CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Testing infrastructure for experimental chips

Master's thesis in Embedded Electronic System Design

SINAN DING

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Testing infrastructure for experimental chips

SINAN DING

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Testing infrastructure for experimental chips
SINAN DING

Supervisor: Lars Svensson, Department of Computer Science and Engineering
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2021

Testing infrastructure for experimental chips
SINAN DING
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As transistor sizes shrink and performance requirements of experi-mental chips increase, verifying and testing the chips becomes more and more complex. One traditional way of testing is generating external input signals to test the chip and evaluating the testing results outside the chip but this is a costly approach. A on-chip testing approach, in which the generation of input signals and evaluation of output signals are integrated with the chip, would be an interesting alternative. This low-cost approach has been used in testing a recent Forward-Error-Correction chip (FEC-chip). The aim of this thesis project is to generalize the testing evaluation setup used in testing experimental chips to make it more effective and efficient. In this project, a testing system, in which an MCU can read/write data to/from experimental chips via an SPI interface and a CRC error detection scheme was used to improve the data transaction reliability of the testing system, was designed and implemented. A PCB supporting the new testing system was designed. The results of the thesis project can be used to testing other designed chips with its low-cost and fast data transmission speed features.

Keywords: Experimental chips, testing, MCU, SPI, CRC check, PCB

Acknowledgements

First of all, I would like to thank my supervisor Prof. Lars Svensson, my examiner Prof. Per Larsson-Edefors, my teacher Lena Peterson, and Phd students Victor Åberg and Erik Börjesson for suggesting the thesis topic and helping me define the thesis's scope and prioritize the objectives. Secondly, I would like to thank my supervisor Lars Svensson for invaluable feedback and discussions; Victor Åberg and Erik Börjesson, for appreciated discussion throughout the project; Per Larsson-Edefors, for helpful feedback and comments of this thesis. Finally, I would like to thank my grandparents, my parents, and my friends for encouraging me to finish this project.

Sinan Ding, Gothenburg, January 2021

Contents

List of Figures

List of Tables

1
Introduction

As a result of technology scaling, component densities of Very Large Scale Integrated (VLSI) circuits and printed circuit board (PCB) have increased significantly [1], and the operating frequencies of the chip can now reach several GHz [2]. As a consequence of that, verifying and testing experimental chips becomes as complicated as developing them. The huge amount of test data should be transferred reliably with high frequencies to the chip. Usually, the testing equipment that can reliably transfer test data in high-speed is extremely expensive. Therefore, the testing cost becomes a significant part of the total project cost.

In a testing system, the input signals must be supplied to the experimental chip, and the output signals of the chip must be evaluated. Figure 1.1 shows the traditional outside-chip approach: the input signals generation part and output signals evaluation part is outside the chip. The generated input signals will be transferred across the chip edge into the chip. After the input signals get processed inside the chip the corresponding output signal will be transported across the chip edge to the outside evaluation part for further evaluation [3]. However, this approach is not very cost effective. Dependably carrying the high-bandwidth signals across the chip edges in high-performance design usually needs expensive external test equipment. Besides, there is also a need for chip pins which is a precious resource.

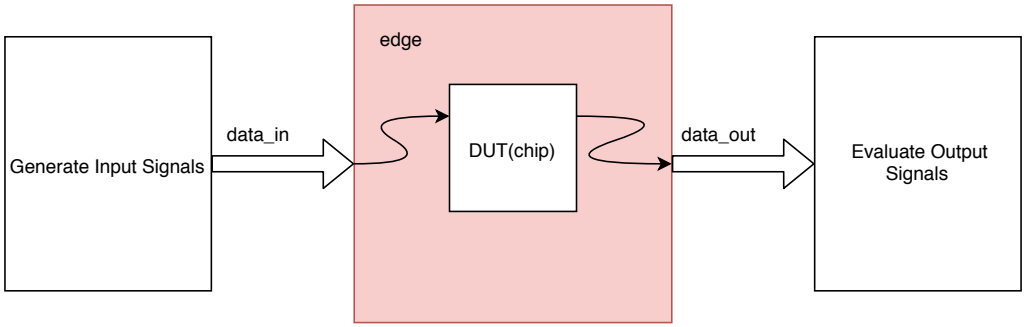


Figure 1.1: Outside chip approach

In contrast with the outside-chip approach, an on-chip approach is using extra hardware to generate input signals to the design-under-test (DUT) unit and evaluate output signals; hardware which is integrated inside the experimental chip [3]. The on-chip approach relaxes the requirement on expensive external test equipment and is more economical since only power and clock signals need to be applied to PCBs. This approach can be used in testing application specific integrated circuits (ASICs),

for example, in two recent forward-error-correction chips (FECs) [4] [5]. Here, the input pseudo-random test data was generated with a linear feedback shift register(LFSR). Those input data were fed into a simulated communication channel where some bits were flipped by LFSRs generated noise. The chip performance in [4] [5] can be evaluated by comparing the data fed into the simulated channel with data after the FEC decoder.

1.1 Aim

The existing testing approach used in the present FEC-chips can be generalized and then be used for other test designs [3] [6]. The thesis aims to generalize the existing testing approach and make the experimental chip evaluation setup more effective and efficient. The results will be used to test other experimental chips and reduce the time and expense of the chip design.

The thesis objectives are:

1. to define and document the interfaces to an Arduino board;
2. to design a new PCB shield that can be used for testing other experimental chip;
3. to develop the Arduino software that can handle other shields designed to the specifications;
4. to develop some generic IP blocks to support interfaces that are used for Arduino to set parameters and read results on the experimental chip;
5. to investigate some features and alternative solutions:
    - The original testing setup can only run one test site at a time, it would be interesting to find a solution to run several test sites in parallel or concurrently to speed up the testing procedure.
    - With more complex chip design, there might be more external test equipment need to connect to the pins of the chip to conduct the experiments so the PCB shield and the Arduino software should be upgraded to control those test equipment.
    - There are more powerful computer cards than Arduino that can communicate to PC and the PCB, such as Raspberry Pi. It is also attractive to investigate those powerful computer cards that can be used in the testing infrastructure.

1.2 Thesis Outline

The thesis outline is as following: the Chapter 2 reviews the basic relevant technical background in the thesis project. Chapter 3 explains the system architecture and design decisions. Chapter 4, 5, and 6 presents the design implementation on Arduino and VHDL, and PCB design respectively. The testing and evaluation setup, and the experiment results are presented in Chapter 7. Chapter 8 and 9 discuss the experiment results get in Chapter 7, and draws conclusions.

2
Technical background

This chapter explains the theory and techniques used in this project. Starting with the introduction of the FEC-chip evaluation setup mentioned in Chapter 1, brief introductions of serial communication interface and error-detecting scheme follow. Next, the FIFO buffer is introduced. Finally, the meatastability phenomena and synchronizers is described.

2.1 Present FEC-chip evaluation setup

As mentioned in the Chapter 1, the on-chip approach has been used in testing two recent Forward-Error-Correction chips (FEC-chip). Figure 2.1 shows the FEC-chip evaluation setup based on the on-chip approach [6]. The on-chip data-generation and -evaluation hardware are integrated with the device under test (DUT). A PCB with sockets accepts the experimental chip. The external lab clock generators and power supplies connect to some pins of the socket to provide the clock signals and power to the chip. The other pins of the socket can be used to control the chip, send data to the chip, and retrieve data from the chip under the microcontroller's control. An Arduino, a microcontroller, connects to PC via a USB connection and can read and control the I/O pins of the socket via the Arduino software. The software for the Arduino and also MATLAB routines may be run on any PC to control and monitor the Arduino and the lab supplies and clock generator via the Internet.
The most obvious advantage of this on-chip test setup is the low cost. Besides, since the microcontroller plays a role as a gateway, when the one side of the gateway is conceived, the other side's development becomes easier.

To make the current setup become more efficient and effective, there are some areas that can be improved:

- The Arduino communicates with the PCB through General-purpose input/output (GPIO) pins which is hardware resource inefficient since most of the GPIO pins of the Arduino are occupied. The communication using GPIO pins cannot support very high speed data transmission thus it would be a bottle-neck for more advanced design.
- The parameters sent to DUT may be flipped during transmission. There is no error detection process in the transmission so the wrong parameters may be set and the test result might make no sense. Due to the complexity of the DUT, each experiment of testing the DUT takes long long time. It is too late to modify the experiment parameters after getting the test results triggered

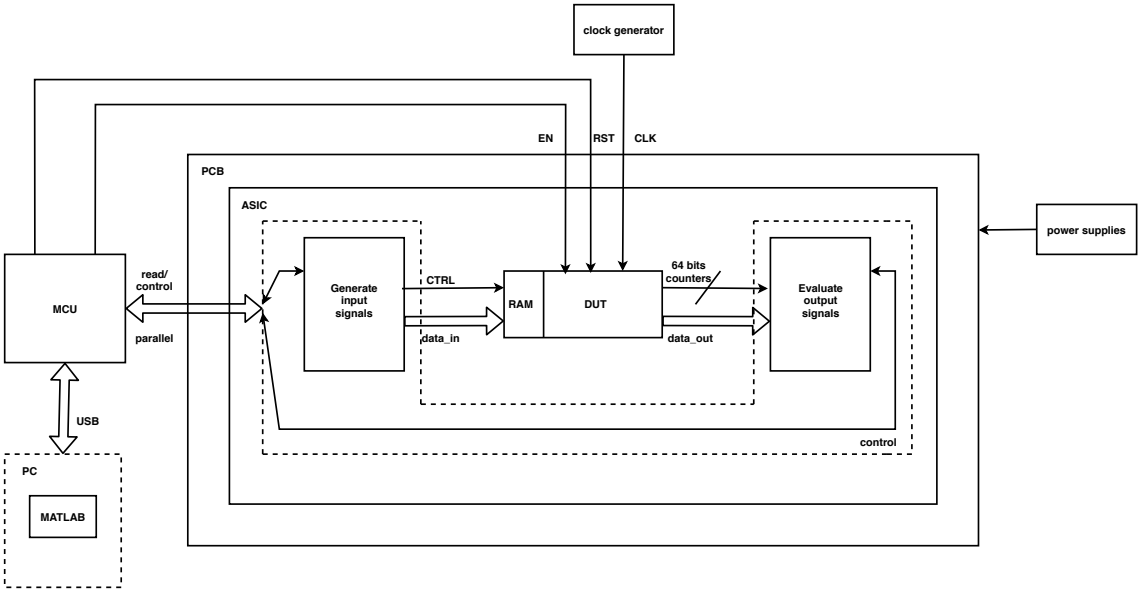


Figure 2.1: FEC-chip evaluation setup based on on-chip approach.

by wrong testing parameters. As a result of this, the testing-time cost will increase significantly by any unreliable transmission.

- There might be more than one experiments to be done. Therefore, running test sites in parallel or concurrently becomes interested as it can get several experiments results at the same time.

The challenge to achieve these improvements is realizing the improvements while keeping the low cost benefit of the evaluation setup.

The communication interface between the Arduino and the PCB can be upgraded to some serial interfaces with higher hardware efficiency and higher speed. The multiple test sites running rely on a certain communication protocol/interface. Section 2.2 gives an brief introduction to some serial interfaces. Some error-detecting schemes can be used to improve the transmission reliability. Two commonly used error-detecting techniques are presented in Section 2.3. The Arduino and the experimental chip are two asynchronous subsystems of the whole evaluation setup. Data can be passed between two asynchronous systems through a first-in-first-out (FIFO) buffer. Basic knowledge of FIFO buffer is described in Section 2.4. Data travel from the Arduino to the experimental chip need to be synchronized first then they can be used for further processing; otherwise, the asynchronous data will cause the flip-flops of the experimental chip enter metastable state and the in-between output value of the flip-flops will propagate through the entire experimental chip. The metastability phenomenon and how to prevent it is explained in Section 2.5.

2.2 Serial communication interface

Serial communication sequentially transmits data bit by bit in communication channels or computer bus. There are two types of serial communication interface, one is

synchronous serial communication interface, such as Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI), and JTAG; and another is asynchronous serial communication interface, such as Universal asynchronous receiver-transmitter (UART).

2.2.1 Universal asynchronous receiver-transmitter (UART)

Universal asynchronous receiver-transmitter (UART) is a hardware device for transmitting/receiving data asynchronously and serially [8]. UARTs can be used to transmit data between a computer and peripheral device serial port. In order to address the asynchronous problem, UARTs need extra start and stop bits concatenated with every bytes. Since there is no clock information transferred on the serial line, the UARTs uses oversampling scheme to measure the middle point of each bit where the bit is valid. That is, UARTs samples the same bit several times. As a consequence of this, the UART cannot reach high data rate. UARTs can support speed from 300 baud to 115 200 baud which might be inadequate for future's advanced chip design so it won't be explained in detail in this section.

2.2.2 Inter-Integrated Circuit (I2C)

Inter-Integrated Circuit (I2C) is a hardware protocol that allows multiple slave devices to communicate with one master or several master devices in a synchronous and serial way [9]. It suits for low-speed communication between on-board peripherals. I2C consists of two wires, called Serial Data Line (SDA) and Serial Clock Line (SCL). Serial Clock Line (SCL) is a unidirectional line and usually generated by the master device. Serial Data Line (SDA) is bidirectional. Each slave on the I2C bus is distinguished by its unique 7-bit address. There are two transmission modes of the I2C protocol: fast mode and high-speed mode. In the fast mode, the maximum of SCL rate is 400 kHz; and in the high speed mode, the maximum of SCL rate is 3.4 MHz. Since the I2C protocol cannot provide high speed transfer rate (the upper limit is 3.4 MHz in high-speed mode), other details of the I2C won't be expanded in this section.

2.2.3 Serial Peripheral Interface (SPI)

Unlike the I2C protocol which is half-duplex and with low transmission speed, serial Peripheral Interface (SPI) is a full-duplex, synchronous serial data bus that allows transmitting data from a master device to slave device(s) [8]. The SPI doesn't define a limit speed, it depends on the device on the SPI bus. In general the SPI speed can provide higher transmission speed than the I2C protocol. The SPI protocol has four signal wires: MISO, Master In Slave Out; MOSI, Master Out Slave In; SCLK, Serial Clock; and SS, Slave Select which is active low, when SS is low, the slave is selected. The SCLK and SS are generated by the master device.

The SPI master needs to configure the clock phase (CPHA) and clock polarity (CPOL) when it transmits data. The clock phase determines when would the data are changed or sampled at the rising or trailing edge of the clock [10]. The clock

phase 0 changes data at the trailing edge of the clock, and sampling the data at the leading edge of the clock. The clock phase 1 is vice versa. The clock polarity determines the polarity of the clock, the clock is idle at high or low, the leading edge and the trailing edge are rising edge or falling edge. The CPOL 0 is a clock idles at low and a clock cycle contains a pulse of 1. The leading edge of the CPOL 0 clock is a rising edge, and the trailing edge of the CPOL 0 clock is a falling edge. The CPOL 1 is the opposite of the CPOL 0. The combination of clock polarity and phase is called mode. Since the there are two types of clock polarity and phase respectively, there are four types of mode in total. Mode "00" and Mode "11" are the mostly used modes in SPI communication.

The conceptual diagram of SPI is depicted in Figure 2.2; there are 8-bit shift registers at both master and slave side [11, p.326]. These two shift registers are connected in a ring via MISO and MOSI line. The master device generates SPI clock (SCK). At the beginning of data transaction between the master and the slave, data are loaded into the shift registers. Then the data stored in both shift registers will right shift by one bit on an edge of the SCK, and these one bit data will be sampled at a opposite edge of the SCK when they are stable.

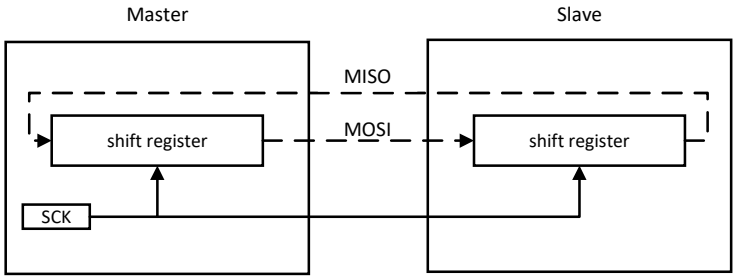


Figure 2.2: conceptual diagram of SPI

Consider SPI mode 3 (mode "11") as an example to illustrate SPI data transaction, see Figure 2.3. The SPI master initiates the communication by asserting the SS to low; then SPI clock is generated with initially high. On the first falling edge of the SPI clock (the red dash line in the figure), data stored in the shift register will shift out one bit on the MISO line, and the data on the MOSI line will shift in one bit at the same time. Then these one bit data will be sampled at the next transition edge (rising edge, the blue dash line in the figure). In the mode 3, changing data of the data line always happens on the falling edge of SPI clock and sampling data of the data line always happens on the rising edge of SPI clock until all 8-bits data get exchanged.

SPI master device can communicate with multiple slave devices. There are two configurations between the SPI master and SPI slaves: one is star-topology configuration, another is daisy-chain configuration [11, p.326]. Assume there are two slave devices connected to the master device. Figure 2.4a depicts the star-topology con-

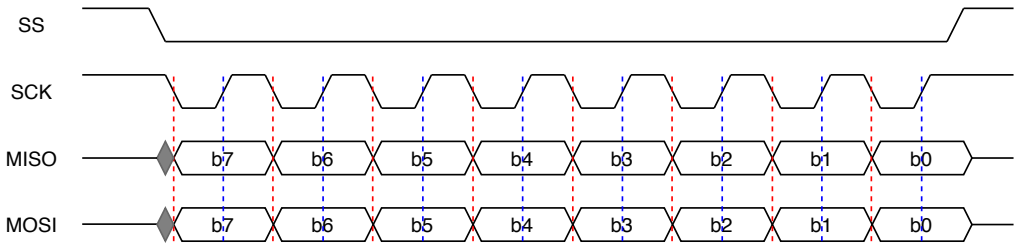
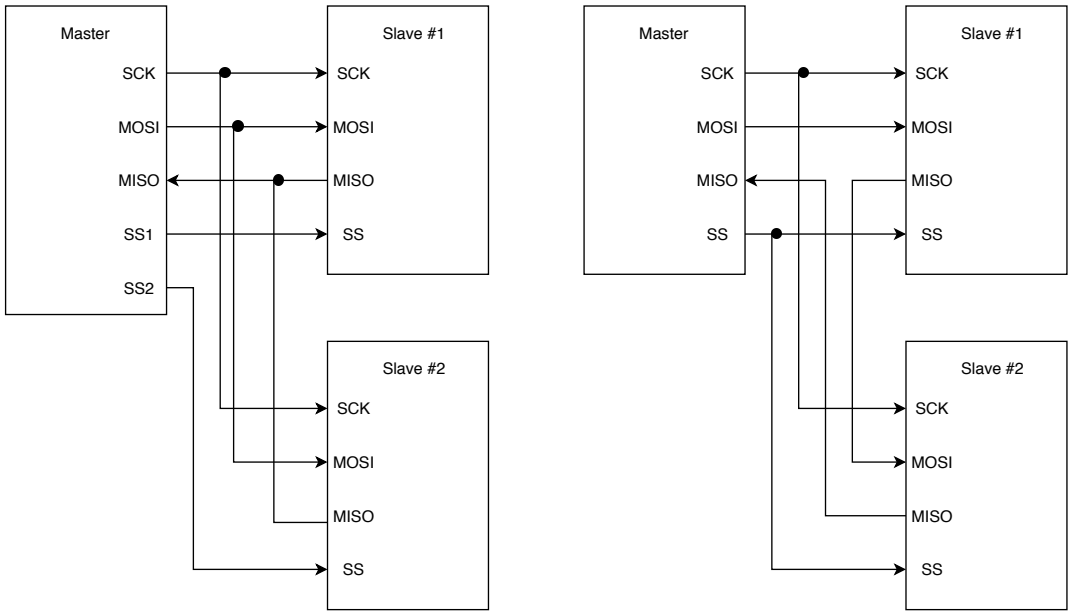


Figure 2.3: Timing diagram of SPI mode 3. The red dash line indicates the falling edge of the SCK, and the blue dash line indicates the rising edge of the SCK.

figuration. In the star-topology configuration, the SPI master provides independent chip select pins to each slave device but those slave devices use common data lines and clock lines. When the chip select pin of the slave is inactive, the slave's MISO line is disabled and put it into high-impedance state. The daisy-chain configuration is shown in Figure 2.4b, the SPI slaves share chip select pins and SPI clock pins, the SPI master device provides the MOSI line to the first slave device, and the output MISO of the first slave device is the input of the second slave device, and so on, the final slave device's output goes back to the master device via MISO line of it.



(a) star-topology configuration (b) daisy-chain configuration

Figure 2.4: The diagram of star-topology and daisy-chain configuration with two slave devices

The star-topology configuration has advantages such as easily connect new slaves to the master, the signals don't need to go through all slave nodes, and the likelihood of data collision get eliminated. However, since the number of chip select pins is equal to the number of slaves, the amount of slave numbers will get limited by the pin num-

bers of the master device. Also, when considering the PCB design of star topology configuration, the clock skew and signal reflection may wreak the signal integrity. Thus, it's better to use daisy-chain configuration when there are many slave devices.

The SPI block of Arduino series contains SPI control logic, clock generating logic, and Pin control logic [12] [13]. There are three important registers inside the SPI block of Arduino, SPI status register, SPI control register, and SPI data register. The SPI control register stores the SPI interrupt enable, SPI enable, data order, master/slave select, clock polarity, clock phase, and SPI clock rate information. The SPI status register stores the SPI interrupt flag, write collision flag, and double SPI speed bit. The SPI data register stores data being read or write via SPI bus. When putting data into the SPI data register, the SPI clock starts to generate.

Arduino provides an SPI library so that the user doesn't need to manipulate on the SPI registers mentioned above. Table 2.1 summarizes the main functions of the SPI library [14]. It is worth mentioning that what the SPI.transfer function does is to put the data onto the SPI data register to initiate the data transmission [12] [13]. The return value of this function is the received data. So, when we only send data via SPI bus, the return value can be ignored. The last coming byte on the SPI bus will be stored in the Buffer register for further use. Therefore, the first returned byte on the SPI bus is an old byte brought by the last SPI.transfer() call. So, the first returning byte can be regarded as a junk data (has nothing with this time's SPI.transfer() call).

Table 2.1: Table of SPI library function.

function	description
SPI.begin()	initialize the SPI bus
SPI.beginTransaction()	initialize the SPI bus with settings
SPI.endTransaction()	stop the SPI bus
SPISettings()	configure the maximum SPI speed, data order (MSB first or LSB first), and data mode
SPI.transfer()	read/write data via SPI bus

2.3 Error-detecting scheme

When transmitting digital data over communication channels, noise or interference of the channel would introduce errors to the transmitted data. Error-detecting schemes detect the transmission errors and improve the reliability of the data delivery. Two common error-detecting schemes are Parity check and Cyclic redundancy check (CRC).

2.3.1 Parity check

Parity is the simplest form of error detection code. A parity bit is a bit that appended to a string of binary code. The total number of 1's of the string appended

with a parity bit can be even or odd. There are two kinds of parity bits, even parity bit and odd parity bit. The original total number of 1's of the string is an odd number, after the even parity bit added to the end of the string, the parity of the whole set(include the even parity bit) becomes to an even number, and vice versa. The even parity is 1-bit cyclic redundancy check (CRC).

The parity of the string can be calculated by XOR sum of all the bits, the XOR sum is 0 indicates the parity of the string is even, and vice versa. The parity check can be easily implemented but it can only detect odd number of bit errors since even number of bits flip won't change the parity of the string.

2.3.2 Cyclic redundancy check (CRC)

Cyclic redundancy check (CRC) is widely used in detecting burst errors in blocks of transmitted/received data in serial transmission systems [15]. The CRC code can be calculated by a polynomial long division. The polynomial presentation of the n-bit CRC is n+1 bits which is the divisor of the polynomial long division. The input data is the dividend of the division, and the n-bit reminder is the CRC checksum.

At a transmitter' side, first left-shift the input data with n-bits then the shifted data is divided by the n+1 bits (polynomial) divisor, and the result of the division, a n-bit remainder, is the CRC checksum. The original input data padded with this n-bit remainder will be sent from transmission end. At a receiving end, the received data is divided by the same divisor (polynomial). If the remainder is zero, it is determined that there is no detectable errors during transmission, and vise versa.

Figure 2.5 shows an example of the CRC calculation process. Suppose the input data is "11001010", 3-bit binary CRC needs to encode the input data, and the $(3+1)$-bit polynomial (divisor) is "1011". The first step is to left shift the input data with 3 bit, i.e. right pad 3 zeros after the input data. The padded input data is the dividend of the polynomial long division. Then left align the divisor with the dividend. The second step is to XOR the dividend and the divisor if the most significant bit of the dividend is not zero. The result of the XOR calculation becomes the new dividend. If the most significant bit of the dividend is zero, right shift the divisor until it meet a non-zero most significant bit. Repeating the second step for several times until the polynomial of the XOR result is not larger than the polynomial of the divisor. The last 3-bit of the XOR result is the remainder of the polynomial long division, namely the CRC code. Here the CRC code is "100". The padded input data becomes "11001010100" and will be sent to the receiver.

At the receiver side, the receiver will perform a CRC check process with a polynomial long division similar with the CRC calculation process. The difference is the dividend becomes to the received data. The divisor is still as same as the divisor used in the CRC calculation process, i.e."1011". If the remainder of the division is zero, that means the received data is the same as the transmitted data from the transmitter. Otherwise, there are errors in the received data. Figure 2.6a and Fig-

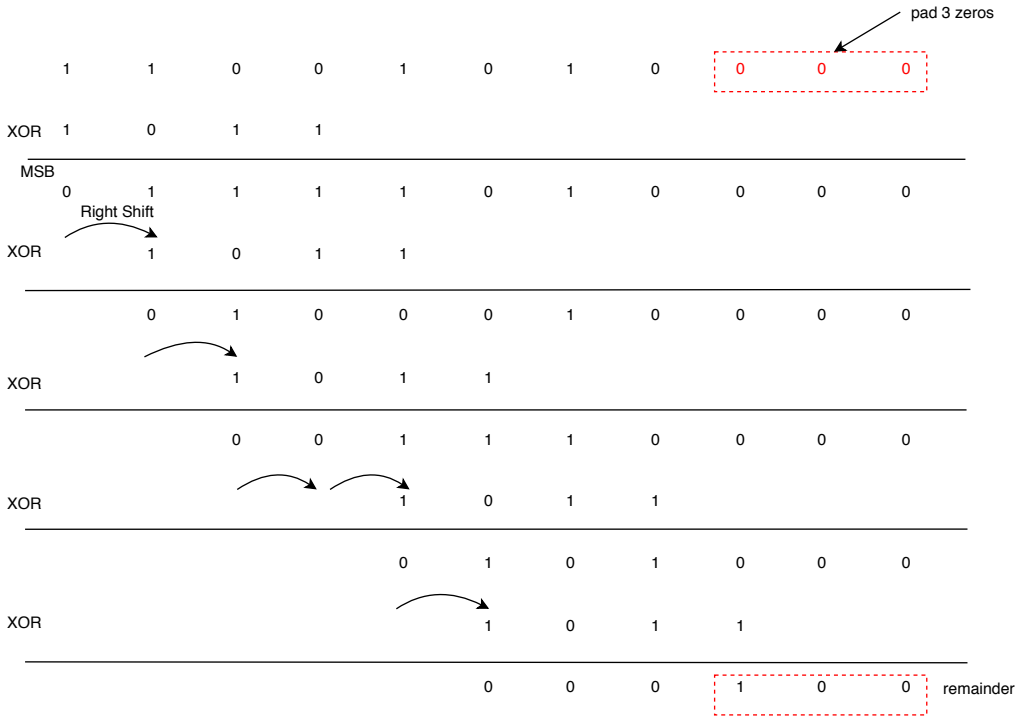


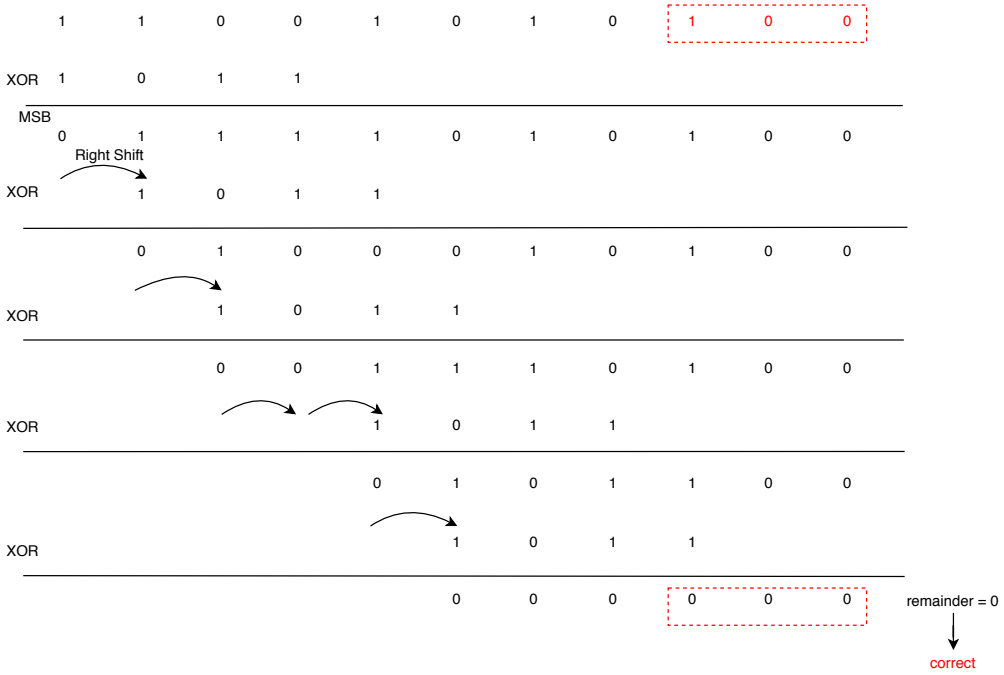
Figure 2.5: CRC calculation process. The input data is "11001010", the divisor is "1011", and the calculated CRC checksum is "100".

ure 2.6b shows two example of the CRC check process with the received data are "11001010100" and "11001011100". The remainder of the division when the received data is "11001010100" is zero, that means no errors happens in the transmission. The remainder of the division when the received data is "11001010101" is "011", that means some bits flipped during the transmission.

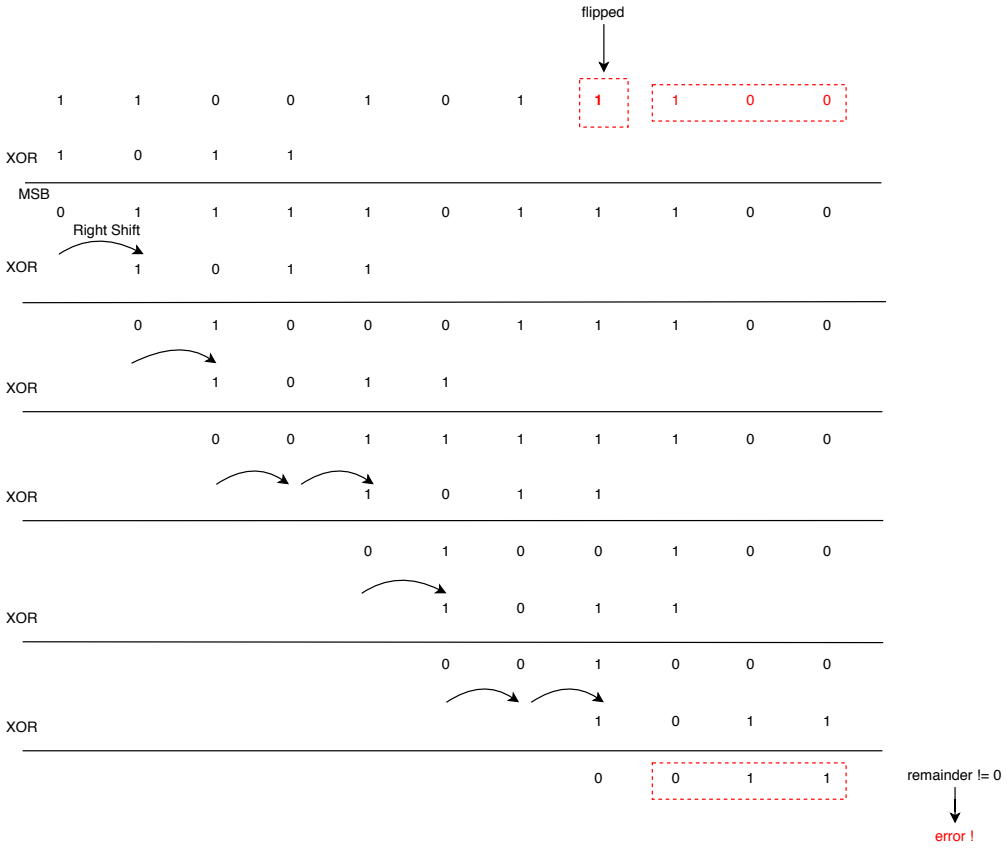
2.4 FIFO buffer

A first-in-first-out (FIFO) buffer can be used to pass data between two asynchronous systems [11, p.153]. A FIFO uses *wr* and *rd* signal to control the write and read data of the FIFO. The data will be written into the bottom of FIFO when *wr* is activated, and the data can be read from the top of the FIFO when *rd* is activated. There are two read configurations of the FIFO, first word fall through (FWFT) configuration and standard configuration. In the FWFT configuration, the data at the top of the FIFO will be passed to the read port automatically without control signals. In the standard configuration, the data at the top of the FIFO will be retrieved from the FIFO after the *rd* is asserted.

A circular queue that connects the end (tail) of the queue to the front (head) can be used to implement the FIFO buffer [11, p.154]. Two pointers write pointer and read pointer can identify the position of the write and read operation in the queue. When the buffer is empty, the write pointer and the read pointer points to the same



(a) The received data have no errors.



(b) The received data have errors.

Figure 2.6: CRC check process

position of the queue. The write/read pointer will move forward by one step when there is a write/read operation. Since it is a circular buffer, when the buffer is full, the write point and the read pointer point to the same position also. Therefore, it is impossible to distinguish these two status "empty" and "full" only depends on the position which the pointer points. Two flip-flops can be used to keep the empty and full status respectively based on the *wr* and *rd* signals.

There are two types of the FIFO buffer, synchronous FIFO and asynchronous FIFO. For the synchronous FIFO, its write and read operation are controlled by the same clock, and the asynchronous FIFO is vice versa. The asynchronous FIFO can be used as a synchronizer in a clock domain crossing (CDC) design. Since the counters inside the asynchronous FIFO must cross different clock domain, using binary counters to track the write/read pointers' movement of the asynchronous FIFO can be error-prone. Using grey counter in which the two successive counter value only differs in one-bit to do the track can eliminate the error probability when the counters work [16, p.652-653].

2.5 Metastability and synchronizer

The timing constraints of a flip-flop (FF) include setup time and hold time [16, p.216]. The setup time (T_{setup}) is a period of time that the input data of the FF must be stable before it be sampled at the clock edge. The hold time (T_{hold}) is a time interval that the input data must be stable after the clock edge. When the input data toggles during the T_{setup} or T_{hold}, which is close to the sampling edge, it can easily enter the metastable status in which the output of the FF is between "0" and "1" and cannot be decided to either "1" or "0" [16, p.612-614] [17]. The FF will enter stable status eventually after a period of time. The metastable status is unavoidable but a synchronizer can provide enough time to the FF and let it enter the stable status. The synchronizer confines the metastability condition inside itself and stops the in-between value propagate to the downstream logic [16, p.617-620]. In a clock domain crossing (CDC) design, the signals which travel from one clock domain to another clock domain must be synchronized to prevent the metastability.

The most commonly used synchronizer is two-stage FF since it is simple and robust [16, p.617-620]. Figure 2.7 shows the two-stage FF synchronization circuit. However, the knowledge of the input signals are going to synchronized is needed beforehand; the input signal of the two-stage FF synchronizer should be stable at least two clock cycles otherwise it cannot be sampled at the rising edge of the second clock [17].

A four-phase handshaking protocol is illustrated in Figure 2.8; it can be used to synchronize signals without the knowledge of the signals and the relative rate of the two clocks [16, p.617-620]. Two control signals *req* and *ack* are used in the protocol. The sender sends the data first, and the *req* signal followed. The *req* will be synchronized by the receiver, and after the receiver read the *req_sync* it will send the *ack* to the sender. The sender will synchronize *ack* and then change the

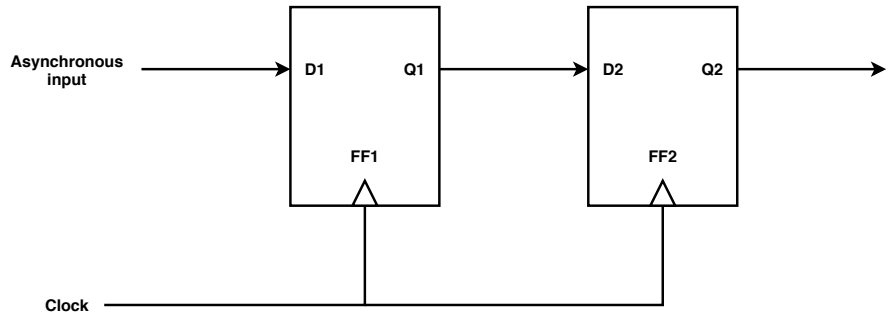


Figure 2.7: two-stage FF synchronization circuit

status of the *req*. After the *req* changes status, the *ack* also changes. There is also a two-phase handshaking protocol which is simpler than the four-phase handshaking protocol. The handshaking protocol assume the minimum knowledge of the two subsystems so that it is overhead and not efficient. It is not suitable to transfer a large trunk of data between two clock domains because of its high overhead. The asynchronous FIFO can fast transfer data between two clock domains so that it is widely used when move a bundle of data cross the clock domains [17].

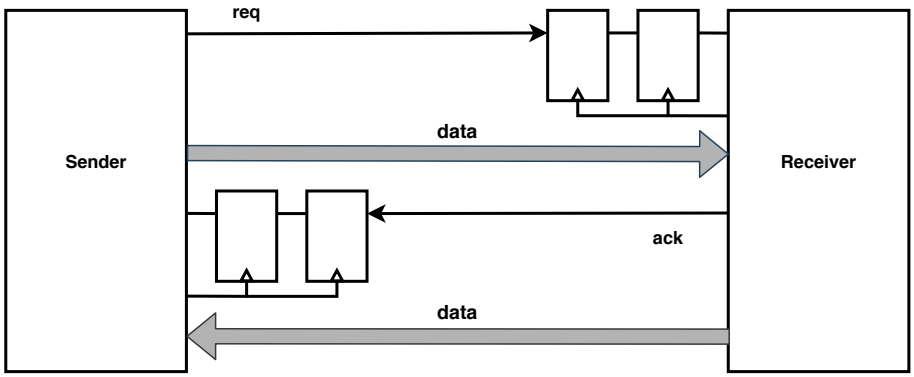


Figure 2.8: Four-phase handshake synchronization circuit

3

Design Implementation Overview

This chapter will give an overview of the design implementation of the thesis project. The following sections will describe system architecture and data flow.

3.1 System Architecture

The system architecture overview is shown in Figure 3.1. Compared with the present on-chip approach, the current communication interface between an microcontroller unit (MCU) and the PCB is adapted to SPI. In this thesis project, Arduino board is used as the MCU. Through the SPI interface, the Arduino can send or retrieve data to or from the experimental chips. The Arduino can write parameters into the RAM and read the written parameters back to ensure the correct parameters has stored in the RAM. The Arduino board can also retrieve testing result from the chip. A generic RAM access block can let the Arduino and other design blocks of the experimental chips to read/write data from/to the SRAM of the DUT. The input signal generation part can fetch the stored parameters to generate the corresponding input signals. To avoid burst error during data transaction between the Arduino and the experimental chip, CRC error detection approach is used. At the transmitter side (the Arduino or the chip), each data byte will be followed by its one byte CRC checksum before it is going to send via SPI. The receiver (the chip or the Arduino) will do a CRC check respect to the received data byte and its CRC checksum. The system can support multiple test sites running feature which rely on the SPI implementation. To fulfil all the new features and changes, a new PCB is designed.

The design implementation can be summarized into two categories: software implementation on Arduino and VHDL; hardware implementation on PCB. The next section will explain the software implementation and its data flow briefly.

3.2 Data flow

As mentioned above, the software implementation can be divided into Arduino implementation and VHDL implementation. There is an SPI interface between the Arduino and the experimantal chip. The Arduino is served as master device of the SPI interface, and the chip is severed as slave device of the SPI interface. So, the implementation of the SPI interface contains implementation on Arduino software and VHDL. Signals transfer from the Arduino to the chip need to be resynchonized

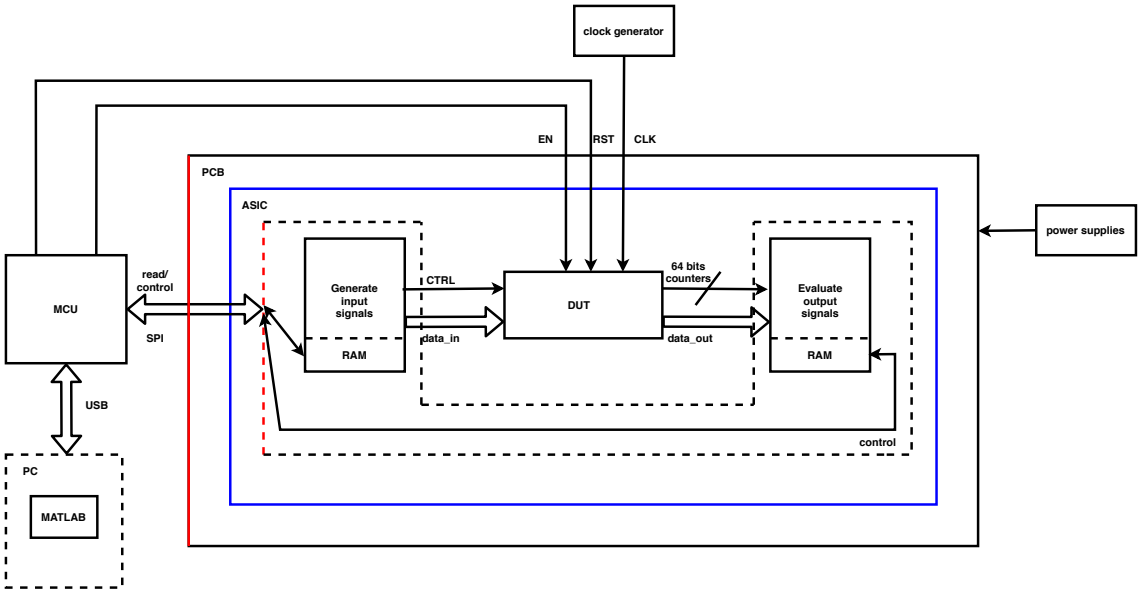


Figure 3.1: New on-chip FEC-chip evaluation setup

on the chip side first, a synchronization interface implemented on VHDL is needed.

The data flow of the software implementation is illustrated in Figure 3.2. As a master device of the SPI interface, the Arduino sends data packets to a chip. The data packets contains information about command code, addresses, and parameters which will be written into the RAM. The chip will receive the data packets via the SPI interface, resynchronize them, and decode data packets into command byte, address byte, and parameter bytes. The command byte will be decoded into control signals; some of the control signals will be used to control write/read operations of the RAM, other control signals will be used to determine which kind of data is going to send via the SPI interface. A CRC check process will check the received data packet if there are some transmission errors and report it to the Arduino via a GPIO pin of the Arduino. The data is going to send to the Arduino will be added a header and appended its CRC checksum and then send via the SPI interface. Through the header, the Arduino or PC can distinguish which chip the data belongs to. Each chip connected to the Arduino has its own unique header. After the Arduino receives data packets come from the chip, it will check if there are transmission error via a CRC check process. The retrieved data and the error information will be upload to PC for further processing. More implementation details will be explained in Chapter 4 and Chapter 5.

3.3 Design decisions

This section will present the design decisions and reasons for those decisions.

- **MCU**

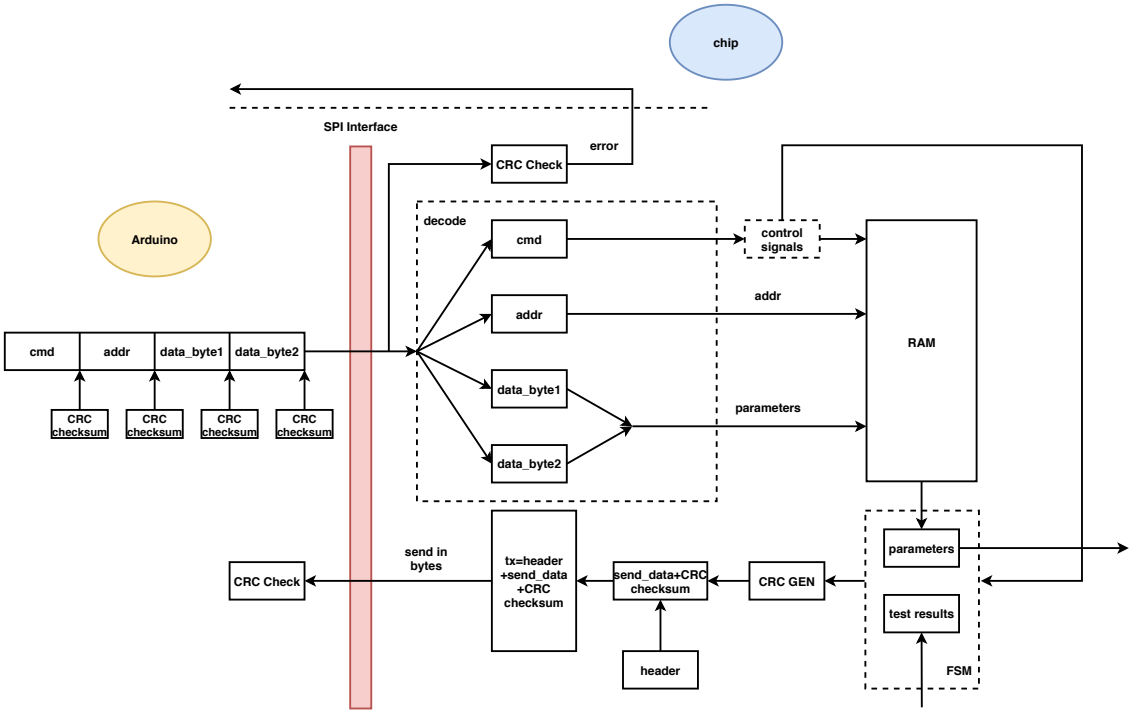


Figure 3.2: Data flow of the software implementation. The red vertical square represents for the SPI interface. The left part of the figure shows the data flow on the Arduino side, and the right part of the figure shows the data flow on the VHDL side.

In this project, Arduino board is used as the MCU. The main reason to choose Arduino is that the present evaluation setup uses Arduino board, and development based on the existing design implementation can save some time. Another reason is that Arduino is well-documented and has good forum so that it is easy to find resources to solve problems during development.

- **Communication interface**
  The communication interface between the Arduino and the experimental chip is an SPI interface. The reason to use SPI interface is that UART and I2C interface cannot support high speed data transaction so that they may not meet speed requirements for more advanced design.
- **Synchronizer**
  There are three ways to synchronize signals cross clock domains, two-stage flip-flop, handshake scheme, and asynchronous FIFO. Since the SS line, SCK line, MOSI line are single bit output from Arduino, using two-stage flip-flop as synchronizers at the chip side is enough.
- **Error detection**
  Since parity check cannot detect even number of bit error, CRC is used as an error detection scheme in this project.

4

Design Implementation on Arduino

The Arduino software mainly manages the SPI interface, generates CRC checksum, and employs CRC check to check whether the received data is right or not. The Arduino board sends commands to the experimental chip, retrieves data from the experimental chip, and checks the retrieved data. The following section will describe the CRC generate/check process, SPI interface, and multiple test kit running respectively.

4.1 CRC generating/check process

The purpose of the CRC generating/check process is to improve the reliability of the data transaction. The CRC generating/check process is easy to implement on the Arduino software. The basic operations of CRC generating process includes append zeros to the input data, MSB check, XOR calculation, and shift. The append zeros to the input data should be done once at the initial stage of the CRC generating process. The MSB check, XOR calculation, and shift operations should be done repeatedly until a remainder is obtained. Thus, these three kinds of operations should be put into a for loop, and an index is needed to control the iteration of the for loop. When the polynomial of the index is not greater than the polynomial of the divisor which indicates that a remainder, namely the CRC checksum, has been obtained, the iteration should be stopped.

The CRC check process is similar to CRC generating process, the difference is that at the initial stage the input data with the CRC checksum will be given to the CRC check function instead of appending zeros to the input data. The same for loop to calculate the polynomial division will be used in the CRC check process. when the iteration stops, check the remainder and return the check results. If the remainder is zero which means there is no error in the input data, and vice versa.

Every byte the Arduino sends to the experimental chip is needed to be followed by its CRC checksum. The experimental chip also does the same thing. Hence, the CRC generating function will be called every time there is a byte to be sent, and the CRC check function will be called every time there are two bytes received. More details about how the CRC generating/checking process are combined with the SPI

transaction will be discussed in the next section.

4.2 SPI interface

The Arduino board communicates with experiment chips via an SPI interface. The basic SPI protocol is used for bytes exchange between the Arduino board and the experimental chip, on top of it there is a more high-level interface to control the content to exchange and when to exchange [18] [19]. The SPI protocol is a master-slave fashion protocol, the Arduino board is a master device which initiates the SPI communication, configures the slave clock frequency(SCK), transmission speed, transmission mode, transmission order(MSB first or LSB first), and decides the data value and timing to transaction. The experimental chip is a slave device which only activates when the chip select pin is set to low. In this thesis project, one SPI master communicates with two SPI slaves in a star-topology configuration. The two SPI slaves works with the same clock control (SCK), share the data lines (MOSI, and MISO), but are dedicated with independent chip select pins (SS). The following two subsections will illustrate the basic SPI protocol implementation on the Arduino board and a message exchange protocol on top of it respectively.

4.2.1 Basic SPI protocol

The basic SPI protocol defines the one byte data exchange process. There are two kinds of implementation of the basic SPI protocol on Arduino, software SPI and hardware SPI. The software SPI means to use an Arduino provided SPI library to complete the SPI protocol, and the hardware SPI means to manipulate the SPI related registers of the processor of the Arduino board to complete the SPI protocol. The hardware SPI can save time for calling the SPI library to reach faster SPI operations. However, the hardware SPI requires the user or developer to have good understanding on microcontroller, including knowledge on registers and interrupt management. The low-level programming also increases the maintaining difficulties. Thus, in this thesis project, the software SPI is used.

Several Arduino functions can be used to complete the one-byte SPI transmission process. Figure 4.1 illustrates a complete SPI transaction flow, which starts with SPI.begin(), the SPI libary is called and the SCK and MOSI are pulling to low, and the SS is pulling to high [14]. Then followed by SPI.beginTransaction() calling with SPISettings inside, the SPI port starts to use with certain transmission speed, order, and mode. When the Arduino is configured as a master device, the maximum SPI speed is half of MCU's CPU speed [12] [13]. In this project, the transmission speed is 4 MHz, the data order is most significant bit first (MSB), the mode is mode 3. The chip select pin (SS) is set to low to select the slave. The SPI.transfer() is used to transmit data between master and slave. Since the default state of the SCK and MOSI is set to low when the SPI.begin() is called, but the SCK is idle high for mode 3, an extra pulling SS to high should be executed before setting the SS to low to

make sure the SCK can be idle high when the data transaction begins. Noticed that the SPI communication is a full-duplex communication, that means the first coming byte data is data has already existed on the SPI bus. The data will be sent from the master to the slave via MOSI line, and the return data will be received via MISO line. After finishing the transmission, the chip select pin will be brought to high to deselect the slave. The SPI transaction will be finished with a SPI.endTransaction() calling.

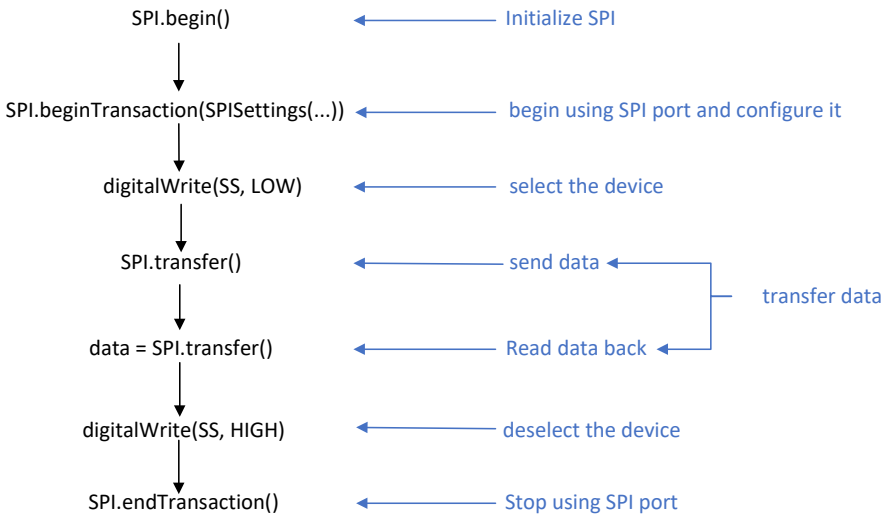


Figure 4.1: A complete SPI transaction flow on Arduino

The data operations in the basic SPI protocol include sending data and retrieving data, and these two kinds of operations are employed by SPI.transfer() call. When the master device send data to the slave device, the return value of SPI.transfer() is ignored. When the master device read data from the slave device, the return value of SPI.transfer() is the retrieval value from the slave device. Combining the two basic data operations, more kinds of operations will be created. Section 4.2.2 will discuss operations created based on the two basic data operations and how the master device manages them.

4.2.2 Message exchange protocol

On top of the basic SPI protocol, a message exchange protocol manages what kind of data is going to exchange and when to exchange between the master device and the slave device. As described in Section 4.2.1, more data operation will be created by combining the sending data and retrieving data operation, that is, more commands will be created. There are four commands that Arduino can send to the experimental chip, **writeData**, **readDataTest**, **readDataParam**, and **fetchDataParam**. The **writeData** and **fetchDataParam** only send data (commands) to the slave device; the **readDataTest** and **readDataParam** combines the sending data and retrieving data operations: first sending command to the slave device then retrieving data

back from the slave device. The **writeData** command can write data to a certain address of the RAM. The **readDataTest** command can read the test results from the experimental chip. The **readDataParam** command is used to read back the parameters written into the RAM. With **writeData** and **readDataParam** commands, parameters can be written into the RAM and can be checked whether it has been written correctly to the RAM. The **fetchDataParam** command can inform the other design blocks to fetch the parameters stored in a specific address of the RAM. Table 4.1 shows the command name and its command code.

Table 4.1: Table of command name and command code.

command name	command code
writeData	0x01
readDataTest	0x02
readDataParam	0x03
fetchDataParam	0x04

Since there might be more than one experimental chip run in parallel or concurrently, the commands send to the experimental chips should specify which chip it will send to. In other words, the Arduino board should send commands to the slave with a dedicated chip select pin. Also, the master device should be able to distinguish which chip the coming bytes belongs to. Hence, a header will be added to the coming data before sending to the master device to indicate the source of the data. In order to improve the reliability of the data sent to the slave device, each byte sent to the chip will be followed by its CRC checksum so that the slave device can carry out CRC check process. Symmetrically, the Arduino side should also perform CRC check prcoess to bytes received from the chip to make sure the received data is correct. So, to retrieve one byte data from the chip four bytes data have to be sent to the chip; the header, the header's CRC checksum, the data, and the data's CRC checksum will return one after the other. Because the data sent to the chip will be decoded by the same decoding component which decodes the writeData data packet, all packets send to the chip should be keep same length. The writeData data packet includes four bytes data contains command code, address, and parameters, and four bytes CRC checksum to each of the byte, so it is eight bytes long in total. That means all data packets send to the chip should be eight bytes long.

The command code is known beforehand so its CRC checksum can be calculated via a MATLAB script without calling CRC generating function. However, the address or parameters which are going to be written into the RAM is a unknown knowledge to the program so their CRC checksum should be calculated with CRC generating function call.

There are four functions which are employed in Arduino software to complete the four commands' function mentioned above: **writeData(uint8_t addr, uint16_t data, int chip_select)**, **readDataParam(uint8_t addr, uint8_t data_byte[ ], uint16_t error_byte[ ], int chip_select)**, **fetchDataParam(uint8_t addr, int chip_select)**, and **readDataTest(uint8_t data_byte[ ], uint16_t er-**

ror_byte[ ], int chip_select). More details about how these functions work will be discussed below.

writeData(uint8_t addr, uint16_t data, int chip_select)
The **writeData(uint8_t addr, uint16_t data, int chip_select)** function is employed to send data to a pointed address of a specific experimental chip. Inside the writeData function, the SS is brought to low to select the chip, and then send the command, address and data, each will be followed by its CRC checksum, finally raise up the SS line to deselect the chip. Since there is no need to read back the exchanged data from chips in this situation, the returned data can be ignored. Table 4.2 shows the data stream is send to the chip of writeData command (Assuming data 0x0A0B will be written into address 0x01 of the RAM). The left column indicates the order of the sending byte. For example, the first sending byte is the command which value is 0x01.

Table 4.2: Table of writeData sending packet.

index	content	send
1	command	0x01
2	command's CRC checksum	0xCA
3	addr	0x01
4	addr's CRC checksum	0xCA
5	data_byte1	0x0A
6	data_byte1's CRC checksum	0xEC
7	data_byte2	0x0B
8	data_byte2's CRC checksum	0x26

readDataParam(uint8_t addr, uint8_t data_byte[ ], uint16_t error_byte[ ], int chip_select)
The **readDataParam(uint8_t addr, uint8_t data_byte[ ], uint16_t error_byte[ ], int chip_select)** function can be used to read parameters stored in a specific address of the RAM of the chip. The sending command part is similar to the writeData function. The last two data bytes sent to the chip is not important in this case, so sending dummy data is okay. Table 4.3 shows the command packet is going to send to the chip. After the command packet arrives at the chip, the packet will be decoded into control signals and the corresponding data will be uploaded to the MISO line. As the SPI master device (Arduino) doesn't know when the slave data will be ready to send back, and the slave device cannot initiate a data transaction, the master has to wait for enough time until the slave data is ready. The SS line will be brought to high when the command sending is finished, after waiting for enough time to let the slave data be ready, the SS line will be brought to low again to start the reading data process. In this way, the sending command stage and the retrieving data stage are separated and the data read back is a "ready" data of the chip.

Table 4.3: Table of readDataParam sending packet.

index	content	send
1	command	0x03
2	command's CRC checksum	0x94
3	addr	0x01
4	addr's CRC checksum	0xCA
5	dummy	0xFF
6	dummy's CRC checksum	0x42
7	dummy	0xFF
8	dummy's CRC checksum	0x42

The conceptual diagram of the whole process (Assuming data stored in the address 0x01 of the RAM will be read) is shown in Figure 4.2. The returned parameters are two bytes long, and including the header and header's CRC checksum and the CRC checksum of each parameter byte, eight bytes will be returned. Since the SPI protocol is a full-duplex protocol, the first return byte is a returned value of last time's SPI.transfer() call so it can be regarded as a junk byte. Therefore, SPI.transfer() function need to call nine times at the retrieving data stage, the first time will return a junk byte, and the other eight times will return the header, the header's CRC checksum, parameters and their CRC checksum. A CRC check function should be called every time one byte parameter and its CRC checksum returned.

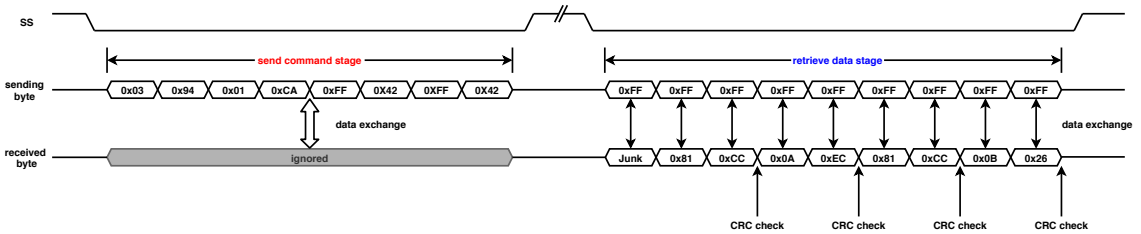


Figure 4.2: The conceptual diagram of the whole readDataParam process. The grey shadow part of the diagram means don't care data. The first time the SS is set to low, command packet is sent to the chip. After that, the SS line is raised up to high. When the SPI master device (Arduino) wants to read data back, the SS line is brought to low again. The first return byte is a junk data, then the parameters stored in the RAM with their header(assume the header is 0X81) and CRC checksum are returned.

fetchDataParam(uint8_t addr, int chip_select)

The **fetchDataParam(uint8_t addr, int chip_select)** function is used to fetch the parameters stored in a specific address of the RAM of the chip. The fetched parameters will be put into a register so that other design blocks can fetch them. Similar with the writeData function, this function doesn't care about the returned data either. So the Arduino can ignore the returned data. Table 4.4 shows the command packet is going to send to the chip when this function is called (Assuming

another design block wants to fetch the data stored in the address 0x01 of the RAM).

Table 4.4: Table of fetchDataParam sending packet.

index	content	send
1	command	0x04
2	command's CRC checksum	0xBC
3	addr	0x01
4	addr's CRC checksum	0xCA
5	dummy	0xFF
6	dummy's CRC checksum	0x42
7	dummy	0xFF
8	dummy's CRC checksum	0x42

readDataTest(uint8_t data_byte[ ], uint16_t error_byte[ ], int chip_select)
The **readDataTest(uint8_t data_byte[ ], uint16_t error_byte[ ], int chip_select)** function is used to retrieve test results from the experimental chip. The **readDataTest** is similar to the **readDataParam** function, the number of retrieved data and the CRC check times might be different. Table 4.5 shows the command packet is going to send to the chip. In this thesis project, the test results is 8 bytes long, as illustrated above, retrieving one byte from the chip needs four bytes sent to the chip, thus the 1 byte junk data plus 32 bytes data would return and 16 times CRC check should do.

Table 4.5: Table of readDataTest sending packet.

index	content	send
1	command	0x02
2	command's CRC checksum	0x5E
3	dummy	0xFF
4	dummy's CRC checksum	0x42
5	dummy	0xFF
6	dummy's CRC checksum	0x42
7	dummy	0xFF
8	dummy's CRC checksum	0x42

With these four functions, the Arduino can send commands to the chip and retrieve data from the chip if necessary. Calling these four functions with different chip select pin numbers, the Arduino can send different commands to different chips and also read data back from those chips. The next section will explained the concurrent test sites running.

4.2.3 Concurrent test sites running

There might be several test sites run concurrently or in parallel. Therefore, each command function should be called with a dedicated chip select pins. The first coming byte of every four bytes packet shows the source of the data (which chip). The data come from different chip have different headers. In this thesis project, one chip's header is 0x80, another chip's header is 0x81. With header 0x80, its CRC checksum is 0x06; with header 0x81, its CRC checksum is 0xCC.

The Arduino plays the role of the master device, and the experimental chips act as slave devices. The next chapter will present how slave devices cooperative with the master device (Arduino) to complete data transaction and its implementation.

5

Design implementation on VHDL

The VHDL implementation contains a synchronizer interface, an SPI interface, a CRC control block, a RAM access block, and an finite state machine (FSM) to mange data given into the CRC control block. Figure 5.1 displays the conceptual diagram of the VHDL implementation. Arrows in the diagram shows the data flow of the implementation mentioned in the Chapter 3.2. Each rectangle of the diagram represents one of the design blocks of the VHDL implementation. The synchronizer interface synchronizes signals travel from Arduino to the chip. The synchronized signals are connected to the SPI interface and then complete data transaction. The CRC control block mainly performs the CRC check process to data coming from the Arduino via SPI, and generates the CRC checksum to data is going to send to the Arduino and then concatenates a header and the CRC checksum to the sending data. The data received through the SPI interface will be decoded to control signals in the RAM access block. Some of the control signals will control the write and read operations of the RAM, and the other control signals will be passed into the FSM to determine which kind of data will be fed into the CRC control block. The following subsections will explain the implementation of each block in detail.

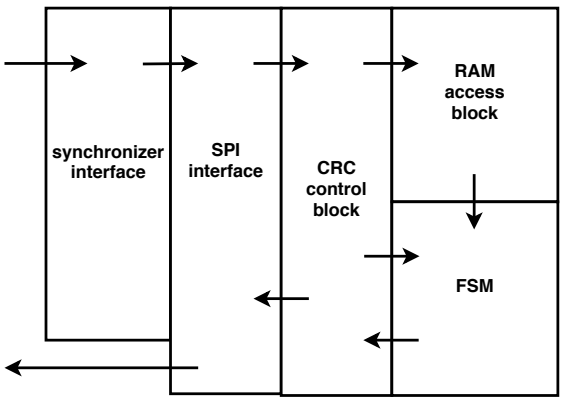


Figure 5.1: Conceptual diagram of the VHDL implementation. Each rectangle represents a design block, and arrow of the figure represents the data flow.

5.1 Synchronizer interface

In the communication between Arduino and the chip, the Arudino boards works in slow clock domain and the chip works in a much more higher clock frequency domain so that signal will transfer cross two different clock domains. Metastability

problem introduced by clock domain cross should be handled in the communication. Figure 5.2 shows the block representation of the synchronizer interface. There are three two-stage flip-flop synchronizers in the synchronizer interface to synchronize SS signal, SCK signal, and MOSI signal respectively. The MISO line speed is as same as the Arduino master speed, so the MISO line can be directly connected to Arduino MISO pin. The synchronized SS, SCK, and MOSI signals will be passed to the SPI interface. The next section will describe how the SPI interface works.

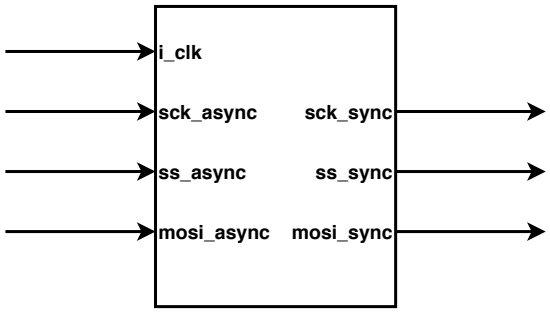


Figure 5.2: Block representation of the synchronizer interface

5.2 SPI interface

The SPI interface is in charge of data exchange between the Arduino and the chip. Figure 5.3 shows the block representation of the SPI interface. Except for four basic SPI signals, namely chip select *i_ss*, serial clock *i_sck*, master-in-slave-out signal *o_miso*, and master-out-slave-in signal *i_mosi*, there are other signals contained in the SPI interface: clock signals *i_clk* and reset signal *i_rst* are provide by the chip; an 8-bit signal *i_byte_tx* is used to store data which are going to send on the MISO line; an 8-bit signal *o_byte_rx* is used to output received data; a flag signal *o_rxDone* which lasts for one clock cycle can indicate 8-bit (one byte) data has received.

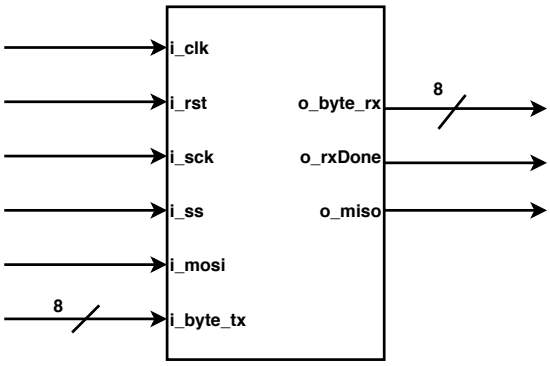


Figure 5.3: Block representation of the SPI interface

The experimental chips serve as slave devices and it should be in same data mode

as the master device, i.e. mode 3; data of MISO line and MOSI line will be changed on the falling edge of the SCK and sampled on the rising edge of the SCK. So, edge detection components that can detect the rising and falling edge of the SCK are needed. The following process only happens when the SS line is brought to low. A counter routines the number of the bits have exchanged during SPI transaction and it will augment one on every rising edge of the SCK. When all 8-bits (one byte) data have been shifted in on the MOSI line, the flag *o_rxDone* will raise up. This flag will be used to control the CRC control block, RAM access block, and the FSM block.

There are two registers are used to store data in bytes: *byte_rx* and *byte_tx*. The *byte_rx* register stores received data, and on every rising edge of the SCK one-bit MOSI data will be shifted into the least significant bit of the register to update the register. The *byte_tx* register stores data are going to send, the initial sending data *i_byte_tx* will be loaded into the register when the counter values is zero; and then on every falling edge of the SCK a zero will be shifted into the least significant bit of the register and the most significant bit of the register will be shifted out to the MISO line simultaneously.

Since there might be more than one chip are connected to the master device, a tri-state buffer is needed on the MISO output. When the SS line is high, which means the chip is de-selected, the MISO output will be in high-impedance state. To do so, data collision on the MISO line can be avoided. Correspondingly, the MISO pin of the chip should be defined as tri-state pin type in PCB design.

Figure 5.4 shows how the RX part of the SPI interface works. Suppose Arduino sends byte 0xca to the chip via the SPI interface. In mode 3, the SCK is in idle high. Noticed that there is a period of time between SS is brought to low to SCK is brought to low, that is because only when SPI data is put into the data line the Arduino starts to generate SCK. On every falling edge of the SCK, one-bit data is shifted into the MOSI line but it won't be sampled until the rising edge of the SCK. On every rising edge of the SCK the counter will augment by one, and the sampled one-bit MOSI data will be shifted into the byte_rx register. After all 8-bits get sampled the rxDone flag will raise up and the value of the byte_rx register is the byte sent by Arduino. The received byte 0xca will be passed to CRC control block. The next section will describe how CRC control block works.

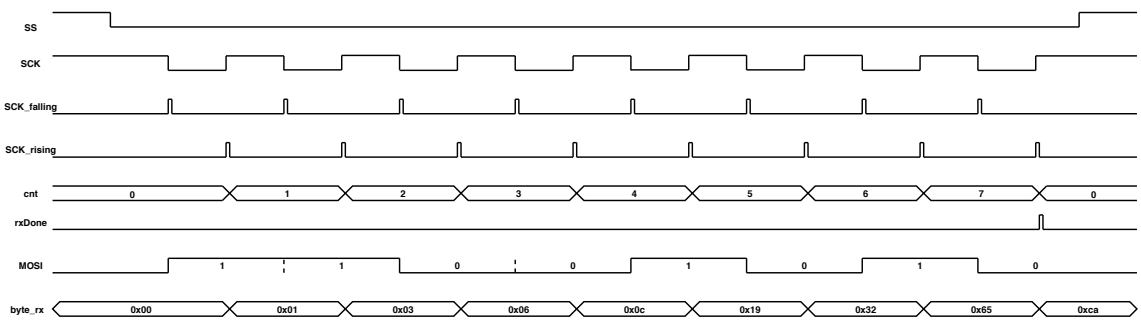


Figure 5.4: Timing diagram of RX part of the SPI interface

5.3 CRC control block

The main functions of CRC control block include CRC checking to the received data from the SPI interface and generating CRC checksum of each data byte and appending the checksum to the data byte before transmitting via SPI. Figure 5.5 shows the block representation of the CRC control block. The upper part above the dash line shows signals from the synchronizer interface and the SPI interface to other blocks, and the bottom part below the dash line shows signals from the other blocks to the SPI interface. Signal *i_ss_sync* comes from the synchronizer interface. Signal *i_load* comes from the *o_rxDone* signal of the SPI interface, and *i_byte_rx* signal is the *o_byte_rx* of the SPI interface. Signal *i_en_crcg* is a enable signal of a CRC generator component inside the CRC control block. Signal *i_load2crcg* controls data loading into the CRC generating component. Signal *i_byte2crcg* comes from the FSM and represents the input data of the CRC generating component. Signal *o_byte_tx* will be connected to the SPI interface and sent to the Arduino.

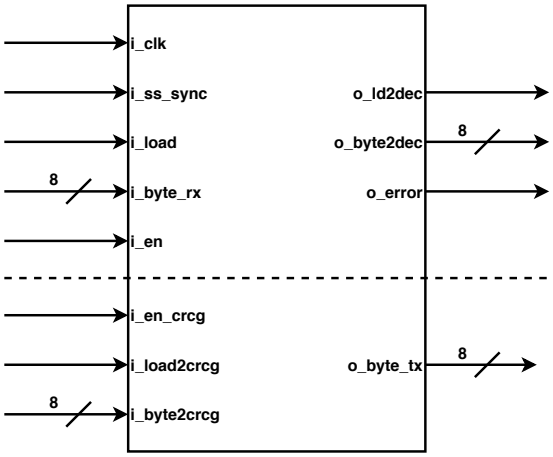


Figure 5.5: Block representation of the CRC control block

Figure 5.6 shows the block diagram of the CRC control block, where the upper branch is for stripping off the CRC checksum and passing the data bytes to RAM access block, the middle branch is for CRC checking process, and the lower branch is for appending CRC checksum to data byte and sending. Since the data stream come in one byte data followed by one byte CRC checksum, and the CRC checksums don't need to be passed to the RAM access block, a RX FIFO is needed here to store the received data byte in order. The *i_ss_sync* signal is used as enable signal of the RX FIFO. A counter is used to control the write and read control signal of the RX FIFO. The counter will augment by one every time the rxDone flag, namely *i_load* signal, raises up and goes back to zero when the counter is equal to one. The coming data byte will be written into the RX FIFO when the counter is zero, and read out from the RX FIFO when the counter is one. In this way, the first, third, fifth, and seventh arrived data will be written into the RX FIFO and the second, fourth, sixth, and eighth arrived byte(CRC checksum byte) won't be stored in the FIFO.

Signal *ld2dec* means "load to decoder" and the signal is used for informing the decoder of the RAM access block to start decoding data packets. The RX FIFO mentioned above uses a FWFT configuration so that the data written into the FIFO will be available at the data output port of the FIFO one clock after the *wr* is active. So, the *ld2dec* should be one clock later than the *wr* to indicate available FIFO data output. Signal *byte2dec* means "bytes to decoder". The data output of RX FIFO will be buffered at a register and then be passed to the RAM access block after the *ld2dec* signaled.

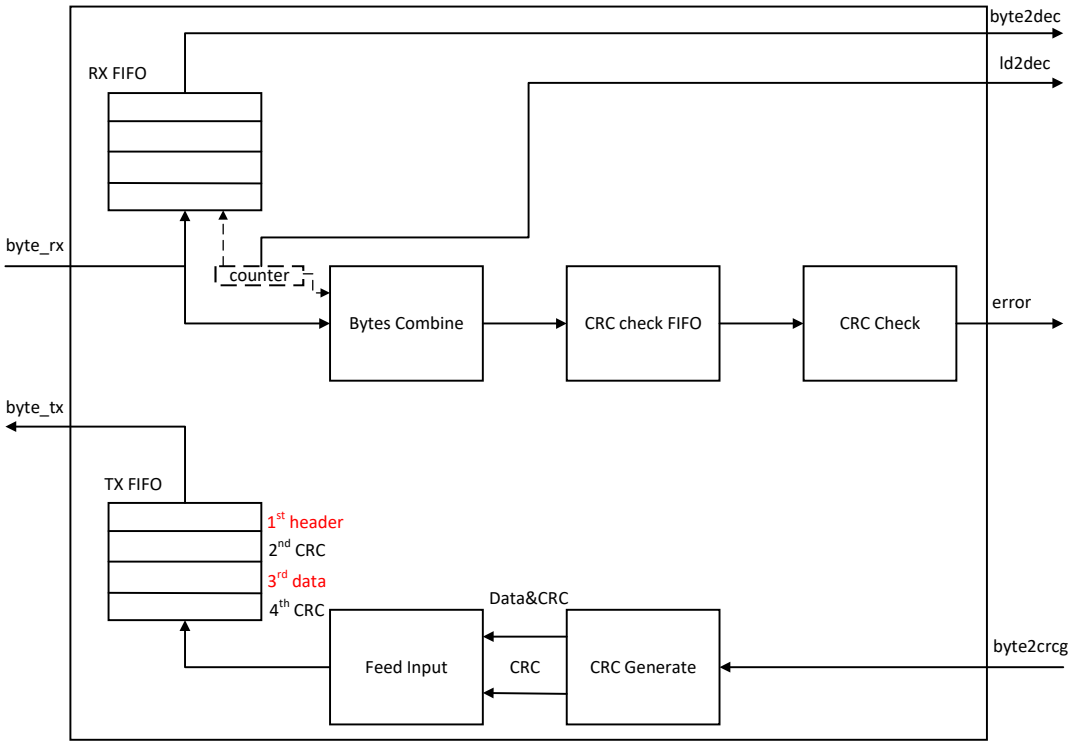


Figure 5.6: Block diagram of CRC control block. The *byte_rx* signal is coming from SPI interface, the *byte_tx* signal connects to the TX line of the SPI interface. The *byte2dec* is a signal will pass to the RAM access block to decode. The *ld2dec* signal indicates the *byte2dec* is going to shift out from the CRC control block. The *byte2crcg* is a signal comes from other design blocks and will be fed into the CRC generator. The error signal is a output signal to indicate that the bursting error exists in the received bytes.

Input data of a CRC check process should be data together with its CRC checksum, that means the input data length should be two bytes long. A bytes_combine block in which two coming bytes will be combined into one word data packet will be needed. The counter mentioned above can be used to control the one word packet shift out. When the counter is one, one word packet will be shifted out. The CRC check process takes some time, and before one CRC check process finished the new data fed into the CRC check component will be ignored. Hence, a FIFO is needed

to temporarily buffer the one word packet until the last time's CRC check will be done. A finish flag of the CRC check component will be used as the read control signal of the FIFO. Only when there is no ongoing CRC check process, i.e. the finish flag raised up, the new data can be fed into the CRC check component. The result of CRC checking process will be given to the *error* port.

Input data bytes are going to feed into the CRC generate component may be parameters stored in the RAM, or the testing result generated by the experimental chip. A finite state machine will be used to choose proper data give into the CRC generating block based on the instructions' decoding results. The FSM will be explained later on. The generated CRC checksum and the original data byte will be sent into a feed input component. The function of the *feed_input* component is to add data header and append CRC checksum to data byte so that the data stored in the TX FIFO will be header, header's CRC checksum, data byte, and CRC checksum of the data byte in turn. In this way, the received data on the Arduino side are also in one byte data followed by one byte CRC checksum format, and the Arduino board can know in which chip the data come from. A component called *N_clk_timer* works in the *feed_input* component to make sure the *feed_input* component only works four clock cycles to get the four bytes output once one byte checksum is generated. The following two parts will explain the implementation of the CRC generator and the CRC checker.

5.3.1 CRC generator

As same as the CRC generating implementation on the Arduino, CRC generation process starts with zero padding to the input data, then checking MSB, XOR calculation, and shifting operations will be repeatedly executed until getting the CRC checksum. An finite state machine can be used to implement the CRC generator. Figure 5.7 shows the state machine diagram of CRC generate process. There are seven states in the state machine, *s_init*, *s_load*, *s_cmp*, *s_msbck*, *s_xor*, *s_shiftr*, and *s_done*. Assume that the FSM is initially in the *s_init* state. It moves to the *s_load* state when there is a load signal asserted. Inside the *s_load* state, the input data will be padded a series of zeros, the number of padded zeros is equal to the width of CRC checksum. The divisor will also be padded a series of zeros to make the new padded zeros dividend and the divisor have the same width. Then the FSM moves to the *s_cmp* state. An index is used to indicate in which bit the CRC generation process is performed. If the index is greater than the width of CRC checksum in this state, it implies that the polynomial of the dividend is greater than the polynomial of the divisor, and the FSM will enter the *s_msbck* state; otherwise, it indicates that a CRC checksum has already generated, and the FSM will enter the *s_done* state. A finish flag will raise up to show the completion of the generation process in the *s_done* state. In the state *s_msbck*, when the MSB of the dividend is '1', the FSM will enter the state *s_xor* and calculate the XOR of the dividend and the divisor; otherwise, it will enter the state *s_shiftr*, both dividend and the divisor will right shift by one bit until the MSB of the dividend is '1'.

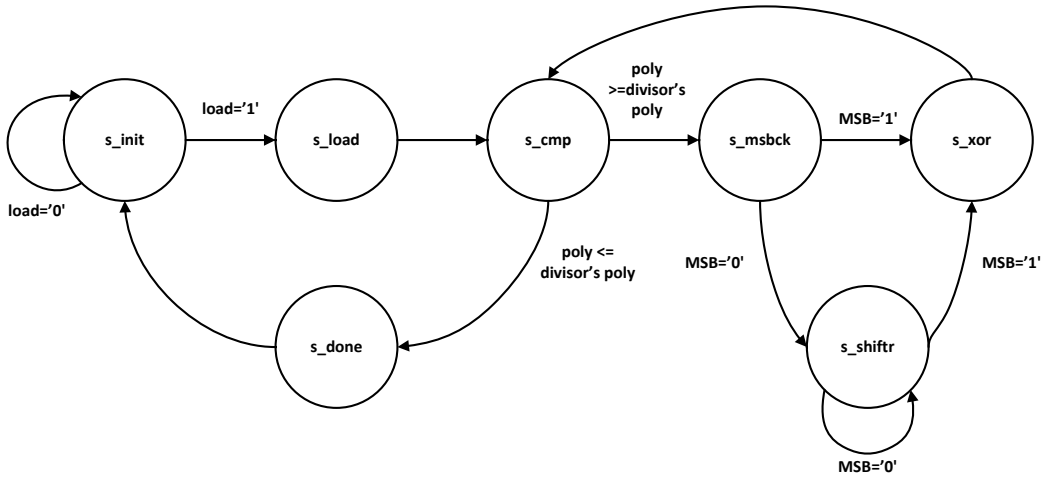


Figure 5.7: State diagram of CRC generation process.

5.3.2 CRC checker

The CRC check process is similar to the CRC generating process, but there is no need to pad zeros to the input data. Figure 5.8 illustrates the state diagram of the CRC checking process. The state diagram is similar to the state diagram of the CRC generating process, except the transition from the *s_cmp* state to the *s_erck*. At the *s_cmp* state, if the index is not greater than the width of CRC checksum, it indicates that the division should stop, the value of the remainder is the division result, and the FSM will move to the *s_erck* state. If the remainder is zero that implies there is no transmission error of the input string; otherwise, an error flag will raise up.

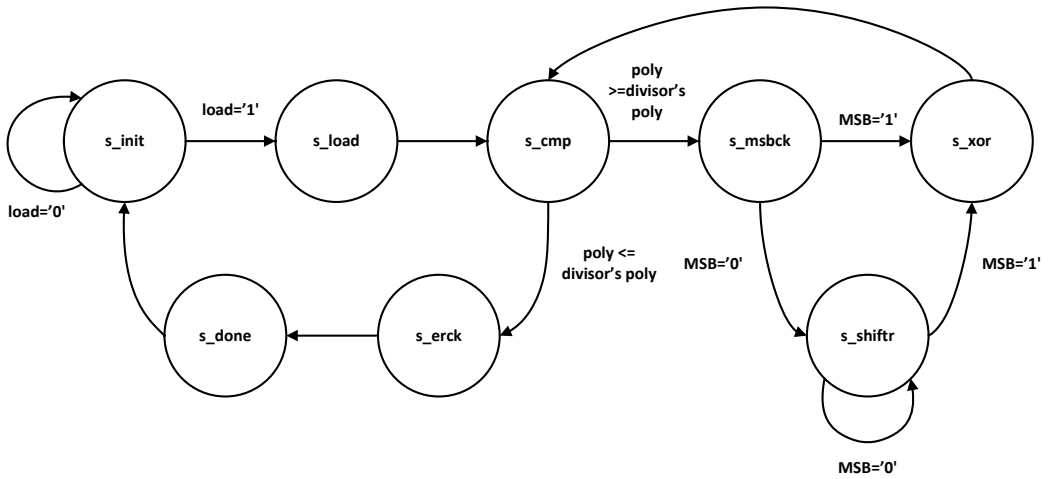


Figure 5.8: State diagram of CRC checking process.

5.4 RAM access block

Data bytes signal *byte2dec* will be passed to the RAM access block from the CRC control block. The main function of the RAM access block is reading or writing data to or from RAM. Figure 5.9 presents the block representation of the RAM access block. Signal *i_ss_sync* comes from the synchronizer interface and is used as a enable signal of a packet decoder. Signal *i_din* is the *byte2dec* of CRC control block. Signal *i_ld* is the *ld2dec* signal of the CRC control block. *o_WRPARAM*, *o_RDTST*, *o_RDPARAM*, and *o_FTPARAM* are control signals are going to send to an FSM to trigger the FSM enter corresponding states. Parameters stored in the RAM can be read from the port *o_para_out*. Figure 5.10 shows the block diagram of the RAM access block. The *ld2dec* signal which is provided by the CRC control block is used for controlling the data load into packet decoder. The packet decoder decodes the received data bytes into instructions, address, and parameters which will be written into the RAM. The instruction decoder decodes the instructions into control signals. Some of the control signals can control the read and write action of the RAM, and other control signals will be ported out to trigger the FSM enter different states. Decoded parameters can be written into RAM when write control signal is asserted, and the data stored in the RAM can be read when the read control signal is asserted, and it will be sent to parameter out register. The following subsections will introduce the packet-decoder, instruction decoder, and dual-port ram in detail.

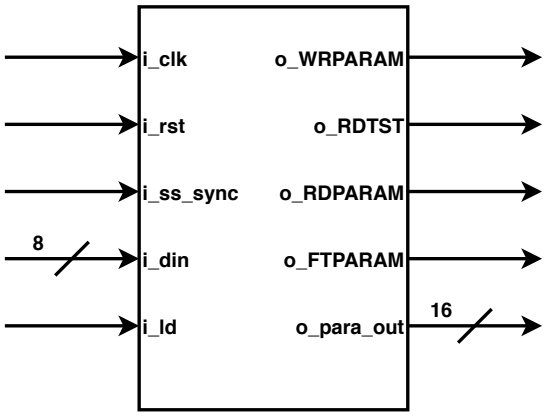


Figure 5.9: Block representation of the RAM access block

5.4.1 Packet-decoder

The packet decoder decodes the received data bytes into instructions, write/read address, and parameters which will be written into the RAM sequentially. The *ld2dec* signal which is provided by the CRC control block acts as a load signal to control bytes loading into the packet decoder. In this thesis project, the first coming byte from the control block will be decoded to instruction byte (command), the second byte will be decoded to the address byte, and the third and fourth byte will be decoded to the parameters byte. Since all coming bytes need to go through this

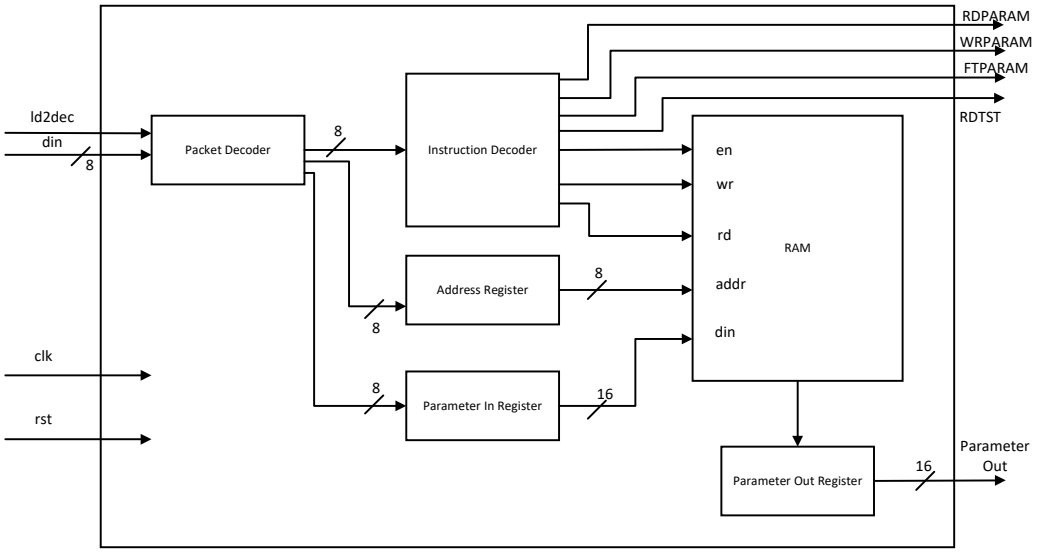


Figure 5.10: Block diagram of RAM access block. The blue words shows the signal provided by other design block. The ld2dec signal and din signal come from the CRC control block. The RDPARAM, WRPARAM, FTPARAM, and RDTST signal are decoding results of the instruction decoder.

packet-decoder, all function command packets sending from the SPI master Arduino should be in four bytes format. For instance, when read test result, the read test command packet contains four bytes and only the first byte contains useful information: the first byte is the instruction byte, and the other three bytes are dummy bytes.

To avoid triggering wrong read/write operations of the RAM, the earlier arriving byte should wait for the later arriving bytes so that data bytes sent to the Instruction-decoder, address register, parameters-in register can be aligned to a same time instance. For example, assuming the former instruction byte is 0x01 (writeData), and the new arrived instruction byte is 0x03 (readDataParam). When the new address byte and parameters bytes haven't arrived, the address byte and the parameters bytes of the writeData command will be used. As a result of this, wrong address's data will be retrieved from the RAM. When all four bytes have been decoded, a flag will be raised up. The instruction byte will be decoded into control signals in the instruction-decoder. The next subsection will explain how the instruction-decoder works.

5.4.2 Instruction-decoder

The flag signal of the packet decoder will serve as an enable signal of the instruction decoder. The instruction decoder decodes the instruction bytes according to the bytes value and then asserts corresponding control signals. Table 5.1 shows the decoding rules of the instruction-decoder. There are two types of control signals will be generated by the instruction-decoder: the active low control signals *en*, *wr*, and *rd* which are used to control the read and write operation of the RAM; and active high control signals *RDPARAM*, *WRPARAM*, *FTPARAM*, and *RDTST* which are used as trigger signals to trigger the FSM enters different state. For example, when the instruction byte is 0x03, that is, readDataParam, the *en* signal and *rd* signal is set to low, the *wr* signal is set to high, and the *RDPARAM* signal is also set to high. The *WRPARAM*, *FTPARAM*, and *RDTST* signal are set to low. When the decoded result is read test result, the RAM should be blocked, either write or read operation are allowed.

Table 5.1: Table of decoding rules of the instruction decoder

i_din		o_en	o_wr	o_rd	o_WR-PARAM	o_RD-PARAM	o_RD-TST	o_FT-PARAM
writeData	0x01	'0'	'0'	'1'	'1'	'0'	'0'	'0'
readDataParam	0x03	'0'	'1'	'0'	'0'	'1'	'0'	'0'
readDataTest	0x02	'1'	'1'	'1'	'0'	'0'	'1'	'0'
fetchDataParam	0x04	'0'	'1'	'0'	'0'	'0'	'0'	'1'

5.4.3 Simple Dual-port RAM

There are several types of dual-port RAM implementations. One collected requirement [7] of this thesis project is the Arduino board should be able to send parameters to the chip, and those parameters should be stored at the SRAM of the chip, so that other parts of the chip can fetch some parameters when needed. That means the read operations should independent with write operations, only when the read signal is active, data can be read from the RAM. Thus, the simple dual-port RAM has *we* control signal and *rd* control signal to control write and read operations separately. The SRAM is a $256 \cdot 16$ large RAM, the address width of the RAM is 8, and the data width of the RAM is 16.

5.5 FSM

As mentioned above, there will be two types of data of the experimental chip can be sent via SPI interface: the parameters stored in the RAM, and the test results. An FSM is needed to manage which kind of data is going to be fed into the CRC generating block and then be sent via SPI interface. Figure 5.11 shows the block representation of the FSM. Signal *i_WRPARAM*, *i_RDPARAM*, and *i_RDTST* are control signals come from the RAM access block. Signal *i_para_out* comes from the parameter register of the RAM access block. Signal *i_din_test* is the test

results generated by the chip. The output signals of the FSM block include data bytes *o_byte2crcg* which are going to pass to the CRC control block, enable signal *o_en_crcg* which controls the CRC generator of the CRC control block, and load signal *o_load2crc* to control loading data into the CRC generator.

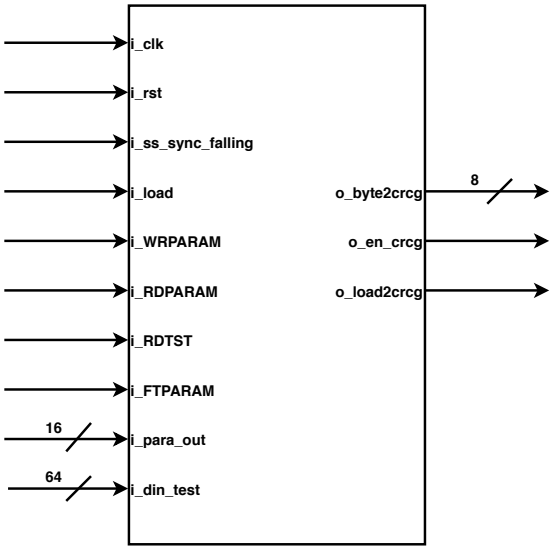


Figure 5.11: Block representation of the FSM

There are seven state of the FSM, *s_init*, *s_wr*, *s_ftpara*, *s_wtrdp*, *s_wtrdt*, *s_rdtst*, *s_rdpara*. On the Arduino implementation, commands need reading data back are divided into two stages: sending command stage and retrieving data stage. These two stages are separated through SS high/low operation, SS will be brought to high after sending command finished, and then be brought to low to enter the retrieving data stage after waiting for enough time to ensure the slave data is ready. As corresponding to this, the states of the FSM can be divided summarized into three categories: states without reading data back, namely state *s_wr* and *s_ftpara*; states waiting for an SS falling edge, namely state *s_wtrdp* and *s_wtrdt*; and states reading data, i.e.state *s_rdtst* and *s_rdpara*. When RDPARAM signal or RDTST signal are raised up to high, the FSM will enter wait reading state until a synchronized SS falling edge is detected. Then the FSM will switch to the reading data state.

Figure 5.12 illustrates the state diagram of the FSM. The following part will describe control path of the FSM first, the data path of the FSM will be described later on. A signal called *op_state* is the concatenation of the control signals and will be used to indicate the FSM enter different states. Starting with the state *s_init*, when WRPARAM raises to high, the FSM moves to the state *s_wr* and then jumps back to the state *s_init*. When RDPARAM raises to high, the FSM jumps to the state *s_wtrdp*. Inside the state *s_wtrdp*, the FSM will switch to the state *s_rdpara* once a synchronized SS falling edge is detected. Once the FSM enters the state *s_rdpara*, a counter will be triggered. The counter is used to count how many bytes of data have been read by the Arduino, that is, how many bytes the Arduino and the chip have been exchange with each other. So, the flag signal *rxDone* of the SPI interface can

be used to control the counter's increment. The counter will augment by one every time there's a high *i_load* signal of the FSM (rxDone signal of the SPI interface). As mentioned in 4.2.2, for retrieving data from the experimental chip, there must be four times the number of data bytes plus one dummy data comes from the Arduino board. Hence, when the counter counts up to four times the number of data bytes plus one, the FSM moves back to the *s_init* state. The reading test results' states transition is similar to the transition of reading parameters, the difference is the upper limit to let the state machine jump back to the state *s_init*.

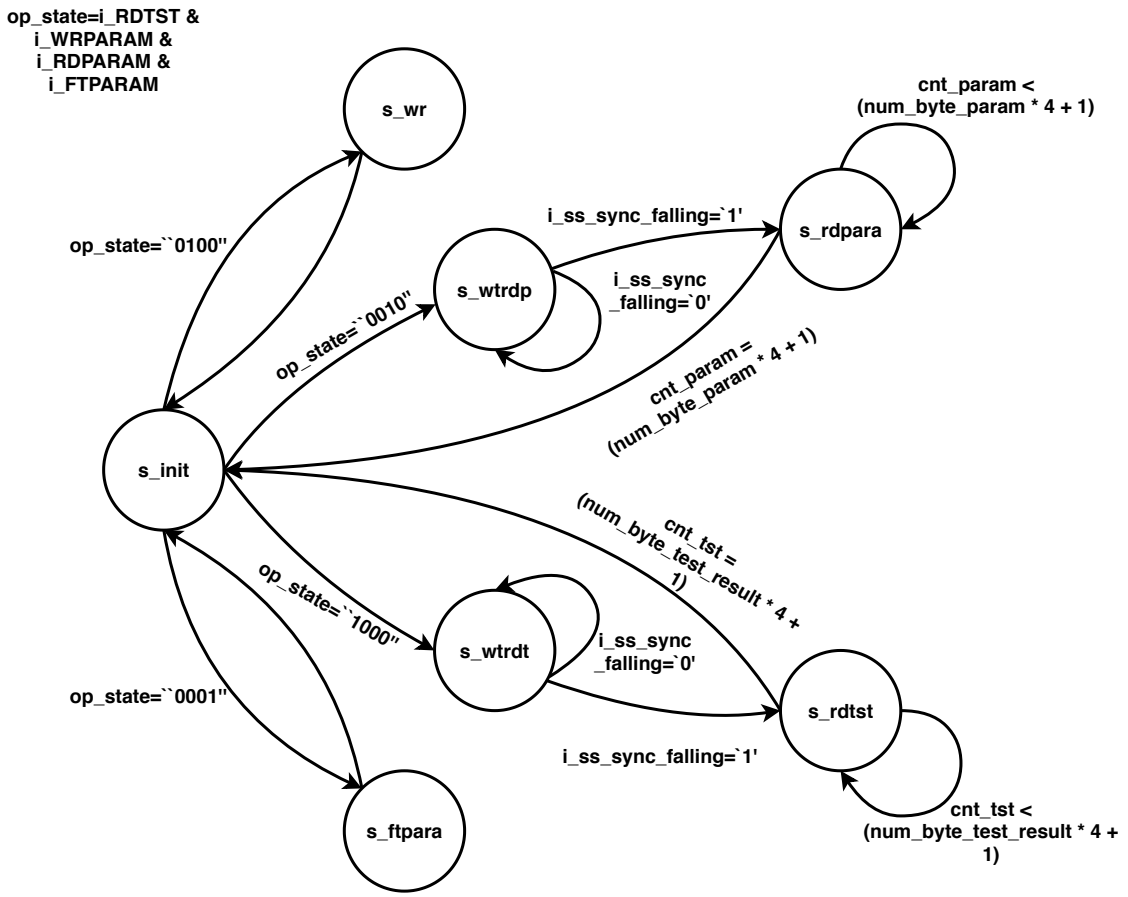


Figure 5.12: State diagram of the FSM

The above paragraph described the control path of the FSM, next the data path will be explained. The data path of the FSM mainly determines the data bytes will be given to the CRC generator, signal *byte2crcg*; the enable signal of the CRC generator, signal *en_crcg*; and the load data signal of the CRC generator, signal *load2crcg*. Since data bytes passed into the FSM may be longer than one byte but the CRC generator and the SPI interface can only manipulate data in bytes, so a component called *split* is employed to split a trunk of data into bytes, and it will shift out one byte data every time a enable signal of the component is asserted. The split data should be ready before the *load2crcg* is asserted to ensure no repeat data will be loaded into the CRC generator. The split bytes data will be assigned to *byte2crcg* and go through the CRC generating block to generate CRC checksum.

The signal *en_crcg* will be enabled at the state *s_rdpara* and the state *s_rdtst* since the CRC generating block only works at these two states.

5.6 Top Module

As explained at the beginning of this chapter, the top module of the VHDL implementation connects the synchronizer interface, the SPI interface, the CRC control block, the RAM access block, and the FSM. An edge detection component is used to detect the synchronized SS falling edge will generate the *i_ss_sync_falling* signal and pass it to the FSM. Figure 5.13 shows the block representation of the top module. Signal *i_sck*, *i_ss*, *i_mosi*, and *o_miso* are SPI interface related signals. Signal *i_en* controls subblock CRC checker of the CRC control block, and the CRC check result *o_error* will be reported back to Arduino board. The Arduino board and other design blocks of the chip can fetch parameters stored in the RAM from the *o_para_out* port when *o_FTPARAM* is raised up to high. When *o_RDTST* is raised up to high, the test results *i_din_test* generated by the chip will be uploaded to the Arduino board via the SPI interface.

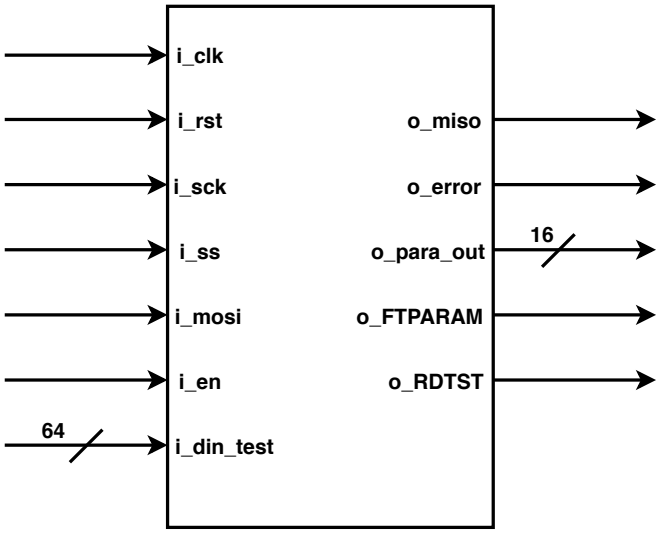


Figure 5.13: Block representation of the top module on VHDL implementation

6
PCB Design

The designed testing infrastructure should be able to used in testing other experimental chips, so the present PCB design used in testing the FEC chip need to be upgraded. The communication interface between the Arduino and the chip will be upgraded to SPI interface, the SPI interface uses 4 pins instead of 32 pins used before for data transmission between the Arduino and the chip so that many pins of both Arduino and the chip can be saved. The existing PCB is provided power by external lab power supplies which is expensive and inconvenient. The power management design of the new PCB should be able to supply more power on board to reduce the cost of using external power supplies while keeping the power dissipation and the thermal characteristics in proper range. The existing PCB uses voltage translators to translate 5 V signals of the Arduino to 0.8 V. The new PCB will still use voltage translators to translate voltage of signals but the number of the translators will change. The new PCB design of the project can be divided into three parts: power management, the experimental chip pin design, and the voltage level translating. The following section will present design related to these parts.

6.1 Power management

The power management of the PCB is an important part and it should follow these principles:

- low-cost, simple, adaptable, safe, and accurate;
- supply more power on-board if it's possible;
- low power dissipation;
- low heat generation;
- enough design margin so that it can be updated to other chip testing without re-design the PCB.

Starting from the second principle, the task is supplying more power on-board to reduce the external power supplies using. There are three types of power on the PCB: *VDD_core_internal*, *VDD_core_external*, and *VDD_IO*. The *VDD_core_internal* and *VDD_IO* are generated on the PCB, and the *VDD_core_external* is provided by external lab power supplies. Since there are two experimental chips on the PCB, there should be two *VDD_core* provided to the two chips separately. One of the *VDD_core* should be able to switch between the *VDD_core_internal* and *VDD_core_external*. As there are two chips connected to the Arduino, there should

be two *VDD_core_internal* generated. Voltage regulators can be used to supply power on the PCB. There are two types of voltage regulators: linear voltage regulator and switching voltage regulator. The linear voltage regulators are cheap, simple, low noise but with low power efficiency while switching voltage regulators are expensive, complex, higher noises but with high power efficiency. As one of the PCB design principles is low-cost and simple so linear voltage regulators will be the starting option. Figure 6.1 shows a simple schematic of typical application of adjustable linear voltage regulator. VIN is provided by rail supply. When the voltage of EN pin falls into the threshold range the voltage regulator will be enabled. The FB provides a feedback voltage. The output voltage of the voltage regulator is adjustable by changing the radio of R_1 and R_2. The output voltage of the voltage regulator can be calculated by

$$V_{out} = V_{REF} \cdot (1 + \frac{R_1}{R_2}) \tag{6.1}$$

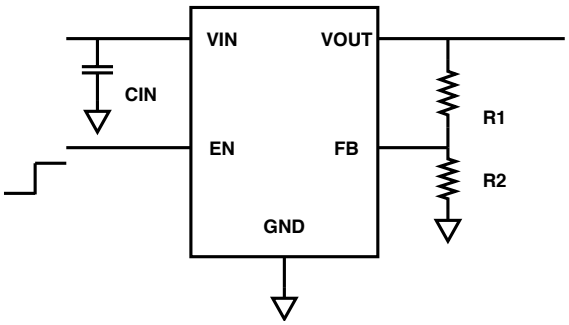


Figure 6.1: Simple schematic of typical application of adjustable linear voltage regulator.

Power dissipation and thermal characteristic are two key factors of choosing linear voltage regulators. The estimated power dissipation of linear voltage regulator can be calculated by:

$$P_{regulator} = (V_{in} - V_{out}) \cdot I_{load} \tag{6.2}$$

With certain I_{load}, small $V_{in} - V_{out}$ can reduce the power dissipation of the voltage regulator. The dropout voltage is defined as the minimum difference value of $V_{in}$and V_{out} to let the regulator work within specification. Low-dropout (LDO) linear voltage regulator is a kind of voltage regulator with small dropout specification and it will be used in the project.

Thermal resistance θJA(°C/w) can be used to choose ICs with good thermal characteristic, it shows the number of degree the chips will heat up above the ambient air temperature per each watt of drained power. In general, components which thermal resistance multiplied by their power dissipation is less than 10°C belong to "low power" components. With around θJA(°C/w)=100(°C/w), the approximate power dissipation is 100 mW, that means a component which power dissipation is above 100 mW needs extra thermal path to dissipate the heat. If the thermal resistance of

a component is about 20°C/w to 30°C/w, the dissipated power can up to 500 mW. To simplify the design, 100 mW can be used as a upper limit value of choosing voltage regulators.

Table 6.1 shows the design requirements regarding the voltage and current range of the *VDD_core_internal* and *VDD_IO* [20]. Observing the given requirements, it can be seen that the minimum voltage and maximum voltage of *VDD_core_internal* are smaller than the *VDD_IO*, but the minimum current and maximum current are larger than the *VDD_IO*. The minimum voltage of the *VDD_core_internal* is 0.25 V which is too small to find a LDO regulator. Figure 6.2 shows a solution [21] to provide small voltage combined with voltage regulator and voltage reference. Instead of connecting the R_2 to the ground, connecting it with the V_{REF} pin of the voltage reference, the voltage reference should be higher than the V_{FB} (the V_{REF} in Equation 6.2). Noticed that the current on the the V_{REF} pin will also flow through R_1, so the V_{out} can be calculated by:

$$V_{out} = V_{FB} - \frac{R_1 \cdot (V_{REF} - V_{FB})}{R_2} \tag{6.3}$$

From the equation above, it can be seen that with a fixed V_{FB}, the larger the V_{REF} is the smaller the V_{out} will be. When the required V_{out} is higher than V_{FB}, there is no need to connect the R_2 to the voltage reference, the R_2 should be connected to ground; otherwise, connect the R_2 to the output of voltage reference.

With this voltage reference connects to the voltage regulator solution, voltage regulator which can provide 0.8 V to 3.3 V, and 0.6 A to 3 A output is acceptable for both *VDD_IO* and *VDD_core_internal*. TPS7A7002 [22] is used in this project. TPS7A7002 is a LDO voltage regulator which can provide adjustable voltage output as low as 0.5 V. Its thermal resistance is θJA(°C/w)=47(°C/w), and the maximum current output is 3 A. The reference voltage of TPS7A7002 is 0.5 V and its input voltage range is from 1.425 V to 6.5 V. The voltage range of *VDD_IO* is more flexible, so the design of *VDD_IO* will be discussed first, and the *VDD_core_internal* will be discussed later on.

Table 6.1: Table of voltage and current range of two on-board supply power.

	V		I	
	min	max	min	max
VDD_core_internal	0.25 V	1.5 V	0.6 A	3 A
VDD_IO	0.8 V	3.3 V	50 mA	100 mA

6.1.1 VDD_IO

For *VDD_IO*, assume the power dissipation is 100 mW, the dropout of the voltage regulator for VDD_IO is between 100 mW/100 mA = 1 V to 100 mW/50 mA = 2 V, that is, the input voltage of the specific voltage regulator is between 0.8 V + 1 V =

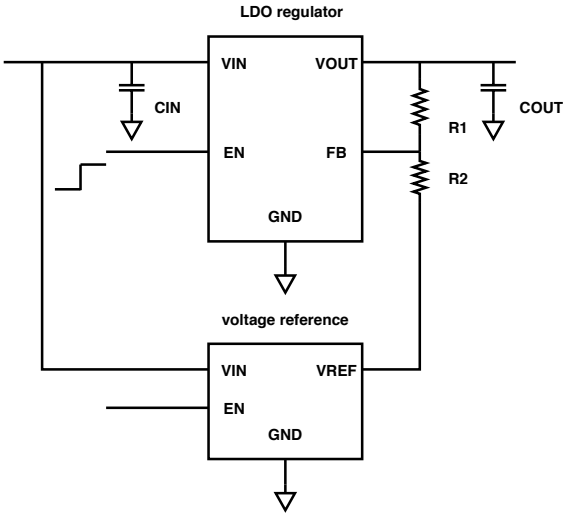


Figure 6.2: Simple schematic of generating small voltage with LDO regulator and voltage reference.

1.8 V to 3.3 V + 2 V = 5.3 V. The design requirement of LDO regulator for *VDD_IO* can be summarized as in Table 6.2. TPS7A7002 fulfils the requirements, so it can be used in the project.

Table 6.2: Table of design requirements of voltage regulator for *VDD_IO*

	V_{in}		V_{out}		I_{load}	
	min	max	min	max	min	max
VDD_IO	$\leq$ 1.8 V	$\geq$ 5.3 V	0.8 V	3.3 V	50 mA	100 mA

Suppose V_{in} = 5 V, V_{out} = 0.8 V, and I_{load} = 100 mA, the power dissipation can be calculated to be 0.42 W according to Equation 6.2, and the IC will heat up 0.42 W · 47(°C/w) = 19.74°C, plus the room temperature 25°C, the temperature is approximately 45°C which is under 125°C.

6.1.2 VDD_core_internal

As described before, finding a LDO voltage regulator to supply voltage less than 0.8 V might be hard, so the design task can be divided into two tasks: finding a linear voltage regulator can generate 0.8 V or above voltage; finding voltage reference can provide reference voltage larger than the reference voltage of the LDO regulator so that the final output of LDO regulator can be 0.25 V small. The output voltage of the voltage regulator should not be bigger than 1.5 V too much for safety reason, with this condition, low power dissipation requirement, and the maximum output current around 1 A less than 3 A, the design requirement of LDO regulator for *VDD_core_internal* is summarized in Table 6.3. The V_{in} of the LDO regulator should be close to the V_{out} to dissipate less power.

Table 6.3: Table of design requirements of LDO regulator for **VDD_core_internal**

	V_{in}		V_{out}		I_{load}	
	min	max	min	max	min	max
VDD_core_internal	$V_{out,min}+ \leq 1\,V$	$V_{out,max}+ \leq 1\,V$	0.8 V	1.5 V	0.6 A	3 A

TPS7A7002's reference voltage is 0.5 V. Therefore, the output of the voltage reference should be higher than 0.5 V. As illustrated in the Equation 6.2, once the V_{REF} is fixed the output voltage of the LDO regulator only depends on the resistors' ratio. As the voltage reference also depends on the resistors' ratio of voltage regulator, the resistors' value should be determined after finding a voltage reference IC.

The next step is choosing a proper voltage reference. As the same input rail provides power to the LDO regulator and voltage reference, and the output voltage of the voltage reference also depends on the resistors' ratio of voltage regulator, this step should also take the output of LDO regulator into consideration. The procedure of choosing the voltage reference is using Equation 6.2, Equation 6.3, and the boundary value of the output voltage of the LDO regulator and the voltage reference to determine the range of the resistors' ratio and the output voltage of the voltage reference. The detailed procedure will be shown under below.

Substitute the $V_{out,min} = 0.8\,V$, $V_{out,min} = 1.5\,V$, and $V_{REF} = 0.5\,V$ into the Equation 6.2, we can get

$$0.8\,V \leq V_{out} = 0.5 \cdot (1 + \frac{R_1}{R_2}) \leq 1.5\,V \tag{6.4}$$

The solution to this inequality equation is

$$0.6 \leq \frac{R_1}{R_2} \leq 2 \tag{6.5}$$

Here, we get the first resistors' ratio range is [0.6, 2].

The final output voltage of the voltage regulator should be at least 0.25 V. The 0.25 V can be served as a boundary value. Substitute the $V_{out,min} = 0.25\,V$ and $V_{FB} = 0.5\,V$ into the Equation 6.3, we have

$$V_{out} = V_{FB} - \frac{R_1 \cdot (V_{REF} - V_{FB})}{R_2} = 0.5\,V - \frac{R_1 \cdot (V_{REF} - 0.5\,V)}{R_2} = 0.25\,V \tag{6.6}$$

The solution to this equation is

$$\frac{R_1}{R_2} = \frac{0.25}{V_{REF} - 0.5} \tag{6.7}$$

Here, we get the second resistors' ratio range. Consider the Equation 6.5 and the Equation 6.7 together, we have

$$0.6 \leq \frac{0.25}{V_{REF} - 0.5} \leq 2 \tag{6.8}$$

The solution to this inequality is

$$0.625\,\mathrm{V} \leq V_{REF} \leq 0.92\,\mathrm{V} \tag{6.9}$$

Here, we get the first requirement of the voltage reference: $0.625\,\mathrm{V} \leq V_{REF} \leq 0.92\,\mathrm{V}$. Usually the voltage reference can provide output larger than 1 V, so the target voltage reference should be around 1 V.

Another important parameter of choosing the voltage reference is the input voltage $V_{in,ref}$. Since the VIN rail of the voltage regulator also provides power to the voltage reference, i.e. $V_{in,ref} = V_{in,reg}$, the $V_{in,ref}$ of the voltage reference should keep close to the V_{out} of the voltage regulator to reduce dissipated power. A suitable V_{in} without dissipating too much power should be around 2 V. Two requirements of choosing the voltage reference are summarized in Table 6.4.

Table 6.4: Table of design requirements of voltage reference for **VDD_core_internal**

	V_{in}	V_{out}	
		min	max
voltage reference	2.0 V	1.2 V	2.0 V

Some voltage reference ICs can fulfil the requirements listed in the Table 6.4, with specific output voltage of the LDO regulator, different reference ICs can be used. Suppose the final V_{out} is smaller than 0.25 V, a possible V_{REF} could be 1.25 V, then LM4140 [23] can be used. LM4140 is a voltage reference IC that can provide 1.25 V reference voltage with a minimum supply voltage of 1.8 V. Substitute $V_{REF} = 1.25\,\mathrm{V}$, $V_{out} \leq 0.25\,\mathrm{V}$ into Equation 6.3, it can be calculated to $R_1/R_2 \geq 0.33$. One requirement of the TPS7A7002 is the $27\,\mathrm{k\Omega} \leq R_2 \leq 33\,\mathrm{k\Omega}$. Pick E96 series resistor $R_2 = 30.1\,\mathrm{k\Omega}$, the closest resistor value in the E96 series is $10.2\,\mathrm{k\Omega}$. The final output voltage is 0.246 V which is smaller than 0.25 V. When R_2 is connected to ground, the generated power is 0.67 V; if the R_2 is connected to the output of voltage reference LM4120, the generated power will be 0.246 V.

When input voltage of the *VDD_IO* and *VDD_core_internal* are close to each other, the two power inputs can be provided by same power rail. To do so, one external power supply will be saved.

6.2 Experimental chip pin design

For the pin design, a 48-pin package is used. There should be four pins for the SPI interface. One of the four SPI interface pins is MISO pins, it should be configured at tri-state pins otherwise it will cause pin conflict. There are 8 pins for controlling function, and 8 pins for debugging function. Two pins are used for providing clock signals to the chip, and 22 pins are used to providing power to the chip, the I/O, and the ground. Since the I/O signals are slow two **VDD_IO** pins lays on the opposite

side of the chip are enough. In order to avoid the ripple in the ground plan, more GND pins are used than **VDD_IO/VDD_core** pins. There are still 4 pins left and it can be used as GNDs and VDDs.

6.3 Voltage level translating

Since the Arduino's GPIO's voltage is 5 V, but the experimental chip's pin voltage is around 1 V, thus voltage translator are needed between the Arduino and the chip to let them communicate with each other. In this thesis project, voltage translator LSF0108 [24] is used. Pull-up resistors are connected to the input and output side of the voltage level translating IC. In order to simplify the layout, resistor-network are used to reduce the number of resistors. Table 6.5 shows the resistors' value of these three voltage translator when the current through each pass resistor is at 15 mA.

Table 6.5: Table of resistors value of resistor network

	V_{refB}	V_{refA}
LS0108	330 Ω	30 Ω

7
Results

This chapter will present results get in the project. Before showing the results, testing methodologies and evaluation setup used in this project will be described first. The results of the project include the testing results of the evaluation setup, utilization result of the VHDL implementation on an FPGA board, and the PCB design schematic.

7.1 Testing methodologies

The testing methodologies used in this project include unit testing and system testing. The unit testing was performed on Arduino implementation and VHDL implementation. The following subsections will describe the unit testing briefly.

7.1.1 Unit testing

The unit testing on Arduino implementation tested three important functions on an Arduino UNO board: SPI communication, CRC generating function, and CRC checking function. To test the SPI communication of the Arduino, a loopback test was performed:

- Connecting the MOSI pin of the Arduino to the MISO pin;
- Calling SPI.transfer( ) function to send a specific byte;
- Reading the return value of the SPI.transfer( ) call and print this value on a serial monitor.

The expected result of this unit test is the returned value printed on the serial monitor is as same as the send value.

Before testing the CRC generating function, given a certain byte value, its CRC checksum should be calculated by a MATLAB script first. Then the following steps are performed:

- Passing the certain byte as a input data of the CRC generating function;
- Printing the CRC checksum on the serial monitor;
- Comparing the generated CRC checksum with the MATLAB script calculated CRC checksum.

The expected result of the CRC generating unit test is the generated CRC checksum is equal to the calculated CRC checksum.

The unit testing of the CRC checking function is similar to the CRC generating unit testing. The difference are the input data passed to the CRC checking function is two-bytes long: one byte data followed by its CRC checksum; and the binary checking result will be printed on the serial monitor. Feeding the input data without transmission error and with transmission error to the CRC checking function, the expected test result is the CRC checking function should be able to detect the transmission error and report it correctly.

The unit testing on the VHDL implementation was performed on a NEXYS 4 FPGA board [25], and the general idea of the unit testing is using switches of the FPGA board to give the input signals of the unit under test (components or blocks) and giving the output of the unit under test to LED lights of the FPGA board. The unit testing of the SPI interface is in similar manner with the test on the Arduino, a loop-back test was performed. The SCK signal of the SPI interface is provided by a clock generator component on VHDL. The MOSI pin was connected to the MISO pin of the SPI interface. The 8-bits *i_byte_tx* was provided by 8 switches of the FPGA board, and the 8-bits *o_byte_rx* was given to 8 LED lights of the SPI interface. The expected result of the unit testing is the *o_byte_rx* LED displays is equal to the input *i_byte_tx* eventually. The other components' or blocks' unit testing are followed the general testing idea mentioned above and they won't be explained in detail here.

All components/blocks passed the unit testing. The system testing of this project was conducted with an Arduino UNO board, a NEXYS 4 FPGA board, and a voltage level translating circuit on a bread board. The next section will describe the system testing.

7.1.2 System testing and evaluation

The initial testing and verifying setup had PC (MATLAB), an Arduino board, and an FPGA board. The Arduino serves as a gateway between the PC and the chip, the testing was intended to start with testing and verifying functions of the Arduino and the FPGA board, and then move to the functions between PC and the Arduino board, finally combine them all. However, designing and implementing functions between the Arduino and the FPGA board, and designing PCB schematic have run out of the time so that functions between the PC and the Arduino have been left out of the project. As a consequence of that, the system testing and evaluation only tested and verified functions between the Arduino and the FPGA board.

The system testing setup of the thesis project includes an NEXYS 4 FPGA board, and Arduino UNO board, and a voltage level translating circuit layed on a bread board. Since the maximum voltage of pmod pins of the NEXYS 4 FPGA board is 3.3 V, and the maximum voltage of GPIO pins of the Arduino board is 5 V, a voltage level translating circuit is needed between the FPGA board and the Arduino. All signals come from the Arduino board should be adapted to 3.3 V. Because 3.3 V can be recognized as voltage high by Arduino UNO, an unidirectional voltage level

translator is enough for this case, signals generated by the FPGA board can be connected to the Arduino directly. In this thesis project, an M74HC4050 [26] is used as a voltage level translator. However, in PCB design, the voltage level translators between the Arduino board and the experimental chip should be bi-directional to obtain more reliable signal transaction. All signals come from the experimental chip should be shift to 5 V first then be passed to the Arduino board. Figure 7.1 shows the simple schematic of the testing setup. The SPI related GPIO pins of the Arduino is given to the voltage level translating circuit and Table 7.1 shows the pins connection between the Arduino UNO and the FPGA. With another Arduino moudle, the SPI related pins might different.

Table 7.1: Table of pins connection between Arduino UNO and NEXYS 4 FPGA.

	Arduino UNO	NEXYS 4 FPGA
MOSI	11	ja[1]
MISO	12	ja[2]
SCK	13	ja[3]
SS	10	ja[4]
SS2	9	ja[4](chip#2)

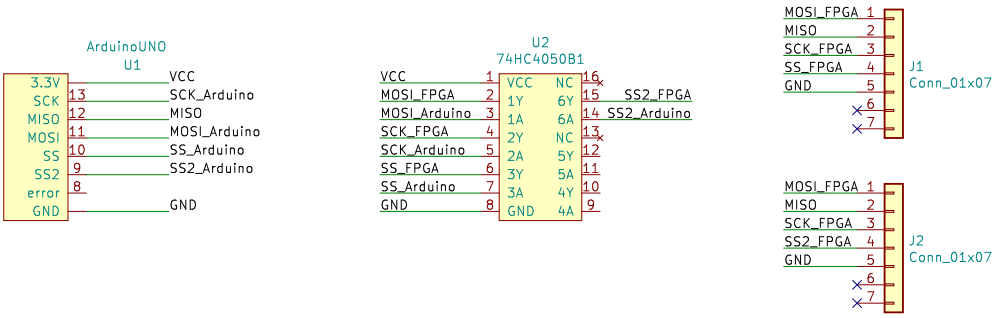


Figure 7.1: Simple schematic of evaluation setup

The Arduino board sends command packets to the FPGA board, and waits for the FPGA board sends packet back. Using the Arduino's serial monitor, the retrieved data can be seen via it. An Arduino program **sketch_master.ino** plays the role as the SPI master, and a VHDL source code **top_test.vhd** serves as the SPI slave. The **top_test** entity contains the **top** component and a ROM component. To test the readDataTest function, the test results can be fixed given by a certain 8-byte long data; or it can be given by a ROM stored different 8-byte long data. The **ROM** can be generated by Xilinx Block Memeory Generator [27], series 8-byte long data can be written into a coefficient (COE) file to initialize the ROM. The output of parameter register is given to LED lights of the FPGA board.

There are two testing scenarios: only one slave device (the experimental chip) is connected to the master device Arduino; two slave devices are connected to the master device. The signle slave test scenario can be used to test command functions, and the two slave scenario are used to test multiple test sites running function. Some test vectors used in the single slave scenario can be used in the two slave scenario. Testing cases used in these two scenarios are described below.

Testing scenario 1: only one slave connected to the master device, Arduino sends command packets to the experimental chip 1, the header of the data packet is 0x81. Executing following test cases in order, observing test results on the serial monitor of Arduino IDE and the LED lights of FPGA:

1. write parameter 0x0A0B to RAM address 0x01.
2. write parameter 0x0C0B to RAM address 0x02.
3. write parameter 0xEC0B to RAM address 0x03.
4. read parameter stored in the RAM address 0x01.
5. read parameter stored in the RAM address 0x02.
6. read parameter stored in the RAM address 0x03.
7. fetch parameter stored in the RAM address 0x01.
8. read test results; suppose the test results is fixed given by 8 byte long value 0x0123456789ABCDEF.
9. read test results stored in a ROM.

After executing the first seven test cases, the LED lights displayed as 0x0A0B, 0x0C0B, and 0xEC0B sequentially, then displayed as 0x0A0B again. Since only readDataParam and readDataTest function will return value to the Arduino, only test case 4-6 and 8-9 will display values on the serial monitor of the Arduino IDE. List 7.1 shows the serial print displayed on the serial monitor after running test case 4. The first returned byte was the header 0x81, then followed by the header's CRC checksum 0xCC; next was the first byte of the parameter 0x0A and its CRC checksum 0xEC. The second byte of the parameter was 0x0B, before received 0x0B, the header 0x81 and the header's CRC checksum came first, finally the CRC checksum of the second byte came. The last four lines of the printed log only displayed the data and the CRC check result. All error check result was '0', which means no error.

Listing 7.1: test results of test case 1 to 4

```
read parameters of Slave#1 begin
data_byte = 81
crc_byte = CC
data_byte = A
crc_byte = EC
data_byte = 81
crc_byte = CC
data_byte = B
crc_byte = 26
0: data = 81; error = 0
1: data = A; error = 0
```

```
2: data = 81; error = 0
3: data = B; error = 0
```

To verify the CRC check function of the Arduino, data input with errors have to be produced on purpose. There are two ways can generate wrong data. The first way is modifying the CRC checksum of the header to let a wrong CRC checksum appended after the header when it sends to the Arduino. The second way is removing the first SPI.transfer( ) call to retrieve junk data and performing the CRC check process directly from the second SPI.transfer( ) call (the new first SPI.transfer( ) call). If we start the CRC checking process from the junk value, all the CRC check results should be '1'. List 7.2 shows the test results with solution two; All CRC check results were '1'.

Listing 7.2: test results of test case 1 to 4 with CRC error

```
read parameters of Slave#1 begin
data_byte = 81
crc_byte = 81
data_byte = CC
crc_byte = A
data_byte = EC
crc_byte = 81
data_byte = CC
crc_byte = B
0: data = 81; error = 1
1: data = CC; error = 1
2: data = EC; error = 1
3: data = CC; error = 1
```

List 7.3 shows the test results of the test case 8. The 8-byte long data 0x0123456789ABCDEF were split into bytes and returned orderly, the even index printed sentences showed the header of the returned packets, and the odd index print sentences showed the returned test results of the chip.

Listing 7.3: test results of test case 8

```
read test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
2: data = 81; error = 0
3: data = 23; error = 0
4: data = 81; error = 0
5: data = 45; error = 0
6: data = 81; error = 0
7: data = 67; error = 0
8: data = 81; error = 0
9: data = 89; error = 0
10: data = 81; error = 0
```

```
11: data = AB; error = 0
12: data = 81; error = 0
13: data = CD; error = 0
14: data = 81; error = 0
15: data = EF; error = 0
```

Test case 9 is similar to test case 8. To make sure the the different data stored in the ROM can be read as test results, readDataTest function should be called in the loop function of the Arduino so that every iteration of the loop a different value will be shifted into the top module of the VHDL implementation. Since the log was long, so it won't be shown fully here (the log contains 16 iterations). List 7.4 shows the first two round of the test result of this test case. The test results showed that when the test data of the chip changed, it can be read back via the readDataTest function.

Listing 7.4: the first two rounds of the test results of test case 9

```
round = 0
read test result of Slave#1 begin
0: data = 81; error = 0
1: data = FE; error = 0
2: data = 81; error = 0
3: data = DC; error = 0
4: data = 81; error = 0
5: data = BA; error = 0
6: data = 81; error = 0
7: data = 98; error = 0
8: data = 81; error = 0
9: data = 76; error = 0
10: data = 81; error = 0
11: data = 54; error = 0
12: data = 81; error = 0
13: data = 32; error = 0
14: data = 81; error = 0
15: data = 10; error = 0
round = 1
read test result of Slave#1 begin
0: data = 81; error = 0
1: data = F; error = 0
2: data = 81; error = 0
3: data = ED; error = 0
4: data = 81; error = 0
5: data = CB; error = 0
6: data = 81; error = 0
7: data = A9; error = 0
8: data = 81; error = 0
9: data = 87; error = 0
```

```
10: data = 81; error = 0
11: data = 65; error = 0
12: data = 81; error = 0
13: data = 43; error = 0
14: data = 81; error = 0
15: data = 21; error = 0
```

Another interesting question is how large data the readDataTest function can process. By now the test results data of the chip are 8-byte long. The data length might increase in the future, it can be also smaller than 8 bytes. Thus, a boundary test should be implemented here. Giving different length fixed data as test results of the chip, varied data length from 1 byte to 10 bytes, and 20 bytes, the boundary test was performed. The test result of the boundary test is shown in List A.1, since 20 byte data has already a large number, data length even large won't be tested in this project. If the design passed 20 bytes, it can also pass testing with data length less than 20 bytes.

Testing scenario 2: two slaves connected to the master device, Arduino sends command packets to two experimental chips. The header of chip 1 is 0x81, and the header of chip 2 is 0x80. The purpose of test cases in the testing scenario 2 is verifying the multiple test sites running function. So, if the returned data of different chip can carry different headers with same commands but with different address or data value the returned data is correct, then it can be proved that the multiple test sites running can work properly. Same test vectors used in testing scenarios 1 can be used here. The following test cases were performed:

1. write parameter 0x0A0B to ram address 0x01 of chip 1
2. write parameter 0x0C0B to ram address 0x01 of chip 2
3. read parameter stored in the ram address 0x01
4. read test results (assume the test results of chip 1 is 0x0123456789ABCDEF, and the test results of chip 2 is 0xFEDCBA9876543210)

The test results of test case 1 and 2 are shown in List 7.5. As it was shown in the list, all data packets come from different chip differs with different headers. Parameter 0x0A0B of chip 1 and parameter 0x0C0CB of chip 2 were returned with no errors. The test results of test case 3 and 4 are shown in List 7.6, test result data of two chips were retrieved with different headers and without errors.

Listing 7.5: test results of test case 1 and 2

```
read parameters of Slave#1 begin
0: data = 81; error = 0
1: data = A; error = 0
2: data = 81; error = 0
3: data = B; error = 0
read parameters of Slave#2 begin
0: data = 80; error = 0
```

```
1: data = C; error = 0
2: data = 80; error = 0
3: data = B; error = 0
```

Listing 7.6: test results of test case 3 and 4

```
read test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
2: data = 81; error = 0
3: data = 23; error = 0
4: data = 81; error = 0
5: data = 45; error = 0
6: data = 81; error = 0
7: data = 67; error = 0
8: data = 81; error = 0
9: data = 89; error = 0
10: data = 81; error = 0
11: data = AB; error = 0
12: data = 81; error = 0
13: data = CD; error = 0
14: data = 81; error = 0
15: data = EF; error = 0
read test result of Slave#2 begin
0: data = 80; error = 0
1: data = FE; error = 0
2: data = 80; error = 0
3: data = DC; error = 0
4: data = 80; error = 0
5: data = BA; error = 0
6: data = 80; error = 0
7: data = 98; error = 0
8: data = 80; error = 0
9: data = 76; error = 0
10: data = 80; error = 0
11: data = 54; error = 0
12: data = 80; error = 0
13: data = 32; error = 0
14: data = 80; error = 0
15: data = 10; error = 0
```

7.2 Utilization result of the VHDL implementation

After synthesizing the VHDL code on Vivado based on the NEXYS 4 FPGA board, an utilization report was generated by Vivado. Appendix B shows the report. Both

LUT and memory utilization are very low.

7.3 PCB design

The Arduino and VHDL implementation took up most of the time so that only a schematic of PCB was created, the layout and manufactured of the PCB have been left out. Figure C.1 shows the schematic of the PCB. The whole PCB schematic was segmented into several parts, the left part is the Arduino component, the middle part is the voltage level translating circuits, the right part are two experimental chips are connected to the Arduino through the voltage level translating circuits, and the left down corner part is the power management part.

7.4 ASIC power and area

The ASIC simlulation of the VHDL implementation has been left outside of this project. It was planned in the beginning of the project and was performed before this summer, but during the summer the whole design was substantially changed, and those changes and updates was time consuming, so it became impossible to redo it due to the time limit of the project.

8

Discussion

This chapter will discuss the results obtained in Chapter 7. Starting with the analysis of the results, limitation and evaluations to the design implementation will follow.

8.1 Results

After running the test cases 1 to 3 in the testing scenarios 1, the LED lights were off. Then the FPGA board displayed sequentially as 0x0A0B, 0X0C0B, and 0XEC0B after executing test case 4 to 6. The LED lights displayed as 0x0A0B again after executing test case 7. The test results suggest the Arduino can communicate with the SPI interface, and write parameters into the RAM of the chip and read parameters stored in the RAM of the chip. Since only when the read control signal of the RAM block is asserted the parameters stored in the RAM can be read from the parameter register of the RAM access block, and the first three write command didn't enable the read control signal, the LED lights were turned off. When reading parameters stored in certain addresses of RAM, parameters were sent to parameter register so the LED displayed as those values. After fetching parameter stored in the RAM address 0x01, value 0x0A0B displayed on the LED lights. Similar to read parameters, the fetched parameters were sent to parameter register so the LED lights displayed as value 0x0A0B which was written into address 0x01 after running the test case 1.

The test results showed in the List 7.1 and List 7.2 suggests that the CRC generating function on the VHDL implementation can work properly, the data packet can be sent with headers, and the CRC checking function is correct. All data bytes are coming as one byte data followed by one byte CRC checksum manner, in order to get one byte useful data four bytes are needed: first byte is the header indicates the source of the data(which chip), the second byte is the header's CRC checksum, and the third byte is the useful data byte, and the fourth byte is data bytes' CRC checksum. When there is transmission error, the Arduino can detect it and report it.

Test results presented in the List 7.3 suggests that the Arduino can retrieve the test results generated by the chip successfully, all eight bytes test results were returned with header. The test results showed in the List A.1 suggests that the Arduino can read back at least 20 bytes long data from the chip.

Test results showed in the List 7.5 and 7.6 suggests that the multiple test sites

running function can work correctly: different chips' data packet with their headers were read back, and no data collision happened during transmission.

All testing results suggests that the main functions of the testing system, namely SPI interface, CRC checking/generating on Arduino, CRC generating on VHDL, accessing RAM of the chip, and retrieving test results of the chip, can work properly. The CRC check function on the VHDL implementation was verified okay in unit testing, but the checking result the error signal cannot be reported back to the Arduino. The reason of this will be explained in the next section.

From the post-synthesize utilization report showed in the Appendix B, it can be seen that both LUT and memory utilization of the VHDL implementation on the NEXYS 4 FPGA is very low. This suggests that the VHDL implementation doesn't use too much resources of the FPGA board, and there is a larger portion of the FPGA resource left for testing experimental chips.

Although the testing results described above suggests the main functions of the testing system work properly, some limitations should be noticed. In testing and evaluation part to the project design, the test results of experimental chips were provided with fixed given value or different value stored in a ROM on the FPGA. Some important features of the system when the testing results are provided by some other DSP or FEC chip might have been hided, which would bias the evaluation of the designed testing system. If there is a DSP/FEC chip connected behind the existing evaluation setup illustrated in section 7.1.2 to provide a real testing results of design chips, some effort need to put to make the DSP/FEC chip upload data to the other parts of the evaluation setup, such as solving synchronization problems between the DSP/FEC chip and the NEXYS 4 FPGA board, and creating another communication interface between the DSP/FEC chip and the FPGA board, or an interface between DSP/FEC chip and Arduino.

8.2 Evaluation of the design implementation

8.2.1 CRC calculating implementation

The CRC generator and CRC checker on the VHDL implementation is a serial implementation which consists of shift registers and on every clock it can only process one-bit data. The advantage of the serial implementation is the low hardware complexity, but it can not get the CRC calculation results with a fast speed [28]. Since there is no high speed requirement of the testing system, the serial implementation is acceptable. However, the serial CRC implementation may not fulfil the specific high speed requirements for testing other high-performance design chips. Parallel CRC implementation can overcome the low speed drawback.

8.2.2 CRC check error reported back problem

One problem in the design is Arduino cannot catch CRC check error reported on the VHDL implementation. Since clock signals provided by the experimental chip are much faster than the SCK generated by Arduino, the time Arduino needs to complete one time digital read/write is much longer than one clock tick of the chip. That means before the Arduino can finish reading the error reported by the chip, the error signal might has already changed. One possible solution to solve this problem is four-phase handshaking scheme. In the handshaking scheme, the error signal must be resynchronized on a hardware two-stage flip flop before they are sent to the Arduino. Clock signals and power supply must be provided to the hardware flip-flop which decrease the hardware efficiency of the design. Therefore, the handshaking scheme was not used to solve this problem in the project.

Since the initial reason to add an error detection procedure to the system is to avoid writing wrong parameters into the chip and trigger wrong testing experiments, a compensation to report error back to the Arduino is using the readDataParam function read parameters written into the RAM back to the Arduino and comparing them with parameters send to the chip. The comparing can be done at the Arduino side, and the Arduino report the checking results to PC. Due to the time limit of the project, this comparing and checking function has left out. With the compensation solution wrong test experiments triggered by wrong parameters can be avoided, however, transmission error happens in sending data stages of command fetchDataParam cannot be reported to the Arduino; the address of the fetchDataParam may be wrong. As a consequence of this, there is still possibility that a wrong experiment will be launched. This problem may be solved by changing the computer card from Arduino to CPLD or other FPGA board, since synchronization in VHDL implementation is easier than synchronization between VHDL implementation and an MCU. At that time, four-phase handshaking scheme could be used to fixing the cannot catch CRC check error problem.

If the Arduino can catch the CRC check error successfully, error handling could be done at the Arduino or move above to PC. When there is a CRC check error, the Arduino can resend the command without issuing the PC, or the Arduino can upload this error to PC to let PC resend the command. It depends on the user how to deal with this error information.

8.2.3 Performance regarding the RAM

In the project, an SRAM is used for storing parameters for testing the design chips. The RAM can store at most 256 16-bit parameters. With less knowledge about the specific design chip, the performance of the RAM hasn't been take into consideration. At the planning stage of the project, one desired bonus feature is storing waveform in the RAM [7]. However, it turns out achieve the main features has been used up all the time, and this bonus feature has been left. It would be interesting to store waveform in the RAM and let design chips to arbitrarily fetch symbols stored in the RAM [29]. At this point, how large waveform the RAM can store and how fast the

design chip can fetch the symbols becomes attractive topics of the project.

8.2.4 Data waste in the designed system

From the testing results, it can be seen that there are data wasted in the testing system, every useful byte retrieved from the chip will cost other three bytes: one byte header, the header's CRC checksum, and CRC checksum of the useful byte. The reason of the data redundancy include hardware limitation, and the data transaction reliability.

The Arduino's SPI library is an 8-bit oriented SPI implementations, the Arduino can only exchange one byte long data with the chip. Increasing the data process length on Arduino or on VHDL may require more complex implementation to split larger data into bytes before data transaction. So, the existing data waste is kept in the system.

There may be rare cases that tri-state buffers of the MISO line of the two chips doesn't work properly so that the two chips may upload data on the MISO line simultaneously. To improve the data transmission reliability, each byte is going to send via SPI interface should carry its header and header's CRC checksum. In reality, the probability may be so low so the design can be relaxed a bit. A header and the header's CRC checksum will only add before the first byte. Take the eight bytes test results sending as an example, if only add header and header's CRC checksum before the first byte, it only needs $1 + 2 + 8 \cdot 2 = 19$ bytes exchanging on the SPI interface and saved 14 bytes.

8.2.5 SPI interface

The SPI interface increases the data transmission speed and use less hardware resource. However, since the SPI communication relies on the chip select line (SSand the SPI clock (SCK), once the Arduino board cannot provide stable SS or SCK, the SPI interface cannot work. For example, in the testing stage of the project, there were several times that the SCK line was not stable by jump wire's connection so that the MISO line and the MOSI line signals got random delay. As a result of this, wrong data was retrieved back to the Arduino.

8.2.6 PCB design

Low-dropout linear voltage regulator was used in this project to generating power on-board. The solution is low-cost, simple, and low noise. However, linear voltage regulators always waste more power compare to the switching voltage regulator, and with low power efficiency. Combined with switching voltage regulator and the LDO voltage regulator may be a good solution [30]: use a switching voltage regulator followed by a LDO voltage regulator; the switching voltage regulator can provide an efficiency but with noise output, and the LDO voltage regulator can give a noise-free

output. On the other hand, the output of the voltage regulator is fixed after the input voltage and the resistors' ratio are fixed. This solution is not flexible, and the adjustment of the voltage output range is hard.

9
Conclusion

This chapter summarizes the results and relates them to the aim and objectives that were set at the beginning of the project. The possible future development will be discussed at the end.

9.1 Objective fulfillment

The project aims to generalize the existing testing approach and make the experimental chip evaluation setup more effective and efficient. In the project, the original interface between the Arduino and the experimental chip has been updated to an SPI interface. The original input and output pins for data transaction have been replaced by SPI MISO and MOSI pins, fewer pins are used so that a smaller chip package can be used and the price of the whole testing system gets decreased. Besides, since SPI communication can achieve faster data transmission, a large trunk of data can be exchanged during a short period of time. Two experimental chips are connected to the Arduino via the SPI interface so that these two chips can be performed with different experiments and finally speed up the research process. From SPI interface aspect, the testing infrastructure has become more efficient and more effective.

A CRC check/generating function is added to the data transmission process. Data transaction reliability will be improved by the error-detect scheme. The wrong testing parameters won't trigger further testing process, and the wrong testing results can be detected so that they won't be used for analysis. Thus, the whole testing process becomes more efficient and more reliable.

A RAM access block that can be used to store different kinds of testing parameters has been added. As a storage of the parameters is independent with the type of the experimental chips, the RAM access block can be generalized to other chips. When there are some particular parameters changed to launch a different experiment, only these parameters need to sending to the chip, other parameters can be fetched from the RAM. To do so, possible data transaction errors may be reduced, and the workload of the whole system get decreased. The RAM access block makes the whole design more general and efficient.

A new PCB was designed, more general pin names have been used for the Arduino and the chip, so it can be generalized to test other design chips. Due to the COVID-

19 pandemic and the time limit of the project, only a schematic of the PCB was created, more accurate analysis of the PCB design has been left. The PCB hasn't been laid out nor manufactured yet so that testing work of the hardware part of the testing infrastructure was never completed. A good advantage of the project design is that the whole design is divided into several design blocks (layers), so that optimizations or alternatives can be targeted on single blocks. In this way, the testing infrastructure can be easily generalized and expanded.

Because of the time limit of the project, there was no time to implement function in MATLAB to control the Arduino to communicate with the chips, and there was no time to explore other powerful computer cards that can be used in this project.

9.2 Future work

This section will discuss about some possible development in the future. The future work includes PCB design, storing waveform in the RAM, exploring other powerful computer cards, and other development. The following subsection will describe each topic in detail.

9.2.1 PCB design

The voltage output of the LDO voltage regulator is fixed after the VIN and the ratio of $R1/R2$ are determined. It is not flexible and a digital potentiometer should replace the fixed value $R1/R2$ to provide adjustable and programmable voltage output. Usually digital potentiometers are controlled by MCU via I2C/SPI/parallel communication. Since the SPI related pins of the Arduino have been used, and the parallel communication will use more hardware resources, the digital potentiometer used in the project should be I2C controlled. To control the digital potentiometer IC via the I2C, Arduino software development regarding controlling the IC via I2C need to be done. The PCB design can start from choosing proper digital potentiometer. The next step is layout the PCB according to the designed schematic, and printing the new PCB. The testing of the PCB should focus on testing the power generation part of the PCB since this part should be safe enough to avoid damage experimental chips. In order to avoid the generated power damage experimental chips, safely control the voltage range generated on the PCB is necessary. One possible solution is to using analog read pins of the Arduino to read the voltage value back and output the power to the chip only when the voltage falls into the safe range.

9.2.2 Storing waveform in the RAM

As mentioned in section 8.2.3, exploring the performance of the RAM would be interesting to the project. How large the RAM could be, how fast the design chips can fetch the symbols from the RAM, and the relationship between the area and the access speed of the RAM could be attractive points. When fast access data stored in

the RAM, some errors might happen, so the access error rate is also an interesting point.

9.2.3 Exploring other powerful computer card

Arduino board provides the low-cost feature to the testing infrastructure but it has some drawbacks to limit the efficiency of the testing infrastructure. So, it is attractive to explore other powerful computer cards in the future. This subsection will shortly discuss some drawbacks of using Arduino in the project and then give some possible alternatives to the Arduino regarding its drawbacks.

As explained in section 8.2.2, Arduino board cannot catch the CRC check error reported from the chip. One of the reasons is that synchronization between an FPGA/ASIC design and a MCU on the MCU side is hard. However, synchronization on FPGA/ASIC side can be simply achieved by using some flip-flops. On the other hand, the Arduino sets the SPI communication configurations via SPI.settings() (a software call), but the specific configuration cannot pass to the VHDL implementation directly. As a consequence of this, the SPI configuration is fixed on the VHDL implementation. For example, in this project, the SPI mode is mode 3, both Arduino and VHDL implemenation should keep the same mode. If the Arduino side changes to other mode, the SPI implementation on VHDL should also change. If the specific configuration can be passed to the VHDL implementation directly, like CPOL and CPHA, the SPI implementation on VHDL can be adapted to an interface with CPOL and CPHA port so that the SPI mode can be changed easily. When both master device and slave device are implemented on VHDL, this can be achieved easily. From the synchronization aspect and the directly set configuration aspect, using CPLD or FPGA as a computer card might be a good choice. Also, if the whole testing system can be implemented on SoC, that would be good to enhance the testing performance significantly [31].

If the compensation method mentioned in section 8.2.2 can fulfil the users' expectation, some other powerful computer cards such as Arduino Due and Raspberry Pi can be used in the project. In the Arduino board used in the present FEC chip evaluation setup is Arduino Mega2560 which provides 5 V GPIO signals and these signals must be adapted to 0.8 V signals via voltage translators. Voltage shifter/-translator ICs that can shift signals from 5 V to 0.8 V are seldom. From this aspect, a computer card that can provide 3.3 V GPIO signals or even lower voltage signals are attractive. The Arduino Mega2560 which is used in the existing FEC chip evaluation and the Arduino board used in the developing stage of the project are using 8-bit microcontroller processor which can not provide fast speed read/write operation and the largest number the Arduino can represent is not large enough. Thus, seeking another powerful computer cards that have powerful calculation ability is interesting. The Arduino Due can provide 3.3 V GPIO signals and has a 32-bit Arm core that fulfils the above two requirements so it may be a possible alternative. If the voltage range of GPIO pins is not important in developing then Raspberry Pi could be a alternative to the computer card.

9.2.4 Other development

As explained in section 8.2, the serial CRC calculation may not meet high performance of the experimental chips. It would be interesting to explore the parallel CRC implementation on VHDL, and comparing the serial CRC implementation with the parallel CRC implementation on ASIC timing, power, area aspect regarding the different input data length and input data values.

The data waste problem can be improved with only add header and header's checksum to the first transaction byte as described in section 8.2. It is worth to reduce the data waste with this method to lower the cost of the design further more.

The testing methodologies used in the project are unit testing and system testing. With more mature design, performance testing is necessary, such as stress test, endurance testing. Those testing can be performed to evaluate the reliability and robustness of the testing system.

Bibliography

[1] Charles E. Stroud, "An Overview of BIST," in A Designer's Guide to Built-In Self-Test, Dordrecht, Netherlands: KLUWER ACADEMIC PUBLISHERS, 2002, ch. 1. [Online]. Available: https://doi-org.proxy.lib.chalmers.se/10.1007/b117480, Accessed on: Apr 03, 2020.

[2] Wang, Laung-Terng, et al. "Preface," in System-On-Chip Test Architectures: Nanometer Design for Testability, Burlington, MA, USA: Elsevier Science Technology, 2008, Preface. [Online]. Available: https://ebookcentral.proquest.com/lib/chalmers/detail.action?docID=330094, Accessed on: Apr 03, 2020.

[3] Erik Börjesson, Lars Svensson, and Per Larsson-Edefors. *Private communication*. Gothenburg, 2020.

[4] Kevin Cushon, Per Larsson-Edefors, Peter Andrekson, "A High-Throughput Low-Power Soft Bit-Flipping LDPC Decoder in 28 nm FD-SOI," in ESSCIRC, 2018.

[5] C Fougstedt et al. "preparation" in Implementation and Evaluation of Soft-Assisted Product Decoder in a 22-nm FD-SOI Technology.

[6] Lars Svensson, *Project description*. Gothenburg, 2020.

[7] Victor Åberg, Erik Börjesson, Sinan Ding, *Project spec collection meeting*. Gothenburg, 2020.

[8] M. Poorani and R. Kurunjimalar, "Design implementation of UART and SPI in single FGPA," 2016 10th International Conference on Intelligent Systems and Control (ISCO), Coimbatore, 2016, pp. 1-5, doi: 10.1109/ISCO.2016.7726983.

[9] Warren W. Gay, Book title: Raspberry Pi Hardware Reference. Berkeley, CA, USA: Apress, 2014. [Online]. Available: https://doi-org.proxy.lib.chalmers.se/10.1007/978-1-4842-0799-4 , Accessed on: Apr 02.

[10] Volnei A. Pedroni, "VHDL Design of Serial Communications Circuits," in Circuit Design with VHDL , MIT Press, 2004, pp.375-422.

[11] Pong P. Chu, FPGA Prototyping by VHDL Examples: Xilinx MicroBlaze MCS SoC, 2th ed., Hoboken, NJ, USA: John Wiley Sons, 2017, ch. 15.1. [Online]. Available: `https://chalmers.skillport.eu/skillportfe/assetSummaryPage.action?assetid=RW$7216:_ss_book:125700#summary/BOOKS/RW$7216:_ss_book:125700`, Accessed on: Sep 28, 2020.

[12] Microchip: ATmega48/88/168 - Complete Datasheet, `http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48_88_168_\protect\T1\textbraceleftmegaAVR`-Data-Sheet-40002074.pdf

[13] Microchip: ATmega640/1280/1281/2560/2561 - Complete Datasheet, http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega640-1280-1281-2560-2561-Datasheet-DS40002211A.pdf

[14] Arduino: SPI library, https://www.arduino.cc/en/reference/SPI

[15] CYCLIC REDUNDANCY CHECK CIRCUIT AND COMMUNICATION SYSTEMI HAVING THE SAME FOR MULT-CHANNEL COMMUNICATION, by Jae-Young Kwak. (2006, Mar 15,)*. US 7,890,835 B2 [Online]. Available: https://patents.google.com/patent/US7890835, Accessed on: Apr 03.

[16] Pong P. Chu, RTL HARDWARE DESIGN USING VHDL Coding for Efficiency, Portability, and Scalability, Hoboken, NJ, USA: John Wiley Sons, 2006, [Online]. Available: https://ieeexplore.ieee.org/servlet/opac?bknumber=5237648, Accessed on: Sep 28, 2020.

[17] R. Ginosar, "Metastability and Synchronizers: A Tutorial," in IEEE Design Test of Computers, vol. 28, no. 5, pp. 23-35, Sept.-Oct. 2011, doi: 10.1109/MDT.2011.113.

[18] C. Vonk, "Math Talk," 2015. [Online]. Available: https://coertvonk.com/hw/logic/connecting-fpga-and-arduino-using-spi-13067, Accessed on: Oct 05, 2020.

[19] DP & SM, "FPGA HDL Basics," 2018. [Online]. Available: https://www.arduino.cc/en/Tutorial/Foundations/VidorHDL, Accessed on: Oct 05, 2020.

[20] Victor Åberg, Erik Börjesson, Sinan Ding. *E-mail communication about PCB design.* Gothenburg, 2020.

[21] Maxim Integrated Products,Inc., "BUCK REGULATOR GENERATES ULTRA-LOW OUTPUT VOLTAGE," 2004. [Online]. Available: https://www.maximintegrated.com/en/design/technical-documents/app-notes/3/3200.html, Accessed on: Jan 21, 2021.

[22] TEXAS Instrument, "TPS7A7002 Very Low Input, Very Low Dropout 3-A Regulator With Enable," 2017. [Online]. Available: https://www.ti.com/lit/ds/symlink/tps7a7002.pdf?ts=1606913998903&ref_url=https%253A%252F%252Fwww.google.com%252F, Accessed on: Dec 03, 2020.

[23] TEXAS Instrument, "LM4140 High Precision Low Noise Low Dropout Voltage Reference," 2016. [Online]. Available: https://www.ti.com/lit/ds/symlink/lm4140.pdf?ts=1606911747147&ref_url=https%253A%252F%252Fwww.google.com%252F, Accessed on: Dec 03, 2020.

[24] TEXAS Instrument, "LSF010x 1/2/8 Channel Auto-Bidirectional Multi-Voltage Level Translator for Open-Drain and Push-Pull Applications," 2020. [Online]. Available: https://www.ti.com/lit/ds/symlink/lsf0108.pdf?ts=1611252454194&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FLSF0108, Accessed on: Jan 21, 2021.

[25] Digilent Inc., "Nexys 4 DDR Reference Manual," 2016. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual, Accessed on: Jan 21, 2021.

[26] STMicroelectronics, "M74HC4050 HEX BUFFER/CONVERTER," 2001. [Online]. Available: https://www.st.com/resource/en/datasheet/cd00002560.pdf, Accessed on: Jan 21, 2021.

[27] Xilinx, "Block Memory Generator v8.4 LogiCORE IP Product Guide," 2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf, Accessed on: Jan 21, 2021.

[28] Jie Li, Shanshan Liu, Pedro Reviriego, Liyi Xiao, Fabrizio Lombardi. (2020). Scheme for periodical concurrent fault detection in parallel CRC circuits. IET Computers Digital Techniques, 14(2), 80–85. https://doi.org/10.1049/iet-cdt.2018.5183

[29] Erik Börjesson, Lars Svensson, and Per Larsson-Edefors. *Questions in final presentation*. Gothenburg, 2020.

[30] John Teel, "How to Pick the Right Voltage Regulator(s) for Your Design," 2019. [Online]. Available: https://predictabledesigns.com/how-to-pick-the-right-voltage-regulators-for-your-design/, Accessed on: Jan 21, 2021.

[31] K. Lee, T. Hsieh, C. Chang, Y. Hong and W. Huang, "On-Chip SOC Test Platform Design Based on IEEE 1500 Standard," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 7, pp. 1134-1139, July 2010, doi: 10.1109/TVLSI.2009.2019978.

[32] Frisk, D. (2016) A Chalmers University of Technology Master's thesis template for LaTeX. Unpublished.

A

Appendix 1

Listing A.1: test results of different length test results

```
read 1 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
read 2 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
2: data = 81; error = 0
3: data = 23; error = 0
read 3 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
2: data = 81; error = 0
3: data = 23; error = 0
4: data = 81; error = 0
5: data = 45; error = 0
read 4 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
2: data = 81; error = 0
3: data = 23; error = 0
4: data = 81; error = 0
5: data = 45; error = 0
6: data = 81; error = 0
7: data = 67; error = 0
read 5 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
2: data = 81; error = 0
3: data = 23; error = 0
4: data = 81; error = 0
5: data = 45; error = 0
6: data = 81; error = 0
7: data = 67; error = 0
8: data = 81; error = 0
9: data = 89; error = 0
read 6 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
2: data = 81; error = 0
3: data = 23; error = 0
4: data = 81; error = 0
5: data = 45; error = 0
6: data = 81; error = 0
7: data = 67; error = 0
8: data = 81; error = 0
9: data = 89; error = 0
10: data = 81; error = 0
11: data = AB; error = 0
read 7 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
2: data = 81; error = 0
3: data = 23; error = 0
```

```
4: data = 81; error = 0
5: data = 45; error = 0
6: data = 81; error = 0
7: data = 67; error = 0
8: data = 81; error = 0
9: data = 89; error = 0
10: data = 81; error = 0
11: data = AB; error = 0
12: data = 81; error = 0
13: data = CD; error = 0
read 8 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = 1; error = 0
2: data = 81; error = 0
3: data = 23; error = 0
4: data = 81; error = 0
5: data = 45; error = 0
6: data = 81; error = 0
7: data = 67; error = 0
8: data = 81; error = 0
9: data = 89; error = 0
10: data = 81; error = 0
11: data = AB; error = 0
12: data = 81; error = 0
13: data = CD; error = 0
14: data = 81; error = 0
15: data = EF; error = 0
read 9 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = EF; error = 0
2: data = 81; error = 0
3: data = 1; error = 0
4: data = 81; error = 0
5: data = 23; error = 0
6: data = 81; error = 0
7: data = 45; error = 0
8: data = 81; error = 0
9: data = 67; error = 0
10: data = 81; error = 0
11: data = 89; error = 0
12: data = 81; error = 0
13: data = AB; error = 0
14: data = 81; error = 0
15: data = CD; error = 0
16: data = 81; error = 0
17: data = EF; error = 0
read 10 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = CD; error = 0
2: data = 81; error = 0
3: data = EF; error = 0
4: data = 81; error = 0
5: data = 1; error = 0
6: data = 81; error = 0
7: data = 23; error = 0
8: data = 81; error = 0
9: data = 45; error = 0
10: data = 81; error = 0
11: data = 67; error = 0
12: data = 81; error = 0
13: data = 89; error = 0
14: data = 81; error = 0
15: data = AB; error = 0
16: data = 81; error = 0
17: data = CD; error = 0
18: data = 81; error = 0
19: data = EF; error = 0
read 20 byte test result of Slave#1 begin
0: data = 81; error = 0
1: data = CD; error = 0
```

```
2: data = 81; error = 0
3: data = EF; error = 0
4: data = 81; error = 0
5: data = 1; error = 0
6: data = 81; error = 0
7: data = 23; error = 0
8: data = 81; error = 0
9: data = 45; error = 0
10: data = 81; error = 0
11: data = 67; error = 0
12: data = 81; error = 0
13: data = 89; error = 0
14: data = 81; error = 0
15: data = AB; error = 0
16: data = 81; error = 0
17: data = CD; error = 0
18: data = 81; error = 0
19: data = EF; error = 0
20: data = 81; error = 0
21: data = 1D; error = 0
22: data = 81; error = 0
23: data = EF; error = 0
24: data = 81; error = 0
25: data = 1; error = 0
26: data = 81; error = 0
27: data = 23; error = 0
28: data = 81; error = 0
29: data = 45; error = 0
30: data = 81; error = 0
31: data = 67; error = 0
32: data = 81; error = 0
33: data = 89; error = 0
34: data = 81; error = 0
35: data = AB; error = 0
36: data = 81; error = 0
37: data = CD; error = 0
38: data = 81; error = 0
39: data = E0; error = 0
```

B

Appendix 2


```
------------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2019.1 (win64) Build 2552052 Fri May 24 14:49:42 MDT 2019
| Date         : Sun Nov 22 18:20:20 2020
| Host         : DESKTOP-EUC18TO running 64-bit major release  (build 9200)
| Command      : report_utilization -file top_utilization_synth.rpt -pb top_utilization_synth.pb
| Design       : top
| Device       : 7a100tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs*                |  758 |     0 |     63400 |  1.20 |
|   LUT as Logic             |  730 |     0 |     63400 |  1.15 |
|   LUT as Memory            |   28 |     0 |     19000 |  0.15 |
|     LUT as Distributed RAM |   28 |     0 |           |       |
|     LUT as Shift Register  |    0 |     0 |           |       |
| Slice Registers            |  601 |     0 |    126800 |  0.47 |
|   Register as Flip Flop    |  584 |     0 |    126800 |  0.46 |
|   Register as Latch        |   17 |     0 |    126800 |  0.01 |
| F7 Muxes                   |    4 |     0 |     31700 |  0.01 |
| F8 Muxes                   |    2 |     0 |     15850 |  0.01 |
+----------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typically
lower. Run opt_design after synthesis, if not already completed, for a more realistic count.


1.1 Summary of Registers by Type
--------------------------------

+-------+--------------+-------------+--------------+
| Total | Clock Enable | Synchronous | Asynchronous |
+-------+--------------+-------------+--------------+
| 0     |            _ |           - |            - |
| 0     |            _ |           - |          Set |
| 0     |            _ |           - |        Reset |
```

```
| 0     |            _ |         Set |            - |
| 0     |            _ |       Reset |            - |
| 0     |          Yes |           - |            - |
| 19    |          Yes |           - |          Set |
| 526   |          Yes |           - |        Reset |
| 0     |          Yes |         Set |            - |
| 56    |          Yes |       Reset |            - |
+-------+--------------+-------------+--------------+
```

2. Memory

```
+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    |  0.5 |     0 |       135 |  0.37 |
|   RAMB36/FIFO*    |    0 |     0 |       135 |  0.00 |
|   RAMB18          |    1 |     0 |       270 |  0.37 |
|     RAMB18E1 only |    1 |       |           |       |
+-------------------+------+-------+-----------+-------+
```

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

3. DSP

```
+-----------+------+-------+-----------+-------+
| Site Type | Used | Fixed | Available | Util% |
+-----------+------+-------+-----------+-------+
| DSPs      |    0 |     0 |       240 |  0.00 |
+-----------+------+-------+-----------+-------+
```

4. IO and GT Specific

```
+-----------------------------+------+-------+-----------+-------+
|          Site Type          | Used | Fixed | Available | Util% |
+-----------------------------+------+-------+-----------+-------+
| Bonded IOB                  |   90 |     0 |       210 | 42.86 |
| Bonded IPADs                |    0 |     0 |         2 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |         6 |  0.00 |
| PHASER_REF                  |    0 |     0 |         6 |  0.00 |
| OUT_FIFO                    |    0 |     0 |        24 |  0.00 |
| IN_FIFO                     |    0 |     0 |        24 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |         6 |  0.00 |
| IBUFDS                      |    0 |     0 |       202 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        24 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        24 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       300 |  0.00 |
| ILOGIC                      |    0 |     0 |       210 |  0.00 |
| OLOGIC                      |    0 |     0 |       210 |  0.00 |
+-----------------------------+------+-------+-----------+-------+
```

5. Clocking

```
+------------+------+-------+-----------+-------+
|  Site Type | Used | Fixed | Available | Util% |
+------------+------+-------+-----------+-------+
| BUFGCTRL   |    1 |     0 |        32 |  3.13 |
| BUFIO      |    0 |     0 |        24 |  0.00 |
| MMCME2_ADV |    0 |     0 |         6 |  0.00 |
| PLLE2_ADV  |    0 |     0 |         6 |  0.00 |
| BUFMRCE    |    0 |     0 |        12 |  0.00 |
| BUFHCE     |    0 |     0 |        96 |  0.00 |
```

BUFR	0	0	24	0.00

6. Specific Feature

Site Type	Used	Fixed	Available	Util%
BSCANE2	0	0	4	0.00
CAPTUREE2	0	0	1	0.00
DNA_PORT	0	0	1	0.00
EFUSE_USR	0	0	1	0.00
FRAME_ECCE2	0	0	1	0.00
ICAPE2	0	0	2	0.00
PCIE_2_1	0	0	1	0.00
STARTUPE2	0	0	1	0.00
XADC	0	0	1	0.00

7. Primitives

Ref Name	Used	Functional Category
FDCE	509	Flop & Latch
LUT2	421	LUT
CARRY4	124	CarryLogic
LUT4	120	LUT
LUT6	103	LUT
LUT5	81	LUT
LUT3	78	LUT
LUT1	74	LUT
IBUF	70	IO
FDRE	56	Flop & Latch
RAMD32	42	Distributed Memory
OBUF	19	IO
FDPE	19	Flop & Latch
LDCE	17	Flop & Latch
RAMS32	14	Distributed Memory
MUXF7	4	MuxFx
MUXF8	2	MuxFx
RAMB18E1	1	Block Memory
OBUFT	1	IO
BUFG	1	Clock

8. Black Boxes

Ref Name	Used

9. Instantiated Netlists

Ref Name	Used

C
Appendix 3

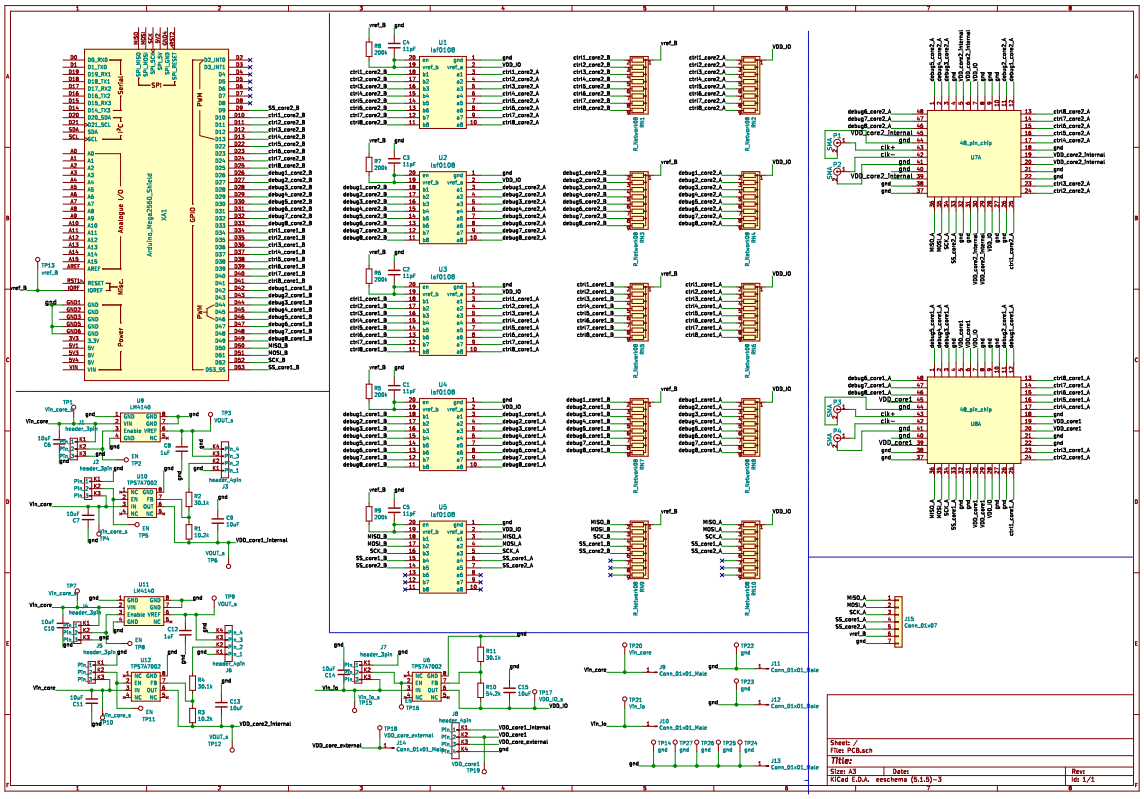


Figure C.1: PCB schematic