METHOD AND APPARATUS FOR ACCELERATED FORMAT TRANSLATION OF DATA IN A DELIMITED DATA FORMAT

Abstract: Various methods and apparatuses are described for performing high speed format translations of incoming data, where the incoming data is arranged in a delimited data format. As an example, the data in the delimited data format can be translated to a fixed field format using pipelined operations. As another example, the data in the delimited data format can be translated to a mapped variable field format using pipelined operations. A reconfigurable logic device can be used in exemplary embodiments as a platform for the format translation.

Figure 2

- Incoming Byte Stream (200) (In Delimited Data Format)
- Translation Engine (202)
- Reformatted Byte Stream (204) (In New Format to Improve Downstream Processing Performance)
Method and Apparatus for Accelerated Format Translation of Data in a Delimited Data
Format

Cross-Reference and Priority Claims to Related Patent Applications:

This patent application claims priority to (1) U.S. provisional patent application serial no. 61/793,285, filed March 15, 2013, and entitled "Method and Apparatus for Accelerated Format Translation of Data in a Delimited Data Format", and (2) U.S. provisional patent application serial no. 61/717,496, filed October 23, 2012, and entitled "Method and Apparatus for Accelerated Format Translation of Data in a Delimited Data Format", the entire disclosures of each of which are incorporated herein by reference.

Introduction:

A delimited data format is a common format used for passing data between data processing systems or over networks, particularly with respect to passing record-oriented data. Delimited data formats are platform-independent, and they use a very simple set of tags to represent data. With a delimited data format, data characters are organized into a plurality of fields. A field delimiter (FDL) character is used to separate data fields, a record delimiter (RDL) character is used to separate records, and a shield character is used to shield data characters within data fields that also happen to serve as the field delimiter character or the record delimiter character.

The comma separated value (CSV) format is a common delimited data format. With the CSV format, a comma is typically used as the FDL character, a newline is typically used as the RDL character, and a quotation mark is typically used as the shield character. However, other characters can be employed. For example, a pipe or tab character as the FDL character, an apostrophe character as the shield character, etc. Figure 1 shows an exemplary portion of a record in a delimited data format.

In the example of Figure 1, the record is a patient medical record 100 comprising a plurality of different fields (e.g., name, address, etc.). The data from this record 100 can be represented in the CSV format via data 102 in Figure 1. Each field 104i of the record can be separated by the FDL character 106. However, it may be the case that the character used as the FDL character 106 also exists within the data as a data character. In the example of Figure 1, this is shown by the commas 110 that are present in the data for Fields 1 and 3 (\044, and \043). In such situations, to prevent a misinterpretation of these commas as field delimiters, the CSV format operates to use a shield character 108 at the start and end of the
field that contains the data character 10 which matches the FDL character 106. In the example of Figure 1, quote marks serve as the shield character 108. Thus, the data St. Louis, MO becomes "St. Louis, MO". The use of shield characters raises another possible misinterpretation with respect to data characters 112 in a field that happen to match the shield character 108 (see the quotation marks used for the data string ("Jim") in Field 1 (1000). To prevent a misinterpretation of these quotation marks as shield characters, the CSV format also operates to use a shield character 108 adjacent the data character that happens to match the shield character. Thus, the data string ("Jim") appears as ("Jim") in the CSV format.

Delimited data formats present significant challenges in connection with processing the delimited data using software. The inherently serial process of moving byte by byte through a file to look for delimiters and shield characters does not map well to general purpose processors. For example, suppose it is desired to validate whether the zip code field of the file shown in Figure 1 contains a valid zip code. A software-based system would need to process each byte of the file up through Field 4 (104) in order to know that Field 4 has been located. Only then can the processing software validate the zip code data. This byte-by-byte processing requirement creates a bottleneck that detracts from the throughput of a processing system.

As solution to this problem, the inventors disclose various techniques for performing high speed format translations of incoming data, where the incoming data is arranged in a delimited data format.

In accordance with an exemplary aspect disclosed herein, the data in the delimited data format can be translated into outgoing data having a structured format, the structured format being configured to permit a downstream processing component to jump directly to a field of interest in the outgoing data without requiring that component to analyze all of the bytes leading up to the field of interest.

An example of a structured format that can be used toward this end is a fixed field format. With a fixed field format, each field of the outgoing data has a fixed length and is populated with data characters that belong to the same field of the incoming data. If there are not enough data characters for that incoming field to fill the fixed length of the outgoing field, then padding characters can be added to the outgoing field. By employing fields of a fixed length, any downstream processing can quickly and easily target specific fields of the outgoing data for further processing by simply jumping to the location of the targeted field. Because the fixed field layout is well-defined, a downstream processing component will be
able to know the byte offset for the field of interest, which means that only simple pointer arithmetic would be needed for the processing component to jump to the field of interest.

Another example of a structured format that can be used is a mapped variable field format, where the fields of a record can be of variable length. With a mapped variable field format, each field of the outgoing data can have a variable length based on the amount of data to be populated into the field. Header information can then be used to identify where the field and record boundaries are located (such as through the use of record length and field offset identifiers) to permit a downstream processing component to jump directly to a field of interest in the outgoing data without requiring that component to analyze all of the bytes leading up to the field of interest.

In an exemplary embodiment, a reconfigurable logic device can be employed to perform this data translation. As used herein, the term "reconfigurable logic" refers to any logic technology whose form and function can be significantly altered (i.e., reconfigured) in the field post-manufacture. This is to be contrasted with a general purpose processor (GPP), whose function can change post-manufacture, but whose form is fixed at manufacture. An example of a reconfigurable logic device is a programmable logic device (PLD), such as a field programmable gate array (FPGA). As used herein, the term "general-purpose processor" (or "GPP") refers to a hardware device having a fixed form and whose functionality is variable, wherein this variable functionality is defined by fetching instructions and executing those instructions, of which a conventional central processing unit (CPU) is a common example. Exemplary embodiments of GPPs include an Intel Xeon processor and an AMD Opteron processor. Furthermore, as used herein, the term "software" refers to data processing functionality that is deployed on a GPP or other processing devices, wherein software cannot be used to change or define the form of the device on which it is loaded. By contrast, the term "firmware", as used herein, refers to data processing functionality that is deployed on reconfigurable logic or other processing devices, wherein firmware may be used to change or define the form of the device on which it is loaded.

Furthermore, the data translation task can be broken down into a plurality of subtasks, where each subtask can be performed by a plurality of data processing modules arranged to operate in a pipelined fashion with respect to each other. Thus, while a downstream module in the pipeline is performing a subtask on data that was previously processed by an upstream module in the pipeline, the upstream module in the pipeline can be simultaneously performing its subtask on more recently received data. An exemplary data translation pipeline can comprise (1) a first module configured to convert the incoming data arranged in
the delimited data format to an internal format stripped of the field delimiter characters and the record delimiter characters of the incoming data while preserving the data characters of the incoming fields, (2) a second module downstream from the first module, the second module configured to remove the shield characters from the converted data having the internal format, and (3) a third module downstream from the second module, the third module configured to translate the output of the second module to the outgoing data having the fixed field format or the mapped variable field format.

Through such a modular approach, the pipeline is amenable to accelerated data translation via any of a number of platforms. As mentioned above, reconfigurable logic can be used as a platform for deploying the modules as hardware logic operating at hardware processing speeds via firmware deployed on a reconfigurable logic device. Moreover, such a pipeline is also amenable to implementation on graphics processor units (GPUs), application-specific integrated circuits (ASICs), chip multi-processors (CMPs), and other multi-processor architectures.

The inventors also disclose that the pipeline can be configured to ingest and process multiple characters per clock cycle. This data parallelism can be another source for acceleration relative to conventional solutions.

These and other features and advantages of the present invention will be described hereinafter to those having ordinary skill in the art.

Brief Description of the Drawings:

Figure 1 depicts an example of data organized into a delimited data format.

Figure 2 depicts an exemplary translation engine in accordance with a disclosed embodiment.

Figure 3 depicts an exemplary system comprising a translation engine and a data processing stage downstream from the translation engine.

Figure 4 depicts an exemplary system comprising a translation engine, a data processing stage downstream from the translation engine, and a translation engine downstream from the data processing stage.

Figure 5 depicts an exemplary system similar to that of Figure 4, specifically showing field-specific data processing operations within the data processing stage.

Figure 6 depicts an exemplary fixed field format.

Figure 7 depicts the data of Figure 1 organized in a fixed field format.

Figures 8(a) and (b) depict examples of suitable platforms for the translation engine.
Figures 9(a) and (b) depict exemplary printed circuit boards for use as a coprocessor for the embodiments of Figures 8(a) and (b).

Figure 10 depicts an example of how a firmware pipeline can be deployed across multiple reconfigurable logic devices.

Figure 11 depicts an example of a pipeline that can be deployed by a translation engine to convert delimited data to fixed field data.

Figure 12 depicts an exemplary pipeline for a variable record gate (VRG) module.

Figure 13 depicts a state machine for an exemplary quote masker circuit.

Figures 14(a) and (b) depict exemplary delimiter finder circuits.

Figure 15 depicts an exemplary shift register logic circuit and an exemplary field identifier logic circuit.

Figure 16 depicts an exemplary quote removal (QRM) module.

Figure 17(a) depicts an exemplary variable-to-fixed (V2F) module.

Figure 17(b) depicts a state machine for the V2F module of Figure 17(a).

Figure 18 depicts an exemplary pipeline that can be deployed by a translation engine to convert fixed field data to delimited data.

Figure 19 depicts an exemplary fixed-to-variable (F2V) module.

Figure 20 depicts an exemplary quote addition (QAD) module.

Figure 21 depicts an exemplary variable inverse record gate (VIR) module.

Figure 22 depicts an exemplary arrangement for a processing module, where the processing module includes a bypass path and a processing path.

Figure 23 depicts an example of a pipeline that can be deployed by a translation engine to convert delimited data to mapped variable field data.

Figure 24 depicts an exemplary mapped variable field format.

Figure 25 depicts exemplary code for jumping directly to a desired field in mapped variable field data.

Figure 26 depicts an exemplary variable-to-mapped (V2M) module.

Figure 27 depicts an exemplary pipeline that can be deployed by a translation engine to convert mapped variable field data to delimited data.

Figure 28 depicts an exemplary mapped-to-variable (M2V) module.

Figure 29 depicts an example of a pipeline that can be deployed by a translation engine to convert delimited data to a structured data format, wherein a hardware-accelerated data processing stage operates on the output variable format data output from the QRM module.
Figure 30 depicts an example of how field-specific regular expression pattern matching can be performed by a hardware-accelerated regular expression pattern matching engine.

Detailed Description:

Figure 2 depicts an exemplary translation engine 202 that is configured to translate an incoming byte stream 200 having a delimited data format into a reformatted byte stream 204 having the structured format that is geared toward high performance downstream processing such that a downstream processing component can jump directly to fields without analyzing the data characters of the reformatted byte stream 204. As noted, this structured format can be a format such as a fixed field format or a variable mapped field format. Once again, Figure 1 shows exemplary data that can serve as byte stream 200. As will be understood, the bytes of the byte stream 200 can serve as data characters, record delimiters characters, field delimiter characters, and shield characters.

Figure 3 shows the delivery of the reformatted byte stream 204 to a data processing stage. The data processing stage will be able to select fields of the reformatted byte stream for targeted processing without further analyzing the data characters of the reformatted byte stream 204, thereby greatly improving the throughput performance of the system. The data processing stage then performs data processing operations on the selected fields to generate a processed byte stream 302. This processed byte stream 302 can also exhibit the structured format of the reformatted byte stream 204. The data processing stage 300 can be implemented in software via a GPP, in firmware via reconfigurable logic, or any other platform desired by a practitioner.

For example, the data processing stage can be configured to perform various processing operations as part of data quality checking in connection with extract, transfer, and load (ETL) operations for a database. Some exemplary processing operations can include:

- Address Validation: A field expected to contain an address can have the address data validated as to whether it exhibits a correct postal service-recognized address format.
- Email Validation: A field expected to contain an email address can be validated as to whether it exhibits a correct email address format.
- Date Validation: A field expected to contain a date can be validated as to whether it exhibits a date in the correct range and format.
• Query/Replace: The data characters in a select field can be translated from one set to another set (e.g., mapping codes from one code set to another code set or replacing codes with natural language descriptions of such codes).

• Field Masking/Tokenization: The data characters in a selected field can be obfuscated or tokenized for security purposes.

• Filtering/Searching: The data characters in selected fields can be matched against various search criteria.

It should be understood that these are but a few of exemplary data processing operations that can be performed by the data processing stage 300.

Furthermore, it should be understood that these data processing operations can be legacy data processing operations that are implemented in software on processors of a practitioner. Also, if desired, a practitioner can deploy such data processing operations via reconfigurable logic to achieve still further acceleration. Examples of hardware-accelerated data processing operations that can be performed by the data processing stage 300 include data processing operations such as regular expression pattern matching, approximate pattern matching, encryption/decryption, compression/decompression, rule processing, data indexing, and others, such as those disclosed by U.S. Pat. Nos. 7,636,703, 7,702,629, 8,095,508 and U.S. Pat. App. Pubs. 2007/0237327, 2008/01 14725, 2009/0060197, and 2009/0287628, the entire disclosures of each of which being incorporated herein by reference.

Figure 4 depicts an exemplary embodiment where the processed byte stream 302 is translated by a translation engine 400 into a byte stream 402 having a target format. For example, a practitioner may desire that the system re-translate the byte stream 302 back into a delimited data format. In such an embodiment, the translation engine 400 can perform the complementary inverse of the translation operations performed by translation engine 202 to return the data to the delimited data format. Translation engine 400 can also be hardware-accelerated via reconfigurable logic and modularized via processing modules arranged in a pipeline as explained in connection with the translation engine 202.

Figure 5 depicts a similar system that highlights how the output of the translation engine 202 can feed field-specific data processing operations 500 at the data processing stage 300. It should also be understood that for software-based embodiments of the data
processing stage 300, record-specific threads can be running in parallel to provide additional
acceleration.

Figure 6 depicts an exemplary fixed field format that can be exhibited by byte stream
202 can operate to populate each field of the fixed field output with data characters of the
corresponding field in the byte stream having the delimited data format. Should there not be
enough data characters in the byte stream to fill the fixed field, padding characters can be
added to complete the field. In the event that there is insufficient space in the fixed field for
all data characters in a field of the delimited data format byte stream, the translation engine
202 can flag a data overflow condition and take appropriate measures through exception
handle logic. Figure 7 depicts an example where the data of Figure 1 has been translated
into a fixed field format where each field has a fixed length of 24 bytes. It should be well
understood that a field length of 24 bytes is exemplary only, and other field lengths can be
readily employed. It should also be understood that each field need not have the same fixed
length. For example, a practitioner can choose to define a field length of 36 bytes for Field 1,
a field length of 64 bytes for Field 2, a field length of 64 bytes for Field 3, a field length of 16
bytes for Field 4, and so on. A practitioner can choose such fixed field lengths for each field
based on expected characteristics of the data.

In an embodiment where the translation engine 202 is implemented in reconfigurable
logic, examples of suitable platforms for such a translation engine 202 are shown in Figures
8(a) and (b). Figure 8(a) depicts a system 800 employs a hardware-accelerated data
processing capability through coprocessor 840 to process the incoming byte stream 200.
Within system 800, a coprocessor 840 is positioned to receive byte stream 200 that streams
into the system 800 from a network 820 (via network interface 810).

The computer system defined by processor 812 and RAM 808 can be any commodity
computer system as would be understood by those having ordinary skill in the art. For
example, the computer system may be an Intel Xeon system or an AMD Opteron system.
Thus, processor 812, which serves as the central or main processor for system 800, preferably
comprises a GPP (although this need not be the case).

In this exemplary embodiment, the coprocessor 840 comprises a reconfigurable logic
device 802. Preferably, the byte stream 200 streams into the reconfigurable logic device 802
by way of system bus 806, although other design architectures are possible (see Figure 9(b)).
The reconfigurable logic device 802 can be a field programmable gate array (FPGA),
although this need not be the case. System bus 806 can also interconnect the reconfigurable

- 8 -
logic device 802 with the processor 812 as well as RAM 808. In an exemplary embodiment, system bus 806 may be a PCI-X bus or a PCI-Express bus, although this need not be the case.

The reconfigurable logic device 802 has firmware modules deployed thereon that define its functionality. The firmware socket module 804 handles the data movement requirements (both command data and target data) into and out of the reconfigurable logic device, thereby providing a consistent application interface to the firmware application module (FAM) chain 850 that is also deployed on the reconfigurable logic device. The FAMs 850i of the FAM chain 850 are configured to perform specified data processing operations on any data that streams through the chain 850 from the firmware socket module 804. Examples of FAMs that can be deployed on reconfigurable logic in accordance with the exemplary translation engine 202 are described below.

The specific data processing operation that is performed by a FAM is controlled/parameterized by the command data that FAM receives from the firmware socket module 804. This command data can be FAM-specific, and upon receipt of the command, the FAM will arrange itself to carry out the data processing operation controlled by the received command. For example, within a FAM that is configured to perform a shield character find operation, the FAM's shield character find operation can be parameterized to define the character that will be used as the shield character. In this way, a FAM that is configured to perform a shield character find operation can be readily re-arranged to perform a different shield character find operation by simply loading parameters for a new shield character in that FAM. As another example, a command can be issued to the one or more FAMs that are configured to find a delimiter character (e.g., a record delimiter character or field delimiter character) so that the FAM can be tailored to different delimiter characters without requiring a full reconfiguration of the reconfigurable logic device.

Once a FAM has been arranged to perform the data processing operation specified by a received command, that FAM is ready to carry out its specified data processing operation on the data stream that it receives from the firmware socket module. Thus, a FAM can be arranged through an appropriate command to process a specified stream of data in a specified manner. Once the FAM has completed its data processing operation, another command can be sent to that FAM that will cause the FAM to re-arrange itself to alter the nature of the data processing operation performed thereby. Not only will the FAM operate at hardware speeds (thereby providing a high throughput of data through the FAM), but the FAMs can also be flexibly reprogrammed to change the parameters of their data processing operations.
The FAM chain 850 preferably comprises a plurality of firmware application modules (FAMs) 850a, 850b, … that are arranged in a pipelined sequence. However, it should be noted that within the firmware pipeline, one or more parallel paths of FAMs 850i can be employed. For example, the firmware chain may comprise three FAMs arranged in a first pipelined path (e.g., FAMs 850a, 850b, 850c) and four FAMs arranged in a second pipelined path (e.g., FAMs 850d, 850e, 850f, and 850g), wherein the first and second pipelined paths are parallel with each other. Furthermore, the firmware pipeline can have one or more paths branch off from an existing pipeline path. A practitioner of the present invention can design an appropriate arrangement of FAMs for FAM chain 850 based on the processing needs of a given translation operation.

A communication path 830 connects the firmware socket module 804 with the input of the first one of the pipelined FAMs 850a. The input of the first FAM 850a serves as the entry point into the FAM chain 850. A communication path 832 connects the output of the final one of the pipelined FAMs 850m with the firmware socket module 804. The output of the final FAM 850m serves as the exit point from the FAM chain 850. Both communication path 830 and communication path 832 are preferably multi-bit paths.

The nature of the software and hardware/software interfaces used by system 800, particularly in connection with data flow into and out of the firmware socket module are described in greater detail in U.S. Patent Application Publication 2007/0174841, the entire disclosure of which is incorporated herein by reference.

Figure 8(b) depicts another exemplary embodiment for system 800. In the example of Figure 8(b), system 800 includes a data store 842 that is in communication with bus 806 via disk controller 814. Thus, the byte stream 200 that is streamed through the coprocessor 840 may also emanate from data store 842. Furthermore, the data store 842 can be the target destination for the output from the translation engine 202 and/or the data processing stage 300 if desired by a practitioner. Data store 842 can be any data storage device/system, but it is preferably some form of mass storage medium. For example, data store 842 can be a magnetic storage device such as an array of Seagate disks.

Figure 9(a) depicts a printed circuit board or card 900 that can be connected to the PCI-X or PCI-e bus 806 of a commodity computer system for use as a coprocessor 840 in system 800 for any of the embodiments of Figures 8(a)-(b). In the example of Figure 9(a), the printed circuit board includes an FPGA 802 (such as a Xilinx Virtex 5 or an Altera Stratix V FPGA) that is in communication with a memory device 902 and a PCI-e bus connector.
A preferred memory device 902 comprises SRAM and DRAM memory. A preferred
PCI-X or PCI-e bus connector 904 is a standard card edge connector.

Figure 9(b) depicts an alternate configuration for a printed circuit board/card 900. In
the example of Figure 9(b), one or more network controllers 908, and one or more network
connectors 910 are also installed on the printed circuit board 900. Any network interface
technology can be supported, as is understood in the art. Hardware logic can be used as the
internal connector between the FPGA, memory, and network controller. It should be noted
that a disk interface technology can be used in addition to or in place of the network
controller and network connector shown in Figure 9(b).

It is worth noting that in either the configuration of Figure 9(a) or 9(b), the firmware
socket 804 can make memory 902 accessible to the bus 806, which thereby makes memory
902 available for use by an OS kernel as the buffers for transfers to the FAMs from a data
source with access to the bus. It is also worth noting that while a single FPGA 802 is shown
on the printed circuit boards of Figures 9(a) and (b), it should be understood that multiple
FPGAs can be supported by either including more than one FPGA on the printed circuit
board 900 or by installing more than one printed circuit board 900 in the system 800. Figure
10 depicts an example where numerous FAMs in a single pipeline are deployed across
multiple FPGAs.

Translation Engine 202 - Fixed Field Format

Figure 11 depicts an exemplary pipeline that can be employed by the translation
engine 202 to convert delimited data to a fixed field format. The pipeline can comprise (1) a
first module configured to convert the incoming data arranged in the delimited data format to
an internal format stripped of the field delimiter characters and the record delimiter characters
of the incoming data while preserving the data characters of the incoming fields, (2) a second
module downstream from the first module, the second module configured to remove the
shield characters from the converted data having the internal format, and (3) a third module
downstream from the second module, the third module configured to translate the output of
the second module to the outgoing data having the fixed field format. In this example, the
first module can be referred to as a variable record gate (VRG) module, the second module
can be referred to as a quote removal module (QRM) given that quote marks are used as the
shield character in this example, and the third module can be referred to as a variable-to-fixed
(V2F) module. Each module can be configured to operate in parallel in a pipelined manner.
As such, while the V2F module is operating on data previously processed by the VRG and

- 11 -
QRM modules, the QRM module is operating on data previously processed by the VRG module, and the VRG module is operating on newly received data, and so on as data continues to stream into the pipeline.

**VRG Module:**

Figure 12 depicts an exemplary arrangement for a VRG module. The components of the VRG module shown in Figure 12 can also be implemented as modular circuits in a pipelined chain. The VRG module can generate an output byte stream that is marked with control data to identify information such as which bytes correspond to a start of record, an end of record, a start of field, and an end of field. Thus, downstream modules need not reparse the bytes to gather that information. With reference to the operations described herein, it should be understood that the various circuit components of the VRG module can process the bytes of the byte stream in chunks (e.g., 64 bit (8 byte) or 128 bit (16 byte) chunks) per clock cycle. Thus, the component circuits can be configured to provide data parallelism by ingesting and processing multiple characters in the byte stream per clock cycle.

A first circuit in the VRG can be configured to process the shield characters that are present in the byte stream 200 to distinguish between the bytes that are eligible for downstream consideration as to whether they correspond to a delimiter character (e.g., the bytes that are present in a field that has not been shielded by a shield character) and the bytes that are ineligible for downstream consideration as to whether they correspond to a delimiter character (e.g., the bytes that are present in a field that has been shielded by a shield character). In this example, such a circuit can be referred to as a quote masker (QM) circuit.

A second circuit in the VRG that is downstream from the QM circuit can be configured to process the output of the QM circuit to locate the presence of delimiter characters in the byte stream. In this example, such a circuit can be referred to as a delimiter finder (DLF) circuit.

A third circuit in the VRG that is downstream from the DLF circuit can be configured to process the output of the DLF circuit to detect empty fields, remove the delimiter characters from the byte stream, and mark the bytes which correspond to data characters at the start of a record and end of a record. In this example, such a circuit can be referred to as a shift register logic (SRL) circuit.

A fourth circuit in the VRG that is downstream from the SRL circuit can be configured to process the output of the SRL circuit to generate a field identifier that identifies...
which field each data character of the byte stream belongs to and mark the bytes which correspond to data characters at the start of a field and end of a field. In this example, such a circuit can be referred to as a field ID logic (FIDL) circuit.

Figure 13 provides additional detail regarding the QM circuit. Once again, in this example, the shield character is a quote mark, so quotes will be used throughout this example to refer to the shield character. However, it should be understood that characters other than quote marks could be used as the shield character. As noted, the QM circuit is configured to mark each byte of the byte stream with an indicator of whether or not it is a valid candidate as a delimiter (i.e. NOT protected by the shield character). Figure 13 depicts exemplary state diagrams that can be employed by the QM circuit to implement this task. Figure 13 shows two states: CLOSED ("Close Quote") and OPEN ("Open Quote"). In the CLOSED state, which is the initialization state, the quotes have been closed, and characters are open for consideration as a delimiter. While in this state, any character that is not a quote character will be marked with a "Delimiter Valid" (DV) flag set to true, meaning that the character is a candidate delimiter character. Upon observing a quote character, this machine will transition to the OPEN state, meaning that the data is inside a quote and thus shielded by the quote character. Any character other than a quote character will be marked with a DV flag set to false, indicating that the character is not a candidate to be a delimiter. Upon detection of another quote character, this state machine will transition back to CLOSED, meaning that next character is no longer being shielded by quote marks. This toggling behavior also accommodates the possible presence of double quotes in the byte stream which are meant to internally shield data characters that happen to be quote marks (see the portion of Field 1 in Figure 1 comprising ""Jim"" - all of Field 1 has been shielded by quote marks, so that quote mask should not change upon encountering the internal double quotes in the byte stream).

From the open data state, if a quote mark is detected in the byte stream, the state machine will transition to the closed quote state, while any other character in the byte stream means the state machine will remain in the open data state.

It should be understood with the diagram of Figure 13 that one can ignore the DV status bits for the actual quote characters because configuration restrictions prevent shield characters and delimiter characters from overlapping. In this model, some quotes will be marked as valid, and others will not, but regardless of their marking they will never be considered a delimiter, as will be understood upon review of Figure 14.
The QM circuit thus outputs the bytes of the byte stream where each byte is
associated with a DV flag to indicate whether the associated byte should be processed to
assess whether it contains a delimiter character.

Figure 14(a) provides additional detail regarding an example of a DLF circuit. A data
register can be loaded with the current byte under consideration. A mask register can be
loaded with the DV flag associated with the byte loaded in the register. A first match key
register can be loaded with the RDL character, and a second match key register can be loaded
with the FDL character. The byte in the data register can be logically ANDed with the DV
data in the mask register. Thus, from the description above, (1) if a byte has been identified
by the QM register as being eligible for consideration as to whether it contains a delimiter
character, its associated DV flag is equal to 1, and the output of the AND operation will pass
the byte to a matching stage, and (2) if a byte has been identified by the DV register as being
ineligible for consideration as to whether it contains a delimiter character, its associated DV
flag is equal to 0, and the output of the AND operation will pass a zero value to a matching
stage (thereby causing the matching stage to find no match with respect to the delimiter
characters which are assumed to be different characters than the zero value).

A first comparator in the matching stage compares the RDL character with the AND
operation output. Based on the outcome of that comparison, a control signal can be applied
to a multiplexer to govern whether an RDL flag associated with the byte under consideration
will go to a state indicating the byte under consideration corresponds to the RDL character
(e.g., high) or to a state indicating the byte under consideration does not correspond to the
RDL character (e.g., low). Similar matching logic can be employed to test the AND
operation output against the FDL character to yield an FDL flag associated with the byte
under consideration. Furthermore, for embodiments where the DLF circuit is implemented in
reconfigurable logic, the parallelism capabilities provided by the reconfigurable logic mean
that the RDL character matching operation and the FDL character matching operation can be
performed simultaneously.

Thus, the output of the DLF circuit shown by Figure 14(a) will be a stream of
outgoing bytes and their associated RDL and FDL flags.

Figure 14(b) depicts an example of a DLF circuit where the DLF circuit is configured
to ingest multiple characters per clock cycle (e.g., 3 characters per clock cycle as shown in
the example of Figure 14(b)). Thus, the data shift register through which the byte stream is
passed will have a multi-character data width (once again, a 3 character width in this
example). Similarly, the data shift register through which the DV mask is passed will also
have a data width that corresponds to the data width of the data shift register for the byte stream. Each clock cycle, the 3 characters of the data shift register and the DV masks corresponding to those three characters can be processed in parallel through replicated AND gates, comparators, and multiplexers to test the characters for matches against the RDL character and the FDL character. Upon completion of a cycle, the data shift registers can be configured to perform a shift by three characters to load the next set of characters for processing.

Figure 15 provides additional detail regarding the SRL circuit and the FIDL circuit. The SRL circuit and the FIDL circuit can cooperate to pack the data headed downstream. FDL and RDL characters are removed from the byte stream, a count of skipped fields (e.g., empty fields) is generated, and the data characters that serve as field and record boundaries are marked. Further still, each field can be tagged with a field identifier for use by downstream processing. The output of the FIDL circuit can thus be the data characters of the byte stream and control data associated with those characters. This control data can take the form of a structured module chain interface (SMCI) protocol. The SMCI protocol can include a start of field (SOF) data, end of field (EOF) data, start of record (SOR) data, end of record (EOR) data, field identifier data, and count data, the count data being indicative of how many bytes should be consumed (e.g., how many bytes are valid in a transaction (transmission of a data word). For a data width of 8 bytes, for example, the count can range from 0-8 depending upon how many of the bytes are valid.

The SRL circuit of Figure 15 can employ three shift registers - a data shift register through which the characters of the byte stream are pushed, a RDL shift register through which the RDL flag data is pushed, and a FDL shift register through which the FDL flag data is pushed.

Logic 1500 can be configured to:

- Find the "leading" delimiter in the FDL or RDL register (the first character in the data register for which the corresponding FDL or RDL flag is high). The record/field found flag can be set as appropriate when a leading delimiter is found.
- Check the RDL and FDL flags following the leading delimiter to determine if an empty or skipped field/record is present. An empty/skipped field is a field with no data. Such an empty/skipped field appears in the byte stream as back to back FDL characters (as indicated by the FDL flag data). An empty/skipped record is a record
with no data. Such an empty/skipped record appears in the byte stream as back to back RDL characters (as indicated by the RDL flag data).

- If there are back to back delimiters in the byte stream, determine a count of the empty fields/records and pull those off the shift register. This count is communicated as the Fields Skip output of the SRL circuit in Figure 15.

- If non-empty fields are found, use the position of the delimiter (communicated as a bit in the field/record found register) to indicate how much data to pull off for the given field. This information can be communicated as the Data Count output of the SRL circuit in Figure 15.

The shift logic 1502 can then operate in a fashion to cause the shift register to consume or strip off the delimiters. Thus, when delimiter characters are found in the byte stream based on the SMCI data, the shift logic 1502 can cause the shift register to shift out the delimiter characters while holding a data valid signal low. In this fashion, the delimiter characters are effectively dropped from the outgoing data stream.

The FIDL circuit then takes in the output of the SRL circuit in a register output and processes that output to generate an EOR flag and EOF flag for the data characters in the byte stream. Based on the delimiter following the data being pulled, the logic can determine whether to send an EOF or EOR marker (by checking the delimiter that triggered then end of the field/record). Logic 1504 and 1506 operate as a counter that increments the Field ID each time a new field in a record is encountered (in response to the skipped count, the EOR flag and the EOF flag). Thus, the Field ID can operate as an array index such that the first field has a Field ID of 0, the second field has a Field ID of 1, and so on. Furthermore logic 1508 operates to generate SOR and SOF flags from the EOR and EOF flags. The SOR/SOF/EOF/EOR data, count data, and Field ID data produced by the FIDL circuit can serve as the SMCI protocol control data associated with the outgoing bytes.

It should also be understood that the VRG module can be internally pipelined such that the QM circuit, the DLF circuit, the SRL circuit, and the FIDL circuit are configured to operate simultaneously in a pipelined fashion.

**QRM Module:**

Figure 16 depicts an exemplary arrangement for a QRM module. The QRM module is configured to strip the quotes used as the start and end of a field as shield characters and convert two consecutive quotes into a single quote.
The quote finder logic 1600 receives the data and SMCI signal from the VRG module output, and performs matching operations on the data to locate the characters that match the quote character. If a quote character in the data stream is at the start of a field (as indicated by the SOF flag in the SMCI control data), then the quote finder logic 1600 can mark that quote character for removal. If a quote character in the data stream is at the end of a field (as indicated by the EOF flag in the SMCI control data), then the quote finder logic 1600 can also mark that quote character for removal. Furthermore, if consecutive quote characters are found in the data stream, then the quote finder logic can mark the first quote for removal. Alternatively, the quote finder logic can be configured to merely mark the locations of quote characters in the data stream.

Thus, the quote finder logic 1600 provides the data stream, its associated SMCI control data, and the quote removal markers to quote conversion logic 1602. The quote conversion logic is configured to remove the single quotes from the data stream and replace the double quotes with single quotes. A shift register repacks the data from the quote conversion logic to accommodate the quote removals. Thus, the output of the shift register comprises the data stream and its corresponding SMCI control data.

The QRJVI module can also be internally pipelined such that the quote finder logic 1600, the quote conversion logic 1602 and shift register operate simultaneously in a pipelined fashion.

**V2F Module:**

Figure 17(a) depicts an exemplary arrangement for a V2F module. The V2F module can hold a map of field lengths to use for the fixed field format. The V2F module can use this map to fit the fields of the data stream to their appropriate length in accordance with the target fixed field format. The V2F module will pad out any field in the data stream shorter than the specification field length with a padding character, which can be a configurable special character. For ease of reference, these padding characters can be referred to as zeros for purposes of discussion. The V2F module will also output an overflow error for any field in the data stream longer than the specification field length.

The LUT stores a table of field widths that can be sent in from software. This table will thus have the length for each field as specified by software on startup. Thus, it should be understood that through these specified field lengths, each of the fields of the output fixed field formatted-data can have its own length that need not be the same length as the other fields. The index into this table represents the ID of a given field, and the value at that
location represents the given field length. The last field identifier, and consequently the last populated field in the LUT, is stored in a last field identifier (max_fid) which is stored separately from the LUT. It is worth noting that some fields in this table can have a specified length of zero, meaning they are to be eliminated from output data records. (This can be used to eliminate fields that are generally not present in the input data.)

An input state machine takes in the data stream and SMCI control data from the QRM module and compares it with the field identifiers from the LUT to reconcile the incoming fields with the expected fields for each record. The start of each field for the incoming data is marked in the SMCI data by the SOF flag while the end of each field is marked in the SMCI data by the EOF flag. Further still, the Field ID of the SMCI data will identify the field to which the current data of the data stream corresponds. From this information, the input state machine can transition between states of PROCESSING, COMPLETE, and OVERFLOW. Figure 17(b) depicts an exemplary state machine diagram for the input state machine of Figure 17(a).

In the PROCESSING state, if the field identifier for the incoming data (fid_in) matches the field identifier for the current field from the LUT (current_fid), then the incoming data can be sent to the output state machine for processing. However, while in the PROCESSING state, if fid_in does not match current_fid (and an EOR marker is not present), then this means that a gap in the incoming fields exists, and an empty field should be sent to the output state machine for processing. The next current_fid from the LUT is then processed.

If fid_in is greater than max_fid while the input state machine is in the PROCESSING state, the state machine transitions to the OVERFLOW state. This condition indicates that the input record included more fields than expected. While in the OVERFLOW state, the input state machine sends the overflow fields to the output state machine until an EOR marker is encountered in the incoming data. Upon encountering the EOR marker in the incoming data, the input state machine will transition back to the PROCESSING state.

If fid_in does not match max_fid and the EOR marker is present in the incoming data while the input state machine is in the PROCESSING state, this means that the incoming record had fewer fields than expected and we transition to the COMPLETE state. While in the COMPLETE state, the input state machine sends size zero fields to the output state machine and increments to the next current_fid from the LUT. Once current_fid reaches max_fid, the input state machine transitions back to the PROCESSING state.
The input state machine reports a data value indicative of the size of each identified field as it receives SOF markers from the input SMCI interface (current_field_size). For empty fields that are added to fill in a gap in a record, the current_field_size can be zero. For non-empty fields, a counter can be employed to identify how many bytes are present in each field (from the SOF and EOF markers in the SMCI control data associated with the incoming data).

The output state machine operates to fill fields with bytes of the incoming data or padding characters as necessary, and identify those fields which are overflowing with bytes of the incoming data as necessary. The output state machine can progress from a PROCESSING state (during which time the data stream fills the output data shift register that contains the output field) to a PADDING state (during which time padding characters are added to the output field) upon detection of a field incomplete condition. The field incomplete condition can occur if the current_field_size for an input field is less than the corresponding field length for the output field. Once the output field has been filled to the current_field_size, the output state machine can transition to the PADDING state.

While in the PADDING state, the remaining space in the output field is filled with padding characters until the padding characters added to the output field have caused the output field to reach the size of its field length. The output state machine can then return to the PROCESSING state.

The output state machine can also progress from the PROCESSING state to the OVERFLOW START state upon detection of a field overflow condition. The field overflow condition can occur if the current_field_size for an input field is greater than the corresponding field length for the output field. If this condition is detected, the output state machine can transition to the OVERFLOW START state. When in the OVERFLOW START state, an overflow start command (CMD) can be sent and the data shift register is flushed. The output state machine then progresses to the OVERFLOW state (during which time the overflow data is sent). Upon encountering the EOF flag for the overflowing field, the output state machine will progress to the OVERFLOW END state. During the OVERFLOW END state, an overflow end command (CMD) can be sent, and the shift register is flushed. Thus, overflowing fields are framed by overflow commands in the output data.

A command/data multiplexer is configured to provide either the CMDs from the output state machine or the content of the data shift register (SR) as an output. The state of the output state machine will govern which multiplexer input is passed as the multiplexer.
output. Thus, if the output state machine is in the OVERFLOW START or OVERFLOW END states, the multiplexer will pass command data indicative of these states to the output. While the output state machine is in the PROCESSING, PADDING, or OVERFLOW states, the multiplexer will pass the content of the output data shift register to the output.

Accordingly, the V2F will output a fixed field of data when no overflows are detected. If an overflow is detected, a CMD signal frames the overflow data so that exception handling can further process the overflowing field.

Thus, the V2F module is able to deliver the data of the input byte stream 200 to the data processing stage 300 as a byte stream in a fixed field format.

10

**Translation Engine 400 - Fixed Field Format:**

If it is desired to translate the processed data output of the data processing stage back to a delimited data format, the translation engine 400 can be configured with a pipeline of processing modules that effectively perform the inverse of the operations performed by the pipeline of Figure 11. Figure 18 depicts an exemplary pipeline that can be deployed by the translation engine 400. A fixed-to-variable (F2V) module can convert the incoming data in a fixed field format to the variable format having the SMCI control protocol. A quote addition (QAD) module downstream from the F2V module can insert shield characters into the data stream at appropriate locations as per the target delimited data format. A variable inverse record gate (VIRG) module downstream form the QAD module can insert FDL and RDL characters into the data stream at appropriate locations to thereby generate an output data stream in the target delimited data format.

Figure 19 depicts an exemplary embodiment for the F2V module. Incoming data is shifted through a shift register, and a LUT of field lengths is used to ascertain the length of each incoming field. A field creator delineates the different fields of the incoming data and generates the associated SMCI control protocol data for those fields.

Figure 20 depicts an exemplary embodiment for the QAD module. The QAD module can inspect the incoming data for shield characters and delimiter characters to insert shield characters at appropriate locations as per the delimited data format. For example, if it detects a data character within a field that does not serve as an FDL character but matches the FDL character, the QAD module will operate to wrap that field with quote marks. The QAD module can also operate to strip the incoming data of padding characters that may have been added to the fields to fillout the fixed fields. A special character logic in the QAD module can operate to detect and mark all special characters (shield characters, FDL characters, and
RDL characters) in the data stream for populating the data and header queues. A padding clipper that then culls the data stream of padding characters and shift registers can be employed to repack the outgoing data.

Figure 21 depicts an exemplary VIR module. The VIR module can take in the data output from the QAD module together with the associated SMCI control data to insert actual RDL characters and FDL characters at appropriate locations in the data stream via processing logic triggered by the SMCI control data and corresponding shift registers. Thus, the output of the VIR module will be a stream of data in the delimited data format.

Translation Engine 202 - Mapped Variable Field Format

Figure 23 depicts an exemplary pipeline that can be employed by the translation engine 202 to convert delimited data to a mapped variable field format. The pipeline can comprise (1) a first module configured to convert the incoming data arranged in the delimited data format to an internal format stripped of the field delimiter characters and the record delimiter characters of the incoming data while preserving the data characters of the incoming fields, (2) a second module downstream from the first module, the second module configured to remove the shield characters from the converted data having the internal format, and (3) a third module downstream from the second module, the third module configured to translate the output of the second module to the outgoing data having the variable mapped field format. In this example, the first module can be a VRG module as described above, and the second module can be a QRM module as described above. The third module can be referred to as a variable-to-mapped (V2M) module. Each module can be configured to operate in parallel in a pipelined manner. As such, while the V2M module is operating on data previously processed by the VRG and QRM modules, the QRM module is operating on data previously processed by the VRG module, and the VRG module is operating on newly received data, and so on as data continues to stream into the pipeline.

Figure 24 depicts an exemplary mapped variable field format that can be exhibited by byte stream 204 produced by the pipeline of Figure 23. Each record can have a variable length, wherein the record comprises data fields, also of variable length. Header information is included with the records to map the record boundaries and field boundaries. For example, a record header can include a length for the subject record and a count of the number of fields contained in the record. The field header can identify offsets into the record for each field. This can be expressed as an array of integer values, where each integer value represents the offset to a given field in the record such that the first integer in the array maps to a first field.
of the record, a second integer in the array maps to a second field of the record, and so on. The field header can then be followed by the data fields of the record. These fields can have a variable length, thereby providing for a more compact record where the need for padding bytes can be eliminated. Once again, the field offsets of the field header provide a mapping function that allows for direct access of a desired field. Thus, the translation engine 202 of Figure 23 can populate the fields and the headers with data and information to tailor the record size as appropriate for the data.

Figure 25 depicts an exemplary snippet of code that allows for direct access to a desired field of a record. To retrieve a specific field's starting address, for example, a client would simply need to index into the field array of the field header and add the indexed offset to the address of the beginning of the message (record).

**JVM Module:**

Figure 26 depicts an exemplary arrangement for a V2M module. The V2M module can convert the data in the SMCI format from the QRM module to generate outgoing data in the variable mapped field format.

Incoming data is stored in a record FIFO buffer. The record FIFO buffer also includes a register that will identify when an EOR signal is present in the SMCI information, marking the end of that record. Depending upon the maximum record size, the record FIFO buffer can be internal memory in the hardware (e.g., internal to an FPGA chip for an embodiment where the V2M module is deployed on an FPGA) or it can be external to the hardware. The size of the record FIFO should be sufficient to buffer an entire record.

Registers are also used to keep a running count of incoming field and record information so that the V2M module can track the number of fields in each record, the byte offsets of each field of the record, and the total byte length of each record. Upon encountering appropriate markers in the SMCI control data, the header FIFO buffer can be written to include information such as the field offsets and record byte length/field count.

An output state machine then operates to generate the outgoing data in the mapped variable field format using data from the record FIFO buffer to populate the record fields, and using the information in the header FIFO buffer to populate the record header and field header. Upon encountering an EOR signal in the SMCI control data, the V2M can then progress to the next record to construct the mapped variable field output.

Thus, the V2M module is able to deliver the data of the input byte stream 200 to the data processing stage 300 as a byte stream in a mapped variable field format.
Translation Engine 400 - Mapped Variable Field Format:

If, for an embodiment where mapped variable field formatting is used, it is desired to translate the processed data output of the data processing stage back to a delimited data format, the translation engine 400 can be configured with a pipeline of processing modules that effectively perform the inverse of the operations performed by the pipeline of Figure 23. Figure 27 depicts an exemplary pipeline that can be deployed by the translation engine 400 for this purpose. A mapped-to-variable (M2V) module can convert the incoming data in a mapped variable field format to the variable format having the SMCI control protocol. A QAD module as described above downstream from the M2V module can insert shield characters into the data stream at appropriate locations as per the target delimited data format. A VIR module as described above downstream from the QAD module can insert FDL and RDL characters into the data stream at appropriate locations to thereby generate an output data stream in the target delimited data format.

Figure 28 depicts an exemplary embodiment for the M2V module.Incoming data is processed by an input state machine to interpret the record header and field header of each record to identify where the field boundaries in the data exist. Record header data and field header data are stored in staging registers. Output logic can process the data in the various registers to remove the header data and generate appropriate SMCI control data for the field data that is parsed directly from the input stream.

Hardware-Accelerated Data Processing Stage

It should be understood that, in embodiments where the field-specific data processing stage 300 is implemented in hardware (such as on an FPGA), the data processing stage 300 can take the form of a hardware-accelerated data processing stage 2900 as shown in Figure 29. Such a hardware-accelerated data processing stage 2900 can tap into the output of the QRM module to operate on the data internally formatted to the SMCI protocol.

Examples of hardware-accelerated data processing that can be performed by stage 2900 include data processing operations such as regular expression pattern matching, approximate pattern matching, encryption/decryption, compression/decompression, rule processing, data indexing, and others, such as those disclosed by the above-referenced and incorporated U.S. Pat. Nos. 7,636,703, 7,702,629, 8,095,508 and U.S. Pat. App. Pubs. 2007/0237327, 2008/01 14725, 2009/0060197, and 2009/0287628. This hardware-
accelerated data processing can be field-specific by leveraging the information present in the SMCI signal to identify record and field boundaries.

An example of field-specific hardware-accelerated data processing is shown by Figure 30 with respect to regular expression pattern matching. A practitioner may have a desire to perform regular expression pattern matching with respect to different patterns for different fields of the data. Examples of different pattern types for there may be a desire to perform regular expression pattern matching include email patterns, uniform resource locator (URL) patterns, social security number (SSN) patterns, credit card number patterns, and others.

As shown in Figure 30, different fields of the data can be mapped to different regular expression pattern matching operations. For example, Fields 1, 3, and 4 of the data can be mapped to regular expression pattern matching that is configured to detect email patterns. Field 2 of the data can be mapped to regular expression pattern matching that is configured to detect URL patterns. Field 5 of the data can be mapped to regular expression pattern matching that is configured to detect some other pattern type (e.g., an SSN pattern).

In an exemplary embodiment, several different regular expression pattern matching modules can be instantiated in the hardware platform (e.g., reconfigurable logic such as an FPGA) for operation at the same time, whereby one of the regular expression pattern matching modules is configured to detect email patterns, another of the regular expression pattern matching modules is configured to detect URL patterns, and another of the regular expression pattern matching modules is configured to detect the other pattern.

However, in another exemplary embodiment, a single regular expression pattern matching module can be instantiated in the hardware platform, such as the regular expression pattern matching module described by the above-referenced and incorporated U.S. Pat. No. 7,702,629. The transition table memory that stores data to key the regular expression pattern matching module to search for a particular pattern can then be loaded with transition data for an email pattern, URL pattern, or another pattern on an as needed basis at run-time as different fields stream through.

Selective Enabling and Disabling of Engines and Processing Modules:

It should also be understood that command data can be inserted into the data stream to enable and disable various modules of the processing pipeline deployed by the translation engine(s) as appropriate for a processing task. For example, in an embodiment where both translation engine 202 and translation engine 400 are employed (for example in reconfigurable logic), and if the destination for the delimited data is a database, a practitioner
may choose to disable the translation engine 400. The disabled translation engine 400 would thus act as a pass through while remaining instantiated on the reconfigurable logic. As another example, if the incoming delimited data does not include shield characters, command data can be employed to disable the QM circuit of the VRG module and the QRM module. Such disabled modules would thus act as pass through components while remaining instantiated on the reconfigurable logic.

Figure 22 depicts an exemplary arrangement for a processing module to support a selective enabling/disabling functionality. The module 2200 of Figure 22 can include a command parser block, a logic block downstream from the command parser block, and a stream merge block downstream from the command parser block and the logic block.

The command parser block operates to receive the incoming data stream (which in this example is incoming data and associated SMCI control protocol; however, this need not be the case) and interpret the content of that stream to determine whether the incoming data is to be processed by the logic block or to bypass the logic block. Two criteria can determine whether data or commands will be processed by a module. For commands specifically, a module ID is present in a command to denote which specific module the command targets. There can be a special case for a module ID of zero that denotes the command applies to the entire chain. In addition to command routing, a context identifier can be used to denote which stream of data is currently being processed. Different modules can be bound to different contexts or streams.

Command messages are used to toggle the "plumbing" of a given module chain, turning modules ON or OFF (pass through) for a given context, and are used to mark changes in the active context. As a result, commands are sent through to set up the active data routes for a context and are used to denote which context is active. After the command setup, data will be processed by that configured chain until new commands arrive to enable / disable modules or toggle a context switch.

The command parser is responsible for inspecting command headers to note whether or not the command is intended for the given module, and it is responsible for following context switching commands that denote the active context.

When the module is in pass through, or is observing data from a context for which it is not bound, all data will be sent through the bypass channel 2202 rather than through the logic block. To disable an entire engine (such as translation engine 400), all of the modules that make up that engine can be disabled.
The logic block can implement any of the processing tasks described herein for the translation engine (e.g., the VRG module, the QM circuit, the V2F module, etc.).

The stream merge block operates to merge the output of the logic block and the information on the bypass channel to generate an output from the module. Data from the bypass channel will be given precedence over data from the logic block (if both are available), and the stream merge block is responsible for ensuring that data and commands are merged in on proper data boundaries.

The exemplary embodiments described herein can be used for a wide array of data processing tasks where performing data translations at low latency and high throughput are desired. Any enterprise in which data in a delimited format is widely used as the mode of communicating data records from location to location is expected to greatly benefit from use of the disclosed embodiments. For example, medical records and health care data are often communicated via a delimited data format and would benefit from improvements in how such data is processed (particularly in connection with data quality checking operations and database ETL operations).

While the present invention has been described above in relation to its exemplary embodiments, various modifications may be made thereto that still fall within the invention’s scope. Such modifications to the invention will be recognizable upon review of the teachings herein. Accordingly, the full scope of the present invention is to be defined by the appended claims and their legal equivalents.
WHAT I S CLAIMED IS:

1. A method comprising:
   a reconfigurable logic device receiving an incoming stream comprising a plurality of bytes arranged in a delimited data format, the incoming byte stream being representative of data arranged in a plurality of fields, the incoming byte stream comprising a plurality of data characters and a plurality of field delimiter characters, the field delimiter characters defining a plurality of boundaries between the fields;
   the reconfigurable logic device processing the received byte stream to identify the field delimiter characters that are present in the received byte stream; and
   the reconfigurable logic device translating the received byte stream to an outgoing byte stream arranged in a fixed field format based on the identified field delimiter characters, the outgoing byte stream comprising a plurality of the data characters of the received byte stream arranged in a plurality of fixed-size fields.

2. The method of claim 1 wherein the incoming byte stream further comprises a plurality of shield characters;
   wherein the processing step further comprises the reconfigurable logic device identifying the shield characters that are present in the received byte stream; and
   wherein the translating step further comprises the reconfigurable logic device translating the received byte stream to the outgoing byte stream having the fixed field format based on the identified field delimiter characters and the identified shield characters.

3. The method of claim 2 wherein the translating step comprises the reconfigurable logic device removing the identified field delimiter characters from the outgoing byte stream.

4. The method of claim 3 wherein the translating step further comprises the reconfigurable logic device removing the identified shield characters from the outgoing byte stream.

5. The method of claim 2 further comprising the reconfigurable logic device converting the received byte stream to an internal format tagged with associated control data that identifies the boundaries between the fields.
6. The method of claim 5 wherein the converting step further comprises the reconfigurable logic device generating a shield character mask associated with the received byte stream to identify the bytes in the received byte stream that are eligible for consideration as to whether they contain a field delimiter character.

7. The method of claim 6 wherein the converting step further comprises the reconfigurable logic device processing the bytes of the received byte stream and the generated shield character mask to generate field delimiter flag data associated with the received byte stream, the field delimiter flag data being indicative of whether an associated byte corresponds to a field delimiter character.

8. The method of claim 7 wherein the incoming byte stream is further representative of a plurality of records, at least one of the records comprising at least one of the fields, the incoming byte stream further comprising a plurality of record delimiter characters, the record delimiter characters defining a plurality of boundaries between the records, and wherein the converting step further comprises the reconfigurable logic device processing the bytes of the received byte stream and the generated shield character mask to generate record delimiter flag data associated with the received byte stream, the record delimiter flag data being indicative of whether an associated byte corresponds to a record delimiter character.

9. The method of claim 8 wherein the converting step further comprises the reconfigurable logic device identifying any empty fields that exist within the received byte stream based on the field delimiter flag data and the record delimiter flag data.

10. The method of claim 9 wherein the converting step further comprises the reconfigurable logic device removing the field delimiter characters and the record delimiter characters from the internally formatted byte stream based on the field delimiter flag data and the record delimiter flag data.

11. The method of claim 10 wherein the converting step further comprises the reconfigurable logic device generating control data associated with the internally formatted byte stream, the control data comprising (1) a start of field flag, (2) an end of field flag, (3) a start of record flag, (4) an end of record flag, and (5) a field identifier.
12. The method of any of claims 5-11 wherein the shield character identifying step further comprises the reconfigurable logic device performing a shield character removal operation on the bytes of the received byte stream.

13. The method of claim 12 wherein the shield character removal performing step comprises the reconfigurable logic device (1) distinguishing between the data characters that match the shield character and the shield characters, and (2) removing the identified shield characters.

14. The method of any of claims 5-13 further comprising the reconfigurable logic device generating the outgoing byte stream in the fixed field format from the internally formatted byte stream and the associated control data.

15. The method of claim 14 wherein the generating step further comprises the reconfigurable logic device filling a register corresponding to a fixed length field with the data characters of a field of the internally formatted byte stream based on the associated control data.

16. The method of claim 15 wherein the generating step further comprises the reconfigurable logic device filling the register with padding characters if there are not enough data characters of the field of the internally formatted byte stream to complete the fixed length field.

17. The method of any of claims 12-16 further comprising:
   - the reconfigurable logic device providing the outgoing byte stream to a data processing component for processing thereby; and
   - the data processing component selectively targeting a field of the outgoing byte stream for processing without analyzing the data characters of the outgoing byte stream.

18. The method of claim 17 further comprising:
   - the reconfigurable logic device receiving processed data representative of the outgoing byte stream from the data processing component; and
   - the reconfigurable logic device translating the processed data back to the delimited data format.
19. The method of claim 2 further comprising:
   the reconfigurable logic device converting the received byte stream to an internal
   format tagged with associated control data that identifies the boundaries between the fields;
   the reconfigurable logic device performing a shield character removal operation on
   the bytes of the received byte stream; and
   the reconfigurable logic device generating the outgoing byte stream in the fixed field
   format from the internally formatted byte stream and the associated control data; and
   wherein the reconfigurable logic device performs the converting step, the shield
   character removal performing step, and the generating step simultaneously with respect to
   each other in a pipelined fashion.

20. The method of any of claims 1-19 wherein the reconfigurable logic device performs
    the processing and translating steps for a plurality of characters in the byte stream per clock
    cycle.

21. The method of any of claims 1-20 wherein the delimited data format comprises a
    comma separated value (CSV) format.

22. An apparatus comprising:
    a reconfigurable logic device configured to (1) receive an incoming stream
    comprising a plurality of bytes arranged in a delimited data format, the incoming byte stream
    being representative of data arranged in a plurality of fields, the incoming byte stream
    comprising a plurality of data characters and a plurality of field delimiter characters, the field
    delimiter characters defining a plurality of boundaries between the fields, (2) process the
    received byte stream to identify the field delimiter characters that are present in the received
    byte stream, and (3) translate the received byte stream to an outgoing byte stream arranged in
    a fixed field format based on the identified field delimiter characters, the outgoing byte
    stream comprising a plurality of the data characters of the received byte stream arranged in a
    plurality of fixed-size fields.

23. An apparatus for data format translation, the apparatus comprising:
a plurality of processing modules arranged in a pipeline, the processing modules configured to operate simultaneously in a pipelined fashion, the pipeline configured to translate incoming data in a delimited data format to outgoing data in a fixed field format.

24. The apparatus of claim 23 wherein the pipeline comprises:
   a first module configured to convert an incoming stream of data in a delimited data format to an internal variable format having associated control data to identify records and fields in the data;
   a second module downstream from the first module, the second module configured to remove shield characters from the data in the internal variable format; and
   a third module downstream from the second module, the third module configured to convert the data in the variable format into the outgoing data in the fixed field format.

25. The apparatus of claim 24 wherein the first module is further configured to simultaneously test the same portion of the incoming data stream to determine whether the tested data stream portion comprises record delimiters or field delimiters.

26. The apparatus of any of claims 23-25 wherein the pipeline is deployed on a reconfigurable logic device.

27. The apparatus of any of claims 23-26 wherein the pipeline is further configured to ingest and process a plurality of characters of the incoming data per clock cycle.

28. An apparatus comprising:
   a data translation pipeline, the pipeline comprising (1) a first module configured to convert incoming data arranged in a delimited data format to an internal format, the incoming data in the delimited data format comprising a plurality of data characters, a plurality of field delimiter characters, a plurality of record delimiter characters, and a plurality of shield characters, the converted data having the internal format being stripped of field delimiter characters and record delimiter characters while preserving data characters of incoming fields, (2) a second module downstream from the first module, the second module configured to remove shield characters from the converted data having the internal format, and (3) a third module downstream from the second module, the third module configured to translate the output of the second module to outgoing data having a fixed field format.
29. The apparatus of claim 28 wherein the first module is further configured to simultaneously test the same characters of the incoming data to determine whether the tested characters are record delimiter characters or field delimiter characters.

30. An apparatus comprising:
   a variable record gate module configured to convert an incoming stream of data in a delimited data format to an internal variable format having associated control data to identify records and fields in the data.

31. An apparatus comprising:
   a shield character masker module configured to mask fields of an incoming data stream that are wrapped by shield characters.

32. The apparatus of claim 31 further comprising:
   a delimiter finder module downstream from the shield character module, the delimiter finder module configured to detect delimiter characters in the incoming data stream based on a mask generated by the shield character masker module.

33. The apparatus of claim 32 wherein the delimiter characters are field delimiter characters.

34. The apparatus of claim 32 wherein the delimiter characters are record delimiter characters.

35. An apparatus comprising:
   a delimiter finder module configured to detect delimiter characters in the incoming data stream.

36. The apparatus of claim 35 wherein the delimiter characters are field delimiter characters.

37. The apparatus of claim 35 wherein the delimiter characters are record delimiter characters.
38. The apparatus of claim 35 wherein the delimiter characters comprise field delimiter characters and record delimiter characters; and wherein the delimiter finder module is further configured to simultaneously test whether the same portion of the incoming data stream includes field delimiter characters or record delimiter characters.

39. An apparatus comprising:
   a module configured to convert data in a variable format with associated control data into outgoing data having a fixed field format.

40. An apparatus comprising:
   a translation engine configured to translate incoming data in a fixed field format to outgoing data in a delimited data format.

41. A method comprising:
   a reconfigurable logic device receiving an incoming stream comprising a plurality of bytes arranged in a delimited data format, the incoming byte stream being representative of data arranged in a plurality of fields, the incoming byte stream comprising a plurality of data characters and a plurality of field delimiter characters, the field delimiter characters defining a plurality of boundaries between the fields;
   the reconfigurable logic device processing the received byte stream to identify the field delimiter characters that are present in the received byte stream; and
   the reconfigurable logic device translating the received byte stream to an outgoing byte stream based on the identified field delimiter characters, the outgoing byte stream arranged in a structured format and being representative of the data in the fields of the received byte stream, the outgoing byte stream comprising a plurality of the data characters of the received byte stream, the structured format being configured to permit a downstream processing component to jump from field to field in the outgoing byte stream without analyzing the data characters of the outgoing byte stream.

42. The method of claim 41 wherein the incoming byte stream is further representative of a plurality of records, at least one of the records comprising at least one of the fields, the
incoming byte stream further comprising a plurality of record delimiter characters, the record delimiter characters defining a plurality of boundaries between the records;

wherein the processing step further comprises the reconfigurable logic device identifying the record delimiter characters that are present in the received byte stream; and

wherein the translating step further comprises the reconfigurable logic device translating the received byte stream to the outgoing byte stream having the structured format based on the identified field delimiter characters and the identified record delimiter characters.

43. The method of claim 41 wherein the structured format is further configured to permit the downstream processing component to jump from record to record in the outgoing byte stream without analyzing the data characters of the outgoing byte stream.

44. The method of any of claims 42-43 wherein the translating step further comprises the reconfigurable logic device removing the identified record delimiter characters from the outgoing byte stream.

45. The method of any of claims 41-44 wherein the incoming byte stream further comprises a plurality of shield characters;

wherein the processing step further comprises the reconfigurable logic device identifying the shield characters that are present in the received byte stream; and

wherein the translating step further comprises the reconfigurable logic device translating the received byte stream to the outgoing byte stream having the structured format based on the identified field delimiter characters and the identified shield characters.

46. The method of claim 45 wherein the translating step further comprises the reconfigurable logic device removing the identified shield characters from the outgoing byte stream.

47. The method of any of claims 41-46 wherein the translating step further comprises removing the identified field delimiter characters from the outgoing byte stream.
48. The method of any of claims 41-47 further comprising the reconfigurable logic device providing the outgoing byte stream to the downstream processing component for processing thereby.

49. The method of claim 48 further comprising the downstream processing component performing a plurality of processing operations on the outgoing byte stream to generate processed data from the outgoing byte stream.

50. The method of claim 49 wherein the processing operations include a plurality of extract, transfer, and load (ETL) database operations.

51. The method of claim 49 wherein the processing operations comprise a plurality of data validation operations.

52. The method of any of claims 41-51 further comprising the reconfigurable logic device translating the processed data back to the delimited data format of the received byte stream.

53. The method of any of claims 41-52 wherein the downstream processing component is implemented on the reconfigurable logic device.

54. The method of any of claims 41-52 wherein the downstream processing component is implemented in software on a processor.

55. The method of any of claims 41-54 wherein the delimited data format comprises a comma separated value (CSV) format.

56. The method of any of claims 41-55 wherein the structured format comprises a fixed field format.

57. The method of any of claims 41-55 wherein the structured format comprises a mapped variable field format.
58. The method of any of claims 41-57 wherein the reconfigurable logic device performs the processing and translating steps for a plurality of characters in the byte stream per clock cycle.

59. A method comprising:
   receiving data in a delimited data format;
   converting the received data to a structured format; and
   performing a plurality of processing operations on the converted data to generate processed data in the structured format; and
   loading the processed data into a database; and
   wherein the converting step is performed by a reconfigurable logic device.

60. The method of claim 59 wherein the structured format comprises a fixed field format.

61. The method of claim 60 wherein the converted data comprises a plurality of data fields, each data field having a known field length, wherein the processing operations comprise a plurality of field-specific data processing operations, and wherein the performing step comprises targeting a specific field of the converted data for a field-specific processing operation without analyzing the data content of the data fields.

62. The method of claim 59 wherein the structured format comprises a mapped variable field format.

63. The method of claim 62 wherein the converted data includes header data indicative of where boundaries exist between a plurality of records in the converted data and where boundaries exist between a plurality of fields in the converted data.

64. The method of claim 63 wherein the converted data comprises a plurality of data fields, the data fields having a variable lengths, wherein the processing operations comprise a plurality of field-specific data processing operations, and wherein the performing step comprises targeting a specific field of the converted data for a field-specific processing operation based on the header data without analyzing the data content of the data fields.
65. The method of any of claims 59-64 wherein the data processing operations comprise data quality checking operations as part of an extract, transfer, load (ETL) procedure.

66. The method of any of claims 59-65 wherein the at least one of the processing operations is performed by software executed by a processor.

67. The method of any of claims 59-66 wherein the converting step comprises converting a plurality of characters of the received data to the fixed field format per clock cycle.

68. The method of any of claims 59-67 wherein the at least one of the processing operations is performed by a reconfigurable logic device.

69. A method comprising:

   a reconfigurable logic device receiving an incoming stream comprising a plurality of bytes arranged in a delimited data format, the incoming byte stream being representative of data arranged in a plurality of fields, the incoming byte stream comprising a plurality of data characters and a plurality of field delimiter characters, the field delimiter characters defining a plurality of boundaries between the fields;

   the reconfigurable logic device processing the received byte stream to identify the field delimiter characters that are present in the received byte stream; and

   the reconfigurable logic device translating the received byte stream to an outgoing byte stream arranged in a mapped variable field format based on the identified field delimiter characters, the outgoing byte stream comprising a plurality of the data characters of the received byte stream arranged in a plurality of variable-size fields.

70. The method of claim 69 wherein the incoming byte stream further comprises a plurality of shield characters;

   wherein the processing step further comprises the reconfigurable logic device identifying the shield characters that are present in the received byte stream; and

   wherein the translating step further comprises the reconfigurable logic device translating the received byte stream to the outgoing byte stream having the mapped variable field format based on the identified field delimiter characters and the identified shield characters.
71. The method of claim 70 wherein the translating step comprises the reconfigurable logic device removing the identified field delimiter characters from the outgoing byte stream.

72. The method of claim 71 wherein the translating step further comprises the reconfigurable logic device removing the identified shield characters from the outgoing byte stream.

73. The method of claim 70 further comprising the reconfigurable logic device converting the received byte stream to an internal format tagged with associated control data that identifies the boundaries between the fields.

74. The method of claim 73 wherein the converting step further comprises the reconfigurable logic device generating a shield character mask associated with the received byte stream to identify the bytes in the received byte stream that are eligible for consideration as to whether they contain a field delimiter character.

75. The method of claim 74 wherein the converting step further comprises the reconfigurable logic device processing the bytes of the received byte stream and the generated shield character mask to generate field delimiter flag data associated with the received byte stream, the field delimiter flag data being indicative of whether an associated byte corresponds to a field delimiter character.

76. The method of claim 75 wherein the incoming byte stream is further representative of a plurality of records, at least one of the records comprising at least one of the fields, the incoming byte stream further comprising a plurality of record delimiter characters, the record delimiter characters defining a plurality of boundaries between the records, and wherein the converting step further comprises the reconfigurable logic device processing the bytes of the received byte stream and the generated shield character mask to generate record delimiter flag data associated with the received byte stream, the record delimiter flag data being indicative of whether an associated byte corresponds to a record delimiter character.

77. The method of claim 76 wherein the converting step further comprises the reconfigurable logic device identifying any empty fields that exist within the received byte stream based on the field delimiter flag data and the record delimiter flag data.
78. The method of claim 77 wherein the converting step further comprises the reconfigurable logic device removing the field delimiter characters and the record delimiter characters from the internally formatted byte stream based on the field delimiter flag data and the record delimiter flag data.

79. The method of claim 78 wherein the converting step further comprises the reconfigurable logic device generating control data associated with the internally formatted byte stream, the control data comprising (1) a start of field flag, (2) an end of field flag, (3) a start of record flag, (4) an end of record flag, and (5) a field identifier.

80. The method of any of claims 73-79 wherein the shield character identifying step further comprises the reconfigurable logic device performing a shield character removal operation on the bytes of the received byte stream.

81. The method of claim 80 wherein the shield character removal performing step comprises the reconfigurable logic device (1) distinguishing between the data characters that match the shield character and the shield characters, and (2) removing the identified shield characters.

82. The method of any of claims 73-81 further comprising the reconfigurable logic device generating the outgoing byte stream in the mapped variable field format from the internally formatted byte stream and the associated control data.

83. The method of claim 82 wherein the generating step further comprises the reconfigurable logic device determining byte lengths for the fields that are present in the internally formatted data based on the associated control data and generating field header data for the outgoing byte stream indicative of the determined byte lengths for the fields.

84. The method of claim 83 wherein the field header data generating step comprises the reconfigurable logic device computing an array of byte offset values indicative of boundaries for a plurality of fields of a record in the outgoing byte stream.

85. The method of any of claims 80-84 further comprising:
the reconfigurable logic device providing the outgoing byte stream to a data processing component for processing thereby; and
the data processing component selectively targeting a field of the outgoing byte stream for processing without analyzing the data characters of the outgoing byte stream.

86. The method of claim 85 wherein the outgoing byte stream includes a plurality of record headers and a plurality of field headers, the record headers comprising data indicative of where boundaries exist between a plurality of records in the outgoing byte stream, the field headers comprising data indicative of where boundaries exist between a plurality of fields in the records, and wherein the selectively targeting step comprises the data processing component selectively targeting the field based on the data in the field headers.

87. The method of any of claims 85-86 further comprising:
the reconfigurable logic device receiving processed data representative of the outgoing byte stream from the data processing component; and
the reconfigurable logic device translating the processed data back to the delimited data format.

88. The method of claim 70 further comprising:
the reconfigurable logic device converting the received byte stream to an internal format tagged with associated control data that identifies the boundaries between the fields;
the reconfigurable logic device performing a shield character removal operation on the bytes of the received byte stream; and
the reconfigurable logic device generating the outgoing byte stream in the mapped variable field format from the internally formatted byte stream and the associated control data; and
wherein the reconfigurable logic device performs the converting step, the shield character removal performing step, and the generating step simultaneously with respect to each other in a pipelined fashion.

89. The method of any of claims 69-88 wherein the reconfigurable logic device performs the processing and translating steps for a plurality of characters in the byte stream per clock cycle.
90. The method of any of claims 69-89 wherein the outgoing byte stream includes a plurality of record headers and a plurality of field headers, the record headers comprising data indicative of where boundaries exist between a plurality of records in the outgoing byte stream, the field headers comprising data indicative of where boundaries exist between a plurality of fields in the records.

91. The method of any of claims 69-90 wherein the delimited data format comprises a comma separated value (CSV) format.

92. An apparatus comprising:
   a reconfigurable logic device configured to (1) receive an incoming stream comprising a plurality of bytes arranged in a delimited data format, the incoming byte stream being representative of data arranged in a plurality of fields, the incoming byte stream comprising a plurality of data characters and a plurality of field delimiter characters, the field delimiter characters defining a plurality of boundaries between the fields, (2) process the received byte stream to identify the field delimiter characters that are present in the received byte stream, and (3) translate the received byte stream to an outgoing byte stream arranged in a mapped variable field format based on the identified field delimiter characters, the outgoing byte stream comprising a plurality of the data characters of the received byte stream arranged in a plurality of variable-size fields.

93. An apparatus for data format translation, the apparatus comprising:
   a plurality of processing modules arranged in a pipeline, the processing modules configured to operate simultaneously in a pipelined fashion, the pipeline configured to translate incoming data in a delimited data format to outgoing data in a mapped variable field format.

94. The apparatus of claim 93 wherein the pipeline comprises:
   a first module configured to convert an incoming stream of data in a delimited data format to an internal variable format having associated control data to identify records and fields in the data;
   a second module downstream from the first module, the second module configured to remove shield characters from the data in the internal variable format; and
a third module downstream from the second module, the third module configured to convert the data in the variable format into the outgoing data in the mapped variable field format.

95. The apparatus of claim 94 wherein the first module is further configured to simultaneously test the same portion of the incoming data stream to determine whether the tested data stream portion comprises record delimiters or field delimiters.

96. The apparatus of any of claims 93-95 wherein the pipeline is deployed on a reconfigurable logic device.

97. The apparatus of any of claims 93-96 wherein the pipeline is further configured to ingest and process a plurality of characters of the incoming data per clock cycle.

98. An apparatus comprising:

a data translation pipeline, the pipeline comprising (1) a first module configured to convert incoming data arranged in a delimited data format to an internal format, the incoming data in the delimited data format comprising a plurality of data characters, a plurality of field delimiter characters, a plurality of record delimiter characters, and a plurality of shield characters, the converted data having the internal format being stripped of field delimiter characters and record delimiter characters while preserving data characters of incoming fields, (2) a second module downstream from the first module, the second module configured to remove shield characters from the converted data having the internal format, and (3) a third module downstream from the second module, the third module configured to translate the output of the second module to outgoing data having a mapped variable field format.

99. The apparatus of claim 98 wherein the first module is further configured to simultaneously test the same characters of the incoming data to determine whether the tested characters are record delimiter characters or field delimiter characters.

100. An apparatus comprising:

a module configured to convert data in a variable format with associated control data into outgoing data having a mapped variable field format.
101. An apparatus comprising:
   a translation engine configured to translate incoming data in a mapped variable field format to outgoing data in a delimited data format.

102. An apparatus comprising:
   a data translation pipeline, the pipeline comprising (1) a first module configured to convert incoming data arranged in a delimited data format to an internal format, the incoming data in the delimited data format comprising a plurality of data characters, a plurality of field delimiter characters, a plurality of record delimiter characters, and a plurality of shield characters, the converted data having the internal format being stripped of field delimiter characters and record delimiter characters while preserving data characters of incoming fields, and (2) a second module downstream from the first module, the second module configured to remove shield characters from the converted data having the internal format;
      a hardware-accelerated data processing stage configured to perform a data processing operation on output from the second module to thereby generate processed data.

103. The apparatus of claim 102 wherein the first module is further configured to simultaneously test the same characters of the incoming data to determine whether the tested characters are record delimiter characters or field delimiter characters.

104. A method comprising:
   converting incoming data arranged in a delimited data format to an internal format, the incoming data in the delimited data format comprising a plurality of data characters, a plurality of field delimiter characters, a plurality of record delimiter characters, and a plurality of shield characters, the converted data having the internal format being stripped of field delimiter characters and record delimiter characters while preserving data characters of incoming fields;
   removing shield characters from the converted data;
   performing at least one hardware-accelerated processing operation on at least a portion of the converted data to generate processed data;
   loading the processed data into a database; and
   wherein the converting step is performed by a reconfigurable logic device.
105. The method of claim 104 wherein the converted data includes associated control data indicative of where boundaries exist between a plurality of records in the converted data and where boundaries exist between a plurality of fields in the converted data.

106. The method of claim 105 wherein the at least one hardware-accelerated data processing operation comprises a plurality of field-specific hardware-accelerated data processing operations, and wherein the performing step comprises targeting a specific field of the converted data for a field-specific hardware-accelerated processing operation based on the associated control data without analyzing the data content of the data fields.

107. The method of any of claims 104-106 wherein the data processing operations comprise data quality checking operations as part of an extract, transfer, load (ETL) procedure.

108. The method of any of claims 104-107 further comprising:

   converting the shield-removed converted data into data in a fixed field format; and
   performing, by software executed by a processor, at least one processing operation on at least a portion of the data in the fixed field format to generate additional processed data; and
   loading the additional processed data into the database.

109. The method of any of claims 104-107 further comprising:

   converting the shield-removed converted data into data in a mapped variable field format; and
   performing, by software executed by a processor, at least one processing operation on at least a portion of the data in the mapped variable field format to generate additional processed data; and
   loading the additional processed data into the database.

110. The method of any of claims 104-109 wherein the converting step comprises converting a plurality of characters of the received data to the internal format per clock cycle.

111. A method comprising:
receiving data in a delimited data format, the received data comprising a plurality of data fields;

converting, by hardware logic, the received data into a structured data format, the structured format including header data indicative of where boundaries exist within the converted data between the data fields;

identifying and locating a data field of the converted data based on the header data without analyzing content of the data fields; and

performing a processing operation on the identified and located data field.
Name: Doe, James ("Jim")
Street Address: 123 Main Street
City/State: St. Louis, MO
Zip Code: 63101
Telephone Number: 314-555-5555
SSN: 555-55-5555
Treatment Date: 8/4/2012
Treatment Code: JR4CL

Figure 1
Figure 6
<table>
<thead>
<tr>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
<th>Field 6</th>
<th>Field 7</th>
<th>Field 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doe, James (&quot;Jim&quot;)</td>
<td>123 Main Street</td>
<td>St. Louis, MO</td>
<td>63101</td>
<td>314555 - 5555</td>
<td>8/4/2012</td>
<td>J R 4 C L</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 7*
Figure 8(a)
Figure 8(b)
Figure 12
Figure 13

- **Close Quote**
  - *Initial State*
  - DV = 1

- **Open Quote**
  - DV = 0

  - DV = 0: Delimiter Valid Off
  - Not a candidate to be a delimiter

  - DV = 1: Delimiter Valid On
  - Character is a valid candidate to be a delimiter

- **Non-Quote**
- **Quote**
Figure 14(b)
Figure 15

Field ID Logic

Data count
EOR
EOF
field ID

Register Outputs

Skip

SR Removal Logic

find next delimiter
fields skip
data count

SR

Record Delimiter

SR Count

Field Delimiter

Data

Shift
Figure 17(a)
Figure 19
QAD: Quote Addition Module
logic (smci in, smci out)

Figure 20
Figure 21

VIRG Module
(Variable to CSV)

- Data width + max delimiter width
  *Add RDL and FDL characters based on smci signals (EOR, EOF, etc.)

- SR(0)
  SR(1) (3 x SR(0))
  *staging and packing for output
- If command was:
  - not sent to me;
  - has error;
  - "return to software" set, OR
  - is global
  -> send incoming to bypass channel

- If data is not bound to my context,
  it is sent on the bypass channel

- Otherwise, data sent to processing logic

Figure 22
Figure 23

Figure 24
struct RecordEntry{
    xuint16 record_size;
    xuint16 num_fields;
    xuint16 *offset_array;
    char *data_pointer;
};

// Method to read out a given field
char *GetField(RecordEntry *record, int field_id)
{
    char *myfield;  // location to store field
    int start_offset = offset_array[field_id];
    int end_offset = record_size;  // Assume end of record
    if (num_fields > field_id) // ...unless there's another field after ours
    {
        end_offset = num_fields[field_id+1];
    }
    int field_len = end_offset - start_offset;
    myfield = new char[field_len];
    memcpy(myfield, &data_pointer[offset_array[field_id]], field_len);
    return myfield;
}
Figure 29

Email RegEx Pattern Matching (3000)
URL RegEx Pattern Matching (3002)
Other RegEx Pattern Matching (3004)

Field 1 (3102) Field 2 (3104) Field 3 (3106) Field 4 (3108) Field 5 (3110)

Figure 30