UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC 181A Winter 2024

Lab 2: Matrix Multiplication in Hardware

Background and Motivation

Matrix multiplication is a common operation in many real-world applications such as machine learning, wireless communication, image processing, video analysis and processing, computer games and graphics, computational finance, 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 *cij* 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 8×8 product matrix requires 15 arithmetic operations (8 multiplies and 7 adds) in addition to 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 performance significantly. The core computation in matrix multiplication is multiplying two numbers and adding the product to a running sum (also called an accumulated sum). This calculation is efficiently performed with a multiply-accumulate or MAC operation. 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.

1. Prelab

Review the material on the following handouts posted on the course webpage:

- *Memories*
- *M10K memories*
- *Control circuits and counters*
- *Finite state machines*

2. M10K Memory-Based Accumulator

The M10K memory block is a synchronous, true dual-port memory block, with registered inputs and optionally registered outputs, available in Arria V and Cyclone V family devices. (It is very similar to the M9K you may have used on the DE10-Lite board.) The M10K block may be configured as true dual-port, simple dual-port, or single-port RAM or ROM.
https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/reference/glossary/def_m10k.htm

See the handout *FPGA Resource Usage* to determine whether or not an M10K block was instantiated.

Design a circuit that performs the following operations in order:

1. After reset by a push button switch, load an M10K memory with the following 16 *unsigned* 8-bit values:
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
2. Next, the contents are read one word at a time and summed in a 16-bit accumulator built by a single 16-bit adder and a 16-bit register. Because the inputs are unsigned, sign extension is not needed.
3. Display the final sum on the 7-segment display in hexadecimal format.

The circuit is controlled by a state machine. Almost certainly the most likely place to make errors is in timing signals so that addresses, data, and control signals all arrive at the correct time. There are two solutions: 1) guess and probably never finish, or 2) work very diligently with your timing diagrams and block diagrams and do not type in any verilog until you are sure your design will work.

Submit:

1. A pipelined block diagram
2. A detailed timing diagram that includes all key signals

Demonstrate the operation of your design to your TA in lab.

3. Understanding Matrix Multiplication in MatLab

Execute the following commands in matlab:

```
>> a = [1 2 3 4 ; 5 6 7 8 ; 9 10 11 12 ; 13 14 15 16]
>> b = [0 0 0 0 ; 1 1 1 1 ; 2 2 2 2 ; 3 3 3 3]
>> c = [0 1 2 3 ; 0 1 2 3 ; 0 1 2 3 ; 0 1 2 3]
>> d = round(rand(4) * 100)
>> a * b
>> a * c
>> a * d
```

write matlab code to perform matrix multiplies using * and + with real (non-matrix) inputs for both:

- 4×4 * 4×4 matrix multiplier, and verify using the data specified and a*b, a*c, and a*d
- 8×8 * 8×8 matrix multiplier, and verify using your own data

To eventually make the answers displayable on your FPGA board, the individual numbers in the output matrix will be XOR'd together to form a single number. In matlab this is done with the bitxor() function which can be understood with some simple examples such as the following:

```
>> bitxor(3,12)       % 0011 XOR 1100
>> bitxor(12,3)       % 1100 XOR 0011
>> bitxor(63,63)      % 111111 XOR 111111
```

Write matlab code to calculate the XOR of all of the members of the three matrix products you calculated in the first part of this section. This is probably best done with two for loops.

Submit your matlab code.

4. Matrix Multiplier Processor Using One MAC

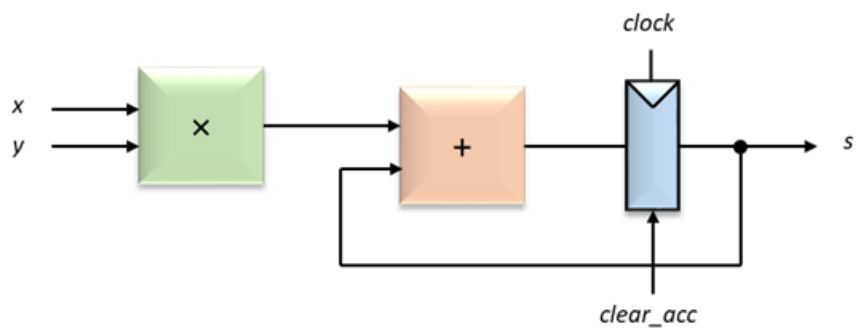

Figure 1. Circuit diagram of a multiply-accumulator (MAC)

One of the most fundamental building blocks for digital signal processing is the Multiply Accumulate (MAC) operation which functions as follows: Given inputs x and y and accumulator register s, the operation consists of

$$s \leftarrow s + xy$$

The example hardware shown in Figure 1 calculates one MAC operation per clock cycle. With MAC hardware, each element of a matrix multiplication can be calculated efficiently utilizing multiple MAC operations.

Design a module to multiply two 4×4 matrices ($c = a \times b$) with 8-bit unsigned inputs, and utilize one RAM to store input a, one RAM to store input b, and one RAM to store 20-bit unsigned output c. The processor must process one MAC per cycle once it is in steady state. Remember to access matrix a by rows and matrix b by columns.

Design a state machine that performs these three major functions in order when the *start* signal is asserted: 1) load data into the memory holding input matrix a and the memory holding input matrix b, 2) compute the matrix multiply, and 3) compute the XOR checksum of result c (see below for details). Since the design must work on the DE1-SoC, the source data of a and b must be stored in the hardware and not the testbench. Use any method you like (e.g., see ROM section in the *Memories* handout) to store the input data. Take care to watch for any cases where data may be dependent across these functions such as "read after write" hazards.

Design a hardware block that generates two different sets of input data for a and b, depending on the position of an SW switch. The generator should be easy to implement in hardware and must include the following operations at least once in the matrix multiplies: 0 × 0, 128 × 128, and 255 × 255.

To verify the output *c* is correct, *after* (not while) the entire matrix multiplication is complete, read the results of *c* from its RAM, XOR all of the values, and display the resulting 20-bit word in hexadecimal on the 7-segment displays, controlled by SW switches using a scheme chosen by you.

Implement a performance counter that counts the number of clock cycles the actual matrix multiply requires, so only step (2) in the three step process described previously. Using details chosen by you, the number of cycles is to be displayed on the 7-segment displays in hexadecimal when one or more SW switches are appropriately set.

Some key input and output signals for your module are given 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.

Use LEDs, 7-segment displays, SW switches, and push buttons to show internal values of your design to aid debugging.

Submit
1. A pipelined block diagram
2. A detailed timing diagram that includes all key signals

Demonstrate the operation of your design on your DE1-SoC to your TA in lab.

5. Matrix Multiplier Processor Using Four MACs

Performing matrix multiplication with one hardware MAC the way it was described in Section 4 does not offer a significant speedup over what we might be able to achieve on a simple processor.

Since the M10K memory blocks can be configured with different word widths up to 40 bits, take advantage of this capability to compute 4 MACs per cycle using 4 hardware MAC units. Read four values of *a* and four values of *b* every cycle and utilize temporary register(s) and appropriate control and datapaths to write the exact same results *c* that your single-MAC design wrote. There are several ways to organize this computation and the details of the design are up to you. All other requirements are the same as with the single-MAC design.

Submit
1. A pipelined block diagram
2. A detailed timing diagram that includes all key signals

Demonstrate the operation of your design on your DE1-SoC to your TA in lab.