ELEC 522 Project 3
Aedan Cullen

Description of C++ code and architecture constraints

My C++ code used in Vitis HLS, including the testbench described in the subsequent section, can be found in the "fpga_hls" directory. The "solution" directory generated by Vitis HLS is in the "Vitis_HLS" directory.

Overall, the design of my systolic array in this project follows the same principles used in my Project 2 design: input matrices are loaded at the top and left, and results are read out to the left along shared buses that are used in turn by all processing elements in the row. I have written my C++ code as two nested loops which perform all input, processing, and output steps using conditional statements within the inner loop; this means that the "dataflow" aspect of the design (input -> process -> output) is somewhat implicit in contrast to the Xilinx matrix multiplication example. I personally prefer this way of writing the code.

If this code is synthesized by Vitis HLS without any pragma directives present (the first variant I tried), the two loops will be run iteratively and a high latency of 31 cycles is obtained (Furthermore, a subsequent step in the matrix computation cannot be started until the 32nd cycle!) As expected, this architecture only uses one DSP. Its best clock period is comparable to that of my chosen design (described later).

Next, I tried inserting an HLS PIPELINE directive within the inner loop, with a desired initiation interval of 1 cycle. This still results in the use of only one DSP, but the individual conditional operations of loading and outputting data (within that inner loop) have now been parallelized so the initiation interval is decreased to 24 cycles.

A third point in the design space is achieved with the HLS PIPELINE II=1 directive outside the inner loop (the column loop) while inside the outer loop (the column loop). This results in utilization of 4 DSPs (one row's worth) with an initiation interval of 12. This design seems to provide a reasonable intermediate choice between the usage of 1 DSP and the usage of a full 16 DSPs.

As described at the bottom of this page, I chose to move the HLS PIPELINE directive outside both loops of my multiplication code, in order to use 16 multipliers and achieve an initiation interval of 1 cycle with a latency of 6 cycles.

Description of C++ testbench code

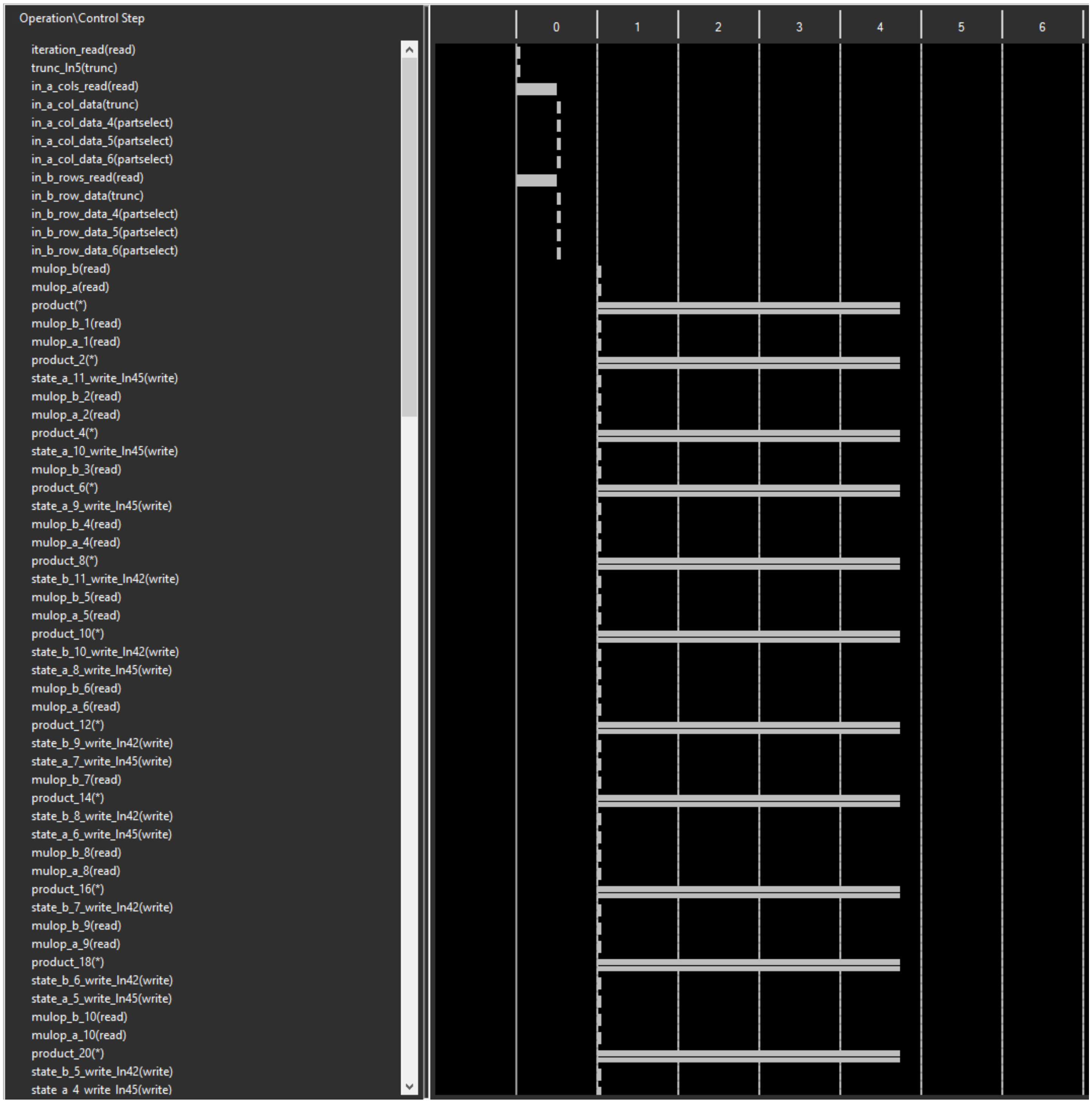
My testbench code starts with two plain C-style 2D arrays to encode the test matrices, and then constructs the necessary "skewed" matrix structures as it fills the HLS streams that feed the systolic array. It then runs the HLS C++ function (systolicmul) the appropriate number of times, and "de-skews" the output. Note that the output from the first clock cycles is discarded so that the testbench only begins collecting results when they begin appearing at the output. After filing another 2D array with the values from the output stream, each element in this result array is compared to the ground-truth multiplication result as computed by software.

Final choices of constraint configurations and advantages of final design

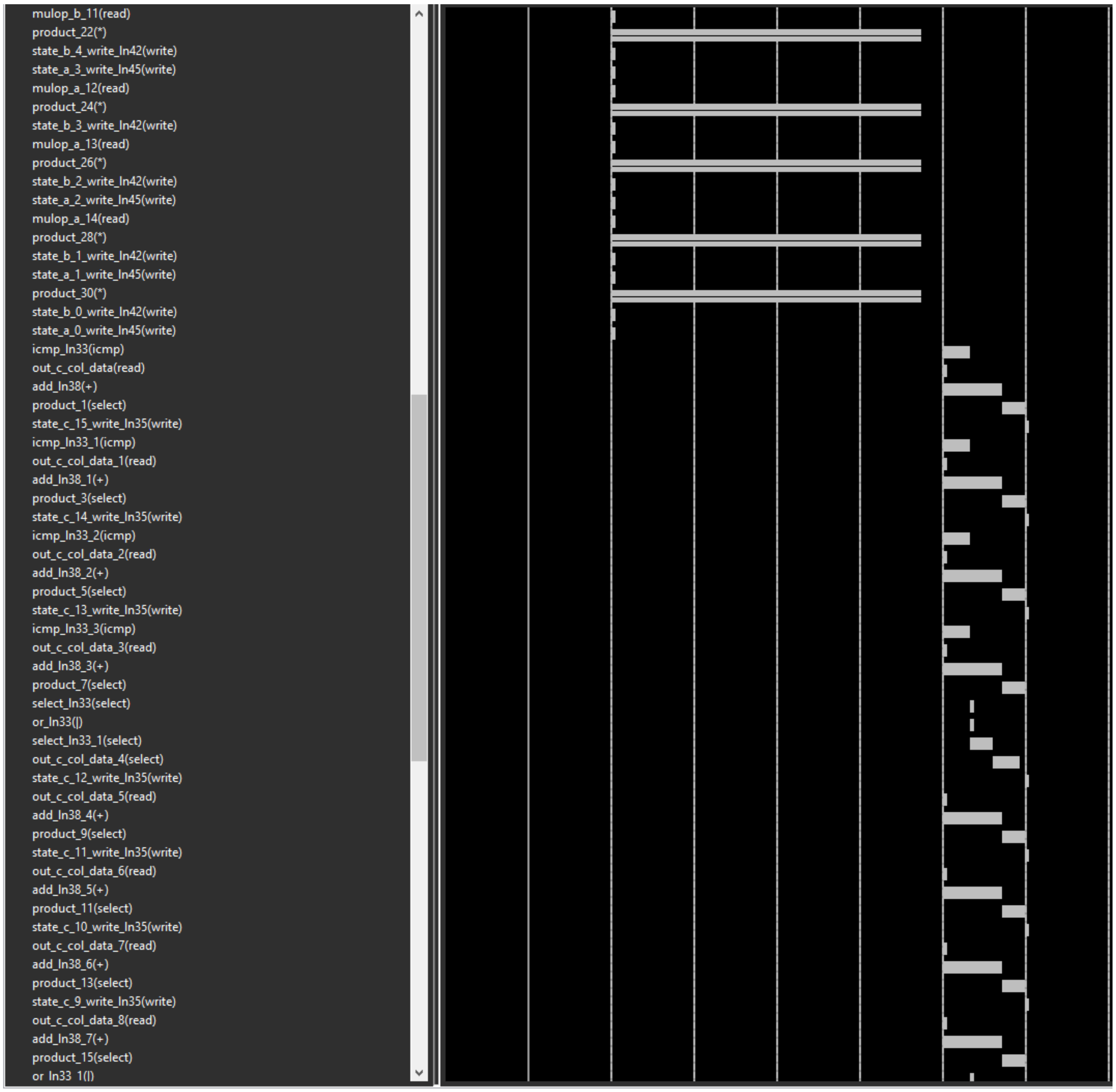
My final design uses a single HLS PIPELINE II=1 directive outside both nested loops, which serves to instruct Vitis HLS to unroll them both completely with an initiation interval of 1 clock cycle. This results in the use of 16 multipliers and accumulators that operate in parallel, achieving the best throughput possible. This design has the advantage that it matches the architecture of my previous Model Composer design from Project 2 (and, as we will see later in this report, Vitis HLS manages to achieve a slightly higher clock frequency and thus slightly higher overall throughput.) Vitis HLS decides to pipeline the multipliers with this directive (this automatic choice is mentioned in its log output). The disadvantage, of course, is the use of 16 multipliers. Fewer could be used by specifying the aforementioned alternative directives if area or power consumption was more important than absolute performance/throughput.

Vitis HLS scheduling results

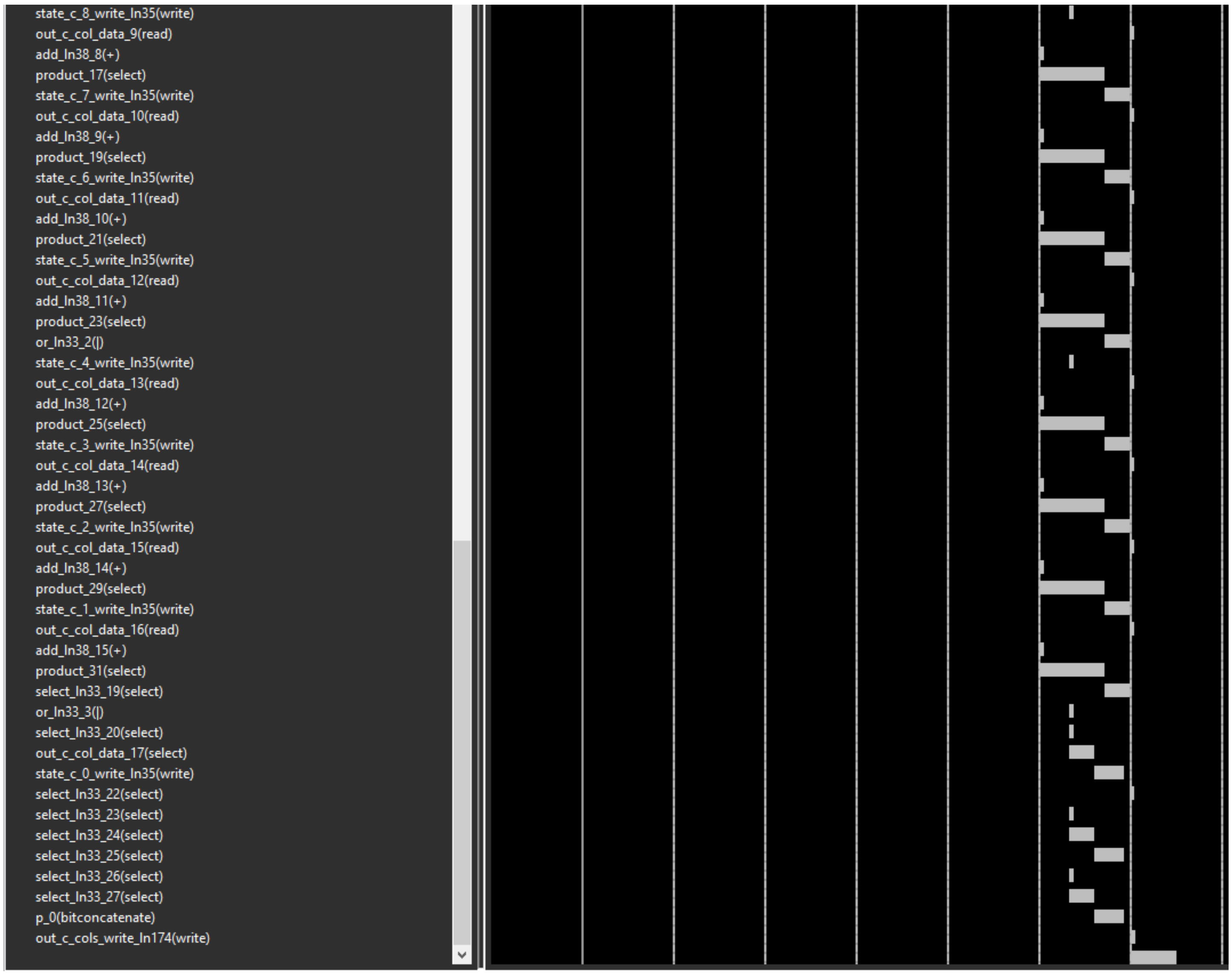
Images of the Gantt chart are shown below (continued on subsequent pages.) The 16 multipliers used in my chosen design are clearly visible, and as expected the multiplications are pipelined across 4 cycles by Vitis HLS for performance. Essentially clock cycle 5 is used for the accumulations, and cycles 0 and 6 are consumed by the operations of loading data from the stream into the edges of the matrix and storing data back into the stream. The operations called "select" that also take place in cycle 5 are related to the choice of output values from the proper element in each row that has completed its accumulation.



```
iteration_read(read)
trunc_ln5(trunc)
in_a_cols_read(read)
in_a_col_data(trunc)
in_a_col_data_4(partselect)
in_a_col_data_5(partselect)
in_a_col_data_6(partselect)
in_b_rows_read(read)
in_b_row_data(trunc)
in_b_row_data_4(partselect)
in_b_row_data_5(partselect)
in_b_row_data_6(partselect)
mulop_b(read)
mulop_a(read)
product(*)
mulop_b_1(read)
mulop_a_1(read)
product_2(*)
state_a_11_write_ln45(write)
mulop_b_2(read)
mulop_a_2(read)
product_4(*)
state_a_10_write_ln45(write)
mulop_b_3(read)
product_6(*)
state_a_9_write_ln45(write)
mulop_b_4(read)
mulop_a_4(read)
product_8(*)
state_b_11_write_ln42(write)
mulop_b_5(read)
mulop_a_5(read)
product_10(*)
state_b_10_write_ln42(write)
state_a_8_write_ln45(write)
mulop_b_6(read)
mulop_a_6(read)
product_12(*)
state_b_9_write_ln42(write)
state_a_7_write_ln45(write)
mulop_b_7(read)
product_14(*)
state_b_8_write_ln42(write)
state_a_6_write_ln45(write)
mulop_b_8(read)
mulop_a_8(read)
product_16(*)
state_b_7_write_ln42(write)
mulop_b_9(read)
mulop_a_9(read)
product_18(*)
state_b_6_write_ln42(write)
state_a_5_write_ln45(write)
mulop_b_10(read)
mulop_a_10(read)
product_20(*)
state_b_5_write_ln42(write)
state_a_4_write_ln45(write)
```

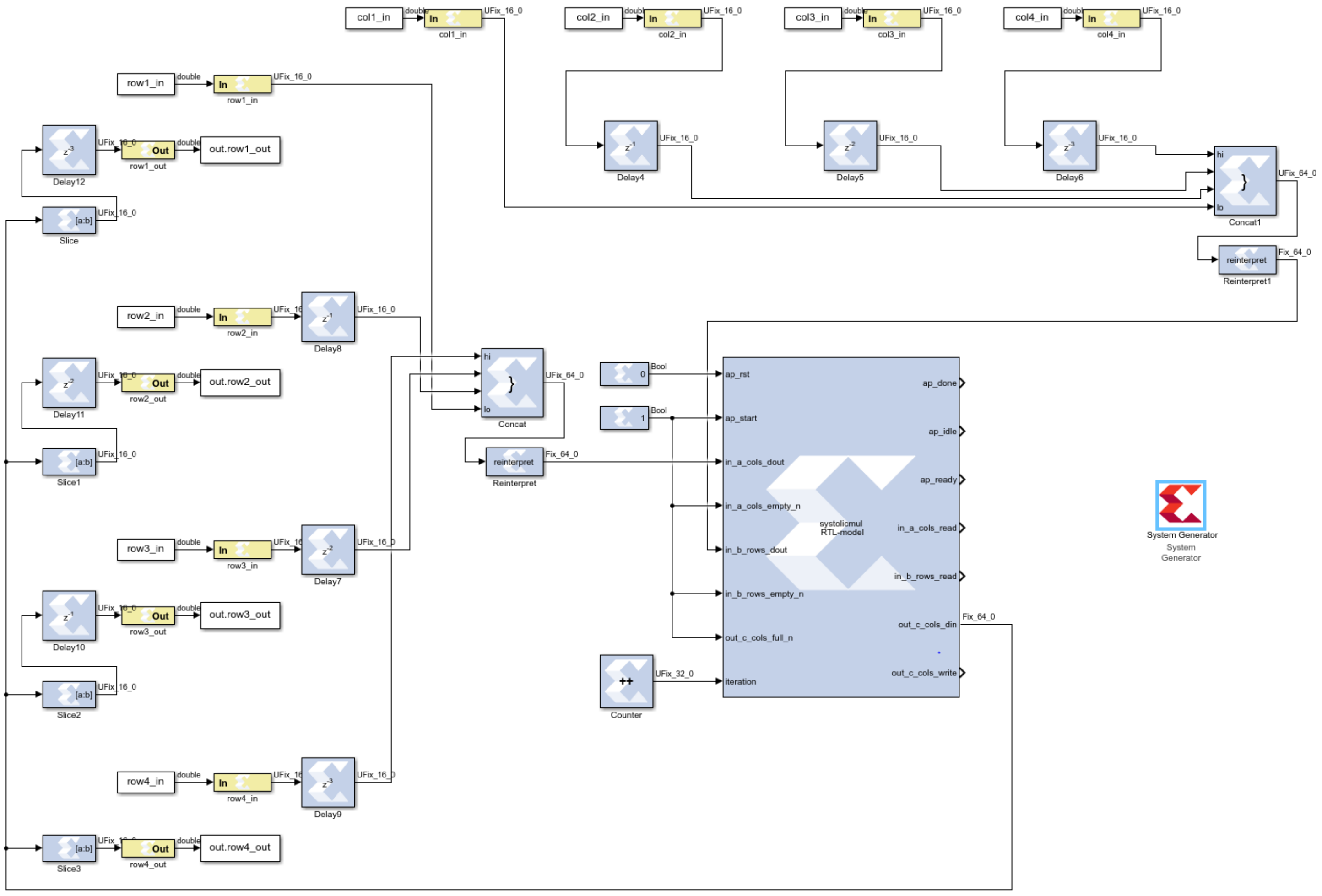


```
mulop_b_11(read)
product_22(*)
state_b_4_write_ln42(write)
state_a_3_write_ln45(write)
mulop_a_12(read)
product_24(*)
state_b_3_write_ln42(write)
mulop_a_13(read)
product_26(*)
state_b_2_write_ln42(write)
state_a_2_write_ln45(write)
mulop_a_14(read)
product_28(*)
state_b_1_write_ln42(write)
state_a_1_write_ln45(write)
product_30(*)
state_b_0_write_ln42(write)
state_a_0_write_ln45(write)
icmp_ln33(icmp)
out_c_col_data(read)
add_ln38(+)
product_1(select)
state_c_15_write_ln35(write)
icmp_ln33_1(icmp)
out_c_col_data_1(read)
add_ln38_1(+)
product_3(select)
state_c_14_write_ln35(write)
icmp_ln33_2(icmp)
out_c_col_data_2(read)
add_ln38_2(+)
product_5(select)
state_c_13_write_ln35(write)
icmp_ln33_3(icmp)
out_c_col_data_3(read)
add_ln38_3(+)
product_7(select)
select_ln33(select)
or_ln33(|)
select_ln33_1(select)
out_c_col_data_4(select)
state_c_12_write_ln35(write)
out_c_col_data_5(read)
add_ln38_4(+)
product_9(select)
state_c_11_write_ln35(write)
out_c_col_data_6(read)
add_ln38_5(+)
product_11(select)
state_c_10_write_ln35(write)
out_c_col_data_7(read)
add_ln38_6(+)
product_13(select)
state_c_9_write_ln35(write)
out_c_col_data_8(read)
add_ln38_7(+)
product_15(select)
or_ln33_1(|)
```

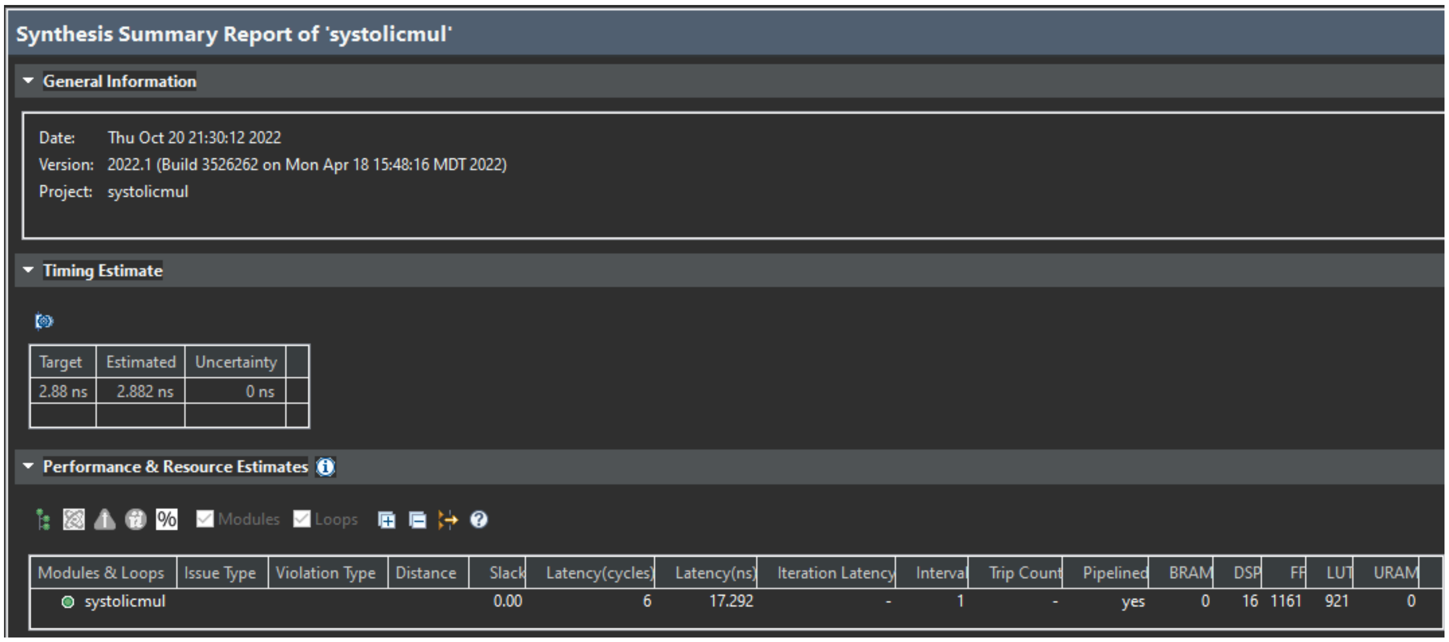


Model Composer HDL Netlist timing analysis

The Vitis HLS block is integrated in Model Composer as shown below. The input and output blocks used are essentially identical to those used in my Project 2 design, with the exception that Vitis HLS concatenates the four 16-bit-wide elements of each input and output stream into a single 64-bit-wide signal, which must be constructed/split using Concat and Slice blocks for compatibility with the individual timeseries that my MATLAB test code expects. As in Project 2, delay blocks are used in hardware to provide the "skewing" of the matrices. This can of course be performed in software just as easily by removing any additional delays in hardware; the C++ testbench of my Vitis HLS code serves as an example of this.



In Vitis HLS, the estimated shortest possible clock period was 2.882ns (347MHz):



Synthesis Summary Report of 'systolicmul'

General Information

Date:	Thu Oct 20 21:30:12 2022
Version:	2022.1 (Build 3526262 on Mon Apr 18 15:48:16 MDT 2022)
Project:	systolicmul

Timing Estimate

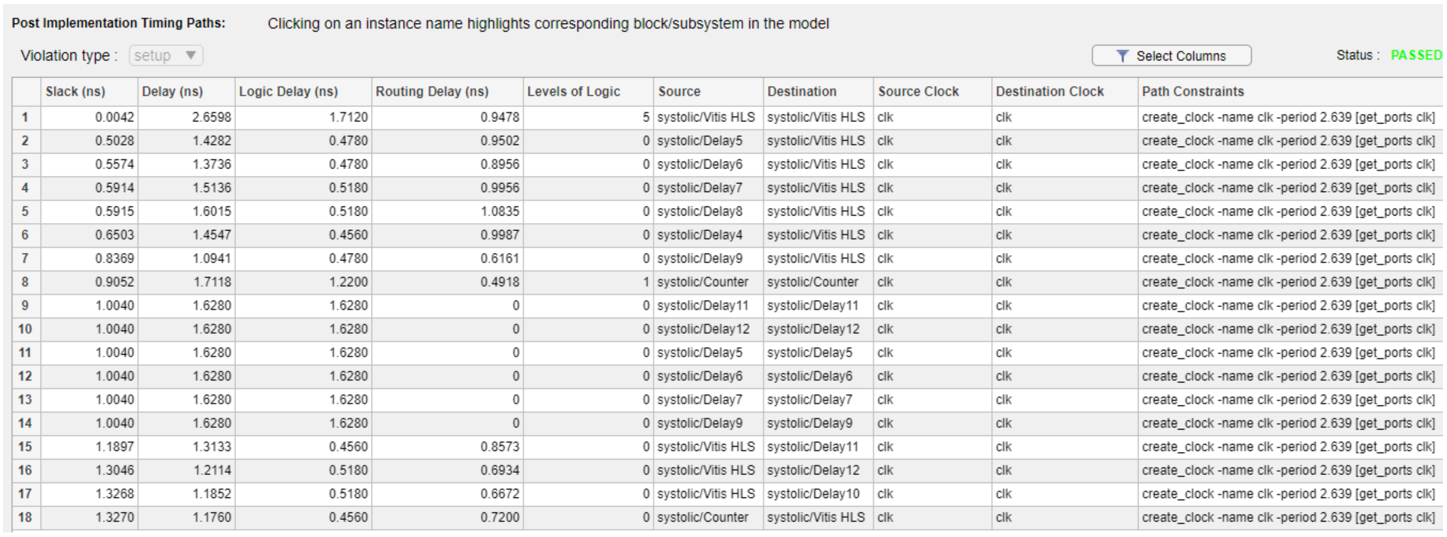
Target	Estimated	Uncertainty
2.88 ns	2.882 ns	0 ns

Performance & Resource Estimates

Modules   Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
● systolicmul				0.00	6	17.292	-	1	-	yes	0	16	1161	921	0

Using post-implementation timing analysis in Model Composer, the shortest possible clock period is 2.639ns (379MHz):



Post Implementation Timing Paths: Clicking on an instance name highlights corresponding block/subsystem in the model

Violation type : setup

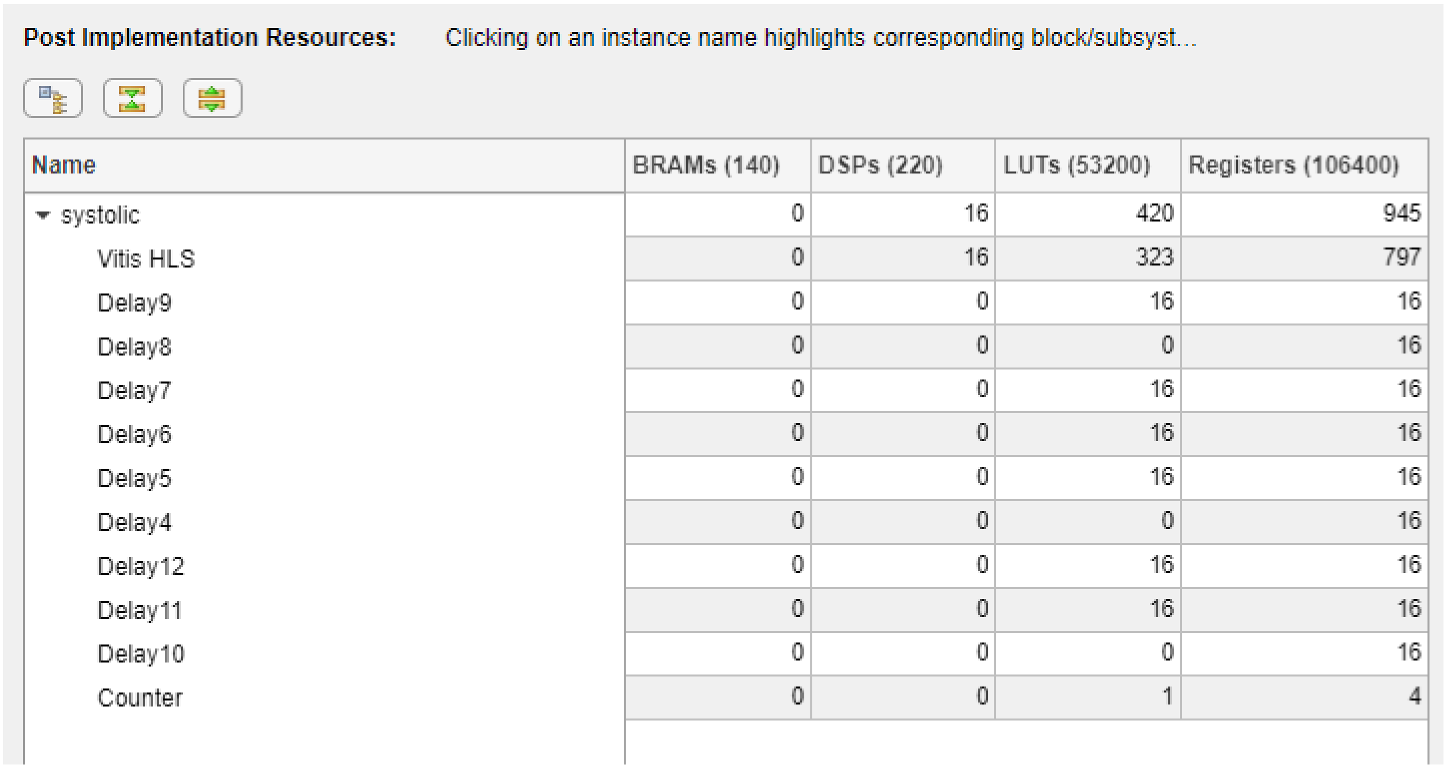
Select Columns

Status : PASSED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock	Path Constraints
1	0.0042	2.6598	1.7120	0.9478	5	systolic/Vitis HLS	systolic/Vitis HLS	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
2	0.5028	1.4282	0.4780	0.9502	0	systolic/Delay5	systolic/Vitis HLS	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
3	0.5574	1.3736	0.4780	0.8956	0	systolic/Delay6	systolic/Vitis HLS	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
4	0.5914	1.5136	0.5180	0.9956	0	systolic/Delay7	systolic/Vitis HLS	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
5	0.5915	1.6015	0.5180	1.0835	0	systolic/Delay8	systolic/Vitis HLS	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
6	0.6503	1.4547	0.4560	0.9987	0	systolic/Delay4	systolic/Vitis HLS	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
7	0.8369	1.0941	0.4780	0.6161	0	systolic/Delay9	systolic/Vitis HLS	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
8	0.9052	1.7118	1.2200	0.4918	1	systolic/Counter	systolic/Counter	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
9	1.0040	1.6280	1.6280	0	0	systolic/Delay11	systolic/Delay11	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
10	1.0040	1.6280	1.6280	0	0	systolic/Delay12	systolic/Delay12	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
11	1.0040	1.6280	1.6280	0	0	systolic/Delay5	systolic/Delay5	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
12	1.0040	1.6280	1.6280	0	0	systolic/Delay6	systolic/Delay6	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
13	1.0040	1.6280	1.6280	0	0	systolic/Delay7	systolic/Delay7	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
14	1.0040	1.6280	1.6280	0	0	systolic/Delay9	systolic/Delay9	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
15	1.1897	1.3133	0.4560	0.8573	0	systolic/Vitis HLS	systolic/Delay11	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
16	1.3046	1.2114	0.5180	0.6934	0	systolic/Vitis HLS	systolic/Delay12	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
17	1.3268	1.1852	0.5180	0.6672	0	systolic/Vitis HLS	systolic/Delay10	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]
18	1.3270	1.1760	0.4560	0.7200	0	systolic/Counter	systolic/Vitis HLS	clk	clk	create_clock -name clk -period 2.639 [get_ports clk]

Model Composer HDL Netlist resource utilization

Post-implementation resource utilization from Model Composer is shown below. As expected 16 DSPs are used. A surprisingly small number of lookup tables are used. by the Vitis HLS block. The number of registers needed is also significantly lower than the initial estimate given within Vitis HLS.



Post Implementation Resources: Clicking on an instance name highlights corresponding block/subsyst...

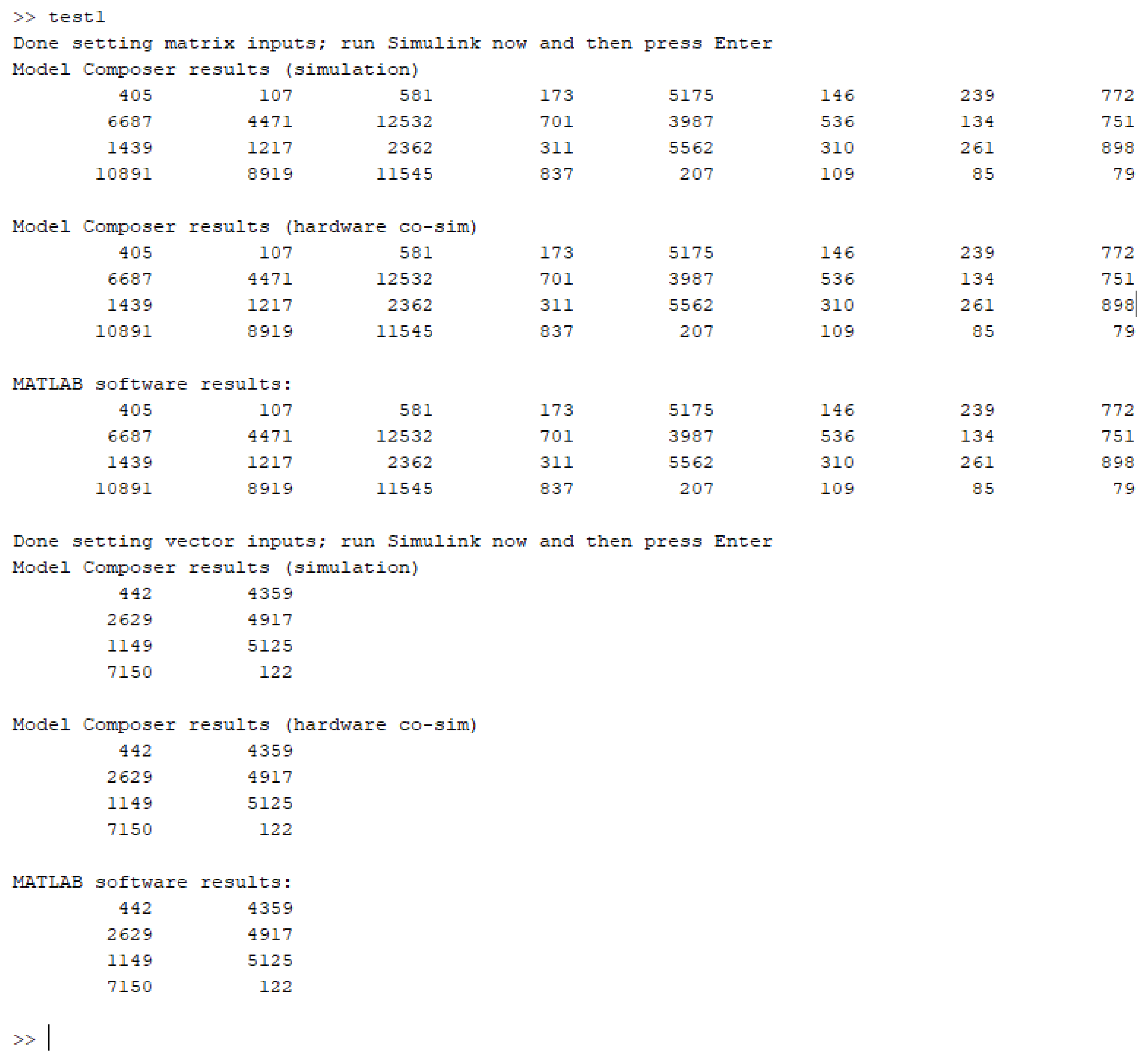
Name	BRAMs (140)	DSPs (220)	LUTs (53200)	Registers (106400)
▾ systolic	0	16	420	945
Vitis HLS	0	16	323	797
Delay9	0	0	16	16
Delay8	0	0	0	16
Delay7	0	0	16	16
Delay6	0	0	16	16
Delay5	0	0	16	16
Delay4	0	0	0	16
Delay12	0	0	16	16
Delay11	0	0	16	16
Delay10	0	0	0	16
Counter	0	0	1	4

Information on throughput

Like my Project 2 design, this HLS design is able to begin a subsequent matrix multiplication on the clock cycle immediately following the last cycle of the first multiplication. No extra time is needed for configuring or loading/unloading the matrix, as the unloading is performed continuously. In other words, both input and output matrices are packed together side-by-side and can be fed into the array in a fully pipelined manner. The MATLAB *test1.m* script serves as an example of how this is done for a series of matrix multiplications. Once all pipeline stages are performing computations, only four cycles are required per multiplication overall.

Testing of the Vitis HLS block in Model Composer

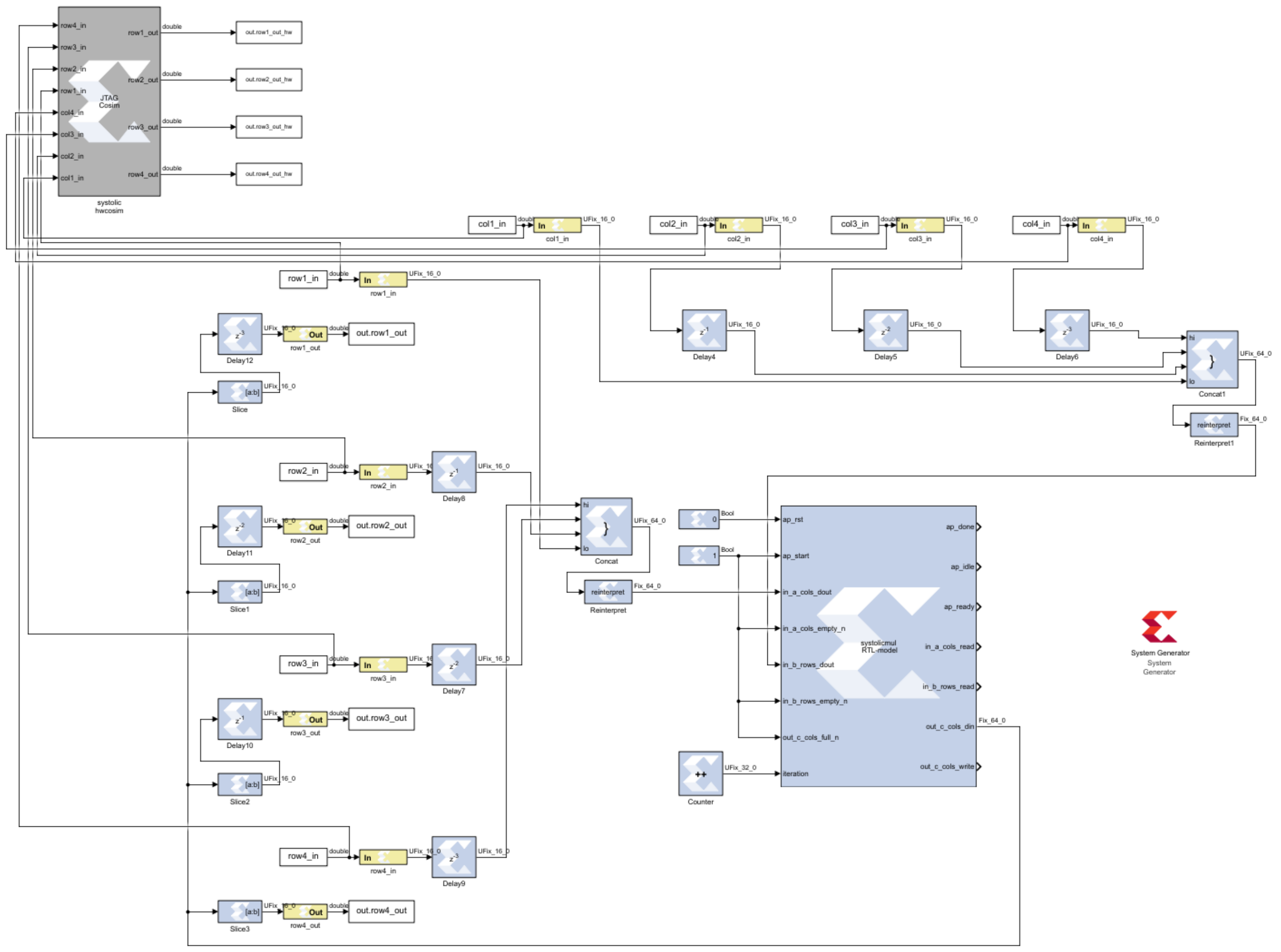
The test scripts used for my Project 2 submission have been adapted for use with the Vitis HLS block in Model Composer by altering the timing at which the results are expected to accommodate the additional latency of this implementation. The script *test1.m* expects to use the *systolic_hwcosim.slx* file, and runs both simulation and hardware co-sim and compares the results to the correct computed results from MATLAB. *test1.m* uses the other scripts starting the *din* and *dout* in order to load and retrieve data. Two pipelined matrix-matrix multiplications are being performed (and subsequently two matrix-vector multiplications) with the result data displayed in a horizontally-concatenated form. The first result is *mat1*mat3*, and the second result is *mat2*mat4.* The vector results are *mat1*vec1* and *mat2*vec2*. The *simulation_results.txt* file provided contains this expected output.



```
>> test1
Done setting matrix inputs; run Simulink now and then press Enter
Model Composer results (simulation)
         405          107          581          173         5175          146          239          772
        6687         4471        12532          701         3987          536          134          751
        1439         1217         2362          311         5562          310          261          898
       10891         8919        11545          837          207          109           85           79

Model Composer results (hardware co-sim)
         405          107          581          173         5175          146          239          772
        6687         4471        12532          701         3987          536          134          751
        1439         1217         2362          311         5562          310          261          898
       10891         8919        11545          837          207          109           85           79

MATLAB software results:
         405          107          581          173         5175          146          239          772
        6687         4471        12532          701         3987          536          134          751
        1439         1217         2362          311         5562          310          261          898
       10891         8919        11545          837          207          109           85           79

Done setting vector inputs; run Simulink now and then press Enter
Model Composer results (simulation)
         442         4359
        2629         4917
        1149         5125
        7150          122

Model Composer results (hardware co-sim)
         442         4359
        2629         4917
        1149         5125
        7150          122

MATLAB software results:
         442         4359
        2629         4917
        1149         5125
        7150          122

>>
```

Hardware-in-the-loop co-simulation results

A screenshot of the hardware co-sim configuration in Model Composer is shown below. The numerical results of the hardware co-simulation have been verified in the section above.



Comparison with Project 2, and discussion on latency

In Project 2, 16 DSPs were used, 495 LUTs, and 832 registers. The achievable clock period was 2.75ns (364MHz). With Vitis HLS, I am still using 16 DSPs, but the number of LUTs has fallen to 420. The number of registers has risen to 935. The achievable clock period has improved to 2.639ns (379MHz). Overall, I think this is an impressive result, considering that an engineer might consider it much easier to write the C++ code for this project than to construct the systolic array by hand in Project 2. It is likely that the more sophisticated scheduling in Vitis HLS has allowed more operations to be packed within each clock cycle (as is evident in the Gantt chart above). However, I am not convinced that there is any advantage to using 4 clock cycles for the multiplications. Manually instantiating multipliers in Model Composer gives the designer complete control over their pipelining, and by manually adjusting in during Project 2 I had determined that there was no advantage to increasing the latency of the multipliers beyond two cycles for 16-bit-wide operands (though there was certainly a disadvantage to decreasing it to 1 cycle).

The latency of this design is given by Vitis HLS as 6 cycles. However, it is important to note that Vitis is not considering the fact that we must run the "systolicmul" function multiple times (2*SIZE-1, where SIZE is 4 for a 4x4 matrix) to perform a complete matrix multiplication. It is more helpful to consider the extra clock cycles that this design adds in comparison to my Project 2 design: rather than using two cycles for the multiply, one for the accumulation, and outputting a result on the next cycle, the HLS design uses one cycle to load inputs, four for the multiply, one for the accumulation, one to place the result in the output stream, and the result is then ready on the following cycle. Four additional cycles have been added on top of what I had in Project 2, and as such the array indexing notation in the *dout_matmat.m* file has changed from starting at the 10th index of the output timeseries to the 14th index. (Of course analogous changes are made in the other *dout* scripts.)