**A GENERAL PURPOSE REAL-TIME SoC BASED MATRIX MANIPULATOR**

**Project**

**ECE 5730 –Embedded System Design Using FPGAs**

**Submitted by**

**Cory Ness**

**Kokila Subramanian**

**Shivani Devatha**

**Submitted to**

**Professor Subramaniam Ganesan**



**Fall 2021**

**Table of Contents**

[Abstract 3](#_Toc89621939)

[Objective 4](#_Toc89621940)

[System-level Block Diagram 4](#_Toc89621941)

[Hardware Components 5](#_Toc89621942)

[**UART-TTL Converter:** 5](#_Toc89621943)

[**DE1-SoC FPGA Development Kit:** 6](#_Toc89621944)

[Matrix Manipulation Theory 6](#_Toc89621945)

[**Pseudo code for Matrix Addition:** 6](#_Toc89621946)

[**Pseudo code for Matrix Subtraction** 7](#_Toc89621947)

[**Pseudo code for Multiplication:** 7](#_Toc89621948)

[**Pseudo code for Transpose of Matrix:** 8](#_Toc89621949)

[**Pseudo code for Determinant of Matrix:** 8](#_Toc89621950)

[VHDL Code 9](#_Toc89621951)

[**VHDL code for UART Decoder:** 9](#_Toc89621952)

[**VHDL code for UART Transmitter:** 10](#_Toc89621953)

[**VHDL code for FIFO:** 11](#_Toc89621954)

[Code 13](#_Toc89621955)

[**VHDL code for UART Decoder** 13](#_Toc89621956)

[**VHDL code for UART Transmitter:** 14](#_Toc89621957)

[**VHDL code for FIFO:** 15](#_Toc89621958)

[References 18](#_Toc89621959)

Abstract

Matrix manipulation includes operations such as addition, subtraction, multiplication, inverse, and transpose etc., on the array of data stored as rows and columns. It is an essential process used in diverse fields of science and commerce including but not limited to Computer technology, Optics, Geology, Cryptography, Network Theory, Robotics and Animations, and Finance. Real-time matrix operations include a large number of computations, required for process control, and data & signal processing, which directly impacts the system performance. Real-time matrix calculation becomes a bottleneck for performance of fast system applications as it requires large computation power, memory and time.

In this project, we present a design which can perform matrix manipulations such as addition, subtraction, scalar and matrix array multiplication, determinant of matrix, co-factor and other vital matrix operations required for real-time signal processing. The matrix manipulator is developed using Intel DE1-SoC development board with Cyclone V processor is interfaced with FT232R FTDI UART to establish serial communication between the FPGA board and PC. The matrix calculator is implemented in the NIOS II soft-core processor present on DE1-SoC board, the inputs are fed from a host computer onto FPGA board via UART serial port, a USB to TTL convertor module will be used to establish serial communication. The final computational results are fetched using the serial communication onto the remote PC.

Objective

Real-time signal processing system involves large computations and complex process control operations on massive array of data which is highly time-consuming. Currently, most of matrix operations are performed using software. The expansion of matrix dimension and reduction in processing speed has become bottleneck for the most of the existing real-time fast system applications. The objective of this project is to develop a Real-time Matrix-Manipulator using UART to TTL convertor for serial communication, Intel DE1-SoC FPGA Development Kit, and TRDB- TLM Touch screen LCD Display. The input matrix values are served into the FPGA board using Serial communication from a remote system. The matrix operations are executed in NIOS-II processor available in DE1-SoC board. The manipulated resultant matrix is displayed in the LCD display. The NIOS –II processor is used to control system operation and perform basic matrix operations such as addition, subtraction, Multiplication, finding determinant, co-factor, and other essential matrix operations required for real-time signal processing.

System-level Block Diagram

The system-level Block diagram for the Real-time Matrix Manipulator is shown in figure 1. The major functional blocks include the Remote PC, UART TTL convertor, the DE1-SoC board and the TRDB-TLM LCD display.



SDRAM

UART RX

NIOS II

Matrix Operation Algorithm

SSRAM

UART TX

UART

PIO

**USB to UART**

Serial communication

**DE1-SoC**

**PC**

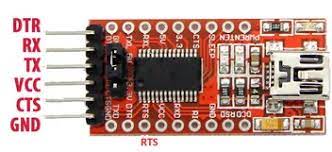
**Figure 1: System-level Block Diagram**

Hardware Components

The developed Real-Time Matrix Manipulator consists of the following hardware components:

1. UART-TTL Convertor – FT232R FTDI
2. DE1-SoC Development Kit

**UART-TTL Converter:**

******

**Figure 2. FT232R FTDI USB UART**

The USB UART-TTL converter used is an FT232R FTDI (Future Technology Device International Ltd.) chip on a breakout board that simplifies serial communication by reducing the external components. Figure 2 shows the package of the USB UART. It is a fully integrated chip with 1024 bit EEPROM, clock generation, and USB termination resistors. The FT232R serial UART can support data rate transfer rates from 300 baud to 3 Mbaud at the TTL levels (FT232R, n.d.). High data throughput is achieved through the 128 byte receive buffer and the 256 byte transmit buffer that utilizes smooth buffer technology. The UART interface will support 7 or 8 data bits with 1 or 2 bits for representing odd, even, mark, space and non-parity bits (FT232R, n.d.). The configurable I/O pins, integrated +3.3V level convertor provides output drive strength for various applications. It can be used as interface with MCU/ PLD/FPGA based designs, PDA to USB data transfer, smart card readers, Industrial control, Hardware and Wireless modems, Bar Code Readers, and Cellular Phone USB data transfer cables and interfaces ((FT232R, n.d.).

The communication is established by making use of the UART Rx and Tx lines going to pins 1 and 3 on the 20 pin expansion header of DE1-SoC board. It allows any computer to open a serial port and communicate to the DE1-SoC over a COM port, inputting the equations needed for the Matrix equations. In addition, the DE1-SoC can transmit data back to the computer by sending UART data to the FTDI board, which will display on the serial port screen for the user.

**DE1-SoC FPGA Development Kit:**

The DE1-Soc Development kit provides a hardware platform that combines the System-on Chip (SoC) FPGA and dual-core Cyclone- A9 embedded core which can be used for wide industrial application improvement. It provides users the power of flexibility in design, paired with high-processor, low-power processing system. It provides seamless integration of both SoC and ARM based hard processor system (HPS), including processor, memory and peripheral interfaces over a high-bandwidth interconnect backbone. The DE1SoC board includes hardware such as high-speed DDR3 memory, video and audio capability, Ethernet, networking and much more.

Matrix Manipulation Theory

Matrices are representation of array of data commonly used in extensive real-world data processing, research and scientific studies. Matrix Multiplication provides good approximation for complicated calculation of real-time sensitive engineering applications signal processing, image processing, network theory and many more. Matrix manipulation is engaged with every part of day-to-day life including data base management, encryption that helps security, 3D gaming, robotics and animation, create models for economic and business, construction, physics and geology. The common matric calculations involved are addition, subtraction, multiplication, transpose, finding the determinant value and inverse of the matrix.

**Pseudo code for Matrix Addition:**

Matrix addition can be performed only when both the matrices under operation are of same size. It is commonly employed in data, image and signal processing. The void matrix\_addition() performs the following activity:

for (i=0;i<r1;i++)

{

for (j=0;j<c1;j++)

{

res[i][j]= a[i][j]+b[i][j];

}

}

**Pseudo code for Matrix Subtraction**

Matrix subtraction is a linear algebraic operation which can be performed only when both the matrices under operation are of same size. The void matrix\_subraction() performs the following operation:

for (i=0;i<r1;i++)

{

for (j=0;j<c1;j++)

{

res[i][j]= a[i][j]-b[i][j];

}

}

**Pseudo code for Multiplication:**

Matrix Multiplication can be either scalar or matrix multiplication, and is the most challenging algebraic operation. Matrix multiplication can be either scalar, vector or matrix multiplication. Scalar multiplication is a element-wise multiplication of matrix elements using a real or complex value. In the below defined code, *void scalar\_matrix\_multiplication()* executes matrix multiplication using constant value k.

for (i=0;i<r1;i++)

{

for (j=0;j<c1;j++)

{

res[i][j] = k\*a[i][j];

}

}

Multiplication of matrices can be executed under the condition that the number of rows of first matrix should be equal to number of columns in the second matrix (i.e. [A]i x j\* [B]m x n = [C]i x n.

if (r1 != c2)

{

printf("Matrix multiplication cannot be performed\n");

}

else

for (i=0;i<r1;i++)

{

for (j=0;j<c2;j++)

{

res[i][j]=0;

for (k=0; k<c1;k++)

res[i][j] += a[i][k]\*b[k][j];

}

}

**Pseudo code for Transpose of Matrix:**

Transpose is simply interchanging the rows and columns. It is generally employed when multiple matrix exists and the dimensions without transposing is not amendable for matrix operations. In real-time one usage of transpose is data manipulation, where the application requires the use of orthonormal matrix. In image processing switching rows and columns of pixels can result in exchanging the domain and space of the image transformation. In neural networks, frequent process weight and inputs of different sizes where the dimensions do not meet the requirement of algorithm can be accomplished using simple transpose of matrix.

for (i=0;i<c1;i++)

{

for (j=0;j<r1;j++)

{

res[i][j] = a[j][i];

}

}

**Pseudo code for Determinant of Matrix:**

Determinant is the number associated with square matrix that encodes information about the matrix. Determinant of matrix is used to check the consistency of any system and to solve linear equations.

for (int f = 0; f < n; f++)

{

// Getting Cofactor of a[0][f]

temp= malloc(2\*sizeof\*temp);

for (i=0; i< 2; i++)

{

temp[i]= malloc(2\*sizeof\*temp[i]);

}

i=0;

// Looping for each element of the matrix

for (int row = 0; row < n; row++)

{

for (int col = 0; col < n; col++)

{

// Copying into temporary matrix only those

// element which are not in given row and

// column

if (row != 0 && col != f)

{

temp[i][j++] = a[row][col];

// Row is filled, so increase row index and

// reset col index

if (j == n - 1)

{

j = 0;

i++;

}

}

}

}

if(f%2==0)

{

(D += a[0][f]\*(matrix\_det(temp, n - 1,n - 1)));

}

else

{

D -= a[0][f]\*(matrix\_det(temp, n - 1,n - 1));

}

// terms are to be added with alternate sign

D1=D;

VHDL Code

**VHDL code for UART Decoder:**

The UART Decoder accepts 2 inputs and has 2 outputs. The first input is Clk, with a frequency set by the generic variable ClockRate. The second input is Rx, which is the pin state of the UART Rx pin. The first output is Complete, a single bit that is low while receiving Uart data and high once the Uart byte is fully decoded. The second output is Data, an 8-bit vector that is decoded value from the Rx pin. The VHDL code for the UART decoder works as follows:

1. Input the clock rate and baud rate using the generic map, as this will not change throughout operation.
2. Wait for the start-bit from UART Rx (High->Low Transition).
3. Begin counting the clock cycles until we reach the necessary baud rate.
4. Halfway-through the count, sample the Rx line and save it in data\_register, this ensures we sample when the data line is stable.
5. Once the count reaches the baud value, reset the count
6. Repeat until all 10 bits have been sampled (start bit, data bits, stop bit).
7. Once all bits have been sampled, set the Complete signal to indicate Data is stable
8. Wait for next start-bit.

This isn’t sufficient, however, for the NIOS processor to use, since it will stream all the data in immediately. We can make use of interrupts inside the NIOS processor to react whenever the Complete signal rises, but it will take time away from performing calculations. We decided to make use of the decoder by also implementing a FIFO (First In First Out) buffer, which will buffer in up to 128 bytes of UART data to be read out by the NIOS processor when it’s ready.

**VHDL code for UART Transmitter:**

The VHDL code for the UART transmitter is very similar to the Decoder, but in reverse. It has 3 inputs and a single output. The first input is Clk, the system clock with frequency specified in the generic ClockRate. The second input is Send, a single bit that triggers the send sequence on a low to high transition. The third input is Data, an 8-bit vector that is the value to send over UART. It also has a single output, Tx, which should be tied to the Tx pin on the DE1 board. The transmitter works as follows:

1. Input clock rate and baud rate using generic map, as this will not change throughout operation
2. Wait for low->high transition on Send input
3. Bring Tx low, and begin counting clock cycles until we reach necessary baud rate
4. Once the count has rolled over, change the Tx state based on the bit of Data
5. Repeat until all 8 data bits have been sent
6. Bring Tx high to send the stop bit.
7. Reset all variables back to original states and wait for next low->high transition.

Following this we can control the transmitter tying a parallel IO in the NIOS processor to send the start signal, and the data can be tied to another parallel IO. The Clk and Tx bits are tied to the 50 Mhz clock and UART Tx pin respectively, and this is everything necessary to integrate a UART transmitter into our NIOS design.

**VHDL code for FIFO:**

The VHDL code for the FIFO is very simple and elegant. It uses 4 inputs and 1 output. The first input is complete, which indicates that data is complete and ready to be pushed into the FIFO. The next is data\_in, an 8 bit vector that is the data to be pushed in. The third is pop, which will pop out the data and point data\_out to the next data to be sent out. The final input is clear, which clears and resets the FIFO so that it’s as if it had just started. The only output of the FIFO is data\_out, an 8-bit vector that is the data to be sent out. The operation for the FIFO is very simple, first configure the generic num to the amount of data you want to buffer. Then, tie the data\_in signal to some data you want buffered, and whenever you press complete, it will be buffered into the FIFO. Meanwhile, data\_out is always the oldest data in the buffer, and you can point data\_out to the next oldest data by pressing pop. Clear can be tied to a debug pin so that you can manually clear the FIFO during development. The process for the FIFO is as follows:

1. Configure the depth of the FIFO with the num generic value
2. Wait for low->high transition on the complete input or pop input or clear input
3. If low->high on complete…
   1. Copy data\_in into the mem (an array of size num of 8-bit vectors) signal at location start
   2. Increment start by one
4. If low->high on pop…
   1. Increment stop by one
5. If low->high on clear…
   1. Set start and stop both to 0
6. Check if start is equal to stop
7. If yes…
   1. Output “00000000”
8. Otherwise…
   1. Output the value in mem at location stop

Code

**VHDL code for UART Decoder**

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

USE ieee.numeric\_std.all;

ENTITY Uart\_Decoder IS

GENERIC(

BaudRate : IN INTEGER := 9600;

ClockRate : IN INTEGER := 50\_000\_000

);

PORT(

Clk, Rx : IN STD\_LOGIC;

Complete : OUT STD\_LOGIC;

Data : OUT STD\_LOGIC\_VECTOR(7 DOWNTO 0)

);

END Uart\_Decoder;

ARCHITECTURE Behavior OF Uart\_Decoder IS

SIGNAL started : STD\_LOGIC; -- Whether we've started or not

SIGNAL bit\_location : Integer RANGE 0 TO 9 := 0; -- Where to save the bit

SIGNAL data\_reg : STD\_LOGIC\_VECTOR(9 DOWNTO 0); -- Local copy of the data, including start and stop bit

BEGIN

PROCESS (Rx, bit\_location)

BEGIN

IF bit\_location = 9 THEN

started <= '0';

ELSIF Rx = '0' AND Rx'event THEN

started <= '1';

END IF;

END PROCESS;

PROCESS (Clk, Rx, started)

VARIABLE counter : Integer RANGE 0 TO ((ClockRate / BaudRate)+1) := 0;

BEGIN

IF Clk = '1' AND Clk'event THEN

IF started = '1' THEN

counter := counter + 1;

IF counter = ((ClockRate) / BaudRate) THEN

counter := 0;

bit\_location <= bit\_location + 1;

ELSIF counter = (ClockRate / (BaudRate \* 2)) THEN

data\_reg(bit\_location) <= Rx;

END IF;

ELSE

bit\_location <= 0;

counter := 0;

END IF;

END IF;

END PROCESS;

Complete <= NOT started;

Data <= data\_reg(8 DOWNTO 1);

END Behavior;

**VHDL code for UART Transmitter:**

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

USE ieee.numeric\_std.all;

ENTITY Uart\_Transmitter IS

GENERIC(

BaudRate : IN INTEGER := 9600;

ClockRate : IN INTEGER := 50\_000\_000

);

PORT(

Clk, Send : IN STD\_LOGIC;

Data : IN STD\_LOGIC\_VECTOR(7 DOWNTO 0);

Tx : OUT STD\_LOGIC

);

END Uart\_Transmitter;

ARCHITECTURE Behavior OF Uart\_Transmitter IS

SIGNAL started : STD\_LOGIC; -- Whether we've started or not

SIGNAL bit\_location : Integer RANGE 0 TO 9 := 0; -- Where to save the bit

BEGIN

PROCESS (Send, bit\_location)

BEGIN

IF bit\_location = 9 THEN

started <= '0';

ELSIF Send = '1' AND Send'event THEN

started <= '1';

END IF;

END PROCESS;

PROCESS (Clk, started, Data)

VARIABLE counter : Integer RANGE 0 TO ((ClockRate\*2) / BaudRate) := 0;

BEGIN

IF Clk = '1' AND Clk'event THEN

IF started = '1' THEN

counter := counter + 1;

IF counter = ((ClockRate) / BaudRate) THEN

counter := 0;

bit\_location <= bit\_location + 1;

ELSIF counter = (ClockRate / (BaudRate\*2)) THEN

-- Send the start bit

IF(bit\_location = 0) THEN

Tx <= '0';

-- Send data bits

ELSIF bit\_location > 0 AND bit\_location < 9 THEN

Tx <= Data(bit\_location - 1);

-- Send the stop bit

ELSE

Tx <= '1';

END IF;

END IF;

ELSE

bit\_location <= 0;

counter := 0;

Tx <= '1';

END IF;

END IF;

END PROCESS;

END Behavior;

**VHDL code for FIFO:**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity FIFO is

generic (num : in integer := 128);

port ( complete : in std\_logic;

data\_in : in std\_logic\_vector(7 downto 0);

pop, clear : in std\_logic;

data\_out : out std\_logic\_vector (7 downto 0));

end FIFO;

architecture behavior of FIFO is

type fifo\_size is array (0 to num-1) of std\_logic\_vector (7 downto 0);

signal mem : fifo\_size; --declaring 32 buffers with 8 bit wide

signal stop : integer range 0 to num-1 := 0;

signal start : integer range 0 to num-1 := 0;

begin

--at the rising edge of complete

--data is received into buffer

--if index of mem reaches num (32) the index is changed ot 0

process(complete, pop, data\_in, clear)

variable newStart : integer range 0 to num-1;

variable newEnd : integer range 0 to num-1;

variable toSend : std\_logic\_vector(7 downto 0);

begin

if clear = '1' then

start <= 0;

stop <= 0;

else

if complete'event and complete = '1' then

newEnd := stop - 2;

if not(start = newEnd) then

start <= start + 1;

if(start = num-1) then

mem(0) <= data\_in;

else

mem(start + 1) <= data\_in;

end if;

end if;

end if;

--assuming pop as a 1 bit input

--first value from the queue is stored into data\_out object

--queue is updated

newStart := start;

if pop'event and pop = '1' then

if not (newStart = stop) then

stop <= stop + 1;

end if;

end if;

end if;

-- Get the current head of the ring buffer if we're not empty

if not (start = stop) then

if(stop = num - 1) then

toSend := mem(0);

else

toSend := mem(stop + 1);

end if;

else

toSend := "00000000";

end if;

data\_out <= toSend;

end process;

end;

References

*Future Technology Devices International LTD FT232R USB ...* (n.d.). https://ftdichip.com/wp-content/uploads/2020/08/DS\_FT232R.pdf.