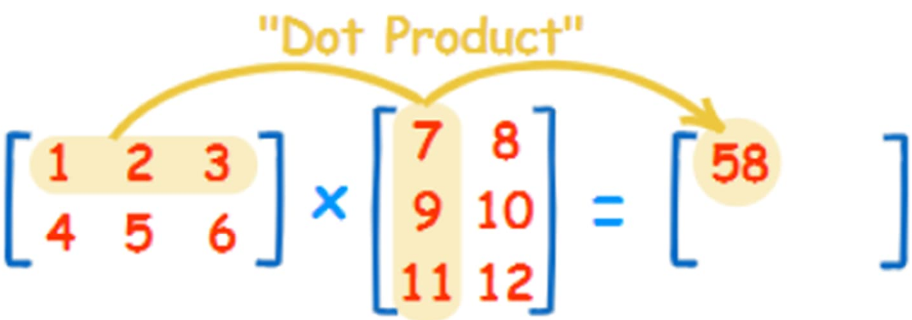
Matchlib Dot Product Example Design

Stuart Swan
Platform Architect
Siemens EDA
17 April 2024

Introduction

This example shows how a simple matrix multiplication (aka "dot product") design can be designed in SystemC using Matchlib and synthesized to RTL using Catapult HLS. The goals of this design are to demonstrate how Matchlib pre-HLS models are throughput accurate, and to show how memory architecture can be tailored to achieve desired design goals.

During matrix multiplication of the A * B matrices, the rows of matrix A are multiplied with the columns of matrix B and then added. This is repeated across all rows and columns. See the diagram below.



Matrix Multiplication

Matrices are usually arranged in raster order in memory. This means that matrices are arranged sequentially row by row. In this design example, the input and output matrices are streamed into and out of the design in row by row order. Within the design, the multiplication operation needs to occur on a row of the A matrix and a column of the B matrix. To do this, we form a transpose matrix of B and then perform the matrix multiplication on the A matrix and the transposed B matrix.

In this example we will explore three different architectures for forming the transpose matrix and performing the matrix multiplication.

Note that the testbench for all the designs in this example uses a 2 ns clock.

Design #1 – Single Process

In the first design for the matrix multiplication, we use a single process that performs the transpose operation and the multiply accumulate operations. If you view the file mat_mul_single_process.h, you will see:

```
 7 #pragma hls_design top
 8 class matrixMultiply  : public sc_module {
 9  public:
10   sc_in<bool> CCS_INIT_S1(clk);
11   sc_in<bool> CCS_INIT_S1(rstn);
12
13   Connections::In <ac_int<8>> CCS_INIT_S1(A);
14   Connections::In <ac_int<8>> CCS_INIT_S1(B);
15   Connections::Out<ac_int<8+8+3>> CCS_INIT_S1(C);
16
17   SC_CTOR(matrixMultiply) {
18     SC_THREAD(run);
19     sensitive << clk.pos();
20     async_reset_signal_is(rstn, false);
21   }
22
```

On lines 13, 14, and 15, we declare the input and output ports for the A, B, and C matrices. All the matrix data is streamed in and out element by element in row order. In the same file, you will see:

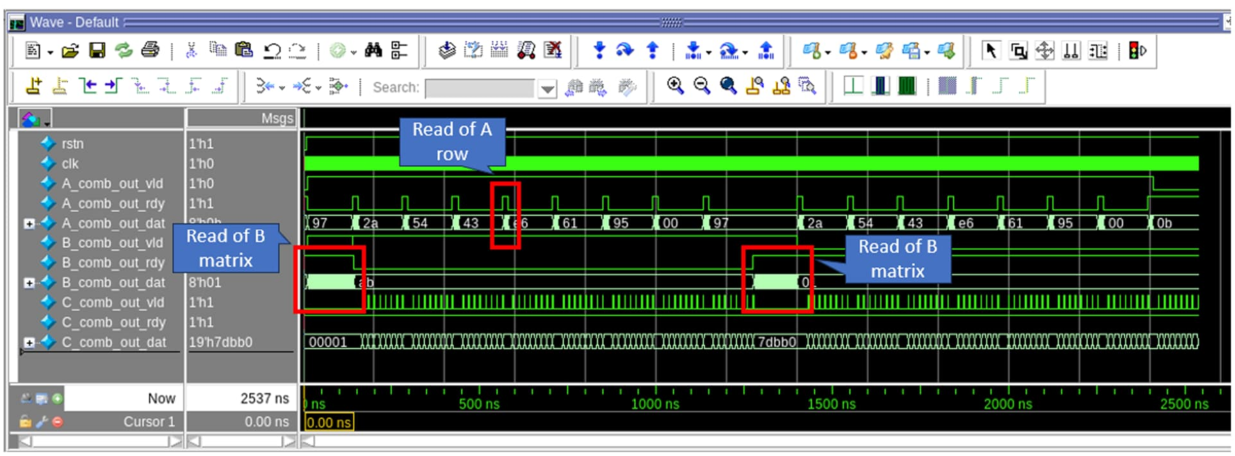
```
23   void run() {
24     A.Reset();
25     B.Reset();
26     C.Reset();
27     ac_int<8> A_row[8];
28     ac_int<8> B_transpose[8][8];
29     wait();
30     #pragma hls_pipeline_init_interval 1
31     #pragma pipeline_stall_mode flush
32     while (1) {
33       ac_int<8+8+3> acc = 0;
34       TRANSPOSEB_ROW:for (int i=0; i<8; i++) { // Transpose operation must complete first
35         TRANSPOSEB_COL:for (int j=0; j<8; j++) {
36           B_transpose[j][i] = B.Pop();
37         }
38       }
39       ROW:for (int i = 0; i < 8; i++) {
40         CPY_A:for (int c=0; c<8; c++){ //Copy one row from A
41           A_row[c] = A.Pop();
42         }
43         COL:for (int j = 0; j < 8; j++) {//Multiply row of A against all cols of B
44           acc = 0;
45           // Cannot unroll MAC loop since it would result in memory port contention..
46           #pragma hls_unroll no
47           MAC:for (int k = 0; k < 8; k++) {
48             acc += A_row[k] * B_transpose[j][k];
49             #ifndef __SYNTHESIS__
50             wait();//wait used to simulate not unrolling the loop in hardware
51             #endif
52           }
53           C.Push(acc);
54         }
55       }
56     }
57   }
58 };
59
```

On line 36 we read in the B matrix and form the B_transpose matrix. Note the reversal of the I and J indices for B_transpose. On line 41 we store an entire row of the A matrix into A_row. Once that is done, we can perform the matrix multiplication on line 48, and push out the sum on line 53. Note that we place a "wait()" statement on line 50 that causes each loop iteration to consume a clock cycle in the pre-HLS simulation. If this were not done, the pre-HLS simulation would complete all the iterations of the loop in zero time. Note that the creation of the B_transpose matrix must be fully completed before the matrix multiplication can begin in this design. This is true in the pre-HLS simulation, and it also will be true in the post-HLS simulation since Catapult HLS cannot merge the loops together.

If you type:

```
make single_process
./single_process
make view_wave
```

You will see the pre-HLS waveforms similar to:

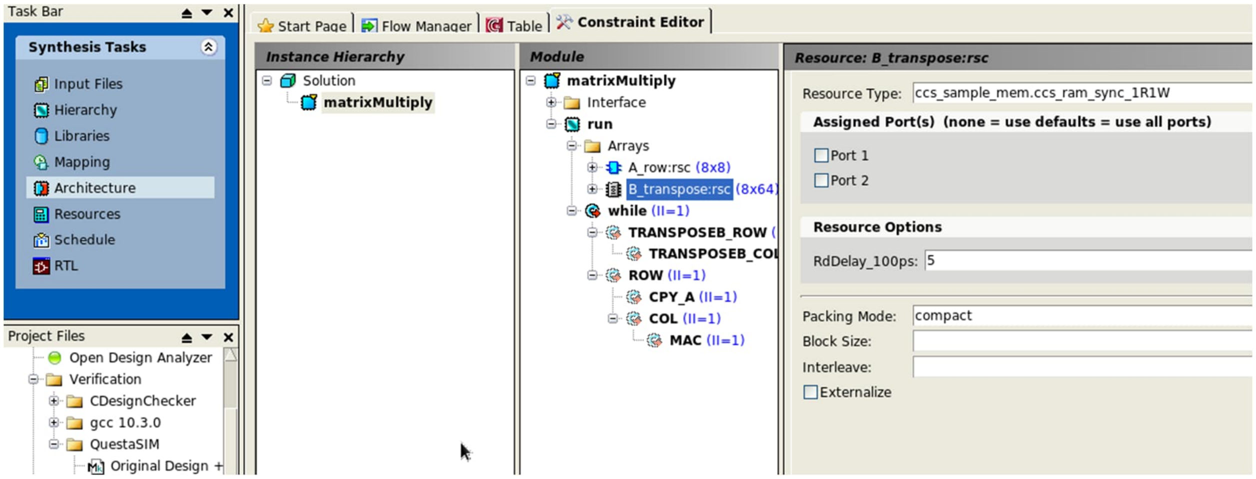


This design has the limitation that there is no overlapping of reading of input data with the matrix multiplication operation. It also has the limitation that the matrix multiplication loop cannot be unrolled during HLS because it would result in a requirement for too many ports to access the B_transpose matrix. Note that this design takes about 2500 ns to execute.

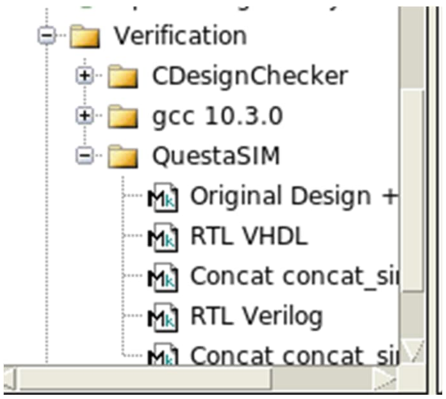
Now run HLS on this design:

```
catapult -f go_hls1.tcl &
```

After HLS is finished, click on the Architecture icon and then expand the Arrays folder and click on the B_transpose item. You will see that the B_transpose array has been mapped to a 1R1W RAM. In other words, it has one read port and one write port.



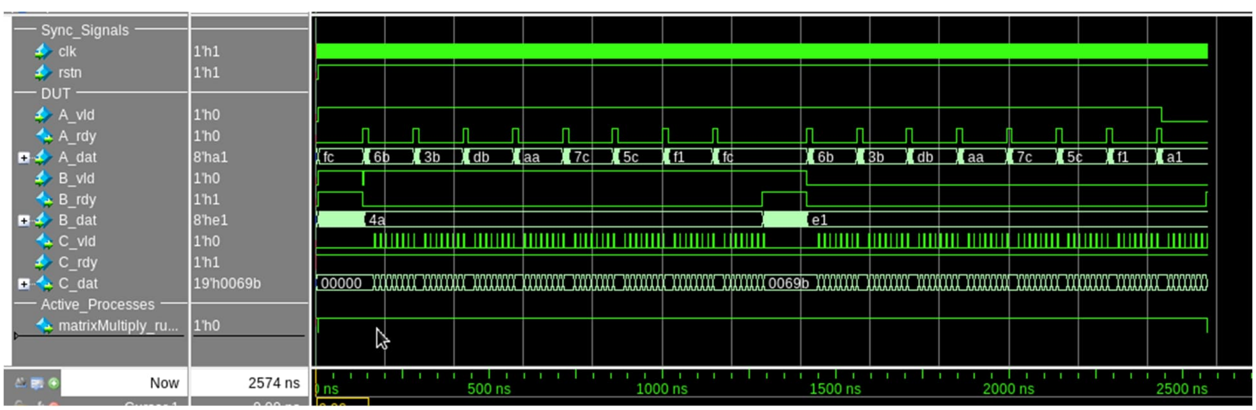
Double click on the RTL Verilog icon shown below to launch Questa on the RTL:



In the Questa command window type:

```
run -all
wave zoom full
```

This will then show:



You should note the very close correspondence and timing of the post-HLS and pre-HLS waveforms. This correspondence exists because Matchlib simulations are "throughput accurate".

Design #2 – Multiple Processes

In the second design for the matrix multiplication, we use a multiple process design that performs the transpose operation, packing of the A row data, and the multiply accumulate operations in three separate processes. If you view the file mat_mul_multi_process.h, you will see:

```
 9 #pragma hls_design top
10 class matrixMultiply  : public sc_module {
11  public:
12   sc_in<bool> CCS_INIT_S1(clk);
13   sc_in<bool> CCS_INIT_S1(rstn);
14
15   Connections::In <ac_int<8>> CCS_INIT_S1(A);
16   Connections::In <ac_int<8>> CCS_INIT_S1(B);
17   Connections::Out <ac_int<8+8+3>> CCS_INIT_S1(C);
18
19   ac_shared<ac_int<8> [64*2]> B_transpose;
20   Connections::Combinational <array_t<ac_int<8>,8>> CCS_INIT_S1(A_row);
21   Connections::SyncChannel CCS_INIT_S1(sync); // memory synchronization between threads
22
23   SC_CTOR(matrixMultiply) {
24     SC_THREAD(transpose);
25     sensitive << clk.pos();
26     async_reset_signal_is(rstn, false);
27
28     SC_THREAD(mac);
29     sensitive << clk.pos();
30     async_reset_signal_is(rstn, false);
31
32     SC_THREAD(pack_A);
33     sensitive << clk.pos();
34     async_reset_signal_is(rstn, false);
35   }
```

On line 19, we declare B_transpose to be a shared memory that stores 8 bit elements. The size of the array is 64*2, since we use a ping-pong synchronization scheme to enable B data to be read while the mat_mul operation also executes. On line 21 we declare the sync channel to coordinate access to the ping-pong memory. On line 20 we declare data channel to transmit the A_row data, so that packing of the A_row items can occur in parallel with the mat_mul operations. In the same file, you will see:

```
37   void transpose() {
38     B.Reset();
39     sync.reset_sync_out();
40     bool ping_pong = false;
41     wait();
42
43     while (1) {
44       #pragma hls_pipeline_init_interval 1
45       #pragma pipeline_stall_mode flush
46       TRANSPOSEB_ROW0:for (int i=0; i<8; i++) { // Transpose operation must complete first
47         TRANSPOSEB_COL0:for (int j=0; j<8; j++) {
48           B_transpose[j*8 + i + 64*ping_pong] = B.Pop();
49         }
50       }
51       ping_pong = !ping_pong;
52       sync.sync_out();
53     }
54   }
55
56   void pack_A() {
57     A.Reset();
58     A_row.ResetWrite();
59     wait();
60
61     #pragma hls_pipeline_init_interval 1
62     #pragma pipeline_stall_mode flush
63     while (1) {
64       array_t<ac_int<8>,8> A_dat;
65       for (int i=0; i<8; i++) {
66         A_dat.data[i] = A.Pop();
67       }
68       A_row.Push(A_dat);
69     }
70   }
```

On line 48 we read the B matrix to form the B_transpose matrix. On lines 51 and 52 we switch the ping_pong variable to control which half of the memory we write into, and on line 52 we notify the mac thread when new data is ready.

On line 66 we read the A matrix data, and when we have a full row we push it out on line 68.

Within the same file, you will see:

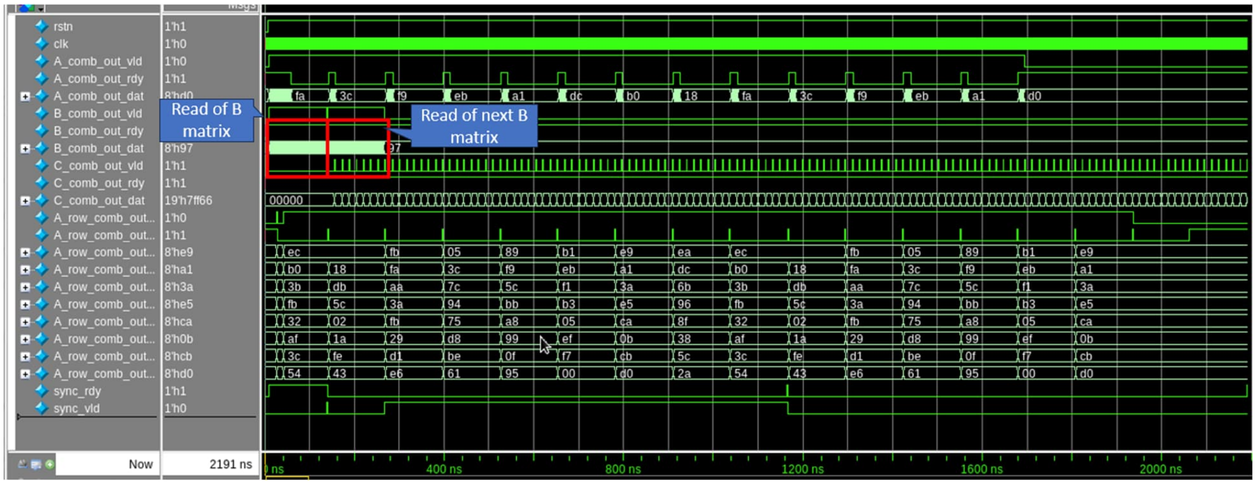
```
72  void mac() {
73    C.Reset();
74    sync.reset_sync_in();
75    A_row.ResetRead();
76    array_t<ac_int<8>,8> A_dat;
77    ac_int<8> B_dat;
78    bool ping_pong = false;
79    wait();
80
81    while (1) {
82      ac_int<8+8+3> acc = 0;
83      sync.sync_in();
84      #pragma hls_pipeline_init_interval 1
85      #pragma pipeline_stall_mode flush
86      ROW:for (int i = 0; i < 8; i++) {
87        A_dat = A_row.Pop();
88        COL:for (int j = 0; j < 8; j++) {
89          acc = 0;
90          MAC:for (int k = 0; k < 8; k++) { // Cannot unroll - would result in mem port contention
91            B_dat = B_transpose[j*8 + k + 64*ping_pong];
92            acc += A_dat.data[k] * B_dat;
93            #ifndef __SYNTHESIS__
94            wait();//need a wait for simulation if loop not unrolled for accurate timing
95            #endif
96          }
97          C.Push(acc);
98        }
99      }
100     ping_pong = !ping_pong;
101   }
102 }
```

On line 83 we synchronize with the transpose process so that the B_transpose data is ready in memory. On line 87 we read an entire A_row of data, and then on line 92 we perform the multiply accumulate operation. Note that we still cannot unroll the MAC loop on line 90 since it would require too many ports on the B_transpose memory, which is still mapped to a 1R1W memory.

If you type:

```
make multi_process
./multi_process
make view_wave
```

You will see the pre-HLS waveforms similar to:

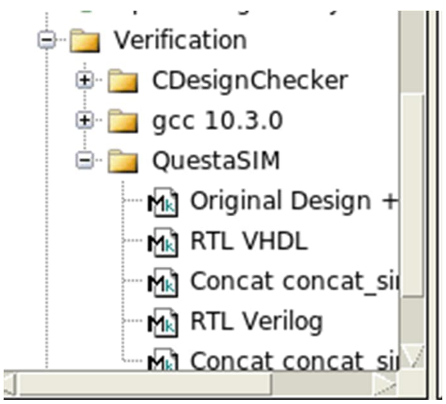


Because of the ping-pong memory for the B_transpose matrix and the separate process for reading the B data, we see that both B matrices are fully read early in the simulation. The mat_mul operations and the output of the C data begins right after the first B matrix is read. The reading of the A matrix is able to occur in parallel with the mat_mul operations. Note, however, that the MAC loop is still rolled, so the overall execution time for this design is only a little better than the first design, despite the additional parallelism enabled by the multiple processes.

Now run HLS on this design:

```
catapult -f go_hls2.tcl &
```

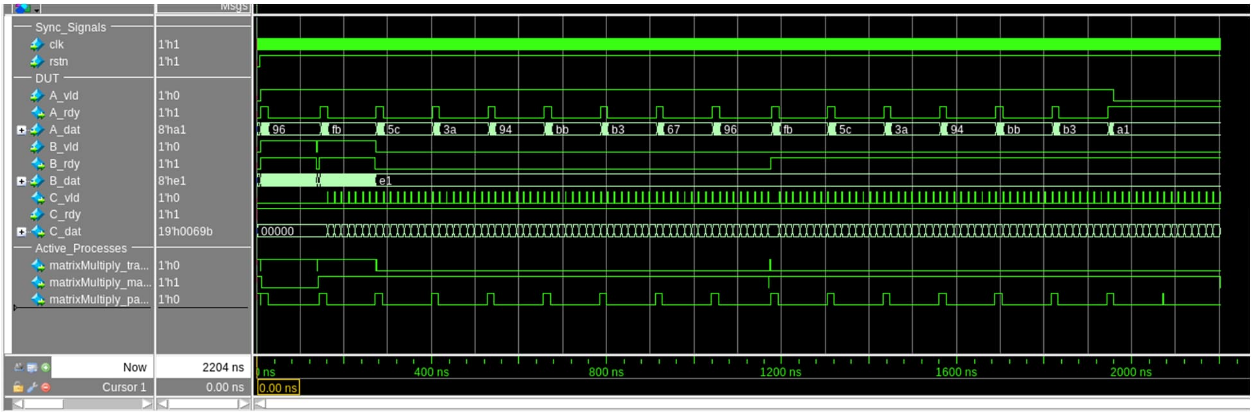
Double click on the RTL Verilog icon shown below to launch Questa on the RTL:



In the Questa command window type:

```
run -all
wave zoom full
```

This will then show:



Again, you should note the very close correspondence and timing of the post-HLS and pre-HLS waveforms. This correspondence exists because Matchlib simulations are "throughput accurate".

Design #3 – Banked Memory and Multiple Processes

In the third design for the matrix multiplication, we use a multiple process design that performs the transpose operation, packing of the A row data, and the multiply accumulate operations in three separate processes. In addition, we use a banked memory for the B_transpose memory, which will enable the MAC loop to be unrolled fully to increase performance. If you view the file mat_mul_banked_multi_process.h, you will see:

```
11 #pragma hls_design top
12 class matrixMultiply  : public sc_module {
13  public:
14   sc_in<bool> CCS_INIT_S1(clk);
15   sc_in<bool> CCS_INIT_S1(rstn);
16
17   Connections::In <ac_int<8>> CCS_INIT_S1(A);
18   Connections::In <ac_int<8>> CCS_INIT_S1(B);
19   Connections::Out<ac_int<8+8+3>> CCS_INIT_S1(C);
20
21   ac_shared_bank_array_2D<ac_int<8>, 8, 8*2> B_transpose;
22   Connections::SyncChannel sync; // memory synchronization between threads
23   Connections::Combinational <array_t<ac_int<8>,8>> CCS_INIT_S1(A_row);
24
25   SC_CTOR(matrixMultiply) {
26     SC_THREAD(pack_A);
27     sensitive << clk.pos();
28     async_reset_signal_is(rstn, false);
29
30     SC_THREAD(transpose);
31     sensitive << clk.pos();
32     async_reset_signal_is(rstn, false);
33
34     SC_THREAD(mac);
35     sensitive << clk.pos();
36     async_reset_signal_is(rstn, false);
37   }
```

This code is the same as in the second design except for line 21. On line 21 we declare a 2-dimensional banked array that stores 8 bit elements. There are 8 banks, and each bank stores 8*2 elements since we are still using a ping-pong memory organization to enable concurrent reading and processing of data.

The pack_A function is the same as in the previous design. The transpose function now is:

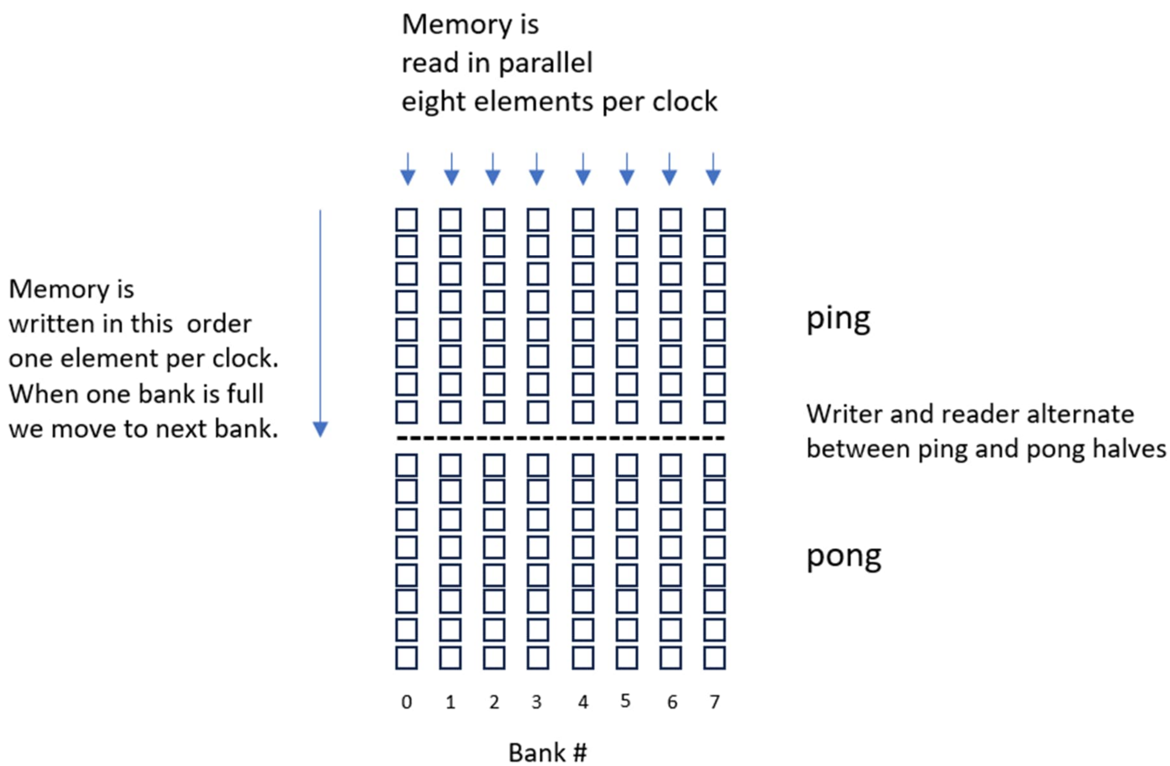
```
59   void transpose() {
60     B.Reset();
61     sync.reset_sync_out();
62     bool ping_pong = false;
63     wait();
64
65     while (1) {
66       #pragma hls_pipeline_init_interval 1
67       #pragma pipeline_stall_mode flush
68       TRANSPOSEB_ROW0:for (int i=0; i<8; i++) { // Transpose operation must complete first
69         TRANSPOSEB_COL0:for (int j=0; j<8; j++) {
70           B_transpose[i][j + 8*ping_pong] = B.Pop();
71         }
72       }
73       ping_pong = !ping_pong;
74       sync.sync_out();
75     }
76   }
```

We see on line 70 that we read one item from the B matrix and store it in the proper position in the banked memory, considering the current ping_pong settings. These transpose loops could be unrolled during HLS, but there would be no benefit in either performance or area, so we leave them rolled.

Within the same file, you will see:

```
78    void mac() {
79      C.Reset();
80      A_row.ResetRead();
81      sync.reset_sync_in();
82      array_t<ac_int<8>,8> A_dat;
83      ac_int<8> B_dat;
84      bool ping_pong = false;
85      wait();
86
87      while (1) {
88        ac_int<8+8+3> acc = 0;
89        sync.sync_in();
90        #pragma hls_pipeline_init_interval 1
91        #pragma pipeline_stall_mode flush
92        ROW:for (int i = 0; i < 8; i++) {
93          A_dat = A_row.Pop();
94          COL:for (int j = 0; j < 8; j++) {
95            acc = 0;
96            #pragma hls_unroll yes
97            MAC:for (int k = 0; k < 8; k++) { // loop can be unrolled without mem port contention..
98              B_dat = B_transpose[k][j + 8*ping_pong];
99              acc += A_dat.data[k] * B_dat;
100           }
101           C.Push(acc);
102         }
103       }
104       ping_pong = !ping_pong;
105     }
106   }
107 };
```

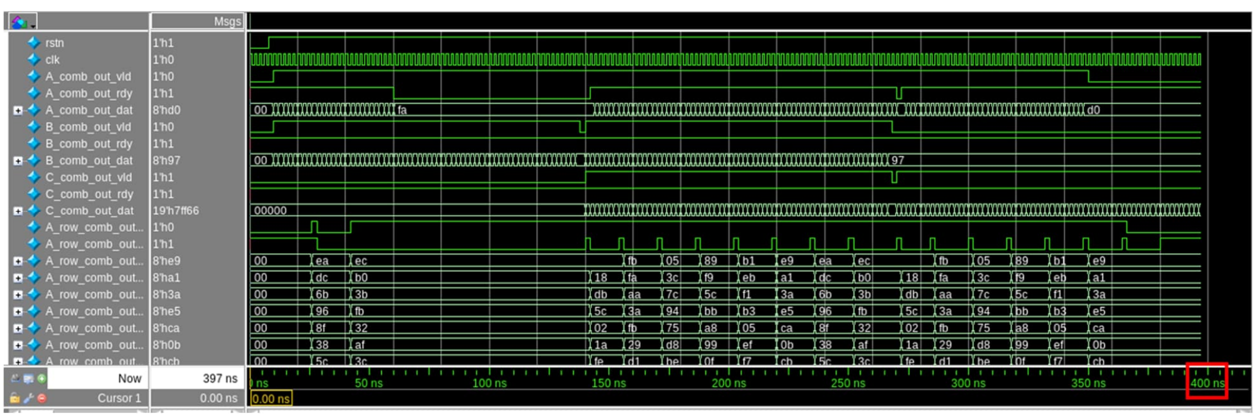
On line 97 is the MAC loop, which we now unroll fully during HLS. When this is done, the k index to the B_transpose memory becomes a constant during HLS, and Catapult can see that there is no memory port contention. This allows the entire MAC loop to complete in a single cycle (note however that now the design will require 8 multipliers in HW). On line 101 we push out a new dot product result, which now will occur about every clock cycle.



If you type:

```
make banked_multi_process
./banked_multi_process
make view_wave
```

You will see the pre-HLS waveforms similar to:



Note the following in the waveforms above:

- The reading of the A and B matrices happens concurrently with the mat_mul operation and the output of the C matrix.

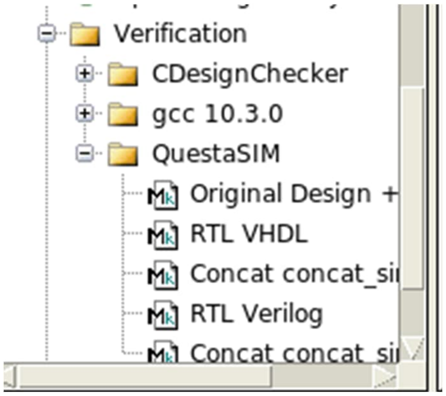
- The output of the C matrix is completely overlapping with the reading of the second B matrix.

- The simulation completes in about 400 ns, which is much faster than the > 2000 ns for the previous two designs. This is because the MAC loop is now able to be completely unrolled due to the use of the banked B_transpose memory. Remember that this design uses a 2 ns clock.

Now run HLS on this design:

```
catapult -f go_hls3.tcl &
```

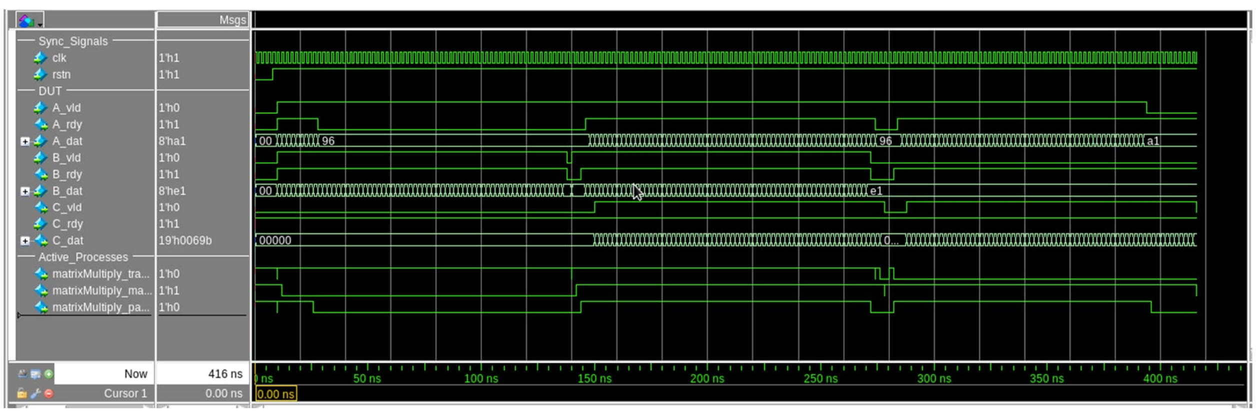
Double click on the RTL Verilog icon shown below to launch Questa on the RTL:



In the Questa command window type:

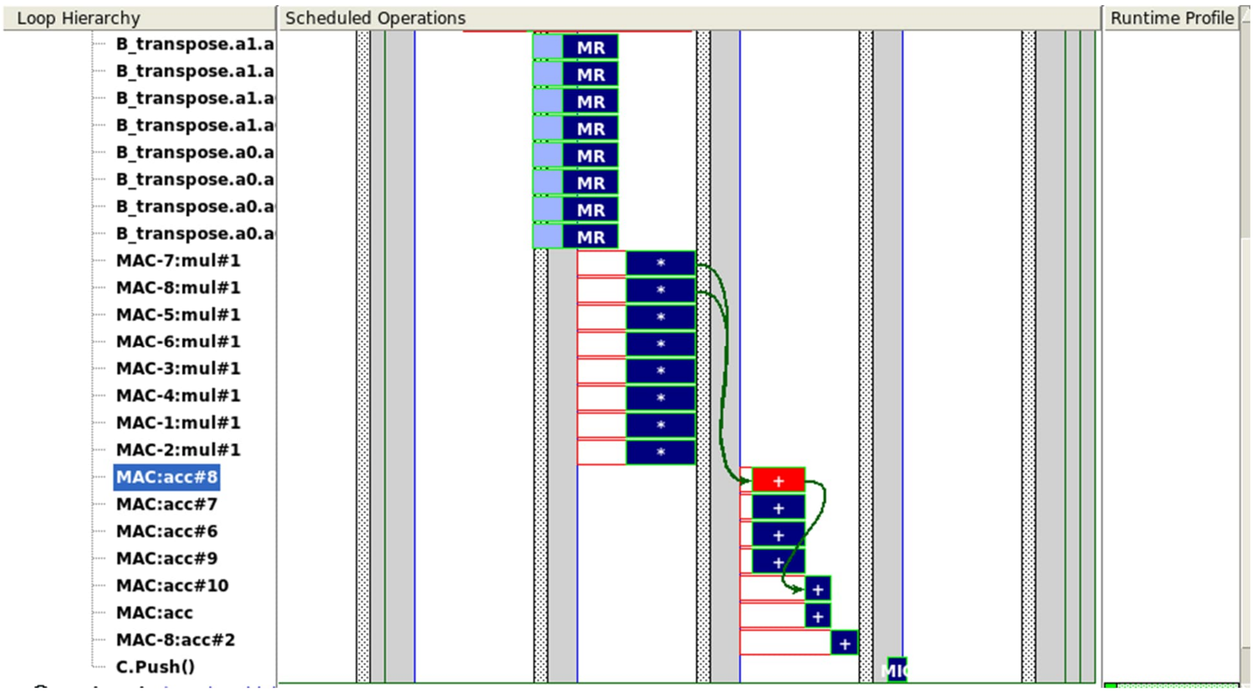
```
run -all
wave zoom full
```

This will then show:



Again, you should note the very close correspondence and timing of the post-HLS and pre-HLS waveforms. This correspondence exists because Matchlib simulations are “throughput accurate”.

If we view schedule in Catapult for the operations in the MAC (multiply accumulate) loop, we see:



We can see that the overall pipeline for this loop has several clock cycles of latency, but on each cycle 8 memory read operations occur, and 8 multiplications and corresponding additions occur. In the post-HLS RTL, all these details are present. In the pre-HLS model, the memory read operations and the 8 multiplications and corresponding additions occur in zero time.

Despite these differences between the two models, the overall performance accuracy of the pre-HLS model compared to the post-HLS model is very high.