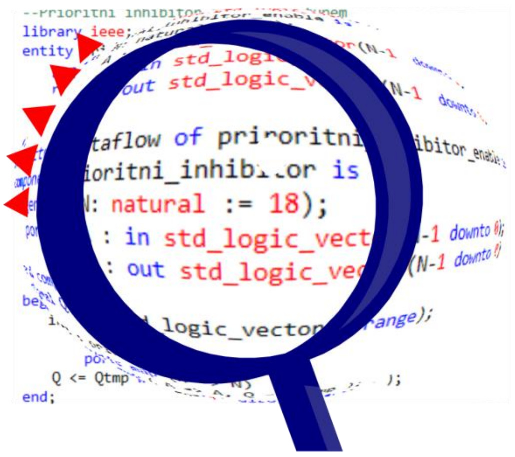


Circuit Design with VHDL Modeling Styles "dataflow" and "structural"

Richard Šusta

Department of Control Eng.
CTU-FEE in Prague

Version beta 1.0 - October 31th, 2019

Content

1 Introduction

Origin of the text

In 2013, I had created "VHDL by Examples" that I upgraded in the summer of 2019, according to my teaching experience. I expanded parts more difficult for students and added new examples.

Overview

VHDL means **V**ery High-Speed Integrated Circuit **H**ardware **D**escription **L**anguage. Note that its name does not include the word "programming" but "description". Sometimes it also expresses program code, especially when creating simulation tests, but VHDL primarily targets physical circuit synthesis. Good designers in it "do not pound" mechanically commands, but concentrate on circuit structures.

VHDL contains a wide range of constructions, but only a part of them is suitable for physical realization of circuits:

1. Some statements deal with physical circuit synthesis,
2. other can also describe the interconnection of individual modules,
3. and simulations can test designs by their responses to generated inputs.

The textbook focuses on the physical synthesis by "dataflow" modeling style that forms the basis of VHDL, and the "structural" modeling style, which is an integral part of larger designs.

- **dataflow** modeling style assembles circuits by concurrent commands that describe the flow of data between inputs and outputs. It can be directly synthesized, and therefore, it is rightly called the basic style. Even circuits created by other styles contain dataflow parts so that we primarily deal with it.
- **behavioral** modeling style describes circuits by sequential statements that are close to classical programming. It makes design easier, and many teaching materials are dedicated to him.
  However, we should note here that behavioral style cannot be directly synthesized. The compiler must first flip the entire code into concurrent commands before building the circuit.
  If users do not understand dataflow, they can hardly imagine what they create. The behavioral style simplifies designs, but we must use it wisely; otherwise, we obtain junk circuits. We explained it in lectures.
- **structural** modeling style specifies the interconnections of modules created by dataflow or behavioral styles. It can describe large designs better than the complex jumble of interconnections in giant symbolic schemes. For smaller circuits, where the mutual relations are apparent, the schemes are still used in professional practice. In Quartus, we draw them in **B**lock **D**iagram/Schematic **F**ile (files *.bdf).

The description by the dataflow modeling best suits for combinational circuits where logical operations can express the dependences of outputs on inputs. Theoretically, it also allows writing sequential circuits, but there is no longer an advantage, so the behavioral style is much better for them. Therefore, we focus only on combinational circuits, but they also form part of the sequential circuit designs that we keep for the lectures. Moreover, we do not avoid dataflow parts because some commands exist only here, such as creating circuit instances, the topic of the whole chapter 6.

Prerequisites

We suppose that readers:

- know the entire prerequisite of APOLOS, which describes the basics of logic circuits and integer number formats. You can find it at http://dcenet.felk.cvut.cz/edu/fpga/guides.aspx .
- have installed a free version of the **Altera Quartus II** development environment that offers the same functionality as the commercial version. It compiles only slightly slower because it uses on-

ly one processor core and does not allow partial compilations, which does not bother our smaller educational designs because their translation slows down a few seconds. They find the necessary installation files at http://dcenet.felk.cvut.cz/edu/fpga/install_en.aspx.

- utilize **Quartus version 13.0.1.232**, the last version supported FPGA circuits family Cyclon II, which we have in development boards DE2 (http://dcenet.felk.cvut.cz/edu/fpga/de2_en.aspx) used in practical exercises. There is no point in installing a higher version of Quartus.
- can create a Quartus II project for the DE2 board, including "Pin Assignments" of the DE2 board from the "DE2_pin_assignments.csv" file. They also find the guide at: http://dcenet.felk.cvut.cz/edu/fpga/guides.aspx.

Commercial editor **Sigasi** (http://www.sigasi.com) also facilitates work by highlights errors when writing VHDL code. Its free version offers everything for smaller files and switches to limited options for larger ones, but still detects errors, which Quartus can't do.

Quartus and Sigasi VHDL editors do not lock access to opened files. They read all their content into memory so that an external application can overwrite a file that is currently open. The editors automatically detect a change and offer an update. We can use multiple editors in parallel for the same file.

Sigasi switches to the full version in the FEL building on Charles Square if you enter our license server. It can be accessed exclusively from the local network felk.cvut.cz — conditions of our Sigasi school license.

@Examples

The text contains 20 solved examples, denoted by roman numerals @Example I. to @Example XX., that gradually introduce into VHDL. Readers can test their understanding by 9 tasks that they find in the inserted chapters *** Practice 1 to *** Practice 9. After solving them, they can compare their codes with the results presented in Chapter 8 - Appendix A: Solutions of Practices.

Why VHDL?

In the US, the Verilog HDL language is predominantly used. It has a C-like syntax. In Europe, VHDL is preferred. It remotely resembles PASCAL, a once-widely used language already isolated. More precisely, VHDL borrowed its syntax from the ADA language for military systems, and PASCAL did the same.

When we were deciding in 2009 whether to teach VHDL or Verilog, professional designers who have been using both languages for years have tended to VHDL. Although it is a verbose and more strict-type language than strongly-typed programming languages like Java or C ++, and its limits that complicate beginners' lives, it significantly increases the chance that the assembled circuit works to our liking.

Verilog was created three years before VHDL as a programming language for circuit simulations. Over time, physical synthesis was added to it. It offers partial similarity to C, including wider type conversions. Both, however, are an excellent trap for beginners, because the techniques learned from C programs are not always suitable for circuits. Professional designers say that people who first learned VHDL have better mastered the correct style of synthesis. Moreover, they quickly retrained to Verilog if they needed it later, but the opposite is not the case.

Syntax highlights and typos

I apologize that color syntax highlighting in VHDL codes is not uniform. Unfortunately, Quartus copies everything as plain text. I added highlights manually and with the aid of various external environments. I postponed unification to the future corrections of the text.

2 VHDL syntax basics and concurrent assignment statement <=

In the textbook, we use VHDL‘93 only**.** Why? Quartus also knows VHDL 2008, which introduced some more advanced commands. However, ModelSim, the circuit simulation application, reliably interprets the entire version 93, but it only allows fragments from version 2008, because its constructs are difficult to emulate. Because of this, we stay with VHDL‘93.

2.1 Syntax basics and identifiers

Comments in VHDL begin with a pair of dashes "--" and end with a line end. There is no way to comment on whole blocks, you have to mark each line separately, but Quartus II can insert comment dashes at the beginning of all selected lines and remove them again.
Don't use diacritics or national alphabet letters, even in comments! Compilers sometimes take them, sometimes not. They reliably understand only pure ASCII.

Identifiers begin with a lowercase or uppercase letters A-Z. The first letter may be followed by lowercase and uppercase letters A-Z and numbers 0-9. The length of the identifiers is not limited.

Underscore characters _ can also be used, but may not be at the beginning or end of an identifier, and no two underscores are allowed in succession.

<u>Examples of allowed names</u>: A1, A_1, a12_b7_Z

<u>a not allowed</u>: A1_ (underscore at the end), A__1 (two underscores in succession), end (keyword),
A$B (not allowed character $)

Note" VHDL‘93 also offers extended names to be enclosed by \ \ characters, between them we can use prohibited characters. For example, extended names are \123.45\, \ END \, or \ A$B\. Using them often causes problems, so it is better to avoid them. They are intended only for special operations and automatically generated codes.

VHDL is case insensitive. However, it is good practice to write identifiers and keywords in the same way. If you type the name of the input signal as "CLOCK_50", "clock_50", "Clock_50" and "ClocK_50", the compiler evaluates everything as the same identifier, but we keep the code more transparent if we write the signal the same way, e.g., **CLOCK_50 or Clock_50**.
Note Some development environments, such as Sigasi, issue a warning if the rule is not followed.

The identifier must be unique and not a keyword, and there are many. VHDL keywords are:

```
abs, access, after, alias, all, and, architecture, array, assert, attribute,
begin, block, body, buffer, bus, case, component, configuration, constant,
disconnect, downto, else, elsif, end, entity, exit, file, for, function,
generate, generic, group, guarded,
if, impure, in, inertial, inout, is, label, library, linkage, literal, loop, map, mod,
nand, new, next, nor, not, null, of, on, open, or, others, out,
package, port, postponed, procedure, process, pure,
range, record, register, reject, rem, report, return, rol, ror,
select, severity, shared, signal, sla, sll, sra, srl, subtype, then, to, transport, type,
unaffected, units, until, use, variable, wait, when, while, with, xnor, xor
```

Used libraries can add other names. In LSP subject, we work with two the most frequent:
ieee.std_logic_1164 and ieee.numeric_std.

2.2 Data type std_logic

When designing circuits, we do not suffice with logical '0' and '1'. The resulting circuit may have a high impedance at the output, or we want to specify that we do not care about the value, and so on.

Type **std_logic** is enumerated with values: '1', '0', 'X', 'Z','U', '-', 'L', 'H', 'W' of Multi Value Logic MVL-9.

Forcing 1	**´1´**	Logic 1, identical to the power supply **Vcc** (Voltage at Common Collector) in the symbolic diagrams created in Quartus
Forcing 0	**´0´**	Logic 0, identical to **GND** (Ground) in the symbolic diagrams created in Quartus
Unutilized	**´U´**	Circuit simulation reports that output value is not yet known as it has not been initialized yet.
Don't Care	**´-´**	Indicates an arbitrary value that we don't care about it.
Forcing Unknown	**´X´**	Unknown value, the simulation cannot determine it.
High Impedance	**´Z´**	Disconnected the output, also known as hi-Z or tri-stated output.
Weak 1, Weak 0, Weak unknown	***´H´,´L´, ´W´***	*Outputs of particular circuits, e.g., with an open collector. Cyclone FPGAs do not include them, so we cannot use them.*

Table 1 - Type std_logic

Data types derived from std_logic are preferably selected in VHDL synthesis for signals bound to inputs and outputs because VHDL compilers can easily realize them. Tables define logical operations with them, see chapter 9, page 72.

VHDL also contains data types"boolean" and "bit" — the both also belong to enumerated types.

- *Type boolean with values TRUE a FALSE is intended only for conditions, see later.*
- *Type bit with values ´0´ and ´1´ is utilized only by simulations or by computational parts. In synthesis, it has disadvantages because VHDL compilers cannot sometimes detect circuit errors as a shortcut of signal sources.*

Warning — data types std_logic and bit are unconvertible to each other by any typecast. We can convert them only by conditional assignment, see chapter 4.2 on page 30.

2.3 Signal - wire and concurrent assignment <=

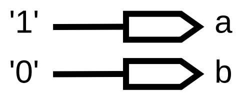
The signal definition creates an element equivalent to a conductor, and it is **not a variable"**. In Verilog, its analogy is called directly "wire". The assignments to the signal always require notation <= called "concurrent assignment". All such assignments are made simultaneously in parallel. If we have definitions of signals somewhere

```
signal a, b : std_logic;
```

and we later connect them to '1' a '0' by commands:

```
a <= '1'; b <= '0';   -- '1' a '0' values of enumerated types std_logic
```

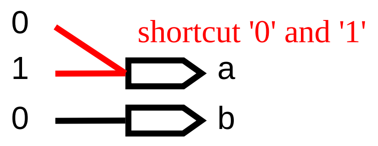
then, the both assignments are performed at one instance of time regardless of their order in a code. No time priority exists in the circuit that we have created by the statements. Both wires operate concurrently!



Furthermore, there is a strict rule that signal **elements can only be assigned once**! If we write::

```
a <= '1';
b <= '0';
a <= '0';   -- error
```

then we create the circuit



The Quartus shows the error message "Error (10028): Can't resolve multiple constant drivers". In our code, we have connected logical '0', which is usually the ground, with logical '1', which often identifies with the supply voltage. Oops, we have designed a sparking short between power and ground :-(

Note: If you see multiple assignments in any VHDL code, they are in particular sections (such as process, function, and procedure) written in the VHDL behavioral modeling style, which uses a higher level of abstraction. The compiler first converts it to a dataflow modeling style by introducing auxiliary signals to satisfy the strict limit of a single assignment. Only then it can build the circuit.

2.4 Operations with types std_logic and boolean

VHDL offers all general operators. Enumerated type std_logic allows the following operators.

Priority		Data type of result	Operator
Highest	⬇	the same as operands [(1)]	**not**
		boolean (true, false)	**= /=**
Least		the same as operands [(1)]	**and or nand nor xor xnor**

[(1)] In ieee.std_logic_1164, type std_logic has defined operators listed above for **std_logic** returning std_logic result, and also for type boolean, for which they return boolean.

[(2)] Note that logical operations do not have any priority to each other!

Table 2 - Operator with std_logic

We design logical majority function that has three inputs A, B a C, and its output Y is in logic ´1´ only if at least two its inputs are in logic ´1´.

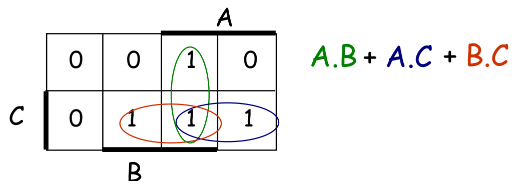
We design the majority function either from Karnaugh's map or by the following logical reasoning:

From three inputs, we can create three pairs A-B, A-C, and B-C. If any of them has both terms in '1', then its result Y is always '1'. So we connect members of pairs by AND operator giving '1' only if both inputs are in '1'.

We bound three ANDs by OR logical operators, which result is in ´1´ when at least one AND gives '1':

Y = (A and B) or (A and C) or (B and C)

We can create the circuit in Quartusu with the aid of graphic editor as *.bdf (**B**lock **D**iagram/Schematic **F**ile) file majorita.bdf, but we now prefer writing in VHDL.



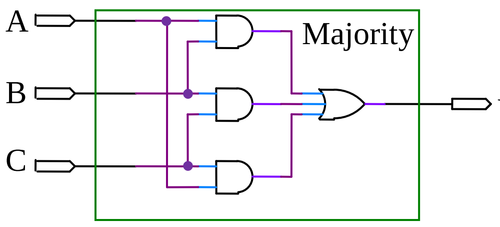


Figure 1 - 3-input majority

Suppose that we have somewhere defined signals **a**, **b**, **c** a **y**, then VHDL equation of the majority is:

```
y <= (a AND b) OR (a AND c) OR (b AND c);          --Eq2.4-1
```

We specify the order of logical operations by round parenthesis.

Note: Both C and Java languages introduced && operator priority over || for convenience although Boolean algebra doesn't know it. In VHDL, you must specify the order of logical operator execution.

If we omit the parentheses and write the previous equation as y <= a AND b OR a AND c OR b AND c; --Eq2.4-2

then the compiler reports *Error (10500): VHDL syntax error near text "OR"; expecting ";"*, because it uncompromisingly requires the separation of different types of logical operations with parentheses.

We can use the same binary operand several times without inserting parentheses, for example, "a XOR b XOR c", but adding a different operand, such as OR, already requires parentheses.

Older VHDL compilers did not check our equation (Eq2.4-2) without parentheses and supposed its evaluation in the order of operands which gave the very different circuit>

y <= (((((a AND b) OR a) AND c) OR b) AND c); --Eq2.4-3

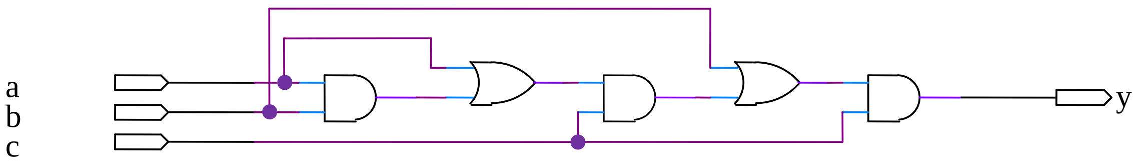


Figure 2 - Schematic corresponding to `y <= a AND b OR a AND c OR b AND c;`

For that reason, newer versions of Quartus contain strict checks of parentheses in logical equations, including version 13 of Quartus that we use in the practical exercises.

Mousetraps of std_logic type

If somebody had written equation Eq2.4-1 by the following breakneck way:

y <= (a='1' AND b='1') OR (a='1' AND c='1') OR (b='1' AND c='1'); --Eq2.4-4

he saw mysterious report: *Error (10327): VHDL error can't determine definition of operator ""="" -- found 0 possible definitions.*

However, if we introduce a new signal, in the VHDL code section reserved for definitions:

signal x : boolean;

also, we modify the equation Eq2.4-4 to

x <= (a='1' AND b='1') OR (a='1' AND c='1') OR (b='1' AND c='1'); --Eq2.4-5

the compiler accepts the code.

We can also compare std_logic types with the equals = and inequalities /= operators, but they give a boolean enumeration result consisting only of TRUE or FALSE values that are unconvertable by typecasts to a 9-value std_logic type.

Note: We can convert them by conditional assignment introduced in chapter 4.2 **Chyba! Nenalezen droj odkazů.** *on page 30.*

In the majority equation, superfluous = operators do not make sense. After all, everything can be expressed by std_logic!

Result boolean	Equivalent giving result std_logic	
a='1'	**a**	
a='0'	NOT a	
a \= b	**a** XOR **b**	XOR is non-equivalence ≠
a=b	**a** XNOR **b**	negated XOR is equivalence **=**

Beware, the following equation:

y <= (a='1' AND b) OR (a AND c) OR (b AND c); --Eq2.4-6

is totally wrong a generates another mysterious mistake *Error (10500): VHDL syntax error near text ";"; expecting "end", or "(", or an identifier, or a concurrent statement.*

The first comparison a='1' gives boolean result that we cannot combine with std_logic types. The error comes up somewhere at a later stage of translation, and so a bizarre message appears.

Quartus does not translate code to assembler instructions as compilers of programming languages, but it creates a meta-circuit diagram (RTL Map) at the beginning and further optimizes it. If he fails to create RTL Map, it sometimes shows an ambiguous report. A mistake may be anywhere before the hitch.

On the other hand, let's be pleased that VHDL depends exceptionally on types because their precise use discovers many errors. We cannot debug an assembled circuit, step through wires, and read wire values. We only see or measure signals that we have connected to outputs. Others remain hidden, and therefore careful design helps us, as VHDL outcome is more difficult to debug than a traditional program.

Summary: The std_logic type is implemented as an enumerated type whose values are listed on page 7. If we have the definition:

```
        signal t, x, y, z : std_logic;
```

then, we can assign to these wires only std_logic types, i.e., either results of logical operations with std_logic types or constants of its enumerated set, for instance:

```
        t<=´Z´; x<=´1´;  y<=´0´; z <= x AND y;
```

The constants ´1´ and ´0´ are in quotation marks, but these are not characters (VHDL type character), but values of enumerated std_logic type defined in ieee.std_logic_1164, in a style analogous to enum types in Java or C #, simply (*for the exact definition, see chapter* 9 *on page 72*):

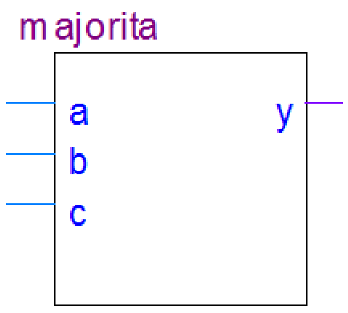
```
        type std_logic is ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

2.5 @Example I. - VHDL code of majority function

The VHDL code of the majority function is not complicated. We start from a universal template that we explain in the next section. It suits for other designs. The highlights show the parts that we have added:

VHDL code majorita.vhd | **Schematic symbol**

```
-- Majority function 2/3
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity majorita is
        port (   a, b, c : in std_logic;
                 y : out std_logic );
end;
architecture dataflow of majorita is
begin
        y <= (a AND b) OR (a AND c) OR (b AND c);
end;
```



From the VHDL code, Quartus can build a schematic symbol that we can use in the schematic editor. Quartus also shows us its internal schema created during the introductory pass (RTL Map Viewer):

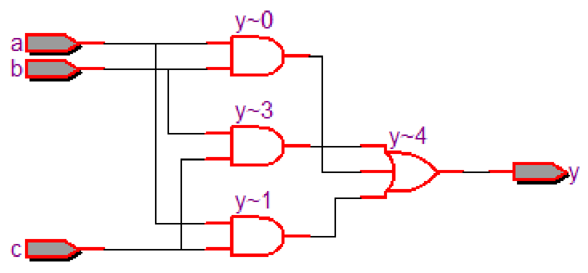


Figure 3 - Internal schema created by Quartus compiler

The auxiliary identifiers generated by the compiler contain ~. We must not use this character in our identifiers. Quartus had reserved it for himself.

In appendix C on page 74, we find out a complete guide on how to create a VHDL file, compile it, generate its schematic symbol, view the internal RTL logic diagram, and test the design by simulation.

2.6 VHDL template

Here, we explain the VHDL code template that we used to write the majority because we can apply it to other circuits. Once we understand its structure, we can quickly create additional VHDL codes by copying the template and editing the marked locations.

We add references to lines L01 to L15 for explanation purposes, and page specifies link to descriptions.

VHDL code	Line	Page	
-- comment	L01	12	*optional, but very appropriate comment*
library ieee;	L02	12	*library activation*
use ieee.std_logic_1164.all;	L03		*types derived from std_logic*
use ieee.numeric_std.all;	L04		*signed and unsigned numeric types*
entity our_name is	L05	12	*entity beginning - external appearance of curcuit*
port (	L06	13	*I/O definitions*
-- inputs/outputs	L07		*here, we write inputs and outputs*
);	L08		
end;	L09		*end of entity*
architecture dataflow of our_name is	L10	14	*circuit implementation description*
--definitions	L11		*here, we insert definitions of auxiliary elements*
begin	L12		
--implementation	L13		*here, we describe required function*
end;	L14		

Trinity naming in VHDL file - our name

For the file, we must choose a valid VHDL identifier unique within our design (i.e., Quartus project). We must write the chosen identifier in three places!

1. When saving a file, we use it for **filename**, to which we add only the * .vhd extension
2. We write it as the name of the **entity**,
3. and type it in the **architecture** header between the of and is keywords.

For example, if we select a "majorita", then we save the file as majorita.vhd, and correct the template:

```
entity majorita is   -- L05
```

and

```
architecture dataflow of majorita is        -- L11
```

We may never break the triple-naming rule. Otherwise, the compiler either does not find our design, or it fails to incorporate it into our created circuit. Such a misdemeanor is a common mistake, so we always check that we have fulfilled the triple name.

We should save the file in the directory of our Quartus project and always keep the entire path Linux file system compatible, i.e., no spaces in the path, diacritics, or Windows particular extensions!

Note: Quartus is a native Linux application that runs under Cygwin on Windows.

In our educational laboratories, we must never open Quartus projects on network drives. Their path contains $ character in full filename obtained from the OS. Do not be misled by the fact that we do not see $ in Windows Explorer, which often shows only placeholder, not the real full path.

USB drives are not suitable either. During translation, Quartus creates a flood of auxiliary files at the project location. If it starts writing to USB, the translation extends from tens of seconds to tens of minutes.

L01 of template - commentary

Although this is only an optional possibility, it is useful to describe what the code does, so we do not forget about our intentions once upon a future☺.

L02 and L04 of template - libraries

Libraries are an advantage of VHDL over Verilog. Under libraries, we can imagine remote analogies of commands using C# or import in Java. They are not equivalents of #include C language that embeds whole files. In VHDL, definitions only are imported, but the final effect is similar.

Command "library" (line **L02** of the template) activated library, here standard ieee specifies "Institute of Electrical and Electronics Engineers (IEEE)". *Note: IEEE has established pronunciation "I triple E".*

On line **L03**, we extract from it the package declaring the standard logic "std_logic_1164". We take everything (all) from it. Likewise, we also insert numeric_std. *Note: The "all" option is the most common, but we can select only one element from a library when we solve name-conflict in more complex projects.*

- **ieee.std_logic_1164** - contains definitions and operations with std_logic-based types.
- **ieee.numeric_std** - contains recommended definitions of operations with integer, signed, and unsigned types. The library replaces the older ieee.std_logic_arith, which had many different versions written by individual companies, making it difficult to port the codes. As a result, the IEEE Committee created the international standard ieee.numeric_std. The new codes use numeric_std. Our initial VHDL codes do not contain number operations yet, we use them later, but the library is still inserted to keep the template universal. Quartus has all the most common libraries precomplicated, so adding it does not slow down the compilation in any way. (Verified:-)

L05 and L09 - block entity

The circuit design includes a mandatory entity block declaration that begins with **L05** in our template:

```
entity our_name is
```

where entity is a keyword, and "our_name" represents the circuit name we assign, it must match the VHDL file name and architecture. The entity block can contain several definitions (*but it may not have one, it is empty for testbench tests*). Our example has only one definition "port" (**L06**) of inputs and outputs.

On **L09,** block entity ends by keyword end;

The **entity** block determines the external appearance of the circuit, i.e., its inputs and outputs visible to users. It generally behaves quite differently from the classes known from C ++, Java, and C #, but we can give a slight analogy. The name of the entity corresponds to the class name - and from the class, here from the entity, we create instances, which in our case are circuits built according to a prescription described in architecture, with the aid of map command, which is similar to a constructor. We dedicate whole chapter 6 to map.

We can terminate the entity block in abbreviation by end, as in our example, or we can append entity keyword or also the name after end, which is allowed since VHDL'93 (used in the LSP course). Thus, our entity block can have a shape:

entity our_name is -- deklarations end;	entity our_name is -- deklarations end entity;	entity our_name is -- deklarations end our_name;	entity our_name is -- deklarataions end entity our_name;

Table 3 - Entity declaration termination options in VHDL'93 and hogher

Truncation of terminating end is also allowed for some other declarations, e.g., the architecture block on lines **L10** to **L14**, where the same principle can be applied. The choice depends only on the programmer. Shorter blocks can have only end, but for large blocks, we repeat the keyword of the initial declaration of a block, here an entity, or we add a block name or both to know what end refers to.

L6 to L08 - port - inputs and outputs

The port definition can appear only within the entity declaration and at most once. It specifies the inputs and outputs that we can see from outside, a kind of analogy to "public" elements of class.

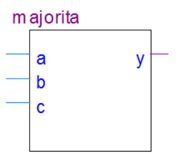
It defines lists of signals (signal-names), their data types (signal-type) and modes (mode), which determine the type of input or output.

Note that a semicolon is missing after the last signal-type -its typing is a frequent syntactic error.

```
entity entity-name is
  port (signal-names : mode signal-type;
        signal-names : mode signal-type;
        ...

        signal-names : mode signal-type);
end entity-name;
```

Our code majorita.vhd defines three inputs a, b, c, and one output y, which we specify:

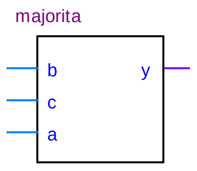
```
entity  majorita is
port (  a, b, c : in std_logic;
               y : out std_logic );
end;
```



In the port definitions, we either put a comma-separated list of signals, or we write each signal can be written on a separate line to allow adding a comment on the line.

The order of definitions in port block is arbitrary, but it also determines the order of inputs and outputs on the generated schematic symbol for use in the symbolic schema. If their order or names are changed, the tag must be regenerated, see appendix C (chapter 10 on page 74) If there was no change in port declarations, the previous symbol remains valid.

```
entity majorita is
 port ( b : in std_logic;
        y : out std_logic;
        c : in std_logic;
        a : in std_logic );
end;
```



Input and output modes:

- We can only read **in**-mode input because an external signal source drives it by its output. Thus, any input cannot be written from our code to avoid a short circuit. Therefore, in-mode is read-only.
- • In contrast, **out**-mode specifies our output to which we can assign a value from inside the circuit but we cannot read its value. Therefore, out-mode is write-only. Moreover, be careful, because it is a wire, i.e., signal that we can assign only once.

If we need to read output value, we can use buffer-mode, but the compiler must reserve a slightly more extensive element for it, with the wire leading back to the inside of the circuit.

The last mode is the type of inout, which allows both reading and writing data. However, it requires a complex bidirectional driver element and is thus used only exceptionally, for example, on buses. On the DE2 board, I2C audio bus has bi-directional audio I/O with inout-mode.

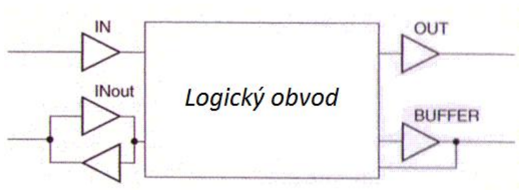


Figure 4 - Modes in circuits

For designs intended for physical circuit synthesis, we prefer only in and out modes that are the easiest for implementations. All professional designers do so.

L10 and L14 of template Architecture

The architecture block describes the operation of the circuit itself. In our code, it begins at **L10** with the keyword architecture followed by the name of the created architecture and the name of the entity for which we write the architecture. Then, the begin end command block follows;

On line **L11**, we could also define internal auxiliary signals, but our code did not need them.

For ending the architecture block with end keyword (**L14**), we can apply similar options as for entity blocks. After the keyword end it is possible to repeat the keyword architecture and also its name dataflow, similar to entity block, see Table 3 on page 13.

*Note:We can consider entity and architecture blocks as parts of an electronic device, such as a digital alarm clock. The **entity** describes elements placed on its box surfaces, such as buttons, buzzer, or display elements. The **architecture** then corresponds to the internal wiring of the alarm clock. The wiring is hidden from the outside observer inside the box.*

If we innovate the alarm, we can only change the controls on its surface, but we left the internal wiring the same — therefore, we only made adjustments to its entity, i.e., to external appearance. Equally, we can remodel the internal wiring (architecture), while maintaining the same controls, or modify both.

We select dataflow as architecture name according to the VHDL style we used. However, it is not a rule. Another common name is synth and circuits with a clock signal usually use rtl (register-transfer level).

The architecture name, here the dataflow, is not a global identifier and belongs only to its entity. In other words, in all entities in other files, we can name their architectures again dataflow.

Note: A mandatory architecture name offers the ability to create several different architectures for a single entity, that is, multiple architecture blocks. We employ such possibilities, for example, when we need one circuit function for its realization and another one to speed up simulations. Using multiple architectures, of course distinguished by different names, is a more advanced topic because we also need to add a VHDL file to the project with the configuration command. In an LSP course, each entity can have a single architecture.

2.7 @Example II. - VHDL code of 4-input XOR

XOR operation is associative and commutative; see https://en.wikipedia.org/wiki/Exclusive_or; therefore, its does not depend on the order of operations. The following equations give identical results:

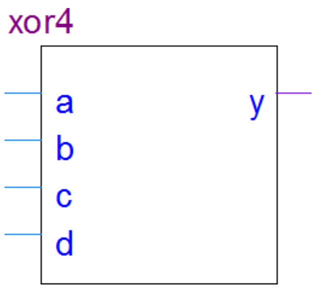
```
y1 <= (a XOR b) XOR (c XOR d);   y2 <= (((a XOR b) XOR c) XOR d);
y <= a XOR b XOR c XOR d;
```

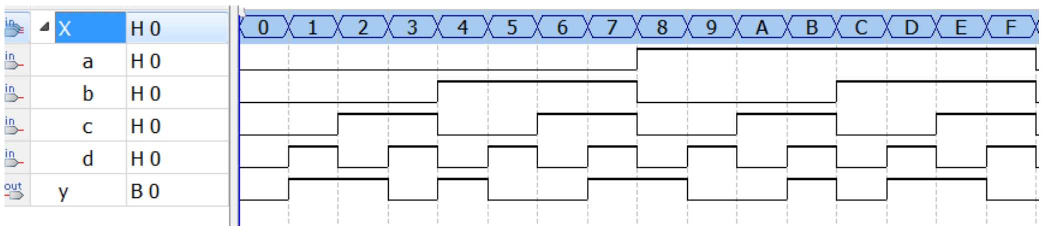
We chose the last simple equations, and we write 4-input xor with the aid of our template:

VHDL code - xor4.vhd

```
-- 4-input XOR
library ieee;   use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity xor4 is
        port (   a, b, c, d : in std_logic;
        y : out std_logic );
end;
architecture dataflow of xor4 is
begin
        y <= a XOR b XOR  c XOR d;
end;
```

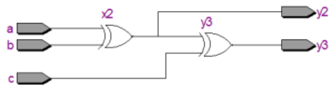
Schematic symbol





If we check our xor4 by simulations, then we see that it gives output '1' only if it has an odd number of inputs in logical '1' ─ in other words, a 4-input XOR tests whether the input has odd parity.

Note: Each multi-input XOR outputs '1' only on its odd number of inputs in '1'. We can prove it by mathematical induction. The presumption holds for 2-input XOR and remains valid after adding another XOR behind it.



2.8 @Example III. - Majorita2 with the indication of 2 inputs in '1'

Task: Add another output to the previous majority circuits that reports that just two inputs are in '1'.

a	**b**	**c**	**y** -majority	**y2 -** 2 in '1'
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	**1**	**1**
1	0	0	0	0
1	0	1	**1**	**1**
1	1	0	**1**	**1**
1	1	1	**1**	0

We can derive the equation of new y2 output by Karnaugh's map, but we get it faster by logical reasoning. Output y2 is the same as y, except for all inputs in '1'. However, we cannot use the already calculated value of y, because it has write-only output mode!

y2 <= y AND NOT (a AND b AND c); -- error --Eq2.7-1

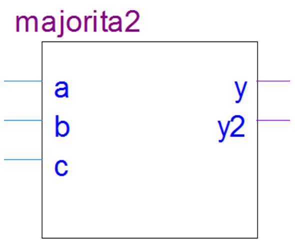
We could change y mode to buffer, but we use a more elegant solution and create an auxiliary signal, which we call tmp. Improved circuit, named perhaps Majority2, implements functions:

tmp <= (a AND b) OR (a AND c) OR (b AND c); --Eq2.7-2

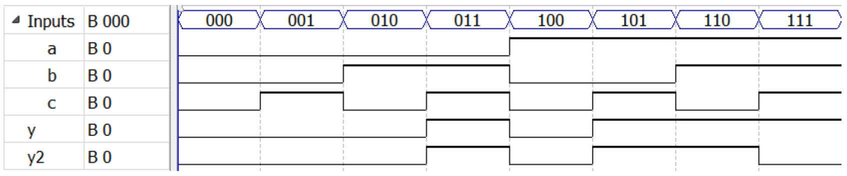
y<= tmp; y2 <= tmp AND NOT (a AND b AND c); --Eq2.7-3

VHDL code majorita2.vhd	Schematic symbol

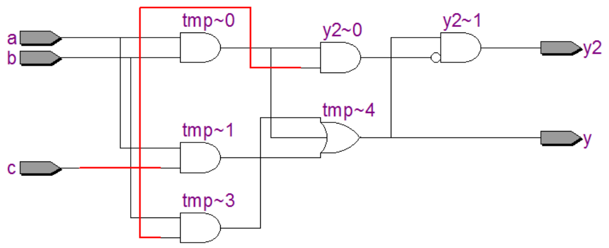
```
--Majority 2/3 plus the indication of 2 true inputs
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity majorita2 is
        port (   a, b, c : in std_logic;
                 y, y2 : out std_logic );
end;
architecture dataflow of majorita2 is
signal tmp : std_logic;
begin
   tmp <= (a AND b) OR (a AND c) OR (b AND c);
   y<= tmp; y2 <= tmp AND NOT (a AND b AND c);
end;
```



The output of simulator:



RTL Map - the internal circuit diagram created by Quartus in the initial compilation steps:



Note that the compiler cleverly used the term (a AND b) in the expression for tmp to realize the 3-input AND. It performed so-called group minimization, which reduces the total number of operations in all functions. We describe it in LSP lectures.

The RTL Map is only the first translation step, and the compiler minimizes it later, but we should always look at RTL View because it describes how Quartus understood our VHDL code.

The signals in combination circuits mean only named wires, so we do not increase the complexity of the circuit by definition of auxiliary signals because the wires do not allocate memory, unlike the variables in the programs.

Advanced note for completeness: If we prescribe that <= is replaced with a register, even auxiliary signals can increase complexity. However, this requires inserting the edge detection of a clock, which is also possible with a dataflow modeling style, but with complications. VHDL behavioral modeling style, which we keep for lectures, is much better for similar operations. Without the edge detection of a clock, all signals remain mere wires with nameplates!

2.9 *** Practice 1: - Majorita123

Extend Majorita2 circuit with two more outputs. The first one, output y3, indicates 3 inputs in '1', and the next output y1 reports that exactly one input is in '1'. When you make the design yourself, you can compare it to the possible solution on page 59. If you write the code a little differently than the solution, but with the same functionality, then you also designed the right circuit.

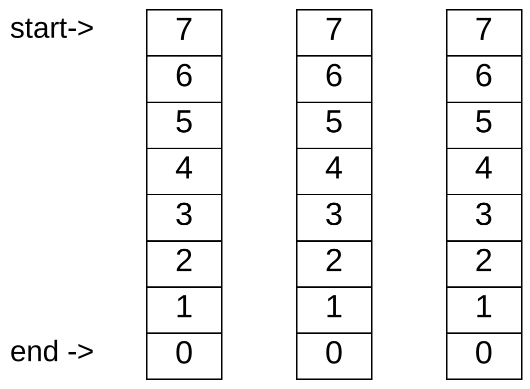
2.10 Data type std_logic_vector

Sometimes, we need to handle wire groups as a single unit. In VHDL, we can use std_logic_vector data type that is a one-dimensional array of std_logic elements. Its explanation is a slightly more difficult topic because there is no longer a reasonable analogy in C programs.

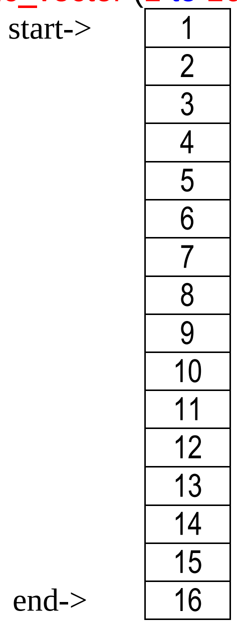
- Type std_logic_vector has no numeric values; it is just a bundle of numbered wires. The count of wires can be from one element upwards.
- Although it is a one-dimensional array whose indexes are non-negative numbers (VHDL data type natural), they can run ascending (to) or descending (downto) and start with any natural value.
- **Descending numbering is preferred**, although this is defined by the longer keyword downto ☺, since the array then begins with the higher element; thus, the ordering of elements corresponds to the binary number arrangement where the bit with the highest weight lies to the left.
- With the std_logic_vector type, we can perform the same operations as with std_logic, see Table 2 on page 8, selected operation is repeated for all std_logic signals bound in std_logic_vector.
- **Moreover, we have no memory cells here!** The std_logic_vector type is not an exact analogy of vectors. When we assign a new value to it, we create only connections! It is not about storing data. If a source value changes, it is immediately transferred to the destination by ordinary wires.

We can best imagine std_logic_vector as a series of wires with plugs that have sequential numbers, but their numbering started from a non-negative value and proceeded in descending or ascending order.

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



```
signal Z: std_logic_vector (1 to 16);
```



We enclose constants of std_logic type by ' ' apostrophes, and in std_logic_vector type, we use quotation marks " ", even if the vector is only one member. The vectors correspond to strings. Generally, the '0' element is of type std_logic, but "0" is of type std_logic_vector (0 downto 0), a vector with one member. We can, of course, define a group of signals with one member as signal xv : std_logic_vector (100 to 100);
The direction does not matter, and the initial index can be any positive number. Even if there is one member, we assign logical constants 0 and 1 with quotation marks, e.g., for xv above as xv<="1";

We write indexes in parentheses, i.e., in round brackets! We can address the only one element of the above xv as xv(100), i.e., according to the index given in its definition range.

If a vector joins several signals, only one of them or a subgroup of them can be selected, i.e., a subvector. For the above definitions, we may use:

Operation	***Method of realization***
B(5) <= A(0);	Signal B(5) is connected by wire with A(0), both are std_logic.
A(1) <= '0';	Signal of std_logic type in vector A at index 1 is connected to '0'.

C <= "01010110";	Signals in C are connected in order of their indexes to listed values:
	C(7)<='0'; C(6)<='1'; C(5)<='0'; C(4)<='1';
	C(3)<='0'; C(2)<='1'; C(1)<='1'; C(0)<='0';

B(7 downto 4) <= A(5 downto 2);	B(7) <= A(5); B(6) <= A(4); B(5) <= A(3); B(4) <= A(2);

If we select a subgroup, we must keep the order of numbering in definition range, i.e., downto or to. We cannot reverse it. If we try to write Z(15 downto 14), the compiler announces error: "*…range direction of object slice must be same as range direction of object…*".

Note. Editor of schematics (.bdf) in Quartus uses shorter notation in Pascal-like style. For example, notation A[2..0], Z[1..7]*, and *A[1] correspond to VHDL: A(2 downto 0), Z(1 to 7) a A(1).*

We can write the assignment of a constant into std_logic_vector with the length dividable by 4 with the aid of hexadecimal notation with X in front of the first parentheses, e.g., X"5F" is equivalent to "01011111".

Logical operations with std_logic_vector types are executed on per element base.

Statement: C <= A AND B; means the same as the following:

C(7) <=A(7) AND B(7); C(6) <=A(6) AND B(6); C(5) <=A(5) AND B(5); C(4) <=A(4) AND B(4);
C(3) <=A(3) AND B(3); C(2) <=A(2) AND B(2); C(1) <=A(1) AND B(1); C(0) <=A(0) AND B(0);

We can present example of how to use the signals defined above. In the figure, the vector was drawn horizontally for spatial reasons and the left side corresponds to the start element, i.e., left-most of the range:

A<="10101100"; B<=X"5F"; C<=A AND B; Z(2 to 9)<=C; Z(14 to 15)<= C(4 downto 3);

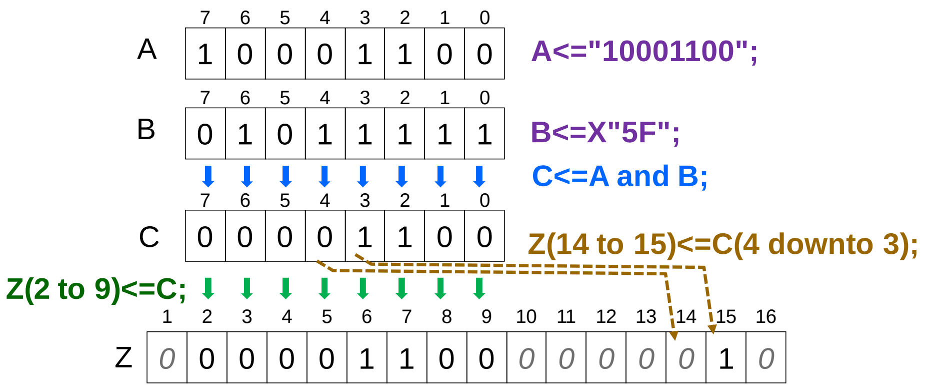


Figure 5 - Operation with std_logic_vector

Operator &

If we have two definitions:

```
signal nv: std_logic_vector (1 downto 0);
signal n1, n0: std_logic;
```

then we can connect n1 and n0 with nv by two ways. First, by members:

```
nv(1)<= n1;  nv(0)<= n0;
```

Second, you can write the above line briefly using & operator to join them into vector.

```
nv <= n1 & n0;
```

Note Operator & represents here an analogy of the concatenation operator **.** *of PHP web language.*

We can create complex joinings with the aid of & operator. If we have somewhere definitions:

```
signal dv, ev: std_logic_vector (5 downto 0);
```

then the next statements:

```
ev <= dv(1 downto 0) & '0' & dv(5 downto 3);
```

create connections:

output	d(1)	d(0)	'0'	dv(5)	dv(4)	dv(3)
connected to	↓	↓	↓	↓	↓	↓
signal	ev(5)	ev(4)	ev(3)	ev(2)	ev(1)	ev(0)

In the previous statement, we can also surround 0 by quotations marks:

```
ev <= dv(1 downto 0) & "0" & dv(5 downto 3);
```

because operator & has many overloaded operations that allow also joining std_logic with std_logic_vector type so we can write either '0' or "0".

3 @Example IV. - Decoder by with...select

Assignment: Design a decoder for a bar indicator.

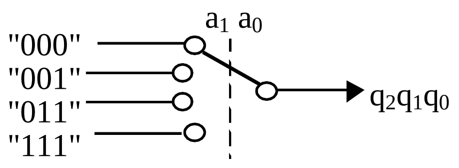
In its three-bit output, there are bits in logical '1'. Their count corresponds to input taken as an unsigned number, as shown in the truth table on the right.

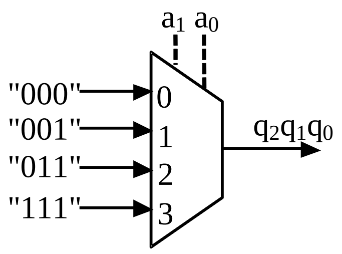
Input a0 is the lower bit, and a1 is the higher bit.

Truth Table of Decoder					
Input			**Output**		
Value	a_1	a_0	Q_2	Q_1	Q_0
0	0	0	0	0	0
1	0	1	0	0	**1**
2	1	0	0	**1**	**1**
3	1	1	**1**	**1**	**1**

Solution: We could write three logic equations, one for each output. Such a procedure is neither universal nor transparent. It does not suit for longer bar indicators, e.g., for a decoder with a four-bit address that needs fifteen output equations. So we find out another, more flexible solutions.

A switch could solve this task. The position of its slider corresponds to its input value, and its contact specifies the required output for the slider position. There is an analogy to the switch in the form of a multiplexer, which only outputs the value from the input selected by the address.





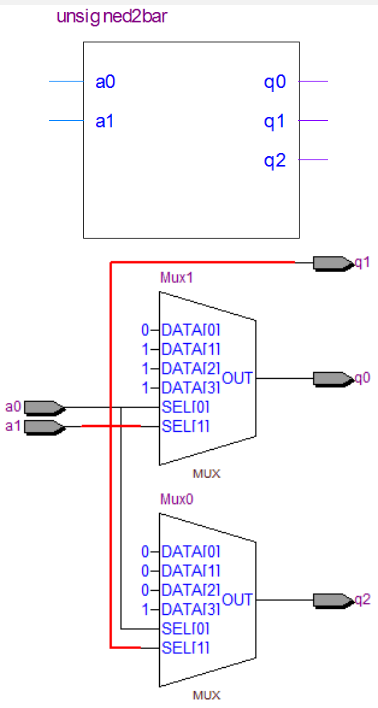
Note: In a circuit, CMOS-based switches can effectively create multiplexers.

In VHDL, we define a multiplexer by with...select statement.

VHDL code unsigned2bar .vhd

```
--Bar indicator / cz  linearni ukazatel
library ieee;   use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity unsigned2bar is
        port (    a0, a1 : in std_logic;
                  q0, q1, q2 : out std_logic );
end;
architecture dataflow of unsigned2bar is
signal av: std_logic_vector(1 downto 0);
signal qv: std_logic_vector(2 downto 0);
begin
        av <= a1 & a0;
        with av select
        qv <=   "000" when "00",
                "001" when "01",
                "011" when "10",
                "111" when "11";
        q2<=qv(2); q1<=qv(1); q0 <= qv(0);
end;
```

Symbol and schema



We see from RTL Map that Quartus has detected that output q1 is in '1' whenever input a1 is '1', and so its direct wire. The other two inputs are connected multiplexers.

The figure shows only the result of the initial VHDL pass when the compiler converts the code into a suitable RTL Map. Then, it minimizes the map according to the capabilities of the target FPGA (Technology Map). The result consists of logical equations with AND and OR, see Table 4.

```
q0<=  a1 AND a0;  q1<= a1;  q2<=a1 OR a0;                    --Eq3-1
```

We could also find out equation Eq3-1 from the logic truth table. However, we have written a multiplexer as a demonstration that in VHDL, it is not so much about the most cost-effective code in terms of minimum commands, but its correct structure. If it is well-chosen and specifies the circuit structure, then Quartus does all necessary minimization itself.

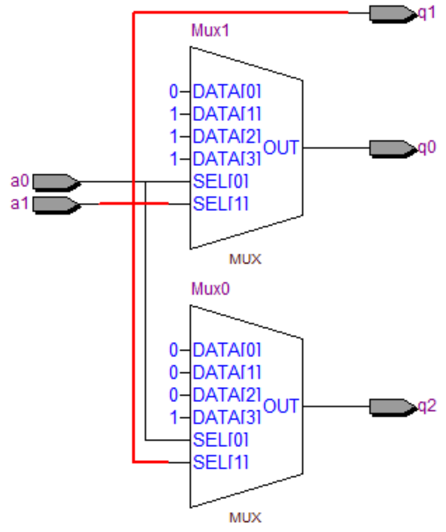
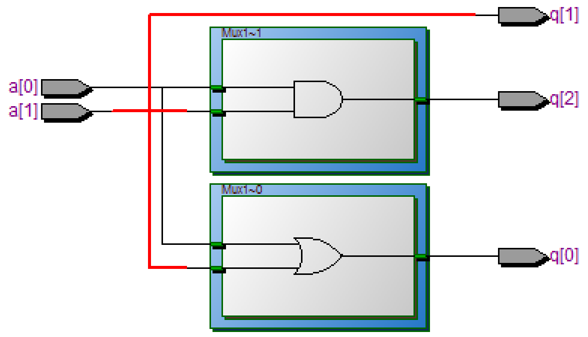
First passes - RTL Map	Minimalization - Technology Map
 Mux1 0 DATA[0] 1 DATA[1] 1 DATA[2] 1 DATA[3] OUT SEL[0] SEL[1] MUX Mux0 0 DATA[0] 0 DATA[1] 0 DATA[2] 1 DATA[3] OUT SEL[0] SEL[1] MUX a0 a1 q1 q0 q2	 *Note Quartus has created std_logic vector groups during minimization .*

Table 4 - bar indicator - first RTL Map and its minimization

<u>Question:</u> **Why do we define av and do not write directly with a & b select?** Well, we didn't write because we didn't want to take too long a frog jump to avoid breaking VHDL study mood. If we omit the definition of av and write the multiplexer directly as:

```
with a1 & a0 select
qv <=  "000" when "00",
       "001" when "01",
       "011" when "10",
       "111" when "11";
```

then the compiler shows: *Error (10327): VHDL error: can't determine definition of operator ""&"" -- found 2 possible definitions.*

Overloaded operator & can process more types. In the original statement, av <= a1 & a0; the compiler could derive the correct type from the left side of the assignment, on which the std_logic_vector signal stood. In a with-select statement for multiplexer, choice addresses are specified by constants that do not have a unique type.

So we have to select the return type manually using a type-help, which consists of a type with apostrophe ' before expression enclosed in parentheses, such as:

```
with std_logic_vector'(a1 & a0) select
qv <=  "000" when "00",
       "001" when "01",
       "011" when "10",
       "111" when "11";
```

Attention, this is not a typecast! We only specify our wish that for many possible types, this one should be chosen. If it is possible, the compiler does so.

Properties of with select

The address values after when must b**e constants,** and we must list all possible addresses - the switch (multi-plexor) must always switch to one input. Otherwise it can't. We may also use the keyword others, which means all values not yet listed. Using it, we could rewrite the statement like this:

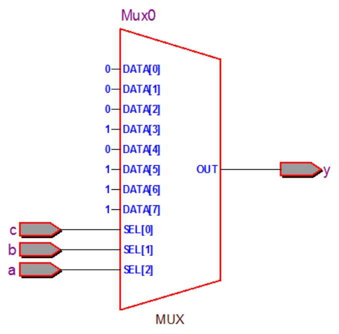
```
with std_logic_vector'(a1 & a0) select
         qv <= "000" when "00",  "001" when "01",  "011" when "10",
                "111" when others;
```

By character | , we can **group several condition.** If we wanted to write the previous majority from page 10 with a multiplexer, we can do it this way:

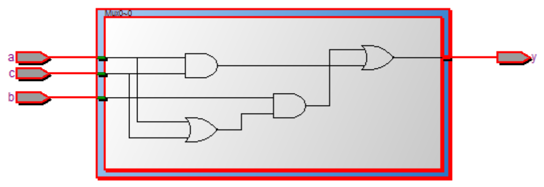
```
library ieee;    use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity    majoritaMux is
          port (   a, b, c : in std_logic;   y : out std_logic );
end;
architecture dataflow of majoritaMux is
begin
   with std_logic_vector'(a & b & c) select
          y <=    '1' when "011" | "101" | "110" | "111",
                  '0' when others;
end;
```

If we look at the translation of majorityMux code, its initial step includes a multiplexer minimized to the logical equation Eq2.4-1 that we used for our first majority.

Úvodní průchod - RTL Map



Minimalizace - Technology Map



Rovnice: (a AND c) OR((a OR c) AND b)
= (a AND c) OR (a AND b) OR (c AND b)

Figure 6 - Compilation of MajoritaMux

Advanced note: *The VHDL standard also allows using scope if the address type has a defined arrangement, which suggests the possibility for changing* when *the line at the top:*

```
   when "011" | "101" | "110" | "111",
```

to comprehend statement when "011" | "101" to "111",

Quartus also compiles it correctly, but std_logic_vector *has no ordering defined, so it's a non-standard construction that may not work in other compilers.*

The portable code could look like this:

```
with unsigned(std_logic_vector'(a & b & c)) select
   y <=    '1' when "011" | "101" to "111",
           '0' when others;
```

Here we first specified that we want to understand std_logic_vector'(a & b & c) as an unsigned number that has an already defined order, so we can replace three consecutive unsigned binary values "101" | "110" | "111" *by range* "101" to "111".

Library ieee.numeric_std *defines* unsigned *again as a group of signals, but with defined arithmetic operations. We deal with them in a paragraph on page 27.*

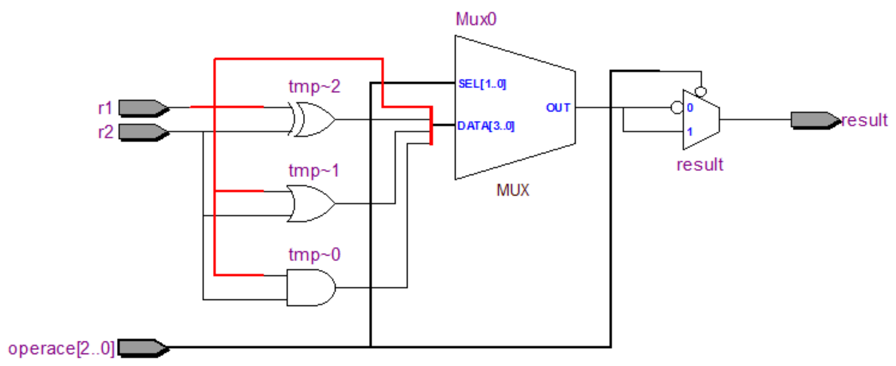
3.1.b @Example V. - Logical ALU

The keyword when **can be preceded by a logical expression** because it specifies an input value. We demonstrate this on a logical ALU that can perform basic logical operations with inputs r1 and r2**.**

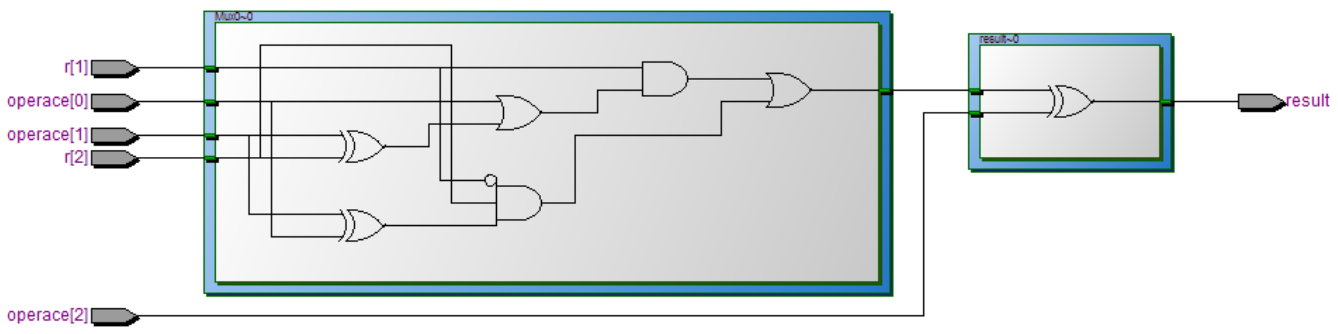
```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity LogALU is
port (  oper : in std_logic_vector(2 downto 0);        -- select required operation
        r1, r2 : in std_logic;                          -- inputs into ALU
        result : out std_logic);                        -- result of ALU
end;
architecture dataflow of LogALU is
signal tmp : std_logic;
begin
   with  oper(1 downto 0) select
         tmp <= r1 AND r2 when "00",
                r1 OR r2 when "01",
                r1 XOR r2 when "10",
                r1 when others;
   with  oper(2) select   result <= tmp when '0', NOT tmp when '1';
end;
```

We have chosen single-bit input r1 and r2 of std_logic type to simply schematic. We could certainly change r1 and r2 to std_logic_vector of any length, but in that case RTL Map and Technology Map would contain the same elements as we see below, but in larger numbers — each bit of input would have its own.

First passes - RTL Map



Minimalization - Technology Map



Quartus converted the first multiplexer into the 4-input logic circuit but replaced with XOR gate the second multiplexer performing the negation of the result.

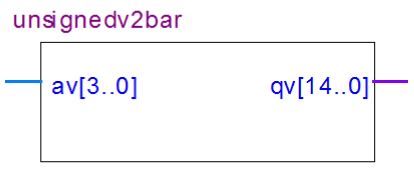
Check that statemant:

```
        with operace(2) select result <= tmp when '0', NOT tmp when '1';
```
is fully equvalent with simple command:
```
                              result <=  operace(2) XOR tmp;
```

3.2 @Example VI. - Decoder for long bar indicator

We can extend the previous bar indicator to multi-bit output. We use our template again:

```
--Long bar indicator
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity unsignedv2bar is
port    (  av : in std_logic_vector(3 downto 0) ;
           qv : out std_logic_vector(14 downto 0));
end;
```



```
architecture dataflow of unsignedv2bar is
signal tmp : std_logic_vector(15 downto 0);
begin
   with  av select
      tmp <=  X"0000" when X"0",    X"0001" when X"1",    X"0003" when X"2",    X"0007" when X"3",
              X"000F" when X"4",    X"001F" when X"5",    X"003F" when X"6",    X"007F" when X"7",
              X"00FF" when X"8",    X"01FF" when X"9",    X"03FF" when X"A",    X"07FF" when X"B",
              X"0FFF" when X"C",    X"1FFF" when X"D",    X"3FFF" when X"E",    X"7FFF" when X"F";
   qv <= tmp(14 downto 0);
end;
```

We have defined an auxiliary vector tmp for possibility to use the hexadecimal notation X ".." that can specify only lengths divisible by 4, which qv does not meet. First, we connected the values to tmp and output the necessary signals to it. We did not increase the complexity, because signal tmp(15), which is not used in the resulting qv, will be removed while minimizing RTL Map to Technology Map.

Even so, our code is far from elegance. If we look at the keywords when, where we specify the choices, we find out just numbers from 0 to 15 written in hexadecimal notation. In reality, they are indexes into an array! We could write our code more elegantly, but first we need to look at the array creation.

3.3 Creating custom types and attributes in VHDL

Standard library ieee.std_logic_1164 defines std_logic_vector as the array:

```
        type std_logic_vector is array (natural range<>) of std_logic;
```

where

- type is keyword
- std_logic_vector is here the name of created data type;
- natural means integer number limited to the range from 0 to maximal integer number of the used implementation of a compiler (in Quartusu up to 2^{31}-1)
- range is the keyword of interval. In connection with natural, it lays inside 0 to maximum integer.
- range <> means unspecified range, we must write it when using type.

If we define two types with the same structure, they are different from the compiler point of view, e.g.:

```
        type std_logic_vector_A is array (7 downto 0) of std_logic;
        type std_logic_vector_B is array (7 downto 0) of std_logic;
        signal slvA1, slvA2 : std_logic_vector_A;
        signal slvB1, slvB2  : std_logic_vector_B;
```

If we now assign by <= the constant values, then everything goes well; we work with members of vectors that have all std_logic data types

```
slvA1 <= X"12"; slvB1 <= X"34";  -- OK, it is possible
```

If we try to assign memebers of both groups to each other, it deos not work:

```
slvB2<=slvA1;  slvA2<=slvB1;      -- Error  - different typesi
```

Strongly typed languages, including VHDL, do not test the matches of the internal structures. They verify only names of data types. However, there is a possibility to define a **subtype** narrowing the original type. If we rewrite the definitions above as:

```
subtype std_logic_vector_A is std_logic_vector (7 downto 0);
subtype std_logic_vector_B is std_logic_vector (7 downto 0);
signal slvA1, slvA2 : std_logic_vector_A;
signal slvB1, slvB2  : std_logic_vector_B;
```

then the compiler accepts the assignements, because the subtypes are considered fully compatible with the default data type and all its subtypes:

```
slvA1 <= X"12";  slvB1 <= X"34";  slvB2<=slvA1;  slvA2<=slvB1;
```

Summary: By using **type** definitions, we create isolated data types, while **subtype** allows including a new data type into members of an already existing data type group.

VHDL attributes allow querying definition parameters. It is a distant analogy of "property" known from classes in Java or C #, but they are only not separated by dots, but the apostrophes. Let's add definitions:

```
subtype std_logic_vector_C is std_logic_vector (5 to 27);
signal slvC : std_logic_vector_C;   --i.e., signal slvC : std_logic_vector (5 to 27);
```

then data types based on arrays allow the following attributes.

Attribute used with	slvA1, slvB1	slvC	
LEFT	7	5	*left value of the range*
RIGHT	0	27	*right value of the range*
LOW	0	5	*the lower value of the range*
HIGH	7	27	*the higher value of the range*
LENGTH	8	23	*number of elements in range*
ASCENDING	*FALSE*	*TRUE*	*Boolean TRUE when defined with* ***to***
RANGE	7 **downto** 0	5 **to** 27	*definition range*
REVERSE_RANGE	0 **to** 7	27 **downto** 5	*reverse definition range*

Table 5 - Attributes of types based on array

Attributes of ranges the VHDL get range data type, which is useful for defining similar signals. We can refer to the previous definitions for faster changes when the original range needs to be adjusted.

```
signal X : std_logic_vector(31 downto 0);
signal novy1 : std_logic_vector(X'RANGE);          -- (31 downto 0)
signal novy2 : std_logic_vector(X'REVERSE_RANGE);          -- (0 to 31)
```

We can also define an array of arrays, so we introduce it for our large bar indicator, again with a length of 16 to use the X "" hexadecimal assignment.

First, we must define array as a type, then we can create its instances:

```
type BarArray_t is array(0 to 15) of std_logic_vector(15 downto 0);
```

Only now we can create an initialized array constant:

```
constant barArray : BarArray_t := (  X"0000", X"0001", X"0003", X"0007", X"000F",X"001F",X"003F", X"007F",
                                     X"00FF", X"01FF",X"03FF", X"07FF", X"0FFF", X"1FFF", X"3FFF", X"7FFF");
```

We could also define barArray as a signal:

```
signal barArray : BarArray_t := (   X"0000", X"0001", X"0003", X"0007", X"000F",X"001F",X"003F", X"007F",
                                    X"00FF", X"01FF", X"03FF", X"07FF", X"0FFF", X"1FFF", X"3FFF", X"7FFF");
```

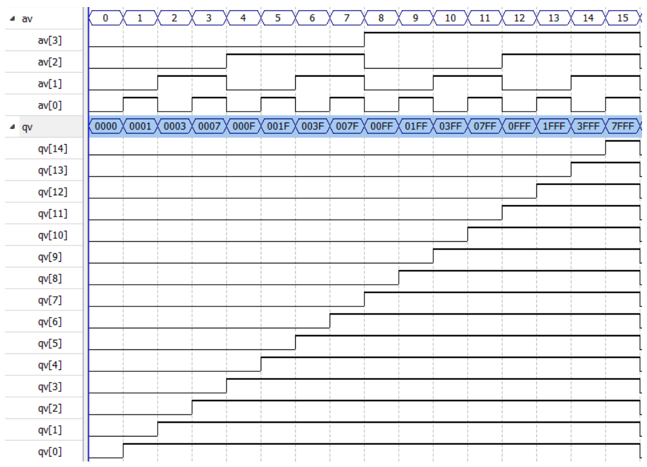
However, when using constant, we have specified that no output can be connected to the values.

3.4 @Example VII. - Decoder for large bar indicator with array

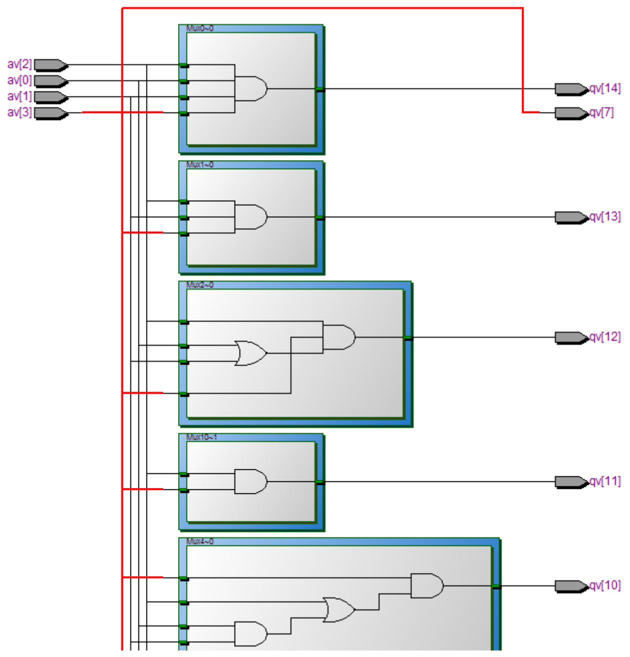
By using attributes and arrays, we simplify the previous code.

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;

entity unsignedv2bar1 is
   port (  av : in std_logic_vector(3 downto 0) ;
           qv : out std_logic_vector(14 downto 0));
end;

architecture dataflow of unsignedv2bar1 is
type BarArray_t is array(0 to 15) of std_logic_vector(15 downto 0);
constant barArray : BarArray_t := (X"0000", X"0001", X"0003", X"0007", X"000F", X"001F",X"003F", X"007F",
                                   X"00FF", X"01FF",X"03FF", X"07FF", X"0FFF", X"1FFF", X"3FFF", X"7FFF");
signal tmp : std_logic_vector(barArray(0)'RANGE);

begin
  tmp <= barArray( to_integer(unsigned(av)) );
  qv <= tmp(qv'RANGE);
end;
```

Simulation



A part of Technology Map



Indexes are always integer numbers, so our new code includes to_integer(unsigned(av)) conversion of std_logic_vector to an integer scalar. The conversion has two phases. First, we specify that we want to consider the group of signal av as an unsigned number, and then we convert it to integer.

Note:We can simplify tTwo-step conversion std_logic_vector to integer by defining our conversion function. VHDL offers a wide range of overloading, even for operators. We describe in lectures.

We could write the last lines **tmp** <= **barArray**(to_integer(unsigned(**av**))); **qv** <= **tmp**(qv'RANGE); by brief signal statement as

```
qv <= barArray( to_integer(unsigned(av)) )(qv'RANGE);
```

i.e., without auxiliary tmp signal. Over time, we may use such abbreviated statements if we know well how used them, but we do not have to. Condensed statements save only code text. The circuits remain the same. Also in VHDL, the rule holds that the code should remain clear to our eyes.

Integers in VHDL

VHDL takes integers as dimensionless scalars. Unlike processors, the circuit can use any length according to its momentary needs. The compiler implementation in Quartus allow integes from 1 bit up to 32 bits. The VHDL standard itself does not limit the maximum length of the integers.

In conventional computers, if we say integer, we know its bit length, according to the type of runtime environment, but in VHDL this does not hold. Because of this, input or output of entities should not use integer. Some implementations forbid integers in entity ports for compatibility problems.

For entity inputs and outputs, we can use signed and unsigned, which have exactly defined ranges in the same style as std_logic_vector:

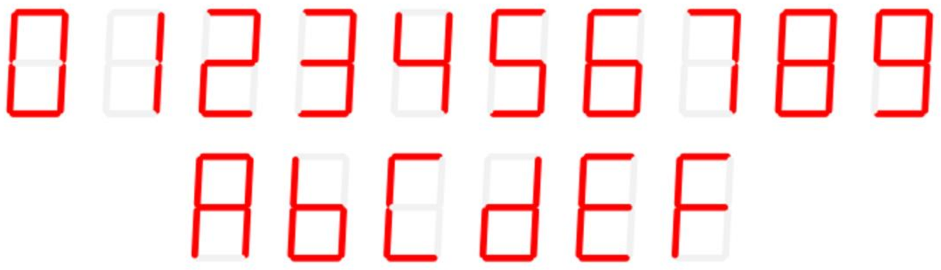
```
type std_logic_vector is array (natural range<>) of std_logic;
type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range <>) of std_logic;
```

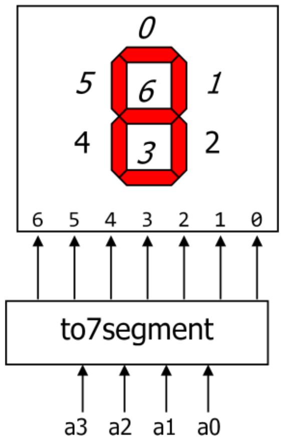
The definitions signed and unsigned differ in the fact that they allow arithmetic operations.

Question: Why std_logic_vector has no arithmetics? It is intentionally without arithmetics so we must first specify its binary number format! Poorly designed circuits can cause significant damages, so VHDL is over-typed to avoid confusion and unwanted operation. We have already mentioned in the introduction that VHDL came out from ADA programming language designed for military system design, where faultlessness matters.

3.5 *** Practice 2: Decoder for 7-segment display

Build a circuit with four inputs **a3**, **a2**, **a1** and **a0** that shows a hexadecimal digit on the 7-segment display. We see the most frequent numbering of segments on the right. Input **a3** has the highest weight and **a0** the lowest.





We write for the DE2 board where a segment lights on '0' applied to its corresponding input. For example, if we want to light up the segments to show number 1, we send "1111001" to the 7-segment inputs.

If we created the 7-segment display decoder by logic functions, it would take us several hours. Using an array similarly to previous code, we assemble it in minutes.

Try to design the circuit by yourself. Look at its RTL and Technology Map.

You find out one possible solution in the appendix on page 60.

4 @Example VIII. - VHDL statement when...else

Task:

For three interrupt request inputs Int3, Int2, Int1, design a priority decoder that sends the number of the highest active request.

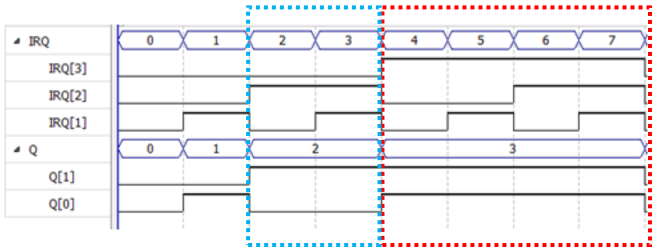
If Int3 with the highest priority is in '1', the output is 3, regardless of states of other inputs. Int1 has the lowest priority, and its number 1 appears on the output only if the remaining inputs are in '0'. If there is no active interrupt request, the output is 0, see the truth table.

Priority decoder					
Inputs			**Outpus**		
IRQ3	**IRQ2**	**IRQ1**	Q1	Q0	Number
1	**-**	**-**	**1**	**1**	3
0	**1**	**-**	**1**	**0**	2
0	**0**	**1**	**0**	**1**	1
0	**0**	**0**	**0**	**0**	0

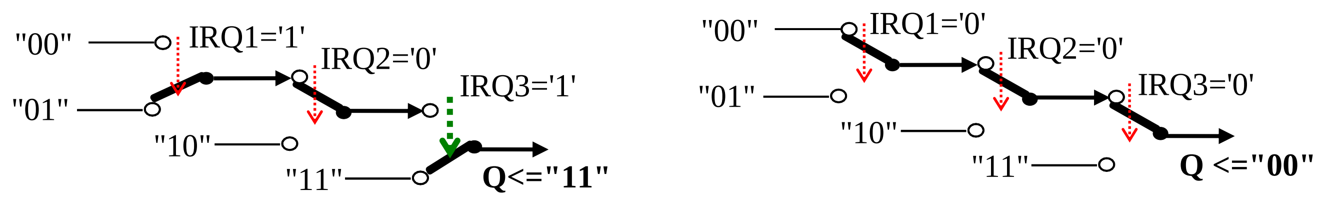
We could, of course, construct logic equations or create a multiplexer by construction with ... select

```
    with unsigned(IRQ) select              --by conversion to unsigned, we define ordering
       q <=   "11" when "100" to "111",    -- we utilize ordering to short the choice list by range
              "10" when "010" | "011",
              "01" when "001",
              "00" when others;
```

We can also solve more interrupt requests, e.g. 15, by a similar procedure when we write a lengthy list of the input ranges. We rather use the fact that higher level interrupt forces ignoring all lower inputs.

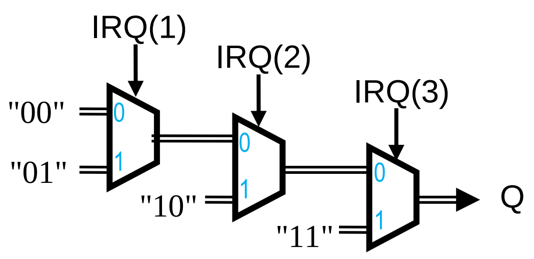


A cascade of two-input multiplexers offers such a feature. Individual interrupt requests form conditions for switching of particular multiplexers.



If the first IRQ3 switch is down, then a combination of "11", i.e., number 3, is sent to output Q, regardless of the positions of the other switches. If IRQ3 is up, the influence of other switches according to the order (priority) of the conditions is applied.

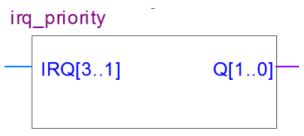
We aggregate requests into an IRQ vector, in order of their priority, to easier handle them. We draw the switches with a schematic symbol of the multiplexer:



Similar cascade corresponds to VHDL construction: `when …else:`

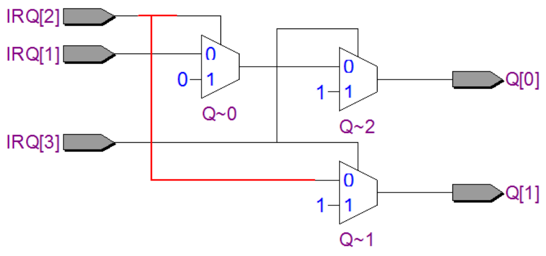
VHDL code irq_priority.vhd | **Schematic symbol**

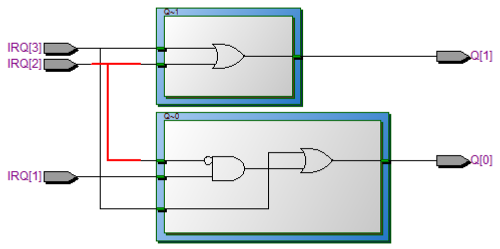
```
--Interrupt request priority
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity irq_priority is
   port (  IRQ : in std_logic_vector(3 downto 1);
           Q :    out std_logic_vector(1 downto 0) );
end;

architecture dataflow of irq_priority is
begin
   Q<= "11" when IRQ(3)='1' else
       "10" when IRQ(2)='1' else
       "01" when IRQ(1)='1' else
       "00";
end;
```



The condition after when **requires** boolean type, and therefore, we cannot insert just logic value, *when IRQ(3) else* results in an error. We need a boolean result, so we wrote IRQ(3)='1'.

RTL Map | Technology Map





First, Quartus has built a cascade of multiplexers for each bit separately and reduced its level. Then, it minimized it. Moreover, it reduced the cascade correctly:

```
Q(0)<=  '1' when IRQ(3)='1' else       ←reduced       Q(0) <=  '1' when IRQ(3)='1' else
        '0' when IRQ(2)='1' else        in RTL                 '0' when IRQ(2)='1' else
        IRQ(1);                                                '1' when IRQ(1)='1' else
                                                               '0';
Q(1)<=  '1' when IRQ(3)='1' else       ← reduced      Q(1) <=  '1' when IRQ(3)='1' else
        IRQ(2);                         in RTL                 '1' when IRQ(2)='1' else
                                                               '0' when IRQ(1)='1' else
                                                               '0';
```

Statement when else allows using expressions both in terms of conditions and on inputs, while the previously discussed with...select statement only allows utilizing expressions on its inputs.

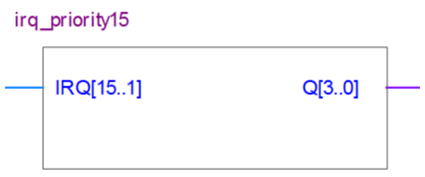
Note that **we do not repeat the previous conditions** inside when ... else statement. It would be pointless to write something in style:

```
Q<=   "11" when IRQ(3)='1' else
      "10" when IRQ(3)='0' and IRQ(2)='1' else
      "01" when IRQ(3)='0' and IRQ(2)='0' and IRQ(1)='1' else
      "00";
```

Even if the code is not incorrect, the highlighted parts implicitly follow from the previous conditions. The values "10", "01", "00" can be selected only when INT (3) = '0'. Redundant tests just make the code unclear. "*So why are they inserted? - excepts as bites for the delete key!*" ☺

What would the code look like if we needed more interrupt inputs? We add more rows by the well-known copy-paste-edit method.

4.1.a @Example IX. - Priority decoder with 15-inputs

VHDL code irq_priority15.vhd	Schematic symbol
See code below	

```
--Interrupt request priority
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity irq_priority15 is
        port (   IRQ : in std_logic_vector(15 downto 1);
                 Q : out std_logic_vector(3 downto 0) );
end;
architecture dataflow of irq_priority15 is
begin
  Q<=  X"F" when IRQ(15)='1' else
       X"E" when IRQ(14)='1' else
       X"D" when IRQ(13)='1' else
       X"C" when IRQ(12)='1' else
       X"B" when IRQ(11)='1' else
       X"A" when IRQ(10)='1' else
       X"9" when IRQ(9)='1' else
       X"8" when IRQ(8)='1' else
       X"7" when IRQ(7)='1' else
       X"6" when IRQ(6)='1' else
       X"5" when IRQ(5)='1' else
       X"4" when IRQ(4)='1' else
       X"3" when IRQ(3)='1' else
       X"2" when IRQ(2)='1' else
       X"1" when IRQ(1)='1' else X"0";
end;
```

Table 6 - Priority decoder

If we try its simulation, we obtain a long graphical output. Computerized simulations, called testbenches, suit better for complicated circuits in HDL languages. We present them at lectures.

4.2 Conditional Assignment

C-language has **? :** conditional assignment operator. Two-input multiplexer represents its circuit analogy. We can create it either by with... select command or by when...else.

The with ... select command always creates a single multiplexer that has only two inputs for the conditional assignment. The cascade of multiplexers when...else have also a single two-input multiplexer, i.e., both constructions lead to the same result in the case of conditional assignment.

If we have, for example, signals:

```
signal A, B, C: std_logic_vector (7 downto 0);
signal sel, x, y, z: std_logic;
```

we can write the conditional assignment by two ways:

when ... else		**with ... select**
A<= **B** when sel='1' else **C**;	*←equivalent→*	with sel select **A**<= **B** when '1', **C** when '0';
x<=**y** when sel='1' else **z**;	*←equivalent→*	with sel select **x**<= **y** when '1', **z** when '0';

In practice, shorter notation when ... else is more often preferred. Remember, however, there is boolean value after when.

On the other hand, the with...select statement requires constants with values known in compile-time and data type identical to signal used for selections.

4.3 *** Practice 3: 8-input priority inhibitor

Task: Design the circuit with 8 inputs av(7 downto 0) and 8 outputs qv(7 downto 0). The circuit passes only the highest input in the '1' state on its numerically matching output, which means that one output only is at most in '1' - that's why we called the circuit a priority inhibitor.

If we have only 3 inputs av(2 downto 0) and 3 outputs qv(2 downto 0), then the truth table looks as:

Úplná pravdivostní Table					
Vstupy			Výstupy		
av(2)	**av(1)**	**av(0)**	**qv(2)**	**qv(1)**	**qv(0)**
0	0	0	0	0	0
0	0	1	0	0	1
0	1	*0*	0	1	0
0	1	*1*	0	1	0
1	*0*	*0*	1	0	0
1	*0*	*1*	1	0	0
1	*1*	*0*	1	0	0
1	*1*	*1*	1	0	0

Zkrácená pravdivostní Table					
Vstupy			Výstupy		
av(2)	**av(1)**	**av(0)**	**qv(2)**	**qv(1)**	**qv(0)**
0	0	0	0	0	0
0	0	1	0	0	1
0	1	-	0	1	0
1	-	-	1	0	0

Table 7 - The truth table of priority inhibitor with 3 inputs and outputs

Where would we use such a circuit? Imagine that we have 8 switches controlling some functions, and we want only one switch closed at a time. The user may inadvertently or intentionally ("tempter one") make multiple choices at once. Our priority inhibitor suppresses them according to switch priorities, so we are confident in selecting a maximum of one function.

Design VHDL code.

Look at RTL Map and Technology map. Check your solution by its simulation.

Hint: We can quickly design the code by editing the previous task, and Table 7 suggests how to write it.

One possible solution is presented in the appendix on page 62.

5 @Example X. - VHDL statement generic and for-generate

In chapter 3, on page 20, we created the bar indicator with the truth table on the right, which we have later extended to more inputs.

We previously based our solution on a predefined array, which we had to adjust for each required length of the output.

Truth Table of Indicator				
Address input		**Output**		
Hodnota	**av**	qv(2)	qv(1)	qv(2)$_0$
0	00	0	0	0
1	01	0	0	**1**
2	10	0	**1**	**1**
3	11	**1**	**1**	**1**

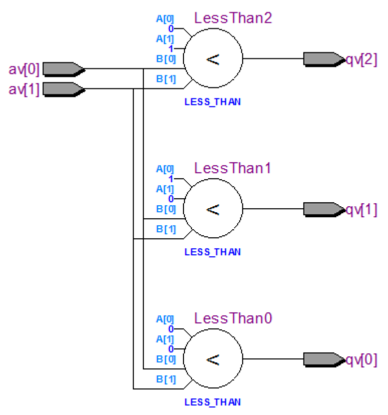
In this section, we show how to create a universal code, a bar indicator for any length that no longer needs the usage of any predefined array.

It follows from the truth table that we can write the bar indicator code with the aid of conditional assignments, taking into account that usual arithmetic operations also exist for the unsigned type, including comparison operators, which have the same notation as in C language except for equals and not equal, they are in VHDL as = (not C ==) and /= (not C !=).

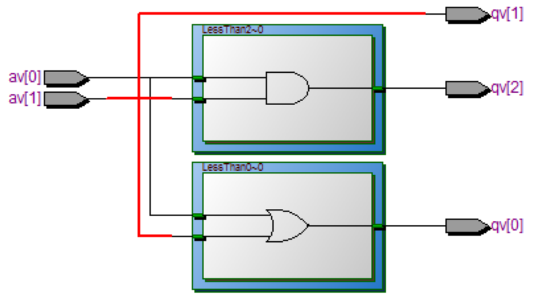
```
--Bar indicator
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity unsignedx2bar is
    port (  av : in std_logic_vector(1 downto 0) ;
            qv : out std_logic_vector(2 downto 0));
end;
architecture dataflow of unsignedx2bar is
signal x : unsigned(av'RANGE);
begin
    x <= unsigned(av);
    qv(0) <= '1' when x>0 else '0';
    qv(1) <= '1' when x>1 else '0';
    qv(2) <= '1' when x>2 else '0';
end;
```

In the initial step, the compiler creates very different RTL Map meta-scheme that employes comparators, but after the following minimization, it generates a logic circuit identical to the previous result with the with-select statement, see Table 4, on page 21.

Initial RTL Map



Technology Map



We can quickly multiply the conditions for individual bits by copying with subsequent editions since they differ only in the indexes and the comparisons. However, such a procedure is not universal. VHDL also allows writing analogies for the cycle, but it does not repeatedly execute the same part of code with different values, because this is not possible inside circuits. The for-generate loops are always translated in style known as inline expansion from programming that results in multiple copies of a loop body with different parameters.

In our example, the for-generate has the following shape corresponding to command replacements:

For-loop		**Generated connection statements**

`rep: for i in 0 to 2 generate`
`        qv(i) <= '1' when x>i else '0';`
`    end generate;`
	$\longrightarrow$	`qv(0) <= '1' when x>0 else '0';`
		`qv(1) <= '1' when x>1 else '0';`
		`qv(2) <= '1' when x>2 else '0';`

The generation cycle has a dual syntax, either for one conditional generation:

```
label: if <expression> generate  <concurrent-statements>
        end generate [label];
```

alternatively, for repeated generations:

```
label: for <parameter> in  <range>  generate   <concurrent-statements>
        end generate [label] ;
```

where

- If we write the first possibility, then **<expression>** giving boolean type must be globally static, i.e., it must evaluate to a constant value known in compile-time.
- **label** is a mandatory identifier separated by a colon. We have to define it even if we don't use it anywhere. In the example above, we chose the name "rep", but we could enter another one, such as "cycle1", and so on.
- **<parameter>** represents a unique cycle parameter that takes gradually values specified by <range> . We cannot use here any already defined signal. The parameter is created inside for-generate loop, exists only here, and statements can only read its value, not overwrite.
- **<range>** specifies an interval in the same way as in case of arrays, either with downto or to.
- end generate [label]; — we must terminate the statement by either end generate or end generate label; Keyword generate after end is mandatory, we may not omit it here;
- **<concurrent-statements>** contains block of one or more "concurrent" statements, i.e., **<=** assignment, or statements with..select , when..else, or also for-generate (we can nest for-generate loops). It is also possible to insert port map statement, the topic of chapter 6.
- **Statements inside for-generate must always be complete.** Although we could sometime take great advantage of generating partial statements, such as individual lines of conditions in "when...else" or adding members into expressions, no such constructions are possible.
- We can use for generate statements only in begin-end blocks of architectures, not in entities. It is also possible to add definitions of local variables. In this case, begin keyword is inserted after generate. We add local declarations before it (the same arrangement as architecture - begin - end;). Those interested can find more detailed information in the VHDL reference manuals, but they should first read chapter 6.6 on page 49, containing the example of such definition and important discussion of possible problems.
- Do not confuse the concurrent statement for generate of dataflow modeling style with the sequential statement for loop, which already belongs to the behavioral modeling style. The compiler transforms all for loop statements during compilation into for generate constructions.

5.1 @Example XI. - Declaration generic

By changing the range in for-generate, we can create an arbitrarily long array, but we still do not obtain a universal circuit. We need to parameterize our design, i.e., to utilize a particular analogy of the parameters inside constructors od classes in classical programming.

To do so, we introduce generic definition, which is inserted into the entity and defines read-only elements changeable when creating circuit instances with map statements, the topic of chapter 6.

```
--Configurable bar indicator
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;

entity uint2bar0 is
    generic (  AV_LENGTH : integer := 4;  -- bit length of av input
               QV_LENGTH : integer := 16); -- bit length of qv output
    port (  av : in std_logic_vector(AV_LENGTH-1 downto 0) ;
            qv : out std_logic_vector(QV_LENGTH-1 downto 0));
end;

architecture dataflow of uint2bar0 is
signal x : unsigned(av'RANGE);
begin
    x <= unsigned(av);
 rep: for i in 0 to QV_LENGTH-1 generate
      qv(i) <= '1' when x>i else '0';
   end generate;
end;
```

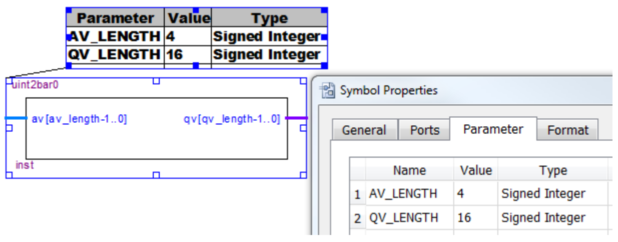
The generic definition that has recently appeared in the entity section may contain several parameters but VHDL implementations usually allow here only integer types:

```
        generic(   name1: type1 := default_value1;   name2: type2 := default_value2;
                   . . .
                   nameN: typeN := default_valueN  );
```

Note that, similarly to port blocks, we do not write ; character after the last element of the generic list.

Defined generic parameters always behave as constants inside the VHDL code, and we must specify their default values assigned by **:=,** not by <= "concurrent assignment" because they are not connections. The compiler replaces all parameters with the constant values entered in their entity instances.

The schematic symbol now contains parameter specifications that can have different values for each instance. In the symbolic schema editor, we change the values of generic parameters with the aid of context menu of the selected symbol.



Parameter	Value	Type
AV_LENGTH	4	Signed Integer
QV_LENGTH	16	Signed Integer

	Name	Value	Type
1	AV_LENGTH	4	Signed Integer
2	QV_LENGTH	16	Signed Integer

Figure 7 - Usage of bar indicator

Note: In Quartus block diagram schematic editor, we can show/hide generic parameter values for all instances in the menu. Click by right mouse anywhere on the free editor area and check/unckeck the option Show->Show Parameter Assignments.

5.2 @Example XII. - Checking generic parameters

If someone enters the wrong generic parameters in an instance of our uint2bar circuit, e.g., zero or negative numbers, then the compiler announces errors. However, the user only knows that something was wrong without a lack of detailed information.

To check the generic parameter values, we can add checks into the entity section. They are called passive processes because they are executed only at compile time. The compiler creates no circuits from them. They serve solely for checking consistency of parameters.

```
--Configurable bar indicator with the checking of generic parameters
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity uint2bar is
     generic (   AV_LENGTH : integer := 4;  -- bit length of av input
                 QV_LENGTH : integer := 16); -- bit length of qv output
     port (  av : in std_logic_vector(AV_LENGTH-1 downto 0) ;
             qv : out std_logic_vector(QV_LENGTH-1 downto 0));
begin -- passive process - its statements are executed in compile time only
   assert AV_LENGTH >0 AND QV_LENGTH >0
   report "uint2bar requires AV_LENGTH >0 and QV_LENGTH >0 " severity failure;

   assert  2**AV_LENGTH-1 <= QV_LENGTH
   report "uint2bar displays only range 0 to " & integer'Image(QV_LENGTH-1) & " of av input"
   severity warning;
end;

architecture dataflow of uint2bar is
signal x : unsigned(av'RANGE);
begin
        assert false report "AV max value ="& integer'Image(2**av'LENGTH-1) severity note;

         x <= unsigned(av);
  rep: for i in 0 to QV_LENGTH-1 generate  qv(i) <= '1' when x>i else '0';
        end generate;
end;
```

Statement assert differs from throw exceptions of programming languages. No reasonable way exists how to implement exceptions in circuits. All asserts evaluate at compile-time and correspond to the same name construction in Java, C # and C ++. For example, C programs add assert analogy with #include <assert.h>

In VHDL, assert statements have 3 parts:

```
assert <condition>  report <string> severity <severity_level>;
```

its report and severity parts are optional; we can omit any of them or both.

- <condition> an expression returning boolean type. Its value is evaluated only in compile-time, so we can utilize complex constructions. Operator ** in the expression 2**AV_LENGTH-1 calculates 2^{AV_LENGTH} -1, i.e., the largest possible value of unsigned with given bit length.
- <string> represents text shown on false, i.e., on unsatisfied condition;
- <severity_level> specifies requested compiler response, which is either:
  - **note** — displays text with the mere information flag;
  - **warning** — represents warning message;
  - **error** — denotes error that breaks the compilation immediately.

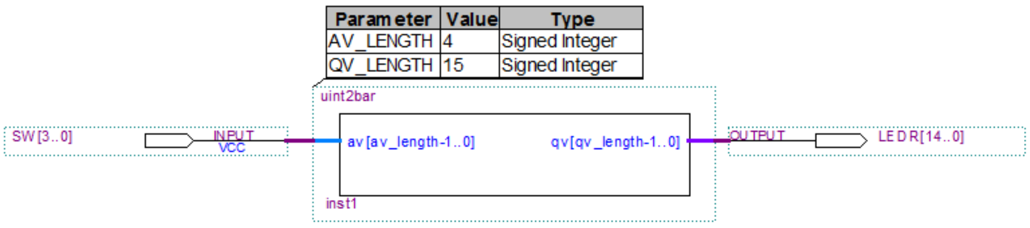
  *Note: If we omit "*severity <severity_level>*" part, the default responce is* error.

We can insert several assert statements and use them also in begin end sections of architecture blocks. Even here, however, their condition must evaluate to a constant known in compile-time.

The second and third asserts in the example above show the constructions of texts. We convert numbers to strings by the Image() attribute that belongs to the type, and it is predefined for most of them. We merge texts by &.

<u>Note:</u> Reports serve for simulating of circuits or detecting of inconsistencies. They do not exist in circuits. Creating text output is not a strong point of VHDL. However, some external libraries allow almost C-style prints and freely usable for non-commercial purposes, for example, one is at https://www.easics.com/products/freesics .

The circuit below

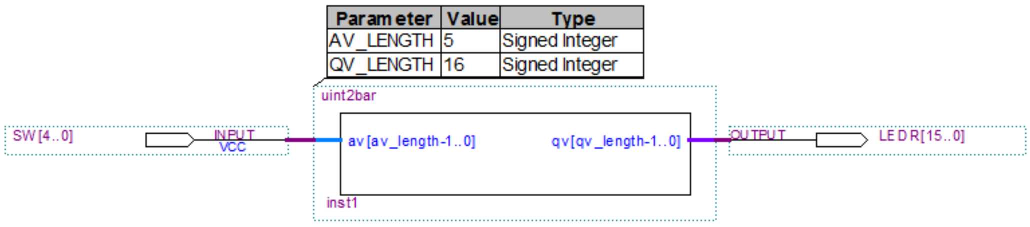


Parameter	Value	Type
AV_LENGTH	4	Signed Integer
QV_LENGTH	15	Signed Integer

announces only:

Info (10544): VHDL Assertion Statement at uint2bar.vhd(18): assertion is false - report "AV max value =15" (NOTE)

If we enter different generic parameters:



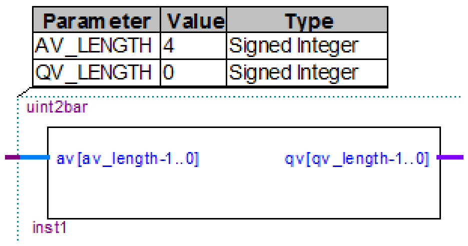
Parameter	Value	Type
AV_LENGTH	5	Signed Integer
QV_LENGTH	16	Signed Integer

we see reports:

Warning (10651): VHDL Assertion Statement at uint2bar.vhd(11): assertion is false - report "uint2bar displays only range 0 to 16 of av input" (WARNING)

Info (10544): VHDL Assertion Statement at uint2bar.vhd(18): assertion is false - report "AV max value =31" (NOTE)

In the symbolic schema editor, the compilation fails if we use incorrect values :



Parameter	Value	Type
AV_LENGTH	4	Signed Integer
QV_LENGTH	0	Signed Integer

because the compiler always converts any symbolic editor schema to VHDL code and the error occurs when doing so:

Error (275050): Bus range -1 is illegal in signal "port "qv[qv_length-1..0]" (ID uint2bar:inst1)"

If we would create an instance of our circuit directly in VHDL code, see chapter 6, we see the message:

Warning (10445): VHDL Subtype or Type Declaration warning at uint2bar.vhd: subtype or type has null range

Error (10652): VHDL Assertion Statement at uint2bar.vhd(9): assertion is false - report "uint2bar requires AV_LENGTH >0 and QV_LENGTH >0 " (FAILURE or ERROR)

5.2.a What else is missing?

The last code is already close to professional VHDL descriptions for its high universality. It can remain unchanged for different bus widths, and yet is not perfect. It still misses:

- **Testbench** – any professional design is considered incomplete unless there is also a testbench file written in a VHDL behavioral modeling style for automatic simulation. We explain at the lectures.
- **License** – describes distribution rights. Akademic code have "GNU General Public License".
- **Language** – If we want to make the VHDL code more accessible, we choose English for both comments and identifiers.
- **Comments** - we have included only the most necessary texts to shorten the lines and we put descriptions in the following text. Add comments at least to the entity block and the main variables so users do not need to decipher their meaning. Include an explanation of what the code actually performs, and possibly the author's name. Also, it is useful to insert notes in front of code subunits specifying their purpose.
- Note: Over time, many designers find out that they wrote comments for themselves, so that they later know what they had written long ago and why. ☺

5.3 *** Practice 4: Universal priority inhibitor

Task: Design a circuit passing only the highest input in '1' on its numerically matching output, so that at most one output is in '1'. You have solved the circuit for 8 inputs on page 31. Try to modify its code for any number of inputs from 2 to N, and even with parameter value checking.

Hint: The for-generate loop does not allow adding rows to when-else. Write the inhibitor for signals using conditional commands. However, it is not possible to have a universal flag since we can only connect to the signal once. So create an auxiliary vector, e.g., named inh (inhibitor) with range matching the output. Using OR operation, create recursive conditions that some higher output is already in '1'. Inh(i) values propagates in lower indexes. The output with the highest weight lies outside for-generate loop, because the N-1 output is always equal to the N-1 input — only outputs with lower indexes i are blocked by inh.

One possible solution is on page 63.

5.4 Initialization of variable length vectors by association others=>

The keyword others means all unused values. It is suitable for associative assignment of values to vectors for which we specified lengths by generic, so we do not know their length in advance because we specify their different sizes at the time of usage. We write association by => operator.

If we use an association in constants, then on its left side stands the indices in the array, with possible directions both to and downto, and on the right side the required value of its member(s).

The following demo example illustrates the options best:

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity asociace is
    generic(N:natural:=8);
    port ( X1, X2, X3, X4, X5, X6, X7 : out std_logic_vector(N-1  downto 0)  );
begin
    assert N>=7 report "Example requires N>=7" severity failure;
end entity;
architecture structural of asociace is
begin                                                 --   for N = 8        for N = 12
    X1 <= (others=>'0');                              --   X1="00000000"    X1="000000000000"
    X2 <= (others=>'1');                              --   X2="11111111"    X2="111111111111"
    X3 <= (others=>'Z'); --                           --   X3="ZZZZZZZZ"    X3="ZZZZZZZZZZZZ"
    X4 <= (0=>'0', others=>'1');   --                 --   X4="11111110"    X4="111111111110"
    X5 <= (N-1=>'0', others=>'1');          --        --   X5="01111111"    X5="011111111111"
    X6 <= (N-1=>'0',  1=>'0', others=>'1');       --  --   X6="01111101"    X6="011111111101"
    X7 <= (N-1 downto N-3=>'1', 1 to 3=>'1', others=>'0');  --   X7="11101110"    X7="111000001110"
end;
```

Note that we specify the value for the initialization of an association identically to the array element data type, even if the result affects multiple members because we crete associations to array elements. The std_logic_vector type has members of the std_logic enumeration data type, so we write the right-hand side of the associations in ' ' apostrophes.

In the initialization of X7, we used both the downto and to directions what we can here because we define associations for creation an initialization constant in compile-time, and the result initilizes the vector.

For example, if we wanted to give X7 the same value as the last associative list:

```
    X7 <= (N-1 downto N-3=>'1', 1 to 3=>'1', others=>'0');
```

then we need more <= statements and we do not escaped others :

```
    X7(N-1 downto N-3) <= "111";
    X7(3 downto 1) <="111"; -- Here, we select the part of X7, thus, we cannot reverse defined range
    X7(N-4 downto 4) <=(others=>'0');  X7(0)<='0';
```

Middle part of vektor X7(N-4 downto 4) has variable length and we cannot initilzed it without others. Moreover, the case N=7 annouces an error here, so we should insist on assert N>=8.

In statements, we strictly keep direction downto from the definition of our vector. The attempt of usage to direction (e.g., *X7(1 to 3) <="111";*) results in error message: "*…range direction of object slice must be same as range direction of object*".

5.5 Converting integer to unsigned and signed

Since integer data type has a variable length in VHDL, when converting from it, we must always specify the width of signed and unsigned types by a constant expression with a value known at compile-time. For example, if we have:

```
        signal xs : signed(10 downto 0);  signal xu : unsigned(5 downto 0);
        signal anyinteger : integer;
```

then we convert to the selected width by library ieee.numeric_std functions to_signed() and to_unsigned(), where we specify the width preferably the target signal length attribute, but we can alsu use a constant:

```
        xs <= to_signed( -512, xs'LENGHT);  xu<=to_unsigned(anyinteger, 6);
```

5.6 *** Practice 5: Extend interrupt priority with generic

Extend priority of interrupts from page 28 with the aid of generic definitions to universal code.

VHDL has more complex structure in this practice, so we add a fragment of its statement as a hint. Complete the marked sections. You can check your solution on page 65.

```
--Universal interrupt priority decoder
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity irq_priorityN is
   { write here correct definitions of generic parameters IRQ_MAX and Q_LENGTH  }
   port (  -- interrupt request, higher index has higher priority
           IRQ : in std_logic_vector(IRQ_MAX downto 1);
           -- unsigned number of the highest active interrupt input
           Q : out std_logic_vector(Q_LENGTH-1 downto 0) );
   { insert here tests of generic parameters consistency }
end;
architecture dataflow of irq_priorityN is
type qtmp_array_t is array { finish array type definition };
signal qtmp : qtmp _array_t;
begin
    qtmp (IRQ'LOW) <= (others=>'0') when IRQ(IRQ'LOW)='0'
                                          else to_unsigned(IRQ'LOW, Q_LENGTH);
    { insert here  for generate statement }
    Q <= std_logic_vector(qtmp(IRQ'HIGH));
end;
```

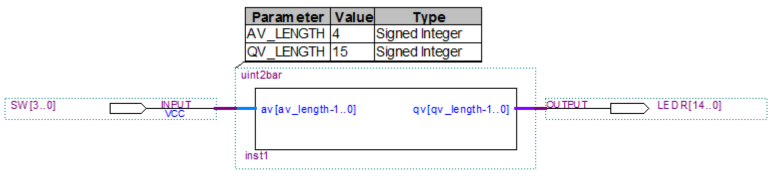
6 VHDL by Structural Modeling Style

VHDL also allows describing designs by hierarchical structure - we divide the circuit into simpler entities, which we debug separately. Finally, we define their interconnections by a text description similarly to the schema editor. VHDL does not offer clarity in case of a small graphic scheme, but it described better complex interconnections with the following advantages:

- we write text statements faster than drawings of diagrams, even for simple circuits;
- we can easier compare changes in texts among several versions;
- VHDL becomes much simpler than graphical diagrams in case of many interconnections;
- we open the possibility to debug VHDL circuits by simulation tools such as ModelSim.

6.1 @Example XIII. - Usage of bar indicator from VHDL

In the previous section, we have created a universal bar indicator and checked it in a schematic. However, every change of the generic parameters requires the edition of the SW input and LEDR output ranges.



Parameter	Value	Type
AV_LENGTH	4	Signed Integer
QV_LENGTH	15	Signed Integer

Figure 8 - Bar indicator in Quartus schematic editor

We can quickly change ranges in one place only if we create the whole schema in VHDL. Let us look at the complete code first, and then we describe its parts. For the sake of explanation, we have numbered the lines, and there are descriptions in the following paragraph for highlighting numbers of lines:

```
-- VHDL code with uint2bar  - downloadable into DE2 board                    Line     Page
library ieee;                                                                 --L01
use ieee.std_logic_1164.all; use ieee.numeric_std.all;                        --L02
library work;                                                                 --L03    41
entity demo_unit2bar is                                                       --L04
     generic (AV_LENGTH1 : integer:=4; QV_LENGTH1 : integer:=15);             --L05    41
     port (  SW :  in  std_logic_vector(AV_LENGTH1-1 downto 0);               --L06    41
             LEDR :  out  std_logic_vector(QV_LENGTH1-1 downto 0));           --L07
end;                                                                          --L08
architecture dataflow of demo_unit2bar is                                     --L09
component uint2bar                                                            --L10    41
          generic (   AV_LENGTH : integer:=4; -- bit length of av input       --L11
                      QV_LENGTH : integer:=15); -- bit length of qv output    --L12
          port( av : in std_logic_vector(AV_LENGTH-1 downto 0);               --L13
                qv : out std_logic_vector(QV_LENGTH-1 downto 0));             --L14
end component;                                                                --L15
begin                                                                         --L16
   inst1 : uint2bar                                                           --L17    42
      generic map(   AV_LENGTH => AV_LENGTH1,                                 --L18    43
                     QV_LENGTH => QV_LENGTH1)   -- no ; after ) !             --L19
      port map(av => SW, qv => LEDR);                                         --L20
end;
```

L03 – library work

The statement library work opens access to the entities we have in the Quartus project Files tab. We do this with the library statements, even if work is not a library. Its meaning is more similar to the keyword this in object programming languages. It indicates the currently open project. We did not have to insert a separate line L03. We can append work in the previous library statement at line L01:

```
library ieee, work;
```

L04 - generic

We introduced generic parameters with identifiers AV_LENGTH1 and QV_LENGTH1, i.e., different from uint2bar parameter only just for clarity. Instead of them, we could utilize AV_LENGTH and QV_LENGTH, but then it would be less apparent to which definition they relate to, if to uint2bar or demo_uint2bar.

L05 a L06 – port

- If the file is **Top-Level entity** (we see it in Quartus Project Navigator tab Hierarchy), then the compiler maps input and outputs according to Assignments. In case of DE2 board, we have imported the assignments from file "DE2_pin_assignments.csv" (the main menu Assignments ->Import Assignments). Then, SW defines the connection to the signals that lead from the DE2 switches to the FPGA inputs, while the LEDR refers to FPGA outputs driving red LEDs.
- If the file **is not** **Top-Level entity**, then the compiler never maps identifiers according to Assignments. It takes them as ordinary input and output pins. In our entity demo_unit2bar.vhd, LEDR, and SW remain in mere identifiers. However, using an identifier with name from Assignments, we specify the connection to assignment signal, if another circuit embeds the entity as its subordinate part.

L10 to L15 – component

For safety reasons, VHDL requires the insertion of the full header. It is called component. We create it by copying necessary parts from the VHDL entity of the circuit to which we refer. Our example had it in chapter 5.1 on page 34 and then extended in chapter 5.2 on page 35:

```
entity uint2bar is
   generic (   AV_LENGTH : integer := 4;  -- bit length of av input
               QV_LENGTH : integer := 16); -- bit length of qv output
   port (av : in std_logic_vector(AV_LENGTH-1 downto 0) ;
         qv : out std_logic_vector(QV_LENGTH-1 downto 0));
begin -- passive process - its statements are executed in compile time only
   assert AV_LENGTH >0 AND QV_LENGTH >0
   report "uint2bar requires AV_LENGTH >0 and QV_LENGTH >0 " severity failure;
end;
```

If we have a passive process in the entity, that is, the part opened with begin, in which we check generic parameters, we do not include it, we only insert information, not code. We replace the entity keyword by component and we write end compoment; instead of end; Alternatively, you can utilize a longer end component uint2bar; but we cannot close it only by end.

```
component uint2bar
   generic (   AV_LENGTH : integer:=4;  -- bit length of av input
               QV_LENGTH : integer:=15); -- bit length of qv output
   port(   av : in std_logic_vector(AV_LENGTH-1 downto 0);
           qv : out std_logic_vector(QV_LENGTH-1 downto 0));
end component;
```

Necessary insertions of components prevent mistaken reference to another circuit that has the randomly same name but different architecture. VHDL requires not only matching circuit names and parameter types, but also all input and output names. If it finds nothing with perfect matching, it reports an error.

Notes:

1. We can insert as many components as we wish, whether we use them or not. The translator does not require references to their declarations. Inserting a component does nothing because it is only an informational value. Therefore during code development, we do not need to delete components immediately that we do not currently utilize in map statements.
2. We can place component blocks into a package, something similar to the * .h files of C, and then we just insert paste reference to the package, see later in Chapter 7 on page 55, about creating libraries by declaring library packages.
3. We can reference the signals and parameters listed in a component only in corresponding map sections, see below. They are inaccessible from other parts of the code.
4. *VHDL'93 introduced the incomplete type declarations suitable for complex constructions, where we can omit the default values of the generic parameters AV_LENGTH, QV_LENGTH, or even vector ranges in port section. Allowed a shortened component declaration but not recommended* is:

```
component uint2bar
          generic (    AV_LENGTH, QV_LENGTH : integer);
          port(  av : in std_logic_vector;
                 qv : out std_logic_vector);
end component;
```

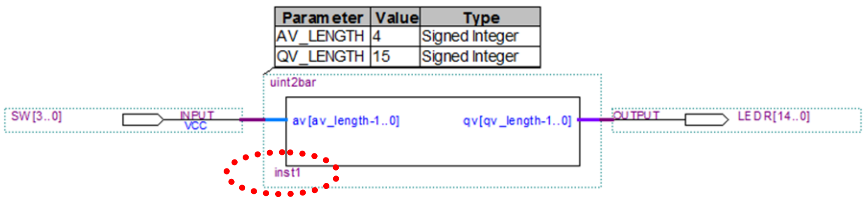
VHDL codes found on the web sometimes use similar abbreviations, but it is advisable to write complete declarations - they generally have fewer portability problems.

L17 – instance of circuit

On line L17

inst1 : uint2bar

we are starting to create an instance of uint2bar, where inst1 is its name, which can be any valid unique VHDL identifier. In the schematic diagram, we see the instance names for each circuit at the bottom left, see Figure 8, and we present its copy below:



Parameter	Value	Type
AV_LENGTH	4	Signed Integer
QV_LENGTH	15	Signed Integer

We can create instances only in the begin-end architecture block, and each must have a unique identifier, preceded by no declaration. It is not an analogy of signal or constant, but it means the insertion of a complete circuit built according to its description in its VHDL file with the same name!

If we want to create additional instances from uint2bar, we can. If we once inserted its component, then we create as many its instances as we need, of course, distinguished by unique identifiers. Moreover, every single instance of the circuit is completely separate and does not share any part of it with another.

Note: In the first approximation, we can consider a circuit as a very distant analogy of creating an instance of a class in object programming, where the class behaves like a template, according to which new statement creates a new object by calling the appropriate constructor.

But the analogy ends here! *The fundamental difference between circuits and classical programs lies in the complete separation of their instances. They do not share any parts. In the case of physical syn-*

thesis, each usage of a circuit means its new complete insertion. The same is also performed for VHDL functions or procedures. Their every evocation does not point to some shared code as in classical programming, but they always refer to separate physical implementations of the entire circuit, that is, to the aforementioned inline expansion.

L18 až L20 - statement map

The instance creation statement that started in line L17 continues with a pair of map sections that define the signals attached to the circuit instance, we can write either named associations => or positional.

Named associations (keyword notation) have syntax:

```
generic map(  generic_name1 => value1,
              generic_name2 => value2,
              ...
              generic_nameN => valueN )
 port map(  port_name1 => signal_name1,
            port_name2 => signal_name2,
            ...
            port_nameN => signal_nameN );
```

Beware -

* we separate members by commas, not semicolons — we write the list of assciations

* no ; after) — the statemant continues.

* we again specify the list of associations separated by commas.

* Only after) we write semicolon — the end of statemant.

where

- If a component has no generic parameters, we do not enter generic map section.
- The designation generic_nameX specifies the identifier of a parameter identical to its declaration in the generic components section, and valueX defines the value assigned to it in the instance being created. If we write valueX by an expression, its result must be known at compile-time – the parameter behaves as a constant inside the whole instance.
- The designation port_nameX specifies an input or output identifier identical to the declaration in the component port section, and signal_name1 determines to which signal is connected.
  - ➢ If we write an association to **port_nameX**, which is the component's input, its value can also be specified by an expression with non-constant value at compile-time, unlike generic parameters, but it is not recommended. Quartus allows the expressions in port map; unfortunately the ModelSim compiler often rejects them for implementation reasons. Because of this, we first assign expressions to auxiliary signals and then we map them to inputs.
  - ➢ **Output** must be certainly always mapped to a signal.
- In the case of associative lists, we can present their members in **any order**, but it is recommended to keep it for clarity.

Pozitional associations (positional notation) have simplified syntax:

```
generic map(  value1, value2,
              ...
              valueN )
port map (  signal_name1, signal_name2,
            ...
            signal_nameN );
```

Naturally, all values in positional associations must appear exactly in the order of the declarations of the individual parameters or inputs and outputs in the component block.

We can freely decide if we use name associations or positional. For simpler components, we chose less verbose positional associations, especially in cases where the declarations of the used components used are embedded in the entity, so we quickly see what we have assigned. For more complex components or components defined in external libraries, we prefer more explicit name associations, which besides offer more excellent immunity to changes in the original entities.

6.1.a Stejné názvy generic parametrů

The previous code would work even if we have named the parameter identifiers in the generic entity section by the same as in the uint2bar component uint2bar:

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_unit2bar is
     generic (AV_LENGTH : integer:=4; QV_LENGTH : integer:=15);
     port (  SW :  in  std_logic_vector(AV_LENGTH-1 downto 0);
             LEDR :  out  std_logic_vector(QV_LENGTH-1 downto 0));
end;
architecture dataflow of demo_unit2bar is
  component uint2bar
     generic (   AV_LENGTH : integer:=4; -- bit length of av input
                 QV_LENGTH : integer:=15); -- bit length of qv output
     port( av : in std_logic_vector(AV_LENGTH-1 downto 0);
           qv : out std_logic_vector(QV_LENGTH-1 downto 0));
end component;
begin
   inst1 : uint2bar
          generic map(AV_LENGTH => AV_LENGTH,
                      QV_LENGTH => QV_LENGTH)   -- no semicolon ; after ) !
          port map(av => SW, qv => LEDR);
end;
```

Although the component has its parameter identifiers and inputs and outputs, we can only reference them in the associations of corresponding map statements, where the compiler searches identifiers on the left side only in the component declaration, while the right side always refers to definitions inside of the VHDL architecture, therefore, it distinguishes proper names. VHDL codes often utilize the same generic identifiers in its entity and used component because they describe pure connections.

Note: Language C alsou utilize the seperation of name spaces of identifiers, see:

```
struct test { int v; };
int main()
{
       test test;
       test.v = 1; test.v++; printf("%d", test.v);
       return 0;
}
```

The C program prints the correct result 2. In the yellow line, it searches the left identifier **test** *in a different namespace than the name of the defined variable* ***test****. We presented the above C program as a demonstration of namespaces, and it is not a hint to the proper creation of identifiers!*

6.1.b @Example XIV. - Priority inhibitor with test and blank inputs

We now expand the previous code with two service inputs, test, and blank, which allow quickly checking, irrespective of the current input state, whether some LED is defective or permanently lit. Connect the test signals to KEY pushbuttons of the DE2 board:

test	blank	
'1'	-	*all LEDR outputs are in '1'*
'0'	'1'	*all LEDRoutputs are in '0'*
'0'	'0'	*circuit behaves as priority inhibitor*

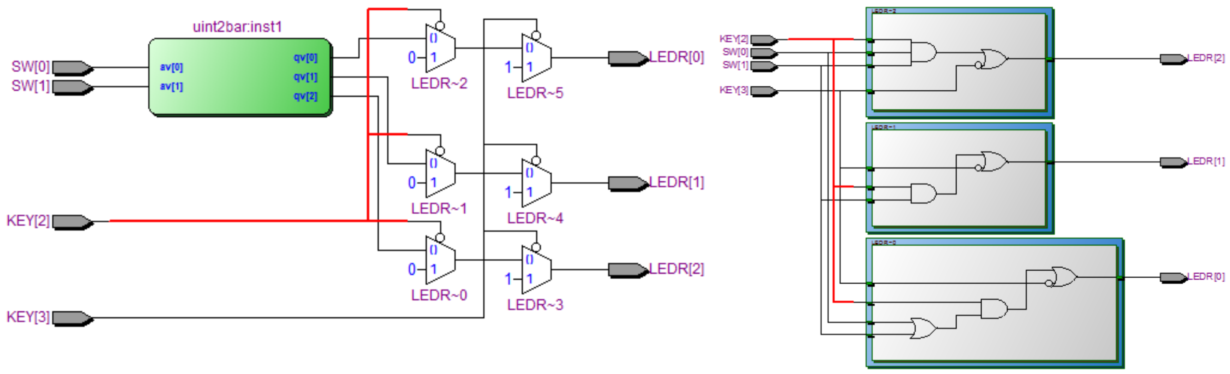
Industrial equipments usually contains similar inputs for checking of outputs.

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_unit2barTest is
  generic ( AV_LENGTH : integer:=2; QV_LENGTH : integer:=3);
  port (      SW :  in  std_logic_vector(AV_LENGTH-1 downto 0);
              KEY : in std_logic_vector(3 downto 2);
              LEDR :  out  std_logic_vector(QV_LENGTH-1 downto 0));
end;
architecture dataflow of demo_unit2barTest is
  component uint2bar
      generic (   AV_LENGTH : integer:=4; -- bit length of av input
                  QV_LENGTH : integer:=15); -- bit length of qv output
      port(   av : in std_logic_vector(AV_LENGTH-1 downto 0);
              qv : out std_logic_vector(QV_LENGTH-1 downto 0));
  end component;
signal tmp : std_logic_vector(LEDR'RANGE);
begin
    LEDR <=  (others=>'1') when KEY(3)='0' else
             (others=>'0') when KEY(2)='0' else tmp;
    inst1 : uint2bar generic map(AV_LENGTH, QV_LENGTH) port map(SW, tmp);
end;
```

We have performed the following edits of demo_unit2bar:

- We saved demo_unit2bar.vhd file as demo_unit2barTest.vhd, and also changed identifies in its entity and architecture accordingly.
- We have reduced the number of inputs in the generic entity section to verify that it works, so we showed that we only needed to change values here, and we do not have to edit its component. Its generic parameters are set in the map section.
- We have added the KEY buttons of the DE2 board to the entity, KEY (3) for testing LEDs and KEY (2) for their blanking.
- Note that the KEY buttons give '0' when pressed, otherwise '1'. For this reason, we tested them on '0', i.e., pressed states.
- We have defined the auxiliary signal tmp, which has a LEDR output range, and we have connected qv output of the component in map port to signal tmp.
- We added when-else statement in architecture with respect to the priority of test over blank function. We wrote it intentionally before creating the instance as proof that the result really does not depend on the order of concurrent statements.
- We have modified associations in map statement to positional ones as an example of this writing style. We can utilize them without any risk within such a small circuit.

The initial RTL Map shows that the compiler has involved a multiplexer cascade behind our uint2bar. A small circle indicates an inverted input. The compiler has further minimized RTL Map to Technology Map that already describes a simple logic circuit. OR gates replace the multiplexer operated by pressing the KEY (3) button. AND gates realized the second multiplexer controlled by KEY(2) push-button, blanking of all diodes.



Structural VHDL allows for faster changes. If we were laboriously to add new functions to the symbolic diagram, we draw it a much longer time.

6.2 *** Practice 6: 7segment decoder with suppressing leading zeros

When displaying a number, we sometimes prefer the suppressing leading 0. For example, instead of the long number "0012", we want to see only " 12", i.e., the first two zeros do not light up.

We can realize it by one additional input, which we mark RBI, ripple blanking input, and output RBO, ripple blanking output.

Číslice na d,c,b, a	d, c, b, a	RBI	RBO	Funkce
0	"0000"	1	1	zhasnutý segment
0	"0000"	0	0	zobrazena 0
X"1" až X"F"	/= "0000"	X, tj. 1 nebo 0	0	zobrazeno číslo 1 až F

Table 8 - 7segment decoder with suppressing leading 0s

If we chain such decoders, we easily create arbitrary long output. We connect '1' to input RBI of higher digits because there is nothing before it. We interconnect the following RBO with RBI. If we always want at least one digit visible, then we connect RBI of the rightmost decoder to '0'.

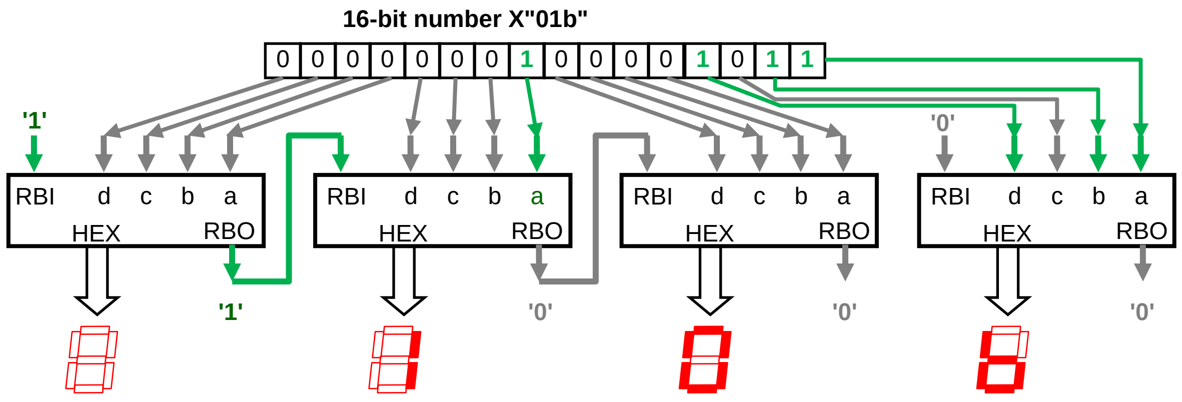


Figure 9 - Čtyři zřetězené 7segmentové dekodéry

Task assignment:

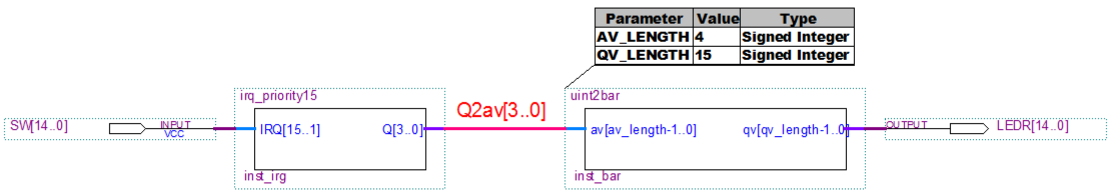
Add RBI and RBO inputs to the previous 7segment decoder, which you have already solved in exercise 3.5 on page 27. Insert this circuit as an instance into the new code.

Instructions: Proceed in the same way as we did with demo_unit2barTest. Now, create only one decoder with input and output for concatenation. We create chaining of 4 decoders for displaying a 16-bit number in the next separate task on page 54. We discuss multiple instances before it.

See the appendix on page 66 for a possible solution.

6.3 @Example XV. - Connection of several circuits

Now we try to connect two circuits by VHDL structural style. We employ our priority interrupt circuit, and we display its output on our bar indicator. The result looks like the following:

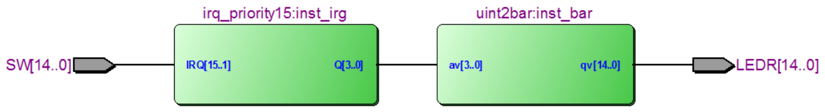


There is a new element in the schema, the named wire, highlighted in red, which we created in the Quartus schema editor. We selected the wire, and in its context menu, we chose Properties and typed our name along with the range in editor notation. If we do not name the wire in the schema, the compiler assigns it an automatically generated name during the conversion of our schema. Here we have done this as a reference to the VHDL code in which we must always define the wires for all connections between the circuits.

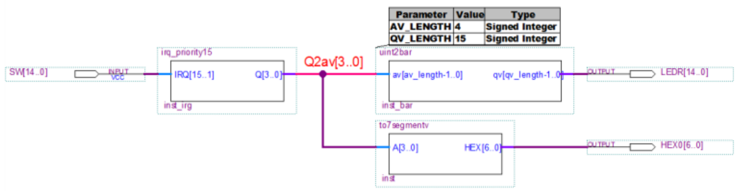
```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_irq_bar is  port ( SW :  in  std_logic_vector(14 downto 0);
                               LEDR :  out  std_logic_vector(14 downto 0));
end;
architecture dataflow of demo_irq_bar is
  component uint2bar
       generic ( AV_LENGTH : integer:=4; QV_LENGTH : integer:=15);
       port(   av : in std_logic_vector(AV_LENGTH-1 downto 0);
               qv : out std_logic_vector(QV_LENGTH-1 downto 0));
  end component;
  component irq_priority15 is
       port (  IRQ : in std_logic_vector(15 downto 1);
               Q : out std_logic_vector(3 downto 0) );
  end component;
  signal Q2av :  std_logic_vector(3 downto 0);
begin
  inst_bar : uint2bar generic map(Q2av'LENGTH, LEDR'LENGTH) port map( Q2av, LEDR );
  inst_irg : irq_priority15 port map( IRQ=>SW, Q=>Q2av );
end architecture;
```

- In the definition of the auxiliary signal **Q2av,** we have to write the whole range because we cannot refer to the output **Q** of the component irq_priority15. Its members ca appear only on the left side of map associations. We could solve this difficulty by defining the subtype in a package, which we discuss in chapter 7 on page 55.
- When creating an inst_bar instance, we could omit its generic map section because we set lengths that match their default values. However, we decided to insert also the generic map and include vector-derived lengths to make the code more readable.

RTL Map shows us the circuit that we wanted. This time, we do not show the inside of circuits because we have already shown them in previous chapters. If we want to look at them here too, we can select the circuits of interest and enter the context menu Display Content/Hide Content.



6.4 *** Practice 7: Adding our 7-segment display



Task: Extend the previous VHDL code demo_irq_bar by a 7-segment display, as shown in the figure. If you only have a 7-segment decoder with separate inputs a0, a1, a2, and a3, write association to them in port map.

We find a possible solution on page 67.

Note: You could also make another 7-segment decoder with vector input av (3 downto *0), called to7segmentv, for example, and insert it.*

For its creation, however, use the original circuit to7segment *with separate inputs, and place it into* to7segmentv *as a component. From it, create an instance whose inputs you map to* Q2av *vector. Do not do* to7segmentv.vhd *by copying the entire to7segment code with the intention of its editing to vector input. This way represents a wrong style, which duplicates possible mistakes. If you insert an instance, then you refer to its existing code, which remains in one place, and its possible corrections are automatically reflected in other parts.*

6.5 @Example XVI. - More instances of majority

Assignment: Suppose we have a vector containing the outputs of 6 triplets of detectors, possibly from light barriers. So we have 18 inputs, with inputs with indexes 0 to 2 belonging to the first three detectors, indexes 3 to 5 to the second three, and so on, up to the last sixth triad with indexes 15 to 17.

We wish to process each of the three detectors with the majority function, that is, at least two detectors must be in '1' at a time. For example, we could utilize the decision based on the majority function when the disruption of one beam can cause a small animal or insect, while only a more massive object cuts two beams at once.

We do not write 6 logical equations, but we insert instances of our majority. We write the code for DE2 board. Switches SW(17 downto 0) stands for detectors, and we display the obtained result on green LEDs LEDG(5 downto 0).

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_majorita6g is
     port (  SW :  in  std_logic_vector(17 downto 0);
             LEDG :  out  std_logic_vector(5 downto 0));
end;
architecture dataflow of demo_majorita6g is
component majorita is port( a, b, c : in std_logic; y : out std_logic);
end component;
begin
     inst1 : majorita port map(a=>SW(0), b=>SW(1), c=>SW(2), y=>LEDG(0));
     inst2 : majorita port map(SW(3), SW(4), SW(5), LEDG(1));
     inst3 : majorita port map(SW(6), SW(7), SW(8), LEDG(2));
     inst4 : majorita port map(SW(9), SW(10), SW(11), LEDG(3));
     inst5 : majorita port map(SW(12), SW(13), SW(14), LEDG(4));
     inst6 : majorita port map(SW(15), SW(16), SW(17), LEDG(5));
end;
```

Figure 10 - Majority 6 groups with 3 elements

The resulting code shows the multiple usages of the instance creation command. In the first line, we used name associations for the demonstration, while in the following, we used positional associations.

And we created the circuit expressly; we wouldn't draw the scheme so vividly. We left it to compilers.

The initial diagram of RTL Map can be seen on the right, while for the upper majority, we have also shown its internal involvement. Others hidden in blocks have precisely the same structure.

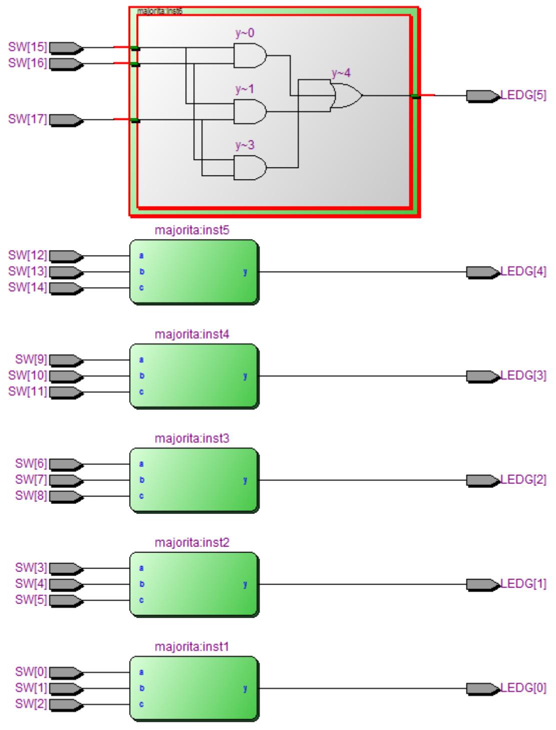


Figure 11 - RTL Map šesti majorit

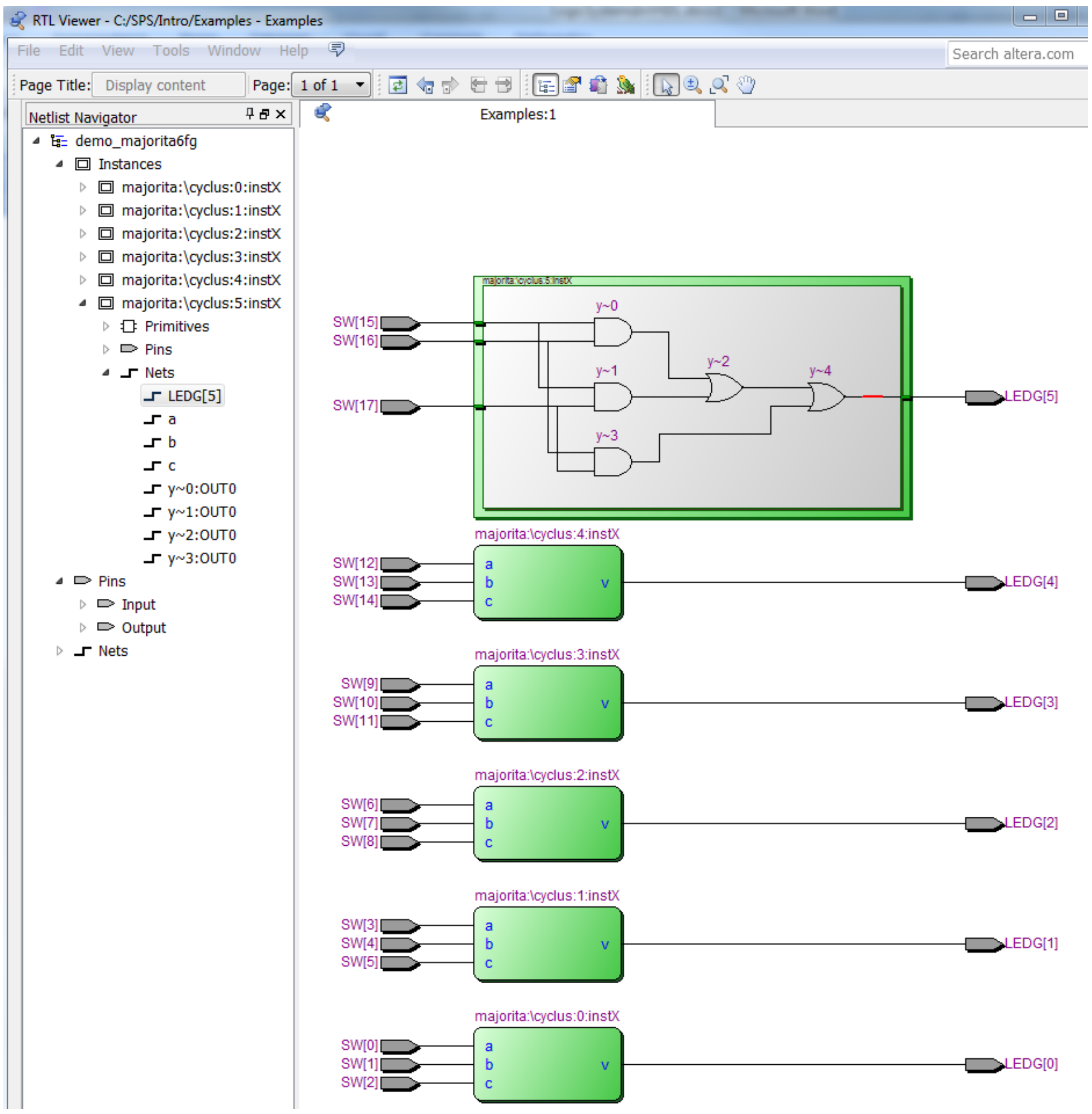
6.6 @Example XVII. - Vektor majority s for generate

Does the question arise as to whether in the previous example you could use VHDL for-generate command? Let's try it. We create a new VHDL file demo_majorita6fg.vhd and write the code:

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_majorita6fg is     port (  SW :  in  std_logic_vector(17 downto 0);
                                       LEDG :  out  std_logic_vector(5 downto 0));
end;
architecture dataflow of demo_majorita6fg is
component majorita is     port( a, b, c : in std_logic; y : out std_logic);
end component;
begin
   cyclus: for i in 0 to 5 generate
      instX : majorita port map(a=>SW(3*i), b=>SW(3*i+1), c=>SW(3*i+2), y=>LEDG(i));
   end generate;
end architecture;  -- for clarity of our code, we have emphasized that this end belongs to architecture
```

Kód 1 - Tvorba instance ve for generate

The circuit description generates the same simple RTL Map as without the for-generate statement. Although instX is repeatedly inserted during translation, Quartus replaced it with automatically generated identifiers from "\cyclus:0:instX" to "\cyclus:5:instX".



6.6.a Misuse of the auxiliary variable

Inside the previous code, the expressions 3*i, 3*i+1 a 3*i+2 arise the illusion that it would make it advantageous to introduce an auxiliary variable to save triple multiplication. In the C program, it might be considered a good style. In VHDL, however, an analogous operation is often **reversed in disaster**.

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_majorita6fg_wrong is
  port ( SW : in  std_logic_vector(17 downto 0); LEDG :  out  std_logic_vector(5 downto 0));
end;
architecture dataflow of demo_majorita6fg_wrong is
component majorita is   port( a, b, c : in std_logic; y : out std_logic);
end component;
begin
      cyclus: for i in 0 to 5 generate
          signal n : integer range 0 to 15; -- the range hint for compiler to simplify RTL Map,
      begin
          n<=3 * i;
          instX : majorita port map(a=>SW(n), b=>SW(n+1), c=>SW(n+2), y=>LEDG(i));
      end generate;
end architecture;  -- for clarity of our code, we have emphasized that this end belongs to architecture
```

Figure 12 - Instance komponenty ve for - generate

The code does not annouces compilation erros, and yet it si not good.

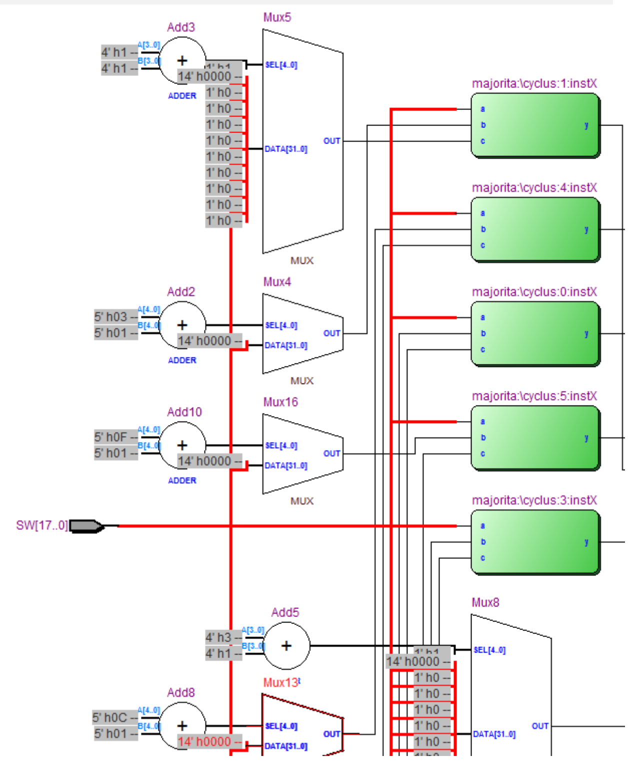
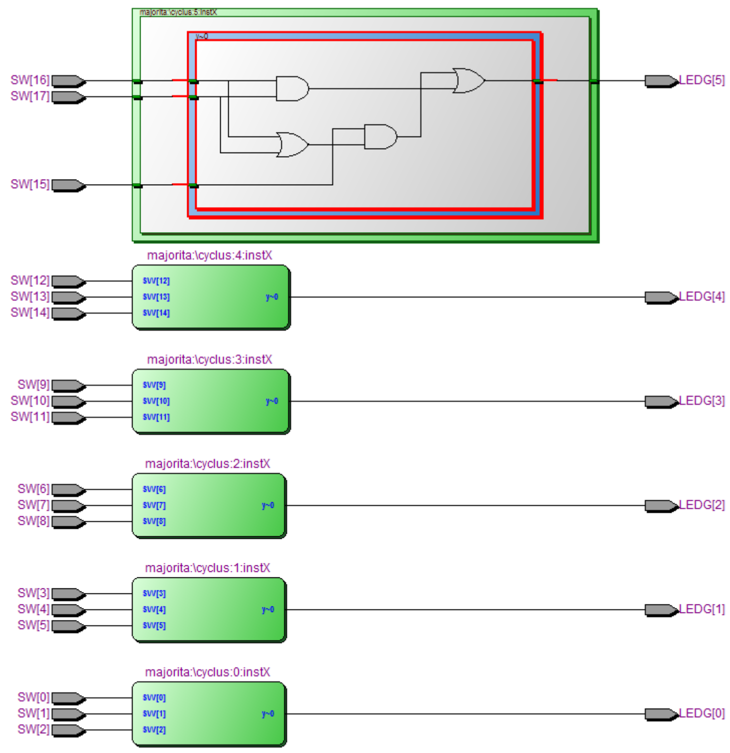
Part **RTL Map** - *too big, something is wrong:-(*	**Technology Map** - *Fortunately, it is minimal, OK*
	

Table 9 - Repeated instance with the aid for-generate

In RTL Map, we see that the compiler replaced individual multiplications of 3*i by constants 0, 3, 6, to 15, since **i** parameter is read-only inside the loop, which is compiled in the synthesis code always with the inline expansion style. The adders perform additions, and 32-input multiplexers stand for indexing of SW elements. Unused multiplexer inputs were connected with '0' (shown in the figure as 1'h0).

We provided the compiler with hints that the intermediate variable n within the cycle is in range 0 to 15. Although compiling would work fine without this specification, RTL Map would look even more complicated than now, since the intermediate variable n would in it modeled 32-bit signed integer.

Repeatedly used instX instance labels inside the loop were replaced with internal names from \cyklus:1:instX to \cyklus:6:instX and temporary variables solved multiple assignments of n in six inserted assignments n<=3*i; Here, too, the compiler has already applied behavioral compilation and replaced repetitive entries using automatically generated identifiers.

Fortunately, we deliberately assigned to n only cycle-dependent value of **i**, so that after minimizing the wiring, we obtained proper circuit as Technology Map shows. It is exactly the same it as in the previous code without for-generate.

But why was RTL Map so complicated? We hid before the compiler that we process separate triads. Although we have all the inputs in one vector, the compiler did not find their binding. When modeling the opening code as RTL Map, it found that the indices on the SW vector members depend on n signal, whose value is set elsewhere, and it inserted 32-input multiplexers to select one of the 18 elements.

Note: We have to realize here that we humans know that n *is used only in the* port map *for three members derived from* 3*i *because we see the full code. The compiler only tracks a fraction of it. The situation is analogous to image processing.We see the whole image at once, but the algorithm processes only a small fragment at a time. Of course, heuristics could be added to Quartus to detect similar situations. There are many in it, but not for every possible case, as an increase in their number would significantly increase translation time. We have to tell the compiler ourselves*

If we see the RTL Map more complicated than the expected structure of our designed circuit, then our code has the wrong structure.

Objection: And what if we have no idea what the result should look like? Well, then we start the design from its wrong side. It is better first to think about how the circuit should work and then write its VHDL code. In doing so, we check that the result at least somewhat corresponds to our initial idea.

The complicated RTL Map is not a tragedy until the complexity of the resulting circuit does not grow too much, and the compilation time increases only by a few seconds or even milliseconds. However, if we find something more complicated, we need to verify that the compiler still manages to minimize our design. Unfortunately, looking at the Technology Map is only useful for smaller circuits that remain clear. For the more complicated ones, it does not give us much advice. There, we analyze the delay times between input and output and the number of used elements.

Note: We should emphasize here that ***the definition of an auxiliary signal or variable is not automatically a wrong step****. In VHDL codes, this often makes the resulting circuit much smaller but certainly not in all cases! There are situations, such as the one above, when similar attempts turn the opposite. How to recognize them? Either based on experience or always from RTL Map.*

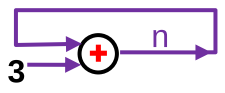
6.6.b Fatal error: Reassign a signal in for-generate

We should write for-generate very carefully. In the following "suicidal" VHDL, we increase n by 3 in the for-generate loop. In the classic program, it is a normal operation, here, it is a liquidation construction:-(

Quartus does not announce any errors; it only issues warnings that the result is a lousy circuit with a tendency to oscillate. Increasing n by 3 would not matter in the C program, but here we connect the circuit!

```
begin -- architecture
    cyclus: for i in 0 to 5 generate
      signal n : integer range 0 to 15:=0; -- assigning initial value,
    begin
      n <= n+3;  -- fatal operation not only inside for-generate, but in many other concurrent parts
      instX : majorita port map(a=>SW(n), b=>SW(n+1), c=>SW(n+2), y=>LEDG(i));
    end generate;
end architecture;  -- for clarity of our code, we have emphasized that this end belongs to architecture
```

The for-generate cycle **must not use** any value created in the previous cycle passes! Without detection of the rising edge of the clock signal, there is no memory element usable in FPGA circuits. The compiler must insert strictly prohibited unstable combination loops. We explain them in more detail in the lectures.



Conclusion: We have demonstrated that we cannot blindly imitate techniques for writing classical programs in VHDL codes designated for physical circuit synthesis.

6.7 @Example XVIII. - Statement for generate-for separated inputs

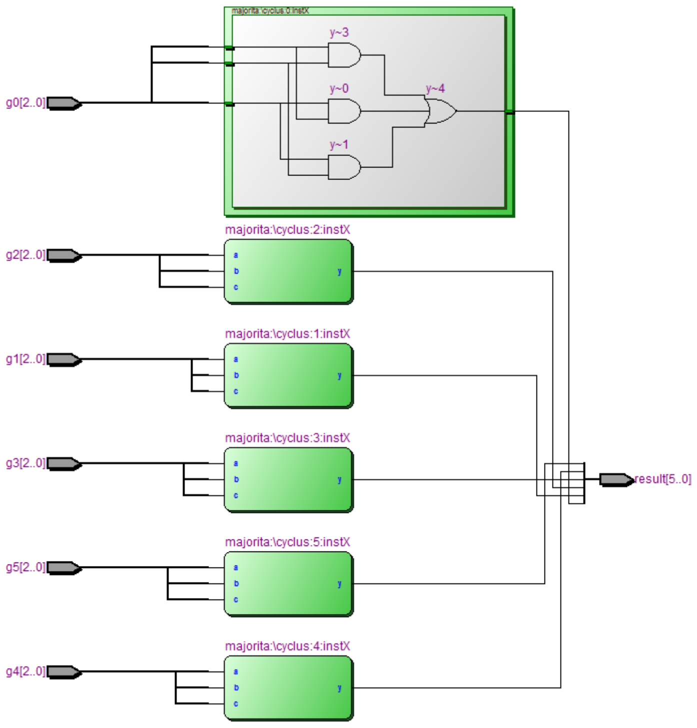
VHDL does not know how to generate identifiers dynamically.

If the three sensors in the port section were in separate inputs with different names, we wouldn't build the generate loop so quickly because we cannot build the identifier as a string and then use it. Some languages can do so, like Matlab, but not VHDL. Indeed, even the C language does not allow such tricks, and its codes solve similar needs with the aid of proper dictionaries.

In VHDL, we can employ an initialized field:

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity majorita6ga is
     port (  g0, g1, g2, g3, g4, g5 :  in  std_logic_vector(2 downto 0);
             result :  out  std_logic_vector(5 downto 0));
end;
architecture dataflow of majorita6ga is
component majorita is   port( a, b, c : in std_logic; y : out std_logic);
end component;
   type gv_t is array (0 to 5) of std_logic_vector(g0'RANGE);
   signal gv : gv_t := (g0, g1, g2, g3, g4, g5); -- initialized array
begin
     cyclus: for i in 0 to 5 generate
        instX : majorita port map(a=>gv(i)(0), b=>gv(i)(1), c=>gv(i)(2), y=>result(i));
     end generate;
end architecture;  --we have emphasized that this end belongs to architecture
```

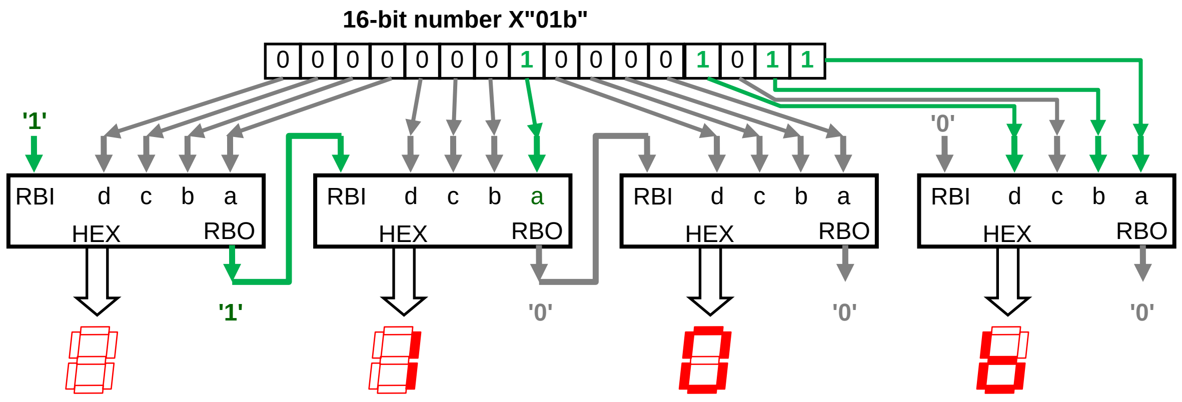
The resulting code also has a very clear RTL Map, for unambiguous description and indexes derived from the generete cycle parameter.



Initialized array is sometimes useful for other larger codes, where it can simplify more complicated constructions where it is possible to utilize for generate.

6.8 *** Practice 8: Showing 16-bit number on 7-segment display

You have already built a 7-segment display circuit with RBI input and an RBO output. Try to use it now, as shown Figure 9 on page 46. You can see its copy below.



Assignment: Write the VHDL code of circuit that takes the binary number set on SW(15 downto 0) switches of the DE2 and displays it hexadecimally on four 7-segment displays:

HEX0, HEX1, HEX2, HEX3 : std_logic_vector(6 downto 0);

where HEX0 designates the rightmost display, i.e., with a lower weight, while HEX3 has the highest weight.

* Realize the circuit with suppressing the initial 0.

* Take advantage of instantiating from an existing 7-segment decoder circuit with RBI and RBO, and define the interconnector signals that you use in each port map. The for-generate not worth it here because the instances have a different I/O structure.

* Check the result in RTL Map.

* Alternatively, you can create a variant in which suppressing leading zeros is turned on or off by another signal. *Hint: This is a straightforward change whose implementation requires more thought than writing. All you need is a little detail, a work for few seconds. Try to figure it out yourself.*

In the appendix on page 68, you can compare your both solutions with possible realizations.

6.9 *** Practice 9: Priority inhibitor for three groups

In exercise 5.3 on page 37, you designed the universal priority inhibitor. Use it for processing of group, where only one signal of six is allowed.

Suppose again that you have the input from SW(17 downto 0) switches.

Assignment: SW switches (5 downto 0) form the first group, of which we want only signal in '1' that has the highest priority send to the corresponding output. The remaining SW groups (11 downto 6) and SW (17 downto 12) are also processed analogously.

* Connect the output of your circuit to the red LEDs, LEDR(17 downto 0).

* Use the for-generate this time to insert a priority inhibitor instance three times.

* Check in the RTL Map if you obtained proper circuit.

* You can also try a variant where 6 triplets are handled in the same way, thus, at most only one signal is passed in '1', i.e., to SW (2 downto 0) forms the first group, SW (5 downto 3) the second, etc. And could you quickly adapt the circuit in 10 seconds at most? Yes, when using for-generate. Try it.

See the appendix on page 70 for possible solutions to both tasks.

7 @Example XIX. - Creating our own library ("package")

Inserting component declarations increases the security of the code, but as their number increases, the clarity of the code decreases. C language simplifies the situation by #include preprocessor directives, and C # by the using statement. In Java, we can link the created package with the import command.

VHDL applies the same concept called the package that represents significant advantage VHDL over Verilog, which does not know anything like it.

We create packages by inserting the necessary declarations and definitions. The basic syntax of the package below contains the main elements that may appear in it during the synthesis. Yellow highlights indicate the parts that we already know. Others, except for the alias, belong to the VHDL behavioral style:

```
<library statement> <use clauses>
package  package_name  is
    <components>
    <constant full definitions with assigned values>
    <constants deferred - no assigned values>
    <type and subtype declarations>
    <aliases definitions of alternatives names for existing objects>
    <attribute declarations and specifications>
    <functions headers>
    <procedure headers>
end  [package]  [package_name];

package body  package_name  is
    <assigned values for deferred constants>
    <other declarations internally used inside body>
    <complete definitions of functions>
    <complete definitions of procedures>
end  [package body]  [package_name];
```

The package has two parts, something like dividing the circuit into entity and architecture. Introductory part

```
package package_name is
end [package]  [package_name];
```

declares elements visible from the outside. Its name package_name represents a valid VHDL identifier we assign, and the terminating end can be written similarly to an entity, see Table 3 on page 13. We can only wite end or add more details in case of longer blocks:

```
package body package_name is
end [package body] [package_name];
```

performs full definitions a the possibility mention above hold also for its end.

Again, we keep the rule of triple naming. If we create a package and name it lsp_codes, then

1. we save the file as lsp_codes.vhd,
2. we write the same name also in the interface part: package lsp_codes is
3. and we do not forget to place the same name in the body:
   package body lsp_codes is

We show the definition of such package, where we insert only a single component so that the text and the following example of the package can fit on one page:

```
-- File lsp_codes.vhd
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;                    --L01
package lsp_codes is                                                                    --L02
   constant LEDR_LEN:integer:=18;                                                       --L03
   constant LEDG_LEN:integer;  -- deferred constant                                     --L04
   subtype slv_ledr  is  std_logic_vector(LEDR_LEN-1 downto 0);                         --L05
   component majorita is port( a, b, c : in std_logic; y : out std_logic);              --L06
   end component;                                                                       --L07
   alias LG : integer is LEDG_LEN;                                                      --L08
end; -- or  also e.g.  end package;                                                     --L09
package body lsp_codes is                                                               --L10
   constant LEDG_LEN:integer:=9; -- assigning value to deferred constant                --L11
end;  -- or  also e.g.  end package body;                                               --L12
-- File  demo_majorita6fg_package.vhd
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;              --R01
use work.lsp_codes.all;                                                                 --R02
entity demo_majorita6fg_package is                                                      --R03
   port ( SW :  in  slv_ledr; LEDG :  out  std_logic_vector(LG-1 downto 0));            --R04
end;                                                                                    --R05
architecture dataflow of demo_majorita6fg_package is                                    --R06
begin                                                                                   --R07
   cyclus:  for i in 0 to 5 generate                                                    --R08
            instX : majorita port map(a=>SW(3*i), b=>SW(3*i+1), c=>SW(3*i+2), y=>LEDG(i));  --R09
            end generate;                                                               --R10
   LEDG(LEDG_LEN-1 downto 6)<="100"; --equal to LEDG(LEDG'HIGH downto 6)<="100";        --R11
end architecture;  -- for clarity, we have emphasized that this end belongs to architecture  --R12
```

L03 and L04 - constants

In a package, we can define constants, either complete, including their initializations, or make only a declaration of constant, which means giving identifier and type, but without a value. The second case is known as deferred constant. We use it when we calculate the constant value by a more complex expression, for example as deriving the value from other constants or in case of creating an initialized array, so that a complicated expression with many members does not reduce the readability of the header. The definition is postponed to the package body.

We **must not use deferred constant** in the package header declarations which needs its value, since the constant gain it in the body after its definition. We cannot use our deferred LEDG_LEN constant in the header either for the subtype or for the component, because the constant has no known value yet.

> *Note: VHDL is not an object programming in which definitions within an entire class are valid from its very beginning. VHDL is more similar to non-object C, where anything exists only after executing the corresponding command. Our deferred constant gets its value in line L11.*

L05 - subtype

We defined the subtype **slv_ledr** and used **LEDR_LEN** constant that already has its full definition. We could then apply our new subtype to other package header declaration if needed.

Packages often contain subtype definitions, especially for larger sub-circuit assemblies. We are referring to package definitions in all sub-entities and derived all necessary elements with the aid of the package.

For example, if you build a processor, we design a subtype for its address data bus. If we want to increase/decrease its width, we change it only in one place — we fix only the subtype in the package.

L06 - component

Here we insert a single component, but we could have included more of them, perhaps from all the files we created.

L08 - alias

The alias statement does not create a new object but only gives an alternative name to an existing one, which must be only elements of constant, signal, or variable (that is an element of behavioral style). Therefore, it is not possible to give another name to a type, or a component, for example. We assigned a nickname to the deferred constant LEDG_LEN, which we could, because the alias does not use its unassigned value.

L11 - definition of deferred constant

The constant definition is the only statement in the package body that we know so far. All others possible belong to behavioral style, as functions and procedures.

R01 - libraries in file that uses a package

In the file where we want to reference the package, we must also include all the libraries that we use in it. The compiler does not transfer package libraries; only the package worked with them. The compiler loads only the elements specified in the package header; in other words, it stores their resulting definitions in internal tables as pairs composed of the identifier and its definition.

Introduction note: VHDL does not use a direct analogy to the C #include commands — the C preprocessor inserts their complete code. VHDL treats libraries analogously to Java import *command or C#* using *directive employed for importing from another C# namespace. It inserts the result, but not the sub-elements, that were used inside a package to create definitions and declarations.*

R02 - using package

We insert our package lsp_codes as libraries std_logic_1164 and numeric_std; even these are basically packages. Since we have lsp_codes.vhd in the Quartus project directory, we specify that the package belongs to work, see the description of its meaning on page 41. So we write use work.lsp_codes.all;

R04 - using subtypes and constants from package

Once we embedded the package in our code, then its compiled, and all deferred constants get their values. We can also use the package definitions in entity ports to obtain a highly variable code.

R09 - instances of components in package

We did not insert any component definition this time; we don't have to, we already have them in the package. We only create their instances by map statements.

Of course, we must have somewhere files that define component architectures, either directly among the project files, or somewhere along the path defined by the library directory list that we specified in the Quartus menu: Tools->Options->General->Libraries: Project Libraries.

Note: If we inserted component of our majorita *circuit between lines R06 and R07, that is, between architecture and its begin, nothing would happen if it were identical to the package. However, if it were different, even in the name of input or output, the inserted component became decisive for our code.*

R11 - usage of constant

We've added a line with LEDG to demonstrate that we can refer to both the LEDG_LEN constant and its LG alias. Both identifiers refer to the same object.

7.2 @Example XX. - Distribution - all in one

If we place everything is in a single file, we create confusing code, so it discourages style. However, it is useful for distributing the final debugged code. Quartus reads the whole file in one stream and creates all the definitions therein. S=We can save the example from the previous chapter into one file with the content:

```
-----------------------------------------------------------------------------------------
-- Following text was saved as  demo_majorita6fg_distribution.vhd
      -----------------------------------------------------------------------------------------
     library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
     package lsp_codesL is
        constant LEDR_LEN:integer:=18;
        constant LEDG_LEN:integer;  -- deferred constant

        subtype slv_ledr  is  std_logic_vector(LEDR_LEN-1 downto 0);

        component majoritaL is port( a, b, c : in std_logic; y : out std_logic); end component;
        alias LG : integer is LEDG_LEN;
     end; -- or  also e.g.  end package;

     package body lsp_codesL is
        constant LEDG_LEN:integer:=9; -- assigning value to deferred constant
     end;  -- or  also e.g.  end package body;
     -----------------------------------------------------------------------------------------
     library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
     use work.lsp_codesL.all;
     entity demo_majorita6fg_library is
        port ( SW :  in  slv_ledr; LEDG :  out  std_logic_vector(LG-1 downto 0));
     end;
     architecture dataflow of demo_majorita6fg_library is
     begin
        cyclus: for i in 0 to 5 generate
                 instX : majoritaL port map(a=>SW(3*i), b=>SW(3*i+1), c=>SW(3*i+2), y=>LEDG(i));
                 end generate;
        LEDG(LEDG_LEN-1 downto 6)<="100"; --equal to LEDG(LEDG'HIGH downto 6)<="100";
     end architecture;  -- for clarity, we have emphasized that this end belongs to architecture
     -----------------------------------------------------------------------------------------
     library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
     entity majoritaL is port ( a, b, c : in std_logic; y : out std_logic ); end;
     architecture dataflow of majoritaL is
     begin y <= (a AND b) OR (a AND c) OR (b AND c);
     end;
-----------------------------------------------------------------------------------------
```

We renamed all entities so that they do not interfere with existing files in the project. We take into account their correct order, because objects must already exist before using them. We started with a package whose definitions apply in the following sections. We have placed the block majorityitaL at the end because the package has already inserted its header so we can reference its before its processing.

To translate a file, we make it a Top-Level entity. Alternatively, we could turn it into a package that we would use from another Top-Level entity, but in that case we must save under the package name, i.e. as lsp_codesL.vhd; otherwise, it might not be found.

Notice the repeated insertions of libraries! With the end of package body or end of architecture block, the previous library and use definitions become invalid, so they must be reloaded. Therefore, joining multiple files is easier because previously imported packages do not interfere with the following parts.

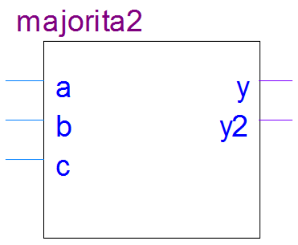
8 Appendix A: Solutions of Practices

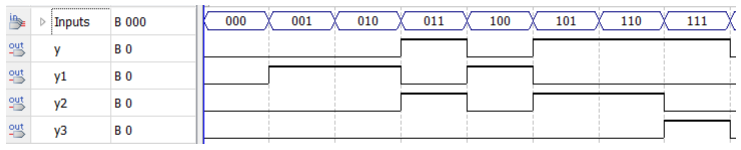
8.1 Practice 1: Majorita123

There is one possible solution to the task introduced on page 10:

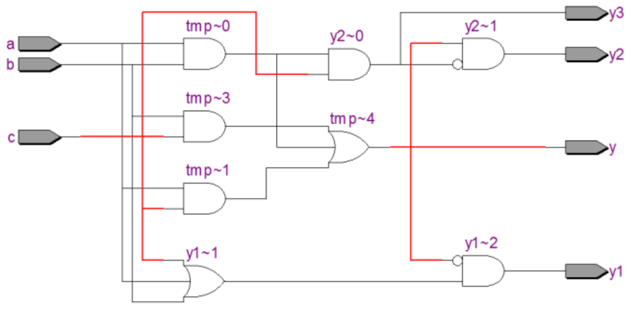
VHDL code majorita123.vhd **Schematic symbol**

```
-- Majorita with signaling 1 and 3 inputs in 1
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity majorita123 is
        port (   a, b, c : in std_logic;
                 y, y1, y2, y3 : out std_logic );
end;
architecture dataflow of majorita123 is
signal tmp : std_logic;
begin
        tmp <= (a AND b) OR (a AND c) OR (b AND c);
        y<= tmp; y2 <= tmp AND NOT (a AND b AND c);
        y1<=(a OR b OR c) AND not tmp;
        y3<=a AND b AND c;
end;
```

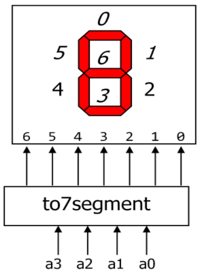




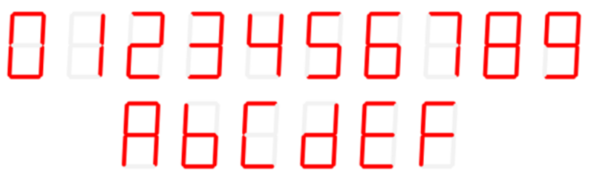
```
Inputs  B 000   000  001  010  011  100  101  110  111
y       B 0
y1      B 0
y2      B 0
y3      B 0
```



8.2 Practice 2: Decoder for 7-segment display



We can solve the task from page 27 by an array in which we store the segment values for hexadecimal numbers. We are writing for the DE2 development board, where the segment is on when the corresponding output is in '0'.

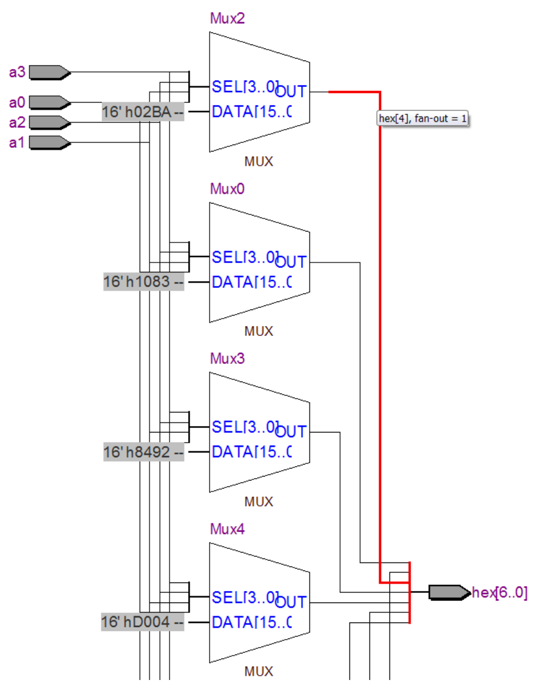


We join inputs a3, a2, a1, and a0 by the operator & with type-hint of the target data std_logic_vector as an unsigned number. We convert it to an integer and use as an index into hexarray.

```
-- Display hexadecimal number on 7segment display
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity to7segment is
    port(   a0, a1, a2, a3 :in std_logic; -- a3.. a0 input hexadecimal number a0-lsb  a3-msb
            hex: out std_logic_vector(6 downto 0) ); -- hex[6..0] output to 7 segment display
end;
architecture dataflow of to7segment is
  type hexarray_t is array(0 to 15) of std_logic_vector(hex'RANGE);
  constant hexarray : hexarray_t :=
     ("1000000","1111001","0100100","0110000", "0011001","0010010","0000010","1111000",
      "0000000","0010000","0001000","0000011","1000110","0100001","0000110","0001110");
begin
        hex<= hexarray(to_integer(unsigned(std_logic_vector'(a3 & a2 & a1 & a0))));
end;
```

Note: If you encoded '1' as a segment lighting, it does not matter, the NOT operator will resolve the conversion we can also applied it to a vector, i.e,, to a result selected by an index:

```
        hex<= NOT hexarray(to_integer(unsigned(std_logic_vector'(a3 & a2 & a1 & a0))));
```



We present a part of the RTL Viewer schema on the right. Quartus converted the array first into 16-input multiplexers for hex output bits written abbreviated as hex[6..0], i.e. as hex(6 downto 0). The multiplexers have address inputs, which are designated SEL [3..0], i.e., SEL(3 downto 0), to which the address wires a3, a2, a1 and a0 are connected.

Each multiplexer has 16 DATA inputs [15..0] with values written as brief constants. For example, 16'h02BA for the Mux2 multiplexer is 16 bits X"02BA", i.e., "0000001010111010", which determines the hex segment (4) illumination in hexadecimal digits F, E, d, C, b, A, 8, 6, 2 and 0.

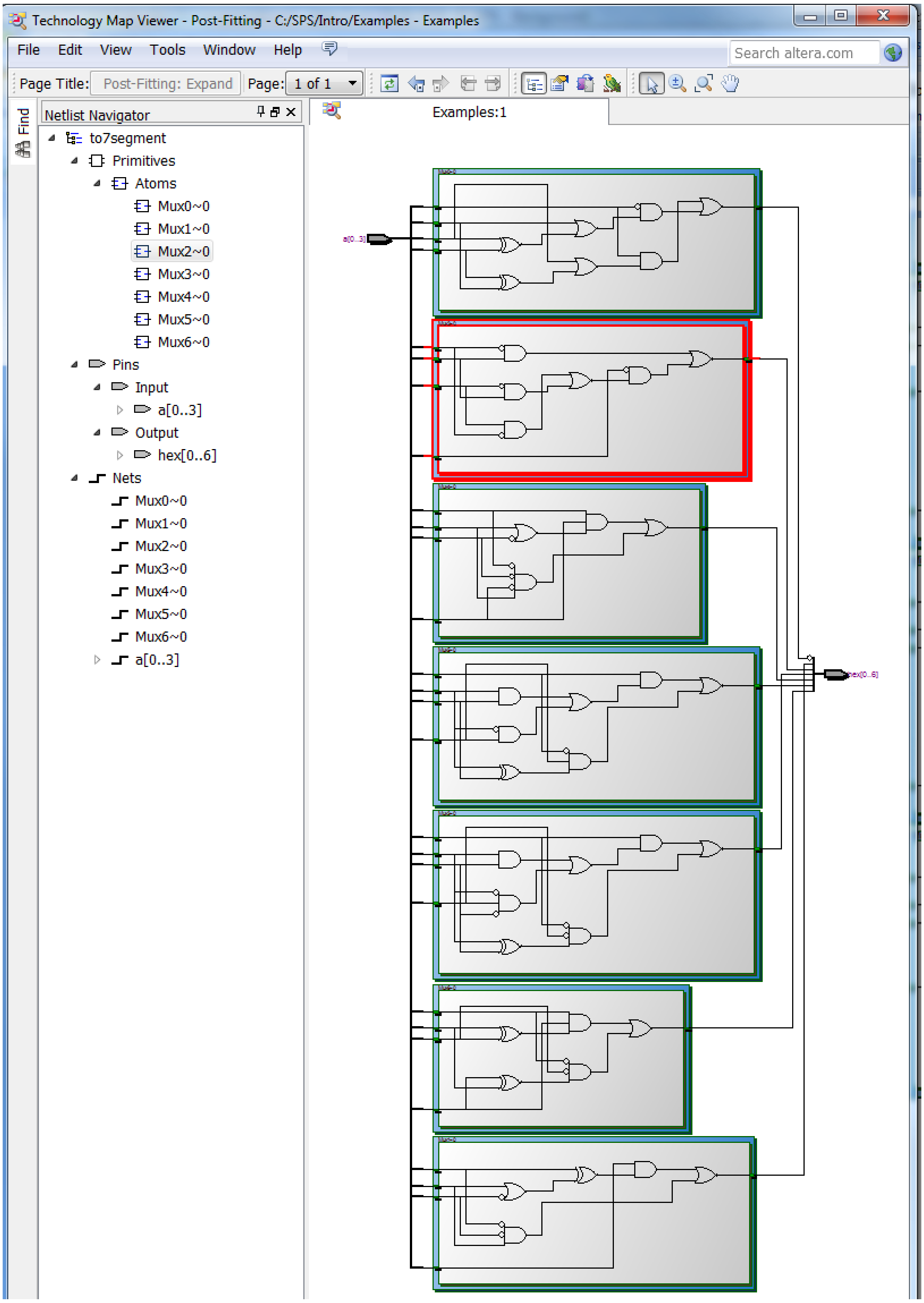
The fact that Mux2 leads to the hex (4) segment can be found, for example, by hovering the mouse on the wire. Mux# multiplexer names were automatically generated during translation and generally did not relate to the output numbers.

The context help of the Mux2 output has the form

hex [4], fan-out = 1"

The first hex [4] indicates where the selected wire leads. The number behind the fan-out specifies how many elements are connected by a wire, i.e., the output load; here only one.

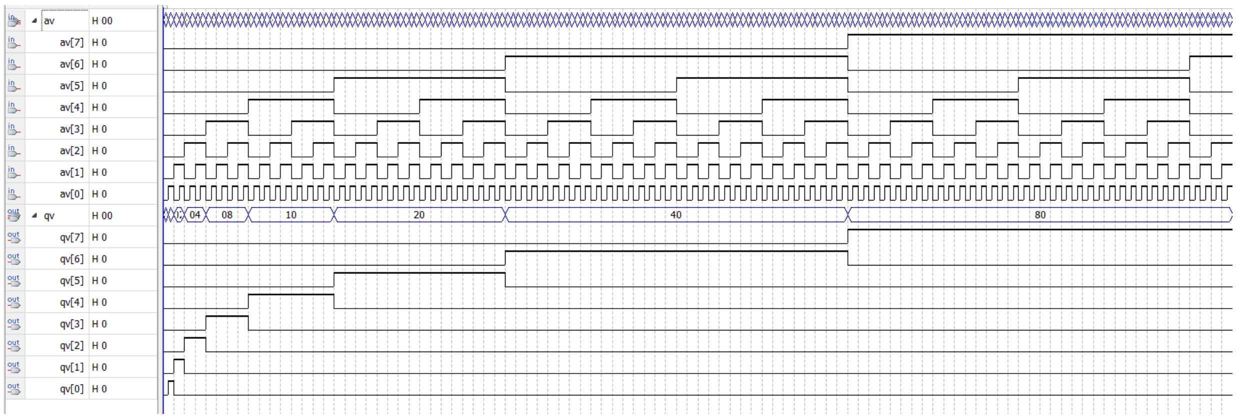
Technology Map Viewer shows the actual implementation where we can again find Mux2. We see that Quartus did not try group minimization. It had no reason to do so, because one logic element in the Cyclon II circuit can realize just one 4-input logic function, precisely as one segment needs.



8.3 Practice 3: Prioritní inhibitor

We can solve the task from page 31 by modifying the example from chapter 4 on page 28.

```
--Priority inhibitor 8 inputs
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity prioritni_inhibitor8 is
        port (  av : in std_logic_vector(7 downto 0);
                qv : out std_logic_vector(7 downto 0));
end;
architecture dataflow of prioritni_inhibitor8 is
begin
qv <=  X"80" when av(7)='1' else
       X"40" when av(6)='1' else
       X"20" when av(5)='1' else
       X"10" when av(4)='1' else
       X"08" when av(3)='1' else
       X"04" when av(2)='1' else
       X"02" when av(1)='1' else
       X"01" when av(0)='1' else
       X"00";
end;
```



8.4 Practice 4: Universal priority coder

In the practice on page 37, we again utilize our template.

If we want a variable length, then we have to compose the output by individual signals. We create an auxiliary signal inh (*inhibit*) that transmits information about activation of some priority output to '1' so it has the range equal to the output qv. If we accept the rule that a higher index has a higher priority, then:

```
qv(qv'HIGH)<=av(qv'HIGH); inh(qv'HIGH)<=av(qv'HIGH);
```

Outputs with less priority, we build for-loop with i parameter in range Q'HIGH-1 downto 0:

```
qv(i) <= NOT  inh(i+1) AND av(i);
```

We activate qv(i), if its corresponding input is '1' and no higher priority output is in '1'. We propagate activation information in inh vector. If its higher member was in '1' or corresponding input is '1', this item is in '1', which propagates to all lower membres and sets them to '1'.

```
inh(i) <= inh(i+1) OR av(i);
```

```
--Priorityi inhinitor
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity prioritni_inhibitor is
        generic( N: natural := 18);  -- delka vektoru > 1
        port (  av : in std_logic_vector(N-1 downto 0);
                qv : out std_logic_vector(N-1 downto 0) );
begin
        assert N>1 report "required N>1" severity failure;
end;
architecture dataflow of prioritni_inhibitor is
signal inh: std_logic_vector(qv'RANGE);
begin
        qv(qv'HIGH)<=av(av'HIGH); inh(qv'HIGH)<=av(av'HIGH);
        cyklus: for i in qv'HIGH-1 downto 0 generate
                qv(i)<=NOT inh(i+1) AND av(i); inh(i)<=inh(i+1) OR av(i);
        end generate;
end;
```

Think about whether the outcome would change if we reversed the generation cycle?

```
begin
        qv(qv'HIGH)<=av(av'HIGH); inh(qv'HIGH)<=av(av'HIGH);
        cyklus: for i in 0 to qv'HIGH-1 generate
                qv(i)<=NOT inh(i+1) AND av(i); inh(i)<=inh(i+1) OR av(i);
        end generate;
end;
```

A classic sequential program would undoubtedly create a malfunctioning solution. But in the concurrent code, we connect only the signals!

Check your circuits in the simulator, either downto or to generate loop. For simulation, reduce the generic value of N to 7, for example, to allow displaying of all. Simple Simulation Waveform Editor does not allow simulation to run longer than 100 microseconds, so it can test the highest priority inhibitor with a maximum of N = 12. ModelSim, which we learn in lectures, offers the simulation of any duration.

Note: Take in account information from Appendix C that we must change simulation end time (main simulation Waveform EditorMenu: Edit-> Set End Time...) *before inserting any signals, i.e., immediately after the * .wvf editor is opened. Later adjustments of end time may, unfortunately, cause the exception in Quartus followed by its crash!*

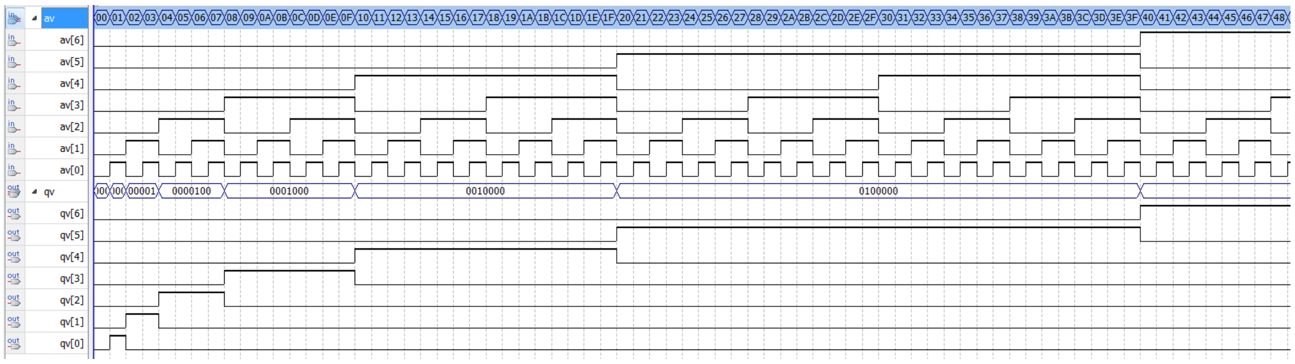


Figure 13 - Part of simulation for N=7

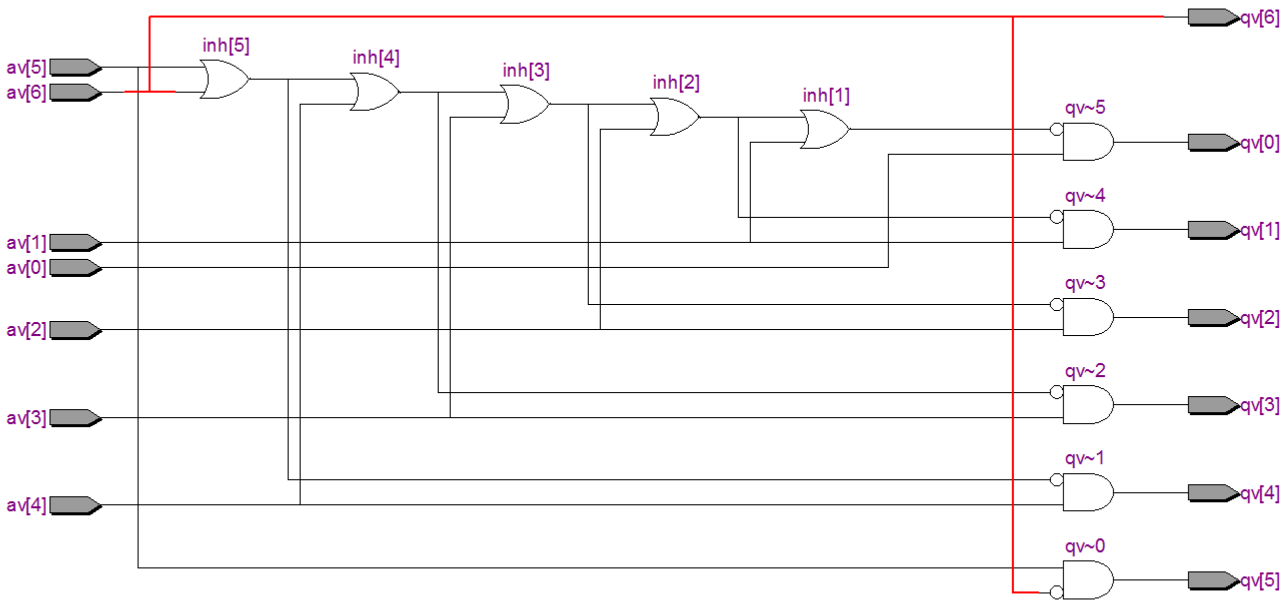


Figure 14 - RTL Viewer schéma pro N=7

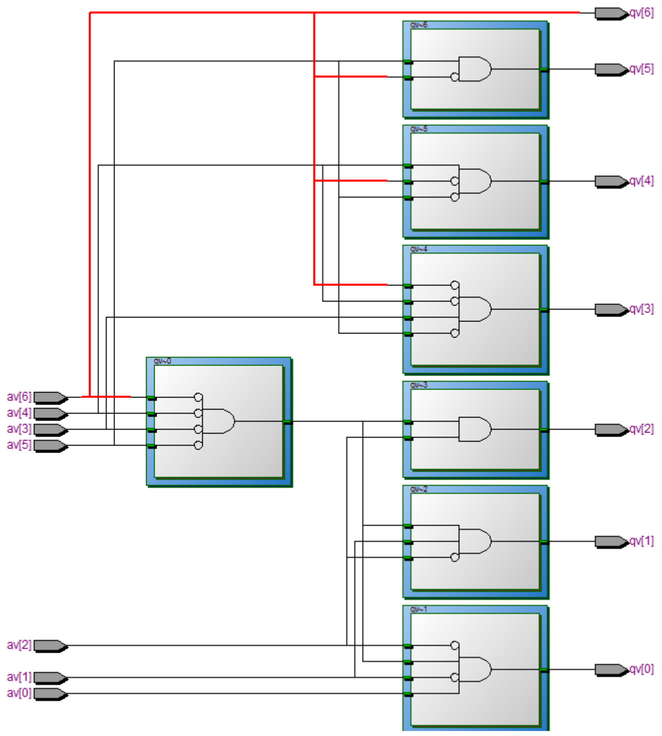


Figure 15 - Technology Map for N=7

8.5 Practice 5: Universal interrupt coder

For practice from page 39, we present a possible solution, in which we emphasize the parts added into old interrupt coder:

```vhdl
--Universal interrupt priority decoder
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity irq_priorityN is
   generic (   IRQ_MAX : integer:=15; --number of interrupt inputs
               Q_LENGTH : integer:=4); -- width of number output
   port (  -- interrupt request, the higher index has higher priority
           IRQ : in std_logic_vector(IRQ_MAX downto 1);
           -- unsigned number of the highest active interrupt
           Q : out std_logic_vector(Q_LENGTH-1 downto 0) );
   begin
   assert IRQ_MAX>=2 report "Required IRQ_MAX >=2" severity failure;
   assert Q_LENGTH>=2 report "Required Q_LENGTH >= 2" severity failure;
   assert 2**Q_LENGTH > IRQ_MAX report "Q_LENGTH too low for encodinq IRQ_MAX"
   severity warning;
end;
architecture dataflow of irq_priorityN is
type qtmp_array_t is array (IRQ'RANGE) of unsigned(Q'RANGE);
signal qtmp : qtmp_array_t;
begin
    qtmp (IRQ'LOW)<= (others=>'0') when IRQ(IRQ'LOW)='0'
                                       else to_unsigned(IRQ'LOW, Q_LENGTH);
     cyclus: for i in IRQ'LOW+1 to IRQ'HIGH generate
                qtmp (i) <= qtmp (i-1) when IRQ(i)='0' else to_unsigned(i, Q_LENGTH);
                end generate;
    Q <= std_logic_vector(qtmp(IRQ'HIGH));
end;
```

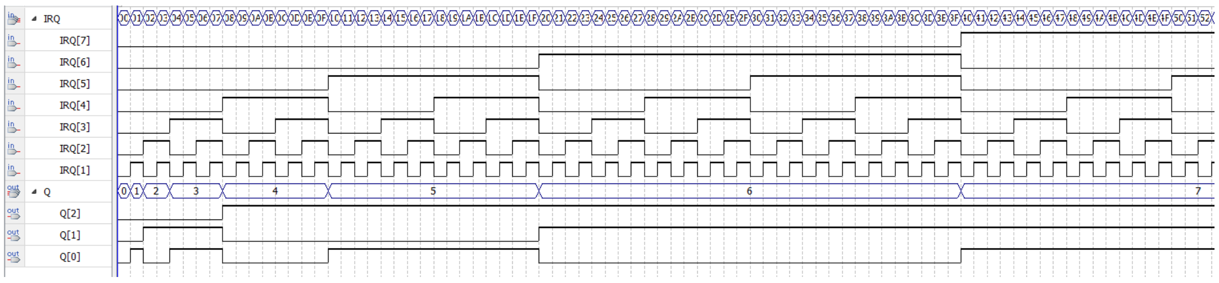


Figure 16 - Část simulace pro N=7

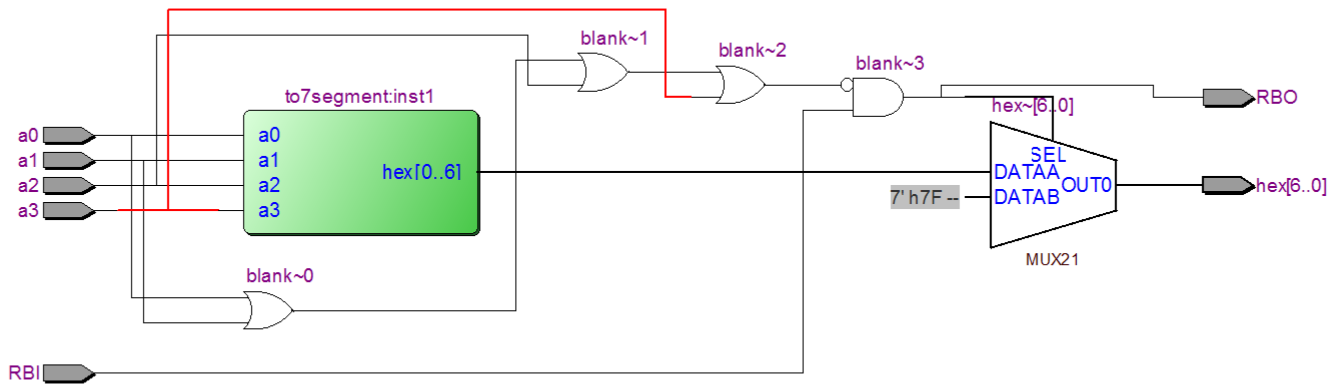
Note: We could much easier build the circuit by a behavioral style. The compiler would do the unavoidable definition of the qtmp auxiliary array and use it in expansions of repeated writes to concurrent commands allowing a single assignment. However, the above code corresponds to how translation could transform behavioral code into concurrent.

8.6 Practice 6: 7segment decoder with blanking leading zeros

The practice on page 46 has, for example, this solution:

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity to7segment_RBIO is
        port (  a0, a1, a2, a3 :  in  std_logic; -- a3.. a0 input hexadecimal number a0-lsb  a3-msb
                RBI : in std_logic; -- ripple blank in
                hex : out std_logic_vector(6 downto 0); -- hex[6..0] output to 7 segment display
                RBO: out std_logic); -- ripple blank out
end;

architecture dataflow of to7segment_RBIO is
component to7segment is
        port(   a0, a1, a2, a3 :in std_logic;
                hex: out std_logic_vector(6 downto 0) );
end component;

signal tmp : std_logic_vector(hex'RANGE);
signal blank : std_logic;
begin
        blank <= RBI AND NOT (a0 OR a1 OR a2 OR a3); -- RBI='1' and all inputs = '0'
        inst1 : to7segment port map(a0, a1, a2, a3, tmp);
        hex <= (others=>'1') when blank='1' else tmp; -- On DE2, hex led is ON on '0'
        RBO <= blank;
end;
```

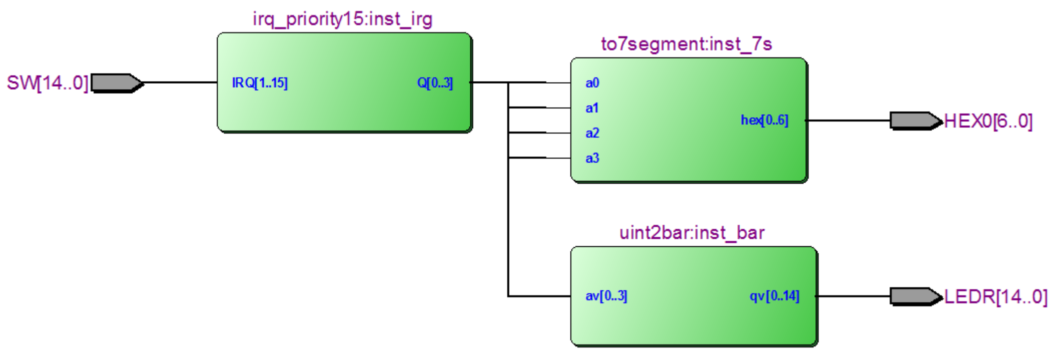
Quartus built the circuit from 7segment decoder extended by the multiplexor, which suppressed hex-lighting if inputs='0' and RBI='1'.



8.7 Practice 7: Adding 7segment display

We solve the practice on page 48 by inserting component to7segment. We create its instance that we connect to already existing signal Q2av and output HEX0 added into the port section of the entity. The identifier corresponds to rightmost digit of board DE2. If we set demo_irq_barh as the Top-Level entity, Quartus creates connection to HEX0 assignment.

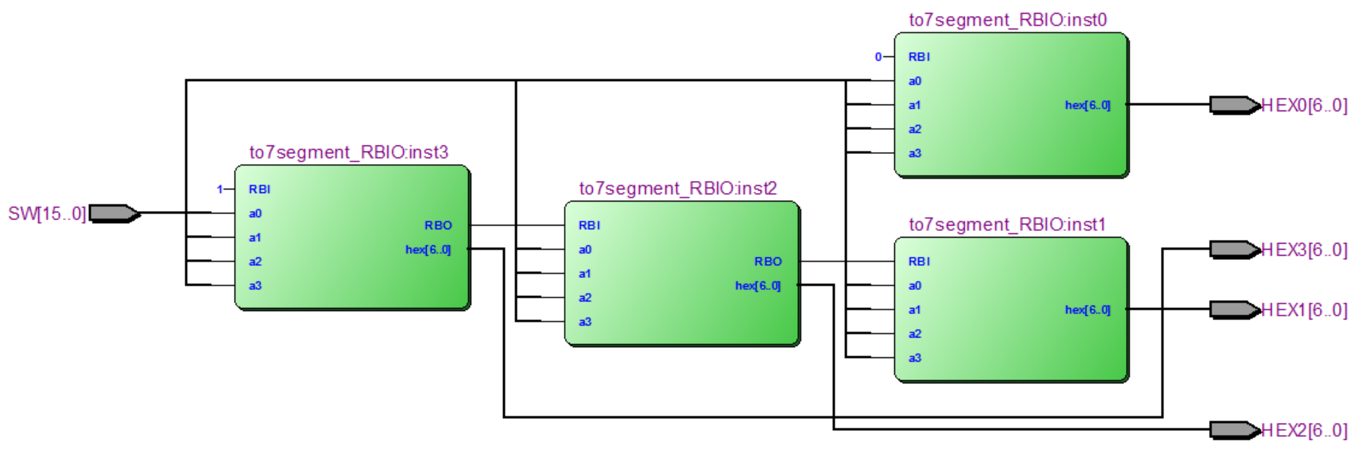
```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_irq_barh is  port ( SW :  in  std_logic_vector(14 downto 0);
                              LEDR :  out  std_logic_vector(14 downto 0);
                              HEX0: out std_logic_vector(6 downto 0));
end;
architecture dataflow of demo_irq_barh is
  component uint2bar
      generic ( AV_LENGTH : integer:=4; QV_LENGTH : integer:=15);
      port(   av : in std_logic_vector(AV_LENGTH-1 downto 0);
              qv : out std_logic_vector(QV_LENGTH-1 downto 0));
  end component;
  component irq_priority15 is
      port (  IRQ : in std_logic_vector(15 downto 1);
              Q : out std_logic_vector(3 downto 0) );
  end component;

component to7segment is
      port(   a0, a1, a2, a3 :in std_logic;
              hex: out std_logic_vector(6 downto 0) );
end component;
  signal Q2av :  std_logic_vector(3 downto 0);
begin
  inst_bar : uint2bar generic map(Q2av'LENGTH, LEDR'LENGTH) port map( Q2av, LEDR );
  inst_irg : irq_priority15 port map( IRQ=>SW, Q=>Q2av );
  inst_7s: to7segment port map(   a0=>Q2av(0),a1=>Q2av(1), a2=>Q2av(2), a3=>Q2av(3),
                                  hex=>HEX0);
end architecture;
```



8.8 Practice 8: Show 16-bit number on 7segment display

In practice from page 54, we do not utilize for-generate loop because there are only few same repetitions. We instead use copy-paste-edit of the command for creating instances, in which we address the inputs from the switches SW and HEX digits outputs. For connections according to the picture, we create two auxiliary signals s3to2 and s2to1.

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_uint16toHex is
port (  SW :  in  std_logic_vector(15 downto 0);
        HEX0, HEX1, HEX2, HEX3: out std_logic_vector(6 downto 0));
end;
architecture dataflow of demo_uint16toHex is
  component to7segment_RBIO is
     port (  a0, a1, a2, a3 :  in  std_logic; -- a3.. a0 input hexadecimal number a0-lsb  a3-msb
             RBI : in std_logic; -- ripple blank in
             hex : out std_logic_vector(6 downto 0); -- hex[6..0] output to 7 segment display
             RBO: out std_logic); -- ripple blank out
  end component;
signal s3tos2, s2tos1 : std_logic;
begin
     inst3: to7segment_RBIO
               port map(  a0=>SW(12),a1=>SW(13), a2=>SW(14), a3=>SW(15),
                          RBI=>'1', hex=>HEX3, RBO=>s3tos2);
     inst2: to7segment_RBIO
               port map(  a0=>SW(8),a1=>SW(9), a2=>SW(10), a3=>SW(11),
                          RBI=>s3tos2, hex=>HEX2, RBO=>s2tos1);
     inst1: to7segment_RBIO
               port map(  a0=>SW(4),a1=>SW(5), a2=>SW(6), a3=>SW(7),
                          RBI=>s2tos1, hex=>HEX1);
     inst0: to7segment_RBIO
               port map(  a0=>SW(0),a1=>SW(1), a2=>SW(2), a3=>SW(3),
                          RBI=>'0', hex=>HEX0);
end architecture;
```



If we want to make the adjustment that allows switching on and off the suppression of leading zeros, then this is a ridiculously primitive change. If we connect '0' on the RBI of the highest digit instead of the original '1', so we choose to display the digit, we switch off the option entirely. After that, all lower digits always shine.

So we can expand the input of SW and plug in SW [16] to RBI inst3. For example, editing your code might look like this:

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;

entity demo_uint16toHexR is
port (  SW :  in  std_logic_vector(16 downto 0);
        HEX0, HEX1, HEX2, HEX3: out std_logic_vector(6 downto 0));
end;
architecture dataflow of demo_uint16toHexR is
  component to7segment_RBIO is
      port (  a0, a1, a2, a3 :  in  std_logic; -- a3.. a0 input hexadecimal number a0-lsb  a3-msb
              RBI : in std_logic; -- ripple blank in
              hex : out std_logic_vector(6 downto 0); -- hex[6..0] output to 7 segment display
              RBO: out std_logic); -- ripple blank out
  end component;
signal s3tos2, s2tos1 : std_logic;
begin
      inst3: to7segment_RBIO
                port map(  a0=>SW(12),a1=>SW(13), a2=>SW(14), a3=>SW(15),
                           RBI=>SW(16), hex=>HEX3, RBO=>s3tos2);
      inst2: to7segment_RBIO
                port map(  a0=>SW(8),a1=>SW(9), a2=>SW(10), a3=>SW(11),
                           RBI=>s3tos2, hex=>HEX2, RBO=>s2tos1);
      inst1: to7segment_RBIO
                port map(  a0=>SW(4),a1=>SW(5), a2=>SW(6), a3=>SW(7),
                           RBI=>s2tos1, hex=>HEX1);
      inst0: to7segment_RBIO
                port map(  a0=>SW(0),a1=>SW(1), a2=>SW(2), a3=>SW(3),
                           RBI=>'0', hex=>HEX0);
end architecture;
```

8.9 Practice 9: Priority inhibitor for three sixes

In practice from page 54, we take advantage of for-generate loop, which allows shortering code:

```
--Priority inhibitor for 3 groups with 6 members
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_priority_inhibitor3x6 is
   port (  SW :  in  std_logic_vector(17 downto 0);
           LEDR : out std_logic_vector(17 downto 0) );
end;

architecture dataflow of demo_priority_inhibitor3x6 is

component prioritni_inhibitor is
   generic( N: natural := 7);  -- length of vector > 1
   port (  av : in std_logic_vector(N-1 downto 0);
           qv : out std_logic_vector(N-1 downto 0) );
end component;

begin
     cyclus:  for i in 0 to 2 generate
          instX : prioritni_inhibitor
                 generic map(N=>6)
                 port map(av=>SW(6*i+5 downto 6*i), qv=>LEDR(6*i+5 downto 6*i));
              end generate;
end architecture;
```

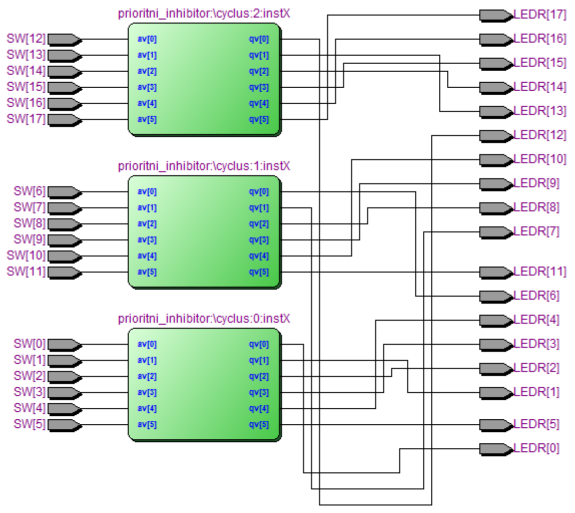

prioritni_inhibitor:cyclus:2:instX
prioritni_inhibitor:cyclus:1:instX
prioritni_inhibitor:cyclus:0:instX
SW[0] … SW[17]
av[0] … av[5]
qv[0] … qv[5]
LEDR[0] … LEDR[17]

Figure 17 - Prioritní inhibitor pro 3 šestice

We did not intentionally use any auxiliary variable, but we wrote repeatedly 6*i. We discussed the reasons on page **Chyba! Záložka není definována.**, where we mentioned unsuitable VHDL codes.

Changing to 6 triplets takes a few seconds. You need only to change the numerical value 2 to 5, 6 to 3, and 5 to 2. We certainly did not redraw any block diagram so swiftly☺

```
...
begin --Architecture block of priority inhibitor for 6 groups with 3 members
     cyclus:  for i in 0 to 5 generate
          instX : prioritni_inhibitor
                 generic map(N=>3)
                 port map(av=>SW(3*i+2 downto 3*i), qv=>LEDR(3*i+2 downto 3*i));
              end generate;
end architecture;
```

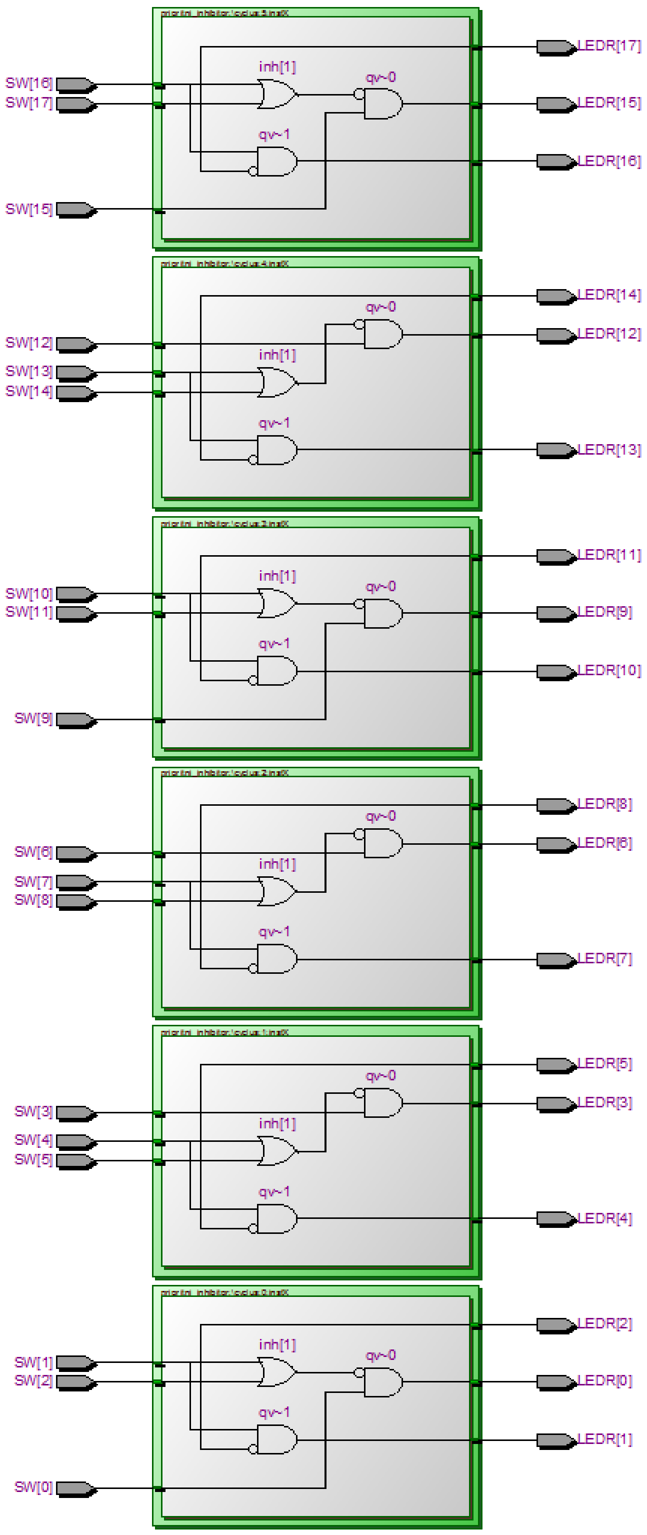


Figure 18 - Priority inhibitor for 6 triplets

9 Appendix B: Operations with std_logic - Resolution tables

Library ieee.std_logic_1164 defines std_logic data type in two steps. First, it introduces 9-value logic std_ulogic with enumerated members, we already know:

```
type std_ulogic is ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak Unknown
                     'L',  -- Weak  0
                      'H',  -- Weak 1
                     '-'   -- Don't care );
```

Here, ulogic means unresolved — we do not know the value in case of connecting more outputs to one point, which occurs, for example, on data buses. Then, the library uses std_ulogic for the creation

```
subtype std_logic is resolved std_ulogic;
```

in which special function resolved solved this situation according to the following table:

connection	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Table 10 - Definition of resolved function for std_logic

Note: On the Cyclone II FPGA in DE2 boards, we can couple multiple outputs only if they have the same constant value, or one of them at most is in the '0' or '1' state and the others are in high impedance 'Z'.

FPGA Cyclone II does not contain logic capable of handling other situations because it has no open collector circuits ('L', 'W', and 'H' values) that allow multiple connections (these are called Wired -OR, see https://en.wikipedia.org/wiki/Wired_logic_connection).

However, the universal library solves all cases.

The tables presented on the next page specify the results of logical operations with std_ulogic, and also with its subtype std_logic.

AND	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'0'	'U'	'U'	'U'	'0'	'U'	'U'
'X'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'Z'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'W'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'L'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'H'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'-'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'

OR	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'1'	'U'	'U'	'U'	'1'	'U'
'X'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'0'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'
'Z'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'W'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'L'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'H'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'
'-'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'

NOT	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'X'

XOR	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'1'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'X'
'Z'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'W'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'L'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'H'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'X'
'-'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'

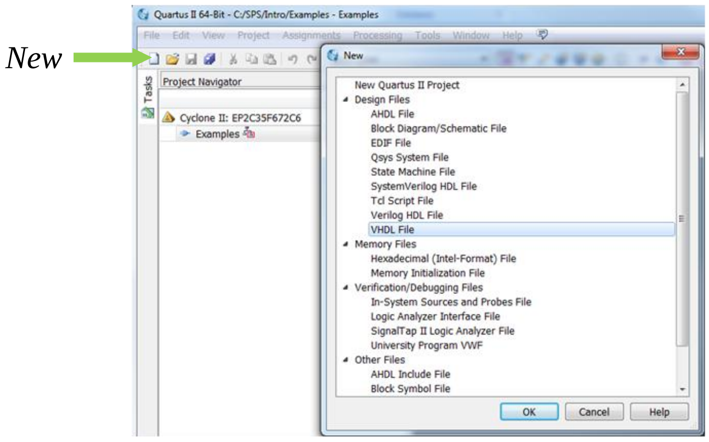
Table 11 - Logical operations

~o~

10 Appendix C: Create and compile a VHDL file majorita.vhd

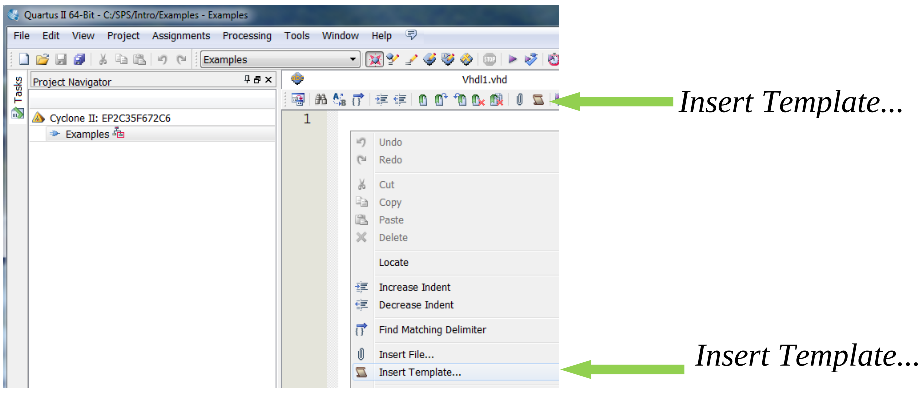
First, we have to create a Quartus project by following the instructions on DCENET site: http://dcenet.felk.cvut.cz/edu/fpga/doc/CreateNewQuartusProject.pdf .

We name our project, for instance, "Examples" and insert a new VHDL file into it, which can be done either from the Quartus menu by choosing File-> New... or by clicking on the New icon on the toolbar. We select VHDL File option in the dialog and click [Ok].


New
Quartus II 64-Bit - C:/SPS/Intro/Examples - Examples
File Edit View Project Assignments Processing Tools Window Help
Tasks
Project Navigator
Cyclone II: EP2C35F672C6
Examples
New
New Quartus II Project
Design Files
AHDL File
Block Diagram/Schematic File
EDIF File
Qsys System File
State Machine File
SystemVerilog HDL File
Tcl Script File
Verilog HDL File
VHDL File
Memory Files
Hexadecimal (Intel-Format) File
Memory Initialization File
Verification/Debugging Files
In-System Sources and Probes File
Logic Analyzer Interface File
SignalTap II Logic Analyzer File
University Program VWF
Other Files
AHDL Include File
Block Symbol File
OK
Cancel
Help

Into our VHDL file, we copy our template from chapter 2.6 "VHDL template" on page 11.

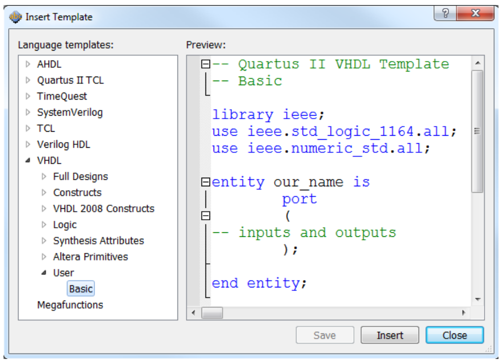
Alternatively, in Quartus Tools-> Options-> Text Editor - User template library directory, we can specify our user template directory that is **different** from the pre-installed templates in folder C:\altera. Then, we pop-up the template dialog either by clicking on the "Insert Template" icon or via the context menu of the VHDL editor window.



In the template dialog, we create a new template using the context menu in the Language Template section. We rename it to Basic (double-click on the automatically created name), and we insert/write the text of our template text into the Preview section. We save the result.

If our template is already stored, we can anytime insert it by the following steps:

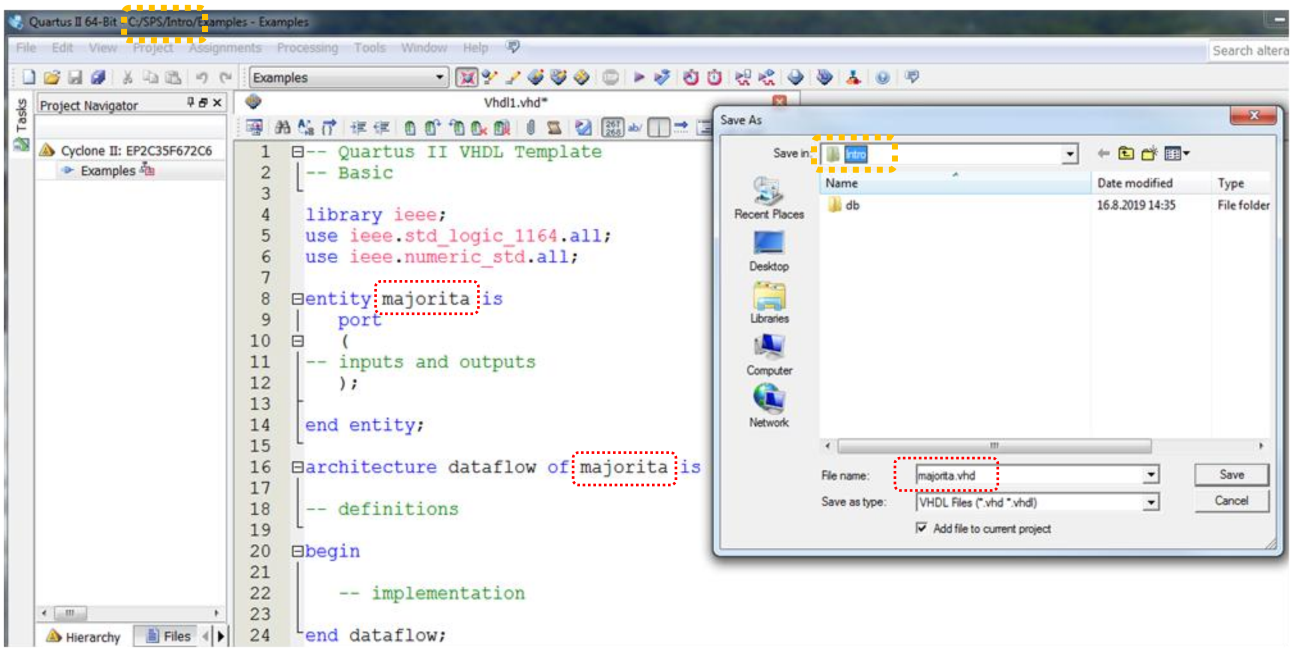
1. We pop-up the template dialog by selecting Insert Template in VHDL editor window.
2. In the dialog, we select our template from the left tree: VHDL -> User -> Basic
3. By clicking Insert, we insert the selected template at the current cursor position in the VHDL editor window.
4. Finally, we close the dialog by OK.



```
-- Quartus II VHDL Template
-- Basic

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity our_name is
        port
        (
-- inputs and outputs
        );

end entity;
```

10.1 Saving VHDL file

In the newly created VHDL file, we immediately perform naming trinity. We assign "majorita" to the entity, to the architecture, and finally, we save the file as "majorita.vhd" by File-> Save As ... When saving, we carefully check whether we **store into the folder of our project**. The Linux Quartus application running under Cygwin occasionally takes another folder from Windows. We have to be careful about this.

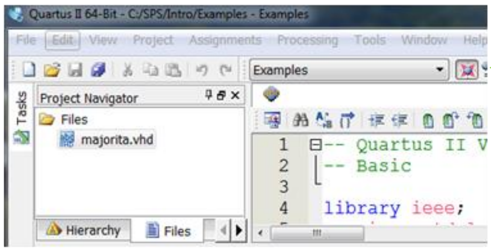


```
-- Quartus II VHDL Template
-- Basic

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity majorita is
    port
    (
-- inputs and outputs
    );

end entity;

architecture dataflow of majorita is

-- definitions

begin

    -- implementation

end dataflow;
```

Name	Date modified	Type
db	16.8.2019 14:35	File folder

File name: majorita.vhd
Save as type: VHDL Files (*.vhd *.vhdl)
☑ Add file to current project

We switch "Projekt Navigator" window on tab Files and verify, whether we see the name of our file without leading directories (/ characters), thus, the file is located in the main directory of our project.



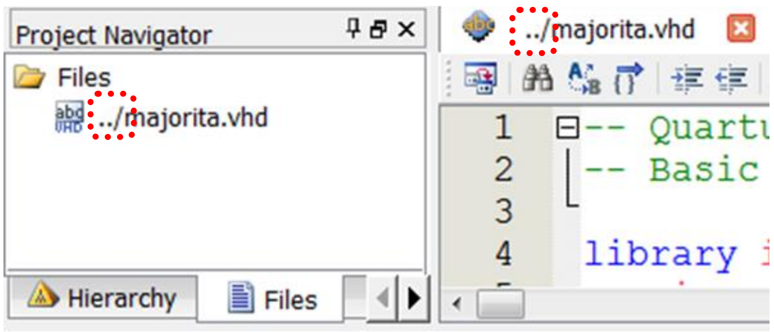
If we see / before the filename, we accidentally saved it in another directory.

We take advantage of Quartus not locking access to the opened files. It loads them in its internal memory.

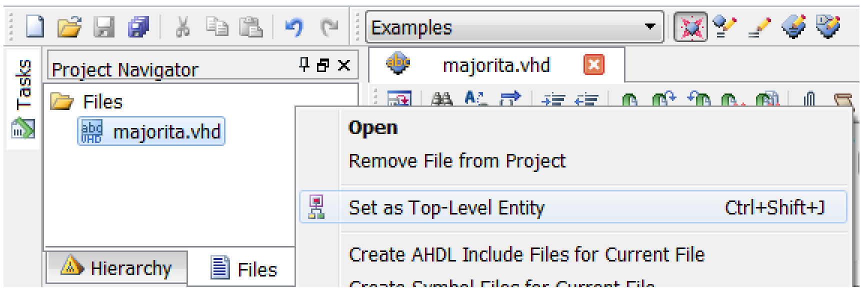
In the "Project Navigator" window on the Files tab, we use [Delete] key to remove the file from the list of our project, and we also delete it on the disk.

Then, we save VHDL file opened in the editor to the correct directory.

File majorita.vhd is not stored in the project root directory!



If we have the file in the proper folder, we select it in Project Navigator tab Files, and in its context menu, we select "Set as To-Level Entity", so majority.vhd becomes the primary entity.



Windows "Project Navigator" on its Hierarchy tab shows the file selected as Top-Level entity.

> *Note: The Top-Level entity option determines what we build. If we have forgotten to change it, then a different circuit remained Top-Level entity, and not our desired one. We are compiling and downloading the result into the development board, and we are surprised that our damned design, despite all the adjustments, still does not operate. In classical programming, we could encounter an analogous situation when we were accidentally starting the previous version of a program and not a new compiled.*

We write majority code by editing the inserted template:

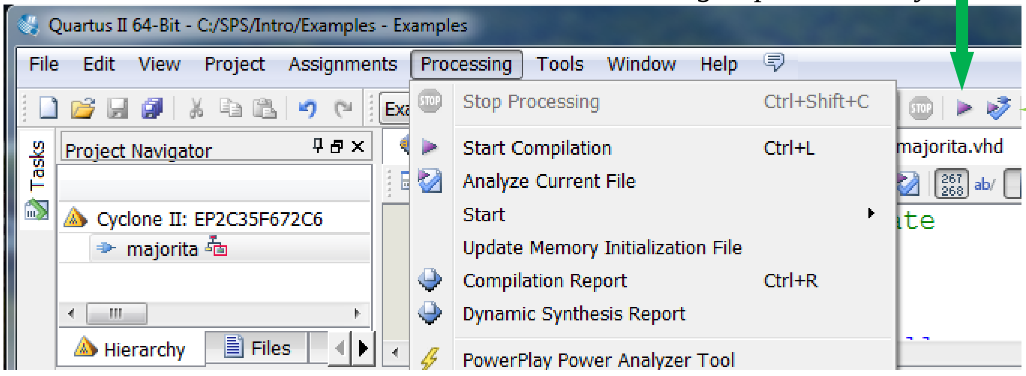
```vhdl
-- Majority 2 of 3
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity  majorita is
        port (  a, b, c : in std_logic; y : out std_logic );
end;
architecture dataflow of majorita is
begin
        y <= (a AND b) OR (a AND c) OR (b AND c);
end;
```

10.2 Compilation

When compiling, Quartus always checks the syntax of all the files listed in our project. This step only represents an initial phase followed by minimizing and assembling the circuit defined as Top-Level Entity.

So we can only invoke the first phase of translation, relatively fast, and suitable for error checking. In the end, we choose a full compilation.

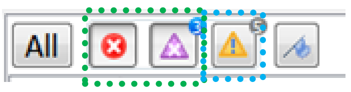
***Full compilation** with checking of syntax of all files in the project and building Top-Level entity*



***Partial compilation** - syntax check of all files in the project and analysis of *.vhd file shown in editor window. No circuit is*

10.3 Errors and warning during compilation

Quartus displays quite much information at its window "Messages" [View->Utility Windows -> Messages]. **Why?** In industrial practice, it often runs as an external slave-tool that cooperates with other design applications. It supplies them with reports about essential facts.



If we use Quartus in window-mode, we can sort messages by their ID, filter their texts, or select their category. The most severe message category is Error, then Critical Warnings - keep both categories always selected with buttons in the Message window (framed in green in right picture).

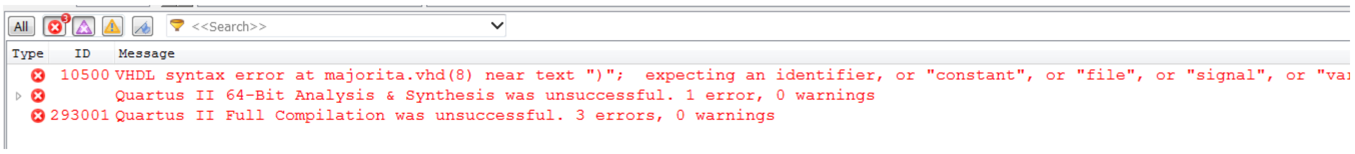
Note that here we have Critical Warnings and Warnings. Critical warnings correspond in importance to the C language warnings. On the other hand, we usually overlook statements marked as "Warnings" since these are only minor reports.

We can neglect the last category, Information, which usually serves as a source of data for external tools. Only its the most recent information is essential - if the result of compilation was successful.

Here is an example of how to look for errors. Suppose we made a common mistake in majorita.vhd code - we wrote an extra semicolon in the port section:

```
port (  a, b, c : in std_logic;
                 y : out std_logic; );
```

The compiler shows: (*To display its message window, select from menu View->Utility Windows->Messages*)



```
Type   ID     Message
  ⊗  10500 VHDL syntax error at majorita.vhd(8) near text ")";  expecting an identifier, or "constant", or "file", or "signal", or "va
▷ ⊗        Quartus II 64-Bit Analysis & Synthesis was unsuccessful. 1 error, 0 warnings
  ⊗ 293001 Quartus II Full Compilation was unsuccessful. 3 errors, 0 warnings
```

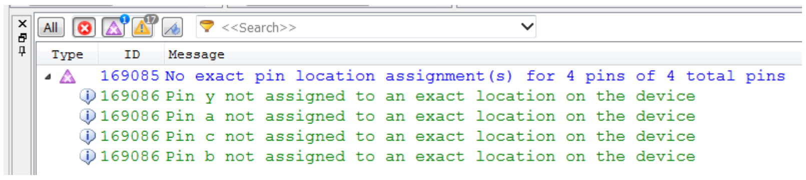
The first line

Error (10500): VHDL syntax error at majorita.vhd(8) near text ")"; expecting an identifier, or "constant", or "file", or "signal", or "variable"

results in the error, the following lines present only its results. We find out our VHDL filename, followed by the number of the line where the error was detected. As a rule, we can double-click on the message, and the editor window displays the line. Only sometimes, we need to scroll to the line manually.

The given line does not necessarily contain inappropriate code. Errors can sometimes occur somewhere before the line, see the discussion on page 9. We'll track down their causes, fix them, and compile the corrected code. Quartus always saves all files before any compilation, so everything is always on the disk.

After the successful compilation of majorita.vhd, we see a critical warning:



```
Type    ID     Message
◢ △   169085 No exact pin location assignment(s) for 4 pins of 4 total pins
    ⓘ 169086 Pin y not assigned to an exact location on the device
    ⓘ 169086 Pin a not assigned to an exact location on the device
    ⓘ 169086 Pin c not assigned to an exact location on the device
    ⓘ 169086 Pin b not assigned to an exact location on the device
```

Quartus tells us that the input and output pins of Top-Level entity file do not lead to inputs and outputs of the FPGA circuit. If we plan to download the circuit to the development board, this is a severe error. If we only want to simulate the result, then we do not mind unconnected pins.

The "Compilation Repor" window is also essential:

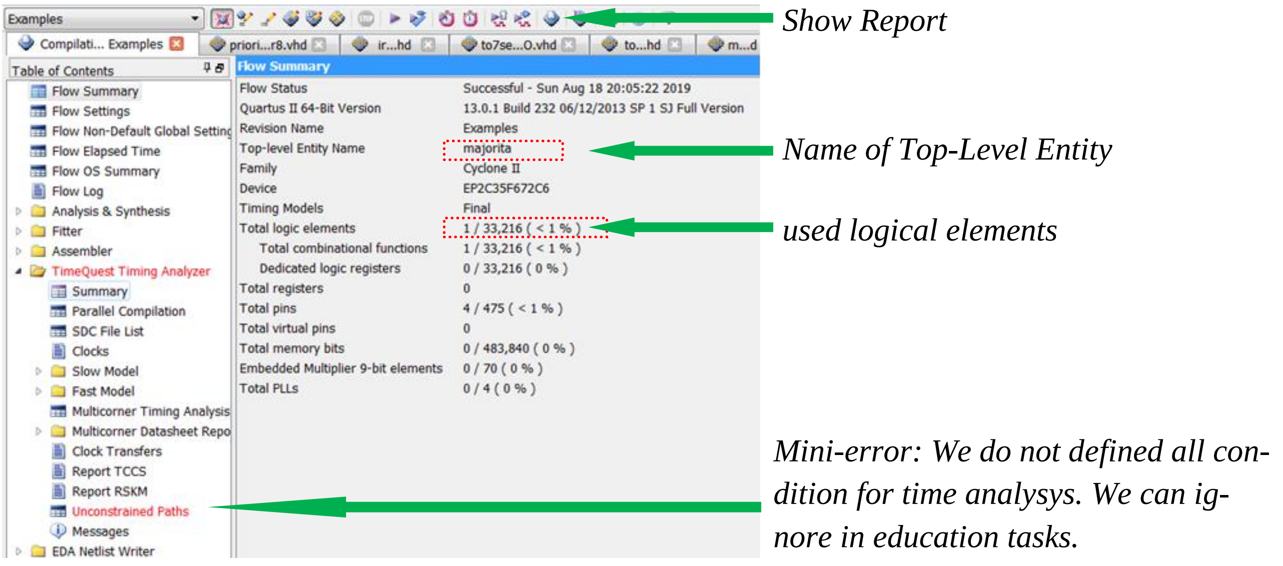


Table of Contents
- Flow Summary
- Flow Settings
- Flow Non-Default Global Setting
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
- TimeQuest Timing Analyzer
  - Summary
  - Parallel Compilation
  - SDC File List
  - Clocks
  - Slow Model
  - Fast Model
  - Multicorner Timing Analysis
  - Multicorner Datasheet Repo
  - Clock Transfers
  - Report TCCS
  - Report RSKM
  - Unconstrained Paths
  - Messages
- EDA Netlist Writer

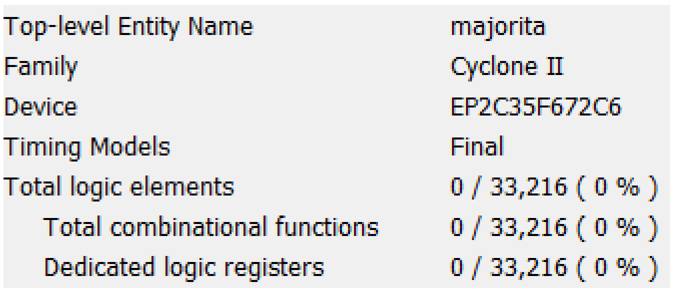
Flow Summary	
Flow Status	Successful - Sun Aug 18 20:05:22 2019
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	Examples
Top-level Entity Name	majorita
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	1 / 33,216 (< 1 %)
Total combinational functions	1 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	4 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

The report shows what we have translated and the number of logical elements (LE) consumed by our circuit. Indeed it should use at least 1 LE. If there is 0 LE, something is very wrong.

For example, if we incorrectly wrote the majority equation as:

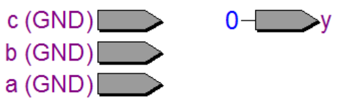
```
y <= (a AND b) XOR (a AND b);
```

the code compiles without errors, but we saw:



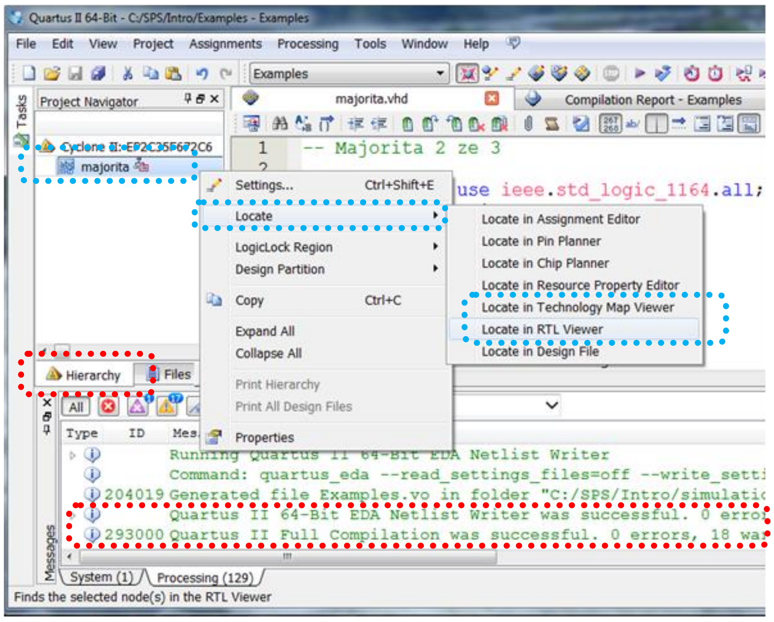
Top-level Entity Name	majorita
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	0 / 33,216 (0 %)
Total combinational functions	0 / 33,216 (0 %)
Dedicated logic registers	0 / 33,216 (0 %)

The minimization discovered that the equation was a logical contradiction, so the result corresponded to the pure connection of output y to '0'.



10.4 Showing RTL Map and Technology Map

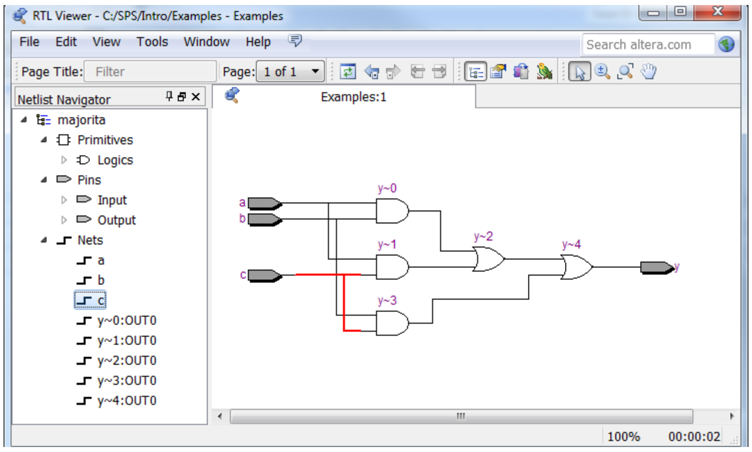
We can also display both maps after a partial compilation, but it is better to perform the full compilation. In Project Navigator window Hierarchy tab, we recall the Top-Level entity context menu by right-clicking and select Locate option:



```
1   -- Majorita 2 ze 3
    use ieee.std_logic_1164.all;
```

```
        Running Quartus II 64-Bit EDA Netlist Writer
        Command: quartus_eda --read_settings_files=off --write_setti
204019  Generated file Examples.vo in folder "C:/SPS/Intro/simulatic
        Quartus II 64-Bit EDA Netlist Writer was successful. 0 erro
293000  Quartus II Full Compilation was successful. 0 errors, 18 wa
```

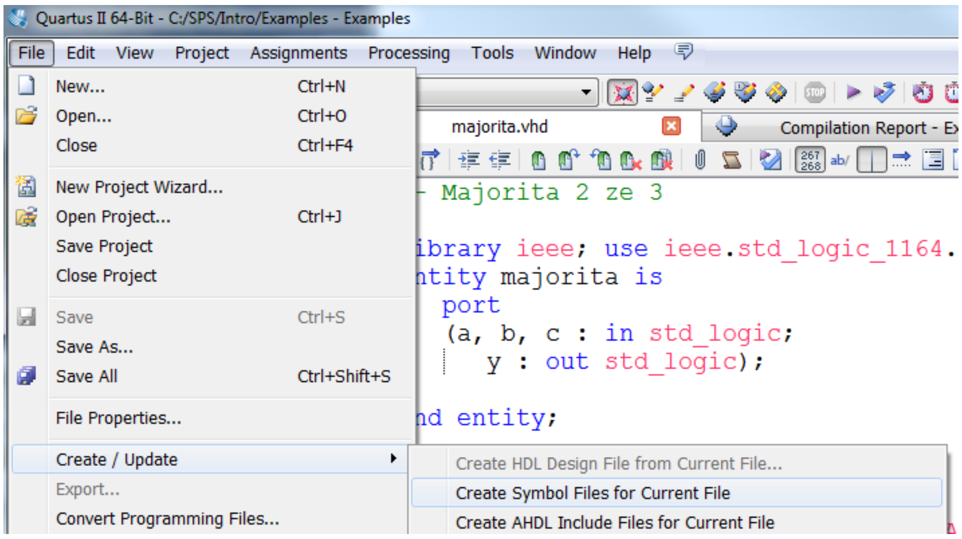
In both shown maps, RTL, or Technology, we can choose elements and change their visualization by their context menus. We can adjust drawing either from the main menu Tools-> Options or Tools-> Customize, or via Viewer Options... We can also locate individual members in Netlist Navigator panel (If it is hidden, we open it from View menu). We freely play here; everything is read-only.



10.5 Create symbol for Quartus symbolic editor

We only need symbols if we want to use our circuits in Quartus symbolic editor. Otherwise, we don't have to create them; VHDL codes don't work with them.

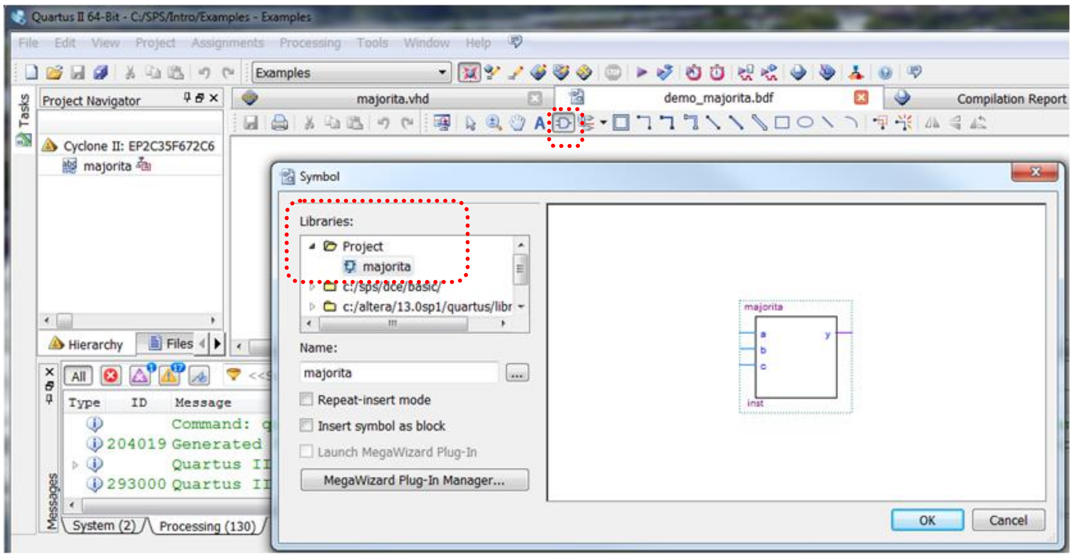
We open our VHDL file to be visible in the editor. Its compilation should be successful otherwise the creation fails. From the main menu, we select File-> Create/Update-> Create Symbol Files for Current File



```
- Majorita 2 ze 3

ibrary ieee; use ieee.std_logic_1164.
ntity majorita is
  port
  (a, b, c : in std_logic;
     y : out std_logic);

nd entity;
```

The symbol file * .bsf has text format, and it contains only I/O pin list and drawing in a LISP-like format. We can view the file with a text editor, but we do not make changes here because its structure is too untransparent. We have mentioned its text format to emphasize that * .bsf does not contain any VHDL code, just input/output identifiers, and drawing description. **The symbol remains valid until we change the inputs and outputs**, that is, only if we change its entity section, we must recreate the symbol. If we edit only the architecture, there is no need to rebuild *.bsf symbol.

To insert the symbol, we call the Symbol Tool of symbolic file editor (either from the toolbar or double-click anywhere on the free area of the drawing window). We find out our symbol in library Project.



```
Command: q
204019 Generated
       Quartus II
293000 Quartus II
```

*__Important note:__ Quartus has created the symbol in the same directory where we stored our * .vhd file. If the file is outside of both main directory of our project and the paths specified as the library directories during the initial project configuration, then the symbol may not be found. In such case we need to change first the file location, see chapter 10.1 on page 75, and then, generate our symbol again.*

We can connect inputs and outputs to our majorita circuit either by manually or we save a bit of work.

Using the right mouse button, we pop up the context menu of majorita, and we automatically generate inputs and outputs for it by selecting "Generate Pins for Symbol Ports". Then, we rename inserted "I/O pins" according to the "assignments" of the DE2 board, either through the context Properties of the individual I/O pins or by double-clicking on the pin.

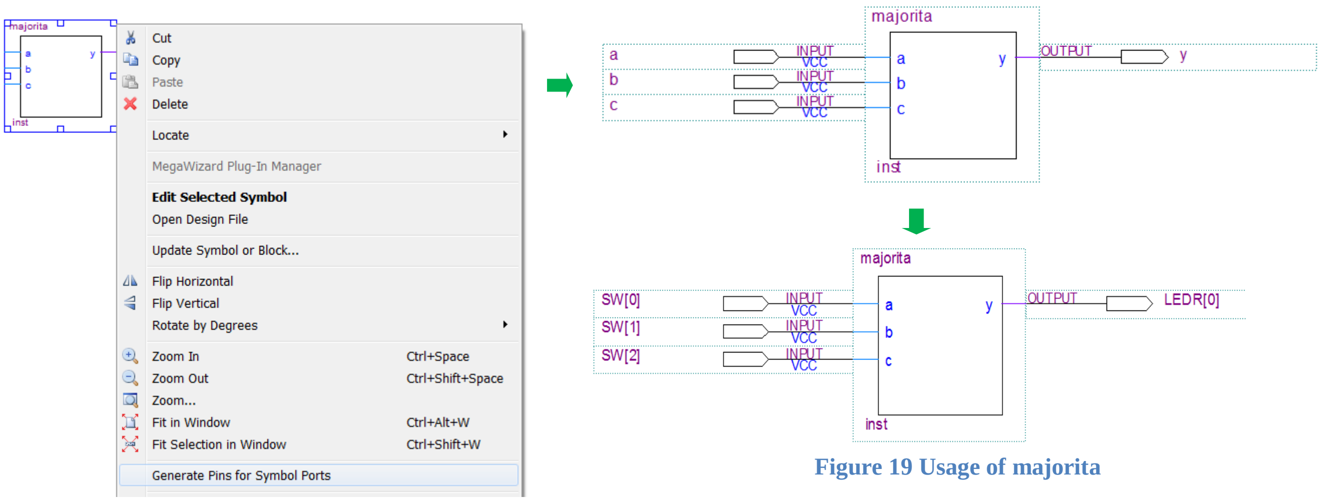


Figure 19 Usage of majorita

We can perform the most comfortable renaming of a pin by selecting only displayed I/O name. We double-clicked directly on the text inside the pin that is not marked. If it is marked then clicking anywhere on the open area unmarked it, and then, it is possible to select the text itself. After editing the text, we press Enter key, which moves the edition to the next input/output lying in the graphical editor below. Moreover, renaming multiple similar outputs can be speeded up by copying the appropriate prototype of the names into the clipboard, pasting it, correcting it, and moving to the next one by Enter key.

Note: Using a symbolic editor is easy, but laborious. In VHDL, we can write Figure 19 by port map *introduced in chapter* 6*, by the following way:*

```
-- Majority 2 of 3
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity  demo_majorita is
        port (  SW : in std_logic_vector(2 downto 0);
                LEDR : out std_logic_vector(0 to 0));
end;
architecture dataflow of demo_majorita is
   component majorita is port (a, b, c : in std_logic; y : out std_logic );
   end component;
begin
        inst : majorita port map(SW(0), SW(1), SW(2), LEDR(0));
end;
```

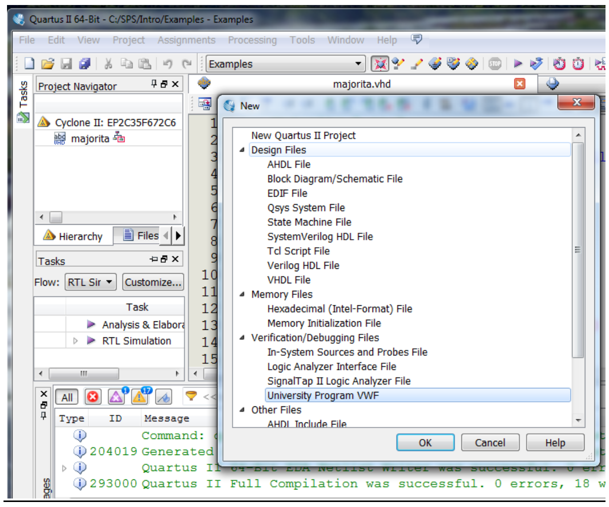
10.6 Simulation

Suppose we have a majority circuit created, and its full compilation went without error, i.e., the last one was an informative report "*Info (293000): Quartus II Full Compilation was successful. 0 errors...*".

Choose "File-> New…" from the Quartus main menu, and in the dialog that appears, select: "University Program VWF" (Vector Waveform File)

Click [OK]

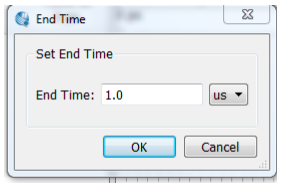
Window Simulation Waveform Editor opens. We immediately save its content by File->Save As... as majorita_Waveform.vwf



```
       Command: 
204019 Generated
       Quartus II 64-Bit EDA Netlist Writer was successful. 0 err
293000 Quartus II Full Compilation was successful. 0 errors, 18 w
```

Note: In Quartus, we should not distinguish files only by a simple extension. We should give them different names to prevent frequent questions about which of them we want to open.

We can **change the simulation time** now, i.e., before we insert any signals. Then, it is no longer possible. (*To be more accurate, Quartus usually crashes on an internal error if we try to set a longer end-time after inserting signals. Adjustments to a shorter time are usually OK anytime.)*

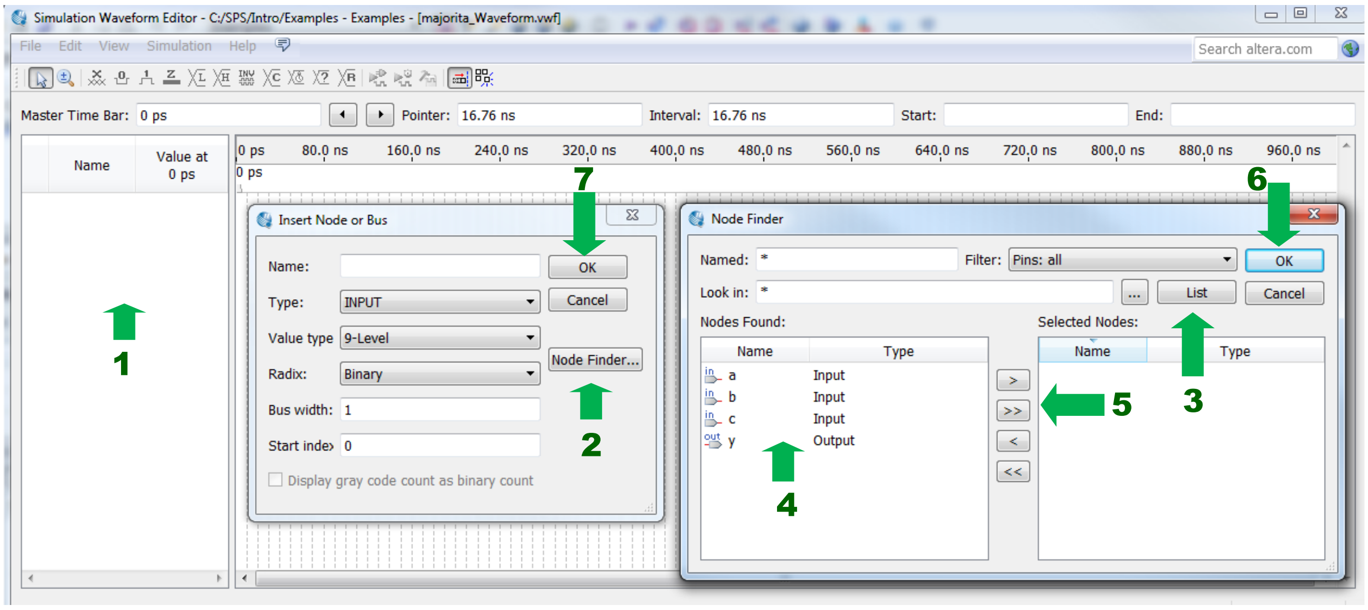


End Time: 1.0 us

To do so, we pop-up the simulation time dialog by Edit->Set End Time...

Default value is 1 µs and maximum ent time 100 µs.

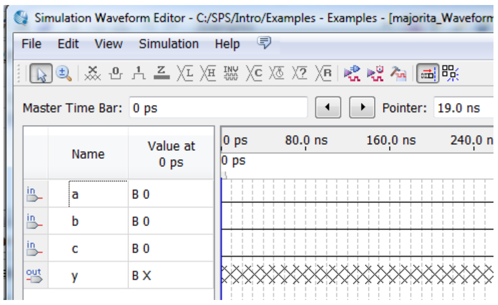
Now we follow the steps:



1. In menu, we choose Edit->Insert->Insert Node or Bus
   or we do double click by left mouse in Name window, see arrow 1.
2. In dialog Insert Node or Bus, we can enter one identifier of input or output, then confirm and pop-up dialog again. It is not possible to define their list.
   If we want to save work, we search for all inputs and outputs by pressing [Node Finder…].
3. In dialog Node Finder, we let default *, which means all, and filter Pins: all, i.e., we want inputs and outputs of our circuit. We only click [List].

4. We either select inputs and outputs in Nodes Found by Ctrl + left-mouse and use [>] button to copy them into Selected Nodes,
5. or we click [>>] button to copy all, what is the case of our majorita circuit.
6. By [OK] button, we close Node Finder dialog.
7. By [OK] button, we also close Insert Node or Bus dialog.

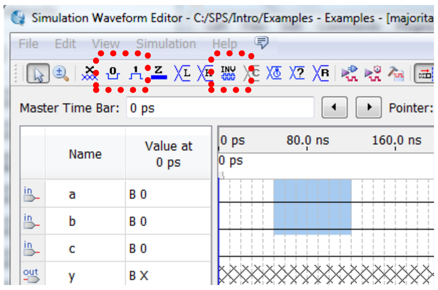
If you did everything correctly, the selected signals appeared in the window Simulation Waveform Editor :


Simulation Waveform Editor - C:/SPS/Intro/Examples - Examples - [majorita_Waveform
File Edit View Simulation Help
Master Time Bar: 0 ps
Pointer: 19.0 ns
0 ps 80.0 ns 160.0 ns 240.0 n
0 ps

Name	Value at 0 ps
a	B 0
b	B 0
c	B 0
y	B X

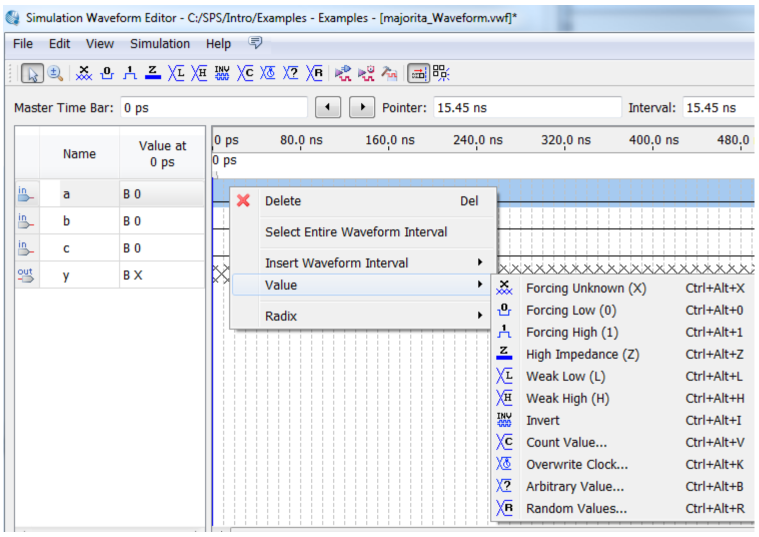
If we see signals in a different order than we wish we can rearrange them. Just drag the rows in the Name panel to other positions by mouse.

Now we draw the test signals for the inputs. Left-click to select the entire signal by clicking on its name, or a part of one or more signals to be changed at once. We press any of the marked buttons to define 0, 1, or inverse already drawn waveform.


Simulation Waveform Editor - C:/SPS/Intro/Examples - Examples - [majorita
File Edit View Simulation Help
Master Time Bar: 0 ps
Pointer:
0 ps 80.0 ns 160.0 ns
0 ps

Name	Value at 0 ps
a	B 0
b	B 0
c	B 0
y	B X

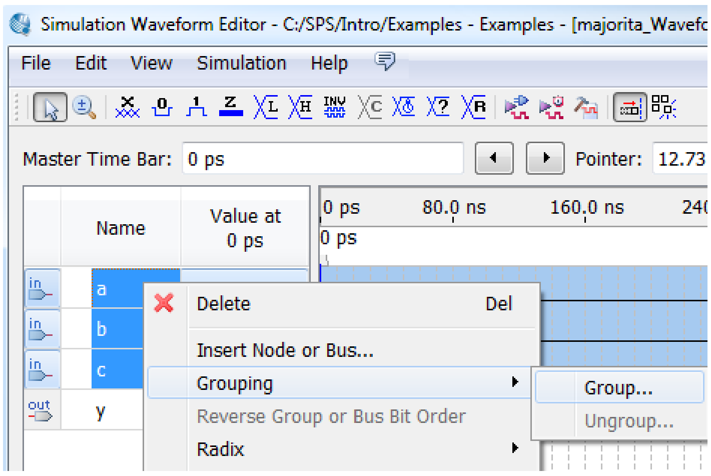
Alternatively, we can insert signals via the context menu (right mouse), or from the toolbar, or from the Edit-> Value menu, or by keyboard shortcuts.

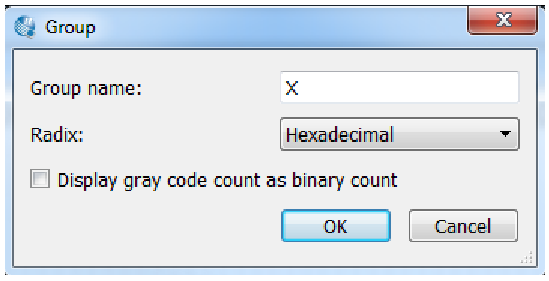

Simulation Waveform Editor - C:/SPS/Intro/Examples - Examples - [majorita_Waveform.vwf]*
File Edit View Simulation Help
Master Time Bar: 0 ps
Pointer: 15.45 ns
Interval: 15.45 ns
0 ps 80.0 ns 160.0 ns 240.0 ns 320.0 ns 400.0 ns 480.0
0 ps
Delete Del
Select Entire Waveform Interval
Insert Waveform Interval
Value
Radix
Forcing Unknown (X) Ctrl+Alt+X
Forcing Low (0) Ctrl+Alt+0
Forcing High (1) Ctrl+Alt+1
High Impedance (Z) Ctrl+Alt+Z
Weak Low (L) Ctrl+Alt+L
Weak High (H) Ctrl+Alt+H
Invert Ctrl+Alt+I
Count Value... Ctrl+Alt+V
Overwrite Clock... Ctrl+Alt+K
Arbitrary Value... Ctrl+Alt+B
Random Values... Ctrl+Alt+R

Name	Value at 0 ps
a	B 0
b	B 0
c	B 0
y	B X

If we want to enter all logic combinations, we use a counter, but first we have to create a group of signals.

1. In subwindow Name, we select signals for grouping, and by right mouse, we pop-up context menu, where we choose Grouping->Group… In the dialog, we enter any suitable name, such as X. Attention, we never choose something that looks like a keyword in VHDL or Verilog, such as in, out or wire; otherwise the simulation may fail. If we can also set a suitable radix, in which we prefer to list the group. We give OK.


Simulation Waveform Editor - C:/SPS/Intro/Examples - Examples - [majorita_Wavefo
File Edit View Simulation Help
Master Time Bar: 0 ps
Pointer: 12.73
Name
Value at 0 ps
0 ps 80.0 ns 160.0 ns 240
a
b
c
y
Delete Del
Insert Node or Bus...
Grouping
Reverse Group or Bus Bit Order
Radix
Group...
Ungroup...

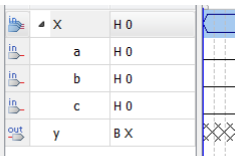

Group

Group name: X

Radix: Hexadecimal

Display gray code count as binary count

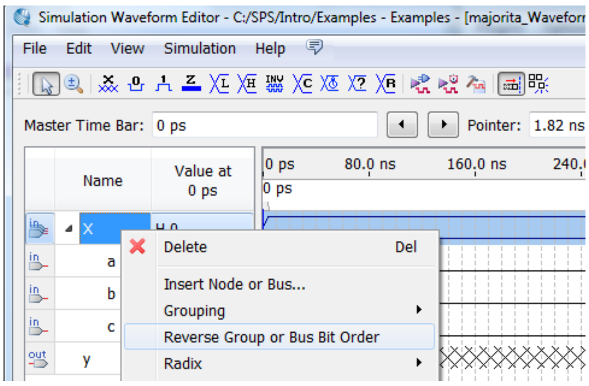
2. We created the group of signals:

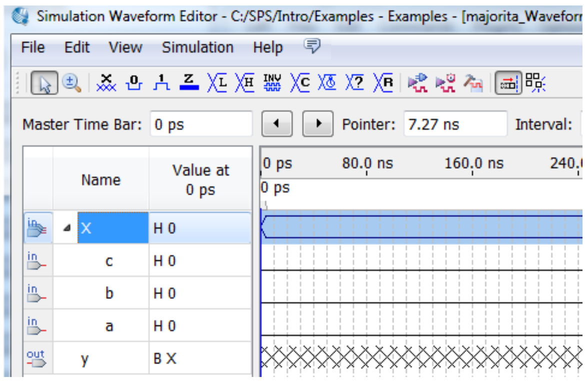


X		H 0
	a	H 0
	b	H 0
	c	H 0
y		B X

3. The counter counts from the lowest signal, which has the most repeated changes, to the others above it. If we want to reverse the weights of wires, we can do this by selecting a group and choosing either from menu Edit->Reverse Group or Bus Bit Order

or by context menu of the group name.

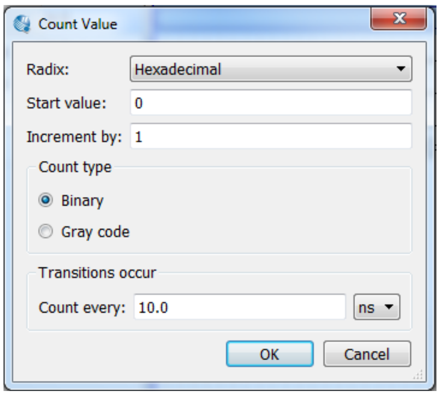

Simulation Waveform Editor - C:/SPS/Intro/Examples - Examples - [majorita_Wavefor
File Edit View Simulation Help
Master Time Bar: 0 ps
Pointer: 1.82 ns
Name
Value at 0 ps
0 ps 80.0 ns 160.0 ns 240.0
X
a
b
c
y
Delete Del
Insert Node or Bus...
Grouping
Reverse Group or Bus Bit Order
Radix


Simulation Waveform Editor - C:/SPS/Intro/Examples - Examples - [majorita_Wavefor
File Edit View Simulation Help
Master Time Bar: 0 ps
Pointer: 7.27 ns
Interval:
Name
Value at 0 ps
0 ps 80.0 ns 160.0 ns 240.0

X		H 0
	c	H 0
	b	H 0
	a	H 0
y		B X

By the same way, we can change dispaly radix anytime.

4. Select the group identifier (or select only a part of the group signal) and press the button on the toolbar, or from windows menu Edit->Value->Count Value.


Count Value

Radix: Hexadecimal

Start value: 0

Increment by: 1

Count type

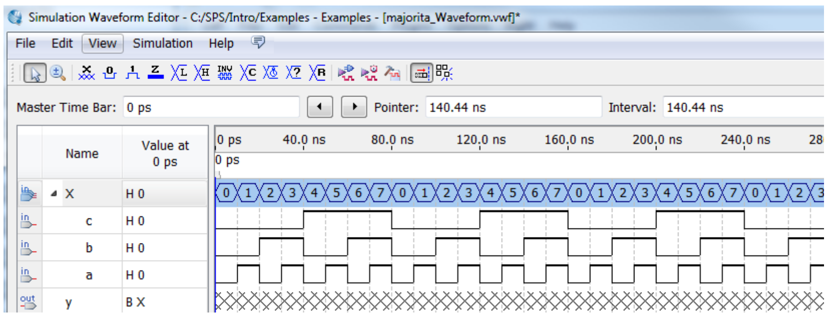
- Binary [✓]
- Gray code

Transitions occur

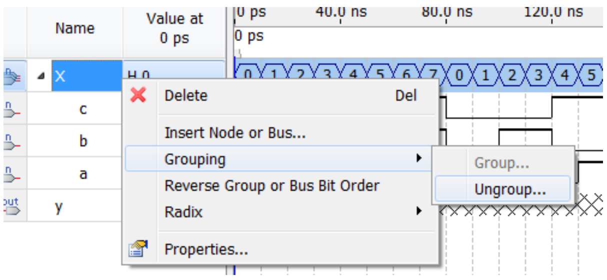
Count every: 10.0 ns

In the dialog that appears, fill counting period. The value of 10 ns is the default. We can leave for the majority. Select a counter type, either binary counting in the sequence of decimal numbers or Gray code with numbers arranged by the way that only one bit is changed at a time. Here, we also see the option to adjust a radix.

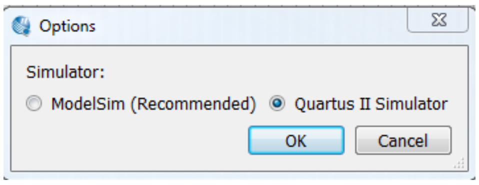
After [OK], the signals are filled with automatically created waveforms, see below. Ctrl + mouse wheel changes zoom. We can also adjust the zoom via main menu View->


Simulation Waveform Editor - C:/SPS/Intro/Examples - Examples - [majorita_Waveform.vwf]*
File Edit View Simulation Help
Master Time Bar: 0 ps
Pointer: 140.44 ns
Interval: 140.44 ns
Name
Value at 0 ps
X H 0
c H 0
b H 0
a H 0
y B X

5. We can then delete the group via the Grouping-> Ungroup context menu or leave it depending on us. Sometimes, but rarely, he simulator rejects the group name because it conflicts with something, so we have nothing else left than its ungrouping.



6. Now, we select the simulation type. We choose Simulation->Options from main menu:

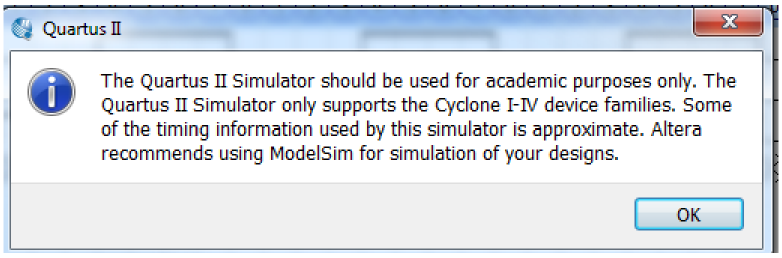

Options

Simulator:

ModelSim (Recommended) [✓ Quartus II Simulator]

OK Cancel

In the dialog, we prefer Quartus II Simulator for the University Program, which always works. Altera added it in Quartus at the request of the universities, so we confirm an education usage:


Quartus II

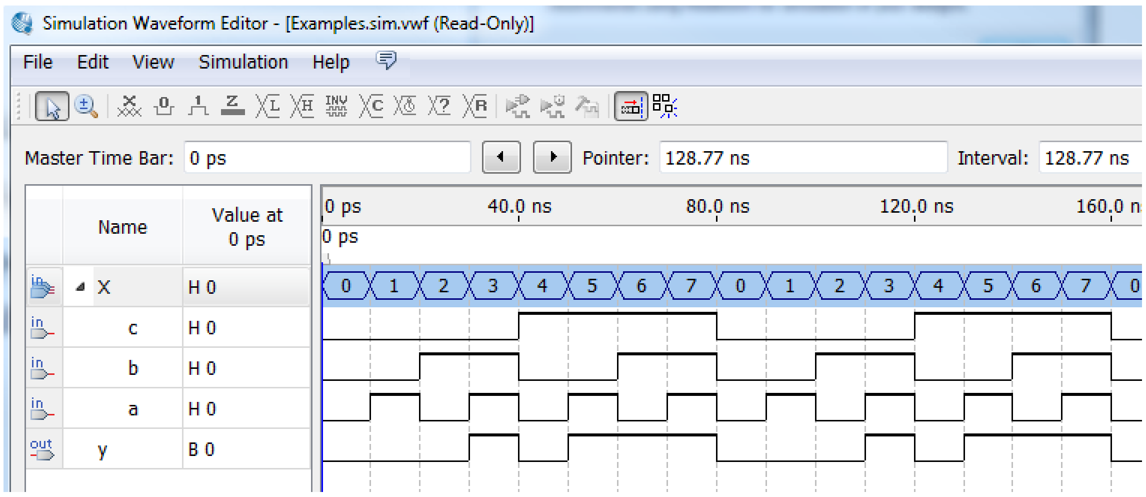
The Quartus II Simulator should be used for academic purposes only. The Quartus II Simulator only supports the Cyclone I-IV device families. Some of the timing information used by this simulator is approximate. Altera recommends using ModelSim for simulation of your designs.

OK

Of course, we can also try ModelSim later, if we have a properly created project and set paths to ModelSim-Altera: Tools-> Options-> EDA Tool Option (the default value created when installing Quartus on a C disk is C:\altera\13.0sp1\modelsim_ase\win32aloem).

7. We run: Simulation->Run Functional Simulation, which simulates logic circuits as logical operations ☺ The previous sentence looks strange, but it is not. Real logic circuits also have time characteristics such as delay time, causing numerous parasitic phenomena targeted by time simulations. These belong to more advanced levels, and we discuss them at the lecture. Meanwhile, all we need are logical operations.

8. The simulator builds the testbench program in Verilog and runs it. Then, we see a new window with results that we cannot edit, just view, as its title states. To change the waveforms and perform a different simulation, we must return to the original window and rerun the simulation.

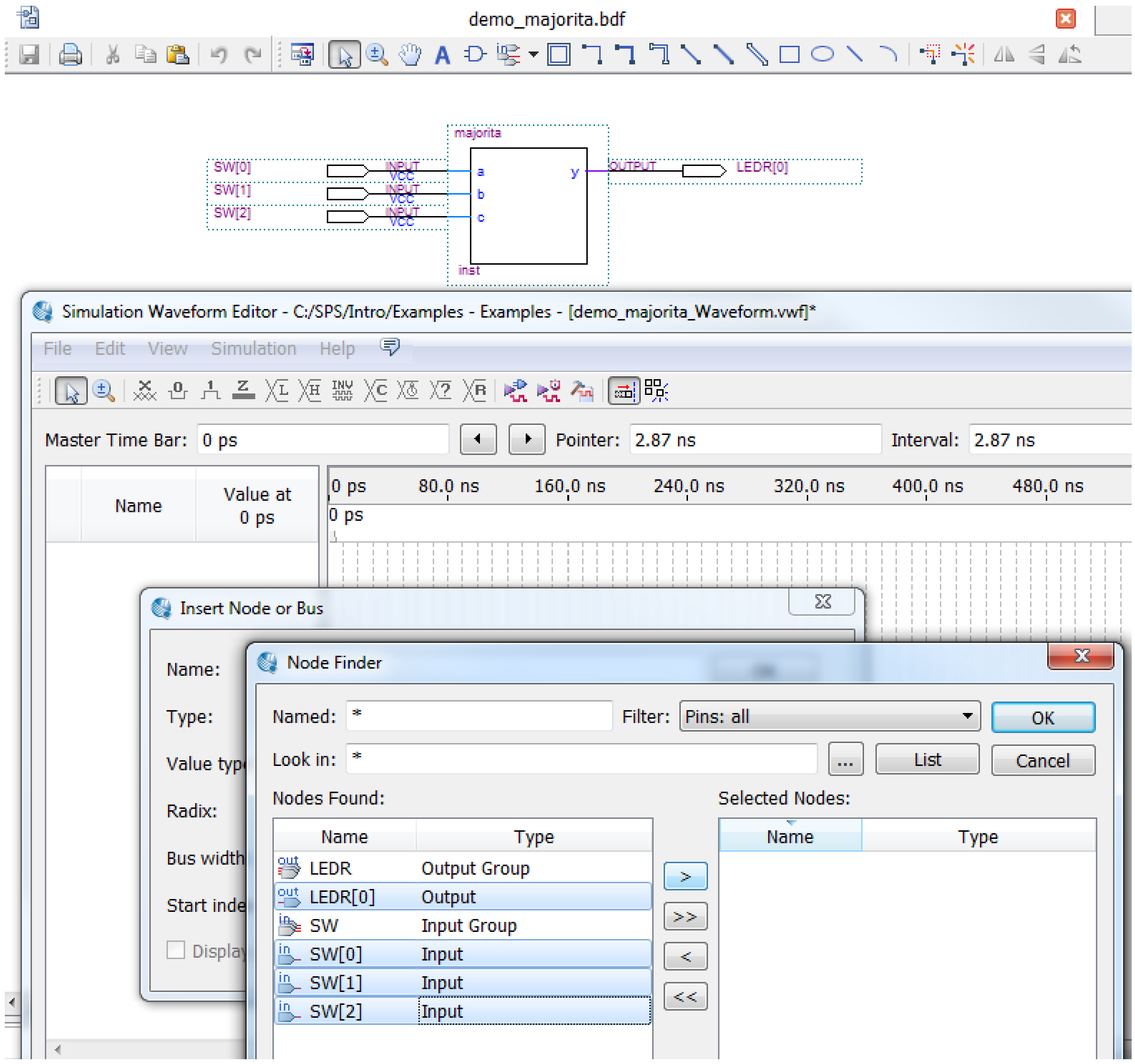


We can find other simulation possibilities in the guide Introduction to Quartus II Simulation on the webpage http://dcenet.felk.cvut.cz/edu/fpga/doc/Quartus_II_Simulation.pdf .

The built-in simulator serves for teaching purposes only. ModelSim allows more advanced circuit testing, but it already requires writing testbench files. The built-in simulator creates testbenches for us, but only with limits. It does not include all possible operations.

10.7 Inserting indexed signals

If we want to simulate a circuit with vectors, such as the one in Figure 19 on page 80, then Input and Output Groups also appear in the signal list. It is not possible to insert everything; we can choose either groups or individual signals, which we prefer here. The Quartus simulator converts our Vector Waveform File to testbench code used for actual calculation of the waveforms. In the pin assignments of Quartus definitions, the SW and LEDR groups have different ranges, which the built-in academic simulator sometimes can't handle, so the simulation fails. In other cases, we can insert groups.



Master Time Bar: 0 ps Pointer: 2.87 ns Interval: 2.87 ns

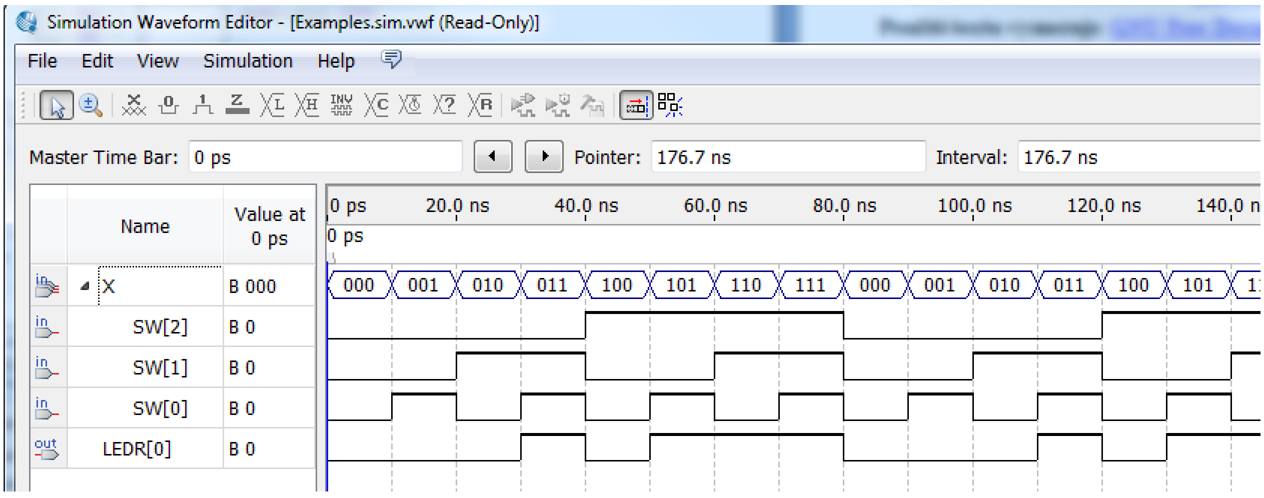
Named: * Filter: Pins: all

Look in: *

Nodes Found:

Name	Type
LEDR	Output Group
LEDR[0]	Output
SW	Input Group
SW[0]	Input
SW[1]	Input
SW[2]	Input

We join groups ourselves and insert the values into them; then we run the simulation.


Simulation Waveform Editor - [Examples.sim.vwf (Read-Only)]
File Edit View Simulation Help
0 ps 20.0 ns 40.0 ns 60.0 ns 80.0 ns 100.0 ns 120.0 ns 140.0 n
000 001 010 011 100 101 110 111 000 001 010 011 100 101

Master Time Bar: 0 ps Pointer: 176.7 ns Interval: 176.7 ns

Name	Value at 0 ps
X	B 000
SW[2]	B 0
SW[1]	B 0
SW[0]	B 0
LEDR[0]	B 0

11 Conclusion

I wrote the Czech version of this teaching material from July to August 2019 for students of Logic Systems and Processors course given at the Department of Control Engineering at CTU-FEE Prague. In two following months, I translated the text into English for our foreigner students.

Some confusion in the interpretation, misspellings, or other printer's gremlins could certainly survive many corrections. The author welcomes if you help him to catch them all.

Author: Richard Šusta, richard@susta.cz, http://susta.cz/

Homepage of English version: http://dcenet.felk.cvut.cz/edu/fpga/guides.aspx

Homepage of Czech original: http://dcenet.felk.cvut.cz/edu/fpga/navody.aspx