A PARALLEL ALGORITHM FOR ASN.1 ENCODING/DECODING

by

Carlton Charles Agnel Joseph

B.Eng. (EE), McGill, 1990.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF ELECTRICAL ENGINEERING

We accept this thesis as conforming

to the required standard

T[redacted]**IBIA**

September 1992

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

(Si

Department of ELECTRICAL ENGINEERING.

The University of British Columbia
Vancouver, Canada

Date Oct. 13, 1992.

DE-6 (2/88)

Abstract

Computers differ in many ways: the characters may be coded in ASCII, EBCDIC or UNICODE; integers can vary in size; processors can operate in Big-Endian or Little-Endian format; and programming languages can use different memory representations for data. The International Standards Organization (ISO) provides an Open System Interconnection (OSI) seven-layer reference model[20] to facilitate heterogeneous inter-computer communication.

Multimedia technology places new demands on inter-computer communication speeds because of the higher bandwidth requirements of voice and image data. Various implementations of high speed networks meet this demand at the lower layers. The presentation layer of the ISO reference model is the dominant component of the total protocol processing time[5]. Hence, the presentation layer represents a bottleneck in communication systems that use high speeds networks. The presentation layer is traditionally implemented in software but now a combined parallel hardware/software approach is taken.

Prior to this thesis, VASN.1[3][24][25] was the only parallel hardware/software combination that achieved high throughput in the presentation layer. The parallel algorithm of VASN.1 limits the scalablity of hardware and hence the achievable parallelism. VASN.1 users are expected to use the parallel data structure[1] that it uses. This limitation reduces the flexibility for VASN.1 users and incurs an additional overhead for handling the data structure.

The parallel system presented in this thesis avoids the serious problems found in VASN.1. A new parallel algorithm has been developed which provides users with flexible serial data structures[2]. The encoding algorithm converts flexible serial data structures into parallel information which allow simultaneous processing. The conversion from a serial to a parallel information has not been previously explored in the literature. The decoding algorithm also provides maximum parallelism while providing a flexible serial data structure for applications to use. The algorithm has been designed in such a way that any changes in

1 A parallel data structure is a data structure that contains extra information required for a parallel algorithm.

2 A serial data structure is a data structure that holds no extra information required for a parallel algorithm.

the ASN.1 specification can be easily incorporated into the system. The final advantage is that the hardware scales gracefully and allows for more parallelism than VASN.1. The initial simulations are promising and show that the throughput achievable by the parallel algorithm is five times greater than that achievable by a serial implementation.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my gratitude to Dr. Mabo Ito for his suggestions and for supporting me throughout my graduate studies. I also would like to thank Dr. Gerald Neufeld for his insight on ASN.1 and his very valuable comments. A special thanks to Mike Sample for providing me with statistics on ASN.1 processing and Donna Fuller for her support during the preparation period of this thesis. Lastly, I would like to thank my parents for their continuous support, encouragement and understanding.

1 Introduction

Section 1.1 of this chapter covers the need for communication in heterogeneous networks. Section 1.2 presents information showing that the computation cost of the presentation layer accounts for the majority of the total OSI protocol cost[5]. Section 1.3 suggest a parallel approach to reduce this cost. Section 1.4 discusses the approaches of other researchers and their relative advantages and disadvantages. Section 1.5, the final section of the introduction, states the objectives and the outline of this thesis.

1.1 Communication in Heterogenous Networks

In industry, at home and in educational institutions, there are a large variety of computer hardware configurations available from various manufacturers. There exists a need for computers based on different hardware structures to communicate with one another. Several vendors address this issue by implementing protocols so that networks with heterogenous computers can communicate. Two of these vendor-specific protocols are Sun Microsystems' XDR[6] and Apollo Computer's NDR[8].

To aid in the goal of truly inter-operable heterogenous systems, ISO has constructed the OSI seven-layer protocol reference model[17][20]. Figure 1 shows the top three layers of the OSI seven-layer model. Any layer n of **Host A** communicates with its peer on layer n of **Host B** using a preset protocol. For instance, the presentation layer's peers in **Host A** and B communicate using a presentation protocol. The protocol exchanges information by exchanging data units known as protocol data units (PDU)[3]. The presentation layer in **Host A** and **Host B** do not directly exchange PDUs. The PDU is propagated down through to the lower layers where layer one is responsible for exchanging information between **Host A** and **Host B**. Within the OSI model the presentation layer is responsible for the transformation of information from a machine-dependent format to a machine-independent format and vice-versa. Section 1.1.1 presents an outline of how this is achieved in a standard way.

[3] A list of all the acronyms used in this thesis is presented in Appendix A

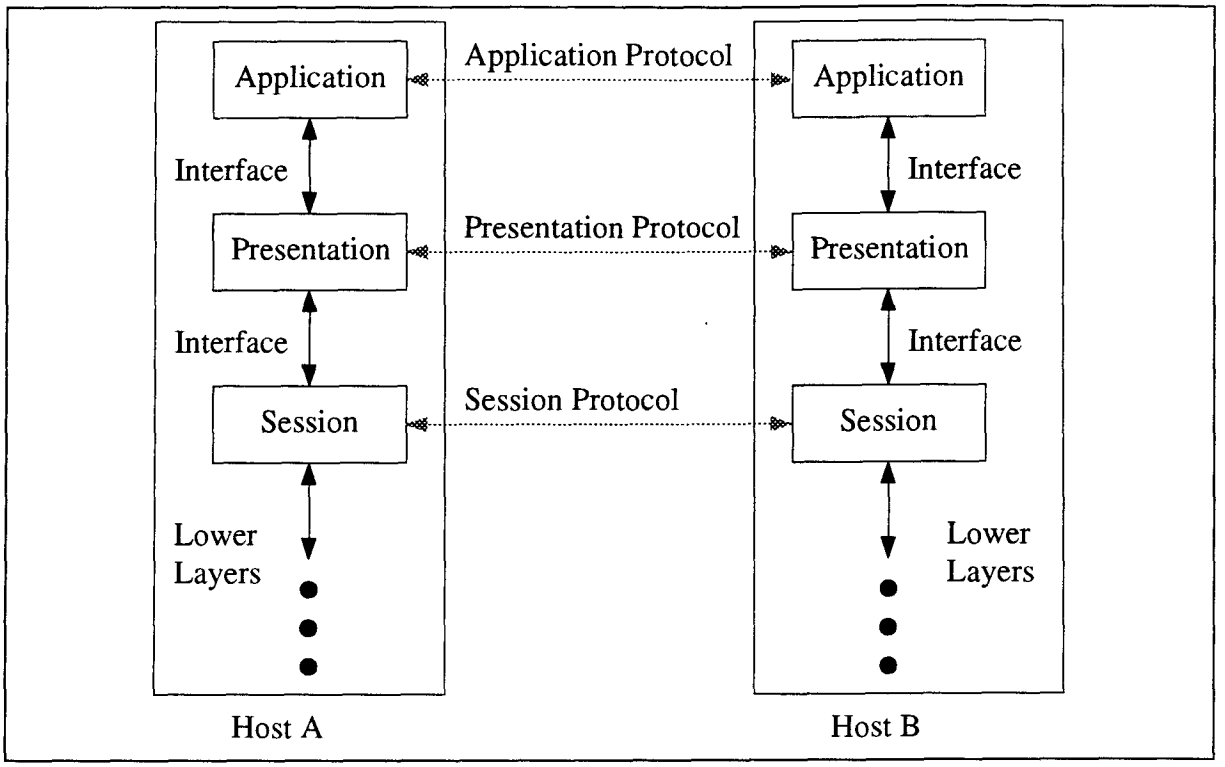

Figure 1 Abbreviated OSI Seven Layer Model

1.1.1 ISO Presentation Layer Conversion Standard

This section presents a summary of the roles that Abstract Syntax Notation One (ASN.1[26]) and the Basic Encoding Rules (BER[27]) play in the presentation layer. References [11][4] present additional detail on BER and ASN.1.

ASN.1 can specify the data structures of the presentation protocol data units (PPDU) when two machines communicate (figure 1). The ASN.1 specification can then be compiled into routines that encode/decode to/from *machine-specific/machine-independent* formats. BER are used when transforming data from a machine-specific format to a machine-independent format and vice-versa. Figure 2 shows a possible communication scenario, where **Host A** transmits information to **Host B**.

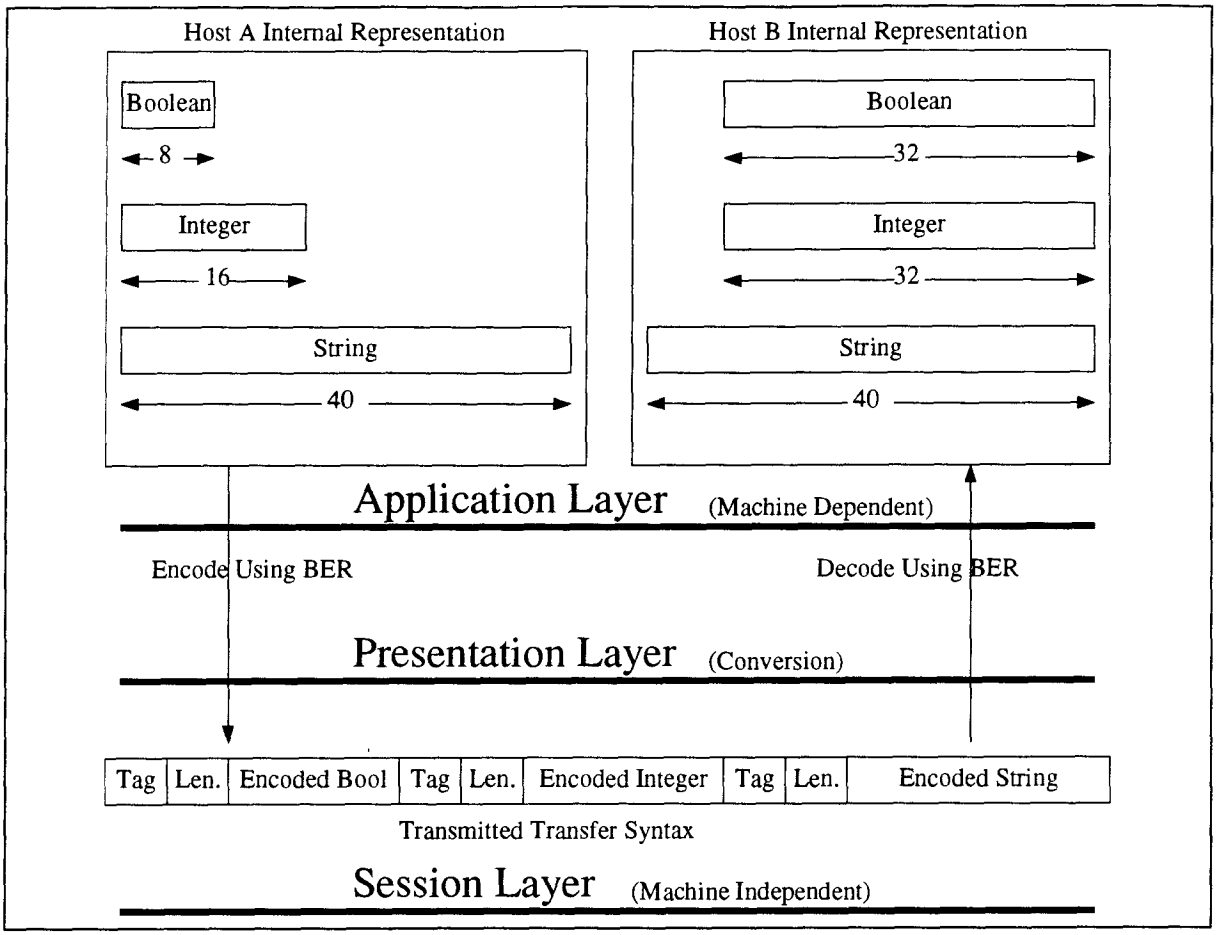

Figure 2 Example of Local Representation to Transfer Syntax

In figure 2 one can see that the internal representation of the data on the two machines is different and hence BER are used to convert the data into a generic transfer syntax (TS) and then back to an internal representation. This diagram shows that the TS consists of a tag, a length and the encoded value (TLV[4]). In the triples, the tag represents the information that is stored in the encoded part, the length specifies the encoded part's length and the encoded part holds the actual encoded information. BER specify how information is encoded/decoded. Figure 3 shows an ASN.1 representation of a data structure that holds personnel information. Notice that the data representation is similar to those used in most high level programming languages.

[4] TLV = Tag Length Value.

```
PersonnelInformation ::= SET {
          Name IA5String,
          Age  INTEGER,
          Sex  BOOLEAN }
```

Figure 3 ASN.1 Description of Personnel Information

The ASN.1 description given in figure 3 is machine-independent and the internal representations shown in figure 2 are the machine-specific information for one person. **Host A** and **Host B** in figure 2 are exchanging personnel information for one person. When both the machines receive an ASN.1 description of *PersonnelInformation* they are responsible for maintaining value instances of this data in their machine-specific format and converting it to/from a machine generic format when transmitting/receiving.

1.2 Presentation Layer Cost

When using the OSI protocol stack, transferring from a machine-dependent to a machine-independent format can consume 97%[5] of the total processing time. Unpublished simulations carried out at the University of British Columbia (UBC) show that the cost is closer to 70% when the more efficient CASN.1[12][5] implementation is used in the presentation layer. The data gathered by Takeuchi[19] can also provide the processing cost associated with the presentation layer. Takeuchi assumes the processing costs per packet of the OSI lower layers are constant[6]. In the presentation layer the cost per byte is calculated for the ASN.1 PersonnelRecord shown in Appendix E. For the ASN.1 PersonnelRecord specification, the encoding/decoding instruction count for the presentation layer is $Encode = 350 + 27 * bytes$ and $Decode = 400 + 60 * bytes$[7] . The spread sheet extract in figure 4 shows the cost of the presentation layer with reference to the total protocol processing cost. **Data Set One** shows the cost for sending/receiving PDUs at the lower layers. **Data Set Two** shows that the

5 Details of CASN.1 will be given later.

6 The assumption is correct if the CRC checking in the transport layer is omitted or computed by a hardware assist at no cost.

7 Page 55 & Appendix C page 106–108 of Ref.[19].

presentation layer can consume a significant portion of the processing power as the packet size increases.

Data Set One

Layer	Send	Receive
2	40	45
3	160	160
4	400	375
5	400	450
Total	1000	1030

Data Set Two

Bytes/Pkt	Layers 2-5		Layer 6		% Layer 6	
	Encode	Decode	Encode	Decode	Encode	Decode
100	1000	1030	3050	6400	75.31	86.14
200	1000	1030	5750	12400	85.19	92.33
400	1000	1030	11150	24400	91.77	95.95
800	1000	1030	21950	48400	95.64	97.92
1600	1000	1030	43550	96400	97.76	98.94
3200	1000	1030	86750	192400	98.86	99.47
6400	1000	1030	173150	384400	99.43	99.73
12800	1000	1030	345950	768400	99.71	99.87

Figure 4 ASN.1 Processing Cost.

The ASN.1 PersonnelRecord type in Appendix E consists of IA5String string types and one integer type. This leads to two important conclusions:

- Since the cost for encoding/decoding a strings is low, the costs attributed to the presentation layer in figure 4 are mostly due to the encoding/decoding of the tag and the length.

- Figure 4 already shows that the presentation layer occupies a significant portion of the protocol stack even with a simple encoding/decoding to perform. The percentage of the processing time that the presentation layer uses is significantly higher for complex types such as integers and reals.

1.3 Initial Parallel Analysis

The initial parallel analysis is carried out by gathering statistics for the CASN.1[12] tool. CASN.1 takes an ASN.1 specification as input and generates C code that is capable of encoding/decoding PPDUs. In CASN.1, procedure calls are made to primitive encoding/decoding routines that are shown in table 1. These calls take primitive types and transfer them back and forth from machine-specific format to machine-independent transfer syntax TLV triples.

ASN.1 Type	Encode Routine	Decode Routine
Boolean	Encode_bool()	Decode_bool()
Integer	Encode_int()	Decode_int()
Bit String	Encode_bits()	Decode_bits()
Octet String	Encode_octs()	Decode_octs()
Null	Encode_null()	Decode_null()
Object Identifier	Encode_oid()	Decode_oid()
Universal Time	Encode_UTCTime()	Decode_UTCTime()
Generalized Time	Encode_GNLTime()	Decode_GNLTime()
External	Encode_ext()	Decode_ext()

Table 1 CASN.1 Primitive encoding/decoding Routines

By using primitive encoding/decoding routines with constructor types such as sets and sequences, complex data types can be described. A set of primitive data types is shown in figure 3. The only difference between sets and sequences is that in a sequence the order of the primitive data types is maintained and in sets it is not. The use of set and sequence data types can be nested.

The primitive encoding/decoding calls constitute most of the time in CASN.1. An estimate of the number of instructions used by these primitive calls was constructed by compiling CASN.1's implementation of the primitives to assembler[8]. Table 2 presents the total number of assembler instructions counted[9]. The static analysis in table 2 shows that significant gains are achieved if the primitives are executed in parallel. Significant speedup is gained even if the cost incurred by the parallel algorithm is high. Table 3 shows how the cost of a parallel algorithm is amortized over the number of processors. The table ignores the requests placed on scarce resources such as a shared bus. It also assumes that the parallel algorithm does not incur any communication overhead. These effects are apparent in the final evaluation of the system and thus the linear speedup shown in table 3 is not achieved.

Primitives	Encoding Instruction Count	Decoding Instruction Count
Integer	583	479
Boolean	529	457
Bits	668	523
OCTET String	672	477
Null	515	432

Table 2 Primitive Routines Instruction Count.

Processors→	1	1	2	3		m
Primitives↓	Serial	Parallel	Parallel	Parallel		Parallel
1	100	200	200/2	200/3		200/m
2	200	400	400/2	400/3		400/m
n	100n	200n	200n/2	200n/3		200n/m

Table 3 Parallelism Speed Gain

The final analysis of the primitive encoding/decoding routines proves that even when encoding/decoding is optimized for the simplest cases parallelism is applicable due to

[8] The cc —S option is used on a SPARC 2.

Appendix C provides the pro and cons of using a static analysis.

[9] The count includes the cost of encoding/decoding the tag and the length.

the number of instructions that need to be performed. Table 4 presents the number of instructions[10] executed for encoding/decoding some of the primitive types assuming the following simplifications:

- All the tags and lengths are in short form.
- The instruction counts for NULL, BOOLEAN, INTEGER and OCTET String, include the instructions required to encode/decode the tag and length.
- The INTEGER instruction counts were generated for a one byte positive integer.
- The OCTET String instruction counts were generated for octets that start and end on a word boundary. The total length of the octet must be less than 127 bytes.

Routine	TS size in bytes	Encode Instruction Count	Decode Instruction Count
NULL	2	12	13
BOOLEAN	3	18	19
INTEGER	3	20	16
OCTET String	2+n	27+1.5n	21+1.5n
TAG & LEN.	2	17	14

Table 4 Minimal Instructions for Encoding/Decoding

The instruction count presented in table 4 does not consider:

- The cost incurred by procedure calls.
- The cost incurred when allocating memory.
- The variable cost of handling non word-aligned OCTET Strings.
- The cost of handling complex primitive encodings/decodings.
- The cost of maintaining data dependencies within complex structure types.

---

10 The Motorola 88100[10] RISC processor instruction set was used.

The above analysis shows that if the primitive encoding/decoding calls are executed fast then a significant speedup can be achieved. In the best case, the calls return as soon as they are made. This thesis takes the approach that when a primitive call occurs it is pushed onto a FIFO queue for servicing, therefore the calls take little time. The problem of servicing these calls quickly is solved using the architecture proposed in chapter 2.

Multiple processors, reading from the FIFO, are implemented to service the calls quickly. This essentially increases the number of instructions executed in a cycle. If there are n processors, then in one cycle, n instructions are executed while in a single processor system only one instruction is executed. Figure 5 shows how time is saved by encoding in parallel with n processors. The encoded values could be an individual's personnel information as discussed in section 1.1.1. Note that there exists a time skew in the parallel encoding shown in figure 5. A similar skew also exists in the decoding algorithm. The following is the description of the skew.

When encoding, the skew times are associated with the following actions.

- **T1** is the time delay between assigning each primitive call to the FIFO. The instructions executed between the assignments account for the delay and are explained in chapter 3.
- **T2** is the time delay from T1 to the time the primitive call receives service.

When decoding, the skew times are associated with the following actions.

- **T1** is the time required to read the length of the present encode value so that the next value's starting address is known.
- **T2** is the delay between the end of T1 and when computation starts on the next value. The delay is due to the communication required by the parallel algorithm. The communication required by the decoding parallel algorithm is explained in chapter 4.

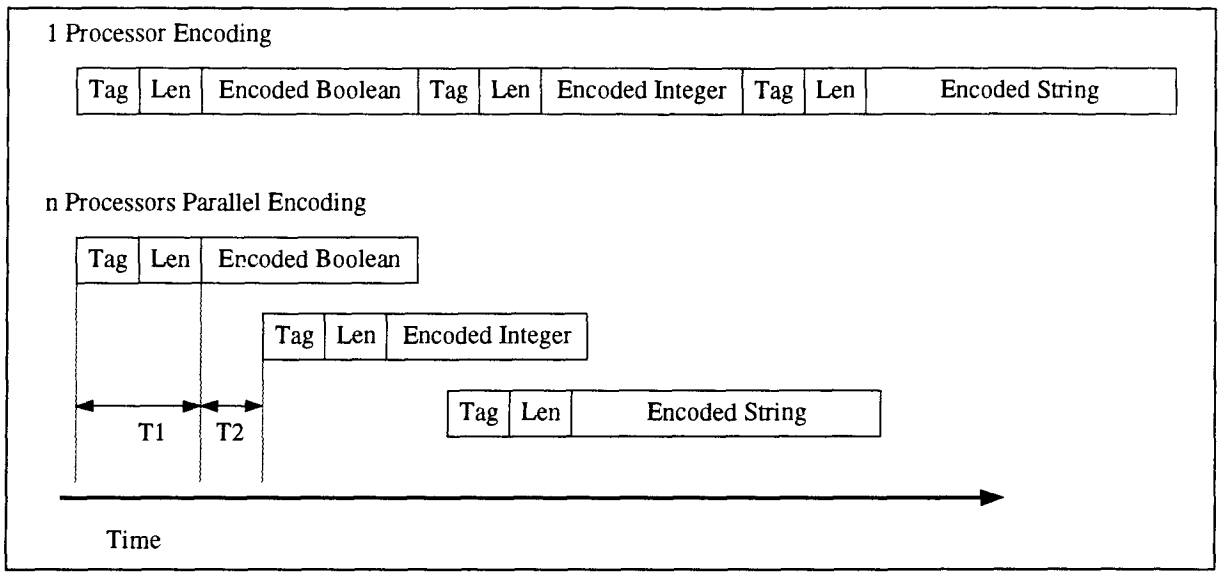

Figure 5 Serial vs Parallel Encode Time

1.4 Previous Work with ASN.1

Some of the previous work carried out in relation to the presentation layer in the ISO model includes the ISODE[23] project, the CASN.1[28] project, the a2c project[15] and the VASN.1[3][24][25] project. Section 1.4.1 briefly discusses the ISODE, CASN.1 and a2c single-processor software used for ASN.1 encoding/decoding. Section 1.4.2 describes VASN.1, a multiprocessor ASN.1 encoder/decoder.

1.4.1 Single Processor ASN.1 Implementations

The ISODE, CASN.1 and a2c projects are single processor programs[11] that take an ASN.1 specification and create encoding/decoding routines to transfer to/from *machine-independent transfer syntax/machine-dependent format.* The problem with using single-processor software to perform this function is that the achievable transfer rates are unacceptably low for high speed networks. A 8 Mbits/s transfer rate was achieved by running a2c on a MIPS R3260 (approximately 18 specmarks) with 48 megabytes of RAM.

[11] The programs are Unix-based C programs.

1.4.2 Multi Processor ASN.1 Implementations

Figure 6 shows the parallel hardware that VASN.1 uses to achieve a higher throughput compared to a single processor implementation.

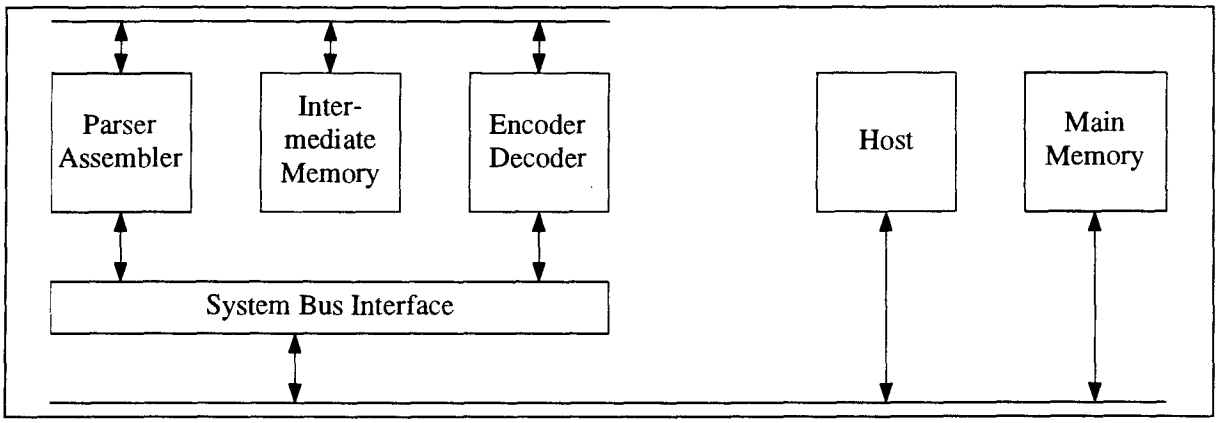

Figure 6 VASN.1 Overview

The following factors limit the hardware used in the design and simulations:

- The design is based on a 16–bit microprocessor; hence, a complicated memory mapping scheme is required to map to today's typical 32–bit architecture. This mapping could be expensive.
- The hardware setup is non-expandable because a ROM holds the encoding/decoding routines.

The above problems could be solved by adapting the algorithm to a 32–bit machine and converting the ROM that holds the encoding/decoding routine to a downloadable RAM. However, VASN.1 fails to eliminate some of the other problems as easily. Figure 7 is the block diagram of the hardware that is used for the Parser/Assembler and the Encoder/Decoder of figure 6. The following factors limit the hardware shown in figures 6 and 7:

- The **Intermediate Memory** block of figure 6 is an associative memory scheme that depends on the algorithm VASN.1 uses. The scaling of the hardware is difficult due

to the associative memory and the **Intermediate Memory** block will not operate when ASN.1 types have long tags.

- Figure 7 shows that two queues are used for servicing the interrupts generated by slave processors. This method for handling interrupts produces unfair arbitration if the sampling rate of the two queues is low[2].
- In figure 7, information exchanged between the slave processors is transmitted on **Bus A** via the **Processor Master**. Information that the slave processor needs from the system or local bus is retrieved on **Bus B** via the **Interface Processor**. The setup is restrictive because it partitions the utilization of the two busses and leads to a high utilization of one bus and a low utilization of the other. The total throughput of the multiple bus system is not exploited.
- In figure 7 the arbiter services the requests for the system bus and for the local bus made by the four slave processors, hence the arbiter can become a bottleneck in the system.

This thesis proposes an implementation that avoids the problems of VASN.1 by choosing a more flexible hardware/software configuration.

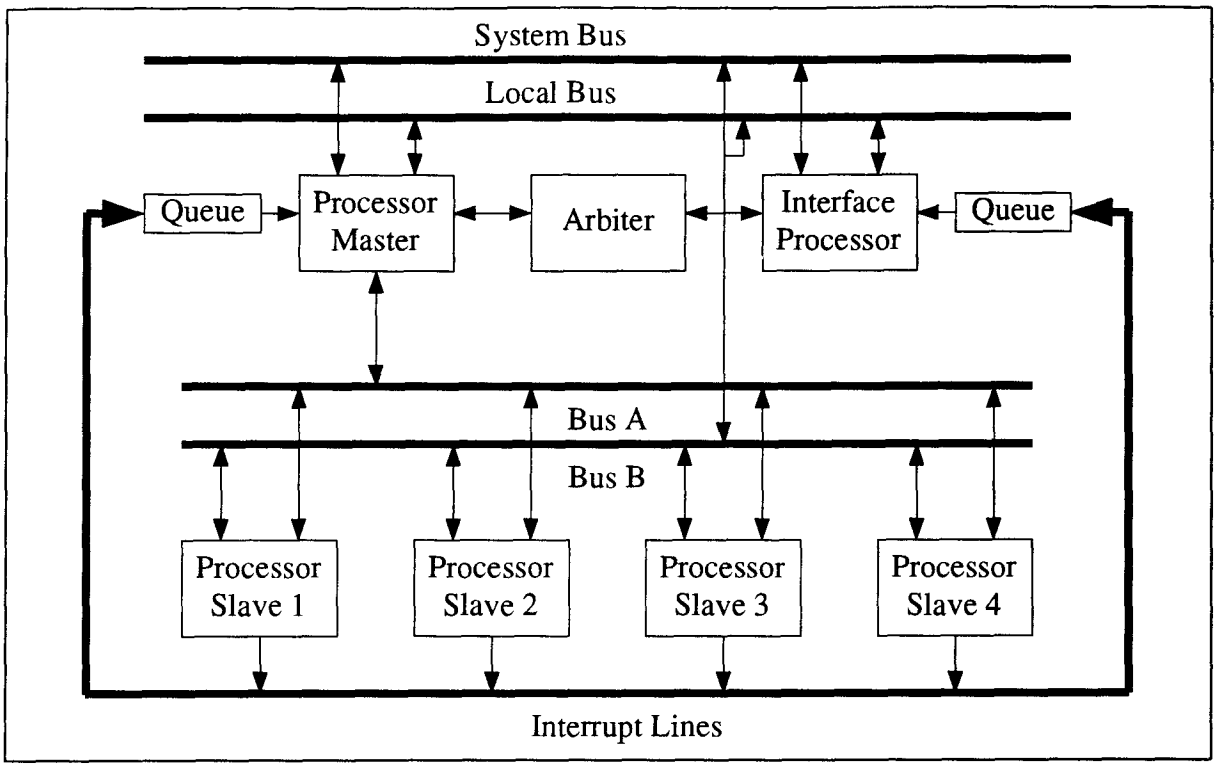

Figure 7 Parser/Assembler and Encoder/Decoder

The encoding/decoding algorithms represent the last hidden cost for VASN.1. The local representation that VASN.1 uses for its encoding input and decoding output is a parallel data structure that holds all the hierarchical information of the equivalent ASN.1 description. The data structure has the following implications:

- An application would either use the parallel data structure if they want to encode/decode data via VASN.1.
- Or an application can convert the parallel data structure into one that is more convenient.

Either case incurs an expensive programming cost because of the lack of flexibility and extraneous information in the data structure. This processing cost is the cost of parallelizing a serial data structure. VASN.1 attributes a zero cost for the processing. The algorithm presented in this thesis provides the application with flexible data structure and does the serial to parallel conversion at a low cost.

1.5 Thesis Objectives and Outline

This section presents a list of thesis objectives that were achieved and the organization of the thesis. Prior to this thesis, VASN.1 was the only parallel system for encoding/decoding ASN.1 types. Therefore many of the objectives listed below are solutions to problems that VASN.1 exhibits.

1. Devise a parallel ASN.1 encoding/decoding algorithm that is capable of higher throughput than present serial implementations.
2. Devise an algorithm that can take serial data and transform it into parallel data for optimal encoding/decoding.
3. Devise an ASN.1 encoding/decoding algorithm that can be incorporated with applications at a minimal cost.
4. Design a scalable parallel architecture that is capable of running the algorithm at high speeds.

The following is the outline of this thesis. Chapter 2 presents the parallel architecture. Chapters 3 and 4 describe the encoding and decoding algorithms in detail. Chapter 5 provides the functional specification of the hardware. Chapter 6 presents the details of how the simulations of the parallel architecture were carried out and chapter 7 presents the simulation results. Chapters 8 and 9 provide the conclusions and suggest directions that future work may follow.

2 Parallel Architecture

Figure 5 shows that in a single processor implementation the time required to encode the data is the sum of the time required to encode each primitive type. In the parallel version the primitive types encode nearly simultaneously thus saving time. The same principle applies to the decoding routines. Section 2.1 provides a high level description of the parallel hardware needed for the efficient execution of the parallel algorithm. Section 2.2 presents an outline of the parallel encoding/decoding algorithm. Section 2.3 explains how the host system and the parallel ASN.1 system share the same address space.

2.1 Hardware Architecture

Figure 8 depicts the parallel setup that allows primitive encoding/decoding routines to run simultaneously. This section presents a high level description of the hardware and chapter 5 presents details of the bus arbitration scheme, the processors and the queue-processor. The host or a processor can write primitive calls into the FIFO queue. The information in the FIFO queue is stored in the **Queue Memory**.

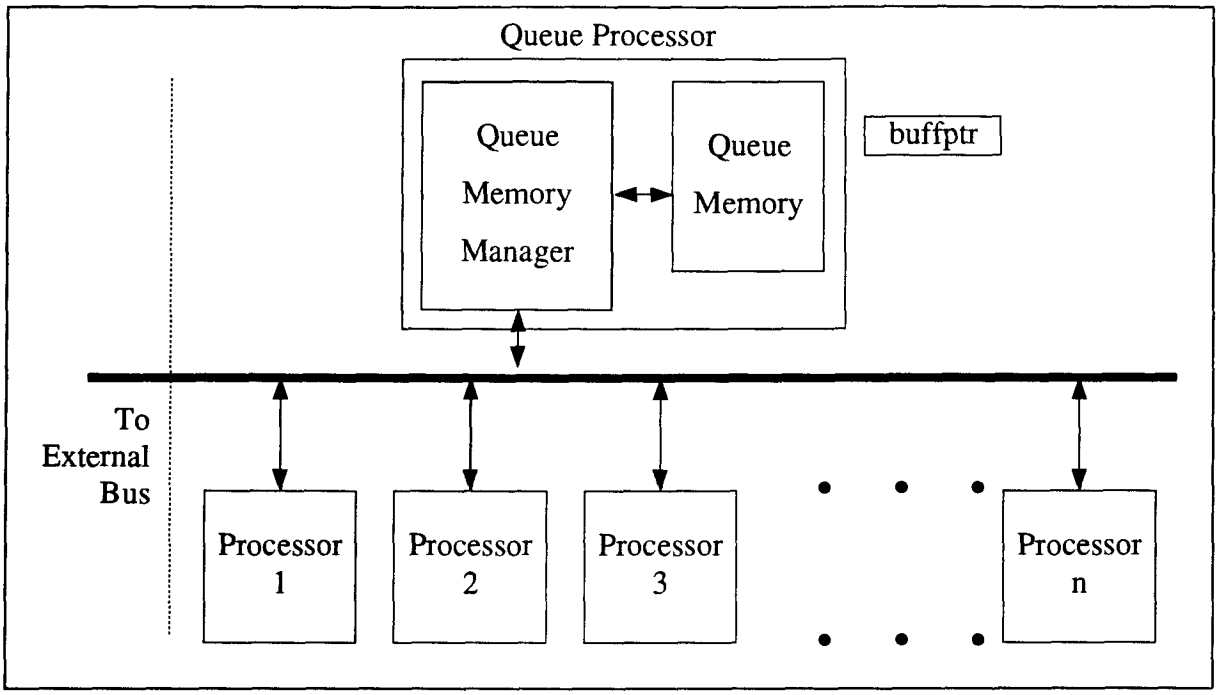

Figure 8 Parallel Hardware Setup

The **Queue Memory Manager** is responsible for accesses made to the **Queue Memory** and for distributing primitive encoding/decoding calls to the processors for servicing. The encoding/decoding algorithms in chapters 3 and 4 need multiple words of information to be passed to the processors. If a processor requires n words of information, then n words can be written into the **Queue Memory** and the **Queue Memory Manager** can distribute these words to the processor. Another option is to write the *address* of the n words into the **Queue Memory** and then the **Queue Memory Manager** can redistribute the *address*. A **Queue-Processor** can be designed with either method but in our implementation the *address* method is chosen. The **Queue Memory Manager** is also responsible for recognizing certain values as commands. Constants that represent these commands can not be used as memory location pointers. Table 5 lists these commands with a brief explanation of each. The complete explanation of these commands occurs in chapters 3 and 4.

Command	Explanation
wait for free	Wait for all the processors to be free before assigning out the queue element.
add control	Increments the control counter.
end signal	Decrements the control counter. If the control counter is zero execute a wait for free then interrupt the host.

Table 5 Queue-Processor Commands.

The processing elements in our system each have a local instruction memory that stores the primitive encoding/decoding routines that are to be executed. When a primitive type is encoded, the resulting TS must be stored in memory, hence the processor needs some mechanism to allocate memory. Memory is also needed when decoding a TS to a machine representation. The buffptr in figure 8 is a register that holds a pointer to the starting memory location of a free memory pool. The host is responsible for setting up the buffptr pointer to point to the available memory. When the free memory pool is set up, it could consist of contiguous or noncontiguous memory blocks. For the initial simulations, the contiguous memory approach explained in section 2.1.1 is taken.

2.1.1 Contiguous Memory Management

Figure 9 shows what the memory space looks like at initialization. The **buffptr** holds the starting address of the free memory space.

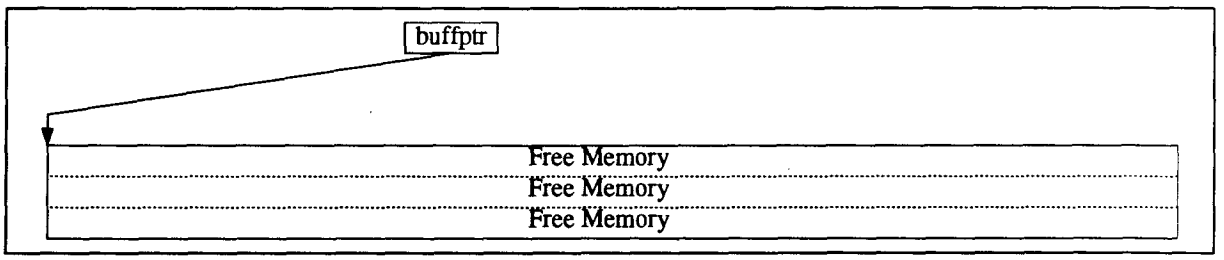

Figure 9 Contiguous Memory: Initial

When a process needs x bytes of memory it preforms the following operations without releasing the shared bus:

- Reads **buffptr**.
- Increments the read value by x.
- Writes the new value to **buffptr**.

The memory, after a memory allocation, looks like figure 10.

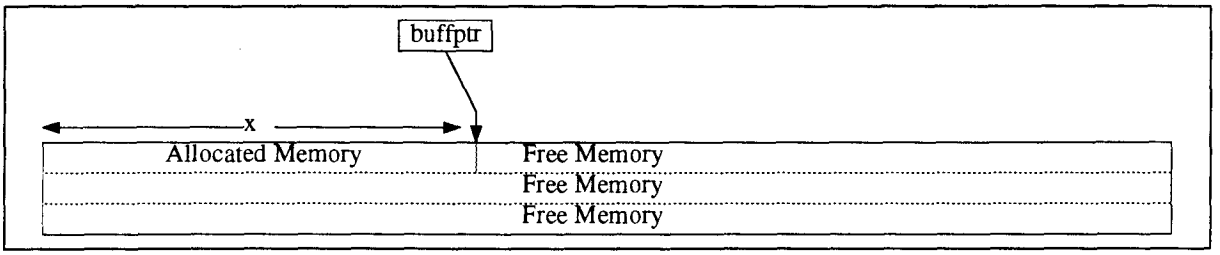

Figure 10 Contiguous Memory: Allocated

2.2 Parallel Algorithm Outline

The following section provides a brief outline of the encoding/decoding algorithms. Chapters 3 and 4 present the full details of the algorithms.

When encoding, the host processor parses its data structure and writes the primitives into the FIFO queue. The queue-processor then distributes the primitives to the free processors. Since the host processor knows the exact data that it wants to encode, it is responsible for assigning all the primitive calls to the queue-processor. Simulations verify that the primitive calls take most of the processing time therefore the time spent by the host on parsing the data structure is minimal.

When decoding, the host processor writes the starting address of the TS to the FIFO queue. The queue-processor then assigns the address to a free processor P1. P1 then reads the tag and length of the first element in the TS and calculates the starting location of the second element in the TS. P1 then writes the starting address of the second element in the TS to the FIFO queue. The queue-processor then assigns the address to the free processor P2. P2 performs the same process that P1 has performed. The process is repeated by other processors until the complete TS is decoded.

2.3 Memory Mapping Scheme

The memory scheme that the host system and the ASN.1 parallel hardware use is depicted in figure 11. The host system segments the memory space into I/O address space and memory address space, which allows I/O registers to be mapped to I/O address spaces.

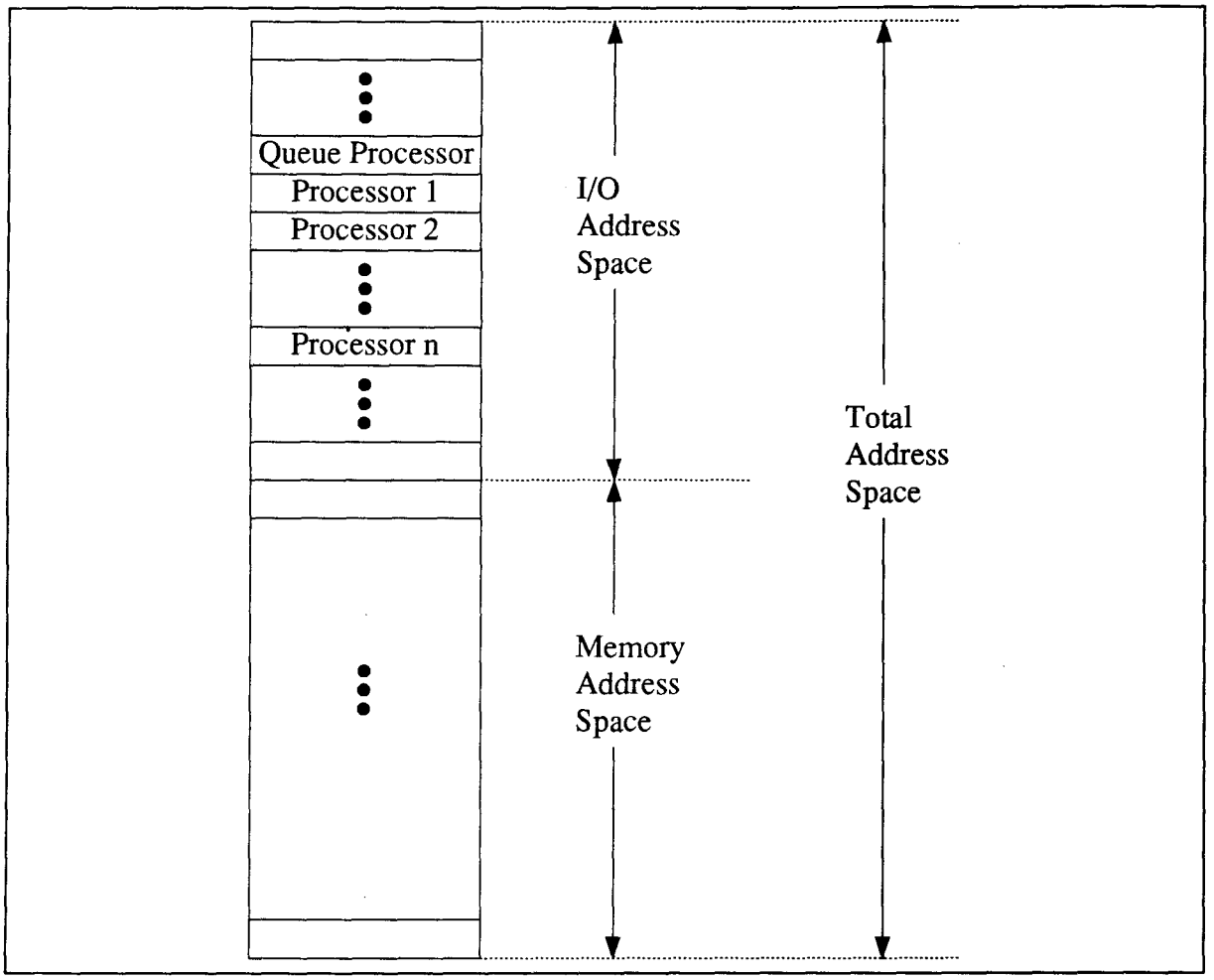

Figure 11 Memory Mapping Scheme

Figure 11 shows that the queue-processor and the processors of the parallel hardware are mapped to some of the I/O addresses. Thus, when the host system wants to assign work to the queue-processor it writes to the I/O mapped location of the queue-processor.

3 Encoding

In the encoding process, information is taken from the application layer, encoded and passed to the session layer. The data structures received from the application layer and provided to the session layer are presented in section 3.1. Once these data structures are defined then the explanation of how an ASN.1 protocol specification is used to generate encoding routines for an application is discussed in section 3.2. Section 3.3 then explains the functionality of the encoding procedures that the applications use. Finally, section 3.4 provides the control flow of the parallel architecture and also concludes the running example of the chapter.

3.1 Encoding Interface Data Structures

The application layer provides a C data structure of the value instance that it needs to encode. The data structure is generated according to the ASN.1 specification and the mapping rules provided in figure 12. A complete list of these mapping rules is found in Ref.[28]. When a value instance is encoded it is presented to the session layer in *IDX* structures. An *IDX* structure is defined in figure 12. Figure 13 shows an ASN.1 specification and figure 14 shows an application's equivalent C type. When the value instance shown in figure 13 is encoded, the resulting output is presented in figure 15. A hardware component can convert the *IDX* structure shown in figure 15 into a contiguous TS. The hardware component can be pipelined with the parallel encoder and will not effect the total throughput of the system.

ASN.1 Type	C Type
INTEGER	int
Character String	OCTS
SEQUENCE {}	struct {}
SET {}	struct {}
SEQUENCE OF {}	LIST
SET OF {}	LIST

```
typedef struct OCTS {
 struct OCTS  *next; /* pointer to next OCTS node */
 int          len; /* # of bytes in OCTS string */
 char         *data; /* pointer to the OCTS string */
} OCTS;

typedef struct LIST_ITEM {
 struct LIST_ITEM  *next; /* next list item */
 char              *item; /* list item */
} LIST_ITEM;

typedef struct LIST {
 LIST_ITEM  *top; /* first item in the list */
 LIST_ITEM  *last; /* last item in the list */
} LIST;

/* encoded output goes into IDX structures */
typedef struct IDX {
 struct IDX  *next; /* next idx */
 int         *len; /* # of bytes in IDX data */
 char        *data; /* information of the IDX */
} IDX;
```

Figure 12 Partial List of Mapping Rules

```
recordType ::= [APPLICATION 1] IMPLICIT SEQUENCE {
 scores   SEQUENCE OF REAL,
 name     [1] IA5String,
 age      [2] INTEGER
}

value instance = {{70.12,80.34,90.56},"John",50}
```

Figure 13 ASN.1 Example

```
#define APPTAG    0x61
#define NAMETAG   0xa1
#define AGETAG    0xa2
typedef struct recordType {
 LIST      scores;
 OCTS      name;
 int       age;
}recordType;
```

Figure 14 C Type Equivalent of Figure 13

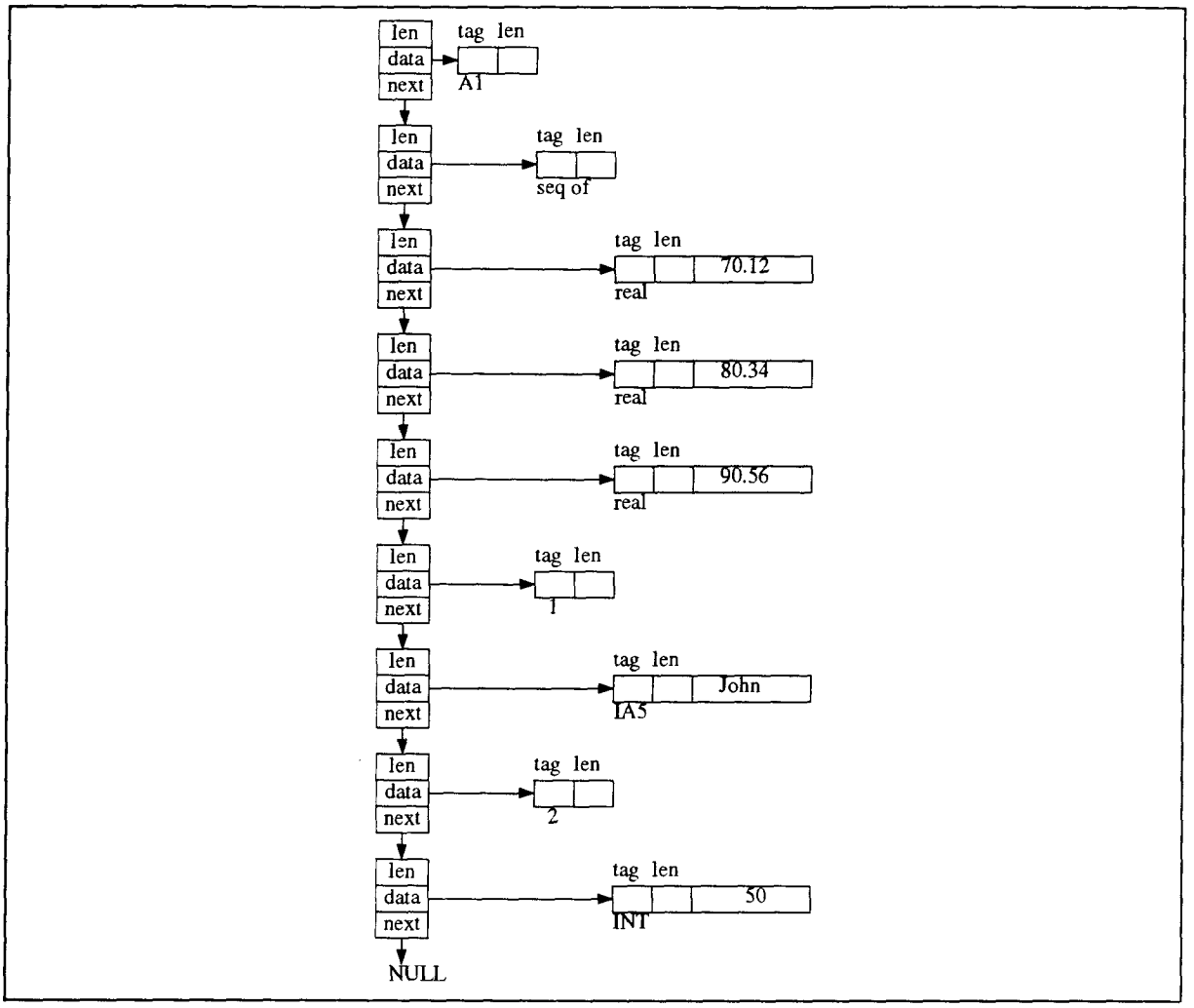

Figure 15 Transfer Syntax of Figure 13 in IDX Structures

3.2 Encoder Generation from ASN.1 Specification

To generate encoded values, like those shown in figure 15, all the values that need to be encoded are written to the parallel ASN.1 encoder/decoder hardware. A program that reads in the ASN.1 specification and generates the encoding routines that the application uses is necessary. CASN.1 can be modified to generate these encoding routines. The encoding routines for the parallel algorithm can be identical to the code generated by the present version of CASN.1. An example of the encoding routines required for the specification of figure 13 is shown in figure 16. The actual function of the *QEncode ()* is to write the encode routine

to the queue-processor. The *EncodeStructBegin ()* and *EncodeStructEnd ()* routines function are optimized for parallelism and the detailed explanation of them is provided in section 3.3.

```
EncodeData (pcode)
 pcodeType *pcode;
{
 EncodeStructBegin (APPLICATION1TAG);

  EncodeSeqOf1 (&pcode->scores, SEQOFTAG)

  EncodeStructBegin (NAMETAG);
   QEncode (OCT, &pcode->name, tag);
  EncodeStructEnd ();

  EncodeStructBegin (AGETAG);
   QEncode (INT, &pcode->age, tag);
  EncodeStructEnd ();

 EncodeStructEnd ();
}



EncodeSeqOf1 (list, tag)
 LIST *list; tagType tag;
{
 LIST_ITEM *item;
 EncodeStructBegin (tag);
  for (item = list->top; item; item = item->next)
   QEncode (REAL, item->item, REALTAG);
 EncodeStructEnd ();
}
```

Figure 16 Generated Encoding Routines for Figure 13

3.3 Encoders Functions and Data Structures

The *EncodeStructBegin ()*, *EncodeStructEnd ()* and the *QEncode ()* procedures are explained in sections 3.3.1—3.3.3. The cost for each of the calls is estimated and compared

to the equivalent call in CASN.1. The determination of the cost of the calls is shown in Appendix B.

The information shown in figure 17 is assigned to the processing element when a primitive call is made. Within the data structure in figure 17 the following are defined:

- *rtag* holds the tag of the value to be encoded and a value that specifies the encoding routine to be used.
- *data* holds a pointer to the data to be encoded.
- *idxptr* holds a pointer to the idx structure that is updated with the encoded value.
- *updateptr* holds a pointer to an integer that is incremented by the total number of TS bytes generated by the primitive encoding routine. The integer is used for calculating the cumulative length in structures.

```
typedef struct eqelemType {
 int      rtag; /* tag & encode routine */
 char     *data; /* data to be encoded */
 idxType  *idxptr; /* encoded data location */
 int      *updateptr; /* cumulative length */
} eqelemType;
```

Figure 17 Encode Queue Element.

3.3.1 EncodeStructBegin ()

When an EncodeStructBegin call is made the length of the structure cannot be encoded because the lengths of the elements in the structure are unknown. There exists a data dependency between the elements of the structure and the length of the structure; hence the length of the structure can only be calculated after all the elements in the structure are encoded. This data dependency must be maintained somehow and must be generally expandable for SEQUENCES, SET, SET OF and SEQUENCE OF. A tree that maintains the data dependency is constructed by using the data structures presented in figure 18.

```
typedef struct nodeType {/* called structure node */
 struct nodeType *parent;
 struct nodeType *next;
 int             lent;
 idxType         *idx; /* idxType same as IDX */
 tagType         *tag;
} nodeType;

typedef struct levelType {/* called level node */
 struct levelType  *plus;
 struct levelType  *minus;
 nodeType          *head;
 nodeType          *tail;
}
```

Figure 18 Encoding Cumulative Length Data Structures

In the remainder of this thesis a node of type *nodeType* is called a structure node and node of *levelType* is called a level node. Figure 19 shows the data dependency structure constructed for the value instance given in figure 13. This data dependency structure is generated according to the following rules.

Initially the tree consists of:

a. one level node known as the root level
b. one structure node at the root level
c. a present level pointer pointing to the root level
d. a deepest level pointer pointing to the root level

When an EncodeStructBegin call is made and the next level exists:

a. a new structure node is added to the tail of the next level
b. the parent field of the new structure node points to the present levels last node
c. the present level pointer now points to the next level
d. the tag of the EncodeStructBegin call is saved in the new structure node

When an EncodeStructBegin call is made and the next level is null:

a. a level node along with an accompanying structure node is added after the present level
b. the parent field of the new structure node points to the present levels last node
c. the present level pointer now points to the next level
d. the deepest level pointer now points to the next level
e. the tag of the EncodeStructBegin call is saved in the new structure node

The maximum cost for calling EncodeStructBegin is 35 while the equivalent routine in CASN.1 costs a minimum of 22. The CASN.1 cost is lenient because it assigns the EncodeTag routine within the EncodeStructBegin routine a zero cost.

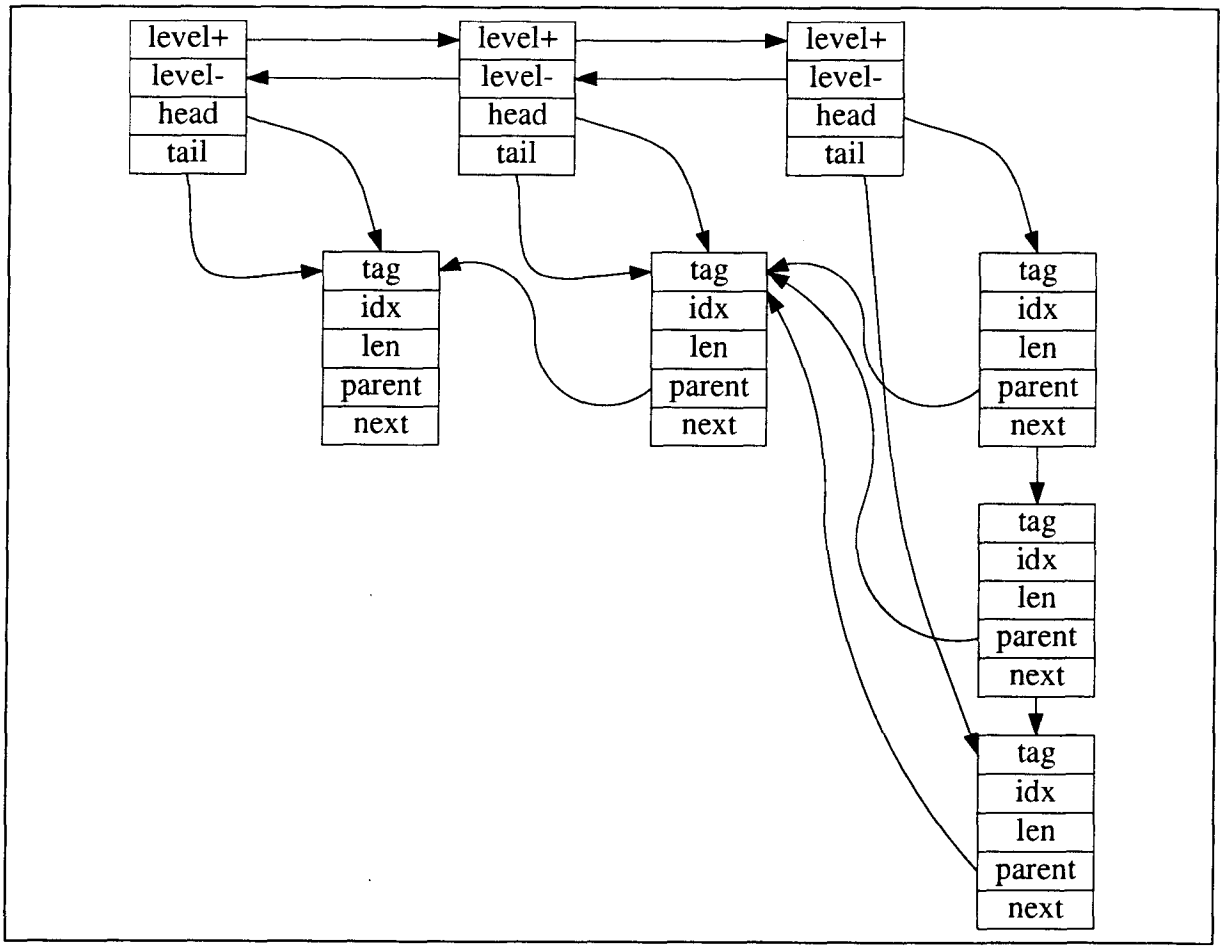

Figure 19 Data Dependency Graph of Figure 13

3.3.2 EncodeStructEnd ()

When an EncodeStructEnd call occurs the elements within a structure have been assigned to the queue-processor. The length of the structure becomes known when all the elements within the structure are encoded. The tag and length of a structure can be encoded at the end of the EncodeStructEnd call or postponed until a later time. The postponed option is used here. The reason for this choice is explained by using the ASN.1 specification shown in figure 20.

```
postponedType ::= SEQUENCE {
  SEQUENCE {
    real1   REAL,
    real2   REAL
  },
  SEQUENCE {
    real3   REAL,
    real4   REAL
  }
}
```

Figure 20 Postponed ASN.1 Type

If there are four processors available to encode a value instance of the ASN.1 specification in figure 20 and the structure lengths are calculated immediately after the EnodeStructEnd call, then only two of the four processors are utilized. Once the values of *real1* and *real2* are assigned to the processors the EncodeStructEnd routine is executed. For the correct length of the sequence to be known one must wait until *real1* and *real2* are encoded and the length of the structure is updated by them. The same thing happens for *real3* and *real4*. Thus at any time only two of the four processors are utilized. If the length calculations of the structures are postponed until the end of all the primitive encodings, then *real1–real4* can be encoded simultaneously on four processors. Once all the reals are encoded then the two

sequences that make up the *postponedType* can encode their tag and length. Finally the tag and length of the *postponedType* is encoded.

EncodeStructEnd moves the level pointer to the previous level. If the previous level is the root level then all the structures, represented by the structure nodes, in the data dependency graph can be assigned to the queue-processor. The structures below the deepest level pointer are assigned to the queue-processor first, next the structures of the previous level are assigned to the queue-processor and this continues until the root level is reached. After all the structures of a level are assigned to the queue-processor, the **wait for free** command is queued. The command tells the queue-processor to wait for all the processors to become free before assigning the next element in the queue. This is necessary because the data in the previous level is dependent on the lengths calculated by the present level.

The cost for calling EncodeStructEnd is 6 for the non-root level and is variable for the root level according to $cost = 25 * nodes + 7 * levels$. In the worst case every EncodeStructBegin call creates a structure node and a level node. The cost for assigning this data dependency graph to the queue-processor is $cost = 32 * EncodeStructEnd$. Hence the maximum cost for the EncodeStructEnd is proportional to the number of times the EncodeStructEnd is called and is $cost = 38 * EncodeStructEnd$. The simplest case of CASN.1's equivalent call costs is $cost = 40 * EncodeStructEnd$.

3.3.3 QEncode ()

The QEncode call is equivalent in function to the primitive encoding routines in CASN.1. This is where the parallel algorithm does significantly better than CASN.1. The total cost of writing this information into the queue is 16 whereas in CASN.1 the cost for a primitive encoding depends on the primitive.

There are three parameters passed to the QEncode (), namely, a value specifying the encoding routine that should be used, the tag for the transfer syntax and the pointer to the data structure to be encoded. This information is written to the queue—processor along with an allocated idx pointer and a cumulative length pointer. The cumulative length is extracted

from the data dependency structure and is the length field of the node that is at the tail of the present level node.

3.4 Encoding Flow Trace

The queue information for the value instance of the ASN.1 specification given in figure 13 is shown in figure 21. A partial parallel trace of the program is shown in figure 22. In figure 21 the first element of the queue is at the bottom of the figure. The first five elements in the queue are for encoding the 3 REALS, the IA5String and the INTEGER. All the non-bold boxed values in the diagram point to queue element structures, one of which is shown to the right in figure 21. The sixth element, **wait for free**, is a signal to the queue-processor not to assign another element until all the processors are free (i.e. until all the previously assigned elements have been encoded). The next elements in the queue are the sequence of element, the nametag element, the agetag element and then the **wait for free** signal. The last encodable element in the queue is the APPLICATION 1 element which is followed by the **end signal** that instructs the queue-processor to interrupt the host when all the processors are free. Sections 3.4.1—3.4.3 explain the actions that the short statements in figure 22 perform.

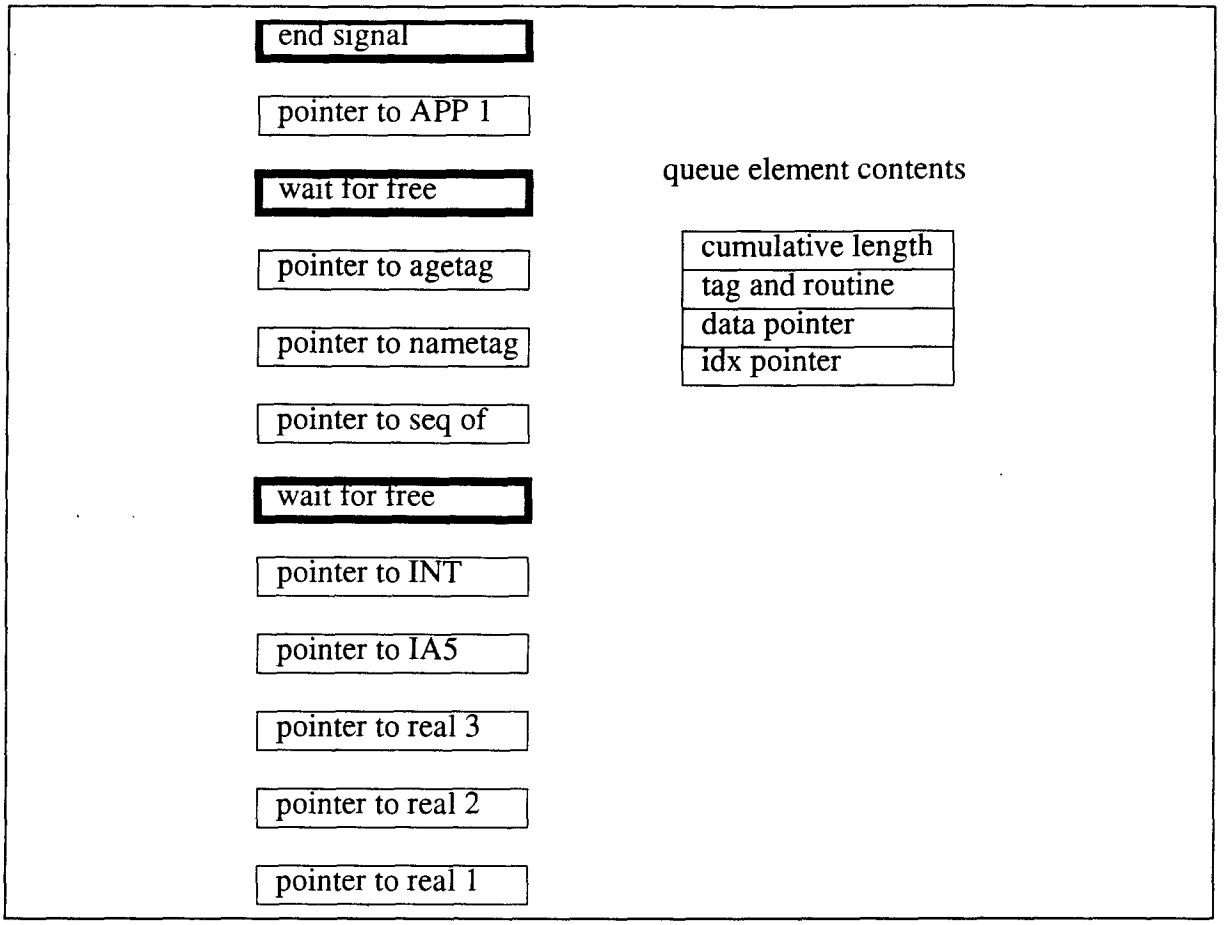

Figure 21 Queue with the Encode Example Calls

3.4.1 Host Responsibilities

When the host wants to encode a TS it has the option of either creating the complete queue information and then assigning the work to the queue-processor or assigning the work to the queue-processor as it parses its data structure. In either case, the host puts all the primitive encoding routines into the FIFO. The host then assigns, to the queue-processor, all the structures from the deepest level of the data dependency graph followed by the **wait for free** signal. It then assigns all the structures from the previous level followed by the **wait for free** signal. This process continues until the root level is reached. At the root level the host writes an **end signal** into the FIFO to signify that the encoding list is complete. When

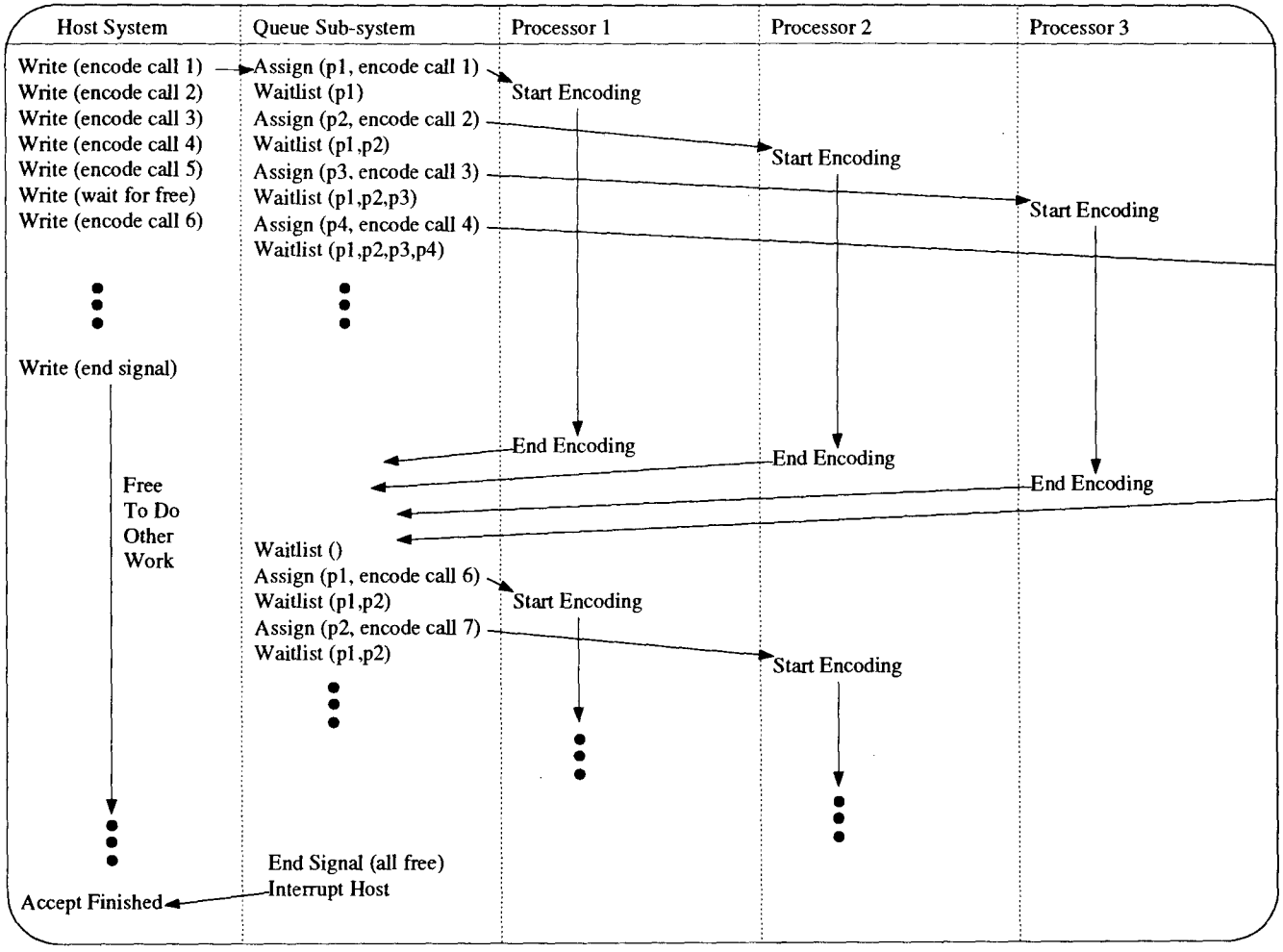

Figure 22 Partial Flow Diagram for Small Encoding Example

the queue-processor encounters this **end signal** and all the processors are free it sends an interrupt to the host to signal the completion of the encoding.

The following describes the actions performed in the flow diagram of figure 22.

- **Write (buffptr)** write the starting address of the memory pool that is available for the encoding routines into **buffptr**. This is the first action that the host performs and is not shown on the diagram.
- **Write (encode call i)** write into the queue-processor any of the possible primitive encode calls.
- **Write (wait for free)** write to the queue-processor a synchronization symbol that signals that all the processors must have a free status before the other elements in the queue are assigned to the processors.
- **Write (End_Signal)** write to the queue-processor a symbol that signals the end of a encoding session.
- **Accept Finished** accept that the encoding is finished. One possible action is the release of the unused buffer space that is pointed to by **buffptr**.

3.4.2 Queue-Processor

During encoding, the queue-processor reads elements inserted into the queue by the host and assigns the work to one of the n processors. Whenever the queue-processor encounters the **wait for free** signal it waits for all the processors to become free. When the queue-processor encounters an **end signal** it stops assigning work. After the processors have finished their encoding the queue-processor signals to the host that the encoding is complete. The following describes the algorithm that the queue-processor performs.

1. Read an element from the queue. If the queue is empty, then block.
2. If the element is a **wait for free** then:

   Waitlist (p1,p2,...,pn) block until all the encoding processors have finished before assigning the next work.

else if the element is a **End_Signal** then.

Waitlist (p1,p2,...,pn) block until all the encoding processors have finished before interrupting the host.

Interrupt Host signal the host of the encoding completion and wait for the acknowledge.

else the element is a **encode call i**

Assign (px, *encode call i*) assign a processor x to encode a specific call. If all the processors are busy, block and wait for one to become free.

3 Start over at step 1.

3.4.3 Processors

A processor blocks until it is assigned a primitive encode call by the queue-subsystem. The processor then encodes the primitive call and upon completion informs the queue-subsystem that it is free. The following explains the actions that the processor executes:

1. **Start Encoding** receives work from the queue processor and starts encoding the primitive.
2. **End Encoding** finishes the encoding and signals the queue-processor of its free availability status.
3. **WaitAssign (encode call i)** waits for the queue-processor to assign a primitive call. WaitAssign is not drawn on the diagram for clarity but would occur in the space before the **Start Encoding** and after the **End Encoding**.

4 Decoding

The decoding chapter follows the same format as the encoding chapter. Section 4.1 discusses the data structure requirements of the application layer and session layer. Section 4.2 explains how the decoding is achieved for ASN.1 specifications. Section 4.3 presents the requirements of the software that is needed to generate the decoder. Finally section 4.4 presents a flow trace of the parallel algorithm.

4.1 Decoding Interface Data Structures

Prior to decoding, the session layer provides a continuous memory block which holds the TS to be decoded. When the decoding is complete the result is made available to the application layer in a data structure created by using the mapping rules given in table 12.

4.2 Decoder Generation from ASN.1 Specification

For an optimal parallel decoding strategy a type tree needs to be generated for the ASN.1 specification. Generation of such a type tree is outlined in [13][16][23]. The type tree is generated off-line and loaded when an instance of a transfer syntax needs to be decoded. To decode a received TS the application needs to write to the parallel ASN.1 decoder the starting address of the TS, the finishing address of the TS, the address of the type tree to be used for decoding this TS instance and the memory location to store the decoded result or a pointer to the decoded result. The data structure used for communicating this information from the host to the parallel decoder is shown in figure 23. The contents and usage of the type tree are discussed in section 4.3.

```
typedef struct qdType {
 nodeType *tt; /* pointer to a node of the type tree */
 char     *tsptr; /* pointer to the start of the TS */
 char     *update; /* mem. location to be updated or used */
 char     *tsend; /* pointer to the end of the TS */
```

Figure 23 Decoding Queue-Element.

4.3 Decoders Functions and Data Structures

For a functional decoder to operate it is necessary to construct a type tree and the traversing routine for this type tree. The rules for generating the type tree are presented in this section and the traversing routines are explained in section 4.4. The type tree is constructed of nodes that have the data structure as shown in figure 24.

```
typedef struct nodeType {
 int             tag;
 int             meminst;
 struct nodeType *option;
 struct nodeType *listptr;
 struct nodeType *next;
} nodeType;
```

Figure 24 Node Data Structure of Type Tree

The *tag* field holds the tag that is to be found in the transfer syntax and a value which indicates the decoding routine to be used. The *meminst* field is discussed in the next paragraph. The *option* field is used for elements in a SET. The *listptr* field is used for SEQUENCE/SET and OF types. The *next* field is used for pointing to the next element expected.

The *memisnt* field is made up of three sub-fields; the *allocation* field, *offset* field and a *bit marker* field. The *allocation* field specifies the amount of memory to allocate at this node. The *offset* field specifies at which offset within a memory location the result of the decode is stored. The *bit marker* field contains an UPDATE bit, an OF bit and a CONSTRUCTOR

bit. The UPDATE bit specifies whether the present node needs to allocate memory for the decoded output or update a memory pointer passed to it. The OF bit is set whenever the present node corresponds to a SEQUENCE OF or SET OF. The CONSTRUCTOR bit should be set whenever the present node corresponds to a SEQUENCE or SET tag.

The rules stated in sections 4.3.1 to 4.3.3 are used when generating nodes of the type tree. Two example type trees are shown in figure 25. Notice that the set type tree in this figure occupies more memory than the sequence type tree. The memory requirements of the set type tree can be reduced by adding a level of indirection; but by doing so, the total number of memory references increase which slows down the overall speed of the system. This essentially becomes a speed/memory trade off.

4.3.1 Node for Primitive Types

The *tag* field holds the tag of the primitive and a value which indicates the decoding routine to be used. The *option* pointer is set only if the primitive is an element of a SET. The *listptr* is always NULL and the *next* node pointer points to the next node expected.

Within the *meminst* field the *allocate* sub-field is zero. The *offset* sub-field holds the offset location used to store the result. The UPDATE, CONSTRUCTOR and OF bit fields are not set.

4.3.2 Node for SEQUENCE and SET Types

The *tag* field holds the tag of the SEQUENCE/SET and a value which indicates the decoding routine to be used. The *option* pointer is non NULL if the SEQUENCE/SET is an element within a SET. The *listptr* pointer points to the first element expected within the SEQUENCE/SET. The *next* pointer points to the first element expected after the SEQUENCE/SET.

Within the *meminst* field the *allocate* sub-field is zero. The *offset* sub-field is zero. The CONSTRUCTOR bit is set. The OF bit is not set. The UPDATE bit is set only on the root node of the tree.

4.3.3 Node for SEQUENCE OF and SET OF Types

The *tag* field holds the tag of the SEQUENCE OF/SET OF and a value which indicates the decoding routine to be used. The *option* pointer is non NULL when the SEQUENCE OF/SET OF is an element of a SET. The *list* pointer points to the first element expected within the SEQUENCE OF/SET OF. The *next* pointer points to the first element expected after the SEQUENCE OF/SET OF.

Within the *meminst* field the *allocate* sub-field is set to the size of the SEQUENCE OF/SET OF structure. The *offset* sub-field holds the offset location of a LIST structure. The CONSTRUCTOR bit is not set. The OF bit is set. The UPDATE bit is set only on the root node of the tree.

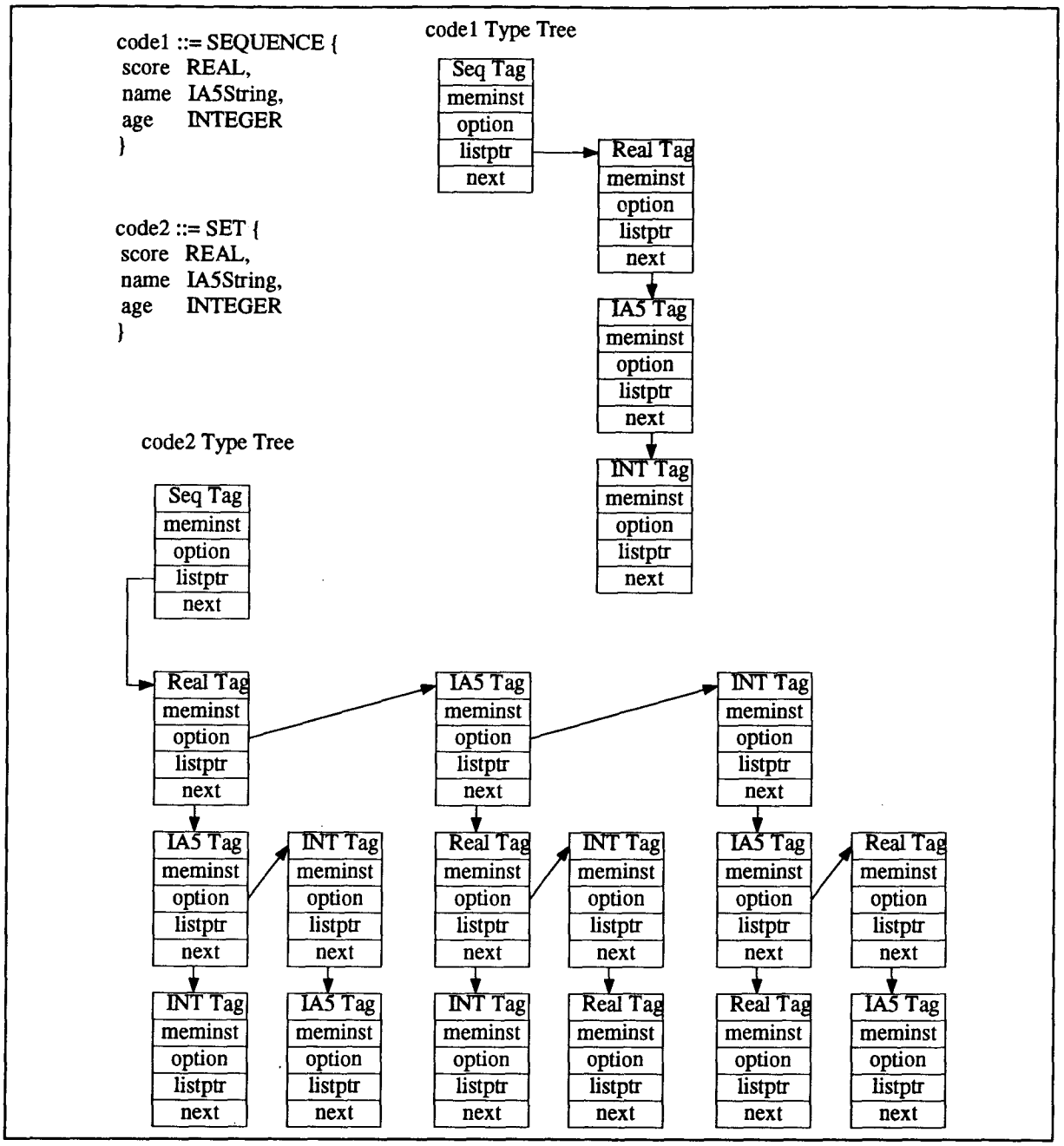

Figure 25 Two Decode Type Trees Comparing SET and SEQUENCE

4.4 Decoding Flow Trace

To describe flow control of the decoding algorithm we use the transfer syntax in figure 26. This transfer syntax is the value instance of the ASN.1 specification given in figure

13 of chapter 3. The decoding type tree generated for this ASN.1 specification is shown in figure 27.

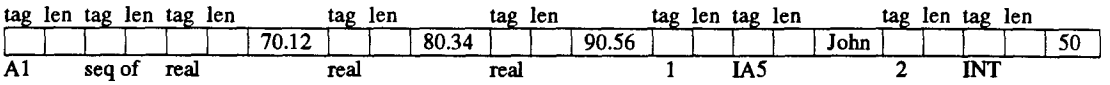

Figure 26 Transfer Syntax of Figure 13

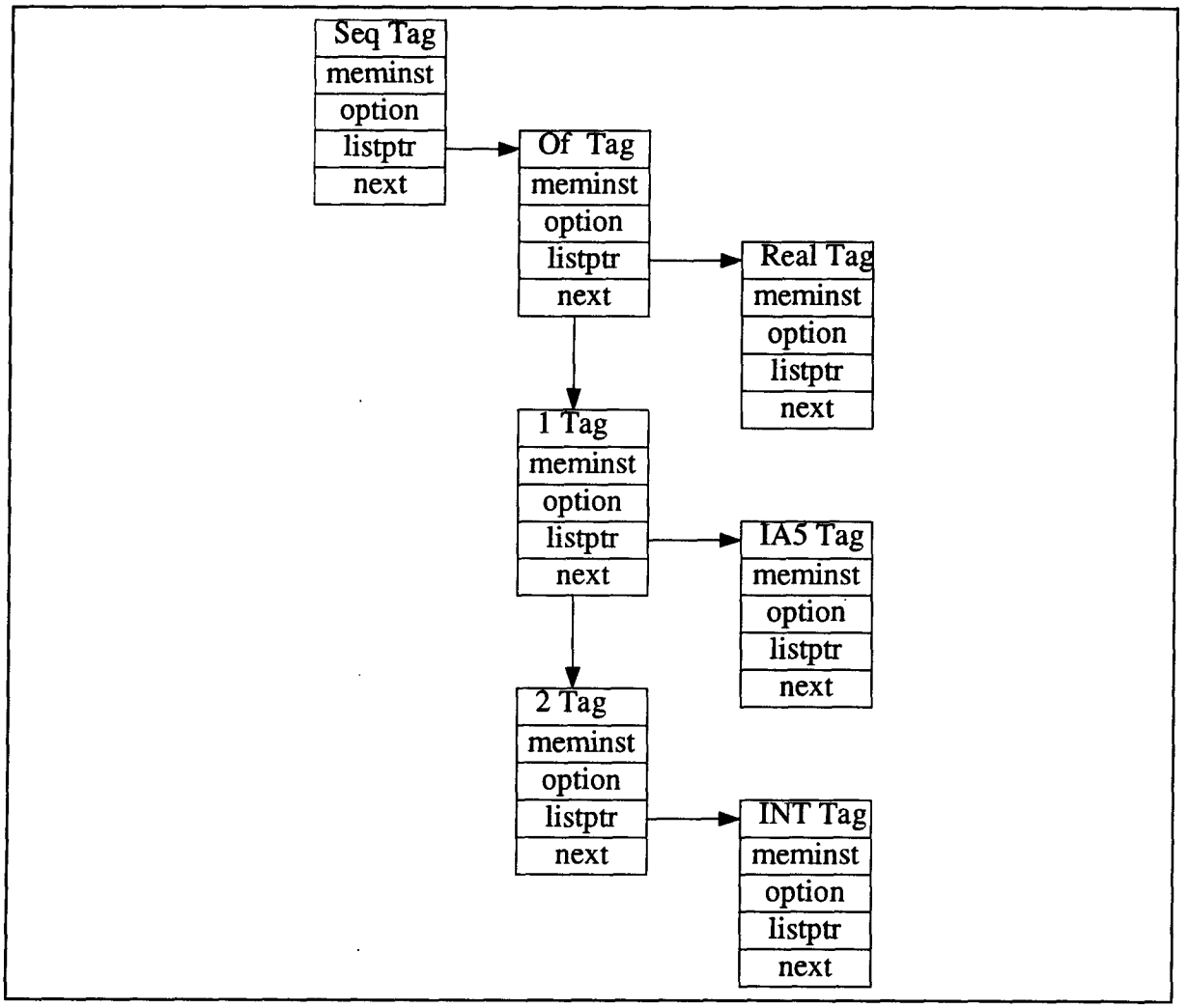

Figure 27 Decode Type Tree of Figure 13

For the decoding to start a decoding queue-element must be written into the queue by the host. The queue—processor then assigns this element to a free processor and continues to do so whenever an element is available in the queue. The queue—processor stops assigning

values to decode when it sees a special end marker, at which time it signals the host that decoding has finished. Figure 29 shows the graphical representation of the control flow that occurs when the TS of figure 26 is decoded. Sections 4.4.1 to 4.4.3 explain the actions of the short statements in figure 29. For clarity purposes the control flow figure does not include all the signals that are transmitted in the system.

4.4.1 Host Responsibilities

When a host wants to decode a TS it sets up the free buffer pointer and then writes a decoding queue-element into the queue-processor. The host then waits for the queue—processor to generate an interrupt upon completion of the decoding.

The following describes the actions that are performed on the flow diagram of figure 29.

- **Setup (buffptr)** set up a pointer to the available free memory that the processor can use.
- **Write (decode call 1)** write the first decode queue element into the queue-processor.
- **Acknowledge Interrupt** acknowledge the interrupt of the queue-processor.

4.4.2 Queue-Processor

The algorithm that the queue-processor performs is shown in figure 28. The queue-processor reads elements inserted into the queue and assigns the non-command element to one of the free processors. The functions of the command elements such as the **wait for free**, **add control** and **end signal** are discussed next. When the queue-processor sees a **wait for free** signal it waits for all the processors to become free before it assigns any other work. When the queue-processor sees an **add control signal** it increments the control counter that is initially zero. When the queue-processor sees an **end signal** it decrements the control counter. If the control counter is zero when the queue-processor sees an **end signal** it interrupts the host and waits for an acknowledgment. Note that the functions that the queue-processor performs for encoding are a subset of the functions described here; hence, the same queue-processor is used for both encoding and decoding.

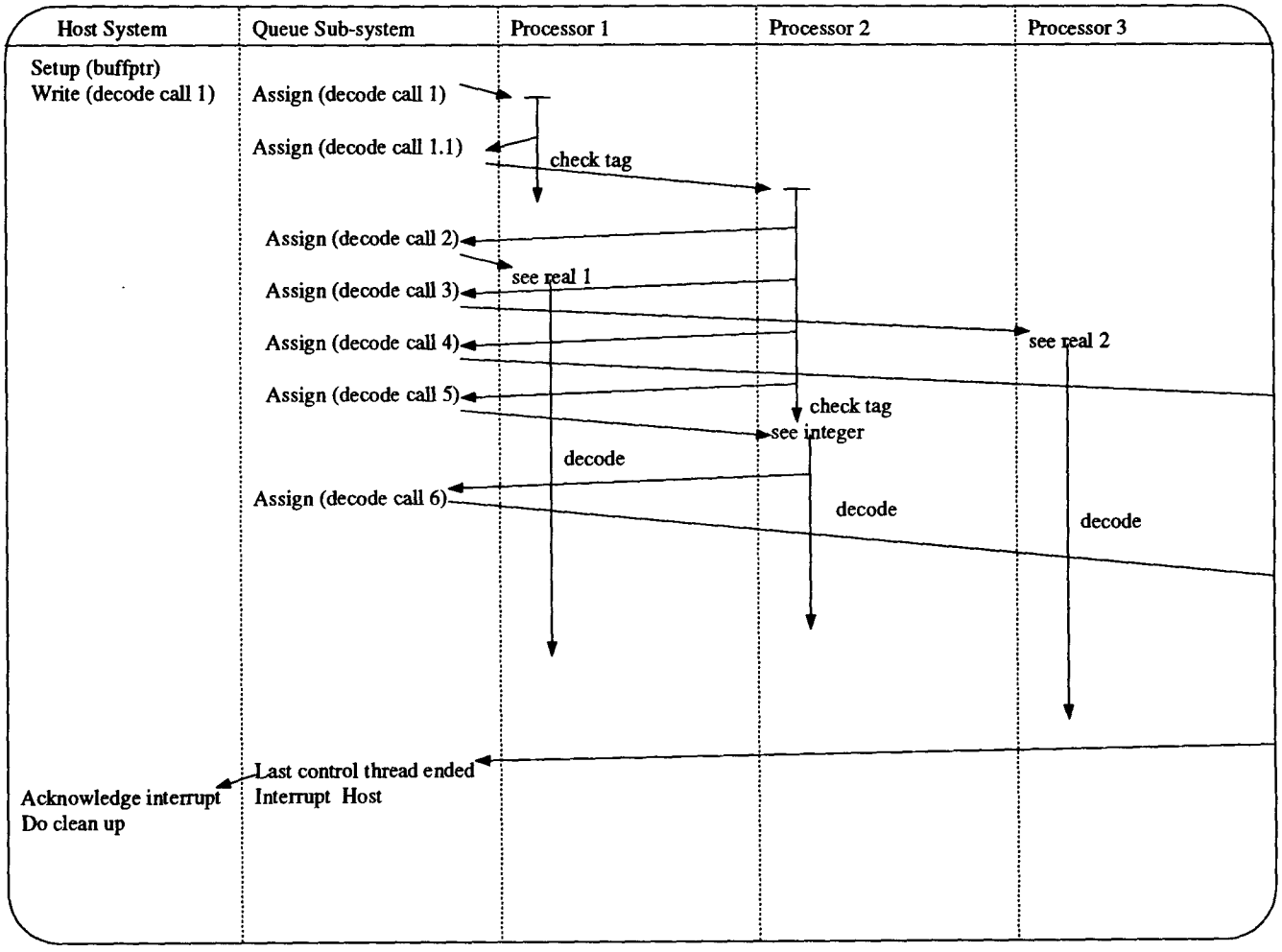

Figure 29 Partial Flow Diagram For Decoding Example

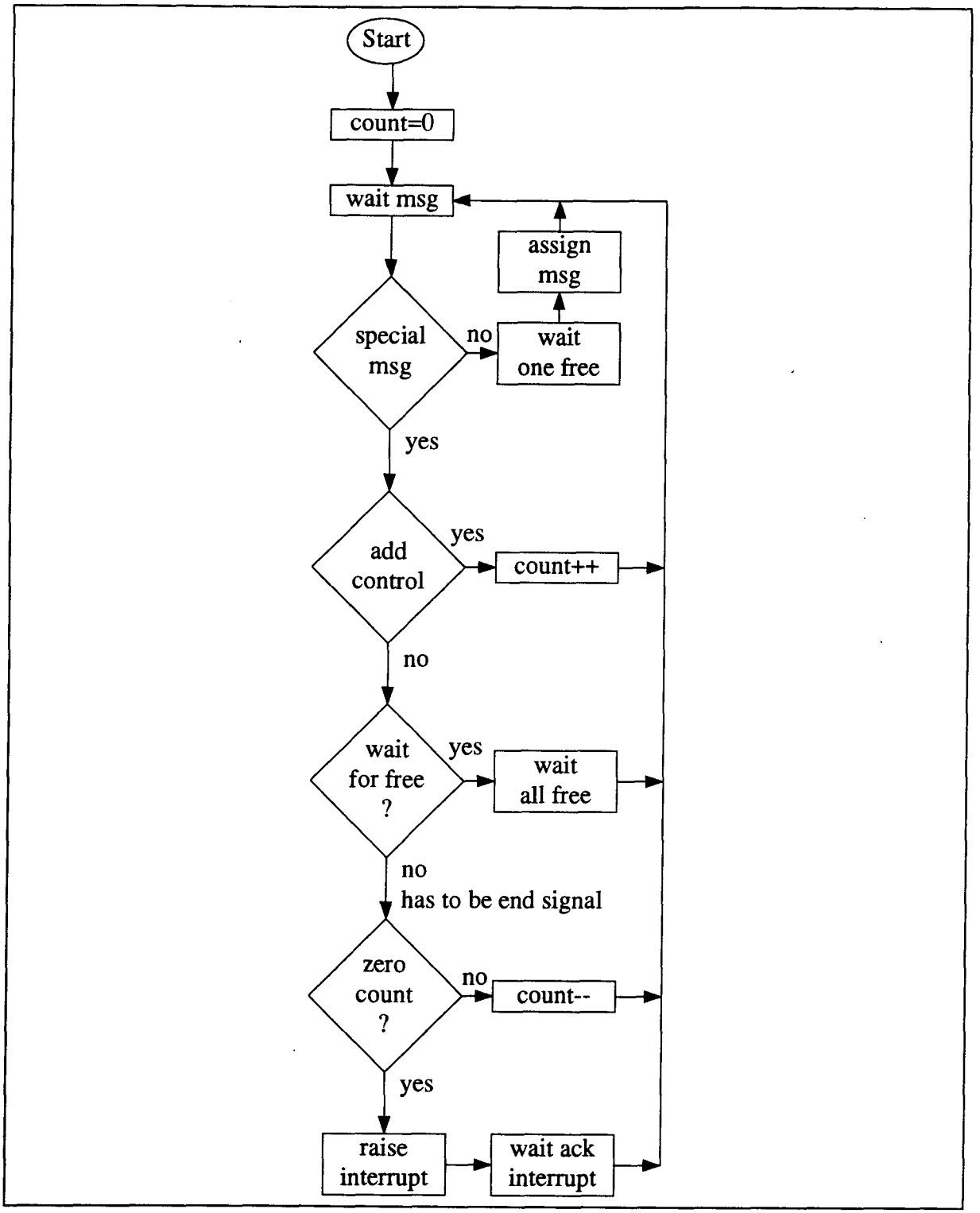

Figure 28 Algorithm of Queue-Processor

4.4.3 Processors

The decoding algorithm used by the processors is depicted in figure 30. A processor signals the queue-processor that it is free and then it waits for an address of a decode queue element to be written to its I/O mapped location. Henceforth whenever a processor assigns work this means it constructs a queue element and then writes a pointer to this information into the queue. Also the queue element pointer that the processor receives is referred to as `rec` and the pointer that the processor assigns to the queue processor is referred to as `q`. The rest of this section explains all the actions shown in figure 30.

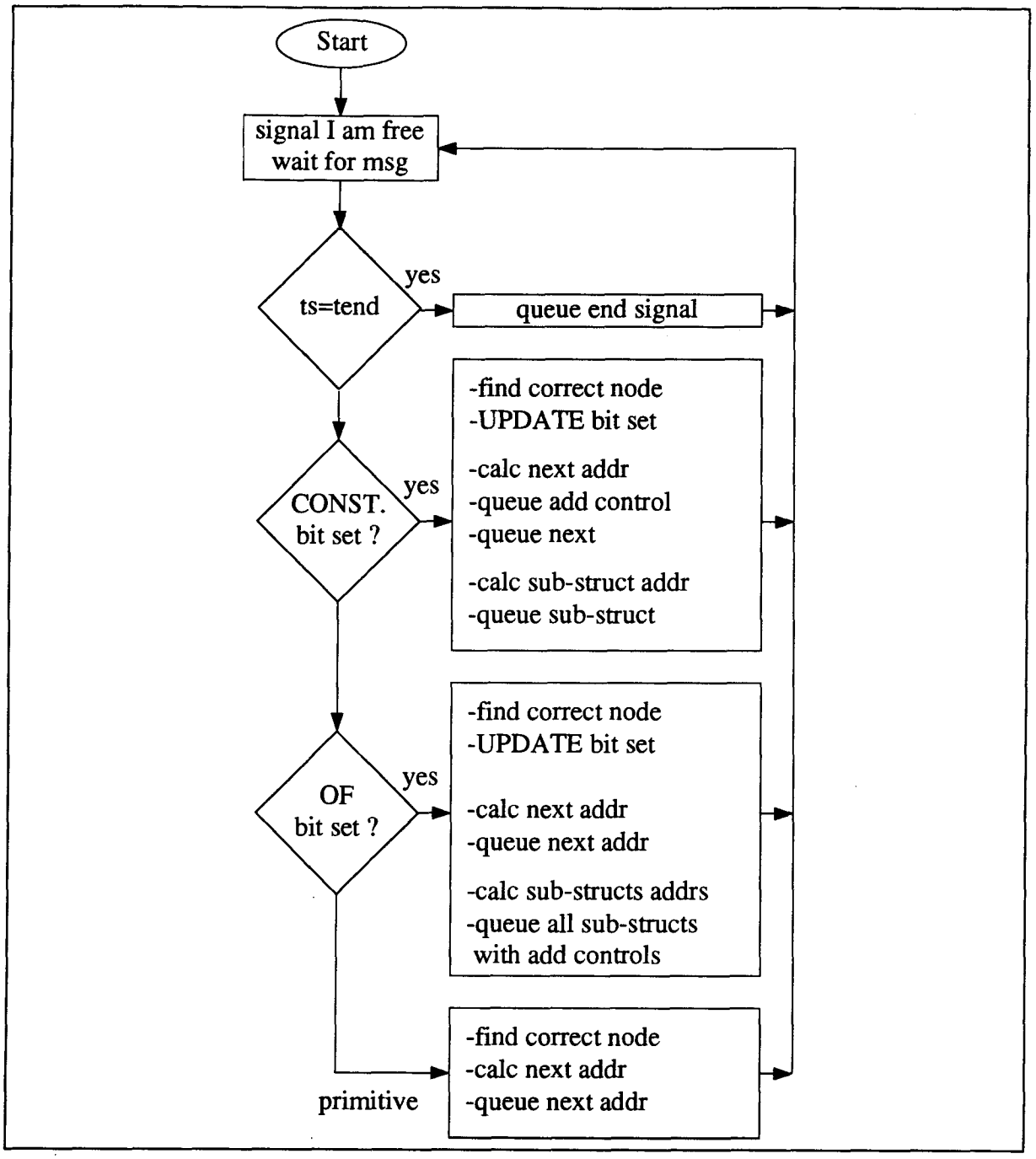

Figure 30 Decoding Algorithm for Processors

tsptr = tsend If tsptr = tsend then the processor is at the end of the decoding segment and has to signal the queue-processor that one control trace of the algorithm has ended.

```
if (rec->tsptr == rec->tsend)
 write end signal to queue-processor;
```

UPDATE bit set The UPDATE bit is set only for the first node in the type tree. This signals that the amount of memory specified in the *allocation* sub-field of *meminst* (figure 24) should be allocated and that the queue element *update* pointer should be updated with the address of the allocated memory. The following C like pseudo code segment summarizes these actions:

```
q->tt = rec->tt->next;
q->tsptr = rec->tsptr +
           length in length field of (rec->tsptr);
q->update = allocate memory;
q->tsend = rec->tsend;
*q->update = allocate memory;
```

find correct node When the processor is not at the end of a control trace it checks to see it is at the correct type tree node. If the nodes *option* pointer is not NULL, it then decodes the TS tag and determines if the tag at the present node matches this TS tag. If the TS tag and tag at the node are not the same it follows the *option* pointer to the optional node and then compares the TS tag to the tag at that node. The processor continues following the *option* pointer until it finds a matching tag. If the processor does not find a matching tag a signal is generated signifying that there is an error in the incoming transfer syntax.

Note that the elements within a SEQUENCE always have a NULL *option* pointer but for elements within a SET the *option* pointer can be non NULL. This means that the tag of an element in a SEQUENCE need not be decoded before the next element is assigned to the queue-processor; but for elements within a SET, the tag has to be decoded and matched before the next element is assigned to the queue-processor. This will make SEQUENCE decoding faster than SET decoding. The following C like pseudo code segment summarizes these actions:

```
tt = rec->tt;
while (tt != NULL) {
 if (tt->tag == tag of (rec->tsptr) break;
 if (tt->option == NULL) error; else tt = tt->option;
}
if (tt->option == NULL) error;
```

CONSTRUCTOR bit set If the CONSTRUCTOR bit at the present node of type tree is set then the *find correct node* and *UPDATE bit set* operations are performed. The CONSTRUCTOR bit set means that we are at the start of a SEQUENCE/SET. At this point the processor calculates the end of the SEQUENCE/SET which is the starting location of the next element expected. It then informs the queue that one more control thread is being added and assigns, to the queue-processor, the element expected after this sequence. The processor then calculates the starting location of the first element of this sequence and assigns this element to the queue-processor. Finally if the *option* pointer is NULL the processor checks to see if the TS tag matches the present nodes tag. The following C like pseudo code segment summarizes these actions. Note that tt value used below is defined in *find correct node*.

```
q1->tt = tt->next;
q1->tsptr = rec->tsptr +
            size of tag and len of (rec->tsptr) +
            length in length field of (rec->tsptr);
q1->update = rec->update;
q1->tsend = rec->tsend;
write q1 to queue-processor;
write add control to queue-processor;
q2->tt = tt->listptr;
q2->tsptr = rec->tsptr +
            size of tag and len of (rec->tsptr);
q2->update = rec->update;
q2->tsend = rec->tsptr +
            size of tag and len of (rec->tsptr) +
            length in length field of (rec->tsptr);
write q2 to queue-processor;
match tt->tag and tag of rec->tsptr;
```

OF bit set If the OF bit of the present node of the type tree is set then the *find correct node* and *UPDATE bit set* operations are performed. The OF bit set means that we are at the start of a SEQUENCE OF/SET OF. At this point the processor calculates the end of the SEQUENCE OF/SET OF which is the starting location of the next element. I then assigns this element to the queue-processor. Next the processor scans through the elements of the SEQUENCE OF/SET OF performing only length calculations, to arrive at the next SEQUENCE OF/SET OF element. At each SEQUENCE OF/SET OF element the processor assigns the element to the queue-processor and informs the queue-processor to add one tread. Finally, if the *option* pointer for this OF structure is NULL the processor checks to see if the TS tag matches the tag of the present node. The following C like pseudo code segment summarizes these actions. Note that tt value used below is defined in *find correct node.*

```
q->tt = tt->next;
q->tsptr = rec->tsptr +
           size of tag and len of (rec->tsptr)  +
           length in length field of (rec->tsptr);
q->update = rec->update;
q->tsend = rec->tsptr +
           size of tag and len of (rec->tsptr);
write q to queue-processor;
endptr =  rec->tsptr +
          size of tag and len of (rec->tsptr) +
          length in length field of (rec->tsptr);
q->tt = tt->listptr;
q->tsptr = rec->tsptr +
           size of tag and len of (rec->tsptr);
while (endptr != q->tsptr) {
 q->tsend = q->tsptr + length in length field of (q->tsptr);
 q->update = allocate memory give in sub-file of tt->meminst;
 write q to queue-processor;
 q->tsptr = q->tsend;
}
match tt->tag and tag of rec->tsptr;
```

primitive If neither the OF bit nor the CONSTRUCTOR bit is set then this is a primitive decode element and the first action is to assign out the next element to be decoded. The

primitive element is then decoded and stored at the update pointer location adjusted with the *offset* quantity in *meminst.* The following C like pseudo code segment summarizes these actions. Note that tt value used below gets defined in *find correct node.*

```
q->tt = tt->next;
q->tsptr = rec->tsptr +
           size of tag and len of (rec->tsptr) +
           length in length field of (rec->tsptr);
q->update = rec->update;
q->tsend = rec->tsend;
write q to queue-processor;
decode the primitive;
```

5 Hardware Models

The algorithms suggested in the previous sections were simulated on the parallel hardware presented in figure 8. The elements in this hardware are specified at the behavior level using VHDL[1][9]. Sections 5.1—5.3 describe the operations of each of the components.

5.1 Bus Arbitration Method

A rotating daisy chain (RDC) is chosen for the bus arbitration method. A daisy chain and a rotating daisy chain bus arbitration setup are shown in figure 31. A brief outline of both the arbitration schemes is provided in the next paragraph. A RDC is chosen for the bus arbitration method because this is a distributed method that is scales easily. The RDC method also offers the best bus arbitration scheme for the amount of hardware needed for implementation[2]. Ref.[21] presents various implementation methods for the RDC along with their relative advantages.

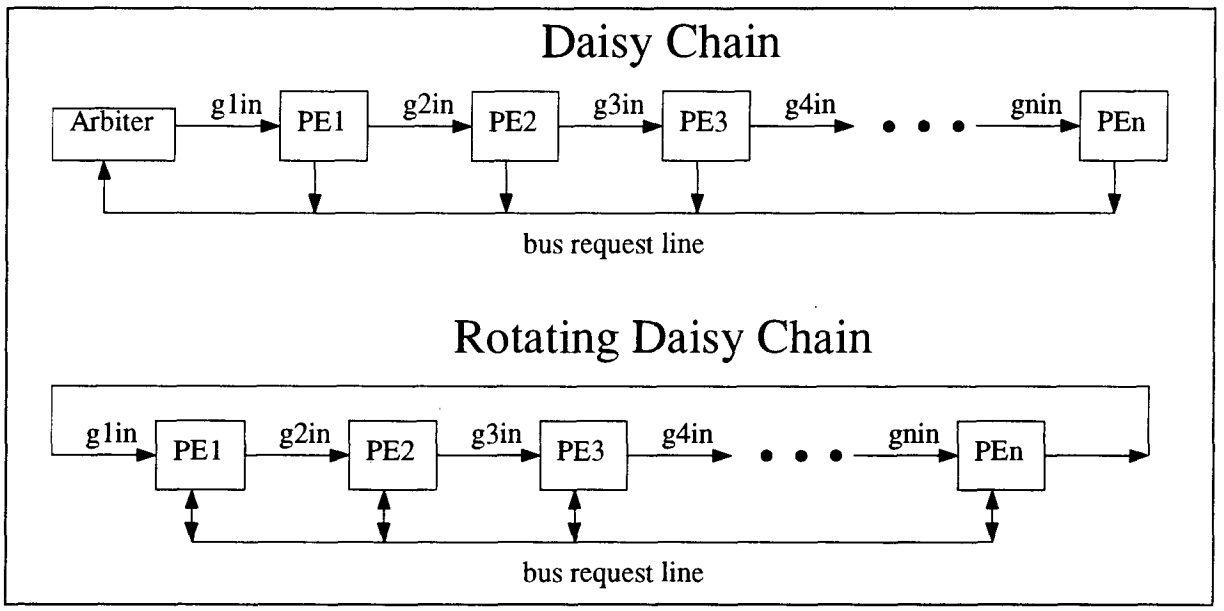

Figure 31 Daisy Chain and Rotating Daisy Chain

In the daisy chain bus arbitration scheme shown in figure 31 any of the n processing elements (PE) can request the bus by raising the **bus request line**. The arbiter will then

raise the **g1in** grant signal. The **g1in** signal allows **PE1** access to the bus; but if **PE1** does not require the bus, it will propagate the **g1in** signal to **g2in**. The grant signal can propagate through to any of the n PEs that request the bus. This bus arbitration scheme is unfair as **PE1** has the highest priority because whenever it requests the bus it will always get the **g1in** grant signal. In this system, **PEn** has the lowest priority because when **PEn** requests the bus it will get the grant if none of the previous (n-1) PEs require the bus. At a processor PEi the processors to the right of PEi have lower priorities and processors to the left of PEi have higher priorities.

The RDC initially starts with one PE responsible for generating a bus grant signal. This PE has the lowest priority while the first PE to the right has highest priority. The PE that generates the grant signal is the arbiter. When a PE requests the bus and receives the bus grant signal this PE then assumes lowest priority and becomes responsible for generating the bus grant signal for the next arbitration. This moving of the bus granting capabilities from one PE to another maintains an even priority system[2]. Another advantage of using the RDC method is the relative ease with which other systems can connect into the RDC. When another system wants to share the bus with the elements on the bus, it simply includes itself in the RDC. The overhead associated with including a arbiter in every PE is low. Figure 32 shows that the hardware requirements are minimal.

Figure 32 shows a three PE RDC implementation. Each processing element has a one bit bus grant register and a one bit bus request register. The operation of this three PE RDC is explained by using the timing diagram of figure 33 on page 54. The signals **/B_RQ<1–3>** are bus request signals generated by the PEs. The **/req<1–3>** are the clocked **/B_RQ<1–3>** signals and **/t<1–3>** are the grant signals that the PEs use. When the system initializes, only one bus grant register is set and this is signal **/t1** in the timing diagram. Now, **/t1** is responsible for generating the bus grant upon initialization. When one PE receives a bus grant from another, this sets its bus grant register and now it is responsible for generating the next bus grant signal. The next action in the timing diagram occurs when PE 3 requests

for the bus by raising the **/B_RQ3** line which in turn gets clocked and raises **/req3**. Finally **/t3** rises signaling that PE 3 now owns the bus. PE 3 is now responsible for generating the next bus grant and so PE 1 is released of this duty. The timing diagram of figure 33 contains some interesting scenario such as two PEs requesting the bus at the same time. The reader is free to explore the timing diagram and no further explanation is provided.

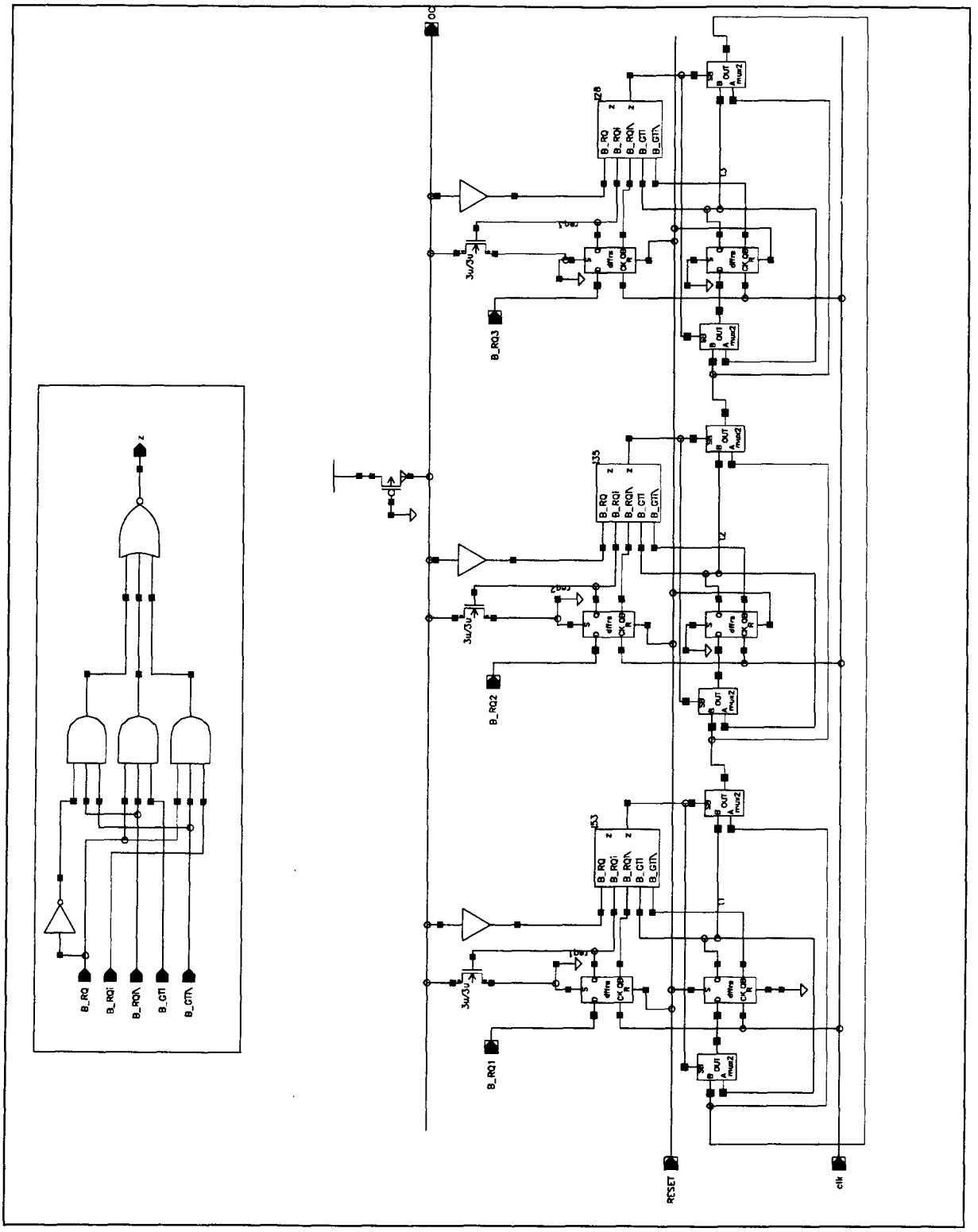

Figure 32 Partial Diagram of RDC Arbitration

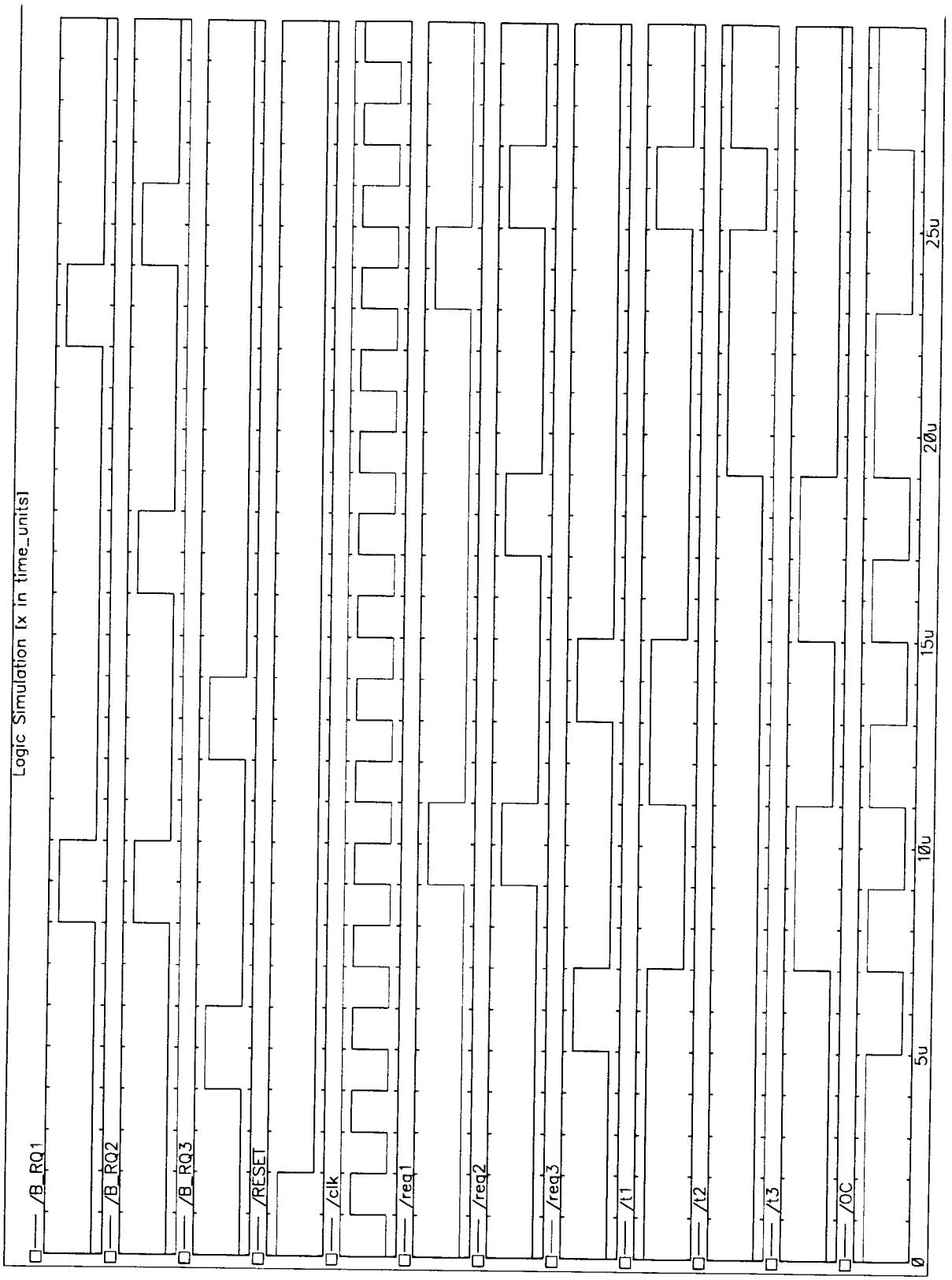

Figure 33 RDC Timing Diagram

5.2 Processor

The proposed RISC processor shown in figure 34 is used for the processing elements (PE) in the parallel architecture. The architecture is similar to the Motorola MC88100[10] and the Sun SPARC[18] architecture but the instruction set is optimized for encoding/decoding. The processing element is a dual bus 32–bit processor with a 32–bit fixed length instruction set. A 32–bit processor is chosen because memory management and mapping is compatible with contemporary 32–bit address space systems. A 32–bit fixed length instruction format is chosen to make the instruction decoder simpler and to allow an instruction to be fetched per memory read. This processor can exploit super-scalar parallelism because the integer unit, the data unit and the instruction unit are designed to operate simultaneously.

The core processor in figure 34 has a smaller instruction set than the Motorola MC88100 or the Sun SPARC and it has some special instructions that are not available on either. These special instructions are used to implement atomic memory read/write cycles and to access local memory. These special instructions make this processor more suited for running the parallel algorithm described above than either the Motorola MC88100 or the Sun SPARC.

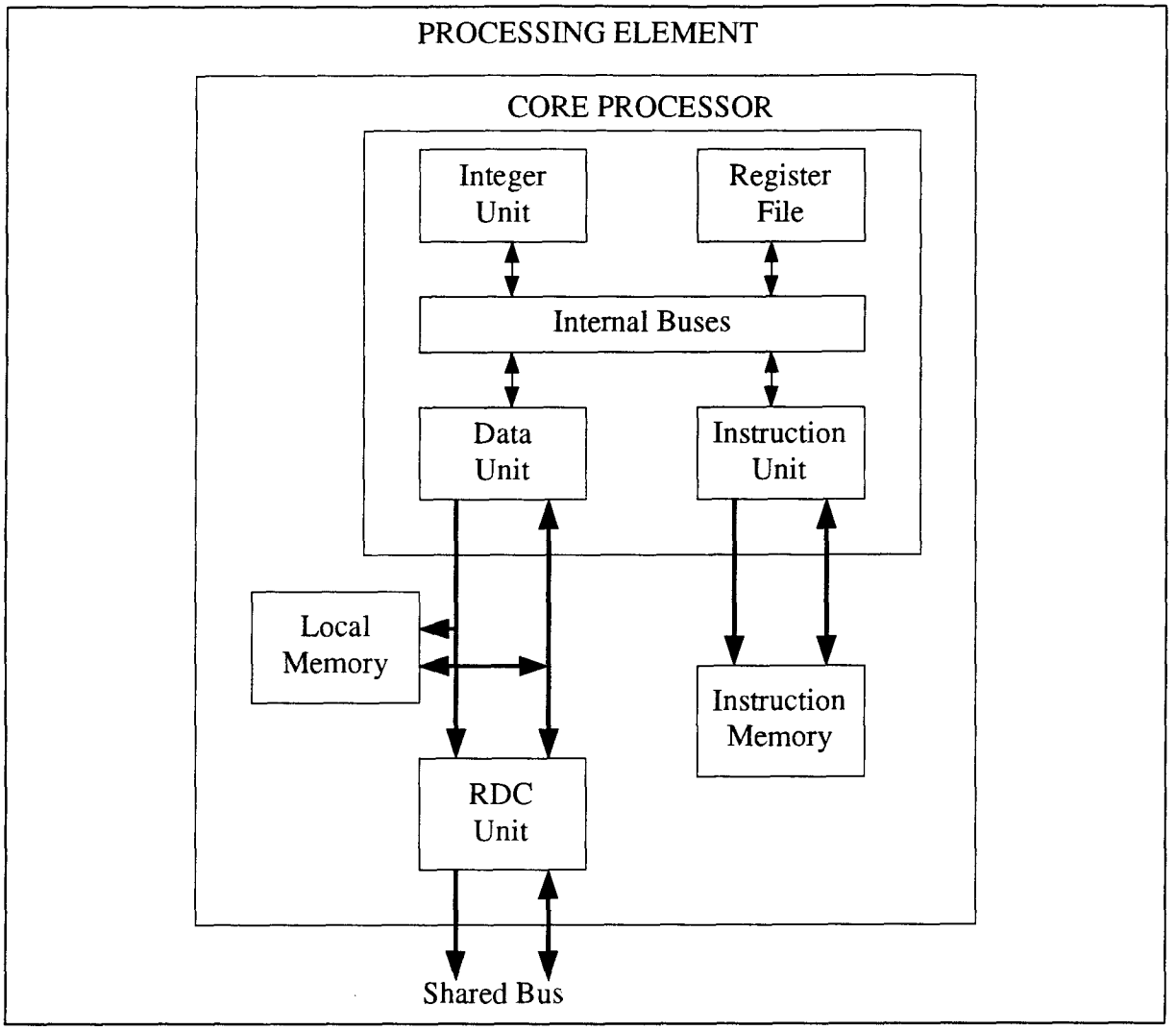

Figure 34 Processor Block Diagram

Sections 5.2.1 to 5.2.4 describe the main units in the block diagram and discuss the instruction set of the PE which is presented in table 7. Table 6 shows all input/output signals of the PE and provides a brief explanation of each.

5.2.1 Integer Unit

The integer unit executes the first 7 instructions in table 7. The unit executes all the arithmetic and logical operations in one clock cycle. The integer unit works only in a register to register mode. There are no complex instructions for modifying memory locations.

5.2.2 Data Unit

Loading from and storing to memory is performed by this unit. Depending on the load/store operation that is being executed, the data unit sets an internal line that activates the local memory or the RDC element of section 5.1. A typical memory read and write timing diagram is given in Appendix D.

5.2.3 Instruction Unit

The instruction unit does memory reads which follow the read timing diagram depicted in Appendix D. The instruction unit implements a two stage pipeline with delayed branching. This allows an instruction to be executed in the slot immediately after a branch operation so that the cost associated with branching can be reduced by compiler writers and assembly language programmers.

5.2.4 Register File

The register file consists of a 32 32–bit registers. Not all of these registers are available for general programming use. The following text explains the functions of the special registers.

Reg(0) = Zero Register Use of register 0 as a zero register is a hardware convention. Register zero is wired to zero by the hardware. Programs are allowed to write into register zero, but this has no effect on its contents.

Reg(26) = Frame Pointer Use of register 26 for the frame pointer is a software convention. The frame pointer points to the base address of the stack frame of each procedure.

Reg(27) = Stack Pointer Use of register 27 for the stack pointer is a software convention. The stack pointer points to the present location in the stack.

Reg(28) = Return Address Use of register 28 for the return address is a hardware convention. Whenever a branch to a subroutine call is made the return address is stored into register 28.

Reg(29) = Queue Address Use of register 29 for the queue address is a software convention. The queue address is the I/O mapped address of the queue-processor.

Reg(30) = Processor Address Use of register 30 for the processor address is a software convention. The I/O mapped address of the processor is stored in this location. When a processor writes to its own I/O mapped location this means that it is free for work assignment.

Reg(31) = Wait Result Use of register 31 for the **wait** instruction's result is a hardware convention. When the processor executes the **wait** instruction, the information that is written to the processors I/O mapped location is stored in register 31.

Signal	Functional Description
A<31..0>	Address lines.
D<31..0>	Data lines.
READ	Signals memory read.
WRITE	Signals memory write.
READY	Signals a completion of a memory access.
LOCAL	Signals a write to the local memory.
ATOMIC	Signals that a atomic read, increment, write is taking place.
BUSRQ	Bus request line.
BUSGTIN	Bus grant in line.
BUSGTOUT	Bus grant out line.
RESET	The asynchronous reset line.
CLK1	Phase one of the clocking signal.
CLK2	Phase two of the clocking signal.

Table 6 Processing Elements Signals

Instruction	Description Of Arithmethic And Logic Instructions
shift left	Shifts a register left.
shift right	Shifts a register right.
add	Adds two registers or a register and an immediate.
subtract	Subtracts between two registers or a register and an immediate.
XOR	Bitwise XOR of two registers.
AND	Bitwise AND of two registers.
OR	Bitwise OR of two registers.

Instruction	Descriptions Of Control Flow Instructions
briq	Branch to an address specified by a register and an immediate.
brsr	Same as briq but the return address is saved.
wait	Blocks until a memory write is executed on the memory mapped location of the processor.

Instruction	Description Data Load / Store Instructions
local load	Loads to a register from the local memory.
local store	Stores to the local memory the contents of a register.
global load	Loads to a register from a global memory.
global store	Stores to the global memory the contents of a register.
atomic increment	Loads a register from memory, increments it and stores the result back into memory without releasing the shared bus.
load high	Loads to a register's high bits the immediate constant.

Table 7 Instructions Executable On a Processing Element.

5.3 Queue-Processor

The queue-processor maintains the availability status of the processors in the parallel system and distributes the work load accordingly. Figure 35 shows the block diagram of the queue-processor. The Queue Memory Manager is the controller of the queue-processor and implements the algorithm presented in figure 28. The block structure of the Queue

Information block is shown in figure 36 and the block structure of the Processor State Information block is shown in figure 37.

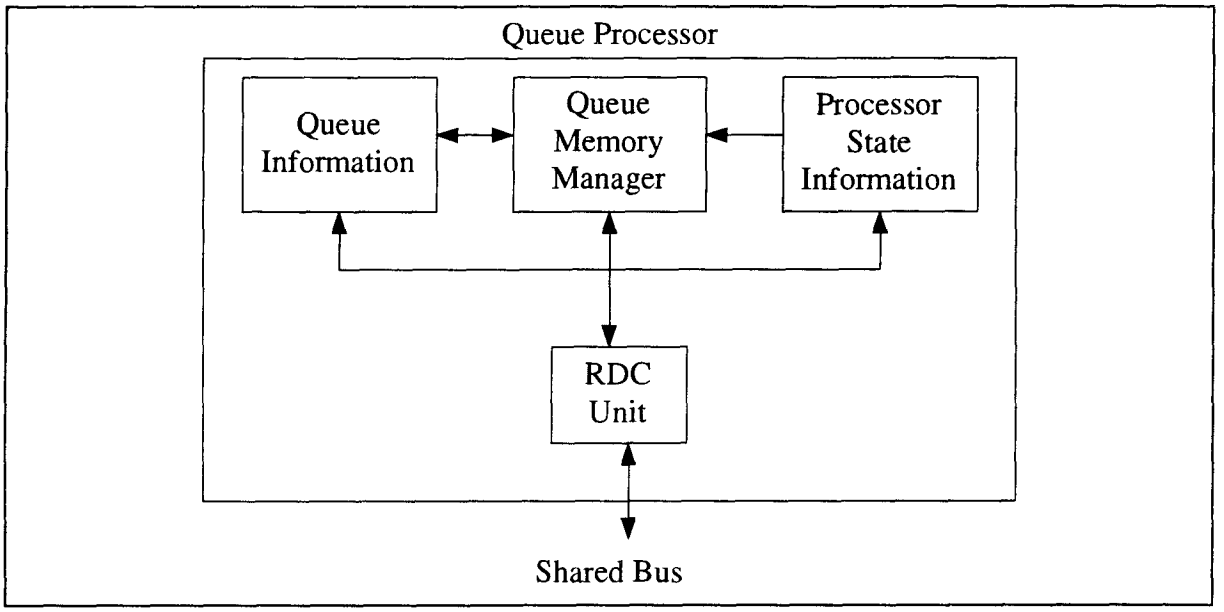

Figure 35 queue-processor

Whenever a word of information is written to the I/O mapped address of the queue-processor this information is actually written into the queue memory location pointed to by the first queue element pointer (figure 36). The first queue element pointer is then incremented so that it is ready for the next write to the I/O mapped location of the queue-processor. When a word of information is assigned to the queue, the information is read from the location that the last queue element pointer is pointing to. Again the last queue element pointer gets incremented after this operation is performed. The first and last queue element pointers are implemented using a counter that gets incremented whenever information is written to the I/O mapped location of the queue.

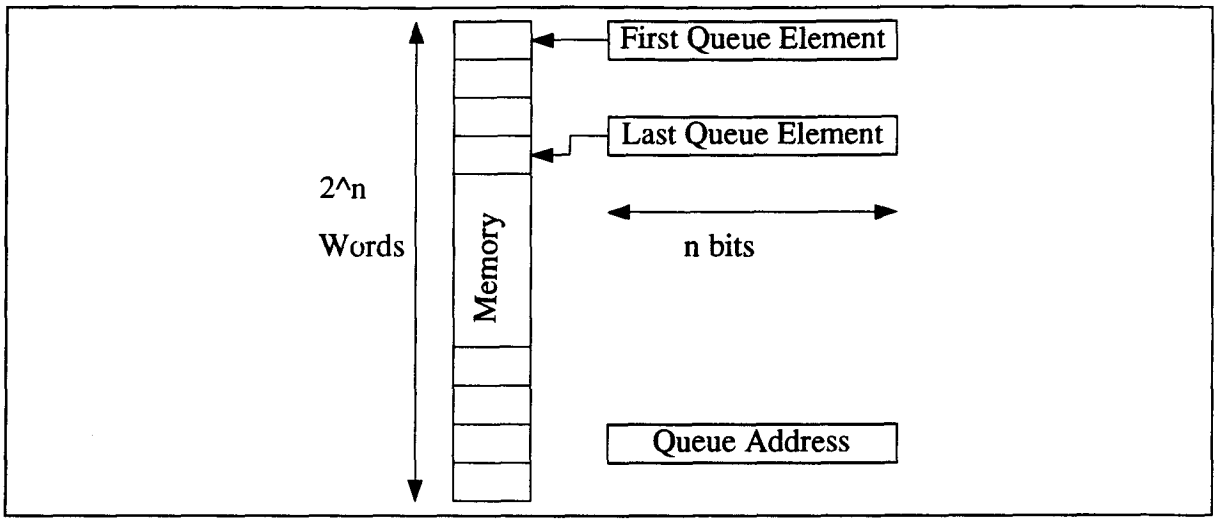

Figure 36 Queue Information

When the queue-processor sees that there is an element in the queue and that any one of the processing elements is free, it initiates a write to the I/O mapped location of the free processor. This increments the last element pointer of the queue (figure 36) which signifies that the job is assigned to a processor. After the processor finishes the job it signals the queue-processor of its availability status. The processor signals that it is free by writing any value into its I/O mapped address. Figure 37 shows a simplified diagram of how the queue-processor and four PEs can maintain correct processor availability states. In figure 37, **comp** is a comparator and the other components are explained below. Correct processor availability states are maintained by the queue-processor. It maintains a toggle bit (**p<1–4> state**'s in figure 37) for every processor that it assigns work to. This bit toggles on each write to the I/O mapped location of a processor (**p<1–4> addr** in figure 37). Initially all bits are set to 1 signifying that all processors are busy (**p<1–4>** = 1) and upon start-up each processor writes to its I/O mapped location which signals that it is free. When there is work for a processor to do, the queue-processor writes the information to any of the free processors. Suppose p4 gets the work written to it (ie **w** = 1 and **AddrBus** = **p4 addr**). This changes **p4**'s **state** in the queue-processor to 1, which signifies that processor 4 is busy. Once the processor finishes its work, it writes any information to its I/O mapped location (ie **w** = 1

and **AddrBus = p4 addr**) which resets **p4**'s **state** in the queue-processor and signals that this processor is free. The queue-subsystem can then assign this processor new work.

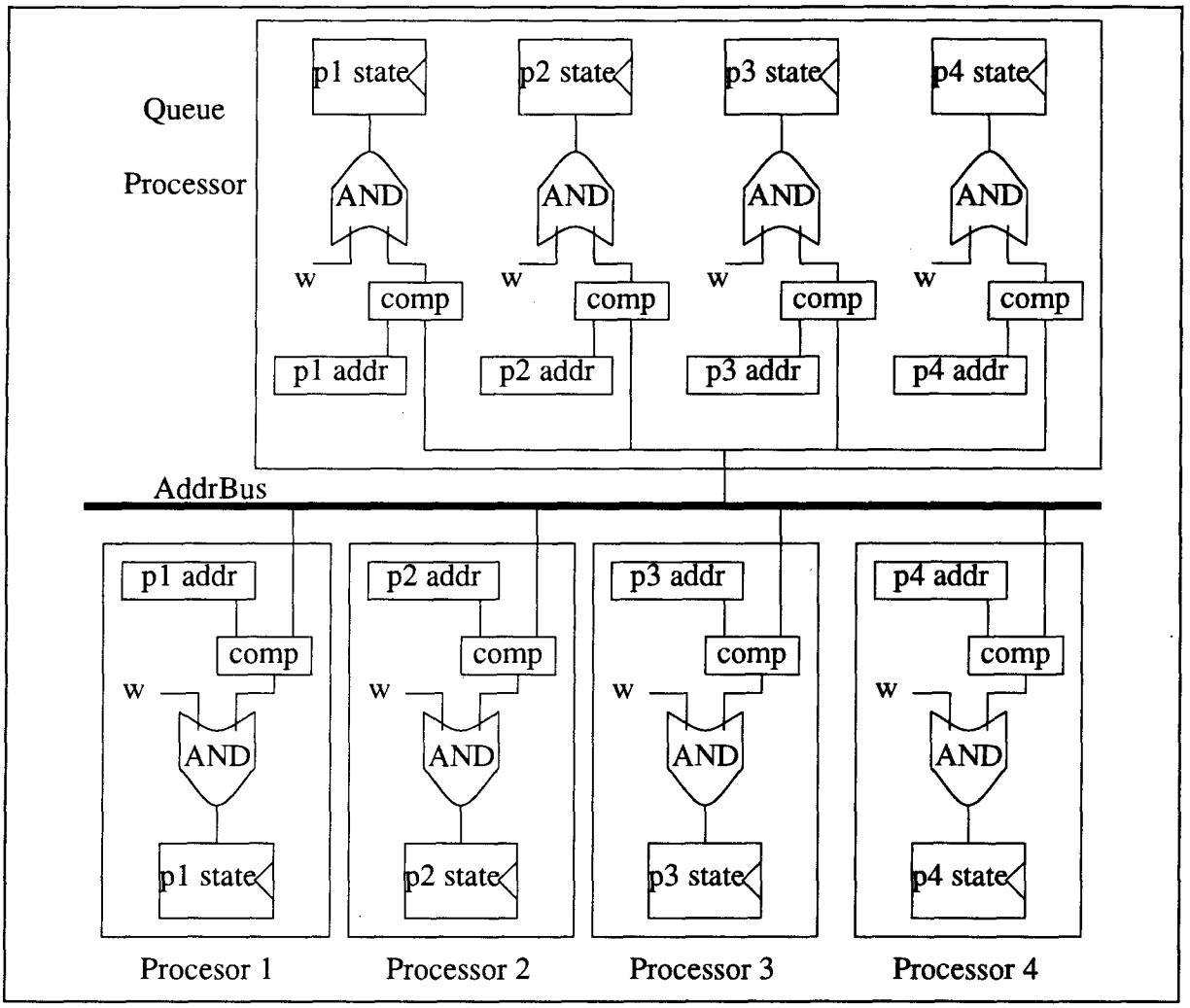

Figure 37 Processor State Information

6 Simulations

This section describes how the parallel encoding/decoding algorithms were developed using Unix System V IPC calls and then simulated on the parallel VHDL model. Figure 38 shows the setup of all the software used to achieve the results presented in chapter 7. Sections 6.1—6.5 provides details on each of the modules in figure 38 and section 6.6 explains the complete simulation cycle. Section 6.1 discusses the tools used to generate the algorithm. Section 6.2 discusses the tool used to create the memory images that are used in the VHDL simulation. Section 6.3 specifies the features of the assembler used to create the code for the VHDL simulation. Section 6.4 explains how the VHDL model uses the information that is generated for it. Section 6.5 explains the utilities used throughout the simulation process.

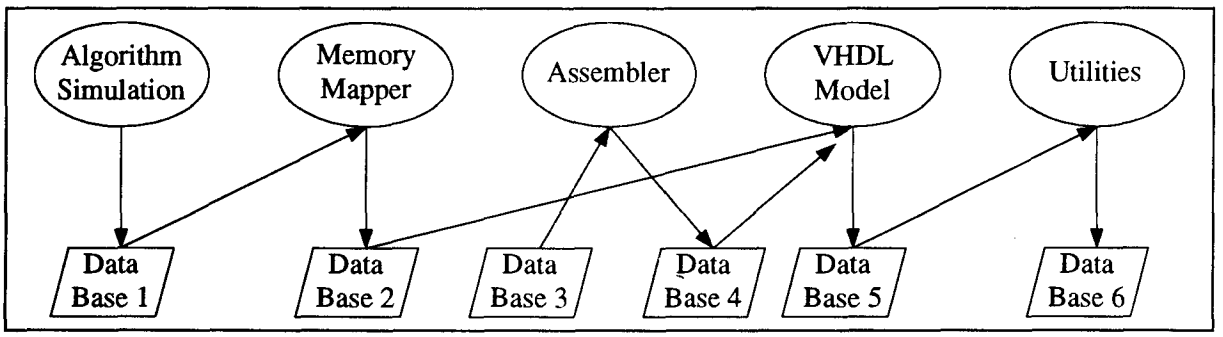

Figure 38 Software used in Simulation

6.1 Simulation with System V IPC

System V IPC calls are used because they allow shared memory and message queues to be set up in a similar manner to the hardware configuration. During this project, four different algorithms were simulated on a Unix based machine using System V IPC calls. The final algorithm is the best of the four and it was presented in the previous sections of this thesis. The setup of the IPC communication software had to resemble the final hardware architecture so that porting of code to the VHDL model would not require redefining the algorithm. With this restriction in mind, the software was setup as shown in figure 39.

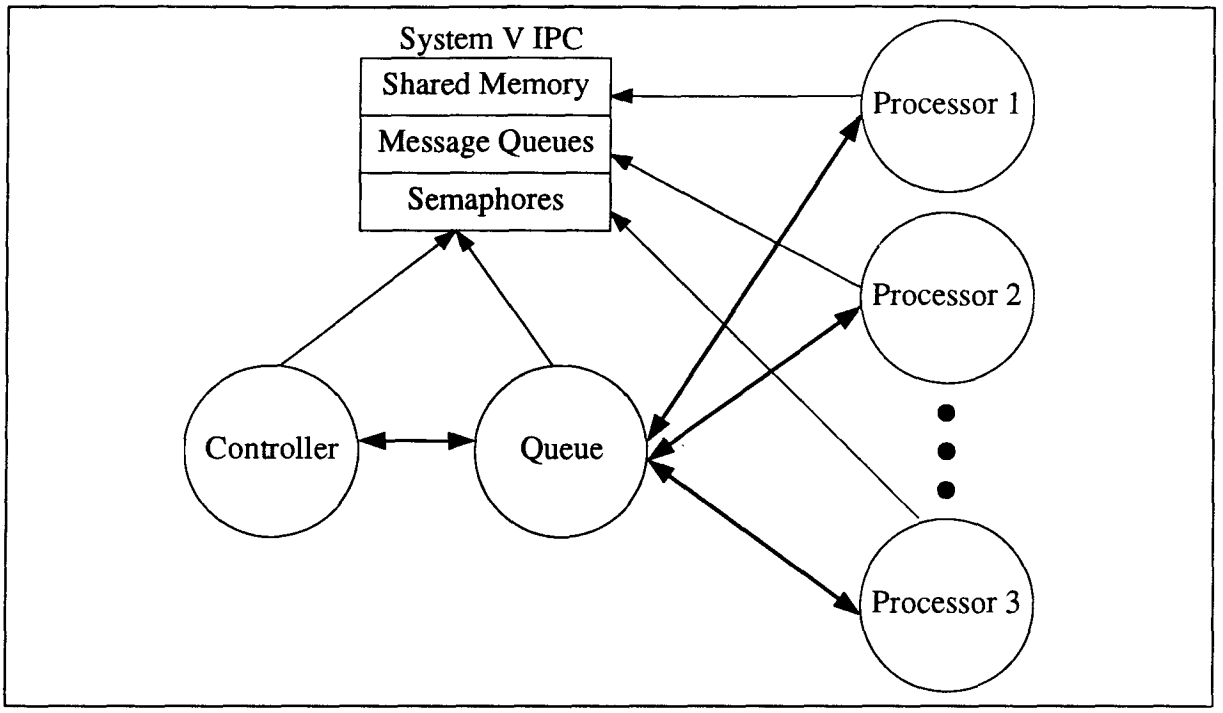

Figure 39 Development of Algorithm using Unix and System V IPCs

The functionality of the controller, the queue and processor is mapped to the hardware equivalents of the host, the queue-processor and the processors. The System V IPC code is designed so that the modules containing information to be reused in the memory mapper are in separate files. Data Base 1 consists of the System V IPC code. For encoding, the information that generates the encoding queue elements is kept in a separate file. The encoding information is the same information shown in figure 16 of chapter 3. For decoding, the information that generates the type tree is also kept in a separate file.

6.2 Memory Mapper

The memory mapper takes its input from Data Base 1 and generates the initial memory images of the shared memory and the queue-processor. The encoding memory mapper generates the shared memory image that holds the following: the data structure to be encoded, the queue elements for this data structure and the pointer to the available free memory. It also sets up the queue-processor with the appropriate pointers to the shared memory queue

elements. The decoding memory mapper generates the shared memory image that holds the following: the transfer syntax, the decode type tree, the first queue element and the pointer to the available free memory. It also sets up the queue-processor with the appropriate pointer to the first queue element. The shared memory images generated are written to Data Base 2.

6.3 Assembler

The assembler reads in the text file to be assembled and generates an output file that is a text file with binary numbers. This pseudo binary file is the input to the processors in the VHDL model. Data Base 3 holds all the assembler code and the output of the assembled code goes into Data Base 4. The assembler provides these three basic functions:

1. Converts text description to opcodes.
2. Allows labels and calculate the destination address of branch instructions.
3. Constructs multiple simple instructions from a single assembler instruction.

The assembler does not flag potential errors that occur from the delayed branch operation. This is a programmer's responsibility. Figure 40 shows examples of an assembler text file.

```
{ Contents:    This file is a sample assembler code. }

begin
        addq    r1,r0,5;     { load 5 into register 1 }
loop:   subq    r1,r1,1;     { decrement register 1 }
        briqz   stop;        { branche on zero to stop label }
        add     r0,r0,r0;    { no op }
        briq    loop;        { branches to loop label }
        add     r0,r0,r0;    { no op }
stop:   halt;                { stop processing }
end.
```

Figure 40 Example Assembler Code

Comments are allowed anywhere within the assembler code provided they are surrounded by "{}". The above code sets register one equal to five, decrements it until it is zero, and then halts the processor. *Briqz* is a branch on zero to the label. The *add* instruction after the branch instructions is present for a delayed branch error not occur. The rest of this code should be self explanatory.

6.4 VHDL Model

Upon starting up, the VHDL model loads the processors with their binary program from Data Base 4. The VHDL model also loads the shared memory image and the queue memory image from Data Base 3. The model then runs the simulation until completion, terminates, and produces its output in Data Base 5.

6.5 Utilities

Some utilities were constructed to aid in the debugging process and to analyze the output of the VHDL simulation. The following is a list of them:

- **mm2hex** The VHDL simulator deals with text files of binary numbers. This utility converts these pseudo binary files to hexadecimal so that it is easier to read.
- **mm2idx** This program takes the memory dump of an encoding run and produces the contents of the idx structure. The task of verifying the encoded result is easier with this tool.
- **bin2hex** When the VHDL model is put into debug mode it produces information on each instruction executed such as the contents of the instruction register, the PC and various other registers. This program takes this pseudo binary file and converts it to a neater hexadecimal format.
- **combine** Interleaves binary and hex files so that either the binary or hex value can be consulted depending on which makes interpretation easier.

6.6 Description of a Complete Simulation Cycle

The steps required to complete a simulation cycle and achieve the results presented in chapter 7 are explained in this section.

The steps required to generate a simulation result for an encoding are:

1. Choose an ASN.1 specification and a value instance.
2. Construct the C data types, place the value instance in the C structures and generate the CASN.1 like structured encoding routines.
3. Compile the data from list item 2 with the System V IPC code. Since the data types and the structure encoding routines are hand generated they may have errors in them. Finding these errors by running the System V simulation is easier than looking through the pseudo binary files and hex files generated by the VHDL simulations.
4. Use the memory mapper software to create the memory images for the encode.
5. Construct the assembler code needed by the VHDL processors for encoding.
6. Run the simulation of the VHDL model using the data generated by 4 and 5.
7. Verify encoding results using the utilities discussed in 6.5 and gather simulation data.

The steps required to generate a simulation result for a decoding are:

1. Choose an ASN.1 specification and a value instance.
2. Construct a decode type tree in C.
3. Run the encoding simulation and use the mm2idx utility of section 6.5 to extract the transfer syntax of the encoding routine.
4. Compile the data from list item 2 with the System V IPC code. Since the decode type tree is hand generated there may be errors present in the type tree. Finding these errors by running the System V simulation is easier than looking through the pseudo binary files and hex files generated by the VHDL simulations.
5. Use the memory mapper software to create the memory images for the decode.
6. Construct the code needed by the VHDL processor for decoding.

7. Run the simulation of the VHDL model using the data generated by 5 and 6.
8. Verify decoding result using the utilities discussed in 6.5 and gather simulation data.

7 Results

The initial choice of the tests to simulate the parallel algorithm raised many questions, some of which are discussed in the following sections. Section 7.1 discusses the need for standard test vectors for ASN.1 encoders/decoders and section 7.2 presents the statistics of the simulation.

7.1 Need for Standard Test Vectors

At present no standard tests are used when encoding/decoding speeds are quoted. Comparing results is difficult because test vectors can be chosen to bring out the optimal performance of specific encoders/decoders. Two possible pathological cases are:

- In VASN.1 strings are not copied from the incoming TS to the local data structure. Instead a pointer in the local representation is set to the data. An ASN.1 complier like CASN.1 copies the strings. If we are dealing with 1 Kbyte per string then the performance of VASN.1 is significantly faster.
- Suppose we assume an ASN.1 type is defined as one string and that the string octets had to be copied from the incoming TS to the local data structure. If we compared CASN.1 to the parallel algorithm presented in this thesis, the speeds of the two would be approximately equal. When strings of information must be copied, with little or no processing, the access to memory becomes the bottleneck.

7.2 Simulation Statistics

It is impossible to compare the parallel algorithm presented in this thesis and the algorithm in VASN.1[3]. The result in [3] presents the total number of instructions executed for an unknown abstract syntax and value instance. The reconstruction of the results of VASN.1 would require acquisition of the hardware models and re-simulation with a known abstract syntax and value instance. Since the reconstruction is not possible, the parallel algorithm is compared to the code generated by UBC's ASN.1 compiler.

The processors in the parallel algorithm were simulated at a speed of 25 MHz and used the instructions per byte count shown in Appendix F. The results are summarized below.

ASN.1→		Personnel Rec.		Ten Reals		Ten Integers	
Units→		Mbits/s		Mbits/s		Mbits/s	
# Proc.	Oper.↓	Encode	Decode	Encode	Decode	Encode	Decode
1	noop	34.3	8.6	99.5	24.2	26.1	6.3
2	noop	59.6	15.5	167.5	29.1	43.9	7.6
10	noop	101.5	22.6	208.2	28.8	54.6	7.6
1	copy	13.7	7.2	40.8	21.0	11.9	5.8
2	copy	23.9	13.2	65.4	28.4	19.2	7.6
10	copy	40.0	20.3	80.2	28.1	24.0	7.5
1	50%	6.7	2.5	13.9	3.4	4.1	2.0
2	50%	12.1	4.94	25.5	6.6	7.4	3.7
10	50%	28.1	15.1	61.6	16.9	16.5	6.05
1	100%	4.5	1.6	8.3	1.8	2.5	1.2
2	100%	8.4	3.2	15.6	3.6	4.7	2.3
10	100%	21.4	10.9	45.9	12.0	12.5	5.0
1	CASN	.253	.242	—	—	—	—

Table 8 Simulation Results in Mbit/s.

The first row of table 8 shows the three different ASN.1 types simulated with the parallel algorithm. Column one shows the number of processors the algorithm is simulated with. Column two shows the actions that the processors execute. These actions and the analysis of their results are as follows:

- **noop** When a primitive call is made no operations are executed. The simulation provides the upper limit on the throughput of the algorithm and the overhead associated with the

algorithm.

- **copy** When a primitive call is made, the contents in the data structure of the application are copied to the TS for encoding and vice-versa for decoding. The throughputs for the **noop** simulations are higher than the throughputs for the **copy** simulations due to the shared bus contention caused by the copying of the IA5String types. The IA5String types require almost no processing and must be copied over the shared bus.
- **50%** When a primitive call is made, the **copy** operation is executed and as well as 50% of the instruction/byte count generated by the serial program. This simulation provides a middle ground result in relation to the copy and the 100% option specified below.
- **100%** When a primitive call is made, the copy operation is executed and as well as 100% of the instruction/byte count generated by the serial program. This simulation provides a lower limit on the achievable throughput.
- **CASN** This is the encoding/decoding rate for CASN.1[12]. This result was achieved by running CASN.1 on a SUN 3/260 under UNIX 4.2 BSD. The SUN 3/260 machines use a 25MHz 68020 Motorola processor. The single processor simulation of the parallel algorithm produces higher throughput than the single processor implementation CASN.1 because CASN.1 was designed only for functionality and not optimized for speed. The single processor simulation of the parallel algorithm uses the instruction/byte count generated by an optimized version of CASN.1, named a2c, that is under development at UBC.

The higher speeds achieved when encoding are attributed to the following:

- When encoding, the compiled encoding routine approach is followed while in decoding the type tree approach is followed. Ref.[16] notes that the compiler approach is usually faster than the type tree approach due to the extra processing required for traversing the type tree.
- When decoding, the extra processing requirements of a type tree increases the amount of communication overhead in the parallel system which slows down the decoding speeds.

- When encoding, pipelining and parallelism are achieved by having the host and the parallel hardware operate simultaneously. When decoding, only parallelism is exploited because the parallel hardware operates alone.
- When decoding, the input TS is compared to the expected value. When encoding, no comparison operation is performed.

From the data presented in table 8 it is difficult to determine the optimal number of processors to use in this parallel setup. Deciding the margin of diminishing returns is a difficult problem because the maximum number of processors used depends on how hierarchical the ASN.1 structure is and the data instance being encoded/decoded. For example, if there is an ASN.1 definition of two octets then the maximum processor utilization is two and the memory bandwidth is the bottleneck. In another case suppose that there is as ASN.1 type defined as a SEQUENCE of 20 object identifiers. It is possible that 20 processors could be kept busy decoding these object identifiers without running into a memory bandwidth problem. Thus the parallel version of the software performs at a speed that is comparable to the serial version in the worst case. The worst case occurs when there is one octet in the ASN.1 type and the memory bandwidth is the bottleneck. In all other cases the parallel version significantly outperforms the serial version.

8 Future Work

This section describes enhancements to the present algorithm, which were not tested for this thesis project, but are suggested for future work.

8.1 Simulations

The following is a list of some of the improvements that can be made to the simulation system:

- See the impact of programming primitive encoding/decoding in assembly language.
- Make an ASN.1 compiler that reads in text files of ASN.1 and generates the equivalent C type, the structures encoding routines and the decoding type tree.
- Modify the back-end of a public domain C compiler so that it produces output that the VHDL model can directly use.
- Determine the effect of changing the single bus connection network. A good starting point for this is a double bus with independent RDC arbitration.

8.2 Processing Element

The following improvements that can be made to the processing elements are listed in order of importance. The constraining factor for the suggestions below is the total available die space on an IC.

- Install a two window register file in each processor. The PE calls subroutines that cause the environment of the PE to be saved to memory. On the return of the subroutine the environment is restored from memory. This causes memory reads and writes thereby that slows down the execution speed of the processor.
- On chip memory shown in Figure 41 is used as a data cache. The advantage of an on chip cache is that the load on the shared bus decreases because the processing elements can use burst reading/writing of memory reducing the contention for the shared bus.

This allows the processor to acquire control of the bus faster and reduce the total waiting time for the bus.

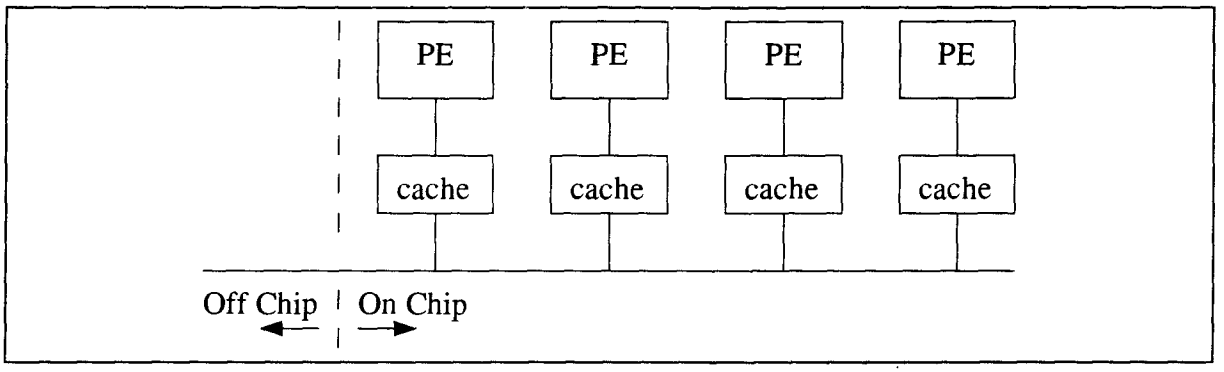

Figure 41 Local Cache Setup

8.3 Parallel use of PEs

The following provides the improvements that can be made to the parallel hardware. Again the constraining factor is the total available die area on an IC.

- The single shared bus can be replaced by a multiple shared bus with data caching as shown in figure 42. This decreases the time a processor spends waiting for a bus and increases the work performed. The parallel hardware shown in 41 and 42 can be specified and simulated to determine which is better.
- Multiple ED library calling queues in the parallel processor. This would allow more than one encoding/decoding to take place and can also prioritize the system. This would raise the system performance and PE utilization.

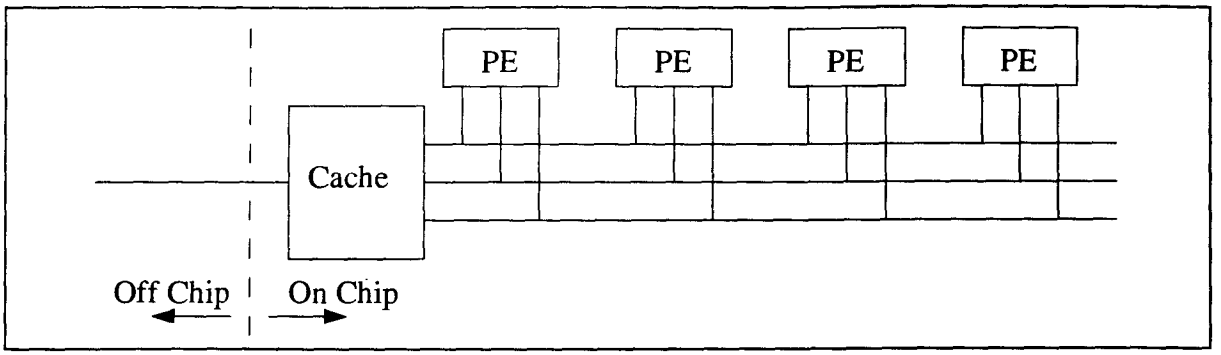

Figure 42 Multi-Bus Global Cache

8.4 Fabrication Constraints

- When more elements, such as large register files and data caches, are put onto a die the total area occupied on the die increases and hence there is a need for fault tolerance and testability. Hardware designs must keep this in mind.
- An initial hand layout instead of an automatically generated layout could save on chip area and would improve the speed of the processor. Ref.[7][14][22] contain examples of the hand layouts found in literature.

9 Conclusion

This thesis has shown that a parallel ASN.1 encoder/decoder allows higher throughput than a serial implementation. Table 9 presents the worst case simulation results for the ASN.1 PersonnelRecord type specified in Appendix E. The results in table 9 are normalized with respect to the serial algorithm of CASN.1 and the parallel algorithm executed on one processor. The results in table 9 normalized with respect to the parallel algorithm running on one processor provide a better indication of the speedup achievable by the parallel algorithm. Comparing CASN.1 to the parallel algorithm is unfair because the code generated by CASN.1 is not optimized for speed and the values used in the parallel algorithm simulations are derived from an optimized version of CASN.1. The optimized version of CASN.1 is named a2c and is in the final stages of development at UBC.

		CASN.1 Normalized		Par. 1 proc. Normalized	
Algorithm	Processors	Encode	Decode	Encode	Decode
CASN.1	1	1	1	0.056	0.15
Parallel	1	17.8	6.6	1	1
Parallel	10	84.6	45.0	4.8	6.8

Table 9 Worst Case Simulation Results for PersonnelRecord.

The simulation results derived in this thesis can not be compared to simulation results of VASN.1 because the published throughputs of VASN.1 do not specify the ASN.1 type or value instances that were used. The major features of the parallel system simulated in this thesis and a structural comparison of it to VASN.1 are presented below.

- An optimal ASN.1 processing element can be designed based on the instruction usage statistics gathered during the simulations.
- A distinct interface to the parallel system has been provided for users. This will allow for easy integration with other systems.

- The rotating daisy chain bus arbitration scheme that is used in the parallel system of this thesis provides the best bus arbitration scheme for the required amount of hardware[2]. In the VASN.1 system the bus arbitration is preformed by four processing elements which is an inefficient use of the processing power.
- The hardware of VASN.1 uses an associative memory while the hardware presented in this thesis does not. Due to the complexity of implementing an associative memory, the system implementation time and the system area requirements will be high in VASN.1.
- VASN.1 has a limited number of processors due to its hardware configuration. The hardware designed in this thesis is structured and scales gracefully. This allows more processing elements to be included in the parallel encoding/decoding which increases the achievable parallelism.
- VASN.1 does not do a parallel to serial conversion of the data structures it provides for the users but assumes all users accept parallel data structures. The users of VASN.1 must pay the high cost of programming their applications to handle parallel data structures. The algorithm of this thesis provides a serial to parallel data structure conversion at a low cost hence user's programs are not required to work with parallel data structures. An algorithm for converting a serial data structure to a parallel data structure in the presentation layer has never been proposed in the literature.

Based on the above data one can conclude that the basis for a significantly better parallel algorithm than VASN.1 has been provided and the initial simulation results are promising.

Bibliography

[1] Peter J. Ashenden. The VHDL Cookbook. This is postscript document available by FTP from chook.adelaide.edu.au, July 1990. University of Adelaide.

[2] W. L. Bain, J.R. Ahuja, and S.R. Ahuja. PERFORMANCE ANALYSIS OF HIGH-SPEED DIGITAL BUSES FOR MULTIPROCESSING SYSTEMS. In *Proceedings of the 8^{th} Annual Symposium on Computer Architecture*, volume 8, pages 107–133, Minneapolis, Minnesota, May 12–14, 1981.

[3] M. Bilgic and B. Sarikaya. AN ASN.1 ENCODER/DECODER AND ITS PERFORMANCE. In *IFIP PSTV X*, pages 133–150, Ottawa, Canada, June 12–15, 1990.

[4] D. Chapman. A Tutorial on Abstract Syntax Notation One (ASN.1). Transmission #25 by Open System Data Transfer, December 1986. Omnicom information service, Omnicom Inc., ISSN 0741286X.

[5] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. *SIGCOMM '90*, 24(4):200–208, September 1990.

[6] Sun Microsystems Inc. *XDR: External Data Representation Standard, (RFC 1014) in Internet Working Group Requested for Comments.* Network Information Center, SRI International, Menlo Park, Calf., June 1987. No. 1014.

[7] Robert W. Sherburne Jr., Manolis G.H. Katevenis, David A. Patterson, and Carlo H. Sequin. DATAPATH DESIGN FOR RISC. In *CONFERENCE ON ADVANCED RESEARCH IN VLSI, MIT*, pages 53–62, Cambridge, Massachusetts, January 1982.

[8] Mike Kong. *Network Computing System Reference Manual.* Prentice-Hall, Inc., 1987. ISBN 0136170854.

[9] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. *VHDL: hardware description and design.* Kluwer Academic Publishers, 1989. ISBN 0–7923–9030–X.

[10] Motorola. *MC88100 RISC MICROPROCESSOR USER'S MANUAL*, 1989. Order Number: MC88100UMAD/AD, page 1–1 — 1–13.

[11] Gerald W. Neufeld and Son Vuong. An Overview of ASN.1. *Computer Networks and ISDN Systems*, 23:393–415, 1992.

[12] Gerald W. Neufeld and Yueli Yang. The Design and Implementation of an ASN.1–C Compiler. *IEEE Transactions on Software Engineering*, 16(10):1209–1220, October 1990.

[13] U.S. Department of Commerce. NBS, "User Guide for the NBS Prototype Compiler for Estelle, final report". Technical report, National Bureau of Standards, October 1987. Report No. ICST/SNA-87/3.

[14] D. Pucknell and K. Eshraghian. *Basic VLSI Design Principles and Applications.* Prentice-Hall, 1985. ISBN 0–13–067851–1.

[15] Mike Sample and Gerald W. Neufeld. A High Performance ASN.1 Compiler. *in preparation at UBC Computer Science*, 1992.

[16] Mike Sample and Gerald W. Neufeld. Support for ASN.1 within a Protocol Testing Environment. In *Proceedings of the IFIP TC/WG 6.1 Third International Conference of Formal Description Techniques for Distributed Systems and Communications Protocols*, pages 295–302, Madrid, Spain, November 5–8 1990.

[17] W. Stallings. *Data and Computer Communication.* New York: Macmillan, 1988.

[18] Sun Microsystems Inc. *A RISC Tutorial*, 1988. Part Number: 800–1795–10, pages 1–15, Revision A of 9 May.

[19] Len Takeuchi. STUDY OF OSI PROTOCOL PROCESSING ENGINES. Master's thesis, Department of Electrical Engineering, University of British Columbia, 1991.

[20] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1988. ISBN 0–13–162959–X.

[21] Kenneth J. Thurber, E. Douglas Jensen, Larry A. Jack, Larry L. Kinney, Peter C. Patton, and Lynn C. Anderson. A systematic approach to the design of digital bussing structures. In *AFIPS Conference Proceedings, Fall Joint Computer Conference*, volume 41 part 2, pages 719–740, 1972.

[22] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison-Wesley, 1985.

[23] The Wollongong Group, 1129 San Antonio Rd. Palo Alto, CA, USA. *ISODE, The ISO Development Environment: User Manual*, February 1990. Version 6.102 Volume 1 and 4.

[24] W. Wu, M. Bilgic, and B. Sarikaya. VHDL MODELING AND SYNTHESIS OF AN ASN.1 ENCODER/DECODER. In *Canadian Conference on Very Large Scale Integration, CCVLSI '90*, pages 1.5.1–1.5.8, Westin Hotel, Ottawa, Ontario, October 21–23, 1990.

[25] Wayne Wu. VASN1: A HIGH-SPEED MULTI-RISC EMBEDDED SYSTEM FOR

ASN.1 ENCODING AND DECODING. Master's thesis, Department of Electrical Engineering, Concordia University, September 1991.

[26] Recommendation X.208. *Specification of Abstract Syntax Notation One (ASN.1)*, pages 57–130. CCITT Blue Book, Volume V.III, Fascicle VIII.4, International Telecommunications Union, Geneva, Switzerland, 1989.

[27] Recommendation X.209. *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, pages 131–151. CCITT Blue Book Volume V.III, Fascicle VIII.4, International Telecommunications Union, Geneva, Switzerland, 1989.

[28] Yueli Yang. ASN.1–C COMPILER FOR AUTOMATIC PROTOCOL IMPLEMENTATION. Master's thesis, Department of Computer Science, University of British Columbia, 1988.

Appendix A Acronym Definitions

ASN.1	Abstract syntax notation one
BER	Basic encoding rules
FIFO	First in first out
IC	Integrated circuit
IPC	Inter-process communication
ISO	International Standards Organization
PDU	Protocol data unit
PE	Processing element
PPDU	Presentation protocol data unit
RDC	Rotating daisy chain
TLV	Tag, length and value
TS	Transfer syntax
VHSIC	Very high speed integrated circuits
VHDL	VHSIC hardware description language
UBC	University of British Columbia

Appendix B Cost per Instruction Mapping

Action	Cost
allocate memory	5
assignment	2
complex assign	3
write to q	2
logic operation	1
add or sub	1
stack push	3
stack pop	2

Note that CASN.1 does not do any bounds checking on the stacks. The cost increases if CASN.1 does.

Thesis parallel algorithm.

EncodePrimitive Tag

Cost	Operation
5	1 allocate memory
10	5 assignments
1	1 write to queue

EncodePrimitive Non Tags

Cost	Operation
5	1 allocate memory
18	9 assignments
1	1 write to queue

EncodeStructBegin New Level and Node

Cost	Operation
2	1 if statement
5	1 allocate memory
28	14 assignments

EncodeStructBegin New Node

Cost	Operation
2	1 if statement
5	1 allocate memory
22	11 assignments

```
EncodeStructEnd (non root level)
        Cost            Operation
        4               2 assignments
        2               1 if statement
EncodeStructEnd (root level)
        Cost/node       Operation
        5               1 allocation
        16              8 assignments
        1               1 logic OR
        1               1 write to queue
        2               1 loop statement
        Cost/level      Operation
        4               2 assignments
        1               1 write to queue
        2               1 branch
-----------------------------------------------------
CASN.1
-----------------------------------------------------
EncodePrimitive
        20  for boolean (this is a guess)
        60  for int (this is a guess)
        300 for real (this is a guess)
EncodeStructBegin
        Cost            Operation
        3               1 complex assignment
        9               3 stack push
        10              encode tag
EncodeStructEnd (for short form definite length)
        Cost            Operation
        4               2 pops
        8               4 assignment
        3               1 complex assignment
        5               1 allocate memory
        2               1 assignment (IO_OUT)
        2               1 branch
        7 or 10         link (explained below)
        9               3 complex assignment
link with empty stack (used in EncodeStructEnd)
        2               1 pop
        2               1 if not chosen
        3               1 push
```

```
link with non empty stack (used in EncodeStructEnd)
        2                1 pop
        2                1 if chosen
        3                1 assignment
        3                1 push
```

Appendix C Pros and Cons of Static Analysis

Compilation of CASN.1 primitive routines into assembler to generate approximate instruction count has two problems.

The first problem is that the analysis of the C code is a static one. The primitive encode/decode instruction count generated is static so one should take into account the difference between a static analysis and a dynamic analysis. Generally not all the assembler instructions are executed, hence a static analysis count results in a greater overall instruction count. Also within the fragments of code there are loops that execute a number of times and these extra instructions are unaccounted for in the static analysis. Finally, within the code some routines get called more than once but the assembled code includes all subroutines once. This reduces the total instruction count in the static analysis.

The second problem is that of application-dependent extra code included in the static assembler count. The error checking code for CASN.1 should not be considered instructions for encoding/decoding primitive types. This extra code for the encoding/decoding process is specific to the implementation. To get an accurate count of the instructions that need to be executed for a primitive encode/decode one should construct the primitive routines in assembler and then analyze the resulting code by hand. By using varied data sets to test the primitive routine, instruction counts can be calculated by using loop unrolling and other such methods. Note that even though the implementation-dependent code does not constitute the minimum instructions for encoding/decoding it does represent the overhead incurred in implementing a functional algorithm.

Although the instruction counting method is limited, it is presented to show that a significant number of instructions must be executed to encode/decode a primitive type and hence parallelism is applicable.

Appendix D Processors Memory Read and Write Cycle

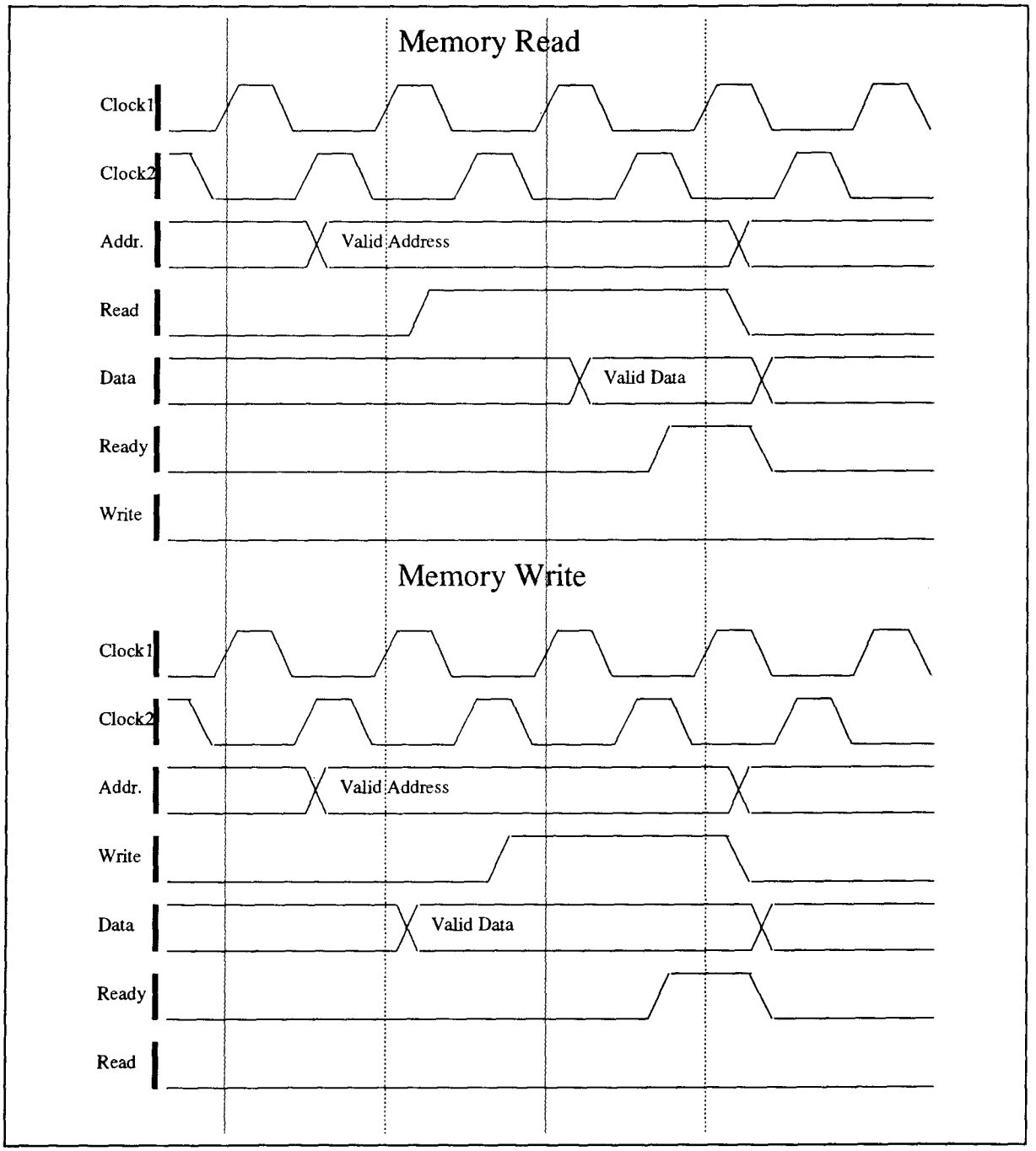

Figure 43 Memory Read/Write Timing Diagrams

Appendix E ASN.1 PersonnelRecord Type

```
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
                        Name,
title           [0]     IA5String,
                        EmployeeNumber,
dateOfHire      [1]     Date,
nameOfSpouse    [2]     Name,
children        [3]     IMPLICIT SEQUENCE OF
                        ChildInformation DEFAULT {}
}

ChildInformation ::= SET {
                        Name,
dateOfBirth     [0]     Date
}

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE {
givenName               IA5String,
initial                 IA5String,
familyName              IA5String
}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= [APPLICATION 3] IMPLICIT IA5String -- YYYYMMDD


value instance ::=
{ {givenName "John", initial "E", familyName "Smith"},
 title
  "The Big Cheese",
  99999,
 dateOfHire
  "19820104",
 nameOfSpouse
  {givenName "Mary", initial "L", familyName "Smith"},
 children {
  {{givenName "James", initial "R", familyName "Smith"},
```

```
   dateOfBirth "19570210"},
  {{givenName "Lisa", initial "M", familyName "Smith"},
   dateOfBirth "19590621"}}
}
```

Appendix F ASN.1 Types Instruction Counts

The instructions per byte values that are used in the simulation are generated by profiling a2c. Table 10 presents the average instructions executed per byte for three different data structures. The first is the PersonnelRecord ASN.1 type specified in Appendix E. The second data structure is specified in 44 and consists of ten integers. The final data structure is ten reals. The ten real type is the same as the ten integer type, with the integers replaced by reals.

```
teninstType ::= SEQUENCE {
 int1 INTEGER,
 int2 INTEGER,
 int3 INTEGER,
 int4 INTEGER,
 int5 INTEGER,
 int6 INTEGER,
 int7 INTEGER,
 int8 INTEGER,
 int9 INTEGER,
 int0 INTEGER
}

value instance = {5,5,5,5,5,5,5,5,5,5}
```

Figure 44 Ten Integers Type

	PersonnelRecord	Ten Integers	Ten Reals
Encode	14	21	13
Decode	35	46	66

Table 10 Simulations Instruction per Byte Values.