DIGITAL ELECTRONICS PROJECT

One wire digital thermometer

Marc Salas Huetos

INDEX

1. Introduction

Those are the principal specifications of the project and the designs to be developed.

- For this project has to be used a DS18S20 1-Wire Digital Thermometer.
- The device has to be able to display from -55 to 125 degrees Celsius, and has to have a way to convert form Degrees Celsius to Degrees Fahrenheit
- The Value of the temperature has to be displayed on the 7 segments display.
- The refreshment rate has to be minimum 1 second
- For the logic has to be used a NEXYS A7 Digilent Board powered by an ARTIX-7 Xilinx FPGA
- The design has to be Full synchronous, which means that All input signals have to by synchronized and all flip-flops have to be sensitive to the same clock
- The design has to be hierarchical and modular

The designs in that case cover until the maximal precision that the DS18S20 Thermometer can give, being able to display that maximal precision only in degrees centigrade. And the refreshment rate is approximately 800 milliseconds.

2. Functional description

The Description of the project will start with a functional description of every part of the project. In these paragraphs we will describe How all the device work together, and how all the problems have been solved. It will start with the 1 wire protocol routines used to communicate with the DS18S20 device, after that, a description of how have been done the conversion from binary to BCD and to degrees Fahrenheit, and finally a brief description about how the maximum precision information has been achieved, processed and displayed.

2.1. One wire protocol description and used commands

As we previously commented, the protocol used to communicate between Master (FPGA) and Slave (DS18S20) Is the One Wire Protocol. As his name indicates, this protocol is used with only one wire. This leads that all the communications, transmissions and receptions have to be implemented in only that wire not allowing to occur a transmission and a reception at the same time interval. For this reason, a hierarchy between devices have to be implemented to avoid problems on the line. We need a master and the slaves, in this case only one. And is for this reason also that in order to maintain the line hierarchy, and avoiding an information crush the DS18S20 only will answer to the FPGA after certain commands.

If we take a look at the figure 1, we can see the hardware implementation of the protocol. We can see there that the line will be continuously connected to pull up, thus, if the line is not pulled down by the master or the slave, in the Rx port we will see all time the line with a logic ‘1’ received. The same for the Rx port of the DS18S20. We will call this state the steady state for all the communications description.

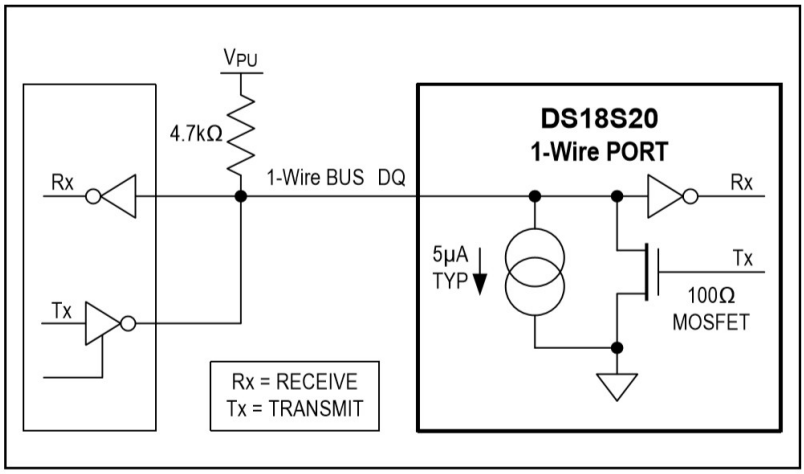

Figure 1. Hardware implementation of the 1 Wire protocol

Despite the multiple options that the thermometer has, we will only use a simple sequence that will be repeated every time that we refresh the thermometer lecture. In our case will be repeated nonstop, every 800 milliseconds, the time that this routine take to be done. This routine can be seen in the figure 2. This sequence has been designed following the datasheet of the thermometer, and is based on the tables of flow chart that are specified in the document.

Can be observed that every routine to be executed has to be initialized by an initialization sequence that is conformed of a master reset pulse and the answer of the thermometer. After this step we have to specify that the next commands are destined to our slave device. This can be done specifying with a rom command the Identification number of the device in the case of multiple slaves. In this case due to we have a single slave, we can execute a skip rom command (CCh) in order to specify that the following command is for all the devices in the line, in our case only the thermometer.

After that initialization routine and skipping the rom command has to be sent the proper command. In the first iteration is going to be sent a convert temperature command (44h) and wait for the maximal conversion multiplied by a security coefficient, in this case a 4% more which will lead to 780 milliseconds. In the second iteration we will execute a read scratchpad command (BEh) and immediately after that we will send read time slots in order to read the complete Scratchpad memory, and after that even is not specified in the figure 2 the master will wait 5 milliseconds before starting again all the sequence.

Is not the objective of this report explain how the write time slots and the read time slots has to be sent, but has to be specified at least that except on the initialization sequence where done for security, the presence pulse is sampled twice, all time that the slave have to write something to the line, the master is going to sample the line only once. If considered as

interesting, the timings took to generate the read and the write time slots are specified in their respective FSM design in the following parts of the report. In general, all the timing values are oversized in order to avoid all the possible errors.

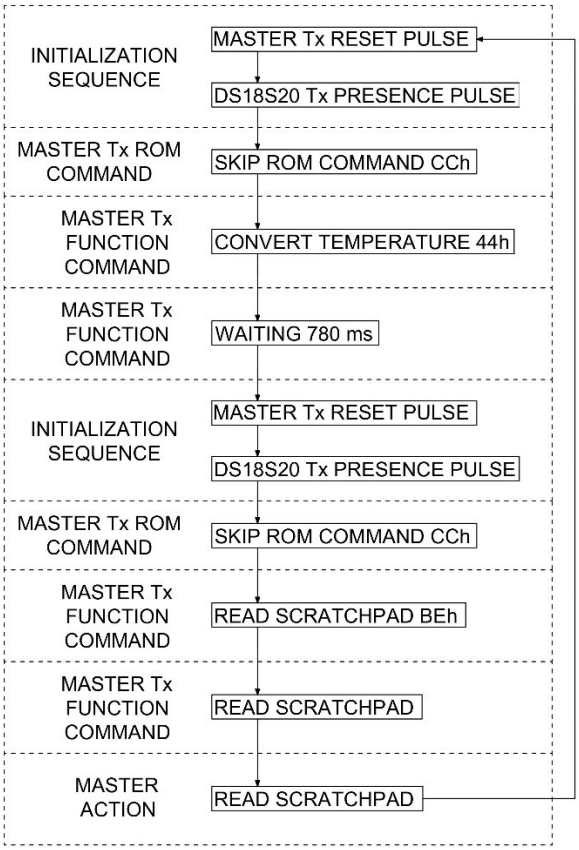

Figure 2. Sequence of 1 Wire protocol commands

2.2. Binary to centigrade degrees

The non-maximum precision conversion to a degree centigrade is the easiest one to perform. Can be easily done in different ways, but in this case has for similitude to the Binary to Fahrenheit conversion method been done doing a count until the binary number itself. It is properly explained in the next paragraphs.

The first think to do is fix the minus if the temperature is negative, this information can be easily extracted if we see whatever bit of the second byte sent. In the same way we will do a logical not to the first byte allowing to extract the digits information if the second byte is again FFh.

At this point if the value is not zero, we will start to count until the counter achieve the expression of the first byte of the digital output (Less significant byte). In every iteration, we will swap the decimals digit from zero to five and from five to zero again. The starting count will be zero if the number is positive and 1 if the number is negative due to the bias produced by the two’s complement notation used by the sensor that can be seen in the table 1.

Every two numbers counted we will add one to the unit’s digit until it reaches the digit number 9 when we will swap it to 0 again and at the same

moment add 1 to the ten's unit. The same will happen with the hundreds digit. We can understand that process as a natural counting, but every two numbers. For a more detailed explanation, see the diagram of the FSM of the T2Cd_FSM in the next chapters of the report.

T(ºC)	Digital output (Binary) MSB	LSB
-1,5	0000 0000	0000 0011
-1,0	0000 0000	0000 0010
-0,5	0000 0000	0000 0001
0,0	0000 0000	0000 0000
0,5	1111 1111	1111 1111
1,0	1111 1111	1111 1110
1,5	1111 1111	1111 1101

Table 1. Conversion table from Binary to BCD in degrees Centigrade

It has to be specified also that the display of the temperature will differentiate between the digit 0 and the null character. If we represent the output of the characters in the 7 segments display as this vector {s h d u}, corresponding the 's', the 'h', the 'd' and the 'u' as the sign display, the hundreds display, the tens display and the units display respectively, the number temperature -10 degrees will be represented as { - 1 0}. The sign display will take a null character since is not necessary for this representation and the sign will be represented in the place of the hundreds place, the same will occur with the representation of the number 9 that will be represented as { 9} . This will be achieved with a simple combinational logic that will be connected at the output of the block that generates the conversion.

2.3. Binary to Fahrenheit degrees

The conversion form binary to Fahrenheit degrees will be done with a similar technique that the binary to Centigrade, being this conversion a little bit more complex.

As in the binary to degrees the first think that has to be done is set if the temperature in degrees centigrade is negative, what will suppose the start point for the futures operations. First of all, if the more significant Byte (MSB) is FFh we will negate the less significant Byte (LSB) of the digital output and the Two's complement bias will be corrected. After doing that we will consider if the temperature belongs to the negative or the positive in the domain of the Fahrenheits. Two conditions have to be respected for that, the MSB has to be FFh and the already negated LSB has a value over 00100011b. These two conditions fix the temperature below the -17.5 degrees Centigrade, what mean that we reached the negative values for the Fahrenheit degrees. This frontier can be observed with the four values in yellow in the table 2.

Once we know if we are above or below the 0 in Fahrenheit degrees, we can fix the sign digit, and we will start the count at 0.5 in the case we are above the 0 or at -0.4 in the case we are below of it. We will at this point add 35 or subtract 36 units to the (LSB) in order to be consistent with the point where we start the count.

After all that preparation the initial conditions for the count are fixed and as in the conversion form binary to Centigrade degrees will start counting from 0 to the value of the (LSB). In this case the decimal digit won't vary from 5 to 0 and from 0 to 5 consecutively, the decimal will decrease a unit in every count switching to 9 every time that we reach the 0, a complete iteration can be observed in orange in the table 2.

The units, tens and hundred counters will increase as in the conversion to degrees Centigrade, but this time will increase a unit in every count. The only think that we have in account is that, every time the decimal reaches the value of 0, the units, tens and hundred counters have to keep the value. This phenomenon can be observed also in the table 2 for the red values.

Again, as in the binary to Centigrade degrees converter the display of the temperature will differentiate between the digit 0 and the null character, what will lead to a more esthetic and efficient display.

T(ºC)	T(F)
-19,5	-3,1
-19,0	-2,2
-18,5	-1,3
-18,0	-0,4
-17,5	0,5
-17,0	1,4
-16,5	2,3
-16,0	3,2
-15,5	4,1
-15,0	5,0
-14,5	5,9
-14,0	6,8
-13,5	7,7
-13,0	8,6
-12,5	9,5
-12,0	10,4
-11,5	11,3
-11,0	12,2
-10,5	13,1
-10,0	14,0
-9,5	14,9

Table 2. Conversion table from BCD in degrees Centigrade to BCD in degrees Fahrenheit

2.4. Centigrade degrees to precision centigrade degrees

As an extra work in this project has been implemented also the lecture and display of the maximum precision values. This information is included in the Count remain and the Count per degrees ºC that are the 6th and the 7th Byte on the Scratchpad memory. The maximum precision temperature can be performed as showed in the equation 1.

$$TEMPERATURE = TEMP_{READ} - 0.25 + \frac{COUNT_{¿} - COUNT_{REMAIN}}{COUNT_{¿}} \quad Eq.1$$

All this calculus can be really difficult to be performed and can suppose a big amount of time and resources spend to develop not only a divider to perform the calculus. The alternative that has been performed is really simple and easy to implement since is based only in combinational logic.

If we study properly the datasheet of our thermometer, we can see that in reality the COUNT_PER_C Byte is wired by hard to 16. The only Byte that give us new information is the COUNT_REMAIN Byte. This byte will vary from 1 to 16. This will give us the 16 new possible precision values for every integer instead of the 2 possible decimal values for the non-precision lecture.

If we study the table of non-precision values versus the precision values, we can see that decimal digits won't vary for the count remain value in this three ranges, the positives, the negatives and the zero. This leads that we have the decimal values only with a decoder structure.

For the non-decimal part, we can notice that will coincide the non-precision value with the precision value, except for the positives and negatives that we have to subtract one unit to the integer part for the values of count remain higher than 12 for the positives and lower than this same value for the negative ones. This operation can be performed with some concatenated adders/subtractors. Finally, the sign can be maintained for the positive and negative values and added if the non-precision value is 0 and the count remain is again higher than 12.

Count remain	Non precision value	Precision value	Non precision value	Precision value	Non precision value	Precision value
1	2	2,6875	0	0,6875	-2	-1,3125
2	2	2,6250	0	0,6250	-2	-1,3750
3	2	2,5625	0	0,5625	-2	-1,4375
4	2	2,5000	0	0,5000	-2	-1,5000
5	2	2,4375	0	0,4375	-2	-1,5625
6	2	2,3750	0	0,3750	-2	-1,6250
7	2	2,3125	0	0,3125	-2	-1,6875
8	2	2,2500	0	0,2500	-2	-1,7500
9	2	2,1875	0	0,1875	-2	-1,8125
10	2	2,1250	0	0,1250	-2	-1,8750
11	2	2,0625	0	0,0625	-2	-1,9375
12	2	2,0000	0	0,0000	-2	-2,0000
13	2	1,9375	0	-0,0625	-2	-2,0625
14	2	1,8750	0	-0,1250	-2	-2,1250
15	2	1,8125	0	-0,1875	-2	-2,1875
16	2	1,7500	0	-0,2500	-2	-2,2500

Table 3. Conversion table from BCD in degrees Centigrade to BCD in precision degrees Centigrade

2.5. Display module

To perform the display module, we will multiplex the seven segments display using a frequency of 1kHz in order to be done at a frequency above

100Hz in order to have a permanent display for our eyes (Retinal persistence) but below to 5kHz because of set up time of LEDs.

We will have to perform in the same way a BCD to Seven segments decoder. This conversion and the respective value displayed can be seen in the table 4. This decoder has been slightly changed in order to be able to perform the null character and the negative character.

BCD	7S	Display
0000	0000001	‘0’
0001	1001111	‘1’
0010	0010010	‘2’
0011	0000110	‘3’
0100	1001100	‘4’
0101	0100100	‘5’
0110	0100000	‘6’
0111	0001111	‘7’
1000	0000000	‘8’
1001	0001100	‘9’
1010	1111110	‘--’
1011	1100000	‘B’
1100	0110001	‘C’
1101	1000010	‘D’
1110	1111111	‘ ‘
1111	0111000	‘F’

Table 4. BCD to Seven Segments Decoder and values displayed

3. Hardware description

The hardware used in order to implement all the functionalities previously described is hierarchical and modular as required. Is as well Full synchronous, and has been designed in order to test all the blocks by itself, only modifying some code.

It has to be introduced here the concept of the modified finite state machines that has been used in the implementation of the project. The architecture and concept of the modified FSM used multiple times in this project can be observed in the figure 3. This modified FSM has the classical three blocks that corresponds to a process. The Next state decode that has a sensitivity list with all the input list. The sync process, that as his name indicate is the only synchronous process, which has the clock as sensitivity list.

The last process is the output decode that has only the current state as sensitivity list, this leads that the outputs depend of the current state and the inputs only. A clear way to increase the utility of the finite state machines is memorize the outputs, allowing to perform counters, timers, arithmetic operations and multiple other type of operations that are necessary memorized variables.

A way to implement memory process can be adding another block independent of the FSM. The way to perform this concerning the FSM will be adding the variable at the outputs and return this variable after being

memorized also as an input. Even this is a totally valid option that can lead to a clearer way to perform it there is two ways to do it.

On one hand, we can generate a memory block next to every FSM where we want to perform memorized variables, what will suppose a big number of blocks and not really clear structure. On the other hand, we can generate a general memory block which has a more logical meaning considering this a hard hardware design, what will suppose a block in the top structure of the designs and a lot of internal variables described in all the block hierarchy, what will suppose a code difficult to read and understand and really difficult to debug and test.

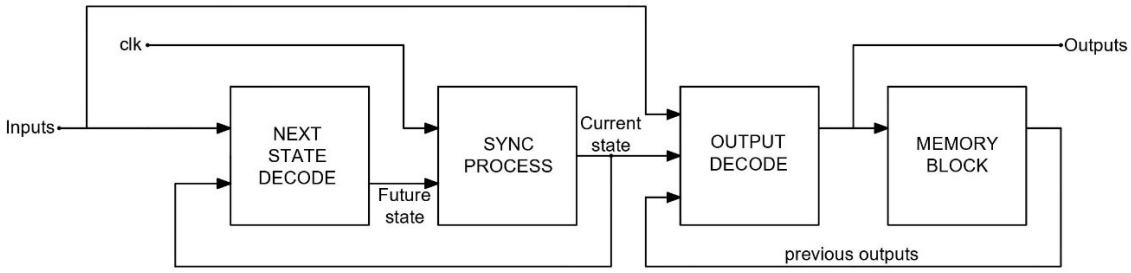

Figure 3. Modified finite state machines with memory

The solution proposed and implemented on this project consists in to generate a 4th process to the code of every FSM, also synchronous, that has to be implemented a memorized variable. This block will perform as some registers and will allow us to easily test the MFSMs (memorized variables finite state machine). From now on we will use this term really regularly since this type of architecture has been really used on the designs of this project.

The full system structure can be observed in the figure 4. As we can observe there is only four inputs for the full system. Those are the clock and the reset that will be the general ones for all the synchronous designs performed. The “sel” that will be two switches in our FPGA to perform if we want to select to display the temperature in degrees centigrade, in degrees Fahrenheit or in a precision degree centigrade and the “ow_line_in” that will perform the input of the one wire line. For the outputs we can distinguish the “ow_line_out” that will perform the output of the one wire line, the AN that will perform the multiplexing 7 segments display, the “BCD” that will bring the 7 segments decoded information and the “dp” that will perform the dot point.

The blocks inside this first Top structure that will be explained in detail in the next chapter are the Top communications structure that will perform all the one wire protocol, an MFSM for the conversion from binary to degrees centigrade, a second MFSM for the conversion from binary to degrees Fahrenheit, the combinational logic that will perform the conversion form degrees centigrade to precision degrees centigrade that has already been explained, a big multiplexer used to switch between the three modes to display the temperature and finally the display module used his name indicates to perform the display of the temperature to the seven segments.

We can notice also that there is a component called BUFT that will perform the connection with the one wire line. This component will suppose a high impedance with the line when the ‘T’ input is connected to a logic ‘1’ which will lead to the line to be pulled-up by the resistor connected to VCC that can be seen on the figure 1 and will suppose a 0 to the line if the ‘T’ input in connected to a logic ‘0’ since the line will be connected to the input of the BUFT that will be connected to ground by hard.

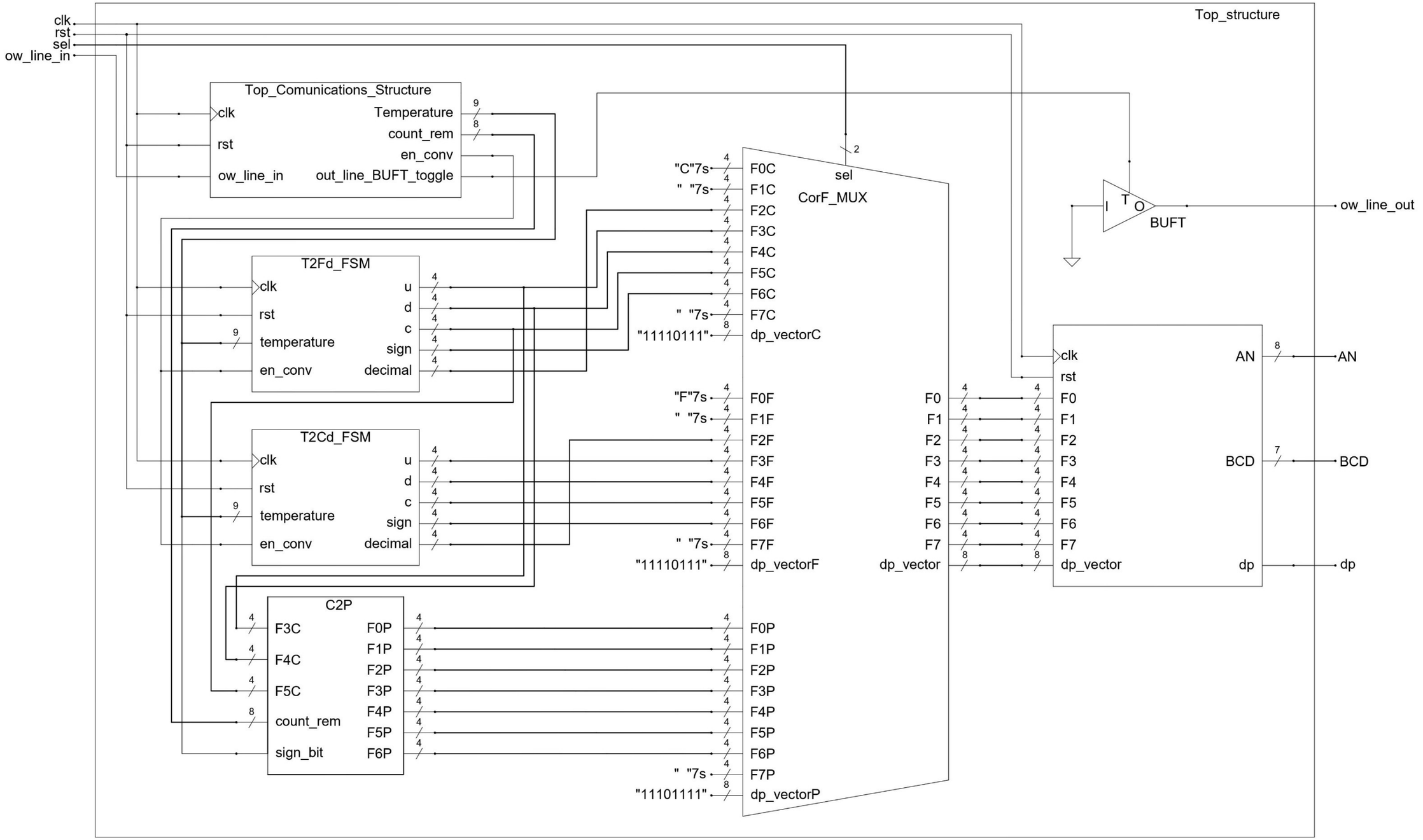

Figure 4. Top Structure

3.1. Top Communications structure

The communications structure block as we mentioned before is the one used to perform all the one wire protocol used to stablish the communication with the sensor. This block performs the initialization sequence, as well as to generate the write time slots and the read time slots.

This block will have the clock and the reset as inputs since has synchronous blocks inside it. It has the “ow_line_in” since we have to sample the line not only in the read time slots, but in the initialization sequence also. As outputs we will have the information needed to display the temperature in the seven segments, the “Temperature” array conformed by the less significant byte of the two bytes of the temperature and the less significant bit of the second byte that will give us the sign of the temperature.

We will obtain as an output the COUNT_REMAIN byte and not the COUNT_PER_C byte since as explained before, this don’t apport any information since is wired by hard. Finally we will have as an output the “en_conv” bit which will allow us to enable the converter blocks T2Cd_FSM and T2Fd_FSM and the “out_line_BUFT_toggle” that is the byte that will control the state of the line being this output connected to the ‘T’ input of the BUFT buffer.

Inside the “Top_comunications_structure” we can find five MFSM, called “BRAIN”, “TIMER_FSM”, “Initialization_FSM”, “Write_Routines_FSM” and “Read_Scratchpad_FSM”, all of those blocks are synchronous, so will be all connected by the reset and the clock inputs. We can notice also that the full project has been developed with only one timer, the “TIMER_FSM” that can be enabled and presaled by all the other blocks of the “Top_communications_structure” block. This can be explained since the timer won’t be never enabled by two different blocks at the same time.

The “Initialization_FSM” as its name indicates, is the block that will generate the Initialization routine. The “Write_Routines_FSM” is the block that will generate the write time slots and will send the necessary commands. The “Read_Scratchpad_FSM” will generate the Read time slots and will sample the line at the precise moment.

Finally, the “BRAIN” block is as his name indicates the block responsible to lead the rest of blocks. This one will enable the other blocks at the precise moment and will in all moment indicate what routine write to the “Write_Routines_FSM”. This block has to enable the conversion of the value of the temperature through the “en_conv” output. As well, all the output data will pass through this block.

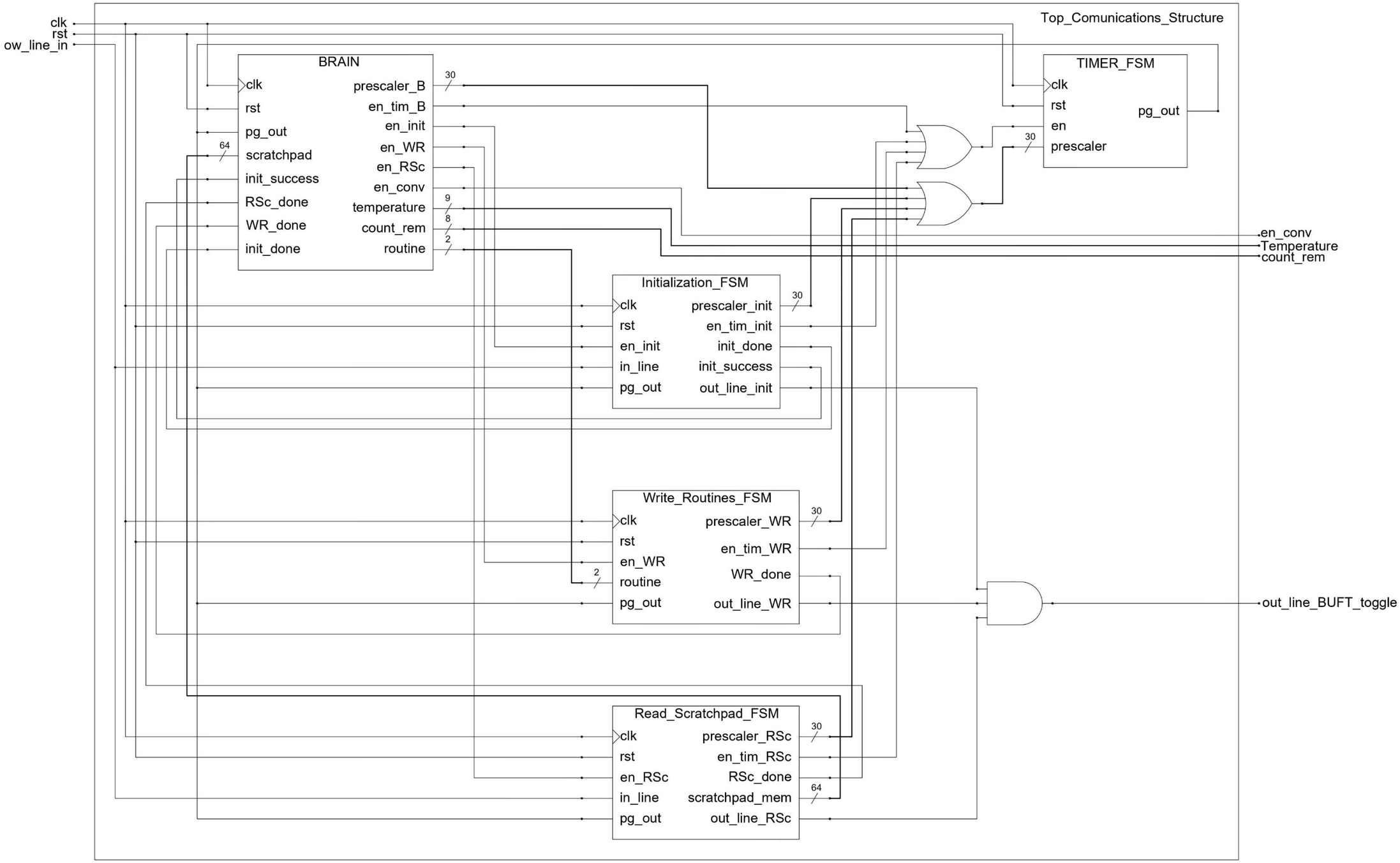

Figure 5. Top Communications Structure

3.1.1. TIMER_FSM

The Timer MFSM is the only timer used in all the project. This Timer can be enabled by all the blocks responsible to perform the one wire protocol. As we said is a timer based on the fact that the time of change from one state to the other is one clock cycle. The FSM state diagram of this block can be seen in the figure 6 and some of the outputs are represented on the table 5.

To understand the state diagrams of this project has to be kept in mind that exist “two types of variable”, the non-memorized as “pg_out” that are only taking the set ‘1’ values only in the state where are indicated and will take a reset ‘0’ value in the states where are not indicated and the memorized ones as “count” that will be set to a certain value in a state and if are not modified the variable will keep this value.

The “two types of variables” can be distinguished here by the way to be used, in one hand the non-memorized variables when are set, are only indicated, as in the state 5 for instance. On the other hand, the memorized variables have to be modified by the operator “<=”. We have to distinguish also when we actualize a variable with the value ‘0’ means that this variable has one bit and is a “std_logic” if we actualize it with a value “0” means that has more than one bit, is a “std_logic_vector” and all of the bits are set to the value ‘0’.

Even if it’s not on the syntax of the outputs, if we for instance do the operation “count = count_p+1” mean that we are adding one decimal unit to the previous value of count which is a “std_logic_vector”, what means that we will have to do some variable conversion in order to do the arithmetic operation like unsigned or integer.

Concerning the transitions between states. If it’s not specified an else condition, means that if the condition of the transition is not met, the next state will be itself for example the S0 state. The “x” transition means that there is no condition to switch from a state to another state, and thus, will only stay on that state during one clock cycle.

If we take a look to the state diagram of the figure 6, we can see that the steady state is S0. In S1, the prescaler value will be read and the count until the prescaler value can start. That count will be done in the states S3 and S4 where will add a unit to the counter in every pass through one of this two states. After the prescaler value is reached the timer will emit a clock pulse in state S5 that announce the end of the count.

The state S6 has been added only to permit wait on a same state if we activate the timer through another FSM. We avoid a false activation of the timer in this way. If this S6 hasn’t been added then the activation and the wait until the pg_out sets to one should be done in two different states.

Output	**Code**
O	mem_prescaler <= prescaler
O1	count <= count_p+1

Conditions	**Code**
C	count = mem_prescaler

Table 5. Output and Conditions table for the TIMER_FSM block

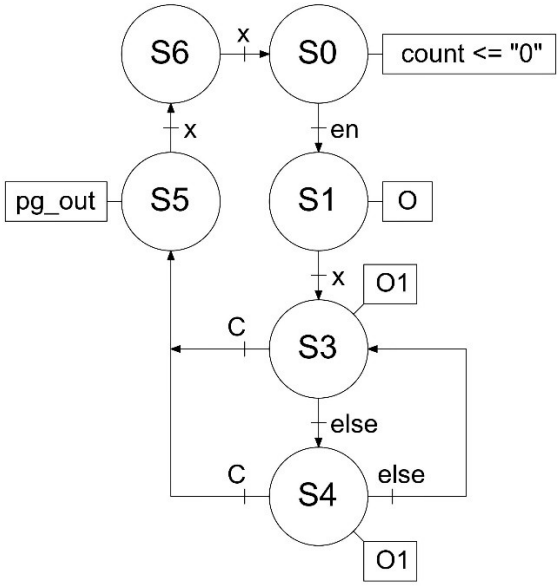

Figure 6. TIMER_FSM state diagram

3.1.2. BRAIN

The Brain is the main MFSM of the “top_communications_structure” block. If we look at the state diagram of this block in the figure 7 can be easily seen that is a more technical version of the sequence of 1 wire protocol that can be seen on the figure 2.

There is not a steady state in this FSM but the starting and reset state is S0 where we set the flag memory to ‘0’. This memorized variable called flag is the one that will make take a route or another. Being not necessary to define twice the states to enable the initialization routine and sending the skip rom command. In S1, we will enable the “Initialization_FSM” to proceed and we will jump to the S2 state after receiving that the initialization sequence has been done and has been received the presence pulse.

In S2 and S2_2 states we will enable the “Write_Routines_FSM” block with a routine “11” which corresponds to a skip_ROM as can be seen in the table 6. After this done, we will decide which route take. If the flag is ‘0’ state S4 and S4_1 will be activated in order to enable the “Write_Routines_FSM” block with a routine “10” which corresponds to a Convert T command and then we wait for the 780 milliseconds and swap from 0 to 1 the flag memory.

In the other hand If the flag is ‘1’ state then S7 and S7_1 states we will enable the “Write_Routines_FSM” block with a routine “01” which corresponds to a Read_Scratchpad. After that S8 and S8_1 states will enable the “Read_Scratchpad_FSM” block to send the read time slots and to sample the line to read the full memory. Once this is done, we wait for the 5 milliseconds and we will read the scratchpad memory from the “Read_Scratchpad_FSM” block and we will enable the conversion of temperature and swap from 1 to 0 the flag memory in order to re-initialize the cycle.

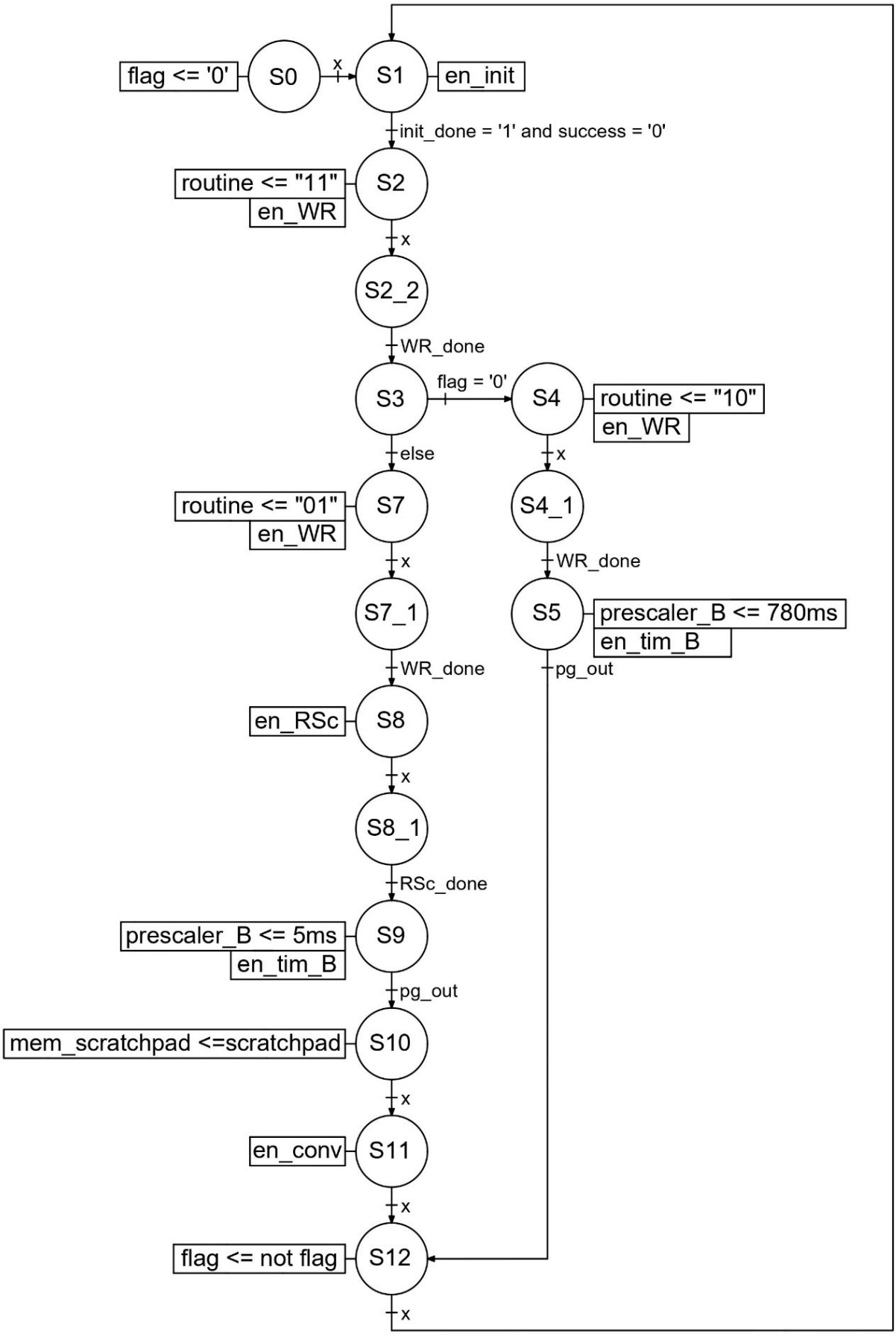

Figure 7. BRAIN state diagram

3.1.3. Initialization_FSM

“Initialization_FSM” is the block of type MFSM responsible of the Initialization routine. The state diagram of the figure 8 is the diagram that corresponds to that block and is composed of an almost sequential row of states. Before describing this diagram, it has to be specified that the variable “out_line_init” as the variable “out_line_WR” and “out_line_RSc” for the next blocks take this time as default the value ‘1’. This means that when this variable appears in the diagram is reset to the value ‘0’, since this variable represents the state of the line. When take a value ‘0’ the line is pulled down by the BUFT Buffer.

As in the other finite state machines, the S0 state is the steady state. After being enabled in S1 the timer is initialized with a prescaler value of 500 microseconds. While this time, the line will be pulled down. During the S2 state, 60 microseconds the line will be free again. We can realize easily that those timings represent the ones specified in the datasheet of the thermometer to perform the Initialization routine that can be seen in the figure 9.

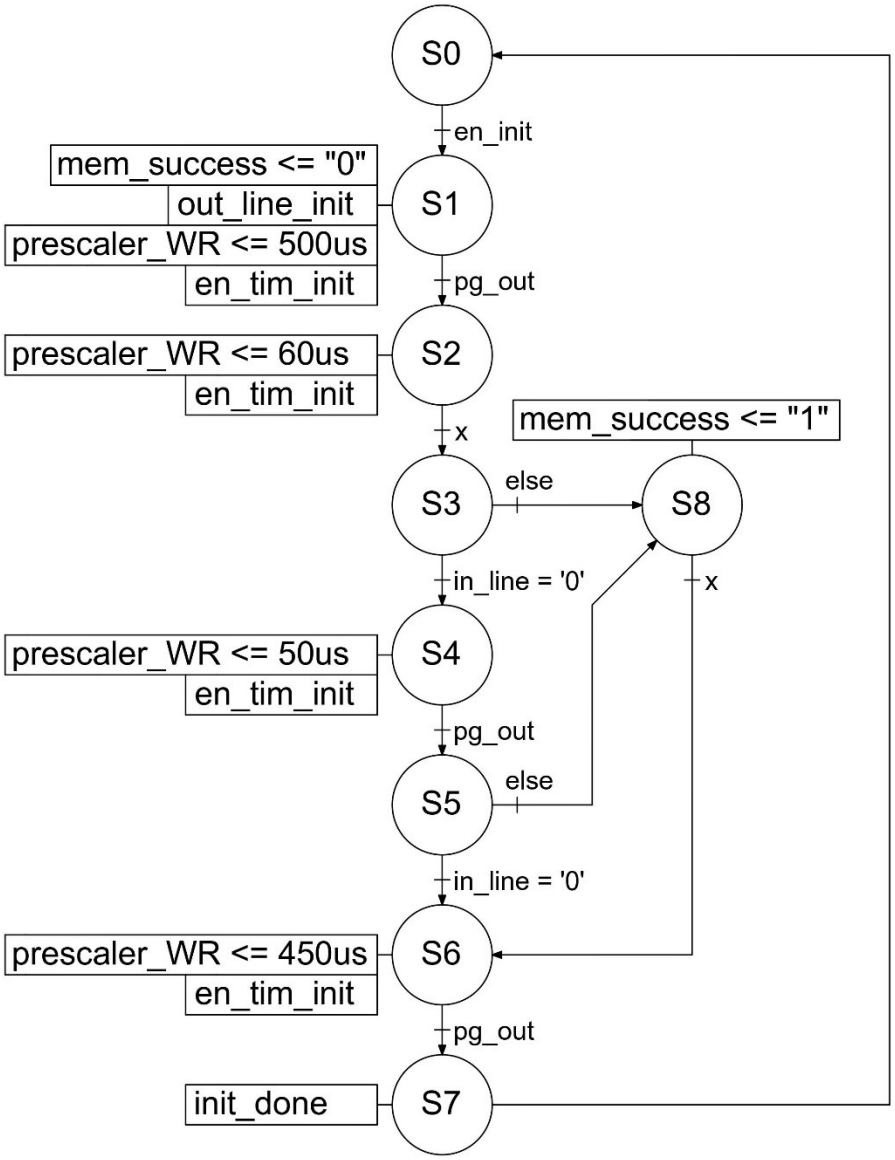

Figure 8. Initialization_FSM state diagram

The transition from S3 to S4 we do the first sampling on the line, if its value is ‘0’ we will wait for 50 microseconds more and we will sample again in the transition from the state S5 to S6. If both samples are ‘0’ we consider as a valid initialization pulse and will return to the S0 state after indicate that the initialization has been done. If only one of the samples is ‘1’ then we will pass through the S8 state where the variable “mem_success” will get set with a ‘1’ value before indicating that the sequence is done.

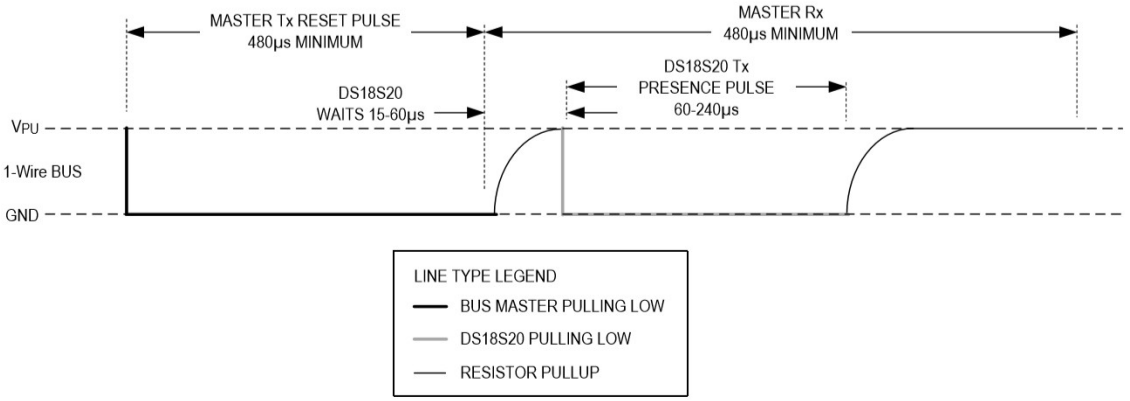

Figure 9. Initialization timing

3.1.4. Write_Routines_FSM

The “Write_Routines_FSM” block is a MFSM that as his name indicates will generate the commands, performing the write time slots with value 0 and 1. The different Commands that this block can generate are the observables in the table 6. We can distinguish the Skip ROM Command, the Convert T command and the Read Scratchpad command. Every one of those commands will be generated in a byte composed of eight write time slots codified in hexadecimal and transmitted in binary starting by the less significant bit.

Command	**HEX notation**	**BIN command**	**Code**
Skip ROM	[CCh]	11001100b	11
Convert T	[44h]	01000100b	10
Read Scratchpad	[BEh]	10111110b	01

Table 6. Commands, notation of those and code to execute them

If we now observe to the state diagram in the figure 10, we can see the two different paths. The one to generate a ‘0’ write time slot and another to generate a ‘1’ write time slot.

The S0 state, as for the others FSMs represent the steady state. After being enabled we will take the routine that has to be generated picked by the “BRAIN” block. And we will enter to the write time slots generator. In the conditions C0 and C1 totally specified in the table 7 are reflected the bit position depending on what command has to be written. If the condition C0 is met a ‘0’ write time slot will be generated, since we will stay in S5 for 70

microseconds with the line pulled down as indicated and then 5 microseconds of recovery at the state S6 allowing the line to get pulled up again.

If the condition C1 is met a ‘1’ write time slot will be generated, since we will stay in S3 for only 10 microseconds with the line pulled down and then adding the timing of S4 and S6 we will let the line get recover for 65 microseconds. After that in S7 we will add a unit to the counter in order to continue with the scripture on the next less significant bit, until we reach the 8th when after write it will return to the steady state again passing trough S3 and releasing all the variables used in the write of the complete command.

We can again realize that what we are doing is following the timing diagram of the write time slots indicated in the thermometer datasheet that can be seen in the figure 11, but in our case the timings are taken with security to be sure that are corrects and we don’t work at the limit.

Condition	**Code**
C0	((mem_routine = "11") **and** (count = 0 **or** count = 1 **or** count = 4 **or** count = 5))**or** ((mem_routine = "01") **and** (count = 0 **or** count = 6))**or**((mem_routine = "10") **and** (count = 0 **or** count = 1 **or** count = 3 **or** count = 4 **or** count = 5 **or** count = 7))
C1	((mem_routine = "11") **and** (count = 2 **or** count = 3 **or** count = 6 **or** count = 7))**or** ((mem_routine = "10") **and** (count = 2 **or** count = 6))**or**((mem_routine = "01") **and** (count = 1 **or** count = 2 **or** count = 3 **or** count = 4 **or** count = 5 **or** count = 7))

Table 7. Condition table for the TIMER_FSM block

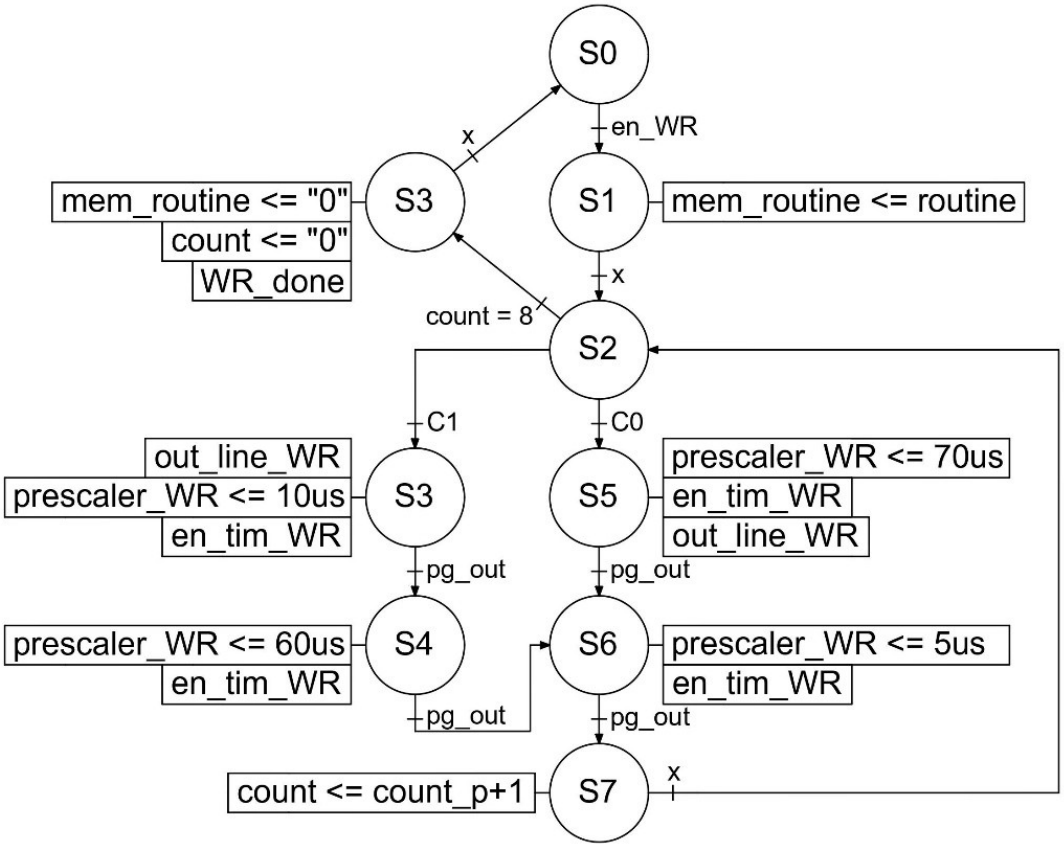

Figure 10. Write_Routines_FSM state diagram

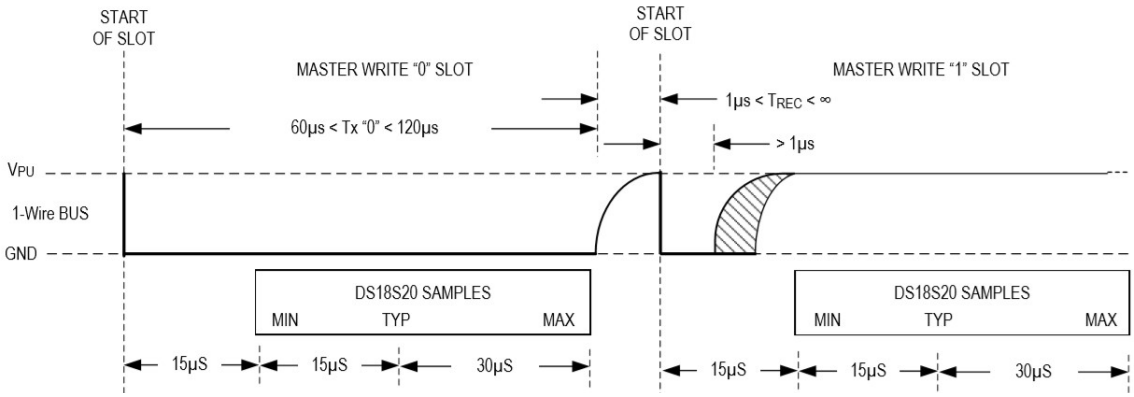

Figure 11. Write time slots timing diagram.

3.1.5. Read_Scratchpad_FSM

The last MFSM to be exposed id the “Read_Scratchpad_FSM”. This MFSM block deal with the generation of the read time slots and the sampling of the line at the indicated moment.

S0 is the steady state and we will get out of it after being enabled. After that will start the generation of the read time slots. This will start is S1 waiting with the line pulled down for two microseconds. After those two will stay in S2 for 12 microseconds more with the line released. Then we will keep the value of the line in the position 63 of the “scratchpad_mem” variable that will store the complete scratchpad memory.

After that sampling, we will wait until the end of the read time slot in S4 for 55 more microseconds. Once this is done if the scratchpad memory is not totally read, we have to shift left the memory to wait for next information bits. This will be done in the state S7 and also will add one unit to the total amount of read bits. When this counter reaches the bit number 64 will release the counter in the state S6 and will return to the state S0.

We can again realize that what we are doing is following the timing diagram of the read time slots indicated in the thermometer datasheet that can be seen in the figure 12, but in our case the timings are taken with security to be sure that are corrects and we don’t work at the limit.

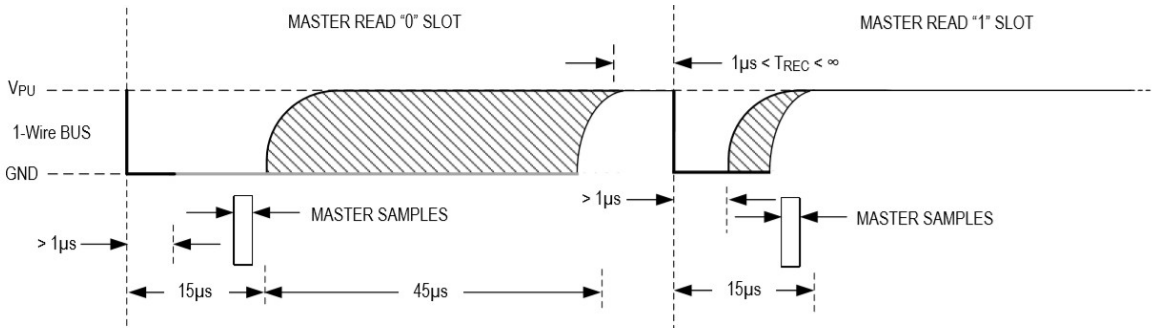

Figure 12. Read time slots timing diagram.

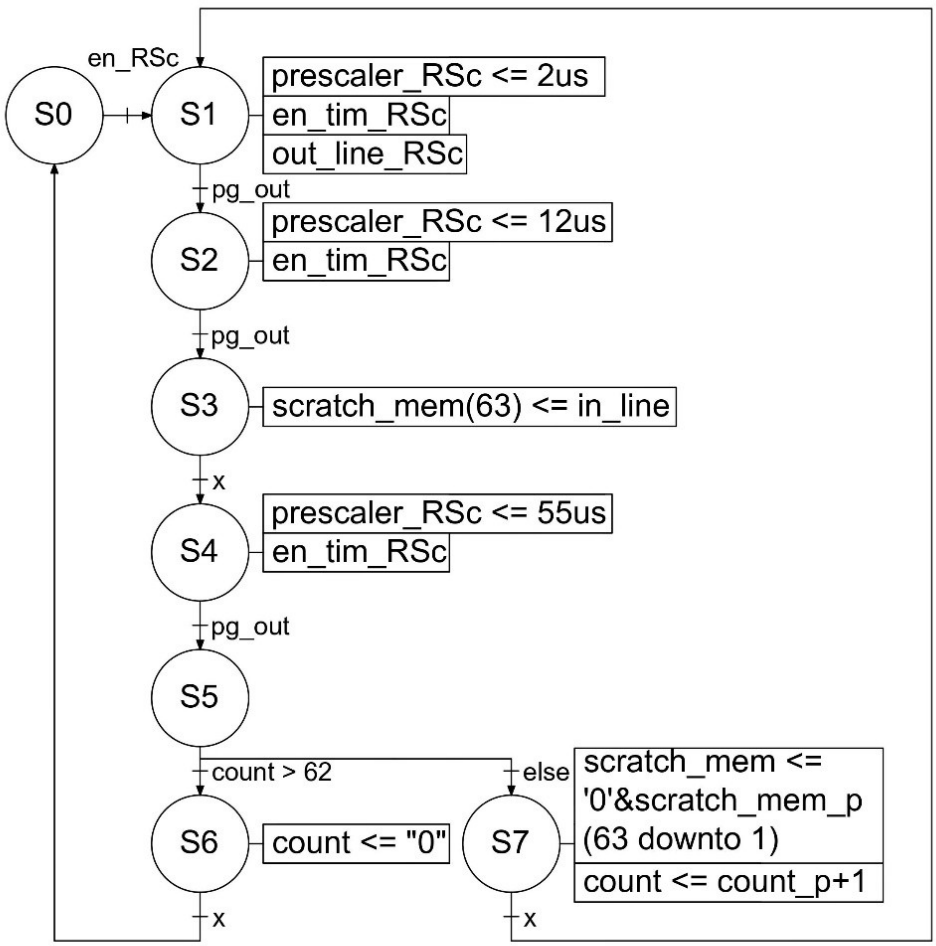

Figure 13. Read_Scratchpad_FSM state diagram

3.2. T2Cd_FSM

The "T2Cd_FSM" which is localized outside the "Top_Communications_Structure" block is the responsible to convert from the temperature data compiled by the "Read_Scrachpad_FSM" block to a displayable data (In degrees centigrade and in BCD). The corresponding state diagram can be seen in the figure 13.

After being enabled, all the counters will be released and the temperature input is stored in the "meme_value" register in S1. After that if the temperature is lower than 0 the negative ("1010") will be set in the corresponding register and the value of temperature will be the complementary adding one to correct the biased Two's complement, else the sign will take the null value ("1110").

Once all is set, the count will start from S3 where the decimal will take the value '5' and the counter will add 1 to the count. In S4, S5 and S6 we will add a unit to the count in BCD and the decimal will reset to 0. The conditions to enter on S4, S5 and S6 and their outputs can be seen in the table 8. If in S2 the counter is equal to the temperature value (Temperature equal to 0^{o} C) or in S4 or in S8 after the counter is equal to the temperature (C1) value means that the count is finished and we have in the "mem_u", "mem_d", "mem_c", "decimal" and "mem_sign" registers the correct value

of temperature in BCD. Then we can return to the steady state S0 where all that registers will keep in the last value until the FSM is enabled again.

Output	**Code**
O1	mem_u <= “0”; mem_d <= “0”; mem_c <= “0”; counter <= “0”; mem_value <= temperatura;
O2	**If** mem_value(8) = 1 **then** mem_sign <= “1010”; mem_value(7 downto 0) <= (not mem_value(7 downto 0))+1; **else** mem_sign<=”1110”; **end if;**
O3	Decimal <= “0101”; counter <= counter_p+1;
O5	mem_u <= “0”; mem_d <= “0”; mem_c <= mem_c_p+1; counter <= counter_p+1; decimal <= “0”;
O6	mem_u <= “0”; mem_d <= mem_d_p+1; counter <= counter_p+1; decimal <= “0”;
O7	mem_u <= mem_u_p+1; counter <= counter_p+1; decimal <= “0”;

Conditions	**Code**
C1	mem_value(7 downto 0) = counter
C2	mem_value(7 downto 0) /= counter **and** mem_u = “1001” **and** mem_d = “1001”
C3	mem_value(7 downto 0) /= counter **and** mem_u = “1001” **and** mem_d /= “1001”
C4	else

Table 8. Output and Condition table for the T2Cd_FSM block

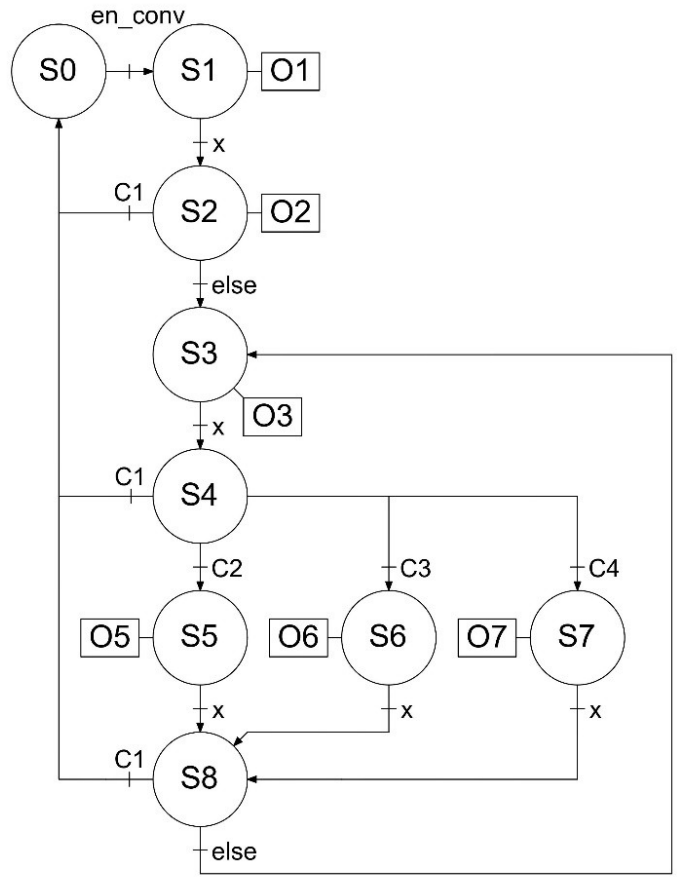

Figure 14. T2Cd_FSM state diagram

3.3. T2Fd_FSM

The “T2Fd_FSM” which is also outside the “Top_Communications_Structure” block will convert from the temperature data compiled by the “Read_Scrachpad_FSM” block to a displayable data (In degrees Fahrenheit and in BCD). The state diagram can be seen in the figure 14.

We first start memorizing the value of the temperature in the mem_value register in S1. Once this is done, if the temperature is lower than 0 the value of temperature will be the complementary adding one to correct the biased Two’s complement in S2. If we face a negative value and the memorized value is bigger than 35 then we will subtract 36 to this value and we will set negative the sign value as well as set to 4 the decimal value, else will add 35, will set the sign to a null character and the decimal to 5. All of that will be done in S3. If it’s not clear why all of this is done, check the chapter 2.3 of this report.

Once this preparation is done, we can start the count. We will enter to the states S5, S6, S7 or S8 adding one unit in every count except if the decimal value meets the 0. And we will decrease the units of the decimal part or will release the counter to 9 if this value reaches the 0. The conditions to enter to those states and the outputs of this ones can be found in the table 9.

If in S2 or in S4 the temperature value meets with the counter value (C1) then the steady state S0 will be reached and the values of the temperature in BCD and in Fahrenheit degrees will stay in the “mem_u”, “mem_d”, “mem_c”, “decimal” and “mem_sign” registers.

Output	**Code**

O2	**If** mem_value(8) = ‘1’ **then**
	mem_value(7 downto 0) <= (not mem_value_p(7 downto 0))+1;
	end if;

O3	**If** mem_value(8) = 1 **and** mem_value(7 downto 0) > “00100011”**then** mem_value(7 downto 0) <= mem_value_p(7 downto 0) - 36; mem_sign <= “1010”; decimal <= “0100”; **else** mem_value(7 downto 0) <= mem_value_p(7 downto 0) + 35; mem_sign <= “1110”; decimal <= “0101”; **end if;** mem_u <= “0”; mem_d <= “0”; mem_c <= “0”;
O5	Decimal <= “1001”; counter <= counter_p+1;
O6	mem_u <= “0”; mem_d <= “0”; mem_c <= mem_c_p+1; counter <= counter_p+1; decimal <= decimal_p-1;
O7	mem_u <= “0”; mem_d <= mem_d_p+1; counter <= counter_p+1; decimal <= decimal_p-1;
O8	mem_u <= mem_u_p+1; counter <= counter_p+1; decimal <= decimal_p-1;

Conditions	**Code**
C1	mem_value(7 downto 0) = counter
C2	mem_value(7 downto 0) /= counter **and** decimal = “0”
C3	mem_value(7 downto 0) /= counter **and** decimal /= “0” **and** mem_u = “1001” **and** mem_d = “1001”
C4	mem_value(7 downto 0) /= counter **and** decimal /= “0” **and** mem_u = “1001” **and** mem_d /= “1001”
C5	else

Table 9. Output and Condition table for the T2Fd_FSM block

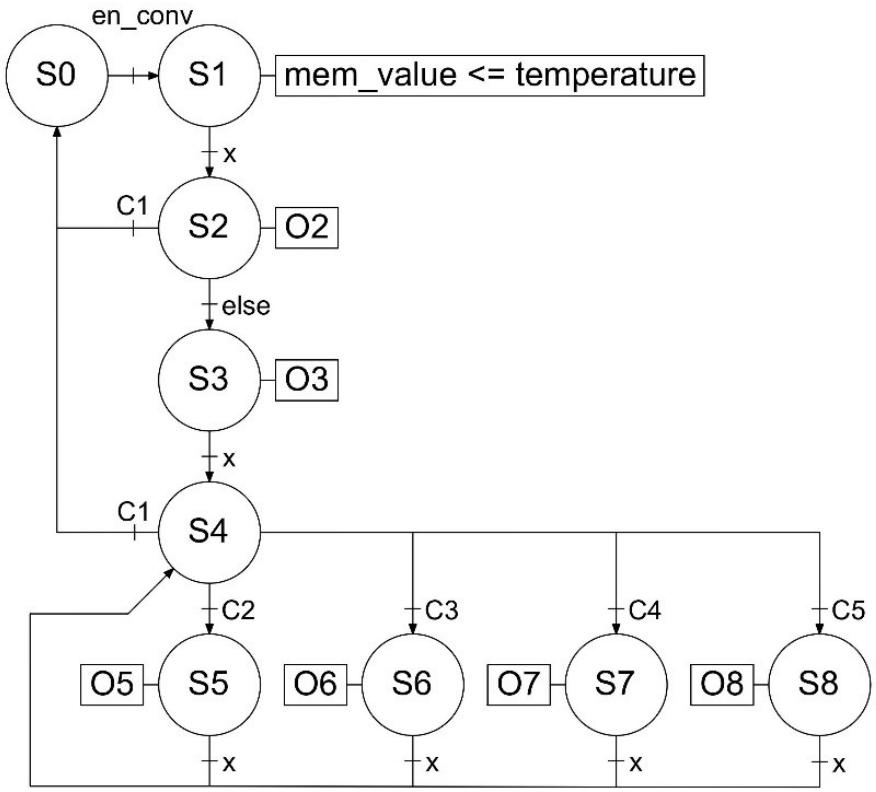

Figure 15. T2Fd_FSM state diagram

3.4. display_module

The last block to be described is the “display_module” block, which will not be explained in detail because is the same architecture used in the development of one of the TDs developed in class. This block converts the BCD data to 7 segments data and can be seen in the figure 14. In this case, the data has to be multiplexed at 1kHz as has been argued previously. This block is formed by the data control, the BCD to Seven Segments and the AN_GEN or the generator of the AN output.

The easiest part to describe is the “BCD2SS” block which is a decoder form BCD data to the seven segments data. This conversion can be observed in the table 4 and is a full combinational block based on a simple process.

The data control is also a combinational block. In this case it is a multiplexer controlled by the AN signal. Due to this signal is different every millisecond the data value will switch form F0 to F7 every millisecond.

The “Clock_divider” and the “an_counter” are the responsible full synchronous blocks inside the “AN_GEN” to generate the signal AN that will enable one of the eight seven segments display. The clock divider is based in a counter. If we count clock cycles, we are able to do a timer, such as the “TIMER_FSM”, with the difference that this will count a fixed number of clocks and will keep generating the ‘s’ signal (a clock cycle after the count) indefinitely.

Finally, the “an_counter” is a really simple FSM, which in every clock cycle will generate the eight bits code corresponding to enable one of the seven segments display.

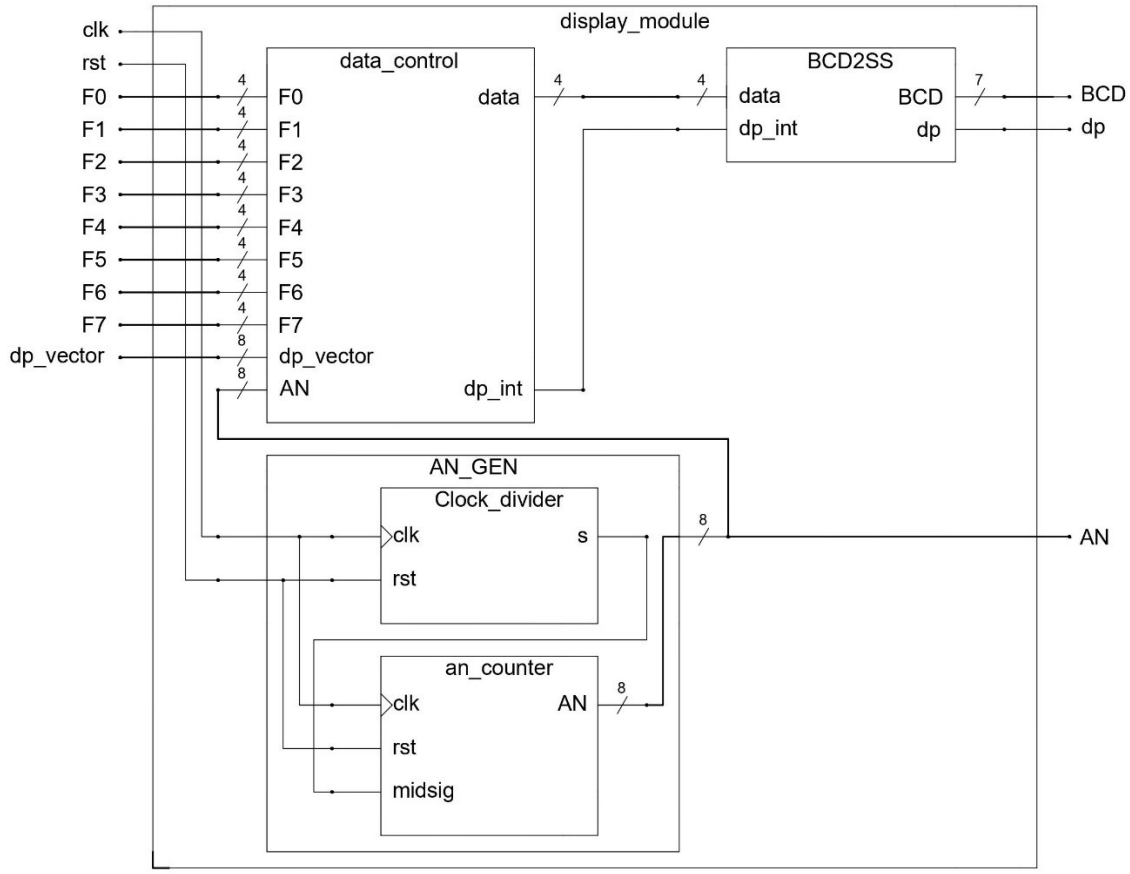

Figure 16. display_module structure

4. Conclusions

Once the project has been finished, we can extract some conclusions from it.

- The MFSM is a really easy way to solve a lot of Hardware problems, the versatility, the simplicity once one can understand the concept and the applications that can be developed with this concept are enormous. Almost all the project is based in this concept and that is why the part of the communications of the project that can become quite complex has been developed and implemented in less than a week.
- All the project is full synchronous working with a crystal of 100MHz. Hierarchical and modular also. There is no modulus that can’t be tested by its own.
- The project has been Implemented and tested and work absolutely fine. To test the negative temperatures has been added a not gate in the output of the temperature array at the output of the “Top_Communications_Structure” block. Even that, the top communications structure by itself have been tested with a cold spray

and oscilloscope too since was designed and implemented before the start of the lockdown.

- If we add all the maximum times that the general cycle of communications reflected in the figure 2 take 9.3 milliseconds. If we now add the conversion time will reach the 790 milliseconds. A way to improve it can be trying to optimize the conversion time. This can be done sending read time slots during the conversion period until the answer to those turn out to be ‘1’.