '1' ), ( '1', '0' ) ), ( ( '1', '0' ), ( '0', '1' ) ) );
type look_up_b is array (bit,bit,bit) of bit;
constant output_b: look_up_b :=
  ( ( ( '0', '1' ), ( '1', '0' ) ), ( ( '1', '0' ), ( '0', '1' ) ) );
signal s_a: bit := output_a('1')('0')('1');
signal s_x: look_up_1 := output_a('1')('0');
signal s_b: bit := output_b('1','0','1');
```

- **Record**

```
type my_rec is record   num: integer;   tm: time;   end record;
constant rec1: my_rec := ( 10, 5 ns );
signal s_n: integer := rec1.num;
signal s_t: time := rec1.tm;
```