High Level IC Design

Programming project

Eleni Chatzikiyriakou
Summary

This report deals with the design of an alarm clock system for Xilinx Spartan 2 and 3 FPGA devices. It is a step by step description of the design entry, synthesis and implementation phases and provides relevant background on each step.
# Contents

Introduction ........................................................................................................................................... 5

Background ........................................................................................................................................... 5

1. Introduction to VHDL for logic synthesis and Implementation......................................................... 6
   Synthesis............................................................................................................................................... 8
   Implementation.................................................................................................................................. 12

2. Design entry .................................................................................................................................... 13
   2.1. Lab 5 ............................................................................................................................................... 13
   2.2. Lab 6 ............................................................................................................................................... 14
   2.3. Lab 7 ............................................................................................................................................... 15
   2.4. Lab 8 ............................................................................................................................................... 16
   2.5. Lab 9 ............................................................................................................................................... 17
   2.6. Lab 10 ........................................................................................................................................... 18
   Summary ............................................................................................................................................ 19

3. Synthesis......................................................................................................................................... 20
   COUNT4............................................................................................................................................. 20
   DDRV4 ............................................................................................................................................... 23
   Discussion.......................................................................................................................................... 25

4. Implementation................................................................................................................................. 26
   DISPLAY_MUX .................................................................................................................................... 26
   STORED_TIME_INT ............................................................................................................... 28
   Discussion.......................................................................................................................................... 28

Conclusions ......................................................................................................................................... 29

Appendix A .......................................................................................................................................... 30
   A.1 DDRV4 specification schematic ................................................................................................. 30
   A.2 COUNT4 specification schematic ............................................................................................... 31
Introduction

Field Programmable Gate Arrays (FPGAs) are hardware that can be programmed in a very low level. These devices are not created to serve a specific function, but rather to be later customized according to the requirements. Compared to full custom ICs, they provide the flexibility of reconfiguration and re-programming in conjunction with high speed. Reconfiguration reduces cost, since no new hardware is needed for upgrading to support new features or for correcting low level errors. Speed is not as high as in full custom hardware, but significantly higher than programmable devices (i.e. microcontrollers).

The configuration of a device is performed in simple steps gradually passing from a more abstract description of the desired functionality to a final hardware design description. Each step is performed with the aid of equivalent software tools. An outline of this process is given in the following section.

Background

Designing an FPGA device using a process of distinct steps provides the advantage of abstraction. A designer can create the description of a circuit that, with the help of various software tools, will be able to be implemented on different physical devices with minimal effort. The designer has the ability to intervene during each step in order to optimize the final circuit, but does not require full knowledge of the device at hand to provide the design description. In detail, some of the most important steps in the development process are:

1. Specification and Design entry
   Describes the requirements of the design, how are its various components are related to each other. This is done either in schematic form or by using hardware description languages such as VHDL.

2. Synthesis
   Synthesis creates the logic description based on the design entry description of the previous step. It also aims at optimizing the circuit that the designer wants to create.

3. Physical design
   The logic circuit provided by the previous step is mapped on the desired hardware. This step is concerned with fitting the various logic processes in physical structures. After this is completed, the final configuration can be passed to the device and dictate its functionality.

These steps are iterative, as various changes or error corrections are performed in each one providing feedback for the previous ones. Simulation of the design at any point during development is an important feedback mechanism as it can reveal any logical or functional errors.

In this report, ModelSim SE Plus 5.7f is used for the development of the VHDL code and the simulation of the design IO. For the synthesis and implementation, Xilinx ISE v12.3 is used with target devices of the Xilinx Spartan 3 family.
1. Introduction to VHDL for logic synthesis and Implementation

Labs 1-3

As an introduction to VHDL language, the most important elements of the switch design are examined. The code can be found on the website of the course (switch.vhd and t_switch.vhd). Each code begins with the library definitions which will be featured in all the rest of the designs. Library IEEE is explicitly called, while libraries std and work are visible by default. Libraries contain code, data types and other features that become available when the library is included in the design. The IEEE library is used for declaring signals of type std_logic. This signal has 9 levels of state contrary to a signal of type bit that can only be described by the states on ‘1’ and off ‘0’.

The file switch.vhd contains all that is needed to fully describe the circuit and is made up of two primary parts (figure 1.1):

1. The entity switch in which the statements of the input and output ports (pins) of the circuit are placed. A direction (IN, OUT, INOUT or BUFFER) is specified for each port along with its type (bit, integer, std_logic etc). This merely describes the interface of the circuit.

   ```vhdl
   entity SWITCH is
     port ( .. I/O ports.. );
   end SWITCH;
   ```

2. Architecture RTL of the entity ‘switch’ that contains the internal behaviour of the circuit. Processes, signals and component declarations can be placed here.

   ```vhdl
   architecture RTL of SWITCH is
   begin
     ALARM_SWITCH: process( ALARM_IN, SILENT )
     begin
       ...code for process...
     end process;
   end;
   ```

The process ALARM_SWITCH of the architecture is the only part of the VHDL code that is treated sequentially and not concurrently. The behaviour can be fully examined at the synthesis stage and the file switch.vhd has all the elements that the compiler needs to create the logic circuit (synthesize). However, it is much easier to have a first estimate of the I/O behaviour of the circuit in order to locate any existing logical errors. This is done in the simulation process.

---

![Figure 1.1. Structure of the switch.vhd file](image-url)
The simulation is performed using a test bench file (t_switch.vhd). The circuit has already been fully described in switch.vhd, therefore only instantiating it is necessary. To instantiate it a reference to the switch entity must be made. This is called a component and has a similar structure to the declaration of the entity.

```vhdl
component SWITCH
port (
    ALARM_IN   : in std_logic;
    SILENT     : in std_logic;
    ALARM_OUT  : out std_logic
);
end component;
```

The IO signals of this component are mapped to internal signals of the same number and equivalent type. The mapping is performed at the creation of an instance that can be called UUT (unit under test). After the mapping all the inputs of the design can be controlled and all the outputs monitored.

![Diagram of component](image)

**Figure 1.2. Basic elements of the switch test bench file**

![Simulation output](image)

**Figure 1.3. Simulation of the switch circuit**

Figure 1.2 shows the basic elements of the test bench file, while figure 1.3 shows the output of the simulation process. In detail, the latter shows the states of the input and output ports and how they change in time. Any logical inconsistencies or errors can be observed here. In this case, the design works as intended. The simulation starts with both the `alarm_in` and `silent` = ’0’, then `alarm_in` becomes ‘1’ which activates the buzzer, after which `silent` becomes ‘1’ and silences it.
**Synthesis**

The next step is synthesis of the design. The RTL (Register Transfer Level) schematic is produced. This provides a logic description of the circuit. This is, also, how the circuit would be described if using traditional non-reprogrammable logic. As shown in figure 1.4 the output is an AND gate with $\text{alarm\_in}$ and $\text{silent}$ signals as inputs. Signal $\text{silent}$ is negated before the AND operation. This can be described with the following expression:

$$\text{alarm\_in} \ \text{AND} \ (\text{NOT} \ \text{silent}) \Rightarrow \text{alarm\_out}$$

![Figure 1.4. Synthesized RTL schematic for SWITCH design](image)

While the RTL schematic provide a very intuitive diagram regarding the operation of the circuit, this however is not how the final hardware implementation will be. To understand the latter a brief description of the components of the FPGA technology must first be presented.

**FPGA description**

FPGAs consist of three primary components, namely, configurable logic blocks (CLBs), I/O blocks (IOBs) and a programmable interconnection network. Depending on the technology, an FPGA board can be programmed once or multiple times. Xilinx devices contain memory-based logic blocks that are re-configurable. A diagram that describes the primary parts of an FPGA board is depicted in figure 1.5 [1].
Figure 1.5. Basic structure of an FPGA board [1]

CLBs

The CLBs perform all the logic operations. The complete design is split into small parts and each part is implemented in one block. The operations are even further divided and assigned to logic cells (LCs) that reside inside the CLBs. Such distribution of functions depends on the architecture of the device. There are different structures of logic blocks for different FPGA devices.

For example, in Xilinx Spartan 2 and 3 devices two groups of two LCs each, make up a CLB. Each group makes up a slice. A simplified diagram of the structure of a slice for the Spartan 2 family is depicted in figure 1.6. Slices contain lookup tables (LUT), which perform most of the logic operations, carry and control logic and flip flops. The circuitry of these components includes multiplexers and logic gates that can be used individually for performing various tasks.
Figure 1.6. Structure of a Xilinx Spartan II CLB slice consisting of two LCs

**LUTs**

LUTs are also called function generators because of their operation. Given certain input values, they provide an equivalent output. One or more logic gates are combined in one LUT. The number of the input and output pins depends on the architecture. LUTs of Spartan 2 and 3 have four inputs and an output. It is not mandatory that all inputs are used when implementing a function. For example, to implement the gate shown in figure 1.4 two inputs are used. If, on the other hand, more input pins are needed, multiplexers in the carry and control logic are used to combine the LUTs.

The decision inside a LUT is made by the decoder. A conceptual diagram of a LUT is shown in figure 1.7. All the possible outputs are kept in storage cells. These cells are SRAM memory tables for Spartan 2 and later memory architectures for Spartan 3 devices.
Flip flops

As can be seen on the slice of figure 1.6, flip flops constitute the memory elements of the CLB. They contain the following IO pins:

- SET (S), sets the initial value
- RESET (R), resets to the initial value
- Output (Q)
- Delay (D), controls the output Q
- Clock (CK) for synchronization and
- Clock enable (EC)

Flip flops produce an output whose value depends on the initial value, the delay and whether there was a reset or not. When a clock pulse is needed, there are two possible cases: a) They either act edge-triggered, changing the state of the output only at a specific clock event (i.e. at the rising edge) or b) as level-sensitive latches changing the state of the output only during the time the clock is at a specific state (i.e. only during high or low state). Under certain conditions, unsynchronized latches (transparent) latches can be produced. As will be seen later, under these conditions a flip flop with no clock pulse is created at the synthesis stage, where a combination of logic gates would most probably be needed.

I/O blocks

These blocks are used for interfacing the board to external devices. They contain circuitry that can be conceptually seen as a group like in the case of a CLB. This group can contain flip flops, multiplexers and other components used to perform various functions with the IO data. For example, the data might not be routed directly to internal CLBs, but wait for a clock event to happen first.

Of the most important components of an IOB are IO buffers. These are used to isolate signals coming in and out of the device from the internal circuit. In the place and route diagram, the signals inside CLBs that are connected to buffers of IOBs take the suffix IBUF or OBUF.

Interconnection Network

Components inside a CLB or IOB are directly connected to each other. However, a direct connection of every CLB with all the others would dramatically increase hardware complexity. FPGA devices therefore include different levels of connections that can mainly be classified into two categories: Direct (usually local) and general interconnections (through a routing matrix). Direct connections like those inside a CLB are dedicated lines which means that they are much faster than general interconnections. It is therefore preferred that logic that uses the
same signals be implemented on the same CLBs. The general interconnections include components that control routing of the signalling which differs in different FPGA devices. Xilinx Spartan 3 devices there are four kinds of lines: long lines connect one out of every six CLBs, hex lines (one out of three), double lines (connect to every other CLB).

**Implementation**

The implementation of the SWITCH design is done for a Spartan 3 device. These devices are not very different than the Spartan 2 family. This step produces the routed design diagram which represents a detailed system level schematic of the design in hardware. All signals, connections, components and their functionality are given in this schematic. The involved components are those covered in the previous sections.

Figure 1.8 shows the part of the CLB that is used for implementing the logic gate shown in figure 1.4. The signals ALARM_IN and SILENT are provided by IOBs (not shown in the figure). They enter the LUT at inputs A1 and A3 respectively. The output of the LUT is sent through a multiplexer to a third IOB providing signalling to a device outside of the FPGA board.

![LUT Implementing AND2b1 gate of figure 1.4](image)

The implementation tool provides the following expression for the LUT:

\[ \langle G \rangle = \overline{A3} \cdot A1 \]

which corresponds exactly to the logic expression of figure 1.4 ((NOT A3) AND A1).

This concludes the development of a simple switch design. After this step the configuration can be downloaded to the device. The FPGA board has a special configuration memory which dictates the functionality of the device when loaded with the required data. Below follows a step by step development of the complete alarm clock design.
2. Design entry

In this section, a more detailed description of the VHDL language is given using the code for the alarm clock design. As already mentioned, VHDL provides the design entry description of the circuit. The specification is transformed to a form that the synthesis tool can understand and use.

2.1. Lab 5

Since the code for lab 4 is included in lab 5, the former was omitted from this report. The code for lab 5 is given in Appendix E.1.

A four-bit multiplexer is created. The basic components of DISMUX.vhd are the entity ‘DISMUX’ and the processes ‘SOUND’ and ‘DISP’. Processes were chosen because of the flexibility of the sensitivity list. The code inside the process is executed when one of the signals in the list changes. For example, DISPLAY_TIME must change each time CURRENT_TIME changes whatever the value of CURRENT_TIME may be. In this example, the sensitivity lists include signals SHOW_A and CURRENT_TIME.

One decision that has to be made is whether to put the code in one process or more. In this case, choosing one process can provide the desired results, but in order to make the code more intuitive, the functions were separated into two processes.

The test bench file T_DISMUX.vhd is fairly simple. Inside its process we try to examine each combination of CURRENT_TIME, DISPLAY_TIME and SHOW_A signals. Figure 2.1 shows that the design behaves as intended. It starts with SHOW_A = ‘0’, so the value displayed is that of CURRENT_TIME. When SHOW_A is raised, the display time changes to ALARM_TIME (at 10 ns) and then back to CURRENT_TIME when SHOW_A is reset (20 ns). Also, SOUND_ALARM is raised only when ALARM_TIME = CURRENT_TIME (30 ns).

![Figure 2.1. Simulation of DISMUX design](image)

We are therefore correctly driving DISPLAY_TIME and SOUND_ALARM, accounting for all possible changes of the signals that define their values. This is achieved by covering all the cases of possible states of the latter signals. This is an important issue in VHDL since not accounting for a specific case can produce unexpected results like latches. That is called an incomplete assignment. For example, if only the value of SOUND_ALARM for ALARM_TIME = CURRENT_TIME was specified then what assumption should the compiler make when ALARM_TIME ! CURRENT_TIME?

Latches are flip flops whose output immediately follows the input when the enable is active and whose output freezes when the enable is inactive. Instead, in this case, a logic gate is needed (LUT).
2.2. Lab 6

The code of the previous exercise is modified to provide an output for a seven segment display (Appendix E.3). Process DISP changes to accommodate the translation of the 4-digit DISPLAY_TIME number to the 7-segment display equivalent code. Constant signals are provided to help in this translation; one for each decimal digit (ZERO_SEG, ONE_SEG etc). Also, internal signal display_time is introduced. This is used as an intermediate signal that holds the four-digit (binary) value to be displayed.

Before performing the translation, process SOUND is substituted with an equivalent concurrent statement:

\[
\text{SOUND\_ALARM} \leftarrow '1' \text{ when } \text{CURRENT\_TIME} = \text{ALARM\_TIME} \text{ else } '0';
\]

This statement will be executed on each change of the signals on its right side. It is an alternative to process SOUND. This is also how the translation will be performed.

It is important that display_time is correctly specified before the translation. The two functions cannot be performed in one process because a change in the value of a signal only happens after the end of the process. If we placed everything in process DISP then each time it would execute the 7 segment display would take the value that was previously assigned to display_time. We can therefore either use the style of the aforementioned concurrent statement for the translation, or create a second process. The former was chosen because it is more compact and straight-forward.

Simulation results of the test bench in Appendix E.4 are presented in figure 2.2. The 7 segment display correctly changes from ‘0’ to ‘1’, ‘2’ etc. as current_time increases. At current_time = ‘2’, sound_alarm is raised according to alarm_time and when show_a is set (20 ns) the display changes equivalently.

Figure 2.2. Simulation results for seven segment display driver design

A more thorough understanding of VHDL processes was achieved. Also, a case where a single process could not provide the desired results was presented. Concurrent statements replaced some of the processes. While code in processes is sequential, different processes occur concurrently. This helps us understand why a concurrent statement can replace a process. Decision on one method over the other depends on the requirements. Because concurrent statements are closer to the hardware logic, they are preferred by the writer when they can provide the desired functionality.
2.3. **Lab 7**

The completed code for this lab is given in Appendix E.5. First, an internal signal `STORED_TIME_INT` is introduced which can be read, contrary to the output signal `STORED_TIME` that can only be written to. The former signal drives the latter and therefore acts as an intermediate.

According to the specification, we want to reset the signal at ‘RESET’ and increase its value at ‘INC’. In this case we don’t have to worry about incomplete statements like in the cases where RESET or INC = 0. The explanation for this is provided in the Synthesis section of this report, where a description of the counter that is created is provided. We code this by using a simple if-elsif statement.

In this lab, the concept of synchronous design is introduced. All alterations to signals happen in the event of an external signal clock pulse. As already stated, a clock in the design means the creation of a synchronous flip flop. In this example, a change in `STORED_TIME_INT` does not happen until the rising edge of a clock pulse. This gives time for other assignments to complete before carrying on with the operation and allows the inputs to ‘settle’ to their correct value.

From the simulation output in figure 2.3 we can deduce that there are two conditions that must be met for an increase in `STORED_TIME` to happen. Both a clock rise and INC=1 must happen at the same time. If only one is true then the operation is not completed. The same applies for resetting it.

![Figure 2.3. Simulation results for COUNTER design](image)

Except for synchronization, another important issue in this design is the connection between the internal signal `STORED_TIME_INT` and the output `STORED_TIME`. We can verify that the connection works by including the internal signal in the simulation diagram. This is especially useful in later exercises where complexity increases.
2.4. Lab 8

In this lab the concept of hierarchy is introduced. By including instances of entities inside other entities code complexity is reduced and code re-use increased. The instances in this case are DDRV components connected together as shown in Appendix A.1. Instances were introduced in this report with test bench files. The concept is the same but in this case the functionality of the final circuit is expanded. This means that contrary to the instances in the test bench files, these instances will be synthesized and included in the final circuit. The code of this part is given in Appendix E.7.

To complete this code, we first had to make a note of what functionality was already provided by the included components. Regarding signal SOUND_ALARM, equivalent signals inside each instance were used to check whether CURRENT_TIME = ALARM_TIME, because that is when SOUND_ALARM is set for each digit. In DDRV4, a four-bit internal signal (sound_alarm_int) was created and each of its bits was connected to each SOUND_ALARM output of the DDRV instances. The when-else statement was then used to check whether all digits of the clock equal ALARM_TIME so that the general SOUND_ALARM is set:

```
SOUND_ALARM <= '1' when sound_alarm_int = "1111" else '0';
```

For the DISPLAY signals all functionality is already provided by the instances. All that is needed is connecting each DDRV4 output signal to the right DDRV instance output signal. Also, SHOW_A signal is common for the whole design, so the same signal is connected to all equivalent inputs.

Simulation results are shown in figure 2.4. Every 10 ns CURRENT_TIME increases 1 sec starting from 06:59. ALARM_TIME is set to 07:00. Thus, after the first 10 ns SOUND_ALARM is raised. This proves that the concurrent statement is working properly. Also, the 7-segment display is correctly showing the equivalent codes. For example, at 0 ns LS_HR is 6_{10}, which corresponds to the code "0000010" (SIX_SEG). Lastly, after 30 ns, SHOW_A is set which only changes LS_MIN at the display because the rest of the digits are the same (07:01).

![Figure 2.4. Simulation results for DDRV4 design](image)

Extending functionality of the included components requires working on some IO signals of the instances. This can be achieved by creating internal signals and working on them. The difference of this concept in relation to the internal signal in lab 7 (STORED_TIME_INT) is that in the latter we wanted to read an output of the design,
which is impossible to do. In DDRV4, we can read the outputs of the internal instances but a function is required to be performed before guiding them to the output.

While working for SOUND_ALARM, the question of whether it should be synchronous or not arose. Because this signal depends only on the counter in relation to ALARM_TIME, it is very unlikely that the input will oscillate. The signal thus remained asynchronous.

2.5. Lab 9

Again, using levels of hierarchy, a four-digit counter is created. The COUNT instances for this design and their interconnections are depicted in Appendix A.2. To be able to work on the signals of the instances, a read/control list, as shown in table 2.1, was created.

<table>
<thead>
<tr>
<th>READ</th>
<th>CONTROL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN, HOUR, CLK, STORED_TIME</td>
<td>INC, RESET</td>
</tr>
</tbody>
</table>

Table 2.1. Read/control list for signals in COUNT4 design

This table shows that to be able to control the STORED_TIME signals of the instances, internal signals (LS_MIN_ST, MS_MIN_ST etc) have to be created. Internal signals are also created for the INC and RESET since no inputs to COUNT4 are provided that can be connected to them. Instead, they depend on the MIN and HOUR inputs of the COUNT4 entity. The complete code is given in Appendix E.9.

To develop the behaviour of the design, nine concurrent statements were created. Specifically, four of them control the increase of each register of the COUNT instances i.e.

-- XX:X9 increase MS min when 9 minutes + 1 minute
MS_MIN_INC <= '1' when (LS_MIN_ST = "1001" AND MIN = '1')
   else '0';

four control the RESET i.e.

-- rollover when XX:X9 + minute increase
LS_MIN_RESET <= '1' when (RESET = '1')
or (LS_MIN_ST = "1001" AND MIN = '1')
   else '0';
and one intermediate is used for indicating that an hour increase should happen:

-- XX:59 + 1 minute
HOUR_INT <= '1' when (LS_MIN_ST = "1001" AND MS_MIN_ST = "0101" AND MIN = '1')
   else '0';

The absence of clock use in the COUNT4 code is attributed to the fact that synchronization is provided by the COUNT instances. However, since the clock is external to the design, it must be given as input for COUNT to use. Revisiting COUNT, an explanation for the usefulness of synchronization in the design can now be provided. An illustrative example is given in figure 2.5. Signalling of LS_HR_INC is provided by two different paths. One is the
The output from the first path may initially toggle. By using synchronization for \textit{LS\_HR\_INC}, the first signal will have time to reach its final state before making the assignment.

For this test bench, in order to reduce the required code, MIN signal was kept high while checking for rollover functionality. The correct functionality of MIN = ‘0’ can be seen at the beginning of the first simulation (Appendix B.1.a).

In this lab it became obvious that VHDL is not as software programming. It describes hardware which is mainly composed of signals that change states triggered by specific events. This assumption leads to the use of concurrent statements that greatly simplify the produced code. If synchronization is required, concurrent statements are maybe not the best choice. But since clock functionality was provided by the internal entities, these statements are expected to produce the desired circuit.

\section*{2.6. Lab 10}

The complete alarm design does not need any new code regarding its behaviour and we are primarily concerned with doing the right connections. The connections can be seen in the code listed in Appendix E.11. To correctly connect the instances together we created internal signals for all the outputs of internal entities that had to be connected to other internal entities. All the rest of the connections were made directly to IO signals of the ALARM design. \textit{ALARM\_TIME} and \textit{CURRENT\_TIME} are both \textit{COUNT4} instances that need the functionality of \textit{MIN} and \textit{HOUR} increase. This functionality can be thought of as four buttons that the user can press to increase either one of them. When an \textit{ALARM\_TIME} increase is needed, the user can simultaneously press the \textit{SHOW\_A} button and \textit{ALARM\_MIN} or \textit{ALARM\_HOUR} to observe alarm time changes.

In the test bench, a function was created to facilitate reading the output of the seven segment display. It converts 7-digit code in decimal notation. Since the test bench is not synthesized we do not have to worry about whether this function is implementable or not. The procedure followed is:

- Initialize signals and alarm clock at 07:00, verify that alarm time was set (Appendix B.2.a).
- Start the clock by setting MIN = 1
- Check RESET, SOUND and SILENT correct operation (Appendix B.2.b)
- Check rollover from 23:59 to 00:00 and to 00:10 (Appendix B.2.c)

When checking if the global RESET works properly, we notice that SOUND_ALARM is raised since both ALARM and CURRENT time point to 00:00 hours. This feature is used to check SOUND_ALARM and SILENT operation. SOUND_ALARM is also raised at the very beginning of our simulation (Appendix B.2.a). This is because the digits of both the clock and the alarm are initialized by default to 0.

It is also noticed that the SILENT operation is not exactly realistic. For example, in a real alarm clock the silent button would be pressed once and it would stop the buzzer completely. In this design, in order for that to happen SILENT has to stay high at least until the next minute increase. This functionality could have been embedded in the design but was omitted for reasons of simplicity.

Summary

This concludes the design entry phase of the procedure. The operation of the design was split in parts and assigned to different entities. Some of these entities were embedded in others. This hierarchical separation allows the code to become more intuitive and reduced in size. When working with VHDL it is very helpful having some essential knowledge of synthesis processes (presented in the next section). With this knowledge, the chances of creating code that is implementable and that will be translated to the desired functionality are increased. In the next section this translation is presented in terms of small parts of the design.
3. Synthesis

As already mentioned, the RTL schematic provides the logic description. It dictates the logic gates, registers etc. that need to be implemented on the desired FPGA board. For different target devices, synthesis might produce different RTL outputs.

The complete RTL schematic for the alarm design is shown in Appendix C.1. This is just the interface of the design and the internal entities. It does not provide any information about the inner logic. The latter can be understood examining each individual component. For example, the SWITCH logic has been described in section 1 of this report (figure 1.4). Functionality of the COUNT4 instances is described next.

COUNT4

It is obvious that the two COUNT4 instances produced the same RTL output therefore only ALARM_COUNTER is examined. Examination starts from a simple expression of its code:

\[
\text{LS\_MIN\_INC} \leftarrow \text{'1'} \text{ when (MIN = '1') else '0'}; \\
\]

which is a direct connection of the input ALARM_MIN to the INC input of COUNT_LS_MIN, as shown in figure 3.1.

A little more complex example is the following:

\[
\text{HOUR\_INT} \leftarrow \text{'1'} \text{ when (LS\_MIN\_ST = "1001" AND MS\_MIN\_ST = "0101" AND MIN = '1') else '0'}; \\
\]

In this expression, for each condition that has to be true, a logic gate is created. For example, for \(LS\_MIN\_ST = "1001"\) to be true, the following must hold:

\[
(LS\_MIN\_ST_3) \text{ AND (LS\_MIN\_ST_0) AND NOT (LS\_MIN\_ST_1) AND NOT (LS\_MIN\_ST_2)} = 1 \\
\]

having in mind that \(LS\_MIN\_ST_0\) is the first and \(LS\_MIN\_ST_3\) is the last bit. This immediately creates a logic gate, as shown in figure 3.2.b. Notice that the input pins of this gate are a scrambled version of the pins of COUNT_LS_MIN. This probably happens because the synthesis tool has to follow some specifications regarding the logic gates that it creates. In other words, as soon as it recognises the functionality of the code, it maps it to specific logic structures.
Another logic gate is created for $MS_{\text{MIN}_ST}$ (figure 3.2.a). Along with signal $ALARM_{\text{MIN}}$, the outputs are fed to a third AND gate that combines the three expressions (figure 3.2.c). The output of this last gate is later fed to circuits that use HOUR_INT to check whether an hour increase takes place. If HOUR_INT, which checks when an hour increase happens, was not created in the code, the synthesis tool would probably create this last gate on its own and use it each time the examined expression was called in the code. This confirms that HOUR_INT created no further functionality; instead it was used to increase code readability.

Figure 3.2. RTL schematic of a logic expression
The complete RTL schematic for the COUNT4 component can be seen in Appendix C.2. Apart from the complex diagram of logic gates it also contains the four COUNT instances which are just counters increasing the value of each digit.

Counters are circuits that increment or decrement a stored value by one. In the alarm clock design, four-bit counters are needed to increment the STORED_TIME signals. The synthesis schematic for one of them is shown in figure 3.3. It is the job of the synthesis tool to ‘spot’ this kind of circuits in the code. In this case, the code that created the up counter is the following:

```vhdl
if CLK'event and CLK = '1' then
  if RESET = '1' then
    STORED_TIME_INT <= "0000";
  elsif INC = '1' then
    STORED_TIME_INT <= STORED_TIME_INT + 1;
  end if;
end if;
```

This is the case that an incomplete statement is not a problem. Recalling the theory, transparent latches created from incomplete statements are unwanted since most of the time logic gates are needed in their place. Latches are a hassle because of their lack of synchronization. In this case, a latch is needed, and the synchronization provided to it converts it to an edge-triggered flip flop. This assures that no unwanted STORED_TIME increase will happen since this operation will be performed after the inputs have settled.

All counters in this design are synchronous. The synthesis tool recognised a 4-bit synchronous up counter with synchronous reset (Sclr) from the code that was provided. Each time INC signal is high, STORED_TIME increases at the next clock event. Resetting also needs a clock event to happen.

There are various ways that a counter can be implemented on hardware (with the use of flip flops, registers etc.), but at this point the exact implementation is unknown. It will completely depend on the specific target board.
**DDRV4**

As described in the specification, DDRV4 is a multiplexer that selects the appropriate display according to the ‘show alarm’ signal. Each digit of the alarm clock is a DDRV instance, therefore also a MUX, having the common SHOW_A signal as the selector.

Inside the DDRV components, a small logic circuit similar to that in figure 3.4 is created for each bit of the DDRV digit. This is a one-bit MUX circuit. The output of the two AND gates can never be high at the same time because one of them has the inverted SHOW_A signal as input. The OR gate then lets the signal that SHOW_A dictates pass through.

![Diagram](image)

**Figure 3.4. One bit multiplexer - Selects between ALARM_TIME(0) and CURRENT_TIME(0)**

Four such circuits control one digit, as shown in the RTL diagram of a DDRV component in Appendix C.4. The code that created them is the following:

```plaintext
if( SHOW_A = '0') then
    display_time <= CURRENT_TIME;
else
    display_time <= ALARM_TIME;
end if;
```

The other two components of a DDRV instance are a comparator and an Mrom (figure 3.5). The first compares an ALARM_TIME digit with a CURRENT_TIME digit. DataA(3:0) is ALARM_TIME(3:0) and DataB(3:0) is CURRENT_TIME(3:0). If they match, SOUND_ALARM is set for this digit. The code that created the comparator is the following:

```plaintext
SOUND_ALARM <= '1' when ALARM_TIME = CURRENT_TIME
    else '0';
```
Recalling the code of DDRV4 in Appendix E.7, a case statement was created for the translation of a four-bit digit value, to a 7-segment equivalent code. The Xilinx Synthesis Technology User Guide [7] mentions the following regarding to an Mrom:

"a ROM can be inferred when all assigned contexts in a Case or If...else statement are constants"

In terms of logic circuit notation, an Mrom chooses a specific output given a certain input. This resembles very much a LUT as described in section 1 of this report. However, the output of the Mrom is a seven bit signal and, as already mentioned, LUTs of Xilinx devices have only one output. How this is implemented in hardware is described in the next section (Implementation).

![Figure 3.5. Comparator and ROM components of DDRV4 instance](image)

Since DDRV4 contains very little behavioural code it only has one logic component, an AND gate. As can be seen in figure 3.6 its job is to check when all the digits of the CURRENT_TIME equal those of ALARM_TIME and raise the general SOUND_ALARM signal:

```
SOUND_ALARM <= '1' when sound_alarm_int = "1111"
    else '0';
```
Discussion

This concludes the description of the RTL schematic. This schematic is very much dependent on the tool used for synthesizing. It is this tool that ‘recognizes’ logic circuits in our code (gates, counters, comparators etc) and combines them together to form a logic description. Many of these circuits are still very abstract because only after the implementation phase we can be sure of the final hardware structure. The RTL description, however, is very close to the final design and has the advantage of being more comprehensible and intuitive than the device-specific description.
4. Implementation

Implementation of the design will be made for two Xilinx devices of the Spartan 3 family, the XC3S50 and the XC3S5000. These devices are structurally similar, but have significant differences in their sizes. For example, the first incorporates 50K system gates (hence the name) while the second 5M. Their differences can be observed in figure 4.1, where a table with the device utilization summary is presented for each device. It is shown, for example, that the 37 occupied IOBs of the alarm clock design constitute a 29% of the available IOBs in XC3S50 and only 5% of the XC3S5000.

<table>
<thead>
<tr>
<th>Target Device:</th>
<th>xc3s5000-3fp2900</th>
<th>No Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Utilization</td>
<td>Used</td>
<td>Available</td>
</tr>
<tr>
<td>Number of Slice Flip Flops</td>
<td>32</td>
<td>66,560</td>
</tr>
<tr>
<td>Number of 4 Input LUTs</td>
<td>111</td>
<td>66,560</td>
</tr>
<tr>
<td>Number of occupied Slices</td>
<td>62</td>
<td>33,280</td>
</tr>
<tr>
<td>Number of Slices containing only related logic</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>Number of Slices containing unrelated logic</td>
<td>0</td>
<td>62</td>
</tr>
<tr>
<td>Total Number of 4 input LUTs</td>
<td>111</td>
<td>66,560</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>37</td>
<td>533</td>
</tr>
<tr>
<td>Number of BUF/MUXs</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Average Fanout of Non-Clock Nets</td>
<td>3.97</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target Device:</th>
<th>xc3s50-5pq208</th>
<th>No Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Utilization</td>
<td>Used</td>
<td>Available</td>
</tr>
<tr>
<td>Number of Slice Flip Flops</td>
<td>32</td>
<td>1,536</td>
</tr>
<tr>
<td>Number of 4 Input LUTs</td>
<td>111</td>
<td>1,536</td>
</tr>
<tr>
<td>Number of occupied Slices</td>
<td>62</td>
<td>768</td>
</tr>
<tr>
<td>Number of Slices containing only related logic</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>Number of Slices containing unrelated logic</td>
<td>0</td>
<td>62</td>
</tr>
<tr>
<td>Total Number of 4 input LUTs</td>
<td>111</td>
<td>1,536</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>37</td>
<td>124</td>
</tr>
<tr>
<td>Number of BUF/MUXs</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Average Fanout of Non-Clock Nets</td>
<td>5.97</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1. Device utilization summary for XC3S5000 (left) and XC3S50 (right)

Family 3 is very similar to Spartan 2 but contains some advanced features like newer memory technologies. Other differences are related to naming of components and routing lines which change to make them more easily distinguishable.

DISPLAY_MUX

Examination begins with XC3S50 device. The signal DISPLAY_MUX/DDRV_LS_MIN/display_time<0> can be found under the BEL category (basic element of logic) in the routed design editor. It resides inside the slice named “DISPLAY_LS_MIN_6_OBUF”. The number of the slice is SLICE_X5Y31 where the numbers after the X and Y provide information about the location of the slice inside the CLB (left, right etc) and the location of the CLB on the board. This is a SLICEL (right-hand or logic only) as shown in figure 4.2. This specific slice is only used for logic only and not for memory operations.
Figure 4.2. Structure of a CLB of the Spartan 3 family of FPGAs [6]

Figure 4.3 contains the LUT that is used for the signalling of display_time<0>. The ISE informs us of the functionality of the LUT with the following expression:

\[ G = (\neg A_4 \cdot A_1) + (A_4 \cdot A_2); \]

Where A1 and A2 are shown in figure 4.3, A3 is the input buffer of the ‘show alarm’ button and G is the output. This is essentially an implementation of the multiplexer for a display_time component as seen in figure 3.4. It selects the correct display_time signal depending on the show_alarm flag. All three logic gates of figure 3.4 are contained in this LUT. This significantly saves hardware resources. As a reminder, this is just one bit of the four-bit LS_MIN digit.

As shown in this figure, the output bypasses the flip flop following the LUT, therefore there is no clock pulse controlling this signal. Changes in the output immediately follow changes in the input. This is not a problem for the design since both STORED_TIME input signals are synchronized and an immediate response from the SHOW_A button is desirable.

The second LUT of the same slice is an implementation of one bit of the 7-segment display translation. It is one of the output pins of the Mrom component of figure 3.5. There exists one LUT for each one of them. The
inputs are all DDRV_LS_MIN/display_time signals. With further examination it is observed that the LUTs for most of the 7-segment bits reside in common slices with display_time signals, which is convenient since at least one input for the former is the output of the latter.

![Diagram of LUT for one output digit of the Mrom component of figure 3.5](image)

**STORED_TIME_INT**

Signal ALARM_COUNTER/COUNT_MS_HR/stored_time_int<0> is under the components category and resides inside a slice of the same name. The routed design schematic of the circuit that feeds this signal is given in Appendix D.1. In this case the inputs bypass the LUT and are fed to the synchronous flip flop. The names of the input signals and their connections to the flip flop are as follows: MS HR stored_time_int<0> connects to the D input, MS HR INC to Clock Enable, MS HR RESET to Reset and finally CLK. Clock enable (CE) of the flip flop enables its functionality. Whenever it is high, the output takes the value opposite of the D input as long as there is a specific clock event according to the flip flop configuration (level-sensitive, edge-triggered). In essence, this flip flop changes the last bit (least significant) of the MS HR digit synchronously whenever there is an INC signal on the line, or resets it at RESET. This signal represents one of the four bits of an up counter equivalent to the one in figure 3.3.

Because the last bit will always toggle from one value to the other, there is no need for additional circuitry to control it. Stored_time_int<1>, however, toggles after two changes of stored_time_int<0>. To accomplish this functionality a LUT is used in conjunction with a flip flop that controls timing, as shown in Appendix D.2. The inputs to the LUT are stored_time_int 0 and 1, with a XOR operation performed between them (modulo 2 addition). In a similar manner, there exists a slice that performs the operations for stored_time_int 2 and 3, with the help of LUTs.

Implementing the same design for the Spartan XC3S5000 device it is observed that there are no differences in the structure inside the CLBs. It is, however, obvious that, as described earlier, the design occupies much less space on the board.

**Discussion**

It is now apparent that the implementation tool uses output produced by the second step to provide the description of the hardware and not the initial VHDL description. This requires the development process to be monitored in each step to avoid unwanted circuitry from being created. The device specific description is not as easily comprehensible as the RTL output, therefore, the latter can also be used as a guide for the designer.
Conclusions

All the development phases have been completed successfully. In section 2, the design entry procedure was described while the last two sections were concerned with the synthesis and implementation of the design. Because the main objective of this report was to provide adequate description of each step, no feedback was provided for optimization purposes. Also, no changes were made to the produced schematics. In a real world example, the last two procedures are very common and constitute some of the reasons re-configurable hardware is constantly gaining popularity. It also became apparent that with just the knowledge of VHDL language and basic operation of FPGA devices custom circuits could be created. This low level of complexity is another factor increasing the use of these devices. Future directions include a higher level of abstraction and more automatic processes to eliminate the need of human intervention.
Appendix A

System level schematics

A.1 DDRV4 specification schematic
A.2 COUNT4 specification schematic
Appendix B

Simulation Results

B.1 Simulation results for COUNT4.vhd

B.1.a. Initialize, RESET and HOUR increase

B.1.b. Rollover from 00:09 to 00:10

B.1.c. Rollover from 00:59 to 01:00

B.1.d. Rollover from 09:59 to 10:00

B.1.e. Rollover from 11:59 to 12:00
B.1.f. Rollover from 23:59 to 00:00

B.2 Simulation results for ALARM.vhd

B.2.a. Set ALARM_TIME to 07:00 (65 ns), SHOW_A (70 ns)
B.2.b. RESET (2090 ns), SOUND_ALARM (2095 ns) and SILENT (2130 ns) functionality

B.2.c. Rollover from 23:59 to 00:00 and to 00:10
Appendix C
Synthesis results

C.1 RTL schematic of ALARM.vhd
C.2 RTL schematic of COUNT4 instance
C.3 RTL schematic of DDRV4 instance
C.4 RTL schematic of DDRV instance
Appendix D

Implementation results

D.1. Routed design schematic for stored_time_int<0>
D.2. Routed design schematic for stored_time_int<1>
Appendix E

Alarm clock complete code

E.1 DISMUX.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity DISMUX is
port (
    ALARM_TIME : in std_logic_vector(3 downto 0);
    CURRENT_TIME : in std_logic_vector(3 downto 0);
    SHOW_A : in std_logic;
    DISPLAY_TIME : out std_logic_vector(3 downto 0);
    SOUND_ALARM : out std_logic
);
end DISMUX;

architecture RTL of DISMUX is
begin
--------------------------------------------
--Processes DISP
--Change DISPLAY_TIME according to SHOW_A
--------------------------------------------

DISP: process( SHOW_A, CURRENT_TIME )
begin
    if (SHOW_A = '0') then
        DISPLAY_TIME <= CURRENT_TIME;
    else
        DISPLAY_TIME <= ALARM_TIME;
    end if;
end process;

--------------------------------------------
--Process SOUND
--Sound the alarm at ALARM_TIME
--------------------------------------------

SOUND: process( CURRENT_TIME )
begin
    if (ALARM_TIME = CURRENT_TIME) then
        SOUND_ALARM <= '1';
    else
        SOUND_ALARM <= '0';
    end if;
end process;

end RTL;
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity T_DISMUX is
end T_DISMUX;

architecture TEST of T_DISMUX is

--instance
component DISMUX
port (
    ALARM_TIME : in std_logic_vector (3 downto 0);
    CURRENT_TIME : in std_logic_vector (3 downto 0);
    SHOW_A : in std_logic;
    DISPLAY_TIME : out std_logic_vector (3 downto 0);
    SOUND_ALARM : out std_logic
);
end component;

--internal signals
signal T_CURRENT_TIME : std_logic_vector (3 downto 0);
signal T_ALARM_TIME : std_logic_vector (3 downto 0);
signal T_SHOW_A : std_logic;
signal T_DISPLAY_TIME : std_logic_vector (3 downto 0);
signal T_SOUND_ALARM : std_logic;

begin

-- port map
UUT : DISMUX port map ( ALARM_TIME => T_ALARM_TIME,
    CURRENT_TIME => T_CURRENT_TIME,
    SHOW_A => T_SHOW_A,
    DISPLAY_TIME => T_DISPLAY_TIME,
    SOUND_ALARM => T_SOUND_ALARM );

---------------------------------------

-- Process to generate stimulus
---------------------------------------

STIMULUS: process
begin
    T_ALARM_TIME <= "0010"; -- set alarm to ‘2’
    T_CURRENT_TIME <= "0000"; -- start the time at ‘0’
    T_SHOW_A <= '0'; -- show current time
    wait for 10 ns;
    T_SHOW_A <= '1'; -- show alarm time
    wait for 10 ns;
    T_SHOW_A <= '0'; -- show current time
    wait for 10 ns;
    T_CURRENT_TIME <= "0010"; -- current time = ‘2’
end process;

end TEST;
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity DDRV is
port (  
  ALARM_TIME : in std_logic_vector (3 downto 0);
  CURRENT_TIME : in std_logic_vector (3 downto 0);
  SHOW_A : in std_logic;
  SOUND_ALARM : out std_logic;
  DISPLAY : out std_logic_vector (6 downto 0)
);
end DDRV;

architecture RTL of DDRV is
begin

-- constants for the segments to light for each digit 0 to 9 on a 7 segment display
-- 0 = LED segment ON, 1 = LED segment OFF
constant ZERO_SEG : std_logic_vector(6 downto 0):= "0100000";
constant ONE_SEG  : std_logic_vector(6 downto 0):= "1111001";
constant TWO_SEG  : std_logic_vector(6 downto 0):= "1000100";
constant THREE_SEG : std_logic_vector(6 downto 0):= "1010000";
constant FOUR_SEG : std_logic_vector(6 downto 0):= "0011001";
constant FIVE_SEG : std_logic_vector(6 downto 0):= "0010010";
constant SIX_SEG : std_logic_vector(6 downto 0):= "0000010";
constant SEVEN_SEG : std_logic_vector(6 downto 0):= "1111000";
constant EIGHT_SEG : std_logic_vector(6 downto 0):= "0000000";
constant NINE_SEG : std_logic_vector(6 downto 0):= "0011000";

--internal signal declarations, use this signal to pass the selected time
--signal to the seven segment translation process
signal display_time : std_logic_vector( 3 downto 0 ) := "0000";

begin

--sound the alarm at ALARM_TIME
SOUND_ALARM <= '1' when ALARM_TIME = CURRENT_TIME else '0';

--change display_time
DISP: process( SHOW_A, CURRENT_TIME, ALARM_TIME )
begin
if (SHOW_A = '0') then
    display_time <= CURRENT_TIME;
else
    display_time <= ALARM_TIME;
end if;
end process;

-- translate from four-digit binary value to 7-digit display code
DISPLAY <= ZERO_SEG when display_time = "0000" else
           ONE_SEG when display_time = "0001" else
           TWO_SEG when display_time = "0010" else
           THREE_SEG when display_time = "0011" else
           FOUR_SEG when display_time = "0100" else
           FIVE_SEG when display_time = "0101" else
           SIX_SEG when display_time = "0110" else
           SEVEN_SEG when display_time = "0111" else
           NINE_SEG when display_time = "1001" else
           EIGHT_SEG;
end RTL;

E.4 T_DDRV.vhd

Library IEEE;
use IEEE.Std.Logic_1164.all;

entity T_DDRV is
end T_DDRV;

architecture TEST of T_DDRV is

component DDRV
port (
    ALARM_TIME: in std_logic_vector (3 downto 0);
    CURRENT_TIME: in std_logic_vector (3 downto 0);
    SHOW_A : in std_logic;
    DISPLAY: out std_logic_vector (6 downto 0);
    SOUND_ALARM: out std_logic
);
end component;

signal T_CURRENT_TIME: std_logic_vector (3 downto 0);
signal T_ALARM_TIME: std_logic_vector (3 downto 0);
signal T_SHOW_A : std_logic;
signal T_DISPLAY: std_logic_vector (6 downto 0);
signal T_SOUND_ALARM: std_logic;

begin

    uut : DDRV port map ( ALARM_TIME => T_ALARM_TIME,
                           CURRENT_TIME => T_CURRENT_TIME,
                           SHOW_A => T_SHOW_A,
                           DISPLAY => T_DISPLAY,
                           SOUND_ALARM => T_SOUND_ALARM );
-- Process to generate stimulus ...

STIMULUS: process
begin
    T_ALARM_TIME <= "0010";
    T_CURRENT_TIME <= "0000";
    T_SHOW_A <= '0';
    wait for 5 ns;

    T_CURRENT_TIME <= "0001";
    wait for 5 ns;
    T_CURRENT_TIME <= "0010";
    wait for 5 ns;
    T_CURRENT_TIME <= "0011";
    wait for 5 ns;
    T_CURRENT_TIME <= "0100";
    T_SHOW_A <= '1';
    wait for 5 ns;
    T_CURRENT_TIME <= "0101";
    wait for 5 ns;
    T_CURRENT_TIME <= "0111";
    T_SHOW_A <= '0';
    wait for 5 ns;
    T_CURRENT_TIME <= "1000";
    wait for 5 ns;
    T_CURRENT_TIME <= "1001";
    wait;
end process STIMULUS;
end TEST;

E.5. COUNT.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity COUNT is
port ( 
    INC : in std_logic;
    CLK : in std_logic;
    RESET : in std_logic;
    STORED_TIME : out std_logic_vector(3 downto 0)
);
end COUNT;

architecture RTL of COUNT is
signal STORED_TIME_INT : std_logic_vector(3 downto 0) := "0000";
begin
-- internal signal to hold the output signal
STORED_TIME <= STORED_TIME_INT;

-- process to increase or reset stored time
PULSE: process(CLK, RESET)
begin
if CLK'event and CLK = '1' then
if RESET = '1' then
STORED_TIME_INT <= "0000";
elsif INC = '1' then
STORED_TIME_INT <= STORED_TIME_INT + 1;
end if;
end if;
end if;
end process;
end RTL;

E.6. T_COUNT.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity T_COUNT is
end T_COUNT;

architecture TEST of T_COUNT is
component COUNT
port (""
STORED_TIME : out std_logic_vector (3 downto 0);
INC : in std_logic;
CLK : in std_logic;
RESET : in std_logic"
);
end component;

signal T_CLK : std_logic := '0';
signal T_INC : std_logic;
signal T_STORED_TIME : std_logic_vector (3 downto 0);
signal T_RESET : std_logic;

constant PERIOD : time := 20 ns;

begin
-- instantiate UUT
uut: COUNT port map(
STORED_TIME => T_STORED_TIME,
CLK => T_CLK,
INC => T_INC,
RESET => T_RESET
);


T_CLK <= not (T_CLK) after PERIOD/2;

-- Stimulus:

STIMULUS : process
begin
    T_INC <= '1';
    T_RESET <= '0';
    wait for period;

    T_INC <= '0';
    wait for period;

    T_INC <= '1';
    wait for period;

    T_INC <= '0';
    wait for period;

    T_RESET <= '1';
    wait for period;

    T_RESET <= '0';
    wait for period;

    T_INC <= '1';
    wait;
end process STIMULUS;
end TEST;

E.7 DDRV4.vhd

Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity DDRV4 is
port (  ALARM_TIME_MS_HR : in std_logic_vector (3 downto 0);
        ALARM_TIME_LS_HR : in std_logic_vector (3 downto 0);
        ALARM_TIME_MS_MIN : in std_logic_vector (3 downto 0);
        ALARM_TIME_LS_MIN : in std_logic_vector (3 downto 0);
        CURRENT_TIME_MS_HR : in std_logic_vector (3 downto 0);
        CURRENT_TIME_LS_HR : in std_logic_vector (3 downto 0);
        CURRENT_TIME_MS_MIN : in std_logic_vector (3 downto 0);
        CURRENT_TIME_LS_MIN : in std_logic_vector (3 downto 0);
        DISPLAY_MS_HR : out std_logic_vector (6 downto 0);
        DISPLAY_LS_HR : out std_logic_vector (6 downto 0);
);
architecture RTL of DDRV4 is

signal sound_alarm_int : std_logic_vector(3 downto 0);

component DDRV
  port(
    ALARM_TIME : in std_logic_vector(3 downto 0);
    CURRENT_TIME : in std_logic_vector(3 downto 0);
    DISPLAY : out std_logic_vector(6 downto 0);
    SHOW_A : in std_logic;
    SOUND_ALARM : out std_logic
  );
end component;

begin
  -- Hour most significant digit
  DDRV_MS_HR:DDRV port map (  
    ALARM_TIME => ALARM_TIME_MS_HR,
    CURRENT_TIME => CURRENT_TIME_MS_HR,
    DISPLAY => DISPLAY_MS_HR,
    SHOW_A => SHOW_A,
    SOUND_ALARM => sound_alarm_int(3));

  -- Hour less significant digit
  DDRV_LS_HR:DDRV port map (  
    ALARM_TIME => ALARM_TIME_LS_HR,
    CURRENT_TIME => CURRENT_TIME_LS_HR,
    DISPLAY => DISPLAY_LS_HR,
    SHOW_A => SHOW_A,
    SOUND_ALARM => sound_alarm_int(2));

  -- Minutes MSB
  DDRV_MS_MIN:DDRV port map (  
    ALARM_TIME => ALARM_TIME_MS_MIN,
    CURRENT_TIME => CURRENT_TIME_MS_MIN,
    DISPLAY => DISPLAY_MS_MIN,
    SHOW_A => SHOW_A,
    SOUND_ALARM => sound_alarm_int(1));

  -- Minutes LSB
  DDRV_LS_MIN:DDRV port map (  
    ALARM_TIME => ALARM_TIME_LS_MIN,
    CURRENT_TIME => CURRENT_TIME_LS_MIN,
    DISPLAY => DISPLAY_LS_MIN,
    SHOW_A => SHOW_A,
    SOUND_ALARM => sound_alarm_int(0));

  -- Alarm is set when all 4 digits of the clock are equal to that of the alarm
  SOUND_ALARM <= '1' when sound_alarm_int = "1111" else '0';

end RTL;
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity T_DDRV4 is
end T_DDRV4;

architecture TEST of T_DDRV4 is

component DDRV4
port {
  ALARM_TIME_MS_HR : in std_logic_vector (3 downto 0);
  ALARM_TIME_LS_HR : in std_logic_vector (3 downto 0);
  ALARM_TIME_MS_MIN : in std_logic_vector (3 downto 0);
  ALARM_TIME_LS_MIN : in std_logic_vector (3 downto 0);
  CURRENT_TIME_MS_HR : in std_logic_vector (3 downto 0);
  CURRENT_TIME_LS_HR : in std_logic_vector (3 downto 0);
  CURRENT_TIME_MS_MIN : in std_logic_vector (3 downto 0);
  CURRENT_TIME_LS_MIN : in std_logic_vector (3 downto 0);
  DISPLAY_MS_HR : out std_logic_vector (6 downto 0);
  DISPLAY_LS_HR : out std_logic_vector (6 downto 0);
  DISPLAY_MS_MIN : out std_logic_vector (6 downto 0);
  DISPLAY_LS_MIN : out std_logic_vector (6 downto 0);
  SHOW_A : in std_logic;
  SOUND_ALARM : out std_logic
};
end component;

-- declare test bench signals
signal T_CURRENT_TIME_LS_MIN : std_logic_vector (3 downto 0);
signal T_CURRENT_TIME_MS_MIN : std_logic_vector (3 downto 0);
signal T_CURRENT_TIME_LS_HR : std_logic_vector (3 downto 0);
signal T_CURRENT_TIME_MS_HR : std_logic_vector (3 downto 0);
signal T_ALARM_TIME_LS_MIN : std_logic_vector (3 downto 0);
signal T_ALARM_TIME_MS_MIN : std_logic_vector (3 downto 0);
signal T_ALARM_TIME_LS_HR : std_logic_vector (3 downto 0);
signal T_ALARM_TIME_MS_HR : std_logic_vector (3 downto 0);
signal T_SHOW_A : std_logic;
signal T_SOUND_ALARM : std_logic;
signal T_DISPLAY_MS_HR : std_logic_vector (6 downto 0);
signal T_DISPLAY_LS_HR : std_logic_vector (6 downto 0);
signal T_DISPLAY_MS_MIN : std_logic_vector (6 downto 0);
signal T_DISPLAY_LS_MIN : std_logic_vector (6 downto 0);

begin

uut : DDRV4 port map {
  ALARM_TIME_MS_HR => T_ALARM_TIME_MS_HR,
  ALARM_TIME_LS_HR => T_ALARM_TIME_LS_HR,
  ALARM_TIME_MS_MIN => T_ALARM_TIME_MS_MIN,
  ALARM_TIME_LS_MIN => T_ALARM_TIME_LS_MIN,
  CURRENT_TIME_MS_HR => T_CURRENT_TIME_MS_HR,
  CURRENT_TIME_LS_HR => T_CURRENT_TIME_LS_HR,
};
CURRENT_TIME_MS_MIN => T_CURRENT_TIME_MS_MIN,
CURRENT_TIME_LS_MIN => T_CURRENT_TIME_LS_MIN,
DISPLAY_MS_HR => T_DISPLAY_MS_HR,
DISPLAY_LS_HR => T_DISPLAY_LS_HR,
DISPLAY_MS_MIN => T_DISPLAY_MS_MIN,
DISPLAY_LS_MIN => T_DISPLAY_LS_MIN,
SHOW_A => T_SHOW_A,
SOUND_ALARM => T_SOUND_ALARM);

-- Process to generate stimulus ...

STIMULUS: process
begin

-- alarm time = 07:00
T_ALARM_TIME_MS_HR <= "0000";
T_ALARM_TIME_LS_HR <= "0111";
T_ALARM_TIME_MS_MIN <= "0000";
T_ALARM_TIME_LS_MIN <= "0000";

-- current time = 06:59
T_CURRENT_TIME_MS_HR <= "0000";
T_CURRENT_TIME_LS_HR <= "0110";
T_CURRENT_TIME_MS_MIN <= "0101";
T_CURRENT_TIME_LS_MIN <= "1001";
T_SHOW_A <= '0';
wait for 10 ns;

-- current time = 07:00
T_CURRENT_TIME_MS_HR <= "0000";
T_CURRENT_TIME_LS_HR <= "0111";
T_CURRENT_TIME_MS_MIN <= "0000";
T_CURRENT_TIME_LS_MIN <= "0000";
wait for 10 ns;

-- current time = 07:01
T_CURRENT_TIME_MS_HR <= "0000";
T_CURRENT_TIME_LS_HR <= "0111";
T_CURRENT_TIME_MS_MIN <= "0000";
T_CURRENT_TIME_LS_MIN <= "0000";
wait for 10 ns;

-- current time = 07:02, show the alarm
T_CURRENT_TIME_MS_HR <= "0000";
T_CURRENT_TIME_LS_HR <= "0111";
T_CURRENT_TIME_MS_MIN <= "0000";
T_CURRENT_TIME_LS_MIN <= "0010";
T_SHOW_A <= '1';
wait for 10 ns;

-- current time = 07:02, show CURRENT_TIME
T_CURRENT_TIME_MS_HR <= "0000";
T_CURRENT_TIME_LS_HR <= "0111";
T_CURRENT_TIME_MS_MIN <= "0000";
T_CURRENT_TIME_LS_MIN <= "0010";
T_SHOW_A <= '0';
wait;
end process STIMULUS;
end TEST;

E.9 COUNT4.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity COUNT4 is
port(
  HOUR :in std_logic;
  MIN :in std_logic;
  CLK :in std_logic;
  RESET :in std_logic;
  MS_HR :out std_logic_vector(3 downto 0);
  LS_HR :out std_logic_vector(3 downto 0);
  MS_MIN :out std_logic_vector(3 downto 0);
  LS_MIN :out std_logic_vector(3 downto 0)
);
end COUNT4;

ARCHITECTURE RTL of COUNT4 is
signal MS_HR_INC, LS_HR_INC, MS_MIN_INC, LS_MIN_INC :std_logic;
signal MS_HR_RESET, LS_HR_RESET, MS_MIN_RESET, LS_MIN_RESET :std_logic;
signal MS_HR_ST, LS_HR_ST, MS_MIN_ST, LS_MIN_ST :std_logic_vector(3 downto 0);
signal HOUR_INT :std_logic := '0';

component COUNT
port ( 
  INC : in std_logic;
  CLK : in std_logic;
  RESET : in std_logic;
  STORED_TIME : out std_logic_vector(3 downto 0) );
end component;

begin
COUNT_LS_MIN:COUNT port map(
  INC => LS_MIN_INC,
  CLK => CLK,
  RESET => LS_MIN_RESET,
  STORED_TIME => LS_MIN_ST);
COUNT_MS_MIN:COUNT port map(
  INC => MS_MIN_INC,
  CLK => CLK,
  RESET => MS_MIN_RESET,
STORED_TIME => MS_MIN_ST);

COUNT_LS_HR:COUNT port map(
  INC => LS_HR_INC,
  CLK => CLK,
  RESET => LS_HR_RESET,
  STORED_TIME => LS_HR_ST);

COUNT_MS_HR:COUNT port map(
  INC => MS_HR_INC,
  CLK => CLK,
  RESET => MS_HR_RESET,
  STORED_TIME => MS_HR_ST);

MS_HR <= MS_HR_ST;
LS_HR <= LS_HR_ST;
MS_MIN <= MS_MIN_ST;
LS_MIN <= LS_MIN_ST;

--INCREASE internal HOUR signal
-- XX:59 + 1 minute - shows that an hour increase should happen
HOUR_INT <= '1' when (LS_MIN_ST = "1001" AND MS_MIN_ST = "0101" AND MIN = '1')
  else '0';

--INCREASE statements
LS_MIN_INC <= '1' when ( MIN = '1' ) --increase minutes
  else '0';

MS_MIN_INC <= '1' when ( LS_MIN_ST = "1001" AND MIN = '1' ) --XX:X9 increase MS min when 9 minutes + 1 minute
  else '0';

LS_HR_INC <= '1' when ( HOUR_INT = '1' ) --XX:59 - increase hour
  or ( HOUR = '1' )
  else '0';

MS_HR_INC <= '1' when ( LS_HR_ST = "1001" AND HOUR_INT = '1' ) --X9:59 - increase MS hour when hour is 9 + hour increase
  or ( LS_HR_ST = "1001" AND HOUR = '1' ) --X9:XX
  else '0';

--ROLLOVER statements
LS_MIN_RESET <= '1' when ( RESET = '1' )
  or ( LS_MIN_ST = "1001" AND MIN = '1' ) --rollover when XX:X9 + minute increase
  else '0';

MS_MIN_RESET <= '1' when ( RESET = '1' )
  or ( LS_MIN_ST = "1001" AND MS_MIN_ST = "0101" AND MIN = '1' ) --rollover when XX:59 + minute increase
  else '0';

LS_HR_RESET <= '1' when ( RESET = '1' )
  or ( MS_HR_ST = "0010" AND LS_HR_ST = "0011" AND HOUR_INT = '1' ) --23:59 + minute increase
  else '0';
or (LS_HR_ST = "1001" AND HOUR_INT = '1') -- X9:59 + minute increase
or (MS_HR_ST = "0010" AND LS_HR_ST = "0011" AND HOUR = '1') -- 23:XX + hour increase
else '0';

MS_HR_RESET <= '1' when (RESET = '1')
        or (MS_HR_ST = "0010" AND LS_HR_ST = "0011" AND HOUR_INT = '1') -- 23:59 + minute
        increase
        or (MS_HR_ST = "0010" AND LS_HR_ST = "0011" AND HOUR = '1') -- 23:XX + hour increase
        else '0';
end RTL;

E.10 T_COUNT4.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity T_COUNT4 is
end T_COUNT4;

architecture TEST of T_COUNT4 is

component COUNT4
port(
    HOUR : in std_logic;
    MIN : in std_logic;
    CLK : in std_logic;
    RESET : in std_logic;
    MS_HR : out std_logic_vector(3 downto 0);
    LS_HR : out std_logic_vector(3 downto 0);
    MS_MIN : out std_logic_vector(3 downto 0);
    LS_MIN : out std_logic_vector(3 downto 0)
);
end component;

signal T_HOUR : std_logic;
signal T_MIN : std_logic;
signal T_CLK : std_logic := '0';
signal T_RESET : std_logic;
signal T_MS_HR : std_logic_vector(3 downto 0);
signal T_LS_HR : std_logic_vector(3 downto 0);
signal T_MS_MIN : std_logic_vector(3 downto 0);
signal T_LS_MIN : std_logic_vector(3 downto 0);

constant PERIOD : time := 20 ns;

begin
-- instantiate DUT

uut: COUNT4 port map(
    HOUR => T_HOUR,
    MIN => T_MIN,
    CLK => T_CLK,
    RESET => T_RESET,
    MS_HR => T_MS_HR,
    LS_HR => T_LS_HR,
    MS_MIN => T_MS_MIN,
    LS_MIN => T_LS_MIN
);

-- infinite clock generator

T_CLK <= not T_CLK after PERIOD/2;

STIMULUS : process
begin
    T_MIN <= '0';
    T_HOUR <= '0';

    -- check initial values
    wait for period*2;

    -- pulse reset
    T_RESET <= '1';
    wait for period*2;
    T_RESET <= '0';
    wait for period*2;

    -- increase the HOUR
    T_HOUR <= '1';
    wait for period*10;
    T_HOUR <= '0';

    -- reset
    T_RESET <= '1';
    wait for period*2;
    T_RESET <= '0';
    wait for period*2;

    -- T_MIN stays '1', testing 100 periods
    T_MIN <= '1';
    wait for period*100;
    wait;
end process STIMULUS;

end TEST;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ALARM is
port(
    CLK : in std_logic;
    RESET : in std_logic;
    TIME_HOUR : in std_logic;
    TIME_MIN : in std_logic;
    ALARM_HOUR : in std_logic;
    ALARM_MIN : in std_logic;
    SHOW_A : in std_logic;
    SILENT : in std_logic;
    SOUND_ALARM : out std_logic;
    DISPLAY_LS_MIN : out std_logic_vector(6 downto 0);
    DISPLAY_MS_MIN : out std_logic_vector(6 downto 0);
    DISPLAY_LS_HR : out std_logic_vector(6 downto 0);
    DISPLAY_MS_HR : out std_logic_vector(6 downto 0)
); end ALARM;

architecture STRUCT of ALARM is
signal TIME_MS_HR, TIME_LS_HR, TIME_MS_MIN, TIME_LS_MIN : std_logic_vector(3 downto 0);
signal ALARM_MS_HR, ALARM_LS_HR, ALARM_MS_MIN, ALARM_LS_MIN : std_logic_vector(3 downto 0);
signal SOUND_ALARM_INT : std_logic;
component SWITCH
port ( 
    ALARM_IN : in std_logic;
    SILENT : in std_logic;
    ALARM_OUT : out std_logic
); end component;

component DDRV4
port ( 
    ALARM_TIME_MS_HR : in std_logic_vector (3 downto 0);
    ALARM_TIME_LS_HR : in std_logic_vector (3 downto 0);
    ALARM_TIME_MS_MIN : in std_logic_vector (3 downto 0);
    ALARM_TIME_LS_MIN : in std_logic_vector (3 downto 0);
    CURRENT_TIME_MS_HR : in std_logic_vector (3 downto 0);
    CURRENT_TIME_LS_HR : in std_logic_vector (3 downto 0);
    CURRENT_TIME_MS_MIN : in std_logic_vector (3 downto 0);
    CURRENT_TIME_LS_MIN : in std_logic_vector (3 downto 0);
    DISPLAY_MS_HR : out std_logic_vector (6 downto 0);
    DISPLAY_LS_HR : out std_logic_vector (6 downto 0);
    DISPLAY_MS_MIN : out std_logic_vector (6 downto 0);
    DISPLAY_LS_MIN : out std_logic_vector (6 downto 0);
    SHOW_A : in std_logic;
    SOUND_ALARM : out std_logic
); end component;
component COUNT4
port (  
HOUR :in std_logic;
MIN :in std_logic;
CLK :in std_logic;
RESET :in std_logic;
MS_HR :out std_logic_vector(3 downto 0);
LS_HR :out std_logic_vector(3 downto 0);
MS_MIN :out std_logic_vector(3 downto 0);
LS_MIN :out std_logic_vector(3 downto 0)
);  
end component;

begin
ALARM_COUNTER:COUNT4 port map(
    HOUR => ALARM_HOUR,
    MIN => ALARM_MIN,
    CLK => CLK,
    RESET => RESET,
    MS_HR => ALARM_MS_HR,
    LS_HR => ALARM_LS_HR,
    MS_MIN => ALARM_MS_MIN,
    LS_MIN => ALARM_LS_MIN
);

TIME_COUNTER:COUNT4 port map(
    HOUR => TIME_HOUR,
    MIN => TIME_MIN,
    CLK => CLK,
    RESET => RESET,
    MS_HR => TIME_MS_HR,
    LS_HR => TIME_LS_HR,
    MS_MIN => TIME_MS_MIN,
    LS_MIN => TIME_LS_MIN
);

DISPLAY_MUX:DDRV4 port map(
    ALARM_TIME_MS_HR => ALARM_MS_HR,
    ALARM_TIME_LS_HR => ALARM_LS_HR,
    ALARM_TIME_MS_MIN => ALARM_MS_MIN,
    ALARM_TIME_LS_MIN => ALARM_LS_MIN,
    CURRENT_TIME_MS_HR => TIME_MS_HR,
    CURRENT_TIME_LS_HR => TIME_LS_HR,
    CURRENT_TIME_MS_MIN => TIME_MS_MIN,
    CURRENT_TIME_LS_MIN => TIME_LS_MIN,
    DISPLAY_MS_HR => DISPLAY_MS_HR,
    DISPLAY_LS_HR => DISPLAY_LS_HR,
    DISPLAY_MS_MIN => DISPLAY_MS_MIN,
    DISPLAY_LS_MIN => DISPLAY_LS_MIN,
    SHOW_A => SHOW_A,
    SOUND_ALARM => SOUND_ALARM_INT
);

ALARM_SWITCH:SWITCH port map (
end STRUCT;

E.12 T_ALARM.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--This testbench uses a function called to_int which converts the seven segment format outputs to integer format.
--This allows the output from device under test to be displayed in human readable form
--The outputs from your alarm clock (DISPLAY_LS_MIN, DISPLAY_MS_MIN....) are assigned to internal integer variables
--via the to_int function.

entity T_ALARM is
end T_ALARM;

architecture TEST of T_ALARM is

component ALARM
port(
    CLK : in std_logic;
    RESET : in std_logic;
    TIME_HOUR : in std_logic;
    TIME_MIN : in std_logic;
    ALARM_HOUR : in std_logic;
    ALARM_MIN : in std_logic;
    SHOW_A : in std_logic;
    SILENT : in std_logic;
    SOUND_ALARM : out std_logic;
    DISPLAY_LS_MIN : out std_logic_vector(6 downto 0);
    DISPLAY_MS_MIN : out std_logic_vector(6 downto 0);
    DISPLAY_LS_HR : out std_logic_vector(6 downto 0);
    DISPLAY_MS_HR : out std_logic_vector(6 downto 0)
);
end component;

-- signals
signal T_CLK : std_logic := '0';
signal T_RESET : std_logic;
signal T_TIME_HOUR : std_logic;
signal T_TIME_MIN : std_logic;
signal T_ALARM_HOUR : std_logic;
signal T_ALARM_MIN : std_logic;
signal T_SHOW_A : std_logic;
signal T_SILENT : std_logic;
signal T_SOUND_ALARM : std_logic;
```vhdl
-- constants for the segments to light for each digit 0 to 9 on a 7 segment display
-- 0 = LED segment ON, 1 = LED segment OFF
constant ZERO_SEG  : std_logic_vector(6 downto 0) := "0100000";
constant ONE_SEG   : std_logic_vector(6 downto 0) := "1111001";
constant TWO_SEG   : std_logic_vector(6 downto 0) := "1000100";
constant THREE_SEG : std_logic_vector(6 downto 0) := "1010000";
constant FOUR_SEG  : std_logic_vector(6 downto 0) := "0011000";
constant FIVE_SEG  : std_logic_vector(6 downto 0) := "0010010";
constant SIX_SEG   : std_logic_vector(6 downto 0) := "0000010";
constant SEVEN_SEG : std_logic_vector(6 downto 0) := "1111000";
constant EIGHT_SEG : std_logic_vector(6 downto 0) := "0000000";
constant NINE_SEG  : std_logic_vector(6 downto 0) := "0011000";

-- convert 7 seg outputs to integers for clarity
function to_int ( vec_in : std_logic_vector(6 downto 0) ) return integer is
  variable result : integer range 0 to 9;
  begin
    case vec_in is
      when ZERO_SEG => RESULT := 0; -- 0
      when ONE_SEG  => RESULT := 1; -- 1
      when TWO_SEG  => RESULT := 2; -- 2
      when THREE_SEG => RESULT := 3; -- 3
      when FOUR_SEG => RESULT := 4; -- 4
      when FIVE_SEG => RESULT := 5; -- 5
      when SIX_SEG  => RESULT := 6; -- 6
      when SEVEN_SEG => RESULT := 7; -- 7
      when EIGHT_SEG => RESULT := 8; -- 8
      when NINE_SEG => RESULT := 9; -- 9
      when others   => RESULT := 0; -- X
    end case;
    return result;
  end;

-- integer signals
signal INT_LS_MIN, INT_MS_MIN,
    INT_LS_HR, INT_MS_HR : integer range 0 to 9;

begin
  -- assign ints via function for translation of 7seg values
  INT_LS_MIN <= to_int(T_DISPLAY_LS_MIN);
```

INT_MS_MIN <= to_int( T_DISPLAY_MS_MIN );
INT_LS_HR <= to_int( T_DISPLAY_LS_HR );
INT_MS_HR <= to_int( T_DISPLAY_MS_HR );

-- instantiate UUT

uut : ALARM port map(
    CLK => T_CLK,
    RESET => T_RESET,
    TIME_HOUR => T_TIME_HOUR,
    TIME_MIN => T_TIME_MIN,
    ALARM_HOUR => T_ALARM_HOUR,
    ALARM_MIN => T_ALARM_MIN,
    SHOW_A => T_SHOW_A,
    SILENT => T_SILENT,
    SOUND_ALARM => T_SOUND_ALARM,
    DISPLAY_LS_MIN => T_DISPLAY_LS_MIN,
    DISPLAY_MS_MIN => T_DISPLAY_MS_MIN,
    DISPLAY_LS_HR => T_DISPLAY_LS_HR,
    DISPLAY_MS_HR => T_DISPLAY_MS_HR
);

-- infinite clock generator

T_CLK <= not T_CLK after PERIOD/2;

-- STIMULUS : process
begin

    -- initialize signals
    T_TIME_HOUR <= '0';
    T_TIME_MIN <= '0';
    T_SILENT <= '0';
    T_SHOW_A <= '0';
    T_RESET <= '0';

    -- set alarm to 07:00
    T_ALARM_HOUR <= '1';
    T_ALARM_MIN <= '0';
    wait for period*7;
    T_ALARM_HOUR <= '0';

    -- verify alarm time
    T_SHOW_A <= '1';
    wait for period*2;
    T_SHOW_A <= '0';

    -- start clock
    T_TIME_MIN <= '1';
    wait for period*200;
T_TIME_MIN <= '0';

-- check RESET, SOUND and SILENT
T_RESET <= '1';
wait for period*2;
T_RESET <= '0';
wait for period*2;
T_SILENT <= '1';
wait for period*2;
T_TIME_MIN <= '1';
wait for period*2;
T_SILENT <= '0';

-- set alarm to 07:00
T_ALARM_HOUR <= '1';
T_ALARM_MIN <= '0';
wait for period*7;
T_ALARM_HOUR <= '0';
wait;
end process STIMULUS;
end TEST;
References


