Timothy Andersen
Maharshi Shah
David Ton

FPGA DESIGN - CDA 4253

Final Project Report

Matrix Operation with Serial Communication on a FPGA

Design and Architecture

For this project we start out with data being sent from a terminal program connected to the port of the FPGA circuit. For this we type in the data specifically and it is sent through a UART receiver by using the FIFO method. This data will be describing the matrix dimensions and the numbers inside both matrices and sending them to the interpreter module.

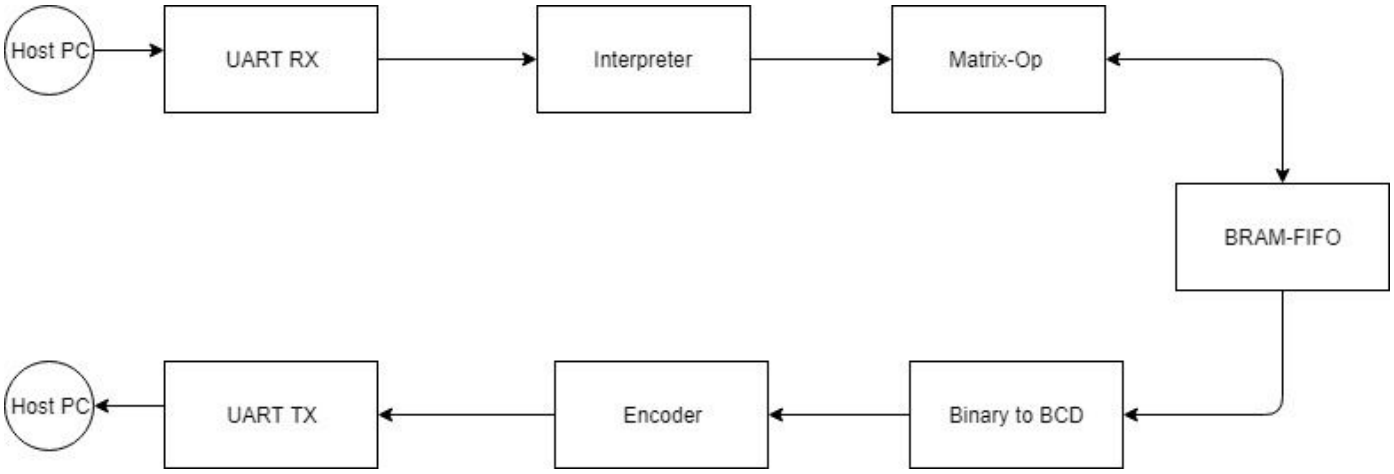

Figure 1: Block Level Design

Design Assumption

The project requires us to use a *fixed but arbitrary* matrix for operation. Hence, we decided to keep the dimensions of the matrix set beforehand and perform the operation (multiply or add) according to the input. This means that our design would accept only a set number of elements for the two matrices, depending on the dimension of matrix A and B set in the matrix multiplication and addition packages. The number of rows and columns would be fixed in order to create the desired dimensions for the input matrices.

Interpreter Unit:

The interpreter module will be analyzing the output of the UART RX and organizing it in a way that the matrix op module will be able to understand. To do this we created a finite-state machine (figure 3) that will expect data in a certain order. The first state will be determining if you need to add or multiply and will go down the one of the two routes depending on a 1 or 0 from the previous state. Each route will have 3 states, each expecting input for the numbers inside of matrix A and B. In figure 4 we can see the code for state_add_A_in, which is very similar to the states state_add_B_in, state_A_in and state_B_in. The number of inputs the state is expecting will be determined by the dimensions that have been pre-defined in the matrix op module. After the user will hit the new line key (ascii 10) at the end of each route, state_B_in and state_add_B_in respectively, those matrices will be sent to the last states in each route, state_finished and state_add_finished which will have matrix A and matrix B make them into the outputs shown in figure 2. The reason for the two routes is to separate the variables for add and multiply for the matrix op in figure 2 rather than having a control input which complicates things.

```
add_mult <= '0';
add_matrix_1_out <= add_matrix_A;
add_matrix_2_out <= add_matrix_B;
```

```
add_mult <= '1';
mult_matrix_1_out <= matrix_A;
mult_matrix_2_out <= matrix_B;
```

Figure 2: The end of the two states, state_add_finished and state_finished respectively

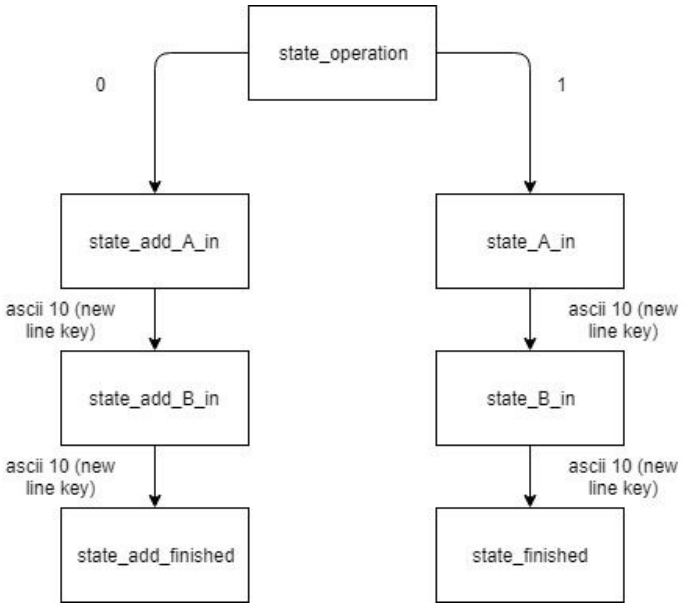

Figure 3: Finite-State Machine (FSM) for interpreter

```
elsif (state_reg = state_add_A_in) then

            if (A_in_finished = '0') then
                            if (unsigned(parallel_in) >= 48 and unsigned(parallel_in) <= 57) then
                                --temp <= temp + unsigned(parallel_in) * position;
                                --position <= position * 10;
                                temporary_array(digits_int) <= unsigned(parallel_in) - 48;
                                digits_int <= digits_int + 1;
                            elsif (unsigned(parallel_in) = 32) then
                                temp <= (others => '0');
                                --position <= "0000000000000001";
                                for k in 0 to 2 loop
                                    --temp = temp + temporary_array(k) * 10 ^ (digits_int - k - 1)
                                    temp <= temp + (temporary_array(k) * 10**(2 - k));
                                end loop;
                                add_matrix_A(i)(j) <= temp;
                                digits_int <= 0;
                                j <= j + 1;

                                if (j < num_cols) then
                                    --Do nothing
                                else
                                    j <= 0;
                                    i <= i + 1;
                                    if (i < num_rows) then
                                    --Do nothing
                                    else
                                        i <= 0;
                                        A_in_finished <= '1';
                                    end if;
                                end if;
                              end if;

                              else
                                if (unsigned(parallel_in) = 10) then
                                    state_reg <= state_add_B_in;
                                end if;
                            end if;
```

Figure 4: Code for state_add_A_in

Matrix Operation unit:

The matrix op module is composed of a top module with two packages, one for matrix addition(mat_add.vhd) and one for matrix multiplication(mat_mult.vhd). These packages act as usable libraries and define the size and operation on the 8 bit matrices A and B. They can be called as a function in another vhdl module (that acts as a main file) and uses the properties of that package to perform the appropriate operation.

For our project, the package files are built to do the following things:

- Create a data structure of a matrix since the program needs to understand it is an array of arrays. The number of tuples is the number of rows and the number of elements inside the tuple acts as the number of columns. Each tuple defines the elements inside the first row.

- Use synthesizable for loops inside the package definition to perform the multiplication or addition based on matrix properties i.e. for multiplication, the number of columns of matrix A and the number of rows of matrix B need to be the same. As for addition, the dimensions of both matrices need to be equal. This makes the design more controlled and the code more comprehensible.

Below is the code used for the packages created in vhdl:

- Matrix multiplication package:

```
type t11 is array (0 to numcol_1 - 1) of unsigned(7 downto 0);--
type t1 is array (0 to numrow_1 - 1) of t11; --4*3 matrix
type t22 is array (0 to numcol_2 - 1) of unsigned(7 downto 0); --
type t2 is array (0 to numrow_2 - 1) of t22; --3*5 matrix
type t33 is array (0 to numcolC - 1) of unsigned(15 downto 0);
type t3 is array (0 to numrowC - 1) of t33; --4*5 matrix as outpu

function matmul ( a : t1; b:t2 ) return t3;

end mat_mul;

package body mat_mul is
function matmul ( a : t1;
                  b : t2 ) return t3 is
    variable i,j,k : integer:=0;
    variable prod : t3:=(others => (others => (others => '0')));

begin
for i in 0 to (numrow_1 - 1) loop --(number of rows in the first
    for j in 0 to (numcol_2 - 1) loop --(number of columns in the
        for k in 0 to (numrow_2 - 1) loop --(number of rows in th
            prod(i)(j) := prod(i)(j) + (a(i)(k) * b(k)(j));
        end loop;
    end loop;
end loop;
return prod;

end matmul;

end mat_mul;
```

Figure 5: mat_mult package code

- Matrix addition package:

```
type t11 is array (0 to numcol_1 - 1) of unsigned(7 downto 0);
type t_1 is array (0 to numrow_1 - 1) of t11;
type t22 is array (0 to numcol_2 - 1) of unsigned(7 downto 0);
type t_2 is array (0 to numrow_2 - 1) of t22;
type t33 is array (0 to numcolC - 1) of unsigned(15 downto 0);
type t_3 is array (0 to numrowC - 1) of t33;

function matadd ( a : t_1; b: t_2) return t_3;

end mat_add;

package body mat_add is
function matadd (a : t_1;
                 b : t_2 ) return t_3 is
    variable i,j : integer:=0;
    variable sum :t_3:=(others => (others => (others => '0')));

begin
for i in 0 to (numrow_1 - 1) loop
    for j in 0 to (numcol_1 - 1) loop
         sum(i)(j) := sum(i)(j) + (a(i)(j) + b(i)(j));

    end loop;
end loop;
return sum;

end matadd;
```

Figure 6: mat_add package

The simulation waveform for the matrix operation is below. The dimensions of the multiplication matrices in the testbench are (6x7) and (7x5) for A and B respectively , resulting in a 6x5 matrix C. The matrices used for addition are of the size 2x2 each.

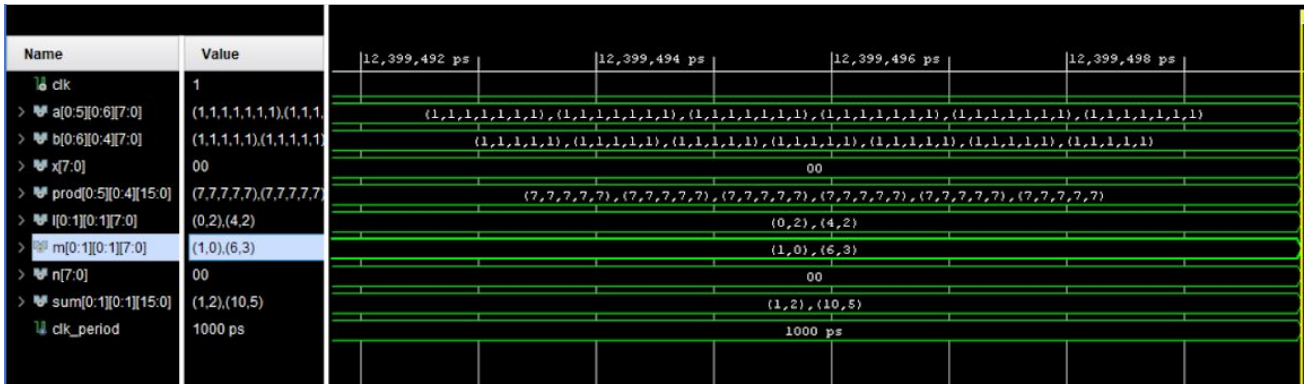

Figure 7: Simulation Waveform for Matrix Operation

BRAM based FIFO:

A BRAM based FIFO can be instantiated using the IP COREGEN from the vendor. This helped us straight away implementing a FIFO unit without typing in extra logic.The BRAM-FIFO used is a Common Clock BRAM using the FIFO generator in the IP list of block designs.Below is the Block Design created for a Common Clock BRAM based FIFO using IP COREGEN in Vivado.

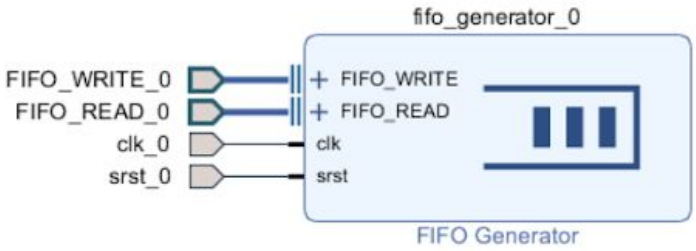

Figure 8: BRAM based FIFO

Once the block design was generated and validated, a wrapper module was created that could be used and edited for our design.The wrapper created using the block design generates the logic required with appropriate empty and full signals that can be directly used without having to manually write the external logic for it.

Instantiation was a better option rather than writing the fifo logic and inferring a BRAM. Instantiating the BRAM is the most efficient way of using a RAM and is supported by all kinds of RAMs. The matrix obtained after performing the operation (matrix C), this 16-bit matrix is written to a simple dual-port BRAM that will store the matrix into memory. From the memory, matrix C will be written to a bin to BCD module that will convert the 16-bit binary number to four 4-bit BCD's.

Encoder:

As the matrix C is 16 bits wide, the terminal needs to receive an input that is ASCII coded. This would require breaking the 16 bits into 4 bcd values which can then be encoded into The inputs to the encoder are the four 4-bit BCD numbers and the main purpose of this module is encode the inputs into a 8-bit ascii number. To get the data sent in the correct order to the terminal, we have made an event for button click on each number. Without the button click, the data will be sent in a very random order to the terminal.

The output of the encoder will be then sent to the UART Transmitter module which is designed to send 1-bit of data to the terminal with each button click (btnU). This is sent 1-bit at a

time by utilizing a FIFO. Once the entire matrix is outputted on the terminal, the next button click will give out a special character meaning it is complete.

Instructions to Operate

1) First state is expecting a 0 or a 1 for either add or multiply, no space needed
2) The next set of numbers will be matrix A, starting with row1 and then row2, etc. Each number will have a space in between except for the last one. Once all numbers are inputted, the interpreter is expecting the "new line" key to go to the next state, which is ascii 10.
3) The next set of numbers will be matrix B, starting with row1 and then row2, etc. Each number will have a space in between except for the last one. Once all numbers are inputted, the interpreter is expecting the "new line" key to go to the next state, which is ascii 10.
4) Hitting btnU will display a special character "!" on the terminal the same amount of times as (num_row*num_col + 1). This is loading values of matrix C to memory in the correct order.
5) After that the next button clicks will start displaying 1-bit of values of matrix C from the encoder to terminal. When matrix C is fully shown on the terminal, the next button click will display the same special character "!".

VHDL vs. Software Matrix Operator

One of the main differences between the software and VHDL matrix operator is that the software version is done in one sequential state machine. The software matrix operator provided has a series of instructions that will tell the CPU what to do. The VHDL matrix operator we designed is a collection of digital circuits that are connected in parallel. The VHDL code is in many subsequent modules and are synthesized to the hardware (FPGA). We have chosen to use a state machine for a portion of our design, while the software version does not have that choice.