Chapter 1

Introduction

In this chaper, we give an overview of the relevant concepts about the Circuit Emulation Adaptor (CEA), focusing on its functionality as well as timing specifications. Thereafter, we introduce the general framework of the corresponding hardware implemenation of the adaptor.

1.1 Voice Traffic Over Data Networks

The proliferation of switched Gigabit Ethernets on top of wavelength division multiplexing equipment in metropolitan areas directs networks equipment manufacturers toward building circuit emulation adapters. These adapters shall protect their customer's investments in existing private branch exchanges (PBXs) and GSM base stations. Given this situation, the joint CTI Circuits over Packets (CoP) technology transfer project between TIK/ETH and network equipment manufacturer Siemens-Schweiz heads toward laying the foundations for implementing such adapters.

1.1.1 Circuit Emulation Adaptor

In the traditional voice telecommunication setup, dedicated circuit-switched based infrastructures are deployed to ensure the availability and quality of the communication. In contrast, Internet and other data communcation networks sit on top of the packet routing regime, providing mainly best effort services. With the recent significant capacity expansion of data communication networks, voice over packet-based data network becomes possible. Circuit Emulation Adaptor is therefore proposed to bridge the legacy accesss telephony equipment to the relative new core data tranportation networks (see figure 1.1). This approach minimizes the conversion cost and, thus, is

of particular interest to the commercial service providers.

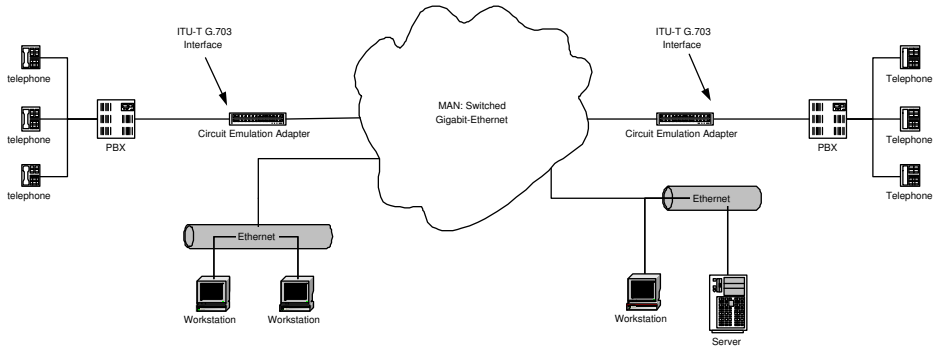

Figure 1.1: Proposed Voice over Packet Network Conversion Setup

1.1.2 Importance of Time

The interface from the traditional telephony access networks (PDH) is defined by ITU-T G.703 while the interface to the data transportation networks is based on the Gigabit Ethernets i.e. IEEE 802 family. The difficulty of the conversion from circuit based voice traffic to packet based data traffic lies in the process of packetizing the incoming voice traffic. More specifically, the CEA should accurately synchronize the receiver buffer play-out rate of the receiving adapter to the packetizing rate of the sending adapter to prevent receiver buffer over- or underflows within several days of operation (see figure 1.2). Moreover, for multiple PDH interfaces, prevention of over- or underflow also applies to the synchronization of peer interfaces across the emulation. This requirement is necessary since PDH signals are plesynchronous, i.e. the signal frequency has a $\pm$50ppm tolerance, which has to be preserved. This preservation is necessary to avoid a violation of the MTIE (maximum time interval error) requirement for PDH signals. It should also be noted that the synchronization requirements and the Quality of Serice (QoS) requirements for circuit emulation are much more stringent than the corresponding requirements for VoIP.

1.2 Hardware Realization

The synchronization algorithms and realization guidlines of CEA, developed under the joint technology transfer project between Computer Engineering and Networks Laboratory and Siemens Switzerland, have been evaluated

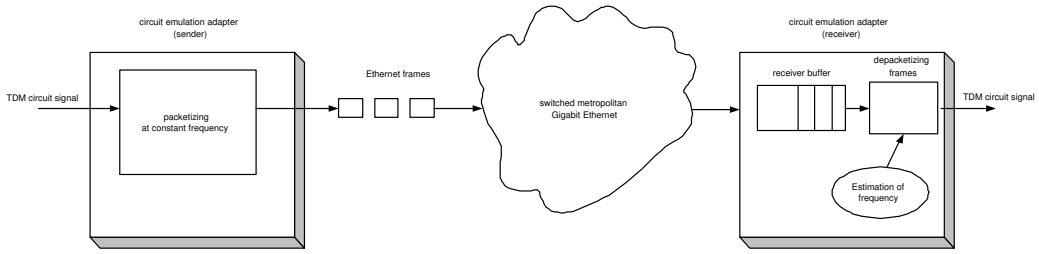

Figure 1.2: Frequency Synchronization

through computer simulation with satisfactory result. To prove the effectiveness of the proposed synchronization algorithms, a simplified version of CEA, or CEA Demonstrator, is to be implemented in hardware logic. The implementation includes the synchronization algorithm as well as a protocal stack and basic timing functionality so that the performance of the synchronization algorithm can be measured.
More specifically, the CEA board is divided into two parts. One is to be fully implemented by hardware logic, which includes the complete network communication protocol stacks such as IEEE 802.3, UDP/IP, RTP and etc. The other is a mini-CPU itself or micro-Blazer. It provides an operating system in which the synchronization algorithm is implemented in C. The communication between these two parts is enabled through the synchronization clock, which signals the start of an MAC frame. This technical report focuses on the hardware implementation of the logic part of the CEA board.

1.2.1 Design Requirements

The essential funcationalities of the CEA pertaining to the time synchronization algorithm include the process of the voice data from PDH interface, data communication through packet based Ethernet and finally the ability to cooperate with various clock signals. Accordingly, the following detailed design requirements are identified.

- Implementing the UDP, IP, RTP, and SaTOP layer of the protocol stack including static configuration for IP-Address, port number etc. that previously have been mapped into registers.
- Implementing a dual buffer at the sending side to continuously retrieve bytes from the PDH input signal while generating and including time stamps and sequence numbers.

- Implementing a multiplexer at the sending side that periodically flushes the dual buffers of (in the future hopefully more than one) interface that run in plesynchronous mode. This multiplexer is controlled with a frame rate of 255.3/8 Bytes to make payload either 255 or 256 bytes long.
- Implementing receiver time stamping functionality.
- Integration of existing synchronization algorithms in collaboration with the advisors.
- Functional testing of the board.

From the implementation point of view, the CEA boards can be broadly classified into two logical paths - the forward path through which the CEA board spits out the voice bytes and the backward path through which the CEA board receives the incoming bytes. The whole implementation, accordingly, is comprised of two phases. During each phase, we complete the entire design cycle for one path, including functionality analysis, interface design, state diagram specification, VHDL coding, hardware realization and testing. The advantage of this divide-and-conquer approach not only facilitates the debugging process but also serves to verify the feasibility of the design ideas.

1.2.2 FPGA

Digital circuits serve as essentially as functional mappings defined by the true table of inputs and outputs. The conventional way of implementing such a truth table is based on connecting combinational circuits composted logic gates such as AND, OR as well as NOT. This approach is rather time consuming and resource inefficient.
Since all logic mappings can be realized by either Sum Of Products (SOP) or Product Of Sums (POS), we can build up arrays of "AND-OR-INVERT" structures and connect them with programmable interconnections. Once the truth table is specified by VHDL codes, the corresponding logic can be realized by "fusing" the interconnections among the structures.
This is the central principle of the commercial Field Programmable Gate Arrary or FPGA. Typical FPGA is a collection of logic modules and block that are interconnected through programmable connections. The *Logical Blocks* are used to implement combinational and sequential logic and they are connected by *fusible wires*. Finally, they are surrounded by the *I O Blocks* for external connections.

To be more specific with Xilinx FPGA, which is used in the actual implementation, we give an overview of the Xilinx FPGA by mentioning its building blocks. Xilinx FPGA consists of three major segments, namely, Configurable IO Blocks or IOBs for short, Configurable Logic Blocks (CLBs) and Interconnects. The IOBs provide a programmable interface between internal logic array and IO pins. The CLBs perform user-specific logic functions. And the interconnects are programmable to form networks on the FPGA chips.
To sum up, the FPGA provides a collection of programmable logic components. And once the complete logic of a device is fully specified by the VHDL codes, we can simply implement the logic by "fusing" the FPGA chips.

Chapter 2

Design Methodology, Principle and Hardware Constraints

The essence of digital design lies in the ability to read in sequences of '1's and '0's and output the corresponding sequences at the right time. The design methodology,therefore, helps to produce the circuit logic that connects the input to the output in a certain logical binding manner.
We start with the traditional design technique based on Medium Scale Integrated (MSI) Circuit Models and translate it into the clocked synchronous state machine design afterwards.

2.1 From Concept to Practice

Perhaps the most intuitive way to represent a logic design problem is to form a conceptual mapping from the specific design functionalities to the corresponding MSI devices e.g. data buffers, multiplexers and etc. The modern digital design tools, however, enable us to deal with design from a different perspective, namely, through specifying the clocked synchronous behaviour of the target design entity. In the following two sections, we will first illustrate the design from a functional perspective before switching to the behavioral state machine model, on which the actual implementation is based.

2.1.1 Functional Perspective

Although the eventual implementation is not strictly based on the funcational specification of the design, the analysis does generate insights that are otherwise not visible from the other perspective. We show the complete

design in fig 2.1.
The complete CEALite is divided into two parts i.e. forward path and backward path. The forward path interfaces with the external T3 station clock and generates the dummy telephony traffic. It then stamps the data with the proper clock information from the IP_T0 module before it adjusts the appropriate counter values. Finally, it encapsulates the data packets with SATOP/RTP, UDP as well as IP headers and forwards the complete frame to MAC transmitter.
The backward path performs the reverse operations of the forward path. Namely, it stamps the recipient time of the incoming frame, strips off the headers from the incoming frames, stores the relevant counter values to the block RAM and, finally, output the payload data for storage.
Both forward path and backward path interface with the IP_T0 clock module, which is, in turn, controlled by the Microblaze. Microblaze is a mini operating system that runs on Xilinx FPGA device. The synchronization algorithm written in C is implemented in the Microblaze. In other words, the Mircroblaze performs the synchronization adjustments through the IP_T0 clock module and its intelligence comes from the synchronization algorithm in C. The Mircoblaze integration as well as the software programming are out of the scope of this project. We restrain our interface only up to the IP_T0 clock module.

Dual Buffer Overflow and Transmission Delay

As mentioned earlier, the targeting rate of synchronization clock signal is at a frame rate of 255.3/8 Bytes. This is equivalent to an time interval about 0.96 millisecond. Since the clock from PDH interface has a maximum deviation of 4.6 ppm (parts per million), this translates to 0.0046 bits during the 0.96 millisecond interval. In other words, to accumulate an error of 1 bit, we need to incur the maximal possible deviation for over 200 continuous bits. This is highly unlikely and, hence, provided that the synchronization algorithm functions properly, we will encounter no buffer overflow issues and incur no tranmission delay caused by the overflow.

2.1.2 State Machine without Hardware Constraints

After looking at the design from the intuitive picture, we now switch to the behavioral model. Instead of focusing on how we can implement the specific functions, we shall look at what the system needs to do at the right time. This allows us to define a highly logical and systemic timing behavior of the system, leaving the detailed gate level implementation to the software tools.

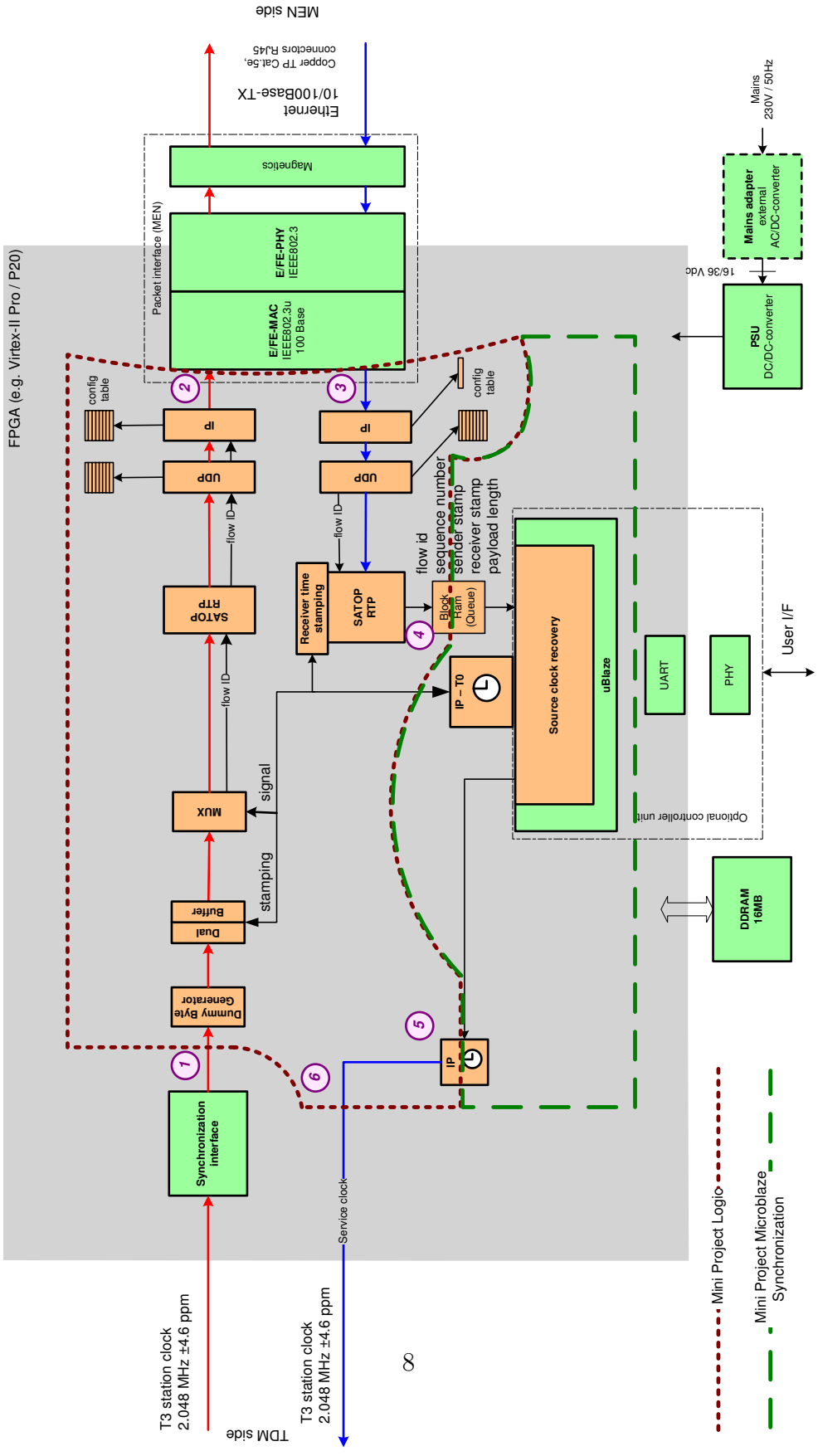

Figure 2.1: MSI Circuit Model of CEA Demonstrator

In this manner, we translate the design requirements as a State Machine, which is treated as a 'black box' as far as the detailed implementation is concerned. The whole state machine is completely specified by the number of states, the corresponding output signals as well as the possible transitions from one state to another.
Pursuant to our design, our state machine model must deal with controlling the MAC layer communications module, accessing external memory blocks and working with the synchronization clock signals. For example, the state machine should activate the MAC transmission as soon as the synchronization clock requests the current MAC frame to be transmitted.
To comprehensively capture the parameters of the State Machine, we adopt the notation of Algorthmic State Machine (ASM) chart. ASM chart bears a superficial resemblance to flow charts. The major difference is that ASM charts have the concept of a sequence of time intervals built in. Flow charts decribe only the sequence of events, not their duration. In ASM charts, the machine changes from present state to the next state at the active clock transition, and remains there for the duration of the clock cycle. Hence, everything inside the rectangle takes zero time to execute while the signal will be only effective at the next clock transition.
Typically, the name of the state is enclosed in a circle. The state rectangular box includes the unconditional outputs in that state. If an output signal also depends on the value of an input signal, it is called conditional output. This is indicated by writing the signal in an oval box. When the state transition is dependend upon certain input signal, the signal is enclosed by a decision diamond, which is a part of the state. Kindly note that the test of the signal does not require a separate clock period - it is done in the same state.
We are now in a position to discuss the proposed ASM charts for the forward path i.e. the transmission path of the CEA. We show the complete chart in fig 2.2.
There are altogether three different states in the design. In State 0, the state machine mainly reads the traffic data from PDH and prepares for later MAC layer transmission. State 1 is the state when the actual MAC layer transmssion starts while, at the same time, the state machine continuously monitors the traffic from PDH interface. In the last state, State 3, the state machine stops the MAC layer transmission and returns to the intial state. We discuss each state in details in the following sections.

State 0: PDH READ, MAC START

As mentioned earlier, in State 0, there are two major tasks for the state machine. One is to read, store and post-process the voice traffic from the

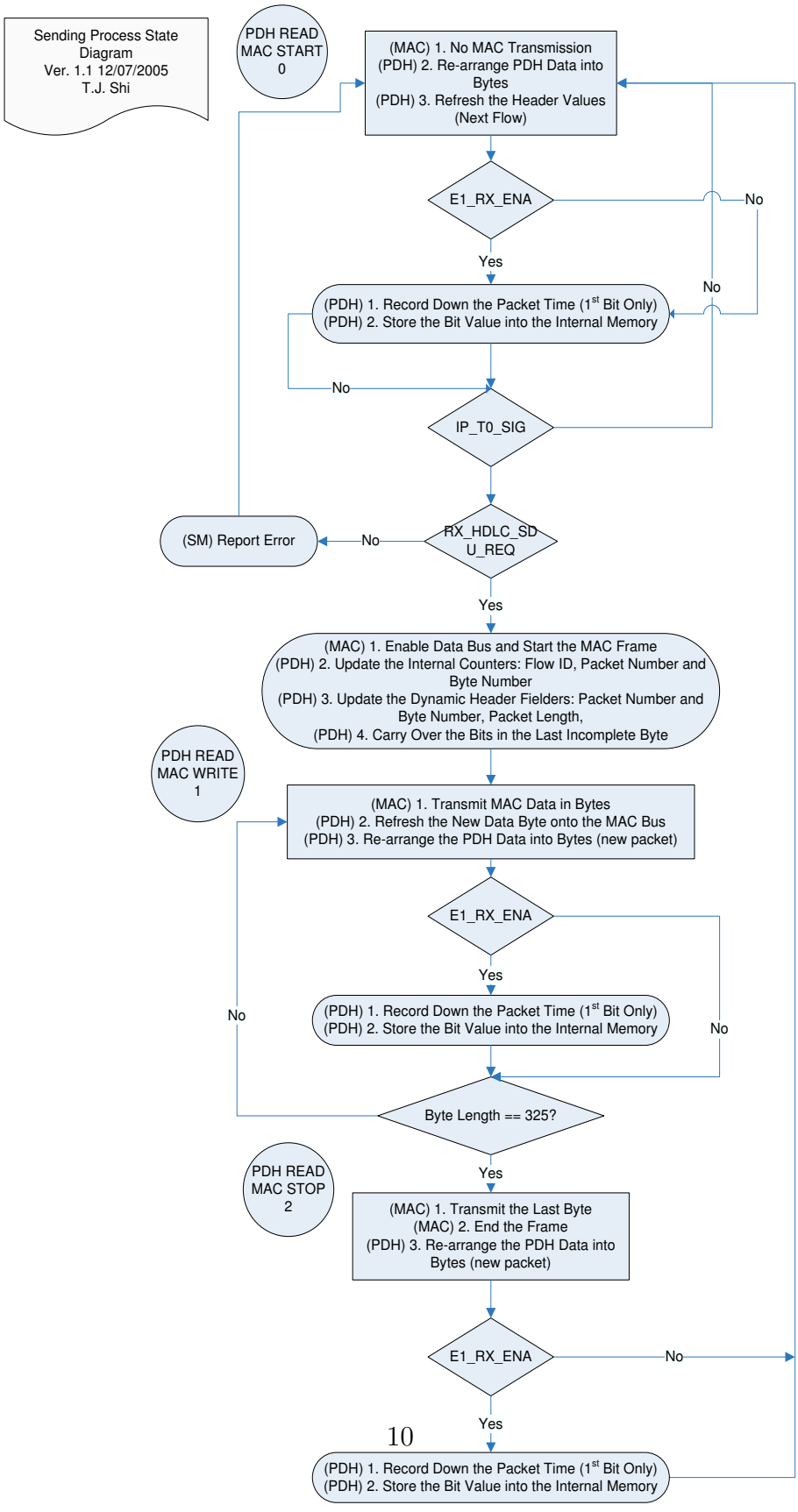

Figure 2.2: ASM Charts of the Forward Path of CEA Design

PDH interface. In particular, we need to re-assemble the data in bit form into byte from since the latter is the internal data format for the state machine. Besides storing the payload data, the state machine also needs to refresh the header information of the entire frame from MAC layer onwards. These 70 Byte header information comes from the MAC layer, the LLC/SNAP layer, the IPv4 layer, the UDP layer, RTP layer as well as the SAToP layer. Most of the fields are static in nature; thus, they are stored in a Block RAM. Some are time sensitive so they must be updated very frequently. The detailed memory allocation schemes are treated in the next chapter.
The second major task of the intial state is to send relevant signals to start the MAC transmission in the next clock cycle as soon as the synchronization clock dictates so. This includes flashing in real time the relevant counter information to the header fields, sending the appropriate control signals to the MAC transmitter and initializing the backup buffer to allow parallel transmission and receiving.

State 1: PDH READ, MAC WRITE

State 1 does the same job as State 0 as far as the PDH side is concerned. The major difference is that the actual MAC transmission of PDH data is handled in this state in parallel with processing the data traffic from PDH interface.
More specifically, a transmission loop is maintained inside this state to spit out bytes in each clock cycle. Since we have internally constructed the entire MAC frame before the actual transmission, the process is relatively simple. It simply fetches the byte one after another each clock cycle and stop when the counter reaches the fixed frame size. We, however, need a seperate state to stop the MAC transmission.

State 2: PDH READ, MAC STOP

State 2 exists only for one clock cycle and it does exactly the same as State 1 except for only one signal, which is raised high to stop the MAC transmission. It still stores new PDH data (if any). The state machine returns to State 0 unconditionally after State 2.

2.2 Implementation of ASM Charts

ASM Charts depict the complete behavior of a state machine. To translate it into a fully functional hardware module, we make use of the hardware design languages or HDL, namely VHDL and Verilog. Both languages serve

the same purpose but with a different modelling style. We choose VHDL as the HDL in this project for the ease of integration since the codings from Siemens are done in VHDL, too.
As HDL programming is by no means the topic of this thesis, we shall instead briefly mention the capability of VHDL that facilitates this implementation. In essence, VHDL allows us to describe the behavior of the state machine exactly the same way as ASM charts. Once we have translated the ASM charts into VHDL program, VHDL compiler is able to automatically map the behavioral state machine to logical gates, which can be then burned into the programmable logical gates. We shall demonstrate this process briefly in Chapter 7.

Chapter 3

Memory Access Scheme - Block RAM Specification

In this chapter, we discuss the major information flow in the context of State Machine design as well as the relevant memory allocatoin schemes to store the data. Firstly, we classifly the information flow into three categories. Based on this classification, we propose the design strategy that minimizes the design complexity and facilitates the information communication with the State Machine. It is noted that optimization of resource utilization is of lower importance in the design. Lastly, we illustrate the detailed memory allocation scheme.

3.1 Sources of Information

The major information is classified into Dynamic, Configurable and Static in the descending order of their change frequency. All the counting based information is maintained as the counters inside the State Machine; thus, they are internal to the State Machine. Data Bit from PDH interface, Time Stamp and Transmission Ready Signal from IP_T0 Interface, are data and control signals dependent on the sources of separate clocks so they have to be made external. IP and Port address are classified as external to facilitate possible future modifications.
The following list states the signal name, followed by their respective sources. The brackets enclose their corresponding classifications.

- Data Bit - PDH Interface *(Dynamic)*
- Time Stamp -IP_T0 Interface *(Dynamic)*

- Transmission Ready Signal - IP_T0 Interface *(Dynamic)*
- Source IP Address/Port Address - Block RAM *(Configurable)*
- Destination IP Address/Port Address - Block RAM *(Configurable)*
- Packet Counter - Internal *(Dynamic)*
- Byte Counter - Internal *(Dynamic)*
- Packet Length - Internal *(Dynamic)*
- Flow ID - Internal Counter *(Dynamic)*

  Flow ID is stored in the sequential counter inside the State Machine. The counter is incremented once every time the State Machine receives a Transmission Ready Signal, which flow switch in the next clock cycle.
- Other Fields - Block RAM *(Static)*

3.2 Design Strategy

Strategy is nothing but deciding on trade-offs. We therefore list the design trade offs and the corresponding choice made with the goal to minimize the design complexity and facilitate information flow with the State Machine. Note that the choice will be quite different if our focus is instead on resource optimization - in this case, we will then need to apply complicated design techniques in order to save memory space and etc.

3.2.1 External v.s. Internal

In our design, we store the relevant information as much as possible as internal data units inside the State Machine so as to minimize the data communication from the State Machine to the external interfaces. This includes the Packet Counter, Byte Counter and Flow ID. Except Flow ID, they are maintained inside the State Machine as internal counters as a per flow basis.
This approach is, certainly, in contrast to the traditional modular design method, which enhances the system robustness and faciliates error localization.

3.2.2 Interface Complexity v.s. Access Complexity

Since the external data are of different sizes, e.g. Port Number is 16 bits while IPV4 Address is 32 bits, we face the trade off between Interface Complexity v.s. Access Complexity. If we allow different types of external interface data bus, access to the data is much simpler. On the other hand, if we only allow uniform memory access interface for the ease of State Machine design, we face more complicated data access challenges. We choose the later as we stick to the design strategy mentioned earlier.

3.2.3 Block RAM Efficiency v.s. Access Complexity

Another factor that complicates the data access is the Block RAM efficiency. If we store different types of data (dynamic, static and etc) on difference pieces of Block RAM, we save perhaps hugely on memory usage. However, the State Machine is then required to be equipped with multiple access interfaces and pooling the small pieces of data at exact instant of time. This requirement obviously burdens the State Machine design; therefore, we will trade off Block RAM Efficiency for the simplicity of State Machine design.

3.3 Allocation Scheme

Allocation is a mapping from the different sources of data onto the proposed storage blocks. We show the mapping below. In the following design, we cater to 4 independent PDH interfaces. The external chips with interface connection to the State Machine are T3 PDH connection, IP_T0 synchronization clock and one Block RAM with 32 bit data bus and 2 KB storage capacity. In this case, we will need to have 4 PDH data buffers, 4 IP_T0 time stamp buffers, 4 packet length counters, 4 byte number counters and finally 4 header information buffers.
We illustrate the detailed mapping in Fig 3.1.

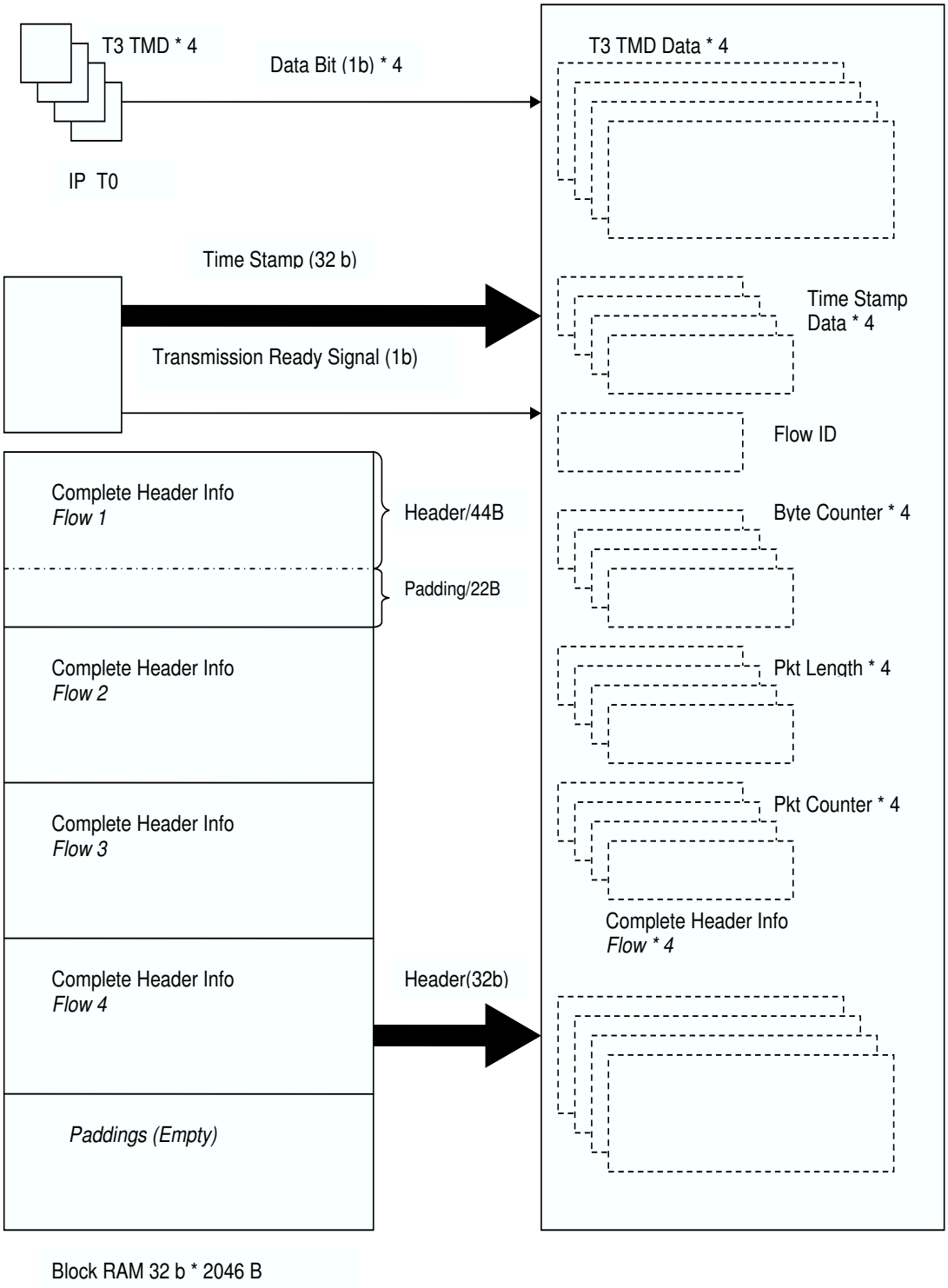

Figure 3.1: Memory Allocation Scheme

Chapter 4

Design with Industrial Constraints

In the previous chapters, a comprehensive introduction of the design of the CEA is discussed – primarily from the functional and logical persepectives. While everything presented is theoretically sensible, we still need to further polish the entire design so as to meet the industrial expectations, particularly in terms of hardware constraints and modular design requirements.
In this chapter, we should state the practical constraints that we face and propose modified new design scheme accordingly. It is our hope that with the background knowledge from previous chapters, the further complication of the design introduced here can be appreciated more easily.

4.1 The Constraints

For the real implementation, the design is often subject to severe hardware constraints. And in order to accomodate the hardware limitations, we will have to complicate the logic of the design and, consequently, even flow of the State Machines. Specific to our design, our initial assumption that the memory space of the FPGA is sufficient enough to store the entire MAC frame is, unfortunately, not valid. This is due to the cost considerations as the FPGA with the expected number of internal registers is simply too expensive. Consequently, we will need to store the frame data externally to the state machines and poll for the appripriate bytes at the right time during the MAC transmssion.
The second contraint that we need to live with is the modular design requirements. Contrary to our previous one-state-machine design principle,

modular design approach is ideal for future reuse of the existing implementations. In other words, we need to split the previously proposed state machine to several smaller components, each with more specific tasks. The overhead that we need to deal with is the extra logic to coordinate among these components. The advantages are neverthelss obvious - it is easier to reuse smaller components and the debugging is also simplified.

4.2 Modified Design Concept

As discussed previously, we are to strike a fine balance between the logic simplicity and design simplicity. The design sophistication comes from the dissolvance of the original one state machine and the external storage of the MAC frame data.
The modified design now consists of three seperate state machines - *the bit to byte converter, the data packaging state machine* and *the MAC transmitter*. The *bit to byte converter* serves to pre-process the data bit originated from the PDH interface to the byte format. And, afterwards, all the data communications among the different system components are defined on a per byte basis. This is to facilitate the rather complex data flow as we shall see shortly.
The second state machine - *data packaging state machine* - focuses on processing the PDH data or the MAC payload data. In essence, it stores the payload data onto the external memory space i.e. *Data Block RAM* and also calculates the appropriate MAC header fields. This is the main logic controller of the whole design.
The third component, namely the *MAC transmitter*, interfaces with the MAC wrapper and, as implied by its name, transmits the complete MAC frame. Interestingly, the logic of the transmitter is the most complex among all three components. This is due to the fact that we no longer have a complete MAC frame ready before transmission; and, hence, the transmitter needs to poll the right frame byte at the right time from either *Data Block RAM* or *Header Block RAM*.
To further compicate the design, we will need to store certain dynamic header fields mentioned earlier into the *Data Block RAM* instead of the *Header Block RAM* because of the accessibility limitations. Note that the Block RAM is generally accessible from two ports for both writing and reading. The *Header Block RAM*, however, has to be interfaced with the micro-Blaze, which configures the header information and with the *MAC Transmitter*, which retrieves the header information for transmissions. As a result, the dynamic header fields that are calculated during the run time in

the *Data Block RAM* can only be stored in the *Data Block RAM* as this is the only block RAM that it has the access interface.
We conclude this section by summing up the functions of two block RAMs that we use. The *Data Block RAM* is used primarily to store the PDH data and certain dynamic header fields. It is divided equally into two parts - one for data transmssions and the other for simultaneous PDH data storage. The *Header Block RAM* is dedicated to store header information and is configurable from the micro-Blazer.

4.3 Data Storage and Data Flow

The MAC frame bytes are unfortunately scattered over many storage locations in the new design. It is therefore important to fully specify how the bytes are stored and communicated so that they can reach the right components at the right time.
We illustrate the complete Data Storage scheme in Fig 4.1. In the figure, we view the entire design purely from the data storage perspective. The PDH voice data bit gets converted in the *bit byte converter* and is, subseqently, called MAC Payload Data. It then goes through the *data processing state machine* so that the relevant counters can be incremented accordingly. It is then stored in the Data BLK RAM. Majority of the MAC header data are stored in the Heard BLK RAM, simply waiting to be polled from the *MAC transmitter* during the transmission.
If we look at the two BLK RAMs more closely, the Data BLK RAM actually stores two sets of MAC payload data per flow - one for current transmission and the other for simultaneous buffering of the PDH data. However, we only need to store one set of dynamic MAC header data since it only makes sense to fix their values the moment the transmission starts. We, thus, only store the set of fields that go with the MAC frame that is currently being transmitted. The Header BLK RAM is much simpler as it simply maintains one complete set of header fields for every flow.
Four dynamic header fields - Time Stamp, Packet Counter, Byte Counter and Packet Length - however are under the control of *data processing state machine* and are, therefore, stored in the Data BLK RAM. We should point out that the MAC transmitter does NOT construct a complete MAC frame before sending it out. Instead, it merely polls the appropriate byte one clock before transmitting it and store it locally. A more abstract data flow is shown in Fig 4.2.

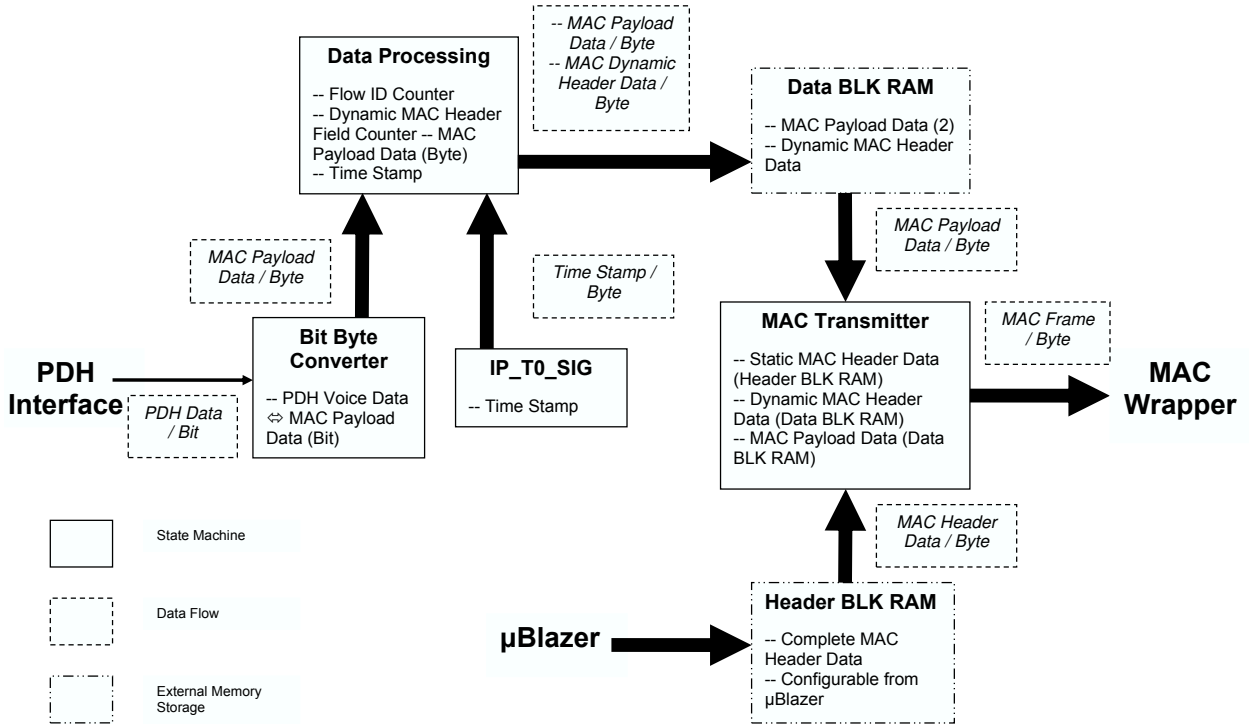

Figure 4.1: Modified Data Storage Scheme

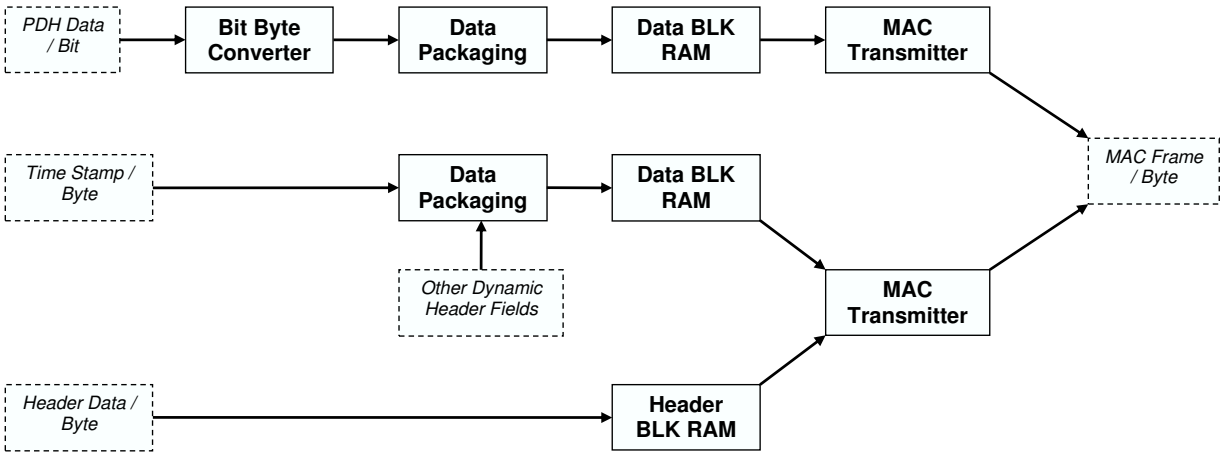

Figure 4.2: Data Flow Abstraction

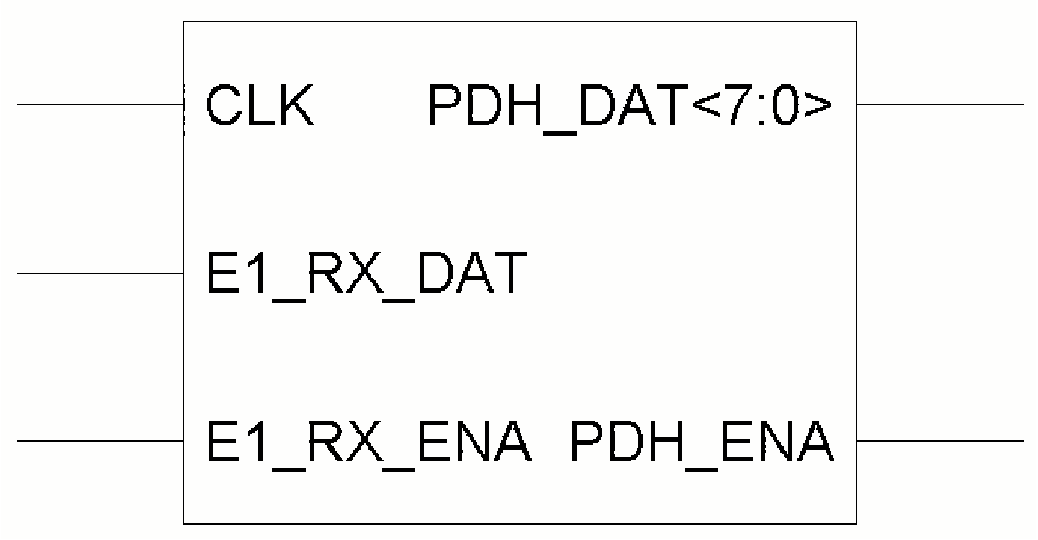

Figure 4.3: Bit to Byte Converter Interface

4.4 Modular System Design Description

After looking at the entire system design from a high level perspective, we now zoom into the detailed descritption of individual component of the system. For each of the component, we start with its interface description, followed by the corresponding state diagram and we end with the description of the design rational. The VHDL codes, however, are attached in the Appendix instead.

4.4.1 Bit to Byte Converter

The interface of the Bit to Bite Converter is shown in Fig 4.3. On the input side, E1_RX_DAT and E1_RX_ENA interfaces with the PDH and retrieve the data bit by bit. On the output side, PDH_DAT and PDH_ENA provide the Data Packaging state machine with the PDH data in bytes.
In Fig 4.4, we show the corresponding ASM chart of the converter. The bit to byte converting process is triggered by the E1_RX_ENA signal. The converter then accumulates 8 bits and raise the PDH_ENA signal high to notify the data packaging state machine that the byte is ready. The whole process is relatively easy and we only need one state for the converter.

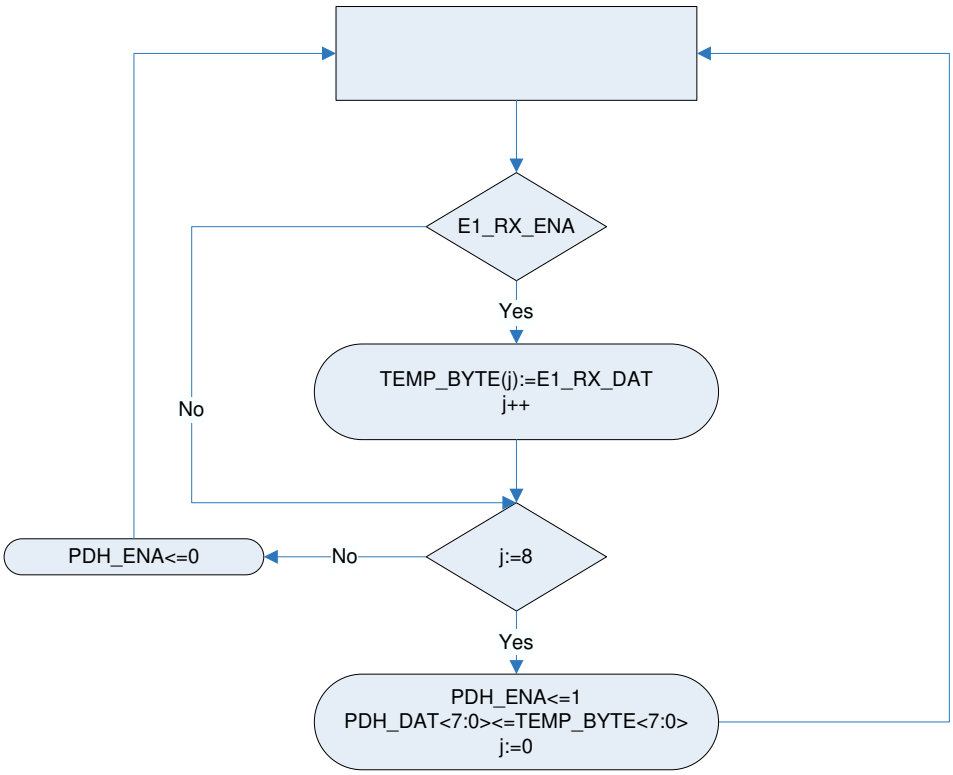

Figure 4.4: Bit to Byte Converter ASM Chart

4.4.2 Data Packaging State Machine

As mentioned earlier, the Data Packaging State Machine (Fig 4.5)is the central logic of the whole system. It mainly interfaces with the Data Block RAM through D_ADDRA, D_DIA, D_DIPA, D_ENA, D_RSTA and D_WEA. In additional, it also maintains all the dynamic header fields and Flow ID for the full implementation. Accordingly, it also interfaces with the synchronization clock signals namely IP_T0_DAT and IP_T0_SIG. Lastly, the state machine also temporally stores the PDH data through PDH_DAT and PDH_ENA.
The logic of the state machine is also the most complicated among the three, even though it has only ONE state (Fig 4.6). The complication comes from the requirements of simultaneously buffering the incoming PDH data as well as transmitting the internally maintained MAC field values during the start of the MAC transmission. Remember that we do not construct a complete MAC frame before transmission so we will have to make sure that the data byte is ready for transmission at the right time. This mainly explains the three consecutive yes or no logic judgement structure of the design.

4.4.3 MAC Transmitter

Lastly, we present the MAC Transmitter. Logically, it is the most complex among the three as it has to interface with four different entities i.e. the Header Block RAM (H_ADDRA, H_DIA, H_DIPA, H_ENA, H_RSTA and H_WEA), the Data Block RAM (D_ADDRA, D_DIA, D_DIPA, D_ENA, D_RSTA and D_WEA), the Synchronization Clock (IP_T0_DAT and IP_T0_SIG) and the MAC Transmission Wrapper (MAC_REQ, MAC_DAT, MAC_ENA, MAC_END, MAC_ERR, MAC_RST and MAC_START). We show the complete interface in Fig 4.7.
Although there are 13 states designed for MAC Transmitter, the logic behind is rather simple. As we can tell from the ASM Chart (Fig 4.8 and Fig 4.9), the majority of the states are of rather similar structure. They make sure the MAC poll the right data byta at the right time and right place so that the next byte to be transmitted is stored in the internal buffer one clock cycle before the actual transmission. The zig-zag arrangment of these states reflects the complexity we discussed earlier - the transmitter needs to poll the header fields from both the Header Block RAM and the Data Block RAM in an alternative manner. It is also noted that the MAC Transmitter often stays in the

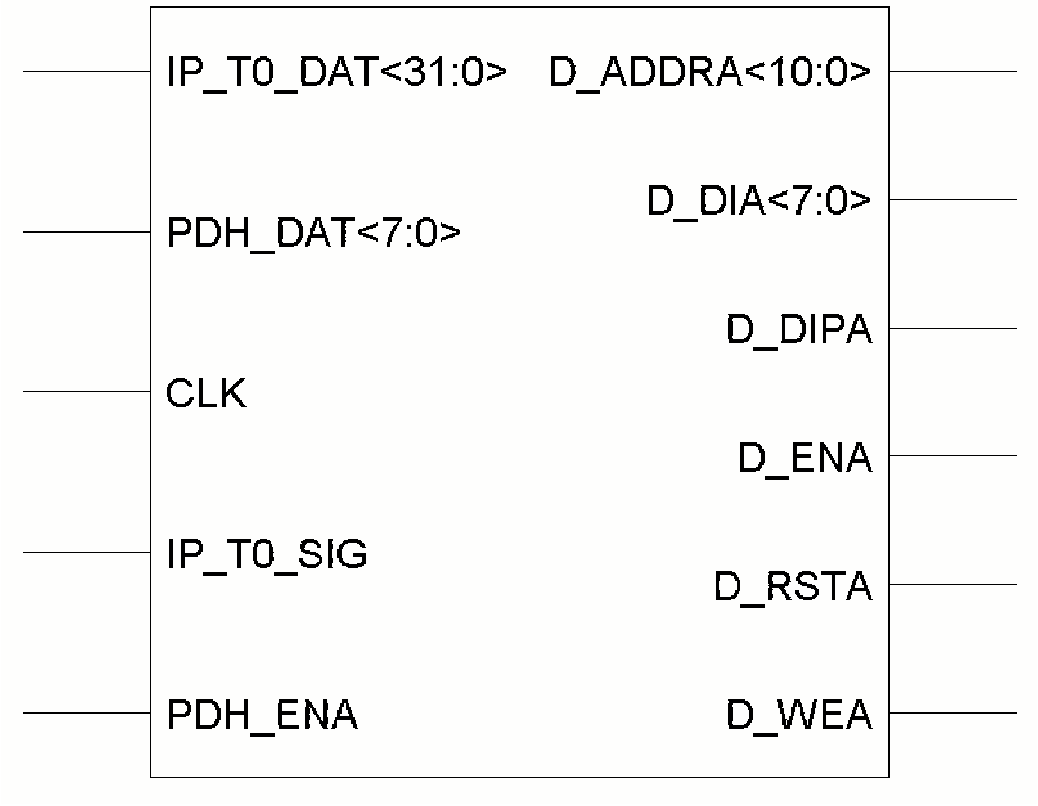

Figure 4.5: Data Packaging State Machine Interface

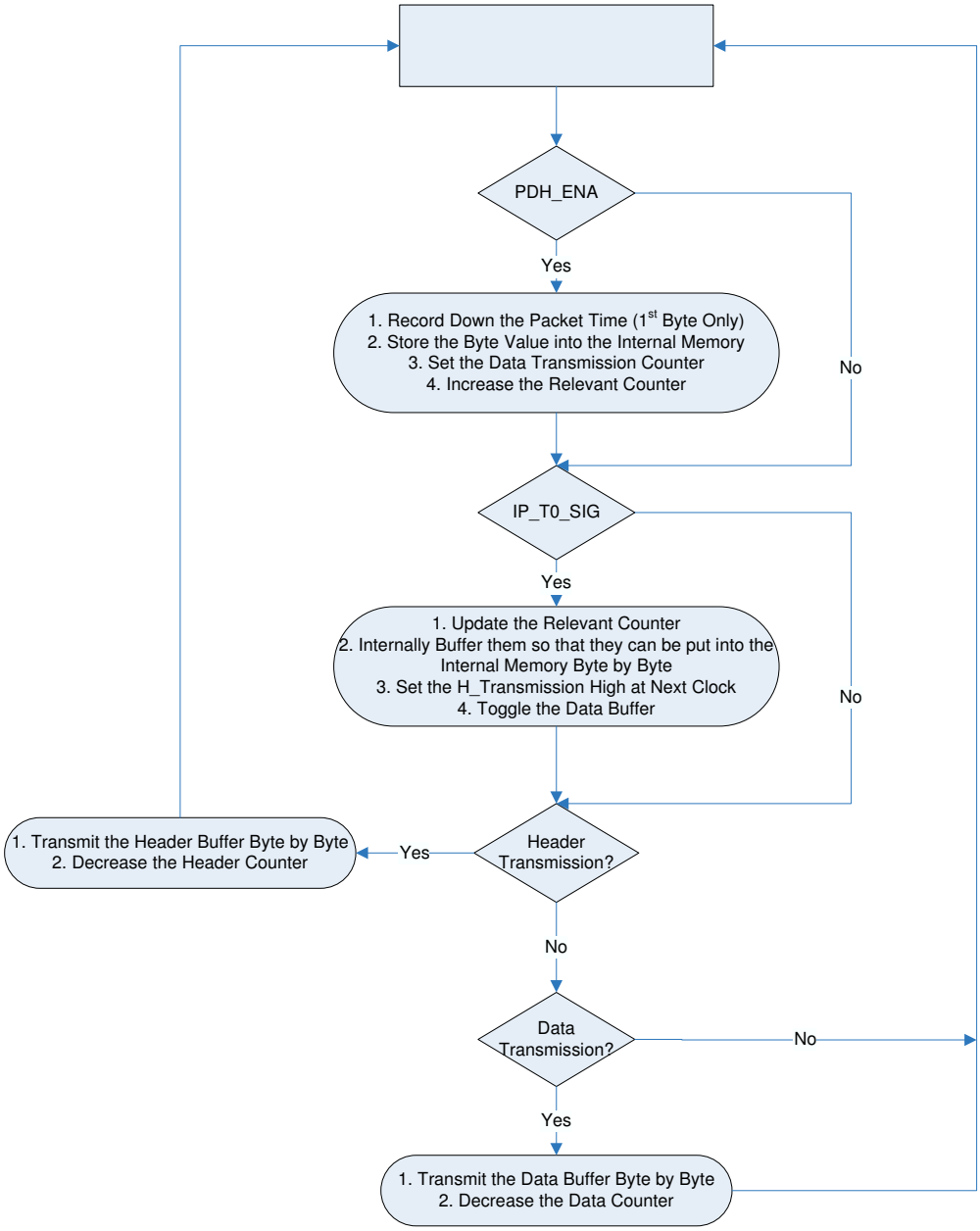

Figure 4.6: Data Packaging State Machine ASM Chart

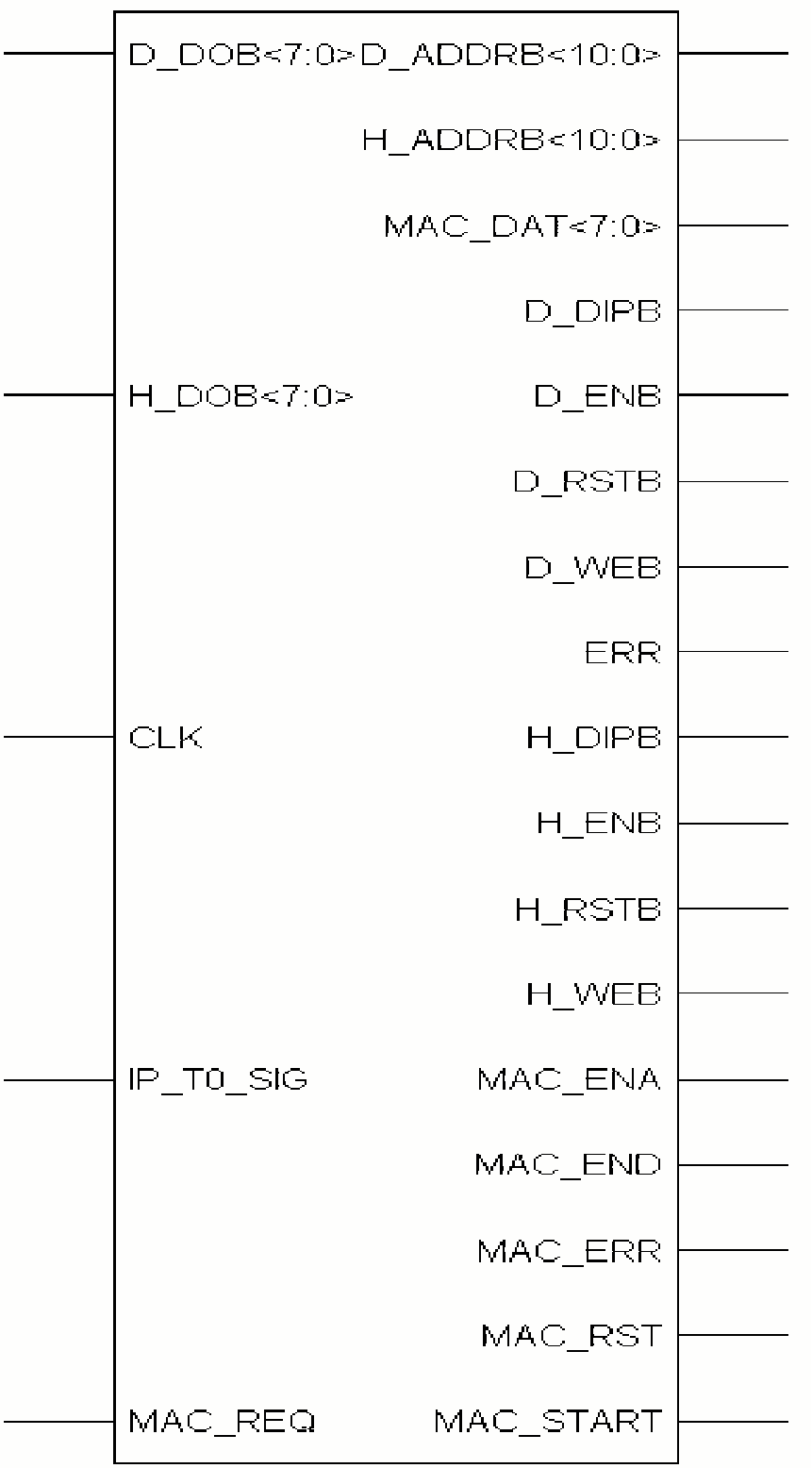

Figure 4.7: MAC Transmitter Interface

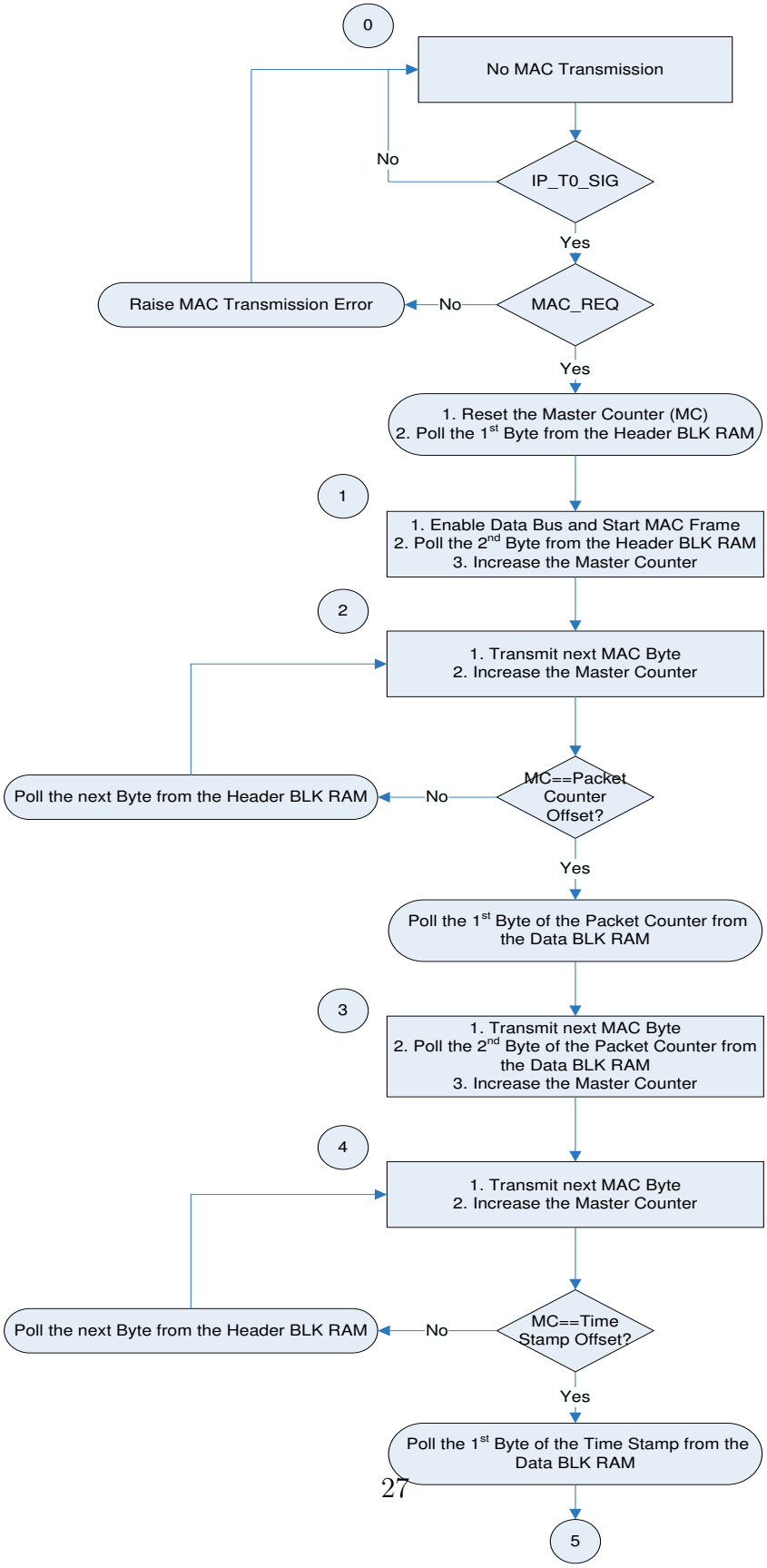

Figure 4.8: MAC Transmitter ASM Chart

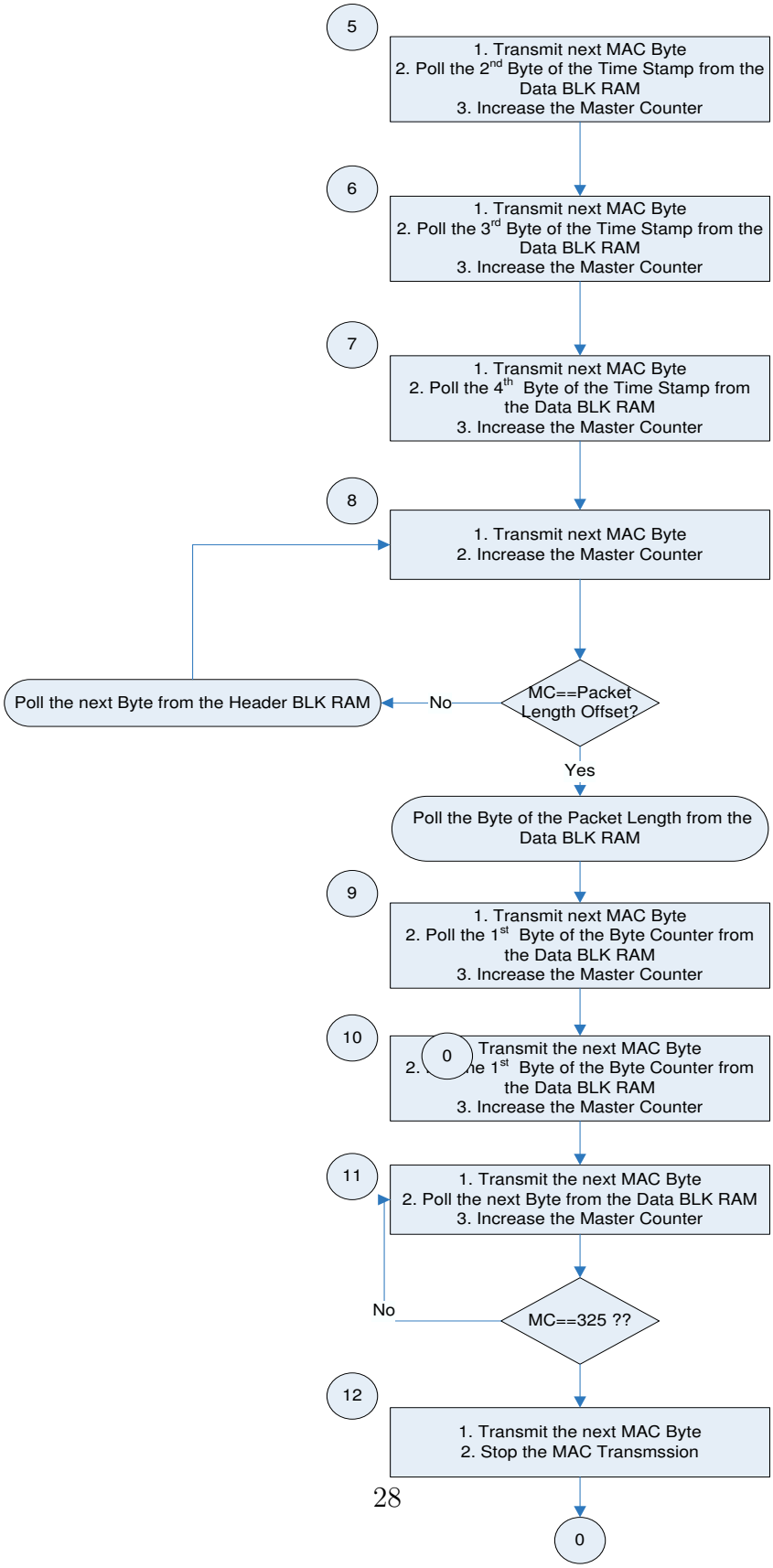

Figure 4.9: MAC Transmitter ASM Chart *(Continued)*

Chapter 5

Backward Path

After the extensive discussion of the VHDL design techniques and principles from the previous chapters, we present the complete design of the Backward Path of the CEALite implementation in this chapter.
The Backward Path is the counterpart of the Forward Path and it serves to process the frames sent from the Forward Path at the receiver end. Its basic function includes examining the header values, verify CRC check results, updating the microBlaze with proper header values and output the payload voice data.

5.1 Overview

The entire backward path consists of two state machines - Header Analyzer and Info Extractor, two Block RAMs - Config_BLK_RAM and Data_BLK_RAM, the IP_SIG clock as well as the micro-Blazer internal registers.
The Header Analyzer block interfaces with the MAC Wrappers, decides whether the incoming MAC frame is intended for the CEA, strips away the protocol headers and pass down the relevant data to the Info Extractor block. It also refreshes the value of the IPMAC address of the CEA as well as the UDP port-to-Flow_ID lookup table from the Config_BLK_RAM on a per frame basis.
The Info Extractor block takes all the necessary data from the Header Analyzer block including the Flow_ID and stores it with other header values into the Data_BLK_RAM. As soon as the data comes in, Info Extractor is responsible for taking the time stamp from the IP_SIG clock of the incoming frame too. It also has the direct access to an internal register of the micro-Blazer, where the current index pointer of the

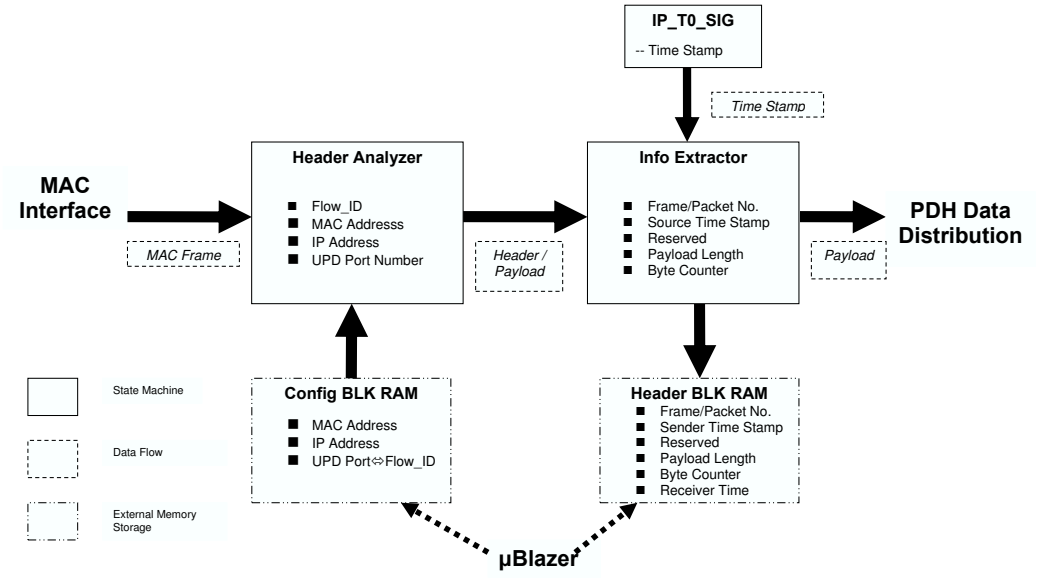

Figure 5.1: Schematic Overview of the Backward Path

Data_BLK_RAM is updated. Lastly, it has a pseudo-data output interface that outputs the payload data i.e. the telephony traffic. We output sufficient information so that the data output interface can be be connected directly with the block RAM.
We show the complete schematic of the backward path in Fig 5.1. The input and output interface of the backward path is shown in Fig 5.2

5.2 Header Analyzer

The Header Analyzer plays the role of admission control of the input MAC frames. It verifies both the IP address and the MAC address of the incoming frame against their preset counterparts of the CEA board. If both of the values are correct, it then retrieves the corresponding Flow_ID that the current frame belongs to through looking up the values in the mapping table. The frame will be simply discarded if any of the above three fields, namely IP address, MAC address as well as UDP port number, are invalid. Once the frame is admitted, the Flow_ID together with the header fields starting from the byte number 56 (i.e. frame/packet counter) all way till the end of the MAC frame (excluding the trailer) are forwarded to the Info Extractor for further processing. As soon as one frame is forwarded, it refreshes the IP address, MAC address as well as the mapping table from

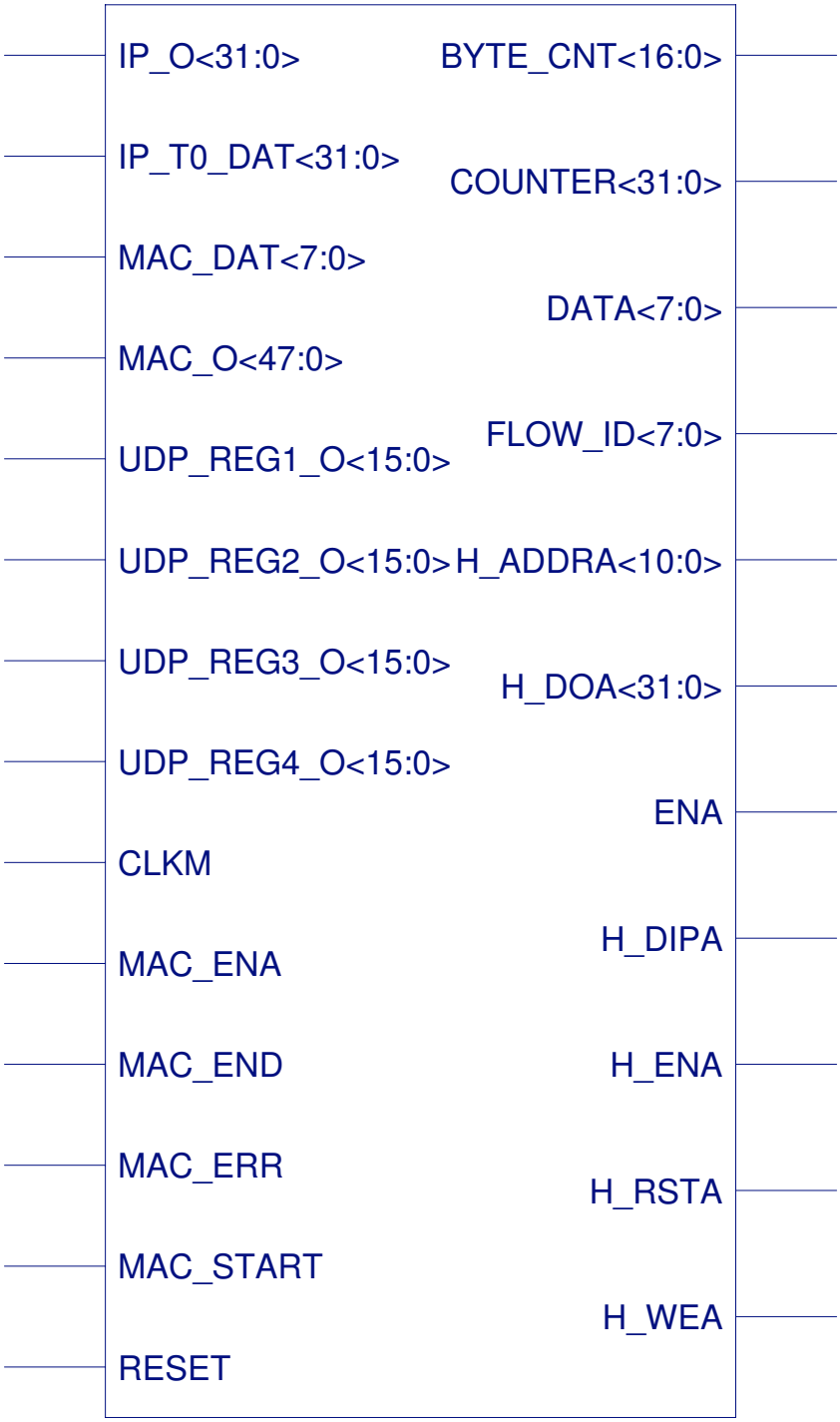

Figure 5.2: Interface of the Backward Path

the Config_BLK_RAM, which is configurable from the micro-blazer. The values are only refreshed once per frame since we don't expect the IP Add, MAC Add and the Port to Flow_ID binding change often.
The interface of the Header Analyzer and its flow chart are shown in the Fig 5.3 and Fig 5.4 respectively. In the actual implementation, it is also possible to replace the block RAM with the dedicated registers so that we can refresh the values in one clock cycle. Note that the ENA signal marks the start and end of the DATA frame and remains high for an effective frame.

5.3 Info Extractor

The Info Extractor takes the input data from the Header Analyzer. As soon as the Header Analyzer signals the incoming frame data is ready by raising the "ENA" signal, the Info Extractor stamps the receiving time from the IP_SIG clock. It then extracts the header values and stores them into the Data_BLK_RAM according to the following table. To facilitate the IO for both the Info Extractor and the micro-blaze, each field value, no matter how many number of bit they have, occupies one 32 bit memory location in the block RAM. Further, for the ease of numbering, each the field values of each MAC frame will take the chunk of 8*32 memory space, instead of 7*32. The expected Header block RAM should be of the size 2048*32; thus, the mirco-Blaze has a turn over time of 256 ms.
Since we will buffer multiple set of field values from the previous MAC frames, an index pointer has to be specified and made available to the micro-Blaze so that it can read off the newly modified entries from the block RAM. The index point always points to the latest updated header entry by the Info Extractor. The micro-Blaze also keeps an internal counter and it points to the entry of the block RAM that is last read by the micro-Blaze. Hence, the difference of the index pointer and the internal counter marks the chunk of memory locations to be read. This index number is communicated through a dedicated connection to an internal register of the micro-Blaze. The index number is on a per set basis.
Though not required in the CEAlite implementation, the interface to output the payload data as well as the relevant counter values are still implemented. We aim to output sufficient information so that a corresponding data distributor unit can be built to interface with the E1 connections. The complete interface of the Info Extractor and its flow chart is shown in the next two figures (Fig 5.5 and Fig 5.6.

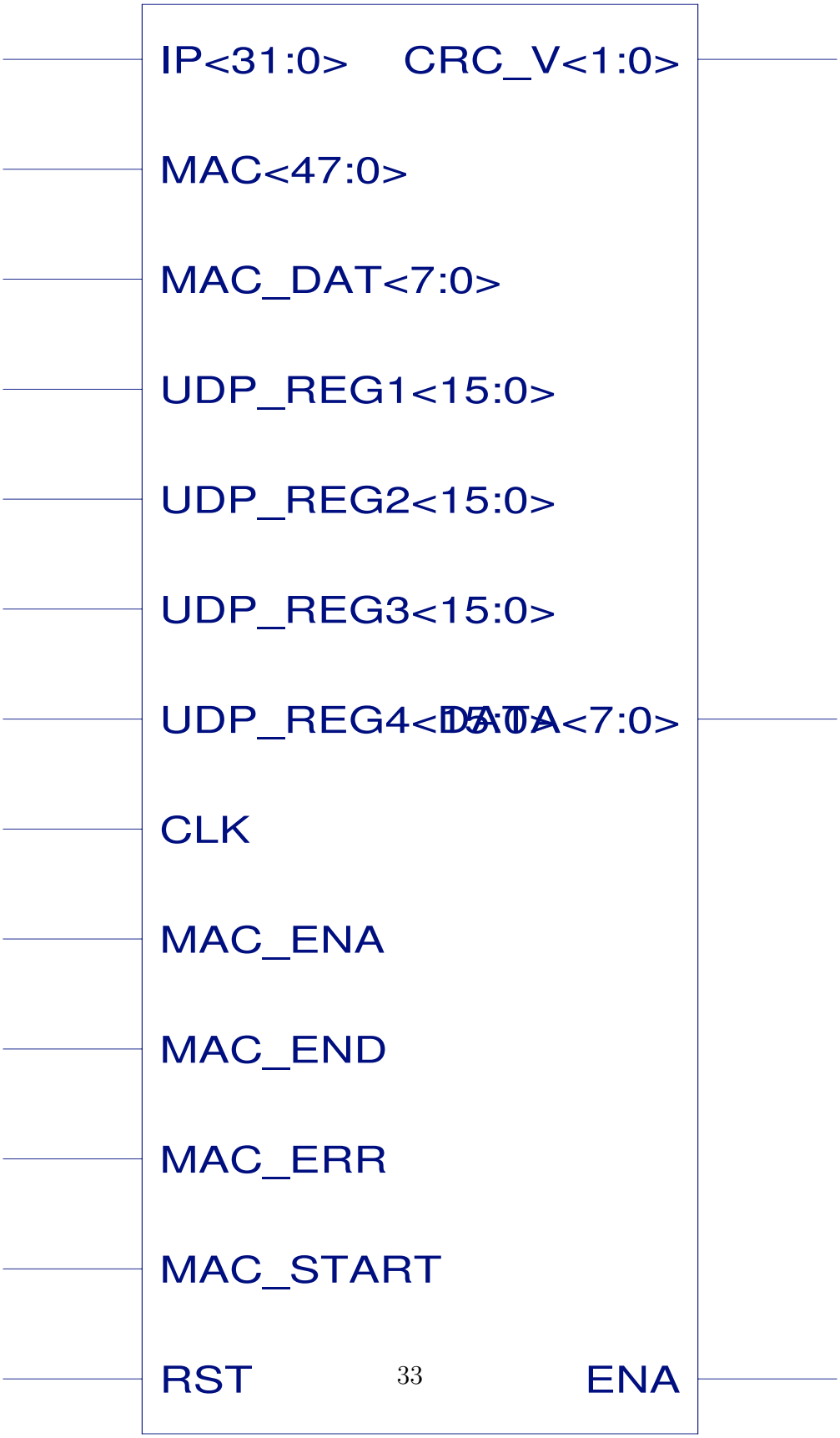

Figure 5.3: Flow Chart of the Header Analyzer

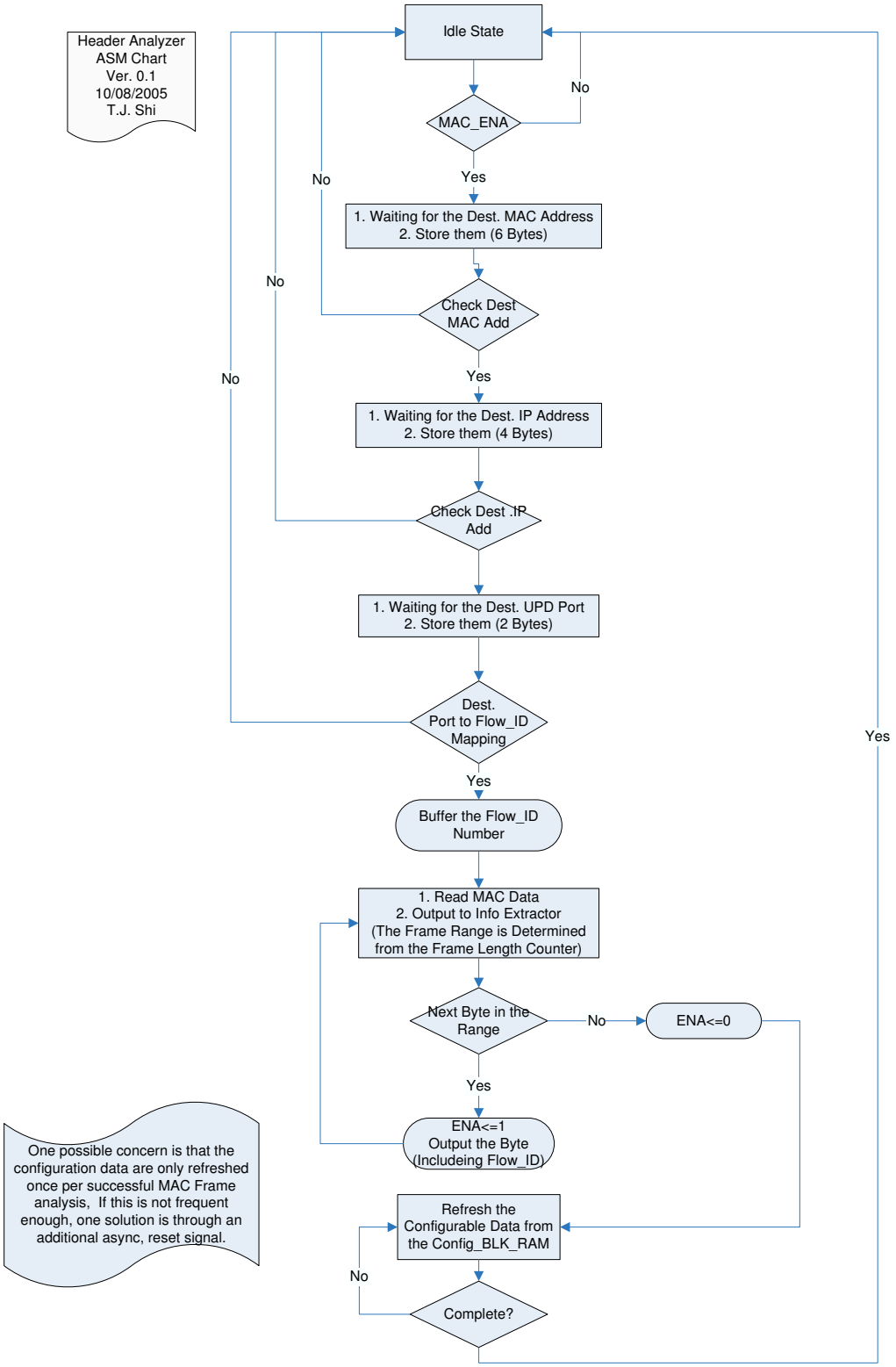

Figure 5.4: Interface of the Header Analyzer

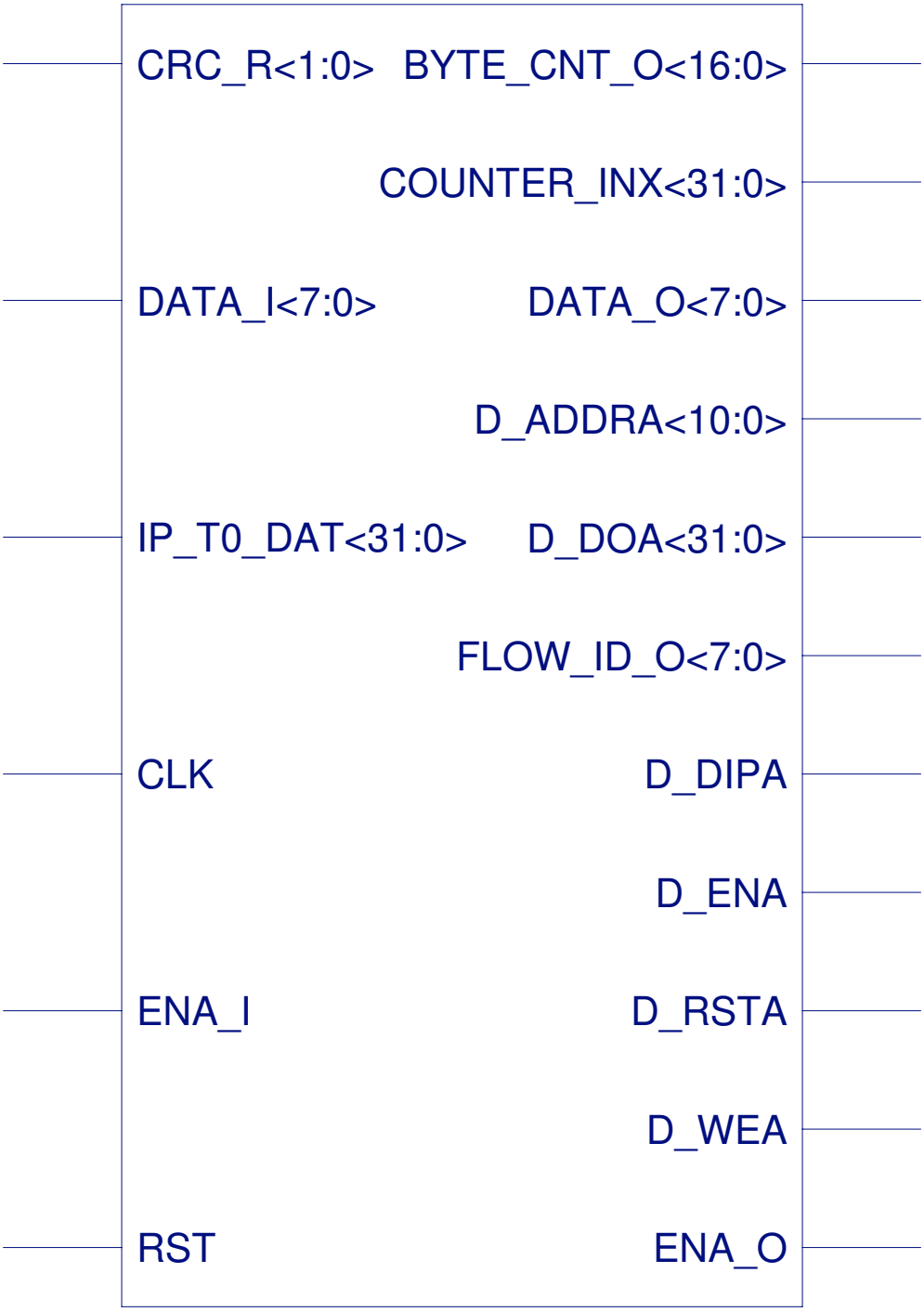

Figure 5.5: Flow Chart of the Info Extractor

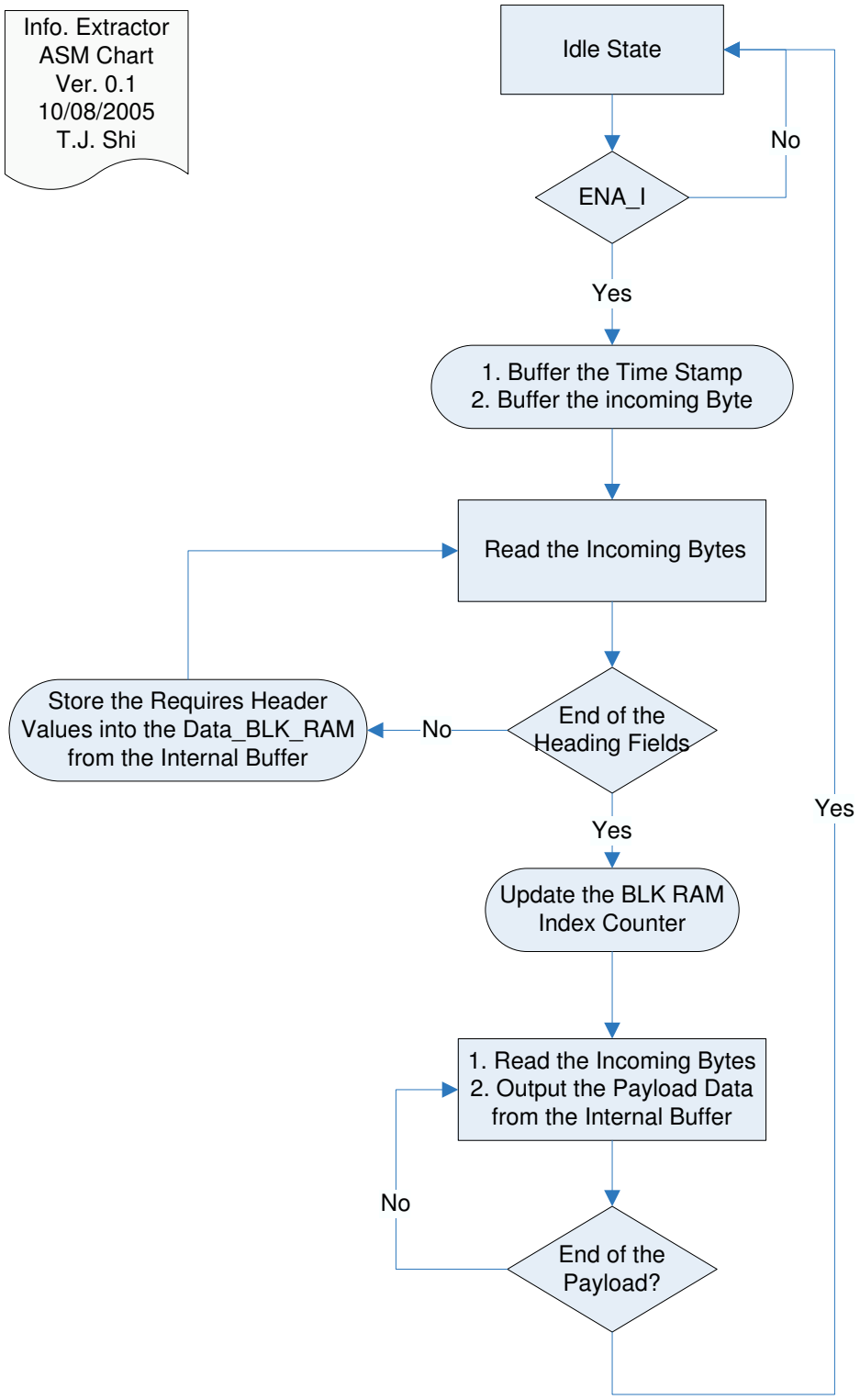

Figure 5.6: Interface of the Info Extractor

Chapter 6

From Software to Hardware

As we have discussed in the first chapter, the eventual goal of programming VHDL is to produce the hardware logic on the Xilinx FPGA chips so that they can be used as the actual hardware components. We describe briefly in this chapter the concepts of converting the software VHDL codings to actual hardware implementation.

6.1 The Outsiders' Process

In previous chapters, we mainly focus on the design principles of the CEALite. We now discuss the operational aspects of the VHDL hardware design. In Fig 6.1 , we show the complete design cycle.
Accordingly, our previous discussions correspond to the "Synthesize" step of the design, where we draft the VHDL source code and translate it into Netlist. Up to this step, we are still in the "software" phase of the design since the entire design is based on the computer software simulations. However, this is actually the core of the design as the behavior of the hardware is fixed in this stage. The rest of the steps are automated by Xilinx software and, thus, are merely mechanical.
In the "Implementation" step, we map the logic gates and their interconnections according to the appropriate family of FPGA chips. The operation includes implementing the CLBs through the look-up tables as well as interweaving the sets of CLBs by fusing their interconnections. Once the implementation is complete, this information is further translated into a string of '0's and '1's (bitstream) where '0's and '1' indicate the open or closed switches inside the FPGA. As a final step, the bitstream is downloaded to the FPGA chip. The fusible interconnections and electric switches are then "burned" in response to the binary bits in the stream. If

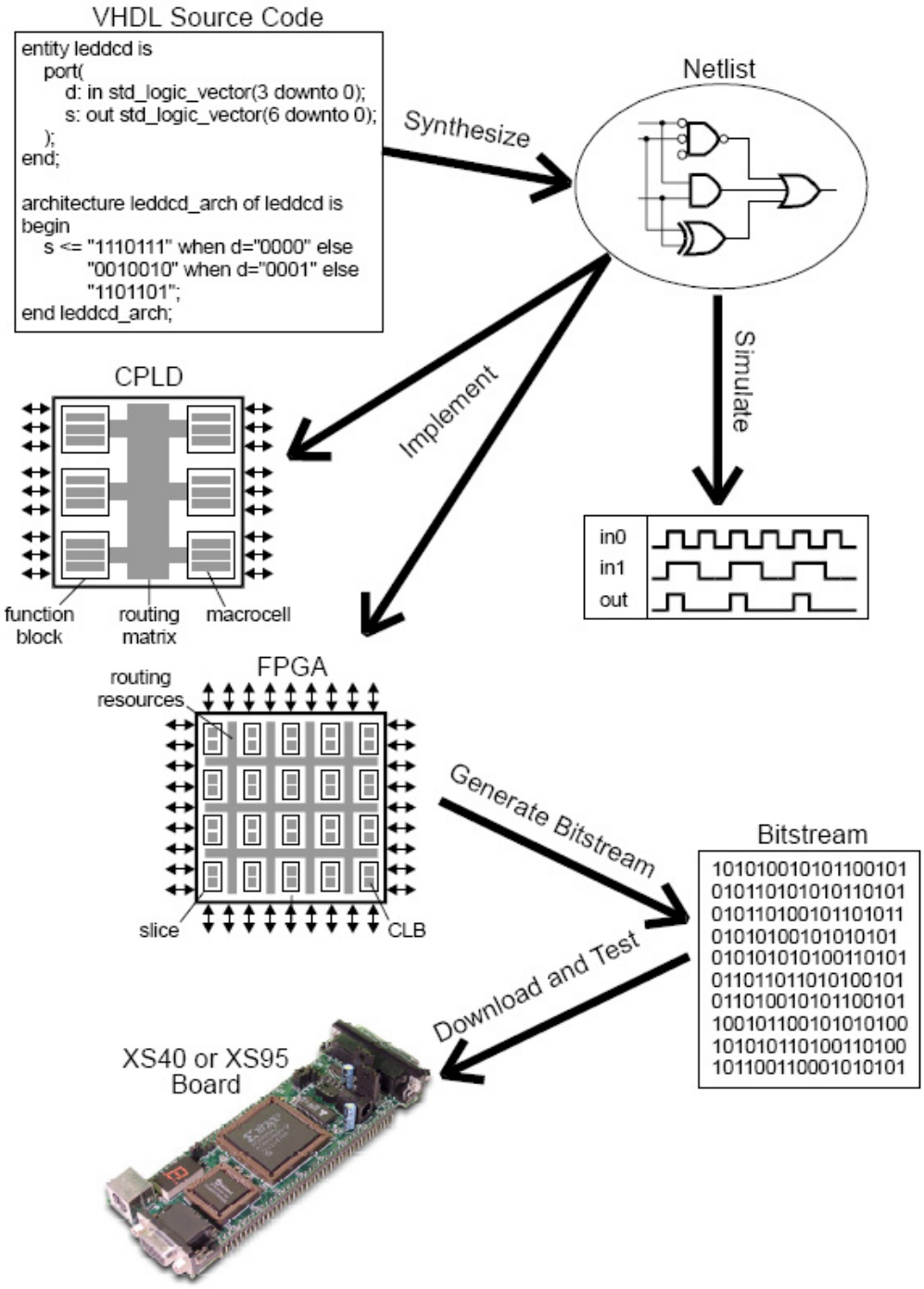

Figure 6.1: Complete VHDL Design Cycle

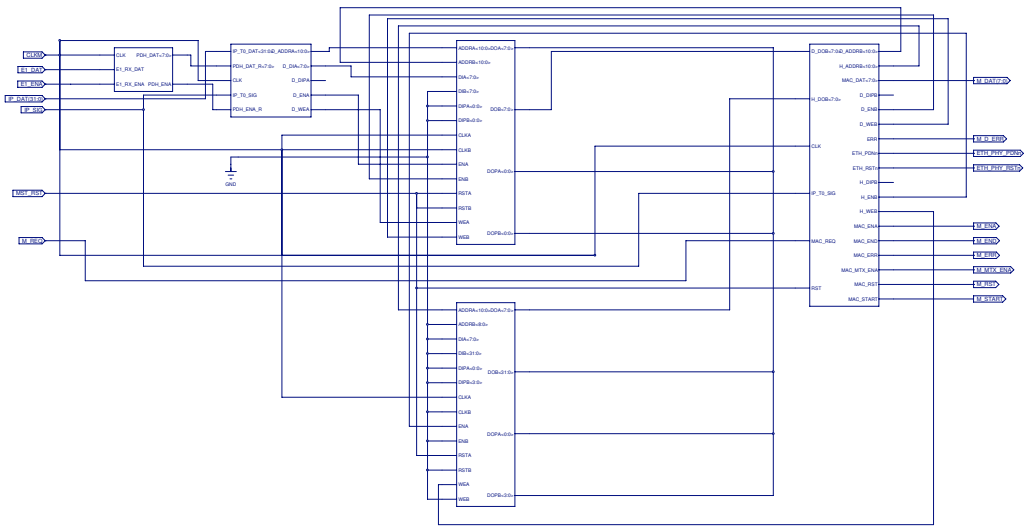

Figure 6.2: Block Interconnection of the Forward Path

one is lucky enough, the FPGA will now function properly with respect to the design.
The readers are advised to go through commercial instruction manuals for a step by step tutorial of complete process.

6.2 The Insiders' View

To provide more insights to the hardware implementation, we attempt to merge the gap between the state machine view of CEALite and its functional perspective as logic components. In the end, from hardware point of view, everything boils down to the logical gates.
We first show the interconnections of the different components of the forward path (fig. 6.2) and backward path (fig. 6.3). They are still represented as logical components of the state machines.
Now we should zoom into the more detailed logic gate perspective. Unfortunately, most of the state machines involve numerous logic gates and extremely complicated connections – they are just too much for analysis. We therefore choose one of the the simplest modules for demonstration. The following figure (fig. 6.4) shows logical construction of the the bit to byte module.
In order to see the real logic gates, we now further zoom into one of the data registers which store the bit values (fig. 6.4) . We observe that there are five input signals in the module with one NOT gate and one AND gate.

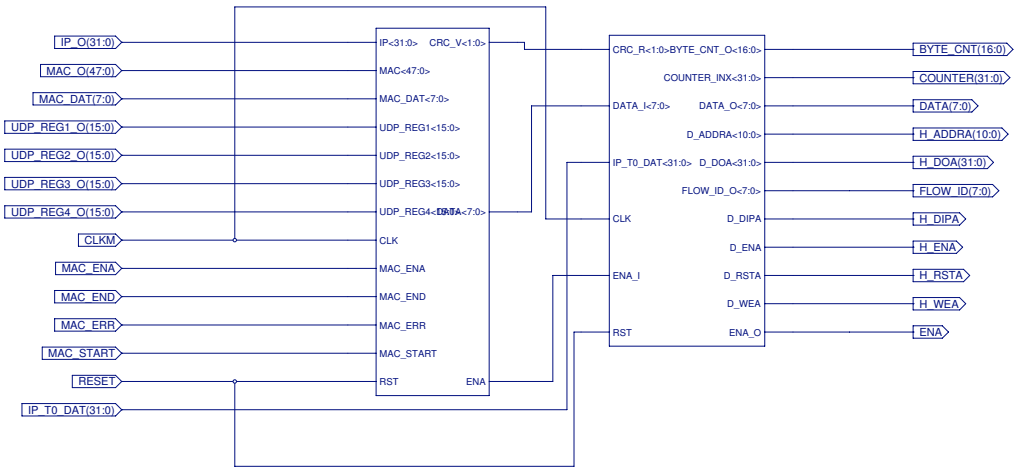

Figure 6.3: Block Interconnection of the Backward Path

It is easily seen that the whole register is enabled only when the enable signal is high. The output of an AND gate is high only if all of the inputs are high. This is consistent with the "ENA" signal designed earlier. The digital design engineers focus more on the state machine designs and pay minimal attention to the hardware translations.
However, due to the space constraint and large complexity, we are unable to demonstrate more translation details. They are normally an indispensable part of digital design and are no longer done manually. Indeed, this translation process can be fully defined and automated by computer software such as Xilinx Integrated Design Environment.

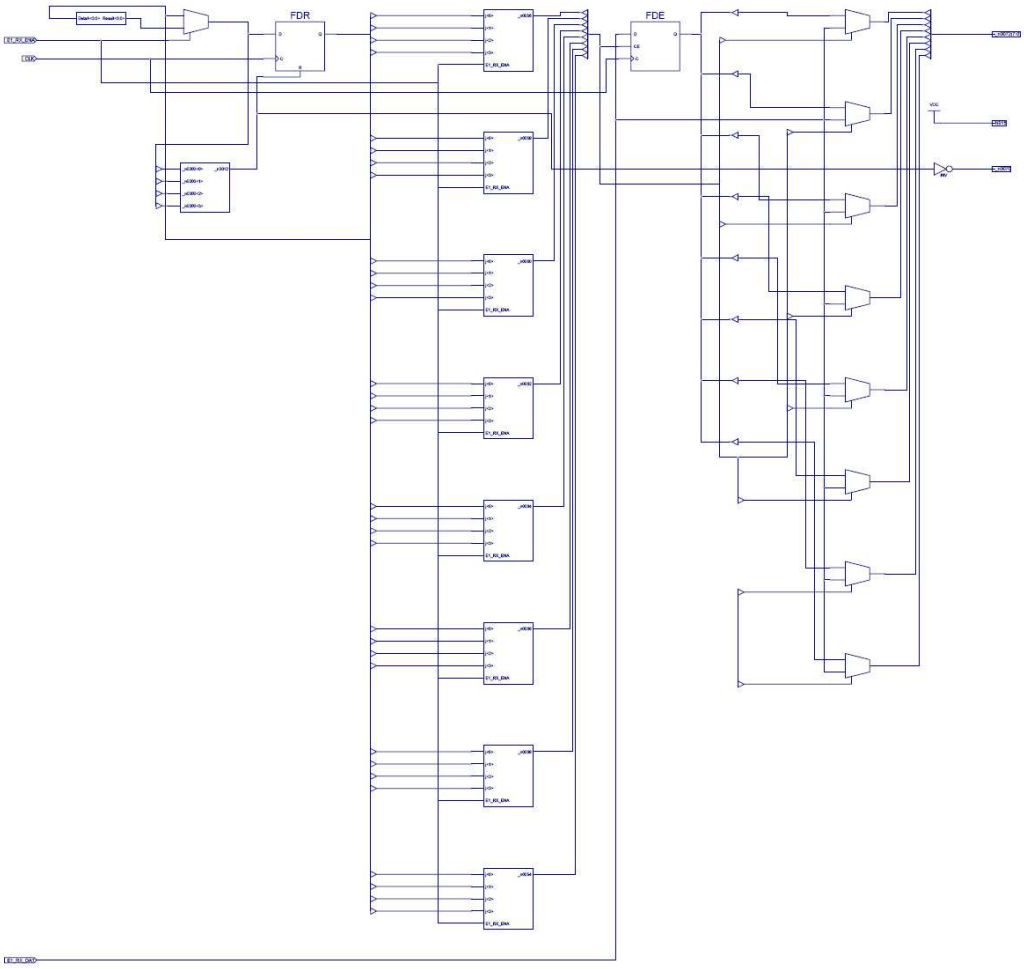

Figure 6.4: Logical View of Bit to Byte Module

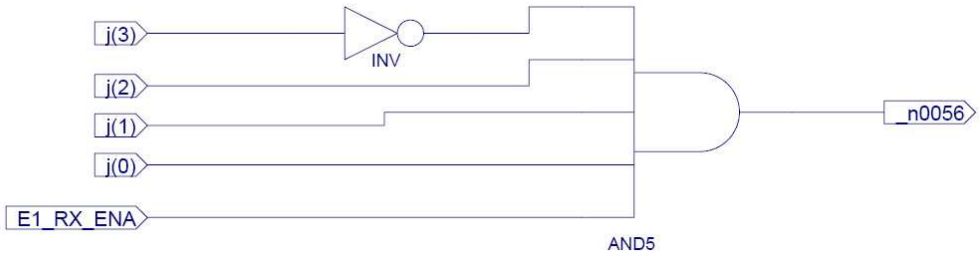

Figure 6.5: Logic Gate Interconnection of Register Component

Chapter 7

Conclusion and Further Work

In this project, we applied state machine based design concept to implement a test version of Circuit Emulation Adapter (CEA). To facilitate the design and integration, we further adopted the modular design approach to divide the state machines to several modules with a small increased interfacing cost. Throughout the design, we attempted to strike a fine balance between design complexity and logic clarity, interfacing cost and modularity, as well as storage efficiency and access convenience. The forward and backward paths are then translated into VHDL codings and, finally, are implemented onto the Xilinx FPGA devices. Both forward and backward paths are verified through time simulation and hardware testings. For further work, we need to integrate both paths with the Mircoblaze using VHDL. Then the entire implementation can be exported to Xilinx FPGA devices for hardware testings. This work will be jointly taken up by ETH, Zurich and Siemens Research, Switzerland.