VHDL Language and Application

Course Version VHDL 9.0

Lab Manual **Revision 1.0**

Table of Contents

VHDL Language and Application

Module 1: About This Course

Lab 1-1 Introduction

Objective: To give an overview of the lab course.

The goal of this lab is to introduce you to the lab course and brief you on how this lab is structured.

Introduction

These lab exercises are designed to make you more familiar with the VHDL-based design methodology by using it in some typical design situations. They are organized in a way that takes you step-by-step through what to do first and then slowly removes the guidance as you become more familiar with the tools and language.

Tool Specific Information

This lab book and the VHDL lab exercises are completely tool-independent; they can be used with any combination of popular simulation and synthesis tools. Please refer to your vendor documentation for specific instructions for simulating or synthesizing your VHDL code

Overview of Labs

The first few labs are to familiarize you with the tools that you are using and the basic steps involved in simulating and synthesizing a small design.

Subsequent labs involve writing your own VHDL code. The labs have been designed to follow each other and progress in a logical manner as in a real design situation. The goal is not to complete all the exercises during the course but to learn from each exercise at your own pace.

One Common Project

To make the labs more interesting, they are all parts of a single design that you will build up over the course. The design is an Alarm Clock with a seven-segment display driver, counting mechanism, alarm generator, etc.

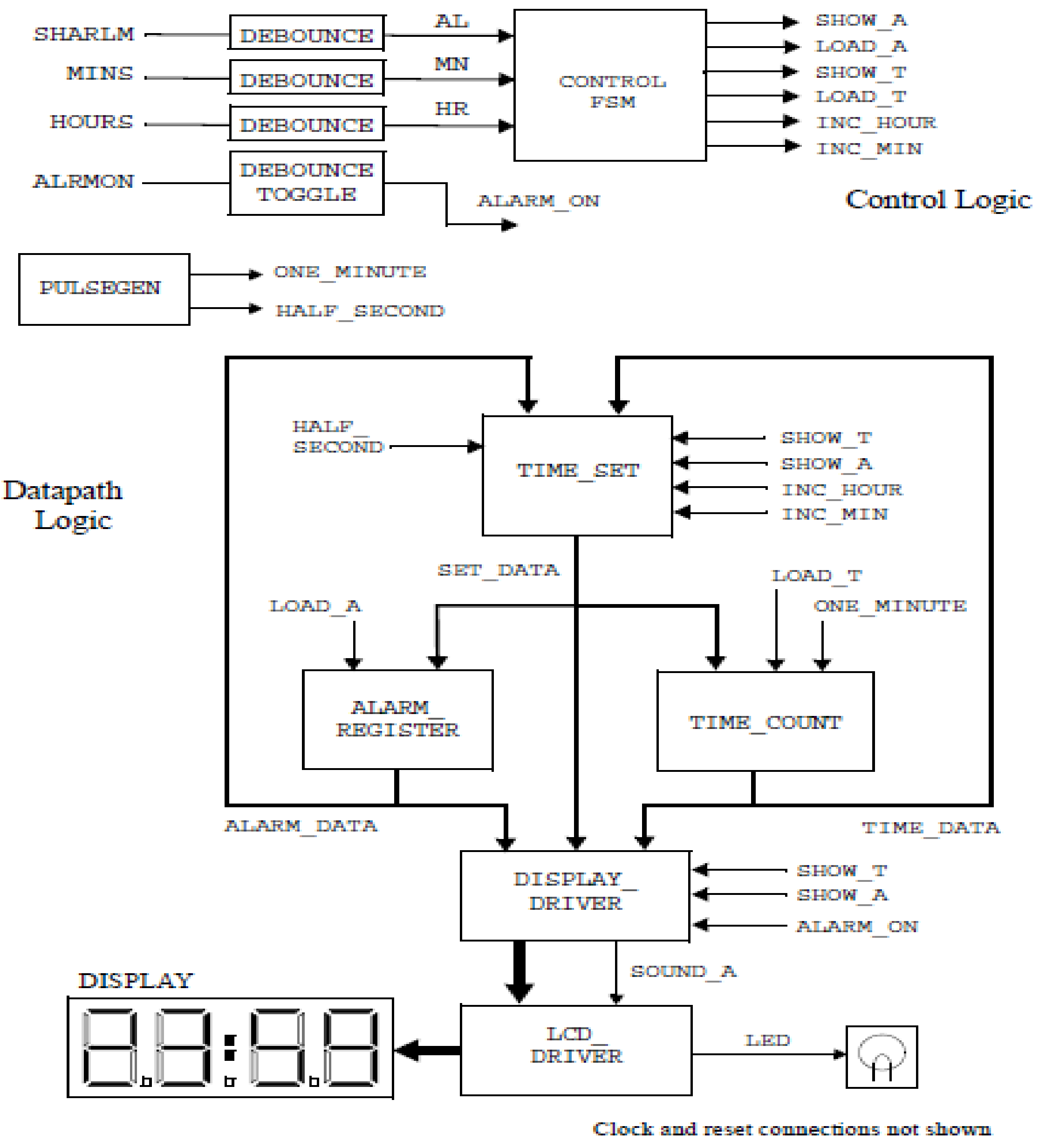
End of Lab

Lab 1-2 Alarm Clock Design

Objective: To become familiarized with the Alarm Clock design.

The goal of the lab exercise is to familiarize you with the Alarm Clock that we design throughout this course.

This is the block diagram of the final Alarm Clock design that we are going to implement throughout this lab course. During the lab exercises, you may find this diagram useful to determine where in the final circuit a particular component will be connected. You will also need to consult this diagram during the final lab exercises to show how the components should be connected.



End of Lab

Module 2: VHDL Application

There are no labs for this module

Module 3: Language Introduction

There are no labs for this module

Module 4: Signals and Data Types

There are no labs for this module

Module 5: VHDL Operators

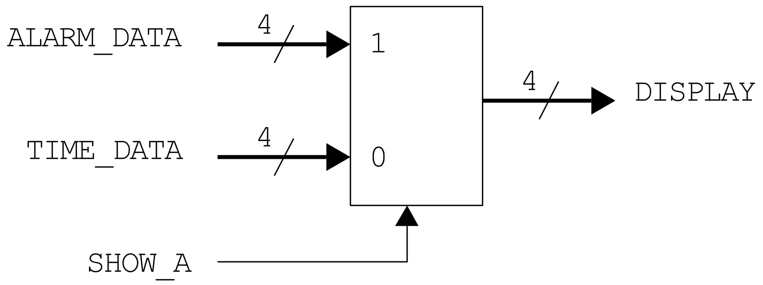
There are no labs for this module

Module 6: Sequential Statements

Lab 6-1 Familiarization with Multiplexer Design

Objective: To become familiarized with the Multiplexer model.

The goals of the lab exercise are to familiarize yourself with a simple Multiplexer model. We will use this design over the next few lab exercises to become familiar with the simulation and synthesis tools.



Specifications

- If `SHOW_A` is `'1'`, the input `ALARM_DATA` is assigned to the output `DISPLAY`.
- If `SHOW_A` is `'0'`, the input `TIME_DATA` is assigned to the output `DISPLAY`.
- `TIME_DATA`, `ALARM_DATA` and `DISPLAY` are all 4-bit `std_logic_vector` arrays.

Designing the MUX

1. Go to the path where lab directories are stored:

   `<download path>/vhdl/labs`

2. We will work on the file `mux.vhd`. Take a look at the file's contents in a text editor.

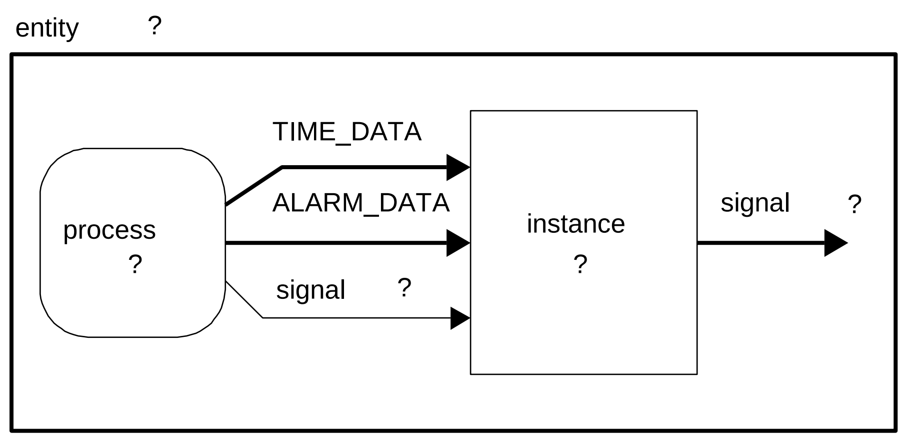
3. A testbench has been provided to simulate and verify the multiplexer design. The testbench contains an instance of the design `MUX`, and a VHDL process that generates some stimulus.

   a. View the contents of the testbench file `t_mux.vhd`.

   b. Check that the component declaration in the testbench and the entity have matching names and port lists.

Note: You are not expected to understand everything in the testbench at this point of the course.

The following diagram shows the interconnection of the objects in the test bench. By comparing the diagram with the VHDL code for the test bench, name each of the items shown as "?" below:



Note: Remember from the training material that VHDL needs a *configuration* to link an instance of a component with an entity and architecture. The Multiplexer testbench uses a *default configuration,* where the name and port list of the Multiplexer component declaration in the testbench matches the name and port list of the Multiplexer entity.

Answer: `TB_MUX` is the entity name. `STIMULUS` is the process that drives the MUX inputs. `DUT` is the name of the MUX instance. `SHOW_A` is the other input driven into the DUT, while `DISPLAY` is the output.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 6-2, where you will be compiling and simulating the Multiplexer design.

End of Lab

Lab 6-2 Familiarization with Your VHDL Simulator

Objective: To compile and simulate the model of the multiplexer.

The goal of this exercise is to become familiar with compiling, debugging, and simulating an HDL design with your simulator. We will use the multiplexer design described in Lab Exercise 6-1 as the VHDL model for this exercise. We use Cadence's Xcelium™ Simulator to simulate our design. However, you can use any tool of your choice.

Steps to Run the Simulation

1. Go to the path where lab directories are stored:

   ```
   <download path >/vhdl/labs
   ```

2. We will work on files `mux.vhd` and `t_mux.vhd`, which are the design and testbench files for the mux implementation as mentioned in the previous lab. The testbench has been provided to simulate and verify the multiplexer design. The testbench contains an instance of the design and a VHDL process that generates some stimulus.

   Note: There is both a syntax (compilation) error **and** functional (simulation) error with the design, which you will need to find and fix.

   a. Compile all the files that need to be compiled:

   ```
   xmvhdl -messages -v200x mux.vhd t_mux.vhd
   ```

   b. Elaborate the testbench entity `TB_MUX` from its compile library `WORKLIB`:

   ```
   xmelab WORKLIB.TB_MUX -access rwc -linedebug
   ```

   c. Simulate the design in GUI mode on the testbench entity from its compile library:

   ```
   xmsim WORKLIB.TB_MUX -gui
   ```

3. Better still, you can combine all three steps into one command using `xrun`:

   ```
   xrun mux.vhd t_mux.vhd -top TB_MUX -access rwc -v200x -gui -linedebug
   ```

   Where:

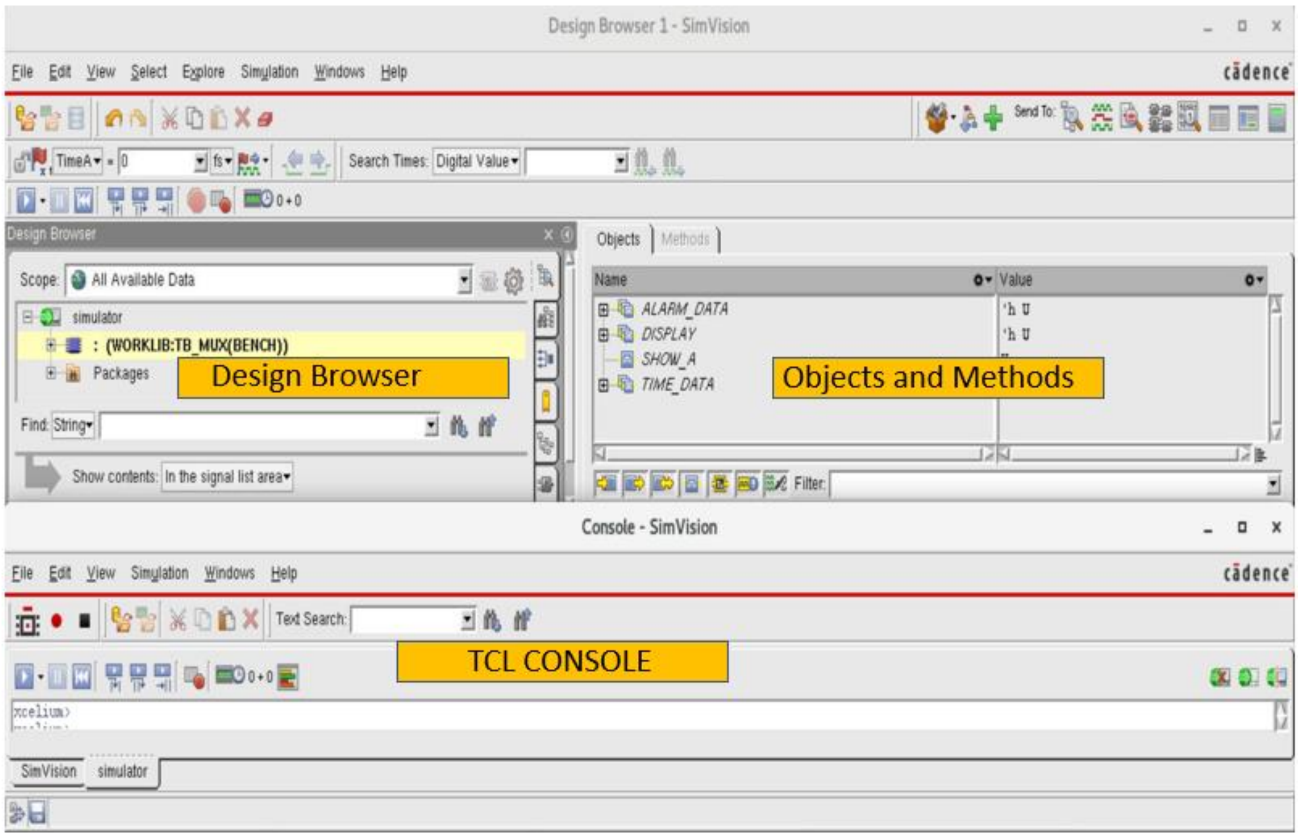
   - The testbench entity name is the argument to the `-top` switch.
   - `-v200x` is required to compile updated features such as `process all`, etc.
   - `-linedebug` enables the full features of source code browser.

4. The main front-end Graphical User Interface (GUI) will appear. It has three main areas: Design Browser, Console, and Objects window.

   a. The Design Browser lists the entire design and testbench hierarchy. We can expand and navigate to any of the components in the hierarchy.

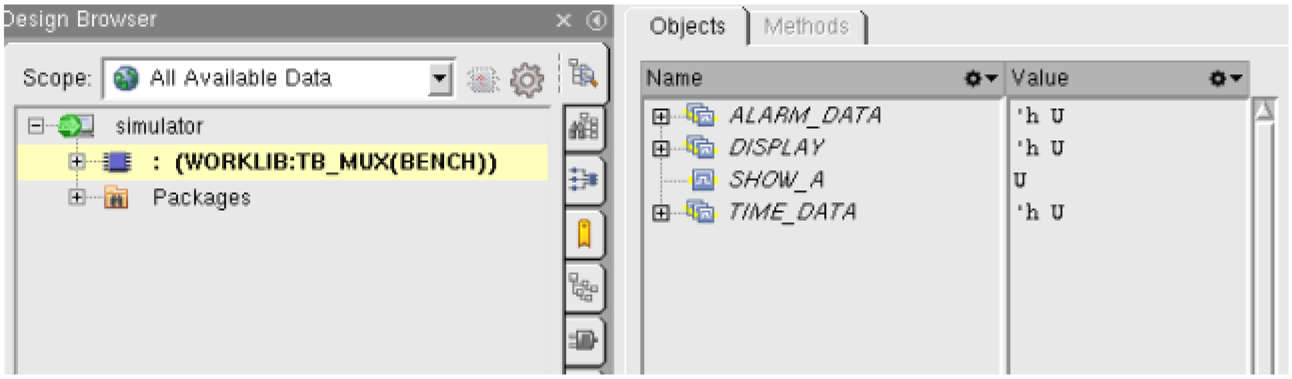
   b. The Objects window lists all signals of the component selected in the Design Browser.

   c. The Console allows us to enter the Tcl commands to control the simulation and to develop and execute control scripts. This is also where the Tcl shell commands and return values are echoed, and compiler error messages are displayed.



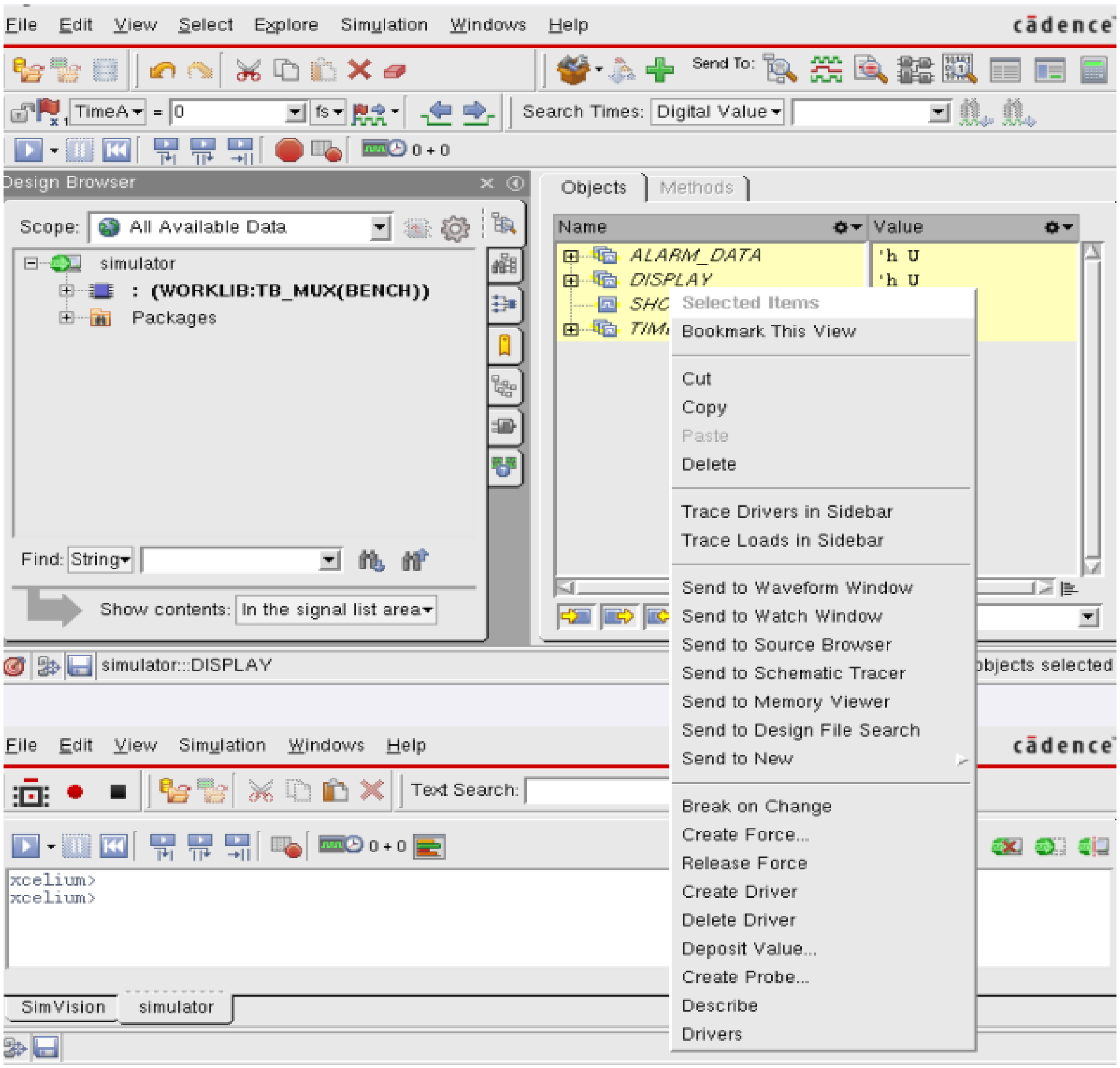
5. Run the simulation.

   a. Click on testbench file `WORKLIB:TB_MUX` in the Design Browser window. This shows all the signals in the file under the objects tab to the right.



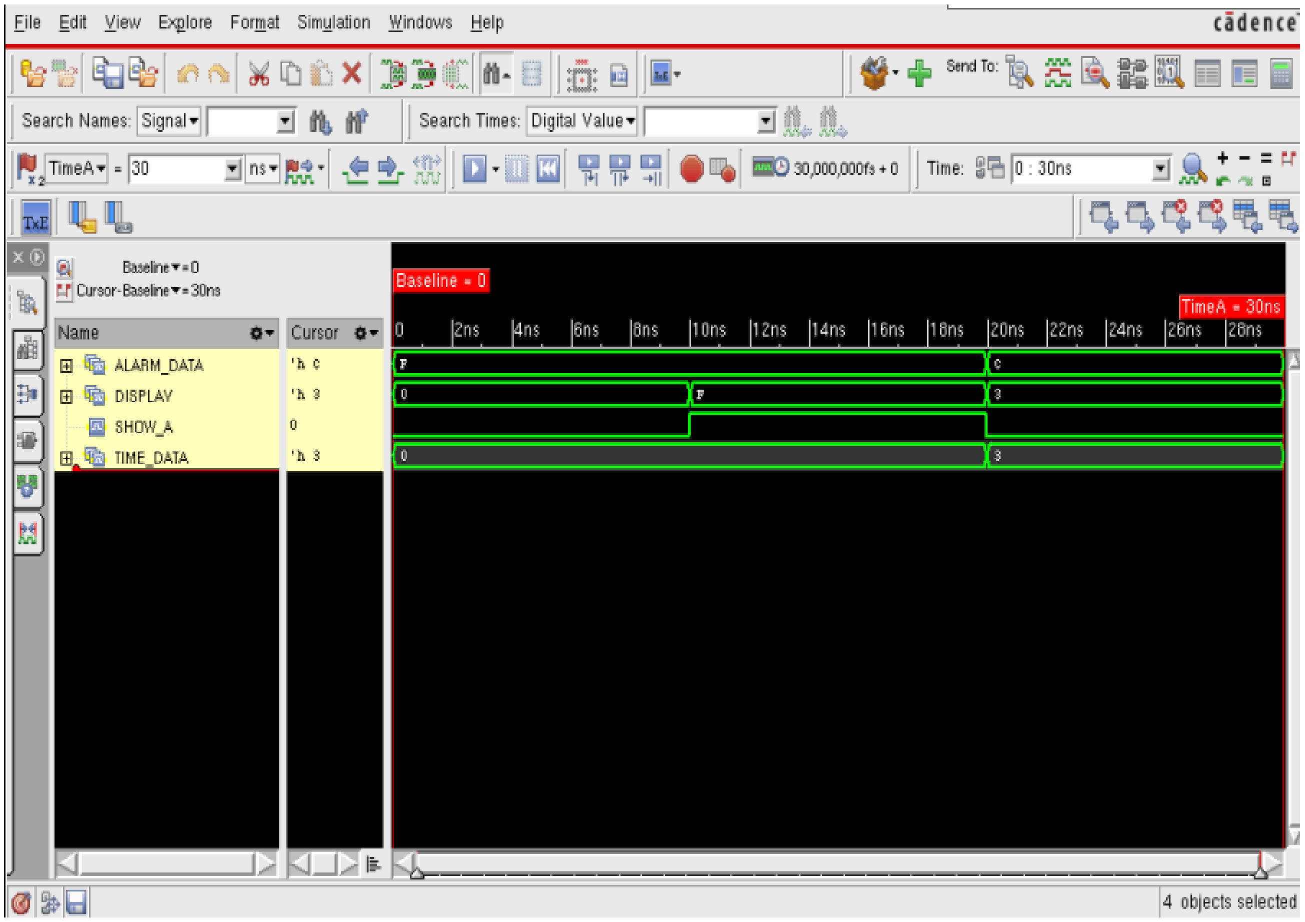
Name	Value
ALARM_DATA	'h U
DISPLAY	'h U
SHOW_A	U
TIME_DATA	'h U

b. Select the signals under the object tab, **right**-click on them and add the signals to the waveform window by selecting the **Send to Waveform Window** option from the drop-down menu.



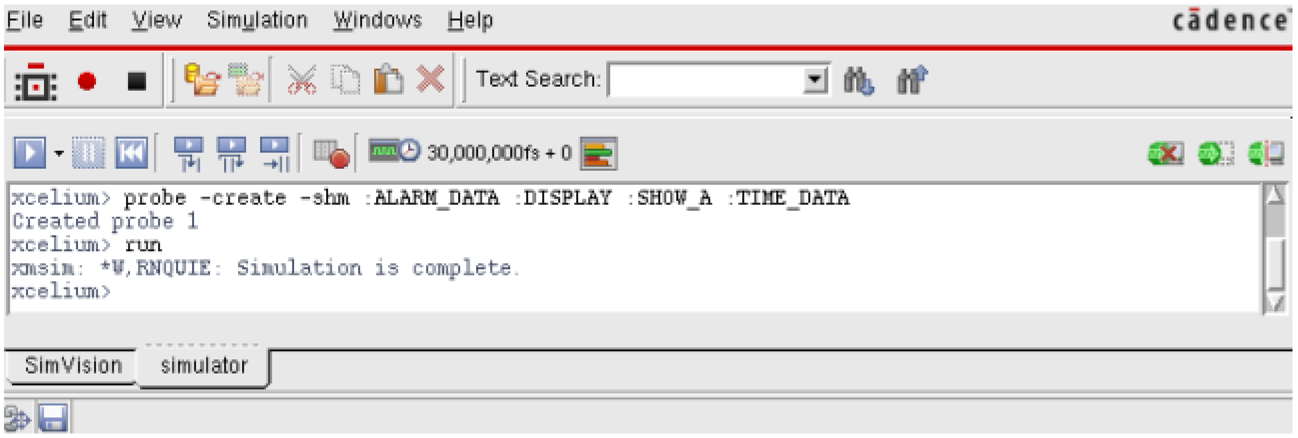
```
xcelium>
xcelium>
```

c. Run the simulation by clicking on the **Play icon** from the toolbar on the top of the waveform window.



d. Observe the waveform and analyze the results.

e. The simulation log can be found in the Console window, where you can type simulation commands at the prompt and observe the log.



```
xcelium> probe -create -shm :ALARM_DATA :DISPLAY :SHOW_A :TIME_DATA
Created probe 1
xcelium> run
xmsim: *W,RNQUIE: Simulation is complete.
xcelium>
```

On Completion of this Lab Exercise

Please move on to the next lab exercise, Lab 6-3, where you will be synthesizing the multiplexer design.

End of Lab

Lab 6-3 Familiarization with Your Synthesis Tool

Objective: To synthesize the multiplexer to produce a gate-level netlist.

The goals of the lab exercise are to synthesize the Multiplexer design and inspect the gate-level design produced. You can use any synthesis tool of your choice. Here, we use Cadence's Genus™ synthesizer to synthesize our design.

Preparing for Synthesis

Genus is an extremely powerful and sophisticated tool with many features and options, but the basic steps required to synthesize the HDL design are straightforward and easy to follow. It is very painful to execute these commands one by one at the command prompt. Hence, we list all these commands in a Tcl file and run this script at the command prompt.

The `genus_shell.tcl` script consists of the following list of Genus tool commands to synthesize the design:

1. Check the Genus tool is installed and that the required environment variable is set.
2. Set the `design` variable to `mux` (the top-level VHDL design entity name).
3. Select the target technology for synthesis.
4. Read in the HDL design.
5. Elaborate the design hierarchy.
6. Check for any unresolved references using `check_design` command.
7. Specify any constraints, for example, the clock constraints.
8. Synthesize the design to optimize the logic and map to the target technology.
9. Write the synthesized netlist in a format that can be used by layout tools.
10. Invoke the GUI to view the schematic.
11. Generate and save any reports required.

The following sections describe these steps in more detail using the commands from the `genus_shell.tcl` script.

Selecting the Target Technology

To set the technology for our synthesized design, we need to load the required target technology library.

Genus provides a tutorial library for demonstration purposes. We will use this library for this lab course. Alternatively, you could use any other technology library that you have in your installation.

```
set_db library $env(GENUSHOME)/share/synth/tutorials/tech/tutorial.lib
```

Reading the Input Design

1. The following command in the `genus_shell.tcl` script specifies the language and the list of files for our design:

   ```
   read_hdl -vhdl mux.vhd
   ```

   Where `–vhdl` specifies a VHDL design.

   Alternatives are:

   - `-v2001` for Verilog 2001 compliant designs
   - `-sv` for SystemVerilog
   - `-v1995` for Verilog 1995 compliant designs

   And `mux.vhd` is a space-separated list of HDL files in the required compilation order for the HDL you are using. As we have a single file, the script uses the `$design` variable.

   Any errors in this step will be displayed in the console.

Building the Design Hierarchy

Once you have read in the HDL design files, you need to convert the design into a technology and language-independent netlist and link the design hierarchy.

1. The following command in `genus_shell.tcl` elaborates the design hierarchy:

   ```
   elaborate MUX
   ```

   where `MUX` is the name of the top-level design unit. It is important to understand that this is the design unit name (the entity name for VHDL) and not the name of a file.

2. We then write out a *generic* netlist file:

   ```
   write_hdl > $design.vs
   ```

where $design.vs is the name of the generic design netlist file, using the `design` variable.

Specifying Clock Constraints

If you have a clock in your design, you need to specify the required clock speed. A purely combinational circuit like the MUX will have no clock, so although `genus_shell.tcl` contains clock constraint examples, these are commented out.

The following syntax defines a clock constraint:

```
define_clock -period <required_clock_period> -name <value>
```

Similarly, we can set input and output delays with respect to a reference clock `refclk`:

```
external_delay -input <delay_value> -clock refclk [all_inputs]
external_delay -output <delay_value> -clock refclk [all_outputs]
```

Synthesizing the Design

1. By default, synthesis is run with medium effort. The `genus_shell.tcl` script uses this default. The synthesis effort can be changed using this attribute:

   ```
   set_attribute syn_generic_effort low|medium|high|express
   ```

2. We can synthesize the RTL blocks to generic gates, using the given constraints, and perform RTL optimization:

   ```
   sync_generic
   ```

3. Optionally we can set the effort level to map and optimize to the technology library:

   ```
   set_attribute sync_map_effort medium
   Set_attribute sync_opt_effort medium
   ```

   Then we synthesize the design to technology gates and optimize:

   ```
   sync_map
   sync_opt
   ```

   These commands map the design to the cells described in the supplied technology library and perform optimization.

   When you synthesize the design, the tool shows you informational messages on synthesis progress. You can also view the log file in a separate window. For a real-world design, you typically examine these messages in detail.

4. We then write out the mapped and optimized netlist file:

   ```
   write_hdl > $design.vg
   ```

Generating Reports from the GUI

The Genus Graphical User Interface (GUI) is invoked as follows:

```
gui_show
```

The schematic pane in the GUI shows a mapped gate-level design of the HDL containing technology-specified library cells. You can examine the schematic for the synthesized design and produce reports on area, timing, etc.

To view the report on area, select **Report > Datapath > Area** on the GUI.

Invoking the Synthesis Tool

1. Go to the path where the lab directories are stored:
   ```
   <download path >/vhdl/labs
   ```

2. We will work with the following files:
   a. `mux.vhd`, which is the design file for the mux implementation.
   b. `genus_shell.tcl`, which is the Tcl script file of Genus tool commands.

3. Execute the synthesis as follows:
   ```
   genus -f genus_shell.tcl
   ```

Genus GUI

The script file command `gui_show` invokes the GUI, which contains these main features:

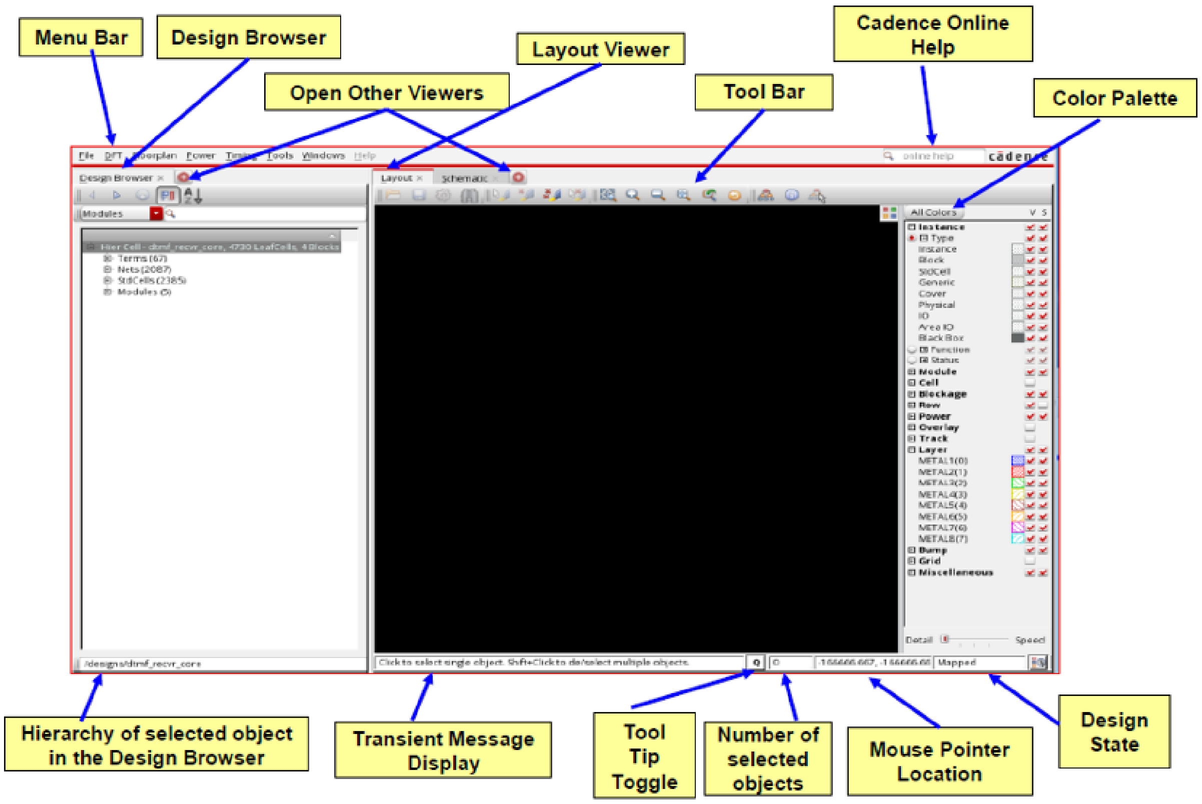
1. Menu bar

2. Viewers
   a. Design Browser
   b. Layout viewer
   c. Schematic viewer
   d. HDL viewer
   e. Object attributes

3. Layer Control

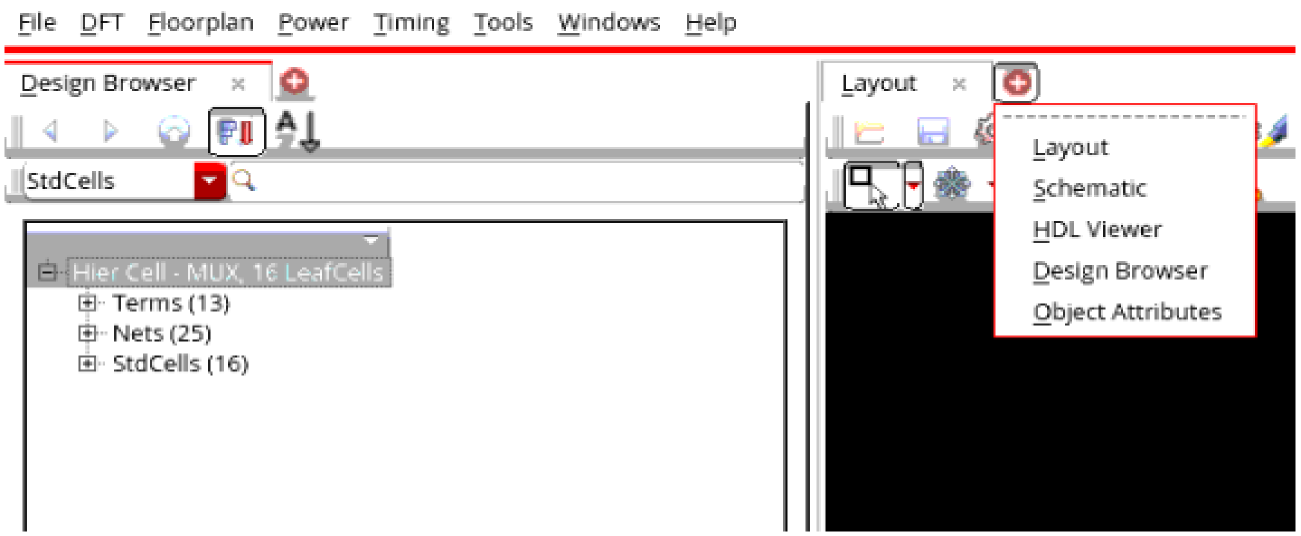
4. Toolbar

5. Status bar



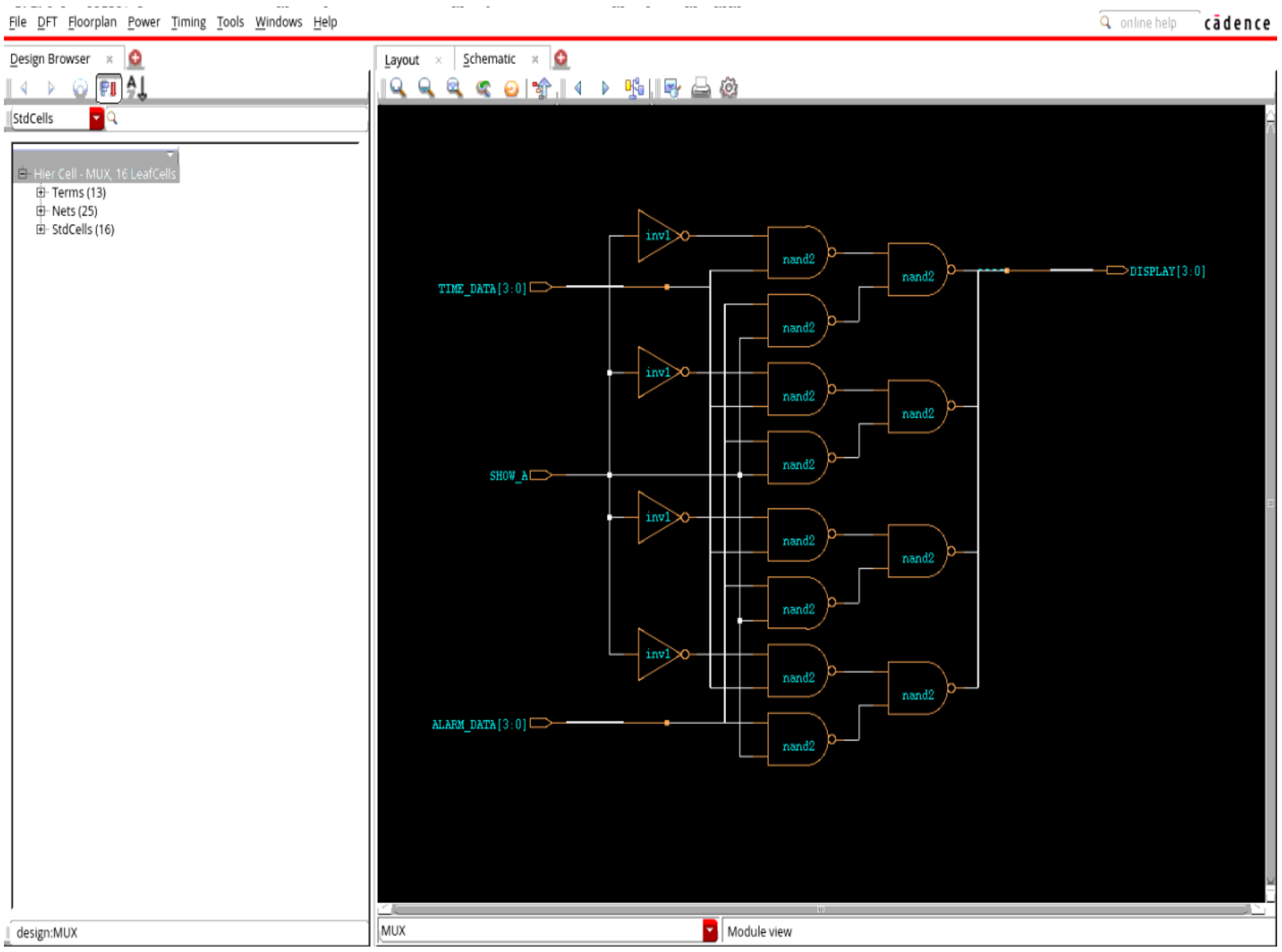
Exploring Different GUI Viewers

1. Click on the sign to open the schematic viewer.



2. View the schematic.

   This should match the intended hardware.

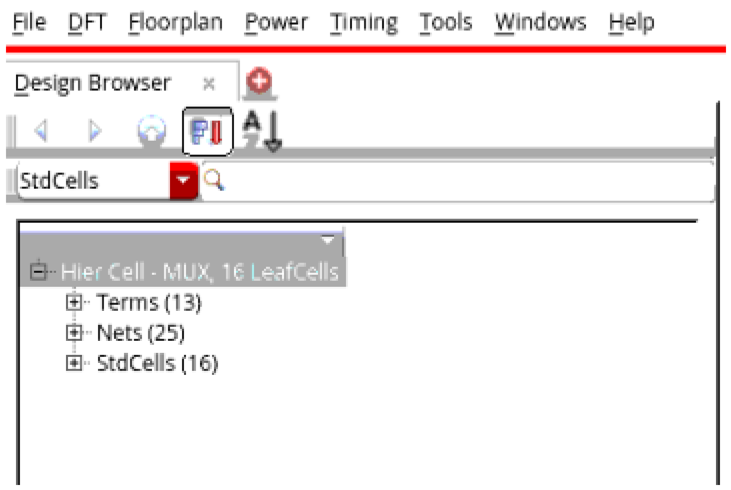


3. View the contents of the design browser.

   The design browser is categorized based on the object types:

   - Heir Cell
   - Modules
   - Terms
   - Nets
   - StdCells

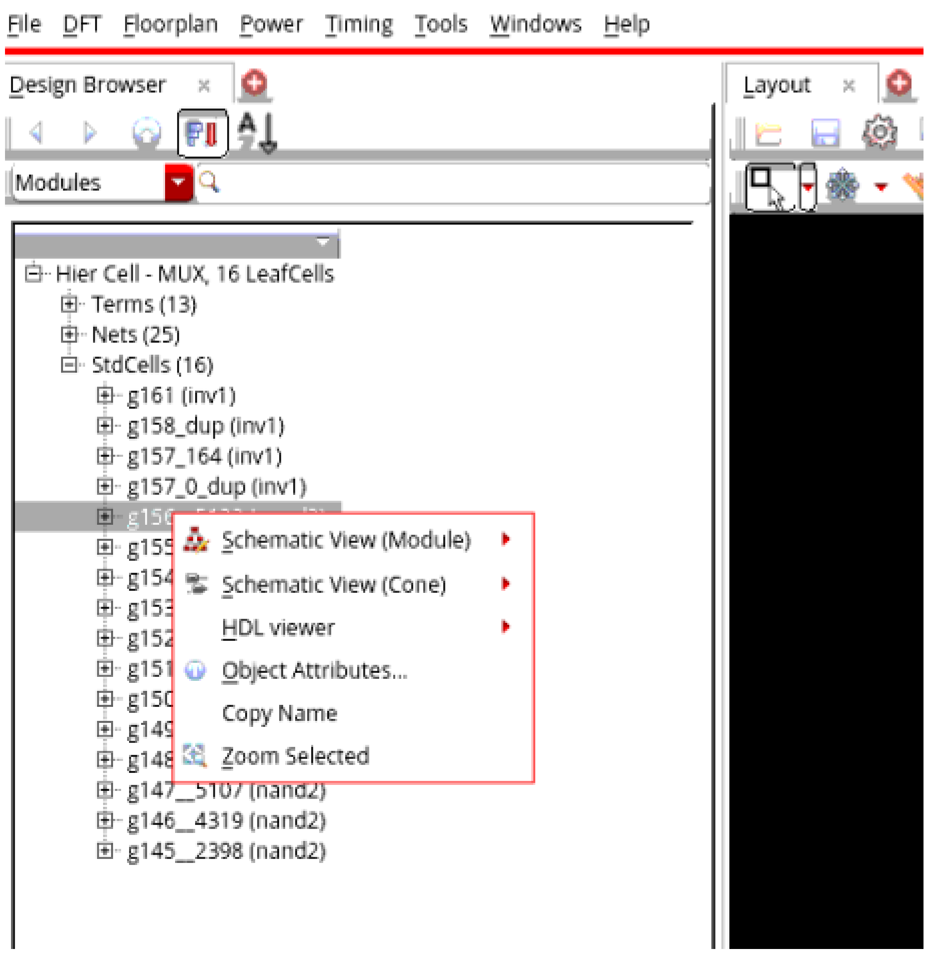
   a. Click the **+** sign next to different categories and explore the objects.

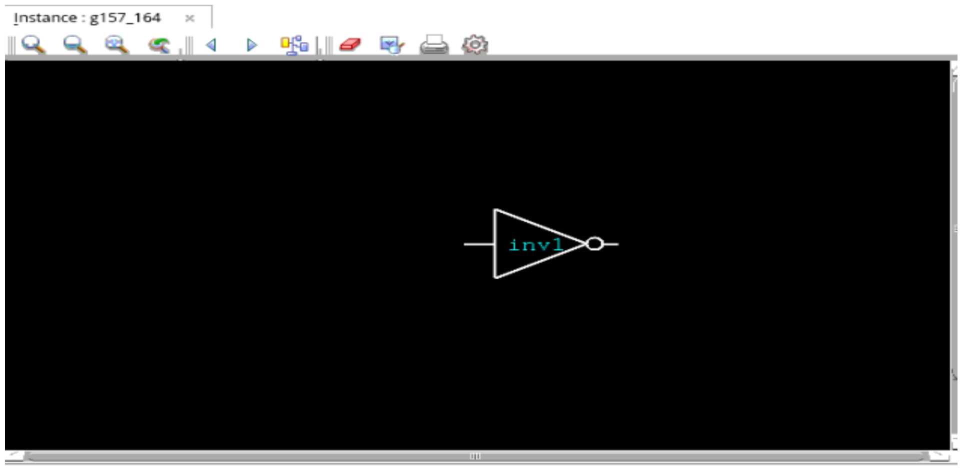


4. Cross-probe in the design browser.

   To cross probe any object, **right**-click on the object name. This opens the context menu. From the context menu, you can choose to cross-probe the design in the HDL viewer or the schematic viewer.

   a. **Right**-click on any object under modules to open the main HDL and new schematic view.

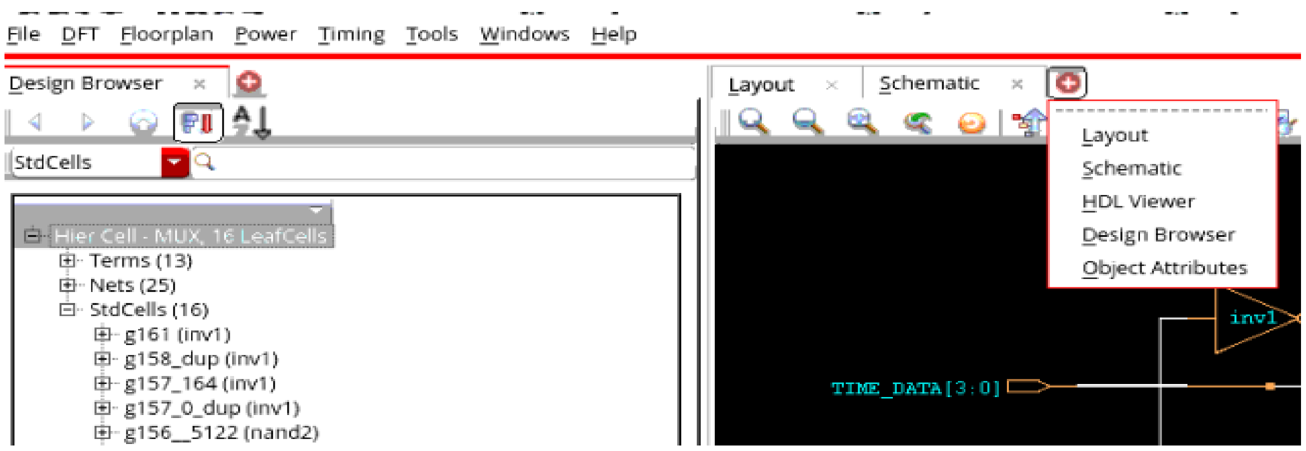




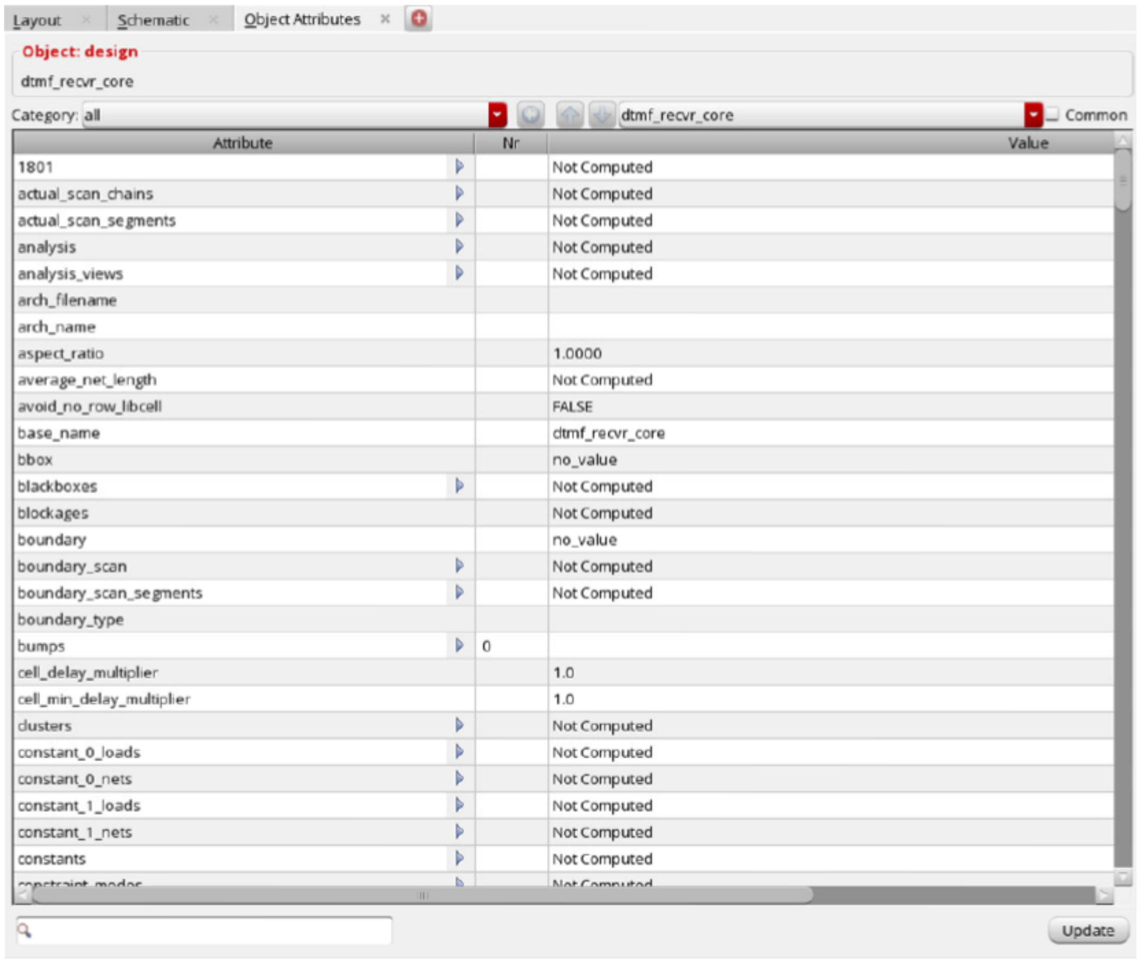
5. Open the object attribute viewer.

   a. Click on the **+** sign to open the object attribute viewer.

      The object attribute displays the value of the attributes of an object or the complete design.

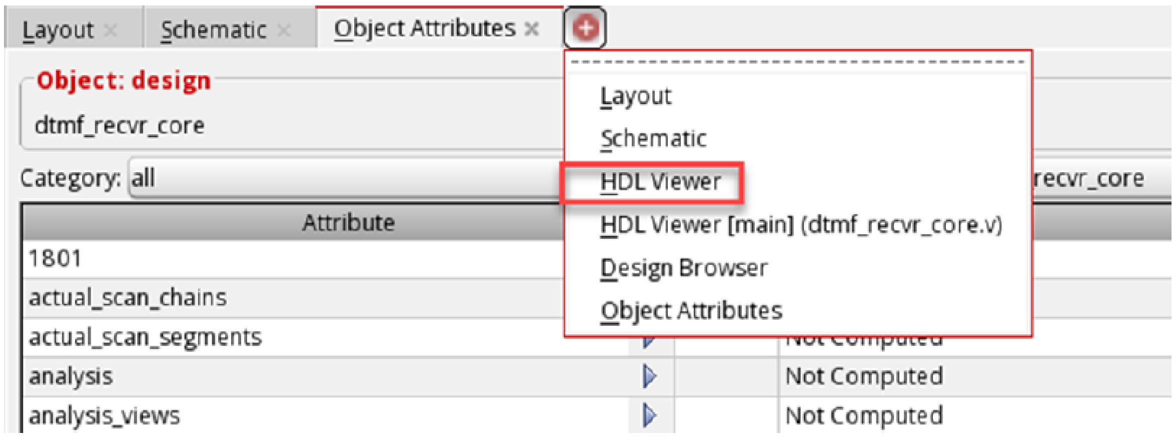


b. Explore various attributes and their values in this view.



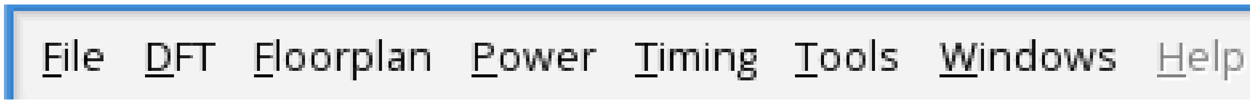
Attribute	Nr	Value
1801		Not Computed
actual_scan_chains		Not Computed
actual_scan_segments		Not Computed
analysis		Not Computed
analysis_views		Not Computed
arch_filename		
arch_name		
aspect_ratio		1.0000
average_net_length		Not Computed
avoid_no_row_libcell		FALSE
base_name		dtmf_recvr_core
bbox		no_value
blackboxes		Not Computed
blockages		Not Computed
boundary		no_value
boundary_scan		Not Computed
boundary_scan_segments		Not Computed
boundary_type		
bumps	0	
cell_delay_multiplier		1.0
cell_min_delay_multiplier		1.0
clusters		Not Computed
constant_0_loads		Not Computed
constant_0_nets		Not Computed
constant_1_loads		Not Computed
constant_1_nets		Not Computed
constants		Not Computed
constraint_modes		Not Computed

6. Open the HDL viewer.



Thus, the viewer helps to view the RTL and cross-probe the design with the other viewers. It requires cross-probing to be enabled from the toolbar.

7. Explore different reporting tabs present on the menu bar.



8. Exit the software:

```
exit
```

On Completion of This Lab Exercise

Please move on to the next lab exercise, Lab 6-4, where you will be expanding the multiplexer design.

End of Lab

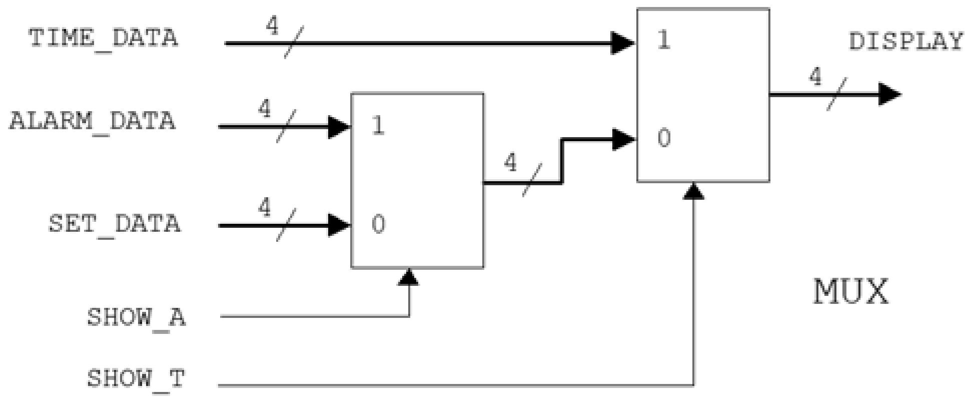
Lab 6-4 Expanding the Multiplexer

Objective: To expand the multiplexer by adding additional functionality.

The goals of this exercise are to become familiar with the structure and priority of an `if` statement by modifying the basic multiplexer to add an extra data input and data selector.

Specification

We need to add another data input `SET_DATA` and selector `SHOW_T` to the Multiplexer design. `SET_DATA` is used in the Alarm Clock when changing the time or setting a new alarm. The `SHOW_T` selector takes priority over the `SHOW_A` input.



Functionality

- If `SHOW_T` is `'1'`, the input `TIME_DATA` is assigned to the output `DISPLAY`.
- Otherwise, if `SHOW_A` is `'1'`, the input `ALARM_DATA` is assigned to the output `DISPLAY`.
- Otherwise, the input `SET_DATA` is assigned to the output `DISPLAY`.
- `TIME_DATA`, `ALARM_DATA`, `SET_DATA` and `DISPLAY` are all 4-bit `std_logic_vector` arrays.

Modifying the Multiplexer Design

1. Edit the Multiplexer design to add the new functionality. Remember, you will need to edit the entity to add the new input ports and edit the architecture to add the new behavior. Keep the architecture process to a single `if` statement.

2. Compile the design and fix any syntax errors you find.

Simulating the Design

1. Edit the `t_mux.vhd` testbench to link to the new Multiplexer design and provide a stimulus for the new inputs. Remember to edit the `component` declaration and instantiation to match the port list of the modified Multiplexer design.

   Tip: You will need to add stimulus to check the new Multiplexer operation as follows:

SHOW_A	SHOW_T	DISPLAY
'0'	'0'	SET_DATA
'0'	'1'	TIME_DATA
'1'	'0'	ALARM_DATA
'1'	'1'	TIME_DATA

2. Compile the testbench, simulate the design, and fix any functional errors you encounter.

Synthesizing the Design

Compile the design into your synthesis tool to check that the Multiplexer design has the correct syntax for synthesis. Synthesize the design and examine the gate level description produced.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 6-5, where you will be adding the alarm functionality to the Multiplexer design

End of Lab

Lab 6-5 Adding the Alarm Clock

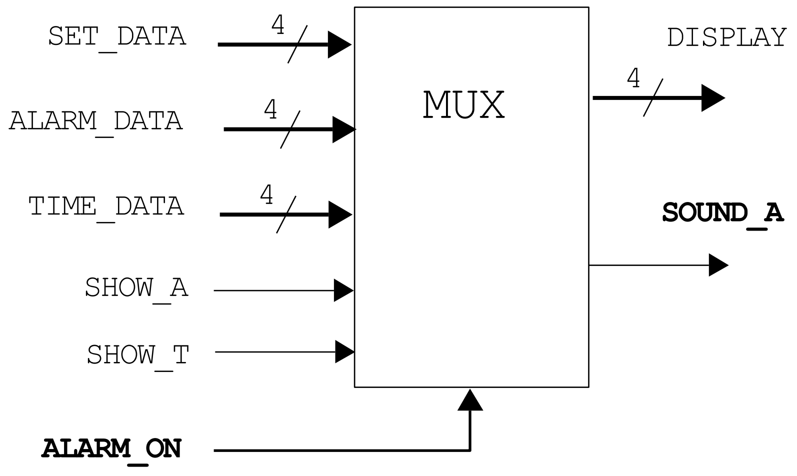
Objective: To add the alarm functionality for the previous lab design.

The goals of this exercise are to understand the use of multiple sequential statements in a process by modifying the Multiplexer design of Lab 6-4 to add alarm functionality.

Specification

We need to detect when the `TIME_DATA` is equal to the `ALARM_DATA`, so we can sound the alarm. This is done using a new output `SOUND_A`. To indicate when the alarm is disabled, or turned off after sounding, we have another input `ALARM_ON`.

The Multiplexer interface will now look like this:



New Functionality

- If `ALARM_ON` is `'1'` and input `TIME_DATA` has the same value as `ALARM_DATA` then `SOUND_A` is `'1'`.
- Otherwise, `SOUND_A` is `'0'`.

Adding the Alarm functionality

1. Edit your Multiplexer design from Lab 6-4 to add the new alarm functionality.

   Tip: Add the new functionality by inserting an extra statement in the existing process.

2. Compile the design and fix any syntax errors you find.

Simulating the Design

1. Edit the Multiplexer testbench from Lab 6-4 to link to the new design and provide a stimulus for the new input. Remember to edit the testbench component declaration and instantiation to match the port list of the modified design.

2. Compile the testbench, simulate the design, and fix any functional errors you encounter.

Synthesizing the Design

1. Compile the design into your synthesis tool to check that the design has the correct syntax for synthesis.

2. Synthesize the design and examine the gate-level description produced.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 8-1, where you will be making the final changes to the Multiplexer design by adding a Seven-Segment Display Driver.

End of Lab

Module 7: Concurrent and Sequential Statements

There are no labs for this module

Module 8: Simulation Cycle and Process Control

Lab 8-1 Adding a Seven-Segment Display Driver

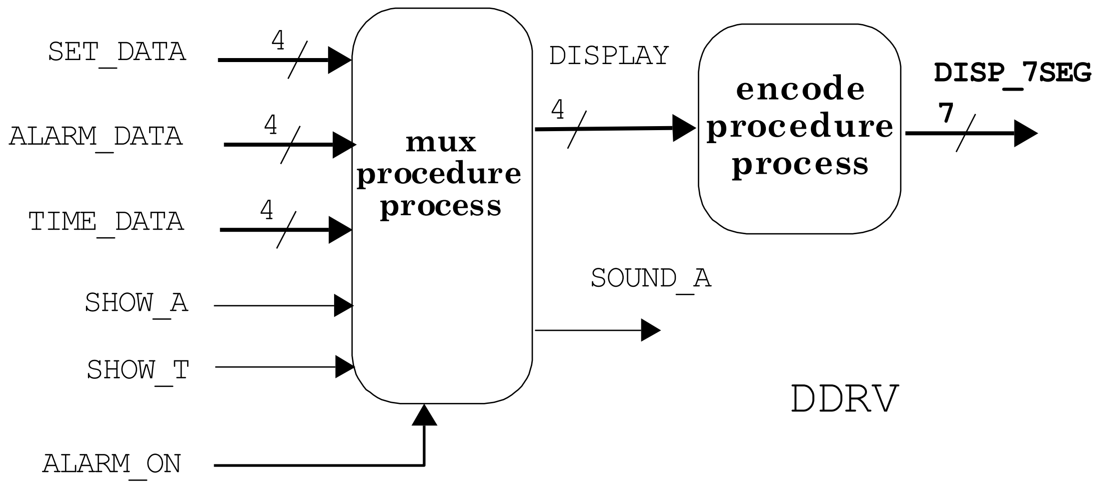
Objective: To create a seven-segment display driver.

The goal of this exercise is to modify the Multiplexer design of Lab 6-5 by adding a new combinational process to encode the output and drive a seven-segment display.

Design Specification

We will create a new design called `DDRV`. This will have the same basic function as our Multiplexer design, except that the output is now seven bits wide to drive a seven-segment display, rather than the four bits to represent the value in binary.

The `DDRV` process structure will look like this:



New Functionality

1. The 4-bit `DISPLAY` digit must be encoded into its 7-bit equivalent.

2. For `DISPLAY` values that do not represent a decimal digit, e.g., `""`, an error should be shown on the seven-segment display.

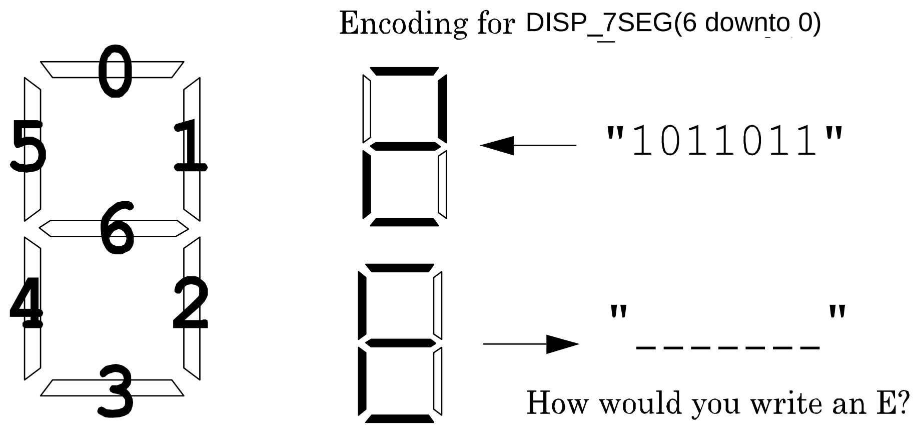
Verification

So far, we have verified our code by examining the design waveforms during simulation. It would be better if we could view a graphical representation of our display to verify that the correct segments of the seven-segment display are being lit. This is known as visualization. In this lab, we will use such a technique for viewing the `DISP_7SEG` output.

Background Information

Seven-Segment Encoding

We will be using a seven-segment LCD display for each digit of the Alarm Clock. The input to the display is a 7-bit vector where each bit represents one segment in the display. If the bit is `'`, the segment is “lit.”



Bit - segment relationship

The encoding for the seven-segment display is defined for you in a package named `ALARM_TYPES` in the file `p_alarm_types.vhd`.

Note: Binary values `"1010"` to `"1111"` cannot be represented on the seven-segment display. For all these, as well as any unknown values such as `"UUUU”`, we will indicate an error condition by outputting the encoding to display an `E`.

Visualization for the Seven-Segment Display

VHDL can write the value of any signal to an external text file in a format that can be displayed in a graphical window. A VHDL procedure called `DISP1` is supplied to perform this operation. This functionality allows us to verify our design using a graphical representation of `DISP_7SEG` rather than examining bit patterns in a waveform viewer.

- The `DISP1` procedure is defined for you in a package named `P_DISP1` in the file `p_disp1.vhd`.
- You are not expected to understand the procedure declaration or how to write to external files at this stage. These topics are covered in the modules on Functions and Procedures and Testbench Coding Styles.

Modifying the Design

The seven-segment encodings for each digit are declared as constants in the `ALARM_TYPES` package in the file `p_alarm_types.vhd`.

1. Edit this file to add a constant with the encoding to display the letter `E`. We will use this to indicate an error condition in the output.
2. Compile the `ALARM_TYPES` package into your work directory.
3. Copy your MUX design to a new file named `ddrv.vhd` and rename them to `DDRV`; also, remember to change the entity name referenced in the architecture.
4. Add a `use` statement before the `DDRV` entity to make the contents of the `ALARM_TYPES` package visible to your design:

   ```
   use work.ALARM_TYPES.all;
   ```

5. Add a combinational to convert the 4-bit `DISPLAY` digit into its equivalent seven-segment encoding. Use your error constant to output an `E` for all the 4-bit values without a single digit equivalent (e.g.,`″1010″` to `″1111″`). Remember to update the name and type of the output port.

   Tip: A `case` statement is the easiest construct to use.

6. Compile the `DDRV` design.

Modifying the Testbench

1. Copy the `t_mux.vhd` testbench file from the previous exercise to a new file named `t_ddrv.vhd` and rename the test bench entity to `T_DDRV`.
2. Update the component declaration and instantiation to match the `DDRV` design.

3. We need to add a stimulus to check the 7-segment encoding. We can use a `for loop` statement to achieve this.

   a. Add the following `use` clause to the top of your testbench immediately after the `std_logic_1164` reference as shown:

   ```
   library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.numeric_std.all;     // ADD THIS LINE HERE
   ```

   b. The following code will apply a full binary sequence to a testbench signal, e.g., `TIME_DATA`, at intervals of 10 ns. It uses language features that are covered in the Arithmetic Operators module on day 2 of the VHDL Application Workshop:

   ```
   for I in 0 to 10 loop
     TIME_DATA  <= std_logic_vector(to_unsigned(I, 4));
     wait for 10 ns;
   end loop;
   ```

4. Add the visualization procedure**,** `DISP1`, using the following **concurrent** statement (i.e., the statement is placed *outside* of a process):

   ```
   DISP1(DISP_7SEG);
   ```

5. Don't forget to add a `use` statement before your entity so that the package containing the visualization procedure, `P_DISP1,` is visible to your design:

   ```
   use work.P_DISP1.all;
   ```

Simulating the Design

1. Compile the `T_DDRV` testbench.

2. Simulate the design.

3. If you receive error messages when you elaborate the design or start the simulator, then it is possible you did not modify the names of the ports and the component correctly when you created the `DDRV` and `T_DDRV` designs. Note the following points:

   a. The Design Driver entity name (`DDRV)` must match exactly the testbench component declaration name (`DDRV)`. Also, the Design Driver entity port list must match that of the testbench component declaration. These are the rules for a default configuration link between the testbench and entity.

b. Connection of `DDRV` ports to testbench signals is made in the port map of the component instantiation statement. Each port in the `DDRV` component declaration should be mapped to a testbench signal.

Output Visualization

The testbench will create a file called `clock.txt` containing a character representing the seven-segment display.

1. View `clock.txt` file in an editor. You may need to keep refreshing the file to see the changes in the output.

2. Run the simulation in short steps, so each output value can be observed in the seven-segment display.

3. **Find the bug!** There is a bug in one of the fixed values which define the seven-segment display output. Use the simulation and output visualization to find the error. Fix the error and re-simulate to confirm the bug is fixed.

Synthesizing the Design

1. Compile the design into your synthesis tool to check the design is synthesizable.

2. Synthesize the design and examine the gate-level description produced.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 13-1, where you will create a registered process.

End of Lab

Module 9: Variables and Sequential Statements

There are no labs for this module

Module 10: Arithmetic Operators

There are no labs for this module

Module 11: VHDL Coding Styles

There are no labs for this module

Module 12: The Synthesis Process

There are no labs for this module

Module 13: Definition of RTL Code

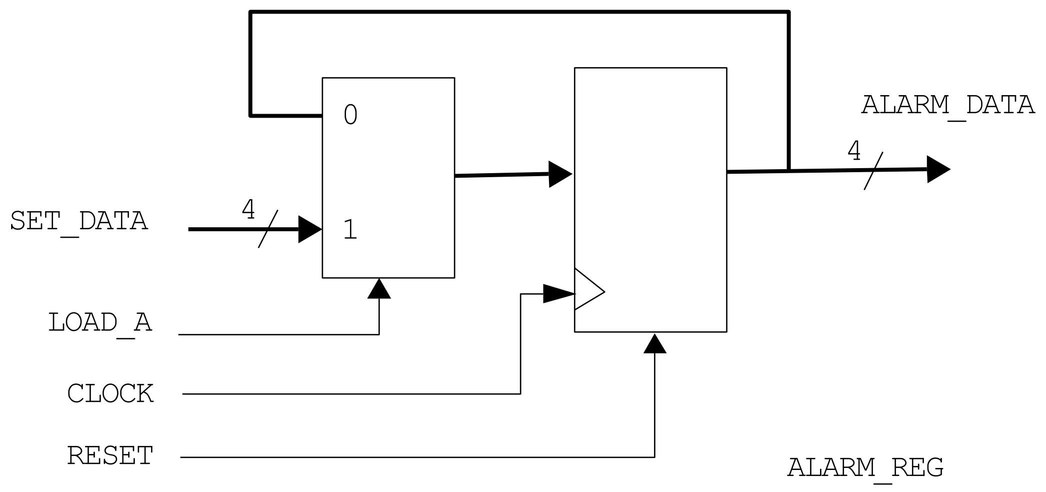
Lab 13-1 Alarm Register

Objective: To create a register to store the alarm time of the clock.

The goals of this exercise are to create a loadable, registered process to store the alarm time of the clock.

Specification

The alarm register is a simple register configuration that either retains its existing value or loads a new value:



The functionality is as follows:

- If `LOAD_A` is `'1'`, the input `SET_DATA` is stored in the register.
- Otherwise, the current value of `ALARM_DATA` is retained in the register.
- `RESET` is asynchronous and active low.
- The register is clocked on the rising edge of `CLOCK`.
- `ALARM_DATA` and `SET_DATA` are all 4-bit `std_logic_vector` arrays.

Creating the Design

1. Create a new file called `alarm_reg.vhd`, containing an entity called `ALARM_REG` and architecture called `RTL`.

2. Write the Alarm Register model.

   Hint: To successfully create a registered process, you must follow a strict coding style. The easiest way to implement a loadable register is to use synchronous feedback. Therefore, you must use a process with the following structure:

```
<name>: process (CLOCK, RESET)
begin
  if <RESET is active> then
    <assign reset value to output>
  elsif <rising edge on CLOCK> then
    if <need to load a new value> then
      <load new value>
    end if;
  end if;
end process <name>;
```

Modifying the Testbench

Some of the testbench code is supplied in the file `t_alarm_reg.vhd.`

1. Complete the testbench model by adding the missing information indicated by the comments.

2. Add the required stimulus to verify your design functions correctly.

Simulating and Synthesizing the Design

1. Simulate the design and verify the functionality.

2. Synthesize the design and examine your results. Check that the synthesis tool has correctly inferred four registers from your design.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 14-1, where you will be using a more complex registered process to create a counter.

End of Lab

Module 14: Synthesis of Mathematical Operators

Lab 14-1 Pulse Generator

Objective: To create a pulse generator with the required timing.

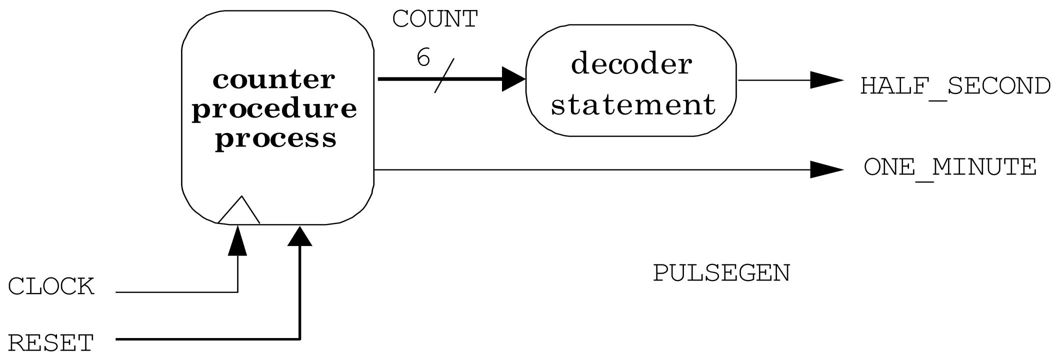
The goals of this exercise are to create a counter which will divide down the main clock and generate pulses with the required timing. Creating the counter will also give you experience in using a vector arithmetic package.

Specification

The Alarm Clock requires a ONE_MINUTE timing pulse for the normal time operation and a HALF_SECOND pulse for adjusting the alarm and time. Both pulses are generated from the main 128Hz clock using a counter.

For a 128Hz clock, the counter must count 128*60 = 7680 cycles to mark one minute of time. This requires a 13-bit range for the counter ($7679_{10} = 1110111111111_2$).

Similarly, the counter must count 128/2 = $64_{10} = 1000000_2$ cycles to mark one half-second of time.



Functionality

- The counter process must count 7680 cycles on the positive edge of the clock.
- ONE_MINUTE will be set to 1′ for one clock cycle when *all* 13 bits of COUNT are ′0′. This can be included as part of the counter process.
- HALF_SECOND will be set to 1′ for one clock cycle when the 6 *least* significant bits of COUNT are ′0′.
- COUNT is a 13-bit signal of type unsigned.
- The counter is positive edge triggered, and RESET is asynchronous and active low.

Background Information

Arithmetic of `std_logic_vector` Types

You need to use relational and arithmetic operators on vector types to create the counter process and decode logic. As we saw in the Arithmetic Operators module, we need to use the IEEE `numeric_std`. Vector Arithmetic Package for this. Declarations for the packages are listed in your tool documentation. Please refer to the package listings when creating your code.

Creating the Design

This is a design where careful structuring of combinational and registered processes is important. Remember, when `COUNT` is zero, both `ONE_MINUTE` and `HALF_SECOND` must be `'1'` in the same clock cycle.

1. Create a new VHDL design file called `pulsegen.vhd`. Name the entity `PULSEGEN` and the architecture `RTL`.

2. Add a `library` and `use` clause to reference the IEEE `numeric_std` Vector Arithmetic Package.

3. Create a counter to count from 0 to 7679 (7680 cycles) using a signal named `COUNT`.

   Hint: Use the following process structure:

```
<name>: process (CLOCK, RESET)
begin
  if <RESET is active> then
    <assign reset value to COUNT>
  elsif <rising edge on CLOCK> then
    if <COUNT = maximum> then
      <clear COUNT>
    else
      <increment COUNT>
    end if;
  end if;
end process <name>;
```

4. We could add another piece of combinational logic to decode `COUNT` and create the `ONE_MINUTE` timing pulse. However, we already detect when `COUNT` is zero when we make the assignment as part of the counter functionality. Therefore, we could include the assignments for `ONE_MINUTE` as part of the counter process.

5. Write the code to create the `ONE_MINUTE` output.

6. Add a combinational process or concurrent statement to decode the 6 least significant bits of `COUNT` to create the `HALF_SECOND` output.

Completing the Test Bench

1. Modify the supplied test bench in the file `t_pulsegen.vhd`.

2. You will need to generate a clock signal in your test bench, and this requires the `CLOCK` signal to be initialized to `’1’` or `’0’`.

   Initialize `CLOCK` when the signal is declared:

   ```
    signal CLOCK : std_logic := '0';
   ```

Simulating and Synthesizing the Design

1. Simulate the design and verify the functionality. Check that `ONE_MINUTE` and `HALF_SECOND` are both high for one clock cycle only, and when `COUNT` is zero, `ONE_MINUTE` and `HALF_SECOND` are both high for the same clock period.

2. Synthesize the design and examine your results. Check that the synthesis tool has correctly inferred the correct number of registers from your design.

   Note: If you created `ONE_MINUTE` by adding code to the registered process, then you will have an additional register for the `ONE_MINUTE` output. If `ONE_MINUTE` was created using a combinational process or concurrent statement, you would have additional combinational logic but no register for `ONE_MINUTE`.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 15-1, where you will create the Alarm Clock Controller state machine.

End of Lab

Module 15: Finite State Machine (FSM) Design and Analysis

Lab 15-1 Alarm Clock Controller

Objective: To design the alarm clock controller to control the alarm clock.

The goals of this exercise are to create a simple Finite State Machine (FSM) which will control the alarm clock.

Specification

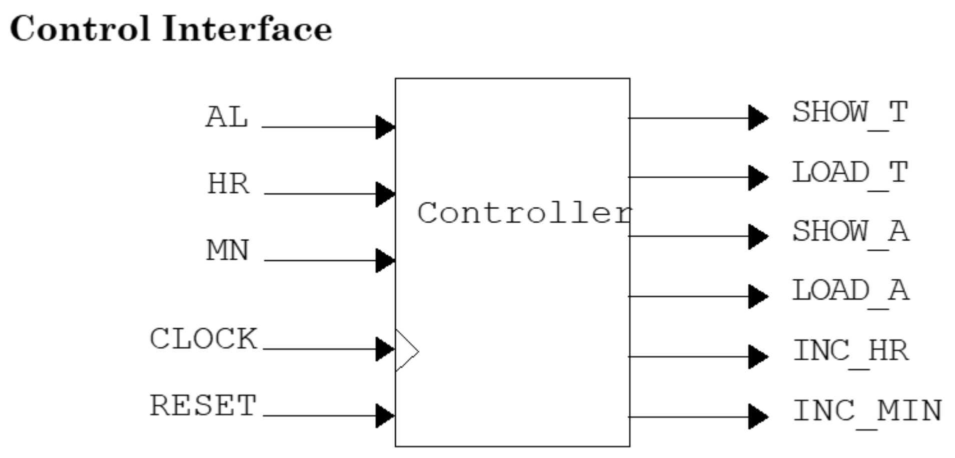
The controller state machine has three inputs generated from momentary buttons:

- `AL` – Alarm Mode. When pressed and held, the Alarm Clock shows the current alarm, and pressing the `HR` or `MN` button will change the alarm. When released, the current time is displayed, and the `HR` or `MN` button will change the time.
- `HR` – Hour Set. When pressed and held, the hour digits of the display increment.
- `MN` – Minute Set. When pressed and held, the minute digits of the display increment.

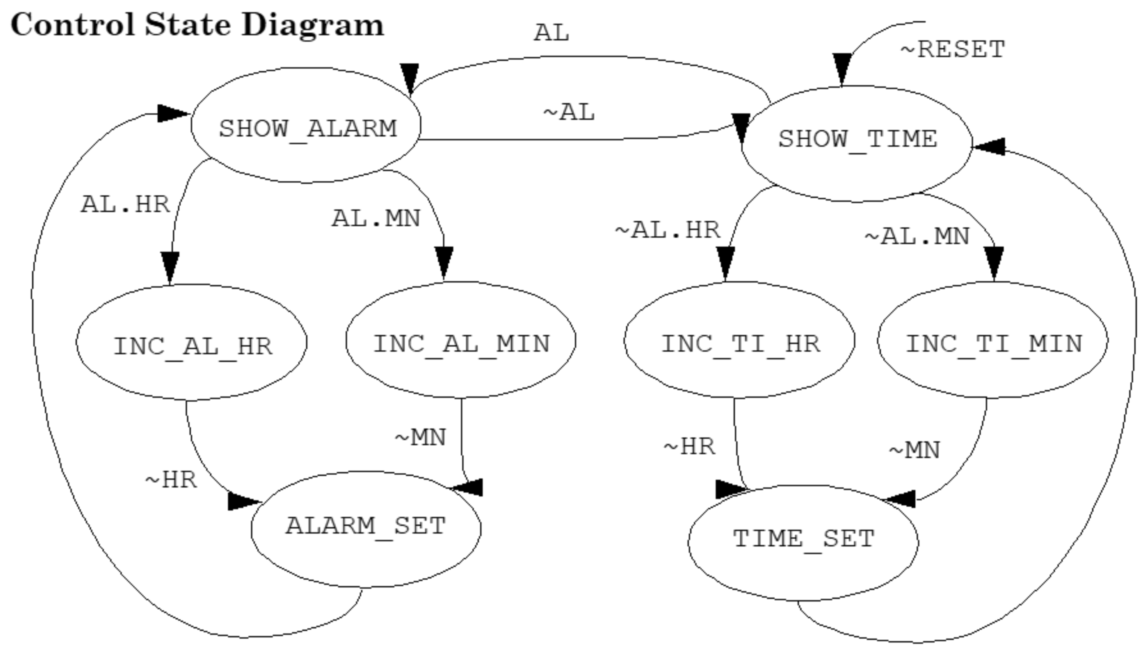
The state machine has six outputs, with the following functionality.

- `SHOW_T` – Selects the current time for the `DISPLAY_DRIVER` and `TIME_SET` components, displaying the time and enabling the time for adjustment.
- `LOAD_T` – Stores an adjusted time as the new time.
- `SHOW_A` – Selects the alarm time for the `DISPLAY_DRIVER` and `TIME_SET` components, displaying the alarm and enabling the alarm for adjustment.
- `LOAD_A` – Stores an adjusted alarm as the new alarm.
- `INC_HR` – Increments hour digits in `TIME_SET` component.
- `INC_MIN` – Increments minute digits in `TIME_SET` component.

The FSM is positive edge triggered, and `RESET` is asynchronous and active low.



The Alarm Clock Controller must generate the outputs from the inputs according to the following state diagram and output decode table.



- State transition conditions use “~” to mean the input is `'0'`, and “`.`” to indicate the logical "and" of two conditions.

  i.e., `~AL.HR` is the condition (`AL ='0'`) and `(HR = '1')`.

- The initial state of the FSM is `SHOW_TIME`.

- If no condition is shown, the transition occurs on the next clock edge.

- `HR` has a higher priority than `MN` if both are pressed at the same time.

Output Decode Table

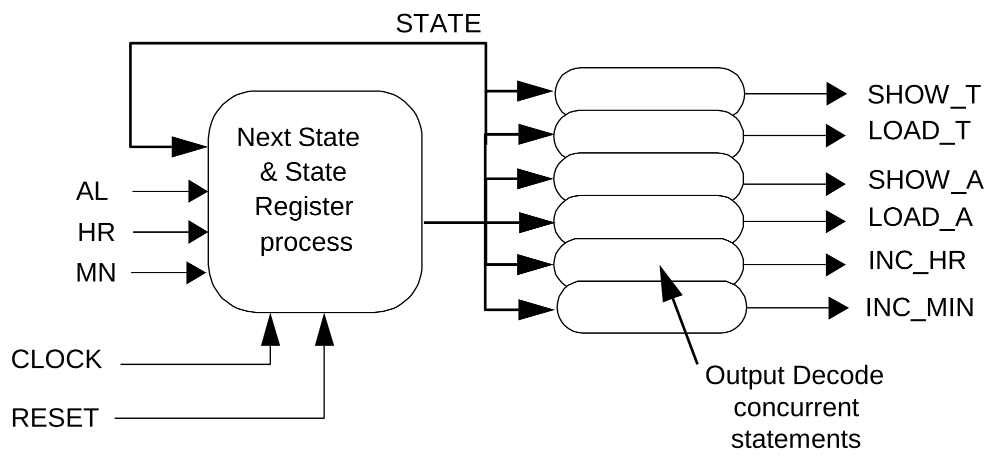
STATE / OUTPUTS	SHOW_T	LOAD_T	SHOW_A	LOAD_A	INC_HOUR	INC_MIN
SHOW_TIME	1	0	0	0	0	0
INC_TI_HR	0	0	0	0	1	0
INC_TI_MIN	0	0	0	0	0	1
TIME_SET	0	1	0	0	0	0
SHOW_ALARM	0	0	1	0	0	0
INC_AL_HR	0	0	0	0	1	0
INC_AL_MIN	0	0	0	0	0	1
ALARM_SET	0	0	0	1	0	0

Note: This is not the most efficient implementation of the Alarm Clock Controller, but it is the simplest, safest, and easiest to test.

- Create the entity `CONTROL` and the architecture `RTL` for the FSM in a file named `control.vhd`.
- Declare an enumerated type for the Controller states.
- Write the model for the `CONTROL` FSM.

  Tip: Use a single registered process for the next-state and state register logic. Use concurrent statements to decode the state and assign the outputs.

Suggested FSM Process Structure


STATE
Next State & State Register process
AL
HR
MN
CLOCK
RESET
SHOW_T
LOAD_T
SHOW_A
LOAD_A
INC_HR
INC_MIN
Output Decode concurrent statements

Completing the Test Bench

1. Examine the supplied test bench in the file `t_control.vhd`.

2. Check the testbench Controller component declaration and instantiation matches your `CONTROL` entity.

The testbench uses an “array of records” data type containing both stimulus and expected results in one data structure. The testbench applies the stimulus and checks that the FSM output matches the expected results.

Simulating and Synthesizing the Design

1. Compile and simulate the design. The testbench uses assertion statements to report any errors that are detected. Fix any errors found.

2. Synthesize the design and examine your results. Check that the synthesis tool has correctly inferred the correct number of registers from your design.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 16-1, where you will create the full counting mechanism for the Alarm Clock.

End of Lab

Module 16: Synthesis Coding Styles

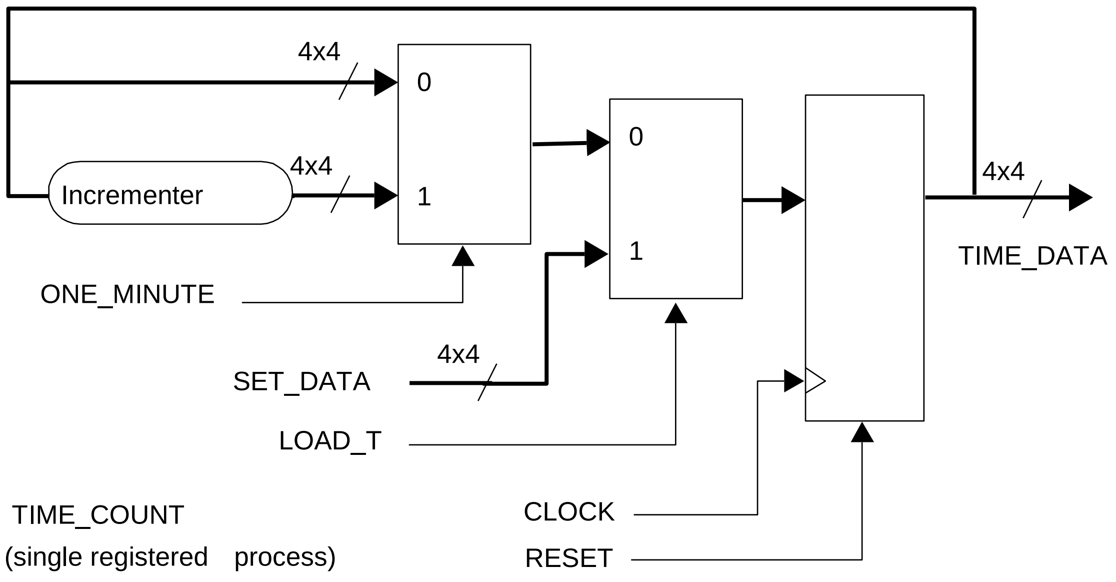
Lab 16-1 Full Counting Mechanism for the Alarm Clock

Objective: To design a counting mechanism to count hours and minutes of the alarm clock.

The goal of this lab exercise is to create a complex, multi-digit counter using a sophisticated "array of array" data type, which will count the hours and minutes of the Alarm Clock.

Specification

The Alarm Clock `TIME_COUNT` component will count the `ONE_MINUTE` pulses generated by`PULSEGEN` to generate the four clock digits that will be displayed. The counter will be loadable so that we can set a new time value.



Functionality

- `TIME_DATA` and `SET_DATA` consist of four 4-bit arrays, one 4-bit array for each digit in the Alarm Clock. This is Binary Coded Decimal (BCD) format.
- The incrementer operates on all four digits, incrementing them in the pattern of a normal 24-hour clock.
- The incrementer only increments the time value when `ONE_MINUTE` is `'1'`. `ONE_MINUTE` will be `'1'` for only one `CLOCK` period. While `ONE_MINUTE` is `'0'`, the time remains unchanged.

- If `LOAD_T` is `'1'` on the rising edge of `CLOCK`, the counter is loaded from `SET_DATA`.
- The register is positive edge triggered. `RESET` is asynchronous and active low.

Background Information

Binary Coded Decimal

On the LCD of the Alarm Clock, we need to display the four digits of the time or alarm individually in decimal. Therefore, it is easier to use a separate 4-bit array for each decimal digit. This format is called Binary Coded Decimal (BCD).

The data structure to hold BCD can be created with an "array-of-arrays":

```
type DISPLAY_T is array (natural range <>) of unsigned(3 downto 0);
subtype DISPLAY_4 is DISPLAY_T(3 downto 0);
```

These type declarations can be found in the `ALARM_TYPES` package. The-of-arrays allow us to represent the time data as follows (where MS means Most Significant, and LS Least Significant).

array index	3	2	1	0
contents	MS hour	LS hour	MS minute	LS minute
binary	"0001"	"0101"	"0010"	"0011"
decimal	1	5	2	3

Although this structure makes the final display easier, it does make incrementing the time a little more difficult.

Counting Algorithm

- `TIME_COUNT` must increment the time in the pattern of a normal 24-hour clock, i.e., from `00:00` to `23:59` and back to `00:00`, where the two-minute digits count up to `59` and the two-hour digits to `23`.
- A set of rules defining the behavior of the time counting mechanism are below (you may be able to devise a more efficient algorithm to increment the time):

```
if MS_HOUR = 2, LS_HOUR = 3, MS_MIN = 5 and LS_MIN = 9
   set MS_HOUR, LS_HOUR, MS_MIN, LS_MIN to 0
```

```
elsif LS_HOUR = 9, MS_MIN = 5 and LS_MIN = 9
   set LS_HOUR, MS_MIN, LS_MIN to 0
   increment HOUR_TEN

elsif MS_MIN = 5 and LS_MIN = 9:
   set MS_MIN, LS_MIN to 0
   increment LS_HOUR

elsif LS_MIN = 9:
   set LS_MIN to 0
   increment MS_MIN

else:
   increment LS_MIN
```

Implementation

The declaration for the `DISPLAY_4` type can be found in the `ALARM_TYPES` package in the file `p_alarm_types.vhd`. The type declaration is placed in the package as we will be using this type with other components in the design.

1. *Modify the supplied model* in the file `timecount.vhd`, using the information above. Use a **single registered process** for the counting logic.

   Tip: Remember that VHDL does not allow you to read the value of output ports. You will need to declare local signals to implement the counting algorithm and then assign the output ports from these signals.

2. *Modify the supplied testbench* in the file `t_timecount.vhd`. You will need to add a stimulus to check the rollover for each of the time digits. Use `LOAD_T` to load time values from `SET_DATA` and clock the counter to check for correct operation. The rollover from `23:59` to `00:00` is especially important to check for!

3. Compile and simulate the design to verify the functionality.

4. Synthesize the design and check your results. Confirm that the synthesis tool has correctly inferred the correct number of registers from your design.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 18-1, where you will create a component to adjust the time and alarm of the Alarm Clock.

End of Lab

Module 17: Functions and Procedures

There are no labs for this module

Module 18: Advanced Concurrent VHDL

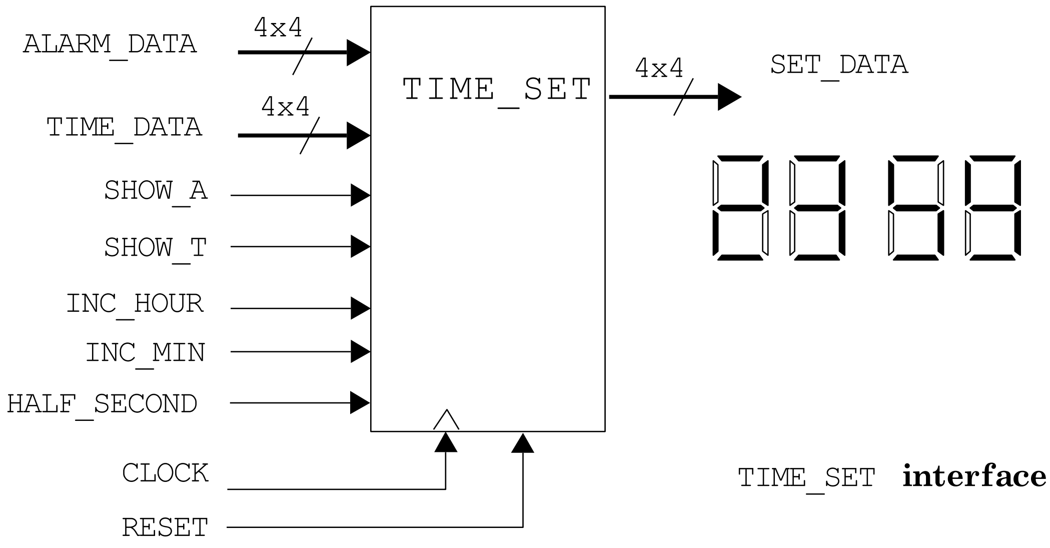
Lab 18-1 Time and Alarm Adjustment

Objective: To design the time and alarm adjustment mechanism.

The goal of this lab exercise is to use parameterized, reusable counter components in a structural design to create the time and alarm adjustment mechanism.

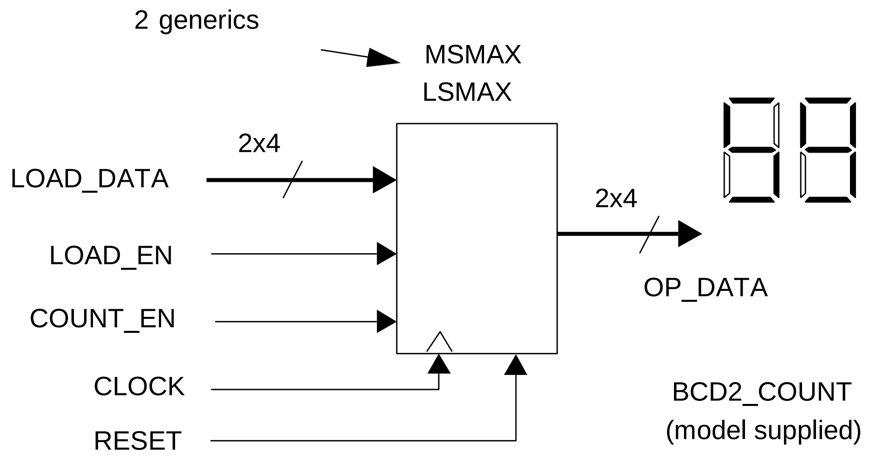
Specification

The `TIME_SET` component adjusts the time and alarm, under the control of the signals generated by the Alarm Clock Controller state machine. The current alarm or current time can be loaded using `SHOW_A` or `SHOW_T`. The minutes or hours digits are separately adjusted using `INC_HOUR` or `INC_MIN`. Digits are incremented on every `HALF_SECOND` pulse.

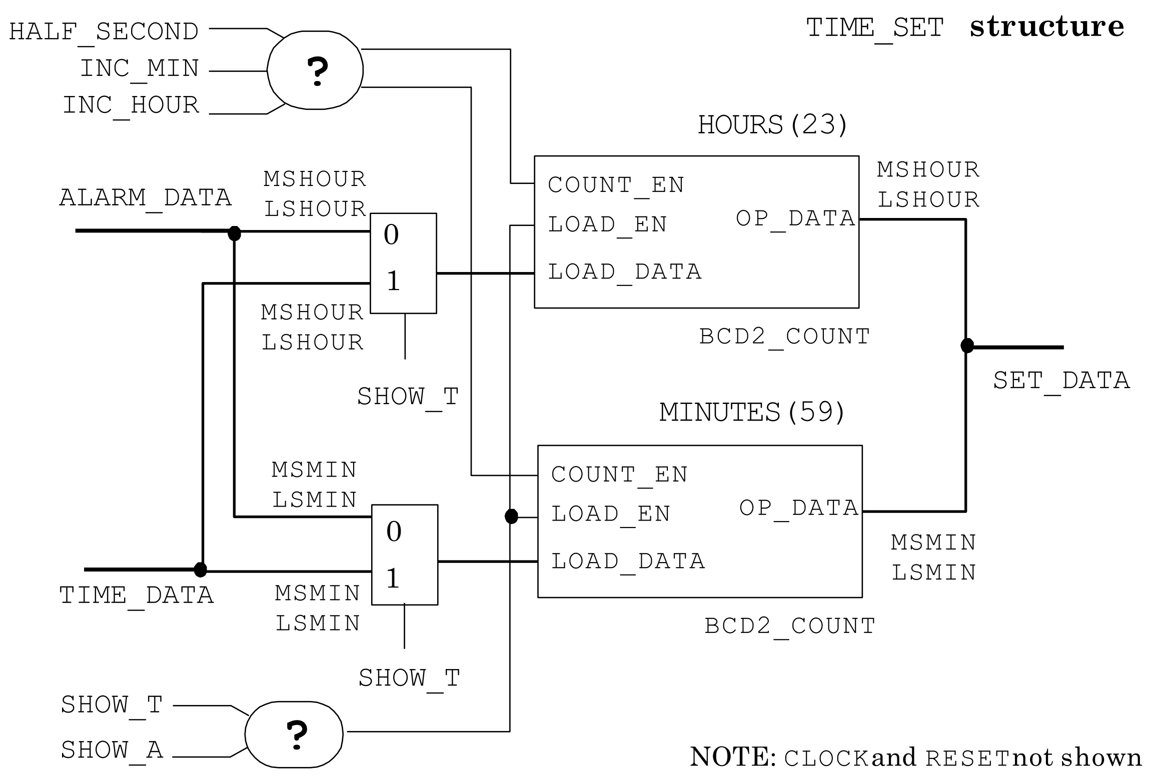


`TIME_SET` **interface**

To set the time and alarm, `TIME_SET` uses two loadable 2-digit BCD counters. As you already have experience in creating BCD counters, these are provided as a reusable component `BCD2_COUNT`, using generics to set the maximum count for each digit.



TIME_SET must make two instantiations of BCD2_COUNT, one for the hours, digits, and one for the minutes.



Where the relationship between BCD digit and DISPLAY_4 array-of-arrays is as follows:

array index	3	2	1	0
BCD digit	MSHOUR	LSHOUR	MSMIN	LSMIN

Functionality

- `TIME_SET` contains two instantiations of `BCD2_COUNT` (`HOURS` and `MINUTES`).
- For both, `LOAD_EN` is `'1'` when either `SHOW_A` or `SHOW_T` is `'1'`.

`HOURS` Instantiation

1. Maximum count is 23. Use these values for the generics.
2. `LOAD_DATA` is assigned from the 2 *most* significant elements of `TIME_DATA` when `SHOW_T` is `'1'`; otherwise, it is assigned from the 2 *most* significant elements of `ALARM_DATA`.
3. `COUNT_EN` is `'1'` when both `HALF_SECOND` and `INC_HOUR` are `'1'`.
4. `OP_DATA` is assigned to the 2 *most* significant elements of `SET_DATA`.

`MINUTES` Instantiation

1. Maximum count is 59. Use these values for the generics.
2. `LOAD_DATA` is assigned from the 2 *least* significant elements of `TIME_DATA` when `SHOW_T` is `'1'`; otherwise, it is assigned from the 2 *least* significant elements of `ALARM_DATA`.
3. `COUNT_EN` is `'1'` when both `HALF_SECOND` and `INC_MIN` are `'1'`.
4. `OP_DATA` is assigned to the 2 *least* significant elements of `SET_DATA`.

Implementation

1. Compile the file `bcd2_count.vhd`, which contains the `BCD2_COUNT` component.
2. Write the `TIME_SET` model using the information above. Type declarations for the 2- and 4-digit BCD arrays can be found in the `ALARM_TYPES` package. Remember, you will need a generic map when instantiating the `BCD2_COUNT` components.

3. *Modify the supplied testbench* in the file `t_time_set.vhd`. You will need to add a stimulus to check the rollover for each of the time digits. The hours rollover from `23` to `00`, and the minutes rollover from `59` to `00` are especially important to check for!

4. Compile and simulate the design to verify the functionality.

5. Synthesize the design and check your results. Confirm that the synthesis tool has correctly inferred the correct number of registers from your design.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 19-1, where you will be finally putting together the complete Alarm Clock design.

End of Lab

Module 19: Advanced Data Types

Lab 19-1 Putting It All Together

Objective: To assemble the Alarm Clock from the components.

The goal of this lab exercise is to create a structural model which will assemble the complete Alarm Clock from the components you have already designed.

Specification

We can now assemble the complete Alarm Clock design from the components we have ready developed, together with some technology-specific components provided for you.

The Alarm Clock Block Diagram in Lab 1 describes how the design is assembled. You need to modify two of your existing components to handle all four digits of the Alarm Clock display.

- **Alarm Register**: Needs to read and store 4-digit data types.
- **Display Driver**: Needs to be expanded from single to 4-digit data types.

In addition, there are three technology-specific components supplied for you in order to complete the design:

- **Debounce:** The Alarm Clock is controlled by push buttons. The Debounce component synchronizes these button inputs to the design clock and “debounces” the signals, making the inputs switch cleanly without glitches or multiple transitions.
- **Debounce Toggle:** Same as Debounce, except it converts a push-button input (“on” when pressed, “off” when released) into a toggle (toggle between “on” and “off” when pressed). This is used with the `ALRMON` input to turn the Alarm on and off.
- **LCD Driver:** The LCD Driver performs several functions:
  - Creates the reference clock for driving the LCD.
  - Converts the display segment signals into driving waveforms with the correct timing for the specific LCD.
  - Contains a `FLASH` process to pulse any of the display segments. In the LCD driver supplied, this is used to pulse the colon segment of the display when the alarm is enabled.
  - Contains code to strobe the LED. This is used to indicate when the alarm is sounding.

Alarm Register

The `alarm_reg.vhd` component completed in Lab 13-1 operates on a single digit of the Alarm Clock. You need to modify the component to handle all four digits.

1. Copy your Alarm Register component into a file named `alarm_reg4.vhd` and rename the entity to `ALARM_REG4`.

2. Modify the `ALARM_REG4` component to handle all four digits of the display. Depending on your code, this could be achieved simply by changing data types to the `DISPLAY_4` type declared in the `ALARM_TYPES` package.

Display Driver

The `ddrv.vhd` component completed in Lab 8-1 operates on a single digit of the Alarm Clock. You need to modify the component to handle all four digits.

1. Copy your Display Driver component into a file named `ddrv4.vhd` and rename the entity to `DDRV4`.

2. Modify the `DDRV4` component to handle all four digits of the display. Depending on your code, this could be achieved by:

   a. Changing the input data types to `DISPLAY_4` type declared in the `ALARM_TYPES` package.

   b. Creating four instances of the Seven Segment decoder process to decode each digit of the display.

      Tip: You can use a `generate` statement to instantiate processes or concurrent statements as well as components.

```
BIN2SEVEN: for I in 0 to 3 generate

  ENCODE: process (DISPLAY(I))
  begin
    ...
  end process ENCODE;

end generate BIN2SEVEN;
```

   c. Changing the output data type to a 4-element array of 7-bit vectors.

      Tip: Check the `ALARM_TYPES` package for a suitable type declaration.

Alarm Clock

Refer to the block diagram in Lab 1-2 for help in creating the structural model.

1. Add declarations for your components to the Alarm Clock structural model in the file `alarm_clock.vhd`.

2. Instantiate a `DEBOUNCE` component for each of the `SHARLM`, `MINS` and `HOURS` push-button inputs. Instantiate a `DEBOUNCE_TOGGLE` component for the `ALRMON` input.

   Tip: You could use a `generate` statement to instantiate the `DEBOUNCE` components. You will need to concatenate the inputs into a vector. Then you can use the generate loop variable to index the vector and connect each input to an instantiation. You could even use `if..generate` statements to instantiate both types of Debounce components in one generate. Alternatively, you could create individual instantiations.

3. Instantiate and connect your components, declaring any necessary additional local signals as required.

4. The outputs from the Display Driver are concatenated together with three decimal points (set to `'0'` in this design) and the Alarm Active (`ALARM_ON)` signal into a vector named `LCD_IN`.

   Make sure the outputs from your Display Driver component match the signal names used in the `LCD_IN` concatenation. Edit the names in the concatenation if required.

5. Compile the design hierarchy, making sure you also compile the pre-supplied components `DEBOUNCE` (`debounce.vhd); DEBOUNCE_TOGGLE (debounce_toggle.vhd)` and `LCD_DRIVER` (`lcd_driver.vhd).`

Completing the Testbench

A testbench is provided in the file `t_alarm_clock.vhd`. This file also contains code to extract the display digits from the LCD drive signals.

For this lab exercise, we will only check the basic time counting of the Alarm Clock; therefore, the push-button inputs are kept at `'1'`, and only the `CLOCK` and `RESET` are driven with waveforms. In the next lab, we will test the full functionality with a sophisticated script-driven testbench.

1. Check the testbench in the file `t_alarm_clock.vhd` to make sure you understand how the Alarm Clock is tested.

Visualization of the Seven-Segment Display

We can use a new version of the display visualization procedure to handle all four digits of the clock. A new procedure `DISP4` has been defined in the package `P_DISP4` in the file `p_disp4.vhd`. This procedure is called in the testbench, allowing us to view all the display digits in the output file.

Simulating and Synthesizing the Design

1. Compile the testbench and invoke the simulator. Use your simulator navigation windows to *check that the structure of your design is complete* before simulation!

2. Run the simulation and use a waveform viewer to check the time successfully counts from `00:00` to `23:59` and back to `00:00`.

3. With a 10 ns clock period and 7680 clocks/minute (see Lab 14-1 Pulse Generator), the simulation must be run for at least **112 ms** (10 x 7680 x 60 x 24 ns).

4. Depending on your simulator/platform, this simulation run may take several minutes. Time for a well-earned break!

5. Synthesize the design and examine your results.

On Completion of This Exercise

Please move on to the next lab exercise, Lab 21-1, where you will be creating a more sophisticated testbench to fully verify the Alarm Clock design.

End of Lab

Module 20: Testbench Coding Styles

There are no labs for this module

Module 21: Testbench Applications

Lab 21-1 Script-Driven Testbench

Objective: To develop a script-driven testbench to verify the complete design.

The goal of this lab exercise is to create a sophisticated script-driven testbench to verify your model of the complete Alarm Clock design

The complete Alarm Clock is more complex than anything we have tested so far, so we will use a different approach to the design of the testbench. In this lab, instead of creating an “in-line” sequence of assignments to test our design, we will use a modular, script-driven, self-checking approach to verify the full functionality of the Alarm Clock.

Modular Testbench

To keep our testbench well-organized, we will use separate subprograms to test each major function of our Alarm Clock design. The design can then be verified by calling these subprograms in the desired order with the required data.

Script-Driven Testbench

To make our testbench as flexible as possible, the sequence of subprograms and the associated data will be controlled by a series of user-defined commands read in from a text file. This makes it easy to write the code for our testbench; we simply read in a new command from the external text file, check what command it is and then call the appropriate subprogram. Once the testbench is written, creating new test sequences is very easy.

Self-Checking Testbench

We also want a convenient way of checking that our design is working correctly. Manually checking simulation waveform displays becomes impractical with complex designs and long simulation runs; a simple “pass/fail” test is required. Our self-checking testbench will use two approaches:

- We will sample the design outputs and write out the values to an external trace file. Then we can easily check if the simulation behavior is correct by comparing the trace file to a set of known good results.

- We will use assertion messages inside the testbench to check for and report on specific error conditions.

Testbench Commands

The specific commands we will use are:

Functionality	Command	Procedure	Operation
Show time (normal operation)	`TIMED DDDD`	`DO_TIMED`	Advance time to `DD:DD`
Display current alarm	`SHALM`	`DO_SHALM`	Show set alarm
Set time hour	`TI_HR DD`	`DO_TI_HR`	Increment time hour to `DD`
Set time minute	`TI_MN DD`	`DO_TI_MN`	Increment time minute to `DD`
Set alarm hour	`AL_HR DD`	`DO_AL_HR`	Increment alarm hour to `DD`
Set alarm minute	`AL_MN DD`	`DO_AL_MN`	Increment alarm minute to `DD`
Enable and disable alarm	`ALTOG`	`DO_ALTOG`	Toggle alarm state

where `D` is an integer between 0 and 9.

Our testbench will perform the following actions:

- Read a command from the external file.
- Read the correct number of data values depending on the command.
- Call the appropriate procedure with the correct arguments to execute the operation.
- Check for successful completion of the operation and report a message if an error is found.
- Execute all the commands in the file.

Note: Our testbench must be robust enough to reject mistyped commands, like "SHLAM", and handle invalid operations like TI_HR 99.

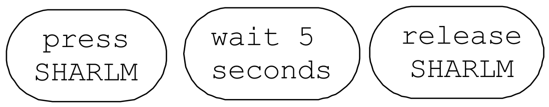
Test Subprograms Implementation

The package ALARM_TEST in the file p_alarm_test.vhd contains definitions for some of the testbench procedures. Your first objective is to complete procedures for all the testbench commands.

1. Edit the file p_alarm_test.vhd to examine the procedure definitions.

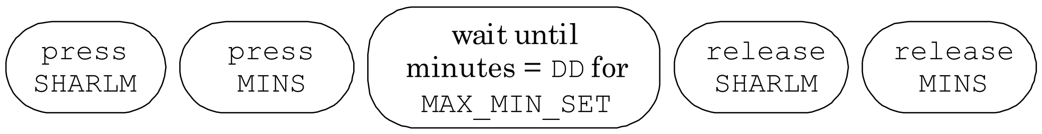
   The p_alarm_test.vhd package currently defines DO_SHALM and DO_AL_MN procedures for the commands SHALM and AL_MN as follows:

DO_SHALM



```
procedure DO_SHALM (signal SHARLM: out std_logic) is
begin
  SHARLM <= '0';
  wait for FIVE_SEC_PERIOD;
  SHARLM <= '1';
end DO_SHALM;
```

DO_AL_MN



```
procedure DO_AL_MN (SETTING: in string(1 to 2);
    signal MS_MIN, LS_MIN : in std_logic_vector(3 downto 0);
    signal SHARLM : out std_logic;
    signal MINS: out std_logic) is

  -- convert character setting to std_logic_vector
  variable MS_SET : std_logic_vector(3 downto 0) := CHAR2VEC(SETTING(1));
  variable LS_SET : std_logic_vector(3 downto 0) := CHAR2VEC(SETTING(2));

begin
  SHARLM <= '0';
  MINS <= '0';
  wait until ((MS_MIN = MS_SET) and (LS_MIN = LS_SET)) for MAX_MIN_SET;
  assert ((MS_MIN = MS_SET) and (LS_MIN = LS_SET))
     report "timeout on alarm minute set to " & SETTING
     severity ERROR;
  SHARLM <= '1';
  MINS <= '1';
end DO_AL_MN;
```

We cannot predict what the alarm will be when the `DO_AL_MN` subprogram is called. Therefore, we must monitor the Alarm Clock display digits until they reach the required values, using a `wait until` statement. This requires the display digits to be passed into the procedure as `SIGNAL` class input parameters.

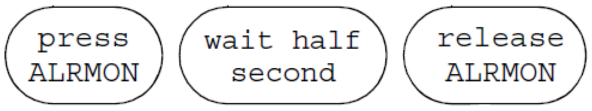
However, if there is a problem with the design or an invalid command like `AL_MN 99` is entered, the digits may never reach the required values. Therefore, the procedure must incorporate a “timeout.” We add a `for` clause to the `wait` statement and an `assert` to achieve this:

```
wait until <output = <value> for <time period>;
assert <output> = <value>
   report “timeout…” severity error;
```

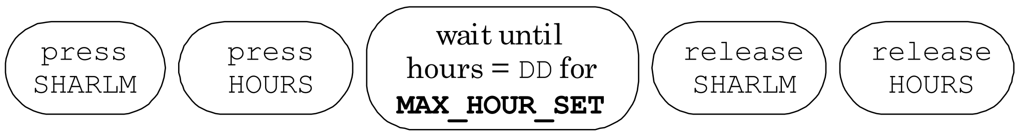
This will wait until the condition is true, up to the timeout value. We then check the condition; if it is false, then the condition “timed out” and we report this as an error. The timeout values are declared as constants in the package.

2. Complete the DO_ALTOG, DO_AL_HR, DO_TI_MN, DO_TI_HR and DO_TIMED procedures as follows:

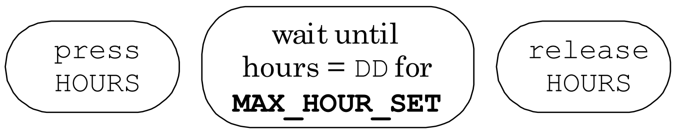
DO_ALTOG



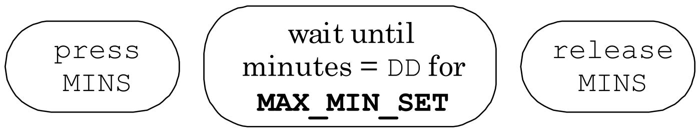
DO_AL_HR



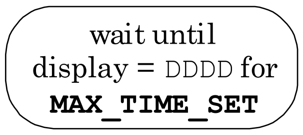
DO_TI_HR



DO_TI_MN



DO_TIMED



Testbench

A cut-down version of the testbench is given in the file t_ac_script.vhd. This only supports the DO_SHALM and DO_TI_MN procedures.

1. Edit the COMMAND_INPUT process in t_ac_script.vhd to add support for the other commands.

The trace file is created in the process `TRACE_OUTPUT`. This captures the four digits of the Alarm Clock display, the current simulation time, and the status of the Alarm. For readability, the display digits should be written as characters (e.g., `23:59`) instead of vector values.

- Create a function `BCD2CHAR` to convert the output digits from vector to character format.
- Simulate the testbench using the command file `command.txt` provided. Compare the output file created, `trace.txt`, with the expected result file `gold.txt`. You can use operating system commands like `diff` in Linux to do this.
- If your files are not identical, debug your design to find and fix the error. Note that you might get minor differences in the output, particularly in the values of outputs sampled before the reset.

On Completion of This Exercise

Congratulations! You have completed the Alarm Clock Design.

End of Lab

Module 22: Gate-Level Simulation

There are no labs for this module

Module 23: Application of Configuration

There are no labs for this module

Module 24: Design Organization and Management

There are no labs for this module