Design and Verification of a Simplified Ethernet Network Interface Card (NIC)

This project implements a Layer-2 Ethernet MAC with a digital PHY abstraction and a simple host interface for academic study and simulation.

Objective

This project aims to achieve the following objectives.

1. Understand the structure and mechanism of communication using the internet.
2. Learn about various communication protocols like udp, and work upon bit-level-networking.
3. Implement Simple-As-Possible ethernet card using verilog on FPGA.

Introduction

Network Interface card is a driver that helps in receiving and transmitting data from the internet to computer.

Data is transported in the form of analog signals through air or through port. This signal is then processed to digital by a NIC chip (It is also known as LAN adapter/WAN adapter). This signal is then transported to various parts of a computer to actually display the output as expected by a user.

First, we will go on with the learning phase where we will cover the basics required and then we will get on to the planning phase where we describe how the simplified NIC is going to work and write the required programs. Then we will get on to the testing and execution phase, where we either write testbench or check manually the working of our programs.

How Exactly Data is Transferred ?

To understand exactly how data is being transferred we need to have some pre-requisite knowledge on the following topics. So we will have basic understanding on these topics and get back to the project.

☐ Combinational and Sequential Circuits. (taught in course)

☐ VERILOG (synthesizable, non-synthesizable code, timing analysis, blocking, non-blocking assignments, FSMs, etc.) (taught in course)

☐ FPGA architecture.

☐ Computer architecture.

☐ Networking Protocols.

☐ Physical Layer Interfacing.

LEARN PHASE :

If you are not sure of the first two topics then please refer to the following links to learn about them. These two topics are very important for the development of Network Interface Cards.

☐ Combinational and Sequential Circuits. (taught in course) : https://youtube.com/playlist?list=PLwjK_iyK4LLBC_so3odA64E2MLgIRKafl&si=o-JGiCIEqELKF-8H

☐ VERILOG (synthesizable, non-synthesizable code, timing analysis, blocking, non-blocking assignments, FSMs, etc.) (taught in course) : https://www.chipverify.com/verilog/verilog-tutorial

FPGA ARCHITECTURE

In the real world, software is always flexible but hardware is not. You can easily change software but it is always hard to make changes in hardware. Imagine if hardware is also flexible. There comes FPGA : Field Programmable Gate Array. It is a hardware that can be programmed to do tasks and upgrade the system by changing the program as and when required.

You can get to know more about FPGA through this link : https://youtube.com/playlist?list=PLXHMvqUANAFOviU0J8HSp0E91lLJInzX1&si=gnqNAan7Jhnq7waX

Or you can refer to the notes here : https://drive.google.com/file/d/1cqxEG8MJDiZYbuiFhrPjZ4pXOOaEFsLq/view?usp=sharing

NETWORKING PROTOCOLS (VERY IMPORTANT)

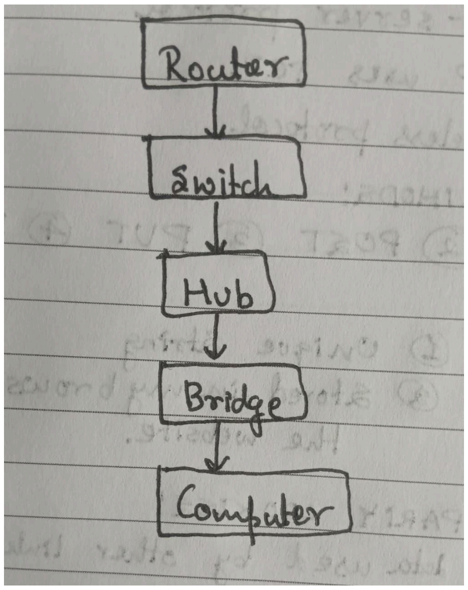
In order to understand how NIC works, it is very much a necessity to understand NETWORK PROTOCOLS. To understand network protocols, you can refer to the link given : https://youtu.be/IPvYjXCsTg8?si=VY8hOYA_8QH7gTOz

To give you a general idea, there are some models in the market that arrange all the processes that take place to transfer data through the internet in some sort of layers. The two major models are OSI MODEL and TCP MODEL.

Before diving into these protocols we need to know some of the definitions first :

1. REPEATER : Operates at physical layer.Its job is to regenerate the signal over same network before the signal becomes too weak. It's a two port device.
2. HUB : It is a multiport repeater. Hub cannot filter data. So it sends data packets to all connected devices. They do not have the intelligence to find the best path. There are two types of hubs.
    a. ACTIVE HUB : They have their own power supply, can clean, boost and relay signals along the network. They serve as both repeater and wiring centre. It is used to extend distance between wiring nodes.
    b. PASSIVE HUB : They collect wiring from nodes, power source from active hub, relay without cleaning and boosting, it cannot be used to extend between nodes.
3. BRIDGE : Operates at Data Link Layer. It is basically a repeater + function of filtering data based on MAC address of source and destination. It is used to interconnect two LANs working on same protocol. It is a two port device. It can be divided into two parts :

   a. TRANSPORT BRIDGES : Stations are unaware of the bridge's existence. Reconfiguration of stations is unnecessary whether or not a bridge is added. It makes use of two processes, 1. Bridge forwarding and 2. Bridge Learning.
   b. SOURCE ROUTING BRIDGES : Routing operation is performed by source station, frames specify which route to follow. The host can discover a frame called discovery frame which spreads through the entire network using all possible paths to destination.
4. SWITCH : Multiport bridge with a buffer + design to boost efficiency and performance, does error checking before forwarding data.
5. ROUTERS : Routes the data based on IP addresses. Routers mainly connect LANs and WANs together and have a dynamically updating routing table based on which they make decisions on routing data packets.
6. GATEWAY : Passage to connect two networks together that may work on different models. This is also called PROTOCOL CONVERTER. It is more complex.
7. BROUTER : It is a bridging router. It combines both bridge and router. It is capable of filtering LAN traffic.



8. IP ADDRESS :
9. MAC ADDRESS :
10. PORT NUMBER :

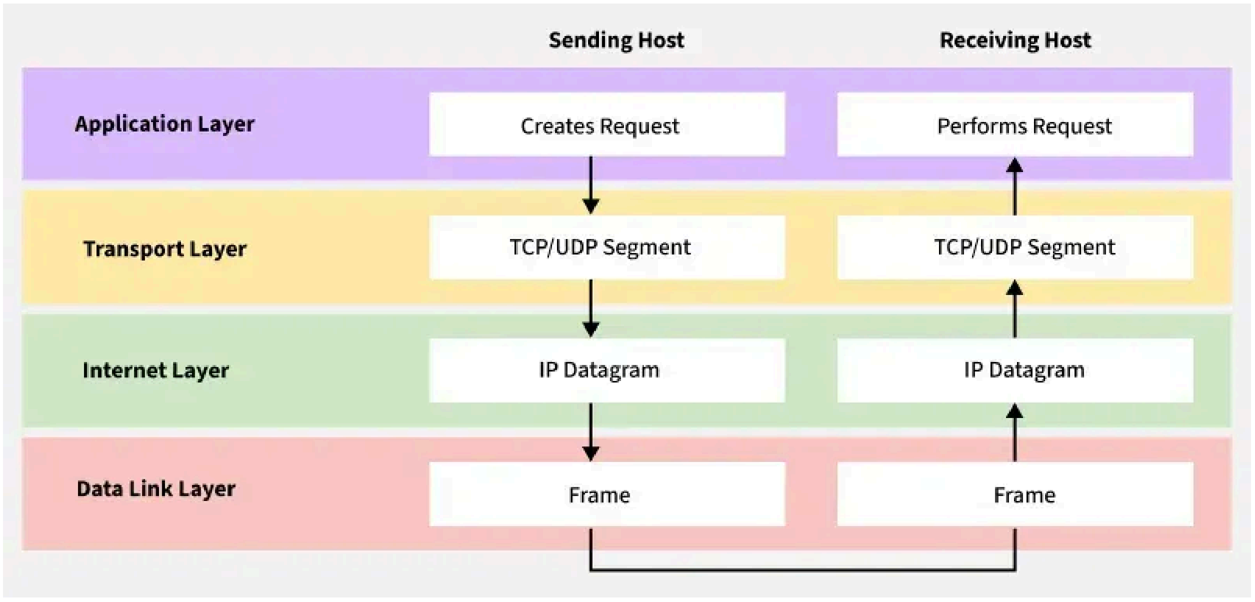
11. SOCKET : Interface between processes and the internet to help messages reach their destination.

TCP(TRANSMISSION CONTROL PROTOCOL)/IP(INTERNET PROTOCOL) MODEL :

It has four layers :

1. APPLICATION : The Application Layer is the top layer of the TCP/IP model and the one closest to the user. This is where all the apps you use like web browsers, email clients, or file sharing tools connect to the network. It acts like a bridge between your software (like Chrome, Gmail, or WhatsApp) and the lower layers of the network that actually send and receive data.
2. TRANSPORT : The Transport Layer is responsible for making sure that data is sent reliably and in the correct order between devices.
3. NETWORK/INTERNET : The Internet Layer is used for finding the best path for data to travel across different networks so it can reach the right destination.
4. NETWORK ACCESS : The Network Access Layer is the bottom layer of the TCP/IP model. It deals with the actual physical connection between devices on the same local network like computers connected by cables or communicating through Wi-Fi.

One of its goals is to make sure that the data sent by the sender arrives safely and correctly at the receiver's end. To do this, the data is broken down into smaller parts called packets before being sent (statement from geeksforgeeks).



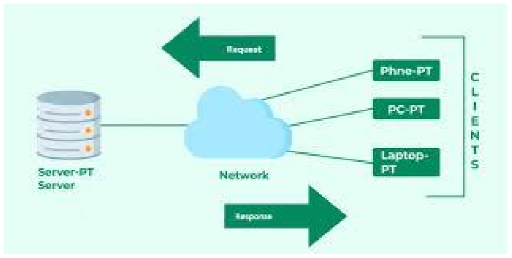
OSI (OPEN SYSTEMS INTERCONNECION) MODEL :

It is developed by INTERNATIONAL ORGANISATION OF STANDARDISATION (ISO).

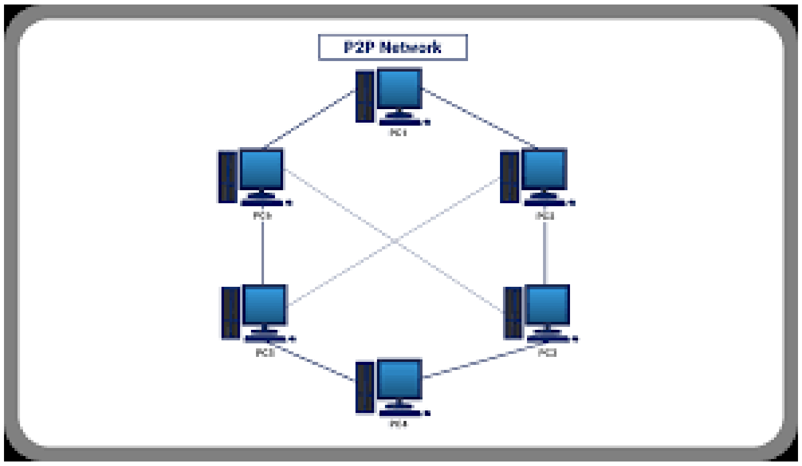
It consists of seven layers.

7.APPLICATION LAYER :

1. This is where user interaction takes place. Examples would be whatsapp, X,etc.
2. Application Layer has its own protocols that it has to follow for the proper transmission of data.
    a. CLIENT- SERVER ARCHITECTURE:



b. P2P (PEER TO PEER) eg : torrent.



(NOTE : ping = Round Trip TIme (the duration for a network packet to travel from a source (like your computer) to a destination (like a server) and for an acknowledgment or response to return to the source, measured in milliseconds))

6. And 5. PRESENTATION AND SESSION LAYER :

Session Establishment: Initiates and negotiates communication parameters (e.g., authentication, duplex mode).

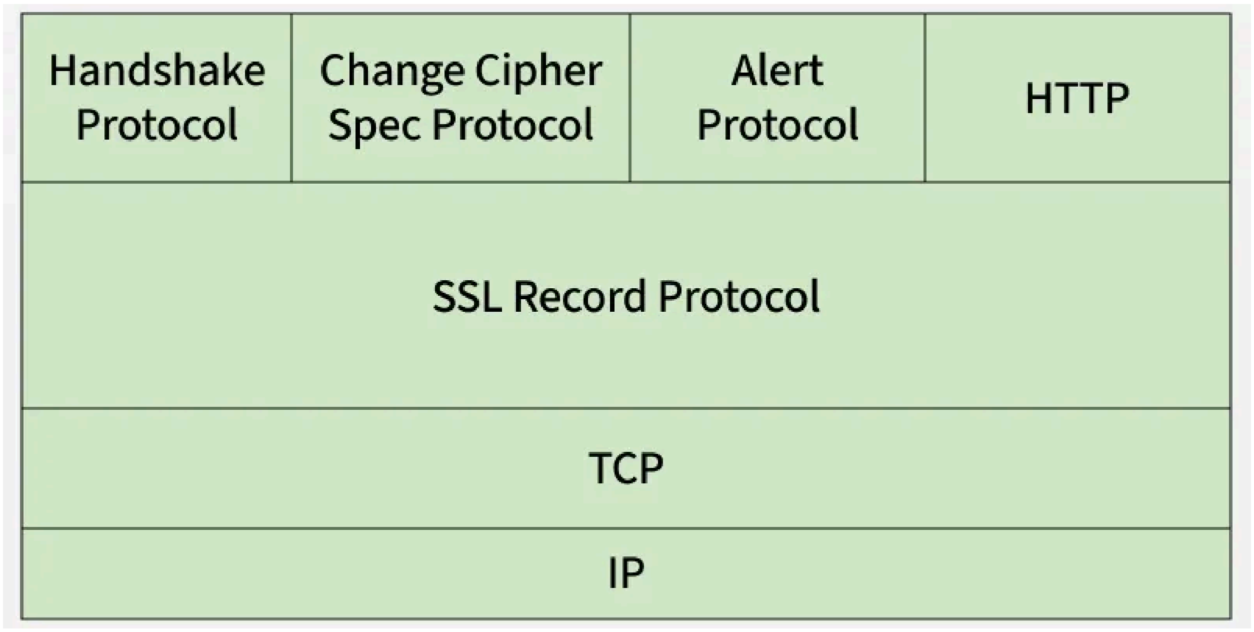
- **Communication Synchronization:** Keeps data streams in order using checkpoints.
- **Activity & Dialog Management:** Controls turns, prevents collisions, and avoids duplication.
- **Resynchronization & Recovery:** Recovers from failures using synchronization points.
- **Session Termination:** Gracefully ends communication after all data is exchanged.

The Presentation Layer (Layer 6) in the OSI model acts as a data translator, ensuring data from one system's Application Layer is readable by another's. Protocols like PPTP(point to point tunneling protocol), MIME(multipurpose Internet Mail Extension)protocol come under this layer.

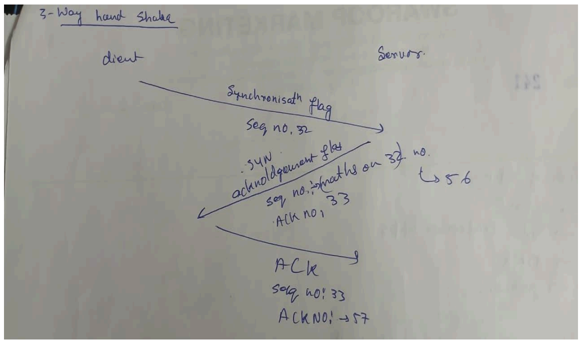
If we dig deeper we get to know that the session layer has one layer of its own called Secure Socket Layer (SSL).

Secure Sockets Layer (SSL) is an Internet security protocol that encrypts data to ensure secure communication between devices over a network. SSL ensures secure communication through three main mechanisms:

1. **Encryption:** Data transmitted over the network is encrypted, preventing unauthorized parties from reading it. If intercepted, encrypted data appears as an unreadable jumble of characters.
2. **Authentication:** SSL uses a handshake process to authenticate both the client and server, ensuring each party is legitimate and not an imposter.
3. **Data Integrity:** SSL digitally signs transmitted data to detect any tampering, ensuring that the data received is exactly what was sent.



Now the given figure below shows the way handshake authentication happens :



Change Cipher Spec (CCS) Protocol is a very simple but critical signal that tells the other party: "Stop listening to me in plain text; everything I send after this byte will be encrypted."
Think of the encryption process like a high-security gate with a light:

- **The Key Exchange:** This is where the Client and Server agree on a "Secret Password" (the Session Key). Now they both have a copy of the password.

- **The CCS (The Light):** This is a literal signal. When the Client sends a CCS, it's like turning on a green light that says, "Everything coming through this gate from now on is locked with that password we just agreed on."

Who can decrypt it?

Since both the Client and the Server helped generate the **Session Keys**, they both have the "mathematical tools" to decrypt the messages.

Alert Protocol alerts the systems if something goes wrong.

While HTTP is a whole different story and we will cover it when we discuss Internet protocols. Similarly we will discuss IP and TCP in the later parts.

4. TRANSPORT LAYER :

While transportation of messages is done by network layer, Transport layer does transportation of data from network to the application.

The transport layer has its own protocols. Mainly TCP and UDP.

Before we know more about them let us first define two basic divisions of protocols:

(In networking, a State refers to the information a system remembers about a conversation.)

1. STATEFUL PROTOCOLS : A stateful protocol requires the system to remember the status of the communication.

   How it works: The protocol tracks the "state" of the connection (e.g., *Initiated, Authenticated, Transferring, Closing*). If a message arrives out of order or without a prior "Hello," the protocol will reject it.

   Analogy: A **Phone Call**. You say "Hello," the other person says "Hello" back. You have a conversation based on what was said earlier. If you suddenly say "Goodbye," it makes sense because you were already in the "talking" state.

   Eg : TCP, FTP(FILE TRANSFER PROTOCOL), SSL, etc.

2. STATELESS PROTOCOLS : A stateless protocol treats every request as an independent transaction. It has no "memory" of previous interactions.

   How it works: Every message must contain **all** the information needed to understand and process the request.

   Analogy: A **Vending Machine**. Every time you put in a coin and press a button, it doesn't care if you bought five sodas a minute ago. It only cares about the current coin and the current button press.

   Eg : HHTP, UDP, IP, etc.

TCP :

TCP is defined by three main pillars :

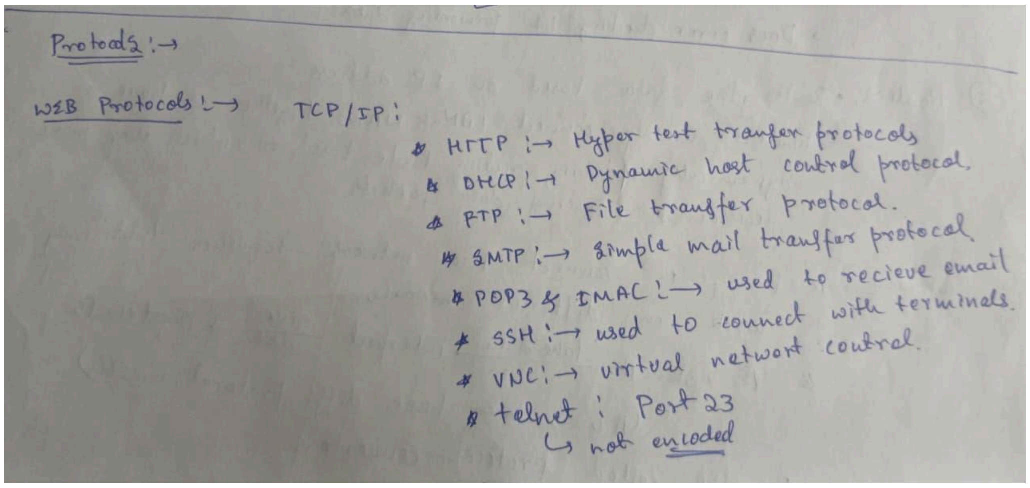
Connection-Oriented: Before any data is sent, a formal "handshake" must occur to synchronize the sender and receiver.

Reliability: It uses **Positive Acknowledgment with Retransmission (PAR)**. If a packet is lost or corrupted (detected via a checksum), the sender resends it.

Ordered Delivery: Each byte is assigned a **Sequence Number**. If packets arrive out of order due to network routing, TCP reassembles them correctly before passing them to the application.

Application Layer sends a lot of raw data, TCP divides this in chunks, and adds addresses. It may also collect data from network layers. It has congestion control algorithms built in. It has features like error control and it is a full duplex protocol.

In fact web protocols are TCP/IP protocols and they are many in number. The given image shows some of them :



Protocols :→

WEB Protocols :→ TCP/IP :

* HTTP :→ Hyper text tranfer protocols
* DHCP :→ Dynamic host control protocol.
* FTP :→ File transfer protocol.
* SMTP :→ Simple mail transfer protocol.
* POP3 & IMAC :→ used to recieve email
* SSH :→ used to connect with terminals.
* VNC :→ virtual network control.
* telnet : Port 23
  ↳ not encoded

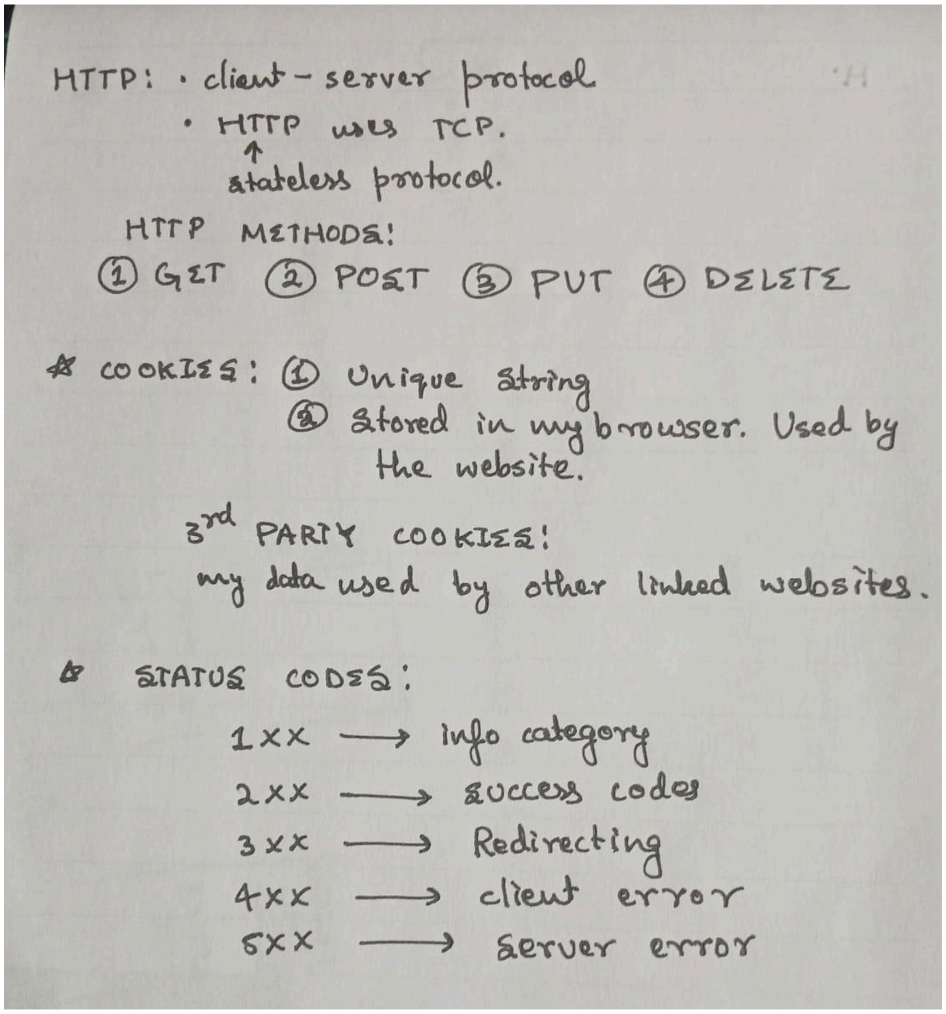
IMPORTANT NOTE :

If tcp/ip model is web protocol then why do we compare it with OSI model ?

TCP is just a protocol in the Transport layer. However, when people say "The TCP/IP Model," they are referring to a **4-layer architectural framework** that competes with the **7-layer OSI Model**.

We don't compare the *TCP protocol* to the *OSI model*; we compare the **TCP/IP 4-layer architecture** to the **OSI 7-layer architecture**.

HTTP:



HTTP: • client - server protocol
• HTTP uses TCP.
↑
stateless protocol.

HTTP METHODS:
① GET ② POST ③ PUT ④ DELETE

☆ COOKIES: ① Unique string
② Stored in my browser. Used by the website.

3rd PARTY COOKIES:
my data used by other linked websites.

☆ STATUS CODES:

1xx ⟶ info category
2xx ⟶ success codes
3xx ⟶ Redirecting
4xx ⟶ client error
5xx ⟶ server error

3. NETWORK LAYER :

Network Layer is the third layer from the bottom (Layer 3) and the fifth layer from the top in the OSI (Open Systems Interconnection) Model. It is responsible for ensuring end-to-end packet delivery across multiple interconnected networks.

4. **Logical Addressing:** Assigns unique IP addresses to devices, ensuring accurate identification and communication across networks.
5. **Packetization**: Encapsulates transport layer segments into packets for efficient transmission.
6. **Host-to-Host Delivery**: Ensures reliable delivery of packets from the sender to the intended receiver across diverse networks.
7. **Forwarding**: Moves packets from the input interface of a router to the appropriate output interface based on the destination IP.
8. **Routing**: Determines the optimal path for packets to travel across multiple networks using routing algorithms and protocols.
9. **Fragmentation and Reassembly**: Splits large packets into smaller fragments to match the maximum transmission unit (MTU) of a network, and reassembles them at the destination.
10. **Subnetting**: Divides larger networks into smaller subnetworks for efficient addressing and traffic management.
11. **Network Address Translation (NAT)**: Maps private IPs to public IPs for internet communication, conserving address space and adding security.

Each device is assigned a unique logical address (IP address).

- Data from the transport layer is encapsulated into packets, with source and destination IPs attached.
- Routers analyze the destination address and determine the best available path.
- Packets traverse the network hop-by-hop, moving across routers until reaching the destination.
- If the packet size exceeds the MTU, it is fragmented into smaller units.
- At the destination, the fragments are reassembled into the original data.

- If errors occur (e.g., unreachable destination), protocols like ICMP send error messages back to the source.

Protocols Operating at the Network Layer

- IP (Internet Protocol – IPv4/IPv6)
- ICMP (Internet Control Message Protocol)
- ARP (Address Resolution Protocol)
- RARP (Reverse Address Resolution Protocol)
- NAT (Network Address Translation)
- IPSec (Internet Protocol Security)
- MPLS (Multiprotocol Label Switching)

Routing Protocols

- RIP (Routing Information Protocol)
- OSPF (Open Shortest Path First)
- BGP (Border Gateway Protocol)

2.DATA-LINK LAYER :

1. is the second layer from the bottom of OSI. Its major work is to ensure error free transmission of data.
2. Also responsible for encoding, decoding, and organizing the outgoing and incoming signals.
    1. It is divided into two sub layers : 1. **Logical Link Control (LLC) 2. Media Access Control (MAC)**
        1. LLC : This sublayer of the data link layer deals with multiplexing, the flow of data among applications and other services, and LLC is responsible for providing error messages and acknowledgments as well.

2. MAC : MAC sublayer manages the device's interaction, responsible for addressing frames, and also controls physical media access. The data link layer receives the information in the form of packets from the Network layer, it divides packets into frames and sends those frames bit-by-bit to the underlying physical layer.

1.PHYSICAL LAYER (PHY):

It is the bottom most layer of OSI. It is basically the hardware that is used to transmit data, either through ports , wired or wireless. It can convert signals from analog to digital or digital to analog.

Now, since we have basic understanding about protocols, internet and different models, we will theoretically define how a simplified ethernet card is going to work.

NOTES :

➢ CHECKSUMS :
➢ CONGESTIONS :
➢ TIMERS :
➢

PLANNING PHASE :

Step 001 :

We will install PuTTy software on our laptop. Connect USB 2.0 to USB to UART converter of FPGA. We will find the COM PORT in PuTTy that connects us virtually to fgpa. Now we can send data to fpga. FPGA receives it in the form of UART packets.

INPUT	OUTPUT

USB 2.0	UART PORTS
Data via PuTTy	UART packets

Step 002 :

To actually "get" the text inside your FPGA, you need a UART Receiver module. This module monitors the RX pin. Because UART is asynchronous (no shared clock), it uses a technique called Oversampling.

Baud Rate: You must decide how fast to talk (e.g., 115,200 bits per second).

Sampling: The FPGA clock is much faster than the baud rate. If your clock is 100MHz and baud is 115,200, the FPGA waits for approximately 868 clock cycles between each bit.

The Logic: 1. Wait for the RX line to drop from '1' to '0' (The **Start Bit**). 2. Wait 1.5 bit-periods to reach the middle of the first data bit. 3. Sample the bit, store it in a shift register, and wait 1 bit-period for the next one. 4. After 8 bits, the data_valid signal goes high.

This is done using the verilog logic given in uart_data.v file.

Step 003 :

Since, data has been fetched from the source, now our duty is to add necessary protocol requirements and make it transportable via ethernet. So, first is transport layer, we have two choices, one is TCP and the other is UDP. UDP is much more easier to implement than TCP. Since this is a simplified NIC, we are going to use UDP protocol.

UDP PROTOCOL

In User Datagram Protocol data may or may not be delivered, data order may change, data may change. It is a STATELESS protocol. It uses checksums, but does not do anything about errors.

Basically when UDP envelope looks like this :

Field	**Size**	**Description**
Source Port	16 bits	The port number of the sender. It's optional; if not used, it is set to zero.
Destination Port	16 bits	The port number of the receiver. This is required to deliver the "letter" to the right application.
Length	16 bits	The total length of the UDP header and the data combined (minimum value is 8).
Checksum	16 bits	Used for error-checking the header and data. It is optional in IPv4 but mandatory in IPv6.

Ports are necessary to ensure that data has reached the required application. For example, if whatsApp had to receive data from WhatsApp on another device, then the source port would be the port number of whatsApp on the sender's device and destination port would be the port number of whatsApp on the receiver's device. Since we are not using any application we will assign some high number like 5000.

So now, data is enveloped using verilog code in udp_envelope.v file.

After UDP envelope which is necessary for internal transportation, next comes ipEnvelope

IP ADDRESSES :

IP addresses help to find a device on the internet globally to send or receive data. There are two types of IP addresses. 1. IPv4 2. IPv6

IPv4 :

It is a 32 bit binary number. This implies 2^32 possibilities which is approximately 4.3 billion addresses. Since it is hard for us to read binary numbers, we divide it into four 8-bit groups called OCTETS.

Each octet can range from 00000000 to 11111111 which i s 0 to 255 decimal numbers.

IPv4 has 4 octets and they are separated using dots. Example : 192.168.0.0

An IP address is split into two logical parts. Think of it like a phone number where the first few digits are the **Area Code (Network)** and the rest is the **Subscriber Number (Host)**.

Subnet Mask (e.g., 255.255.255.0): This tells the hardware: "The first 24 bits are the Network ID. Ignore them when looking for devices inside your own house."

CIDR Notation: You'll often see /24 or /16. This is just a shortcut for counting the number of "1" bits in the mask. A /24 means the first 24 bits are for the network.

Private IPs: Used inside your home or lab (like 192.168.x.x or 10.x.x.x). Millions of people use the exact same private IP address simultaneously because they are separated by routers.

Public IPs: Globally unique. Only one device in the world can have a specific public IP at any given time.

NAT (Network Address Translation): Your router acts as a translator, taking data from your private IP and sending it out to the internet using its one single Public IP.

Because we ran out of the 4.3 billion IPv4 addresses, **IPv6** was created.

IPv6:

Size: 128 bits (2128 addresses—more than the number of grains of sand on Earth).

Format: Hexadecimal (e.g., 2001:0db8:85a3:0000:0000:8a2e:0370:7334).

IP PACKETS :

Since, we have understood what IP addresses are, let us look into how IP addresses are packed and how does IP headers look like :

Since IPv6 is complicated, we will implement IPv4.

IPv4 header + udp envelope makes up IP PACKETS. We already implemented an udp envelope, we will now implement an IP header.

The IP header has a 20-bytes(160 bits) block of data at the front of the udp envelope. It is organised into 32-bit rows.

Row	**0 - 3**	**4 - 7**	**8 - 15**	**16 - 31**
1	Version (4)	IHL(5)	Type of Service (ToS) (0)	Total Length (29)
2	Identification (1) - 16 bits	Flags - 3 bits (3'b010)	Fragment Offset (0)-13bits	

3	Time to Live (TTL) - 8 bits (8’d64)	Protocol - 8bits (17)	Header Checksum - 16 bits (0)	
4	**Source IP Address (32 bits)**			
5	**Destination IP Address (32 bits)**			

Row 1: The Basics

- **Version (4 bits):** * **Value:** Almost always 4 (0100 in binary).
    - **Purpose:** Tells the hardware which IP version is being used. If your NIC sees a 6 (0110), it knows it’s an IPv6 packet and needs to use a different parsing logic.
- **IHL (Internet Header Length - 4 bits):** * **Purpose:** Tell the NIC how long the header is.
    - **Logic:** It counts 32-bit words. The minimum is 5 (5×4 bytes=20 bytes). If this value is greater than 5, it means "Options" have been added to the header, and your UDP data will start further back in the stream.
- **Type of Service / DSCP (8 bits):** * **Purpose:** Used for **Quality of Service (QoS)**.
    - You can use these bits to prioritize certain traffic. For example, you could program your FPGA to send "Video Stream" packets to a fast lane and "Background Downloads" to a slow lane.
    - Not required for a simple NIC.
    - **Differentiated Services Code Point = DSCP**
- **Total Length (16 bits):** * **Purpose:** The size of the **entire packet** (Header + Payload) in bytes. (20 bytes of header + 8 bytes of port number + 1 byte of data = 29)

    - **Max Value:** 65,535 bytes. Your FPGA uses this to know exactly when the current packet ends and the next one begins.

Row 2: Fragmentation (The "Puzzle" Logic)

- **Identification (16 bits):** * **Purpose:** A unique ID for the packet.
    - **Logic:** If a large packet is broken into smaller pieces (fragmented) to fit through a small "pipe" in the network, all pieces will have this same ID so the receiver can put them back together.
    - If this value is set to 1 then the sender has assigned the integer value "1" to this specific datagram. Many operating systems use a counter that increments by 1 for every packet sent from a source to a destination; an ID of 1 often indicates it is the **first packet** sent after a counter reset or session start.
- **Flags (3 bits):** * **Purpose:** Control bits for fragmentation.
    - *Bit 0:* Reserved (always 0).
    - *Bit 1 (DF):* **Don't Fragment**. If set, routers aren't allowed to break this packet up. We will be using this as our packet is not any big.
    - *Bit 2 (MF):* **More Fragments**. If set to 1, it tells the receiver, "Wait, there are more pieces of this puzzle coming!"
- **Fragment Offset (13 bits):** * **Purpose:** Tells the receiver where this specific piece fits in the original large packet. It's like the "page number" of a book. It is essential because IP packets are "connectionless," meaning fragments of the same original message can arrive at the destination **out of order** or via different paths.
- While the *Identification* field tells the receiver which fragments belong to the same original packet, the **Fragment Offset** tells the receiver exactly **where each piece fits** in the reassembly.
- Since we are using the UDP protocol, we are going to set this to 0.

Row 3: Handling & Security

- **Time to Live / TTL (8 bits):** * **Purpose:** Prevents packets from looping forever if there's a mistake in the network.

    - **Logic:** Every time a packet hits a router, the router subtracts 1 from this number. If it hits 0, the router throws the packet away.
    - When TTL (Time-To-Live) is set to 8'b01000000, it means the value is an **8-bit binary number (01000000)**, which translates to the decimal value **64**, indicating a packet can survive **64 hops (router traversals)** before being discarded to prevent infinite loops in a network, a common default for systems like Linux/macOS.
- **Protocol (8 bits):** * **Purpose:** Tells your NIC what "language" the next layer is speaking.
    - **Values: 17** for UDP, **6** for TCP, **1** for ICMP (Ping). Your FPGA state machine uses this to decide whether to trigger your UDP-processing logic.
- **Header Checksum (16 bits):** * **Purpose:** Error detection.
    - **Logic:** The sender does some math on the header bits and puts the result here. Your FPGA will perform the same math on the arriving bits. If the results don't match, it means the header was corrupted (e.g., by electrical noise on the cable), and the NIC should drop it immediately.
    - UDP protocol does not require this. So we set this to zero as well.

SINCE WE ARE BUILDING A SIMPLIFIED NIC, WE WILL HARDWIRE IP ADDRESSES OF SOURCE AND DESTINATION.

Byte Offset	**Field Name**	**Description**
0	Version / IHL	0x45 (IPv4, 5 words long).
2-3	Total Length	Length of Header + UDP + Data.

8	Time to Live (TTL)	Usually 0x40 (64 decimal).
9	Protocol	0x11 (This tells the PC there is **UDP** inside).
10-11	**Header Checksum**	**Critical:** Must be calculated or the PC will drop the packet.
12-15	**Source IP**	Your FPGA's IP address.
16-19	**Dest IP**	Your Laptop's IP address.

This has been implemented using verilog in the ipEnvelope.v file.

Step 003 :

Now, we have formed the packet, it is necessary for us to wrap packets into frames and make it ready to transmit via ethernet cable.

There are many Ethernet PHY interfaces and data has to be transported according to their conditions and those depend upon the FPGA you are using. The FPGA I have has a MII interface.

The **media-independent interface** (**MII**) was originally defined as a standard interface to connect a Fast Ethernet (i.e., 100 Mbit/s) medium access control (MAC) block to a PHY chip. The MII is standardized by IEEE 802.3u and connects different types of PHYs to MACs. Being *media independent* means that different types of PHY devices for connecting to different media (i.e. twisted pair, fiber optic, etc.) can be used without redesigning or replacing the MAC hardware. Thus any MAC may be used with any PHY, independent of the network signal transmission medium.

MII interface takes input as 4 bits so we need to modify our logic according to that. Also we need to take care of the Endianness rule. We will discuss these things in the testing part. Right now, the only thing we need to know is that after the IP packet is ready, we now need to add some more layers on it to make it transferable. We call these final packets as frames. What NICs accept are frames.

Frame consists of these fields :

Field	**Size (Bytes)**	**Purpose**
Preamble	7	A pattern of 10101010 to help the receiver clock synchronize.
SFD (Start Frame Delimiter)	1	The byte 10101011 that says "Data starts now!"
Destination MAC	6	The hardware address of the receiver.
Source MAC	6	Your NIC's hardware address.
EtherType / Length	2	Tells the system if the "payload" is IPv4, IPv6, or ARP.

Payload (Data)	46 – 1500	This is where your IP packets and actual data live.
FCS (Frame Check Sequence)	4	A CRC (Cyclic Redundancy Check) to ensure no bits were flipped.

1. The Preamble (7 Bytes)

This is a sequence of alternating 1s and 0s (10101010...).

- In high-speed networking, the receiver's clock might be slightly out of sync with the sender's. The Preamble acts like a "metronome," allowing the receiving PHY chip to lock onto the timing of the incoming bits.
- Most FPGAs and NIC controllers never see this; the **PHY chip** consumes it and only triggers your logic once the frame actually starts.

2. Start Frame Delimiter / SFD (1 Byte)

The SFD is the byte 10101011.

- Notice the last bit is a 1 instead of a 0. This breaks the preamble pattern and tells the hardware: **"The very next bit is the start of the Destination MAC address."**
- This is your "Go" signal. In Verilog, this is often where your data_valid signal would go high.

3. Destination MAC Address (6 Bytes)

- This tells the network switch where to send the frame.
- If the MAC doesn't match your NIC's address (and it's not a broadcast address like FF:FF:FF:FF:FF:FF), your FPGA should usually "drop" the frame immediately to save memory and processing power.

4. Source MAC Address (6 Bytes)

- This identifies who sent the data.
- When your NIC sends a response, you will need to "flip" these; the old Source MAC becomes your new Destination MAC.

5. EtherType / Length (2 Bytes)

This field has two possible roles depending on its value:

- **Type:** If the value is ≥1536 (0x0600), it tells you what protocol is inside the payload (e.g., 0x0800 for **IPv4**, 0x86DD for **IPv6**, or 0x0806 for **ARP**).
- **Length:** If the value is small (<1500), it simply indicates how many bytes of data follow.
- Your FPGA's parser will use this to decide which "branch" of logic to send the data to (e.g., an IPv4 processing module).

6. Payload (46 – 1500 Bytes)

This is the "cargo." It contains the actual IP packet, UDP headers, and your data.

- **Padding:** Ethernet frames must be at least 64 bytes total. If your data is too small (like a 1-byte "Hello"), the hardware adds "padding" (zeros) to reach the minimum size.
- **MTU:** The standard maximum is 1500 bytes. If you go over this, it's called a **Jumbo Frame**.

7. Frame Check Sequence / FCS (4 Bytes)

This is a **CRC-32** (Cyclic Redundancy Check) checksum.

- As electricity travels down a wire, noise can flip a 1 to a 0. The sender calculates a math formula based on the frame data and puts the result here.

- The receiver performs the same math. If the receiver's result doesn't match the FCS, the frame is corrupted and your NIC must discard it. This is usually the last thing your FPGA checks before moving the data into your **FIFO.**

- ➢ **MII AND UDP PROTOCOLS USE FIFO METHOD OF DATA TRANSMISSION.**
- ➢ **MAC (Media Access Control) Addresses are imprinted on your drivers that take care of your data. I.e, ip address identifies your device, when data reaches routers, it looks for a mac address on your device. MAC address is assigned to your ethernet driver and wifi driver. If mac address used is of ETHERNET driver then data is transmitted via ETHERNET driver. If the mac address does not match any of the mac addresses of the device, then error protocols are initiated.**

Once the frame is ready, it is sent via ethernet cable to the required destination, in our case, it is a laptop, where the NIC of the laptop reads the 4 bit data incoming and follows the necessary protocols to read or discard the data.

This is implemented using verilog code in frame.v file.

Step 004 :

Until now, we have taken care of transmission of data. Noe, we will take care of receiving the data. In simplified NIC, receiver is not much of a hurdle. We are going to ignore many of the complicated issues and just focus on receiving the data.

Just like frames sent, we receive frames as well.

But frames received are in 4 bits(nibbles) but we need 8 bits data to easily work upon. So we first stitch two nibbles together using some logic .

This is done using verilog code written in nibble_stitcher.v file.

The next step is to parse the data we get from nibble stitchers. While parsing we check for SFD(start frame delimiter), check if destination and receiver mac address is correct, check ether_type, ignore IP header and UDP_header because port number is not required. Then we collect data from payload and transmit data via UART to the laptop.

Transmission of data via UART is written using verilog in uart_tx.v file.

The parsing is written using verilog in the parser.v file.

Step 005 :

THE WHOLE TRANSMITTER AND RECEIVER FILES ARE INTEGRATED USING VERILOG IN exec_nic.v FILE WHICH IS TOP LEVEL ENTITY. WE THEN DO THE SIMULATION OF ALL VERILOG FILES TO ENSURE THAT PROGRAMS WRITTEN ARE WORKING FINE AND DO NOT CAUSE ANY DAMAGE TO FPGA.

Step 006 :

WE THEN SYNTHESISE THIS FILE AND RUN IT ON FPGA AND CHECK IF IT IS ACTUALLY WORKING.

TESTING AND SIMULATION PHASE :

1. Uart_data.v file

   Understanding UART PACKETS :

In UART (Universal Asynchronous Receiver/Transmitter), data is sent **serially** (one bit at a time) over a single wire. Unlike Ethernet, UART is "asynchronous," meaning there is no shared clock signal between the sender and receiver.

To make this work, the data is wrapped in a specific **packet structure** (often called a "frame") so the receiver knows exactly when the data starts and ends.

1. The Idle State (High)

Before any data is sent, the UART line is held at a **Logic High (1)** voltage level. This is called the "Idle" state. Holding it high allows the receiver to distinguish between a disconnected wire (Logic 0) and a working line that is simply waiting for data.

In UART (Universal Asynchronous Receiver/Transmitter), data is sent **serially** (one bit at a time) over a single wire. Unlike Ethernet, UART is "asynchronous," meaning there is no shared clock signal between the sender and receiver.

To make this work, the data is wrapped in a specific **packet structure** (often called a "frame") so the receiver knows exactly when the data starts and ends.

1. The Idle State (High)

Before any data is sent, the UART line is held at a **Logic High (1)** voltage level. This is called the "Idle" state. Holding it high allows the receiver to distinguish between a disconnected wire (Logic 0) and a working line that is simply waiting for data.

2. Start Bit (1 Bit)

The packet begins when the sender pulls the line from **High to Low (0)** for exactly one bit-period.

- **The Purpose:** This transition acts as a "wake-up call" for the receiver.

Hardware Note: The receiver sees this falling edge and starts its internal timer. Because both sides have agreed on a **Baud Rate** (speed), the receiver knows to sample the line at specific intervals after this bit.

3. Data Frame (5 to 9 Bits)

This is the actual information you are sending (like an ASCII character).

- **Size:** Most commonly **8 bits** (1 byte).

- **Endianness:** UART almost always sends the **LSB (Least Significant Bit) first**.
- **Hardware Note:** In your FPGA, you would typically use a shift register to collect these 8 bits as they arrive.

4. Parity Bit (0 or 1 Bit)

This is an optional bit used for very basic error checking.

- **Even Parity:** The bit is set so that the total count of "1s" in the data + parity bit is an even number.
- **Odd Parity:** The bit is set so the total count of "1s" is odd.

- **The Purpose:** If a single bit gets flipped by electrical noise, the parity check will fail, and the receiver knows the data is corrupted.
- **Hardware Note:** In modern systems, Parity is often set to "None" because it cannot detect if *two* bits are flipped.

5. Stop Bits (1 or 2 Bits)

To signal the end of the packet, the sender pulls the line back to **Logic High (1)** for at least 1 or 2 bit-periods.

The Purpose: This ensures the line returns to the Idle state so that the next Start Bit (which is a transition to Low) can be clearly detected. It also gives the receiver a tiny "breather" to process the byte before the next one arrives.

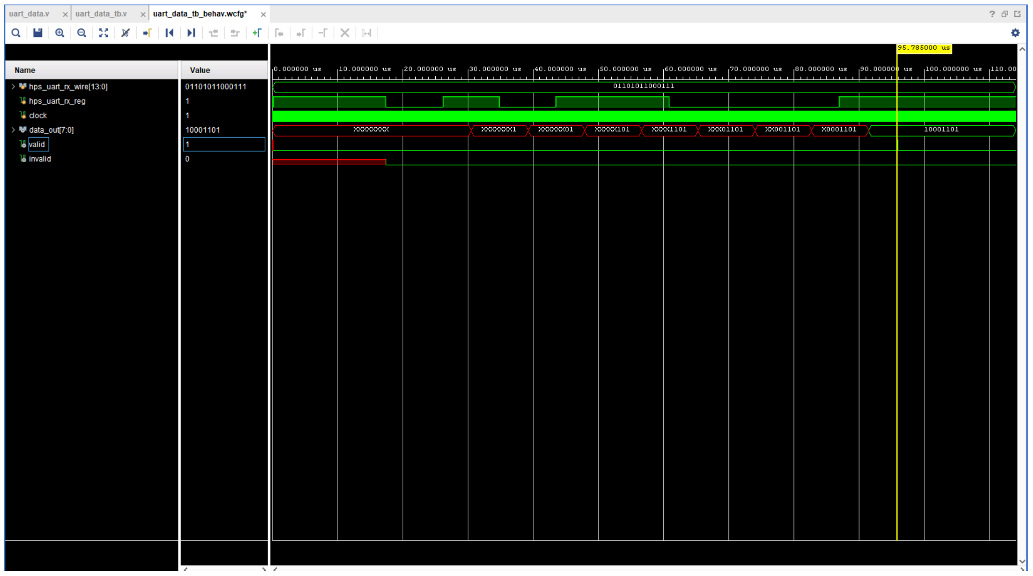
EXPLAINING THE CODE (USED AI AS IT WOULD CONSUME LOT OF TIME TYPING THIS) :

This module is a **Finite State Machine (FSM)** designed to receive a single byte of data over UART at a specific baud rate by sampling the incoming hps_uart_rx signal. It starts in the **START** state, where it monitors the line for a falling edge (the Start bit), though currently, your logic checks for specific bit patterns to trigger an invalid flag or transition to the next state. Once the start of a frame is detected, it moves to the **WORK** state, which uses two nested counters: a bit-period counter (counter) and a bit-index counter (increment). To ensure data reliability, it waits for approximately half a bit duration (434 clock cycles) before sampling the signal to capture the bit at its most stable center point, eventually shifting 8 bits into the data_out register. After all 8 bits are collected, the FSM checks for a valid **Stop bit** (Logic High) to transition to the **DONE** state, where it pulses the valid signal high to notify the rest of the system that a full byte is ready for processing before resetting itself back to the START state.

Key Observations in your logic:

- **The Sampling Point:** You used 434 as your sampling point and 868 as the full bit period. This is perfect for **115200 baud** on a **50 MHz clock**.
- **The Increment Logic:** Your data_out[increment-1] logic is clever, but be careful in the START state—if increment is 0, increment-1 could cause an underflow issue in some synthesizers.
- **The Invalid Flag:** Currently, your code flags invalid if it sees certain transitions. In a standard UART receiver, "invalid" is usually reserved for a **Framing Error** (when the Stop bit is missing).

SIMULATION :

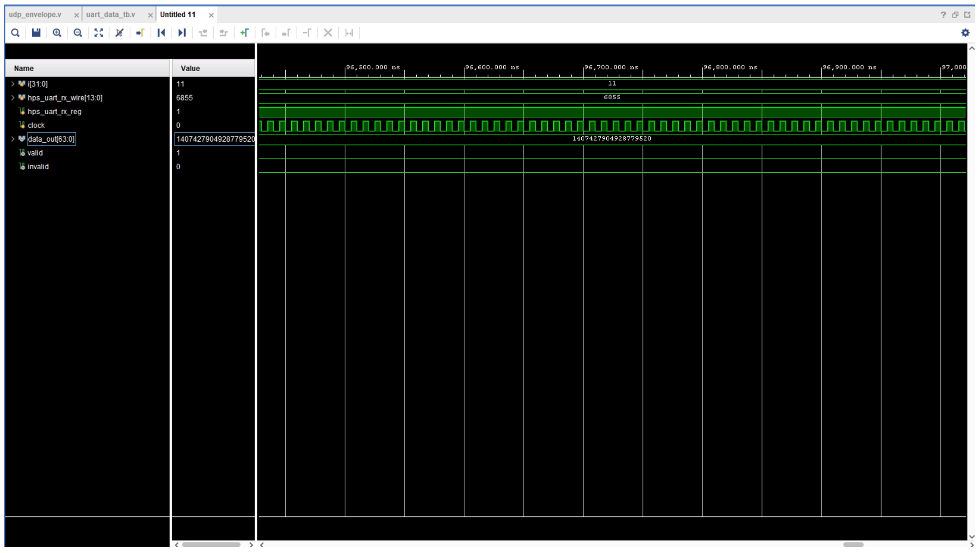


Name	Value
hps_uart_rx_wire[13:0]	01101011000111
hps_uart_rx_reg	1
clock	1
data_out[7:0]	10001101
valid	1
invalid	0

2. Udp_envelope.v

Most of this is explained in the planning phase so we will just do the simulation verification here.

SIMULATION :

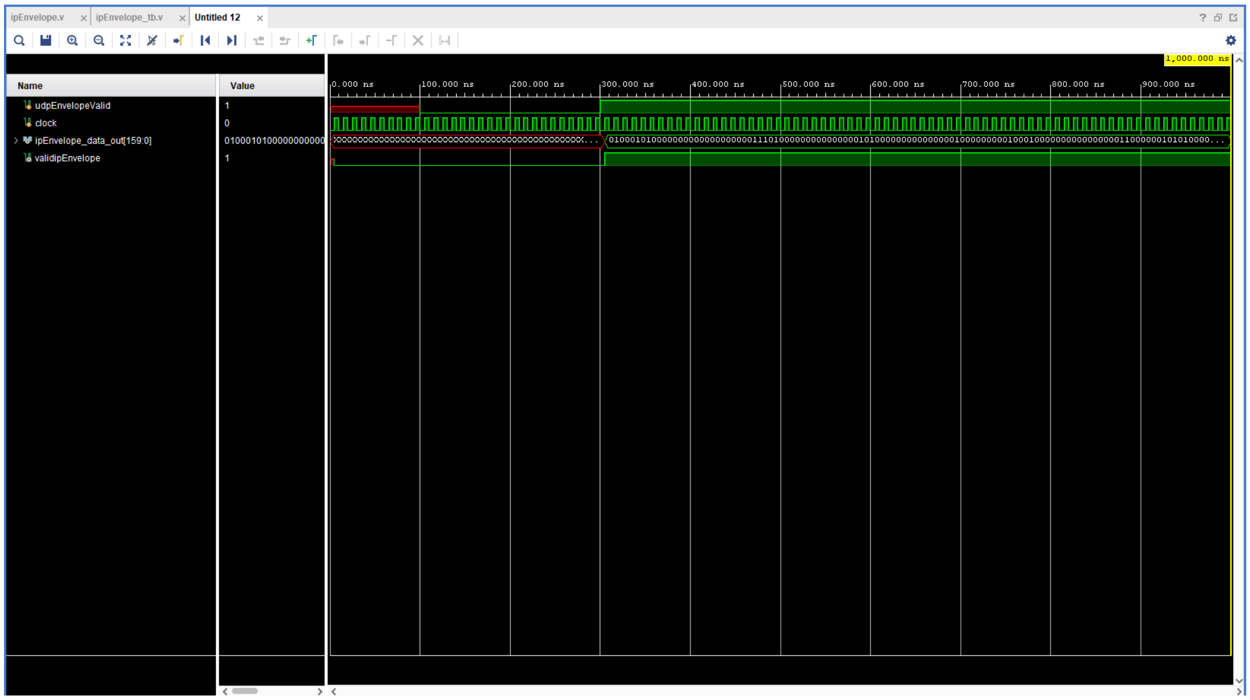


Name	Value
i[31:0]	11
hps_uart_rx_wire[13:0]	6855
hps_uart_rx_reg	1
clock	0
data_out[63:0]	140742790492877952
valid	1
invalid	0

3. ipEnvelope.v

Even in this section we have gained sufficient knowledge in the planning section.

SIMULATION :



Name	Value
udpEnvelopeValid	1
clock	0
ipEnvelope_data_out[159:0]	01000101000000000000
validipEnvelope	1

4. Crc32.v

WHY CRC32 :

The primary reason we implement CRC32 in the NIC frame is to detect **bit-flips**.

When data travels at high speeds (like 100 Mbps or 1 Gbps) over copper or fiber, electrical noise, heat, or interference can easily turn a 0 into a 1.

a. **How it works:** Before sending the frame, your NIC runs the entire frame (Header + Payload) through a mathematical formula (polynomial). The result is a 32-bit "fingerprint" called the **Frame Check Sequence (FCS)**, which is tacked onto the end of the frame.
b. **The Receiver's Job:** When your FPGA receives the frame, it performs the same math. If the receiver's result doesn't match the 32-bit FCS at the end

of the frame, the NIC knows the data is corrupted and **drops it immediately**.

c. **Why 32 bits?** A 32-bit check is the "sweet spot." It is statistically strong enough to detect almost any error in a frame up to 1500 bytes (MTU), but small enough that an FPGA can calculate it in real-time without slowing down the connection.

LOGIC :

The goal of CRC (Cyclic Redundancy Check) is to detect if any bits changed during transmission. It uses **Polynomial Long Division**, but in a special type of math called **Galois Field 2 (GF(2))**.

Process

- Complement the first 32 bits of the frame
- The first n-bits (addressing through the data) of the frame are the coefficients of the polynomial
- The polynomial M(x) is degree n-1
- M(x) is multiplied by x^{32} and divided by G(x) which produces remainder of degree ≤31
- The coefficients of R(x) are the 32 bit sequence which is appended to the end of the frame

First, the divisor is agreed by both receiver and sender. In your case, In your Verilog code, this is represented by the **Hexadecimal value**: 32'h04C11DB7.

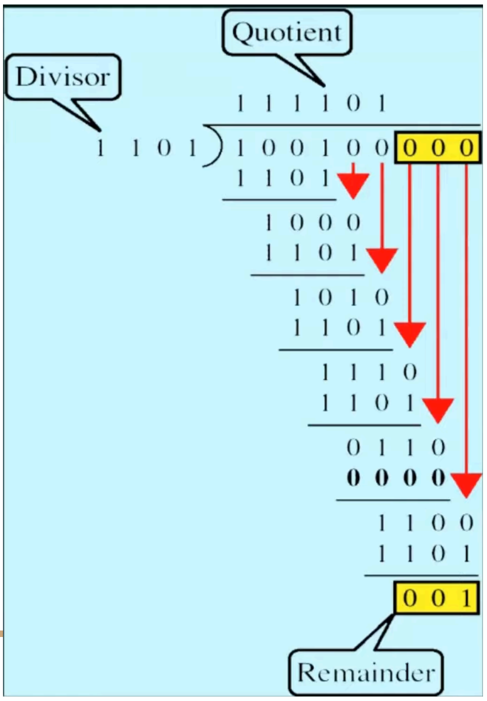
Step 1 : Find length of divisor. Let it be called "L".

Step 2 : Append L-1 bits to the original message.

Step 3 : Perform binary division operation.

Step 4 : The remainder of the division is CRC value.

Note that CRC must be of L-1 bits.

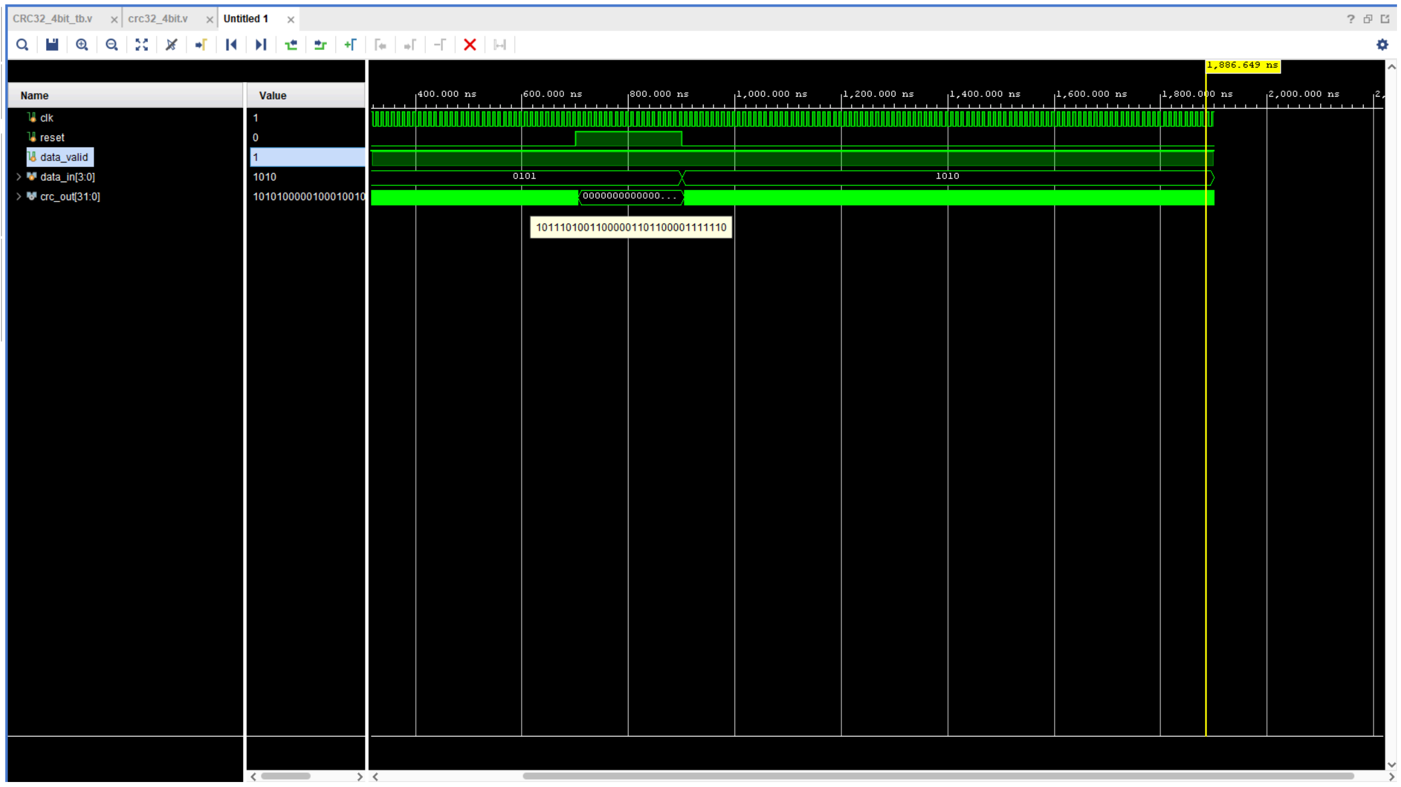


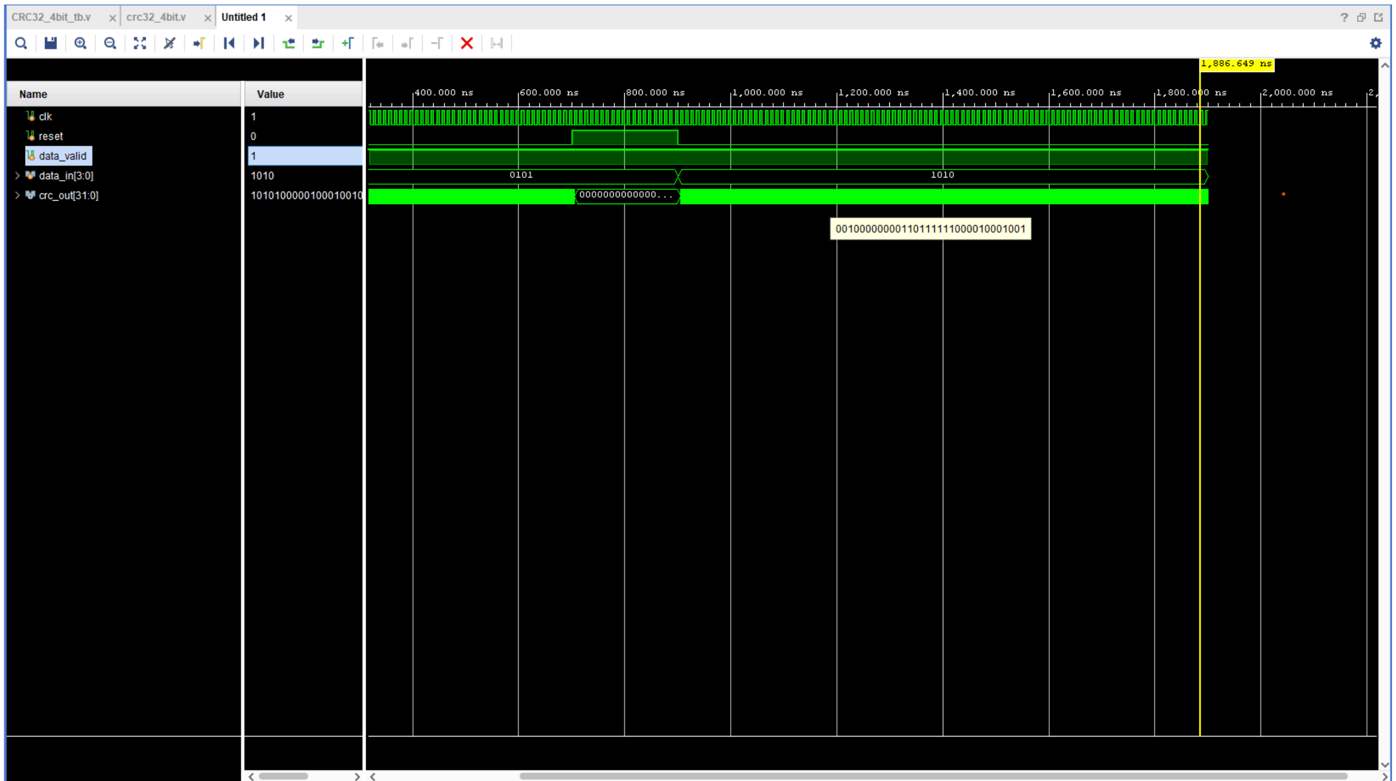
Is it not possible that same remainder will be left for different

data bits ?

The answer is yes, but it is 1 in 4.29 billion cases. So we generally believe in CRC.

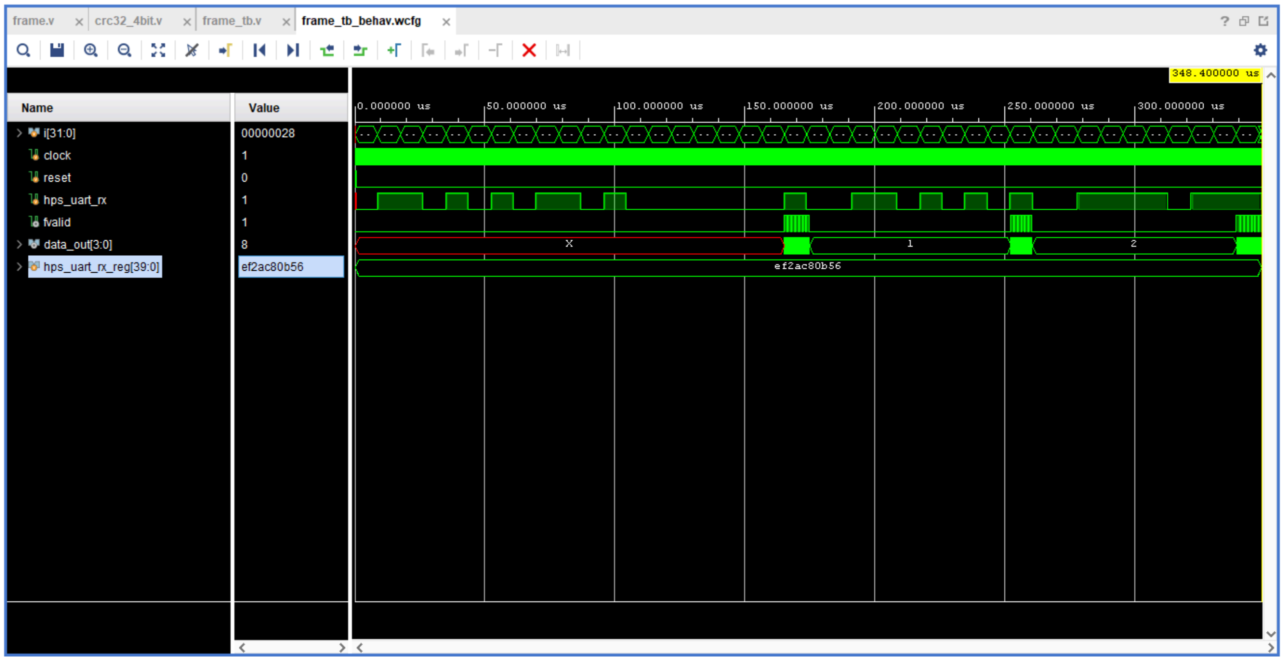
SIMULATION :


```
Name              Value
clk               1
reset             0
data_valid        1
data_in[3:0]      1010
crc_out[31:0]     101010000010001001
```


```
Name              Value
clk               1
reset             0
data_valid        1
data_in[3:0]      1010
crc_out[31:0]     101010000010001001
```

5. Frame.v :

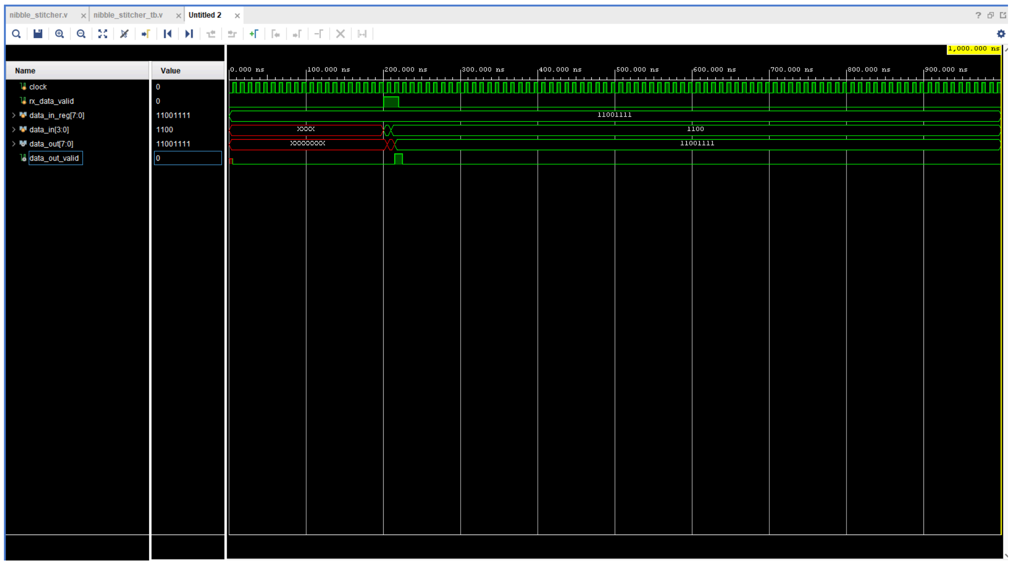
SIMULATION :



Name	Value
i[31:0]	00000028
clock	1
reset	0
hps_uart_rx	1
fvalid	1
data_out[3:0]	8
hps_uart_rx_reg[39:0]	ef2ac80b56

6. nibble_stictcher.v :

SIMULATION :



Name	Value
clock	0
rx_data_valid	0
data_in_reg[7:0]	11001111
data_in[3:0]	1100
data_out[7:0]	11001111
data_out_valid	0

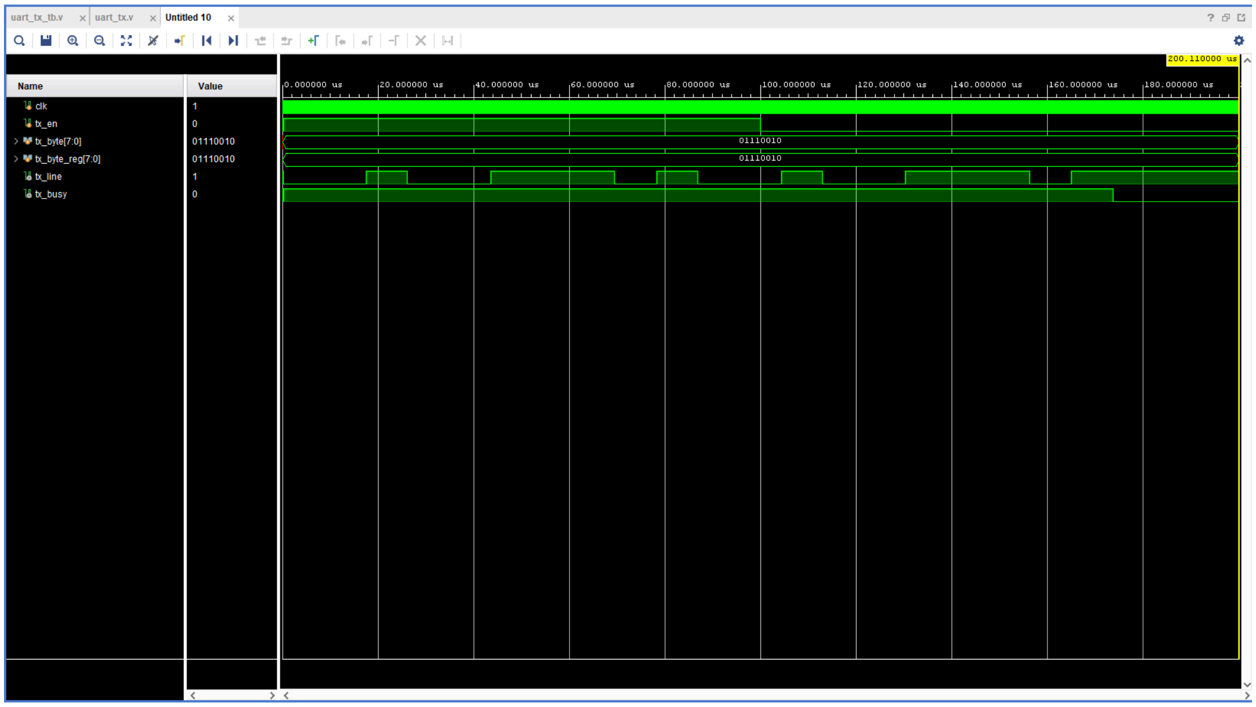
7. Parser.v :

Parser basically opens the frame, checks for all the necessary things like, mac addresses, fcs etc.

SIMULATION : WE WILL TRY TO SIMULATE IT ONCE BEFORE IMPLEMENTING IT ON FPGA AS IT REQUIRES READY MADE FRAMES.

8. uart_tx.v :

SIMULATION :



9. Exec_nic.v : INTEGRATES ALL THE FILES.

SOURCE_FILES :
https://drive.google.com/drive/folders/1mOsn70LnNKEVmUzS13TWT4MY-O1UROQJ?usp=sharing

EXECUTION FILE WILL BE UPLOADED ONCE DONE.

(REFERENCES :

Geeksforgeeks : https://www.geeksforgeeks.org/computer-networks/nic-full-form/

Youtube links :
https://youtube.com/playlist?list=PLXHMvqUANAFOviU0J8HSp0E91lLJInzX1&si=ACHgfKLMTzbaM4i5

https://youtu.be/hWXESYhHTMo?si=GR9FWRSKX_UIoxjP

https://youtu.be/niN94QR5v0Y?si=m1N6vE7HmozuozX7

https://youtu.be/IPvYjXCsTg8?si=Ag9XAP_MiU-ngllf

https://youtu.be/YbVbzduicy0?si=lSulX2V5d0RcAdqu

https://youtu.be/n6lPuep66aM?si=LQ8dzSDRSx3J2HIy

https://youtu.be/wn_P9n4El3k?si=Bfi8T3KL0WE7O_Tt

https://youtu.be/lsky_bgZoKA?si=5oiPESDlRj7fJNkW

https://youtu.be/YbVbzduicy0?si=wcskDMdz9tKdRmUD

https://youtu.be/78tkdc6Lq_8?si=VYIvjitYqVdH1my3

https://youtu.be/A9g6rTMblz4?si=IRJIrAUncan8Oa4x

gemini, chatGPT, wikipedia, and many more).

THANK YOU