P4-SDNet

User Guide

UG1252 (v2018.2) October 12, 2018

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/12/2018	2018.2	Added Table Values section.
04/27/2018	2018.1	Released with SDNet 2018.1 without changes from the previous version.
01/10/2018	2017.4	Updated command line options under Compiling with p4c-sdnet.
12/20/2017	2017.4	Changed document title from "*P4-SDNet Translator User Guide*" to "*P4-SDNet User Guide*". Replaced "Xilinx $P4_{16}$ Feature Support" section with "Xilinx $P4_{16}$ Language Support". Added "Xilinx $P4_{16}$ Extensions". Added $P4_{16}$ XilinxStreamSwitch to "Xilinx $P4_{16}$ Supported Architectures". Added "Appendix B: Sample Deparser Using packet_mod Extern".
10/27/2017	2017.3	Released with SDNet 2017.3 without changes from the previous version.
08/24/2017	2017.2	Updated the *--help* flag current tool information under Compiling with p4c-sdnet. Updated *@Xilinx_MaxPacketRegion()* default value from 8,192 to12,144 in Table 2. Changed *MaxPacketRegion* from 8192 to 1518*8, *ControlStruct* to *switch_metadata_t*, *DROP_PORT* to *0xF*, and *LPM* to *ternary* in Appendix A: Sample $P4_{16}$ Program for XilinxSwitch Architecture sample program.
05/15/2017	2017.1	Initial release

Table of Contents

Revision History 2

P4-SDNet User Guide

P4 and SDNet 4
Xilinx $P4_{16}$ Language Support 5
Xilinx $P4_{16}$ Extensions 8
Xilinx $P4_{16}$ Supported Architectures 10
Compiling $P4_{16}$ for SDNet 13
Appendix A:
Sample $P4_{16}$ Program for XilinxSwitch Architecture 15
Appendix B:
Sample Deparser Using packet_mod Extern 18
Xilinx Resources 19
Solution Centers 19
Documentation Navigator and Design Hubs 19
References 19
Please Read: Important Legal Notices 20

P4-SDNet User Guide

P4 and SDNet

P4 is an emergent language standard for describing programmable data planes that can target a wide range of technologies including CPUs, FPGAs, and NPUs. An article published in ACM SIGCOMM CCR in 2014, with authors from Stanford, Princeton, Intel, Microsoft, and Google, laid the foundation. Shortly thereafter, the P4 Language Consortium (P4.org [Ref 1]) was established, with Xilinx as a founding member. Currently the consortium has about 65 member companies and 15 universities world-wide participating to shape the development of the language.

The initial language specification now called $P4_{14}$ was released in early 2015. However, limitations were quickly identified and the community began exploring new features. The new $P4_{16}$ specification was released in May, 2017.

The Xilinx SDNet high-level design environment was created earlier to simplify the design of packet processing data planes that target FPGA hardware. Some of SDNet's design goals were very similar to those of the P4 language. However, SDNet places more emphasis on custom architectures.

SDNet allows programmers to build new data planes by explicitly specifying the dataflow graph of processing engines. SDNet processing engines have specialized behavior and include: ParsingEngines, LookupEngines, EditingEngines, TupleEngines, and UserEngines; each generated according to an application-specific processing. The basic types are similar to the top-level components found in P4 such as the parser and the control blocks. However, unlike P4, SDNet lets developers explicitly declare the architecture at the level of point-to-point connections (in P4, only control flow is specified and the architecture is abstracted away). To implement a P4 specification, the P4-SDNet compiler maps the control flow onto a custom data plane architecture of SDNet engines. This mapping chooses appropriate engine types and customizes each of them based on the P4-specified processing. More information about the SDNet design environment can be found in the *SDNet Packet Processor User Guide* (UG1012) [Ref 2].

Xilinx $P4_{16}$ Language Support

The P4 language is target-independent by design. The specification outlines most of the expected behavior, but to accommodate different target platforms some behavior is defined as architecture-specific (e.g. table properties, and extern objects). The expectation is that each target compiler back-end can define its own level of support in these specific cases.

This section outlines the language support currently provided by Xilinx's P4-SDNet compiler, which is called p4c-sdnet. More information about using the tool, and its libraries, is provided in Compiling $P4_{16}$ for SDNet, page 13.

Extern Objects

The $P4_{16}$ core library defines two extern objects (*packet_in* and *packet_out*) for handling packet data into and out of the P4 program. Both are supported by p4c-sdnet, with the following restrictions:

- The *packet_in* extern can only be used as a parser block argument.
- The *packet_out* extern can only be used as a control block argument.
- The length() method for *packet_in* is not supported.

The library also defines the Xilinx experimental extern object *packet_mod* for handling streaming packet data. More information on this is provided in Xilinx $P4_{16}$ Extensions, page 8.

Currently, no other extern objects are supported by p4c-sdnet, and users cannot define their own extern objects.

Extern Functions

The $P4_{16}$ core library does not define any extern functions. However, p4c-sdnet does allow users to define and use their own extern functions. For each call to an extern function, p4c-sdnet generates a unique SDNet UserEngine. You must provide the RTL source code, and the interface must match what is expected by the SDNet data plane. For each UserEngine, the SDNet compiler generates a subdirectory in its work directory. The interface definition can be found in a Verilog stub file (`engine_name.v.stub`) generated by the compiler.

Note: Multiple calls to the same extern function are possible. Because of this, each corresponding UserEngine has a unique name in the generated SDNet code. The `sdnet_info.json` file can be used to correlate P4 extern function calls with SDNet UserEngines (see SDNet Info File, page 14 for more information).

Before writing a UserEngine, you are encouraged to read the *SDNet Packet Processor User Guide* (UG1012) [Ref 2]. All extern functions should have a `@Xilinx_MaxLatency()` annotation. Otherwise, the data plane assumes a maximum latency of one cycle.

It is possible for extern functions to maintain state, but the state is local to each calling point (i.e. multiple calls to the same function result in multiple states in the data plane). When a corresponding UserEngine is stateful, be aware that sometimes it is possible for the engine to receive valid tuple inputs when the engine's operation is not wanted by the P4 program because of conditional program flows. To address this, p4c-sdnet includes a *stateful_valid* bit in every extern function UserEngine interface to indicate whether the engine should actually operate or should just pass tuples through. Extern UserEngines that are not stateful can safely ignore this bit.

Table Properties and Match Kinds

The $P4_{16}$ language specification defines standard and additional table properties. All of the standard properties (key, actions, default_action), and one additional property (size) are supported by p4c-sdnet with the following restrictions:

- All keys for a given table must have the same match type.
- A table can have at most one LPM key.
- The *default_action* is assigned at compile time, and cannot be changed at runtime.

The $P4_{16}$ core library defines three table match kinds (exact, ternary, and lpm). All of these match kinds are supported by p4c-sdnet. In addition, p4c-sdnet also supports a direct match kind. More information on this is provided in Xilinx $P4_{16}$ Extensions, page 8.

SDNet uses Xilinx IP blocks to implement its lookup tables. The match kind of a table determines restrictions on the ranges of table property values: key size, number of elements, and depth. Table 1 summarizes these restrictions, but more details can be found in product guides PG189 [Ref 3], PG190 [Ref 4], and PG191 [Ref 5].

Table 1: **Match Kind Restrictions**

Match Kind	Key Size (bits)	Element Size (bits)	Depth
exact	[12, 384]	[1, 256]	[1, 512K]
ternary	[1, 800]	[1, 400]	[1, 4K]
lpm	[8, 512]	[1, 512]	[7, 64K]
direct	[1, 16]	[1, 512]	[2, 64K]

If a key is too small for the given table, p4c-sdnet will automatically prepend the key with zeros and issue a warning. If a key is too large for a given table, p4c-sdnet will issue an error

Table Values

Special attention needs to be given to each value stored within a table. Each value in the table needs to be the concatenation of the *action_id* and the *action_data* rather than just a value.

The following is an example of a concatenated bitvector to be stored as the value in a table:

```
<action_id><action1_data><action2_data>
```

The *action_id* has length in bits equal to ceiling(lg(#actions+1)). Note that the data for all actions is stored in all entries – it is not like a C union where the different values are overlapped. In other words, the length in bits is equal to the total lengths of all the actions' parameters.

The *action_id* is determined from the list of actions, which is described within the p4c-sdnet info file. Generating this info file requires using the `--sdnet_info <info_filename>` argument. The following is an example command based on a provided example design:

```
"p4c-sdnet forward.p4 -o forward.px --sdnet_info forward_p4_info.json"
```

The action IDs are listed in the data structure for the table within the generated `forward_p4_info.json` file, for example:

```
"px_lookups" : [
    {
      "px_name" : "forwardIPv4",
      "p4_name" : "forwardIPv4",
      "px_class" : "LookupEngine",
      "px_type_name" : "forwardIPv4_t",
      "match_type" : "TCAM",
      "action_ids" : {
        "Forward.forwardPacket" : 1,
        "Forward.dropPacket" : 2
      },
```

The following are example entries that can be inserted into this table (`.tbl` format for TCAM):

```
1. A000102 FFFFFFF 11
2. A000103 FFFFFFF 12
3. 7F000002 FFFFFFF 16
4. 7F000003 FFFFFFF 17
5. C0A80B37 FFFFFFF 1A
6. C0A80BFF FFFFFFF 1B
```

In each of the above entries for the `ForwardIPv4` table, the hexadecimal value on the rightmost column is the concatenation of the `Forward.forwardPacket` *action_id* and a 4-bit output port.

Annotations

The P4 language strives to be agnostic from any underlying hardware architecture. This methodology requires p4c-sdnet always to assume worst-case parameters, and this will likely compromise overall performance. However, p4c-sdnet supports the custom P4 annotations listed in Table 2 to help improve performance.

Table 2: **Xilinx Specific P4 Annotations**

Annotation	Valid P4 Objects	Description
`@Xilinx_MaxPacketRegion()`	parser/control blocks	The maximum packet depth (in bits). *Default: 12,144*
`@Xilinx_MaxLatency()`	extern functions	The maximum latency (in cycles) for an extern function call. *Default: 1*
`@Xilinx_ControlWidth()`	extern functions	The width of the software control interface address space (in bits) for an extern function call. *Default: 0*
`@Xilinx_ExternallyConnected`	table objects	Exposes the table's request and response tuples at the top level of the design.

Other Unsupported Features

The following other features of the $P4_{16}$ language specification are currently not supported in p4c-sdnet:

- Variable length header fields
- Error types: error values, and the verify() function
- Calling the exit() method from inside an action statement

Xilinx $P4_{16}$ Extensions

The Xilinx $P4_{16}$ extensions can be found in the `xilinx_core.p4` library, which is in the same directory as the standard P4 library headers (`data/p4include/`). These library extensions are usable by the various Xilinx-defined architectures.

Direct Match Table

In addition to the three predefined *match_kind* identifiers, the p4c-sdnet compiler also supports a *direct* identifier.

The identifier definition can be found in the `p4include/xilinx_core.p4` library:

```
match_kind {
  direct
}
```

The *direct* identifier is usable inside a table's key property, exactly like the predefined types. A direct match table uses the key as an index into its entries. The table has 2^N entries where N is the key's size in bits. At runtime, a direct table cannot miss, and returns an undefined value if the entry is not initialized by the control plane before use.

Mutable Packets

CAUTION! *The mutable packet is an experimental feature. It offers direct access to packet headers during deparsing, and can help reduce resource use and latency in larger designs.*

The $P4_{16}$ language provides two built-in externs for manipulating packet data. The *packet_in* extern is used by a parser block to extract data from the packet, and the *packet_out* extern is used by a control block to insert data back into the packet. The current P4 paradigm for packet handling is that all headers are separated from the payload before being reinserted during deparsing.

The p4c-sdnet compiler supports a new, experimental extern for deparsing called *packet_mod*. This extern can be used directly to modify the original packet headers in situ without the need for them to be separated from the payload. It does this by maintaining a packet cursor that iterates over the original packet from beginning to end, similar to the cursor used by *packet_in*, but instead of only reading packet data it can only write packet data.

The *packet_mod* extern is defined in the `p4include/xilinx_core.p4` library with the following methods:

```
extern packet_mod {
  void update<H>(in H hdr);
  void update<H>(in H hdr, bit<32> mask);

  void extract<H>();
  void extract<H>(in bit<32> varFieldSizeInBits);

  void emit<D>(in D data);

  void advance<H>();
  void advance(in bit<32> sizeInBits);

  bit<32> length();
}
```

The update() method directly modifies a header value starting at the packet cursor, and then advances the cursor by the header's size. An optional 32-bit mask value can be used to select a subset of fields for update. Each bit of the mask represents a single field in the

header with the most significant bit corresponding to the first header field (in its declaration), and the least significant bit corresponding to the last header field. When calling the update() method without a mask, all header fields are updated. A current restriction is that headers can have a maximum of 32 fields in total, in other words, the mask can have at most, 32 bits.

The `extract()` method removes a header starting at the packet cursor. The header can contain one variable-length field, in which case an expression for its length must be passed to this method as a second argument.

The `emit()` method inserts data before the packet cursor. The argument can be a header, header stack, or union type. It can also be a struct containing only fields with such types.

The `advance()` method moves the packet cursor forward without modifying any packet data. The method can be called with a header type and no arguments, in which case the cursor will be advanced by the header's size in bits. The method can alternatively be called with a constant value or expression argument for the number of bits by which the cursor is to be advanced.

The `length()` method returns the remaining number of bits from the packet cursor to the end of the packet. This method is not currently supported by p4c-sdnet.

The *packet_mod* extern in p4c-sdnet is used from inside a parser block. This is in contrast to the *packet_out* extern, which is used inside a control block. A parser is used because the *packet_mod* extern traverses a packet beginning-to-end similar to the *packet_in* extern, making it more natural to describe the control flow with parser transition statements.

An experimental architecture including the *packet_mod* extern, called XilinxStreamSwitch, is supported in p4c-sdnet. A sample deparser for XilinxStreamSwitch using *packet_mod* can be found in Appendix B: Sample Deparser Using packet_mod Extern, page 18.

Xilinx $P4_{16}$ Supported Architectures

The $P4_{16}$ specification has architecture-language separation: this gives target providers the flexibility to describe the nature of P4-programmable components within their own packet forwarding architectures. This architecture description gives signatures for container holes that P4 developers can fill with their desired functionality. The containers are specified in a generic fashion to maintain P4's protocol independence. Xilinx's p4c-sdnet compiler currently supports three architectures that developers can target:

- XilinxSwitch
- XilinxStreamSwitch
- XilinxEngineOnly

XilinxSwitch Architecture Description

Designs targeting the XilinxSwitch architecture generate a pipeline with three customizable block containers. Figure 1 shows how the blocks are connected within the pipeline. The block arguments are defined generically, which allows developers the flexibility to define how much header and control data is passing though the switch.

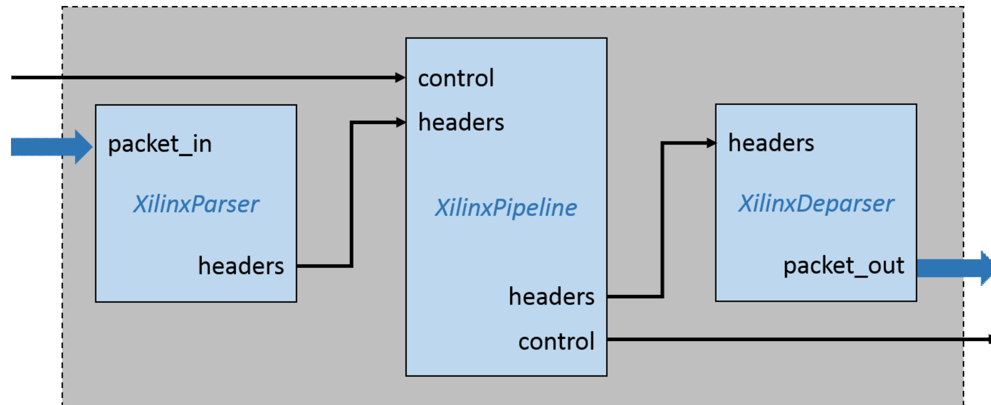

Figure 1: **XilinxSwitch Layout**

The first container (XilinxParser) is a parsing block that extracts headers from the packet. The next container (XilinxPipeline) is a control block that can be used to modify header and control data. The last container (XilinxDeparser) is another control block specifying the order headers that should be de-parsed back into the packet.

The P4$_{16}$ source code for the XilinxSwitch architecture description is as follows:

```
parser XilinxParser<H>(packet_in pkt, out H headers);
control XilinxPipeline<H,C>(inout H headers, inout C control);
control XilinxDeparser<H>(in H headers, packet_out pkt);

package XilinxSwitch<H,C>(XilinxParser<H> prsr,
                          XilinxPipeline<H,C> pipe,
                          XilinxDeparser<H> dprsr);
```

An example of P4-programmed components for this architecture is provided in Appendix A: Sample P4$_{16}$ Program for XilinxSwitch Architecture.

XilinxStreamSwitch Architecture Description

CAUTION! *The XilinxStreamingSwitch architecture is an experimental feature. It is provided for experimentation with the packet_mod extern.*

The XilinxStreamSwitch architecture is similar to the XilinxSwitch architecture. Both share the same parsing, and ingress pipeline, descriptions. However, the deparser description differs between the two architectures.

The $P4_{16}$ source code for the XilinxStreamSwitch architecture description is as follows:

```
parser XilinxParser<L>(
    packet_in packet, out L local);

control XilinxPipeline<L,C>(
    inout L local, inout C metadata);

parser XilinxStreamDeparser<L>(
    in L local, packet_mod packet);

package XilinxStreamSwitch<L,C>(
    XilinxParser<L>          user_defined_parser,
    XilinxPipeline<L,C>      user_defined_pipeline,
    XilinxStreamDeparser<L> user_defined_deparser);
```

XilinxEngineOnly Architecture Description

The XilinxEngineOnly architecture does not generate a complete forwarding pipeline. It can be used to generate stand-alone SDNet engines without any system building logic (i.e., the generated SDNet code will not specify multiple engines with connections). For generality, the architecture has no pre-defined P4 block containers or packages. Developers can define these components in accordance with their needs.

This architecture is useful for developers needing only a single block (e.g., only a packet parser), or advanced developers wishing to do their own SDNet system building. The following code outlines a $P4_{16}$ parser design that can be used to extract a 5-tuple from network packets. Compiling this design with p4c-sdnet generates a single SDNet ParsingEngine.

```
struct five_tuple_t {
     bit<8>        proto;
     bit<32>       ip_src;
     bit<32>       ip_dst;
     bit<16>       sport;
     bit<16>       dport; }

parser FiveTuple_t(packet_in p, inout five_tuple_t t);
package XilinxEngineOnly(FiveTuple_t container);

parser MyParser(packet_in pkt, inout five_tuple_t tup) {
     ...
}

XilinxEngineOnly(MyParser()) main;
```

When targeting the XilinxEngineOnly architecture the `--no_arch` flag is required.

Compiling $P4_{16}$ for SDNet

The P4 compiler is included with the SDNet environment. It can be used in conjunction with the SDNet tools to compile P4_16 code. Tools, libraries, and examples can be found in the SDNet install locations:

- Executable: `bin/p4c-sdnet`
- Libraries: `data/p4include/`
- Examples: `examples/p4examples/`

Operating System Support

The P4 compiler is currently only supported on 64-bit Linux operating systems. Although it might run on many distributions, the following are officially supported:

- Ubuntu
- CentOS

Compiling with p4c-sdnet

Use the following command to compile $P4_{16}$ source code at the Linux command prompt. The resulting output file can then be used as a source file within the SDNet design environment.

```
$ p4c-sdnet input.p4 -o output.sdnet
```

The following command line options, common to most P4 compilers, are supported:

Option	Description
`--help`	Print list of command line options
`--version`	Print compiler version
`-I path`	Specify include path (passed to preprocessor)
`-D arg=value`	Define macro (passed to preprocessor)
`-U arg`	Undefine macro (passed to preprocessor)
`-E`	Preprocess only, do not compile (prints program on stdout
`-o outfile`	Write output to outfile (if omitted, no output file is generated)
`--Wdisable [=diagnostic]`	Disable a compiler diagnostic, or disable all warnings if no diagnostic is specified
`--Wwarn [=diagnostic]`	Report a warning for a compiler diagnostic, or just treat all warnings as warnings if no diagnostic is specified
`--Werror [=diagnostic]`	Report an error for a compiler diagnostic, or treat all warnings as errors if no diagnostic is specified

In addition, the -v option can be used to get verbose reports when debugging or reporting issues to Xilinx. Multiple -v flags can be used together to increase the verbosity level of the output, e.g.:

```
$ p4c-sdnet -v input.p4
$ p4c-sdnet -v -v input.p4
```

There are three additional options specific to p4c-sdnet:

--no_arch	Generate only SDNet engines (i.e. no system)
--sdnet_info outfile	Write SDNet switch information to outfile
--toplevel_name name	Specify SDNet's top-level switch name (default is the switch architecture's name)

If multiple switches are generated within one design, each with a different name using the -toplevel option, then the -prefix command line option must be used when compiling each with SDNet to ensure disjoint name spaces.

SDNet Info File

When compiling from P4 for SDNet, certain P4 object and variable names can be transformed. This is necessary for various compiler optimization passes. While most of these changes are irrelevant to the user, some might be relevant when interfacing with the control plane or when using Vivado Synthesis. A JSON file is generated along with the SDNet source code if the --sdnet_info flag is passed to p4c-sdnet. The JSON file contains useful information about the compilation including, but not limited to:

- The original P4 name of SDNet objects (e.g. UserEngines, LookupEngines, etc.)
- LookupEngine request and response tuple fields
- UserEngine input and output tuple fields

Appendix A:
Sample P4$_{16}$ Program for XilinxSwitch Architecture

This is a sample P4$_{16}$ program for the XilinxSwitch Architecture. A copy can be found at: `examples/p4examples/forward.p4.`

```
typedef bit<48>     MacAddress;
typedef bit<32>     IPv4Address;
typedef bit<128>    IPv6Address;

header ethernet_h {
    MacAddress          dst;
    MacAddress          src;
    bit<16>             type; }

header ipv4_h {
    bit<4>              version;
    bit<4>              ihl;
    bit<8>              tos;
    bit<16>             len;
    bit<16>             id;
    bit<3>              flags;
    bit<13>             frag;
    bit<8>              ttl;
    bit<8>              proto;
    bit<16>             chksum;
    IPv4Address         src;
    IPv4Address         dst; }

header ipv6_h {
    bit<4>              version;
    bit<8>              tc;
    bit<20>             fl;
    bit<16>             plen;
    bit<8>              nh;
    bit<8>              hl;
    IPv6Address         src;
    IPv6Address         dst; }

header tcp_h {
    bit<16>             sport;
    bit<16>             dport;
    bit<32>             seq;
    bit<32>             ack;
    bit<4>              dataofs;
    bit<4>              reserved;
    bit<8>              flags;
    bit<16>             window;
    bit<16>             chksum;
    bit<16>             urgptr; }

header udp_h {
    bit<16>             sport;
    bit<16>             dport;
    bit<16>             len;
    bit<16>             chksum; }
```

```
struct headers_t {
    ethernet_h          ethernet;
    ipv4_h              ipv4;
    ipv6_h              ipv6;
    tcp_h               tcp;
    udp_h               udp; }

@Xilinx_MaxPacketRegion(1518*8)  // in bits
parser Parser(packet_in pkt, out headers_t hdr) {

    state start {
        pkt.extract(hdr.ethernet);
        transition select(hdr.ethernet.type) {
            0x0800  : parse_ipv4;
            0x86DD  : parse_ipv6;
            default : accept;
        }
    }
    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        transition select(hdr.ipv4.proto) {
            6       : parse_tcp;
            17      : parse_udp;
            default : accept;
        }
    }
    state parse_ipv6 {
        pkt.extract(hdr.ipv6);
        transition select(hdr.ipv6.nh) {
            6       : parse_tcp;
            17      : parse_udp;
            default : accept;
        }
    }
    state parse_tcp {
        pkt.extract(hdr.tcp);
        transition accept;
    }
    state parse_udp {
        pkt.extract(hdr.udp);
        transition accept;
    }
}


control Forward(inout headers_t hdr, inout switch_metadata_t ctrl) {
    action forwardPacket(switch_port_t value) {
        ctrl.egress_port = value;
    }
    action dropPacket() {
        ctrl.egress_port = 0xF;
    }

    table forwardIPv4 {
        key             = { hdr.ipv4.dst : ternary; }
        actions         = { forwardPacket; dropPacket; }
        size            = 63;
        default_action  = dropPacket;
    }
```

```
        table forwardIPv6 {
            key              = { hdr.ipv6.dst : exact; }
            actions          = { forwardPacket; dropPacket; }
            size             = 64;
            default_action   = dropPacket;
        }

        apply {
            if (hdr.ipv4.isValid())
                forwardIPv4.apply();
            else if (hdr.ipv6.isValid())
                forwardIPv6.apply();
            else
                dropPacket();
        }
    }


    @Xilinx_MaxPacketRegion(1518*8)  // in bits
    control Deparser(in headers_t hdr, packet_out pkt) {
        apply {
            pkt.emit(hdr.ethernet);
            pkt.emit(hdr.ipv4);
            pkt.emit(hdr.ipv6);
            pkt.emit(hdr.tcp);
            pkt.emit(hdr.udp);
        }
    }

    XilinxSwitch(Parser(), Forward(), Deparser()) main;
```

Appendix B:
Sample Deparser Using packet_mod Extern

This is a sample deparser targeting the XilinxStreamSwitch architecture, using the packet_mod extern:

```
@Xilinx_MaxPacketRegion(1518*8) /* in bits */
parser DeparserImpl(in headers_t hdr, packet_mod pkt)
{
  state start {
    /* skip Ethernet header */
    pkt.advance(112);
    transition select(hdr.ethernet.etherType) {
      0x0800  : deparse_ipv4;
      default : accept;
    }
  }
  state deparse_ipv4 {
    /* only update TTL, and checksum fields */
    pkt.update(hdr.ipv4, 0b0000000010100);
    transition select(hdr.ipv4.protocol) {
      6       : deparse_tcp;
      17      : deparse_udp;
      default : accept;
    }
  }
  state deparse_tcp {
    /* only update dport */
    pkt.update(hdr.tcp, 0b0100000000);
    transition accept;
    }
  state deparse_udp {
    /* only update dport */
    pkt.update(hdr.udp, 0b0100);
    transition accept;
    }
}
```

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

1. *P4 Language Consortium website* (http://p4.org)
2. *SDNet Packet Processor User Guide* (UG1012)
3. *Exact Match Binary CAM Search IP for SDNet SmartCORE IP Product Guide* (PG189)

4. *Ternary Content Addressable Memory (TCAM) Search IP for SDNet SmartCORE IP Product Guide* (PG190)
5. *Longest Prefix Match (LPM) Search IP for SDNet SmartCORE IP Product Guide* (PG191)

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

© Copyright 2017-2018 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.