RISC-V Based Microcontroller using Arrow Deca Max 10 FPGA development Board

Interim Report
2^{nd} Draft

Submitted by

Kaung Min Khant (PN1046592)

May 5, 2023

Contents

Chapter 1

Introduction

1.1 Project Background

This research explores the design and implementations for a micro-controller based on open-source RISC-V RV32E Instruction Set Architecture(ISA). This research concerns with detail designs of the CPU and peripherals, along with detail documentation for each module. This project emphasizes on modularity with extensibilty on desired hardware and instruction sets. This research will be published on GitHub repository.

On the web, a lot of projects and processors based on the popular RISC-V ISA can be found. However, only little information about detail implementations about the ISA is provided. This poses a challenge for students and researchers who want to explore implementation about the ISA.

In order to provide simple yet detailed implementation of the ISA, this research is introduced. Many open-source projects utilized Verilog HDL. This research will use VHDL, which is an industrial standard.

1.2 Literature Review

Processor design starts by choosing an Instruction Set Architecture (ISA). There are two options, commercial and open source ISAs. Most of the commercial ISAs are closed source and proprietary. Its licensing is expensive. Hence, it was decided to build a processor using an open source ISA. The advantage of this methodology would be to have full control over every aspect of the design, and allowing modularity for the hardware. RISC-V is an emerging open source ISA. Release of “Rocket Chip” [1] triggered development of processors based on RISC-V. A soft core based on chisel [2] called “ZSCALE” [1] is released based on RISC-V RV32IM extension. The verilog version of “ZSCALE” is released as “VSCALE” [3]. A microcontroller based on RISC-V ISA named “mRISC-V” is described in [4]. [5]. These designs, however, lack the detail documentation of the actual implementation for a beginning researcher to understand. Thus this issue is addressed here with a hardware implementation simple enough to understand and detailed documentations.

1.3 Aims and Objectives

This research is proposed for construction of RISC-V Hardware using VHDL and implemented on FPGA.

The objectives of this research are as follow.

1. To build a complete microcontroller, yet simple enough to follow, based on RISC-V RV32E ISA using VHDL.

2. To offer detail documentation of the implementation of the hardware

3. To offer hardware extension for future developement

4. To offer assemble for firmware creation

5. To offer implementation support for hardware designers to build their own hardwares based on the required projects

1.4 Expected Outcomes

The expected outcomes of the research are as follows

1. The CPU must work based on the specified ISA, in this case, RICS-V RV32E ISA
2. The CPU must be tested
3. The overall micro-controller must have standard peripherals and they must be tested
4. The documentation of all of the hardware implements must be provided
5. The research will be published on GitHub Repo

Chapter 2

Microcontroller CPU Design and Implementation

Chapter 3

Peripheral Designs and Implementations

3.1 UART - Universal Asynchronous Receiver Transmitter

This universal asynchronous receiver transmitter (UART) module is written for Field Programmable Logic Array (FPGA) implementation with VHDL. This module can be duplicated and controlled using external logic or controllers. The module can be initialized with desired clock and baudrate. Only one baudrate is used for both transmitter and receiver. Little-endianess is used for both transmission and reception. These can be modified if required. This module has been tested using loop-back test method, described below.

3.1.1 Block Diagram and I/O

3.1.1.1 Overview

Insert overview block diagram with inputs and outputs

Input/Output Table

Input Name	Function	Output Name	Function
clk	main clock input	**uart_tx_line**	UART RX line
tx_start	transmitter start bit	**tx_busy**	transmitter busy bit
data_in	32 bit data input	**rx_busy**	receiver busy bit
uart_tx_line	UART TX line	**tx_done**	transmitter done bit
rx_mode_32	32 bit Receiver Mode	**rx_done**	receiver busy bit
		rx_done	receiver busy bit
		data_out	8 bit receiver data out
		data_out_32	32 bit receiver data out

3.1.2 Usage

The UART module consists of transmitter and transmitter. The transmitter and receiver can work simultaneously without interfering with each other. The module has 32 bit data input for transmission with 8 bit or 32 bit data output for reception.

The transmission can be initiated by setting **tx_start** bit. The designer should clear **tx_start** bit after setting it to prevent it from triggering multiple times. Data will be transmitted via **uart_tx_line**. During transmission, **tx_busy** bit will be set to indicate transmission and will be cleared after transmission. **tx_done** bit will be pulsed high once the transmission is completed. The designer can detect the rising edge, falling edge or the pulse level to determine transmission completion.

The receiving is initiated automatically upon receiving the start bit on **uart_rx_line**. **rx_busy** is set during receiving and cleared upon completion. **rx_done** is pulsed once upon completion. The designer must use falling edge of the pulse to ensure the data has been stored within the receiver. If 32 bit receiver mode is selected, **rx_done** is pulsed only after receiving 4 bytes.

The received data will be available on **data_out** and 32 bit **data_out_32** lines based on **rx_mode_32** input. If **rx_mode_32** input is set, the receiver will expect four 8 bit data packets to complete 32 bit. If **rx_mode_32** input is cleared, the receiver will only expect a single 8 bit data packets. (*Note: Protection feature should be implemented to provide the failure of receiving four bytes to the system. Currently the system will always set* rx_busy *if it fails to complete the reception*)

3.1.2.1 Initialization

The module has two generic variables which needs to be set according to the desired clock frequency and baudrate.
input_clock_frequency : system input clock in Hz *Example: 10e6*
baudrate : desired baud rate for transmission and reception *Example: 9600*

3.1.3 UART Architecture

3.1.3.1 Data Frame

The data frame for the module is chosen for the standard data frame. It is composed as figure ?. Data frame error is checked for every receiving data frame to ensure data integrity. (*Parity bits can be implement as optional feature*)

Insert data frame photo here.

3.1.3.2 Implementation

Implemetation can be found on the GitHub listed in A.1. The transmitter and receiver are implemented as separated modules which are then initialized and combined with top-level VHDL module. The implementation can be accessed through the following GitHub repo: insert github repo here.

3.1.3.2.1 Transmitter Implementation The transmitter receives 32 bit data, which needs to be seperated, since standard UART only transmits 8 bit of data at a time. (*This behavior can be modified using necessary controls and implementation*). It is accomplished by simple data truncation to respective data groups.

The module is set to work every system clock cycle. However, in order to match the desired baudrate, counters are utilized. Once the transmitter start signal (in this case: **send**) is set, the necessary flags are raised and transmission is started on the next clock cycle. In order to transmit all 4 bytes of data, a counter `word` is utilized to keep track. Each segment is modified to fit into correct data frame with a single stop bit and start bit. According to the system clock cycle and desired baudrate, the count value for full bit can be obtained using

$$full\_bit\_count = \frac{System\_clock}{baudrate}$$

According to UART specifications, each data bit need to be transmitted at the middle of the full bit count to ensure data integrity. `index` is used to keep track of bit position. The data is transmitted on `tx` signal. Once all the bits are transmitted and transmission is completed. **tx_done** output is pulsed high as soon as the transmission is completed and transmission line is kept high until **send** signal is asserted.

3.1.3.2.2 Receiver Implementation The receiver will receive data on **rx** signal line at a given baudrate. The calculation is identical as transmitter baudrate calculation. Full bit counting is used the same way as the transmitter. At every rising edge of system clock cycle, the **rx** line is monitored, and once the start bit is properly detected, the process of receiving starts and necessary flags are raised. The start bit on the **rx** line must be held low for half bit count in order to be read as valid data, as per UART specification. `index` is used to keep track of bit addressing.

Currently, two modes of receiving is implemented: 8 bit mode and 32 bit mode. This mode can be changed using **mode32** signal. If **mode32** signal is set, 32 bit receiving mode is selected.

In 8 bit receiver mode, the data on the **rx** signal at half bit count to ensure data integrity. **rx_done** is pulsed high at the end of receiving 8 bit data.

`index` variable is used to keep track of bit addressing. In 32 bit receiver mode, `word` variable is used to keep track of 8 bit word addressing. **rx_done** is pulsed high at the end of receiving 4 bytes. All flags are reset after each completion.

In both modes, data packet integrity check is performed on the received data. In 8 bit mode, the defected data will be ignored and will assert **frame_error** signal and wait for resend. In 32 bit mode, the defected word will be ignored and will assert **frame_error** signal and wait for resend.

Chapter 4

Testings

Chapter 5

Conclusion

To be filled when the research can be concluded.

Appendices

Appendix A

Some Appendix

A.1 Source Code

Source Codes can be found on the GitHub Page: Dummy Link

A.2 Requirements

A.2.1 Hardware and Software Requirements

Hardware and software requirements are as follow

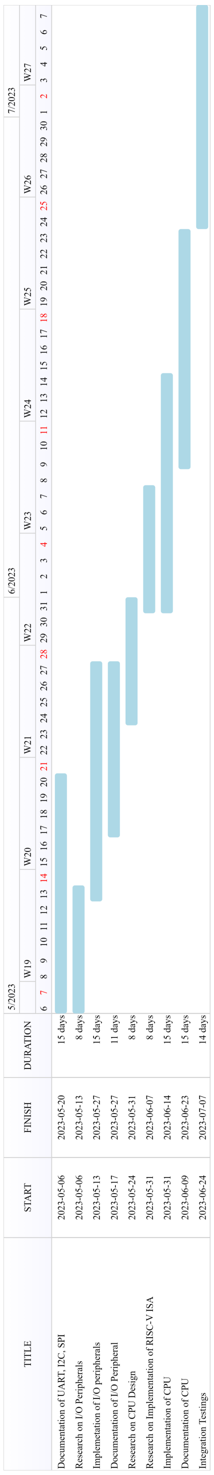
1. Arrow Deca FPGA Development Board
2. Intel Quartus Prime Lite Software
3. ModelSim Lite simultation Software

A.2.2 Skill Requirements

The following skills are required.

1. VHDL Programming
2. Computer Organization and Designe for RISC-V ISA
3. Micro-controller Peripheral Design
4. Micro-controller Communication Design
5. Assembly for RISC-V ISA

A.3 Timeline



TITLE	START	FINISH	DURATION
Documentation of UART, I2C, SPI	2023-05-06	2023-05-20	15 days
Research on I/O Peripherals	2023-05-06	2023-05-13	8 days
Implemetation of I/O peripherals	2023-05-13	2023-05-27	15 days
Documentation of I/O Peripheral	2023-05-17	2023-05-27	11 days
Research on CPU Design	2023-05-24	2023-05-31	8 days
Research on Implementation of RISC-V ISA	2023-05-31	2023-06-07	8 days
Implementation of CPU	2023-05-31	2023-06-14	15 days
Documentation of CPU	2023-06-09	2023-06-23	15 days
Integration Testings	2023-06-24	2023-07-07	14 days

Bibliography

[1] Y. Lee, A. Ou, and A. Magyar, "Z-scale: Tiny 32-bit risc-v systems with updates to the rocket chip generator," tech. rep., The International House, Berkely, California, June 2015.

[2] "Chisel." https://www.chisel-lang.org/, 2015.

[3] "Verilog version of z-scale, vscale." https://github.com/LGTMCU/vscale/tree/master/src, 2016.

[4] C. Duran *et al.*, "A 32-bit risc-v axi4-lite bus based microcontroller with 10-bit sar adc," *VII Latin American Symposium on Circuits and Systems (LASCAS)*, 2016.

[5] S. Budi, P. Gupta, K. Varghese, and A. Bharadwaj, "A risc-v isa compatible processor ip for soc," Master's thesis, Indian Institute of Science (IISc), year = 2020, month = jul, address =.