UNIVERSITY OF MORATUWA

Faculty of Engineering

Registered Module No: EN4020

ADS SoC project REPORT

Group No: 5

Name	Index No
Vakeesan K.	190643G
Ranasinghe K.K.H	190494A
Shyamal M.A.L	180601T
Wanigarathna A.D.	190660F
Wansooriya W.M.H.O.	190664V

Date of Submission:

December 29, 2023

Department of:

Electronic and Telecommunication Engineering

1 Introduction

Parallel Matrix multiplication is considered a challenging process in most of the hardware accelerators. Our task is to build a matrix multiplication IP using the Zynq processor system where we have to design the PL(hardware accelerator) and the PS(software workflow). We used the ZYBO (Zynq 7000 development board) and Vivado 2018.3 with SDK.

2 Overall system

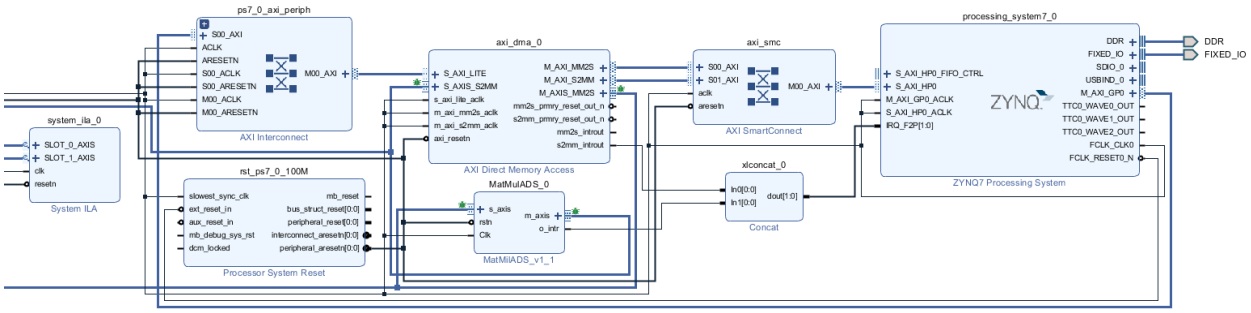

Figure 1: Overall system diagram

There are several third party IPs used in our system such as zynq processing system, system reset, axi interconnect, axi smart connect, AXI DMA, and concatenation block. Our matrix multiplication IP is directly connected with the DMA at the AXI stream interface. then the interrupt comes out from the ip and the DMA's interrupt pins are concatenated together and connected at the PL fabric interrupt of the PS part(zybo's zynq processor can handle 16 bits of interrupt signal in one Fbaric interrupt port). when it comes to the DMA block it can be connected the DDR part from the HP or ACP port of the zynq processor. In our case we use the HP slave port of the processor to connect M AXI S2MM and M AXI MM2S sing the AXI smart connect. Moreover, the basic commands from the processor to the DMA is handled by the AXI lite interface of the DMA block. here the AXI bus and the AXI Lite interfaces are connected using the AXI interconnect module. Furthermore we connect the system ILA ip for debugging purposes.

3 Matrix multiplication IP

As we are going through the SoC design of the matrix multiplication ip gives us the ability to multiply two matrices(both number of rows and columns to be same). This IP module can be customized to provide the scalability in size of the bit as well as the number of rows or columns. Our ip has several sub-modules inside the Top module those are Cache Buffer,

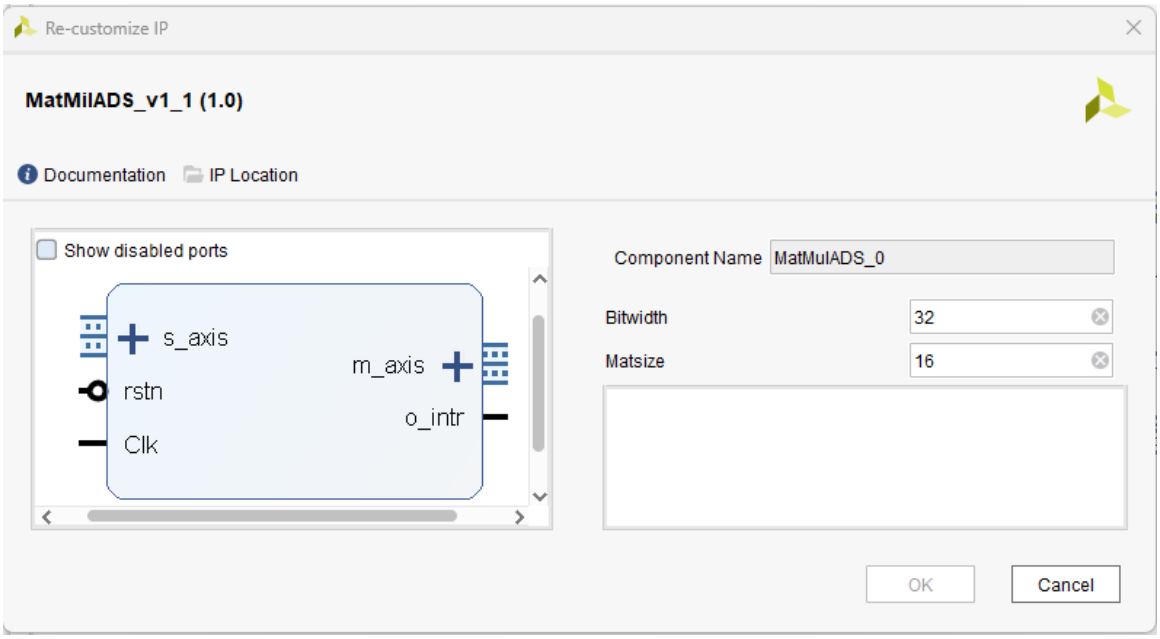

Figure 2: Customization window of the IP

PL controller, AXIout, and processing elements. Since the top module is connected with the DMA through the AXI stream interfaces (both master and slave), the Top module accommodates master signals such as o_ready, o_valid and o_data likewise slave interface has the i_ready, i_valid, and i_data. in addition to that we have the interrupt output signal too.

let's look at each modules one by one. Cache buffer works as a storage element to write and retrieve the data. for the writing process, we actually need the write to enable, Address and the i_data. PL controller will give the write enable signal and the address to store the input data to the buffer. then the cache buffer sends one column of matrix A to all the PE elements and the other input to the processing element is each row from matrix B (A x B = C). then they calculate the vector vector multiplication and store the values back to the one particular row of the cache. finally, we can access the column and give the column of the data back to the AXIout module to deliver the AXI stream data

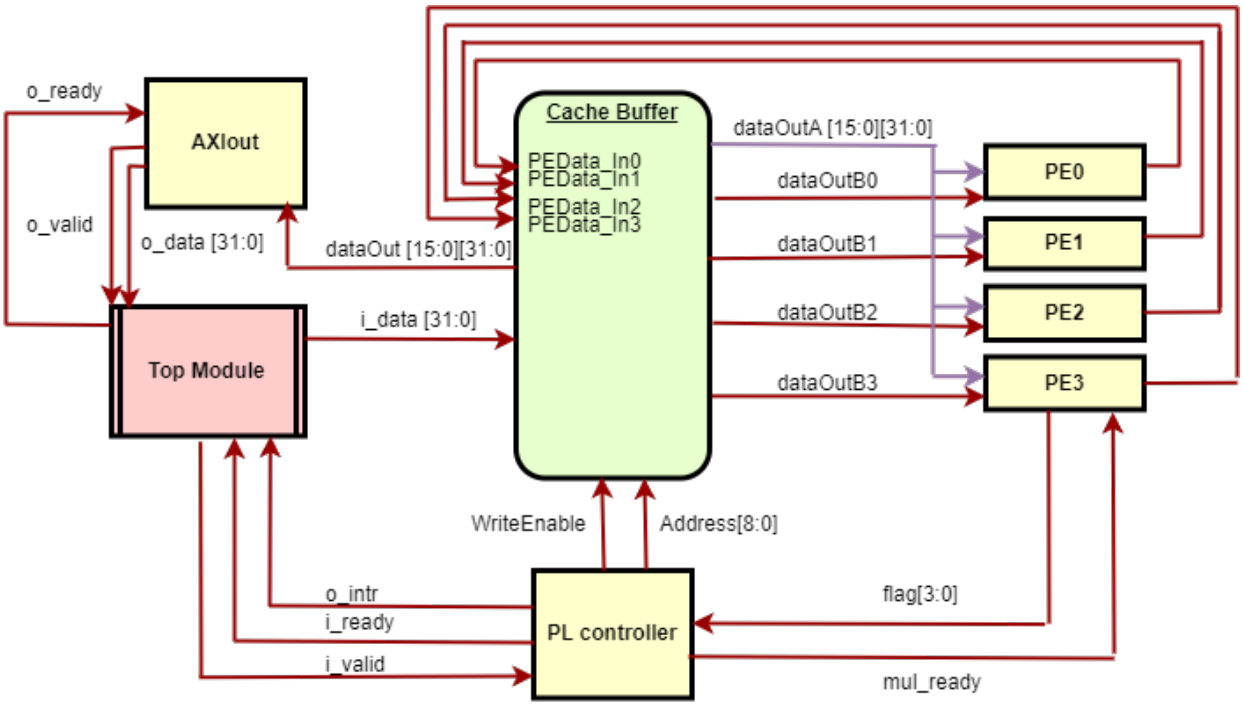

Figure 3: Block diagram of the IP

to DMA. After we send the data we have to receive the next column of the matrix A and that process is handled by the o_interrupt pin. PL controller raises the interrupt when the Flag signal is 4'b1111(this is the concatenated flag signal from each module). PE parts start the process from the mul_ready signal coming out from the PL controller.

4 Verification

When it comes to the verification, First we check our matrix multiplication IP with a test bench file we wrote in system Verilog like other files. Then we wrote a C code in the PS

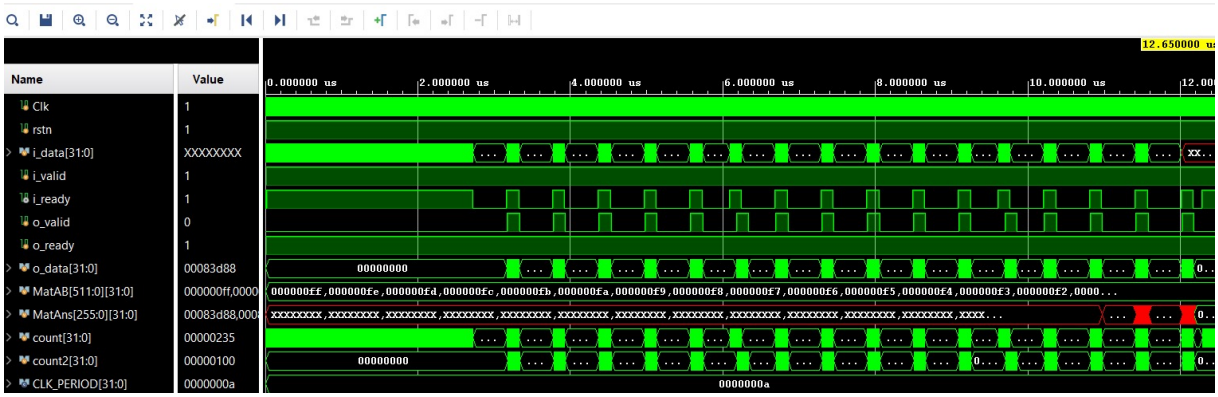

Figure 4: Simulation result of the test bench file

part to verify whether the received data from the PL part is correct or not. Following classical matrix multiplication algorithm is used to get the software result for verification.

```
void multiply(int M1[MAT_SIZE][MAT_SIZE], int M2[MAT_SIZE][MAT_SIZE],
    int M3_SW[MAT_SIZE][MAT_SIZE])
{ int i,j,k;
  for(i=0; i<MAT_SIZE; i++){
    for(j=0;j<MAT_SIZE;j++){
      M3_SW[i][j]=0;
      for(k=0;k<MAT_SIZE;k++)
        M3_SW[i][j]+=M1[i][k]*M2[k][j];
    }
  }
}

```

Listing 1: Classical matrix multiplication in C

after that, we compared the result from the PS(software) and PL(hardware) to raise the error flag to print the position and the error data to the J-tag terminal window (we didn't use the UART for the debugging window). That error checking(verification) C code is given below.

```
for (i=0; i<MAT_SIZE;i++){
    for(j=0;j<MAT_SIZE;j++){
        if((M3_HW[i][j]-M3_SW[i][j])!= 0){
            printf("Non-optimized Matrix mul error at the row %d and
    column %d\n",i+1,j+1);
            printf("Hardware output %d and software output %d", M3_HW[i
    ][j],M3_SW[i][j]);
            break; //can be ignored if you want to write all error data
        }
    }
}

```

Listing 2: Resultant matrix's verification code in C

5 Timing evaluation

We can calculate throughput using the below equation.

$$Throughput = \frac{TotalOperations}{ExecutionTime}$$

The number of operations required to multiply a 16×16 matrix is given by $16 \times 16 \times 16 \times 2$.

The total number of clock cycles the SoC requires to calculate the answer is 1216. This consists of 512 clock cycles for data transfer and 44×16 clock cycles for calculations.

The clock speed of the SoC is 100 MHz.

The throughput is calculated as follows:

$$Throughput = \frac{16 \times 16 \times 16 \times 2}{1216 \times \left(\frac{1}{100 \times 10^6}\right)} = 6.73 \times 10^8$$

Therefore, the throughput is 6.73×10^8.

In addition to the throughput, we checked the time comparison using the XTime API in "xtime_l.h" header file in SDK. first we initialized the timing instances to record the starting and finishing time for both software and hardware processing. then we print the values to the J-tag terminal.

```
1    printf("PS took %.2f us. to calculate the product \n", 1.0*(
   tprocessorEnd-tprocessorStart));
2    printf("PL took %.2f us. to calculate the product \n", 1.0*(tFPGAEnd
   -tFPGAStart));
3
```

Listing 3: Time comparision between PS and PL

Obliviously, our expected hardware processing time is much more significant than the software result($O(s^3)$) due to the DMA block usage under 32-bit transmission and receiving capability. But the throughput analysis can be interpreted that we could get the ultimate benefit from our IP if we are using the bigger size matrices.

6 ANNEX

6.1 Analysis of Processing Element (PE)

In this task, we designed our processing elements as follows: It takes two 1X16 vectors, calculates the dot product between them, and returns the answer. We considered that all input elements are 32-bit signed numbers, and the output element is also a 32-bit signed number. However, when two 32-bit numbers are multiplied, it results in a 64-bit answer. To maintain a constant (32) bit length throughout the system, we only considered the least significant 32 bits in the answer. Thus, each input element in the matrices must be in the range of -32768 to 32767.

The following RTL diagram shows our first approach to the processing element (PE). Here, it includes 16 multiplication units, 15 adders, and 31 32-bit registers. Totally it takes 6 clock cycles to get the dot product from two vectors.

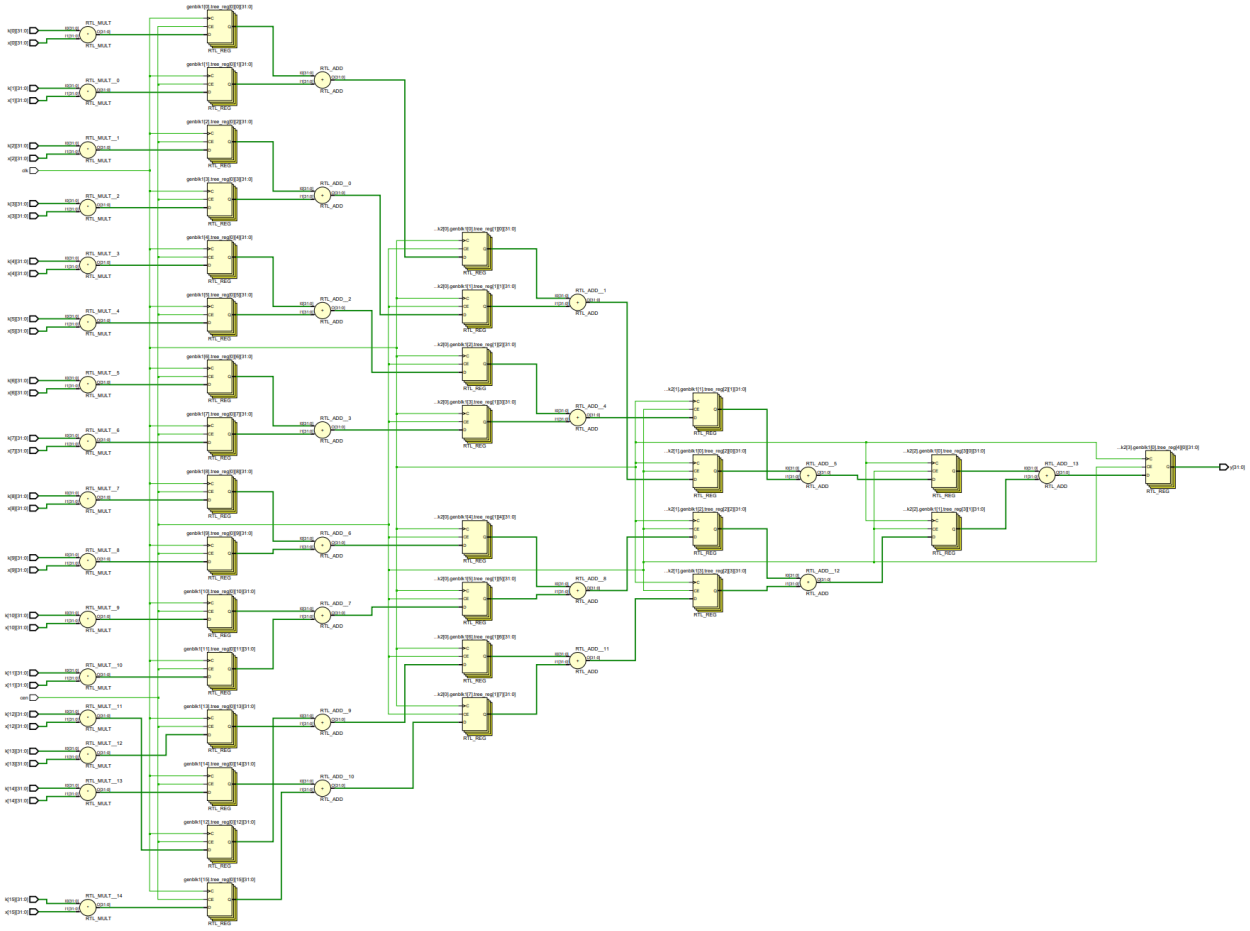

Figure 5: Processing Element before

Since this processing element (PE) incorporates numerous registers and adders, it consumes a significant amount of resources on our FPGA board. Consequently, implementing multiple instances of such PEs on our Zybo board became unfeasible. As a solution, we had to reduce resource utilization by introducing a multiplexer (mux) unit after the multiplication results. The Mux will go through 8 elements and select the inputs to adders using a counter as the switch. The adders are designed as sequential in this method. Thus it takes 8 clock cycles to get the addition of half of the resultant vector. And another adder is used to add the first and second half sums. Hence altogether it takes 11 clock cycles to give the output. Now this module has 16 multiplication units, 3 32-bit adders, 21 32-bit registers.

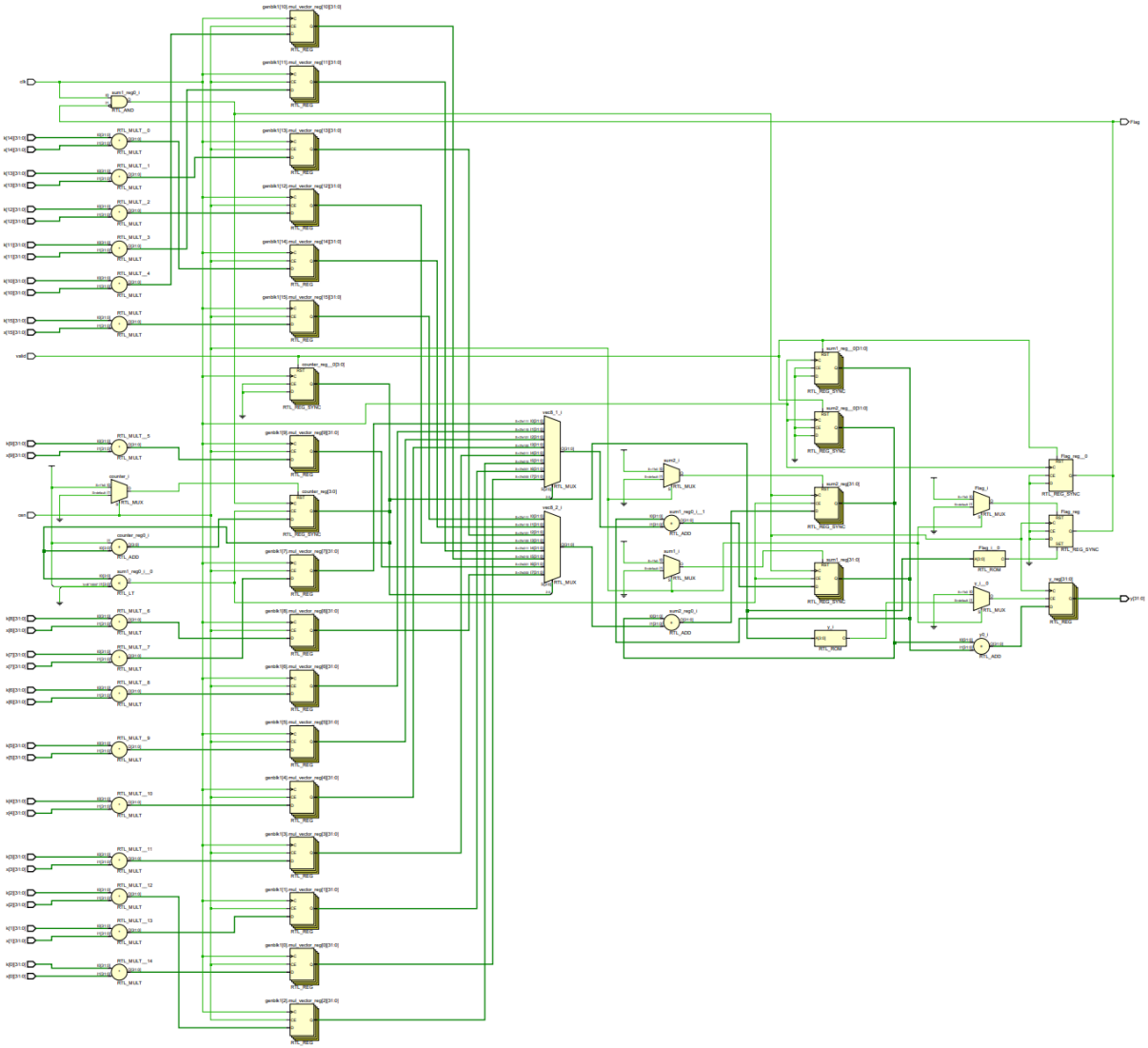

Figure 6: Processing Element after

6.2 Analysis of Arbitrary dimensions

In this task, our PEs are designed to work only with 16x16 matrices. But the user should have the freedom to give any desired size matrix as the input. To overcome this issue we used Strassen's Algorithmic approach. Strassen's Algorithm is a method for matrix multiplication that reduces the number of required multiplications by using a divide-and-conquer approach. It breaks down matrices into smaller submatrices and employs recursive formulas, decreasing the multiplication count from eight to seven. While it theoretically improves time complexity for large matrices, in practice, its overhead may limit its efficiency compared to traditional methods. The following figure illustrates how Strassen's algorithm works for a 4x4 matrix.

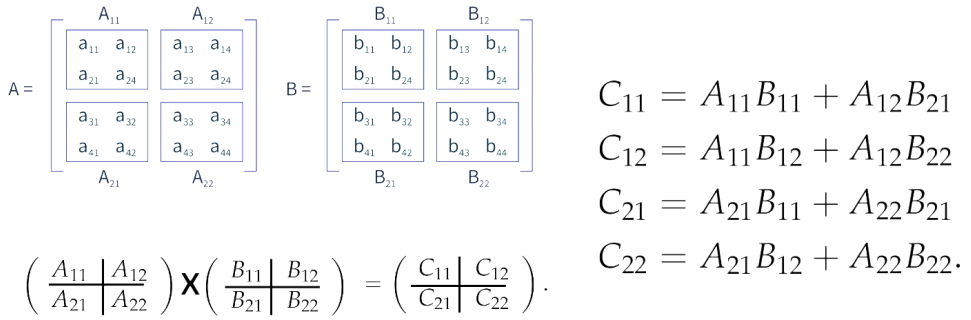

Figure 7: Strassen's algorithm

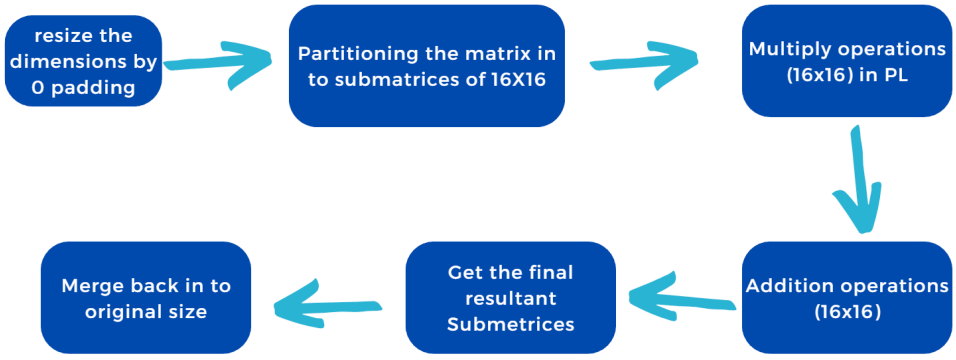

Figure 8: Overall Process for arbitrary dimensional matrices

In our SoC architecture, a 16x16 sub matrix approach is employed. Within the Processing System (PS) component, any input matrix is zero-padded to ensure that its dimensions are divisible by 16. Subsequently, the matrix is partitioned into 16x16 sub matrices. Each pair of corresponding (according to strains algorithm) sub matrices is transmitted to the Programmable Logic (PL) section for multiplication. Following this, the addition

operations are performed sequentially back in the PS component. The resulting sub matrices are then merged to obtain the final output, and any padded zeros are removed in the process.

Consider the time to multiply two 32X32 matrices. If we improved our hardware (PE) architecture design and storage units by adding 16 more multipliers, 2 more muxes and 2 32-bit adders. Comparing with our current approach, It will take following number of clock cycles for the multiplication process in the PE.
with improving PE

$$= 12 \times 32 \times 32 = 12288$$

with Strassen's algorithm

$$= 11 \times 16 \times 16 \times 4 = 11264$$

since all the 16x16 addition operations are happening in the PS part we can neglect the time for that. But when considering the data transfer timings. improved hardware method might do the job faster since the current method has more data transmissions between Ps and PL.

6.3 Analysis of using Block RAM instead of Cache Buffer

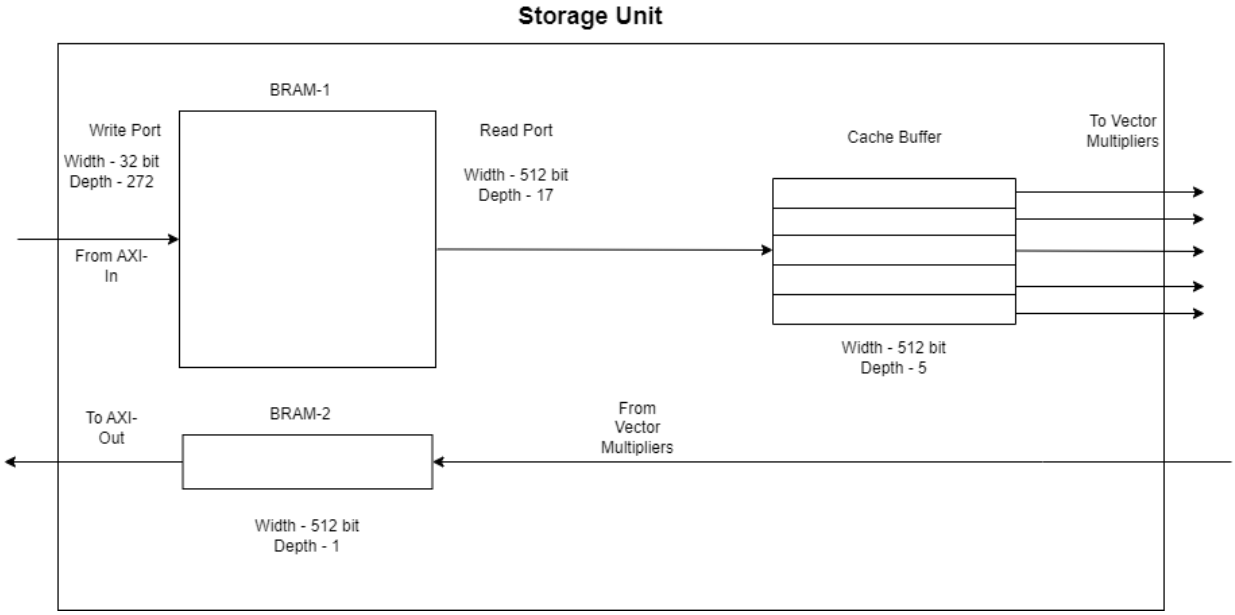

Figure 9: Alternative Storage Unit Implementation

Instead of using a register file as a cache buffer, we can implement above storage unit.

It utilizes the block memory that is available in the FPGA. In the above design we use one, two port block RAM and one single port block RAM. In the dual port block ram, one port is used to read and other to write. According to our requirement in the BRAM-1, we have to use 32-bit wide write port and a 512-bit wide read port. In the BRAM - 1, the transpose of Matrix B and one line of Matrix A will be stored at a time. In the BRAM - 2, one line of the answer matrix will be stored. Since our PEs require 5 rows simultaneously, had to implement a small register bank to store them temporarily. Using the above storage unit simplifies the synthesized design because it uses idle BRAM tiles and does not need to synthesize complex register structures and a big multiplexer to address them. The above storage unit utilizes 20% of available BRAM tiles. But above dual port ram spends two clock cycles for read and write operations. So using the above module as a substitute for a cache buffer will increase the time that it takes to generate an answer.