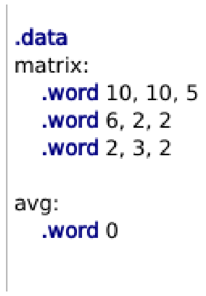
Architecture project 2 part 4

1. Team Composition:

At the beginning, we decided it would be best to split off on branches and see what we could come up with on our own. After each getting stuck individually, we came back after break and started working in a discord call to collaborate. We would both have the simulator up on screen debugging until somebody found what was needed. Masen brought in the original division code used for understanding how to implement division. Josh ended up writing more of the source code for the two versions. We both ran most of the tests to double check that we got the same cycle counts and then split up the analysis. Masen ran the analysis testing counts to make graphs and Josh made the graphs from the data.
Our original member roles were dispersed as we just started working together at the same time to figure out each part.

2. Method of Implementation:

First the number of rows and columns is stored into two different registers a1 and a2. The matrix is written in this style.



```
.data
matrix:
   .word 10, 10, 5
   .word 6, 2, 2
   .word 2, 3, 2

avg:
   .word 0
```

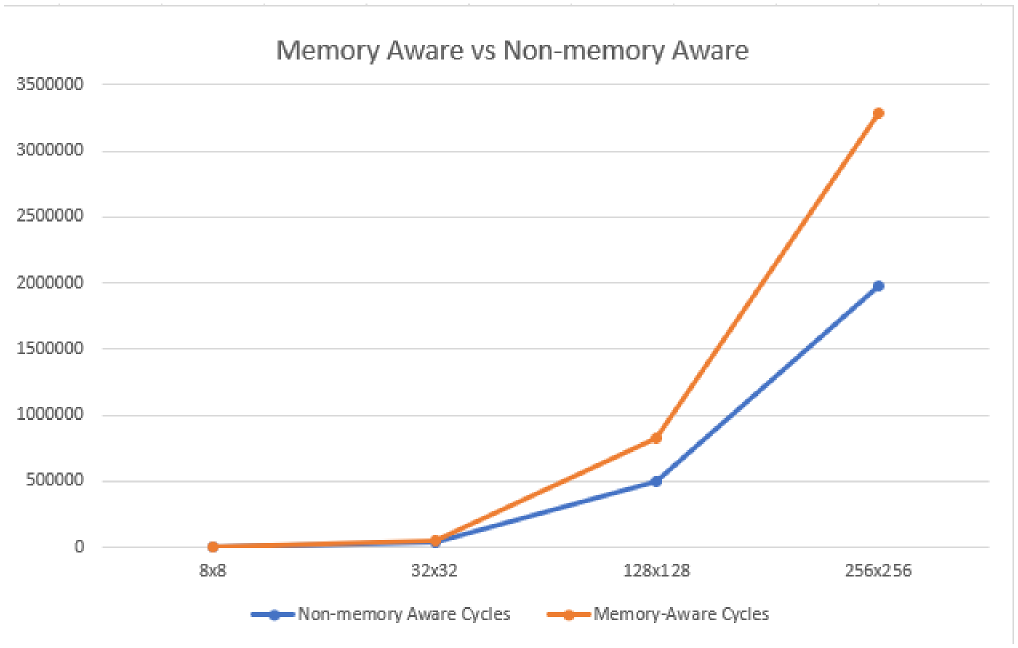
The matrix is loaded into the a0 register; the program goes into avg_columns function where it enters the outer loop and checks if all of the columns of the matrix have been processed, by comparing the current column index in register a4 to the total number of columns in register a5.

The program then moves into the inner loop and calculates the average for a specific column by performing a summation of the matrix elements in the column and then increments the column index. It continues until all rows have been processed.

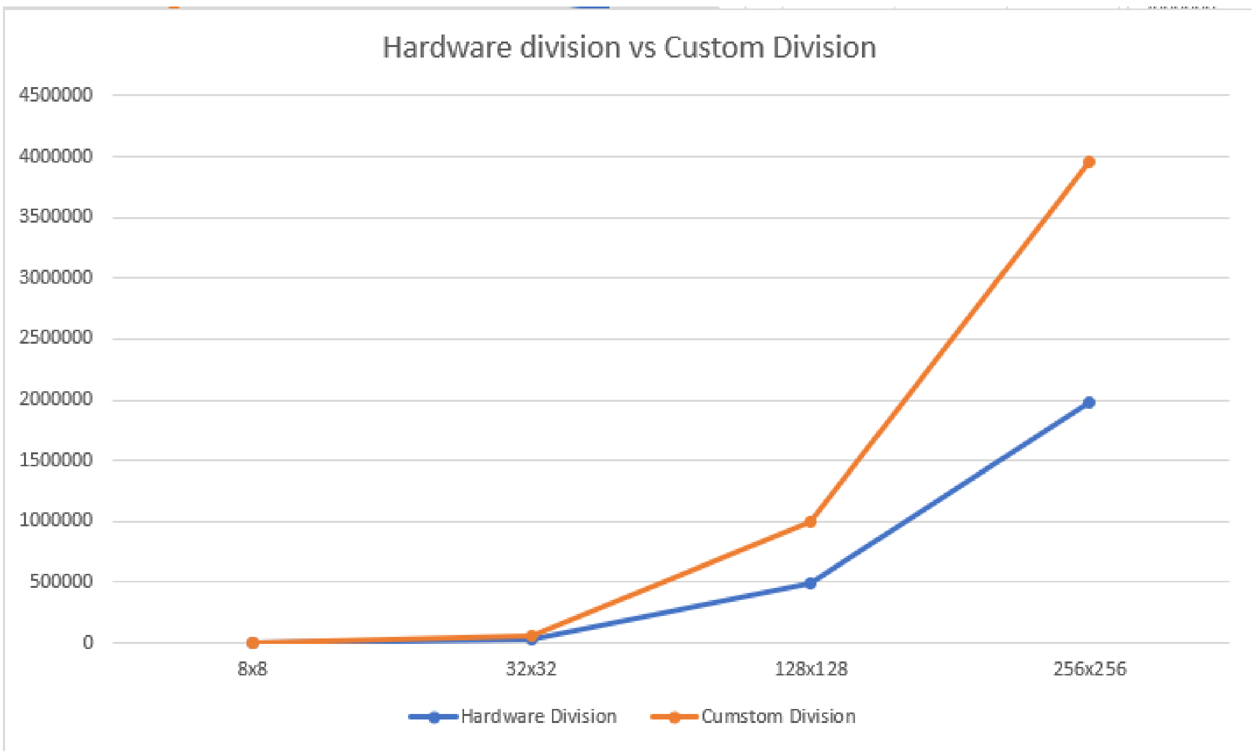
Each column is added by using a series of left shifts to traverse the array. The sum of a column is stored into register a4 then divided by register a5 which stores the number of columns. The result is then stored into 0(a3) where it starts building the array of averages.

Before starting a new column iteration the variables for the inner loop are reset such as the sum of the elements in the column and row index.

3. <u>Analysis of Performance:</u>



The memory aware implementation is faster than the non-memory aware version because the non-memory aware implementation uses an extra variable to calculate the averages and fill the array. This means extra instructions are executed to move the variable into the array holding the averages, causing the number of cycles to increase as the number of columns in the matrix increases.



Hardware division is much faster than our custom division function. We were unable to implement our custom division function into our program, but determined that the function will only execute as many times as there are columns in the matrix. Our function takes 35 cycles to execute while the hardware division takes only 8 cycles. By taking the difference between the two (27) we estimated that if we were able to implement our division function on an 8x8 matrix it would add an additional 216 cycles to both versions of the program. As shown in the graph, as the columns increase each division method increases with the custom function growing at a significantly faster rate.

4. <u>Cache vs Division in Hardware:</u>

We could not get the simulator to give us different values by changing the cache values with block size, sets, and associativity but we think that the costs of adding so much to the cache to the point of quadrupling these values is not better for performance than just adding the hardware division function. We can see that the hardware division does have a major impact especially as the array grows so as that trend continues, it becomes more worth it to implement hardware division. Changing the cache values will affect more in the architecture design too so it would be more worth it if you had to choose one to choose implementing hardware division.

5. Additional Comments:

We started with a division function that we thought could be modified to work with this new program but it was not working well. After a while we tried to write a new division function that used the built in mult and add and whatever else we needed to make it more simple. We got one that seemed to work but didn’t work with our main source. We ended up using the built in divu in our main source code to move forward and make sure we could get the rest done more correctly. The difference between our custom division and the built in function was significant for cycle count. When we got to testing the difference between that cache block and the other variables we got no difference in values. We tested just about every possibility for each variable on optimized and unoptimized code and it never changed the results until the block size was over 32 which is not possible.