Instituto Superior Técnico

MEEC21 - Electrical and Computer Engineering

Digital System Design - L2 Report

2023/2024 – P1

October 11, 2023

Group 1

Jean-Baptiste Denoël, 1108427
Tommaso Fava, 1108632

Contents

1 Introduction

This document presents the work done for the L2 activity of the Digital System Design course. Here, the goal was to design a dot product calculator of two vectors p and w of size 4, focusing on schedule and resource sharing.

$$D(p, w) = ((p_1 \cdot w_1) \cdot ((p_2 \cdot w_2) + ((p_3 \cdot w_3) + (p_4 \cdot w_4)))) \tag{1}$$

Vivado 2023.1 was again used as the unique tool for all the steps, but this time no implementation on the board was performed. Hence, the design is evaluated only by a simulation controlled by a custom testbench.

2 Dataflow

The design phase started by defining the dataflow graph of one dot product calculation as showed in Figure 1 . To address this, it was followed the order of operations determined by the Equation 1, taking care of their sequence. In this design 4 multipliers and 3 adders are used but only 2 multipliers and 2 adders are available, that is why it's fundamental to share operators.

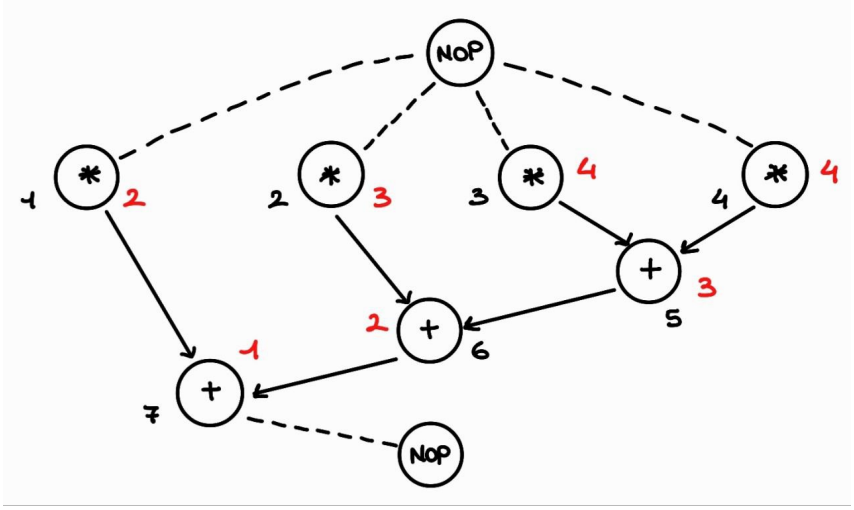

Figure 1: Diagram of the dataflow (labels of operators in black, order in the priority list in red)

3 Priority List

In order to share operators, a priority list is defined using the critical path as a metric. This defines which operations must be performed before the others.

Knowing that, the idea is to look at the dataflow graph to identify the operations with the longest path (until the final node *NOP*) and put them at the beginning of the list. This will then be adjusted taking into consideration the dependencies between those operations.

Regarding the graph showed in Figure 1, the priority list is reported in Table 1.

Operator	Priority CP	Dependancies
3 (*)	**4**	
4 (*)	**4**	
2 (*)	**3**	
5 (+)	**3**	(3,4)
1 (*)	**2**	
6 (+)	**2**	(2,5)
7 (+)	**1**	(1,6)

Table 1: Priority list using the critical path as metric.

4 List Scheduling

The following point is the list scheduling, which is about assigning a clock cycle to each operation and can be performed once the available resources have been specified. Here, the assignment conceded two adders and two multipliers, with the first operating in half clock cycle and the latter in one.

The first graph can thus be reshaped to assign the blocks in different rows corresponding to consecutive clock cycles. The order of the operators is determined by the priority list created before. The resulting schedule is shown in Figure 2, where two adders are part of the same cycle because of their execution time (half cycle each).

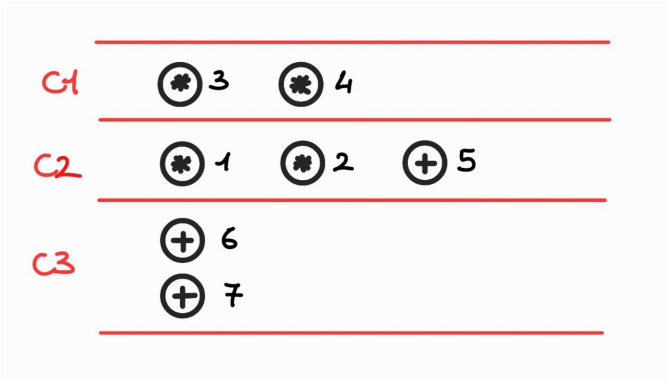

Figure 2: List scheduling.

5 Circuit Design

Based on the schedule, it is possible to use VHDL language to design a circuit that implements the requested dot product calculation.

5.1 Resource binding

Before starting to design the circuit, there is the need to map each specification object to the selected resource. This process is called resource binding and its goal is to use the smallest number of operators.

To achieve it, one must try to assign different operations to the same component but this is possible (compatible operations) only if they are of the same type and are not executed on the same time. Those information are easily displayed in the schedule, from which is possible to build the Table 2.

Operation	**Compatibility**
3 (*)	1, 2
4 (*)	1, 2
2 (*)	3, 4
5 (+)	6, 7
1 (*)	3, 4
6 (+)	5
7 (+)	5

Table 2: Compatible operations.

The resulting binding is showed in Figure 3. As one can see, the structure is the same used for the schedule but here different colours are used to distinguish the components (same color means same shared operator). Moreover, the registers are also considered.

Those registers are responsible for storing the data (*z1*, *z2*, *z3*, *z4*, *z6*) between the cycles. Here, the idea is again to minimize the resources and thus some registers are used with different input in different moments. This works particulary well for *R1* and *R2*, while the usage of *R4* in addition to *R3* is due to the choice to improve clarity and avoid an additional multiplexer (trade-off between registers and multiplexers).

Before moving on, for the sake of completeness, lack of *z5* is motivated by the fact that this was originally the entry for a register between blocks 6 and 7, which was removed since there can not be a register between two operators working in the same clock cycle.

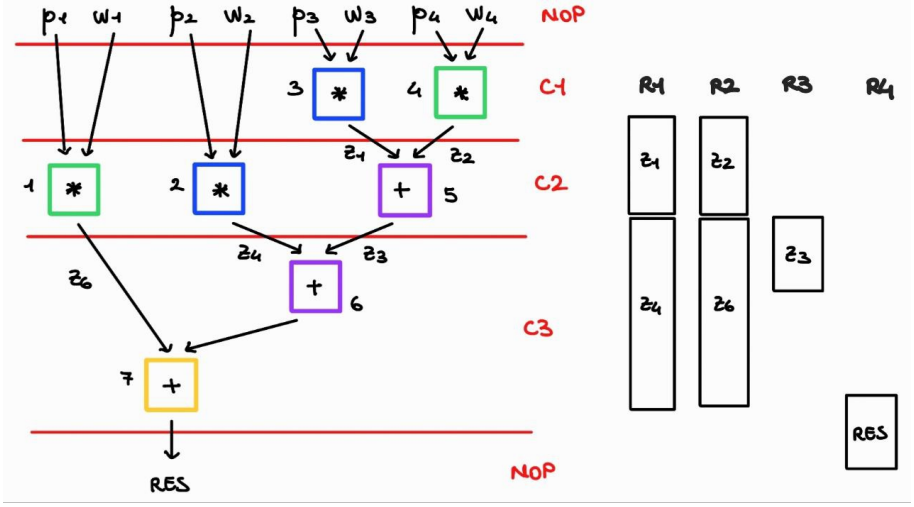

Figure 3: Binding with 2 MUL + 2 ADD.

Figure 4 helps to understand better the behaviour of these registers, showing both the input and enable signals. There is a total of 12 registers: *R[1-4]* described before and others 8 which store the input values (*RP[1-4]*, *RW[1-4]*). The goal is to define the need for multiplexers as well as the sequence of the enables signals.

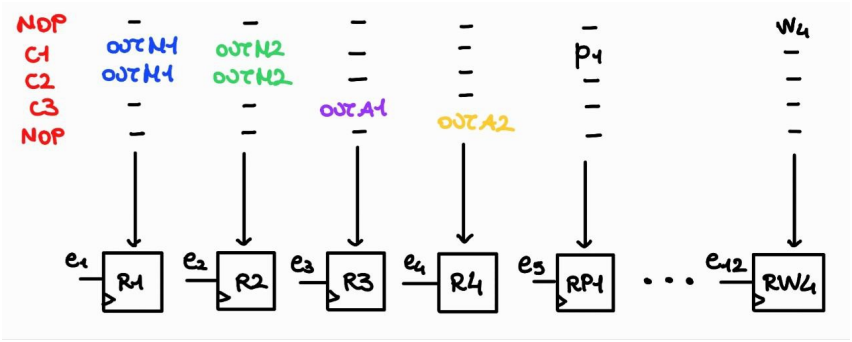

Figure 4: Register sharing.

Considering those signals with respect to the clock cycles (keeping in mind that a register storing a value in cycle j must be enabled in cycle $j-1$), the scheme reveals that no multiplexer is needed because none of the registers processes data from more than one bus. Table 3 shows instead the bits for the enables, which are identified directly from the diagram.

	R1	R2	R3	R4	RP1	RP2	RP3	RP4	RW1	RW2	RW3	RW4
State	**e1**	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**	**e8**	**e9**	**e10**	**e11**	**e12**
BEGIN	0	0	0	0	0	0	1	1	0	0	1	1
C1	1	1	0	0	1	1	0	0	1	1	0	0
C2	1	1	1	0	0	0	0	0	0	0	0	0
C3	0	0	0	1	0	0	0	0	0	0	0	0

Table 3: Enables bit for the intermediate registers.

In addition to the register sharing, the sharing of operators must be considered. A similar analysis has been made in Figure 5.

Here, the diagrams points out the need of 5 multiplexers for *R1*, *R2* and *R3* to allow operations on data coming from different registers. With the personal choice of using bit *0* for the first cycles and *X* when the operator will not be used (meaning that the following register won't be enabled), the sequence of control bits is reported in Table 4.

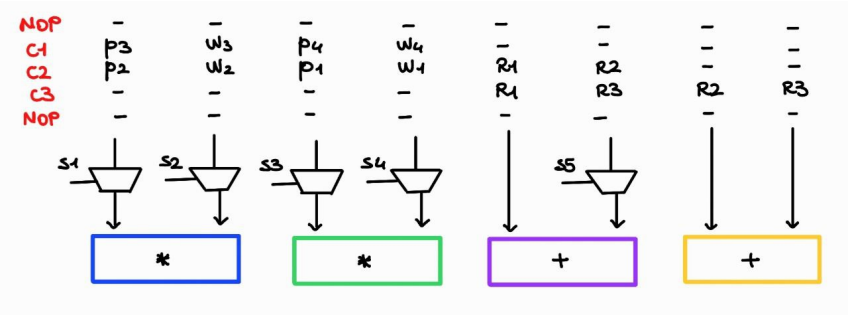

Figure 5: Operators sharing.

State	s1	s2	s3	s4	s5
BEGIN	X	X	X	X	X
C1	0	0	0	0	X
C2	1	1	1	1	0
C3	X	X	X	X	1

Table 4: Selectors bit for the multiplexers.

The final result of all this steps is the circuit architecture shown in Figure 6.

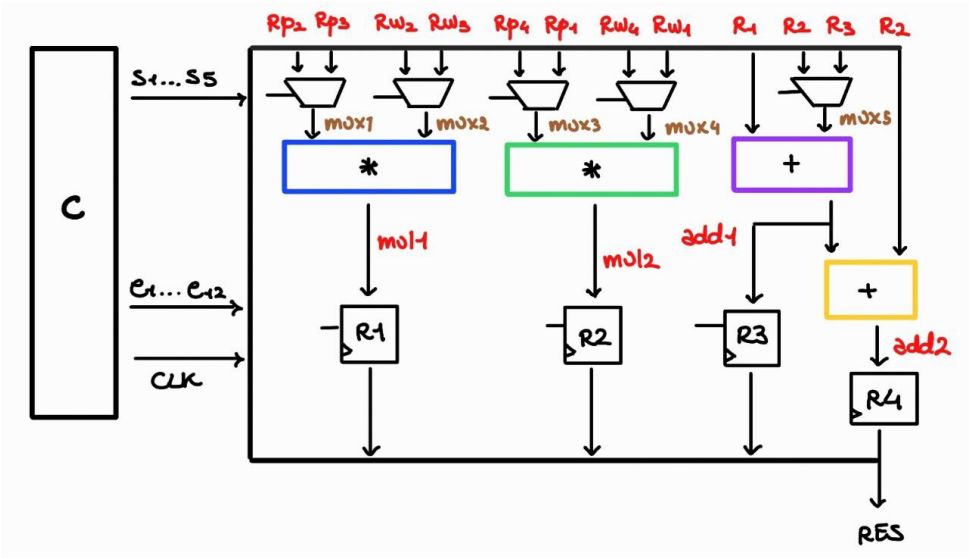

Figure 6: Final circuit architecture.

5.2 FSM with Datapath

The VHDL design of the circuit follows the sketched architecture. The structure is a finite-state machine with a datapath entity operating based on the signals (enables, selectors) coming from the control unit.

The datapath is the first thing to be designed and its ports - which are a direct consequence of the architecture - are shown in Table 5, with the size of p and w being specified in the assignment (vector p has integer values in range [0-255] while the vector w in [-128, 127]). Those signals are then used together with the internal signals from the multiplexers (*mux[1-5]*), the operators (*mul[1-2]*, *add[1-2]*) and the registers (*R[1-4]*) to obtain the desired behaviour, which involves the usage of the type *signed* for all the data entering the operators and a particular attention to the size of the vectors to avoid overflows. Moreover it must be highlighted that there is a need for padding *R2* when used as an input for *mux5* because *R3* is larger than it.

For the control unit, from the assignment the circuit must have an active-high *rst* and a *done* signal which is set when the result has been stored in the output register. Adding this with the knowledge gained from the schedule, the result is the one shown in Figure 7.

Port	Type	Direction
clk	std_logic	in
rst	std_logic	in
p[1-4]	std_logic_vector(7 downto 0)	in
w[1-4]	std_logic_vector(7 downto 0)	in
e[1-12]	std_logic	in
s[1-5]	std_logic	in
res	std_logic_vector(7 downto 0)	out

Table 5: Datapath ports.

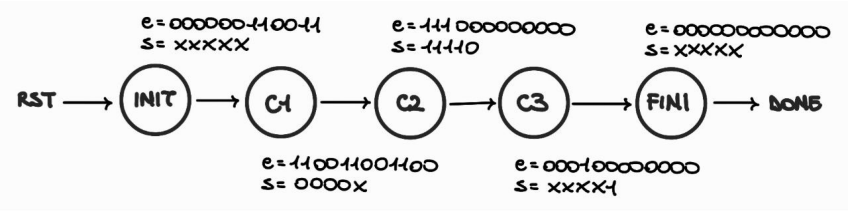

Figure 7: Diagram of the control unit.

The top view of the design is then shown in Figure 8.

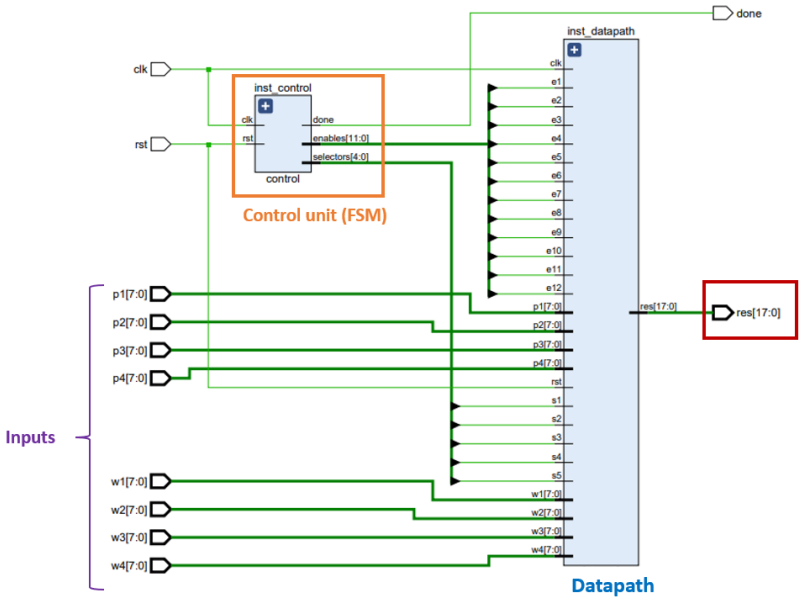

Figure 8: Schematic of the top file.

5.3 Testbench

After a synthesis without errors, the behavioural correctness of the design was checked by a custom testbench file.

Here, a single instruction is needed to launch the computation:

$$rst <= \text{`0'}$$

This choice is due to the fact that the code is structured to expect integers from the user, and so it is really fast to try new pairs by modifying the values and relaunching the simulation.

Hence, the result of a behavioural simulation with a possible set of data is shown in Figure 9, where the final value is correctly calculated and available after 3 clock cycles, as imposed in the control unit. After verifying the correctness with other values, the workflow moved into the implementation phase.

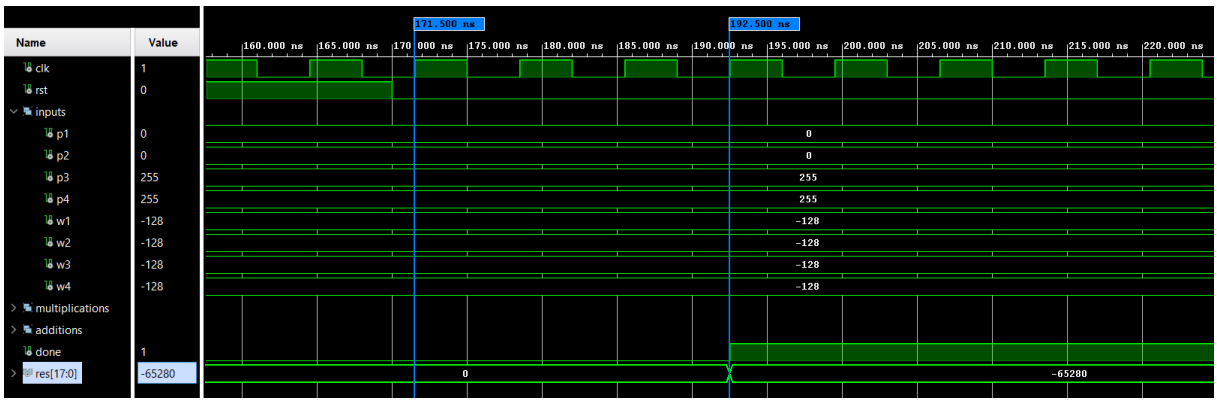

Figure 9: Result of behavioural simulation.

5.4 Utilization analysis

After the implementation, the designed circuit requires the resources presented in Table 6.

Resource	Utilization	Utilization (%)
LUT	229	1.10
FF	137	0.33
IO	85	80.19

Table 6: Basys3 utilization summary.

As it can be noticed, due to the simplicity of the datapath entity, the consumption of the FPGA primitives is pretty low. Anyway, many IO are used because of the eight 8-bit vectors as input.

5.5 Timing analysis

A reasonable post-implementation timing analysis is one of the key elements a designer must have before going into the hardware. Here, this is also more important because the assignment asked to:

1. Select the most appropriate clock period;
2. Quantify the performance of the circuit and identify its latency;
3. Indicate the critical path;
4. Justify by a post-implementation timing simulation.

First of all, the slack with a clock of 10 ns (100 MHz) was checked. Since both setup and hold time were positive, clock period in the constraints file (*.xdc*) was reduced until a negative WNS was find. The smallest possible value is 7 ns (142.857 MHz), which leads to the slack shown in Table 7.

Slack	Value (ns)
WNS	0.052
WHS	0.253
WPWS	1.500

Table 7: Post-implementation timing summary with 7 ns clock period.

Fixed the clock period, Vivado allows the identification of the critical path by simply looking at the first position in the list that appears when clicking on the WNS value, being the setup analysis the one to care about. In this case, the total delay is 6.939 ns (3.039 logic + 3.900 net) and Figure 10 shows the schematic of the path.

The new clock can then be added in the testbench to perform a post-implementation timing simulation. Differently from the behavioural simulation, this analysis takes into account the delays of the signals on the hardware and it is the best metric to evaluate the design.

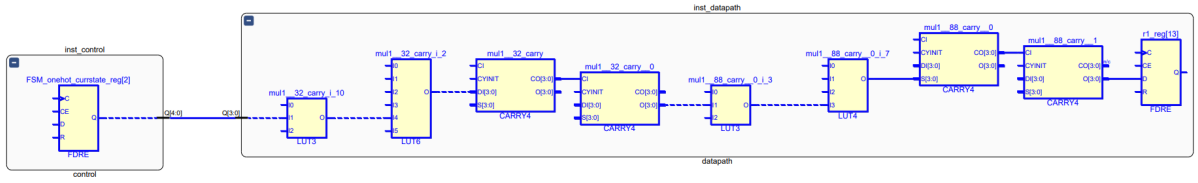

Figure 10: Critical path of the circuit.

Figure 11 shows the result of a post-implementation timing simulation performed with the same data of Figure 9. The computation ends correctly with this vectors and with all the others tested, meaning that the circuit will probably work well in the FPGA.

Regarding the delay, the difference with the behavioral simulation is clear from the third marker, which is about 9 seconds late than the second one. Anyway, considering the clock period, the total amount of time from the first rising edge of the *clk* and the availability of the result is only $\approx$ 30 ns, still a good performance.

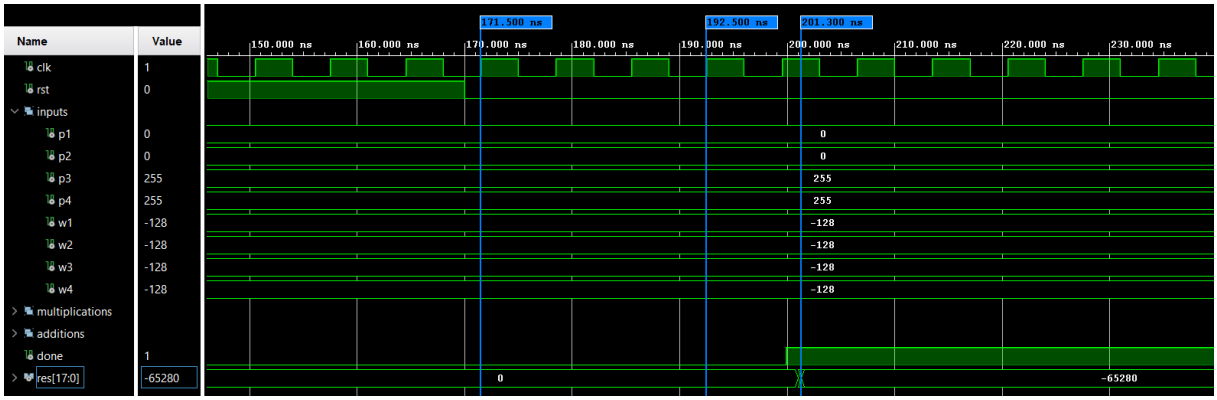

Figure 11: Result of post-implementation timing simulation.

6 Pipelining

The last task was to consider the same dot product applied to vectors of size 1000 instead of 4. The two vectors p and w are stored in one independent 500×32-bit memory with 2-port for reading, and the goal is to maximize the circuit performance while using a limited amount of resources.

To do this, a MAC architecture that works as a multiplier and accumulator is used:

$$\sum_{i=1}^{N=1000} (x_i \cdot y_i)$$

Having 2 ports for reading, only 4 values at a time can be accessed for every vector. All the 4 available multipliers are then used to calculate the result of $p_1 \cdot w_1$, $p_2 \cdot w_2$, $p_3 \cdot w_3$, $p_4 \cdot w_4$. After that, they are added two by two to obtain pw_{1+2} and pw_{3+4}, which are finally added together to obtain $pw_{1+2+3+4}$. To extend this to 1000 values, this process must be repeated 249 times adding the result to the previous one with another adder.

To resume, the architecture is composed of 4 multipliers and 4 adders. As one multiplication still requires one clock cycle and the addition can be performed in half clock cycle, with this basic architecture the result will be processed according to the following data:

$$T_{clk} > T_{preg} + 2.5 \times 7 + t_{setup\_acc} > 17.5 \text{ ns}$$
$$\textbf{Execution time} > 250 \times 17.5 = 4375 \text{ ns}$$
$$\textbf{Throughput} \approx 4 \text{ pairs}/17.5\text{ns}$$

In order to optimize the process, 2 pipeline registers can be added as in Figure 12 to do different operations at the same time. Table 8 illustrates the behaviour during the first 4 clock cycles.

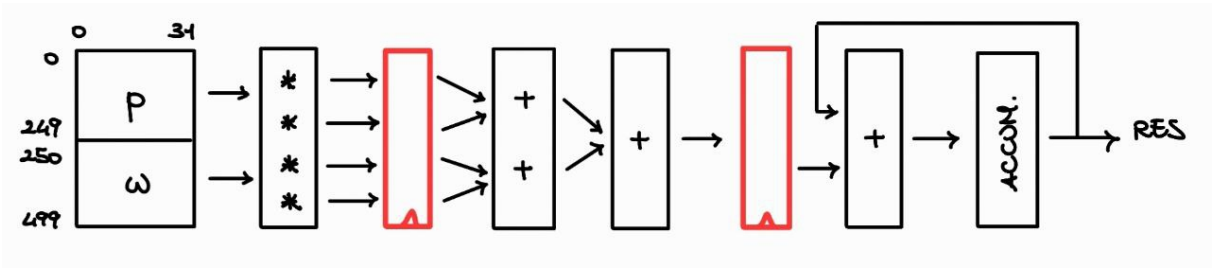

Figure 12: Block diagram of the datapath of the pipelined architecture.

C1	$p_{1\to4}$, $w_{1\to4}$	extract data
C2	p_1w_1, p_2w_2, p_3w_3, p_4w_4, $p_{5\to8}$, $w_{5\to8}$	4 multipliers + extract data
C3	pw_{1+2}, pw_{3+4}, pw_{12+34}, $pw_{1\to4}$, $p_{9\to12}$, $w_{9\to12}$	3 adds + 4 mult + extract data
C4	pw_{tot}, pw_{5+6}, pw_{7+8}, pw_{56+78}, $pw_{5\to9}$, $p_{13\to16}$, $w_{13\to16}$	1 add + 3 adds + 4 mult + extract data

Table 8: Example of the first 4 clock cycles.

The advantage of pipelining is that it increases the throughput and reduces the total execution time by decreasing the clock period:

$T_{clk} > T_{preg} + t_{Pmult} + t_{setup\_acc} > 7$ ns
Execution time $> (250 + 2) \times 7 = 1764$ ns
Throughput ≈ 4 pairs/7ns

7 Conclusion

At the end of the work, two main goals have been achieved:

1. Design of a dot product calculator for two vectors of 4 8-bit elements that operates at 142.857 MHz and computes the result in ≈ 30 ns;
2. Diagram of a pipelined architecture for a dot product calculator for two vectors of 1000 8-bit elements that improves the throughput of $\approx 150\%$ and reduces the execution time of $\approx 60\%$.

To reach this a new workflow has been adopted. Differently from the previous experience, the VHDL design started only after an accurate analysis which, starting from a simple dataflow, involved the definition of a schedule based on critical path, the resource binding and the sharing of both registers and operators. This is certainly a good step toward a better understanding of how to design a digital system and makes everything more orderly and comprehensible.

In addition, for the first time a timing analysis was performed. This allowed to clarify the practical role of both setup and hold slack, highlighting their role not only in the circuit real operation but also in the clock selection, crucial to speed up the performance of a certain implementation.

To conclude, it is sufficient to say that through this project a valuable experience has been gained. As already happened with the first assignment, there are no doubts that the work done here will play a significant role for the development of the third and last project.