Title: SYSTEM AND METHOD FOR DESIGNING AND IMPLEMENTING PACKET PROCESSING PRODUCTS

(57) Abstract: A system and method for allowing a user to create instructions for building a packet processing integrated circuit. The system includes a user interface for allowing a user to define a desired packet processing algorithm (4) using a plurality of discrete packet processing blocks (22, 24, 28, 30), each of the blocks corresponding to a portion of the desired packet processing algorithm (4). The system allows the user to define connections (10) between the plurality of packet processing blocks (22, 24, 28, 30) and the connections to provide a list of instructions in a hardware description language for producing an integrated circuit capable of executing the desired packet processing algorithm (19).
SYSTEM AND METHOD FOR DESIGNING AND IMPLEMENTING PACKET PROCESSING PRODUCTS

SPECIFICATION

BACKGROUND OF THE INVENTION

FIELD OF THE INVENTION

The present invention relates to digital component design and implementation systems and, more particularly, to a system and method for designing and implementing packet processing products.

RELATED ART

Computer-based communications are dominated by the transmission of packets of data. Typically, a packet contains a payload, i.e., a portion of an overall data message, surrounded by a number of header bits or bytes, that are used to insure that the payload is transmitted and received without error. The header bits or bytes can be divided into a number of fields designating commands, responses, packet characteristics, etc. The fields can take on one or more values depending on the particular protocol used. Some protocols are custom-designed, while others, such as asynchronous transfer mode (ATM) or Transmission Control Protocol/Internet Protocol (TCP/IP), are standardized. For any type of protocol, there is a need to extract and examine the header bits or bytes to make decisions as to how to classify a type of packet, where to route the packet, and whether to drop or temporarily store (queue) the packet for future processing. The header must be parsed, bits or bytes examined or processed, and then routing decisions must be made.

Various hardware and software products have, in the past, been developed for designing and implementing products for processing and classifying data packets. In one approach, parsing, decision, and routing functions are implemented in software modules executed by the host processor and memory of the receiving computer. Processing large amounts of data in real time is often slow, since doing so puts a strain on processor resources. A second approach is to
use a specialized microprocessor and associated hardware, called a network processing unit (NPU). The NPU provides a programmable interface for programming nearly any type of protocol functionality. However, the ability to program nearly every aspect of a transmission packet protocol burdens an NPU with a large amount of functionality, rendering an NPU both expensive and slow (low data rates). Also, the time needed for a developer to program an NPU may take several hours to days, which can be cost prohibitive. Another approach is to design a customized application specific integrated circuit (ASIC). This approach often wastes large numbers of gates to achieve only limited functionality, and is thus not cost effective. As such, there is a lack of an adequate system or methodology for designing and implementing packet parsing and classification products, wherein such products can be designed and implemented.

Accordingly, what would be desirable, but has not yet been provided, is a system and method for designing and implementing packet processing products which addresses the foregoing limitations.
SUMMARY OF THE INVENTION

The present invention relates to a system and method for designing and implementing packet processing products, wherein a user can create instructions for building a packet processing integrated circuit. The system includes a user interface for allowing a user to define a desired packet processing algorithm by defining a plurality of discrete, packet processing blocks, each of the blocks corresponding to a portion of the desired packet processing algorithm, as well as connections between the plurality of packet processing blocks. The system processes the plurality of packet processing blocks and the connections to provide a list of instructions in a hardware description language for producing an integrated circuit capable of executing the desired packet processing algorithm. The list of instructions can be delivered to a customer, or the customer can be provided with an integrated circuit constructed using the list of instructions. The customer can also be provided with a NETLIST generated using said list of instructions.

The packet processing blocks of the present invention include a Packet Processing Unit (PPU), a Packet Modification Unit (PMU), and a Decision and Forwarding Unit (DFU). The PPU includes functionality for extracting a header of a packet; for pointing to a portion of the header of a predetermined width using a predetermined index of a bit location in the header; for comparing the data represented by the portion of the header with at least one predetermined value; and for declaring a match when the result of the comparison is true. A variation of a PPU, called a PPUX, includes functionality for accessing an external Content-Addressable Memory (CAM) or Random-Access Memory (RAM). The PMU includes functionality for extracting a packet; pointing to a portion of the packet of a predetermined width using a predetermined index of a bit location in the packet; and modifying the portion of the packet. A packet can be modified in one of three ways: deletion, insertion, or overwriting a portion of the packet. The DFU can perform one of drop, queue, and forwarding operations on packets coming from at least one PPU, PPUX, or PMU. The PPU, PPUX, PMU, and DFU can be programmed by an external microprocessor.
Further features and advantages of the invention will appear more clearly on
a reading of the detailed description of an exemplary embodiment of the invention,
which is given below by way of example only with reference to the accompanying
drawings.
BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention, reference is made to the following detailed description of an exemplary embodiment considered in conjunction with the accompanying drawings, in which:

**FIG. 1** is a flowchart showing a process according to the present invention for designing a packet processing product;

**FIG. 2A** is a screen shot of a window in a graphical user interface (GUI) according to the present invention for choosing a type of packet processing block to be configured;

**FIG. 2B** is a screen shot of a window in a graphical user interface (GUI) for selecting configuration parameters for generating a Packet Processing Unit (PPU) of the present invention;

**FIG. 2C** is a screen shot of a window in a graphical user interface (GUI) for selecting configuration parameters for generating a Packet Modification Unit (PMU) of the present invention;

**FIG. 2D** is a screen shot of a window in a graphical user interface (GUI) according to the present invention for selecting configuration parameters for generating a Decision and Forwarding Unit (DFU) of the present invention;

**FIG. 3** is a block diagram of a plurality of packet processing blocks according to the present invention for designing a packet processing product;

**FIG. 4** is a block diagram showing, in greater detail, a Packet Parsing Unit (PPU) of the present invention;

**FIG. 5** is a block diagram showing, in greater detail, a Packet Parsing Unit with an external interface to a CAM/RAM (PPUX) of the present invention;

**FIG. 6** is a block diagram showing, in greater detail, a Packet Modification Unit (PMU) of the present invention;

**FIG. 7** is a block diagram showing, in greater detail, the Decision and Forwarding Unit (DFU) of the present invention; and

**FIG. 8** is a block diagram showing a sample packet processor design for determining the queuing precedence of a VLAN/non-VLAN frame.
DETAILED DESCRIPTION OF THE INVENTION

Referring to FIG. 1, a process according to the present invention for designing packet processing products is shown. The present invention allows a user to design packet processing products using a high-level programming language which generates a NETLIST for generating a hardware design specification of a digital circuit. A NETLIST describes the connectivity of an electronic design. The design process begins at step 1, wherein a set of user requirements and specifications are received, which may be in the form of a packet parsing architecture or a packet parsing and classification algorithm. Typically, these requirements are in the form of a text description of the system to be generated. At step 2, the description is translated by the user or provider into a textual or graphical design using packet processing blocks which include Packet Parsing Units (PPU), Packet Parsing Units with an external interface to a CAM/RAM (PPUX), Packet Modification Units (PMU), and Decision and Forwarding Units (DFU), which will be described hereinbelow with reference to FIGS. 3-7.

As an example of step 2, if the customer needs a firewall that accepts TCP packets and rejects UDP packets, then three PPUs and one DFU are required. One of the PPUs is devoted to determining a source IP address; a second PPU is devoted to extracting a destination IP address; and a third PPU is devoted to distinguishing between TCP and UDP packets. The three PPUs are connected in parallel (since the information can be extracted simultaneously from the same packet), and the "match" outputs of the PPUs (to be described with reference to FIG. 4) and a source packet is forwarded to a DFU. Once the source and destination addresses are extracted from the packet and the type of packet is extracted, the DFU takes each match input and the packet and makes a decision: If the packet is a TCP packet and the source and destination addresses are allowed, then the packet is passed on, otherwise the packet is to be dropped. Thus, in step 2, the user can select the required number and combination of packet processing blocks to be used in the design.
At step 3, the packet processing block requirements, including their required inputs and outputs, are entered into a connection document, which can be a text based EXCEL™ spreadsheet or a VISIO™ block diagram. Typical inputs to the connection document include entries for each PPU and DFU block, which may include an index representing the point of entry into a packet to be processed, and whether a lookup in an internal table of data in a PPU is required.

Once the connection document has been completed, then, at step 4, packet processing blocks, e.g., each PPU and DFU, can be configured. Configuring a packet processing block involves taking a "default" packet processing block file, such as a generic PPU or DFU file, and modifying portions of it and setting variables within each file. Code for the packet processing blocks to be described in FIGS. 4-7 (written in pseudo-code) can be found in Appendices A-E and G-L attached hereto. In particular, the pseudo-code for the PPU calls code found in the following appendices: a file for describing a generic header extraction block called a Hardware Lookup Unit (HLU) (see Appendices D and K), and a file for describing a generic Match/Lookup Unit (MLU) (see Appendices E and L). Both the HLU and MLU will be described hereinbelow as part of the description of the PPU. The packet processing blocks are implemented in a hardware design language (HDL) which models digital circuits, with gates, flip flops, counters, and other logic in a C-like software language. In some implementations, the "pruning" process can be performed by manually copying and editing a maximally configured processing block file, or by applying a preprocessor in the form of shell scripts to cull code from and substitute variables within a maximally configured processing block files. Preprocessing shell scripts, as is known in the art, can include textual or graphically-based user prompts for answering questions about specific parameters desired by the user for a particular block.

FIGS. 2A-2D show one possible example of graphical user interface (GUI) which can be used to enter parameters for packet processing blocks. A Main generation GUI window 13 is presented to the user, as shown in FIG. 2A. One of a number of radio buttons 13 is selected by the user to indicate the type of
processing block to be configured. Depending on the processing block chosen, a configuration window 15 is displayed, one for each type of processing block (i.e., PPU/PPUX (see FIG. 2B); PMU (see FIG. 2C); and DFU (see FIG. 2D)). Each configuration window 15 contains a field 16 for naming the processing block. A series of configuration screen elements 17 are presented to the user for allowing parameters of each processing block to be specified by the user (including, e.g., data bus width, start of packet width, end of packet width, maximum header words, qualifier width, result width, result expression, external memory parameters, number of interfaces, etc.), and which may vary according each type of processing block. Finally, the user can click on either a "Generate" button 18 to cause the particular processing block code to be generated, or a "Cancel" button 19.

The GUI code can pass the input parameters to a preprocessor, such as a preprocessor called "veriloop2." The pseudo-code for veriloop2 can be found in Appendix F. Veriloop2 first performs substitutions into appropriate variables using the parameters passed from the GUI. Veriloop2 then searches for constructs such as name-value pairs, conditional constructs, and loops having a particular syntax, and then culls the maximally configured packet processing block file to produce a preprocessed header-like library files, each containing a function or class representing a particular PPU, DFU, etc. Pseudo code for types of preprocessor constructs can be found in Appendix G. Pseudo code for sample pre-processed files of FIG. 8 can be found in Appendices H-L. Note that there is only one PPU/MLU/HLU file for all three PPUs, which share the same number of inputs/outputs and share the same general structure. The number of PPUs that need to be generated depends upon the degree of parallelism needed for a particular design. If all the operations for a number of PPUs can be performed in series, then one PPU is needed, since all that changes between instances of PPUs is the input parameters (e.g. opcode, mask, etc.). There is one generated PPU for each parallel operation. There are separate DFU Appendices (i.e., Appendices B, H, and I because each DFU can have a different number of inputs/outputs).
The present invention distills the implementation of maximally configured processing blocks into common sub-blocks which have unique names (e.g., PPIM, DFU_2) or modules which have inputs and outputs that can be interconnected in such a way as to perform all of the functions necessary for implementing a desired packet processing product. The common blocks described herein are preferably instantiations of packet processing blocks written in VHDL, Verilog, or System C, but other suitable hardware description languages can be used. The software implementation of packet processing blocks is platform independent, and can be written in a platform independent language such as JAVA. As such, packet parser/classifier functionality of the present invention can run both in Windows and in different versions of the Unix operating system, as well as others. In a GUI, the programmer/designer can invoke instances of these common modules using a C-like application programming interface (API) surrounded by other C-like code for interconnecting the sub-blocks.

At step 5, integration is performed. Integration involves declaring instantiations of each processing block by name, and making connections between instantiated packet parsing blocks in a top-level main program file (the top-level main program file is similar to the file containing the main() function call in C language). These connections are called "wires" or "signals" which are declared like variables, and associations are made between two processing block instances which have a common wire. For example, signal "x" in PPU1 ties to signal "y" in the top level file. Signal "z" of DFU1 also ties to signal "y" in the top level file. In this way, signal "x" of PPU1 is tied to Signal "z" of DFU1 which may also be tied to one or more other signals. Certain input parameters can also be "hard-coded" within the top-level file.

At this point, all source HDL code has been generated which together can constitute a fully designed product. At step 6, if the customer desires only the design, then at step 7, the generated packet processing block files and the top level file can be delivered to the customer. If the customer desires to have a NETLIST, then at step 8, the generated files are run through a commercially-available
synthesis tool, as is known in the art. Sample synthesis tools include Design Compiler from Synopsis, Precision Synthesis from Mentor Graphics, Simplify from Synplicity, or XST from Xilinx. The synthesis tool behaves like an optimizing compiler which produces a NETLIST for producing an electrical schematic for a custom integrated circuit which is implemented with a minimum number of logic gates, flip-flops, counters, etc. The type of NETLIST generated depends on whether the customer desires to have a foundry-specific device, e.g. a Xilinx FPGA or a generic ("virtual") NETLIST which is not specific to a particular vendor's product. Customers which are EDA (electronic design automation) vendors desire a non-specific NETLIST. The NETLIST could be a foundry-specific or "virtual" bitstream or binary file that is delivered to customer.

At step 9, if the customer does not desire to have a digital integrated circuit delivered to them, then at step 10, the NETLIST is delivered to the customer, otherwise, at step 11, the NETLIST is run through a place and route program, which physically constructs the gates defined in the NETLIST on a silicon die and interconnects them. The choice of a place and route tool depends on whether the packet parser/classifier is to be implemented as an ASIC (fixed logic) or an FPGA (programmable logic). Sample place and route programs include Quartus II from Altera and ISE from Xilinx. At step 12, the integrated circuit is delivered to the customer.

With reference to FIG. 3, a block diagram of a graphical design environment using packet processing blocks according to the present invention for designing a packet processing product, indicated generally at 20, is depicted. The blocks 20 can be implemented in a text-based or graphical design environment. The environment 20 includes combinations of any number of Packet Parsing Units (PPUs) 22, PPUXs 24 (which are PPUs that can access CAM/RAM memory 26), Packet Modification Units (PMUs) 28, and Decision and Forwarding Units (DFUs) 30. The PPUs 22, PPUXs 24, PMUs 28, and DFUs 30 can be connected by a designer in a variety of ways to create parsing/classification logic for any desired packet processing algorithm. The PPUs 22 operate on packet headers 21.
packet itself can be passed through the environment 20 intact. Alternatively, only
the packet header 21 is passed through the environment, which requires the
creation and passing of a pointer to the packet data to be output after the DFUs 30.
The packets are stored in memory upon arrival and retrieved from memory upon
departure. A copy of the header 21 and a pointer to the packet location is passed
to the development environment 20. The length of the copied header 21 is
variable. It starts at a programmable position in the header 21 and ends at the last
field that must be processed. A PPU takes a header 21 and can seek, i.e., locate,
any field of constant or variable length. Once the field is found in the header 21,
the PPU 22 can perform a check on that field, such as whether the field is equal to
or greater than a given value, or matches a particular value, and then output that
value depending on the operation performed.

PPUXs 24 are PPUs that can perform lookups or searches using external
random-access memories (RAMs) or CAMs (a CAM is defined as a RAM-like
memory which can determine whether an input value is present in the memory
device). A PMU 28 is a PPU which allows fields in the header of a packet or the
packet itself to be modified by means of insertions, deletions, or substitution of
bytes. In contrast, the PPUs 22 and PPUXs 24 only allow the fields of a packet
header to be examined. Any number of PPUs 22, PPUXs 24, and PMUs 28 can be
chained together in series or in parallel to implement complex expressions. The
DFUs 30 combine the output of one or more PPUs 22 and/or PPUXs 24 and/or
PMUs 28 using a programmable condition, and then forward the header to one of a
plurality of outputs. The outputs can represent Boolean True and False values, and
decisions as to whether to drop, forward, or queue the packet. The DFUs 30 make
decisions to forward, drop, or enqueue packets based on the results from the PPUs
22. For example, the output of the last DFU in the chain, such as the DFU labeled
"A", can be a queue ID, i.e. of the queue implemented in an external traffic manager
31.

The traffic manager 31 is a device which performs a set of actions and
operations for a network to guarantee the operability of the network. Traffic
Management (TM) is exercised in the form of traffic control and flow control. In the context of the present invention, the traffic manager 31 operates on a packet stream once the classification & processing is done on a packet (i.e. once it passes from PPU/DFU blocks). For example, PPU/DFU blocks are used to figure out the priority number of a packet. The traffic manager is given that priority number and the packet to do a traffic control operation to guarantee that high priority packets pass before low priority packets.

With reference to FIG. 4, a block diagram of the PPU 22 is depicted. The PPU 22 performs basic parsing of the packet header 21 and may perform mathematical/logical operations on the parsed fields of packet header 21. The PPU 22 includes a plurality of inputs and outputs 32-83. The function of each input and output 32-83, as well as the values that each input or output handle, are described with reference to Table 1 hereinbelow.
<table>
<thead>
<tr>
<th>Signal</th>
<th>Ref.</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk</td>
<td>32</td>
<td>In</td>
<td>Clock input</td>
</tr>
<tr>
<td>Rst</td>
<td>34</td>
<td>In</td>
<td>Asynchronous reset (Active high)</td>
</tr>
<tr>
<td>DataIn</td>
<td>36</td>
<td>In</td>
<td>Packet header data input</td>
</tr>
<tr>
<td>SOHIn</td>
<td>38</td>
<td>In</td>
<td>Start of header input (Active High)</td>
</tr>
<tr>
<td>EOHIn</td>
<td>40</td>
<td>In</td>
<td>End of header input (Active High)</td>
</tr>
<tr>
<td>InVal</td>
<td>42</td>
<td>In</td>
<td>Data In valid indication (Active High)</td>
</tr>
<tr>
<td>DataOut</td>
<td>44</td>
<td>Out</td>
<td>Packet header output</td>
</tr>
<tr>
<td>SOHOut</td>
<td>46</td>
<td>Out</td>
<td>Start of header output (Active High)</td>
</tr>
<tr>
<td>EOHOut</td>
<td>48</td>
<td>Out</td>
<td>End of header output (Active High)</td>
</tr>
<tr>
<td>OutVal</td>
<td>50</td>
<td>Out</td>
<td>Data Out valid output (Active High)</td>
</tr>
<tr>
<td>Qual/Enb</td>
<td>52</td>
<td>In</td>
<td>Qualifier/Enable input that is checked using the Qualifier Condition 54 below to enable the PPU 22 on a packet by packet basis. The Qual/Enb 52 can be the Result 70 from a previous PPU.</td>
</tr>
<tr>
<td>QualCond</td>
<td>54</td>
<td>In</td>
<td>Qualifier Condition: The PPU operation is enabled if the result of the check of the Qual/Enb input 52 using the Qualifier condition 54 is true. The Qualifier Condition 54 can be: Always True, Equal, Less Than, Less Than or Equal, Greater Than, Greater Than or Equal, etc.)</td>
</tr>
<tr>
<td>Index</td>
<td>56</td>
<td>In</td>
<td>Index that points to a byte position in the header relative to the start of packet. The first byte in the header has an Index of 0.</td>
</tr>
<tr>
<td>Width</td>
<td>58</td>
<td>In</td>
<td>Width of the field to be operated on</td>
</tr>
<tr>
<td>Mask</td>
<td>60</td>
<td>In</td>
<td>The Mask value is ANDed with the data to be operated on. This allows checking of only</td>
</tr>
<tr>
<td>Signal</td>
<td>Ref.</td>
<td>Type (In/Output)</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Opcode</td>
<td>62</td>
<td>In</td>
<td>The Opcode specified the operation to be performed. The opcodes are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- EQ: Equal to Param1 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- LT: Less Than Param1 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- LE: Less Than or Equal to Param1 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- GT: Greater Than Param1 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- GE: Greater Than or Equal to Param1 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- RNG: Check if within range &lt;Param1 64, Param2 66&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- LUP: Look up</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- SPCL: Special programmable expression which can use PARAM1 64, PARAM2 66, four special purpose registers provisioned in PPU 12, Index 56, Width 58, and Qualifier/Condition 54.</td>
</tr>
<tr>
<td>Param1</td>
<td>64</td>
<td>In</td>
<td>Most opcodes use the parameter Param1</td>
</tr>
<tr>
<td>Param2</td>
<td>66</td>
<td>In</td>
<td>The RNG opcode uses Param2 to indicate the end of the range</td>
</tr>
<tr>
<td>Match</td>
<td>68</td>
<td>Out</td>
<td>Match is asserted (high) if the result of the operation is true</td>
</tr>
<tr>
<td>Result</td>
<td>70</td>
<td>Out</td>
<td>The Result output is controlled by a logical or arithmetic expression on any of the inputs. For example, to output a Result that drives the Index input 56 of the next PPU so that it points to a data field that is 2 bytes ahead:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result = Index + 2</td>
</tr>
<tr>
<td>ResVal</td>
<td>72</td>
<td>Out</td>
<td>Indication that Result 70 is valid</td>
</tr>
<tr>
<td>SeqOut</td>
<td>74</td>
<td>Out</td>
<td>Sequence number used for synchronization between PPUs and a DFU. This value</td>
</tr>
<tr>
<td>Signal</td>
<td>Ref #</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MapWrRd*_n</td>
<td>76</td>
<td>In</td>
<td>Map write enable is used to program internal registers. Active high is writing, active low for reading</td>
</tr>
<tr>
<td>MapAddr</td>
<td>78</td>
<td>In</td>
<td>16 Address locations are provisioned for following usage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0 = PPU ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x1 = Qualifier enable condition value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x2 = Address to program internal Lookup table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x3 = Data to program internal Lookup table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Note: The address needs to be written first followed by data)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x4-0x7: Used for special purpose registers which can be used in any equation for the special Operation code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x8-0xF: For future use</td>
</tr>
<tr>
<td>MapWrData</td>
<td>80</td>
<td>In</td>
<td>Write data for the PPU map</td>
</tr>
<tr>
<td>MapRdData</td>
<td>81</td>
<td>Out</td>
<td>Read back data from the PPU map</td>
</tr>
<tr>
<td>SeqIn</td>
<td>82</td>
<td>In</td>
<td>An optional externally defined sequence number. This may be used in place of an internally generated sequence number for a PPU.</td>
</tr>
<tr>
<td>TAG</td>
<td>83</td>
<td>In</td>
<td>An optional user defined label to be associated with the packet header</td>
</tr>
</tbody>
</table>

**TABLE 1**

The terms in brackets in FIG. 4 accompanying a specific input or output represents the bit width of the input or output, in standard HDL syntax. For example, if the input Dataln 3 6 is to be 32 bits wide, then the variable DW is set to
32 such that $\text{Dataln}[36]$ is expressed in an HDL file as $\text{Dataln}[\text{DW}-1:0] = \text{Dataln}[32-1:0] = \text{Dataln}[31:0]$, where "$31$" represents the last bit and "$0\) represents the first bit.

The input Clk 32 is supplied from external hardware, such as the clock of a microprocessor. The input Rst 34 is used to cause the PPU to go into a pre-defined state where most internal variables and outputs are set to an initial value. This condition is usually needed at power-up of the hardware in logic systems to stabilize the system before execution of a packet processing algorithm. The system is initially Reset. A predetermined amount of time later, when it is known that all circuits have stabilized, then the circuit is put into operation by toggling Rst 34.

The PPU 22 includes a Hardware Lookup Unit (HLU) 84, a Delay/FIFO module 86 containing an optional Delay Line 88 or a FIFO 90, a Match and Lookup Unit (MLU) 92, Result Generation (process) 94, Sequence Generation (process) 96, an Output Alignment (process) 98, interconnected as shown. The sub-blocks 84-98 are implemented as modules or processes. A module is similar to a class or subclass in an object-oriented language like C++, while a process is similar to a function. The PPU also contains (not shown) a predetermined but limited number of internal general-purpose registers for storing and retrieving values for comparisons, lookups, etc.

A stream of data is continuously presented to the input Dataln 36 of the HLU 84. No data of the input stream is stored in a memory. In such circumstances, it is the job of the HLU 84 to extract information from a packet and present that information to the other blocks of the PPU 22. The HLU 84 takes a snapshot of the data stream according to the location in the data stream specified by the inputs Index 56 and Width 58. The inputs SOHIn 38, EOHIIn 40, and InVal 42 allow for fine tuning of locating data from the output of other PPUs, PPUXs, PMUs, or external hardware. SOHIn 38, EOHIIn 40, and InVal 42 tell the PPU 22 how to delimit data a packet header. SOHIn 38 tells the hardware where packet starts and EOHIIn 40 tells the hardware when a packet header ends. Once the packet starts, then at
every clock cycle, the data presented at Dataln 36 is either valid or invalid, as indicated by the input InVal 42. The extracted header bits are present as an output CompDat 100 and as an input to the MLU 92. CompDat 100 stands for the data that needs to be compared in the MLU 92.

The Delay/FIFO module 86 is used to synchronize the outputs of the PPU 22 to be presented to a subsequent block, such as a DFU. The Delay/FIFO module 86 is needed because the inputs to the PPU, such as Dataln 36, along with the control input signals SOHln 38, EOHln 40, and InVal 42, need to be aligned in time in the Output Alignment process 98 with intermediate outputs of other sub-blocks of the PPU 22, such as the Match output 110 of the MLU 92, which may be delayed relative to the inputs due to delays in processing within the MLU 92. The MLU 92 performs its decision making (e.g., a comparison of a bit within Dataln 36 with a user specified parameter (Parami )) without full packet storage. Therefore, Dataln 36 along with the control input signals SOHln 38, EOHln 40, and InVal 42 are pipelined to the Result Generation process 94 and the Output Alignment process 98 by way of intermediate I/O Val_i 102, SOH_i 104, EOH_i 106, and Data_i 108. There are fixed delays (measured in clock cycles) associated with processing in the in Result Generation process 94 and the MLU 92. There is a variable delay associated with the HLU 84 depending upon value of Index 56. The inputs described above must be delayed in the Output Alignment process 98 by the sum of the aforementioned individual delays. For example, if Index 56 is 8, then CompDat 100 is received at the MLU 92 eight clock cycles after Dataln 36 arrives at the PPU 22. If the MLU 92 processes CompDat 100 in three clock cycles, then the PPU 22 inputs need to be delayed by 8 + 3 clock cycles in the Output Alignment process 98. The choice of the optional Delay Line 88 or the FIFO 90 depends on the size of the delay needed. A FIFO always works but requires using scarce memory in the PPU 22. Thus, if only a few clock cycles worth of delay up to about 16 clock cycles are needed, then the Delay Line 88 is used, otherwise the FIFO 90 is used.
The MLU 92 performs the bulk of the packet parsing and classification operation to be performed on one unit of a packet processing algorithm. The MLU 92 is programmable, i.e., it can compare the data/fields extracted in the HLU 84 with values stored in internal registers by means of the inputs Opcode 62, Parami 64, Param2 66, and Mask 68 and declares a match or no match which appears on the internal output Match 110, which, in turn, appears as an output of the Result Generation process 94. The inputs QualEnb 52 and QualCond 54 enable or disable the MLU 92 depending on certain conditions. The operation to be performed in the MLU 92 are enabled if the result of the check of the QualEnb 52 using the QualCond 54 is true. QualEnb 52 is a value stored in a qualEnb register (not shown) which is user programmable through an address map. The Qualifier Condition 44 can be: Always True, Equal, Less Than, Less Than or Equal, Greater Than, Greater Than or Equal, etc.

For example, if the user desires only to allow IPV6 packets, then QualEnb 52 can be programmed through the qualEnb register (not shown) to be the value 6. QualCond 54 is set to Equal To (EQ). The packet type is retrieved from a mode register from an external CPU. If the packet type is 6 (IPV6), then the MLU 92 is enabled; if the packet type is 4 (IPV4), then the MLU 92 is disabled, and no comparison takes place. If it is desired to have all types of IP packets, then QualCond 54 is set to Less Than or Equal (LE) or Always True.

The match/no-match functionality of the MLU 92 is performed on the portion of the DataIn 36 packet header pointed to by Index 56 and Width 58. Additional inputs Mask 60, Opcode input 62, Parami 64, and optionally Param2 66 are needed to perform the comparison/match/no-match operation. The MLU 92 performs a seek and operation function.

The seek function finds a data field in a packet header (not shown) based on an offset from the start of the packet header indicated by the input Index 56. If Index 56 is 0, then the first byte of the packet header is indicated. An Index 56 of six indicates the seventh byte from the beginning of the packet header. The interconnections that can be made to the Index input 56 include a fixed value (e.g.
4), a value stored in an internal user defined control register, or the result output 70 of another PPU, PMU, or DFU. If the Index input 56 is driven from another PPU, PMU, or DFU, the value placed on the Index input 56 is variable, depending on the condition(s) evaluated in the previous PPU, PMU, or DFU.

The operation function performs a check, an extraction, or a lookup on "Data_Field", which is the contents of the packet header pointed to by the Index input 56 of width equal to the value in bits placed on the Width input 58. The general expression of the operation is

\[
\text{Op(Data\_Field AND Mask, Param1, Param2)}
\]

The Data_Field may be filtered (AND'ed) with the Mask input 60. Op" is one of the opcodes placed on the Opcode input 62 given the Param input 64, and optionally the Param2 input 66. The types of operations are shown in Table 2 below:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Param1</td>
<td>Equal: Check if the Data field is equal to Param1.</td>
</tr>
<tr>
<td>LT</td>
<td>Param1</td>
<td>Less Than</td>
</tr>
<tr>
<td>LE</td>
<td>Param1</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>GT</td>
<td>Param1</td>
<td>Greater Than</td>
</tr>
<tr>
<td>GE</td>
<td>Param1</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>RNG</td>
<td>Param1, Param2</td>
<td>In Range between Param1 and Param2</td>
</tr>
<tr>
<td>LUP</td>
<td>-</td>
<td>Look Up</td>
</tr>
<tr>
<td>SP</td>
<td>-</td>
<td>Search: or special operation for a PPU using PARAM1, PARAM2, four Special purpose registers provisioned in PPU, Index, Width &amp; Qualifier.</td>
</tr>
</tbody>
</table>

TABLE 2
For example, a single MLU can be programmed to check if an IP address less than 224.XX.XX.XX, by specifying the following values:

- Opcode = LT
- Parami = 224
- Index = Points to IP DA or SA and can be adjusted automatically for VLAN tagging using a PPU.

As another example, to point to the beginning of an Ethernet frame payload for both untagged and VLAN tagged frames:

- Index = 14 (Type/Length)
- Opcode: EQ
- Parami: 0x8100
- QualCond = True
- Match (True): Index = 20
- Match (False): Index = 16

The inputs MapWrRd_n 76, MapAddr 78, and MapWrData 80, and the output MapRdData 81 are used as the interface between an external microprocessor and the internal registers of the PPU 22 to allow for reading of and writing to the registers. The PPU 22, PPUX 24, PMU 28, and DFU 30 can contain a user defined number of internal registers for packet header manipulation either internally or via an external microprocessor. The opcodes LUP and SPCL can be used to directly manipulate data in internal registers.

The output Match 110 of the MLU 92 is fed to the input of the Result Generation process 94 to be described hereinbelow. The Match output 110 is True if the operation performed in the MLU 92 is True, or False otherwise. The Result Generation process 94 takes the Match output 110, the outputs of the Delay/FIFO module 86, and optionally a tag value present on TAG 83 and produces the result output iResult 112, which is fed as an input to the Output Alignment process 98 and ultimately is the output Result 70 of the PPU 22. The Result Generation process 94 also outputs iResVal 114, which indicates when iResult 112 is valid. This is needed as a handshaking device, since result generation can take more than a single clock
cycle. iMatch 116 is the value of Match 110 passed along from the MLU 92. Assuming the MLU 92 was enabled, iResult 112 can take on two values corresponding to the True or False evaluation of the operation performed in the MLU 92. The True/False result values can be fixed or an arithmetic or logical function of any of the PPU 22 inputs. The iResult output 112 is later passed through the Output Alignment process 98 to be described hereinbelow as Result 70, which can be used to drive a DFU input or any input of another PPU or a PMU. Result 70 can also be a complex expression that the user may want to program. This allows the Index 56, QualEnb 52, Opcode 62, or Param<1,2> 64, 66 inputs of a PPU to be driven with different values depending on the Result 70 output of other PPU.

The PPU 22 generates or forwards a sequence number using the Sequence Generation process 96. The sequence number can optionally come from an external process/hardware via the input SeqIn 82 and passed along to a DFU; otherwise sequence numbers are internally generated within a PPU 22 using the Sequence Generation process 96. The sequence number, which appears as an internal output iSeq 118, is passed through the Output Alignment process 98 to a DFU through the PPU output SeqOut 74. Sequence numbers are incremented sequentially for each use of a PPU and are used for internal synchronization of all the inputs of a DFU. Sequence numbers are needed because different PPU can present their output packet header data, match data, and results at different times. For example, one PPU may index at bit 0 of an incoming packet, in which case match output may appear at an input to a DFU after three clock cycles. If another PPU indexes on a VLAN type field, then index is set to block 5 or 6, which gives its results to the same DFU after 6 + 3 clock cycles. The DFU takes the matches packet headers, and sequence number from each of the PPU and arranges them in correct sequence to be described hereinafter.

The Output Alignment process 98 aligns all outputs to the start of packet (SOP) or the end of packet (EOP). This is done in order to provide proper delineation of the output signals of one PPU to the next PPU/PPUX/PMU/DFU. For
example, if PPU1 is connected to PPU2, and PPU1 operates either on an 802.3 Ethernet frame or an Ethernet type 2 frame, then PPU1 examines a byte field which is either 20 bytes or 40 bytes from the beginning of a packet header. Therefore, all outputs of PPU1 need to be aligned on SOP as a requirement for input to PPU2. As another example, some protocols use trailer insertion, e.g., inserting a checksum at the end of a packet. Therefore, outputs are aligned at EOP.

With reference to FIG. 5, a block diagram of a PPUX 24 is depicted. A PPUX 24 has the same I/O signals and sub-blocks as the PPU 22 except for additional I/O needed to access an external CAM/RAM 220. Elements illustrated in FIG. 5 which correspond to the elements described above in connection with the PPU 22 of FIG. 5 have been identified by corresponding reference numbers increased by one hundred. Unless otherwise indicated, both the PPU 22 and the PPUX 24 have the same construction and operation.

In a PPU, as mentioned earlier, there is a predetermined number of internal registers/memory which can be programmed by a user. A typical need for programmed memory is for performing a lookup of values by MLU 192. For example, if there is a need to compare Parami 164 to one hundred IP addresses, then internal memory is used. However, if the number of lookups and hence values to be stored in memory is on the order of thousands of bytes or more, then it may be necessary to store and retrieve these values to/from an external CAM/RAM 220.
With reference to FIG. 6, a block diagram of a Packet Modification Units (PMU) 28 is depicted. A PMU allows for modification, i.e., insertion, deletion, or replacement, of bytes in a packet, including both the header and payload data. The PMU 28 includes a Delay/FIFO module 300 containing an optional Delay Line 302 or a FIFO 304, a Modification Unit (MU) 306, a Result Generation process 308, a Sequence Generation process 310, and an Output Alignment process 312, interconnected as shown. These sub-blocks 300-312 are implemented as software modules or processes.

The inputs InVal 314, SOHIn 316, EOHIn 318, Dataln 320, Tagln 322, Rst 324, and Clk 326 have the same functionality as is found in the PPU 22 and the PPUX 24. The delay/FIFO module 300 can be used to synchronize the inputs InVal 314, SOHIn 316, EOHIn 318, Dataln 320, and Tagln 322 with the outputs of the Result Generation Process 308 and the outputs of the Modification Unit (MU) 306 as is done in the PPU 22, but it also provides a second function: to delay incoming packet data by an amount equal to the number of bytes that may be inserted into a packet in the Modification Unit 306. This delay is not needed for removing or overwriting data in a packet. As with the PPU 22, the choice of the optional Delay Line 302 or the FIFO 304 depends on the size of the delay needed.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Ref. in FIG. 5</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRdAddr</td>
<td></td>
<td>Out</td>
<td>Memory address output</td>
</tr>
<tr>
<td>XRdEnb</td>
<td></td>
<td>Out</td>
<td>Read signal (Active High)</td>
</tr>
<tr>
<td>XRdData</td>
<td></td>
<td>In</td>
<td>Read data from memory</td>
</tr>
<tr>
<td>XRdVal</td>
<td></td>
<td>In</td>
<td>Read data valid input (Active High)</td>
</tr>
</tbody>
</table>

TABLE 3
If only a few clock cycles worth of delay (a few words to be inserted) are needed, then the Delay Line 302 is used, otherwise the FIFO 304 is used. As with the PPU 22, InVal 314, SOHIn 316, EOHIn 318, and Dataln 320 are pipelined to the a Modification Unit (MU) 306 as the intermediate outputs Val_i 328, SOH_i 330, EOH_i 332, and Data_i 334.

Val_i 328 is also directed to the Result Generation Process 308. The Result Generation Process 308 has a different purpose from the one found in a PPU 22. The intermediate outputs iResVal (result valid) 358 and iResult (the result) 360 are not based on a field value, but reflect the number of bytes inserted. Like a PPU 22, iResult 360 becomes the output Result 378 which can be used as an input to another PPU/PPUX/PMU/DFU. It can also be a complex expression that the user may want to program. The Sequence Generation Process 310 with the optional SeqIn input 362 has the same functionality as in the PPU 22.

The Modification Unit (MU) 306 inserts/modifies/removes data as specified by a user. The MU 306 is specified at preprocessing time as one of an inserting type, modifying type, or removing type PMU. The type of operations performed by the input signals ByteOffset 336, ByteValid 338, and ByteData 340 are shown in Table 4 below:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Ref. in FIG.</th>
<th>Type (IN/OUT)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteOffset</td>
<td>336</td>
<td>IN</td>
<td>Byte Offset for insertion/deletion/modification starting at zero bytes from the beginning of the packet</td>
</tr>
<tr>
<td>ByteValid</td>
<td>338</td>
<td>IN</td>
<td>Valid (Active High) for the number of clock cycles needed to insert N bytes</td>
</tr>
<tr>
<td>ByteData</td>
<td>340</td>
<td>IN</td>
<td>The N bytes of data to be inserted or overwritten (not used for deletion)</td>
</tr>
</tbody>
</table>

TABLE 4
The inputs MapWrRd_n 342, MapAddr 344, and MapWrData 346, and the output MapRdData 348 provide a future programming interface for an external microprocessor to allow for the reading and writing from/to internal registers of the PMU 28 to, for example, dynamically program an MU to either insert, delete, or modify a packet at run time. ValM 350, SOH_i 352, and EOH_i 354 are passed after a delay intact from their corresponding inputs to the MU 306 to the Output Alignment process 312. The modified packet, represented as the intermediate input/output Data_i 356 is also presented to the Output Alignment process 312. The Output Alignment process 312 has the same purpose and functionality as found in the PPU or PPUX, i.e., aligning all intermediate outputs iSeq 362, iResVal 358, iResult 360, ValM_i 350, SOH_i 352, EOH_i 354 and Data_i 356 on either the start of packet (SOP) or the end of packet (EOP) to become the aligned outputs SeqOut 366, OutVal 368, SOHOut 370, EOHOut 372, DataOut 374, ResVal 376, Result 378, and TagOut 380.

With reference to FIG. 7, a block diagram of a Decision and Forwarding Unit (DFU) 30 is depicted. The DFU 30 performs drop, queue, or forward operations based on input from 1 to N PPU's, PPUXs, PMUs, or other DFUs. The DFU 30 includes a plurality of inputs and outputs 400-444. The function of each input and output 400-444, as well as the values each input or output can take on, are described with reference to Table 5 hereinbelow.
<table>
<thead>
<tr>
<th>Signal</th>
<th>Ref. #</th>
<th>Type</th>
<th>In/Out</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIn</td>
<td>400a-400n</td>
<td>In</td>
<td></td>
<td>Result from PPU 0 to N-1</td>
</tr>
<tr>
<td>MIn</td>
<td>402a-402n</td>
<td>In</td>
<td></td>
<td>Match from PPU 0 to N-1</td>
</tr>
<tr>
<td>RInSeq</td>
<td>404a-404n</td>
<td>In</td>
<td></td>
<td>Sequence Number from PPU 0 to N-1. The DFU matches the sequence number among all its input ports to ensure that it is operating on results for the same packet</td>
</tr>
<tr>
<td>RInVal</td>
<td>406a-406n</td>
<td>In</td>
<td></td>
<td>Result valid from PPU 0 to N-1 (Active High)</td>
</tr>
<tr>
<td>ROutAVal</td>
<td>408</td>
<td>Out</td>
<td></td>
<td>Result output port A valid (Active High)</td>
</tr>
<tr>
<td>ROutBVal</td>
<td>410</td>
<td>Out</td>
<td></td>
<td>Result output port B valid (Active High)</td>
</tr>
<tr>
<td>ROutDVal</td>
<td>412</td>
<td>Out</td>
<td></td>
<td>Result output port D valid (Active High)</td>
</tr>
<tr>
<td>ROut</td>
<td>414</td>
<td>Out</td>
<td></td>
<td>Result output. The result is based on the evaluation of a logical expression of the match and result inputs</td>
</tr>
<tr>
<td>SeqOut</td>
<td>416</td>
<td>Out</td>
<td></td>
<td>Sequence number output. This sequence number is output with the results corresponding to the input sequence number</td>
</tr>
<tr>
<td>DValIn</td>
<td>418a-418n</td>
<td>In</td>
<td></td>
<td>Data valid from PPU 0 to N-1</td>
</tr>
<tr>
<td>SOHIn</td>
<td>420a-420n</td>
<td>In</td>
<td></td>
<td>SOH from PPU 0 to N-1</td>
</tr>
<tr>
<td>EOHIn</td>
<td>422a-422n</td>
<td>In</td>
<td></td>
<td>EOH from PPU 0 to N-1</td>
</tr>
<tr>
<td>DataIn</td>
<td>424a-424n</td>
<td>In</td>
<td></td>
<td>Data from PPU 0 to N-1</td>
</tr>
<tr>
<td>DValOut</td>
<td>426</td>
<td>Out</td>
<td></td>
<td>Data valid output</td>
</tr>
<tr>
<td>SOHOut</td>
<td>428</td>
<td>Out</td>
<td></td>
<td>SOH output</td>
</tr>
<tr>
<td>EOHOut</td>
<td>430</td>
<td>Out</td>
<td></td>
<td>EOH output</td>
</tr>
<tr>
<td>DOut</td>
<td>432</td>
<td>Out</td>
<td></td>
<td>data output</td>
</tr>
</tbody>
</table>
Referring again to FIG. 7, the DFU 30 includes sub-blocks Latch 445a-445n, Data Selection MUX 446, Result Generation process 448, and Output Alignment process 450. The triangles within FIG. 7 are for blocking together intermediate outputs and do not themselves have inherent functionality. All sub-blocks are processes. Latch 445a-445n latches the incoming results, data, and other output signals coming from 0 to N-1 PPU/PPUXs/PMUs to be processed at a later time inside the DFU 30. The Latch 445a-445n are necessary since each PPU/PPUX/PMU may present packet data at different times. Four signals from each Latch 445a-445n, namely iDValln 452a-552n, iSOH 454a-554n, iEOH 456a-456n, and iData 458a-458n, corresponding to the latched inputs DValln 418a-418n, SOH 420a-420n, EOH 422a-422n, and Data 424a-424n, respectively, and

<table>
<thead>
<tr>
<th>Signal</th>
<th>Ref. #</th>
<th>Type (In/Out)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapWrRd_n</td>
<td>434</td>
<td>In</td>
<td>Map read/write enable is used to program internal registers. Active high is writing, active low for reading</td>
</tr>
<tr>
<td>MapAddr</td>
<td>436</td>
<td>In</td>
<td>16 Address locations are provisioned for following usage 0x0 = DFU ID 0x1-0xF: For future use</td>
</tr>
<tr>
<td>MapWrData</td>
<td>438</td>
<td>In</td>
<td>Write data for the DFU map</td>
</tr>
<tr>
<td>MapRdData</td>
<td>440</td>
<td>Out</td>
<td>Read back data from the DFU map</td>
</tr>
<tr>
<td>Clk</td>
<td>442</td>
<td>In</td>
<td>Clock input</td>
</tr>
<tr>
<td>Rst</td>
<td>444</td>
<td>In</td>
<td>Asynchronous reset (Active high)</td>
</tr>
</tbody>
</table>

TABLE 5

<table>
<thead>
<tr>
<th>Signal</th>
<th>Ref. #</th>
<th>Type (In/Out)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapWrRd_n</td>
<td>434</td>
<td>In</td>
<td>Map read/write enable is used to program internal registers. Active high is writing, active low for reading</td>
</tr>
<tr>
<td>MapAddr</td>
<td>436</td>
<td>In</td>
<td>16 Address locations are provisioned for following usage 0x0 = DFU ID 0x1-0xF: For future use</td>
</tr>
<tr>
<td>MapWrData</td>
<td>438</td>
<td>In</td>
<td>Write data for the DFU map</td>
</tr>
<tr>
<td>MapRdData</td>
<td>440</td>
<td>Out</td>
<td>Read back data from the DFU map</td>
</tr>
<tr>
<td>Clk</td>
<td>442</td>
<td>In</td>
<td>Clock input</td>
</tr>
<tr>
<td>Rst</td>
<td>444</td>
<td>In</td>
<td>Asynchronous reset (Active high)</td>
</tr>
</tbody>
</table>
representing together data signals from each PPU/PPUX/PMU, belong to groups, which are fed together to the Data Selection MUX 446. Likewise, four signals from each Latch 445a-445n, namely iRlnVal 459a-459n, iMln 460a-460n, iRln 462a-462n, and iRlnSeq 464a-464n corresponding to the latched inputs RlnVal 406a-406n, Mln 402a-402n, Rln 400a-400n, and RlnSeq 404a-404n, respectively, and representing together control/result signals from each PPU/PPUX/PMU, belong to groups, which are fed together to the Result Generation process MUX 448. The Data Selection MUX 446 selects one of the sets of N-1 data groups and forwards the data group to the output group which includes iDValOut 466, iSOHOut 468, iEOHOut 470, and iDOOut 472 as inputs to the Output Alignment Process 450. The Result Generation Process 448 has a similar purpose to that found in the PPU/PPUX, namely, generating a result iRout 482 which depends on the evaluation of a programmable logical expression which may depend on the value of the inputs Rln[0 - (N-1)] 400a-400n and/or Min [0 - (N-1)] 402a-402n. In addition, the evaluation of this complex logical expression can determine an output port to which the packet is to be routed, i.e., the pass along/queue outputs A and B, or the drop port D, represented as active high enabling intermediate outputs iROutAVal 476, iROutBVal 478, and iROutDVal 480. These outputs are passed along to the Output Alignment Process 450, which has the same purpose and function as the PPU 22, PPUX 24, and PMU 28. The intermediate outputs 466-482 become the DFU outputs DValOut 426, SOHOOut 428, EOHOut 430, DOut 432, SeqOut 416, ROutAVal 408, ROutBVal 410, and ROutDVal 412, and Rout 414, respectively.

With the addition of a group of external AND gates and control outputs ROutAVal 408, ROutBVal 410, and ROutDVal 412, the output DOut 432 is routed to one of three output ports: DOutA 484, DOutB 486, or DOutD 488. Typically, DOutA 484 and DOutB 486 can be used for normal output and DOutD 478 can be used for dropping a packet (not shown). Alternatively, DOutD 488 can be used as a third routing output port. For the normal ports DOutA 484 and DOutB 486, the packet is either forwarded to a destination, or another chain of PPUs/PPUXs/PMUs, or sent to a queue of a traffic manager.
As an example of the operation of the Data Selection MUX 446 and Result Generation process 448, if the DFU 30 has two PPU inputs DIn[O] and DIn[I], and two match inputs Min[0] and Min[1], then the following conditions exist:

Output packet to Port DOutA if Min[O] is True and Min[1] is True;
Output packet to Port DOutB if Min[O] is True and Min[1] is False; and
Output packet to Port DOutD if Min[O] is False and Min[1] is False.

The design environment of the present invention can be connected to a set of internal PPU/PPUX/PMU/DFU registers and programmed through a microprocessor interface. The operations that the microprocessor would perform are reads and writes to/from the registers. Table 6 below shows a sample interface for a microprocessor manufactured by Freescale, Inc. (formerly Motorola):

<table>
<thead>
<tr>
<th>Signal</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UP_CLK</td>
<td>In</td>
<td><strong>Clock</strong>: This is the clock for the µP interface.</td>
</tr>
<tr>
<td>UP_CS</td>
<td>In</td>
<td><strong>Chip Select</strong>: This active low signal enables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the core to respond to microprocessor cycles.</td>
</tr>
<tr>
<td>UP_RWn</td>
<td>In</td>
<td><strong>Read/Write</strong>: Read (high) / Write (low) signal</td>
</tr>
<tr>
<td>UPREADY</td>
<td>Out</td>
<td><strong>Ready</strong>: Active low signal asserted by the core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to indicate the successful transfer of read or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write data.</td>
</tr>
<tr>
<td>UP_A[15:0]</td>
<td>In/Out</td>
<td><strong>Address Bus</strong>: 16-bit address driven by the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>microprocessor to address the core registers.</td>
</tr>
<tr>
<td>UP_IRQ</td>
<td>Out</td>
<td><strong>Interrupt Request</strong>: Active low signal asserted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by the core to indicate that an event was</td>
</tr>
<tr>
<td></td>
<td></td>
<td>detected.</td>
</tr>
</tbody>
</table>

**TABLE 6**
The possible types of interconnections between DFUs and PPUs are numerous. Depending on the application, the control inputs of the PPUs or DFUs can be driven with fixed values (hardwired), from programmable registers, or from the outputs of other PPUs or DFUs. Table 7 shows the options for control signal connections, with some typical examples of standard packet processing:

<table>
<thead>
<tr>
<th>PPU Control Input</th>
<th>Connected To</th>
<th>Description/Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Qualifier</strong></td>
<td>Fixed Value</td>
<td>PPU always enabled</td>
</tr>
<tr>
<td></td>
<td>Register</td>
<td>Enable/disable under software control</td>
</tr>
<tr>
<td></td>
<td>Other PPU/DFU</td>
<td>Enable/disable conditionally depending on result from other PPU/DFU</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>Fixed Value</td>
<td>Index is fixed. Example: MAC Destination or Source Address</td>
</tr>
<tr>
<td></td>
<td>Register</td>
<td>Index is software programmable</td>
</tr>
<tr>
<td></td>
<td>Other PPU/DFU</td>
<td>Index depends on result from other PPU/DFU. Example: IP Destination Address for untagged or VLAN frames</td>
</tr>
<tr>
<td><strong>Width</strong></td>
<td>Fixed Value</td>
<td>Width is fixed. Example: MAC Destination or Source Address</td>
</tr>
<tr>
<td></td>
<td>Register</td>
<td>Width is software programmable</td>
</tr>
<tr>
<td></td>
<td>Other PPU/DFU</td>
<td>Width depends on result from other PPU/DFU. Example: IPv4 or IPv6 Address</td>
</tr>
<tr>
<td><strong>Mask</strong></td>
<td>Fixed Value</td>
<td>Mask value fixed or not used</td>
</tr>
<tr>
<td></td>
<td>Register</td>
<td>Mask value is software programmable</td>
</tr>
<tr>
<td></td>
<td>Other PPU/DFU</td>
<td>Mask value depends on result from other PPU/DFU</td>
</tr>
<tr>
<td><strong>Opcode</strong></td>
<td>Fixed Value</td>
<td>Opcode is fixed. Example: Equal</td>
</tr>
<tr>
<td></td>
<td>Register</td>
<td>Opcode is software programmable</td>
</tr>
</tbody>
</table>
Each PPU/PPUX/PMU/DFU is configurable at synthesis time using the parameters shown in Table 8:

<table>
<thead>
<tr>
<th>Other PPU/DFU</th>
<th>Opcode changes depending on result from other PPU/DFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Value</td>
<td>Parameter(s) value is fixed. Example: Check for fixed MAC Address</td>
</tr>
<tr>
<td>Register</td>
<td>Parameter(s) value is software programmable. Example: Check for programmable MAC Address.</td>
</tr>
<tr>
<td>Other PPU/DFU</td>
<td>Parameter(s) value depends on result from other PPU/DFU. Example: Check TTL field in IP packet</td>
</tr>
</tbody>
</table>
TABLE 8

With reference to FIG. 8, a block diagram is depicted showing a sample packet processing algorithm design using the present invention. In this example, the packet processing algorithm relates to extracting the precedence field of an IP
packet for a VLAN/Non-VLAN frame from a packet header 500 belonging to a packet 499. Pseudo code which implements the two DFUs and the three PMUs of FIG. 8 can be found in Appendix H-L. A top-level file for the example of FIG. 8, expressed in pseudo code, can be found in Appendix M. The precedence field is used as the QID of the queue into which the packet is to be stored in a traffic manager. The packet header 500 is fed to a Dataln input 502 of a PPU 504. The PPU 504 determines first whether the inputted packet header 500 belongs to a virtual LAN (VLAN) frame or a non-VLAN frame by pointing to byte 12 of the header (Index = 12) with a field width of 2 bytes. The operation to be performed is:

\[
\text{EQ(DataField} (\text{byte } 12, \text{ width } 2) \text{ AND Mask } = 0xFFFF, \text{ Parami } = 0x8100, \text{ Param2 } = 0)
\]

If packet header 500 points to a VLAN frame, then the Result output 506 of the PPU 504 is set to point to the location or offset in the packet header 500 of the IP address in a VLAN type frame, otherwise it points to the location in the packet header 500 of the IP address in a non-VLAN frame. This IP address is fed to the Index input 508, along with the header 500 to a second PPU 510. In the PPU 510, the most significant byte is checked and must be less than 224, signifying that the input IP address is valid. The operation to be performed is:

\[
\text{GE(DataField} (\text{byte } = \text{ MSB of IP address, width } = 1) \text{ AND Mask } = 0xFF, \text{ Parami } = 224, \text{ Param2 } = 0)
\]

The packet header 500 is then passed to the Din[0] input 512 of a DFU 514. If the DA field of the IP address is \(\geq 224.0.0.0\), then the packet is to be dropped by placing the header on the DOutD output 516 of an AND gate 518 connected to the DFU 514. Otherwise, the packet 499 is forwarded to a third PPU 520 with the Index input 522 of the PPU 520 pointing to the "type of service" field (ToS) in the header 500 based on whether the packet 499 belongs to a VLAN or non-VLAN
frame. The ToS tells the application how a datagram should be used, e.g. delay, precedence, reliability, minimum cost, throughput etc. Depending on the value of the ToS field, one can change a priority assigned to a packet which is then sent to a traffic manager which processes the packet based on the set priority.

In the PPU 520, the IP precedence field is extracted from the header 500 with the following operation:

$$\text{EXTR}(\text{Data\_Field}(\text{byte} = \text{ToS field location}, \text{width} = 1) \text{ AND Mask} = \text{OxFF}, \text{Param1} = 2 \text{ (start)}, \text{Param2} = 3 \text{ (len)})$$

The IP precedence field is fed to the Din[0] input 524 of a second DFU 526. The DFU 526 places the packet header on the DOutA output 528 of an AND gate 530 for queueing, and the precedence field is placed on the DOutB output 532 of an AND gate 534. The precedence field functions as the Queue Identifier (QID) for the packet to be queued and both inputs 536, 538 are fed to a traffic manager 540. The traffic manager 540 outputs the classified packet on output 542 and the QID on output 544.

The present invention is subject to numerous variations and modifications. For example, the packet processing blocks having other types of functionality can be provided, such as:

- checksum or CRC generation and/or checking
- packet content modification/editing
- packet header removal
- packet header or trailer addition (e.g., for downstream processing)
- per flow rate control

As an alternative to a textual programming interface for implementing a given packet parser/classifier, the programmer/designer can use a graphical design program such as OrCAD or Microsoft Visio to draw and interconnect sub-blocks
with input windows for entering interconnecting expressions and entering program inputs.

The present invention has several advantages over prior art packet processing products. The present invention can be used to produce an inexpensive piece of digital hardware, while the prior art products are limited to programs running on a microprocessor. The present invention is scalable to handle simple to complex classification tasks, and software modules can be connected and configured in a variety of ways.

It will be understood that the embodiment described herein is merely exemplary and that a person skilled in the art may make many variations and modifications without departing from the spirit and scope of the invention. All such variations and modifications are intended to be included within the scope of the present invention as defined in the appended claims.
APPENDIX A

// This file contains generic Packet Processing Unit (PPU)

Object PPU
PARAMETERS {
   // configuration constant
   DataWidth = OEQ (DataWidth);
   Log2DataWidth = OEQ (Log2DataWidth);
   MaxHdrWords = OEQ (MaxHdrWords);
   Log2MaxHdrWords = OEQ (Log2MaxHdrWords);
   QualifierWidth = OEQ (QualifierWidth);
   ResultWidth = OEQ (ResultWidth);
   FieldWidth = OEQ (FieldWidth);
   Log2FieldWidth = OEQ (Log2FieldWidth);
   LookupAvailable = OEQ (LookupAvailable);
   LookupDepth = OEQ (LookupDepth);
   Log2LookupDepth = OEQ (Log2LookupDepth);
   TAGBits = OEQ (TAGBits);
   SobBits = OEQ (SobBits);
   EobBits = OEQ (EobBits);
   UseFIFOSTorage = OEQ (UseFIFOSTorage);
   PPUID = OEQ (PPUID);
  [IndexWidth = Log2DataWidth + Log2MaxHdrWords]
}

INTERFACES {
   // Result output expression is @Result_Expression
   // clock & reset
   input Rst;
   
   input CIk;
   // input packet header
   input InVal;
   input [SobBits-1:0] SOHIn;
   input [EobBits-1:0] EOHIn;
   input [DataWidth-1:0] DataIn;
   // output packet header
   output OutVal;
   output [SobBits-1:0] SOHOut;
   output [EobBits-1:0] EOHOut;
   output [DataWidth-1:0] DataOut;
   // control signal
   input [QualifierWidth-1:0] QualEnb;
   input [2:0] QualCond;
   input [IndexWidth-1:0] Index;
   input [Log2FieldWidth-1:0] Width;
   input [2:0] Opcode;
   input [FieldWidth-1:0] Parm1;
   input [FieldWidth-1:0] Parm2;
   input [FieldWidth-1:0] Mask;
   // if_generate (TAGBits != 0>
}
input [TAGBits-1:0] TAG;
//end_generate
// Micro-processor bus
input [3:0] MapAddr;
input [31:0] MapWrite rData;
// Result values
output [31:0] MapRdData;
// end_generate
output [7:0] Seqln;
// if_generate (GenSeq==0)
output [7:0] SeqOut;
// end_generate

//.... **** * **** .... ******** ******** ******** ******** ******** ********
// internal signal declaration
//********* ********* ********* ********* ********* ********* ********* *********
signal ExtOutVal;
signal [SobBits -1:0] ExtOutSob;
signal [EobBits -1:0] ExtOutEob;
signal [DataWidth- 1:0] ExtOutDat;
signal [FieldWidth- 1:0] ExtHdrValue;
signal [IndexWidth- 1:0] AdjustedIndex = Index + Width;
signal ExtHdrValid;
signal ExtHdrValid [NoOfLookups -1:0];
signal ExtHdrValue [NoOfLookups -1:0];
signal [Log2MaxHdrWords:0] WordNo [NoOfLookups -1:0];
signal [Log2DataWidth- 1:0] StartIndex [NoOfLookups -1:0];
signal [Log2DataWidth- 1:0] DataShift;
signal [Log2LookupDepth- 1:0] LkWrAddr [Log2LookupDepth -1:0];
signal [FieldWidth- 1:0] LkWrData [FieldWidth -1:0];
signal [FieldWidth- 1:0] LkWrRData [FieldWidth -1:0];
constant FieldWidthMultiple = (FieldWidth+DataWidth-1)/DataWidth;

FOR i from 1 to (NoOfLookups- 1) increment 1
WordNo[i] = (Index [IndexWidth- 1:Log2DataWidth] +i);
StartIndex [i] = 0,-
ENDLOOP

signal iOutVal;
signal iOutMatch;
signal QEnable,-
signal [QualifierWidth-1 :0] QualEnbCondition;
signal LkWrEnb;
signal [Log2LookupDepth- 1:0] LkWrAddr;
signal [FieldWidth-1:0] LkWrData;
signal [FieldWidth-1:0] LkWrRData;

// Special purpose registers for the MLU Blocks for custom instruction
// TEMPLATE CLASS instantiation for HLU
HLU
{
    OEQ(NoOfLookups),
    SobBits, EobBits, DataWidth,
    Log2DataWidth,
    MaxHdrWords, Log2MaxHdrWords,
    DataWidth
}
HeaderExtract_NameInstance
{
    // reset and clock
    Rst, CIk,
    // decoding rules
    WordNo, StartIndex,
    // input data
    InVal, SOHIn, EOHIn, Dataln,
    // Decoded Header Values
    ExtHdrValid, ExtHdrValue,
    // output data
    ExtOutVal, ExtOutSob, ExtOutEob, ExtOutDat
};

// data byte aligment
PROCESS (on change to ExtHdrValue)
    //if_generate (Custom_Shift == 0)
    Adjust ExtHdrValue with DataWidth, FieldWidth &
    AdjustedIndex[Log2DataWidth-1:0] to have the correct byte aligment at byte zero
    //else_generate
    Adjust ExtHdrValue with custom shift to have the correct byte aligment defined by the user
    //end_generate
ENDPROCESS

// extract qualifier enable
PROCESS (on rising edge of CIk)
    IF Rst is high
        assign initial value to QEnable
    ELSE

set QEnable to disabled state Zero to begin
// get the case statement for enable
CASE (QualCond)
    3'd0 : set QEnable to enable state One to Unconditional
    3'd1 : set QEnable to enable state One to if QualEnbCondition
        Equal to QualEnb
    3'd2 : set QEnable to enable state One to if QualEnbCondition
        Greater than QualEnb
    3'd3 : set QEnable to enable state One to if QualEnbCondition
        Greater than or Equal to QualEnb
    3'd4 : set QEnable to enable state One to if QualEnbCondition
        Less than QualEnb
    3'd5 : set QEnable to enable state One to if QualEnbCondition
        not Equal to QualEnb
default : QEnable = 0;
ENDCASE
ENDIF
ENDPROCESS

// MLU Template class instance
MLU
( FieldWidth, Log2FieldWidth, LookupAvailable, LookupDepth,
  Log2LookupDepth )
MLU_NameInstance
(
   // reset and clock
   RSt, CIk,
   // control signals
   QEnable, Opcode, Width, Param1, Param2, LkWrEnb, LkWrAddr, LkWrData,
   LkWrRBDData,
   SPReg_0, SPReg_1, SPReg_2, SPReg_3,
   // input check values
   ExtHdrValid, ExtHdrValue, Mask,
   // results
   iOutVal, iOutMatch
);

// ******************************************************

// I/O out signal
//if_generate (LookupAvailable==1 || UseFIFOStorage == 1)
if LookupAvailable is available or UseFIFOStorage option is enabled
// internal signals
  signal iEnbOut;
signal iValOut;
signal [SobBits-1:0] iSOHOut;
signal [EobBits-1:0] iEOHOut;
signal [DataWidth-1:0] iDataOut;
signal OutState,-
// assign the fifo output to be the output
OutVal = iValOut;
SOHOut = iSOHOut;
EOHOu t  = iEOHOu t;
DataOut  = iDataOut;
Result  = @Result_Expression;
ResVal  = (iSOHOut [SobBits-1] & iValOut);

// Packet FIFO template class
PacketFifo
{
    Log2MaxHdrWords, (DataWidth+SobBits+EobBits-1)
}
PacketFifo_NameInstance
{
    // write I/F
    (Rst ), (Cik ),
       (ExtOutVal), (ExtOutEob [EobBits-2 :01],ExtOutDat)) ,
    ([ExtOutSob,ExtOutEob [EobBits-2 :01],ExtOutDat]) ,
    (Rst), (Cik),
    (iEnbOut), (iValOut),
    (iEOHOut [EobBits-1]),
    (iSOHOut,iEOHOut [EobBits-2 :01],iDataOut )),
    //flags
    (open) , (open), (open)
}.

// transfer from the fifo out to the user
// after the result is declared
PROCESS (on rising edge of cik) : OUTPUT_ALINGEMENT_PROCESS
  IF Rst is high
        assign reset values to iEnbOut
        assign reset values to OutState
        assign reset values to Match
  ELSE // transfer the data when lookup is done
        CASE (OutState)
            Idle :
                turn off iEnbOut to the fifo
                // check if valid received form he lookup
                if iOutVal from MLU is received high
                give iEnbOut to the fifo
                iEnbOut = 1.
                declare Match based on iOutMatch value
                set OutState to goto next stage Xfr
            ENDIF
            Xfr :
                keep reading the fifo
                if Valid eop received from the FIFO go back to IDLE
                stage
            ENDCASE
            ENDF
            ENDP
ELSE // No Lookup Available
        //end_generate
        // reclock signals
signal [Log2MaxHdrWords+2 : 0] SellIndex;
signal ExtOutVal_r [MaxHdrWords+2 : 0];
signal [SobBits-1 : 0] ExtOutSob_r [MaxHdrWords+2 : 0];
signal [EobBits-1 : 0] ExtOutEob_r [MaxHdrWords+2 : 0];
signal [DataWidth-1:0] ExtOutDat_r [MaxHdrWords+2 : 0];
signal iResVal;

OutVal = ExtOutVal_r [SellIndex];
SOHOut = ExtOutSob_r [SellIndex];
EOHOut = ExtOutEob_r [SellIndex];
DataOut = ExtOutDat_r [SellIndex];
Result = @Result_Expression,-
ResVal = iResVal;
// on reclock will do for no lookup

PROCESS (on rising edge of CIk) : OUTPUT_ALIGNMENT_PROCESS
IF Rst is high
assign reset values to iResVal
assign reset values to Match
assign reset values to SellIndex
ELSE
Reclock the signals ExtOutVal_r, ExtOutSob_r, ExtOutEob_r & ExtOutDat_r\n
to generate a MaxHdrWords+3 word pipeline for future usage
// default
set iResVal to Zero
ENDIF
// gen expression
IF iOutVal is set to high
declare iResVal to be valid
set Match as iOutMatch
ENDIF
ENDIF
ENDPROCESS

//if_generate (LookupAvailable==1 | I UseFIFOStorage == 1)
ENDIF
//end_generate

---

**sequence out process**
PROCESS (on rising edge of CIk) : SEQUENCE_GENERATION
IF Rst is high
assign reset values to SeqOut
ELSE
// if_generate (GenSeq==0)
IF valid SOHIn is received
    set SeqOut equal to SeqIn
ENDIF
// else_generate
// increment the sequence no if valid InSOH is detected
IF valid SOHIn is received
    increment SeqOut by one
ENDIF
// end_generate
ENDIF
ENDPROCESS

// Register map

// register map process
PROCESS (on rising edge of CIk) : REGISTER_MAP
IF .Rst is high
    assign reset values to MapRdData, QualEnbCondition, LkWrEnb, LkWrAddr, LkWrData, SPReg_0, SPReg_1, SPReg_2, SPReg_3,
ELSE
    case address decoding
    IF user requested a write operation
        CASE (MapAddr)
            // PPU ID NO Write
            // Enable Condition
            1: latch QualEnbCondition from MapWrData
            // map address
            2: latch LkWrAddr from MapWrData
            // map data
            3: issue write to user look table using LkWrEnb
               latch LkWrData from MapWrData
            4: latch SPReg_0 from MapWrData
            5: latch SPReg_1 from MapWrData
            6: latch SPReg_2 from MapWrData
            7: latch SPReg_3 from MapWrData
        ENDCASE
        // if reading
        ELSE
            CASE (MapAddr)
                // PPU ID read only
            ENDCASE
        ENDIF
    ENDIF
ENDCASE
// if reading
ELSE
    case address decoding
    IF user requested a write operation
        CASE (MapAddr)
            // PPU ID NO Write
            // Enable Condition
            1: latch QualEnbCondition from MapWrData
            // map address
            2: latch LkWrAddr from MapWrData
            // map data
            3: issue write to user look table using LkWrEnb
               latch LkWrData from MapWrData
            4: latch SPReg_0 from MapWrData
            5: latch SPReg_1 from MapWrData
            6: latch SPReg_2 from MapWrData
            7: latch SPReg_3 from MapWrData
        ENDCASE
        // if reading
        ELSE
            CASE (MapAddr)
                // PPU ID read only
            ENDCASE
        ENDIF
    ENDIF
ENDCASE
// if reading
ELSE
    case address decoding
    IF user requested a write operation
        CASE (MapAddr)
            // PPU ID NO Write
            // Enable Condition
            1: latch QualEnbCondition from MapWrData
            // map address
            2: latch LkWrAddr from MapWrData
            // map data
            3: issue write to user look table using LkWrEnb
               latch LkWrData from MapWrData
            4: latch SPReg_0 from MapWrData
            5: latch SPReg_1 from MapWrData
            6: latch SPReg_2 from MapWrData
            7: latch SPReg_3 from MapWrData
        ENDCASE
        // if reading
        ELSE
            CASE (MapAddr)
                // PPU ID read only
            ENDCASE
        ENDIF
    ENDIF
ENDCASE
0:
  set MapRdData to PPUID
  // Enable Condition
1:
  set MapRdData to QualEnbCondition
  // map address
  // address needs to be set
  // for reading data
  // map data
3:
  set MapRdData to LkWrRBData
4:
  set MapRdData to SPReg_0
5:
  set MapRdData to SPReg_1
6:
  set MapRdData to SPReg_2
7:
  set MapRdData to SPReg_3
ENDCASE
ENDIF
ENDIF
ENDPROCESS
APPENDIX B

// This file contains decision forwarding unit (DFU)

Object DFU
PARAMETERS ()
// configuration constant
DataWidth = OEQ(DataWidth);  
InResultWidth = OEQ(InResultWidth);  
OutResultWidth = @EQ (OutResultWidth);  
NumPPUConnects = @L2 (NumPPUConnects);  
Log2NumPPUConnects = @L (@EQ (NumPPUConnects));  
DFUID = OEQ(DFUID);  
SopBits = OEQ (SopBits);  
EopBits = OEQ (EopBits);  
)
INTERFACES ()
// clock & reset
input Rst;
input ck;  
// Micro-processor bus
input [2:0] MapWrDrd_n;
input MapAddr;
input [31:0] MapWrData;  
output [31:0] MapRdData;  
// input buses
input [NumPPUConnects-1:0] RInValI;  
input [NumPPUConnects-1:0] MIn;
// for_generate @i (NumPPUConnects)
input [InResultWidth-1 :0] RIn_EQ(i);  
input [7:0] RInSeq_EQ (i);  
// end_generate  
// in packet header
input [NumPPUConnects-1 :0] DValIn;  
// for_generate @i (NumPPUConnects)
input [SopBits-1 :0] SOHIn_EQ (i);  
input [EopBits-1 :0] EOHIn_EQ(i);  
input [DataWidth-1:0] DIn_EQ(i);  
// end_generate  
// out result
output ROutAVal ;  
output . ROutBVal ;  
output ROutDVal ;  
output [OutResultWidth-1:0] ROut;  
output [7:0] SOUT;  
// out packet header common
output DValOut;  
output [SopBits- 1:0] SOHOut;  
output [EopBits- 1:0] EOHOut;  
output [DataWidth-1:0] DOut;  
output OutOfSeqErr;  
);  
// internal signals
signal StartOutputData ;
signal SequenceCheck ;
signal [NumPPUConections-1 : 0] iRInVal;
signal [NumPPUConections-1 : 0] iMIN;
signal [InResultWidth-1 : 0] iRIn [NumPPUConections-1 : 0];
signal [7:0] iRInSeq [NumPPUConections-1 : 0];

//for_generate ®i (NumPPUConections)
RIn [®EQ(i)] = RIn_@EQ(i);
RInSeq[@EQ(i)] = RInSeq_@EQ(i);
SOHIn [@EQ(i)] = SOHIn_@EQ(i);
EOHIn [OEQ(i)] = EOHIn_@EQ(i);
DIn [OEQ(D)] = DIn_@EQ(i);
//end_generate

signal [InPPUConections- 1 : 0] Match = MIN OR iMIN,-
signal [InResultWidth-1 : 0] Result [NumPPUConections- 1 : 0];
signal [7:0] Sequence [NumPPUConections- 1 : 0];
signal [Log2NumPPUConections : 0] LastArrivalData ;
signal [Log2NumPPUConections : 0] LastArrivalResult ;
FOR i from 1 to (NumPPUConections-1) increment 1
Result [i] = (RInVaI [i]) ? RIn [i] : iRIn[i];
Sequence [i] = (RInVaI [i]) ? RInSeq[i] : iRInSeq [i];
ENDLOOP

PROCESS (on rising edge of CIK) : VAIJE_LATCH_PROCESS
IF Rst is high
    assign initial value to iRInVal
    assign initial value to iMIN
ELSE
    // latch value for I/F
    FOR i from 1 to (NumPPUConections-1) increment 1
    IF(RInVaI[i] == 1)
        iRInVal [i] <= RInVaI [i];
        iMIN[i] <= MIN [i];
        iRIn[i] <= RIn [i];
        iRInSeq [i] <= RInSeq [i];
    ENDIF
ENDIF
ENDPROCESS

// latch all the signals to start processing
// process if all I/F have given valid data
PROCESS (on rising edge of CIK) : DATA_SELECTION_MUX_PROCESS
IF Rst is high
    assign initial value to ROutAVal
    assign initial value to ROutBVal
    assign initial value to ROutDVal
    assign initial value to DValOut
    assign initial value to SOHOut
assign initial value to EOHOut
assign initial value to DOut
assign initial value to StartOutputData
ELSE
  //default
  assign default value to ROutAVal
  assign default value to ROutBVal
  assign default value to ROutDVal
  assign default value to DValOut
  assign default value to SOHOut
  assign default value to EOHOut
  // if all I/F have given valids
  IF (RInVal OR iRInVal) expression is all ones
    IF user defined @MATCH_Expression_A is true
      ROutAVal <= 1;
      ELSE IF user defined @MATCH_Expression_B is true
        ROutBVal <= 1;
        ELSE
          ROutDVal <= 1;
        ENDIF
      ENDIF
    ELSE
      Clear all latched values on iRInVal, iMin
      // start the transfer of data
      set StartOutputData & SequenceCheck to One
      Latch the port which gave the data last
    ENDIF
  // start outputting data
  IF (RInVal are received by each connected PPU) OR (StartOutputData is set )
    IF (RInVal OR iRInVal) expression is all ones
      set LastArrivalData to the channel which gave last RInVal
      ENDIF
    // output data
    DValOut = DValIn [LastArrivalData];
    SOHOut = SOHIn [LastArrivalData];
    EOHOut = EOHIn [LastArrivalData];
    DOut = DIn [LastArrivalData];
  endif
  // clear the StartOutputData on EOP
  IF valid EOHOut is received
    reset StartOutputData to zero
  ENDIF
ENDIF
ENDPROCESS

// latch all the signals to start processing
// process if all I/F have given valid data
PROCESS (on rising edge of CIk) : RESULT_GENERATION_PROCESS
IF Rst is high
  assign initial value to ROut
  assign initial value to SOut
  assign initial value to OutOfSeqErr
assign initial value to SequenceCheck
ELSE
    assign default value to OutOfSeqErr
    assign default value to SequenceCheck
    // start outputting data
    IF (RinValI are received by each connected PPU) OR (StartOutputData is set )
        IF (RinValI OR iRInVal) expression is all ones
            set LastArrivalResult to the channel which gave last RInValI
        ENDIF
        IF (@MATCH_Expression_A> ROut = @Result_Expression_A;
        ELSE IF(@MATCH_Expression_B) ROut = @Result_Expression_B;
        ELSE ROut = @Result_Expression_D;
        ENDIF
        SOut = RinSeq [LastArrivalResult].
    ENDMIF
ELSE
    //OutOfSeqErr
    IF ( (SequenceCheck ACTIVE) AND
        (ANY OF THE RECEIVED SEQUENCE NUMBER DO NOT MATCH) )
        declare OutOfSeqErr to be active
    ENDMIF
ENDIF
ENDPROCESS
// output alignment process
PROCESS (on rising edge of CIk) : OUTPUT_ALINGMENT_PROCESS_PROCESS
    IF Rst is low
        reclock to align with the result outputs ROutAVal
        reclock to align with the result outputs ROutBVal
        reclock to align with the result outputs ROutDVal
        reclock to align with the result outputs DValOut
        reclock to align with the result outputs SOHOut
        reclock to align with the result outputs EOHOut
        reclock to align with the result outputs DOut
    ENDMIF
    IF Data output delay is more
        reclock to align with the result outputs ROut
        reclock to align with the result outputs SOut
    ENDMIF
ENDIF
ENDPROCESS
// register map
PROCESS (on rising edge of CIk) : REGISTER_MAP_PROCESS
    IF Rst is high
        assign initial value to MapRdData
    ELSE
        // case address decoding
        IF user requested a read (MapWrRd_n == 0)
            CASE (MapAddr)
                // PPU ID read only
                0 : MapRdData = DFUID;
            ENDCASE
        ENDMIF
    ENDIF
ENDIF
ENDPROCESS
APPENDXX C

This file contains generic packet modification unit block (PMU)

Object PMU

PARAMETERS {
  // configuration parameter
  DataWidth = $\equiv (\text{DataWidth})$;
  Log2DataWidth = $\equiv \text{Log}_2(\text{DataWidth})$;
  MaxHdrWords = $\equiv (\text{MaxHdrWords})$;
  Log2MaxHdrWords = $\equiv \text{Log}_2(\text{MaxHdrWords})$;
  ResultWidth = $\equiv \text{Log}_2(\text{MaxHdrWords} \cdot \text{DataWidth}/8)$;
  FieldWidth = $\equiv (\text{MaxHdrWord} \cdot \text{DataWidth})$;
  FieldValidWidth = $\equiv (\text{MaxHdrWords} \cdot \text{DataWidth}/8)$;
  TAGBits = $\equiv (\text{TAGBits})$;
  SobBits = $\equiv \text{SobBits}$;
  EobBits = $\equiv \text{EobBits}$;
  MaxPktSize = 16;
  PIUID = $\equiv (\text{PIUID})$;
  IndexWidth = $\text{Log}_2(\text{DataWidth} + \text{Log}_2(\text{MaxHdrWords})$;
}

INTERFACES {
  // clock & reset
  input Rst;
  input CIk;
  // input packet header
  input [SobBits-1:0] SOHIn;
  input [EobBits-1:0] EOHIn;
  input [DataWidth-1:0] Dataln;
  // output packet header
  output OutVal;
  output [SobBits-1:0] SOHOut;
  output [EobBits-1:0] EOHOut;
  output [MaxPktSize-1:0] DataOut;
  input [FieldValidWidth-1:0] HdrByteOfList;
  input [FieldWidth-1:0] HdrByteVal;
  //if_generate (TAGBits !=0)
  output [TAGBits-1:0] TAGIn;
  //if_generate (GenSeq==0)
  input [7:0] Seqln;
  // Result values
  output ResVal;
  output [ResultWidth-1:0] Result;
  // Micro-processor bus
  input [3:0] MapAddr;
  input [31:0] MapWrData,-
  output [31:0] MapRdData;
  // Result values
  output ResVal;
  output [ResultWidth-1:0] Result;
  //if_generate (GenSeq==0)
  input [7:0] Seqln;
output [7:0] SeqOut;
}

{ // internal signals
signal [DataWidth-1 : 0] HdrWord [MaxHdrWords -1:0];
signal [(DataWidth/8) -1:0] HdrWordMod [MaxHdrWords -1:0]; // full mod i.e. Zero -- No bytes valid
FOR i from 0 to (MaxHdrWords-1) increment by One
j = (MaxHdrWords-i-1) ;
HdrWord [j] = HdrData [((i+1) *DataWidth) -1 : (i*DataWidth) ];
HdrWordMod[j] = HdrByteVal [((i+1) * (DataWidth/8)) -1 : (i* (DataWidth/8)) ];
}

ENDDOPL

// endGenerate

output [7:0] SeqOut;
/////// internal signals
signal [DataWidth-1 : 0] HdrWord [MaxHdrWords -1:0];
signal [1:0] SeqOut;

FOR i from 0 to (MaxHdrWords-1) increment by One
j = (MaxHdrWords-i-1) ;
HdrWord [j] = HdrData [((i+1) *DataWidth) -1 : (i*DataWidth) ];
HdrWordMod[j] = HdrByteVal [((i+1) * (DataWidth/8)) -1 :

process (on rising edge of CIK)
IF Rst is high
assign reset values to HdrExtState
assign reset values to lnSop_i
assign reset values to lnSop_l
assign reset values to lnEop_i
assign reset values to lnDat_i
assign reset values to lResult
assign reset values to CurPktIndex
//if_generate (TAGBits!=0)
assign reset values to TAGOut


// Special purpose signalisters for the MLU Blocks for custom instruction
signal [31:0] SPsignal_0;
signal [31:0] SPsignal_1,-
signal [31:0] SPsignal_2,-
signal [31:0] SPsignal_3 ;
FOR i from 0 to (MaxHdrWords-1) increment by One
HdrState [i] = i;

process (on rising edge of CIK)
IF Rst is high
assign reset values to HdrExtState
assign reset values to lnSop_i
assign reset values to lnSop_l
assign reset values to lnEop_i
assign reset values to lnDat_i
assign reset values to lResult
assign reset values to CurPktIndex
//if_generate (TAGBits!=0)
assign reset values to TAGOut

};
ELSE
  // counter for word positions
  Reset, Increment & Clear CurPktlndex based on InSop. InEop &
  InVal
  IF InVal is active
    CASE (Opcode)
      Opr_Ins : // Insertion
        IF CurPktlndex < HdrByteOf fset
          InVal_i = InVal;
          InDat_i = InDat;
        ELSE IF CurPktlndex == HdrByteOf fset OR m range of the bytes to be inserted
          InVal_i = HdrByteVal,
          InDat_i = HdrData;
        ENDIF
      ELSE
        InVal_i = InVal_r [Adjusted OFFSET based on no of bytes inserted],
        InDat_i = InDat_r [Adjusted OFFSET based on no of bytes inserted];
    ENDIF
    Opr_Mod : // Modification
      IF CurPktlndex < HdrByteOf fset
        InVal_i = InVal;
        InDat_i = InDat;
      ELSE IF CurPktlndex == HdrByteOf fset OR in range of the bytes to be modified
        InVal_i = HdrByteVal,
        InDat_i = HdrData;
      ELSE
        InVal_i = InVal_r;
        InDat_i = InDat_r;
      ENDIF
    ENDIF
    Opr_Rtn : // Removal
      IF CurPktlndex < HdrByteOf fset
        InVal_i = InVal;
        InDat_i = InDat;
      ELSE IF CurPktlndex == HdrByteOf fset OR in range of the bytes to be deleted
        InVal_i = 0,
        ELSE
          InVal_i = InVal_r;
          InDat_i = InDat_r;
      ENDIF
    ENDIF
  END_CASE
ENDIF
ENDIF
ENDPROCESS // always
// Pipe-Line Stage
PROCESS (on rising edge of CIk)
  create pipeline relcock for signals InVal_r
  create pipeline relcock for signals InSop_r
  create pipeline relcock for signals InEop_r
  create pipeline relcock for signals InDat_r
ENDPROCESS
OutVal = InVal_i,-
SOHOut = InSop_i,-
EOHOut = InEop_i,-
DataOut = InDat_i;
ResVal = OutVal AND SOHOut;
// out alignment process
PROCESS (on rising edge of Clk) : OUTPUT_ALIGNMENT_PROCESS
IF Rst is low
  IF Result output delay is more
    reclock to align with the result outputs OutVal
    reclock to align with the result outputs SOHOut
    reclock to align with the result outputs EOHOut
    reclock to align with the result outputs DataOut
  ENDIF
  IF Data output delay is more
    reclock to align with the result outputs ResVal
    reclock to align with the result outputs Result
  ENDIF
ENDIF
ENDPROCESS

// sequence out process
PROCESS (on rising edge of Clk) : SEQUENCE_GENERATION
IF Rst is high
  assign reset values to SeqOut
ELSE
  //if_generate (GenSeq==0)
  IF valid SOHIn is received
    set SeqOut equal to Seqln
  ENDIF
  //else_generate
  // increment the sequence no if valid InSOH is detected
  IF valid SOHIn is received
    increment SeqOut by one
  ENDIF
  //end_generate
ENDIF
ENDPROCESS

*******

// Register map

**********

// register map process
PROCESS (on rising edge of Clk) : REGISTER_MAP
IF Rst is high
  assign reset values to MapRdData
  assign reset values to Opcode (i.e. Opcode present)
  assign reset values to SPsignal_O
  assign reset values to SPsignal_1
  assign reset values to SPsignal_2
assign reset values to $P_{signal_3}$

ELSE

    // case address decoding
    IF map write is requested $\text{MapWrRd}_n==1$

        CASE (MapAddr)
            // PIU ID NO Write
            1: OpCode = MapWrData;
            4: $SP_{signal_0} = MapWrData$;
            5: $SP_{signal_1} = MapWrData$;
            6: $SP_{signal_2} = MapWrData$;
            7: $SP_{signal_3} = MapWrData$;
            ENDCASE

        ENDIF

    ENDIF

ENDIF

ENDPROCESS

}
APPENDIX D

This file contains generic hardware look-up unit (header extraction block) (HLU)

Object HLU
PARAMETERS {
  NoOfLookups $\equiv$ (NoOfLookups);
  SobBits $\equiv$ (SobBits);
  EobBits $\equiv$ (EobBits);
  DataBits $\equiv$ (DataBits);
  Log2DataBits $\equiv$ @L2 (DataBits);
  MaxHdrWords $\equiv$ (MaxHdrWord);
  Log2MaxHdrWords $\equiv$ @L2 (MaxHdrWords);
  PartSize $\equiv$ (PartSize);
}

INTERFACES {
  // reset and clock
  input Rst;
  input CIk;
  // decoding rules
  input [Log2MaxHdrWords-1:0] WordNo [NoOfLookups-1:0];
  input [Log2DataBits-1:0] StartIndex [NoOfLookups-1:0];
  // input data
  input InVaI;
  input [SobBits-1:0] InSob; // Top bit is SOP
  input [EobBits-1:0] InEob; // Top bit is EOP
  input [DataBits-1:0] InDat;
  // Decoded Header Values
  output [PartSize-1:0] HdrValid [NoOfLookups-1:0];
  output [PartSize-1:0] HdrValue [NoOfLookups-1:0];
  // output data
  output OutVal;
  output [SobBits-1:0] OutSob,-
  output [EobBits-1:0] OutEob,-
  output [DataBits-1:0] OutDat;
};

IF Rst is high
  assign initial value to HdrValid
  assign initial value to HdrValid
  assign initial value to HdrValue
  assign initial value to HdrWordCntr
  // reset
ELSE

    // get header word counter ticking
    IF inVal is high
        // EOP seen
        IF InEob is high
            set HdrWordCnt to zero
        ELSE IF MaxHdrWords is not equal to MaxHdrWords
            increment HdrWordCnt by one
        ENDIF
    // header extraction loop
    for Ip = 0 to (NoθfLookups-1) increment Ip by one

        //IF header word is reached and data is valid
        IF current word is valid header word
            // issue valid word indication command
            declare HdrValid [Ip] valid
            // adjust the data to start index
            assign HdrValue [Ip] to (InDat LEFTSHIFT StartIndex [Ip])
        ENDIF
    ENDLOOP
ENDIF // clock
ENDPROCESS // always
// output assingment
OutVal = InVal;
OutSob = InSob;
OutEob = InEob;
OutDat = InDat;
}
APPENDIX E

This file contains generic Match/Lookup Unit (MLU)

Object MLU

PARAMETERS {
    FieldWidth = @EQ(FieldWidth);
    Log2FieldWidth = @L2(FieldWidth);
    LookupAvailable = @EQ(LookupAvailable);
    LookupDepth = @EQ(LookupDepth);
    Log2LookupDepth = @L2(LookupDepth);
    NoParallelLookup = @EQ(NoParallelLookup);
    Log2NoParallelLookup = @L2(NoParallelLookup);
}

INTERFACES {
    // reset and clock
    input Rst;
    input CIk;
    // control signals
    input [2:0] Opcode;
    input [Log2FieldWidth:0] Width;
    input [FieldWidth-1:0] Param1;
    input [FieldWidth-1:0] Param2;
    input LkRdEnb;
    input [Log2LookupDepth-1:0] LkRdAddr;
    input [FieldWidth-1:0] LkRdData;
    output [FieldWidth-1:0] LkRdRBData;
    input [31:0] SPReg_0;
    input [31:0] SPReg_1;
    input [31:0] SPReg_2;
    input [31:0] SPReg_3;
    // input check values
    input [FieldWidth-1:0] InVal;
    input [FieldWidth-1:0] InData;
    input [FieldWidth-1:0] InMask;
    // results
    output OutVal;
    output OutMatch;
};

// internal block generation
signal [Log2LokupDepth-1:0] LkRdAddr;
//if_generate (NoParallelLookup < 2)
signal [FieldWidth-1:0] LkRdData;
//else_generate
signal [FieldWidth-1:0] iLkRdRBData;
//end_generate
signal LklnStart;
signal LklnProgress;
signal LnkProcessState;
constant Idle = 1'b0;
constant Check = 1'b1;

//if_generate (LookupAvailable == 1)
//if_generate (NoParallelLookup < 2)
// TEMPLATE CIASS instantiation for SramRIRWl
SramRIRWl
  (
    Log2LookupDepth ,
    FieldWidth
  )
SramRIWl_NameInstance
  ();

// else_generate
// if no of parallel I/F more than 1
signal [NoParallelLookup- 1:0] LkWrEnb [®EQ(i) - LkWrEnb &
(LkWrAddr [Log2LookupDepth-1 :Log2LookupDepth-Log2NoParallelLookup] == i)]
ENDLOOP
//for_generate Θi (NoParallelLookup)
SramRIRWl
  (Log2LookupDepth-Log2NoParallelLookup)
  ,
  FieldWidth
)  
SramRIWl_NameInstance_@i

// Write I/F with Readback option
CIk, LkWrEnb, LkWrAddr, LkWrData, LkWrRBData,
// Read only I/F
CIk, 1'b1, LkRdAddr, LkRdData

//else_generate
// if no of parallel I/F more than 1
signal [FieldWidth- 1:0] LkWrRBData [®EQ(i) - LkWrRBData &
LkWrAddr [Log2LookupDepth-1:Log2LookupDepth-Log2NoParallelLookup] == i]]
ENDLOOP
//end_generate
//end_generate

// adjust the comparision to the width user requested
// The actual process
PROCESS' (on rising edge of CIk)
IF Rst is high
assign reset values to OutVal
assign reset values to OutMatch
assign reset values to LklnStart
assign reset values to LklnProgress
assign reset values to LnkProcessState
ELSE
  //default
  Clear OutVal & LklnStart

  // if the Match unit is enabled than try to do something
  IF QEnable is active high
    CASE (Opcode)
      3'd0 : // EQ: Equal to Param1
        IF InVal observed
          OutVal = 1;
          OutMatch = ((InData & InMask) == (Param1 & InMask));
        ENDIF
      3'd1 : // LT: Less Than Param1
        IF InVal observed
          OutVal = 1;
          OutMatch = ((InData & InMask) < (Param1 & InMask));
        ENDIF
      3'd2 : // GE: Greater Than or Equal to Param1
        IF InVal observed
          OutVal = 1;
          OutMatch = ((InData & InMask) >= (Param1 & InMask));
        ENDIF
      3'd3 : // GT: Greater Than Param1
        IF InVal observed
          OutVal = 1;
          OutMatch = ((InData & InMask) > (Param1 & InMask));
        ENDIF
      3'd4 : // RNG: Check if within range <Param1, Param2>
        IF InVal observed
          OutVal = 1;
          OutMatch = ((InData & InMask) >= (Param1 & InMask)) &
          ((InData & InMask) <= (Param2 & InMask));
        ENDIF
      3'd5 : // LUP: Look up
        IF LookupAvailable is active
          OutVal = 1;
          OutMatch = 1;
          // start lookup if the module is enabled for look
          up
        ELSE
          LklnStart = 1;
        ENDIF
    ENDIF
  ENDIF

  3'd7: // EXTR: Extract
IF InVal observed
  OutVal = 1;
  OutMatch = @EXTR_Expression;
ENDIF
default :
  IF InVal observed
    OutVal = 1;
    OutMatch = 1;
  ENDIF
ENDIF
ELSE IF InVal observed
  OutVal = 1;
  OutMatch = 0;
ENDIF
LENCASE

// else just return okay
ELSE IF InVal observed
  OutVal = 1;
  OutMatch = 1;
ENDIF
// process a lookup query
case (LnkProcessState)
  Idle:
    // clear Look up in progress begin
    LkInProgress = 0;
    // start processing
    if(LkInStart == 1)
      LkRdAddr = 0;
      LklnProgress = 1;
      LnKProcessState = Check;
    ENDIF
    Check:
      // declare match and exit the search
      //if generate (NoParallelLookup < 2)
      IF ((InData & InMask) == (LkRdData & InMask))
      //else generate
      IF( //for generate @i (NoParallelLookup)
        @IS (i==0) ? (i : (i (InData & InMask) == (LkRdData[@EQ i]
          & InMask))) //end generate
      //endif generate
      //else generate
        OutVal = 1;
        OutMatch = 1;
        LklnProgress = 0;
        LnKProcessState = Idle;
      // endif lookup limit reached
      ELSE IF address limit reached without match
        OutVal = 1;
        OutMatch = 0;
        LklnProgress = 0;
        LnKProcessState = Idle;
        // lookup address increment
      ELSE
        increment LkRdAddr by appropriate value based on
        NoParallelLookup
      ENDIF
    ENDCASE
//end CASE
//end_generate
ENDIF
ENDPROCESS
}
APPENDIX F

Vβ riloop2

// global variables
declare line_count = 0, input_f ilename, output_f ilename;
declare map ParamNum;
declare map ParamStr;

// ********************** ******** *** ******** ********
void error O
{
    Print appropriate error message;
    exit;
}

int Log2 (int n)
{
    int 1;
    for(l=1, n--; n>l; n=n>>l, l++);
    return 1;
}

int Expression (string e)
{
    declare tmptstr;
    tmptstr = e;
    for each s variable in string
    {
        FIND the corresponding value s from ParamNum, ParamStr maps;
        REPLACE s by value of s in tmptstr;
    }
    // unix or cygwin environment is required for bash script
    CALL bash expression evaluator and pass it tmptstr;
    return result from bash expression evaluator,
}

string VeriLoop2 (FILE in_file)
{
    declare buff size = 1024, buffer [buff size], word [buff size], frame.
    Match;
    declare loop_count;
    while END of input file is reached
    {
        GET LINE from file into buffer,-
        GET first word from buffer into word,-
        if (word — "®L2 (" )
        {
            GET next word from buffer;
            Append Log2 (word) to frame;
            SKIP ")" from input file,
        }
        if (word — "@EQ(" )
        {
            GET next word from buffer;
            Append Expression (word) to frame;
            SKIP ")" from input file;
        }
if (word == "//if_generate")
{
    get Expression following "//if_generate" into buffer,
    if (Expression (buffer, i) == 1)
    {
        do {
            GET LINE from file into buffer;
            frame = frame + buffer;
        } while (buffer != "//end_generate")
        // Don't store generate this piece of code
        else
        {
            do {
                GET LINE from file into buffer;
                if (word == "//elseif_generate")
                {
                    get Expression following "//if_generate" into buffer,
                    if (Expression (buffer, i) == 1)
                    {
                        do {
                            GET LINE from file into buffer,
                            frame = frame + buffer;
                        } while (buffer != "//end_generate")
                        break;
                    }
                }
                // do not add to frame
                } while (buffer != "//end_generate")
            }
    }
}
else if (word == "//let_generate") //let_generate @VariableName1 (expression)
                    //let_generate @VariableName2 "string"
    {
        // create new variable
        GET name value pair from the following two variables;
        Populate ParamNum, ParamStr using the name value pair;
    }
else if (word == "//for_generate")
{
    // do (expression) number of iterations of the code,
    // replacing every occurrence of @index_char with iteration index
    // [0; (expression) -1]
    GET loop_count from the next word following "//for_generate";
    GET buffer from lines between "//for_generate" and corresponding "//end_generate";
    for(i=0; i<loop_count; i++)
    {
        REPLACE reference to loop variable in buffer with i.
        APPEND buffer to frame;
    }
}  
else  
{
    // pass through the line to buffer
    APPEND buffer to frame;
}
}  
// while
return frame ;
}  
// veriloop2
int main ()  
{  
    declare FILE out_file, in_file, ini_file;  
    declare buffer, key, i, k;  
    Get input filename from global buffer into input_filename;  
    // open input file
    in_file = file_open (input_filename);  
    Get input filename from global buffer into output_filename;  
    // open output file
    out_file = file_open (output_filename);  
    // read in parameters from input line
    for remaining parameter on the command line
    {  
        if string consists of '='  
        {  
            GET name value pair <PARAM>=<VALUE>;  
            Populate ParamNum, ParamStr using the name value pair;
        }  
        else  
        {  
            error ();
        }
    }  
    // process source file
    buf = VeriLoop2 (in_file);  
    // output processed string to file
    output buf to file out_file;  
    // close files
    file_close (in_file);  
    file_closeout _file);  
}
**APPENDIX G**

**Supported Constructs**

1. Value replace construct specified by:
   
   ```@EQ(RValue)```
   
   which is replaced by the value of the `RValue`

2. For generate construct specified by:
   
   ```//for_generate @var (loopvalue)```
   
   code needed to be generated `loopvalue` times
   
   ```//end_generate```

3. If generate construct specified by:
   
   ```//if_generate (conditioni)```
   
   code needed to be generated if `conditioni` is true
   
   ```//else_if_generate (conditionN) - optional```
   
   code needed to be generated if `conditionN` is true
   
   ```//else_generate - optional```
   
   code needed to be generated if none of the above condition match
   
   ```//end_generate```

4. Is generate construct specified by:
   
   ```@IS(Condition)?(ValueIFTrue):(ValueIFFalse)```
   
   which is replaced by `ValueIFTrue` if `Condition` is true else replaced by `ValueIIFalse`
APPENDIX H

/// This file contains decision forwarding unit (DFU_1)

Object DFU
PARAMETERS (
  // configuration constant
  DataWidth = 32;
  InResultWidth = 8;
  OutResultWidth = 8;
  NumPPUConnects = 1;
  Log2NumPPUConnects = 1;
  DFUID = 0;
  SopBits = 1;
  EopBits = 4;
)
INTERFACES (
  // clock × reset
  input Rst;
  input CIk;

  // Micro-processor bus
  input [2:0] MapWrRd_n ;
  input [31:0] MapAddr ;
  output [31:0] MapRdData ;

  // input buses
  input [NumPPUConnects - 1:0] RinVal ;
  input [NumPPUConnects - 1:0] Min ;
  input [InResultWidth- 1:0] Rin_0 ;
  input [7:0] RinSeq_0 ;
  // in packet header
  input [NumPPUConnects-1:0] DValIn ;
  input [SopBits-1:0] SOHIn_0 ;
  input [EopBits-1:0] EOHIn_0 ;
  input [DataWidth-1:0] Din_0 ;

  // out result
  output ROutAVal ;
  output ROutBVal ;
  output ROutDVaI ;
  output [OutResultWidth-1:0] ROut ;
  output [7:0] SOut ;

  // out packet header common
  output DValOut ;
  output [SopBits -1:0] SOHOut ;
  output [EopBits -1:0] EOHOut ;
  output [DataWidth-1:0] DOu t ;
output OutOfSeqErr;

// internal signals
signal [NumPPUConnects-1 :0] StartOutputData;
signal SequenceCheck;
signal [NumPPUConnects-1 :0] iRInVal;
signal [NumPPUConnects-1 :0] iMIn;
signal [InResultWidth-1 :0] iRIn [NumPPUConnects-1 :0];
signal [InResultWidth-1 :0] iRInSeq [NumPPUConnects-1 :0];
signal [7:0] Match = MIn OR iMIn;
signal [7:0] Result [NumPPUConnects-1 :0];
signal [7:0] Sequence [NumPPUConnects-1 :0];
signal [Log2NumPPUConnects :0] LastArrivalData;
signal [Log2NumPPUConnects :0] LastArrivalResult;

FOR i from 1 to (NumPPUConnects-1) increment 1
   Result [i] = (RInVal[i]) ? RIn [i] : iRIn [i];
   Sequence[i] = (RInVal[i]) ? RInSeq[i] : iRInSeq[i];
ENDLOOP

PROCESS (on rising edge of CIk) : VALUE_LATCH_PROCESS
IF Rst is high
   assign initial value to iRInVal
   assign initial value to iMIn
ELSE
   // latch value for I/F
   FOR i from 1 to (NumPPUConnects-1) increment 1
      IF (RInValCi) = 1
         iRInVal [i] <= RInVal [i];
         iMIn[i] <= MIn [i];
         iRIn [i] <= RInCi;
         iRInSeq[i] <= RInSeq[i];
      ENDIF
   ENDLOOP
// latch all the signals to start processing
// process if all I/F have given valid data
PROCESS (on rising edge of CIk) : DATA_SELECTION_MUX_PROCESS
IF Rst is high
    assign initial value to ROutAVal
    assign initial value to ROutBVal
    assign initial value to ROutDVal
    assign initial value to SOHOut
    assign initial value to EOHOut
    assign initial value to DOut
    assign initial value to StartOutputData
ELSE
    //default
    assign default value to ROutAVal
    assign default value to ROutBVal
    assign default value to ROutDVal
    assign default value to DValOut
    assign default value to SOHOut
    assign default value to EOHOut
ENDIF
// if all I/F have given valids
IF (RInValI OR iRInVal) expression is all ones
    IF user defined Match [0] is true
        ROutAVal <= 1;
    ELSE IF user defined Match [0] is true
        ROutBVal <= 1;
    ELSE
        ROutDVal <= 1;
ENDIF
// start the transfer of data
set StartOutputData & SequenceCheck to One
Latch the port which gave the data last
ENDIF
// start outputing data
IF (RInValI OR iRInVal) expression is all ones
    set LastArrivalData to the channel which gave last RInValI
ENDIF
// output data
DValOut = DValIn [LastArrivalData];
SOHOut = SOHIn [LastArrivalData];
EOHOut = EOHIn [LastArrivalData];
DOut = DIn [LastArrivalData];
endif
// clear the StartOutputData on EOP
// if valid EOHOut is received
reset StartOutputData to zero
ENDIF
ENDPROCESS

// latch all the signals to start processing
// process if all I/F have given valid data
PROCESS (on rising edge of Clk) : RESULT_GENERATION_PROCESS
IF Rst is high
  assign initial value to ROut
  assign initial value to SOut
  assign initial value to OutOfSeqErr
  assign initial value to SequenceCheck
ELSE
  // default
  assign default value to OutOfSeqErr
  assign default value to SequenceCheck
ENDIF
IF (RInVaI are received by each connected PPU) OR (StartOutputData is set )
  IF (RInVaI OR iRInVaI) expression is all ones
    set LastArrivalResult to the channel which gave last RInVaI
  ENDIF
  IF (Match [O]) ROut = Result [O];
  ELSE IF (Match [O]) ROut = Result [OJ ;
  ELSE
    ROut = 0 ;
  ENDIF
  SOut = RInSeq [LastArrivalResult] ;
ENDIF

//OutOfSeqErr
IF ( (SequenceCheck ACTIVE) AND
  (ANY OF THE RECEIVED SEQUENCE NUMBER DO NOT MATCH) )
declare OutOfSeqErr to be active
ENDIF
ENDPROCESS
// out alingment process
PROCESS (on rising edge of CIk) : OUTPUT_ALINGMENT_PROCESS
IF Rst is low
   IF Result output delay is more
      reclock to align with the result outputs ROutAVal
      reclock to align with the result outputs ROutBVal
      reclock to align with the result outputs ROutDVal
      reclock to align with the result outputs DVaIOut
      reclock to align with the result outputs SOHOut
      reclock to align with the result outputs EOHOut
      reclock to align with the result outputs DOut
   ENDIF
ENDIF
IF Data output delay is more
   reclock to align with the result outputs ROut
   reclock to align with the result outputs SOut
ENDIF
ENDPROCESS

// register map
PROCESS (on rising edge of CIk) : REGISTER_MAP_PROCESS
IF Rst is high
   assign initial value to MapRdData
ELSE
   // case address decoding
   IF user requested a read (MapWrRd_n = = 0 )
      CASE (MapAddr)
         // PPU ID read only
         0 : MapRdData = DFUID;
      ENDCASE
   ENDIF
ENDIF
ENDPROCESS
APPENDIX I

// This file contains decision forwarding unit (DFU_4)

Object DFU PARAMETERS
// configuration constant
DataWidth = 32;
InResultWidth = 8;
OutResultWidth = 8;
NumPPUConnects = 4;
Log2NumPPUConnects = 2;
DFUID = 0;
SopBits = 1;
EopBits = 4;

INTERFACE

input Rst;
input CIk;

// Micro-processor bus
input [2:0] MapWrRd_n ;
input [31:0] MapWrData ;
output [31:0] MapRdData ;

// input buffer
input [NumPPUConnects-1:0] RInVal ;
input [NumPPUConnects-1:0] MIn;
input [InResultWidth-1:0] RIn_0 ;
in [7:0] RInSeq_0;
input [InResultWidth-1:0] RIn_1 ;
in [7:0] RInSeq_1;
input [InResultWidth-1:0] RIn_2 ;
in [7:0] RInSeq_2;
input [InResultWidth-1:0] RIn_3 ;
in [7:0] RInSeq_3 ;

// in packet header
input [NumPPUConnects-1:0] DVAI_in ;
input [SopBits-1:0] SOHIn_0 ;
input [EopBits-1:0] EOHIn_0 ;
input [DataWidth-1:0] DIn_0 ;
input [SopBits-1:0] SOHIn_1 ;
input [EopBits-1:0] EOHIn_1 ;
input [DataWidth-1:0] DIn_1 ;
input [SopBits-1:0] SOHIn_2 ;
input [EopBits-1:0] EOHIn_2 ;
input [DataWidth-1:0] DIn_2 ;
input [SopBits-1:0] SOHIn_3 ;
input [EopBitS-l:0] EOHIn_3;
input [DataWidth-l:0] DIN_3;

// out result
output ROutAVaI;
output ROutBVal;
output ROutDVal;
output [OutResultWidth-l:0] ROut;
output [7:0] SOut;

// out packet header common
output [SopBits-l:0] DValOut;
output [EopBitS-l:0] SOHOut;
output [DataWidth-l:0] DOut;
output OutOfSeqErr;

{ // internal signals
  signal StartOutputData;
  signal SequenceCheck;
  signal [NumPPUC artificially connected-l:0] iRInVal;
  signal [NumPPUC artificially connected-l:0] iMIn;
  signal [InResultWidth-l:0] RIn;
  signal [7:0] RInSeq;
  signal [SopBits-l:0] SOHIn;
  signal [EopBitS-l:0] EOHIn;
  signal [DataWidth-l:0] DIN;

  RIn[0] = RIn_0;
  RInSeq[0] = RInSeq_0;
  SOHIn[0] = SOHIn_0;
  EOHIn[0] = EOHIn_0;
  DIN[0] = DIN_0;
  RIn[1] = RIn_1;
  RInSeq[1] = RInSeq_1;
  SOHIn[1] = SOHIn_1;
  EOHIn[1] = EOHIn_1;
  DIN[1] = DIN_1;
  RIn[2] = RIn_2;
  RInSeq[2] = RInSeq_2;
  SOHIn[2] = SOHIn_2;
  EOHIn[2] = EOHIn_2;
  DIN[2] = DIN_2;
  RIn[3] = RIn_3;
  RInSeq[3] = RInSeq_3;
  SOHIn[3] = SOHIn_3;
  EOHIn[3] = EOHIn_3;
  DIN[3] = DIN_3;

  RIn [NumPPUC artificially connected-l:0];
  RInSeq [NumPPUC artificially connected-l:0];
  SOHIn [NumPPUC artificially connected-l:0];
  EOHIn [NumPPUC artificially connected-l:0];
  DIN [NumPPUC artificially connected-l:0];
signal [NumPPUConnects- 1:0] Match = Min OR iMin,-
signal [InResultWidth-1 :0] Result = [NumPPUConnects-1:0] ;
signal [7:0] Sequence = [NumPPUConnects- 1:0];
signal [Log2NumPPUConnects :0] LastArrivalData ;
signal [Log2NumPPUConnects :0] LastArrivalResult ;
FOR i from 1 to (NumPPUConnects-1) increment 1
Result [i] = (RInVal[i]) ? RIn[i] : iRInti];
Sequence [i] = (RInValIi]) ? RInSeqti] : iRInSeqti] ;
ENDLOOP

PROCESS (on rising edge of CIk) : VALUE_LATCH_PROCESS
IF Rst is high
assign initial value to iRInVal
assign initial value to iMin
ELSE
// latch value for I/F
FOR i from 1 to (NumPPUConnects-1) increment 1
IF(RInVal[i] ! = 1)
iRInVal[i] <= RInVal[i] ;
iMin [i] <= Min [i] ;
iRn [i] <= Rn [i] ;
iRnSeq [i] <= RnSeq {i} ;
ENDIF
ENDLOOP
ENDIF
ENDPROCESS

// latch all the signals to start processing
// process if all I/F have given valid data
PROCESS (on rising edge of CIk) : DATA_SELECTION_MUX_PROCESS
IF Rst is high
assign initial value to ROutAVal
assign initial value to ROutBVal
assign initial value to ROutDVal
assign initial value to DValOut
assign initial value to SOHOut
assign initial value to EOHOut
assign initial value to DOut
assign initial value to StartOutputData
ELSE
// default
assign default value to ROutAVal
assign default value to ROutBVal
assign default value to ROutDVal
assign default value to DValOut
assign default value to SOHOut
assign default value to EOHOut
// if all I/F have given valids
IF (RInVaI OR iRInVal) expression is all ones
  IF user defined ScMatch is true
    ROutAVal <= 1;
  ELSE IF user defined |Match is true
    ROutBVal <= 1;
  ELSE
    ROutDVal <= 1;
  ENDIF
CLEAR all latched values on iRInVal, iMIn
// start the transfer of data
set StartOutputData & SequenceCheck to One
Latch the port which gave the data last
ENDIF

// start outputting data
IF (RInVaI are received by each connected PPU) OR (StartOutputData is set)
  IF (RInVaI OR iRInVal) expression is all ones
    set LastArrivalData to the channel which gave last RInVaI
  ENDIF
  IF (RInVaI OR iRInVal) expression is all ones
    set LastArrivalData to the channel which gave last RInVaI
  ENDIF
  // output data
  DValOut = DValIn [LastArrivalData];
  SOHOut = SOHIn [LastArrivalData];
  EOHOut = EOHIn [LastArrivalData];
  DOut = DIn [LastArrivalData];
ENDIF

// clear the StartOutputData on EOP
IF valid EOHOut is received
  reset StartOutputData to zero
ENDIF

ENDPROCESS

// latch all the signals to start processing
// process if all I/F have given valid data
PROCESS (on rising edge of CIk) : RESULT_GENERATION_PROCESS
IF Rst is high
  assign initial value to ROut
  assign initial value to SOut
  assign initial value to OutOfSeqErr
  assign initial value to SequenceCheck
ELSE
// default
assign default value to OutOfSeqErr
assign default value to SequenceCheck

// start outputting data
IF (RInVaI are received by each connected PPU) OR (StartOutputData is set )
  IF (RInVaI OR iRInVal) expression is all ones
      set LastArrivalResult to the channel which gave last RInVaI
  ENDIF
  IF (tMatch) ROut = Result[0];
  ELSE IF(Match) ROut = Result [1];
  ELSE        ROut = 0;
  ENDIF
  SOut = RInSeq [LastArrivalResult];
ENDIF

// OutOfSeqErr
IF ( (SequenceCheck ACTIVE) AND
      (ANY OF THE RECEIVED SEQUENCE NUMBER DO NOT MATCH))
   declare OutOfSeqErr to be active
ENDIF
ENDPROCESS

// output alignment process
PROCESS (on rising edge of C1k) : OUTPUT_ALINGMENT_PROCESS_PROCESS
IF Rst is low
  IF Result output delay is more
      reclock to align with the result outputs ROutAVal
      reclock to align with the result outputs ROutBVal
      reclock to align with the result outputs ROutDVal
      reclock to align with the result outputs DValOut
      reclock to align with the result outputs SOHOut
      reclock to align with the result outputs EOHOut
      reclock to align with the result outputs DOut
  ENDIF
IF Data output delay is more
  reclock to align with the result outputs ROut
  reclock to align with the result outputs SOut
ENDIF
ENDPROCESS

// register map
PROCESS (on rising edge of C1k) : REGISTER_MAP_PROCESS_PROCESS
IF Rst is high
   assign initial value to MapRdData
ELSE
  // case address decoding
  IF user requested a read (MapWrRd_n == 0)
    CASE (MapAddr)
      // PPU ID read only
      0: MapRdData = DFUID;
    ENDCASE
    ENDF
  ENDF
ENDPROCESS

}
APPENDIX J

---
// This file contains Packet Processing Unit(s) (PPU)
// -------------------------------

Object PPU
PARAMETERS {
// configuration constant
DataWidth = 32;
Log2DataWidth = 5;
MaxHdrWords = 8;
Log2MaxHdrWords = 3;
QualifierWidth = 2;
ResultWidth = 8;
FieldWidth = 16;
Log2FieldWidth = 4;
LookupAvailable = 0;
LookupDepth = 8;
Log2LookupDepth = 3;
TAGBits = 0;
SobBits = 1;
EobBits = 4;
UseFIFOStorage = 0;
PPUID = 0;
IndexWidth = Log2DataWidth + Log2MaxHdrWords;
}
INTERFACES {
// Result output expression is ExtHdrValue
// clock & reset
input Rst;  // clock & reset
input CIk;
// input packet header
input [SobBits-1:0] SOHIn;
input [EobBits-1:0] EOHIn;
input [DataWidth-1:0] Dataln;
// output packet header
output OutVal;
output [SobBits-1:0] SOHOut;
output [EobBits-1:0] EOHOut;
output [DataWidth-1:0] DataOut;
// control signal
input [QualifierWidth-1:0] QualEnb;
input [2:0] QualCond;
input [IndexWidth-1:0] Index;
input [Log2FieldWidth :0] Width;
input [2:0] Opcode;
input [FieldWidth-1:0] Paraml;
input [FieldWidth-1:0] Param2;
input [FieldWidth-1:0] Mask;
// Micro-processor bus
input MapWrRd_n;
input [3:0] MapAddr,
input [31:0] MapWrData,-
output [31:0] MapRdData;

// Result values
output Match;
output Result;
output [7:0] SeqOut,
}

// internal signal declaration
signal [3:0] MapAddr,
input [31:0] MapWrData,-
output [31:0] MapRdData;
// Result values
output Match;
output Result;
output [7:0] SeqOut,
}

// Special purpose registers for the MLU Blocks for custom instruction
signal [31:0] SPReg_0;
signal [31:0] SPReg_1;
signal [31:0] SPReg_3;

// Special purpose registers for the MLU Blocks for custom instruction
signal [31:0] SPReg_0;
signal [31:0] SPReg_1;
signal [31:0] SPReg_3;

constant Idle = 0;
constant

Xfr = 1;

// TEMPLATE CLASS instantiation for HLU
HLU(0,
SobBits, EobBits, DataWidth,
Log2DataWidth,
MaxHdrWord ε, Log2MaxHdrWords,
DataWidth)

HeaderExtract_Name Instance
{
// reset and clock
Rst, CIk,
// decoding rules
WordNo , StartIndex,
// input data
InVaI, SOHIn, EOHIn, Dataln,
// Decoded Header Values
ExtHdrValid, ExtHdrValue,
// output data
ExtOutVal, ExtOutSob, ExtOutEob, ExtOutDat
};

// data byte alignment
PROCESS (on change to ExtHdrValue)

Adjust ExtHdrValue with DataWidth, FieldWidth &
AdjustedIndex[Log2DataWidth-1 :0] to have\nthe correct byte alignment at byte zero
ENDPROCESS

// extract qualifier enable
PROCESS (on rising edge of CIk)

IF Rst is high
assign initial value to QEnable
ELSE
set QEnable to disabled state Zero to begin
// get the case statement for enable
CASE (QualCond)
 3'd0 : set QEnable to enable state One to Unconditional
 3'd1 : set QEnable to enable state One to if QualEnbCondition
Equal to QualEnb
 3'd2 : set QEnable to enable state One to if QualEnbCondition
Greater than QualEnb
 3'd3 : set QEnable to enable state One to if QualEnbCondition
Greater than or Equal to QualEnb
 3'd4 : set QEnable to enable state One to if QualEnbCondition
Less than QualEnb
3'd5 : set QEnable to enable state One to if QualEnbCondition
not Equal to QualEnb
default : QEnable = 0.

ENDCASE
ENDIF
ENDPROC 3

// MLU Template class instance
MLU ( Fieldwidth, Log2FieldWidth, LookupAvailable, LookupDepth,
      Log2LookupDepth )
MLU_NameInstance
{
    // reset and clock
    Rst, CIk,
    // control signals
    QEnable, Opcode, width, Parana, Param2, LkWrEnb, LkWrAddr, LkWrData,
    LkWrRBData ,
    SPReg_0, SPReg_1, SPReg_2, SPReg_3 ,
    // input check values
    ExtHdrValid, ExtHdrValue, Mask,
    // results
    iOutVal, iOutMatch
};

// I/O out signal
// reclock signals
signal [Log2MaxHdrWords+2 :0] SelIndex,
signal [SobBits-1 :0] ExtOutSob_r [MaxHdrWords+2 :0];
signal [EobBits-1 :0] ExtOutEob_r [MaxHdrWords+2 :0];
signal [DataWidth-1 :0] ExtOutDat_r [MaxHdrWords+2 :0];
signal iResVal;

OutVal = ExtOutVal_r [SellIndex];
SOHOut = ExtOutSob_r [SellIndex];
EOHOut = ExtOutEob_r [SellIndex];
DataOut = ExtOutDat_r [SellIndex];

Result = ExtHdrValue;
ResVal = iResVal
// on reclock will do for no lookup
PROCESS ( on rising edge of CIk ) :
OUTPUT_ALINGEMENT_PROCESS
IF Rst is high
assign reset values to iResVal
assign reset values to Match
assign reset values to SellIndex
ELSE
Reclock the signals ExtOutVal_r, ExtOutSob_r, ExtOutEob_r & ExtOutDat_r
\n
...
set iResVal to Zero

IF valid SOH in is received
   // word lookup + 1 match clock
   Calculate Sellndex based on WordNo of the last lookup
ENDIF

// gen expression
IF iOutVal is set to high
   declare iResVal to be valid
   set Match as iOutMatch
ENDIF

ENDPROCESS

//
******************************************************

// sequence out process
PROCESS (on rising edge of CIk) : SEQUENCE_GENERATION
   IF Rst is high
      assign reset values to SeqOut
   ELSE
      // increment the sequence no if valid InSOH is detected
      IF valid SOHIn is received
         increment SeqOut by one
      ENDIF
   ENDIF
ENDPROCESS

******
// Register map
******
//.
******
// register map process
PROCESS (on rising edge of CIk) : REGISTER_MAP
   IF Rst is high
      assign reset values to MapRdData, QualEnbCondition, LkWrEnb,
      LkWrAddr, \n      LkWrData, SPReg_0, SPReg_1, SPReg_2, SPReg_3,
   ELSE
      // case address decoding
      IF user requested a write operation
         CASE (MapAddr)
            // PPU ID NO Write
            // Enable Condition
            1:
               latch QualEnbCondition from MapWrData
               // map address
      
      
      2:
latch LkWrAddr from MapWrData
   // map data
3:
   issue write to user look table using LkWrEnb
   latch LkWrData from MapWrData
4:
   latch SPReg_0 from MapWrData
5:
   latch SPReg_1 from MapWrData
6:
   latch SPReg_2 from MapWrData
7:
   latch SPReg_3 from MapWrData
ENDCASE
// if reading
ELSE
CASE (MapAddr)
   // PPU ID read only
0:
   set MapRdData to PPUI D
   // Enable Condition
1:
   set MapRdData to QualEnbCondition
   // map address
   // address needs to be set
   // for reading data
   // map data
3:
   set MapRdData to LkWrRBData
4:
   set MapRdData to SPReg_0
5:
   set MapRdData to SPReg_1
6:
   set MapRdData to SPReg_2
7:
   set MapRdData to SPReg_3
ENDCASE
ENDIF
ENDPROCESS
Object HLU
PARAMETERS
\( \text{NoOf Lookups} = 0; \)
\( \text{SobBits} = 1; \)
\( \text{EobBits} = 4; \)
\( \text{DataBits} = 32; \)
\( \text{Log2DataBits} = 5; \)
\( \text{MaxHdrWords} = 8; \)
\( \text{Log2MaxHdrWords} = 3; \)
\( \text{PartSize} = 16; \)
}

INTERFACES {
// reset and clock
input Rst;
input CIk;

// decoding rules
input \[\text{Log2MaxHdrWords} : 0\] \text{WordNo} \[\text{NoOfLookups} - 1:0\];
input \[\text{Log2DataBits} - 1 : 0\] \text{StartIndex} \[\text{NoOfLookups} - 1:0\];
// input data
input \[\text{SobBits} - 1: 0\] \text{InSob}; // Top bit is SOP
input \[\text{EobBits} - 1: 0\] \text{InEob}; // Top bit is EOP
input \[\text{DataBits} - 1: 0\] \text{InDat};

// Decoded Header Values.
output \text{HdrValid} \[\text{NoOfLookups} - 1:0\];
output \[\text{PartSize} - 1 : 0\] \text{HdrValue} \[\text{NoOfLookups} - 1:0\];
// output data
output \text{OutVal};
output \[\text{SobBits} - 1: 0\] \text{OutSob};
output \[\text{EobBits} - 1: 0\] \text{OutEob};
output \[\text{DataBits} - 1: 0\] \text{OutDat};
};

// configuration parameter
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\n
\text{Hd}rW\text{ordCntr};

PROCESS (on rising edge of CIk)
IF Rst is high
  assign initial value to HdrValid
  assign initial value to HdrValid
  assign initial value to HdrValue
  assign initial value to HdrWordCnt
  // reset
ELSE
  // get header word counter ticking
  IF InVal is high
    // EOP seen
    IF InEob is high
      set HdrWordCnt to zero
    ELSE IF MaxHdrWords is not equal to MaxHdrWords
      increment HdrWordCnt by one
  ENDIF
ENDIF

// header extraction loop
for Ip = 0 to (NoθfLookups-1) increment Ip by one

// IF header word is reached and data is valid
// current word is valid header word
// issue valid word indication command
// declare HdrValid [Ip] valid
// adjust the data to start index
assign HdrValue [Ip] to (InDat LEFTSHIFT StartIndex [Ip])
ENDIF
ENDLOOP

ENDIF // Clock
ENDPROCESS // always

// output asignment
OutVal = InVal;
OutSob = InSob;
OutEob = InEob;
OutDat = InDat;
APPENDIX L

```hindi
// This file contains Match/Lookup Unit (MLU)

Object MLU
PARAMETERS {
  FieldWidth = 16;
  Log2FieldWidth = 4;
  LookupAvailable = 0;
  LookupDepth = 8;
  Log2LookupDepth = 3;
  NoParallelLookup = 0;
  Log2NoParallelLookup = 1;
}

INTERFACES {
  // reset and clock
  input Rst;
  input Clk;
  // control signals
  input QEnable;
  input [2:0] Opcode;
  input [Log2FieldWidth-1:0] Width;
  input [FieldWidth-1:0] Param1;
  input [FieldWidth-1:0] Param2;
  input LkWrEnb;
  input [Log2LookupDepth-1:0] LkWrAddr;
  input [FieldWidth-1:0] LkWrData;
  output [FieldWidth-1:0] LkWrRBData;
  input [31:0] SPReg_0;
  input [31:0] SPReg_1;
  input [31:0] SPReg_2;
  input [31:0] SPReg_3;

  // input check values
  input InVaI;
  input [FieldWidth-1:0] InData;
  input [FieldWidth-1:0] InMask;
  // results
  output OutVal;
  output OutMatch;

};

// internal block generateion

signal [Log2LookupDepth-1:0] LkRdAddr;
signal [FieldWidth-1:0] LkRdData;
signal [FieldWidth-1:0] LkWrRBData;
signal LkInStart;
signal LklnProgress;
```
signal LnkProcessState;
constant Idle = l'b0;
constant CChheeeccckk = l'"b1;

IF adjust the comparision to the width user requested

// The actual process
PROCESS (on rising edge of CIk)
  IF LkStart is high
    assign reset values to OutVal
    assign reset values to OutMatch
    assign reset values to LklnStart
    assign reset values to LklnProgress
    assign reset values to LnkProcessState
  ELSE
    //default
    Clear OutVal & LklnStart

    IF QEnable is active high
      CASE (Opcode)
        3'd0 : // EQ: Equal to Paraml
          IF InVal observed
            OutVal = 1;
            OutMatch = ((InData & InMask) == (Paraml & InMask));
        ENDIF
        3'd1 : // LT: Less Than Paraml
          IF InVal observed
            OutVal = 1;
            OutMatch = ((InData & InMask) < (Paraml & InMask));
        ENDIF
        3'd2 : // LE: Less Than or Equal to Paraml
          IF InVal observed
            OutVal = 1;
            OutMatch = ((InData & InMask) <= (Paraml & InMask));
        ENDIF
        3'd3 : // GT: Greater Than Paraml
          IF InVal observed
            OutVal = 1;
            OutMatch = ((InData & InMask) > (Paraml & InMask));
        ENDIF
        3'd4 : // GE: Greater Than or Equal to Paraml
          IF InVal observed
            OutVal = 1;
            OutMatch = ((InData & InMask) >= (Paraml & InMask));
        ENDIF
        3'd5 : // RNG: Check if within range <Paraml ,Param2>
          IF InVal observed
            OutVal = 1;
            OutMatch = ((InData & InMask) >= (Paraml & InMask))
                          & (InData & InMask) = (Param2 & InMask));
      ENDIF
      ENDIF
  ENDIF
}
ELSE IF InVaI observed
OutVal = 1;
OutMatch = 1;
ENDIF
ENDPROCESS

9'd6 : // LUP: Look up
IF InVaI observed
  IF LookupAvailàble is active
    OutVal = 1;
    OutMatch = 1;
  // start lookup if the module is enabled for look
else
  LklnStart = 1;
ENDIF
end

3'd7: // EXTR: Extract
IF InVaI observed
  OutVal = 1;
  OutMatch = 1;
ENDIF
ELSE IF InVaI observed
  OutVal = 1;
  OutMatch = 0;
ENDIF
ENDCASE

//else jut return okay
ELSE IF InVaI observed
  OutVal = 1;
  OutMatch = 0;
ENDIF

ENDIF
ENDPROCESS

}
What is claimed is:

1. A system for designing packet processing products, comprising:
   a user interface for allowing a user to define a desired packet processing
   algorithm using a plurality of discrete packet processing blocks, each of said blocks
   corresponding to a portion of said desired packet processing algorithm;
   means for allowing the user to define connections between said plurality of
   packet processing blocks; and
   means for processing said plurality of packet processing blocks and said
   connections to provide a list of instructions in a hardware description language for
   producing an integrated circuit capable of executing said desired packet processing
   algorithm.

2. The system of Claim 1, further comprising an integrated circuit constructed
   using said list of instructions.

3. The system of Claim 1, further comprising a NETLIST generated using said
   list of instructions.

4. The system of Claim 1, wherein said plurality of packet processing blocks
   further includes a Packet Processing Unit (PPU) for:
   extracting a header of a packet;
   pointing to a portion of the header of a predetermined width using a
   predetermined index of a bit location in the header;
   comparing the data represented by the portion of the header with at least
   one predetermined value; and
   declaring a match when the result of the comparison is true.

5. The system of Claim 4, wherein said Packet Processing Unit further includes
   means for accessing an external Content-Addressable Memory (CAM) or Random-
   Access Memory (RAM).
6. The system of Claim 5, wherein said Packet Processing Unit further includes a Hardware Lookup Unit for extracting a desired portion of header of a packet based on
determining when a start of header bit is active;
determining the number of bits for which the data stream of the packet is valid after said start of header bit is active; and
determining when the end of header bit is active; and
extracting said header based on said start of header bit, said number of bits, said end of header bit, said index, and said width.
7. The system of Claim 6, wherein said Packet Processing Unit further includes a Delay/FIFO module for delaying the extracted header by the sum of a predetermined number of clock cycles and a variable number of clock cycles based on said predetermined index.
8. The system of Claim 7, wherein said Delay/FIFO module is implemented using a delay line.
9. The system of Claim 7, wherein said Delay/FIFO module is implemented using a FIFO.
10. The system of Claim 7, wherein said Packet Processing Unit further includes a Match and Lookup Unit for determining if a user defined match of a condition is true to generate a match output based on a comparison of said desired portion of header of a packet with one or more user defined parameters and a predetermined logical condition.
11. The system of Claim 10, wherein said Packet Processing Unit further includes a Result Generation process for generating a result output based on one of a fixed expression, an arithmetic expression, and a logical expression.
12. The system of Claim 11, wherein said result output is used as an input to another Packet Parsing Unit.
13. The system of Claim 11, wherein said Packet Processing Unit further includes a Sequence Generation process for generating a sequence number for use by a Decision and Forwarding Unit.
14. The system of Claim 13, wherein said Packet Parsing Unit further includes an Output Alignment process for aligning said packet header with said result output and said match output.

15. The system of Claim 14, wherein said packet header, said result output, and said match output are aligned on a start of packet boundary.

16. The system of Claim 14, wherein said packet header, said result output, and said match output are aligned on an end of packet boundary.

17. The system of Claim 14, wherein said Packet Parsing Unit has a plurality of internal programmable registers.

18. The system of Claim 14, wherein said Packet Parsing Unit is programmable from an external microprocessor.

19. The system of Claim 1, wherein said plurality of packet processing blocks further includes a Packet Modification Unit (PMU) for:
   
   extracting a packet;
   
   pointing to a portion of the packet of a predetermined width using a predetermined index of a bit location in the packet; and
   
   modifying the portion of the packet.

20. The system of Claim 19, wherein modifying the portion of the packet further includes means for deleting the portion of the packet.

21. The system of Claim 19, wherein modifying the portion of the packet further includes means for overwriting the portion of the packet.

22. The system of Claim 19, wherein modifying the portion of the packet further includes means for inserting data at the position in the portion of the packet pointed to by index.

23. The system of Claim 19, wherein said Packet Modification Unit further includes a Delay/FIFO module for delaying the packet by the sum of a predetermined number of clock cycles and a variable number of clock cycles based on a number of bytes to be inserted.

24. The system of Claim 23, wherein said Delay/FIFO module is implemented using a delay line.
25. The system of Claim 23, wherein said Delay/FIFO module is implemented using a FIFO.
26. The system of Claim 23, wherein said Packet Modification Unit further includes a Modification Unit for modifying said portion of the packet based on a ByteOffset input indicating said index and ByteValid input indicating the number of clock cycles needed for modifying the packet.
27. The system of Claim 26, wherein said Modification Unit further includes a ByteData input for providing bytes to be inserted into the packet.
28. The system of Claim 26, wherein said Packet Modification Unit further includes a Result Generation process for generating a result output based on a number of bytes inserted into the packet.
29. The system of Claim 28, wherein said result output is used as an input to one of another Packet Parsing Unit and another Packet Modification Unit.
30. The system of Claim 28, wherein said Packet Modification Unit further includes a Sequence Generation process for generating a sequence number for use by a Decision and Forwarding Unit.
31. The system of Claim 30, wherein said Packet Modification Unit further includes an Output Alignment process for aligning said packet with said result output.
32. The system of Claim 31, wherein said packet and said result output are aligned on a start of packet boundary.
33. The system of Claim 31, wherein said packet header and said result output are aligned on an end of packet boundary.
34. The system of Claim 19, wherein said Packet Modification Unit is programmable from an external microprocessor.
35. The system of Claim 1, wherein said plurality of packet processing blocks further includes a Decision and Forwarding Unit (DFU) for performing one of drop, queue, and forwarding operations on at least one packet.
36. The system of Claim 35, wherein said Decision and Forwarding Unit performs one of drop, queue, and forwarding operations on at least one packet.
based on at least one match output and at least one result output of a Packet Processing Unit.

37. The system of Claim 36, wherein said Decision and Forwarding Unit further includes a first Latch for latching said at least one incoming packet and a second Latch for latching a result output associated with said at least one incoming packet.

38. The system of Claim 37, wherein said Decision and Forwarding Unit further includes a Data Selection Multiplexer for selecting one of said at least one incoming packet for output to one of a drop, queue, and forwarding port.

39. The system of Claim 38, wherein said Decision and Forwarding Unit further includes a Result Generation process for selecting a result output and a match output associated with said at least one incoming packet.

40. The system of Claim 39, wherein said match output and said result output determines to which port said packet is forwarded.

41. The system of Claim 39, wherein said result output is based on one of a fixed expression, an arithmetic expression, and a logical expression.

42. The system of Claim 39, wherein said Decision and Forwarding Unit further includes an Output Alignment process for aligning said packet with said result output.

43. The system of Claim 42, wherein said packet and said result output are aligned on a start of packet boundary.

44. The system of Claim 42, wherein said packet header and said result output are aligned on an end of packet boundary.

45. The system of Claim 35, wherein said Decision and Forwarding Unit is programmable from an external microprocessor.

46. The system of Claim 1, further including a packet processing block for checksum or CRC generation or checking.

47. The system of Claim 1, further including a packet processing block for packet header removal.

48. The system of Claim 1, further including a packet processing block for packet header or trailer addition.
49. The system of Claim 1, further including a packet processing block for per flow rate control.

50. A method for designing packet parsing and classification products, comprising the steps of:
   - providing a user interface for allowing a user to define a desired packet processing algorithm using a plurality of discrete packet processing blocks, each of said blocks corresponding to a portion of said desired packet processing algorithm;
   - allowing the user to define connections between said plurality of packet processing blocks; and
   - processing said plurality of packet processing blocks and said connections to provide a list of instructions in a hardware description language for producing an integrated circuit capable of executing said desired packet processing algorithm.

51. The method of Claim 50, further comprising the step of constructing an integrated circuit using said list of instructions.

52. The method of Claim 51, further comprising the step of generating a NETLIST using said list of instructions.

53. The method of Claim 51, further comprising the step of filling out a connection document based on the plurality of packet processing blocks.

54. The method of Claim 53, wherein said connection document is implemented in a graphical user interface.

55. The method of Claim 54, further including the step of configuring the plurality of packet processing blocks from a plurality of files each containing a different type of packet processing block of maximal functionality.

56. The method of Claim 55, wherein said step of configuring further includes the step of using a preprocessor to perform substitution, looping, and branching to cull a customized packet processing block from said packet processing block of maximal functionality.

57. The method of Claim 56, further including the step of instantiating the plurality of processing blocks and making connections between said packet processing blocks in a top level file.
58. The method of Claim 50, wherein said plurality of packet processing blocks further includes a Packet Processing Unit (PPU) for:

   extracting a header of a packet;

   pointing to a portion of the header of a predetermined width using a predetermined index of a bit location in the header;

   comparing the data represented by the portion of the header with at least one predetermined value; and

   declaring a match when the result of the comparison is true.

59. The method of Claim 58, wherein said Packet Processing Unit further includes means for accessing an external CAM or RAM.

60. The system of Claim 59, wherein said plurality of packet processing blocks further includes a Packet Modification Unit (PMU) for:

   extracting a packet;

   pointing to a portion of the packet of a predetermined width using a predetermined index of a bit location in the packet; and

   modifying the portion of the packet.

61. The system of Claim 60, wherein said plurality of packet processing blocks further includes a Decision and Forwarding Unit (DFU) for performing one of drop, queue, and forwarding operations on at least one packet.
1/9

1. Receive User Requirements and Specifications
2. Create Design Using Packet Processing Blocks
3. Enter Design into a Connection Document
4. Configure Packet Processing Blocks Using a Preprocessor
5. Instantiate Each Processing Block and Make Connections Between Blocks in a Top Level File
6. Does the Customer Want Packet Processing Block Files?
   - No: Generate NETLIST
   - Yes: Deliver Packet Processing Block Files and Top Level File to Customer
7. Deliver NETLIST to Customer
8. Does the Customer Want an Integrated Circuit?
   - No: End
   - Yes: Run Through Place and Route Program
9. Deliver Integrated Circuit to Customer

FIG. 1
FIG. 2C

PMU Generation

- Data Bus Width: 32
- Start of Packet Width: 1
- End of Packet Width: 4
- Max. Header Words: 8
- Operation: Insert/Remove/Modify
- Result Width: 8
- Result Expression: 5

Generate | Cancel

FIG. 2D

DFU Generation

- Start of Packet Width: 1
- End of Packet Width: 4
- Data Bus Width: 32
- No. of Interfaces: 8
- Result Expression: R[10]

Generate | Cancel
INTERNATIONAL SEARCH REPORT

International application No

PCT/US07/12583

A CLASSIFICATION OF SUBJECT MATTER
IPC(8) - G06F 07/38 (2007.10)
USPC - 712/221

According to International Patent Classification (IPC) or to both national classification and IPC

B FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC(8) - G06F 07/38 (2007.10)
USPC - 712/221, 220, 711.1/138

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
Databases USPTO WEST System (US, USPG-PUB, EPO, DERWENT), MicroPatent

C DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim</th>
</tr>
</thead>
</table>

D Further documents are listed in the continuation of Box C

Date of the actual completion of the international search
13 November 2007

Date of mailing of the international search report
15 JAN 2008

Name and mailing address of the ISA/US
Mail Stop PCT, Attn: ISA/US, Commissioner for Patents
P.O. Box 1450, Alexandria, Virginia 22313-1450
Facsimile No 571-273-3201

Authorized officer
Blame R Copenheaver
PCT Helpdesk 571-272-4300
PCT OSP 571-272-7774

Form PCT/ISA/2 10 (second sheet) (April 2005)