



US 20070016554A1

(19) **United States**(12) **Patent Application Publication****Dapp et al.**(10) **Pub. No.: US 2007/0016554 A1**(43) **Pub. Date:****Jan. 18, 2007**(54) **HARDWARE ACCELERATED VALIDATING
PARSER****Publication Classification**(76) Inventors: **Michael C. Dapp**, Endwell, NY (US);
Eric C. Lett, Endwell, NY (US); **Sai
Lun Ng**, Vestal, NY (US)(51) **Int. Cl.****G06F 17/30** (2006.01)**G06F 9/45** (2006.01)(52) **U.S. Cl.** **707/2; 717/143**

Correspondence Address:

MILES & STOCKBRIDGE PC**1751 PINNACLE DRIVE****SUITE 500****MCLEAN, VA 22102-3833 (US)**

(57)

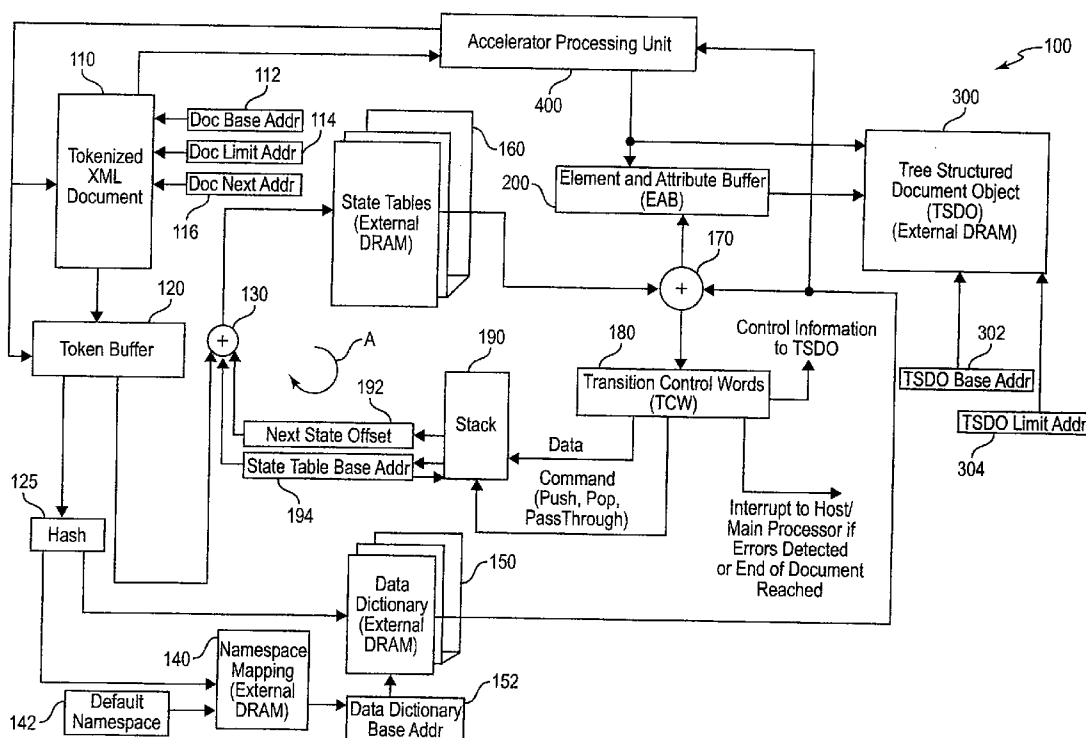
ABSTRACT

A hardware accelerated validation parser is provided to remove a large portion if not all of the processing and overhead burden of validation parsing from a host processor by parallel access to both a state table and a data dictionary based on a token and merging and selective redirection of the respective outputs thereof; a portion of a transition control word (TCW) formed by the merged data being used to advance through the state table and a portion of the TCW being used to control formation of a tree structured data object (TSDO) corresponding to a text document in a language such as XMLTM which supports interoperability and platform independence. A stack is provided to accommodate nesting of elements and aggregate elements. The formation of the TSDO can be and preferably is performed asynchronously and autonomously in parallel with the validation parsing.

(21) Appl. No.: **11/457,943**(22) Filed: **Jul. 17, 2006****Related U.S. Application Data**

(63) Continuation of application No. 10/334,086, filed on Dec. 31, 2002, now Pat. No. 7,080,094.

(60) Provisional application No. 60/421,774, filed on Oct. 29, 2002.



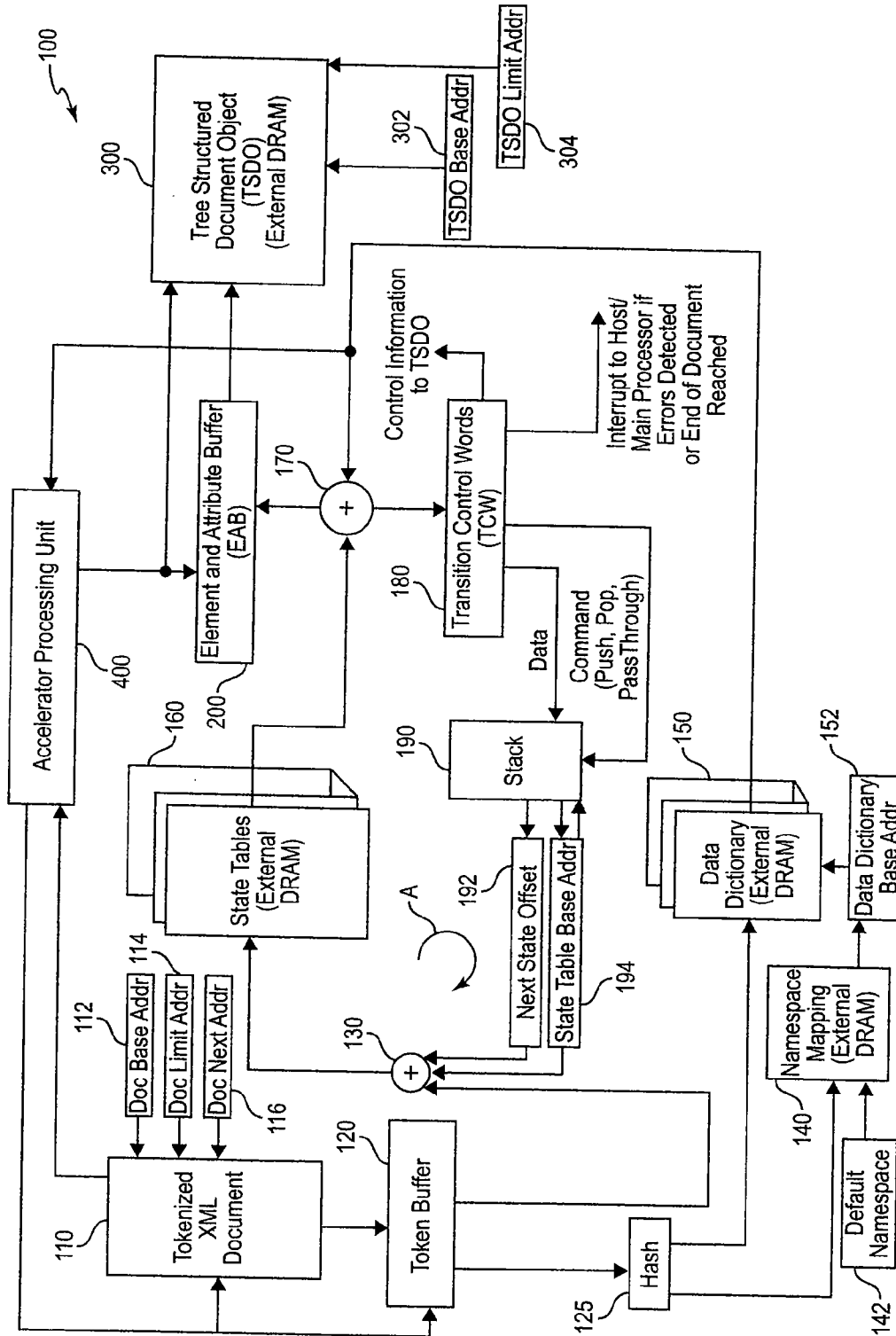


Figure 1

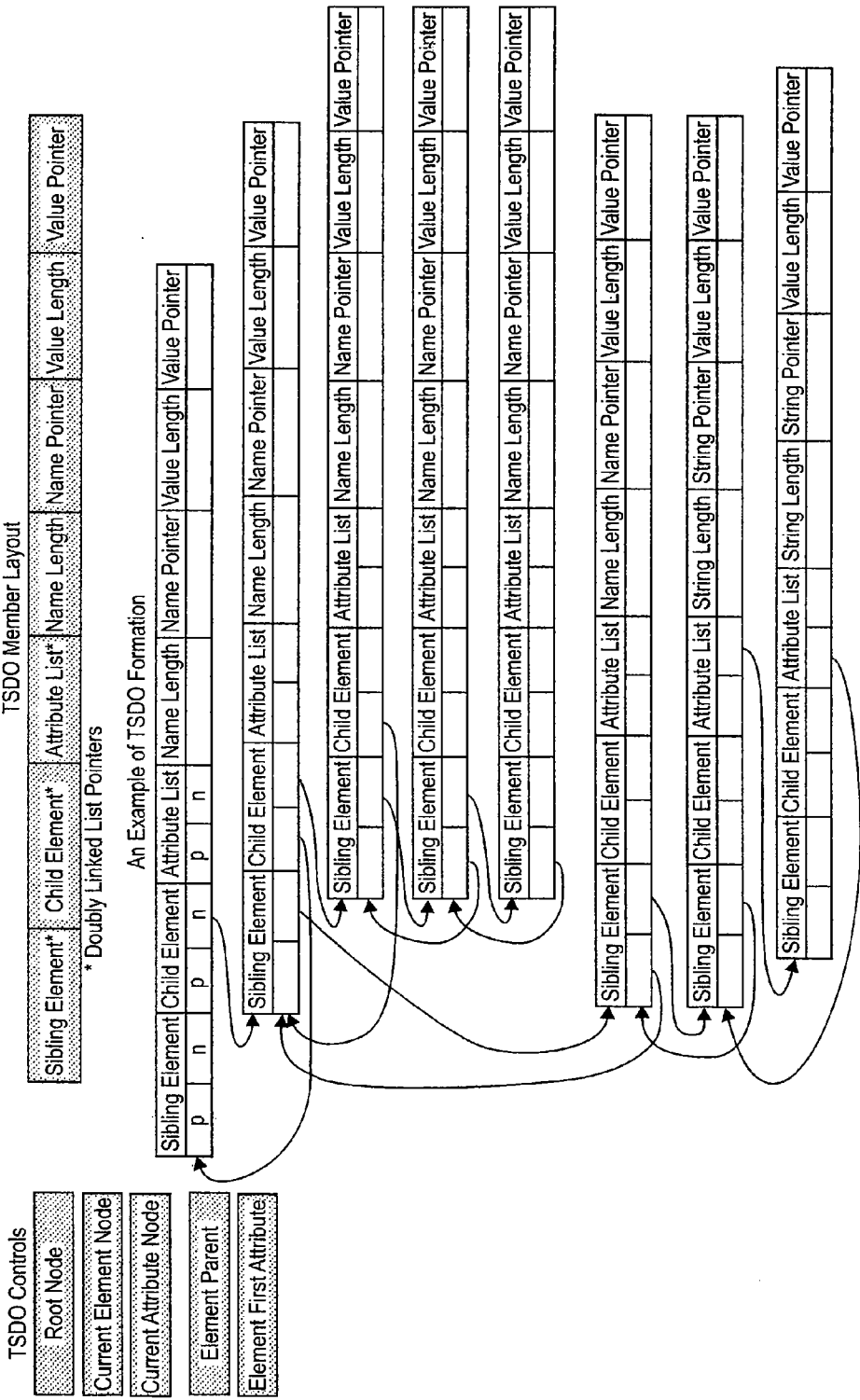


Figure 1A

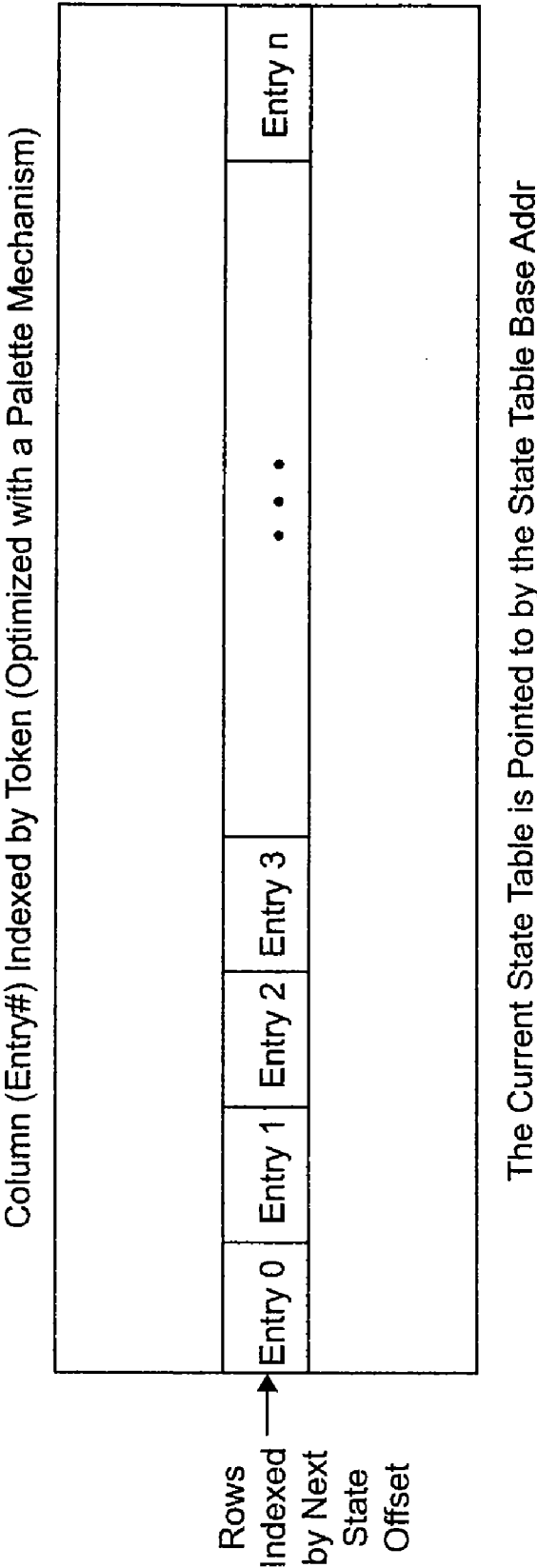


Figure 2

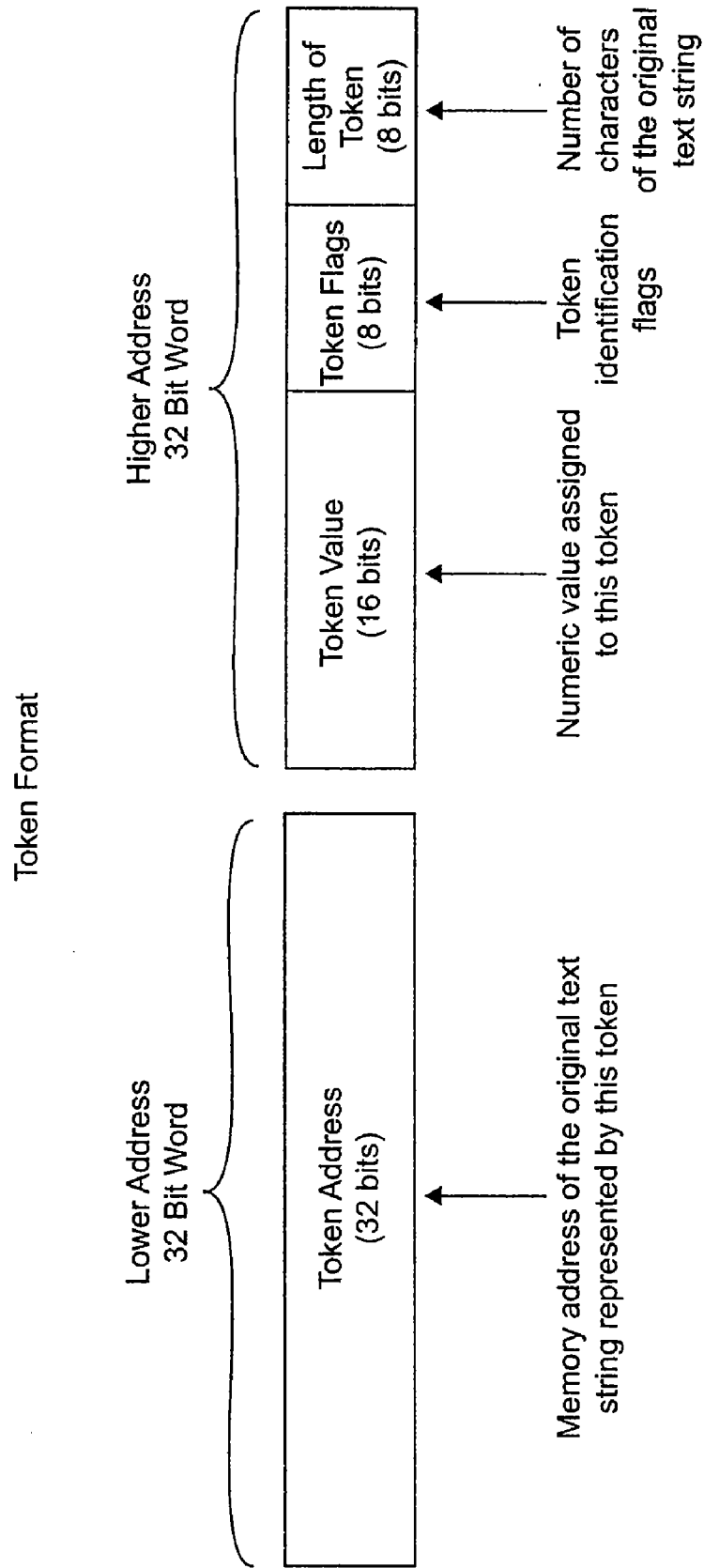


Figure 3

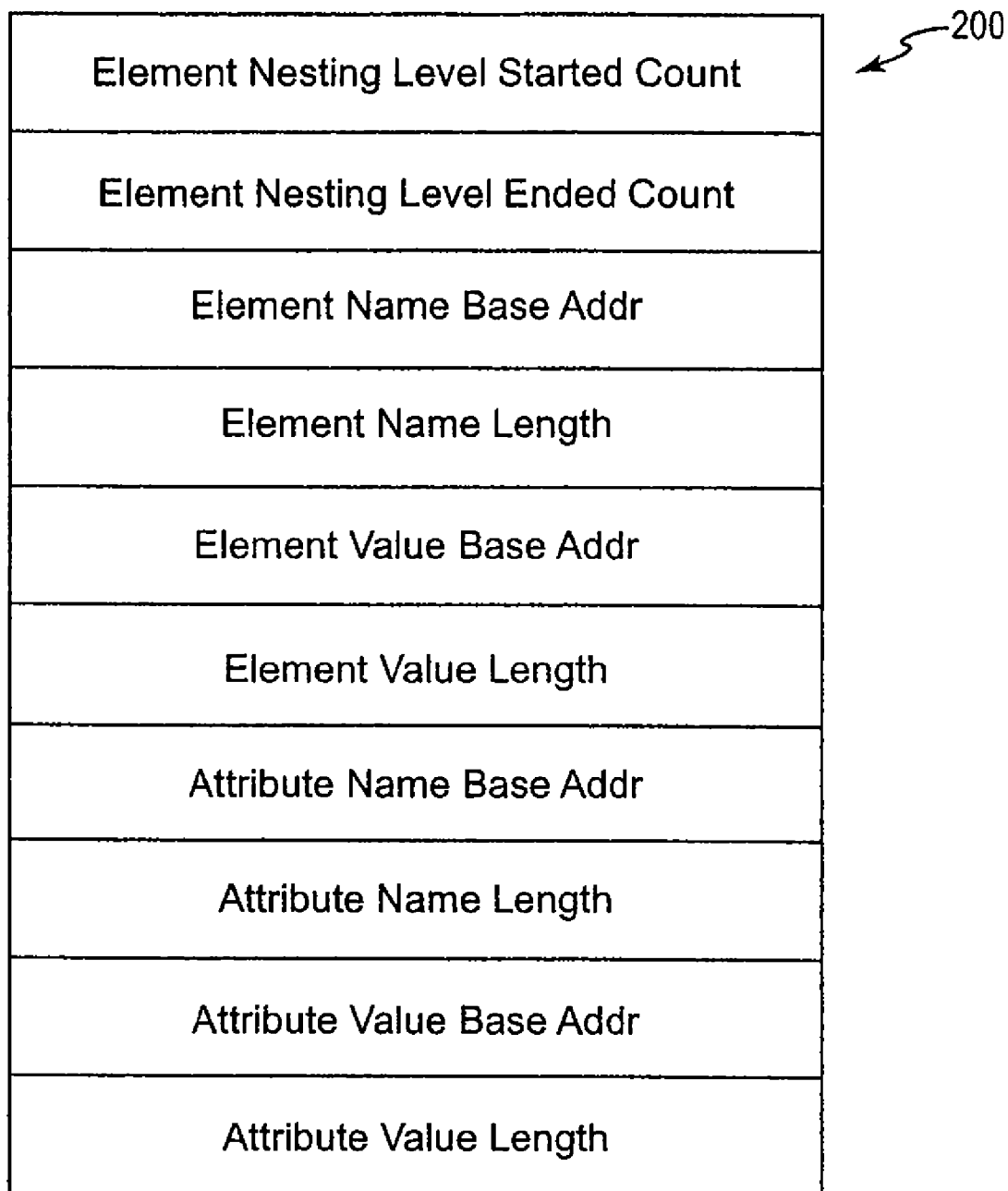


Diagram 200 is a vertical stack of 11 rectangular boxes, each containing a text label. An arrow points from the number 200 to the top of the stack.

Element Nesting Level Started Count
Element Nesting Level Ended Count
Element Name Base Addr
Element Name Length
Element Value Base Addr
Element Value Length
Attribute Name Base Addr
Attribute Name Length
Attribute Value Base Addr
Attribute Value Length

Figure 4

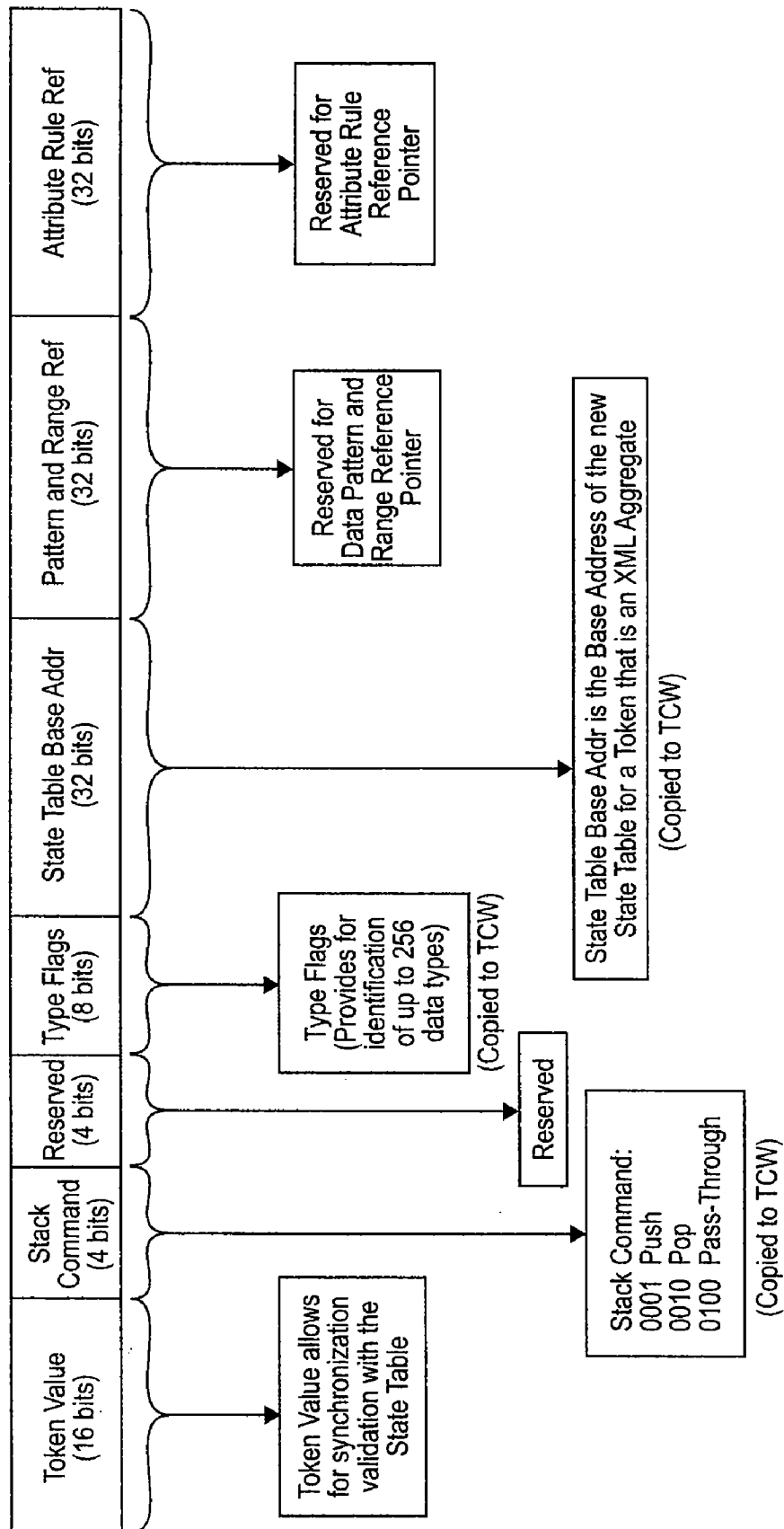
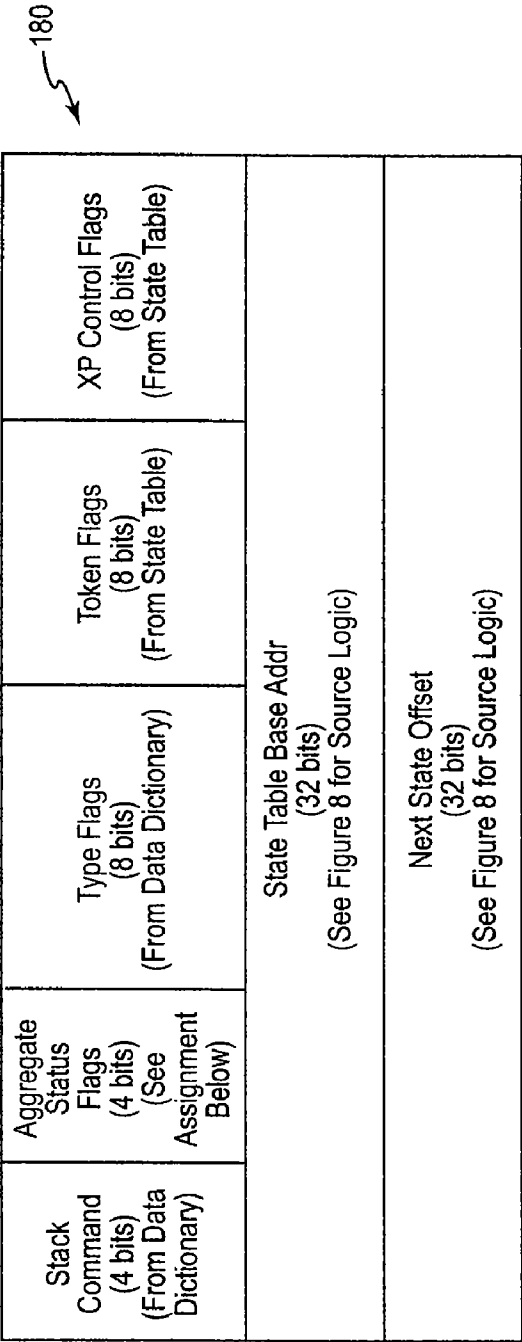


Figure 5



The TCW is basically a combination of control flags and state transition controls from the State Table and the Data Dictionary with certain rules applied. The Data Dictionary Token Value field is not being carried over to the TCW since the current token is readily available in the Token Buffer. The TCW Aggregate Status Flags field is an added field to keep track of the presence of XML aggregates as followed:

Aggregate Status Flags Assignments			
--1	--	--	(bit 0 = 1) - Current XML Element is an Aggregate
--1	--	--	(bit 1 = 1) - Previous XML Element is an Aggregate
-x	--	--	(bit 2 = x) - Reserved
x	--	--	(bit 3 = x) - Reserved

Figure 6

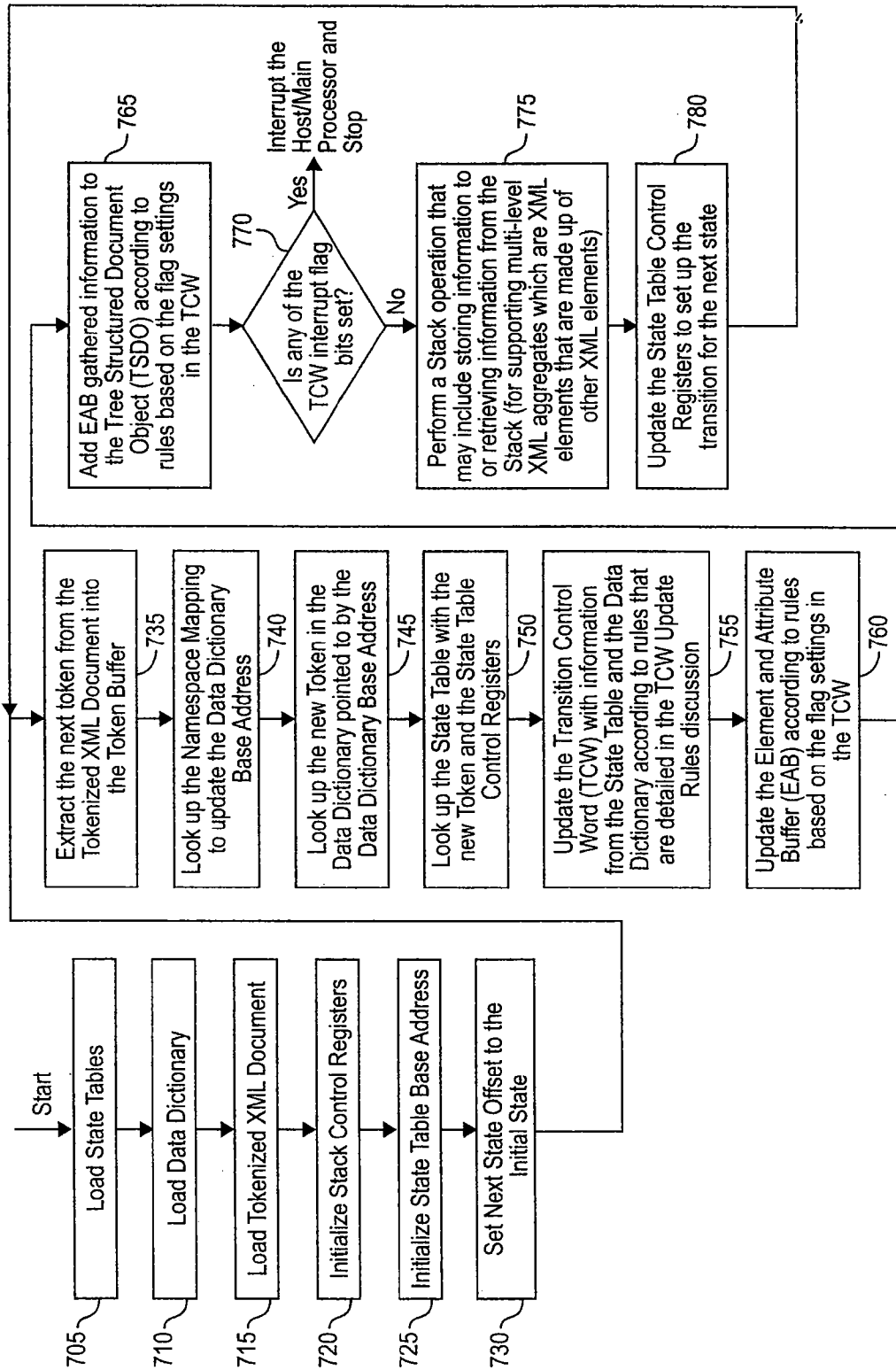


Figure 7

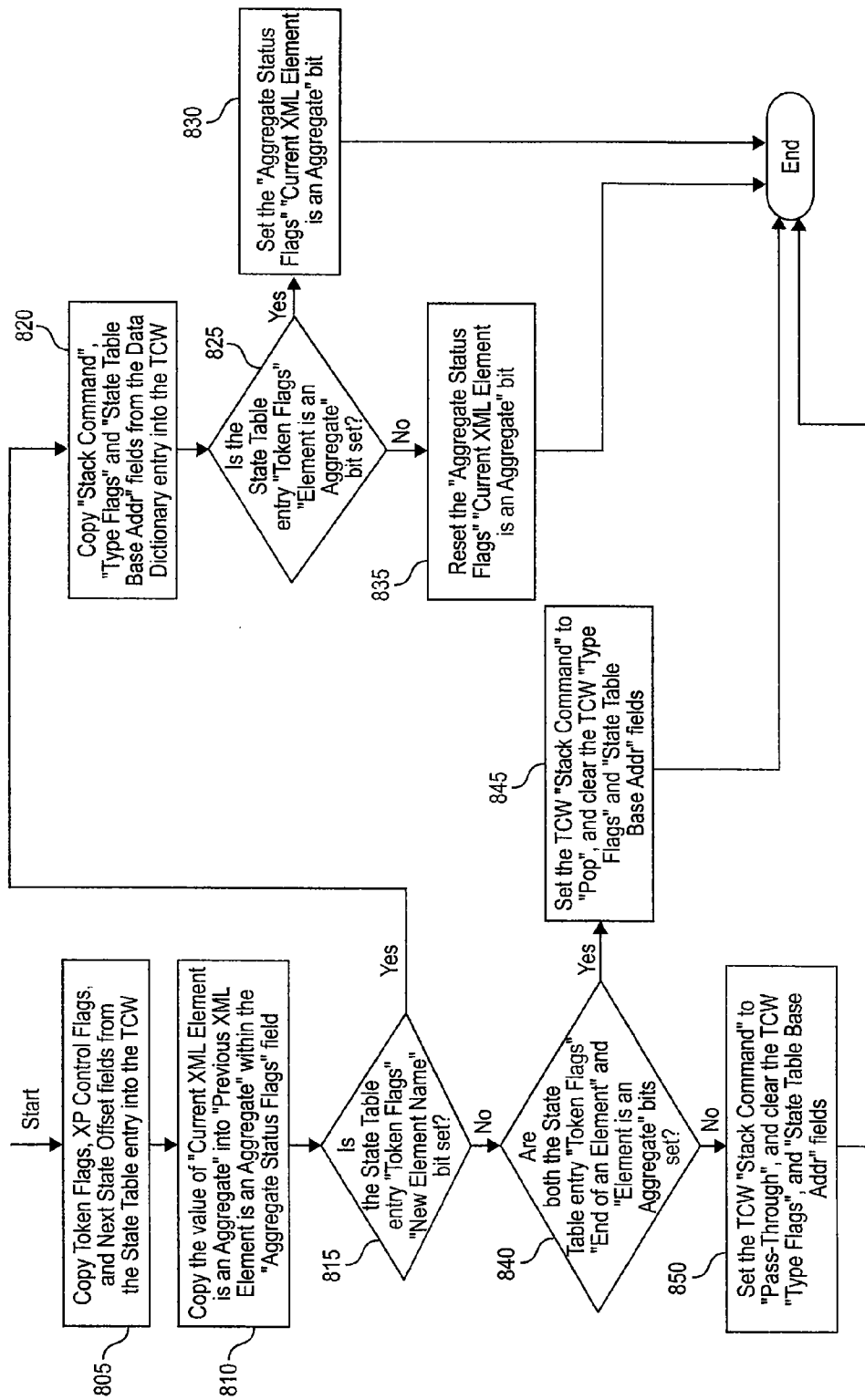


Figure 8

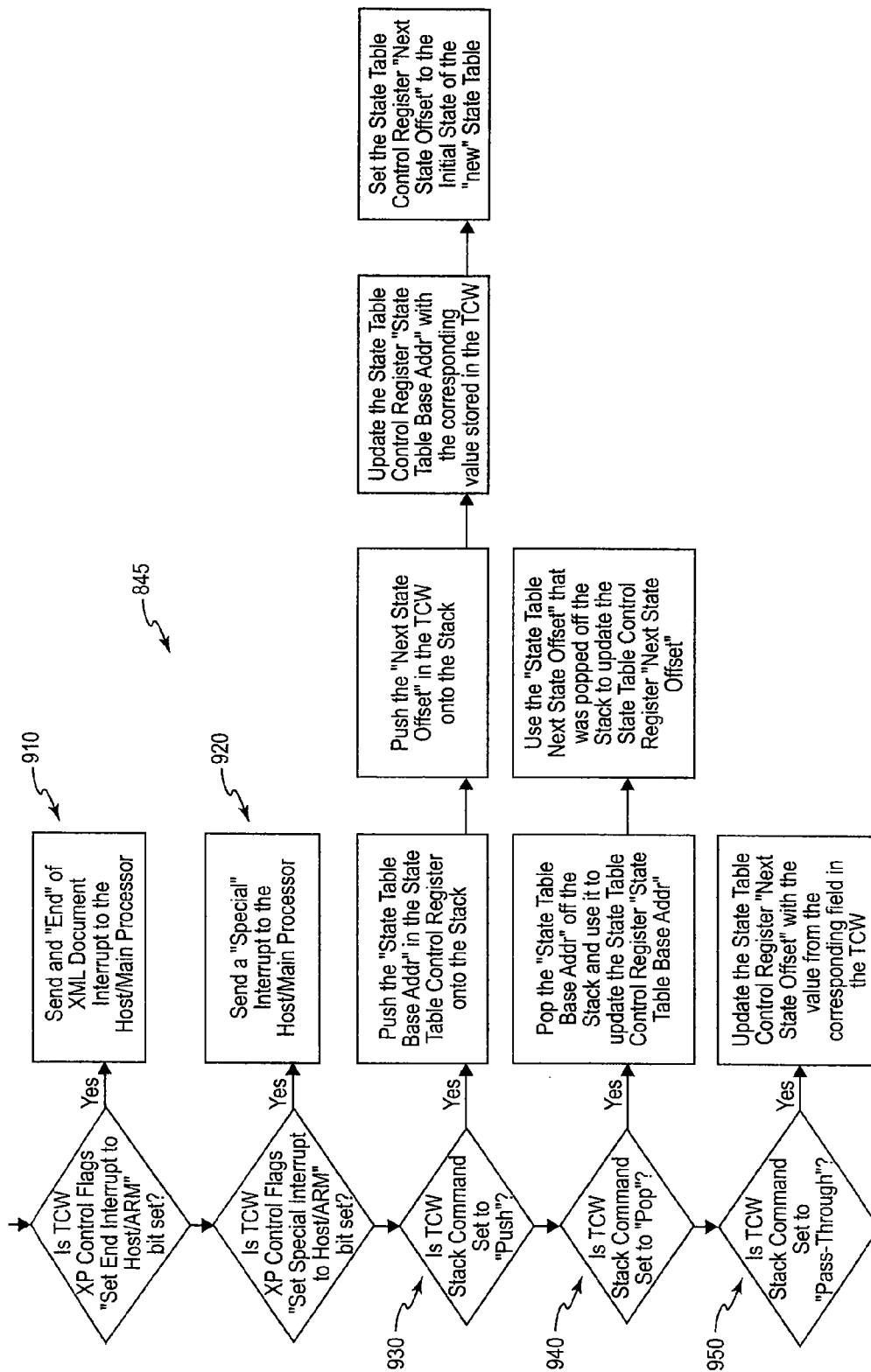


Figure 9

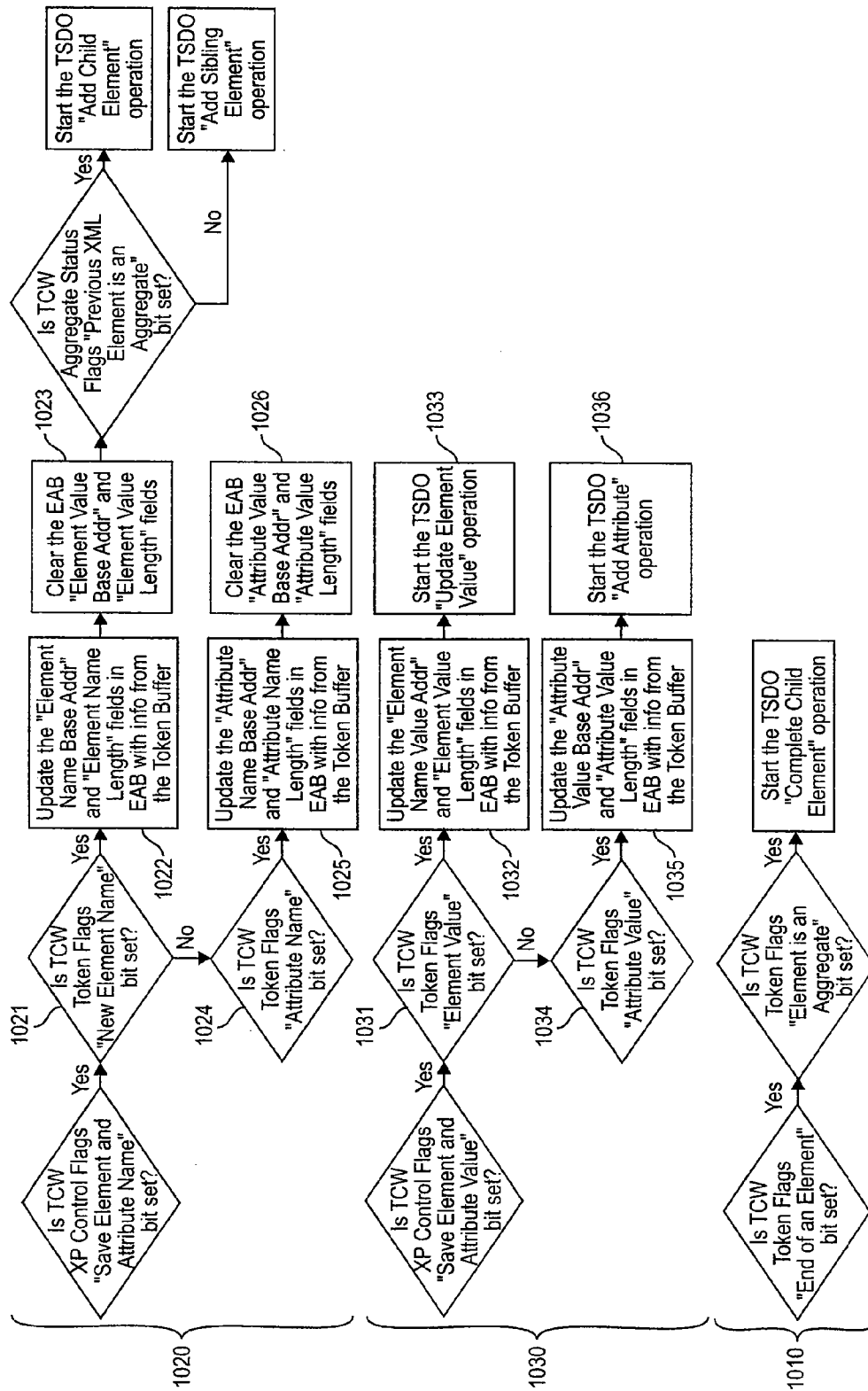


Figure 10

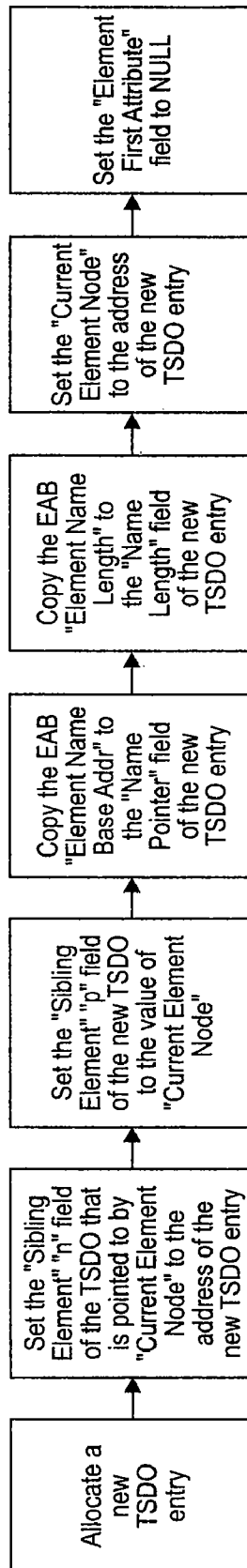


Figure 11A

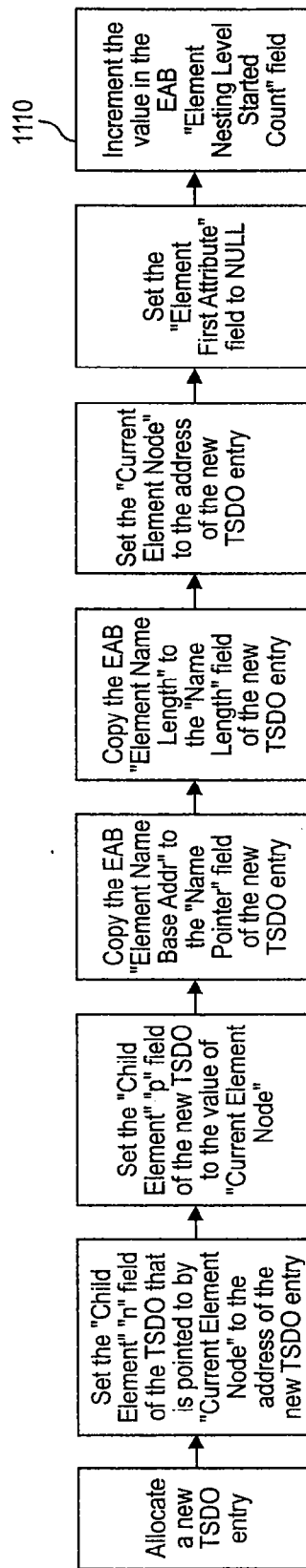


Figure 11B

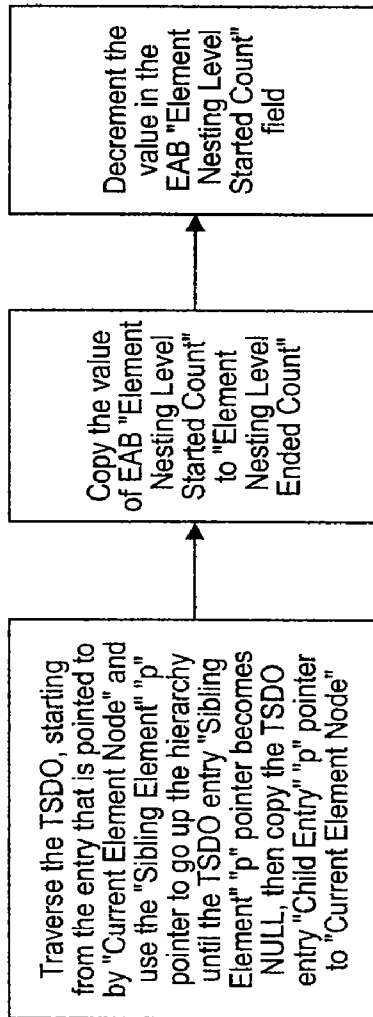


Figure 11C

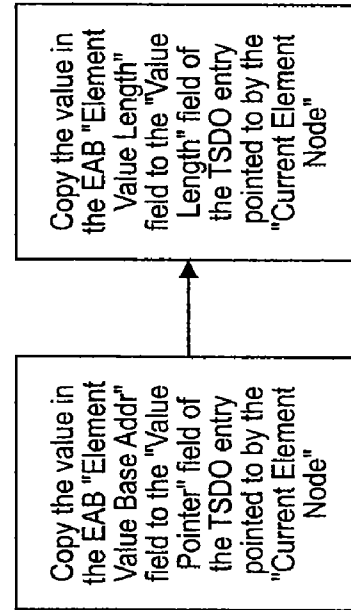


Figure 11D

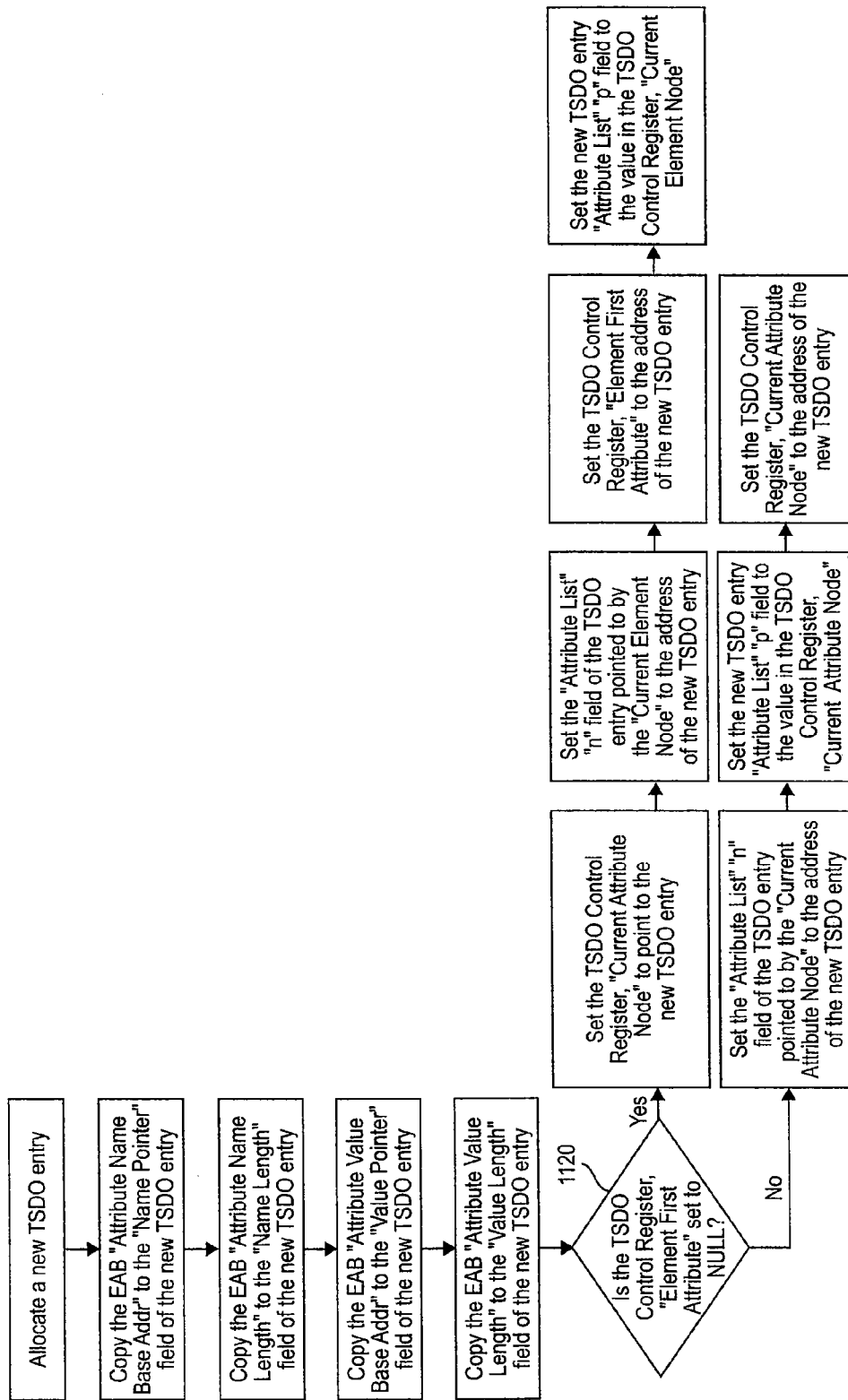


Figure 11E

HARDWARE ACCELERATED VALIDATING PARSER

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is a continuation of U.S. patent application Ser. No. 10/334,086, now U.S. Pat. No. 7,080,094, filed Dec. 21, 2002, which claims benefit of priority of U.S. Provisional Patent Application Ser. No. 60/421,774, filed Oct. 29, 2002, the entire disclosures of which are hereby fully incorporated by reference. Further, this application is related to U.S. patent application Ser. Nos. 10/331,879 and 10/331,315 (Docket numbers FS-00767 and FS-00766, corresponding to U.S. Provisional Patent applications 60/421,773 and 60/421,775, respectively) which are assigned to the assignee of this invention and are also fully incorporated by reference herein.

BACKGROUND OF THE INVENTION

[0002] 1. Field of the Invention

[0003] The present invention generally relates to validating parser processing for parsing and validating documents, such as XMLTM documents, for use in individual data processors interconnected by a network and, more particularly, to hardware validating processors for acceleration of the validation of such documents.

[0004] 2. Description of the Prior Art

[0005] The field of digital communications between computers and the linking of computers into networks has developed rapidly in recent years, similar, in many ways to the proliferation of personal computers of a few years earlier. This increase in interconnectivity and the possibility of remote processing has greatly increased the effective capability and functionality of individual computers in such networked systems. Nevertheless, the variety of uses of individual computers and systems, preferences of their users and the state of the art when computers are placed into service has resulted in a substantial degree of variety of capabilities and configurations of individual machines and their operating systems, collectively referred to as "platforms" which are generally incompatible with each other to some degree particularly at the level of operating system and programming language.

[0006] This incompatibility of platform characteristics and the simultaneous requirement for the capability of communication and remote processing and a sufficient degree of compatibility to support it has resulted in the development of object oriented programming (which accommodates the concept of assembling an application as well as data as a group of more or less generalized modules through a referencing system of entities, attributes and relationships) and a number of programming languages to embody it. Extensible Markup LanguageTM (XMLTM) is such a language which has come into widespread use and can be transmitted as a document over a network of arbitrary construction and architecture.

[0007] In such a language, certain character strings correspond to certain commands or identifications, including special characters and other important data (collectively referred to as control words) which allow data or operations to, in effect, identify themselves so that they may be,

thereafter treated as "objects" such that associated data and commands can be translated into the appropriate formats and commands of different applications in different languages in order to engender a degree of compatibility of respective connected platforms sufficient to support the desired processing at a given machine. The detection of these character strings is performed by an operation known as parsing, similar to the more conventional usage of resolving the syntax of an expression, such as a sentence, into its component parts and describing them grammatically.

[0008] When parsing an XMLTM document, a large portion and possibly a majority of the central processor unit (CPU) execution time is spent traversing the document searching for control words, special characters and other important data as defined for the particular XMLTM specification being processed. This is typically done by software which queries each character and determines if it belongs to the predefined set of strings of interest, for example, a set of character strings comprising the following "<command>", "<data type=dataword>", "</command>", etc. If any of the target strings are detected, a token is saved with a pointer to the location in the document for the start of the token and the length of the token. These tokens are accumulated until the entire document has been parsed.

[0009] This process must then be followed by processing in order to evaluate the tokens against rules and definitions contained in a "document model" such as the specification of a document type definition (DTD) or an XMLTM schema in order to assure that the collection of tokens and the character strings they represent in the document are well-constructed to form an unambiguous and internally consistent document, in its entirety. This processing is known as validation and generally proceeds in much the same fashion as processing for finding character strings of interest discussed above but operating on sixteen-bit (or longer) tokens corresponding to sequences of bytes rather than single eight-bit (or longer) bytes representing characters and checking for consistency between tokens and the content or arguments of other tokens to accommodate the self-definition characteristics and properties of languages such as XML, SGMLTM (of which XMLTM is a simplified form) and HTMLTM (which is essentially a special case of XMLTM) which support platform independence and interconnectivity.

[0010] Both the parsing for finding tokens and the parsing for validation are generally implemented using a conceptually table-based finite state machine (FSM) or state table to search for these strings of interest or consistency between elements found and represented by tokens. The state table resides in memory and is designed to search for the specific patterns of characters or tokens in the document. For parsing to find character strings of interest, the current state is used as the base address into the state table and the ASCII representation of the input character or the token is an index into the table. Character strings of interest may be of any of several types such as an element, an attribute/attribute list or data and elements may be simple elements or aggregates and may be nested. The parsing for validation principally looks at the types of character strings presented and the nesting itself to determine which elements or tokens are associated with another specific token(s) and the hierarchical relationship between them.

[0011] The goal of this processing is not only to determine that the document is a valid document that conforms to the

language (e.g. XMLTM) standard and have the correct structure as defined by a DTD or XMLTM schema in its entirety but to develop a hierarchical data structure such as a tree structured document object in which the structure will fully represent the informational content of the data. Therefore, while parsing to find character strings of interest is very time consuming and processor intensive, parsing for validation is much more so. That is, since the XMLTM data, for example, are textual and not only the data but the data structure, which may be freely specified to express the informational content, must be extracted from such text, it can be readily appreciated that the required processing is particularly time consuming and processor intensive.

[0012] At the same time, the potential complexity of the processing needed to properly handle aggregate elements and flexible nesting that may be in multiple hierarchical levels complicates the use of special purpose or hardware processors to reduce the processing load on the CPU of the local computer. That is, while it is generally recognized that special purpose or hardware processors can often provide increased processing speed in comparison with general purpose processors due to the reduced overhead for control of the general purpose processor, itself, it is not assured that a special purpose processor will be feasible or provide any significant advantage in performance as the processing function becomes more complex or with increased requirements for flexibility. In general, increased complexity and/or requirements for flexibility of function can only be accommodated by much increased hardware requirements which may not be economically justified for many applications or for the performance gain that may be possible. It is for this reason that validation parsing has been performed on programmed general purpose computers despite the processing time required.

SUMMARY OF THE INVENTION

[0013] The present invention to provide a hardware accelerator for validation processing in which substantial performance gains are derived with limited hardware.

[0014] In order to accomplish these and other objects of the invention, method of accelerating validation parsing and a hardware accelerated validation parser for a tokenized text document in a computer language supporting platform independence and interoperability is provided comprising, an arrangement for retrieving data from both a data dictionary and a state table in accordance with a token, and adder for merging the data from the state table and the data dictionary to form a transition control word, and adder for merging part of the transition control word with another token to retrieve further data from the state table, and logic for forming a tree structured data object corresponding to the tokenized text document under control of part of the transition control word.

BRIEF DESCRIPTION OF THE DRAWINGS

[0015] The foregoing and other objects, aspects and advantages will be better understood from the following detailed description of a preferred embodiment of the invention with reference to the drawings, in which:

[0016] FIG. 1 is a high level schematic diagram of the hardware validating parser accelerator in accordance with the invention,

[0017] FIG. 1A is a diagram of an exemplary tree structured document object illustrative of formation thereof,

[0018] FIG. 2 is a diagram of a preferred logical layout of state tables in accordance with the invention,

[0019] FIG. 3 is a diagram illustrating a preferred format of a token,

[0020] FIG. 4 is a diagram illustrating a preferred logical layout of the element and attribute buffer of FIG. 1 in accordance with the invention,

[0021] FIG. 5 is a diagram of a preferred exemplary data dictionary entry format,

[0022] FIG. 6 is a diagram illustrating a preferred logical layout of transition control words (TCW) in accordance with the invention,

[0023] FIG. 7 is a flow chart illustrating an overview of the operation of the validating parser accelerator illustrated schematically in FIG. 1,

[0024] FIG. 8 is a flow chart illustrating operation of the invention to implement TCW update rules, and

[0025] FIGS. 9, 10 and 11A-11E are flow charts illustrating TCW operations.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT OF THE INVENTION

[0026] Referring now to the drawings, and more particularly to FIG. 1, there is shown in high-level schematic form, an overview of the hardware validating parser accelerator system in accordance with the invention. It is to be understood that the invention operates on a document which has already been parsed for provision or setting of tokens corresponding to character strings of interest in an original document which is generally in textual form, as is typical of XMLTM documents. (While the invention will be described in connection with the preferred application to parsing of XMLTM documents, it is to be understood that the principles of the invention as will be evident from the following description are also applicable to parsing of files embodied in any programming language and, particularly, to object-oriented programming language with respect to which it is desirable to identify objects and their structure in accordance with fixed or user-definable rules.) This tokenized document is stored in memory/buffer 110 which is accessed in accordance with registers 112, 114 and 116 containing the document base address, the document limit address and the document next address, respectively.

[0027] Tokens are fetched in sequence and stored in token buffer 120. Portions of the token are then used to provide a portion of an address to adder 130 (which provides an address into state table 160), an address into the data dictionary 150 and to a namespace mapping memory (an incident of XMLTM to avoid problems which could be engendered by different developers using the same element names in regard to different documents which is well-understood by those skilled in the art and need not be discussed further) 140. In regard to the namespace mapping memory 140 which provides a base address into the data dictionary 150 through register 152, a default address is provided from register 142.

[0028] Considering the remainder of FIG. 1 at a very high level of abstraction, adder 170, transition control word (TCW) register 180, stack 190, adder 130 and state table 160 form a loop, generally indicated by arrow A which allows the state table to advance from one state to another. Stack 190 is basically for the purpose of interchanging order of tokens as they are considered in order to accommodate evaluation of nested tokens and to determine if the associations or relationships between parent and child elements is correct and well-constructed. Control of the stack is derived from a combination of information from the state table 160 and the data dictionary 150 which is derived in parallel based on the current token. Therefore, stack 190 may be considered as accommodating particular features (e.g. aggregates) of XML™ syntax and further discussion deferred until the detailed discussion below.

[0029] Thus, the gross function of loop A is to derive information from the state tables including a next state, add further data at adder 170, derive data, stack commands, interrupts and control signals for developing the final tree structured data object (TSDO) from the combined signals in the TCW register 180 and add the next state and/or the base address in registers 192 and 194, respectively to the next token. The extraction of a control signal for providing an addition of a node to the TSDO corresponds to a validation of the current token and a detected error will be represented by the issuance of an interrupt. The TCW is basically a combination of control flags and state transition controls from the state table 160 and data dictionary 150 with certain update and interrupt and stack rules applied.

[0030] The TCW register thus functions to merge and redirect control signals and data retrieved from among the contents of the state table 160 and the data dictionary 150 which represents the permissible syntax of tokens such that the effect of progress through the state table is controlled in a manner to detect errors and, while no error has yet been detected, to properly construct the TSDO 300 with data buffered at element and attribute buffer 200 in accordance with base and limit addresses supplied from registers 302, 304 respectively. This architecture supports standard and established types of document definitions being used as a data dictionary by virtue of accommodation by the state table contents which can be arranged at will to do so. The accelerator processing unit 100 is responsible for orchestrating the operations of the invention which will be described below and to construct the TSDO.

[0031] A host processor 400 which interfaces with the accelerator processing unit is also provided and used to control initiation of operation of the hardware accelerator in accordance with the invention. It will be appreciated from the following that the role of the host processor 400 in the validation parsing process is reduced, by virtue of the hardware accelerator of the invention, operations requiring only simple memory accesses and to provide interrupt servicing. In essence, substantially all validation processing overhead corresponding to the following of document syntax and element nesting and aggregate elements is removed to the validation parsing accelerator and the processor operations are limited to simple responses to processing calls, as needed, from the hardware accelerator 100. A support processor can also be provided to handle some or all of the processing calls from the hardware accelerator 100.

[0032] With the above overview as background, it will be helpful to an understanding of the hardware accelerated validating parser of the invention to discuss an exemplary TSDO 300 which is the goal of the validation parsing process performed by the invention with reference to FIG. 1A which illustrates formation of an exemplary TSDO. The TSDO is preferably constructed in memory as a doubly linked list data structure having seven elements (e.g. sibling element, child element, attribute list, name length, name pointer, value length and value pointer) per member/node as illustrated in the TSDO member layout. The eight rows of elements depicted, each corresponding to the TSDO member layout, thus correspond to eight individual members of the TSDO and are offset horizontally to indicate sibling and parent/child relationships.

[0033] To form a doubly linked list, several elements each contain two pointers p, n, to the previous and next member containing the previous and next element of the specified or same type. Thus a "next sibling element pointer" points to the next member equally indented with the current member. Similarly, pointers are provided to and from attribute list elements and the attributes of members. The "previous child element" points to the parent member. As shown in the third through fifth rows and second, sixth and seventh rows, previous and next pointers form chains among sibling members. The remaining elements are lengths and locations of actual data. From this information five controls are preferably provided in correspondence with a current member/node (e.g. the parent node, the current element node, the current attribute node, the element parent and the first attribute), as illustrated. These controls basically track whether the current member is an element or an attribute and the nodes on which they exist, the first attribute of the attribute list of an element, the immediate parent member and the root node. Following these controls allows traversing of the tree to locate any desired information while the nature of each piece of information is fully and unambiguously defined in the structure of the TSDO.

[0034] Similarly, it may be helpful in understanding the invention to review the characteristics of the data files operated upon. In the exemplary case of XML™, the elements and data will be embodied in a file which is essentially text but following rules which can be freely structured for different classes of documents. The rules embodied in the text document are, in effect, summarized in a file called "document model" which is used to validate the text document and which may be considered as separate from the text file although it need not be. The rules may be freely defined by a developer but some standardized sets of rules exist and which are often utilized as a matter of convenience.

[0035] The file defining the rules followed by an XML™ document or document in any other interoperable language, at the current time, is referred to as a document model which generally follows one of several forms although others are currently under development. A document model defines the elements which can appear within the document along with attributes that can be associated with a given element and also defines structural information about the document such as child-parent relationships of elements, the sequence in which the child elements can appear and the number of child elements as well as whether an element is empty or can include text as well as default values for attributes. Docu-

ment type definitions (DTDs) are a well-known example of a description of XMLTM document models.

[0036] The DTD language was developed specifically for defining validation rules for SGMLTM documents. As alluded to above, XMLTM is a simplified sub-set of SGMLTM and DTDs can also be used to define XMLTM validation rules. It should be recognized, however, that since the information required for validation of a particular document or class of documents must be the same regardless of the form in which it is transmitted or utilized, conversion between types of expressions of the validation information (e.g. DTDs and XMLTM schema) should, in theory, be substantially trivial and discussion in terms of DTDs should be equally applicable to any other form of the same information. By the same token, details of DTD syntax, which is very complex with heavy reliance on typographical symbols, is not of importance to the principles of the invention and need not be discussed in detail.

[0037] It should also be understood that XMLTM documents (and documents in other languages supporting platform independence and interoperability) principally provide a data structure and use of that data structure requires the ability to programmatically traverse the data structure to selectively access the data. A software module capable of reading XMLTM documents and providing access to their content and structure is referred to as an XMLTM processor or XML API which may also be freely implemented by developers although it is the common and recommended practice to use accepted, commercially available and industry standard APIs generally as a matter of being able to run under any compliant implementation of the same API.

[0038] There are currently two principal API specifications that may be currently regarded as industry standards: the Document Object Model (DOM) and the Simple API for XMLTM (SAX). The invention will be described hereinafter with reference to the DOM since it is more generalized and those skilled in the art will be enabled thereby to practice the invention using SAX. The DOM is based on an in-memory tree representation of the XMLTM document. When an XMLTM document is loaded into a processor, the processor must build an in-memory tree structure which properly represents the document. (Conversely, validation is essentially the construction of the tree structure which follows a properly constructed document.) The DOM also defines the programmatic interface (including the names of the methods and properties) that should be used to programmatically traverse an XMLTM tree and manipulate its elements, values and attributes. In other words, the TSDO data structure developed in the course of validation supports the DOM APIs or other similar APIs and implementations allowing use of the content of the document.

[0039] With the foregoing as background, it can be readily understood that processing for validation parsing of an XMLTM or other document in a language supporting interoperability using software on a general purpose computer can be extremely processor intensive and is slowed by the need for many memory accesses for multiple comparisons in regard to the content, structure and syntax specified for each element of a DTD or the like. The number of elements in a DTD or the like is theoretically unlimited and can run into the thousands in relatively simple data structures while the number of attributes child elements and sibling elements

may be as large as necessary and the data document may easily contain millions of instances of any given element or attribute. In other words, the very generality which must be accommodated in software on a general purpose processor imposes major processing complexities and burdens. In comparison, it will be seen from the following discussion that the hardware accelerated validation parser in accordance with the invention handles data in a relatively simple and consistent pipelined manner which can be performed at very high speed with a relatively small amount of hardware since the comparisons and the rules with which the comparisons are performed are embodied in the DTD embodied in the data dictionary and the signals present in the state table entries which can be rapidly merged and redirected to control parallel operations of the processor concurrent with the rapid sequential evaluation of tokens in loop A of FIG. 1.

[0040] Referring now to FIG. 2, The logical layout of the state table 160 is schematically shown. The state tables are built in accordance with a particular DTD or the like and accommodates all permissible elements. It should also be understood that the token effectively contains the next state since the next state is specified in respective state table entries (as shown in FIG. 3 and discussed below) which are, in part, indexed by the sixteen bit token values. As depicted in FIG. 1, the value in the token buffer 120 (or a portion of the value) is combined at adder 130 with the state table base address and the next state offset address to access an entry in the state table.

[0041] Specifically, it is preferred to provide the state table in sections indexed by the state table base address to reduce high speed storage requirements and then to index a column of the state table section using the token (which can be optimized through use of a palette mechanism) and to index rows by the next state offset. In such a case, the three portions of the address can be simply concatenated to form the full address as can be done, for example, by simply loading different portions of a register from the respective address portion sources.

[0042] The preferred format of the state table entries is illustrated in FIG. 3. Each state table entry is preferably of sixty-four bits in length and divided into two thirty-two bit words. Of course, other formats could be used as will be apparent to those skilled in the art.

[0043] The lower address word is, in turn, divided into a sixteen bit token value and two eight bit flag fields for the token flags and the control flags, respectively. (The tokens are preferably defined in a manner consistent with a hardware parser accelerator described in concurrently filed U.S. Provisional Patent Application 60/421,775 (Attorney's docket No FS-00766/02890051PR), hereby fully incorporated by reference and their form is otherwise not of particular importance to the practice of the invention in accordance with its basic principles. It should, however, be noted in this regard that the sixteen-bit token value field is somewhat redundant for the present invention since the token value is already being used for indexing the state table columns and thus could be designated as a reserved field.) The token flags are principally provided to track nesting, whether or not a given element is an aggregate, and the type of element represented by the token. Individual flags represented by each of the eight bits of the field are preferably:

Increment nesting, element is an aggregate, new element name, element value, attribute name, attribute value, end of element and decrement nesting, respectively. Similarly, the individual flags represented by individual bits of the control flag field are preferably: set end interrupt to host/main processor, set special interrupt to host/main processor, (Reserved), stop state table engine processing (these control flags are copied into the TCW as shown in FIGS. 9 and 10), save element or attribute name, save element or attribute value, character palette skip enable which is largely redundant but included to correspond to the hardware processor accelerator described in the above-incorporated application and could be a reserved field here), and end current token.

[0044] This data from the state table is merged with data from the data dictionary (corresponding to a DTD, XML™ schema or the like) at adder 170, preferably as a concatenation thereof and portions of the combined result are provided to the element and attribute buffer; the preferred logical layout of which is illustrated schematically in FIG. 4. The preferred format of the data dictionary entry is illustrated in FIG. 5.

[0045] The data dictionary is preferably structured as a hash table; the hash key being derived at 125 from the token. The preferred format of the data dictionary entry is one hundred twenty-eight bits in length. The entries are derived from the DTD, XML™ schema or the like describing the rules which are (to be) followed by the data document in XML™ or other language supporting interoperability. (An external component will translate the textual information within a DTD, XML™ schema or the like into the data dictionary format, as will be understood by those skilled in the art. The actual mechanics to perform the translation thus need not be further discussed.) Sixteen bits hold a token value corresponding to the current token from register 120 and are used to compare to the token value information derived from the state table as an internal check of the proper operation of validation parser, particularly for synchronism between the state table and the data dictionary. (That is, this comparison is not necessary in normal operation but may be valuable for testing or debugging.) Four bits are provided for the stack command flags; of which three are used (e.g. for push, pop and pass-through, respectively) and one reserved. The stack command flags indicate, for example, nesting and aggregation of elements required by a given element. Four further bits completing a byte are also reserved. Eight bits are provided for type flags. These bits are used to identify the data type (e.g. Boolean, binary, decimal, etc.) associated with the element. With eight bits, 256 different data types can be identified using the type flags field. It should be noted that the type flags field is not necessary to the operation of the invention in accordance with its basic principle but allows for upgrading to allow validation of value to match against the element data type. These fields are followed by three thirty-two bit words providing the state table base address, and two pointers to the data pattern and range reference and attribute rule reference, respectively. Of the above fields, the stack command flags, the type flags and the state table base address associated with the current token are directed to the TCW register 180 while the remainder are used for comparisons with the token and/or the associated test of the (e.g. XML™) document, preferably in special purpose logic circuits which may be free-running and which respond very quickly.

[0046] The preferred logical layout of the transition control word (TCW) and the register 180 provided to buffer it is illustrated in FIG. 6. Use of a buffer register is, in theory, not necessary but is considered desirable and thus preferable as a simple and inexpensive expedient to assure synchronism as data and control signals are redirected, as alluded to above. It is important to an understanding of the invention to observe the sources of signals applied to the TCW and the respective portions of the validating parser architecture of FIG. 1 to which they are directed. Details of the source/sink for each field is shown in FIGS. 8-10.

[0047] The preferred logical layout of the TCW as shown in FIG. 6 comprises three thirty-two bit words including the state table base address received from the data dictionary and forwarded to register 194 and the next state offset received from the state table and forwarded to register 192. The remaining thirty-two bits include a four bit stack command field received from the data dictionary and used to control operation of the stack 190, a four bit aggregate status flag field (of which only two bits are preferably used to indicate whether the current token and/or the previous token are aggregates since aggregates can include elements at different levels of a tree structure), a eight-bit type flag field received from the data dictionary (the derivation being shown in FIG. 8 and their usage shown in FIG. 10, as noted above), and two eight-bit fields for the token flags and control flags received from the state table and used to control EAB and TSDO operations as shown in FIG. 10. The data dictionary token value field is not carried over from either the data dictionary or the state table since (assuming these values compare favorably) the token value is readily available in the token buffer.

[0048] It should be noted that the complete definition of a token is shown in FIG. 3A. The hash value of a token text string associated with certain tokens (with token flags set to new element names) is used to index the data dictionary. The token value is a special numeric value assigned to the token and is used to look up the columns of the state table. Sometimes it is a generic value indicating that the token is representing a character string literal or an integer value. Other times it is an encoded number of an element name or a tag name.

[0049] Referring again to FIG. 4, the element and attribute buffer (EAB) will now be discussed in greater detail. It will be appreciated by those skilled in the art that the above described portions of the architecture will have made data available from which all fields of the EAB can be filled by an operation essentially comprising a memory access and a comparison which is simple, straightforward and rapidly executed by the accelerator processing unit while syntax, nesting and aggregate tracking and operations are handled entirely by the finite state machine and the associated accelerator registers in accordance with the invention.

[0050] Specifically, a given parsed element can be any one of an element type, and attribute type or a value type; each of which must be treated differently in the TSDO 300. The EAB collects and holds structure-specifying pointers to point to attributes or data around a specific node (not yet placed in the TSDO) to develop, when complete, the doubly linked structure TSDO as alluded to above with reference to FIG. 1A. With reference to specific fields of the EAB, element nesting starting and ending counts may be counted/

accumulated directly from the token flags (e.g. first and eighth bits) discussed above in connection with FIG. 3 while the name base addresses and lengths for elements and attributes and their values corresponding to each token are readily available from the token buffer 120. The parent/child and aggregate relationship and type and control information for the current node of the TSDO 300 is directly available from the TCW as illustrated in FIG. 6, allowing a new node to be added to the TSDO as each element/token of the tokenized document is validated.

[0051] Referring now to FIG. 7, the overall operation and function of the hardware validation parsing accelerator 100 in accordance with the invention will be summarized. The operation of the hardware accelerator is started by initialization through loading the state table 160 and data dictionary 150 with data (705, 710) corresponding to the document to be parsed and validated in a format such as the preferred format discussed above. The tokenized document to be validated is then loaded (715) into memory 110, the stack control registers and state table base address are initialized (720, 725) and the next state offset is set to the initial state (730).

[0052] Processing of the tokenized document begins with extracting (735) the first (or next) token into the token buffer 120. The token is hashed into a hash key which is used for data dictionary look up operations. A look-up operation 740 is then performed in memory 140 to update the data dictionary base address with which a look-up operation 745 can be performed in the data dictionary 150 corresponding to the data dictionary base address. The state table may be concurrently accessed 750 using the new token and current contents of the state table registers 192 and 194 provided through adder 130. Next, through adder 170, the TCW register is updated 755 with the data derived from the data dictionary 150 and the state table 160 in accordance with rules which will be discussed in detail below in connection with FIGS. 8 and 9. The EAB 200 is then updated 760 according to rules based on the flag settings in the TCW. An interrupt can concurrently be sent to host/main processor if the interrupt flag is set in the TCW in accordance with a control flag contained the state table entry. If an interrupt is not issued, the information gathered in the EAB is added 765 to the TSDO in accordance with rules based on token and control flag settings, checked at 770 for issuing an interrupt, in the TCW which are also derived from the accessed state table entry. Concurrently, a push, pop or pass-through operation 775 is performed in stack 190 to support multi-level aggregates (e.g. XMLTM elements which are made up of other XMLTM elements). Unless data is pushed onto the stack, new base address and next state offset data will be output from the stack and registers 192 and 194 are updated 780. The hardware accelerator has now, by the above processes, validated a token and added a node to the TSDO and is now reset (for proper evaluation of syntax, nesting and aggregate elements at registers 192, 194) of the next token through repetition of the above processes.

[0053] The TCW update rules are quite simple and are illustrated in FIG. 8. First, the token flags, control flags and next state offset fields are copied 805 to the TCW register/buffer 180 from the state table entry. Then, within the aggregate status flags field, the "current element is an aggregate" flag is copied 810 into the "previous element is an aggregate" flag. If (815) the "new element name" flag is

set in the token flags field, the "stack command", "type flags" and "state table base address" fields are copied 820 from the data dictionary output into the TCW and, if (825) the "element is an aggregate" in the token flags field is set, the "current element is an aggregate" flag bit in the "aggregate status flags" field is set 830. Otherwise, the "current element is an aggregate" flag is reset 835. If the "new element name" flag in the token flags field is not set, and the "end of an element" flag bit and the "element is an aggregate" flag bit in the token flags field are set, as determined at 840, the stack command field "pop" flag bit is set and the type flags and state table base address fields are cleared 845, as shown in FIG. 9. Otherwise, the stack command field "pass-through" flag bit is set and the type flags and state table base address fields are cleared 850.

[0054] As illustrated in FIG. 9, for issuance of interrupts, the control flag bits derived directly from the state machine output for entry corresponding to the current token are simply followed (as indicated by dashed arrows) or tested in parallel to provide an end interrupt or special interrupt to the host/main processor as indicated at 910 or 920. Similarly, if the stack command (derived directly from the data dictionary 150 output) is "push", the state table base address and the next state offset are pushed onto the stack 190 and the state table base address register 192 and next state offset register 194 are updated from the corresponding TCW fields as indicated at 930. If the stack command flag corresponds to a "pop" command, the state table base address and next state offset values are popped off of the stack and used to update registers 192, 194 respectively, as indicated at 940. If the "pass-through" stack command field flag is set, registers 192, 194 are updated from the respective TCW fields with no operation being performed on the stack 190, as indicated at 950.

[0055] Similarly, the operations in the EAB and TSDO are also preferably controlled by flags in the TCW as illustrated in FIG. 10. If the "end of an element" token flag is set and the "element is an aggregate" token flag is also set the TSDO operation to complete a child element, as will be discussed below, is triggered, as shown at 1010. Otherwise, the "save element of attribute name" flag is used in operations 1020 and 1030 (and will be assumed to be set in the following discussion) in combination with other flags to trigger appropriate operations based on the element type as may be reflected in those flags. If (1021) the "new element name" flag is additionally set the element name base address and element name length fields in the EAB are updated 1022 from the token buffer 120 and the element value base address and length fields are cleared 1023 in the EAB. If (1024) the "attribute name" flag is set, the attribute base address and length fields are reset 1025 in the EAB from the token buffer 120 and the attribute value base address and length fields are cleared 1026 in the EAB. If (1031) the "element value" flag is set, the element value base address and length fields are updated 1032, 1033 from the token buffer. If (1034) the "attribute value" flag is set, the attribute value base address and length fields are updated 1035, 1036 from the token buffer 120. In summary, chosen fields in EAB 200 are updated and cleared based on element type.

[0056] If the "save element or attribute name and the "new element name" are both set, as described above, and the "previous element is an aggregate" flag in the aggregate status flags field of the TCW is also set, the additional

operation of starting the “add child element” operation, as will be described below, is triggered and the EAB “element nesting start count” field is incremented. Otherwise, if the “previous element is an aggregate” flag is not set, the “add sibling element” operation in the TSDO is triggered.

[0057] If, on the other hand, the combination of “save element or attribute” and “element value” flags are set, the “update element value” operation, described below, is triggered in the TSDO. Similarly, the combination of “save element or attribute” and “attribute value” flags trigger the TSDO to start the “add attribute” operation.

[0058] The five TSDO operations alluded to above may be performed autonomously under control of the accelerator processing unit 400 once initiated, as described above, by the hardware validation parser accelerator in accordance with the invention. Such processing may be performed concurrently with the validation parsing operation, further supporting the acceleration of processing by the invention. All of these operations are very simple, short and straightforward and thus may be executed quickly with little, if any, host processor burden.

[0059] The “add sibling element” operation illustrated in FIG. 11A comprises allocation of a new TSDO entry, setting the “next” pointer of the current node to the newly allocated entry address, setting the “previous” pointer of the new entry to the current node address, copying the EAB element name base address and length fields to the corresponding fields (see FIG. 1A) in the new entry, and setting the “current element node” and the “element first attribute” of the TSDO controls (FIG. 3) to the new entry and null, respectively. The operation of “add child element” illustrated in FIG. 11B is identical to the “add sibling element” operation except for performing the additional step 1110 of incrementing the EAB “nesting level start count” field.

[0060] The “complete child element” operation illustrated in FIG. 11C is performed by traversing the TSDO structure formed to that point starting with the entry pointed to by “current element node” TSDO control field and using the sibling element “p” pointer to move up the tree hierarchy until a sibling element “p” pointer becomes null. Then copy the pointer in the “child entry” “p” pointer of that TSDO entry to the “current element node” TSDO control field. Then the “nesting entry start count” TSDO control field is copied to the “nesting level end count” TSDO control field and the “nesting entry start count” TSDO control field is decremented. The “update element value” TSDO operation illustrated in FIG. 11D involves only the copying of the element value base address and length EAB fields to the “value” pointer and “value length” fields of the TSDO entry pointed to by the “current element node” TSDO control field.

[0061] The “add attribute” TSDO operation begins with allocation of a new TSDO entry and copying the attribute name base address and length and attribute value base address and length pointers to the new entry. Then, if (1120) the “element first attribute” TSDO control register is null, the “current attribute node” TSDO control is set to point to the new TSDO entry and the “n” and “p” pointers of the current node and the new entry are linked as discussed above while “element first attribute” of the current node is set to the new entry (e.g. before the “p” pointer is added to the new entry). If the element first attribute is not null, the “n” and

“p” pointers of the current node and the new entry are linked and the “current attribute node” TSDO control is set to the new TSDO entry.

[0062] In view of the foregoing, it is seen that the invention provides an apparatus and method to provide extremely rapid validation parsing of an XML™ or other language document supporting interoperability while removing such processing operations and complex supporting overhead from a host processor; resulting in substantial acceleration of the validation parsing process. The acceleration is particularly supported by the potentially and preferably autonomous operation of forming a TSDO in parallel with the parsing for validation and parallel retrieval of data dictionary and state table information. The hardware required to provide such acceleration is very simple and very limited in quantity and, hence, inexpensive and highly cost-effective.

[0063] While the invention has been described in terms of a single preferred embodiment, those skilled in the art will recognize that the invention can be practiced with modification within the spirit and scope of the appended claims.

We claim:

1. A validation parser for a tokenized text document in a computer language supporting platform independence and interoperability, comprising:

a first memory element adapted to receive data from both a data dictionary and a state table in accordance with a token;

a first combiner adapted to merge said data from said state table and said data dictionary to form a transition control word;

a second combiner adapted to merge part of said transition control word with another token to retrieve further data from said state table; and

a logic element adapted to form a tree structured data object corresponding to said tokenized text document under control of part of said transition control word.

2. The validation parser of claim 1, further comprising:

a stack controller adapted to use said transition control word to derive the next transition state in supporting nested data structures defined in a supported language.

3. The validation parser of claim 2, further comprising:

a second memory element adapted to form a data construct from the tokenized text document.

4. The validation parser of claim 3, wherein said second memory element comprises an element and attribute buffer.

5. The validation parser of claim 1, further comprising:

a structured document object adapted to form a data construct from the tokenized text document.

6. The validation parser of claim 5, wherein said structured document object is adapted to receive data from an element and attribute buffer.

7. The validation parser of claim 1, wherein the tokenized text document is formatted in accordance with the extensible mark-up language.

8. A computer readable medium upon which is embedded a sequence of programmed instructions which, when executed by a processor, cause the processor to perform steps comprising:

retrieving data from both a data dictionary and a state table in accordance with a token;

merging said data from said state table and said data dictionary to form a transition control word; and

merging part of said transition control word with another token to retrieve further data from said state table.

9. The computer readable medium of claim 8, in which the instructions further comprise the step of:

operating a stack with said transition control word to derive the next transition state.

10. The computer readable medium of claim 8, in which the instructions further comprise the step of:

verifying that an input stream conforms to one of a set of valid input sequences.

11. The computer readable medium of claim 10, in which the instructions further comprise the step of:

producing a notification when an input stream deviates from a set of valid, allowable input sequences.

12. The computer readable medium of claim 8, in which the instructions further comprise the step of:

producing a notification when an input stream deviates from a set of valid, allowable input sequences.

13. A validation parser for a tokenized text document in a computer language supporting platform independence and interoperability, comprising:

a register adapted to receive data from both a data dictionary and a state table in accordance with a token;

a first adder adapted to merge said data from said state table and said data dictionary to form a transition control word;

a second adder adapted to merge part of said transition control word with another token to retrieve further data from said state table; and

a logic element adapted to form a tree structured data object corresponding to said tokenized text document under control of part of said transition control word.

14. The validation parser of claim 13, further comprising:

a stack controller adapted to use said transition control word to derive the next transition state in supporting nested data structures defined in a supported language.

15. The validation parser of claim 14, further comprising:

a buffer adapted to form a data construct from the tokenized text document.

16. The validation parser of claim 15, wherein said buffer comprises an element and attribute buffer.

17. The validation parser of claim 13, further comprising:

a structured document object adapted to form a data construct from the tokenized text document.

18. The validation parser of claim 17, wherein said structured document object is adapted to receive data from an element and attribute buffer.

19. The validation parser of claim 13, wherein the tokenized text document is formatted in accordance with the extensible mark-up language.

* * * * *