USB 3.0 PHYSICAL LAYER IMPLEMENTATION

CHAPTER 1

UNIVERSAL SERIAL BUS

USB is a serial bus standard used to interface devices to a host computer. It's faster access to data, plug and play capabilities by hot swapping without rebooting the computer has made it a popular interface among various devices. USB 3.0 is the next upcoming version of the USB with a SuperSpeed data rate (5 Gbps), backward compatibility and a host of enhanced features above the earlier versions of the USB. This would enable the USB to be a part of devices used for signal processing and other applications where faster data access is a challenging requirement.

The original motivation for the Universal Serial Bus (USB) came from several considerations, two of the most important being:

- **EASE OF USE**

  The lack of flexibility in reconfiguring the PC had been acknowledged as an hindrance to its further deployment. The combination of user-friendly graphical interfaces and the hardware and software mechanisms associated with new-generation bus architectures have made computers less confrontational and easier to reconfigure. However, from the end user's point of view, the PC's I/O interfaces, such as serial/parallel ports, keyboard/mouse/joystick interfaces, etc., did not have the attributes of plug-and-play.

- **PORT EXPANSION**

  The addition of external peripherals continued to be constrained by port availability. The lack of a bidirectional, low-cost, low-to-mid speed peripheral bus held back the creative proliferation of peripherals such as storage devices, answering machines, scanners, PDA's, keyboards, and mice. Existing interconnects were optimized for one or two point products. As each new function or capability was added to the PC, a new interface had been defined to address this need.

  Initially, USB provided two speeds (12 Mb/s and 1.5 Mb/s) that peripherals could use. As PCs became increasingly powerful and able to process larger amounts of data, users needed to get more and more data into and out of their PCs. This led to the definition of the USB 2.0 specification in 2000 to provide a third transfer rate of 480 Mb/s while retaining backward compatibility. In 2005, with wireless technologies becoming more and more capable, Wireless USB was introduced to provide a new cable free capability to USB.

USB is the most successful PC peripheral interconnect ever defined and it has migrated heavily into the Consumer Electronics and Mobile segments. In 2006 alone over 2 billion USB devices were shipped and there are over 6 billion USB products in the installed base today. End users "know" what USB is. Product developers understand the infrastructure and interfaces necessary to build a successful product.

USB has gone beyond just being a way to connect peripherals to PCs. Printers use USB to interface directly to cameras. PDAs use USB connected keyboards and mice. The USB On-The-Go definition provides a way for two dual role capable devices to be connected and negotiate which one will operate as the "host." USB, as a protocol, is also being picked up and used in many nontraditional applications such as industrial automation.

Now, as technology innovation marches forward, new kinds of devices, media formats, and large inexpensive storage types are converging. They require significantly more bus bandwidth to maintain the interactive experience users have come to expect. HD Camcorders will have tens of gigabytes of storage that the user will want to move to their PC for editing, viewing, and archiving. Furthermore existing devices like still image cameras continue to evolve and are increasing their storage capacity to hold even more uncompressed images. Downloading hundreds or even thousands of 10 MB, or larger, raw images from a digital camera will be a time consuming process unless the transfer rate is increased. In addition, user applications demand a higher performance connection between the PC and these increasingly sophisticated peripherals. USB 3.0 addresses this need by adding an even higher transfer rate to match these new usages and devices.

Thus, USB (wired or wireless) continues to be the answer to connectivity for PC, Consumer Electronics, and Mobile architectures. It is a fast, bidirectional, low-cost, dynamically attachable interface that is consistent with the requirements of the PC platforms of today and tomorrow.

CHAPTER 2

OVERVIEW OF USB 3.0

This chapter presents an overview of Universal Serial Bus 3.0. USB 3.0 is similar to earlier versions of USB in that it is a cable bus supporting data exchange between a host computer and a wide range of simultaneously accessible peripherals. The attached peripherals share bandwidth through a host-scheduled protocol. The bus allows peripherals to be attached, configured, used, and detached while the host and other peripherals are in operation. USB 3.0 utilizes a dual-bus architecture that provides backward compatibility with USB 2.0. It provides for simultaneous operation of SuperSpeed and non-SuperSpeed (USB 2.0 speeds) information exchanges.

2.1 DUAL BUS ARCHITECTURE

USB 3.0 is a physical Super Speed bus combined in parallel with a physical USB 2.0 bus. This Dual bus architecture is shown in the figure given below.

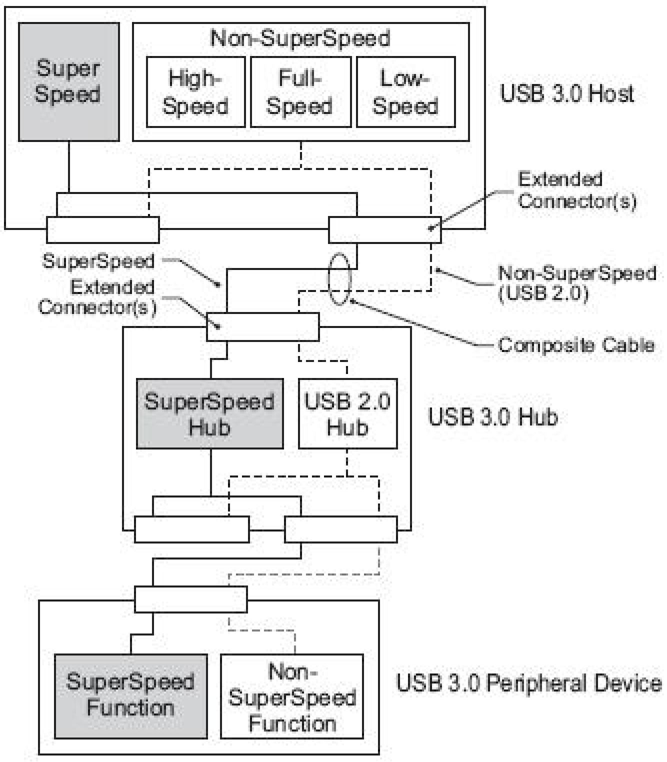


Figure 2.1: Dual bus architecture

It has similar architectural components as USB 2.0, namely:

- USB 3.0 interconnect
- USB 3.0 host
- USB 3.0 hub
- USB 3.0 devices

2.1.1 USB 3.0 INTERCONNECT

The USB 3.0 interconnect is the manner in which USB 3.0 and USB 2.0 devices connect to and communicate with the USB 3.0 host. The USB 3.0 interconnect inherits core architectural elements from USB 2.0, although several are augmented to accommodate the dual bus architecture. The baseline structural topology is the same as USB 2.0. It consists of a tiered star topology with a single host at tier 1 and hubs at lower tiers to provide bus connectivity to devices.

2.1.2 USB 3.0 HOST

A USB 3.0 host interacts with USB devices through a host controller. To support the dual-bus architecture of USB 3.0, a host controller must include both SuperSpeed and USB 2.0 elements, which can simultaneously manage control, status and information exchanges between the host and devices over each bus.

The host includes an implementation-specific number of root downstream ports for SuperSpeed and USB 2.0. Through these ports the host:

- Detects the attachment and removal of USB devices
- Manages control flow between the host and USB devices
- Manages data flow between the host and USB devices
- Collects status and activity statistics
- Provides power to attached USB devices

USB System Software inherits its architectural requirements from USB 2.0, including:

- Device enumeration and configuration
- Scheduling of periodic and asynchronous data transfers
- Device and function power management
- Device and bus management information

2.1.3 USB 3.0 HUB

USB 3.0 hubs are a specific class of USB device whose purpose is to provide additional connection points to the bus beyond those provided by the host. Hubs have always been a key element in the plug-and-play architecture of the USB. Hosts provide an implementation-specific number of downstream ports to which devices can be attached. Hubs provide additional downstream ports so they provide users with a simple connectivity expansion mechanism for the attachment of additional devices to the USB. In order to support the dual-bus architecture of USB 3.0, a USB 3.0 hub is the logical combination of two hubs: a USB 2.0 hub and a SuperSpeed hub (hub in Figure 2.1). The USB 2.0 hub unit is connected to the USB 2.0 data lines and the SuperSpeed hub is connected to the SuperSpeed data lines. A USB 3.0 hub connects upstream as two devices; a SuperSpeed hub on the SuperSpeed bus and a USB 2.0 hub on the USB 2.0 bus. The SuperSpeed hub manages the SuperSpeed portions of the downstream ports and the USB 2.0 hub manages the USB 2.0 portions of the downstream ports.

Hubs detect device attach, removal, and remote-wake events on downstream ports and enable the distribution of power to downstream devices. A SuperSpeed hub consists of two logical components: a SuperSpeed hub controller and a SuperSpeed repeater/forwarder. The hub repeater/forwarder is a protocol-controlled router between the SuperSpeed upstream port and downstream ports. It also has hardware support for reset and suspend/resume signaling. The SuperSpeed controller responds to standard, hub-specific status/control commands that are used by a host to configure the hub and to monitor and control its ports.

SuperSpeed hubs actively participate in the (end-to-end) protocol in several ways, including:

- Routes out-bound packets to explicit downstream ports.

- Aggregates in-bound packets to the upstream port.
- Propagates the timestamp packet to all downstream ports not in a low-power state.
- Detects when packets encounter a port that is in a low-power state. The hub transitions the targeted port out of the low-power state and notifies the host and device (in-band) that the packet encountered a port in a low-power state.

2.1.4 USB 3.0 DEVICES

All SuperSpeed devices share their base architecture with USB 2.0. They are required to carry information for self-identification and generic configuration. They are also required to demonstrate behavior consistent with the defined SuperSpeed Device States. All devices are assigned a USB address when enumerated by the host. Each device supports one or more pipes through which the host may communicate with the device. All devices must support a designated pipe at endpoint zero to which the device's Default Control Pipe is attached. All devices support a common access mechanism for accessing information through this control pipe.

A USB 3.0 device must provide support for both SuperSpeed and at least one other non-SuperSpeed speed. The minimal requirement for non-SuperSpeed is for a device to be detected on a USB 2.0 host and allow system software to direct the user to attach the device to a SuperSpeed capable port. A device implementation may provide appropriate full functionality when operating in non-SuperSpeed mode. Simultaneous operation of SuperSpeed and non-SuperSpeed modes is not allowed for peripheral devices. USB 3.0 devices within a single physical package (i.e., a single peripheral) can consist of a number of functional topologies including single function, multiple functions on a single peripheral device (composite device), and permanently attached peripheral devices behind an integrated hub(compound device).

2.1.5 USB 3.0 CONNECTION MODEL

The USB 3.0 connection model accommodates backwards and forward compatibility for connecting USB 3.0 or USB 2.0 devices into a USB 3.0 bus. Similarly, USB 3.0 devices can be attached to a USB 2.0 bus. The mechanical and electrical backward/forwards compatibility for USB 3.0 is accomplished via a composite cable and associated connector assemblies that form the dual-bus architecture. USB 3.0 devices accomplish backward compatibility by including both SuperSpeed and non-SuperSpeed bus interfaces. USB 3.0 hosts also include both SuperSpeed and non-SuperSpeed bus interfaces, which are essentially parallel buses that may be active simultaneously.

The USB 3.0 connection model allows for the discovery and configuration of USB devices at the highest signaling speed supported by the device, the highest signaling speed supported by all hubs between the host and device, and the current host capability and configuration.

2.2 COMPARISION BETWEEN USB 2.0 AND USB 3.0

USB 3.0 is a dual-bus architecture that incorporates USB 2.0 and a SuperSpeed bus. The table given below summarizes the key architectural differences between USB 3.0 and USB 2.0.

Characteristic	**Superspeed USB**	**USB2.0**
Data Rate	Superspeed(5.0 Gbps)	Low-speed(1.5Mbps),full-speed(12Mbps),and high-speed(480 Mbps)
Data interface	Dual simplex, four wire differential signaling separate from USB2.0 signaling simultaneous bi-directional data flows	Half-duplex two wire differential signaling Unidirectional data flow with negotiated directional bus transitions
Cable signal count	Six: four for super-speed data path and two for non-super speed data path	Two: Two for low-speed/full-speed/high-speed data path
Bus transaction protocol	Host directed asynchronous traffic flow. Packet traffic is	Host directed, polled traffic flow

	explicitly routed	Packet traffic is broadcast to all devices
Power management	Multi-level link power management supporting idle, sleep, and suspend states. link-device- and function-level power management	Port-level suspend with two levels of entry/exit latency Device-level power management
Bus power	Same as for USB2.0 with a 50% increase for unconfigured power and 80% increase for configured power	Support for low/high bus power devices with lower power limits for un-configured and suspended devices.
Port state	Port hardware detects connect events and brings the port into operational state ready for super speed data communication	Port hardware detects connect events. system software uses port commands to transition the port into an enabled state(i.e., can do USB data communication flows
Data transfer types	USB2.0 types with super speed constraints. Bulk has streams capability	Four data transfer types: control, bulk and interrupt and isochronous.

2.3 SUPERSPEED ARCHITECTURE

The SuperSpeed bus is a layered communications architecture that is comprised of the following elements:

• **SuperSpeed Interconnect:** The SuperSpeed interconnect is the manner in which devices are connected to and communicate with the host over the SuperSpeed bus. This includes the topology of devices connected to the bus, the communications layers, the relationships between them and how they interact to accomplish information exchanges between the host and devices.

• **Devices.** SuperSpeed devices are sources or sinks of information exchanges. They implement the required device-end, SuperSpeed communications layers to accomplish information exchanges between a driver on the host and a logical function on the device.

• **Host.** A SuperSpeed host is a source or sink of information. It implements the required host-end, SuperSpeed communications layers to accomplish information exchanges over the bus. It owns the SuperSpeed data activity schedule and management of the SuperSpeed bus and all devices connected to it.

Figure 2.2 illustrates a reference diagram of the SuperSpeed interconnect represented as communications layers through a topology of host, zero to five levels of hubs, and devices.

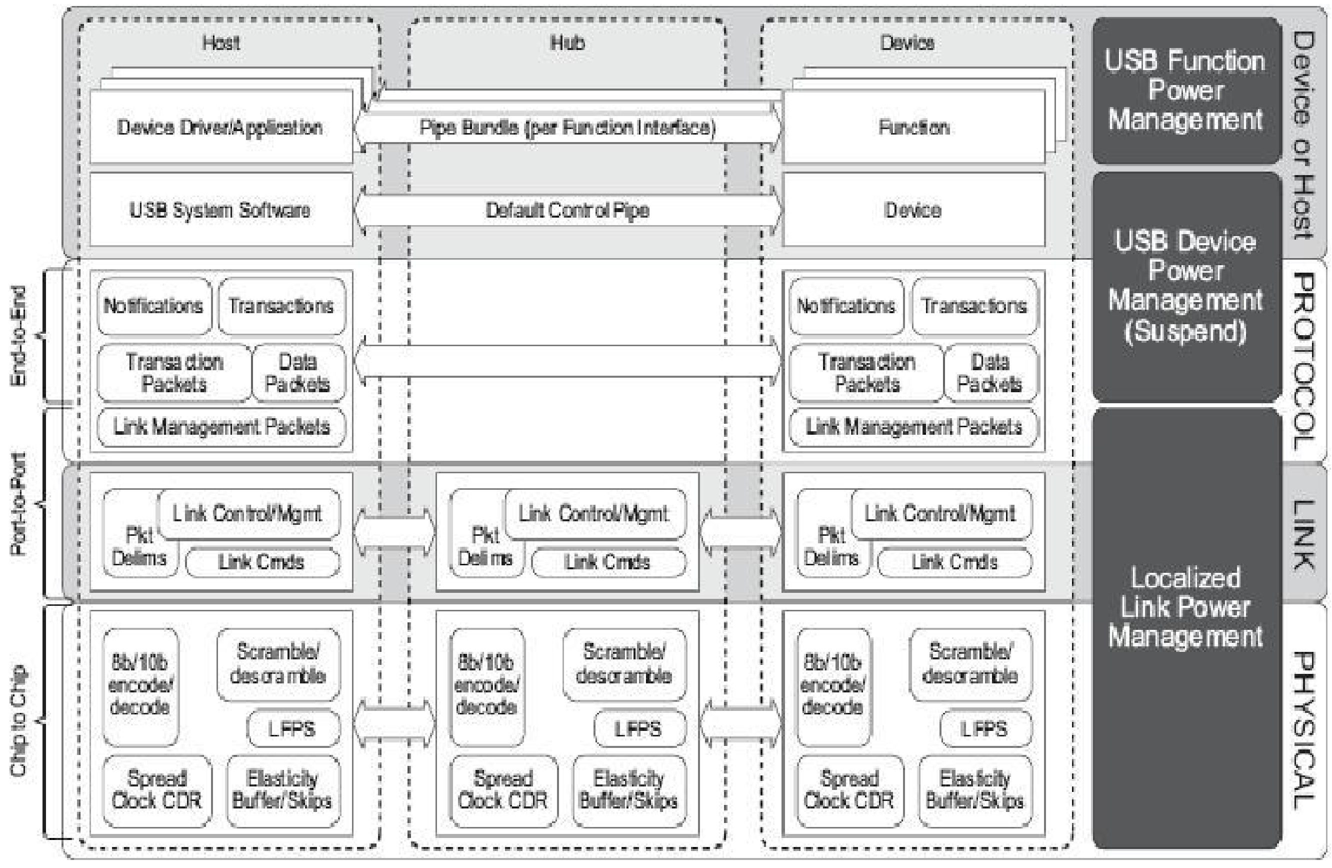


Figure 2.2: SuperSpeed bus communication layers

The columns (device or host, protocol, link, physical) realize the communications layers of the SuperSpeed interconnect. The three, rows (host, hub, and device) illustrate the topological relationships between devices connected to the SuperSpeed bus. The right-most column illustrates the influence of power management mechanisms over the communications layers. A brief overview of the three communication layers is given below.

2.3.1 PHYSICAL LAYER

The physical layer is the subject of this project and is explained in detail in the next chapter.

2.3.2 LINK LAYER

A SuperSpeed link is a logical and physical connection of two ports. The connected ports are called link partners. A port has a physical part and a logical part. The physical layer defines the physical portion of a port and the link layer defines the logical portion of a port and the communications between link partners.

The logical portion of a port has:

• State machines for managing its end of the physical connection. These include physical layer initialization and event management, i.e., connect, removal, and power management.

• State machines and buffering for managing information exchanges with the link partner. It implements protocols for flow control, reliable delivery (port to port) of packet headers, and link power management.

• Buffering for data and protocol layer information elements.

The logical portion of a port also:

• Provides correct framing of sequences of bytes into packets during transmission; e.g., insertion of packet delimiters

• Detects received packets, including packet delimiters and error checks of received header packets (for reliable delivery).

• Provides an appropriate interface to the protocol layer for pass-through of protocol-layer packet information exchanges.

The physical layer provides the logical port an interface through which it is able to:

• Manage the state of its PHY (i.e., its end of the physical connection), including power management and events (connection, removal, and wake).

• Transmit and receive byte streams, with additional signals that qualify the byte stream as control sequences or data. The physical layer includes discrete transmit and receive physical links, therefore, a port is able to simultaneously transmit and receive control and data information.

2.3.3 PROTOCOL LAYER

This protocol layer defines the “end-to-end” communications rules between a host and device. The SuperSpeed protocol provides for application data information exchanges between a host and a device. This communications relationship is called a pipe. It is a host-directed protocol, which means the host determines when application data is transferred between the host and device. SuperSpeed is not a polled protocol, as a device is able to asynchronously request service from the host on behalf of a particular endpoint.

All protocol layer communications are accomplished via the exchange of packets. Packets are sequences of data bytes with specific control sequences which serve as delimiters managed by the link layer. Host transmitted protocol packets are routed through intervening hubs directly to a peripheral device. They do not traverse bus paths that are not part of the direct path between the host and the target peripheral device. A peripheral device expects it has been targeted by any protocol layer packet it receives. Device transmitted protocol packets simply flow upstream through hubs to the host.

Packet headers are the building block of the protocol layer. They are fixed size packets with type and subtype field encodings for specific purposes. A small record within a packet header is utilized by the link layer (port-to-port) to manage the flow of the packet from port to port. Packet headers are delivered through the link layer (port-to-port) reliably. The remaining fields are utilized by the end-to-end protocol.

Application data is transmitted within data packet payloads. Data packet payloads are preceded (in the protocol) by a specifically encoded data packet headers. Data packet payloads are not delivered reliably through the link layer (however, the accompanying data packet headers are delivered reliably). The protocol layer supports reliable delivery of data packets via explicit acknowledgement (header) packets and retransmission of lost or corrupt data. Not all data information exchanges utilize data acknowledgements.

Data may be transmitted in bursts of back-to-back sequences of data packets (depending on the scheduling by the host). The protocol allows efficient bus utilization by concurrently transmitting and receiving over the link. For example, a transmitter (host or device) can burst multiple packets of data back-to-back while the receiver can transmit data acknowledgements without interrupting the burst of data packets. The number of data packets in a specific burst is scheduled by the host.

Furthermore, a host may simultaneously schedule multiple OUT bursts to be active at the same time as an IN burst. The protocol provides flow control support for some transfer types. A device-initiated flow control is signaled by a device via a defined protocol packet. A host-initiated flow control event is realized via the host schedule (host will simply not schedule information flows for a pipe unless it has data or buffering available). On reception of a flow control event, the host will remove the pipe from its schedule. Resumption of scheduling information flows for a pipe may be initiated by the host or device. A device endpoint will notify a host of its readiness (to source or sink data) via an asynchronously transmitted “ready” packet. On reception of the “ready” notification, the host will add the pipe to its schedule, assuming that it still has data or buffering available.

Independent information streams can be explicitly delineated and multiplexed on the bulk transfer type. This means through a single pipe instance, more than one data stream can be tagged by the source and identified by the sink. The protocol provides for the device to direct which data stream is active on the pipe.

Devices may asynchronously transmit notifications to the host. These notifications are used to convey a change in the device or function state. A host transmits a special packet header to the bus that includes the host's timestamp. The value in this packet is used to keep devices (that need to) in synchronization with the host. In contrast to other packet types, the timestamp packet is forwarded down all paths not in a low power state. The timestamp packet transmission is scheduled by the host at a specification determined period.

CHAPTER 3

PHYSICAL LAYER

The physical layer defines the PHY portion of a port and the physical connection between a downstream facing port (on a host or hub) and the upstream facing port on a device. The SuperSpeed physical connection is comprised of two differential data pairs, one transmit path and one receive path. The nominal signaling data rate is 5 Gbps. The electrical aspects of each path are characterized as a transmitter, channel, and receiver; these collectively represent a unidirectional differential link. The channel includes the electrical characteristics of the cables and connectors. The figure below shows the transmitter and the receiver.

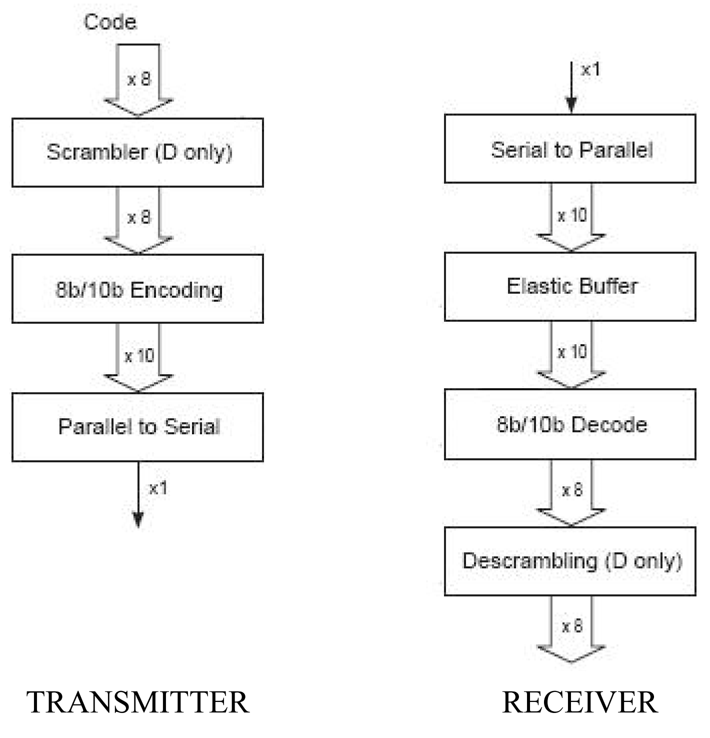


Figure 3.1: USB 3.0 transmitter and receiver

Each of the blocks in the above figure is explained in detail in the subsequent chapters.

At the electrical level, each differential link is initialized by enabling its receiver termination. The transmitter is responsible for detecting the far end receiver termination as an indication of a bus connection and informing the link layer so the connect status can be factored into link operation and management. When receiver termination is present but no signaling is occurring on the differential link, it is considered to be in the electrical idle state. When in this state, low frequency periodic signaling (LFPS) is used to signal initialization and power management information. The LFPS is relatively simple to generate and detect and uses very little power.

Each PHY has its own clock domain. The USB 3.0 cable does not include a reference clock so the clock domains on each end of the physical connection are not explicitly connected. Bit-level timing synchronization relies on the local receiver aligning its bit recovery clock to the remote transmitter's clock by phase-locking to the signal transitions in the received bit stream. The receiver needs enough transitions to reliably recover clock and data from the bit stream.

To assure that adequate transitions occur in the bit stream independent of the data content being transmitted, the transmitter encodes data and control characters into symbols using an 8b/10b code. Control symbols are used to achieve byte alignment and are used for framing data and managing the link. Special characteristics make control symbols uniquely identifiable from data symbols. A number of techniques are employed to improve channel performance. For example, to avoid overdriving and improve eye margin at the receiver, transmitter de-emphasis may be applied when multiple bits of the same polarity are sent. Also, equalization may be used in the receiver with the characteristics of the equalization profile being established adaptively as part of link training. Signal (timing, jitter tolerance, etc.) and electrical (DC characteristics, channel capacitance, etc.) performance of SuperSpeed links are defined with compliance requirements specified in terms of transmit and receive signaling eyes.

The physical layer receives 8-bit data from the link layer and scrambles the data to reduce EMI emissions. It then encodes the scrambled 8-bit data into 10-bit symbols. This parallel data is converted to serial data by the serializer for transmission over the physical connection. The bit stream is recovered from the differential link by the receiver, assembled into 10-bit symbols by the deserializer, stored in the elastic buffer, decoded and descrambled, producing 8-bit data that are then sent to the link layer for further processing.

The complete design of the physical layer is covered in the following chapters.

CHAPTER 4

SCRAMBLER

4.1 Introduction to Scrambler

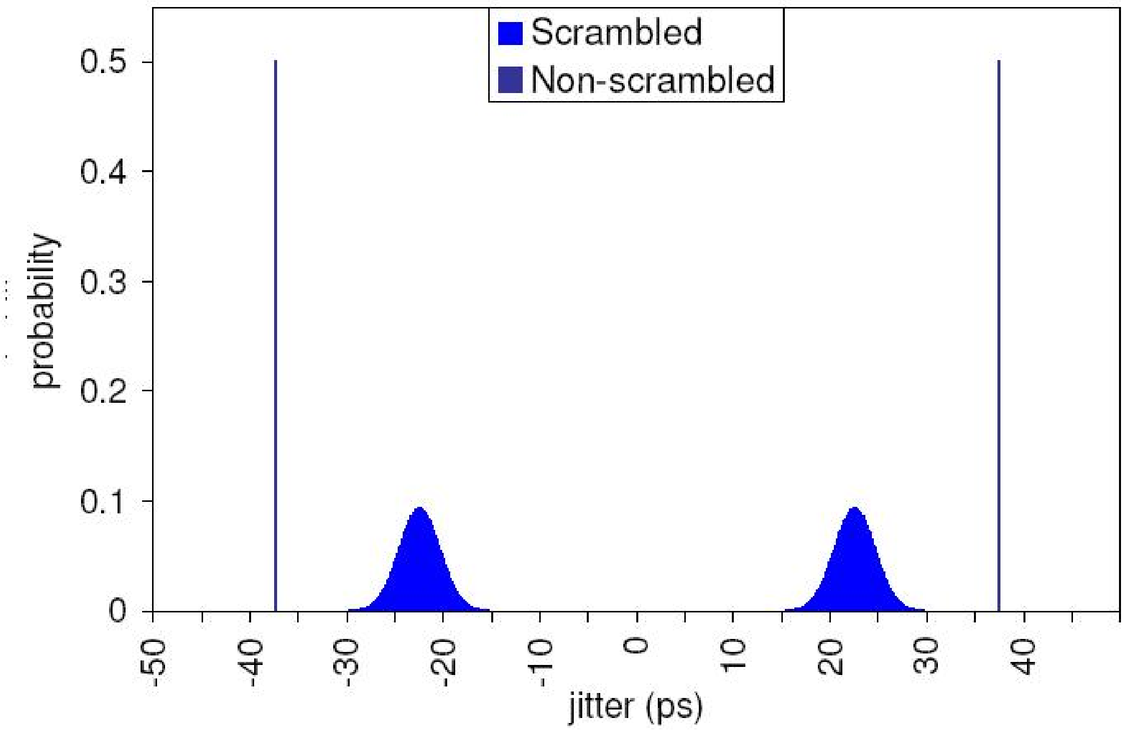
A scrambler is a device which manipulates the input data before transmitting so that the transmitted sequence becomes unintelligible. At the receiver point, the original data cannot be recovered unless there is a corresponding descrambling device. The manipulated data is a random sequence. This is achieved by applying a pseudo random sequence to the input data. Although Scrambling idea looks to be a form of encryption, in practice it is not so. It can be called as a **non-secret encryption** since the cyclic generator polynomial (Pseudo random sequence generator) is publicly declared or well known to the users at the receiving end. However in the days of the Second World War, Scramblers were used for information security. Today Scramblers are used for any of the following purposes:

- It aids the timing recovery circuit at the receiver by eliminating long strings of 1's and 0's.
- It makes the transmitted data stream to be more dispersed thus enabling to meet maximum power spectral density requirements.
- It eliminates a high frequency of bit transitions in binary streams which causes Electromagnetic Interference (EMI) noise.

Scrambler is used in the field of Telecommunications particularly for satellite, radio and PSTN systems and a host of other electronic devices. A scrambler does not eliminate any of the sequences be it any undesirable sequence but replaces each of the sequences with other sequences. Hence it changes the probability of occurrence of vexatious sequences.

4.2 Purpose of Scrambler in USB 3.0

Apart from the above stated uses, scrambler in USB 3.0 is used for reducing the probability of the channel jitter



4.3 Scrambler- A detailed description

A scrambler consists of a Pseudo random generator in the form of a Linear Feedback Shift Register (LFSR). The incoming data stream is added (modulo 2) or multiplied with the LFSR sequence. In the former case, it is called as an **Additive Scrambler** and in the latter case it is called as a **Multiplicative scrambler**. In USB 3.0, an additive scrambler is used. The operation of this scrambled will be discussed in the forthcoming sections.

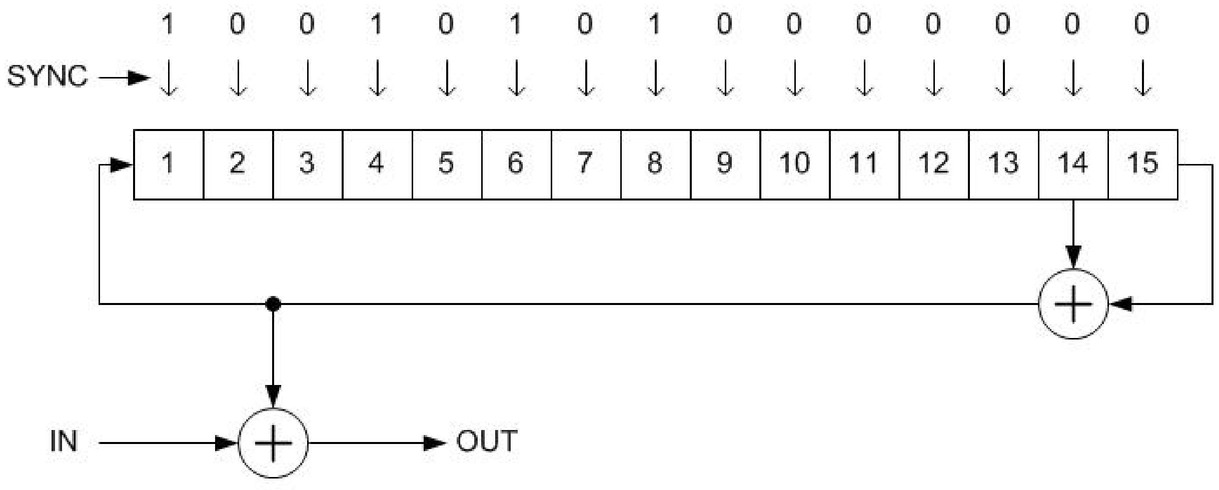


Fig 4.1: An Additive Scrambler

A sync-word is used in the input data stream to achieve a synchronized operation with the scrambler and the descrambler. At the occurrence of a sync-word, both the LFSR's of scrambler and descrambler reset their values to the initialized condition and once again regenerate their sequences. The addition operation is performed as an XOR operation. Each bit of the input data is XORed with the corresponding LFSR bits. Hence this operation can be performed serially as well as parallel. Accordingly, scramblers are classified as serial or parallel scramblers. In order to achieve a synchronous operation between the LFSR and the input data byte i.e. to take care that a given polynomial is XORed with a particular data and to successfully repeat this action on the descrambler, both the LFSR and data bite are implemented in registers connected to the same clock.

4.3.1 Linear Feedback Shift Register (LFSR)

An LFSR is a shift register whose input bit is a linear function of its previous state. The linear function here is either XOR or XNOR function. It is used to continuously generate a random sequence. Since the operation of the register is deterministic, its output can be determined from the previous state. The LFSR is constructed using a **generator polynomial** with the number of states (2^n-1) being determined by the degree of the polynomial. Each sequence arising out of the LFSR is random in nature i.e. the bits are simply 1's and 0's placed randomly and they do not follow any pattern between them. This meets the needs of scrambling the data. An example of an LFSR is explained in the next section by taking the USB 3.0 scrambler.

4.4USB 3.0 Scrambler

The Scrambling function is implemented prior to 8b/10b encoding on the transmitter side and on the receiving side, it is implemented after the 10b/8b decoding function. The input data is of byte length and the LFSR is a 16-bit polynomial.
As shown in the Fig 4.2, output bits of the LFSR (D0-D15) are serially XORed with the input data byte (D0-D7) to give the scrambling output. At each clock cycle, bit D15 of the LFSR is XORed with data bit D0. Bits are then advanced through D1-D7 (data register) and D8-D15 (LFSR) in the next clock instants. To perform descrambling, same functions must be implemented on the receiver side.

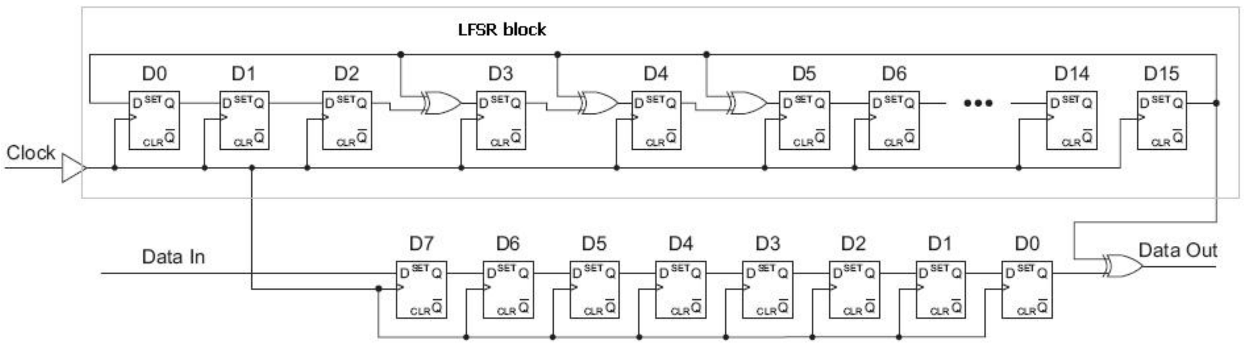


Fig 4.2: LFSR with Scrambling Polynomial

4.4.1 USB 3.0 Scrambler Specifications

The following are the specifications of the USB 3.0 scrambler to be implemented:

1. Generator polynomial is $X^{16}+X^{5}+X^{4}+X^{3}+1$.
   i) no. of states: $2^{n}-1=65535$.
2. Input data=1 byte.
3. LFSR=16 bit.
4. Sync word is COM (K28.5)
5. The initialized value of the LFSR is FFFFH. Every time COM leaves the Transmitter LFSR shall be reset to FFFFH and every time COM enters the receiver, LFSR shall be reset to FFFFH.
6. K codes shall not be scrambled.
7. All D-codes except those within the Training sequence sets shall be scrambled.

4.4.2 Circuit description of USB 3.0 Scrambler

The above figure shows the USB 3.0 scrambler specifications as released by the USB-IF. It consists of two parts as described earlier, an input data stream register (bottom) and the LFSR (top). The above shown scrambler is a serial scrambler. The input data is given serially at the rate of 1 bit per clock cycle. Also, the bits shift right once for every clock cycle. The bit positions that have their outputs affected by the previous state are called the taps. In the diagram the taps are [16, 5, 4, 3, and 1]. The rightmost bit of the LFSR is called the output bit. The taps are XORed sequentially with the output bit and then fed back into the leftmost bit. The sequence of bits in the rightmost position is called the output stream (D8-D15). When the system is clocked, bits that are not taps are shifted one position to the right unchanged. The taps, on the other hand, are XORed with the output bit before they are stored in the next position. The new output bit is the next input bit. The effect of this is that when the output bit is zero all the bits in the register shift to the right unchanged, and the input bit becomes zero. When the output bit is one, the bits in the tap positions all flip (if they are 0, they become 1, and if they are 1, they become 0), and then the entire register is shifted to the right and the input bit becomes 1.

The above LFSR used is a Galois type LFSR. Galois LFSRs do not concatenate every tap to produce the new input (the XOR'ing is done within the LFSR and no XOR gates are run in serial, therefore the propagation times are reduced to that of one XOR rather than a whole chain), thus it is possible for each tap to be computed in parallel, increasing the speed of execution.

4.4.3 Design & implementation

Implementation of a serial scrambler for byte operations consumes 8 clock cycles for scrambling each byte. Since speed is the main criteria in USB 3.0 operations, we go in for a parallel implementation of the scrambler.

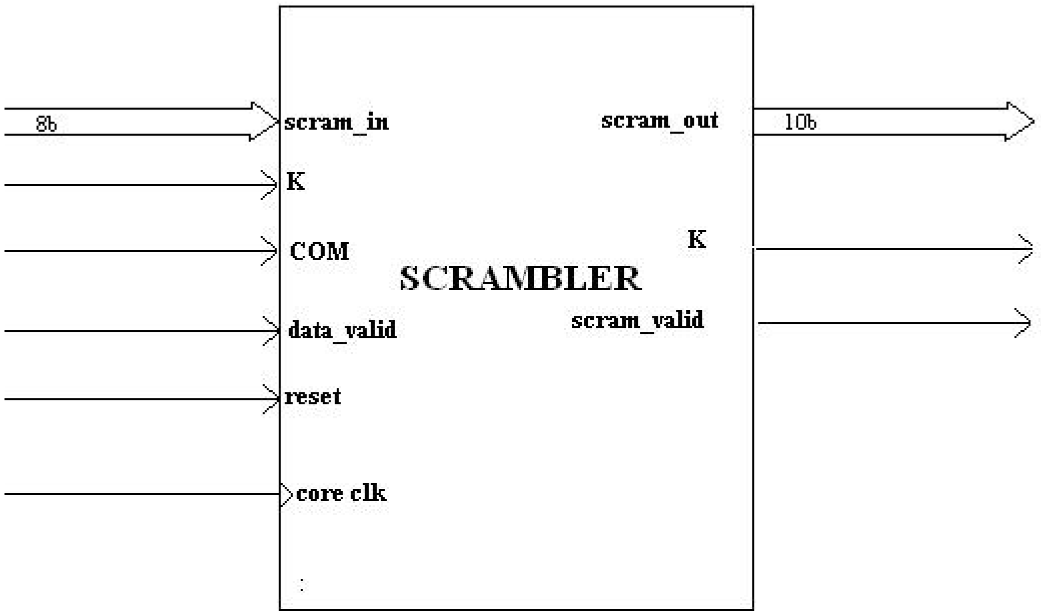


Fig 4.3:Top Level view

The Fig 4.3 shows the top level view of a scrambler. The definitions of the input's and output's are as follows:

Scram_in: 8-bit input to the scrambler.

Core_Clk: LFSR outputs its data at every clock cycle.

Reset: negedge reset. Output of the scrambler is 00000000 when reset is low.

Data_valid: When set to high indicates that the given input data is a valid one. Scrambler's Output is only considered if the valid signal is high.

COM: sync word indicating start of a new frame.

K, BRST: control codes which are bypassed. They are also propagated to the next in line modules.

Scram_out: 8-bit scrambled output.

Scram_valid: is a signal that goes high for the valid output of the scrambler. This signal is given to the encoder as a valid signal.

4.4.4 Design using Logic elements

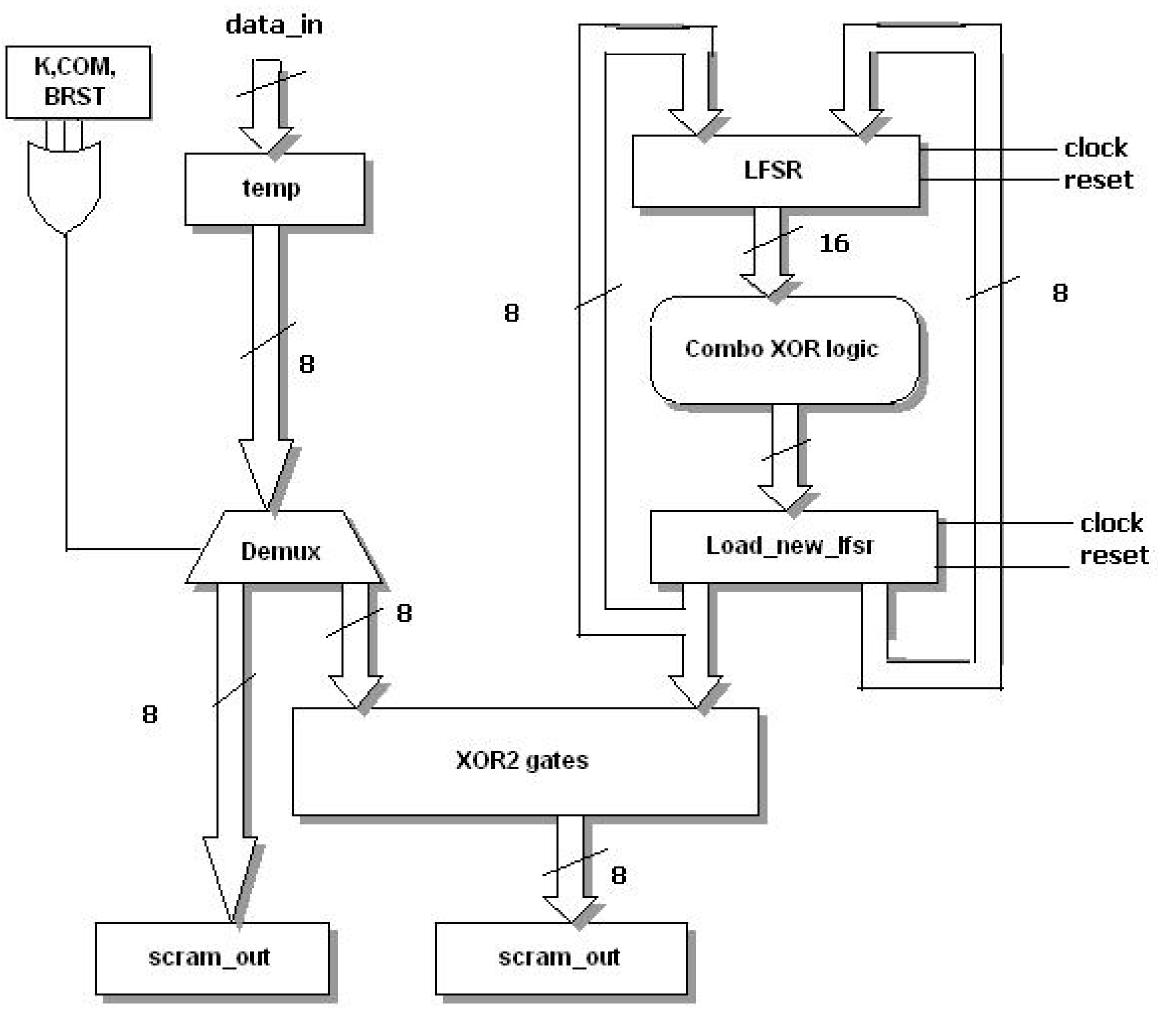


Fig 4.4:Design of a Scrambler using Logic Elements

Figure shows the design of a scrambler using logic elements like registers, flip-flops and demultiplexer. Clearly, the design can be viewed as implementing two different functionalities. The left hand side of the design is concerned with the handling of the control information (k codes) and is a combinatorial logic whereas the right side part implements the scrambling operation (Synchronous operation). Both entities execute concurrently.

At the rising edge of the clock, the initialized value is clocked into the LFSR flop. At the end of the clock cycle, LFSR output is generated. This generated output is given as a feedback again to the LFSR only in the next clock cycle. Here, we are ensuring that by generating one LFSR output in each clock cycle, the operation of scrambler is cycle accurate.

The incoming data is placed in a register *temp.* A demultiplexer is used to detect whether the given input is a control signal (k=1) or data. If the input is data, it is XORed with the generated LFSR output (*load_new_lfsr*) to produce the scrambled output. Otherwise, if a control signal, it is directly passed to the output register *scram_out* and it does not get scrambled.

Chapter 5

DESCRAMBLER

Descrambler is the same as the scrambler except it is used at the receiver side. While descrambling at the receiving end, it has to be ensured that the LFSR polynomial which is used to XOR the input data is only used for the corresponding scrambled data.

5.1 Summary

Cycle time of operation: 1 core clock cycle.

Using parallel scrambler has reduced the cycle consumption from 8 clocks to 1 clock cycle.

LFSR simulation results have been provided in appendix.

CHAPTER 6

ENCODER

The 8B/10B encoding scheme is used in the physical layer of a number of current and emerging technologies such as Gigabit Ethernet, Rapid I/O, USB, PCI express, etc. This code is particularly well suited for high-speed local area networks and similar data links, where the information format consists of packets, variable in length, from about a dozen up to several hundred 8-bit bytes. The encoding scheme encodes 8-bit bytes into 10-bit symbols using partitioned block encoding logic. The code has several advantages some of which are listed below.

- **DC balance:** A DC-balanced serial data stream means that it has the same number of 0's and 1's for a given length of data stream. A code that is free of dc, or one that has a constant DC component regardless of data patterns, provides many advantages for fiber optic and electromagnetic wire links. DC-balance is important for certain media as it avoids a charge being built up in the media. Since this code is DC-balanced it allows receivers to function at lower signal-to-noise ratios, which is specifically beneficial to the active gain of receivers.

- **Clock recovery:** The maximum numbers of contiguous 0's or 1's in the serial data stream is defined as the run-length. Smaller the run-length greater the data transitions within a specified length of data. Data transitions are essential for clock recovery. The greater the transition density, the more chances there are for the receiver to recover the clock properly. Many serial data transmission standards utilize 8B/10B encoding to ensure sufficient data transitions for clock recovery.

- **Error detection:** Error detection property is inherently built into the code. Concepts like disparity[1] and running disparity[2] are introduced for this purpose.

[1] Will be explained later

[2] Will be explained later

Disparity can be 0, +2 or -2. Any other value will be considered as a disparity error. Running disparity can be either positive or negative. Based on certain rules, the errors arising out of running disparity violation can also be detected.

- **Special characters:** In addition to the valid 256 data characters, this code provides an additional set of 12 characters called special characters. They are generally used to establish byte synchronization to mark the start and end of packets, and sometimes to signal control functions such as ABORT, RESET, SHUTOFF**,** IDLE, and for link diagnostics. A subset of three special characters called comma characters can be used to indicate the proper byte boundaries and for instantaneous acquisition or verification of byte synchronization. Comma characters are singular in the sense that the bit sequence occurring in comma characters are not present in any data or other special characters.

In order to achieve the above mentioned characteristics the 8B/10B encoding scheme introduces redundancy in a controlled manner to the 8-bit data. This makes the encoded data 10-bits wide. This is all done with a comparatively low overhead of 25 percent (each 10-bit symbol contains 8 bits of information) versus, for example, a Manchester code with its 100 percent overhead.

6.1 8B/10B ENCODING SCHEME

The 8B/10B encoding scheme is based on the research paper "A DC-balanced, partitioned-block, 8B/10B transmission code" [3] . In this paper several new concepts and terminologies have been explained. The details of which, are necessary to understand the 8B/10B encoding scheme completely and are given below.

[3] Will be explained later

[3] Will be explained later

[3] Copyright IBM; A.X. Widmer and P.A. Franaszek, A DC-BALANCED, PARTITIONED-BLOCK, 8B/10B TRANSMISSION CODE, *IBM Journal of Research and Development,* Volume 27, Number 5, September 1983.

6.1.1 8B/10B CODE MAPPING

The 8 bit input data to the encoder is labeled as ABCDEFGH with A as the LSB and H as the MSB. The 10 bit output data is labeled as abcdeifghj with a as the LSB and j as the MSB. The input data is divided into two blocks of 5 bits and 3 bits. The 5 bit block ABCDE is encoded into abcdei. This is known as 5B/6B block. The 3 bit block FGH is encoded into fghj. This is known as 3B/4B block. Encoding the 8 bit data by dividing it into two sub blocks of 5 bits and 3 bits is known as partitioned block encoding. These sub-blocks are referred to in the form *D.a.b*, for data blocks, or *K.a.b*, for special blocks. In both cases, 'a' represents the value of the 5 bit block to be encoded, and 'b' represents the value of the 3 bit block to be encoded. The D codes are the valid data characters which are 256 in number and the K codes are control characters which are 12 in number.

6.1.2 DISPARITY

The disparity of a code is the difference between the number of 1’s and 0’s in the code. Disparity can be 0, +2 or -2. For codes having a disparity of 0, the number of 0’s and 1’s are equal. For codes having a disparity of +2, the number of 1’s are greater than the number of 0’s.For codes having a disparity of -2, the number of 0’s are greater than the number of 1’s. If both the 4-bit and 6-bit blocks have 0 disparity, then the combined 10-bit encoded data will also have 0 disparity. This will create a DC-balanced code. However, this is not possible. Because only 6 out of the 16 possible values of the 4-bit block have 0 disparity, they are not enough for encoding the 8 values of the 3-bit block. Likewise, only 20 values of the 6-bit block have 0 disparity and they are not enough for encoding the 32 values of the 5-bit block. Because both the 4-bit and 6-bit blocks have even number of bits, the disparity cannot be +1 or -1. Therefore, the values with a disparity of +2 and -2 are also used in the 8b/10b coding scheme.

6.1.3 RUNNING DISPARITY

Running disparity is a record of the cumulative disparity of every encoded word, and is tracked by the encoder. The running disparity can be either positive (RD+) or negative (RD-). To guarantee neutral average disparity, a positive running disparity must be followed by neutral or negative disparity; a negative running disparity must be followed by neutral or positive disparity. The concept of running disparity is illustrated in the figure below (Figure 6.1).

As seen from the figure, if the current running disparity is positive, the data should be encoded in such a way that the code disparity should be either 0 or -2. If the code disparity is 0, then the next running disparity is same as the current running disparity. In this case it remains positive and the encoded value will be taken from RD+ column[4]. If it is -2, then the next running disparity will be negative and RD- column[4] will be used.

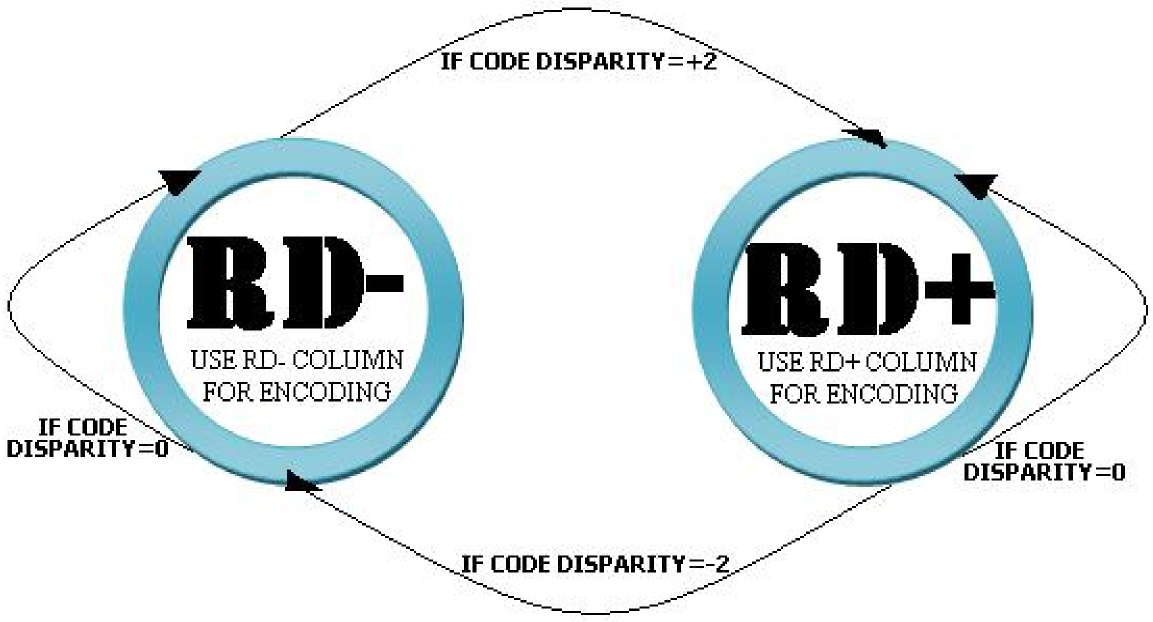


Figure 6.1: State diagram for determining running disparity

If the current running disparity is negative, the data should be encoded in such a way that the code disparity should be either 0 or +2. If the code disparity is 0, then the next running disparity remains negative and RD- column will be used. If it is +2, then the next running disparity will be positive and RD+ column will be used for encoding.

The disparity rules are given in the following table

[4] Refer appendix A of USB 3.0 specifications for RD+ and RD- columns

CURRENT RUNNING DISPARITY	CODE DISPARITY	NEXT RUNNING DISPARITY
-	0	-
-	+	+
+	0	+
+	-	-
-	-	Error/Invalid
+	+	Error/Invalid

Table 6.1: Running disparity rules

6.2 ENCODER DESIGN CONCEPT

The 8B/10B encoder is designed using portioned block logic, as two separate 5B/6B and 3B/4B encoders. The section below describes the 5B/6B and 3B/4B encoding logic.

6.2.1 5B/6B ENCODING LOGIC

The input to the 5B/6B encoder is 5-bit data ABCDE and output is 6-bit encoded data abcdei. The 5B/6B encoding is done according to the rules mentioned in Table 5.2.

The first column in Table 2**,** headed by **"NAME"**, gives the 32 decimal equivalents for the input lines ABCDE; assuming A is the low-order bit and E the high order bit. For regular data (D.x) the line K must be held at 0; a few code points can be part of special characters which are recognizable as other than data; such code points are named D/K.x or K.x and have an x or 1 in column K. To encode special characters the K line must be 1.

In the **"CLASSIFICATIONS"** columns, L04 means that there are no 1's but four 0's in ABCD; L13 means that there is one 1 and three 0's in ABCD, etc. The letter "L" indicates that this logic function or classification is part of the 5B/6B encoder. Analogous functions labeled "P" are defined for decoding.

NAME	**ABCDE K**	**CLASSIFICATIONS**		**D-1**	**abcdei**	**D0**	**abcdei ALTERNATE**
		BIT ENCODING	**DISPARITY**				
D.0	00000 0	L04	L22'.L31'.E'	+	011000	-	100111
D.1	10000 0	L13.E'	L22'.L31'.E'	+	100010	-	011101
D.2	01000 0	L13.E'	L22'.L31'.E'	+	010010	-	101101
D.3	11000 0	L22.E'		x	110001	0	
D.4	00100 0	L13.E'	L22'.L31'.E'	+	001010	-	110101
D.5	10100 0	L22.E'		x	101001	0	
D.6	01100 0	L22.E'		x	011001	0	
D.7	11100 0		L31.D'.E'	-	111000	0	000111
D.8	00010 0	L13.E'	L22'.L31'.E'	+	000110	-	111001
D.9	10010 0	L22.E'		x	100101	0	
D.10	01011 0	L22.E'		x	010101	0	
D.11	11010 0			x	110100	0	
D.12	00110 0	L22.E'		x	001101	0	

D.13	10110 0			x	101100	0	
D.14	01110 0			x	011100	0	
D.15	11110 0	L40	L22’.L31’.E’	+	1**0**1**0**00	-	010111
D.16	00001 0	L04,L04 .E	L22’.L13’.E	-	0**11**01**1**	+	100100
D.17	10001 0	L13.D’. E		x	10001**1**	0	
D.18	01001 0	L13.D’. E		x	01001**1**	0	
D.19	11001 0			x	110010	0	
D.20	00101 0	L13.D’. E		x	00101**1**	0	
D.21	10101 0			x	101010	0	
D.22	01101 0			x	011010	0	
D/K.23	11101 x		L22’.L13’.E	-	111010	+	000101
D.24	00011 0	L13.D.E	L13.D.E	+	00**1**1**0**0	-	110011
D.25	10011 0			x	100110	0	
D.26	01011 0			x	010110	0	
D/K.27	11011 x		L22’.L13’.E	-	110110	+	001001
D.28	00111 0			x	001110	0	
K.28	00111 1	L22.K	K	-	00111**1**	+	110000

D/K.29	10111 x		L22’.L13’.E	-	101110	+	010001
D/K.30	01111 x		L22’.L13’.E	-	011110	+	100001
D.31	11111 0	L40,L40 .E	L22’.L13’.E	-	1**01011**	+	010100

Table 6.2: 5B/6B encoding

An accent to the right of a symbol is used to represent complementation; E’ means the complement of *E,* a dot (.) stands for the logical AND function. In the column under the left **“abcdei”** heading, there are listed all the code points which are generated directly by the 5B/6B logic functions from the ABCDE inputs. The coding table is such that a minimal number of bits are changed on passing through the encoder, and the changes which are required can be classified into a few groups applicable to several code points. The extra digit “i” is added with a normal value of 0.

When the inputs meet the logical conditions listed under **“BIT ENCODING”**, then the bold type bits are changed to the values shown in the left “abcdei” column; e.g., if L04 holds, the b and *c* digits are forced to 1’s, as shown for D.0 and D.16. The second entry in the “bit encoding” column for D.16 (L04**.**E) and D.31 *(*L40.E) applies to the i-digit. For lines with no classification entry, the ABCDE bits translate unchanged into abcde and the added i-bit is a 0.

The “**ALTERNATE abcdei**” column to the right of Table 2 shows the complement for those ABCDE inputs which have alternate code points. Individual 6B (and 4B) sub blocks are complemented in conformity with the disparity rules.

The column **“D - 1”** indicates the required running disparity for entry to the adjacent sub block to the right. An x in the “D- 1” column means that (D - 1) can be + or. As an example for encoding of the first line D.0 of Table 2; If the running disparity matches (D - 1) = +, the output of the encoder will be 011000; otherwise the entire sub block is complemented to 100111.

The **“D0”** column indicates the disparity of the encoded sub block to the left, which is 0, +2, or -2. The disparities for the alternate code points on the right side of Tables 2 and 3 are exact complements of those to their left, and are not shown.

The **“disparity classifications”** indicate under which running disparity column the encoded data exists. If it matches the current running disparity, then it is left as it is. If it does not, then it is complemented.

6.2.2 3B/4B ENCODING LOGIC

The input to the 3B/4B encoder is 3-bit data FGH and the output is 4-bit encoded data fghj. The 3B/4B encoding is done according to the rules mentioned in Table 6.3.

NAME	FGH K	CLASSIFICATIONS		D-1	fghj	D0	fghj ALTERNATE
		BIT ENCODING	DISPARITY				
D/K.x.0	000 x	F’.G’.H’	F’.G’	+	0**1**00	-	1011
D.x.1	100 0	(F !=G).H ’		x	100**1**	0	
D.x.2	010 0			x	010**1**	0	
D/K.x.3	110 x	(F !=G).H ’	F.G	-	1100	0	0011
D/K.x.4	001 x		F’.G’	+	0010	-	1101
D.x.5	101 0			x	1010	0	
D.x.6	011 0			x	0110	0	
D.x.P7	111 0		F.G,F.G.H	-	1110	+	0001

D/K.y.A7	111 x		F.G,F.G.H	-	**0**11**1**	+	1000
K.28.1	100 1	F.G.H.(S+ K)	(F !=G).K	+	100**1**	0	0110
K.28.2	010 1	(F!=G).H’	(F !=G).K	+	010**1**	0	1010
K.28.5	101 1	(F!=G).H’	(F !=G).K	+	1010	0	0101
K.28.6	011 1		(F !=G).K	+	0110	0	1001

Table 6.3: 3B/4B encoding

Table 6.3 follows the conventions and notations of Table 2. In Table 3 some lines have two entries in the column for the classification of disparity; the left classification refers to the entry disparity D - 1, and the right one to D0.

The encoding of D.x.P7 (primary 7) and D/K.y.A7 (alternate 7) requires an explanation. The D/K.y.A7 code point was introduced to eliminate the run length 5 sequences in digits eifgh. The A7 code replaces the P7 encoding whenever

$$[(e = i = 1). (D - 1 = -)] \text{ OR } [(e = i = 0). (D - 1 = +)] \text{ OR } (K = 1)$$

Note that whenever K = 1, FGH = 111 is always translated into fghj = 0111 or its complement.

6.3 ENCODER SYSTEM LEVEL DESIGN

The top level design of the encoder is given below.

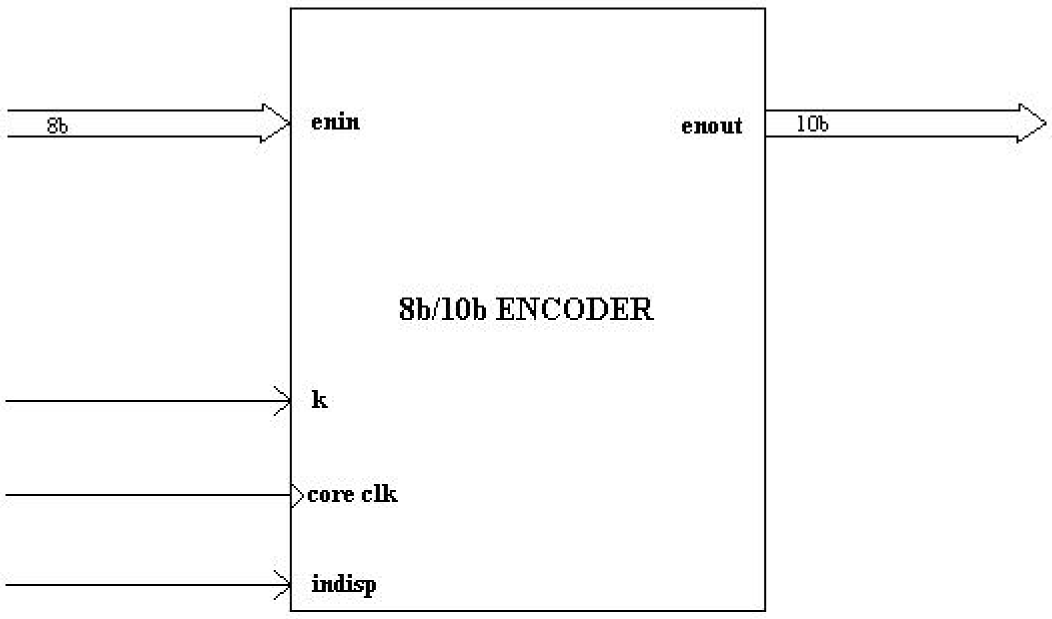


Figure 6.2: 8B/10B encoder

The scrambled data from the scrambler is given as input to the 8b/10b encoder. This 8 bit data is encoded according to the 8b/10b partitioned block encoding rules to give a 10 bit symbol. Whether the output symbol is a K code or a D code depends on the K input to the encoder. If K=0, the input byte is encoded into a D code. If K=1, the input byte is encoded into a K code. According to the USB 3.0 specifications, the transmitter is permitted to pick any initial running disparity. This provision is provided in the encoder through the INDISP input.

6.4 INTERNAL ARCHITECTURE

The internal architecture of 8B/10B encoder is given in Figure 6.3.

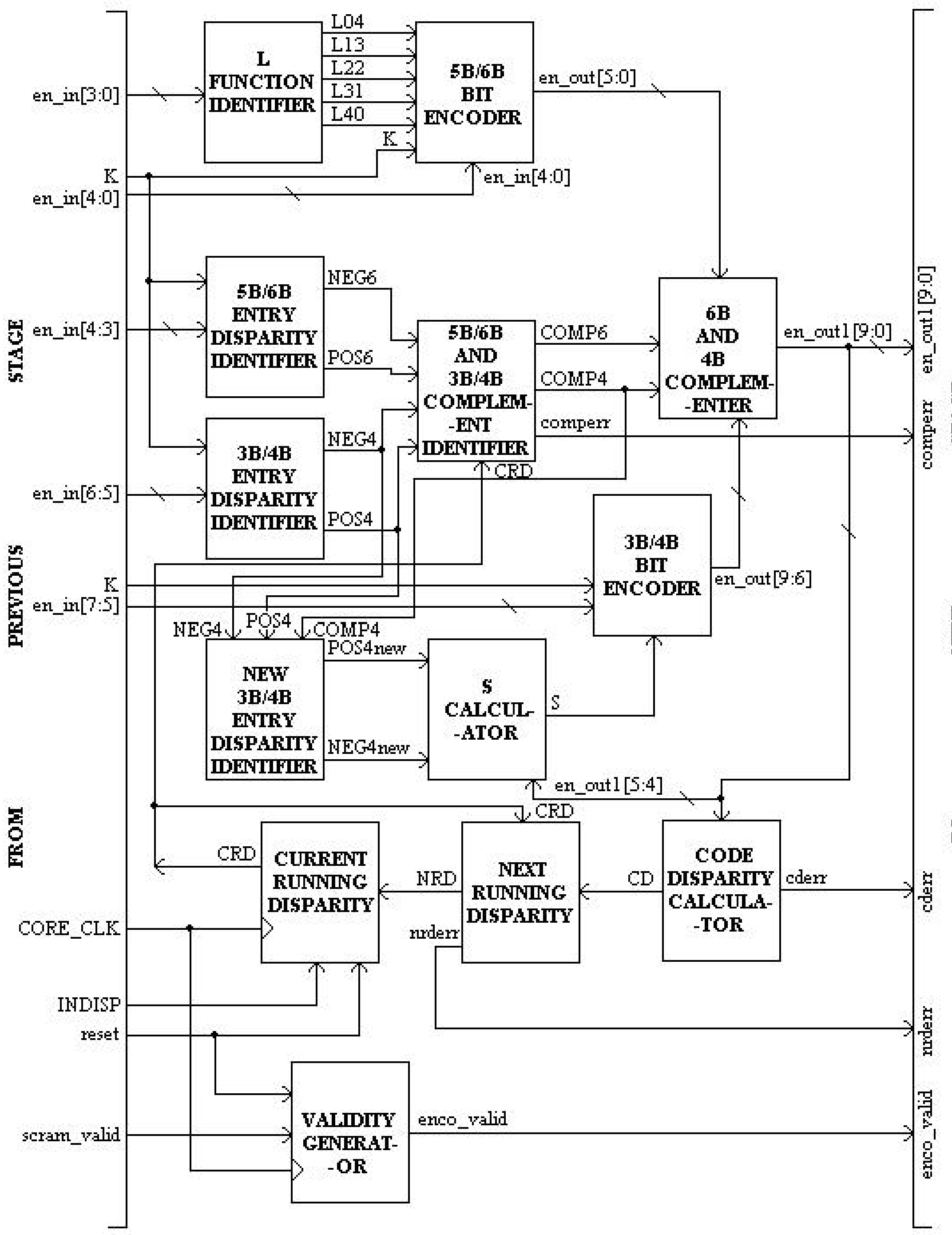


Fig 6.3: Internal architecture of encoder

The 8-bit data from the scrambler is given to the input buffer. During the positive edge of the core clock, the data and its complement become available to the 16-bit internal bus[5]. During the positive edge of the same clock, current running disparity also becomes available to the circuit. At the output, the 10-bit encoded data is passed on to the parallel-in-serial-out shift register.

6.5 INTERNAL MODULES

This section describes all the modules present in the encoder as seen in the internal architecture.

6.5.1 L FUNCTION IDENTIFIER

From Table 6.2, we observe that, there are 5 L functions. They are

L31: Three 1's; One 0

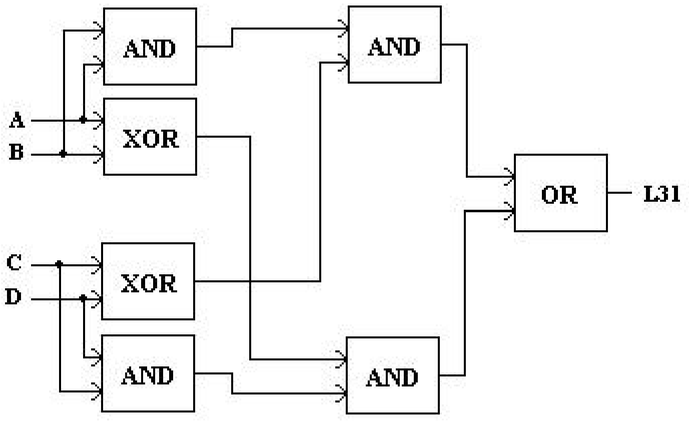


Figure 6.4: L31

L04: Zero 1's; Four 0's

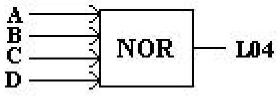


Figure 6.5: L04

L40: Four 1's; Zero 0's

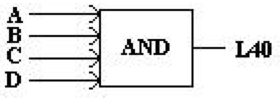


Figure 6.6: L40

[5] The bus is considered only for design purpose. In actual implementation there is no such bus.

L13: One 1; Three 0's

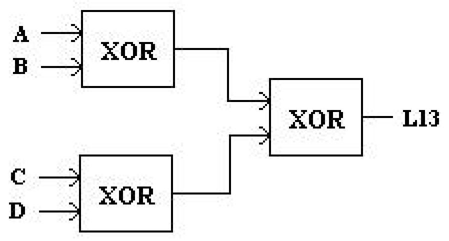


Figure 6.7: L13

L22: Two 1's; Two 0's

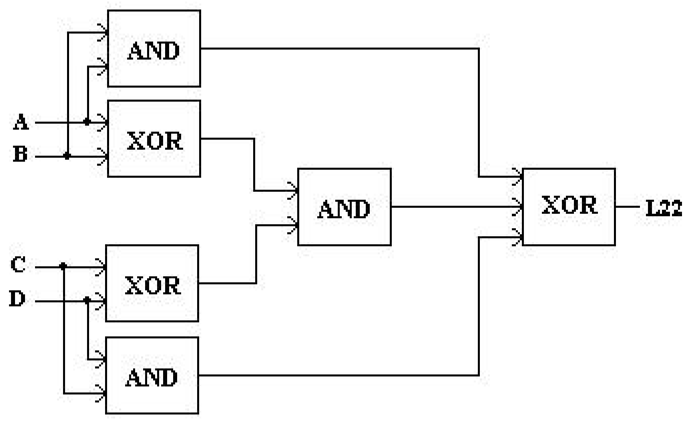


Figure 6.8: L22

6.5.2 5B/6B BIT ENCODER

This block encodes the 5-bit data at its input to 6 bits. The inputs to this block are data from the bus and the K signal. From table 2 under "BIT ENCODING" it can be observed that, there are certain class of encoding functions which complement certain input bits while passing them to the output. As mentioned earlier, "i" bit has a normal value of 0. It can be changed by the encoding functions. The encoding functions and the bits they complement, is listed in the table below.

ENCODING FUNCTIONS	COMPLEMENTED BITS

L04 L13.E’ L22.E’ L40 L04.E L13.D’.E L13.D.E L22.K L40.E	bc e i bd i i ce i i

Table 6.4: 5B/6B Encoding functions

The figure below shows the 5B/6B encoder.

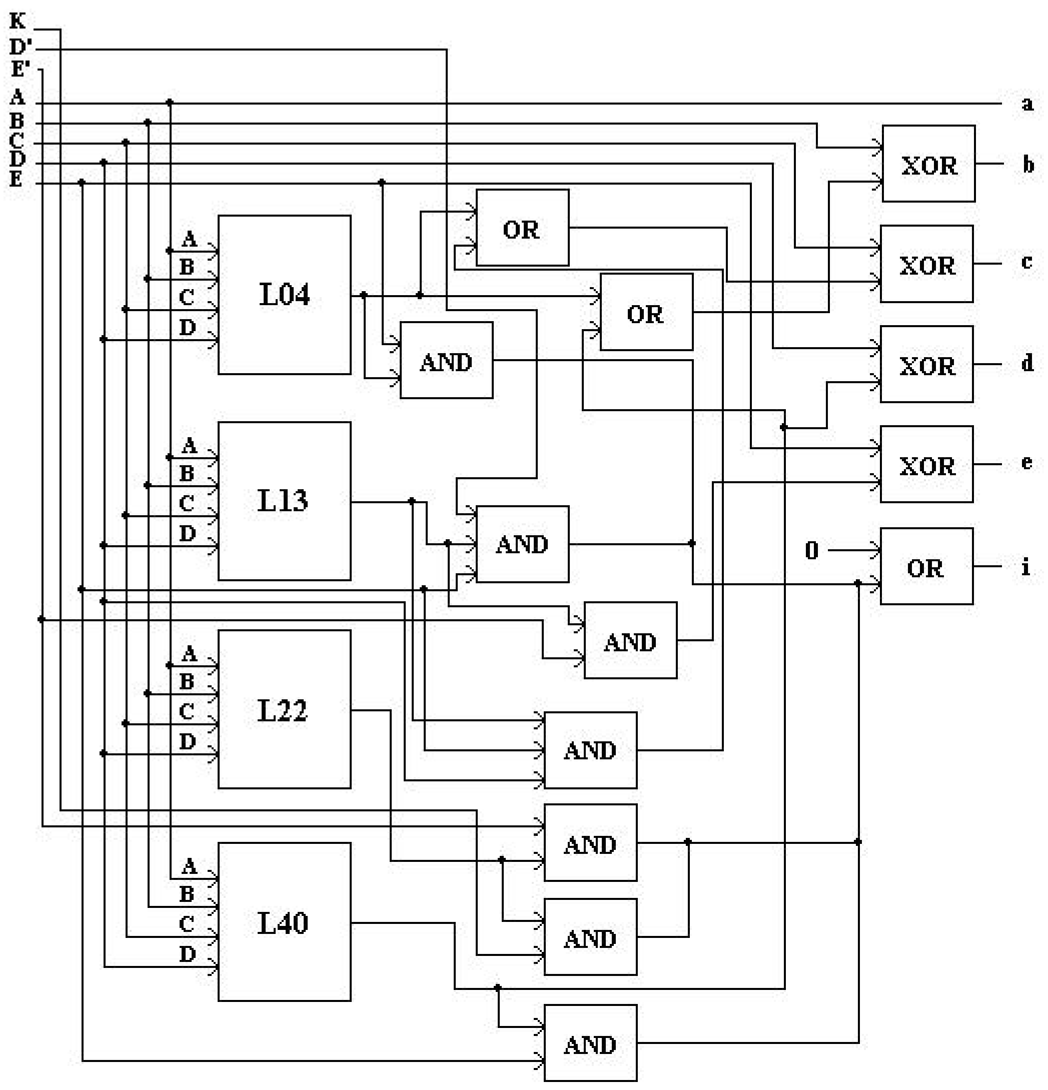


Figure 6.9: 5B/6B encoder

If any of the encoding function becomes true it complements the bits as mentioned in the above table. This complementation is done by the XOR gates at the output. If all the encoding functions are false, the inputs are passed as it is to the output.

6.5.3 5B/6B ENTRY DISPARITY IDENTIFIER

This block determines whether the 5B/6B encoder has encoded the 5-bit input by taking values from either the positive running disparity column or the negative running disparity column. The table below shows conditions for entry into respective columns.

DISPARITY FUNCTIONS	ENTRY DISPARITY
L22'.L31'.E'	+

L31.D'.E'	-
L22'.L13'.E	-
L13.D.E	+
K	-

Table 6.5: Disparity functions with their associated entry disparities for 5B/6B encoder

The figure below shows the 5B/6B entry disparity identifier.

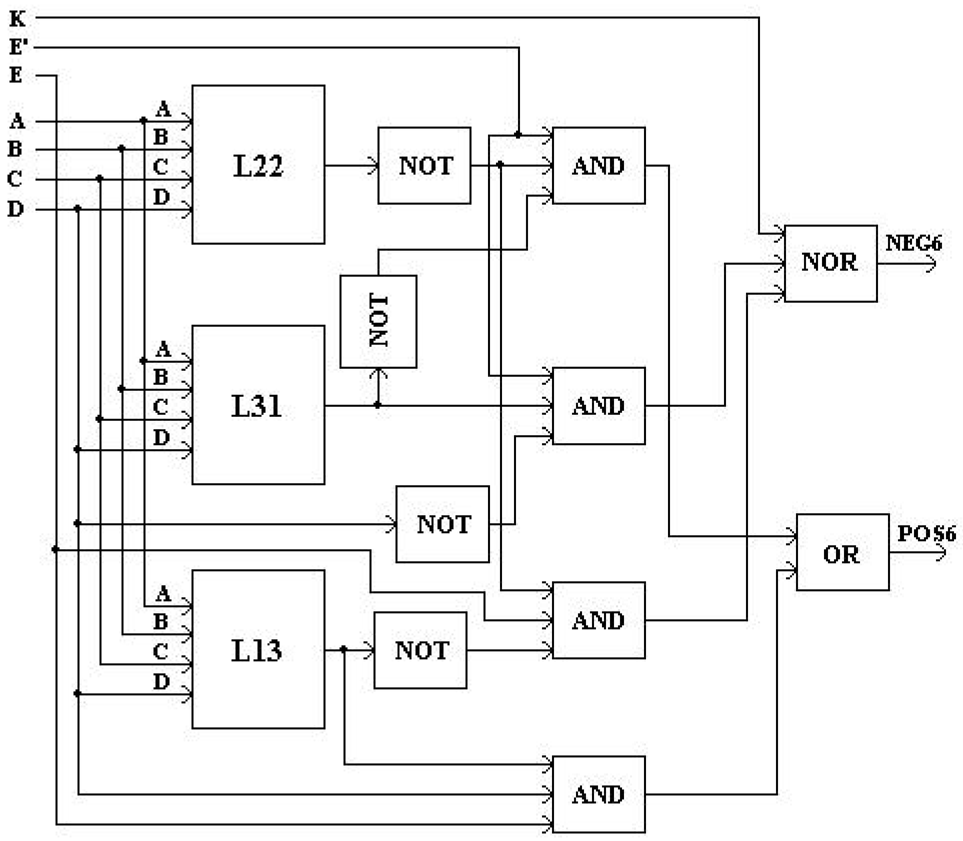


Figure 6.10: 5B/6B entry disparity identifier

6.5.4 3B/4B ENTRY DISPARITY IDENTIFIER

This block determines whether the 3B/4B encoder has encoded the 3-bit input by taking values from either the positive running disparity column or the negative running disparity column. The table below shows conditions for entry into respective columns.

DISPARITY FUNCTIONS	**ENTRY DISPARITY**
F'.G'	+
F.G	-
(F != G).K	+

Table 6.6: Disparity functions with their associated entry disparities for 3B/4B encoder

The figure below shows the 3B/4B entry disparity identifier.

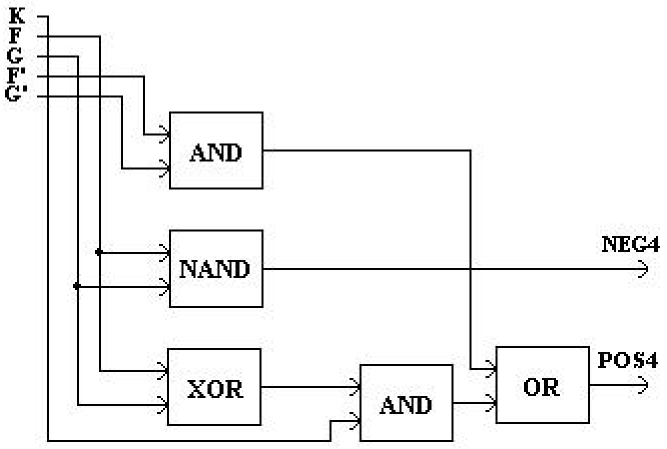


Figure 6.11: 3B/4B entry disparity identifier

6.5.5 5B/6B AND 3B/4B COMPLEMENT IDENTIFIER

NEG4, POS4, NEG6, POS6 and CRD are given as input to this block. The outputs are signals COMP6 and COMP4 which determine whether the 6B or the 4B blocks have to be complemented. If any error condition arises then comperr goes high. The table below shows the conditions under which the COMP6 and COMP4 signals will be generated.

POS6	**NEG6**	**POS4**	**NEG4**	**CRD**	**COMP6**	**COMP4**

0	1	0	1	0	0	0
0	1	0	1	1	0	0
1	1	0	1	1	0	0
1	1	0	1	0	1	0
0	0	0	1	1	1	0
0	0	0	1	0	0	0
0	1	1	1	1	0	0
0	1	1	1	0	0	1
0	1	0	0	1	0	1
0	1	0	0	0	0	0
1	1	1	1	1	0	1
0	0	0	0	1	1	0
1	1	1	1	0	1	0
0	0	0	0	0	0	1
0	0	1	1	0	0	1
0	0	1	1	1	1	1
1	1	0	0	0	1	1
1	1	0	0	1	0	0

Table 6.6: Conditions for the generation of COMP4 and COMP6 signals

6.5.6 NEW 3B/4B ENTRY DISPARITY IDENTIFIER

If the COMP4 signal is asserted, it means that either POS4 or NEG4 were different from the CRD, hence the 4B code is complemented. Under such a condition, this block calculates the latest value of POS4 and NEG4, which are labeled as POS4new and NEG4new.

6.5.7 S CALCULATOR

This block computes S, which is required to determine whether P7 (primary 7) or A7 (alternate 7) encoding should be used when FGH=111.

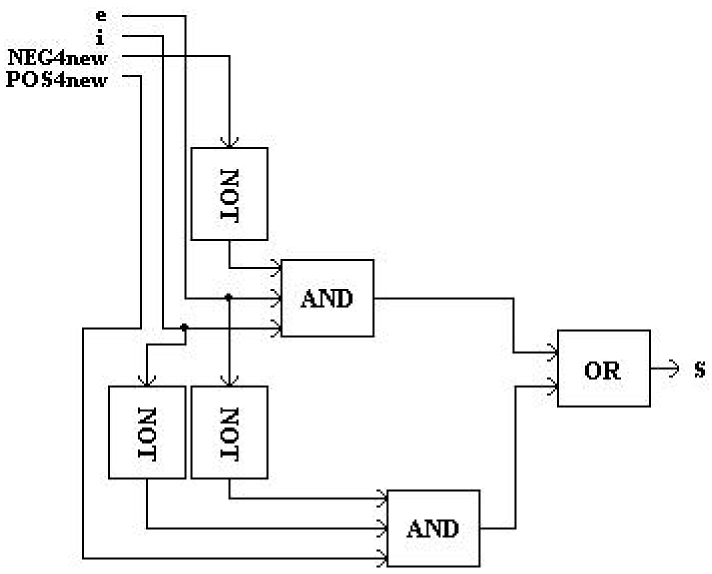


Figure 6.12: S calculator

6.5.8 3B/4B ENCODER

The 3-bit input data is encoded into 4 bits at the output. The "BIT ENCODING" functions of Table 6.3 are used in this circuit. The table below determines which of the input bits should be complemented to generate the encoded data.

DISPARITY FUNCTIONS	ENTRY DISPARITY
F'.G'	g
F.G	j
(F != G).K	fj

Table 6.7: 3B/4B Encoding functions

The j-bit has a normal value of 0. It can be changed to 1 by the encoding functions. The complementation is done by the XOR gates. The figure below shows 3B/4B encoder.

When all the encoding functions are zero, the input is passed as it is to the output. The "S" input is given to the circuit to avoid a run length of 5 in digits eifgh as mentioned in earlier sections.

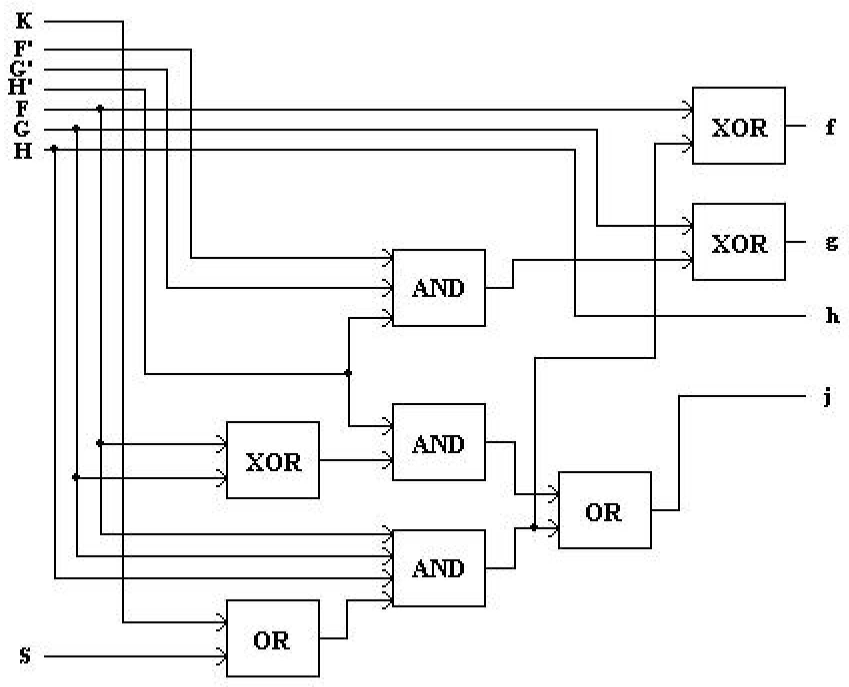


Figure 6.13: 3B/4B encoder

6.5.9 CODE DISPARITY CALCULATOR

This block performs the function of the adder. It determines the number of 1's in the 10-bit encoded output. This is used to determine the next running disparity. The signal cderr goes high whenever an error condition arises in this block.

6.5.10 NEXT RUNNING DISPARITY CALCULATOR

This block calculates the next running disparity based on the code disparity (CD) and the current running disparity (CRD). The table below shows the conditions for the generation of the next running disparity. 0 indicates negative next running disparity and 1 indicates positive next running disparity. If any error condition occurs nrderr goes high.

CRD	**CD**	**NRD**
0	0	0
0	+	1
1	0	1
1	-	0

Table 6.8: Next running disparity rules

6.5.11 CURRENT RUNNING DISPARITY CALCULATOR

During the positive edge of the clock, this circuit provides the current running disparity to the encoder circuit. The inputs are initial disparity (INDISP), core clock, reset and next running disparity. For the first byte of data the current running disparity is INDISP. For subsequent bytes, CRD is NRD calculated during the previous clock cycle.

6.5.12 6B AND 4B COMPLEMENTER

Based on the signals COMP4 and COMP6, this block complements the 6-bit encoded data, 4-bit encoded data, both or neither. The figures given below show the 4-bit and 6-bit complementer.

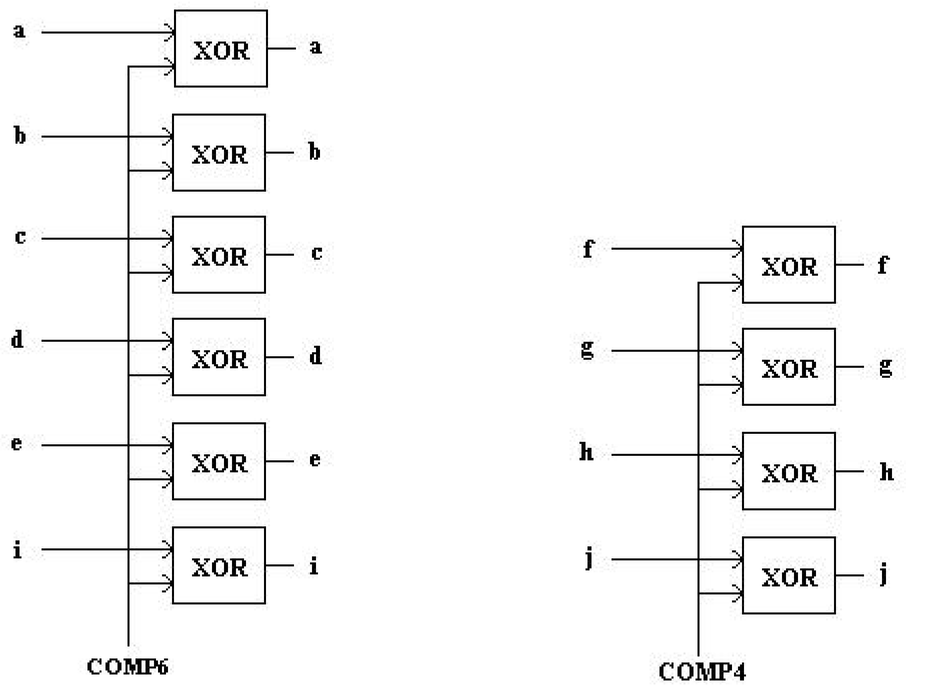


Figure 6.14: 6B and 4B Complementer

6.5.13 VALIDITY GENERATOR

The encoder circuit begins its operation once the valid signal from the previous stage scram_valid goes high. The valid signal out of the encoder enco_valid is generated after a clock cycle delay after asserting scram_valid.

CHAPTER 7

DECODER

The Decoder module is based exactly on the same principle of the Encoder module. An attribute present in the decoder module is its ability of error detection. The concepts of Running Disparity and Code Disparity remain the same as in encoder and are hence not explained here. Only a slight difference is in the encoding tables, the classification functions were denoted by 'L' where as in decoder it is denoted by 'P'. We straight away jump into the error detection schemes of the decoder.

Error Detection

There are two types of errors that will be detected:

1. Running Disparity error
2. Coding Disparity error.

Running Disparity Error: This scheme uses the same disparity generator block used in the encoder to calculate the NRD. The Nrd err bit becomes high for the last two cases as shown in the table.

CRD	CD	NRD	Nrd err
0	0	0	0
0	+	1	0
1	0	1	0
1	-	0	0
0	-	-3	1
1	+	+3	1

Coding Disparity Error: This scheme checks to see whether the incoming code disparity falls in either 0,+ 2 or -2 categories. If this condition is not met, a code disparity eror cd err bit is raised. i.e. if the no. Of 1's is less than 4 or greater than 6.

In addition, other error detection schemes such as the alternating disparity rule between 3b/4b and 5b/6b blocks can be utilised.

Also, the following conditions are violations of coding rules and can be attributed to errors.

- a=b=c=d
- P13.e'.i'
- P31.e.i
- e=i=f=g=h
- i≠e=g=h=j
- (e=i≠g=h=j).(c=d=e)'
- P31'.e.i'.g'.h'.j'
- P13.e'.i.g.h.j

On flagging of either of the error bits, the decoder output is made zero.

7.1 6B/5B DECODER.

From the [5:0] bits of the decoder input , their disparity class is determined and they are decoded accordingly as given in the below table.

Name	abcdei	Decoding class	Disparity class	ABCDE	K	D.1	DO
D.0	011000	P22.b.c. (e=i)	P22.e'.i'	00000	0	+	-
D.0	100111	P22.b'.c'.	P22.e.i	00000	0	-	+

		(e=i)					
D.1	100010	P13.i’	P13.i’	10000	0	+	-
D.1	011101	P31.i	P31.i	10000	0	-	+
D.2	010010	P13.i’	P13.i’	01000	0	+	-
D.2	101101	P31.i	P31.i	01000	0	-	+
D.3	110001			11000	0	X	0
D.4	001010	P13.i’	P13.i’	00100	0 +	+	-
D.4	110101	P31.i	P31.i	00100	0	-	+
D.5	101001			10100	0	X	0
D.6	011001			01100	0	X	0
D.7	111000		P31.d’.e’.i’	11100	0	-	0
D.7	000111	P13.d.e.i	P13.d.e.i	11100	0	+	0
D.8	000110	P13.i’	P13.i’	00010	0	+	-
D.8	111001	P31.i	P31.i	00010	0	-	+
D.9	100101			10010	0	X	0
D.10	010101			01010	0	X	0
D.11	110100			11010	0	X	0
D.12	001101			00110	0	X	0
D.13	101100			010110	0	X	0
D.14	011100			001110	0	X	0
D.15	101000	P22.a.c.(e=i)	P22.e’.i’	11110	0	+	-
D.15	010111	P22.a’.c’. (e=i)	P22.e.i	11110	0	-	+
D.16	011011	P22.b.c. (e=i)	P22.e.i	00001	0	-	+
D.16	100100	P22.b’.c’. (e=i)	P22.e’.i’	00001	0	+	-
D.17	100011			010001	0	X	0
D.18	010011			001001	0	X	0
D.19	110010			11001	0	X	0
D.20	001011			00101	0	X	0
D.21	101010			010101	0	X	0
D.22	011010			001101	0	X	0

D/K.23	111010	P31.e		11101	x	-	+
D/K.23	000101	P13.e’	P13.e’	11101	x	+	-
D.24	001100	a’.b’.e’.i’	P22.e’.i’	00011	0	+	-
D.24	110011	a.b.e.i	P22.e.i	00011	0	-	+
D.25	100110			0 10011	0	X	0
D.26	010110			01011	0	X	0
D/K.27	110110		P31.e	11011	x	-	+
D/K.27	001001	P13.e’	P13.e’	11011	x	+	-
D.28	001110			00111	0	X	0
D.28	001111	c.d.e.i	P22.e.i	00111	1	-	+
D.28	110000	c’.d’.e’.i’	P22.e’.i’	00111	1	+	-
D/K.29	101110	P31.e		10111	x	-	+
D/K.29	010001	P13.e’	P13.e’	10111	x	+	-
D/K.30	011110		P31.e		x	-	+
D/K.30	100001	P13.e’	P13.e’	01111	x	+	-
D.31	101011	P22.a.c.(c=i)	P22.e.i	11111	0	-	+
D.31	010100	P22.a’.c’. (e=i)	P22.e’.i’	11111	0	+	-

Table 7.2: 6B/5B Decoding

7.2 4B/3B DECODER

From the [9:6] bits of the decoder input , their disparity class is determined and they are decoded accordingly as given in the below table.

Name	fghj	decoding class	disparity class	FGH	K	D-1	D0
D/K.x.0	0100	f’.h’.j	f’.h’.j’	000	x	+	-
D/K.x.0	1011	f.h.j	f.h.j	000	x	-	+
D/ K.x.1	1001			10 0	x	X	0
K.28.1	0110	c’.d’.e’.i’.(h≠j)		100	1		0
D/K.x.2	0101			010	x	X	0
K.28.2	1010	c’.d’.e’.i’.(h≠j)		010	1		0
D/K.x.3	1100		f.g.h’.j’	110	x	-	0
D/K.x.3	0011	f’.g’.h.j	f’.g’.h.j	110	x	+	0
D/K.x.4	0010		f’.g’.j’	001	x	+	-
D/K.x.4	1101	f.g.j	f.g.j	001	x	-	+
D/K.x.5	1010			101	x	X	0
K.28.5	0101	c’.d’.e’.i’.(h≠j)		101	1		0
D/K.x.6	0110			011	x	x	0
K.28.6	1001	c’.d’.e’.i’.(h≠j)		011	1		0
D.x.7	1110		f.g.h	111	0	-	+
D.x.7	0001	f’.g’.h’	f’.g’.h’	111	0	+	-
D/K.x.7	0111	g.h.j	g.h.j	111	x	-	+
D/K.x.7	1000	g’.h’.j’	g’.h’.j	111	x	+	-

7.3 DESIGN &IMPLEMENTATION

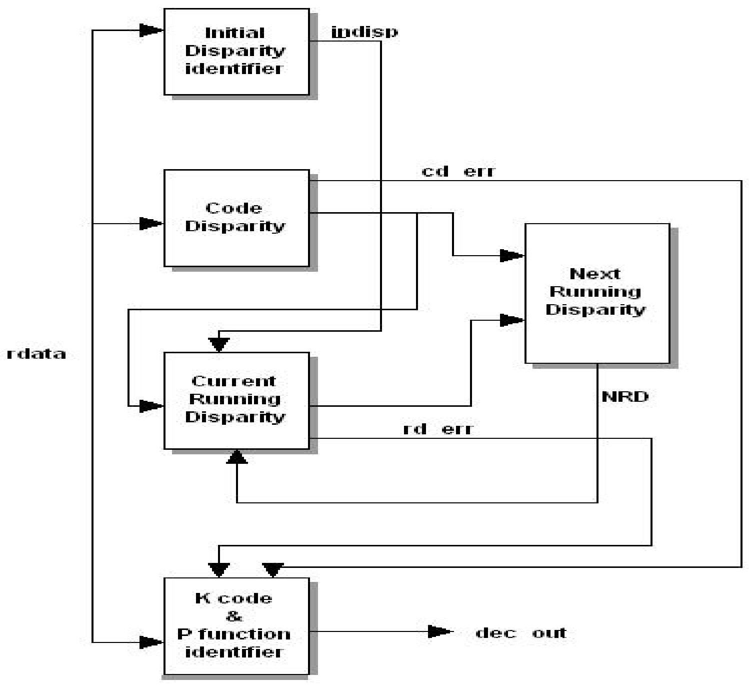

Initial Disparity identifier
indisp
Code Disparity
cd err
Next Running Disparity
rdata
Current Running Disparity
NRD
rd err
K code & P function identifier
dec out

Fig 7.1: internal Architecture of decoder

The input data from the buffer *rdata* is classified into it's respective classes for both 6B/5B and 4B/3B decoding. Initially, on detection of the sync word, initial disparity is identified and is assigned as the CRD. Using the values of CRD and the simultaneously calculated value of CD, NRD is calculated and is assigned as the CRD in the next clock cycle. This process of calculating CRD, CD and NRD continues for each data. The design is cycle accurate since NRD which becomes the CRD for the next appearing data is assigned only at the next rising edge of clock.

In the process of calculating CD and NRD, if there is an error, then the current data is not decoded and the output is fluffed to zero indicating appropriate errors (*cd err, rd err*).

Summary:

Cycle time of operation: 1 core clock cycle.

Decoder output values and simulation results have been provided in appendix.

CHAPTER 8

PARALLEL IN SERIAL OUT SHIFT REGISTER

The parallel-in/ serial-out shift register stores data, shifts it on a clock by clock basis, and delays it by the number of stages times the clock period. In addition, parallel-in/ serial-out really means that we can load data in parallel into all stages before any shifting ever begins. This is a way to convert data from a parallel format to a serial format. By parallel format we mean that the data bits are present simultaneously on individual wires, one for each data bit as shown below. By serial format we mean that the data bits are presented sequentially in time on a single wire or circuit as in the case of the ”data out” on the block diagram below.

8.1 BLOCK DIAGRAM

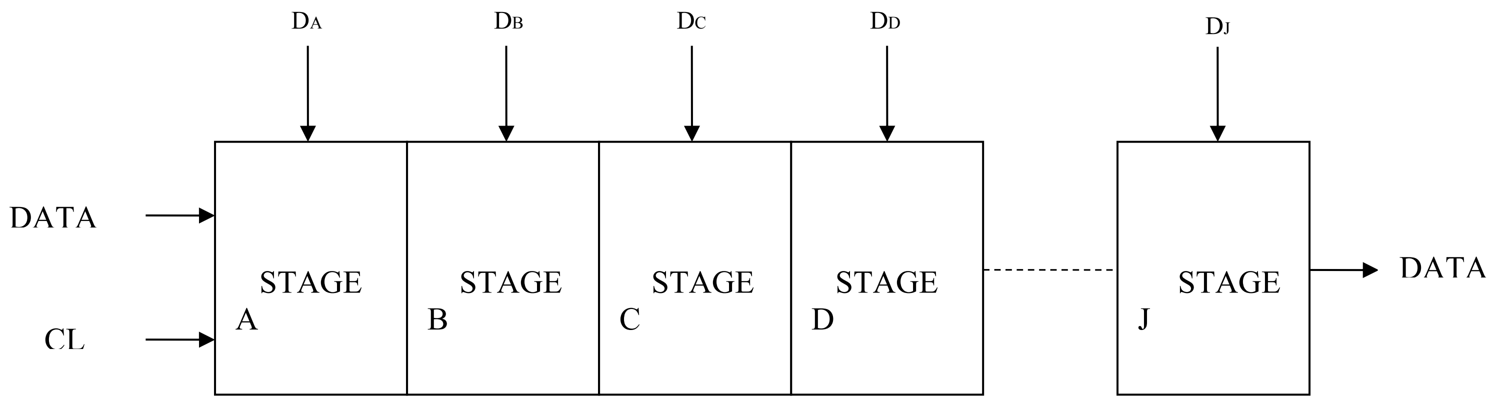


Fig 8.1: Block Diagram of PISO

Below we take a close look at the internal details of a 10-stage parallel-in/ serial-out shift register. A stage consists of a type **D** Flip-Flop for storage, and an AND-OR selector to determine whether data will load in parallel, or shift stored data to the right.

8.2 PISO REGISTER SHOWING PARALLEL LOAD PATH

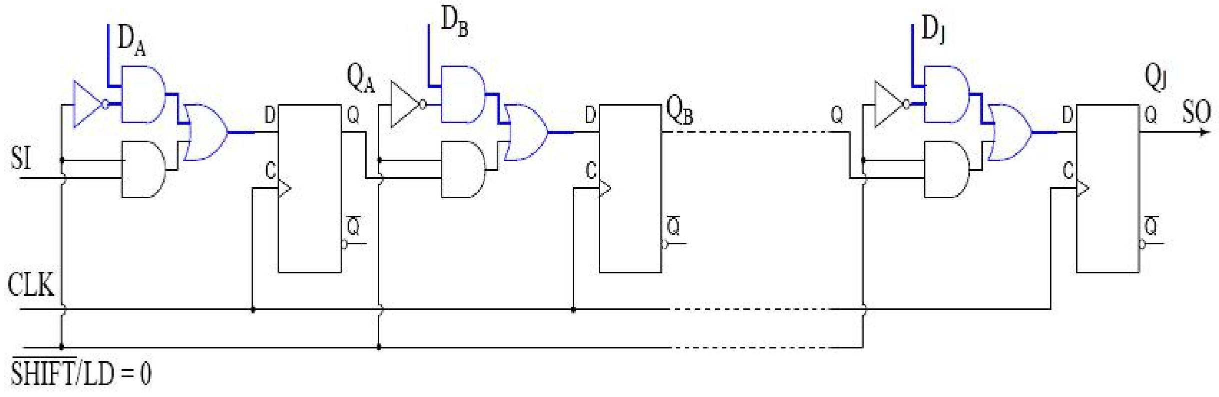


Fig 8.2: PISO Register Showing Parallel Load Path

Above we show the parallel load path when SHIFT/LD' is logic low. The upper NAND gates serving DA DB DJ are enabled, passing data to the D inputs of type D Flip-Flops QA QB respectively. At the next positive going clock edge, the data will be clocked from D to Q of the ten FFs. Ten bits of data will load into QA QB DJ at the same time. The type of parallel load just described, where the data loads on a clock pulse is known as synchronous load because the loading of data is synchronized to the clock. This needs to be differentiated from asynchronous load where loading is controlled by the preset and clear pins of the Flip-Flops which do not require the clock. Only one of these load methods is used within an individual device, the synchronous load being more common in newer devices

8.3 PISO REGISTER SHOWING SHIFT PATH

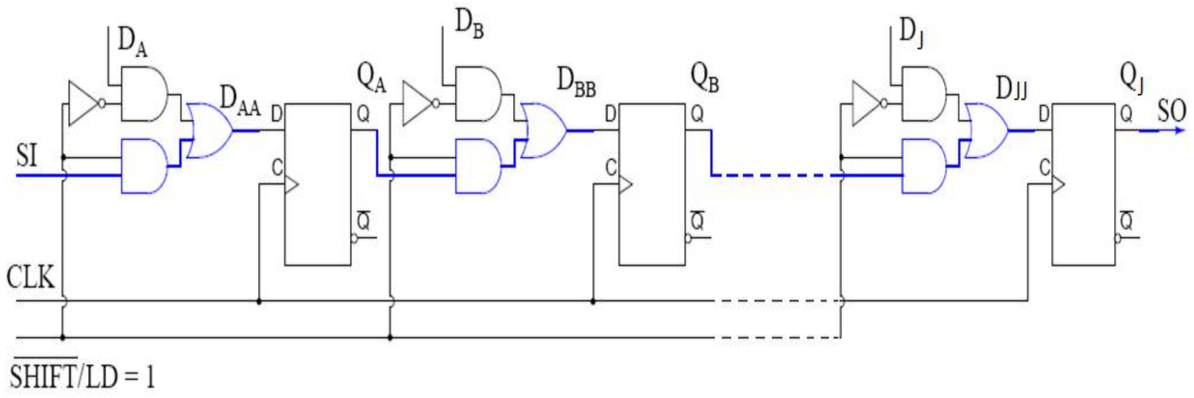


Fig8.3: PISO Register Showing Shift Path

The shift path is shown above when SHIFT/LD' is logic high. The lower AND gates of the pairs feeding the OR gate are enabled giving us a shift register connection of SI to DA, QA to DB , QB to DC , QJ to SO. Clock pulses will cause data to be right shifted out on successive pulses.

Summary

- Cycle of operation: In 1 clock cycle (core clock), 10 bits of data will be transmitted serially.
- Simulations of PISO can be viewed in the Transmitter and Physical Layer simulations.

CHAPTER 9

SERIAL IN PARALLEL OUT SHIFT REGISTER

A serial-in parallel-out shift register converts data from serial format to parallel format. If four data bits are shifted in by four clock pulses via a single wire at data-in, below, the data becomes available simultaneously on the four Outputs QA to QD after the fourth clock pulse.

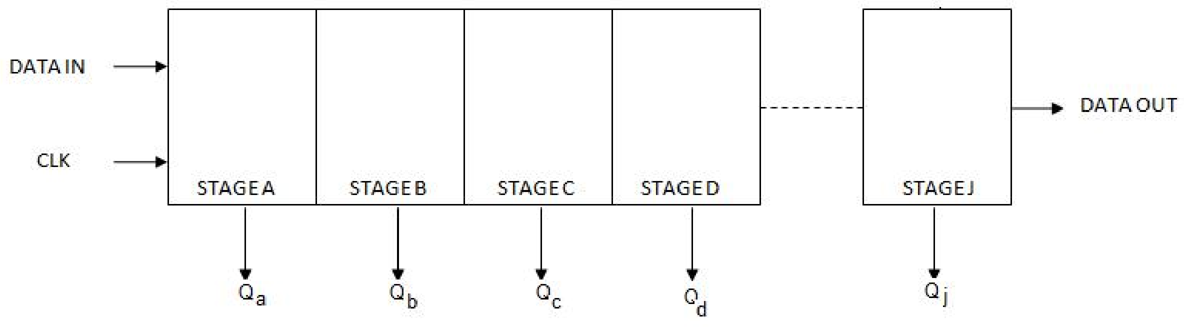


Fig 9.1: Block Diagram of SIPO

The practical application of the serial-in parallel-out shift register is to convert data from serial format on a single wire to parallel format on multiple wires.

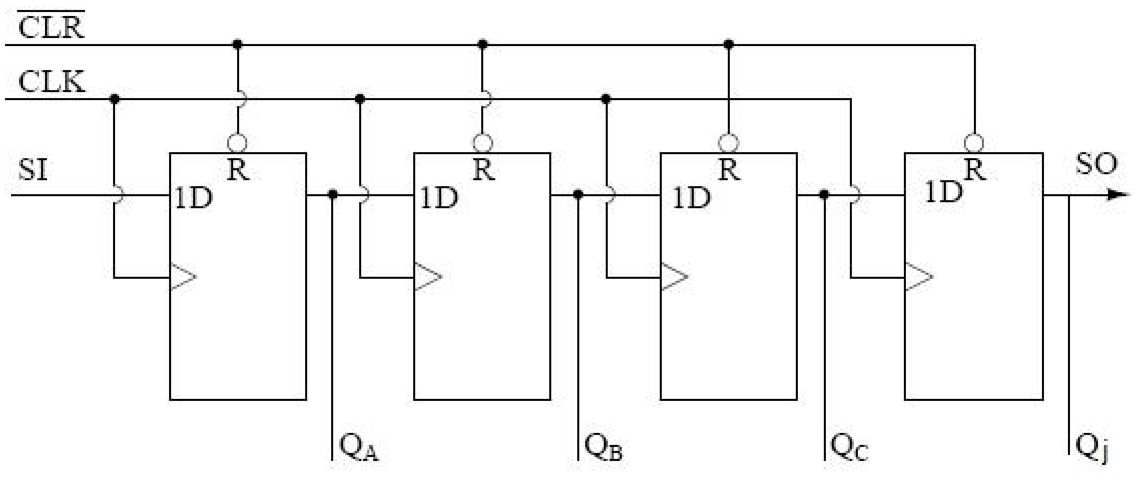


Fig 9.2: Serial in Parallel Out Shift Register

The above details of the serial-in/parallel-out shift register are fairly simple. It looks like a serial-in/ serial-out shift register with taps added to each stage output. Serial data shifts in at **SI** (Serial Input). After a number of clocks equal to the number of stages, the first data bit in appears at SO (QJ) in the above figure. In general, there is no SO pin. The last stage (QJ above) serves as SO and is cascaded to the next package if it exists.

Summary

- Cycle of operation: In 1 clock cycle (core clock), 10 bits of data will be received serially.
- Simulations of SIPO can be viewed in the Receiver and Physical Layer simulations.

CHAPTER 10

FIFO BUFFER

10.1 Introduction

FIFO is an acronym for First In, First Out, an abstraction in ways of organizing and manipulation of data relative to time and prioritization. This expression describes the principle of a queue processing technique or servicing conflicting demands by ordering process by first-come, first-served (FCFS) behavior.

FIFOs are used commonly in electronic circuits for buffering and flow control which is from hardware to software. In hardware form a FIFO primarily consists of a set of read and write pointers, storage and control logic. Storage may be SRAM, flip-flops, latches or any other suitable form of storage. For FIFOs of non-trivial size a dual-port SRAM is usually used where one port is used for writing and the other is used for reading.

A synchronous FIFO is a FIFO where the same clock is used for both reading and writing. An asynchronous FIFO uses different clocks for reading and writing. Asynchronous FIFOs introduce metastability issues. A common implementation of an asynchronous FIFO uses a Gray code (or any unit distance code) for the read and writes pointers to ensure reliable flag generation. One further note concerning flag generation is that one must necessarily use pointer arithmetic to generate flags for asynchronous FIFO implementations.

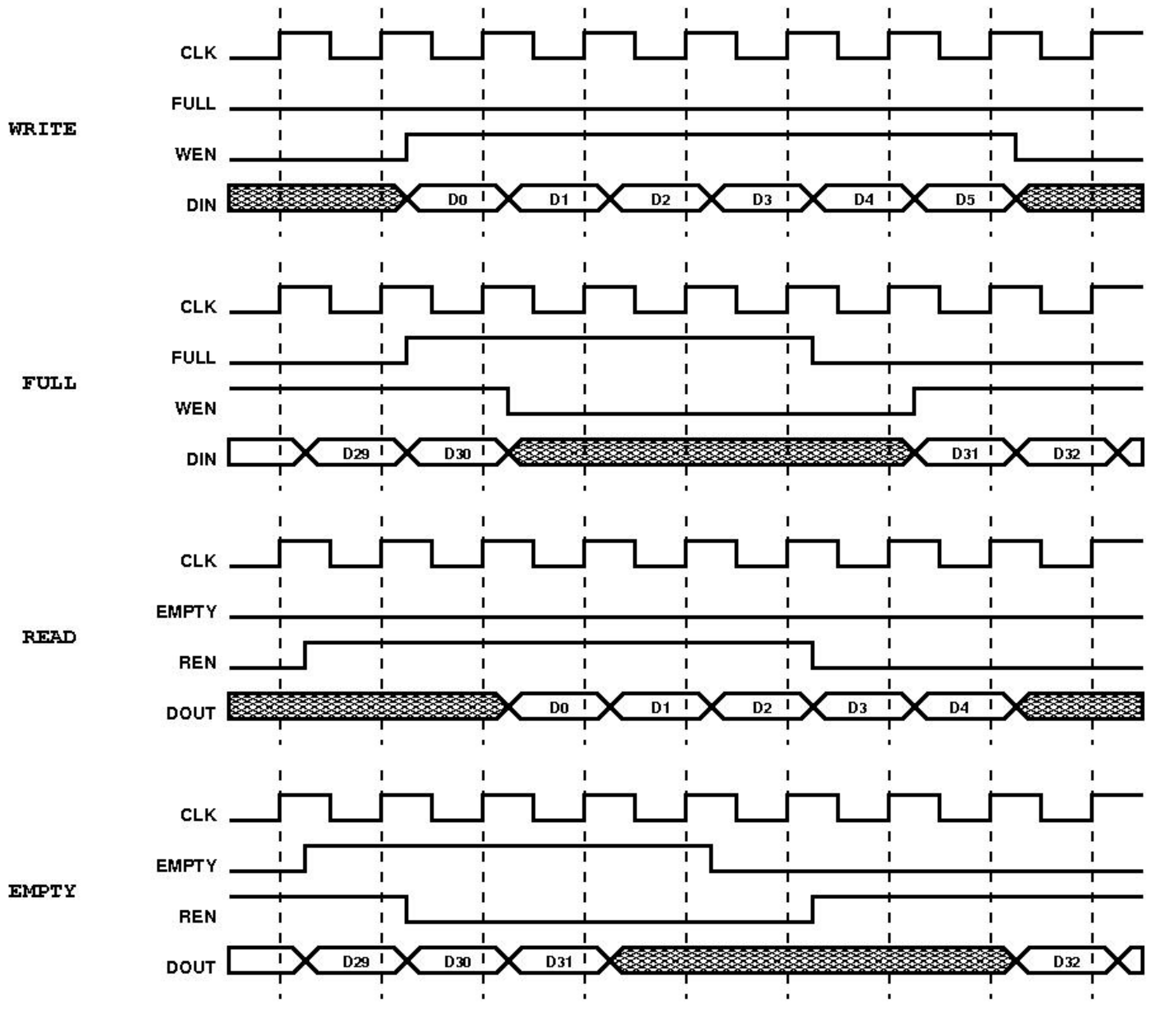


Fig 10.1: Timing Information

Pointer implementation

One Gray code counter style uses a single set of flip-flops as the Gray code register with accompanying Gray-to binary conversion, binary increment, and binary-to-Gray conversion.

A second Gray code counter style, the one described uses two sets of registers, one a binary counter and a second to capture a binary-to-Gray converted value. The intent of this Gray code counter is to utilize the binary carry structure, simplify the Gray-to-binary conversion; reduce combinational logic, and increase the upper frequency limit of the Gray code counter.

The binary counter conditionally increments the binary value, which is passed to both the inputs of the binary counter as the next-binary-count value, and is also passed to the simple binary-to-Gray conversion logic, consisting of one 2-input XOR gate per bit position. The converted binary value is the next Gray-count value and drives the Gray code register inputs.

This implementation requires twice the number of flip-flops, but reduces the combinatorial logic and can operate at a higher frequency. In FPGA designs, availability of extra flip-flops is rarely a problem since FPGAs typically contain far more flip-flops than any design will ever use. In FPGA designs, reducing the amount of combinational logic frequently translates into significant improvements in speed. The **ptr** output of the block diagram in Figure is an n-bit Gray code pointer.

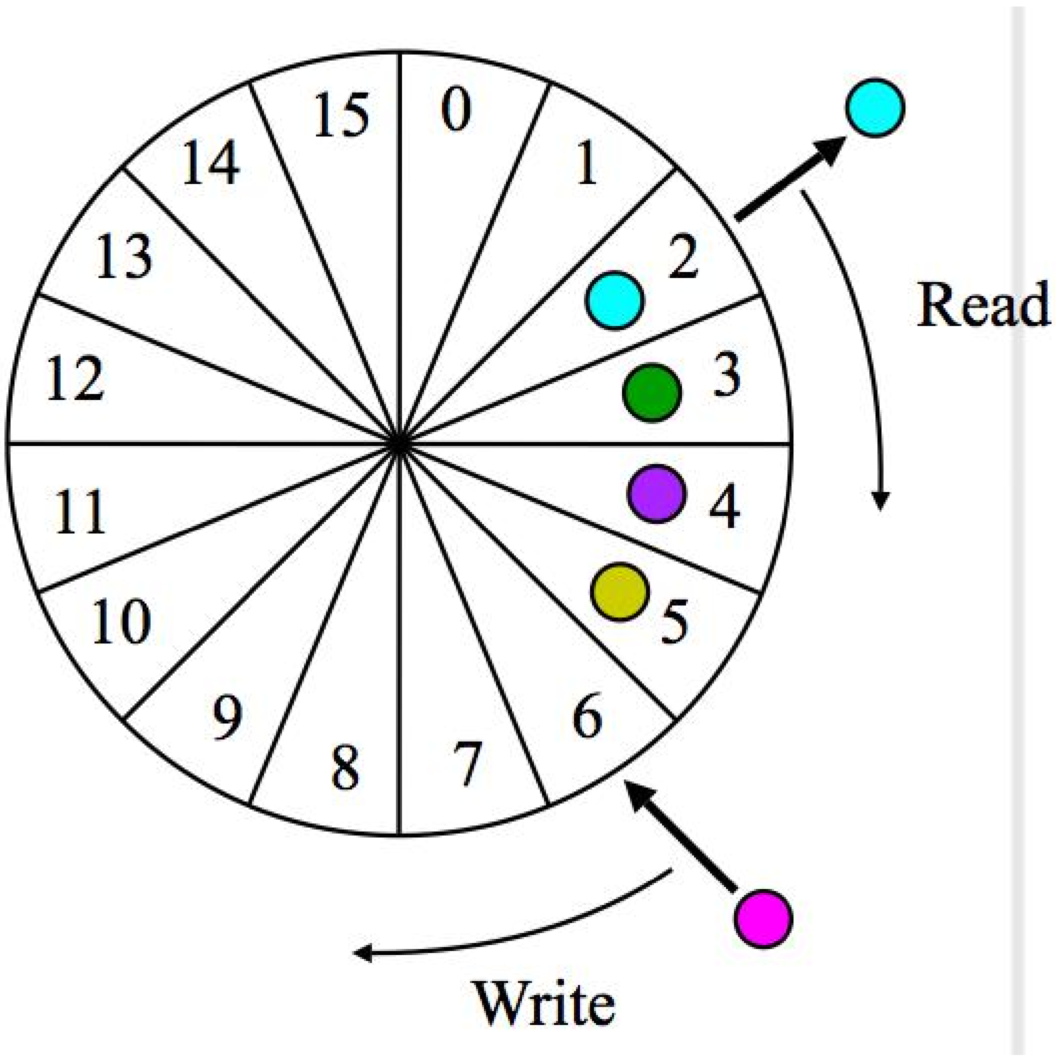


Fig 10.2: Pointer Implementation

.

. **Full & empty detection**

As with any FIFO design, correct implementation of full and empty is the most difficult part of the design.

There are two problems with the generation of full and empty:

First, both full and empty are indicated by the fact that the read and write pointers are identical. Therefore, something else has to distinguish between full and empty. One known solution to this problem appends an extra bit to both pointers and then compares the extra bit for equality (for FIFO empty) or inequality (for FIFO full), along with equality of the other read and write pointer bits.

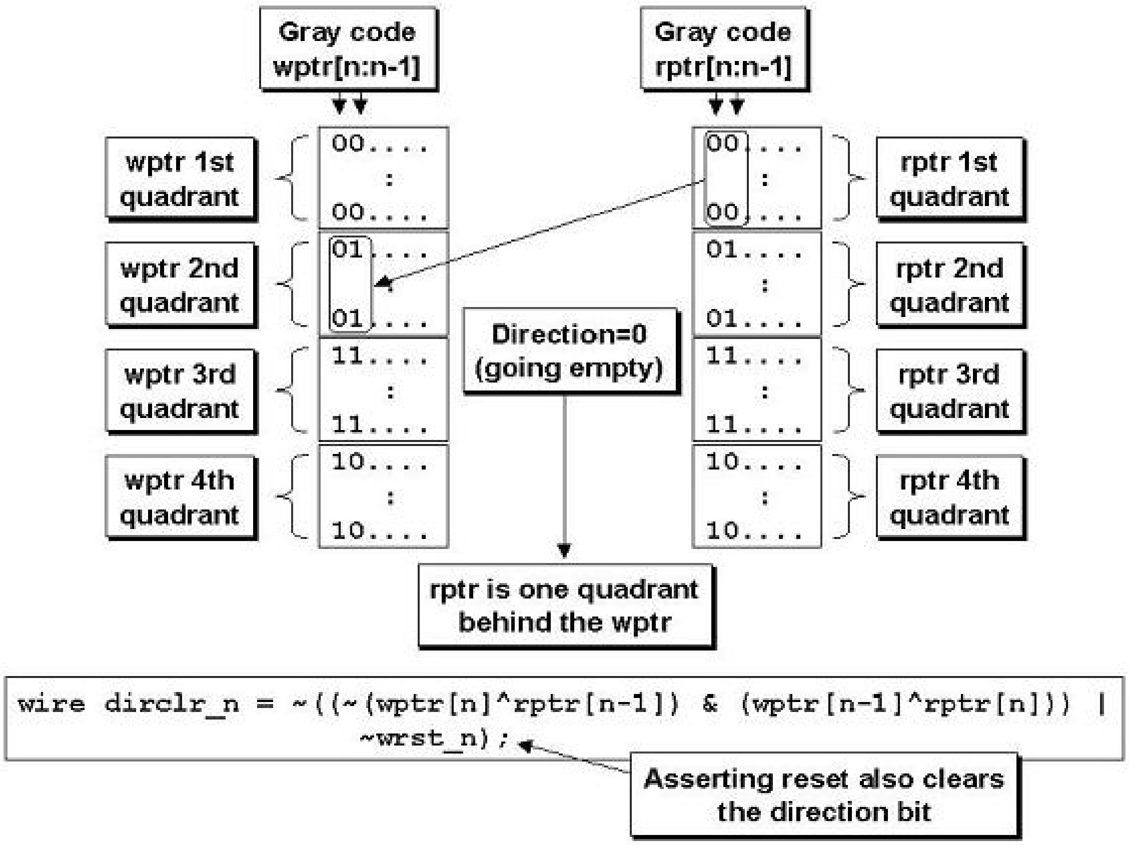
Another solution is to divide the address space into four quadrants and decodes the two MSBs of the two counters to determine whether the FIFO was going full or going empty at the time the two pointers became equal. If the write pointer is one quadrant behind the read pointer, this indicates a "possibly going full" situation as shown. When this condition occurs, the **direction** latch is set. If the write pointer is one quadrant ahead of the read pointer, this indicates a "possibly going empty" situation as shown. When this condition occurs, the **direction** latch of Figure is cleared When the FIFO is reset the **direction** latch is also cleared to indicate that the FIFO “is going empty” (actually, it *is* empty when both pointers are reset). Setting and resetting the **direction** latch is not timing-critical, and the direction latch eliminates the ambiguity of the address identity decoder.

The second, and more difficult, problem stems from the asynchronous nature of the write and read clocks.

Comparing two counters that are clocked asynchronously can lead to unreliable decoding spikes when either or both counters change multiple bits more or less simultaneously. The solution described uses a Gray count sequence, where only one bit changes from any count to the next. Any decoder or comparator will then switch only from one valid output to the next one, with no danger of spurious decoding glitches.

10.2 FIFO

The FIFO style described does asynchronous comparison between Gray code pointers to generate an asynchronous control signal to set and reset the full and empty flip-flops.



```
wire dirclr_n = ~((~(wptr[n]^rptr[n-1]) & (wptr[n-1]^rptr[n])) |
                    ~wrst_n);
```

Fig 10.3: FIFO is going empty because the rptr trails the wptr by one quadrant.

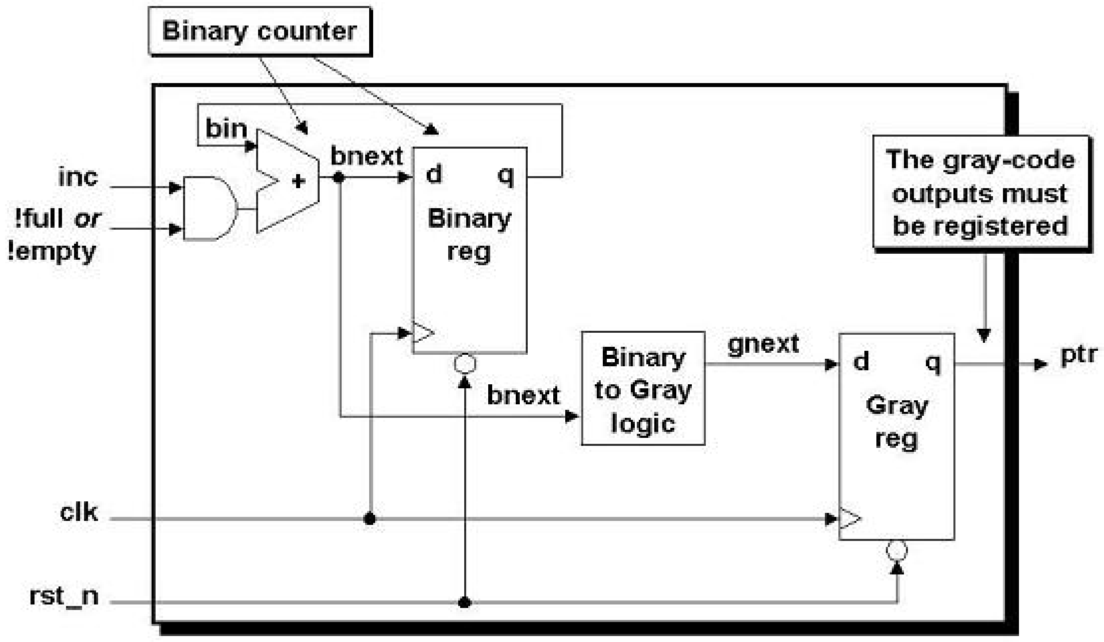
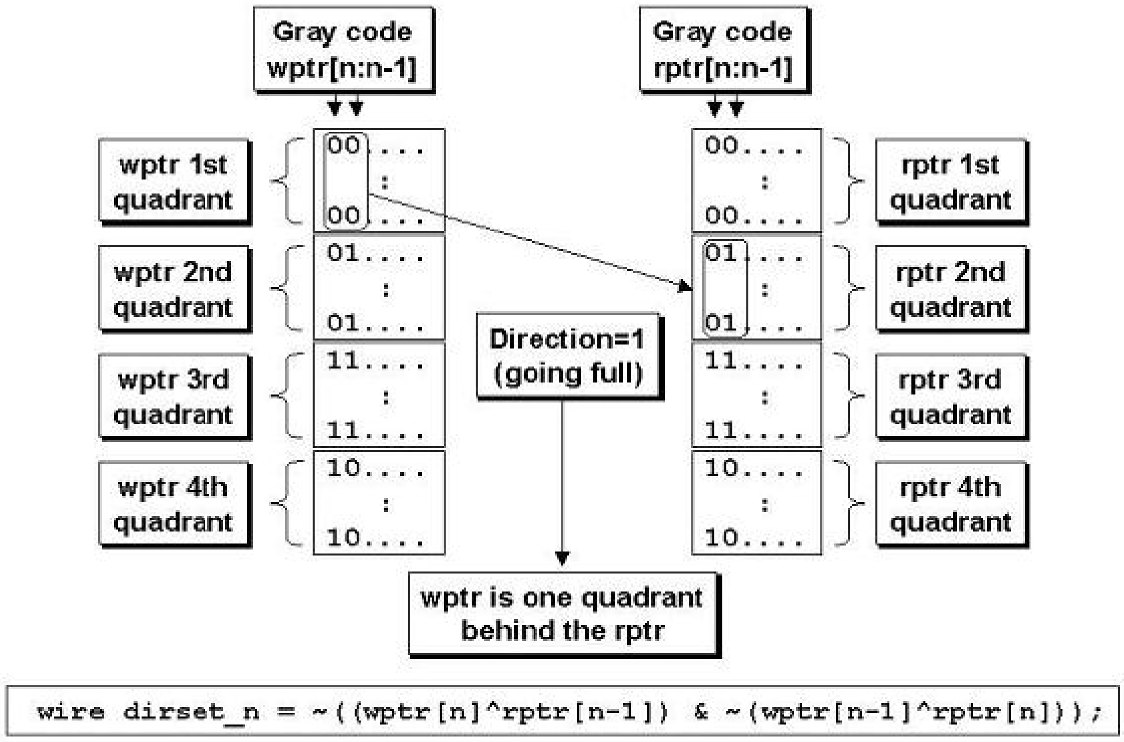


Fig 10.4: Dual n-bit gray code counter style# 2



```
wire dirset_n = ~((wptr[n]^rptr[n-1]) & ~(wptr[n-1]^rptr[n]));
```

Fig 10.5: FIFO is going full because the wptr trails the rptr by one quadrant.

.

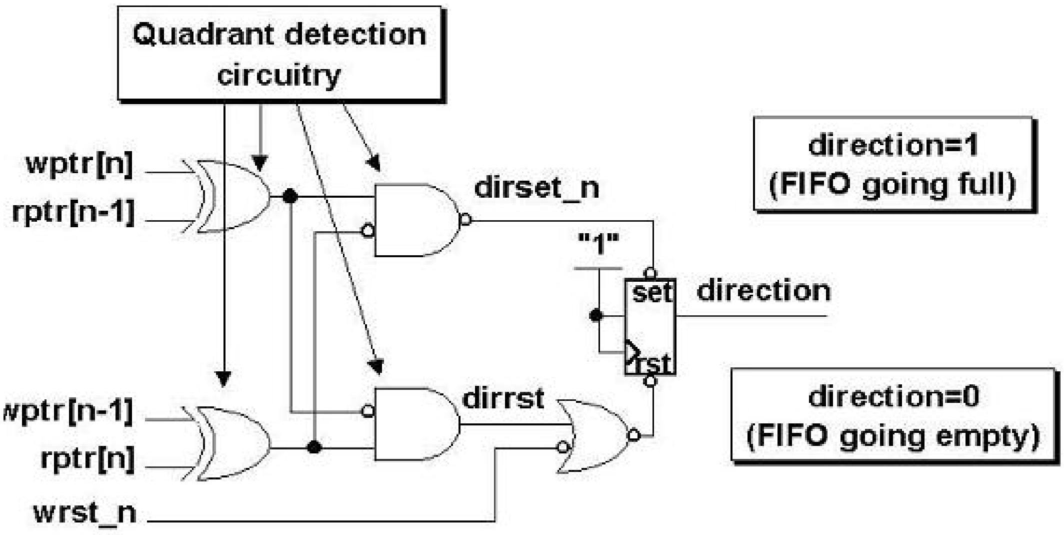

Quadrant detection circuitry
wptr[n]
rptr[n-1]
dirset_n
"1"
set
direction
rst
dirrst
wptr[n-1]
rptr[n]
wrst_n
direction=1 (FIFO going full)
direction=0 (FIFO going empty)

Fig 10.6:FIFO Direction Quadrant Detection Circuitry.

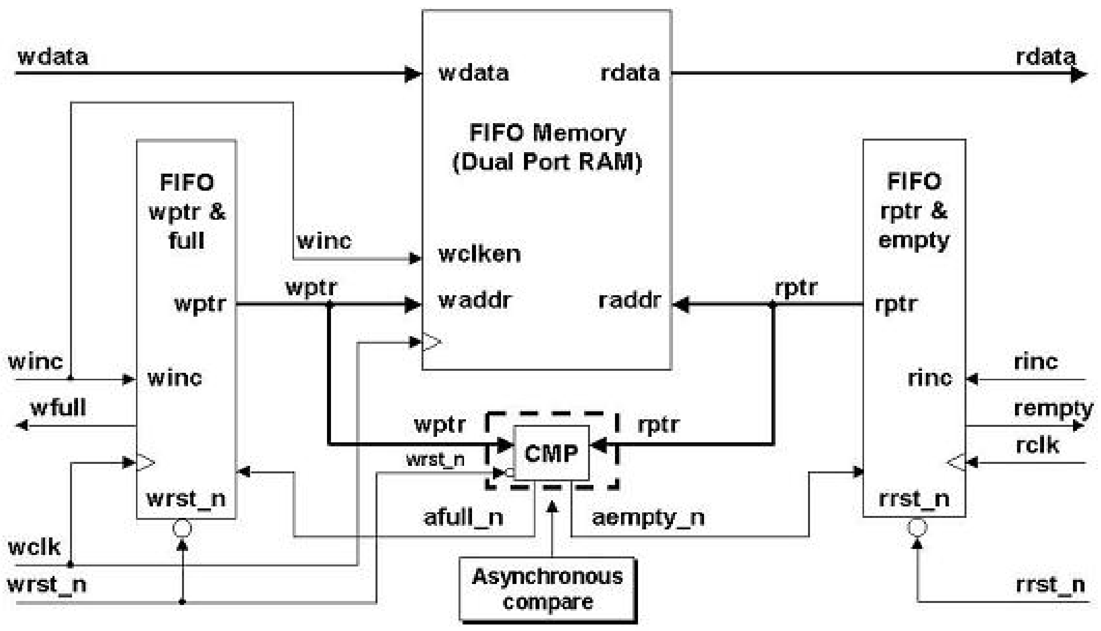


Fig 10.7: FIFO2 partitioning with asynchronous pointer comparison logic

The top-level wrapper-module includes all clock domains. The top module is only used as a wrapper to instantiate all of the other FIFO modules used in the design. If this FIFO is used as part of a larger ASIC or FPGA design, this top-level wrapper would probably be discarded to permit grouping of the other FIFO modules into their respective clock domains for improved synthesis and static timing analysis.

The FIFO memory buffer is accessed by both the write and read clock domains. This buffer is most likely an instantiated, synchronous dual-port RAM. Other memory styles can be adapted to function as the FIFO buffer.

Asynchronous pointer-comparison module used to generate signals that control assertion of the asynchronous "full" and "empty" status bits. This module only contains combinational comparison logic. No sequential logic is included in this module.

• **rptr_empty** module is mostly synchronous to the read-clock domain and contains the FIFO read pointer and empty-flag logic. Assertion of the **aempty_n** signal (an input to this module) is synchronous to the **rclk**-domain, since **aempty_n** can only be asserted when the **rptr** incremented, but de-assertion of the **aempty_n** signal happens when the **wptr** increments, which is asynchronous to **rclk**.

• **wptr_full** module is mostly synchronous to the write-clock domain and contains the FIFO write pointer and full-flag logic. Assertion of the **afull_n** signal (an input to this module) is synchronous to the **wclk**-domain, since **afull_n** can only be asserted when the **wptr** incremented (and **wrst_n**), but de-assertion of the **afull_n** signal happens when the **rptr** increments, which is asynchronous to **wclk.**

10.3 RTL implementation for FIFO

The fifo top-level module is a parameterized module with all sub-blocks instantiated following safe coding practices using named port connections.

The FIFO memory buffer could be an instantiated ASIC or FPGA dual-port, synchronous memory device. The memory buffer could also be synthesized to ASIC or FPGA registers using the RTL code in this module.

Async_cmp is an asynchronous comparison module, used to compare the read and write pointers to detect full and empty conditions.

10.3.1 Asynchronous generation of full and empty

In the **async_cmp** shown, **aempty_n** and **afull_n** are the asynchronously decoded signals. The **aempty_n** signal is asserted on the rising edge of an **rclk**, but is de-asserted on the rising edge of a **wclk**. Similarly, the **afull_n** signal is asserted on a **wclk** and removed on an **rclk**.

The empty signal will be used to stop the next read operation, and the leading edge of **aempty_n** is properly synchronous with the read clock, but the trailing edge needs to be synchronized to the read clock. This is done in a two-stage synchronizer that generates **rempty**.

The **wfull** signal is generated in the symmetrically equivalent way.

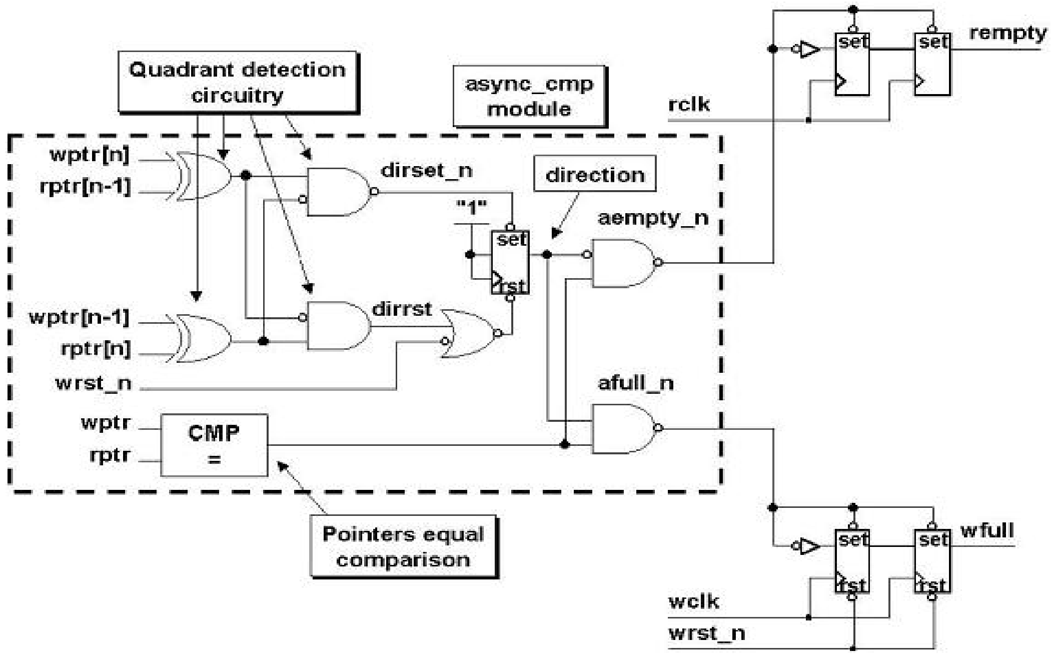


Fig 10.8: Asynchronous pointer comparison for full and empty.

10.3.2 Resetting the FIFO

The first FIFO event of interest takes place on a FIFO-reset operation. When the FIFO is reset, four important things happen within the **async_cmp** module and accompanying full and empty synchronizers of the **wptr_full** and **rptr_empty** modules the connections between the **async_cmp**, **wptr_full** and **rptr_empty** modules:

1. The reset signal directly clears the **wfull** flag. The **rempty** flag is not cleared by a reset.
2. The reset signal clears both FIFO pointers, so the pointer comparator asserts that the pointers are equal.
3. The reset clears the **direction** bit.
4. With the pointers equal and the **direction** bit cleared, the **aempty_n** bit is asserted, which presets the **rempty** flag.

10.3.3 FIFO-writes & FIFO full

The second FIFO operational event of interest takes place when a FIFO-write operation takes place and the **wptr** is incremented. At this point, the FIFO pointers are no longer equal so the **aempty_n** signal is de-asserted, releasing the preset control of the **rempty** flip-flops. After two rising edges on **rclk**, the FIFO will de-assert the **rempty** signal. Because the de-assertion of **aempty_n** happens on a rising **wclk** and because the **rempty** signal is clocked by the **rclk**, the two-flip-flop synchronizer is required to remove metastability that could be generated by the first **rempty** flip-flop.

The second FIFO operational event of interest takes place when the **wptr** increments into the next Gray code quadrant beyond the **rptr** (see discussion of Gray code quadrants). The **direction** bit is cleared (but it was already clear).

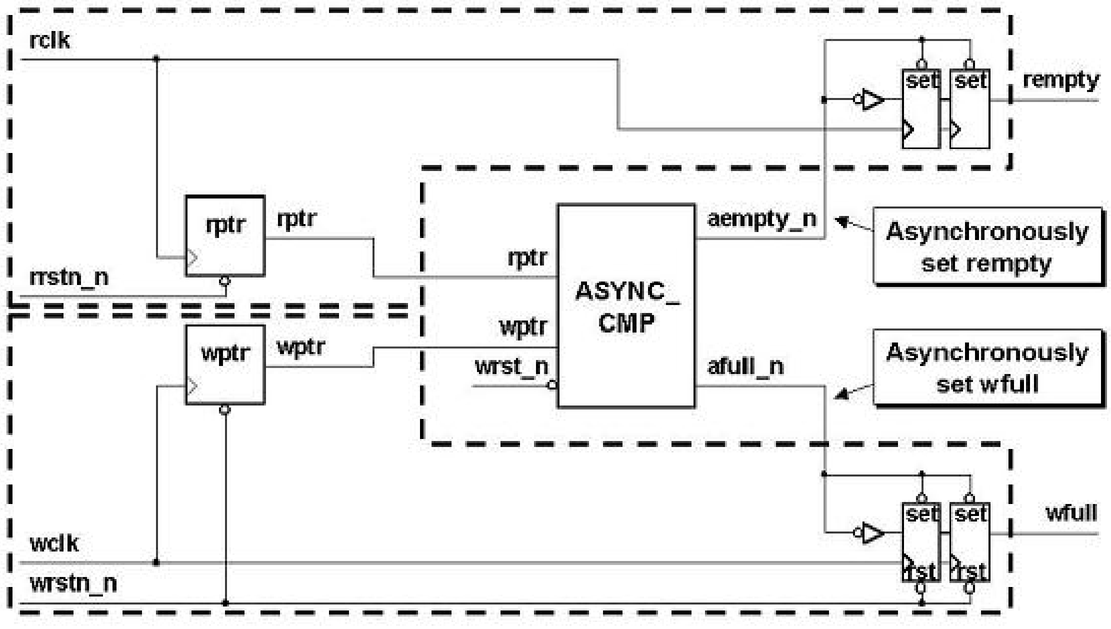


Fig 10.9: async_cmp module connection to rptr_empty and wptr_full modules.

The third FIFO operational event of interest occurs when the **wptr** is within one quadrant of catching up to the **rptr** as described. When this happens, the **dirset_n** bit is asserted low, which sets the **direction** bit high. This means that the **direction** bit is set long before the FIFO is full and is not timing critical to assertion of the **afull_n** signal.

The fourth FIFO operational event of interest is when the **wptr** catches up to the **rptr** (and the **direction** bit is set). When this happens, the **afull_n** signal presets the **wfull** flip-flops. The **afull_n** signal is asserted on a FIFO-write operation and is synchronous to the rising edge of the **wclk**; therefore, asserting full is synchronous to the **wclk**.

The fifth FIFO operational event of interest is when a FIFO-read operation takes place and the **rptr** is incremented. At this point, the FIFO pointers are no longer equal so the **afull_n** signal is de-asserted, releasing the preset control of the **wfull** flip-flops. After two rising edges on **wclk**, the FIFO will de-assert the **wfull** signal. Because the de-assertion of **afull_n** happens on a rising **rclk** and because the **wfull** signal is clocked by the **wclk**, the two-flip-flop synchronizer is required to remove metastability that could be generated by the first **wfull** flip-flop capturing the inverted and asynchronously generated **afull_n** data input.

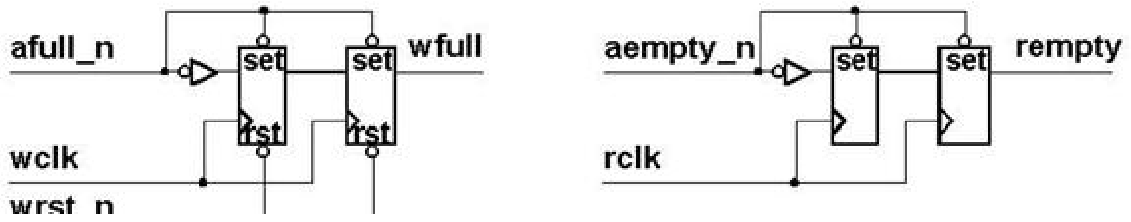


Fig 10.10: asynchronous empty and full generation.

During operation, **wfull** is generated synchronous to the write clock in a similar way that **rempty** is generated synchronous to the read clock. The **afull_n** signal is asserted as a result of a write clock, and the leading (falling) edge is thus naturally synchronous to the write clock. The trailing (rising) edge is, however caused by the read clock, and must, therefore be synchronized to the write clock. The same timing issues related to the setting of the full flag also apply to the setting of the empty flag.

10.3.4 FIFO-reads & FIFO empty

The sixth FIFO operational event of interest takes place when the **rptr** increments into the next Gray code quadrant beyond the **wptr**. The **direction** bit is again set (but it was already set).

The seventh FIFO operational event of interest occurs when the **rptr** is within one quadrant of catching up to the **wptr**. When this happens, the **dirrst** bit is asserted high, which clears the **direction** bit. This means that the **direction** bit is cleared long before the FIFO is empty and is not timing critical to assertion of the **aempty_n** signal.

The eighth FIFO operational event of interest is when the **rptr** catches up to the **wptr** (and the **direction** bit is zero). When this happens, the **aempty_n** signal presets the **rempty** flip-flops. The **aempty_n** signal is asserted on a FIFO-read operation and is synchronous to the rising edge of the **rclk**; therefore, asserting empty is synchronous to the **rclk**. Finally, when a FIFO-write operation takes place and the **wptr** is incremented. At this point, the FIFO pointers are no longer equal so the **aempty_n** signal is de-asserted, releasing the preset control of the **rempty** flip-flops. After two rising edges on **rclk**, the FIFO will de-assert the **rempty** signal. Because the de-assertion of **aempty_n** happens on a rising **wclk** and because the **rempty** signal is clocked by the **rclk**, the two-flip-flop synchronizer is required to remove metastability that could be generated by the first **rempty** flip-flop.

10.3.5 Full and empty critical timing paths

Using the asynchronous comparison technique, there are critical timing paths associated with the generation of both the **rempty** and **wfull** signals.

The **rempty** critical timing path, consists of (1) the **rclk**-to-q incrementing of the **rptr**, (2) comparison logic of the **rptr** to the **wptr**, (3) combining the comparator output with the direction latch output to generate the **aempty**_n signal, (4) presetting the **rempty** signal, (5) any logic that is driven by the **rempty** signal, and (6) resultant signals meeting the setup time of any down-stream flip-flops clocked within the **rclk** domain. This critical timing path has a symmetrically equivalent critical timing path for the generation of the **wfull** signal.

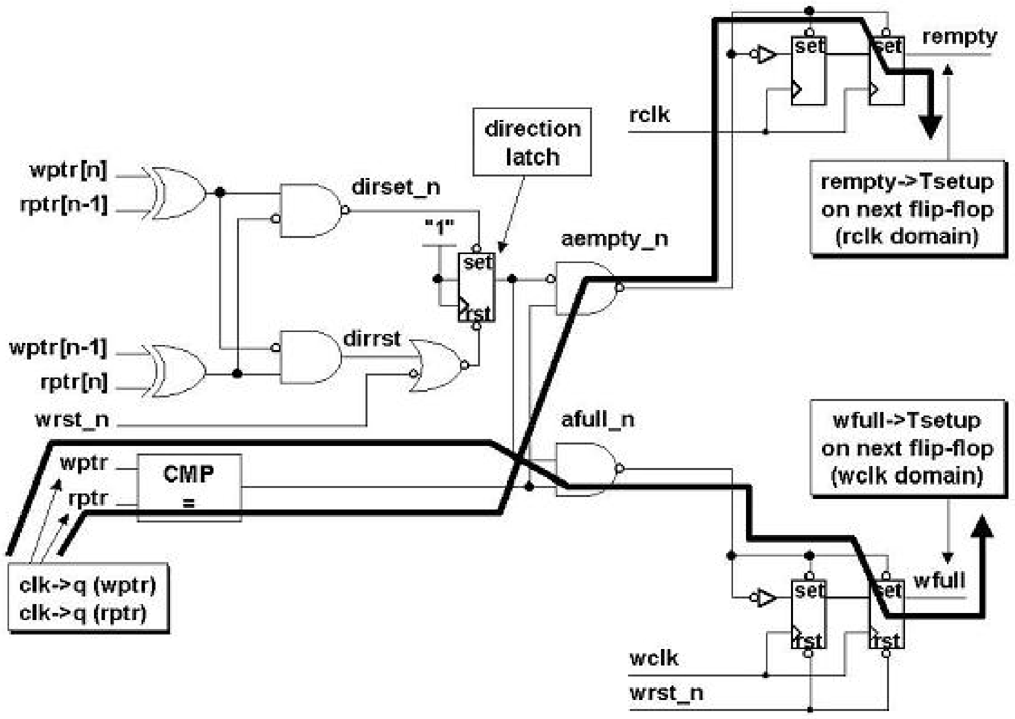


Fig 10.11: critical timing paths for asserting rempty and wfull.

The read pointer is an n-bit Gray code counter. The FIFO **rempty** output is asserted when the **aempty_n** signal goes low and the **rempty** output is de-asserted on the second rising **rclk** edge after **aempty_n** goes high (a rare metastable state could cause the **rempty** output to be de-asserted on the third rising **rclk** edge). This module is completely synchronous to the **rclk** for simplified static timing analysis, except for the **aempty_n** input, which is de-asserted asynchronously to the **rclk**.

The last always block in this module is the asynchronously preset **rempty** signal generation. The presetting signal is the **aempty_n** input, which is asserted when the **rptr** is incremented by the **rclk** (synchronous to this block) as long as the **rempty** critical timing path is satisfied. Removal of the **rempty** signal occurs when the write pointer increments, which is asynchronous to the **rclk** domain. Because reset removal is asynchronous to the **rclk** domain, a two-flip-flop synchronizer is required to synchronize **aempty_n** removal to the **rclk** domain.

This module encloses all of the FIFO logic that is generated within the write clock domain (except synchronizers).

The write pointer is an n-bit Gray code counter. The FIFO **wfull** output is asserted when the **afull_n** signal goes low and the **wfull** output is de-asserted on the second rising **wclk** edge after **afull_n** goes high (a rare metastable state could cause the **wfull** output to be de-asserted on the third rising **wclk** edge). This module is completely synchronous to the **wclk** for simplified static timing analysis, except for the **afull_n** input, which is de-asserted asynchronously to the **wclk**.

The last always block in this module is the asynchronously preset **wfull** signal generation. The presetting signal is the **afull_n** input, which is asserted when the **wptr** is incremented by the **wclk** (synchronous to this block) as long as the **wfull** critical timing path is satisfied. Removal of the **wfull** signal occurs when the read pointer increments, which is asynchronous to the **wclk** domain. Because reset removal is asynchronous to the **wclk** domain, a two-flip-flop synchronizer is required to synchronize **afull_n** removal to the **wclk** domain. The **wfull** signal must also go low when the FIFO is reset.

10.3.6 Asynchronous concerns, questions and answers

This section captures a number of the questions, concerns and answers that lead to believe this coding style does indeed work.

Generation of the **aempty_n** control signal is straightforward. Whenever the read pointer (**rptr**) equals the write pointer (**wptr**), and the **direction** latch is clear, the FIFO is empty.

The empty flag is used only in the read clock domain and since the read pointer, incremented by a read clock, causes the empty flag to be set, assertion of the empty flag is always synchronous in the read clock domain. As long as the empty flag meets the critical empty-assertion timing path, there is no synchronization problems associated with asserting the empty flag.

The de-assertion of **aempty_n** is caused by the write clock incrementing the write pointer, and is thus unrelated to the read clock. The de-assertion of **aempty_n** must, therefore, be synchronized in a dual-flip-flop synchronizer, clocked by the read clock. The first flip-flop is subject to metastability but the second flip-flop is included to wait for the metastability to subside, just like any other multi-clock synchronizer.

Since **aempty_n** is started by one clock and terminated by the other, it has an undefined duration, and might even be a runt pulse. A runt pulse is a Low-High-Low signal transition where the transition to High may or may not pass through the logic-"1" threshold level of the logic family being used.

If the **aempty_n** control signal is a runt pulse, there are four possible scenarios that should be addressed:

(1) The runt signal is not recognized by the **rempty** flip-flops and empty is not asserted. This is not a problem.

(2) The runt pulse might preset the first synchronizer flip-flop, but not the second flip-flop. This is highly unlikely, but would result in an unnecessary, but properly synchronized **rempty** output, that will show up on the output of the second flip-flop one read clock later. This is not a problem.

(3) The runt pulse might preset the second synchronizer flip-flop, but not the first flip-flop. This is highly unlikely, but would result in an unnecessary, but properly synchronized **rempty** output (as long as the empty critical timing is met), that will be set on the output of the second flip-flop until the next read clock, when it will be cleared by the zero from the first flip-flop. This is not a problem.

(4) The most likely case is that the runt pulse sets both flip-flops, thus creating a properly synchronized rempty output that is two read-clock periods long. The longer duration is caused by the two-flip-flop synchronizer (to avoid metastable problems as described below). This is not a problem.

The runt pulse cannot have any effect on the synchronizer data-input, since an **aempty_n** runt pulse can only occur immediately after a read clock edge, thus long before the next read clock edge (as long as critical timing is met).

The **aempty_n** signal might also stay high longer and go low at any moment, even perhaps coincident with the next read clock edge. If it goes low well before the set-up time of the first synchronize flip-flop, the result is like scenario (4) above. If it goes low well after the set-up time, the synchronizer will stretch **rempty** by one more read clock period.

If **aempty_n** goes low within the metstability-catching set-up time window, the first synchronizer flip-flop output will be indeterminate for a few nanoseconds, but will then be either high or low. In either case, the output of the second synchronizer flip-flop will create the appropriate synchronized **rempty** output.

The next question is, what happens if the write clock de-asserts the **aempty_n** signal coincident with the rising **rclk** on the dual synchronizer? The first flip-flop could go metastable, which is why there is a second flip-flop in the dual synchronizer.
But the removal of the setting signal on the second flip-flop will violate the recovery time of the second flip-flop.

Will this cause the second flip-flop to go metastable? We do not believe this can happen because the preset to the flip-flop forced the output high and the input to the same flip-flop is already high, which we believe is not subject to recovery time instability on the flip-flop.

Last question. Can a runt-preset pulse, where the trailing edge of the runt pulse is caused by the **wclk**, preset the second synchronizer flip-flop in close proximity to a rising **rclk**, violate the preset recovery time and cause metastability on the output of the second flip-flop? The answer is no as long as the **aempty_n** critical timing path is met. Assuming that critical timing is met, the **aempty_n** signal going low should occur shortly after a rising **rclk** and well before the rising edge of the second flip-flop, so runt pulses can only occur well before the rising edge of an **rclk**.

Again, symmetrically equivalent scenarios and arguments can be made about the generation of the **wfull** flag.

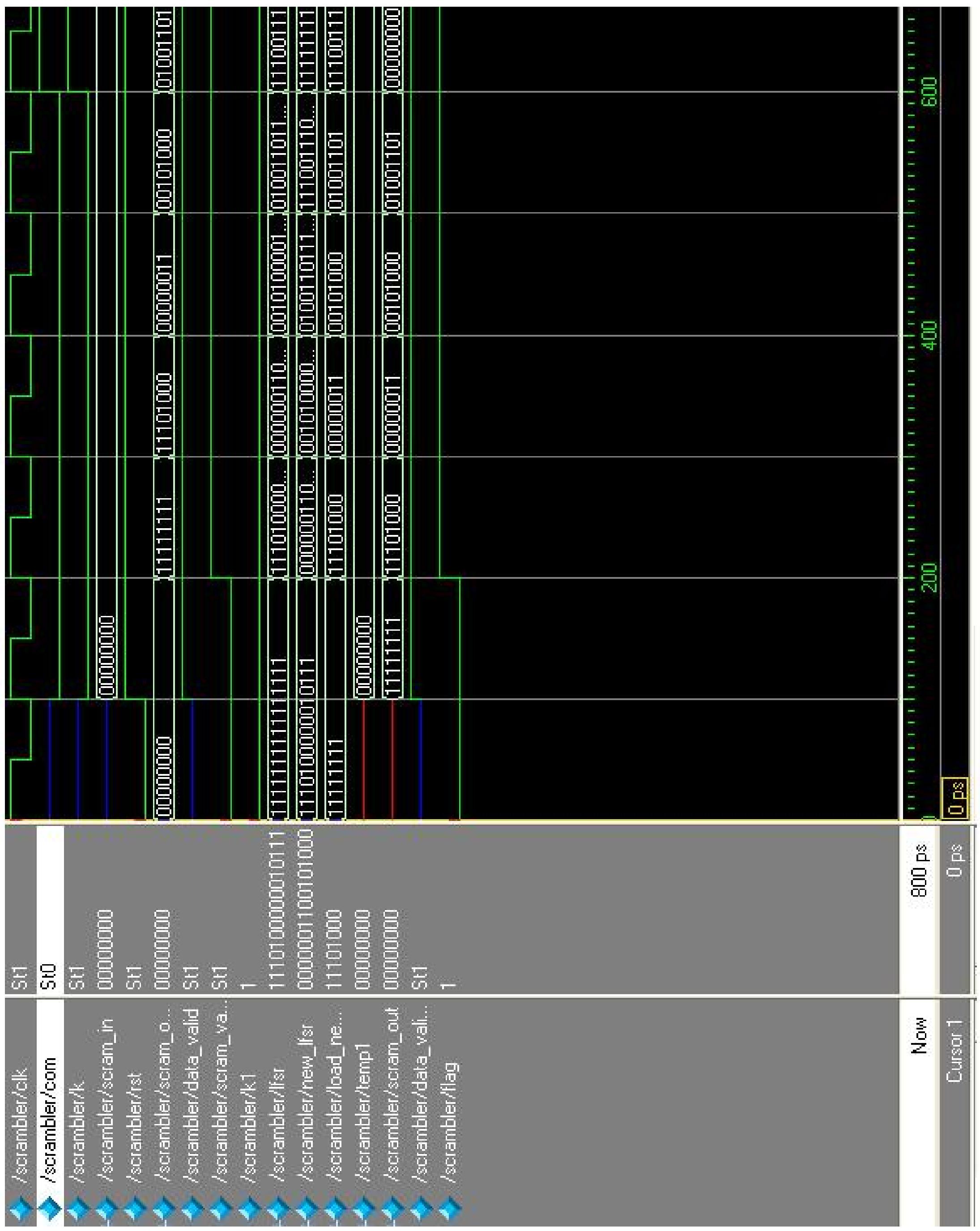
CHAPTER 11

SIMULATIONS

The Verilog code is executed using XILINX 9.1i and the simulations are obtained using MODELSIM. The simulations along with the brief descriptions are given in this chapter.

11.1 SCRAMBLER

The figure 11.1 shows the simulation for the Scrambler. An active low reset is applied to the circuit. When the signal data_valid goes high, it indicates that the valid data is available at the scrambler input. After a clock delay the signal scram_valid goes high indicating the valid scrambler output. K is an external signal from the higher layer.



```
/scrambler/clk          St1
/scrambler/com          St0
/scrambler/k            St1
/scrambler/scram_in     00000000
/scrambler/rst          St1
/scrambler/scram_o...   00000000
/scrambler/data_valid   St1
/scrambler/scram_va...  St1
/scrambler/k1           1
/scrambler/lfsr         1110100000010111
/scrambler/new_lfsr     0000001100101000
/scrambler/load_ne...   11101000
/scrambler/temp1        00000000
/scrambler/scram_out    00000000
/scrambler/data_vali... St1
/scrambler/flag         1
Now                     800 ps
Cursor 1                0 ps
```

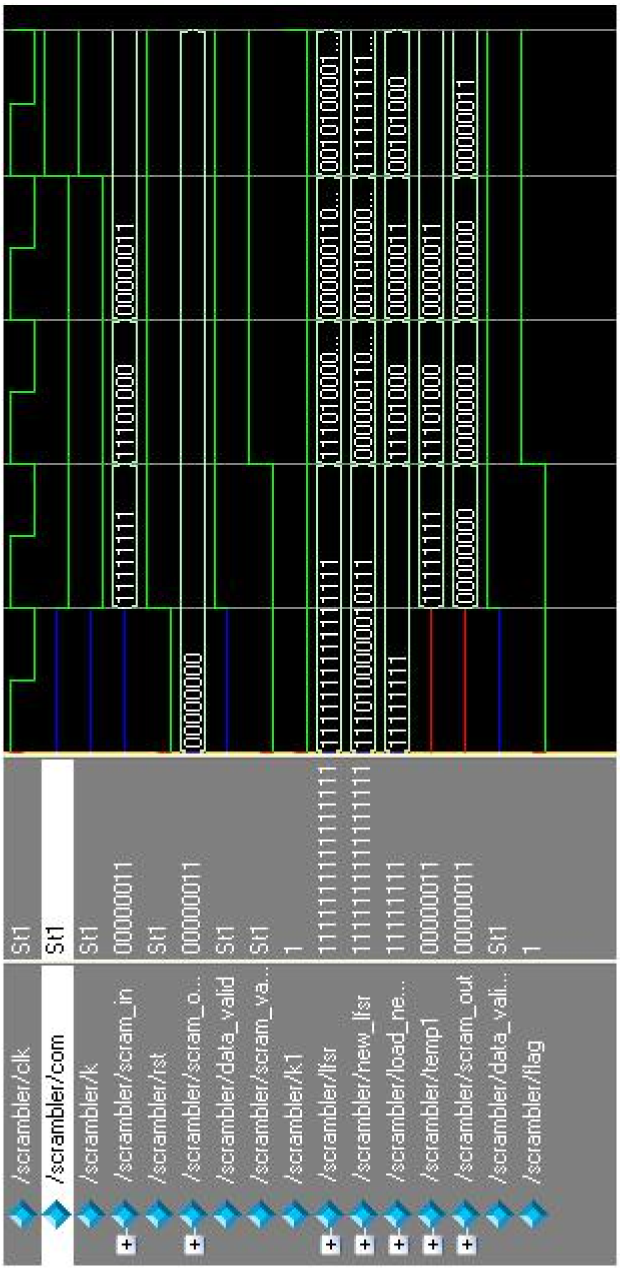
Fig 11.1 Scrambler simulations

Input sequence: 00000000b

Output sequence: scram_out1.

It can be observed that for K=COM=1, LFSR=FFFFH.

11.2 DESCRAMBLER



```
/scrambler/clk          St1
/scrambler/com          St1
/scrambler/k            St1
/scrambler/scram_in     00000011
/scrambler/rst          St1
/scrambler/scram_o...   00000011
/scrambler/data_valid   St1
/scrambler/scram_va...  St1
/scrambler/k1           1
/scrambler/lfsr         1111111111111111
/scrambler/new_lfsr     1111111111111111
/scrambler/load_ne...   11111111
/scrambler/temp1        00000011
/scrambler/scram_out    00000011
/scrambler/data_vali... St1
/scrambler/flag         1
```

Fig 11.2: Descrambler Simulations

Output: 00000000b.
It can be observed that by giving the output of the scrambler as the input to descrambler, input data is recovered.

11.3 ENCODER

The figure 11.3 shows the simulation for the encoder. An active low reset is applied to the circuit. The output is purposefully delayed by a clock cycle, in order to accommodate the maximum combinational block delay. When the signal scram_valid goes high, it indicates that the valid data is available at the encoder input. After a clock delay the signal enco_valid goes high indicating the valid encoder output. K is an external signal from the higher layer. INDISP is the initial disparity. Once the signal scram_valid goes low, the signal enco_valid goes low after one clock cycle. The other signals explained in the design are also visible in the figure.

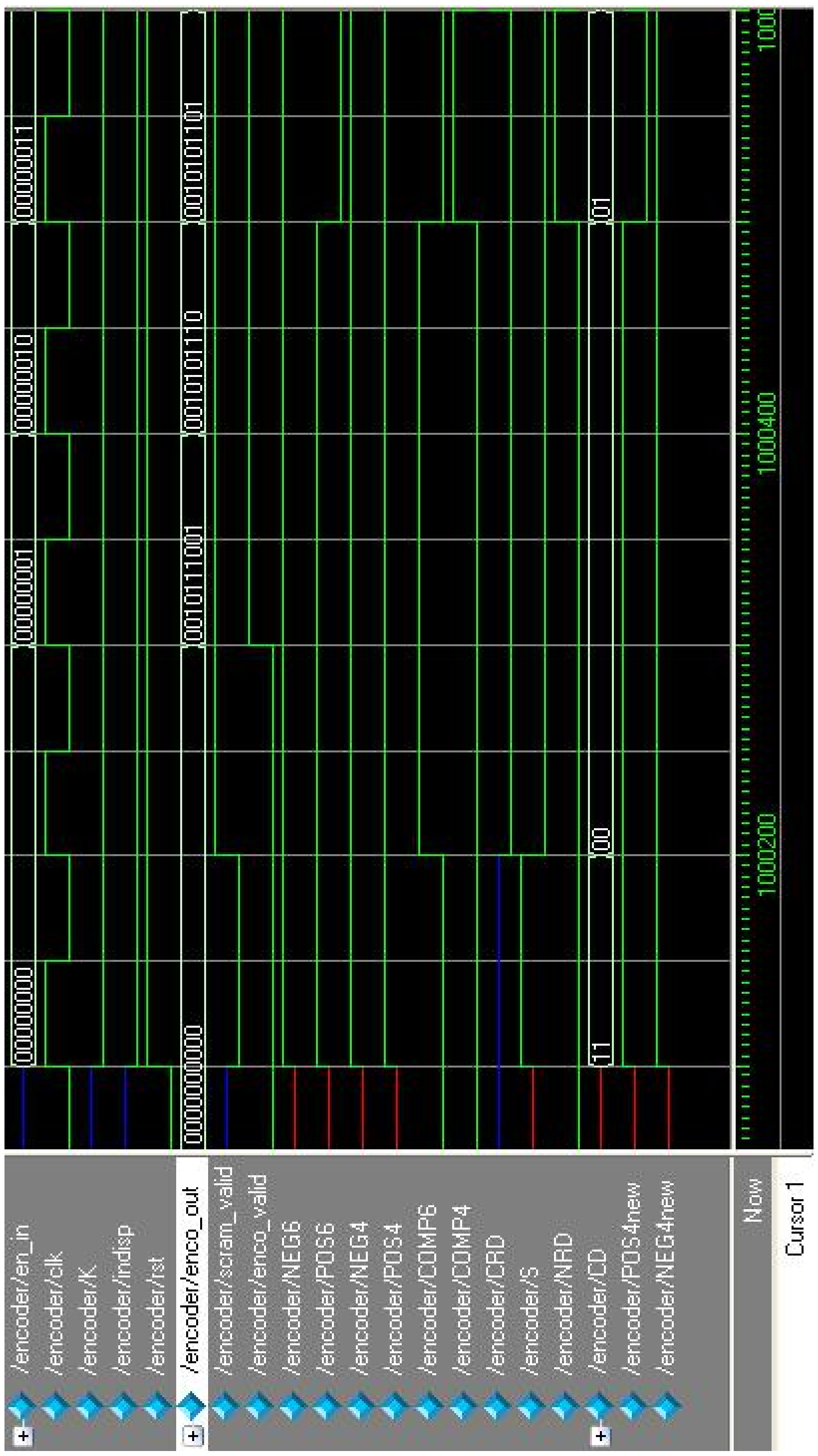
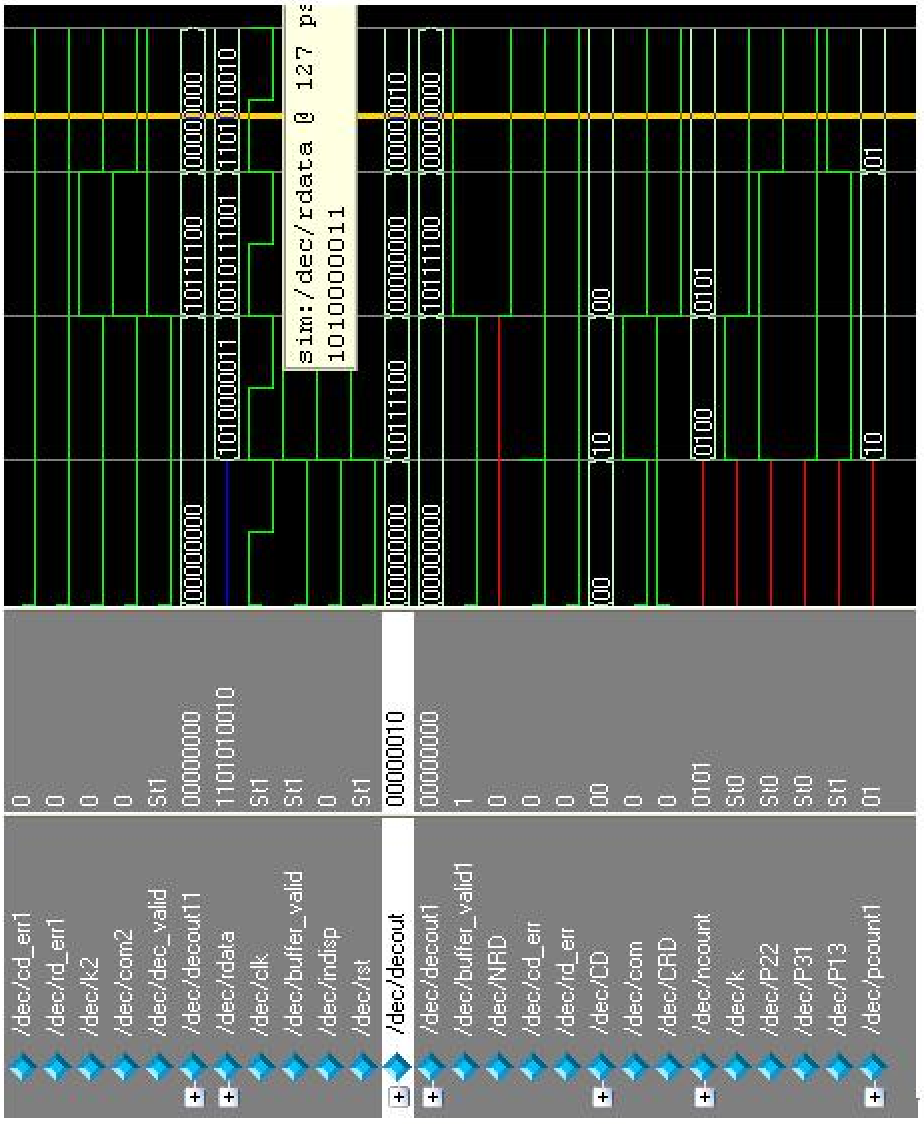


Figure 11.3: Encoder simulation

11.4 DECODER

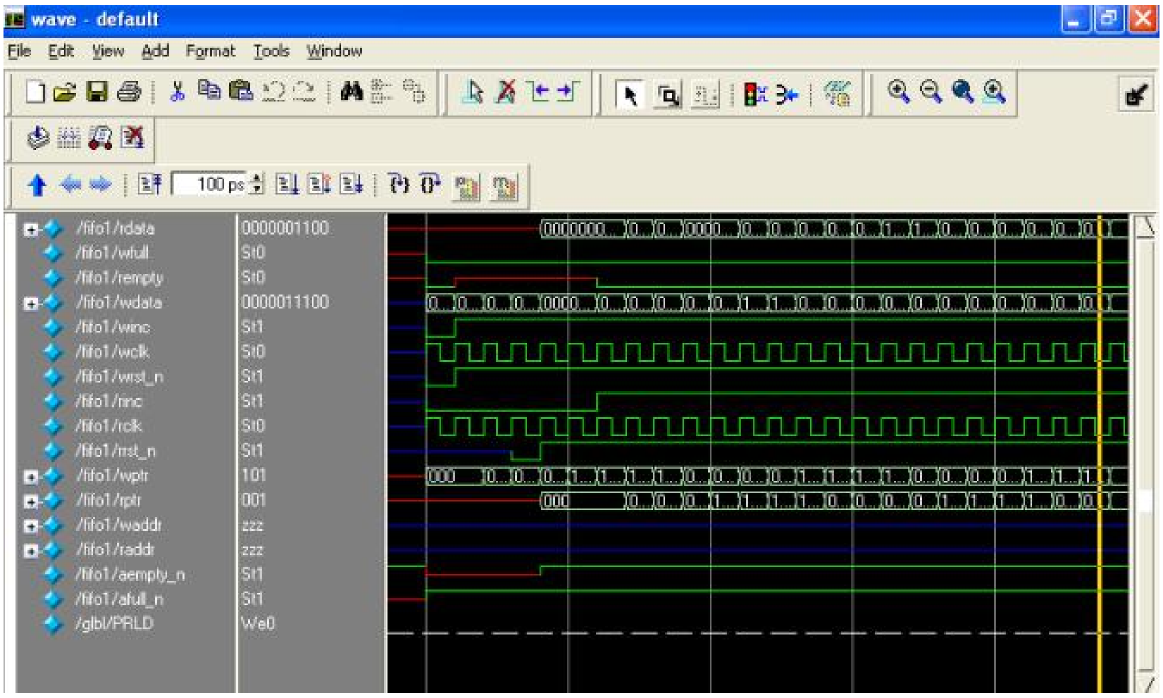


```
/dec/cd_err1         0
/dec/rd_err1         0
/dec/k2              0
/dec/com2            0
/dec/dec_valid       St1
/dec/decout11        00000000
/dec/rdata           1101010010
/dec/clk             St1
/dec/buffer_valid    St1
/dec/indisp          0
/dec/rst             St1
/dec/decout          00000010
/dec/decout1         00000000
/dec/buffer_valid1   1
/dec/NRD             0
/dec/cd_err          0
/dec/rd_err          0
/dec/CD              00
/dec/com             0
/dec/CRD             0
/dec/ncount          0101
/dec/k               St0
/dec/P22             St0
/dec/P31             St0
/dec/P13             St1
/dec/pcount1         01

sim:/dec/rdata @ 127 ps
1010000011
```

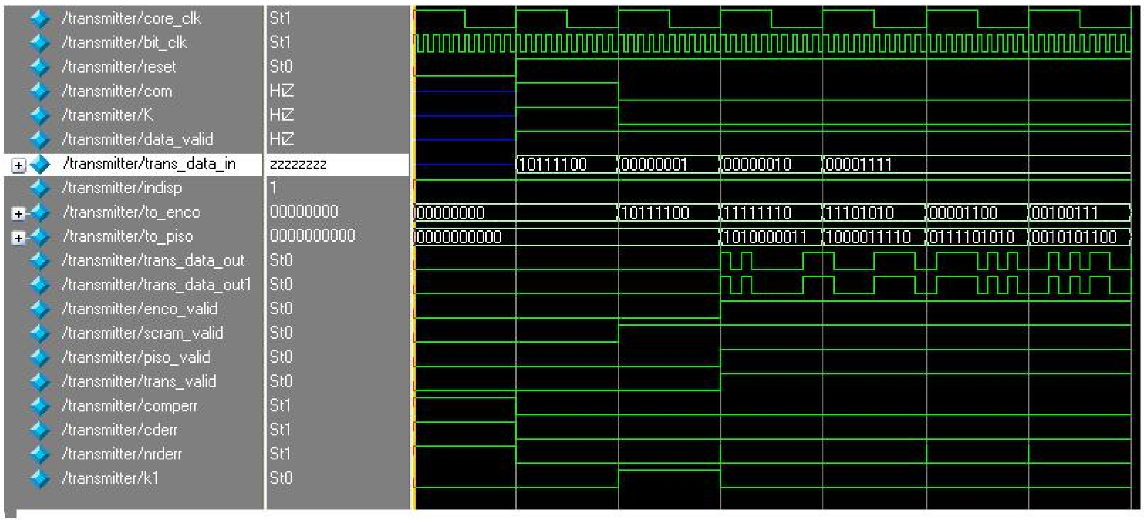
Fig 11.4 Decoder simulations

11.5 FIFO BUFFER



```
/fifo1/rdata      0000001100
/fifo1/wfull      St0
/fifo1/rempty     St0
/fifo1/wdata      0000011100
/fifo1/winc       St1
/fifo1/wclk       St0
/fifo1/wrst_n     St1
/fifo1/rinc       St1
/fifo1/rclk       St0
/fifo1/rrst_n     St1
/fifo1/wptr       101
/fifo1/rptr       001
/fifo1/waddr      zzz
/fifo1/raddr      zzz
/fifo1/aempty_n   St1
/fifo1/afull_n    St1
/glbl/PRLD        We0
```

11.6 TRANSMITTER



```
/transmitter/core_clk         St1
/transmitter/bit_clk          St1
/transmitter/reset            St0
/transmitter/com              HiZ
/transmitter/K                HiZ
/transmitter/data_valid       HiZ
/transmitter/trans_data_in    zzzzzzzz
/transmitter/indisp           1
/transmitter/to_enco          00000000
/transmitter/to_piso          0000000000
/transmitter/trans_data_out   St0
/transmitter/trans_data_out1  St0
/transmitter/enco_valid       St0
/transmitter/scram_valid      St0
/transmitter/piso_valid       St0
/transmitter/trans_valid      St0
/transmitter/comperr          St1
/transmitter/cderr            St1
/transmitter/nrderr           St1
/transmitter/k1               St0
```

Fig 11.6: Transmitter Simulations

Output: trans_data_out (serial sequence).

11.7 RECEIVER

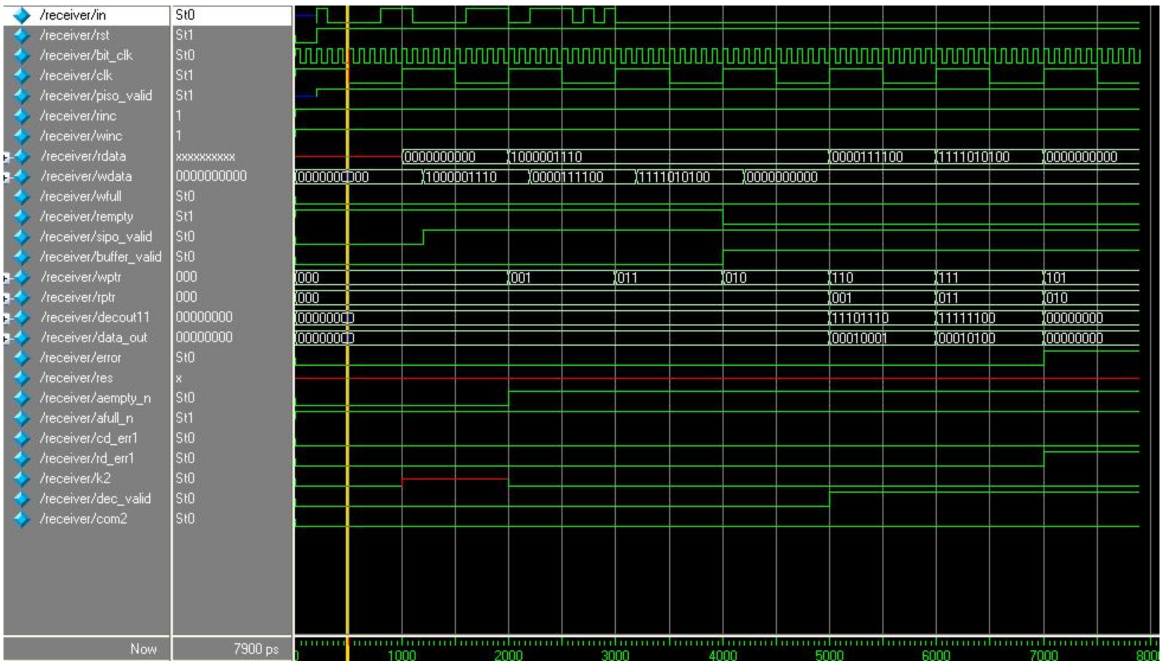


Fig 11.7: Receiver Simulations

11.8 PHYSICAL LAYER

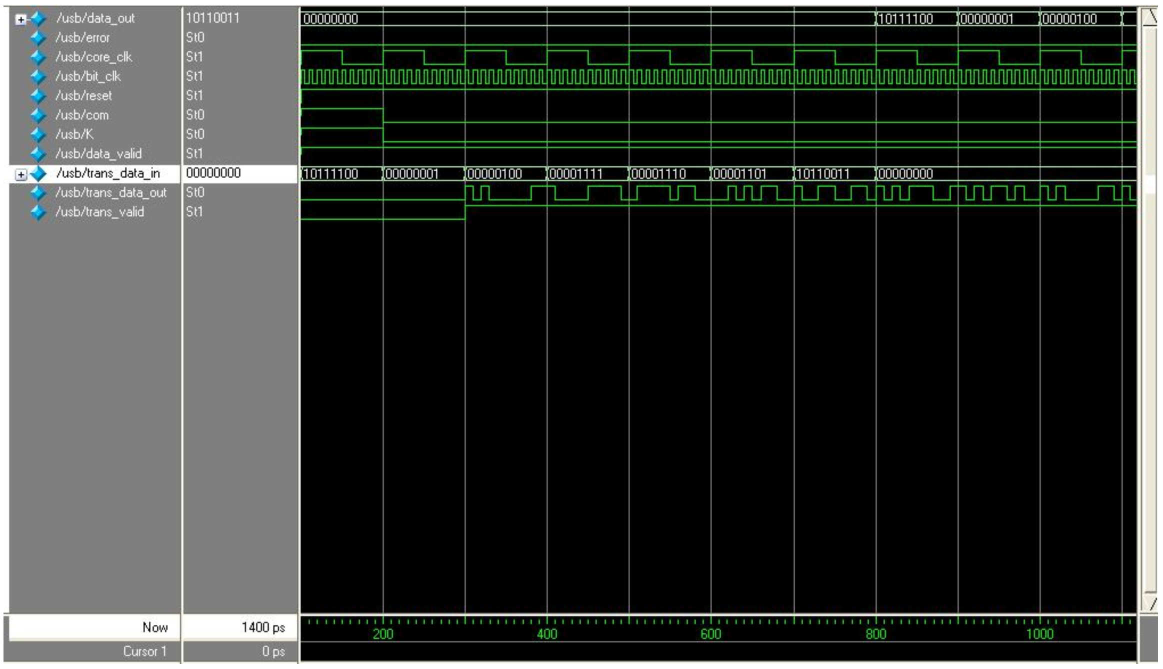


Fig 11.7: Physical Layer Simulations

It can be observed that trans_data_in=data_out.

CHAPTER 12

CONCLUSION

The physical layer of USB 3.0 Super-speed bus architecture was designed and implemented using one of the most widely used hardware languages, Verilog in a period of less than 4-months. This implementation is completed even before products using such a protocol have reached the market. The project was started much later than Nov, 12, 2008 when the specification was introduced and completed before May, 21, 2009 when USB Implementers' Forum reviewed USB 3.0 protocol in Tokyo, Japan. We conclude this work counting on the works remaining for the complete USB 3.0 system level implementation:

- Framework, Protocol, Link layer and software implementation which takes not less than 6-months.

CHAPTER 13

BIBLIOGRAPHY

- Universal Serial Bus 3.0 specification, Revision 1.0, November 12, 2008 available at www.usb.org.
- A DC-Balanced, Partitioned –Block, 8B/10B Transmission Code, A.X. Widmer, P.A. Franaszek, IBM J.Res. Develop. Vol 27. N0.5. September 1983.
- Implementing an 8b/10b encoder/decoder for Gigabit Ethernet in the Actel SX-FPGA family.
- www.wikipedia.org
- www.intel.com

APPENDIX A Synthesis Information

Final Results

RTL Top Level Output File Name	: usb.ngr
Top Level Output File Name	: usb
Output Format	: NGC
Optimization Goal	: Speed
Keep Hierarchy	: NO

Design Statistics

# IOs	: 24

Cell Usage:

# BELS	: 350
# INV	: 3
# LUT2	: 17
# LUT2_L	: 2
# LUT3	: 91
# LUT3_D	: 4
# LUT3_L	: 6
# LUT4	: 150
# LUT4_D	: 12
# LUT4_L	: 17
# MUXF5	: 38
# MUXF6	: 10
# FlipFlops/Latches	: 194
# FDC	: 53
# FDCE	: 101
# FDCP	: 2
# FDE	: 2
# FDP	: 35
# FDPE	: 1
# Clock Buffers	: 2
# BUFG	: 1

```
#      BUFGP                   : 1
# IO Buffers                   : 23
#      IBUF                    : 12
#      OBUF                    : 11
```

Device utilization summary:

Selected Device: **3s400pq208-4 (Spartan3 FPGA)**

```
 Number of Slices:               195 out of   3584     5%
 Number of Slice Flip Flops:     194 out of   7168     2%
 Number of 4 input LUTs:         302 out of   7168     4%
 Number of IOs:                  24
 Number of bonded IOBs:           24 out of    141    17%
 Number of GCLKs:                  2 out of      8    25%
```

Timing Summary:

Speed Grade: -4

```
   Minimum period: 13.238ns (Maximum Frequency: 75.540MHz)
   Minimum input arrival time before clock: 3.378ns
   Maximum output required time after clock: 10.782ns
```

Partition Resource Summary:

No Partitions were found in this design.