# Report 10

Filippo Nevi

July 2021

## 1 XML definitions

The first step to implement the multiplication algorithm was the creation of its XML file and the modification of the XML of the controller.

In order to adapt the already existing definition of the controller, I added two boolean variables: **Input_ready**, which signals to the multiplier that it has to start the computation, and **Result_ready**, which signals to the controller that the result of the multiplication is ready to be read. Eight 8-bit variables were to be added: four represent the input numbers, and the other four are used to receive the result of the operation.

The description of the new module has its own *model name*, *GUID* and *model identifier*. It has eight 8-bit variables, that are equivalent to the ones defined in the controller, and three boolean ones, that represent the **Reset**, **Input_ready** and **Output_ready** signals.

## 2 C implementation

### 2.1 Controller

In order to link the multiplier, I had to edit the number of signals for each type that are defined in the file `controller.h` and add the declarations of the signals in the struct *ModelInstance* in the same file. Then, these signals had to be defined in the **controller_implementation** function. Finally, I implemented some utility functions:

- `return_fixed_threshold`: this function converts the threshold from double (64 bits floating point) to `uint16_t`. This procedure will lose some data, but it has to be done because each operand of the multiplication is passed as two 8-bit variables;
- `fixed_32_to_floating`: returns a double number, starting from four 8-bit variables (in this case, the result of the multiplication);
- `integer_part_mask`: returns the most significant byte of a 16-bit variable;
- `fractional_part_mask`: returns the least significant byte of a 16-bit variable.

## 2.2 Multiplicator

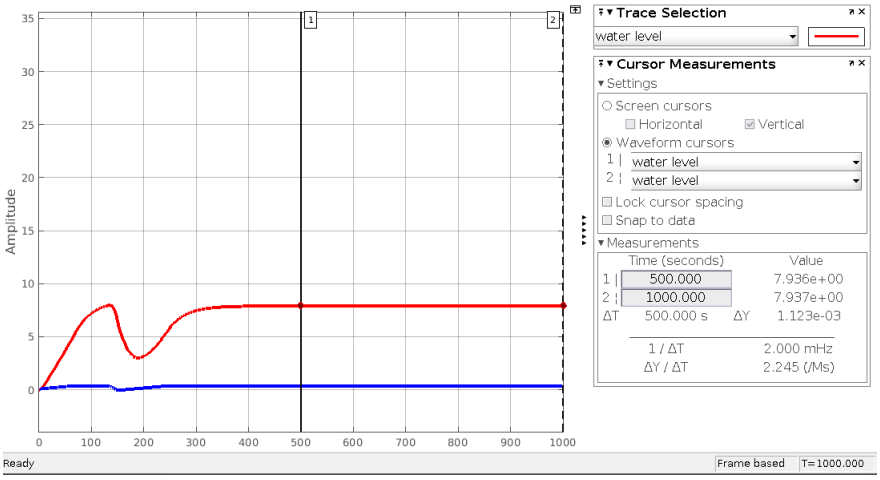
The Multiplicator system implements the same EFSM that was defined in the second assignment of this course. For maintaining the intermediate computation values, I added some internal variables that are not transmitted to the controller: **product**, **number_a**, **number_b** and **counter** (which are initialized in the function `setStartValues`). Then, I defined the utility functions:

- `return_32_bit_number`: this function assembles four bytes and returns the equivalent 32-bit variable;
- `return_16_bit_number`: this function assembles two bytes and returns the equivalent 16-bit variable;
- `split_product`: splits the product into four bytes and assigns the values to the corresponding output signals.

# 3 Simulink

In the Simulink model, I linked the controller to the multiplicator, adding two delays to the **Input_ready** and **Output_ready** signals, so the systems will have time to receive the data before the flags are set as `true`.

The simulation waveforms are the following:




The water level is the red wave: it is slightly different from the original system that used the `*` operator instead of a separate FMU, but once it gets stabilized, the levels of the two solutions are almost the same. The small difference between them is due to the loss of accuracy when converting the threshold from `double` to `uint16_t` in the controller system.