University of California, Davis
Department of Electrical and Computer Engineering

EEC180 DIGITAL SYSTEMS II Winter Quarter 2025

LAB 6: Matrix Multiplication in Hardware

Background and Motivation

Matrix multiplication is a common operation in many real-world applications such as wireless communication, image processing, video analysis and processing, computer games and graphics, machine learning, and simulation of many physical and chemical processes, among others. If we have two $n \times n$ matrices A and B, then their product C = A*B is also a square $n \times n$ matrix whose entry c_{ij} in column i and row j is given by the equation

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

For example, if $n = 8$, then

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} + a_{15}b_{51} + a_{16}b_{61} + a_{17}b_{71} + a_{18}b_{81}$$

Matrix multiplication on a conventional computer can be slow because typically processors have only one Arithmetic Logic Unit (ALU). Each element of the 8x8 product matrix requires 15 arithmetic operations on top of any address calculations needed to access the right memory contents and thus a full matrix multiply can take many clock cycles.

Implementing matrix multiplication in hardware allows us to take advantage of parallelism and high memory bandwidth to improve the performance significantly. The core computation in matrix multiplication is multiplying two numbers and adding it to a running sum (called multiply-accumulate or MAC). So, in this laboratory exercise we will start with the implementation of a MAC in hardware and then use that MAC to realize two implementations of matrix multiplication and evaluate their performance.

You don't have to implement the hardware on the DE10 board in this lab but you should still make sure that your designs synthesize properly and estimate their hardware resources.

Task 1 - Implement a MAC

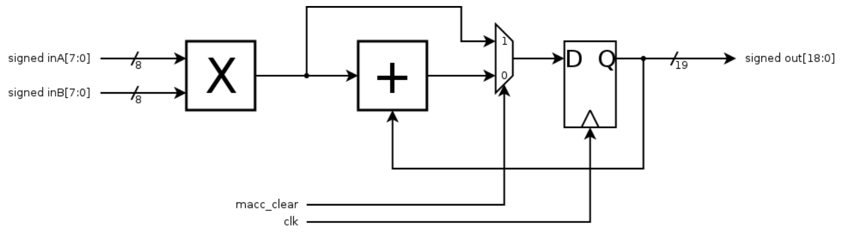


Figure 1: Block Diagram for a Multiply Accumulator

One of the fundamental building blocks in digital signal processing is the Multiply Accumulator (MAC) shown in Figure 1. The operation of a MAC is as follows: Given inputs x and y and accumulator register s, the operation

$$s \leftarrow s + xy$$

is performed every clock cycle. With this operation, one can see how each element of C can be calculated over a number of clock cycles.

Write a Verilog program to implement the MAC module. Your module should take two 8-bit signed 2's complement inputs and use a 19-bit signed accumulator to produce the result. It is important to be able to clear the MAC when each entry in C is calculated. Therefore, you should also include an input signal *macc_clear* that when asserted assigns the product of the inputs directly to the output as shown in the block diagram.

You may use the operator " * " in Verilog to instantiate a multiplier and " + " to instantiate an adder.

Write a testbench using the skeleton code provided to verify your design. Your testbench does not have to automatically verify your results but should thoroughly test your design. Synthesize your design and tabulate the resources Show the output waveforms and resource utilization on the FPGA to your TA.

Task 2 - Matrix Multiply with One MAC

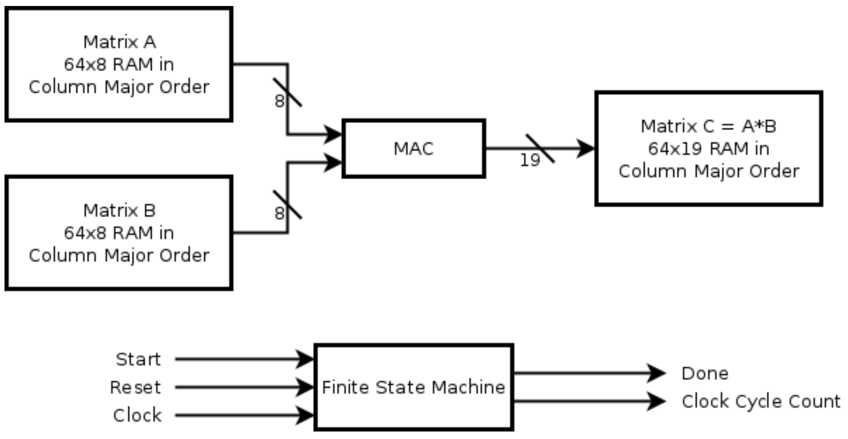


Figure 2: High-level block diagram for Task 2.

You will now design a module to multiply two 8x8 matrices, implementing the function C = A * B by instantiating only one of the MAC modules you have designed. Access one element from each of the input RAMs at a time and use your MAC in such a way that one entry of the product matrix C is calculated about every 8 or 9 clock cycles. This algorithm is very similar to how you would multiply two matrices by hand. Remember that you must *access matrix B by columns* while *matrix A needs to be accessed by rows*.

The input and output signals for your module are given in the table below.

Signal Name	**Signal Type**	**Description**
clk	Input	Clock
start	Input	Start signal. Your module should not begin calculation until this signal is asserted high
reset	Input	Reset signal. Should reset your module to its default waiting state. Note, this should NOT clear the contents of any RAM.
done	Output	Completion signal. This signal should be '0' while your module is running, and '1' when it has completed the multiplication.
clock_count[10:0]	Output	Number of elapsed clock cycles. This signal should be set to zero when the module begins calculation and increment by 1 every clock cycle until the multiplication is complete, at which point it stops incrementing.

In order to test your module successfully with the provided testbench, you should make sure to do following:

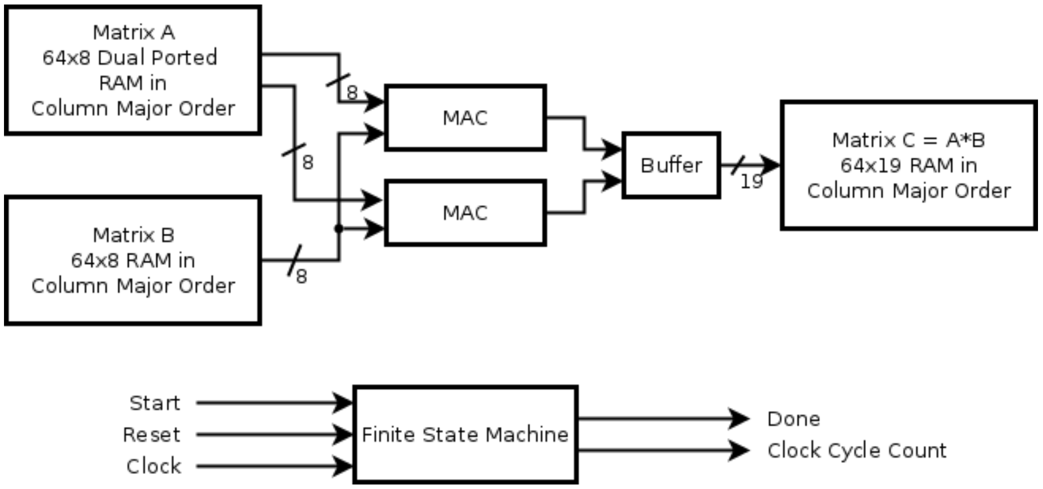
- Initialize the contents of the RAMs associated with the A and B matrices to the contents of "ram_a_init.txt" and "ram_b_init.txt" respectively. Use the $readmemb() task inside of an initial block to do this.
- You should write the output RAM as its own module and instantiate it in your top-level design. It is very **important** that when you instantiate the RAM, you give it the instance name "**RAMOUTPUT**" and in your RAM module, the memory vector is named "**mem**". Make "**mem**" a signed variable so ModelSim interprets its contents as 2's complement numbers. These steps are important so the testbench accesses the correct signals.

Make the following assumptions:

- Matrix A and Matrix B are 8x8 matrices stored *in column major order* in two independent RAM modules.
- The entries of matrices A and B are 8-bit signed numbers.
- Matrix C is an 8x8 matrix stored in a RAM in column major order.
- The entries of matrix C are 19-bit signed numbers.

Verify your module with the provided testbench, making only minimal changes to the testbench if necessary so signal and module names are consistent. Demonstrate the simulation to your TA and record the number of clock cycles the computation took.

Task 3 - Matrix Multiply with Two MACs



Matrix A 64x8 Dual Ported RAM in Column Major Order
Matrix B 64x8 RAM in Column Major Order
8
8
8
MAC
MAC
Buffer
19
Matrix C = A*B 64x19 RAM in Column Major Order
Start
Reset
Clock
Finite State Machine
Done
Clock Cycle Count

Figure 3: Example architecture for performing matrix multiplication using two MACs.

Performing matrix multiplication the way we did in Task 2 does not exploit any parallelism and thus does not offer much speedup over what we might be able to achieve on a simple processor. Since the M9K memory blocks are dual ported, we can read memory contents from two separate addresses each clock cycle. We can take advantage of this and parallelize our matrix multiplication hardware. One way you might consider doing this is to read one column from B and two rows from A at a time. This technique means that two entries of C can be calculated in parallel.

Modify your module from Task 2 in the following ways:

A) Dual port just one of the input matrix memory RAMs. You can reference for help in implementing a dual port RAM. Remember that values will only be read from this RAM, making it really a ROM.
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/max-10/ug_m10_memory.pdf

B) Instantiate two MAC modules designed in Task 1.
C) Make any necessary changes to your control logic to facilitate calculating multiple entries in the product matrix at a time.

Assume that your output RAM **cannot** be dual ported. Thus, it might be necessary to buffer values computed by the MACs so they can be written to memory one at a time.

Verify your design using the same testbench used in Task 2. Show your simulation results to your TA and record the number of clock cycles that the computation took.

Task 4 - Optimize **your design to improve throughput.**

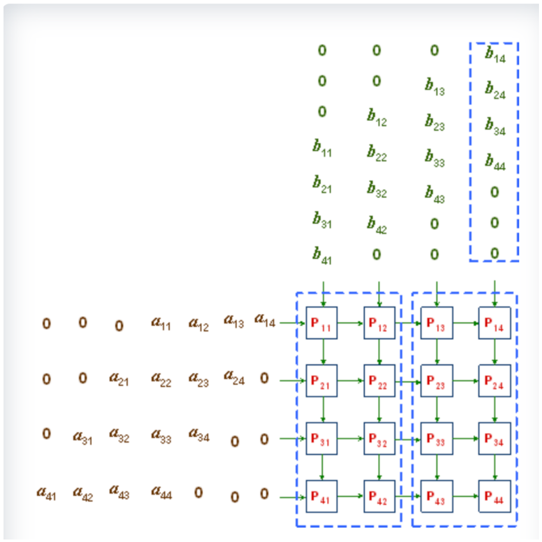
Optimize your design in Task III to have better throughput (MAC/s) by adopting parallelism in terms of MAC usage and pipelining the datapath.
This task will be graded relatively, which means each group will get the points out of 50 based on their designs and corresponding throughput.
To receive any points for this task you should have a **fully synthesized and functional design**.

Extra Credit [100 Points]

Here are two possible opportunities for extra credit. You can also come up with your own ideas to improve the performance of matrix multiplication.

1. <u>**Systolic Array Based Matrix Multiplication**</u>

Conventional matrix multiplication has a complexity of O(n^3) which is terrible when n is large. One way to improve the efficiency of matrix multiplication is to take advantage of spatial parallelism, using a 2D array of processor (or compute elements). Such architectures are called systolic arrays. Tensor Processing Unit (TPU) is a specialized processor for large scale matrix multiplication based on the idea of systolic arrays that forms the hardware backbone of Google's machine learning implementations.
At a high level Figure show how systolic array based matrix multiplication works.



b_{11} b_{21} b_{31} b_{41}
b_{12} b_{22} b_{32} b_{42}
b_{13} b_{23} b_{33} b_{43}
b_{14} b_{24} b_{34} b_{44}
a_{11} a_{12} a_{13} a_{14}
a_{21} a_{22} a_{23} a_{24}
a_{31} a_{32} a_{33} a_{34}
a_{41} a_{42} a_{43} a_{44}
P_{11} P_{12} P_{13} P_{14}
P_{21} P_{22} P_{23} P_{24}
P_{31} P_{32} P_{33} P_{34}
P_{41} P_{42} P_{43} P_{44}

Figure 4 How Systolic Array Based Matrix Multiplication Works [Courtesy: https://ecelabs.njit.edu/ece459/lab3.php}

The two matrices *A* and *B* are shifted into the boundary processors in column 1 and row 1, respectively, as shown in Figure 4. The leading and trailing 0s in rows and columns are employed so that elements a_{ir} and b_{rj} arrive at processor P_{ij} simultaneously for the operation $a_{ir} \bullet b_{rj}$ to be performed. c_{ij} is initialized to 0 in P_{ij}, for all *i, j* = 1, 2, 3, 4. At the end, processor P_{ij} will contain c_{ij}, for $1 \leq i, j \leq 4$. Whenever a processor P_{ij} receives two inputs **b** and **a** from the north and the west, respectively, it performs the following set of operations, in this order:

1. it calculates a ● b;
2. it adds the result to the previous value cij , and stores the result in c_{ij} ;
3. it sends a to $P_{i,j+1}$, unless j = 4; and
4. it sends b to P_{i+1}, j, unless i = 4.

Implement a 8x8 systolic array multiplier. Verify its operation. Compare the number of cycles of the systolic array implementation with your results in Task 4.

2. Sparse matrix vector Multiplication (SpMV)

In many emerging applications the matrices are very sparse, which means most of the entries in the matrices are zero. The straightforward algorithm that we are using thus far is extremely inefficient for sparse matrices since we are wasting time and power and memory bandwidth/capacity storing and multiplying by zeros. See https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format) for examples of how sparse matrices can be stored. Implement a Sparse parallel Matrix Vector multiplier based on the CSR format – assume the matrix is 16x16 and vector is 16x1 and assuming the sparsity is 25% which means only 4 elements per row are non-zero.

Lab Requirements [300 points]

Lab6 is divided into 4 milestones. Make sure you meet the deadlines for each milestone. All the tasks (including extra credit) must

Milestones	When?	Points
Milestone 1 – Task 1 & Task 2	Week February 17	150
Milestone 2 – Task 2	Week February 24	100
Milestone 3 – Task 3	Week March 4	50
Milestone 4 – Extra Credit	Week March 10	100

completed and signed off by March 13, 2025.

A. Task 1 [50 points]

1. Write a synthesizable model for a MAC.

2. Write a testbench to verify the operation of the circuit.
3. Verify your design through functional simulation. Demonstrate your simulation to your TA and have them sign a verification sheet when simulation is successful.
4. Estimate the resources usage – the number of flip-flops, logic elements, memory blocks, embedded multipliers etc.

B. Task 2 [100 points]

1. Design a circuit and write the Verilog to perform matrix multiplication of two 8x8 matrices by instantiating a single MAC.
2. Verify your design using the provided testbench. Demonstrate your simulation to your TA and have them sign a verification sheet when successful.
3. Estimate the resources usage – the number of flip-flops, logic elements, memory blocks, embedded multipliers etc.
4. Estimate the performance of your design in terms of number of clock cycles using the performance counter and identify the bottleneck of your design.

C. Task 3 [100 points]

1. Dual port one of the input matrix RAM modules and design a circuit that performs matrix multiplication using two MAC units.
2. Verify your design using the provided testbench. Demonstrate your simulation to your TA and have them sign a verification sheet when successful.
3. Estimate the resources usage – the number of flip-flops, logic elements, memory blocks, embedded multipliers etc.
4. Estimate the performance of your design in terms of number of clock cycles using the performance counter and identify the bottleneck of your design.

D. Task 4 [50 Points]

1. Optimize your Task III design to have better throughput.
2. Verify your design using the provided testbench. Demonstrate your simulation to your TA and have them sign a verification sheet when successful.

Lab6 is divided into 4 milestones. Make sure you meet the deadlines for each milestone. All the tasks (including extra credit) must completed and signed off by March 13, 2025.

Milestones	**When?**	**Points**
Task 1 & Task 2	Week February 17	100
Task 3	Week February 24	100
Task 4	Week March 4	50
Extra Credit	Week March 10	100

Prelab

Design the state diagram for the controller for Task 2.

Lab report

Submit all Verilog code that you wrote in .v files in a single zip file. Don't submit any of

the instructor given test benches. [15 Points]
Submit a brief report on the lab and explain the reasoning behind the area/performance optimization methods that you adopted. [15 Points]

Testbench Description

A testbench to verify your design has been provided. A brief description of its operation is provided here. Refer to tb_lab6.v to review the sections described:

PART 1: The testbench initializes the matrix multiplication module by asserting a reset for 2 clock cycles, waiting idly for 2 clock cycles, then asserting the start signal for one clock cycle.

PART 2: The testbench then waits for the "done" signal to be set by the module. To ensure termination, a fork-join construct is used and if 2000 clock cycles pass before the done signal is set, the testbench automatically finishes.

PART 3: The A and B matrices are displayed. The expected result matrix C is calculated and displayed. The contents of RAM in the matrix multiply module are extracted and displayed.

PART 4: The contents of RAM are compared to the expected result. If all entries match what is expected, a success message is displayed. Otherwise, the indices of the mismatching elements are displayed.

FREQUENTLY ASKED QUESTIONS

Please read the following contents before sending emails. Also, please don't just send questions on debugging without sending .v files and with adequate background information stating your efforts in solving the issues. Hopefully, it helps.

Can I write data from 2 MACs to output RAM at the same clock cycle?

No, output RAM is a single port memory. So, you can't write two data at the same clock cycle.

How to transfer data from 2 MACs to output RAM?

You can think of buffering the output from 2 MACs and then, pass it on to output RAM one by one. You can implement a buffer as a 19-bit register.

In extra credits part a. do we need to implement matrix multiplication of order 16*16 or each element in the matrix is of 16-bits instead of 8-bits?

Order of matrix is 16*16, so there are 256 elements in the matrix.

Can I use FOR loops to fetch data from RAM A and RAM B and do the matrix multiplication?

It’s not recommended, instead you can try some simple addressing scheme to support column major format. For example, to produce the first element of output matrix (row 0, col 0), you need to fetch data from RAM A at address 0, 8, 16, 24, 32, 40, 48, 56 for matrix A row 0 and data from RAM B at address 0,1,2,3,4,5,6,7 for matrix B col 0.

I am getting number of clock cycles ~550 for single MAC multiplication. How to solve this issue?

Double check the latency with MAC output, whether you’re reading data from both RAM A and RAM B every clock cycle and whether you’re generating an output every 8 clock cycles or not. Moreover, the clock cycle count logic should be implemented only for multiplication execution state in your FSM; don’t include reset state.

Output from MAC for first row is correct, but the following output are all wrong.

Make sure that your MAC Reset is active every 8 clock cycles. Once this signal gets asserted, product of two input will proceed to output.

In sparse matrix multiplication, how many non-zero elements I should consider and how to create such matrix?

In the lab manual, it says that sparsity is 25% which means only 4 elements are non-zero per row. You can use sparse() in MATLAB or just manually create a matrix manually with given condition.

I am getting output all correct on waveform, but 1 clock cycle earlier. Therefore, on transcript window I am getting all incorrect output.

Make sure that write enable for output RAM has a single clock cycle latency to match with MAC register delay.
