A PROTOCOL DECODING ACCELERATOR (PDA)

By

CHING LEONG WAN

B. Eng. (High Distinction), Carleton University, Canada, 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF ELECTRICAL ENGINEERING)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

February 1990

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical Engineering

The University of British Columbia
Vancouver, Canada

Date Feb 27, 1990

Abstract

With the increasing need for distributed processing and computer networking, the demand for open systems interconnection (OSI) has also increased. In [Davis-88], Davis et al propose a new generation portable protocol tester that will be able to provide conformance testing for OSI protocol implementations. In this thesis report, a specialized programmable hardware module, called protocol decoding accelerator (PDA), is designed to be used as the PDU decoder engine being defined in the Davis architecture. PDU decoding is the process of parsing the PDU header fields into a data structure that can be more readily used by other processes. Decoding can be time consuming because there is a large variety of PDU fields and formats.

Conventional approach to PDU decoding is often implemented as software program designed for general purpose processor architecture. However, most general purpose processors do not handle PDU decoding efficiently. There are other VLSI protocol controllers, but they all have limited programmability and flexibility.

The PDA is developed based on a simple instruction set with dedicated hardware to optimize important functions. Using selected PDU types and decoding programs from OSI layer 2 to 4 protocols, the resulting PDA design shows a minimum of 16 times faster average execution time and about five times smaller program size when compared to a 68000 system.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I sincerely thank my supervisor, Dr. Mabo Ito, for his valuable suggestion, advice and guidance throughout the course of my research. I am also grateful to the insightful comments from the examining committee.

I extend my deep appreciation to the UBC/IDACOM project group for providing the research facility and financial support. Special thanks go to the following people from the UBC/IDACOM group for their constructive technical suggestion and assistance: Curtis Anderson, Issac Chan, Susan Chan, Bill Davis, David Hauck, Benard Lee, Viola Lee, Dennis Lo, Michael McLaren, Michael Sample, Helen See, Brian Smith, and David Wong.

I also wish to express my gratitude to Winnie Lam and Ashley Katz for their moral support and inspiration.

Finally, to my parents and brother I express my heartfelt appreciation for their constant support, encouragement, and patience.

Chapter 1. Introduction

With the increasing need for distributed processing and computer networking, the demand for open systems interconnection (OSI) has also increased. In an effort to encourage this capability, international bodies, such as International Organization for Standardization (ISO) and International Telegraph and Telephone Consultative Committee (CCITT), have been establishing communication protocol standards. In addition, ISO have also developed standardized test suites to enable protocol implementations to be tested for conformance to these standards. In [Davis-88], Davis et al propose a new generation portable protocol tester that will be able to provide conformance or detailed diagnostic testing functions. In the rest of this thesis report, this new architecture will be referred to as Davis' architecture. One of the elements in this architecture is the *PDU decoder engine*, which is responsible for parsing the Protocol Data Unit (PDU) into a data structure that can be more readily used by other processes. PDU decoding is one of the most fundamental steps in protocol implementation as information is often encoded onto a predefined message format for transmission, and decoding must be applied to retrieve the information for processing at the receiving end. Decoding can be complex and time consuming because there is a large variety of PDU fields and formats depending on the PDU type and the options in effect [Svobodova-89].

Most conventional protocol implementations are software programs designed for general purpose processor architecture. The PDU decoding is often embedded as part of the protocol processing function. This approach suffers several drawbacks. Firstly, general purpose processors do not handle PDU decoding efficiently. Most general purpose processors are designed to operate with byte (8 bits), word (16 bits), and long word (32

bits) values; whereas PDU decoding involves bit manipulation heavily. Secondly, the PDUs are usually stored in a link list data structure, and there is large overhead for general purpose processor to access the element of such a link list for decoding. Thirdly, the decoding is integrated into the rest of protocol processing, making it difficult to have a streamlined, modular design. Krishnakumar et al suggest a VLSI implementation with a different functional partition [Krishnakumar-87]. They propose that the core protocol processing can be handled by a major unit, and the low level functions, such as PDU parsing can be done by satellite units. In their design, the satellite unit being used for PDU parsing is called *Message Parser.* The Message Parser only performs syntactic analysis on a PDU, and its design is based on the context free nature of the PDU format. However, a lot of protocols require the decoding program to keep track of a few PDU parameters and do not exhibit a clear context free characteristic. Furthermore, their approach is to generate VLSI layouts from formal specification by using parameterized hardware architecture. Thus, a new VLSI layout is needed to implement each different protocol. There are other specialized VLSI protocol controllers as described in the survey done by Krishnakumar et al [Krishnakumar-89]. Most of these protocol controllers are designed for specific protocols, and they have very limited programmability and flexibility to handle a wide range of protocols and system environment. In particular, they do not fit into Davis' architecture.

Currently, a project [Chan-89, Anderson-89] is underway to implement Davis' architecture. The *PDU decoder engine* is implemented as a software procedure for a general purpose processor platform, but the architecture partition allows the decoder to be implemented by VLSI hardware design as well. It is excepted that a specialized VLSI implementation can provide significant improvement in PDU decoding.

In this thesis report, a specialized programmable hardware module, Protocol Decoding Accelerator (PDA), is proposed to be used as the *PDU decoder engine* defined in Davis' architecture. The PDA is designed to handle PDU decoding for OSI protocols from Layer 2 to 4 [CCITT-88]. The VLSI Technology's VGT200 series gate array technology [VLSI-88a] is chosen as the hardware design environment. PDU decoding involves parsing the PDU into a data structure that can be more readily used by other processes. PDU decoding can be broken down into PDU field extraction, PDU access, and result storage functions. The PDU field extraction contains functions, such as bit manipulation, arithmetic operation, data skipping, and remaining length calculation. The PDU access deals with reading the PDU data bytes that are saved in a link list structure, and the result storage is concerned with putting data and pointer values into memory. To support an efficient implementation, the PDA is designed by using a small and simple instruction set with hardware assisted functions. Other salient features of the PDA include: separate program and system memory interfaces, automatic PDU access algorithm, three level FIFO for PDU access, and one level buffer for writing results. With this high degree of optimization, the PDA is expected to be superior to the conventional approach of using software decoding programs on a general purpose microprocessor architecture. When compared to a Motorola 68000 system using equivalent decoding programs for CCITT X.25 and X.224 protocols, it has been found that the average execution speed of the PDA is at least 16 times faster than the 68000 system. In addition, it has also been found that the PDA program size is about five times smaller than the 68000 equivalent.

The rest of this thesis report is organized in the following manner. Chapter 2 describes the OSI protocol PDU format and Davis' architecture system environment in order to identify the design requirement imposed on the PDA. Chapter 3 explains the architecture

of PDA and its instruction set. Chapter 4 evaluates the design and explores the performance of PDA. Chapter 5 provides concluding remarks. Following the bibliography, there are five appendices. Appendix A provides a list of acronyms being used throughout this thesis report. Appendix B shows the PDU formats for the protocols being discussed. Appendix C describes the PDA Interface Signals. Appendix D and E give a sample program for decoding X.25 link layer in PDA and 68000 code respectively. Finally, Appendix F contains the schematics of the PDA design.

Chapter 2. Design Requirement

The PDA is designed to handle PDU decoding for OSI protocol from layer 2 to 4 in Davis' architecture. This chapter first describes the PDU format and the associated decoding algorithm. Then the Davis' architecture is presented to address the system environment issue. In the PDU format discussion, the following Layer 2 to 4 protocols are chosen as they are widely accepted OSI standards: CCITT X.25 link layer [CCITT-84a], CCITT X.25 packet layer [CCITT-84a], and CCITT X.224 transport layer [CCITT-84b] protocols (A description of the PDU format is given in Appendix B).

2.1. Protocol Data Unit (PDU) Format

The International Organization for Standardization (ISO) has defined a seven layer reference model for open systems interconnection (OSI) [CCITT-88]. Each of these layers is defined by a service definition and a protocol specification. The service definition specifies the available service primitives for adjacent layers' activity; whereas the protocol specification defines the functions between peer entities. The user data passes between adjacent layers as part of a service primitive is known as a *service data unit* (SDU). The control message that conveys between peer entities is known as *protocol data unit* (PDU). Figure 1 summarizes the relationship of this standard terminology. For an (N+1) entity to send an (N+1) PDU to its peer, it issues an (N) service primitive which includes the (N+1) PDU as the (N) SDU. Then the (N) entity inserts an (N) protocol control information (PCI) with the (N) SDU to form the (N) PDU. PCI contains the PDU identifier and a series of

parameters as defined in the protocol specification. Similarly, the (N) entity issues an (N-1) service primitive to request (N-1) service, and so on until the physical layer is reached. Figure 2 depicts this PDU formation. In the receiving end, when an (N) entity received an (N) PDU, it decomposes the (N) PCI and acts according to the parameters in the PCI as defined by the protocol. This decomposition is referred to as PDU decoding. To define the term formally, decode is the process of parsing the PCI parameters into a data structure that can be more readily used by other processes. Since the PCI is often put at the beginning of a PDU, it is also commonly referred to as the *header*. In the rest of this thesis report, PCI and header are terms being used interchangeably.

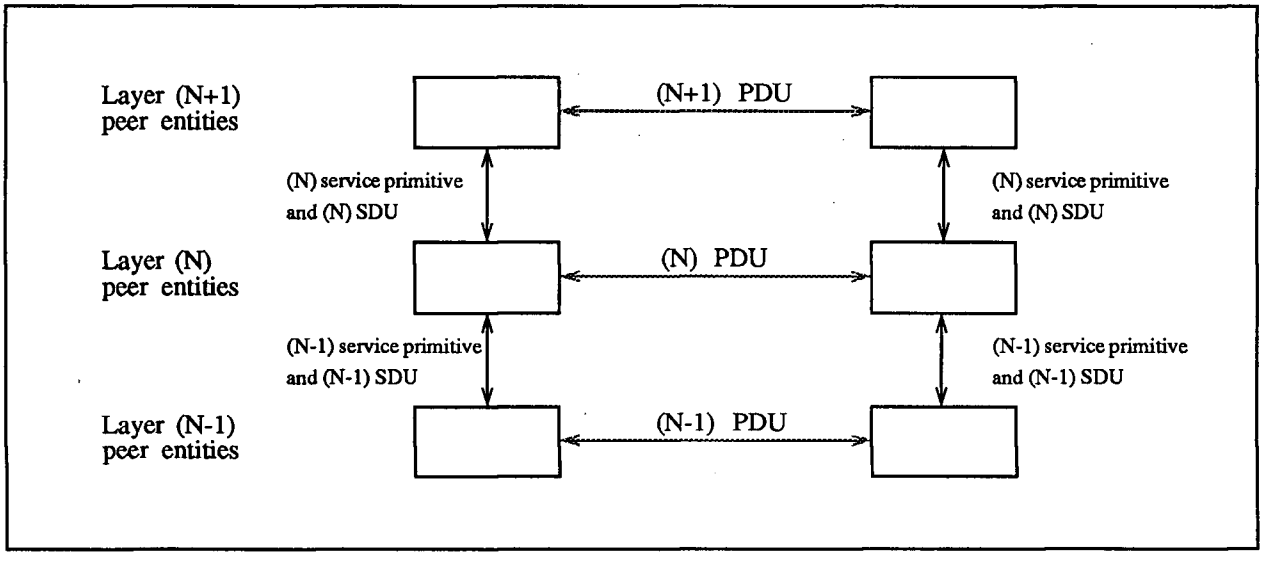


Figure 1. OSI Terminology for Peer Entity Cooperation

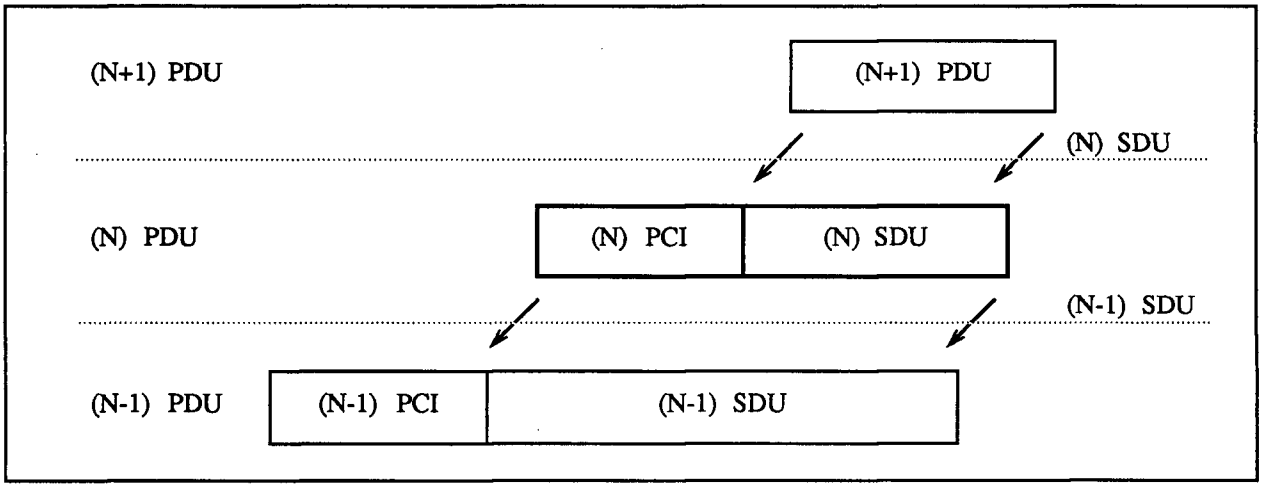


Figure 2. (N) PDU Composition

The PDU format for individual protocol layer is presented next.

The X.25 link layer uses a very simple PDU format. All parameters in the header have fixed length and at fixed position. In this case, decoding simply involves going through the PDU starting at the beginning and extracting the parameters accordingly. The parameters has various size ranging from one to sixteen bits, so bit extraction and justification are needed. One additional parameter that is useful to the protocol processing is the length of the user data in the *information frame*. This is not provided in the header and the decoder has to derive it explicitly. The X.25 link layer also provides the service of either basic or extended mode of sequence numbering, which have slightly different header format. The selection of this service can be passed to the decoder when it is invoked.

The header fields in X.25 packets have both fixed and variable length fields. The fixed length fields may have size that is not byte bounded, so bit manipulation is again needed. Most of the variable length fields have an associated length field to specify its size and the maximum size is 128 bytes. In an efficient implementation, these fields can be represented by its pointer and length. Then the decoder can skip the rest of the field and proceed decoding of the next field. For other variable length fields, such as *call user data* in the *call request packet*, they are always the very last fields in the PDU so that the length can be deduced from the end of the PDU. The position of header fields in X.25 packet layer are not fixed due to the existence of variable length fields. Hence, the decoder must provide other mechanism, such as absolute pointer, to allow these fields to be accessed directly. Similar to the X.25 link layer, the X.25 packet layer has different service options that results in variation of header format. These options are needed to pass to the decoder.

The X.224 transport layer PDU header contains three parts. Each header has a one byte *Length Indicator* (LI), a *fixed part*, and a *variable part*. The LI gives the total header length, which is up to a maximum of 254 bytes. The fixed part contains frequently occurring parameters. All of these parameters have fixed length and position. A few of the fields in the fixed part have non byte bounded size, hence bit manipulation is needed. All parameters in the *variable part* use the same structure for encoding. This structure is consisted of a *parameter code*, a *parameter length indication*, and the *parameter value* field. The parameter code is an eight bit value that identifies the parameter type. The parameter length, which is also an eight bit value, specifies the length of the following parameter value field. To decode the variable part, it is required to keep track of the remaining header length as specified by LI and the current parameter length to verify any possible discrepancy. The X.224 transport protocol supports different options and services that affect some of the PDU header format.

These include the class of service, the use of normal or extended mode, and the selection of checksum option in class 4. These options and services are connection dependent, so it may require two pass decoding. The first pass is to decode the PDU partially so that the protocol processing unit can identify the PDU type and the connection reference. The value of these options and services can then be determined and fed to the decoder for the second pass, which will continue to decode the rest of the PDU accordingly.

It is important to note that in all cases, it is only necessary to go through the PDU from the beginning to the end just once during the decoding. There is no need to refer back to any previous input data. This sequential access characteristic makes it possible to use a data prefetch algorithm as will be explained in the later chapter.

Another aspect of decoding is error handling. It is desirable to check the validity of the header fields as early as possible to avoid wasting processing time. A lot of state independent errors can easily be detected. In contrast, errors such as possible range of parameter value or presence of certain optional fields are impossible to determine without the protocol state information. As a result, state dependent errors are left to be resolved by the protocol processing function. When an error is detected, it is sufficient just to indicate the error type in most cases. However, it may also require to specify the location in which the error occurs as in the X.224 transport protocol.

2.2. System Environment

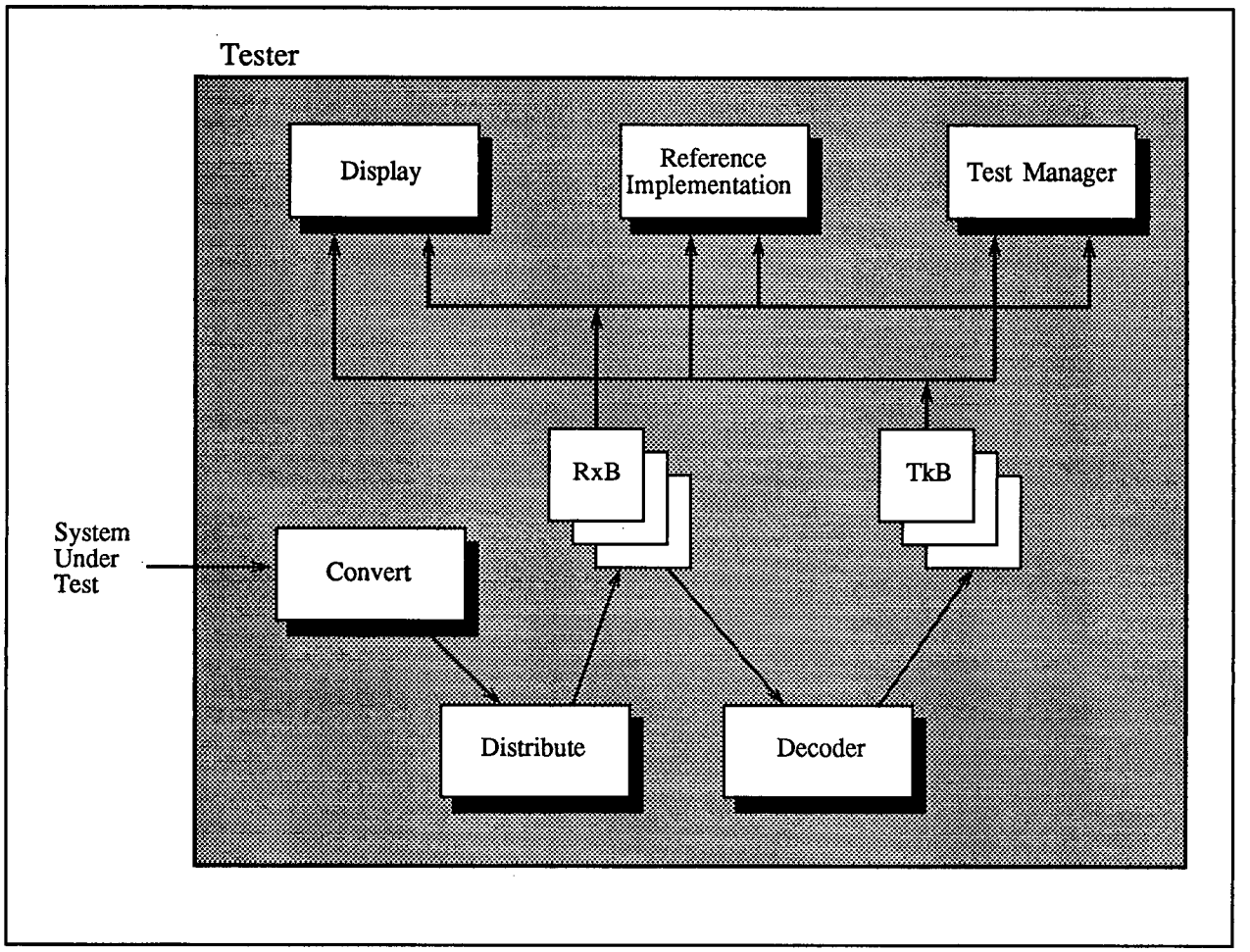


Figure 3. Partial Data Flow for Davis' Architecture

A partial data path of Davis' architecture is shown in Figure 3. The received data from the *System Under Test* is stored into the *Receive Buffer* (RxB) through the *Convert* and *Distribute* processes. Then the *Decoder* can be invoked to decode the protocol header and save the resulting data structure into the *Token Buffer* (TkB). Other processes, such as *Display*, *Reference Implementation*, and *Test Manager*, can access both RxB and TkB to carry out other tester functions.

The RxB is organized as a linked list of 2^n byte fixed size buffers with n being configurable from 5 to 8 at set up. Figure 4 shows the RxB linked list element format. The first four bytes in the element contains the pointer to the next element. The next two bytes are *Dstart* and *Dend*, which are the offsets from the beginning of the element to the first data byte and last data byte respectively. The sixth byte contains the *Use Count* and *Lock*, which are used for buffer management. The seventh byte is the *Flag*, which indicates special characteristic of the element. In particular, when bit 0 is set, the element contains the last data byte of a PDU. Finally, the data portion follows. Since *Dstart* and *Dend* are used to locate the data bytes of a PDU, it is possible to prefix or append other information, such as timestamp in the data portion. If the PDU spans over several RxB elements, the *Next Buffer Pointer* is used to jump to the following element. Only forward link is provided as there is no protocol function requires accessing the PDU in a backward manner.

TkB is a contiguous block of memory up to 256 bytes, and can be mapped to any data structure for storing the result of decoding. For illustration, an example for X.25 link layer *information frame* is shown in Figure 5. Figure 5(a) gives the PDU format and Figure 5(b) shows the data structure defined in C language for storing the decoder result. Finally, Figure 5(c) depicts the TkB mapping to the data structure. In the Example, the *type* and *error*

tokens indicate the frame type and the result of the decoding respectively. The *addr*, *p_bit*, *nr* and *ns* are fields being extracted and right justified from the PDU. Lastly, the *info_length* and *info_pointer* tokens are used to reference the information field. The *info_length* is the length of the information field in the received PDU and is calculated by the decoder. The *info_pointer* is the address of the beginning of the information field in the RxB. Sensible TkB mapping is necessary due to the limited size of the buffer. For instance, the information field mentioned above may be over hundred bytes in length. For this kind of field, it should be represented by its length and its pointer to the RxB, which only occupied a total of 6 bytes in the TkB. This way, a small TkB can be used for mapping to complicate PDU format. In addition, overhead in data copying can also be reduced. Decoding programs for data link layer to transport layer have been written and shown that a 256 byte limit on TkB is adequate. Multiple TkB can be used if necessary, but the decoding algorithm becomes very complex.

Access to the RxB and TkB is expected to be very frequent as they are being shared by many modules. Especially in the case of RxB, saving data into RxB must take the highest priority in order to keep up with the real time input data stream. Consequently, the access time to these buffers by the decoder is excepted to be long and variable. The design of the decoder must take this into account to reduce performance degradation due to poor system memory response time.

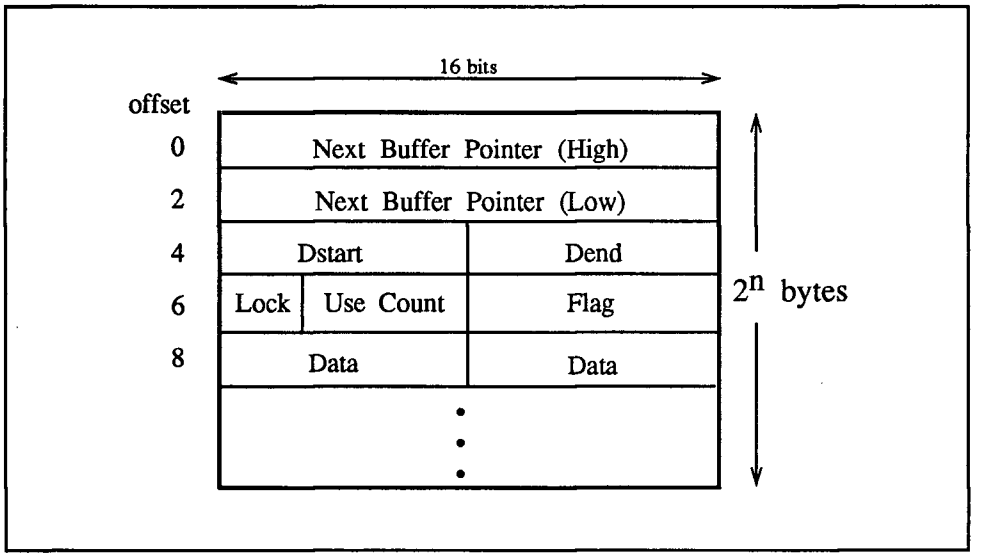


Figure 4. Receive Buffer Element Format

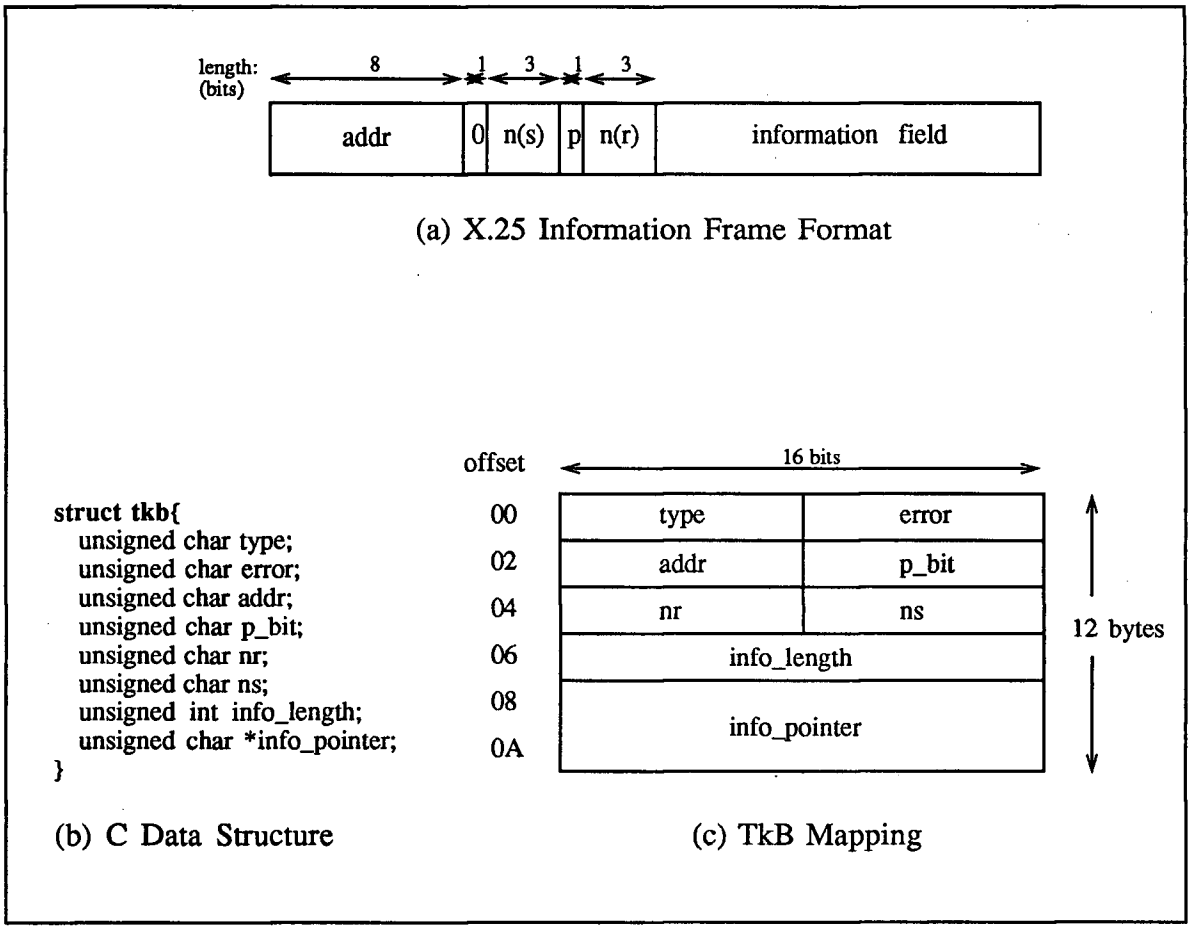


Figure 5. TkB Example for the X.25 Information Frame

Chapter 3. PDA Architecture

This chapter explains the PDA architecture and its instruction set. An overview is presented first, followed by description of the components in PDA. The instruction set is then discussed. Finally, an implementation and cost estimate is given.

3.1. Architecture Overview

The PDA is designed to interface with a custom arbiter, which in turn communicates with the main processor and system memory as shown in Figure 6. Figure 7 depicts the PDA interface signals. (A detail description of these signals is listed in Appendix C.) The PDA has a separate data memory and program memory interfaces to increase performance. The data memory is used for PDA configuration, RxB and TkB access, and loading PDA programs from system memory. The program memory contains PDU decoding program for various protocols. Decoding programs are small in general, so high speed static RAM can be used for the program memory. Lastly, the system interface is used for providing handshake signals to set and display the operating state of PDA.

The PDA supports three operation modes: *configuration, program execution, and program loading*. In configuration mode, all PDA configuration registers can be accessed through the data memory interface. This allows PDA options and decoding parameters to be set up. In program execution mode, the PDA executes the decoding program and signals upon completion. In this mode, the data memory interface is used to access the RxB and

TkB; while the program memory interface is used to read the decoding program. Finally, in the program loading mode, the PDA permits programs to be downloaded from data memory to program memory.

Figure 8 shows the functional block diagram of the PDA. It consists of three basic blocks: Data Memory Interface (DMI), Program Memory Interface (PMI), and Central Control Unit (CCU).

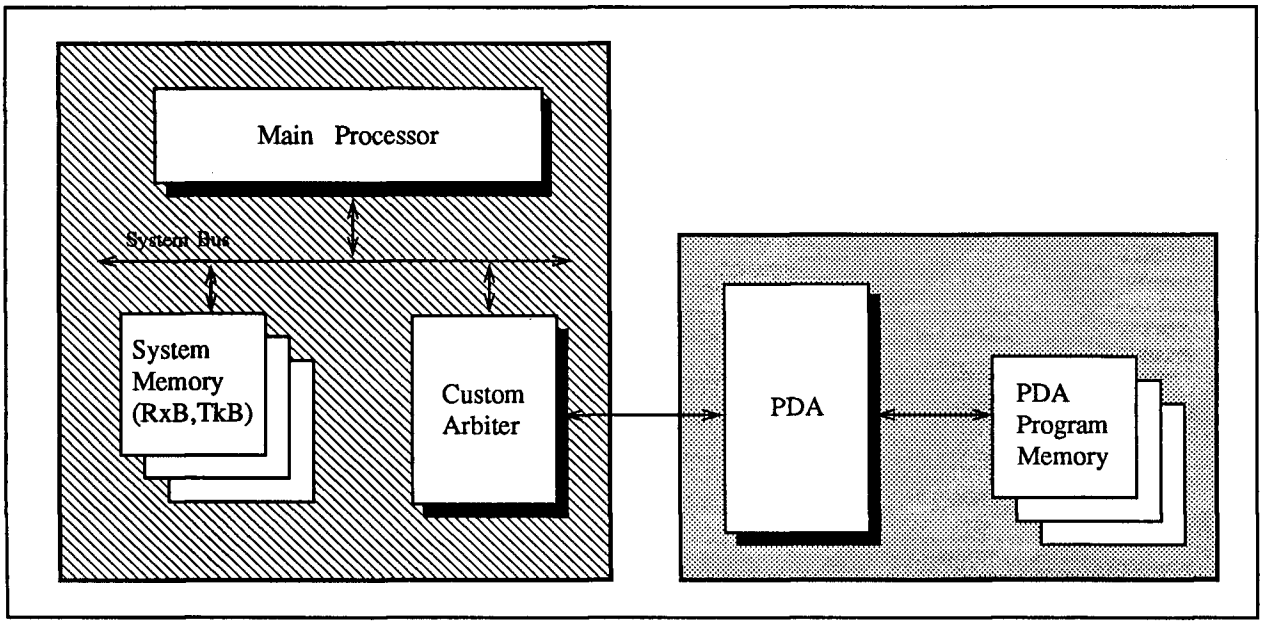


Figure 6. Simplified System Block Diagram

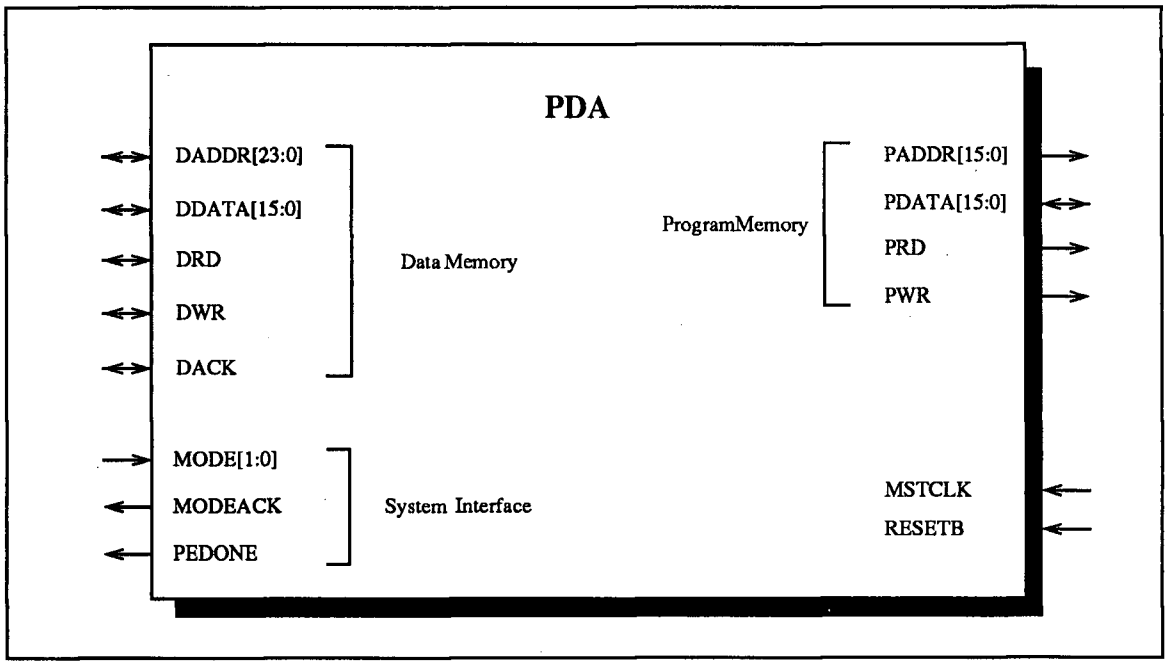


Figure 7. PDA Interface Signals

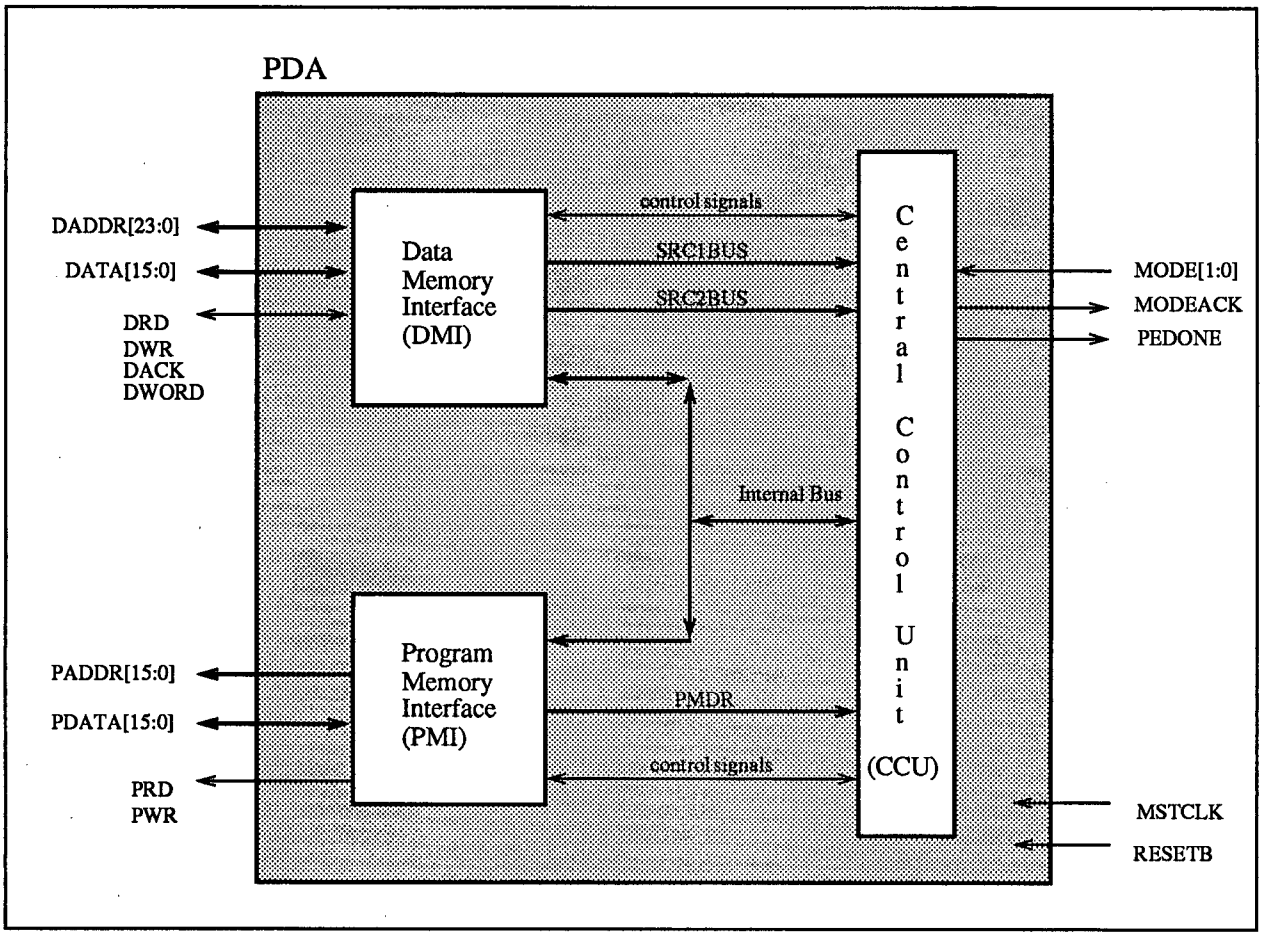


Figure 8. PDA Simplified Block Diagram

3.2. PDA Components

The three basic blocks in PDA are described in the following.

3.2.1. Data Memory Interface (DMI)

The function of DMI is to handle system memory access. The system memory uses an explicit acknowledge signal to indicate the completion of a request cycle. A block diagram of DMI is shown in Figure 9. In configuration mode, the DMI accepts read/write requests from the system to access PDA configuration registers. It requires at least two system clock cycles to process a request. In program execution mode, the DMI is used to read RxB and write TkB resided in the system memory. It implements an automatic RxB data prefetch algorithm to alleviate the need of an explicit instruction for reading RxB data. Built-in detection for end of RxB element and end of PDU is also included in the prefetch algorithm. A three level deep FIFO (First In First Out) buffer is used to hold the prefetch data. At the output side, there is a one level holding register for writing to TkB. The use of these buffers allows other instructions to be executed in parallel while servicing a system memory request. Consequently, it reduces the performance degradation due to slow system memory access time.

The following subsections describe each of the DMI components in more detail.

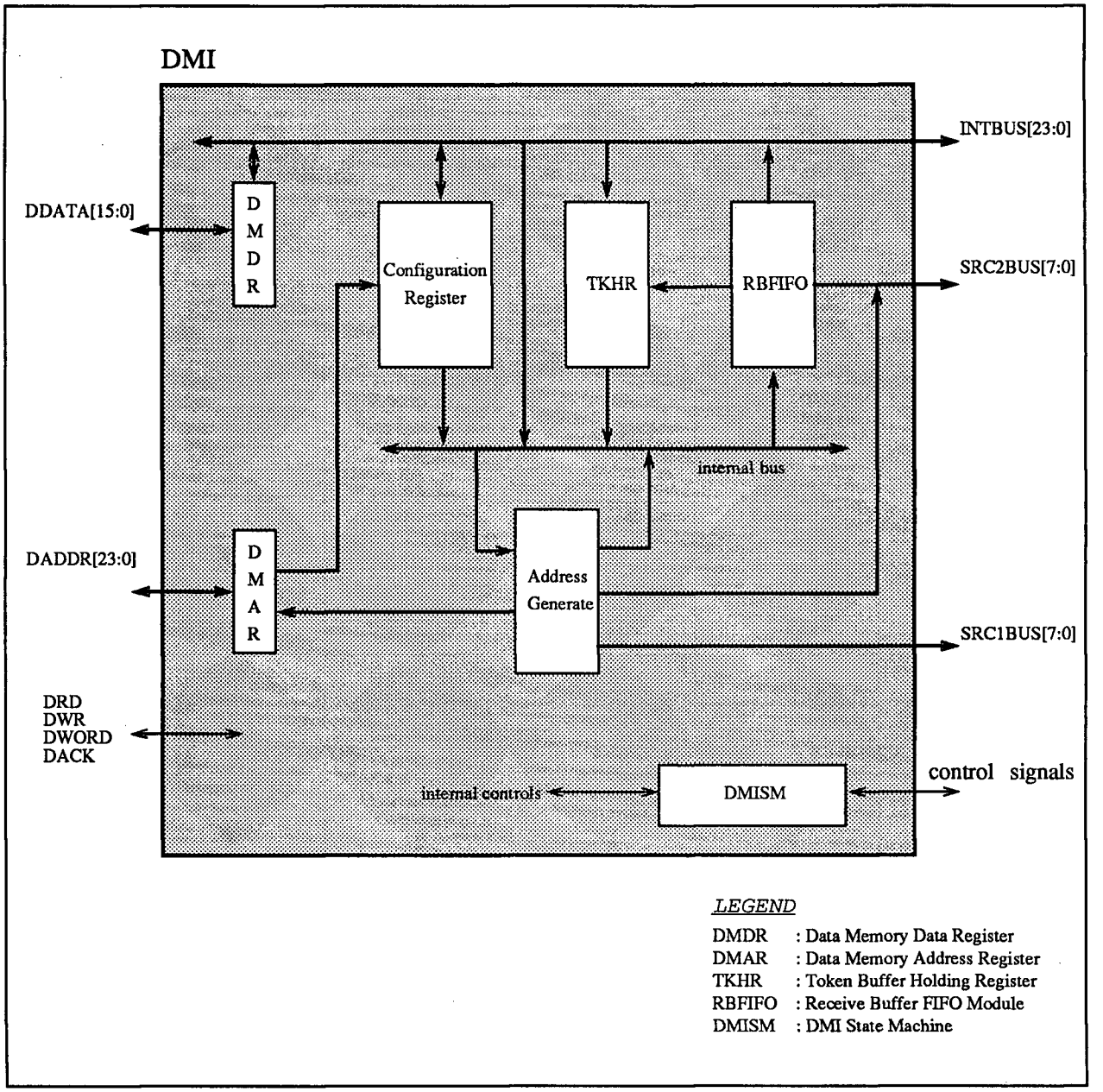


Figure 9. Simplified Block Diagram of DMI

3.2.1.1. DMI State Machine (DMISM)

The DMISM is a state machine supplying control signals to the DMI internal modules. In configuration mode, it monitors the data memory interface to handle system read/write operation to the PDA configuration registers. In program execution mode, the DMISM controls the automatic RxB access algorithm and the pipelined TkB write request function. When the PDA just switches to program execution mode, the DMISM begins to load the RxB element parameters, which includes the Dstart, Dend, and Flag as described in the previous chapter. Armed with this information, the DMISM proceeds to fetch the first data byte or word depending on whether the starting address is at an odd or even boundary. After the first data prefetch, the DMISM returns to idle state and continues to monitor the internal signals awaiting for the following possible operations. Firstly, if the RBFIFO is less than two level full and the end of PDU has not yet reached, the DMISM will make a RxB data read request. There are two possible cases under this condition. If the end of the current RxB element has been processed, the parameters from the following RxB element in the link list has to be loaded into PDA first. Then the RxB data read request can be carried out using this new RxB element. However, if the end of current RxB element has not been read, the next RxB data can be obtained at the successive location from the last read. Secondly, if the TKHR is full, the DMISM will generate the corresponding TkB write request. The procedure involves loading the TKHR data values into DMDR and calculating the memory address by adding the offset specified in TKHR to the TkB base pointer. The RxB read and TkB write requests are handled in a round-robin fashion so that neither one can monopolize the use of the data memory interface.

As will be described later, it is possible to select an option so that the RxB element

at the end of the last program execution can be inherited for the current program execution. This eliminates the starting steps of loading in the RxB element parameters and the first RxB data fetch. As a result, multiple-pass decoding can be run with less overhead.

3.2.1.2. PDA Configuration Registers

Table 1 lists the eight configuration registers. The first register, PDACR, allows selection of PDA options. Bit 0 is for selecting if program control will automatically jump to the address specified in *TRAPP* register when an out-of-data condition is detected. Out-of-data occurs when the PDA program instruction requests more RxB data after the end of PDU is reached. This option, when selected, helps to reduce the use of conditional instruction to check for out-of-data at the end of every RxB access instruction. Regardless of the bit 0 setting, the condition code *od* will always be set so that conditional instructions can still be used to detect such a out-of-data violation. Nullification of the prefetch instruction can be enabled by setting bit 1 of PDACR register. As will be explained in the later section, some of the jump instructions use the delay-by-one branch technique, in which the instruction immediately following the jump instruction, referred to as delay slot instruction, is always executed. If bit 1 is set, the delay slot instruction will be nullified to a *nop* instruction. Consequently, it is not necessary to put a *nop* after every jump instruction explicitly. Sometimes, it is possible to fill the delay slot with a useful instruction, one that will not affect the outcome regardless of whether the jump is taken or not. In such a case, the nullification can be disabled to take advantage of using the prefetched instruction. Bit 2 of PDACR

register is for selecting if execution is to resume from the previous RxB pointer and decoding program. When the PDA is put into program execution mode with bit 2 not set, it first fetches the data byte from the RxB specified by RBPL and RBPH registers. Then it executes the decoding program starting from the address given in PMBP register. However, when bit 2 is set, the PDA resumes from the RxB and decoding program used at the end of the last execution. This option is useful for PDU decoding that requires more than one pass, such as in the case of some transport PDUs that require connection identification before the exact format is known.

RBPH and RBPL registers specify the RxB base pointer. This pointer refers to the first RxB element of the PDU to be decoded. These registers are used only if bit 2 of PDACR is not set. TBPL and TBPH registers contain the TkB base pointer. All TkB reference in the decoding program will use these registers to calculate the complete address. PMBP register gives the address of the first instruction in the decoding program. This is used only if bit 2 of PDACR is not set. TRAPP register provides the address of the instruction to begin execution when an out-of-data occurs as explained in the above. This register is used only if bit 0 of PDACR is set. Finally, CNTXT register contains the context information necessary for decoding a particular PDU. This includes the selection of various services or options that are crucial to the decoding program. The use of this CNTXT register provides a mechanism for passing additional parameters to the decoder.

Table 1. PDA Configuration Register Description

Name	Addr	Description
PDACR	0	PDA option selection (3 bits): bit 0 = set to enable program control transfers to the address specified in TRAPP register when an out-of-data condition is detected. If bit is not set, program control remains at normal path. Nonetheless, the condition code *od* will always be set. bit 1 = set to enable nullification of the prefetch instruction when the current instruction results in a jump to other location. If bit is not set, the prefetch instruction will not be nullified. bit 2 = set to indicate that execution resumes from the previous RxB pointer and decoding program. If bit is not set, execution will start from the new RxB and decoding program as specified in RBPL, RBPH, and PMBP.
RBPL	1	RxB Base Pointer low word. (16 bits)
RBPH	2	RxB Base Pointer high byte. (8 bits)
TBPL	3	TkB Base Pointer low word. (16 bits)
TBPH	4	TkB Base Pointer high byte. (8 bits)
PMBP	5	Program Memory Base Pointer. (16 bits)
TRAPP	6	Out of Data violation trap address. (16 bits)
CNTXT	7	Context parameters. (8 bits)

3.2.1.3. Receive Buffer FIFO (RBFIFO)

The RBFIFO is a three level FIFO to store the prefetched RxB data. Each level contains a <data, pointer, eopflag> triplet, where data is the data byte from the RxB, pointer is the corresponding absolute address, and eopflag is a flag indicating if the corresponding data is the last byte in the PDU. The RBFIFO saves the data from RxB and presents to CCU when needed. A head and a tail values are used to keep track of the FIFO operation. Input data byte from RxB is put at the tail of the RBFIFO, whereas, outgoing data byte to CCU is removed from the top of the RBFIFO. Sometimes the pointer to the RxB data byte is needed, therefore the RBFIFO also keeps track of this value. Saving the 24 bit full address is necessary for this purpose because it may not be simple to derive the address in a link list structure instead of a linear storage. Since the DMI and CCU are independent machines running in parallel, it is necessary to lock the buffer from CCU when loading data. The RxB prefetch algorithm is straight forward. Whenever the RBFIFO is less than two level full, a prefetch request is generated unless the end of PDU has reached. The rationale of choosing a three level deep FIFO as oppose to a smaller size is to take full advantage of the 16 bit system data bus. Each system data request can potentially fill two levels of the buffer. Choosing a size of one level will obviously results in almost twice as many requests. Adopting a size of two will also lead to changing the prefetch algorithm to request only when the buffer is empty. As a result, the CCU is more susceptible to blocking from waiting for next RxB data.

3.2.1.4. Token Buffer Holding Register (TKHR)

Whenever a write to TkB is necessary, the data is first saved onto the TKHR. Then an output request is generated to the DMISM. Further TkB writes will be blocked until the outstanding request is serviced. The TKHR is consisted of three elements, the request type, an 8 bit offset, and a 24 bit data. There are three request types based on the data size: byte (8 bit) request, word (16 bit) request, and pointer (24 bit) request. For word and pointer requests, the TkB location is assumed to be at a word boundary. In the pointer request, the 24 bit data is padded with 8 leading zeros to form a 32 bit value, and two consecutive word write requests are done to put this value in TkB. In all cases, the offset value is used to derive the memory location of the request by adding it to the TkB base pointer.

3.2.1.5. Address Generate

The Address Generate Module contains a 24 bit adder to calculate the address for system memory input and output. In addition, it also includes a few registers to hold the current RxB parameters and a comparison logic for end of RxB and PDU detection.

3.2.2. Program Memory Interface (PMI)

The PMI handles program memory access. This interface uses a fix access time of two times the system clock period. Figure 10 shows a block diagram of PMI. The program counter together with the incrementer allows sequential instruction access. Forward and backward relative jumps are supported by the 16 bit adder and the pass-or-complement logic. The program counter is resettable so that an absolute address can be loaded to PMAR as well.

The PMISM provides control signals according to the PDA operating modes. In program execution mode, the PMI reads data from the program memory to PMDR. While in program loading mode, the PMI writes data to program memory from PMDR. It issues a read request every two clock cycles unless a halt signal is received from the CCU, in which case, the next read request is postponed until the current CCU instruction execution is completed. At the beginning of a read cycle, the address is being latched into PMAR and PC at the same time the PRD signal is asserted. One clock cycle later, the PMDR latch is switched on. At the end of two clock cycles, the value in PMDR will be valid and the PMDR latch is switched off. The PMI is then ready for the next read request. The 16 bit address bus provides a 64K direct addressable range. This is sufficient as the decoding programs are small in general.

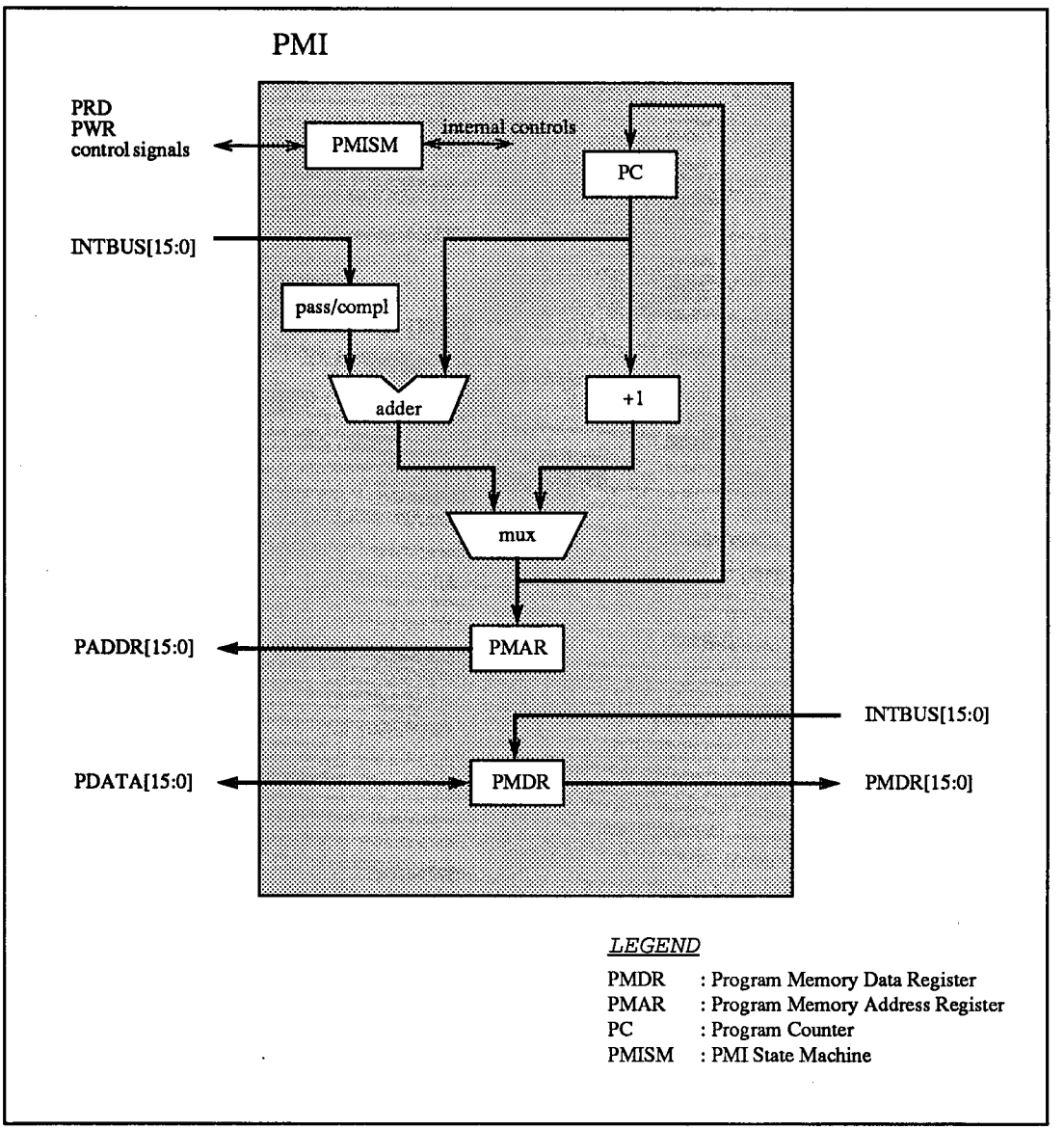


Figure 10. PMI Simplified Block Diagram

3.2.3. Central Control Unit (CCU)

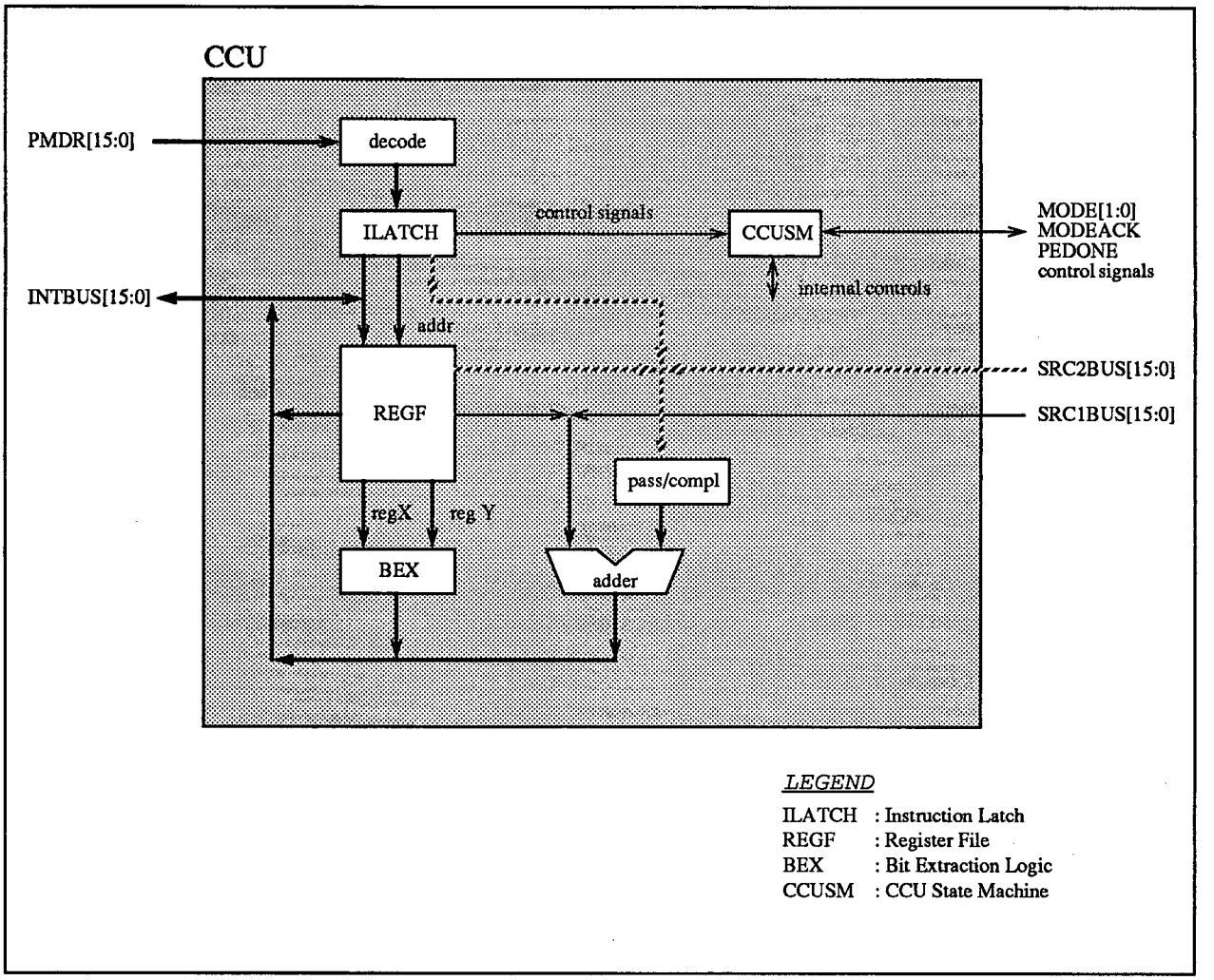


Figure 11. CCU Simplified Block Diagram

The CCU is mainly responsible for instruction decode and execution. It also provides timing signals to other modules based on the current mode setting and machine states. Figure 11 displays the block diagram of CCU. In both configuration and program loading mode, the CCU is essentially idle allowing DMI and PMI to take over the task of configuration and program loading. In program execution mode, the CCU first informs the DMI and PMI to start prefetching RxB data and program instruction respectively. As shown in Figure 11, instruction enters the CCU by going through decoding and then is being latched for execution. After the first instruction is ready for execution, the CCU will continue to operate in a pipelined fashion as illustrated in Figure 12.

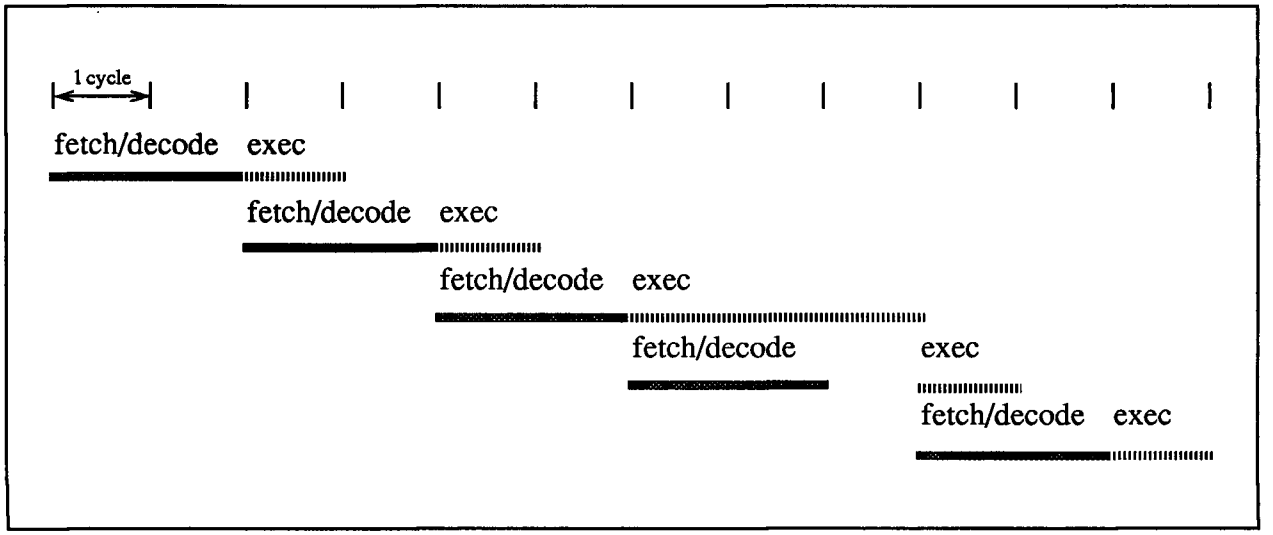


Figure 12. CCU Execution Pipeline

The instruction fetch and decode phase in the CCU execution pipeline always uses two clock cycles, whereas, the instruction execution phase may take up from one to multiple clock cycles. To provide internal work space, the REGF module is used. It contains 16 registers as summarized in Table 2. Register X and Y are 8 bit registers that can be used in any instruction that specified byte size register as operand. They are also used exclusively for the bit extraction (BEX) logic. The rest of the registers are labelled as R1 to R14. All of these registers are 16 bit and can be byte or word accessible. All can be used in any instruction that require register reference. Register R14 carries a special function, as it is being used for length calculation in the *skip* and *skip.end* instructions. The BEX module deals with the bit shifting and extraction operation. It contains a barrel shifter and masking logic to perform all bit manipulation in one clock cycle. The 16 bit adder and pass-or-complement logic are used for addition and subtraction functions.

The CCUSM sets the operating mode of the PDA according to the mode request signals. It also consists of the instruction execution unit and other hardwired state machines to support the PDA instructions.

Table 2. Internal Register Description

Name	Description
X	8 bit general register and is also used for the bit extraction logic.
Y	8 bit general register and is also used for the bit extraction logic.
Rn	16 bit general register with n = 1 to 14. All can be byte or word accessible. R14 is also being used in the *skip* and *skip.end* instructions.

3.3. PDA Instruction Set

Instruction set design has a great impact on a processor architecture. In the PDA design, it is necessary to identify and optimize the number of instructions being used to implement protocol decoding. A small and simple instruction set provides a number of implementation advantage that can substantially enhance performance [Patterson-80]. For instance, a simple control structure can be adopt to allow faster execution. Such a design also shortens the design and debugging time as well. The design methodology used in PDA is to choose a simple efficient instruction set, and optimize important functions by hardwired logic implementation. Table 3 lists the PDA instruction set which has a total of 26 instructions including all addressing mode. Table 4 describes the condition codes being used. The instruction set employs a fix 16 bit format so that all operands are fetched with the instruction. The *jp.li.cc* is the only exception as it requires a 16 bit operand. There are only two addressing modes, either *immediate* or *register*. In immediate mode, the operand is supplied by the instruction. This includes size of 4, 6, 8, 10, and 16 bit values. In register mode, the operand is one of the internal registers listed in Table 2.

The PDA instruction set can be classified into four groups: *RxB related*, *TkB related*, *data manipulation*, and *program control*. The RxB related instructions deal with loading RxB data into internal register or skipping over data in the RxB. The *get.b* and *get.w* instructions remove data from the top of the RBFIFO and load it into the specified register. If there is not enough data in RBFIFO, these instructions will be blocked until the next RxB fetch is completed. *Get.b* and *get.w* require a minimum of one and three clock cycles to complete respectively. The *skip* instruction is used to skip over the specified number of data bytes in the RxB. It is used for decoding long fields that will be represented by its pointer and

length. This is a complicated process as it may be necessary to transverse the RxB until enough bytes are skipped. The entire algorithm is implemented in hardware and require at least five clock cycles to complete. The *skip.end* instruction is used to skip to the end of the PDU and calculate the number of bytes being skipped. This number and the specified offset is then loaded onto TKHR awaiting for a TkB write request. This instruction shares the same algorithm used in the *skip* instruction and is implemented as hardwired state machines. This instruction requires at least four clock cycles to complete. It should be note that register R14 is being used for the length calculation in both *skip* and *skip.end* instructions. Therefore, caution must exercised if R14 is also being used elsewhere in the program. All RxB related instructions, with the exception of *skip.end*, may cause the out-of-data condition, in which, more data is required than the remaining number of bytes left in the PDU. The out-of-data violation may force the program control to transfer to the address specified in the TRAPP register as described above.

The TkB related instructions involve with writing to TkB. All these instructions execute the similar function of putting a data value and an offset onto TKHR, which in turn generates a TkB write request. If there is an outstanding request, the instruction is blocked until the previous request is processed. There are three types of TkB write requests as explained earlier. *Out.b* and *out.w* are used for byte and word write request respectively. *Outptr* is used for pointer request. For the pointer request, the pointer is obtained from the element on the top of the RBFIFO. In this case, only the pointer value is being used, the element itself is not removed from the RBFIFO. All TkB related instructions require a minimum of one clock cycle to execute.

Data manipulation instructions handle internal register operation. All of these

instructions are executed in one clock cycle. The *extract* instruction extracts bit field from register X and saves the result onto register Y. It provides all bit field extraction and justification functions including concatenation of bit fields. The logical expression that it performs is as follow:

$$((X >> b) \& n) | (Y \& (0x0ff >> d))$$

where X and Y are the contents of register X and Y respectively; b and d are values from operand1 and n is the content of operand2 in the instruction. The >>, &, and | operators are right shift, bitwise AND, and bitwise inclusive OR as defined in the C programming language [Kernighan-78]. The *mov.b* and *mov.w* instructions are used to move the content of an internal register or immediate values to another internal registers. The data is not examined as it is moved, so no condition code is set. The *cmp* instruction subtracts its operands and set the condition codes accordingly. Its operands are not modified by the instruction. For the *sub* instruction, 16 bit subtraction is done and the result is saved in the subtractor operand. Appropriate condition codes are set according to the result of the subtraction.

The program control instructions may be used to alter the normal sequential flow of the program. The *nop* instruction does not perform any operation. The processing states, other than the program counter, is unaffected. Execution continues with the instruction immediately following the *nop*. The *end.cc* instruction terminates program execution if the condition code is set; otherwise, it does not perform any action. Before terminating the program execution and returning to configuration mode, the PDA will wait until all pending actions are completed. Thus, all RxB data prefetch, outstanding TkB write request, and program instruction prefetch will be completed. The PDA also notifies the system of program termination by asserting the PEDONE signal. The *jp.cc.si* is one of the three jump

instructions available. If the condition code is set, program execution continues at location given by the sum of the current program counter and the operand; otherwise, no operation occurs. The operand is a 10 bit two's complement immediate value giving a forward and backward range of 511. This jump instruction uses the delay-by-one branch technique, which is being adopt by many other RISC or pipelined processors [Gross-82, Patterson-81, Radin-83]. In the delay-by-one branch case, the instruction immediately following the jump instruction is always executed regardless of whether the jump is taken or not. Hence, the prefetch during the execution of the jump instruction is not wasted. However, it is difficult to always find a useful instruction to fit into the delay slot. *Nop* instruction can be used but it suffers two drawbacks. Firstly, the program size will increase. Secondly, the jump not taken case may suffer from slower execution. To avoid adding in *nop* explicitly, instruction nullification can be used. It nullifies the prefetch instruction to a *nop* if the jump is taken; and it will leave the prefetch instruction untouched if the jump is not taken. This feature is inspired from the HP Precision Architecture Processor [Mahon-86]. The HP scheme, however, is much more sophisticated as it allows nullification on a per instruction basis and it can also extend to other instruction type as well. The *jp.cc.r* instruction is identical to *jp.cc.si* except the jump location is specified by the content of an internal register. This instruction provides a mean of implementing subroutine call. Consider the program segment shown in Figure 13. In this example, the main program sets up the return address in register R12 before jumping to the subroutine program. At the end of the subroutine, the *jp.cc.*r instruction is used to return to the main program. The use of subroutine can often reduce program size and increase program modularity, although the execution time may suffer.

```
/* main program */
MAIN:               .
                    .
              mov.b.ri      r12, MAINCONT
              jp.si         SUBBEGIN
MAINCONT:           .
                    .



/* subroutine */
SUBBEGIN:           .
                    .
              jp.r          r12
```

Figure 13. Illustration of *jp.cc.r* Instruction

Finally, the last jump instruction, *jp.cc.li*, causes program to continue at the specified address if the condition code is set. This instruction is 32 bit long with the 16 bit immediate value occupying the second instruction word. If the jump is taken, it occurs after the operand is fetched and no delayed branch is used. If the jump is not taken, no operation occurs. In this case, the operand is fetch and nullified to a *nop* so that the normal program flow is not affected.

Table 3. PDA Instruction Set Summary

Opcode	CC set	Description
(a) RxB related		
get.b(reg)	od, ep	get the next RxB byte into the specified register.
get.w(reg)	od, ep	get the next RxB word into the specified register.
skip(reg)	od, ep	skip the specified number of bytes.
skip.end.r(reg) skip.end.i(imm8)	ep ep	skip to the end of PDU and output the number of bytes (16 bit value) being skipped at the specified offset.
(b) TkB related		
out.b.rr(reg,reg) out.b.ri(reg,imm8) out.b.ii(imm4,imm8) out.b.ir(imm8,reg)	none	output the specified byte as token at offset. Operand1 is the value and operand2 is the offset.
out.w.rr(reg,reg) out.w.ri(reg,imm8)	none	output the specified word as token at offset. Operand1 is the value and operand2 is the offset.
outptr.r(reg) outptr.i(imm8)	none	output the current internal RxB pointer as token at the specified offset.
(c) Data Manipulation		
extract(imm6,reg)	none	extract bit field from register X and save onto register Y. Operand1 gives the amount of shift and bit mask. Operand2 specifies where the result should place in register Y.
mov.b.rr(reg,reg) mov.b.ri(reg,imm8) mov.w(reg,reg)	none	move the content of operand1 to operand2.

Table 3. PDA Instruction Set Summary (con't)

Opcode	CC set	Description
cmp.rr(reg,reg) cmp.ri(reg,imm8)		compute (operand1 - operand2) and set the condition code accordingly. The condition code affected are: neq, eq, lt, gt, le.
sub.rr(reg,reg) sub.ri(reg,imm8)		perform (operand1 = operand1 - operand2) operation. The condition code affected are: neq, eq, lt, gt, le.
(d) Program Control		
nop	none	perform no operation.
end.cc	none	terminate program execution if the condition code is set.
jp.cc.si(imm10)	none	jump to the specified offset relative to the current instruction position. The offset is a 10 bit two's complement number to allow a range of +511 and -511.
jp.cc.r(reg)	none	jump to the specified location if the condition code is set.
jp.cc.li(imm16)	none	jump to the specified location if the condition code is set. The operand is in the next instruction. No delayed branch for this jump.

Table 4. PDA Instruction Set Condition Code Description

Name	Description
al	always flag.
ep	end of pdu flag. Set when the last byte in the PDU has been processed.
od	out of data flag. Set when more RxB data is requested after the end-of-pdu.
neq	not equal flag. Set if the result of *cmp* or *sub* instruction is nonzero.
eq	equal flag. Set if the result of *cmp* or *sub* instruction is zero.
lt	less than flag. Set if the result of *cmp* or *sub* instruction is negative.
gt	greater than flag. Set if the result of *cmp* or *sub* instruction is positive and nonzero.
le	less than or equal flag. Set if the result of *cmp* or *sub* instruction is negative or zero.
c*n*	the *n*th bit of the CNTXT register. n is 0 to 7.

3.4. Implementation and Cost Estimate

The PDA has been implemented with VGT200 series gate arrays using VLSI Technology's software tools [VLSI-88a]. The schematics of the design are shown in Appendix F. The VGT200 series gate array is utilizing CMOS 1.5 micron technology with a typical two-input NAND gate delay of 0.56 ns [VLSI-88b]. Most of the design is done using behavior language modeling with estimated delay added. Some of the control logic are implemented as logic gates and hence provide a realistic simulation. The PDA design has been simulated to run up to a clock speed of 20 MHz. The estimate gate count is 10,500 as shown in Table 5. The register file in CCU is the largest module which takes up about one third of the total gate count. The large number of registers are required to keep track of the PDU information so that it is not necessary to refer back to previous data bytes. The total gate count also includes about 10% of test logic.

The VGT200-042 gate array base has approximately 12,790 usable gates and could be used for the PDA design. The production unit cost is estimated at US$ 50.00. In addition, a program memory is required to complete the PDA system. This can be implemented by an 8K x 16 35ns static RAM module, which adds an additional US$ 20. The total unit cost becomes US$ 70. There is also a Non-Recurring Engineering (NRE) charge associated with mask and prototype generation. This is typically about US$ 50,000. Obviously, the VLSI design is very expensive when comparing with the software solution, in which the production cost is insignificant because the microprocessor and memory are already exist.

Table 5. PDA Gate Count Estimate

DMI			3770
	Configuration Register	1300	
	TKHR	400	
	RBFIFO	1400	
	Address Generate	1350	
	DMISM	230	
	other logic	90	
PMI			730
	register storage	320	
	PMISM	60	
	other logic	350	
CCU			5100
	REGF	3400	
	adder logic	350	
	BEX	160	
	decode logic	510	
	CCUSM	500	
	other logic	680	
Test Logic (~10%)			1000
			10500 gates

Chapter 4. Analysis and Evaluation

In this chapter, the PDA design is evaluated. It begins with examining the instruction distribution in three decoding programs to understand the instruction usage pattern in PDU decoding. Then the PDA design is subject to comparison with other systems. The comparison is based on program size and program execution measurement. Finally, the PDA performance in relation to system environment changes is discussed.

4.1. Instruction Set Usage

PDU decoding programs for X.25 link layer, X.25 packet layer, and X.224 transport layer protocols have been developed using the PDA instruction set (Appendix D provides a PDA program listing for the X.25 link layer). Table 6 lists the instruction distribution for all three programs. The most frequently used instruction is highlighted by underlining its value at each column. It is obvious from the table that the decoding programs are dominated by data manipulation instructions with TkB related and program control instructions about evenly distributed. The total number of instruction of a program gives a relative measure of the complexity of the decoding algorithm. The result has confirmed that the transport PDU is the most complex of all three. Instruction nullification and out-of-data program control transfer options are being used in all the programs. Therefore, no nop appears in any program and the use of conditional jump instruction is also reduced.

Table 6. Instruction Distribution for Three Decoding Programs

	link	packet	transport	total
get.b	9	16	25	50
get.w	1	0	23	24
skip	0	4	2	6
skip.end.i	1	4	3	8
skip.end.r	0	0	0	0
out.b.ii	16	16	28	60
out.b.ir	0	0	0	0
out.b.ri	20	34	26	80
out.b.rr	0	0	4	4
out.w.ri	1	0	24	25
out.w.rr	0	0	1	1
outptr.i	1	8	6	15
outptr.r	0	0	3	3
extract	24	26	12	62
mov.b.ri	27	42	109	178
mov.b.rr	0	0	0	0
mov.w	0	0	0	0
cmp.ri	12	23	35	70
cmp.rr	0	0	2	2
sub.ri	0	0	3	3
sub.rr	0	0	1	1
nop	0	0	0	0
end.cc	9	17	16	42
jp.cc.si	25	36	85	146
jp.cc.r	0	0	3	3
jp.cc.li	0	0	0	0
RxB related	11	24	53	88
TkB related	38	58	92	188
data man.	63	91	162	316
program con.	34	53	104	191
total	146	226	411	783

4.2. PDA Performance

The PDA has been tested in a simulated environment as shown in Figure 14. The *Data Memory System* consists of an interface to provide the proper handshake and a memory storage to hold the RxB and TkB. The program memory contains the various decoding programs. Both data and program memories are dedicated to PDA, and hence their access times are constant.

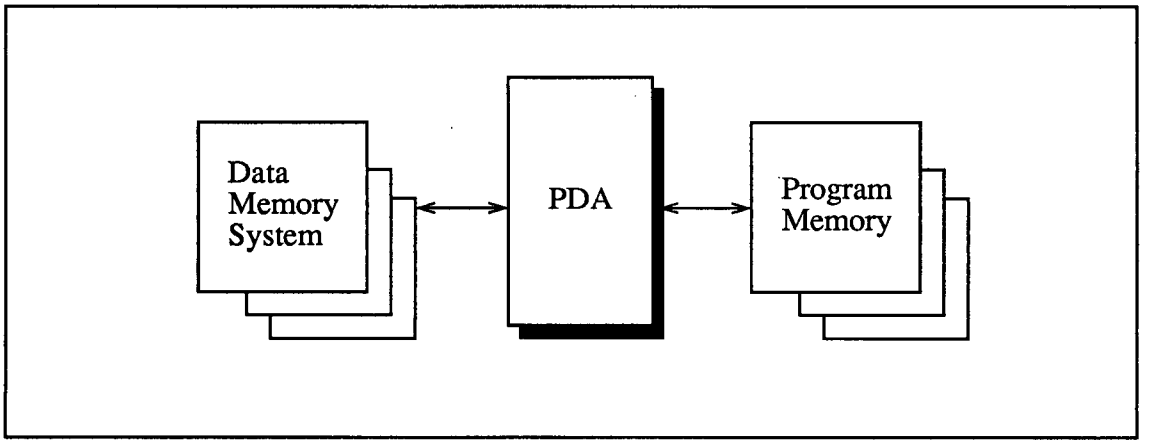


Figure 14. PDA Simulation System

Two other systems are chosen to compare with the PDA. The first one is a subsystem of the IDACOM MPT368.2 protocol tester [IDACOM-89]. There are two front-end CPU boards inside the MPT with each board comprising of a 16 MHz 68000 microprocessors and 1 Mbyte of memory. It is possible to isolate one front-end CPU board

from the rest of the system, in which case, most of the multiprocessor communication overhead is eliminated, resulting in a dedicated system as shown in Figure 15. An equivalent set of decoding programs has been written in C language and compiled into 68000 binary code to run on the MPT subsystem (A C program listing for decoding the X.25 link layer can be found in Appendix E). The Green Hills' C compiler is being used to generate highly optimized 68000 codes [OASYS-87]. The second system used for comparison is made up of an ideal CPU with a data memory providing the RxB and TkB. The ideal CPU takes zero time to fetch and execute any instruction except memory access instruction which requires the number of clock cycles imposed by the memory limit. This is the ideal case in which there is zero processing overhead. Therefore, the execution time of the ideal CPU system represents the minimum attainable time in a single memory interface architecture. These two systems are referred to as *68000* and *ideal CPU* in the rest of this paper.

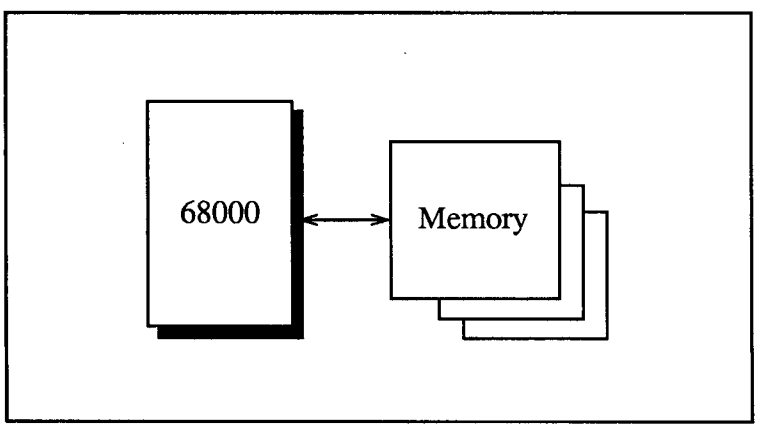


Figure 15. Isolated MPT Subsystem

4.2.1. Program Size Measurement

The program size often reflects how optimize the instruction set and architecture is for an application. Table 7 lists the decoding program size in PDA and 68000 codes. The result shows that the 68000 program is about five times as large as the PDA equivalent. The smaller PDA program size can be attributed to a small optimized instruction set and hardware assisted function. The hardware assisted function especially plays a major role. For example, the RxB access is completely supported by the hardware and no explicit instruction is required. A compact program size makes it economical to use dedicated program memory to help reducing access to system memory. Note that for the 68000 code, there are some common routines among the decoding programs, therefore, the total is not equal to the sum of individual decoding programs.

Table 7. Program Size Comparison

Program	PDA Code (i)	68000 Code (ii)	Factor ((ii)/(i))
X.25 link layer	292 bytes	1630 bytes	5.58
X.25 packet layer	380 bytes	1830 bytes	4.82
X.224 transport layer	822 bytes	4508 bytes	5.48
Complete three layers	1494 bytes	7224 bytes	4.84

4.2.2. Program Execution Time Measurement

Table 8 tabulates the average execution speed of the PDA, the 68000 and the ideal CPU systems in terms of system clock cycle.

Table 8. Average Execution Speed Comparison

Program	PDA (i)	68000 (ii)	Ideal (iii)	((ii)/(i))	((i)/(iii))
X.25 link layer	108 cycles	1892 cycles	31 cycles	17.52	3.48
X.25 packet layer	142 cycles	2685 cycles	47 cycles	18.91	3.02
X.224 transport layer	270 cycles	4526 cycles	86 cycles	16.76	3.14

For each decoding program, a typical set of various PDU types is being decoded and the result is averaged. The result is expressed in terms of system clock cycle instead of absolute elapse time in order to normalize the effect of system clock speed difference. The PDA system is simulated at a 20 MHz system clock with three clock cycle system memory access time. Whereas the 68000 system is running at 16 MHz utilizing zero-wait state memory. In both cases, the time requires to set up the decoding parameters is also accounted for. In the PDA system, these decoding parameters are the eight configuration registers described in Table 1 in the previous chapter. After setting up the configuration

registers, the system can put PDA in program execution mode and wait for task completion as signalled by PEDONE. In addition, the setup time for reconfiguring the configuration registers during the second pass of a two-pass decoding is also included in the result. For the 68000 system, the decoder is implemented as procedure call with three parameter variables. These variables are RxB pointer, TkB pointer, and selection of protocol options. This overhead for making a procedure call is included in the execution time measurement. Finally, the result from the ideal CPU system is obtained from multiplying the sum of RxB read and TkB write requests by the system memory access time. The system memory access time is the same as in the PDA simulation, which is three clock cycles. It is also assumed that one read request can always fetch two data bytes from the RxB to take full advantage of the 16 bit data bus interface. In all three systems, the RxB element size is set at 32 bytes for the simulation.

The result from Table 8 shows that the PDA outperforms the 68000 by at least 16 times in all three decoding programs. In addition, the ideal CPU system executes about three times faster than the PDA in all cases. Table 8 only gives the relative performance of PDA. It is also possible to express the result in terms of number of PDU decoded per second as shown in Table 9. These numbers are obtained from dividing the PDA clock speed, which is 20 MHz in this case, by the result in Table 8. It should note that, firstly the result is based on the average execution time for a selected set of PDU types. Therefore, the result may vary for other PDU types. Secondly, these numbers represents a maximum attainable condition because a constant three clock cycle system memory access time is being used for the simulation. In a real system, the system memory access time is expected to be variable and slow.

Table 9. PDA Average Execution Speed

Program	Execution Speed
X.25 link layer	185,000 PDU/sec
X.25 packet layer	141,000 PDU/sec
X.224 transport layer	74,000 PDU/sec

Table 10. I Frame Decoding Speed Comparison

	PDA (i)	68000 (ii)	Ideal (iii)	((ii)/(i))	((i)/(iii))
X.25 I Frame	108 cycles	1966 cycles	48 cycles	18.20	2.25

To examine the result more closely, the following analysis focus on decoding the *information frame* (I frame) in X.25 link layer. The I frame has an information field length of 26 bytes and it takes up two RxB elements. It is expected that the TkB fields shown in Figure 5(c) are properly filled in after the decoding. In the PDA program, 108 clock cycles is required for the I frame decoding. Of the total, 27 clock cycles (25%) are spent on configuration, and the rest (75%) is spent on executing the 23 instructions shown in Table 11. All instructions are operating at a rate of one instruction in two clock cycles with the following exception. The two get.b instructions use more time because they need to wait for the prefetch data to enter the internal RBFIFO module. Some of the TkB write request instruction also use more time because they are blocked when the previous request has not been serviced yet. Another time consuming instruction is skip.end, which calculates the remaining length of the PDU. This calculation involves determining the sum of the remaining length of the current RxB and the second RxB elements. After the calculation, the result is saved to the TKHR awaiting for TkB write request. The entire instruction requires 28 clock cycles. Finally, the end instruction also takes more time as it has to wait for the outstanding TkB write request to complete before signalling end of execution. The only conditional jump instruction that results in jump taken is instruction number 12. The following nop is the result of instruction nullification. It has been found that the DMI module is busy handling system memory request 85.19 % of the time during the PDA program execution mode. This indicates that the decoding is efficiently keeping the DMI busy and is taking advantage of the system memory waiting time to execute other instructions in parallel.

Table 11. PDA Instruction Trace for I Frame Decoding

Instruction Trace			Execution Time
[1]	out.b.ii	slp_ok, slp_error_off	2 cycles
[2]	mov.b.ri	rr13L, slp_empty	2 cycles
[3]	get.b	rrY	5 cycles
[4]	out.b.ri	rrY, slp_addr_off	2 cycles
[5]	mov.b.ri	rr13L, slp_no_control_field	2 cycles
[6]	get.b	rrX	3 cycles
[7]	mov.b.ri	rr1L, m1	2 cycles
[8]	extract	0, 4, rr1L	2 cycles
[9]	out.b.ri	rrY, slp_pf_off	2 cycles
[10]	extract	0, 0, rr1L	2 cycles
[11]	cmp.ri	rrY, 0	2 cycles
[12]	jp.eq.si	SLP_I	2 cycles
[13]	nop		2 cycles
[14]	out.b.ii	slp_i, slp_type_off	2 cycles
[15]	jp.cntxt0.si	SLP_I128	2 cycles
[16]	mov.b.ri	rr1L, m3	2 cycles
[17]	extract	0, 1, rr1L	2 cycles
[18]	out.b.ri	rrY, slp_ns_off	2 cycles
[19]	extract	0, 5, rr1L	2 cycles
[20]	out.b.ri	rrY, slp_nr_off	3 cycles
[21]	outptr.i	slp_info_ptr_off	4 cycles
[22]	skip.end.i	slp_info_len_off	28 cycles
[23]	end		4 cycles

The C program requires 1966 clock cycles to decode the same I frame in the MPT subsystem. The program tracing, which displays only those statements that are being executed, is shown in Table 12. The execution time for each statement is also given. The execution time is derived from its compiled 68000 code [Motorola-82]. The decoding starts at the *slp_decode* procedure. It begins with a procedure call to *init_var* to set up the internal variables necessary to keep track of the RxB access. Then the I frame can be read from the RxB a byte at a time using the *get_next_byte* procedure. In the get_next_byte procedure, explicit check is made to detect end of PDU and end of current RxB element condition. When end of PDU is detected, it implies that a possible frame error has occurred. When end of current RxB element is detected, the next RxB element in the link list should be used for subsequent RxB data access. To calculate the information field length in the I frame, the *get_remain_len* procedure is used to determine the remaining length of the PDU. A total of 147 68000 instructions are executed.

The speed advantage of the PDA over 68000 is largely due to the following factors. Firstly, the PDA incorporates hardware assisted functions to reduce the number of instruction and to gain faster execution. The PDA has an implicit RxB access algorithm with built-in end of PDU and end of current RxB element checking. Another feature that is supported by the hardware is the remaining length calculation. These two hardware assisted functions essentially replace the supporting procedures defined in the C program. The special bit extraction logic also helps to reduce instruction count. Secondly, the PDA instruction has a faster execution rate. The small PDA instruction set enables all instruction but one to specified in one word (16 bits) and the instruction can be fetched in one read. In contrast, most of the 68000 instructions requires multiple reads to load the entire instruction. Furthermore, the PDA uses a fast dedicated program memory to ensure fast

instruction issue rate. Thirdly, the separate system and program memory interfaces in the PDA enables parallel operation. The PDA can continue to execute other instruction while the system memory interface is handling the previous system memory request.

The ideal CPU system takes only 48 clock cycles to decode the same I frame. This execution time is consisted of 7 read and 9 write requests to the system memory. The read requests are used to access the RxB, while the write requests are used to fill in the TkB. The 48 clock cycles execution time is the minimum possible speed to decode the given I frame. The execution time difference between the PDA and the ideal CPU systems is contributed by the following factors. Firstly, there is no decoding parameter set up time in the ideal CPU system. However, about 25% of the execution time is accounted for configuration in the PDA system as mentioned above. Secondly, to complete a system memory access in the ideal CPU system, only three clock cycles are needed. Whereas, the PDA requires four clock cycles. The one additional clock cycle is needed by the PDA internal logic to begin servicing a system memory request. Thirdly, the PDA requires finite amount of time to execute an instruction. Although the separate system and program memory interfaces together with the RBFIFO and TKHR modules permits parallel instruction execution during a system memory access, there are always instances in which instruction is executing without taking advantage of this parallelism. Fourthly, the prefetch algorithm used in the PDA always reads the next RxB data when the RBFIFO is empty and end of PDU has not yet reached. In the ideal CPU system, RxB data is fetched only when needed. In this particular case of decoding the given I frame, the PDA generates two additional RxB read requests to prefetch the content of the information field, which is being skipped over by the ideal CPU system.

Table 12. C Program Trace for I Frame Decoding

C Program Trace	Exec. Time
`/*`	
`*  Variable Declaration`	
`*/`	
`unsigned char eopdu, *cur_byte_ptr, error_value;`	
`unsigned char last_rxb_flag, cur_byte, cntxt;`	
`int  validcount;`	
`unsigned int  cur_word;`	
`struct rxbuffer {`	
`  struct rxbuffer *next_rxbptr;`	
`  char  dstart;`	
`  char  dend;`	
`  unsigned char  flag;`	
`  unsigned char  data[25];`	
`} *cur_rxbptr;`	
`struct rxbuffer rbuf[20];`	
`struct tkbuffer {`	
`  unsigned char type, error, addr, pf;`	
`  unsigned char nr, ns, vr, vs, cr, w, x, y, z;`	
`  int  frmr_cntl, info_len;`	
`  unsigned char *info_ptr;`	
`};`	
`struct tkbuffer *cur_tkbptr;`	
`struct tkbuffer tbuf[20];`	
`/*`	
`*  Supporting Procedure Declaration`	
`*/`	
`void get_next_buffer()`	
`{`	14 cycles
`  cur_rxbptr  = cur_rxbptr->next_rxbptr;`	40 cycles
`  last_rxb_flag = (cur_rxbptr->flag) & 1;`	48 cycles
`  validcount  = cur_rxbptr->dend - cur_rxbptr->dstart + 1;`	116 cycles
`  cur_byte_ptr  = cur_rxbptr->data + cur_rxbptr->dstart - 7;`	120 cycles
`}`	38 cycles

Table 12. C Program Trace for I Frame Decoding (con't)

C Program Trace	Exec. Time
`void get_next_byte(err_code)`	
`unsigned char err_code;`	
`{`	12 cycles
`  if (eopdu)`	
`  {  }`	26 cycles
`  cur_byte = *cur_byte_ptr;`	32 cycles
`  cur_byte_ptr++;`	24 cycles
`  validcount--;`	24 cycles
`  if (validcount == 0)`	26 cycles
`  {  }`	
`}`	16 cycles
`void init_var(rxbptr, tkbptr)`	
`struct rxbuffer *rxbptr;`	
`struct tkbuffer *tkbptr;`	
`{`	46 cycles
`  eopdu = 0;`	20 cycles
`  cur_rxbptr = rxbptr;`	16 cycles
`  cur_tkbptr = tkbptr;`	16 cycles
`  last_rxb_flag = (cur_rxbptr->flag) & 1;`	52 cycles
`  validcount  = cur_rxbptr->dend - cur_rxbptr->dstart + 1;`	112 cycles
`  cur_byte_ptr  = cur_rxbptr->data + cur_rxbptr->dstart - 7;`	108 cycles
`  while ((validcount == 0) && (!eopdu))`	36 cycles
`  {  }`	
`}`	28 cycles
`int get_remain_len()`	
`{`	14 cycles
`  int count;`	
`  count = validcount;`	16 cycles
`  while (!last_rxb_flag)`	36 cycles
`  {`	
`    get_next_buffer();`	18 cycles
`    count += validcount;`	18 cycles
`  }`	
`  return(count);`	20 cycles
`}`	16 cycles

Table 12. C Program Trace for I Frame Decoding (con't)

C Program Trace	Exec. Time
/*	
* X.25 Link Layer Decoding Procedure	
*/	
void slp_decode(rxbptr, tkbptr, cntxt)	
unsigned char cntxt;	
struct rxbuffer *rxbptr;	
struct tkbuffer *tkbptr;	
{	92 cycles
unsigned char temp;	
init_var(rxbptr, tkbptr);	46 cycles
cur_tkbptr->error = 1;	40 cycles
get_next_byte(2);	38 cycles
cur_tkbptr->addr = cur_byte;	36 cycles
get_next_byte(6);	38 cycles
cur_tkbptr->pf = (cur_byte >> 4) & 1;	70 cycles
if (!(cur_byte & 1))	38 cycles
{	
cur_tkbptr->type = 8;	28 cycles
if (cntxt & 1)	16 cycles
{ }	
else	
{	
cur_tkbptr->ns = (cur_byte) >> 1) & 0x07;	68 cycles
cur_tkbptr->nr = (cur_byte) >> 5) & 0x07;	76 cycles
}	
cur_tkbptr->info_ptr = cur_byte_ptr;	52 cycles
cur_tkbptr->info_len = get_remain_len();	38 cycles
}	
else	
{ }	
}	52 cycles

4.2.3. System Parameters Analysis

As mentioned previously, the system memory response time can be slow. Thus, it is important to investigate how this affects the PDA performance. The same set of PDU types are being decoded in PDA with different system memory access time (SMAT) ranging from 3 to 12 clock cycles. In each decoding program, the execution times for the various PDU types using the same SMAT are being averaged. Then the incremental increase in execution time using a given system memory access time is calculated by the following equation:

$$I_n = T_n - T_{n-1}, \quad n = 4, 5, ..., 12$$

where I_n is the incremental increase in execution time using a SMAT of n clock cycles, and T_n is the execution time using a SMAT of n clock cycles.

The incremental increase in execution time versus the SMAT is plotted in Figure 16. All three decoding programs exhibit the same trend. When using a SMAT of four clock cycles, the increase in execution time is small. As the access time increases up to six clock cycles, there is a sharp increase in execution time. When the access time further increases beyond six clock cycles, the amount of increase in execution time gradually becomes a constant value. The shape of the graph can be explained by the following argument. When a fast SMAT is used, the system memory access can be processed quick enough to reduce the blocking of subsequent RxB or TkB related instructions. Hence, the effect on execution speed is small. As the SMAT increases, more time is required to handle the system memory access. However, the time between two subsequent RxB or TkB related instructions remains constant because this is the execution speed of other instruction types, which all have fixed execution time. Eventually, the SMAT is so slow that all RxB and TkB

related instructions are needed to wait for system memory service. At this point, a sharp increase in execution speed is expected. If the SMAT increases beyond this point, all RxB and TkB instructions will increase their waiting time by the SMAT increase. As a result, the increase in execution time becomes constant.

Another notable observation from Figure 16 is that the Transport program has the largest increase in execution time when the SMAT increases, while X.25 Link program has the smallest increase. This is because the Transport program executes more RxB and TkB related instructions.

In summary, when the SMAT increases, the decoding program execution time increases as well. The amount of increase is depending on two factors. The first factor is the number of other instruction types between two RxB or TkB related instructions. This number gives a measure of how long a system memory access can take before causing the subsequent RxB or TkB related instruction to be blocked. The second factor is the number of RxB and TkB instructions being executed. At large SMAT, the larger the number of RxB and TkB instructions being executed, the larger the increase in the execution time.

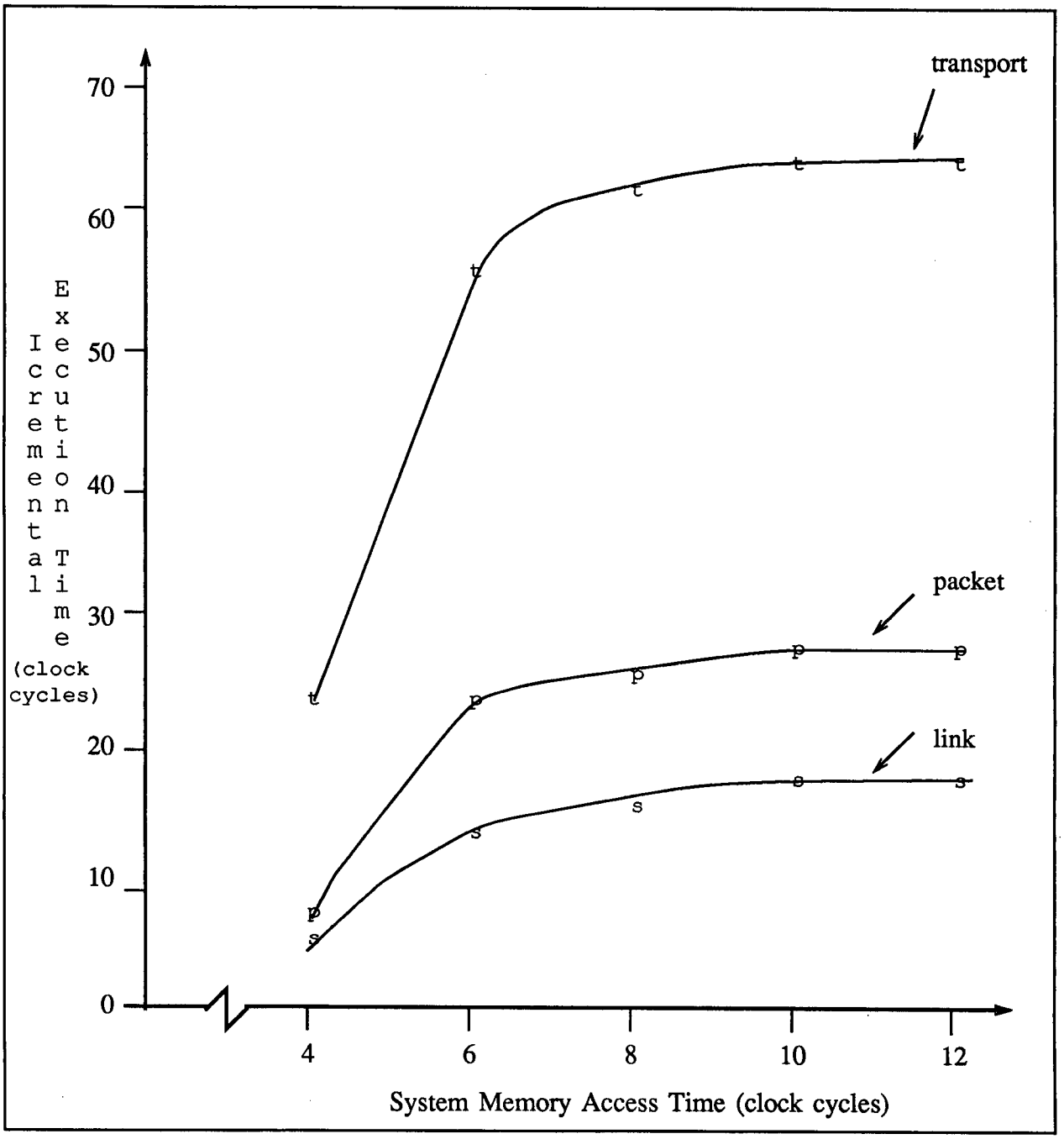


Figure 16. Incremental PDA Execution Time vs System Memory Access Time

The selection of the RxB element size is also important in the efficiency of decoding. As mentioned in previous chapter, the RxB element size can be configured by the system to have different size of 32, 64, 128, and 256. For PDU types that are small and always occupy only one RxB element, there is no performance difference in relation to the RxB element size. However, for those lengthy PDU types that span over several RxB elements, the larger the RxB element size, the less RxB element is required, and less overhead incurred to transverse the PDU. To illustrate this point, an X.25 I frame with data length of 400 bytes are stored in the RxB using the various RxB element sizes mentioned above. The execution speed of decoding these I frames are shown in Table 13. The result shows that there is almost a four times improvement in execution speed between using the smallest and largest RxB element size. Besides decoding execution speed, there are other factors involved in choosing the optimum RxB element size, such as, memory buffer efficiency [Smith-89].

Table 13. I Frame Decoding Execution Speed For Various RxB Size

	RxB Element Size (bytes)			
	32	**64**	**128**	**256**
X.25 I Frame	428 cycles	228 cycles	148 cycles	108 cycles

Chapter 5. Conclusion

5.1. Summary

In this thesis report, a specialized programmable hardware module called PDA is designed to handle PDU decoding for OSI protocols from Layer 2 to 4 in the architecture of a new generation portable protocol tester proposed by Davis et al [Davis-88]. From examining the PDU format and system environment, it has established that protocol decoding requires the following features:

- bit manipulation to handle non byte bounded field,
- comparison function to identify the PDU or parameter types,
- ability to skip over a specified number of data bytes,
- calculation of the remaining length of the PDU,
- arithmetic operation to check for possible length indicator discrepancy,
- internal register storage to hold any data in the PDU that may be needed later,
- able to accept additional parameters that are crucial to decoding, such as protocol option selection,
- support efficient access to the RxB link list structure, and
- capable of putting data value or its corresponding pointer into TkB.

Based on these requirements, the PDA is designed by using a small and simple instruction set with important functions being optimized in hardware implementation. The instruction set has a total of 26 instructions using a fixed 16 bit format. The hardware assisted functions include automatic RxB access, skip bytes function, PDU remaining length calculation, and bit extraction logic. Other salient features of the PDA design are listed in the following:

- separate program and system memory interfaces to allow parallel execution of instruction and system memory access,
- a three level deep RBFIFO to hold the prefetched RxB data,
- a TKHR to buffer the write request to TkB,
- the use of delay-by-one branch technique and instruction nullification option to provide an efficient branch implementation, and
- an internal register file that comprises of two 8 bit and fourteen 16 bit registers.

The PDA performance has been established by comparing with two other systems based on program size and program execution time measurement. One of the systems is essentially consisted of a 68000 microprocessor with no-wait-state memory. The other system has an ideal CPU that uses zero time to fetch and execute any instruction except for external memory reference. Using equivalent decoding programs for CCITT X.25 and X.224 protocols, it has been found that the 68000 program size is about five times the PDA equivalent. The smaller PDA program size reflects that the PDA has a more efficient instruction set and architecture for PDU decoding. The execution speed measurement also reinforces the PDA superiority over the 68000 system. Based on the above decoding programs, the average execution speed of the PDA is at least 16 times faster than the 68000 system. When comparing with the ideal CPU system, which represents the minimum achievable execution time, the PDA is only about four times slower.

The PDA performance under changes in system environment is also examined. Using the same decoding programs as mentioned above, it has been found that the incremental increase in execution time due to increase in system memory access time has the following pattern. At small system memory access time, the increase in execution time is small. As

the system memory access time increases up to a point, in which all system memory instructions are blocked, a sharp increase in execution time is observed. Beyond this point, the amount of incremental increase gradually becomes a constant value.

It has been found that the selection of RxB element size has a significant impact on the efficiency of PDU decoding. In decoding a special X.25 I frame, it has been observed that there is almost a four times improvement in execution speed between using the minimum (32 bytes) and the maximum (256 bytes) RxB element size.

5.2. Future Research

To justify the expensive manufacturing cost of the PDA design, it is important to investigate if the PDA is capable of handling the complexity of decoding higher layer OSI protocols. Preliminarily study has done on generating decoding programs for CCITT X.225 session layer protocol [CCITT-84C]. The study has suggested that it is feasible to apply the PDA design for decoding session PDUs. For the presentation and application layers, the abstract syntax notation one (ASN.1) is being used as a standard [CCITT-84d] to encode any data structure for transmission. ASN.1 is devised as an universal notation to represent any data types or structure in an unambiguous manner. It is felt that the current PDA design of using the TkB to save the result of decoding may not be adequate for representing the data structure retrieves from ASN.1 decoding. Much more research work is required to extend the current design to handle ASN.1 decoding. Recent interest in this area [Caneschi-87, Nakawaji-88] can provide a good start for this research.

Another area of interest would be designing a PDU encoder as a counterpart of the decoder. Encoding is deemed to be more straight forward because all the parameters are known and no error checking is necessary. However, other specific issues, such as memory allocation problem, require careful attention. Furthermore, optimization technique, such as header prediction [Clark-89], should also be investigated.

Bibliography

[Anderson-89] C. Anderson and M.R. Ito, "Architecture and Design of VLSI Hardware for Protocol Testing", IWPTS, International Workshop on Protocol Test Systems, October, 1989.

[Caneschi-87] F. Caneschi and E. Merelli, "An Architecture for an ASN.1 Encoder/Decoder", Computer Networks and ISDN System, Vol. 14, No. 2-5, 1987.

[Clark-89] D.D. Clark et al, "An Analysis of TCP Processing Overhead", IEEE Communications Magazine, June, 1989.

[CCITT-84a] CCITT Recommendation X.25, "Interface between Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) for Terminals Operating in the Packet Mode and connected to Public Data Networks by Dedicated Circuit", 1984.

[CCITT-84b] CCITT Recommendation X.224, "Transport Protocol Specification Open Systems Interconnection for CCITT Applications", 1984.

[CCITT-84c] CCITT Recommendation X.225, "Session Protocol Specification Open Systems Interconnection for CCITT Applications", 1984.

[CCITT-84d] CCITT Recommendation X.208, "Specification of Abstract Syntax Notation One (ASN.1)", 1984.

[CCITT-88] CCITT Recommendation X.200, "Reference Model of Open Systems Interconnection for CCITT Applications", 1988.

[Chan-89] R.I. Chan et al., "A Software Environment for OSI Protocol Testing Systems", International Symposium on Protocol Specification, Testing, and Verification, June 1989.

[Davis-88] W.B. Davis, "Architecture and Design of a Portable OSI Protocol Tester", invited paper, IWPTS, International Workshop on Protocol Test Systems, Vancouver, October, 1988.

[IDACOM-87] IDACOM Electronics Ltd., *MPT User Manual, MPT368.2 Multiport Network Analyzer*, November, 1987.

[Gross-82] T.R. Gross and J.L. Hennessy, "Optimizing Delayed Branches", Proceedings of Micro-15, October, 1982.

[Kernighan-78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[Krishnakumar-87] A.S. Krishnakumar, B. Krishnamurthy, and K. Sabnani, "Translation of Formal Protocol Specifications to VLSI Design", International Symposium on Protocol Specification, Testing, and Verification, June 1987.

[Krishnakumar-89] A.S. Krishnakumar and K. Sabnani, "VLSI Implementations of Communication Protocols - A Survey", IEEE Journal on Selected Areas in Communications, Vol. 7, No. 7, , September 1989.

[Mahon-86] M.J. Mahon et al., "Hewlett-Packard Precision Architecture: The Processor", Hewlett-Packard Journal, Vol. 37, No.8, August, 1986.

[Motorola-82] Motorola Inc., *MC68000 16-Bit Microprocessor User's Manual*, 3rd Edition, 1982.

[Nakawaji-88] T. Nakawaji et al, "Development and Evaluation of APRICOT", Proceedings, 2nd International Symposium on Interoperable Information System, Nov. 1988.

[OASYS-87] OASYS, Division of XEL, Inc., *OASYS Compiler Family User's Guide*, November, 1987.

[Patterson-80] D.A. Patterson and D.R. Ditzel, "The Case for the Reduced Instruction Set Computer", Computer Architecture News, Vol. 8, No. 6, October 1980.

[Patterson-81] D.A. Patterson and C.H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer", Proceedings, The 8th Annual Symposium on Computer Architecture , May 1981.

[Radin-83] G. Radin, "The 801 Minicomputer", IBM Journal of Research and Development, May 1983.

[Smith-89] B.R. Smith, "UBC/IDACOM OSI PT Environment Manual", UBC-IDACOM Technical Report, March 17, 1989.

[Svobodova-89] L. Svobodova, "Implementing OSI Systems", IEEE Journal on Selected

Areas in Communications, Vol. 7, No. 7, , September 1989.

[VLSI-88a] VLSI Technology, Inc., *VGT Users Guide*, 1988.

[VLSI-88b] VLSI Technology, Inc., "VGT 200, 1.5 MICRON Gate Array Series", May 1988.

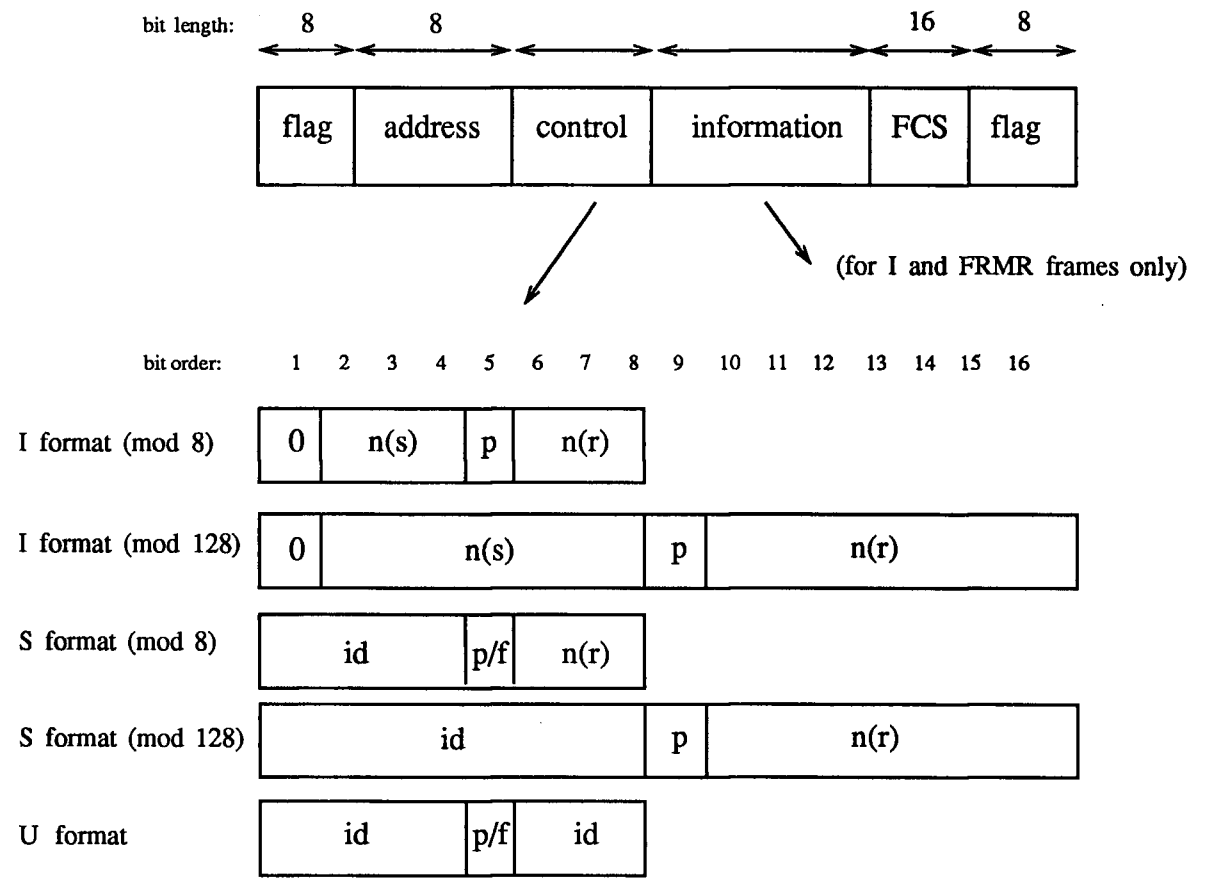
Appendix A. Acronyms

Following is a list of acronyms being used throughout this paper.

CCITT	International Telegraph and Telephone Consultative Committee
CCU	Central Control Unit, a submodule inside PDA.
DMI	Data Memory Interface, a submodule inside PDA.
ISO	International Organization for Standardization
LI	Length indicator in transport PDUs.
OSI	Open Systems Interconnection
PCI	Protocol Control Information
PDA	Protocol Decoding Accelerator
PDU	Protocol Data Unit
PMI	Program Memory Interface, a submodule inside PDA.
SMAT	System Memory Access Time
SDU	Service Data Unit
RBFIFO	Receive Buffer FIFO, a submodule inside PDA.
RxB	Receive Buffer
TkB	Token Buffer
TKHR	Token Holding Register, a submodule inside PDA.

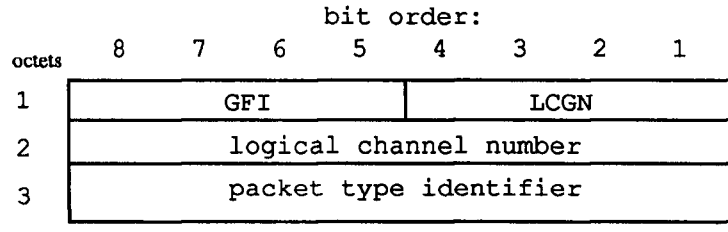
Appendix B. PDU Header Formats

Appendix B.1. X.25 Link Layer LAPB Frame Formats

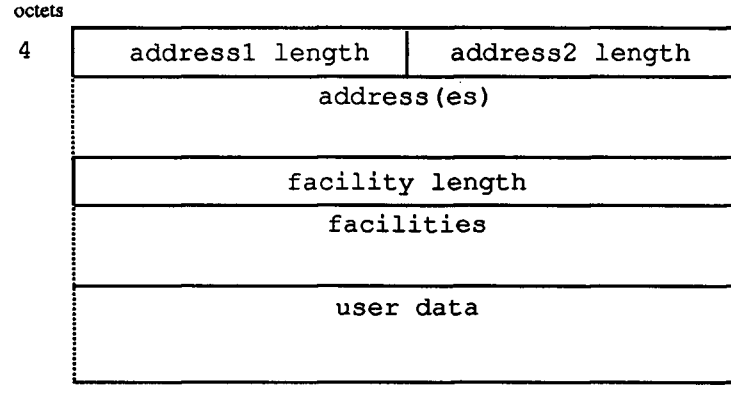


Appendix B.2. X.25 Packet Formats

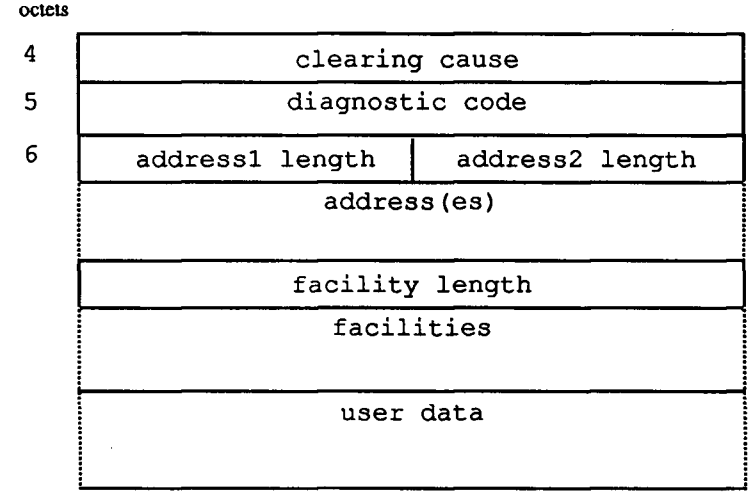
All packets use this format. All deviations are described in the following.



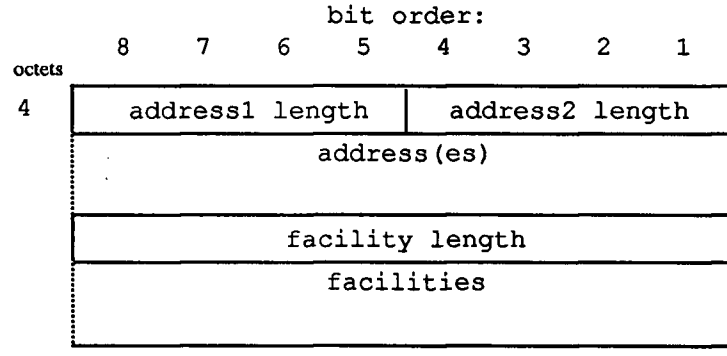
call request,
incoming call,
call accepted, and
call connected packets:



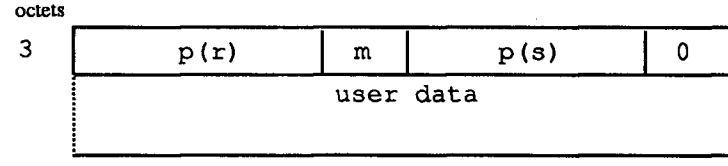
clear request, and
clear indication packets:



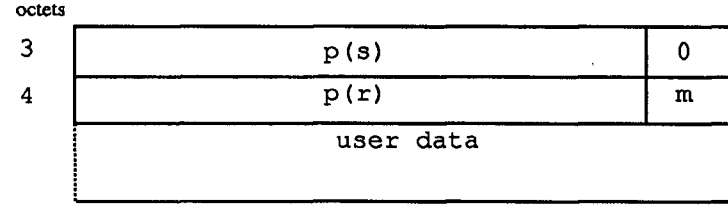
DTE and DCE clear confirmation packets:



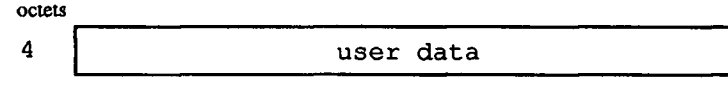
DTE and DCE data packets (mod 8):



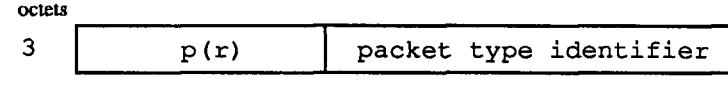
DTE and DCE data packets (mod 128):



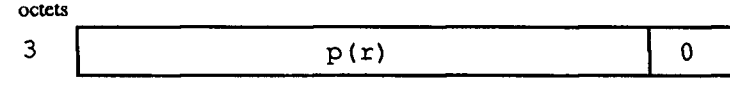
DTE and DCE interrupt packets:



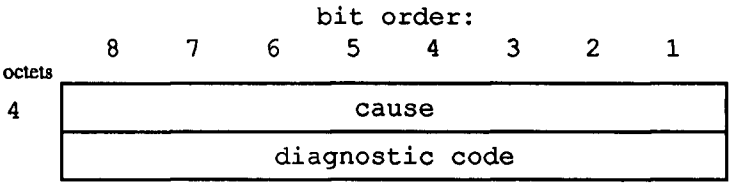
DTE and DCE RR,
DTE and DCE RNR, and
DTE and DCE REJ packets (mod 8):



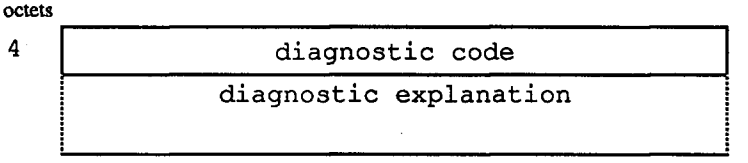
DTE and DCE RR,
DTE and DCE RNR, and
DTE and DCE REJ packets (mod 128):



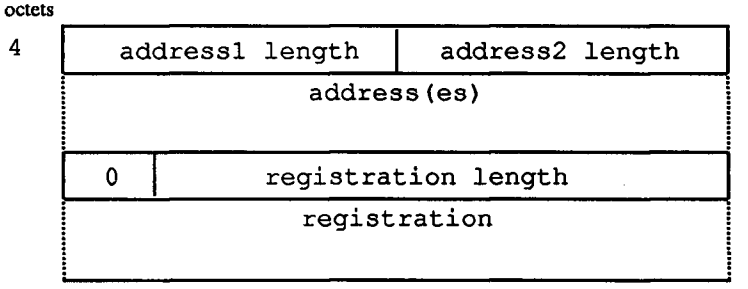
reset request,
reset indicaion,
restart request, and
restart indication packets:



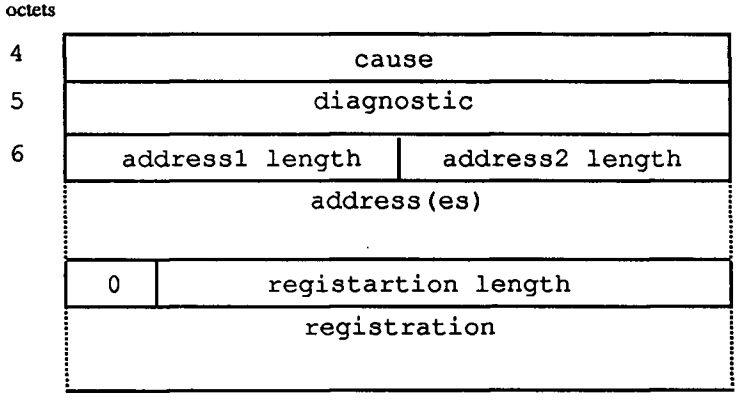
diagnostic packets:



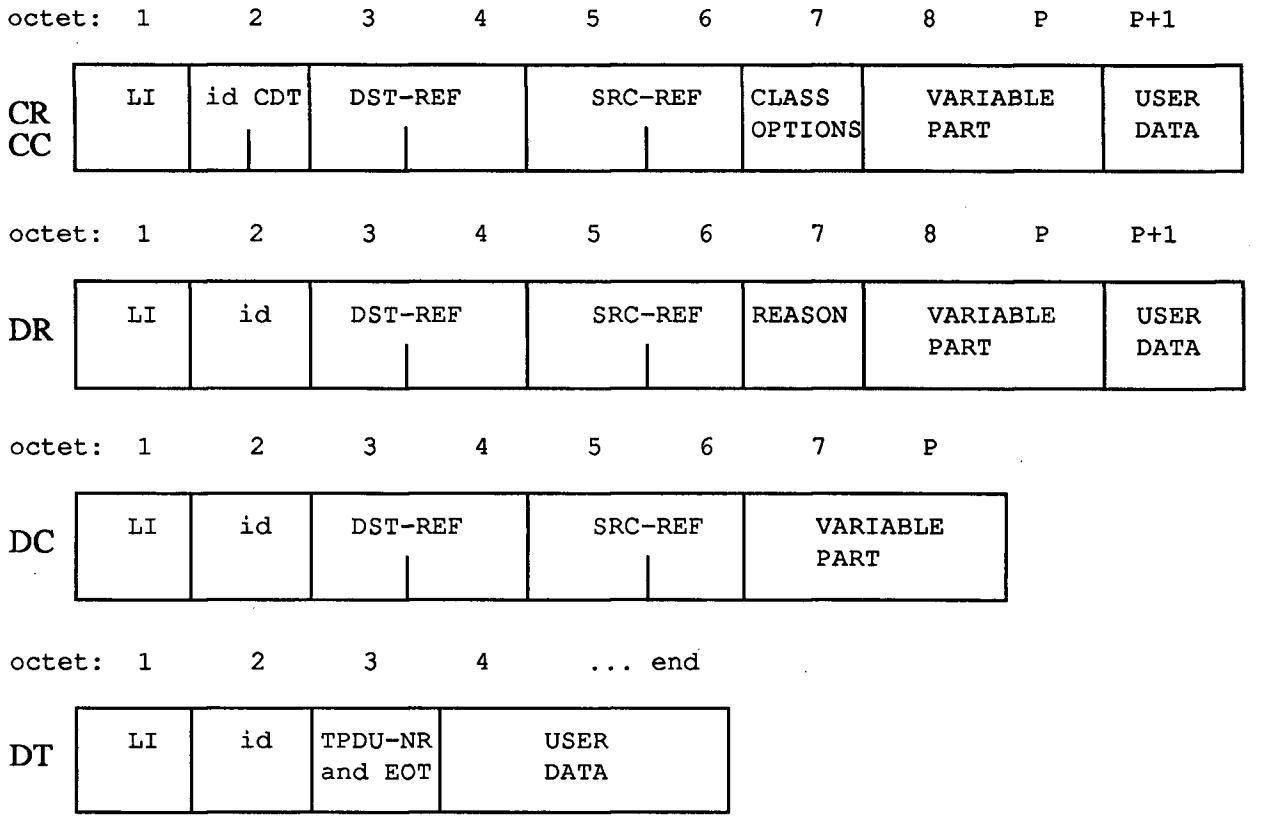
registration request packets:



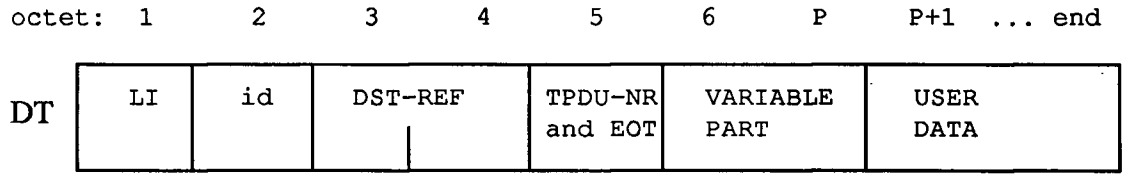
registration cofirmation packets:



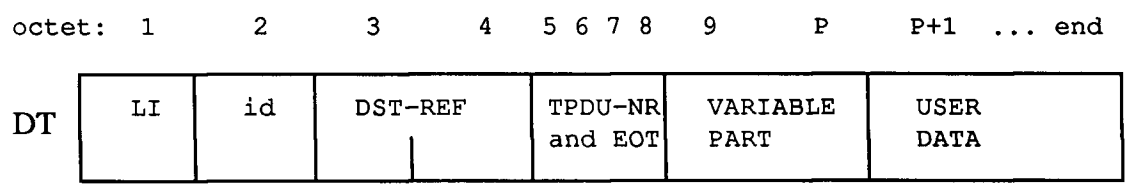
Appendix B.3. X.224 TPDU Formats



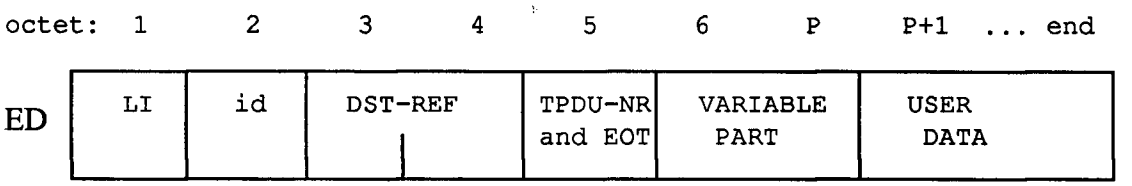
(a) normal format for class 0 and 1



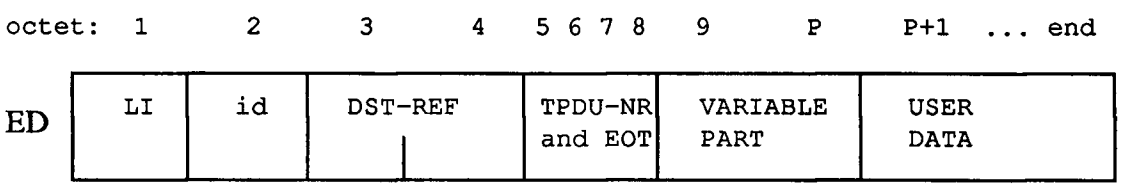
(b) normal format for class 2, 3, and 4



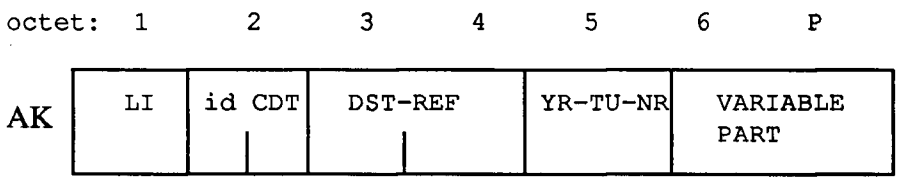
(c) extended format for class 2, 3, and 4



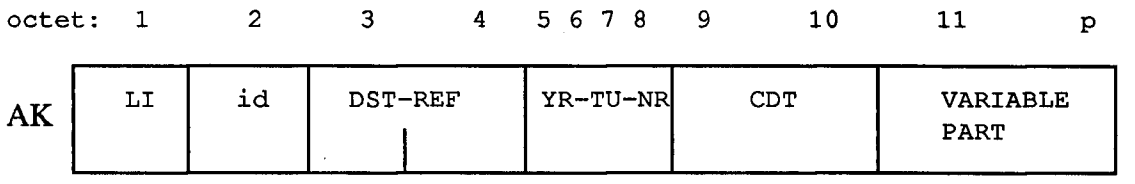
(b) normal format



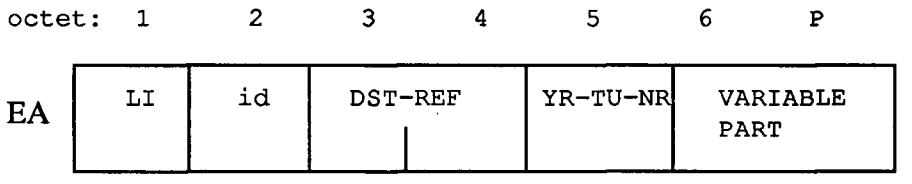
(b) extended format



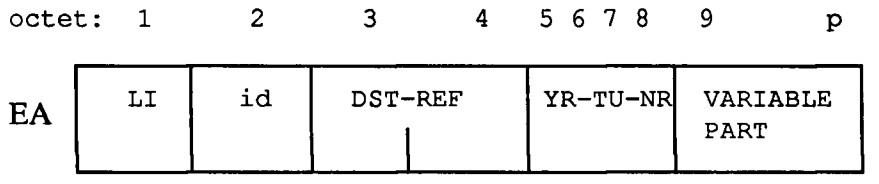
(a) normal format



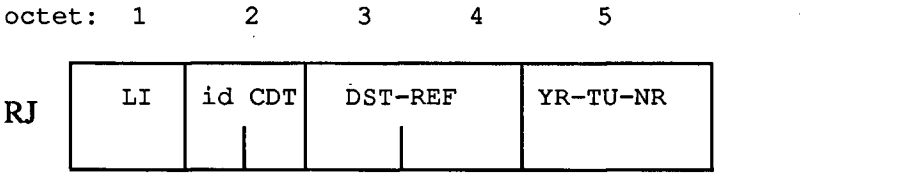
(b) extended format



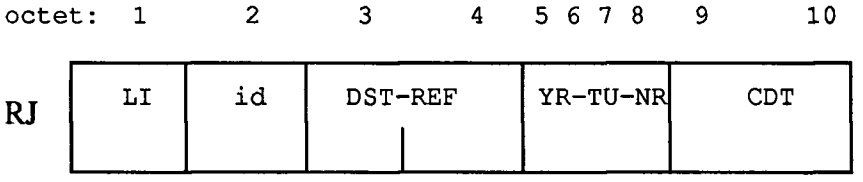
(a) normal format



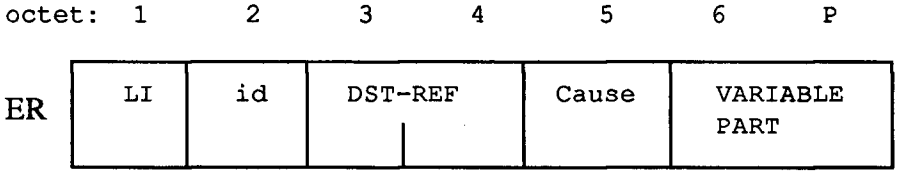
(b) extended format



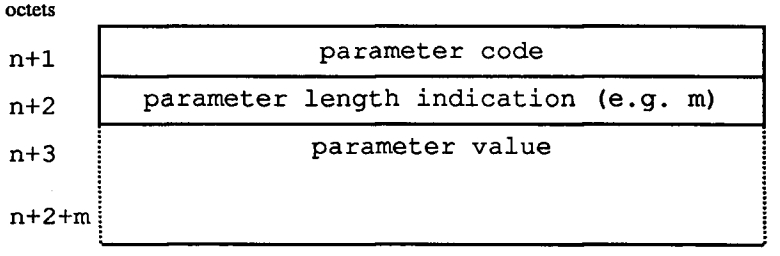
(a) normal format



(b) extended format



Variable part defines less frequently used parameters. Each parameters is structured as follow:



Appendix C. PDA Interface Signal Description

DADDR[23:0] is a 24 bit data memory address bus. This bus can be configured either as output or input. In program execution or program loading mode, it is an output bus to address the system memory. In configuration mode, this is an input bus to address the PDA internal registers.

DDATA[15:0] is a 16 bit bidirect data memory data bus. In program execution mode, it is used to read data from RxB and output data to TkB. In program loading mode, it carries the data from the system memory to be loaded into the PDA program memory. In configuration mode, it provides communication between the system and the PDA internal registers.

DRD is the data memory read request signal. It can be configured either as output or input. In program execution or program loading mode, it is an output to request a read cycle from the system memory. In configuration mode, this signal is an input to request a read cycle from the PDA internal registers.

DWR is the data memory write request signal. It can be configured either as output or input. In program execution mode, it is an output to request a write cycle from the system memory. In program loading mode, this signal is an output and is set to 'LO'. In configuration mode, this signal is an input to request a write cycle from the PDA internal registers.

DWORD is the data memory request mode signal. When 'HI', it indicates the request is a word (16-bit) access, and when is 'LO', it indicates the request is a byte (8-bit) access. It can be configured either as output or input. In program execution mode, it is an output. In program loading mode, this signal is an output and is set to 'HI'. In configuration

mode, this signal is an input.

DACK is the data memory acknowledge signal. It can be configured either as output or input. In program execution or program loading mode, it is an input to indicate that the data memory access is completed. In configuration mode, this signal is an output to indicate the current access to the PDA internal register is completed.

MODE[1:0] is a two bit signal to control the operating mode of PDA. When MODE[1:0] is '00' or '11', the PDA is set to configuration mode, in which, the PDA internal registers can be accessible. When MODE[1:0] is '01', the PDA is set to program execution mode, in which the PDA will executes the program in the PDA program memory. When MODE[1:0] is '10', the PDA is set to program loading mode, in which, the PDA will act as a DMA controller to transfer the program from system memory to the PDA program memory.

MODEACK is the mode configuration acknowledge signal. When asserted, it indicates that the PDA has switched to the mode specified in MODE[1:0].

DONE is a signal to indicate that an end instruction is executed and the PDA has changed from program execution mode to configuration mode.

PADDR[15:0] is the 16 bit PDA program memory address bus.

PDATA[15:0] is the 16 bit PDA bidirect program memory data bus.

PRD is the PDA program memory read request signal.

PWR is the PDA program memory write request signal.

MSTCLK is the system clock to PDA. It is anticipated that the PDA will operate at 20 MHz.

RESETB is the master reset signal.

Appendix D. X.25 Link Layer PDA Decoding Program

The following is a PDA program for decoding X.25 link layer.

```
#include "slp.pda.h"
#include "extract.h"
#include "pdareg.h"
#include "slp.branch.h"

/* out of data exception routne */
SLP_OD:         out.b.ri        rr13L, slp_error_off
                end

/* main */
SLP:            out.b.ri        slp_ok, slp_error_off
                mov.b.ri        rr13L, slp_empty
                get.b           rrY                 /* process frame address */
                out.b.ri        rrY, slp_addr_off
                                                    /* process control field */
                mov.b.ri        rr13L, slp_no_control_field
                get.b           rrX
                mov.b.ri        rr1L, m1
                extract         0, 4, rr1L
                out.b.ri        rrY, slp_pf_off          /* process PF bit */

                extract         0, 0, rr1L            /* decode frame type */
                cmp.ri          rrY, 0
                jp.eq.si        SLP_I
                mov.b.ri        rr1L, m4
                extract         0, 0, rr1L
                cmp.ri          rrY, 1
                jp.eq.si        SLP_RR
                cmp.ri          rrY, 5
                jp.eq.si        SLP_RNR
                cmp.ri          rrY, 9
                jp.eq.si        SLP_REJ
                mov.b.ri        rr1L, g7m3
                extract         5, 0, rr1L
                cmp.ri          rrY, 1F
                jp.eq.si        SLP_SABM
                cmp.ri          rrY, 23
                jp.eq.si        SLP_DISC
                cmp.ri          rrY, 0F
                jp.eq.si        SLP_DM
                cmp.ri          rrY, 33
                jp.eq.si        SLP_UA
```

```
                cmp.ri          rrY, 47
                jp.eq.si        SLP_FRMR
                cmp.ri          rrY, 3F
                jp.eq.si        SLP_SABME

                out.b.ii        slp_invalid_control_field, slp_type_off
                end

SLP_I:          out.b.ii        slp_i, slp_type_off
                jp.cntxt0.si    SLP_I128
                mov.b.ri        rr1L, m3
                extract         0, 1, rr1L
                out.b.ri        rrY, slp_ns_off   /* process ns, nr fields */
                extract         0, 5, rr1L
                out.b.ri        rrY, slp_nr_off
SLP_I_CONT:     outptr.i        slp_info_ptr_off
                skip.end.i      slp_info_len_off
                end

SLP_I128:       mov.b.ri        rr1L, m7
                extract         0, 1, rr1L              /* process ns field */
                out.b.ri        rrY, slp_ns_off

                mov.b.ri        rr13L, slp_no_nr_byte  /* process nr field */
                get.b           rrX
                mov.b.ri        rr1L, m1
                extract         0, 0, rr1L                 /* process P bit */
                out.b.ri        rrY, slp_pf_off
                mov.b.ri        rr1L, m7
                extract         0, 1, rr1L
                jp.si           SLP_I_CONT

SLP_RR:         out.b.ii        slp_rr, slp_type_off
SLP_NR:         jp.cntxt0.si    SLP_NR128

                mov.b.ri        rr1L, m3
                extract         0, 5, rr1L              /* process nr field */
SLP_NR_OTHER:   out.b.ri        rrY, slp_nr_off
                end.ep
                out.b.ii        slp_too_long, slp_error_off
                end

SLP_NR128:      mov.b.ri        rr1L, m4
                                        /* check the rest of control field */
                extract         0, 4, rr1L
                cmp.ri          rrY, 0
                jp.eq.si        SLP_NR128_CONT
                out.b.ii        slp_invalid_control_field, slp_type_off
```

```
                end
                                                    /* process P/F, nr fields */
SLP_NR128_CONT: mov.b.ri        rr13L, slp_no_nr_byte
                get.b           rrX
                mov.b.ri        rr1L, m1
                extract         0, 0, rr1L
                out.b.ri        rrY, slp_pf_off
                mov.b.ri        rr1L, m7
                extract         0, 1, rr1L
                jp.si           SLP_NR_OTHER

SLP_RNR:        out.b.ii        slp_rnr, slp_type_off
                jp.si           SLP_NR

SLP_REJ:        out.b.ii        slp_rej, slp_type_off
                jp.si           SLP_NR

SLP_SABM:       out.b.ii        slp_sabm, slp_type_off
SLP_UEND:       end.ep
                out.b.ii        slp_too_long, slp_type_off
                end

SLP_SABME:      out.b.ii        slp_sabme, slp_type_off
                jp.si           SLP_UEND

SLP_DISC:       out.b.ii        slp_disc, slp_type_off
                jp.si           SLP_UEND

SLP_DM:         out.b.ii        slp_dm, slp_type_off
                jp.si           SLP_UEND

SLP_UA:         out.b.ii        slp_ua, slp_type_off
                jp.si           SLP_UEND

SLP_FRMR:       out.b.ii        slp_frmr, slp_type_off
                                   /* process rejected frame control field */
                mov.b.ri        rr13L, slp_frmr_no_cntl_field
                jp.cntxt0.si    SLP_FRM128
                get.b           rr12L
                out.b.ri        rr12L, slp_frmr_control_off
                                            /* process vs, c/r, vr fields */
                mov.b.ri        rr13L, slp_no_vs_byte
                get.b           rrX
                mov.b.ri        rr1L, m3
                extract         0, 1, m3
                out.b.ri        rrY, slp_vs_off
                extract         0, 5, m3
                out.b.ri        rrY, slp_vr_off
```

```
                mov.b.ri        rr1L, m1
                extract         0, 4, m1
                out.b.ri        rrY, slp_cr_off
                                                    /* process wxyz fields */
SLP_FRMR_CONT:  mov.b.ri        rr13L, slp_no_wxyz_byte
                get.b           rrX
                mov.b.ri        rr1L, m1
                extract         0, 0, rr1L
                out.b.ri        rrY, slp_w_off
                extract         0, 1, rr1L
                out.b.ri        rrY, slp_x_off
                extract         0, 2, rr1L
                out.b.ri        rrY, slp_y_off
                extract         0, 3, rr1L
                out.b.ri        rrY, slp_z_off
                jp.si           SLP_UEND

SLP_FRM128:     get.w           rr12
                                   /* process rejected frame control field */
                out.w.ri        rr12, slp_frmr_control_off

                mov.b.ri        rr13L, slp_no_vs_byte    /* process vs byte */
                get.b           rrX
                mov.b.ri        rr1L, m1
                extract         0, 0, rr1L
                cmp.ri          rrY, 0
                jp.neq.si       SLP_FRMR_MOD128_ERR
                mov.b.ri        rr1L, m7
                extract         0, 1, rr1L
                out.b.ri        rrY, slp_vs_off
                                                /* process c/r, vr fields */
                mov.b.ri        rr13L, slp_no_vr_byte
                mov.b.ri        rr1L, m1
                extract         0, 0, rr1L
                out.b.ri        rrY, slp_cr_off
                mov.b.ri        rr1L, m7
                extract         0, 1, rr1L
                out.b.ri        rrY, slp_vr_off
                jp.si           SLP_FRMR_CONT

SLP_FRMR_MOD128_ERR: out.b.ii   slp_invalid_vs_byte, slp_error_off
                end
```

Appendix E. X.25 Link Layer C Decoding Program

The following is a C program for decoding X.25 link layer.

```
/*
 *   Variable Declaration
*/
unsigned char eopdu, *cur_byte_ptr, error_value, last_rxb_flag, cur_byte;
unsigned char cntxt;
int  validcount;
unsigned int cur_word;

struct rxbuffer {
   struct rxbuffer *next_rxbptr;
   char                dstart;
   char                dend;
   unsigned char       flag;
   unsigned char       data[25];
} *cur_rxbptr;

struct rxbuffer rbuf[20];

struct tkbuffer {
   unsigned char type, error, addr, pf, nr, ns, vr, vs, cr, w, x, y, z;
   int  frmr_cntl, info_len;
   unsigned char *info_ptr;
};

struct tkbuffer *cur_tkbptr;
struct tkbuffer tbuf[20];

/*
 *   Supporting Procedure Declaration
*/
void get_next_buffer()
{
   cur_rxbptr    = cur_rxbptr->next_rxbptr;
   last_rxb_flag = (cur_rxbptr->flag) & 1;
   validcount    = cur_rxbptr->dend - cur_rxbptr->dstart + 1;
   cur_byte_ptr  = cur_rxbptr->data + cur_rxbptr->dstart - 7;
}

void get_next_byte(err_code)
unsigned char err_code;
{
   if (eopdu)
```

```
      cur_tkbptr->error = err_code;
   cur_byte = *cur_byte_ptr;
   cur_byte_ptr++;
   validcount--;
   if (validcount == 0) {
      if (last_rxb_flag)
         eopdu = 1;
      else
         get_next_buffer();
   }
}

void init_var(rxbptr, tkbptr)
struct rxbuffer *rxbptr;
struct tkbuffer *tkbptr;
{
   eopdu = 0;
   cur_rxbptr = rxbptr;
   cur_tkbptr = tkbptr;
   last_rxb_flag = (cur_rxbptr->flag) & 1;
   validcount    = cur_rxbptr->dend - cur_rxbptr->dstart + 1;
   cur_byte_ptr  = cur_rxbptr->data + cur_rxbptr->dstart - 7;
   while ((validcount == 0) && (!eopdu)) {
      if (last_rxb_flag)
         eopdu = 1;
      else
         get_next_buffer();
   }
}

int get_remain_len()
{
   int count;
   count = validcount;
   while (!last_rxb_flag) {
      get_next_buffer();
      count += validcount;
   }
   return(count);
}

/*
 * X.25 Link Layer Decoding Procedure
*/
void slp_decode(rxbptr, tkbptr, cntxt)
unsigned char cntxt;
struct rxbuffer *rxbptr;
struct tkbuffer *tkbptr;
```

```
{
   unsigned char temp;

   init_var(rxbptr, tkbptr);
   cur_tkbptr->error = 1;
   get_next_byte(2);
   cur_tkbptr->addr = cur_byte;
   get_next_byte(6);
   cur_tkbptr->pf = (cur_byte >> 4) & 1;

   if (!(cur_byte & 1)) {
      cur_tkbptr->type = 8;
      if (cntxt & 1) {
         cur_tkbptr->ns = (cur_byte >> 1) & 0x07f;
         get_next_byte(7);
         cur_tkbptr->pf = cur_byte & 1;
         cur_tkbptr->nr = (cur_byte >> 1) & 0x07f;
      }
      else {
         cur_tkbptr->ns = (cur_byte >> 1) & 0x07;
         cur_tkbptr->nr = (cur_byte >> 5) & 0x07;
      }
      cur_tkbptr->info_ptr = cur_byte_ptr;
      cur_tkbptr->info_len = get_remain_len();
   }
   else
   {
      temp = cur_byte & 0x0f;
      if ((temp == 1) || (temp == 5) || (temp == 9))
{

         if (temp == 1)       cur_tkbptr->type = 5;
         else if (temp == 5) cur_tkbptr->type = 6;
         else if (temp == 9) cur_tkbptr->type = 7;

         if (cntxt & 1)
            if (((cur_byte >> 4) & 0x0f) == 0) {
               get_next_byte(7);
               cur_tkbptr->pf = cur_byte & 1;
               cur_tkbptr->nr = (cur_byte >> 1) & 0x07f;
               if (validcount)
                  cur_tkbptr->error = 3;
            }
            else
               cur_tkbptr->type = 4;
         else {
            cur_tkbptr->nr = (cur_byte >> 5) & 0x07;
            if (validcount)
```

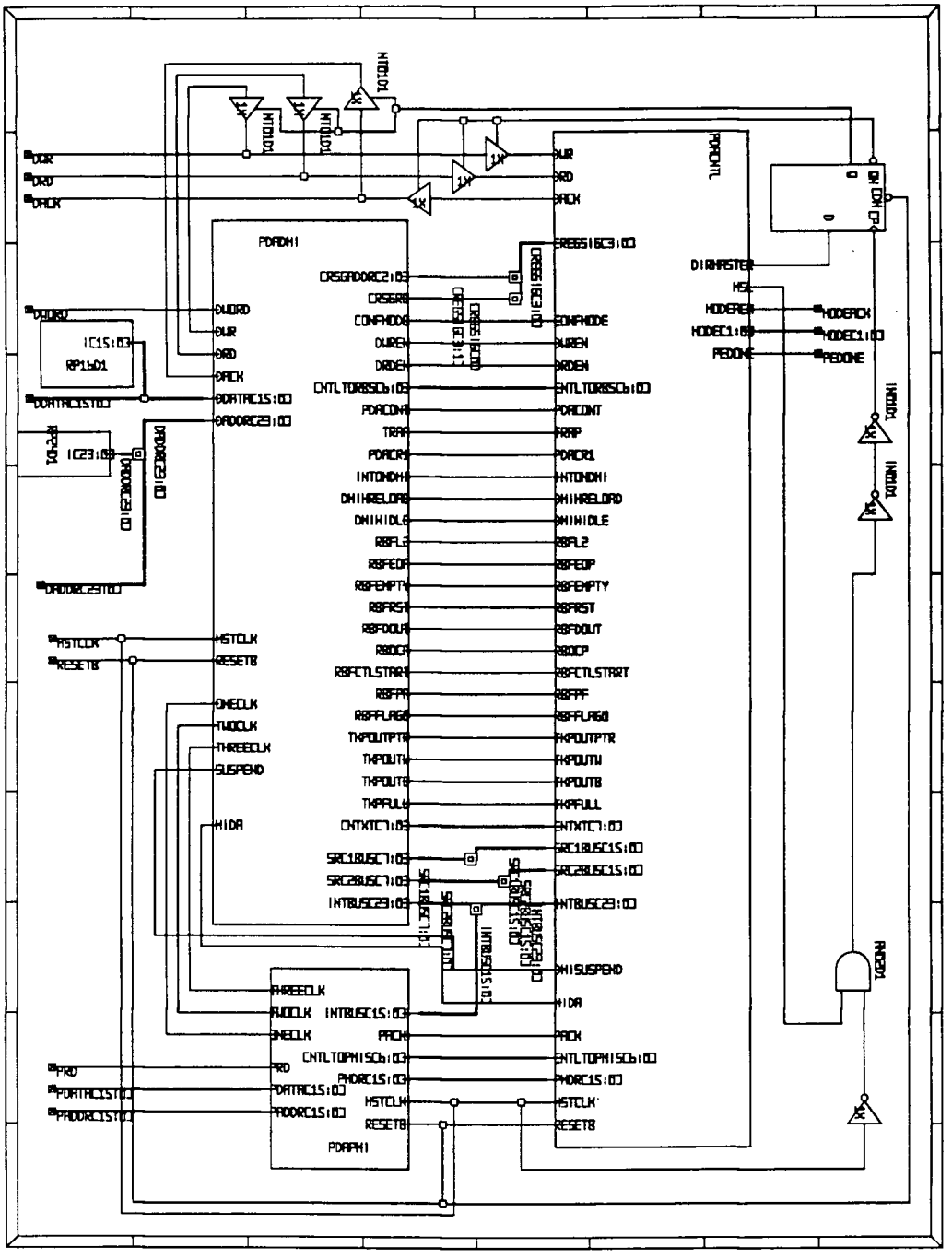
```
                cur_tkbptr->error = 3;
            }
        }
        else
        {
            temp = cur_byte & 0x0ef;
            if (temp == 0x2f) {
                cur_tkbptr->type = 0;
                if (validcount)
                    cur_tkbptr->error = 3;
            }
            else if (temp == 0x043) {
                cur_tkbptr->type = 1;
                if (validcount)
                    cur_tkbptr->error = 3;
            }
            else if (temp == 0x00f) {
                cur_tkbptr->type = 3;
                if (validcount)
                    cur_tkbptr->error = 3;
            }
            else if (temp == 0x063) {
                cur_tkbptr->type = 2;
                if (validcount)
                    cur_tkbptr->error = 3;
            }
            else if (temp == 0x87) {
                cur_tkbptr->type = 4;
                get_next_byte(8);

                if (cntxt & 1) {
                    cur_tkbptr->frmr_cntl = ((int) cur_byte) << 8;
                    get_next_byte(8);
                    cur_tkbptr->frmr_cntl = cur_tkbptr->frmr_cntl | ((int)
cur_byte);
                    get_next_byte(9);
                    cur_tkbptr->vs = (cur_byte >> 1) & 0x07f;
                    if (cur_byte & 1)
                        cur_tkbptr->error = 5;
                    else {
                        get_next_byte(11);
                        cur_tkbptr->vr = (cur_byte >> 1) & 0x07f;
                        cur_tkbptr->cr = cur_byte & 0x01;
                        get_next_byte(10);
                        cur_tkbptr->w = cur_byte & 0x01;
                        cur_tkbptr->x = (cur_byte >> 1) & 0x01;
                        cur_tkbptr->y = (cur_byte >> 2) & 0x01;
                        cur_tkbptr->z = (cur_byte >> 3) & 0x01;
```

```
                if (validcount)
                    cur_tkbptr->error = 3;
            }
        }
        else {
            cur_tkbptr->frmr_cntl = (int) cur_byte;
            get_next_byte(9);
            cur_tkbptr->vs = (cur_byte >> 1) & 0x07;
            cur_tkbptr->vr = (cur_byte >> 5) & 0x07;
            cur_tkbptr->cr = (cur_byte >> 4) & 0x01;
            get_next_byte(10);
            cur_tkbptr->w = cur_byte & 0x01;
            cur_tkbptr->x = (cur_byte >> 1) & 0x01;
            cur_tkbptr->y = (cur_byte >> 2) & 0x01;
            cur_tkbptr->z = (cur_byte >> 3) & 0x01;
            if (validcount)
                cur_tkbptr->error = 3;
        }
    }
    else if (temp == 0x06f) {
        cur_tkbptr->type = 9;
        if (validcount)
            cur_tkbptr->error = 3;
    }
    else
        cur_tkbptr->type = 4;
  }
 }
}
```

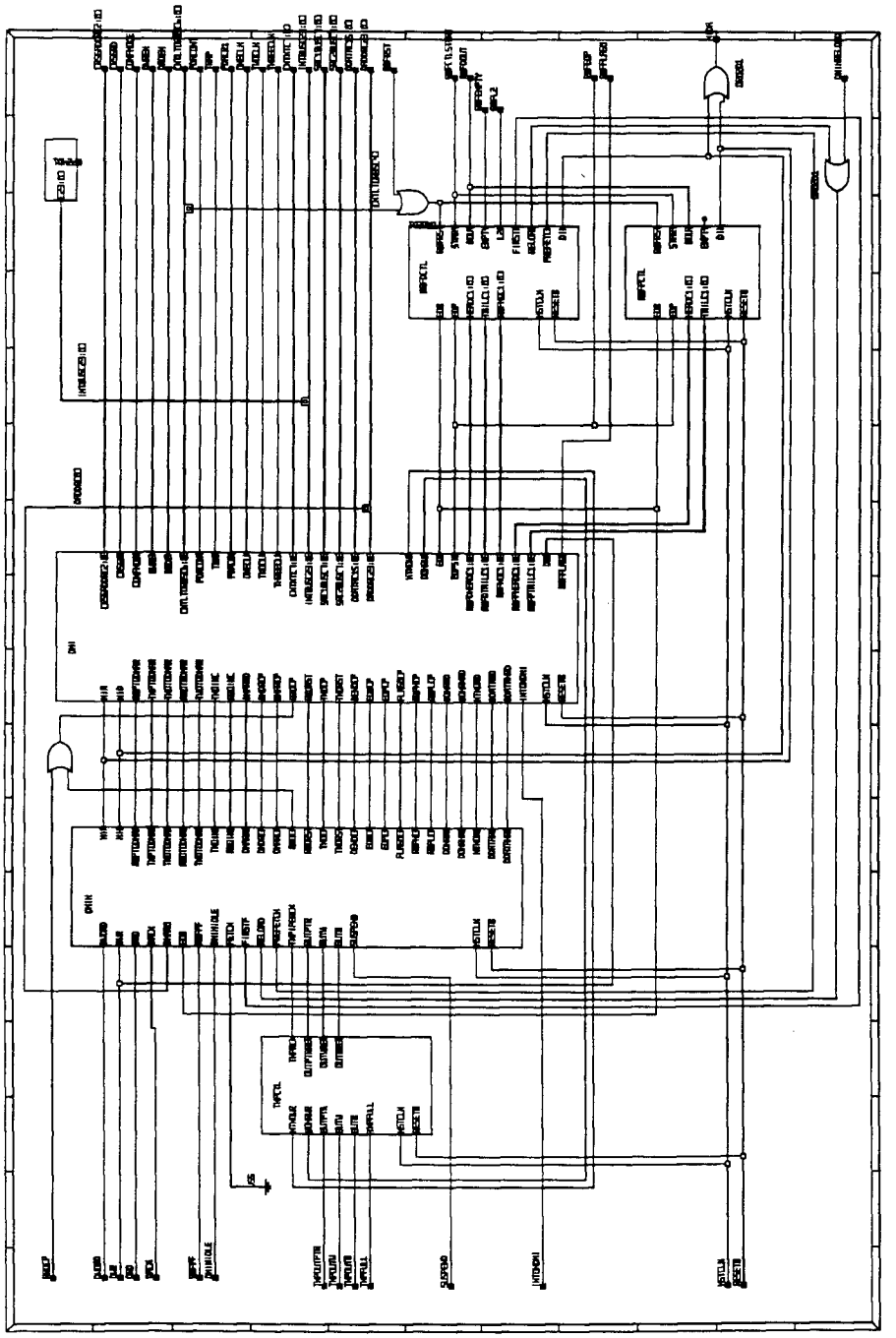
Appendix F. Schematics of the PDA Design

The following pages contain the schematics of the PDA design as implemented in the VLSI Technology's VGT200 series gate array technology design environment.

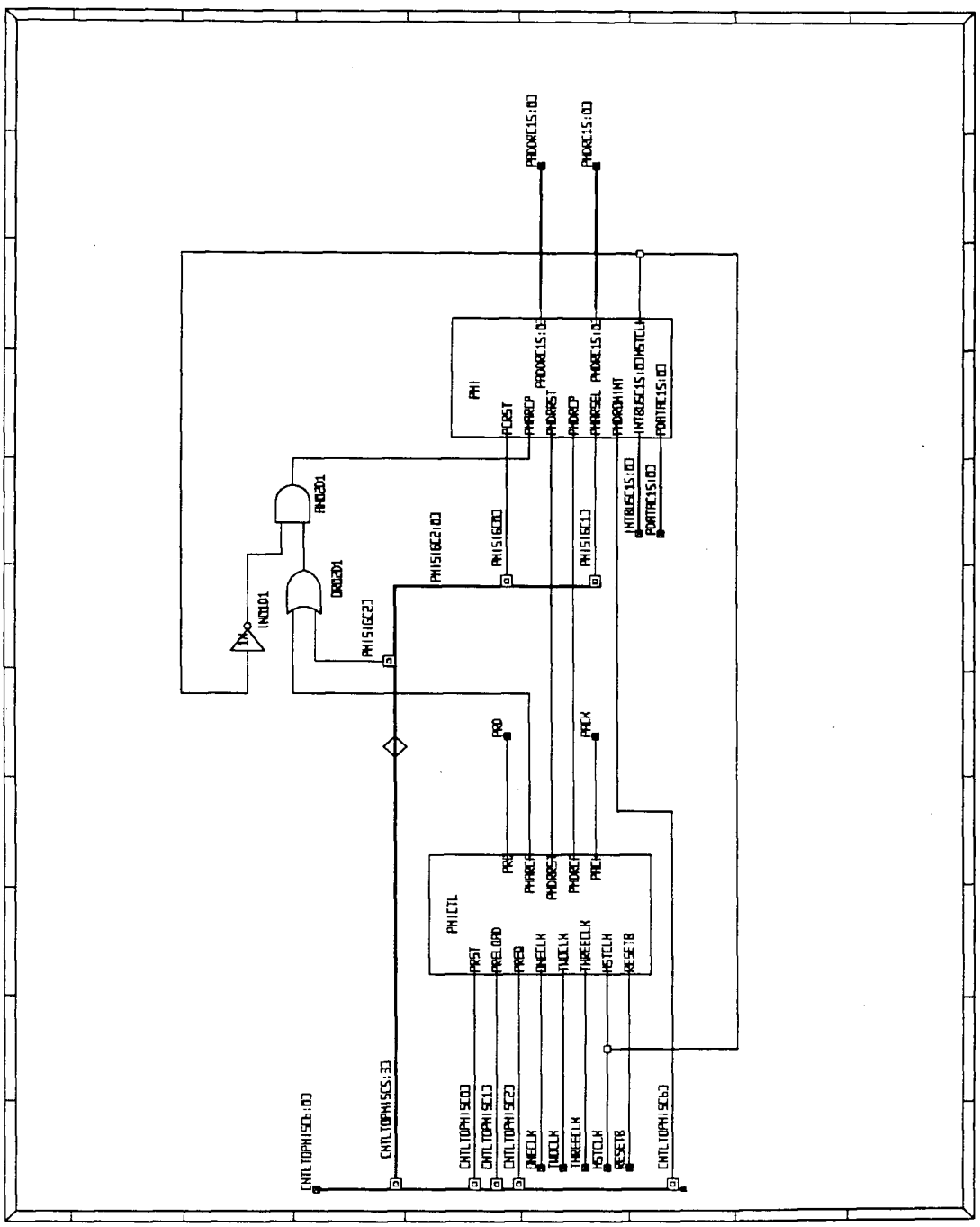
PDA :


PDADM1
PDAPM1
PDACNTL
DUR
DRD
DACK
DUDRD
DATRC1S1:0
DADRC2S1:0
HSTCLK
RESETB
ONECLK
TWOCLK
THREECLK
SUSPEND
HIDA
CNTXTC7:0
INTBUSC23:0
CONFMODE
DUREN
DRDEN
CNTLTOPHISC6:0
PDACNT
TRAP
PDACRL
INTONDMI
DHIHRELOAD
DHIHIDLE
RBFL2
RBFEOP
RBFEMPTY
RBFRST
RBFDOUT
RBDCP
RBFCTLSTART
RBFPF
RBFFLAGO
TXPOUTPTR
TXPOUTW
TXPOUTR
TXPFULL
DHISUSPEND
PACK
PHDRC1S1:0
PEDONE
HODERCK
HODEC1:0
INTDIO1

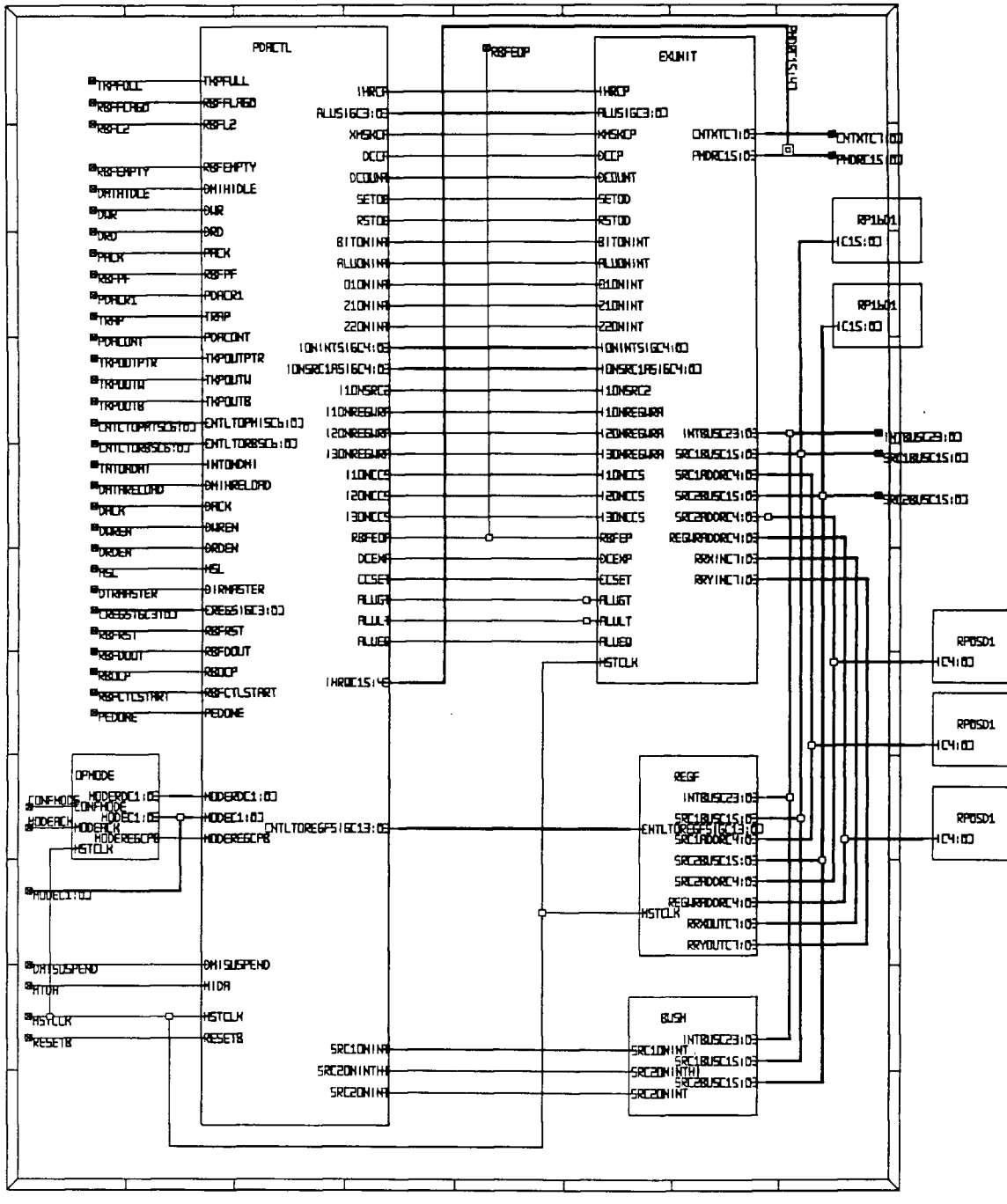
PDA.PDADMI:



PDA.PDAPMI:


PMI
PMICTL
AND201
OR201
IND101
PMISIG[2:0]
PMISIG[2]
PMISIG[0]
PMISIG[1]
PRD
PWR
CNTLTOPMISC[3:0]
CNTLTOPMISC[3:3]
PRST
PRELOAD
PRED
ONECLK
TWOCLK
THREECLK
HSTCLK
RESETB

PDA.PDACNTL:


PDACTL
EXUNIT
OPMODE
REGF
BUSH
RP1x01
RP8SD1
TXFFULL
RBFFLAGD
RBFL2
RBFEMPTY
DATAIDLE
DWR
DRD
PACK
RBFPF
PDACR1
TRAP
PDACONT
TXPOUTPTR
TXPOUTW
TXPOUTB
DATARECORD
DACK
DOREN
DOREN
MSL
DIRMASTER
RBFRST
RBFDOUT
RBDCP
RBFCTLSTART
PEDONE
MODEC1:0
HSTCLK
RESETB
DATSUSPEND
HIDA
SRC1ONINT
SRC2ONINTH
SRC2ONINT
CNTXTC7:0
PNDRC15:0
INTBUSC23:0
SRC1BUSC15:0
SRC2BUSC15:0
SRC1ADDRC4:0
SRC2ADDRC4:0
REGURADDRC4:0
RBXINC7:0
RBYINC7:0
RBXOUTC7:0
RBYOUTC7:0
C15:0
C4:0