Politecnico di Torino

DIGITAL SYSTEMS ELECTRONICS

04OIHNX - A.A. 2024/2025

Prof. G. Masera

Laboratory assignment no. 6 – A simple digital filter

DUE DATE: 22/04/2025 DELIVERY DATE: 25/04/2025

GROUP 08 - Contributions:

- Daniele Becchero (308299) – 33.3%
- Bohotici Ionut Viorel (300061) – 33.3%
- Simone Viola (310779) – 33.3%

The members of the group listed above declare under their own responsibility that no part of this document has been copied from other documents and that the associated code is original and have been developed expressly for the assigned project.

Introduction

This report presents the design phase and the validation of a circuit that executes a digital filtering function using two memory elements, MEM_A and MEM_B, where input samples are stored and processed sequentially. The filtering process follows a defined discrete-time equation, incorporating a multiplierless data path that relies solely on addition and bit-shift operations to perform necessary computations.

Given the constraints of memory size and precision, a saturation mechanism is introduced to ensure that output values fit within the 8-bit representation. Once all data samples are processed and stored in MEM_B, the system enters a waiting state, ready to execute another filtering sequence upon receiving a new start signal.

The purpose of this report is to design the internal architecture and facing implementation challenges associated with a complex circuit.

Pseudocode

A C-like pseudocode has been developed to represent the filtering algorithm and clarify the logic behind the FSM controlling the operations. The text file is named *GROUP08_LAB06_PSEUDOCODE_C.txt*.

Delivered files

The following files have been delivered for the implementation of the complete digital filter:

- `digitalFilter_tb.vhd`: testbench designed to simulate the overall functionality of the digital filter;
  - `digitalFilter.vhd;`
- `digitalFilter.vhd:` main file that integrates and defines the overall architecture of the digital filter;
  - **Datapath**
    - `Datapath.vhd:` digital filter datapath main file
      - **Modulo counter**
        - `moduloCounter.vhd:` architecture of modulo-counter
          - `elementaryCounter.vhd :` elementary counter component
          - `flipflop.vhd:` flip flop D with asynchronous reset
          - `mux.vhd:` multiplexer component
          - `comparator.vhd:` comparator component
      - **Processing Unit**
        - `processingUnit.vhd`: architecture of processing unit
          - `RCA_13bit.vhd:` 13-bits Ripple Carry Adder component
          - `multiplier_025.vhd:` “two shift from right” multiplier component
          - `multiplier_05.vhd:` “one shift from right” multiplier component
          - `multipler_2.vhd:` “one shift from left” multiplier component
          - `multiplier_4.vhd:` “two shift from left” multiplier component
          - `shiftRegister.vhd:` shift register component
          - `regn.vhd:` register with parametric size
        - `processingUnit_tb.vhd:` testbench designed for the Processing Unit
          - `processingUnit.vhd`: architecture of processing unit
      - **Converter Unit**
        - `converterUnit.vhd`: architecture of converter unit
          - `RCA_11bit.vhd:` 11-bits Ripple Carry Adder component
          - `fullAdder.vhd:` full adder component

        - `MUX2x1_11bit.vhd`: 11-bits, 2 inputs, 1 output mux
        - `MUX3x1_8bit`: 8-bits, 3 inputs, 1 ouput mux
        - `intervaldetector.vhd`: behavioral interval signed detector
        - `regn.vhd` *(already used in previous structures)*
    - `converterUnit_tb.vhd`: testbench designed for the Converter Unit
        - `converterUnit.vhd`: architecture of converter unit
  - `mux10bit.vhd`: 10-bit mux
  - `delayUnit.vhd`: delay unit made of multiple registers
  - `regn.vhd` *(already used in previous structures)*
- `datapath_tb.vhd`: testbench designed for the Converter Unit
  - `Datapath.vhd`: digital filter datapath main file
- **Control Unit**
  - `controlUnit.vhd`: digital filter Control Unit main file
  - `controlUnit_tb.vhd`: testbench designed for the Control Unit
    - `controlUnit.vhd`: digital filter Control Unit main file
- **Memory**
  - `memory.vhd`: digital filter memory

Design description

The hardware of the digital filter is based on a complex system. The architecture features four building blocks: the control unit (CU), the data path (DP) and two memory elements (MEM), interconnected as shown below.

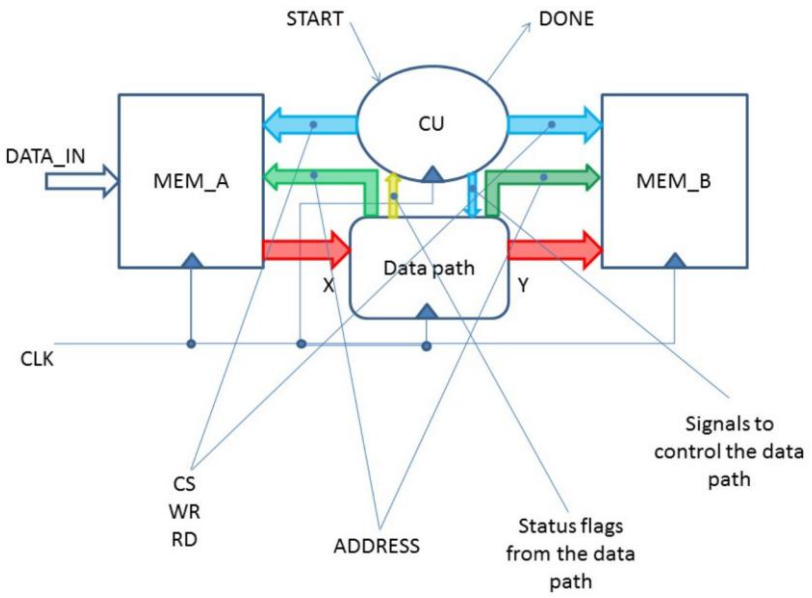

Figure 1 - Digital filter structure

The data path is the component responsible for executing all arithmetic and logic operations, while the control unit provides the necessary instructions to guide these operations. The control unit operates based on a finite-state machine and requires input status signals from the data path to transition between states. Similarly, the behaviour of the data path depends on control signals from the control unit to execute the appropriate operations.

The top-level entity of the design is the VHDL file named *digitalFilter.vhd*, and it includes the control-unit, the data path and the two memory elements, replicating the scheme above.

Memory element

This component is a memory element designed as a register file. Its architecture consists of an array of 1024 registers, each capable of storing an 8-bit signed value.

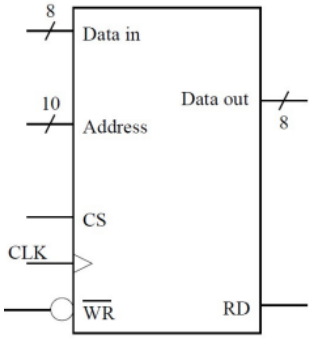

Figure 2 - Memory element building block

This component has been described in VHDL using a behavioural style, as requested in the assignment requirements. The file name is *memory.vhd*; the entity includes three input ports to control its behaviour:

- CS (Component Selector): Enables the memory functionality when HIGH.
- WR (Write): Sets the device to write mode when HIGH.
- RD (Read): Sets the device to read mode when HIGH.

Additionally, the component features a 10-bit wide input port for selecting the correct address in the array. It also includes two 8-bit wide data ports: one for input data and one for output data. The VHDL description of the component ensures that the entire memory is set at 0 at startup.

Writing to the memory is a synchronous operation described with a process block that monitors the rising edge of the clock. When both CS and WR are HIGH, the value at DATA_IN is written to the memory location specified by INTEGER_ADDRESS.

Reading from the memory, on the other hand, occurs asynchronously. When CS and RD are HIGH, the output signal DATA_OUT replicates the value stored at the location specified by INTEGER_ADDRESS. If either CS or RD is LOW, the DATA_OUT port transitions to a high-impedance state, as discussed in the course material about the DRAM READ timing.

In VHDL, the component has been described as follows:

```
-- Synchronous write process
process (CLK)
begin
     if rising_edge(CLK) then
          if CS = '1' and WR = '1' then
               mem(INTEGER_ADDRESS) <= DATA_IN;
          end if;
     end if;
end process;

-- Asynchronous read process
DATA_OUT <= mem(INTEGER_ADDRESS) when (CS = '1' and RD = '1')
else
     (others => 'Z'); -- High impedance when inactive
```

Notice that, since the array data structure requires access via an integer index, the binary address value is first converted into an integer for proper array accessing.

Control Unit

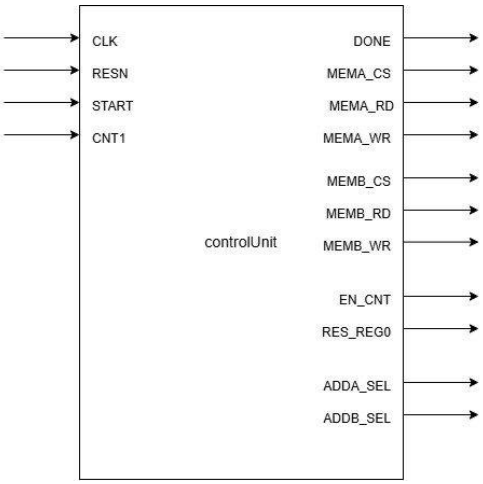

Figure 3 - Control unit block

The control unit is implemented by using a Moore-type FSM described in VHDL with a three-processes style. The state diagram is represented below.

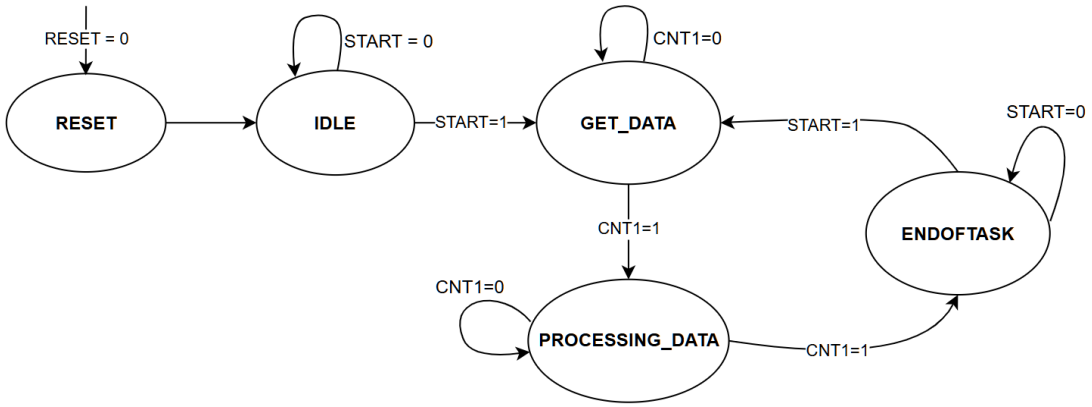

Figure 4 - Control unit state diagram

Each output signal, which depends only on the current state, is represented in Table 1.

STATE	RESET	IDLE	GET_DATA	PROCESS_DATA	ENDOFTASK
MEMA_CS	0	0	1	1	0
MEMA_RD	0	0	0	1	0
MEMA_WR	0	0	1	0	0
MEMB_CS	0	0	0	1	0
MEMB_RD	0	0	0	0	0
MEMB_WR	0	0	0	1	0
EN_CNT	0	0	1	1	0
RES_REG0	0	0	0	1	0
ADDA_SEL	0	0	0	0	0
ADDB_SEL	0	0	0	1	0
DONE	0	0	0	0	1

Table 1 - Output values of the control unit based on the FSM state

In the practical implementation on the FPGA, the FSM is synthesized by Quartus based on the user-specified synthesis settings. The states can be encoded using a minimal-bit approach, one-hot encoding, or other more complex encoding schemes to achieve the optimal balance between hardware resources (basically, elementary logic elements or look-up tables) and performances.

Datapath

Since the data path is the component that has to process the data stream, it is composed of many different building blocks.

Computing unit

This unit must process the input stream in order to compute the correct result to store in the second memory, according to the equation

$$Y(n) = -0.5 \cdot X(n) - 2 \cdot X(n-1) + 4 \cdot X(n-2) + 0.25 \cdot X(n-3) \qquad (1)$$

For the first three samples, the input values corresponding to negative indices are set to zero. For design purposes, the formula (1) can be rewritten as follows

$$Y(n) = -\big(0.5 \cdot X(n) + 2 \cdot X(n-1)\big) + \big(4 \cdot X(n-2) + 0.25 \cdot X(n-3)\big) \qquad (2)$$

The final result is obtained by subtracting two expressions, where each expression consists of the sum of two terms, appropriately scaled by a positive amount. The coefficients, in magnitude, are powers of two, so the product between the coefficient and the input data is obtained basically by shifting left or right the input value, expressed in 2's complement, by a certain amount. The details are shown in Table 2.

Coefficient (magnitude)	Bit shift direction	Shift amount
0.25	Right	2 positions
0.5	Right	1 position
2	Left	1 position
4	Left	2 positions

Table 2 - Multiplication by using bit-shifting algorithm

Since the result must be computed without any intermediate approximation, the circuit requires data structure manipulation to accommodate all possible values. As the 8-bit input may be shifted either left or right by two positions, the minimum bit allocation required increases to 12 (8 of original vector, plus 2 additional bits left and 2 additional bits right). To prevent overflow conditions, an additional bit is allocated, resulting in a resized data width of 13 bits in 2's complement format, with two decimal digits.

Once the 13-bit result is computed using the formula (2), it may not be an integer. Therefore, it must first be approximated to the nearest lower or upper integer before being converted back into an 8-bit signed format for storage in the second memory. If the rounded decimal number falls outside the representable range of an 8-bit signed value, it will be clamped to the maximum possible value (01111111) if it exceeds the upper bound, and to the lowest possible

value (10000000) if it falls below the lower bound. If the result is within the 8-bit 2's complement range, it will be resized to fit within 8 bits.

Processing unit

This unit is dedicated to the arithmetic computation of the formula, with no approximations. It is composed of a shift register, four multiplication blocks, three adders and some registers.

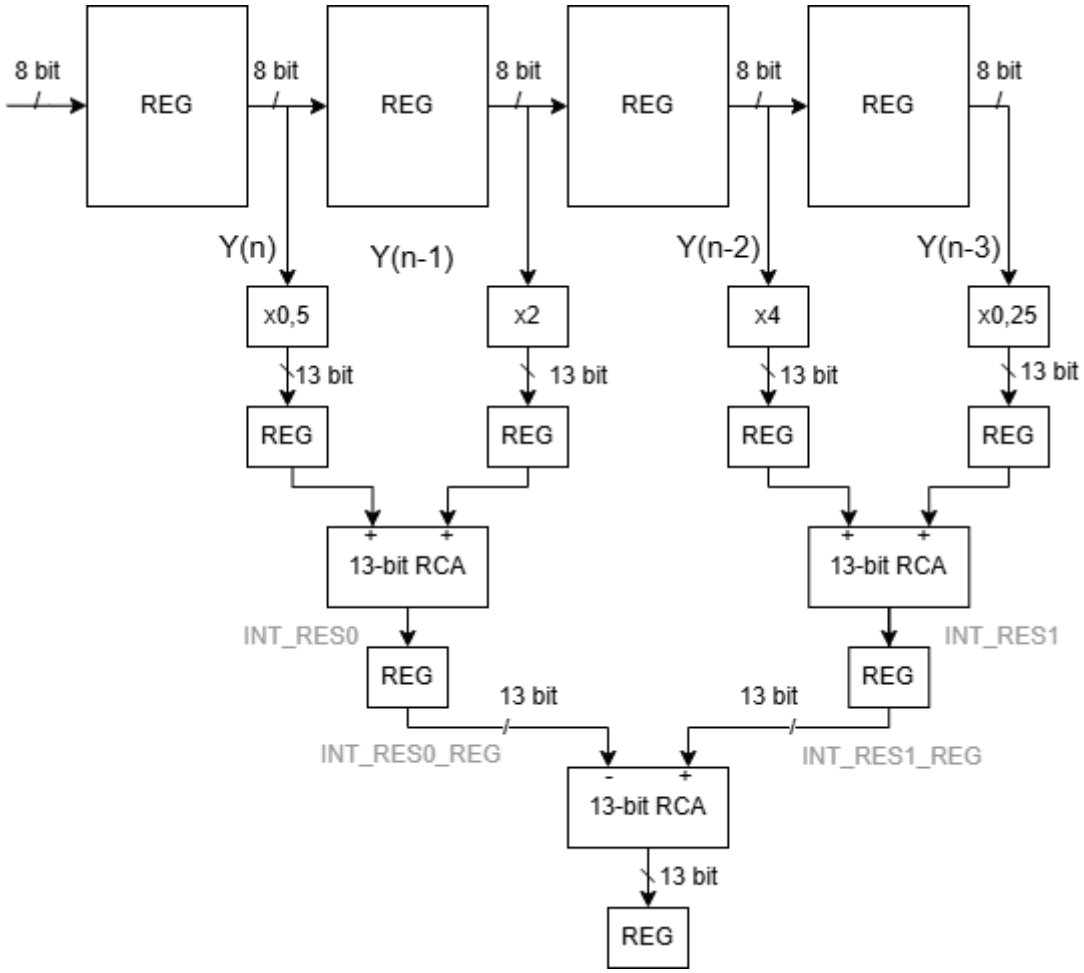

Figure 5 - Processing unit internal architecture

First, a shift register is used to store the four inputs required for computation. This shift register consists of four 8-bit registers connected in cascade, with each register's output available for processing. When not used, the shift register is always cleared by using the `SHIFTREG_EN` signal, which is provided by the control unit and it is connected to the RES input of each register (the RESET of `regn` is asynchronous, and active-low, so the register is cleared when `SHIFTREG_EN` is LOW). The outputs of these registers are then fed into four fully combinational multiplier blocks, which performs both the resize operation and the multiplication. Notice that the resize operation is required to handle values with no approximation.

The operations of the multiplier blocks are implemented using two simple concatenation statements. For example, the multiplication by 2 is achieved as follows:

```
-- Extend the input
IN1_RESIZED <= (IN1(7)&IN1(7)&IN1(7)&IN1&"00");

-- Multiply by 2 by shifting left once
Q1 <= (IN1_RESIZED(11 downto 0)&"0");
```

The other multiplier blocks use the same statements, with different indexing depending on the multiplication coefficient. The two statements may be condensed in a single one, but it will result harder to be understood, so we decided to use two different statements to separate the two steps of resizing and multiplication, while preserving the concurrent execution of VHDL.

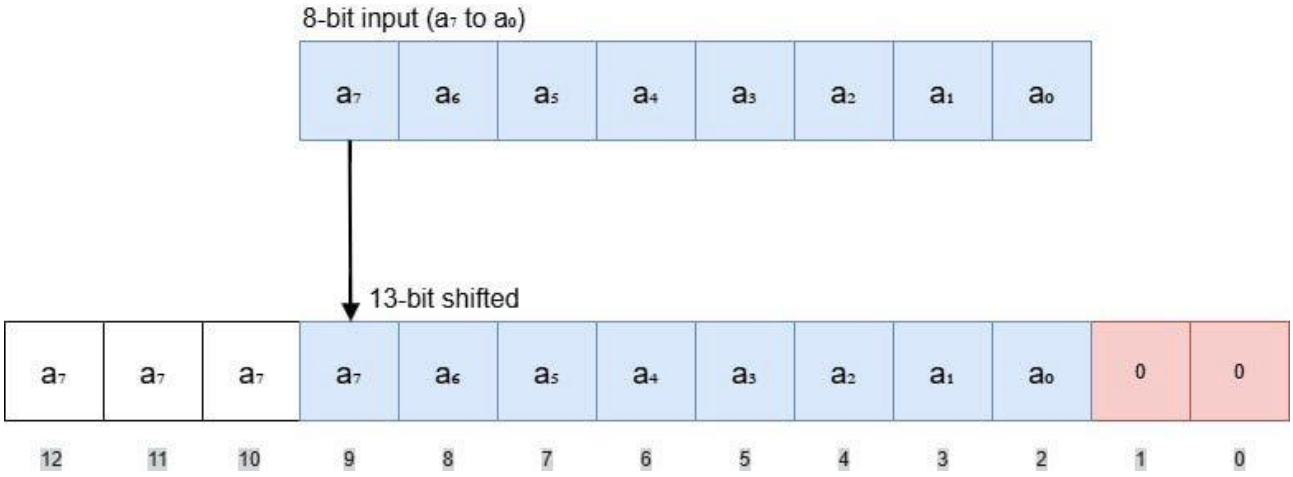

Figure 6 - Vector resize operation. In blue, the original 8-bit signed vector (integer). In red, the fractional part, set to 0 since the 8-bit vector is representing an integer signed value. In white, there are the extra three bits of the integer part.

When resizing or shifting a signed vector to the right, it is crucial to preserve the sign bit. This means that the new MSB(s) must match the original MSB.

As discussed above, the multiplier blocks perform the multiplication by the magnitude of the coefficient; the sign is adjusted in the next levels. The resized and multiplied data are stored in a 13-bit-wide register, which provides inputs to two 13-bit adders. The first adder computes

$$\left(0.5 \cdot X(n) + 2 \cdot X(n-1)\right) \tag{4}$$

while the second adder computes

$$\left(4 \cdot X(n-2) + 0.25 \cdot X(n-3)\right) \tag{5}$$

The outputs of these two adders are then stored in two 13-bit registers, which are subsequently connected to another 13-bit adder. This final adder computes the difference between the two results obtained in the previous stage and store the result in a new register.

The 13-bit adder is derived from the 16-bit ripple-carry adder used in Lab 3, with the primary difference being the number of bits, which directly determines the number of full-adders involved in the design. The processing path of the RCA is critical for the circuit timing, and for a faster implementation, the adder may be upgraded to either a Carry-Select Adder (CSEA) or a Carry-Skip Adder (CSKA).

It is important to note that, although the 13-bit vector contains both integer and fractional components, they can be treated as a single 13-bit integer vector during the summation process.

Due to the four levels of registers involved, the correct output is generated with a four-clock-cycle delay. However, this delay is not critical, as it is not a propagation delay, but rather a dead-time between the moment the first data is pushed into the first register and the moment its corresponding output is produced.

After processing the first data, all subsequent data is evaluated sequentially, ensuring a continuous flow. As a result, the entire data processing step is completed in 1027 clock cycles: 4 clock cycles of dead-time, followed by 1023 cycles for processing all remaining data. The timing details are given in the appropriate section.

Converter unit: Digital Processing and Saturation Mechanism

Once the processing unit's calculations are completed, the result is a 13-bit wide value. Bits with indices '1' and '0' are used to represent the fractional part of the processing unit's result. Since the data to be written into MEM_B must be 8 bits long, a converter is required to transform this 13-bit signed value, suitable for computation within the processing unit, into an 8-bit signed value, while respecting the saturation and approximation rules specified in the assignment. The design of this circuit initially followed a behavioural approach before transitioning to lower levels of abstraction to improve performance. This process ultimately led to the structural implementation used in the final version of the circuit, which we will introduce after first examining the simpler, initial behavioural model. The converter operates through a sequence of conversion phases:

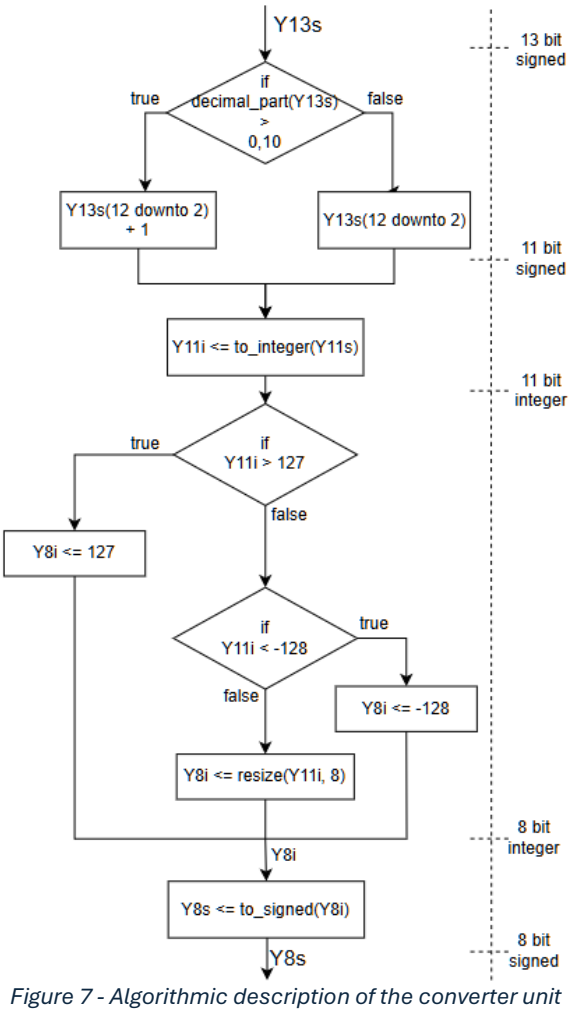

Figure 7 - Algorithmic description of the converter unit

1. The process begins by analysing the 13-bit signed output from the processing unit (`Y13s`). It checks whether the fractional part exceeds the binary value 0,10 (0,50 in decimal). This is necessary because the fractional component can only take on one of four values: 00, 01, 10, 11. Values with a fractional part of 10 or 11 are rounded up by incrementing the integer portion by 1. Following this rounding operation, the data is resized from 13 bits to 11 bits (named `Y11s`). The results remain in signed format but, at this stage, excludes the two bits originally allocated for the fractional component.

2. To facilitate subsequent comparisons and to identify saturation thresholds, the 11-bit signed value is first converted into an integer (named `Y11i`). It is then evaluated through two conditional constructs to determine whether the value exceeds the upper bound of 127 (01111111), falls below the lower bound of -128 (10000000), or lies within the valid range. Once this check is complete, the final step involves converting the integer back into a signed format, yielding the desired 8-bit signed result (named `Y8s`).

Now, after implementing this version in VHDL and verifying the correct functionality of the behavioural approach, we were not able to understand the correct timing of the circuit, so we proceeded to design a structural version that operates in the same way. The equivalent circuit, whose file is named `converterUnit`, is shown in Figure 8 below:

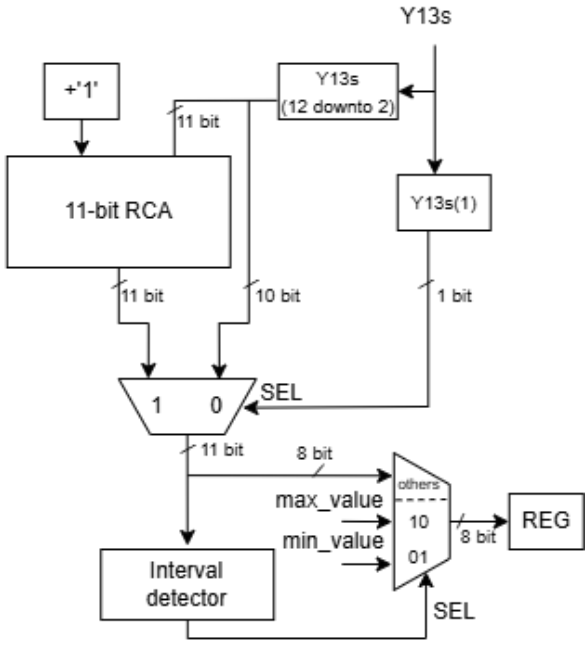

Figure 8 - Converter unit internal architecture

1. The 13-bit signed input is first decomposed into its integer part, represented by `Y13s(12 downto 2)`, which is then incremented by '1' using a Ripple Carry Adder (previously utilized in past laboratory sessions) named `RCA_11bit`. Regarding the fractional part, it is sufficient to examine only bit '1' to determine whether the rounding is necessary. This decision is made using a `mux2to1_11bit`, which selects between the integer part of the incremented value (for rounding up) and the integer part of original value (for rounding down), based on the state of bit with index '1'.
2. The next stage involves a component named `IntervalDetector`, which performs comparisons to determine whether the value falls within the allowable range, or if it exceeds either the max_value 127 (01111111) or min_value -128 (10000000) limits. The output of the `IntervalDetector` is then used as the selection signal for a `mux3x1_8bit`, which chooses whether to clamp the value to one of the two bounds or

to pass through the previously resized 8-bit value. The output of this final multiplexer is then stored in a 8-bit register (`regn`).

This structural implementation is the one used in the submitted code. To synchronize the operation of each block within the digital filter, the converter unit is also driven by an input clock signal. After verifying its correct functionality through a dedicated testbench, we confirmed that the entire conversion process completes in one clock cycle.

Memory addressing

Once the circuit enters GET_DATA mode, memory addressing occurs sequentially, with the address incrementing by one position at each clock cycle. This setup requires a modulo counter as an address driver. Since there are 1024 memory positions, the modulo counter must be configured as a 1023-modulo counter (starting from 0).

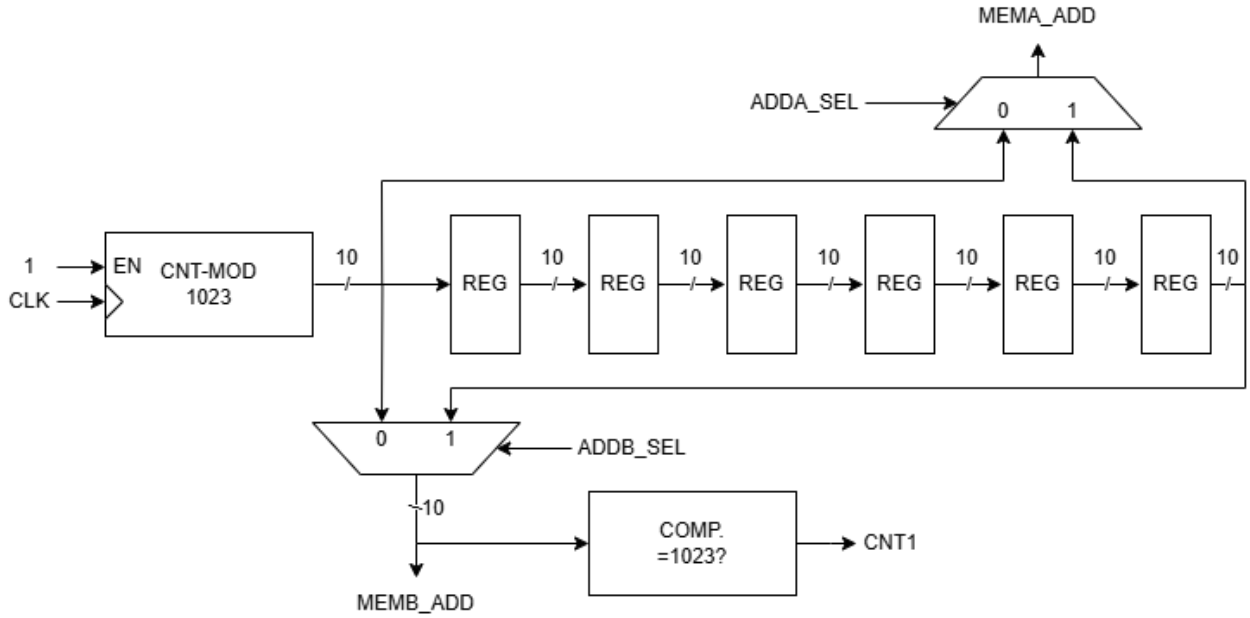

Figure 9 - Memory addressing internal architecture

The circuit first writes all data into MEM_A before proceeding to process and store results in MEM_B. During the writing phase, the counter's output directly drives the address port of MEM_A. However, during the processing and writing into MEM_B, the addresses of MEM_A and MEM_B differ due to the dead-time introduced by the computing unit. To compensate for this delay, the counter's output must be delayed by six clock cycles before driving the address port of MEM_B. This delay is achieved using six 10-bit registers connected in cascade, where the output of the comparator is fed into the first register. The address to be sent to each memory is selected by the corresponding multiplexer, driven by the control unit. Notice that for this circuit, the address of MEM_A may be connected directly to the output of the counter and the address of MEM_B may be connected to the delayed version of the counter's output, but we have preferred to add the two selection multiplexers in order to improve clearness and allow complex addressing if required in other circuits.

The state transitions are driven when the last useful address is reached. Therefore, due to the dead-time in the processing phase, the last useful address must be evaluated separately. The output of the multiplexer whose select the address of MEM_B is also used to feed the input of a comparator to check when the last address has been reached, and so when the processing is complete.

Circuit timing

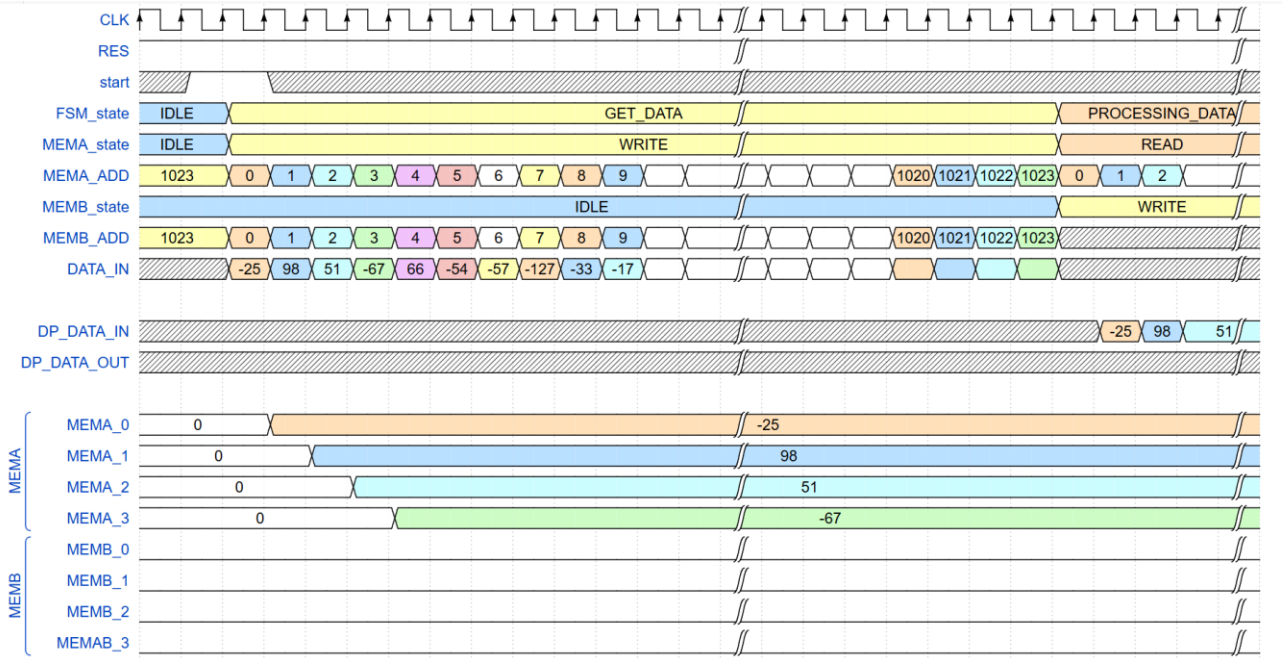

Figure 10 - Circuit timing during GET_DATA phase

The circuit timing varies depending on its current status. During the IDLE phase, pressing START triggers a transition into the GET_DATA phase, where data from the DATA_IN port must be sequentially stored into MEM_A.

In this phase, the counter is enabled and begins counting up to 1023. The counter's output is directly connected to the address port of MEM_A, which is in WRITING mode. As a result, at each clock cycle, the data present at the DATA_IN port is written into MEM_A at the address specified by the counter. This process is repeated until the output of the counter reach 1023, which triggers the CU to transition into PROCESSING_DATA phase. The entire GET_DATA phase is completed in 1024 clock cycles.

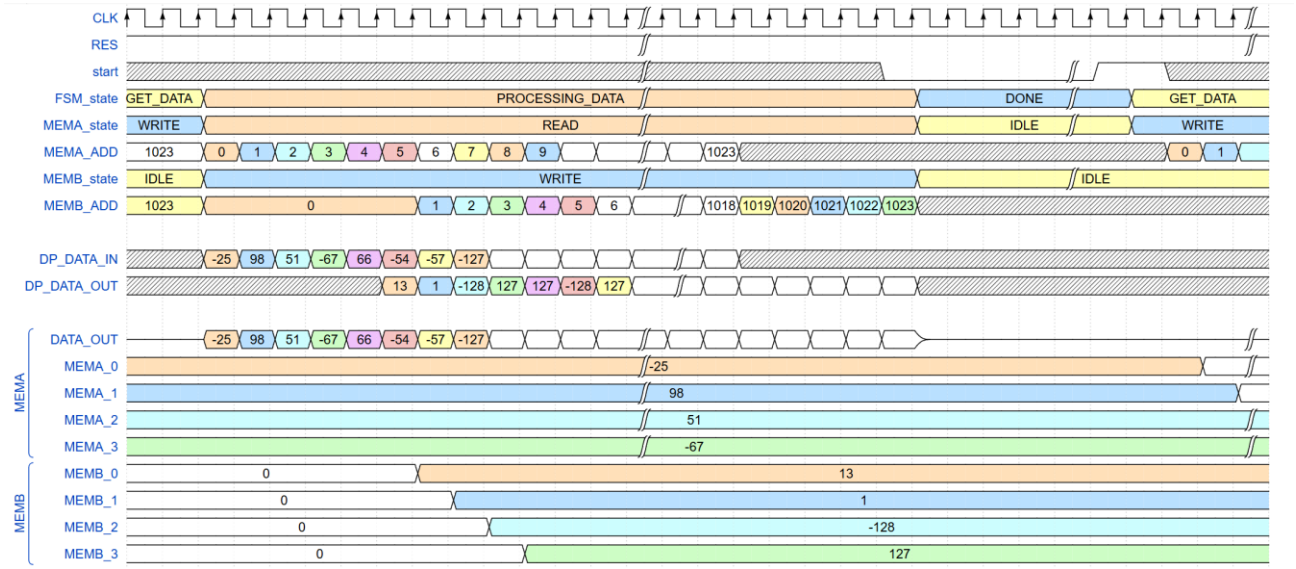

Figure 11 - Circuit timing in PROCESSING_DATA phase

During the processing phase, the counter remains enabled, and its output is directly connected to the address port of MEM_A, which is in READ mode. The output of MEM_A transitions from a high-impedance state to the stored value at the specified address. This process occurs instantaneously, as the read operation is asynchronous.

Meanwhile, MEM_B is in WRITE mode, with its address port connected to a delayed version of the counter's output. This delay is necessary to compensate for the dead-time introduced by the five register levels in the data path, ensuring that each data point is stored in the correct location in MEM_B. The overall processing operation is completed in 1029 clock cycles.

The circuit remains in the PROCESSING_DATA phase until the address of MEM_B reaches 1023, indicating that all data has been processed and stored in the second memory. Once the entire process is complete, the circuit transitions into the ENDOFTASK phase, during which the DONE signal is set HIGH. The circuit remains in this state until START is triggered HIGH, initiating the next cycle.

Functional simulation

During the design phase, for some sensitive blocks was also developed a testbench to validate the functionality of the entities. Then, a final testbench has been developed to validate the entire process.

Control unit

The testbench for the CU is very straightforward. Within a process, the input values assume different values and the CU evaluate the corresponding output based on the current state. In figure there is a fragment of the waveform results of the ModelSim simulation.

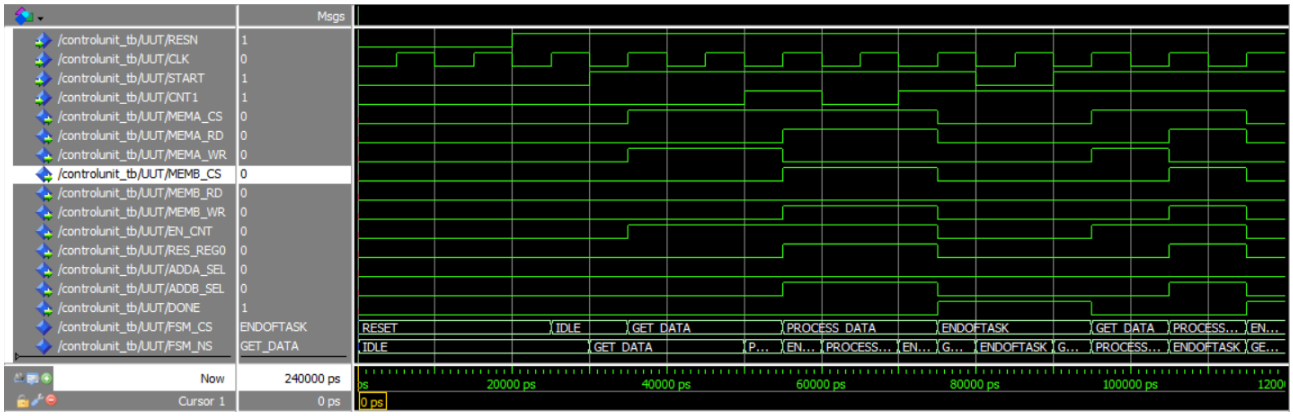

Figure 12 - Control unit simulation results in ModelSim

The simulation results match the circuit specifications, so the proper functionality of the CU has been validated.

Processing unit

The testbench for the processing unit is crucial for its validation. This circuit processes numbers in a 13-bit signed format with a fractional part, making it essential to verify the correct functionality of all components. Ensuring accuracy at this stage prevents undesired behaviour in subsequent subcircuits, where identifying the root cause of errors could be significantly more challenging.

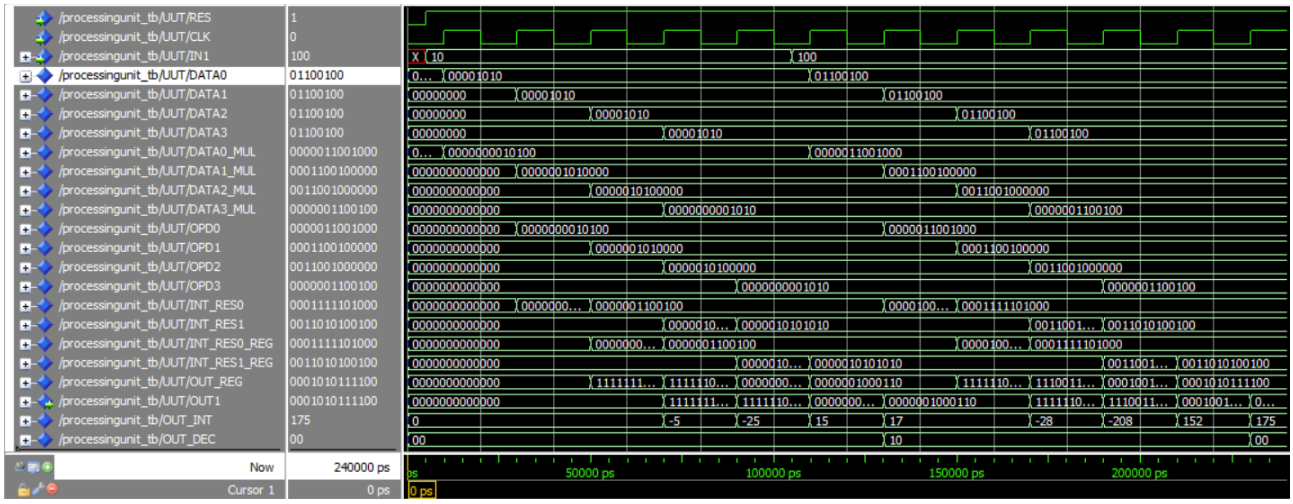

Figure 13 - Processing unit simulation results in ModelSim

The signals IN1 and the integer part of the output (named OUT_DEC) has been represented in decimal form to speed up the validation process. The inputs at DATA_IN have been chosen to be simple to allow the mental computation of each result without the need of a calculator.

The first thing that can be noticed is how the input data goes through the pipeline of registers (signals DATAx) and each multiplied and resized version (DATAx_MUL) is computed in parallel. The next clock cycle, the multiplied values are stored into the second level of registers (OPDx) and the signals are forwarded to the two RCAs, that evaluate the two sums independently and store the results in the third level of registers. In another clock cycle, the last RCA perform the subtraction between INT_RES1_REG and INT_RES0_REG and store the result in the output register in the next clock cycle. Notice that the output corresponding to the first sample is evaluated correctly with four clock cycles of delay that is not present for the next samples.

Sequence #	DATA0	DATA1	DATA2	DATA3	RESULT
1	10	0	0	0	-5
2	10	10	0	0	-25
3	10	10	10	0	15
4	10	10	10	10	17.5
…	…	…	…	…	…
5	100	10	10	10	-27.5
6	100	100	10	10	-207.5
7	100	100	100	10	152.5
8	100	100	100	100	175

Table 3 - Simulation sequences and corresponding result computed with the calculator

As this simulation has been completed without any issue, another simulation with different input data has been run to validate the process with almost random input values

Converter unit

This entity is also crucial for the overall functionality of the circuit, so another testbench has been developed to validate the circuit designed for the converter unit.

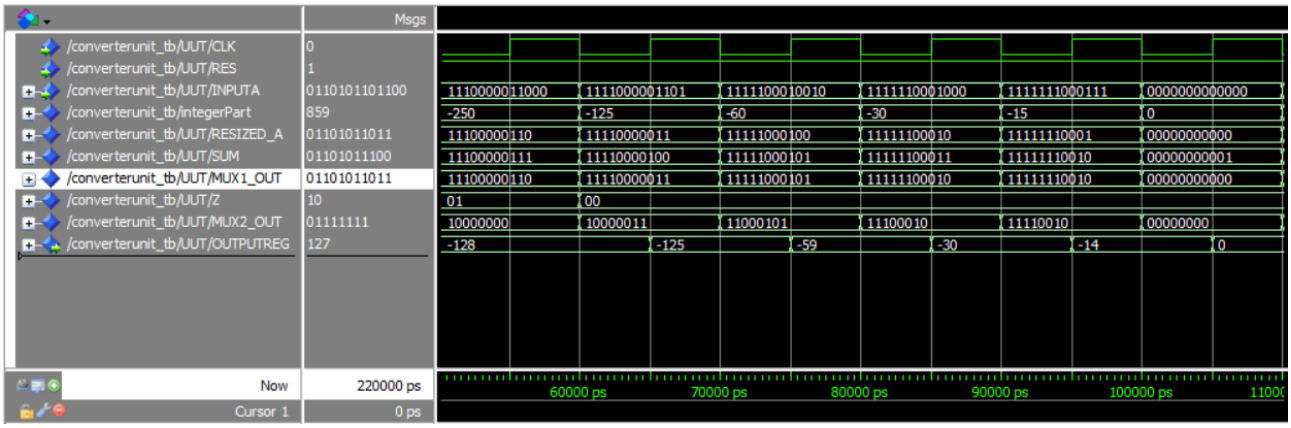

Figure 14 - Converter unit simulation results in ModelSim

In figure is represented a portion of the simulation waveform generated by ModelSim. The complete simulation result may be seen running the simulation of converterUnit_tb on

ModelSim. The integer part of INPUTA have been represented in decimal form as `integerPart`. Likewise, the output 8-bit value have been represented in decimal form to evaluate immediately if the conversion result is correct or not. Given these results, we confirm that the circuit works according to the specifications. For each input, the corresponding output is evaluated with one clock cycle of delay.

Entire data path

Once the previous components have been tested individually, it is a good practice to simulate if the entire data path behaves as expected. Notice that just the `processingUnit` and the `converterUnit` have been simulated in ModelSim, because the modulo-counter, the comparator and the multiplexers, which are also part of the data path, have already been tested in the previous laboratories.

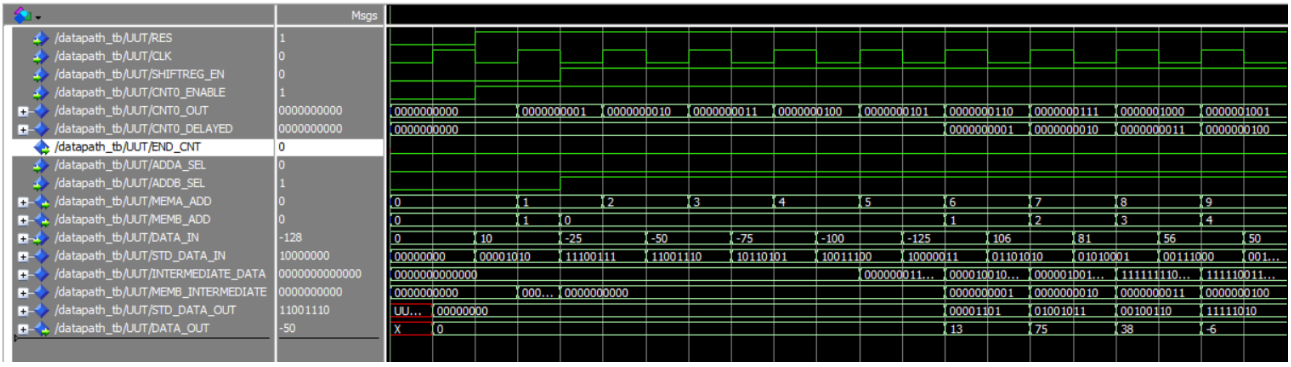

Figure 15 - Datapath simulation result in ModelSim

With this waveform result, we confirm that the circuit works correctly. Notice the timing: the output corresponding to the first input is evaluated with 5 clock cycles of delay, and then the result of the remaining stream is computed sequentially, with no dead-time between consecutive inputs. In simulation, the modulo-counter have been set as 10-modulo counter and the comparator checks if the output of the counter is equal to 9 to speed up the entire process, but in real circuit they must be set as 1024-modulo counter and the comparator to check if the output is equal to 1023.

Entire circuit simulation

Since the circuit is composed of different building blocks, it is crucial to verify if all of them works as requested, taken individually. Then, the complete circuit must be simulated to ensure the proper functionality overall.

One way to validate the functionality of the digital filter, the same functionality has been implemented in MATLAB. The script, named `digitalFilter.m`, replicates exactly the behaviour of the filter.

It begins by generating an array called MEM_A, which simulates the memory storing 1024 input samples. These elements are randomly generated, in fact, the code specifies:

```
MEM_A= randi([-128, 127], 1, 1024, 'int8');
```

Then the memory B was initialized before the operation of filtering.

To simulate the behavior of the hardware filter precisely, the code handles the first three output samples (Y(0), Y(1), and Y(2)) separately, because some of the past samples (X(-1), X(-2), etc.) do not exist at the first iterations and are assumed to be zero. The remaining output samples are computed in a loop for indices 3 to 1023. The operation is straight allocated in MEM_B.

Although the input samples are 8-bit integers, the calculations are performed using floating-point operations (double) to avoid rounding errors during the weighted summation. The results are then cast and saturated back to 8-bit signed integers for compatibility with hardware constraints. It is important to notice that the code applies saturation to keep the result in the range from −128 to127 (8-bit, 2's complement format).

Running the script generates the random vector MEM_A and the corresponding filtered stream, named MEM_B. One execution has generated the values that have been used as stimuli in ModelSim. The values are collected in the spreadsheet file named *digitalFilter_datastream.xlsx*. A portion of the input signal sequence and the corresponding filtered signals obtained in MATLAB is presented in Table 4.

Sample Index	Input (MEM_A)	Output (MEM_B)
1	-25	13
2	98	1
3	51	-128
4	-67	127
5	66	127
6	-54	-128
7	-57	127
8	-127	-22
9	-33	29
10	-17	-128

Table 4 - Example of input stream and the corresponding output filtered with MATLAB

The same portion of input stream has been applied to the digital filter described in VHDL to perform a simulation in ModelSim and then compare the two results. The waveform result of the simulation in GET_DATA status is shown in figure.

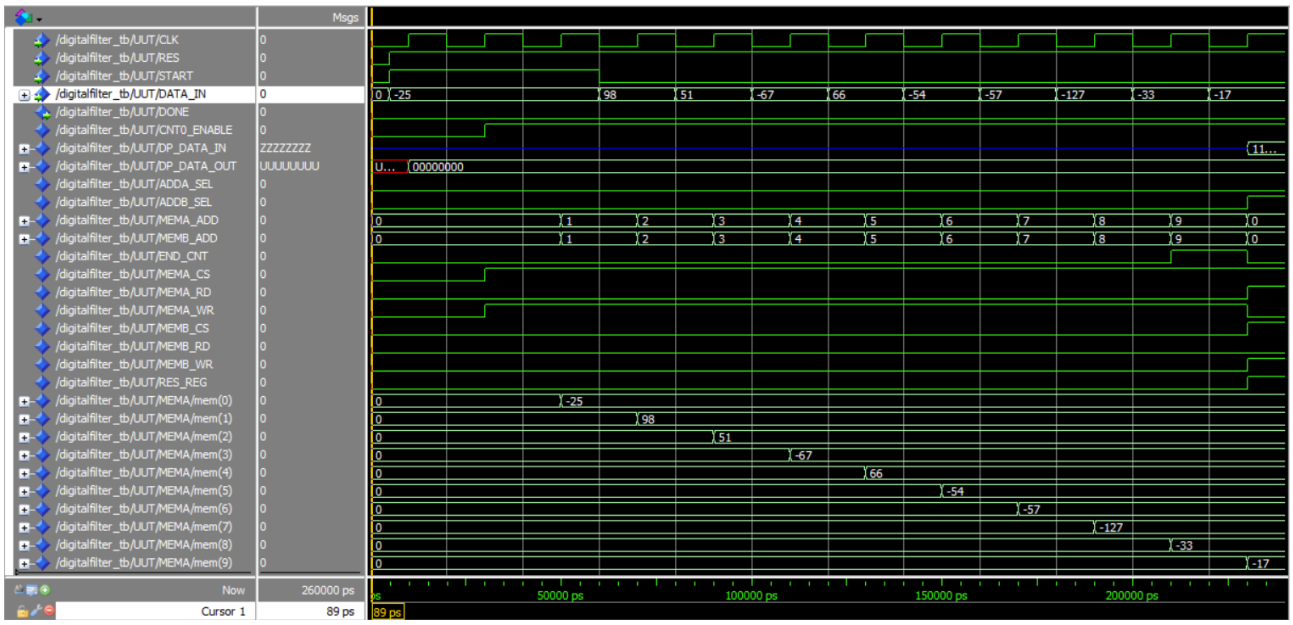

Figure 16 - Digital filter simulation results in ModelSim from the RESET status to IDLE and then to GET_DATA phase.

The circuit writes the data present at DATA_IN port in the memory location specified by MEMA_ADD, according to the specifications. After the 10 samples have been stored in the first memory, the counter has reached the boundary of the memory space allocated, so the CU transitions into PROCESSING_DATA status. Remember that the boundary address has been set to 9 for simulation purposes, to speed up the entire process of validation. The waveform result of the circuit in PROCESSING_DATA phase is shown in figure below.

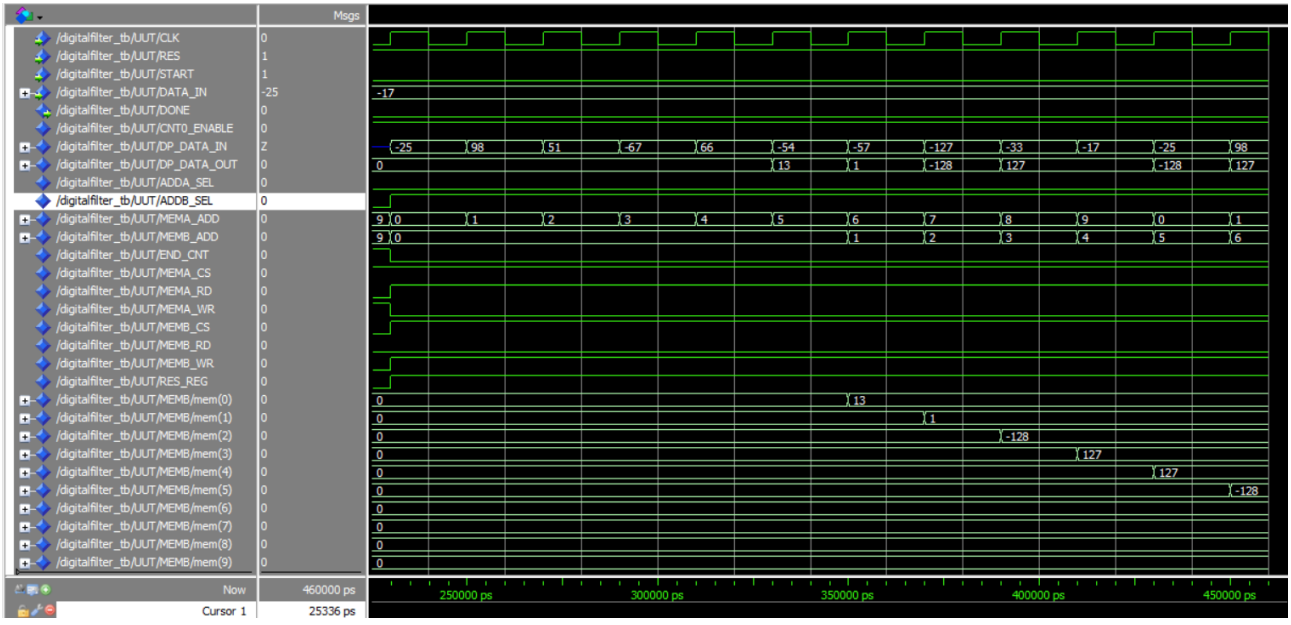

Figure 17 - Digital filter simulation results in ModelSim in PROCESSING_DATA phase.

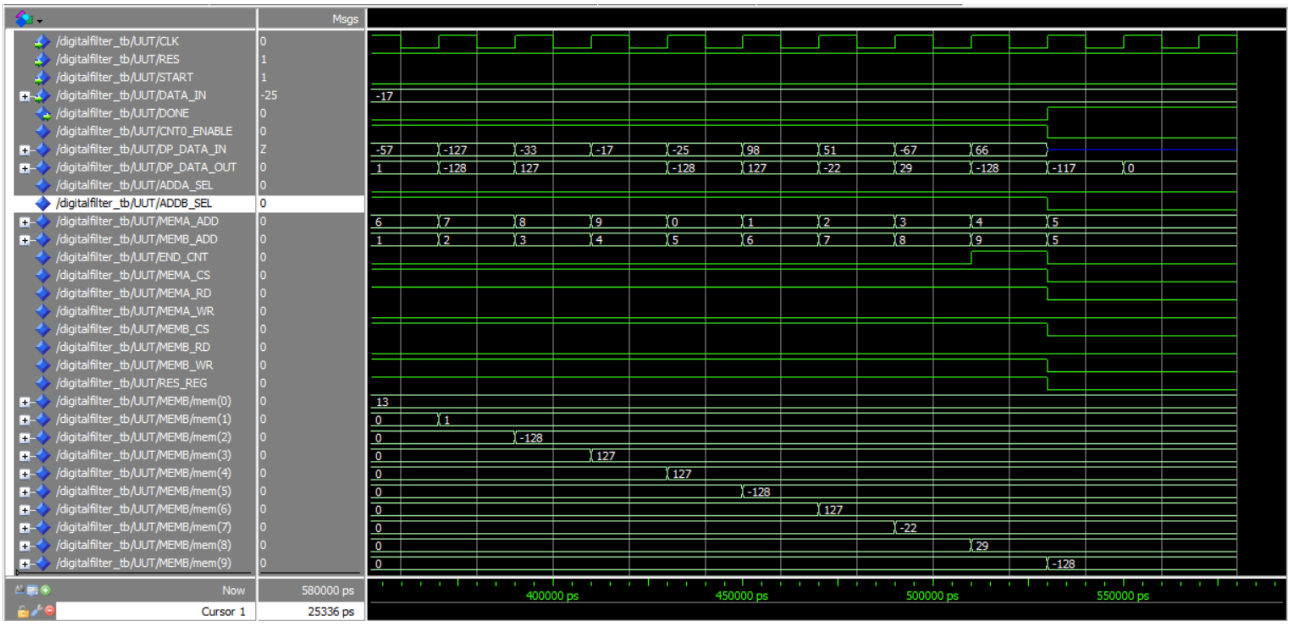

Figure 18 - Digital filter simulation results in ModelSim from the second part of PROCESSING_DATA phase to ENDOFTASK phase

The second memory is filled with a delay of six clock cycles relative to the applied input stream. The filtered version of the input stream aligns with the MATLAB script results, confirming that the entire circuit operates according to specifications. Additionally, the simulation results adhere to the timing requirements outlined in the relevant section above.

Other considerations

Once the overall circuit has been validated, it is ready to be used as a part of a bigger circuit implemented on a FPGA. An important parameter to know is the maximum operating frequency of the device, that can be computed by Quartus. In case of the FPGA used in the other laboratory experiences, the ALTERA Cyclone V 5CSEMA5F31C6N, the maximum operating frequency is approximately 317 MHz when the FSM is using the one-hot encoding, which is the faster than the minimal-bits encoding. The design constraint file used for the timing analysis is named *Lab6.sdc*.

Conclusions

This lab report presented the design and implementation of a digital filter using memory-based processing. The circuit successfully stores input samples, applies a filtering equation using only addition and shift operations, and ensures proper saturation for 8-bit representation. The multiplierless data path helped optimize hardware resources while maintaining accuracy and accelerating the entire process. Simulation results confirmed that the filtered output matched expected values from MATLAB, validating the circuit's correctness.

A critical aspect of ensuring the success of this implementation was the simulation of each individual block before integrating them into the final system. By thoroughly testing and verifying the behaviour of smaller components – such as memory operations, arithmetic computations, and control signals – the development process became more efficient and less prone to unexpected errors. This approach also allowed for early detection of potential design flaws, reducing debugging time and minimizing costly revisions in later stages.

Furthermore, a well-structured design organization played a key role in achieving an efficient development process. Clearly defining module interfaces, maintaining consistency in data handling, and following a hierarchical approach ensured that each subsystem contributed seamlessly to the overall functionality. By combining detailed simulation and organized development practices, the circuit was successfully implemented with optimized performance, reliability, and adherence to specifications.

Sources and notes

For all the VHDL files provided, an online tool has been used to format the code to ensure coherence between all the different files. The tool is free to use, and it is available at this link: VHDL Beautifier.

For the block diagrams, the online tool used in this document can be reached at this link: draw.io.

The waveform in figures 10 and 11 have been created with Wavedrom, which can be reached at this link: Wavedrom.

The VHDL files were developed using course materials produced by Prof. G. Masera (available on the course webpage) and the following eBooks recommended by the professor:

1. Mealy, Bryan, and Fabrizio Tappero. *Free Range VHDL*. 2016.
2. Ashenden, Peter J. *The VHDL Cookbook*. 1990.