



US007788402B1

(12) **United States Patent**
Keller et al.

(10) **Patent No.:** **US 7,788,402 B1**
(45) **Date of Patent:** **Aug. 31, 2010**

(54) **CIRCUIT FOR MODIFICATION OF A
NETWORK PACKET BY INSERTION OR
REMOVAL OF A DATA SEGMENT**

(75) Inventors: **Eric R. Keller**, Boulder, CO (US);
Graham F. Schelle, Boulder, CO (US);
Philip B. James-Roxby, Longmont, CO
(US)

(73) Assignee: **Xilinx, Inc.**, San Jose, CA (US)

(*) Notice: Subject to any disclaimer, the term of this
patent is extended or adjusted under 35
U.S.C. 154(b) by 503 days.

(21) Appl. No.: **11/799,898**

(22) Filed: **May 3, 2007**

(51) **Int. Cl.**
G06F 15/16 (2006.01)
H04L 12/56 (2006.01)
H04L 12/28 (2006.01)
H04J 3/12 (2006.01)

(52) **U.S. Cl.** **709/238**; 370/392; 370/401;
370/528; 709/200

(58) **Field of Classification Search** 709/200,
709/238; 370/392, 401, 528
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,500,796	A	3/1996	Black, Jr.	
6,477,683	B1 *	11/2002	Killian et al.	716/1
6,665,725	B1	12/2003	Dietz et al.	
6,687,247	B1 *	2/2004	Wilford et al.	370/392
6,771,646	B1 *	8/2004	Sarkissian et al.	370/392
6,794,896	B1	9/2004	Brebner	
7,047,303	B2	5/2006	Lingafelt et al.	
7,398,502	B1	7/2008	Kulkarni et al.	

7,440,457	B2	10/2008	Okuno
7,552,042	B1	6/2009	Brebner et al.
2002/0067757	A1	6/2002	Philips et al.
2002/0080771	A1	6/2002	Krumel
2002/0176378	A1	11/2002	Hamilton et al.
2003/0177187	A1	9/2003	Levine et al.
2005/0226238	A1	10/2005	Hoskote et al.
2008/0002576	A1	1/2008	Bugenhagen et al.
2008/0002670	A1	1/2008	Bugenhagen et al.

(Continued)

OTHER PUBLICATIONS

U.S. Appl. No. 11/799,897 by Brebner et al., filed May 3, 2007 by
Xilinx, Inc.

(Continued)

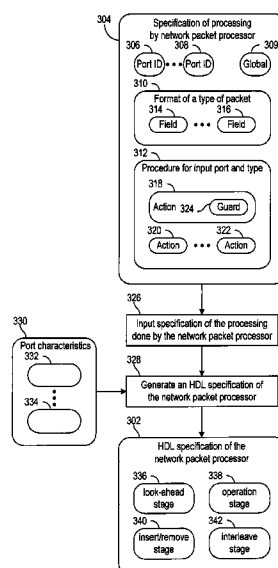
Primary Examiner—Saleh Najjar
Assistant Examiner—Vitali Korobov

(74) *Attorney, Agent, or Firm*—LeRoy D. Maunu; James
McGee; Lois D. Cartier

(57) **ABSTRACT**

A state machine circuit converts a first network packet into a second network packet according to modification actions from a textual language specification. Each modification action is either an insertion action inserting a data segment or a removal action removing a data segment. Each state corresponds to a pairing of a first data word from the first packet and a second data word from the second packet. Each state selects the data units of the second data word from the data segment of each insertion action and the data units of both the first and a prior data word. Each state specifies one or more next states including the state corresponding to the pairing of either the first or a next data word after the first data word in the first sequence and either the second or a next data word after the second data word in the second sequence.

19 Claims, 8 Drawing Sheets



U.S. PATENT DOCUMENTS

2008/0002676	A1	1/2008	Wiley et al.	
2008/0002716	A1	1/2008	Wiley et al.	
2008/0005156	A1 *	1/2008	Edwards et al.	707/102
2008/0049626	A1	2/2008	Bugenhagen et al.	
2008/0049628	A1 *	2/2008	Bugenhagen	370/244
2008/0049630	A1	2/2008	Kozisek et al.	
2008/0049769	A1 *	2/2008	Bugenhagen	370/401
2008/0049927	A1	2/2008	Wiley et al.	
2008/0052387	A1	2/2008	Heinz et al.	
2008/0052393	A1	2/2008	McNaughton et al.	
2008/0052394	A1 *	2/2008	Bugenhagen et al.	709/224
2008/0133518	A1 *	6/2008	Kapoor et al.	707/6
2008/0162390	A1 *	7/2008	Kapoor et al.	706/20

OTHER PUBLICATIONS

U.S. Appl. No. 11/799,953 by James-Roxby et al., filed May 3, 2007 by Xilinx, Inc.
 U.S. Appl. No. 11/799,860 by Keller et al., filed May 3, 2007 by Xilinx, Inc.
 U.S. Appl. No. 11/799,966 by Keller et al., filed May 3, 2007 by Xilinx, Inc.

U.S. Appl. No. 10/769,330, filed Jan. 30, 2004, James-Roxby et al.
 U.S. Appl. No. 11/818,722, filed Jun. 14, 2007, Brebner et al.
 U.S. Appl. No. 11/818,788, filed Jun. 14, 2007, Attig et al.
 U.S. Appl. No. 11/818,792, filed Jun. 14, 2007, Brebner.
 U.S. Appl. No. 11/818,811, filed Jun. 14, 2007, Attig et al.
 Attig, Michael et al., "Systemic Characterization of Programmable Packet Processing Pipelines," *Proc. of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2006, pp. 195-204, IEEE, Piscataway, New Jersey, USA.
 Sun, Hairong et al., "A Performance Model of Partial Packet Discard and Early Packet Discard Schemes in ATM Switches," Mar. 14, 2000, pp. 1540-1553, *Computer Communications*, vol. 24 (2001).
 Kountouris, Apostolos et al., "A Method for the Generation of HDL Code at the RTL Level from High-Level Formal Specification Language," *Proc. of the 40th Midwest Symposium on Circuits & Systems*, Aug. 3, 1997, pp. 1095-1098, IEEE, Piscataway, New Jersey, USA.
 Shiro, Takeshi et al., "A Processor Generation Method from Instruction Behavior Description Based on Specification of Pipeline Stages and Functional Units," *Proc. of the 2007 Asia and South Pacific Design Automation Conference*, Jan. 23, 2007, pp. 286-291, IEEE Computer Society, Washington, DC, USA.

* cited by examiner

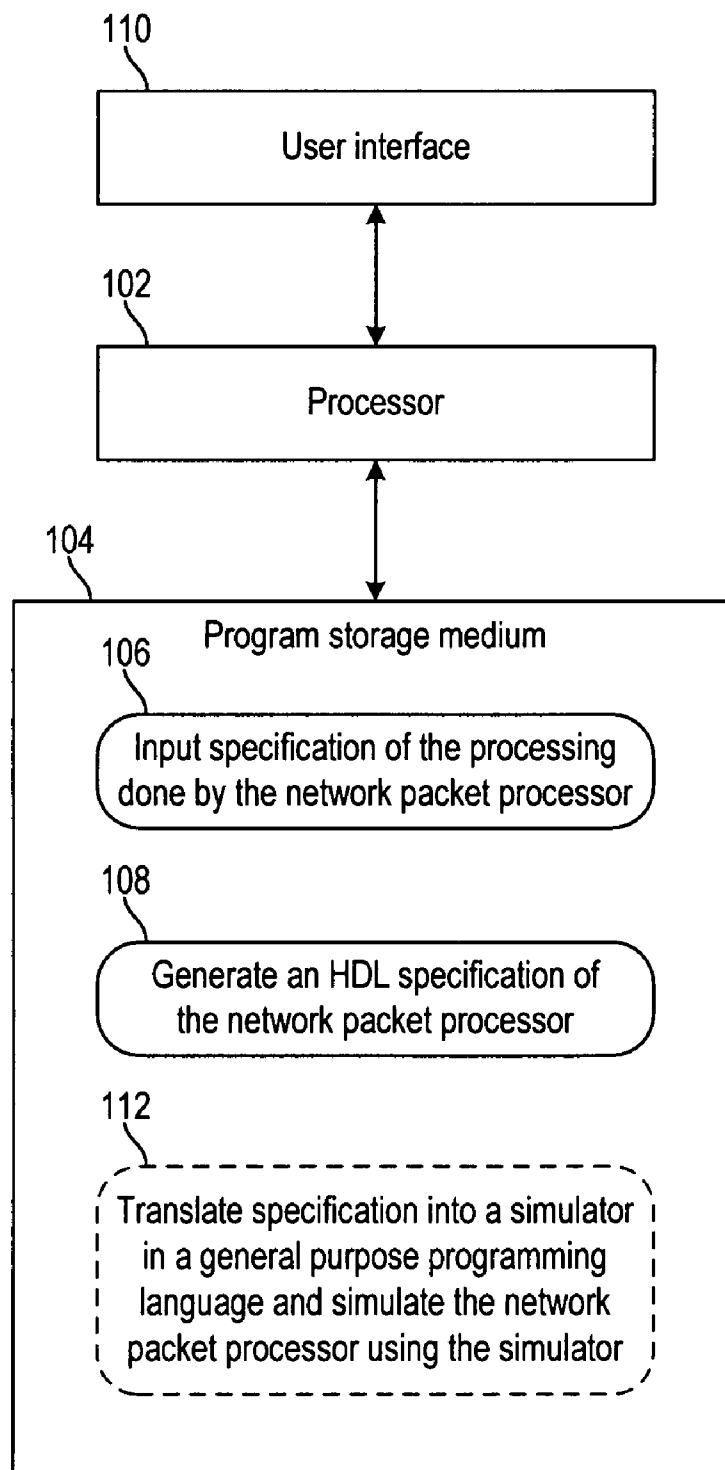


FIG. 1

```

202 { element IPOptionizer {
    // Ports
204 { input packetin : Packet;
      output packetout : Packet;
      output memoryreq : MemoryAccess;
      // Ethernet MAC + LLC + SNAP header
      format header = ( pad : 16,
                        source : 48,
                        destination : 48,
                        length : 16,
                        LLC : 24,
                        SNAP : 40
                        );
      // IP encapsulated in Ethernet packet
      format IPpacket = ( Ethernet : header,
                          version : 4,
                          hlen : 4,
                          TOS : 8,
                          length : 16,
                          id : 16,
206 { flags : 3,
        offset : 13,
        TTL : 8,
        protocol : 8,
        checksum : 16,
        source : 32,
        destination : 32,
        payload : *
        );
      // Memory data format – some sort of IP options field
      format optionfield = ( flag : 1,
                             class : 2,
                             number : 5,
                             length : 16,
                             data : 104
                             );
207 { // Global variable (initialization not shown)
      var default : 8;
      // Packet processing example
      handle IPpacket on packetin {
210 { var memorydata : optionfield;
212 { set TTL = TTL - 1;
213 { set checksum = default;
214 { read memorydata from memoryreq [ id ];
216 { [ version == 4 ] insert memorydata after destination;
218 { forward on packetout;
      }
    }
  }
}

```

FIG. 2

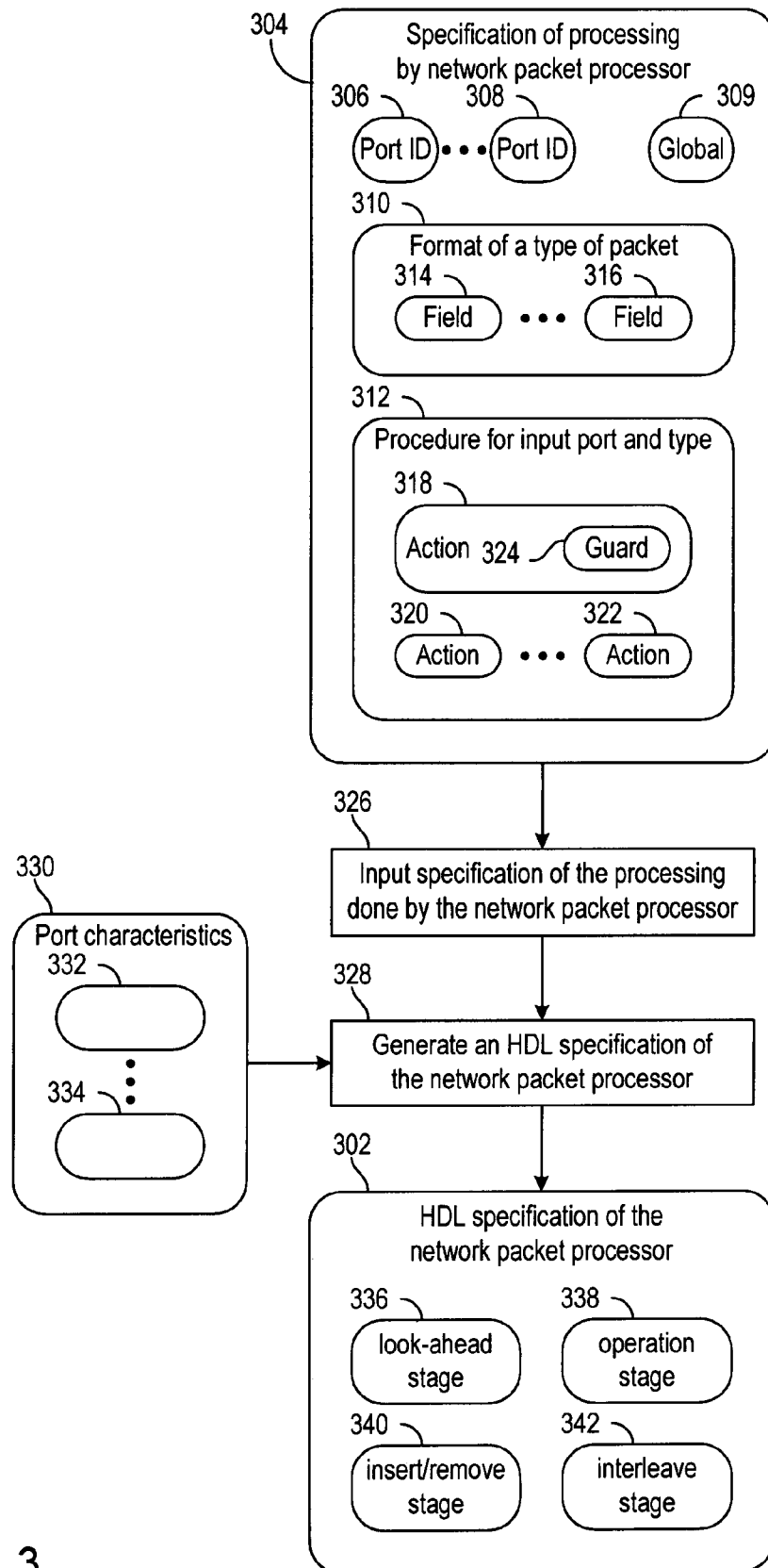


FIG. 3

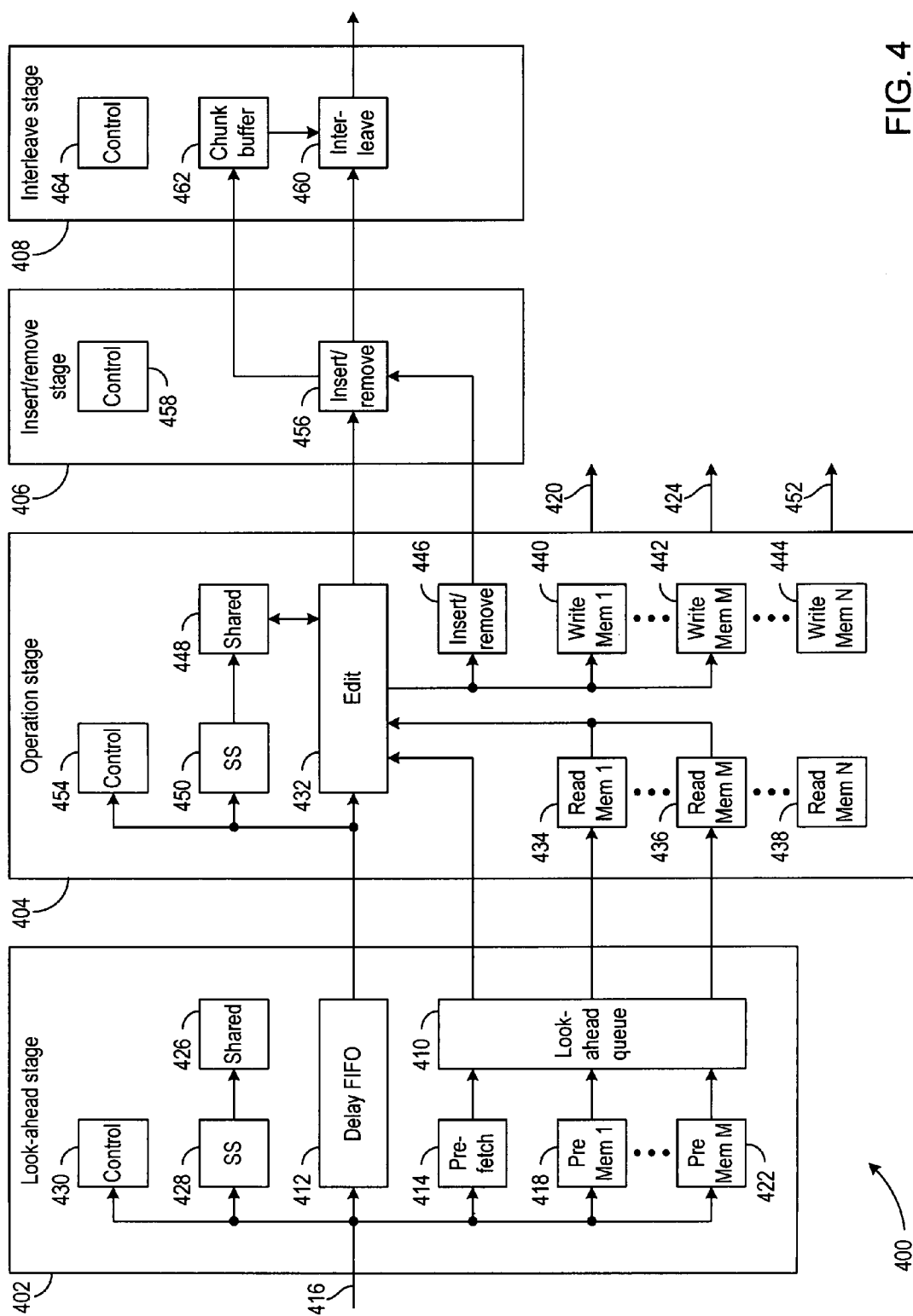


FIG. 4

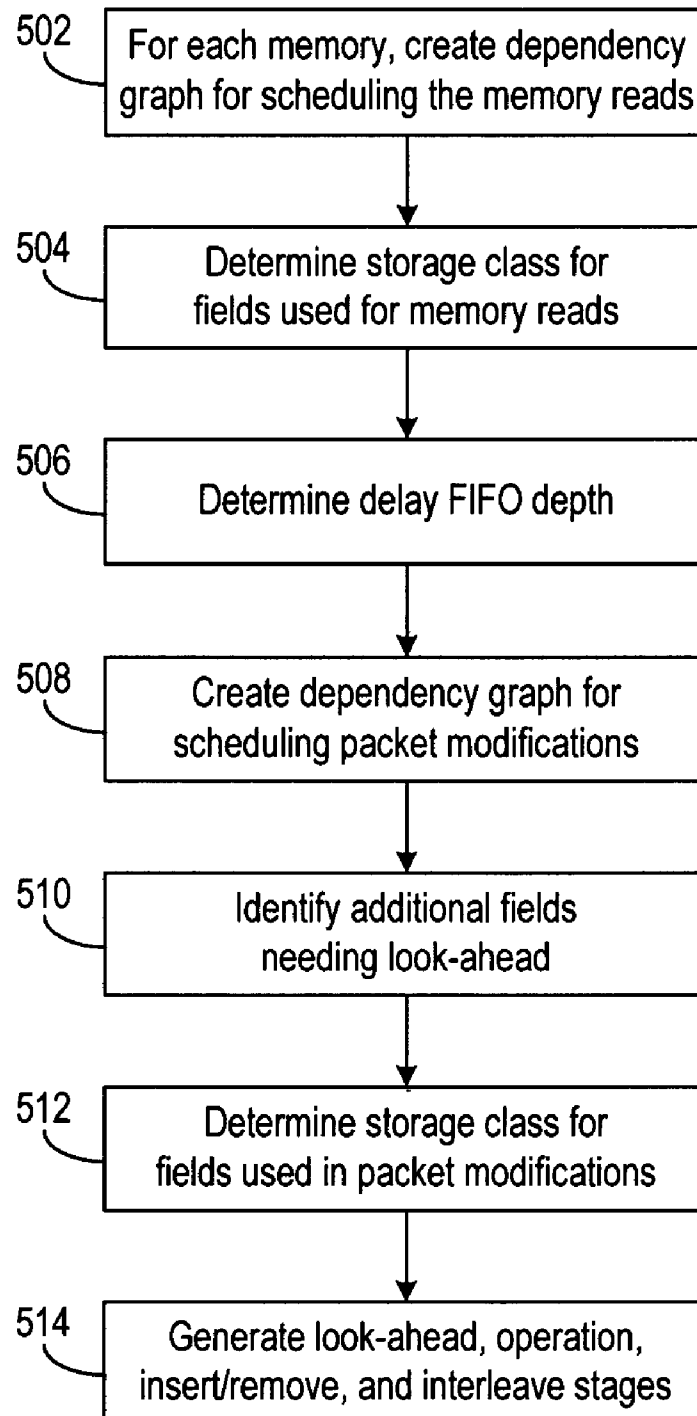


FIG. 5

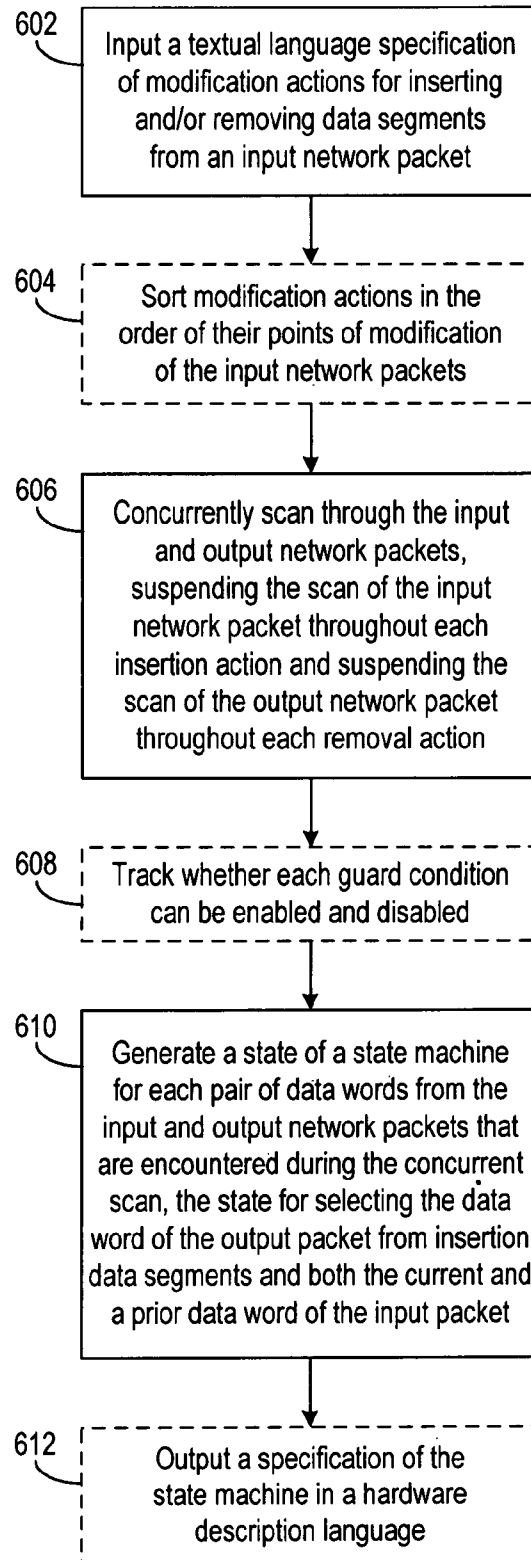


FIG. 6

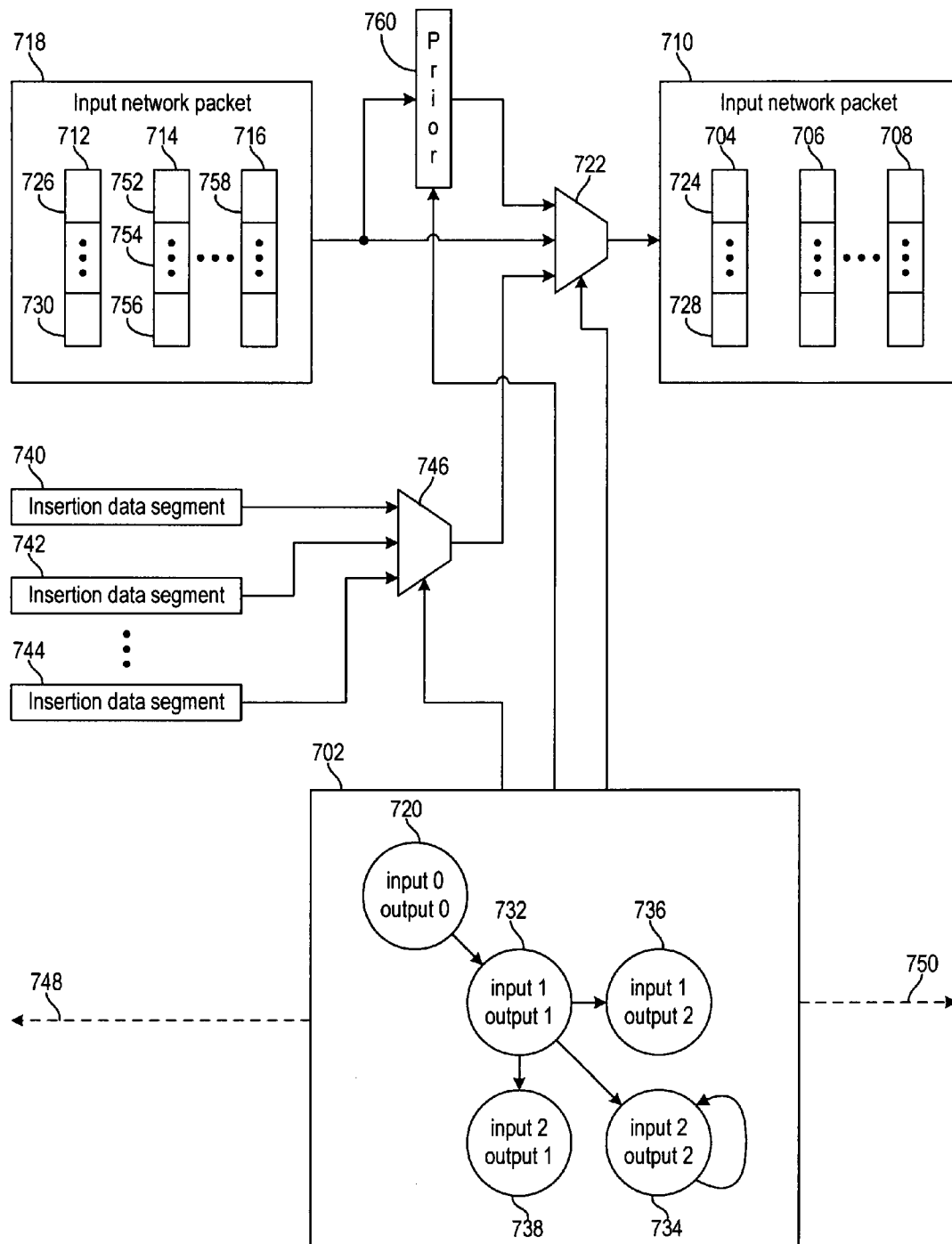
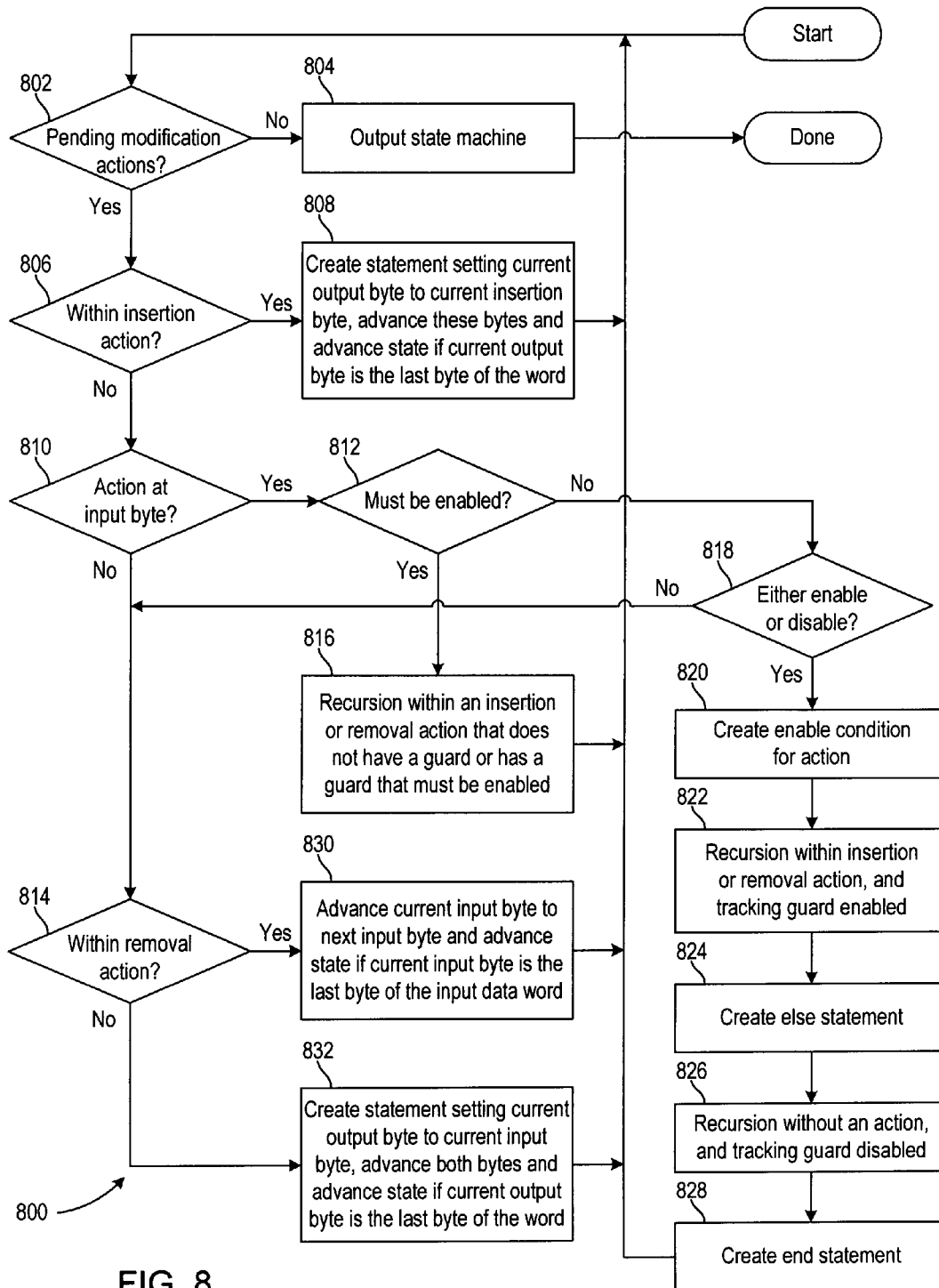


FIG. 7



1

CIRCUIT FOR MODIFICATION OF A NETWORK PACKET BY INSERTION OR REMOVAL OF A DATA SEGMENT

FIELD OF THE INVENTION

The present invention generally relates to processors of network packets, and more particularly to generation of a network packet processor.

BACKGROUND

A network packet processor may input a stream of network packets, manipulate the contents of the network packets, and output another stream of modified network packets. The manipulations may implement a protocol for processing network packets. For example, the network packet processor may implement a protocol layer of a communication protocol, and for a high-level packet received from a higher protocol layer and delivered to a lower protocol layer for eventual transmission on the communication media, the manipulations may encapsulate the high-level packet within a low-level packet of the lower protocol layer.

A protocol designer may develop a specification of the processing of network packets by a network packet processor. A hardware designer, such as an integrated circuit designer, may create a hardware implementation of a network packet processor that complies with the requirements specified by the protocol designer. Thus, development of a network packet processor implemented in hardware may require the specialized knowledge and skills of at least two separate fields of expertise. Because multiple fields of expertise may be required during development of a hardware implementation of a network packet processor, development of a network packet processor may be time consuming and expense.

Rapid development of the protocol or implementation of a network packet processor may improve the timeliness and/or quality of the network packet processor. For example, if a hardware implementation can be generated quickly given a specification of the processing of network packets, deficiencies discovered in an initial implementation may be addressed by modifying the specification and generating an improved implementation. Additional iterations of improvement may be possible if the hardware implementation can be generated quickly from a protocol specification.

The present invention may address one or more of the above issues.

SUMMARY

Various embodiments of the invention provide a circuit for converting a first network packet into a second network packet. A state machine converts the first network packet into the second network packet according to modification actions from a textual language specification. Each modification action is either an insertion action for inserting a data segment into the first network packet or a removal action for removing a data segment from the first network packet. The state machine includes states that each correspond to a pairing of a first data word from a first sequence of data words in the first network packet and a second data word from a second sequence of data words in the second network packet. Each data word in the first and second network packets include a same number of data units. Each state selects the data units of the second data word from the data segment of each insertion action and the data units of both the first data word and a prior data word to the first data word in the first sequence. Each

2

state specifies one or more next states including the state that corresponds to the pairing of either the first data word or a next data word after the first data word in the first sequence and either the second data word or a next data word after the second data word in the second sequence.

Various other embodiments of the invention provide a method for generating a state machine that converts a first network packet into a second network packet. A textual language specification is input that includes modification actions. Each modification action is either an insertion action for inserting a data segment into the first network packet or a removal action for removing a data segment from the first network packet. The method concurrently scans through first data units in a first sequence of data words in the first network packet and second data units in a second sequence of data words in the second network packet. Each data word in the first and second network packets includes a same number of the data units. The scan through the first data units in the first sequence of data words is suspended throughout each insertion action. The scan through the second data units in the second sequence of data words is suspended throughout each removal action. A respective state of the state machine is generated for each pairing encountered during the scan of a first data word from the first sequence and a second data word from the second sequence. The state selects the data units of the second data word from the data segment of each insertion action and the data units of both the first data word and a prior data word to the first data word in the first sequence.

BRIEF DESCRIPTION OF THE DRAWINGS

Various aspects and advantages of the invention will become apparent upon review of the following detailed description and upon reference to the drawings in which:

FIG. 1 is a block diagram of a system for generating a hardware description language specification of a network packet processor in accordance with various embodiments of the invention;

FIG. 2 is a listing of an example textual language specification of a network packet processor in accordance with various embodiments of the invention;

FIG. 3 is a data flow diagram of a process of generating a hardware description language specification of a network packet processor in accordance with various embodiments of the invention;

FIG. 4 is a block diagram of a network packet processor that is generated in accordance with various embodiments of the invention;

FIG. 5 is a flow diagram of a process for generating a pipeline for processing network packets in accordance with various embodiments of the invention;

FIG. 6 is a flow diagram of a process for generating a circuit for modifying a network packet in accordance with various embodiments of the invention;

FIG. 7 is a block diagram illustrating a state machine for inserting or removing data segments from a network packet in accordance with various embodiments of the invention; and

FIG. 8 is a flow diagram of a process for generating a circuit for inserting and/or deleting data segments from a network packet in accordance with various embodiments of the invention.

DETAILED DESCRIPTION

Various embodiments of the invention permit a network packet processor to be implemented from a specification of the required processing of network packets by the network

3

packet processor. The implementation of the network packet processor may be a hardware implementation in a programmable logic device (PLD) that may be quickly generated from the specification of the processing protocol of the network packet processor. The generated hardware implementation may have low latency and may efficiently utilize the resources of the PLD.

FIG. 1 is a block diagram of a system for generating a hardware description language (HDL) specification of a network packet processor in accordance with various embodiments of the invention. A processor 102 reads and executes instructions from a program storage medium 104, and the execution of the instructions causes processor 102 to generate the HDL specification of the network packet processor. Software modules 106 and 108 may include the instructions of the program storage medium 104.

Execution of the instructions of software module 106 may cause processor 102 to input a specification of the processing to be performed by the network packet processor. In one embodiment, the network packet processor may receive network packets and transmit a modification of the received network packets. The specification of the processing by the network packet processor may be a protocol specification of the modifications to be performed by the network packet processor. For example, the network packet processor may implement a protocol layer of a communication protocol, and for a high-level packet received from a higher protocol layer and delivered to a lower protocol layer for eventual transmission on the communication media, the specification may describe the encapsulation of the high-level packet within a low-level packet of the lower protocol layer. The encapsulation specified by the protocol specification may include insertion of a header with a particular format before the beginning of the high-level packet. The header may be generated from information in the incoming high-level packet and from information that is stored within the network packet processor.

Execution of the instructions of software module 106 may cause processor 102 to input the specification from the user interface 110. In one embodiment, the user interface 110 may permit directly inputting a textual language specification of the processing expected from the network packet processor. In another embodiment, the textual language specification may be input from a data file stored in program storage medium 104 or another storage medium of the system.

Execution of the instructions of software module 108 may cause processor 102 to generate an HDL specification of the network packet processor from the textual language specification input by processor 102 during execution of the instructions of software module 106. The network packet processor generated using software module 108 may implement the manipulations of the textual language specification.

Execution of the instructions of the optional software module 112 may cause processor 102 to translate the specification of the network packet processor into a simulator in a general purpose programming language and to simulate the network packet processor using the simulator.

FIG. 2 is a listing of an example textual language specification of a network packet processor in accordance with various embodiments of the invention. Various embodiments of the invention may translate the example textual language specification into a network packet processor that processes network packets as specified. The specification includes a declaration 202 of the name of the network packet processor, a declaration 204 of the ports of the network packet processor, a declaration 206 of formats of types of network packets and other data structures, a declaration 207 of a global variable,

4

and a specification 208 of a handler procedure for processing the network packets by the network packet processor.

As specified in ports declaration 204, the network packet processor has an input port with identifier "packetin" for receiving network packets, an output port with identifier "packetout" for transmitting network packets, and a memory port with identifier "memoryreq" for accessing an external memory. As specified in the handler specification 208, the network packets received at the input port are modified and the modified network packets are transmitted from the output port. Certain of the received network packets are modified using data that is read from an external memory using the memory port. It will be appreciated that a network packet may also be modified using state data that is stored within the network packet processor.

The handler specification 208 specifies processing of network packets that have type "Ippacket" and are received at the input port "packetin." The handler specification 208 includes a declaration 210 of a local variable and specifications of five actions, including set action 212, set action 213, memory read action 214, insert action 216, and forwarding action 218. For each network packet of type "Ippacket" that is received at the input port "packetin," set action 212 decrements the "TTL" field of the network packet. Set action 213 sets the value of the "checksum" field to a value of the global variable of declaration 207. Memory read action 214 reads a memory at the address given by the "id" field of the network packet. The insert action 216 includes a guard condition within the square brackets that enables the insert action 216 when the "version" field of the network packet has a value of 4, and otherwise the insert action 216 is disabled. If the insert action 216 is enabled by the guard condition, the insert action 216 inserts data from the memory read action 214 after the "destination" field of the network packet. Forwarding action 218 transmits the modified network packet at the output port "packetout."

The value of the global variable of declaration 207 could be updated by a handler procedure during the processing of one network packet, and the updated value could be used during the processing of another network packet by the handler procedure. Thus, the global variable of declaration 207 may provide state data that is preserved between network packets. In contrast, the local variable of declaration 210 may be temporary data that should be reinitialized for each network packet that is processed.

The textual language for specifying the processing of a network packet handler may closely correspond with the techniques that a designer uses to specify a communication protocol. A designer of communication protocol may develop the textual language specification without having the specialized knowledge required to develop an integrated circuit, and various embodiments of the invention may translate the textual language specification of a communication protocol into an efficient implementation of the communication protocol in a network packet processor. Thus, various embodiments of the invention permit a designer of a communication protocol to develop an integrated circuit for efficiently implementing the communication protocol even though the designer does not have the skills for designing an integrated circuit.

FIG. 3 is a data flow diagram of a process of generating a hardware description language (HDL) specification 302 of a network packet processor in accordance with various embodiments of the invention. The HDL specification 302 may be generated from a textual language specification 304 of the processing performed by the network packet processor.

The textual language specification 304 of the network packet processor may include port identifiers 306 and 308, global variables 309, a format 310 or formats for one or more

5

types of network packets and possibly other data structures, and a procedure **312** for each input port and each type of network packet that is expected at the input port of the network packet processor. The port identifiers **306** and **308** may include respective identifiers for the input, output, and any memory ports of the network packet processor. The global variables **309** may provide state data that is preserved between network packets. A format **310** includes the fields **314** and **316** of a type of network packet, and a field **314** or **316** may be declared in the format **310** as having a specified name and a specified size, such as a number of data bits of the field. The fields **314** and **316** may be specified in format **310** in an order corresponding to an order of the network packet manager receiving or transmitting the fields in the type of network packets.

A procedure **312** may include actions **318**, **320**, and **322** for manipulating network packets of a particular type that are received at a particular input port. Examples for actions **318**, **320**, and **322** include a set action for setting a value of a field **314** as a function of global variables **309** and/or fields **314** and/or **316**, an insert action for data insertion at a field **314** as a function of global variables **309** and/or fields **314** and/or **316**, a remove action for data removal at field **314** as a function of global variables **309** and/or fields **314** and/or **316**, an update action for updating one of the global variables **309** as a function of the global variables **309** and/or fields **314** and/or **316**, a forward action for transmitting modified network packets from an output port of the network packet processor, a memory read action for reading a value from a memory, and a memory write action for writing a value to the memory.

An action **318** may include a guard condition **324** for enabling and disabling the action **318**, with the action **318** performed in response to the guard condition **324** being satisfied and the action **318** being bypassed in response to the guard condition **324** not being satisfied. Another action **320** might not include a guard condition and the action **320** may always be performed on each network packet that is processed using procedure **312**. Examples for guard condition **324** include a check of a value of a field **314** of a type of the network packets, a comparison between two fields **314** and **316** of a type of the network packets, and a comparison between a field **314** and one of the global variables **309**.

At step **326**, the specification **304**, which specifies the processing performed by the network packet processor, is input. At step **328**, the HDL specification **302** of the network packet processor is generated. To generate the HDL specification **302** of the network packet processor, a specification **330** may be input that specifies the characteristics **332** and **334** of the ports having identifiers **306** and **308**. Example characteristics include a width for each of the ports and a signaling protocol for each of the ports.

The generated HDL specification **302** may include a look-ahead stage **336**, an operation stage **338**, an insert/remove stage **340**, and an interleave stage **342**. The look-ahead stage **336** may synchronize among the fields **314** and **316** that are accessed by the actions **318**, **320**, and **322** of the procedure **312**. The look-ahead stage **336** may make certain of fields **314** and **316** concurrently available for processing according to the actions **318**, **320**, and **322** of the procedure **312**. The operation stage **338** may implement the actions **318**, **320**, and **322** that are not insert or remove actions. For each set action, the operation stage may set one of fields **314** and **316** according to the set action. The operation stage **338** may evaluate a guard condition **324** for an action **318**, and bypass performing the action **318** when the guard condition **324** is not satisfied. The insert/remove stage **340** may perform data insertion and/or data removal at one or more of fields **314** and **316** according

6

to the actions **318**, **320**, and **322** that are insert or remove actions. The data insertion or data removal may be bypassed for an insert or remove action with a guard condition that is not satisfied. The interleave stage **342** may ensure that the modified network packet follows rules for interleaving network packets.

A sequence of network packets may be received at an input port of the network packet processor, with each network packet including a series of data chunks. The data chunks for the network packets may be serially received in an interleaved manner. For example, network packet A may include data chunks A[0], A[1], and A[2], and network packet B may include data chunks B[0], B[1], and B[2], which are serially received at an input port of the network packet processor in the interleaved order A[0], B[0], A[1], B[1], A[2], B[2]. For a network packet processor with an insert/remove stage **340** that modifies network packet A by removing a portion of data chunk A[1], to output full data chunks the network packet processor may need to fill in modified data chunk A[1] with data from subsequent chunks of network packet A. Similarly, for a network packet processor with an insert/remove stage **340** that modifies network packet B by inserting data into chunk B[1], to output proper data chunks the network packet processor may need to move data from modified data chunk B[1] to subsequent chunks of network packet B.

In one embodiment, a data chunk is the data width of the input and output ports of a network packet processor. The interleaving rules may permit interleaving of network packets between each data transfer across the width of a port of the network packet processor. After a data insertion or data removal with a size that is not a multiple of the data chunk, the interleave stage **342** may pad the modified network packet with enough null data to make a full data chunk at the end of the network packet. It will be appreciated that the interleave stage **342** may be useful for generating complete data chunks across the full width of the input and output ports even though data chunks for different network packets are never interleaved.

In another embodiment, a data chunk is a burst of a fixed or variable number of data transfers across the width of a port of the network packet processor, and interleaving of network packets may occur between the bursts of the data chunks. For a variable number of data transfers, the interleaving rules may specify an upper limit on the number of data transfers within a burst. After a data insertion that increases the size of a network packet beyond the fixed number of data transfers allowed for a burst or beyond the upper limit on a variable number of data transfers for a burst, the interleave stage **342** may divide the modified network packet into multiple bursts to satisfy the interleaving rules.

It will be appreciated that the port characteristics **330** may specify both the width of the ports and the burst lengths for the ports. In addition, port characteristics **330** may specify whether network packets are received in an interleaved manner by the network packet processor.

FIG. 4 is a block diagram of a network packet processor that is generated in accordance with various embodiments of the invention. The network packet processor is a pipeline **400** that includes a look-ahead stage **402**, an operation stage **404**, an insert/remove stage **406**, and an interleave stage **408**. Stages **402**, **404**, **406**, and **408** of the pipeline **400** are configured by a textual language specification of the processing by the network packet processor.

The textual language specification may specify one or more actions for modifying the fields of a network packet as a function of the fields of the network packet and global variables corresponding to stored state **428** and **450** within the

network packet processor. For example, a set action may modify a field of a network packet as a function of another field of this network packet, and these two fields may be received at different times during a serial reception of the data chunks of the network packet. Each modification of a field may be completed upon receiving the field providing the field is modified as a function of stored state and fields that are already available from the network packet. In contrast, a modification of a field might not be possible upon receiving the field when the field is modified as a function of a field or fields that are later received. The look-ahead stage **402** may effectively advance the availability of each field that is used to modify an earlier-received field, such that the operation stage **404** may perform the modification for each set action upon receiving the data chunk that includes the field that is modified by the set action.

To effectively advance the availability of certain fields, the look-ahead stage **402** may promptly provide these fields from look-ahead queue **410**, while the look-ahead stage **402** may provide other fields with delayed availability from delay FIFO **412**. Various fields may be concurrently available from the look-ahead queue **410** and the delay FIFO **412**. The delay FIFO **412** may have a depth equaling the maximum amount of relative advancement that is needed. If no fields need advanced availability for a particular textual language specification, the look-ahead stage **402** may be omitted from the pipeline **400**.

Certain fields of a network packet that need advanced availability may be selected by pre-fetch block **414** and stored in look-ahead queue **410** during reception of the network packet at the input port on line **416**. Because these fields are promptly made available to the operation stage **404** from look-ahead queue **410**, these fields are effectively advanced relative to fields that are later made available to the operation stage **404** from delay FIFO **412**.

The network packet processor may access a memory or memories during the processing of a network packet. For example, an address for a memory read action may be formed from a field of the network packet and a set action may modify the same field of the network packet using the value read from the memory at the address. The set action may include a guard condition that enables modifying the same field with a modification value, and either the guard condition or the modification value of the set action may be dependent on the value read from the memory. If a delay interval is required for reading the memory, the modification of the field might not be immediately possible upon receiving the field. The delay interval may be a latency of the memory and/or a time interval needed to form wide data by reading narrow data multiple times from the memory. The look-ahead stage **402** may also effectively advance the availability of certain fields needed for reading from one or more memories.

For the memory read actions that need advanced availability and that access the memory connected to the memory port on line **420**, memory pre-fetch block **418** may capture any fields needed to generate any addresses for these memory read actions. Memory pre-fetch block **418** may also capture the fields needed to generate any guard conditions for these memory read actions.

Similarly, memory pre-fetch block **422** may capture the fields needed to determine any addresses and any guard conditions for the memory read actions that need advanced availability for the memory connected to the memory port on line **424**. When advanced availability is needed for a value read from a memory for a memory read action, the amount of advanced availability may contribute to determining the depth of the delay FIFO **412**.

The stored state block **428** may include state data for certain of the global variables of the specification of the processing of the network packet processor. Some of the global variables may be used in actions needing advanced availability, and the stored state block **428** of look-ahead stage **402** may store the state data for these global variables, while the stored state block **450** of operation stage **404** may store the state data for the remaining global variables.

Certain fields of a network packet may be referenced by multiple actions of the textual language description. For example, a specific field may be referenced by two set actions, which modify respective fields that are in different data chunks and are consequently received by the pipeline **400** at different times. Thus, certain fields may need to be available at multiple time steps and to eliminate duplicated storage these fields may be stored in a register within shared block **426**. Shared block **426** may make these fields and/or certain temporary variables available to one or more of blocks **414**, **418**, and **422**.

The control block **430** may control the sampling of certain fields of a network packet by the blocks **428**, **426**, **414**, **418**, and **422** as specified by the certain actions of the textual language specification.

The operation stage **404** may implement the actions of the textual language specification excluding the insert and remove actions. The edit block **432** may perform the set actions on each network packet as the data chunks of the network packet are received from the delay FIFO **412** of the look-ahead stage **402**. The edit block **432** may perform the set actions using fields with advanced availability from the look-ahead queue **410** of the look-ahead stage **402** and memory read results from the memory read blocks **434**, **436**, and **438**.

The edit block **432** may determine information that is provided to the memory write blocks **440**, **442**, and **444**, and to the insert/remove block **446**. Certain fields and/or temporary variables may be shared between the shared block **448** and the blocks **432**, **434**, **436**, **438**, **440**, **442**, **444**, and **446**. The edit block **432** may update the values of any temporary variables in the shared block **448**.

Because the memory connected to the memory port on line **420** may have a memory read action that produces a read result needing advanced availability, the memory read block **434** may receive certain fields with advanced availability from the look-ahead queue **410**. These fields may be used to determine any address or any guard conditions for this memory read action. Any address or guard condition for this memory read action may also be determined from other fields that do not need advanced availability and state data from stored state **450**.

The memory read block **434** may implement all of the memory read actions that target the memory connected to the memory port on line **420**. Depending upon whether or not the read result needs advanced availability for a memory read action accessing the memory port on line **420**, the memory read block **434** may generate an address or guard condition for the memory read action using information from the look-ahead queue **410**, or the memory read block **434** may generate an address or guard condition using information from the shared block **448**.

The memory read block **438** may implement all of the memory read actions that target the memory connected to the memory port on line **452**, and none of these memory read actions may need advanced availability of the read result. Because the memory read block **438** does not need to obtain advanced information from the look-ahead queue **410**, no connection is shown between the look-ahead queue **410** and the memory read block **438**. For example, the memory con-

nected to the memory port on line **452** may be a FIFO or configuration register that does not require an address and hence advanced availability is not needed for the result read from the memory because the memory may be read at any time.

It will be appreciated that the actual number of memory read blocks **434**, **436**, and **438** for a particular textual language specification may depend on the number of memories accessed by the memory read actions of the textual language specification. In addition, the connections between the look-ahead queue **410** and the memory read blocks **434**, **436**, and **438** may depend on whether advanced availability is needed for the memory reach actions of the textual language specification.

For each memory write action of the textual language specification, the edit block **432** may provide the write value along with an optional address and optional guard condition to the write memory blocks **440**, **442**, and **444**. Write memory blocks **440**, **442**, and **444** may implement the memory write actions that write to the respective memories connected to memory ports on lines **420**, **424**, and **452**.

For each insert action and each remove action of the textual language specification, the insert/remove block **446** may determine the position of the data insertion or data removal and the amount of data to be inserted or removed. The insert/remove block **446** may also determine the value of any guard condition for enabling and disabling the data insertion or data removal. For data insertion, the insert/remove block **446** may determine the insertion data. The insert/remove block **446** may obtain information for determining the position and amount of the data insertion or data removal and the insertion data from the edit block **432** and/or the shared block **448**.

The control block **454** may manage the operation of the blocks **432**, **434**, **436**, **438**, **440**, **442**, **444**, **446**, and **448** of the operation stage **404**.

The insert/remove stage **406** implements the insert actions and remove actions of the textual language specification. The insert/remove stage may be omitted from the pipeline **400** for a specific textual language specification that does not include any insert or remove actions. The insertion/removal block **456** performs the data insertion of any insert actions and the data removal of any remove actions according to information received from the insert/remove block **446** of the operation stage **404**. The control block **458** controls the operation of the insertion/removal block **456**. The insertion/removal block **456** may serially receive potentially modified data chunks of a network packet from the edit block **432** of the operation stage **404**. While a data insertion or data removal is not in progress, the data chunks received by the insertion/removal block **456** are the data chunks output by the insertion/removal block **456**.

Each insert action and each remove action of a textual language specification may optionally include a guard condition. An action that does not include a guard condition is always enabled, and an action with a guard condition is enabled when the guard condition is determined to have an enabling value.

For a remove action that is enabled, upon receiving the data chunk of the network packet that includes the position of the data removal, the insertion/removal block **456** may output a partial data chunk that includes the portion of the data chunk before the position of the data removal. The insertion/removal block **456** may suspend outputting data chunks until the specified amount of data is removed and the insertion/removal block **456** may subsequently resume outputting data chunks of the network packet. The first data chunk output on resuming output may be a partial data chunk. In one embodi-

ment, multiples of a byte may be inserted and removed and the insertion/removal block **456** outputs a number of bytes in parallel for each word of the burst of a data chunk. Each byte that is output may have a corresponding data valid signal to indicate whether the byte is included in a full or partial data chunk.

For an insert action that is enabled, upon receiving the data chunk of the network packet that includes the position of the data insertion, the insertion/removal block **456** may output a full data chunk and then switch to outputting the insertion data. The final data chunk of the insertion data may be a partial data chunk and the insertion/removal block **456** may subsequently resume outputting full data chunks of the network packet. During the data insertion, the insert/remove stage **406** may stall the operation stage **404** and the look-ahead stage **402** to prevent data loss.

The interleave stage **408** may ensure that the modified network packet follows rules for interleaving network packets. The interleave stage **408** may eliminate the gaps in the valid data from any partial data chunks created by the insert/remove stage **406**. The interleave stage **408** may divide a network packet into multiple bursts to satisfy the interleaving rules. The interleave stage **408** may be omitted for a textual language specification that only inserts and removes data that is always a multiple of the size of the data chunk. The interleave stage **408** may be eliminated along with the insert/remove stage **406** for a textual language specification that does not include any insert or remove actions.

The interleave block **460** may generate each data chunk that is output from the pipeline **400** by selecting from either the chunk buffer **462** or the insertion/removal block **456** of the insert/remove stage **406**. The chunk buffer **462** may contain a portion of the preceding network packet. The control block **464** may be a state machine that controls the operation of the chunk buffer **462** and the interleave block **460**.

FIG. **5** is a flow diagram of a process for generating a pipeline for processing network packets in accordance with various embodiments of the invention. The processing of network packets by the pipeline may be specified in a textual language specification, and this specification may be analyzed to generate a specification of the pipeline in a hardware description language. Synthesis tools may generate a hardware implementation of the pipeline from the generated specification. For example, ISE Foundation available from Xilinx, Inc. may be used to create an implementation of the pipeline in a programmable logic device.

At step **502**, the specification of the processing of the pipeline may be analyzed to determine the number of memories accessed by the pipeline. For each memory, a dependency graph may be created for scheduling the memory reads that access the memory. Each memory read may require an optional address and an optional guard condition, and each memory read may provide a result value. The addresses and guard conditions may be calculated from certain fields of a network packet and a node may be added to the dependency graph for each of these fields and for each read result from the memory. These nodes are marked with an availability time relative to a start of the network packet. An availability time for a field may be readily calculated from a position of the field within the network packet and a data width of an input of the pipeline for receiving the network packet. For a read result that is dependent on a set of fields for calculating the address and guard condition, the read result may be marked with an availability time that is the sum of the latency for the memory read and the latest availability time of the fields in the set. It will be appreciated that for a memory read without an address and without a guard condition, such as a read from a configu-

11

ration memory, the result of the memory read may be marked as available at time zero corresponding to the beginning of the network packet.

The optional address and optional guard condition of each memory read may also be dependent on a local variable that is further dependent on certain fields of a network packet, global variables, or the results of other memory reads. Additional nodes may similarly be added and marked in the dependency graph for these indirect dependencies.

Once each dependency graph is resolved to assign an availability time to each node in the graph, each memory read is scheduled to provide the read result at the availability time of the read result. For a memory having a non-zero read latency, the memory read may be issued in advance to provide the read result at the required time. Because a result from a memory read may be used to determine the address or guard condition of another memory read, perhaps indirectly via a local variable, iteration may be required to completely resolve the dependency graph.

At step 504, a storage class may be determined for each field used to generate the address and guard conditions of the memory reads. For a field with an availability time that matches the issue time of every memory read that uses the field, the storage class may be a wire class indicating a direct connection. For a field with an availability time in advance of the issue time of every memory read that used the field, the storage class may be a register class. Otherwise, the storage class may be a register with a bypass path.

At step 506, a depth of the delay FIFO of the look-ahead stage may be determined. The delay FIFO may receive a stream of network packets from an input of the pipeline and the delay FIFO may output the stream of network packets after a fixed time delay, which is set by the depth of the delay FIFO.

In one embodiment, the depth may be selected to have the delay FIFO output the start of a network packet at a time when the results are available for all of the memory reads. Thus, the depth of the delay FIFO may equal the largest value of the availability times of the read results from the dependency graph or graphs. The delay FIFO may be regarded as effectively advancing the memory read results to ensure availability at time zero when the start of the network packet is output from the delay FIFO.

In another embodiment, a usage time and a difference between the usage time and the availability time is determined for each memory read result and the depth of the delay FIFO may equal the largest difference for the memory read results that have a usage time before the availability time.

At step 508, a dependency graph may be created for scheduling the modifications of a network packet. A network packet may be modified by changing a field of the network packet, inserting data into a network packet, or removing data from a network packet. Various fields of the network packet may be used to determine, for example, a modification value for changing a field of a network packet or a packet-dependent length of data to be inserted or removed at a field of the network packet. Certain fields of the network packet may also be used to determine a guard condition for enabling or disabling a field modification or a data insertion or deletion.

A node may be added to the dependency graph for each of these fields and these nodes may be marked with an availability time of the receiving the field relative to a start of the network packet. Additional nodes may be added for the calculated modifications and these additional nodes may be marked with corresponding availability times. From the dependency graph, the modifications of the network packet may be scheduled at the availability times determined for the

12

guard conditions of the modifications, the modification values, the variable locations of data insertion or removal, the variable lengths of data insertion or removal, and the insertion data values.

The FIFO depth selected at step 506 may provide inadequate advancement of some of the fields used for packet modifications. At step 510, these additional fields needing look-ahead may be identified, and the FIFO depth determined at step 506 may be increased and consequently the storage classes determined at step 504 may be adjusted. At step 512, the storage classes are determined for the fields used for packet modifications.

At step 514, a specification of the look-ahead, operation, insert/remove, and interleave stages may be generated in a hardware description language.

FIG. 6 is a flow diagram of a process for generating a circuit for modifying a network packet in accordance with various embodiments of the invention. Optional steps of the process are shown with a dotted outline. The circuit may modify the network packets as the data words of the network packet flow through the circuit. The circuit may modify an input network packet according to insertion actions for inserting data and removal actions for removing data as specified by a textual language specification.

The circuit may include a state machine that controls the progress of the modification of the network packet. Each state of the state machine may track a relationship between a data word being received in the input network packet and a potentially modified data word being transmitted in the output network packet. A linear sequence of states may track this relationship until reaching the data word of an insertion or removal action that has a guard condition for enabling and disabling the action. For a state of the state machine corresponding to a data word associated with an action with a guard condition, this state may be specified to branch to one next state if the guard condition is satisfied and another next state if the guard condition is not satisfied.

At step 602, the textual language specification is input that specifies modification actions for inserting and/or removing data segments from an input network packet. Each insertion action may specify a point in an input network packet for inserting a data segment of a specified length and each removal action may specify a point in the network packet for deleting a data segment of a specified length. At optional step 604, the modification actions may be sorted in the order of their modification points because the modification actions may be processed in the sorted order.

At step 606, the process may scan through the input network packet concurrently with a scan through the output network packet, which is a modification of the input network packet according to the specification input at step 602. The input and output network packets may each include a sequence of data words, and the concurrent scanning through the input and output network packets may scan through the data words of the input network packet concurrently with a scan through the data words of the output network packet.

The concurrently scanning through the data words of the input and output network packets may be a synchronized scan until reaching a data word of the input network packet for a modification action. If the modification action is an insertion action, the scan of the data words of the input network packet may be suspended for the length of the data segment of the insertion action while the scan may continue for the data words of the output network packet. Similarly, if the modification action is a removal action, the scan of the data words of the output network packet may be suspended for the length of

13

the data segment for the removal action while the scan may continue for the data words of the input network packet.

At optional step 608, the process may track whether the guard conditions of the modification actions can be enabled or disabled. For example, certain states of the state machine may be reachable after a branch for a satisfied guard condition of modification action. If another modification action at one of these reachable states has the same guard condition, then this guard condition should also be unconditionally satisfied and the reachable state does not need a branch for the guard condition being unsatisfied. Tracking whether the guard conditions can be enabled and disabled may simplify the generated state machine and eliminate unreachable states from the state machine.

At step 610, a state of the state machine is generated for each pair of a data words encountered during the scan through the input and output network packets. Each state may correspond to an input data word from the input network packet and an output data word from the output network packet. The state may select the data units, such as data bytes, of the output data word from the data units of the input data word, a preceding data word in the input network packet, and from any insertion data segments. For example, an insertion action without a guard condition may insert one byte at the last byte of an input data word and the corresponding output data word may be the input data word with the last byte replaced with the insertion byte. The next state of the state machine may select the next output data word as the replaced byte followed by all but the last byte from the current input data word, with the replaced byte coming from a preceding data word to the current data word in the input network packet.

At optional step 612, a specification may be output that specifies the state machine in a hardware description language, such as VHDL or Verilog.

FIG. 7 is a block diagram illustrating an example state machine 702 for inserting or removing data segments from a network packet in accordance with various embodiments of the invention. The state machine may control the generation of a sequence of output data words 704, 706, and 708 of an output network packet 710 from a sequence of input data words 712, 714, and 716 of an input network packet 718 and from insertion data segments 740, 742, and 744.

State 720 of state machine 702 may correspond to the pairing of input data word 712 with output data word 704. The specification of the processing might not include any modification actions for input data word 712, such that state machine 702 controls multiplexer 722 to select the data unit 724 of output data word 704 from the data unit 726 of input data word 712 and to select the data unit 728 of the output data word 704 from the data unit 730 of the input data word 712. Because there may be no modification actions for input data word 712, the scanning through the input and output network packets 718 and 710 may advance to the next input data word 714 and the next output data word 706, and state 732 may correspond to these data words 714 and 706.

The specification may include an insertion action and a removal action at the corresponding input data word 714 for state 732. Both the insertion action and the removal action may include a respective guard condition. The next state for state 732 may be one of states 734, 736, or 738 depending upon the guard conditions.

If both guard conditions of the insertion and removal actions are not satisfied, then the scan through the input and output network packets 718 and 710 may proceed to state 734 for the subsequent data words 716 and 708.

If the insertion action is enabled by its guard condition and the removal action is disabled by its guard condition, state

14

machine 702 may direct multiplexers 722 and 746 to select one of data segments 740, 742, and 744 for insertion into the output network packet 710 beginning at a specified data unit within data word 706. The insertion action may also specify the appropriate one of data segments 740, 742, and 744 to insert into the network packet 710. In one embodiment, the state machine 702 may optionally assert a flow control signal on line 748 to stall the input network packet 718 at data word 714 until the insertion is complete. To track that an insertion is in progress, the state machine 702 may transition to state 736 corresponding to the pairing of stalled input data word 714 and a subsequent output data word 708. It will be appreciated that the state machine 702 may include a sequence of transitions from state 736 to additional states to track the progress of the insertion.

If the removal action is enabled by its guard condition and the insertion action is disabled by its guard condition, the state machine 702 may track the progress of the data removal by a transition from state 732 to state 738 corresponding to the pairing of an input data word 716 with output data word 706. In one embodiment, the state machine 702 may optionally deassert a data valid signal on line 750 to indicate a repeated outputting of data word 706 until the removal is complete. It will be appreciated that the state machine 702 may include a sequence of transitions from state 738 to additional states to track the progress of the data removal.

If both guard conditions of the insertion and removal actions are satisfied, the state machine 702 may serially perform the insertion and the removal or the state machine 702 may overlap the insertion and removal for a number of data words corresponding to the action with the shorter length, followed by completing the other action. If the state machine 702 serially performs the actions, the insertion action may be completed before beginning the removal action or vice versa. The progress of the insertion and removal may be tracked by additional states and/or transitions of state machine 702 that are not shown in FIG. 7.

It will be appreciated that an example state machine 702 may control an insertion action and a removal action with guard conditions that are complementary conditions. For such an example state machine 702, the insertion and removal actions would not be enabled concurrently.

An insertion or removal action may insert or remove a number of data units that is not a whole number of data words. For example, a removal action may specify a removal with a length of one data unit beginning at data unit 752 of data word 714 of input network packet 718. If there is no enabled insertion action at data word 714, then output data word 706 may be selected as the unaligned data units 754 and 756 of input data word 714 and the data unit 758 of input data word 716. Prior data word 760 may store the data word preceding a current data word of the input network packet 718. Thus, input data word 714 may be stored in prior data word 760 when the current data word is data word 716. Multiplexer 722 may select the output data word 706 from data word 758 of input data word 716 and the input data words 754 and 756 that are stored in prior data word 760.

Because all actions may be processed upon reaching state 734 and certain other states of state machine 702, any alignment shift between the data words of the input and output network packets 718 and 710 may be fixed for the rest of the these network packets 718 and 710. State 734 may track the fixed alignment by transitioning to itself. It will be appreciated that separate logic (not shown) may detect the end of network packets 718 and 710 and reset state machine 702 for the next network packet.

15

FIG. 8 is a flow diagram of a process 800 for generating a circuit for inserting and/or deleting data segments from a network packet in accordance with various embodiments of the invention. The process 800 may generate a description of a state machine for controlling the modification of the network packet according to actions for inserting and/or deleting the data segments.

In one embodiment, a network packet may include data words that each include a number of data bytes. Process 800 may serially scan through the data bytes of a type declaration of the network packet and may evaluate each modification action beginning at the data byte matching a specified position for the modification action in the network packet.

Decision 802 may check whether all modification actions are processed. Process 800 outputs the state machine at step 804 if all modifications are processed. Otherwise, process 800 proceeds to decision 806.

Decision 806 checks whether the current output byte is within an insertion action. If the current output byte is within an insertion action, process 800 proceeds to step 808; otherwise process 800 proceeds to decision 810. At step 808, a current state of the state machine is updated to set a value of the current output byte to the value of the current insertion byte of the insertion data segment for the insertion action. The current output byte and the current insertion byte are both advanced to a next byte, but the current input byte remains the same. If the current output byte is advanced from the last byte of a data word to the first byte of the next data word, then a new state is generated corresponding to the combination of the stalled input data word and this next output data word, and the new state becomes the current state. If the current insertion byte is advanced beyond the end of the insertion data segment, then the insertion action is completed. After step 808, process 800 returns to decision 802.

Decision 810 checks whether a modification action is associated with the current input byte of the current data word of the input network packet. For a modification action associated with the current input byte, process 800 proceeds to decision 812 and otherwise process 800 proceeds to decision 814. Decision 812 checks whether the modification action should be enabled. A modification may be enabled because the modification action does not have a guard condition or because the modification action has a guard condition that should be enabled. The current state of the state machine may be reachable only when certain guard conditions evaluate to a particular result, and the evaluated results for these guard conditions may imply that the current modification action has its guard condition satisfied. If the modification action should be enabled, process 800 proceeds to step 816. If the modification action could be either enabled or disabled, process 800 proceeds to decision 818.

At step 816, process 800 may be recursively invoked to process the modification action or actions at the current input byte. The modification actions processed at step 816 may be marked or removed from queue such that these modification actions are not processed again during the recursion. The recursion may complete by skipping step 804. After the recursion, process 800 may return to decision 802.

Decision 818 checks whether the guard condition of the current modification action or actions may be either enabled or disabled, depending on the values of the fields of the input network packet for example. If the guard condition may be either enabled or disabled, process 800 proceeds to step 820. Otherwise, the guard condition should always be disabled and process 800 proceeds to step 814.

At step 820, an enable condition is generated for testing that the guard condition is enabled for the current modifica-

16

tion action. The enabled condition may be an expression for checking the values of one or more fields of the input network packet according to a specification of the modification action. At step 822, process 800 may be recursively invoked including tracking that the guard condition is enabled. For example, a subsequently encountered modification action may have the same guard condition, such that the recursion of process 800 should proceed from decision 812 to step 816 during the processing of this subsequently encountered modification action.

At step 824, an "else" condition is generated that matches the enable condition generated at step 820. At step 826, process 800 may be recursively invoked including tracking that the guard condition is disabled. For example, a subsequently encountered modification action may have the same guard condition as the current modification action, such that the recursion of process 800 should proceed from decision 812 to decision 818 to decision 814, effectively skipping the subsequently encountered modification action during its processing. At step 828, an "end" condition is generated that matches the enable condition generated at step 820 and the "else" condition generated at step 824.

Decision 814 checks whether the current input byte is within a removal action. If the current input byte is within a removal action, process 800 proceeds to step 830; otherwise process 800 proceeds to step 832. At step 830, the current input byte is advanced to a next byte, but the current output byte remains the same. If the current input byte is advanced from the last byte of a data word to the first byte of the next data word, then a new state is generated corresponding to the combination of this next input data word and the stalled output data word, and the new state becomes the current state. The removal action is completed when a specified length of data is removed.

At step 832, a current state of the state machine is updated to set a value of the current output byte to the value of the current input byte. The current input byte and the current output byte are both advanced to a next byte. If the current input byte is advanced from the last byte of a data word to the first byte of the next data word, then a new state may be generated corresponding to the combination of this next input data word and the stalled output data word, and the new state becomes the current state. After steps 830 and 832, process 800 returns to decision 802.

Example 1 is an example of a state machine that may be generated by process 800. The most significant byte of a word is the first byte of a data word in Example 1. An insertion action may specify inserting two bytes of an insertion data segment beginning at the third byte of the second data word of the input network packet.

EXAMPLE 1

State 0:

```
out_word (31 downto 24)<=in_word (31 downto 24);
out_word (23 downto 16)<=in_word (23 downto 16);
out_word (15 downto 8)<=in_word (15 downto 8);
out_word (7 downto 0)<=in_word (7 downto 0);
next_state<=State1;
```

State 1:

```
out_word (31 downto 24)<=in_word (31 downto 24);
out_word (23 downto 16)<=in_word (23 downto 16);
if (guard(0)) then
out_word (15 downto 8)<=insert_word (31 downto 24);
out_word (7 downto 0)<=insert_word (23 downto 16);
next_state<=State2;
```

17

```

else
out_word (15 downto 8)<=in_word (15 downto 8);
out_word (7 downto 0)<=in_word (7 downto 0);
next_state<=State3;
endif;

```

State 2:

```

out_word (31 downto 24)<=prev_in_word (15 downto 8);
out_word (23 downto 16)<=prev_in_word (7 downto 0);
out_word (15 downto 8)<=in_word (31 downto 24);
out_word (7 downto 0)<=in_word (23 downto 16);
next_state<=State2;

```

State 3:

```

out_word (31 downto 24)<=in_word (31 downto 24);
out_word (23 downto 16)<=in_word (23 downto 16);
out_word (15 downto 8)<=in_word (15 downto 8);
out_word (7 downto 0)<=in_word (7 downto 0);
next_state<=State3;

```

Those skilled in the art will appreciate that various alternative computing arrangements, including one or more processors and a memory arrangement configured with program code, would be suitable for hosting the processes and data structures of the different embodiments of the present invention. In addition, the processes may be provided via a variety of computer-readable media or delivery channels such as magnetic or optical disks or tapes, electronic storage devices, or as application services over a network.

While particular embodiments of the present invention have been shown and described, it will be obvious to those skilled in the art that changes and modifications may be made without departing from this invention in its broader aspects and, therefore, the appended claims are to encompass within their scope all such changes and modifications as fall within the true spirit and scope of this invention.

What is claimed is:

1. A circuit for converting a first network packet into a second network packet, the circuit comprising:

a state machine adapted to convert the first network packet into the second network packet according to at least one modification action from a textual language specification, wherein:

each of the at least one modification action being one of an insertion action for inserting a data segment into the first network packet and a removal action for removing a data segment from the first network packet;

the state machine including a plurality of states that each correspond to a pairing of a first data word from a first sequence of data words in the first network packet and a second data word from a second sequence of data words in the second network packet;

each data word in the first and second network packets including a same number of a plurality of data units, the state for selecting the data units of the second data word from the data segment of each insertion action of the at least one modification action and the data units of both the first data word and a prior data word to the first data word in the first sequence; and

each state specifying at least one next state that includes one of the states that corresponds the pairing of one of the first data word and a next data word after the first data word in the first sequence and one of the second data word and a next data word after the second data word in the second sequence;

the state machine being configured to concurrently scan through a first plurality of data units in a first sequence of data words in the first network packet and a second plurality of data units in a second sequence of data words

18

in the second network packet, wherein the scanning through the first plurality of data units is suspended throughout each insertion action, and the scanning through the second plurality of data units is suspended throughout each removal action; and

a look-ahead stage, an operation stage coupled to the look-ahead stage, an insert/remove stage that includes the state machine and is coupled to the operation stage, and an interleave stage that is coupled to the insert/remove stage.

2. The circuit of claim 1, wherein the state machine is automatically generated in response to the textual language specification.

3. The circuit of claim 1, wherein the look-ahead stage synchronizes between at least two of a plurality of fields of the first network packet, the operation stage modifies at least one of the fields of the first network packet, the insert/remove stage performs each insertion action and each removal action of the at least one modification action, and the interleave stage ensures that the second plurality of network packets follows at least one rule for interleaving network packets.

4. The circuit of claim 1, wherein one of the at least one modification action of the textual language specification includes a data position for the data segment of the modification action, the data position at a data unit of a data word from the first sequence of the first network packet.

5. The circuit of claim 4, wherein the modification action of the textual language specification also includes a guard condition for enabling and disabling the modification action.

6. The circuit of claim 5, wherein a state, which corresponds to the pairing of the data word of the data position and a data word from the second sequence, of the state machine specifies the at least one next state that includes one of the states in response to the guard condition enabling the modification action and a different one of the states in response to the guard condition disabling the modification action.

7. A processor-implemented method for generating a circuit for converting a first network packet into a second network packet, the processor-implemented method comprising:

inputting a textual language specification that includes a plurality of modification actions, each modification action being one of an insertion action for inserting a data segment into the first network packet and a removal action for removing a data segment from the first network packet;

concurrently scanning through a first plurality of data units in a first sequence of data words in the first network packet and a second plurality of data units in a second sequence of data words in the second network packet, each data word in the first and second network packets including a same number of the data units, wherein the scanning through the first plurality of data units in the first sequence of data words is suspended throughout each insertion action of the modification actions, and the scanning through the second plurality of data units in the second sequence of data words is suspended throughout each removal action of the modification actions;

generating a respective one of a plurality of states of a state machine of the circuit for each pairing, which is encountered during the concurrently scanning, of a first data word from the first sequence and a second data word from the second sequence, the respective state for selecting the data units of the second data word from the data segment of each insertion action of the at least one

19

modification action and the data units of both the first data word and a prior data word to the first data word in the first sequence;

generating a look-ahead stage, an operation stage coupled to the look-ahead stage, an insert/remove stage that includes the state machine and is coupled to the operation stage, and an interleave stage that is coupled to the insert/remove stage; and

outputting a specification of the state machine, the look-ahead stage, the operation stage, the insert/remove stage, and the interleave stage in a hardware description language.

8. The processor-implemented method of claim 7, wherein the inputting of the textual language specification includes inputting the textual language specification that includes a data position and a data length for the data segment of each insertion action of the modification actions.

9. The processor-implemented method of claim 8, wherein the currently scanning through the first and second pluralities of data units in the first and second sequences of data words includes suspending the scanning through the first plurality of data units in the first sequence of data words between reaching the data position of the insertion action in the first plurality of data units in the first sequence of data words and subsequently reaching the data length of the insertion action in the scanning through the second plurality of data units in the second sequence of data words.

10. The processor-implemented method of claim 8, wherein the inputting of the textual language specification includes inputting the textual language specification that includes a guard condition for an insertion action of the modification actions, and for a respective state of the state machine corresponding to the pairing of the first data word from the first sequence and the second data word from the second sequence with the first data word corresponding to the data position for the data segment of the insertion action, the generating of the respective state includes generating the respective state for selecting, in response to the guard condition enabling the insertion action, the data units of the second word from the data segment of the insertion action and the data units of both the first data word and the prior word to the first data word in the first sequence, and generating the respective state for selecting, in response to the guard condition disabling the insertion action, the data units of the second word from the data units of both the first data word and the prior word to the first data word in the first sequence.

11. The processor-implemented method of claim 7, wherein the inputting of the textual language specification includes inputting the textual language specification that includes a data position and a data length for the data segment of each removal action of the modification actions.

12. The processor-implemented method of claim 11, wherein the currently scanning through the first and second pluralities of data units in the first and second sequences of data words includes suspending the scanning through the second plurality of data units in the second sequence of data words between reaching the data position of the removal action in the first plurality of data units in the first sequence of data words and subsequently reaching the data length of the removal action in the scanning through the first plurality of data units in the first sequence of data words.

13. The processor-implemented method of claim 7, wherein the currently scanning through the first and second pluralities of data units in the first and second sequences of data words includes terminating the currently scanning through the first and second pluralities of data units in the first and second sequences of data words in response to processing

20

all of the modification actions during the currently scanning through the first and second pluralities of data units in the first and second sequences of data words.

14. The processor-implemented method of claim 13, wherein for the state of the state machine corresponding to the pairing of the first data word from the first sequence and the second data word from the second sequence at the terminating of the concurrently scanning, the state specifies the state as a next state of the state machine.

15. The processor-implemented method of claim 7, wherein one of the modification actions of the textual language specification includes a data position for the data segment of the modification action and a guard condition for enabling and disabling the modification action, the data position at a data unit of the first data word from the first sequence for one of the states of the state machine.

16. The processor-implemented method of claim 15, wherein the state of the state machine specifies a next state that is one of the states in response to the guard condition enabling the modification action and another one of the states in response to the guard condition disabling the modification action.

17. The processor-implemented method of claim 7, further comprising tracking each guard condition of the modification actions during the concurrently scanning through the first plurality of data units in the first sequence of data words and the second plurality of data units in the second sequence of data words, the tracking of the guard condition including determining the guard condition cannot enable the modification action that includes the guard condition in response to the guard condition of another modification action already being one of enabled and disabled.

18. The processor-implemented method of claim 7, further comprising sorting the modification actions from the textual language specification into an order of the scanning through the first plurality of data units in the first sequence of data words in the first network packet.

19. A program storage medium, comprising:

a processor-readable device configured with instructions for generating a circuit for converting a first network packet into a second network packet, wherein execution of the instructions by one or more processors causes the one or more processors to perform operations including, inputting a textual language specification that includes a plurality of modification actions, each modification action being one of an insertion action for inserting a data segment into the first network packet and a removal action for removing a data segment from the first network packet;

concurrently scanning through a first plurality of data units in a first sequence of data words in the first network packet and a second plurality of data units in a second sequence of data words in the second network packet, each data word in the first and second network packets including a same number of the data units, wherein the scanning through the first plurality of data units in the first sequence of data words is suspended throughout each insertion action of the modification actions, and the scanning through the second plurality of data units in the second sequence of data words is suspended throughout each removal action of the modification actions;

generating a respective one of a plurality of states of a state machine of the circuit for each pairing, which is encountered during the concurrently scanning, of a first data

21

word from the first sequence and a second data word from the second sequence, the respective state for selecting the data units of the second data word from the data segment of each insertion action of the at least one modification action and the data units of both the first data word and a prior data word to the first data word in the first sequence; 5
generating a look-ahead stage, an operation stage coupled to the look-ahead stage, an insert/remove stage that

22

includes the state machine and is coupled to the operation stage, and an interleave stage that is coupled to the insert/remove stage; and
outputting a specification of the state machine, the look-ahead stage, the operation stage, the insert/remove stage, and the interleave stage in a hardware description language.

* * * * *