H-EHE-SOC – Assignment-2

A piano in VHDL

Remko Welling
HAN_UNIVERSITY OF APPLIED SCIENCES
Arnhem, the Netherlands,
April 29, 2025

Document history

Version	Date	Reviewer	Note/Changes
211	29-4-2025	WLGRW	Revision on assignment 0 and grammar corrections.
210	16-5-2023	WLGRW	Textual correction
209	19-5-2022	WLGRW	Removed artefacts in comments referring to previous design
208	17-5-2022	WLGRW	Addition to user manual of piano
207	25-5-2021	WLGRW	Minor changes to assignment 6 and 7
205	8-4-2021	WLGRW	Updates to facilitate the option of reusing a 7-segment driver.
204	2-2-2021	WLGRW	Release
201	3-11-2020	WLGRW	Start revision in preparation to SOC class 2021
103	8-6-2020	WLGRW	Corrections
102	13-5-2020	WLGRW	Corrected figure 24, Credits to Wesley Oldenburg
101	28-4-2020	WLGRW	Corrections
100	9-4-2020	WLGRW	Initial release and publication at #OO
029	9-4-2020	WLGRW	Review comments processed
027	7-4-2020	WLGRW	First version for review
005	1-4-2020	WLGRW	Intermediate version
004	3-3-2020	WLGRW	Continued
003	27-2-2020	WLGRW	Continued
000	16-2-2020	WLGRW	Start of document

Todo

Date	What	Impact
16-2-2020	Copyright statement shall be verified to meet HAN regulations.	Low

About this document

This document is using icons to indicate the priority or type of information. The used icons are:

Attention: Important information about the topic that will have significant effect.

Learning objective: What we will learn.

Information: Additional information about the topic.

Suggestion: Ideas to help or to point into a specific direction.

Further reading: There is more information available about this topic.

Responsible disclosure

Students and lecturers are encouraged to send in their experiences, problems, errors, any other observation while using this document. Please send your feedback to one of the lecturers, so we can improve this paper. Thank you.

Copyright

Disclaimer

The author of this paper reserves the right to apply changes at any time without prior notice. If any changes are made, the revised paper shall be posted on Onderwijs Online immediately. Please check the latest information posted herein to inform yourself of any changes.

1 Contents

2 Lab materials

This paragraph contains mandatory and optional hard- and software that is required for the assignments.

2.1 Literature

During lab classes the following book is used for reference:

Circuit Design and Simulation Using VHDL, 2nd Edition, Copyright 2010 MIT Press, by Volnei A. Pedroni; ISBN 978-0-262-01433-5. Available through this link (Link was verified 11-6-2019).

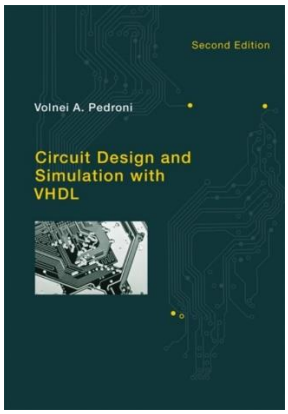

Figure 1: Theory book Pedroni

2.2 Hardware

During lab-classes the terasIC DE10-Lite Board will be used. See: terasIC or Intel (Links verified 12-6-2019).

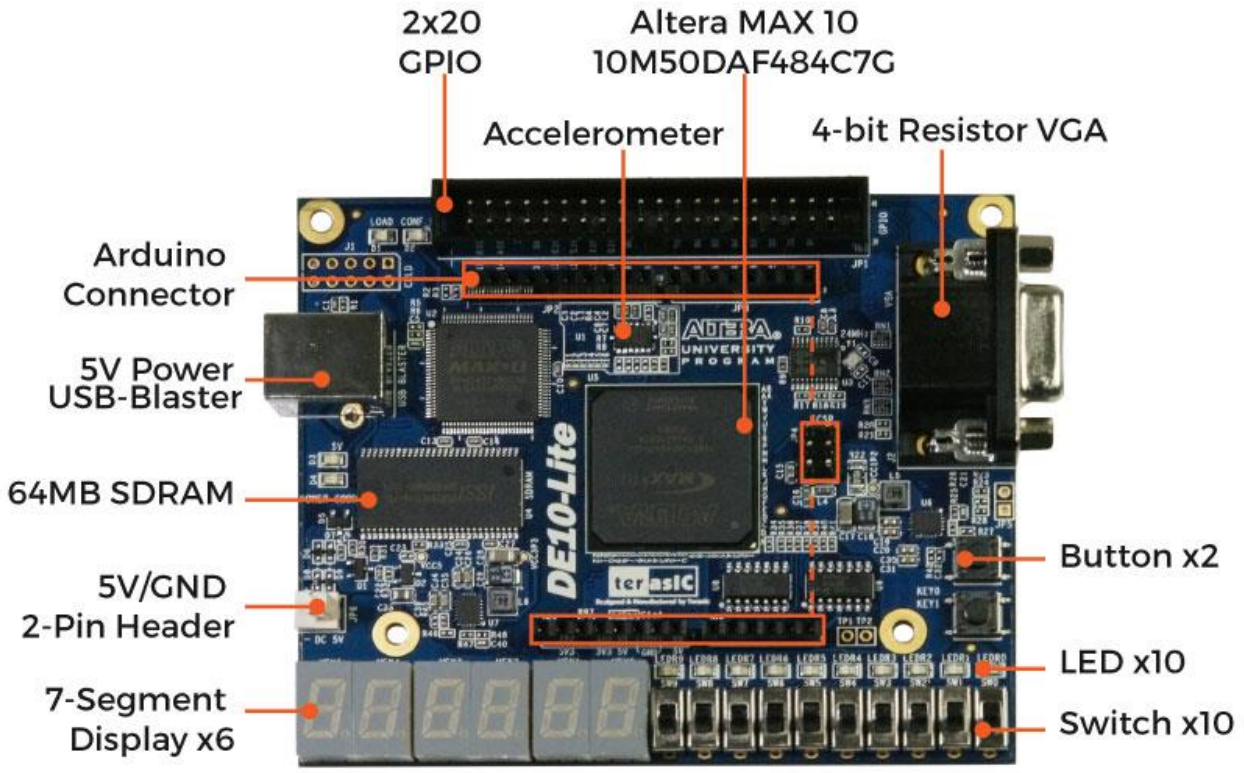

Figure 2: DE10-Lite prototype board (source: terasIC)

Usage of the board is described in the document "Logic Circuits - VHDL Practical Sessions Manual" that is available through Onderwijs Online.

In the following documents you will find a description of the FPGA board (=DE10-Lite board) which we use. Among other things, you can find the available buttons, keys, LED's and displays. Use it as a help for

finding the physical position of a component (SW, LEDR, KEY etc.) on the board. Get it directly from the web using the provide URL or download the pdf copy:

- DE10-Lite_User_Manual.pdf from Onderwijs Online.
- DE10-Lite user manual from Intel (verified 12-6-2019)

2.2.1 Obtain the board

There are two ways to obtain the DE10-Lite board:

1. Purchase through HAN ARLE: The DE10-Lite board can be purchased at HAN ARLE (through Henk Schepers or his colleagues) for € 65, -
2. Purchase trough terasIC: The DE10-Lite board can be purchased at terasIC for $ 55,- (academic price). Follow this link, keep a credit-card ready and plan for 3 weeks or more time to deliver at the address provided.

2.3 Software

2.3.1 IDE for FPGA

Intel® Quartus® Prime Design Software[1] is the design software that will be used to design circuits and program the DE10-Lite board. Quartus provides design entry, synthesis to optimization, verification, and simulation.

The instructions for software installation, hardware implementation, and lab files administration can be found in the following manual:

- Logic Circuits - VHDL Practical Manual v19_007.pdf

Download the following file which is needed for FPGA pin assignment of DE10-Lite board for all Quartus II projects. Its use is explained in the manual linked above. Consult the DE10-CV FPGA board manual to see which name is connected to which component on the board.

- pin_assignments_DE10_LITE_with_comment_v20190306_JGTP.qsf

2.3.2 GIT

The project files for the assignments can be downloaded from gitlab.

- https://gitlab.com/wlgrw/han-soc-assignment-vhdl-piano

It is recommended to download and install TortoiseGit to allow versioning of your assignment files.

Download TortoiseGit here: https://tortoisegit.org/

[1] Source: https://nl.mouser.com/new/Intel/intel-quartus/

3 SOC Assignment overview

In this assignment we will create an electronic piano using a PS/2 keyboard, a Terasic De10-Lite FPGA prototype board, and a small loudspeaker. The piano is represented using and input-process-output diagram in Figure 3.

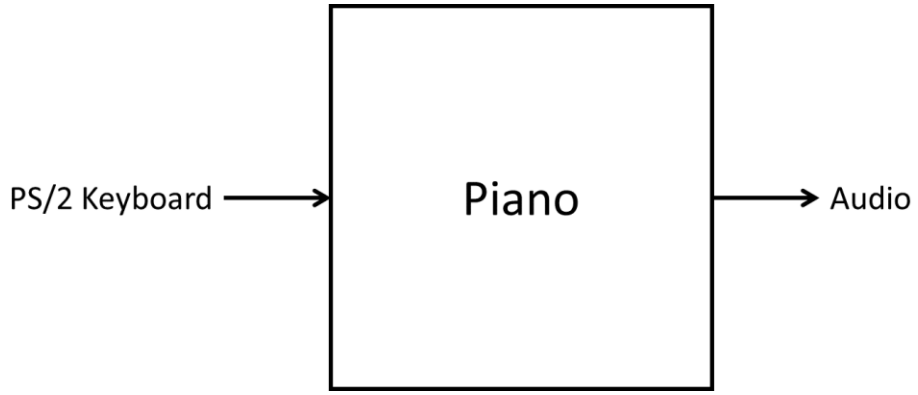

Figure 3: Input- Process- Output diagram of the VHDL piano

A PS/2 keyboard shall be connected to the De10-Lite. For this an Arduino shield shaped interface board is used that will piggyback on the Arduino connectors of the DE10-lite. This board is called “the Piano-shield”. The piano shield carries a speaker that will make the keystrokes on the piano audible and a PS/2 female min-din connector to which the PS/2 keyboard can be connected.

Using a PS/2 to USB adapter it is possible to replace the PS/2 keyboards that are rare by a generic available USB keyboard.

The PS/2 keyboard will substitute a normal instrument keyboard as being shown in Figure 7 in paragraph 4 on page 10. In this figure the notes are written on the black and white keys.

The PS/2 keyboard which we will use for this assignment has no white and black keys and the keys are not arranged as on an instrument like a piano or organ. Therefore, we will use the keys of the PS/2 keyboard in a manner that represents the instrument keyboard. This is visualized in Figure 8 paragraph 4 on page 10.

With this layout the black keys are on the row with the number-keys while the white keys are on the row with the letters “QWERTY”.

With a real instrument the musician can press multiple tones at the same time but that will not be possible with this implementation of the piano. Therefore, we will not assess the student on their musical qualities.

Background knowledge on Scales (Suits, 2020):

- As scale consist of 12 full and 12 half tones. The tones are A, Ais, B, C, Cis, D, Dis, E, F, Fis, G, Gis which than repeats as A+, Ais+, B+ and so on.
- Tone 'A' has a frequency of 440 Hertz.
- Tone 'A+' is 1 octave higher and has a frequency of 880 Hz.
- There is a constant such that $\text{Tone}_i = C \times Tone_i$, while $Tone_i + 12 = 2 \times Tone_i$
- Therefore: $Tone_i = C\big(C(C \dots (C \times Tone_i) \dots)\big) = C12\ Tone_i$
- From these two formulas C can be retrieved. C is the 12th power root from the number 2.
- The result is C = 1,05946.

The architecture of the VHDL piano is visualized in Figure 4. The PS/2 keyboard data is fed in to the first component (*Readkey*) that processes the keystrokes and presents them to the *tone_generation* component. Using the keys pressed on the keyboard *tone_generation* generates the tones that are linked to the keys on the keyboard and steps up and down in tone height (octave steps up and down).

For debugging purposes, 4 times a 7-segment display is used to show the hexadecimal values of the keystrokes on the keyboard.

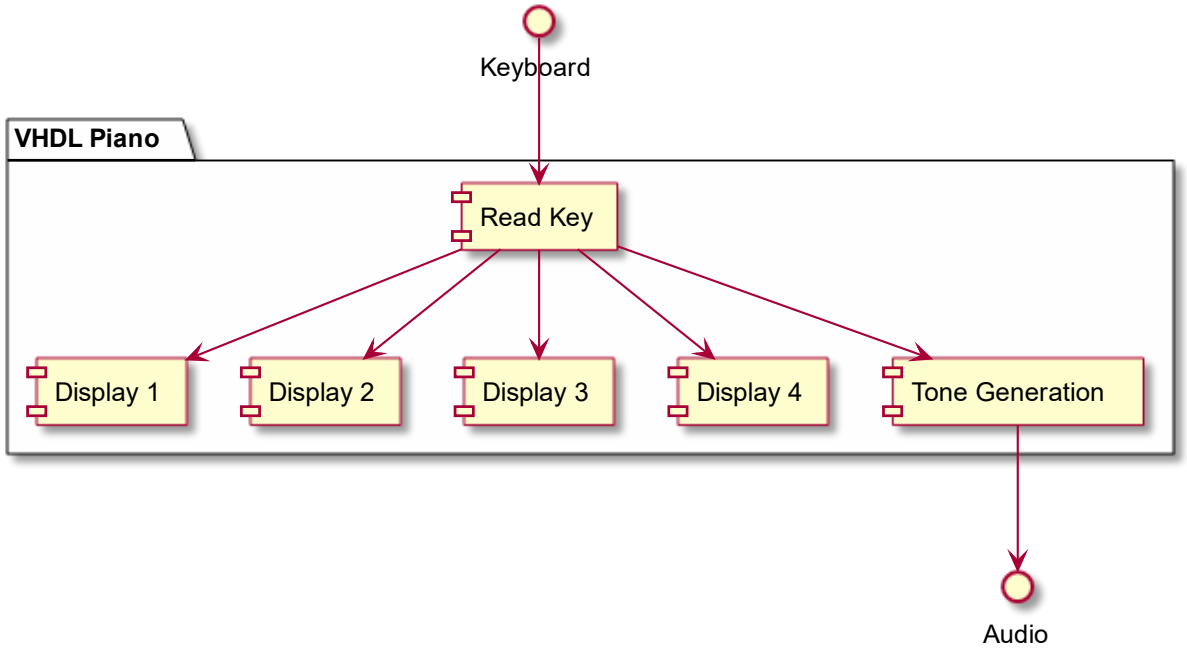

Figure 4: Architecture diagram of VHDL Piano

3.1 *readKey* component

The purpose of this component is to decode the serial data that is sent from the PS/2 keyboard and present it in a parallel to *tone_generation*.

Component *readKey* is built from 3 other components: *ShowKey* is converting serial data from the PS/2 asynchrony serial data to parallel data, *ConstantKey* presents the hexadecimal value for the key being pressed for the press duration, while component *clock-domain-crossing* is converting from one clock domain to another.

The architecture of *readKey* is shown in Figure 5. The *readKey* component has 5 outputs. The first output is the code of the key that is pressed on the keyboard. The other 4 ports are used to connect 4 7-segment displays to. These displays will show the code sent from the keyboard when pressed of release a key. Hen no key is pressed the displays will show 0x00 or are beings switched off.

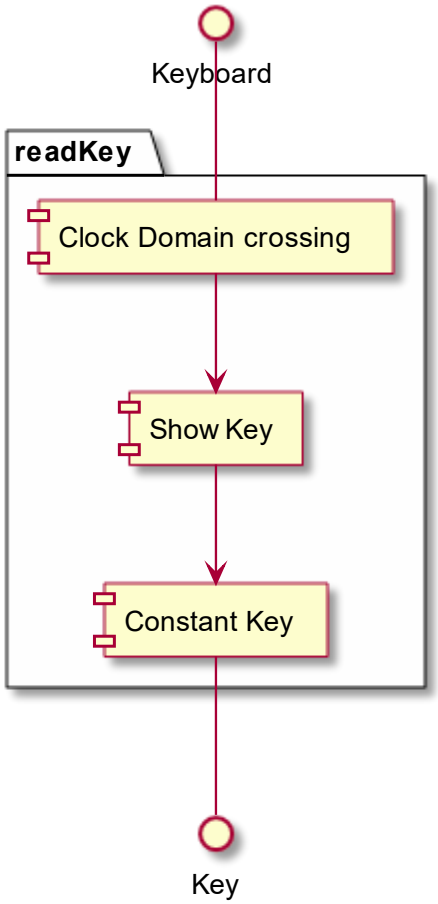

Figure 5: readKey architecture

3.2 *tone_generation* component

Component *tone_generation* received the information for the tone to be generated from *readKey* . For the duration a key is pressed, the corresponding tone is generated. *tone_generation* is divided into the components *Key2Pulselength*, *ClockGenerator* and *Key2Audio* which is generating the actual tone. See Figure 6 for the architecture of *tone_generation.*

Generation of the audio tone is done by *Key2Pulselength* which is providing the period time for the audio tone based on a given frequency generated by *ClockGenerator.* When the frequency generated by *ClockGenerator* is multiplied or divided by 2 the audio tone will increment or decrement by an octave.

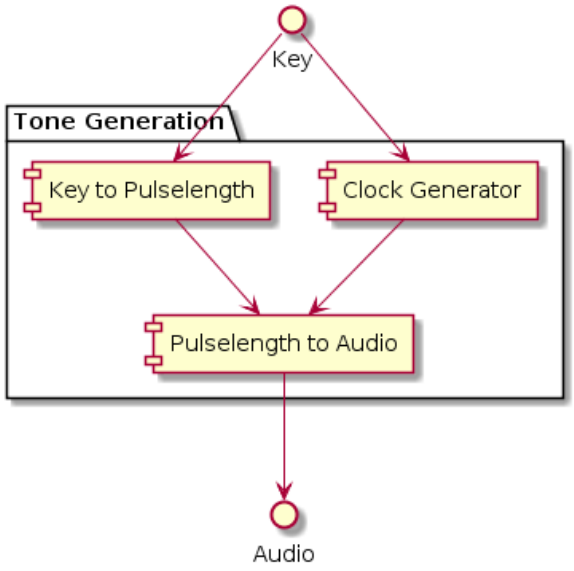

Figure 6: Architecture diagram Tone Generation

4 User interface and manual

When the DE10-Lite is programmed with the VHDL-piano-code as PS/2 compatible keyboard shall be connected using a Arduino shield shaped interface board that will piggyback on the Arduino connectors of the DE10-lite. This board is called the “Piano-shield”. The piano shield also has a speaker that will make the keystrokes on the keyboard audible.

The PS/2 keyboard will substitute a normal instrument keyboard as being shown in Figure 7. In this figure the primary notes are written on the black and white keys.

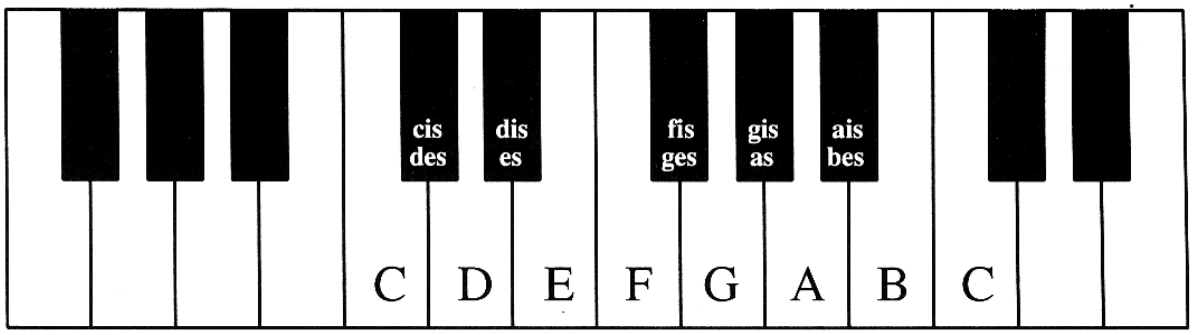

Figure 7: Instrument keyboard with notes

The PS/2 keyboard which we will use for this assignment has no white and black keys and the keys are not arranged as on an instrument like a piano or organ. Therefore, we will use the keys of the PS/2 keyboard in a manner that represents the instrument keyboard as presented in Figure 8.

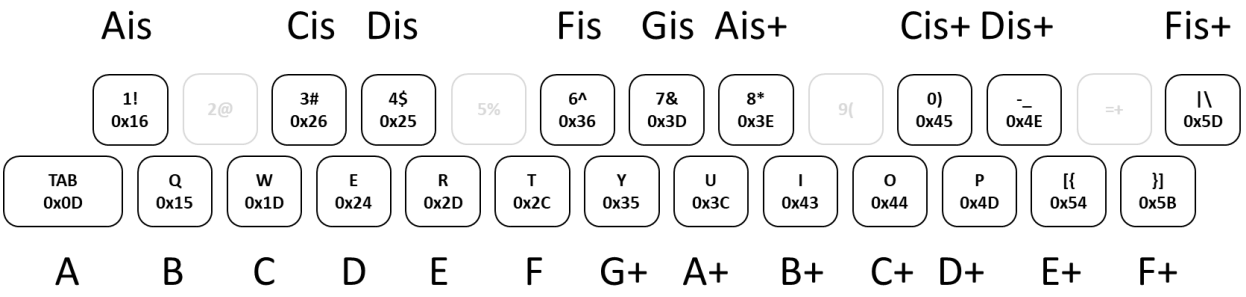

Figure 8: Notes and key arrangement of the PS/2 keyboard.

With this layout the black keys are on the row with the number-keys while the white keys are on the row with the letters “QWERTY”.

The tone keyboard is increase with one octave by pressing key A and lowered with one octave pressing key Z.

The DE10-Lite FPGA prototype board is expanded with a shield in an Arduino formfactor that has the PS/2 mini-din connector and a speaker. The user interface of VHDL piano is visualised in Figure 9.

Each keystroke, when mapped to the keys that will generate a tone, will result in the corresponding tone to be heard on the speaker. Audio is also displayed on LED1. The hexadecimal value of current key being pressed will be displayed on HEX2 and HEX3 while the hexadecimal value representing the previous key will be displayed on HEX0 and HEX1.

KEY1 of the DE10-Lite board will reset the VHDL piano. A reset is indicated with LED0 being off.

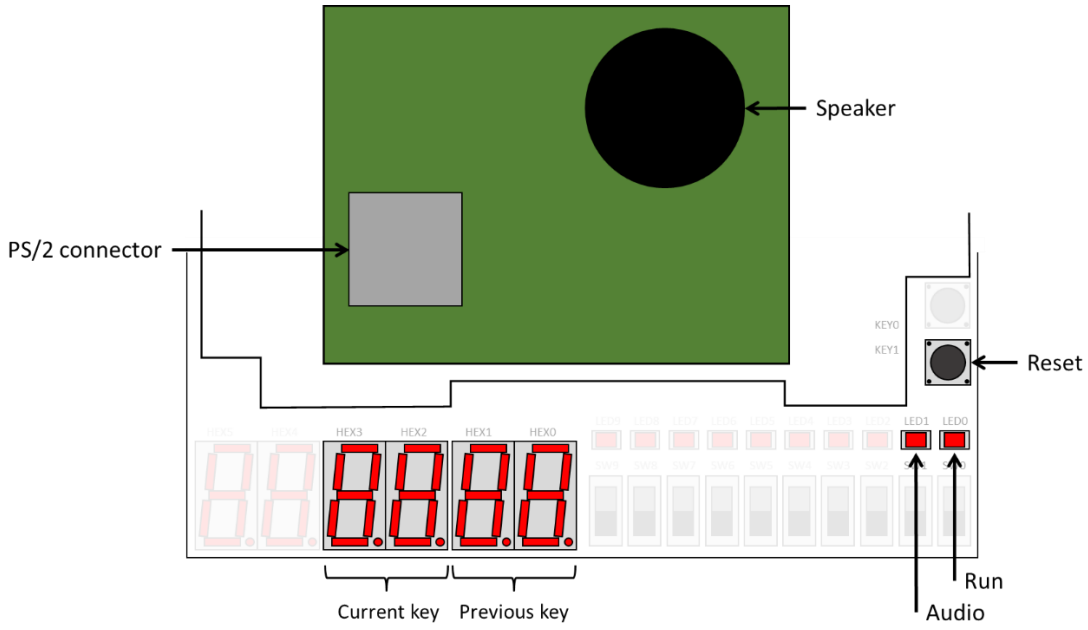

Figure 9: User interface for VHDL piano in DE10-Lite FPGA board

5 Assignments

The construction of the VHDL piano is partitioned into 7 assignments. Each assignment will result in a built component or in combining 2 or more components into another component. In the last assignment all components will be built together into the top-level entity of the VHDL piano.

The assignments are structured in 3 parts.

1. Theory
2. Design description
3. Construction directives, using a Work breakdown structure (WBS), and testing the operation using either waveform analysis or a testbench.

5.1 Assignment 0: 7-segment driver and structural VHDL

Assignment 0 is divided into part a and part b. With part a the 7-segment driver is implemented while with part b, the 7-segment driver is reused as a component in 2 different ways.

In this assignment students will learn to use 7 out of 4 decoders as a component in a VHDL architecture. Also, the use of the GENRATE statement is exercised.

If you have a 7-segment driver from a previous assignment you may want to reuse this drive for this assignment.

5.1.1 Theory

5.1.1.1 7-segement displays

A Display Decoder (7-segment decoder) is a combinational circuit which decodes a 4-bit input value into 7 of output lines to drive a display (Electronic Tutorials, 2019). 7-segment LED displays provide a convenient way of displaying information or digital data in the form of numbers, letters or even alpha-numerical characters.

Typically, 7-segment displays consist of seven individual coloured LED's (called the segments), within one single display package. To produce the required numbers or HEX characters from 0 to 9 and A to F respectively, on the display the correct combination of LED segments need to be illuminated. BCD to 7-segment Display Decoders such as the 74LS47 do just that.

A standard 7-segment LED display generally has eight (8) input connections, one for each LED segment and one that acts as a common terminal or connection for all the internal display segments. Some single displays have an input to display a decimal point in their lower right- or left-hand corner.

In electronics there are two important types of 7-segment LED digital display:

1. The Common Cathode Display (CCD) – In the common cathode display, all the cathode connections of the LED's are joined to logic "0" or ground. The individual segments are illuminated by application of a "HIGH", logic "1" signal to the individual Anode terminals.
2. The Common Anode Display (CAD) – In the common anode display, all the anode connections of the LED's are joined together to logic "1" and the individual segments are illuminated by connecting the individual Cathode terminals to a "LOW", logic "0" signal.

The displays that are used on the DE10-Lite FPGA prototype board are of the type of CAD. The display and the circuit to switch the individual LEDs on and off is shown in Figure 10.

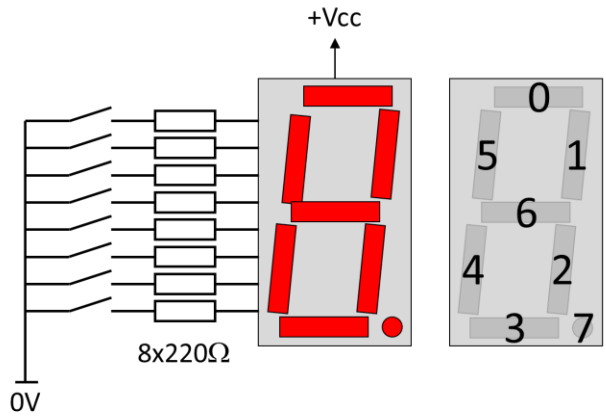

Figure 10: 7-segment display with example circuit for testing and segment numbering.

The VHDL code presented in this document is designed to deliver decoder functions that can be used in in combination with any FPGA development board that used CAD type displays like the DE10-Lite.

5.1.1.2 Structural VHDL

Component instantiation is a concurrent statement that can be used to connect circuit elements at a very low level or most frequently at the top level of a design. A VHDL design description written exclusively with component instantiations is known as Structural VHDL. Structural VHDL defines behaviour by describing how components are connected.

Study Pedroni paragraph "8.3 COMPONENT"

The most common way of writing an instantiation, is by declaring a component for the entity you want to instantiate. There are a few different places where you can write your component declaration.

Method 1: COMPONENT instantiation

1. Declare a VHDL Component in the VHDL Architecture

You can write the VHDL component declaration in the declarative part of the architecture. The advantage of this style is that you can see the generic list and port list of your component in the same file you are typing your instantiation. On the downside, if you want to instantiate that entity many times, you will have many copies of the component in many different places.

2. Component instantiation

The instantiation statement connects a declared component to signals in the architecture. The instantiation has 3 key parts:

- Label - Identifies unique instance of component
- Component Type - Select the desired declared component
- Port Map - Connect component to signals in the architecture

An example of COMPONENT instantiation is found in Code snippet 1.

Code snippet 1: Code example using COMPONENT instantiation

```
-- ----------------------------------------------------------------------
LIBRARY ieee;
USE     ieee.STD_LOGIC_1164.all;
-- ----------------------------------------------------------------------
ARCHITECTURE LogicFunction OF clock_domain_crossing IS

   -- Declaration of COMPONENT in declarative part of ARCHITECTURE
   COMPONENT flipflop PORT (
      D,
      clk,
      reset : in STD_LOGIC;
      Q     : out STD_LOGIC);
   END COMPONENT;

   -- signals declarations
   SIGNAL    kbdatasignal, kbclocksignal : STD_LOGIC;

BEGIN
   -- component instantiations
   c0: flipflop PORT MAP (kbdata,        clk, reset, kbdatasignal );
   c1: flipflop PORT MAP (kbdatasignal,  clk, reset, kbdatasync );
   c2: flipflop PORT MAP (kbclock,       clk, reset, kbclocksignal );
   c3: flipflop PORT MAP (kbclocksignal, clk, reset, kbclocksync );

END LogicFunction; -- end architecture
-- ----------------------------------------------------------------------


-- ----------------------------------------------------------------------
LIBRARY ieee;
USE     ieee.STD_LOGIC_1164.all;
-- ----------------------------------------------------------------------
ENTITY flipflop IS -- generic function of flipflop
   PORT(
      D     : in  STD_LOGIC;
      clk   : in  STD_LOGIC;
      reset : in  STD_LOGIC;
      Q     : out STD_LOGIC
      );
END flipflop;
-- ----------------------------------------------------------------------
ARCHITECTURE Logic OF flipflop IS
BEGIN
   flipflop_process: PROCESS (clk, reset) IS
   BEGIN
      IF reset = '0' THEN
         Q <= '0';
      ELSIF rising_edge(clk) THEN
         Q <= D;
      END IF;
   END PROCESS;
END Logic;
-- ----------------------------------------------------------------------
```

Method 2: ENTITY instantiation

Since VHDL'93, you really don't need VHDL components anymore. You can instantiate the ENTITY directly as presented in Code snippet 2:

Code snippet 2: Example code using ENTITY Instantiation

```
ARCHITECTURE structure OF orgeltje IS
   -- declarative part of ARCHITECTURE
BEGIN
   -- create 7-segement decoders and connect to signals and ports
   Display0 : work.seg7dec port map ( s_sw0, HEX0 ); -- 7-segment decoder 0
   Display1 : work.seg7dec port map ( s_sw1, HEX1 ); -- 7-segment decoder 1
   Display2 : work.seg7dec port map ( s_sw2, HEX2 ); -- 7-segment decoder 2
   Display3 : work.seg7dec port map ( s_sw3, HEX3 ); -- 7-segment decoder 3
END structure;
```

In this case the ENTITY *seg7dec* is declared in a separate VHDL-file. The instantiation statement is the same as being used with COMPONENT instantiation where the component type is prefixed with "work." to indicate that the ENTITY mentioned here is within the scope of the project.

See paragraph "8.3 Component" in Pedroni on page 203.

5.1.1.3 Generate statement

The generate statement is an efficient way to specify designs with a regular structure. It provides a mechanism for conditional compilation.

Study Pedroni paragraph "5.5 The GENERATE Statement"

The generate statement simplifies description of regular design structures. Usually, it is used to specify a group of identical components using just one component specification and repeating it using the generate mechanism. (Renerta, 2020)

A generate statement consists of three main parts:

- generation scheme (either for scheme or if scheme);
- declarative part (local declarations of subprograms, types, signals, constants, components, attributes, configurations, files and groups);
- concurrent statements.

The generation scheme specifies how the concurrent structure statement should be generated. There are two generation schemes available: *for* scheme and *if* scheme.

The *for*-generation scheme is used to describe regular structures in the design. In such a case, the generation parameter and its scope of values are generated in similar way as in the sequential loop statement.

The example in Code snippet 3 presents VHDL code that connects 4 signals from PORT **SW** to 4 SIGNALs from SIGNAL **s_sw0**. In Code snippet 4 the VHDL code from Code snippet 3 is replaced by a GENERATE

statement. While in this example the efficiency gained from replacing 4 assignment statements by a GENERATE statement is negligible, with bigger and more complex structures the use of GENERATE is beneficial.

Code snippet 3: Connecting PORS to SIGNALS using VHDL ASSIGNMENT statement

```
s_sw0(0) <= SW(0);
s_sw0(1) <= SW(1);
s_sw0(2) <= SW(2);
s_sw0(3) <= SW(3);
```

Code snippet 4: VHDL code using GENERATE statement.

```
G0: FOR i IN 0 TO 3 GENERATE
    s_sw0(i) <= SW(i);
END GENERATE;
```

5.1.2 Design

Binary numbers are made up using 4 data bits (a *nibble* or *half a byte*) that range from 0 through to F, shown in Figure 11.

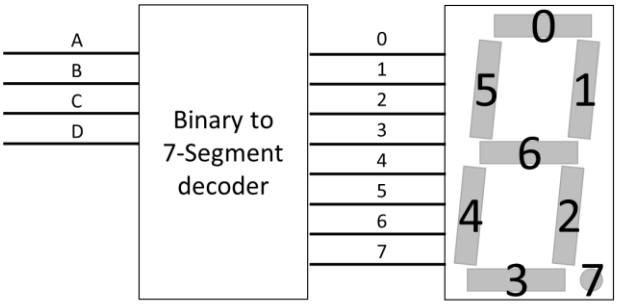

Figure 11: Signals involved.

To display the number “3” for example, segments 0, 1, 2, 3 and 6 would need to be illuminated. If we wanted to display a different number or letter, then a different set of segments would need to be illuminated. Then for a 7-segment display, we can produce a truth table giving the segments that need to be illuminated to produce the required character as shown in Table 1.

Table 1: Conversion table form binary to 7-segment

Meaning	Dot	D	C	B	A	0	1	2	3	4	5	6	7	Display
zero	X	0	0	0	0	0	0	0	0	0	0	1	0	0
one	X	0	0	0	1	1	0	0	1	1	1	1	0	1
two	X	0	0	1	0	0	0	1	0	0	1	0	0	2
three	X	0	0	1	1	0	0	0	0	1	1	0	0	3
four	X	0	1	0	0	1	0	0	1	1	0	0	0	4
five	X	0	1	0	1	0	1	0	0	1	0	0	0	5
six	X	0	1	1	0	0	1	0	0	0	0	0	0	6
seven	X	0	1	1	1	0	0	0	1	1	1	1	0	7
eight	X	1	0	0	0	0	0	0	0	0	0	0	0	8
nine	X	1	0	0	1	0	0	0	0	1	0	0	0	9
a	X	1	0	1	0	0	0	0	1	0	0	0	0	a
b	X	1	0	1	1	1	1	0	0	0	0	0	0	b
c	X	1	1	0	0	0	1	1	0	0	0	1	0	c
d	X	1	1	0	1	1	0	0	0	0	1	0	0	d
e	X	1	1	1	0	0	1	1	0	0	0	0	0	e
f	X	1	1	1	1	0	1	1	1	0	0	0	0	f
off	0	X	X	X	X	X	X	X	X	X	X	X	0	
on	1	X	X	X	X	X	X	X	X	X	X	X	1	.

Using K-map it is possible to derive logic functions that translates if a led shall be illuminated based on the input signals. In the case of the 7-segment display this will result in 84 K-maps and 32 SOP formulas. This is labour intensive and time consuming. Therefore, this design is using the WITH – SELECT statement in VHDL as presented in Code snippet 5.

Code snippet 5: VHDL WITH - SELECT example

```
-- Display decoder. This code is using "WITH - SELECT" to encode 6
-- segments on the HEX diplay. This code is using the CONSTANTS
-- that are defined at GENERIC.

    WITH input SELECT
         display(0 TO 6) <=
            hex_zero  WHEN "0000", -- 0
            hex_one   WHEN "0001", -- 1
            hex_two   WHEN "0010", -- 2
            hex_three WHEN "0011", -- 3
            hex_four  WHEN "0100", -- 4
            hex_five  WHEN "0101", -- 5
            hex_six   WHEN "0110", -- 6
            hex_seven WHEN "0111", -- 7
            hex_eight WHEN "1000", -- 8
            hex_nine  WHEN "1001", -- 9
            hex_a     WHEN "1010", -- a
            hex_b     WHEN "1011", -- b
            hex_c     WHEN "1100", -- c
            hex_d     WHEN "1101", -- d
            hex_e     WHEN "1110", -- e
            hex_f     WHEN "1111", -- f
            (OTHERS =>'1') WHEN OTHERS;
```

For maintenance purpose the VHDL code is using CONSTANTS for the signal that is send to the 7-segment display.

Code snippet 6: VHDL example using CONSTANT

```
--! Because the LEDs are controlled using inverted logic we have to apply a
--! '1' to switch the LED off.

--                                Segment number -> 0123456
--                                                  -------
   CONSTANT hex_zero:  STD_LOGIC_VECTOR(0 TO 6) := "0000001"; -- 0
   CONSTANT hex_one:   STD_LOGIC_VECTOR(0 TO 6) := "1001111"; -- 1
   CONSTANT hex_two:   STD_LOGIC_VECTOR(0 TO 6) := "0010010"; -- 2
   CONSTANT hex_three: STD_LOGIC_VECTOR(0 TO 6) := "0000110"; -- 3
   CONSTANT hex_four:  STD_LOGIC_VECTOR(0 TO 6) := "1001100"; -- 4
   CONSTANT hex_five:  STD_LOGIC_VECTOR(0 TO 6) := "0100100"; -- 5
   CONSTANT hex_six:   STD_LOGIC_VECTOR(0 TO 6) := "0100000"; -- 6
   CONSTANT hex_seven: STD_LOGIC_VECTOR(0 TO 6) := "0001111"; -- 7
   CONSTANT hex_eight: STD_LOGIC_VECTOR(0 TO 6) := "0000000"; -- 8
   CONSTANT hex_nine:  STD_LOGIC_VECTOR(0 TO 6) := "0000100"; -- 9
   CONSTANT hex_a:     STD_LOGIC_VECTOR(0 TO 6) := "0001000"; -- a
   CONSTANT hex_b:     STD_LOGIC_VECTOR(0 TO 6) := "1100000"; -- b
   CONSTANT hex_c:     STD_LOGIC_VECTOR(0 TO 6) := "0110001"; -- c
   CONSTANT hex_d:     STD_LOGIC_VECTOR(0 TO 6) := "1000010"; -- d
   CONSTANT hex_e:     STD_LOGIC_VECTOR(0 TO 6) := "0110000"; -- e
   CONSTANT hex_f:     STD_LOGIC_VECTOR(0 TO 6) := "0111000"  -- f
```

In this assignment a 7-segment driver will be used as a component. This component will be instantiated multiple times to drive the 4- HEX-displays. To drive the HEX displays the switches **SW**[0..9] will be used. See architecture diagram in Figure 12.

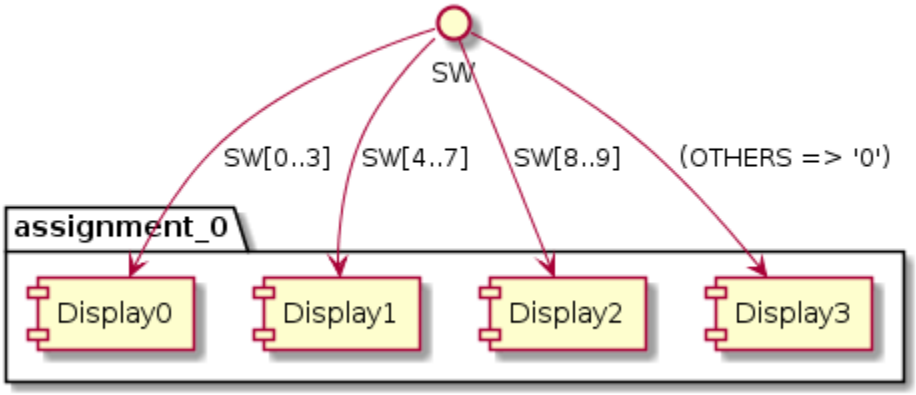

Figure 12: Assignment 0 Architecture diagram

Because the 7-segment driver will be reused in the VHDL piano the PORT definitions provided are mandatory to ensure connectivity. The input-process-output diagram with the required ports is given in Figure 13.

For the implementation of the ARCHITECTURE of the 7-segment driver the 7-segment driver that was designed for the ALU can be reused.

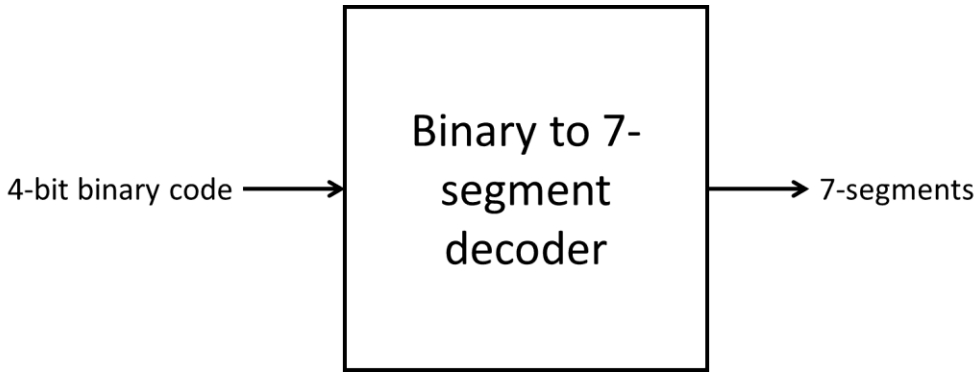

Figure 13: Input-process-output diagram for binary to 7-segment decoder

5.1.3 Realisation and verification

With this assignment a multiplexer will be implemented in VHDL to control the 7 segments of a 7-segmend LED-display using 4 control signals.

The operation of the implemented VHDL code is tested using waveform or by manually switching the switches to encode 4-bits binary data and read the characters from the display.

5.1.3.1 Activity 1

The following activities are mandatory to be followed while the whole assignment depends on the use of this repository.

- ☐ Download the files for assignment 0 from Gitlab: https://gitlab.com/wlgrw/han-soc-assignment-vhdl-piano
- ☐ Open with Quartus the project “seg7dec” in folder “assignment_0” or create a new project with the top-level ENTITY name *seg7dec*.
- ☐ Verify if the ENTITY *seg7dec* has implemented the PORTS according to the specification in Code snippet 7.

Code snippet 7: Library and ENTITY of component seg7dec

```
-- ----------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- ----------------------------------------------------------------------
ENTITY seg7dec IS
   port(
      C       : IN  STD_LOGIC_VECTOR(3 downto 0);
      display : OUT STD_LOGIC_VECTOR(0 to 7)
   );
END ENTITY seg7dec;
-- ----------------------------------------------------------------------
```

5.1.3.2 Activity 2

The following activities apply for the design of a new 7-segment driver. In the case of the reuse of an existing 7-segment driver other steps are applicable to the discretion of the student.

- ☐ Determine the functions of the display driver
- ☐ Design the characters to be displayed.
- ☐ Setup architecture
- ☐ Implement functions
- ☐ Implement the 7 out of 4 decoder or 7-segment display driver in the ARCHITECTURE of ENTITY seg7dec.

The use of CONSTANTS will improve the code quality.
See paragraph "2.6 GENERIC" of Pedroni.

- ☐ Compile the project
- ☐ Test the Implementation of the 7-segment driver display using the truth table for the 7 out of 4 decoder and a Waveform simulation.

For the verification of ENTITY *seg7dec* you may use a truth table to compare the results of the individual 7-segments at a given 4-bit input using a truth table where the value of the 4- and 7-bit vector is presented as a decimal value. A truth table can be found in Worksheet 1 on page 59.

- ☐ Test functions of the new display driver using Switches and a 7-segment display

5.2 Assignment 1: showKey

In this assignment we will create COMPONENT *showKey*. COMPONENT *showKey* is part of COMPONENT *readkey.*

showKey reads the serial data from the PS/2 keyboard and presents the serial bytes parallel to *constantKey*. See the architecture diagram in Figure 17.

With this assignment knowledge will be gained on the PS/2 interface, the sequential processing of serial data and converting to parallel data and testing and verification using a testbench.

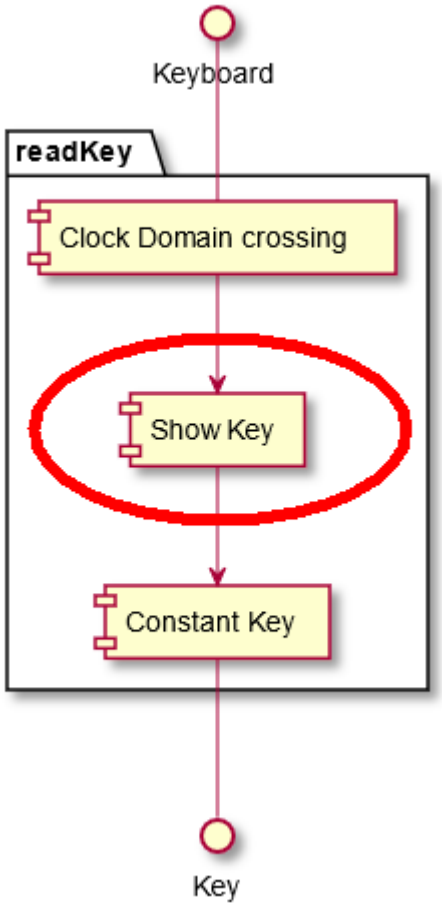

Figure 14: readKey architecture

5.2.1 Theory

5.2.1.1 PS/2 serial data

For the following description we rely on the documentation of PS/2 port (Wikimedia, 2020) and PS/2 Mouse/Keyboard Protocol (Chapweske, 2020).

The PS/2 interface is using a serial connection with the signals **kbclock** and **kbdata**. The data is sent over the signal **kbdata** while accompanied by the signal **kbclock** that informs when data on **kbdata** is valid. The Data line changes state when Clock is high, and that data is valid when Clock is low. See Figure 15

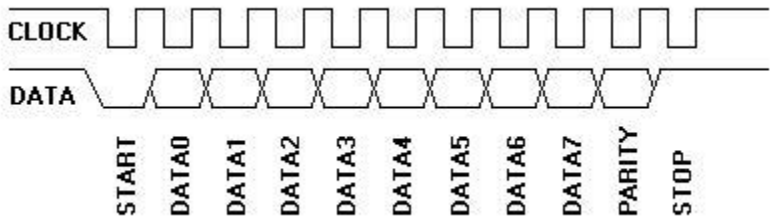

Figure 15: Device-to-host communication. (Chapweske, 2020)

- When a key is pressed on the keyboard a unique 8-bit code is generated for that key.
- When, for example the TAB-key is pressed, the code for the TAB-key, represented by the hex value 0x0D, is sent.
- When a key is pressed continuously the corresponding code will be generated repeatedly.
- When a key is released two codes are sent. These are 0xF0 to indicated that a key is released and the code for the key that was released. In this case TAB-key is released so the second code sent is 0x0D.
- When no key is pressed the keyboard is not generating a clock on **kbclock**
- The PS/2 protocol sends 11 bits when a key is pressed: 1 start bit, 8 bits keycode starting with LSB, 1 parity bit and 1 stop bit.
- **kbdata** is stable at the falling edge of **kbclock**.

5.2.1.2 Using a testbench for verification of a design.

The testbench is a specification in VHDL that plays the role of a complete simulation environment for the analysed system (Unit Under Test, UUT). A testbench contains both the UUT as well as stimuli for the simulation.

The UUT is instantiated as a component of the testbench and the architecture of the testbench specifies stimuli for the UUT's ports, usually as waveforms assigned to all output and bidirectional ports of the UUT.

The entity of a testbench does not have any ports as this serves as an environment for the UUT. All the simulation results are reported using the assert and report statements. (Renerta, 2020)

Altera and Intel documentation:

- Using ModelSim to Simulate Logic Circuits for Altera FPGA Devices: https://cseweb.ucsd.edu/classes/sp11/cse141L/lab1/Using_ModelSim.pdf
- https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_gs_msa_qii.pdf

Other documentation:

- Quartus II Testbench Tutorial: https://class.ece.uw.edu/271/peckol/doc/DE1-SoC-Board-Tutorials/ModelsimTutorials/QuartusII-Testbench-Tutorial.pdf
- Quartus II Tutorial, Paragraph 5. Simulating a design: https://class.ece.uw.edu/271/peckol/doc/DE1-SoC-Board-Tutorials/ModelsimTutorials/QuartusII-MSimTutorial.pdf

To aid the assignment the VHDL code for *showKey* is presented in Code snippet 8. The testbench is using the PROCEDURE-statement to generate the PS/2 protocol. Although the testbench is generating the parity bit, in the implementation of the VHDL piano we ignore this bit.

Do note that it is not the purpose to implement the testbench in hardware. Therefore, this testbench is using VHDL statements that cannot be synthesised. On the other hand, these statements simplify the generation of signals and present the outputs of the *showKey* component.

The ENTITY showKey_test:

The testbench is communicating with the outside world. As a result of that, there are no in- or output PORTs defined in the ENTITY. The declared SIGNALs are required within the testbench to connect *showKey* with all generated signals.

The procedure send_byte:

In our test environment we want to send the code of a key to component *showKey*. Procedure send_byte executes the PS/2 data protocol where keycode is the input for this procedure. This is followed by generation of parity bit and stop bit. To conclude, the procedure is generating the clock from the PS/2 keyboard.

Instantiation of showKey

Within the ARCHITECTURE of the testbench component *showKey* is instantiated first.

Process

Within the ARCHITECTURE, in a PROCESS statement three characters, 0xC1, 0X32 and 0x4C are being sent. When this is completed, the correct value shall be in signals **read_byte**, **scancode**, **dig0** and **dig1**.

Code snippet 8: Testbench for ENTITY showKey

```
-- Vhdl Test Bench template for design:  showKey
--
-- Simulation tool: ModelSim-Altera (VHDL)
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;

--ENTITY showKey_vhd_tst IS
ENTITY tb_showKey IS
END tb_showKey;

--ARCHITECTURE showKey_arch OF showKey_vhd_tst IS
ARCHITECTURE showKey_arch OF tb_showKey IS
-- constants
-- signals
SIGNAL byte_read : STD_LOGIC;
SIGNAL dig0      : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL dig1      : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL kbclock   : STD_LOGIC;
SIGNAL kbdata    : STD_LOGIC;
SIGNAL reset     : STD_LOGIC;
SIGNAL scancode  : STD_LOGIC_VECTOR(7 DOWNTO 0);

-- Procedure for this example.
PROCEDURE send_byte(
   CONSTANT byte     : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL pr_kbclock : OUT STD_LOGIC;
   SIGNAL pr_kbdata  : OUT STD_LOGIC
)
IS
   VARIABLE odd_parity : STD_LOGIC;
   VARIABLE data       : STD_LOGIC_VECTOR(10 DOWNTO 0);
BEGIN
   -- Generate paritybit
```

```
    odd_parity := '1';
    FOR i IN 7 DOWNTO 0 LOOP
       odd_parity := odd_parity XOR byte(1);
    END LOOP;
    data := '1' & odd_parity & byte & '0';
    -- Send off data
    FOR I IN 0 TO 10 LOOP
       pr_kbdata <= data(i);
       pr_kbclock <= '1';
       WAIT FOR 20 ns;
       pr_kbclock <= '0';
       WAIT FOR 20 ns;
    END LOOP;
    pr_kbclock <= '1';
END send_byte;

COMPONENT showKey -- UUT
    PORT (
       byte_read : BUFFER STD_LOGIC;
       dig0      : BUFFER STD_LOGIC_VECTOR(7 DOWNTO 0);
       dig1      : BUFFER STD_LOGIC_VECTOR(7 DOWNTO 0);
       kbclock   : IN STD_LOGIC;
       kbdata    : IN STD_LOGIC;
       reset     : IN STD_LOGIC;
       scancode  : BUFFER STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END COMPONENT;

BEGIN
    DUT : showKey -- UUT
    PORT MAP (
       -- list connections between master ports and signals
       byte_read => byte_read,
       dig0 => dig0,
       dig1 => dig1,
       kbclock => kbclock,
       kbdata => kbdata,
       reset => reset,
       scancode => scancode
    );

init : PROCESS
-- variable declarations
BEGIN
    -- code that executes only once
    reset <= '0';
    WAIT FOR 20 ns;
    reset <= '1';
    -- stuur eerste key 1C binair als 0001 1100 of hexadecimaal X"1C"
    Send_byte( X"1C", kbclock, kbdata ); -- 'X' is to show that value is HEX
    WAIT FOR 300 ns;
    -- stuur tweede key 32 binair als 0011 0010 of hexadecimaal X"32"
    Send_byte( X"32", kbclock, kbdata );
    WAIT FOR 300 ns;
    -- stuur tweede key 4D binair als 0100 1100 of hexadecimaal X"4D"
    Send_byte( X"4D", kbclock, kbdata );
    WAIT;
END PROCESS init;
```

```
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
   -- code executes for every event on sensitivity list
   WAIT;
END PROCESS always;

END showKey_arch;
```

5.2.2 Design

In the implementation of component *showKey* the PORT definitions in the ENTITY block of ENTITY *showKey* are provided with the input-process-output diagram in Figure 16.

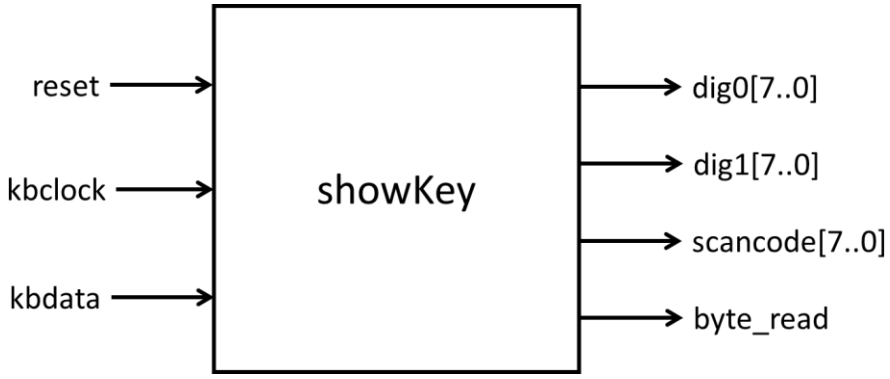

Figure 16: Input-process-output diagram for component showKey

For the implementation of *showKey* we use the description we provided with paragraph 5.2.1.1. The serial data is converted to a parallel byte.

When the value on port **scancode** is valid the signal on port **read_byte** goes up from '0' to '1' (rising edge). This happens when all 11 bits from the keyboard are received by *showKey.* As soon as the first bit from the keyboard is received by *showKey* port **read_byte** drops from '1' to '0'.

For the purpose of debugging ports **dig0** and **dig1** present the value of the current pressed key (**dig0**) and the last key pressed key (**key1**). When no key is pressed **dig0** will present the value 0x00.

When the **reset** is active (active **reset** is '0') **byte_read, scancode, dig0** and **dig1** are reset to 0x00 and '0'.

This description is visualized in the flow diagram of Figure 17.

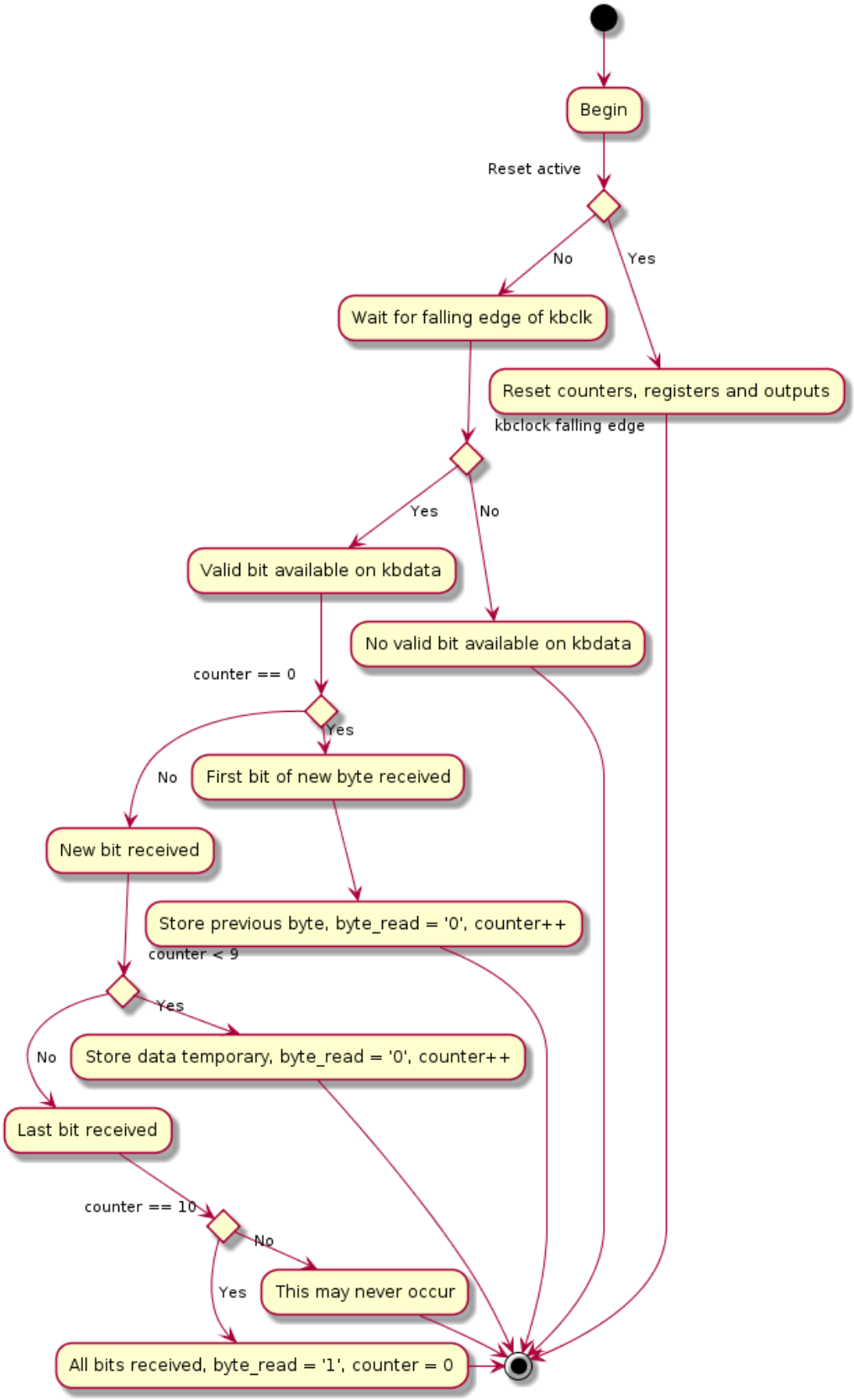

Figure 17: ShowKey Flowchart

5.2.3 Implementation of component showKey

- ☐ Download the files for assignment 1 from Gitlab: https://gitlab.com/wlgrw/han-soc-assignment-vhdl-piano

5.2.3.1 Part A of assignment 1: Implementation of ENTITY showKey

- ☐ Open with Quartus the project "showkey" in folder "assignment_1" or create a project with the top-level ENTITY named *showKey.*

- ☐ Verify if the ENTITY *showKey* has implemented the PORTS according to the specification in Code snippet 9.

Code snippet 9: Library and ENTITY block showKey

```
-- ----------------------------------------------------------------------
ENTITY showKey IS
   PORT(
      reset      : IN  std_logic;
      kbclock    : IN  std_logic;
      kbdata     : IN  std_logic;
      dig0, dig1 : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
      scancode   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
      byte_read  : OUT std_logic
   );
END showKey;
-- ----------------------------------------------------------------------
```

- ☐ Write VHDL code in the ARCHITECTURE of ENTITY *showKey* to implement to read the PS/2 serial data and present the serialized data parallel accompanied by a signal in port **byte_read** when data is read from the keyboard and valid data is available.

  For the implementation follow the instruction of paragraph 5.2.2.

Further reading:
https://www.digikey.com/eewiki/pages/viewpage.action?pageId=70189075

5.2.3.2 Part B of assignment 1: Create a testbench for ENTITY showKey

In this assignment a testbench will be used instead of the waveform analysis we have used so far.

- ☐ Implement a testbench for component *showKey* with the help of the example in Code snippet 8.

- ☐ Solve the syntax errors in the testbench.

- ☐ Compile the testbench.

5.2.3.3 *Part C of assignment 1: Test and verification*

- ☐ Test and verify the functionality of ENTITY *showKey* using the testbench that was created in "Part B of assignment 1: Create a testbench for ENTITY showKey"
- ☐ Create a screenshot of 3 sent bytes 0xC1, 0x32 and 0x4D.
- ☐ Present the result of your test and verification activities to the teacher.

5.3 Assignment 2: Component *readKey*

In this assignment we will create COMPONENT *constantKey.* COMPONENT *constantKey* will be combined with COMPONENT *showKey* and COMPONENT *crossDomaincrossing* into the COMPONENT *readKey.*

COMPONENT *constantKey* will present the hexadecimal byte that represents the key that is pressed on the PS/2 keyboard for the duration that it is pressed. See the architecture diagram in Figure 18.

COMPONENT *crossDomaincrossing* is given and will solve the issue where signals from different clock domains must connect to each other.

Using structural VHDL, COMPONENT *clockDomainCrossigng, showKey,* and *constantKey* will be combined into a single component named *readKey.*

In this assignment *showKey* and *constantKey* are developed as part of the assignment while *showKey* was developed in assignment 2 and *clockDomainCrossing* and *readKey* are provided.

For instruction purpose *clockDomainCrossing* and *readKey* are discussed in the theory of this assignment.

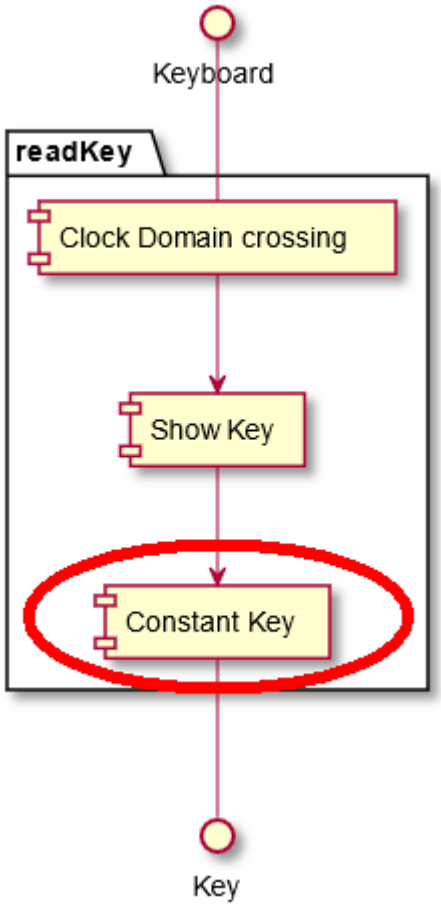

Figure 18: readKey architecture

With this assignment knowledge will be gained on state machines in VHDL and Cross Domain Crossing.

5.3.1 Theory

5.3.1.1 State machines

In digital systems, there are two basic types of circuits: combinational logic circuits and sequential logic circuits.

In combinational logic circuits, the outputs depend solely on the inputs. In sequential logic circuits, the outputs depend not only on the inputs, but also on the present state of system (i.e., the values of the outputs and any internal signals or variables).

There are two types of finite state machines. These are the Mealy FSM and the Moore FSM. The fundamental difference between these two types lies in the management of the outputs:

- The output of the Mealy FSM depends on the present state and inputs.
- The outputs of a Moore machine depend only on the present state and not on the inputs.

Figure 19 shows the Mealy FSM while Figure 20 schematizes the Moore FSM.

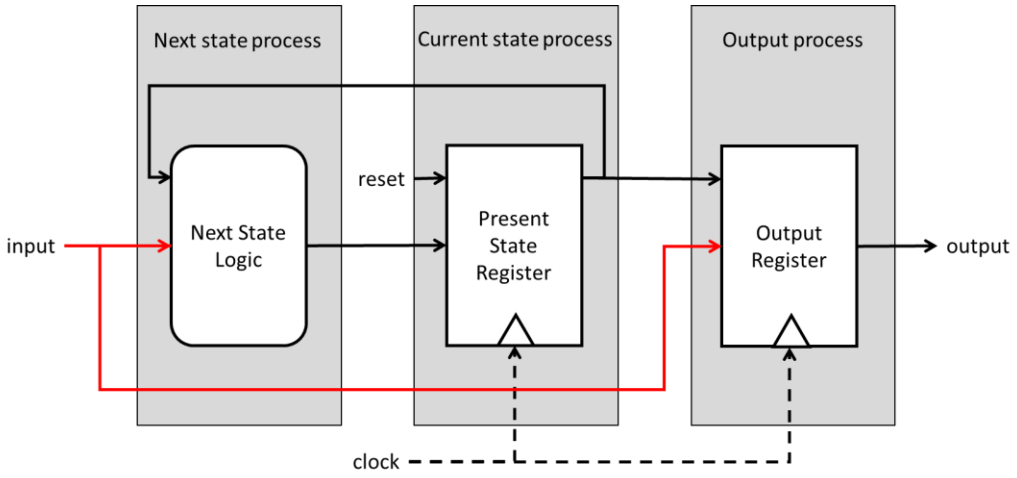

Figure 19: Mealy FSM schematic view

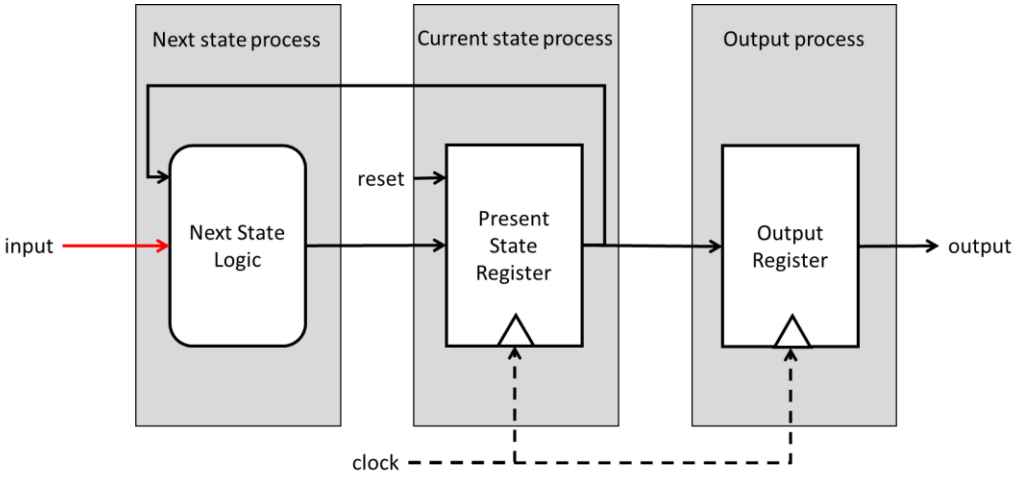

Figure 20: Moore FSM schematic view

The Moore FSM are preferable to the Mealy FSM since the output of the Moore FSM depends only on the current machine state. No assumptions or check on the inputs must be performed to generate the output of the FSM, so the output decoding is simpler to handle.

Moreover, if the output is combinatorial, i.e. not registered, the Moore FSM is safer with respect to Mealy FSM: The long combinatorial path between input and output can generate glitch on the machine output or can reduce dramatically the design timing performances.

State machines are visualized using UML state diagrams as shown in Figure 21. Here the initial state is State_1 that is reached after start. During state_1 action 1 is executed. When condition 1 is met the state changes from state_1 to state_2 that executes action 2. When condition 2 is met the FSM changes to state_3 and executes action 3. Because no condition is set to exit state_3 immediately the FSM changes state to state_1.

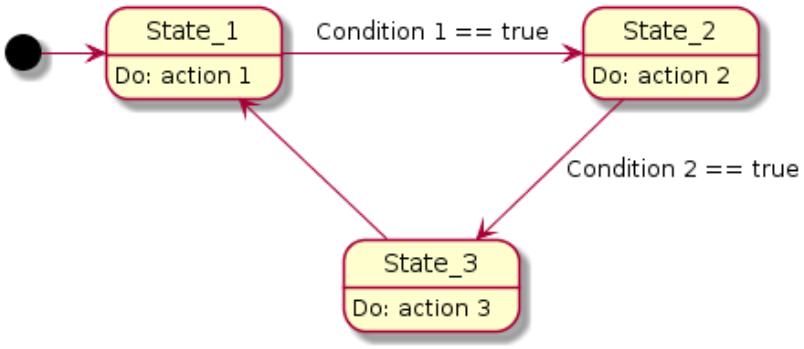

Figure 21: Example state diagram

In VHDL the three blocks are implemented as 3 separate processes that execute the tasks of the FSM. These tasks are:

1. Next state logic. Here the next state is set based upon input information.
2. Current state logic. This logic sets next state to current state and when a reset is executed the FSM is det to the initial state.
3. Output register. This logic translates the state of the FSM in to output signals.

Study paragraph 11 VHDL design of State Machines in Pedroni.

In VHDL an FSM is normally implemented using sequential logic in 3 PROCESS blocks. As the PROCESS blocks are sequential logic in a concurrent environment the order of the PROCESS blocks is irrelevant for the operation. To transfer information from and to the PROCESS blocks SIGNALS are being used. These SIGNALS carry the current- and next-state of the FSM.

To increase readability of the code an enumerated TYPE is defined to represent the state names.

In Code snippet 9, the VHDL implementation of the example FSM presented that was presented in Figure 21.

It is important for the operation of the FSM that the **next_state** assignment is applied in the input decoder at any times. E.g. make sure that **next_state** is assigned a state in <u>any path</u> that can be followed in the switch CASE and IF – THAN operations.

In the case that the compiler is warning about “inferring latches”, the previous rule is not obeyed.

Code snippet 10: FSM example in VHDL

```
-----------------------------------------------------------------------
ENTITY fsm_example IS
   PORT (
      clk    : IN  STD_LOGIC;
      reset  : IN  STD_LOGIC;
      input  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
      output : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
   );
END fsm_example;
-----------------------------------------------------------------------
ARCHITECTURE fsm_implementation_1 OF fsm_example IS
--! "typedef" for enum state_type.
   TYPE state_type is (
      state_1,
      state_2,
      state_3
   );
--! signals of the type state_type to contain current- and next-state
--! to allow echange of information between PROCESSes.
--! SIGNALs ar initialized with state_1 value to enforce initial state
--! when powering FSM.

   SIGNAL current_state,
          next_state     : state_type := state_1;
BEGIN
--! state_decoder
--! =============
--! This PROCESS is PROCESSing state changes each clk and executing
--! async reset.
   state_decoder: PROCESS (clk, reset) IS  -- watch reset and clk
   BEGIN
      --! reset all output signals of FSM
      --! set FSM to inital state
      IF reset = '0' THEN -- Reset (async).
         current_state <= state_1;
      ELSIF rising_edge(clk) THEN
         current_state <= next_state;
      END IF;
   END PROCESS;
--! input_decoder
--! =============
--! this PROCESS contains the tests and conditions for each state
   input_decoder : PROCESS (current_state)
   BEGIN
      CASE current_state is
         WHEN state_1 =>
            IF (input = "0001") THEN
               next_state <= state_2;
            ELSE
```

```
                --! prevent inferring latches
                next_state <= current_state;
            END IF;
         WHEN state_2 =>
            IF (input = "0010") THEN
                next_state <= state_3;
            ELSE
                --! prevent inferring latches
                next_state <= current_state;
            END IF;
         WHEN state_3 =>
            next_state <= state_1;
         WHEN OTHERS =>
            --! prevent inferring latches
            next_state <= current_state;
      END CASE;
   END PROCESS;
--! output decoder
--! =============
--! this PROCESS is performing actions that apply for each state
   output_decoder : PROCESS (current_state)
   BEGIN
      CASE current_state IS
         WHEN state_1 =>
            output <= "0001";
         WHEN state_2 =>
            output <= "0010";
         WHEN state_3 =>
            output <= "0011";
         WHEN OTHERS =>
            output <= "1111";
      END CASE;
   END PROCESS;
END fsm_implementation_1;
-------------------------------------------------------------------------
```

5.3.1.2 Clock Domain Crossing

Clock Domain Crossing (CDC) occurs when data is transferred from a domain driven by one clock to a domain driven by another clock.

In our VHDL piano we transfer data from the keyboard accompanied by the low-speed clock (20 kHz) generated by the keyboard to the FPGA that runs in a 50 MHz clock. When crossing a clock domain 3 main issues can occur. These are:

1. Metastability
2. Data loss
3. Data Incoherency

In the following paragraphs we briefly look into these phenomena. While discussing, we will name the domain with the slow clock source the source and the domain to which we will transfer the data that has the high-speed clock the destination.

Metastability

If the transition of a signal happens very close to the active edge of the clock in the target, it could lead to setup or hold violation at the destination domain. As a result, the signal in the destination domain

may oscillate for an indefinite amount of time. Thus, the signal is unstable and may or may not settle down to some stable value before the next clock edge of the destination domain arrives. This phenomenon is known as metastability and the logic driving the destination domain is said to have entered a metastable state. (Verma & Dabare, 2020)

The solution for metastability is the use of synchronizers. Synchronizers consist of 2 sequential flipflops that are driven by the same high-speed clock. This will result in a delay of one clock cycle for the applied signal before it enters the destination domain. Doing so the period in which the first flipflop might be in a metastable phase is isolated and so discarded.

This structure is mainly used for single and multi-bit control signals and single bit data signals in the design. Other types of synchronization schemes are required for multi-bit data signals such as MUX recirculation, handshake, and FIFO. (Verma & Dabare, 2020)

Data loss

Whenever data is generated by the source, it may not be captured by the destination domain in the very first cycle of the destination clock because of metastability. If each transition on the source signal is captured in the destination domain, data is not lost. In order to ensure this, the source data should remain stable for some minimum time, so that the setup and hold time requirements are met with respect to at least one active edge of destination clock.

In order to prevent data loss, the data should be held constant in the source domain long enough to be properly captured in the destination domain. In other words, after every transition on source data, at least one destination clock edge should arrive where there is no setup or hold violation so that the source data is captured properly in the destination domain. This can be implemented by using a state machine (FSM) to generate source data at a rate, such that it is stable for at least 1 complete cycle of the destination clock. This can be generally useful for synchronous clocks when their frequencies are known. For asynchronous clock domain crossings, techniques like handshake and FIFO are more suitable (Verma & Dabare, 2020).

Data Incoherency

As seen in the previous paragraphs, whenever data is generated in the source clock domain, it may take 1 or more destination clock cycles to capture it, depending on the arrival time of active clock edges. Consider a case where multiple signals are being transferred from one clock domain to another and each signal is synchronized separately using a multi-flop synchronizer. If all the signals are changing simultaneously and the source and destination clock edges arrive close together, some of the signals may get captured in the destination domain in the first clock cycle while some others may be captured in the second clock cycle by virtue of metastability. This may result in an invalid combination of values on the signals at the destination side. Data coherency is said to have been lost in such a case. (Verma & Dabare, 2020).

With the implementation of the VHDL piano we will not suffer from data incoherency but likely from data loss and possible from metastability. Therefore, the data and clock signals from the PS/2 interface will be fed through a clock domain crossing component that is built with 2 flipflops. The logic circuit is displayed in Figure 25.

Further reading: See Pedroni paragraph “12.4 Frequency Meter (with LCD)” sub paragraph “Including a Synchronise in the Design” on page 336.

5.3.2 Design

In the design paragraph of this assignment we discuss the individual components of *readKey* in the order as presented in the architecture diagram in Figure 18.

5.3.2.1 Component Clock Domain Crossing

Because the signals from the PS/2 keyboard are generated using a 20 kHz clock we must synchronise this signal to the internal 50 MHz clock of the FPGA. COMPONENT *clock-domain-crossing* will implement this function. The input-process-output diagram fr COMPONENT *clock-domain-crossing* will is presented in Figure 22.

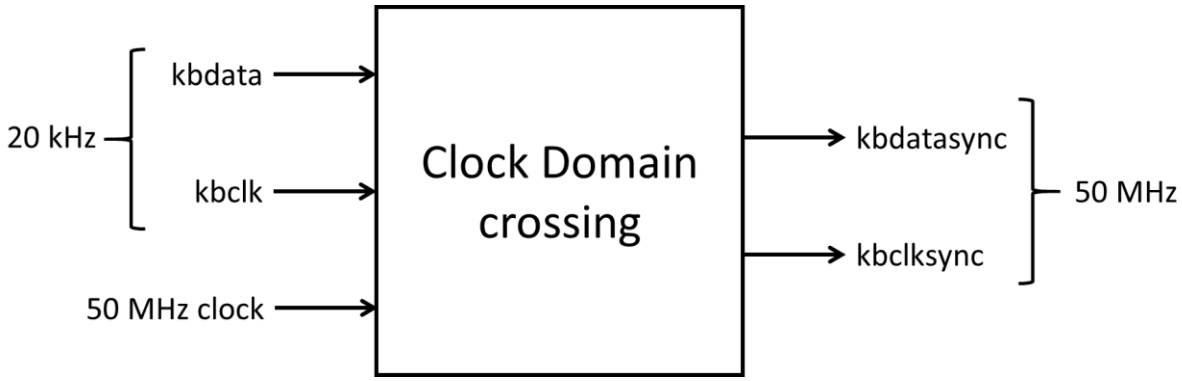

Figure 22: input-process-output diagram for component clock-domain-crossing

5.3.2.2 Component constantKey

Component *constantKey* will present on PORT **key,** for the duration that a key is pressed on the keyboard, the hexadecimal code that is associated with the pressed key. To determine is the value on PORT **scancode** is valid PORT **byte_read** is evaluated.

For the purpose of debugging PORTs **dig1** and **dig2** will present the current and last value of PORT **key**.

The input-process-output diagram is shown in Figure 23.

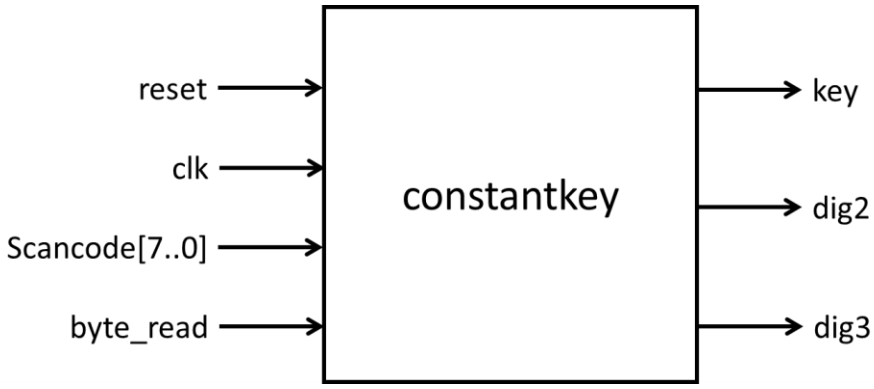

Figure 23: input-process-output diagram for component constantKey

The ENTITY that shall be used is given in Code snippet 11 and can be downloaded from the git-repository.

Code snippet 11: Library and ENTITY block constantKey

```
-------------------------------------------------------------------------
ENTITY constantKey IS
   PORT (
      reset     : IN  STD_LOGIC;
      clk       : IN  STD_LOGIC;
      scancode  : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
      byte_read : IN  STD_LOGIC;
      key       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
      dig2,
      dig3      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
   );
END constantKey;
-------------------------------------------------------------------------
```

5.3.2.3 Description of component constantKey:

When **byte_read** is '1' a valid hexadecimal code is available on **scancode**. In our VHDL piano this code shall be converted in a tone that corresponds with the pressed key on the PS/2 keyboard.

There are two ways to implement the use of **scancode** and **byte_read**:

1. Test for **byte_read** to be '1', or
2. test for a change of state of **byte_read** from '0' to '1' or test for "rising-edge".

In the implementation of the VHDL piano the use of raising edge will be implemented. At the rising edge of **byte_read** the value of **scancode** will latched into a register.

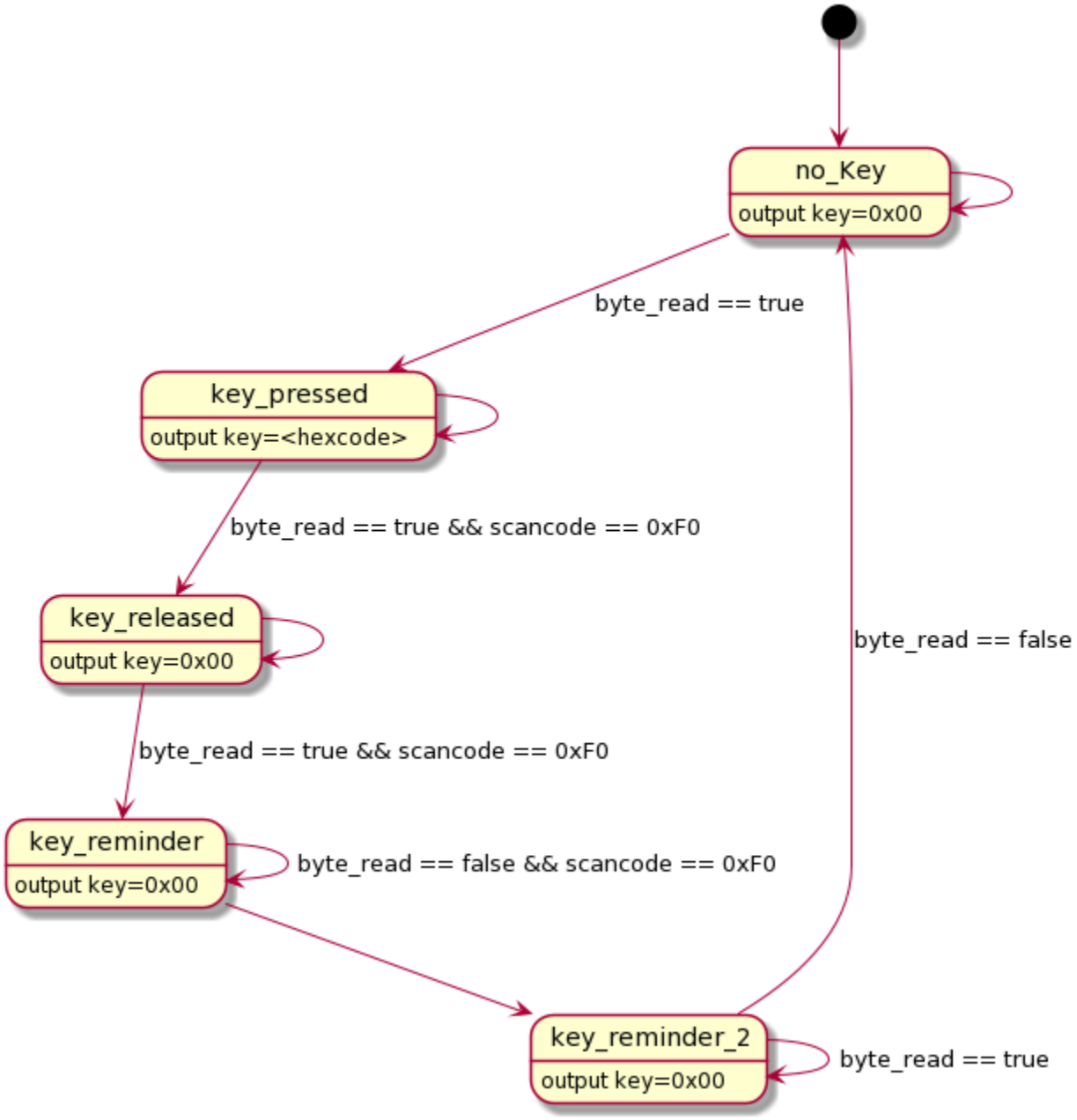

Figure 24: constantKey State diagram

Because we want to have the information presented when a key is pressed and when a key is released outputs **dig2** and **dig3.**

Output **key** will present 0x00 when no key is pressed, the hexadecimal value of the key that is pressed and temporary the code that a key is released 0xF0. Therefore, we need a minimum of 3 states.

Summarized:

1. When no key is pressed: output **key** = 0x00.
2. When a key is pressed: output **key** = hexadecimal value of the key pressed.
3. When a key is released: output **key** = 0x00.

5.3.3 Realisation and verification

- ☐ Download the files for assignment 2 from Gitlab: https://gitlab.com/wlgrw/han-soc-assignment-vhdl-piano

5.3.3.1 clockDomainCrossing

Component *clock-domain-crossing* is a synchroniser for both data and clock signals from the PS/2 keyboard. A single synchroniser with 2 flipflops is displayed in Figure 25. Component *clock-domain-crossing* has 2 synchronisers. One for data and one for clock.

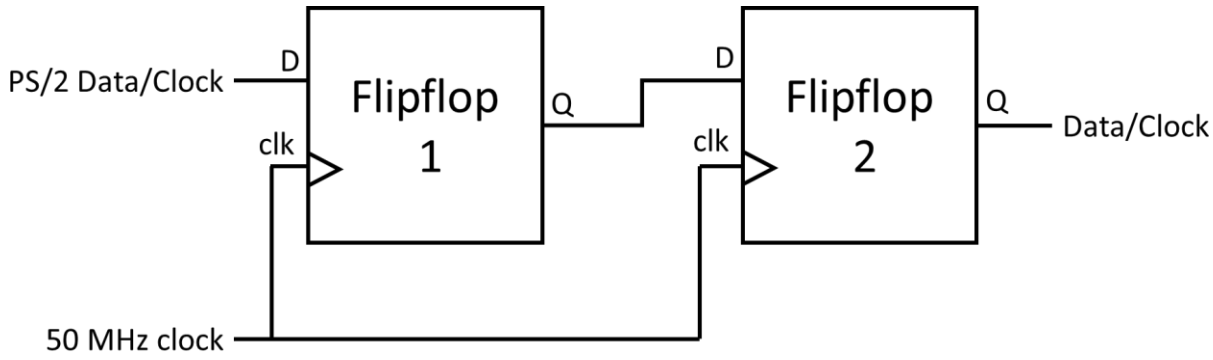

Figure 25: Generic clock domain crossing circuit.

The VHDL code is using structural VHDL to create a single flipflop as a component that is instantiated 4 times and interconnected using SIGNALS **kbdatasignal** and **kbclocksignal.**

- ☐ Study the VHDL code in clock_domain_crossing.vhd that is part of directory Assignment_2 in the repository.

5.3.3.2 constantKey

- ☐ Open with Quartus the project “constantKey” in folder “assignment_2_constantKey” or create a project with the top-level ENTITY named *constantKey.*

- ☐ Verify if the ENTITY *constantKey* has implemented the PORTS according to the specification in Code snippet 12.

Code snippet 12: ENTITY block constantKey

```
-------------------------------------------------------------------
ENTITY constantKey IS
   PORT (
      reset     : IN  STD_LOGIC;
      clk       : IN  STD_LOGIC;
      scancode  : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
      byte_read : IN  STD_LOGIC;
      key       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
      dig2,
      dig3      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
   );
END constantKey;
-------------------------------------------------------------------
```

- ☐ Verify the FSM that is presented for *constantKey* in Figure 24.
- ☐ Make necessary changes to the state diagram when applicable.
- ☐ Implement the VHDL code for the FSM in the template that is provided in file "constantKey.vhd". Use 3 PROCESS statements for the state- input- and output-decoder.
- ☐ Compile the VHDL code. Make sure that no warnings about "inferring latches" exist!
- ☐ Study the testbench "constantKey_tb.vhd"
- ☐ Compile the testbench.
- ☐ Verify the implementation of *constantKey* using the testbench that was provided with this assignment.
- ☐ Create a screenshot of the following sequence:
  - Press key 'a' → 0x52
  - Release key 'a' → 0xF0
  - Wait,
  - Press key 'a' → 0x52
  - Release key 'a' → 0xF0
- ☐ Present the result of your test and verification activities to the teacher.

5.3.3.3 readKey

readKey itself does not add new functionality to the VHDL piano but combines components *clockDomainCrossing, showKey* and *contantKey* into a new component using structural VHDL.

The result of compiling *readKey* is shown in Figure 26 where all components can be found that are part of *readkey*.

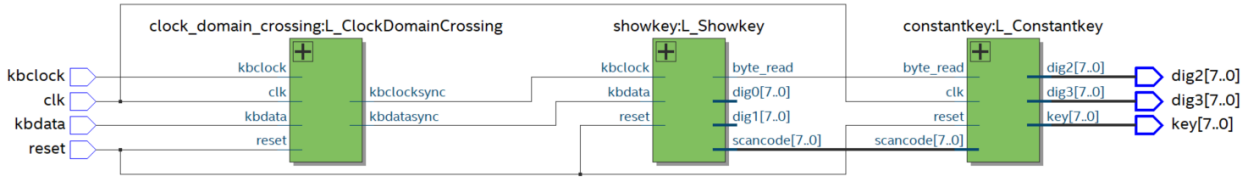

Figure 26: structural VHDL presentation of components in the RTL-viewer or Quartus

The ARCHITECTURE that shall be used is given in Code snippet 13 and can be downloaded from the git-repository.

Code snippet 13: ARCHITECTURE of readKey

```
-------------------------------------------------------------------
ARCHITECTURE readKey _struct OF readKey  IS
   SIGNAL C_scancode       : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL C_byte_read,
          C_byte_read_sync : STD_LOGIC; -- byte read indication by showKey
   SIGNAL C_kbdata         : STD_LOGIC; -- keyboard serial data
   SIGNAL C_kbclock        : STD_LOGIC; -- keyboard clock
BEGIN
--! component Clock_domain_crossing to interface between 20 kHz
--! and 50 MHz domain
   L_ClockDomainCrossing: work.clock_domain_crossing PORT MAP(
--    + component port
--    |              + Internal port or signal
--    |              |
      reset       => reset,
      clk         => clk,
      kbclock     => kbclock,
      kbdata      => kbdata,
      kbclocksync => C_kbclock,
      kbdatasync  => C_kbdata
   );
--! component showKey to convert serial data from keyboard
--! to parallel data
   L_showKey: ENTITY work.showKey PORT MAP(
      reset     => reset,
      kbclock   => C_kbclock,
      kbdata    => C_kbdata,
      scancode  => C_scancode,
      byte_read => C_byte_read
   );
--! component constantKey that presents key data from keyboard
--! if the key is pressed.
   L_constantKey: ENTITY work.constantKey PORT MAP(
      reset     => reset,
      clk       => clk,
      scancode  => C_scancode,
      byte_read => C_byte_read,
      key       => key,
      dig2      => dig2,
      dig3      => dig3
   );
END readKey _struct;
-------------------------------------------------------------------
```

- ☐ Copy the ARCHITECTURE of readKey into your project.
- ☐ Add the previously developed submodules to your project.
- ☐ Solve all errors and warnings to make the code synthesizable
- ☐ Rewrite the testbench for *showKey* by replacing component *showKey* for component *readKey*

- Solve all errors and warnings to make the code synthesizable
- Extend the stimuli of tb_kbclock and tb_kbdata in the testbench so the pattern of another keypress is being generated:

```
WWWW<no key><no key>PPPP<no key>AAAA<no key>
```

- Simulate *readKey* and test i tand present the timing diagram.

5.4 Assignment 3: Component *clockGenerator*

In this assignment we will create COMPONENT *clockGenerator*. *clockGenerator* will be combined with COMPONENT *key2pulselenght* and *pulselength2Audio* into the COMPONENT *tone_generation.* See Figure 27.

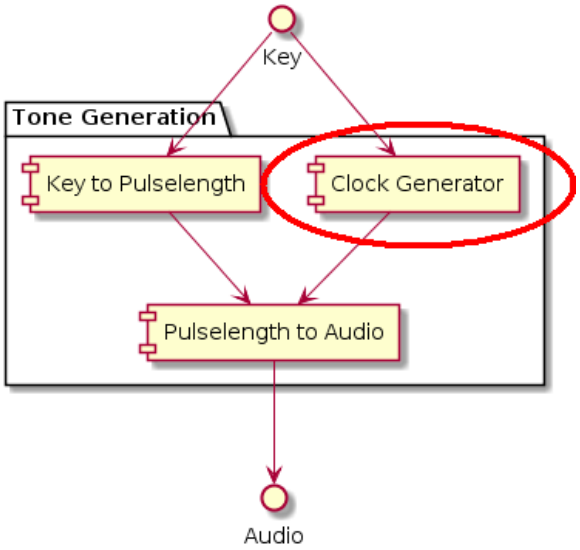

Figure 27: Architecture diagram Tone Generation with COMPONENT clockGenerator

5.4.1 Theory

Often, inside an FPGA design, we have the necessity to generate a local clock from the system clock. With system clock in our case is meant the 50 MHz clock that is available on the DE10-Lite board.

Clock manipulation can be performed up and down. For up conversion Phased Locked Loop (PLL) Synthesisers are being used who are out of scope of this assignment. For down conversions dividers are being used which will be used in VHDL piano.

The simplest clock divider that can be implemented into a FPGA is a divider that divides by a factor 2. This divider can be generalized to a divider by a power of 2. In logic circuits a divider by 2 is implemented using a binary ripple counter. See Figure 28. This binary ripple counter is implemented using D-flip-flops where output ‘Q is fed in to input D while at each rising edge output Q is inverted from its previous value.

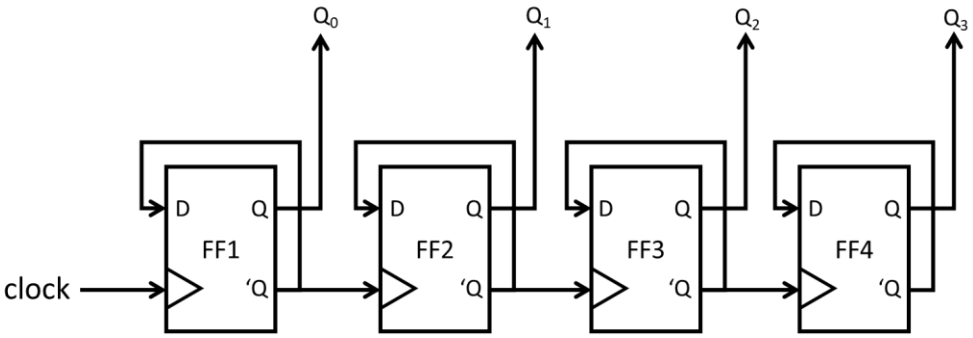

Figure 28: 4-bit binary ripple counter

The truth table for the binary ripple counter in Figure 28 is given in Figure 29. In addition to the logic value '1' and '0' the "signal shape" is drawn in the table. This way it is visualised that the frequency of clock is divided by 2 and available in output Q_0. With this 4-bit ripple counter output Q_0 carries $f_{clk}/2$, output Q_1 carries $f_{clk}/4$, output Q_2 carries $f_{clk}/8$, output Q_4 carries $f_{clk}/16$.

clock	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Q0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1
Q1	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1
Q2	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
Q3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 29: Truth table for 4-bit binary ripple counter with "signal-shape"

In VHDL a binary ripple counter with selectable output can be implemented by using a counter followed by a multiplexer. See Figure 30. Here a synchronous clock signal and an asynchronous reset are provided to the counter that uses a binary word to count with. When the reset is applied the binary word is set to zero while at a rising edge of clock the counter is incremented with one. This is similar as presented in Figure 29.

With the multiplexer as single bit is selected from the counting word. This will result in the divided signal clk_div that has carried the divided clock as a function of the power of 2.

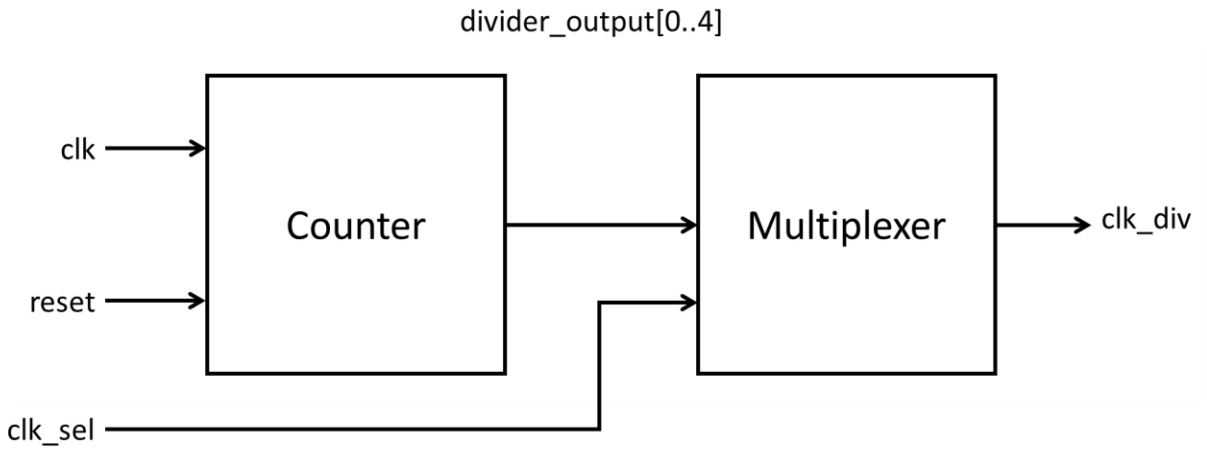

Figure 30: block diagram of variable clock divider

In the VHDL example in Code snippet 14 a PROCESS statement is used for the counter. The counter is implemented in sequential code while the multiplexer is implemented in concurrent code.

Code snippet 14: VHDL-code of clock divider example

```
------------------------------------------------------------------------
LIBRARY ieee;
USE     ieee.STD_LOGIC_1164.all;
USE     ieee.numeric_std.all;
------------------------------------------------------------------------
ENTITY clockdivider_example IS
   GENERIC(
      N: INTEGER := 4
   );
   PORT(
      reset   : in  STD_LOGIC;
      clk     : in  STD_LOGIC;
      clk_sel : in  STD_LOGIC_VECTOR(N DOWNTO 0);
      clk_div : out STD_LOGIC
   );
END clockdivider_example;
------------------------------------------------------------------------
ARCHITECTURE LogicFunction OF clockdivider_example IS
   SIGNAL divider_output : STD_LOGIC_VECTOR(N DOWNTO 0);
   SIGNAL selector : INTEGER RANGE 0 TO 2 ** N := 0;
BEGIN
   devider : PROCESS (clk, reset)
      VARIABLE counter : INTEGER RANGE 0 TO 2 ** N := 0;
   BEGIN
      IF reset = '0' THEN                    -- Async reset
         counter := 0;                       -- set counter to 0
         divider_output <= (OTHERS => '0'); -- set counter to 0
      ELSIF rising_edge(clk) THEN            -- on clock edge
         IF (counter < (2 ** N)) THEN        -- When counter is < 2 power N
           counter := counter + 1;           -- increment counter
         ELSE                                -- counter = 2 power N
           counter := 0;                     -- reset counter to 0
         END IF;
      END IF;                                -- put result on signal
```

```
        divider_output <=
                STD_LOGIC_VECTOR(to_unsigned(counter, divider_output'length));
    END PROCESS;
    selector <= to_integer(unsigned(clk_sel));
    WITH selector SELECT
        clk_div <= divider_output(0) WHEN 0,
                   divider_output(1) WHEN 1,
                   divider_output(2) WHEN 2,
                   divider_output(3) WHEN 3,
                   '0' WHEN OTHERS;
END LogicFunction;
-------------------------------------------------------------------------
```

5.4.2 Design

The component *ClockGenerator* will generate the clock frequency from which the tones will be generated that are audible. These tones will be within one octave. *ClockGenerator* will us the 50 MHz internal clock as a reference.

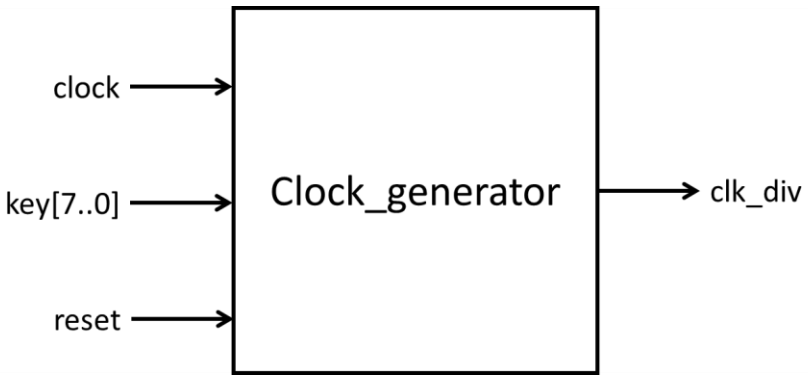

Figure 31: input-process-output diagram for ClockGenerator

Apart from the clock input, another Input to *ClockGenerator* is **key**. **Key** carries the keys pressed from the keyboard where keys 'A' and key 'Z' will affect the tone height. Depending on the key pressed, the clock frequency being generated by *ClockGenerator* will be divided by 2 over 7 octaves. A maximum 3 octaves higher and 3 octaves lower than the default frequency. As a result, port **clk_div** can have 7 different values.

Summarized:

- A clock divider shall be implemented using a 50 MHz ground frequency that will generate a frequencies with a factor $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}$ and $\frac{1}{64}$
- The resulting frequencies will be:
  50 MHz, 25 MHz, 12,5 MHz, 6,25 MHz, 3,125 MHz, 1,5625 MHz or 0,78125 MHz

Note 1: With a **reset** the "middle" frequency of 6,25MHz will be set.

Note 2: Remember that during the keypress the code will be continuously on **key** available. When no counter measure is implemented, *clockGenerator* will see the code at a speed of 50 MHz multiple times and increment or decrement the tone with one octave.

As a result, *clockGenerator* will contain a clock divider, multiplexer and a state machine where the state machine is controlling the multiplexer. See the Architecture diagram of *clockGenerator* in Figure 32.

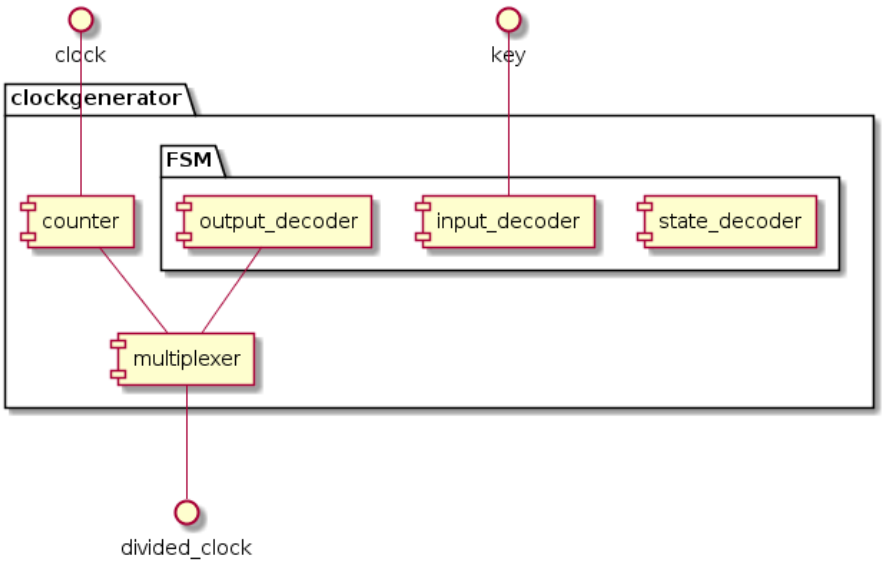

Figure 32: Architecture of ENTITY clockGenerator

The FSM is given in Figure 33.

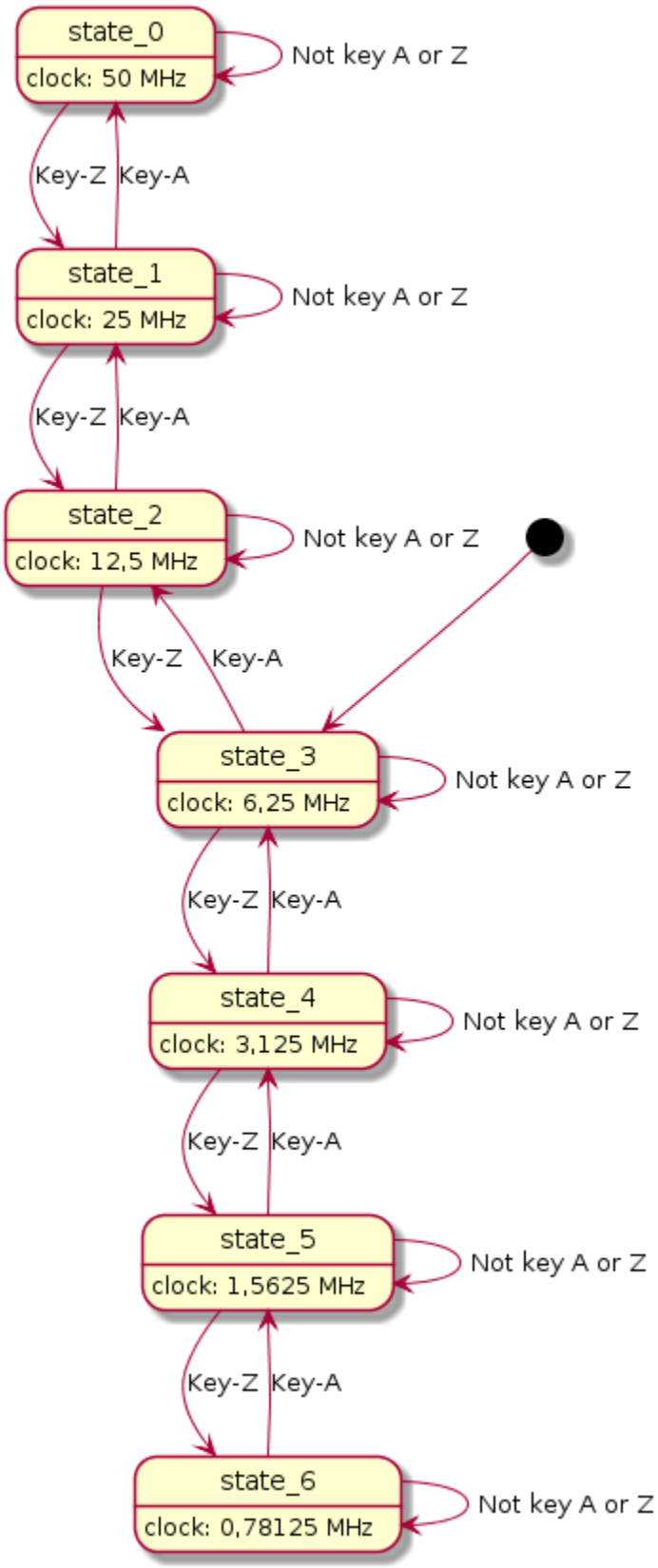

Figure 33: State diagram for component clock_divider

5.4.3 Realisation and verification

- ☐ Download the files for assignment 3 from Gitlab: https://gitlab.com/wlgrw/han-soc-assignment-vhdl-piano

5.4.3.1 Example clock divider

The VHDL code is using structural VHDL to create a single flipflop as a component that is instantiated 4 times and interconnected using SIGNALS **kbdatasignal** and **kbclocksignal.**

- ☐ Open with Quartus the project "clockdivider_example" in folder "assignment_3_clockdivider_example".

- ☐ Study the VHDL code in clockdivider_example.vhd.

5.4.3.2 ClockGenerator

- ☐ Open with Quartus the project "clockGenerator" in folder "assignment_3" or create a project with the top-level ENTITY named *clockgenerator.*

- ☐ Verify if the ENTITY *clockGenerator has* implemented the PORTS according to the specification in Code snippet 15.

Code snippet 15: ENTITY for ClockGenerator

```
-------------------------------------------------------------------
ENTITY ClockGenerator IS
   PORT(
      reset    : in  STD_LOGIC;
      clk      : in  STD_LOGIC;
      key      : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
      clk_div  : out STD_LOGIC
   );
END ClockGenerator;
-------------------------------------------------------------------
```

- ☐ Design the VHDL for the ARCHITECTURE of component *ClockGenerator*

- ☐ Investigate the technology map in RTL viewer and share your findings

- ☐ Simulate the functional operation of *ClockGenerator* with the following steps:

  <reset>, up, up, up, up, down, down, down, down, down, down, down, down, up, up.

5.5 Assignment 4: Component key2pulselength

In this assignment we will create COMPONENT *key2pulselength*. *key2pulselength* will be combined with COMPONENT *clockGenerator* and *pulselength2Audio* into the COMPONENT *tone_generation.* See Figure 34.

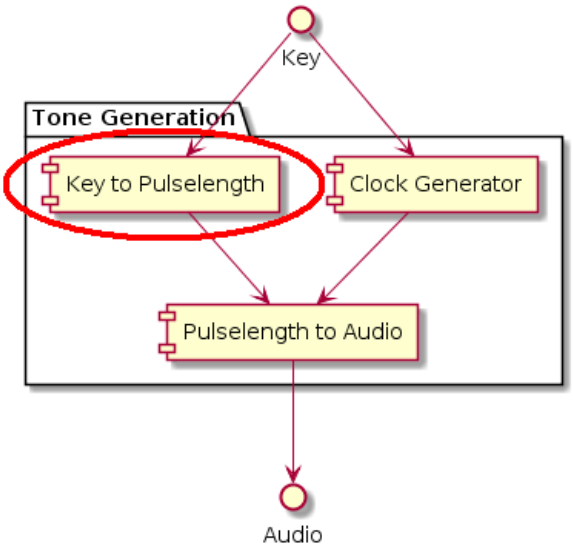

Figure 34: Architecture diagram Tone Generation with COMONENT key2pulselength

5.5.1 Theory

5.5.1.1 Clock divider by n.

In assignment 3 a clock divider was implemented with a "static" division factor by the power of 2. In this assignment a clock divider is implemented with a variable division factor.

A clock frequency can be generated from a single clock if the clock frequency of the source is higher than the frequency wanted. On the contrary not any frequency can be generated from a clock by dividing.

Table 2 presents the clock frequencies from a 50 MHz clock by 11 division factors. In the right column the resulting frequencies are presented. From this we can see that a frequency between 50 and 25 MHz for example cannot be generated using frequency division.

Another way of visualising the gaps in frequencies that cannot be generated is presented in Figure 35. Here the gaps are marked with brackets.

Table 2: Divisor example for divisor value 1 to 11

Divisor	Frequency (MHz)
1	50
2	25
3	16,6
4	12,5
5	10
6	8,3
7	7,14
8	6,25
9	5,5
10	5
11	4,5

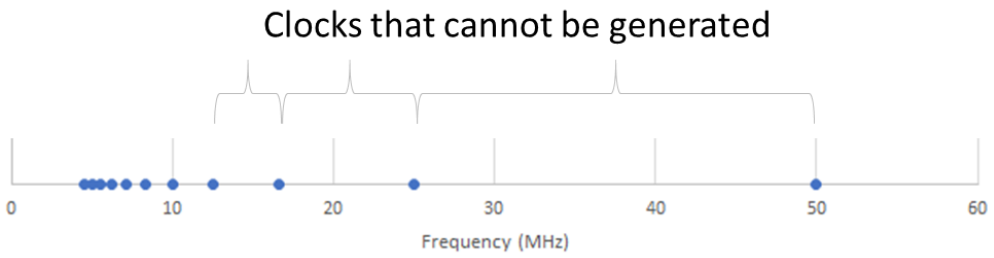

Figure 35: Visual presentation of generated clocks

From the previous results we can learn that, unless a divisor if the clock is resulting in a number without a fraction, it will never be exactly the frequency that is required. Also, we can learn that when the divisor is significant, the offset from the wanted frequency is smaller and eventually acceptable.

5.5.1.2 Tone generation

By default, the COMPONENT *clockGenerator* is generating a clock frequency of 6,25 MHz. This clock frequency is the frequency that was used for the normal tones from the scale. Using this frequency, we can calculate the devisor in COMPONENT *pulselength2audio* needed for the frequency divider that will generate the audio tone. Because of the implementation of *pulseliength2audio* the divisor will express the duration of half a period time of the audio tone.

Table 3: Conversion table from key and tone to divisor.

No	Key	Hex	Tone	Freq (Hz)	Divisor
1	TAB	0D	A	440	7102
2	1!	16	Ais	466	6704
3	Q	15	B	494	6327
4	W	1D	C	523	5972
5	3#	26	Cis	554	5637
6	E	24	D	587	5321
7	4$	25	Dis	622	5022
8	R	2D	E	659	4740
9	T	2C	F	698	4474
10	6^	36	Fis	740	4223
11	Y	35	G	784	3986
12	7&	3D	Gis	831	3762
13	U	3C	A+	880	3551
14	8*	3E	Ais+	932	3352
15	I	43	B+	988	3164
16	O	44	Cis+	1047	2986
17	0)	45	C+	1109	2819
18	P	4D	Dis+	1175	2660
19	-_	4E	D+	1245	2511
20	[{	54	E+	1319	2370
21	]}	5B	F+	1397	2237
22	Back	41	Fis+	1480	2112

5.5.2 Design

In this assignment component *key2pulselength* is designed and implemented. The input-process-output diagram is presented in Figure 36.

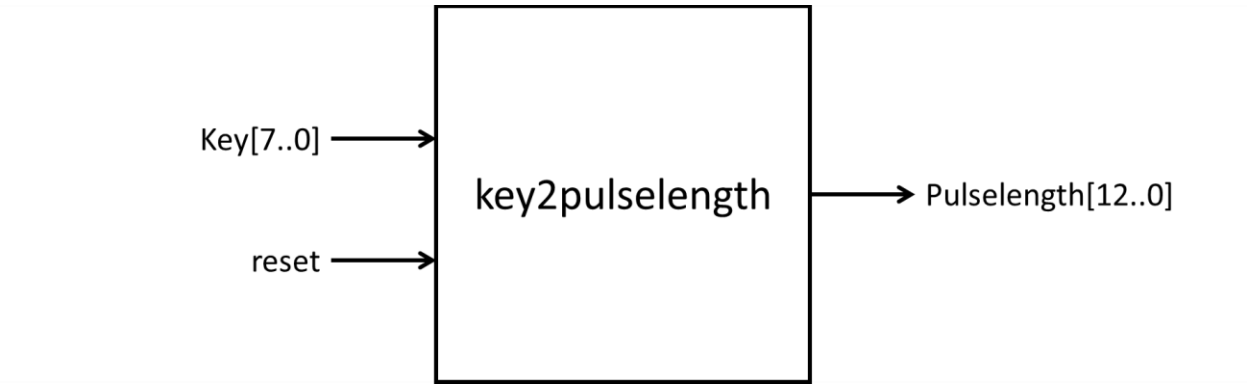

Figure 36: input-process-output diagram of component key2pulselength

In COMPONENT *key2pulselength* PORT **key** is converted to the divisor of the tone represented by an integer value: **pulselength**. **Pulselength** represents the number of times that **clk_div** will generate a clock pulse (single period). Doing so, **pulselength** specifies halve the period time of the audio tone. The combination of **pulselength** and **clk_div** is implemented in *pulselength2audio* in Assignment 5: Component *pulselength2audio*.

5.5.3 Realisation and verification

- ☐ Download the files for assignment 4 from Gitlab: https://gitlab.com/wlgrw/han-soc-assignment-vhdl-piano

For this assignment use the ENTITY that is specified in Code snippet 16

Code snippet 16: Library and ENTITY of component key2pulselength

```
-------------------------------------------------------------------------
ENTITY key2pulselength is
   PORT(
      reset      : IN  STD_LOGIC;
      key        : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
      pulslength : OUT integer RANGE 0 TO 8191
   );
END key2pulselength;
-------------------------------------------------------------------------
```

- ☐ Open with Quartus the project “key2pulselength” in folder “assignment_4” Create the top-level ENTITY for component *key2pulselength.*
- ☐ Design the VHDL code for the ARCITECTURE of component *key2pulselength*
- ☐ Investigate the technology map in RTL viewer and share your findings
- ☐ Simulate the functional operation of *key2pulselength.*

5.6 Assignment 5: Component *pulselength2audio*

In this assignment we will create COMPONENT *pulselength2Audio*. *pulselength2Audio* will be combined with COMPONENT *key2pulselenght* and *clockGenerator* into the COMPONENT *tone_generation.* See Figure 37.

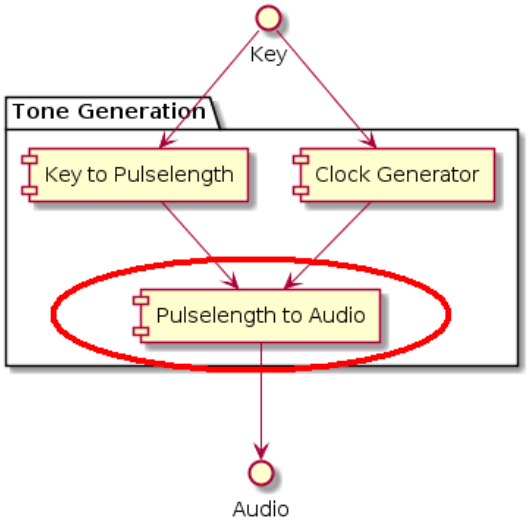

Figure 37: Architecture diagram Tone Generation with COMPONENT pulselength2audio

5.6.1 Design

In this assignment the submodule is designed and implemented that will generate the audio signal for the VHDL piano.

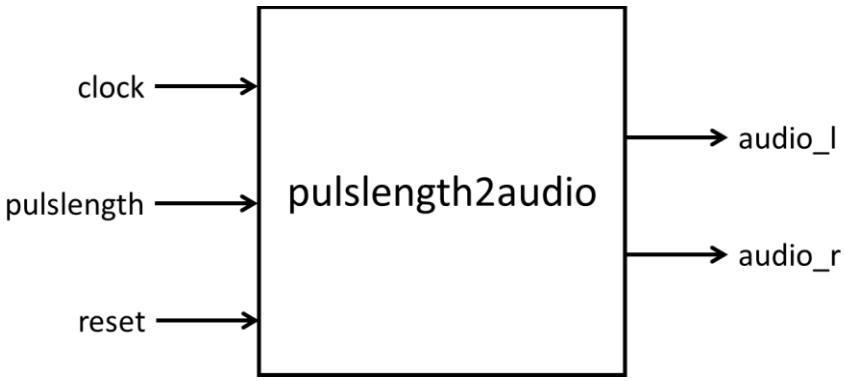

Figure 38: Input-process-output diagram for component pulslength2audio

In this component the base-frequency that is being generated by *clockGenerator* is used to generate the desired audio frequency using information from PORT **pulselength**. In the design **pulselength** is being used to set the maximum value of a counter. When the maximum value is reached the counter is reset and the the audio signal is inverted. As a result of this, half a period of the audio signal is **pulslength** times **clk_div**.

5.6.2 Realisation and Verification

- ☐ Download the files for assignment 5 from Gitlab: https://gitlab.com/wlgrw/han-soc-assignment-vhdl-piano

For this assignment use the ENTITY that is specified in Code snippet 17.

Code snippet 17: ENTITY for COMPONENT pulselength2audio

```
-------------------------------------------------------------------
ENTITY pulselength2audio IS
   PORT(
      reset      : IN  std_logic;
      clk_dev    : IN  std_logic;
      pulslength : IN  integer RANGE 0 TO 131071;
      audiol,
      audior     : OUT std_logic
   );
END pulselength2audio;
-------------------------------------------------------------------
```

- ☐ Open with Quartus the project “pulslength2audio” in folder “assignment_5” or create the top-level ENTITY for component *pulslength2audio.*
- ☐ Verify that the ENTITY block meets the specification that is specified in Code snippet 17
- ☐ Design the VHDL for component *pulslength2audio* that meets the specification.
- ☐ Compile the VHDL code
- ☐ Simulate the functional operation of *pulslength2audio.*
  - Use the value for pulse length: 0, 100, 1000 and 7000 and **clk_div** is 6,25MHz
  - Verify if the period time of **audiol** and **audior**.

5.7 Assignment 6: Component tone_generation

In this assignment the COMPONENT *tone_generation* is created. For an overview see Figure 6.

5.7.1 Design

In this assignment the submodule is implemented that will combine all COMPONENTS that aid in generating the audio tones of the VHDL piano. COMPONENT *tone_generation* transforms the key from the keyboard into a audio tone with the aid of a 50 MHz clock.

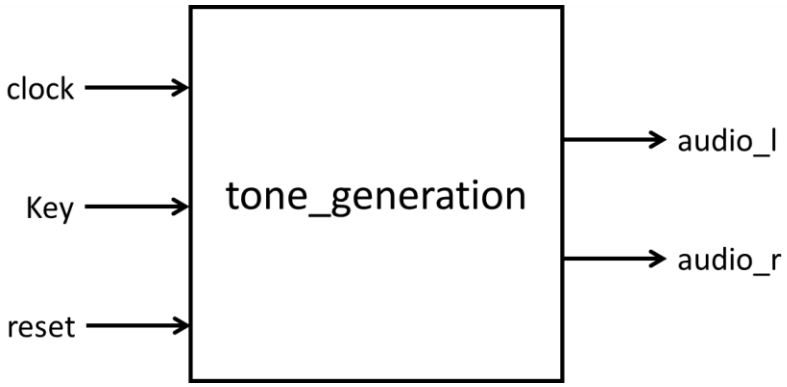

Figure 39: Inpout-process-output diagram for COMPONENT tone_generation

To combine all COMPONENTS structural VHDL is applied that will result in a view of the technology map as displayed in Figure 40.

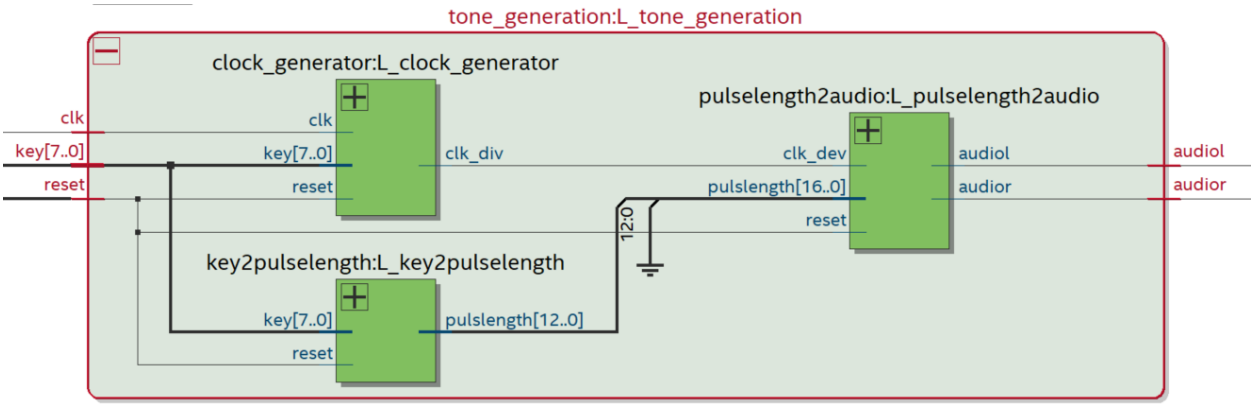

Figure 40: Technology map of tone_generation

5.7.2 Realisation and verification

- ☐ Download the files for assignment 6 from Gitlab: https://gitlab.com/wlgrw/han-soc-assignment-vhdl-piano

- ☐ Create a project in Quartus with the top-level ENTITY “tone_generation”. For this the VHDL file “tone_generation.vhd” in folder “assignment_5” can be used as a template.

For this assignment use the ENTITY that is specified in Code snippet 18.

Code snippet 18: ENTITY for COMPONENT tone_generation

```
-------------------------------------------------------------------
ENTITY tone_generation IS
   PORT(
      clk      : IN  STD_LOGIC;
      reset    : IN  STD_LOGIC;
      key      : IN  STD_LOGIC_VECTOR(7 downto 0);
      audiol,
      audior   : OUT STD_LOGIC
   );
END tone_generation;
-------------------------------------------------------------------
```

- ☐ Verify that the ENTITY block meets the specification that is specified in Code snippet 18.

- ☐ Design the VHDL for component *tone_generation* that meets the specification.

- ☐ Compile the VHDL code

- ☐ Verify the operation using simulation.

5.8 Assignment 7: VHLD piano top-level

In this assignment we will bring all the work from the previous assignments together in the top-level architecture of the VHDL piano.

5.8.1 Design

To combine all COMPONENTS, structural VHDL will be applied as being used in previous assignments. The result of the design is visualised in Figure 41 where the Technology map is displayed as presented by the RTL-viewer.

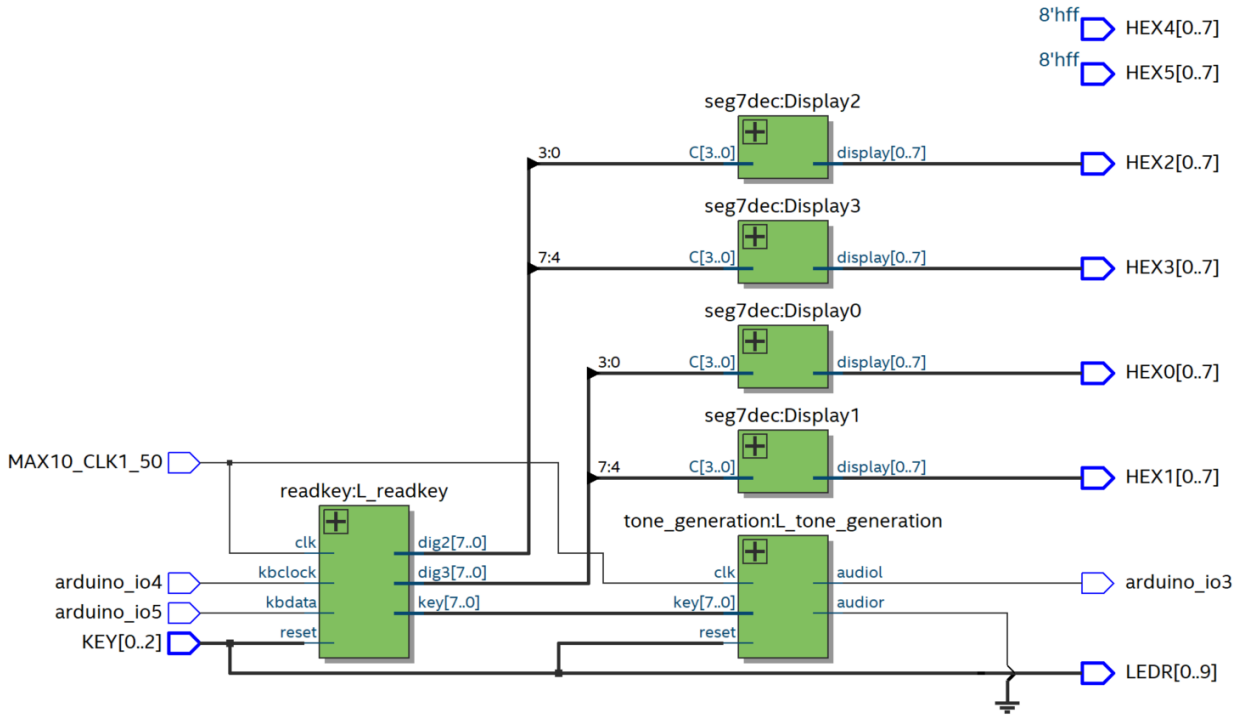

Figure 41: Technology map of the VHDL piano.

Because this final implementation will be connected to the PS/2 keyboard additional information is required for the VHDL implementation. This information is presented in Table 4.

Table 4: pin labels for I/O with DE10-Lite

Arduino Pin	FPGA pin	PORT label	Comment
D3	arduino_io3	Speaker	Audio to speaker
D4	arduino_io4	PS2_CLK	Clock from PS/2 keyboard
D5	arduino_io5	PS2_DAT	Data from PS/2 keyboard
D6	arduino_io6	PS2_CLK2	Clock to PS/2 keyboard (not used in VHDL Piano)
D7	arduino_io6	PS2_DAT2	Data to PS/2 keyboard (not used in VHDL Piano)

5.8.2 Implementation and verification

- ☐ Create a project in Quartus with the top-level ENTITY “piano”. For this the VHDL file “piano.vhd” in folder “assignment_7” can be used as a template.

- ☐ Verify that the ENTITY block meets the specification that is specified inCode snippet 19.

Code snippet 19: ENTITY block for top-level entity piano

```
-------------------------------------------------------------------
ENTITY piano IS
   PORT(
         MAX10_CLK1_50 : IN  STD_LOGIC; --! 50 MHz clock on DE10-Lite

         arduino_io4,                   --! PS2 keyboard clock signal
         arduino_io5 : IN  STD_LOGIC;   --! PS2 keyboard data signal
         arduino_io3 : OUT STD_LOGIC;   --! SPEAKER

         -- 7-segment displays HEX0 to HEX5
         HEX0,
         HEX1,
         HEX2,
         HEX3,
         HEX4,
         HEX5: OUT STD_LOGIC_VECTOR(0 to 7);

         -- Switches for reset (2)
         KEY: IN STD_LOGIC_VECTOR(0 to 2);

         -- LEDS
         LEDR: OUT STD_LOGIC_VECTOR(0 to 9)
        );
END piano;
-------------------------------------------------------------------
```

- ☐ Include all VHDL-files that have been created in previous assignments and that are part of the VHDL piano architecture:
  - clock_domain_crossing.vhd
  - clock_generator.vhd
  - constantKey.vhd
  - key2pulselength.vhd
  - pulselength2audio.vhd
  - readkey.vhd
  - seg7dec.vhd
  - showKey.vhd
  - tone_generation.vhd

- ☐ Design the structural VHDL for *piano* that meets the specification.

- ☐ Compile the VHDL code

- ☐ Verify the operation using simulation.

Use the testbench used for *readKey* while replacing *readKey* for the top-level entity of the VHDL piano.

- Assign pins to the hardware according to the information from Table 4.
- Test the VHDL piano with a PS/2 keyboard and record a video.
- Submit your video to YouTube and send the link to the teacher.
- Submit the project in a private GIT repository for assessment.

6 References

Chapweske, A. (2020, 3 18). *PS/2 Mouse/Keyboard Protocol*. Retrieved from Burton Systems Software: http://www.burtonsys.com/ps2_chapweske.htm

Electronic Tutorials. (2019, 11 1). *Display Decoder*. Retrieved from Electronic Tutorials: https://www.electronics-tutorials.ws/combination/comb_6.html

Renerta. (2020, 4 1). *Generate Statement*. Retrieved from VHDL Language Reference Guide: http://vhdl.renerta.com/mobile/source/vhd00033.htm

Suits, B. (2020, 3 22). *Notes*. Retrieved from Physics of music: https://pages.mtu.edu/~suits/notefreqs.html

Verma, S., & Dabare, A. (2020, 3 18). *Understanding Clock Domain Crossing Issues*. Retrieved from EE|Times: https://www.eetimes.com/understanding-clock-domain-crossing-issues/#

Welling, R. (2019). *Logic Circuits - Final-Assingment .* Arnhem: HAN.

Wikimedia. (2020, 3 18). *PS/2 port*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/PS/2_port

7 Appendix

7.1 Create testbench How to

This how to describes the creation of a testbench for simulation with Modelsim.

- Go in the menu to: Assignments > Settings
- In “EDA Toolsettings > Simulation” set: Output directory to “simulation/modelsim”
- If not opened yet: Open project
- Open VHDL code of the top-level entity.
- In the menu go to: “Processing > Start > Start Test Bench Template Writer” to create a testbench for the top-level entity.

  The testbench is generated and stored in:
  “C:\intelFPGA_lite\<path to your project> \simulation\modelsim”
- Copy the file “<top level entity name>.vht” to the same directory where the .vhd file is located of your top level entity.
- Rename file “<top level entity name>.vht” to file “tb_<top level entity name>.vhd”
- Import the testbench into your project
- Modify the signals to your need.

7.2 Analyse testbench How to

Add info from: https://class.ece.uw.edu/271/peckol/doc/DE1-SoC-Board-Tutorials/ModelsimTutorials/QuartusII-Testbench-Tutorial.pdf

8 Worksheets

8.1 Worksheet 1

	A	B	C	D	1	2	3	4	5	6	7
0											
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											
12											
13											
14											
15											