RISC Architecture Enhancement for Data Communications Applications

Ali M. Elkateeb

A Thesis

in

The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

May 1992

ISBN 0-315 80019 1

ABSTRACT

RISC Architecture Enhancement for Data Communications Applications

Ali Elkateeb, Ph.D.
Concordia University, 1991

Efficient RISC architecture for real-time multi-tasking applications in a data communication environment has not been yet rigorously investigated. The aim of our work is to study the suitability of RISC architectures in a data communication environment, such as Integrated Service Digital Network (ISDN), and techniques to improve their performance. Since a register set structure has a big role in the RISC performance enhancement, the RISC register set structure will be the main focus of our research.

At first, a dynamic scanning approach is used to examine the software characteristics of ISDN protocols and their effect on the RISC performance. Then, a new approach is proposed to evaluate the state-swapping processing overhead under different load conditions in an ISDN environment. This approach offers a simple and yet accurate mean to estimate the state-swapping overhead. It is found that the state-swapping overhead is significant in ISDN processing. A RISC architecture using the Multiple Register Set (MRS) structure is introduced to reduce this overhead. The performance of the MRS-based RISC architecture is evaluated in terms of the size of the MRS, the number of tasks resident on the external memory, and the frequency of their execution. A priority strategy for distributing tasks' states among the MRS is proposed, and its implementation complexity is investigated. It is found that even if the number of tasks existing in the external memory increases, the proposed priority strategy can still maintain adequate performance improvement. Finally, the implementation of the priority strategy in conjunction with MRS structure for ISDN applications is discussed.

ACKNOWLEDGEMENTS

I wish to express my sincerest gratitude to my thesis supervisor, Dr. Tho Le-Ngoc, for his caring guidance, encouragement and patience throughout my research. He was always available to direct and to advise whenever the occasion arose. He has given me unlimited support towards refining my research ideas and developing a broad knowledge in all the related areas to my research. I would like also to acknowledge the financial support provided by Canada NSERC Grants and Quebec FCAR Grant to Dr. T. Le-Ngoc.

I am grateful for Dr. A. Al-Khalili for advising me during my course work. His encouragement and continuous support has helped my progress in the Ph.D program during the beginning of studies at Concordia University. Special thanks for Dr. Hamidreza Jamali, Faouzi Kamoun, Allan Dubeau and Ajas Hussein for their sincere assistance in reviewing of the thesis.

Special thanks are for my wife Shatha and my daughter Israa for their moral support and patience for not seeing me as often as they should.

Table of Contents

List of Figures

List of Tables

CHAPTER I

INTRODUCTION

1.1. Background

The trend towards digital communications has an impact on each type of transmission media and each type of switch used in communication networks. Transmissions over communications links are already heavily digital and will be totally so in the future. These communications links are intended for carrying the user information such as video, voice and data. The processors used within the nodes of these networks are to handle the received digital information and to perform the required communication nodes functions efficiently. The processed information will then passed to the required destination.

Optical fibers in today communications networks have provided a tremendously increased transmission bandwidth. This large transmission bandwidth has encouraged the support of more services and more users. Clearly, the increase in services and the number of users in a network has led to an increase in traffic volume. Consequently, the functions processed by the nodes would have to be performed at a faster rate and hence place a greater load on the processor within each node. For example, the Integrated Service Digital Network (ISDN), which has been proposed in last few years, allows more than one service to be processed simultaneously by the network. The terminals connected to ISDN switches will permit the subscribers to access various communication services through the use of a set of layered protocols that are defined by the I and Q series of the CCITT recommendations [1-4].

The Open System Interconnection (OSI) [20-24] model which was proposed in 1977 by the International Standards Organization (ISO), provides a solution to the problems of compatibility between computers from different manufacturers used in the same communication

network [25,26]. This model eliminates the problems found in information representation, communication techniques, and operating systems between computers used in the communication nodes [27]. Although the OSI model is comprised of a hierarchical seven-layered structure, only the first three layers are required for processing by the network nodes, i.e., physical, data link, and network layers. The ISDN user-network interface, Signalling System number 7 (SS7), X.25 are some examples of such protocols based on OSI model.

In general, the implementation of the first three layers functions can be done in both hardware and software. Layer 1 is always implemented by using hardware, part of the layer 2 functions can be implemented by hardware and the other part by software, while layer 3 is implemented by software. There are many VLSI chips which are used to perform the processing required by the hardware functions inside these layers [37,40,41,44]. Many general purpose processors have been used to support the processing required by the software functions of these three layers [47,54,57]. Designing a specialized processor to enhance the processing of these software functions has also been reported in small scale [49]. However, the specialized processors are not used widely because of their design complexity and high cost.

The latest development in the computer architecture has introduced a new approach in designing processors called the Reduced Instruction Set Computer (RISC) [69]. The reported success of RISCs as a high performance architecture has resulted in intensive research to investigate their performance, complexity, etc. However, the simple hardware design, the short development time and the high performance of the RISC-based processors would make the RISC architecture a promising one not only for general purpose computations but also for special purpose applications.

The work on RISCs has focused on the aspects related to RISC as a counterpart architecture to CISCs in general purpose computations. These researches examined the

instruction set from different directions, such as the justification of the choice of a certain instruction set; statistical programs measurements and evaluations; how the simple instructions were more utilized than the other complex instructions; and what their impact was on the code size; etc. Moreover, these researches also focused on the evaluation of register set organization and the number of these registers as well as their impact on the performance of this architecture. The performance measurements were evaluated by employing conventional benchmarks and by using simulation techniques. So far most of these studies have concentrated on the general purpose computations and have omitted studying this architecture for processing in specialized applications, such as for communications applications and specifically for the processing at the network nodes. The limited researches and studies for RISCs role in data communications network nodes processing along with the need for high efficient processors to cope with the demands to support the requirements speed of these applications, has directed this research towards investigating current RISC architectures with respect to their suitability and possible enhancements to supporting these requirements.

1.2. Thesis organization

The material presented in this thesis is organized into two parts. In the first part, data communications processing is investigated and emphasis is given to the network nodes processing. This part has two major objectives: first is to study the frequently used general operations in such processing, and the second is to highlight the current architecture of communications network nodes in different perspectives. These perspectives can comprise of node hardware structures, the processors used as well as their types, and the multiprocessor supports. Moreover, the investigation to the candidate processor architecture to be used in such processing is done by analyzing and comparing the available architectures, namely CISC

or RISC. The merits and drawbacks of the current RISC architecture are studied. It also focuses on the motivations in using the RISC concept to build high performance data communications processing architectures. Following this, we will also highlight the possible work directions to be investigated in this research.

The second part focuses on the selection of suitable RISC architectures, and to investigate their drawbacks in supporting data communications applications in which task switching operations occur frequently. In chapter 3 attention is focused on the characteristics of the data communications software, by using a RISC-based computer system. Examples of these characteristics are, High Level Language (HLL) statements frequencies, processor-memory traffic, and procedures nested depths. The results shown in this chapter are used in the following chapters to clarify the RISC architecture type which is more appropriate to this application. Chapter 4 investigates the impact of RISC architecture on increasing the task switching overhead by increasing the state-swapping cost. This has been studied from two perspectives: the register set used in RISC, and the use of optimizing compilers. The state-swapping cost is also measured for different data communications processing workloads. In chapter 5, the enhanced RISC architecture for this application is considered by adding Multiple Register Sets' (MRS) organization, viewing the impact of MRS on the performance enhancement of RISC, and by supporting MRS with a strategy to distribute tasks states to these register sets. The effect of this organization on RISC architecture is investigated along with the possible changes that we would include with RISC to incorporate this organization. Also derived and evaluated the numerical method used to measure the performance improvements, which can be achieved in comparison to the RISC architecture with a Single Register Set (SRS). The priority strategy is derived and investigated in order to determine if such a strategy could further increase the performance by using it with MRS and estimate its enhancements. Finally, RISC-based

architecture using MRS is highlighted with the necessary instruction set.

This work is concluded in chapter 6. Some final recommendations for the further work are also presented.

CHAPTER 2

PROCESSING IN COMMUNICATION NETWORK NODES

It is the intent of this chapter to develop background material related to the main aspects of computer architectures for data communications processing. This chapter attempts to investigate the following topics:

i. The processing features of a network node in data communications: This will help us to understand the nature and structure of both hardware and software implemented in these nodes.

ii. The technologies currently used to implement such applications in presently existing systems: This will review the status and possible drawbacks of the actual implementations using currently available technologies.

The outcome from the study of the above topics will derive the motivation for this work. It is also used to set the direction of our work and the general approach to reach our goal.

2.1. Processing Features

To find appropriate support to improve the processing performance in the network nodes, it is important to investigate the processing features. This section will study and analyze the processing features of data communications from three angles: their layer structure, instruction set, and software organization.

2.1.1. Layer Structure

The OSI model distributes the functions of the data communications processing in

seven layers: Physical, Data Link, Network, Transport, Session, Presentation, and Applications (Figure 2.1). A network node usually includes only the first three layers. Since in this work we emphasize on the network node processing, the attention will be only given to these first three layers.

Layer 1 (or Physical Layer) takes the responsibilities for the information transfer over the links. This includes the electrical requirements of a data communication channel, the physical interface between devices and the rules by which bits are passed from one to another. The data link layer (layer 2) attempts to make the physical link reliable and provides the means to activate, maintain and deactivate the link. The main service provided by the data link layer to the higher layers is that of error control and flow control. The network layer services are responsible for establishing, maintaining and terminating communications.

Many protocols based on OSI models have been developed. Although the general functions are similar, each protocol can include some variations. Thus, it is useful to consider one of these protocols as an example for clarify the discussions throughout this work. Since ISDN become very important in today's and future of data communications applications, the ISDN user-network interface protocol is chosen as an example in this thesis.

The ISDN protocols, which are specified by CCITT [1-4], has defined many procedures and capabilities within the first three layers. CCITT has recommended that the Basic Rate Interface (BRI), (2B+D), of layer 1 of the ISDN user-network run at 144 kbit/s, and the Primary Rate Interface (PRI) rate 1544 kbit/s (North America and Japan) and 2048 kbit/s (for Europe). Layers 2 and 3 support a wide range of service capabilities. The physical layer provides simultaneous, bidirectional transmission of information signals synchronized with the network. In the case of the basic access, the physical layer also enables orderly activation and deactivation and regulates simultaneous access by several terminals to the common D-channel. In PRI, this

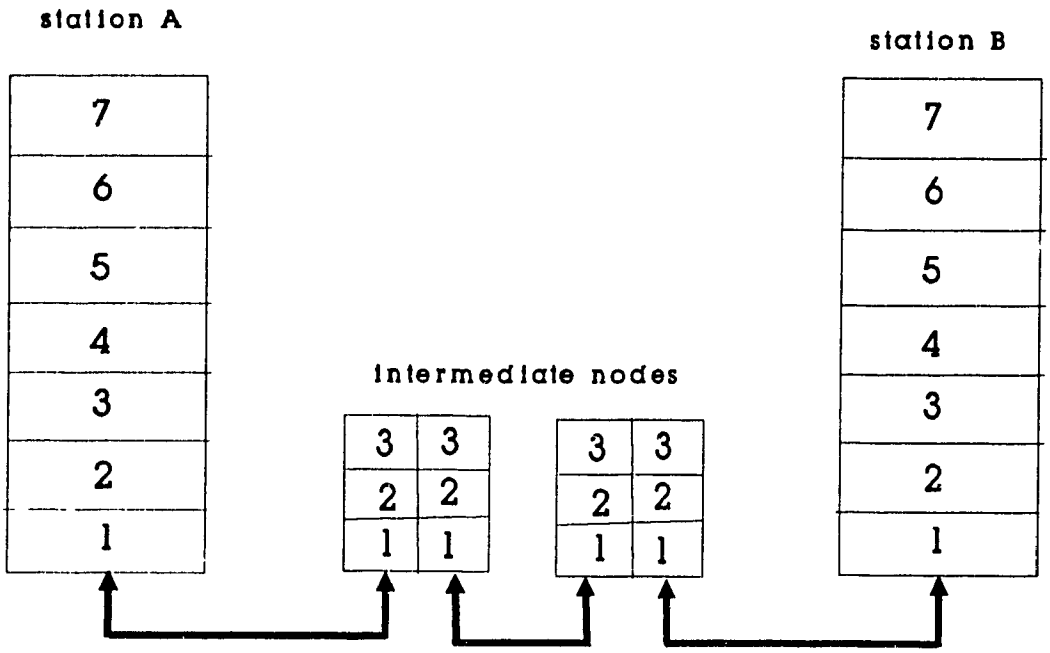


Figure 2.1 ISO Open Syatem Interconnection Reference Model

layer supports the point-to-point configuration, while no activation/deactivation capability is supported. Since this layer is always implemented in hardware, we will focus our attention towards the software aspects of layers 2 and 3.

It was intended by CCITT, that their recommendations for layer 2 and 3 should be applied to a variety of interface structures, such as basic access and primary access [81]. Layer 2 recommendations describe the link access procedure on D-channel, called LAPD. LAPD is on the basis of balanced mode HDLC such as X.25 LAPB. LAPD provides information transfer capabilities via point-to-point data links and via broadcast links. In order to handle multiple terminal installations at the user network interfaces as well as multiple layer 3 entities such as signalling and packet communication, LAPD has the capability of supporting multiple data link connections simultaneously on a single D-channel. In order to identify a specific data link connection, layer 2 address field consists of two subfields, namely, the data link layer Service Access Point Identifier (SAPI) subfield and the Terminal Endpoint Identifier (TEI) subfield. Layer 2 also provides two forms of information transfer services, either the Unacknowledged Information (UI) or the Acknowledged Information (AI) with multiple frame operation only. Layer 2 supports the TEI assignment procedures to automatically assign a TEI value to newly connected user terminal equipment at a specific user-network interface point. In this case no manual setting of a TEI value is necessary every time terminal equipment is connected to a user-network interface point.

Layer 3 recommendations describe the procedures for establishing and clearing of network connections such as circuit-switched connections using B-channel, user-to-user signalling connections using D-channel, packet-switched connections using either D-channel or B-channel. The main feature of this layer is the handling of the wide variety of connections as mentioned above through the same user network interface. This user network interface includes the

procedures for terminals operating in a stimulus mode to facilitate functional expansion for such terminals. The message structure of layer 3 consists mandatory and optional information elements to be discussed in more details in Chapter 3.

2.1.2. The Instruction Set

The use of suitable instructions for specific applications, such as data communications, will enhance the processing and make the processor more efficient than other processors designed for general purpose applications. In order to define the type and quantity of instructions within a set, it is necessary to study and analyze the software functions which are required to be executed by the processor. This study will help us to answer some important questions such as: Are there general purpose or special purpose instructions required with this set? Should the instructions be simple or complex? Are all instruction groups (e.g., logic, arithmetic, etc.) necessary to be supported or could these groups be reduced by eliminating some of them?

The objective of this section is to answer these questions, and in order to do so it is important to first understand the nature of the processing functions that the processor should perform in order to provide the required services.

The software processing part in the communication nodes usually deals with layer 3 and part of layer 2, as layer 1 and lower part of layer 2 are often implemented in hardware. Thus, the instructions used for layer 2 and 3 processing are the ones which define the type of instruction set required for the node processor. The instruction set used for these layers depends on the type of functions required to be handled by these layers. Hence, the study of these functions is important in determining the kind of required instruction set.

The amount of services and the complexity of these two layers become very large due to the different services which are required by different users, and these services are continuously

increasing. For example, the ISDN access nodes are required to support signalling, circuit-switched, packet-switched, and telemetry traffic. Software characteristics of the network node protocols can affect the instruction set in the following manners:

i. Programs implementing these protocols are large and complex, due to the large variety of required functions. For example, the implementation of layer 3 of the ISDN user-network interface protocol (Q.931) is done with about 70K of assembly code [29]. Required functions are of general purpose type such as searching databases, scanning messages, checking service availability, etc. Since the processing of these functions of the general purpose nature needs a processor to support such processing, the general purpose instructions are necessary to be supported in the processor used to handle the data communications functions. Also, note that complex numerical computations are not required.

ii. The processing of these protocols usually requires some of the functions to be processed repeatedly. These repeatedly used functions give the impression that there could be a need for special purpose instructions. The investigation of the nature and the processing of some of these repeatedly used functions will help to have a clear answer. Let us consider the following repeatedly used functions: scanning and wrapping / unwrapping messages. In scanning messages, all messages arriving at layer 2 or layer 3 were required to be scanned to determine the type of service required; identification of the called terminal; and to verify the correctness of the message fields [4]. These messages are partitioned into many fields, each of these fields is dedicated for a particular service. At the receiver, each field is tested, and this processing of message scanning can be realized by a sequence of Compare and Branch instructions. Thus, there is no need for special instructions and therefore the general purpose instructions can be used here. The

other special function, i.e., wrapping / unwrapping operation, is required when the messages pass between the layers, particularly between layers 2 and 3 (usually this is done in layer 1 by the hardware which strips and inserts the flags) in which layer 2 strips off the information which will not be used by layer 3. For instance, the message address part (consisting of two fields, Service Access Point Identifier or SAPI and Terminal Endpoint Identifier or TEI) and the control field within the LAPD frame format messages, in the ISDN user-network interface messages, are inserted back when a response message is received from layer 3. Similarly, this can occur when a signalling message in the ISDN user-network interface is passed on to layer 3 of the call control. If the destination user number is not in this local node, then the signalling message will be forwarded to another node via the Signalling System number 7 (SS7). In order for this operation to be carried out, the signalling message must be embedded within the SS7 message header format [30]. This wrapping/unwrapping operation can be processed by using general purpose instructions in which the wrapped information could be ORed with the original information, while the unwrapped could be handled by ANDing the unrequired information with logical zeros, i.e., masking operation.

It is clear from the above discussion that the instruction set to be supported by the node processor includes simple and basic general purpose instructions such as AND, OR, BRANCH, COMPARE, etc. Special instruction(s) can be useful for this application and enhance the processing performance. For instance, the NTT processor which is designed for data communications nodes processing is supported by some special instructions for the path scanning operation [49]. However, having some special instructions within the instruction set will not change the type of the instruction set from a general purpose one. Many processors with general

purpose instructions are used in such applications, e.g., Intel 80386, Motorola 68020, etc.

2.1.3. Software Organization

2.1.3.1. System Software

When different data communications services with different priorities are concurrently processed, it is important to have the means to process the high priority service first. For instance, if the processor executes functions of two different services such as packet data and telephone signalling, then the packet data processing can be interrupted when the voice service needs processing. Clearly, this happens because the voice service is considered to have a higher priority than the packet data service due to the delay-sensitive characteristics of the voice service. The processing of such services can be managed in different ways. In the first way, the processor can continuously look for the types of messages arriving at the node. This is a polling technique. Clearly, it will require to stop the current processing until the processor finishes the execution for the arrived message. Normally when a signalling message arrives during the processing of a data message, the processor will decide to momentarily stop the data message processing and serve the signalling message. However, if the processing of the data message is very short and does not seriously delay the processing of the signalling message, the processor will continue to serve the data message and store the signalling message in a buffer. When the node processes a signalling message and receives another signalling message, a buffer should also be used to store the arrived message until it reaches its turn for processing. The use of many services with different priorities and more messages types will make the software for managing the system operations very complex and reduce the processing efficiency.

The above mentioned polling technique is not efficient. Interrupt technique can be used to enhance the performance. Each of the services used can be implemented as an interrupt

service routine. However, it is still important to implement and manage the message buffers and to schedule the operation of such service routines based on their priorities. It is also important to give the services with low priority a chance to be processed when the high priority service has many messages in its buffer waiting to be processed. A possible synchronization technique can be implemented by allowing the process of one or few messages of low prioirty service from time to time when the high priority service processes certain number of messages. In addition, the processing of data communications protocols also requires responses and actions in real-time. For instance, if the signalling messages are sent to the other node and no response is received within a specific time interval, the transmitting node will be interrupted by the real-time clock and an appropriate action will be done by the node such as retransmitting the message.

In order to avoid the network node software designer from taking care of synchronization, scheduling, messages buffering, etc., a real-time multi-tasking operating system is normally used. This will simplify the task of developing software for the communication nodes and increase the processing performance. Such operating systems are designed to make the processor to be efficient and to give a quicker response to the tasks which require it most. Thus, the node software will be developed as a group of logically separated modules or tasks rather than a single program [27]. The Intel iRMX [36], Duplex Multi-Environment Real-Time Operating System (DMERT) [54], MCPOS [47], and CTRON [59, 56, 58] are few examples of such operating systems.

Generally, these operating systems suffer from two types of overhead: the messages passing and task switching. Practically, passing a long message between tasks is very time consuming and hence passing the pointers (or tokens) of these messages rather than passing the whole message itself greatly helps in reducing this overhead [35].

Task switching usually occurs due to the suspension of a running task, either deliberately

or voluntarily, to allow another task to run. Basically, tasks are switched during different events such as when an interrupt occurs and requests a task to be run, which preempts the currently running task if this next running task has a higher priority than the current one. Task switching also occurs when a high priority task which is in ready state waiting for an object and this object becomes available; then the task will run immediately if it has a higher priority than the currently running one. Moreover, a task is switched when its time slot or limit has expired, and the operating system will schedule the next task in the ready list to run. The processing overhead of this operation consumes some of the overall processing. Hence, reducing this overhead will improve the system performance.

Different possible ways can be used to reduce the task switching overhead. For instance, designing the tasks in an efficient manner, by keeping the number of tasks minimal and especially by binding the heavily related tasks which need a large amount of data exchange. However, this could have a negative impact on software being developed and modified by making it more complicated, time consuming and costly. In systems where tasks running time is short, it is possible to let every task run to the end, and hence no preemption scheduling is implemented [47]. Clearly, this will reduce the task switching frequency since a task will relinquish the processor only when it has completely finished, thereby not allowing any other task to preempt it. However, this is not applicable where different services are implemented with different priorities and the tasks durations are long.

2.1.3.2. Applications Software

Each of the protocol services can be processed as one or more tasks [27]. As the services provided by these protocols become larger in today's and future communications systems, the software represents the service within these layers will need to have more tasks

where each task will represent a service or some functions of a specific service. For example, in the present ISDN user-network interface protocol, layer 3 or Q.931 call control provides a basic and supplementary signalling service in addition to the User-to-User Information transfer (UUI) and the packet data services. Layer 2, on the other hand, includes the management functions which are called the " TEI assignment" in addition to other functions supported by this layer [31-34].

The distribution of the application tasks are basically dependent on the size of the nodes, i.e., the small size nodes have different task distributions than the medium and large size nodes. In small nodes where the traffic capacity is small, only one processor may be sufficient to manage the processing of the whole node software and to provide the users with the required service within a minimal processing time. The nodes software is basically organized in such a way that to improve the performance. Organized tasks differ from one node to another depending on the amount and type of services required. As the amount of these services increases, the tasks structure becomes more complex since each service might need more than one task. For example, in the ISDN access node with low traffic capacity, the distribution of the tasks can be managed as shown in Figure 2.2 (only the applications tasks are shown here). Depending on the supported services, some nodes might implement only a subset of that shown in Figure 2.2. For instance, in the ISDN point-to-multipoint subscriber radio system which organized as a star topology network the need for the SS7 support as an inter-signalling between the system nodes is not necessary, and can be substituted by a simple signalling system [5-10].

The concurrence nature of these asynchronous tasks have led to the use of mailboxes to transfer messages between each others. A task can be switched to get control of the processor when it is in ready state and has higher priority than other tasks. Otherwise, if tasks are organized in such a way that they have to be switched for each arriving message, the

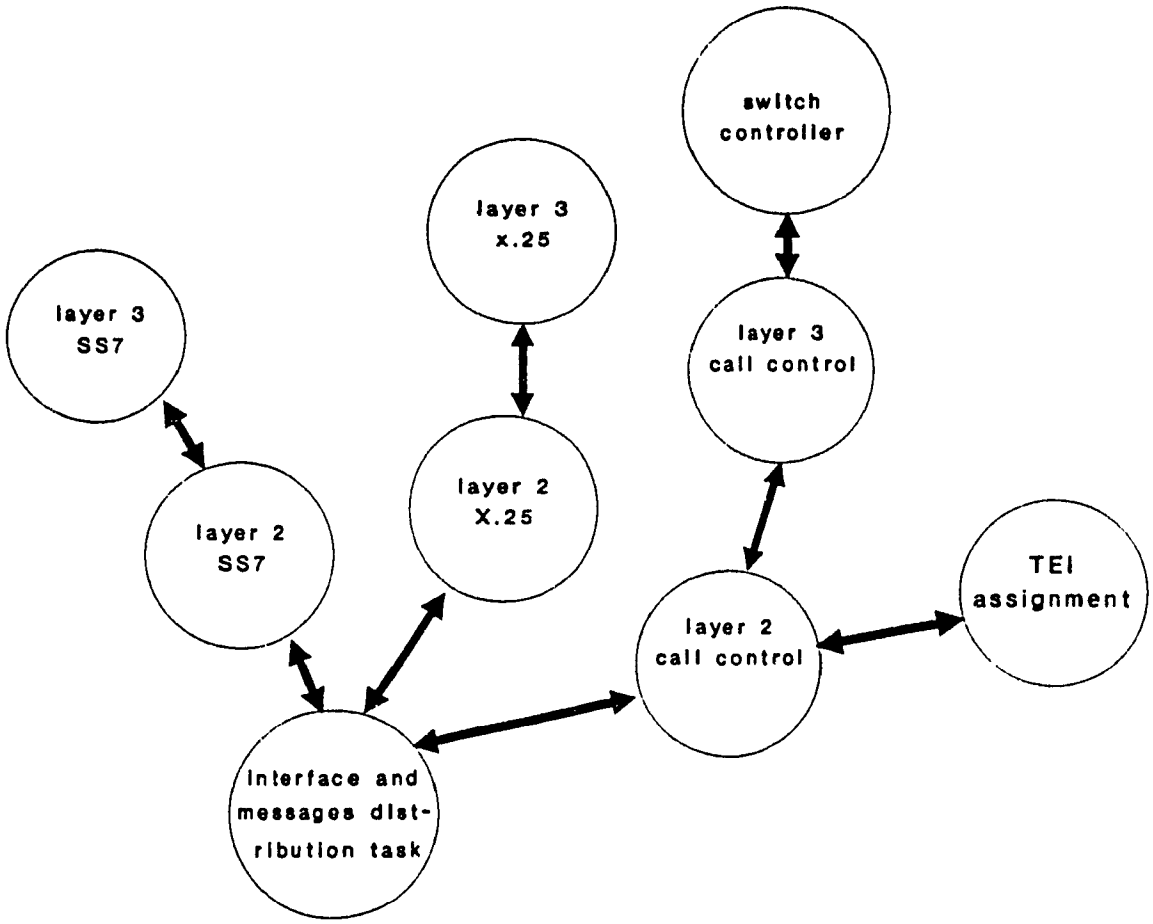
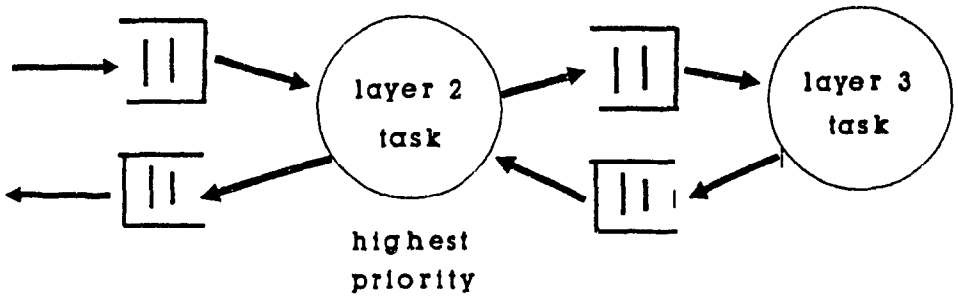


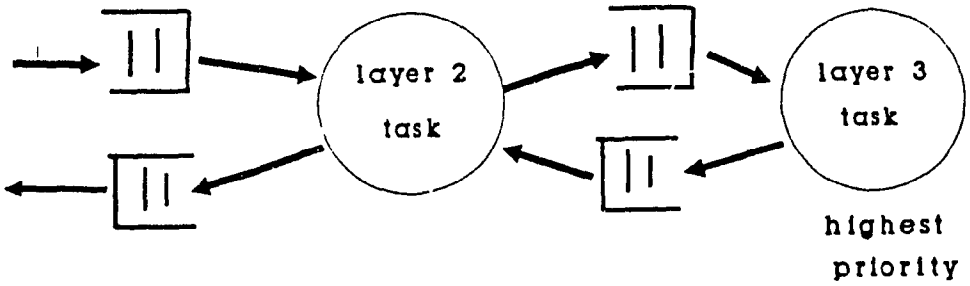
Figure 2.2 Possible Task Distribution in a Low-Capacity Network Node

communications processing overhead is increased and as a consequence the system throughput decreases. Thus, the communications and tasks processing should be balanced. Tasks switched with the arrival of each message makes the processing delay per message a minimum, however, the throughput of the processor will also be reduced due to the high task communication overhead. To illustrate the frequency of task switching in applications under heavy message passing between tasks, we can use a node with two application tasks representing layers 2 and 3 in two separate tasks respectively (Figure 2.3). When layer 2 has a higher priority than layer 3, layer 2 can process the messages in its input mailboxes until no more messages are left at which time task switching will occur allowing layer 3 to start processing (Figure 2.3a). Clearly, if the traffic to the node is high, then layer 3 will not be able to provide processing within an acceptable time limit unless the operating system intervenes to stop a layer 2 task and start a layer 3 task. In another possible tasks organization (Figure 2.3b) assuming that a layer 3 task has a higher priority than a layer 2 task, then each message received by the layer 2 task will be processed and passed to the mailbox of layer 3 task. The layer 3 task will then preempt the layer 2 task, process this message and return a response message (if necessary) to the layer 2 task mailbox. Therefore, with each incoming message to the node, there will be two tasks switched (one switched to layer 3 and one switched back to layer 2). This will increase the task switching overhead and reduce the processing throughput, but provide minimal message processing delay. Should the organization of these tasks be such as to process a few messages before they switch to another task, this will reduce the task switching cost at the expense of increasing the message processing delay. This will then allow each task to process a number of messages before task switching occurs.

In situations where the communication node becomes heavily loaded with traffic, it is not practical to keep all the communication node tasks to be processed confined to just one



A. Layer 2 with Highest Priority



B. Layer 3 with Highest Priority

Figure 2.3 Distribution of Layer 2 and 3 Tasks

processor. Processor costs have dropped drastically in the last decade, making the use of multiprocessor in supporting nodes processing feasible. Node tasks can be partitioned to run in different processors based on the services each provide, i.e. the tasks providing the call control of layer 3 can be separated into one processor, while the tasks used to implement packet data are separated into another processor and so on (Figure 2.4). The distribution of tasks in this way accommodates the adaptation of new services in the future, hence making this a suitable solution for the most recent data communication nodes, i.e. such as ISDN nodes. Task switching overhead associated with the processing of such nodes is the same as the one discussed before for single processor systems. In addition the inter-processor message transferring introduces another overhead. Tasks distribution for multi-processor systems should be configured in such a way as to reduce this overhead. One such method of reducing this overhead is by allocating tasks of one service to the same processor [46]. This will minimize the message transfers between processors thereby reducing this overhead as well as a potential bus bottleneck. In such a configuration inter-processor message transfers only occur when a need arises to pass a message from one service processor to another.

2.1.4 Some Concluding Remarks

The instruction set used in any general purpose processor is generally classified into different groups, as follows :

i. General purpose instructions groups which are usually partitioned into different groups such as data movement, basic logic, arithmetic, branch, etc.

ii. Special instructions to support the system operation, such as instructions for task switching and for virtual memory support.

The processing for data communication nodes is required to perform non numeric

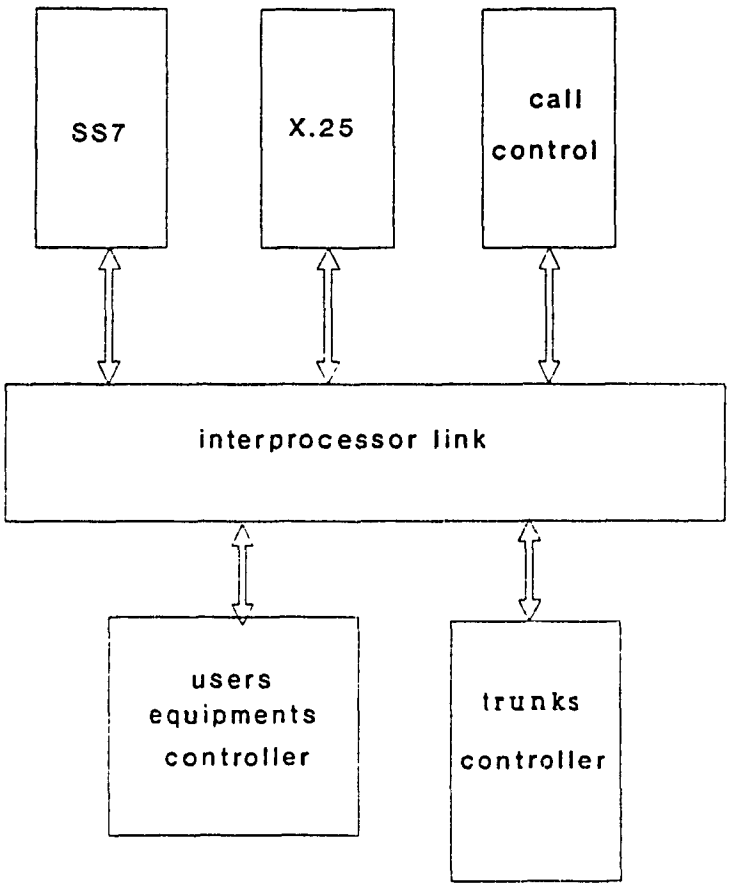


Figure 2.4 Distributed Processing in a Communication Node

functions such as scanning messages, data base searching, etc. Since all of these functions are not numeric-intensive type, the processor used for data communications applications will be different from the processor used for other applications such as robotics and digital signalling processing, which are numeric-intensive applications. That is, the support for the floating point operations can be omitted from the processor support for this application. The support for this operation generally requires a large area of the chip which can be used to support other useful features. The other support for the operating system is still very important to be included within the processor used for this application because the software structure of the network node processing is usually implemented with different software tasks. The support for virtual memory is also very important in view of the large amount of data being used in these types of application.

2.2 Implementation of Network Node Functions Using Existing Technology

Communication protocol processing has been strongly enhanced in the last few years. This enhancement has been achieved with VLSI chips known as protocol controllers [37]. Software functions in both layers 2 and 3 have been implemented by both general purpose processors and specialized processors. Processors used in communication nodes should be suited to multiprocessor configurations in order to enhance node performance for nodes with high traffic demands. This section will emphasize on the current implementations for these software and hardware functions and the type of processors used to enhance the performance.

2.2.1. Implementation of Hardware Functions

The functions which are performed by layer 1 and layer 2 such as bit rating, CRC, flag insertion and deletion, etc., can be implemented by hardware. Progress in VLSI technology has

made possible the realization of special protocol controllers. Processing layer 1 and part of layer 2 can be done by these protocol controllers to relieve the load of the host processor.

Many chips which are available today are designed for processing specific protocols. They cannot be used for different protocols and also become useless if the protocol has been modified. The latter case has led to the design of partially programmable controllers. This feature is very useful to cope with highly possible changes since typical protocol standards are usually imprecise and incomplete [37].

One common feature in the architecture of these protocol controllers is the incorporation of Direct Memory Access (DMA) and buffer management capabilities. The reason behind this is more feasible to store the user data in a common area of the host computer memory as opposed to storing it locally within the protocol controller itself. Moving data to and from the host memory requires a DMA interface. In addition, the protocol controller needs a simple method for accessing and referring to these buffers. This buffer management operation is a significant data management function.

Many chips are available today which are classified as communication controllers such as MC68605 X.25 protocol controller [39], MC68302 Integrated Multiple processor [38], and MC68824 Token Bus controller [40] from Motorola, 82586 Ethernet LAN controller chips from Intel [41], the VLSI implementation of the X.25 protocol from the Nippon Telephone and Telegraph Company (NTT) [42], etc. Some controllers are also available to support the ISDN protocol processing, such as the support for the multiplexing of its D-channels [43], a complete chip set supporting the subscriber access protocol [44, 45]. Clearly, there are many chips available for processing of different data communications protocols and they can be used as peripheral devices of the node host processor(s) to achieve a higher performance.

2.2.2. Implementing Software Functions

The layer 2 and layer 3 functions such as the management, database processing, message scanning, etc., are generally implemented in software and executed by the node host processor. The host processors used in communication network nodes, in general, comprise of general purpose processors or special processors. The general purpose processors are always available and can be used immediately, whereas the special processors can only be used once they have been developed and produced. Clearly, the special processors designed for data communication processing provide higher performance than the general purpose processors, because their design is tailored to a particular application.

2.2.2.1. Existing Host Processors

Basically, the design of the general purpose processors is intended to be used in different applications, and hence their structures are such that to provide a variety of features to satisfy different users. This makes their structures complex. Unfortunately, these processors cannot provide all possible features on the same chip because of certain limitations, such as the number of transistors which can be integrated on one chip, complexity of the designing chip, high costs, etc. These limitations have led to compromises, that is, keeping on chip the features which are believed to be more frequently used than others. For instance, in the Intel 80386 chip the memory management is kept on-chip while the floating point unit is integrated in a special floating point coprocessor. All of these type of processors have some type of supporting chips such as coprocessors to support them in applications that require some feature which is not directly available by these processors. For example, if the application requires many interrupt lines, an interrupt controller support chip can be used to enhance the host processor in dealing with this demand, likewise, a DMA controller can be used when there is a need for high speed data

transfers.

However, the comprehensive structure of these processors has two main drawbacks when using them in special applications:

i. A high percentage of the chip resources will not be used. For instance, a processor with a virtual memory support on chip will not need to be used if the target application requires only a small memory size.

ii. The complexity of their structure has an impact on their performance. Clearly, the use of larger hardware on chip will have an effect such as reducing its operation speed. Moreover, having a large set of complex instructions will contribute to the difficulties of maintaining the streamline of the pipelining operations.

Despite the drawbacks of the general purpose processors, many of today's communication network node processing is achieved by using some of the available general purpose processors due to their low cost and their availability. The HDX10 distributed control digital switching [46], class-5 exchange [47], etc., are few examples for nodes used general purpose processors in their node design.

One of the key points in selecting a processor to be used in communication network nodes is the performance in terms of the processing speed. Therefore, it is crucial to have a host processor capable to achieve this objective. The limited performance of general purpose processors has led to the implementation of some specialized processors to provide the high performance processing requirements of these nodes. Specialized processors always incorporate within their architecture some special supports to enhance their performance when they work in a special environment. Generally, these specialized processors are designed to support specific applications (e.g. image processing, data communications, signal processing), others support the

specific programming languages, (e.g. C. Prolog, smalltalk, Lisp), and some support specific system configurations (e.g. data-flow machines, systolic array) [48].

In communication network nodes, a few special processors are used to enhance the processing. One example of these is the chip set designed by NTT [49]. One of their chips has been specifically designed for instruction execution which are required in such applications such as manipulating of bit-wise data. The processing of these instructions are implemented by a microprogram control unit within the chip. The reason behind having a processor with such instructions is to substitute using a general purpose processor by a more efficient specialized processor. The main objective of the other chips in this chip set is to reduce the amount of circuitry which usually used to interface between the host and the chips used for controlling the subscribers lines, i.e., the work like the controllers chips which mentioned before. This set of VLSI chips is designed to be configured in a multiprocessor environment, and hence can support a small to large network node processing need. The D70 digital switching system and the D51 packet switching system utilize this chip set [50, 51].

2.2.2.2. Multiprocessor Configurations

Communication nodes are required to provide service over a broad range of traffic capacities, i.e. they should efficiently provide services for either small or large traffic capacities at each node with minimal delay. Hence, the processing system of each node should be flexible enough so as to accommodate different traffic capacities requirements. Moreover, with on going development in communication systems, extra services may be required. In order to integrate such services easily then the node required to handle them must be easily upgradeable. These nodes should also be high reliable in order to keep providing services without any interruptions. Such requirements will need to have a high degree of modularity and control distribution in the

communication node design.

To support these requirements, processors used in communication nodes should be able to provide the appropriate enhancement needed in order to easily modify a multiprocessor based node. There are two types of multiprocessor systems used for large and medium node processing: the "tightly coupled", where processors communicate through a shared main memory, and the "loosely coupled" in which each processor has a large local memory. Both types are used in present communication node processing [46]. The high performance achieved by using the loosely coupled structure has made it a more favourable one for these type of applications [52].

To facilitate functional expansion required by these nodes, the node functions should be shared between node processors. A suitable division of the switching program in a system is to dedicate each processor to a certain service; this division is also known as "function sharing" [51]. For instance, the line processor carries out subscriber signalling processing and line concentrator functions, while the call control processor carries out service analysis and call control tasks.

Clearly, sharing the node functions between processors can allow new equipment and services to be introduced without affecting the other processors which are not concerned. Moreover, function sharing reduces the communication between processors and therefore reduces the traffic congestion on the system bus while increasing the overall system performance. In spite of this reduction on the system bus, it is still important, with the large traffic and long message lengths in present communication nodes to reduce these effects even further. For instance, a Message Passing Coprocessor (MPC) can be used to move this overhead from the main processor and therefore increase the processing parallelism and enhance the systems performance, e.g. Intel 80389 MPC [53].

2.3. Reduced Instruction Set Computers (RISC)

The use of an efficient processor to perform functions of higher layers in the data communications is an important issue. The latest developments in computer architectures have made possible to design a high speed general purpose processor based on the new concept of the Reduced Instruction Set Computers (RISC) which can provide the performance higher than the CISC-based processor [69].

In 1975, IBM designed a minicomputer, called IBM 801, in order to achieve a better cost/performance ratio for High-Level-language (HLL) programs which endorsed the idea of simple hardwired control [61]. Although at that time the term RISC was not well defined, this IBM trial is considered to be the pioneer in the building of the RISC machine. At Berkeley, in 1980, Patterson and his research team investigated the RISC architecture and constructed the first RISC processor. By 1981 and 1983 respectively, the RISC I and RISC II processors were the outcomes of this intensive research [62]. Meanwhile, Hennesey's efforts at Stanford University resulted in the microprocessor without interlock pipeline (MIPS) RISC processor [63,64]. The success of RISC architecture have made many of such processors to be available in the last years, such as the CLIPPER [74], MC88000 [65], I860 [66], and R2000/3000 [67], etc.

The performance of the RISC-based processor has been evaluated in the first prototype RISC processor (RISC I). It was measured by comparing RISC I performance with some existing CISC microprocessors [69]. The research outcomes of these RISC prototype machines have shown that their performance is higher than that of existing CISC machines. There are many reasons behind the higher performance of RISCs. They are as follows:

i. Using complex instructions requires additional hardware components placed on the data path that may be part of a critical data path. Longer wires and extra circuitry results in slowing down the overall machine cycle, thus slowing down instruction execution.

ii. Many complex instructions can be executed faster when they are replaced by a sequence of simple instructions. For example, substituting the use of the instruction LOAD MULTIPLE which is used in an IBM-370 with a sequence of simple LOAD instructions has been proven to execute twenty percent faster than its complex counterpart [61].

iii. The simple instructions used with RISCs are typically executed in one cycle. This has a positive effect on pipeline operations by allowing them to operate at a maximum speed. Clearly, complex instructions will force the pipeline to freeze in certain circumstances, and hence the full advantages of using pipelining cannot be explored. The use of simple instructions helps to manage data dependencies easily and efficiently (as will be discussed in the next chapters).

iv. Other reasons such as the use of large specially organized internal registers contribute to the high performance as is characteristics of RISCs. This topic will be discussed in the following chapters.

It is useful to mention here that the technology used in implementing a processor has a great effect on the processor's performance, e.g. implementing processors with ECL technology will provide faster processors than implementing them with CMOS technology. The latest developments with Gallium Arsenide (GaAs) has made it possible for processors to attain a much higher speed than those using ECL technology, have a lower power dissipation and can integrate about 30K transistor on-chip [70, 71]. This achievement is very important when considering the Stanford MIPS which was developed using around 24K transistors, i.e., it is possible to use GaAs to develop a RISC as such taking advantage of this very high speed technology [14]. However, the complexity of the CISC architecture requires a large number of transistors to implement such processors far too many than can be implemented with this technology.

The development of CISC processors is a very costly and time consuming process. The RISC processor design has minimized this to the lowest possible level. This minimization comes from the simplicity of the RISC architecture and the large degree of design regularity. This design regularity and effective utilization of the hardware resources are very important factors in VLSI design. The impact of these factors has reduced the developing cycle of these RISC-based implementations which in turn has reduced their productions cost. This was considered an important aspect in adopting RISCs to design a processor working in different applications, and more specifically the specialized ones.

2.4. Motivations of This Research

There is no doubt that using an efficient processing element in the data communications network nodes is a crucial factor in increasing the overall node performance. This is the main reason why in this research we have focused on searching for a way to improve the efficiency of these nodes through concentration at the processor level.

The research motivations can be summarized by the following points:

i. As was mentioned in the previous part of this work, the RISC concept provides suitable and useful features, e.g., high performance RISC-based processors; a short development cycle at lower costs than CISCs; and very high performance processors with high-speed technologies such as GaAs. These have made the RISC architecture more favourable than the CISC architecture for use with data communications processing software at the network nodes. Moreover, the reported success of the developed RISC-based processors have also attracted our attention to participate in this new area of computer architecture.

ii. Much work has been done in the evaluation of the RISC architecture in general purpose

computations. Unfortunately, few studies have been performed to investigate the suitability of this architecture for specialized applications. Hence this has increased further the importance of this work.

iii. Investigation of data communications software behaviour is very important when searching for a realization of a processor architecture. The unavailability of such a study has added another reason to this research motivation.

iv. In data communications processing (as mentioned in the previous part of this work), there are some operations which are frequently processed (such as task switching and wrapping/unwrapping messages). This fact motivates investigating the adequacy of the present RISC concept to efficiently support such processing operations.

2.5. Work Directions In This Research

Two major issues have emerged in computer architecture designs since the introduction of Berkeley's RISC. One concerns the merits of using an instruction set consisting of simple but fast instructions as opposed to using complex and powerful instructions. A second issue is an organization scheme that keeps more data in fast internal memories, such as registers and caches. However, the study of developing a processor using the RISC architecture for data communications applications is required in order to investigate these two issues.

As mentioned previously, the processors used in the data communications node processing have to be capable of processing a variety of functions which require general purpose type of instructions. The processing of some frequently used operations in such applications does not prevent the use of simple and fast instructions which are recommended by RISC architectures. Using the simple instructions will not require more effort in the software developing for such applications since most of these software are usually written by High Level Languages

such as C.

Today there are many RISC-based processors which are available and they have very efficient instruction sets, e.g. the I860, R2000/3000, MC88000, etc. There is no doubt that a proposed processor for such applications will not require the support of one group of these instructions (the arithmetic instructions) because this application is a non-numeric type, and hence the floating point unit can be omitted from such a processor. This further simplifies the design of the hardwired control unit and provides more free space on the) which can be used to support other architecture features.

Based on the above discussion, we have decided to investigate other major RISC issues which contribute to high performance, i.e., the use of an internal register set. Generally, the number of these registers used in RISCs are generally greater than those used in CISCs. Moreover, the organization of these registers, i.e., windows organization, as will be discussed in the next chapter, in the Berkeley RISC have contributed to an improve the RISC performance. All research and development which have been done relating to this area of RISCs focused on utilizing these registers to improve general purpose computations. Meanwhile, none of these studies have investigated the impact of using these registers on the network nodes applications where their processing is classified as a real-time multi-tasking. We believe that there are two reasons why these studies have not followed through such an investigation:

i. In the early stages of research of developing the RISC, the main concern was given to general purpose computations and showing the superiority of this architecture in comparison with the CISC one. All efforts were concentrated into enhancing the architecture in this direction.

ii. Real-time applications are basically control systems [73]. Support provided to enhance the RISC architecture to improve the performance of one real-time application will not

necessarily work for others.

Utilizing the available register set used in current RISC architectures could have an impact on the RISC performance adopted for this real-time multi-tasking application in two ways:

i. The large number of these registers will increase the processor state and hence reduce processor performance when task switching operations occur frequently.

ii. The register organization is also designed to support general purpose computations and more specifically to process intensive procedure calls with large nested depths. However, this is not the target of network node processing (as will be discussed in the next chapters).

In this research an emphasis is given to search for better ways to enhance the register set within these general purpose RISC architecture type and to improve the performance of such an architecture when it is applied to network node processing. The study of the register set within the RISC architecture, however, raises some issues:

i. How does the present RISC register set perform when it is used for the network node software processing?

ii. What type of register set organization and what number of registers used in current RISCs are appropriate for this application?

iii. What is the impact of using other register set organizations which are not currently used in RISC architectures, i.e., such as non overlapping multiple register sets? Do they give any performance improvements and, if so, what are they?

The main objectives of this research are to answer the above questions.

2.6. The General Approach

To achieve the main objectives of this research, the general approach used in our work presented in the next chapters can be summarized as follows:

i. The first part of this research is devoted to the study of the nature of the data communications software which runs inside the network nodes. This will be done by investigating some data communication software in order to understand their characteristics and to determine which one of the available RISC-based architectures is more appropriate for networks nodes processing.

ii. In the second part, an emphasis will be given in enhancing the architecture through investigation of the register set organization. A special concern will be given to the study of the impact of RISC register set organization for the task switching overhead, which is not given enough study so far [64].

iii. Based on the results of the above two points, a RISC structure for data communications using non-overlapping multiple register sets introduced and investigated. Performance improvement and hardware complexity of the introduced architecture are studied.

CHAPTER 3

ISDN SOFTWARE CHARACTERISTICS

This chapter investigates the software characteristics of data communications and specifically ISDN. The main objective is to study parameters which could be used to design the RISC processor for data communications applications. In particular, we are interested in determining the appropriate structures of internal register sets of the processor. At first, the subject of measuring software characteristics is investigated and its objectives are highlighted. Then methods used for such measurements are outlined. To find the software characteristics of the ISDN processing, two ISDN processing models are proposed and simulated. The results of these simulations are also discussed.

3.1. Software Characteristics: Background

Software characteristic is a term used for investigating the internal structure of program and measuring its parameters, e.g., the type of the instructions used, the frequency of using each of these instructions, the amount of nested procedures depth, etc.. These measurements are very important to select an appropriate structure for the RISC processor suitable for special or general purpose applications. Taking these measurements will help to identify the necessary features that have to be incorporated within the processor in order to improve its performance. As it will be shown in next section, the use of register windows structure with Berkeley RISC is one of such examples.

Roughly speaking measurement could be classified as dynamic or static [75]. In dynamic measurement, we counts the number of times that each HLL statement is executed when the program is running. Conversely, static measurements are performed by counting the number

of occurrences of each HLL statement in the program. Dynamic measurement is preferred because it indicates the frequency of each statement during the real-time execution of the program.

Recently, a number of attempts been deployed in order to determine the characteristics of HLL programs [75]. The main objective of these studies was to identify the type of the HLL statements which occur most often in the program so that they could efficiently supported by the designed processor. Numerous collections of programs have been tested for measuring software characteristics. Knuth [91] has used a collection of FORTRAN programs used for students' exercises. Tanenbaum [63] studied Over 300 procedures, executed from operating system programs and written in a language which support structured programming (SAL). Huck [75] analyzed four programs representing a mix of general purpose scientific computing. The results of all these attempts confirmed the predomination of assignment statements in programs, and therefore suggest that the simple movement of data is of higher importance. Hence, these results were exploded for designing instruction set for computers. The main drawback of these studies, however, is that these results reveal which statements are frequently used but they do not show which statements are frequently invoked during the execution time of a typical program.

3.2. Impact of Software Characteristics on RISC Architecture

Berkeley researchers were the first to investigate the drawback associated with the previous studies [69]. This investigation performed through the study of a collection of different programs (which comprised of typesetting, CAD, Sorting, and file comparison), they found that the procedure call/return operation represented the most time consuming operation and required about 45 ± 19 % of the total memory references executed by the processor [69]. Thus,

reducing the memory references generated by the intensive use of machine instructions, constituting the call/return operations within the HLL statements, has given a large effort towards the RISC architecture. There are two known schemes used to reduce memory references generated due to the call/return operation. In the first scheme, which is proposed by the Berkeley group for the RISC I, a large register set is used within the internal structure of RISC I to keep the operands on-chip and therefore reduce the memory access. This large register set is partitioned into a number of Overlapped Register Sets (ORS). Each set has three sections, where two of these sections are used to pass the procedure parameters between the next and previous procedures, while the third section is used to store the local variables. That is, in each set, the overlapped registers are used to pass parameters among procedures, and the non-overlapped registers are reserved for local variables. Both overlapped and non-overlapped registers constitute one register set, called a window. In addition, there is also another set of registers which are used to store global variables (Figure 3.1). The global registers are not switched during the procedure call/return operations. Selecting the number of registers allocated for local variables and passing parameters, as well as the number of chosen windows is based on statistical results that have been generated from various studies [69]. One study, in particular, has found that with eight windows, a save or restore is needed only 1% of all calls or returns [77]. Other studies have shown that 98% of all procedure calls which were traced dynamically were passed with fewer than six arguments, and that 92% of these procedures used fewer than six local scalar variables [76]. Similar results were reported by the Berkeley RISC team [62]. The Berkeley RISC computers use 8 windows of 16 registers each (6 are overlapped registers and 10 are for local variables), while Pyramid computer employs 16 windows of 32 registers each [60].

The second scheme uses the optimizing compiler to allocate the most frequently used

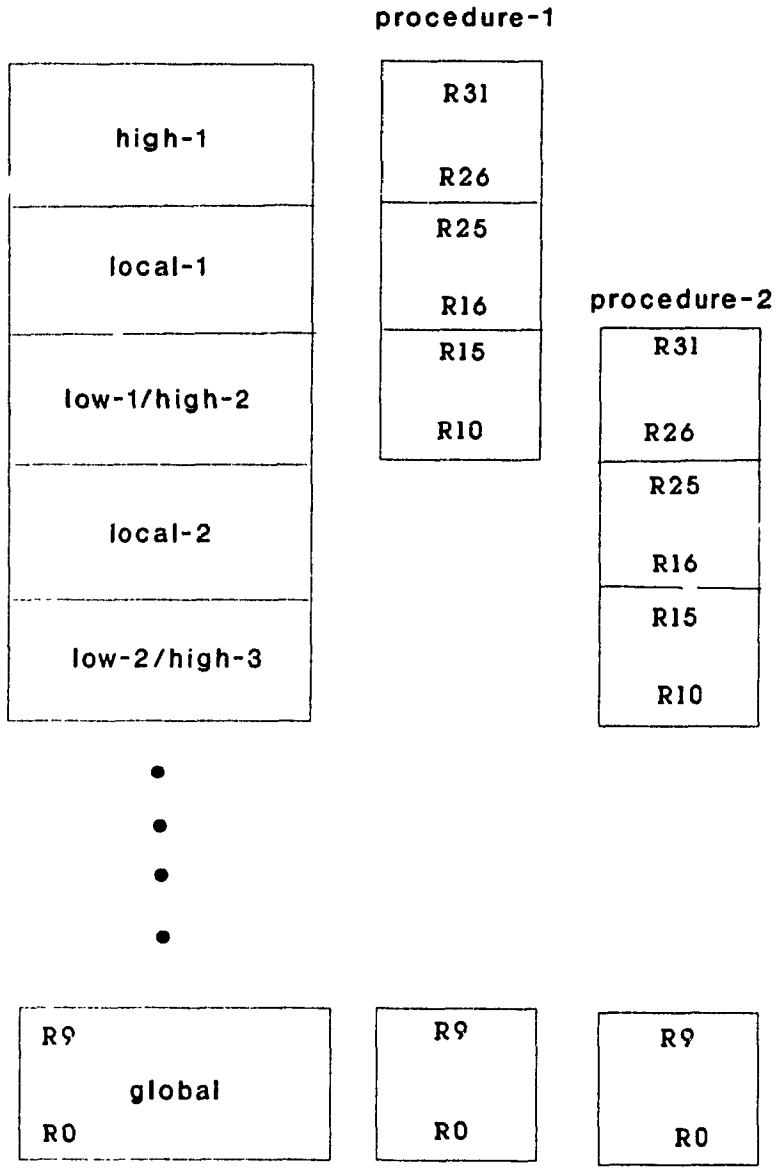


Figure 3.1 RISC Overlapped Register Set Organization

operand to the internal registers. This state-of-the-art in compiler technology is used in the Stanford MIPS architecture to maximize the use of registers. Improvements in compiler technology, mostly in the form of good models for register allocation, makes it possible for compilers to achieve very high register hit rates, and to handle saving/restoring in a more efficient way at procedure call boundaries [62]. In other words the first scheme is fixed by hardware while the second is based on the facility of the optimized compiler (software). The MC88000 [65], R2000/R3000 [67], Intel i860 [66] are examples of processors implementing such techniques.

It was shown in [79] that, in general, the performance of both schemes is essentially equal but in certain cases, the register windowing scheme performing better [80]. For instance, the ORS scheme has a better performance in some cases, such as for those applications involving high frequencies of nested call/return procedures within the limited of the overlapped registers. However, the high percentage of time spent in leaf procedures minimizes the advantages of this technique.

Clearly, the effectiveness of any of the two RISC schemes depends on the type of software which runs on the processor. Thus, it is important to know the characteristics of the data communication applications in order to know which scheme is more appropriate for this type of applications. To achieve this, it is required to determine the factors which have to be measured and to show that some ISDN protocol functions will be simulated. It is also required to determine the measurement method.

3.3. Factors to be Measured From ISDN Software

In order to identify the most suitable RISC architecture for ISDN processing and if this architecture is to be enhanced for ISDN processing, we require that some factors be measured

from ISDN software processing. These factors consist of:

- The utilization of HLL statements.
- The nested procedure depth.
- The size of parameters passed between the procedures.

We will focus on these factors since they can help to clarify one of the main differences (i.e. the register set) between the Berkeley RISC and the Stanford MIPS approaches for designing RISC architectures. These factors also determine which approach is more suitable for our particular application. Studying these factors will help to answer the following questions:

- What is the ratio of the procedures call/return in comparison to the other operations? Does this ratio encourage us to adopt the windowing or using optimized compiler approaches?
- How large is the nested depth of the procedures calls? The answer to this will help to decide on the size of the register set and hence determine the number of overlapped register sets.
- What is the size of parameters passed between the procedures? The answer to this question will help us in determining the size of the registers in each of the overlapped sections within the overlapped register set. This could also show whether the optimizing compiler approach is effective for managing the passing of parameters.
- Which instruction(s) are most frequently used within the programs? The answer to this question will help the processors designer to support these instructions efficiently.

3.4. Measurement Method

Dynamic measurement of software characteristics are used to study the previously

mentioned factors. This is done by running typical ISDN programs on a specific machine.

During the development of RISC I structure, programs were compiled on CISC-based machines such as VAX, PDP-11 and a Motorola 68000 [69]. This was done to show the drawbacks of CISC processors and provided methods to enhance the performance with RISC architecture. In our case, the simulation programs are compiled on a RISC-based machine in order to find an efficient way to enhance the RISC architecture for data communications applications.

3.5. Simulated Models of ISDN Functions

As software characteristics are important in general purpose computations and have helped to adopt the windows structure in RISC I and II, it becomes crucial for the other specialized applications to achieve the same goal of finding the appropriate support. Since ISDN processing nature is of a general purpose type (as mentioned in the previous chapter), the two RISC architectures schemes have already supported this type of processing. The study of ISDN software characteristics will help us to identify which one of these two RISC architecture is more appropriate for ISDN applications. Moreover, this will show whether there exists any better way of enhancing the RISC architecture for ISDN processing.

Choosing the software to be analyzed is an important issue. The size of the ISDN protocol is very large and therefore the full protocol simulation is a very complex and time consuming process. Thus, we decided to simulate typical ISDN functions which can represent the overall software characteristics. Two ISDN functions in different layers have been considered for our studies, namely the Q.931 basic call control (layer 3) and the TEI assignment (layer 2) procedures. This will enable us to study the general characteristics of the software for different layers rather than concentrating on a particular layer, hence making our study more

comprehensive and accurate. The simulation program for these two functions is written in C language.

3.5.1. Q.931 Basic Call Control Model

The goal of call control procedure is to handle the calling user signalling in order to select the desired user and communication services. This procedure is recommended by CCITT Q.930 [82] and Q.931 [4]. The model used to simulate the basic call control for layer 3 consists of the establish/terminat phases (Figure 3.2). In the established phase, the calling user information transfers to the network in the SETUP message. The network confirms through a CALL PROCEEDING that call establishment is in progress and at the same time assigns a vacant B-channel to the calling user terminal. As soon as the destination exchange detects the connection request, it selects the addressed line and transmits the connection request (SETUP) to the terminal equipment of the called side. The called terminal processes the signalling information and informs the network accordingly with an ALERTING (corresponding to telephone rings), and sends only a CONNECT when the terminal becomes ready to accept the call (e.g., telephone handset is lifted). The connection is now set by the network and assigned, on a CONNECT ACK, to the called side terminal equipment. The calling terminal is informed of the progress of the call establishment by the ALERTING and the CONNECT messages. The D-channel in both calling and called sides terminals are used to convey all these messages.

For termination, either the calling or called user terminal, or the network can initiate connection clear at any time and independently from each others by sending the DISCONNECT message. When the call clearing is initiated by the user terminal, the terminal disconnects itself from the traffic channel and releases it before sending the DISCONNECT message. After receiving the DISCONNECT message, the network disconnects the traffic channel, initiates

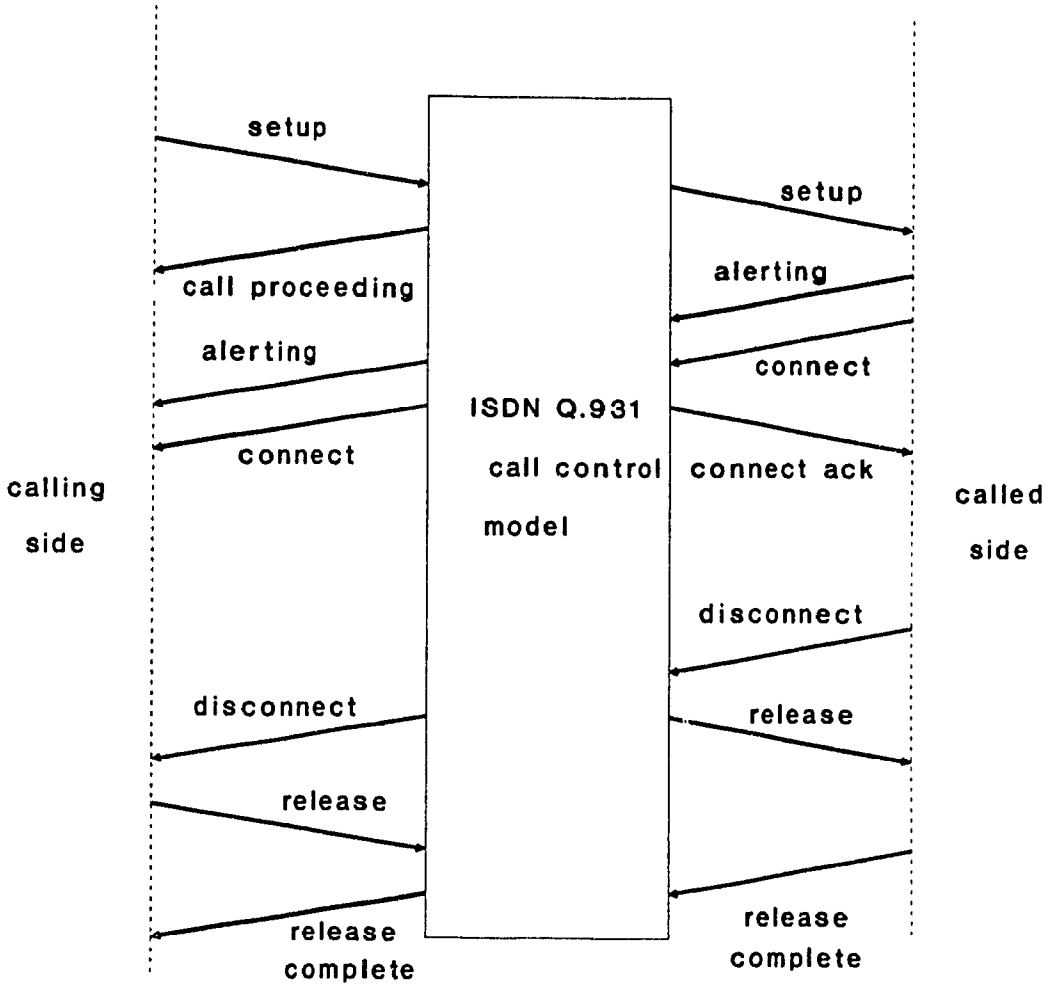


Figure 3.2 Q.931 Task Operation

connection clear within the network and acknowledges with RELEASE. Finally, the user terminal terminates signalling activity by sending RELEASE COMPLETE, and the network terminates the signalling activity. Similarly, the network disconnects the traffic channel from the internal network connection and, with the DISCONNECT message, requests the user terminal to disconnect itself immediately from the traffic channel. The user terminal confirms this with a RELEASE message and the network finally terminates the signalling activity by sending the RELEASE COMPLETE message, upon which signalling activity is also terminated for the user terminal.

The call control model takes care of implementing ISDN signalling functions such as checking the validity of a message, checking for service availability, performing B-channel negotiations, issuing a status messages (if messages errors are detected), and updating databases. All the messages processed are of Q.931 type as recommended by CCITT. These messages are extracted from the information field of the LAPD (Link Access Procedure on the D channel) messages. The information fields (the mandatory and the optional) of the Q.931 messages used here are shown in Figure 3.3.

To make the simulation feasible, we have made the following assumptions:

- Layer 2 has already processed all LAPD information and passed only the Q.931 information part to layer 3.
- All messages passed from layer 2 to layer 3 are of the Q.931 call control type, hence no protocol discriminator information field is used (as recommended by CCITT Q.931 messages format).
- A message length is added to simplify message scanning, and it is used as the first information field in the Q.931 messages.
- Any processing related to hardware functions such as digital switch controlling is not considered in the simulation.

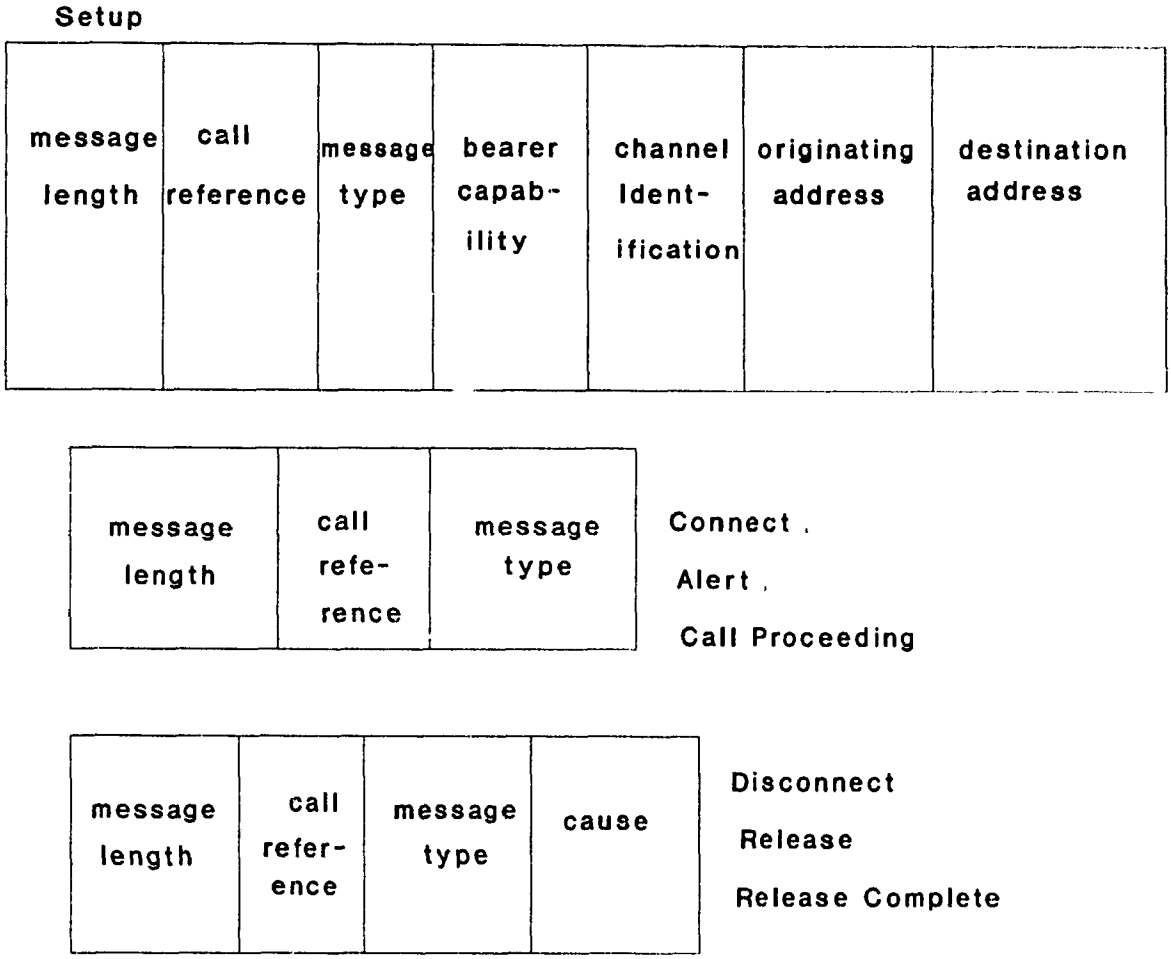


Figure 3.3 ISDN Messages Used in Q.931 Model

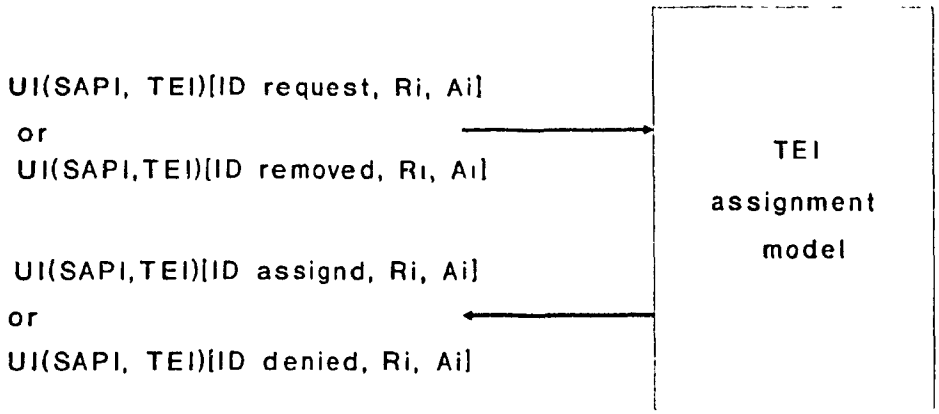
The simulation model is organized as a set of procedures. Each ISDN message and each information field form one procedure. This model also allows future modifications to include more messages and services. The model is organized in one program, however in reality ISDN functions are processed concurrently and the Q.931 protocol is typically processed as one or more tasks in a real-time multi-tasking environment.

In the simulation we consider two types of processing namely message processing (e.g. scanning, testing, etc.) and database processing (e.g. creating record, deleting record, searching).

3.5.2. TEI Assignment Model

Since every terminal equipment in a bus configuration must have a different Terminal Endpoint Identifier (TEI), a management entity within the data link layer automatically handles TEI assignments on the network side. Immediately after a terminal equipment initiates a new TEI value (e.g. after the voltage supply has been restored), it sends a request to the network which in turns will assigns it a free TEI value. Normally the network will hold a store of free TEI values so that it can assign a TEI immediately following a request, thereby eliminating the time-consuming test. There are two schemes recommended in Q.921 for assigning terminals, namely the specific TEI assignment and any TEI assignment. In the first scheme, a certain number is required by the terminal, while for the second any number available by the network is requested by the terminals.

As recommended by CCITT, the TEI assignment model includes both assignments, i.e. specific-TEI and any-TEI (Figure 3.4). The messages which are used in this model represent the TEI management information of the LAPD frame as shown in Figure 3.5. The special management entity identifier used in the TEI management information in the first octet



SAPI: Service Access Point Identifier=63

TEI: Terminal Endpoint Identifier

UI: Unnumbered Information

ID request: Identity request

ID denied: Identity denied

Ai: Action indicator

Ri: Reference number

(): contents of the data link layer address field

[]: contents of the information field

Figure 3.4 Automatic TEI Assignment Task

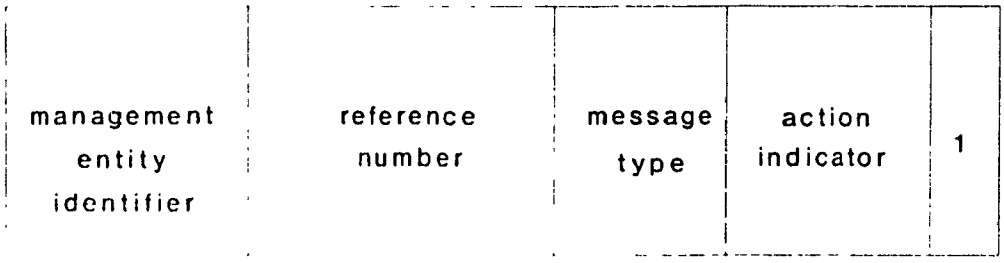


Figure 3.5 TEI Assignment Messages Format

differentiates management from the signalling and packet information. These messages are processed by scanning through their contents in order to decide upon the validity of the messages and upon the type of service required, i.e. whether specific-TEI or any-TEI is required. The model also performs the database searching and updating. Since the messages returned to the terminal in response to terminal requests depend on the possibility of request, the response messages are either identity-assigned or identity-denied. The format of the management messages which simulated here are as recommended by CCITT in layer 2 of the Q.921 protocol [3].

3.6. Results and Discussions

The simulation models are compiled on the R2000 RISC-based machine. The program listing of the simulation included in appendices A and B. The R2000 processor is one among the many processors which are designed based on the Stanford MIPS-RISC [67]. In order to get the data required from these simulation models, we have to run them on R2000-based machine with the data being dynamically collected by testing the models with different types of ISDN messages [11]. Because the models are not running under a real ISDN system, The generation of ISDN messages is simulated. Different messages are simulated with different originating addresses, destination addresses, types of services, certain B-channel numbers, etc., in order to test the functionality of the models. The execution of these messages is dynamically traced to study the software characteristics.

By dynamically scanning and running simulation software, the dynamic frequency of the HLL statements for basic call control and TEI assignment models were calculated for the randomly selected amounts of ISDN messages. Along with these, the average number of machine instructions and memory references per HLL statement type was calculated. By

multiplying the frequency of occurrence of each statement type by its average usage of machine instructions and memory references, the time consumption of each of these statement was obtained as shown in Tables 3.1 and 3.2 respectively.

Columns 3 & 4 of Table 3.1 (the call control model) indicate the actual time spent in executing various statement types. The results show that the ASSIGN statement is the most time-consuming operation for typical ISDN basic call control HLL programs. They also show that the call/return accounts for only 10% of the total memory references. The optimized compiler used in this RISC-based computer system helped reduce the cost of this operation. The SWITCH statement occupied only 0.3% of the total memory references and this constitutes the lowest cost of all statements. The results for TEI assignment (Table 3.2) show that the LOOP statement was the most time-consuming. This is due to the database searching requirements of any-TEI assignment procedure. In our simulation a simple sequential search was used. However, more efficient search techniques could be considered to further reduce searching operation time. Moreover, the procedure call/return was found to use only 2% of the total memory references.

The depth of nested procedures in both models is shown in Figure 3.6. Each call is represented by a line moving down and to the right, and each return by a line moving up and to the right. Selection of a moment in order to see the depth of the nested procedures is chosen where the nested depth reaches a maximum. Obviously there are many other moments where the depth reaches a maximum but selection of only one of these situations is shown in Figure 3.6. The maximum procedure nested depth is 3 (in Q.931), while in TEI assignment model it reaches a maximum depth of 2. Clearly, part of these call/return procedures are leaf procedures.

The number of parameters passed between the procedures in both models is shown in

Table 3.1 Relative Dynamic Frequency of HLL for Q.931 Basic Call Control

statements type	dynamic occurrence (percent)	machine instruction (percent)	memory reference (percent)
ASSIGN	66	49	63
LOOP	4	10	19
IF	11	15	4
SWITCH	4	7	0.3
CALL	12	16	10
GOTO	3	3	3.7

Table 3.2 Relative Dynamic Frequency of HLL for ISDN TEI Assignment

statements type	dynamic occurrence (percent)	machine instruction (percent)	memory reference (percent)
ASSIGN	37	32	46
LOOP	39	45	33
IF	17	18	18
SWITCH	1	1	1
CALL / RETURN	6	4	2
GOTO	--	--	--

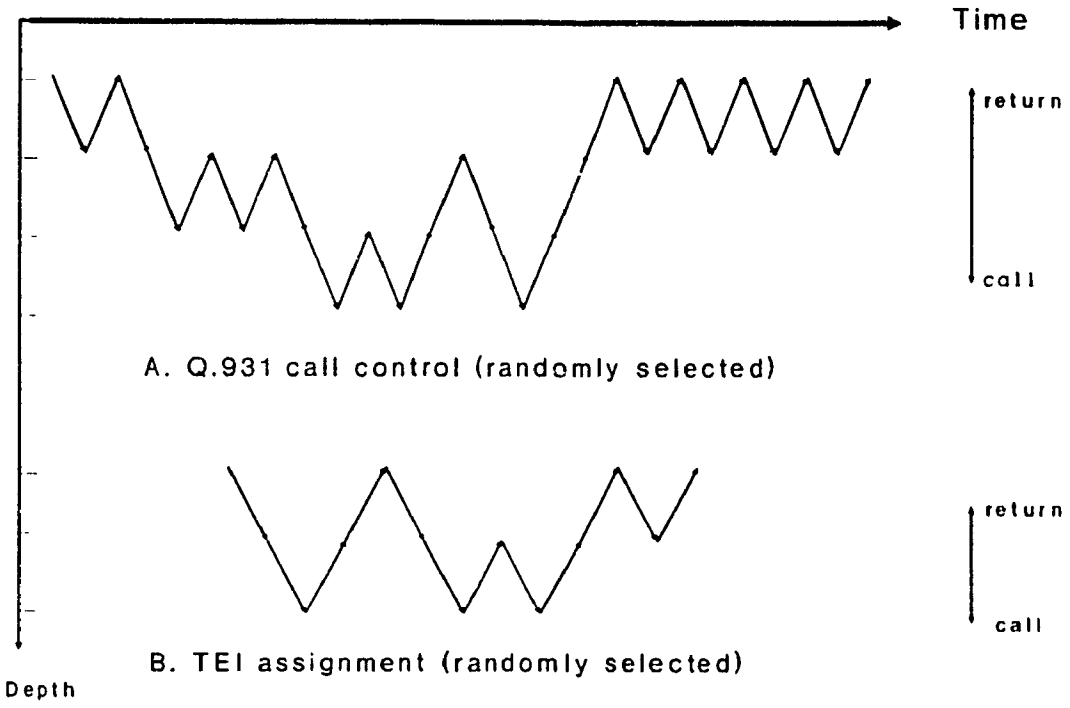


Figure 3.6 Nested procedures behaviour

Table 3.3. The size of passing these parameters is 2 for TEI assignment and less than 6 for call control functions. The smaller size of parameter passing helps the compiler to allocate these parameters passed more efficiently on the CPU registers (32 registers exist within the R2000).

Clearly, all these results were derived by running the test software on a MIPS type RISC-based computer system and hence a windowing type machine was not used. However, these results can be used to see how useful a windowing type machine would be for these types of applications. The windowing or ORS based processors basically work better when the number of nested procedures is large. Generally, the size of ORS's are between 8 to 16 sets. Therefore, using such processors for an application where the nested depths of their procedures is short would not fully take advantage of using such large register sets. The short depth of nested procedures (maximum 3) in ISDN applications efficiently use less than 50% of the register's resources which are available with ORS based processor, assuming that the number of sets is 8. Clearly, this will waste the large register resource which is provided by this type of RISC processors.

This short depth of nested procedures helps the compiler, used in the R2000 RISC based computer, to reduce the cost of the call/return operation to less than 10% of the overall operations cost. Moreover, the small number of passing parameters among procedures helps the compiler to efficiently reduce the call/return costs. Therefore, a single register set associated with an efficient register allocation scheme provided by an optimized compiler is more suitable to ISDN applications than in utilizing an ORS RISC based processor.

Another important point we should mention here is that in ISDN we are dealing with real-time applications where the software is organized in a multi-tasking manner. Thus, use of an ORS will not be useful and can considerably increase the overhead time due to state-swapping which frequently occurs in heavy task switching applications such as in ISDN. This will be

Table 3.3 Number of Parameters Passed Between Procedures

Percentage of executed procedures calls with	Q.931 basic call control (percentage)	TEI assignment (percentage)
< 2 parameters	0	0
2	14	100
3	7	0
4	50	0
5	29	0
> 5	0	0

discussed in next chapter.

CHAPTER 4

PROCESSING OVERHEAD OF RISC STATE-SWAPPING OPERATIONS IN ISDN SOFTWARE PROCESSING

Both structures used for RISC-based processors are aimed at reducing the cost of the procedure call/return operation, which is considered to be most costly in terms of execution time [60, 83]. To this end, a large number of on-chip registers is introduced to retain more parameters on-chip and consequently to reduce the access to external memory. However, having more on-chip registers can seriously affect the task switching overhead. This overhead is associated with the switching from one task to another which requires saving the state of the present task from the processor to memory and loading the state of the ready-to-run task from memory to processor. This state-swapping operation involves the entire set of parameters contained in the on-chip registers. Therefore, a large number of registers associated with RISC architecture requires a long state-swapping operation and hence implies a large overhead in task switching.

Since in most large operating systems the saving of processor state can be a small part of the task switching overhead, and the task switching occurs less frequently than that of procedure calls, most of the previous studies tended to overlook the overhead problem of the state-swapping operation [84-86]. The exact cost of this problem remains however unknown [87]. The large task switching overhead will grow even larger when RISCs are used in real-time multi-tasking applications, since the interaction among the tasks will be more frequent than in the case of other systems such as time-sharing [73].

As discussed in chapter 3, since ISDN processing is considered to be a real-time multi-tasking application, it becomes important to investigate the impact of the internal register organization on the state-swapping overhead. A potential organization is to use Multiple Register

Set (MRS)[1] to reduce the state-swapping overhead. Such an investigation is the focus of this chapter. This overhead in terms of total instruction execution time and total memory references is estimated. Although the processing of ISDN user-network protocol is used in this chapter as an example to evaluate the state-swapping overhead, the results can be extended to other data communications protocols processing.

4.1. Structure of Tasks in ISDN Network Node Processing

In a typical data communications processing, Layer 2 or Layer 3 can be implemented in one or more tasks [27]. This will simplify the implementation of each layer, and will make it possible to modify the tasks in the future (if necessary) with minimum effort. All of these tasks will run under the control of the real-time operating system which is responsible for their synchronization, communications, etc., (as mentioned in Chapter 2). Due to the asynchronous nature of these tasks, the passing of messages from one task to another should be managed through certain message buffers, e.g. mailboxes, rings. Because each service could be implemented in one (or more) task, the messages which exists in the message buffers will be of the same type, e.g., only management messages exist in management task input message buffers. The same thing applies for the signalling and packet data tasks. As an example, Figure 4.1 shows a task distribution of two ISDN user-network interface services, namely management and signalling, in the ISDN node. The LAPD messages are first processed by task 1. Then only the information field of these messages are passed to either task 2 or task 3 depending on the

---

[1]The Multiple Register Sets (MRS) organization [88] is used to reduce or eliminate the state-swapping overhead. In this organization, the task switching can be accomplished efficiently by changing the contents of the current task register in the processor so as to point to the register set containing the state of the selected task.

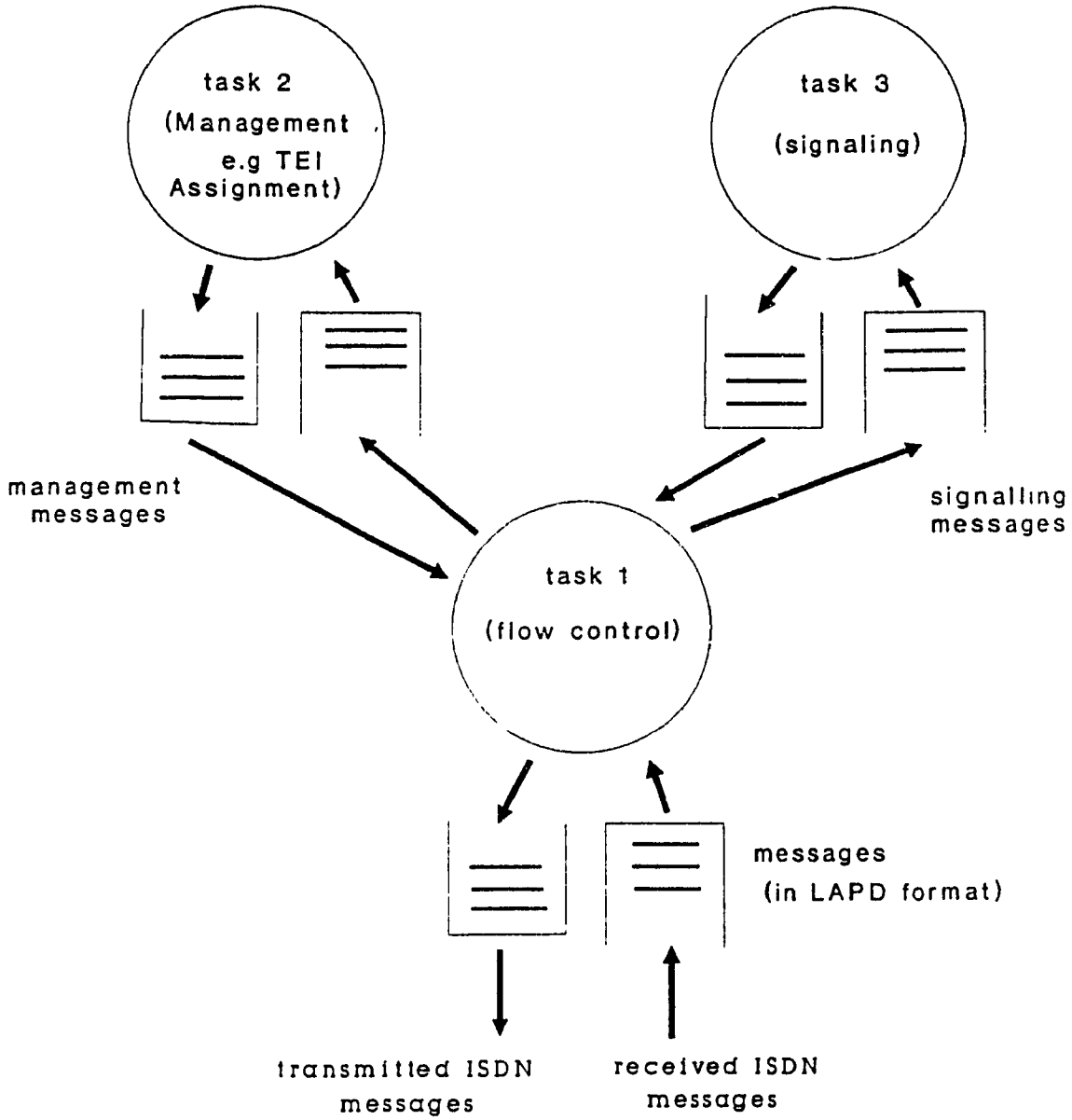


Figure 4.1 Tasks Distribution and Messages Transfer

type of information that exists in the LAPD message (e.g. signalling or management). This message type testing is done by checking the address field, i.e. SAPI and TEI, of the LAPD message in task 1. After task 2 and task 3 process the messages, they will send back response messages to task 1 which will reconstruct the LAPD format message and subsequently sends the message back to the user (if necessary).

When a processor starts a task execution, it will process the messages stored in a task message buffer. This operation will continue until certain events occur, such as when no more message is left in its message buffer, or an external interrupt occurs, or when a high priority task becomes ready to be run as its resources are available, etc. Hence, any ISDN task can be modeled as shown in Figure 4.2.

The processing being performed in each running task is basically dependent on the number of messages in its message buffer. However, it is not necessary that all remaining messages which are left in the task input buffers be processed during the task execution period. The number of processed messages depends on the organization of the tasks as well as other operating system activities. As an example, if a management task has a lower priority than a signalling task, and if we assume that the management task is processing some messages while the signalling task was in sleep mode, waiting for messages to be available in its input buffer and later on received one or more messages, then the operating system will preempt the management task and will start the signalling task.

4.2. Processing Overhead for State-Swapping Operation With RISC

For real-time multi-tasking applications and specially for ISDN applications, using RISC will increase the overhead processing for state-swapping. This increase is due to two factors, one of them is associated with the RISC architecture itself and the other one is the direct

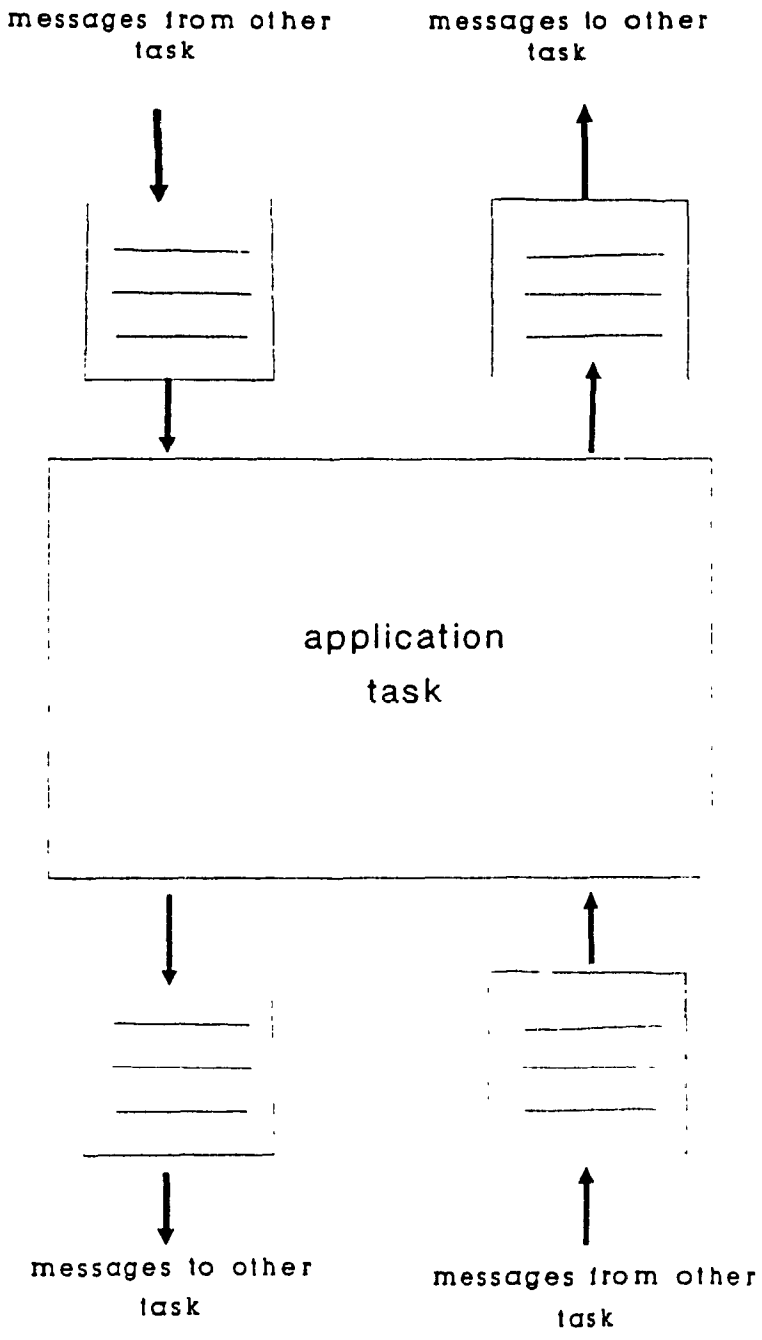


Figure 4.2 ISDN Task Model

consequences of the processing nature of ISDN applications. RISC architecture has the effect of increasing this overhead from two different ways: the use of large internal registers, and the use of optimizing compilers.

4.2.1. Size of Register Set

In general, RISC architecture is characterized by a large number of general purpose registers. In window-based processors, the number of general purpose registers is very large. For instance, there are 138 registers in RISC II as opposed to 256 registers with Pyramid. Obviously, general purpose registers are not the only registers which are involved in the state swapping operation. Registers, like memory management registers, floating point processing registers, program counters, etc., are also to be swapped.

In optimizing compiler-based processors, where there are normally 32 general purpose registers, a smaller state-swapping overhead is generated. Thus, an optimizing compiler-based processor seems to be most appropriate for applications which require fast and heavy task switching, such as in ISDN applications. However, the windowing approach can be justified for such applications only if it can be proved that using such a large number of registers will enhance the processing. Chapter 3 showed that there is no benefit in using a large number of windows in ISDN software processing since the number of windows will always be constrained by the relatively small (around 3) nested depth of procedures. Therefore, a window-based RISC with, say 8 windows (as in the case of RISC II), will not be appropriate since more than half of the on-chip registers are not used.

4.2.2. Optimizing Compiler

The latest developments in compiler design, mostly in the form of good models for

register allocation, made possible for compilers to achieve very high register hit rates and to handle saving and restoring at procedure call boundaries more efficiently [87]. This has reduced the memory references, which are considered as the bottlenecks in improving processor performance.

In RISC, the state-swapping operation is handled as a series of consecutive STORE/LOAD instructions as to save/reload the processor state of present and ready-to-run tasks respectively. The numoer of instructions which are used to perform the state-swapping operation depends on the number of involved registers. Obviously, having more on-chip registers increases memory references during the state-swapping operation.

State-swapping overhead can be measured by two possible methods:

i. In terms of the total task switching processing: The task switching is performed by the operating system and is basically comprised of two parts: the state-swapping handling, and the managing of the switched tasks such as the selection of the task next to be run. The management of the switched tasks is optimized during the system initialization stage where its corresponding program is compiled, hence the related memory references can be reduced. Therefore, the minimization of the total task switching processing time will increase the ratio of state-swapping overhead to the total task switching processing.

ii. In terms of the total application task processing: Since we can express the task switching overhead as a percentage of the total application task processing and since the state-swapping is only one stage of the task switching, then we can evaluate the state-swapping overhead in terms of the application task processing. The application tasks are already compiled and optimized, and hence their memory references are also optimized. As the memory references generated by state-swapping increase in the case of RISC-based processor, the memory references made by executing other application

tasks will decrease due to the use of the optimizing compiler. Therefore, the ratio of overhead processing for the state-swapping operation to the application task processing will increase.

In addition to the reduction in memory references due to the usage of the optimizing compiler, the optimizing compiler is also capable of further reducing the cost of the LOAD instructions by using the *Delay Load technique.* This technique, which is used in many RISC designs, enables the compiler to test the next instruction after the LOAD instruction and see whether or not it needs data from the LOAD instruction [67]. If there is data dependency, the compiler will try to assign a useful instruction after the LOAD. Otherwise, a no operation (NOP) instruction will be inserted after the LOAD instruction. This will reduce the LOAD execution time, i.e. instead of making the processor wait until the LOAD operand is available (which could take one or more cycles) the processor can execute some useful instruction(s) existing in the following delay slot(s) after the LOAD instruction. Ideally, this technique could reduce the LOAD execution time to one cycle only if all the delay load slots can be filled with useful instructions. However, compilers can insert useful instructions only about 90 percent of the time [60].

During the state-swapping operation, the operating system needs to restore the state of next running task by reloading the processor state. This is achieved by issuing a series of LOAD instructions to reload the processor registers. Unfortunately, the load delay technique can not be used during swapping time because the processor is completely idle during that stage and also because there is no way to arrange a useful instruction before the processor state becomes completely restored. Hence, the ratio of memory references required by the state-swapping operation to the memory reference made by the application task processing will become larger. The same discussion holds if the comparison is made with reference to the task switching

processing.

4.3. An Approach to Measuring State-Swapping Overhead

To measure overhead associated with the state-swapping operation, an experiment should be performed by executing an ISDN software program using a RISC type processor controlled by a real-time multi-tasking operating system. This should also be tested under a real ISDN traffic situation. This processing should be dynamically traced in order to calculate the overhead, which is very time consuming task.

It is essential, therefore, to find a less complicated method to estimate this overhead. Two things should be known beforehand: the processing of the state-swapping operation and the processing of the ISDN application tasks. The number of processor registers involved in the state-swapping operation, which is already known, will simplify the calculation of the processing time for this operation. Moreover, our previous work on evaluating some ISDN software (mentioned in the last chapter) from layer 2 and layer 3 can be used for estimating the ISDN application tasks processing. Our purpose is to develop a simple and practical method to evaluate the state-swapping overhead. Some ISDN functions already have software written for them (as mentioned in the previous chapter). The processing associated with the state-swapping operation with each task switching can be estimated easily. The measuring of the state-swapping overhead therefore can be achieved by comparing the processing required for the state-swapping operation with the application software processing.

It is obvious that the amount of processing of each ISDN task is proportional to the number of messages being processed during the task execution time. The operating system acts as the controller to start or to stop a task, or to determine which task is next to run, etc. These activities are the ones which determine the number of messages being processed by each

task, hence there is no definite number of messages which are processed during the task execution period. Therefore, it is possible to simulate each ISDN task separately and measure the amount of processing for a different number of ISDN messages. By doing so, we can simulate each task on an available RISC-based machine without any need to use a real-time multi-tasking operating system. This greatly simplifies our task as we do not need a real ISDN environment to be implemented; and with only a few of the application tasks being simulated, we can have an estimate of the state-swapping overhead [16].

Our approach can be summarized as following:

i. Typical ISDN functions are partitioned and simulated as tasks. These tasks are programs running separately with RISC-based machines under the UNIX operating system. Synchronization and task communication are not included. The two programs to measure the software characteristics described in the previous chapter are used here, where it is assumed that each program is representing one separate task. These two tasks are compiled on a RISC-based machine with an optimizing compiler.

ii. The overhead processing will be estimated for the state-swapping operation in terms of both the total machine instructions and the total memory references.

iii. The amount of processing for each of these two ISDN tasks is measured for different ISDN messages, and in terms of the total machine instruction and memory reference processing.

iv. The average amount of processing for each task is measured in terms of the processing required for an average ISDN message.

v. Finally, the overhead ratio of the overhead processing for the state-swapping to each task processing is calculated for different amounts of those messages which exist in the task input message buffer.

4.4. Measuring State-Swapping Overhead

The processor state determines the amount of processing required for the state-swapping operation. Generally, most RISC processors which utilize optimizing compilers for register allocations, use 32 general purpose registers. Thus, this number will be used in our state-swapping calculations. Moreover, ISDN processing is a non-numeric type of processing, i.e. the numeric processing registers will not be involved in the state-swapping operation. Since other system registers differ from processor to processor, and in order to make our findings machine-independent, these system registers will not be considered in our calculations. Hence, the state-swapping operation will only involve 64 memory references (32 STORE instructions to save the processor state of present task and 32 LOAD instructions to reload the next task state). Since the state-swapping operation involves not only the general-purpose registers but also other registers, this calculation will indicate the lower bound of the state-swapping overhead.

The execution of each of the ISDN messages, in terms of the total machine instructions and in terms of total memory references is measured from these two ISDN task programs. This is done by dynamically scanning the execution of each ISDN message being processed by each relevant task. The ratios of the state-swapping overhead to the processing of each ISDN message type used in both Q.931 basic call control and TEI assignment tasks are shown in Tables 4.1 to 4.4 respectively. The calculations based on the load delay technique have been obtained under the assumptions that the load delay slot is only one instruction, and that the compiler has a success rate of 90% to fill this slot with a useful instruction. The load delay slot, on the other hand, will be filled during the state-swapping operation with NOP instructions (i.e. another 32 NOP instruction will be added to 32 load and 32 store instructions).

Even when each input message buffer for each task has the same type of ISDN message (e.g. signalling, management), the amount of processing required will vary from one message

Table 4.1 Processing Ratio of the State-Swapping Operation to the Q.931 Messages in Terms of Total Memory Reference

Q.931 MESSAGE TYPE	WITHOUT DELAY LOAD	WITH DELAY LOAD
SETUP	0.6	0.9
CONNECT	7.1	9.6
RELEASE_COMPLETE	6.4	8.7
ALERT	12.8	16
DISCONNECT	5.8	8
RELEASE	4.9	6.9

Table 4.2 Processing Ratios of the State-Swapping Operation to the Q.931 Messages in Terms of Total Machine Instructions

Q.931 MESSAGE TYPE	WITHOUT DELAY LOAD	WITH DELAY LOAD
SETUP	0.17	0.3
CONNECT	1.45	2.6
RELEASE_COMPLETE	1.3	2.5
ALERT	2.7	4.8
DISCONNECT	1.4	2.6
RELEASE	1.23	2.4

Table 4.3 Processing Ratio of the State-Swapping Operation to the TEI Assignment Messages in Terms of Total Memory References

TEI ASSIGNMENT MESSAGE TYPE	WITHOUT DELAY LOAD	WITH DELAY LOAD
TEI ASSIGNMENT (ANY TEI)	1.4	1.9
TEI ASSIGNMENT (SPECIFIC TEI)	1.6	2.2
TEI REMOVAL (SPECIFIC TEI ONLY)	1.8	2.4

Table 4.4 Processing Ratios of the State-Swapping Operation to the TEI Assignment Messages in Terms of Total Machine Instructions

TEI ASSIGNMENT MESSAGE TYPE	WITHOUT DELAY LOAD	WITH DELAY LOAD
TEI ASSIGNMENT (ANY TEI)	0.38	0.7
TEI ASSIGNMENT (SPECIFIC TEI)	0.44	0.9
TEI REMOVAL (SPECIFIC TEI ONLY)	0.5	1

to another. For example, in the Q.931 signalling task, the amount of processing required by the SETUP message is different than the processing of the CONNECT message. Therefore, to find the ratio of the state-swapping processing to the different number of the same type of ISDN messages existing in the task input message buffer, an average amount of processing is calculated for the same type of different messages for specific services used by each task separately (tables 4.5 and 4.6).

As the task switching could occur when the task has already processed certain number of messages, the average message processing for both Q.931 and TEI messages is used to find the ratio of the state-swapping to the processing of the different number of these average ISDN messages. Figures 4.3 to 4.6 show the overhead percentage for both tasks in relation to the total memory references and to the total machine instructions processed. These figures illustrate the reciprocal relationships between the overhead ratio and the number of processed messages. It is clear that this overhead becomes larger as the number of messages processed by each task becomes smaller.

The exact number of the messages to be processed with each task is not known. But it is possible to measure the average processing for the state-swapping overhead based on the assumption that the probability of occurrence for all tasks, i.e. independent from the number of messages being processed by each task, is equal. For instance, The probability of running a task with 10 or 25 messages is the same. Table 4.7 shows these average processing overheads when the maximum number of messages processed by each task is 50. These measurements are in terms of the total machine instructions processed.

Table 4.5 Processing Ratios of the State-Swapping Operation to Average ISDN Messages in Terms of Total Memory References

MESSAGE TYPE	WITHOUT DELAY LOAD	WITH DELAY LOAD
Q.931	2.6	3.6
TEI ASSIGNMENT	1.6	2.3

Table 4.6 Processing Ratios of the State-Swapping Operation to Average ISDN Messages in Terms of Total Machine Instructions

MESSAGE TYPE	WITHOUT DELAY LOAD	WITH DELAY LOAD
Q.931	0.64	1.2
TEI ASSIGNMENT	0.43	0.9

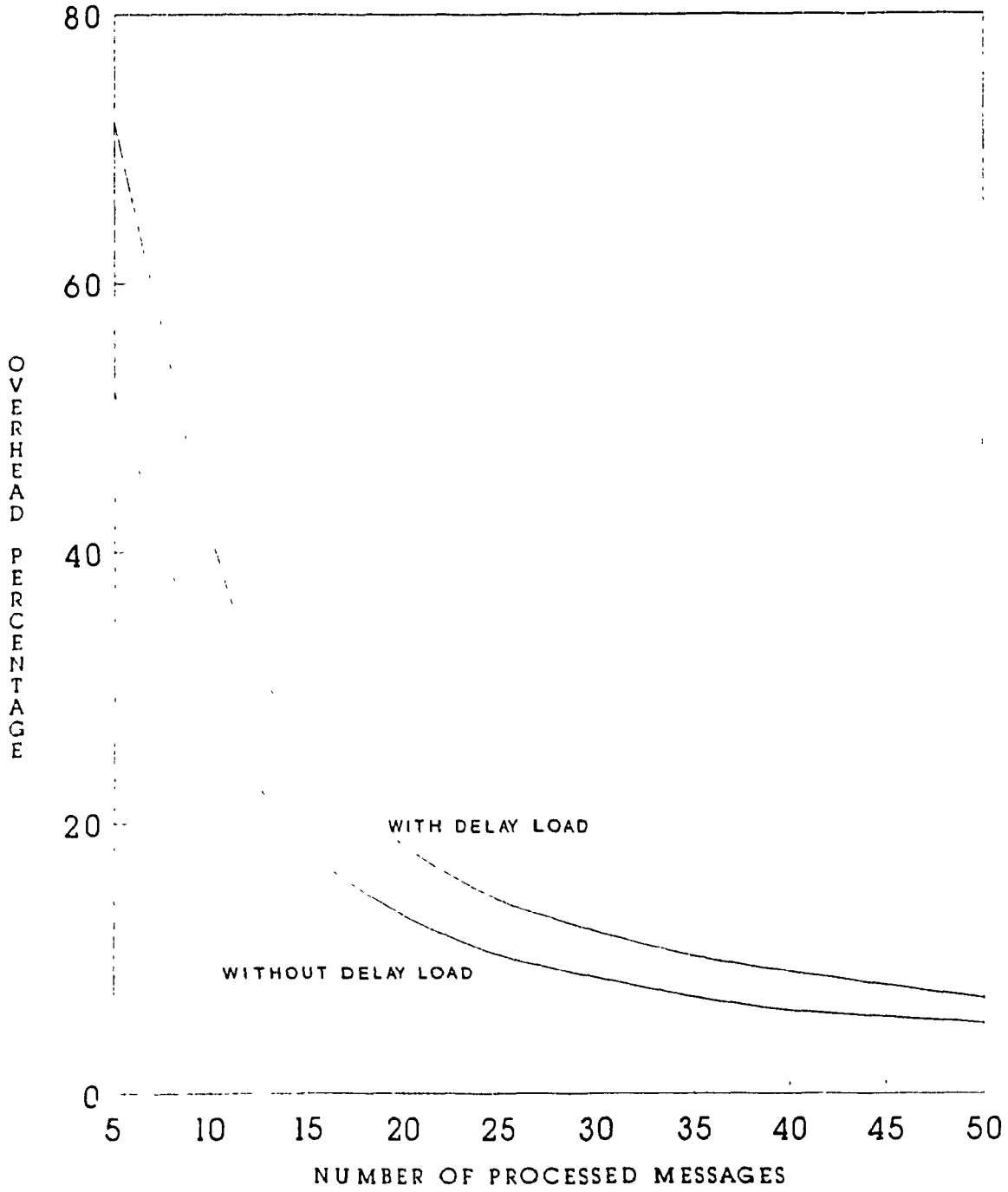


Figure 4.3 State-Swapping Overhead Processing to Total Memory References (Q.931 Task)

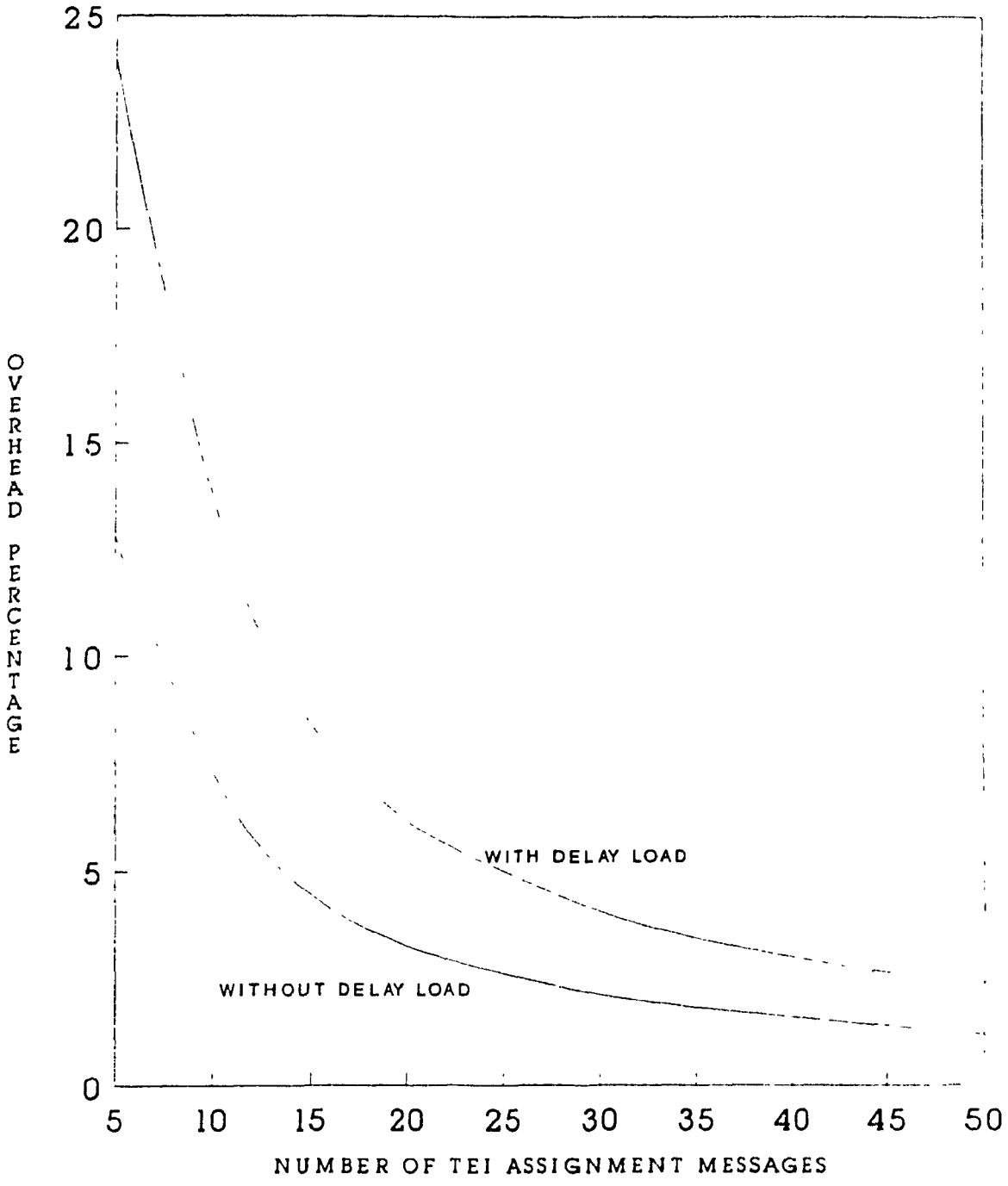


Figure 4.4 State-Swapping Overhead Processing to Total Machine Instructions (Q.931 Task)

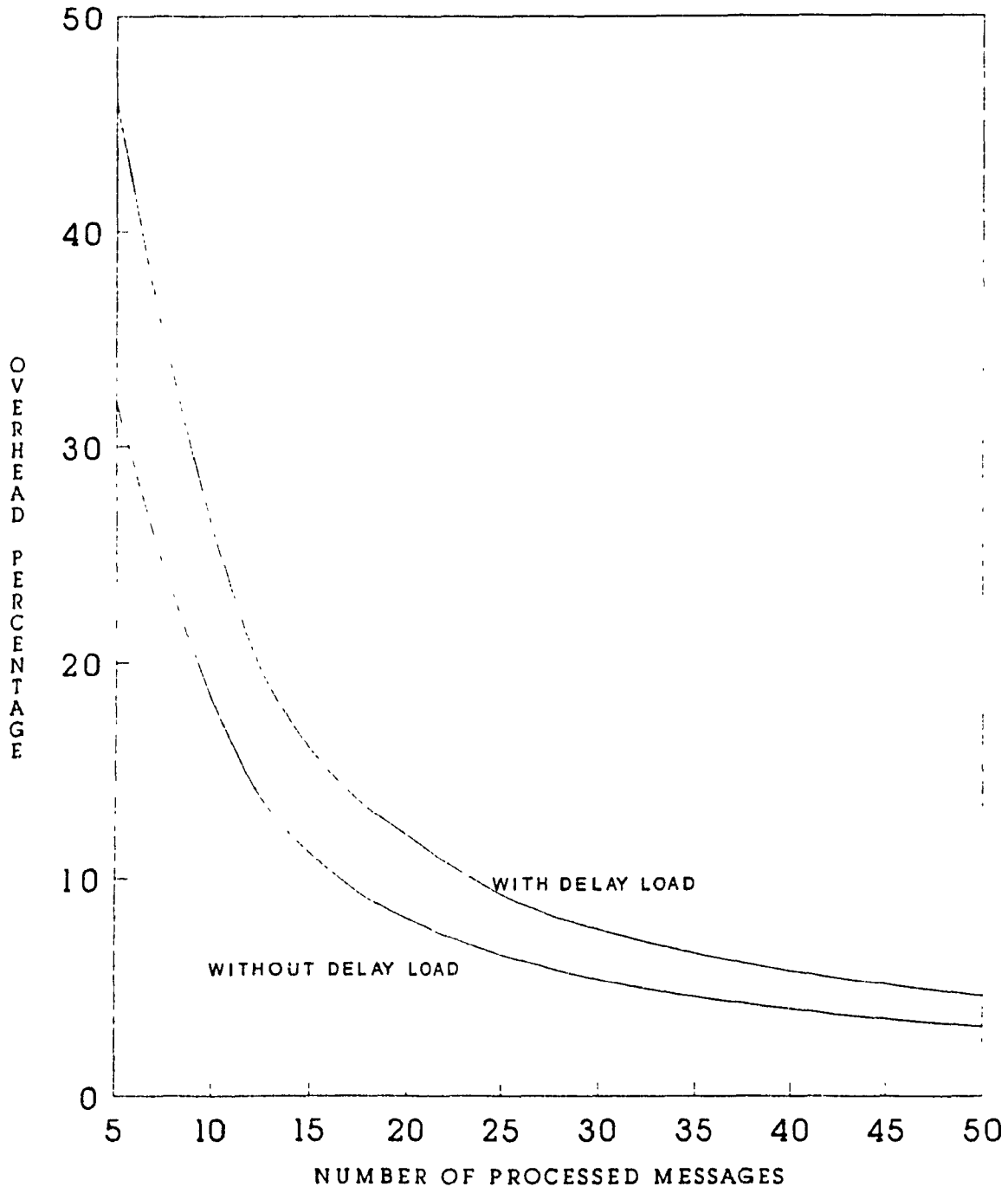


Figure 4 5 State-Swapping Overhead Processing to Total Memory References (TEI Task)

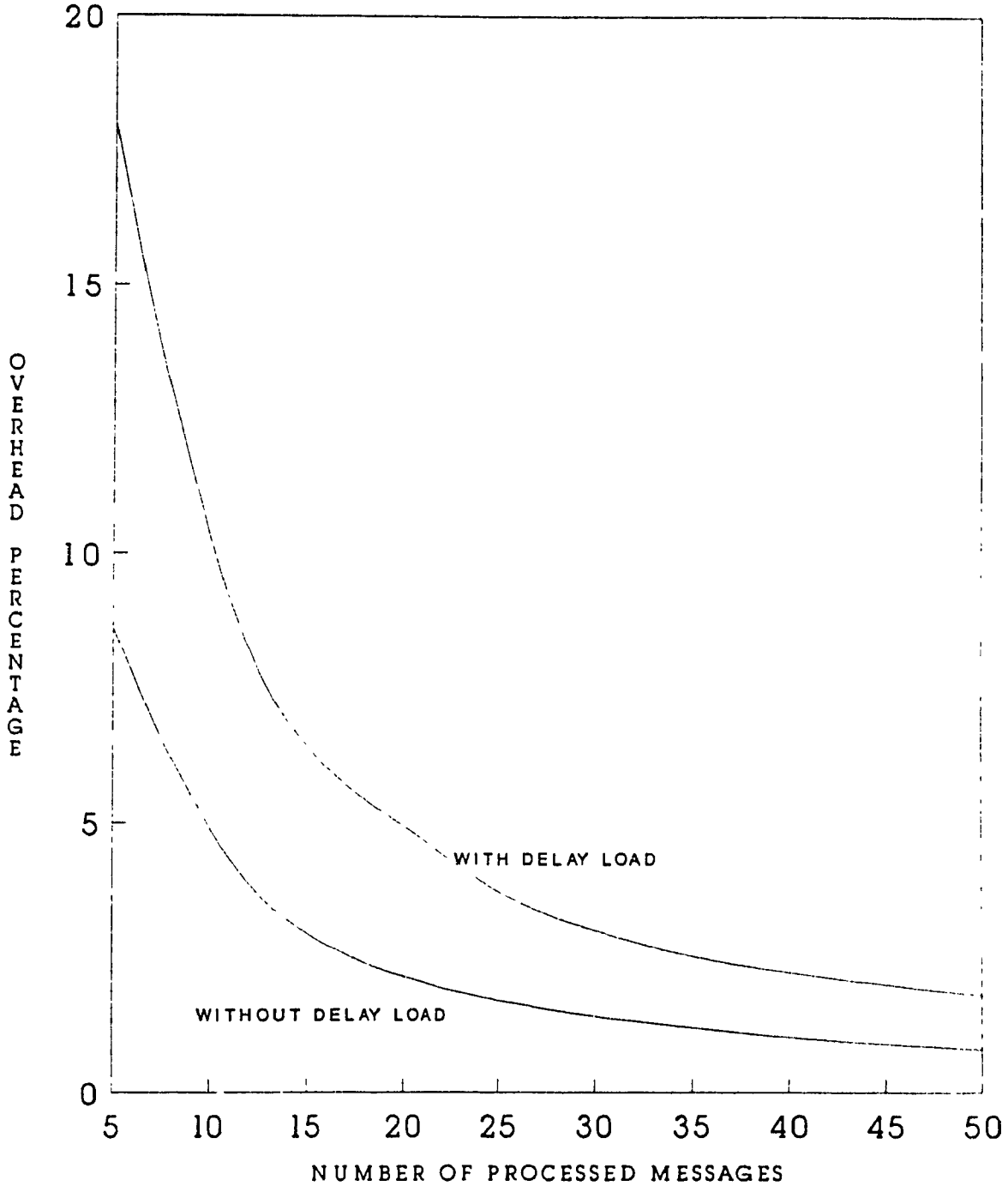


Figure 4 6 State-Swapping Overhead Processing to Total Machine Instructions (TEI Task)

Table 4.7 Average overhead processing for state-swapping operations in terms of total machine instructions

MESSAGE TYPE	WITHOUT DELAY LOAD	WITH DELAY LOAD
Q.931	9.2%	17.3%
TEI ASSIGNMENT	6.2%	13%

4.5. Discussions

The evaluation of the overhead processing associated with the state-swapping operation is important to determine whether the processor should have a special support to enhance this operation or not. As stated earlier, the method to measure the state-swapping overhead in real-time multi-tasking applications, such as ISDN, is complex and time consuming. The complexity arises from the needs to a real-time multi-tasking operating system running on a RISC based machine, and the ISDN applications tasks should be dynamically evaluated under a real ISDN environment. Thus, we have presented, in this chapter, an approach to estimate the processing overhead involved with the state-swapping operation. The method of measurement proposed in this approach to estimate the state-swapping overhead is based on comparing the overhead processing associated with the state-swapping operation to the ISDN application task processing.

The overhead processing of the state-swapping operation is measured based on an assumption that the processor has a typical RISC register set size, i.e., 32 registers. It is

assumed also that only these registers are involving with the state-swapping operation. Generally, other system registers are also involves, and hence the measurement will show only the lower bound of the processing of the state-swapping operation.

Measuring the processing required by each ISDN application task is more complicated than measuring the processing required for the state-swapping operation. Part of this complexity is coming from the fact that the task switching could occur in any time during the processing of the task, and hence there is no gurantee about the number of messages processed by each task when the task is in running state. The other part of this complexity is due to the processing required for ISDN messages which is varied from one message to another. For instance, the SETUP message of the Q.931 task requires amount of processing different than the CONNECT message.

In our approach, the above complexities are overcome by measuring the average processing required by each ISDN application task. This has been achieved by measuring the processing required by each ISDN message, and then taking the average processing required for these messages. As the processing of each application task is proportional to the number of the ISDN messages processed, then the average processing of each of these application tasks is measured with respect to the average processing of ISDN message. The state-swapping overhead can therefore be evaluated by comparing the processing of the state-swapping operation to the ISDN application task processing which processes different number of average ISDN messages. Clearly, the state-swapping overhead will be larger as the number of average ISDN messages, processed by each application task, are less. By assuming that the probability of occurrence of all application tasks which process different number of messages is equal, then the average of state-swapping overhead is evaluated by comparing the state-swapping operation processing to the average ISDN task processing.

The processing of each ISDN application tasks is evaluated in terms of total machine instructions and total memory references processed during the dynamically executed of each application task. Our measurements are also includes the impact of the delay load technique since most of today optimizing compilers use this technique in their design.

The results show that using an optimizing compiler with a RISC-based processor implementing 32 general purpose registers to process ISDN applications, will significantly increase the state-swapping overhead processing. The overhead increases as the number of messages processed by each task becomes smaller. The average processing for the overhead is found to be significant in comparison to the total machine instruction processed (about 17% for the Q.931, and 13% for TEI assignment tasks) [12]. Clearly, the overhead will become larger when it is compared with the total memory references. Hence, the elimination of the state-swapping overhead will therefore reduce the processor-memory traffic associated with the state-swapping operation, and enhance the processor performance. To this end, in the next chapter we will study the impact of using a MRS structure to reduce or eliminate the state-swapping processing for ISDN applications.

CHAPTER 5

MULTIPLE REGISTER SET IN RISC-BASED ARCHITECTURE USED FOR ISDN PROCESSING

This chapter investigates the possible reduction in the large processing overhead which is associated with the state-swapping operation. Our main emphasis will be on the use of the Multiple Register Set (MRS) structure to reduce the state-swapping overhead. At first, the MRS structure is studied and its necessary hardware support is highlighted. Then the possible tasks' states allocation to the MRS, based on the priority approach, is considered in detail. The possible performance improvemen' by using MRS in RISC architecture is evaluated. The impact of using MRS for ISDN processing is also discussed. Finally, the proposed RISC-based architecture using MRS is presented.

5.1. Multiple Register Set Structure

In processors which use a single register set, each task switching requires a state-swapping to save the processor state information of the currently running task into the external memory, and to load the state information of the next-to-run task from the memory into the processor registers. Examples of processor state includes Task State Segment (TSS) [7] or Process Control Block (PCB) [8]). On the other hand, in processors which use the multiple register set (MRS) structure [88], the internal registers are partitioned into a number of register sets, each set contains the processor state information of one task. At any given time, only one of these sets corresponding to the running task is active, while the others are kept in a non-active state. A task register is used to hold the address of the register set of the active task (Figure 5.1). In addition, the task register contains the pointer to the running task descriptor which refers to the location where the task state should be saved in the external memory. The

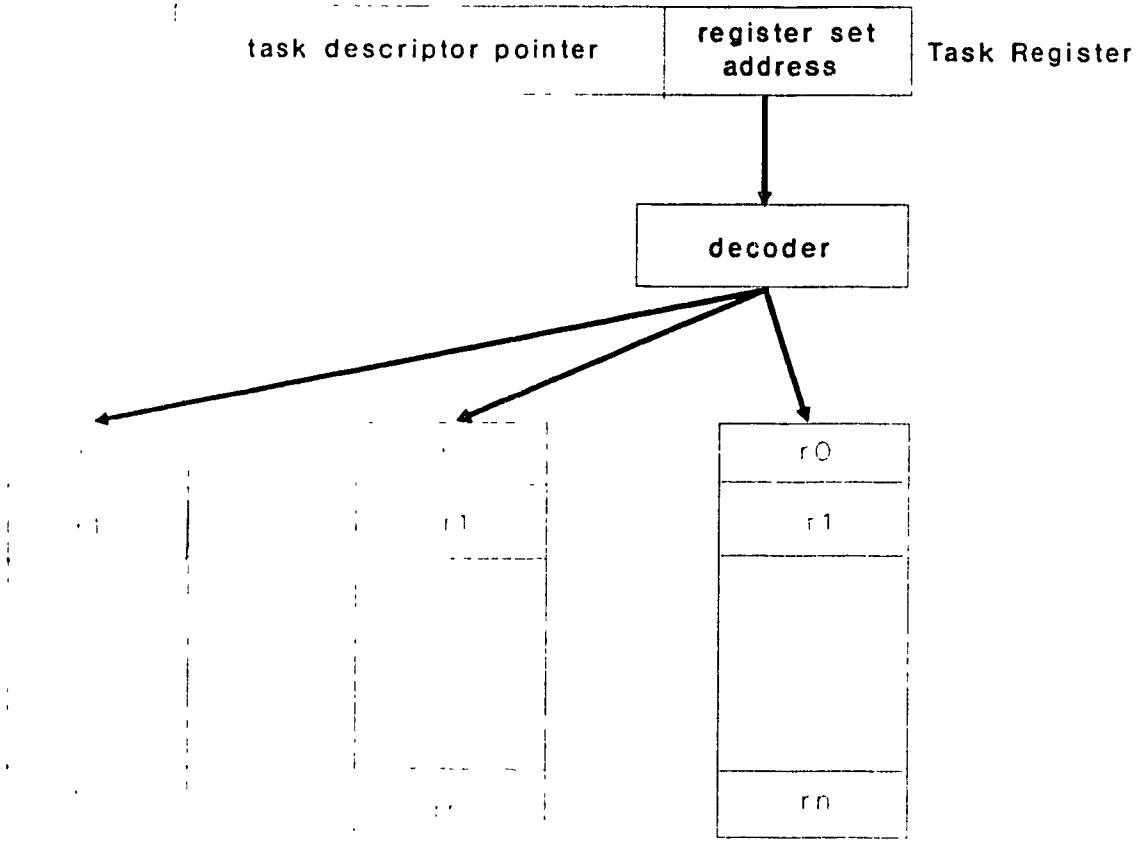


Figure 5.1 MRS General Structure

content of this task register is updated only by the operating system.

Ideally, an MRS-based processor is able to store state information of all tasks and eliminate the state-swapping processing overhead, completely. However, the VLSI implementation imposes a limit on the number of register sets, accommodated by the processor. This constraint on the number of register sets will require the use of a scheme to distribute the tasks' states between MRS and the external memory.

5.2. Allocation of Tasks' States to MRS

Allocation of the tasks' states to MRS depends on the number of register sets in the MRS structure, and on the total number of tasks. Tasks running on any processor include both operating system and applications tasks. The number of tasks and their organization vary from one system to another, depending on the complexity of the application and the services provided by the operating system.

In applications where the total number of tasks is less than or equal to the number of register sets, all task states can be resident on-chip. In such applications, the state-swapping overhead is completely eliminated and the processor performance is maximized. If the number of on-chip register sets is smaller than that of tasks, some tasks' states should reside in the external memory. In this case, the processing overhead for the state-swapping can be reduced, but cannot be totally eliminated. The amount of reduction in the state-swapping processing overhead will depend on reducing the number of times that is required to move the state of the currently running and next-to-run tasks to/from MRS.

There are four situations which can exist in swapping any task state between MRS and the external memory. These are:

i. The next-to-run task state which has already existed on MRS. The state-swapping operation associated with the task switching will not require to load the state of the next-to-run task to MRS.

ii. The next-to-run task state is not on MRS. In this case, the task switching is required to load the task state to MRS.

iii. The currently running task can keep its state on MRS after the task execution is terminated. Task switching will not be involved for storing this task state back to the external memory.

iv. The currently running task cannot keep its state on MRS after this task is terminated. Task switching will be required to store the task state to the external memory.

In general, managing the tasks' states allocation to MRS makes use of an approach based on a certain criterion such as the frequency of task usage or their priorities.

When a task is executed, it is important to know whether its state is already on MRS or in the external memory. Knowing this information will help the operating system to make decision about whether it is required to load the state of the next-to-run task to MRS or to resume the execution of the task immediately if the task state is already on MRS. To achieve this, it is important to have a suitable support. A possible and simple support is to use a bit within the task descriptor, called "exist bit". As any task becomes a running task, its descriptor will be checked by the operating system. The exist bit will be tested in order to determine whether this task state has been already on MRS or not. The exist bit has to be tested for every task switching. This processing does not clearly exist with the SRS structure and hence it is considered as a processing overhead associated with MRS, albeit this overhead is very small.

In addition to the exist bit, each task descriptor should also have other fields (Figure 5.2). The information provided by these fields is necessary for tasks' states allocation to MRS. The priority value and the temporarily raised priority (TR) fields are used in each task descriptor to facilitate the allocation of tasks states to MRS when the priority-based approach is used for task allocations. Details of these fields will be discussed in the next section. The MRS address field is to be used to locate the MRS register set where the next-to-run task has already stored its state. The MRS address will be loaded to the register set address field of the task register so it

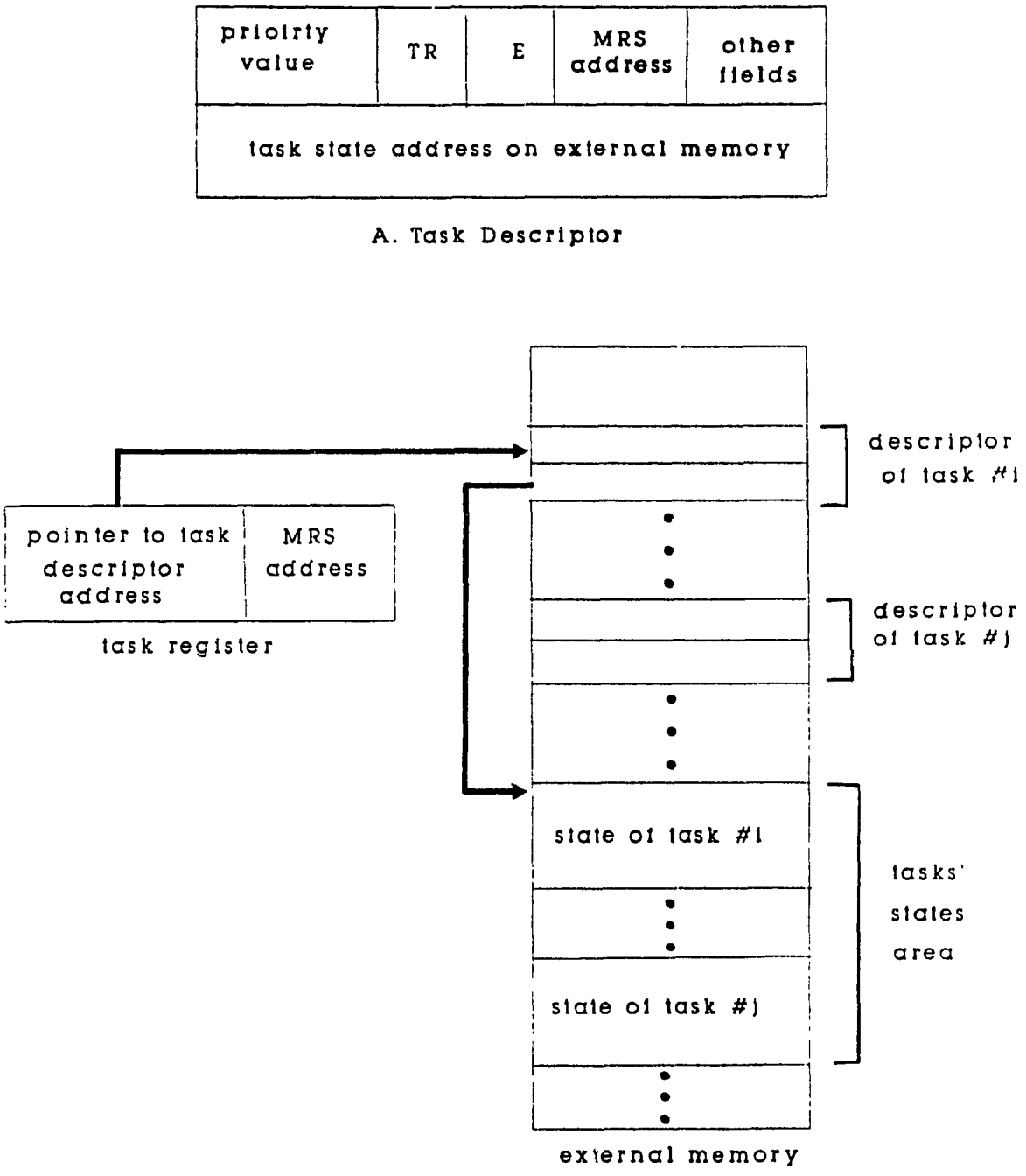


Figure 5.2 Task's Descriptor Organization

can activate this set only. The use of the MRS address field is associated with the use of the exist bit (E) field. That is, when the E field indicates that one set already exists on MRS, holding the state of the next-to-run task, the MRS address field will show which set of MRS has been already used by that task.

In some cases, the state of the currently running task is required to be saved in the external memory at the end of the task execution. In such a case the operating system must know the address of the first location where the state of this task is stored in the external memory (Figure 5.2). This is achieved by including within the task register, the pointer to the task state location in the external memory. This kind of support to the MRS structure is similar to what is being used with SRS structure [13,36]. Generally, tasks' descriptors can have other information field such as information required for the memory management. However, the support for these information will not be considered in this work since the attention is given only to the support of the MRS structure.

Only the use of the exist bit, MRS address, TR, and priority value within the task descriptor are an extra information required to be supported with the MRS structure compared to the SRS structure. The processing involved with testing these fields is simple and it is not a time consuming process. Hence, the processing required for supporting the MRS structure is small and it is not largely exceeding what has been already used with the SRS structure. This feature is an encouraging sign to investigate further the approach for managing the tasks' states allocation to MRS.

5.3. Characteristics of An Approach for Assigning Tasks' States to MRS

To assign tasks' states to MRS, an approach has to be employed. The potential approach should be capable of achieving the following aspects:

i. Minimizing the processing time associated with the execution of the strategy: The use of an approach in assigning the states of the tasks resident on the external memory to MRS

will contribute to some processing overhead. This overhead will be added to the processing overhead of the state-swapping operation. Obviously, as the implemented approach gets more complex, the processing overhead for the state-swapping operation will get larger. As the processing overhead gets larger the performance improvement achieved by reducing state-swapping overhead will become insignificant. It is also important to mention that since MRS is intended to be used for real-time applications, then the processing time spent by the processing of the proposed approach and the state-swapping operation should be as minimum as possible in order to reduce the tasks responses time.

ii. Minimizing the occurrence of the state-swapping operations: The state-swapping operations involved with MRS should be less than that of SRS. The approach should be capable of retaining the tasks' states on MRS. This will guarantee the performance improvement by reducing as much state-swapping as possible with the tasks whose states retained in the external memory.

5.4. The Priority-Based Approach

Different possible approaches can be considered for tasks' states allocation to MRS. Implementing an approach based on task frequencies, for example, will need a special support, such as counters, to keep track of the frequency of each task. Such strategy will add large overhead to the processing of the state-swapping operation due to incrementing the counters and comparing the frequencies of all tasks' states to find the one which is running more than the others. The frequency-based approach is, however, similar to those approaches which have been used in placement algorithms for virtual memory support [13]. Moreover, such an approach will not be appropriate when some tasks, which are less importan than the others, run more frequently. These less important tasks will have a faster responses since their states are already on MRS. Since in this study we consider MRS for real-time applications, it is important to keep

more important tasks states on-chip in order to speed up their responses.

Another possible approach can be based on the use of the tasks' priority as a mean to allocate tasks' states to MRS [15]. As the priority for each task represents its order of importance, then allocating these tasks to MRS will allow them to respond faster. This can be considered as an important issue in real-time systems. Moreover, using the priority level of each task will provide an easy way to allocate task state to MRS. Generally, real-time multi-tasking operating systems support different priority levels to distinguish between the application and operating system tasks. For example, the priority level of 128 is defined in iRMX operating system [36] to determine the boundary between operating system and application tasks, where zero level is the highest level for operating system tasks and 128 is the highest level for applications tasks. This boundary between the operating system tasks and the application tasks will simplify the implementation of the priority-based approach. This boundary will allow us to implement the priority-based approach by assigning either the state of the high priority operating system tasks or the high priority application tasks to MRS. We will elaborate further on this issue later. The use of MRS-based processor for real-time application, specifically supported with a priority-based approach, is not well addressed in the literature. This motivates us to study this approach in more details.

The priority-based approach performs poorly when an operating system with large number of tasks is used since it will be possible to allocate only a small number of operating system tasks to MRS. In this case, the application tasks will not have a chance to be allocated to MRS since the operating system tasks always have a higher priority than the application tasks. However, in specialized applications, it is more suitable to have a small as well as a fast operating system. This will provide these applications with the speed which they require and will avoid using any unnecessary support which might be provided by the large operating systems and might not be useful for this type of applications. This will also help us to assign more operating system tasks to MRS and improve the performance. Therefore, for special purpose applications, such as ISDN,

the chances of assigning tasks' states to MRS will be higher than those of other general purpose applications. In addition, the recent advances in VLSI technology, which make it possible to integrate more transistors on chip, will allow us to provide extra support by having, for example, more register sets in MRS.

In the next section, the priority-based approach will be studied based on both possible strategies, i.e., assigning either the high priority operating system or application tasks to MRS. The complexity of each of these two strategies will also be investigated. This study will take into account the type of tasks used in different systems, i.e., dynamic or static tasks. Implementing these two strategies will be evaluated in terms of complexity and performance.

5.4.1. Strategy for Placing High Priority Tasks of Operating Systems

5.4.1.1. Dynamic System

A dynamic system is defined as a system which contains tasks that can be deleted after their creation [13]. Once a task is created and its state is changed to "running", it should be subsequently assigned to one of the MRS sets. At the initialization phase of operation, all sets within MRS structure are empty and hence each running task will be assigned to one set of MRS.

After all sets of MRS are filled with states of different tasks, then the allocation of each running task to MRS should be managed in a manner to keep only the highest priority tasks on MRS. The assignment of these tasks can occur in one of the two following cases:

i. If the next-to-run task has a higher priority than all the tasks resident on MRS, then the state of this task will be allocated to the set which is occupied by the task with lowest priority. To achieve such tasks allocation, the priority value of the next-to-run task should be compared with the priority values of all tasks which reside on-chip. Clearly, this searching process is a time consuming operation.

ii. If the next-to-run task has a priority lower than or equal to the priority of those tasks resident on MRS, then a register set which is allocated to the task with the lowest priority

should be given to the next-to-run task.

In order to reduce the processing which is associated with the searching operation of the task on MRS with lowest priority, the lowest priority value should be stored after the first search and should be used in comparison whenever the task switching occurs. This will allow the next-to-run task to compare its priority value with that of the task resident on MRS which has the lowest priority. When the next-to-run task has an equal or less priority value than the MRS task with the lowest priority value, the searching operation will be eliminated. In the situation when the next-to-run task has a higher priority value than the lowest priority value of the MRS task, then the searching operation will be necessary (as shown in point i). The searching operation will be also necessary when the MRS reside task with the lowest priority is deleted.

Other processing associated with implementing this priority-based approach is associated with the operation of keep tracking the location of the register set of the task with the lowest priority. This set cannot be assigned to a fixed register set due to two reasons. Firstly, the nature of the dynamic tasks which can cause deletion of a task with the lowest priority among the tasks resident on MRS. Secondly, the allocation of the task state with higher priority to the set on MRS which is occupied by the task with the lowest priority to MRS. This will require saving the set of lowest priority in external memory and using this set by the high priority task. After saving the set with lowest priority to external memory, it will be necessary to scan all tasks on MRS to search for the set occupied by the task with lowest priority. That is, the set used by the task with lowest priority will be dynamically change its place around MRS. Then the search for the set occupied by the task with the lowest priority will be frequently performed.

The "task starvation" problem is usually handled by temporarily raising the priority of the task with the low priority in order to give such a task a chance to run [13]. In the situation where the tasks with temporarily raised priority become the running tasks, the priority strategy will face a problem. This problem occurs since it is required to give one of the register sets, which is

assigned to the lowest priority task, to the task with the temporarily raised priority. As the task with the temporarily raised priority has a higher priority value than the other tasks, then the task state with the temporarily raised priority will reside on the on-chip set instead of swapping its state to the memory after its termination. Tasks with a "real" high priority will, therefore, not be able to reside on the on-chip sets.

To overcome this problem, immediately after the processing of the task with temporarily raised priority is completed, its state should be saved back to the memory. To achieve this, it is required to detect whether or not the task priority has been temporarily raised. Using a "temporarily raised bit" or TR in the task descriptor will allow us to detect if the task priority is raised temporarily or not. The operating system will set this bit when the task priority is temporarily raised. As the next task switching occurs, this bit will be tested in order to determine whether the register set of the lowest priority task belongs to task with temporarily raised priority or not.

The complete mechanism is shown in Figure 5.3. Clearly, if the next running task is on-chip and its priority has not been temporarily raised, the processing overhead with the strategy execution will be insignificant. In any other situation, the processing overhead of this strategy will be large. This large processing overhead due to the execution of the strategy with the MRS structure will increase the amount of processing involved with the state-swapping operation which is intended to be reduced by using MRS.

5.4.1.2. Static System

Any system contains tasks which are kept for ever once they are created, is considered to be static. This characteristic of task operation will not allow any on-chip task to be deleted in any stage of operation, and hence, all the complexities arise due to tasks deletion associated with the priority strategy of dynamic systems can be eliminated. This will make the strategy for the static systems as a subset of the one for the dynamic systems. This is shown in

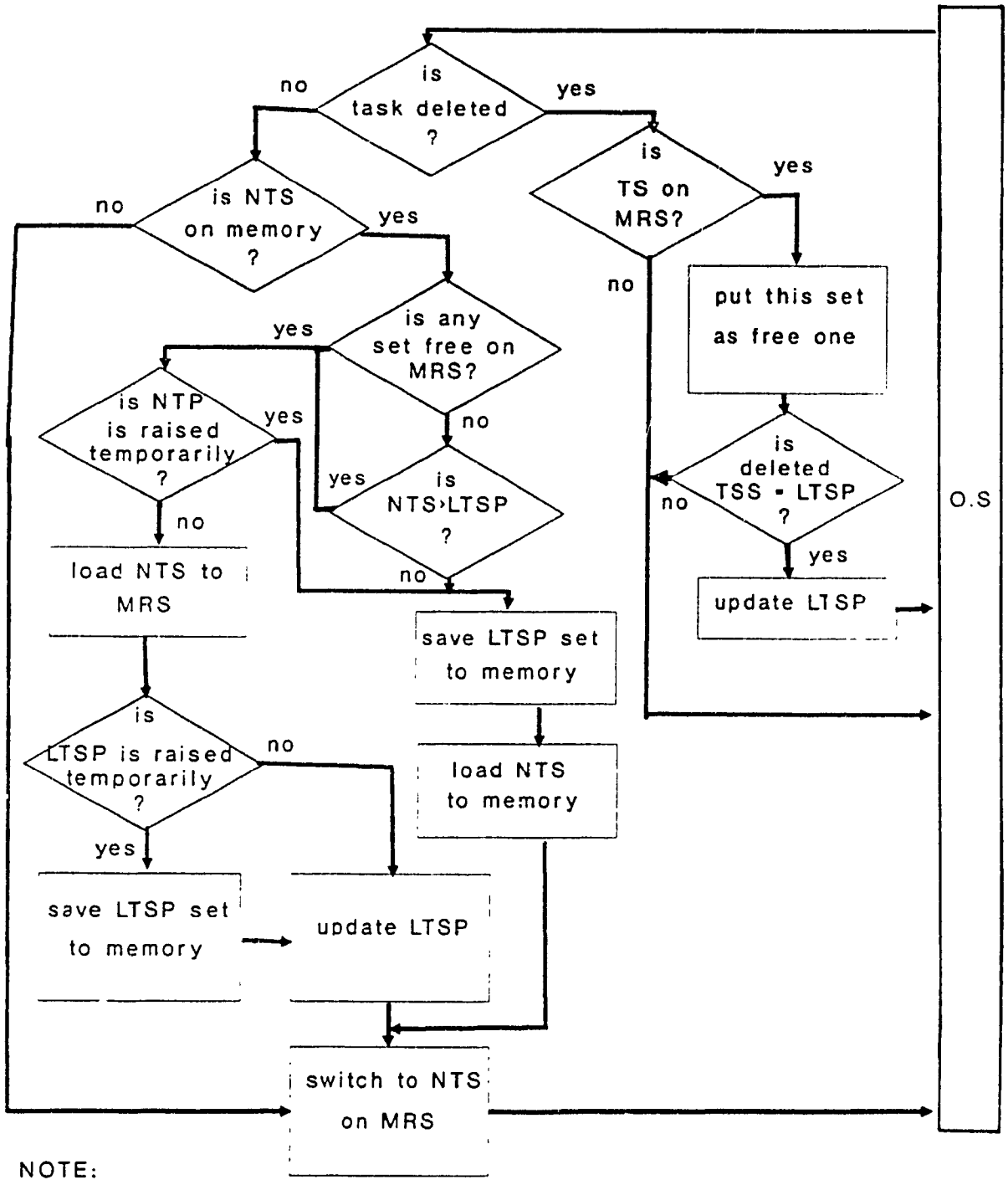
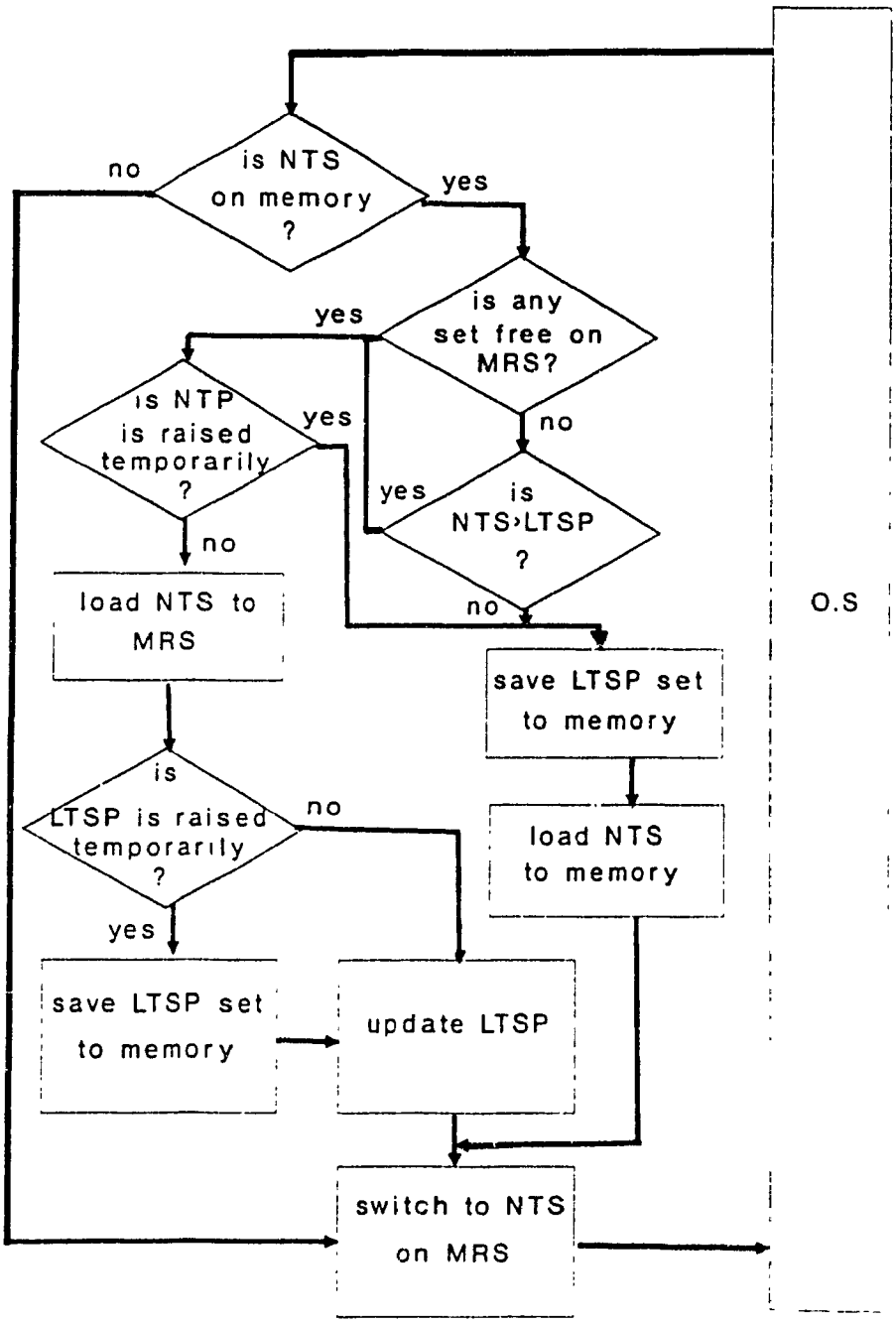


Figure 5.3 Priority Strategy for Dynamic Systems

Figure 5.4.

Since the tasks priorities in static systems were known during system initialization, then the assignment of highest priority tasks to MRS can be moved to the system initialization phase. Assigning the tasks with highest priorities to the on-chip sets during the initialization phase will eliminate the search operation for the on-chip task state with the lowest priority, because no deletion will occur to on-chip task states when the system is operating. The elimination of the searching operation reduces further the processing involved with the implementation of the priority strategy for static systems. It is always necessary to use one set of MRS to allow storage of the state of the next-to-run task if its state does not already exist on-chip. In implementing the priority strategy with static systems, one fixed set of the on-chip sets can be used as a "working set" for all those tasks which do not have already an assigned set on MRS. One fixed set assigned as working set was not simple with dynamic systems because deletion could happen to any tasks' states which exist on any on-chip set.

The task descriptors used to support the priority strategy with static systems are similar to those of dynamic systems with the exception of the temporarily raised bit (TR) and the priority value fields. The TR bit will not used here, since all tasks' states with high priority are assigned during the initialization time and the temporarily raised task will be treated as any other low priority tasks which do not have a permanent assigned register set on MRS. Hence, tasks with temporarily raised priority will be saved automatically back to the external memory when the next-to-run task, which is not resident on the MRS, is loaded to the working set. The priority value field on task descriptor will not required since there is no need with static system to find an MRS set which is occupied by a task with the lowest priority. All these reductions in the priority strategy with static systems will make the implementation of the strategy very simple as shown in figure 5.5.



NOTE:

NTS: next running task state
LTSP: lowest task state priority
NTP: next task priority

Figure 5.4 Priority Strategy for Static Systems

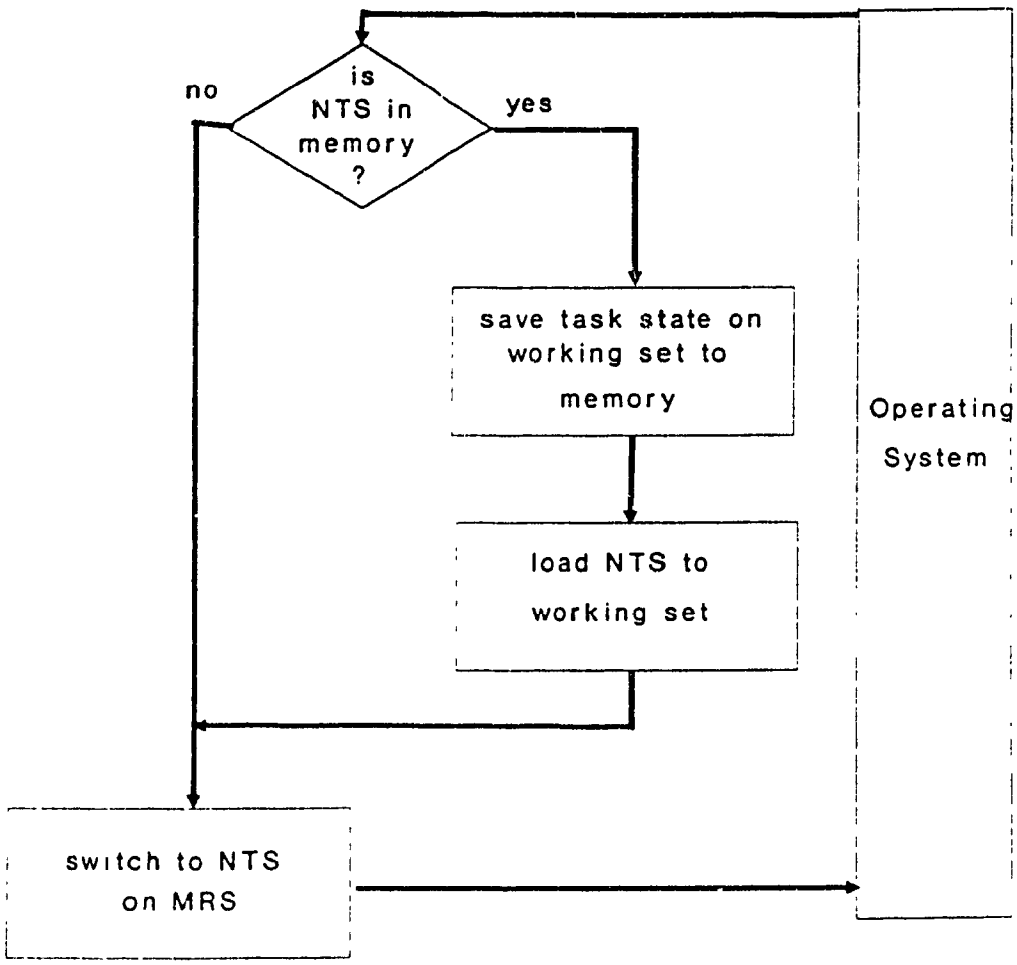


Figure 5.5 Priority Strategy for Static System
(Assigning Task States in System Initilization)

5.4.2. Strategy of Placing the High Priority Application Tasks

In this approach, the tasks with high priorities within the application tasks, will be loaded first to MRS. This approach is most effective when few application tasks are controlled by the general purpose real-time multi-tasking operating system which generally have many operating system tasks, such as iRMX operating system [36].

It is useful to mention here that the drop in the processors cost during recent years has enabled us to connect many processors in different configurations such as cube, mesh, etc. Therefore the application tasks can be distributed on these processors, and hence, each processor will process only a few of them.

The basic mechanism of this approach is similar to the approach of placement of the high priority tasks of the operating systems, except that the high priority of the application tasks will be kept on MRS. When the number of on-chip sets is larger than the number of application tasks, all the application tasks can be resident on MRS. In addition, the highest priority tasks of the operating system can also be used to fill the free on-chip sets (if exist). One set of MRS will be required to be used for the running tasks which do not have their states resident on MRS. Clearly, processing any task, i.e., operating system or application, which does not have its state already allocated to one set of MRS will require a state-swapping operation similar to the one used with SRS.

5.5. Performance Evaluation

Generally, the performance evaluation of RISC architecture is carried out in terms of the capability of this architecture to reduce processor-memory traffic [84,85]. The reduction of the processor-memory traffic is emanated from the execution of the instructions within the processor, and hence, form the reduction in memory access. The reduction of the state-swapping operation will contribute to reduction of the overall processor-memory traffic, and hence will increase the processor performance. However, it is not practical to evaluate the performance improvement

that can be achieved using MRS, in terms of total processor-memory traffic, for general real-time multi-tasking applications due to the following reasons:

i. The processor-memory traffic which is generated due to the application and operating system tasks processing, might differ from one application to another.

ii. The processor-memory traffic corresponding to the state-swapping operation is also different from application to application due to the task frequency and due to the number of registers required to be saved/reloaded being different.

iii. The usage of optimizing compilers is another factor since using these compilers will significantly reduce the processor-memory traffic for the application and system task processing. This amount of reduction depends on the compiler design.

However, this measure can be only possible for a specific real-time multi-tasking application where it may run under a certain RISC processor architecture, as will be shown in Section 5.7.

Since the main reason behind the MRS structure is to reduce the processor-memory traffic, which is generated due to the state-swapping operation, it is possible then to measure the performance improvement by evaluating the processor-memory traffic reduction with MRS compared to the SRS structure. However, this relative performance improvement can be evaluated from two perspectives: the size of MRS (i.e., the number of sets which exist within MRS), and the strategy which is used to allocate the tasks to MRS.

5.5.1. Effects of MRS Size

The contribution of the number of the on-chip sets and the number of tasks in reducing the processor-memory traffic can be evaluated theoretically under certain assumptions. This performance evaluation can be achieved by comparing the processor-memory traffic generated in both MRS and SRS. In SRS, the processor-memory traffic will be:

$$M_{SRS}=2N_RT, \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5.1)$$

where:

N_R : the number of registers in the set,

T : the task switching frequency.

As task switching requires saving of the internal registers to memory and reloading the next running task state to the processor internal registers, then twice the number of registers should be used in calculating M_{SRS}. However, we assume that the number of registers to be saved/reloaded for both present and next-to-run tasks are equal.

For MRS, the processor-memory traffic for the state-swapping operation, is only affected by the number of tasks whose states cannot be residing on any of the processor sets. Moreover, the operational characteristics of the total tasks running on the processor could increase or decrease this traffic. For instance, when the tasks whose states are not resident on the on-chip sets run less than those retained on MRS, the performance improvement by MRS will be enhanced larger, and visa versa. Therefore, the processor-memory traffic due to the state-swapping operation, with the existence of MRS, can be measured by the following empirical formula [13]:

$$M_{MRS}=2TN_R\frac{PX}{(1-P)Z+PX}, \quad \ldots\ldots\ldots\ldots\ldots\ldots (5.2)$$

where:

N_R : number of registers in each of the register set,

X : the number of tasks whose states are resident in the external memory,

Z : the number of on-chip register sets,

P : the probability that a task having its states resident in the external memory is executed.

Noted that the sum (Z+X) represents the total number of tasks.

By assuming that both MRS and SRS are processing the same application, then the processor-memory traffic for MRS compared to SRS will be as follows:

$$\frac{M_{SRS}}{M_{MRS}} = \frac{(1-P)\,Z+PX}{PX} \quad \ldots\ldots\ldots\ldots (5.3)$$

The number of registers N_R is eliminated from Equation (5.3) since we assume that the number of registers within a set of MRS is equal to the number of registers in SRS. Figure 5.6 shows the ratio of the SRS processor-memory traffic to the MRS processor-memory traffic, for different size of MRS, versus the number of tasks retained on the external memory. The results of figures 5.6 (A, B,& C) show that with the MRS size is equal to 8 the reduction of the processor-memory traffic enhances when the number of tasks resident on the external memory is less than or equal to 10. When the probability of the tasks resident on MRS to be run is more than the probability of those tasks resident on the external-memory, then the performance will be further improved. For example, when the total tasks used in a system which has 13 tasks, i.e., 8 assigned to MRS and 5 assigned to the external memory, then the processor-memory traffic for SRS will be about 8 times larger than the one with MRS (Figure 5.6A). This traffic is measured when the tasks resident on MRS have the chance to run four times more than those on the external memory. The processor-memory traffic will be reduced by a factor of two when the probability of running tasks existing on MRS is one quarter of those tasks resident on the external memory (Figure 5.6C).

Generally, increasing the number of register sets will help reducing the processor-memory traffic generated by the state-swapping operation. The large number of registers within each set will increase the effectiveness of optimizing compilers in assigning operand to the on-chip registers, and hence will reduce the processor-memory traffic. However, due to VLSI limitations

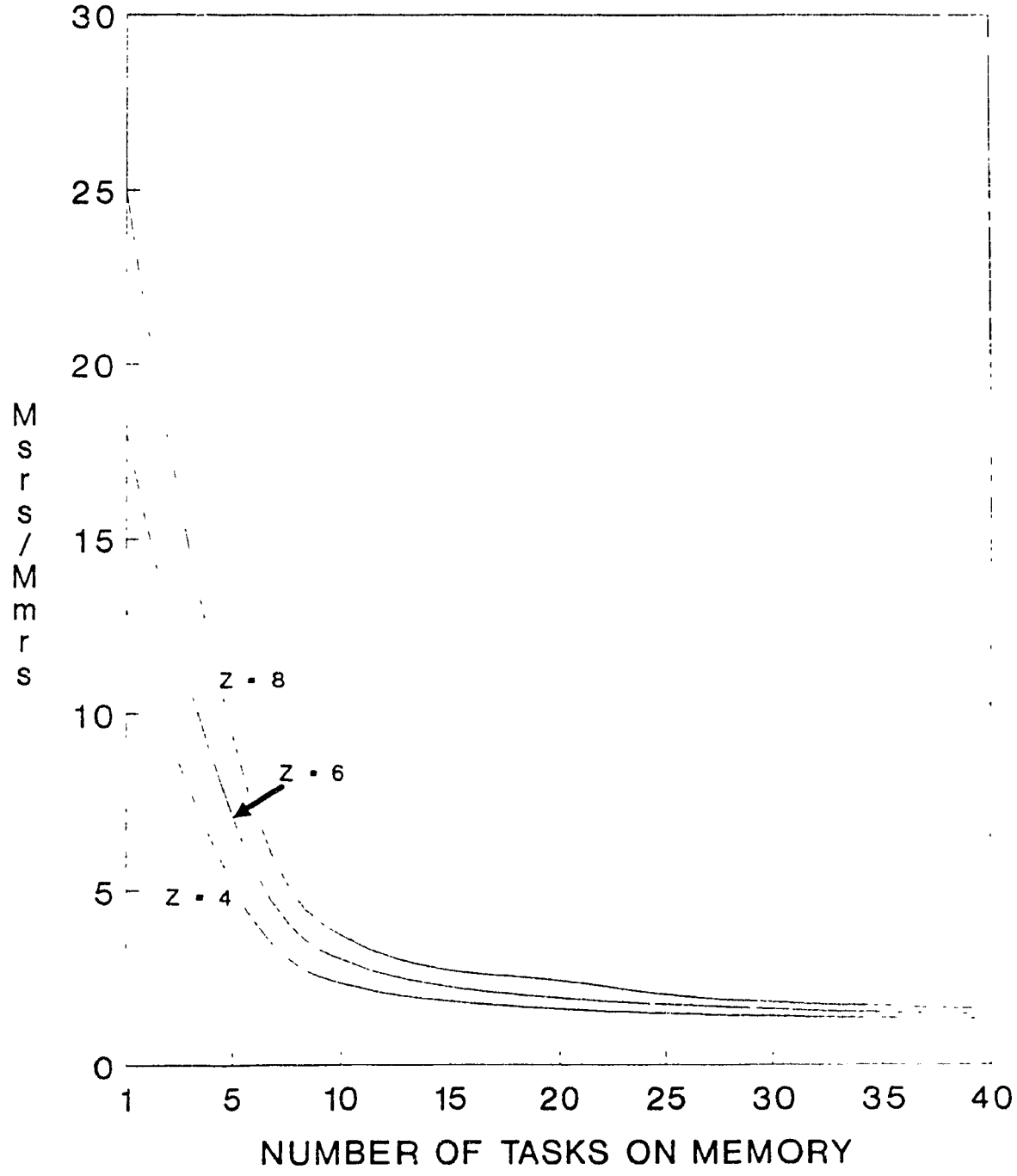


Figure 5.6a Comparing Performance of Mmrs with Msrs when P = 0.25

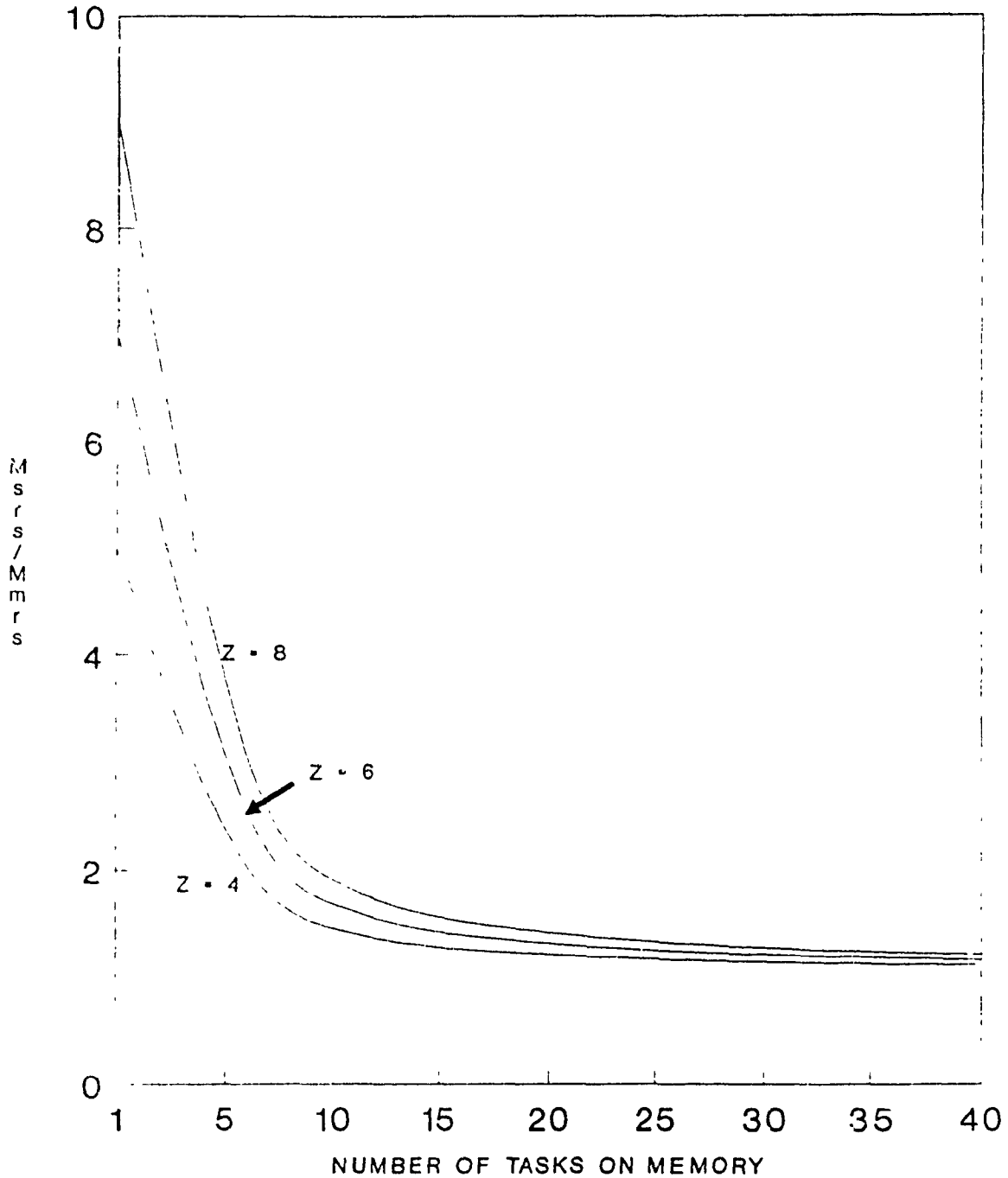


Figure 5.6b Comparing Performance of Mmrs with Msrs when P = 0.5

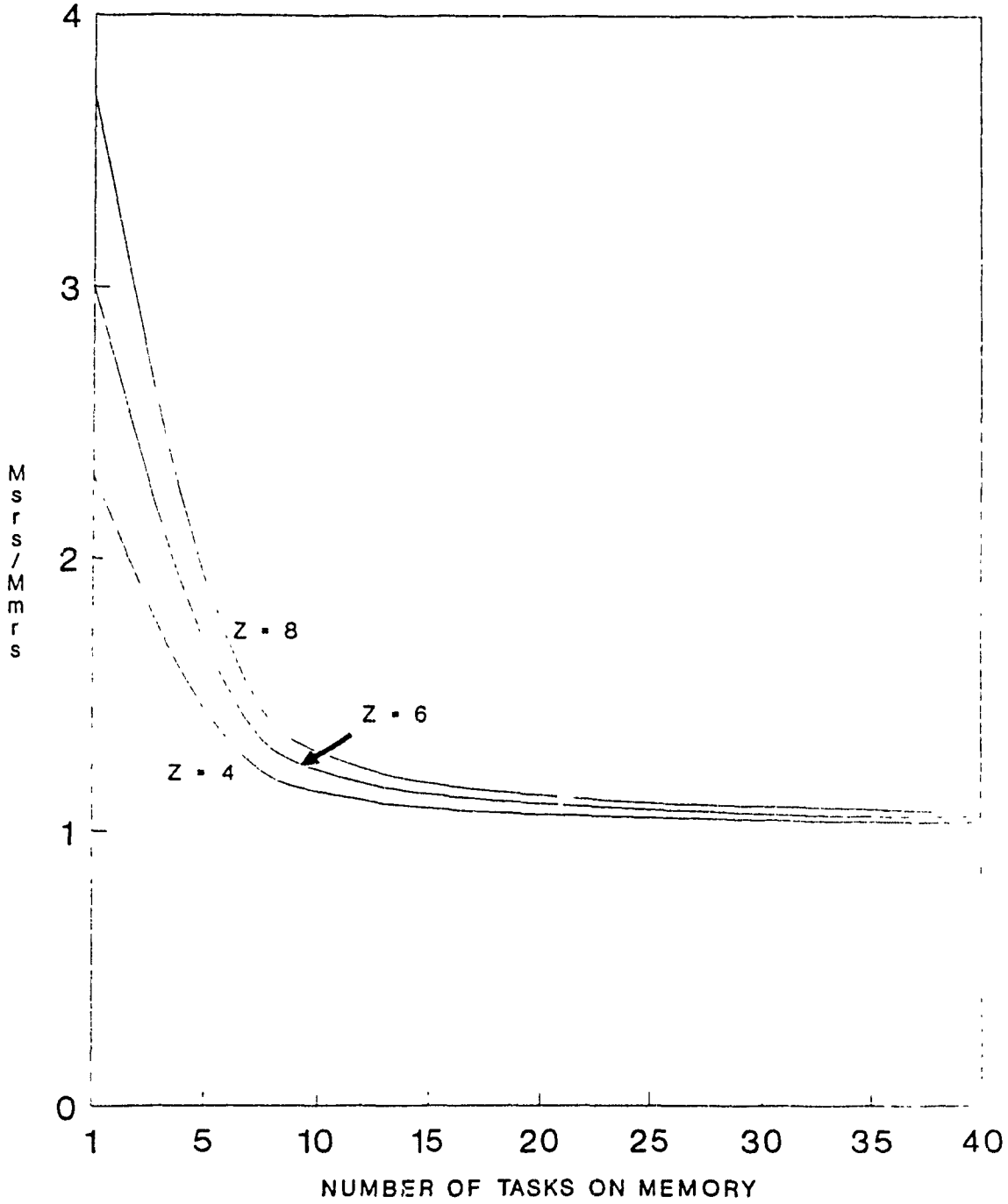


Figure 5.6c Comparing Performance of Mmrs with Msrs when P = 0.75

In integrating large number of registers on-chip sets, the number of these sets can be increased by reducing the number of registers in each set. For instance, instead of having four sets in the MRS structure with 32 registers each, it is possible to have 8 sets with 16 registers each. This will increase the number of task states which can be accommodated by MRS. Reducing the number of registers for each set, associated with the MRS structure, will lead to an increase in the processor-memory traffic generated due to the processing of the system and application tasks, and will also reduce traffic for the state-swapping operation. Conversely, the increase of the number of registers in each set will increase the traffic for the state-swapping operation. As in all currently operating RISC processors (based on the use of optimizing compilers with SRS rather than the one based on ORS), there is a general agreement to use SRS with 32 general purpose registers. Based on this fact, the reduction in the number of registers per set is not investigated in this work and it is assumed to be equal to 32.

5.5.2. Effect of the Priority Strategy

As mentioned earlier, there are two possible ways of implementing the priority strategy: Placement of tasks to MRS by starting firstly with the high priority operating systems tasks, or placement by starting with the high priority application tasks. In order to investigate when a priority strategy is effective and whether it helps to improve the performance, we have to investigate the strategy in different cases based on tasks distributed between the on-chip MRS and on the external memory.

In the strategy for placing high priority tasks of operating system, the reduced amount of the processor-memory traffic generated due to state-swapping operation, depends on the number of operating system tasks which can be resident on MRS, as well the number of application tasks which have the potential to reside on MRS. In order to estimate the performance improvement in each region, we will assume that each application task switching to any other application tasks requires, on average, the service of one operating system task. The performance improvement

in each case can be evaluated as follows:

i. In the first case, all operating system tasks' states can be resident on MRS, while all the task states of the application tasks are resident in the external memory. In such situation, the state-swapping operation which is generated due to the loading/storing the operating system tasks can be eliminated since the operating system tasks are resident on MRS. Hence, based on the assumption mentioned before, using MRS can reduce the processor-memory traffic by a factor of 2, on average, compared to the SRS structure.

ii. In the second case, some of the tasks' states of the application tasks are loaded in MRS, the rest of the application tasks are kept in the external memory, while the operating system tasks are all located in MRS. There is an upper and a lower bound for performance improvement. The upper bound occurs when the application tasks in MRS switch to other application tasks whose states are resident in MRS, i.e., the processor-memory traffic for the state-swapping operation will be completely eliminated within MRS. The lower bound occurs when the application tasks resident on the external memory is switched to another application task on the external memory too (as in part i). Therefore, the processor-memory reduction by MRS will be greater than or equal to 2 of that on SRS.

iii. In the third case, some of the operating system tasks need to be stored in the external memory, i.e., MRS is not enough to accommodate all of the task states of the operating system tasks. Similarly as in the second case, an upper and lower bound can be defined. The upper bound exists when the application tasks switching occurs with the assistance of the operating system tasks resident on MRS (as in first case). Therefore, MRS provides improvements greater than or equal to 2. The worst case condition occurs when the application tasks are switched to another with the intervention of the operating system tasks existing in the external memory, and hence, there is no performance improvement compared to SRS.

Considering the above three cases we can conclude that using priority strategy for assigning tasks' states to MRS starting firstly with operating system tasks, will help us to enhance the performance. The enhancement will occur when the number of the operating system tasks is less than or equal to the size of MRS. As the priority strategy will be ab e to assign these tasks' states to MRS, the performance improvement in comparison with SRS wili be a factor of two or more. Moreover, the priority strategy guarantees the lower bound of performance improvement by two times more than that of SRS, regardless of the number of the tasks existed in the external memory. Implementing the priority strategy with assigning firstly the operating system tasks will be efficient when it is used with operating systems which consists of a small number of tasks in their internal structure.

The same discussion is applied to the priority strategy based on placing the application tasks with high priority to MRS first. In this case, the maximum performance improvement can be achieved when all applications tasks and some of the operating system tasks resident on MRS and the switching from one application task to another one is achieved with the assistance of those operating system tasks existing on MRS. The worst case of the processor-memory traffic reduction for the state-swapping operation occurs when only some of the applications tasks are resident on MRS. Thus, this type of priority strategy is effective when the number of application tasks is small and these tasks are controlled by a general purpose real-time multi-tasking operating system. Assigning all these application tasks to MRS will guarantee the reduction of the processor-memory traffic associated with the state-swapping operation. This reduction can reach (on average) to half of the overall processor-memory traffic of the state-swapping operation.

5.6. Using MRS Structure for ISDN Processing

Generally, there are a number of real-time multi-tasking operating systems which can be used for ISDN processing. Some of these operating systems are general purpose real-time multi-

tasking operating systems, e.g., iRMX [18], and some others are special purpose real-time multi-tasking operating system, which are particularly used to support the processing of telecommunications applications [47,50,19,89]. Since all of these operating systems have a large number of tasks in their internal structure, it is not efficient to assign their tasks' states to MRS. Thus, using MRS supported with assigning high priority tasks of operating system firstly to MRS will not help to improve the performance of the ISDN processing. Clearly, this will put the performance improvement in the worst region.

The declining cost of processors during the past few years has made it possible to use multiprocessor structure within a reasonable cost. Therefore, user tasks for certain applications can be distributed to run on more than one processor in order to increase the processing throughput. This can reduce the number of application tasks which are required to be run on each processor. In ISDN node processing, for example, the user-network interface tasks can run on one processor, the Signalling System number 7 tasks on another processor, trunk controller tasks on a third, and so on [46, 17, 51].

The number of tasks within the users-network interface can be organized in few tasks as follows: call control, layer 3 of X.25, flow control of the link layer, and TEI assignment management tasks (Figure 5.7). However, this is not the only possible way of partitioning these protocols into tasks. Clearly, there are a few application tasks that can be used to run the functions of the ISDN user-network protocols on a processor. The priority of these tasks is represented by their relative importance, for instance, the call control task has the highest priority due to its critical time response behaviour.

As a few number of application tasks are required to be processed by each processor, then it is become appropriate to use MRS to hold all their tasks' states. An MRS with a size of 6 to 8 sets will be useful for ISDN processing. Many other application might also benefit from MRS of this size. Moreover, using the priority strategy which firstly assigning the application tasks to MRS is effective for ISDN processing. It guarantees the reduction of the processor-memory

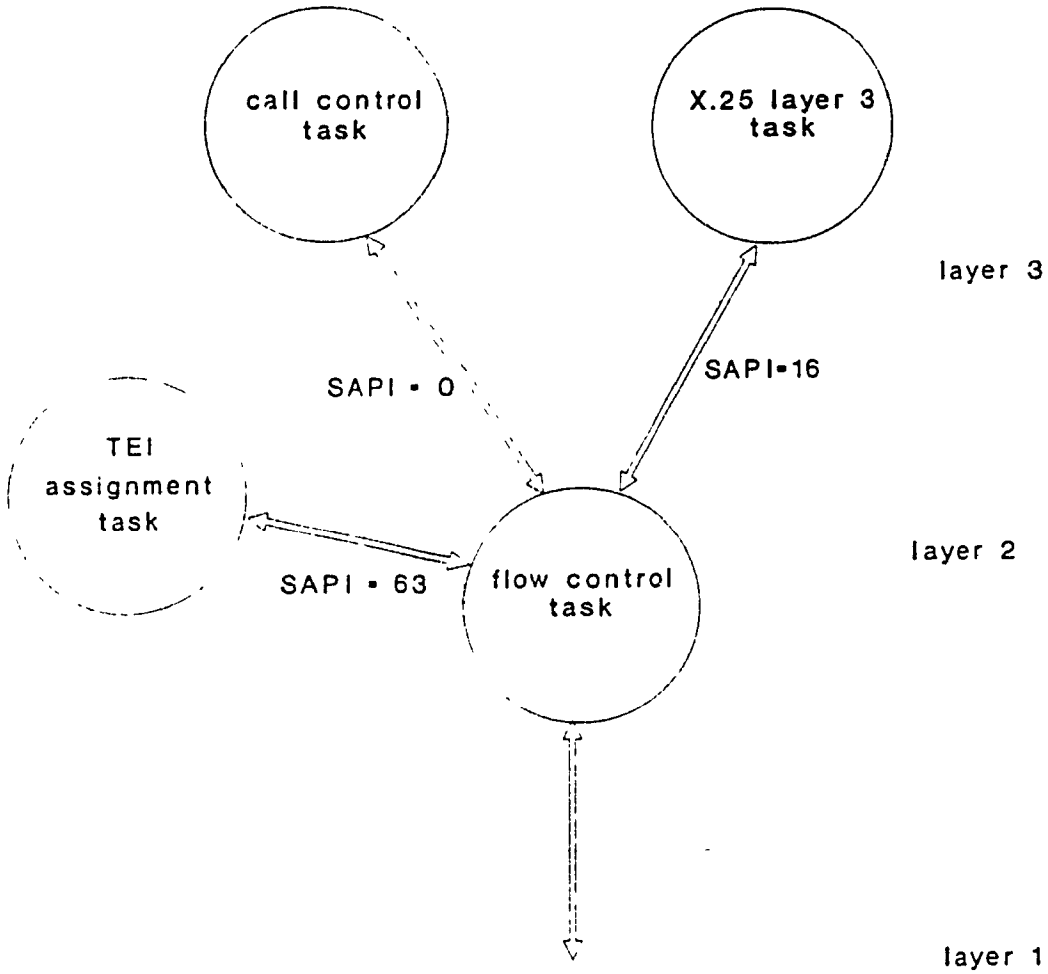


Figure 5.7 ISDN User-Network Interface Processing Tasks

traffic which is associated with state-swapping operating by half, on average. This performance enhancement will be maintained even when the operating system used for such applications with a large number of tasks.

5.7. Incorporating MRS in RISC-Based Architecture

The significance of overhead processing, associated with the state-swapping operation, and the effectiveness of MRS structure to overcome this overhead in ISDN application processing, require an efficient processor by integrating MRS within the RISC architecture. The use of the MRS structure in RISC-based architecture will need some support to allocate the appropriate register set for currently running task and to allocate the descriptor address, which exists in the external memory, for the currently running task. Thus, on-chip task register is used in the MRS-based architecture to provide such support by holding the required information which is necessary to be used by the hardware part of the processor and by the operating system.

The register set address field which exists in the task register will be used by the hardware part of the MRS-based processor to decode the register set of the currently running task. As the task switching occurs, the address of the register set belonged to the running task will be loaded to the task register by the operating system. This address will stay in the task register as long as this task is kept running. On the other hand, the task descriptor pointer field within the task register will be used by the operating system to store the task state which reside in the register set when the decision is being taken by the priority strategy to free one set inside the MRS, in order to load the task state of the next-to-run task.

The evaluation of the MIPS-based architecture in this work has shown the suitability of such architecture for data communication processing. Thus, it is required to support this architecture with the MRS organization. Furthermore, it is necessary to support MRS based architecture with MRS register file decoder and the task register. All these supports are required in addition to the basic hardware parts which are already supported by the MIPS type

architecture. Examples of such basic hardware are 32-bit bidirectional buses, eight sets of registers with 32 register inside each set, two memory interfaces: one for data and one for instruction. Figure 5.8 shows the processor core for the MRS-based data communications processor.

Using MRS organization within the MIPS-based RISC processor could raise a question about the complexity of the hardware for the registers decoder which should be used to decode the register sets and the registers within the active set. Having a complex register decoder with MRS-based architecture in comparison with the register decoder, used in SRS-based architecture, will give an impression that the MRS-based will run at lower processor clock than the SRS-based architecture. In fact, using MRS to replace SRS in MIPS-based type processor will not slow the processor clock for two reasons:

i. The decoding logic for MRS-based architecture has two part. The first part is responsible for decoding the register set. This decoding will be occurred only at the task switching time. The decoder circuit will then select the required set whose address is resident on the task register. This part of the decoding does not occur with the execution of each instruction. The decoding operation occurs only when the task register is being loaded with a new contents. The second part of the register decoder circuit is effective when each instruction is required to access certain register within the register set. This decoding process is exactly similar to what is being used in any RISC-based processor. The required register will be decoded based on the register field in any instruction. Therefore, the register decoding is exactly as in the SRS-based RISC architecture and the register decoding will not increase the instruction cycle time. Consequently, using MRS will not slow the processor clock. The simulation for the MRS organization has supported the above discussion. The decoding time is generated due to the decoding of certain register within the set and the decoding time for selecting a certain set is not generated

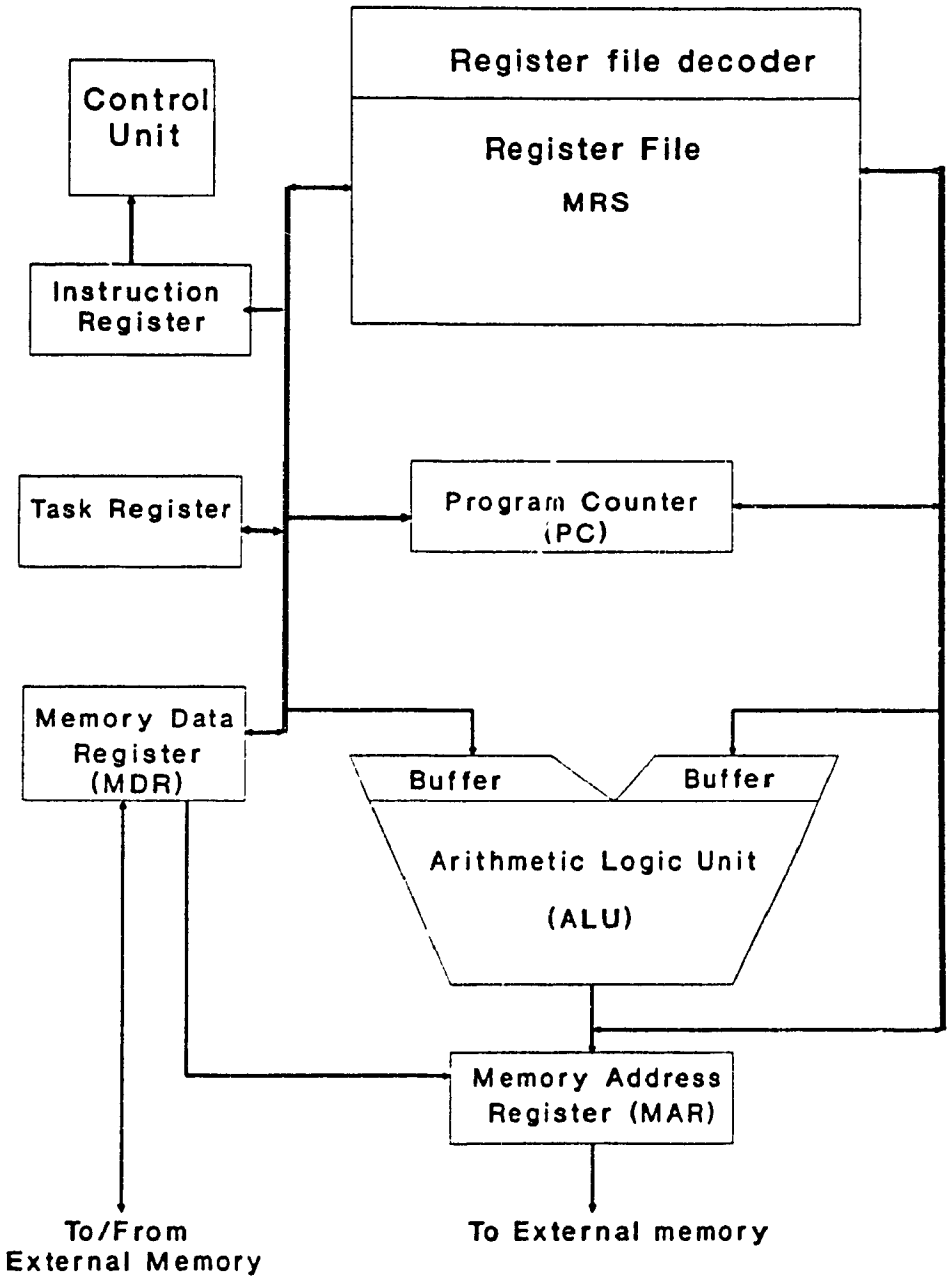


Figure 5.8 MRS-Based processor Core

any overhead since it is already performed when the task switching is occurred. This simulation is achieved by using the Verilog-HDL Programming Language (Figure 5.9 and 5.10).

ii. The decoder logic for the register set of ORS type RISC architecture is more complex than the one with the MRS-based architecture. The complexity with ORS is due to the window switching as well as the decoding of overlapped registers and registers underflow/overflow conditions. Even with this complexity of the register decoding, associated with the ORS-based architecture, the processors, based on this organization such as Sparc, are running with the clock speed similar to the processors based on SRS organization [80]. This gives a clear indication that the processor based on MRS will have similar or even higher clock speed as the SRS-based architecture.

The appropriate instructions set which can be used with this proposed MRS-based architecture is similar to these used with MIPS type of architecture and it can be divided into four groups: ALU, Load/Store, control flow, and special instructions. Since during this work the emphasis is given to the architectural level of the processor for data communications applications, particularly to the register set structure, hence we investigate here, in general, the type of these instructions with each of the above instructions group. Clearly, within the ALU group there is no floating point instructions because the ISDN processing is not numeric intensive type of applications, as we mentioned previously. The special instructions group has instructions for supporting the task register and the system call. This system call will allow us to implement a two level of operating system, i.e., user and privilege level. The Jump and Link instruction within the control flow group supports the procedure call and return where the return address is stored in one of the available registers, and hence there is no need to access the external memory to

```
1     /* This program written in Verilog HDL language to represent the functionality
2        of the Multiple Register Set (MRS) organization.  Two buffers are used to
3        store the read data from the register file (Abuff & Bbuff) */
4
5
6     module  reg_file(Abuff, Bbuff, clock1, R1add, R2add, base, wrenable, datain,
7                      regloc);
8
9             input clock1;              // register clock
10            input wrenable;            // write enable signal
11            input [7:0] R1add;         /* 8 address lines for selecting specific
12                                       register within certain register bank,
13                                       this coresponding to R1 field within
14                                       the instruction format */
15
16            input [7:0] R2add;         /* 8 address lines for selecting another
17                                       reg. within register bank, this coresponding to
18                                       R2 field in the instruction format */
19            input [7:0] base;          /* register bank select (8 bank on maximum) */
20            input [31:0] datain;       //32 lines for input data
21            output [31:0] Abuff, Bbuff;     //output data from register file
22            output [7:0] regloc;       // physical locations of registers
23            reg [7:0] regloc;
24            reg [31:0] Abuff, Bbuff;         //holds data value
25            reg [31:0] reg_content [255:0]; /* array of 256 register represent
26                                           8 memory banks, i.e 32 register/bank */
27
28
29            parameter setup =1,              // assume setup time = 1ns
30                      access_time =8;        //assume access_time = 8ns
31
32
33    // READ CYCLE
34
35            always @(posedge clock1)
36                    if (wrenable == 0)
37                            begin
38                                    regloc = base [7:0] | R1add [7:0];
39                                                  /* calculate physical address
40                                                  for first register */
41                                    #access_time Abuff = reg_content [regloc[7:0]];
42                                                  // select first bank
43                                    regloc = base [7.0] | R2add [7:0];
44                                                  /* calculate physical address
45                                                  for second register */
46                                    Bbuff = reg_content [regloc[7:0]];
47                                                  // read second register
48                            end
49
50    //WRITE CYCLE
51
52
53            always @(posedge clock1)
54                    if (wrenable == 1)
55                            begin
56                                    regloc = base [7:0] | R1add [7:0];
57                                                  /* calculate physical address*/
58                                    reg_content [regloc[7:0]] = #setup datain;
59
60                            end
61
62    endmodule
63
```

Figure 5.9
Listing of Simulated MRS Organization

```
/* This program to generate the test vectors for testing the operation of the M
RS organization */

module test_regfile;

  reg [7:0] R1add;
  reg [7:0] R2add;
  reg [7:0] base;
  reg wrenable;
  reg [31:0] datain;
  reg clock1;
  wire [31:0] Abuff;
  wire [31:0] Bbuff;
  wire [7:0] regloc;

  parameter cycle = 20;

  reg_file file1(Abuff, Bbuff, clock1, R1add, R2add, base, wrenable, datain,
                 regloc);

/* This to write data to all registers within certain bank of register
   (here selected the eighth block), after writting is over the
   stored data is read */

  initial
        begin
          base = 8'b11100000;    /* selecting bank 7 and test
                                    writing and reading in its register */
          clock1 = 0;
          R1add = 00;
          R2add =00;
          datain = 32'hffff0000;

        end
   initial
     begin

      while (R1add <  8'b00011111)
            begin
                 #20 datain = datain + 1;
                     wrenable = 1;        /* write cycle, only one write is
                                             possible in each cycle by using R1add
                                             only, i.e. no needs for R2add here*/
                     R1add = R1add +1;    /* write datain in register defined by
                                             R1add  */
             end

       R1add = 0;                 // start reading all stored data

         while (R1add < 8'b00011111)
             begin
                 #20 wrenable = 0; // initiate read cycle
                      R1add = R1add +1;
                      R2add = R1add + 1;
             end

       #100 $stop;
       end


  always #(cycle/2)clock1 = ~ clock1;    // generate clock cycles

  initial
        $gr_waves("clock1", clock1, "wrenable", wrenable, "datain", datain,
                  "R1add", R1add, "R2add", R2add, "Abuff", Abuff,
                  "Bbuff", Bbuff, "base", base, "regloc", regloc);

endmodule
```

Figure 5.10
Listing of Testing Vectors for the Simulated MRS Organization

get this address. Other instructions are conventional type which are already in use with many processors. Table 5.1 shows such a possible instruction set.

5.8. Discussions

The large overhead processing which is associated with state-swapping operations, specially in ISDN environment, has directed this work to investigate the suitability of MRS for reducing the state-swapping overhead. For the MRS structure, it is important to have some support to enhance MRS operations. This support consists of the need for decoding the register set which is used by the currently running task. There is also a need for knowing the address of the memory locations in which the task state is stored, in order to store back to the memory the content of the register set for any task state which is required to be swapped out of MRS. It is necessary to use a task register with the processor to facilitate the register set decoding and to keep the pointer to the task descriptor address in the external memory. When the number of tasks used is greater than the number of sets with MRS structure, then it is required to support the operation in such environment by having a means to know whether the register set of the tasks is on-chip or in the external memory. To achieve this we proposed an exist bit to be tested by the operating system with each task switching. This bit can be included with the descriptor of each task state.

The investigation of the priority strategy used for assigning tasks states in the dynamic system has shown that it generates a large overhead due to the behaviour of dynamic system. However, the overhead associated with the processing required by execution of the strategy itself, can be reduced. This reduction is emanated from the assigning of the states of the tasks with high priorities to MRS during the initialization time. This reduction will be larger when the strategy used for static systems. Since the static systems allow us to eliminate the priority strategy part which is involved with the task deletion. Moreover, in static systems it is possible to allocate efficiently the tasks states of the high priority tasks to MRS during the system built-up since all

Table 5.1
Data Communications Processor Instruction Set

Instruction group	Instruction	Comments
ALU	Logical AND;	rd <-- rs1 & rs2
	Logical OR;	rd <-- rs1 | rs2
	XOR;	rd <-- rs1 xor rs2
	Logical Shift Left;	rd <-- rs shifted by value in operand
	Logical Shift Right;	rd <-- rs shifted by value in operand
	Shift Left Arithmetic;	rd <-- rs shifted by value in operand
	Shift Right Arithmetic;	rd <-- rs shifted by value in operand
	Logical AND Immediate;	rd <-- rs & value in operand
	Logical OR Immediate;	rd <-- rs | value in operand
	Add;	rd <-- rs1 + rs2, generate trap if overflow exist
	Subtract;	rd <-- rs1 - rs2, generate trap if overflow exist
	Add Immediate;	rd <-- rs1 + operand, generate trap if overflow exist
	Subtract Immediate;	rd <-- rs1 - operand, generate trap if overflow exist
Load/Store	Load Byte Unsigned;	rd <-- (address)
	Load Halfword Unsigned;	rd <-- (address)
	Load Word Unsigned;	rd <-- (address)
	Store Byte Unsigned; Store	rd <-- (address)
	Halfword Unsigned;	rd <-- (address)
	Store Word Unsigned;	rd <-- (address)

Table 5.1
Data Communications Processor Instruction Set (cont'd)

Control Flow	Unconditional Branch; Branch if Equal; Branch if Greater; Branch if Smaller; Branch on Zero; Jump and Link;	PC <-- PC + OFFSET if rd = rs1 Then, PC <-- PC + OFFSET if rd > rs1 Then, PC <-- PC + OFFSET if rd < rs1 Then, PC <-- PC + OFFSET if Rd = 0 Then, PC <-- PC + OFFSET rd <-- PC + 1 Then, PC <-- operand address
Special	Load Task Register; Store Task Register; Call System;	tr <-- (address) (address) <-- tr generate Trap signal and immediate transfer control to exception handler

tasks priority are already defined during the system initialization and will not be deleted during the system operation.

The placement of tasks for MRS in both static and dynamic systems is also studied when the operating systems tasks with high priority are assigned to MRS before other tasks and when the applications tasks with high priority are assigned to MRS before other tasks. We have found in this chapter that each of these two possible placements works better in a certain environment. That is, the assigned operating systems tasks first will improve the performance when the operating system has a small number of tasks and the applications tasks are large. The assign of the applications tasks first to the MRS will work better when the number of applications tasks is small and is controlled by a general purpose real-time multi-tasking operating system.

The performance enhancement which can be achieved by using MRS has been studied with and without using the priority strategy. When the priority strategy is not used, MRS can improve the performance by a certain factor. This factor depends on the size of MRS, the number of tasks which have no place to reside on MRS, and the probability of how large are the tasks whose states resident on MRS running more frequently than the others. The performance enhancement when the number of tasks resident in the external memory is large (about over 15 tasks) will be very small and the lower bound can reach to the situation when MRS and SRS provide the same performance. Reaching the region where there is no improvement in the performance will be fast when the probability of tasks whose states existed on the external memory are running more than the others and the size of MRS is small. The use of the priority strategy to support the operation of MRS will improve the lower bound of the performance enhancement. The lower bound will be independent of the number of tasks used which are resident on the external memory. The lower bound will guarantee that MRS doubles the performance compared to SRS.

For ISDN applications and other applications, specially the applications with a small number of static tasks which are controlled by a general purpose real-time multi-tasking operating

system, the high priority of the applications tasks placement will be more appropriate than the one starts with placement of the operating system tasks. The lower bound of the performance improvement with MRS in such environment will be about 2 times of the one with SRS. This performance improvement is obtained by reducing the processing required for the state-swapping operation. The integration of MRS with the RISC processor is quite simple and similar to the processors using SRS except the need for extra logic in the register set decoding circuit.

CHAPTER 6

CONCLUSION AND FURTHER WORK

In this thesis, we have presented an evaluation study for using RISC-based architecture in ISDN processing, in particular, and data communications nodes processing, in general. This work also investigates the amount of overhead processing involved with the state-swapping operations associated with the task switching and the possible enhancement of the RISC architecture to overcome this overhead by using the MRS structure.

The software simulation of some ISDN functions from layer 2 and 3 of the ISDN user-network interface protocols has allowed us to perform an analysis to investigate the characteristics of the software structure for these protocols. Using a dynamic scanning, we measured an important factors to evaluate the software structure such as the frequency of HLL statements, the nested procedures depth, and the size of the parameter which passed between procedures. The results show that only short depth of the nested procedure always occurs. Also, the relative dynamic frequency of procedures call/return operations constitutes less than 10% of the overall memory references made by all HLL statements used in the software of these protocols. We concluded from these results that the short depth of nested procedure (maximum 3) will leave over 50% of the ORS structure useless if it is implemented within the RISC architecture for ISDN processing. In addition, the memory-references made by the call/return statements was found low when these statements are handled by a RISC processor based on optimizing compiler to reduce processor memory traffic rather than using a large register set as in ORS. These results look favourable to adopting RISC architectures, which are based on using optimizing compilers rather than an ORS organization.

Since a register set within the RISC architecture based on optimizing compiler is usually

much larger than that of the CISC architecture, the impact of the RISC register set on the state-swapping processing overhead is an important issue. The intensive task switching operations in real-time multi-tasking applications, such as in ISDN, has also increased the state-swapping processing overhead. The results of the evaluation of the cost of the state-swapping overhead processing, show that it is significant where the average state-swapping overhead processing is about 17% and 13% of the totally executed machine instructions for the Q.931 and TEI assignment tasks, respectively. These state-swapping overhead processing were measured with the using of an optimizing compiler supported with delay load technique and were tested under load conditions which could reach to 50 messages processed with each task switching. We concluded that the use of RISC architecture with MRS would be useful for ISDN applications. MRS will enhance processor performance by reducing the overhead generated by the state-swapping operation.

The analysis (Chapter 5) shows that the performance improvement of the RISC architecture using MRS is affected by the size of MRS, the number of tasks used in the real-time multi-tasking application, and by the probability of the running tasks which are resident on the external memory. It is indicated that an organization with 8 register sets is adequate to enhance the performance when a total of 18 tasks are used (10 of them states are resident in external memory). However, as the number of tasks resident on the external memory is increased, then the performance improvement will be dropped. In this case, the priority strategy is introduced to enhance the performance by increasing the lower bound of the performance improvement and making this bound independent of the number of tasks existing in the external memory. The lower bound of the performance enhancement is about 2 times better than the SRS. This is based on the assumption that the task switching from one application task to another requires the service of one operating system task, on average. We have also proved that MRS is more

effective in applications with static tasks where the applications tasks with high priority are assigned to MRS first, such as in the ISDN environment.

The contributions of this work can be summarized as follows:

i. The study of the processing nature within the communications network nodes and specifically for ISDN protocols. This has not been studied in the literature.

ii. The study of software characteristics of the software running in ISDN applications has clarified the nature of the register set organization, which is required in RISC-based architecture. This can be considered as the first trial of this type of study for ISDN processing.

iii. The estimation of the state-swapping overhead has added another important finding to this work, where all previous works overlooked the size of this overhead. This study can be considered as the first one of its kind to contribute such measurements to find the size of this overhead in terms of the applications software processing.

iv. The performance of MRS-based RISC architecture has been evaluated and compared with the SRS-based RISC architecture. The approach we used in this evaluation has not been studied in literatures.

v. Finally, supporting MRS-based RISC architecture with the Priority-based approach to allocated tasks states to MRS has not investigated in previous studies. This has added more to our contributions in the area of RISC architecture.

There is no doubt that there are some points in this work that have been left to future studies. These points can be summarized as follows:

i. The MRS-based processor is simple and attractive for ISDN processing. All attention given in this work to the architecture aspects of this processor. The VLSI implementation

is important and has not performed in this work. Such implementation could give answers to many important issues like the physical area requirement of MRS, the processor speed, design complexity, critical paths, etc.

ii. The MRS-based architecture is evaluated with the use of two priority strategies. This evaluation is performed theoretically and based on some assumptions. We believed that it will be useful if the processor based on MRS structure is tested under the real ISDN environment in order to see the actual performance improvement achieved by using the MRS supported with the priority strategies.

iii. The development of the real-time multi-tasking operating system which integrate the priority strategy. This will help to use this strategy practically with ISDN processing and other applications.

REFERENCES

[1] CCITT Recommendations I.431, " Basic User-Network Interface- Layer 1 Specifications ", Geneva, Switzerland, 1984.

[2] CCITT Recommendations Q.920, " ISDN User-Network Interface Data Link Layer, general aspects ", Geneva, Switzerland, 1984.

[3] CCITT Recommendations Q.921, " ISDN User-Network Interface Data Link Specifications ", Geneva, Switzerland, 1984.

[4] CCITT Recommendations Q.931, " ISDN User-Network Interface Layer 3 Specifications ", Geneva, Switzerland, 1984.

[5] Ali Elkateeb and Tho Le-Ngoc, "ISDN-CCS7 Structure and Functions for Point-to-Multipoint Subscriber Radio Systems", *Proceedings of the International Conference on Communications (ICC)* , June 1988, pp 1663-1667.

[6] T. Le-Ngoc, M. Stashin and A. Elkateeb, "ISDN Implementation for a Point-to-Multipoint Subscriber Radio System", *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, November 1988, pp 516-519.

[7] T. Le-Ngoc, M. Stashin and A. Elkateeb, "ISDN Services and Support on a Point-to-Multipoint Radio System", *Proceedings of the Singapore ICCS '88* , November 1988, pp 1.2.1-1.2.5.

[8] T. Le-Ngoc, M. Stashin and A. Elkateeb, " ISDN Signalling and Packet Data Support on Point-to-Multipoint Subscriber Radio Systems", *Proceedings of the ICC '89*, June 1989, pp 1314-1318.

[9] T. Le-Ngoc, M. Stashin and A. Elkateeb, "ISDN Services Over Microwave Systems", *Communications Engineering International (CEI)*, vol 17 no4, June 1989, pp 62-68.

[10] T. Le-Ngoc, M. Stashin and A. Elkateeb, " ISDN Implementations for a Point-to-Multipoint Subscriber Radio System", *Computer Communication*, vol 13 no 3, April 1990, pp 131-135.

[11] A.Elkateeb and Tho Le-Ngoc, "ISDN Software Characteristics for Designing a Specialized RISC Processor", *Proceedings of the Canadian Conf. on Electrical and Computer Eng.*,Canadian Conf. on Electrical and Computer Eng., September 1990, pp 78.4.1 - 78.4.4.

[12] A. Elkateeb and Tho Le-Ngoc, "Processing Overhead of the RISC State-Swapping Operation in the ISDN Software Processing", *IEEE Pacific RIM Conf. On Communications, Computers and Signal Frocessing*, May 1991, pp 433-436.

[13] J. Paterson and A. Silberschatz, *Operating System Concepts*, Addison-Wesly, Inc., 1983.

[14] A. Elkateeb and Tho Le-Ngoc, "Multi-Processor Structure for High Speed Communication applications", *Proc. of the Montech' 87 Conf.*, September 1987, pp 150-152.

[15] A. Elkateeb and T. Le-Ngoc, " A Priority Strategy on RISC for Real-Time Multi-Tasking Software Applications", *Computer Architecture News*, vol 17, no 4, June 1989, pp 62-68.

[16] A. Elkateeb and T. Le-Ngoc, "Evaluation to RISC State-Swapping Processing Overhead in an ISDN Environment", *Computer Communications*, vol 15 no 1, January/February 1992, pp 52-57.

[17] P.Bocker, *ISDN the Integrated Service Digital Network: Concept, Methods, Systems*, Springer-Verlag, 1988.

[18] *Introduction to the iRMX 286 operating system*, Intel corporation, Calif., 1987.

[19] R. Boyed and et.al., "A Distributed Operating System for Reliable Telecommunications Control", *Proc. of 5th Int. Conf. on Software Eng. for Telecomm. Switching System*, Sweden, July 1983, pp 139-141.

[20] P. Jenkins and K. Knighton, " *Open System Interconnection - An Introductory Guide* ", *British Telecommunications Engineering*, Vol. 3, July 1984.

[21] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open System Interconnection ", *IEEE Trans. Commun.*, Vol. COM-28, April 1980, pp 425-432.

[22] A. Tanenbaum, *Computer Networks*, Prentice-Hall, 1981.

[23] *Basic Reference Model for Open System Interconnection*, International Standards Organization, ISO 7498, 1983.

[24] " Special Issue on OPEN System Interconnection ", *Proc. IEEE*, Vol. 71, No. 12, December 1983.

[25] R. Cypers, *Communications Architecture for Distributed Systems*, Addison-Wesley, 1978.

[26] S. Wecker, " DNA: the Digital Network Architecture ", *IEEE Trans. Commun.*, 1980, pp 510-526.

[27] M. Sloman and J. Kramer, *Distributed Systems and Computer Networks*, Prentice-Hall, 1987.

[28] N. Al-Ghitany, *Performance Evaluation of RISC-Based Architecture for Image Processing*, Ph.D thesis, The Ohio State University, 1988.

[29] Personel communications with ITT, ITT seminar on ISDN protocols implementations, Montreal, May 1987.

[30] G. Schlanger, " An Overview of Signalling System No. 7 ", *IEEE J-SAC*, Vol. SAC-4, No. 3, May 1986.

[31] J. Luetchford, " CCITT Recommendations - Network Aspects of the ISDN ", *Proc. of ICC*, June 1985, pp 1067-1071.

[32] R. Roca, " ISDN Architecture", *AT&T Technical Journal*, Vol 65 Issue 1, Jan./Feb. 1986, pp 5-17.

[33] S. Kano, " Layer 2 and Layer 3 ISDN Recommendations ", *IEEE J-SAC*, Vol. SAC-4, No. 3, May 1986.

[34] W. Gifford, " ISDN User-Network Interfaces ", *IEEE J-SAC*, Vol. SAC-4 No. 3, may 1986, pp 343-348.

[35] J. Peterson and A. Silberschatz, *Operating System Concepts*, Addison-Wesley, 1983.

[36] *IRMX 286 Nucleus User's Guide*, intel publication, 1987.

[37] A.Krishnakumar and K. Sabnani, " VLSI Implementations of Communication Protocols- A Survey", *IEEE J-SAC*, vol 7 no 7, September 1989, pp 1082-1090.

[38] *MC 68302 Technical Data book*, Motorola publications, 1989.

[39] J. Erickson, " Protocol Controller Chip Manages X.25 Interface", *Computer Design*, vol 24, September 1985, pp 78-81.

[40] R. Dirvin and A. Miller, " The MC68824 Token Bus Controller - VLSI for the Factory LAN", *IEEE Micro*, June 1986, pp 15-25.

[41] M. Stark and et. al., " A High Functionality VLSI LAN Controller for CSMA/CD Network", *Proc. IEEE Compcon*, February 1983, pp 510-517.

[42] H. Ichikawa and et. al., "High-Speed Packet Switching Systems for Multimedia Communications", *IEEE J-SAC*, October 1987, pp 1336-1345.

[43] R. Kun, " An ISDN Network D-Channel VLSI Architecture", *IEEE J- SAC*, vol 4 no 8, November 1986, pp 1275-1280.

[44] G. Geiger and L. Lerach, " ISDN - Oriented Modular VLSI Chip Set for Central Office and PABX Applications", *IEEE J-SAC*, vo 4 no 8, November 1986, pp 1268-1274.

[45] R. Windhaw and et. al., "Basic Access Network Termination Chip for ISDN 4-Wire Loops", *AT&T Technical journal*, vol 66 issue 6, Nov./Dec. 1987, pp 35-50.

[46] H. Ogata and et. al., "Software Architecture for Multi-Processor Digital Switch System",

Proc. of software engineering for telecommunication switching systems conf., July 1983, pp 139-141.

[47] B. Laws and P. Rubin, "MCPOS - A Realtime Telephone Operating System", *Proc. of IEEE 1985 Phoenix conf. on comp. and comm.*, March 1985, pp 185-189.

[48] G. Myers and et. al., "Microprocessor Technology Trends", *Proceedings of the IEEE*, vol 74 no 12, December 1986, pp 1605-1622.

[49] A. Niwa and T. Yamada, " A 32-bit Custom VLSI Processor for Communications Network Nodes", *IEEE J-SAC*, vol 4 no 1, January 1986, pp 192-199.

[50] H. Watanabe and et. al., "Operating System for VLSI Multiprocessor Systems", *Proc. of fifth int. conf. on software eng. for telecommunications switching system* , July 1983, pp 139-141.

[51] T. Yajima and K. Suda, " Switching System Software Design for ISDN", *IEEE J-SAC*, vol 4 no 8, November 1986, pp 1222-1229.

[52] M. Marsan and et. al., "Comparative Performance Analysis of Single Bus Multiprocessor Architecture", *IEEE Trans. on computers*, vol 31 no 12, December 1982, pp 1179-1191.

[53] *OEM Technical Manual*, Intel publications, 1990.

[54] H. Lycklama and L. Bayer, "The MERT Operating System", *Bell System Technical Journal*, vol 57 no 6, part 2, July-August 1978, pp 2049-2086.

[55] H. Watanable and et. al., "Operating System for VLSI Multiprocessor Systems", *Proc. of fifth int. conf. on software eng. for telecommunications switching system* , July 1983, pp 167-172.

[56] K. Sakamura, " Architecture of the TRON VLSI CPU ", *IEEE Micro*, April 1987, pp 17-31.

[57] T. Ohkubo and et. al., " Configuration of the CTRON ", *IEEE Micro*, April 1987, pp 33-44.

[58] H. Monden, " Introduction to ITRON the Industry-Oriented Operating System ", *IEEE*

Micro, April 1987, pp 45-52.

[59] K. Sakamura, " BTRON the Business Oriented OS ", *IEEE Micro*, April 1987. pp 53-65.

[60] D. Patterson, "Reduced Instruction Set Computer", *Comm. of ACM*, January 1985, pp 8-21.

[61] G. Radin, "The 801 Minicomputer", *Proc. SIGARCH/SIGPLAN symp. architecture support for programming languages and operating systems*, March 1982, pp 39-47.

[62] M. Katevenis, *Reduced Instruction Set Computer Architecture for VLSI*, Ph.D dissertation, Univ. California, Berkeley, October 1983.

[63] A. Tanenbaum, "Imlications of Structured Programming for Machine Architecture", *Comm. of The ACM*, Vol. 21 No.3, March 1978, pp 237-246.

[64] J. Hennessy and et. al., "Design of a High Performance VLSI Processor", *Proc. 3rd Caltech conf. VLSI*, California Inst. Tech., March 1983, pp 33-54.

[65] *M88000 Architecture Specification* , Motorola corp., Schaumburg, 1986.

[66] *i860 programming refernce manual* , Intel corp., Santa Clara, Calif. 1988.

[67] G. Kane, *Mips RISC Architecture* , Prentic-Hall Inc. 1988.

[68] G. Myers and et. al., "Microprocessor Technology Trends", *Proc. of the IEEE*, vol. 74, no. 12, December 1986, pp 1605-1622.

[69] D. Patterson and C. Sequin, " A VLSI RISC", *IEEE computer*, September 1982, pp 8-21.

[70] F. Fox and et. al., " Reduced Instruction Set Architecture for a GaAs Microprocessor System ", *IEEE computer*, October 1986, pp 71-81.

[71] V. Miltinovic and et. al., "Issues of Importance in Designing GaAs Microcomputer Systems", *IEEE computer*, October 1986, pp 45-57.

[72] M. Katevenis and et. al., "RISC: Effective Architectures for VLSI Computers", *VLSI electronics microsystems science*, edited by N. Einspruch, Academic press, NY. 1986.

[73] S. Allowrth and R. Zobel, *Introduction to Real-Time Software Design*, Macmillan education ltd., 1987.

[74] *CLIPPER 32-bit Microprocessor*, User's manual, A Schlumberger Company, New Jersey, 1987.

[75] W. Stalling, *Computer Organization and Architecture*, Macmillan Publishing, 1986.

[76] R. Kelley and R. Clark, "Applying RISC Theory to a Large Computers", *Computer design*, November 1983.

[77] Y. Tamir and C. Sequin, "Strategy for Managing the Register File in RISC", *IEEE Trans. on Computers*, November 1983.

[78] A. Tanenbaum, "Implecation of Structured Programming for Machine Architecture", *Comm. of ACM*, March 1978.

[79] D. Wall, "Register Windows vs. Register Allocatir *Proc. ACM SIGPlan 88 Symp. Programming Design and Implementation*, June 1988, pp 67-78.

[80] R. Piepho and W. Wu, "A Comparison of RISC Architecture", *IEEE Micro*, August 1989, pp 51-62.

[81] S.Kano and et. al., "ISDN User/Network Protocol-Overall Architecture", *Globecomm '82*, November 1982.

[82] CCITT Recommendations Q.930, "ISDN User/Network interface layer 3 - general aspects", Geneva, Switzerland, 1984.

[83] W.Stalling, *Reduced Instruction Set Computer*, IEEE press, 1986.

[84] R. Eickemeyer and J. Patel, "Performance Evaluation of Multiple Register Sets", *Proc. of 14th annual int. Symp. on Computer Architecture*, June 1987, pp 264-271.

[85] C. Hitchcock III and H. Sprunt, "Analyzing Multiple Register Sets", *Proc. of 12th annual int. symp. on Computer Architecture*, June 1985, pp 55-63.

[86] M. Huguet and T. Lang, " A Reduced Register File for RISC Architecture", *Computer Architecture News*, vol 13, no 4, Sepember 1985, pp 22-31.

[87] J. Hennessy, " VLSI processor architecture", *IEEE Tran. on Compu.*, December 1984, pp 1221-1246.

[88] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw Hill Inc., 1984.

[89] J. Swerup, "Software Architecture in a Digital Voice/Data PABX with Distributed Control", *Proc. on software Eng. for Telecomm. switching system*, July 1983, pp 184-189.

[90] Sparc Architecture Maunal, Sun microsystems, Inc., Mountain View, Calif., 1987.

[91] D. Knuth, *An Empirical Study of Fortran Programs*, Technical Report STAN-CS-70-186, Computer Science Dept., Stanford University, Stanford, CA, 1970.

[92] P. Jenkins and K. Knighton, "Open System Interconnection - An Introductory Guide", *British Telecommunications Engineering"*, Vol 3, July 1984.

[93] E. Sternheim and et. al., *Digital Design with Verilog HDL*, Automata Publishing Company, California 1990.

Appendix A

Listing of Q.931 Basic Call Control Program

```
/*****************************************************************************
This program shows the list for the Q.931 Basic Call Control functions. Functions processed in
this program are as follows:
1. Check the validity of a message.
2. check for service availability.
3. B-Channel negotiations.
4. Issuing a status messages.
5. Updating databases.

The establish and terminate phases of the Q.931 Basic Call Control is processed in this program
by several procedures. Each procedure activates by each subfield within the Q.931 messages.
For instance, the check_refno procedure is invoked when the reference number field is detected
in any Q.931 message.

The database structure includes several records which necessary to this program processing.
These records which necessary to this program processing. These records consist the call
reference, calling terminal number, called terminal number, service available, and call status.

In this listing, some Q.931 Basic Call Control messages are shown as an example to illustrate
the program behaviour and response to these messages. Some display messages are printed
to trace and debug the program.

This program is used to process different type of Q.931 Basic Call Control messages. These
messages are dynamically scanned during the processing of this program and as described in
Chapter 3. This program is compiled and tested under the R2000 RISC-based machine.
*****************************************************************************/

    struct  table {
                                    /* table for working database, this for "sub"
                                    user-network interface */
        unsigned short callref[128];
                                    /* call reference */

        unsigned int   originating [128];
                                    /* subscribers numbers */

        unsigned int   destination [128];
                                    /* subscribers number */

        unsigned short status_orig[128];
                                    /* status of originating
                                    user */

          unsigned short status_dest[128];
                                    /* status of destination
                                    user */

          unsigned short service_req[128];
```

```
                                        /* services requested */
        };


main()

{

        short   type;           /* message type */
        short   length;         /* length of message */
        short   *msgp;          /* pointer to the ISDN messages buffer */
        short   *msgp1;         /* pointer to calling messages buffer */
        short   *msgp2;         /* pointer to called messages buffer */
        short   count;          /* counter for how many messages we
                                need to process from each buffer */
        short   count1;
        short   count2;
        unsigned short  refno;          /* reference number */
        unsigned short  skip;           /* flag to skip messages in buffer */

        int     *malloc();

        struct  table   *ptable;
                                        /* the following ISDN messages are just an example of the
                                        messages     used in this work */

        static short calling[61] ={
                        17,     /* message length */
                        200,    /* call reference */
                        0x05,   /* message type (set-up) */
                        0x04,   /* bearer capability */
                        0x00,   /* length = 0 */
                        0x18,   /* channel identification */
                        0x01,   /* length */
                        0x01,   /* B1 channel requested */
                        0x6c,   /* originating information field */
                        0x03,   /* length */
                        0xc0,   /* local calls */
                        0x00,   /* first two digit, only four used */
                        0x01,   /* second two digits */
                        0x70,   /* destination information field */
                        0x03,   /* length */
                        0xc0,   /* local calls */
                        0x00,   /* first two digits, only 4 used */
                        0x04,   /* second two digits */

                        17,     /* message length */
```

```
200,   /* call reference */
0x05,  /* message type (set-up) */
0x04,  /* bearer capability */
0x00,  /* length = 0 */
0x18,  /* channel identification */
0x01,  /* length */
0x01,  /* B1 channel requested */
0x6c,  /* originating information field */
0x03,  /* length */
0xc0,  /* local calls */
0x00,  /* first two digit, only four used */
0x02,  /* second two digits */
0x70,  /* destination information field */
0x03,  /* length */
0xc0,  /* local calls */
0x00,  /* first two digits, only 4 used */
0x03,  /* second two digits */

17,    /* message length */
200,   /* call reference */
0x05,  /* message type (set-up) */
0x04,  /* bearer capability */
0x00,  /* length = 0 */
0x18,  /* channel identification */
0x01,  /* length */
0x01,  /* B1 channel requested */
0x6c,  /* originating information field */
0x03,  /* length */
0xc0,  /* local calls */
0x00,  /* first two digit, only four used */
0x05,  /* second two digits */
0x70,  /* destination information field */
0x03,  /* length */
0xc0,  /* local calls */
0x00,  /* first two digits, only 4 used */
0x06,  /* second two digits */

6,     /* length of release message */
200,   /* reference number */
0x4D,  /* message type = release */
0x08,  /* inf. element identifier= cause */
0x02,  /* inf element length */
0x00,  /* coding standard(CCITT=00)
          location (user =000)*/
0x90,  /* class cause = normal event*/
};     /* end of calling messages buffer */

static short called[20]= {
```

```
                2,      /* alerting message length */
                200,    /* reference number */
                0x01,   /* alerting message type */

                2,      /* connect message length */
                200,    /* reference number */
                0x07,   /* message type (connect) */

                6,      /* disconnect message length */
                200,    /* reference number */
                0x45,   /* message type = disconnect */
                0x08,   /* inf. element identifier= cause */
                0x02,   /* inf element length */
                0x00,   /* coding standard(CCITT=00)
                        location (user =000)*/
                0x90,   /* class cause = normal event*/

                6,      /* length of release_comp message */
                200,    /* reference number */
                0x4A,   /* message type = release complete */
                0x08,   /* inf. element identifier= cause */
                0x02,   /* inf element length */
                0x00,   /* coding standard(CCITT=00)
                        location (user =000)*/
                0x90,   /* class cause = normal event*/
        }; /* end of called messages buffer */

static unsigned int     perm_db[3][11] = { /* permanent database */
                {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                {0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
                };      /* three array :first for user number,
                                second for service availability,
                                both B-channels are available for
                                all subscribers, third for
                                charging  */

msgp1 = &calling[0];            /* pointer to calling messages buffer*/
msgp2 = &called[0];             /* pointer to called messages buffer */

ptable = (struct table *) malloc ((unsigned)
                        sizeof (struct table));

count1 = 1;             /* count1, count2, & count are variables
                        can be changed to any number of messages
                        required to process from each buffer */
count2 = 1;
msgp = msgp2;           /* useful in initialization only */
```

```
                /* this statements to switch between both
                buffers (called and calling) and process
                one message from each( this can be changed
                to any number by changing count1, count2 values */

again:  if (*msgp1 != 0 && count1 != 0) {       /* i.e messages exist in
                                        calling buffer */

              msgp2 = msgp ;  /* update msgp, needed for initialization */

               printf (" content of msgp2 =");

               printf (" %d\n", *msgp2);

               msgp = msgp1;

               printf (" content of msgp1 = ");

               print (" %d\n", *msgp1);

               count = count1; /* to set the number of messages to process */

               count1 -= count;        /* to ensure other buffer will run
                                  next*/

               count2 = 1;     /* not useful for initialization but necessary
                           later on */

               length = *msgp++;

               refno = *msgp++;

               length -= 1;

               type = *msgp++;

               length -=1;

               if (type == 5 && (skip = check_refno(ptable, refno)) == 1){
                                  /* check if refno is already exist with
                                  the setup messages only */

                     printf (" setup message will be skipped: refno in use \n");

                     msgp = msgp + length;
                                                                  /* then skip this message
                                        and go back to the buffer*/
```

```
            msgp1 = msgp;
                                        /* useful if more than one
                        message need to process each
                        time */

            goto again;
          };     /* if */

        goto end;
      }     /* if */

else if (*msgp2 != 0 && count2 != 0) {
                                        /*      i.e messages exist in
                                             called buffer */

      msgp1 = msgp ;  /* update msgp1, needed for intilization */

      msgp = msgp2;

      count = count2; /* to set the number of messages to process */

      count2 -= count;        /* to ensure other buffer will run
                        next*/

      count1 = 1;     /* not useful for intilization but necessary later on */

      length = *msgp++;

      refno = *msgp++;

      length -= 1;

      type = *msgp++;

      length -=1;

      if (type == 5 && (skip = check_refno(ptable, refno)) == 1){
                        /*      check if refno is already exist with the setup messages only
                              */

            printf(" skip setup message since refno in use \n");

            msgp = msgp + length;
                              /*    then skip this message and go back to the buffer*/

            msgp2 = msgp;
                              /*    useful if more than one message need to process
                                    each time */
```

```
                goto again;

                };      /* if */
        };      /*  else if */

end:    while (count != 0) {

        count -= 1;     /* this can be adjusted to any value equal to the messages required
                           to be processed */

        switch (type) {
                                /* switch according to message type */

        case 0x05:

                setup (&msgp, length, ptable, refno, perm_db);
                                /* call  setup procedure */

                break;

        case 0x01:              /* alerting message */

                alerting (&msgp, length, refno, ptable);
                                /* call alerting procedure */

                break;

        case 0x07:              /* connect message */

                connect (length, ptable, refno,perm_db);
                                /*      call connect message procedure */

                break;

        case 0x45:              /* disconnect message */

                disconnect(&msgp, length, ptable, refno, perm_db);

                break;

        case 0x4D:              /* release message */

                release(&msgp, length, ptable, refno);

                break;

        case 0x4A:              /* release complete message */
```

```
                release_comp(&msgp, length, ptable, refno);
                break;
            default:
                printf ("this message is not used in ISDN \n");
                break;
            }       /* switch */
        }           /* while */
        printf(" next ISDN message will be processed \n");
        goto again;     /* start again */
}                   /* main */


/*****************************************************************************
            check reference number procedure
function: to check if the reference number assigned by the calling terminal
        is already in use.
input parameters: ptable,
                refno.
output parameters: value which either 1 when the refno is already in user or
                0 if refno is not in use.
*****************************************************************************/

check_refno(ptable, refno)

struct table   *ptable;

unsigned int    refno;

{

unsigned int    reason;

unsigned short  skip;


        printf (" check refno procedure in progress \n ");

        if ((ptable -> callref[refno]) == 0){
```

```
                    /*    0 is indication of free refno, in CCITT 0 is indication of
                          stimulus terminal (here we assume all terminals are
                          functional type ) */

        printf(" refno not in use \n");

        ptable -> callref[refno] = refno;
                          /*    set reference number on working database */

        skip = 0;

        return (skip);

        }       /* if */

    else {              /*    if refno already in use, then send to calling terminal status message
                          */


        printf (" refno already in use \n ");

        reason = 0x05;  /*          class: invalid message
                          value: invalid reference number */

        status(reason, refno);

        skip = 1;

        return(skip);

      };        /* else */

}
```

```
/*****************************************************************************
            check service procedure
function: to check if the requested services is available to both originating
        and destination terminals. The terminal status is organized as
        follows:
                7               3    2       1      0
                ------------------------------------------
                |                | service  | lines status |
                ------------------------------------------
                service: 0= only B1 channel allowed.
                        1= both B-channels allowed.
                lines status: 00=both free.
                              01=B1 busy.
                              10=B2 busy.
                              11=B1 and B2 busy.
input parameters: ptable,
                refno,
                direction.
output parameters: none.
*****************************************************************************/

check_service(ptable, refno, direction, perm_db)

struct  table   *ptable;

unsigned short  refno;

unsigned short  direction;

unsigned int    perm_db[3][11];


{

        unsigned int    temp_status;    /* temporarily area for user status */

        unsigned int    temp_req;       /*      temporarily area for requested services */

        unsigned int    state;

        unsigned short  reason;

        unsigned int    index;

        if (direction ==0){     /* check service of the originating */

                temp_req = ptable -> service_req[refno];
                                        /*      get service requested from working database,
```

```
                                service request is set by ch.Id */

  temp_req = temp_req & 0x03;     /* mask B-channel field */

  temp_status = ptable -> status_orig[refno];
                          /*      get user status from working db */

switch (temp_req){

   case 0:      /*  no channel requested, this will processed in future */

       break;

   case 1:      /* B1 channel requested */

       if (temp_status == 0 || temp_status == 4)
            /* B1 free and this service available */

            state = (temp_status | 1); /* set B1 busy */

       else if (temp_status == 1 || temp_status == 5) {
            /*      B1 already in use, i.e network assign this while the user send this
                    message */

            reason = 0x11;
                                /* class: normal event.
                                   value: user busy. */

            status (reason, refno);

            printf(" requested B1 channel is busy\n");

            printf (" no update to all database \n");

            goto last;
                          /*  go back with out saving the lines status in both
                              database */

            };     /* else */

       break;

   case 2:            /* B2 channel requested */

       if ( temp_status == 4 || temp_status == 5)
                  /* channels free or B1 busy */

            state = (temp_status | 2);
```

```
                                        /* set B2 busy in working db */

        else if ( temp_status == 0 || temp_status == 1){
                /* service is not allowed or B2 busy */

                reason = 0x15;
                                /* class: normal event.
                                   cause: call rejected */

                 status ( reason, refno);

                printf(" B2 service is not allowed\n");

                printf (" no update to all database \n");

                goto last;
                                /*  go back with out saving the lines status in both
                                    database */

                }

        else if ( temp_status == 6) {   /* B2 busy */

                reason = 0x11;
                                /* class: normal event.
                                   value: user busy */

                status( reason, refno);

                printf(" requested B2 channel is busy\n");

                printf (" no update to all database \n");

                goto last;
                                /*  go back with out saving the lines status in both
                                    database */

                };

        break;

case 3:         /* any channel requested */

        if ( temp_status == 4 || temp_status == 0)
                        /*      Both channels services are allowed and free or B1
                                free */

                state = (temp_status | 1);
```

```
                                        /* set B1 busy */

        else if(temp_status == 5)
                                        /* B2 free */

                state = (temp_status | 2);
                                        /*      assign B2 & set it busy */

        else if(temp_status == 1){
                                        /* no channel free since B2 is not allowed */

                reason = 0x11;
                                      /* class: normal event.
                                         value: user busy. */

                status (reason, refno);

                printf(" B2 service is not allowed\n");

                printf (" no update to all database \n");

                goto last;
                        /*  go back with out saving the lines status in both database */

                };

    };  /* switch */


index = ptable -> originating[refno];
                                /*      get originating user number*/

printf (" originating user number =  ");


printf ("%x\n", index);

printf (" status_orig (in working db) before updating = ");

printf ("%x\n", (ptable -> status_orig[refno]));

printf (" status in perm_db before updating to: ");

printf ("%x\n", perm_db[1][index]);

ptable -> status_orig[refno] |= state;
                        /* update working database */
```

```
        perm_db[1][index] |= state;
                        /* update permanent db */

        printf (" status_orig (in working db) after updating = ");

        printf ("%x\n", (ptable -> status_orig[refno]));

        printf (" status in perm_db after updating to : ");

        printf ("%x\n", perm_db[1][index]);

        }

else{                   /*  check if the destination can provide the service and the lines are
                            not busy */

        temp_status = ptable -> status_dest[refno];
                        /* get user status from working db */

        printf(" status_dest before updating = ");

        printf("%x\n", temp_status);

        if (temp_status == 0 || temp_status == 4){
                                /* B1 free for destination */

           ptable -> status_dest[refno] = temp_status | 1;
                        /* set B1 busy for destination */


           index = ptable -> destination[refno];
                                /* get destination user number */

           perm_db[1][index] |= 1;
                                /* update permanent db */

           printf ("status_dest updated to : ");

           printf ("%x\n", (ptable -> status_dest[refno]));

        }

        else if(temp_status == 5){      /* B2 free */

           ptable -> status_dest[refno] = temp_status | 2;
                        /* set B2 line busy for destination*/

           index = ptable -> destination[refno];
```

```
                                    /* get destination user number */

            perm_db[1][index] |= 2;     /* update permanent db */

            printf ("status_dest updated to : ");

            printf ("%x\n", (ptable -> status_dest[refno]));

        }

        else{

            reason = 0x11;
                            /* class: normal event */
                            /* value: user busy */

            status(reason,refno);

            printf(" service can not be provided by destination \n");

        };

    };      /* if */


last:   return;

}



/*************************************************************************
set up procedure: this procedure is assumed only the bearer capability;
            channel identification; originating and destination
            information field are used.
input parameters: pmsgp (pointer to ISDN-msg )
            len (length of ISDN-msg)
            ptable (pointer to the database table)
output parameters: no.
*************************************************************************/

setup (pmsgp,len, ptable, refno, perm_db)

short   **pmsgp;

short   len;
```

```
struct table *ptable;

unsigned short  refno;

unsigned int perm_db[3][11];

{

      int     test;

      unsigned long   temp = 0;

      unsigned long   proceeding = 0x80000200;
                                   /* fixed format
                                     for proceeding
                                     message,from left:
                                    80 = protocol discriminator.
                                    00 = refno, to be update.
                                    20 = message type ( call
                                         proceeding ).
                                    00 = not used. */

      printf (" setup procedure in progress \n ");


      while (len >0) {

            test = **pmsgp;
                  /*        get information field identifier. This represent length of ISDN
                            message when last information field of set up message processed
                            */

            *pmsgp += 1;
                  /* update ISDN messages buffer pointer (msgp) */

            len -= 1;
                  /* decrement length of the current message */


            switch (test) {

                  case 0x04:

                        len = bearer (pmsgp,len,ptable);

                        break;

                  case 0x18:
```

```
                    len = channel_id (pmsgp, len,
                                refno, ptable);

                    break;

                case 0x6c:

                    len = originating (pmsgp, len, ptable,
                                refno, perm_db);

                    break;


                case 0x70:

                    len = destination (pmsgp, len, ptable,
                                refno, perm_db);

                    break;

            }       /* switch */


        }       /* while */

        /* construct the call proceeding message, this always send after
           the setup message */

        temp |= refno;                  /* get refno */

        temp <<= 16;

        proceeding |= temp;  /* construct the message */
                /* now the call proceeding message is ready to send */

        proceeding &= 0xff00ff00;               /* clear refno for next run */

        return;

}
```

```
/*********************************************************************
                    Alerting procedure
function: received alert message from the called terminal, processed by the
        network and send to the calling terminal to indicate that the
        called terminal has begun to notify human user of the incoming call.
input parameters: len,
                pmsgp,
                refno.
output parameters: non.
*********************************************************************/

alerting(pmsgp, len, refno, ptable)

short    len;

short   **pmsgp;        /* pmsgp = pointer to ISDN messages pointer */

unsigned short  refno;

struct table *ptable;

{

      unsigned short  reason;

      if (ptable -> callref[refno] != refno){
                                        /* refno is not valid */

            reason = 0x53;
                        /* class: invalid message,
                           cause: call identity does not exist */

            status(reason, refno);

            printf (" error in processing alert message\n");

          }

      else  printf(" alert message processed successfully \n");

      return;


}
```

```
/*****************************************************************************
                    connect procedure
function: message received from the called terminal, processed and passed to
        calling terminal to indicate call accepted by the called terminal. the
        processing is involved in testing the validity of the call and
        start charging. Only mandatory fields are used at this model. More
        serviced can be added in the future.
input parameters: pmsgp,
                  len,
                  ptable,
                  refno.
output parameters: non.
*****************************************************************************/

connect (len, ptable, refno, perm_db)

short   len;

struct  table   *ptable;

short   refno;

unsigned int    perm_db[3][11];


{

        int     charging;

        unsigned short  reason;

        unsigned int    index;

        unsigned long   connect_ack = 0x08000f00;
                                        /* connect acknowledge message
                                    without refno. From left:
                                    08= protocol discriminator,
                                    00= reference number to be
                                       updated later on,
                                    0f= message type: connect_ack,
                                    00= not used yet. */

        unsigned long   temp;

        if (ptable -> callref[refno] != refno){
                                        /* refno is not valid */

                reason = 0x53;
```

```
                        /* class: invalid message,
                           cause: call identity does not exist */

        status(reason, refno);

        printf (" error in processing connect message\n");

        return;

      }

    else{

        index = ptable -> originating[refno];

        perm_db[2][index] += 10;
                        /*      add charge to the charging field of the calling terminal only,
                                here we add 10 cent..dummy! */

        printf(" connect message processed successfully \n");

        temp |= refno;

        temp << 16;
                        /* adjust refno to be used to construct connect_ack message */

        connect_ack |= temp;
                        /* construct connect_ack message */
                /* connect_ack message is ready to send */

        temp &= 0xFF00FF00;
                        /* clear refno in connect_ack message for next use */

        printf(" connect_ack message send to appropriate terminal \n");

        return;

      };


}
```

```
/*****************************************************************************
                    disconnect procedure
function: this receive the disconnect message from the terminals ( either
        calling or called terminal), then stop charging , pass the
        message to other side, and send back to the terminal who's
        generate this message the release message.
input parameters: pmsgp,
                  len,
                  ptable,
                  refno,
                  perm_db.
output parameters: none.
*****************************************************************************/

disconnect (pmsgp, len, ptable, refno, perm_db)

unsigned short  **pmsgp;

unsigned short  len;

struct table    *ptable;

unsigned short  refno;

unsigned int    perm_db[3][11];

{
      unsigned int    index;


      unsigned long   release[2];

      unsigned long   temp;

      unsigned short  test;

      unsigned short  reason;

      test = **pmsgp;         /* get inf. element identifier */

      if (test != 0x08){
                                /*    check the availability of the mandatory information element
                                      field, here only cause is mandatory */

            reason = 0x5D;
                                /* class : (101) invalid message
                                   value: mandatory information element is missing */
```

```
        status(reason, refno);

        printf (" disconnect message is incomplete \n");

        *pmsgp += len;  /* skip this message */

        return;

        }

else{                   /* mandatory fields are exist */

        *pmsgp += len;  /* skip the rest of this message */

        index = ptable -> originating[refno];
                        /* get the calling terminal number */

        perm_db[2][index] += 3;
                        /* add an extra charge to the calling terminal charging*/

        release[0] |= 0x08004D08;
                                /* construct the release message. From left:
                              08 = protocol discriminator,
                              00 = refno, to be updated,
                              4D = message type (release),
                              08 = inf. element identifier
                                   (cause).   */

        temp = refno;

        temp <<= 16;                    /* adjust refno */

        release[0] |= temp;

        release[1] |= 0x02009000;
                                 /* construct second word of release message, from left:
                              02 = length,
                              00 = coding standard CCITT
                                   and location ( network).
                              09 = cause class: normal event.

        release[0] &= 0x08004D08;
                                 /* clear refno filed for next run */

              /* release message is ready to send*/

        printf ("disconnect message process successfully and release");
```

```
            printf (" message send to appropriate terminal \n");

            return;

        };

}


/*****************************************************************************
                    release procedure
function: this receive the release message from the terminal which does not
          generate the disconnect message, release all the information in the
          working database except the reference number. It send later the
          release complete message to this terminal.
input parameters: pmsgp,
                  len,
                  ptable,
                  refno.
output parameters: none.
*****************************************************************************/

release (pmsgp, len, ptable, refno)

unsigned short  **pmsgp;

unsigned short  len;

struct table    *ptable;

unsigned short  refno;

{
        unsigned long   release_comp[2];

        unsigned long   temp;

        unsigned short  test;

        unsigned short  reason;

        test = **pmsgp;         /* get inf. element identifier */

        if (test != 0x08){
                                /*      check the availability of the mandatory information element
                                        field, here only cause is mandatory */
```

```
        reason = 0x5D;
                        /* class : (101) invalid message
                           value: mandatory information element is missing */

           status(reason, refno);

        printf (" release message is incomplete \n");

        *pmsgp += len;  /* skip this message */

        return;

        }

else{                   /* mandatory fields are exist */

        *pmsgp += 1;

        len -= 1;

        *pmsgp += 1;
                        /*      update the ISDN message buffer pointer msgp, nothing to
                                be done here to the rest of the cause information element
                                */

        len -= 1;

        *pmsgp += 1;    /*      pass pointer to the second byte of cause info. element
                                (cause) */

        len -= 1;

        ptable -> originating[refno] = 0;
                                        /* clear working database except refno */

        ptable -> destination[refno] = 0;

        ptable -> status_orig[refno] = 0;

        ptable -> status_dest[refno] = 0;

        ptable -> service_req[refno] = 0;


        release_comp[0] |= 0x08005A08;
                                /* construct the release complete message. From left:
                                08 = protocol discriminator,
                                00 = refno, to be updated,
```

```
                                    5A = message type (release),
                                    08 = inf. element identifier
                                        (cause).   */

            temp = refno;

            temp <<= 16;                    /* adjust refno */

            release_comp[0] |= temp;

            release_comp[1] |= 0x02009000;
                                    /* construct second word of release message, from left:
                                    02 = length,
                                    00 = coding standard CCITT
                                        and location ( network).
                                    09 = cause class: normal event.

            release_comp[0] &= 0x08004D08;  /* clear refno filed for
                                        next run */

                /* release_comp message is ready to send */

            printf ("release message process successfully and release");

            printf (" complete message send back to the terminal \n");

            return;

        };
}


/*************************************************************************
                    release complete procedure
function: this receive the release complete message from the terminal who's
          generate the disconnect message, delete the reference number from
          working database.
input parameters: pmsgp,
                  len,
                  ptable,
                  refno.
output parameters: none.
*************************************************************************/

release_comp (pmsgp, len, ptable, refno)

unsigned short  **pmsgp;
```

```
unsigned short  len;

struct table    *ptable;

unsigned short  refno;

{
      unsigned short  test;

      unsigned short  reason;

      test = **pmsgp;         /* get inf. element identifier */

      if (test != 0x08){
                              /*    check the availability of the mandatory information element field,
                                    here only cause is mandatory */

            reason = 0x5D;
                        /* class : (101) invalid message
                           value: mandatory information element is missing */

            status(reason, refno);

            printf (" release_comp message is incomplete \n");

            *pmsgp += len;  /* skip this message */

            return;

            }

      else{                 /* mandatory fields are exist */

            *pmsgp += 1;

            len -= 1;

            *pmsgp += 1;
                              /*    update the ISDN message buffer pointer msgp, nothing to
                                    be done here to the rest of the cause information element
                                    */

            len -= 1;

            *pmsgp += 1;
                              /*    pass pointer to the second byte of cause info. element
                                    (cause) */
```

```
            len -= 1;

            ptable -> callref[refno] = 0;
                                /* clear working database  refno */

            printf ("release_comp message process successfully\n ");


            return;

        };


}



/*****************************************************************************
                    status procedure
function: to construct a status messages and send it to the appropriate
        terminals.
input parameters: reason,
                refno.
output parameters: none.
*****************************************************************************/

status (reason, refno)

unsigned short  reason;

unsigned short  refno;

{

        unsigned long  status_ptr [2];
                            /* status message is 8 bytes */

        unsigned long temp:


        status_ptr[0] |= 0x08007D08;
                                                /* from left
                                            08 = protocc  discriminator
                                            00 = refno, should be updated
                                            7D = message type (status)
                                            08 = inf. element identifier
                                                  (cause).   */
```

```
        status_ptr[1] |= 0x02820000;
                                        /* from left
                                    02 = length of cause (in bytes)
                                    82 = location :local network.
                                         coding standard: CCITT.
                                    00 = class and value (should be
                                           updated)
                                    00 = not used .   */

        temp = temp | refno;

        temp = temp << 16;          /* align refno on 2nd byte */

        status_ptr[0] |= temp;
                                       /* include refno in status message */

        temp = temp ^ temp;         /* clear temp */

        temp = temp | reason;

        temp = temp << 8;           /* align reason in 3rd byte */

        status_ptr[1] = status_ptr[1] | temp;
                                /* update second word of status message */

             /* message is ready to send  */

        temp = temp ^ temp;     /* clear temp for next run */

        return;

}


/*********************************************************************
Bearer capability procedure: this procedure needs some information
about the system hardware. At the present time this does not do any
thing and it is included because it is mandatory with the setup message.
input parameters: msgp and len.
output parameter: len.
*********************************************************************/

bearer (pmsgp,len,ptable)

short  len;

short   **pmsgp;
```

```
struct table    *ptable;

{
        int inf_leng;           /* information field length */
        printf ("bearer procedure in progress \n ");

        inf_leng = **pmsgp;     /* length of information field accessed */
        *pmsgp += 1;
        if (inf_leng == 0) {    /* if this information field length = 0 */
                len -= 1;
                return (len);
                };
}

/*************************************************************************
Channel identification procedure: only the B-channel select is used
                        here.
input parameters: pmsgp,
                  len,
                  refno (reference number),
                  ptable ( pointer to the database table which is structure).
output parameters:len.
*************************************************************************/
channel_id (pmsgp, len, refno, ptable)

unsigned short  **pmsgp;

unsigned short  len;

unsigned short  refno;

struct table *ptable;

{
        unsigned short inf_leng;
```

```
        unsigned short Bchannel;

        printf (" channel identification procedure in progress\n" );

        inf_leng = **pmsgp;     /* get information field length */

        len -= 1;

        *pmsgp +=1;             /* update ISDN messages buffer (msgp) */

        Bchannel = **pmsgp;

        *pmsgp += 1;

        Bchannel = Bchannel & 0x03;
                                /* mask the low order 2 bit, i.e channel select field */

        ptable -> service_req[refno] = Bchannel;
                                /*      set requested services in working db to be tested later */


        len = len - inf_leng;   /* calculate ISDN message length */

        return (len);

}


/*************************************************************************
Originating address: this to process the originating address information
                     field.
input parameters: pmsgp,
                  len,
                  ptable.

output parameters: len.
*************************************************************************/

originating (pmsgp, len, ptable,refno, perm_db)

struct table *ptable;

short **pmsgp;

short  len;
```

```
int  refno;

unsigned int perm_db[3][11];

{

        int inf_leng;

        short direction;        /* to select either destination or originating */

        unsigned int index;

        unsigned int user_status;

        printf (" originating procedure in progress \n ");

        inf_leng = **pmsgp;     /* get length of this inf. field */

        *pmsgp += 1;

        len -= 1;

        direction = 0;          /* 0 for originating, 1 for destination */

        insert_db (pmsgp, refno, direction, ptable);
                                /* insert in db */

        len = len - inf_leng;
                                /* decrement ISDN message length */

        index = ptable -> originating[refno];
                                /* get originating address */

        user_status = perm_db[1][index];
                                /* get the user status from permanent db */

        ptable -> status_orig[refno] = user_status;
                                /* save user status in working database */

        check_service(ptable, refno, direction,perm_db);

        return (len);
}
```

```
/*************************************************************************
Destination address: to process the originating address information field.

input parameters: pmsgp,
                  len,
                  refno,
                  ptable.

output parameters: len.
*************************************************************************/

destination (pmsgp, len, ptable,refno, perm_db)

struct table *ptable;

short **pmsgp;

short  len;

int  refno;

unsigned int perm_db[3][11];


{

      short inf_leng;         /* length of this information field */

      short direction;        /* to enable insert_db routine to update the database */

      unsigned int    index;

      unsigned int    user_status;

      printf (" destination  procedure in progress\n ");

      inf_leng = **pmsgp;
                        /* get length of this infor. field */

      len -= 1;
                       /* update ISDN message length */

      *pmsgp += 1;
                      /* update the ISDN messages buffer pointer */

      direction = 1;
                      /* 1 = destination address */
```

```
        insert_db(pmsgp, refno, direction, ptable);

        len = len - inf_leng;
                        /* update ISDN message length */


        index = ptable -> destination[refno];
                        /* get originating address */

        user_status = perm_db[1][index];
                         /* get the user status from permanent db */

        ptable -> status_dest[refno] = user_status;
                        /* load user status in working database */

        check_service(ptable, refno, direction, perm_db);

        return (len);

}


/*********************************************************************
Insertdb: This to insert in the data base the address of the
        originating or destination subscriber address.
input parameters : pmsgp,
                refno,
                direction,
                ptable.
output parameters: non.
*********************************************************************/

insert_db (pmsgp, refno, direction, ptable)

short   **pmsgp;

int     refno;

short direction;

struct  table   *ptable;


{

        unsigned int    type;
```

```
        unsigned int    temp;

        type = **pmsgp;
                        /* get address type of numbering addressing plan */


        *pmsgp += 1;

        printf (" insert db procedure in progress \n ");
                        /* assume all calls are local */

        type = type & 0x0070;
                        /* mask the type address field */

        if (type == 0x0040) {
                        /* is it local call */

              temp =**pmsgp;
                         /* access high order two digits */

              *pmsgp +=1;

              temp = temp << 8;
                       /* shift left 8 bits */

              type = **pmsgp;
                        /* get next two digits */

              *pmsgp += 1;

              temp |= type;
                        /* put all digits in one  location */

              if (direction == 0)
                        /* this is originating */

                    ptable -> originating[refno] = temp;
                        /* insert originating address in db */

              else

                    ptable -> destination[refno] = temp;

        };

        return;
}
```

Appendix B

Listing of TEI Assignment Program

```
/*****************************************************************************
```

This program shows the list for the TEI assignment functions. The functions processed in this program are:

1. Any TEI assignment.
2. Specific TEI assignment.
3. TEI removal.

Each of the above functions has processed by more than one procedure. Each procedure activates by certain subfield within the TEI assignment messages, such as TEI-Assign, TEI-Denied, and TEI-Removal message.

The database is organized to keep the records for the terminals which are in use with the reference number of the call. The reference number is used as an index to extract the information from the records inside this database.

In this listing, we shows only two messages to illustrate the program behaviour and responses. Some display messages are printed to trace and debug the program functionality.

This program is used to process different TEI assignment messages. These messages are dynamically scanned during the processing of this program and as described in Chapter 3. This program is compiled and tested under R2000 RISC-based machine.

```
*****************************************************************************/

#include <stdio.h>

#define IDENTIFY_REQUEST       0x01
#define IDENTIFY_ASSIGNED      0x02
#define IDENTIFY_DENIED        0x03
#define IDENTIFY_REMOVE        0x06
#define IDENTIFY_CHECK_RESPONSE 0x05
#define MANAGEMENT_ENTITY      0x0f
#define FREE                   0x00
#define BUSY                   0xff

struct database_record{

                unsigned int ri;

                unsigned int taken;

              };    /* keep track of tei and reference number assigned */


unsigned int no=1; /* keeps track of packet being process */

main ()
```

```
{

/* functions used */

  int tei_removal();

  void assign_identify();

  void preferred();

  void any_tei();

  int free_tei();

/* variable declarization*/

  register int i;

  static struct database_record database[64];

  static unsigned packet[]={
                    /*    the following messages are just an  example  of  the  TEI
                          messages */

                 /* first packet */
            0x08,          /* length of packet                          */
            0x3f,          /* SAPI                                  */
            0x7f,          /* terminal endpoint identifier              */
            0x10,          /* control                               */
            0x0f,          /* management entity identifier              */
            0x00,          /* least significant byte of reference number*/
            0x00,          /* most significant byte of reference number */
            0x01,          /* message type - identify request           */
            0xff,          /* action indicator - any tei acceptable     */


            /* second packet */
            0x08,          /* length of packet                          */
            0x3f,          /* SAPI                                  */
            0x7f,          /* terminal endpoint identifier              */
            0x10,          /* control                               */
            0x0f,          /* management entity identifier              */
            0x01,          /* least significant byte of reference number*/
            0x00,          /* most significant byte of reference number */
            0x01,          /* message type - identify request           */
            0xff,          /* action indicator - any tei acceptable     */
```

```
                    };
  int number_of_data_packets=2;  /* number of test packet */

  short int unsigned *send_packet;

  short int unsigned packet2[7];

  int length;                   /* length of packet */

  int error;

/*****************************/
/* Get packet and Manupilate  */
/*****************************/

   int j;

   i = 0;

   while (no<=number_of_data_packets)

    {

     j=0;

     length=packet[i++];
                /* get length of packet */

     if (packet[i++]!=0x3f)
                /* SAPI value wrong, skip to the next packet*/

      {

       i += length;        /* goto next packet */

       printf("SAPI value wrong\n");

       continue;

      }

     if (packet[i++]!=0x7f)
                        /* TEI value wrong, skip to the next packet */

      {
```

```
    i += length - 1;       /* goto next packet */

    printf("TEI value wrong\n");

    continue;

  }

i++;

     /* get control value */

packet2[j++]=packet[i++];
                    /* get management entity */

if (packet2[i-1]!=MANAGEMENT_ENTITY)
                    /* management entity wrong, skip to the next packet */

  {
    i += length -4;        /* goto next packet */


    printf("mangement value wrong\n");

    continue;

  }

packet2[j++]=packet[i++];
                     /* get reference number(LSB) */

packet2[j++]=packet[i++];
                    /* get reference number(MSB) */

packet2[j++]=packet[i++];
                    /* get message type (identity type)*/

packet2[j++]=packet[i++];
                    /* get action indicator */

switch (packet2[j-2])
                    /* check message type */

  {

     case IDENTIFY_REQUEST: assign_identify(packet2,database);    /* try to allocated */

                            send_packet=packet2;
```

```
                    /* send return packet  */
                    packet[i-length+6]=send_packet[3];

                    packet[i-length+7]=send_packet[4];

                  break;


      case IDENTIFY_CHECK_RESPONSE: error=tei_removal(packet2,database);

                    if (error==0)

                     {

                       send_packet=packet2:

                       /* send return packet */
                       packet[i-length+6]=send_packet[3];

                       packet[i-length+7]=send_packet[4];

                     }


                    else

                     {

                       /* ignored packet */

                     }
                    break;

      default: printf("unknown message type\n");

            break;
    }

  }

}
```

```
/*********************************************************
  Name              : assign_identify
  Purpose           : to assign identify to terminal
  Input Parameters  : packet2
                      database
  Output Parameters : database
  Functions called  : preferred
                      free_tei
                      any_tei
*********************************************************/

void assign_identify(packet2,database)

short int unsigned packet2[];

struct database_record database[63];

{

 int not_found;


  if (packet2[4] >=0x80 && packet2[4] <=0xfe)

    {

      preferred(database,packet2);
                        /* assign the specified TEI value requested by terminal */

    }

   else

    {

      not_found=free_tei(database,packet2);
                        /* check if database is free and reference # is not in used */

      if (not_found==0x80)

        return;         /* reference # in used */

      if (not_found)

        {
                        /* database full - return denied value */

          packet2[3]=IDENTIFY_DENIED;
```

```
            printf("%d : identify denied - database full\n",no++);
          }
        if (packet2[4] == 0xff)
          {
            any_tei(database,packet2);  /* assign any TEI value to terminal */
          }
        else
  return;
      }
}


/*********************************************************
   Name              : free_tei
   Purpose           : to clear the TEI
   Input Parameters  : packet2
                       database
   Output Parameters : database
                       not_free = 0x00  (database full)
                                  0x01  (database free)
                                  0x80  (reference #
                                        already taken)
   Functions called  : none
 *********************************************************/

int free_tei(database,packet2)

short int unsigned packet2[];

struct database_record database[63];


{
   int not_free;

   register int i;
```

```
   not_free=1;
   for (i=0;i<63;i++)
    {
      not_free=not_free&&database[i].taken;
      if ((database[i].ri==(packet2[2]<<0x08)+packet2[1])&& (database[i].taken==BUSY))
       {
         /* reference # already in use */
         packet2[4]=IDENTIFY_DENIED;
         printf("%d : ignored identify request - reference # already in database\n",no++);
         return(0x80);
       }
    }
  return(not_free);
}

/*************************************************************
   Name              : any_tei
   Purpose           : to assign any TEI to a terminal
   Input Parameters  : packet2
                       database
   Output Parameters : database
   Functions called  : none
*************************************************************/
void any_tei(database,packet2)
short int unsigned packet2[];
struct database_record database[63];
{
```

```
  register i;

  i=0;

  /* find first free value */

  while(database[i].taken==BUSY && i<=62)

    i++;

  packet2[3]=IDENTIFY_ASSIGNED;

  database[i].ri = (packet2[2]<<0x08) + packet2[1];
                        /* stored reference number in database */

  database[i].taken=BUSY;

  printf("%d : identify assigned to TEI value %d\n",no++,i+0x40);

  packet2[4]=i+0x80;

}



/*********************************************************
  Name              : preferred
  Purpose           : to assign the preferred TEI to
                    a terminal
  Input Parameters  : packet2
                    database
  Output Parameters : database
  Functions called  : none
*********************************************************/


void preferred(database,packet2)

short int unsigned packet2[];

struct database_record database[63];

{
   if (database[packet2[4]-0xc0].taken==FREE)
     {

       /* assigned preferred value to terminal */
```

```
        database[packet2[4]-0xc0].taken=BUSY;

          /* stored the low and high byte as one number */
        database[packet2[4]-0xc0].ri = (packet2[2]<<0x08) + packet2[1];

        packet2[3]=IDENTIFY_ASSIGNED;

        printf("%d : identify assigned to index(referred) %d\n",no++,packet2[4]-0x80);

      }

    else

      {
        /* preferred value already taken */
        packet2[3]=IDENTIFY_DENIED;

        printf("%d : identify denied - preferred value taken(%d)\n",no++,packet2[4]-128);

      }

    return;
}



/*********************************************************
  Name              : tei_removal
  Purpose           : to removed TEI
  Input Parameters  : packet2
                      database
  Output Parameters : database
                      function returns = 0  (no error)
                                         1  (error)
  Functions called  : none
*********************************************************/

int tei_removal(packet2,database)

short int unsigned packet2[];

struct database_record database[63];

{

  register i;
```

```
    if (database[packet2[4]-0xc0].ri== (packet2[2]<<0x08) + packet2[1]
                &&database[packet2[4]-192].taken!=FREE)

     {

       printf("should not occured\n");

       return(1);                    /* error occurred */

     }

    packet2[6]=IDENTIFY_CHECK_RESPONSE;

    if (packet2[4]==0x7f)

     {

      /* clear the complete database */

       for (i=0;i<=64;++i)

         database[i].taken=FREE;

       printf("%d : Removal of all TEI value from equipment\n",no++);

     }

   else

     {

       /* clear specified TEI value in database */
       database[packet2[4]-0xc0].taken=FREE;

       printf("%d : Removal of TEI value %d\n",no++,packet2[4]-0x80);

     }

   return(0);

}
```