DESIGN OF USB 2.0 FUNCTION CORE

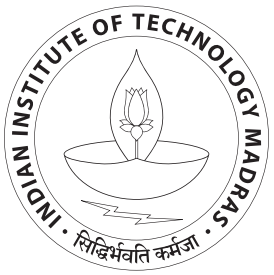
A Project Report

submitted by

NEELESH KUMAR PATEL

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2018

THESIS CERTIFICATE

This is to certify that the report titled **DESIGN OF USB 2.0 FUNCTION CORE**, submitted by **NEELESH KUMAR PATEL**, to the Indian Institute of Technology, Madras,for the award of **Master of Technology**, is a bonafide record of the work done by him under my supervision.The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V Kamakoti
Project Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my project guide Prof. V. Kamakoti sir for his guidence and support and for giving me opportunity to work for this project.

I would like to thank my co-guide Prof. Bobby George sir who has motivated me and supported me to work outside electrical department for this project.

I would also like to thank Dr. Neel Gala, Mr. Arjun, Mr. Rahul, Mr. Vinod and all my RISE lab mates who were always supportive and helped me throughout my project and cleared my doubts.

Lastly I would like to thank my parents for always standing by me and encouraging me to pursue higher studies at IIT Madras.

NEELESH KUMAR PATEL

ABSTRACT

KEYWORDS: PID,Endpoint, packet,Bulk Transfer, Isochronous Transfer

USB is a serial bus which can realize the Plug and Play feature for easy connection of peripherals to PCs. It is a point to point interface in which data rate of over 480Mbit/s can be transferred as per new USB 2.0 Specification. It provides bi-directional, low- cost and high speed serial interface for data transfer. Multiple devices can be attached through a hub to the host. The USB Communication implemented complies with USB 2.0 Specifications essential for basic data transfer and can operate at USB Full speed (12 Mbit/s) and High speed (480Mbit/s).

This project deals with designing of USB 2.0 function core that will provide functional interface between usb devices and host.The heart of the project is the protocol layer block that handles all the standard USB protocols. The designing is done by writing the code in Bluespec System Verilog hardware description language.Compiling and linking of code is done on Bluespec Workstaiton. The differnt types of data transations done by the USB core are verified by simulating the code using testbench waveforms and checking the results on Bluesim simulator.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

USB	Universal Serial Bus
UTMI	USB 2.0 Transceiver Macrocell Interface
PID	Packet Identifier
BSV	Bluespec System Verilog
SOF	Start of Frame
EOP	End of Packet
CRC	Cyclic Redundancy Check
PL	Protocol Layer
DMA	Direct Memory Access
CSR	Control and Status Register
SOC	System on Chip
RTL	Register Transfer Level

CHAPTER 1

INTRODUCTION

The Universal Serial Bus provides standard communication interface between peripheral devices and Host i.e computer. The peripheral devices include mouse, keyboard, printer, data storage devices and many more. With the new USB 2.0 specifications provided by USB.org, data rates of over 480Mb/s are possible.

The USB 2.0 standard can support up to 127 devices and has three different data transfer rates

- Low speed: For keyboards and mice with a data transfer rates at 1.5 Mbps
- Full speed: The USB 1.1 standard rate with a data transfer rates at 12 Mbps
- High speed: The USB 2.0 standard rate with a data transfer rates at 480 Mbit/s

1.1 USB Background

USB was developed by a group of seven companies namely COMPAQ, IBM, INTEL, MISCROSOFT, DEC, NEC and NORTEL in the year 1994.Though the USB 1.0 specification was developed in January 1996 which produced data rates of 1.5Mbit/s (LOW SPEED) and 12 Mbit/s (HIGH SPEED) , the widely used version was 1.1 released in September 1998. The later version, USB 2.0 was released in April 2000. The companied Hewlett-Packard, Alcatel Lucent, Philips and above mentioned firms took a joint initiative so as to generate a high data transfer rate of 480Mbit/s. A supplement release of USB-On-The-Go specification was carried out

in December 2006. This specification enabled two USB devices to communicate with each other without the need of a separate USB host. The latest release is USB 3.0, released in November 2008 and had the objectives to increase the data rate and reduce the power consumption as well as to increase the output power.

1.2 Problem Specification

To design USB 2.0 Core using Bluespec System Verilog Hardware description language that will provide functional interface between peripheral devices and host. The core fully complies to USB 2.0 specifications provided by USB 2.0 org.

- The core supports low speed, full speed and high speed data transfer rates.
- The core handles all USB standard protocols that consists of token packet, data packet and handshake packets.
- It supports diffrent data transfer types that is isochronous transfer, bulk transfer and interrupt transfer and control transfer.
- The core Includes reciever/ transmitter interface, it is provide by UTMI.
- The core provides direct memory access through internal DMA engine to reduce the work load on the function controller or host controller.

CHAPTER 2

SPECIFICATIONS

2.1 System Description

A USB system has the following intrinsic areas

- USB Interconnect
- USB Devices
- USB Host

USB Interconnect:-It describes the manner in which the USB devices are connected and communicates with the attached host. This includes

- Bus topology: The connection model between Host and USB devices
- Inter-layer Relationships: In terms of stack capability, the USB tasks are performed at each layer in the system.
- Data Flow Models: It is the manner in which data traverses in the system over the USB between producers and consumers.
- USB Schedule: The USB provides a shared interconnect. The access to the interconnect are scheduled so as to support isochronous data transfers and to eliminate arbitration overhead.

Bus Topology:-It follows a tier-star topology in which the Hub is at the center of each star and there is a point to point connection between host and hub /function or hub connected to hub/function. The maximum number of tiers allowed is 7 including the root hub.Each compound device covers two tiers; hence it can not be enabled if it is attached at tier-7.This is because only functions can be enabled

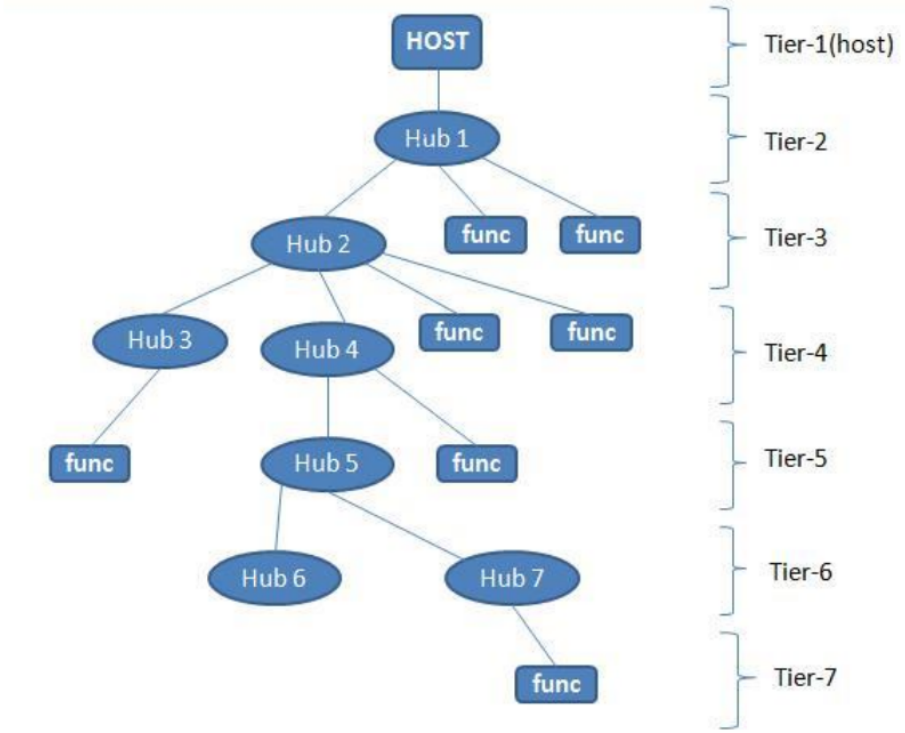


Figure 2.1: USB Bus Topology

at tier-7.

USB Devices:-The USB devices can be either a hub or a function. A hub provides additional attachment points to the USB. A USB device may comprise of many logical sub-devices which are known as device functions. Up to 127 devices can be attached to a single host controller.

USB Host:- In any USB system there is only one Host. A root hub is integrated within the host system to provide one or more attachment points. The USB interface to the Host is known as Host Controller, which may be implemented in a combination of hardware, software or firmware.

2.2 USB data flow

USB systems consist of a host, which is typically a personal computer (PC) and multiple peripheral devices connected through a tiered-star topology. This topology may also include hubs that allow additional connection points to the USB system. The host itself contains two components, the host controller and the root hub. The host controller is a hardware chipset with a software driver layer that is responsible for these tasks:

- Detect attachment and removal of USB devices
- Manage data flow between host and devices
- Provide and manage power to attached devices
- Monitor activity on the bus

At least one host controller is present in a host and it is possible to have more than one host controller. Each controller allows connection of up to 127 devices with the use of external USB hubs. The root hub is an internal hub that connects to the host controller(s) and acts as the first interface layer to the USB in a system.Generally on every computer, there are multiple USB ports.These ports are part of the root hub in the computer. For simplicity, look at the root hub and host controller from the abstract view of a black box that is referred as the host.

USB devices consist of one or more device functions, such as a mouse, keyboard, or audio device for example. Each device is given an address by the host, which is used in the data communication between that device and the host. USB device communication is done through pipes. These pipes are a connection pathway from the host controller to an addressable buffer called an endpoint. An endpoint stores received data from the host and holds the data that is waiting to transmit to the

host. A USB device can have multiple endpoints and each endpoint has a pipe associated with it.

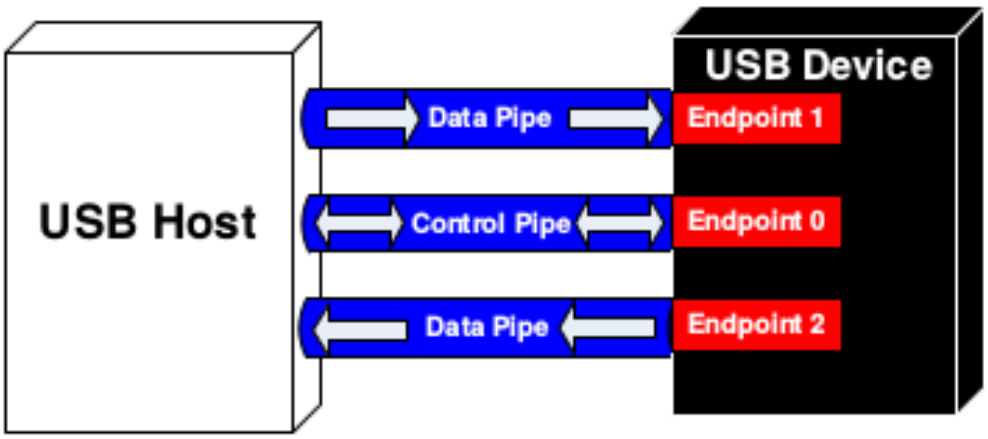


Figure 2.2: USB Pipe Model

2.2.1 Device Endpoints

Endpoints are the terminal points of a communication flow between the host and a device. Each device consists of independent endpoints and has a unique address assigned to it by the system at the time of device attachment. At the time of design each endpoint is given a unique device specific identifier, known as endpoint number. Each endpoint has a device specific direction of data flow. Each endpoint is uniquely referenced by its device address, endpoint number and direction of data flow. The data flow is simplex in nature: either input (device to host) or output (host to device).

Endpoint Characteristics:

- Bandwidth requirement
- Endpoint Number
- Maximun Payload size

- Transfer type
- Data transfer direction
- Bus access frequency

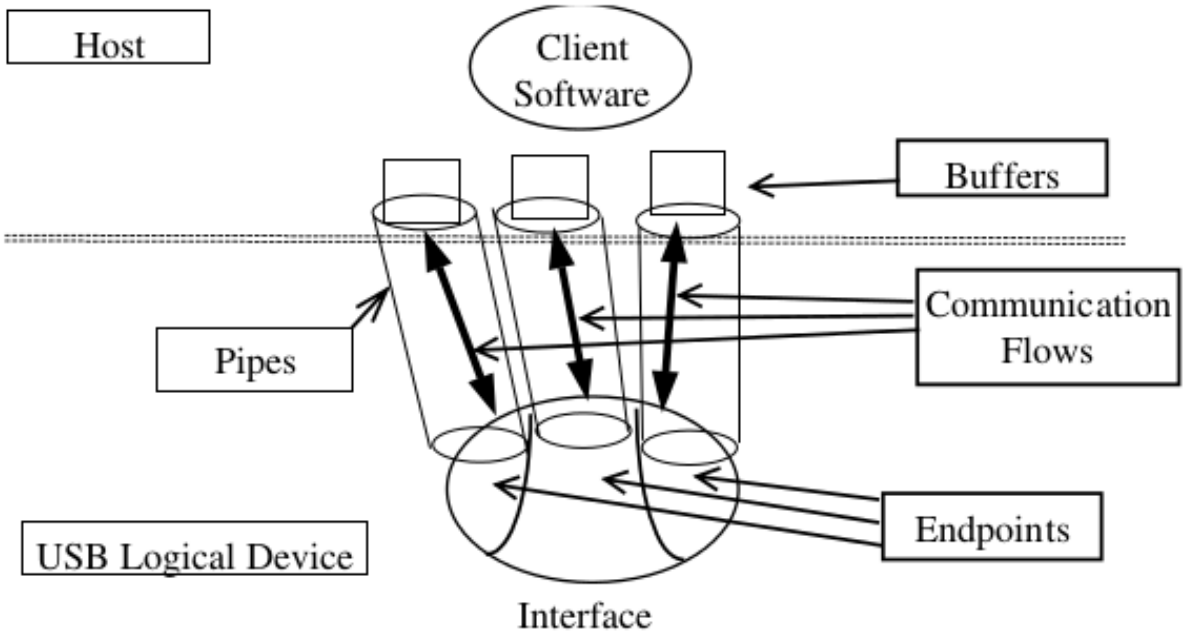


Figure 2.3: Endpoint communication flow

2.2.2 Pipes

USB pipe brings out connectivity between the endpoint on a device and software on the host. It represents the ability to move data between the host software through the memory buffer and device endpoint. Pipes communication exists in two types.

Stream Pipe: The data packet moving in this mode has no USB-defined structure. Stream pipe is unidirectional. It supports isochronous, bulk and interrupt transfers. The data coming in from one end and coming out of the other end has the same order.

Message Pipe: The data packet moving in this mode has some USB-defined structure. Message pipe is bi directional and it supports control transfer. It requires single endpoint number during the communication in both directions. The default control pipe always represents a message pipe.

There are two types of pipes in a USB system, control pipes and data pipes. The USB specification defines four different data transfer types. Which pipe is used depends on the data transfer type.

- **Control Transfers** Used for sending commands to the device, make inquiries, and configure the device. This transfer uses the control pipe.
- **Interrupt Transfers** Used for sending small amounts of bursty data that requires a guaranteed minimum latency. This transfer uses a data pipe.
- **Bulk Transfers** Used for large data transfers that use all available USB bandwidth with no guarantee on transfer speed or latency. This transfer uses a data pipe.
- **Isochronous Transfers** Used for data that requires a guaranteed data delivery rate. Isochronous transfers are capable of this guaranteed delivery time due to their guaranteed latency, guaranteed bus bandwidth, and lack of error correction. Without the error correction, there is no halt in transmission while packets containing errors are resent. This transfer uses a data pipe.

2.2.3 Descriptor Table

when a device is connected to a USB host, the device gives information to the host about its capabilities and power requirements. The device typically gives this information through a descriptor table that is part of its firmware. A descriptor table is a structured sequence of values that describe the device; these values are defined by the developer. All descriptor tables have a standard set of information that describes the device attributes and power requirements. If a design conforms to the requirement of a particular USB device class, additional descriptor informa-

tion that the class must have is included in the device descriptor structure.

A descriptor is a data structure with a specific format. Each descriptor states with a byte-wide field which contains the total number of bytes in it followed by a byte sized field that identifies the descriptor. A device descriptor table includes the general information about a USB device. This information is applicable globally to the device and its entire device configuration. Each USB device had only one device descriptor.

2.3 Communication Protocol

If we look at the USB communication from a time perspective, it contains a series of frames. Each frame consists of a Start of Frame (SOF) followed by one or more transactions. Each transaction is made up of a series of packets. A packet is preceded with a sync pattern and ends with an End of Packet (EOP) pattern. At a minimum, a transaction has a token packet. Depending on the transaction, there may be one or more data packets and some transactions may or may not have a handshake packet. Transactions are an exchange of packets and are comprised of three different packets; a token packet, optional data packet, and a handshake packet. Each packet can contain different pieces of information. What information is included depends on the packet type.

2.3.1 Packet Description

The data transmitted in USB communication are called packets. The packets are sending from host to devices via USB hubs. The packets follow a specific format. It

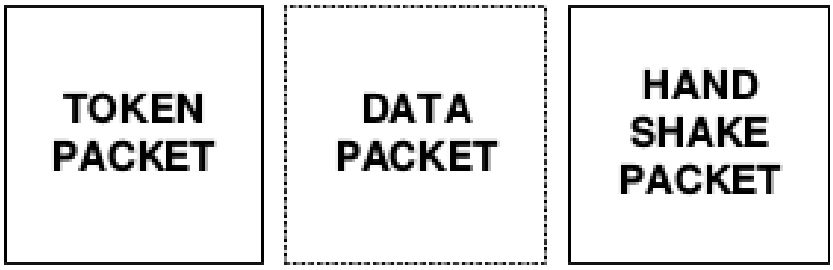


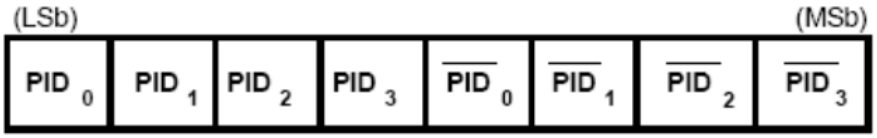
Figure 2.4: USB Transaction

starts with a SYNC pattern which allows the receiver bit clock to synchronize with the data, followed the data bytes of the packet (Pay Load) and concludes with EOP (End of Packet) signal. The data is NRZI (Non Return to Zero Inverted) encoded and in order to ensure sufficiently frequent transitions, it is bit-stuffed. Before and after the packet transmission, the bus remains in the idle state.

USB Packet Fields

On the USB data bus, the LSBit is transmitted first for any data transmission. The USB packet consists of the following fields.

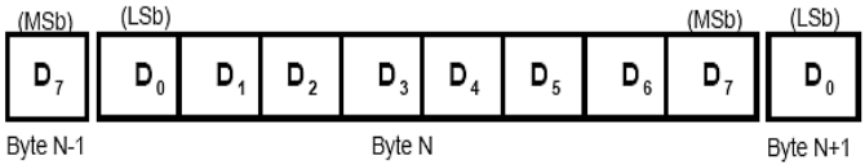
- **SYNC**
  All packets begin with the SYNC field which is 8 bits long at low and full speed or 32 bits long for high speed. It is used to synchronize the receiver clock with that of the transmitter clock. The last 2 bits indicates the end of SYNC field and start of PID field.
- **PID**
  PID or Packet identifier comes immediately after the SYNC field. It consists of 4 bit packet type field followed by a 4 bit check field. This check field is generated by doing one's complement of the packet type field. Error occurs if the Check bits are not complement of the Packet type bits.



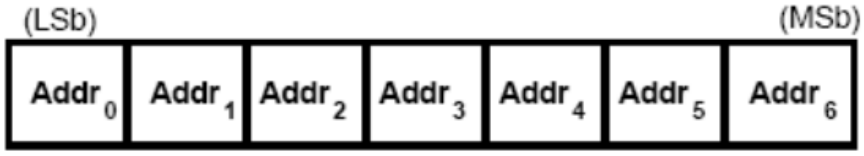
  The PID indicates the type of packet i.e. format of the packet and type of error detection applied to the packet. According to USB 2.0 specifications

there are 17 different PID values are defined. These are mentioned in the Table 2.1

- **DATA**
  The data field must be an integral number of bytes and ranges from 0-1024 bytes. The shifting of data begins from the LSBit. The size of packet varies according to the transfer type used.



- **ADDR**
  The address field (ADDR) specifies the destination of the data packet based on the value of token PID. It is 7 bits in length hence allowing a maximum of 127 devices to be used concurrently. Address 0 is considered invalid because any device which has not been assigned any address must respond to the packets sent at address 0.



- **ENDP**
  The Endpoint field (ENDP) allows flexible addressing of the functions which require more than one endpoint. It is 4 bits long thus providing 16 possible endpoints. A function can support a maximum of 16 IN and 16 OUT endpoints.

- **Frame Number Field**
  This field is 11 bits long and is incremented by the host on per-frame basis. Its maximum value is 7FFH and it is sent in SOF tokens at the beginning of each frame/micro frame.

- **CRC**
  A Cyclic Redundancy Check (CRC) is a value calculated from a number of data bytes to generate a unique value which is transmitted along with the data bytes, and then it is used to validate the correct reception of the data. Token packets have CRC of 5 bits while data packets have a CRC of 16 bits.

- **EOP**
  The packet field concludes with End of Packet (EOP). It is signaled by SE0 (Single Ended Zero) for approximately 2 bit times and then followed by a J for 1 bit time.

Table 2.1: PID Description table

PID Type	PID Name	PID<3:0>	Description
Token	OUT	0001B	address and endpoint number in host to device transaction
	IN	1001B	address and endpoint number in device to host transaction
	SOF	0101B	Start of Frame marker and frame number
	SETUP	1101B	address and endpoint number in host to device transaction for SETUP to a control pipe
Data	DATA0	0011B	Data packet PID even
	DATA1	1011B	Data packet PID odd
	DATA2	0111B	Data packet PID high speed, high bandwidth isochronous transaction in a microframe
	MDATA	1111B	Data packet PID high speed for split and high bandwidth isochronous transactions
Handshake	ACK	0010B	Receiver accept error free data packet
	NAK	1010B	Receivig device can not accept data or transmitting device can not send data
	STALL	1110B	Endpoint is halted or a control pipe request is not supported
	NYET	0110B	No response yet from receiver
Special	PRE	1100B	(Token) Host issued preamble. Enables downstream bus traffic to low speed devices.
	ERR	1100B	(Handshake) Split transaction Error Handshake(reuses PRE value)
	SPLIT	1000B	(Token) High speed Split Transaction Token
	PING	0100B	(Token) High speed flow control probe for a bulk/control endpoint
	Reserved	0000B	Reserved PID

USB Packet Types

There are four different packet formats depending on the type of PID used at the beginning of the packet.

- **Token Packet**
  Token packet indicates the type of transaction to be followed and identify the targeted endpoints. There are 3 types of Token packet IN, OUT and SETUP. IN packet indicates the USB device that the Host wants to read information. OUT packet indicates the USB device that the Host wants to send information SETUP packet is used to begin control transfers.

Table 2.2: Token Packet Format

Field	PID	ADDR	ENDP	CRC5
Bits	8	7	4	5

- **Data Packet**
  Data PIDs DATA0 and DATA1 are used transmitting data of size up to 1024 bytes. These PIDs are used in Low and Full speed links. Data PIDs DATA2 and MDATA are used in High speed mode.

Table 2.3: Data Packet Format

Field	PID	DATA	CRC16
Bits	8	(0-1024)*8	16

- **Handshake Packet**
  Handshake packets reports the status of data transaction and returns values indicating successful data reception or rejection, flow control and halt conditions. There are 4 types of handshake packets: ACK. NACK, NYET, and STALL.

Table 2.4: Handshake Packet Format

Field	PID
Bits	8

- **Start of Frame Packet**
  SOF (Start of Frame) indicates the start of a new frame. It is sent every 1ms on full speed links.

Table 2.5: Start of frame Packet Format

Field	PID	Frame No.	CRC5
Bits	8	11	5

CHAPTER 3

BLUESPEC SYSTEM VERILOG

Bluespec System Verilog is a Hardware Description Language (HDL), which is used for spec- ification, synthesis, modeling and verification of ASIC and FPGA design. With a radically different approach to highlevel synthesis, bluespec offers significantly higher productivity. It allows designers to express intended hardware through high-level constructs, where all behav- ior is described as a set of guarded atomic actions.

3.1 Limitation of Verilog

Verilog focusses more on simulation than logic synthesis. The source text of verilog often ex- plicitly contains aspects of circuit that could be readily determined by the compiler, such as size of registers, width of busses etc. This makes the design less portable. Handling concurrency in hardware is relatively difficult in verilog as the designer should manage all the aspects of hand- shaking between combinational circuits. Shared use of register and other memory resources should also be elaborated. The behavioral specification of design in verilog often consumes multiple clock cycles. Attempts to resolve this problem results in a highly unreadable code with possible bugs. In practice,this problem is solved by separating the combinational and sequential parts of the circuit. Due to these shortcomings, the synthesis and verification of hardware in verilog is slowed down. This is a huge problem during the design of SOC.

3.2 Bluespec

Bluespec is based on atomic transactions, which increases the level of concurrency abstraction above SystemC and RTL without compromising the control over hardware design. It enables automatic synthesis of complex control logic, which is the source of many bugs. This results in highly adaptable, reusable and reconfigurable designs. Control adaptive parametrization in bluespec provides flexibility, where a significantly different micro-architecture can be gener- ated by changing the parameters in the design with the associated control structures generated automatically. Bluespec allows user defined data types and static type checking. It provides several features of the modern high level languages and all of them can be synthesized.

In recent times, several attempts have been made to move the hardware design language to- wards a more software like specification of the circuit behaviour. Languages like C, C++ are used to express designs as sequential programs. However, the semantic gap between the soft- ware model and the hardware results in suboptimal designs with unpredictable speed and area. Bluespec System Verilog tackles this problem by building upon the traditional hardware se- mantics. It exploits advanced concepts from software only for static elaboration and staticverification. It uses the standard hardware structure model of verilog such as modules, module instances, hierarchy etc. For communication between modules it uses the System verilog model of interfaces and interface instances. These added with the advanced features of the high level languages, makes designing and verification in bluespec much faster.

3.3 Features of Bluespec

3.3.1 Modules and Interfaces

Module is the basic element of the hardware design hierarchy in bluespec. A module can be instantiated multiple times, and also different parameters can be passed during every instan- tiation. Unlike verilog, bluespec does not have input, output and in-out pins as interface to modules. Methods are used to drive signals and busses in and out of modules. These meth- ods are grouped together into interfaces. Modules contain rules, which use methods in other modules.

In BSV, the interface declaration is done separately, outside the module definition.

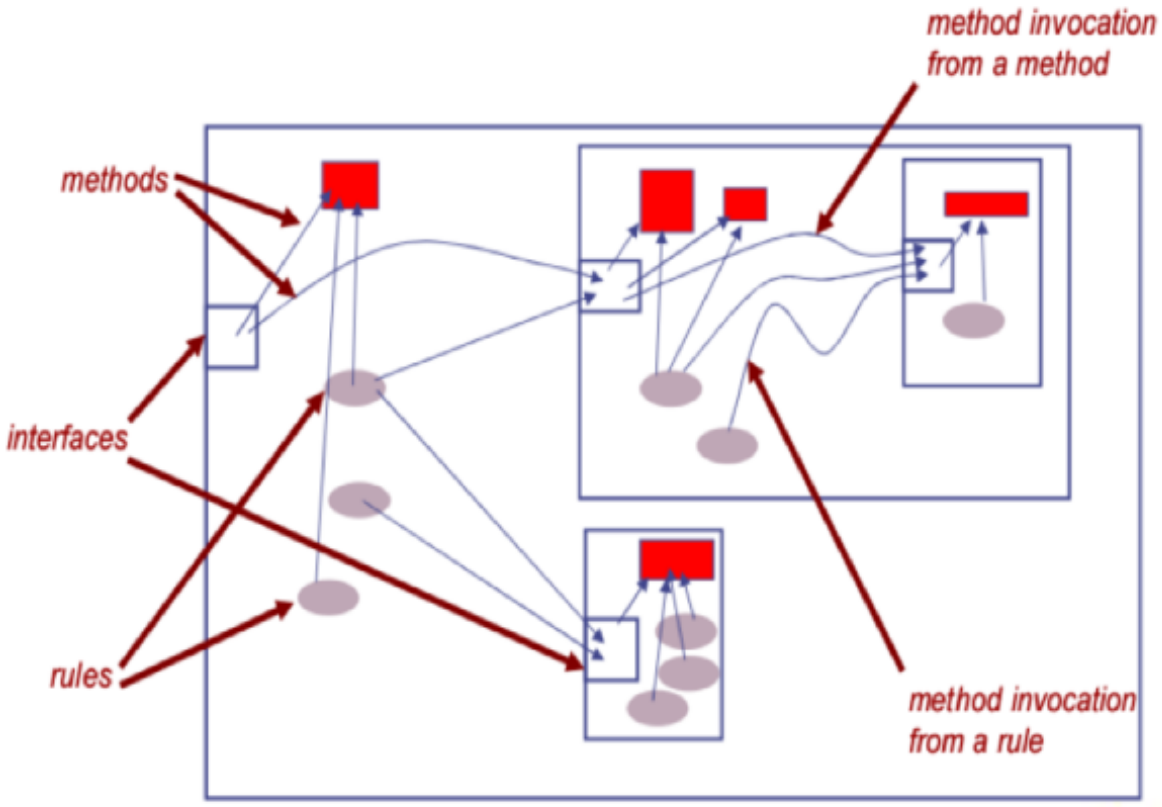


Figure 3.1: Methods, Interfaces and Rules in a module hierarchy

This allows declaration of common interfaces which can be used in multiple mod-

ules,without having to declare them repeatedly. All the modules which share the same interfaces also share same methods and therefore share same number and type of inputs and outputs.

3.3.2 Data Types

In verilog, all the representation is done in bits. Also, ultimately in hardware all computation is done in bits. However, representation in terms of integers, floating point numbers, fixed point numbers etc, makes the process of coding much easier. Different representations may be more appropriate depending on the application environment. By separating out the type abstraction from its bit representation, we can easily change representations without modifying or breaking the rest of the program.

In BSV, every variable has a type and only the values of compatible types can be assigned to a variable. The BSV compiler provides a strong, static type-checking environment. Type check- ing is done before the program elaboration and it ensures that the object types are compatible and the conversion functions are valid for the context. Bluespec also allows the usage of user- defined types. BSV has a type class which can be considered as a set of types. It implements overloading across related data types. Overloading is the ability to use a common name for a collection of types, with the specific type for the variable being chosen by the compiler based on the types on which it is actually used. Functions and operators are shared by all the data types within a type class.
Some common scalar types used in Bits type class are Bit(n), Bool, UInt(n) and Int(n). The values stored in registers, FIFOs and other memory elements and also

the values passed by wires, must be in the Bits type class. Other common data types include Integer, which belongs to the Arith type class and String, which belongs to the Literal type class etc.

3.3.3 Rules

Rules manage the movement of data from one state to another, within the module.It consists of two parts: rule conditions and rule body. Rule conditions are boolean expressions which decide whether the rule can be fired. Rule body is a set of actions for state transitions. Rules in BSV are atomic. The actions within the rule completely describes the state transition. The process of determining the functional correctness of a design is greatly simplified by one-rule-at-a- ime semantics. That is, because of the atomic property of rules, each rule can be looked at in isolation, without considering the actions of the other rules to determine functional correctness. Multiple rules can be executed concurrently in the hardware implementation.

The actions in a rule are executed simultaneously. This can be thought of as similar to the execution of non-blocking statements in always blocks of verilog. Also, as the rule has atomic property, the entire body of rule is executed and there is no partial exe- cution of a rule. When there are several rules within a module, the execution of rules is ordered by the compiler. No two rules can execute simultaneously. The ordering of the rules by the compiler is called scheduling.

3.3.4 Methods

method is a procedure which takes arguments and returns a value. It could also return a value without taking any arguments. It becomes a bundle of wires when translated into RTL. The method definition is written within the definition of the interface and it can be different in different modules sharing a common interface. A method also contains implicit conditions which are handshaking signals and logic automatically generated by the compiler. Methods are of three types: Value Methods, Action Methods and Action Value Methods. Value methods return a value. They do not alter any state within the module. Action methods cause actions to occur. They create state changes within the module. Action value methods are a combination of value methods and action methods. They cause state changes and also return values.

CHAPTER 4

FUNCTIONAL ARCHITECTURE AND OPERATION

4.1 Core Architecture

The following diagram gives the overall view of the core architecture. The host interface acts like a bridge between the internal data memory and control registers with the function controller. The control registers and the data memory interface to the Protocol Layer (PL). The PL interfaces to the UTMI block. Then the UTMI interface to the PHY. Each of the blocks has been described below.

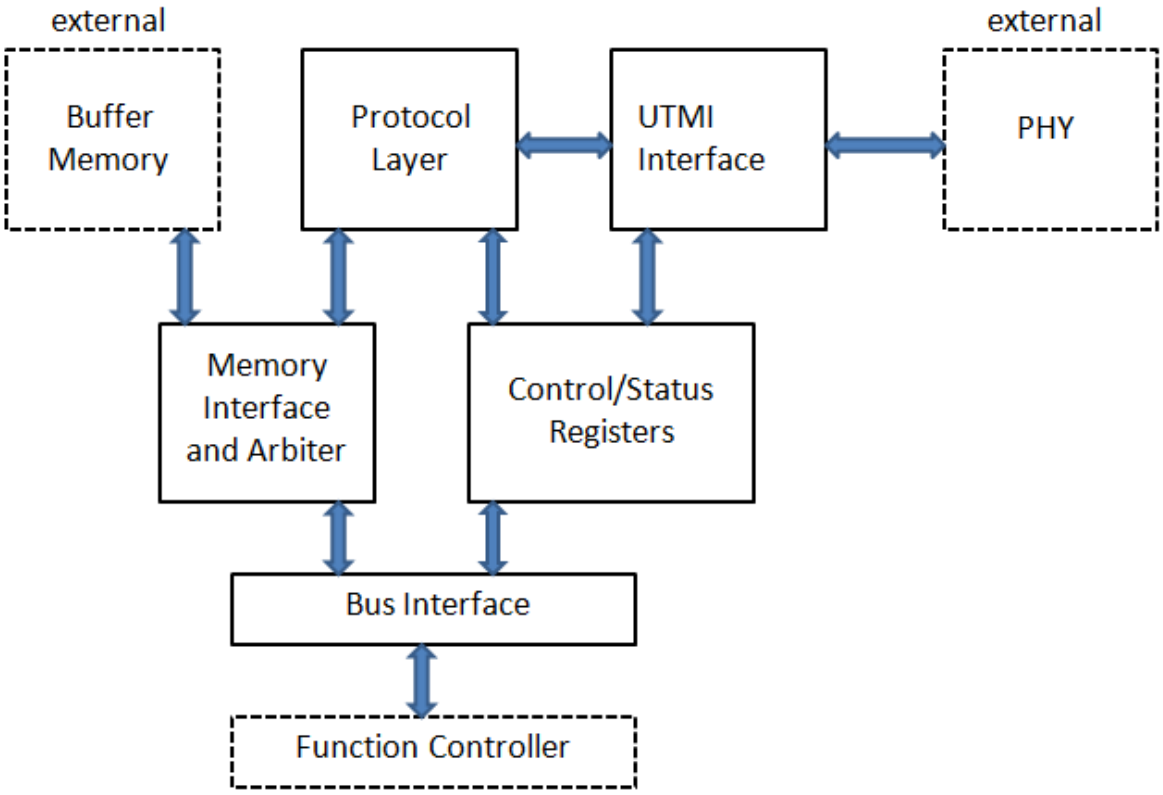


Figure 4.1: USB Core Architecture

4.1.1 Host Interface

It provides a consistent interface between the internal functions of the USB core and the function-defined host or micro-controller. The maximum theoretical throughput of the USB is 480 Mb/s or 60MB/s. On a bus size of 32 bits, 4 bytes are transferred in one cycle. Hence the minimum bandwidth required now for the host is 15 Mwords/s. (1 word = 4 bytes).

4.1.2 Memory Interface and Arbiter

This block arbitrates between the USB core and the host interface so as to access the memory. This block uses a standard single port SRAM (synchronous). This block also performs dataflow management and flow control.

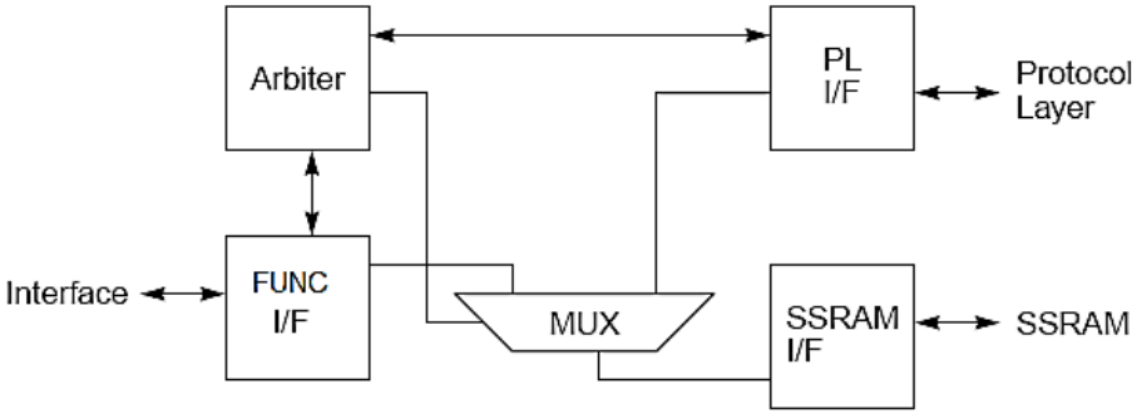


Figure 4.2: Memory Interface and Arbiter

4.1.3 Buffer Memory(SSRAM)

The SSRAM (Synchronous Static Random Access Memory) block is used for buffering the input and the output data.

4.1.4 Protocol Layer

The protocol layer is responsible for all USB data Input/Output and control communications.

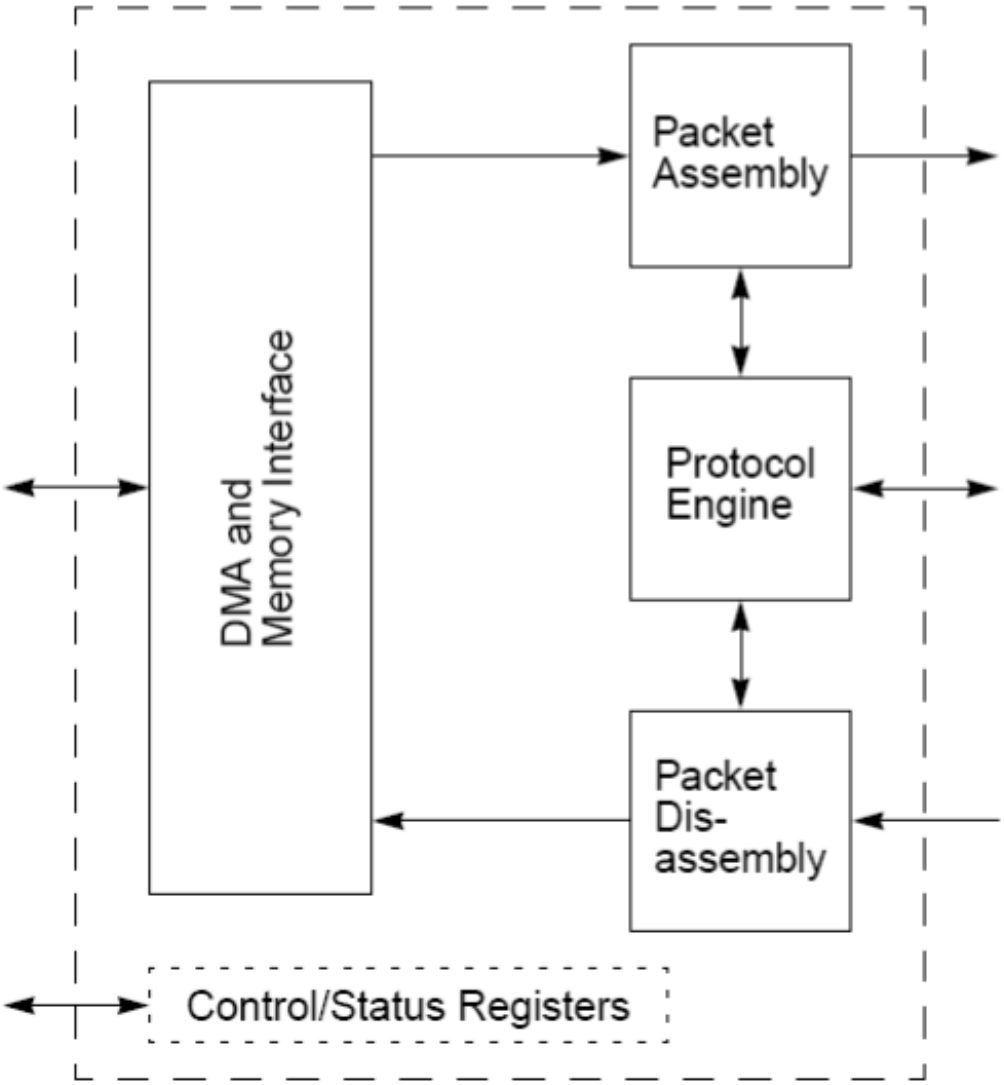


Figure 4.3: Protocol Layer

DMA and Memory Interface

This block interfaces to the data memory. It provides random memory access and also DMA block transfer.According to the instructions given by protocol engine it

perform the data transfer(IN or OUT).

Protocol Engine

Protocol Engine handles all the standard USB protocol handshakes and control correspondence. Those are SOF tokens, acknowledgment of data transfers (ACK, NACK, NYET), replying to PING tokens.This is the main controlling block.It takes decoded informatin from the packet disassembler block and reads the status of control and status registers, according to that information it decides which action to perform, and accordingly it gives instruction to DMA engine and packet assembler.

Packet Assembly

This block assembles data and handshake packets. Data packet consists of PID, data information and CRC check.So according to the insturction given by protocol engine, it first places the pid information and then data(if requested) and finaly CRC bits.Handshake packet consists of only PID information, so it assembles only PID bits.

Packet Disassembly

This block extracts information from packets.It extracts PID information form token packet and sends it to protocol engine. From data packet it extracts PID and data information, it does not extract CRC bits because that is not useful information.

4.2 UTMI Interface

The UTMI stands for USB 2.0 Macrocell Transceiver Interface. This block takes care of low lever USB protocols and signaling. The features which are included in signaling are data serialization and deserialization, bit stuffing and unstuffing, clock recovery and synchronization.The primary focus of this block is to shift the clock domain of the data from the USB 2.0 rate to one that is compatible with the general logic in the ASIC or FPGA.

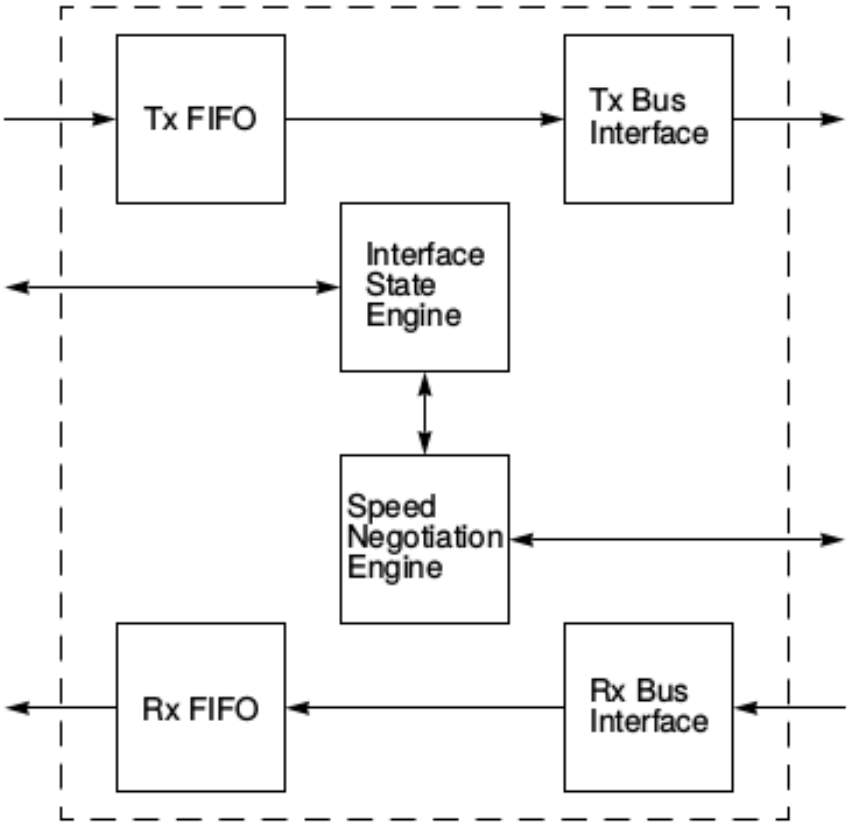


Figure 4.4: UTMI Interface Block

Interface state engine This block tracks the interface state. It controls suspend/resume modes and Full Speed/High Speed switching. An internal state machine keeps track of the state and the switching of the operating modes.

Speed Negotiation Engine This block negotiates the speed of the USB interface and handles suspend and reset detection.

Rx and Tx FIFOs The FIFOs hold the temporary receive and transmit data. The receive FIFO temporarily hold received bytes before the DMS writes them to the SSRAM buffer. The transmit FIFO temporarily holds the bytes to be transmitted.

Rx and Tx FIFOs These blocks ensure proper handshaking with the receive and transmit inter- faces of the PHY.

4.2.1 UTMI Signal Description

Description about various UTMI signals is given in Table 4.1,4.2,4.3,4.4

4.2.2 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 the NRZI decoding, receive state machine remains in RX wait state.

4.2.3 Bit Unstuff Logic

This logic can operates on both FS or HS data rates. The Bit Unstuffer checks each bit of the data stream and if a zero is detected after six consecutive 1s that zero bit is deleted. In FS mode, if error is detected then RxError signal is asserted.

Table 4.1: System Interface Signals

Name	**Direction**	**Active level**	**Description**
CLK	Output	Rising Edge	This output is used for clocking receive and transmit parallel data.
Reset	Input	High	Reset all state machines in the UTMI.
Xcvr Select	Input	N/A	Transceiver Select: It selects between FS/HS transceivers 0: HS transceiver enabled 1: FS transceiver select
Term select	Input	N/A	Termination Select: It selects between FS/HS terminations 0: HS termination enabled 1: FS termination enabled
SuspendM	Input	Low	It places the Macrocell in that mode which draws minimal power from the sources. 0: Macrocell drawing Suspend current 1: Macrocell not drawing suspend current
Line State(0-1)	Output	N/A	Line State: It shows the current state of the receivers. DM DP Description 0 0 : SE0 0 1 : 'J' State 1 0 : 'K' State 1 1 : SE1
OpMode(0-1)	Input	N/A	Operation Mode: It selects the operational modes [1] [0] Description 0 0 0: Normal Operation 0 1 1: Non-Driving 1 0 2: Disable Bit Stuffing and NRZI encoding 1 1 3: Reserved

Table 4.2: USB Interface Signals

Name	**Direction**	**Active level**	**Description**
DP	Bidirectional	N/A	USB data pin: Data+
DM	Bidirectional	N/A	USB data pin: Data-

Table 4.3: Data Interface Signals(Transmit)

Name	**Direction**	**Active level**	**Description**
DataIn	Input	N/A	8 bit parallel data input bus
TxValid	Input	N/A	Transmit Valid: It Indicates that the DataIn bus is valid
TxReady	Output	High	Transmit data ready: If TxValid and TxReady are high then UTMI loads DataIn to Tx register

Table 4.4: Data Interface Signals(Receive)

Name	**Direction**	**Active level**	**Description**
DataOut	Output	N/A	8 bit parallel data Output bus
RxValid	Output	N/A	Receive Valid: It Indicates that the DataOut bus is valid
RxActive	Output	High	Receive Active: It indicates that SYNC has been detected and is active
RXError	Output	High	Receive Error 0: No error 1: Receive error has been detected

Whereas in HS mode, an EOP signal is generated using the bit stuff errors. Hence RxError signal is not asserted in HS mode.

4.2.4 Rx Shift/Hold Registers

This block deserializes the received data , recovered by the DLL and transmit 8-bit parallel data to the bus interface. This module is primarily an 8-bit shift register which does serial to parallel conversion and buffers the deserialized data byte.

4.2.5 Receive State Machine

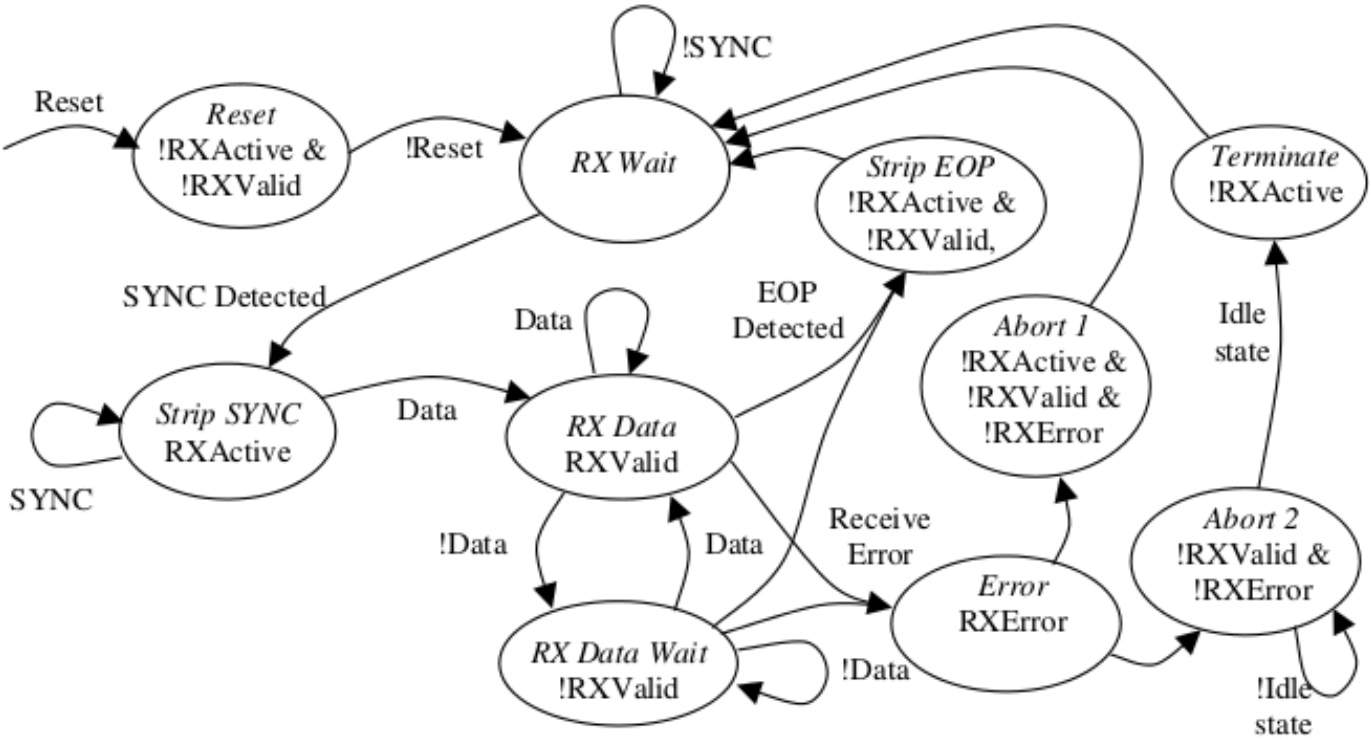


Figure 4.5: Receive State Machine

When the Reset signal is negated, the Receive State Machine enters the Wait state as shown. There it starts looking for a SYNC pattern. When it has detected a SYNC pattern, the state machine will enter the Strip SYNC state and assert

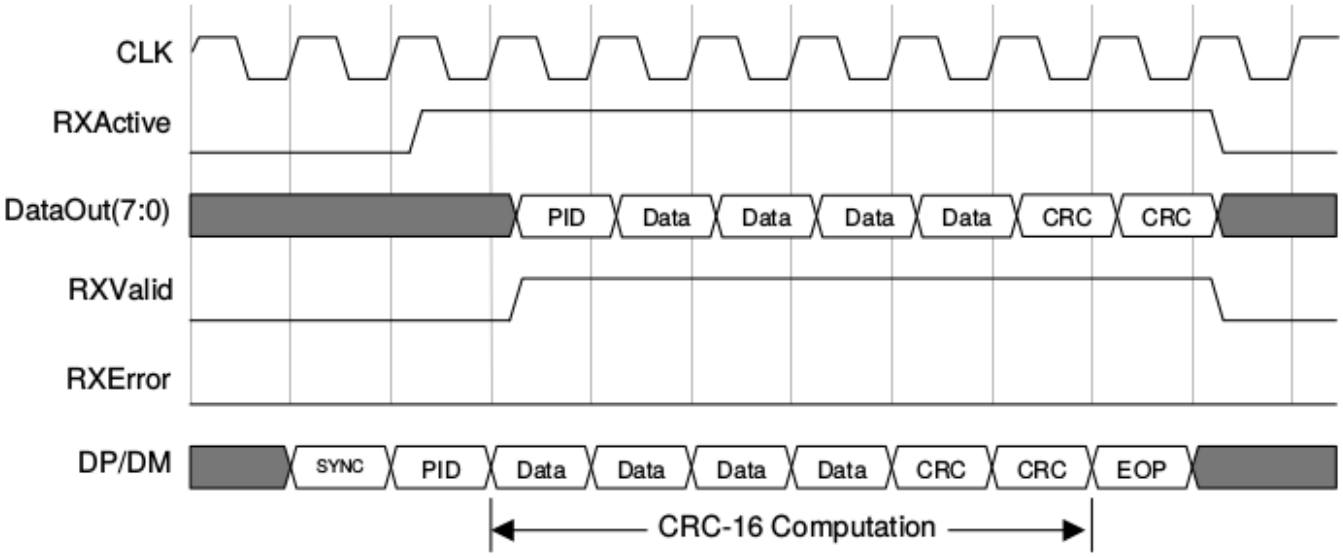


Figure 4.6: Receive Timing for Data Packet

RXActive. After receiving 8 bits of valid serial data, the state machine enters the RX Data state where the data is loaded into the RX Holding Register on the rising edge of CLK and RXValid is asserted. Stuffed bits are then stripped from the data stream. Each time 8 stuffed bits are accumulated, state machine will enter the RX Data Wait state, negating the RXValid. When the EOP signal is detected, the state machine will enter the Strip EOP state and negate RXActive and RXValid. After the EOP has been stripped the RX State Machine will re-enter the Wait state and begin looking for the next packet. If a Receive Error is detected in the process, the state machine enters Error State and RXError is asserted. Then state machine enters either the Abort 1State where RXActive, RXValid, and RXError signals are negated, or the Abort 2 State where only RXValid, and RXError signals are negated.

4.2.6 NRZI Encoder

The transmitted data on DP, DM lines are NRZI encoded. During the NRZI encoding, transmit state machine remains in TX wait state. When bit 1 is received

in the serial data it is negated and transmitted to DP, DM lines. Whereas when bit 0 is encountered, it is directly transmitted on to the DP, DM lines.

4.2.7 Bit Stuff Logic

Bit stuffing is carried out so as to ensure adequate signal transition when sending data. A 0 is inserted after every 6 consecutive 1s before the data is NRZI encoded. This is done to enforce a transition in the NRZI data stream. SYNC pattern enables Bit stuffing. After every 8 bits stuffed into the data stream, TxReady gets negated for one byte time so as to hold up the data on to the DataIn bus.

4.2.8 Tx Shift/Hold Registers

The module receieves 8bit parallel data and buffers in the Tx Hold Reg and then sends to 8- bit shift register. This block primarily converts parallel data to serial data.

When Reset signal is negated the, transmit state machine enters the TX Wait state. In the TX Wait state, the TX state machine looks for the assertion of TXValid and when it is detected, the state machine will enter the Send SYNC state and begin transmission of the SYNC pattern. When the transmitter is ready for the first byte of the packet (PID), it enters the TX Data Load state, TXReady is asserted and the TX Holding Register is loaded. The state machine may enter the TX Data Wait state on completion of the SYNC pattern transmission.The state machine remains in the TX Data Wait state until the TX Data Holding register is available for some more data. In the TX Data Load state, the state machine loads the Transmit Holding register.

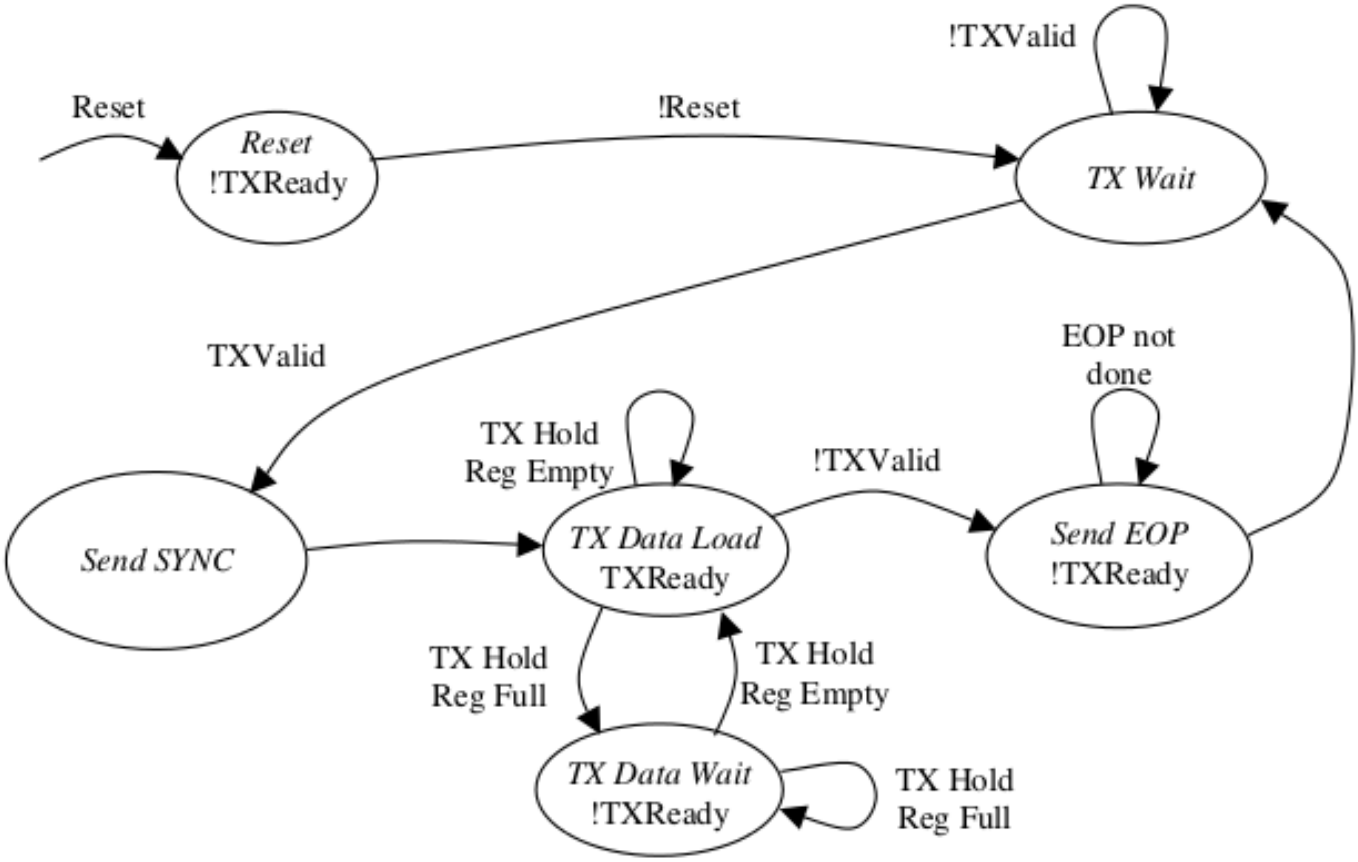


Figure 4.7: Transmit State Machine

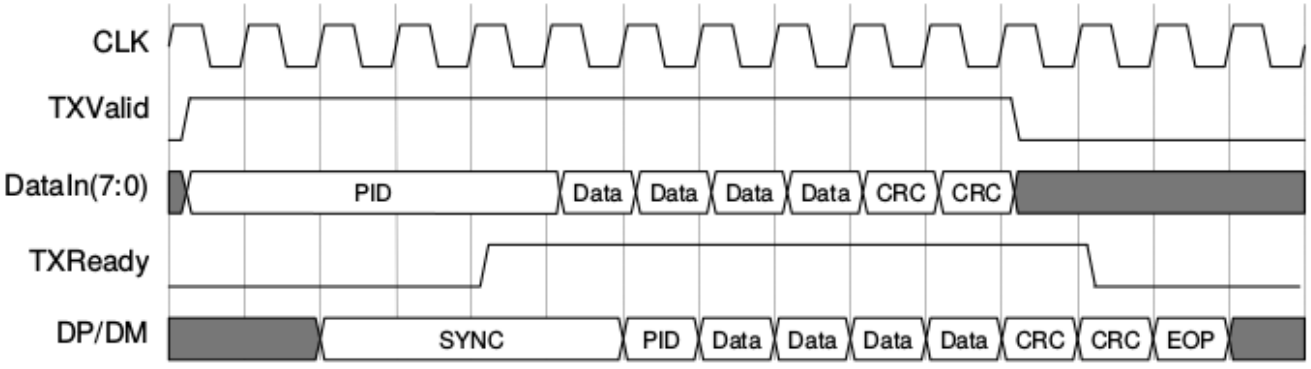


Figure 4.8: Transmit Timing for Data Packet

The state machine will remain in TX Data Load state as long as the transmit state machine can empty the TX Holding Register before the next rising edge of CLK. When TXValid is negated, the TX state machine enters the Send EOP state where it

sends the EOP. During the EOP transmission, TXReady signal is negated and the state machine will remain in the Send EOP state. After the EOP is transmitted the Transmit State Machine returns to the TX Wait state, looking for more data.

4.3 Endpoint Registers

USB 2.0 core can support upto 16 endpoints, there are four registers associated with each endpoint.All registers have same defination for each endpoint.Access specifies the valid access types to that register.RW stands for read and write access, RO for read only access."C" appended to RW or RO indicates that some or all of the bits are cleared after a read.

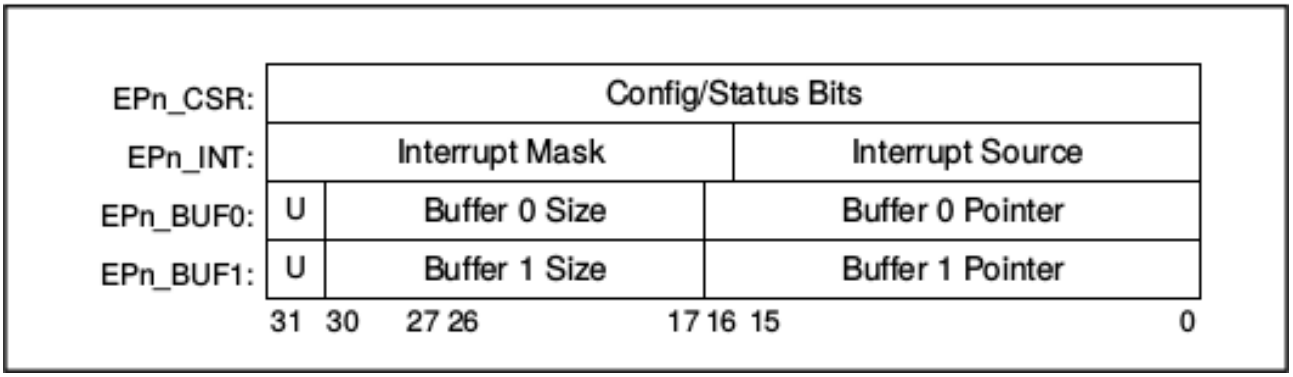


Figure 4.9: Endpoint Registers

4.3.1 Endpoint Control/Status Register (EP_CSR)

This is 32 bit register, it gives the information about the endpoint i.e type of endpoint(IN,OUT or Control), type of trasfer it supports and reports any status back to the controller.Details about every bit of this register is given in Table 4.5

Table 4.5: Endpoint CSR

Bit	Access	Description
31:30	RO	UC_BSEL (Buffer select) This bits must be initialized to zero (first Buffer 0 is used). The USB core will toggle these bits, in order to know which buffer to use for the next transaction. 00: Buffer 0 01: Buffer 1 1X: Reserved
29:28	RO	UC_DPD These two bits are used by the USB core to keep track of the data PIDs for high speed endpoints and for DATA0/DATA1 toggling.
27:26	RW	EP_TYPE (Endpoint Type) 00: Control Endpoint 01: IN Endpoint 10: OUT Endpoint 11: RESERVED
25:24	RW	TR_TYPE (Transfer Type) 00: Interrupt 01: Isochronous 10: Bulk 11: RESERVED
23:22	RW	EP_DIS:Temporarily Disable The Endpoint 00: Normal Operation 01: Force the core to ignore all transfers to this endpoint 10: Force the endpoint in to HALT state 11: RESERVED
21:18	RW	EP_NO (Endpoint Number)
17	RW	LRG_OK 1: Accept data packets of more than MAX_PL_SZ bytes (RX only) 0: Ignore data packet with more than MAXPL_SZ bytes (RX only)
16	RW	SML_OK 1: Accept data packets with less than MAX_PL_SZ bytes (RX only) 0: Ignore data packet with less than MAXPL_SZ bytes (RX only)
15	RW	DMAEN 1: Enables external DMA interface and operation 0: No DMA operation
14	RO	RESERVED
13	RW	OTS_STOP:When set, this bit enables the disabling of the endpoint when in DMA mode, an OUT endpoint receives a packet smaller than MAX_PL_SZ. The disabling is achieved by setting EP_DIS to 01b
12:11	RW	TR_FR Number of transactions per micro frame (HS mode only)
10:0	RW	MAX_PL_SZ:Maximum payload size in bytes

4.3.2 Endpoint Interrupt Mask/Source Register

This register tells about the cause and source of the interrupt.It is also 32 bit wide, details are given in Table 4.6

Table 4.6: Endpoint Interrupt Register

Bit	Access	Description
31:30	RO	RESERVED
29	RW	Interrupt A Enable: OUT packet smaller than MAX_PL_SZ
28	RW	Interrupt A Enable: PID Sequencing Error
27	RW	Interrupt A Enable: Buffer Full/Empty
26	RW	Interrupt A Enable: Unsupported PID
25	RW	Interrupt A Enable: Bad Packet(CRC 16 Error)
24	RW	Interrupt A Enable: Time Out (waiting for ACK or DATA packet)
23:22	RO	RESERVED
21	RW	Interrupt B Enable: OUT packet smaller than MAX_PL_SZ
20	RW	Interrupt B Enable: PID Sequencing Error
19	RW	Interrupt B Enable: Buffer Full/Empty
18	RW	Interrupt B Enable: Unsupported PID
17	RW	Interrupt B Enable: Bad Packet(CRC 16 Error)
16	RW	Interrupt B Enable: Time Out (waiting for ACK or DATA packet)
15:7	RO	RESERVED
6	ROC	Interrupt Status:OUT packet smaller than MAX_PL_SZ
5	ROC	Interrupt Status:PID Sequencing Error
4	ROC	Interrupt Status:Buffer 1 Full/Empty
3	ROC	Interrupt Status:Buffer 0 Full/Empty
2	ROC	Interrupt Status:Unsupported PID
1	ROC	Interrupt Status:Bad packet (CRC 16 error)
0	ROC	Interrupt Status:Time Out (waiting for ACK or DATA packet)

4.3.3 Endpoint Buffer Registers(EP_BUF)

Endpoint Buffer register contains the address of the buffer for that endpoint and specifies the size of the buffer.Each endpoint has two buffer registers, thus allowing double buffering.

Table 4.7: Endpoint Buffer Register

Bit	Access	Description
31	RW	USED This bit is set by the USB core after it has used this buffer. The function controller must clear this bit again after it has emptied/refilled this buffer.This bit must be initialized to 0.
30:17	RW	BUF_SZ Buffer size (number of bytes in the buffer) 16383 bytes max.
16:0	RW	BUF_PTR Buffer pointer (byte address of the buffer)

4.4 Operation

4.4.1 Buffer Operations

The buffer operations are carried out by buffer pointers which points to the I/O data structures in the memory. If the value of buffer pointer is 7FFFh, then it indicates that the buffer has not been allocated yet. The core responds with a NAK to the USB host if all the buffers are not allocated. Data transfer within buffer takes place in a Round Robin fashion. Buffer 0 is used first when data is sent to/from an endpoint. The function controller is notified through an interrupt when the Buffer 0 is empty/full. Buffer 0 can now be refilled/emptied by the function controller. When the second buffer is empty/full, the function controller is interrupted, and the USB core will use buffer 0 again, and so on.

A buffer can be larger than the MAXIMUM_PAYLOAD_SIZE. In that case, many packets can retrieved/placed from/to a buffer. For an OUT endpoint, a buffer must always be multiples of maximum payload size. The buffer size field should always be checked by the software. When the entire buffer has been used, the buffer size is 0. If it is not 0, then the size field indicates, the number of bytes of the buffer that has not been used.

There is no such limitation in case of IN buffers. The core will always transmit

the maximum possible number of bytes, which is the minimum of the payload size and remaining buffer size.Control endpoints can receive and transmit data. Hence in this case, Buffer0 is always an OUT buffer and Buffer1 always an IN buffer.

Buffer Underflow

A buffer underflow condition is specified, when either the expternal DMA engine or the function controller has not filled the, the internal buffer with enough data for one MAXIMUM_PAYLOAD_SIZE packet. In this condition, the USB core replies a NACK to the host when an IN token is received.

Buffer Overflow

A buffer overflow is specified when a packet, that has been received, does not fit into the buffer. The packet is discarded and a NACK is sent to the host. The USB core will continue discarding the received data, and reply with NACK to each OUT token when payload size does not fit into the buffer.

4.4.2 DMA Operation

DMA operation allows a complete transparent movement of data from the USB core to the functions attached. Each point is associated with a pair of dma_req and dma_ack signals. The USB core will use these signals for DMA flow control when the dma_en bit is set in the CSR register. When the buffer contains data or when the buffer is empty and needs to be filled, the dma_req signal is asserted. The DMA responds with a dma_ack signal for each word transferred.

Only Buffer 0 is used in DMA mode. The function controller is notified with an interrupt when the received packet is less that the MAX_PL_SZ. In addition to it,

the Buffer1 is set to the local buffer address of that received packet. So as to carry out uninterrupted DMA transfers, the buffer is padded by the USB core with the MAX_PL_SZ bytes.

4.4.3 USB Core Main Flow Chart

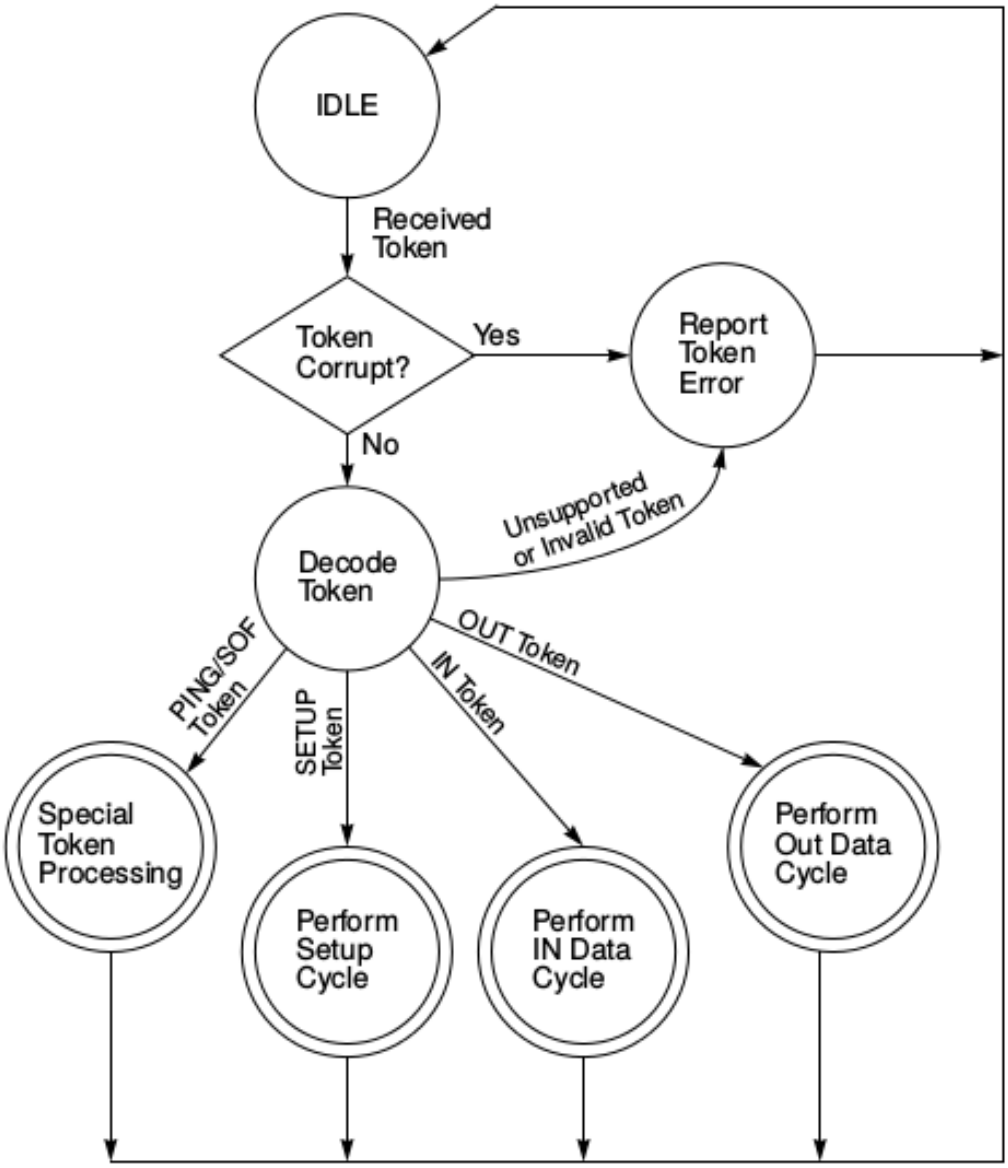


Figure 4.10: USB Core Main Flow chart

The figure 4.10 illustrates the main loop for USB core.It will sit Idle until token is received. After receiving token it will decode the token, extract all the important information from the token and according to that information USB Function Core will decide what operation to perform next.

IN Data Cycle

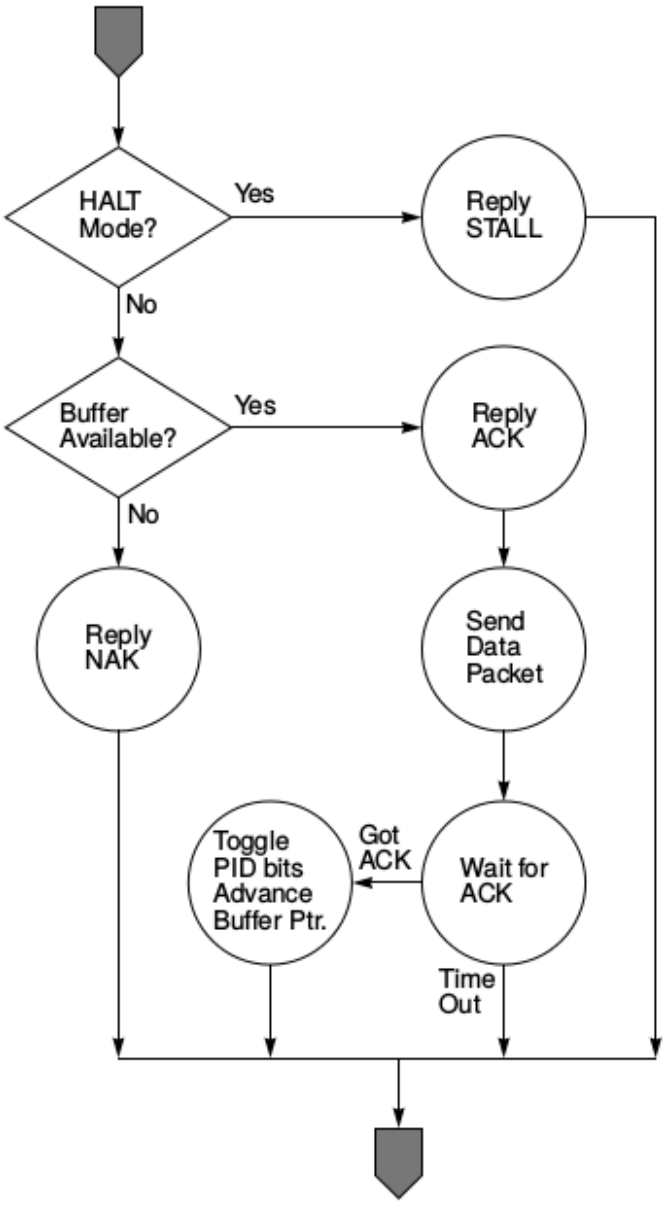


Figure 4.11: IN Data Cycle Flow chart

In this cycle, Data is sent from device to host.If Data transfer is done succesfully then host replies with ACK handshake signal. USB core first checks for the HALT mode,if this mode is on then it will reply with STALL handshake otherwise it will check for the availablity of buffer.If buffer is available then it will send the data packet to host and will wait for ACK from the host.After receiving ACK from host it will toggle PID bits and increament the buffer pointer.The figure 4.11 shows the flow chart for the IN Data Cycle.

OUT Data Cycle

In this cycle, Data is sent from host to device.USB core first checks the HALT mode,if this mode is on then it will reply with STALL handshake otherwise it will wait for the data packet from the host.It will check for the CRC and PID error,if error is there it will reply with NACK handshake signal.If it detects no error then it will accept the data packet and increament the buffer pointer.After succesfull completion of data transfer it will reply with ACK handshake signal.The figure 4.12 shows the flow chart for OUT Data Cycle.

Special Token Processing

The USB Core currently supports only two special tokens in addition to SETUP, IN and OUT tokens. These tokens are SOF and PING. When a SOF token is received, the FRM_NAT register is updated.

The PING token is a special query token for high speed OUT endpoints. It allows the host to query whether there is space in the output buffer or not.The figure 4.13 shows the flow chart for Special Token Processing Cycle.

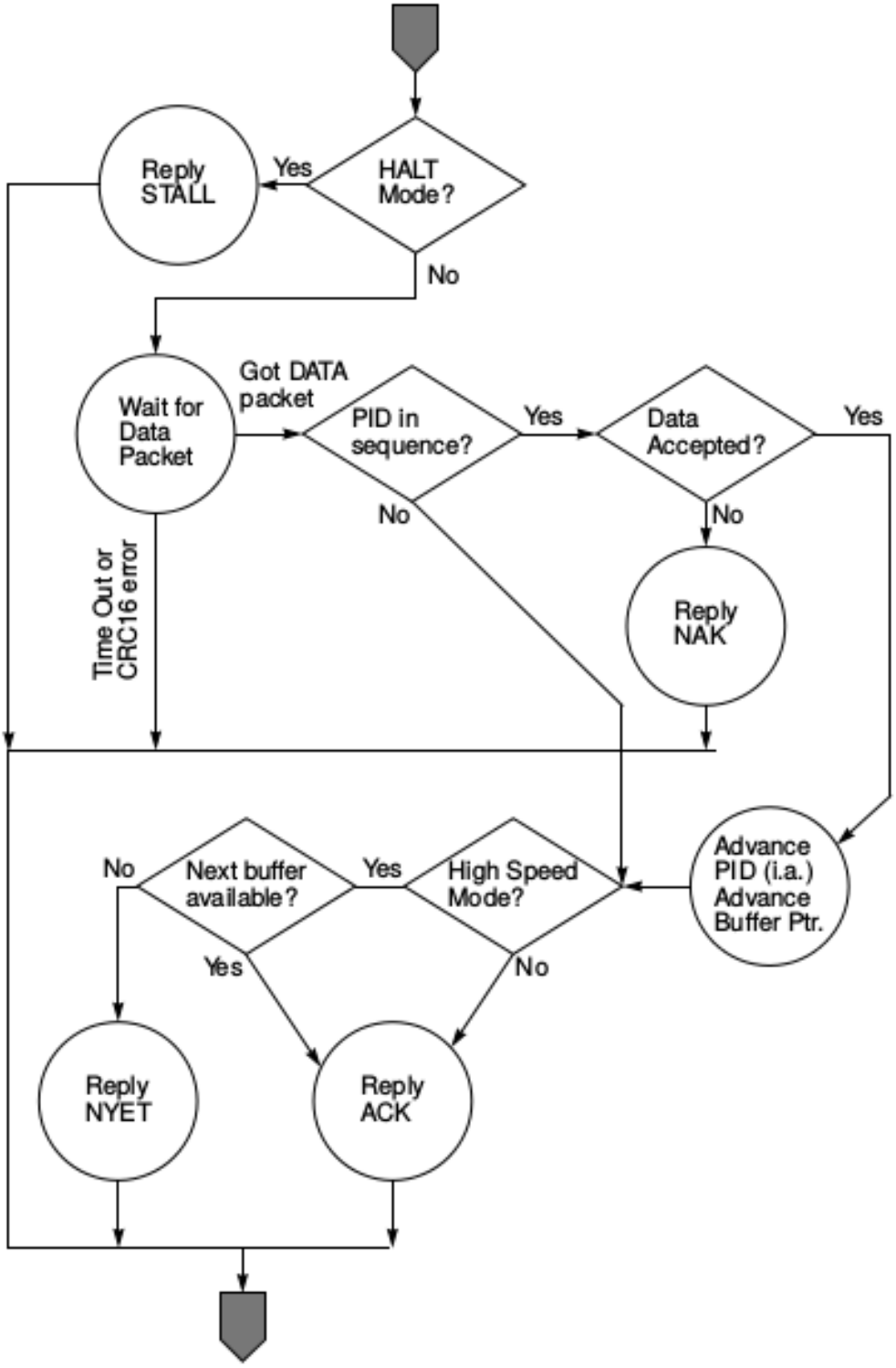


Figure 4.12: OUT Data Cycle Flow chart

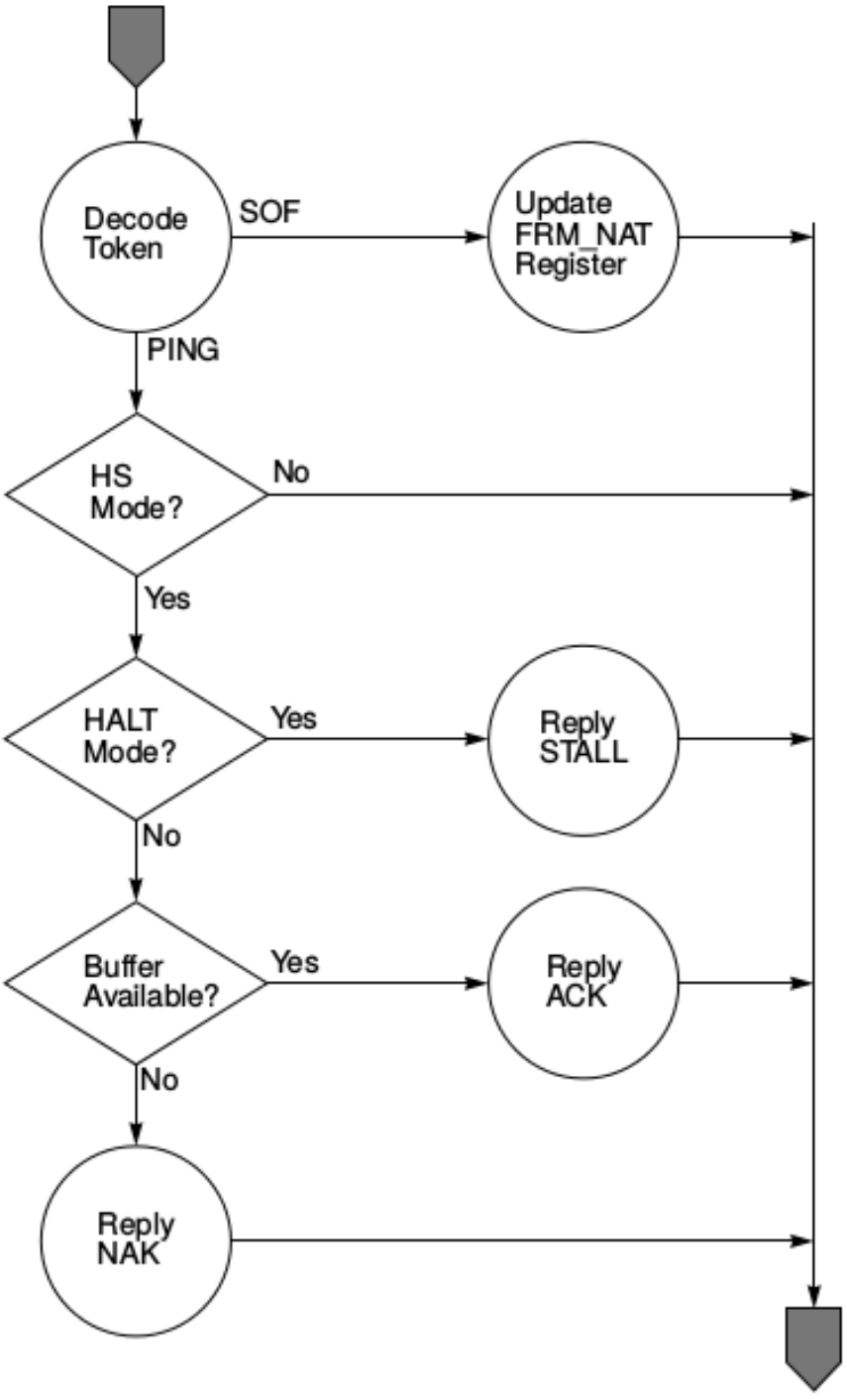


Figure 4.13: Special Token Processing Flow chart

CHAPTER 5

SIMULATION AND RESULTS

5.1 Packet Disassembly Simulation

Packet Diassembler receives packet through UTMI interface, decodes it to know the information contained in the packet.It extracts the PID information and sends the decoded PIDs to protocol engine.It extracts rest of the information from the packet and performs error check and reports to the protocol engine.

5.1.1 Token Packet Disassembly

Figure 5.1 shows the simulation result of disassembling the token packet.We know that token packet consists of PID, address field, endpoint field and CRC field.The token packet is received through UTMI interface through 8-bit sigal rx_data.The data will be valid only when signal rx_valid and rx_active is high and rx_err is low.Token packet consists of 24 bits, first 8 bits is the PID information. In the figure PID information is E1.PID information is extracted when pid_ld_en signal is high and this PID information is stored in m_pid register.Rest of the 16 bits of information containing address field, endpoint field and crc field is stored in token0 and token1 register.When load enable for token0 becomes high i.e token_le1, token0 stores the next 8 bit information succeeding PID information.Here token0 stores E1.When load enable for token1 becomes high i.e token_le2, token1 stores the last 8 bit information of token packet.Here token1 stores last 8 bits that is 6C.From the

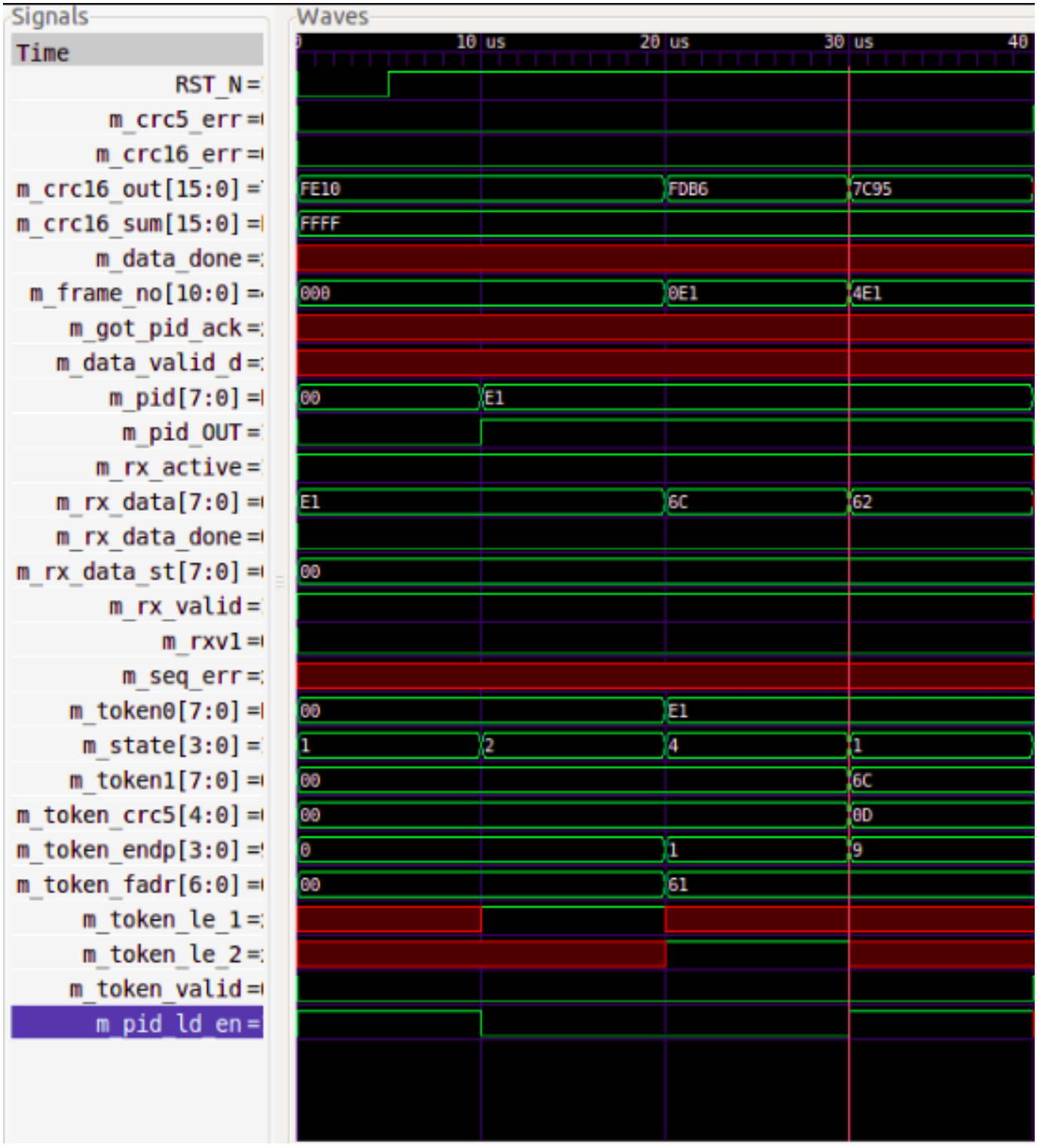


Figure 5.1: Disassembling Token Packet

PID information it is inferred that the token packet is OUT packet, so pid_OUT signal becomes high.With the information contained in token0 and token1 register, endpoint field and address field information is extracted and stored in token_endp and token_fadr registers respectively.

5.1.2 Data Packet Disassembley

Figure 5.2 shows the simulation result of disassembling the data packet.Data packet consists of PID, data field and CRC field.Data packet is recieved through 8-bit rx_data signal.rx_active signal is high and rx_err signal is low that means data can be received.First 8 bit of data packet consists of PID bits, in the figure C3 is PID data.Next bits after PID bits contain data payload, here next 8 bits i.e E1 is data payload.Last 16 bits of data packet consists of CRC check, here 6C62 are the CRC bits of data packet.When pid_ld_en signal is high,PID information i.e C3 is extracted and stored in pid register.From the PID information it is inferred that it is data packet and the type of data packet is DATA0, so pid_DATA and pid_DATA0 signal become high.Main goal is to extract data information from the data packet. CRC information is just for error check, it should not be extracted.Data information is stored in delay registers d0, d1 and d2, the purpose of these registers is to delay the extraction of data information so that CRC data is neglected successfully.When rx_data_done signal becomes high and at the same time rx_data_valid is also high, the data information i.e E1 is stored in rx_data_st register.This stored data information in rx_data_st register becomes input to IDMA engine, where it does the further transaction as directed by protocol engine.

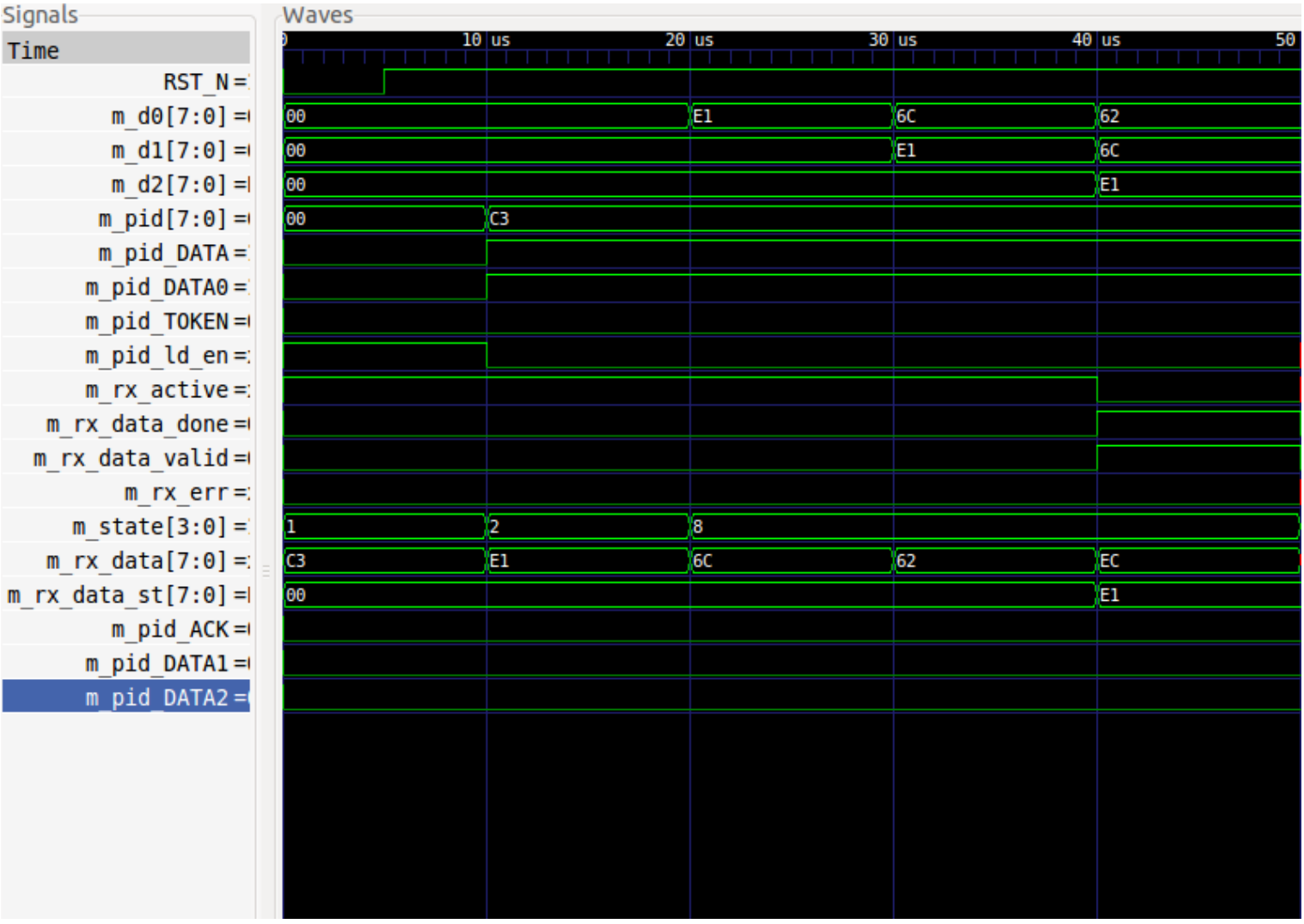


Figure 5.2: Disassembling Data Packet

5.2 Packet Assembley Simulation

Figure 5.3: Assembling Data Packet

Figure 5.3 shows the simulation result of assembling the data packet.Packet assembler assembles the packet according to the instructions given by protocol engine.It receives the data information sent by IDMA engine through 8-bit tx_data_st signal.It assembles the data packet by first inserting data PID, then inserting data information and at last inserting 16 bits CRC information.CRC 16 value is stored in crc16_REV register and its value is 80C1. signal data_pid contains the PID value of data packet and its value is 87, send_data signal is high so this PID value is put

first in the tx_data register.The data received from IDMA engine is contained in tx_data_st signal and its values is A9.Now this data value is inserted in the tx_data register.After thar send_data signal becomes low that means no more data need to be sent.When crc_sel1 signal goes high then higher order 8 bits of CRC16 (here it is 80) are inserted in tx_data register.When crc_sel2 signal goes high then lower order 8 bits of CRC16(here it is C1) are inserted in tx_data register.So data packet is assembled, tx_ready and tx_valid signal is high so this data is sent to UTMI interface.

5.3 Protocol Engine Simulation

Protocol engine receives decoded information from packet assembler and it reads the information stored in register files i.e CSR register. Based on this information it comes to know that which type of transaction to perform Isochronous, bulk or Interrupt and the direction of transaction IN or OUT.Then it instructs packet assembler and IDMA to perform required transaction.

5.3.1 Isochronous IN Transfer

Figure 5.4 and 5.5 show the simulation result of Isochronous IN transfer,that means data flows from device to host and transfer type is Isochronous.From the received CSR register information it is inferred that the buffer0 is to be used, endpoint type is IN and transfer type is Isochronous.So txfr_iso signal becomes high to represent Isochronous transfer, in_ep and pid_IN signal become high to indicate IN cycle.Maximum payload size(here 7C0) and endoint number(here 1) information is obtained through CSR register and represented by max_pl_sz and ep_sel sig-

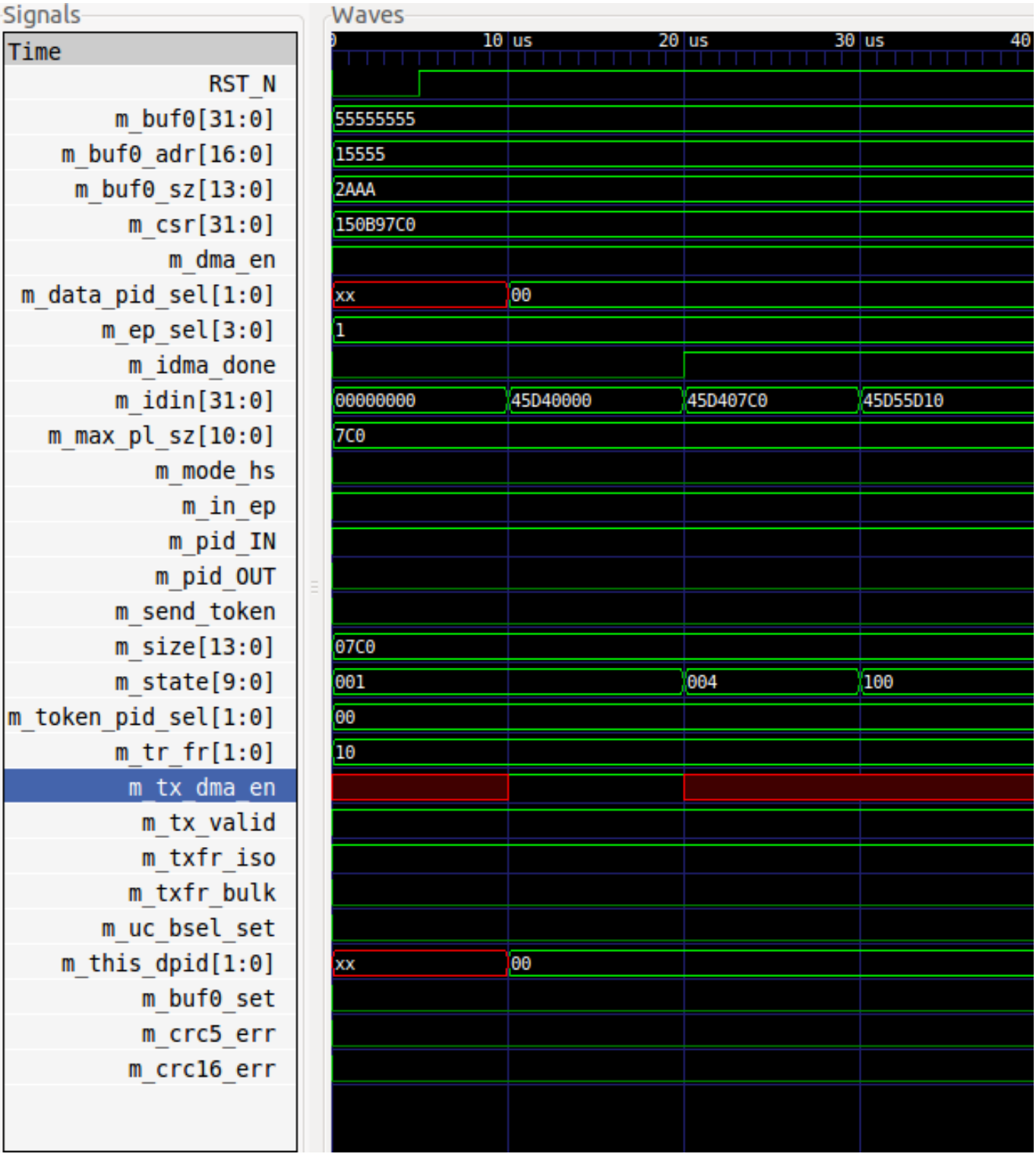


Figure 5.4: Isochronous IN Transfer

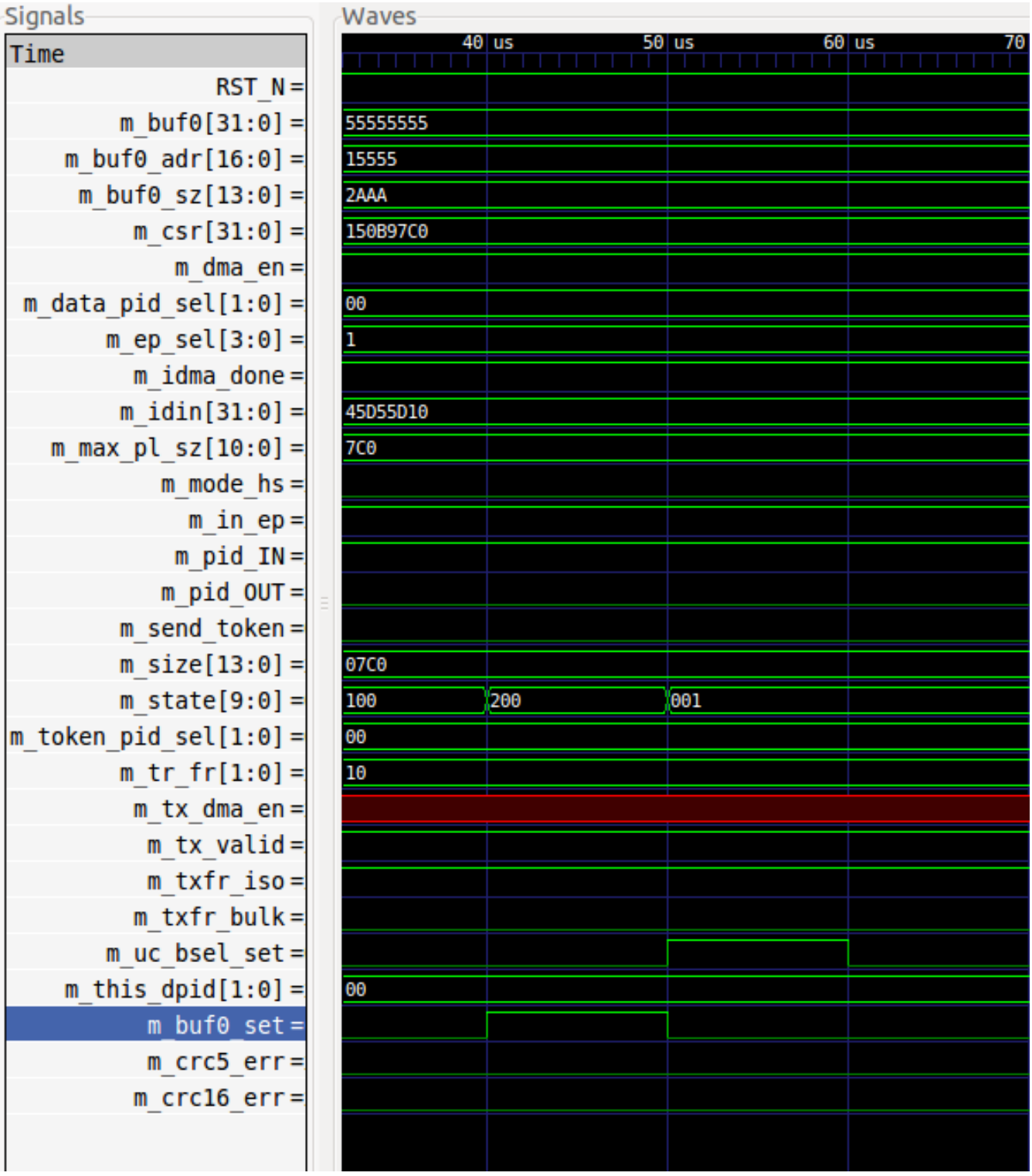


Figure 5.5: Isochronous IN Transfer continued..

nal respectively.Signal data_pid_sel indicates the type of data to be transfered i.e DATA0,DATA1,DATA2 or MDATA.Here data_pid_signal value is 00 that means DATA0 transfer will take place.Buffer0 address i.e 1555 and size i.e 2AAA are represented by buf0_adr and buf0_sz respectively.Signal dma_en is set to high to enable DMA operation whenever required.crc5_err and crc16_err signals are low, these are the error check information received through packet dissembler.After reading CSR register information and packet dissembler information Protocol Engine comes to know that it has to do Isochronous IN transcation, so Protocol Engine instructs IDMA engine to start the transaction by making tx_dma_en signal high.IDMA engine will do the transaction and it will indicate end of the transfer by making idma_done signal high.In Isochronous IN transfer, USB core does not wait for acknowledgement signal from host.After that protocol engine will set the bufo_set signal high, that means reload buf0 with next data.

5.3.2 Bulk OUT Transfer

Figure 5.6 and 5.7 show the simulation result of Bulk OUT transfer,that means data flows from device to host and transfer type is Bulk.Like we have seen in case of Isochronous IN transfer here also CSR register is read and it is inferred that buf0 is to be used, endpoint type is OUT and transfer type is bulk.So out_ep signal turns high to represent OUT endpoint and txfr_bulk signal turns high to represent bulk transfer.Buffer0 address and size are represented by buf0_adr and buf0_sz respectively.Endpoint number 1 is selected that is shown by ep_sel signal, max_pl_sz represnts maximum payload size(here it is 7C0).Here data_pid_sel value is 01 that means DATA1 transfer will take place.Signal pid_OUT is high that says that direction of transfer is OUT i.e from host to device.Now protocol engine has to instruct the IDMA engine to perform OUT transaction, it does that by turning rx_dma_en signal high(refer Figure 5.6).Now DMA operation takes place until rx_data_done signal becomes high.IDMA engine indicates the end of the transfer by turning idma_done signal high.Afer the successful transfer acknowledgement(ACK) signal is sent,ACK is represented by the value 00 on token_pid_sel signal.Signal send_token turns high(refer Figure 5.7) instructing packet assembler to send handshake(ACK) packet.Afrer that buf0_set signal becomes high that means buffer 0 should be reloaded with next data.

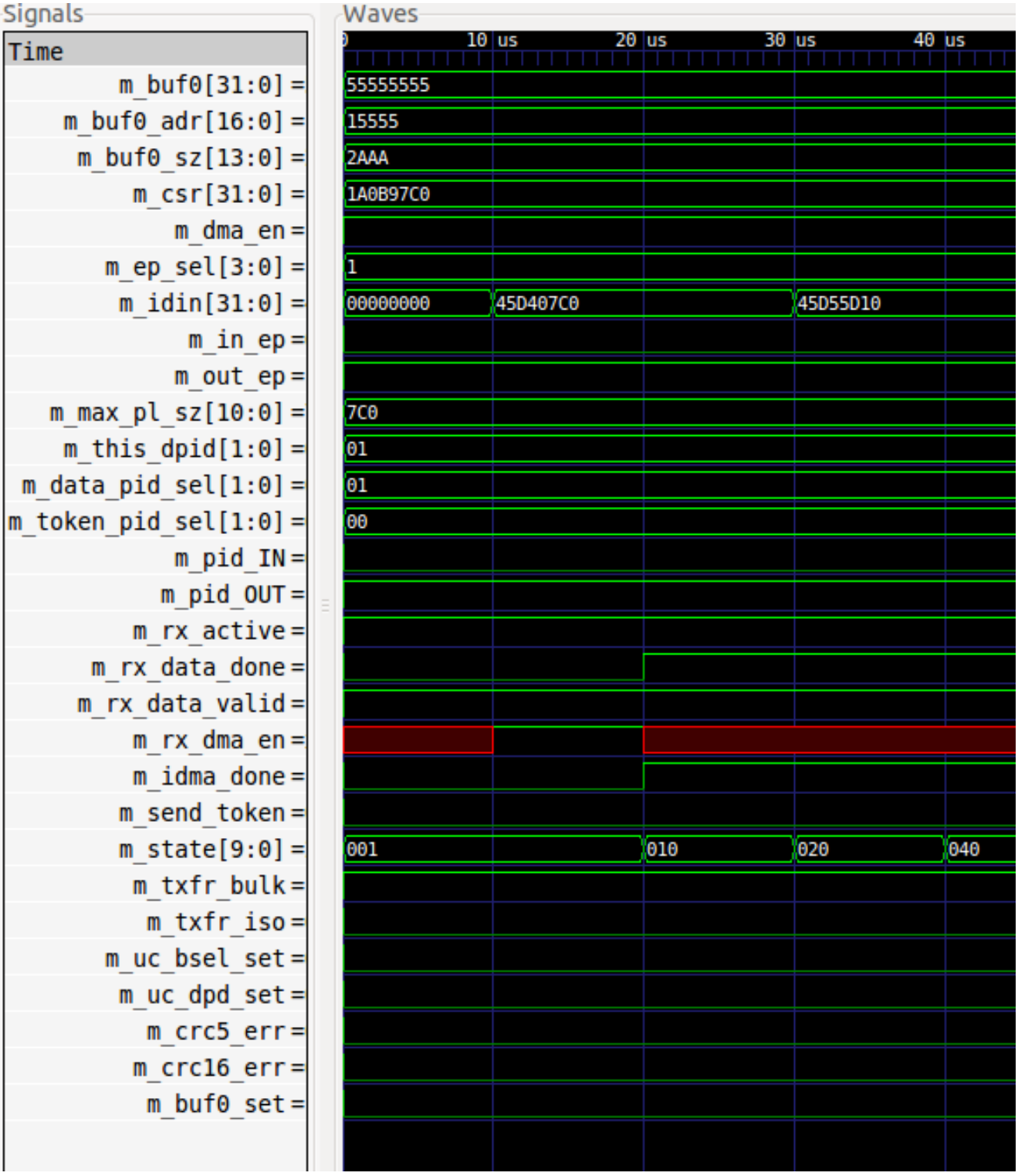


Figure 5.6: Bulk OUT Transfer

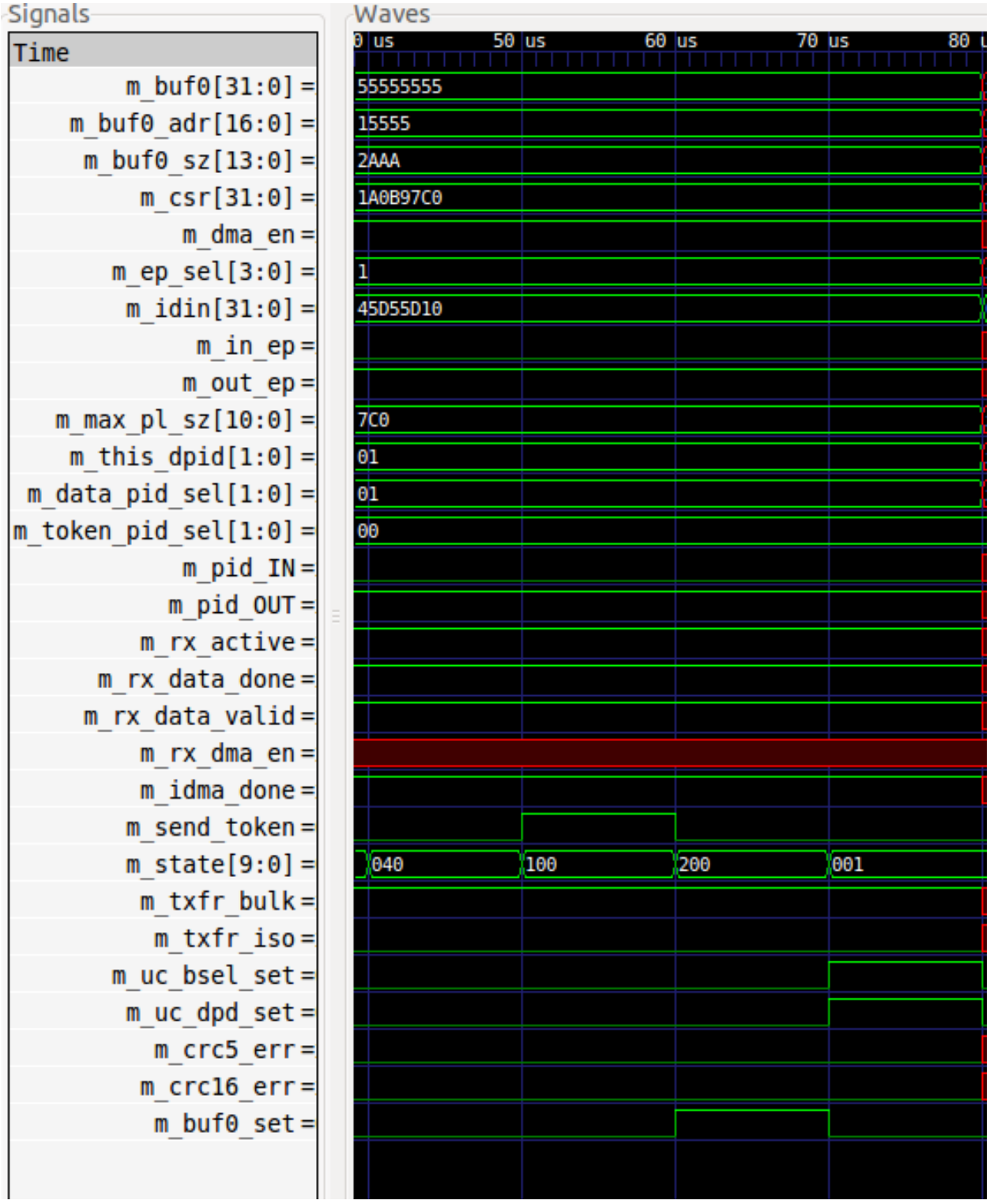


Figure 5.7: Bulk OUT Transfer continued..

5.4 Internal DMA Engine Simulation

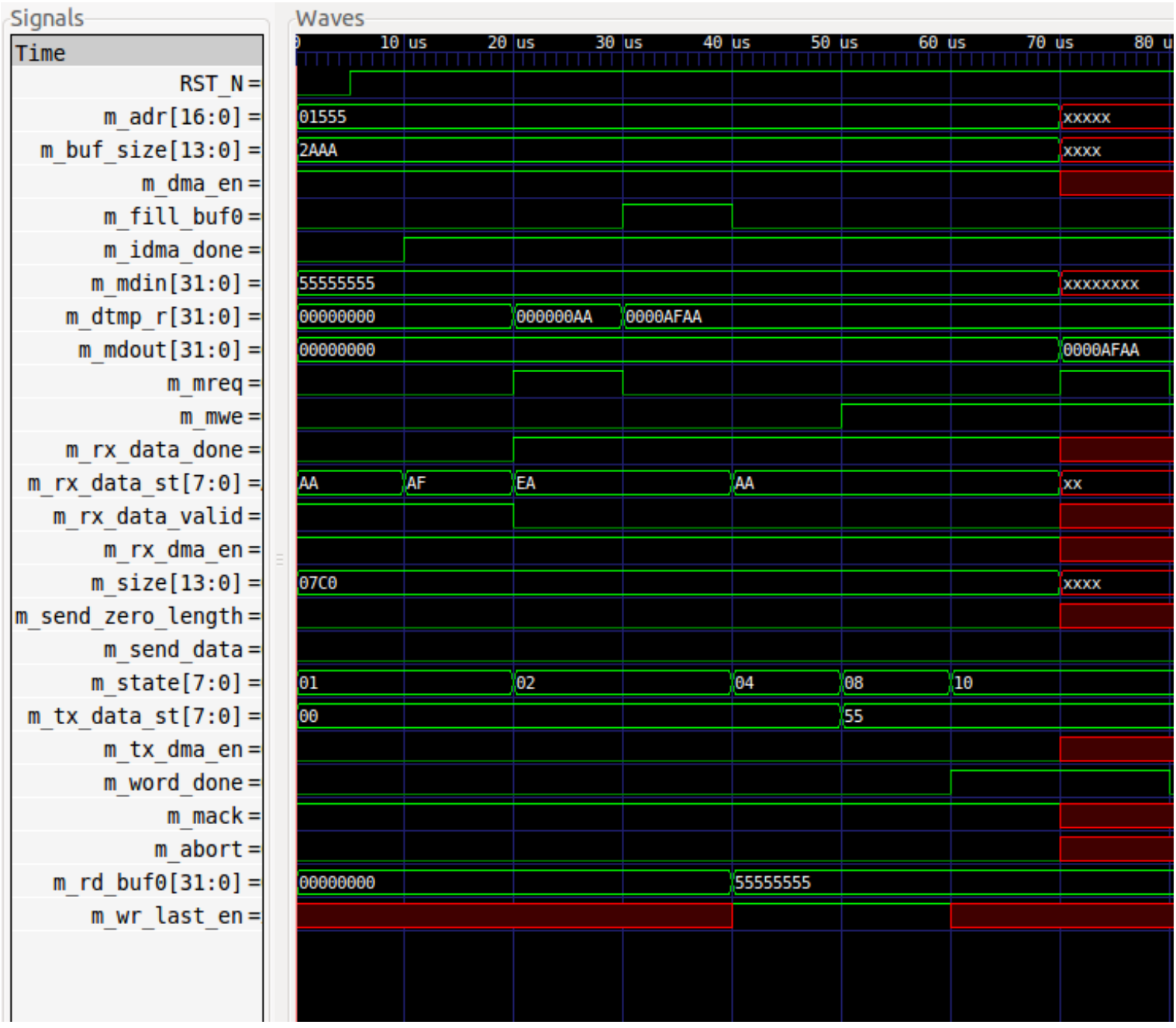


Figure 5.8: Internal DMA Engine OUT cycle

Figure 5.8 shows the simulation result of Internal DMA Engine for OUT cycle.Basically IDMA Engine transfers data between memory and UTMI Interface.During OUT cycle it takes data from packet dissembler and stores it into memory and during IN cycle it takes data from memory and gives it to packet

assembler. Address and size of the buffer are represented by signals adr and buf_size respectively.Contents of buffer0 register are stored in rd_buf0 register.The signal rx_data_st contains the data coming from packet dissembler and needs to be stored in the memory.Here memory is represented by mdout register,dtmp_r is the register that temporary holds the data and sends it to mdout register.Signal rx_data_valid is high that means data is valid and can be transferred.The signal rx_dma_en is high that means IDMA engine will start the data transfer, data transfer will be done until rx_data_done signal becomes high.rx_data_done becomes high after two cycles, so only the data AA and AF should be transferred to mdout register.mreq is memory request signal sendt by IDMA Engine to Memory Arbiter Interface requesting for memory access.The signal mwe stands for memory write enable, it turns high to indicate that data is to be written to the memory.The register mdout stores the data sent by packet dissembler, here it stores first two bytes of data that is AFAA.The data which is sent first is stored first so the final contents of mdout register becomes 0000AFAA.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The USB 2.0 Function Core provides functional Interface between USB peripherals and Host or Computer and enables data transfer between them.As We know that heart of the Function Core is protocol layer, that handles all the standard USB protocols.Protocol layer simulation was successfully done.Protocol Layer consists of different modules including packet assembler, packet disassembler, protocol engine and Internal DMA Engine.Each of the module of the protocol layer is simulated for different inputs and results were verified by checking the waveforms through Bluesim simulator and gtkwave.USB 2.0 supports differnt types of transfer such as Isochronous Transfer, Bulk Transfer and Interrupt Transfer.Simulation for protocol engine is done for both Isochronous and Bulk transfer for IN and OUT cycle respectively by providing the test inputs through testbench and verifying the expected output through gtkwave waveforms.Internal DMA Engine is simulated for OUT cycle that is host to device transaction.So the standard USB protocol that consists of token packet, data packet and handshake packet are verified through simulation.

The code for UTMI Interface and Register file interface is successfully written but In future the goal is to interface all the blocks that is protocol layer, UTMI Interface and Register file Interface successfully so that they can be combined to form USB 2.0 Function Core and that can be synthesized on FPGA to give the hardware that is required.

REFERENCES

[1] *SL811HS Embedded USB Host/Slave Controller From Cypress,Document 38-08008*. Cypress Semiconductor Corporation, 2005.

[2] *Universal Serial Bus Revision 2.0 Specifications*. Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics N.V, April 27,2000.

[3] *USB 2.0 Transceicer Macrocell Interface(UTMI) Specifications*. Intel Corporation, March 29,2001.

[4] **Nikhil, R. S.** and **K. R. Czeck**, *BSV By Example*. Bluespec Inc., 2010, 1.0 edition.

[5] **Usselmann, R.**, *USB Function IP Core*. January 27,2002.