SDNet Packet Processor

User Guide

UG1012 (v2017.1) June 15, 2017

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/15/2017	2017.1	Added notes to move_to_section and increment_offset under Section Subclass. Changed “sizeof()" to "sizeof(SectionName)" in code example under increment_offset. Clarified first paragraph under Tuple Subclass. Changed first instance of "sizeof()" to "sizeof(ETH)" and second instance of "sizeof()" to "sizeof(VLAN)" in code example under A Simple Parsing Engine. Changed "sizeof()" to "sizeof(ETH)" in code example under remove Method. Changed "sizeof()" with "sizeof(SectionName)" under Procedure for Writing SDNet Functional Specifications. Changed first instance of "sizeof()" to "sizeof(ETH)", second instance of "sizeof()" to "sizeof(VLAN)", third instance of "sizeof()" to "sizeof(IPv6)" and fourth instance of "sizeof()" to "sizeof(UDP)" in code example under Section Numbering. Added "named Packet.user" to first sentence under Input Packet File (Optional) and "named Tuple.user" to first sentence under Input Tuple File (Optional).
05/11/2017	2017.1	Initial release.

Table of Contents

Introduction

Overview

The Xilinx® SDNet™ data plane builder generates systems that can be programmed for a wide range of packet processing functions, from simple packet classification to complex packet editing. Systems are described in the SDNet functional specifications. The functional specifications are compiled to generate highly efficient and scalable hardware implementations.

SDNet can also implement designs that are described in the P4 language by first using the P4-SDNet Translator, see the *P4-SDNet Translator User Guide* (UG1252) [Ref 1]. The P4-SDNet Translator produces a new SDNet functional specification based on the P4-specified processing.

SDNet features the following:

- Construction of hierarchical SDNet systems, consisting of a larger variety of different types of engines including: parsing, editing, lookup, tuple, and user engines.
  - *Parsing engines* extract header information or data from packets.
  - *Editing engines* manipulate the contents of packets by inserting, modifying, or removing packet data.
  - *Lookup engines* instantiate IP cores generated from a library of basic types for packet processing including: exact match, longest-prefix match, ternary match, and RAM.
  - *Tuple engines* are designated for manipulating tuples/metadata that might be determined from packets or data originating externally or from some other engine.
  - *User engines* provide a mechanism for incorporating the user's custom IP, provided that it conforms to the SDNet engine interface and also optionally provides a C++ simulation model.
  - *Systems* can be arranged hierarchically to implement larger and more complex packet processing functions.
- Support for three different clock domains so that engines can run at one of the following frequencies:
  - *Line rate* of the packet data bus used typically for engines reading or modifying packets

    - *Packet rate* used for functions that occur once per packet such as an individual lookup
    - *Control rate* which is the speed of the memory-mapped control interface for controlling and configuring engines.
- Systems result in high performance hardware implementations, achieving a range of 10-100 Gb/s.
- System backpressure capability is automatically generated including the insertion of buffers providing dataflow synchronization for engines. This capability enables backpressure of the packet bus for momentarily pausing packet processing.
- A C++ model is generated for high level system simulation of the specification prior to running RTL simulation.
- SDNet supports either LBUS or AXI-Stream signaling protocols for packet interfaces.

This document describes the standard design flow beginning with the user's packet processing requirements to implementing high performance packet processing systems in a Xilinx FPGA. This document is intended for novice users and does not include all of the detailed features supported by SDNet. For a complete description of the SDNet functional specification, see the *SDNet Functional Specification User Guide* (UG1016) [Ref 2].

SDNet Capabilities

The next two sections provide some simple and more complex examples to illustrate how SDNet can be used and to display the expressive capabilities of SDNet. See Dataflow Graph Visualizations, page 57 for more information on dataflow graphs, including connection and engine color coding descriptions.

Simple Examples

The first few examples, shown in Figure 1-1 through Figure 1-4, have a simple dataflow topology. The engines are depicted as nodes and the edges represent the dataflow between engines. Even though these graphs appear simple, the engines perform a fair amount of processing.

Figure 1-1 illustrates a trivial topology of an OpenFlow classifier consisting of a parser, with one extracted tuple, and a lookup engine. The lookup engine is of type TCAM in this example. The source for getting started with this example is provided in Appendix E, SDNet Example System.

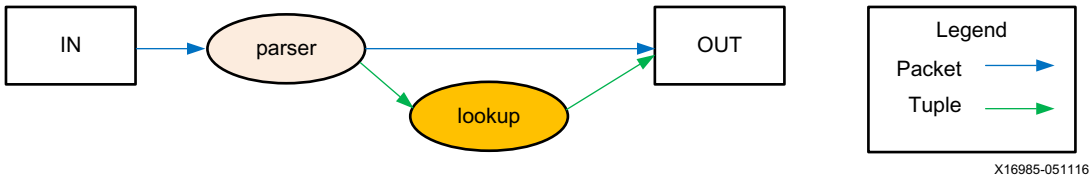


Figure 1-1: **Trivial Topology of an Openflow Classifier**

Figure 1-2 illustrates a trivial hierarchical topology of an MPLS label switched router consisting of a classifier subsystem and an editor. A subsystem is an SDNet system instantiated inside of a parent SDNet system.

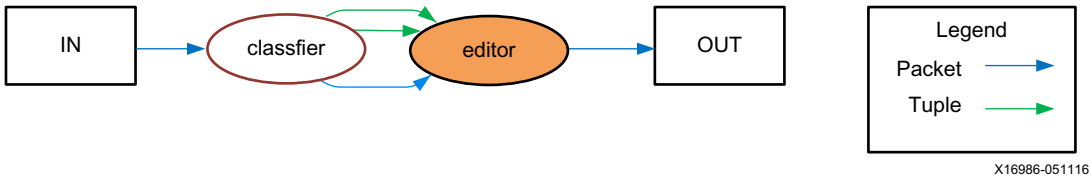


Figure 1-2: **Trivial Hierarchical Topology of an MPLS Label**

Figure 1-3 illustrates the classifier subsystem implementation from the system shown in Figure 1-2. This classifier subsystem consists of a parser, with two extracted tuples, and a lookup engine.

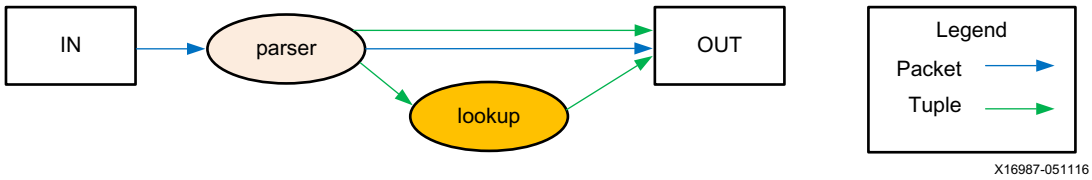


Figure 1-3: **Classifier Subsystem Implementation**

Figure 1-4 illustrates a different router example that utilizes two different types of lookup engines: exact match and longest prefix match.

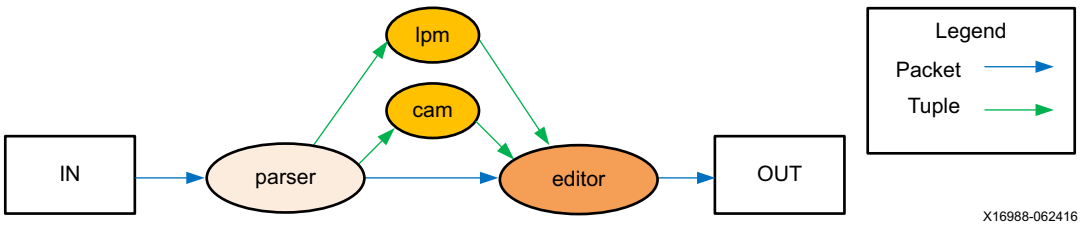


Figure 1-4: **Router Example**

More Complex Examples

Figure 1-5 illustrates an example containing user engines performing the encryption path of an IPSec implementation. The system consists of a parser, two editors, three lookup engines, and three user engines. The user engines in this example implement other functions such as counters and also the encryption function.

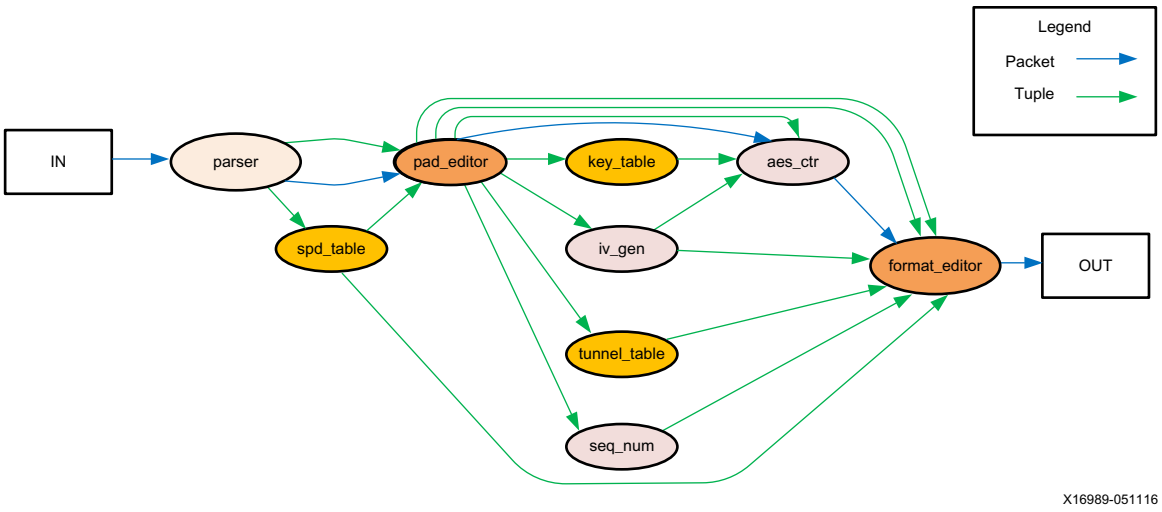


Figure 1-5: **User Engines Performing the Encryption Path of an IPSec Implementation**

Figure 1-6 illustrates another example showing the decryption path of an IPSec implementation.

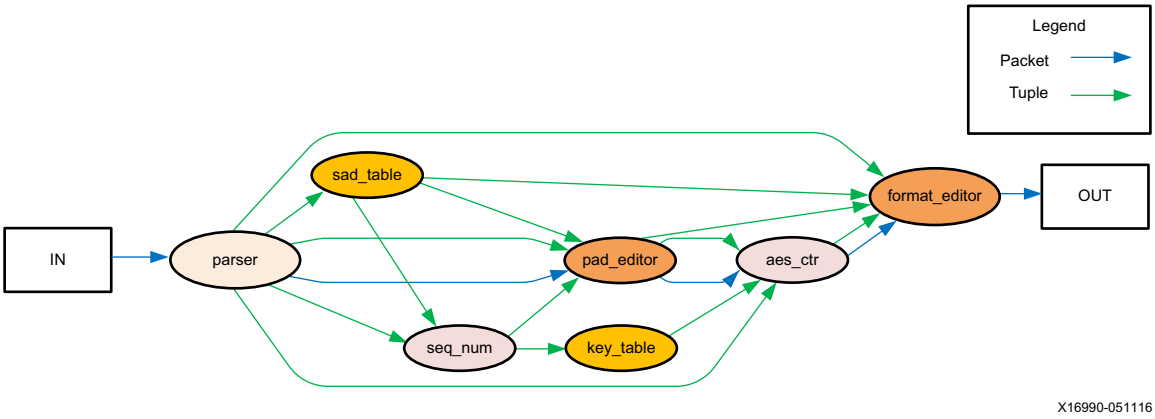


Figure 1-6: **Decryption Path of an IPSec Implementation**

IPSec implementation shown in Figure 1-5 and Figure 1-6 illustrate a more complex type of system that can be built using SDNet and its framework. The AES encrypt and decrypt functions are brought into the SDNet world as user engines and the IP is designed to conform to the SDNet signaling interfaces. A recent translator is available from P4 to SDNet. An example system is shown in Figure 1-7.

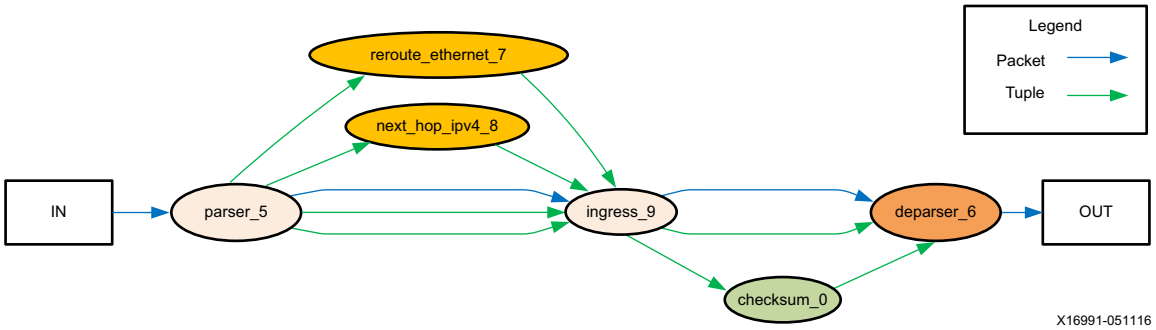


Figure 1-7: **Implement System Coded for P4 Using SDNet**

Figure 1-8 shows a more extreme example of a complex OpenFlow-based system, illustrating expressive capability for the system topology.

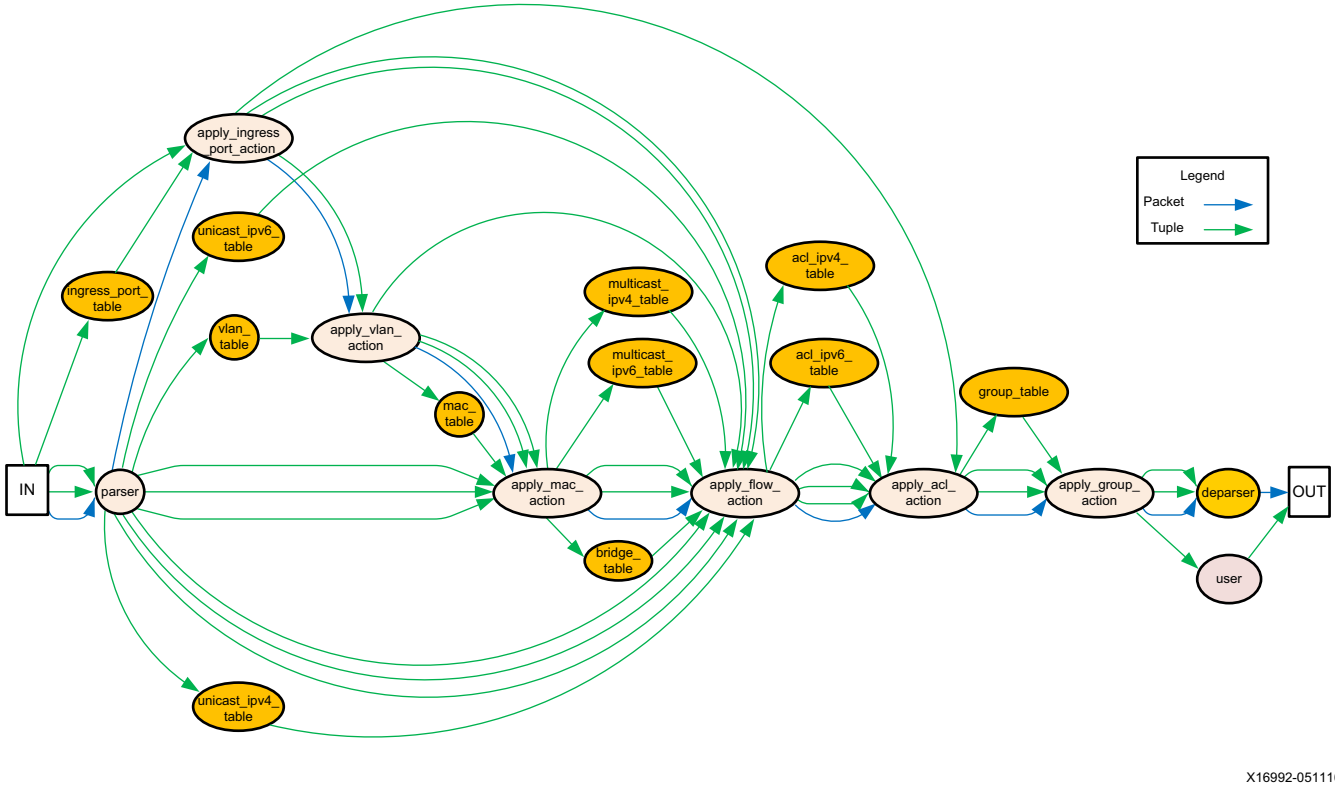


Figure 1-8: **Complex Openflow-based System**

The complex OpenFlow-based system shown in Figure 1-8 illustrates a larger SDNet system for OpenFlow-related processing containing 12 lookup engines, 6 parsers, 1 editor, and 1 user engine. It is possible to implement relatively complex systems using SDNet.

Packet Processing Requirements and Functionality

The packet processing requirements are provided by the user and must include:

- The packet formats to be parsed and/or edited
- The bit fields or subfields to be extracted (parser), or inserted, deleted, or modified (editor)
- The output key format (result of parsing/classification)

How to Use this User Guide

The SDNet framework design flow is shown in Figure 1-9. This rest of this user guide is organized accordingly, as follows.

- Chapter 2, SDNet Dataflow Model describes the basic SDNet dataflow model, from which the specifications are based.
- Chapter 3, SDNet Functional Specifications describes the textual functional specification elements programmed by the user.
- Chapter 4, Compilation describes how to run the SDNet compiler and the files it produces.
- Chapter 5, Simulation describes the two simulation types: high level (C++) model, and RTL model.
- Chapter 6, Importing the Design into Vivado Tools describes how to import the design into the Vivado® Design Suite and incorporate the system into a larger FPGA design.

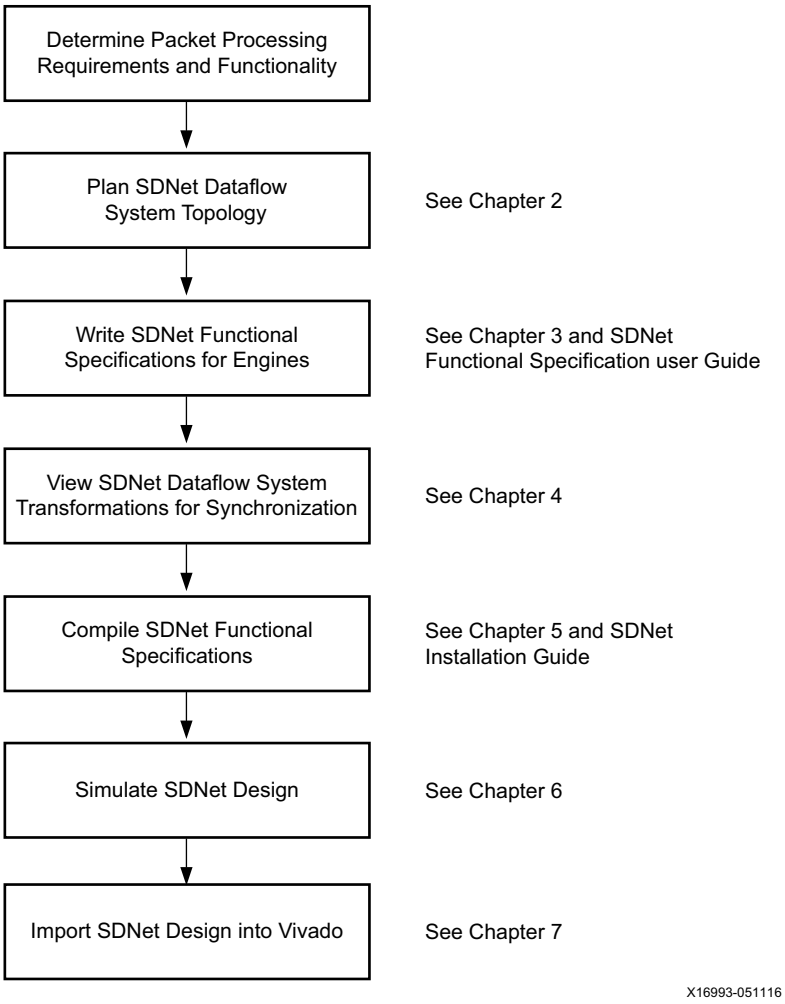


Figure 1-9: **SDNet Framework Design Flow**

SDNet Dataflow Model

SDNet is based on a modular design methodology, consisting of connecting a variety of different types of engines. The dataflow graph is described as a textual description. These engines primarily communicate with the dataflow of packets and tuples to implement larger system behavior. The execution model is reactive and based on a synchronous dataflow model that triggers an engine when all of its inputs arrive. The inputs might be, for example, packets and corresponding metadata communicated as tuples. Ports such as packets and tuples are defined in the following section followed by a description of the different types engines, and how to put these together to build a system.

Ports

Packets

Packet ports are the primary SDNet interface responsible for moving around packets between engines and also to the external world. Currently, engines can only contain up to a single input packet port and a single output packet port. Parsing engines, editing engines, and systems require packet ports. User engines have the option of Packet Ports. All other engines do not have packet ports.

Tuples

Tuple ports are the secondary SDNet interface responsible for communicating packet-related metadata between engines and also to the external world. Tuples can only correspond to a single packet and are processed at the rate of one tuple per packet. Engines can contain multiple tuple ports, each communicating different metadata.

Access

Access ports in SDNet are currently memory-mapped control ports, which are connected behind the scenes by the compiler. These are illustrated in the dataflow model, however, SDNet specifications do not explicitly instantiate or connect these ports. They are connected automatically by the compiler.

Plain

Plain ports in SDNet are used for RTL communication between user engines not captured by the other three types of ports. Connections between plain ports essentially represents un-typed plain wire connections.

Engines

Parsing Engine

Parsing engines (parsers) are used to read and decode packet headers and extract the required information for classification or for later packet modification. Parsers can only read from packets and cannot modify them. They can perform computations on data extracted from packets and transmit the data as output tuples. Parsers have a map construct for including mini tables useful for decoding packets or representing output actions.

Editing Engine

Editing engines (editors) are used for manipulating packets. They cannot read directly from the packet data bus but they can write to the packet datapath to insert, replace, or remove data from packets. Editors typically have multiple input tuples containing data to be written into packets.

Tuple Engine

Tuple engines are mainly for manipulating tuples and performing computations on tuple data. Tuple engines share the same computation capabilities as parsers, but without having any packet ports or packet datapath.

Lookup Engine

IMPORTANT: *Design trade-offs between table dimensions and area can cause tables with very wide keys and large sizes to miss target timing.*

Lookup engines are used to implement a search over a variety of different kinds of tables. SDNet includes a small library of different lookup engine types, which can be one of four types: *EM* (exact match), *LPM* (longest prefix match), *TCAM* (ternary content addressable memory), or *DIRECT* address lookup (RAM). Each of these tables takes a search key as a request tuple and returns a response tuple containing an associated value stored in memory. These tables are described as follows.

EM

EM (exact match) is a basic binary match of a search key to one of the keys stored within the table. The associated value for the key is returned within the response as well as an indication of whether the key produced a match.

Key: $12 < k < 384$
Value: $1 < v < 256$
Depth: $256 < d < 2^{19}$
Hit/Miss indication: Yes

LPM

LPM (longest prefix match) is a form of a ranged match comparing the most significant bits of the key to a table of prefixes in a table. The longest matching prefix has priority over any other matches. The associated value for the longest prefix is returned within the response as well as an indication of whether the key produced a prefix match. For efficient memory utilization the depth of the LPM tables should be a power of two minus one, for reasons related to its implementation.

Key: $8 < k < 512$
Value: $1 < v < 512$
Depth: $7 < d < 2^{16}-1$
Hit/Miss indication: Yes

TCAM

TCAM (ternary content addressable memory) is used for making a pattern match between the search key and a masked key within a table. The per-entry mask represents a wildcard pattern of bits used when performing the comparison. The entries of the table are ordered by priority. The stored associated value for the matched entry with highest priority is returned as well as an indication of a match. The TCAM is actually an algorithmic TCAM that emulates TCAM behavior and maps entries onto distributed LUTRAM.

Key: $1 < k < 800$
Value: $1 < v < 400$
Depth: $1 < d < 4{,}096$
Hit/Miss indication: Yes

DIRECT

DIRECT address match, unlike the others, does not involve a comparison but instead uses the key as the direct address to access the stored value at that location (the table is effectively a RAM). The depth of the table should be equal to 2k.

Key: $1 < k < 16$
Value: $1 < v < 256$
Depth: $7 < d < 2^{16}-1$
Hit/Miss indication: No

User Engine

The user engine class allows users to import custom IP cores into the SDNet specification to take advantage of the SDNet framework for data plane building and system simulation capabilities. In SDNet, user engines specify the interface of the engine but not its behavior. User engines must conform to the synchronous dataflow behavior of the SDNet dataflow model. The user engine is expected to be implemented by the user in VHDL or Verilog with an optional corresponding C behavioral model for high level simulation. User engines can have packet, tuple, and plain ports.

System

In the SDNet model, systems are treated as a type of engine. Systems connect engines together based on their matching port types. Systems can only contain precisely one packet input port and one packet output port. Tuple connections are allowed to fan out but not fan in to engines. Systems can also have plain ports for connections to subsystems or user engines. Subsystems are SDNet systems that are instantiated within a parent SDNet system.

Elaborated Dataflow Model

The following infrastructure/glue logic blocks are added to the elaborated system connection topology during compilation to support the synchronous dataflow, multiple clock domains, and the abstract packet interface type.

Synchronization

Sync Blocks

Sync blocks primarily align packets with corresponding tuples coming from multiple sources (to the line clock domain). Secondly, sync blocks also facilitate backpressure handling. They implement backpressure support by providing buffers to absorb backpressure. Currently, each sync block has one backpressure input acting on the entire sync block.

Sync blocks are sized to accommodate buffering output from engines to hold data that arrives early and to catch a pipeline's amount of data when backpressure occurs. Essentially, a sync block FIFO signals that it is almost full at approximately one pipeline's depth number of clock cycles. At this almost full point, it asserts backpressure on the preceding/upstream sync block. This stops the output of that sync block which is the input into the preceding engine. The sync block has the remaining buffer capability to catch the remaining full pipeline of the engine, while it drains. Because the actual size must be a power of two, the size is rounded up to the next power of two.

Bridge Blocks

Bridge blocks are essentially a wrapped asynchronous FIFO to convert tuples going between the line clock domain (parsers and editors) to the packet rate clock domain (lookup engines).

Protocol Adapters

Protocol adapter blocks are generated at the edges of the system to adapt the packet signaling to an internally used bus type for systems. This is mainly for supporting the encoding of "empty" packets, which are created when an editor removes the full packet data. This is in particular required when the bus type is LBUS for correct operation of empty packets. These are necessary for maintaining the dataflow synchronization with any associated tuples in the system.

SDNet Functional Specifications

SDNet systems are described in the PX language, also referred to as SDNet functional specifications. This chapter provides an overview of the format. For more detailed descriptions, see the *SDNet Functional Specification User Guide* (UG1016) [Ref 2].

Terminology

The term *section* in this document refers in general to any part of a packet that can be in either the header, the payload, or the trailer.

Structure Declaration

The *struct* declaration defines the format of the data structure to be parsed and consists of one or more fields. Typically a structure is defined for each nested protocol. The order of the fields follows the standard networking bit order, with the most significant bit of the first field transmitted first, and the least significant bit of the last field transmitted last.

Example

An Ethernet Media Access Controller (MAC) header consisting of a:

- destination MAC address (48 bits)
- source MAC address (48 bits)
- type field (16 bits)

is defined as follows:

```
struct {
    dmac : 48,
    smac : 48,
    type : 16
}
```

Parsing Engine

The parsing engine class declaration is:

```
class identifier :: ParsingEngine(maxParseSize, maxSectionDepth, firstSection) {
    Statements
}
```

The identifier is the name of the *ParsingEngine* subclass that defines the parsing engine instance. The first parameter (*maxParseSize*) specifies the maximum initial part of the packet analyzed by the engine. The second parameter (*maxSectionDepth*) specifies the maximum number of packet sections (for example nested protocol levels) the engine traverses. The third parameter (*firstSection*) is the name of a section subclass declared in the description that is the first section to be traversed.

The description of the *ParsingEngine* subclass contains four types of statements, arranged in any order that is logical for the user:

1. Section subclass declarations
2. Structure declarations
3. Constant declarations
4. Tuple subclass declarations

Section Subclass

The role of the section subclass is to provide the parser with enough information to extract the relevant bit fields of the current protocol, identify the next encapsulated protocol, and determine the location of the next encapsulated protocol. An example section follows:

```
class identifier :: Section () {
    struct Structure ;
    method move_to_section = classexpr ;
    method increment_offset = valueexpr ;
}
```

move_to_section

The *move_to_section* method specifies the name of the next packet section to be parsed. This either provides the name of a packet section class (which might be the same as the current section class) or provides the special done class. The latter indicates the end of parsing, and is associated with a non-negative integer value that indicates the final status of the parse, for example: `done(0)`.

> ***Note:*** Specifying `done(0)` is necessary for normal termination and to avoid an error condition. Alternatively, specifying `done(value)`, where *value* is non-zero indicates an error has occurred

and will mark the packet's control tuple so that no further processing occurs by any downstream parsers or editors.

Example

When following the Ethernet MAC header, one or more of the following three protocols is expected:

- Virtual Local Area Network (VLAN)
- Internet Protocol v4 (IPv4)
- IPv6

the *move_to_section* method is expressed as:

```
method move_to_section =
    if (type == 0x8100) VLAN
    else if (type == 0x0800) IPv4
    else if (type == 0x86dd) IPv6
    else done(1);
```

where VLAN, IPv4, and IPv6 are section subclasses that define the format of the VLAN, IPv4, and IPv6 protocols.

The above declaration can be expressed using defined constants instead of values, as follows:

```
const VLAN_TYPE = 0x8100;
const IPv4_TYPE = 0x0800;
const IPv6_TYPE = 0x86dd;
const FAILURE = 1;
method move_to_section =
    if (type == VLAN_TYPE) VLAN
    else if (type == IPv4_TYPE) IPv4
    else if (type == IPv6_TYPE) IPv6
    else done(FAILURE);
```

increment_offset

The *increment_offset* method specifies the number of bits to skip in the packet to move from the current packet section to the next section to be processed. This value is used to find the beginning of the next protocol that will be parsed by the subsequent packet processor section. The method should return a non-negative integer value. This means that parsing always proceeds in a forward direction in the packet.

Example

To find the beginning of the next header following the Ethernet MAC header:

```
method increment_offset = 112; // 48+48+16 = 112
```

For sections with a fixed length, the built-in function `sizeof()` can be used as follows:

```
method increment_offset = sizeof(SectionName);
```

where `sizeof(SectionName)` returns the size of the structure declared in the class.

Note: It is not necessary to specify `increment_offset`. The default behavior, if not specified, is to increment the offset by `sizeof(SectionName)`.

It is also possible to use the *set_offset* method to specify an absolute bit position to skip, relative to the start of packet.

Optional SDNet Functional Specification Constructs

A general section class declaration is shown below. The difference between this declaration and the declaration in Section Subclass, page 18 is the addition of the optional map declaration and update methods.

```
class identifier :: Section ( levels ) {
    struct Structure :
    map MapName declarations
    method update = {... , ...}
    method move_to_section = classexpr ;
    method increment_offset = valueexpr ;
}
```

The levels qualifier is optional and is explained in Section Numbering, page 36.

Map Declaration

Map declarations are used to create lookup tables that map keys (for example, extracted subfields) to either section names or non-negative integer constants. Each key is a non-negative integer constant and all keys must be unique. *MapName* is an identifier that can be used to reference the table for lookup operations. The map declaration directs the compiler to create a small internal content-addressable memory (CAM). The maximum number of map entries is 64.

Optionally, a default result can be specified and used when no match is made for a key supplied to the map. If this is not present and no match is made, the result of the map is undefined. A declaration has the following form:

```
(key1, result1),
(key2, result2),
...
(keyN, resultN),
defaultResult
}
```

Example:

The Ethernet Type field is mapped to subclasses as:

```
const VLAN_TYPE = 0x8100;
const IPv4_TYPE = 0x0800;
const IPv6_TYPE = 0x86dd;
const FAILURE = 1;
map types {
(VLAN_TYPE, VLAN),
(IPv4_TYPE, IPv4),
(IPv6_TYPE, IPv6),
done(FAILURE)
}
```

where VLAN, IPv4, and IPv6 are section classes that define the format of the VLAN, IPv4, and IPv6 protocols.

The following statement performs a lookup using the above map declaration:

```
method move_to_section = types(type);
```

For each key (VLAN_TYPE, IPv4_TYPE, and IPv6_TYPE), the above statement associates a subclass with the *move_to_section* method. Note that the above statement is equivalent to the following:

```
method move_to_section =
if (type == VLAN_TYPE) VLAN
else if (type == IPv4_TYPE) IPv4
else if (type == IPv6_TYPE) IPv6
else done(FAILURE);
```

The advantage of using a map declaration over an if-else statement is that the map declaration is implemented using fewer logic resources and results in lower latency.

Tuple Subclass

A tuple represents a data structure consisting of one or more elements (bit fields) that can be used for passing meta-data as input, for internal operations, and/or as output to a downstream engine. An output tuple is used to transmit extracted header information such as a search key or other sideband meta-data such as a port number or the parameters for a downstream engine. A tuple is declared as follows:

```
class tupleIdentifier :: Tuple ( direction ) {
    Structure specification
}
tupleIdentifier objectIdentifier;
```

direction can be in for input only, out for output only, inout for input or output. If direction is omitted, the tuple is used for internal operations.

Example

The following output tuple contains two fields:

- Destination MAC
- VLAN ID

```
class MyTuple :: Tuple (out)
    struct {
        dmac : 48,
        vid : 12
    }
}
MyTuple key;
```

The tuple fields are assigned using the update method described as follows.

update Method

The update method is used to update or assign tuples.

Example

To assign the dmac field in the key tuple:

```
method update = { key.dmac = dst_mac }
```

To assign both dmac and vid fields in the key tuple:

```
method update = { key.dmac = dst_mac,
                  key.vid = vlan_id }
```

dst_mac and *vland_id* are fields belonging to a structure defined in a packet section class, for example:

```
struct {
    dst_mac : 48,
    src_mac : 48,
    type : 16,
    user_prio : 3,
    cfi : 1,
    vlan_id : 12
}
```

A Simple Parsing Engine

The following example implements a simple parsing engine for an Ethernet header followed by a VLAN header, and extracts the destination and source MAC addresses and VLAN ID.

```
// Output tuple to hold extracted packet fields
class MyParsedTuple :: Tuple(out) {
    struct {
    dmac : 48,
```

```
        smac : 48,
        vid : 12
    }
    class VerySimpleParser :: ParsingEngine (64, 2, ETH) {
        // Constant declarations
        const VLAN_TYPE = 0x8100;
        const SUCCESS = 0;
        const FAILURE = 1;
        // Output tuple
        MyParsedTuple parsed;
        // Packet section for an Ethernet MAC header
        class ETH :: Section() {
            struct {
                dmac : 48,
                smac : 48,
                type : 16
            }
            // Load dmac and smac in an output tuple
            method update = {
                parsed.dmac = dmac,
                parsed.smac = smac
            }
            // Move to a VLAN header
            method move_to_section =
                if (type == VLAN_TYPE) VLAN
                else done(FAILURE);
            // To move to a VLAN header
            method increment_offset = sizeof(ETH);
        } // ETH
        // VLAN header class
        class VLAN :: Section() {
            struct {
                pcp : 3,
                cfi : 1,
                vid : 12,
                tpid : 16
            }
            // Load VLAN ID in an output tuple
            method update = {
                parsed.vid = vid
            }
            // increment_offset may not need to be used if we're done
            // parsing the packet, but it is good practice to assign it
            // properly for future use.
            // If saving logic resources is important, increment_offset
            // can be set to zero at the end of parsing.
            method increment_offset = sizeof(VLAN);
            // End of parsing
            method move_to_section = done(SUCCESS);
        } // VLAN
    } // VerySimpleParser
```

Editing Engine

The SDNet packet editing capabilities comprise inserting, removing, or updating any number of bits within a packet header or payload. Two data items are required:

- The starting bit location
- The width of the field being removed, inserted, or modified

The starting bit position is set using the *increment_offset*, *set_offset*, or *set_virtual_offset* methods described as follows.

Editing Engine Offsets and Offset Methods

In the parsing engine there is a single offset (bit count from start of packet) that indicates the current parsing position in the packet. However, in the editing engine, because the packet length can be modified by adding or removing data, users might need to keep track of the current editing positions in both the input (unmodified) packet and the edited packet.

Offset and Virtual Offset

Offset refers to the current bit position in the edited packet and virtual offset, to the current bit position in the original packet. The relationship between offset and virtual offset depends on the change in packet length (from the start of packet to the current position in the edited packet) caused by previous editing actions (see Table 3-1).

Table 3-1: **Offset and Virtual Offset**

Packet Editing from SOP to Current Bit Position	Offset and Virtual Offset
No insertion or deletion Modifications with no length change Number of bits inserted equals number of bits deleted	Offset = Virtual offset
Number of bits inserted exceeds number of bits deleted	Offset > Virtual offset
Number of bits inserted is less than number of bits deleted	Offset < Virtual offset

Offset Methods

The editing engine supports three offset manipulation methods:

- increment_offset
- set_offset
- set_virtual_offset

increment_offset Method

The *increment_offset* method advances the editing position in the packet by the value or expression specified on the right-hand side of the equal sign, as shown in this example:

```
method increment_offset = 8 + 8; // Advance editing position by 2 bytes
```

increment_offset increments both offset and virtual offset by the value of the increment as shown in Figure 3-1.

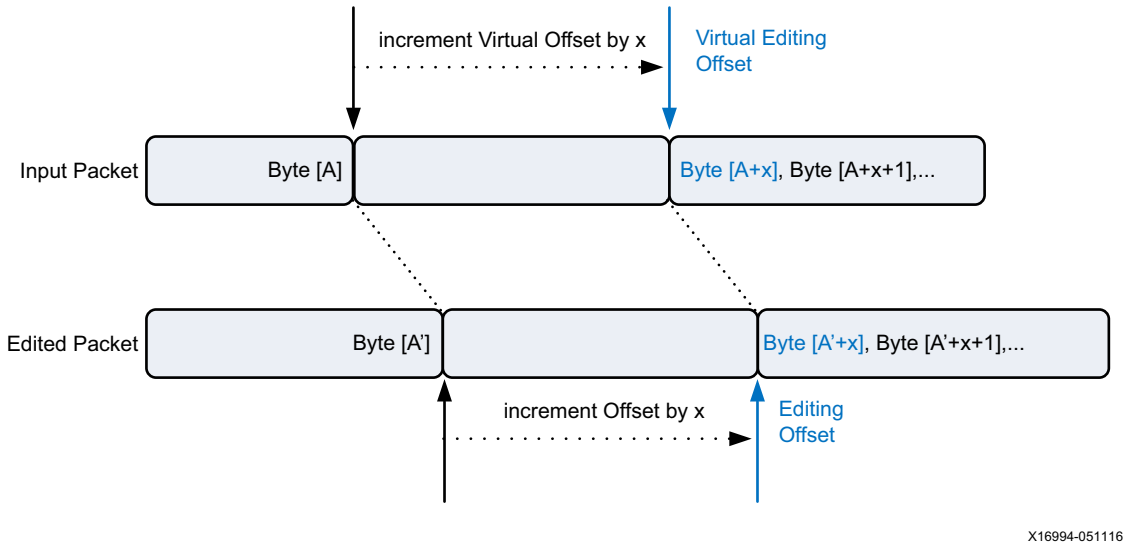


Figure 3-1: ***increment_offset* Method**

set_offset Method

The *set_offset* method sets the editing position (offset) in the packet equal to the value or expression specified on the right-hand side of the equal sign, as shown in this example:

```
const DMAC_LEN = 48;
const SMAC_LEN = 48;
// Set editing position to Ethertype
method set_offset = DMAC_LEN + SMAC_LEN;
```

set_offset sets the offset to an absolute (not relative) value and increments virtual offset by the amount that the offset was effectively incremented relative to its previous position (`x-Offset`), as shown in Figure 3-2.

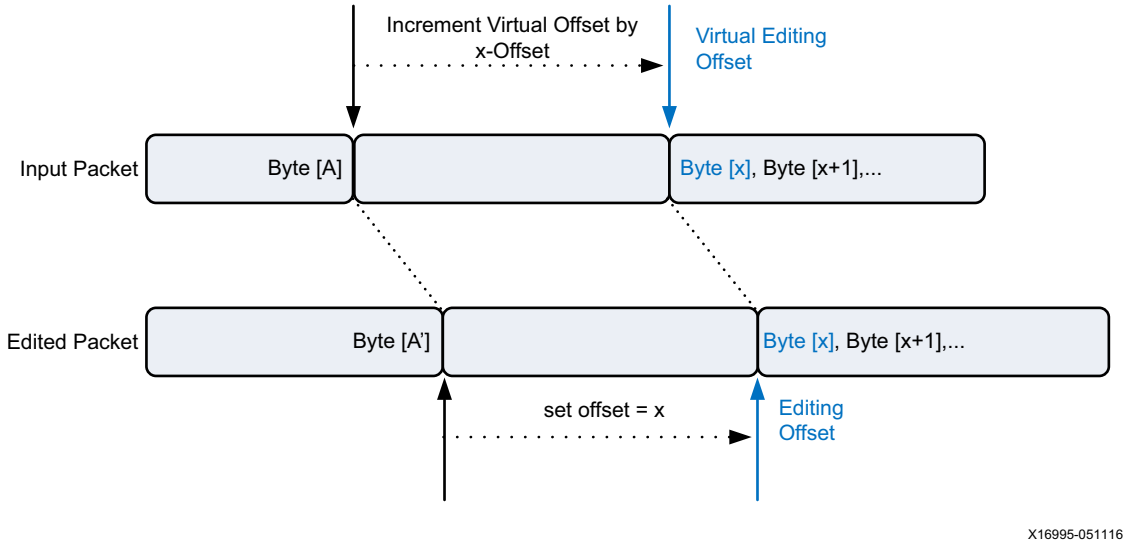


Figure 3-2: ***set_offset* Method**

set_virtual_offset Method

In some cases it might be more convenient to set the editing position relative to the original, unchanged packet. This can be done using the *set_virtual_offset* method as shown in this example:

```
const DMAC_LEN = 48;
const SMAC_LEN = 48;
const TYPE_LEN = 16;
const IP_START = DMAC_LEN + SMAC_LEN + TYPE_LEN
// Set editing position relative to input packet to IP header
method set_virtual_offset = IP_START;
```

set_virtual_offset sets the virtual offset to an absolute (not relative) value and increments the offset by the amount that the virtual offset was effectively incremented relative to its previous position (`x-Virtual Offset`), as shown in Figure 3-3.

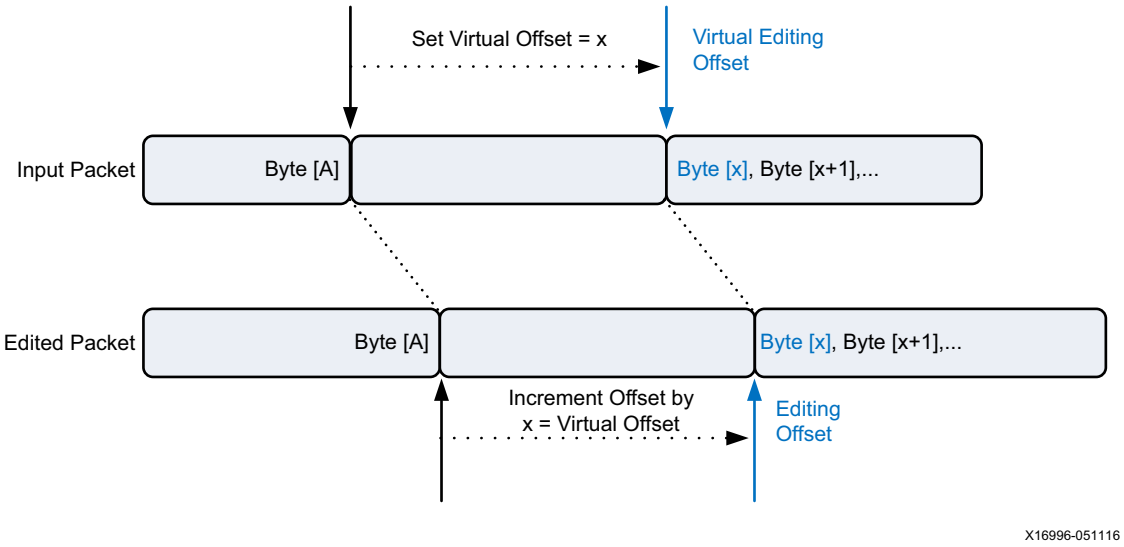


Figure 3-3: ***set_virtual_offset* Method**

Packet Editing Methods

remove Method

The *remove* method specifies that a number of data bits are to be removed at the current offset in the packet, with a value expression giving the number of bits. The special value expression rop() can be used to remove all remaining bits of the packet. The method has the following general form:

```
method remove = valueexpr ;
```

When a structure is present in the section, then a short form can be used when the number of bits removed is equal to the size of the structure:

```
method remove ;
```

valueexpr can be set to rop()to remove the rest of the packet; that is, from the current offset to the end of packet. The effect of the remove method is shown in Figure 3-4.

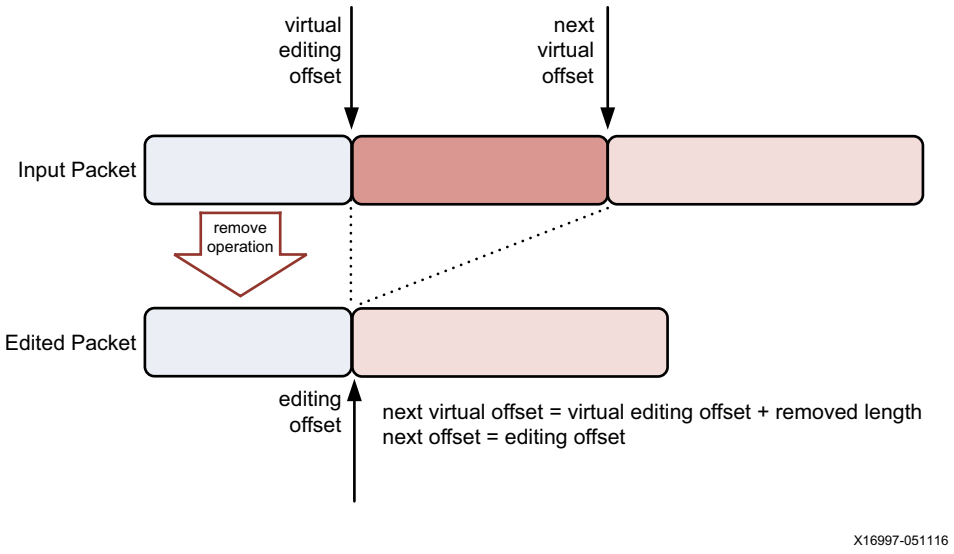


Figure 3-4: ***remove* Method**

Example

To remove the VLAN type and tag following the destination and source MAC addresses:

```
method increment_offset = 96 // skip DMAC and SMAC addresses
method remove = 32 // remove type (16 bits) and VLAN tag (16 bits)
```

Alternatively, this editing action can be specified as follows:

```
// Packet section for Ethernet MAC header
class ETH :: Section() {
    struct {
        dmac : 48,
        smac : 48
    }
    // Move to a VLAN header
    method move_to_section = VLAN;
    // To move to the VLAN header
    method increment_offset = sizeof(ETH);
} // ETH
// VLAN header class
class VLAN :: Section() {
    struct {
        type : 16,
        vlan : 16
    }
    // remove vlan type and tag
    method remove;
}
```

insert Method

The insert method specifies that a number of data bits are to be inserted before the current offset in the packet, with a value expression giving the number of bits. The inserted bits are determined as follows (where struct size is defined as zero if no structure is present in the section):

- If the value of the expression *v* is less than or equal to the struct size, then the initial *v* bits of the structure are used;
- If the value of the expression *v* is greater than the struct size, then the *s* bits of the structure padded with *v-s* trailing zero bits are used.

This method has the following general form:

```
method insert = valueexpr ;
```

When a structure is present in the section, then a short form can be used when the number of bits inserted is equal to the size of the structure:

```
method insert ;
```

The effect of the insert method is shown in Figure 3-5.

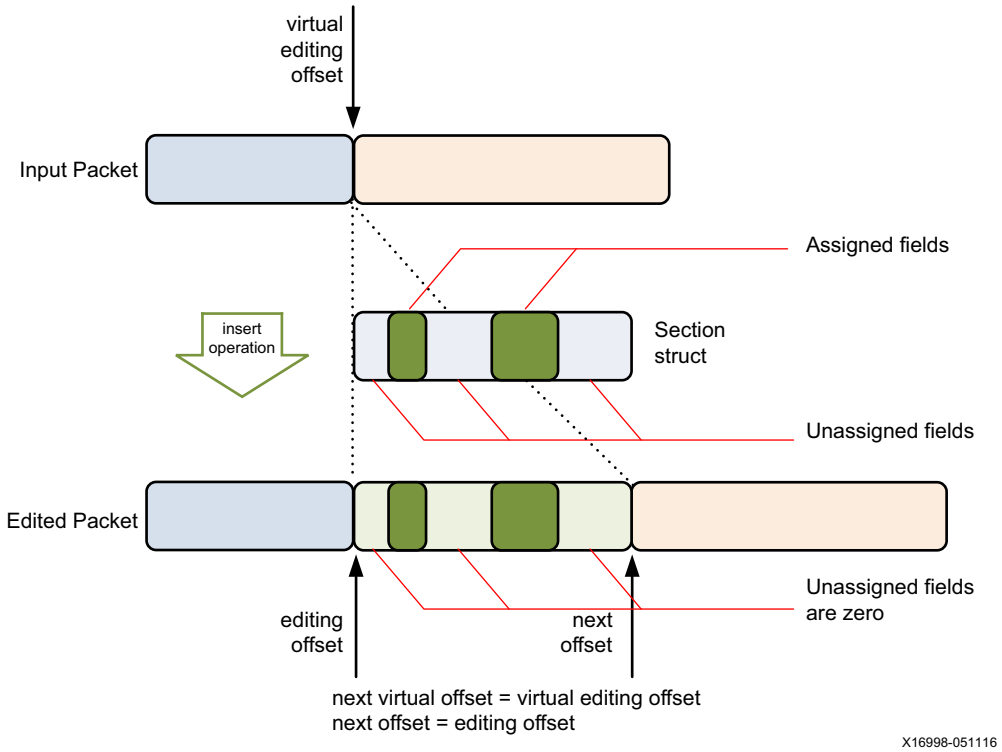


Figure 3-5: ***insert* Method**

Example

To insert a VLAN type and tag following the destination and source MAC addresses:

```
// VLAN header to be inserted
class VLAN :: Section() {
    struct {
        type : 16,
        vlan : 16
    }
    method increment_offset = 96; // skip DMAC and SMAC addresses
    method insert; // insert type (16 bits) and VLAN tag (16 bits)
}
```

Values are assigned to the new fields by means of input tuples using the *update* method. An example of this is shown in the next section.

Modifying Bits in a Packet

The update method can be used to modify bits in a packet without adding or removing bits. Figure 3-6 shows the effect of a modification using the update method.

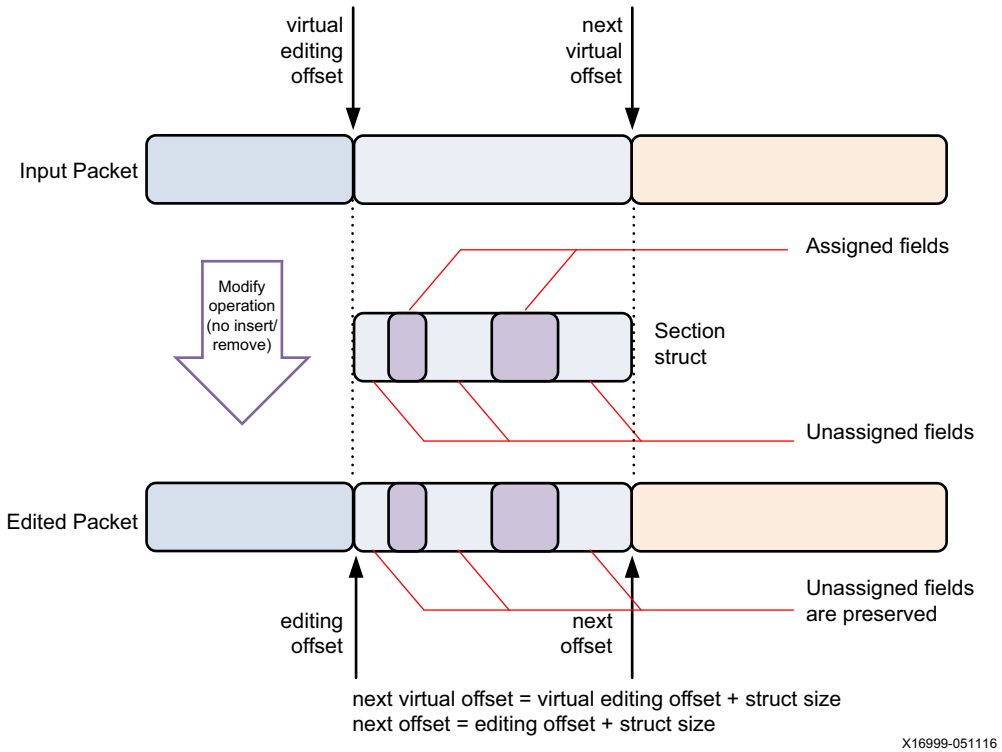


Figure 3-6: **Modifying Bits in a Packet**

Example

To change the VLAN priority bits:

```
// VLAN header class
class VLAN :: Section() {
    struct {
        tpid : 16
        pcp : 3,
        cfi : 1,
        vid : 12,

    }
    method increment_offset = 96; // skip DMAC and SMAC addresses
    method update = {
        pcp = 7 // update VLAN priority to 7
    }
}
```

Replacing a Packet Segment

Replacing a segment of a packet can be done in a single editing step using the *remove* method to remove the old section, and the *insert* method to insert the new section. Both the *remove* and *insert* methods are used in the same packet editing section. Figure 3-7 shows the effect of a replace operation using the *remove* and *insert* methods.

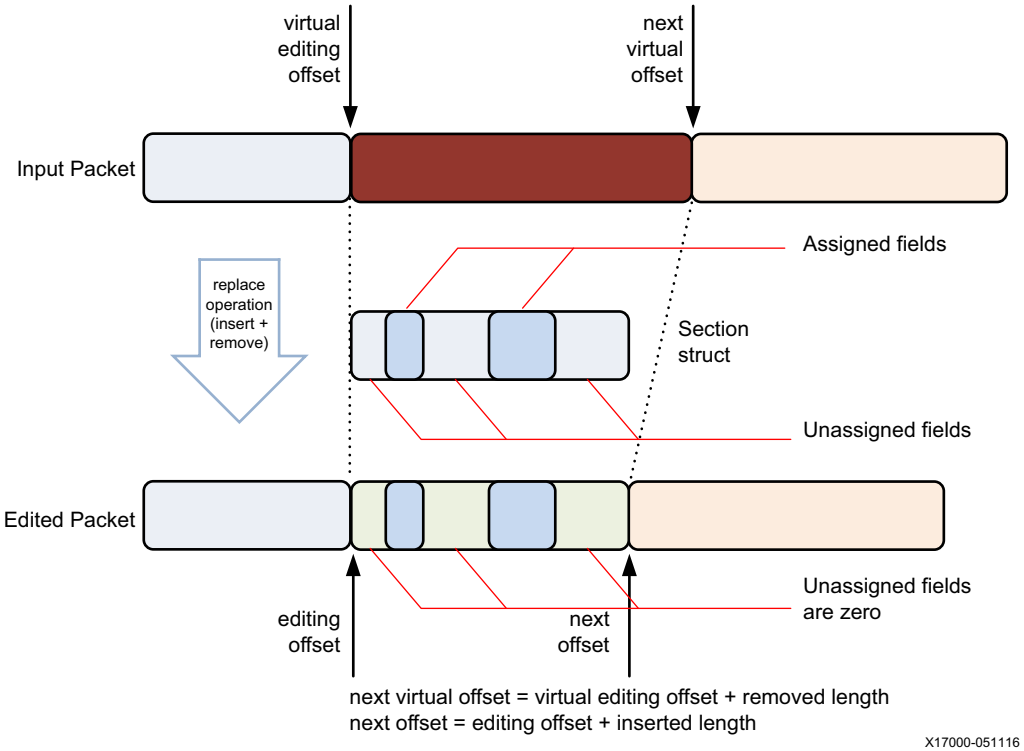


Figure 3-7: **Replacing a Packet Segment**

Example

To replace a 16-bit field in the packet payload with a 32-bit field:

```
class PayloadWord :: Section() {
    struct {
        PWord32 : 32
    }
    method increment_offset = 48+48+16; // skip minimum Ethernet header
    method remove = 16;
    method insert;
}
```

This example assumes that the payload starts after a minimum Ethernet header of 14 bytes. If that is not the case, the specification can be modified to skip over all of the protocol headers and set the editing offset to the beginning of the payload.

Computational Model

Regardless of the order of the statements in the SDNet specification, the packet editor performs the computation and actions in the order shown in Figure 3-8.

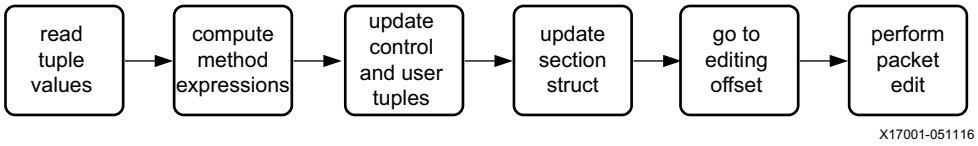


Figure 3-8: **Computational Model**

1. Read tuple values.
   - Load the values of each tuple field used in the packet editing section.
2. Compute method expressions.
   - Calculate the values of all method expressions in parallel.
3. Update control and user tuples.
   - Set control done and section field according to move_to_section method result.
   - Set control offset and virtual offset fields to the bit position immediately following the last edited bit.
   - Load user tuple fields assigned in the update methods with the computed values.
4. Update section struct.
   - Load struct fields assigned in update methods.
5. Go to new offset.
   - Wait for data cycle carrying bit referenced by calculated editing offset.
6. Perform packet edit.
   - Execute editing action (insert, remove, or in-place modify).

An Editing Engine Example

The following editing engine example performs the following actions on an Ethernet/IPv4 packet:

1. Inserts a VLAN tag passed in through an input tuple (for example, input port number).
2. Decrements the IPv4 TTL field and updates the checksum.
3. Removes the CRC from the end of the packet.

```
// tuples used for editing functions
// VLAN tuple carries input port number
class VLAN_Tuple :: Tuple(in) {
    struct {
        PortNum : 12
    }
}
// IPv4 tuple carries key info from IPv4 header
class IPv4_Tuple :: Tuple(in) {
    struct {
        ttl : 8,
        hdr_csum : 8,
        length : 16
    }
}
// Editor engine class definition
class MyEditor :: EditingEngine(1500*8, 3, VLAN) {
    VLAN_Tuple EditTuple1;
    IPv4_Tuple EditTuple2;
    class VLAN :: Section (1) {
        struct {
            tpid: 16,
            pcp : 3,
            dei : 1,
            vid : 12
        }
        // Set values VLAN structure
        // pcp and dei will defaut to zero since they're unassigned
        method update = {
            tpid = 0x8100,
            vid = EditTuple1.PortNum
        }
        // Insertion point after DMAC and SMAC
        method set_offset = 96;
        method insert;
        method move_to_section = IPv4;
    }
    class IPv4 :: Section (2) {
        struct {
            version : 4,
            hdr_len : 4,
            dscp : 6,
            ecn : 2,
            length : 16,
            id : 16,
            flags : 3,
            offset : 13,
```

```
            ttl : 8,
            protocol : 8,
            hdr_csum : 16,
            src_addr : 32,
            dst_addr : 32
        }
        // The method increment_offset has slightly different behavior when used in
        // an EditingEngine, and it is called before any edits.
        // Skip over Ether Type (16)
        method increment_offset = 16;
        // Decrement TTL (if > 0) and update checksum
        method update = {
            ttl = if (EditTuple2.ttl > 0) EditTuple2.ttl - 1 else 0,
            hdr_csum =
                if (EditTuple2.ttl > 0)
                    ~(~EditTuple2.hdr_csum + ~EditTuple2.ttl + EditTuple2.ttl - 1)
                else EditTuple2.hdr_csum // RFC 1624
        }
        method move_to_section = Trailer;
    }
    class Trailer :: Section (3) {
        // Skip over packet (length from IPv4 header)
        method increment_offset = EditTuple2.length;
        // Remove everything till EOP (to remove CRC)
        method remove = rop();
        method move_to_section = done(0);
    }
}
```

Procedure for Writing SDNet Functional Specifications

SDNet functional specifications support any packet format with up to 64 levels of nested protocols. The simple parsing example above had two levels of nesting: Ethernet -> VLAN. This nesting can be extended as required to parse deeper protocol layers. In general, every protocol layer is handled by its own section subclass. For each protocol layer to be parsed, the user creates a section subclass with the following constructs:

1. A *struct* to describe the format of the protocol fields.
2. A *move_to_section* method to indicate all the possible protocols at the next level.
    - If the current packet header is at the last level of encapsulation, *move_to_section* can be set to *done(0)* or *done(value)*, with value representing a user-defined exit code.
    - Alternatively, *done(0)* can indicate success, and *done(non-zero)* can indicate failure.
    - If the current packet payload contains another protocol that needs further parsing, the *move_to_section* is set to the name of the expected header (example Ethernet -> VLAN). The *move_to_section* expression can also contain conditional operators to select among several options, and content-addressable memory (CAM) lookups.

Note: Refer to the *SDNet Functional Specification User Guide* (UG1016) [Ref 2] for a detailed description of allowed expression syntax.

3. An *increment_offset* method to indicate the number of bits needed to reach the next protocol section. If the next protocol layer starts at the end of the current header, *increment_offset* can be set to *sizeof(SectionName)*. Otherwise the offset to the header must be computed using a formula based on information contained in the packet (e.g., IPv4).

   Alternatively, the *set_offset* method can be used to set the processing position using an absolute value (relative to the start of packet).

In addition, the user defines an output tuple that contains the result of the parsing as follows.

1. List the information that needs to be extracted from the packet.
2. Add each field of the required information structure to the output tuple definition.
3. Add an expression for each information field using the *update* method.

Check for conflicts and consistency between all headers that assign into a particular field and ensure that the default value is correct (the default value is zero for 'output' tuples, and user-controlled via input signals or 'inout' tuples).

An example with multiple nested protocols follows, but the notion of section numbering is discussed first.

Section Numbering

The packet processor is configured by the programmer to handle a maximum number of sequential sections in a packet, according to the *maxSectionDepth* parameter. Its usage is as follows:

```
class identifier :: EngineType (maxParseSize,maxSectionDepth,firstSection)
```

maxSectionDepth is usually equal to the number of nested protocol levels to be parsed. The SDNet functional specification compiler determines which packet sections (section classes) can occur at each level of the section stack. By default, the compiler assumes that all section classes can occur at all levels, which can result in an implementation that uses more logic than required. The user can provide information that allows the compiler to configure the packet processor IP more efficiently (that is, process only the expected protocols at a particular level) using section numbering as explained in the following illustrative example.

Example with 5-level Protocol Stack

Consider a 5-level protocol stack as represented graphically in Figure 3-9.

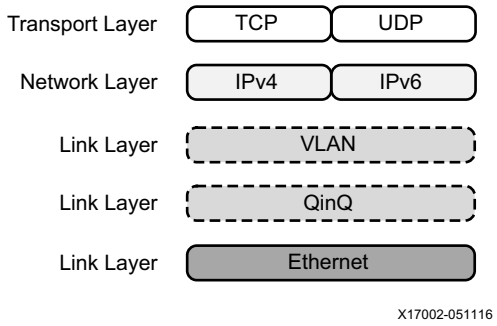


Figure 3-9: **Five-level Protocol Stack**

The queue in queue (QinQ) and VLAN sections are optional, resulting in the possible packet formats shown in Figure 3-10.

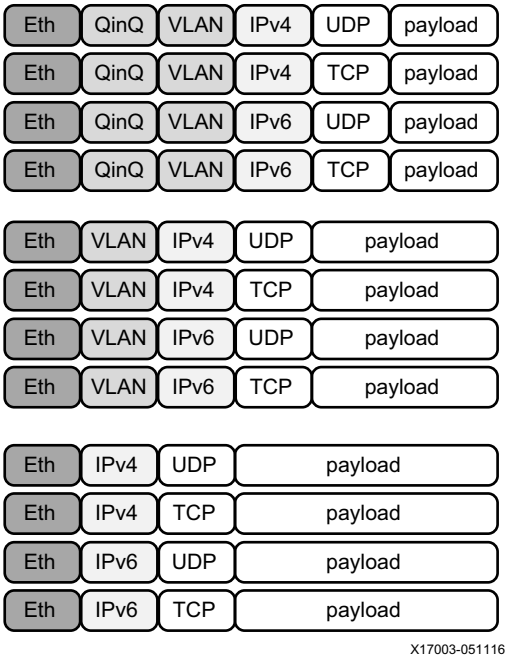


Figure 3-10: **Packet Formats for the Five-level Protocol Stack**

Where:

a. Level 1 is Ethernet MAC.

b. Level 2 can be QinQ, VLAN, IPv4, or IPv6.

c. Level 3 can be VLAN, IPv4, IPv6, Transport Control Protocol (TCP), or User Datagram Protocol (UDP).

d. Level 4 can be IPv4, IPv6, TCP, or UDP.

e. Level 5 can be TCP or UDP.

To process the above five layers, a 5-section packet processor IP is required (Figure 3-11).

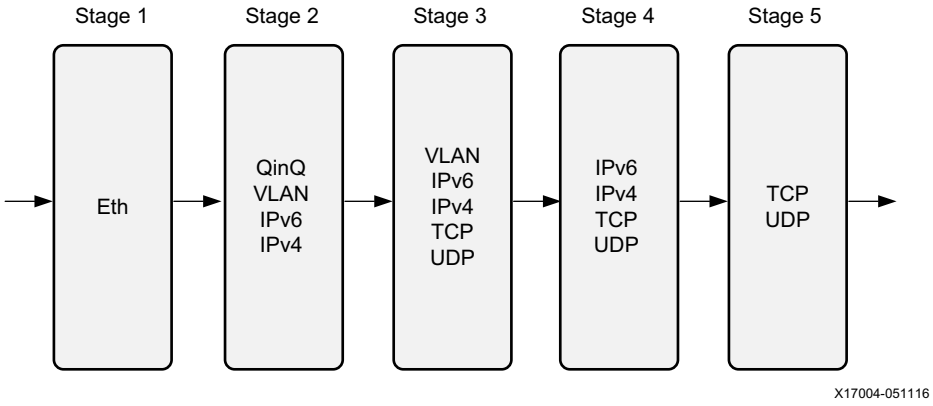


Figure 3-11: **Five Section Packet Processor**

Clearly, each section does not need to process all six protocols. The SDNet functional specification compiler generates a more logic-efficient implementation if it is provided with information about the protocols (classes) each section needs to process. This is done through the levels parameter as follows.

```
class identifier :: Section ( levels ) {
    …
}
```

For example, to indicate that the Ethernet header class (ETH) is only processed in section 1:

```
class ETH :: Section(1) {
```

To indicate that the VLAN header is processed only in sections 2 and 3:

```
class VLAN :: Section(2:3) {
```

The following SDNet functional specification parses the protocol stack and extracts the fields listed in the output tuple below. Note that the use of the level's parameter is in accordance with Figure 3-11.

```
// Parser engine instance:
// Can accept up to 9kB packets (or initial packet sections)
// Instantiate 5 processing sections
// First section is Ethernet header
class SimpleParser :: ParsingEngine (9416*8, 5, ETH) {
    // Constants definitions
    const VLAN_TYPE = 0x8100;
    const IPv4_TYPE = 0x0800;
    const IPv6_TYPE = 0x86dd;
    const TCP_TYPE = 0x06;
    const UDP_TYPE = 0x11;
    // For done() statements
    const SUCCESS = 0;
    const FAILURE = 1;
    // Define output tuple that contains packet fields to be extracted
```

```
class ParsedTuple :: Tuple (out) {
    struct {
        dmac : 48, // Destination MAC address
        smac : 48, // Source MAC address
        tpid : 16, // Tag Protocol Identifier
        ipv4 : 1, // IPv4/v6 flag (0 = IPv6, 1 = IPv4)
        prot : 8, // Protocol encapsulated in IPv4/v6 (TCP/UDP)
        src : 128, // IPv4/v6 source address
        dst : 128, // IPv4/v6 destination address
        sp : 16, // TCP/UDP source port
        dp : 16 // TCP/UDP destination port
    }
}
ParsedTuple fields;
// Ethernet MAC header
class ETH :: Section(1) {
    struct {
        dmac : 48, // Destination MAC address
        smac : 48, // Source MAC address
        type : 16 // Tag Protocol Identifier
    }
    // Extract DMAC and SMAC to output tuple
    method update = {
        fields.dmac = dmac,
        fields.smac = smac
    }
    // Following Ethernet MAC header, VLAN, IPv4, or IPv6 are expected
    method move_to_section =
        if (type == VLAN_TYPE)
            VLAN
        else if (type == IPv4_TYPE)
            IPv4
        else if (type == IPv6_TYPE)
            IPv6
        else
            done(FAILURE);
    // Find next protocol header
    method increment_offset = sizeof(ETH);
} // ETH
// VLAN header
class VLAN :: Section(2:3) {
    struct {
        pcp : 3,
        cfi : 1,
        vid : 12,
        tpid : 16
    }
    // Mapping table of type field to class
    // VLAN can be followed by a second VLAN, IPv4, or IPv6
    map types {
        (VLAN_TYPE, VLAN),
        (IPv4_TYPE, IPv4),
        (IPv6_TYPE, IPv6),
        done(FAILURE)
    }
    // Extract Tpid to output tuple
    method update = {
        fields.tpid = tpid
    }
```

```
    // Find next protocol
    method increment_offset = sizeof(VLAN);
    // For next protocol use class looked up in "types" table
    method move_to_section = types(tpid);
} // VLAN
// IPv4 header
class IPv4 :: Section(2:4) {
    struct {
        version : 4, // Version (4 for IPv4)
        hdr_len : 4, // Header length in 32b words
        tos : 8, // Type of Service
        length : 16, // Packet length in 32b words
        id : 16, // Identification
        flags : 3, // Flags
        offset : 13, // Fragment offset
        ttl : 8, // Time to live
        protocol: 8, // Next protocol
        hdr_chk : 16, // Header checksum
        src : 32, // Source address
        dst : 32, // Destination address
        options : * // IPv4 options - variable length field
        // Not parsed
    }
    // Set IPv4 flag
    // Extract protocol and source and destination addresses
    method update = {
        fields.ipv4 = 1,
        fields.prot = protocol,
        fields.src = src,
        fields.dst = dst
    }
    // Move by a number of bits equal to the value in hdr_len
    // multiplied by 32 (hdr_len is the length of the header in
    // 32-bit words)
    method increment_offset = hdr_len * 32;
    // Next header is TCP or UDP
    method move_to_section =
        if (protocol == TCP_TYPE)
            TCP
        else if (protocol == UDP_TYPE)
            UDP
        else
            done(FAILURE);
} // IPv4
class IPv6 :: Section(2:4) {
    struct {
        version : 4, // Version = 6
        priority : 8, // Traffic class
        flow_label : 20, // Flow label
        length : 16, // Payload length
        next_hdr : 8, // Next protocol
        hop_limit : 8, // Hop limit
        src_addr : 128, // Source address
        dst_addr : 128 // Destination address
    }
    // Extract protocol and source and destination addresses
    method update = {
        fields.ipv4 = 0, // Clear IPv4 flag
        fields.prot = next_hdr,
```

```
                fields.src = src_addr,
                fields.dst = dst_addr
            }
            // Next header is TCP or UDP
            method move_to_section =
            if (next_hdr == TCP_TYPE)
                TCP
            else if (next_hdr == UDP_TYPE)
                UDP
            else
                done(FAILURE);
            // Move to next protocol
            method increment_offset = sizeof(IPv6);
        } // IPv6
        // TCP header
        class TCP :: Section(3:5) {
            struct {
                src_port : 16, // Source port
                dst_port : 16, // Destination port
                seqNum : 32, // Sequence number
                ackNum : 32, // Acknowledgment number
                dataOffset : 4, // Data offset
                resv : 6, // Offset
                flags : 6, // Flags
                window : 16, // Window
                checksum : 16, // TCP checksum
                urgPtr : 16, // Urgent pointer
                options : * // Options (variable length)
            }
            // Extract source and destination ports
            method update = {
                fields.sp = src_port,
                fields.dp = dst_port
            }
            // To find next protocol (for future use)
            method increment_offset = dataOffset * 32;
            // Last protocol to parse
            method move_to_section = done(SUCCESS);
        } // TCP
        // UDP header
        class UDP :: Section(3:5) {
            struct {
                src_port : 16, // Source port
                dst_port : 16, // Destination port
                length : 16, // UDP length
                checksum : 16 // UDP checksum
            }
            // Extract source and destination ports
            method update = {
                fields.sp = src_port,
                fields.dp = dst_port
            }
            // To find next protocol (for future use)
            method increment_offset = sizeof(UDP);
            // Last protocol to parse
            method move_to_section = done(SUCCESS);
        } // UDP
    } // SimpleParser
```

Writing Optimal SDNet Functional Specifications

Besides the section numbering method described above, certain coding techniques can be used in cases where the user desires a more efficient packet processor IP implementation for a given set of protocols.

Processing Nested Protocols in a Single Section

In some cases, it might be possible to combine nested protocol into a single class to reduce the number of sections in the packet processor IP. In the 5-layer protocol design described earlier, five sections are used. If the stacked VLAN format can be processed in section 1 using a single class, then the protocol stack, packet formats and the resulting parser can be simplified as shown in Figure 3-12 to Figure 3-13, respectively.

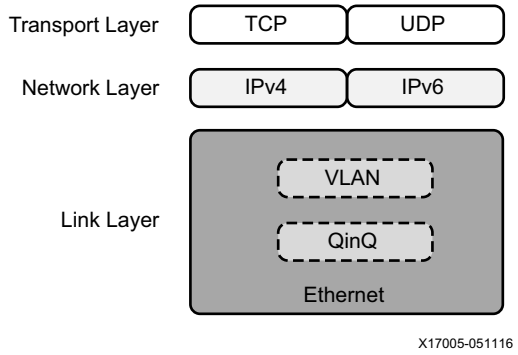


Figure 3-12: **Optimized Five-protocol Stack**

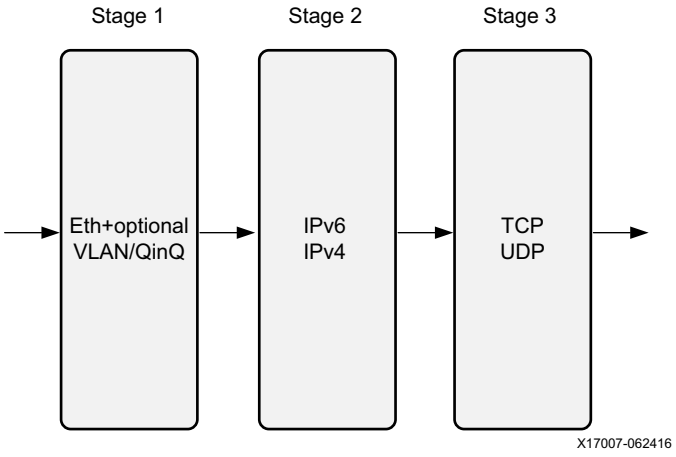


Figure 3-13: **Optimized Parser**

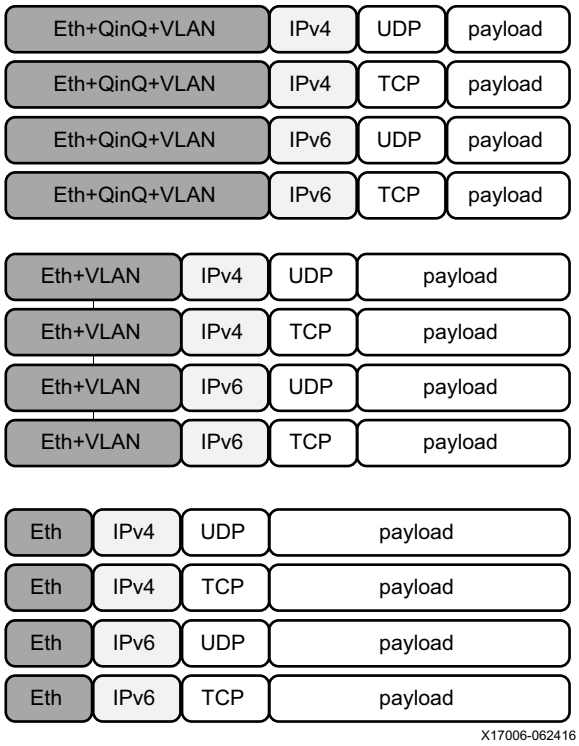


Figure 3-14: **Simplified Packet Format**

Optimized Packet Processor IP

This optimization results in eliminating levels 3 and 4 in the original implementation while extending the Ethernet section at level 1, resulting in a shorter pipeline and leading to fewer resources. The SDNet specification for the updated Ethernet protocol follows.

```
class Ethernet :: Section(1) {
    struct {
        dmac : 48,
        smac : 48,
        type : 16,
        vlan1 : 16,
        type1 : 16,
        vlan2 : 16,
        type2 : 16
    }
    map types {
        (IPv4_TYPE, IPv4),
        (IPv6_TYPE, IPv6),
        done(FAILURE)
    }
    method move_to_section =
    if (type == QINQ_TYPE)
        if (type1 == VLAN_TYPE)
            types(type2)
        else
            done(FAILURE)
    else if (type == VLAN_TYPE)
        types(type1)
    else
        types(type)
    ;
    method increment_offset =
        if (type == QINQ_TYPE)
            112+32+32 // Ethernet + QinQ + VLAN
        else if (type == VLAN_TYPE)
            112+32 // Ethernet + VLAN
        else
            112 // Ethernet
    ;
```

Processing Similar Headers

In some cases, similar headers with similar processing can be combined in a single class. As an example, consider the definition and processing of the TCP and UDP headers shown in Figure 3-12. For both protocols:

- The struct definition starts with:

```
src_port : 16,
dst_port : 16
```

- The extracted fields are:

```
method update = {
    fields.sp = src_port,
    fields.dp = dst_port
}
```

- Both TCP and UDP are the last protocols to be parsed in the stack:

```
method move_to_section = done(SUCCESS);
```

The only difference between the two class definitions is in the *increment_offset* method.

For TCP:

```
method increment_offset = dataOffset * 32;
```

For UDP:

```
method increment_offset = size();
```

However, since both TCP and UDP are the last protocols to be parsed in the stack, we can set *increment_offset* to zero:

```
method increment_offset = 0;
```

Note that if a protocol deeper than TCP/UDP needs to be parsed in the future, the *increment_offset* value would have to be different for TCP and UDP (as initially written) and therefore, this optimization would not be possible.

The combined TCP/UDP class is as follows:

```
class TCP_UDP :: Section(3,4) {
    struct {
        src_port : 16,
        dst_port : 16
        // No need to define the rest of the
        // header since it is not accessed
    }
    method move_to_section = done(SUCCESS);
    method increment_offset = 0;
    method update = {
        fields.sp = src_port,
        fields.dp = dst_port
    }
} // TCP_UDP
```

Tuple Engines

Tuple engines are helper engines for manipulating tuples and performing computations on tuple data. Tuple engines share the same constructs and syntax as parsers. The main differences are that they do not contain packet ports or a maximum offset parameter.

Example

```
class checksum :: TupleEngine(3, state1) {

  class internal :: Tuple {
    struct{
      hc_inv : 17,
      m_p    : 17,
      m_inv  : 17,
      result : 17
    }
  }

  class checksum_ipv4_tup_in :: Tuple(in) {
    struct{
      ttl : 8,
      protocol : 8,
      checksum : 16
    }
  }

  class checksum_ipv4_tup_out :: Tuple(out) {
    struct{
      data : 32
    }
  }

  checksum_ipv4_tup_in ipv4_tup_in;
  checksum_ipv4_tup_out ipv4_tup_out;
  internal vars;

  class state1 :: Section(1) {
      method update = {
    vars.hc_inv = ~ipv4_tup_in.checksum,
    vars.m_p = (ipv4_tup_in.ttl << 8) | ipv4_tup_in.protocol
      }
      method move_to_section = state2;
  }

 class state2 :: Section(2) {
      method update = {
    vars.m_inv = ~(vars.m_p + 0x0100),
    vars.result = vars.hc_inv + vars.m_inv + vars.m_p
      }
      method move_to_section = state3;
  }

  class state3 :: Section(3) {
      method update = {
          ipv4_tup_out.data = ((vars.result & 0xffff) << 16) | ((vars.result & 0x10000) >> 16)
      }
      method move_to_section = done(0);
   }
}
```

In this tuple engine example, there are three sections, which constitute the first class parameter. The second class parameter is the default section. The example recomputes a checksum, implemented as a computation spread over three sections. Tuple engines can have tuples with no direction that are entirely internal to the engine such as the "vars" tuple in this example. Note that sections are required as container classes for the update methods. Tuple engines are required to have at least one input tuple and one output tuple.

Lookup Engines

Lookup engines can be one of four types of search: exact match (EM), longest prefix match (LPM), ternary match (TCAM), or direct address match. The implementations of EM, LPM, and TCAM are algorithmic and based on recent published research in this area.

EM is used when the keys must exactly match an entry in the table. Examples are tables of MAC addresses, when making forwarding decisions. The LPM implementation is based on the IP described further in the *SmartCAM Product Guide* (PG189) [Ref 6].

LPM is typically used when making Internet Protocol (IP) next-hop forwarding decisions. LPM table sizes can scale on the order of tens of thousands of entries when using on-chip memory. Exact match supports similar table sizes to EM, but it can be much more performance efficient for incremental updates. Often networking protocols require only EM or LPM when making forwarding decisions. The LPM implementation is based on the IP described further in the *LPM Product Guide* (PG191) [Ref 8].

TCAM is typically used for multidimensional packet classification over a set of fields. When selecting the type of lookup engine it is important to consider the resource tradeoffs depending on the system design requirements. For example, although TCAM can technically be used in place of exact match, a similarly sized table can greatly exceed the resources of a similar size exact match table. Therefore TCAM is only recommended for tables requiring multidimensional search, consisting of fewer than 4096 entries. This algorithmic TCAM implementation is based on the IP described further in the *TCAM Product Guide* (PG190) [Ref 7]. *Direct* address lookup is the simplest implementation option, and is essentially just a RAM block. It can be useful for implementing small auxiliary tables or tables that are very dense. However, Direct is impractical for large table sizes.

Interface

Lookup engines have three main interfaces: request, response, and configuration. The request interface consists of a tuple containing the search key.

The response interface can be set as a tuple containing precisely one of the following struct formats (the default is format (1)):

0. { value }
1. { hit/miss, value }
2. { hit/miss, value, key }

The configuration interface is a memory-mapped control interface for programming the contents of the lookup table. The configuration interface can be used for performing dynamic updates to the table during operation.

Methods

There are two methods that need to be present in the LookupEngine class: *send_request* and *receive_response*.

- The *send_request* maps an input tuple port to the request interface.

```
method send_request = {
        key = request
    }
```

- The *receive_response* method similarly maps an output tuple port to the response.

```
method receive_response = {
        response = value
    }
```

Example

```
class MyLookup :: LookupEngine(
    TCAM, // type
    256,  // number of entries
    298,  // key_width
    16,   // value_width
    1,    // tuple "format" of the response, represented by the following enumeration:
          //   0: { value }
          //   1: { hit/miss, value }  as mentioned, this is the default format
          //   2: { hit/miss, value, key}
    1     // this specifies whether the lookup is connected externally to the system
)
{
   LookupRequest request;
   LookupResult response;

   method send_request = {
       key = request
   }
    method receive_response = {
       response = value
   }
}
```

Lookup engines can be specified to be externally connected, meaning that the request and response tuple interfaces are brought out to the top level interface of the system. This can be done by setting the corresponding (sixth) LookupEngine class parameter to "1". It is also possible to connect third-party lookup engines by setting the lookup engine to be external from the system and then connecting them to the generated top-level request and response interfaces.

Example

```
class MyLookup :: LookupEngine(
    LPM,  // type
    255,  // number of entries
    128,  // key_width
    8,   // value_width
    1,    // tuple "format" of the response, represented by the following enumeration:
          //   0: { value }
          //   1: { hit/miss, value }  as mentioned, this is the default format
          //   2: { hit/miss, value, key}
    0     // this specifies whether the lookup is connected externally to the system
)
{
   LookupRequest request;
   LookupResult response;

   method send_request = {
       key = request
   }
    method receive_response = {
       response = value
   }
}
```

User Engines

The user engine class allows users to import custom IP cores into the SDNet specification to take advantage of the SDNet framework for data plane building and system simulation capabilities.

The user engine class has two parameters. The first specifies the maximum latency of the engine in cycles. The second parameter is used to indicate the presence of a control/configuration interface for the user engine. It specifies the address width of the configuration port in bits. A zero value is used when there is no configuration port.

Example

```
class AES_CTR :: UserEngine(20, 0) {
    // user engines consist entirely of the interface declaration
    Packet_input      packet_in;
    Packet_output     packet_out;
    PadTupleIn        pad_tuple;
    CryptoTupleIn     key_tuple;
    InitVectorTupleIn iv_tuple;
    AesTupleOut       aes_tuple;
}
```

As noted in the comment within the above code, user engines specify the interface in SDNet functional specification but not the behavior. The user engine is expected to be implemented in VHDL or Verilog with a corresponding C behavioral model for high level simulation.

Example

```
class checksum :: UserEngine(1, 0) {
  class checksum_ipv4_tup_in :: Tuple(in) {
    struct{
      ttl : 8,
      protocol : 8,
      checksum : 16
    }
  }
  class checksum_ipv4_tup_out :: Tuple(out) {
    struct{
      ttl : 8,
      protocol : 8,
      checksum : 16
    }
  }
  checksum_ipv4_tup_in ipv4_tup_in;
  checksum_ipv4_tup_out ipv4_tup_out;
}
```

System Class

The system class is used to specify the dataflow graph of the packet processing system or subsystem. First, a top-level interface consisting of (1) an input packet port, (2) an output packet port, (3) zero or more tuple ports are declared. Next, a set of engines, forming the nodes of the dataflow graph is instantiated to implement the required processing. Finally, the edges of the graph are described using the connect method to complete the graph. Edges connect from the top level interface to the ports of engines according the packet processing dataflow.

At the SDNet level view the system consists entirely of a set of engines arranged in a directed acyclic graph (DAG). The compiler adds infrastructure blocks and necessary glue logic during compilation. These infrastructure blocks include: bridges, a controller, syncs, and protocol adapter blocks, for maintaining the synchronized dataflow behavior throughout the system. The role of these infrastructure blocks is described later in this section.

SDNet systems can be constructed using hierarchy for systems for modular design abstraction. Systems can be instantiated as an additional type of engine.

Example

```
class MySystem :: System {

// Interfaces, packet ports
    Packet_input  instream;
    Packet_output outstream;

// Interfaces, tuple ports
    MyParserTuple forParser;
    MyEditedTuple fromEditor;

// Engine instances
    MyLookup lookup;
    MyParser parser;
    MyEditor editor;

    method connect = {
        parser.packet_in   = instream,
        parser.ParserTuple = forParser,
        lookup.request     = parser.extractedKey,
        editor.packet_in   = parser.packet_out,
        editor.EditorTuple = lookup.response,
        fromEditor         = editor.EditedTuple,
        outstream          = editor.packet_out
    }
}
```

Note that both systems and user engines can contain the plain port type, which is used for miscellaneous RTL signaling that is not one of the other port types. Plain ports contain two

class parameters. The first parameter is the direction. The second parameter is the width in bits. An example declaration of the plain port type is shown below.

```
class MyPlainIn :: Plain(in, 12) { }
```

Top Down Approach

When writing the system class, begin with a block diagram of the system or systems.

1. Capture the top-level interface.
2. Instantiate all of the engines.
3. Connect the paths systematically.

The packet ports in the system dataflow graph need to be fully connected (no ports can be left unconnected). Packet output ports cannot fan out.

All input tuple ports must be connected. Tuple output ports are allowed to fan-out or to be left as unconnected. The recommended procedure is to:

1. Connect all of the packet ports.
2. Connect all of the tuple ports.
3. Connect any of the plain ports (if present).

Bottom Up Approach

Write and test each engine separately and then add them incrementally to build up a larger system. Re-factor the systems with hierarchy later to group related processing into separate systems.

Notes to Keep in Mind

- Packets must follow a single path through the system from a single input port to a single output port.
- There can however be multiple tuple input and output ports depending on the design. Tuples can take more complex paths through the system, including having fan-out connections, as mentioned, to multiple engines.
- The graph can be an arbitrary DAG, however, there is a limitation in that the graph is transformed into a linear pipeline for implementation.

Compilation

Compiler input and output files are shown in Figure 4-1.

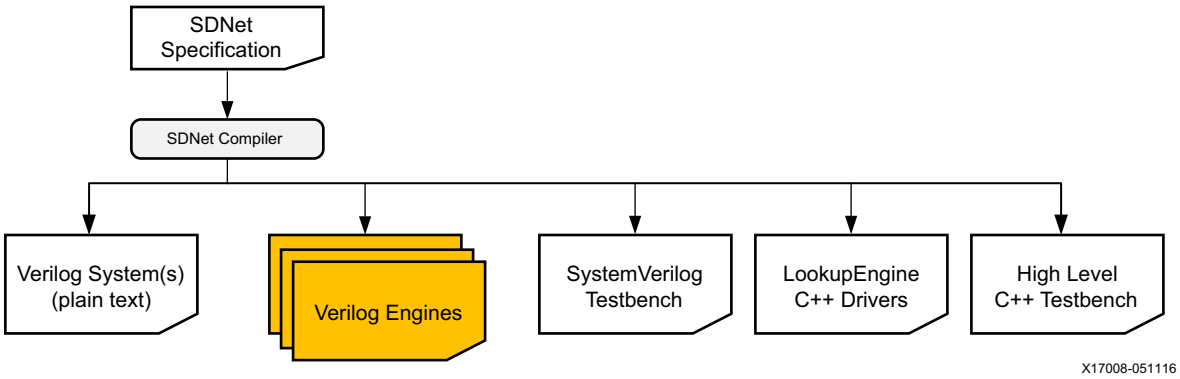


Figure 4-1: **Compiler Input and Output Files**

Requirements

- Java v1.8

Input File Types

SDNet Specification

The only required input file is the SDNet specification of the system and engines, which is provided as an input to the SDNet compiler.

Input Packet File (Optional)

The ***Packet.user*** input packet file is an optional text file containing the stimulus for the input packet data interface in a bus-protocol-independent text format or a Wireshark PCAP binary format.

Text Format

The text format has the following structure:

- Lines beginning with a percent character (%) are considered line comments.
- Lines contain packet data in hex format and optional whitespace to separate bytes or words. Note that the number of hex digits in a packet must be even as each packet should be an integral number of bytes.
- The packet's boundary is signaled with a semicolon (;) character at the end of the packet. It can be immediately following the last byte or on a separate line.

The following is an example of packets in text format:

```
% Packet 1 (67 bytes)
% Ethernet header:[ DstMAC=b1492b9e6979 SrcMAC=89c99c809244
EtherType=86dd ]
b1 49 2b 9e 69 79 89 c9 9c 80 92 44 86 dd
% IPv6 header:[ Version=6 TrafficClass=ec FlowLabel=d4d03
PayloadLen=0009
% NextHeader=11 HopLimit=92
% SrcAddr=dd873450bbabca5591cf02cd37d22c91
% DstAddr=ca90c0d2a2d1fe5fcde91eccf2613a14 ]
6e cd 4d 03 00 09 11 92 dd 87 34 50 bb ab ca 55 91 cf 02 cd 37 d2 2c
1 ca 90 c0 d2 a2 d1 fe 5f cd e9 1e cc f2 61 3a 14
% UDP header:[ SrcPort=9949 DstPort=f74b Length=0009
Checksum=b617 ] 99 49 f7 4b 00 09 b6 17
% Payload
c0
% Ethernet trailer:[ FCS=24f452c5 ]
24 f4 52 c5
;
```

PCAP Format

Refer to the Wireshark documentation for a description of the PCAP file format at:

http://wiki.wireshark.org/Development/LibpcapFileFormat

Input Tuple File (Optional)

The ***Tuple.user*** input tuple file is an optional text file that contains the stimulus for the input tuple interface in a bus-protocol-independent text format. The text format has the following structure:

- Each line corresponds to a packet in the input Packet file. Line 1 -> packet 1, Line 2 -> packet 2, etc.
- Each line contains the values of all "input" and "inout" user tuples in the order of definition. The values are expressed as contiguous digits in hex format with whitespace separators between the fields.

The following is an example of the tuple file in text format. The two columns are user-defined tuples of different widths in the order of their definition in the SDNet specification.

```
11C94BF511DD4D60CBF402C37EA943 B958
119B8533EC288760EE8A96656DBE3D 2A43
06546EE34FAAE874535580717EBD72 8DD6
0658E062000000000000000000000 FFFF
065C80618600000000000000000000 FFFF
00000000000000000000000000000 0000
```

Input Parameters

Command Line Options

The format of the compiler command is:

```
$ sdnet [options] <SDNet specification filename>
```

where *options* specifies a set of optional parameters described in Table 4-1.

Table 4-1: **Optional Parameters**

Parameter	Description
-help	Prints a list of possible parameters and a short description.
-busWidth *w*	Specifies the bus width to be w bits wide. Width must be a power of two. Supported values must be in the range from 16 to 1024.
-busType *type*	Specifies the bus type as either: • lbus: Xilinx® Local Bus used by Interlaken cores and other networking IPs • axi: AMBA Advanced eXtensible Interface (AXI) Stream Interface type
-workDir *dir*	Specifies a relative or an absolute path to the output directory for SDNET design and testbench files. The directory is created if it does not exit. Default value if not specified is: "work".
-controlClock value	Control clock frequency (in MHz).
-lineClock value	Line rate clock frequency (in MHz).
-lookupClock value	"Lookup"/once-per-packet rate clock frequency (in MHz).
Example: *Computing the requirements for the default packet clock rate*: A minimum sized frame of 64 bytes with target throughput of 100 Gb/s, with 512-bit wide datapath, requires 2 cycles because of the framing overhead. •The default *lineClock* is 300 MHz. •The default *lookupClock* is 150 MHz because a minimum size frame requires 2 cycles of spacing. •The default *controlClock* is 100 MHz. The clock rate information is necessary to correctly determine the sizing for the buffers contained by the sync blocks; used for synchronization and backpressure handling purposes.	

Table 4-1: **Optional Parameters *(Cont'd)***

Parameter	Description
-ingressSync	Forces the insertion of a sync block at the ingress to the system, for cases when the packet and input tuples are not already synchronized outside the system by the user. This synchronizes the tuple with the start of packets before feeding into the first engine of the system.
-inputTupleLookupClock	Specifies that input tuples at the ingress to the top level system use *clk_lookup* instead. (The default is for any input tuples to a system to use *clk_line*.)
-skipEval	Does not compile or run the high level C++ model, also called "eval", with the system RTL testbench.
-singleControlPort	Optionally adds the controller decoder block for the purpose of having a single AXI4Lite slave control interface. Alternatively, an AXI4Lite interconnect block from Vivado® Design Suite can be instantiated manually to join the control interfaces into a single slave interface. The recommended address map is generated in `<SystemName>.h`.
-noProtocolAdapters	Generates the top level system without packet protocol adapter blocks, if necessary.
-noBpSupport	Removes backpressure support and reduces the size of the sync block buffers. The amount of reduction in buffer sizes can be observed by comparing the generated `.info` file with a system built without this option.
-noXpm	XPM FIFOs are supported in recent versions of Vivado. To revert to the original version of the FIFO specify this option.
-packetFile *filename*	Packet stimulus filename (in PCAP or text format). The format of the packet text file was described earlier in this chapter.
-tupleFile *filename*	Tuple stimulus filename (in text format). The tuple data is used for the testbench stimulus and output tuple interface checking. The format of the tuple text file was described earlier in this chapter.
-EPE "-help"	Prints legacy options for editing and parsing engines and for stand-alone engines. These should not be used when building systems.
-LE "engine1 -clk_line"	Use the line clock instead of the packet clock for engine named engine 1. The packet clock is the default clock for lookup engines.
-LE "engine1 -agingBits *x*"	For EM, this specified the number of bits for aging out entries (default=0). Because the C++ model is untimed, this feature disables the generation of the C++ model.
-LE "engine1 -latency *x*"	Specifies the worst case latency for external lookup engines. In this case engine1 is specified to have worst case latency of x cycles.
-LE "engine1 -seed *x*"	For EM, this specifies the seed parameter to use for the internal hash function.
-LE "engine1 -shadow"	For LPM, this specifies to add a shadow copy of the table for supporting dynamic updates.
-LE "engine1 -tcam_bram"	For TCAM, this targets BRAM instead of LUTRAM.
-LE "engine1 -backpressure"	Specify for an "externally connected" LookupEngine's interface that needs backpressure support.
-LE "engine1 -em_distributed"	For EM, this sets use of distributed LUTRAM instead of block RAM.
-LE "engine1 -em_hashes *n*"	For EM, this sets n-way hash number of concurrent lookups.
-UE "-help"	Prints a list of options for user engines.

Table 4-1: **Optional Parameters *(Cont'd)***

Parameter	Description
-UE "engine1 -clk_line"	Use the line clock instead of the packet clock for engine named engine 1. The packet clock is the default clock for user engines.
-UE "engine1 -cuts"	This option specifies that the engine should cut all nearby tuple connections in the graph. This option is only valid when the user engine contains input and output packer ports.

Dataflow Graph Visualizations

Requirements

- Graphviz DOT Graph Visualization Software Library (www.graphviz.org)
  - Render the DOT visualizations using the generated script `./run_dot.bash`.

SDNet Level Dataflow Graph

Nodes represent engines and edges represent dataflow between engines. Figure 4-2 shows a simple example system. The connections in the graph have the following color coding according to port types:

- Blue = packet
- Green = tuple
- Red = backpressure
- Gray = memory-mapped control / access
- Pink = "plain" wires

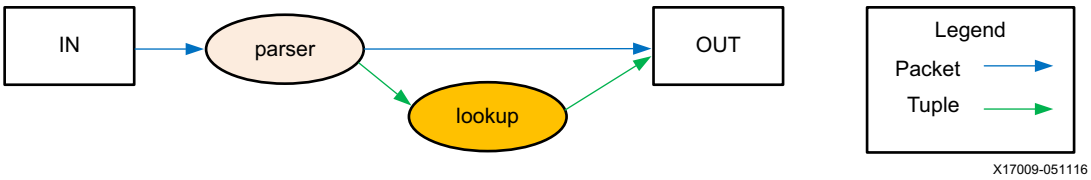


Figure 4-2: **Simple SDNet Level Dataflow Graph Example**

A more complex SDNet dataflow graph example is shown in Figure 4-3. Engines are also color coded based on type:

- Cream = parsing engine
- Dark yellow = lookup engine
- Dark orange = editing engine

- Green = tuple engine
- Salmon = user engine
- Gray = infrastructure block, from elaboration

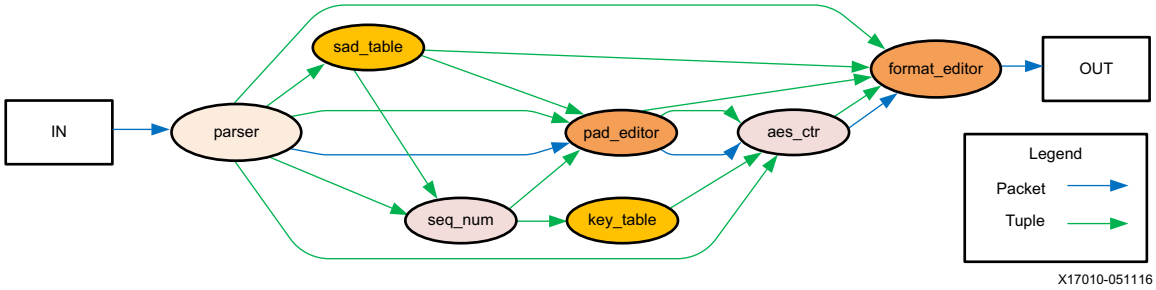


Figure 4-3: **More Complex SDNet Dataflow Graph Example**

Elaborated/RTL Dataflow graph

Figure 4-4 shows the simple example shown in Figure 4-2 elaborated with default compiler settings. Figure 4-5 shows a more complex elaborated/RTL dataflow graph example, from the same system shown in Figure 4-3. Bridges, syncs, and protocol adapters are included in the elaborated views. Connections use the preceding color coding.

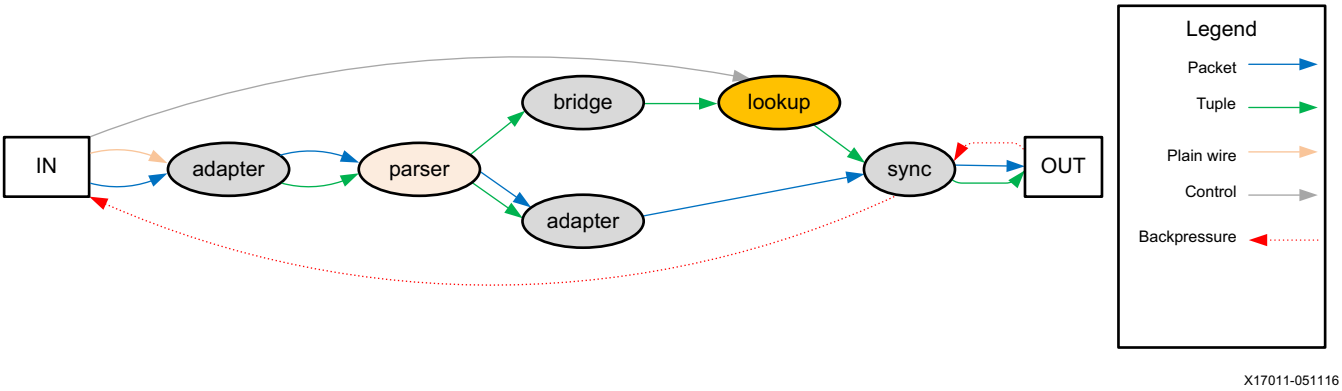


Figure 4-4: **Simple Elaborated/RTL Dataflow Graph Example**

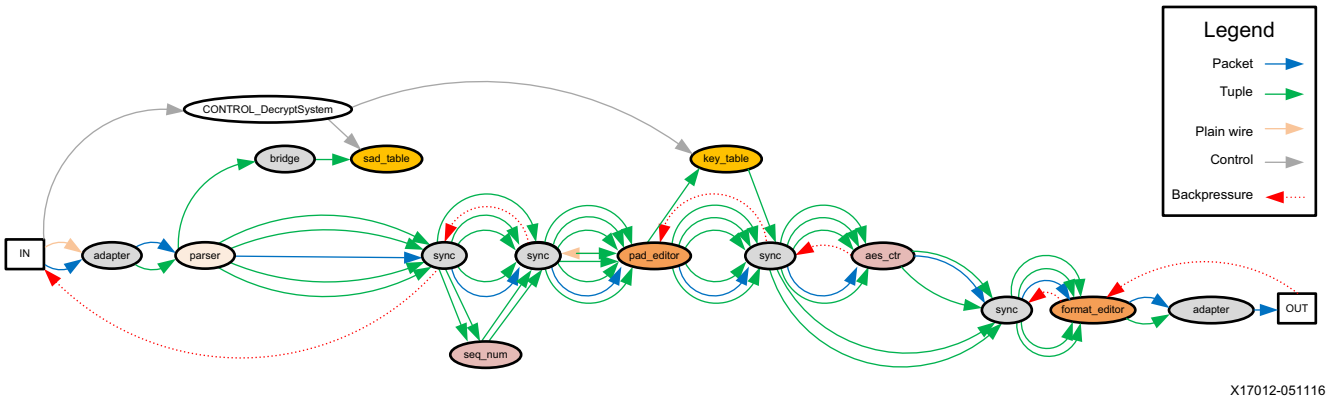


Figure 4-5: **More Complex Elaborated/RTL Dataflow Graph Example**

Output RTL Files and Testbench

Encrypted Verilog for each engine is placed into the directory:

```
<work directory>/<top-level system name>/<engine name>.HDL/
```

Files for the high-level test bench for each engine are placed into the directory:

```
<work directory>/<top-level system name>/<engine name>.TB/
```

Other files are defined as follows:

- Scripts
  - vivado_sim.bash — Vivado simulator 2017.1 compilation and simulation script
  - vivado_sim_waveform.bash — This is the similar to vivado_sim.bash but runs the Vivado GUI with the waveform view
  - questa.bash: — Questa Advanced Simulator (or ModelSim) compilation and simulation script
  - run_dot.bash: — Dot visualization generation script
  - compile.bash: — Compilation script for stand-alone execution of the high level C++ system model
- Testbench Stimulus
  - Packet.user: — Copy of the user-supplied packet stimulus file (if provided)
  - Tuple.user: — Copy of the user-supplied tuple stimulus file (if provided)
  - *.tbl, *.dat: — Files for the contents of the lookup engines should be manually copied into this directory

- Testbench Files
  - <system name>_tb.sv: Top-level SystemVerilog testbench
  - TB_System_Stim.v: Stimulus generator module
  - Check.v: Output checker module
  - user.c: Files for the contents of the lookup engines should be manually copied into this directory
- Miscellaneous files
  - *.info: Reports the command line arguments used to run the compiler as well as the buffer sizes that were determined for generating sync blocks

  An example `.info` file follows:

```
SDNet compiler arguments:
   p4demo.sdnet -workDir mydemo1 -busType axi -packetFile Packet.user


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

Sync 0: S_SYNCER_for_ingress_9
  Ports:
    packet_in_PACKET5, calc_depth: 146, actual_depth: 256,backpressure_thresh: 81
    tuple_in_TUPLE0,calc_depth: 49,actual_depth: 64,backpressure_thresh: 20
    tuple_in_TUPLE1,calc_depth: 73,actual_depth: 128,backpressure_thresh: 41
    tuple_in_TUPLE2,calc_depth: 49,actual_depth: 64,backpressure_thresh: 20
    tuple_in_TUPLE3,calc_depth: 73,actual_depth: 128,backpressure_thresh: 41
    tuple_in_TUPLE4,calc_depth: 73,actual_depth: 128,backpressure_thresh: 41


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

Sync 1: S_SYNCER_for_deparser_6
  Ports:
    packet_in_PACKET3, calc_depth: 147, actual_depth: 256,backpressure_thresh: 74
    tuple_in_TUPLE0,calc_depth: 74,actual_depth: 128,backpressure_thresh: 37
    tuple_in_TUPLE1,calc_depth: 82,actual_depth: 128,backpressure_thresh: 37
    tuple_in_TUPLE2,calc_depth: 74,actual_depth: 128,backpressure_thresh: 37
```

In this example `.info` file, two sync blocks were generated. The calculated depth is rounded up to form the actual depth, because the FIFO size must be a power of two. Note that sync block locations are shown in the elaborated dataflow visualization. The FIFO size can be slightly reduced if the system is compiled using the *-noBpSupport* option, which is described in the list of compiler options.

Simulation

Requirements

- Vivado® 2017.1 Simulator (xsim) can be used as an alternative simulator to Questa Advanced Simulator
- Questa Advanced Simulator v10.4c or later (or ModelSim v10.4c or later)
- gcc v6.2.0

Lookup Table Contents

TBL formats

Lines with "#" are treated as comments.

EM input file format

```
<key(hex)> <value(hex)>
```

TCAM input file format

```
<address(dec)> <key(hex)> <mask(hex)> <value(hex)>
```

LPM input file format

```
<prefix_string> <length(dec)> <value(dec)>
```

- Prefix is either in dot or colon separated format
- If it's in dot-separated format, each chunk is in 8-bit dec, e.g., `192.168.0.120`
- If it's in colon-separated format, each chunk is in 16-bit hex, e.g., `32f3:aef4:0000:0000:0000:0000:0000:0000`

- For LPM, the `.tbl` file must be converted to a `.dat` file using the mapping software prior to RTL simulation as described in Appendix D, LPM Mapping Software.

DIRECT input file format

```
<address(dec)> <value(hex)>
```

High level (C++ simulator)

1. Input packet file must be provided.
2. User-defined tuple file is optional. If the SDNET system has input tuple(s) and there is no input tuple file, then all of the input tuple(s) are set to zero (0).
3. If the system contains lookup engine(s), then the input lookup table file(s) are required. These files MUST be named the same with the lookup engine name found in the SDNET system class declaration, with `.tbl` file extension. The following is a sample system class:

```
class MySystem :: System {
  Packet_input    packet_in;
  Packet_output   packet_out;
  MyParserTuple   ParserTuple;
  MyEditedTuple   EditedTuple;
  MyLookup1       Lookup1;
  MyLookup2       Lookup2;
  MyParser        parser;
  MyEditor        editor;

  method connect = {
        parser.packet_in     = packet_in,
        parser.ParserTuple   = ParserTuple,
    editor.EditorTuple1  = parser.ParsedTuple,Lookup1.request      =
parser.ParsedTuple,
    Lookup2.request      = Lookup1.response,
    editor.EditorTuple2  = Lookup2.response,
    editor.packet_in     = parser.packet_out,
    EditedTuple          = editor.EditedTuple,
    packet_out           = editor.packet_out
  }
}
```

When simulating this example, the C++ model expects two input lookup table files: `Lookup1.tbl` and `Lookup2.tbl` for the lookup engines "Lookup1" and "Lookup2", respectively.

Execution

Run the stand-alone simulation cd into the `<systemName>.TB` directory:

1. Run the `compile.bash` script.

2. Step 1 produces an executable called "a.out", and you can rename this file if necessary.
3. Copy the `Packet.user`, `tuple stimulus`, and `*.tbl` files into this directory.
4. Run the new executable.

RTL level (System Verilog Testbench)

To run the RTL simulation using Questa Advanced Simulator (or ModelSim), first copy any necessary `*.tbl` or `*.dat` files into the main directory and run the `questa.bash` script. The simulation breaks if an error is detected or when all packets have been processed and the simulation results match the eval expected output. If the output packets and output tuples match the values that were expected by the eval model simulation, then the message "TEST_PASSED" appears in the Questa Advanced Simulator (or Modelsim) console.

Importing the Design into Vivado Tools

Requirements

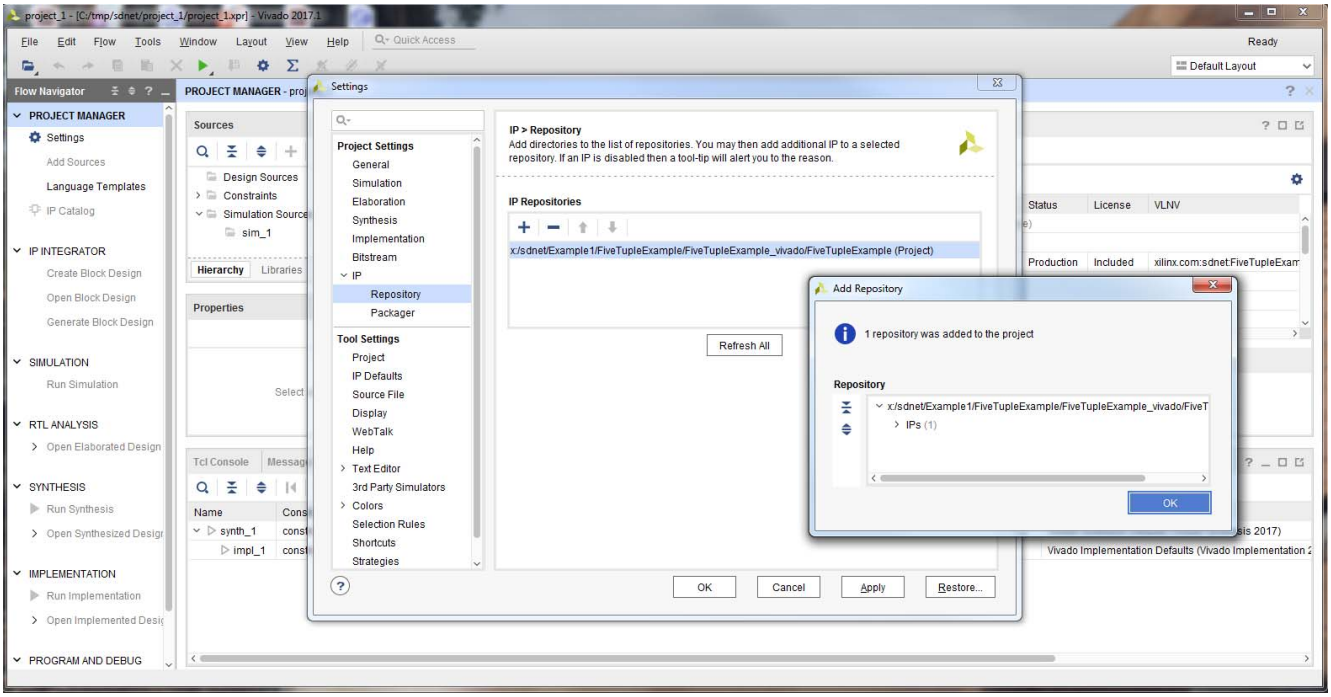
- Vivado® Design Suite 2015.3 or later

Packaging Script

Compiling an SDNet system produces a TCL script for Vivado tools for packaging the system as an IP core within the Vivado IP Catalog. The TCL file is generated by default in the same directory as the source files. To run the TCL script, Vivado version 2015.2 must be used.

1. CD into the work directory from Vivado (the work directory should contain the generated system files and `<systemName>_vivado_packager.tcl script`).
2. In the TCL console, enter `source MySystem_vivado_packager.tcl` (or substitute the name of your system).

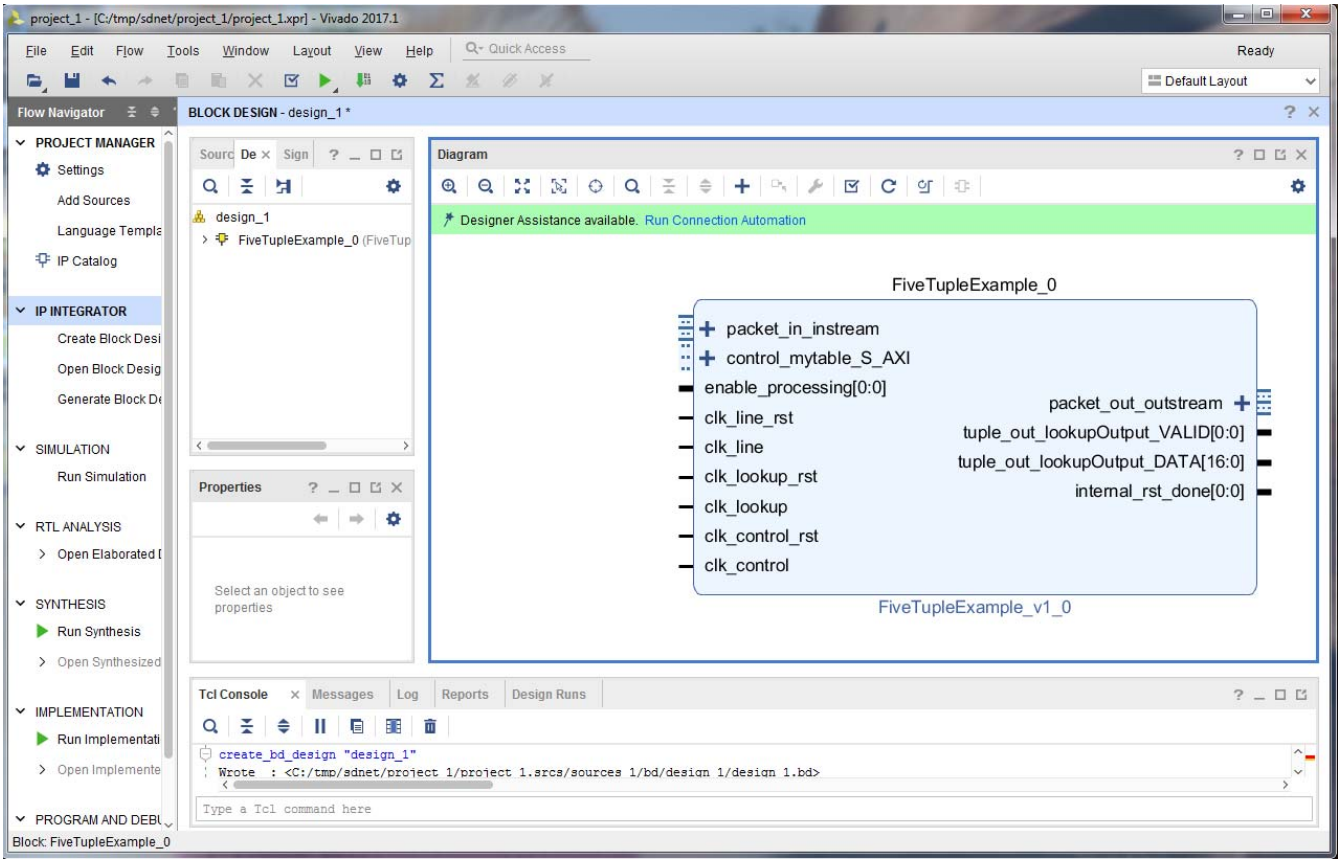
   This creates a Vivado project for the sources, runs the packaging wizard, and closes the newly created project. To use the IP, you must go to the project setting from within the "other" Vivado project (the user-created project in which you wish to instantiate the system), and add the project to the list of IP repositories under the "IP" icon/tab on the left (see Figure 6-1). The repository list is initially empty.



X17014-050917

Figure 6-1: **Configuring the Project Settings to Point to the Correct IP Repository Location**

3. After accessing "IP Catalog" from the panel on the left, you should see your SDNet system as an option under "UserIP" and also "Communications & Networking" (see Figure 6-2).



X17013-050917

Figure 6-2: **Example Instantiation of the MySystem**

a. Connect the input packet port (the associated clock for this port is clk_line).

b. Connect the output packet port (the associated clock for this port is clk_line).

c. Connect any input tuple ports (the associated clock is clk_line unless specified differently by a compiler option). If the compiler option -inputTupleLookupClock is specified, then any input tuples are assumed to be based on clk_lookup instead.

d. Connect any output tuple ports (the associated clock is clk_line).

e. Connect any AXI4Lite memory-mapped control ports (the associated clock is clk_control).

f. Connect the clk_line, clk_lookup, and clk_control ports.

g. Connect the clk_line_rst, clk_lookup_rst, and clk_control_rst ports.

h. Connecting "enable_processing" to `1'b1` is recommended for initial testing.

There are two signals for control and status on the system's RTL interface:

- internal_rst_done signals done when the internal reset has been asserted and internal engines are ready.
- enable_processing turns on and off processing for any new packets entering the system.

Note that the timing constraints for the SDNet packet processor are not currently automated by the compiler and must be added manually to the top level design by the user, e.g., using Vivado Design Suite's Edit Timing Constraints.

The timing constraints should set the clock periods for the clk_line, clk_lookup, and clk_control clock signals as necessary by the design. The default frequencies, unless specified at the command line, are 300 MHz, 150 MHz, and 100 MHz respectively.

It is safe and encouraged to add a false path on reset if there are timing failures along its path because the internal reset signal is held internally for relatively many cycles during assertion.

It is also safe and encouraged to add a false path constraint on paths crossing between clock domains if there are failures, for example between clk_lookup and clk_control.

The System Level Reference Design (SLRD) for SDNet can be used as an appropriate design example:

```
#==============================
# False paths to/from 170 MHz and SDNet Clock
#==============================
set_false_path -from [get_clocks clk_sdnet] -to [get_clocks clk_100]
set_false_path -from [get_clocks clk_100]   -to [get_clocks clk_sdnet]
```

User Engine Stub Files and High Level Simulation

The example user engine IV_GEN is described in the SDNet specification as:

```
class IV_GEN :: UserEngine(1, 8) {
    IndexTupleIn            index_tuple;
    InitVectorTupleOut      iv_tuple;
}
```

The two class parameters in this example specify: (a) that the worst-case latency of this engine will be one cycle, and (b) there is a configuration/control port with an 8-bit wide address.

Generated Stub Verilog File

The stub Verilog file generated by the compiler to implement the user engine by the user follows. For convenience, the relevant tuple formats from the SDNet specification are included in the stub Verilog file.

```
module IV_GEN (
     clk_line,
     clk_control,
     clk_line_rst_high,
     clk_control_rst_low,
     tuple_in_index_tuple_VALID,
     tuple_in_index_tuple_DATA,
     tuple_out_iv_tuple_VALID,
     tuple_out_iv_tuple_DATA,
     control_EN,
     control_RW,
     control_ADDR,
     control_DATAIN,
     control_DATAOUT,
     control_ACK
);

input clk_line /* unused */ ;
input clk_control /* unused */ ;
input clk_line_rst_high /* unused */ ;
input clk_control_rst_low /* unused */ ;
input tuple_in_index_tuple_VALID /* unused */ ;
input [5:0] tuple_in_index_tuple_DATA /* unused */ ;
output tuple_out_iv_tuple_VALID /* undriven */ ;
output [63:0] tuple_out_iv_tuple_DATA /* undriven */ ;
input control_EN /* unused */ ;
```

```
input control_RW /* unused */ ;
input [7:0] control_ADDR /* unused */ ;
input [31:0] control_DATAIN /* unused */ ;
output [31:0] control_DATAOUT /* undriven */ ;
output control_ACK /* undriven */ ;

wire tuple_out_iv_tuple_VALID /* undriven */ ;
wire [63:0] tuple_out_iv_tuple_DATA /* undriven */ ;
wire [31:0] control_DATAOUT /* undriven */ ;
wire control_ACK /* undriven */ ;


/* Tuple format for input: tuple_in_index_tuple
  [5:0]: index
*/

/* Tuple format for output: tuple_out_iv_tuple
  [63:0]: iv
*/
```

Generated Stub .hpp File

The header file stub leaves a place holder for the engine, for example:

```
  // TODO: ***************************
  // TODO: *** USER ENGINE MEMBERS ***
  // TODO: ***************************

  // engine ctor
  IV_GEN(std::string _n, std::string _filename = "") : _name(_n) {

       // TODO: *********************************
       // TODO: *** USER ENGINE INITIALIZATION ***
       // TODO: *********************************

  }
```

The user fills in the header file to provide the implementation for the user engine. For this example the implementation for the IV_GEN engine is provided as follows. In the C++ model the _LV class is used to represent bit vectors. The implementation of the _LV class is provided in `sdnet_lib.hpp`.

```
//########################################################
class IV_GEN { // UserEngine
public:
    // tuple types
    struct IndexTupleIn {
         _LV<6> index;
         IndexTupleIn& operator=(_LV<6> _x) {
              index = _x.slice(5,0);
              return *this;
         }
         operator _LV<6>() { return (index); }
         std::string to_string() const {
              return std::string("(\n")+"\t\tindex = "+index.to_string()+"\n"+"\t)";
         }
    };
```

```
    struct InitVectorTupleOut {
         _LV<64> iv;
         InitVectorTupleOut& operator=(_LV<64> _x) {
              iv = _x.slice(63,0);
              return *this;
         }
         operator _LV<64>() { return (iv); }
         std::string to_string() const {
                return std::string("(\n")   + "\t\tiv = " +iv.to_string()+"\n"+"\t)";
         }
    };

    // engine members
    std::string _name;
    IndexTupleIn index_tuple;
    InitVectorTupleOut iv_tuple;
    uint64_t    initial_value;

    // engine ctor
    IV_GEN(std::string _n, std::string _filename = "") : _name(_n) {
         initial_value = 1;
    }
    // engine function
      void operator()() {
         std::cout <<
"======================================================================" << std::endl;
         std::cout << "Entering engine " << _name << std::endl;
         // input and inout tuples:
         std::cout << "initial input and inout tuples:" << std::endl;
         std::cout << "index_tuple = " << index_tuple.to_string() << std::endl;
         // clear internal and output-only tuples:
         std::cout << "clear internal and output-only tuples" << std::endl;
         iv_tuple = 0;
         std::cout << "iv_tuple = " << iv_tuple.to_string() << std::endl;

         iv_tuple.iv = initial_value++;

         // inout and output tuples:
         std::cout << "final inout and output tuples:" << std::endl;
         std::cout << "iv_tuple = " << iv_tuple.to_string() << std::endl;
         std::cout << "Exiting engine " << _name << std::endl;
         std::cout <<
"======================================================================" << std::endl;
      }
};

//#####################################################
// top-level DPI function
extern "C" void IV_GEN_DPI(const char*, int, const char*, int, int, int);
```

The high level simulation walks along the dataflow graph, in topological order, and calls the "()" operator method for each engine. To make it easy for debugging, it is recommended that verbose print statements are added by the user when implementing the "()" operator method. In this example the user mainly sets the value for the "iv_tuple.iv".

Packet Buses

SDNet systems are generated with a common internal packet bus format called "PI". It is based on the LBUS standard, but it changes the MTY signal to CNT for a positive integer count of the valid bytes. The reason for the CNT signal was to have a normal way of signaling "empty packets" that occur when all of the data is removed by an editor. The other signals have the same timing and semantics as in LBUS.

This is an example Verilog file of an input packet port, with PI signaling:

```
output packet_in_RDY ;
input packet_in_VAL ;
input packet_in_SOF ;
input packet_in_EOF ;
input [511:0] packet_in_DAT ;
input [6:0] packet_in_CNT ;
input packet_in_ERR ;
```

The external AXI stream packet bus does not currently support the TUSER signal. It is currently recommended that the data carried by TUSER be converted to an input tuple and output tuple for passing the data through the SDNet system.

Lookup Engine Drivers

The SystemVerilog testbench also generates a user.c, which is the main source file for controlling and configuring any internal lookup engines. The driver source files are copied into the `<SystemName>/Testbench` directory. There is a separate source and header file for each of the engine types that is also copied into this directory.

The drivers were written assuming a 32-bit architecture would be used for configuring the lookup engines on a device, but the drivers can be modified to support a 64-bit architecture.

LPM Mapping Software

If any LPM lookup engines are part of the system then the contents of the lookup table needs to be preprocessed and mapped into physical addresses for the hardware simulation. In the `<SystemName>/Testbench` folder is a copy of `lpm_cfg.jar`. To run the LPM mapping software enter:

```
java -jar lpm_cfg.jar
```

This displays a list of supported options. An example of the command line parameters that are typically necessary is the following, with the number of levels being the $\log_2$ (table depth), (e.g., 8 levels for 255 entries):

```
java -jar lpm_cfg.jar -i lookupTableName.tbl -o lookupTableName.dat -k 32 -v 8 -l 8
```

SDNet Example System

An OpenFlow classifier example system is provided to get started:

```
class Packet_input  :: Packet(in) {}
class Packet_output :: Packet(out) {}

class Tuple_output  :: Tuple(out) {
    struct {
        output : 11
    }
}

class OF_classifier :: System {
    Packet_input instream;
    Packet_output outstream;
    Tuple_output flowstream;

    OF_parser parser;
    ACL lookup;

    method connect = {
        parser.packet_in = instream,
        lookup.request = parser.fields,
        outstream = parser.packet_out,
        flowstream = lookup.response
    }
}

struct of_header_fields {
    port : 3,  // Arrival port
    dmac : 48, // Ethernet
    smac : 48,
    type : 16,
    vid : 12,  // VLAN
    pcp : 3,
    sa : 32,   // IP
    da : 32,
    proto : 8,
    tos : 6,
    sp : 16,   // TCP or UDP
    dp : 16
}

class ACL :: LookupEngine(TCAM, 1024, 240, 11, 0) {
    class  request_tuple :: Tuple(in) {
        struct of_header_fields;
    }
    class response_tuple :: Tuple(out) {
```

```
            struct {
                flow : 11
            }
        }
        request_tuple request;
        response_tuple response;

        method send_request = {
            key = request
        }

        method receive_response = {
            response = value
        }
    }

    class OF_parser :: ParsingEngine (9416*8, 4, ETH_header) {

        // Interface object for an OpenFlow 1.0 12-tuple
        class Fields :: Tuple(out) {
            struct {
                port : 3,     // Arrival port
                dmac : 48,    // Ethernet
                smac : 48,
                type : 16,
                vid : 12,     // VLAN
                pcp : 3,
                sa : 32,      // IP
                da : 32,
                proto : 8,
                tos : 6,
                sp : 16,      // TCP or UDP
                dp : 16
            }
        }

        Fields fields;

        const VLAN_TYPE = 0x8100;
        const IP_TYPE = 0x0800;

        // Section sub-class for an Ethernet header
        class ETH_header :: Section(1) {
            struct {
                dmac : 48,
                smac : 48,
                type : 16
            }

            method update = {
          fields.dmac = dmac,
                fields.smac = smac,
                fields.type = type
            }

            method move_to_section =
          if (type == VLAN_TYPE)
                    VLAN_header
          else if (type == IP_TYPE)
```

```
                IP_header
            else
                done(1);
    }


// Section sub-class for a VLAN header
    class VLAN_header :: Section(2) {
        struct {
            pcp : 3,
            cfi : 1,
            vid : 12,
            tpid : 16
        }

        method update = {
            fields.vid = vid,
            fields.pcp = pcp
        }

        method move_to_section =
            if (tpid == 0x0800)
                IP_header
            else
                done(1);
    }

    const TCP_PROTO = 6;
    const UDP_PROTO = 17;

    // Section sub-class for an IP header
    class IP_header :: Section(2, 3) {
        struct {
            version : 4,
      hdr_len : 4,
      tos : 8,
      length : 16,
      id : 16,
      flags : 3,
      offset : 13,
      ttl : 8,
      proto : 8,
      hdr_chk : 16,
      sa : 32,
      da : 32
        }

        method update = {
            fields.sa = sa,
            fields.da = da,
            fields.proto = proto,
            fields.tos = tos
        }

        method move_to_section =
            if (proto == TCP_PROTO || proto == UDP_PROTO)
                TCP_UDP_header
            else
                done(1);
```

```
            method increment_offset = hdr_len * 32;
        }


        // Section sub-class for a TCP or UDP header
        class TCP_UDP_header :: Section(3, 4) {
            struct {
                sp : 16,
                dp : 16
            }

            method update = {
                fields.sp = sp,
                fields.dp = dp
            }

            method move_to_section = done(0);

            method increment_offset = 0;
        }
    }
```

A diagram of the system is shown in Figure 1-1.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado® IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

References

Contact Xilinx customer support for access to documents not yet available on the Xilinx website.

1. *P4-SDNet Translator User Guide* (UG1252)
2. *SDNet Functional Specification User Guide* (UG1016)
3. *Packet Processor SmartCORE API User Guide* (UG1013)
4. *SDNet Functional Specification Compiler Installation Guide* (UG1018)
5. *ARM® AMBA® AXI Protocol v2.0* (ARM IHI 0022D)
6. *SmartCAM Product Guide* (PG189)
7. *TCAM Product Guide* (PG190)
8. *LPM Product Guide* (PG191)
9. WireShark website (wireshark.org)
10. Graphviz DOT website (graphviz.org)
11. P4 Consortium website (p4.org)

Please Read: Important Legal Notices