

ÖZYEĞİN UNİVERSİTY

FACULTY OF ENGINEERING

**<EE 300>**

**SUMMER PRACTICE REPORT**

**<KUTAY BULUN>**

**<S021370>**

**INTERNSHIP COMPANY & DEPARTMENT:**

**<MATRİKS BİLGİ DAĞITIM HİZMETLERİ A.Ş./** **Information Technology >**

**<24.09.2023>**

#### SUMMER PRACTICE REPORT

|  |  |
| --- | --- |
| **STUDENT** | |
| **Name** | **Kutay Bulun** |
| **Internship Start Date** | **21.08.2023** |
| **Internship Completion Date** | **18.09.2023** |
| **Total Working Days** | **20** |
| **COMPANY** | |
| **Name** | **Matriks Bilgi Dağıtım Hizmetleri A.Ş.** |
| **Department** | **Information Technology** |
| **Address** | **İTÜ Ayazağa Yerleşkesi Koru Yolu ARI 3 Binası K:10 No:1001 34396 Maslak / İSTANBUL** |
| **SUPERVISOR** | |
| **Name** | **Mücahit Durmuş** |
| **Title** | **Electric and Electronics Engineer** |
| **Department** | **HFT – Digital Design and Verification** |
| **Phone** | **0212 354 54 54** |
| **E-Mail** | **mucahit.durmus@matriksdata.com** |
| **Signature** |  |

**DAILY WORK SUMMARY**

|  |  |  |
| --- | --- | --- |
| **DAY** | **DATE** | **WORK DESCRIPTION** |
| **1** |  |  |
| **2** |  |  |
| **3** |  |  |
| **4** |  |  |
| **5** |  |  |
| **6** |  |  |
| **7** |  |  |
| **8** |  |  |
| **9** |  |  |
| **10** |  |  |

**Student’s Name: Supervisor’s Name:**

**Student’s Signature: Supervisor’s Signature-Stamp:**

**DAILY WORK SUMMARY**

|  |  |  |
| --- | --- | --- |
| **DAY** | **DATE** | **WORK DESCRIPTION** |
| **11** |  |  |
| **12** |  |  |
| **13** |  |  |
| **14** |  |  |
| **15** |  |  |
| **16** |  |  |
| **17** |  |  |
| **18** |  |  |
| **19** |  |  |
| **20** |  |  |

**Student’s Name: Supervisor’s Name:**

**Student’s Signature: Supervisor’s Signature:**

# Abstract

I conducted my internship at MATRİKS BİLGİ DAĞITIM HİZMETLERİ A.Ş. My job was to create a protocol parser that is designed to work at high frequency and do the verification for this design using UVM methodology. The department of the company I did my internship was HFT which stands for high frequency trading. In high frequency trading the crucial element is to minimize the latency so that trading can execute at very high speed, and in terms of trading what is aimed to be optimized is the transfer of the data coming from the stock exchange in a fashioned way so that it is ready to use at the receiver-end, to do this we do register transfer level coding to be loaded onto an FPGA. I learned how to this design so that it works at line rate and learned how to verify if the design works in an industry standard way.

# Introduction

During my internship at MATRİKS BİLGİ DAĞITIM HİZMETLERİ A.Ş., I was presented with a challenging task that lay at the heart of high-frequency trading (HFT) technology. The primary problem was the design and implementation of a protocol parser capable of processing data originating from BISTECH, transmitted at 10G line rate through Ethernet frames. These Ethernet frames contained a complex amalgamation of headers and payloads, each adhering to distinct protocols layered atop one another. My mission was to decode this data stream, unraveling it into its constituent fields, corresponding to the Ethernet, IP, UDP, MoldUDP64, and ITCH layers, respectively.

Addressing this challenge demanded the utilization of a multi-faceted toolkit. To ensure that the protocol parser operated with the required speed and efficiency, I made use of Register Transfer Level (RTL) design, harnessing the capabilities of the Xilinx ISE tool. Moreover, to guarantee the parser's functionality and reliability, I learned Universal Verification Methodology (UVM). This method enabled me to devise a comprehensive verification strategy, making the design not only robust but also readily testable by fellow engineers.

This report chronicles my endeavors, experiences, and achievements during this internship, shedding light on the intricacies of developing cutting-edge technology solutions for the high-frequency trading industry. It explores the problem statement, the tools and techniques employed, and the insights gained along the way. Through this report, I aim to provide an overview of my internship experience, illustrating its significance in the context of modern financial technology.

# Company Description

**In your own words**, describe briefly the company and the department you worked in 1-2 pages.

Company description should include:

* The company’s sector and products
* Any research conducted in the company
* Very brief history of the company and your department (foundation date, very major changes), and your role in this department during your internship.
* Company statistics (size, organization, etc.)

If you are going to use any material that does not belong to you (e.g., company description taken from a website, company organization chart from company documents, a product list from company website), then do not put them here; instead put them into **appendix**.

# <Design and Verification of ITCH Protocol Parser >

## Problem Statement

During my internship in the High-Frequency Trading (HFT) department, I dealt with a critical issue inherent to high-frequency trading systems: minimizing data processing latency. Specifically, I focused on processing incoming data from BISTECH, delivered via Ethernet frames at a rapid 10G rate, and parsing this data into its relevant fields. This task involved designing a protocol parser for ITCH messages from BISTECH. One of the main challenges lies in receiving and processing the data with zero latency. In the HFT industry, such designs are implemented using hardware description languages, ultimately transformed into FPGA circuits for maximum speed and resource efficiency. This process introduced several constraints:

1. Clock Cycle Precision: Ensuring that no clock cycles were lost to enable high-frequency operation with minimal latency.
2. Data Tracking: Developing a system to precisely track the portion of data being processed within each clock cycle, considering that each cycle handles only a 64-bit segment of the Ethernet package.
3. Correct Data Mapping: Guaranteeing that the received 64-bit data was accurately mapped to the correct fields.
4. Continuous Processing: Designing a system capable of steady and uninterrupted data processing since data from BISTECH flows continuously while the stock market operates.

These constraints led to additional complexities that I will discuss in the detailed explanation section.

Another aspect of the problem involved verifying the functionality of the design using the Universal Verification Methodology (UVM), which required writing verification code comprehensible to other engineers. While I managed to create verification code for a portion of the final design during my internship, it encapsulated the essence of UVM.

It's worth noting that these problems were not new to the company, as they had an existing operational model of the protocol parser. However, my internship primarily revolved around learning and developing these solutions from the ground up, as the company had not shared their existing model with me so that it would not block my learning process.

## Tools and Techniques Used

The successful completion of the protocol parser design project relied on a set of essential tools and techniques, each chosen for its specific advantages and compatibility with FPGA and hardware-based clock designs.

Programming Languages:

* *Verilog:* I utilized the Verilog hardware description language (HDL) for Register Transfer Level (RTL) design. Verilog is the industry standard for FPGA and hardware design. It enables the description of circuit behavior at a register and transistor level, facilitating synthesis to transform code into a physical circuit on an FPGA. The advantages of Verilog lie in its widespread use and robust support within FPGA design environments.
* *System Verilog:* To implement the Universal Verification Methodology (UVM) for verification purposes, I employed System Verilog. This object-oriented extension of Verilog offers improved modularity and reusability, making it well-suited for designing verification environments. System Verilog's advantages include its support for advanced verification techniques and seamless integration with Verilog, allowing for a cohesive hardware and verification design.

Software:

* *Xilinx ISE:* For synthesizing a portion of the design and checking Verilog syntax, Xilinx ISE proved indispensable. This tool, while not the only option for synthesis, offered an accessible educational version. However, some alternative software packages exist for synthesis, which may be more feature-rich but often come at a price.
* *VS Code:* To expedite code development, I opted for Visual Studio Code (VS Code). This choice primarily stemmed from personal preference and familiarity with the environment. Alternative code editors are available, but VS Code's versatility met my needs effectively.
* *EDA Playground:* I used EDA Playground for running verification code written in System Verilog. This online tool offered the advantage of supporting System Verilog, which was not natively supported by Xilinx ISE. While other simulation tools exist, EDA Playground provided a convenient, web-based solution for my verification needs.

Hardware:

* The project primarily focused on FPGA-based hardware design, but an FPGA was not available for actual implementation since I was working from home.

Techniques:

* *Pipelining:* To ensure the design operated at line rate with high frequency and minimal clock cycle loss, I employed pipelining. This technique divided the processing into stages, allowing data to flow continuously through the design without interruptions.
* *Parallel Processing:* To efficiently handle incoming data, I utilized parallel processing. This technique involved simultaneously processing data at various parts of the design, ensuring accurate and continuous data mapping.
* *Object-Oriented Programming (OOP):* For verification purposes, I applied object-oriented programming principles. This approach enhanced code accessibility and reusability, enabling an efficient and effective verification environment that could be readily adapted and extended.

In summary, the tools and techniques chosen for the project were carefully selected to address the specific requirements of FPGA-based hardware design and verification. Each tool and technique played a crucial role in achieving the project's goals while aligning with industry standards and best practices.

## Detailed Explanation

To design a protocol parser for ITCH messages, first we should understand how those messages are received, that is n number of ITCH messages which will be used after it is parsed come within an ethernet frame inside the payload of the ethernet package, which is sent by BISTECH. An ethernet frame consists of its headers and payloads. Inside the payload of an ethernet package there is IP frame with its headers and payloads, UDP or TCP frame with its headers and payloads, MoldUDP64 headers and as its payload ITCH messages exactly in that given order. See Figure 1 for the ethernet frame, the data part is its payload that contains all other layers with their headers and payloads. Also see Figure 2 for the rest of the received package as the payload of an ethernet frame, here payload of MoldUDP is ITCH messages.

A diagram of a computer network

Description automatically generated

Figure 1: Ethernet Frame Format

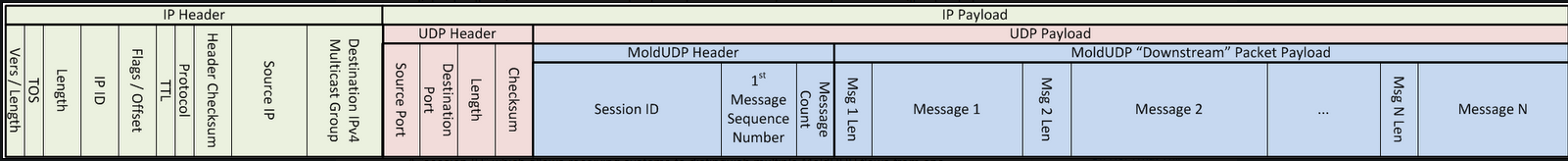


Figure : Downstream MoldUDP packet format

Now that we know the format of a received package, we can start to write the protocol parser in Verilog that will map the fields of these frames to registers correctly. By a design choice I created the design module by module so that it is easy to read and understand, one could also implement all the design inside only one module, which would be hard to understand. See Figure 3 for all the modules in the design that I will explain one by one.

A screenshot of a computer

Description automatically generated

Figure : All the modules in the design

The first step is to design a packet receiver for the data that is coming from BISTECH at line rate of 10G via an ethernet frame. We receive the data 64 bit by 64 bit at each clock cycle until a complete ethernet package is received, then we continue to do this until BISTECH stop sending us ethernet packages, that is when the stock market stops for the day. This is essentially a state machine to implement in Verilog that transitions through some states depending on some control inputs.

In the Verilog code this module is called packetRx. Standard procedure of Verilog convention I follow is, first declaring the inputs and outputs of the module, then declaring the internal registers of the module and making sure they that update their contents at the posedge of the clock each cycle in always@(posedge clk) block of the code, then finally implement the behavior of the module with its default values in always@\* block of the code. This is the same for every other module I implemented so I will not mention this again for the sake of simplicity. First, I implemented a counter to act as a fanout counter to ensure clock cycle precision by keeping track of clock cycles and making sure submodules will start executing depending on the clock cycle, which is parallel processing at its core. Each clock cycle that we receive valid data from BISTECH, I increment the counter by one to achieve a part of the correct data mapping by excluding the sending of corrupted data to submodules. Then, I created a state machine that has two states, initial one and steady one. We start to process data beginning from the initial state if we receive the start of frame signal and data valid signal and then immediately go to steady state which will continue running until this ethernet frame ends. At the steady state, if data is valid, we continue processing and if it is the end of the frame, we only send the valid bits of the 64bit data to submodules and go to initial state to receive the new frame, otherwise if it is not end of the frame we continue running on the steady state. This state machine ensures the continuous processing of the received data. See Figure 4, 5 and 6 for packetRx module, after these Figures I will not add the complete screen shot of the modules since most of it is standard conventions that are similar, and they are too long to capture in a screen shot.

A screen shot of a computer

Description automatically generated

Figure : Input and Output Declaration

A computer screen with white text

Description automatically generated

Figure : Internal Variables and Registers

A computer screen with many colorful text

Description automatically generated

Figure : Behavioral Block, the State Machine

The second step is to design the submodules until the part keeping track of the 64-bit data becomes tricky and complex, which is until ITCH messages. It is complex because we receive n number of ITCH messages and we cannot depend on just the fanout counter coming from packetRx anymore, we need to keep track of which part of the 64 bits are valid somehow. I will talk about this later. But before that, I wrote the codes for modules that will parse the ethernet, IP, UDP, and MoldUDP64 in that given order exactly. All of them execute their job which is to write the correct part of 64-bit data we send from packetRx to relevant register fields of their headers depending on the fanout counter they receive from packetRx. See Figure 7 and 8 for ethernet decoder module and see Figure 1 for how the sizes of registers are chosen to represent the received data correctly that is: a size of a register is equal to number of bytes in that particular field, for instance we have 48-bit dMac register for 6-byte destination MAC address. This is the same for every other field register, registers that contain parts of the ethernet package, in the design. One can easily see the order of these register from Figure 1 and 2, and easily find the correct size of these registers with quick research for further confirmation. For every other decoder I mentioned above, these modules do the same job.

A screen shot of a computer program

Description automatically generated

Figure : I/O's and Registers of Ethernet Decoder

A screen shot of a computer program

Description automatically generated

Figure : Behavioral Block, the Data Mapping

In terms of parallelism, see the behavioral parts of these modules and see how they can process the data at the same clock cycle if they need to in Figure 9, 10, and 11.

A screen shot of a computer program

Description automatically generated

Figure : Data Mapping for IP decoder

A computer screen shot of white text

Description automatically generated

Figure : Data Mapping for UDP decoder

A computer code on a black background

Description automatically generated

Figure : Data Mapping for MoldUDP64 decoder

The third and the hardest part of the design is parsing the ITCH protocol messages that are within the payload of MoldUDP64. There can be variable number of messages and we don’t know exactly how to map the 64-bit data that comes each clock cycle to the field registers, as opposed to preceding part of the design where I could do the math easily. To solve this problem, I came up with an algorithm that writes the correct bits of the 64-bits data to the relevant field register starting from the tth bit of the input data indicated by a tracker, that is where we left of writing the last clock cycle and continues to write until that register has the correct data. Algorithm is like this:

Say that there is register X with size n bits, and there is a 64 bit tracker register that indicates the starting point of writing called t, and 64-bit data is called dataIn. “<< k” is shifting left by k bits and “>> k” is shifting right by k bits.

1. If writing to X is not finished last cycle finish writing. (X = X + dataIn << n-1-t)
2. If writing to X starts this cycle and finishes. (64-t >= n), then:
   1. write to X. (X = dataIn >> t)
   2. set your valid signal to 1.
   3. update the tracker. (t = t + n)
   4. call the algorithm for the next field register.
3. If writing to X does not finish this cycle, then:
   1. Write the part you can do in this cycle. (X = dataIn >> t)
   2. update the tracker. (t = t + n)
4. Go to the next clock cycle.

I wrote the code for every field register in that particular ITCH message following this algorithm and then, optimized it by concatenating field registers so that we write into 64-bit field every clock cycle which allows us to skip some steps of the algorithm by dividing the process into two pieces that is: a best-case scenario and any other scenario. Best-case scenario is that the first writing process we do in that message parser module starts from the first bit where t = 0, and since we write 64 bits each cycle, we do not need to update the tracker at all. Any other scenario is where t is not zero, therefore we need it in the writing process. In all the cases at the final steps of the design we need to set the correct value for tracker as output and set a signal to indicate that this module has done its job and we are ready for the next message. I did this by setting these outputs just one clock cycle before the message module actually ends so that we can achieve correct synchronization between message parser modules, which is the job of the messageParserTop and messageTypeDecoder modules that I will explain as the next step.

Before that I must indicate that I created modules for add order messages, order executed messages, order delete message, order book state message, tick size table entry message and order book directory message following the algorithm I explained above and depending on the specs of these messages that you can find from BISTECH’s documentations, see appendix for that. They are initially indicators of the size of the field registers I created and the correct order to parse them. See the following figures for parsing tick size table entry messages since it is the smallest message to show for the sake of simplicity.

A screen shot of a computer program

Description automatically generated

Figure : I/O and Registers

A screen shot of a computer program

Description automatically generated

Figure : Working of Registers and Default Values



Figure : Reset and Implementation of the Algorithm

A computer screen shot of a program

Description automatically generated

Figure : Implementation of the Algorithm Part 2

The fourth part of the design is writing the modules itchMessageTypeDecoder and messageParserTop to ensure message parser modules will communicate and execute correctly so that we can have continuous stable processing, and ability to make these modules work in parallel to each other depending on their end signals and received 64-bit data each clock cycle.

First, I created the message type decoder to indicate which message parser module to start depending on what the 64-bit data indicates as the message type. Message types are ASCII encoded 8-bit data fields that you can find from ITCH protocol specifications, see appendix for that. What message type decoder does is very similar to the algorithm above except that we have an initial case where the clock cycle is eight, which is the start of the very first ITCH message, and best case that decoding ends this clock cycle and other case that decoding ends in the second clock cycle. Depending on these cases we assign tracker output to correct value, and which message to start to correct value by decoding message type using the ASCII encoding. I sent the out signals as a wire if the process ended in one clock cycle and as a register out if the process ended in two clock cycle to ensure data integrity and synchronization. See the Figures 16 and 17 for the decoding and out signals of the itchMessageTypeDecoder module.

Secondly, I created the messageParserTop module to connect and synchronize message modules and message type decoder module. This module instantiates these modules as a top module and connects their inputs and outputs correctly, especially when to start executing the module and the tracker to start which bit to execute from. See Figure 18 for example instantiation and Figure 19 for control of starting the message type decoder module and the correct tracker assignment depending on which module ended previously.

A computer screen with many white and blue text

Description automatically generated

Figure : Decoding Part 1

A computer screen shot of a code

Description automatically generated

Figure . Decoding Part 2

A screenshot of a computer program

Description automatically generated

Figure : Instantiation of two Modules

A screen shot of a computer

Description automatically generated

Figure : Control of I/O Signals

The last part of the design before verification is connection all the modules before message modules and the message top module that encapsulates message modules at a top module called top. This part is a design convention for the ease of readability which just instantiates all the modules. See figures 20 and 21.

A screen shot of a computer program

Description automatically generated

Figure : Top Module Part 1

A screen shot of a computer code

Description automatically generated

Figure : Top Module Part 2

Now that the design has ended, it is time for verification of the design. Any digital design is practically useless before it is verified, which is simulating the design under various test conditions to ensure it works. There is a concept called Universal Verification Method (UVM) which is an industry standard way of verifying digital designs so that any test can be applied by a verification engineer, which knows the concepts of UVM, to any testbench following this methodology in a stable, comprehensible, and reusable environment. To verify my design, I learned this methodology and wrote a test bench following these standards. Unfortunately, twenty working days were not enough for creating a test bench for the complete design, but I was able to create a working and complete test bench until the message modules of the design, and now I will explain that part.

A UVM testbench makes use of object-oriented programming aspects of System Verilog by creating component classes and test classes that are reusable, clear, and possibly encapsulated. See Figure 22 for visualization of a UVM testbench.

A screenshot of a computer

Description automatically generated

Figure : UVM testbench

I implemented all these classes in System Verilog for my test bench except the subscriber which was not needed for my design. A typical class in UVM consists of a constructor and phases called build, connect, and run. Constructor, build phase and connect phase are standard where you first construct the class by calling the super class, which is a part of the library of UVM that takes of constructing, then in the build phase you also call from super to build the component, then at the connect phase you call again from super to connect the components. The run phase is where you usually write the code that will be executed once the tests run on the testbench. See Figure 23 for a standard way of writing these phases and the constructor except the run phase.

A screen shot of a computer program

Description automatically generated

Figure : Constructor, Build phase and Connect phase

The interface I wrote takes care of communication to design under test by driving inputs to design under test (DUT) and driving outputs from DUT to interface by clocking each related field so that we drive the correct input or get the correct output from DUT.

A screen shot of a computer

Description automatically generated

Figure : Interface

Then I created a sequence item, which is essentially all the inputs or outputs in the interface that we will drive as a test to DUT. Then by initializing these sequence items in the sequence class, I created the test sequences. The base sequence sets the reset signal to 1 and randomizes all other fields to reset the DUT once this sequence is driver via the driver class and the test sequence creates nine different randomized sequence items while keeping the reset signal 0 this time to examine responses of DUT in nine consecutive clock cycle since we are verifying the design until message modules. See Figures 25 and 26 for these tasks in the sequence class.

A computer screen shot of text

Description automatically generated

Figure : Base Sequence Task

A computer code on a black background

Description automatically generated

Figure : Test Sequence Task

Then I created the test class that will run these sequences once it is called. At the run phase of the test class we raise objection to indicate we are running the tests and create the test sequences and drive them, and finally drop the objection when we are done. See Figure 27.

A computer screen shot of code

Description automatically generated

Figure : Run phase of Test Class

Then there is the sequencer class that takes care of sequencing these sequences as its name indicates and the driver class that drives these tests to DUT via the interface I created. At the run phase of the driver class, we create a sequence item and call the task drive to drive the inputs of the design via the interface. See Figure 28. The agent class encapsulates the driver, sequencer and monitor classes so that they are reusable, and the env class encapsulates the agent and the scoreboard class so that whole environment is reusable for various other tests.

A screen shot of a computer program

Description automatically generated

Figure : Run phase and the Driver task of Driver class

Now that we have driven the tests to DUT, it’s time to examine the outputs of the DUT and see if they pass or fail. To correctly get the outputs from the DUT via the interface we create a class called monitor to store and see the inputs and outputs of DUT in a sequence item. Then in scoreboard class we create the conditions to see if DUT fails or passes the tests we are running. In my test I check for depending on the clock cycle we are at whether the contents of the field registers in the design corresponds to correct part of the data being driven to design at that clock cycle. I wrote these conditions for every field until the message modules that display “field passed” if the design runs correctly and “field failed” if there is a mistake. See Figure 29 for the run phase of monitor and Figure 30 for a part of these scoreboard pass of fail conditions.

A screen shot of a computer program

Description automatically generated

Figure : Run phase of Monitor Class

A screenshot of a computer program

Description automatically generated

Figure : A part of Pass or Fail Conditions in Scoreboard Class

Finally, in testbench class that includes all the classes I implemented, we instantiate the interface, the clock, and the DUT just like the top module in the design, then run the test class. I ran the verification in EDA Playground to see the result which I will mention in the next section. Verification of the design ends here since twenty working days were not enough to implement the rest of the verification. Complete design and verification I implemented can be found on my github page I created for this internship, see the third appendix element for that.

## Results

In this section, clearly present the results of your project. In particular, explain the following in detail:

* Was your project completed successfully? If so, show us that your solution indeed is successful. If not, tell us the problems that you encountered, and possible solutions to those problems that could have been applied.
* If your project had quantitatively-measurable goals (performance figures, correctness metric, completing 9 out of 10 features, etc.), then show your results. If your project had qualitative metrics (nice-looking, usable, etc.), then include comments from your supervisor and co-workers, as well as screenshots.

# Conclusions

Conclusions must combine your academic background and your internship experience.

* Explain how your education helped you successfully complete your internship. In particular, explain which topic in which course was used for what purpose in your project.
* Compare your in-class education and assignments with your project assignment during your internship. To what extent they were similar? To what extent they were different?
* Explain how your internship can help your education. What new concepts have you learned that can be used during your classwork?
* Explain which skills and qualifications you have acquired during your internship, and how they may affect your career in the future. Did this internship change your career goals? What were the main influential parts?
* What are your personal views about the company, people, and the work environment?

# Appendix

Include relevant material such as catalogues, product specifications, papers

Product specifications:

* Borsa İstanbul A.Ş. (n.d.). *ITCH Protocol Specification*. Borsaistanbul. <https://borsaistanbul.com/files/bistech-itch-protocol-specification.pdf>
* Borsa İstanbul A.Ş. (n.d.). *MoldUDP64 Protocol Specification*. Borsaistanbul. <https://borsaistanbul.com/files/moldudp64-protocol-specification9890234EF41CE5A6B1C8A4A3.pdf>
* https://github.com/kutaybulun/itch\_protocol\_parser/tree/main

# References

Each information, figure, table, etc. that does not belong to you (has been found online, taken from some other document, etc.) **must** be referenced, or you risk being penalized due to plagiarism.

Figures:

* *Figure 1: (n.d.). Ethernet Frame Format.* Javatpoint.<https://www.javatpoint.com/ethernet-frame-format>
* *Figure 2:* (n.d.). *Dispatches From The Trading Floor - MoldUDP*. Fragmentationneeded. <https://www.fragmentationneeded.net/2012/01/dispatches-from-trading-floor-moldudp.html>
* *Figure 22:* Admin. (n.d.). *UVM Testbench Top*. ChipVerify. Retrieved September 27, 2023, from https://www.chipverify.com/uvm/uvm-testbench-top