ELEC 522 Project 2
Aedan Cullen

Description of design approach

I began my design with the goal to achieve nicely pipelined matrix multiplications, without the need for extra clock cycles between matrices for configuring or unloading the array. My final design can complete a 4x4 matrix multiplication every four clock cycles; the first column of each subsequent 4x4 multiplication is shifted out of the array on the very next cycle after the prior 4x4 multiplication has completed. This means that full utilization of the multipliers and accumulators of the processing elements (PEs) can be achieved for large amounts of input data, with each PE performing a multiply and accumulate (pipelined) every cycle. My design uses 16-bit two's complement integers.

Shifting data into the array is simple: row and column data is moved systolically from the top and left edges of the array. The proper skewed alignment is achieved by adding registers to delay each column and row after the leading column/row. Data is read out of the array using what I and other students have called the "row bus" design, where there is a single output-data bus for each row of the array and each PE asserts its result onto this bus in turn when its accumulation is complete. This means that the readout is not truly systolic: the results from PEs on the right side of each row travel across all PEs in the row to their left within one clock cycle. The reasons for this design choice will be discussed in the following sections.

Matrix-matrix mode uses all four columns of the array, while matrix-vector mode only requires that data is loaded into one column. No special logic is needed to change the behavior of the array, since I have already designed it to require exactly four clock cycles per matrix/vector. Vectors can be either passed sequentially to the first column and each will be completed within four clock cycles, or they can be "stacked" into the other columns and treated as a matrix. My MATLAB test code uses only the first column of PEs for matrix-vector operations. A systolic "cen" (column enable) signal has been included in the design - though always set to 1 right now, this signal could be used for clock gating of the multipliers and accumulators on unused columns to reduce power consumption when performing matrix-vector operations.

Tradeoffs on input and output architecture

If we want to achieve a complete matrix multiplication every four clock cycles without spending extra cycles unloading the matrix to the left, four elements of the result matrix must be unloaded every clock cycle, all the time. If we were to register the result of each neighboring PE to the right, so that each result shifts toward the left of the array by only one PE every cycle, we would not be able to finish shifting out the result of the previous multiplication before the first result of the subsequent multiplication is ready at the first PE of the row.

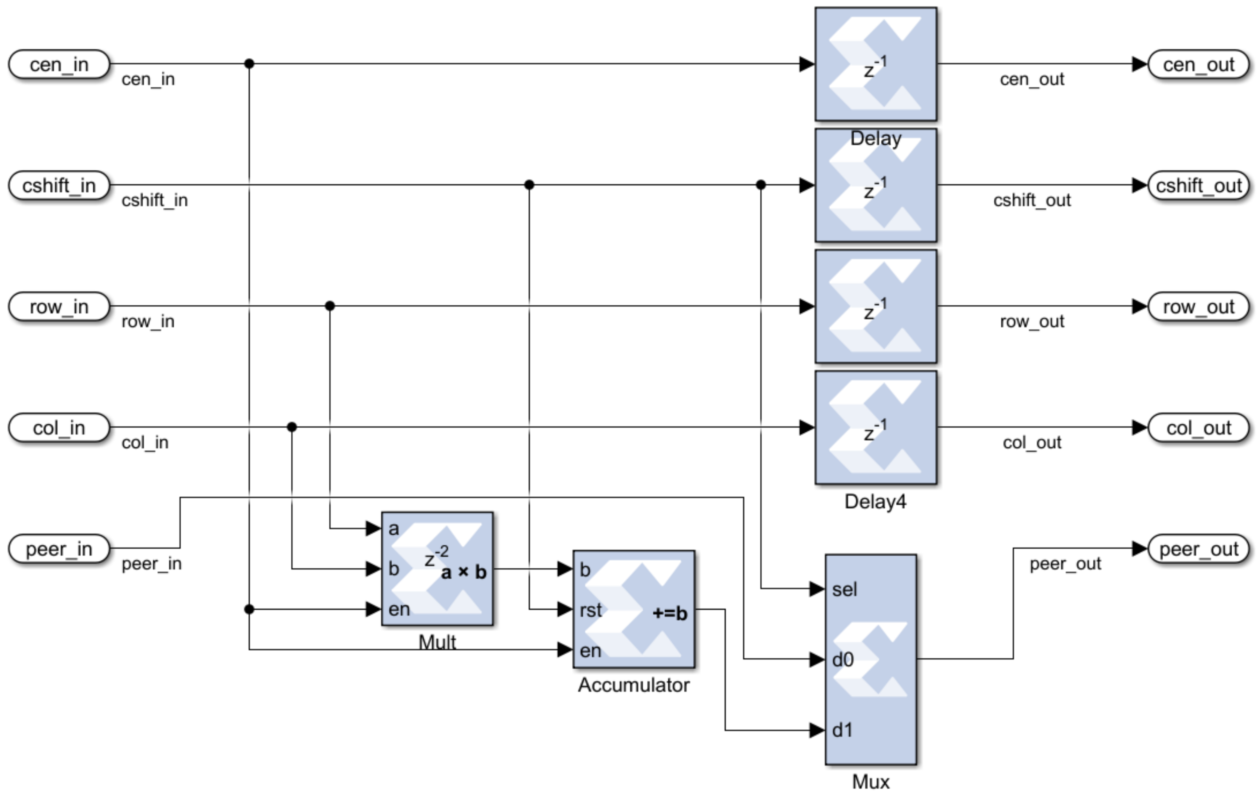
It would be possible to add multiple sets of output registers and additional control logic to "buffer" each row's results within the elements for four clock cycles before beginning the shift-out. In that design, results would wait in a "primary" result register within each PE rather than being shifted out immediately. After all PEs in a row have produced results, the results would be moved to a set of "alternate" result registers within the PEs and then shifted across these during the same time as a new set of four results fills the "primary" result registers. However, this design would still require a control signal spanning the entire row in order to coordinate this primary/secondary result transfer, and it increases the result latency by the width of the matrix.

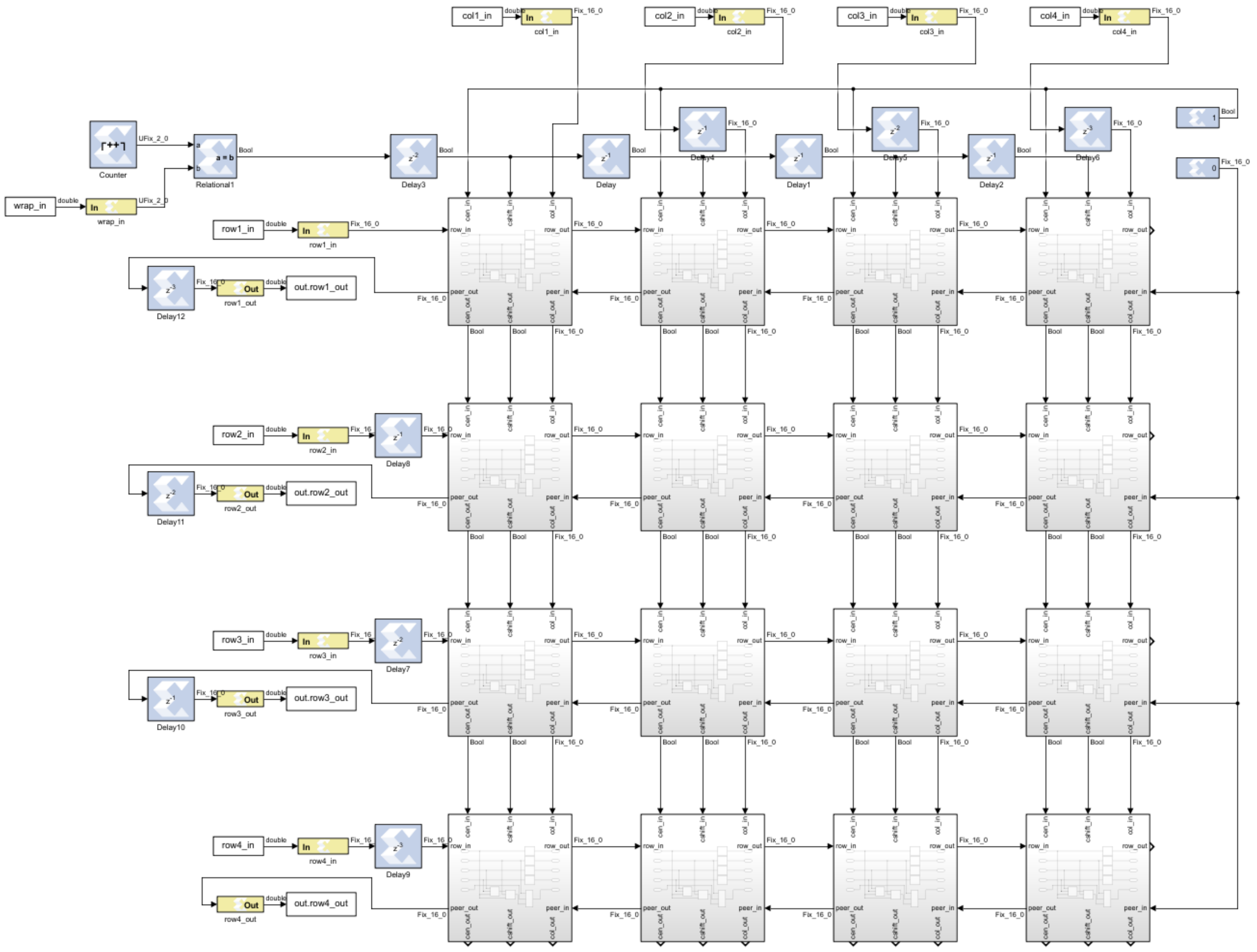
I chose the simpler approach of reading results out immediately when they become available, using a common output data bus across the entire row. This requires output data to traverse an entire row of the matrix within one clock cycle, but it maintains the lowest possible latency for the overall 4x4 multiply and requires only simple control logic for coordinating the readout. In a real implementation, tri-state buffers might be used to prevent the propagation delay incurred by feeding this "row bus" through a multiplexer at each PE.

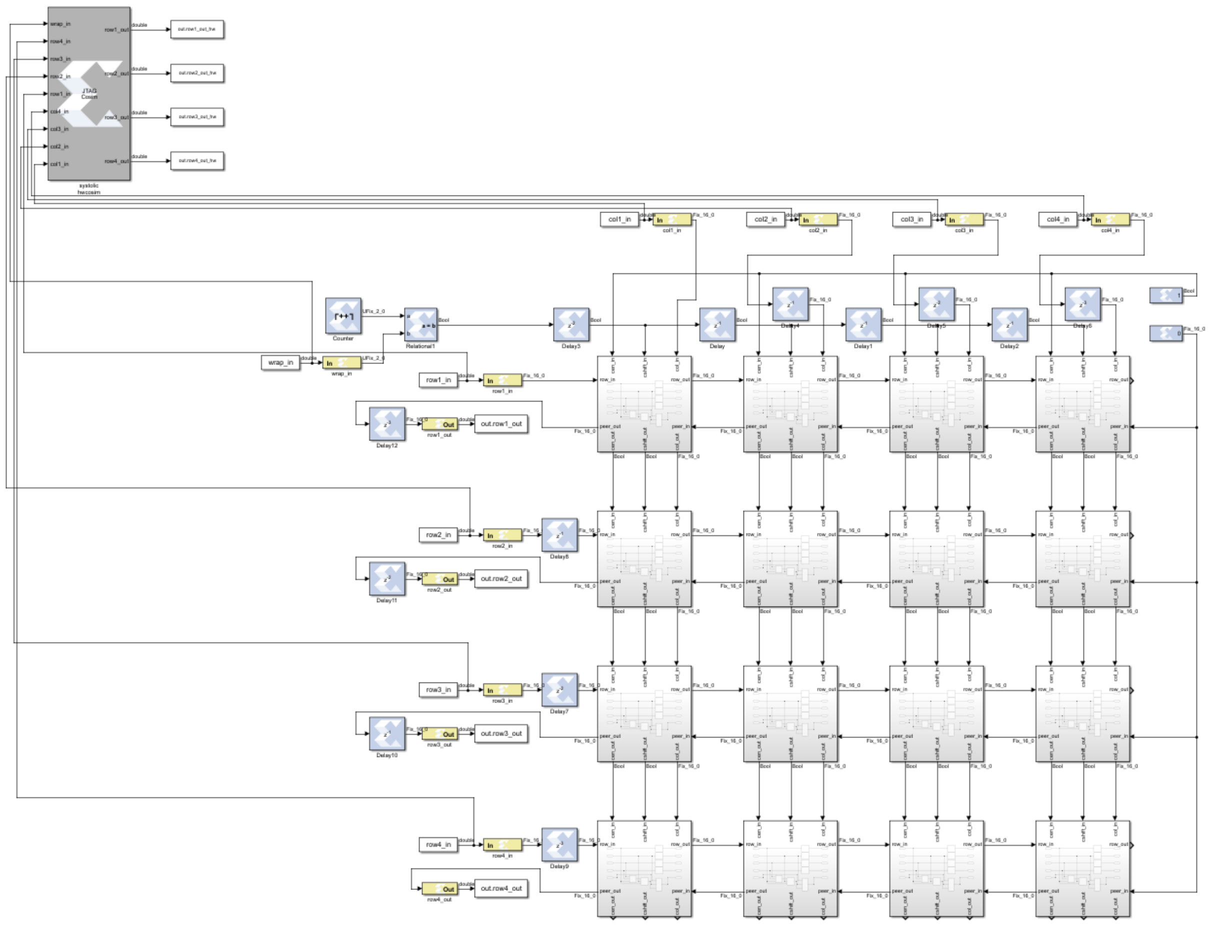
The signal controlling readout is called "cshift" in my design, and is passed systolically down the rows of the array. A counter and comparator at the upper-left corner of the array times this signal correctly for the number of accumulations that must take place. The point at which the counter resets can be configured by an external signal so that a smaller square region of the matrix (e.g. 2x2) can be used if desired.

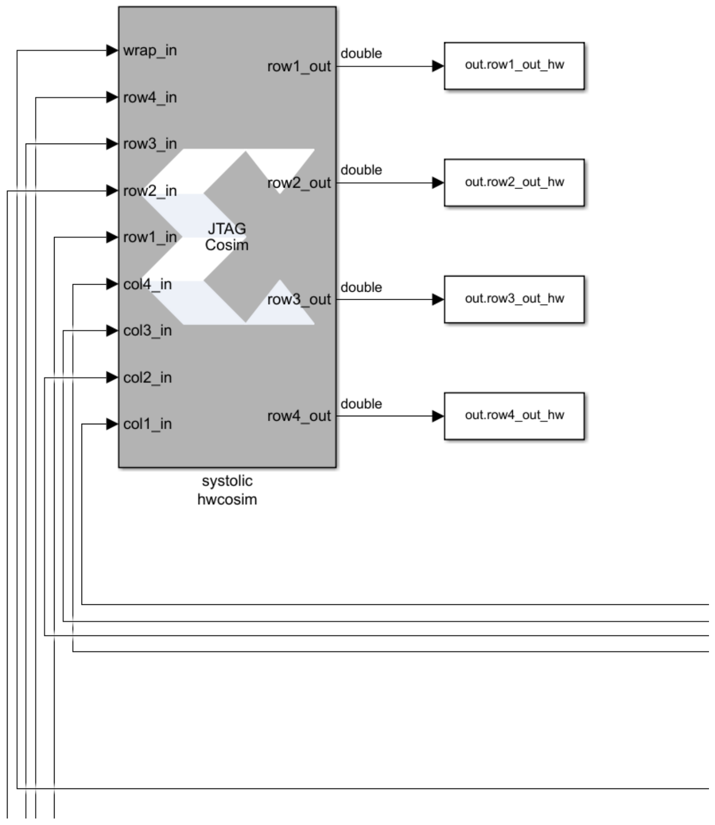
Model Composer files

Screenshots of the processing element, full array, and hardware co-simulation setup for my design are shown below. Simulink files are included with this submission. The processing element itself is contained in the *pelib.slx* file, which is included as a custom library in the *systolic.slx* and *systolic_hwcosim.slx* files.



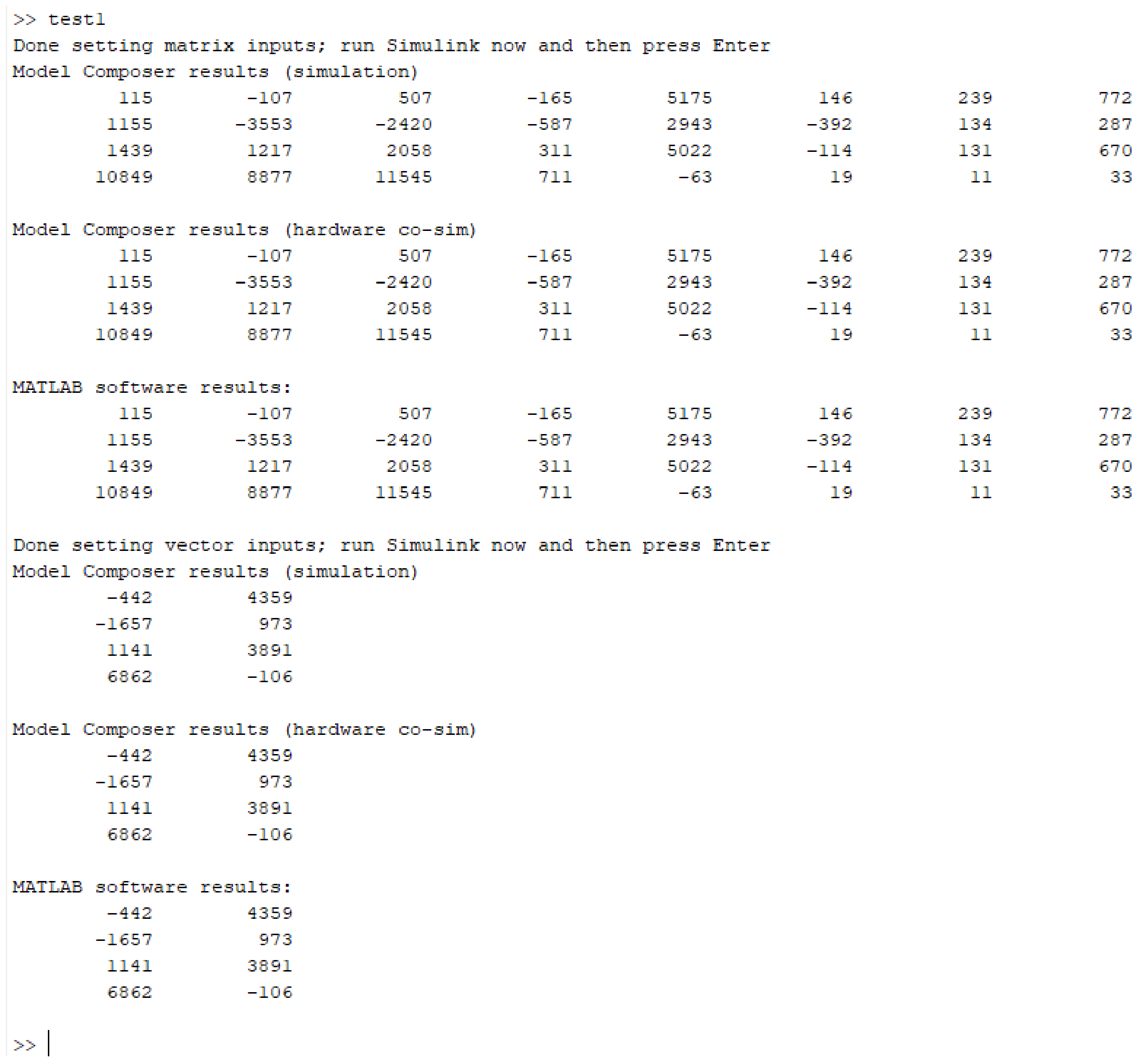

col1_in
col2_in
col3_in
col4_in
Counter
Relational1
a = b
wrap_in
Delay3
Delay
Delay1
Delay2
Delay4
Delay5
Delay6
row1_in
row2_in
row3_in
row4_in
row1_out
row2_out
row3_out
row4_out
out.row1_out
out.row2_out
out.row3_out
out.row4_out
Delay7
Delay8
Delay9
Delay10
Delay11
Delay12
In
Out
double
Fix_16_0
UFix_2_0
Bool
cen_in
cshift_in
col_in
row_in
row_out
peer_out
peer_in
cen_out
cshift_out
col_out
1
0


JTAG Cosim
systolic hwcosim
out.row1_out_hw
out.row2_out_hw
out.row3_out_hw
out.row4_out_hw
col1_in
col2_in
col3_in
col4_in
wrap_in
Counter
Relational1
Delay3
Delay
Delay1
Delay2
Delay4
Delay5
Delay6
row1_in
row2_in
row3_in
row4_in
out.row1_out
out.row2_out
out.row3_out
out.row4_out
Delay12
Delay11
Delay10
Delay8
Delay7
Delay9
Fix_16_0
Bool
double

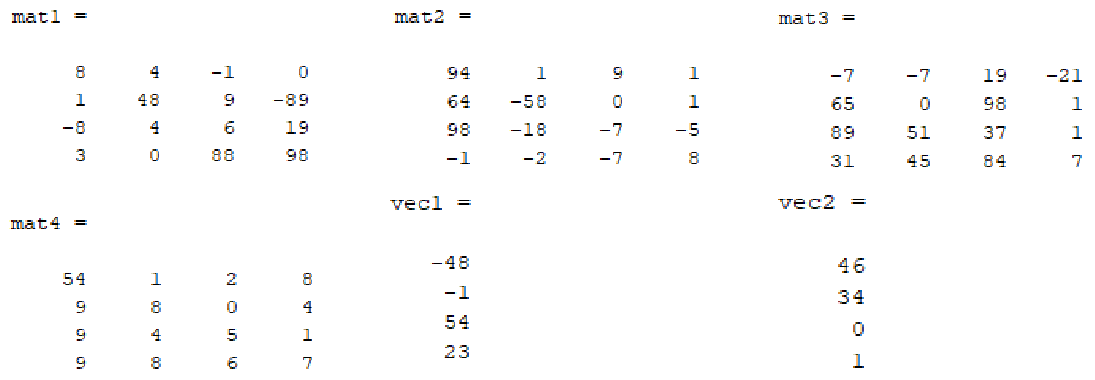

wrap_in
row4_in
row3_in
row2_in
row1_in
col4_in
col3_in
col2_in
col1_in
JTAG
Cosim
row1_out
row2_out
row3_out
row4_out
double
double
double
double
out.row1_out_hw
out.row2_out_hw
out.row3_out_hw
out.row4_out_hw
systolic
hwcosim

Simulation and hardware co-sim results

Screenshots of the output of my MATLAB testing script *test1.m* are shown below. The specific behavior of this script is described in the "Testing methodology" section. *test1.m* exercises Model Composer simulation, hardware co-sim, and displays the correct computed result from MATLAB for comparison. 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*.



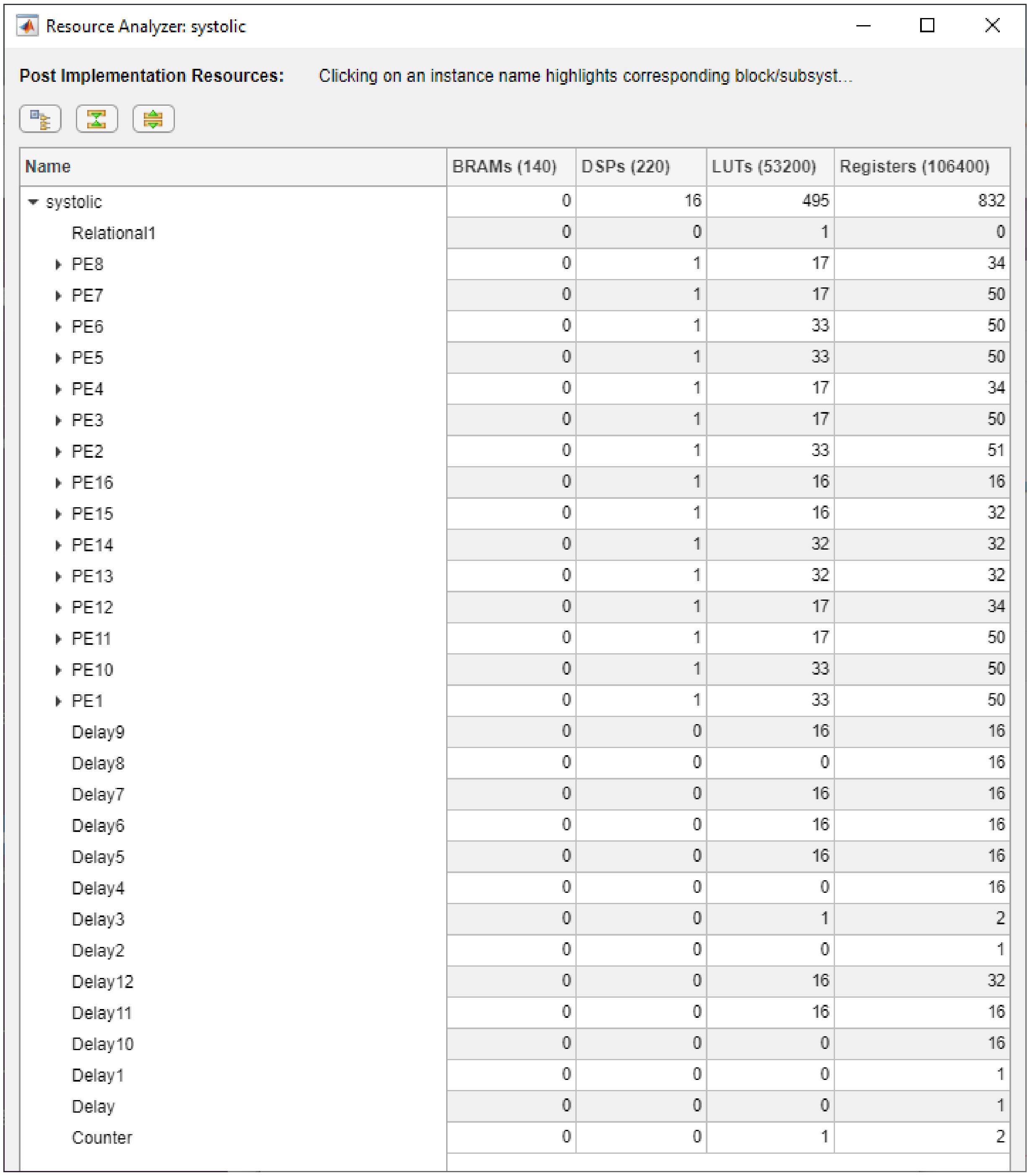
```
>> test1
Done setting matrix inputs; run Simulink now and then press Enter
Model Composer results (simulation)
         115        -107         507        -165        5175         146         239         772
        1155       -3553       -2420        -587        2943        -392         134         287
        1439        1217        2058         311        5022        -114         131         670
       10849        8877       11545         711         -63          19          11          33

Model Composer results (hardware co-sim)
         115        -107         507        -165        5175         146         239         772
        1155       -3553       -2420        -587        2943        -392         134         287
        1439        1217        2058         311        5022        -114         131         670
       10849        8877       11545         711         -63          19          11          33

MATLAB software results:
         115        -107         507        -165        5175         146         239         772
        1155       -3553       -2420        -587        2943        -392         134         287
        1439        1217        2058         311        5022        -114         131         670
       10849        8877       11545         711         -63          19          11          33

Done setting vector inputs; run Simulink now and then press Enter
Model Composer results (simulation)
        -442        4359
       -1657         973
        1141        3891
        6862        -106

Model Composer results (hardware co-sim)
        -442        4359
       -1657         973
        1141        3891
        6862        -106

MATLAB software results:
        -442        4359
       -1657         973
        1141        3891
        6862        -106

>> |
```



```
mat1 =

     8     4    -1     0
     1    48     9   -89
    -8     4     6    19
     3     0    88    98

mat2 =

    94     1     9     1
    64   -58     0     1
    98   -18    -7    -5
    -1    -2    -7     8

mat3 =

    -7    -7    19   -21
    65     0    98     1
    89    51    37     1
    31    45    84     7

mat4 =

    54     1     2     8
     9     8     0     4
     9     4     5     1
     9     8     6     7

vec1 =

   -48
    -1
    54
    23

vec2 =

    46
    34
     0
     1
```

Resource utilization

Full resource utilization information is shown below.



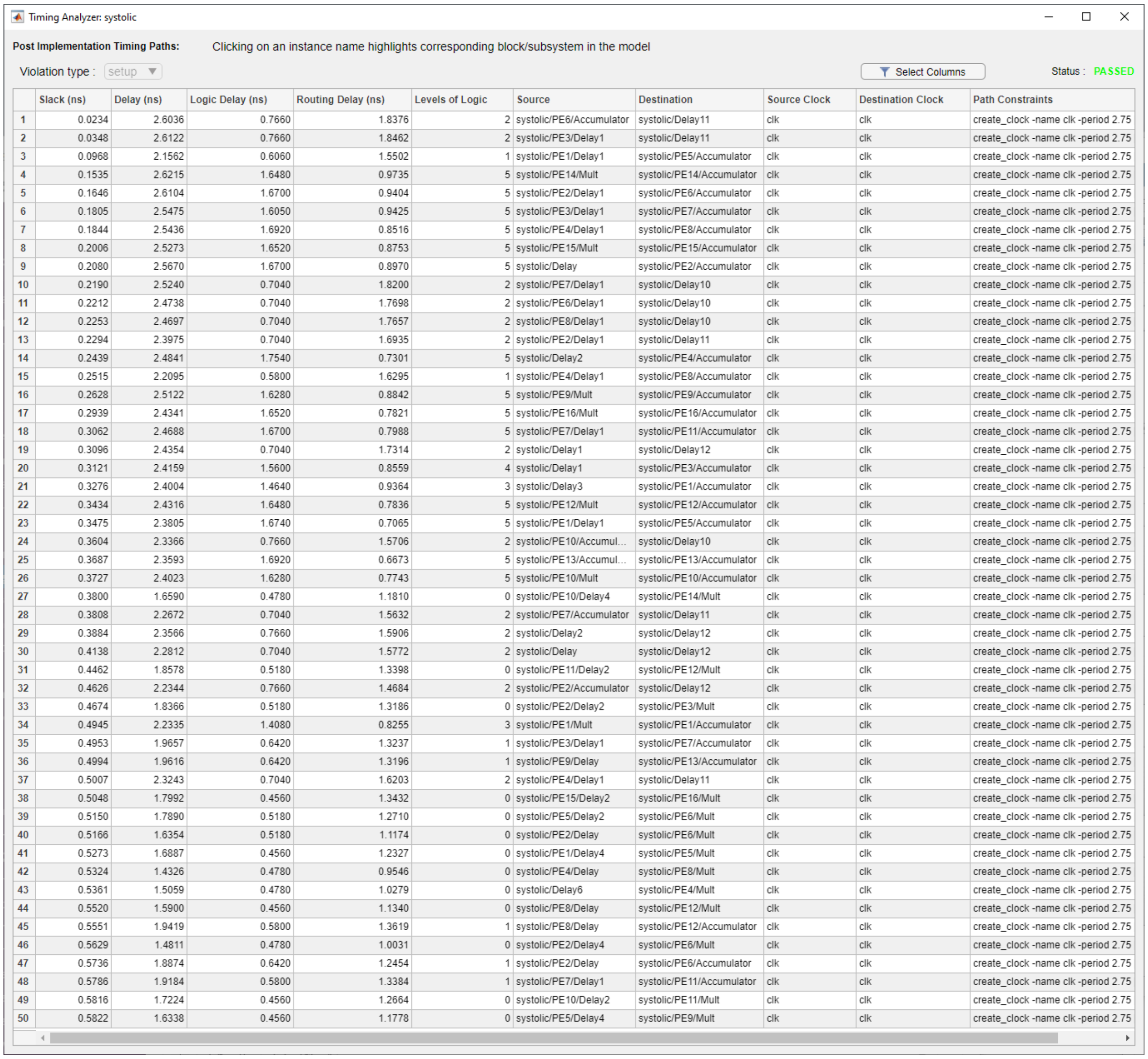
Resource Analyzer: systolic

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

Name	BRAMs (140)	DSPs (220)	LUTs (53200)	Registers (106400)
systolic	0	16	495	832
Relational1	0	0	1	0
PE8	0	1	17	34
PE7	0	1	17	50
PE6	0	1	33	50
PE5	0	1	33	50
PE4	0	1	17	34
PE3	0	1	17	50
PE2	0	1	33	51
PE16	0	1	16	16
PE15	0	1	16	32
PE14	0	1	32	32
PE13	0	1	32	32
PE12	0	1	17	34
PE11	0	1	17	50
PE10	0	1	33	50
PE1	0	1	33	50
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
Delay3	0	0	1	2
Delay2	0	0	0	1
Delay12	0	0	16	32
Delay11	0	0	16	16
Delay10	0	0	0	16
Delay1	0	0	0	1
Delay	0	0	0	1
Counter	0	0	1	2

Timing information

Using post-implementation timing analysis, my design can run with a clock period as short as 2.75 ns (~364 MHz). Full timing information is shown below. With the pipelining approach explained below, the worst 50 paths have slack within 1 ns of each other; there is no single path with drastically worse slack than others.


Timing Analyzer: systolic

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

Violation type : setup

Status : PASSED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock	Path Constraints
1	0.0234	2.6036	0.7660	1.8376	2	systolic/PE6/Accumulator	systolic/Delay11	clk	clk	create_clock -name clk -period 2.75
2	0.0348	2.6122	0.7660	1.8462	2	systolic/PE3/Delay1	systolic/Delay11	clk	clk	create_clock -name clk -period 2.75
3	0.0968	2.1562	0.6060	1.5502	1	systolic/PE1/Delay1	systolic/PE5/Accumulator	clk	clk	create_clock -name clk -period 2.75
4	0.1535	2.6215	1.6480	0.9735	5	systolic/PE14/Mult	systolic/PE14/Accumulator	clk	clk	create_clock -name clk -period 2.75
5	0.1646	2.6104	1.6700	0.9404	5	systolic/PE2/Delay1	systolic/PE6/Accumulator	clk	clk	create_clock -name clk -period 2.75
6	0.1805	2.5475	1.6050	0.9425	5	systolic/PE3/Delay1	systolic/PE7/Accumulator	clk	clk	create_clock -name clk -period 2.75
7	0.1844	2.5436	1.6920	0.8516	5	systolic/PE4/Delay1	systolic/PE8/Accumulator	clk	clk	create_clock -name clk -period 2.75
8	0.2006	2.5273	1.6520	0.8753	5	systolic/PE15/Mult	systolic/PE15/Accumulator	clk	clk	create_clock -name clk -period 2.75
9	0.2080	2.5670	1.6700	0.8970	5	systolic/Delay	systolic/PE2/Accumulator	clk	clk	create_clock -name clk -period 2.75
10	0.2190	2.5240	0.7040	1.8200	2	systolic/PE7/Delay1	systolic/Delay10	clk	clk	create_clock -name clk -period 2.75
11	0.2212	2.4738	0.7040	1.7698	2	systolic/PE6/Delay1	systolic/Delay10	clk	clk	create_clock -name clk -period 2.75
12	0.2253	2.4697	0.7040	1.7657	2	systolic/PE8/Delay1	systolic/Delay10	clk	clk	create_clock -name clk -period 2.75
13	0.2294	2.3975	0.7040	1.6935	2	systolic/PE2/Delay1	systolic/Delay11	clk	clk	create_clock -name clk -period 2.75
14	0.2439	2.4841	1.7540	0.7301	5	systolic/Delay2	systolic/PE4/Accumulator	clk	clk	create_clock -name clk -period 2.75
15	0.2515	2.2095	0.5800	1.6295	1	systolic/PE4/Delay1	systolic/PE8/Accumulator	clk	clk	create_clock -name clk -period 2.75
16	0.2628	2.5122	1.6280	0.8842	5	systolic/PE9/Mult	systolic/PE9/Accumulator	clk	clk	create_clock -name clk -period 2.75
17	0.2939	2.4341	1.6520	0.7821	5	systolic/PE16/Mult	systolic/PE16/Accumulator	clk	clk	create_clock -name clk -period 2.75
18	0.3062	2.4688	1.6700	0.7988	5	systolic/PE7/Delay1	systolic/PE11/Accumulator	clk	clk	create_clock -name clk -period 2.75
19	0.3096	2.4354	0.7040	1.7314	2	systolic/Delay1	systolic/Delay12	clk	clk	create_clock -name clk -period 2.75
20	0.3121	2.4159	1.5600	0.8559	4	systolic/Delay1	systolic/PE3/Accumulator	clk	clk	create_clock -name clk -period 2.75
21	0.3276	2.4004	1.4640	0.9364	3	systolic/Delay3	systolic/PE1/Accumulator	clk	clk	create_clock -name clk -period 2.75
22	0.3434	2.4316	1.6480	0.7836	5	systolic/PE12/Mult	systolic/PE12/Accumulator	clk	clk	create_clock -name clk -period 2.75
23	0.3475	2.3805	1.6740	0.7065	5	systolic/PE1/Delay1	systolic/PE5/Accumulator	clk	clk	create_clock -name clk -period 2.75
24	0.3604	2.3366	0.7660	1.5706	2	systolic/PE10/Accumul...	systolic/Delay10	clk	clk	create_clock -name clk -period 2.75
25	0.3687	2.3593	1.6920	0.6673	5	systolic/PE13/Accumul...	systolic/PE13/Accumulator	clk	clk	create_clock -name clk -period 2.75
26	0.3727	2.4023	1.6280	0.7743	5	systolic/PE10/Mult	systolic/PE10/Accumulator	clk	clk	create_clock -name clk -period 2.75
27	0.3800	1.6590	0.4780	1.1810	0	systolic/PE10/Delay4	systolic/PE14/Mult	clk	clk	create_clock -name clk -period 2.75
28	0.3808	2.2672	0.7040	1.5632	2	systolic/PE7/Accumulator	systolic/Delay11	clk	clk	create_clock -name clk -period 2.75
29	0.3884	2.3566	0.7660	1.5906	2	systolic/Delay2	systolic/Delay12	clk	clk	create_clock -name clk -period 2.75
30	0.4138	2.2812	0.7040	1.5772	2	systolic/Delay	systolic/Delay12	clk	clk	create_clock -name clk -period 2.75
31	0.4462	1.8578	0.5180	1.3398	0	systolic/PE11/Delay2	systolic/PE12/Mult	clk	clk	create_clock -name clk -period 2.75
32	0.4626	2.2344	0.7660	1.4684	2	systolic/PE2/Accumulator	systolic/Delay12	clk	clk	create_clock -name clk -period 2.75
33	0.4674	1.8366	0.5180	1.3186	0	systolic/PE2/Delay2	systolic/PE3/Mult	clk	clk	create_clock -name clk -period 2.75
34	0.4945	2.2335	1.4080	0.8255	3	systolic/PE1/Mult	systolic/PE1/Accumulator	clk	clk	create_clock -name clk -period 2.75
35	0.4953	1.9657	0.6420	1.3237	1	systolic/PE3/Delay1	systolic/PE7/Accumulator	clk	clk	create_clock -name clk -period 2.75
36	0.4994	1.9616	0.6420	1.3196	1	systolic/PE9/Delay	systolic/PE13/Accumulator	clk	clk	create_clock -name clk -period 2.75
37	0.5007	2.3243	0.7040	1.6203	2	systolic/PE4/Delay1	systolic/Delay11	clk	clk	create_clock -name clk -period 2.75
38	0.5048	1.7992	0.4560	1.3432	0	systolic/PE15/Delay2	systolic/PE16/Mult	clk	clk	create_clock -name clk -period 2.75
39	0.5150	1.7890	0.5180	1.2710	0	systolic/PE5/Delay2	systolic/PE6/Mult	clk	clk	create_clock -name clk -period 2.75
40	0.5166	1.6354	0.5180	1.1174	0	systolic/PE2/Delay	systolic/PE6/Mult	clk	clk	create_clock -name clk -period 2.75
41	0.5273	1.6887	0.4560	1.2327	0	systolic/PE1/Delay4	systolic/PE5/Mult	clk	clk	create_clock -name clk -period 2.75
42	0.5324	1.4326	0.4780	0.9546	0	systolic/PE4/Delay	systolic/PE8/Mult	clk	clk	create_clock -name clk -period 2.75
43	0.5361	1.5059	0.4780	1.0279	0	systolic/Delay6	systolic/PE4/Mult	clk	clk	create_clock -name clk -period 2.75
44	0.5520	1.5900	0.4560	1.1340	0	systolic/PE8/Delay	systolic/PE12/Mult	clk	clk	create_clock -name clk -period 2.75
45	0.5551	1.9419	0.5800	1.3619	1	systolic/PE8/Delay	systolic/PE12/Accumulator	clk	clk	create_clock -name clk -period 2.75
46	0.5629	1.4811	0.4780	1.0031	0	systolic/PE2/Delay4	systolic/PE6/Mult	clk	clk	create_clock -name clk -period 2.75
47	0.5736	1.8874	0.6420	1.2454	1	systolic/PE2/Delay	systolic/PE6/Accumulator	clk	clk	create_clock -name clk -period 2.75
48	0.5786	1.9184	0.5800	1.3384	1	systolic/PE7/Delay1	systolic/PE11/Accumulator	clk	clk	create_clock -name clk -period 2.75
49	0.5816	1.7224	0.4560	1.2664	0	systolic/PE10/Delay2	systolic/PE11/Mult	clk	clk	create_clock -name clk -period 2.75
50	0.5822	1.6338	0.4560	1.1778	0	systolic/PE5/Delay4	systolic/PE9/Mult	clk	clk	create_clock -name clk -period 2.75

Scalability of the design

My design should scale reasonably well. There are no global interconnections required by the PEs; all are identical and interact only with their neighbors or with the edges of the array. All signals are systolic and only travel from a processing element to its neighbor, with the exception of data readout. As such, the primary consideration in scalability might be the implementation of this "row bus" readout. It might be the case that eventually it is not possible to read out results fast enough across an entire row within one clock cycle, but it may also be true that the time taken by the multipliers and accumulators would be more significant than this.

Pipelining and latency

I tried to achieve the highest possible clock frequency for my design, while maintaining a reasonable latency for result output. From the time the first elements of the input matrices are present at the top and left edges of the matrix to the time that the first column of the result is shifted out the left, 9 clock cycles are required. Four are needed to get the first value in the leftmost column into to the last PE in the leftmost column, three more are needed for the last value of the leftmost column to be registered in this last PE, and then two additional cycles are needed: one additional cycle for the pipelined multiply, and one for the accumulate.

In other words, I have pipelined my multiplier to take 2 cycles, and the accumulator takes 1 cycle. (This is visible in the above screenshots). If both the multiply and accumulate were performed in one cycle as soon as data is shifted into the PE, it would be possible to reduce the overall latency to 7 cycles; however, performance would suffer. Reducing the multiplier latency to 1 reduces the maximum clock frequency. Increasing the multiplier latency to 3 does not improve the maximum clock frequency.

As mentioned at the very beginning of the report, a series of matrix-matrix or matrix-vector multiplications can be pipelined in my array without requiring any additional clock cycles, spacing, or other "downtime" of the processor elements for loading and unloading. If there is a continuous series of multiplications to be performed, 16 multiplies and 16 accumulates will always be performed on every clock cycle.

Testing methodology

I include several MATLAB files for use in testing my design. *din_matmat.m* and *din_matvec.m* are used to prepare input data for the array; they take sets of matrices and vectors that have been concatenated horizontally. *dout_matmat.m* and *dout_matvec.m* read output data from the non-hardware simulation of the array, while *dout_matmat_hw.m* and *dout_matvec_hw.m* read results produced by the hardware co-simulation. These input and output functions can handle any number of horizontally-concatenated matrices and vectors, and they will feed them to the hardware in the proper pipelined manner.

An overall test script, *test1.m*, is provided to demonstrate how these functions are used and make it easy to test all functionality of the array. It runs two pipelined matrix multiplications, and then two pipelined matrix-vector multiplications. By default, it expects that hardware co-simulation is available, though you can comment out the *dout_matmat_hw* and *dout_matvec_hw* calls to change this. If you load *systolic_hwcosim.slx* and then run *test1* in the MATLAB command window, it will prepare input data and instruct you when to click "Run" in Simulink. Results will be reported, and they should match the results computed in software by MATLAB. The test matrices and vectors used can be seen in *test1.m*.