AMERICAN UNIVERSITY OF BEIRUT

Optimized Vector Norm-2 List Processor

Submitted as a project for EECE 420 with Prof. Mohammad Mansour

AbdulRahman AlHamali

12/12/2015

In this report I discuss the Vector Norm-2 List Processor's basic design, and the optimizations I applied to it

Introduction:

The process of computing the Norm-2 of a complex doubly linked list stored in a dual-ported memory is a lengthy process if not optimized correctly. It involves several time-consuming tasks such as reading from memory, multiplying, accumulating, etc. In addition, it requires us to waste a cycle each time we want to move to the next element in the list. In this report I will discuss the basic algorithm used for finding the norm-2 of the list along with the basic architecture. Then, I will discuss the optimizations applied to the algorithm to minimize the number of cycles wasted when moving the pointers, and the optimizations applied to the datapath to minimize the clock cycle.

Note: The largest possible value of the Norm2 is:

$$2 \times 128 \times (-512)^2 = 67108864$$

We can reach this number if we have 128 elements in the memory, and the real and imaginary of each of those elements is equal to -512 $(1000000000)_b$. This number 67108864 is written as (100000000000000000000000000) in binary. Thus, we need 28 bits to represent this norm.

The Basic Design:

The Algorithm:

The pseudo-code for the basic algorithm is as follows:

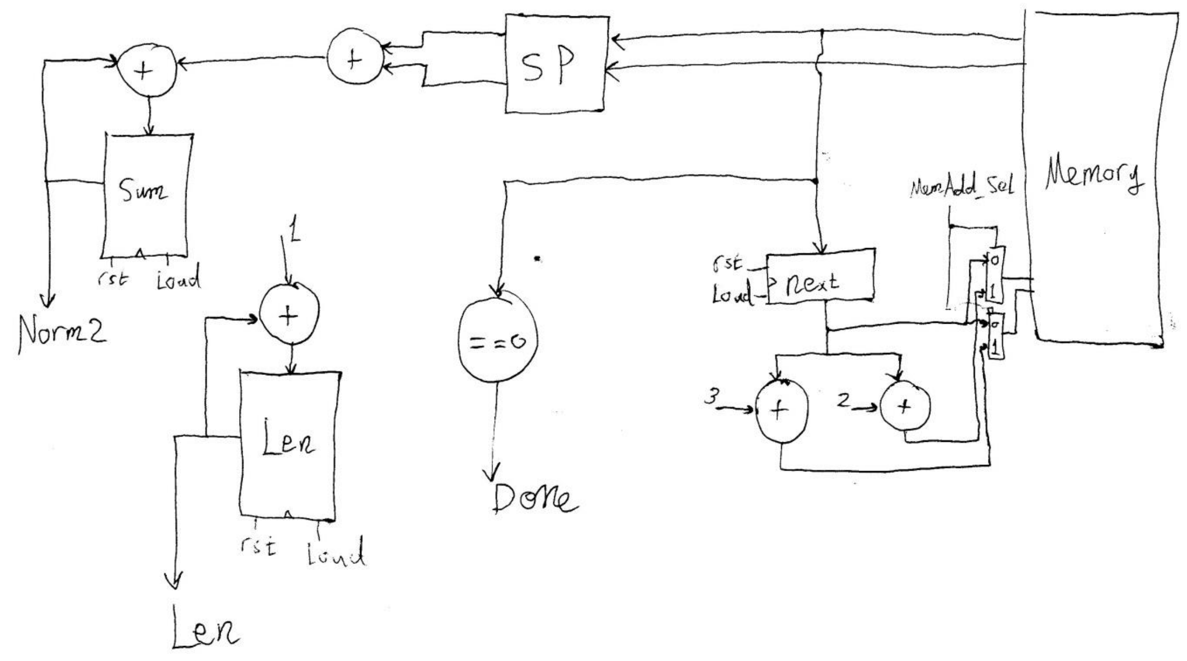
```
If (Start == 1) nextPtr<= 0, Sum<= 0, L<= 0;
repeat
{
      Sum<= Sum + Mem[nextPtr + 2]^2 + Mem[nextPtr + 3]^2, L<= L + 1;
      nextPtr<= Mem[nextPtr];
}Until(nextPtr == 0);

NORM2<= Sum, DONE<= 1, Len<= L;
```

The algorithm initializes all the registers to 0. Then, in the first cycle, it adds to the sum the squared values of the vector element currently pointed to by next, the algorithm also increments Len in the same cycle. In the next cycle, we move the pointer, and we check if we are done, if not, we repeat.

Note: There is an even more basic version in which we don't read both the real and imaginary values in the same cycle, rather each of them in one cycle. However, I didn't include it because it is too basic that contrasting the optimized design against it would look like unjustified bragging.

The Datapath:

The datapath for this design looks as follows:



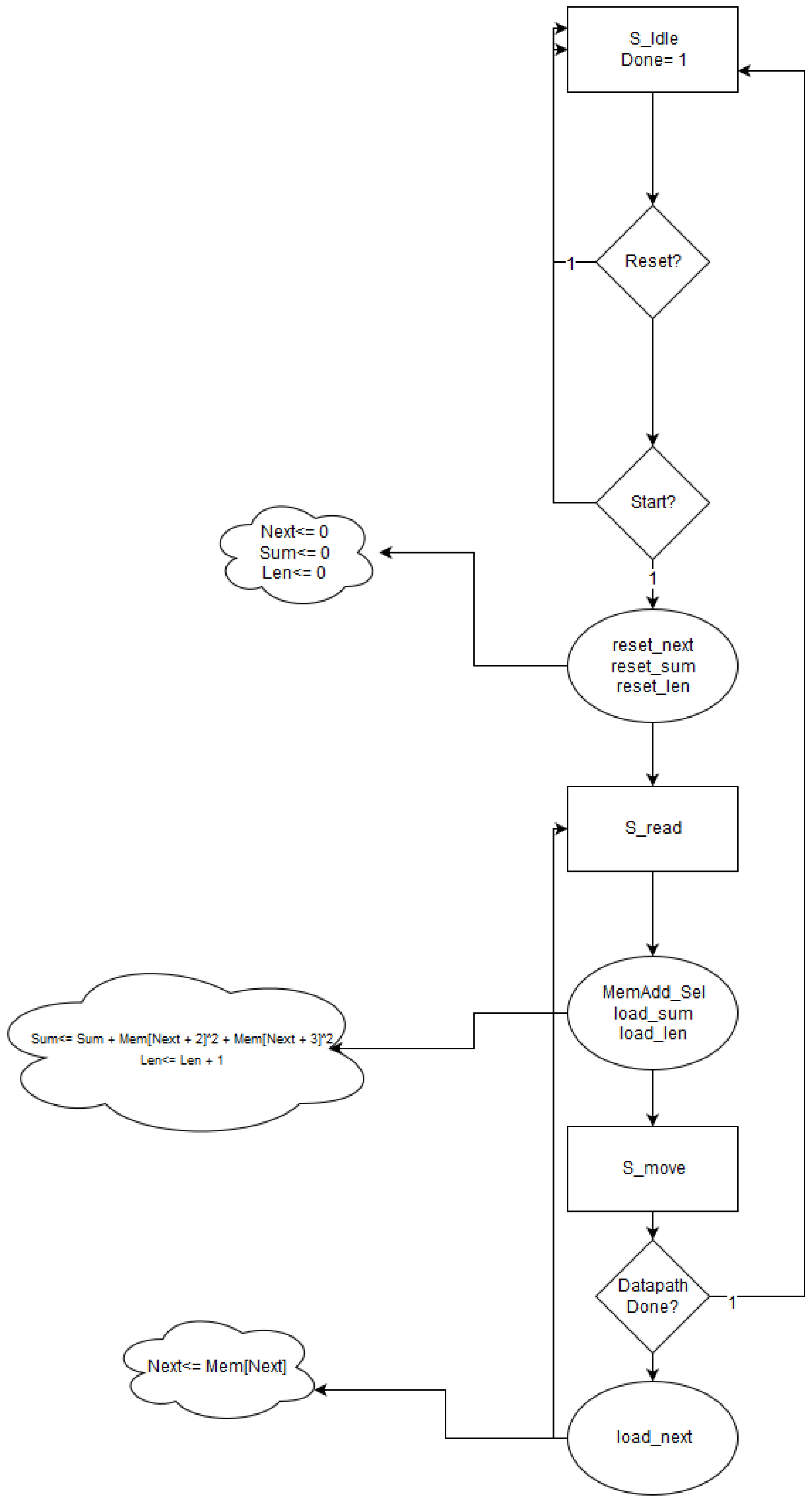
Note: The element SP stands for a Square Pair which just consists of two multipliers, each of which takes a number and squares it.

In this basic datapath, the critical path goes through: the Next register, the adders, the multiplexers, the memory, the multipliers of the SP, the first adder, the accumulation adder and finally the Sum register. Thus, the minimum clock cycle possible is:

$$0.4\ (clock\ to\ Q) + (2\log_2(9) + 1.5)\ (address\ adders) + 0.9\ (multiplexer) + 12\ (memory) \\ + (0.9(10)^2 + 1.5)(multiplier) + (2\log_2(20) + 1.5)\ (adder) \\ + (2\log_2(28) + 1.5)\ (adder) + 0.4\ (setup\ time) = 134.3ns$$

This clock cycle time could be highly improved as we will say later.

The Controller:



The Optimized Design:

The Algorithm:

The normal algorithm reads the elements from the memory as follows:

R-M-R-M-R-M-R-M-R-M-R-M-R-M.....

Where *R* means Read Element, and *M* means Move Pointer

Thus, if we have *n* elements we can read all of them in ***2n*** cycles.

The optimized algorithm on the other hand uses the fact that the linked list is doubly, and that the memory is dual-ported to reads the elements as follows:

RP-RN-M-RP-RN-M-RP-RN-M-RP….

Where *RP* means Read the Element pointed to by Previous, *RP* means Read the Element pointed to by Next, and *M* means Move both the pointers.

Thus, this algorithm can read all the elements in ***1.5n*** cycles.

The problem with this approach, however, is that it is harder to detect the end, because the two pointers will meet in the middle, and the end will depend on whether the number of elements is even or odd. I will post the pseudo-code and then will explain how the operation is handled:

```
If (Start == 1) nextPtr<= 0, prevPtr<= 1, Sum<= 0, L<= 0;
repeat
{
        Sum<= Sum + Mem[nextPtr + 2]^2 + Mem[nextPtr + 3]^2, L<= L + 1;
        nextPtr<= Mem[nextPtr], prevPtr<= Mem[prevPtr], if (Mem[prevPtr] == nextPtr + 1) break;
        Sum<= Sum + Mem[prevPtr + 1]^2 + Mem[prevPtr + 2]^2, L<= L + 1, if (prevPtr == nextPtr + 1) break;

}Until(forever);

NORM2<= Sum, DONE<= 1, Len<= L;
```

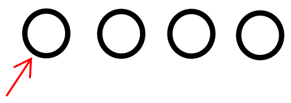
The operation goes as follows:

1- Read the data pointed to by next.

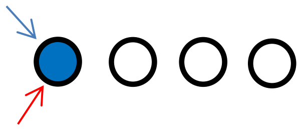
2- Move both pointers, and compare the newly fetched previous pointer to the old next pointer; if they are pointing to the same node, we stop.

3- Read the data from the previous pointer, and compare the previous pointer to the next pointer; if they are pointing to the same node, we stop.

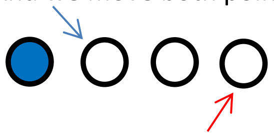
These two cases of comparison correspond to the cases where we have an odd number of elements in the vector, and an even number, respectively. Assume first that we have an even number, say 4. We start by nextPtr<= 0, and prevPtr<= 1. i.e. They are both pointing at the first element.



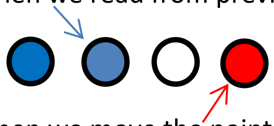
Then, we read from the element pointed to by next:



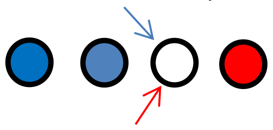
And we move both pointers:



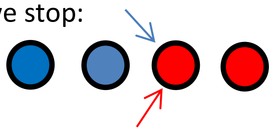
Then we read from previous, then from next:



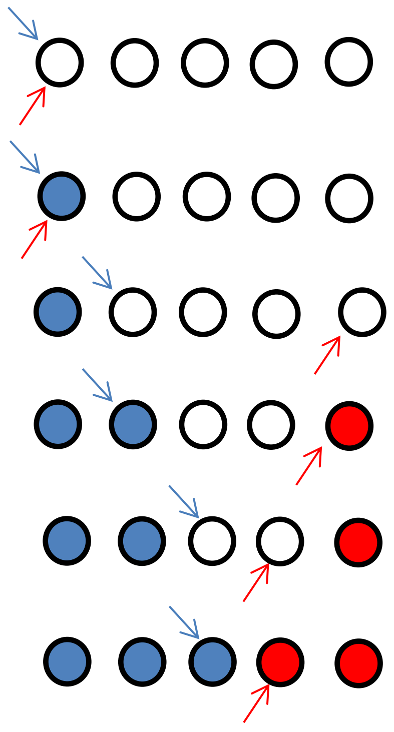
Then we move the pointers:



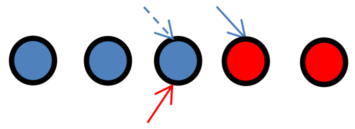
Then, we read from previous, and we find that previous and next are pointing at the same element. So, we stop:



On the other hand, if the list had 5 elements (an odd number), the process would go as follows:



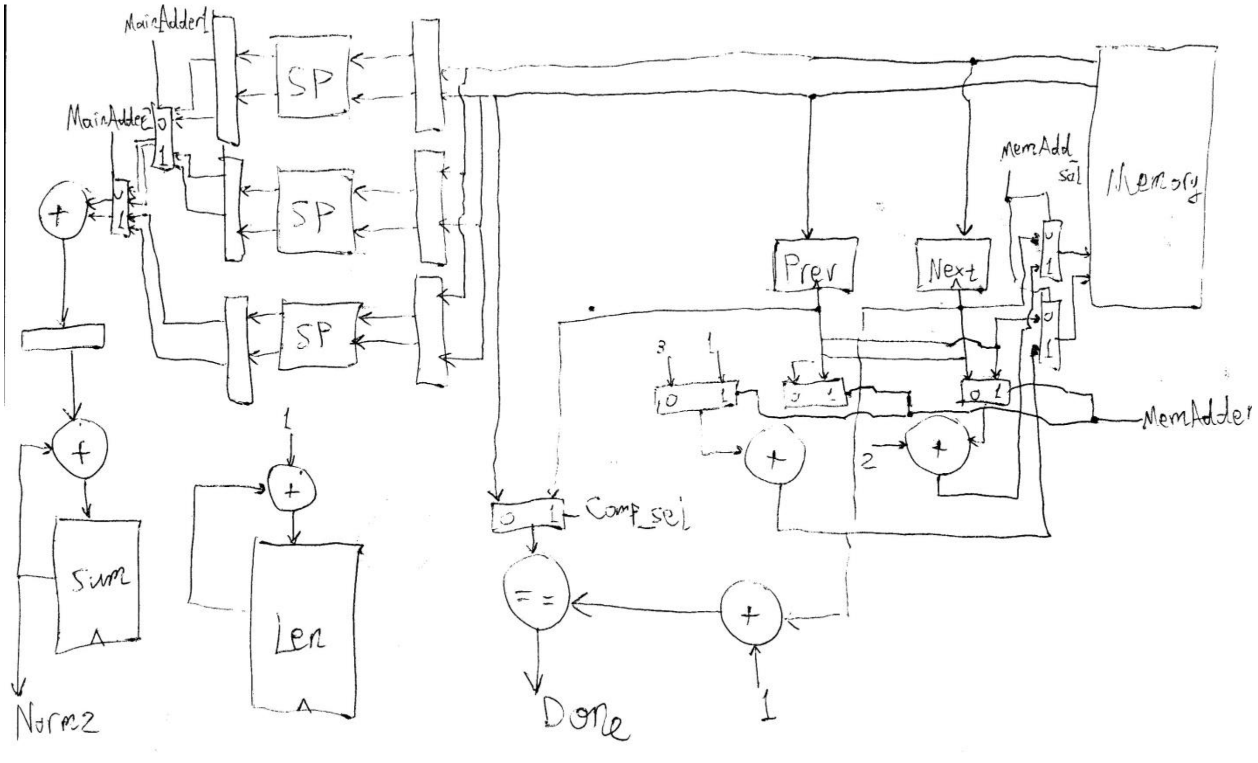
Now, as we move the pointers, we compare the new position of the previous pointer with the old position of the next pointer:



And since they are equal to each other, we stop.

The Datapath:

The optimized datapath looks as follows:



The main optimization in the datapath is to pipeline the system, in order to decrease the clock cycle length. The problem, however, is that we are limited by the latency of the multipliers (the square pairs), which is 91.5 ns. Thus, a trivial pipelined design with a single square pair can only achieve a clock cycle of 92.3 ns (latency of register before multiplier + latency of the multiplier + latency of register after the multiplier).

In order to surpass this limit, I increased the number of multipliers; I created a design in which the clock cycle is limited by the memory operations, not by the multiplications operations. The resulting clock cycle came to be:

$$0.4\ (Clock\ to\ Q) + 0.9\ (multiplexer) + (2\log_2(9) + 1.5)\ (Adder) + 0.9\ (multiplexer) + 12\ (memory) + 0.9\ (multiplexer) + 0.4\log_2(9)\ (Comparator) = 24.21$$

This clock cycle could even be reduced further by removing some of the multiplexers of the adders. However, I didn't do that, because in that case we will need more adders, etc. Which is much more hardware for very little gain in performance.

For such a clock cycle, we find out that a multiplier needs four cycles to finish its multiplication. This, however, doesn't mean that we should get 4 multipliers, because some of the cycles are actually wasted moving the pointers. We can analyze the number of multipliers needed as follows:

RN	M	RP	RN	M	RP	RN	M	RP	RN	M	RP	RN				
	SP0	SP0	SP0	SP0		SP0	SP0	SP0	SP0	SP0	SP0	SP0	SP0			
			SP1	SP1	SP1	SP1	SP1	SP1	SP1	SP1		SP1	SP1	SP1	SP1	
				SP2	SP2	SP2	SP2		SP2	SP2	SP2	SP2	SP2	SP2	SP2	SP2

In this figure, the first line corresponds to the operation that we are doing in memory; we are either reading the element pointed to by next (RN), or moving the pointers (M), or reading the element pointed to by previous (RP). Each of those elements read is to be placed on a square pair in the next cycle. I colored the element read and the square pair it is placed on with the same colors. For example, the first RN is placed on SP0, the first RP is placed on SP1, and so on so forth. From the figure above we can see that we are able to satisfy all the elements read using only three square pairs. Thus, I decided to use only three pairs.

The Control Path:

Each element we read needs to go through 4 stages: Memory – Multiply – Add – Accumulate. In addition, the movement of the pointers needs to go through only 1 stage: Memory. Thus, if we observer the movement of the elements through the datapath. We can notice the following pattern:

MemN	Mult0	Mult0	Mult0	Mult0	Add0	Acc									
	MemM														
		MemP	Mult1	Mult1	Mult1	Mult1	Add1	Acc							
			MemN	Mult2	Mult2	Mult2	Mult2	Add2	Acc						
				MemM											
					MemP	Mult0	Mult0	Mult0	Mult0	Add0	Acc				
						MemN	Mult1	Mult1	Mult1	Mult1	Add1	Acc			
							MemM								
								MemP	Mult2	Mult2	Mult2	Mult2	Add	Acc	
									MemN	Mult0	Mult0	Mult0	Mult0	Add	Acc

Notice that we MemN means read the element pointed to by next, MemP means read the element pointed to by previous, and MemM means move the pointers. In addition, Multx means insert the value read from memory to SP x, and Addx means add the results from SP x.

We can notice that at any clock cycle, we have one of the following running at once:

MemN-Acc

MemM-Add

MemP-Add-Acc

In other words, whenever we are at MemN, we are also accumulating, whenever we are at MemM, we are also adding, and whenever we are at MemP, we are also adding and accumulating.

On the other hand, we can also notice a pattern with which we choose which multiplier to feed and which multiplier to read from. This pattern goes as follows:

Feed into	Read from
0	None
None	0
1	1
2	None
None	2
0	0
1	None
None	1
2	2

The pattern keeps repeating (Feed into 0 and read from none, then feed into none and read from 0, then feed into 1 and read from 1….)

Finally, we need to know that once a Done signal is received from the datapath, we don't want to stop immediately because there are still some values in the pipeline. Thus, when we receive the Done signal, we want to wait a few cycles for the pipeline to finish processing.

Thus, we need a state machine that has 3 states (other than Idle), and we need to counters, one that chooses the SP to feed and the SP to read from, and one to tell us when to stop after receiving a Done signal.

Note: I ran out of time before drawing the ASM for this design. However, the ASM chart can clearly be deduced from the code written in ControlPath.v

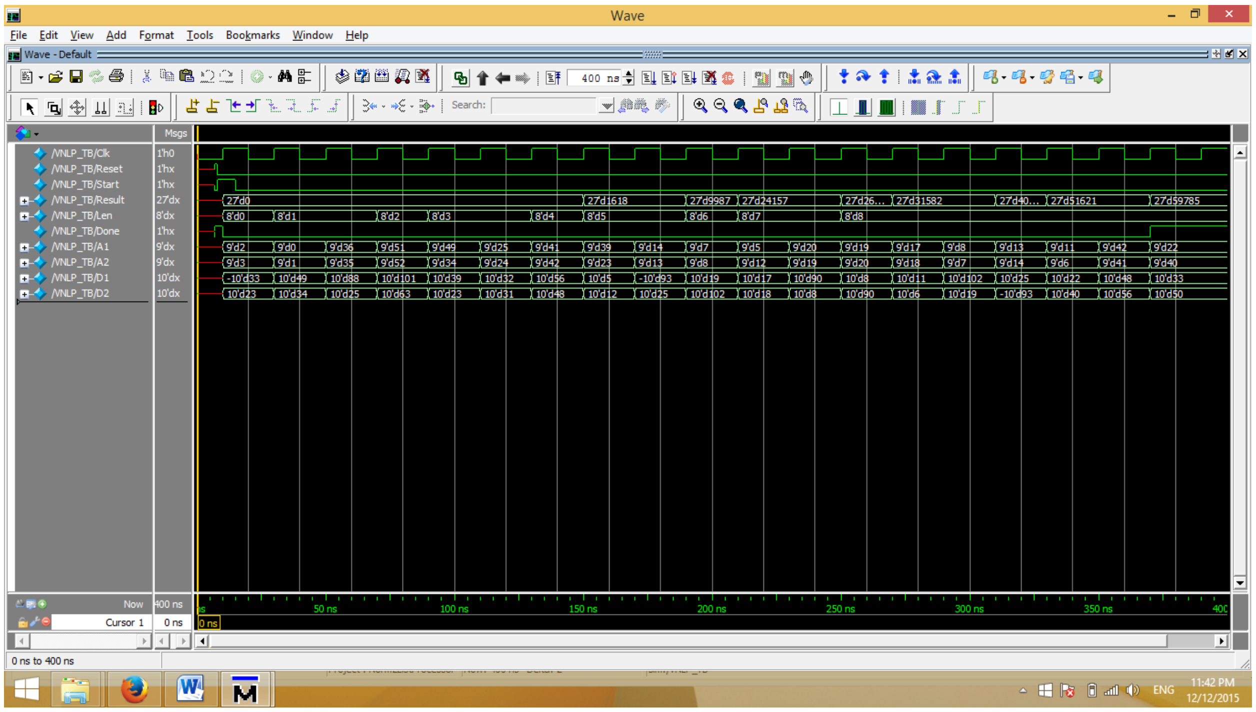
Testing and performance evaluation:

The design created was simulated using Verilog and ModelSim. It was applied to two testbenches:

1- The one provided in the project description. The design found the correct value of the Norm and the Length in:

$$19\ cycles \times 24.21ns = 460ns$$

The following timing diagram shows the performance of the system on this testbench:



2- Another testbench that fills the whole memory with 128 vectors of values $-512-512i$

The design finds the result in:

$$198\ cycles \times 24.21ns = 4793$$

The timing diagram for this system was not shown here because it does not fit. It could be viewed by simulating the testbench VNLP_TB2.

Conclusion:

The new optimized system is much faster than the original system in terms of number of clock cycles and in terms of the maximum clock frequency achievable.