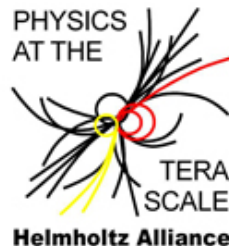


# Basics of VHDL

**Christian Schröder**

Institut für Physik  
Johannes-Gutenberg-Universität Mainz

6th Detector Workshop of the Helmholtz Alliance  
"Physics at the Terascale"  
Mainz, 26/02/2013



# Hardware Description Languages (HDLs)

- Describe, model and simulate the operation of complex digital circuits
- Modelling ranges from switch-level to processor-level (e.g. Pentium, ARM7)
- In contrast to general purpose languages, HDLs provide:
  - ➡ synthesizable subset
  - ➡ structure & instantiation
  - ➡ timing, hardware concurrency & parallel activity flow
- Verilog & VHDL
  - ➡ most-widely used and well-supported HDLs
  - ➡ industry standards
- SystemC & SystemVerilog
  - ➡ system-level design & verification
  - ➡ increased popularity
- Other HDLs represent only a small piece on the total HDL market

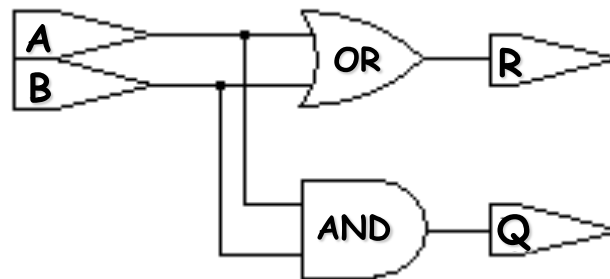
# Why VHDL and not Verilog in this Course ?

- Both Verilog and VHDL are powerful languages
- Verilog
  - ➔ "Loosely-typed" language → faster to write
  - ➔ Less popular among the participants
- VHDL
  - ➔ "Strongly-typed" language → easier to debug
  - ➔ Very-high-speed integrated circuit Hardware Description Language
  - ➔ General purpose parallel programming language (not only electronics)
  - ➔ More popular among the participants

## Verilog

```
module gates(A,B,R,Q);  
  input  A,B;  
  output Q,R;  
  
  assign Q = A & B;  
  assign R = A | B;  
  
endmodule
```

## Schematic



## VHDL

```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity gates is  
  port( A,B: in  std_logic;  
        Q,R: out std_logic);  
end;  
  
architecture implement of gates is  
begin  
  Q <= A and B;  
  R  <= A or B;  
end;
```

# VHDL - General remarks

- VHDL is not case sensitive, but it is strongly recommended to be consistent in using upper/lower case writing
  - ➡ as in every programming language one should apply coding rules and stick to them
- Reserved words (in this talk set in **blue**) must not be used as names (signals, variables, ...)
- Identifiers (names for variables, signals, labels, entities, ...) must be unique within the scope of the namespace (within an entity-architecture block)
- VHDL supports both behavioral and structural (and mixed) design
  - ➡ there are different ways of how to implement the same design
- VHDL supports statements and expressions that cannot be synthesized and may be for simulation only (e.g. delay times, ...)

# Object and data types

- VHDL knows several object types like **signal** (represents a wire), **variable** and **constant** → define types of objects
- Data types define types of values of these objects
- Basic data types are scalars like:
  - ➔ **integer, signed, unsigned, boolean**
  - ➔ **std\_logic** → standard data type for electronic designs with 9 different states (some are only relevant for simulation)
    - **'0', '1'** - logic levels
    - **'U', 'X', 'W'** - uninitialized, unknown, weak unknown
    - **'Z', 'L', 'H'** - high impedance, weak 0, weak 1
    - **'-'** - don't care (for synthesis)
- Composite data types are collections of scalar type
  - ➔ **arrays** (e.g. for buses)
  - ➔ **records** (similar to C)

```
type bit is ('0', '1');
type byte is array(7 downto 0) of bit;
type bus is array(0 to 7) of std_logic;
-- or (btw. this is a comment):
type bus is std_logic_vector(0 to 7);
type myrec is record(A: bit; B: byte)
end record myrec;

. . .

constant MAXVAL: integer := 127;
signal mybus: std_logic_vector(7 downto 0);
variable counter: integer range 0 to 127;
```

# Libraries and types

- Libraries (**library**) are used to aggregate and provide objects and types for re-usage
- Commonly used are some IEEE libraries:
  - ➡ **ieee.std\_logic\_1164** → provides standard types for signals: **std\_logic** and **std\_logic\_vector**
  - ➡ **ieee.std\_logic\_arith** → provides basic arithmetic, conversion, and comparison functions for **signed**, **unsigned**, **integer** and **std\_logic**
  - ➡ **ieee.std\_logic\_unsigned** → provides signed/unsigned arithmetic and conversion for **std\_logic\_vector**

## Example:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

...

signal A,B: std_logic_vector(7 downto 0);

...

if A - 1 >= '0' then
    B <= A - '1';
else ...
```

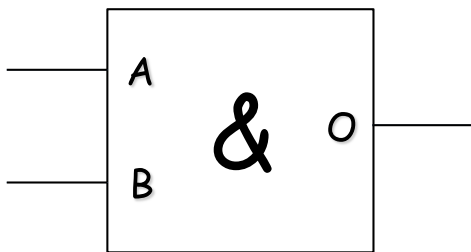
# Entity - Basic building blocks in VHDL

- Basic building block of a VHDL model is the **entity**
- Entity declaration consist of
  - ➔ Name of **entity**
  - ➔ Interface specification
    - Config. Parameter definitions: **generic**
    - I/O port definitions: **port** (**in**, **out** or **inout**)
- Describes only the outside view of a hardware module (implementation independent)

## Entity template

```
entity NAME is  
    generic (LIST);  
    port (LIST);  
end entity NAME;
```

## Example: AND (2 Is, 1 O)



```
entity AND is  
    generic (timeprop: delay_length);  
    port (A,B: in std_logic;  
          O : out std_logig);  
end entity AND;
```

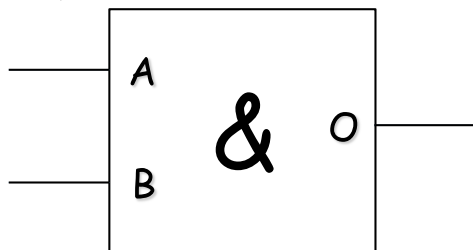
# Architecture - Filling the inside

- An **architecture** describes how an entity operates inside
- Multiple architectures for one entity are possible
  - ➔ alternatives can be declared
- Description style can be structural, behavioral or mixed
  - ➔ Hierarchical/top-down design is possible

## Entity template

```
architecture NAME of ENTITYNAME is
-- local declarations
begin
    -- statements
end architecture NAME;
```

Example: AND (2 I, 1 O)



```
architecture and_behav of and is
begin
    O <= '1' when A='1' and B='1' else
        '0';
end architecture and_behav;
```



# Components - Using design entities

- To allow for hierarchical designs entities can be used as components within other entities through:
  - ➡ Declaration (**component**) → has to match exactly the entity definition
  - ➡ Instantiation with unique name and specification of port connectivity via the **port map** → ordering (and type) has to match the component! declaration!

Example: NAND composed of AND

```
-- and declaration of previous slides
entity and is
  port(A,B: in  std_logic;
        O  : out std_logic);
end entity AND;
```

```
architecture and_behav of and is
begin
  O <= '1' when A='1' and B='1' else
    '0';
end architecture and_behav;
```

```
entity nand is
  port(A,A2: in  std_logic;
        O   : out std_logic);
end entity nand;
```


```
architecture nand_behav of nand is
  component and is
    port(A,B: in  std_logic;
          O  : out std_logic);
  end component and;

  signal AO: std_logic;
begin
  intand1: and port map(A,A2,AO);
  O <= not AO;
end architecture nand_behav;
```

# Port map - alternate way

- The **port map** ordering has to match the component  
OR:
- The signals/ports have to be connected explicitly by names

```
architecture nand_behav of nand is
component and is
  port(A,B: in std_logic;
        O : out std_logig);
end component and;
signal AO: std_logic;
begin
  intand1: and port map (A,A2,AO) ;
  O <= not AO;
end architecture nand_behav;
```



```
architecture nand_behav of nand is
component and is
  port(A,B: in std_logic;
        O : out std_logig);
end component and;
signal AO: std_logic;
begin
  intand1: and port map(
    A => A,
    B => A2,
    O => AO);
  O <= not AO;
end architecture nand_behav;
```

# Concurrent statements

## ■ Concurrent statements

➡ Signal assignment

➡ Conditional signal assignments

➡ Selected signal assignments

## ■ Whenever one of the dependent signals changes the assignment is updated

➡ continuous assignment

## ■ All statements are updated concurrently/in parallel

```
A <= B xor C;
```

```
A2 <= B when C else D;
```

```
with BUS select  
  O <= A when x"01",  
        B when x"0F",  
        C when others;
```

```
A <= B xor C;  
B <= C or not D;  
A <= C nor B;
```

Important:

➡ Signals must not be driven by more than one other signal

# Processes

- Processes themselves are executed concurrently
- Processes are usually triggered by one or more signals  
→ defined by the sensitivity list
- Statements inside processes are executed sequentially
  - ➡ signals hold the last assigned value at the end of the process
- Inside processes variables can be used
- **If-else**-, **case-when**- and **for-loop**-statements can be used inside
- Usually used as clocked processes for pipelined designs

## Entity template

```
[processlabel:] process architecture [(sensitivity-list)] [is]
-- local declarations

begin
    -- statements
end process [process-label];
```

# Example: clocked flip-flop with reset

## ■ D-Flipflop with async. reset

```
DFF : process (RST, CLK)
begin
  if RST = '1' then
    Q <= '0';
  elsif rising_edge (CLK) then
    Q <= D;
  end if;
end process DFF;
```

## ■ D-Flipflop with sync. reset

```
DFF : process (CLK)
begin
  if rising_edge (CLK) then
    if RST = '1' then
      Q <= '0';
    else
      Q <= D;
    end if;
  end if;
end process DFF;
```

- Also `CLK'event and CLK = '1'` instead of `rising_edge (CLK)` is common syntax
- Of course, also `falling_edge ()` can be used for processes on the falling edge (or: `CLK'event and CLK = '0'`)

# Signals vs. variables

- Signals represent physical wires
  - ➔ assignment via "`<=`"
  - ➔ allow communication between processes and components
  - ➔ concurrently connect different design units
  - ➔ update is scheduled with an associated delay (at least a delta delay) within processes
  - ➔ can exist inside and outside of processes
  - ➔ can only be driven by one process at a time
  
- Variables can be seen as placeholder for the assigned value
  - ➔ assignment via "`:=`"
  - ➔ updated immediately (no delay)
  - ➔ only exist inside of processes
  - ➔ follow more the logic of non-HDL programming/easier to read(?)

# Example: signals vs. variables

## ■ Using signals:

```
entity EXAMPLE is
  port( CLK:    in std_logic;
        RESULT: out integer);
end entity EXAMPLE;

architecture VAR of EXAMPLE is
  signal var1: integer :=1;
  signal var2: integer :=2;
  signal var3: integer :=3;
begin
  process (CLK)
  begin
    if rising_edge(CLK) then
      var1 <= var2;
      var2 <= var1 + var3;
      var3 <= var2;
      RESULT <= var1 + var2 + var3;
    end process;
end architecture VAR;
```

## ■ Using variables:

```
entity EXAMPLE is
  port( CLK:    in std_logic;
        RESULT: out integer);
end entity EXAMPLE;

architecture VAR of EXAMPLE is
begin
  process (CLK)
    variable var1: integer :=1;
    variable var2: integer :=2;
    variable var3: integer :=3;
  begin
    if rising_edge(CLK) then
      var1 := var2;
      var2 := var1 + var3;
      var3 := var2;
      RESULT <= var1 + var2 + var3;
    end process;
end architecture VAR;
```

# Other useful details for the Exercises (1)

## ■ `if-else` and `case` statements

- ➡ similar to those available in other high-level programming languages:
- ➡ only inside of processes, eg. for clocked processes  
(clock should be in sensitivity list in case of clocked processes!)

Syntax	Example
<code>if</code> conditional_expression <code>then</code> statement(s) <code>elsif</code> conditional_expression statement(s) <code>else</code> statement <code>end if</code> ;	<code>if</code> rising_edge(clk) <code>then</code> out <= A <code>xor</code> B; <code>if</code> cnt > 0 <code>then</code> cnt <= cnt + 1; <code>end if</code> ; <code>end if</code> ;



# Other useful details for the Exercises (2)

## ■ Generate statement

- ➡ Evaluated during the circuit elaboration step  
→ similar to macros in C
- ➡ used to repeat instantiation constructs
- ➡ used to create conditional instantiations
- ➡ index in the "for" construct has local scope and can be used to pick specific signals from an array in portmap statements.

```
architecture GEN of REG_BANK is
    component REG
        port(D,CLK,RESET : in  std_logic;
              Q           : out std_logic);
    end component REG;
begin
    GEN_REG:
        for I in 0 to 3 generate
            REGX : REG port map
                (DIN(I), CLK, RESET, DOUT(I));
        end generate GEN_REG;
end architecture GEN;
```

## Other useful details for the Exercises (3)

### ■ Numbers

- ➡ Scalar constant: `'0'`, `'1'`
- ➡ Array constant: `"0010"`, `"0100"`, ..
- ➡ hexdec. values: `x"00"`, `x"3F"`, ...

### ■ Arrays

- ➡ concatenation:

`A <= B & C;`

`B <= '0' & '1' & "10";` -- 0110

- ➡ others:

`A <= B"101" & (others => '0');` -- 1010 0000

`B <= (others => '1');` -- 1111 1111

- ➡ named, partial assignement:

`A <= (3=>'1', others=>'0');` -- 0000 1000

## Other useful details for the Exercises (4)

### ■ Operators

- ➡ similar to the operators of other programming languages
- ➡ Can be used in both continuous and procedural assignment

Operator Type	Operator
Arithmetic operators	*, /, +, -, mod, rem
Logical operators	and, or, not, nand, nor, xor, xnor
Relational operators	>, <, >=, <=
Equality	= , /=
Bitwise operators	and, or, nand, nor, xor, xnor
Shift Operator	sll, srl (logical, i.e. fill with 0s) sla, sra (arithmetic)
Concatenation	&