Laboratory #6:
Algoritmic State Machine Chart

Professor Guido Masera

Group 1:

Akouissi Outman	201061
Collura Matteo	201692
D'Ascenzi Marcello	200982
Lanieri Valerio	201045

Abstract

The purpose of this laboratory session is to build a digital filter capable of storing into a memory a previously processed data flow.

Contents

1 Design description

For this laboratory we designed a digital filter made of the following components:

Data Path Control Unit Two memory elements

To better read and understand the functioning, we list below the main signals and their usage.

1.1 Main signals description

– *START*	Few cycles long, it is interpreted as if it is coming from a pushbutton
– *HARD_RESET*	External asynchronous reset that forces the control unit to enter **IDLE** state
– *CL*	When driven high, clears all the elements inside the data path
– *COUNTER_ENABLE*	Enables the counter which increments the memory address pointer
– *CS_A*, *CS_B*	Targets (selects) *MEM_A*, *MEM_B*
– *WR_A*, *WR_B*	Enables writing mode for *MEM_A*, *MEM_B*
– *RD_A*, *RD_B*	Enables reading mode for *MEM_A*, *MEM_B*
– *Q_ENABLE*	Enables registers Q0, Q1, Q2, Q3
– *Q_ADD_SELECT*	2-Bit select signal for the multiplexers *QMUX* and *ADDMUX*
– *SUB*	Defines the operating mode of the adder (addition or subtraction)
– *SR_CTRL*	3-bit signal which controls the Shift Register as described below:

‘000’ Hold (preserves previous value)	‘010’ Load data from input
‘100’ Shift left by one position	‘101’ Shift left by two positions
‘110’ Shift right by one position	‘111’ Shift right by two positions

1.2 Data Path

We divided our data path description into various blocks, since this way it is easier to describe each section according to its function and the related signals.

- **"*Q*" registers**: Four cascaded parallel-in/parallel-out 8-bit registers. The first one stores values from *MEM_A*, while the others acquire the value of the previous register without any modifications. We need those bits unchanged, since we have to use them for more than one cycle of the DSP algorithm. Only the core of our data path will handle mathematical operations. All the registers of our circuit need a load signal in order to be synchronized with the control unit. The "*Q*" registers have a dedicated control signal (*Q_ENABLE*) that enables (or disables) all of them at the same time.
- ***QMUX***: This multiplexer selects one out of the four values stored in the "*Q*" registers and then sources it to the Shift Register below. "00" corresponds to *Q(0)*, "01" to *Q(1)*, and so on.
- **Shift Register**: This component reads one value at a time, sourced by *QMUX*.The input of this element is turned 11-bit long (three more bits with respect to the original). These "guard bits" allow our circuit to safely handle multiplications and additions, since we want to avoid to incur into overflows (and thus, in significant losses of data) halfway through the computation. Overflow errors will only be considered at the end of the pipeline, as the computed value is converted back from 11 into 8 bits.
  Moreover, the Shift Register receives a 3-bit control signal *SR_CTRL* to manage which operation it should perform. Internally, all the four possible operations of shifting (right or left, one or two positions) are performed asynchronously and then input to a multiplexer, driven by 2 bits of *SR_CTRL* (the other control bit is needed as a load enable). The output is stored in a register element that updates with the clock, thus making the whole component work in a synchronous way.
- **Adder**: The adder can handle both additions and subtractions by means of the *SUB* control, which negates the second input and adds a '1' carry-in (2's complement inversion). The second operand input, as already said, comes from the output of the Shift Register; the first input instead comes from *ADDMUX*. The output of this adder is stored inside the Result Register.
- **Result Register**: It receives as input the output of the adder. Then it is enabled only in a few states thanks to combinational logic on *SR_CTRL* bits, to avoid logging temporary results.
- ***ADDMUX***: This multiplexer chooses between the output of the Result Register and a null vector. The output is connected to the first operand input of the adder.
  The vector of zeros is selected only during the first subtraction operation, namely when the Shift Register is reading from *Q(1)* and *Q_ADD_SELECT* is set to "01". When the select signal has any other value, this multiplexer is going to loop the content of the Result Register back to the adder's first operand input.
- **Overflow control**: The occurrence of an overflow is checked only at the end of our path thanks to the presence of the guard bits. In order to ascertain whether our final result can be directly converted back to 8 bits, our circuit needs to check if the first four digits (MSBs) are equal. To this purpose we use a 4-bit AND in parallel with a 4-bit NOR to see if they all are equal to respectively '1' or '0'. If one of these conditions is verified, the number will remain unchanged in two's complement and we can therefore erase three MSBs from the output of the adder, directly storing it in Memory B.
- **Overflow management**: If the above mentioned conditions are not satisfied, we incur into an overflow condition, hence we need to use another multiplexer to discriminate the two possible cases. We exploit the value of the most significant bit to choose the saturated result. If the MSB is '1', the result is negative and clipped, so we output "10000000" (-128). Otherwise, if we have '0' as the MSB, we output its inverse, "01111111" (+127).

1.3 Control Unit

The control unit is the core of the circuit, a finite state machine relying on eighteen different states. Below there is a description for all of them. This machine is of Moore type, since the future state depends on the present state only.

- **IDLE**: The control unit waits for the *START* signal to be high. The *CL* signal is high, hence all the elements in the Data Path are reset (i.e. registers are emptied, counter is set to 0, etc.).
- **STORE**: *CS_A* and *WR_A* are driven high so that *MEM_A* enters into writing mode. Now the *COUNTER_ENABLE* flag is driven high so that the address pointer is incremented at the end of every *STORE* cycle. Then, the incoming data is stored inside *MEM_A* in a sequential way. When the counter reaches the maximum value, the address is automatically reset by the data path and the machine enters into the next state.
- **ACQUIRE**: *CS_A* and *RD_A* are driven high, thus *MEM_A* enters into reading mode. *Q_ENABLE* is driven high so that the first of the four 8-bits registers in the Data Path acquires the data coming from *MEM_A* and the others read from their preceding ones. If *MEM_B* is finally filled, the next state will be **COMPLETED**, otherwise we proceed to the following state.
- **ALG0**: The algorithmic part of the filter starts to operate. *SR_CTRL* is set to “010” so that the Shift Register acquires data from *QMUX*. In this case, it receives the data coming from the first of the four “*Q*” registers, since *MUX_IN* is controlled by a “00” signal as initialized from the beginning.
- **ALG1**: *SR_CTRL* is set to “110” so that the Shift Register performs a right shift.
- **ALG2**: Now we have the first operand ready and flowing inside the adder. *SUB* is set to ‘1’ in order to make the adder perform a subtraction. Since the adder is performing its first operation, we set its first input to “00000000000” through the multiplexer *ADDMUX*. The first operation ($-0.5X(0)$) is then performed. In the future states the multiplexer will send to the first operand of the adder the temporary result. The adder’s result is going to be stored in the Result Register at the next clock rising edge. *SR_CTRL* becomes “010” but this time *Q_ADD_SELECT* is set to “01”, thus the Shift Register can be updated with the content of the second “*Q*” register.
- **ALG3**: *SR_CTRL* is set to “100” so that the Shift Register performs a left shift on its content.
- **ALG4**: Similarly to **ALG2**, *SUB* is set to ‘1’ so that the second subtraction ($-2X(1)$) can be performed. *Q_ADD_SELECT* is set to “10” and *SR_CTRL* is set to “010”, which translates into the acquisition of the third “Q” register’s content by the Shift Register.
- **ALG5**: *SR_CTRL* is set to “101”, which corresponds to a double left shift.
- **ALG6**: *SUB* is ‘0’ so the third operation ($+4X(n-2)$) can be executed. *SR_CTRL* is “010” and *Q_ADD_SELECT* is “11”, thus the last register’s output enters the Shift Register.
- **ALG7**: *SR_CTRL* is set to “111” to perform a double right shift on its content.
- **ALG8**: *SUB* is still ‘0’ in order to perform the last operation ($+0.25X(n-3)$). *SR_CTRL* is set to “010”, although we don’t care about the value which will be stored in the Shift Register.
- **SOURCE**: *CS_B* and *WR_B* are driven high so that *MEM_B* accepts the output of the Data Path. *COUNTER_ENABLE* is driven high so that the counter increments the position of the address pointer before moving back to the **ACQUIRE** state.
- **COMPLETED**: As previously mentioned, when the address pointer reaches the last cell we enter into this status in which the signal *DONE* is driven high. The control unit will then go back to **IDLE**, waiting until *START* is driven again high.

1.4 Memories

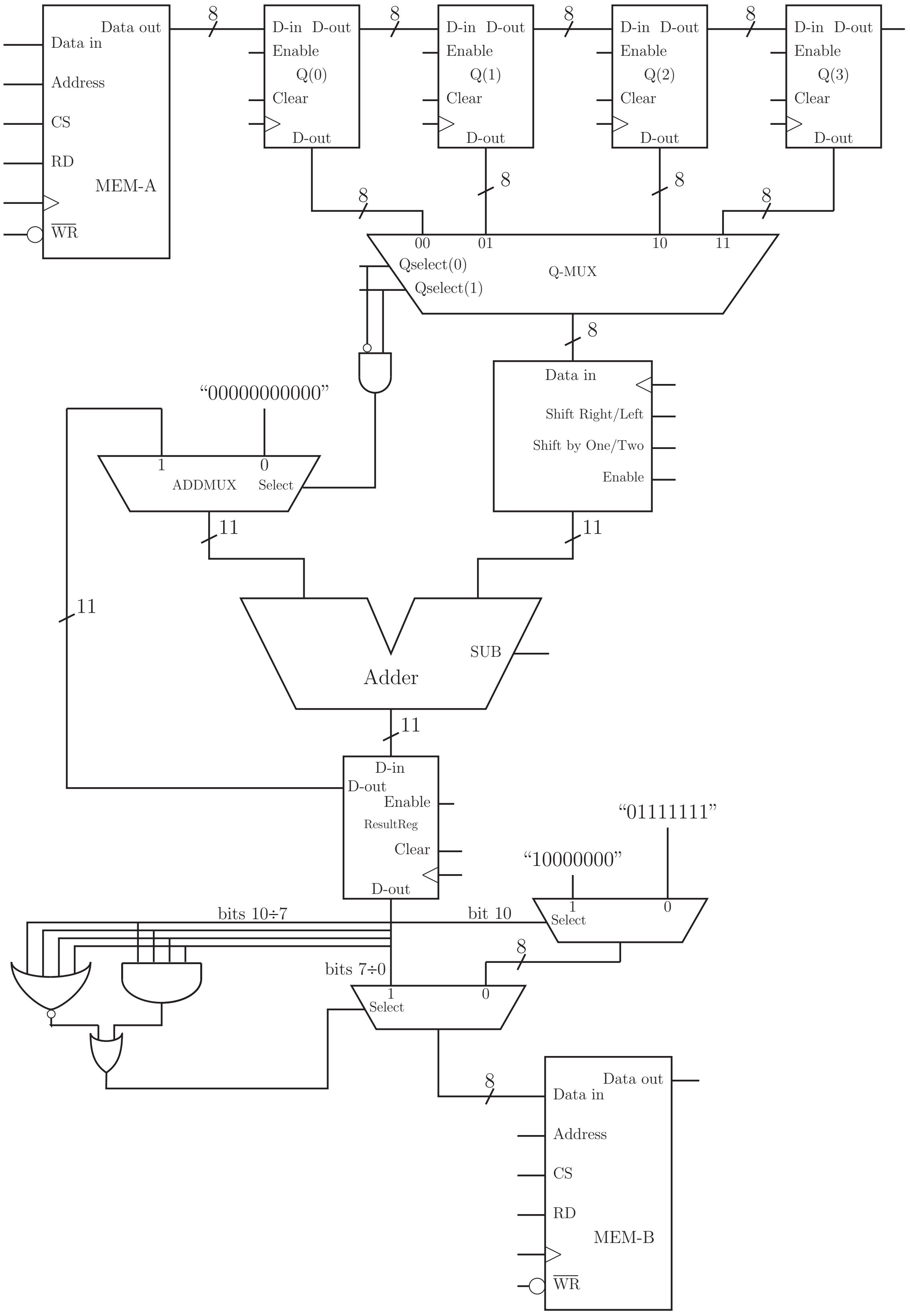
Memory A and memory B share the same design and follow the description given by the assignment: memory can be written only synchronously with the clock's rising edge, whereas the output is sourced in a combinational way. Memory locations are declared using an array, which greatly simplified the code, both from a writing and a reading perspective.

2 Schematics

2.1 Pseudocode - Testing algorithm

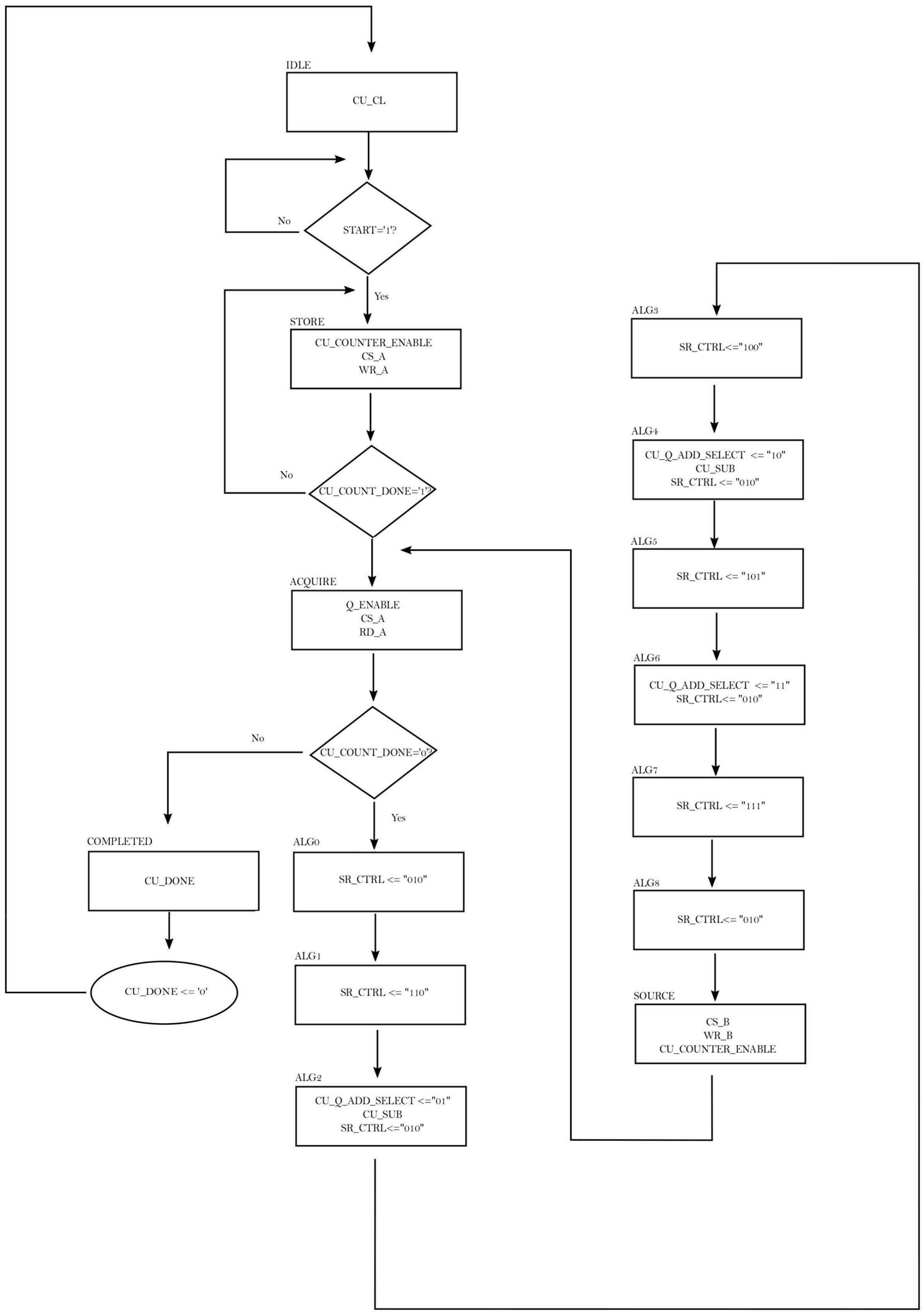
```
#define max 1023

void filter(void){

if(start == 1){
  done    = 0;
  counter = 0;
      while(counter < max){
          cs_a = 1;
          wr_a = 1;
          mem_a(counter) = data_in;
          counter++;
      }
  counter = 0;
      while(counter < max){
          reg3 = reg2;
          reg2 = reg1;
          reg1 = reg0;
          cs_a = 1;
          rd_a = 1;
          reg0 = mem_a(counter);
          cs_a = 0;
          rd_a = 0;

          output = -0.5*reg0-2*reg1+4*reg2+0.25*reg3;
            if(output > 127){
                output = 127;
            }else if(output < -128){
                output = -128;
            }

          cs_b = 1;
          wr_b = 1;
          mem_b(counter) = output;
          output = 0;
          cs_b   = 0;
          wr_b   = 0;
          counter++;
    }
  done = 1;
}
```

2.2 Datapath



Data out
Data in
Address
CS
RD
MEM-A
WR
8
D-in D-out
Enable
Q(0)
Clear
D-out
Q(1)
Q(2)
Q(3)
00
01
10
11
Qselect(0)
Qselect(1)
Q-MUX
Data in
Shift Right/Left
Shift by One/Two
Enable
"00000000000"
1
0
ADDMUX
Select
11
SUB
Adder
D-in
D-out
Enable
ResultReg
Clear
"01111111"
"10000000"
bits 10÷7
bit 10
bits 7÷0
Select
MEM-B

2.3 ASM chart and Control Unit



3 Design Validation

We decided to simplify the Control Unit as much as possible in terms of signals management: we used the same signal to issue multiple, contextual commands to the Data Path (e.g.: *Q_ADD_SELECT*, which both selects the input to the first operand of the adder and also decides which "Q" register is being input to the Shift Register). Also, there was no need to use multiple clear signals, so we used only one.

We exploited generic declarations in order to quickly reuse previously built components and adapt them to our needs, based on the amount of necessary bits (registers). Moreover we implemented multiplexers like *ADDMUX* and *QMUX* using a *with-select* statement.

We also performed several design choices which would simplify the whole project since we were given no limits on the amount of resources to be used, nor the type of elements to be used. This led us to implement the three additional guard bits inside the Data Path to make the result free of approximation errors: in fact this allows the compensation of positive or negative overflows during the process of summation. Analogously, that is the reason why we added the rather uncommon possibility of performing a double left or right shift inside the Shift Register. In fact, this element is not properly a shift register but a simple register which takes as input the result of a combinational logic block, working asynchronously. The only true limitation in the whole assignment was on the usage of a single adder unit.

When designing the ASM chart we decided to privilege clarity and linearity, still maintaining a fairly fast and reliable algorithm. This allows the reader to clearly relate a specific operation to its relative signals and Data Path elements. We have also tried to condensate multiple operations within a single state when possible. For instance, during **ALG4** the second operation is performed while the third register's content is transferred to the Shift Register. This pipelining structure, in addition to the double shift registers, allowed us to skip some cycles in the algorithmic part of our sequence, resulting in a noticeable speed improvement.

For some of the simplest blocks of the Data Path, as for the multiplexer, we did not use a whole discrete component, but only few lines of codes, in order to allow the compiler more freedom in the synthesis process and, thus, to reduce the number of gates needed for the final circuit realization.

3.1 Expected timing

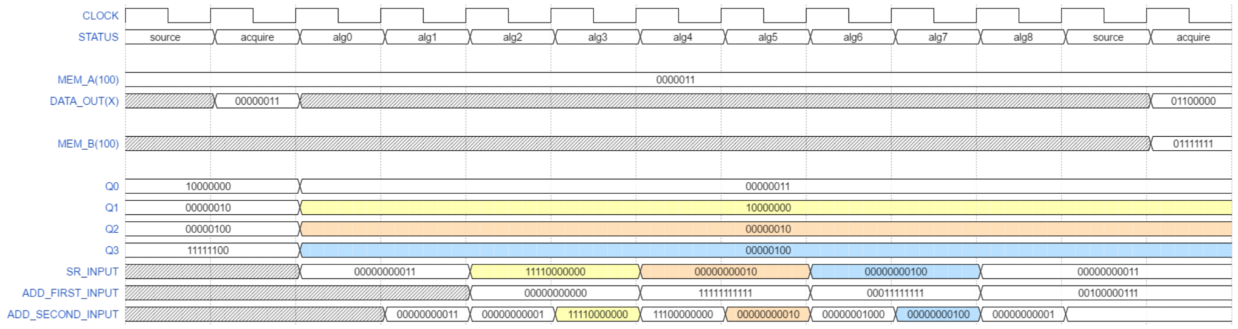


Figure 1: Expected timing considering data evolution

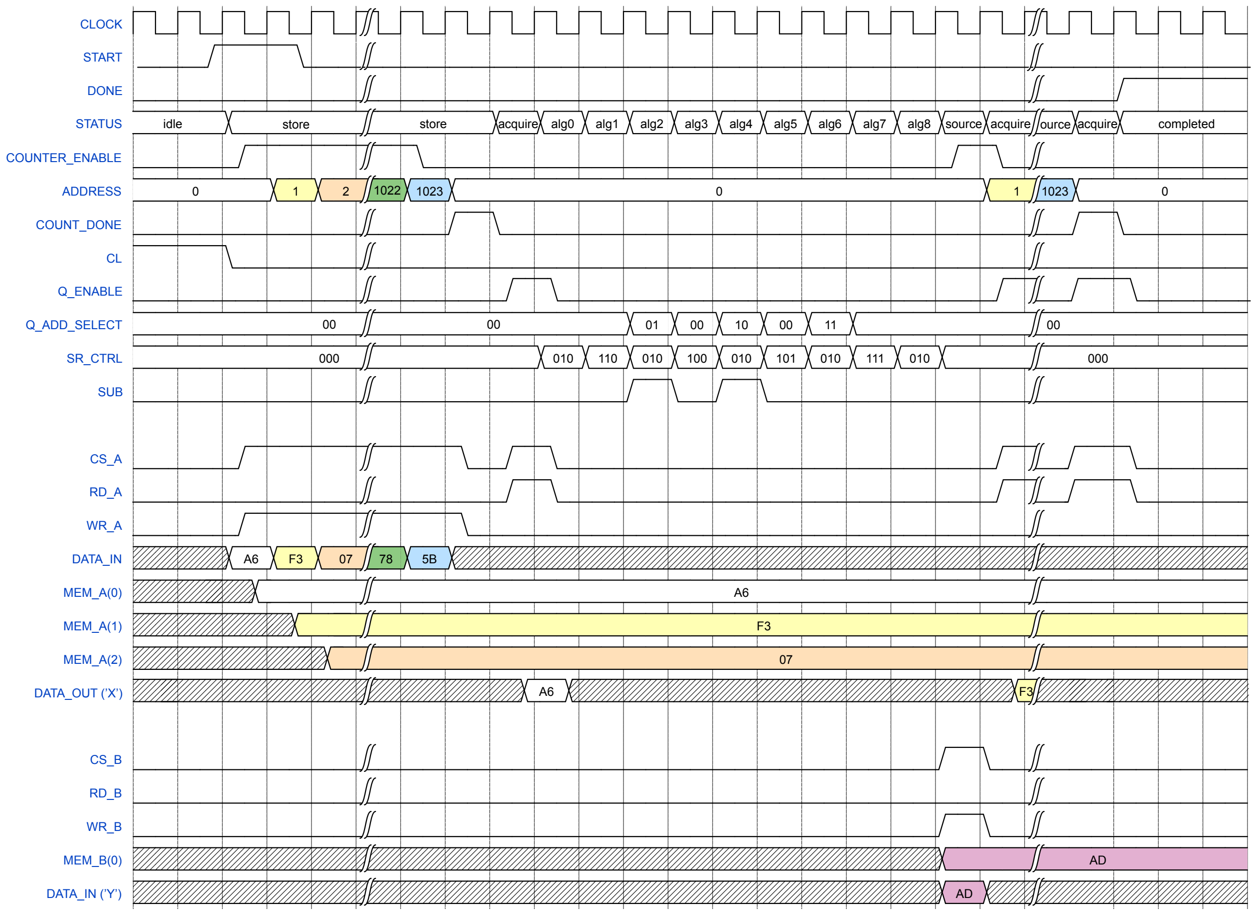


Figure 2: Expected timing in the storing processes

4 Simulation fragments

To check our design we created a testbench and started our simulations on Modelsim with 8-bit memories. We had to test at least four values in a row, since only after three steps all the “Q” registers would be filled with data and the algorithm could be checked in in its entirety. Anyway, we also checked that the first values obtained for the output vector were the result of a sum with zero-initialized elements. To ease the simulation process we initially inputed 5 data instead of 1024. After those few checks, to test both result consistency and overflow handling, we decided to generate an input vector of 1024 random elements with a C code file. By means of the same file we also generated the corresponding expected results. Then we checked on a spreadsheet if they were equal to the ones stored in *MEM_B* by our circuit.

We had to adapt our input signals timing to emulate a realistic operation, first by informing our circuit when to start acquiring values, and only then starting to feed the required data. First of all, the *START* signal is raised to point out the incoming data flux. We do not strictly need any acknowledgment response from the Control Unit, since its only duty is to manage the DSP algorithm. Hence, in this particular case, it will directly start to acquire the incoming samples from the very next clock cycle, entering the **STORE** state (Figure 3). Hence it raises *CS_A*, *WR_A* and *COUNT*, so that *MEM_A* starts being filled up with elements coming from the *DATA_IN* channel.

As soon as the last address (1023) cell is written, the Control Unit enters into the **ACQUIRE** state (Figure 4), in which it acquires the first datum from *MEM_A*. This is followed by performing the various subtraction, addition, multiplication and division operations through the Adder and the Shift Register. These elements are properly controlled during each of the following states (namely, **ALGx**, x being a number from 0 to 8) by the Control Unit, in compliance with our previous description.

In Figure 5 it is shown how the Control Unit exits the algorithmic part when the address 1023 is reached. We see the last **SOURCE** state, in which *CS_B* and *WR_B* are high so that the last computed value can be stored inside the last position of *MEM_B*. Then the completion of the algorithm is assessed during the following **ACQUIRE** state. Finally, we move to the **COMPLETED** state, in which *DONE* rises for one clock cycle. Then we move back to the **IDLE** state, waiting for the *START* signal to rise once more, signaling the beginning of a further filtering process.

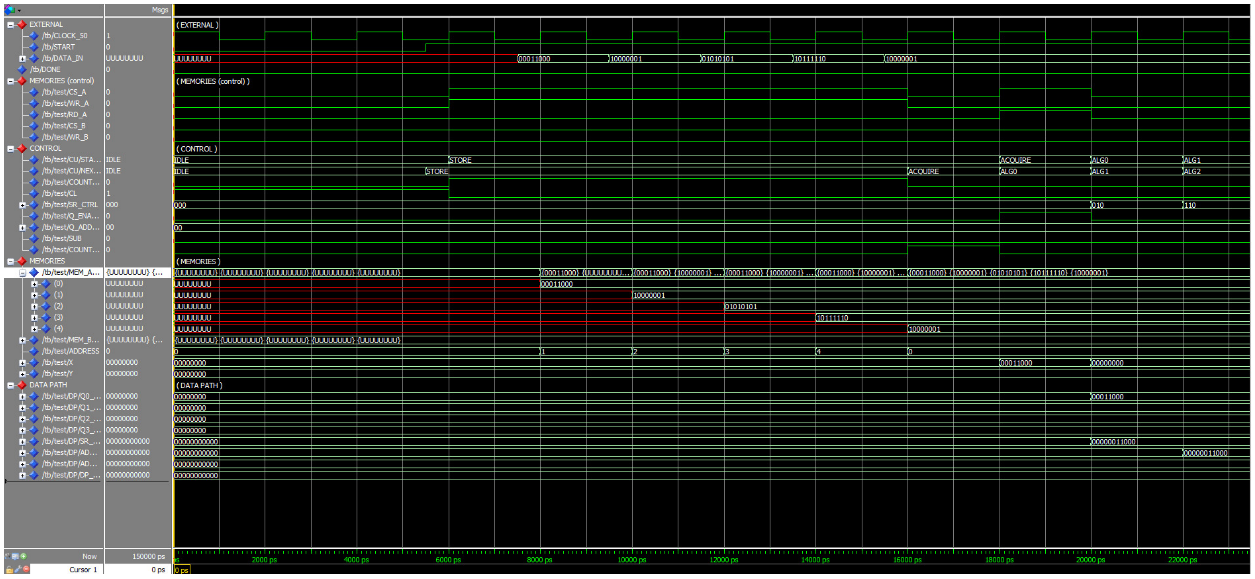


Figure 3: Data storing process

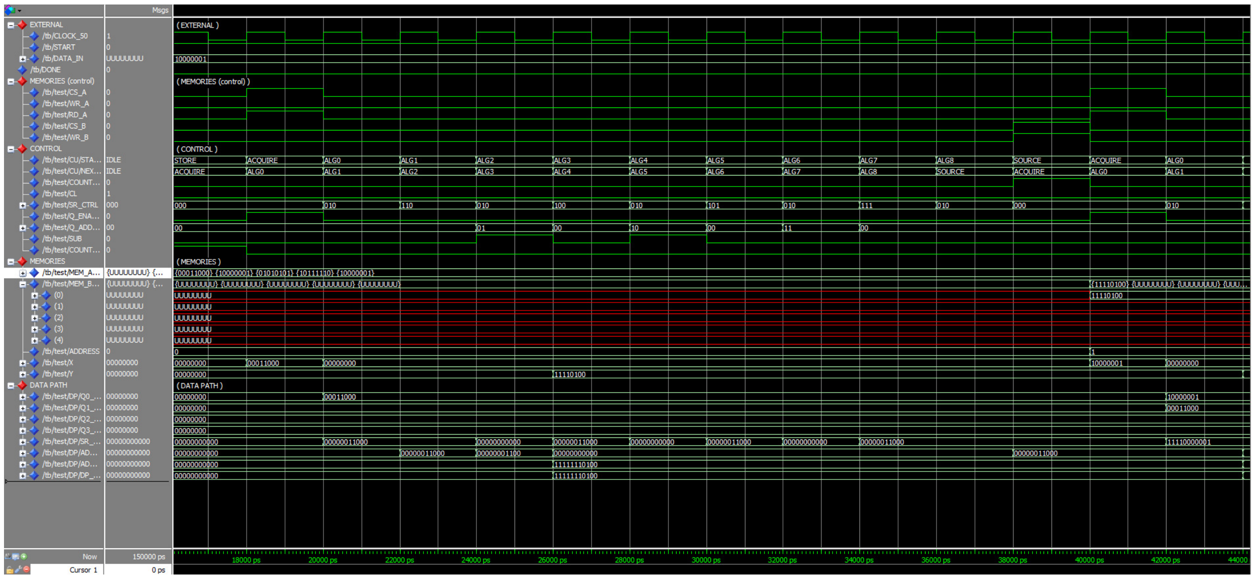


Figure 4: Algorithm cycle

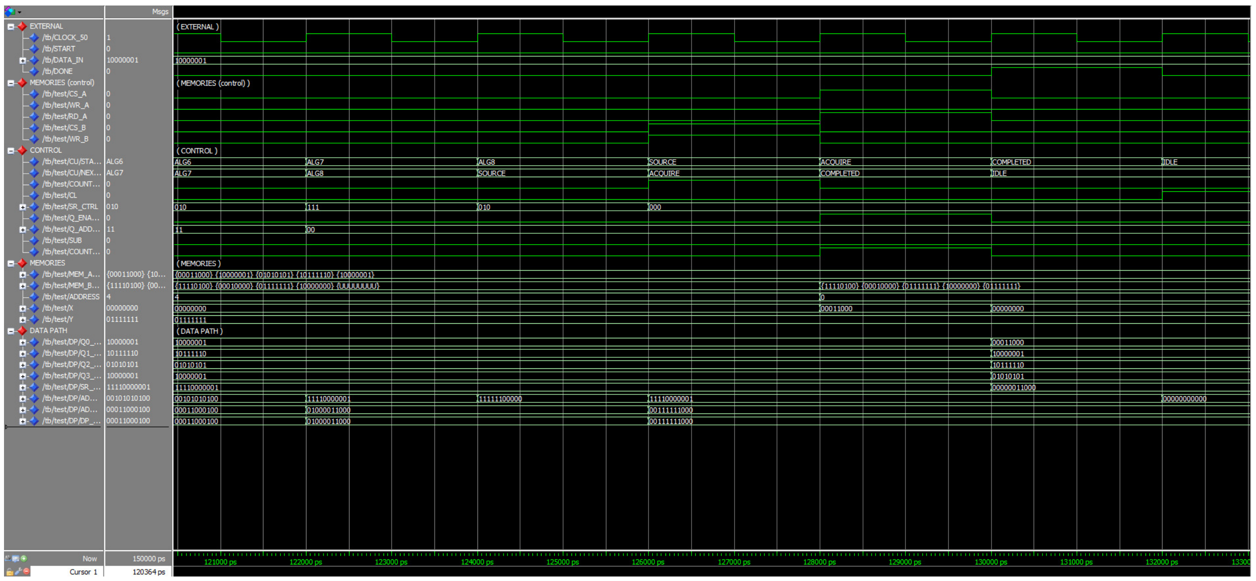


Figure 5: Last transitions up to the end

5 Conclusions

Through simulation we have substantiated the correctness of our design. Our digital filter is now ready and running. The best way to use this kind of circuit is right after an ADC which is sampling an analog signal, and then use a DAC to convert its output back to analog. If the frequencies of our sampled signal are not too high, we obtain a good doppelganger of a filtering process (even if our digital circuit is of the FIR type).