Thermostat Control System 2.0

GROUP 3

SECOND ASSIGNMENT

MEMBERS:

Are Emanuele

Bergonzi Giuseppe

Cellesi Giulio

Mascia Lorenzo

Miccio Carlo Edoardo

Usai Giovanni

PERIOD: MAY 15- JUNE 15

INDEX

INTRODUCTION 6

1. ASSIGNMENT

1.1 First Assignment 7

1.2 Second Assignment 10

2. OLD MODULE

2.1 General System Description 15

2.2 Finite State Machine 17

2.2.1 Description 17

2.2.2 State Transition 18

2.3 Start Driver 19

2.4 Sub_Sum_Driver 19

2.5 Window_Open_Driver 20

2.6 Mux_Clr, Mux_T, Timer 20

2.7 Mux_ID, RT_Driver 21

2.8 Delta_T_Driver 22

2.9 Seven_Seg_Driver 22

2.10 Led_Driver 23

3. IMPLEMENTATION

3.1 Demo 24

3.2 Leds Used 30

3.3 Buttons Used 30

3.4 Switches Used 31

3.5 7-Segment Display 31

3.6 Input Resources 32

3.7 Output Resources 32

3.8 Expected behavior 32

4. TOP THERMOSTAT

4.1 Top Thermostat Datapath 33

4.2 Top Thermostat Description 36

4.3 Top Thermostat Testbench 37

4.3.1 Post Synthesis Functional Simulation 39
4.3.2 Post Synthesis Timing Simulation 40
4.3.3 Post Implementation Functional Simulation 41
4.3.4 Post Implementation Timing Simulation 42
5. SYSTEM CORE
5.1 System Core Datapath 43
5.2 System Core Description 43
5.3 System Core Testbench 45
5.4 Manor 46
5.4.1 Manor Datapath 46
5.4.2 Manor Description 46
5.4.3 Manor Testbench 47
5.4.4 Fsm Start Driver 47
5.4.5 Fsm Start Driver Table 48
5.4.6 Fsm Start Driver Description 48
5.4.7 Fsm Start Driver Testbench 49
5.5 Wine Cellar 49
5.5.1 Cellar Datapath 49
5.5.2 Cellar Description 49
5.5.3 Cellar Testbench 50
5.5.4 Fsm Windows Close 51
5.5.5 Fsm Windows Close Table 51
5.5.6 Fsm Windows Close Description 52
5.5.7 Fsm Windows Close Testbench 53
5.5.8 Top Alarm Datapath 53
5.5.9 Top Alarm Description 54
5.5.10 Top Alarm Testbench 54
5.5.11 Fsm Alarm 55
5.5.12 Fsm Alarm Table 55
5.5.13 Fsm Alarm Description 56
5.5.14 Windows Open Driver Description 57
5.5.15 Windows Open Driver Testbench 57

5.6 Selector Sys Core 58
5.6.1 Selector Datapath 58
5.6.2 Selector Description 58
5.6.3 Selector Testbench 59
5.7 Var Driver 60
5.7.1 Var Driver Datapath 60
5.7.2 Var Driver Description 61
5.7.3 Var Driver Testbench 62
5.7.4 FsmVAR 63
5.7.5 FsmVAR Table 63
5.7.6 FsmVAR Description 64
5.7.7 FsmVAR Testbench 64
5.7.8 FsmCONF 65
5.7.9 FsmCONF Table 65
5.7.10 FsmCONF Description 65
5.7.11 FsmCONF Testbench 66
5.8 Display Led Driver 66
5.8.1 Display Led Driver Datapath 66
5.8.2 Display Led Driver Description 67
5.8.3 Display Led Driver Testbench 67
5.9 Led Driver 68
5.9.1 Led Driver Description 68
5.9.2 Led Driver Testbench 68
5.10 Orologio 68
5.10.1 Orologio Description 68
5.10.2 Orologio Testbench 68
5.11 Temp Hour 69
5.11.1 Temp Hour Description 69
5.11.2 Temp Hour Testbench 69
6. FIFO
6.1 Fifo Description 71
7. UART
7.1 The *top_log_screen* Module 72

7.2 The *GetCharacter* Module 74
7.2.1 The RomParam Module 76
7.2.2 The DecodeMessage Module 76
7.2.3 Testbench 78
7.3 The fsm_logScreenConnection Module 79
7.3.1 Fsm_logScreenConnection Tables 80
7.3.2 Testbench 84
8. SPI
8.1 Introduction 86
8.1.1 SPI Communication Protocol 86
8.2 Parametrization and Slave Selector 86
8.2.1 SPI Master Module 86
8.2.2 SPI Slave Module 87
8.2.3 Slave Selector 88
8.3 SPI Interface for Buttons (Interface_master_slave_buttons) 88
8.3.1 Testbench 89
8.4 SPI Interface FIFO-Display (Top_spi_master_Interface.v) 89
8.4.1 Master FIFO Display 90
8.4.1.1 FMS_Master_FIFO 92
8.4.1.2 FSM Master FIFO Tables 92
8.4.1.3 Testbench 97
8.4.1.4 Datapath Master-FIFO 99
8.4.1.5 Datapath Testbench 100
8.4.2 Top_Slave_FIFO 101
8.4.2.1 Slave_FIFO 102
8.4.2.2 FSM_Slave 102
8.4.2.3 FSM_Slave Tables 103
8.4.2.4 Testbench 105
8.4.2.5 Dp_slave_FIFO 105
8.4.2.6 Testbench General (including Dp_slave_FIFO) 106
8.4.3 Top_SPI_Master_Interface 107
8.4.3.1 FSM Rele 107
8.4.3.2 FSM Rele Tables 107
8.4.3.3 FSM Rele Testbench 108
8.4.4 Top_SPI_Master_Interface Testbench 109

9. COMMS DISPATCHER

9.1 Comms Datapath 114

9.2 Comms Description 114

10. ERRORS AND WARNINGS

10.1 Errors And Warnings 115

11. SYSTEM ANALYSIS

11.1 Performance And Different Implementation's Settings Analysis 116

11.2 Default : Timing Summary 116

11.3 Default : Device Design 117

11.4 Default : Utilization 117

11.5 Flow_areaOptimized_high : Timing Summary 118

11.6 Flow_areaOptimized_high : Device Design 119

11.7 Flow_areaOptimized_high : Utilization 119

11.8 Flow_perfOptimized_high : Timing Summary 120

11.9 Flow_perfOptimized_high : Device Design 121

11.10 Flow_perfOptimized_high : Utilization 121

11.11 Highest frequency for smallest positive slack 123

12. TASKS

12.1 Tasks Structure Explanation 124

12.2 Emanuele Are's Tasks 124

12.3 Giuseppe Bergonzi's Tasks 124

12.4 Giulio Cellesi's Tasks 125

12.5 Lorenzo Mascia's Tasks 125

12.6 Carlo Edoardo Miccio's Tasks 125

12.7 Giovanni Usai's Tasks 126

12.8 Deadlines 126

12.9 Collaboration 126

INTRODUCTION

Welcome to the introduction of our project! This is the second part of the thermostat, in the report we will show the changes or additions compared to the first part. In order to simplify the work, we have also added the descriptions of the previous modules to the report as some parts remained untouched (for the system core). In addition, it will be shown how SPI and Uart were implemented.
You will need a Basys 3 board and to download puTTY.

To test on basys 3 board we raccommand the follow values:

- QTY(200) TU(1000000) for every debouncer (Top_module_Manor and Top_module_Cellar)
- QTY(10) TU(34'd100000000) for the Timer (Top_module_Manor and Top_module_Cellar)
- stab(30) t_base(34'd99999999) for the contasec (Top_module_Cellar> Top_Alarm)
- QTY(5) TU(10000000) for the clock counter (Led_Driver)
- stab(10) t_base(9999999) for the contasec (Orologio) and sx_max(6)dx_max(3) for the orologio(system core)
- QTY(5) TU(100000) for the clock counter (7 seg))

To test on testbench we raccommand the follow values (system core):

- QTY(2) TU(10) for every debouncer (Top_module_Manor and Top_module_Cellar)
- QTY(2) TU(34'd10) for the Timer (Top_module_Manor and Top_module_Cellar)
- stab(1) t_base(34'd50) for the contasec (Top_module_Cellar> Top_Alarm)
- QTY(5) TU(10) for the clock counter (Led_Driver)
- stab(1) t_base(50) for the contasec (Orologio)
- QTY(5) TU(3) for the clock counter (7 seg)

Use (1) (1000) for all of these values for the Top_Thermostat Testbench (7 seg stay (5)(3)).

The global CLOCK and the system RESET represented in all the datapath with (* *) for design simplification.

Color Used Legend:

- input (dark green 2) + Bold
- state (dark orange 1) + Bold
- output (dark red 1) + Bold
- link to another module or external link (gray) + Bold
- link to figure/table (dark cornflower blue)
- figure and table's caption (dark blue 2)

ASSIGNMENT

1.1) First Assignment

Design, simulate, implement and deploy on the BASYS3 a system that simulates the behaviour of a house thermostat. The system must behave as follows:

1) There are always three temperatures to consider: the desired temperature, the current room temperature and the exterior temperature.
    a) The initial room temperature is 25°C.
    b) The user will set the desired temperature through 5 switches. This temperature can be chosen between 0-31°C. Once configured, the selection needs to be confirmed pushing the CONF push button.
    c) As for the external temperature, 2 additional switches need to be configured to notify the system if outside is summer/winter and day/night. This selection influences the system as follows:

Switch season	Switch time	Exterior temperature
Summer	Day	40° C
	Night	20° C
Winter	Day	10° C
	Night	0° C

Table 1-Exterior Temperature

2) Despite the desired temperature is set during the initial configuration, the user can change it during system operation, using the INC push button and the DEC push button to perform a +1/-1 °C, respectively.

3) One additional switch will be used to determine if the windows are open or closed. During system execution, this has a direct impact on the current room temperature:
    a) If room temperature > exterior temperature → -1°C per minute
    b) If room temperature < exterior temperature → +1°C per minute

This condition is expected to influence the automatic behaviour of the thermostat control system: it is expected the window is open, the thermostat control system will remain on, but will not automatically adjust the temperature.

4) When on and operative, the thermostat control system will automatically control the air conditioning/heat pump, with a direct effect on the room temperature of:
   a) -2/+2°C per minute, respectively, if the room temperature more than +6°C/-6°C with respect to the desired one.
   b) -2/+2°C every two minutes, respectively, if the room temperature within (+5 to +2)°C/(-5 to -2)°C with respect to the desired one.
   c) Nothing in all other ranges.

   The current state will be shown through the LEDs:
   d) Heat: all 16 LEDs are blinking with a frequency of 1 second (0.5 seconds ON, 0.5 seconds OFF).
   e) Cold: all the 16 LEDs are blinking with an odd/even pattern (when the odd LEDs are ON, even ones are OFF and vice versa).
   f) OFF: all 16 LEDs are off.

5) During system execution, the 7-segment display will show the desired temperature on the two leftmost digits, and the current room temperature in the two rightmost ones. Additionally, the point of the second digit will be turned ON if the window is open.

   The following tables summarize the I/O resources and the expected behaviour of the system:

Input resources	8 switches, 4 push buttons
Output resources	16 LEDs, 7-segment display

Table 2- I/O Resources 1

Action	Expected behaviour
Switch season	Switch to 1 means it is summer, otherwise it is winter
Switch time	Switch to 1 means it is day, otherwise it is night
Switch windows	Switch to 1 means the windows are open, otherwise they are closed
5 switches	Switches to configure the initial desired temperature
LEDs	It represents the state of the air conditioning/heat pump
CONF button pushed	Confirmation of the temperature selection
INC button pushed	+1°C in the desired temperature
DEC button pushed	-1°C in the desired temperature
RESET button pushed	Asynchronous system reset.

Table 3- Expected behavior 1

1.2) Second Assignment

The system described before is going to evolve as shown in Figure 1.

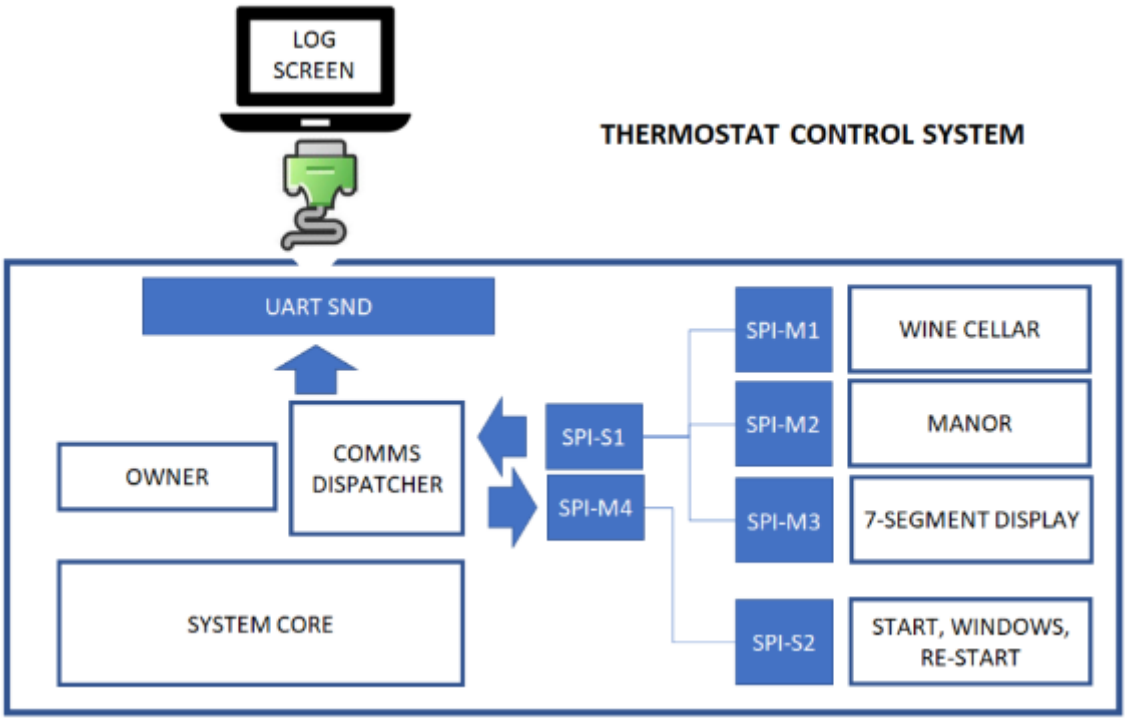


Figure 1-High-level block diagram of the system

You need to consider a house where the owner can manage 2 different areas: the wine cellar and the
main manor.

OWNER

The owner has the responsibility to configure the desired temperatures for the different areas (an optimal value for a wine cellar is 12-14°C). The completion of this operation start controlling the temperature in the wine cellar straight away; while the thermostat control system in the manor, despite being ready, will not be operative until the START switch is not set at 1 by the corresponding input peripheral. Any modification of the configuration protocol of the original system is up to you
and must be explained in the report. Note that:

- the owner will use one ID switch to operate on the different areas and configure their individual temperatures;
- all areas must be properly configured to initiate the system (it is up to you to define reasonable range of values, for wine cellars you can consider 12°-14° a good range of values).

Moreover, the owner decides also upon temperature modifications in the different areas by using the INC and DEC buttons in combination with the ID switches.

SYSTEM CORE

It contains quite all the original system logic to which a global time scanner (hours and minutes starting from 00:00) has to be added to the system. Such scanner starts as soon as the system is successfully configured and will be used to track at what time relevant events happen.

Moreover, it will be used to start dumping on the log screen all manor/cellar information (gathered from the dedicated peripherals) on the log screen every 6 hours. Information will have to be dumped on the screen by reading, in burst mode, the FIFOs contained in the cellar/manor peripherals one at a time. Once this happens the time scanner is restarted.

PERIPHERALS

Cellar and Manor are handled as two output peripherals storing all the requests of temperature modification. FIFO structures have to be used for storing information as follows: time of the request (hour, minute), type of request (1=INC, 0=DEC), current temperature value, summer/winter and day/night status.

The 7-segment display is handled as an output peripheral to convey information on the manor thermostat control system. If you feel any changes/additions is needed with respect to the original assignment, please make sure to justify that in the deliverable.

The START switch, the window open switches, one for the manor and one for the cellar, and the RE-START button are handled as input peripherals. For the manor thermostat control system to work the START switch should be set to 1. Any window open switch has the same impact on system operation as it did before. The RE-START button impacts on the wine cellar alarm as explained below.

COMMS DISPATCHER

Communications are managed by a comms dispatcher, which acts as an interface among the different involved modules. Peripherals are connected to the comms dispatcher via SPI. The log screen is connected to the comms dispatcher via UART. The owner and the system core exchange direct information with the comms dispatcher.

The following table summarizes the different communications and the protocols used. These are the minimum communications that must appear in your system, but if you decide you need more, they can be added if their addition is sufficiently justified in the report.

Block	Sends info to...	Receives info from...	Protocol
Manor	Comms dispatcher	Comms dispatcher	SPI
Cellar	Comms dispatcher	Comms dispatcher	SPI

7-segment Display	Comms dispatcher	Comms dispatcher	SPI
Start, Re-Start, Windows	Comms dispatcher	Comms dispatcher	SPI
Log Screen	-	Comms dispatcher	UART
Owner	Comms dispatcher	-	-
System Core	Comms dispatcher	Comms dispatcher	-

Table 4- Comms Dispatcher

The general flow of the system is the same as before, but please consider the following modifications/additions:

- LEDs are primarily used by the manor thermostat control system, but in one case. The cellar is supposed to stay always as close as possible to the chosen temperature range (+3/-3 with respect to the desired value). Out-of-range conditions for more than 30 minutes should trigger an alarm emulated by keeping all LEDs always on. To exit from this alarm condition the wine cellar must be physically inspected: the cellar WINDOW switch should be set to zero (if high) and the RE-START button pushed. The alarm must have priority wrt to any other LEDs usage.
- Relevant execution information, with proper messages, must be visualized on the log screen, e.g.:
  1) Configuration data,
  2) Manor thermostat control system started/paused,
  3) The manor/cellar window is open.

  It is your responsibility, and part of the evaluation, to make the log as complete and as self-explanatory as possible, and you should explain in the report which situations are reported and when. Please, consider using meaningful messages when printing the information on the screen, e.g., "04:33 Cellar alarm on – temperature YYYY" or "00: 11 Window open in the manor". These are just suggestions, please modify messages as best to fit your system needs and, in your specific case, make sure timing information is shown in the messages.

The following tables summarize the I/O resources and the expected behaviour of the system. Resource mapping is up to designers.

Input resources	11 switches, 5 push buttons
Output resources	16 LEDs, 7-segment display

Table 5- I/O Resources 2

Action	Expected behaviour
Switch season	Switch to 1 means it is summer, otherwise it is winter
Switch time	Switch to 1 means it is day, otherwise it is night
Manor/Cellar Switch windows	Switch to 1 means the windows are open, otherwise they are closed
5 switches	Switches to configure the initial desired temperature
START switch	When set to 1 starts the thermostat control system of the main manor
ID switch	[1] Manor [0] Wine cellar
LEDs	They typically represent the state of the air conditioning/heat pump of the main manor, unless there is an alarm to be handled in the wine cellar
CONF button pushed	Confirmation of the temperature selection
INC button pushed	+1°C in the desired temperature
DEC button pushed	-1°C in the desired temperature
RESET button pushed	Asynchronous system reset.
RE-START button pushed	If the cellar window is closed, when pushed the wine cellar alarm condition is terminated

Table 6- Expected behavior 2

Important notes:

- The system MUST work with the BASYS3 reference frequency and present an asynchronous reset.
- When needed, and not specified otherwise, it is up to you to rescale the timing in a way that during the demonstration it will be possible to see the entire system execution.
- Remember that SPI follows a master-slave philosophy, so the slave cannot send information by itself: it has to receive a request from the master to send any information back. You need to have this in mind to define (and discuss in the report) a correct protocol for

your different peripherals [i.e., checking each peripheral every X seconds or checking them when a particular event has happened].

- When designing the comms dispatcher, please consider that it can only make decisions regarding communications; every other decision must be made by the system core. Moreover, note that communications can happen in parallel, provided that the exchanged messages are not blocking for each other.
- For every communication in your system, you need to define (and justify in the report) both the configuration of your communication protocols (how is the communication started/handled/completed) and the messages that are exchanged (sematic of the exchanged information).
- Please, remember that all the different parts of the proposed system must be validated through simulation. Remember also that the system MUST be simulated (behavioural, post-synthesis, post-implementation) before being deployed on the board, and that both functional and timing simulations must be completed. If, for length reasons, you are not able to complete the timing simulations (provided that you explain what you have tried to carry out and why that did not work) you can avoid reporting them in detail. Additionally, parametric solutions are not only advised but compulsory: if the system is not designed in a parametric fashion, the solution will not be the most adequate and therefore it will not receive the maximal qualification, even if it works.

OLD MODULES

2.1) General System Description

The general functionality of the system is mainly based on the interconnection between both **combinatorial** and **sequential** modules in communication with a single **Finite State Machine**, this implementation approach is done based on the nonlinearity and the logical operation of the system.

The main behaviour of the system depends on **3 different temperatures** values:

- The desired temperature.
- The current room temperature. (Initially is **25°C**)
- The exterior temperature.

The system user has the choice to select, set, his **Desired Temperature**, in **0°/31°** range, using **5 switches**. The desired temperature cannot be taken into consideration until a **Confirmation Button** is pressed.

For the **External Temperature** the system is provided with 2 additional switches to represent the combination of certain criteria that follows the **Season** (Winter/Summer) and **Time** (Night/Day).

Depending on these combinations the system is influenced as follow:

- **Summer/Day** : External temperature = **40°C**
- **Summer/Night** : External temperature = **20°C**
- **Winter/Day :** External temperature = **10°C**
- **Winter/Night** : External temperature = **0°C**

The **Window** status it whether is **Open** or **Closed** is one of the main system crucial conditions that affects the system functionality, **Room Temperature** in particular, and here is reported its impact:

- if Room Temperature **>** Exterior Temperature the temperature must go down by 1°C every 1 minute.
- if Room Temperature **<** Exterior Temperature the temperature must go up by 1°C every 1 minute.

NB: This condition is valid **only** when the **Window** is **Open.**

When **ON** and **Operative**, the system will automatically control the air conditioning and heating pump to maintain the **Room Temperature** as requested by the specifications.

Here listed the different requested effects:

- **-2°C/+2°C** per **minute**, respectively, if the room temperature is more than **+6°C/-6°C** compared to the Desired Temperature.
- **-2°C/+2°C** every **2 minutes**, respectively, if the room temperature is within **(+5°C to +2°C)/(-5°C to -2°C)** compared to the Desired Temperature.
- Otherwise, no change in temperature.

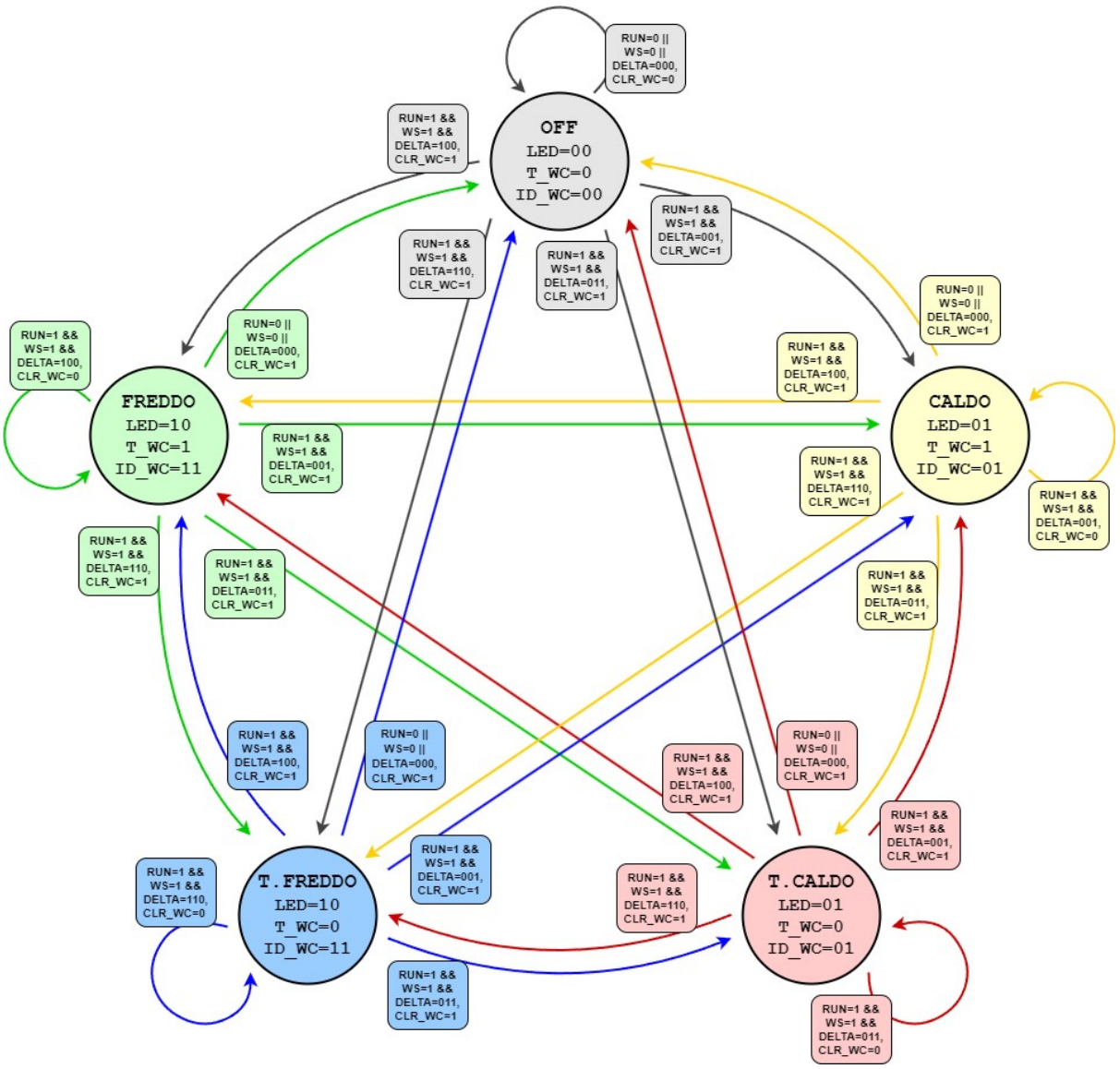
NB: This condition is valid **only** when the **Window** is **Closed.**

For the user to be able to change the desired temperature, even if it is already set at the initial configuration, there are **INCR** button and **DECR** button to **increase** or **decrease** by one 1 respectively.

The system must show, always during its operation, the current state turning ON and OFF the BASYS3 **LEDs** as follow:

- **Heat**: all the **16 LEDs** are blinking with a frequency of 1 second **(0.5 seconds on, 0.5 seconds off).**
- **Cold:** all the **16 LEDs** are blinking with an **odd/even pattern** (when the odd LEDs are ON the even once are OFF and vice versa).

2.2) Finite State Machine



2.2.1) Description

The finite state machine operation is based on confirmation signal (RUN) from the user which will put in action the controlled ventilation of the room, on the fact that the window is whether closed or open (WS) and on the difference between the desired temperature and measured room temperature (so based on DELTA value).

Here explained the 5 different states of the FSM.

States:

OFF: In this state the temperature controller is not operating and the conditions to stay in this state are:

- no user confirmation.
- open window.
- the temperature range is satisfied.

CALDO: In this state the temperature goes up, gradually, by +2° in 2 minutes time and the air flows gradually as well, depending on the temperature variation in time.

TURBO CALDO: In this state the temperature goes up by +2° in 1 minute time and the air flows more intensely since the temperature variation is higher in time.

FREDDO: Same as what happens in CALDO state except for the variation in temperature is -2° over 2 minutes time instead of +2°.

TURBO FREDDO: Same as what happens in TURBO CALDO state except for the variation in temperature is -2° over 1 minute time instead of +2°.

2.2.2) States Transitions

The initial state is OFF and the system stays in this state as soon as its two main inputs (RUN, WS), prioritized over DELTA, are 0.

Once the system detects that the window is closed, **WS=1**, and the confirmation button is pressed, **RUN=1**, it changes its state following the detected temperature difference.

Following the detected temperature difference, the system changes its state as follow:

- **DELTA = 001**: From any state to **CALDO**.
- **DELTA = 011**: From any state to **TURBO CALDO**. • **DELTA = 100**: From any state to **FREDDO.**
- **DELTA = 110**: From any state to **TURBO FREDDO**.

Going back to the initial state (OFF) from the other 4 states depends on the asserting of the 2 main signals (RUN=0, WS=0).

The system finite state machine, as we can see in the table and in the diagram, is hybrid since we do have ¾ of the outputs following **Moore** approach and ¼ following the **Mealy** one.

Moore outputs :

LED: Responsible for turning off and on the LEDs.

- LED = 00: LEDs are off.
- LED = 01: turning on the LEDs asynchronously.
- LED = 10: turning on the LEDs in an odd and even way.

T_WC: Responsible for 1- and 2-minutes managing.

- T_WC = 0: corresponds to 1 minute.
- T_WC = 1: corresponds to 2 minutes.

ID_WC: 2°C Incrementing or Decrementing.

- ID_WC = 00: no effect.
- ID_WC = 01: incrementing.
- ID_WC = 11: decrementing.

The **ID_WC** bits are going to be the input for a **MUX_ID** in the Datapath and there an additional **0** is going to be concatenated to be able to increment or decrement by 2. With this logic will have: **010** and **110** with the MSB representing the sign (**0=+** and **1=-**) and the other two bits represent the absolute number of **2**.

The **CLR_WC** is the only signal that represents the Mealy output. It is important since we need to reset the **TIMER** to zero every time, we change the state so the count of the 1 and 2 minutes works properly.

2.3) START DRIVER

Input: clk (Clock), rst (Reset), C_DB (CONF Debounced)

Output: RUN

C_DB is the debounced signal of button **CONF**, the behaviour mimics the functionality of a relè (that can only turn on devices). By default, the system is in **IDLE** state and the output signal **RUN** is low, when **C_DB** is pressed the module moves to **WAIT** state where it stays only for a clock cycle before changing state and then it moves to **RUNNING** one.

2.4) SUB_SUM_DRIVER

Input: clk (Clock), rst (Reset), DTR [5] (Desired Temperature Registered), DECR_DB (Decrease Debounced), INC_DB (Increase Debounced), RUN

Output: DTF [5] (Desired Temperature Final)

This module is an adder with enable. The input signal **DTR** is the output of the **REG_DT** register, it represents the user set desired temperature. The module ignores **DECR_DB** and **INC_DB** until **RUN** is high. When RUN turns high the module modifies **DTF** depending on **DECR_DB/INCR_DB**. Inside this module there is a form of retroaction, in particular, as soon as RUN turns high the internal adder register will contain the value of

the **REG_DT** register, after that for each cycle it will contain its last output, possibly increased or decreased.

2.5) WINDOW_OPEN_DRIVER

Input: ETR [6] (External Temperature Registered), RTR [6] (Room Temperature Registered)

Output: T_WO (Time Window Open), ID_WO [3] (Increment/Decrement Window Open), CLR_WO (Clear Window Open)

The WINDOW_OPEN_DRIVER module compares ETR with RTR to properly drive the RT_DRIVER when windows are open.

ID_WO can assume 3 values:

- ID_WO=000 if ETR equals RTR
- ID_WO=101 if RTR bigger than ETR
- ID_WO=001 if ETR bigger than RTR

T_WO always equals 0 because it's used by RT_DRIVER and it means that the timer will count to one minute.

CLR_WO goes high when the Room Temperature is more than the External Temperature and in the previous instant, the External was the highest one.

2.6) MUX_CLR, MUX_T, TIMER

MUX_CLR

Input: CLR_WO, CLR_WC

Output: CLR

MUX_T

Input: T_WO, T_WC

Output: TIME

TIMER

Input: TIME, CLR

Output: HIT

The TIMER module is quite simple (it's a timer with a clear signal), but it is interesting to see its interconnection with the "MUX_CLR" and "MUX_T" and, in general, the logic behind them.

First, we need to mark time for 1 or 2 minutes: this depends on the FSM if the windows are closed and on the "WINDOW_OPEN_DRIVER" otherwise (explained below).

Let's have a deep look at the MUX_CLR without caring about how the clear signals are generated (this is explained in FSM_WINDOWS_CLOSE and WINDOWS_OPEN_DRIVER section), their purpose is to have the time calculated properly when needed
(e.g. when it's needed to pass from a 2 minutes count to a 1 minute count, this happens when we pass from "normal mode" to "turbo mode" and vice-versa).

When TIME=0, the maximum threshold is one minute, when TIME=1 the maximum threshold is two minutes.

The other MUX, "MUX_T", receives the information related to the maximum threshold (1 or 2 minutes). HIT goes high only when the threshold is reached, otherwise it stays low. MUXs are driven by the WS signal.

2.7) MUX_ID, RT_DRIVER

MUX_ID

Input: ID_WO, {ID_WC [2], 1'B0}

Output: ID [3] (Increment/Decrement)

RT_DRIVER

Input: clk (Clock), rst (Reset), ID [3] (Increment Decrement), HIT

Output: RTR [6] (Room Temperature Registered)

MUX_ID has input signals from the FSM_WINDOW_CLOSE and WINDOW_OPEN_DRIVER, it's used to drive the RT_DRIVER when windows are closed or open. From the FSM comes a 2 bits signal, there's a concatenation with 1'b0 and then we get a 3 bits signal, from the other side we can see 3 bits
signal that comes without alterations from the WINDOW_OPEN_DRIVER.

That's because when we open the window there's the possibility to match the external temperature with the room temperature (in terms of signals when ETR=RTR) ... We wanted to respect thermodynamics.

RT_DRIVER provides the RTR signal, it performs sums and subtractions based on the input, ID, signals the meaning of which has already been explained in the previous modules.

2.8) DELTA_T_DRIVER

Input: DTF [5] (Desired Temperature Final), RTR [6] (Room Temperature Registered)
Output: DELTA [3]

This module is purely combinatorial: it's a comparator that compares RTR with DTF and detects the difference between them, consequently DELTA changes as follow:

- if the result is between -1 and +1 DELTA = 000.
- if the result is between -2 and -5 DELTA = 001.
- if the result is less than -6 DELTA = 011.
- if the result is between +2 and +5 DELTA = 100.
- if the result is more than +6 DELTA = 110.

Looking at FSM_WINDOWS_CLOSE, the 2nd bit has a specific meaning: if it's high and the absolute value is 6 degrees, the thermostat goes on "Turbo Mode", otherwise it goes to "Normal mode".

To make the combination make sense the MSB & the LSB are used except for when the DELTA=000 so we have:

- if the MSB=1 the system cools the room.
- if the LSB=1 the system heats the room.

2.9) SEVEN_SEG_DRIVER

Input: clk (Clock), rst (Reset), DTF [5] (Desired Temperature Final), WS (Window Switch), RTR [6] (Room Temperature Registered)
Output: AN3, AN2, AN1, AN0, CA, CB, CC, CD, CE, CF, CG, CW

This module is composed of two decoders, the Seven Segment Driver module from ESD and a brand-new part for the dot management (this depends on the value of WS, it will be on if the signal is high and vice-versa).

The Decoders are used because the old driver receives only digit-by-digit numbers, for example: to visualize the number 1234 it is needed to send them sequentially as 4 digits of 4 bits each and not as 11 bits (log2 of 1234 is 11), the decoders are used to perform this task since we do have a predefined value of DTF and RTR.

2.10) LED_DRIVER

Input: LED [2]

Output: LED_OUT [16]

This module is used to manage the LEDs blinking, LED comes from FSM and indicates when the system is refreshing or heating the room and when the system is off or the room temperature is in range. LED_OUT signal drives LEDs, each bit is related to each LED on the BASYS3. Let's see in a more detailed way the functionality of LED [2]:

- **LED=00** system off or room temperature in range, LEDs are off.
- **LED=01** all leds blinking.
- **LED=10** blinking with an even/odd pattern.

IMPLEMENTATION

3.1) Demo

This is the Demo of the whole system.

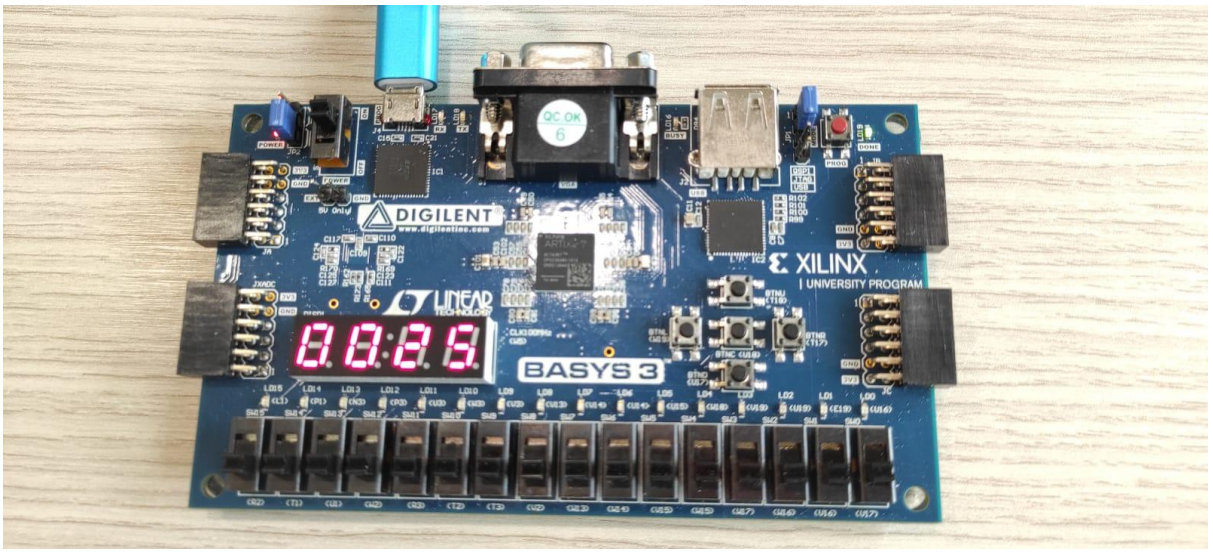


Figure 2-Demo Default

All the switches are down, so you are in the Wine Cellar with the default temperature of 25°C.

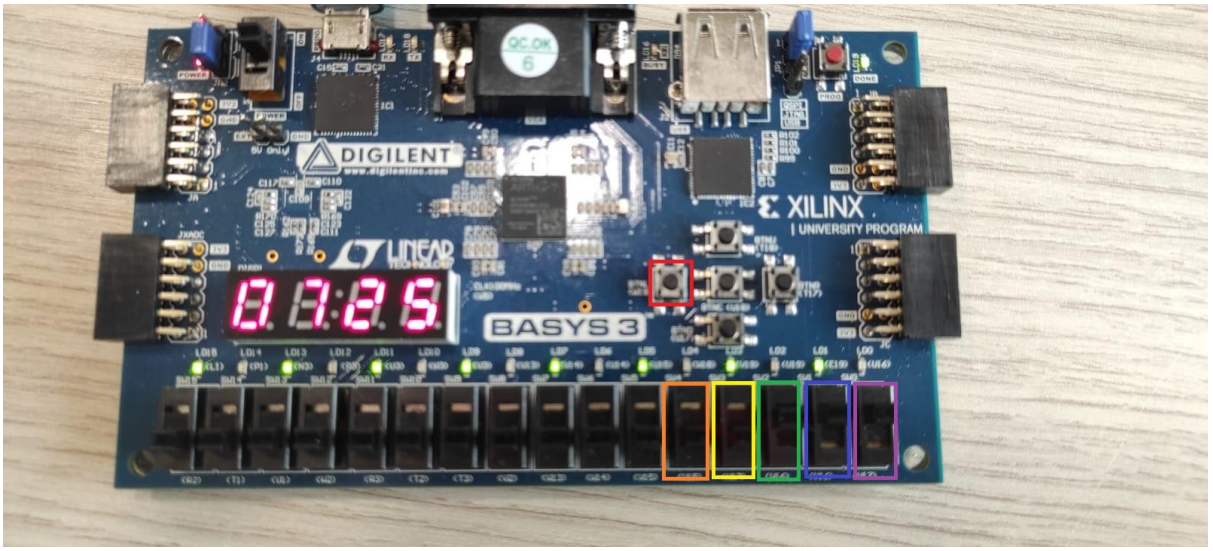


Figure 3-Cellar Conf

The next step is to choose the desired temperature with the 5 switches to the far right, **DT[0]** the value is 1 (purple in the figure 3), **DT[1]** the value is 2 (blue in the figure 3), **DT[2]** the value is 4 (green in the figure 3), **DT[3]** the value is 8 (yellow in the figure 3), **DT[4]** the value is 16 (orange in the figure 3) and press the **CONF** (red in the figure 3).

The Leds will move with the even-odd pattern because the room is cooling.

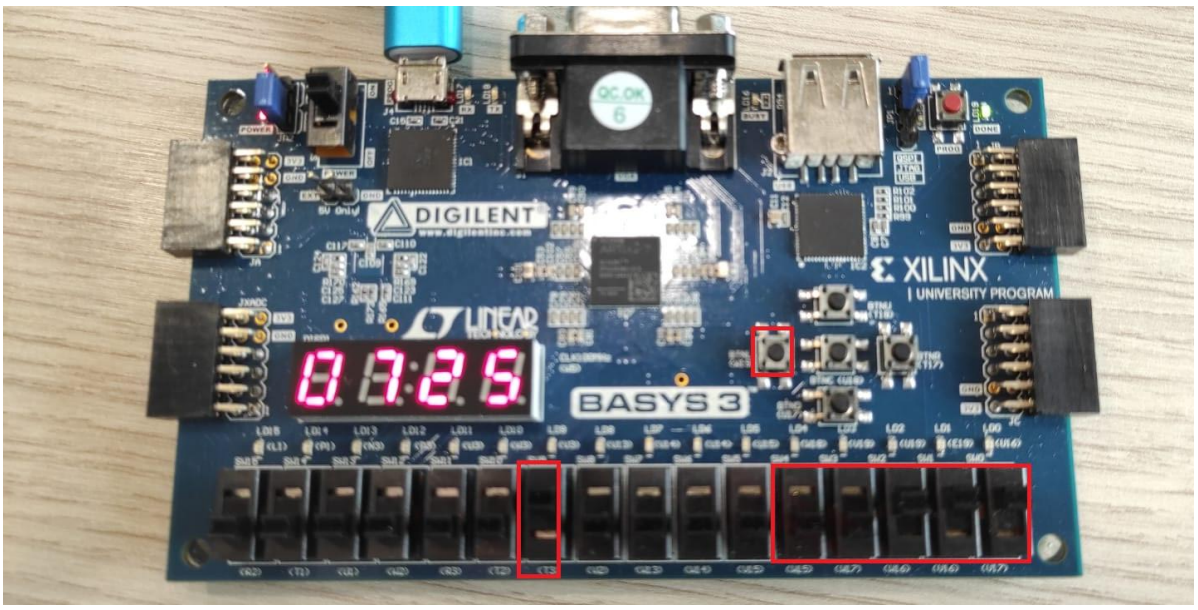


Figure 4-Manor Conf

After you have to raise **ID_SW** for change from Wine Cellar to Manor and repeat the same steps as previously seen (figure 3).

```
Manor: 0:00 System has been configured 25
```

At the end of these steps, on puTTY you'll see this message.

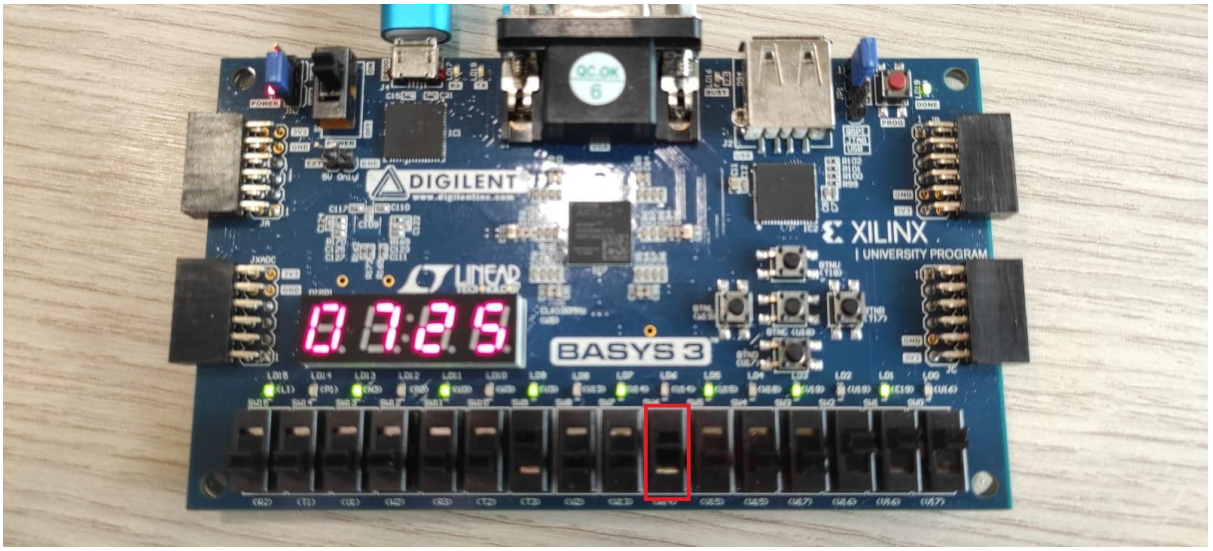


Figure 5-SW_START Demo

To put on/off the system you have to raise/lower the switch **SW_START**.

```
Manor: 0:02 System started! Temperature: 25
```

On puTTY you'll see this message when **SW_START** is raised.

```
Manor: 1:03 System stopped! Temperature: 01
```

On puTTY you'll see this message when **SW_START** is low.

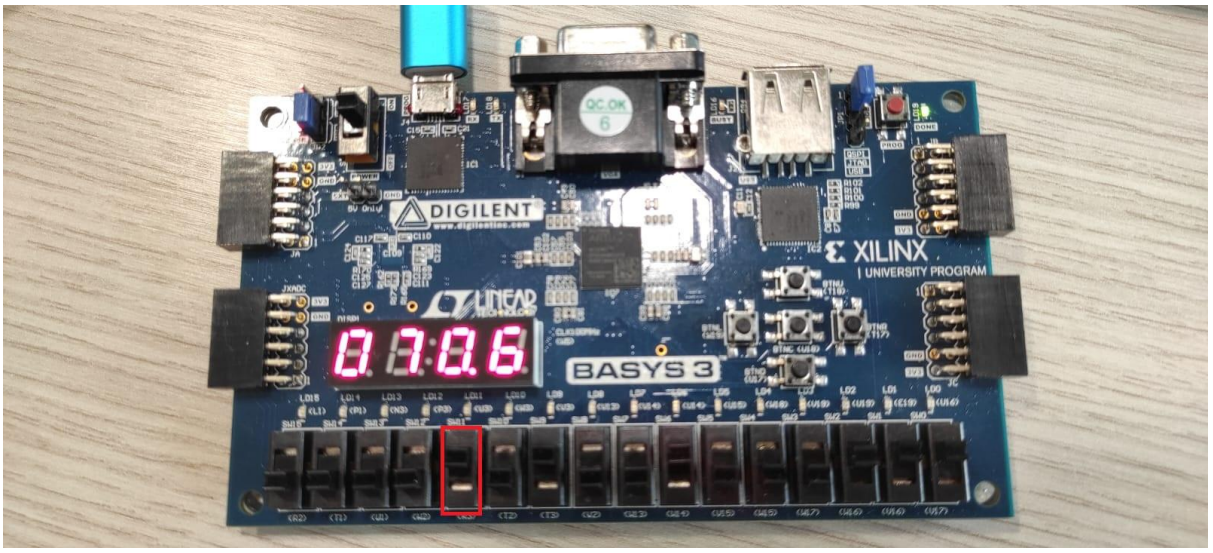


Figure 6-WS_MANOR

To open and close the window in Manor you have to raise/lower the switch **WS_MANOR**.

```
Manor: 5:03 Window is open! Temperature: 07
```

On puTTY you'll see this message when **WS_MANOR** is raised.

```
Manor: 2:00 Window is close! Temperature: 00
```

On puTTY you'll see this message when **WS_MANOR** is low.

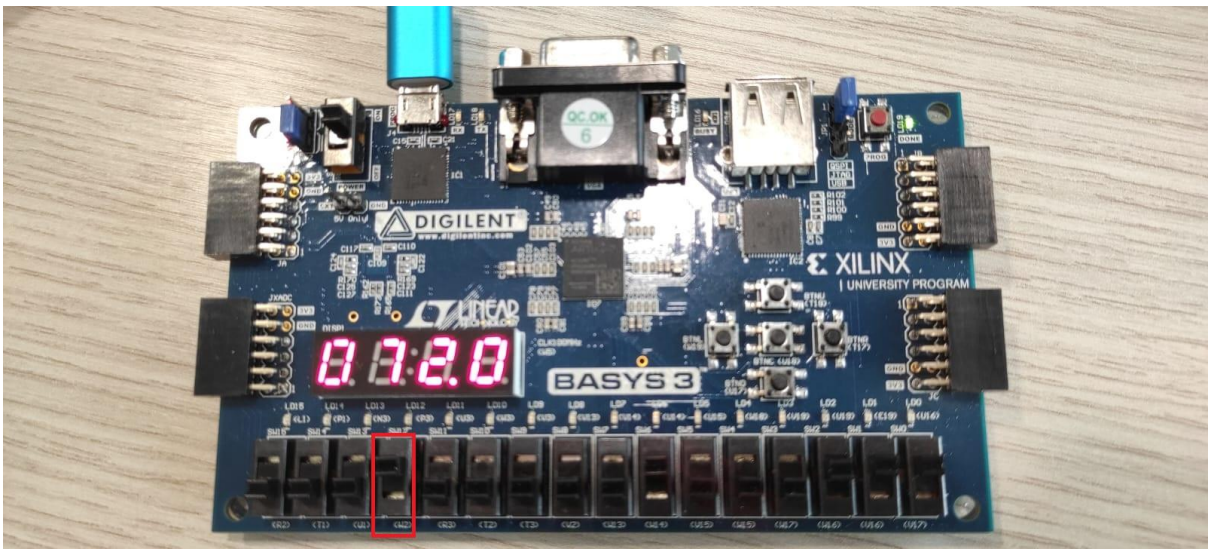


Figure 7-WS_CELLAR

To open and close the window in Wine Cellar you have to raise/lower the switch **WS_CELLAR**.

```
Wine Cellar: 1:03 Window is open! Temperature: 21
```

On puTTY you'll see this message when **WS_CELLAR** is raised.

```
Wine Cellar: 4:01 Window is close! Temperature: 19
```

On puTTY you'll see this message when **WS_CELLAR** is low.

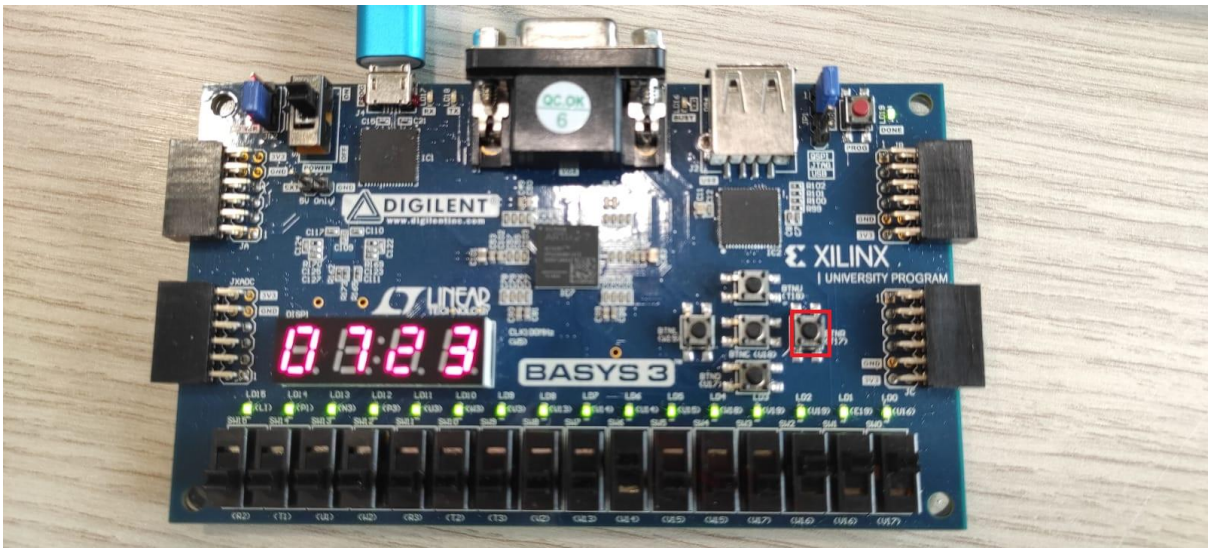


Figure 8-Alarm

Now the system is in Alarm (only in Wine Cellar), this happens when the difference between the relative temperature and the desired

temperature is greater than 3 for 30 minutes. All the leds are on and for resetting this case you have to close the window (**WS_CELLAR**) if it is open and press the **RESTART** button.

```
Wine Cellar: 2:02 Alarm started!!! Temperature: 23
```

On puTTY you'll see this message when the alarm is on.

```
Wine Cellar: 0:00 Alarm restarted! Temperature: 23
```

On puTTY you'll see this message when the alarm is restarted (so **WS_CELLAR** is closed and **RESTART** is pressed).

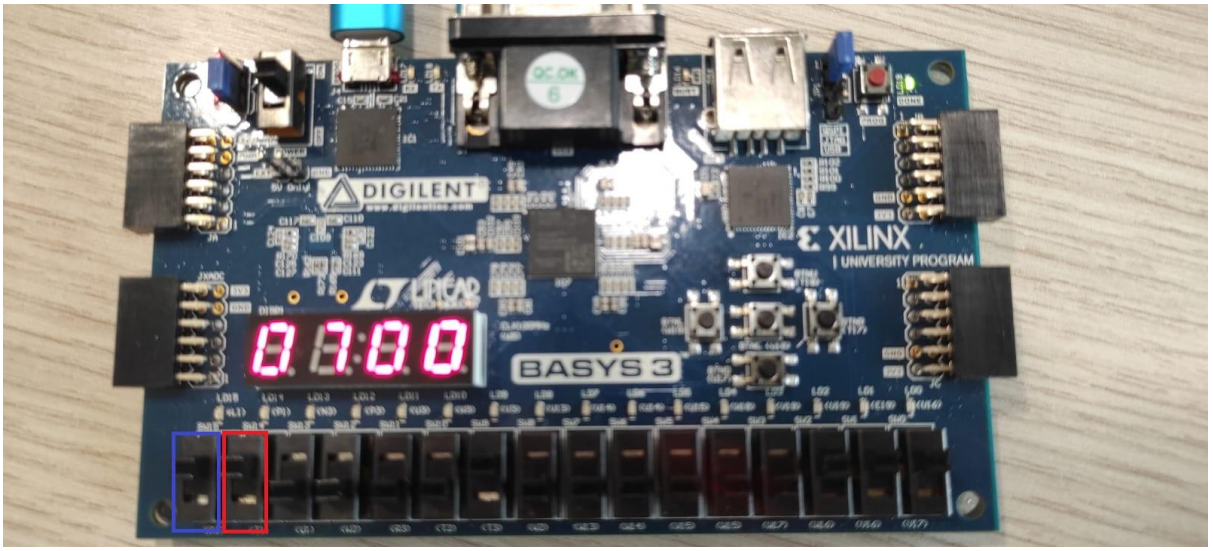


Figure 9-ST and SS

To change the season summer/winter you have to raise/lower the switch **SS** (blue in the figure).

To change the time day/night you have to raise/lower the switch **ST** (red in the figure).

```
Manor: 3:01 Is summer time! Temperature: 00
```

On puTTY you'll see this message when **SS** is raised.

```
Manor: 1:00 Is winter time! Temperature: 25
```

On puTTY you'll see this message when **SS** is low.

```
Manor: 3:00 Good morning! Temperature: 00
```

On puTTY you'll see this message when **ST** is raised.

```
Manor: 1:00 Good night! Temperature: 25
```

On puTTY you'll see this message when **ST** is low.

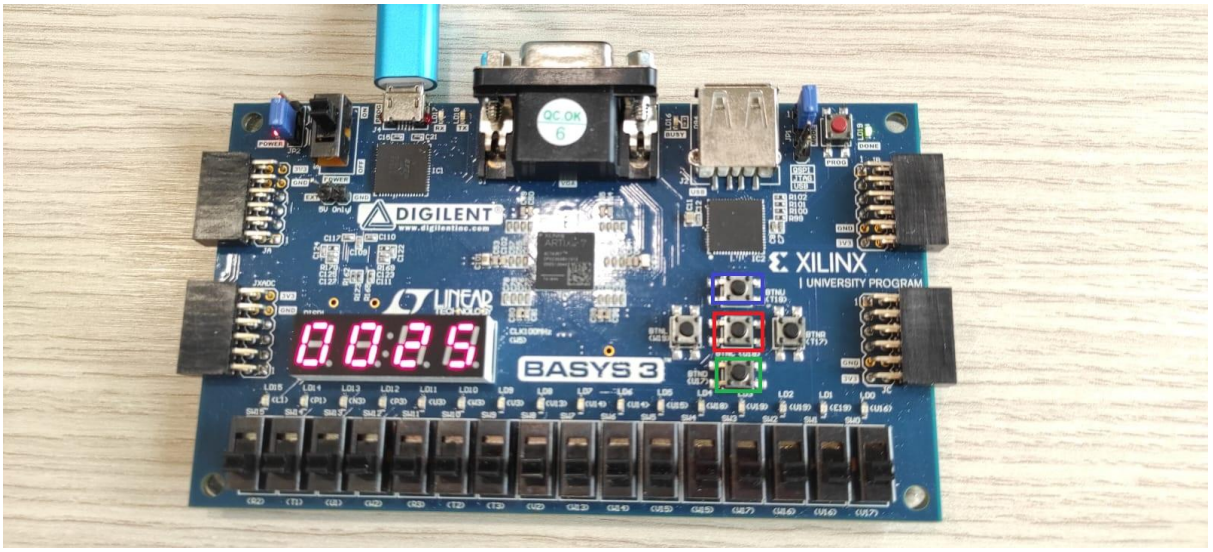


Figure 10-Other Buttons

To increment the desired temperature you have to press the **INCR** button (the blue in the figure 10).

To decrease the desired temperature you have to press the **DECR** button (the green in the figure 10).

To reset the system you have to press the **rst** button (the red in the figure 10).

Also, the Leds will blink because the room warms up.

3.2) Leds Used

NAME LED ON VIVADO	NAME LED ON BASYS
LED_OUT [0]	U16
LED_OUT [1]	E19
LED_OUT [2]	U19
LED_OUT [3]	V19
LED_OUT [4]	W18
LED_OUT [5]	U15
LED_OUT [6]	U14
LED_OUT [7]	V14
LED_OUT [8]	V13
LED_OUT [9]	V3
LED_OUT [10]	W3
LED_OUT [11]	U3
LED_OUT [12]	P3
LED_OUT [13]	N3
LED_OUT [14]	P1
LED_OUT [15]	L1

Table 7-Leds Used

3.3) Buttons Used

NAME BUTTON ON VIVADO	NAME BUTTON ON BASYS
CONF	W19
INCR	T18
DECR	U17
rst	U18
RESTART	T17

Table 8-Buttons Used

3.4) Switches Used

NAME SWITCH ON VIVADO	NAME SWITCH ON BASYS
DT [0]	V17
DT [1]	V16
DT [2]	W16
DT [3]	W17
DT [4]	W15
SW_START	W14
ID_SW	T3
WS_MANOR	R3
WS_CELLAR	W2
ST	T1
SS	R2

Table 9-Switches Used

3.5) 7- Segment Display

NAME ANODE ON VIVADO	NAME ANODE ON BASYS
CA	W7
CB	W6
CC	U8
CD	V8
CE	U5
CF	V5
CG	U7
CW	V7
AN0	U2
AN1	U4
AN2	V4
AN3	W4

Table 10-7 Seg Display

3.6) Input Resources

- 11 switches
- 5 push buttons

3.7) Output Resources

- 16 LEDs
- 7-segment display

3.8) Expected behavior

The expected behavior was previously shown in Table 6- Expected behavior 2

TOP THERMOSTAT

4.1) Top Thermostat Datapath

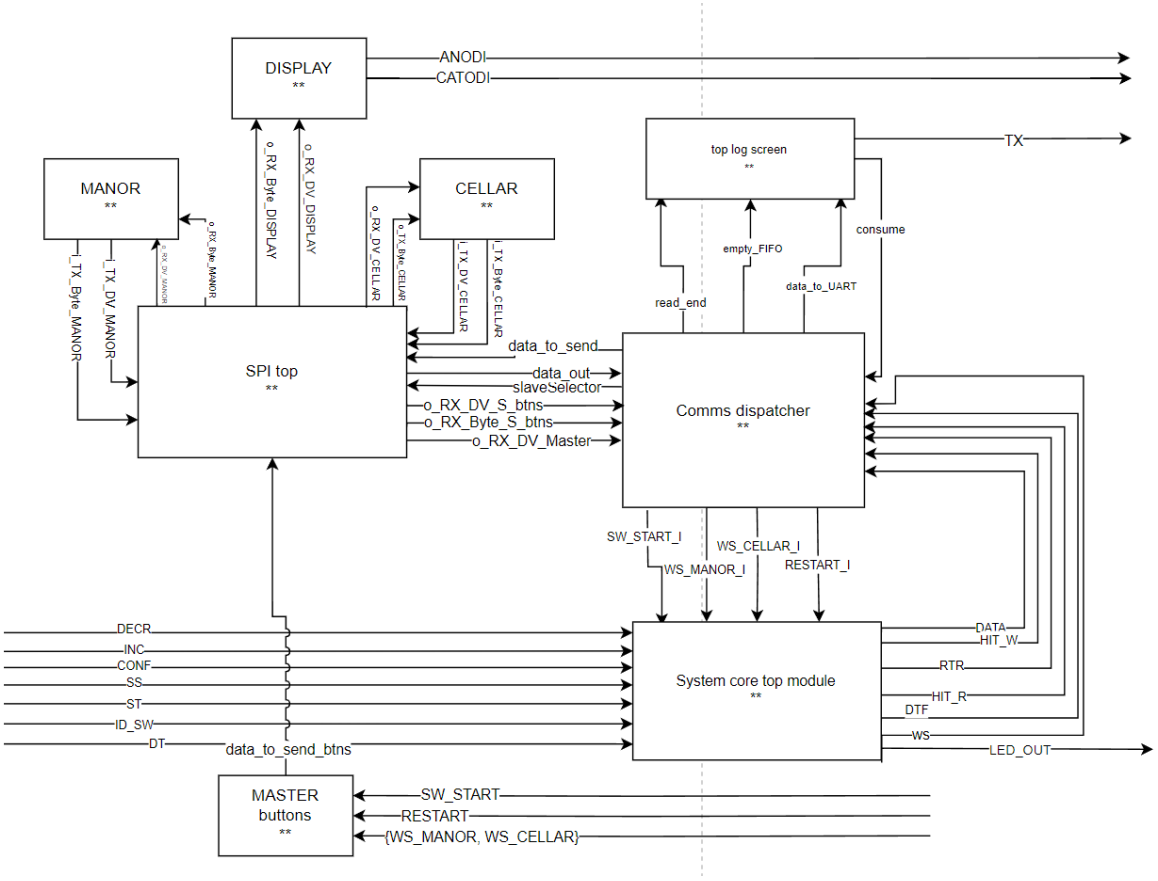


Figure 11-Top Thermostat Datapath

Top Thermostat execution flow

Initially the user see in 7SD the default temperature for both Manor and Cellar equals to 25. The first thing to do is to configure cellar and then manor.

Input managing

Some input like SW_START, WS_MANOR, WS_CELLAR and RESTART reaches the system through SPI system. Here we have a Master Interface which objective is to take the signals, create a packet and transmit the packet through the MOSI line. When o_RX_DV in respective SPI Slave (in Spi_top module) goes high the Slave_buttons in Comms Dispatcher take o_RX_Byte and send the single input to the system core.

Configuration

With ID_SW set to 0 we configure Cellar. Here we use DT switches to choose the desired temperature DTF and we have to press the CONF button. Pressing the CONF button with ID_SW equals to 0, will start the Cellar, entering into TOP_MODULE_CELLAR thanks to SELECTOR_SYST_CORE. Then we have to configure Manor, so rising ID_SW will be displayed the RTR temperature of the Manor (default equal to 25) and pressing CONF button the Manor system is configured and will start only after rising the SW_START, here the signal generated by SELECTOR_SYST_CORE (CONF_MAN) enters into VAR_DRIVER module. Here the FsmCONF signals the configuration of the system and outputs the corresponding OPCODE (that will be used to understand which phrase must be stored in peripherals and then send to UART). Now the system is configured.

Variations

For each input that produces a variation, the execution flow is more or less the same of CONF_MAN. With the difference that for other inputs like start, stop, summer, day ecc... the variation is defined by fsmVAR in VAR_DRIVER module. The behaviour is like FSM_CONF with the difference that the variation will be stored only if the system is already started, based on RUN signal(after the Cellar has been configured it is equal to 1). So every time there is a variation hit_C or hit_M signal will be rised, based on ID_SW (except for signals like SS and ST that are always related to Manor, as has been implemented in VAR_DRIVER). Variation of one of them rises hit_write signal, that will be used to write DATA into SPI. DATA is characterized by an OPCODE (based on the variation), that identifies the message, the id (where), that identifies if we are in Manor or Cellar, the hour, minutes and current temperature of Manor or Cellar (based on the id).

Variations-Alarm

One of the possible variations refers to the alarm. In particular if the difference between DTF and RTR exceed +3/-3 in Cellar the fsm_alarm goes to COUNTING state where a counter is enabled. If the exceed is maintained for more than 30 minutes the fsm_alarm goes to ALARM state and can came back to IDLE state if button restart is pressed and the cellar window is down. Both of this operation produce an hit_write that refers to cellar, and in the alarm state all the leds goes high (as implemented in fsm_modified).

Data storing

So we have understood that whenever a registered variation occurs hit_write signal rises by System Core. This signal goes to Master_fifo_display with DATA signal. Once data transmission to 7SD is finished, fsm_master_fifo goes to WRITE_REQUEST and then WRITE state to write the DATA into one of the FIFOs based on id (5th MSB bit). Here master_fifo_display, placed in Comms Dispatcher, communicate with SPI Master and send the data to SPI Slave (in SPI_top). Spi Slave after receiving the information from MOSI line, communicates the o_RX_Byte to Top_Slave, which objective (when a writing operation must be performed) is to store into the FIFO the 20 bit information that represents the message.

Data Reading

After that Cellar is configured, and RUN signal is high, 'Orologio' in System Core start to count. Once reached 6 hours a hit_read signal raises. This signal enables a reading operation from peripherals FIFOs. In particular fsm_master_fifo in master_fifo_display goes to READ_REQUEST and READ state for both the peripherals (Manor first and Cellar after) in order to read the information stored in FIFOs. Here we have that when a reading operation must be performed, Fsm_slave goes to LEG state and all the data from both FIFOs (once at time) are transmitted to MISO line. In this way the information goes to SPI Master and, every time a data is received from MISO line (as implemented in master_fifo_display) we compute an insertion into auxiliary FIFO.

Log Screen

When we read from Manor and Cellar FIFOs, a signal called read_end goes high. This signal is sent to Top_log_screen, which state machine (fsm_logScreenConnection) takes data from auxiliary FIFO rising consume signal. Here when UART can transmit (available = 1) and we did not reach the end of the message to transmit we start the transmission of the single character to UART. In particular, based on OPCODE of the data that has been read from auxiliary FIFO, one ROM address is indexed in order to select the message that must be sent. We use the inc signal to iterate over the selected message in ROM and send step by step the data to UART. In particular the message to be sent is created in DecodeMessage module, where id, hour, minutes and temperature are acquired from data coming from auxiliary FIFO. When done signal from GetCharacted module is high and reaches the fsm_logScreenConnection, the message has been completely sent to UART. Now if the auxiliary FIFO is empty it means that there are no more

data to be sent, so we can wait a new read_end, otherwise we restart the process sending the other data.

This was just a general description, to understand approximately how the system works. Everything is explained in a better way in the rest of this report.

4.2) Top Thermostat Description

Input:

- clk: Clock
- rst: Reset
- SW_START: When set to 1 starts the thermostat control system of the main manor
- RESTART: If the cellar window is closed, when pushed, the wine cellar alarm condition is terminated
- WS_CELLAR: Switch to 1 means the window is open, otherwise is closed
- DECR: -1°C in the desired temperature
- INCR: +1°C in the desired temperature
- CONF: Confirmation of the temperature selection
- ID_SW: If 1 Manor, 0 Wine cellar
- SS: If 1 set summer, 0 set winter
- ST: If 1 set day, 0 set night
- DT[5]: Switches to configure the initial desired temperature
- WS_MANOR: Switch to 1 means the window is open, otherwise is closed

Output:

- LED_OUT[16]: A signal for LED VALUE
- AN0,AN1,AN2,AN3,CA,CB,CC,CD,CE,CF,CG,CW: Contain the output of the 7 segment display
- TX: Output of the log screen

4.3) Top Thermostat Testbench

Figure 12-Top Thermostat Tesbench 1

This Testbench show the basic system function, it start with **ID_SW** = 0 (Cellar) and when **CONF** = 1 Cellar is configured, next **ID_SW** = 1, and **CONF** = 1 again, the Manor is configured, so the System Core send a "message" of System configured to SPI.

NB: The message "Manor: 0:00 Alarm started!!! Temperature: 00" is a default start message that is not displayed on screen.

For anyone of this variation hit write go high and it enable to write the output of System Core to the correspondent FIFO (based of where's bit)

When **SW_START** = 1 the Manor's system started, so it sent to SPI the message "System is started" (MANOR).

When **WS_MANOR** = 1 the Manor's Window is open, so it sends to SPI the message "Window is Open" (MANOR).

When **SS** = 1 the system is in the Summer season, so it sends to SPI the message "Is summer time", and next **SS** = 0, the system is in Winter season, so it sends to SPI the message "Is winter time" (MANOR).

When **WS_MANOR** = 0 the Manor's Window is closed, so it sends to SPI the message "Window is close!" (MANOR).

The next change is a **ID_SW** = 0, now the next values are routed to Cellar.

When **WS_CELLAR** = 1 the Cellar's Window is open, so it sends to SPI the message "Window is Open" (CELLAR).

When **WS_CELLAR** = 0 the Cellar's Window is closed, so it sends to SPI the message "Window is Close!" (CELLAR).

When **ST** = 1 the system is in day, so it sends to SPI the message "Good Morning", and Next **ST** = 0, the system is night, so it sends to SPI the message "Good Night" (MANOR).

Around 90 us in the tb, the active (alarm) goes high, so will send the message "Alarm started" to SPI, next, **restart** goes high to deactivate the alarm (before doing that we decrease the desired temperature) in fact we can see that active goes down, and this is shown to SPI by "Alarm restarted" messages.

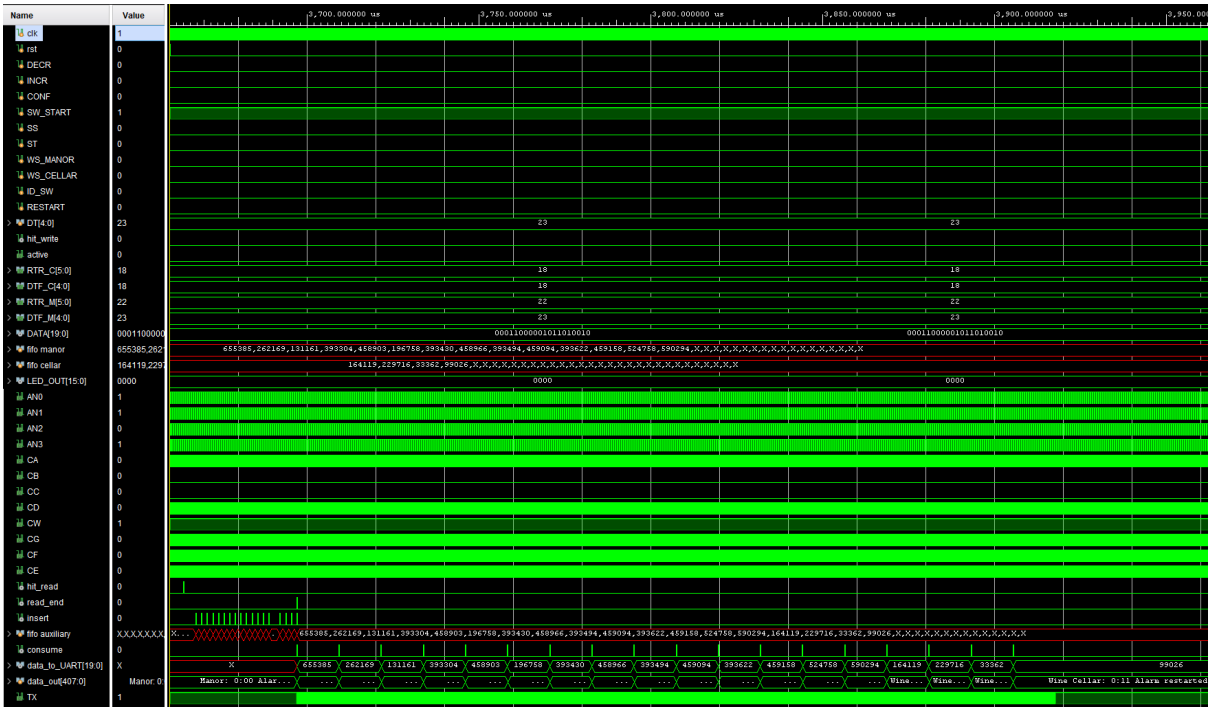


Figure 13-Top Thermostat tesbench 2

This testbench shows messages by Uart to screen, after 3.700us, because it is waiting for a hit_read high (every 6 Hour by Orologio, obviously the timer has been escalated to fit the functionality into the testbench)

4.3.1) POST SYNTHESIS FUNCTIONAL SIMULATION:

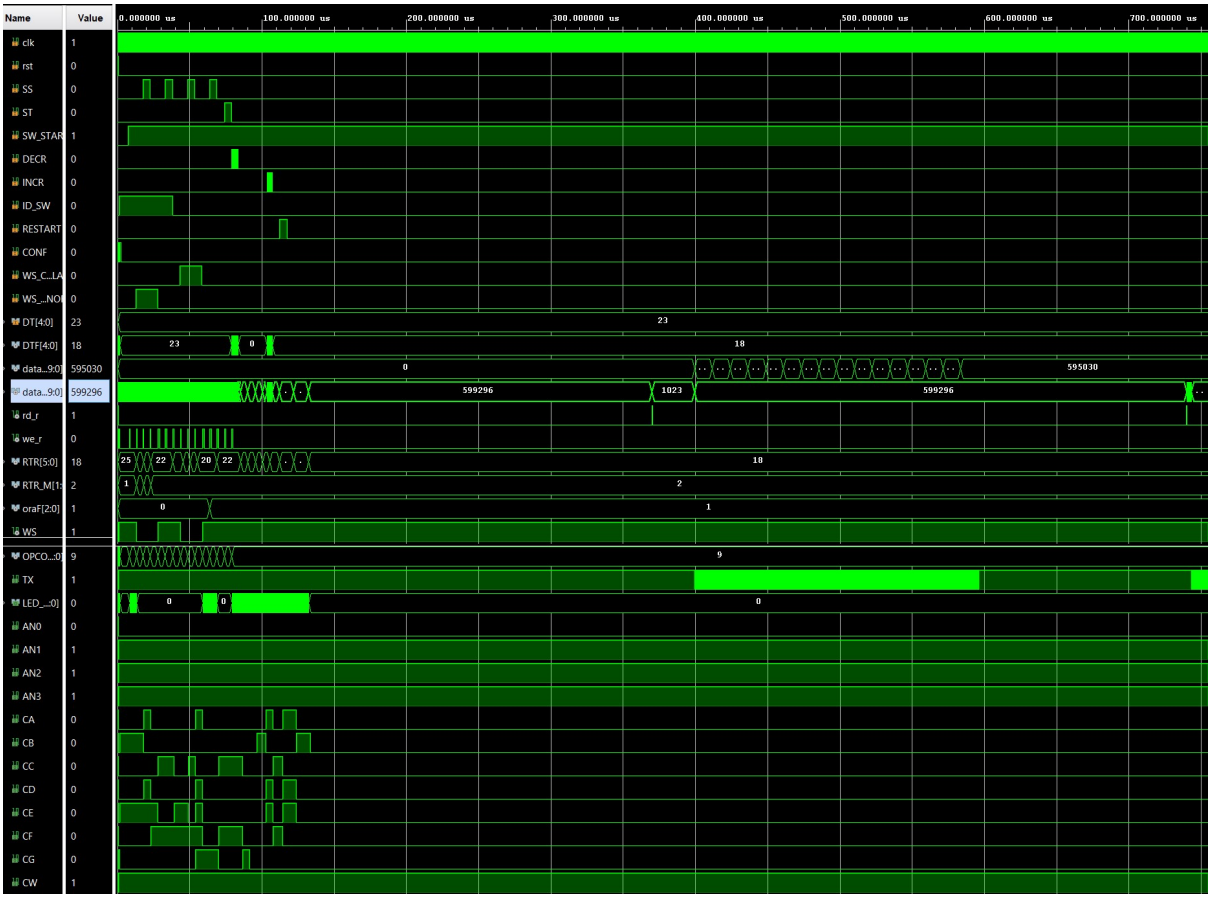


Figure 14- Post Sysnthesis Functional

4.3.2) POST SYNTHESIS TIMING SIMULATION:

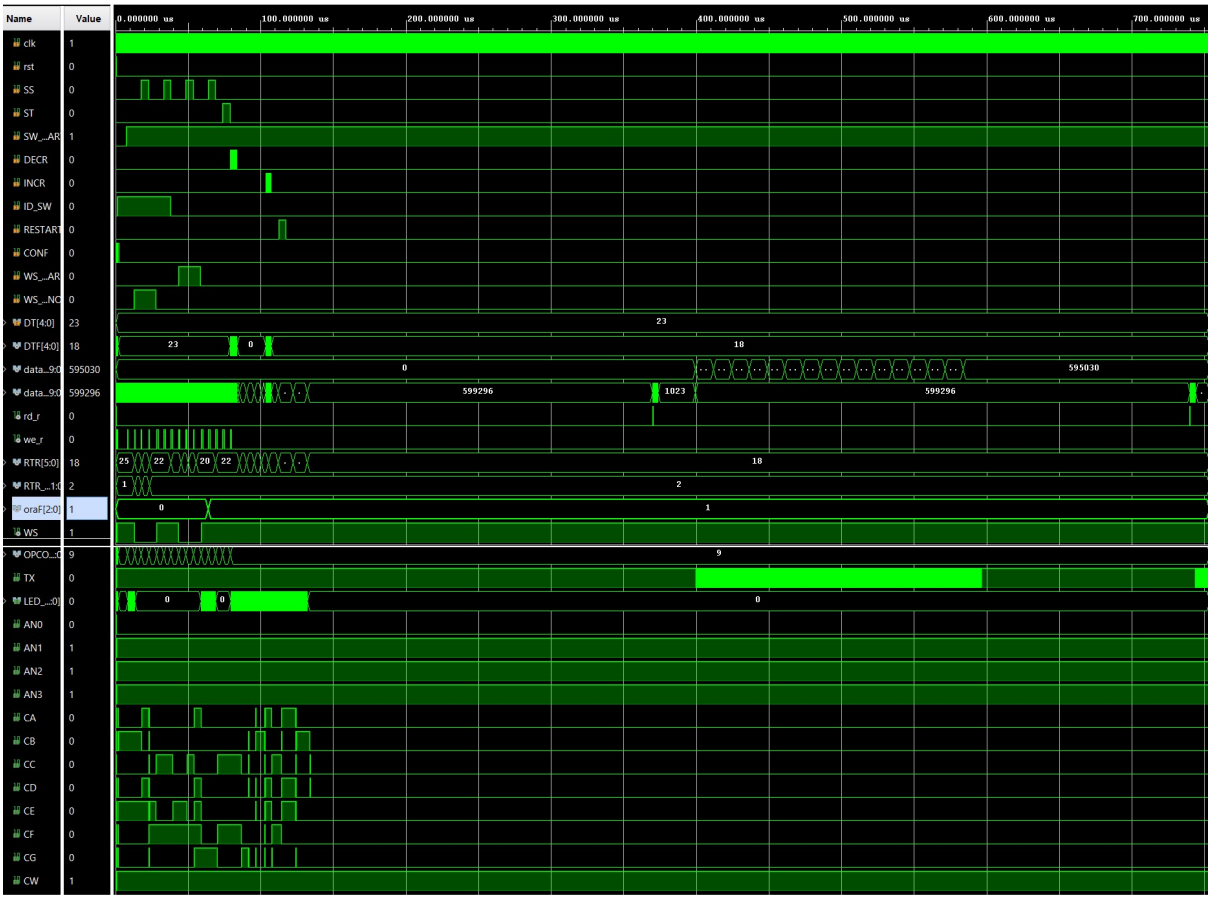


Figure 15-Post Synthesis Timing

4.3.3) POST IMPLEMENTATION FUNCTIONAL SIMULATION:

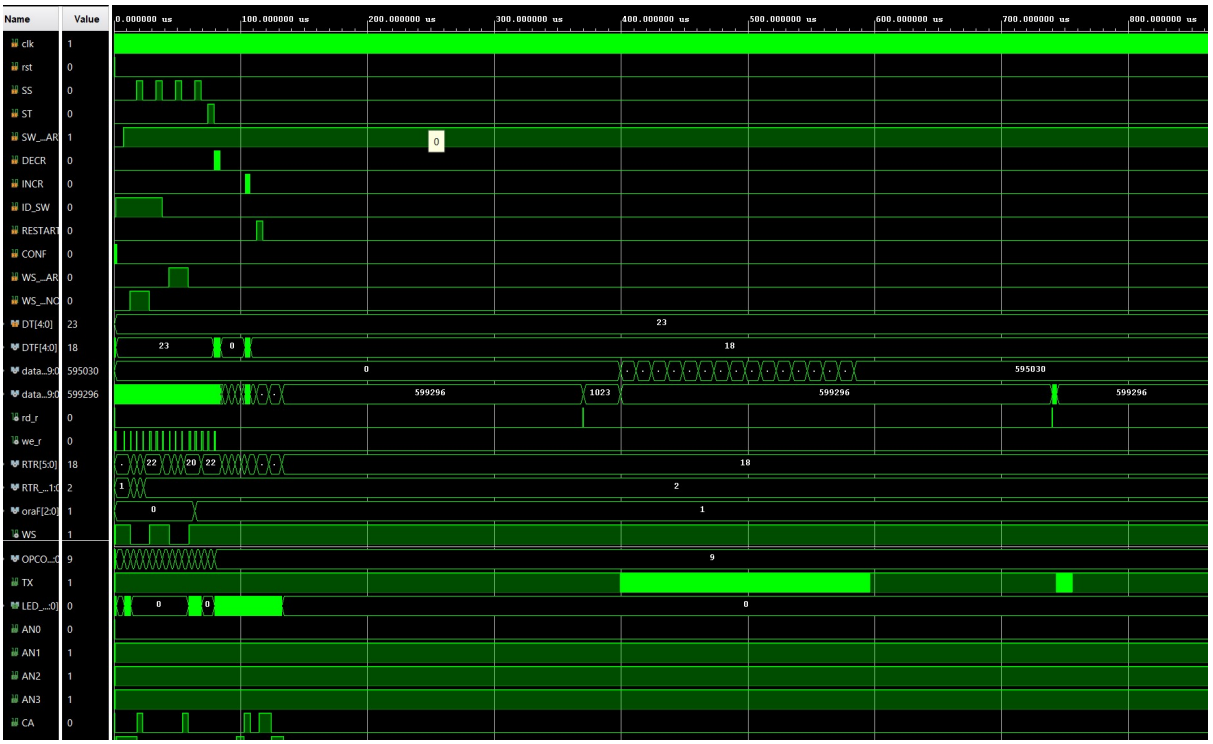


Figure 16-Post Implementation Functional

4.3.4) POST IMPLEMENTATION TIMING SIMULATION:

Figure 17-Post Implementation Timing

As we can see the behaviour is the same for all the post synthesis and post implementation simulations. We have tried to apply different synthesis and implementation settings but we did not manage to have the same signals of behavioural simulation. For the signals that we can see in post-synthesis and post-implementation simulations no glitches have been found.

SYSTEM CORE

5.1) System Core Datapath

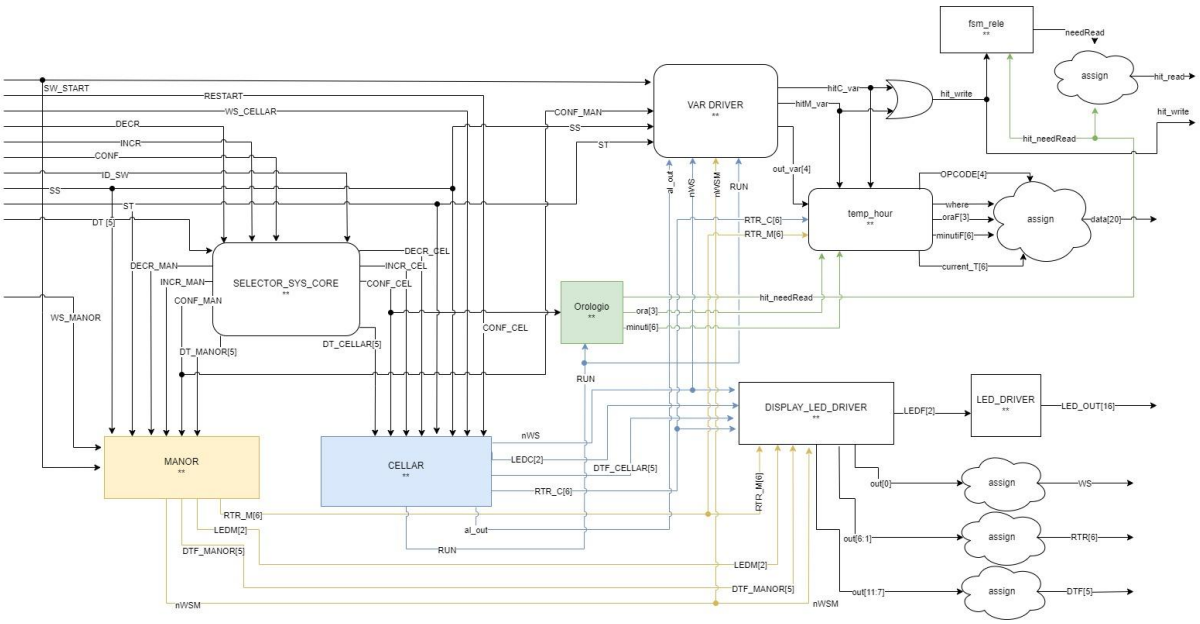


Figure 18-System Core Datapath

5.2) System Core Description

Input:

- clk: Clock
- rst: Reset
- SW_START: When set to 1 starts the thermostat control system of the main manor
- RESTART: If the cellar window is closed, when pushed the wine cellar alarm condition is terminated
- WS_CELLAR: Switch to 1 means the windows are open, otherwise they are closed
- DECR: -1°C in the desired temperature
- INCR: +1°C in the desired temperature
- CONF: Confirmation of the temperature selection
- ID switch: If 1 Manor, 0 Wine cellar
- SS: If 1 set summer, 0 set winter
- ST: if 1 set day, 0 set night
- DT[5]: Switches to configure the initial desired temperature
- WS_MANOR: Switch to 1 means the windows are open, otherwise they are closed

Output:

- RTR[6]: Relative temperature registered
- DTF[5]: Desider temperature final
- WS: Signal to DISPLAY LED DRIVER, it is a signal of Window based by ID_SW, (get the LSB of DISPLAY LED DRIVER output).

- hit_write: If high there is a variation of signal in Var Driver, else there isn't.
- hit_read: Is an output by Orologio, if it is HIGH, it means that Orologio has counted 6 hours.
- data[20]: Contain OPCODE, where, oraF, minutiF, currentT, signal's value.
- LED_OUT[16]: A signal for LED VALUE
- out[12]: Contain the output for the 7 segment display

The System Core manages the entire system. It contains MANOR, CELLAR, SELECTOR_SYS_CORE, VAR_DRIVER, DISPLAY_LED_DRIVER, LED_DRIVER, Orologio, fsm rele and temp_hour modules.

It works with 2 principal Modules, the Manor and the Cellar.

The some inputs (The Selector's input) for the 2 Modules are selected by SELECTOR, based on the **ID_SW**, it routes the input for Manor or Cellar, the others are basically input.

The Orologio manages the hit to refresh a video message to SPI for UART every 6 Hours.

Var Diver manages the variation of the interest signals, for temp_hour, that group by Current Temperature, where, Time (hour, minute), and OPCODE (it is used to select the part of message sent to screen) based on the signal that had a variation.

The fsm Rele, is used to manage **hit_read** signals to SPI.

The Display Led Driver drives the **WS**, **RTR**, **DTF** outputs, and the **LED** value of Manor and Cellar outputs dependent on **ID_SW**.

The LED Driver manages the LEDF output of the Display Led Driver.

5.3) System Core Testbench

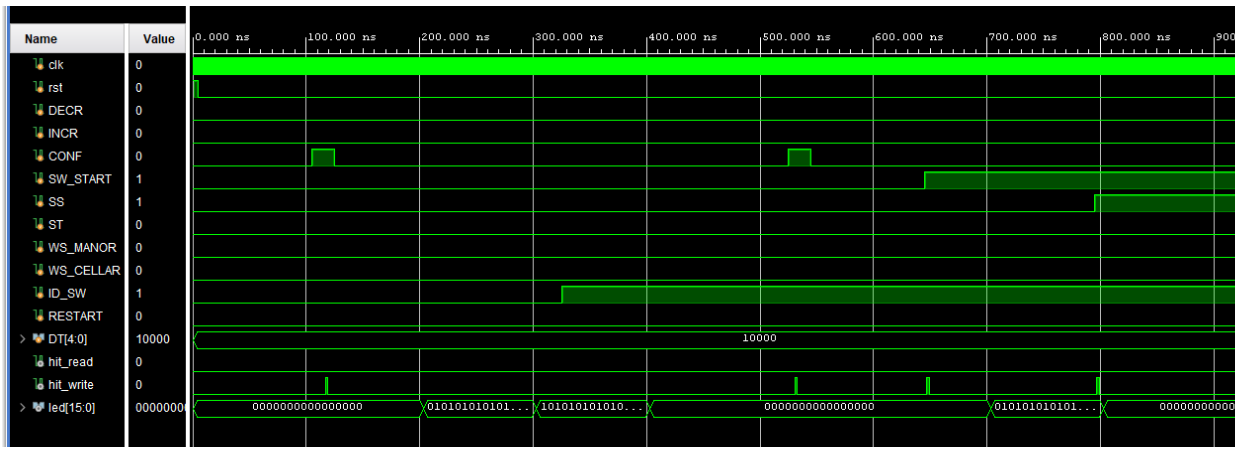


Figure 19-System Core Testbench

In this first part of the testbench we can see how the **leds** of the cellar start right after the first **CONF**.

When the **ID_SW** changes the leds return off because the manor thermostat isn't in function until the **SW_START** is on, and when this happens the **leds** restart to blink due to the changing of the temperature. Then we can see how the **hit_write** always sees the variations and goes to 1.

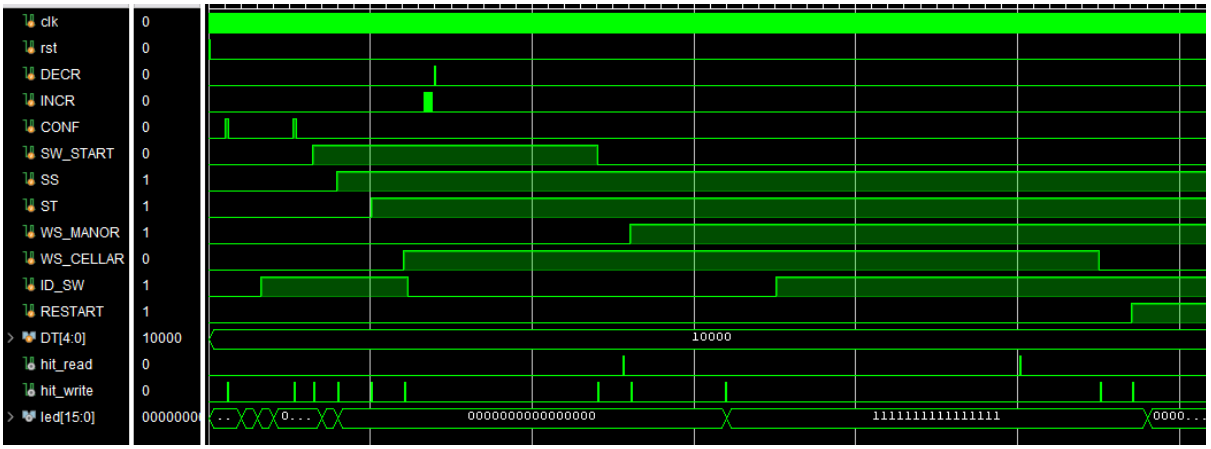


Figure 19-System Core Testbench 2

In this part we can see the system stops at the opening of the **WS_CELLAR** (now the **ID_SW** is 0 so we are seeing the cellar thermostat). Then the alarm is **activated** (all **leds** are 1) because current temperature is very high (**SS** and **ST** are high) and doesnt stop until the **window** is closed and the **restart** is pressed. The alarm shows even if the ID_SW is on the manor.

5.4) Manor

5.4.1) Manor Datapath

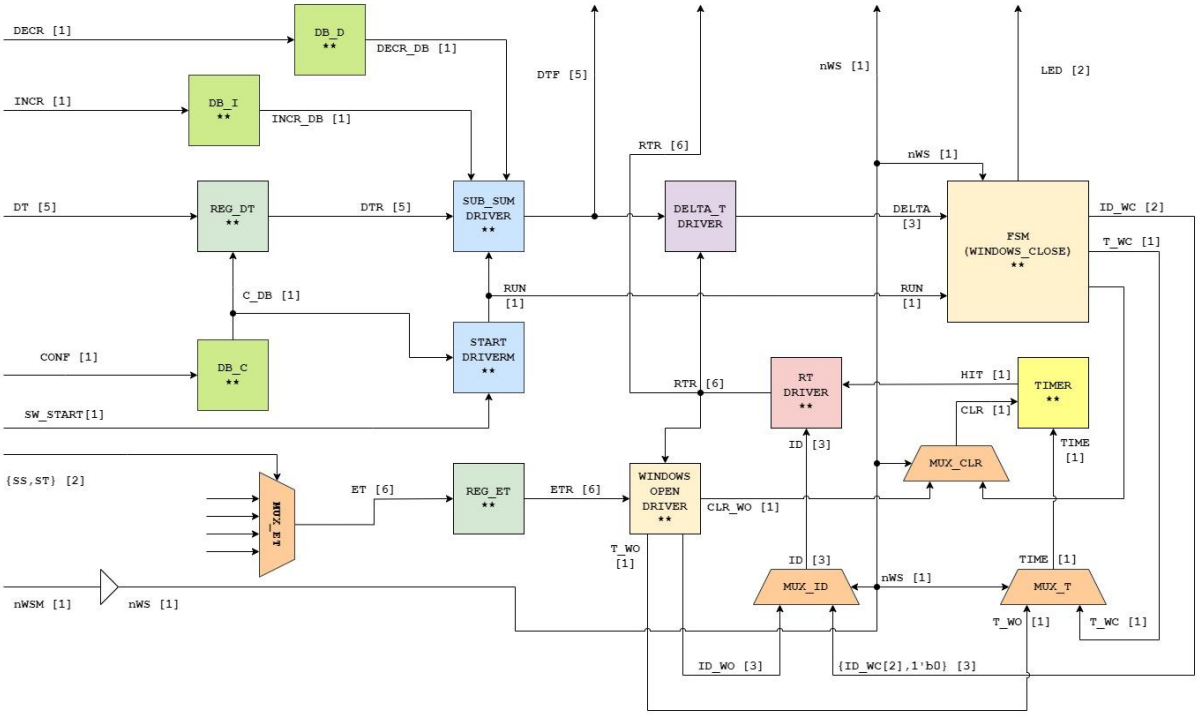


Figure 20-Manor Datapath

5.4.2) Manor Description

Input:

- clk: Clock
- rst: Reset
- DECR: -1°C in the desired temperature
- INCR: +1°C in the desired temperature
- CONF: Confirmation of the temperature selection
- SW_START: When set to 1 starts the thermostat control system of the main manor
- SS: If 1 set summer, 0 set winter
- ST: If 1 set day, 0 set night
- nWSM: Switch to 0 means the windows are open, otherwise they are closed
- DT [5]: Switches to configure the initial desired temperature

Output:

- nWS: Switch to 1 means the windows are open, otherwise they are closed
- DTF [5]: Desiderate Temperature Final
- RTR[6]: Room Temperature Registered
- LED[2]: Leds signals

The behavior of the Manor is the same as the General System Description , except that the desired temperature cannot be taken into

consideration until a **Confirmation Button** is pressed and **SW_START** is high. For this change, the MANOR START DRIVER was modified.

5.4.3) Manor Testbench

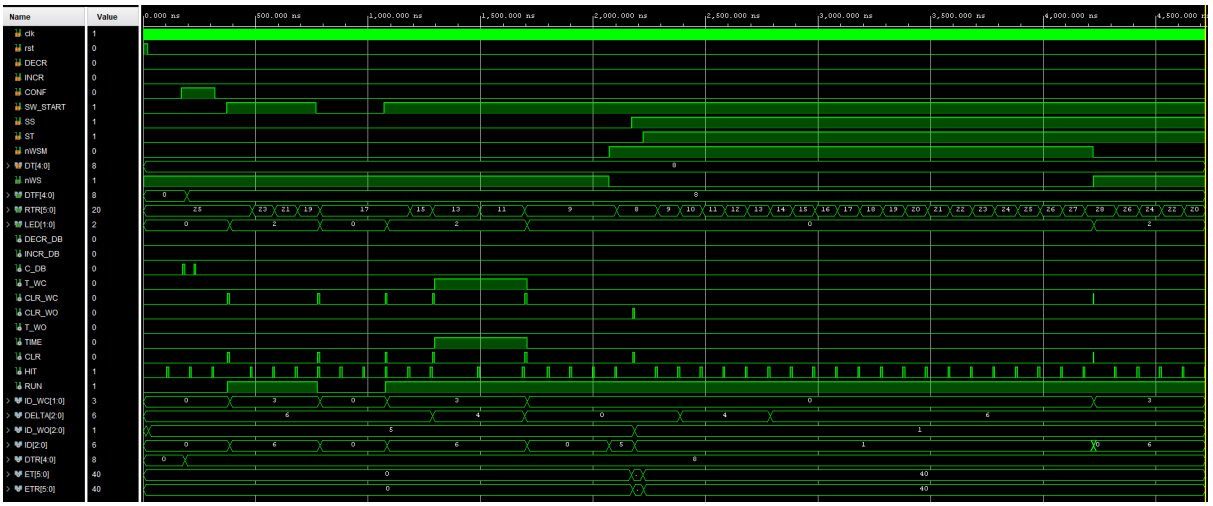


Figure 21-Manor Testbench

In this testbench the additions made to the system are shown, so after pressing **CONF** you will have the option to use **SW_START** that when it is high the system is running and when it is low the system shuts down. In addition to the previous assignment we have corrected the **nWSM** window by negating it to become **nWS** so that when it is low it means it is open while when it is high it means it is closed.

5.4.4) Fsm Start Driver

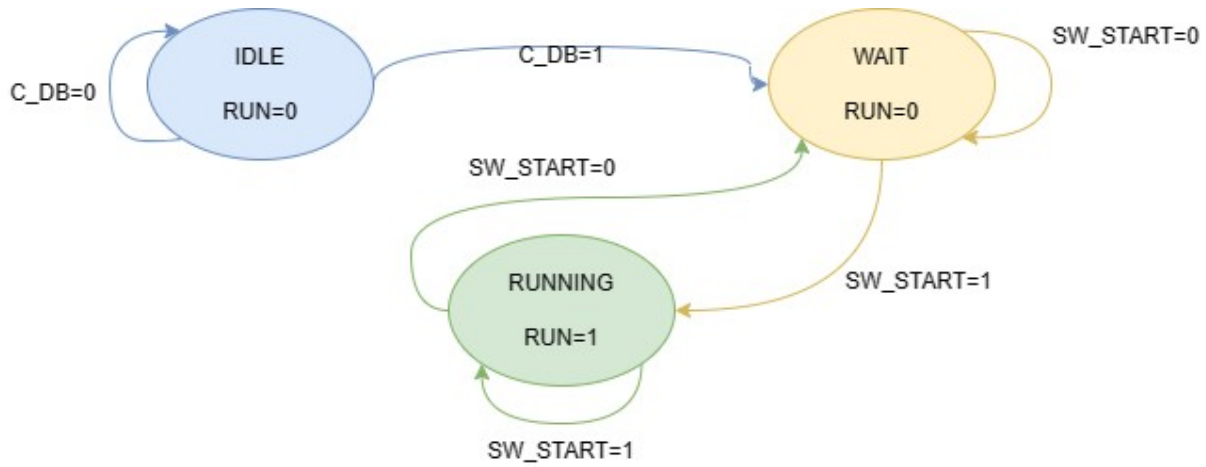


Figure 22- FSM Start Driver

5.4.5) Fsm Start Driver Table

Next state Table

	INPUTS	{SW_START, C_DB}		
STATE	00	01	10	11
IDLE	IDLE	WAIT	IDLE	WAIT
WAIT	WAIT	WAIT	RUNNING	RUNNING
RUNNING	WAIT	WAIT	RUNNING	RUNNING

Table 11-Next State Start Driver

Moore's output table

STATE	OUTPUT
IDLE	RUN=0
WAIT	RUN=0
RUNNING	RUN=1

Table 12- Moore's Output Start Driver

5.4.6) Fsm Start Driver Description

Input:

- clk: Clock
- rst: Reset
- C_DB: button CONF Debounced
- SW_START: When set to 1 starts the thermostat control system of the main manor

Output:

- RUN: If 1 the systems is on otherwise is off

State:

- IDLE
- WAIT
- RUNNING

C_DB is the debounced signal of button **CONF**, the behavior mimics the functionality of a relè (that can only turn on devices). By default, the system is in **IDLE** state and the output signal **RUN** is low, when **C_DB** is pressed the module moves to **WAIT** state where it stays there in **SW_START** is equal to 0 otherwise it change state to **RUNNING**, only in

this state the output signal **RUN** is high. If **SW_START go** to 0, the system will change to **WAIT** state.

5.4.7) Fsm Start Driver Testbench

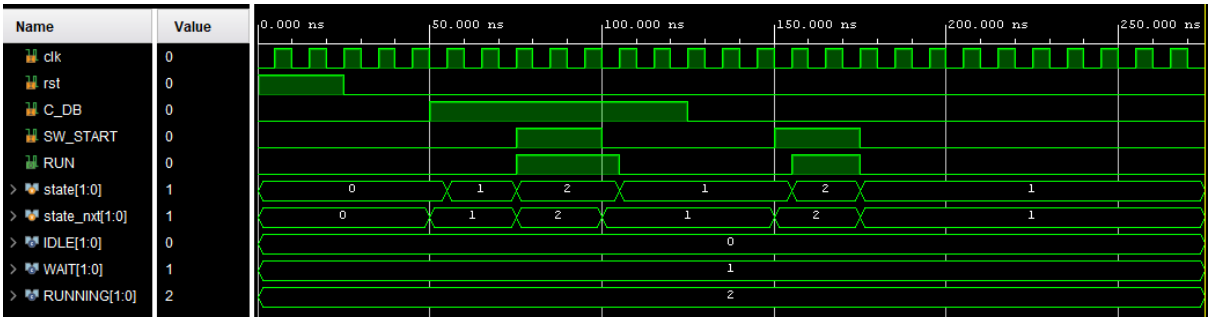


Figure 23-FSM Start Driver Testbench

In this testbench, it is shown how as soon as **C_DB** goes high for the first time, the system becomes dependent on **SW_START** because if it is high **RUN** will equal 1 and if **SW_START** is low **RUN** will equal 0

5.5) Wine Cellar

5.5.1) Cellar Datapath

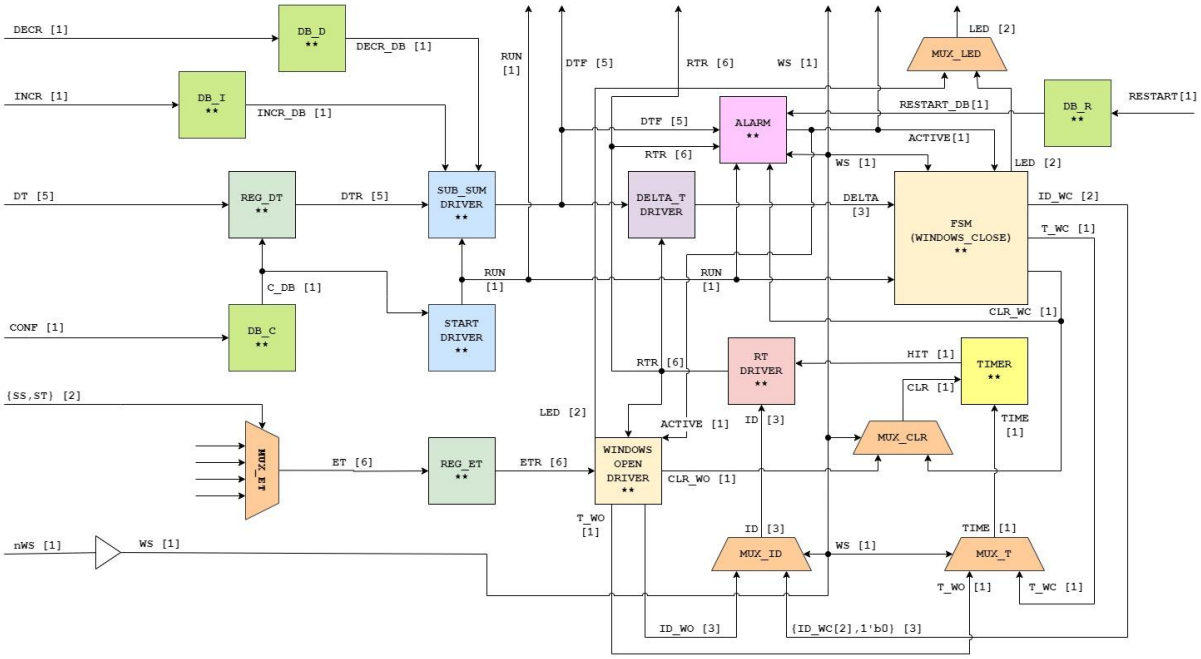


Figure 24- Cellar Datapath

5.5.2) Cellar Description

Input:

- clk: Clock
- rst: Reset
- DECR: -1°C in the desired temperature
- INCR: +1°C in the desired temperature
- CONF: Confirmation of the temperature selection

- RESTART: Button that if the window is closed stops the alarm
- SS: If 1 set summer, 0 set winter
- ST: If 1 set day, 0 set night
- nWS: Switch to 0 means the windows are open, otherwise they are closed
- DT [5]: Switches to configure the initial desired temperature

Output:

- WS: Switch to 1 means the windows are open, otherwise they are closed
- DTF [5]: Desiderate Temperature Final
- RTR[6]: Room Temperature Registered
- LED[2]: Leds signals
- RUN: Will be the enable for the orologio
- active: The alarm is on

The behavior of the Cellar is the same as the General System Description , except that an alarm was added when the desired temperature is ± 3° higher than the current temperature. . For this change, FSM Windows Close, Windows Open Driver and Led Driver have been modified and Top Alarm with its respective modules was added to the system.

5.5.3) Cellar Testbench

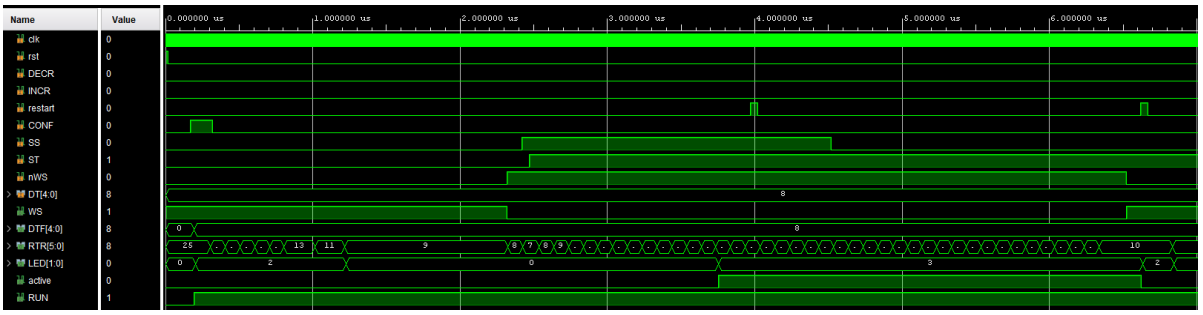


Figure 25- Cellar Testbench

This testbench intended to show the alarm inside Cellar, the system starts as soon as **CONF** is high, having set **DT** to 8°C the room starts to cool down (**RTR** is 25°C by default), we then open the window, in addition to the previous assignment we have corrected the **nWS** window by negating it to become **WS** so that when it is low it means it is open while when it is high it means it is closed. As soon as 30 minutes have passed since **DTF** is less than 3 with respect to **RTR**, the alarm is triggered and to remove it we must press **RESTART** when **nWS** is low.

5.5.4) Fsm Windows Close

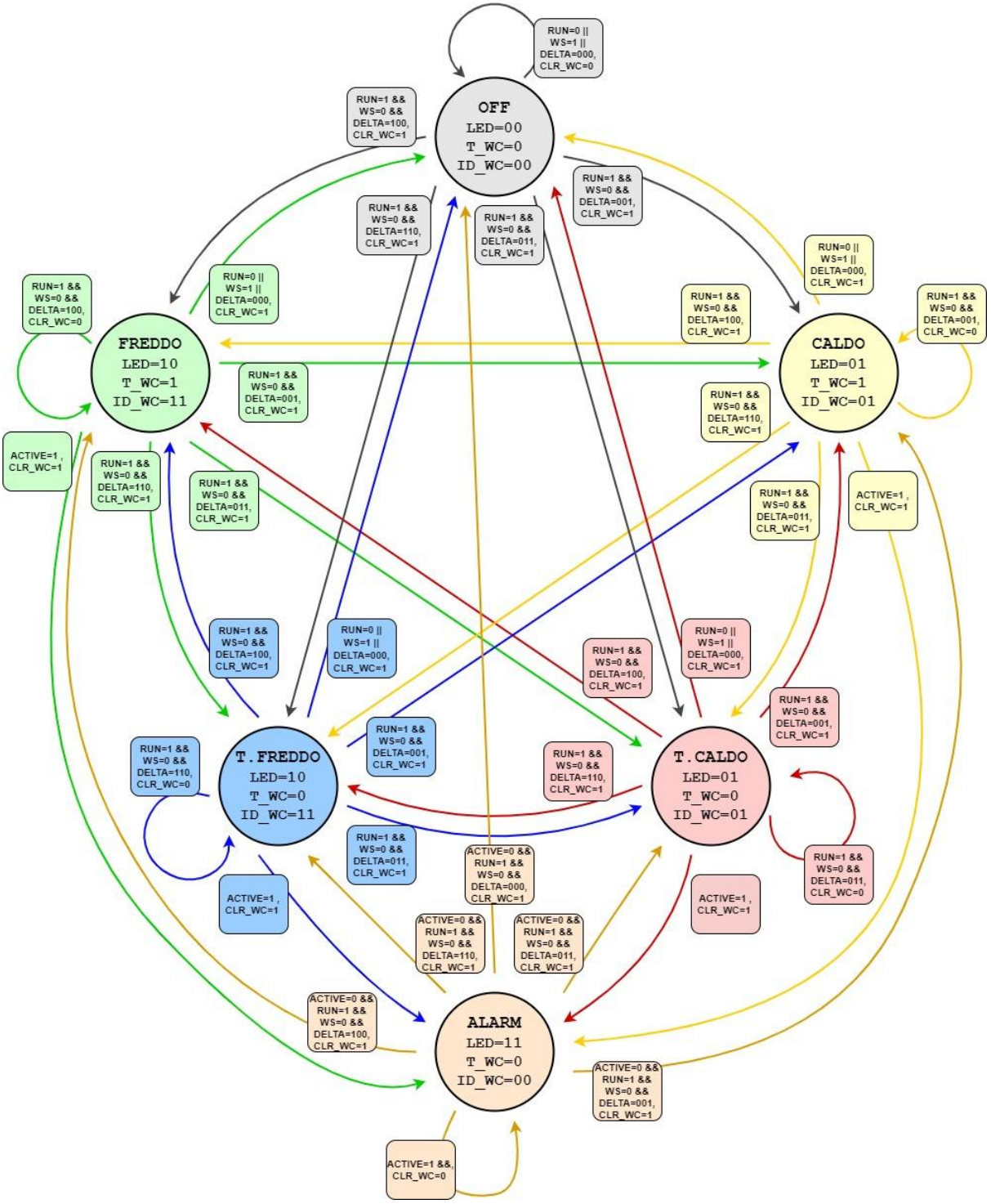


Figure 26- FSM Windows Close

5.5.5) Fsm Windows Close Table

Mealy's Next state Table

	INPUT	{ACTIVE, RUN, WS, DELTA[3]}				
STATE	010001	010011	010100	010110	0****	1****
FREDDO	CALDO/	T.CALDO/	FREDDO/	T.FREDDO	*	ALARM/1

	1	1	0	/1		
CALDO	CALDO/ 0	T.CALDO/ 1	FREDDO/ 1	T.FREDDO /1	*	ALARM/1
T.CALDO	CALDO/ 1	T.CALDO/ 0	FREDDO/ 1	T.FREDDO /1	*	ALARM/1
T.FREDD O	CALDO/ 1	T.CALDO/ 1	FREDDO/ 1	T.FREDDO /0	*	ALARM/1
ALARM	CALDO/ 1	T.CALDO/ 1	FREDDO/ 1	T.FREDDO /1	*	ALARM/0
			STATO* / CLR_WC			

Table 13- Mealy's Next State FSM WC

Moore's output table

STATE	LED	T_WC	ID_WC
ALARM	11	0	00

Table 14-Moore's Output FSM WC

5.5.6) Fsm Windows Close Description

Input:

- clk: Clock
- rst: Reset
- ACTIVE: If 1 the alarm is active
- RUN: The system is on
- WS: windows
- DELTA: tells you in which state to go

Output:

- CLR_WC: Clear windows close
- LED: Led
- T_WC: Responsible for 1- and 2-minutes managing
- ID_WC: 2°C Decrementing or incrementing

State:

- OFF
- CALDO
- FREDDO
- T.CALDO
- T.FREDDO
- ALARM

Respect to the State Machine module, we have added **active** to the input which brings each state to **ALARM**(new state entered) if **active** is 1 and back to the **other states** when it is 0.

5.5.7) Fsm Windows Close Testbench

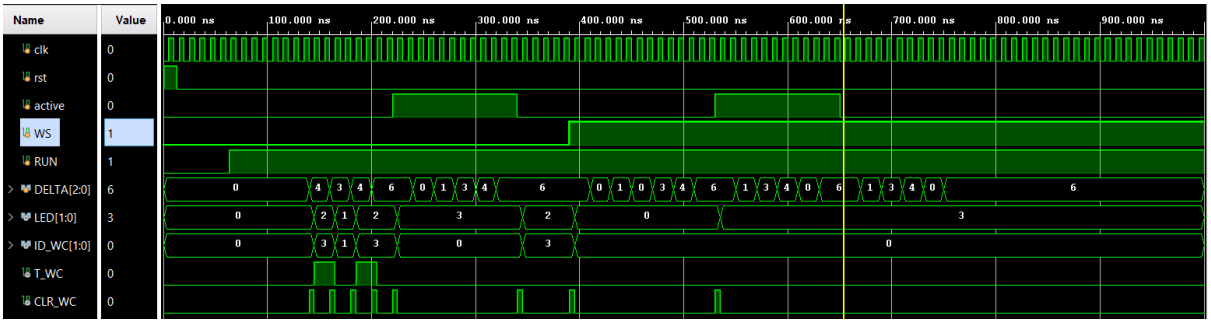


Figure 27-Windows Close Testbench

The cellar's fsm start when **RUN** = 1 and when WS = 0 (because is negated), so when **ACTIVE** =1 the **LED**'s value is 3 (all on) until **ACTIVE** != 1.

After is tested with **WS** = 1, so when **ACTIVE** = 1 the **LED**'s value is 3, but when **ACTIVE** = 0, the **LED**'s value is 3.

5.5.8) Top Alarm Datapath

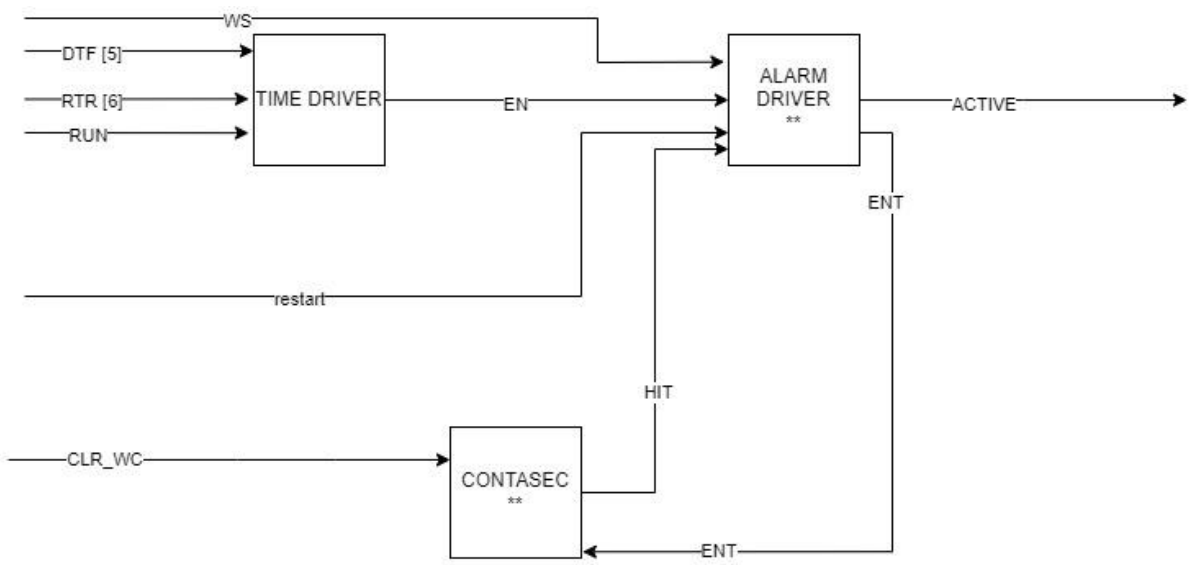


Figure 28-Top Alarm Datapath

5.5.9) Top Alarm Description

Inputs:

- clk: Clock
- rst: Reset
- DTF : desired temperature
- RTR: current temperature
- RUN: if high the system is running
- ws: window
- restart: button that if the window is closed stops the alarm

Outputs:

- active:signal from the alarm going to the state machine

The module consists of 2 sub-modules: alarm driver and 30min_driver. Alarm driver is the state machine, Fsm Alarm.

30min_driver is a combinatorial module that continuously compares the current temperature **RTR** and the desired temperature **DTF** and if the difference between the two enters the alarm range it raises an enable that reaches the machine at alarm states.

5.5.10) Top Alarm Testbench

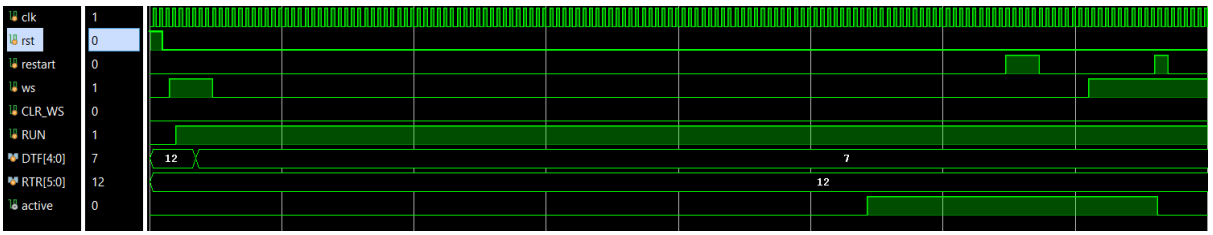


Figure 29- Top Alarm Testbench

In this testbench it is shown how the alarm works, when as soon as 30 minutes pass with the difference of ± 3°C between **RTR** and **DTF** , **active** is activated which is only deactivated by **RESTART** when **WS** is high.

5.5.11) Fsm Alarm

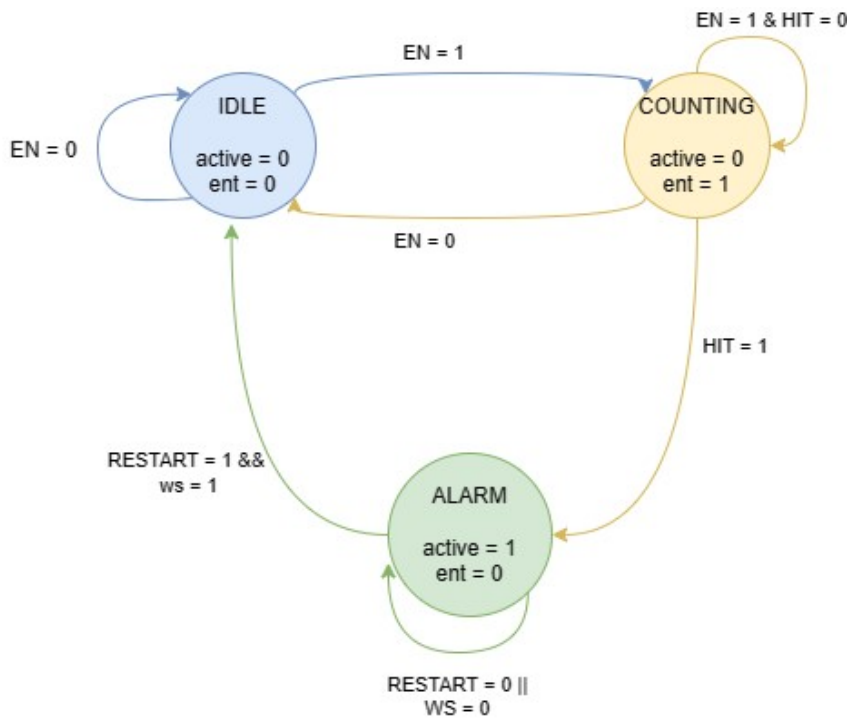


Figure 30- FSM Alarm

5.5.12) Fsm Alarm Table

Next state Table

	INPUTS	{EN, HIT, WS,RESTART}						
STATE	0000	1***	10**	*1**	**00	**01	**10	**11
IDLE	IDLE	COUNTING	COUNTING	IDLE	IDLE	IDLE	IDLE	IDLE
COUNTING	IDLE	x	COUNTING	ALARM	x	x	x	x
ALARM	ALARM	ALARM	ALARM	ALARM	ALARM	ALARM	ALARM	IDLE

Table 15- Next State FSM Alarm

Moore's output table

STATE	ACTIVE	ENT
IDLE	0	0
COUNTIN G	0	1
ALARM	1	0

Table 16-Moore's Output FSM Alarm

5.5.13) Fsm Alarm Description

Input:

- clk: Clock
- rst: Reset
- en : enable from the timer
- restart : button that if the window is closed stops the alarm
- hit:signal from the time counter timer indicating that 30 minutes have been reached
- ws: windows

Output:

- active: signal going to the state machine for warn the alarm
- ent: enable for the timer

State:

- IDLE
- COUNTING
- ALARM

Fsm Alarm manages the generation of the **active** signal sent to the general system status machine and its deactivation via **restart**, it also manages the second-counting timer:

IDLE: initial state in which the 30 minutes are not counted because the temperature values are correct it only move to **counting** if **en** is high.

COUNTING: the difference between the temperatures exceeds +3/-3, as long as **en** is high and it remains in as long as **en** is high, otherwise it return to **IDLE**. If **hit** is raised it go into **ALARM**.

ALARM:. It remains in this state as long as not **restart** is pressed while the **window** is closed, otherwise it remains in the same state.

5.5.14) Windows Open Driver Description

Input:

- clk: Clock
- rst: Reset
- ETR [6]: External Temperature Registered
- RTR [6]: Room Temperature Registered
- active: if high the alarm is active

Output:

- T_WO : Time Window Open
- ID_WO [3] : Increment/Decrement Window Open
- CLR_WO: Clear Window Open
- LED: Led

Respect to the Window_Open_Driver module, we have added **active** to the input which outputs the **LED** value to warn if you are in alarm status.

5.5.15) Windows Open Driver Testbench

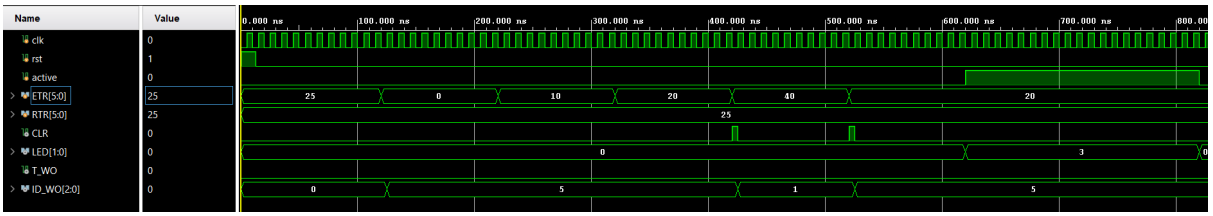


Figure 31- Windows Open Driver Testbench

In this testbench we can see how as soon as **active** goes high, the **LED** value goes from 0 to 3 and when **active** goes down **LED** returns to 0.

5.6) Selector Sys Core

5.6.1) Selector Datapath

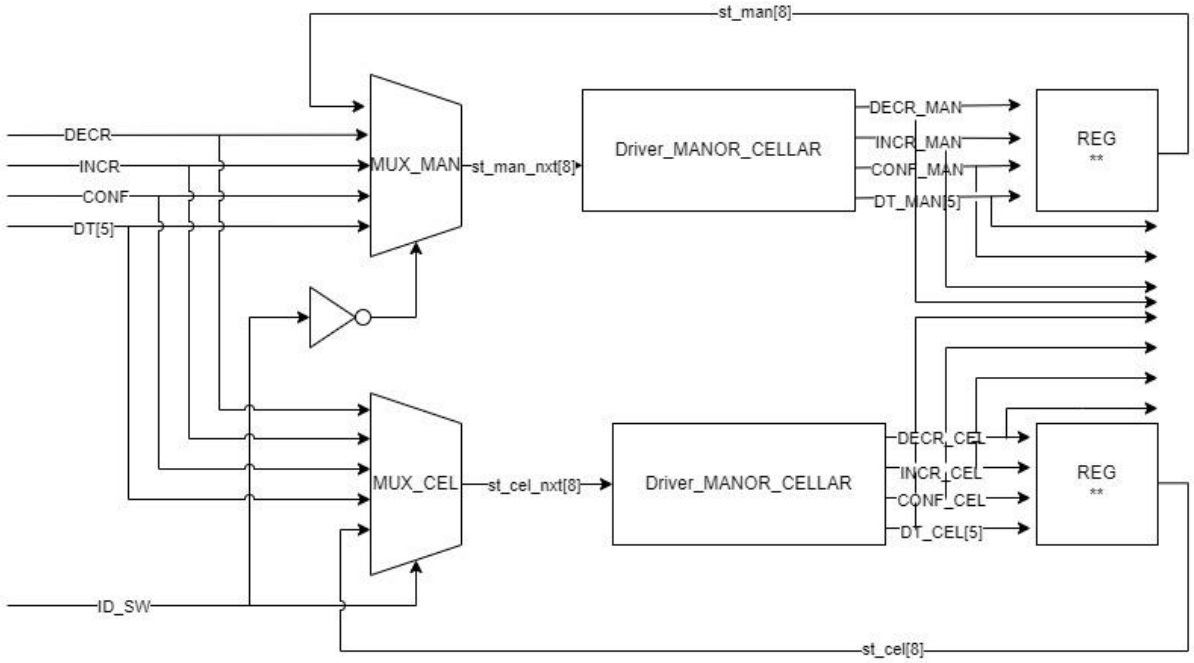


Figure 32-Selector Datapath

5.6.2) Selector Description

Inputs:

- clk: Clock
- rst: Reset
- DECR: -1°C in the desired temperature
- INCR: +1°C in the desired temperature
- CONF: Confirm the temperature selected by DT
- DT[5]: Switches used to set initial desidered temperature
- ID_SW: if 1 Manor, if 0 Cellar

Outputs:

- DECR_MAN: Get DECR signal input if ID_SW = 1
- INCR_MAN: Get INCR signal input if ID_SW = 1
- DECR_CEL: Get DECR signal input if ID_SW = 0
- INCR_CEL: Get INCR signal input if ID_SW = 0
- CONF_MAN: Get CONF signal input if ID_SW = 1
- CONF_CEL: Get CONF signal input if ID_SW = 0
- DT_MAN[5]: Get DT value input if ID_SW = 1
- DT_CEL[5]: Get DT value input if ID_SW = 0

The Selector System Core's work is routing through the **ID_SW** value get by INPUT, for Cellar and Manor.

If **ID_SW** = 1, it get inputs for cell's out, vice versa for manor's out.

In Driver_MANOR_CELLAR, there is always logic to pass the st_nxt value in input for the output respective.

5.6.3) Selector Testbench

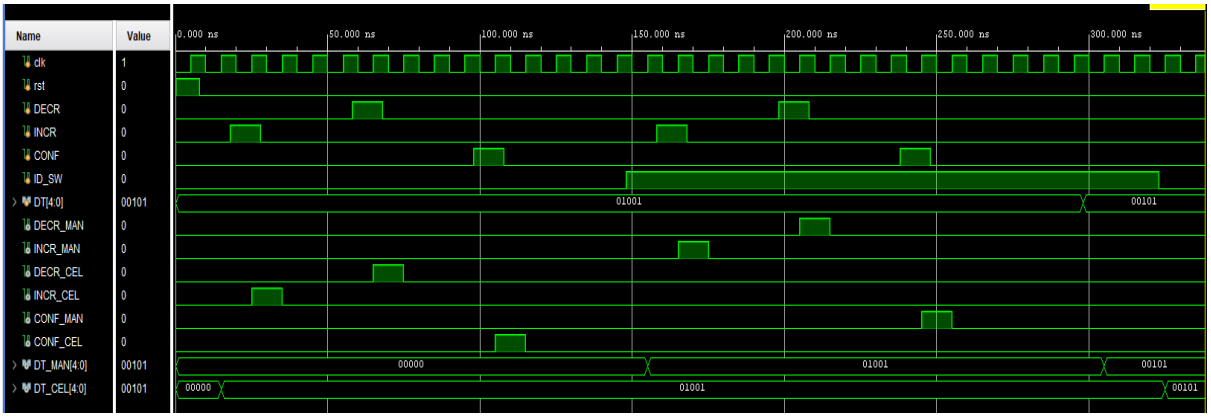


Figure 33-Selector Testbench

In the testbench can be seen that **ID_SW** = 0 updates the corresponding output signal for Cellar, at the next clock cycle.

When **ID_SW** changes to 1, the next variation, updates the value for Manor.

5.7) Var Driver

5.7.1) Var Driver Datapath

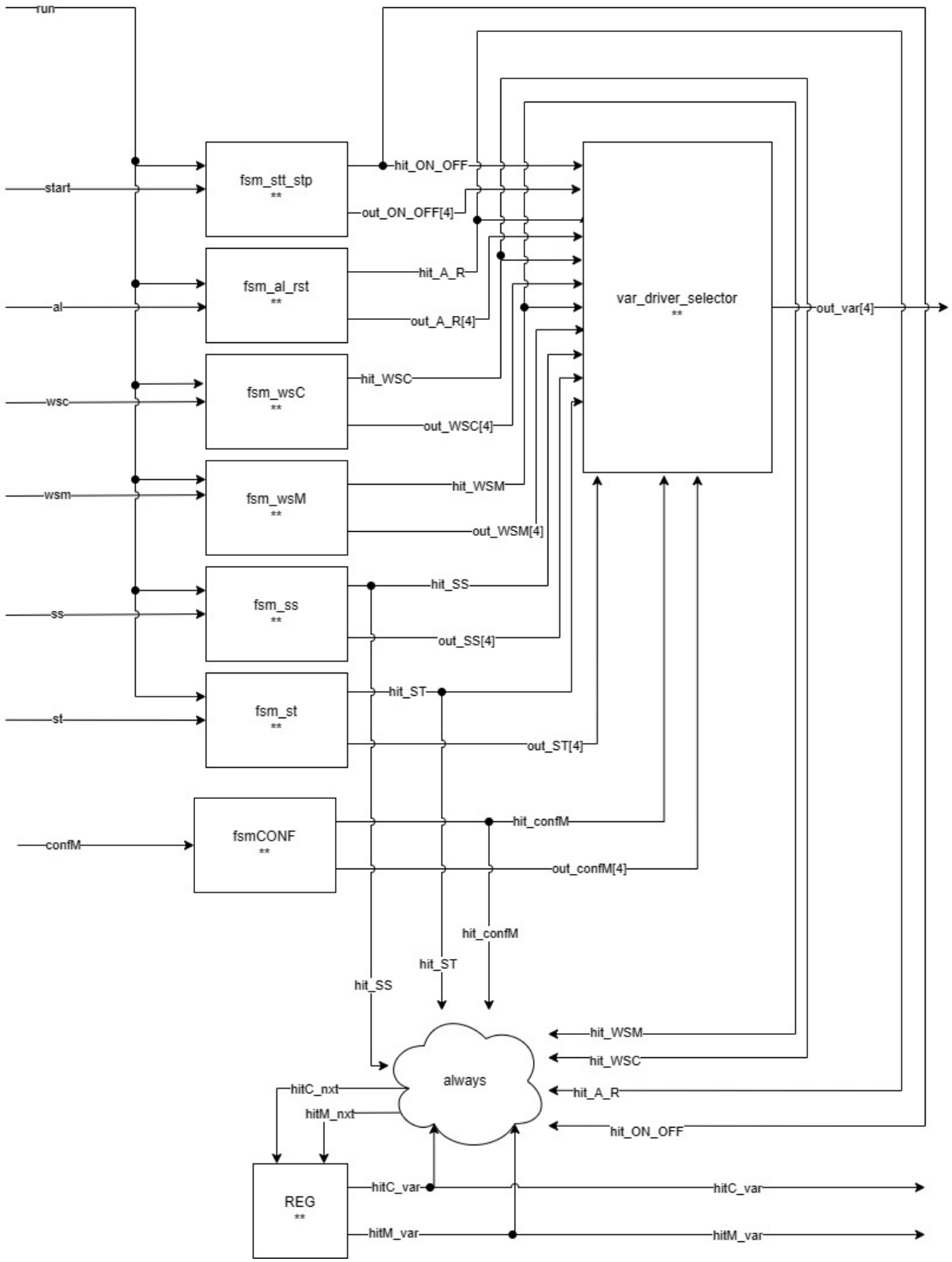


Figure 34- Var Driver Datapath

5.7.2) Var Driver Description

Inputs:

- clk: Clock
- rst: Reset
- start: SW_START
- run: Cellar signal RUN, is 1 if cellar is configured
- al: Alarm signal from cellar
- wsc: WS_CELLAR
- wsm: WS_MANOR
- confM: Is confirm signal from Manor
- SS: season switch
- ST: time switch

Outputs:

- hitC_var: if change one signal to Cellar begin 1 for one clock cycle, else 0
- hitM_var: if change one signal to Manor begin 1 for one clock cycle, else 0
- out_var [4]: get the "ID" of the signal who is change

The module's job is to take control of the variation for each input signal, with fsm dedicated, and generate a signal for OPCODE for UART.

The signal go into the correspondent fsm, after the outputs go to var_driver_selector module, where in base of the **hit** and id of the signal changed, it push in output the correspondent "id", it is used for identify the signal that has changed, next is shown a table with the "id" for signal:

Name	Value in decimal
AL_ON	1
AL_REST	2
WS_OPEN	3
WS_CLOSE	4
START	5
SUMMER	6
WINTER	7

DAY	8
NIGHT	9
CONFM	11
CONFC	12
STOP	15

Table 17-ID Table

Also for hit signals, they are used to generate **hitC** and **hitM** signals, which are sent to signal for temp_hour module which of the signals of Cellar and manor has changed.

The var_driver_selector module has a case logic, which generates the id value in **out_var** on the basis of the hit raised.

5.7.3) Var Driver Testbench

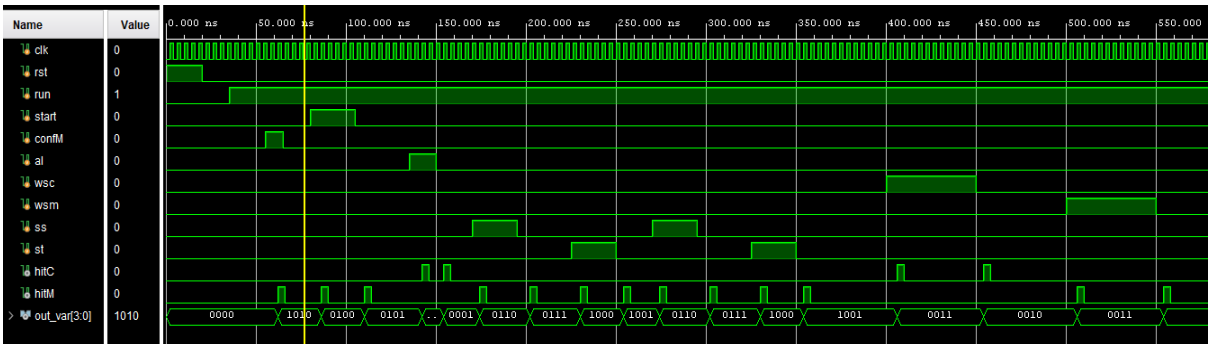


Figure 35- Var Driver Testbench 1

In this testbench is shown in case of **run** = 1, how with the signal variation, "rises" to 1 for one clock cycle the **hit** correspondent of signal, and **out_var** gets the signal's id.

TO CHECK, SEE THE "ID TABLE" AND COMPARE WITH **OUT_VAR** VALUES.

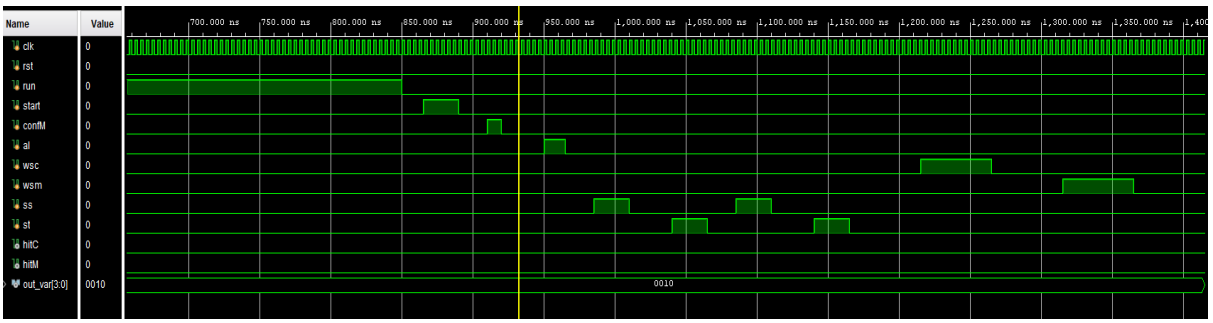


Figure 36- Var Driver Testbench 2

In this testbench is shown the case with **run** = 0, we can see how the Cellar signals are not marked for **hitC** and **out_var** value, because they can't "get up" in this case (It's a little extra check).

5.7.4) FsmVAR

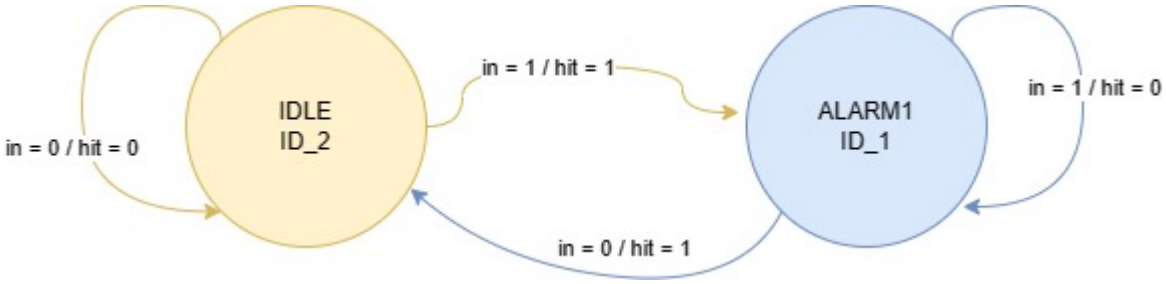


Figure 37- FSM Var

5.7.5) FsmVAR Table

Mealy's Next state Table

STATE	IN = 0	IN =1
IDLE	IDLE/0	ALARM/1
ALARM	IDLE/1	ALARM/0
		STATE nxt / hit

Table 18-Mealy's Next State FSM Var

Moore's output table

STATE	OUTPUT
IDLE	ID_2
ALARM	ID_1

Table 19-Moore's Output FSM Var

ID_1 and ID_2 depends by OPCODE PARAMETER passed:

IN CABLE	ID_1	ID_2
start	START	STOP
al	ALARM ON	ALARM RESET
wsc (window cellar switch)	WINDOWS CLOSE	WINDOWS OPEN
wsm (window manor switch)	WINDOWS CLOSE	WINDOWS OPEN
ss (Season switch)	SUMMER	WINTER
st (Season Time)	DAY	NIGHT

Table 20- ID_1 and ID_2

5.7.6) FsmVAR Description

Inputs:

- clk: Clock
- rst: Reset
- in: it is the signal at which the variation is controlled
- start: Is a run signal.

Outputs:

- hit: is the variation's hit, if state is changing is 1 otherwise 0
- out[4]: get parameter value of correspondent state.

This FSM is used for each input signal of VAR DRIVER, the logic is simple:

Each FSM has 2 parameters and 2 states, each state gets one parameter, it is the same value passed up in the "ID_1 and ID_2 table" of the signal, for each correspondent fsm.

5.7.7) FsmVAR Testbench

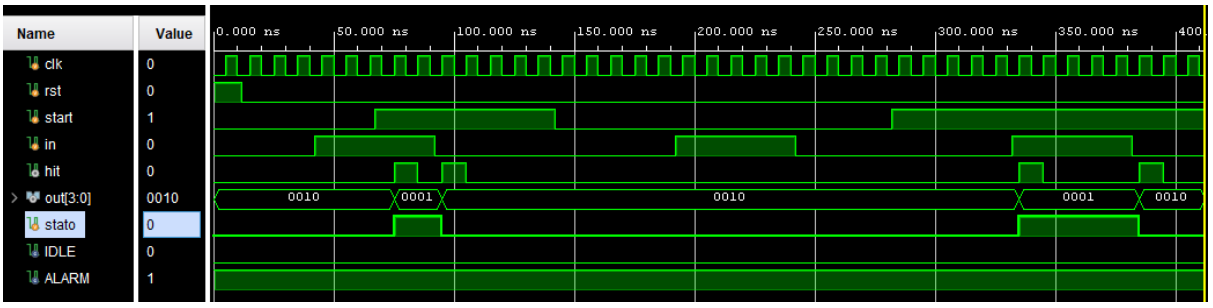


Figure 38- FSM Var Testbench

The out who gets the parameter value passed by VAR DRIVER module, is a MOORE output, but for **hit** is a MEALY output (hit determines the variation of input signal correspondent).

5.7.8) FsmCONF

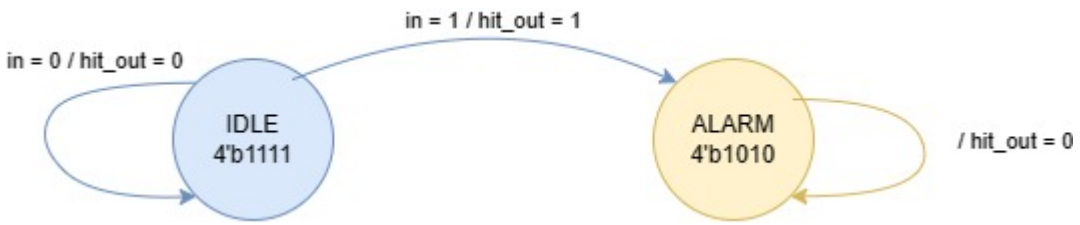


Figure 39-FSM Conf

5.7.9) FsmCONF Table

Mealy's Next state Table

STATE	IN = 0	IN =1
IDLE	IDLE/0	ALARM/1
ALARM	ALARM/0	ALARM/0
		STATE nxt / hit

Table 21-Mealy's Next State FSM Conf

Moore's output table

STATE	OUTPUT
IDLE	4'b1111
ALARM	4'b1010

Table 22-Moore's Output FSM Conf

5.7.10) FsmCONF Description

Inputs:

- clk: Clock
- rst: Reset
- in: Conf button input, by Manor.
- start: Is a Cellar's run signal.

Outputs:

- hit: if there is a confirm press button for the first time is 1, else always 0.
- out[4]: generate the OPCODE with the value of configuration complete or an "error" value (4'b1111).

This FSM sign the configuration of System (when User press only for the first time **conf** to confirm the configuration of last thermostat (MANOR))

5.7.11) FsmCONF Testbench

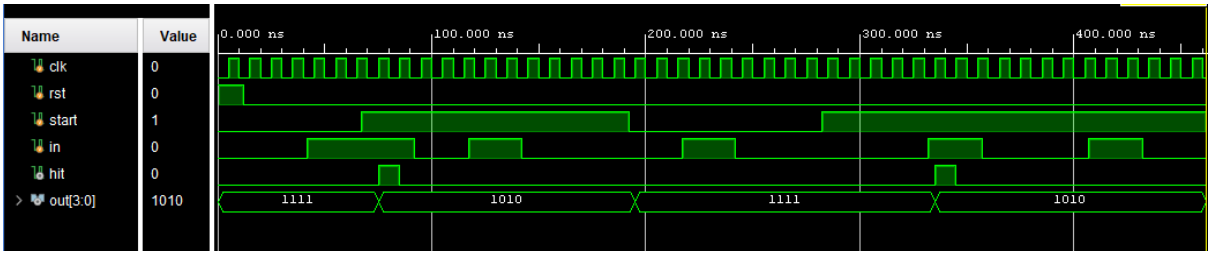


Figure 40-FSM Conf Testbench

How is shown, when the **in** signal is high for the first time, it generates one hit, during **start** (**run**) = 1, the next time, even if confirmation were to be resumed, the **hit** remains low... Only when the system is **reset**, can reconfigure the system.

5.8) Display Led Driver

5.8.1) Display Led Driver Datapath

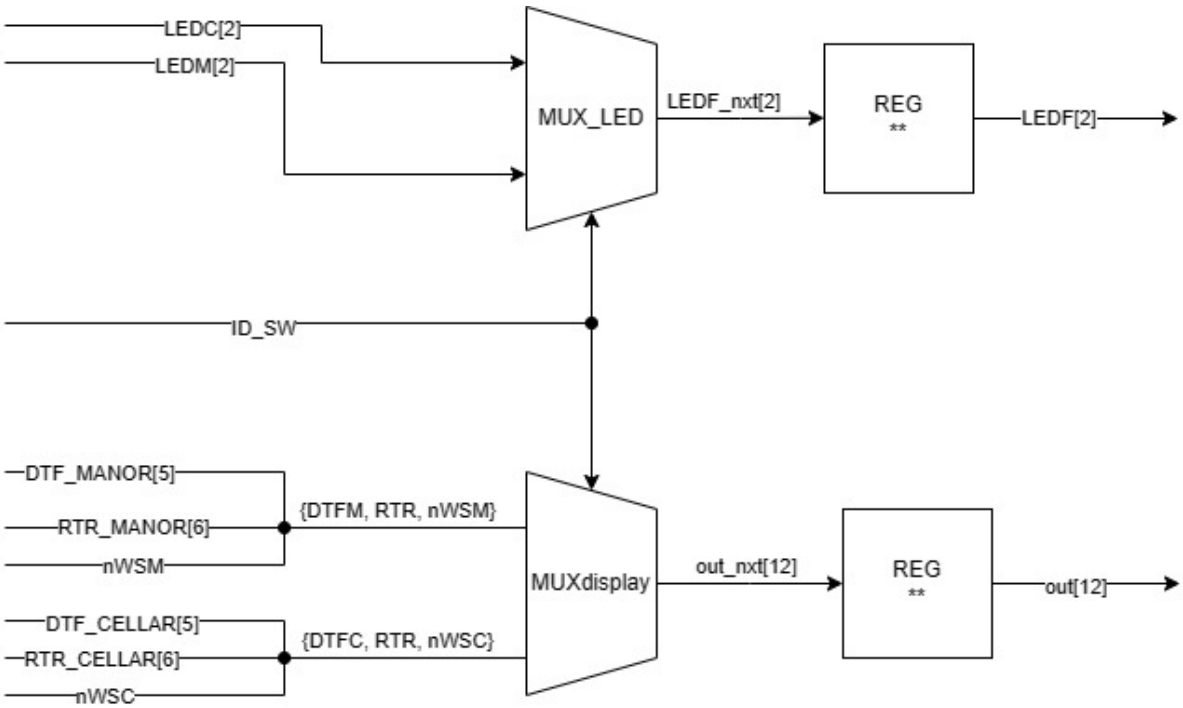


Figure 41- Display Led Driver Datapath

5.8.2) Display Led Driver Description

Inputs:

- clk: Clock
- rst: Reset
- LEDM[1:0]: LED Value from Manor
- LEDC[1:0]: LED Value from Cellar
- DTF_MANOR[4:0]: Manor's Final desired temperature
- DTF_CELLAR[4:0]: Cellar's Final desired temperature
- RTR_MANOR[5:0]: Manor's temperature
- RTR_CELLAR[5:0]: Cellar's temperature
- nWSC: Cellar's Windows switch
- nWSM: Manor's Windows switch
- ID_SW: if 0 is Cellar, else is Manor.

Outputs:

- LEDF[2]: get the input value of LEDM or LEDC in base of ID_SW
- Out[12]: Is a value of DTF, RTR in base of ID_SW value for 7 Segment Driver.

The job of the Display Led Driver module is to define the **LEDF** and out outputs, based on the **ID_SW** value.

It contains 2 mux, one for **LED** output and the other for **DTF**, **RTR**, **nWS**, signal, and their resulting in an exit to the registry.

The out signal is a concatenation of input signals which are output.

5.8.3) Display Led Driver Testbench

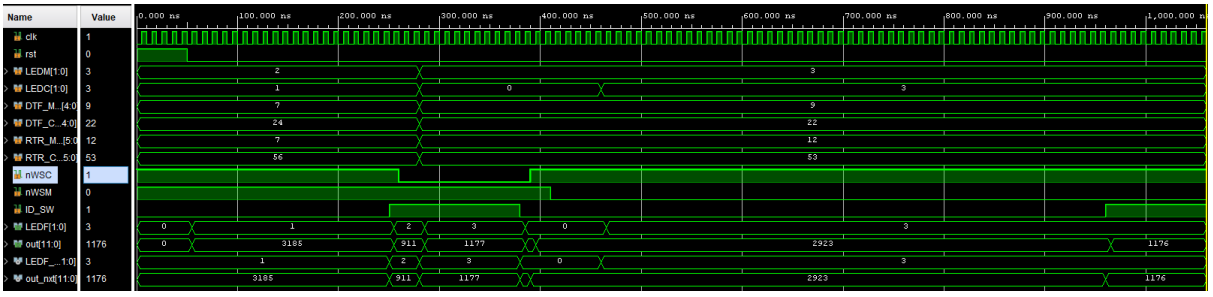


Figure 42- Display Led Driver Testbench

In the testbench is shown when **ID_SW** value (and obviously an input signal variation), the **LED_F** and out take the correspondent input, in tb, for example it can see, at the **start** (after reset high) the combination of output signal get the **LED_CELLAR**, **DTF_CELLAR**, **RTR_CELLAR** and **nWSC** signal, because **ID_SW** is 0.

After, when **ID_SW** is 1, the output signal gets the **LED_MANOR**, **DT_MANOR**, **RTR_MANOR**, and **nWSM**.

If **LEDC**=11, you'll see in Cellar and Manor that the alarm in the Cellar is on (shown in the testbench with the ID_SW=0 and ID_SW=1).

5.9) Led Driver

5.9.1) Led Driver Description

Respect to the Led_Driver module, we have added a new pattern, when LED = 11 all the leds are on.

5.9.2) Led Driver Testbench

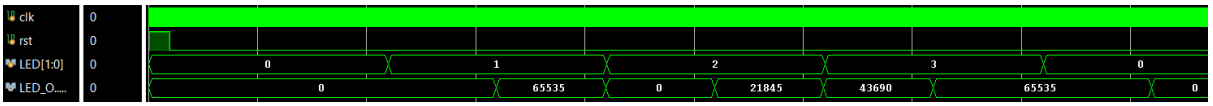


Figure 43- Led Driver Testbench

This testbench shows every **LED**'s scenario (0,1,2,3).

5.10) Orologio

5.10.1) Orologio Description

Inputs:

- clk: Clock
- rst: Reset
- en: tells the timer to start

Outputs:

- ora: it increments every 60 minutes.
- minuti: it increments every time the hit comes from the timer.
- done: it's 1 if 'ora' reaches 6 and restarts the time scanner.

It's a time scanner that starts when the Wine Cellar is configured.
It returns **ora** and **minuti** and a signal **'done'** when it reaches the 6th hour, at that moment the Time Scanner restarts.

5.10.2) Orologio Testbench

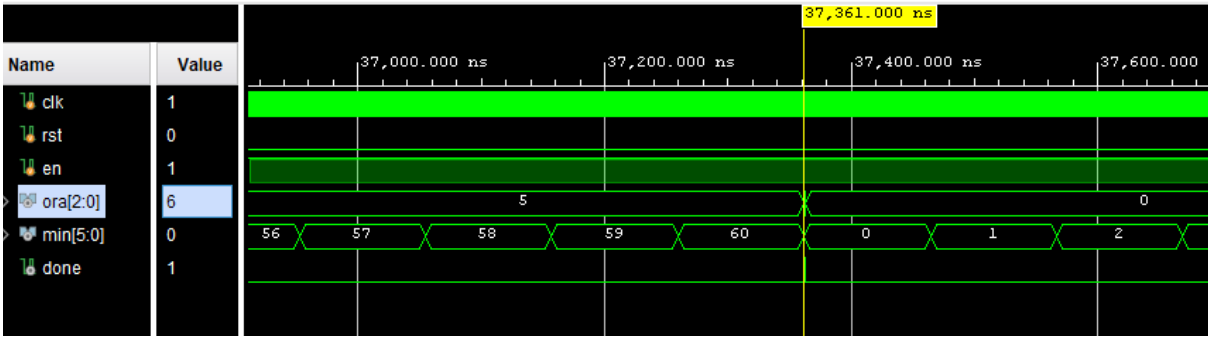


Figure 44-Orologio Testbench

As we can see the module counts as long as the **enable** is high, and set **'done'** at 1 at the 6th hour. Then it restarts.

5.11) Temp Hour

5.11.1) Temp Hour Description

Inputs:

- clk: Clock
- rst: Reset
- in_var[4:0]: is basically the OPcode that comes from the VAR_driver
- RTRM[5:0]: current temperature in the manor
- RTRC[5:0]: current temperature in the wine cellar
- h[2:0]: hour received from 'orologio'
- m[5:0]: minutes received from 'orologio'
- hitC: if 1 there was a variation in the cellar
- hitM: if 1 there was a variation in the manor

Outputs:

- where: if 1 the variation was in the cellar, if 0 it was in the manor
- OPCODE: tells to the UART which type of variation we had
- ora[2:0]: hour in the instant of the variation
- minuti[5:0]: minutes in the instant of the variation
- RTRF[5:0]: current temperature where the variation is happened.

This module chooses the values of **where,ora,minuti,RTRF,OPCODE** depending on the **in_var** and **hitC/M** that tells him where the variation was.

If there is no variation the module doesn't update the registers.

hitC and **hitM** could be 1 together, in that specific case (unlike) the priority is on the Manor.

5.11.2) Temp Hour Testbench

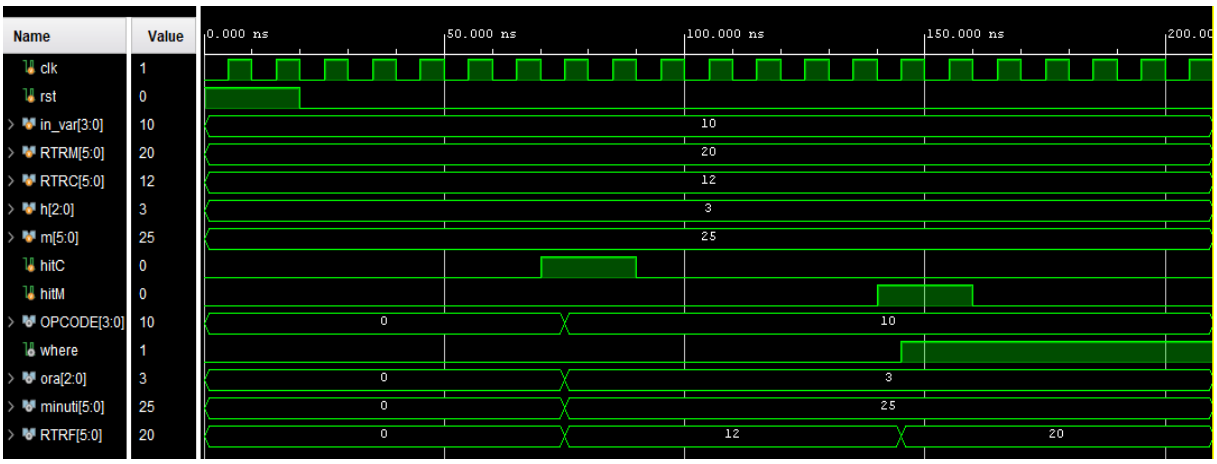


Figure 45-Temp Hour Testbench

In the testbench we can notice that when **"hitC"** is 1 the system returns the current temperature of the cellar(**RTRC**), the values from the orologio, the **OPCODE** and **where**(0 for cellar).

When the **'hitM'** is 1 these values change, becoming the Manor's ones.

FIFO

6.1) Fifo Description

The FIFOs we have used to store all the information about Manor and Cellar are the same modules we have seen during the ESD course. So to implement the burst mode during a reading we did not modify the module itself but, how we will explain in SPI chapter, we just remove data from FIFO until it is completely empty. In this way, whenever we read from a FIFO, in only one communication we read all the content from the memory.

We used FIFO modules to store the data for Manor and Cellar, and another FIFO module (**auxiliary FIFO**). This last module is the module where we store the data we read from Manor and Cellar FIFOs, before sending the same information to the log screen.

UART

Uart communication is managed by **top_log_screen_module**.

7.1) The *top_log_screen* Module

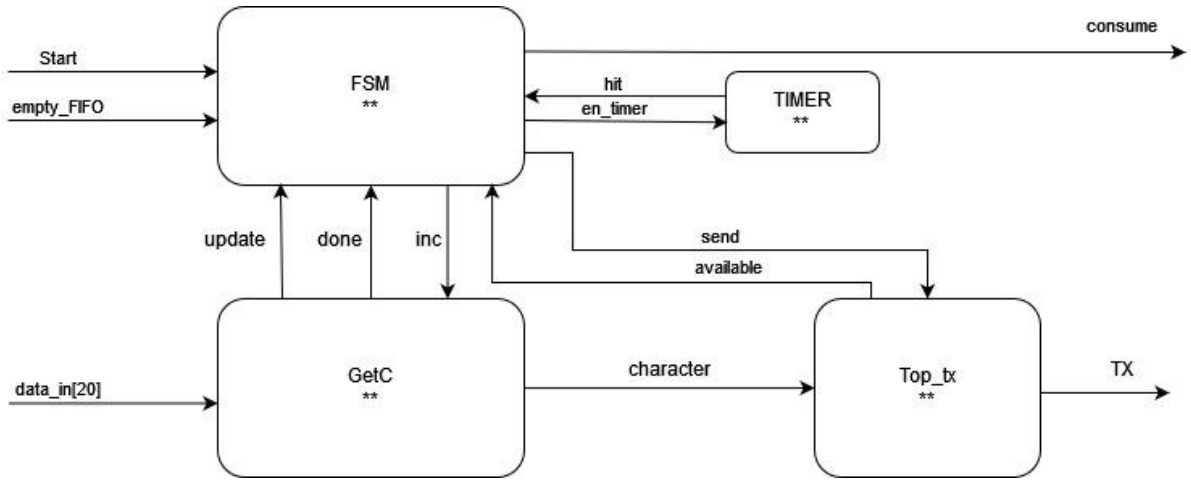


Figure 46- top_log_screen

This module contains all the logic related to the log screen. In detail, it consists of the following four modules:

1. *Top_tx*: This module is responsible for the connection with the log screen through UART.
2. *GetCharacter*: This module outputs an 8-bit signal representing the ASCII code of a character, which is sent to the log screen via UART.
3. *fsm_logScreenConnection*: It manages all the logic between the *top_tx* and *GetCharacter* modules.
4. *timer_param:* a parametric timer that rise the **hit** signal after three clock cycles when **en_timer** is high.

The UART is similar to what we studied in class, with one small difference: the busy signal is replaced by the available signal, which goes high when the **en** (enabling transmission of data) and **transmit_int** (sampling record required by the producer) signals are 0.

We will discuss the other two modules in detail later on.

The module interface is defined as follows:

Input:

- **data_in[20]** → Signal coming from the auxiliary FIFO.It contains information about the event that has happened, the place, the time and the temperature.

- **start** → Signal coming from CommsDispatcher, tells that the FIFOs of Manor and Wine Cellar have been read and now are empty.
- **emptyFIFO** → Signal coming from CommsDispatcher, tells that the auxiliary FIFO is empty.

Output:

- **consume** → Signal outputted to the auxiliary FIFO, it consumes data stored in the FIFO.
- **TX**→ is the serial port of transmission of the UART.

The input **start** signal goes high when the FIFOs of the manor and the wine cellar are empty.
The input ***empty_FIFO*** signal goes high when the auxiliary FIFO, which contains all the data read in burst mode from the previously described FIFOs, is empty.The input signal ***data_in*** contains all the necessary data about the state of our system. It is composed as follows:

- The first 4 bits represent the opcode, it contains the address of the memory we need to read. Each word of the memory contains a different message describing the state of the system. We have eleven different types of messages (which will be shown in the **romParam** module description).
- One bit identifies the place, one if the place is "wine cellar" and zero if it is "Manor".
- Three bits identify the hour in which the event occurs.
- Six bits identify the minutes in which the event occurs.
- The last six bits identify the temperature of the place in which the event occurs.

7.2) The *GetCharacter* Module

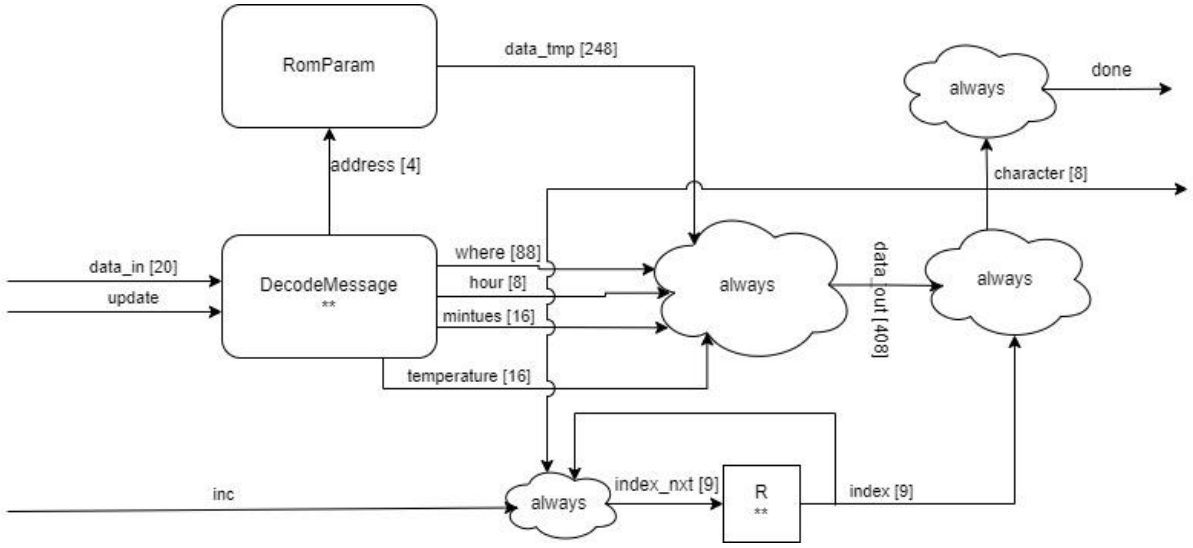


Figure 47- GetCharacter Module

As mentioned before this module outputs an 8-bit signal representing the ASCII code of a character, which is sent to the log screen via UART.

Input:

- **data_in[20]** → Signal coming from the auxiliary FIFO.It contains information about the event that has happened, the place, the time and the temperature.
- **update** → Signal coming from the "fsm_logScreenConnection", tells that the **data_in** has to be decoded.
- **inc** → Signal coming from the "fsm_logScreenConnection", tells that the index has to be increased.

Output :

- **done** → Signal outputted to the FSM, it communicates that we reach the last character of the message.
- **character[8]** → Signal outputted to the UART, it contains the character to be sent to the Log Screen.

The input signal **inc**, when high, increments the local signal *index* by one. This *index* is used to iterate over the message read from the ROM.

The input signal **update**, when high, initiates the decoding of the message, as will be explained in detail in the description of the "**decodeMessage**" module.

The output signal **character** contains an 8-bit representation of the ASCII code of a character.

The module contains two sub-modules:

- The "**romParam**" module, that will read the message from the memory and save it in a local signal called **data_tmp**.
- The "**decodeMessage**" module that will decode the information contained in the **data_in** signal.

Once that we have decode the message the **data_out** signal, that contains the final message, will be composed as follows:

data_out = {where, 16'b0011101000100000, hour, 8'b00111010, minutes, data_tmp, temperature, 8'd12}

Here there is a description of how this message is composed:

-signal **where**: the first bits are used to identify the place: "Manor" or "Wine Cellar",

-then there are 16 bits that represent the two character ": " (colon and space),

-signal **hour**: the character that represent the hour (a value between 0 and 5),

-then there are 8 bits that represent the space character,

-signal **minutes**: the two characters that represent the minutes (a value between 0 and 59),

-signal **data_tmp**: it contains the value that has been read from the ROM (there are eleven different messages saved),

-signal **temperature**: the two characters that represent the temperature,

-at the end we have the bits for the new line.

Then each character of the message will be access thanks to the following code:

```
always @(data_out, index)
begin
   character = data_out[407-8*index -: 8];
end
```

Index will be incremented when the **inc** signal rises and when the character is equal to eleven, that is the ASCII code to start a new line,

the done signal will go up. Indeed, each message ends with the new line character. The **done** signal will be an input of the final state machine described in the "**fsm_logScreenConnection**" module.

7.2.1) The RomParam Module

This module contains a simple memory that serves as a parametric ROM. It stores eleven different messages with the following addresses and corresponding information:

- Address zero: Indicates that the alarm in the wine cellar has been triggered.
- Address one: Indicates that the alarm has been restarted.
- Address two: Indicates that the window has been opened.
- Address three: Indicates that the window has been closed.
- Address four: Indicates that the system has been started.
- Address five: Indicates that the system has been stopped.
- Address six: Indicates that we are in the summer season.
- Address seven: Indicates that we are in the winter season.
- Address eight: Indicates that it is daytime.
- Address nine: Indicates that it is nighttime.
- Address ten: Indicates that the system has been configured.

7.2.2) The DecodeMessage Module

This module decodes the data contained in the **data_in** signal.

Input:

- **data_in[20]** → Signal coming from the auxiliary FIFO.It contains information about the event that has happened, the place, the time and the temperature.
- **update** → Signal coming from the "fsm_logScreenConnection", tells that the **data_in** has to be decoded.

Output:

- **address** → Signal containing the address from which to read from the memory.
- **where**→ Signal containing the ASCII code of the place: can be "Manor" or "Wine Cellar".
- **hour** → Signal containing the ASCII code of the hour (from 0 to 5).
- **minutes** → Signal containing the ASCII code of the minutes (from 0 to 59).
- **temperature** → Signal containing the ASCII code of the current temperature.

The **update** signal, when high, will enable the decoding of the **data_in** messages as follows:

- The first four bits (data_in[19:16]) are saved in the **address** signal,
- A bit (data_in[15:15]) is saved in the local **where_tmp** signal, which indicates the place where the event occurs.
- Three bits (data_in[14:12]) are saved in the local signal **hour_tmp**, which represents the decimal hour value (ranging from zero to five).
- Six bits (data_in[11:6]) are saved in the local signal **minutes_tmp**, which represents the decimal minutes value (ranging from zero to fifty-nine).
- Six bits (data_in[5:0]) are saved in the local signal **temperature_tmp**.

Then, the output signal **where** is assigned the corresponding ASCII codes. If it is equal to zero, it contains the code corresponding to "Wine cellar", and if it is equal to one, it contains the code corresponding to "Manor".

The output signal **hour** is obtained by converting the decimal number to the corresponding ASCII code. This is achieved by simply adding 48 to the value contained in **hour_tmp** since the ASCII code for the number zero is 48.

The output signal **minutes** is calculated using two auxiliary signals, **aux_min0** and **aux_min1**. **aux_min0** contains the ASCII code of the second digit, while aux_min1 contains the ASCII code of the first digit. The last digit of the minutes is stored in **aux_min0** by calculating the remainder after dividing by 10. Then, the minutes are divided by 10 to obtain the first digit, which is saved in **aux_min1**. The **minutes** signal is a concatenation of the two auxiliary signals, properly converted to the ASCII code (like the calculation for the **hour** signal).

The output signal **temperature** is calculated using an algorithm like the one used for the **minutes** signal.

7.2.3) Testbench

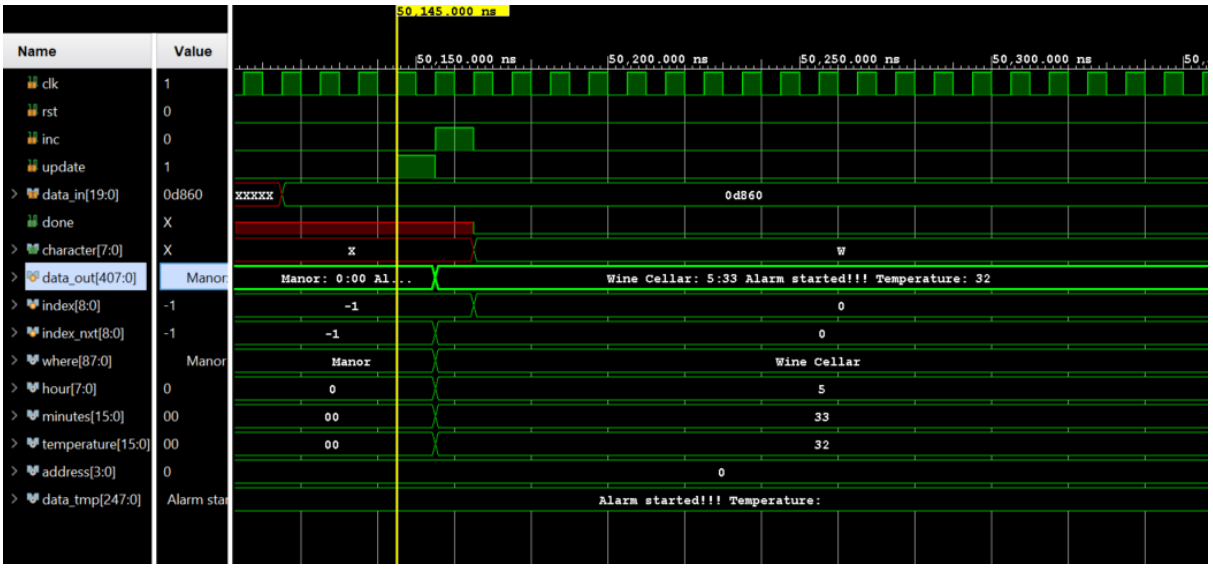


Figure 48-GetCharacter Testbench

In figure 48 we notice that when update is 1 the **data_in** signal is decoded and the **where**, **hour**, **minutes**, **temperature** and **address** are updated and data_out, that contains the message to be sent, is composed.

Firstly, the index is equal to -1 and when inc is 1 it increments.

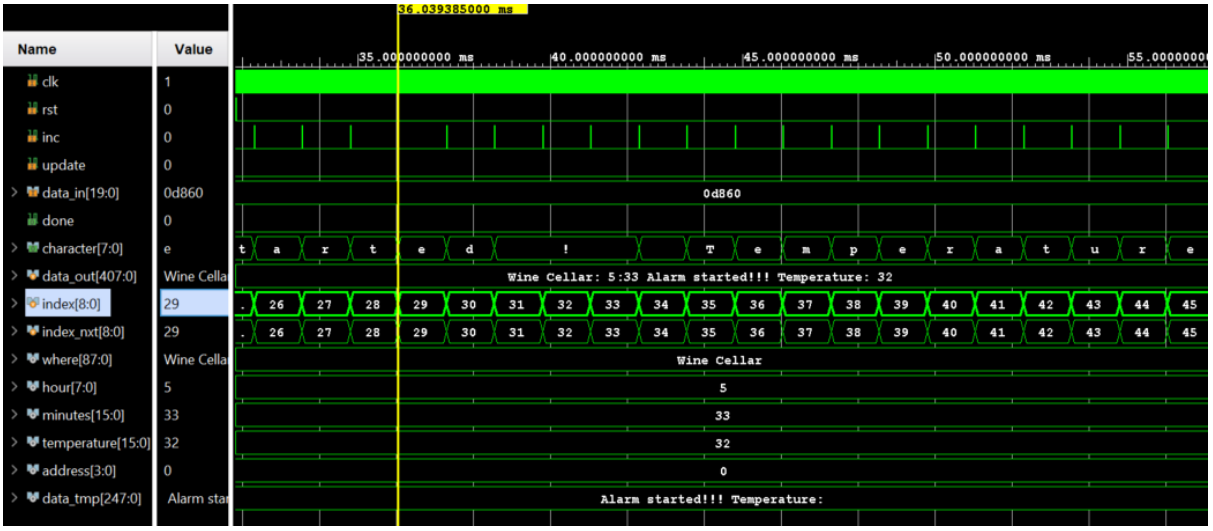


Figure 49-GetCharacter Testbench 2

In figure 49 we notice that every time that inc signal is 1 it increments the index_nxt and a new character is extract from data_out.

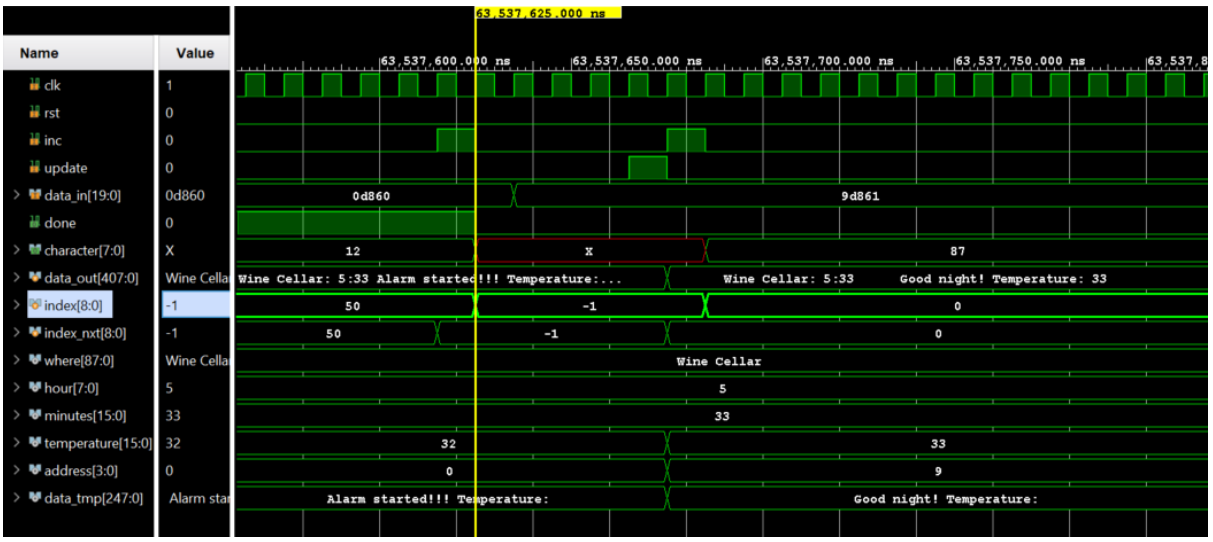


Figure 50-GetCharacter Testbench 3

In figure 50 we noticed that when the last character is reached (character is equal to 12 decimal) the index_nxt is reset to -1. No character is read from data_out until a new update signal comes so that all the values are updated according to the new data_in and a new message can be read, character by character, from data_out.

7.3) The fsm_logScreenConnection Module

Input:

- start → Signal coming from CommsDispatcher, tells that the FIFOs of "manor" and "wine cellar" are both empty.
- stop → Signal coming from CommsDispatcher, tells that the auxiliary FIFO is empty.
- done → Signal coming from GetCharacter, tells that a whole message has been sent correctly to the UART.
- available → Signal coming from UART, tells that the UART is available.
- hit → Signal coming from timer_param, tells that the target has been reached.

Output :

- send → Signal coming out from the FSM, it starts the transmission of a character in the UART.
- en_timer → Signal coming out from the FSM, it enables the timer that will produce the *hit* signal.
- inc → Signal coming out from the FSM, it increases the index in the getCharacter module.
- consume → Signal coming out from the FSM, it consumes a word from the auxiliary FIFO.

- update → Signal coming out from the FSM, it enables the decode of the data received from the auxiliary FIFO in the DecodeMessage module.

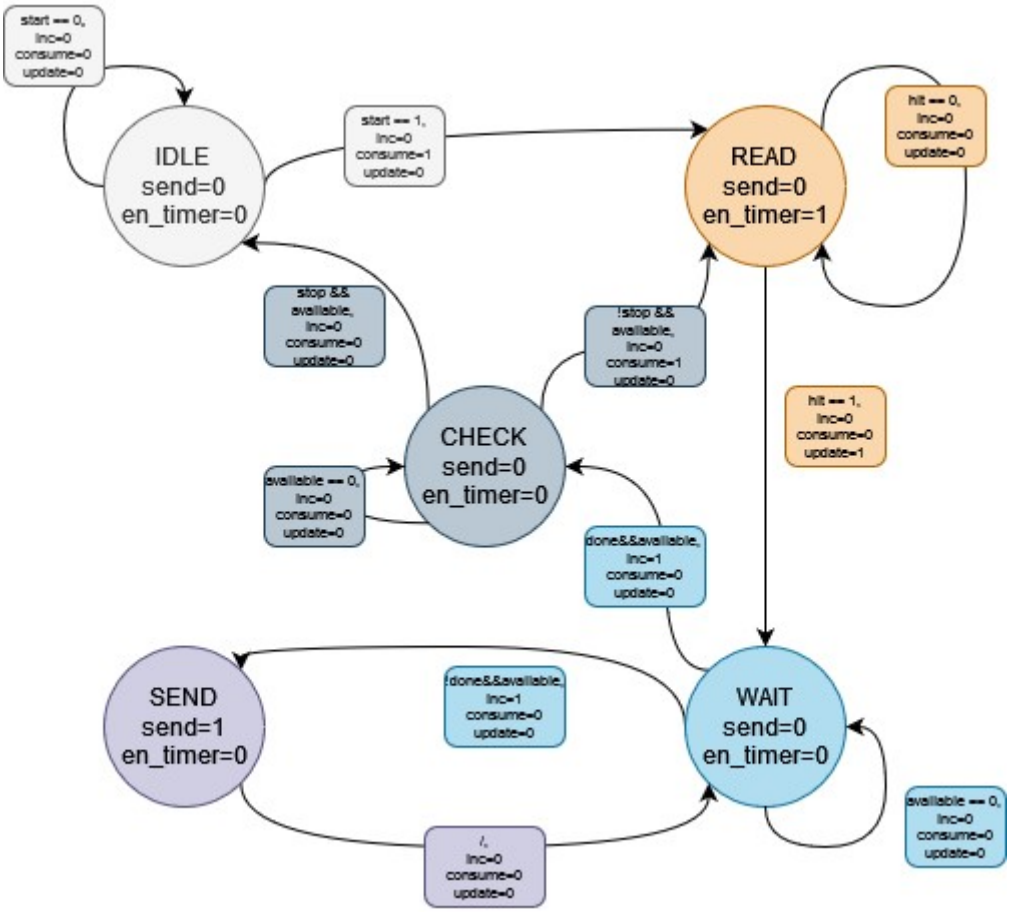


Figure 51-fsm_logScreenConnection Diagram

7.3.1) Fsm_logScreenConnection Tables

FSM_LogScreenConnection

Future State tables

Inputs are in order : {start, stop, done, available, hit}.
Mealy's output are in order : {inc, consume, update}

	IDLE	**READ**	**WAIT**	**SEND**	**CHECK**
00000	IDLE (000)	READ (000)	WAIT (000)	WAIT (000)	CHECK (000)

00001	IDLE (000)	WAIT (001)	WAIT (000)	WAIT (000)	CHECK (000)
00010	IDLE (000)	READ (000)	SEND (100)	WAIT (000)	READ (010)
00011	IDLE (000)	WAIT (001)	SEND (100)	WAIT (000)	READ (010)
00100	IDLE (000)	READ (000)	WAIT (000)	WAIT (000)	CHECK (000)
00101	IDLE (000)	WAIT (001)	WAIT (000)	WAIT (000)	CHECK (000)
00110	IDLE (000)	READ (000)	CHECK (100)	WAIT (000)	READ (010)
00111	IDLE (000)	WAIT (001)	CHECK (100)	WAIT (000)	READ (010)
01000	IDLE (000)	READ (000)	WAIT (000)	WAIT (000)	CHECK (000)
01001	IDLE (000)	WAIT (001)	WAIT (000)	WAIT (000)	CHECK (000)
01010	IDLE (000)	READ (000)	SEND (100)	WAIT (000)	READ (010)
01011	IDLE (000)	WAIT (001)	SEND (100)	WAIT (000)	READ (010)
01100	IDLE (000)	READ (000)	WAIT (000)	WAIT (000)	CHECK (000)
01101	IDLE (000)	WAIT (001)	WAIT (000)	WAIT (000)	CHECK (000)
01110	IDLE (000)	READ (000)	CHECK (100)	WAIT (000)	READ (010)
01111	IDLE (000)	WAIT (001)	CHECK (100)	WAIT (000)	READ (010)
10000	READ (010)	READ (000)	WAIT (000)	WAIT (000)	CHECK (000)
10001	READ (010)	WAIT (001)	WAIT (000)	WAIT (000)	CHECK (000)
10010	READ (010)	READ (000)	SEND (100)	WAIT (000)	IDLE (000)

10011	READ (010)	WAIT (001)	SEND (100)	WAIT (000)	IDLE (000)
10100	READ (010)	READ (000)	WAIT (000)	WAIT (000)	CHECK (000)
10101	READ (010)	WAIT (001)	WAIT (000)	WAIT (000)	CHECK (000)
10110	READ (010)	READ (000)	CHECK (100)	WAIT (000)	IDLE (000)
10111	READ (010)	WAIT (001)	CHECK (100)	WAIT (000)	IDLE (000)
11000	READ (010)	READ (000)	WAIT (000)	WAIT (000)	CHECK (000)
11001	READ (010)	WAIT (001)	WAIT (000)	WAIT (000)	CHECK (000)
11010	READ (010)	READ (000)	SEND (100)	WAIT (000)	IDLE (000)
11011	READ (010)	WAIT (001)	SEND (100)	WAIT (000)	IDLE (000)
11100	READ (010)	READ (000)	WAIT (000)	WAIT (000)	CHECK (000)
11101	READ (010)	WAIT (001)	WAIT (000)	WAIT (000)	CHECK (000)
11110	READ (010)	READ (000)	CHECK (100)	WAIT (000)	IDLE (000)
11111	READ (010)	WAIT (001)	CHECK (100)	WAIT (000)	IDLE (000)

Figure 23- Next state fsm_logScreenConnection

Moore Outputs Table

STATE	SEND	EN_TIMER
IDLE	0	0
READ	0	1
WAIT	0	0
SEND	1	0
CHECK	0	0

Figure 24- Moore's output fsm_logScreenConnection

Functioning

The FSM starts in the **IDLE** state, in this state every output is at 0 until the **start** signal goes to 1, when the FSM will go into the **READ** state. During this change of state, the **consume** signal rises.
In the **READ** state the en_timer is high. The FSM will move to the next state, **WAIT**, only when **hit** is 1 and will raise the **update** signal.
The FSM will stay in the **WAIT** state until the available signal rises. At this point if **done** is 1 it will go into **CHECK** state, otherwise into the **SEND** state. In both cases the **inc** signal rises.
All the outputs in the **WAIT** signal are equal to 0.
In the **SEND** state the send signal rises and at the next clock cycle the FSM will move into **WAIT**.
In the **CHECK** state all outputs are equal to 0. The FSM will stay in this state until **available** is high. Then it will move to **IDLE** if stop is 1 otherwise it will go to **READ** rising the **consume** signal.

7.3.2) TestBench

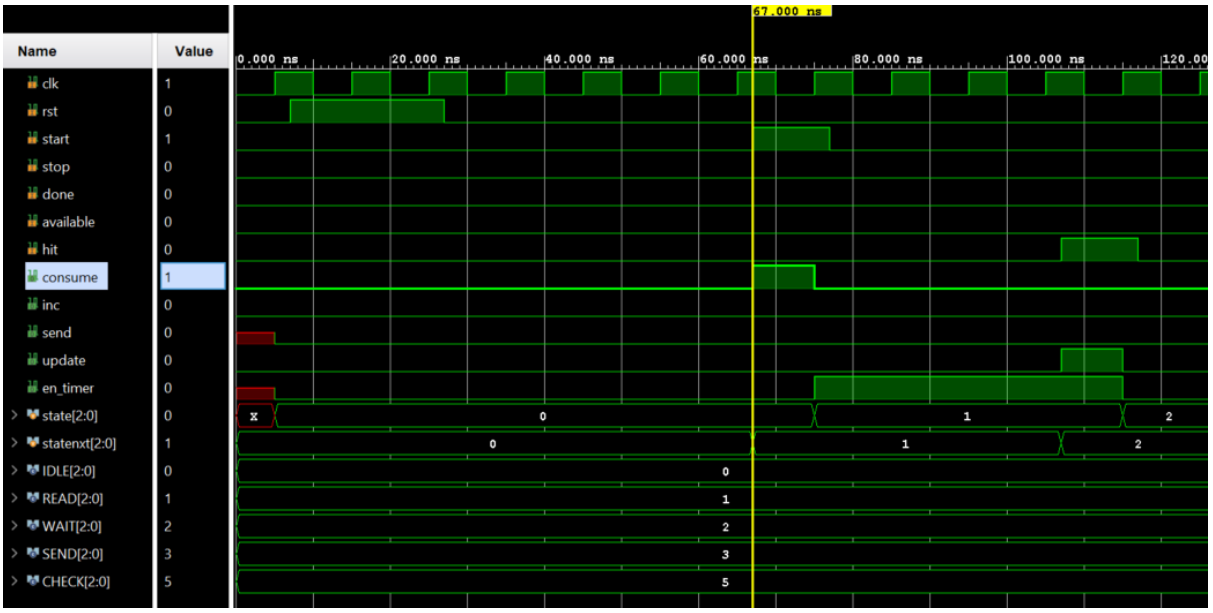


Figure 52-fsm_logScreenConnection Testbench

In fig. 52 we can see how the FSM works. It will move from **IDLE** to **READ** only when **start** is 1. In the transition the output **consume** is raised.
Then we will stay in the **READ** state for four clock cycles: the **en_timer** in this state is high and we will move to the following state, **WAIT**, only when **hit** is high.

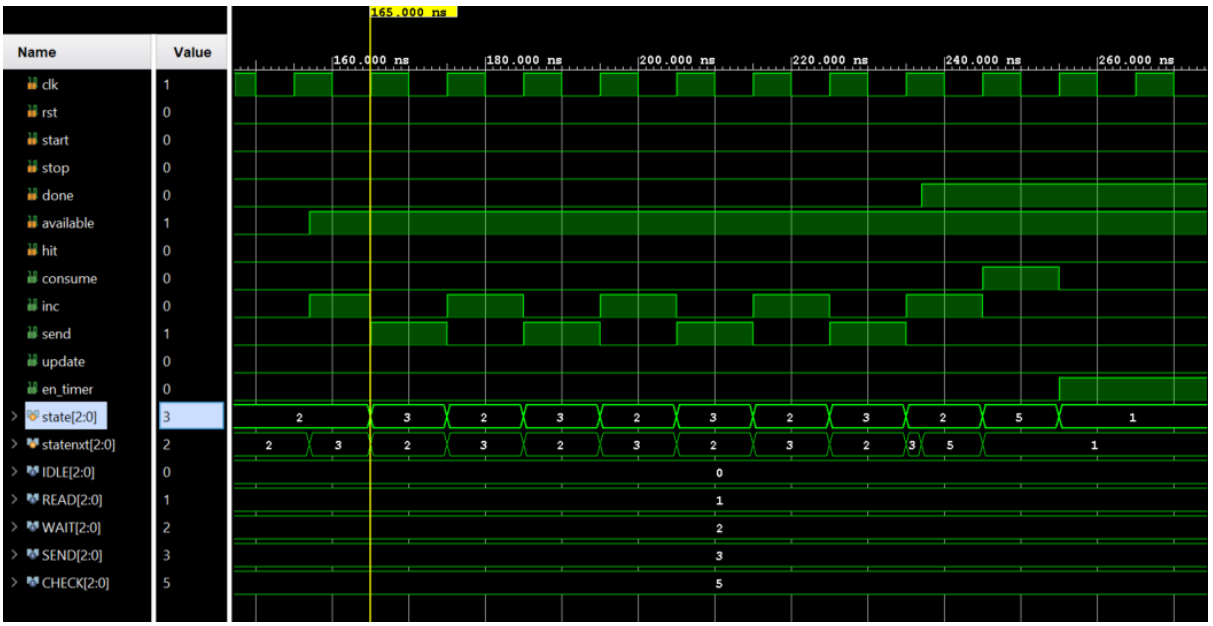


Figure 53-fsm_logScreenConnection Testbench 2

In fig. 53 we can notice then in the meanwhile that **available** is 1 and **done** is 0 we enter in a loop between the **WAIT** and the **SEND** state. The **inc** signal will rise during the transition between **WAIT** and **SEND** and the send signal will be high in **SEND** state.

When **done** is 1 and we are in the **WAIT** state we move to the **CHECK** state.
Here **available** is high and **stop** is low, so we move to the **READ** state. During this transition **consume** rises.

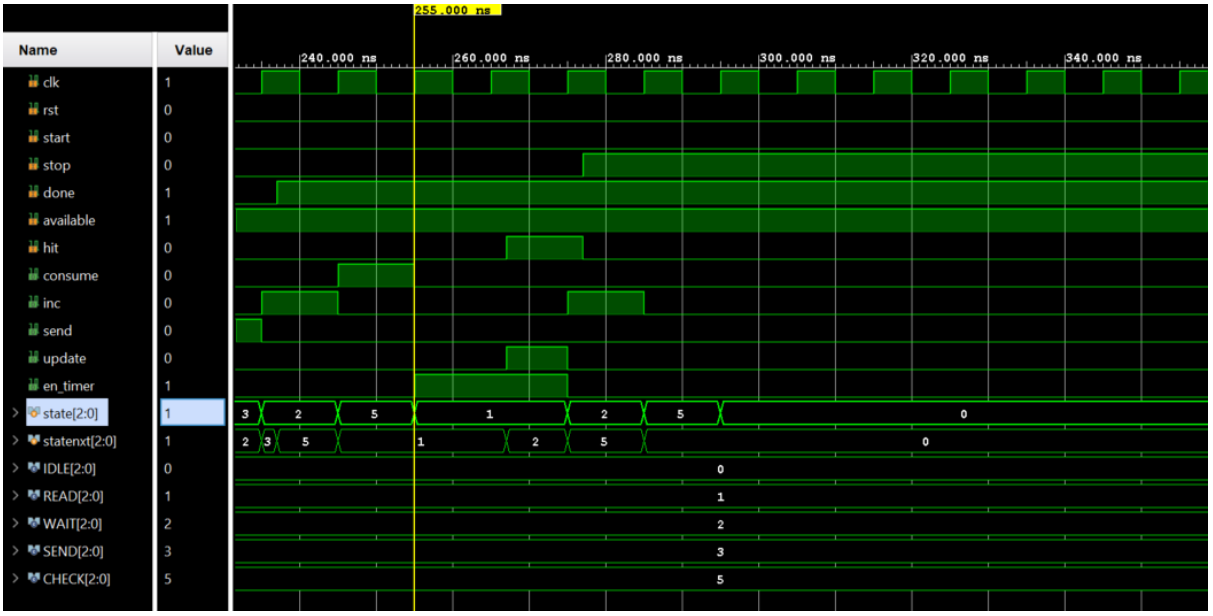


Figure 54-fsm_logScreenConnection Testbench 3

Instead, in figure 54 we notice that from **CHECK** state we move to **IDLE** if **stop** is 1.

SPI

8.1) Introduction

The SPI has been implemented in the system in order to communicate with FIFOs peripheral **Manor** and **Cellar** and **7 Segment Driver**. Furthermore it has been used to acquire signals for **Restart**, **Switch Start** and **Windows Switches**. For all the testbenches we have implemented we used SPI in mode 2 and we have considered the peripherals frequency equals to 25Mhz.

8.1.1) SPI Communication protocol

As we will describe in the next modules, FIFOs peripherals need, because we have to write and read from them, a communication protocol to work. In particular the Master module send along MOSI line MEM or LEG signal to start communication. If the Slave receives from MISO line the LEG signal all the next data that will receive (DUMMY) will be used to shift the SPI register in order to send back to the Master the FIFO's content, through MISO line. Otherwise if the Slave receives from MOSI line the MEM signal, the next data will be memorized to the FIFO.

8.2) Parametrization and Slave Selector

What have we changed to parametrize the SPI?

8.2.1) SPI MASTER MODULE

In SPI Master we have changed :

- Add DATA_SIZE parameter in order to parametrize i_TX_Byte and o_RX_Byte.
- Because of this we changed dimension of prod_data signal, count_data_S and count_data_R (both equal to another parameter we have defined called BIT_DATA = $clog2(DATA_SIZE)).
- About the Reception Channel, where we have the shift register we have assigned to o_RX_Byte the correct value to parametrize the shift register itself.

```
always@(posedge clk, posedge rst)
if (rst==1'b1) o_RX_Byte<={DATA_SIZE{1'b0}};
else if (enableRX==1'b1)
    o_RX_Byte<={o_RX_Byte[DATA_SIZE-2:0],i_SPI_MISO};
```

What about the **submodules** :

- **top_clk_gen_mode**: Here we assigned new value to MAX_EDGE_GEN parameter. In particular this parameter identifies the total number of edges, so it has to be equal to DATA_SIZE * 2.
- **clk_gen_m** : Here we changed nothing, in fact clock generation is independent by data size changing.

- **adapter_M** : Changing MAX_EDGE_GEN parameter we have in fact modified this module, because changing the parameter we have just mentioned, we decide the number of bits to send.
- **register** : We just pass as parameter DATA_SIZE.
- **fsm_SPI_master** : no change.
- **Ready_Gen** : no changes. In fact this module is used to let the SPI accept new data from Producer, and parametrizing the other modules, we have automatically parametrized this module too.
- **counter** : We have just passed the correct parameters MAX = DATA_SIZE and N_BIT = BIT_DATA.

8.2.2) SPI SLAVE MODULE

In SPI Slave we have changed :

- Add DATA_SIZE parameter here too in order to parametrize i_TX_Byte and o_RX_Byte.
- With DATA_SIZE parameter we have parametrized RX_Byte and TX_Byte.
- And with BIT_SIZE we have parametrized RX_cnt and TX_cnt.
- In Reception Channel we have parametrized the shift register, exactly as we did in SPI Master.
- We changed the data received counter, in fact we have to reasign to RX_cnt the initial value (DATA_SIZE-1) when RX_cnt == 0. We have to do this because if DATA_SIZE is not multiple of 2, RX_cnt does not come back to the initial value automatically.
- We have followed the same logic for data transmitted counter. Where we decrease the value of TX_cnt at the same condition as before, but if TX_cnt > 0 too and we reset TX_cnt to the initial value (DATA_SIZE-1) when it is equal to 0.

What about the **submodules** :

- **modeAdapter** : Here we parametrized with DATA_SIZE TX_cnt and RX_cnt. Furthermore for the modes 0 and 2 we enable the reception channel when we reach the value TX_cnt = DATA_SIZE-2 and disable it when we reach RX_cnt = DATA_SIZE-1. And, always for modes 0 and 2 we enable the transmission when RX_cnt = DATA_SIZE-1.
- **counter** : Same changes of SPI Master.

8.2.3) SLAVE SELECTOR

SPI MASTER

In SPI Master we have added an input called slave_selector, that we use to determinate which slave we have to enable. Because we have two spi master, one of which has only one slave (so we have just always assigned SS = 0 to that slave, and we ignore the corresponding master ss output) we have computed the slave selector only for the SPI MASTER-FIFO-DISPLAY. In particular based on slave selector, because we have 3 slaves, we assign to ss signal different values based on the slave selector input.

SPI SLAVE

Here we added an input called SS, that represent the value of the slave selector. In particular we have declared two new signals : ss_o_SPI_MISO and ss_i_SPI_Clk. These signals represent respectively the signals o_SPI_MISO and i_SPI_Clk of the original SPI, and we use these new signals inside the module instead of the original two. In fact what we do is that when SS = 0 we assign to ss_i_SPI_Clk input clock (i_SPI_Clk). In this way everything works normally. But if SS != 0, it means that the slave is not selected so the internal clock of the system (ss_i_SPI_Clk) will be equal to W_CPOL (the value of the clock polarity selected). In this way we maintain the clock at the initial state, in this way the system is "freezed".
For the output MISO instead, if SS = 0 we just assign to the output the effective value computed in the internal module (ss_o_SPI_MISO), else if SS != 0 we disconnect the output, in this way no information arrives to the SPI Master.

8.3) SPI Interface for Buttons (Interface_master_slave_buttons)

Interface_master_slave_buttons is the module we have used in order to test the behaviour of the isolated SPI system for the communication between input and core system.

Here we have implemented the master and slave interface connected to respective SPI in order to communicate buttons and switch values to the system. We have two modules :

- **Master_buttons** (placed in Top_Thermostat) : Simple module that takes as input **start**, **restart** and **windows[2]** and send as output to the i_TX_Byte input of the spi master data_to_send = {start, restart, windows}. The corresponding SPI Master (MASTER_BUTTONS) in SPI_top.v module takes as input i_TX_DV always the value 1, because it must always be ready to send data.

- **Slave_buttons** (placed in Comms Dispatcher): Simple module as well that takes as input the o_RX_Byte from SPI Slave and o_RX_DV. When o_RX_DV signal of the corresponding SPI_Slave arrives send as output the signal data_stored [4], that contains the value of {start, restart, windows}. The corresponding SPI Slave is characteryzed by i_TX_DV and i_TX_Byte equals to 0 because it does not have to send back any signal to the SPI Master.

8.3.1) Testbench

The module Interface_master_slave_buttons we have used for testbenches is characteryzed by start, restart and windows [2] as input and the corresponding output that came out from Slave connected to SPI Slave.

The testbenches down here show that everything works fine. When an input signal changes, the variation is transmitted along the SPI MOSI and reach the Slave where then the data will be unpacked.

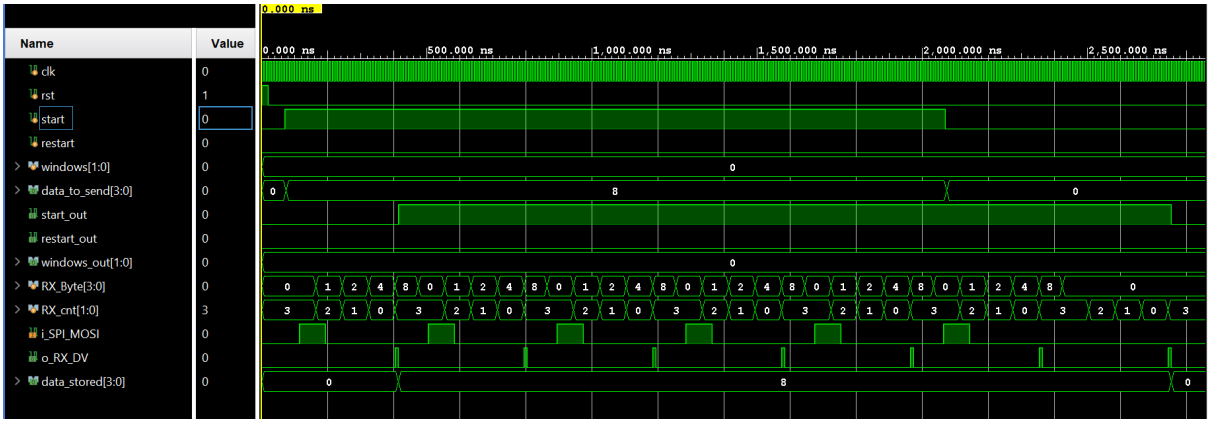


Figure 55- Interface_master_slave_buttons Testbench

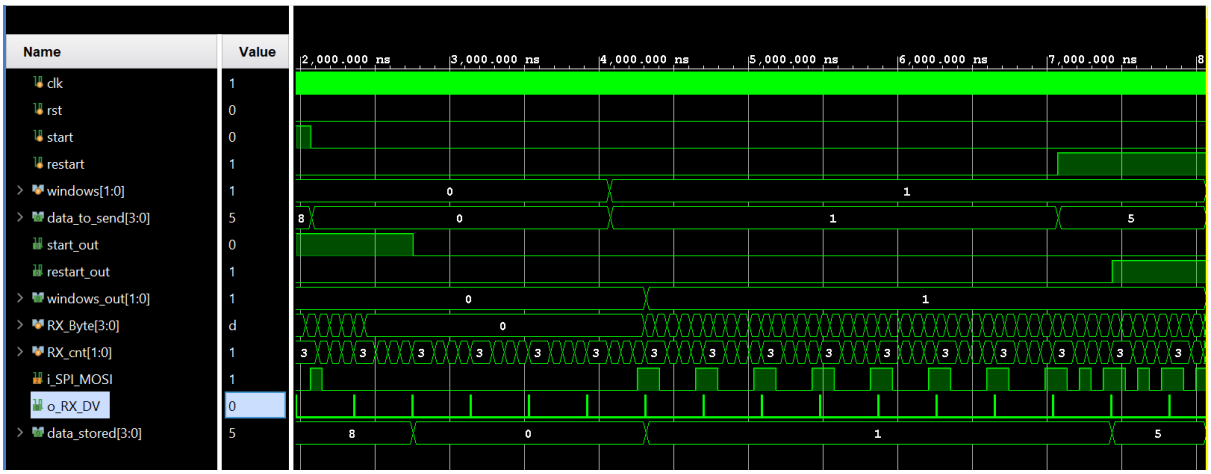


Figure 56- Interface_master_slave_buttons Testbench 2

8.4) SPI Interface FIFO-DISPLAY (Top_spi_master_Interface.v)

Top_spi_master_Interface is the module we have used to test the behaviour of the isolated SPI system for the communication with FIFOs and 7SD.

This represents the SPI Interface that manages the communication with FIFOs and 7 Segment Driver. The default size for the FIFOs that we have decided to implement is 32. So both FIFOs have 32 memory cells available (based on SIZE_FIFO parameter). In particular it is characterized by three modules that interconnect with SPI :

- **Master_fifo_display** (placed in Comms Dispatcher) : Module that receives the data every time there is a variation in some input to store the current state of Manor and Cellar. Furthermore when the hit_read signal arrives from the global timer in the system core, this module performs a reading operation from the two FIFOs (Manor and Cellar) and store the information in auxiliary FIFO.
- **top_slave_fifo** (placed in Top Thermostat) : Module that let the system to write to and read from FIFOs (Manor and Cellar). It checks if the data that has just been arrived from SPI Slave (that has been send by SPI Master thanks to Master_fifo_display) define a reading (LEG) or writing (MEM) operation. After the operation that must be performed is defined the next information is stored in FIFO (if the operation is MEM) or send content of FIFO back to SPI Master until the FIFO itself is empty (if the operation is LEG).
- **top_slave_7SD** (placed in Top Thermostat) : Simple slave that when o_RX_DV signal arrives from corresponding SPI Slaves, it stores the information in o_RX_Byte and unpack the data to transfer the information (DTF, RTR, WS) to the 7SD.

8.4.1) MASTER FIFO DISPLAY

Module that implements the logic of the Master connected to the SPI Master for the communication with the FIFOs.

Input :

- **tx_ready** → Signal coming from the SPI Master, that communicates that the data from MISO has been completely received (linked to o_RX_DV signal from SPI), and so it can send new data.
- **rd, we** → Signal coming from System Core, that communicates that it's time to read from the FIFOs (**rd**) or it's time to write in one of the FIFO (**we**). We do not take directly this signal as input but we take the fsm relè output. We did this because to send or read data from FIFOs we want to wait for the 7 segment driver to have finished

sending data. So we need to store if in the meanwhile there has been a writing or reading request.

- **data_in [20]** → Signal coming from System Core, that represents the data that must be sent to the FIFOs.
- **DTF, RTR, WS** → Signal coming from System Core that must be transmitted to 7SD.

Output :

- **read_end** → Signal that tells the Log Screen manager that the reading from the FIFOs is finished.
- **insert** → Signal used to insert the word that has just been read from the Manor or Cellar FIFO into the **auxiliary FIFO**.
- **data_to_send [20]** → Data to send to the SPI Master.
- **ss [2]** → Slave Selector for the SPI Master, to choose the correct slave.

This module is characterized by two **sub-modules** :

- **FSM_Master_FIFO** → Finite state machine that manage the execution flow of the master and let the system to write or read from the peripherals.
- **Datapath Master-FIFO** → Datapath of the Master.

8.4.1.1) FSM_Master_FIFO

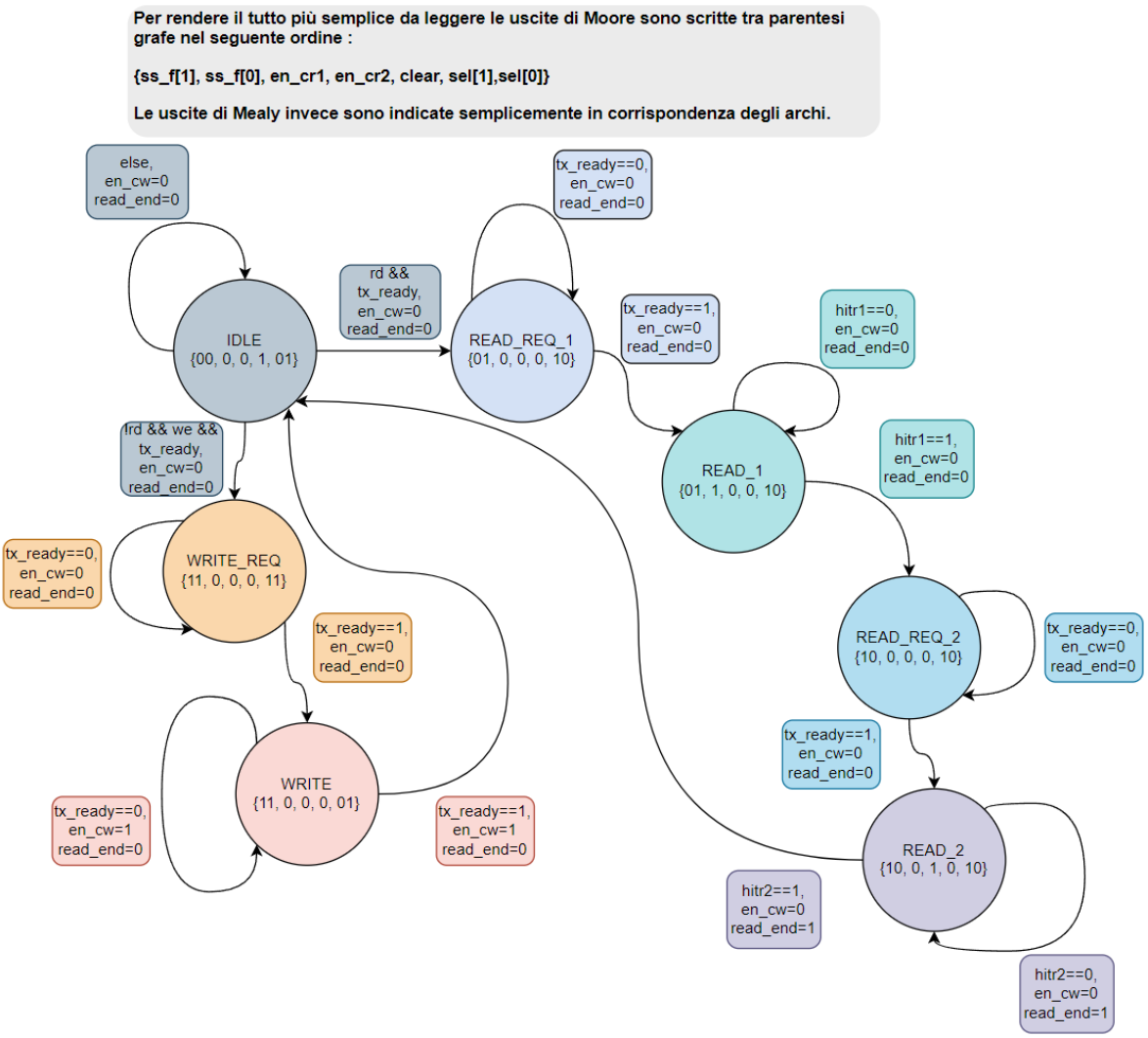


Figure 57- FSM Master FIFO

8.4.1.2) FSM MASTER FIFO TABLES

FSM_Master_FIFO

Future State Table

Inputs are in order : {rd, we, tx_ready, hitr1, hitr2}.
Mealy's output are in order : (en_cw, read_end)

	IDLE	READ_REQ_1	READ 1	READ_REQ_2	READ 2	WRITE_REQ	WRITE

00000	IDLE (00)	READ_ REQ_1 (00)	READ 1 (00)	READ_ REQ_2 (00)	READ 2 (00)	WRITE _REQ (00)	WRITE (00)
00001	IDLE (00)	READ_ REQ_1 (00)	READ 1 (00)	READ_ REQ_2 (00)	IDLE (01)	WRITE _REQ (00)	WRITE (00)
00010	IDLE (00)	READ_ REQ_1 (00)	READ_ REQ_2 (00)	READ_ REQ_2 (00)	READ 2 (00)	WRITE _REQ (00)	WRITE (00)
00011	IDLE (00)	READ_ REQ_1 (00)	READ_ REQ_2 (00)	READ_ REQ_2 (00)	IDLE (01)	WRITE _REQ (00)	WRITE (00)
00100	IDLE (00)	READ 1 (00)	READ 1 (00)	READ 2 (00)	READ 2 (00)	WRITE (00)	IDLE (10)
00101	IDLE (00)	READ 1 (00)	READ 1 (00)	READ 2 (00)	IDLE (01)	WRITE (00)	IDLE (10)
00110	IDLE (00)	READ 1 (00)	READ_ REQ_2 (00)	READ 2 (00)	READ 2 (00)	WRITE (00)	IDLE (10)
00111	IDLE (00)	READ 1 (00)	READ_ REQ_2 (00)	READ 2 (00)	IDLE (01)	WRITE (00)	IDLE (10)
01000	IDLE (00)	READ_ REQ_1 (00)	READ 1 (00)	READ_ REQ_2 (00)	READ 2 (00)	WRITE _REQ (00)	WRITE (00)
01001	IDLE (00)	READ_ REQ_1 (00)	READ 1 (00)	READ_ REQ_2 (00)	IDLE (01)	WRITE _REQ (00)	WRITE (00)
01010	IDLE (00)	READ_ REQ_1 (00)	READ_ REQ_2 (00)	READ_ REQ_2 (00)	READ 2 (00)	WRITE _REQ (00)	WRITE (00)
01011	IDLE (00)	READ_ REQ_1	READ_ REQ_2	READ_ REQ_2	IDLE (01)	WRITE _REQ	WRITE (00)

		(00)	(00)	(00)		(00)	
01100	WRITE_REQ (00)	READ 1 (00)	READ 1 (00)	READ 2 (00)	READ 2 (00)	WRITE (00)	IDLE (10)
01101	WRITE_REQ (00)	READ 1 (00)	READ 1 (00)	READ 2 (00)	IDLE (01)	WRITE (00)	IDLE (10)
01110	WRITE_REQ (00)	READ 1 (00)	READ_REQ_2 (00)	READ 2 (00)	READ 2 (00)	WRITE (00)	IDLE (10)
01111	WRITE_REQ (00)	READ 1 (00)	READ_REQ_2 (00)	READ 2 (00)	IDLE (01)	WRITE (00)	IDLE (10)
10000	IDLE (00)	READ_REQ 1 (00)	READ 1 (00)	READ_REQ_2 (00)	READ 2 (00)	WRITE_REQ (00)	WRITE (00)
10001	IDLE (00)	READ_REQ_1 (00)	READ 1 (00)	READ_REQ_2 (00)	IDLE (01)	WRITE_REQ (00)	WRITE (00)
10010	IDLE (00)	READ_REQ_1 (00)	READ_REQ_2 (00)	READ_REQ_2 (00)	READ 2 (00)	WRITE_REQ (00)	WRITE (00)
10011	IDLE (00)	READ_REQ_1 (00)	READ_REQ_2 (00)	READ_REQ_2 (00)	IDLE (01)	WRITE_REQ (00)	WRITE (00)
10100	READ_REQ 1 (00)	READ 1 (00)	READ 1 (00)	READ 2 (00)	READ 2 (00)	WRITE (00)	IDLE (10)
10101	READ_REQ 1 (00)	READ 1 (00)	READ 1 (00)	READ 2 (00)	IDLE (01)	WRITE (00)	IDLE (10)
10110	READ_REQ 1 (00)	READ 1 (00)	READ_REQ_2 (00)	READ 2 (00)	READ 2 (00)	WRITE (00)	IDLE (10)

10111	READ_REQ_1 (00)	READ 1 (00)	READ_REQ_2 (00)	READ 2 (00)	IDLE (01)	WRITE (00)	IDLE (10)
11000	IDLE (00)	READ_REQ_1 (00)	READ 1 (00)	READ_REQ_2 (00)	READ 2 (00)	WRITE_REQ (00)	WRITE (00)
11001	IDLE (00)	READ_REQ_1 (00)	READ 1 (00)	READ_REQ_2 (00)	IDLE (01)	WRITE_REQ (00)	WRITE (00)
11010	IDLE (00)	READ_REQ_1 (00)	READ_REQ_2 (00)	READ_REQ_2 (00)	READ 2 (00)	WRITE_REQ (00)	WRITE (00)
11011	IDLE (00)	READ_REQ_1 (00)	READ_REQ_2 (00)	READ_REQ_2 (00)	IDLE (01)	WRITE_REQ (00)	WRITE (00)
11100	READ_REQ_1 (00)	READ 1 (00)	READ 1 (00)	READ 2 (00)	READ 2 (00)	WRITE (00)	IDLE (10)
11101	READ_REQ_1 (00)	READ 1 (00)	READ 1 (00)	READ 2 (00)	IDLE (01)	WRITE (00)	IDLE (10)
11110	READ_REQ_1 (00)	READ 1 (00)	READ_REQ_2 (00)	READ 2 (00)	READ 2 (00)	WRITE (00)	IDLE (10)
11111	READ_REQ_1 (00)	READ 1 (00)	READ_REQ_2 (00)	READ 2 (00)	IDLE (01)	WRITE (00)	IDLE (10)

Table 25- Next State FSM Master Fifo

Moore Outputs Table

STATE	SS_F [2]	EN_CR1	EN_CR2	CLEAR	SEL [2]
IDLE	00	0	0	1	01
READ_REQ_1	01	0	0	0	10

READ_1	01	1	0	0	10
READ_REQ 2	10	0	0	0	10
READ_2	10	0	1	0	10
WRITE_REQ	11	0	0	0	11
WRITE	11	0	0	0	01

Table 26- Moore's Output FSM Master Fifo

Input :

- **rd, we, tx_ready.**
- **hitr1, hitr2** → Signals coming from the Datapath, that tell the system the reading from the Manor's FIFO and Cellar's FIFO respectively has been completed.

Output :

- **ss_f [2]** → Signal that will be used in the Datapath to determinate the correct slave selector for the SPI Master.
- **en_cr1, en_cr2** → Signal that enables the counters in the Datapath that are used to determinate the number.
- **clear** → Signal that is used to clear the reading counter for manor and cellar in the datapath.
- **en_cw** → Signal used to enable writing in Manor or Cellar FIFO.
- **read_end** → Signal to communicate that the reading from both Cellar and Manor is completed.
- **sel [2]** → Signal used to select the word that must be sent to the FIFOs.

Functioning

The system start in **IDLE** state. Here **ss_f** = 00, **clear** = 1 and **sel** = 01, so we are sending data to the 7 segment display and we are clearing the reading counter.

If **we** = 1 and **rd** is still equals to 0, we move from **IDLE** to **WRITE_REQUEST** only if **tx_ready** = 1. We decided to wait for **tx_ready** to let the 7 segment driver to finish to send the data it was sending before the arrives of **rd** or **we** signal. When we go in **WRITE_REQUEST** **ss_f** turns to 11, and we set **clear** to 0. In this state our goal is to send **MEM** word to the fifo indicated by id signal from **data_in** in datapath, with **sel** = 11. Then when

tx_ready = 1, it means we have just received a word back from SPI, it means we have sent word **MEM** to the Slave FIFO. Now we get into the **WRITE** state. Here we send to the FIFO the **data_in** present in the datapath with **sel** = 01. When we receive **tx_ready** it means we have completed to send the data, so we increment write counter for manor or cellar (with **en_cw** = 1), to understand how many writing we have computed, and then we go to **IDLE** state again.

If **rd** = 1 we have to read from both the FIFOs. So from **IDLE** state we go to **READ_REQ_1** where we send **LEG** data to the Manor's Slave in order to start to read (with **sel** = 10). Here until the reading counter is not arrived to the corresponding writing counter, we continue sending **DUMMY** (we've decided to continue to send **LEG** with **sel** equals to 10) to SPI Slave to receive all data from the FIFO in Manor. In this state we put **en_cr1** = 1, in order to enable manor's reading counter in the datapath, which value must reach manor's writing counter, to see if we have read as much as we wrote before. When **hitr1** = 1 it means that we have completed to read all data in FIFO, so we go to **READ_REQ_2**. Here we do the same procedure but for Cellar's Slave. So we go to **READ_2** when **tx_ready** = 1 and we come back to **IDLE** raising **read_end** when **hitr2** = 1.

8.4.1.3) Testbench

In this testbench all fms states are explored, and for each state we decided to try all the combinations of signals that should not interfere with execution flow, to show that those signals in a specific state are ignored by state machine.
Because the fsm is divided into two branches, one for writing and the other for reading, we've decided to divide the testbench in two steps.

Writing

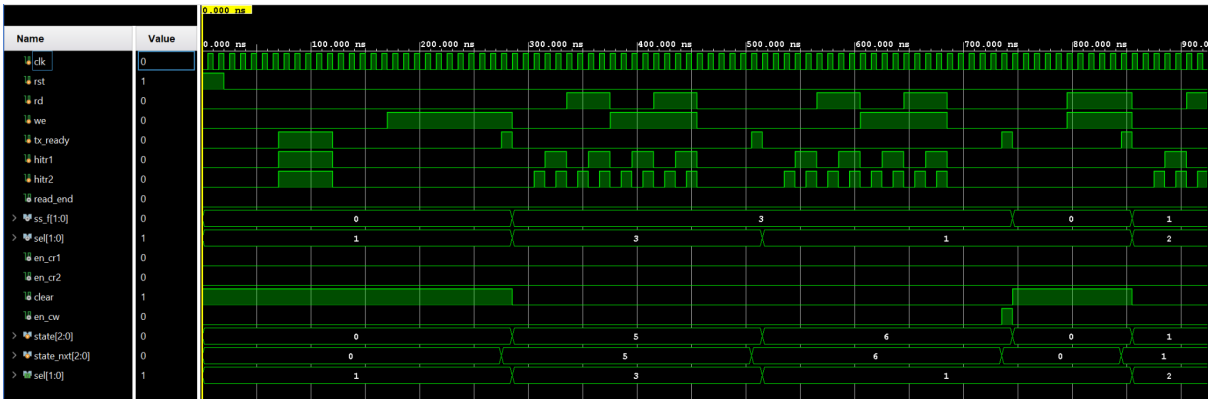


Figure 58- Writing Testbench

Here we can notice that from state 0 (**IDLE**) the FSM moves into state 5 (**WRITE_REQ**) only if **we** = 1 and **tx_ready** arrives. Then we can see that all

the other signals except for **tx_ready** do not let the fsm moves to **WRITE** state (6), where, when **tx_ready** arrives, **en_cw** is raised and **we** come back to **IDLE** state.

Reading

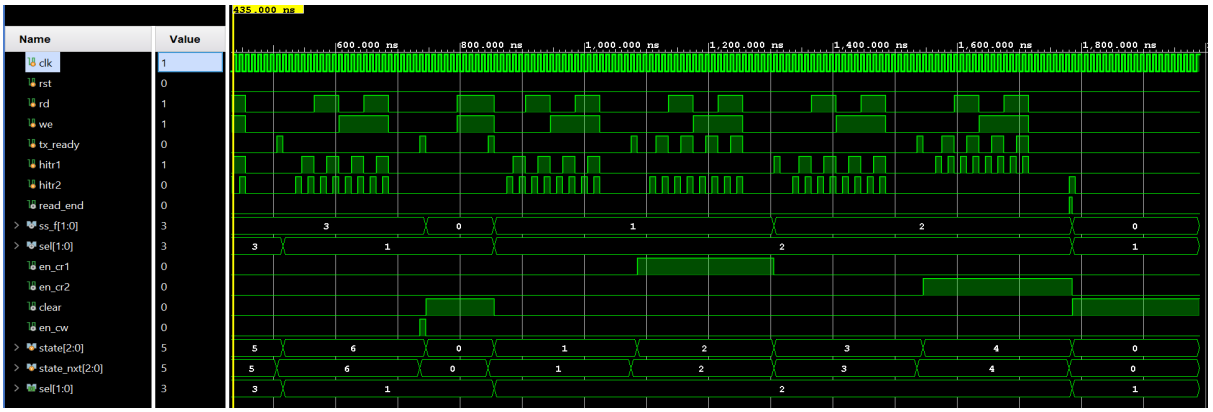


Figure 59- Reading Testbench

About the reading we notice that if **rd** and **we** arrive simultaneously, as soon as **tx_ready** arrives, we go to state 1 (**READ_REQ_1**), because the priority is set on reading. So if **rd** arrives, we go to **READY_REQ_1** (state 1), and we move to **READ_1** (state 2) only if **tx_ready** arrives. In **READ_1** we have to wait to get all data from Manor's FIFO, so we go to **READ_REQ_2** (state 3) only if **hitr1** = 1. Here we send **LEG** for the Cellar's fifo slave, and we go to **READ_2** (state 4) when tx_ready arrives. In **READ_2** we wait for **hitr2**, and when it arrives we raise **read_end** and come back to **IDLE** state.

8.4.1.4) Datapath Master-FIFO

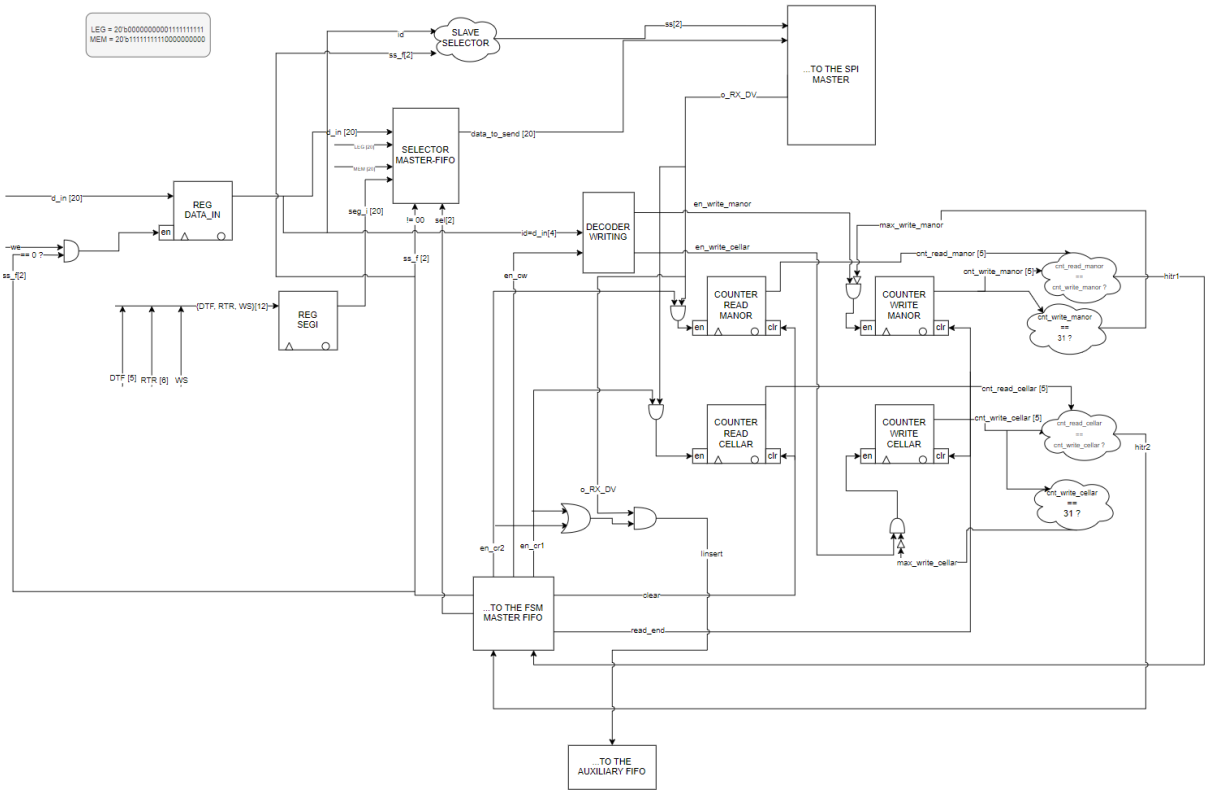


Figure 60- Master-FIFO Datapath

Input :

- ss_f [2], sel [2], en_cr1, en_cr2, read_end, en_cw, clear, sel [2], data_in [20], we, DTF [5], RTR [6], WS.
- o_RX_DV → Is another name to refer to tx_ready.

Output :

- ss [2], data_to_send [20], insert, hitr1, hitr2.

Functioning

The module is characterized by input registers used to memorize data_in and the combinations of DTF, RTR and WS (segi). Data_in register memorize the new data only if **we** signal arrives and ss_f equals to 00, in order to ignor writing request when another writing is being performed in FIFOs. Then we have the **Selector Master-FIFO** that is used to decide which data must be sent to the SPI Master (LEG, MEM, data_in or segi) based on sel and ss_f signals. The module is also characterized by 4 counters. Two that records the number of writing on the Manor and Cellar FIFOs, and the other two that keep track of the number of times

we have read from the FIFOs. In particular we need the writing counters to understand how many information we have written in each FIFO; in this way when we compute a reading we exactly know when we have to stop. Given the en_cw signal from FSM we decide which writing counter we have to increment based on ID signal, that is the fifth MSB bit in data_in signal. We generate hitr1 and hitr2 when the reading counter reaches the value of the corresponding writing one. Then we need to determinate the value of slave selector (ss) based on ss_f and ID signal values, and we compute this decision with the **Slave Selector**. Lastly we have to generate the insert signal in order to enable the writing in auxiliary FIFO. In particular we generate this signal whenever one of the reading counter is enabled(encr1 = 1 or encr2 = 1) and we know that we have just received a new word from MISO line (o_RX_DV = 1).

8.4.1.5) Datapath Testbench

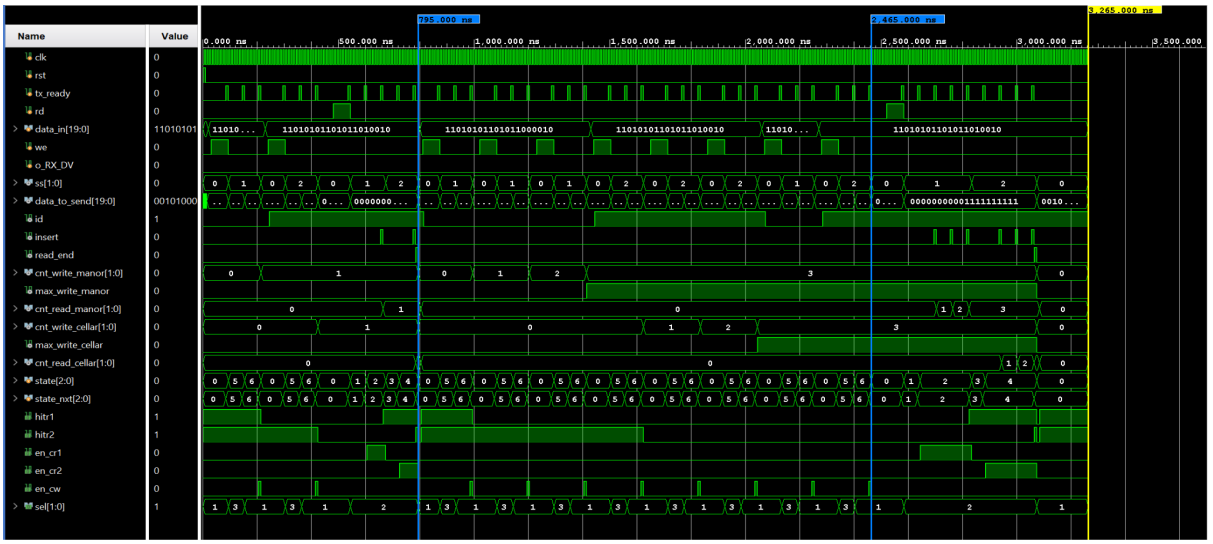


Figure 61-Datapath Testbench

The Datapath's testbench has been realized including the FSM, in order to generate the signals easier, and better understand the system's behaviour. We can divide the testbench in three parts.
In the first part we write data in Manor and Cellar and read once. We can see that when **we** and tx_ready (theorically the 7SD has finished to send data) arrives we move into WRITE_REQ state and we send MEM signal (11111111110000000000) with the combination of ss_f = 11 and sel = 11 that, thanks to the selector module, let us have as data_to_send the MEM value. Here with tx_ready = 1 we move into WRITE state where sel = 01 and we send the data placed in data_in to the SPI. When tx_ready = 1

we move into IDLE state rising en_cw. We do the same for the cellar (sending a data that gives us ID = 1). And then we read from the FIFOs with rd = 1. Here we can notice that insert signal raises every time we are in state 2 (READ_1) or state 4 (READ_2) and tx_ready (o_RX_SV signal from SPI arrives). In fact every time en_cr1 or en_cr2 are equal to 1 and tx_ready arrives, the respective reading counters increase their value until they reach the value of corresponding writing counters. When we have finished to read we can see that read_end signal raises to 1, writing counters are cleared and we came back to IDLE state where the reading counters are cleared as well.
The second part of the testbech has been realized in order to fully writing inside the FIFOs, to see if writing counters work properly or not, and we can see that, when they reach their maximum value, even if we write a new data, their values remain the same. In the general system we write however in the FIFOs, overwriting the data.
In the third part we read once again from the FIFOs to see if insert signal raises properly, and it does.

8.4.2) TOP_SLAVE_FIFO

This module implements the logic of the Slave connected to the SPI_slave for reading and writing from the FIFO.

Input :

- **o_RX_Byte**[20] → Signal coming from the SPI_slave, which represents the full data received from MOSI line.
- **o_RX_DV** → Signal coming from SPI_slave communicates that a single word of the entire data has been received.

Output :

- **i_TX_Byte**[20] → Signal coming from the FIFO, which contains the word to be sent with the MISO line.
- **i_TX_DV** → Signal coming from Slave_fifo communicates that a single word of the entire data is ready to be sent.

This module is characterized by two **sub-modules** :

- **Slave_fifo** → It contains most of the system, which is subdivided into other two modules **Fsm_slave** and **Dp_slave_fifo.**
- **top_FIFO** → FIFO of 20-bit words.

Functioning

The only purpose of this module is to contain the **Slave_fifo** and **top_FIFO.**

8.4.2.1) SLAVE_FIFO

Characteryzed from fsm_slave to Testbench General.

8.4.2.2) Fsm_slave

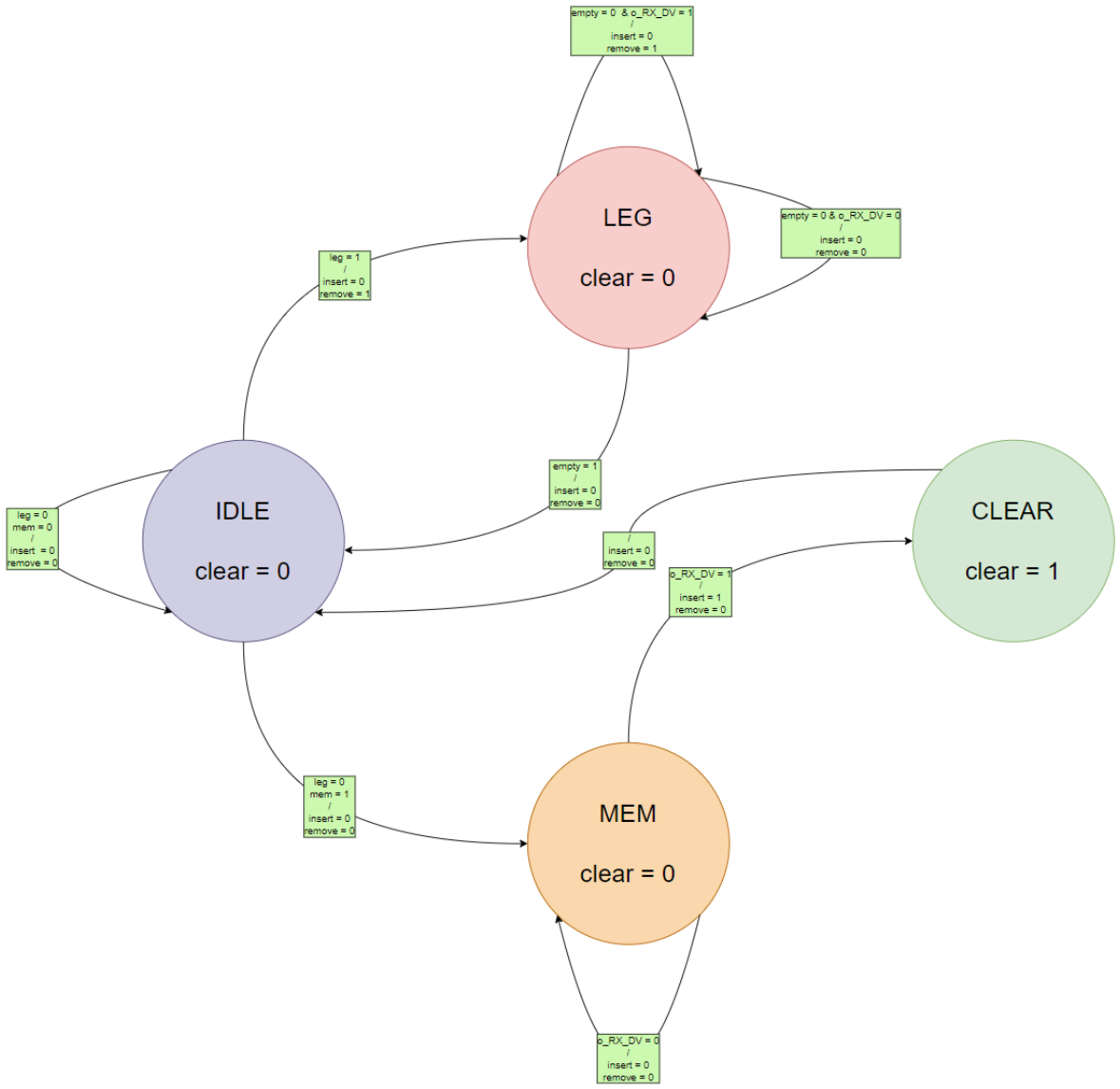


Figure 61- FSM Slave

8.4.2.3) FSM_SLAVE TABLES

FSM_slave

Future State Table

Inputs are in order : {leg,mem,empty,o_RX_DV}.
Mealy's output are in order : (insert,remove)
Moor output is just clear

Future State Table

	IDLE	LEG	MEM	CLEAR
0000	IDLE (0,0)	LEG (0,0)	MEM (0,0)	IDLE (0,0)
0001	IDLE (0,0)	LEG (0,1)	CLEAR (1,0)	IDLE (0,0)
0010	IDLE (0,0)	IDLE (0,0)	MEM (0,0)	IDLE (0,0)
0011	IDLE (0,0)	IDLE (0,0)	CLEAR (1,0)	IDLE (0,0)
0100	MEM (0,0)	LEG (0,0)	MEM (0,0)	IDLE (0,0)
0101	MEM (0,0)	LEG (0,1)	CLEAR (1,0)	IDLE (0,0)
0110	MEM (0,0)	IDLE (0,0)	MEM (0,0)	IDLE (0,0)
0111	MEM (0,0)	IDLE (0,0)	CLEAR (1,0)	IDLE (0,0)
1000	LEG (0,1)	LEG (0,0)	MEM (0,0)	IDLE (0,0)
1001	LEG (0,1)	LEG (0,1)	CLEAR (1,0)	IDLE (0,0)
1010	LEG (0,1)	IDLE (0,0)	MEM (0,0)	IDLE (0,0)
1011	LEG (0,1)	IDLE (0,0)	CLEAR (1,0)	IDLE (0,0)
1100	LEG (0,1)	LEG (0,0)	MEM (0,0)	IDLE (0,0)
1101	LEG (0,1)	LEG (0,1)	CLEAR (1,0)	IDLE (0,0)
1110	LEG (0,1)	IDLE (0,0)	MEM (0,0)	IDLE (0,0)
1111	LEG (0,1)	IDLE (0,0)	CLEAR (1,0)	IDLE (0,0)

Table 27- Next State FSM Slave

Moore Outputs Table

	clear
IDLE	0
LEG	0
MEM	0
CLEAR	1

Table 28- Moore's Output FSM Slave

Input :

- **mem** → Signal coming from Dp_slave_fifo, requires the memorization of the word coming from SPI_slave.
- **leg** → Signal coming from Dp_slave_fifo, requires the reading of every word contained in the FIFO.
- **empty** → Signal coming from FIFO, tells that the FIFO is empty.
- **o_RX_DV**.

Output :

- **clear** → Signal coming out from the FSM, restore the original value (20'd0).
- **insert** → Signal coming out from the FSM, requires the insertion of the word contained in **o_RX_Byte** into the FIFO.
- **remove** → Signal coming out from the FSM, requires the removal of all the words from FIFO.

Functioning

The FSM starts in the **IDLE** state, in this state every output is at 0 until the **leg** or **mem** signal goes to 1, if **leg** = 1 happens the FSM will put **remove** at 1 for 1 clk cycle until it goes into **LEG** state. In this state, the FSM put **remove** to 1 every time **o_RX_DV** goes to 1. The FSM will go into the **IDLE** state as soon as **empty** rises to 1. In the **IDLE** state if **mem** goes to 1 the **FSM** will go into the **MEM** state and when the **o_RX_DV** signal will rise to 1 the FSM will put **insert** to 1 and change state into **CLEAR** state, in this state will put **clear** at 1 and in the next clk cycle will go into the **IDLE** state.

8.4.2.4) Testbench

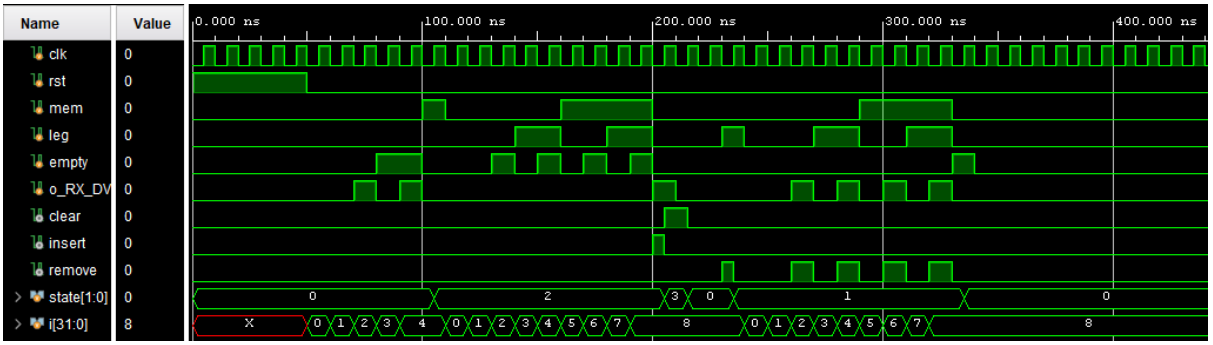


Figure 62- Testbench

In this testbench,(figure 62) for each state are tested every nonsensitive input. We can see that in the **IDLE** state, the transition (to **LEG** or **MEM**) is made just when **mem** or **leg** signals are raised; in the **MEM** state the transition (to **CLEAR**) is made when the **o_RX_DV** signal is raised; in the **LEG** state the transition (to **IDLE**) is made when the **empty** signal is raised; in the **CLEAR** state, the transition (to **IDLE**) is made at the next cycle of clock.

8.4.2.5) Dp_slave_fifo

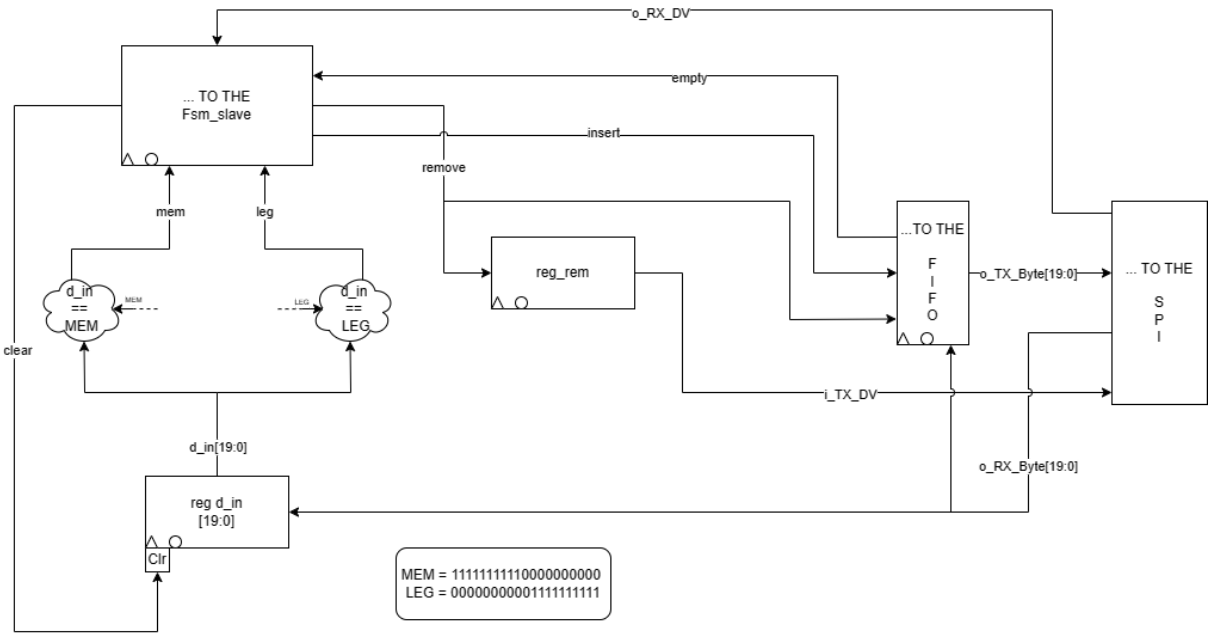


Figure 63- Dp_slave_fifo

Input :

- o_RX_Byte[20], o_RX_DV, clear, remove.

Output :

- i_TX_DV, mem, leg.

Functioning

In this module are created 3 important signals, **mem** and **leg** are created confronting the value of d_in (a register that permits the disassociation between the value contained in the **SPI_slave** register from the verification of a request sequence) with two predetermined sequences which represent the reading and writing request. This module creates also another signal (**i_TX_DV**) it's created delaying the **remove** signal to permit the correct synchronization with the SPI.

8.4.2.6) Testbench General (including Dp_slave_fifo)

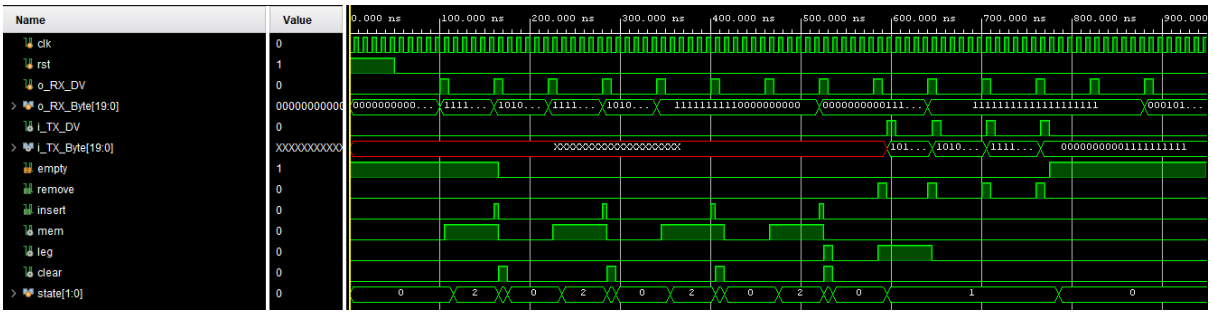


Figure 64- General Testbench

This testbench (figure 64) tests the general response of the system at specific critical and noncritical inputs. Initially, it tests the insertion of some normal words into the FIFO, sending first the **MEM** word and then the actual word. In the last two insertions are memorized into the FIFO **MEM** and **LEG** words, this is supposed to create a bug but thanks to the clear signal it does not append. In the end, the **LEG** word is sent and the FIFO responds with all the data inside itself.

8.4.3) TOP_SPI_MASTER_INTERFACE

As said in Master FIFO Display **we** and **rd** signals pass through fsm relè.

8.4.3.1) FSM RELE

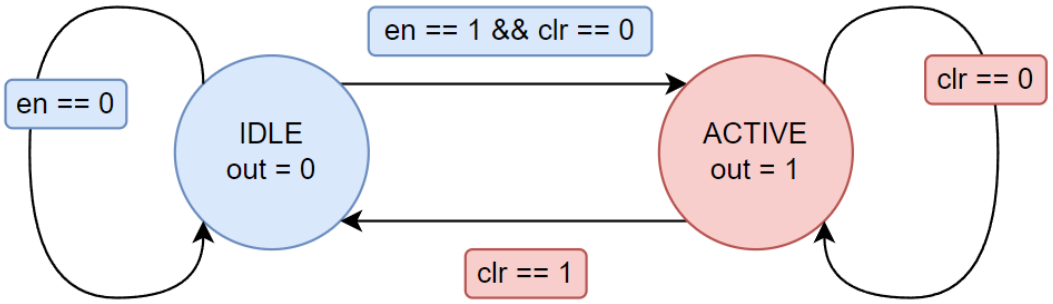


Figure 65- FSM Rele

8.4.3.2) FSM RELE TABLES

FSM_Rele

Future State Table

Input are in order : {en, clr}

	00	01	10	11
IDLE	IDLE	IDLE	ACTIVE	IDLE
ACTIVE	ACTIVE	IDLE	ACTIVE	IDLE

Table 29- Future State FSM Rele

Moore Outputs Table

STATE	OUT
IDLE	0
ACTIVE	1

Table 30- Moore's Output FSM Rele

Functioning

The FSM RELE takes as input **en** signal, that represents the signal that we want to stabilize. In particular if we are in **IDLE** state **out** (stabilized value) is equals to 0, and when **en** = 1 arrives we move to **ACTIVE** state where **out** = 1 until a **clr** signal arrives.

8.4.3.3) FSM RELE Testbench

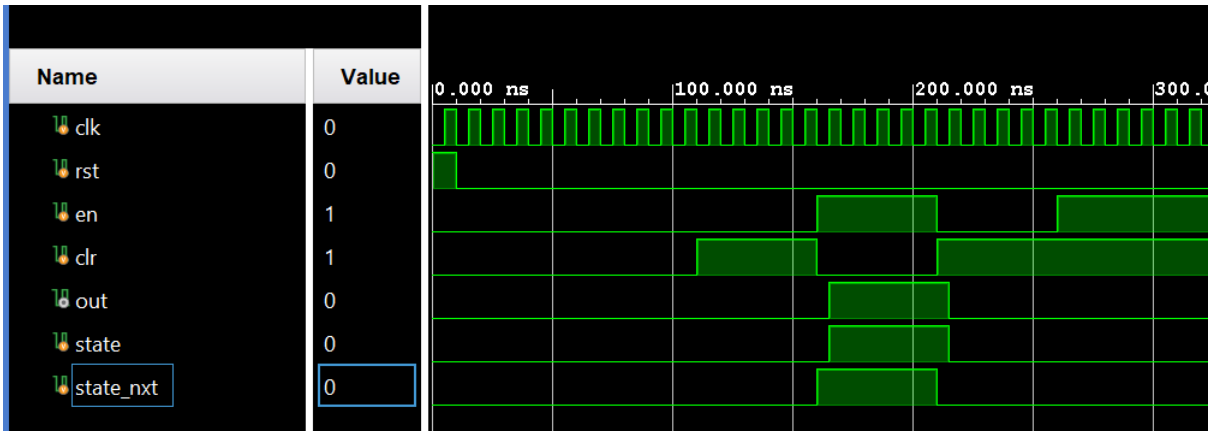


Figure 66- FSM Rele Testbench

Here we can see that the behavior is exactly as expected, and when both **en** and **clr** arrives at the same time, the fsm stays in the **IDLE** state.

TOP_SPI_MASTER_INTERFACE

Now that we have introduce fsm rele, we can come back to **Top_spi_master_interface**. As introduced before it is a temporary module that we used in order to verify the correct functioning of SPI Master and SPI Slave for communication with FIFOs and 7SD. This module contains Master_fifo_display.v, both top_slave_fifo.v for Manor and Cellar, top_slave_7SD, all of them connected to an SPI_top.v module and auxiliary FIFO. We used this module in order to verify the functioning of the SPI interface.

8.4.4) TOP_SPI_MASTER_INTERFACE Testbench

Writing in Manor

Figure 67- Writing Manor Testbench

At the begin of the testbench, where the FSM_Master_FIFO (state fsm master) is in IDLE state the TX_cnt_display in Display FIFO is changing its value, that means the Slave of the 7SD is receiving data and is sending data back to the master along MISO line. Meanwhile both Manor and Cellar SPI Slave are not performing any operation, because ss = 00. When we are in IDLE and we signal arrives we can see how we_rele remains high, in order to let finish the sending of data to the 7SD Slave. Here when o_RX_DV arrives (tx_ready = 1) display has finished, so we can go to WRITE_REQ state where ss_f = 11 and ss = 01, in fact we are writing into Manor. From slave point of view when o_RX_DV arrives we go to MEM state in this case, and when the next o_RX_DV arrives it represent the arrive of the data that must be stored, in fact insert will go to 1 and a new data is stored in Manor's FIFO.

Reading

Figure 68- Reading Manor Testbench

Here we compute a reading. When the rd signal arrives and Master finish sending data to 7SD, FSM_Master_FIFO moves to READ_REQUEST_1 state and sending of data begin. The first slave from which we get data from FIFO is the manor. Here we see in fact that after the arrives of LEG signal, state fsm manor moves to LEG state, and for each o_RX_DV that arrives before the FIFO is empty we send next FIFO data back data to the Master. And in this case, because we only have values in Manor when FSM_Master_FIFO is in READ_1 state and o_RX_DV arrives we insert data into auxiliary FIFO.

Writing request during writing operation

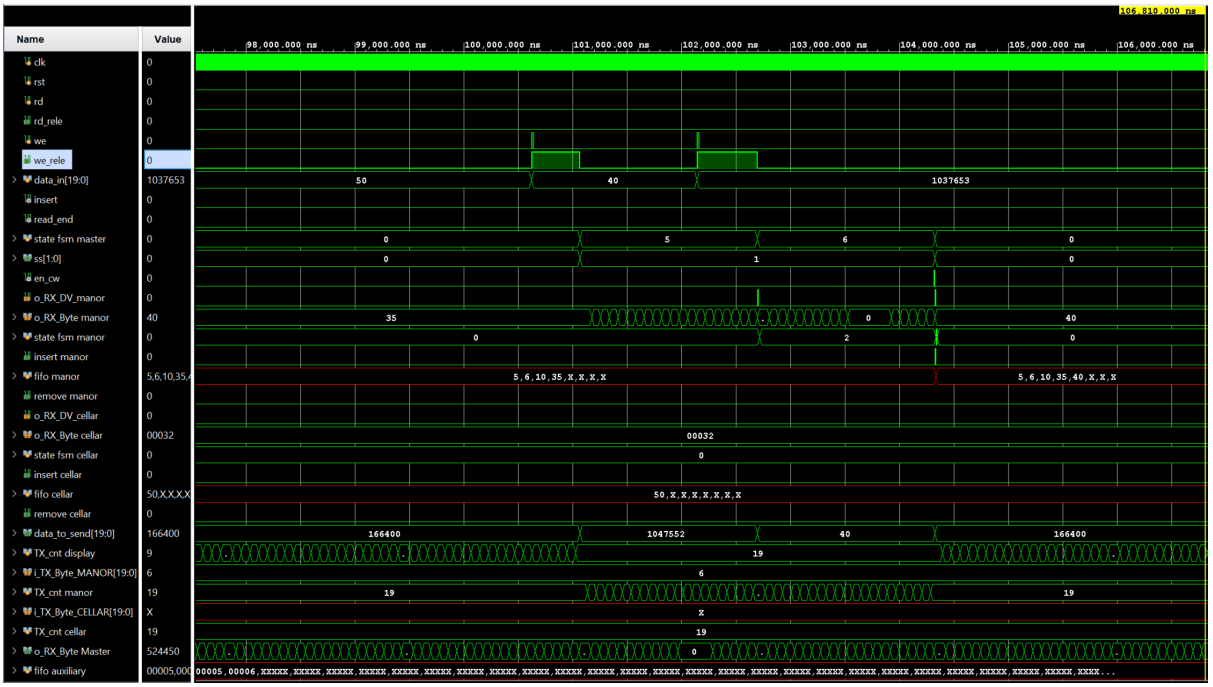


Figure 69- Writing Request Testbench

Here we see what happens when a writing request arrives when another writing operation is being performed. Basically this other writing operation is ignored because, as already explained, ss_f != 00 in that moment, so no data is stored in data_in register.

After a reading we continue to perform writing in both Manor and Cellar in order to fill at least one of the peripheral FIFO.

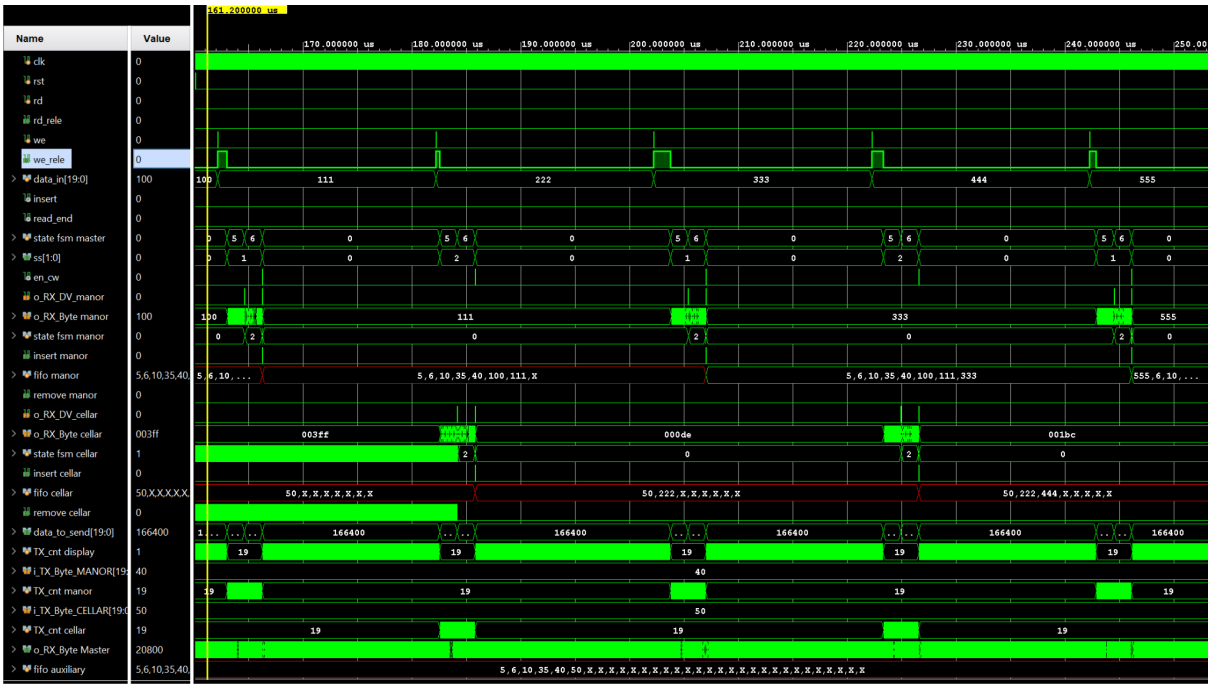


Figure 70-writing Testbench

Last reading

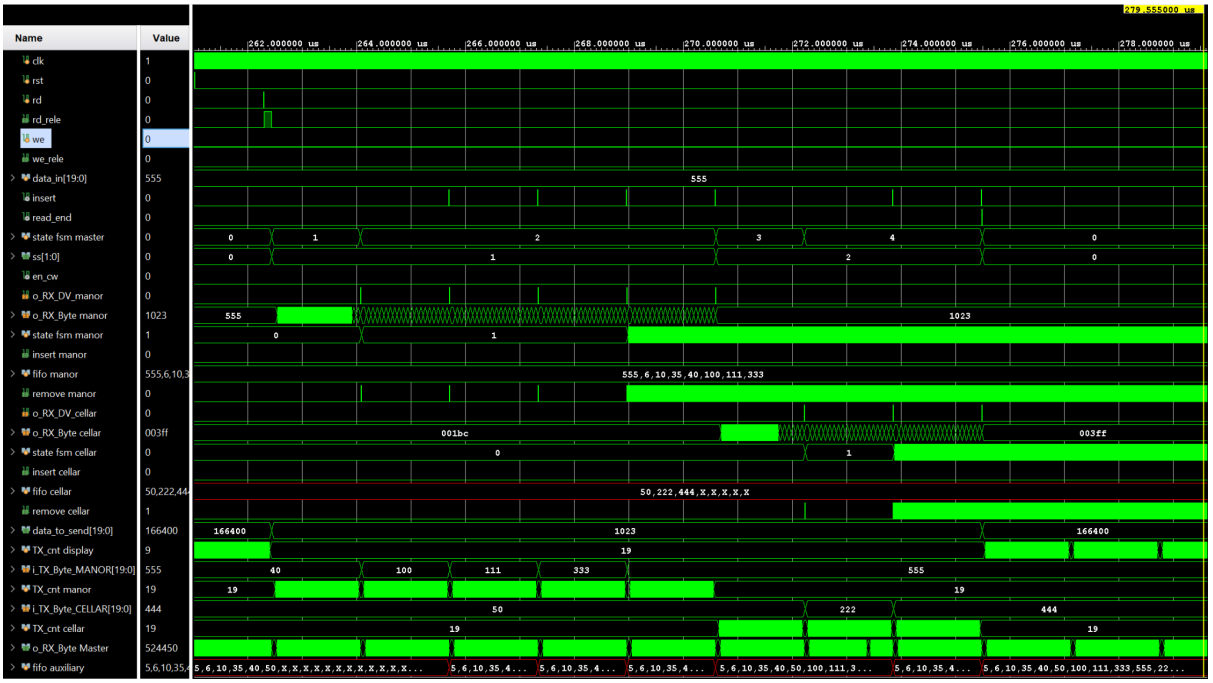


Figure 71- Last readingTestbench

Here we compute the last reading and we can see that everything works properly. We can see in fact that for each new data we have previously inserted into the FIFOs we insert a new data into auxiliary FIFO.

COMMS DISPATCHER

9.1) Comms Datapath

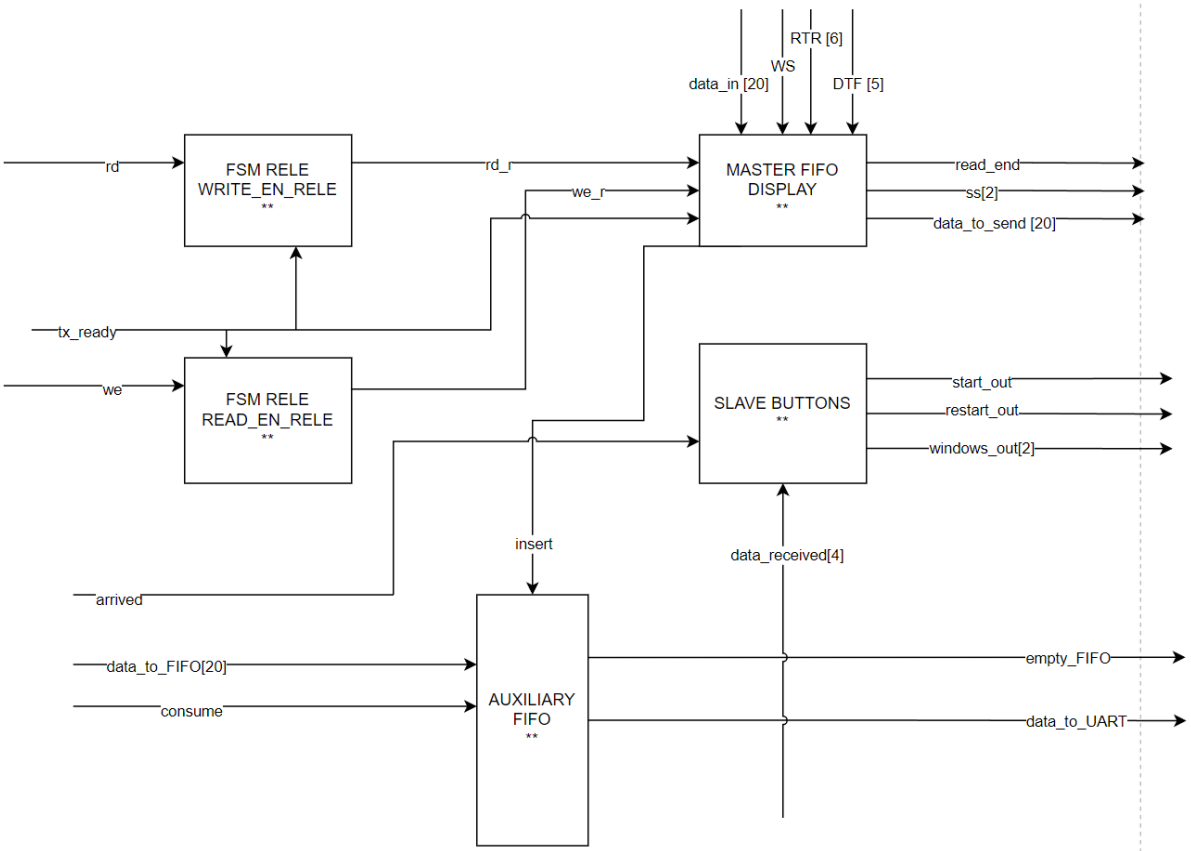


Figure 72- Comms Dispatcher

9.2) Comms Description

Module we have used in order to contain Master_fifo_display module, Slave_buttons and auxiliary FIFO. So these are the modules that are interconnected to the SPI that communicates with all the external peripherals. Furthermore we have decided to add the auxiliary FIFO that will be used for the communication with the log_screen_connection.

ERRORS AND WARNINGS

10.1) Errors And Warnings

```
Synthesis (42 warnings)
  [Synth 8-11065] parameter 'IDLE' becomes localparam in 'DB' with formal parameter declaration list [DB.v:27] (18 more like this)
  [Synth 8-7137] Register o_clk_reg in module clk_gen_M has both Set and reset with same priority. This may cause simulation mismatches. Consider rewriting code [clk_gen_M.v:91] (2 more like this)
  [Synth 8-6014] Unused sequential element skipOne_reg was removed. [adapter_M.v:40] (1 more like this)
  [Synth 8-7129] Port TX_cnt[1] in module modeAdapterS__parameterized0 is either unconnected or has no load (16 more like this)
  [Synth 8-7080] Parallel synthesis criteria is not met
```

Figure 73- Errors and warnings

The warnings are from the synthesis, and caused by the SPI's module imported, about declaration of local parameters and parallel synthesis criteria not met. All these warnings can be safely ignored.

SYSTEM ANALYSIS

11.1) Performance And Different Implementation's Settings Analysis

We tried different synthesis settings, to see how Vivado implement what we have designed on board. The settings we have tried, with a 10 ns clock period (Basys 3 standard clock period), are :

- Default.
- Flow_perfOptimized_high
- Flow_areaOptimized_high.

11.2) Default : Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2,834 ns	Worst Hold Slack (WHS):	0,069 ns	Worst Pulse Width Slack (WPWS):	3,750 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1886	Total Number of Endpoints:	1886	Total Number of Endpoints:	1076

All user specified timing constraints are met.

Figure 74- Default Timing

11.3) Default : Device Design

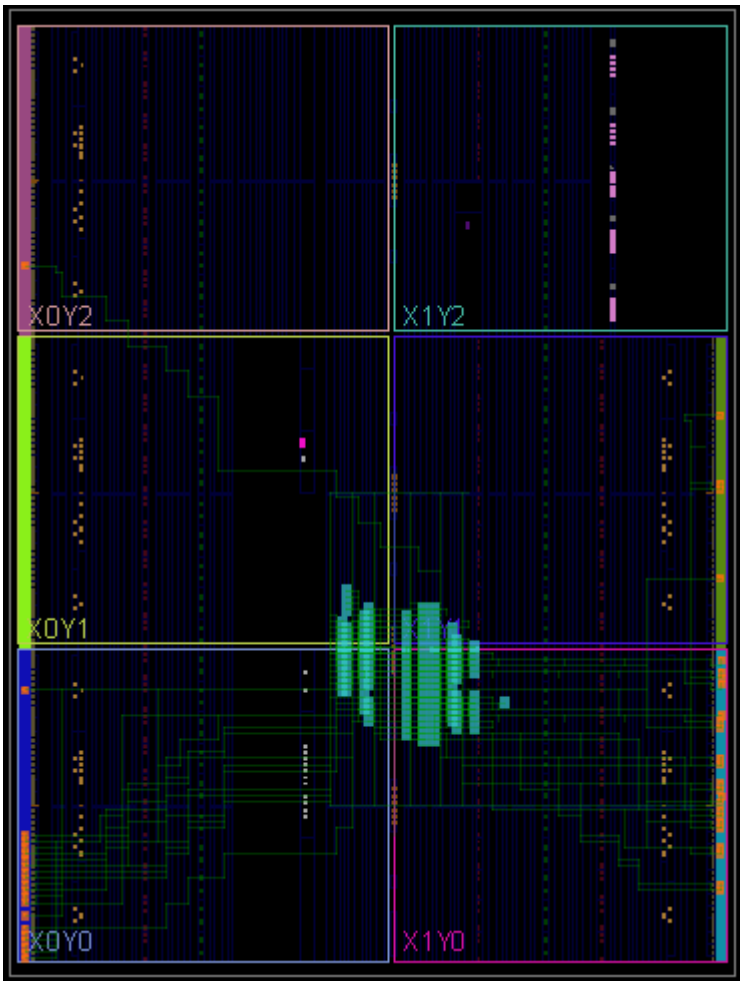


Figure 75- Default Device Design

11.4) Default : Utilization

Utilization **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	831	20800	4.00
LUTRAM	60	9600	0.63
FF	785	41600	1.89
IO	46	106	43.40
BUFG	1	32	3.13

Figure 76- Default Utilization

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	816	20800	3.92
LUTRAM	60	9600	0.63
FF	785	41600	1.89
IO	46	106	43.40
BUFG	1	32	3.13

Figure 77- Default Utilization 2

11.5) Flow_areaOptimized_high : Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1,493 ns	Worst Hold Slack (WHS):	0,084 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1484	Total Number of Endpoints:	1484	Total Number of Endpoints:	961

All user specified timing constraints are met.

Figure 78- AreaOptimized Timing

11.6) Flow_areaOptimized_high : Device Design

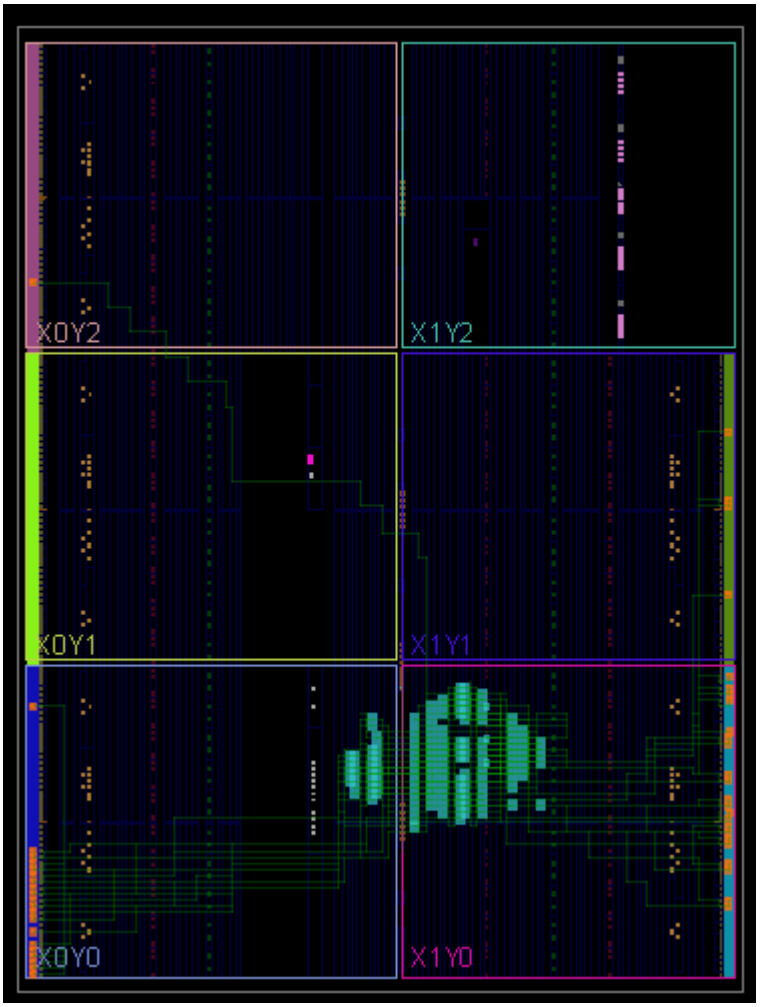


Figure 79- AreaOptimized Device Design

11.7) Flow_areaOptimized_high : Utilization

Utilization **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	705	20800	3.39
FF	726	41600	1.75
BRAM	1.50	50	3.00
IO	46	106	43.40
BUFG	1	32	3.13

Figure 80- AreaOptimized Utilization

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	690	20800	3.32
FF	726	41600	1.75
BRAM	1.50	50	3.00
IO	46	106	43.40
BUFG	1	32	3.13

Figure 81- AreaOptimized Utilization 2

As expected with areaOptimization the performance are poor respect to default settings, but the number of LUT and FF used decreased.

11.8) Flow_perfOptimized_high : Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3,553 ns	Worst Hold Slack (WHS):	0,069 ns	Worst Pulse Width Slack (WPWS):	3,750 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1959	Total Number of Endpoints:	1959	Total Number of Endpoints:	1130

All user specified timing constraints are met.

Figure 82- PerfOptimized Timing

11.9) Flow_perfOptimized_high : Device Design

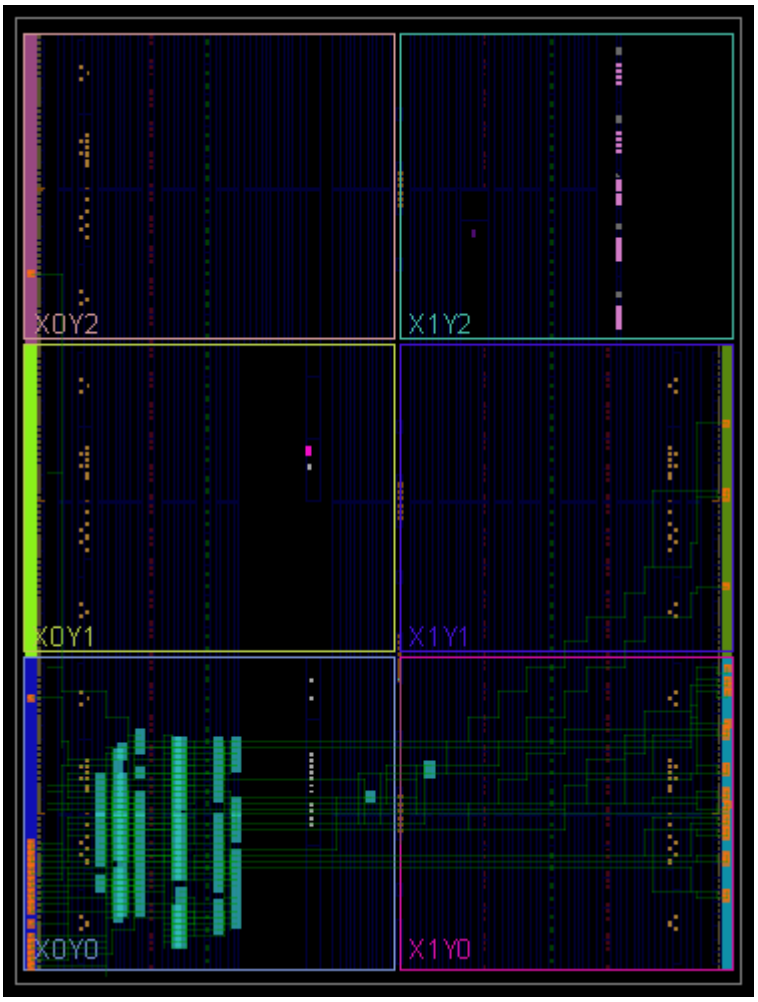


Figure 83- PerfOptimized Device Design

11.10) Flow_perfOptimized_high : Utilization

Utilization **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	1120	20800	5.38
LUTRAM	60	9600	0.63
FF	842	41600	2.02
IO	46	106	43.40
BUFG	1	32	3.13

Figure 84- PerfOptimized Utilization

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1016	20800	4.88
LUTRAM	60	9600	0.63
FF	842	41600	2.02
IO	46	106	43.40
BUFG	1	32	3.13

Figure 85- PerfOptimized Utilization

As expected with perfOptimization the performance is better than default and areaOptimization settings, but the number of LUT and FF that will be used for the implementation on board increased.

11.11) Highest frequency for smallest positive slack

As seen before with a 10 ns clock period, and default synthesis settings, the slack is 2.834ns, which means that timing constraints are met.

Here we try different clock periods to find the highest frequency that guarantees the smallest positive slack.

Clock's period : 7.5 ns

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,334 ns	Worst Hold Slack (WHS):	0,069 ns	Worst Pulse Width Slack (WPWS):	2,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1886	Total Number of Endpoints:	1886	Total Number of Endpoints:	1076

All user specified timing constraints are met.

Figure 86- Clock 7.5ns

So this is the minimum period we implemented that gives positive slack.

Clock's period : 7 ns

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0,166 ns	Worst Hold Slack (WHS):	0,069 ns	Worst Pulse Width Slack (WPWS):	2,250 ns
Total Negative Slack (TNS):	-0,412 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	6	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1886	Total Number of Endpoints:	1886	Total Number of Endpoints:	1076

Timing constraints are not met.

Figure 87- Clock 7ns

Negative slack, so we cannot go so high with frequency.

TASKS

12.1) Tasks Structure Explanation

We have to split the tasks into 6 groups:

- Ideas: planning.
- Graphic design: draw of the schematic, finite state machine or table.
- Screenshot: screenshot of the schematic, finite state machine or testbench.
- Description: description of the chapters.
- Vivado: writing code, testbench and debug.

12.2) Emanuele Are's Tasks

Ideas :planning of the modules that are described in paragraphs 4.1, 4.3, 8.1, 8.1.1, 8.3, 8.4, 8.4.1,8.4.1.1, 8.1.4.4, 8.4.2, 8.4.2.1, 8.4.2.5, 8.4.3, 9.1.

Graphic design: 8.4.2.3 (table 27 and 28)

Screenshot: Screenshot of 4.3, (fig.12 and fig.13) , 8.4.2.2, 8.4.2.4, 8.4.2.5, 8.4.2.6.

Description: Description of 8.4.2, 8.4.2.1, 8.4.2.2, 8.4.2.3, 8.4.2.4, 8.4.2.5, 8.4.2.6

Vivado: Writing code, testbench and debug of 4.1, 4.3, 8.2, 8.2.1, 8.2.2, 8.2.3, 8.4.2, 8.4.2.2, 8.4.2.4, 8.4.2.5, 8.4.2.6, 9.1.

12.3) Giuseppe Bergonzi's Tasks

- Ideas: decisions about changes and additions on the old Thermostat, planning of modules: 5.2,5.4,5.5(.4, .8), 5.7(.6, .8), 5.8, 5.9, 5.10, 5.11
- Graphic design: 5.4.1, 5.5.1, 5.5.2
- Screenshot: 5.3(1,2) ,5.10.2
- Description: 5.3, 5.5.9, 5.10.1 ,5.10.2
- Vivado: writing code, testbench and debug of :5.1, 5.4, 5.4.4, 5.5 (.4, .8), 5.7(.4, .8), 5.8, 5.9, 5.10, 5.11

12.4) Giulio Cellesi's Tasks

Ideas: planning of 7.1, 7.2, 7.2.1, 7.2.2, 7.3

Graphic design: draw of the schematic of 4.1, 7.1, 7.2

Description: 7.1, 7.2, 7.3

Vivado: Writing code, testbench and debug of 4.1, 7.1, 7.2, 7.3

12.5) Lorenzo Mascia's Tasks

Ideas : planning of the modules that are described in paragraphs 4.1, 4.3, 8.1, 8.1.1, 8.3, 8.4, 8.4.1,8.4.1.1, 8.1.4.4, 8.4.2, 8.4.2.1, 8.4.2.5, 8.4.3, 8.4.3.1, 9.

Graphic design : 7.3, 8.4.1.1, 8.4.1.2, 8.4.1.4, 8.4.3.2, 9.1.

Screenshot : 8.3.1, 8.4.1.3, 8.4.1.5, 8.4.3.3, 8.4.4.

Description : Description of 8.1, 8.1.1, 8.2, 8.2.1, 8.2.2 8.2.3, 8.3, 8.4, 8.4.1, 8.4.1.1, 8.4.1.2, 8.4.1.3, 8.4.1.4, 8.4.1.5, 8.4.3, 8.4.3.1, 8.4.3.2, 8.4.3.3, 8.4.4, 9.1, 9.2, 10.1, 6.1, 4.2 (only description).

Vivado : Writing code, testbench and debug of 4.1, 4.3, 8.2, 8.2.1, 8.2.2, 8.2.3, 8.3, 8.3.1, 8.4, 8.4.1,8.4.1.1, 8.4.1.4, 8.4.3, 8.4.3.1, 8.4.4, 9.1.

12.6) Carlo Edoardo Miccio's Tasks

Ideas: Index, all the links, all the captions, imported Assignment and Old Module. Planning of 5.1, 5.3, 5.4, 5.4.1, 5.4.4, 5.5, 5.5.1, 5.5.4, 5.5.8, 5.5.9,5.5.11.

Graphic design: 5.1, 5.4.1, 5.4.4, 5.5.1, 5.5.4, 5.5.8, 5.5.11, 5.9.1.

Screenshot: 3.1, 5.4.1, 5.4.3,5.5.4 5.4.7, 5.5.1, 5.5.3, 5.5.4, 5.5.7, 5.5.8, 5.5.10, 5.5.11, 5.5.15.

Description: description of 3 (all), 5.2, 5.4.2, 5.4.3, 5.4.6, 5.4.7, 5.5.2,5.5.3, 5.5.4, 5.5.6, 5.5.7, 5.5.9, 5.5.10, 5.5.13, 5.5.14,5.5.15,,5.9.1,5.9.2.12.1,12.8,12.9.

Vivado: writing code, testbench and debug of 4.1, 4.3, 5.1, 5.3, 5.4, 5.4.3, 5.4.7, 5.5 , 5.5.1, 5.5.3, 5.5.7, 5.5.10, 5.5.11, 5.5.15, 5.8, 5.9.

12.7) Giovanni Usai's Tasks

- Ideas: planning of modules: 5.1, 5.6, 5.7, 5.7.4, 5.8
- Graphic design: 5.1, 5.4.5, 5.5.5, 5.5.12, 5.6.1, 5.7.1, 5.7.4, 5.7.5, 5.7.8, 5.7.9, 5.8.1.
- Screenshot: 4.3.1, 4.3.2, 4.3.3, 4.3.4, 5.3, 5.6.1, 5.6.3, 5.7.1, 5.7.3, 5.7.4, 5.7.7, 5.7.8, 5.7.11, 5.8.1, 5.8.3.
- Description: description of 5.2, 5.6.2, 5.7.2, 5.7.6, 5.7.10, 5.8.2.
- Vivado: writing code, testbench and debug of 4.1, 4.3, 5.1, 5.3, 5.6.1, 5.6.3, 5.7.1, 5.7.3, 5.7.5, 5.7.7, 5.7.8, 5.7.11, 5.8.1, 5.8.3, 5.11.2.

12.8) Deadlines

We have not set specific deadlines.

12.9) Collaboration

Tasks repeated under the name of more than one person means that they were done in collaboration and by more than one member.

The integration and debug of the complete system (Top_Thermostat) has been performed by all the members of the group.