STUDY, DESIGN AND SIMULATION OF FPGA BASED USB 2.0 DEVICE CONTROLLER

[1] *MS. PARUL BAHUGUNA CD*

[1]M.E. [VLSI & Embedded System Design] Student, Gujarat Technological University PG School, Ahmedabad, Gujarat.

parulbahuguna@ymail.com

ABSTRACT: *The Universal Serial Bus (USB) was introduced to the world of PC to solve the growing problem of how to connect peripherals to a PC. Embedded considerations for using USB in a design are split between need to implement functionality as a peripheral device, a host or both as in OTG. USB is host–centric tree topology where all devices are hooked to the host by a set of hubs. Each device supports up to 32 endpoints: 16 in and 16 out. USB support for embedded design, facilitating the implementation of both peripherals and host functionality. In this project the UBS protocols I will carry out study of USB 2.0 (HI-SPEED USB) protocols, logic design to implement the protocol and verifying the design with simulations. The main goal is to obtain the efficient FPGA based device controller IP core for USB 2.*

Keywords—USB2.0, Super-speed USB, OTG, FPGA, Verilog, CRC, NRZI, Bit-stuffing, PID, UTMI, Synopsys Design Compiler.

I: INTRODUCTION

USB 2.0 is an Industry-wide, host oriented protocol, employing serial bus, supporting up to 127 devices and hot insertion. USB 2.0 represents a great advance in speed while keeping a low-cost solution that supports transfer rates of up to 480 Mbps. With full support for real-time data, voice, audio, and video it is the chosen protocol for most PC peripherals today. Comprehension of various PC configurations and form factors make the USB a multifunctional protocol capable of servicing various solutions.

The USB is a generic protocol making its interface capable of quick diffusion into product. It improves PC's capability by enabling new classes of devices giving the USB a capability to be implemented in new developed devices, advancing with technology. Fully backward compatibility of USB 2.0 for devices built to previous versions of the USB specification.

The protocol engine of USB performs Sync generation and detection, NRZI encoding and decoding, bit-stuffing and d-stuffing, CRC check and generation, packet identifier (PID) generation, decoding and verification, address recognition and handshake evaluation and response. Acting on a received token and analysing the token's PID, address and endpoint number fields, the protocol engine can handle USB packets and transactions based on data sequencing and state machine logic.

Protocol Engine that I have carried out complies with High Speed USB 2.0 specification with a transfer rate of 480Mbps. Protocol Engine performs transaction to/from host. Protocol Engine supports four types of transactions: Control, Bulk, Isochronous, and Interrupt.

Main goal of the USB controller design is to implement it into FPGA which can be used in FPGA based embedded system (as an IP core).

II: USB 2.0 DEVICE CONTROLLER

A typical USB controller contains a USB transceiver, a serial interface engine, buffers to hold USB data, and registers to store configuration, status, and control information relating to USB communications.

The Transceiver

The USB transceiver provides a hardware interface between the device's USB connector and the circuits that control USB communications.

The Serial Interface Engine

The circuits that interface to the transceiver form a unit called the serial interface engine (SIE). The SIE typically handles the sending and receiving of data in transactions. A typical SIE does all of the following:

- Detect incoming packets.
- Send packets such as token, data and handshake.
- Detect and generate Start-of-Packet, End-of-Packet, Reset, and Resume signalling.
- Encode and decode data in the format required on the bus (NRZI with bit stuffing).
- Check and generate CRC codes.
- Check and generate Packet Identifiers (PIDs).
- Convert between USB's serial data and parallel data in registers or memory.

Buffers

USB controllers use buffers to store recently received data and data that's ready to be sent on the bus.

Buffers that hold transmitted or received data are often structured as FIFO (first in, first out) buffers. Each read of a receive FIFO returns the byte that has been in the buffer the longest. Each write to a transmit FIFO stores a byte that will transmit after all of the bytes already in the buffer have transmitted. Yao-Xuan-Lei and Jingcheng (2010) proposed that the data transfer is achieved in slave FIFO mode while the controlling commands uses microprocessors in FPGA's through endpoint-0[11].This offers high-speed data communication.

III: USB PACKET FORMAT

All USB data is sent serially, of course, and least significant bit (LSB) first. USB data transfer is essentially in the form of packets of data, sent back and forth between the host and peripheral devices. Initially, all packets are sent from the host, via the root hub and possibly more hubs, to devices. Some of those packets direct a device to send some packets in reply.

USB packets may consist of the following fields:

1. Sync field: All the packets start with this sync field. The sync field is 8 bits long at low and full speed or 32 bits long for high speed and is used to synchronize the clock of the receiver with that of the transmitter. The last two bits indicate where the PID fields starts.

2. PID field: This field (Packet ID) is used to identify the type of packet that is being sent. The PID is actually 4 bits; the byte consists of the 4-bit PID followed by its bit-wise complement, making an 8-bit PID in total. This redundancy helps detect errors. There is 4 bits to the PID, however to insure it is received correctly, the 4 bits are complemented and repeated, making an 8 bit PID in total.

3. ADDR field: The address field specifies which device the packet is designated for. Being 7 bits in length allows for 127 devices to be supported.

4. ENDP field: This field is made up of 4 bits, allowing 16 possible endpoints. Low speed devices however can only have 2 additional endpoints on top of the default pipe.

5. CRC field: Cyclic Redundancy Checks are performed on the data within the packet payload. All token packets have a 5-bit CRC while data packets have a 16-bit CRC.

6. EOP field: This indicates End of packet. This is signaled by a Single Ended Zero (SE0) for approximately 2 bit times followed by a J for 1 bit time.

The USB packets come in four basic types, each with a different format and CRC field:

1) Handshake packets
2) Token packet
3) Data packet

Handshake Packet

Handshake packets consist of a PID byte, and are generally sent in response to data packets. The three basic types of handshake packets are

1) ACK, indicating that data was successfully received.
2) NAK, indicating that the data cannot be received at this time and should be retried.
3) STALL, indicating that the device has an error and will never is able to successfully transfer data until some corrective action is performed.
4) NYET which indicates that a split transaction is not yet completes.
5) ERR handshake to indicate that a split transaction failed.

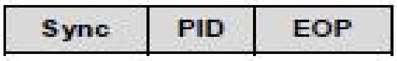


Fig. 1 Handshake Packet format

Token Packet

Token packets consist of a PID byte followed by 11 bits of address and a 5-bit CRC. Tokens are only sent by the host, not by a device.

There are three types of token packets.

1) **In token** - Informs the USB device that the host wishes to read information.
2) **Out token**- informs the USB device that the host wishes to send information.
3) **Setup token** - Used to begin control transfers.

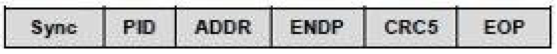


Fig. 2 Token Packet format

Data Packet

There are two basic data packets, DATA0 and DATA1. Both consist of a DATA PID field, 0-1023 bytes of data payload and a 16-bit CRC. They must always be preceded by an address token, and are usually followed by a handshake token from the receiver back to the transmitter.

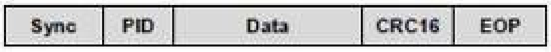


Fig. 3 Data Packet format

SOF Packet

The USB host transmits a special SOF (start of frame) token, containing an 11-bit incrementing frame number in place of a device address. This is used to synchronize isochronous data flows.

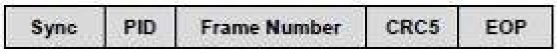


Fig. 4 Start of frame packet format

IV: PROPOSED ARCHITECTURE

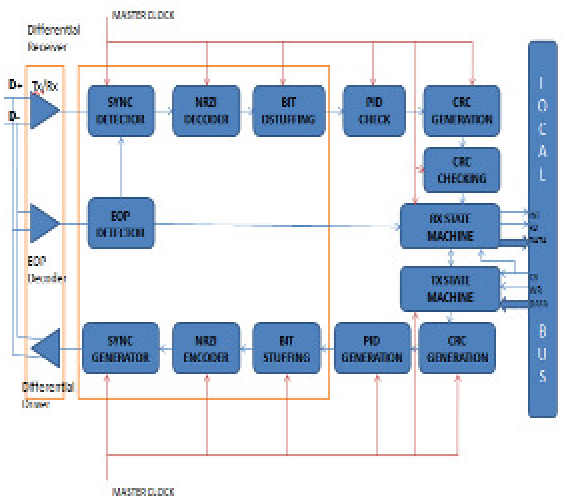


Fig. 5 A Proposed Architecture of USB 2.0 Device Controller with SIE (Serial Interface Engine).

SYNC DETECTOR

When the receiver detects encoded SYNC pattern “00000001”, the Receive state machine will enter into strip sync state where the SYNC pattern is stripped off. To detect the SYNC pattern a state machine is developed. It checks every bit for every rising edge of the clock. If the pattern is detected, a signal called sync detected is enabled. This signal is checked by the Receive state machine. If the signal is high, the Receive state machine will enter into strip sync state where RX active signal is asserted and the state machine will enter into RX data state.

NRZI DECODER

The received data on DP, DM lines are NRZI decoded. The NRZI Decoder simply XOR the present bit with the provisionally received bit. During NRZI decoding, the receiver state machine is in RX wait state. Instead of defining logic 0’s and 1's as voltages, NRZI encoding defines logic 0 as a voltage change, and logic 1 as a voltage that remains the same. Each logic-0 results in a change from the previous state. Each logic-1 results in no change in the voltages. The bits transmit least-significant-bit (LSB) first.

BIT D-STUFFING

This block checks for ‘0’ bit after 6 consecutive 1’s. If there is no ‘0’ bit after six 1’s then there arises NAK. And when it encounters next ‘0’ bit then it removes that extra stuffed ‘0’ bit and continues to check for next consecutive 1’s.

PID CHECK

This block will check for the PID packet. It will check whether the 4-bit of MSB and 4-bit of LSB are compliment of each other or not. They should be compliment of each other, if there is no transmission error. This block compares them and if they are not compliment of each other then it introduces a NAK signal saying that PID doesn’t match.

PID_0	PID_1	PID_2	PID_3	$nPID_0$	$nPID_1$	$nPID_2$	$nPID_3$

Fig. 6 Structure of Packet ID

CRC GENERATION

CRC5:- The CRC5 module is used whenever a token packet is received. A token packet has a crc5 field at its tail which has to be checked on arrival. The data part of the token is composed of the endpoint number and the device’s address. This information is inserted into the *data_in* (11 bit) entrance which then produces the correct CRC5 after a single clock.

CRC16:- The CRC16 module is used whenever a data packet or a setup packet is received. Each data packet in a transfer contains crc16 field in its tail. Upon each clock cycle the data is inserted into the module’s *data_in* (16 bits). After the last word is processed by the module we receive the correct crc16 in the module’s output.

CRC CHECKING

This block will compare the CRC generated and the original CRC. It will check whether the CRC transmitted and CRC received are same or not. The receiver will also generate a CRC for a particular packet.
If both the CRC generated and received are same then it acknowledges and operation is preceded, otherwise, it sends a (NAK) not acknowledge signal. Hence, the whole transaction is stopped and whole packet is transmitted again.

RECEIVER STATE MACHINE

The receiver module is designed by considering all the above specifications. Verilog is used to design the receiver module. The receiver module of the UTMI consists of various blocks such as SYNC detector, NRZI decoder, bit D-stuffing, receive shift and hold Register and EOP detector. A receive state machine is developed by considering all the states given by USB 2.0 receive state machine. Initially the receiver is at Reset state where the reset signal is high and RX active and RX valid signals are low. If reset signal goes low the state of the receiver is changed to RX wait state where it is waiting for SYNC pattern.
The receiver module has been implemented by considering the fallowing specifications.

- When SYNC pattern is detected that should be intimated to the SIE.
- If a zero is not detected after six consecutive 1’s an error should be reported to the SIE.
- When EOP pattern is detected that should be intimated to the SIE.

TRANSMITTER STATE MACHINE

The transmitter module is designed by considering all the above specifications. Verilog is used to design the transmitter module. The transmitter module of the

UTMI consists of various blocks such as SYNC generator; transmit hold and shift register, bit-stuffing, NRZI encoder and EOP generator. A transmit state machine is developed by considering all the states given by USB 2.0 transmit state machine. Initially the transmitter is at Reset state where the reset signal is high. If reset signal goes low state the state of the transmitter is changed to TX wait state where it is waiting for assertion of TX valid signal by the SIE.
The transmitter module of UTMI has been implemented by considering the following specifications.

- The SYNC pattern "00000001" has to be transmitted immediately after the transmitter is initiated by the SIE.
- After six consecutive 1's occur in the data stream, a zero to be inserted.
- The data should be encoded using Non Return to Zero Invert on 1(NRZI -1) encoding technique.
- The EOP pattern two single ended zeros (D+ and D- lines are carrying zero for two clock cycles) and a bit 1 have to be transmitted after each packet or after SIE suspends the transmitter.

PID GENERATION
Packets are of different types. Hence, there is unique predefined packet ID's for each packet type. So these packets ID's are generated according to the operation to be performed. Packet ID's are unique for token, data, and handshake packet, due to which controller can understand which packet transfer is going on.

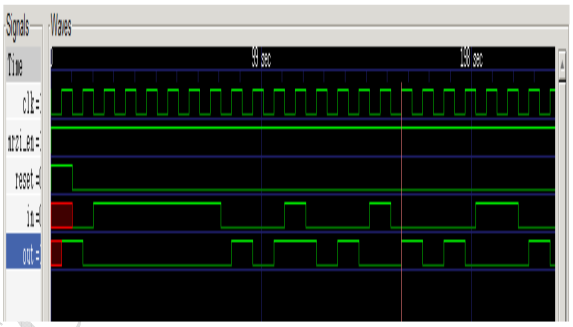
BIT-STUFFING
Bit stuffing is required because the receiver synchronizes on transitions. If the data is all 0's, there are plenty of transitions. But if the data contains a long string of 1's, the lack of transitions could cause the receiver to get out of sync.
If data has six consecutive 1's, the transmitter stuffs, or inserts, a 0 (represented by a transition) after the sixth 1. This ensures at least one transition for every seven bit widths. The receiver detects and discards any bit that follows six consecutive 1's.
Bit stuffing can increase the number of transmitted bits by up to 17 percent. In practice the average is much less. The bit-stuffing overhead for random data is just 0.8 percent, or one stuff bit per 125 data bits.

NRZI ENCODER
This block will do just the reverse of NRZI decoder. It will check when the input bit is high. And then it will toggle the output bit. The result of this block will give NRZI encoded data.

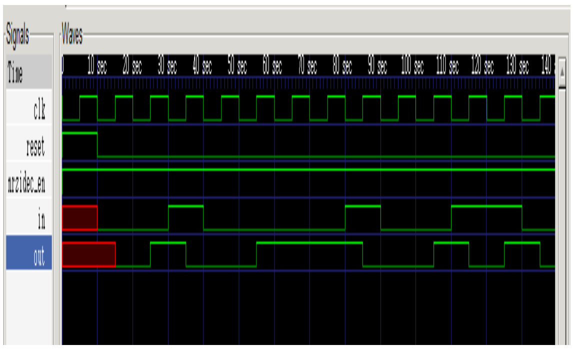
EOP DETECTOR
A state machine is developed for EOP detection, which is invoked at every rising edge of the clock. When two single ended zeroes fallowed by a 'J' state are detected, it asserts a signal called eop_detect which is checked by the Receive state machine at every rising edge of the clock. When this signal is high, the receiver state machine will enter in to Strip EOP state where the EOP pattern is stripped off and RX active, RX valid signals are negated. At the next rising edge of the clock. The Receive state machine will enter into the RX wait state.

V: SIMULATION RESULTS

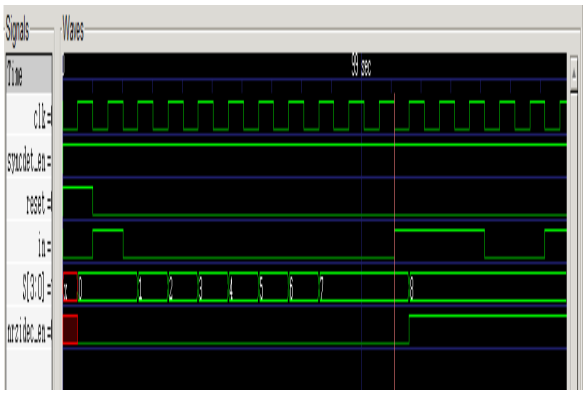
NRZI ENCODER
The original incoming data is converted into NRZI form. Whenever input is zero (0), output bit is toggled and whenever input is 1, output remain the same as previous (i.e. it does not changes). This can be understood by the figure below.



NRZI DECODER
The input sequence is: - 0010000100110 (NRZI encoded data).
Output sequence will be: - 0100111001010 (This is the original data).

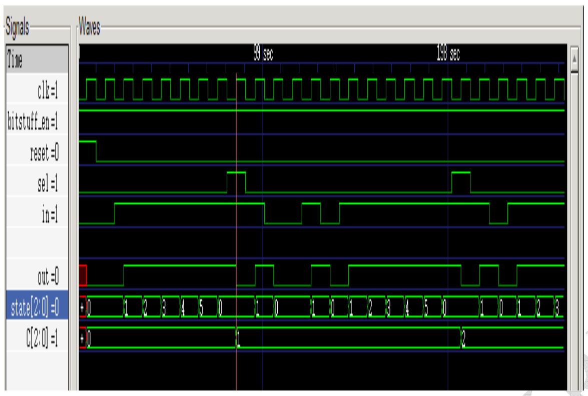


SYNC DETECTION
The Sync "00000001" is detected by this block. Once the sync is detected, it enables the next block. So as we can see the NRZI decoder block will be enables, as nrzidec_en is made high (as an output). This enable will then go and trigger the NRZI decoder block to perform operation.



BIT- STUFFING

After getting six consecutive 1's, a '0' bit is inserted. Here multiplexer operation is used to hold the input data for 1 cycle and add a stuffed bit in between.



VI: CONCLUSION

This design implements the USB2.0 device controller which is efficient to perform all the data operations such as data encoding/decoding, error detection, etc. The results obtained are cross checked with the USB protocol specifications. This provides a functional FPGA based model which can be further used as an IP core.

REFRENCES

[1]*Universal Serial Bus Specification Revision 2.0.* 11 October 2011. Retrieved 8 September 2012.

[2]USB 2.0 Specifications: *http://www.usb.org/developers/docs/usb_20_081810.zip*

[3]USB 2.0 Transceiver Macrocell Interface (UTMI): *http://www.intel.com/technology/usb/download/2_0_xcvr_macrocell_1_05.pdf*

[4]Algorithms for Cyclic Redundancy Code (CRC) Computation: *http://ishaksuleiman.tripod.com/00000.pdf*

[5] www.ieeexplore.iee.org

[6] www.usb.org

[7] www.wikipedia.com

[8] *"USB Complete- 3rd Editions"* By "Jan Axelson".

[9] *"Universal Serial Bus System Architecture"* By "Don Anderson".

[10] John Keithley L. Difuntorum, Kristine Mari U. Matutina, Al Jerome Mervyn Z.Tong Anastacia Ballesil Alvarez, Joy Alinda Reyes Madamba *"A USB 2.0 Controller for an ARM7TDM-S Processor Implemented in FPGA"* Nov. 2011.

[11] Wu Yao, Zhou Xuan; Tang Lei; Li Jingcheng *"A Date Transfer and Control System Solution Based on EZ-USB 2.0 for FPGA Applications"* Nov. 2010.