Technische
Hochschule
Augsburg

VLSI Design

VHDL Fundamentals

J Färber

SS2026

Contents

VHDL Fundamentals

Learning Objectives

After completing this unit, you will be able to:

- Describe a digital circuit using VHDL `ENTITY` and `ARCHITECTURE`
- Use the fundamental data types `std_ulogic` and `std_ulogic_vector`
- Declare internal `SIGNAL`s and use them in concurrent assignments
- Use arithmetic types `unsigned` and `signed` from `ieee.numeric_std`
- Perform type conversions between `std_ulogic_vector`, `unsigned`, and `signed`
- Structure a design into a hierarchy of modules using component instantiation
- Write a self-checking testbench with stimuli generation and `ASSERT`

VHDL Design Unit: Entity and Architecture

Every VHDL design consists of two parts:

- The **`ENTITY`** describes the interface — inputs and outputs (ports)
- The **`ARCHITECTURE`** describes the function — what the circuit does

Think of the entity as the circuit symbol, and the architecture as the schematic inside it.

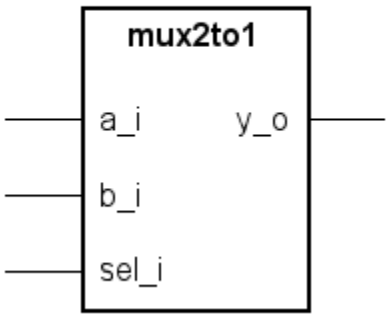


Figure 1: mux2to1 - Schematic Symbol

Library and Package Declarations

Before every VHDL file, include the standard IEEE library:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

For arithmetic operations, also include:

```
USE IEEE.numeric_std.ALL;
```

ENTITY Declaration

Referring to [Mea25], chap. 3.1 Entity

The entity declares all ports with their direction and data type:

```
ENTITY mux2to1 IS
  PORT (
    a_i   : IN  std_ulogic;   -- data input A (selected when sel_i = '1')
    b_i   : IN  std_ulogic;   -- data input B (selected when sel_i = '0')
    sel_i : IN  std_ulogic;   -- select: '1' → a_i,  '0' → b_i
    y_o   : OUT std_ulogic
  );
END mux2to1;
```

Port directions:

Direction	Meaning
`IN`	Input — can only be read inside the architecture
OUT	Output — can only be assigned inside the architecture

> **Naming convention:** input ports end with `_i`, output ports with `_o`.

ARCHITECTURE Body

Referring to [Mea25], chap. 3.3 Architecture

Figure 2: mux2to1 - Schematic

The architecture defines the circuit behaviour. All statements inside `BEGIN ... END` execute **concurrently** — they model hardware, not a sequential program.

```
ARCHITECTURE equation OF mux2to1 IS

BEGIN

  y_o <= (sel_i AND a_i) OR (NOT sel_i AND b_i);

END equation;
```

The label after `OF` must match the entity name exactly. The label after `IS` (here: `equation`) names the architecture. Multiple architectures can exist for the same entity.

Data Types

`std_ulogic` — Single-Bit Signal

`std_ulogic` models a single digital signal. The most important values in synthesis and simulation:

Value	Meaning
`'0'`	Logic zero
`'1'`	Logic one
`'U'`	Uninitialised (simulation only)
`'X'`	Unknown / conflict (simulation only)

`std_ulogic_vector` — Multi-Bit Signal

An array of `std_ulogic` elements — used for buses and registers:

```
SIGNAL data : std_ulogic_vector(7 DOWNTO 0);  -- 8-bit bus
SIGNAL addr : std_ulogic_vector(3 DOWNTO 0);  -- 4-bit address
SIGNAL reg  : std_ulogic_vector(7 DOWNTO 0);  -- 8-bit bus
```

Index convention: `(n-1 DOWNTO 0)` - MSB on the left, LSB on the right.

Assigning individual bits:

```
data(7) <= '1';                    -- assign MSB
data    <= "10110100";             -- assign all 8 bits (binary literal)
data    <= X"B4";                  -- same value as hex literal
address <= X"A";                   -- assign a 4-Bit value
data    <= (OTHERS => '0');        -- assign all bits to '0'
```

Concatenation with &:

```
data <= "0000" & addr;             -- prepend four zeros to addr
reg  <= '1' & data(7 DOWNTO 1;)    -- shift right, injecting a '1' at MSB
reg  <= data(0) & data(7 DOWNTO 1); -- rotate right
```

Internal Signals

Signals declared in the architecture declaration area are internal wires — they connect concurrent statements and instances (see chapter: Design Hierarchy) within the architecture.

The `mux2to1` can use an internal signal to hold the inverted select, making the boolean equation more readable:

```
ARCHITECTURE rtl OF mux2to1 IS

  SIGNAL not_sel_s : std_ulogic;   -- inverted select

BEGIN

  not_sel_s <= NOT sel_i;
  y_o       <= (sel_i AND a_i) OR (not_sel_s AND b_i);

END rtl;
```

Both architectures (`equation` and `rtl`) describe the same circuit — the synthesiser produces identical hardware from either.

Arithmetic Types from `ieee.numeric_std`

For arithmetic operations (addition, subtraction, comparison), `std_ulogic_vector` must be converted to an arithmetic type.

Type	Meaning	Example
`unsigned`	Unsigned binary number	`"0101"` = 5
`signed`	Two's complement signed number	`"1101"` = −3

Arithmetic Operations

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY add4 IS
  PORT (a_i   : IN  std_ulogic_vector(3 DOWNTO 0);  -- operand a
        b_i   : IN  std_ulogic_vector(3 DOWNTO 0);  -- operand b
        sum_o : OUT std_ulogic_vector(3 DOWNTO 0);  -- sum
        co_o  : OUT std_ulogic                      -- carry out
        );
END add4;

ARCHITECTURE rtl OF adder IS

  SIGNAL a, b, sum : unsigned(3 DOWNTO 0);

BEGIN

  a   <= unsigned(a_i);           -- convert from std_ulogic_vector
  b   <= unsigned(b_i);
  sum <= a + b;                   -- arithmetic addition
  y_o <= std_ulogic_vector(sum);  -- convert back

END rtl;
```

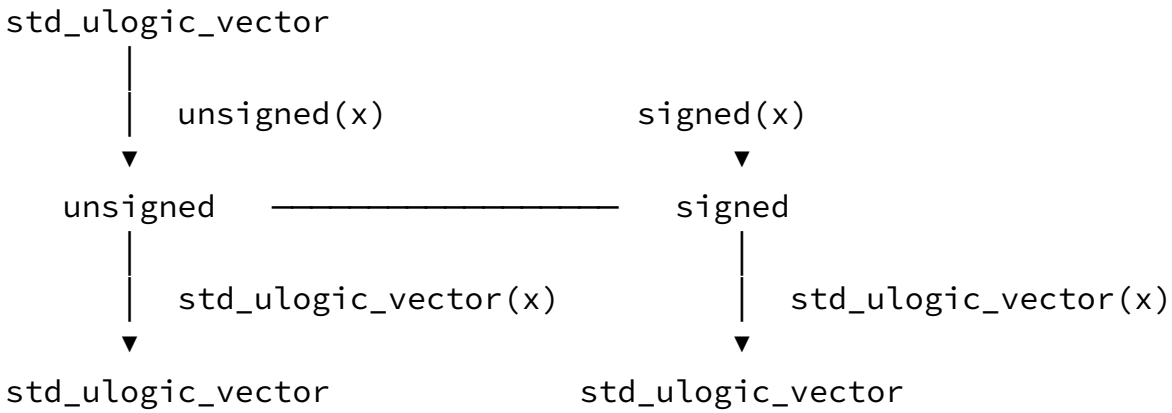
`signed` arithmetic works the same way — and integer literals can be mixed directly with `signed` or `unsigned` operands:

```
SIGNAL a, b   : std_ulogic_vector(3 DOWNTO 0);
SIGNAL sa, sb : signed(3 DOWNTO 0);
SIGNAL sc     : signed(7 DOWNTO 0);
SIGNAL sd     : signed(7 DOWNTO 0);

sc <= sa + sb;       -- addition,    result assigned to wider signal
sc <= sa + 5;        -- signed + integer literal
sc <= sa + (-5);     -- negative integer literal
sc <= -5 + sa;       -- swapped argument order is allowed
sc <= sa - sb;       -- subtraction
sd <= sa * sb;       -- multiplication: result width = sa'length + sb'length
```

> **Multiplication result width:** `sa` and `sb` are 4 bits each — the product can be up to 8 bits wide, so `sd` must be declared as `signed(7 DOWNTO 0)` to hold the full result without truncation.

Type Conversion Overview



Common conversion patterns:

```
-- std_ulogic_vector → unsigned → arithmetic → std_ulogic_vector
SIGNAL a_u   : unsigned(3 DOWNTO 0);
SIGNAL sum_u : unsigned(3 DOWNTO 0);

a_u   <= unsigned(a_i);
sum_u <= a_u + 1;
y_o   <= std_ulogic_vector(sum_u);
```

The **`'length`** attribute returns the bit width of a signal and avoids hardcoding numbers:

```
SIGNAL a, b   : std_ulogic_vector(3 DOWNTO 0);
SIGNAL sa, sb : signed(3 DOWNTO 0);
SIGNAL sc     : signed(7 DOWNTO 0);
SIGNAL ua, ub : unsigned(3 DOWNTO 0);
SIGNAL i      : integer RANGE -30 TO 30;

sa <= signed(a);                  -- typecast: std_ulogic_vector → signed
a  <= std_ulogic_vector(sa);      -- typecast: signed → std_ulogic_vector
ua <= unsigned(a);                -- typecast: std_ulogic_vector → unsigned

i  <= to_integer(sa);             -- convert signed → integer (e.g. for indexing)
sa <= to_signed(-2, sa'length);   -- convert integer → signed, width from 'length

sc <= resize(sa, sc'length);      -- sign extension: 4-bit signed → 8-bit signed
```

> **`resize` and sign extension:** for `signed`, `resize` to a wider type replicates the MSB (sign bit) into the new upper bits — this preserves the two's complement value. For `unsigned`, the upper bits are filled with zeros.
>
> **`INTEGER RANGE`:** the RANGE constraint limits the integer value range. This allows the

> synthesiser to infer a minimal number of bits rather than allocating a full 32-bit register.

Shift Operations

`shift_right` and `shift_left` from `ieee.numeric_std` perform **arithmetic** shifts on `signed` or `unsigned` signals:

```
sa <= shift_right(sa, 1);   -- arithmetic right shift by 1 bit  (÷ 2)
sa <= shift_left(sa, 1);    -- arithmetic left  shift by 1 bit  (× 2)
```

> **Arithmetic vs. logical shift:** for `signed`, `shift_right` fills the vacated MSB positions with the **sign bit**, preserving the two's complement value. `shift_left` fills from the right with `'0'`. These differ from the VHDL operators `srl`/`sll`, which always fill with `'0'`.

Design Hierarchy

Referring to [Mea25], chap. 9.1

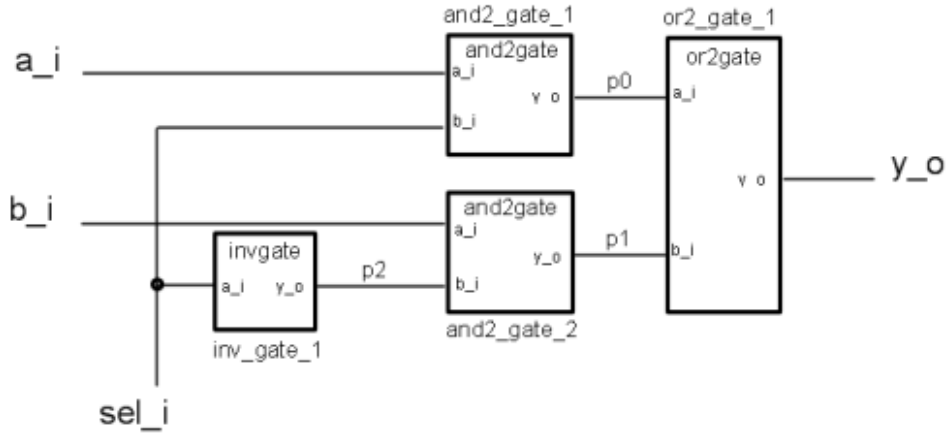


Figure 3: 2-zu-1 Multiplexer - Schematic

Module Hierarchy

```
ENTITY(ARCHITECTURE)


t_mux2to1(tbench)                de1_mux2to1(structure)
  mux2to1(structure)               mux2to1(structure)
    and2gate(equation)               and2gate(equation)
    or2gate(equation)                or2gate(equation)
    invgate(equation)                invgate(equation)
```

Filenames

```
|
+---src
       and2gate_equation.vhd
       invgate_equation.vhd
       or2gate_equation.vhd
       mux2to1_structure.vhd
       t_mux2to1.vhd
       de1_mux2to1_structure.vhd
```

Component Instantiation

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY mux2to1 IS
  PORT (a_i   : IN  std_ulogic;
        b_i   : IN  std_ulogic;
        sel_i : IN  std_ulogic;
        y_o   : OUT std_ulogic
        );
END mux2to1;

ARCHITECTURE structure OF mux2to1 IS

  COMPONENT invgate
    PORT (
      a_i : IN  std_ulogic;
      y_o : OUT std_ulogic);
  END COMPONENT;

  COMPONENT and2gate
    PORT (
      a_i : IN  std_ulogic;
      b_i : IN  std_ulogic;
      y_o : OUT std_ulogic);
  END COMPONENT;

  COMPONENT or2gate
    PORT (
      a_i : IN  std_ulogic;
      b_i : IN  std_ulogic;
      y_o : OUT std_ulogic);
  END COMPONENT;

  SIGNAL wire0, wire1 : std_ulogic;
  SIGNAL wire2 : std_ulogic;

BEGIN

  inv_gate_1 : invgate
    PORT MAP (
      a_i => sel_i,
      y_o => wire2);
```

```
  and2_gate_1 : and2gate
    PORT MAP (
      a_i => a_i,
      b_i => sel_i,
      y_o => wire0);

  and2_gate_2 : and2gate
    PORT MAP (
      a_i => b_i,
      b_i => wire2,
      y_o => wire1);

  or2_gate_1 : or2gate









END structure;
```

Naming Conventions for Files

File	Content
`mux2to1_structure.vhd`	Entity `mux2to1`, Architecture `structure`
`t_mux2to1.vhd`	Testbench entity `t_mux2to1`
`de1_mux2to1_structure.vhd`	Envelope for FPGA prototype testing

Testbench

A testbench is a VHDL file that instantiates the **Device Under Test (DUT)**, applies input stimuli, and observes outputs. The testbench entity has **no ports**.

Testbench Structure

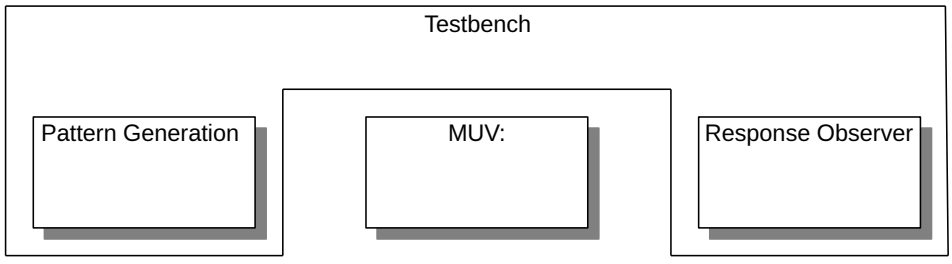


Figure 4: 2-to-1 Multiplexer - Conceptual Diagram of a Test Bench

Combinational Testbench

The mux2to1 has three inputs — all $2^3 = 8$ combinations are tested, grouped by the select signal:

```vhdl
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY t_mux2to1 IS
END t_mux2to1;

ARCHITECTURE tbench OF t_mux2to1 IS

  CONSTANT period : time := 10 ns;

  SIGNAL a_i   : std_ulogic;
  SIGNAL b_i   : std_ulogic;
  SIGNAL sel_i : std_ulogic;
  SIGNAL y_o   : std_ulogic;

BEGIN

  -- Device Under Test
  DUT : ENTITY work.mux2to1(equation)
    PORT MAP (
      a_i   => a_i,
      b_i   => b_i,
      sel_i => sel_i,
      y_o   => y_o
    );

  -- Stimuli
  stimuli_p : PROCESS
  BEGIN

    -- sel = '0' → output follows b_i
    a_i <= '0'; b_i <= '0'; sel_i <= '0'; WAIT FOR period;
    ASSERT y_o = '0' REPORT "Error: sel=0, b=0 → y/=0" SEVERITY failure;

    a_i <= '0'; b_i <= '1'; sel_i <= '0'; WAIT FOR period;
    ASSERT y_o = '1' REPORT "Error: sel=0, b=1 → y/=1" SEVERITY failure;

    a_i <= '1'; b_i <= '0'; sel_i <= '0'; WAIT FOR period;
    ASSERT y_o = '0' REPORT "Error: sel=0, b=0 → y/=0" SEVERITY failure;

    a_i <= '1'; b_i <= '1'; sel_i <= '0'; WAIT FOR period;
    ASSERT y_o = '1' REPORT "Error: sel=0, b=1 → y/=1" SEVERITY failure;

    -- sel = '1' → output follows a_i
    a_i <= '0'; b_i <= '0'; sel_i <= '1'; WAIT FOR period;
    ASSERT y_o = '0' REPORT "Error: sel=1, a=0 → y/=0" SEVERITY failure;

    a_i <= '0'; b_i <= '1'; sel_i <= '1'; WAIT FOR period;
    ASSERT y_o = '0' REPORT "Error: sel=1, a=0 → y/=0" SEVERITY failure;

    a_i <= '1'; b_i <= '0'; sel_i <= '1'; WAIT FOR period;
    ASSERT y_o = '1' REPORT "Error: sel=1, a=1 → y/=1" SEVERITY failure;

    a_i <= '1'; b_i <= '1'; sel_i <= '1'; WAIT FOR period;
    ASSERT y_o = '1' REPORT "Error: sel=1, a=1 → y/=1" SEVERITY failure;
```

```
    WAIT;    -- suspend process – simulation ends
  END PROCESS;

END tbench;
```

Key elements of a combinational testbench:

Element	Purpose
`CONSTANT period`	Defines the time step between stimuli
`stimuli_p` PROCESS	Applies input values sequentially
`WAIT FOR period`	Holds input values for one time step
`ASSERT`	Self-checking: reports errors automatically
`WAIT`	Suspends the process — terminates simulation

Prototype Test Environment

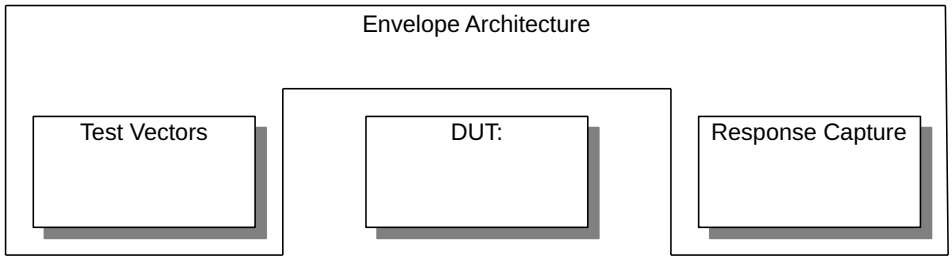


Figure 5: Connection to Peripheral Elements

To test a module on the DE1 FPGA board, an **envelope architecture** connects the module ports to the physical pins of the board (switches, LEDs, 7-segment displays):

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY de1_mux2to1 IS
  PORT (
    SW   : IN  std_ulogic_vector(2 DOWNTO 0);
    LEDR : OUT std_ulogic_vector(3 DOWNTO 0)
  );
END de1_mux2to1;

ARCHITECTURE structure OF de1_mux2to1 IS

BEGIN
```

```
  LEDR(0) <= SW(0);              -- show input a   on LED 0
  LEDR(1) <= SW(1);              -- show input b   on LED 1
  LEDR(2) <= SW(2);              -- show select    on LED 2

  DUT : ENTITY work.mux2to1(equation)
    PORT MAP (
      a_i   => SW(0),
      b_i   => SW(1),
      sel_i => SW(2),
      y_o   => LEDR(3)           -- show result    on LED 3
    );

END structure;
```

The full module hierarchy for one design:

```
Simulation:                    Prototype Testing:

t_mux2to1(tbench)              de1_mux2to1(structure)
  mux2to1(equation)              mux2to1(equation)
```

Summary

- A VHDL design unit consists of an **`ENTITY`** (interface) and an **`ARCHITECTURE`** (function)
- All statements in an architecture execute **concurrently** — they model hardware, not software
- **`std_ulogic`** and **`std_ulogic_vector`** are the fundamental signal types
- **`unsigned`** and **`signed`** from `ieee.numeric_std` enable arithmetic
- **Type conversions** are explicit in VHDL — always convert before arithmetic
- **Internal signals** connect concurrent statements within an architecture
- **Component instantiation** builds design hierarchy with `ENTITY work.x PORT MAP`
- A **testbench** verifies correctness with stimuli, `WAIT FOR`, and `ASSERT`

Reference

- [Mea25] - Mealy, B., Tappero, F.: *Free Range VHDL*, 2025 — Chapters 2, 3, 4