Mr. Phot Koseekrainiramon z5387411

23T2 COMP4601 Week 4&5 Lab Hand-in exercises

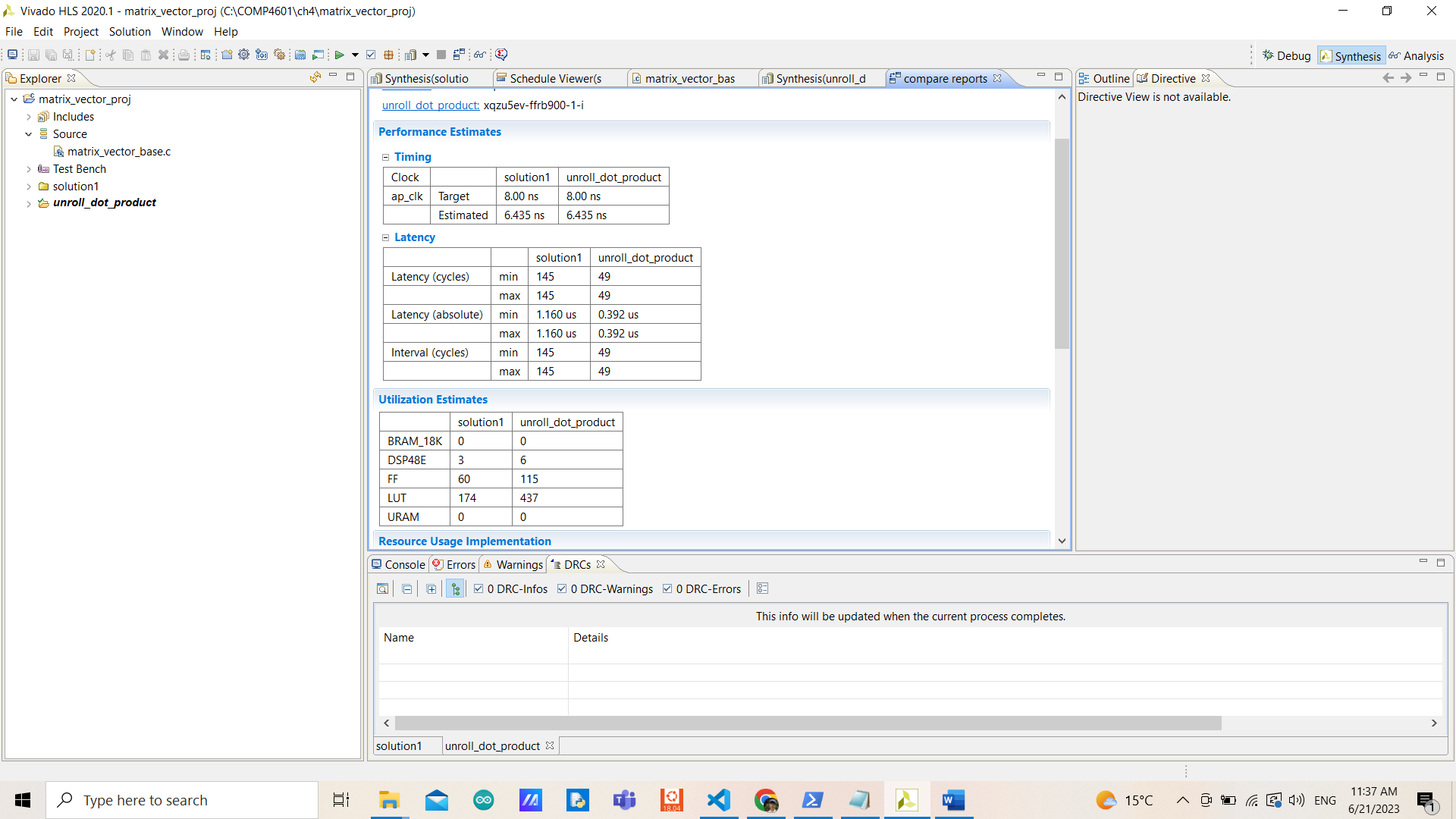
**Exercise 1:**

It has an estimated clock period of 6.435ns with 145 cycles of latency (1.160us absolute latency since we are using an 8ns target clock period, not 10ns). In terms of resource utilization, 3 DSP48Es are used (0.002% of available), with 60 flip-flops (0.0002% of available) and 174 look-up tables (0.001% of available).

dot\_product\_loop which is inside data\_loop has an iteration latency of 2 with a trip count of 8, causing it to take 16 cycles of latency. data\_loop has an iteration latency of 18 with the same number of trip counts, causing it to take 144 cycles of latency.

Only one multiplier and one adder are required.

**Exercise 2:**

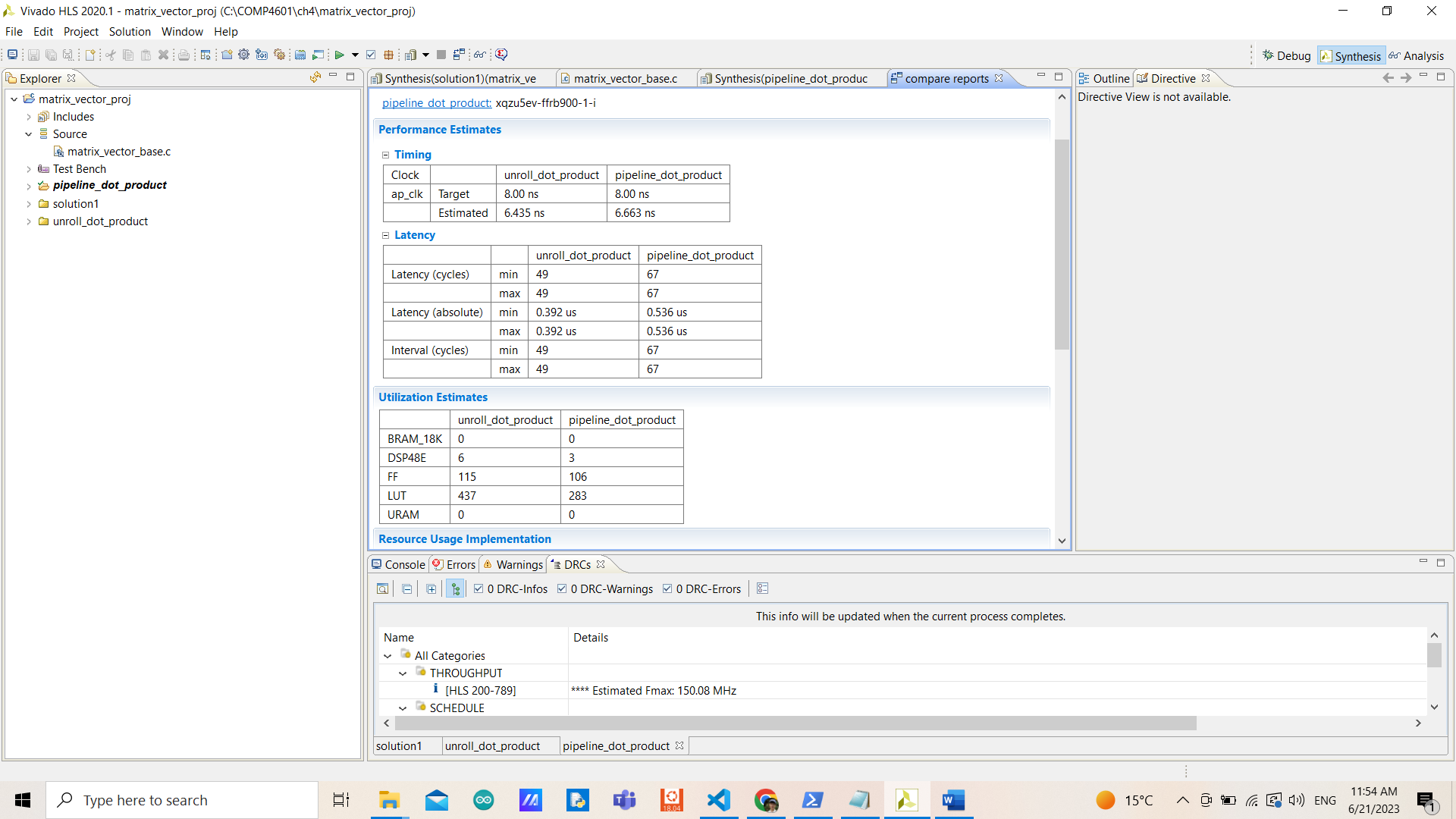


The new design has the same number of estimated clock periods but significantly fewer cycles of latency since the iteration latency of the loop is reduced to 6 with the same number of trip counts. Even though the new design is faster, it used up more resources in exchange. 3 more DSP48Es, 55 more flip-flops, and 263 more look-up tables.

It needs to do 8 multipliers but only 2 multipliers are required (6 DSP48Es) because they are done simultaneously, almost the same for adders but not all are shared.

By unrolling the inner loop, each of the multiplications could be done simultaneously with an adder tree for additions. Since loading an element from an array takes approximately 1 cycle, with a SIZE of 8, we need to load 16 elements (to do multiplication). However, we could load 4 elements at a time and perform 2 sets of multiplication before adding them up. This takes approximately 4-5 cycles since you could start loading the next set of elements while performing multiplication. Then another 1-2 cycles performing multiplication and addition of the last set causing 6 cycles of iteration.

**Exercise 3:**

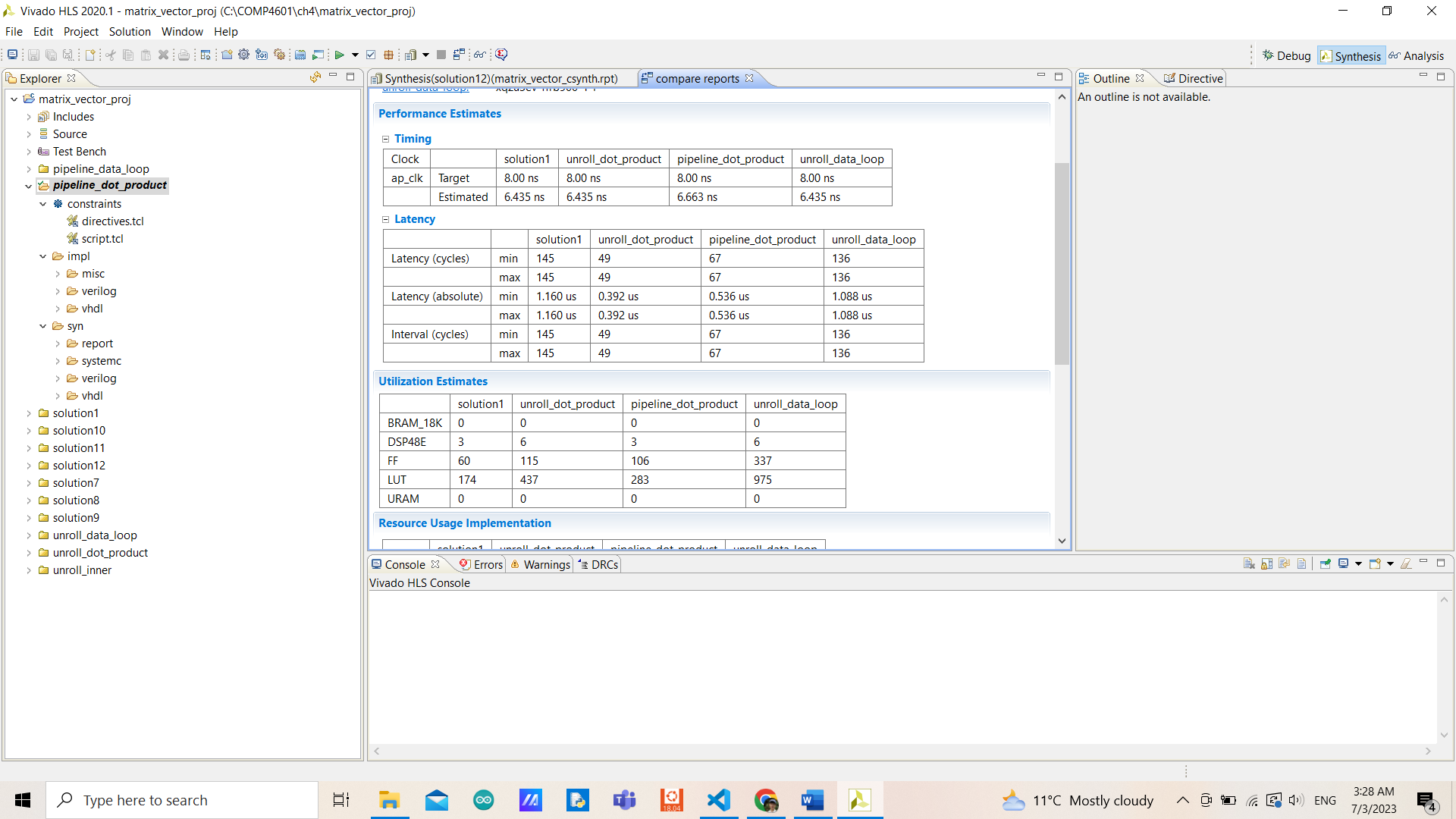


The new design has little higher estimated clock period with 18 more cycles of latency. Overall better resource utilization with 3 fewer DSP48Es since only 1 multiply needs to be done in one of the stages, 9 fewer flip-flops, and 154 fewer look-up tables.

The loop nest is gone from completely unrolling.

By performing only 1 set of multiplication in one iteration, loading 2 elements from arrays take approximately 2 cycles with 1 cycle of multiplication and addition at the end of each iteration causing 3 iteration latency. Since a new multiplication could start every clock cycle on a pipelined, the initial interval is only 1 cycle.

**Exercise 4:**

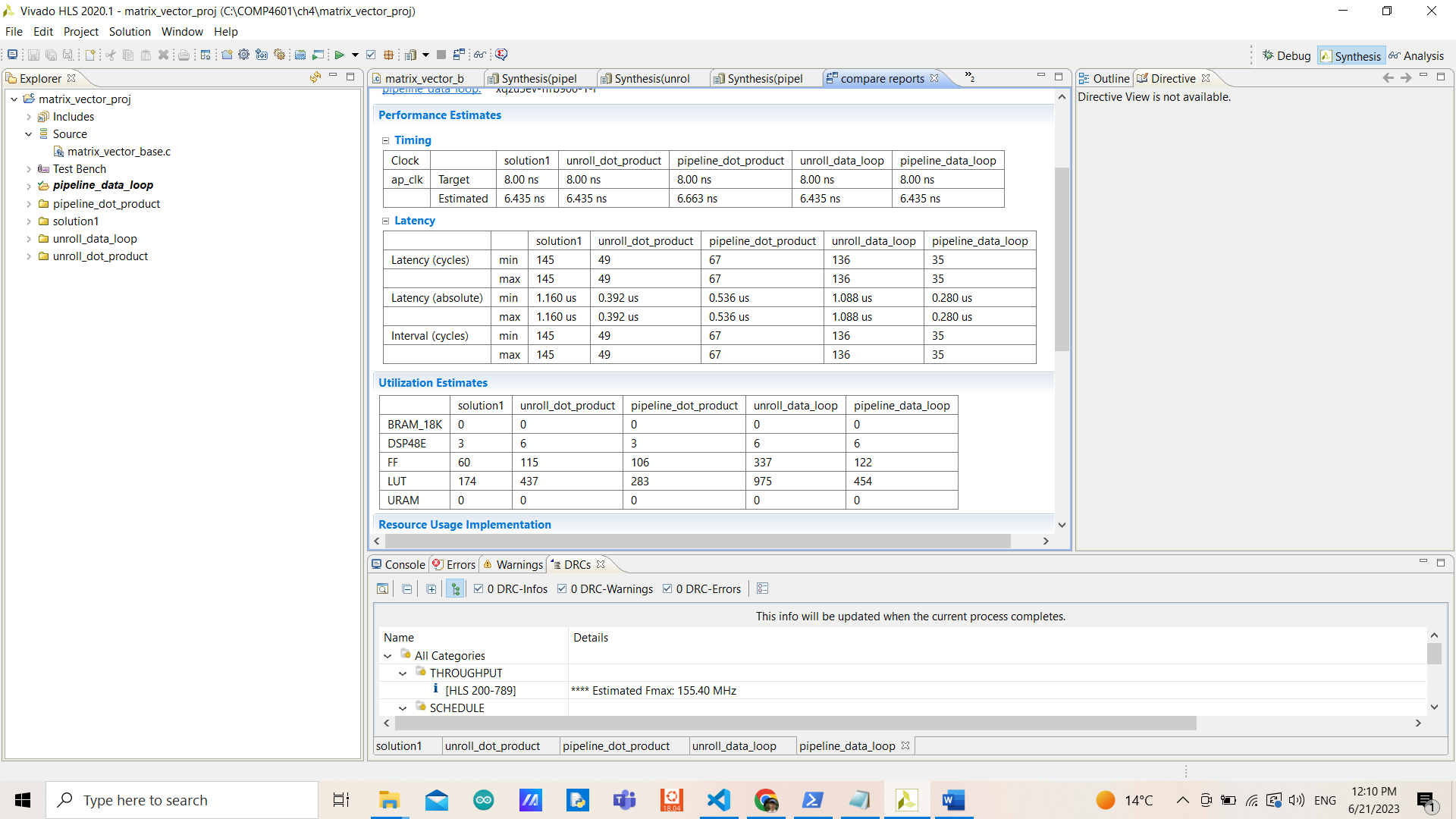


This is totally not worth it since the number of cycles of latency is only reduced by 9 with little less absolute latency than compared to the original design but with a tremendously increased number of resources required.

Compared with unrolling dot\_product\_loop and pipelining dot\_product\_loop, unrolling data\_loop is worse in every perspective (both in terms of performance and utilization). So, it is always better to choose to unroll dot\_product loop than data\_loop.

With 8 dot\_product\_loop operations that need to be performed sequentially, 16 latency each, this is because the inner loop isn’t being unrolled. Only if the inner loop is unrolled, the performance would be better than other designs but comes with a tremendously increased number of resources used.

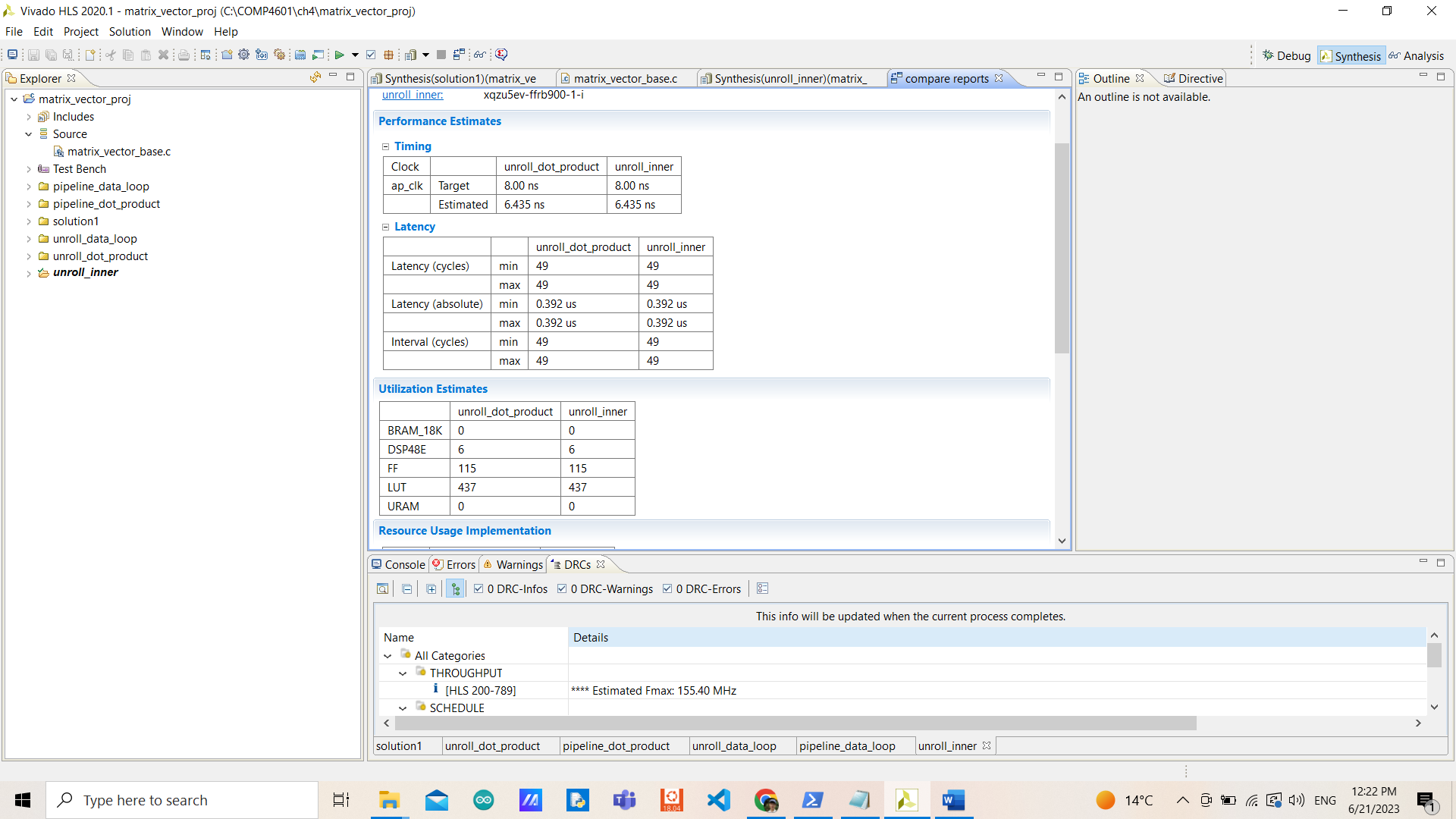
**Exercise 5:**



The inner loop is automatically unrolled (completely) causing the iteration latency to be 6 as explained in exercise 2. But with the initial interval of 4 (since 4 cycles are required to load 16 elements from arrays as explained in exercise 2), even though the trip count is 8 (SIZE), the total latency is only 35 and not 8\*6.

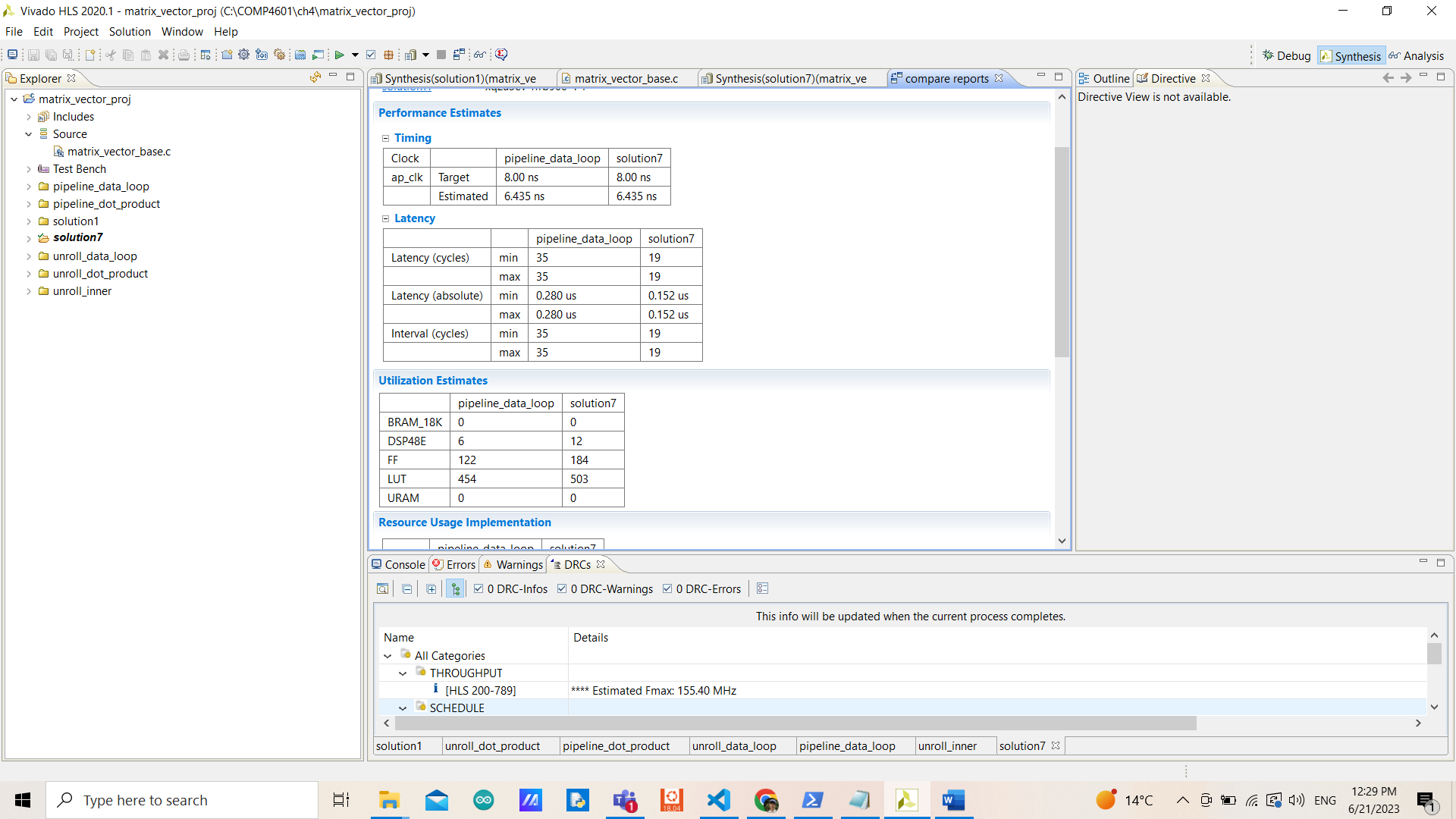
By ranking all of these designs, pipelining the inner loop would be the first since it has the third-best performance (not much differences), however, it only needs 3 DSPs with fewer LUTs and FFs compared to other designs (except the original). Second and third places go for pipelining the outer loop and unrolling the inner loop, with the first and second-best performances almost the same in terms of utilization. The original design goes forth and unrolling the outer loop goes last as explained in exercise 4.

**Exercise 6:**



Everything is the same in terms of performance and utilization comparing manually unrolling the inner loop and unrolling the inner loop using a directive. Since the logic behind the code is the same, one is letting Vivado HLS design from the given directive, another is manually editing from the source code.

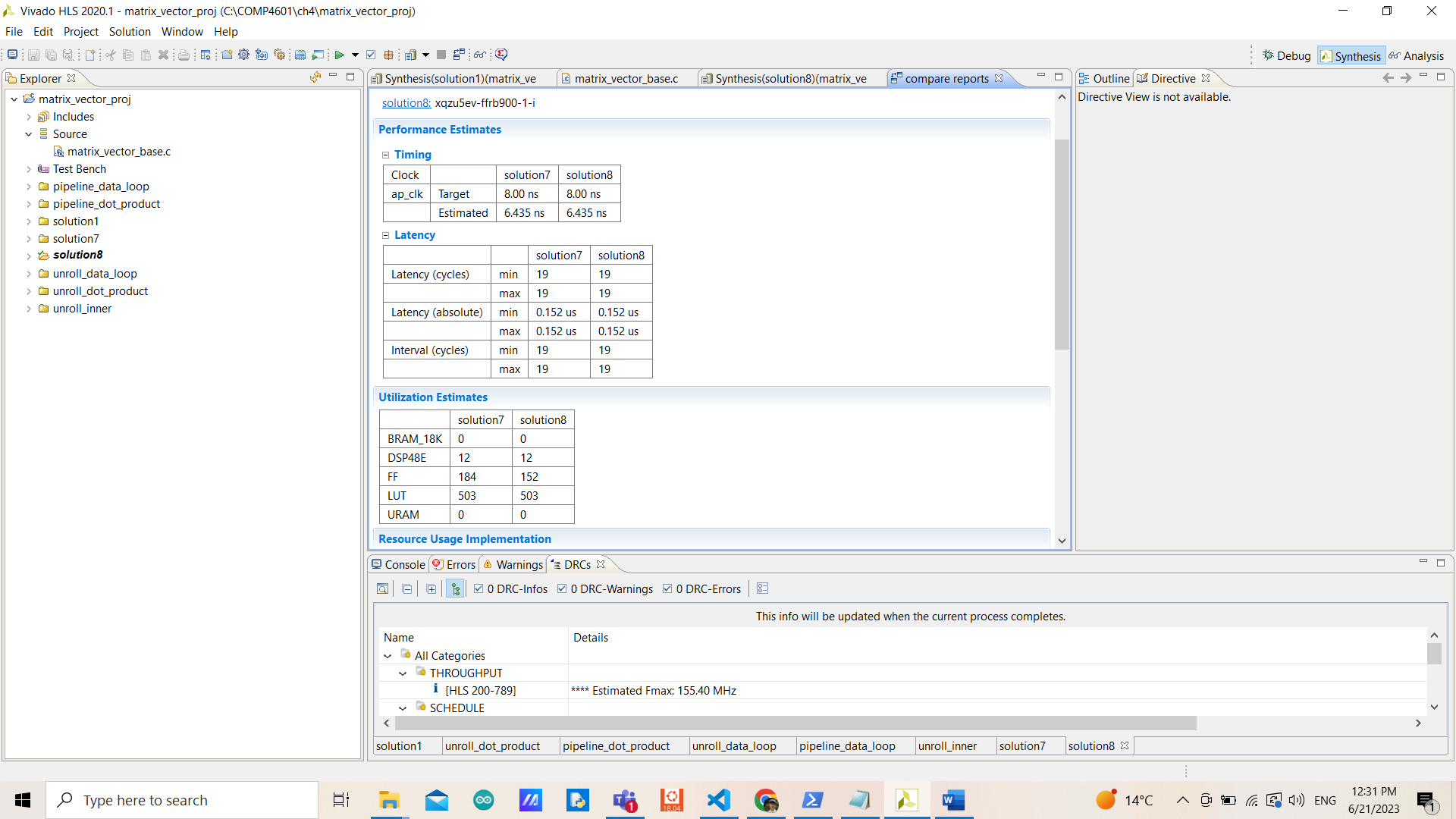
**Exercise 7:**



The new design is faster (with the same number of estimated clock periods but fewer cycles of latency) but with little more resources used (double DSPs since extra multiplications are required, and more flip-flops for array partition that split elements of an array into their own registers) for a trade-off.

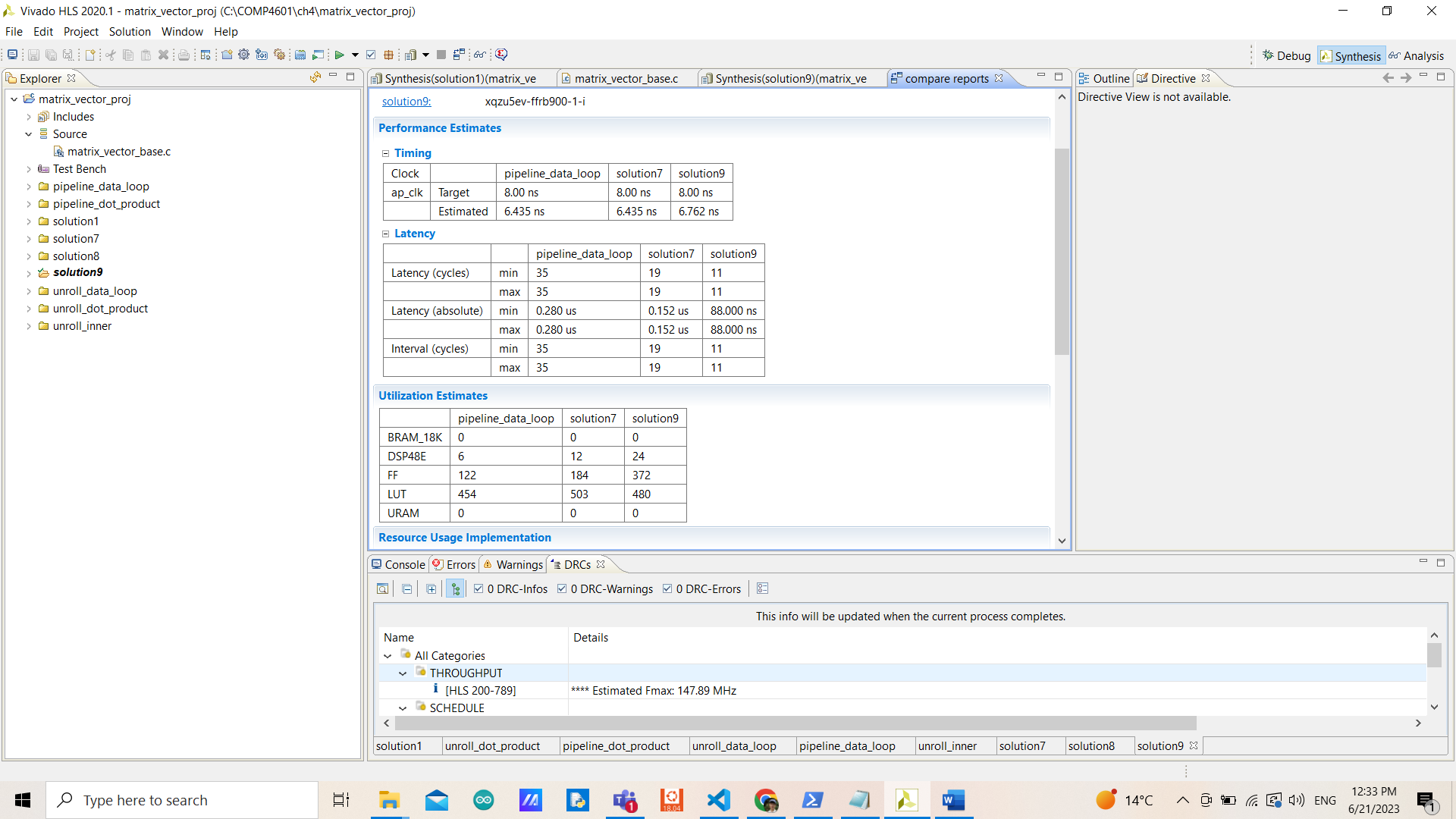
With only 4 iteration latency due to half time of loading elements from arrays and performing twice sets of multiplication since we have them as separated arrays. With the initial interval of 2 due to 2 cycles of loading elements from arrays before performing a multiplication at the end, only 19 cycles are required.

**Exercise 8:**



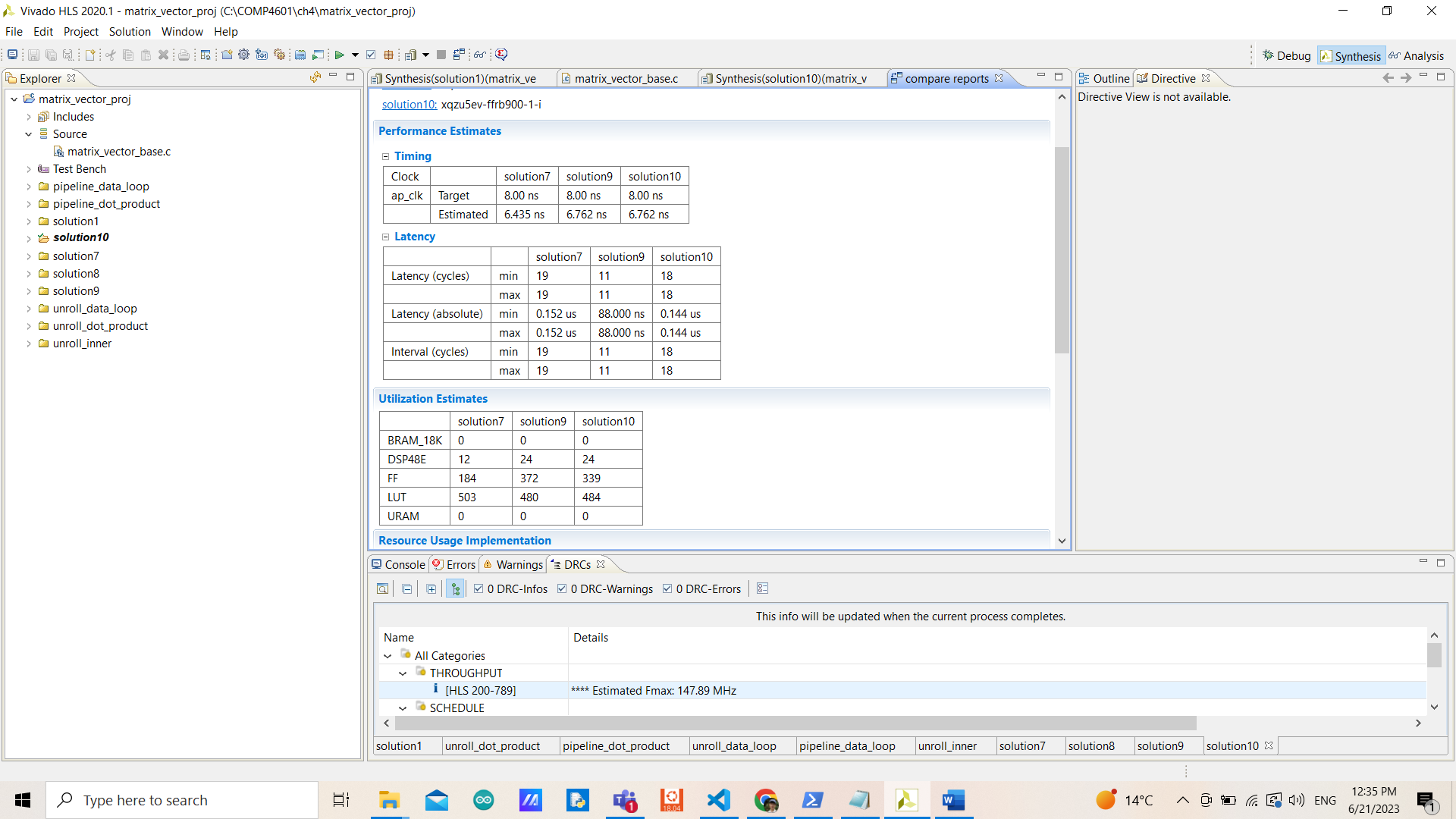
They have the same performance and iteration latency (slightly different multiplication order) but block partitioning requires fewer flip-flops (for registers). This might be because some elements are being stored somewhere else that Vivado HLS doesn’t provide more details, or might be because of some limitations of the device part we use.

**Exercise 9:**



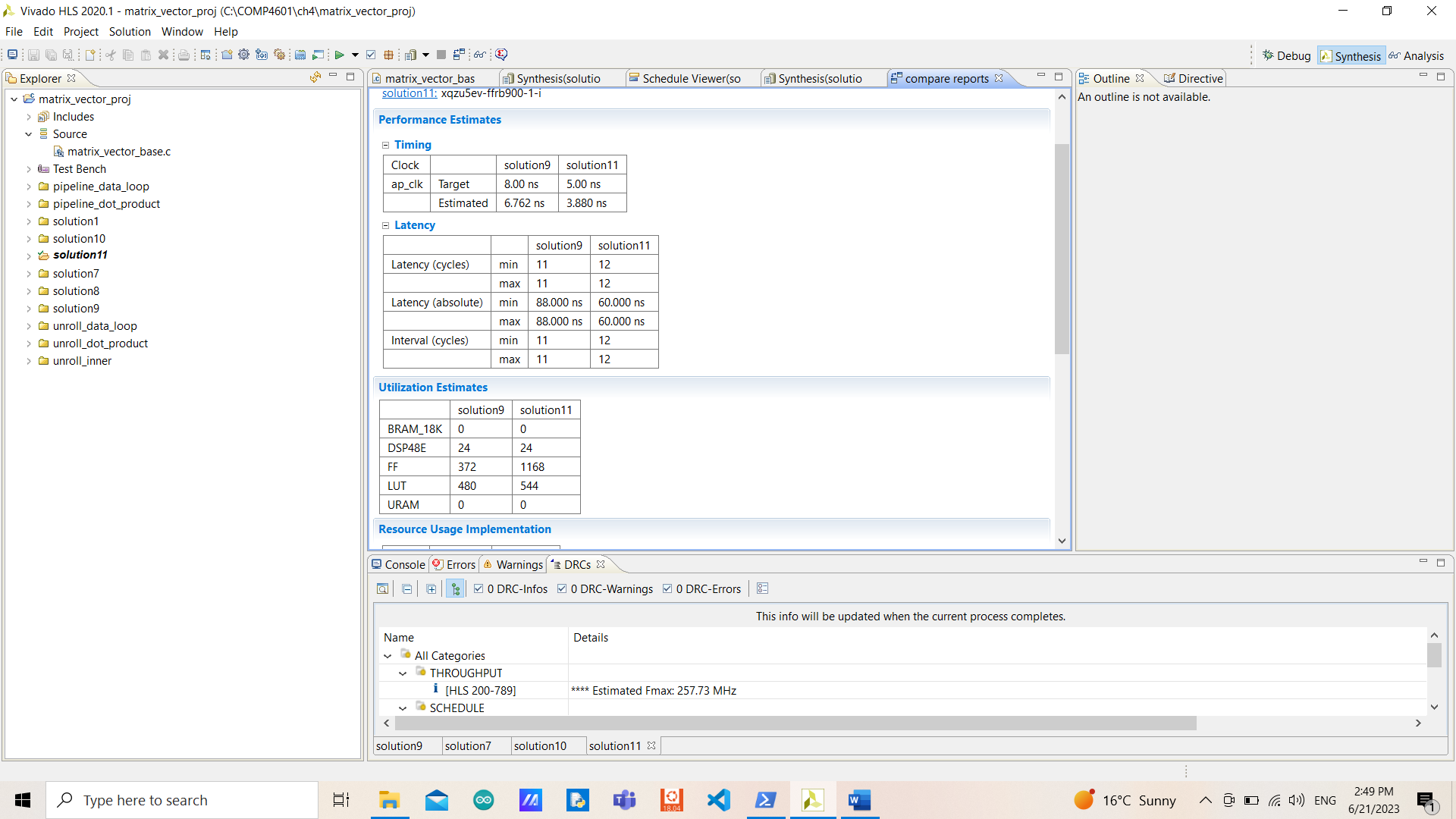
It’s obvious that, in terms of performance, the array partition is better (both complete and non-complete). However, as explained in Exercise 7 that the trade-off couldn’t be avoided. As a complete array partition, double flip-flops and DSP are required for extra multiplications and registers to store elements from the arrays separately from that non-complete array. The logic is the same as Exercise 7 in that we could load more elements from those array partitions at the time causing less iteration latency (3) and initial interval (1). (Even though array partitioning has a higher estimated clock period, with less number of cycles of latency, the performance is better than that in solution 7).

**Exercise 10:**



By forcing an extra initial interval from 1 to 2, the advantage we got from complete partitioning is gone (loading more elements at a time to have less iteration latency) while the resource usage is almost the same from having more registers like solution 9. The performance is almost no different from that of solution 7 since we are wasting 1 extra cycle for nothing when it is ready to start the next operation.

**Exercise 11:**



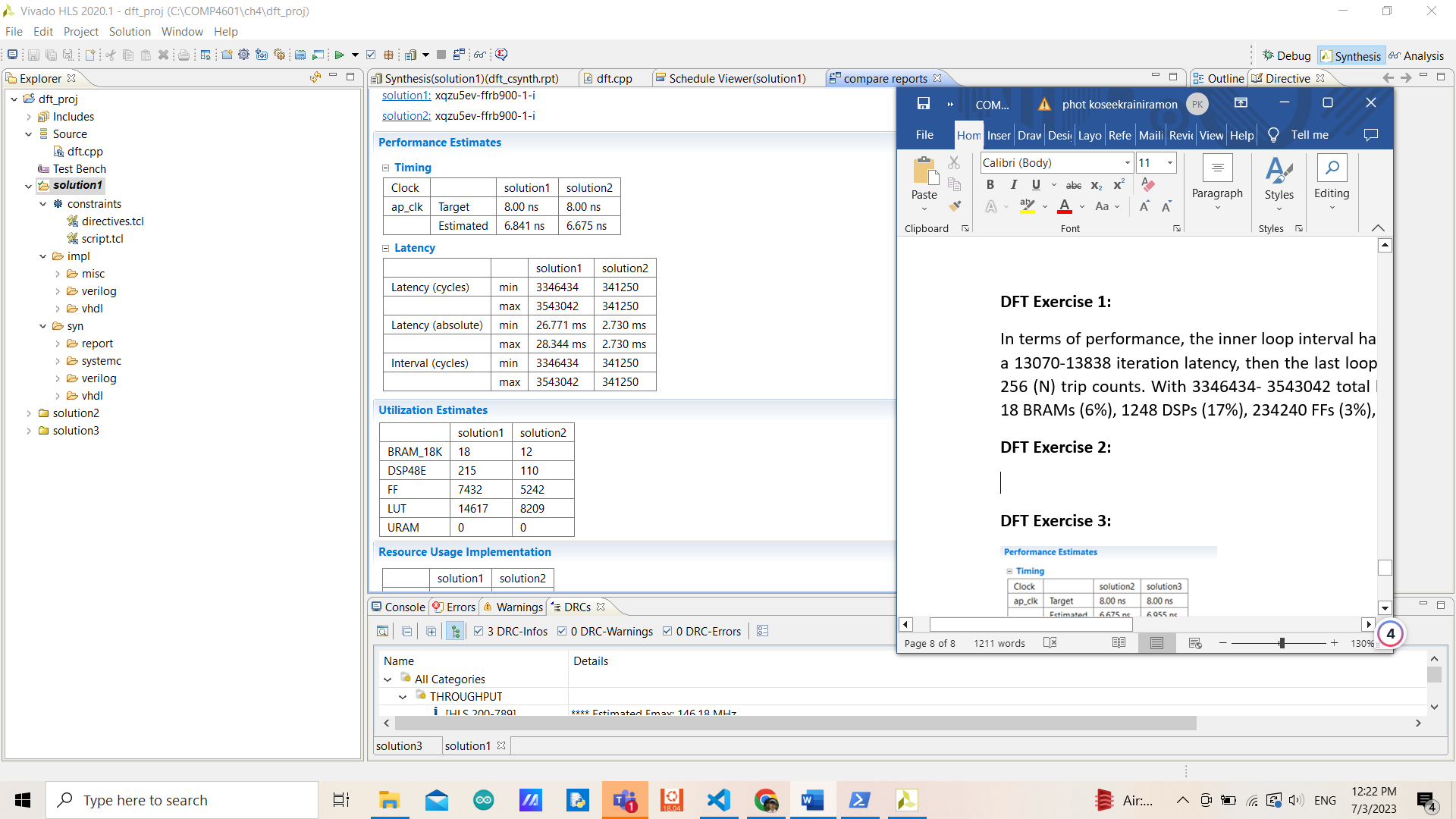
The iteration latency becomes 4, however, the order and the behavior of the operations are the same except for immediately performing multiplications after loading elements, it waits for a cycle of loading elements to pass and then performs the multiplication. This might be the way it tries to meet the target clock cycle. The new design is faster but with a significantly increased number of flip-flops. With significantly less estimated clock period, even with 1 more cycle of latency but in terms of absolute latency, solution 11 is faster than that solution 9.

Using dual-port BRAM might be a way to improve the utilization since that would not require many extra registers.

**DFT Exercise 1:**

In terms of performance, the inner loop interval has an iteration latency of 51-54 and the outer loop has a 13070-13838 iteration latency, then the last loop has an iteration latency of 2 with all of them having 256 (N) trip counts. With 3346434- 3543042 total latency. Most of the resources are used as instances, 18 BRAMs (6%), 1248 DSPs (17%), 234240 FFs (3%), and 117120 LUTs (12%) with a 6.841ms estimated clock period.

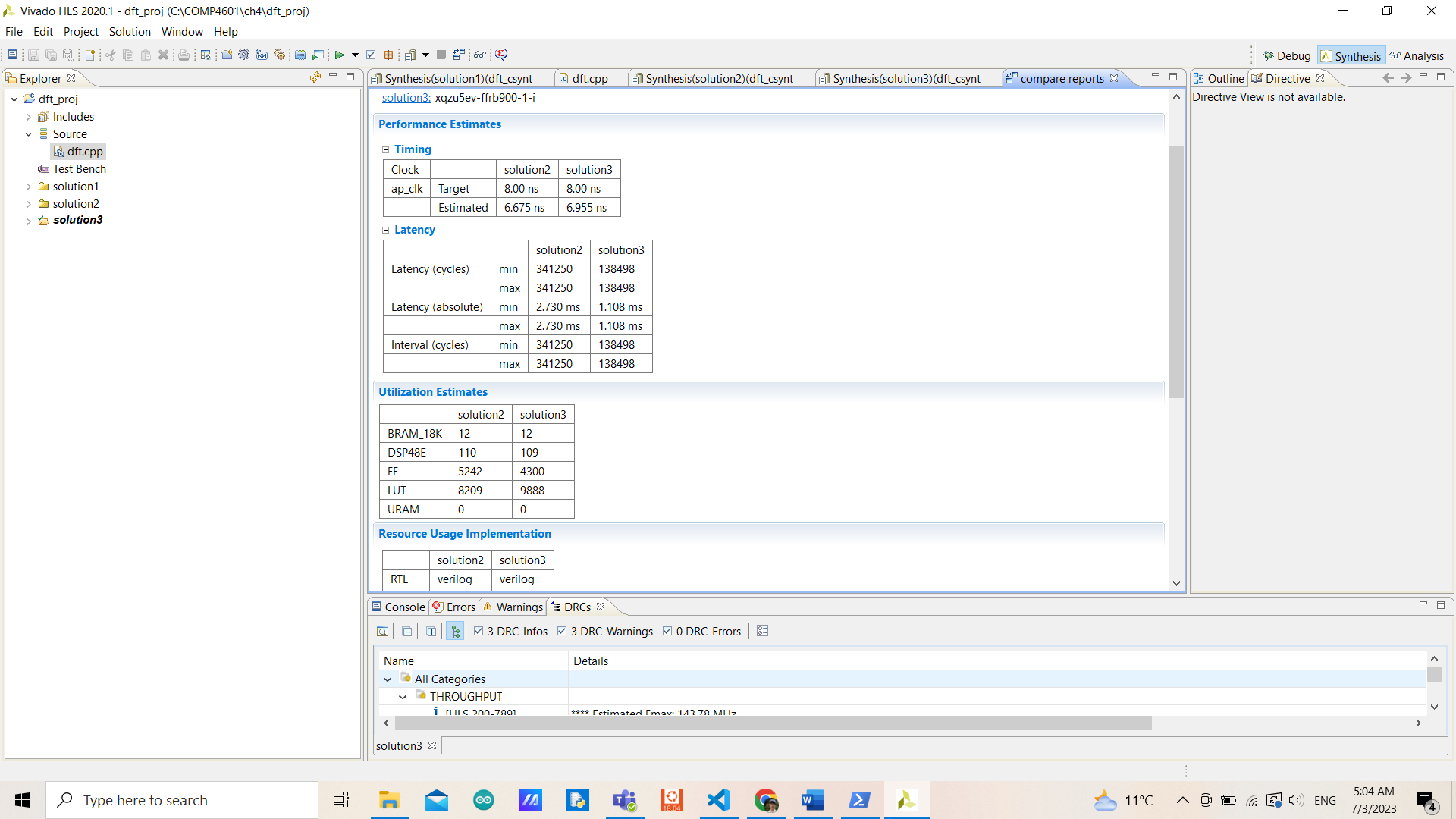
**DFT Exercise 2:**



After pipelining the inner loop, the number of cycles becomes stable with better performance (less estimated clock period as well). Not only performance but in terms of utilization, the new design requires fewer resources from every component.

By pipelining, even if it is impossible to achieve 1 initial interval (we got 5) with only 42 iteration latency, this is ~10 fewer iterations compared to not pipelining causing to have less total number of cycles of iteration. Since there are data dependencies in each floating-point addition (expensive) which takes ~5 cycles, it is necessary to wait for the result before moving on.

**DFT Exercise 3:**

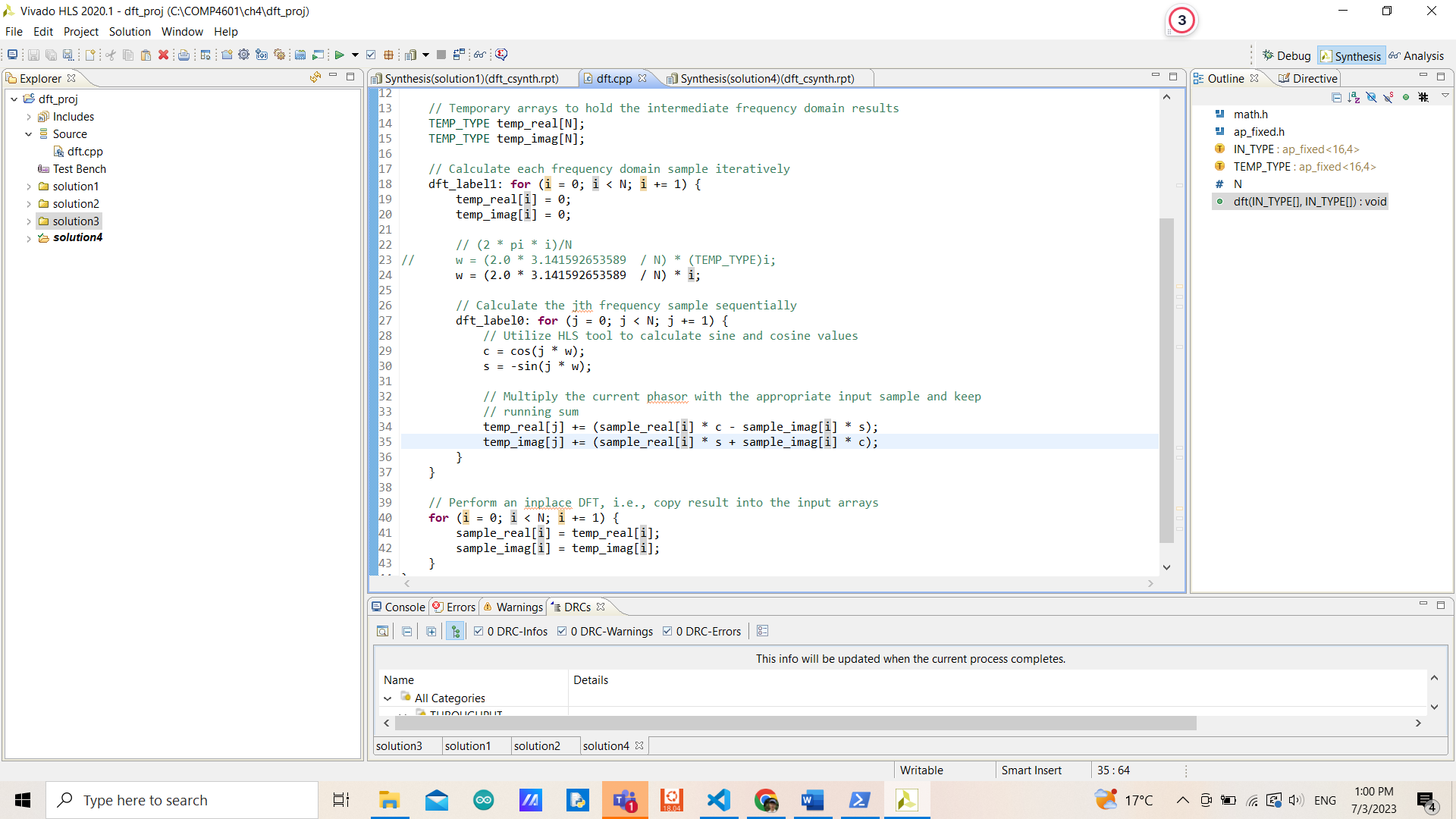


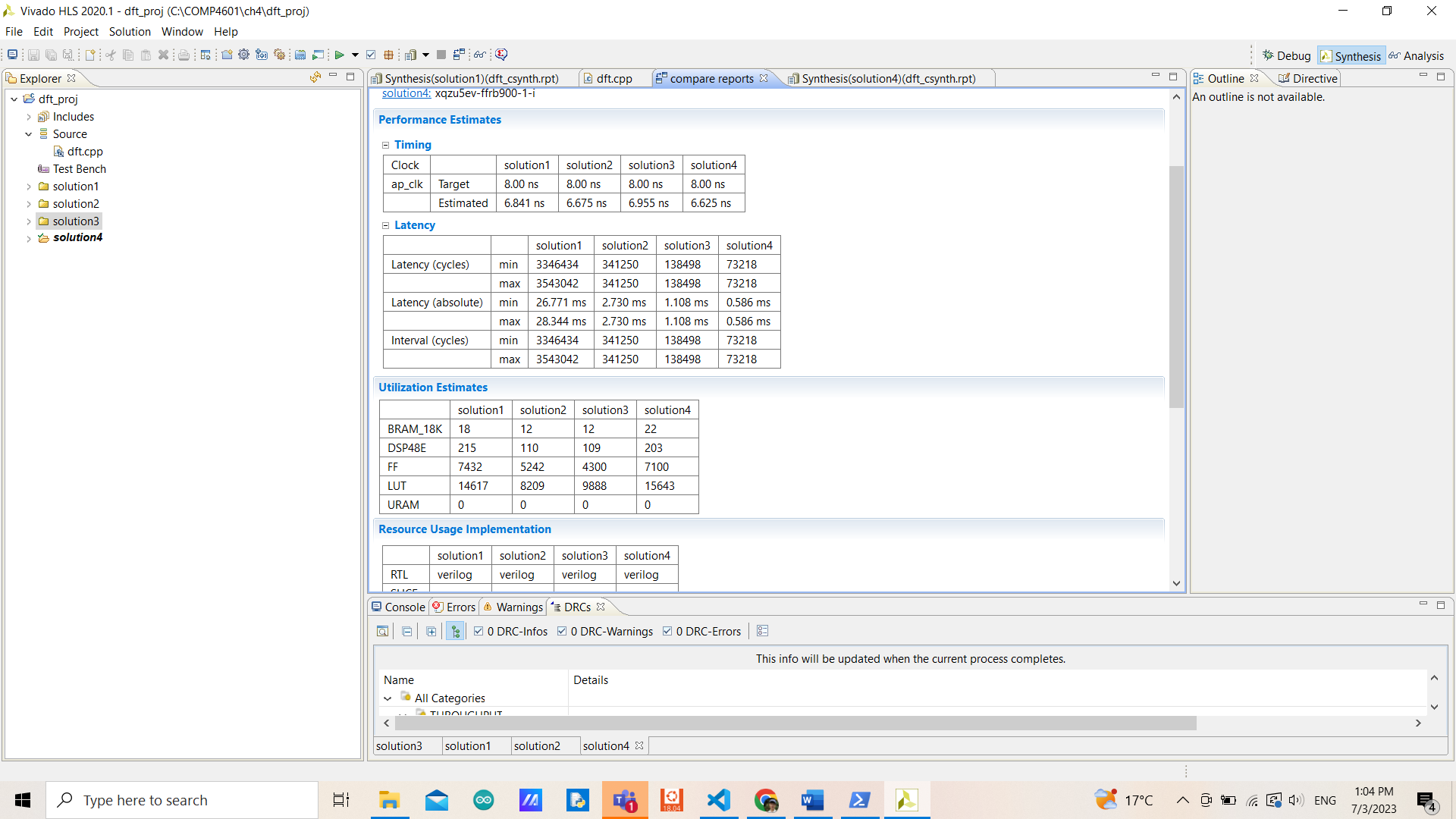
With a little more estimated clock period but significantly fewer cycles of latency, the new design is better in terms of performance with some trad-off on FFs and LUTs causing not much difference in terms of utilization. The reason behind this is that the new design has less initial interval (only 2) since floating-point addition is the main part that makes solution 2 has 5 II, and with fewer bits operation (far less expensive), we could achieve less initial interval.

We forgot to consider the impact of the output in terms of accuracy which is necessary when changing datatype.

We could write a testbench and compare the results from the original design with the new datatype design and see if the errors are minor or not.

**DFT Exercise 4:**





By swapping the inner loop and the outer loop, we achieved an initial interval of 1 resulting in a better performance but significantly increased number of resources required (almost double that of solution 3). The output shouldn’t be affected since the S matrix is diagonally symmetric.