CDA 4253 FPGA System Design Final Project

My Nguyen[1], Osniel Quintana[2], Andys Rivero Castillo[3]
[1]University of South Florida: Department of Computer Science and Engineering, Tampa, USA, mynguyen@mail.usf.edu
[2]University of South Florida: Department of Computer Science and Engineering, Tampa, USA, osnielq@mail.usf.edu
[3]University of South Florida: Department of Computer Science and Engineering, Tampa, USA, andyrivero@mail.usf.edu

***Abstract*—This paper introduces the implementation of common matrix operations on the Basys3 FPGA boards. The FGPA implementation is used to test the matrix operations, which are matrix-to-matrix multiplication and addition. Then, the results from the FPGA implementation will be used to evaluate the performance against a software implementation (C program) of the same operation.**

***Index Terms*—matrix, operations, Basys3 FPGA, FPGA boards, performance, software implementation, hardware implementation, multiplication, addition, matrix-to-matrix, columns, rows, matrices, dimension, VHDL, optimization, design, speed, BAUD rate, clock cycles, binary, BCD, ASCII**

I. Introduction

The implementation of common matrix operations on the Basys3 FPGA boards are matrix-to-matrix multiplication and addition. The general objective of this project is to create a hardware implementation on the Basys3 FPGA board to do the matrix operations. In addition, the FPGA implementation needs to be optimized and evaluated. The FPGA implementation's performance results will be compared with the software implementation's performance results. The given software implementation of the matrix operations is written in C. This paper gives a general overview of the FPGA implementation, a brief explanation of matrix operations, the design architecture and implementation, different optimization methods, results of the two implementations, and the conclusion with future improvements.

II. General overview

Users can input the elements of the input matrices on the PuTTY terminal (host terminal). Users will input the wildcard (*) for matrix multiplication and the addition symbol (+) for matrix addition. Elements of the input matrices are unsigned numbers and 8-bit wide. The input matrices can be either one or two dimensions and the size are arbitrary but fixed. Data inputs are defined in hexadecimal format in an external file to initialize the block RAM. Once all values of the matrices are inputted by the user, the FPGA will compute the matrix operation and the results will be shown on the PuTTY terminal. The results are displayed on the host terminal through UART transmitter, which are in hexadecimal format. Data outputs are 16-bit wide. The display is organized in a matrix format for both inputs and outputs.

III. Matrix Operations

For matrix-to-matrix multiplication, there is an assumption that the number of columns of matrix A is the same as the number of rows of matrix B. For an example, matrix A is 3x2 and matrix B is 2x3, where the columns of matrix A is 2 and rows of matrix B is 2. For matrix-to-matrix addition, there is an assumption that both input matrices have the same dimension. For an example, matrix A is 3x2 and matrix B is 3x2, where the dimensions are the same.

A. Matrix-to-Matrix Multiplication

Matrix-to-matrix multiplication is a bit more complicated. Multiplication of matrix A and matrix B will be row times column, *[m x n]*, where rows are defined by *m* and columns are defined by *n*. To better understand matrix multiplication, for a 2x2 matrix, the first position in the first row of matrix A will be multiplied to the first position in the first column of matrix B. Then the second position in the first row of matrix A will be multiplied to the second position in the first column of matrix B. This process continues until all the rows and columns of matrices A and B are calculated with the final results stored in matrix C. Seen in the figure below, matrix A gets multiplied to matrix B, resulting in matrix C. For an example, a 2x2 matrix A has values 1, 2, 3, and 4 and a 2x2 matrix B has values 5, 6, 7, and 8. The resulting matrix C will have values 19, 22, 43, and 50 by using the given formula in Figure 1.

$$A[m\,x\,n] * B[n\,x\,k] = C[m\,x\,k]$$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,k} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,k} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,k} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,k} \end{bmatrix}$$

$$c_{i,j} = \sum_{x=1}^{n} a_{i,x} b_{x,j}$$

Fig. 1: Matrix-to-matrix multiplication of matrices A and B results in matrix C.

B. Matrix-to-Matrix Addition

Matrix-to-matrix addition is quite simple and straightforward. The values are added from the same location of each matrix. For a 2x2 matrix, the first position in matrix A will be added to the first position in matrix B and the process

continues for the rest of the two matrices. Seen in the figure below, matrix A gets added to matrix B, resulting in matrix C. For an example, a 2x2 matrix A has values 1, 2, 3, and 4 and a 2x2 matrix B has values 5, 6, 7, and 8. The resulting matrix C will have values 6, 8, 10, and 12 by using the given formula in Figure 2.

$$A[m\ x\ n] * B[m\ x\ n] = C[m\ x\ n]$$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} + \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,n} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,n} \end{bmatrix}$$

$$c_{i,j} = a_{i,j} + b_{i,j}$$

Fig. 2: Matrix-matrix addition of matrices A and B results in matrix C.

IV. Design Architecture

The architecture, seen below in Figure 3, includes a transmitter, FSM, and RAM. The transmitter has a UART receiver and transmitter as well as format conversion units for ASCII-to-binary, binary-to-BCD, and BCD-to-ASCII. The FSM, which is the interpreter, holds the matrix operations and interacts with the transmitter and RAM. RAM is split into two blocks to hold the two matrices and it interacts with the FSM.

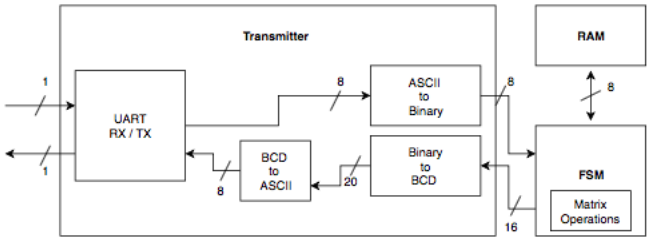

Fig. 3: This FPGA implementation's architecture includes a transmitter, FSM, and RAM.

V. Design Implementation

The three main components for this FPGA implementation are the transmitter (where the UART and format conversion units sit), FSM (where the matrix operations sit), and RAM.

A. Transmitter

The transmitter has two modes, one for reading and one for writing. The reading mode accepts keyboard inputs and presents the inputs as bytes and sends the data back to the host terminal. The writing mode takes the bytes and sends it to the host terminal. Shown in above Figure 3, the transmitter has the UART units (receiver and transmitter) and the format conversion units (ASCII-to-binary, binary-to-BCD, and BCD-to-ASCII).

1) UART Units: The BAUD rate is 9,600 bits per second, thus it takes 1/9,600 seconds to send one bit of data. At 9,600 baud rate, the time between middle bit and another is 1/9,600 seconds with a clock frequency of 100 MHz.

a) Receiver: The UART receiver has two components, the receiver unit and the buffer unit. The purpose of the UART receiver is to get inputs from the host terminal. The UART receiver receives one-bit data in serially from the host terminal and output eight bits in parallel. It then sends the eight bits to the ASCII-to-binary converter. The eight bits are stored in a queue and the queue resets every time a space is detected from the host terminal. Shown below in Figure 4 is how the UART receiver works.

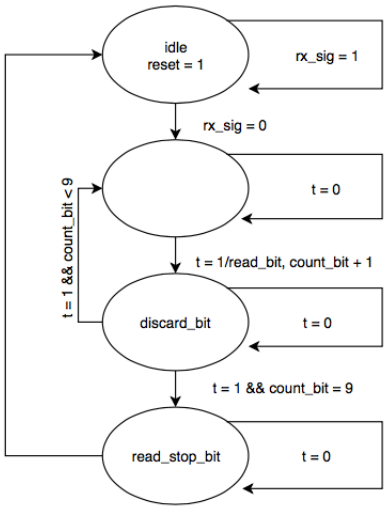

Fig. 4: The UART receiver finite state machine.

b) Transmitter: The UART transmitter has two components, the transmitter unit and the buffer unit. The purpose of the UART transmitter is to receive eight bits in parallel from the BCD-to-ASCII converter and output one-bit serially the results back to the host terminal. Shown below in Figure 5 is how the UART transmitter works.

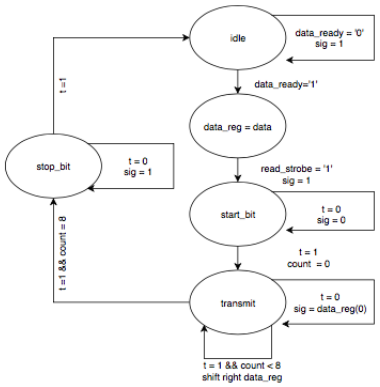

Fig. 5: The UART transmitter finite state machine.

B. Format Conversion Units

To successfully receive inputs from the host terminal, do matrix operations, and output the results back to the host terminal, some format conversions are needed.

a) ASCII-to-Binary: When data is received from the host terminal, the numbers are received as ASCII. Thus, an ASCII-to-binary conversion is needed for the FSM to do the matrix operations. This converter takes in eight bits from the UART receiver and send it over to the FSM for the matrix calculations.

b) Binary-to-BCD: Once the FSM completes the matrix operations, it sends back 16 bits to the binary-to-BCD converter. This converter converts the binary results from the calculations to BCD with 20 bits, sending the results over to the BCD-to-ASCII converter.

c) BCD-to-ASCII: From there, this converter takes in the 20 bits and converter the results back to eight bits prior to sending it through the UART transmitter to reach the host terminal. The results will need to be in ASCII.

C. FSM

The FSM is the interpreter that computes the result for the matrix operations. The implementation of FSM uses switch cases where each case represents different states. The switching between states is synchronized on the rising edge of the clock. Data is read and sent on the rising edge. On the first rising edge, the FSM sends an address to RAM. On the falling edge, RAM has the address. On the next rising edge, the FSM gets the number from RAM at the address that RAM received on the falling edge. This procedure is possible due to RAM optimization (found in section 7A).

1) Matrix Calculations: The FSM is responsible for the matrix calculations. First, the FSM gets the matrix size of matrix A and matrix B. Then, the input values for matrix A and matrix B are written to RAM starting at address 0. The indexing of the address happens based on the following formula

$address = i * totalcols + j$

where *i,j* are the location of the elements in the matrix. Following the input values is the operand, either the wild-card (*) for multiplication or the addition symbol (+) for addition. On the next clock cycle, the FSM does the calculation based on the operand and outputs the results.

a) Multiplication: For matrix multiplication, the FSM obtains the results at *i,j* by multiplying the vectors at row vector *i* on matrix A and column vector at *j* on matrix B. A third variable *k* is used to index. However, to implement matrix multiplication in VHDL, a multi-dimensional array is needed. But given the constraints on RAM and the fact that multi-dimensional arrays are declared in RAM, then this multiplication method cannot be implemented due to waste of space. Thus, the FPGA implementation manually "un-rolls" the *for* loops seen in Figure 6 and only keeps track of the last calculated product. Therefore, this implementation only need 16 bits of memory to hold the matrix multiplication result. Below in Figure 6, once *k* reaches colsA, the number calculated thus far is sent to the UART transmitter and the FSM waits for an acknowledgement signal to do the next operation. Once all operations are done for matrix multiplication, the FSM reset to do another matrix operation.

```
for i = 0 to rowsA
    for j = 0 to colsB
        for k = 0 to colsA
            prod(i,j) = prod(i,j) + A[i,k]*B[k,j]
```

Fig. 6: This algorithm multiplies two matrices.

b) Addition: For matrix addition, RAM is in the reading mode. The indexing of the address is the same as matrix multiplication and it happens based on the following formula

$address = i * totalcols + j$

where *i,j* are the location of the elements in the matrix. The FSM fetches matrix A's inputs at *i,j* and matrix A's inputs at *i,j* simultaneously thanks to RAM optimization (found in section 7A). On the half clock cycle, matrix addition occurs and the results of the matrix addition is sent to UART transmitter. Upon receiving an acknowledgement signal, the next operation happens. Once all operations are done for matrix addition, the FSM reset to do another matrix operation.

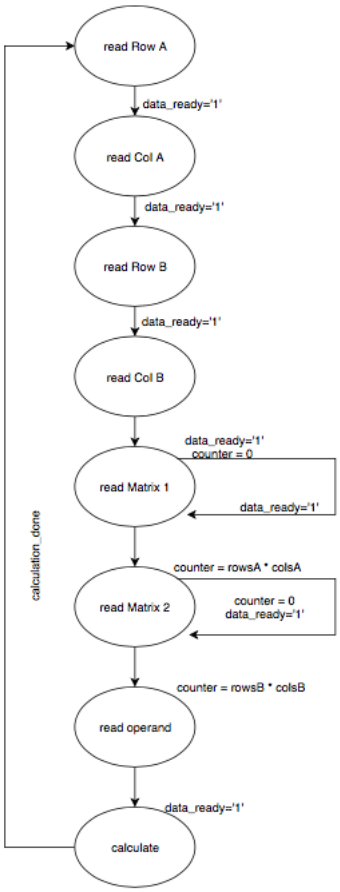

Fig. 7: The interpreter, FSM, contains the two matrix operations.

D. RAM

The size of RAM is 230,400 bytes of memory. The chosen memory for this FPGA implementation is block RAM or BRAM. BRAM is generally used for RAM descriptions with synchronous read/write. The three different ways to infer RAM are read-first mode, write-first mode, and no-change mode. In this FPGA implementation, a no-change mode is used on the falling edge of the clock. The normal no-change mode uses the rising edge of the clock but to optimize the speed of this FPGA implementation, the falling edge is a better choice seen below in Figure 8. To read more on RAM optimization, go to section 7A.

```
21
22  -- no-change BRAM scheme
23  architecture ram_module_ar of ram_module is
24      ---------------------------------------------------------
25      subtype WORD is std_logic_vector (WORD_WIDTH - 1 downto 0);
26      type RAM_TYPE is array (0 to (2**ADDR_WIDTH) - 1) of WORD;
27      ---------------------------------------------------------
28      signal ram : RAM_TYPE;
29  begin
30      process (clk)
31      begin
32          if falling_edge (clk) then
33              if we = '1' then
34                  ram (to_integer (unsigned (addr))) <= data_in;
35              else
36                  data_out <= ram (to_integer (unsigned (addr)));
37              end if;
38          end if;
39      end process;
40  end ram_module_ar;
```

Fig. 8: RAM uses a no-change mode on falling edge of clock.

VI. Execution

The FPGA execution includes PuTTY instructions and LED signals explanations.

A. PuTTY Instructions

Prior to using the PuTTY terminal, the user needs to check the box, "Implicit LF in CR" in settings, as seen below in Figure 9. Using the PuTTY terminal, the user will input the wild-card (*) for matrix multiplication and the addition symbol (+) for matrix addition. Users can either use newline or space for inputting the values of the matrices.

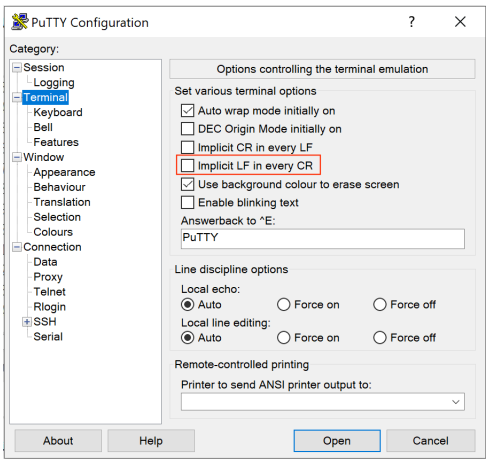

Fig. 9: The PuTTY terminal setting needs to be implicit LF in every CR.

B. LED Signals

For every state in the FSM, shown in Figure 4, the LEDs have different output signals. The purpose LEDs is mainly used for debugging.

VII. Optimization

Optimization of this FPGA implementation can be seen in three different areas: block RAMs, clock cycles, and UART buffer.

A. Block RAMs

The Random-Access-Memory (RAM) is responsible for storing data. For either matrix multiplication or addition, the address of each data is computed as $address = i * totalcols + j$. To optimize the memory, two block RAMs are used instead of one block RAM. The design of using two block RAMs is to hold matrix A in one block RAM and matrix B in the other. The purpose of using two block RAMs is to allow the FSM to read both matrices at the same time. By doing so, the speed was achieved and improved because the FPGA implementation reads inputs from both matrices instead of reading one input at a time. Thus, space was sacrificed to achieve speed.

B. Clock Cycles

To optimize the FPGA implementation, the FSM utilizes the rising edge and RAM uses the falling edge. The FSM is optimized by providing the data on the rising edge to RAM. Therefore, on the falling edge, RAM access data addresses from the FSM, optimizing the clock cycles needed to access data addresses. Normally, RAM access data addresses on rising edges which takes two clock cycles to get each address.

C. UART Buffer

Initially, the FPGA implementation uses a First-In-First-OUT (FIFO) buffer, however, the UART buffer is not needed in this implementation because the data is waiting for the buffer. The purpose of the buffer is to hold the input and output data. In this case, the outputting data was slower than the receiving data. The serial port communication is slower than speed of calculating the elements in matrix. If a buffer was at the speed of the results, then the speed of the FPGA implementation will be optimized. In this FPGA implementation without a buffer, a feedback signal is used to avoid corruption and losing bytes along the way to outputting the results.

VIII. Results

The results differ from hardware to software implementation. Several test cases were chosen at random to compare the speed of the two implementations.

A. Hardware Implementation

Given that there is a bottleneck in the RS232 protocol (USB communication), the clock cycles are calculated for matrix multiplication and matrix addition are seen below.

1) Multiplication: As seen in Figure 1, matrix multiplication is between matrix A and matrix B where the number of columns in matrix A is equal to the number of rows in matrix B. The matrix size is *[m x n] * [n x k]*. Theoretically, the number of clock cycles for matrix multiplication can be calculated as *[m * n * k]* because the matrix multiplication formula is *[m x n] * [n x k]*. For an example, for a 2x2, due to the RAM optimization, a total of eight (2 * 2 * 2 = 8) cycles is needed to complete the multiplication operation. Without the RAM optimization, a total of 16 (2 * 2 * 2 * 2 = 16) cycles is needed to complete the multiplication operation. The computed time for a 2x2 matrix multiplication is 80 nanoseconds because the clock runs at 100 MHz and it takes eight cycles, thus 10 nanoseconds per cycle.

2) Addition: As seen in Figure 2, matrix addition is between matrix A and matrix B or two matrices of size *m x n*. Theoretically, the number of clock cycles for matrix addition can be calculated as *[m * n]* because the matrix addition formula is *[m x n] + [m x n]*. For an example, for a 2x2, due to the RAM optimization, a total of four (2 * 2 = 4) cycles is needed to complete the addition operation. Without the RAM optimization, a total of eight (2 * 2 * 2 = 8) cycles is needed to complete the addition operation. The computed time for a 2x2 matrix addition is 40 nanoseconds the clock runs at 100 MHz and it takes four cycles, thus 10 nanoseconds per cycle.

B. Software Implementation

The software implementation is given by the professor.

1) Multiplication and Addition: According to the software implementation, a *clock library* is used. However, the clock library is not precise because for both operations on a 2x2 matrix, it takes 21,000 nanoseconds to complete the matrix operations.

C. Hardware vs. Software Implementation

Below are three different test cases chosen at random that were evaluated to do a comparison between the hardware implementation verses the software implementation.

1) Multiplication and Addition: 2x2: According to the FPGA implementation, it takes 80 nanoseconds for a 2x2 matrix multiplication and 40 nanoseconds for a 2x2 matrix addition. Comparing to the given software implementation, it takes 21,000 nanoseconds for both matrix operations.

*2) Multiplication: 300x100 * 100x50:* Theoretically, the FPGA implementation for matrix multiplication is $1.5 * 10^6$ nanoseconds versus $3.10 * 10^6$ nanoseconds on the software implementation. Therefore, the FPGA implementation is two times faster the software implementation.

3) Addition: 300x100 + 300x100: Theoretically, the FPGA implementation for matrix addition is $3.0 * 10^5$ nanoseconds verses $2.5 * 10^6$ nanoseconds on the software implementation. Theoretically, the FPGA implementation for matrix addition is 84 times faster than the software implementation.

IX. Conclusion

In conclusion, the FPGA implementation is faster than the C program because it uses a low-level design. Usually, low-level access is faster than high-level access. An improvement for the future includes a backspace function allowing users to delete their inputs if they made mistake on the PuTTY terminal. Another improvement would be that when an input size is too large, matrix multiplication cannot happen in one cycle. The whole FPGA implementation was created from previous knowledge and the UART was built from scratch without references from the textbook or open sources from the Internet.