lektronik
abor

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

ELEKTRO- UND INFORMATIONSTECHNIK

Electronic Design Automation (EDA)

Rechnergestützter Entwurf Digital (RED)

Event-Driven Circuit Modeling Using *VHDL*

Martin J. W. Schubert, Electronics Laboratory,

Ostbayerische Technische Hochschule Regensburg,

Regensburg, Bavaria, Germany

Abstract. A hardware description language (HDL) named VHDL is introduced and applied to different finites state machine designs. Modeling, simulation and synthesis demonstrated by design examples counter, PWM modulator, IIR filter and DC/DC buck converter.

1 Introduction

Key intention of the *US* Department of Defense (*DoD*) during creation of *VHDL* in the 1970's was to develop a digital hardware description language that can be simulated and delivers unambiguous results independent from tools, computers, vendors, operating systems or semiconductor technologies. Synthesis was out of scope of thinking in the 1970's, so that there is few support for synthesis in *VHDL* and consequently strong tool and vendor dependencies, e.g. when defining which *VHDL* signal is connected to which pin of a chip.

VHDL is an acronym for *VHSIC* hardware description language, whereas *VHSIC* is an acronym for very high speed integrated circuit.

Since 1987 VHDL is an IEEE open standard, i.e. everybody use VHDL for free, e.g. for offering commercial models or databases.

VHDL is a concurrent programming language, i.e. the sequence of concurrent statements is insignificant.

VHDL is not case sensitive. Intuitively, this document uses ALL UPPERCASE LETTERS for VHDL keywords and lowercase letters for self-defined identifiers, Whereas some uppercase initials may be used to improve readability, e.g. *DataOutBitWidth*.

The organization of this document is as follows:

Chapter 1 is this introduction,
Chapter 2 summarizes VHDL basics,
Chapter 3 details the directory structure for model files used in this course,
Chapter 4 introduces VHDL models for LTI systems, particularly 2nd order IIR filters.
Chapter 5 offers further details concerning VHDL exceeding the scope of this course,
Chapter 6 draws relevant conclusion and
Chapter 7 offers references.

2 *VHDL* Basics

Fundamentals Summary

1. Design Units

• Entity	= symbol = description to the outer world
• Architecture	= schematic, description of inner realization
• Configuration	configures entity – architecture combinations

2. Code Types

• Structural	= hierarchical = instantiation of submodules
• Concurrent	standard VHDL code, line sequence irrelevant
• Sequential	within subprograms and processes, top-down

3. Data

3.1 Data Objects:

• Signals	assignment with delay, never immediately
• Variables	assignment immediately
• Constants	variables at compile time, constants afterwards

3.2 Data Types:

• Scalar: integer, real, physical, enumerated	examples: ***time*** is physical, ***bit*** is enumerated
• Composed: array, record	array(integer_index), record_name**.**field_name

4. Concurrency

The sequence of concurrent statements is insignificant.

Examples made in this chapter can be simulated with *ModelSim* [*ModelSim*] simulator.

2.1 Design Units: Entity, Architecture, Configuration

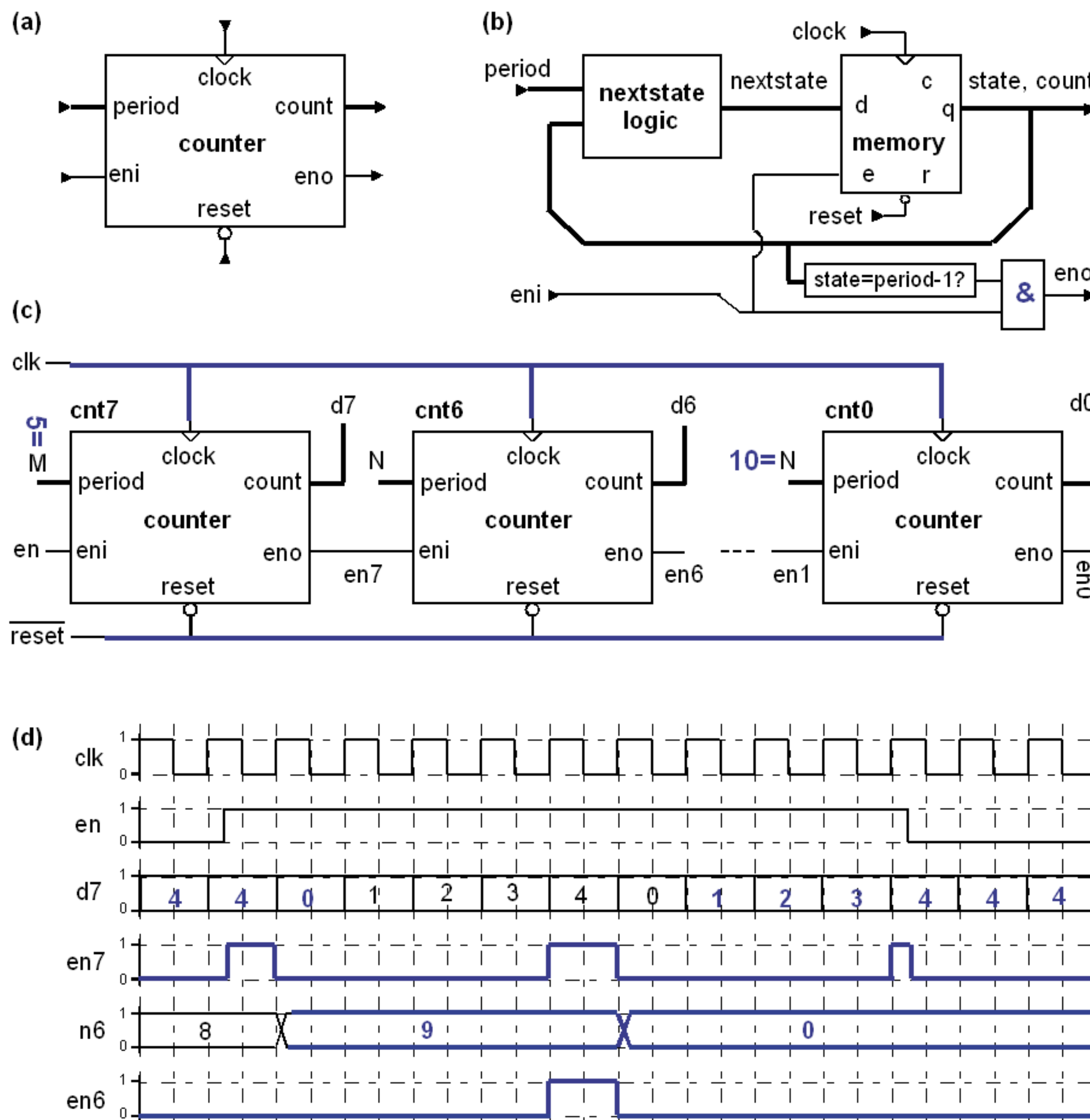


Fig. 2.1: (a) Symbol corresponding to a *VHDL ENTITY*, **(b)** Schematic corresponding to a *VHDL ARCHITECTURE*, **(c)** component instantiation = *structural*, **(d)** timing diagram.

Entity – architecture – configuration: The design in Fig. 2.2 contains in part (a) an example of a symbol = *VHDL entity* and in part (b) a possible schematic = *VHDL architecture*. As different instantiations of the symbol in part (c) may have different architectures, a *VHDL configuration* can be used to determine which entity has to be realized with which architecture. An entity-architecture combination is referred to as component. The usage of component instantiations in part (c) is called structural coding. Part (d) illustrates timing diagrams.

2.1.1 Entity

Fig. 2.1(a) shows the symbol of a counter, referred to as *VHDL entity*. A *VHDL* component consist of entity and architecture, whereas the latter corresponds to the schematic. The *entity* describes the component's communication ports to the outer world.

Listing 2.1.1: VHDL entity *counter* realizing the counter symbol in Fig. 2.2(a).

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
ENTITY counter IS
    GENERIC(period:POSITIVE:=10);
    PORT(reset,clock:IN std_logic;
         eni:IN std_logic;
         eno:BUFFER std_logic;
         count:BUFFER NATURAL RANGE 0 TO period-1
        );
END ENTITY counter;
```

Generics are variables during compile time and constants afterwards. In the example above generic *period* is a constant with defaults to 10 when omitted in the instantiation. Its data type is *POSITIVE*, which is a subtype of *INTEGER* with lowest value 1.

Port signals are *reset*, *clock*, *eni*, *eno*, and *count*.
Direction mode *IN* for *reset*, *clock* and *eni* determines read access only to this signals within the counter.
Direction modes *OUT* and *BUFFER* are similar. The two differences are:

- *BUFFER* mode signals can be read within the component's architecture,
- mode *OUT* signals may have several drivers while *BUFFER* mode signals may have one driver only. This guarantees that what you drive is what you get.

Signals *eni* and *eno*, representing enable-in and enable out, respectively, as well as *reset* and *clock* have data type *std_logic*, a scalar logic type that comprehends state values 'U', 'X', '0', '1', 'Z', 'L', 'H', 'W', '-', standing for *uninitialized, forcing unknown, forcing 0, forcing 1, high impedance, week low, week high, week unknown* and *don't care*, respectively. Usage of *std_logic* requires the top line: Access *LIBRARY ieee* and declare to *USE ALL* declarations made in its package *std_logic_1164*. For more details see → subchapter *data types*.

2.1.2 Architecture

Fig. 2.1(b) shows the schematic of a counter, referred to as *VHDL architecture*. A *VHDL* component consist of entity and architecture. The *architecture* describes the component's inner realization as schematic view. Multiple architectures may be designed for the same entity. The *entity – architecture* assignment is made with a configuration.

In the following, architecture names will begin with prefix *beh* for behavioral or *rtl* for register transfer level, used as synonym for "synthesizable" from historical reasons.

Listing 2.1.2 is a single statement declaring an empty architecture named *rtl_counter_empty* to be associated to entity *counter*. An entity with name must exist when compiling an architecture for it.

Listing 2.1.2: Empty architecture *rtl_counter_empty*.

```
ARCHITECTURE rtl_counter_empty OF counter IS
BEGIN
END ARCHITECTURE rtl_counter_empty;
```

Question: After compilation of listings 2.1.1 and 2.1.2, which values will be found in the following data objects?:

period= …. **, reset=** …. **, clock=** …. **, eni=** …. **, eno=** …. **, count=** ….

Listing 2.1.3 demonstrates: Non-executable declarations are inserted between keywords *IS* and *BEGIN*, executable statements between *BEGIN* and *END* of an architecture statement.

- Text after a double minus sign ("--") is comment.
- String "*RANGE 0 TO period-1*" after data type *NATURAL* will be guarded by the simulator and will cause the synthesizer to spend no more bit than necessary for this signal.

Listing 2.1.3: Minimal architecture *rtl_counter_0*.

```
ARCHITECTURE rtl_counter_0 OF counter IS
  SIGNAL nextstate:NATURAL RANGE 0 TO period-1;
BEGIN
  -- no Counter
  eno   <= '0';
  count <= 5;
END ARCHITECTURE rtl_counter_0;
```

Question: After compilation of listings 2.2.1 and 2.2.3, which values will be found in the following data objects?:

period= …. **, reset=** …. **, clock=** …. **, eni=** …. **, eno=** …. **, count=** ….

Solutions:

Listing 2.1.2:

```
period= 10 , reset='U', clock=  'U', eni= 'U', eno= 'U', count=  0
```

Listing 2.1.3:

```
period= 10 , reset='U', clock=  'U', eni= 'U', eno= '0', count=  5
```

Testing VHDL codes with ModelSim simulator

A related testbench to be used with *ModelSim* [*ModelSim*] simulator is located in directories
...\Model_Files\VHDL\lib_flat
...\Model_Files\VHDL\lib_struct\ModelSim\tb_de1soc_counter
Within these directories there is an ASCII file named *work.do*, and in the former also *work_de1soc.do* which contain names and trees of necessary file to be compiled. To use it,

- Start *ModelSim* simulator > close *Welcome window*.
- *ModelSim* top-line menu > *File* > *Change Directory* ... > navigate to directory *...\Model_Files\VHDL\lib_flat*
- In the *Transcript Window* > type "*do work.do*"
- Command file *work.do* will be worked off until a graphics appears.

The two listings below, 2.1.4 and 2.1.5, will be discussed in more detail in later chapters. Briefly: All counter architectures presented in this chapter were designed for the same *ENTITY counter*. These architectures may be in the same file or in different files. Which architecture is used for which instantiation of a *counter* module depends on the configuration, which will be explained in the next subsection.

Listing 2.1.4 demonstrates how to describe *nextstate*-logic and *state* memory within a single *PROCESS* statement. This model is similar to the schematics of Fig. 2.1(b) but counts down from *perod-1* to 0 and sets flag *eno* at count = 0.

Listing 2.1.4 demonstrates how to describe *nextstate*-logic and *state* memory within two different *PROCESS* statements. This model is equivalent to the schematics of Fig. 2.1(b).

Listing 2.1.4 describe *nextstate*-logic and *state* memory within a single process statement, count down from *period-1* to 0 and set flag *eno* at *count* = 0.

```
ARCHITECTURE rtl_counter_fsm1_dn OF counter IS
  SIGNAL state,nextstate:NATURAL RANGE 0 TO period-1;
BEGIN
  -- Counter
  p_fsm:PROCESS(reset,clock)
  BEGIN
    IF reset='0' THEN
      state <= 0;
    ELSIF clock'EVENT AND clock='1' AND eni='1' THEN
      IF state = 0 THEN -- Begin NextState Logic
        state <= period-1;
      ELSE
        state <= state-1;
      END IF;                  -- End NextState Logic
    END IF;
  END PROCESS p_fsm;
  --
  -- output logic:
  count <= state;                                 -- Moore outputs
  eno <= '1' WHEN (state=0 AND eni='1') ELSE '0'; -- Mealy outputs
END ARCHITECTURE rtl_counter_fsm1_dn;
```

Listing 2.1.5: Describe *nextstate*-logic and *state* memory with two process statements. This model realizes the schematics of Fig. 2.1(b).

```
ARCHITECTURE rtl_counter_fsm2_up OF counter IS
  SIGNAL state,nextstate:NATURAL RANGE 0 TO period-1;
BEGIN
  -- Begin NextState Logic:
  p_NextState:PROCESS(eni,state)
  BEGIN
   nextstate<=state;
   IF eni='1' THEN
      IF state = period-1 THEN
        nextstate<=0;
      ELSE
        nextstate<=state+1;
      END IF;
    END IF;
  END PROCESS p_NextState; -- End NextState Logic
  --
  -- Begin State Memory
  p_StateMemory:PROCESS(reset,clock)
  BEGIN
    IF reset='0' THEN
      state <= 0;
    ELSIF clock'EVENT AND clock='1' THEN
      state <= nextstate;
    END IF;
  END PROCESS p_StateMemory;
  --
  -- output logic:
  count <= state;                                        -- Moore outputs
  eno <= '1' WHEN (state=period-1 AND eni='1') ELSE '0'; -- Mealy outputs
END ARCHITECTURE rtl_counter_fsm2_up;
```

2.1.3 Configuration

In this course, we will mainly use the default configuration of the configurations specification as detailed below.

2.1.3.1 Flat Configuration

Many VHDL designers have never heard of configurations. This is due to the good *VHDL* **default configuration** combining the youngest (last compiled) architecture to the entity it is compiled for.

Important practical hint: In contrast to the *ModelSim* simulator, that obeys the rule of using the very last complied architecture by default, it is not clear to the author how *Quartus II* makes this selection, when several architectures for the same entity are compiled within a file.

2.1.3.2 Flat Configuration Declaration

If we want to configure our design particularly, we can use a configuration declaration as shown in listing 2.1.3.2.

Listing 2.1.3.2: Declare architecture *rtl_counter_fsm1_dn* to fill entity *counter*.

```
CONFIGURATION cfg_counter OF counter IS
  FOR rtl_counter_fsm1_dn
  END FOR;
END CONFIGURATION cfg_counter;
```

2.1.3.3 Structural Configuration Declaration

Hierarchical configurations allow to configure also lower-level components. Configuration declaration and specification are advanced topics.

Listing 2.1.3.3: Structural Configuration declaration.

```
CONFIGURATION cfg_x OF x IS
  FOR rtl_x
    FOR label_1: USE ... ;
    END FOR;
  END FOR;
END CONFIGURATION cfg_x;
```

2.1.3.4 Configuration Specification

Configuration specifications are not an own design unit but statements within the declaration region of an architecture as implied in listing 2.2.3.2.

Listing 21.3.4: Configuration specification.

```
ARCHITECURE rtl_x OF x IS
  FOR label_1:entity_name USE ENTITY work.entity_name(architecture_name);
  FOR OTHERS: entity_name USE CONFIGURATION work.configuration_name;
  -- FOR ALL:… -- this does not tolerate any other configuration specific.
BEGIN
  ...
END ARCHITECURE rtl_counter;
```

2.2 Code Types: Concurrent, Sequential, Structural

VHDL code can be written

- Concurrent
- Sequential
- Structural

2.2.1 Concurrent Modeling

VHDL statements assigning values to signals are concurrent, when this does not occur within sequential code. The sequence of concurrent statements is insignificant. Code is sequential within subprograms and *PROCESS* statements.

Listing 2.2.1: Concurrent signal assignments (a) – (c) using VHDL are equivalent.

(a) Signal *y* is evaluated in the 1st line

```
y   <= (d0 x(n) + s1 ) / c0;
ns1 <=  d1 x(n) + s2 - c1 y;
ns2 <=  d2 x(n)      - c2 y;
```

(b) Signal *y* is evaluated in the 2nd line

```
ns1 <=  d1 x(n) + s2 - c1 y;
y   <= (d0 x(n) + s1 ) / c0;
ns2 <=  d2 x(n)      - c2 y;
```

(c) Signal *y* is evaluated in the 3rd line

```
ns1  <=  d1 x(n) + s2 - c1 y;
ns2  <=  d2 x(n)      - c2 y;
y    <= (d0 x(n) + s1 ) / c0;
```

Listing 2.2.1 illustrates a piece of concurrent code with 3 code lines in 3 different sequences. The result is the same for all of them. This is realized by two means concerning the VHDL driver <=:

1. VHDL drivers operate uni-directional, i.e. *y<=a;* has no impact on signal *a*.
2. VHDL drivers principally have a delay, e.g. *y<=a AFTER 2ns;*

No delay is the same as zero delay: *y<=a* is equivalent to *y<=a AFTER 0ns*. A simulated delay for zero delay is achieved with so-called simulation deltas (Δ). In listing 2.21 all 3 lines would be activated in response to a value change (event) on signal *x* at Δ zero. However, the new values of signals *y, ns1, ns2* are not immediately assigned but scheduled for the next delta. After having worked-off Δ zero, the simulator proceeds to the Δ one. Now the new values scheduled in Δ zero are assigned to signals *y, ns1, ns2*. These signal changes of *y*, ns_1, ns_2 at Δ one trigger a new scheduling of signals ns_1, ns_2, which will be executed in Δ two. When there are no more changes to be scheduled in a next Δ, the simulator proceed to the next time step. An infinite series of Δ-steps, e.g. caused by *c<=NOT c;* are stopped by *ModelSim* simulator at 5000 Δ's.

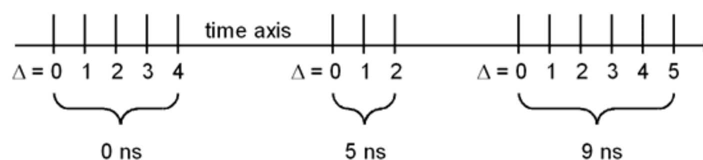


Fig. 2.2.1: The VHDL time axis introduces Δ delays for steps of 0ns.

In VHDL, any signal has to hold its value 3 times, namely for actual, scheduled and last value. This causes a lot of software overhead making event-driven simulation slow and memory consuming. Consequently, big arrays (e.g. pixel images) should be rather modeled as constants than as signals. The last value of signal *x* can be obtained with the corresponding attribute, e.g. with VHDL statement *x_last <= x'LAST_VALUE*.

Several drivers to the same signal in concurrent code make up a bus and require the signal to have a resolved data type, e.g. *std_logic*. For example the concurrent code

```
y <= a; -- with a='0'
y <= b; -- with b='1'
```

will cause signal *y* to get state value '*X*' (forcing unknown).

Typically, we avoid signal assignment with finite delays, because physical gates do not care about user-defined delays. Synthesis software will automatically include delay in case of back-annotation. Then users may choose between simulations with best, typical or worst case parameters for a particular target technology.

2.2.2 Sequential Modeling

Within processes and subprograms VHDL code is processed sequential, namely top-down. Subprograms are out of the scope of this tutorial. Processes statements are infinite loops between *BEGIN* and *END PROCESS* key words running within the same Δ from *WAIT* statement to *WAIT* statement.

WAIT FOR waits for a particular time span, WAIT UNTIL waits until a condition is fulfilled and WAIT ON waits for events (value changes) of at least one signal in the sensitivity list.

Listing 2.3.2.1: general process statement

```
PROCESS
BEGIN
  ...
  WAIT FOR <time>;
  ...
  WAIT UNTIL <condition>;
  ...
  WAIT ON <signal_list>;
END PROCESS;
```

Process labels are optional but helpful in the debugger. The equivalence of listings 2.3.2.2 (a) and (b) makes clear, that a process with sensitivity list runs top-down (i) at *time*=0ns, Δ=0 and for any event 8signal change) in the sensitivity list.

Listing 2.3.2.2: Processes must have either at least one *WAIT* statement or a sensitivity list

<table>
<tr><td>(a): using WAIT statement</td><td>(b): using sensitivity list</td></tr>
<tr><td><pre>label:PROCESS
BEGIN
  ...
  WAIT ON signal1, signal2,…;
END PROCESS label;</pre></td><td><pre>label:PROCESS(signal1, signal2,…)
BEGIN
  ...
END PROCESS label;</pre></td></tr>
</table>

A *PROCESS* creates exact one driver to a signal driven within the process. If the same signal has several subsequent assignments with same delay in a *PROCESS* statement, the last assignment overrides previous assignments. Several assignments with different delays behave more complicated.

Modeling Combinational Logic without Memory with a VHDL Process Statement
To model combinational logic (i.e. logic without memory, e.g. *nextstate* logic) with a VHDL *PROCESS* statement, two basic rules for combinational logic must be obeyed:

1. **All input signals to the process must be listed in its sensitivity list,**
2. **All output signals must be driven in any situation.**

2.2.3 Structural Modeling

Structural code is hierarchical code instantiating submodules using the *COMPONENT* statement within the declaration region of an *ARCHITECTURE*, as illustrated in listing 2.2.3. The component statement is identical to the *ENTITY* statement, with the exception, that keyword *ENTITY* is replaced by keyword *COMPONENT*.

An instantiation in hardware description languages (HDLs) corresponds to a subprogram call in software. During a software subprogram, call a program pointer jumps into the subprogram and jumps back afterwards. On the contrary, a hardware instantiation generates hardware.

Listing 2.2.3: Testbench instantiating module *counter*

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
ENTITY tb_counter IS END ENTITY tb_counter;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ARCHITECTURE beh_tb_counter OF tb_counter IS
  CONSTANT M:NATURAL:=5;
  CONSTANT N:NATURAL:=10;
  SIGNAL count2:NATURAL RANGE 0 TO M;
  SIGNAL count0,count1:NATURAL RANGE 0 TO N;
  SIGNAL reset,clock:std_logic:='0';
  SIGNAL en:std_logic_vector(3 DOWNTO 0);
  --
  COMPONENT counter IS
    GENERIC(period:POSITIVE:=10);
    PORT(reset,clock:IN std_logic;
         eni:IN std_logic;
         eno:BUFFER std_logic;
         count:BUFFER NATURAL RANGE 0 TO period-1
        );
  END COMPONENT counter;
  CONSTANT fclk:REAL:=50.0E6;
  -- configuration specification:
  FOR i_cnt0:counter USE CONFIGURATION WORK.cfg_counter;
BEGIN
  clock <= NOT clock AFTER sec/(2.0*fclk);
  reset <= '0', '1' AFTER 12 ns;
  en(3) <= '1', '0' AFTER 4355 ns, '1' AFTER 4855 ns;
  i_cnt2:counter GENERIC MAP(M) PORT MAP(reset,clock,en(3),en(2),count2);
  i_cnt1:counter GENERIC MAP(N) PORT MAP(reset,clock,en(2),en(1),count1);
  i_cnt0:counter GENERIC MAP(N) PORT MAP(reset,clock,en(1),en(0),count0);
END ARCHITECTURE beh_tb_counter;

-- configuration declaration
LIBRARY WORK; USE WORK.ALL;
CONFIGURATION cfg_tb_counter OF tb_counter IS
  FOR beh_tb_counter
    FOR i_cnt1:counter USE ENTITY WORK.counter(rtl_counter_fsm1_dn); END FOR;
    FOR i_cnt2:counter USE ENTITY WORK.counter(rtl_counter_fsm1_dn); END FOR;
    FOR OTHERS:counter USE CONFIGURATION WORK.cfg_counter; END FOR;
  END FOR;
END CONFIGURATION cfg_tb_counter;
```

2.3 Data

VHDL knows 3 data objects and 6 data types, 4 of them scalar, 2 composed.

2.3.1 Data Objects:

- Signals — assignment with delay or Δ delay, never immediately
- Variables — assignment immediately
- Constants, Generics — variables at compile time, constants afterwards

2.3.2 Data Types:

- Scalar: integer, real, physical, enumerated — examples: ***time*** is physical, ***bit*** is enumerated
- Composed: array, record — array(integer_index), record_name**.**field_name

A signal is a data object that must have a data type. If not initialized, any data object initializes with *data_type'LEFT*, i.e. the leftmost value of its data type.

Examples:

- Data type *INTEGER contains integral numbers, i.e. ... -3, -2, -1,* 0, 1, 2, 3,….
- Data type *NATURAL* with range 0, 1, 2, 3,… is a subtype of *INTEGER*. For port signal *count* its range is limited to 0 – *period-1* to reduce its synthesized bit-width. Signals with data type *NATURAL* initialize by default with 0.
- Data type *POSITIVE* is a subtype of *NATURAL* with range 1, 2, 3, … Signals of data type *POSITIVE* initialize by default with 1.
- Data type *BIT* is an enumerated type and comprehends the state values '0' and '1', i.e. two *ASCII* characters symbolizing forcing 0 and forcing 1, respectively. Signals of this data type initialize by default with '0'.
- Data type *std_logic* is enumerated and comprehends state values 'U', 'X', '0', '1', 'Z', 'L', 'H', 'W', '-', standing for *uninitialized, forcing unknown, forcing 0, forcing 1, high impedance, week low, week high, week unknown* and *don't care*, respectively. Usage of data type *std_logic* requires the top line accessing *LIBRARY ieee* and declaring to *USE* it, typical we use *ALL* declarations made in package *std_logic_1164*.

The significance of *NATURAL* and *POSITIVE* being declared as subtypes of *INTEGER* rather than as types is, that they inherit all features of *INTEGER*, e.g. applicability of operators +, -, *, /, **. In contrast to type declarations, all these features automatically inherit from a subtype declaration.

2.4 VHDL Design for Synthesis

Naming Conventions

- Synthesizable code is typically called **rtl** (register transfer level) Code.
- Non-synthesizable code is typically called **beh**avioral (e.g. testbenches).

Do not Reinvent the Wheel!

According to Wikipedia [13] "*Reinventing the wheel* is a phrase that means to duplicate a basic method that has already previously been created or optimized by others", which doesn't make sense. Consequently,

Describe your circuit as detailed as necessary and as general as possible.

- If code is synthesizable or not depends among others on your libraries. An adder or multiplier from your library is usually better than a self-made one.
- Do not design primitives at gate level! Particularly flipflops (FFs) are optimized by the technology foundry. Use behavioral FF descriptions as detailed below.

Avoid Feedback Loops within Combinational Logic

Feedback loops within combinational logic have a high risk to create latches or cause oscillations. Therefore:

- **Use Finite state machine (FSM) model for digital design**
- **Avoid feedback loops within combinational logic!**

Design for Portability and Reusability

Preserving portability of VHDL models requires to design waiving on features offered by particular soft- and hardware manufacturers.

ASYNCHRONOUS RESET: Most FPGAs have a power reset which resets all FFs. This seduces some designers to code FFs without reset. **When HDL models are ported to an ASIC (application specific integrated circuit), FFs fall randomly into '0' or '1' states.** Test without initialization by global reset is difficult or impossible.

Create simple, reliable and well-tested design units as safe building blocks rather than "spaghetti code"! *Design for reusability* is an important topic [14] that exceeds the scope of this communication.

2.5 Modeling FSMs with *VHDL PROCESS* Statements

2.5.1 Two *PROCESS* Statements for *Nextstate* Logic and *State* Memory

The model with two *VHDL PROCESS* statements is close to the FSM illustrated in Fig. 4.1, as we use one process for the *nextstate* logic and another for the *state* memory. This is demonstrated in listing 2.51.

Listing 2.5.1: Architecture *rtl_counter_fsm2_up* appended to file *counter.vhd*.

```
ARCHITECTURE rtl_counter_fsm2_up OF counter IS
  SIGNAL NextCount:NATURAL RANGE 0 TO cPeriod-1;
BEGIN
  -- Begin NextState Logic:
  p_NextState:PROCESS(EnableIn,count) BEGIN
   NextCount<=count;
   IF EnableIn='1' THEN
      IF count = cPeriod-1 THEN
        NextCount<=0;
      ELSE
        NextCount<=count+1;
      END IF;
    END IF;
  END PROCESS p_NextState; -- End NextState Logic
  --
  -- Begin State Memory
  p_StateMemory:PROCESS(reset,clock)
  BEGIN
    IF reset='0' THEN
      count <= 0;
    ELSIF clock'EVENT AND clock='1' THEN
      count <= NextCount;
    END IF;
  END PROCESS p_StateMemory;
  --
  -- output logic:
  EnableOut <= '1' WHEN (count=cPeriod-1 AND EnableIn='1') ELSE '0';
END ARCHITECTURE rtl_counter_fsm2_up;
```

The above example reflects the state machine composed of *nextstate* logic and *state* memory. When choosing this solution, you should keep in mind the 2 rules for combinational logic:

1. Have all input signals in the sensitivity list of the combinational-logic process and
2. Drive every output-bit of the next-state logic any time in any situation. (Use `ELSE` and `OTHERS` keywords in `IF` and `CASE` statements, respectively!)

One method to guarantee point 2. above is to begin the combinational *nextstate* process with the following code line:

```
  state <= nextstate;
```

When several subsequent signal assignments occur within a process, the last assignments is valid. Therefore, we are sure that any bit of the state vector is driven in any situation and may now drive some of its bits to other values if desired.

2.5.2 One *PROCESS* Statement for *Nextstate* Logic and *State* Memory

There are two design techniques to model the feedback-loop of an FSM: (i) a single *VHDL PROCESS* statement for both and (ii) two *VHDL PROCESS* statements: one for the nextstate logic and the other for the state memory.

Listing 2.5.2: Architecture *rtl_counter_fsm1_dn* appended to file *counter.vhd*.

```
ARCHITECTURE rtl_counter_fsm1_dn OF counter IS
  SIGNAL nextcount:NATURAL RANGE 0 TO cPeriod-1;
BEGIN
  -- Counter
  p_fsm:PROCESS(reset,clock)
  BEGIN
    IF reset='0' THEN
      count <= 0;
    ELSIF clock'EVENT AND clock='1' AND EnableIn='1' THEN
      IF count = 0 THEN           -- begin nextstate logic
        count <= cPeriod-1;
      ELSE
        count <= count-1;
      END IF;                     -- end nextstate logic
    END IF;
  END PROCESS p_fsm;
  --
  -- output logic:
  EnableOut <= '1' WHEN (count=0 AND EnableIn='1') ELSE '0';
END ARCHITECTURE rtl_counter_fsm1_dn;
```

This realization uses the same memory model as the example with 2 process statements below, but replaces the next-state assignment `count<=NextCount;` by the combinational logic. Advantages:

- There are no other signals in the sensitivity list than `reset` and `clock`,
- There is no `ELSE` or `OTHERS` clause to define what happens when `enable`='0',
- Such code is often easier to read, shorter and less error prone to maintain.

To obtain the solution in listing 2.5.2 from listing 2.5.1 …

1. Replace statement `count<=nextcount;` (or `state<=nextstate;`) inside the memory by the next-state logic.
2. Rename `NextCount` to `count` (or `NextState` to `state`) and remove the signal declaration statement for the `nextcount` (or `nextstate`) vector.
3. Remove the process for the nextstate logic.

2.6 Compilation Order Dependence

VHDL allows for the declaration of packages. Declaration of package bodies is optional.

Listing 2.6: Declaration of package and package body

```
PACKAGE pk_constants IS
  CONSTANT pi:REAL:=3.1415926;
  CONSTANT dealy:TIME;  -- this is a deferred constant as no value assigned
  FUNCTION f_square(x:REAL:=0.0) RETURN REAL;
END PACKAGE pk_constants IS

PACKAGE BODY pk_constants IS
  CONSTANT delay:time:=2 ns; -- deferred constant gets value in pk. Body
  --
  -- initialize real numbers with 0.0 to prevent REAL'LEFT -> overflow risk
  FUNCTION f_square(x:REAL:=0.0) RETURN REAL IS
  BEGIN
    RETURN x*x;
  END FUNCTION f_square;
PACKAGE PACKAGE BODY pk_constants IS
```

2.7 Compilation Order Dependence

VHDL has no linker like *C*, *VHDL* has a loader. A *C* compiler compiles all modules of a design in an arbitrary order setting links for subprogram calls, and the linker assembles them to a project. A *VHDL* compiler loads instantiated modules immediately as compiled code. Consequently, loaded code must be available before instantiation. Later re-compilation has no effect. Therefore, *VHDL* projects have to be compiled bottom - up.

When compiling an entity, required packages (e.g. ieee.std_logic_1164) must be available in compiled form, when compiling an architecture the related entity must be available in compiled form and when compiling a configuration the related entities and architectures must be available in compiled form. This leads to the following compile sequence:

1. bottom level → top level
2. package → entity → related architecture(s) → configuration(s) → package body.

As seen from point 2. there is one important exception: Package bodies may compiled last. Consequently, we can declare a constant in a package and assign it a value in the package body. For big designs, only the package body need to be recompiled. This is not possible for constants declaring array sizes, but useful to test the circuit with different delays (e.g. typical, best case, worst case).

3 Structure

3.1 General Directory Structure

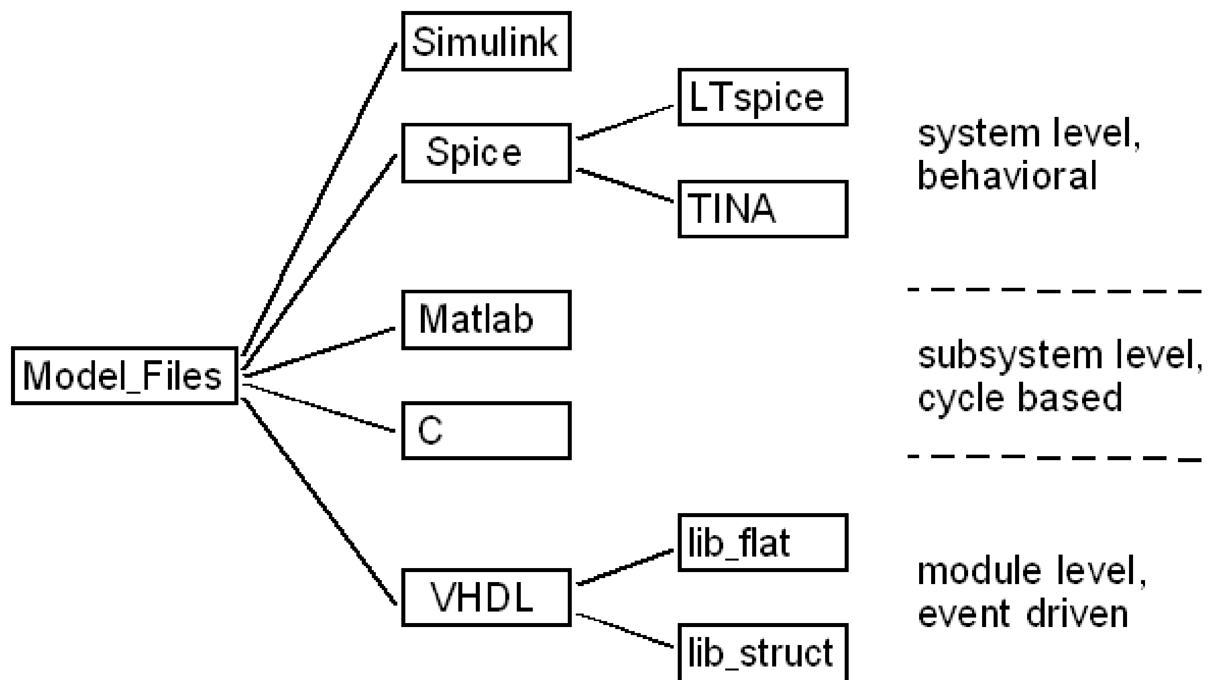


Fig. 3.1: General directory structure for all model files of this course

3.2 Directory Structure for *VHDL* Models

Fig. 3.2 illustrates the directory structure for all VHDL model files and testbenches.

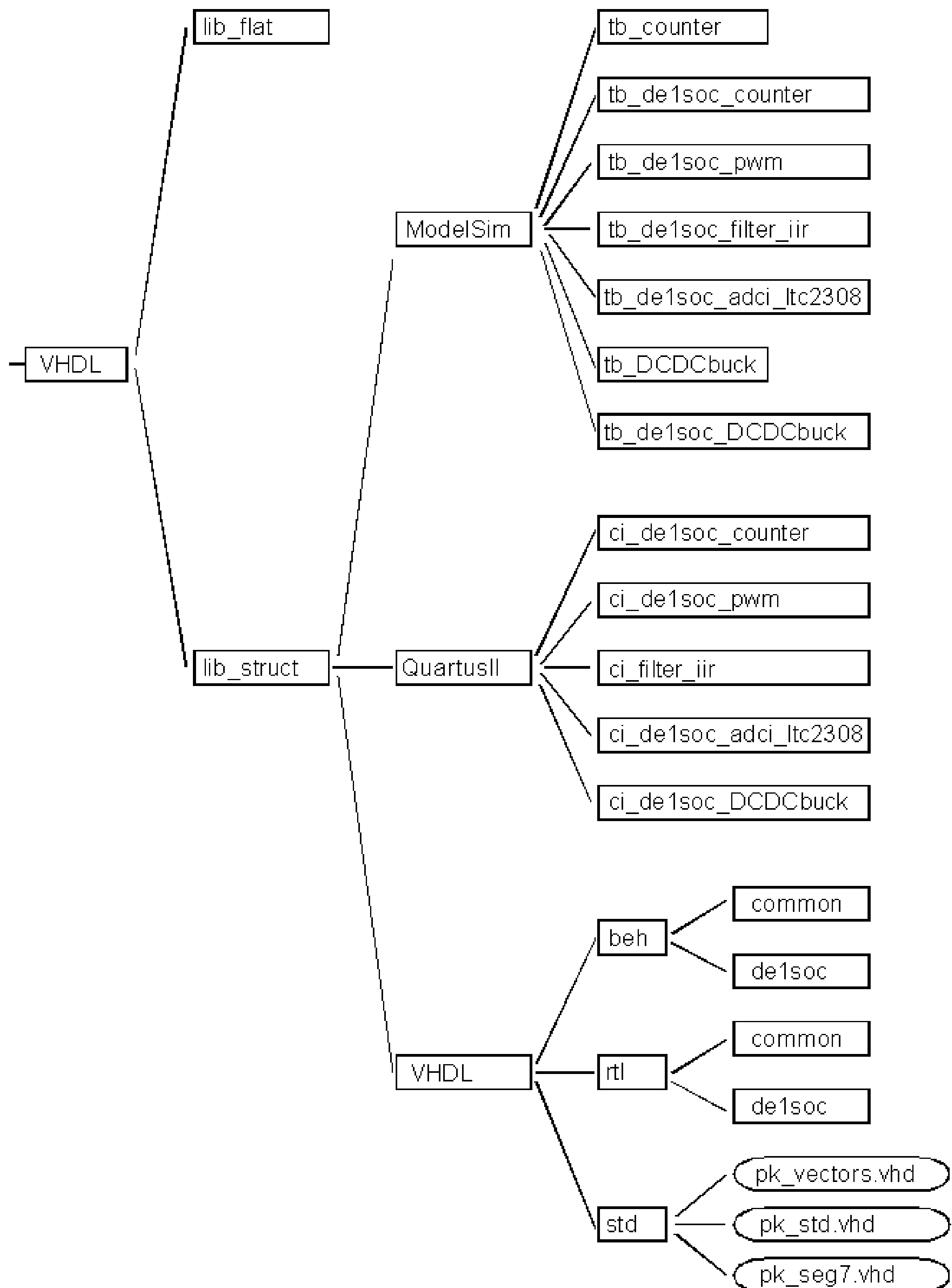


Fig. 3.2: Directory structure for all VHDL files of this course

Significance of directories and files

…\VHDL\lib_flat counter model with several architectures for simulation and synthesis to *DE1-SoC* board.

…\VHDL\lib_struct

\ModelSim: Testbenches

tb_counter:	from *flat* → *struct*: counter model with several architectures
tb_de1soc_counter	from *flat* → *struct*: counter model adopted for *DE1-SoC* board,
tb_de1soc_pwm	counter model extended to pulse-width modulator,
tb_filter_iir	tests different IIR filter models with order $R \leq 2$.
tb_de1soc_adci_ltc2308	testbench for interface to ADC *LTC2308* on *DE1-SoC* board.
tb_DCDCbuck	behavioral model of DC/DC buck converter, clocked with $f_s = 1/T_s$.
tb_de1soc_DCDCbuck	synthesizable model of DC/DC buck converter adopted for *DE1-SoC* board with *DCDCbuck* daughter board.

\QuartusII: Configuration interfaces related to *DE1-SoC* board

ci_de1soc_counter	configuration interface for counter model in *DE1-SoC* board,
ci_de1soc_pwm	configuration interface for pulse-width modulator,
ci_filter_iir	configuration interface to test synthesizable IIR filters, order $R \leq 2$.
ci_de1soc_adci_ltc2308	configuration interface to ADC *LTC2308* on *DE1-SoC* board.
ci_de1soc_DCDCbuck	ci to run *DE1-SOC* board with *DCDCbuck* daughter board.

\VHDL: Tool independent VHDL files

\beh: behavioral (=non-synthesizable) VHDL files

\common: hardware independent VHDL files

adc.vhd	general A/D converter model,
adc_ltc2308.vhd	behavioral model of ADC *LTC2308* on DE1-SoC board,
filterf_canon1.vhd	behavioral model of IIR filter in 1st canonical direct structure using floating point numbers as I/O data,

...

\rtl: register transfer level (=synthesizable) VHDL files

\common: hardware independent VHDL files

adci.vhd	synthesizable interface to general A/D converter model,
adci_ltc2308.vhd	synthesizable interface to ADC *LTC2308* on DE1-SoC board,
filteri_canon1.vhd	synthesizable model of IIR filter in 1st canonical direct structure using integer type numbers as I/O data,

...

\de1soc: VHDL interfaces to *DE1-SoC* board

de1soc_counter.vhd synthesizable interface to fit entity *counter* into *DE1-SOC* board,

...

\std: standard VHDL packages for this course

pk_vectors.vhd	declaration of data types integer_vector, real_vector,…, that are standard for some VHDL compilers and must not be compiled for them.
pk_std.vhd	declaration of some standard function used I this course.
pk_seg7.vhd	utilities to drive 7-sgment displays, e.g. of *DE1-SoC* board.

4 Modeling LTI Systems with VHDL

4.1 Data Structures to Describe LTI Models

The representation of LTI systems of order *R* is accomplished as vectors typically named *$TF_s* and *$TF_z* representing *$TF(s)* and *$TF(z)*, respectively, whereas *$* is a placeholder for a code letter; e.g. *CTF_s* and *CTF_z* for controller transfer functions. (*Matlab* variables with data type *sys* are named *TF$_s* and *TF$_z*, respectively.)

- Time-continuous systems *$TF_s* have vector length is 2*R*+4.
- Time-continuous systems *$TF_z* have vector length from 2*R*+3 to 2*R*+4, whereas missing trailing elements default to 0 with exception of c_0 which defaults to 1.

Table 4.1: Vector notations of LTI systems

(a) Time-continuous system data object

Time-continuous System:	$STF(s)=\dfrac{Y(s)}{X(s)}=\dfrac{a_0+a_1 s+...+a_2 s^{-R}}{b_0+b_1 z^{-1}+...+b_R z^{-R}}$							
Order *R*:	Contents of time- continuous LTI system object *$TF*(*s*) of order *R*							
Index #	1	2	3	…	*R*+3	*R*+4	…	2*R*+4
$TF_s(#)	*0*	*R*	a_0	···	a_R	b_0	…	b_R
Order *2*:	Contents of time-continuous LTI system object *$TF*(*s*) of order 2							
Index #	1	2	3	4	5	6	7	8
$TF_z(#)	*0*	*2*	b_0	c_1	b_2	c_2	c_1	c_2

(b) Time-discrete system data object, c_0 ... c_r may be omitted and default to 1 0…0

Time-discrete System:	$STF(z)=\dfrac{Y(z)}{X(z)}=\dfrac{d_0+d_1 z^{-1}++...d_R z^{-R}}{c_0+c_1 z^{-1}+...+c_R z^{-R}}$							
Order *R*:	Contents of time-discrete LTI system object *$TF*(*z*) of order *R*							
index	1	2	3	…	*R*+3	*R*+4	…	2*R*+4
content	T_s	*R*	d_0	···	d_R	c_0	…	c_R
Order *2*:	Contents of time-discrete LTI system object *$TF*(*z*) of order 2							
index	1	2	3	4	5	6	7	8
content	T_s	*2*	d_0	d_1	d_2	c_0	c_2	c_R

Function *f_filterf_canon1* realizes an Rth order filter in 1st canonical direct structure using floating point Input/output data with *STF_z* defining the time-discrete system transfer function. Statement

```
ASSERT STF_z(1)=0.0 REPORT "rtl_filterf_canon1 Ts is not 0" SEVERITY ERROR;
```

prints the report string with severity note if the condition T_s = *STF_z*(1)=0 is not fulfilled, i.e. when the system *STF_z* it not time-discrete. Statement

```
CONSTANT R:NATURAL:= INTEGER(round(STF_z(2))) -- R = oRder
```

Writes the filter order to the constant named *R*. As function INTEGER( ) truncates, we use function *round*( ) to prevent that some bits of round-noise disturb *STF_z*(2). Statements

```
CONSTANT d:real_vector(0 TO R):=STF_z(3 TO R+3);
CONSTANT c:real_vector(0 TO R):=STF_z(R+4 TO 2*R+4);
```

copy the respective parts of *SFT_z* to coefficients vectors *d(0:R)* and *c(0:R)*.

In Function *f_filterf_canon1*, that uses integral numbers as filter I/O, we find the statements

```
CONSTANT d:integer_vector(0 TO R):=f_mult(STF_z(3 TO R+3),2.0**NoFract);
CONSTANT c:integer_vector(0 TO R):=f_mult(STF_z(R+4 TO 2*R+4),2.0**NoFract);
```

Parameter *NoFract* stands for Number of Fractional bits. We may multiply numerator and denominator of *STF*_s and/or *STF*_z with a factor *F*. This allows for better resolution of coefficients but makes a division by c_0 necessary, which can be realized with a bit-shift operation when $F=2^N$, with *N* being an integral number.

Fig. 4.1 reminders to keep in mind synchronously clocked finite state machine design also for hardware description with *VHDL*.

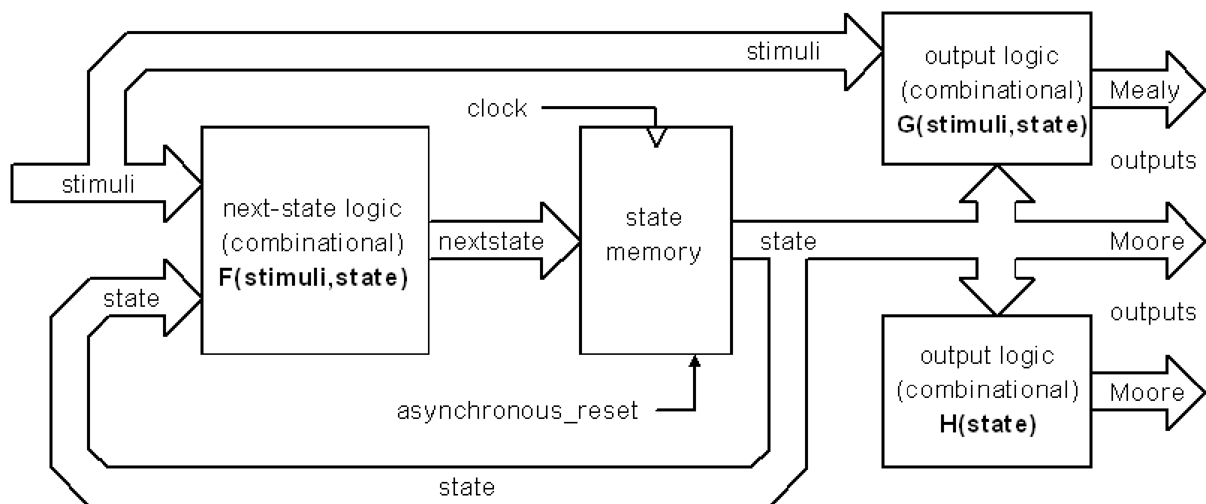
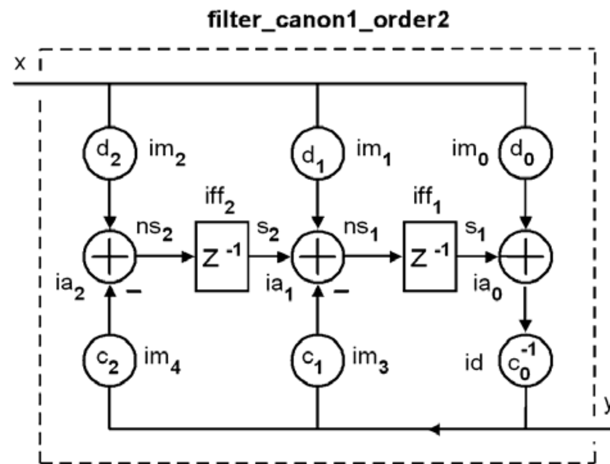


Fig. 4.1: General finite state machine (FSM) model.

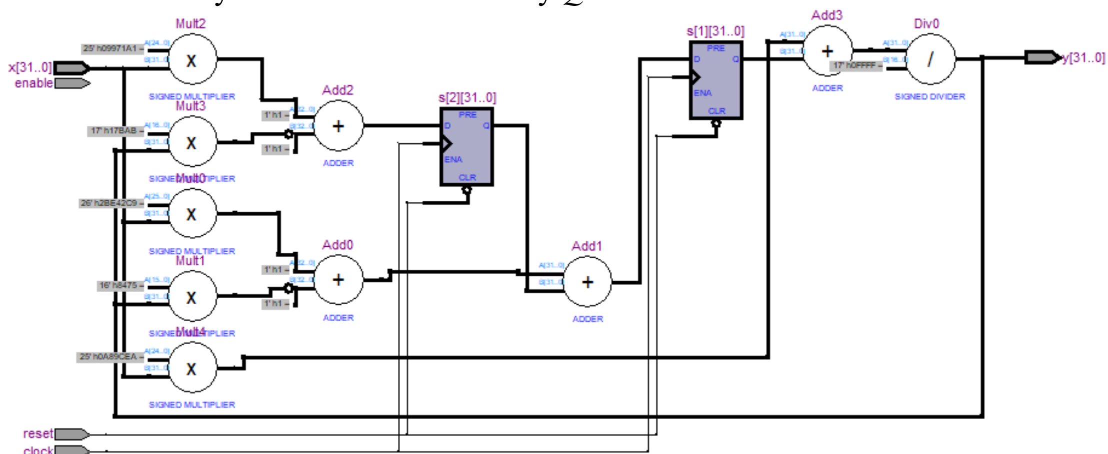
4.2 IIR Filter Description for Synthesis

(a) Schematics of digital 2nd order IIR filter in 1st canonical direct structure

before translation to *VHDL* code



(b) *RTL View* **after** synthesis of *VHDL* code by *Quartus II 8.1*



(c) Timing information of the synthesizer delivers timing information such as t_{su}, t_{co}, t_{pd}:

```
Clock "clock" has Internal fmax of 11.16 MHz …  (period= 89.586 ns)
tsu for register .......        ("x[18]", "clock") is -99.523 ns
tco from "clock" to destination pin "y[29]"       is 90.072 ns
Longest tpd from pin "x[18]" to pin "y[29]" is 100.009 ns
```

(d) Timing information explanation

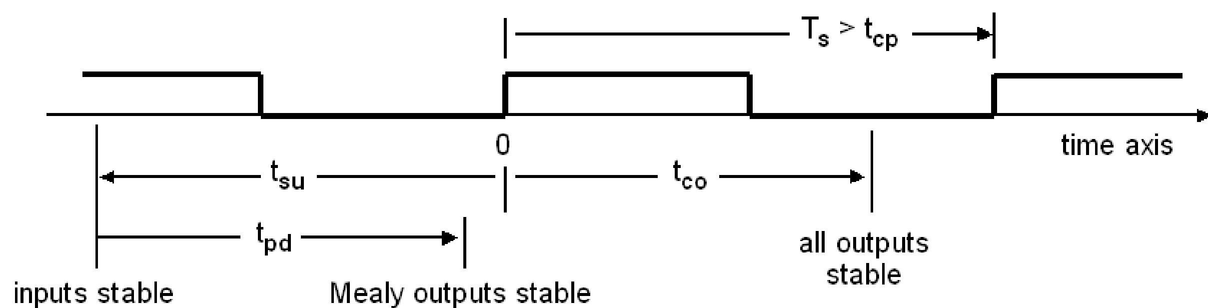


Fig. 4.2: Synthesis of a 2nd order IIR filter with *Quartus II* 8.1 [10].

Fig. 4.2(a) illustrates a 2nd order IIR filter in 1st canonic direct structure. It has an infinite impulse response (IIR) due to the feedback loops and is canonic, as the order of the filter's transfer function equals the number of memories.

Fig. 4.2(b) illustrates the schematics synthesized *Quartus* II 8.1 for *Cyclone II* device *EP2C35F672C6* [6] (Tools > Netlist Viewers > RTL Viewer) of the respective VHDL source code.

Fig. 4.2(c) is output from synthesis with Quartus II 8.1 [10].

Fig. 4.2(d) illustrates the significance of the timing information, detailed in table 4.2.

Table 4.2: Characteristic times and frequencies

Symbol	stands for	meaning		
f_s	sampling frequency	Frequency of data samples being processed		
T_s	sampling period	= 1/f_s		
t_{su}	setup time	Time span that all input signal to nextstate logic must be stable before active clock edge		
t_{co}	clock-to-output time	Time span requried after active clock edge until all output signals are stable		
t_{pd}	propagation delay	Time span from input signals stable until all Mealy outputs stable		
t_{cp}	critical path delay	Time span required between two clock edges: delay through nextstate logic		
f_{smax}	maximum sampling frequency	=1/t_{cp}		

Exercise: Identify the instantiated components of Fig. 4.2(a) in Fig. 4.2(b):

Component	**Fig. 4.1(a)**	**Fig. 4.1(b)**		
State memory for s1	iff_1	s[1]		
State memory for s2	iff_2	s[2]		
Adder of d_0 x + s_1	ia_0	Add3		
Adder of d_1 x + s_2 – c_1 y	ia_1	Add0 + Add1		
Adder of d_2 x – c_2 y	ia_2	Add2		
Division by c_0	id	Div0		
Multiplication by c_1	im_3	Mult1		
Multiplication by c_2	im_4	Mult2		
Multiplication by d_0	im_0	Mult4		
Multiplication by d_1	im_1	Mult0		
Multiplication by d_2	im_2	Mult2		

Explain the timing information t_{su}, t_{co} and t_{pd} in Fig. 4.2(c).

t_{su} …
… stands for setup time… .... value in Fig. 4.2(c) -99,523 ns ....

… meaning: inputs must be stable for tsu before active clock edge ....

t_{co} …
… stands for clock-to-output… .... value in Fig. 4.2(c) 90.072 ns ....

… meaning: time until all outputs stable after active clock edge ....

t_{pd} …
… stands for propagation delay… .... value in Fig. 4.2(c) 100.009 ns ....

… meaning: delay from stimulus change to mealy output change ....

t_{cp} : Critical path delay: What is the critical path delay of an FSM design?

biggest delay of neststate logic, limits maximum frequency
....

Computed t_{cp} From Fig. 4.2(c)

t_{cp} = 1/fmax = 1 / 11.16 MHz = 89.6 ns
....

4.3 VHDL Model of Behavioral IIR Filter

4.3.1 Second Order, 1st Canonical Direct Structure

Listing 4.2.1: Behavioral VHDL model of *filterf_canon1_order2*.

```
USE work.pk_std.ALL; USE work.pk_vectors.ALL;
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
ENTITY filterf_canon1_order2 IS
    GENERIC(order:NATURAL:=2;    -- order of LTI system
      STF_z:real_vector;         -- time-discrete LTI system
      NoFract:NATURAL:=16);          --  Number  of  fractional  bits  in
coefficients
    PORT(
      reset :IN std_logic;       -- reset, low active
      clock :IN std_logic;       -- sampling clock, rising edge active
      enable:IN std_logic;       -- enables sampling clock when '1'
      x     :IN     REAL:=0.0;   -- input
      y     :BUFFER REAL:=0.0);  -- output
END ENTITY filterf_canon1_order2;

ARCHITECTURE rtl_filterf_canon1_order2 OF filterf_canon1_order2 IS
  CONSTANT d:real_vector(0 TO order):=STF_z(1 TO order+1);
  CONSTANT c:real_vector(0 TO order):=STF_z(order+2 TO 2*order+2);
  SIGNAL   ns, s:real_vector(1 TO 2):=(OTHERS=>0.0);
BEGIN
  --
  -- NextState logic
    ns(1) <= d(1)*x - c(1)*y + s(2);
    ns(2) <= d(2)*x - c(2)*y;
  --
  -- State memory
  p_statememory:PROCESS(reset,clock)
  BEGIN
    IF reset='0' THEN
      s <=(OTHERS=>0.0);
    ELSIF clock'EVENT AND clock='1' AND enable='1' THEN
      s <= ns;
    END IF;
  END PROCESS p_statememory;
  --
  -- output logic, Mealy:
  y <= (d(0)*x + s(1)) WHEN c(0)=1.0 ELSE (d(0)*x + s(1))/c(0);
END ARCHITECTURE rtl_filterf_canon1_order2;
```

Listing 4.2.1 shows a VHDL model of the 2nd order IIR filter illustrated in Fig. 4.2(a). In the listing above I/O signals *x* and *y* are declared as floating-point type REAL, so that it is typically not synthesizable, but we can test it without need of A/D and D/A converters.

Exercise:

- Complete listing 4.2.1 and test the code in the *ModelSim* testbench *tb_filter* in the structural directory tree of *Model_Files/VHDL/lib_struct*. Use concurrent code only for the nextstate logic and a process computing nextstate ns_1, ns_2 and a process for the state memory holding state s_1, s_2.
- Compare the results simulated with testbench *Model_Files/VHDL/lib_struct/ModelSim/...* with *Matlab* code *tb_filter* in directory *Model_Files/Matlab/*.

4.3.2 Arbitrary Order, 1st Canonical Direct Structure

Listing 4.2.2: Behavioral VHDL model of *filterf_canon1*.

```
USE work.pk_std.ALL; USE work.pk_vectors.ALL;
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
ENTITY filterf_canon1 IS
    GENERIC(order:NATURAL:=2;     -- order of LTI system
      STF_z:real_vector;          -- time-discrete LTI system
      NoFract:NATURAL:=16);          --  Number  of  fractional  bits  in
coefficients
    PORT(
      reset :IN std_logic;        -- reset, low active
      clock :IN std_logic;        -- sampling clock, rising edge active
      enable:IN std_logic;        -- enables sampling clock when '1'
      x     :IN     REAL:=0.0;    -- input
      y     :BUFFER REAL:=0.0);   -- output
END ENTITY filterf_canon1;

ARCHITECTURE rtl_filterf_canon1 OF filterf_canon1 IS
  CONSTANT d:real_vector(0 TO order):=STF_z(1 TO order+1);
  CONSTANT c:real_vector(0 TO order):=STF_z(order+2 TO 2*order+2);
  SIGNAL   ns, s:real_vector(1 TO order):=(OTHERS=>0.0);
BEGIN
  --
  -- NextState logic
  p_nextstate:PROCESS(x,y,s)
  BEGIN
    ns <= s; -- optional statement, makes sure ns is driven in any case
    l_ns:FOR k IN ns'RANGE LOOP
      IF k < order THEN
        ns(k) <= d(k)*x - c(k)*y + s(k+1);
      ELSE
        ns(k) <= d(k)*x - c(k)*y;
      END IF;
    END LOOP l_ns;
  END PROCESS p_nextstate;
  --
  -- State memory
  p_statememory:PROCESS(reset,clock)
  BEGIN
    IF reset='0' THEN
      s <=(OTHERS=>0.0);
    ELSIF clock'EVENT AND clock='1' AND enable='1' THEN
      s <= ns;
    END IF;
  END PROCESS p_statememory;
  --
  -- output logic, Mealy:
  y <= (d(0)*x + s(1)) WHEN c(0)=1.0 ELSE (d(0)*x + s(1))/c(0);
END ARCHITECTURE rtl_filterf_canon1;
```

Exercise:

- Complete listing 4.2.2 and test the code in the *ModelSim* testbench *tb_filter* in the structural directory tree of *Model_Files/VHDL/lib_struct*.
- Compare the results simulated with testbench *Model_Files/VHDL/lib_struct/ModelSim/...* with *Matlab* code *tb_filter* in directory *Model_Files/Matlab/*.

4.4 VHDL Model of Synthesizable IIR Filter of 2nd Order

4.4.1 Designing the VHDL Model

Listing 4.3: Synthesizable VHDL model of *filteri_canon1_order2*.

```
USE work.pk_std.ALL; USE WORK.pk_vectors.ALL;
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
ENTITY filteri_canon1_order2 IS
    GENERIC(order:NATURAL:=2;     -- order of LTI system
      STF_z:real_vector;          -- LTI system
      NoFract:NATURAL:=16);              --  Number  of  fractional  bits  in
coefficients
                                       PORT(
      reset :IN std_logic;        -- reset, low active
      clock :IN std_logic;        -- sampling clock, rising edge active
      enable:IN std_logic;        -- enables sampling clock when '1'
      x     :IN     INTEGER:=0;   -- input
      y     :BUFFER INTEGER:=0);  -- output
END ENTITY filteri_canon1_order2;

LIBRARY ieee; USE ieee.std_logic_signed."+",ieee.std_logic_signed."*";
ARCHITECTURE rtl_filteri_canon1_order2 OF filteri_canon1_order2 IS
  CONSTANT d:integer_vector(0 TO order):=
                              f_mult(STF_z(1 TO order+1),2.0**NoFract);
  CONSTANT c:integer_vector(0 TO order):=
                      f_mult(STF_z(order+2 TO 2*order+2),2.0**NoFract);
  SIGNAL ns, s:integer_vector(1 TO order):=(OTHERS=>0);
BEGIN
  --
  -- NextState logic
    ns(1) <= d(1)*x - c(1)*y + s(2);
    ns(2) <= d(2)*x - c(2)*y;
  --
  -- State memory
  p_statememory:PROCESS(reset,clock)
  BEGIN
    IF reset='0' THEN
      s <=(OTHERS=>0);
    ELSIF clock'EVENT AND clock='1' AND enable='1' THEN
      s <= ns;
    END IF;
  END PROCESS p_statememory;
  --
  -- output logic, Mealy:
  y <= (d(0)*x + s(1)) WHEN c(0)=1 ELSE (d(0)*x + s(1))/c(0);
END ARCHITECTURE rtl_filteri_canon1_order2;
```

Exercise:

- Complete listing 4.2.1 and test the code in the *ModelSim* testbench *tb_filter* in the structural directory tree of *Model_Files/VHDL/lib_struct*. Use <u>concurrent code only for the nextstate logic</u> computing nextstate ns_1, ns_2 and a process for the state memory holding state s_1, s_2.
- Compare the results simulated with testbench *Model_Files/VHDL/lib_struct/ModelSim/...* with *Matlab* code *tb_filter* in directory *Model_Files/Matlab/*.

4.4.2 Testing the VHDL Model

We typically seek to scale coefficients of

$$STF(z) = \frac{Y(z)}{X(z)} = \frac{d_0 + d_1 z^{-1} + d_2 z^{-2}}{c_0 + c_1 z^{-1} + c_2 z^{-2}}$$

and consequently

$$y_n = c_0^{-1}\,[d_0 \cdot x_n + d_1 \cdot x_{n-1} + d_2 \cdot x_{n-2} + c_1 \cdot y_{n-1} + c_2 \cdot y_{n-2}]$$

such, that c_0=1 so that we can omit the division by c_0. Look into the controller model of the synthesis above. What floating-number values do you find for the coefficients

d_0 = **0.084970…** ...., d_1 = **0.16994…** ...., d_2 = **0.08497…** ....

c_0 = **1.0** ...., c_1 = **-1.5519…** ...., c_2 = **0.8918…** ....

Rounding these numbers to integral values would yield a tremendous error. Consequently, we have to respect fractional bits after the binary point. Therefore, statements

```
CONSTANT d:integer_vector(0 TO order):=
                            f_mult(STF_z(1 TO order+1),2.0**NoFract);
```

multiply all coefficients with $2^{NoFract}$ with *NoFract* being the Number of Fractional bits. The integer coefficients are then computed from the floating-point coefficients as

```
icoef(k) = integer(round(fcoef(k)*2**NoFract))
```

with double asterisk "**" standing for potentiation: VHDL expression 2**K $= 2^K$.

In the VHDL simulation with *ModelSim* the setting is *NoFract*=8.

What happens for *NoFract* = 16?

The controller output becomes more accurate as 16 fractional ....

bits are respected in the computation ....

What happens for *NoFract* = 22?

The controller output wrong due to number overflow as for 22 ....

fractional bits only 10 integral bits left: range -512 … 511 ....

4.4.3 Synthesizing the VHDL Model

Synthesize the VHDL Model of listing 4.3.1. As synthesizable frame (replacing the testbench) use configuration interface *ci_filteri_canon1_order2* in directory *Model_Files/VHDL/lib_struct/QuartusII/ci_filteri_canon1_order2/*.

Exercises:

Use *RTL Viewer* to see the result. Find parameters t_{su}, t_{co}, t_{pd}, t_{cp} in the transcript window. (Hint: This is probably much easier in *Quartus II 8.1* or *Quartus II 10* for *Cyclone II* device *EP2C35F672C6* rather than *Quartus II 18*.)

t_{su} = -22.170 ns .... , t_{co} = 14.641 ns .... , t_{pd} = 23.338 ns .... , t_{cp} = 13.473 ns ....

In the architecture *filteri_canon1_order2* change the code fragment from

```
  CONSTANT d:integer_vector(0 TO order):=
                              f_mult(STF_z(1 TO order+1),2.0**NoFract);
  CONSTANT c:integer_vector(0 TO order):=
                      f_mult(STF_z(order+2 TO 2*order+2),2.0**NoFract);
f_mult(STF_z(order+2 TO 2*order+2),2.0**NoFract);
```

to

```
  CONSTANT d:integer_vector(0 TO order):=
                              f_mult(STF_z(1 TO order+1),2.0**NoFract-1);
  CONSTANT c:integer_vector(0 TO order):=
                      f_mult(STF_z(order+2 TO 2*order+2),2.0**NoFract-1);
```

Synthesize again. What is the difference in the design? What are the new delays?

t_{su} = 113.052 ns .... , t_{co} = 102.093 ns .... , t_{pd} = 110.590 ns .... , t_{cp} = 104.555 ns ....

What is the difference and how can it be explained?

Division by $2^{NoFract}$ can be realized by a bit-shift operation ....

rather than a time-comsuming numerical division ....

4.4.4 Applying the VHDL Model

Your models of *filteri_canon1_order2* should be able to replace 2nd order filter models with same results as *filteri_canon1* in the simulations and hardware of the DC/DC buck converter!

5 Advanced Topics: More Sophisticated VHDL Modeling

This chapter points out some deeper issues of VHDL in application to an example of FIR filter design.

5.1 Design Units

Table 5.1: Building blocks in VHDL

	COMPONENT	LIBRARY	Comments
To the outer world	ENTITY	PACKAGE	Corresponds to a symbol
Realization inside	ARCITECTURE	PACKAGE BODY	Corresponds to a schematics
Combination	CONFIGURATION		

5.2 Compilation Order Dependence

VHDL is compilation order dependent. It has no linker but only a loader. Sub-modules are loaded immediately during the compilation process. We can organize our VHDL code arbitrarily in different files, but it is important that code must be compiled before being instantiated. The compilation sequence is

PACKAGE → ENTITY → ARCHITECTURE → CONFIGURATION → PACKAGE BODY

The package body may be compiled directly after the package but also as lastly.

5.3 Kinds of Code: Concurrent – Sequential – Structural

VHDL code can be written in three different modes: Concurrent is the default mode. In concurrent code, the sequence of statements is irrelevant. Instantiation of components is structural. Code within a PROCESS statement or subprograms code is sequential, i.e. it is processed top-down.

5.4 Data

5.4.1 Data Objects

VHDL handles data by one of the following data objects:

```
1.SIGNAL:     interconnection wires
2.VARIABLE:   local storage of temporary data
3.CONSTANT:   named constant values
```

Table 5.4.1 shows in which environment a data object may be declared and where it may be used, i.e. assigned or read.

Table 5.1: The three data objects and the environments where they may be declared and used.

data object	environment to be declared in	environment to be used in
SIGNAL	concurrent	everywhere
VARIABLE	sequential	sequential
CONSTANT	everywhere	everywhere

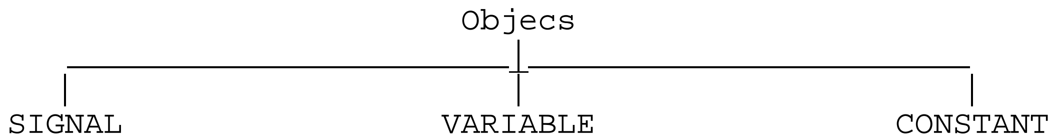


Fig. 5.4.1: VHDL data objects.

5.4.2 Data Types

Data object have to be declared with a data type. Available are scalar the types `INTEGER`, `REAL`, enumerated and `BIT` and the composite data types such `ARRAY` and `RECORD`.

Scheme of data type declaration statements:

```
TYPE type_name IS type_mark;
SUBTYPE subtype_name IS <range_specification> OF type_name;
```

Examples for type and subtype declarations:

```
TYPE BOOLEAN IS (false,true);
SUBTYPE t_address IS RANGE 0 TO 15 OF INTEGER;
SUBTYPE NATURAL IS RANGE 0 TO INTEGER'HIGH OF INTEGER;
```

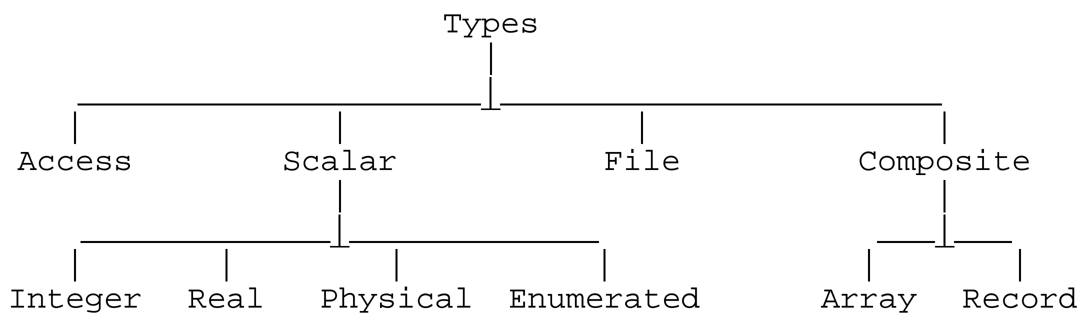


Fig. 5.4.2: VHDL data types.

The assignment of a value to a data object is guarded such, that only values of the same data type or a subtype can be assigned to a data object. The user can specifiy own data types. See e.g. [Schubert: VHDL Skript] Chapter 2: Data for more details.

5.5 Libraries and Packages

5.5.1 Using Existing Libraries and Packages

Reinventing the wheel is inefficient. Better reuse existing code if possible. It is typically organized in libraries, which are composed of packages. A typical library retrieval is

```
(1) LIBRARY ieee;
(2) USE ieee.std_logic_1164.ALL;
(3) USE ieee.std_logic_unsigned."+";
(4) LIBRARY adac_lib;
(5) USE adac_lib.pk_adac.ALL;
(6) USE work.pk_mypack.ALL;
```

The code above has the following meanings:

Line (1): "Retrieve library with name `ieee`".

Line (2): Use `ALL` from package `std_logic_1164` found within library `ieee`.

Line (3): Use only the declaration of the "+" operator found in package `std_logic_unsigned` within library `ieee`. Applied on bit-vectors as operands it will be synthesized as arithmetic summation and the most significant bit will not be interpreted as sign bit. (To treat the first bit as sign bit use the "+" operator from package `std_logic_signed`.)

Line (4): "Retrieve library with name `adac_lib`". As it is not a standard library it must be introduced to the tool, the respective commands are tool dependent.

Line (5): Use `ALL` declarations from package `pk_adac` found within library `adac_lib`.

Line (6): Use `ALL` declarations from package `pk_mypack` found within library `work`. The working library `work` needs no LIBRARY statement, at is always linked.

If the LIBRARY / USE statements are written above ...

- an ENTITY, then they are valid for this entity and its architectures,
- an ARCHITECTURE, then they are valid for this architecture,
- a PACKAGE, then they are valid for this package and its package body,
- a PACKAGE BODY, then they are valid for this PACKAGE BODY.

5.5.2 Creating an Own Package

Package and package body are like entity and architecture. Packages may contain non-executable declarations only while package bodies may contain executable code also. Declarations made within a package are available whenever loading this package. Declarations made in a package body are known only in this package body below the declaration. Listing 4.2 shows Tcl-commands to create a library and symbol for it that can be used within the VHDL code when working with the *ModelSim* simulator.

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE pk_example IS
  -- Declaration of an externally visible constant:
  CONSTANT cExtern:INTEGER:=10;
  --
  -- gate delay as "deferred" constant: no value assigned:
  CONSTANT delay:TIME;
  --
  -- multiplexer: interface declaration only, no executable code:
  FUNCTION mux(sel:INTEGER;vec:std_logic_vector) RETURN std_logic;
  --
  SIGNAL big_array:std_logic_vector(1 TO 40_000);
END PACKAGE pk_example;

PACKAGE BODY pk_example IS
  -- Declaration of an only internally visible constant:
  CONSTANT cIntern:INTEGER:=20;
  --
  -- here the deferred constant has to get its value:
  CONSTANT delay:TIME:=2 ns;
  --
  -- here the function mux has to get its body:
  FUNCTION mux(sel:INTEGER;vec:std_logic_vector) RETURN std_logic IS
  BEGIN
    RETURN vec(sel);
  END FUNCTION mux;
END PACKAGE BODY pk_example;
```

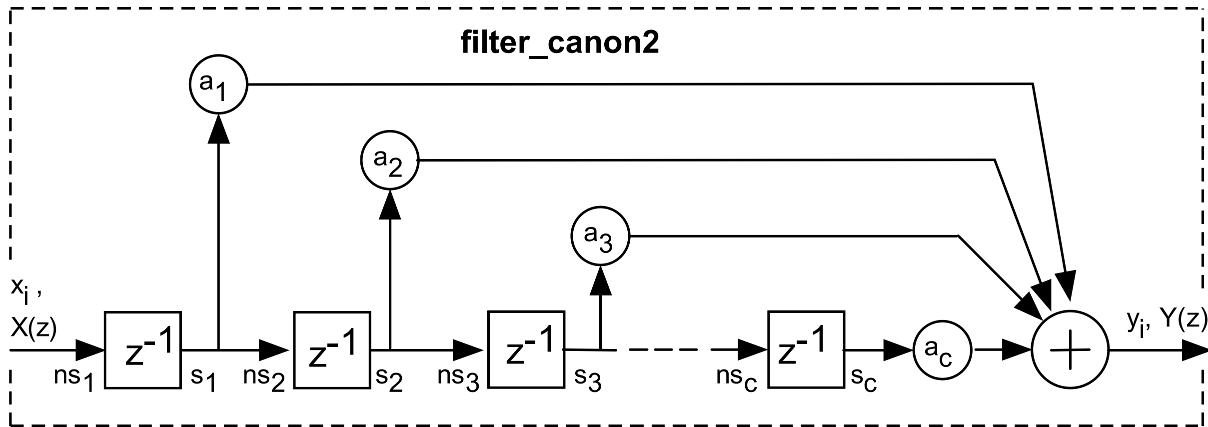
The package named `pk_example` above declares the constant `cExtern`. It will be available in the package body and everywhere where this package is declared. This is different from the constant `cIntern` declared within the package body below. It will be known only within this package body and below its declaration.

The constant `delay` is a so-called deferred constant, as no value is assigned to it in the package. The assignment is deferred (Latin: carried away) into the package body. As the package body may be compiled as last design unit, different delays (e.g. for fast, typical, slow parameters) can be passed to the design by compiling nothing else than the package body.

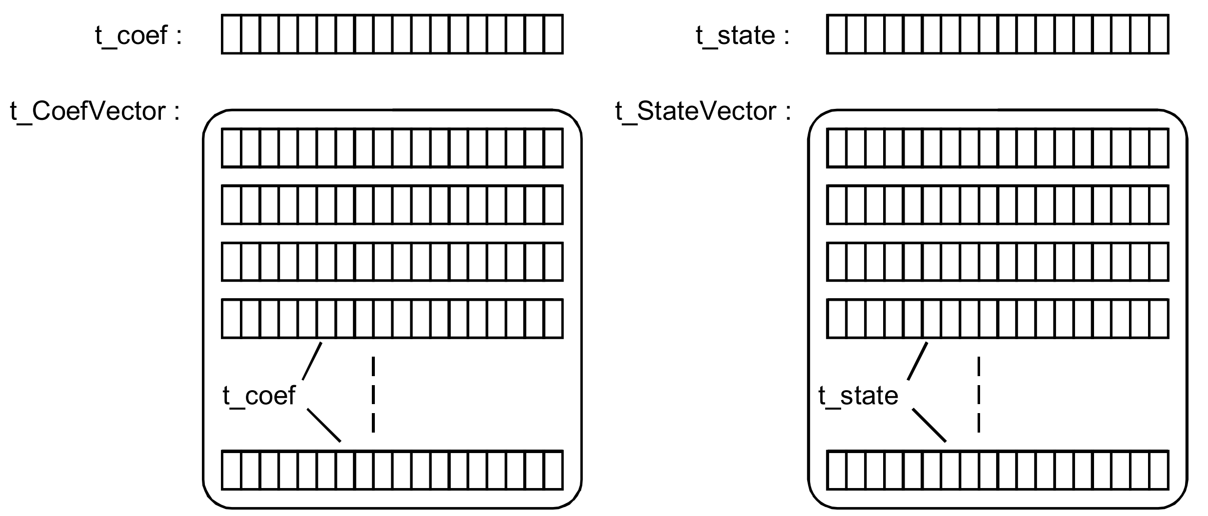
The external interface of the multiplexer function `mux` is declared in the package. In the package body function `mux` gets its executable body. If the declaration in the package is omitted, this function will be known only in the package body and after its declaration.

Signal `big_array` will be declared in any design unit where package `pk_example` is declared. It will be different signals which are independent from each other in any of those design units.

5.6 Arrays for State-Machine Design



(a) Digital Filter in 2nd canonical direct structure, a_i, s_i and ns_i (i=1...c) are bit-vectors.



(b) Data structures to realize the vectors of coefficients (a_i) and data words (s_i, ns_i).

Fig. 5.6: (a) A model that requires a vector of bit-vectors and **(b)** a VHDL realization.

Fig. 5.6(a) illustrates a digital filter that requires vectors of bit-vectors. Fig. 5.6 (b) and listing 5.6 realize a single coefficient a_i alias `CoefVector(i)` as bit-vector of type `t_coef`. All coefficients are summarized in the constant `CoefVector` of type `t_CoefVector`. A single state or next-state is realized as bit-vector of data type `t_state`. A vector of such signals, e.g. named `state` or `NextState`, can be declared as signal of type `t_StateVector`.

Listing 5.6 (a): The package *pk_filter*.

(Note: Some synthesizers cannot assign the values of "deferred constant" **`CoefVector`** in the package body. In this case remove the package body and assign the values in the package. Doing so you loose the capability of changing filter coefficients by recompiling the package bode only, you have to recompile the entire design.)

```
(1)  ---------------------------------------------------------------------
(2)  --  Module            : pk_filter
(3)  --  Designer          : Martin Schubert
(4)  --  Date last modified: 07.03.2011
(5)  --  Purpose           : Data Structures for Digital FIR Filter
(6)  ---------------------------------------------------------------------
(7)  LIBRARY ieee; USE ieee.std_logic_1164.ALL;
(8)  PACKAGE pk_filter IS
(9)    CONSTANT cDataInWidth:POSITIVE:=18;  -- Input-Data BitWidth
(10)   CONSTANT cDataInFract:POSITIVE:=16;  -- No of Input-Data fract. Bits
(11)   CONSTANT cDataOutWidth:POSITIVE:=18; -- Output-Data BitWidth
(12)   CONSTANT cDataOutFract:POSITIVE:=16; -- No of Output-Data fract Bits
(13)   CONSTANT cCoefWidth:POSITIVE:=18;    -- Coefficient's BitWidth
(14)   CONSTANT cCoefFract:POSITIVE:=18;    -- No of Coef's fractional Bits
(15)   SUBTYPE  t_DataIn  IS std_logic_vector(cDataInWidth-1  DOWNTO 0);
(16)   SUBTYPE  t_DataOut IS std_logic_vector(cDataOutWidth-1 DOWNTO 0);
(17)   SUBTYPE  t_coef IS std_logic_vector(cCoefWidth-1 DOWNTO 0);
(18)   TYPE     t_CoefVector IS ARRAY(NATURAL RANGE <>) OF t_coef;
(19)   CONSTANT nCoefs:POSITIVE:=33;
(20)   CONSTANT CoefVector: t_CoefVector(1 TO nCoefs);
(21) END PACKAGE pk_filter;
```

Listing 5.6 (b): The package body *pk_filter*.

```
(22) PACKAGE BODY pk_filter IS
(23)   CONSTANT CoefVector: t_CoefVector(1 TO nCoefs)
(24)                    := (OTHERS=>(cCoefWidth-6=>'1',OTHERS=>'0'));
(25) END PACKAGE BODY pk_filter;
```

5.7 Exercise: (Solutions at the end of this sub-chapter) **(Σ=16P)**

Complete package `pk_filter` in listing 5.7.

Make sure that data type `std_logic` and vectors with elements of this type can be used according to the respective IEEE standard. **(2P)**

Complete line (10) such, that data type `t_DataIn` declares a vector representing a number with `cDataInWidth` bits of type `std_logic` . **(1P)**

Complete line (11) such, that data type `t_coef` declares a vector representing a number with `cCoefWidth` bits of type `std_logic` . **(1P)**

Complete line (12) such, that data type `t_CoefVector` declares a vector with elements type `t_coef` . The index range of this vector can be defined later with natural numbers. **(2P)**

No value is assigned to the constant in line (14). Where does this constant gets its value and what is the correct denomination of such a constant? **(2P)**

....

What do you write over an Entity to make all declarations in Package pk_filter available? **(2P)**

....

Listing 5.7: Package *pk_filter*.

```
(1)  .....................................................................

(2)  .....................................................................
(3)  PACKAGE pk_filter IS
(4)    CONSTANT cDataInWidth:POSITIVE:=8;    -- Input-Data BitWidth
(5)    CONSTANT cDataInFract:POSITIVE:=6;    -- No of Input-Data fract. Bits
(6)    CONSTANT cDataOutWidth:POSITIVE:=18; -- Output-Data BitWidth
(7)    CONSTANT cDataOutFract:POSITIVE:=16; -- No of Output-Data fract Bits
(8)    CONSTANT cCoefWidth:POSITIVE:=18;     -- Coefficient's BitWidth
(9)    CONSTANT cCoefFract:POSITIVE:=18;     -- No of Coef's fractional Bits

(10)   ...TYPE  t_DataIn  IS .............................................

(11)   ...TYPE  t_coef    IS .............................................

(12)   ...TYPE  t_CoefVector IS ..........................................
(13)   CONSTANT nCoefs:POSITIVE:=33;
(14)   CONSTANT CoefVector: t_CoefVector(1 TO nCoefs);
(15) END PACKAGE pk_filter;
```

Write a library statement at (16), that allows for the multiplication of numbers in the *std_logic_vector*-format using '*' (while library is known). **(1P)**

Line (17): Declaration of type `t_product` as funktion of constants such, that a signal of that type matches the product of `t_DataIn` and `t_coef` type signals. **(1P)**

Line (18): declaration of signal `product` such, that we can write the VHDL command: "`product<=DataIn*CoefVector(i);`" **(2P)**

Lines (20) and (21): Complete the declarations of `iPl` und `iPh` such, that (22) works respecting the vector lengths and the number of fractional bits using the respective named constants in package `pk_filter`. (For the computation of `iPl`, `iPh` see document "FSM Design for DSP Using Fixed-Point Numbers" [10]). **(2P)**

```
(16) ...................................................................

(17) ...................................................................

(18) ...................................................................

(19) SIGNAL DataOut:t_DataOut;

(20) CONSTANT iPl: ......................................................

(21) CONSTANT iPh: ......................................................

(22) DataOut <= product(iPh DOWNTO iPl);
```

Solutions:

```
(1)    LIBRARY ieee;
(2)    USE ieee.std_logic_1164.ALL;
(3)    PACKAGE pk_filter IS
(4)      CONSTANT cDataInWidth:POSITIVE:=8;    -- Input-Data BitWidth
(5)      CONSTANT cDataInFract:POSITIVE:=6;    -- No of Input-Data fract. Bits
(6)      CONSTANT cDataOutWidth:POSITIVE:=18;  -- Output-Data BitWidth
(7)      CONSTANT cDataOutFract:POSITIVE:=16;  -- No of Output-Data fract Bits
(8)      CONSTANT cCoefWidth:POSITIVE:=18;     -- Coefficient's BitWidth
(9)      CONSTANT cCoefFract:POSITIVE:=18;     -- No of Coef's fractional Bits
(10)     SUBTYPE  t_DataIn   IS std_logic_vector(cDataInWidth-1  DOWNTO 0);
(11)     SUBTYPE  t_coef     IS std_logic_vector(cCoefWidth-1 DOWNTO 0);
(12)     TYPE     t_CoefVector IS ARRAY(NATURAL RANGE <>) OF t_coef;
(13)     CONSTANT nCoefs:POSITIVE:=33;
(14)     CONSTANT CoefVector: t_CoefVector(1 TO nCoefs);
(15)   END PACKAGE pk_filter;

(16)   USE ieee.std_logic_signed."*";
(17)   TYPE t_product:std_logic_vector(cDataInWidth+cCoefWidth-1 DOWNTO 0)
(18)   SIGNAL product:t_product;
(19)   SIGNAL DataOut:t_DataOut;
(20)   CONSTANT iPl: INTEGER := cDataInFract + cCoefFract - cDataOutFract;
(21)   CONSTANT iPh: INTEGER := iPl + cDataOutWidth - 1;
(22)   DataOut <= product(iPh DOWNTO iPl);
```

5.8 Mixing *INTEGER* and *std_logic_vector* Data Types

5.8.1 The Data Types *INTEGER, NATURAL, POSITIVE*

After synthesis any integer is a bit-vector. In this subchapter we illustrate bit-to-integer and integer-to-bit conversions.

VHDL requires `INTEGER` types to span a data range of at least –2 147 483 647 to +2 147 483 647. It is required that `INTEGER'LEFT=-INTEGER'RIGHT`. Furthermore there are the two predefined subtypes:

```
SUBTYPE NATURAL  IS INTEGER RANGE 0 TO INTEGER'HIGH;
SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGH;
```

The *SUBTYPE* declaration passed the properties of *INTEGER* on to *NATURAL* and *POSITIVE*. Example: When operators such as "+", "-" or "*" are declared for *INTEGER* types, they are automatically declared for its subtypes also.

INTEGER is a 4 or 8 byte signed bit-vector, depending on the compiler. For example the following statements might synthesize to a 32- or 64-bit signal `i` and a 3-bit signal `j`:

```
SIGNAL i:INTEGER;
SIGNAL j:INTEGER RANGE 0 TO 5;
```

If it is not sure if *INTEGER* data range is wide enough a bit-vector declaration should be used:

```
USE ieee.std_logic_signed."+";
...
SIGNAL a, b, y:std_logic_vector(127 DOWNTO 0);
...
Y <= a + b;
```

5.8.2 Synthesizable std_logic_vector – to – INTEGER Conversion

Both packes *std_logic_signed* and *std_logic_unsigned* within library *ieee* declare the following function, interpreting the first bit as sign bit or not, respectively:

```
function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;
```

Concatenation of a leading '0' bit will always deliver an unsigned interpretation. Example:

```
IntegValue <= CONV_INTEGER('0' & StdLogicVector);
```

Package *std_logic_arith* within library *ieee* declares function

```
function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER) return STD_LOGIC_VECTOR;
```

A vector's number of elements can be obtained with attribute **'LENGTH**, e.g.: **vector'LENGTH**.

In the following example we convert signal *int_in* → *bits_io* → *int_out* → *bits_out*, where the last conversion to *bits_out* serves for comparison with *bits_io*.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.CONV_INTEGER;
USE ieee.std_logic_arith.CONV_STD_LOGIC_VECTOR;
ENTITY conv_int_bitvec IS
END conv_int_bitvec;

ARCHITECTURE rtl_conv_int_bitvec OF conv_int_bitvec IS
  SIGNAL bits_io,bits_out:std_logic_vector(7 DOWNTO 0);
  SIGNAL int_in,int_out:INTEGER RANGE -128 TO 127;
BEGIN
  -- conversion integer <-> bitvector
  int_in <= 5, -5 AFTER 1 ns, 100 AFTER 2 ns, -100 AFTER 3 ns;
  bits_io <= CONV_STD_LOGIC_VECTOR(int_in,bits_io'LENGTH);
  int_out <= CONV_INTEGER(bits_io);
  bits_out <= CONV_STD_LOGIC_VECTOR(int_out,bits_out'LENGTH);
END rtl_conv_int_bitvec;
```

5.8.3 Synthesizable Multiplication Using Operator '*':

Both packes *std_logic_signed* and *std_logic_unsigned* within library *ieee* declare the operator '*', interpreting the first bit as sign bit or not, respectively:

```
function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
```

The length of the returned product vector must be the sum of the lengths of the operand vectors.

5.8.4 Synthesizable Addition Using Operator '+'

Both packes *std_logic_signed* and *std_logic_unsigned* within library *ieee* declare several overloadings of operator '+', interpreting the first bit as sign bit or not, respectively.

If both operands are of type *std_logic_vector* the length of the returned sum vector must be the length of the longer operand vector:

```
function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
```

If there is only one *std_logic_vector* input data type, then the length of the returned sum vector must be the length of the input *std_logic_vector*:

```
function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
```

5.8.5 Synthesizable Subtraction Using Operator '-'

Operator '-' is declared much the same as operator '+':

Both packes *std_logic_signed* and *std_logic_unsigned* within library *ieee* declare several overloadings of operator '-', interpreting the first bit as sign bit or not, respectively.

If both operands are of type *std_logic_vector* the length of the returned sum vector must be the length of the longer operand vector.

```
function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
```

If there is only one *std_logic_vector* input data type, then the length of the returned diffence vector must be the length of the input *std_logic_vector*:

```
function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
function "-"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
```

5.8.6 Synthesizable Comparisons

Both packes *std_logic_signed* and *std_logic_unsigned* within library *ieee* declare the several overloadings of comparison operators, interpreting the first bit as sign bit or not, respectively. Declared omparisions are '<', '<=', '>', '>=', '=', '/=' standing for less than, less than or equal, greater than, greater than or equal, equal and unequal, respectively. In the packages mentioned above any of that operators has the following three overloadings as as '<':

```
function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
```

6 Conclusions

This tutorial introduces VHDL and applies it to the example of a DC/DC buck converter.

- VHDL is a concurrent, event-driven modeling language.

Fundamental design units are entity, architecture and configuration, whereas the
- Entity corresponds to a symbol, the
- Architecture corresponds to a schematic and the
- Configuration defines entity – architecture combinations, either as
  + configuration declaration, which is a design unit, or as
  + configuration specification, which is a statement in the architecture.
- Packages may have a package body and are useful to declare data types and functions.

Code can be written
- Concurrent, which is the default,
- Sequential, which is within processes and subprograms, and
- Structural, which defines a hierarchy from modules and submodules.

FSM design
The feedback loop of a finite state machine (FSM) consists of combinational nextstate logic and state memory.
- The state memory has to be written as process.
- The nextstate logic must not contain memory and can be written
  + concurrent statements or as
  + process that does not generate memory.
- Furthermore, nextstate logic and state memory can be written as a single process.

Processes run top-down within a single simulation delta (Δ), when an event happens in their sensitivity list. To describe combinational logic without memory with a process,
- All its input signals must be listed in the sensitivity list,
- All its output signal must be driven in any possible situation.

Data:

We have 3 kinds of data objects:
- Signals, which are always assigned with a delay, at least one Δ,
- Variables, that are assigned immediately and exist within sequential environments only,
- Constants, which can be seen as variables during compile time and constants afterwards.

A data object may have any data type. There are 4 scalar data types:
- Real, which are floating point numbers of either 4 or 8 bytes, depending on the compiler,
- Integer, which are integral numbers of either 4 or 8 bytes, depending on the compiler,
- Physical, which are integers with a dimension, only predefined type is time,
- Enumerated, which is e.g. used to build state-value systems such as Boolean.

Furthermore, there are 2 composed data types:
- Array, whose elements are addressed with integer numbers, and
- Record, whose fields are addressed with field names (corresp. to structs in Matlab and C).

7 References

[1] *1076 IEEE Standard VHDL Language Reference Manual*, Revision of IEEE Std. 1076, 2002 Edition.
[2] *OVI Verilog HDL Language Reference Manual, version 1.0*, Open Verilog International, 1991.
[3] Available: http://www.systemc.org/home/
[4] Simulink HDL Coder: Generate HDL code from Simulink models and MATLAB code, available: http://www.mathworks.de/products/slhdlcoder/.
[5] Available: http://model.com/
[6] Altera Corporation, available: URL: www.altera.com
[7] M. Schubert, "VHDL Course", *Electronic Design Automation Course*, Regensburg University of Applied Sciences, available: http://homepages.fh-regensburg.de/~scm39115/ → Offered Education → Courses and Laboratories → RED → VHDL
[8] Lehmann, Wunder, Selz, Schaltungsdesign mit VHDL, Franzis' Verlag, Poing 1994.
[9] M. Schubert, Script "Systemkonzepte", Regensburg University of Applied Sciences, available: http://homepages.fh-regensburg.de/~scm39115/ → Offered Education → Courses and Laboratories → SK
[10] M. Schubert, "FSM Design for DSP Using Fixed-Point Numbers", *Electronic Design Automation Course*, Regensburg University of Applied Sciences, available: http://homepages.fh-regensburg.de/~scm39115/ → Offered Education → Courses and Laboratories → RED
[11] M. Schubert, "FSM Design for DSP Using Matlab", *Electronic Design Automation Course*, Regensburg University of Applied Sciences, available: http://homepages.fh-regensburg.de/~scm39115/ → Offered Education → Courses and Laboratories → RED
[12] Terasic DE1-SoC Board, available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836.
[13] Available: http://en.wikipedia.org/wiki/Reinventing_the_wheel
[14] Keating, Michael; Bricaud, Pierre, "Reuse methodology manual for System-on-a-Chip Designs", Kluwer Academic Publishers, 1999, ISBN 0-7923-8175-0.
[15] Regensburg University of Applied Sciences, internal network drive k:\Sb\EDA\Altera\Software.