



# **Vitis Networking P4 User Guide (UG1308)**

## **Introduction**

- History of P4 and Vitis Networking P4
- IP Facts
- How to Use this User Guide

## **Target Architecture**

- Standard Metadata
- Interfaces
- Engines
- Built-in Externs
- User Externs

## **Vitis Networking P4 Tool Flows**

- Command Line Interface (CLI) Software Flow
- Vivado Design Suite Hardware Flow
- Launch Simulation
- Run Synthesis and Implementation

## **Vitis Networking P4 Tool Interface**

- s\_axis Interface
- m\_axis Interface
- User Metadata Interface
- User Extern Interface
- Clocks
- Resets
- Back-Pressure
- Latency
- Register Map

## **Runtime Drivers**

- Overview
- Location
- Building
- Runtime Driver Examples
- Driver Memory Usage Estimate
- Porting to Platform

## **Examples of Basic Architectures**

### **Sample P416 Program**

### **Supported P416 Language Features**

### **Nulling Blocks**

## **P4 References and Guidelines**

- P4 Cheat Sheet
- Guidelines for Porting P4 Code to the AMD Architecture

P4 References

## **DPI Simulation**

Important Files

Interaction Between SystemVerilog and C

AXI Interfacing Tasks

Hierarchical Path to AXI Interface

Preparing for Use of Control Plane Drivers

Simulating with Multiple VNP4 Instances

Additional Utilities

Custom Control Plane Driver Usage

## **Resource Optimizations**

CAMS

P4 Coding Styles

## **Debugging**

Debug Tools

## **Additional Resources and Legal Notices**

Finding Additional Documentation

Support Resources

References

Revision History

Please Read: Important Legal Notices

# Introduction

## History of P4 and Vitis Networking P4

P4 is a language standard for describing programmable data planes that can target a wide range of technologies including CPUs, FPGAs, and NPU's. The initial language specification called P4<sub>14</sub> was released in early 2015. Limitations were quickly identified and the community began exploring new features. The new P4<sub>16</sub> specification was released in May, 2017. The AMD Vitis™ Networking P4 solution described here targets P4<sub>16</sub>.

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 (such as 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.

The Vitis Networking P4 (VNP4) high-level design environment has been created to simplify the design of packet-processing data planes that target FPGA hardware. VNP4 is a tool to convert the P4 design intent into an AMD FPGA design solution. VNP4 allows programmers to build new data planes by explicitly specifying the header and packet processing. VNP4 processing engines have specialized behavior and include: parsing engines, match-action engines, and deparsing engines, each generated according to an application-specific requirement. To implement a P4 design the compiler maps the control flow onto a custom data plane architecture of VNP4 engines. This mapping chooses appropriate engine types and customizes each of them based on the P4-specified processing.

Vitis Networking P4 features the following:

- Construction of hierarchical Vitis Networking P4 systems, consisting of a large variety of different types of engines including: Parsing, Deparsing, and Match-Action engines.

### **Parsing engines**

Extract header information from packets.

### **Deparsing engines**

Manipulate the contents of packet headers by inserting, modifying, or removing packet data.

### **Action engines**

Manipulate metadata that might be determined from packets or data originating externally or internally from some other engine.

### **Look-up engines**

Instantiate memory search IP cores generated from a library for packet processing including: exact match (BCAM), longest-prefix match (STCAM), ternary match (TCAM), and RAM (direct) tables.

- Support for 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 look-up.
  - Control rate which is the speed of the memory-mapped control interface for controlling and configuring engines.
- Systems resulting in high performance hardware implementations, achieving up to 200 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.
- Software available for high level system simulation of the design prior to running RTL simulations.
- AXI4-Stream signaling protocol 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 an AMD FPGA.

# IP Facts

Facts about AMD LogiCORE™ IP associated with AMD Vitis™ Networking P4 are as follows:

AMD LogiCORE™ IP Facts Table	
Core Specifics	
Supported Device Family <sup>1</sup>	AMD UltraScale™ , AMD UltraScale+™ , AMD Versal™
Supported User Interfaces	AXI4-Stream and AXI4-Lite Interfaces <sup>2</sup>
Provided with Core	
Design Files	Encrypted Verilog RTL
Example Design	Verilog
Test Bench	Verilog
Constraints File	Xilinx Design Constraint (XDC)
Simulation Model	P4BM C++ Behavioral Model
Supported S/W Driver <sup>3</sup>	Standalone
Software Example Design Application	Standalone, AMD Vivado™ IP integrator
Tested Design Flows <sup>4</sup>	
Design Entry <sup>5</sup>	Standalone, Vivado IP integrator
Simulation <sup>6</sup>	For supported simulators, see the <i>Vivado Design Suite User Guide: Release Notes, Installation, and Licensing</i> (UG973).
Synthesis	Vivado Synthesis

AMD LogiCORE™ IP Facts Table	
Support	
Release Notes and Known Issues	Master Answer Record: N/A
All Vivado IP Change Logs	Master Vivado IP Change Logs: <b>72775</b>
<b>Support web page</b>	
<ol style="list-style-type: none"><li>1. For a complete list of supported devices, see the Vivado IP catalog.</li><li>2. Standard protocols followed, refer to AMBA® AXI and AXI4-Stream Protocol Specifications.</li><li>3. Standalone driver details can be found in <a href="#">Runtime Drivers</a>.</li><li>4. For the supported versions of the tools, see the <i>Vivado Design Suite User Guide: Release Notes, Installation, and Licensing</i> (<b>UG973</b>).</li><li>5. The VNP4 IP is only supported in the Vivado IP catalog running on a Linux operating system (not supported on Windows).</li><li>6. Modelsim, Questa, VCS, Xcelium, and Xsim are supported. Refer to <i>Vivado Design Suite User Guide: Release Notes, Installation, and Licensing</i> (<b>UG973</b>) for information on version compatibility.</li></ol>	

## How to Use this User Guide

The rest of this user guide is organized as follows:

- [Target Architecture](#) describes the Vitis Networking P4 Architecture supported by the Vitis Networking P4 Tool.
- [Vitis Networking P4 Tool Flows](#) describes the top-level interface signals and clocking.
- [Vitis Networking P4 Tool Interface](#) describes the Software and Hardware tool flows for Vitis Networking P4 designs.
- [Runtime Drivers](#) describes the driver software released with the Vitis Networking P4 tool.
- [Examples of Basic Architectures](#) illustrates examples of how Vitis Networking P4 can be used.
- [Sample P416 Program](#) provides a sample P4<sub>16</sub> program.
- [Supported P416 Language Features](#) identifies supported P4<sub>16</sub> language features.
- [Nulling Blocks](#) provides example nulling blocks.
- [P4 References and Guidelines](#) provides P4 references and guidelines.
- [DPI Simulation](#) describes DPI and its use in the example designs.
- [Resource Optimizations](#) provides information on optimizing resources.
- [Debugging](#) includes details about debugging tools.
- [Additional Resources and Legal Notices](#) provides information on AMD resources and document references.

## Target Architecture

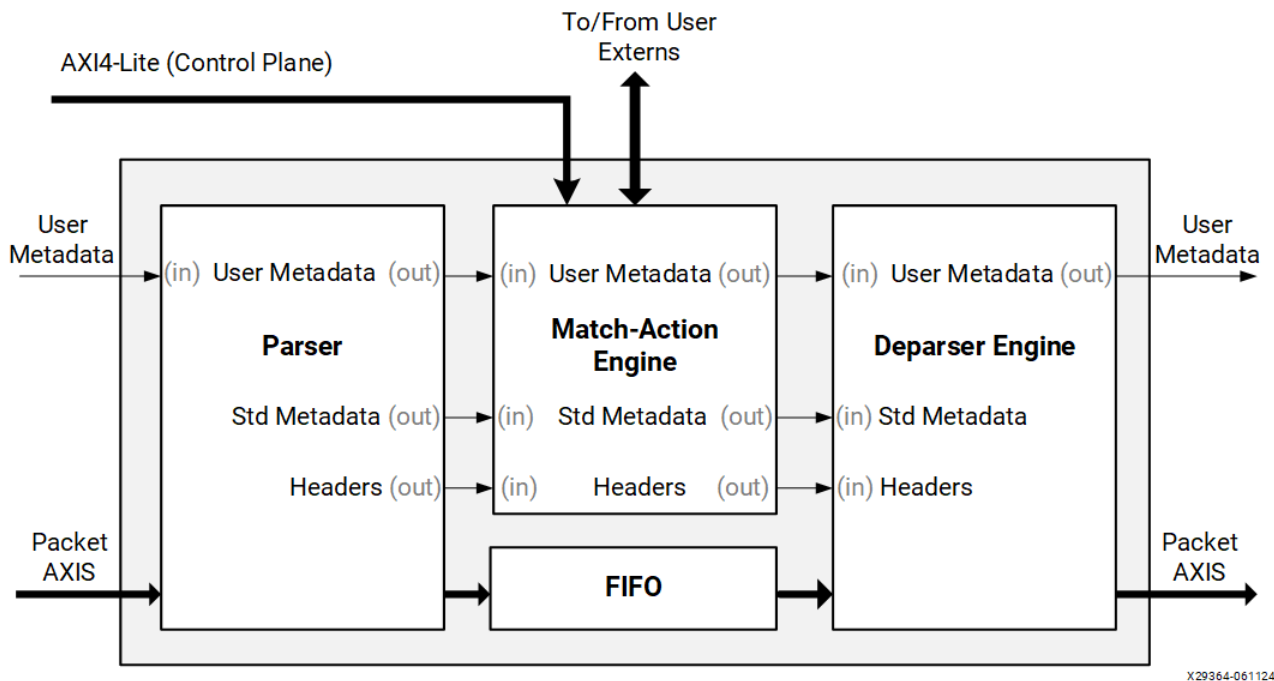
The P4 architecture defines the P4-programmable components and the data plane interfaces between them. The P4<sub>16</sub> specification has architecture-language separation: this gives target providers the flexibility to describe the nature of the P4-programmable components within their own packet processing 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. The AMD Vitis™ Networking P4 compiler supports the Vitis Networking P4 architecture that developers can target.

Designs targeting the Vitis Networking P4 architecture generate a pipeline with three customizable engines. The following figure shows how the engines are connected within the pipeline. The interfaces are defined



generically, which allows developers the flexibility to define how much header and control data is passing through the system.

**Figure: Top-Level Vitis Networking P4 Design**



The first engine (the Parser) is a parsing block that extracts headers from the packet. The next engine (the Match-Action), is a control block that can be used to modify header and control data. The last engine (the Deparser) is another control block specifying the order that headers should be deparsed back into the packet.

User-defined metadata ports are provided at the input and output to/from the Vitis Networking P4 design. These ports are user-defined structures associated with each packet based on the definition from the user's P4 program. Standard metadata generated during the execution of the P4 program is also available to the user - this metadata is provided by the architecture associated with each packet.

User extern ports are provided in the Vitis Networking P4 design, for user externs created in the user's P4 program. User externs themselves are instantiated outside the Vitis Networking P4 instance.

AXI4-Lite memory-mapped ports are used for configuration and control of the Look-up engines and built-in externs.

An example of P4-programmed components for this architecture is provided in [Nulling Blocks: Sample P4<sub>16</sub> Program](#).

If multiple consecutive functions are required (more than one parsing

function for example), multiple serial instances of Vitis Networking P4 IPs are a supported configuration. Similarly, if parallel processing of packet streams is required, Vitis Networking P4 instances can be deployed in parallel.

Although the Vitis Networking P4 architecture requires all three blocks (Parser, Match-Action, and Deparser) to be present, it is possible to 'null' some blocks to focus on the blocks of interest - for example, parsing to create some metadata output without performing any other packet modifications. An example of nulling blocks is provided in [Nulling Blocks](#). The Vitis Networking P4 Architecture definition file, `xsa.p4`, can be found here `<Vivado_install_area>/data/ip/xilinx/vitis_net_p4_v1_0/include/p4/xsa.p4`.


## Standard Metadata

Standard metadata is defined by the architecture and contains the following fields:

### **drop**

This field can be used to drop a packet. The drop field defaults to a value of 0 when a new packet arrives. You can update this field in P4 at any stage in the Parser or Match-Action. At the input of the Deparser, the drop field is evaluated and the packet dropped if the drop field has a value of 1. In the case of a dropped packet, the user metadata can still optionally be output from Vitis Networking P4 (see [Metadata](#) for more information).

---

 **Note:** Because this drop field is evaluated at the input to the Deparser, the corresponding packets are dropped before any headers are emitted into the packet stream and therefore without any potential reduction on packet stream bandwidth.

---

### **ingress\_timestamp**

There is a 64-bit timestamp counter in Vitis Networking P4 that increments every clock cycle from startup. For each new packet that arrives at the Parser input, this standard metadata field is populated with the timestamp counter value. This timestamp value can then be used at any stage in the Parser or Match-Action of the P4 code.


### **parsed\_bytes**

This field is automatically updated throughout the Parser so that it can be assigned to user metadata field(s) at various points in the Parser, to get several different offset values. The field at the end of the Parser provides the number of header bytes were successfully extracted during the Parser execution. It is equivalent to the “nextBitIndex” pointer referred to in the P4 Language Specification (see [Resource Optimizations](#)), but the value is given in bytes rather than bits. It is also equivalent to the byte offset where the packet payload begins following the packet headers.

### parser\_error

This field contains a parsing error code. This value is automatically updated by the Parser to be used in the Match-Action of the P4 code. The error codes are identified in the following table.

**Table: Parsing Error Codes**


Error Code	Type	Description
0	NoError	No error.
1	PacketTooShort	Not enough bits in packet for 'extract'.
2	NoMatch	'select' expression has no matches.
3	StackOutOfBounds	Reference to an invalid element of a header stack.
4	HeaderTooShort	Extracting too many bits into a varbit field.
5	ParserTimeout	Parser execution time limit exceeded.  <b>Note:</b> This error type is not currently used by the Vitis Networking P4 architecture.
6	HeaderDepthLimitExceeded	Extracting a header that exceeds the Header Depth Limit packet offset of 8191 bytes. In this case, the header is marked as invalid and the parser execution is terminated for that packet.
7-99	user defined	New errors can be defined by the user with the use of the 'error' structure. User-

Error Code	Type	Description
		defined errors can only be triggered with verify statements.

Whenever a verify statement is triggered, parsing terminates immediately and the `parser_error` field is updated with the error ID that was triggered. This field can be accessed in the Match-Action of the P4 code and multiple actions can be taken based on the error. For example, to drop the packet:

```
if (smeta.parser_error != error.NoError) {  
    smeta.drop = 1;  
    return;  
}
```

---

 **Note:** VNP4 treats the accept and reject parsing states in exactly the same way, without any architecture-specific distinction. Packets are not automatically dropped when the parser reaches the reject state. You can optionally drop packets or perform other actions in the Match-Action control block based on the `parser_error` metadata, as shown above.

---

## Interfaces

### Packets

Packet ports are the primary Vitis Networking P4 interfaces responsible for moving packets around between engines and also to the external world. Engines can only contain a single input packet port and a single output packet port. Parsing engines, deparsing engines, and systems require packet ports. User externs have the option of packet ports. The AXI4-Stream protocol is used for packet ports.

### Metadata

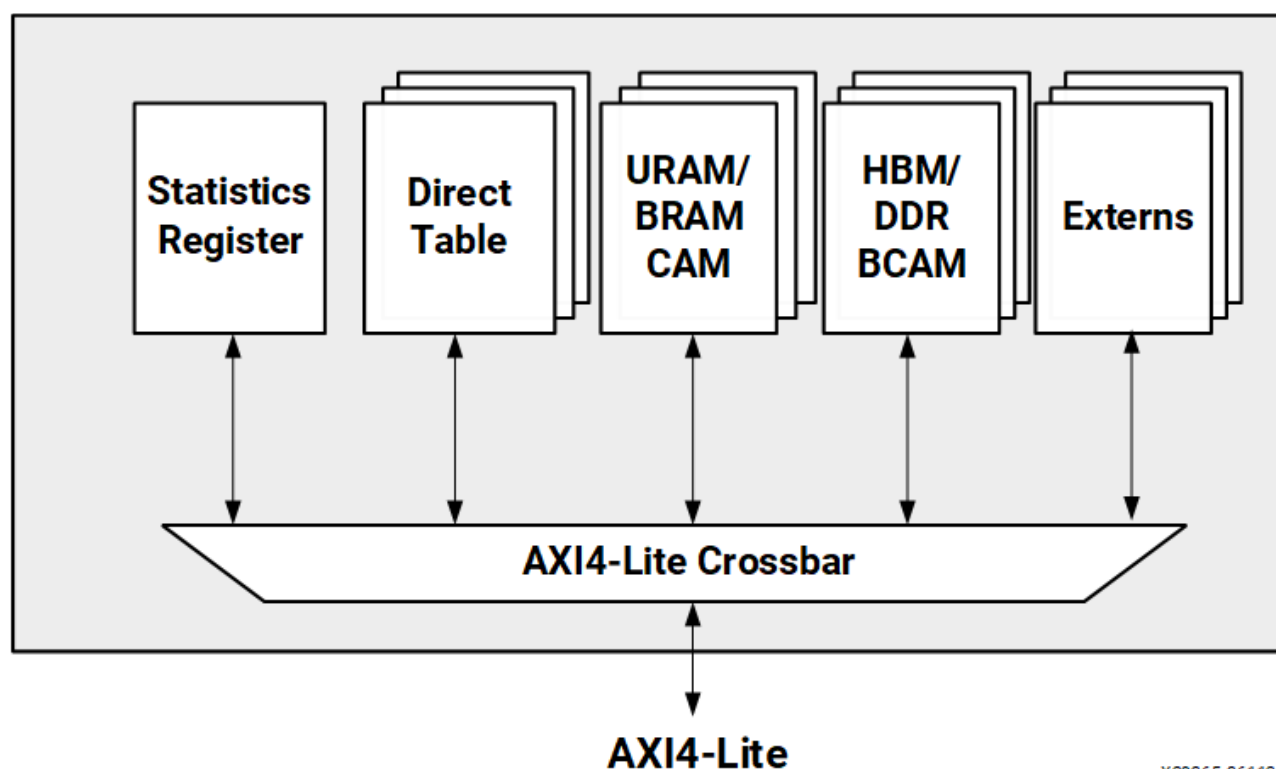
Metadata ports are the secondary Vitis Networking P4 interface responsible for communicating sideband packet-related data between engines and also to the external world. Metadata can only correspond to a single packet and is processed in parallel with the packet.

## AXI4-Lite

AXI4-Lite memory-mapped ports are used to control the contents of the Look-up engines and to configure, read, and write to externs. Each Vitis Networking P4 instance has a single AXI4-Lite slave interface. An AXI4-Lite crossbar is created within Vitis Networking P4 to connect all memory-mapped elements of the Vitis Networking P4 design for a given P4 program, which might include multiple tables (Direct Tables, URAM/BRAM CAMs, or HBM/DDR BCAMs) and/or multiple externs (for example, Counters). The address map grows by 8 kB for each table/extern in the P4 program.

The address width required on the AXI4-Lite slave interface depends on the P4 program (for example, the number of tables). The address width can be found in the generated instantiation template files (`<design_name>.veo/vho`) or in the generated `<design_name>_pkg.sv` file (`localparam S_AXI_ADDR_WIDTH`).

**Figure: AXI4-Lite Slave Interface and AXI4-Lite Crossbar**



## User Externs

User extern ports provide an interface between the Match-Action control

block and a user's own module that resides outside of Vitis Networking P4. There can be multiple user extern modules connected to this interface.

## 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. Parsers only extract the required fields defined by the user/architecture. They can transmit the data as output metadata. Parsers support a wide variety of header types and sequences. These include:

- Fixed and variable header sequences
- Fixed and variable length fields
- Bit and byte granularity

Parsers support these interfaces:

- Packet data input and output
- Standard metadata out
- Header data out
- Used-defined metadata in/out

Parsers do not support:

- Varbit fields in user metadata
- Greater than 8 KB headers (there is an 8 KB header limit in the Parser)

An example of a simple parsing engine is provided in [Sample P416 Program](#).

### Deparsing Engine

Deparsing engines (Deparsers) 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.

Deparsers typically have metadata input containing data to be written into packets.

Deparsers manipulate packet data contents. This includes:

- Header field modification
- Header removal
- Header insertion

Deparsers support the following interfaces:

- Packet data input and output
- Standard metadata out
- User-defined metadata in

Deparsers only modify packet data and do not read nor parse it.

Deparsers do not support the following features:

- Implicit re-ordering of headers, by re-ordering the headers as they are emitted in the Deparsers section of the P4 program
- Implicit duplication of headers, by multiple emit statements in the deparser section of the P4 program
- Conditional deparsing with 'if' statements

An example of a simple deparsing engine is provided in [Sample P416 Program](#).

## Match-Action Engine

Match-Action engines are a combination of Look-up engines and Action engines. Multiple actions are mapped, or activated, by the response of a Look-up engine. The Look-up engine response is used to enable only one action engine and store parameters required by the action.

Match-Action engines perform the following sequence of operations:

- Constructing look-up keys from user defined metadata
- Table look-up using the constructed key and returning a response
- Choosing an action and executing it based on the above response
- Performing programmed computations and returning an output
- Interfacing to/from user externs
- Performing built-in extern functions

An example of a simple Match-Action engine is provided in [Sample P416 Program](#).

## Action Engine

Action engines can manipulate headers, metadata and internally defined match-action scalars and perform computations on the same. Action engines modify headers, metadata and internally defined match-action scalars and not packet data. Action engines are used in the following two scenarios:

- Implicitly: by table engines during match-action processing
- Explicitly: during standalone header, metadata and internally defined match-action scalar manipulation

## Look-Up Engine

Look-up engines (look-up tables) are used to implement a search over a variety of different kinds of direct addressable and content addressable tables. Vitis Networking P4 includes a library of different Look-up engine types, which can be one of seven types:

- Binary content addressable memory - Exact match - EM (BCAM) based in URAM or block RAM
- High bandwidth memory/double data rate (HBM/DDR) - EM (BCAM)
- Semi ternary content addressable memory (STCAM) based in URAM or block RAM
- Ternary content addressable memory - ACL (access control lists) (TCAM) based in URAM or block RAM
- TinyBCAM
- TinyTCAM
- Direct table (DCAM)
- Extern Table

Each of these tables takes a user defined header input, metadata input and/or internal defined match-action scalars as a search key and its response is used to trigger its associated actions and to store defined action parameters. Note that Vitis Networking P4 only allows one look-up per packet for each table. These tables are described as follows.



## URAM/BRAM BCAM (EM)

EM (exact match) is a binary match of a search key to one of the keys stored in 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. This CAM uses internal URAM or BRAM to store data. There are limits for the number of URAMs and BRAMs that can be used at higher frequencies. See *Binary CAM Search LogiCORE IP Product Guide* (PG317), Tables 1 and 6 for parameter limits, and **NUM\_ENTRIES** in *Chapter 5, Design Flow Steps* for more information on these limits.

- Key width: 10 to 1024 bits
- Response width: 1 to 1024 bits
- Hit/miss flag: Yes

Here is an example of how to instantiate a BCAM in a P4 file:

```
table forwardIPv4 {
    key          = { hdr.ipv4.src : exact;
                    hdr.ipv4.dst : exact; }
    actions      = { forwardPacket;
                    dropPacket; }
    size         = 1024;
    default_action = dropPacket;
}
```

For more details on how the compiler selects a BCAM, see [Compiler Table Selection](#). See the *Binary CAM Search LogiCORE IP Product Guide* (PG317) for more information on BCAMs.

## HBM/DDR BCAM (EM)

HBM/DDR BCAM (EM - exact match) is a binary match of a search key to one of the keys stored within the table, inside the HBM/DDR memory. HBM/DDR allows for significantly larger tables to be supported compared to URAM/BRAM-based tables. The associated value for the key is returned within the response as well as an indication of whether the key produced a match. This CAM type is only supported on a Versal™ device. Multiple instances of HBM/DDR BCAM are supported, including HBM and DDR combined within the same P4 instance. The required RAM size for each

instance (RamSizeKBytes) can be found in the generated <design\_name>\_pkg.sv file (see [Generated Files](#)) and is also displayed on the GUI **Tables** tab under **Memory Resources**.

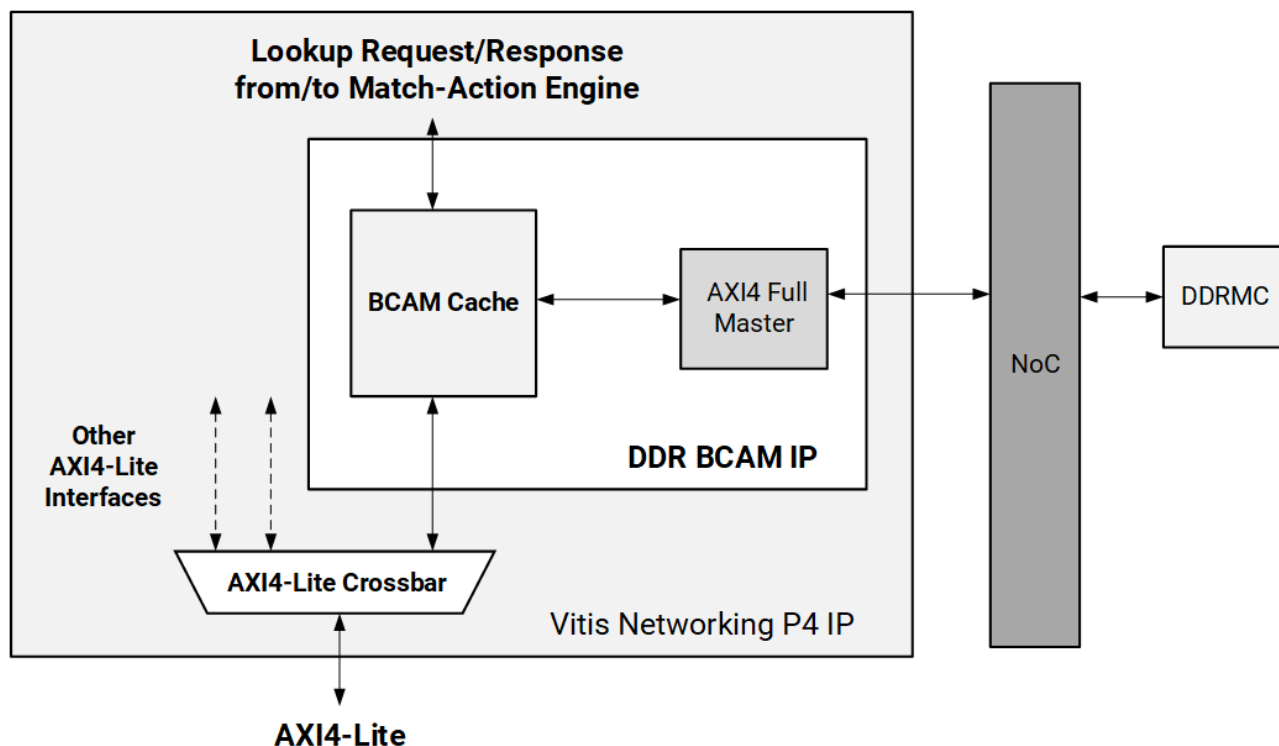
- Key width: 10 to 992 bits.
- Response width: 1 to 1006 bits.
- Look-up rate is dependent on many factors, including if a cache is enabled and what the cache hit rate is.
- Size: 8192 to 60 million entries.
- Hit/miss flag: Yes.

The following is an example of how to instantiate a HBM/DDR BCAM in a P4 file:

```
table forwardIPv4 {  
    key          = { hdr.ipv4.src : exact;  
                    hdr.ipv4.dst : exact; }  
    actions      = { forwardPacket;  
                    dropPacket; }  
    size         = 1048576;  
    default_action = dropPacket;  
}
```

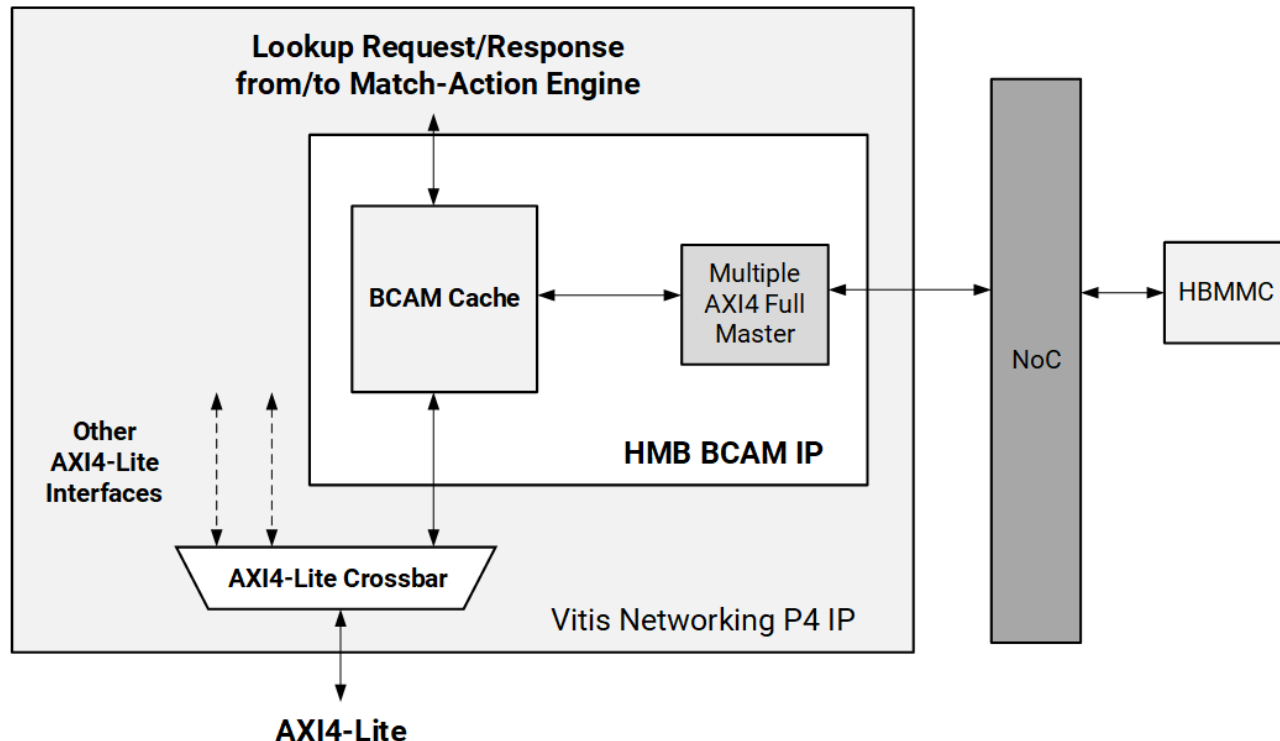
The following block diagram shows a Vitis Networking P4 IP with a DDR BCAM instance.

**Figure: Vitis Networking P4 IP with DDR BCAM**



The following block diagram shows a Vitis Networking P4 IP with a HBM BCAM instance.

**Figure: Vitis Networking P4 IP with HBM BCAM**



For more detailed information on how the compiler selects a BCAM, see [Compiler Table Selection](#). You must also select a 'High Bandwidth RAM' or 'DDR' style in the Vitis Networking P4 GUI (see [Example Design Use \(with HBM/DDR BCAM\)](#)). For more information on HBM/DDR BCAMs, see *Cached*

*DRAM Binary CAM LogiCORE IP Product Guide (PG427).*

## URAM/BRAM STCAM

Semi-ternary CAMs (STCAM) use 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. The STCAM is a restricted version of TCAM for the purpose of more efficient implementation in hardware. It is limited in the number of unique masks as specified by `num_masks`. This CAM uses internal URAM or BRAM to store data. There are limits for the number of URAMs and BRAMs that can be used at higher frequencies. See *Semi-Ternary CAM Search LogiCORE IP Product Guide (PG319)*, Tables 1 and 6 for parameter limits, and **NUM\_ENTRIES** in *Chapter 5, Design Flow Steps*, for more information on these limits.

- Key width: up to 1024 bits
- Response width: up to 1024 bits
- Hit/miss flag: Yes
- Number of masks: 2 to 64

Here is an example of how to instantiate an STCAM in a P4 file:

```
table forwardIPv6 {
    key          = { hdr.ipv6.dst : lpm; }
    actions      = { forwardPacket;
                    dropPacket; }
    size         = 1024;
    default_action = dropPacket;
}
```

For more details on how the compiler selects an STCAM, see [Compiler Table Selection](#). See *Semi-Ternary CAM Search LogiCORE IP Product Guide (PG319)* for more information on STCAMs.

## URAM/BRAM TCAM (ACL)

TCAM (ternary content addressable memory) is used to make 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. This CAM uses internal URAM or BRAM to store data. There are limits for the number of URAMs and BRAMs that can be used at higher frequencies. See *Ternary CAM Search LogiCORE IP Product Guide* ([PG318](#)), Tables 2 and 7 for parameter limits, and **NUM\_ENTRIES** in *Chapter 5, Design Flow Steps* for more information on these limits.

- Key width: up to 992 bits
  - Although the key width can be up to 992 bits, there are further restrictions on the ternary match kind fields. The use of the ternary match kind greatly increases the 'Complexity' calculation and thus degrades the software performance (see *Ternary CAM Search LogiCORE IP Product Guide* ([PG318](#)) for more information). It is therefore recommended to only use the ternary match kind where necessary. A STCAM can often be used instead.
- Response width: up to 1024 bits
- Hit/miss flag: Yes
- Size: up to 32k

Here is an example of how to instantiate a TCAM in a P4 file:

```
table forwardIPv4 {
    key          = { hdr.ipv4.flags : ternary;
                    hdr.ipv4.version : ternary;
                    hdr.ipv4.offset : ternary;
                    hdr.ipv4.hdr_len : ternary;
                    hdr.ipv4.src : exact; }
    actions      = { forwardPacket;
                    dropPacket; }
    size         = 1024;
    default_action = dropPacket;
}
```

For more details on how the compiler selects a TCAM, see [Compiler Table Selection](#). See *Ternary CAM Search LogiCORE IP Product Guide* ([PG318](#))

for more information on TCAMs.

### TinyBCAM

The TinyBCAM is a light-weight register-based version of the BCAM for less than 32 entries. The TinyBCAM requires no additional license. The primary characteristic of the TinyBCAM is that the position of the table entry in the table determines its priority.

- Key width: up to 1024 bits
- Response width: up to 1024 bits
- Size: Up to 32 entries
- Range key match type not supported

Here is an example of how to instantiate a TinyBCAM in a P4 file:

```
table forwardIPv4 {  
    key          = { hdr.ipv4.src : exact;  
                    hdr.ipv4.dst : exact; }  
    actions      = { forwardPacket;  
                    dropPacket; }  
    size         = 20;  
    default_action = dropPacket;  
}
```

### TinyTCAM

The TinyTCAM is a light-weight register-based version of the TCAM for less than 32 entries. The TinyTCAM requires no additional license. The primary characteristic of the TinyTCAM is that the position of the table entry in the table determines its priority.

- Key width: up to 1024 bits
- Response width: up to 1024 bits
- Size: Up to 32 entries
- Range key match type not supported

Here is an example of how to instantiate a TinyTCAM in a P4 file:


```
table forwardIPv4 {  
    key          = { hdr.ipv4.flags : ternary;
```

```

        hdr.ipv4.version : ternary;
        hdr.ipv4.offset : ternary;
        hdr.ipv4.hdr_len : ternary;
        hdr.ipv4.src : exact; }
    actions          = { forwardPacket;
                        dropPacket; }
    size             = 24;
    default_action   = dropPacket;
}

```

---

 **Note:** Because of the small number of entries involved and simplified implementation, the TinyTCAM can fully support a unique mask per entry. As such there is no concept of a TinySTCAM with further restrictions to the number of masks.

---

## DIRECT (DCAM)

Direct address match, unlike the other tables, 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).

- Key width: from 1 to 16 bits
- Response width: from 1 to 7k bits
- Table depth (max. number of entries):  $2^{\text{KeyWidth}}$
- Direct match: True/False
- Hit/miss flag: Yes

Here is an example of how to instantiate a direct table in a P4 file:

```

table forwardIPv4 {
    key          = { hdr.ipv4.tos : exact; }
    actions      = { forwardPacket; NoAction; }
    size         = 64;
    default_action = NoAction;
    direct_match  = true;
}

```

For more details on how the compiler selects a direct table, see [Compiler Table Selection](#). Note that in this type of table the total number of entries specified (size field) is ignored and is instead calculated from  $2^{\text{Keywidth}}$ .

## Extern Table

Extern Tables, unlike other tables, are not implemented inside VitisNetP4 IP, but outside instead, similarly to how User Externs are integrated. Any table can be made an 'extern table' as long as the attribute `user_extern` is used. This attribute takes one or two inputs, depending on the mode required, similarly to how User Externs are configured:

### Fixed latency mode

Only one value is used to specify the fixed latency value in clock cycles. Example:

```
table forwardIPv4 {
    key = { hdr.ipv4.tos : exact; }
    actions = { forwardPacket; NoAction; }
    size = 64;
    default_action = NoAction;
    user_extern = 15; // fixed latency mode, latency =
15 cycles
}
```


### Variable latency mode

Two values are used. The first value is used to specify the minimum latency value in clock cycles. The second value is used to specify the maximum pipeline capacity of the extern table in clock cycles.

Example:

```
table forwardIPv4 {
    key = { hdr.ipv4.tos : ternary; }
    actions = { forwardPacket; NoAction; }
    size = 1024;
    default_action = NoAction;
    user_extern = {5, 30}; // variable latency mode. Min
latency = 5 cycles & max pipe capacity = 30 cycles
}
```

---

 **Note:** In variable latency mode, it is important to connect the ready signals to the provided user extern interface ports. See [Table 1](#).

---

## Compiler Table Selection



The compiler uses the following general rules to select the type of table implemented:

- If all fields are defined as *exact*, a BCAM is selected by default, except in the following circumstances:
  - If the **size** is  $\leq 32$  entries, a TinyBCAM is selected.
  - If the **direct\_match** attribute is specified (and **key** is  $\leq 16$  bits), a Direct Table is selected.
  - If the size is  $\geq 1$  million entries and the target device is AMD Versal™, a DDR BCAM is automatically selected (unless the CAM\_RAM\_STYLE for this table has already been set to a non-Global setting, e.g. HBM).
- For any other combination of fields, a TCAM is selected by default, except in the following circumstances:
  - If the **size** is  $\leq 32$  entries and there are no *range* match fields, a TinyTCAM is selected. There is no reason or advantage to restrict the num\_masks in this implementation.
  - If the **num\_masks** attribute has been specified and there are no *range* match fields, an STCAM is selected.
  - If the key attribute has a single *lpm* field with a field width of  $< 64$  bits, an STCAM is used. Along with the single *lpm* field, the key attribute can optionally have any number of *exact* match fields. In this scenario, **num\_masks** is calculated automatically to equal the field width.

These default choices can be over-ridden by the user (providing the new choice is possible).

#### Table Key Match Fields

Other possible key attribute fields (in addition to *exact*, *lpm*, and *ternary* match fields) are:

- *range* - defines a range of possible values for matching;
- *unused* - used only for padding;
- *field\_mask* - similar to ternary matching, but with the restriction that the only mask values permitted are all bits set to 0 or all bits set to 1.

# Built-in Externs

Section 4.3 of the P4 language specification introduces the concept of an extern, which refers to P4 syntax that can be used to define interfaces between P4 programs and architecture-specific functionality that resides outside of those programs.

The P4 language specification describes two varieties of externs:

- Extern functions, for which the interface between the P4 program and the architecture-specific functionality is described as a function declaration.
- Extern objects, for which the interface between the P4 program and the architecture-specific functionality is described in terms of an object that provides one or more method declarations, which closely resembles a C++ class declaration.

From the perspective of a P4 program, an extern's implementation is a black box, that is, P4 itself is not concerned with what functionality a given extern declaration implements, nor how it is implemented.

One of the main purposes of P4's extern support is to allow toolchain vendors to provide a "standard library" of functions and features that are useful for many P4 programs (for instance, packet counters) and/or would be very difficult to implement efficiently in P4 code (for instance, checksum calculation).

The Vitis Networking P4 architecture supports the following Built-in Externs:

- Counter Extern
- Checksum Extern
- InternetChecksum Extern
- Register Extern

## Counter Externs

The architecture-specific implementation of the Counter Extern closely follows the PSA/PNA specification (see [References](#)). It provides a mechanism for keeping statistics. The control plane can read the counter values. A P4 program cannot read counter values, only update them.

The Counter Extern supports three different counter types:

## PACKETS

Increment values are set to 1.

## BYTES

Increment values are set to the measured packet length as it appears on the AXI4-Stream I/F at the input to VNP4 i.e., before any packet editing is performed. If, for example, FCS bytes are present in the packet, they will be included in the packet length increment value.

## PACKETS\_AND\_BYTES

The lower 35 bits are incremented by the packet length, the upper 29 bits are incremented by 1.


The counters supported are 64-bit counters (combined width in the case of PACKETS\_AND\_BYTES type). The highest number of counters supported per instance is 65,536, and there is no limit to the number of Counter Extern instances allowed. Each counter saturates at its maximum value.

A Counter width of 1-bit is also supported, where 64 counter values are packed together in a 64-bit entry of the register map. This allows for a larger number of counters with significantly less memory usage. The counters still saturate at the maximum value, in this case 1. One example use case is in conjunction with a table to monitor which table entries have been hit.

The Counter Extern is only supported within the Match\_Action Engine 'control' block of a P4 program.

The Counter Extern supports AXI4-Lite control-plane writes and reads (maximum burst size of 128 counters), with an optional clear-on-read. The Counter values are stored in LUTRAM, block RAM, or URAM, depending on the number of counters.

---

 **Note:** ECC is only supported when block RAM or URAM memory type is used.

---

See [Register Map](#) in Chapter 4 for the Counter Extern register map.

The Counter Extern is defined in xsa.p4 as -

```
extern Counter<W, S> {
  Counter(bit<32> n_counters, CounterType_t type);
  void count(in S index);
}
```

An example of a Counter Extern being used in an example P4 program can be seen in the [FiveTuple Example Design](#)).

## Checksum Extern


The architecture-specific implementation of the Checksum Extern closely follows the PSA/PNA specification (refer to [References](#)). It performs a checksum calculation across some specified header/metadata fields, based on a given Hash algorithm. The supported algorithms are:

- CRC32
- CRC16
- ONES\_COMPLEMENT16 (One's complement 16-bit sum used for IPv4 headers)

A custom data input type can be provided (scalars, headers, tuples, structs, etc.). All of the input data must be passed to the Checksum Extern in a single “apply” method call and the result of the checksum calculation is then returned. The input data must be in multiples of 16-bits (32-bits for CRC32) and no greater than 1024 bits, or else the compiler will trigger a warning.

There can be multiple instances of the Checksum Extern in a P4 program. The Checksum Extern is only supported within the Match\_Action Engine ‘control’ block of a P4 program. There is no AXI4-Lite control-plane associated with the Checksum Extern.

---

 **Note:** Checksum calculations over the packet payload is not currently supported.

---

The Checksum Extern is defined in xsa.p4 as -

```
extern Checksum<H>{  
  /// Constructor  
  Checksum(HashAlgorithm_t hash);  
  
  void apply<T, W>(in T data, out W result);  
}
```

An example of a Checksum Extern being used in an example P4 program is provided in [Forward Example Design](#).

## InternetChecksum Extern


The architecture-specific implementation of the InternetChecksum Extern closely follows the PSA/PNA specification (see [References](#)). This extern is limited to the `ONES_COMPLEMENT16` algorithm and is primarily intended for IPv4 checksum calculations. It can also be used for UDP checksum updates, but not full UDP checksum calculations as the calculation over the packet payload is not supported.

In contrast to the Checksum Extern, the InternetChecksum supports multiple different method calls and is not limited to a single method call per packet, for example, `subtract()`, followed by `add()` and followed by `get()`.

Input data must be in multiples of 16 bits and no greater than 1024 bits, otherwise the compiler will trigger a warning.

There can be multiple instances of the InternetChecksum Extern in a P4 program. The InternetChecksum Extern is only supported within the Match\_Action Engine 'control' block of a P4 program. There is no AXI4-Lite control-plane associated with the InternetChecksum Extern.

---

 **Note:** InternetChecksum calculations over the packet payload is not supported.

---

The InternetChecksum Extern is defined in `xsa.p4` as -


```
// Checksum based on `ONES_COMPLEMENT16` algorithm used in
// IPv4, TCP, and UDP.
// Supports incremental updating via `subtract` method.
// See IETF RFC 1624.
extern InternetChecksum {
  InternetChecksum();
  void clear();
  /// Add data to checksum. Data must be a multiple of 16 bits
  long.
  void add<T>(in T data);
  /// Subtract data from existing checksum. Data must be a
  multiple of
  /// 16 bits long.
  void subtract<T>(in T data);
  void get<W>(out W result);

}
```

## Register Extern

The architecture-specific implementation of the Register Extern closely follows the [PSA/PNA specification](#). It provides a mechanism for keeping stateful memories, whose values can be read and written in the data plane. Register Extern values can also be read and written from the control plane.


This extern is limited to:

- Single read and/or write access per packet.
  - Widths from 1 to 4096 bits.
  - Support for up to 64k registers per instance.
  - Support for on-chip memory only.
    - The register values are stored in LUTRAM, block RAM, or URAM, depending on the number of elements.
- 
-  **Note:** ECC is only supported when block RAM or URAM memory type is used.
- 
- Support for atomic read-modify-write operations to a single index per packet.
    - In the case of both read and write access, read must occur first.

In the case of read-modify-write operations, the compiler identifies the action located in between and generates the RTL accordingly to achieve maximum performance. The following limitations apply:

- A maximum latency of four clock cycles is supported. It is important to code as efficiently as possible to be able to achieve maximum performance. Higher latencies lead to lower performance.
- Continuous access to the same index might result in lower performance due to its atomic behavior.
- Actions are allowed in between register reads/writes only if they do not exceed the maximum allowed modify operation latency.

---

 **Note:** The @atomic notation shall be optional in respect of the code sequence bounded by the read and write method calls. In practice, the hardware shall always implement atomic behavior, whether it is specified in the P4 code or not.

---

See [Register Map](#) in Chapter 4 for the Register Extern register map. The Register Extern is defined in xsa.p4 as:

```
extern Register<T, S>{
```

```
Register(bit<32> size);  
void read(in S index, out T result);  
void write(in S index, in T value);  
}
```

An example of a Register Extern being used in an example P4 program can be seen in the TCP Example Design (see [Tcp\\_fsm Example Design](#)).

## User Externs

Another purpose of P4's extern support is to provide a mechanism for users to extend their P4 program with custom functionality that necessarily resides outside of the P4 program - perhaps due to the difficulty of implementing efficiently in P4 code.

This purpose is supported in AMD Vitis™ Networking P4 by means of a UserExtern declaration provided by VNP4 architecture. This declaration enables you to define the interface between your P4 program and the custom functionality that has been implemented externally, in this instance, in RTL. Furthermore, the VNP4 behavioral model can be extended with a software model of the custom functionality, so that it can continue to be a useful tool for verification of generated VNP4 designs.

The following sections describe how to use this mechanism to integrate custom functionality with a VNP4 design using User Externs.

### User Extern Example Design

Vitis Networking P4 installations include example code that can be used as a template for integrating custom functionality. The files required are located at:

{XILINX\_VIVADO}/examples/vitis\_net\_p4\_examples/user\_extrns

This directory contains:

- `user_externs.p4`: a heavily commented P4 source file which demonstrates the use of a simple user extern.
- `user_externs.cp`: a heavily commented C++ source file which contains a minimal example of a user extern software model.
- `include`: a directory containing a set of header files from the behavioral model required to successfully compile a user extern software model.
- `Makefile`: a sample Makefile that shows how to compile the user extern software model into a shared object for use with the behavioral model.

It is strongly suggested that you copy this entire directory to a separate location before making any modifications to these files.

The Makefile can be examined to understand the dependencies needed for compilation. In summary, these are:

- Header files from the behavioral model that have been provided.
- Header files for Boost (distributed with AMD Vivado™ Design Suite).
- Header files for GMP (distributed with Vivado Design Suite).
- GCC (distributed with Vivado Design Suite).
- GNU Binutils (distributed with Vivado Design Suite).

The Makefile requires that you have access to a Vivado Design Suite installation and have sourced either the `settings64.sh` or `settings64.csh` file (depending on the shell used) provided with Vitis Networking P4. Sourcing either script results in an environment variable named `XILINX_VIVADO` being set, which allows the Makefile to locate all of the dependencies fulfilled by the Vivado Design Suite.

## User Extern Declaration

The Vitis Networking P4 Architecture definition file (`xsa.p4`) provides the following declaration:

```
extern UserExtern<I, 0> {  
    UserExtern(bit<16> latency_in_cycles);  
    UserExtern(bit<16> latency_in_cycles, bit<16>  
pipe_capacity_in_cycles);  
    void apply(in I data_in, out 0 result);  
}
```



This is the declaration of an extern object named UserExtern, which VNP4 supports as a user customizable extern. In P4, extern object declarations resemble simplified C++ classes:

- The declaration contains a method with the same name as the declaration itself, such as a constructor.
  - This method is used to instantiate the extern object in the P4 program.
  - It is used to specify the hardware latency of the functionality implemented by the user extern. When a single argument is used in the declaration, the User extern is expected to be in 'fixed latency' mode, where the value specified is the fixed latency in clock cycles. When two arguments are used instead, the User Extern is expected to be in 'variable latency' mode, where the first argument is the minimum latency value in clock cycles and the second the maximum internal pipeline capacity in clock cycles. The user extern in variable latency mode must support back-pressure connected via a 'ready out' port found in the user extern interface. The user extern in variable latency mode can also produce back-pressure onto the VitisNetP4 instance via the 'ready in' port found in the user extern interface.
- The declaration contains a method named apply().
  - This method is invoked by the P4 program to pass data between the P4 program and the custom functionality.
- The declaration uses type specialization, which is similar to C++ template support.
  - When instantiating the extern, the user specifies the type of data shared between the P4 program and the custom functionality.

As mentioned briefly above, a P4 program utilizes this declaration to interact with custom functionality by first creating an instance of UserExtern and then later in the program, invoking the apply() method of the instance.

The following statement shows a sample instantiation of a UserExtern object:

```
UserExtern<bit<12>,bit<12>>(16) minimal_user_extern_example;
```

This instantiation describes an interface to some custom functionality

that has been named "minimal\_user\_extern\_example," which accepts a 12-bit wide input, produces a 12-bit wide output and executes in hardware with a fixed latency of 16 clock cycles.

The following statement illustrates how the interface to the custom functionality described above is used by the P4 program:

```
minimal_user_extern_example.apply(input, output);
```

This statement shows the `apply()` method of `minimal_user_extern_example` being invoked. The effect of this statement in the generated VNP4 IP is that the value stored in the variable `input` is presented to the 12-bit wide input of the custom functionality implemented by the user. 16 clock cycles later, the 12-bit result produced by the custom functionality is propagated to the output variable.

## Restrictions on User Extern Usage

The `UserExtern` declaration provides the flexibility needed to interface between P4 and custom functionality. Users are free to have an arbitrary number of instances of `UserExtern` and because of type specialization, each instance can accept different types of data. Furthermore, the type specialization allows the use of structured data with `UserExtern`, either as an input, an output or both. For example, it is permitted to pass a data structure such as a packet header into an instance of `UserExtern`, which can be very useful for implementing custom functionality such as an application-specific hashing.

However, the following tool-enforced restrictions apply to `UserExtern`:

- `UserExtern` can only be instantiated in the match-action stage of the `XilinxPipeline()`.
- The `apply()` method of a given instance of `UserExtern` can be invoked once and only once in the P4 program.
- Because structured data might be passed between P4 and the custom functionality, nested structures are not supported.

## JSON Description of User Externs

The P4 and C++ source files are a matched pair, that is, the P4 file defines an instance of `UserExtern` and the C++ file defines a software model for this instance. The two files are connected via the JSON file emitted by the P4 compiler. It is important to understand this portion of

the compiler's output and how it is used by the behavioral model for correctly implementing the custom extensions needed to describe the behavior of a UserExtern instance.

When compiled, the JSON file will contain two sections relating to each UserExtern present in the P4 design. The first is an entry in the "extern\_instances" JSON array, which describes the declaration of the instance as follows:

```
"extern_instances" : [  
  {  
    "name" : "Pipeline.minimal_user_extern_example",  
    "type" : "minimal_user_extern_example",  
    "id" : 0,  
    "source_info" : {  
      "filename" : "user_externs.p4",  
      "line" : 90,  
      "column" : 36,  
      "source_fragment" : "minimal_user_extern_example"  
    },  
    "fixed_latency" : 16,  
    "input_width" : 12,  
    "output_width" : 12  
  }  
]
```

The second is an entry in the "actions" JSON array describing the invocation of the instance's apply() method as follows:

```
{  
  "name" : "act_1",  
  "id" : 2,  
  "runtime_data" : [],  
  "primitives" : [  
    {  
      "op" : "_minimal_user_extern_example_apply",  
      "parameters" : [  
        {  
          "type" : "extern",  
          "value" :  
            "Pipeline.minimal_user_extern_example"  
        },  
        {  
          "type" : "extern",  
          "value" :  
            "Pipeline.minimal_user_extern_example"  
        }  
      ]  
    }  
  ]  
}
```

```

    {
      "type" : "field",
      "value" : ["switch_metadata_t", "input"]
    },
    {
      "type" : "field",
      "value" : ["scalars", "output"]
    }
  ],
  "source_info" : {
    "filename" : "user_externs.p4",
    "line" : 111,
    "column" : 5,
    "source_fragment" :
"minimal_user_extern_example.apply(input, output)"
  }
}
]
}

```

The behavioral model uses these definitions to locate the C++ modeling code to execute. The first section is used by the model to identify the C++ class which models a given instance of UserExtern. After finding the correct class, the model allocates an instance of it. The second section is used by the model to simulate the behavior of the apply() method. It tells the model how to locate the instance of the C++ class mentioned previously and what parameters its apply() method accepts.

Of these outputs, the description of the data types of the parameters to the apply() method (indicated with **bold** text in the code) are the most important to be aware of, because the C++ model of the apply() method needs to use corresponding data types.

## Modeling User Externs

The Vitis Networking P4 behavioral model is provided as a precompiled executable, but it supports the dynamic loading of additional shared object (.so) files which can contain code to simulate the behavior of the custom functionality. If a P4 program contains instances of UserExtern, the model must be extended in this way before it will be possible to launch it using the JSON emitted by the compiler for the P4 program in question. Doing so without extending the model causes the model to

treat the portion of the JSON describing the UserExtern as invalid, resulting in an error message that begins as follows:

```
Invalid reference to extern type
'minimal_user_extern_example'
bad_json:
  <unrecognised JSON declaration is printed here>
```

The behavioral model accepts a command line argument, `--load-modules`, which takes a comma-separated list of shared object files as the parameter value. On program startup, the behavioral model parses the list, scans the directories specified by the `LD_LIBRARY_PATH` environment variable for matching files and loads the first match it finds for each entry in the list. The shared object(s) specified should provide suitable definitions for all instances of UserExtern present in the P4 program being modeled.

The `user_externs.cpp` file provided contains a minimal example of the declarations needed to create a shared object that models the UserExtern instance from the `user_externs.p4` file.

The source file is heavily commented, but the steps required are detailed here for convenience:

- The C++ file must define a class with the same name as the instance of the UserExtern in the P4 code.
- The defined class must publicly inherit from a behavioral modeling class named ExternType.
- The defined class must contain the following statement in the public region:
  - `BM_EXTERN_ATTRIBUTES {}`
  - This statement is a macro which generates the appropriate constructor for the class.
- The defined class must contain a public method that conforms to the following specification:
  - Return type: void
  - Name: apply
  - Number of parameters: 2
  - First parameter: const reference of correct data type
  - Second parameter: non-const reference of correct data type
- Following the class definition, the class must be installed using the `BM_REGISTER_EXTERN()` macro.
  - This class takes a single parameter, the name of the class being registered.
- Following the installation of the class definition, the `apply()` method must be installed using the `BM_REGISTER_EXTERN_METHOD()` macro:
  - First parameter: the name of the class for which the method is being registered
  - Second parameter: apply
  - Third parameter: const reference of **correct data type**
  - Fourth parameter: non-const reference of **correct data type**

The **correct data type** to be used depends on the data type of the parameter passed to the `apply()` method in the original P4 program. This is where an understanding of the compiler's JSON output comes into play. Examine the section of the JSON that describes the invocation of the `apply()` method to identify the correct type to use in the C++ code. The following table presents a mapping between the "type" attribute in JSON and the corresponding C++ type:

**Table: Mappings**

JSON TYPE	C++ TYPE	NOTE
hexstr	Data &	Parameter passed in P4 is a constant
runtime_data	Data &	Parameter passed in P4 is an action parameter
field	Field &	Parameter passed in P4 is a local variable/metadata field/header field
header	Header &	Parameter passed in P4 is a structure/header

The behavior modeled in `user_extrns.cpp` is to "round-trip" a value, that is, the input passed is propagated directly to the output. While this example is trivial, far more sophisticated modeling is possible. A full discussion of the possibilities is outside the scope of this document, but users are advised to refer to the definitions in the following files to gain an understanding of the operations that can be performed on the parameters to the `apply()` method:

- `{XILINX_VIVADO}/examples/vitis_net_p4_examples/user_extrns/include/bm/bm_sim/data.h`
- `{XILINX_VIVADO}/examples/vitis_net_p4_examples/user_extrns/include/bm/bm_sim/fields.h`
- `{XILINX_VIVADO}/examples/vitis_net_p4_examples/user_extrns/include/bm/bm_sim/headers.h`

At this point it is worth highlighting a key feature of the UserExtern support in Vitis Networking P4. There is a one-to-one relationship between an instance of the UserExtern object in P4 and the classes needed to model that instance in C++, that is, one C++ class definition per UserExtern instance. This means that an arbitrary number of instances can be modeled, because the link between P4 and the behavioral model is done on a per-instance basis.

## Common Mistakes in Modeling Code

Although all of the indicated steps need to be carried out correctly in order to implement the model, of particular importance is the correct

specification of the data types for the parameters of the `apply()` method in the C++ class. Refer to the previous section, *Modeling User Externs*, for more details. Failure to correctly match up the C++ model with the types used in the JSON results in the behavioral model halting with an assertion message of the following form:

```
p4bm-vitisnetp4: ../behavioral-model-1.11.0/include/bm/
bm_sim/actions.h:331: T
bm::ActionParam::to(bm::ActionEngineState*) const [with T =
bm::Field&]: Assertion `tag == ActionParam::FIELD' failed.
```

The assertion is triggered at the point where the behavioral model tries to invoke the `apply()` method provided and detects that the parameters it accepts are not compatible with the parameters that are being passed. Equally important to the above is ensuring that the `const` keyword is used correctly with the parameters of the `apply()` method. The first parameter must be declared as `const`, while the second parameter must not. Failing to follow this guidance results in the behavioral model halting with an assertion message of the following form:

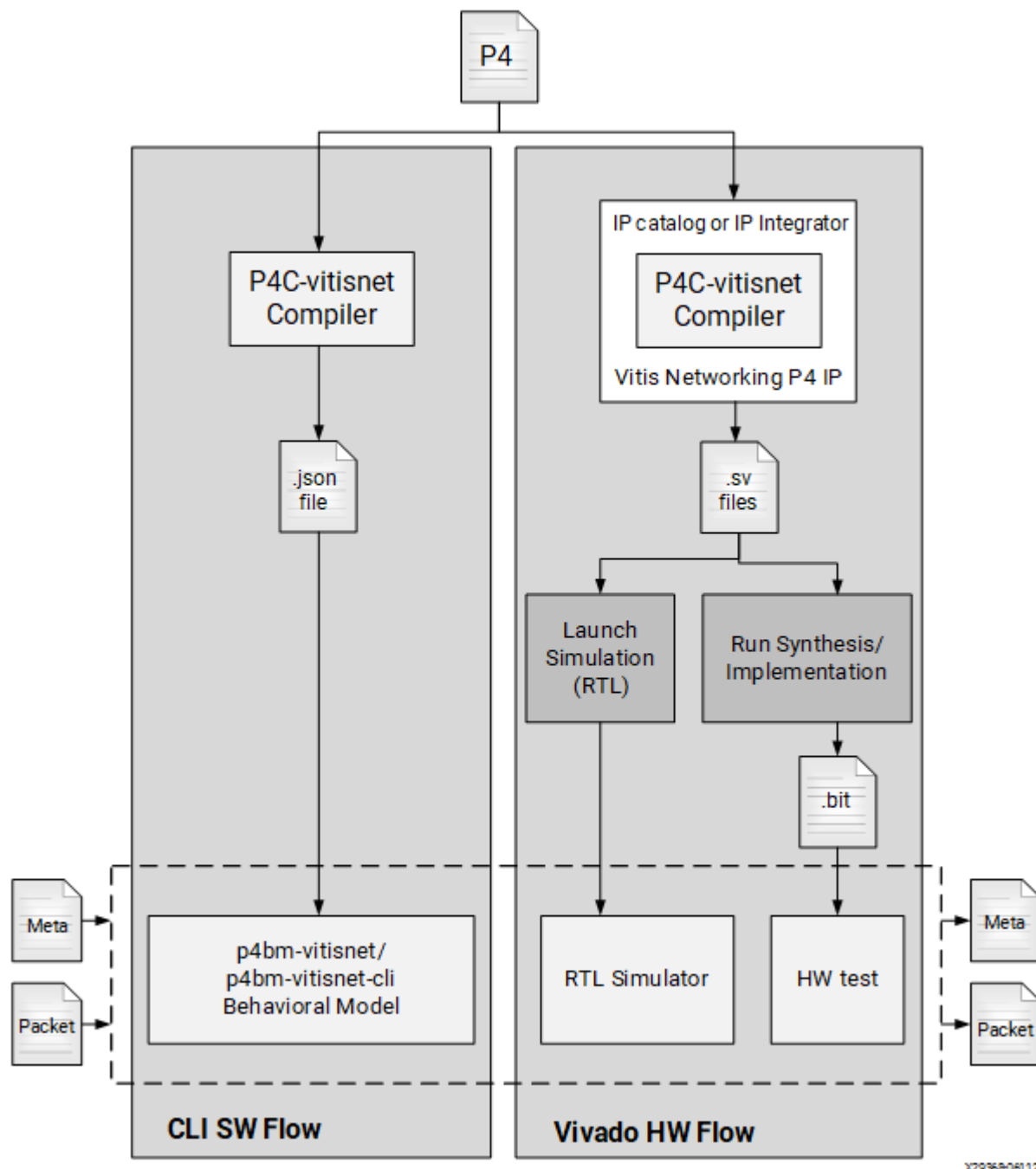
```
Assertion 'Default switch case should not be reachable'
failed, file '../behavioral-model-1.11.0/include/bm/bm_sim/
actions.h' line '286'.
```

## Vitis Networking P4 Tool Flows

The following figure depicts the Command Line Interface (CLI) software and AMD Vivado™ hardware tool flows for AMD Vitis™ Networking P4 designs. The design is first described in the P4 language which is an input to both the software and hardware flows. Refer to the P4 standard (<http://p4.org>) for more information on the P4 language.

**Figure: Vitis Networking P4 SW and HW Tool Flow**





## Command Line Interface (CLI) Software Flow

### P4C\_vitisnet Compiler

The P4C-Vitisnet Compiler takes as input a P4 file and produces an output .json file for use by the P4 Behavioral Model. The compiler is executed as a command line application. It can be run in the Vivado Hardware Flow or in the CLI software flow as follows:

```
p4c-vitisnet
```

The application supports the command line parameters described in the following table.

**Table: Command Line Parameters**

Parameter	Description
--help	Print this help message.
--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).
--nocpp	Skip preprocess, assume input file is already preprocessed.
--pp file	Pretty-print the program in the specified file.
--toJSON file	Dump the compiler IR after the midend as JSON in the specified file.
--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 treat all warnings as warnings (the default) if no diagnostic is specified.
--Werror[=diagnostic]	Report an error for a compiler diagnostic, or treat all warnings

Parameter	Description
	as errors if no diagnostic is specified.
--testjson	[Compiler debugging] Dump and undump the IR.
-T loglevel	[Compiler debugging] Adjust logging level per file (see below).
-v	[Compiler debugging] Increase verbosity level (can be repeated).
--top4 pass1[,pass2]	[Compiler debugging] Dump the P4 representation after passes whose name contains one of `passX' substrings. When '-v' is used this will include the compiler IR.
--dump folder	[Compiler debugging] Folder where P4 programs are dumped.
--emit-externs	[VNP4 back-end] Force externs to be emitted by the backend. The generated code follows the VNP4 JSON specification.
-o outfile	Write output to outfile.

loglevel format is: sourceFile:level,...,sourceFile:level  
where sourceFile is a compiler source file and level is the verbosity level for LOG messages in that file.

## Behavioral Model

The Behavioral Model creates an independent replica of the operations described in the source file to compare against expectations and the RTL implementation. See [Running the Behavioral Model](#) for details.

## Input Files

The Behavioral Model takes as input the .json file, which is provided by the P4C-vitisnet compiler. It also takes as input:

- A packet file (in text format or in PCAP format)
- A metadata file

#### Packet File


The Packet.user input packet file is a 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:** the number of hex digits in a packet must be even as each packet should be an integer 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 2 (89 bytes)
% Ethernet header: [ DstMAC=111111111112 SrdMAC=aaaaaaaaaaab
EtherType=0800 ]
11 11 11 11 11 12 aa aa aa aa aa ab 08 00
% IPv4 header:[ Version=4 HdrLen=5 DSCP=00 ECN=3 Length=004b
ID=4357 Flags=0 Fragment=0000 TTL=39 Protocol=11
Checksum=1d1f SrcAddr=7cf6cd9c DstAddr=cc930a03 ]
45 03 00 4b 43 57 00 00 39 11 1d 1f 7c f6 cd 9c cc 93 0a 0
% UDP header:[ SrcPort=211e DstPort=5b98 Length=0037
Checksum=aa9a ]
21 1e 5b 98 00 37 aa 9a
% Payload
6b fa a4 95 d2 54 47 71 92 29 8b 0f 8d e7 e2 99 08 f0 13 0b
```

```
ef 64 07 3b fe e0 d4 6a ad 3f 5b 3e fd 58 33 49 fc 8f 86 00
1c 4f 00 a0 d0 6d 70
;
```

Further examples can be seen in the Example designs supplied in the tool installation.

#### PCAP Format

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

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

#### Metadata File

The packet.meta input metadata file is a text file that contains the stimulus for the input metadata interface in a bus-protocol-independent text format.

#### Text Format

The text format has the following structure:

- Lines beginning with a percent character (%) are considered line comments.
- Entries are terminated with a semi-colon.
- Field values are always defined in hex:
  - *field1=1234 field2=5678*
  - *field3=9abc*
  - *field4=def0;*
- Lines containing stored persistence values start with '#'. Persistence values remain set until a later definition changes them. There is no semi-colon at the end of persistence value changes:
  - *# field1=1234 field2=5678*
  - *# field3=9abc*
- Lines starting with '@' trigger entry replication, with the number after the symbol being a decimal number. There is no semi-colon at the end of replication lines:
  - *@ 100 field1=1234 field2=5678*
- Lines starting with '@' followed by the '\*' character signal that replication continues for all remaining packets. The simulation stops reading from the file:
  - *@ \* field1=1234 field2=5678*

The following is an example of the metadata file in text format.

```
metadata.new_vlan_pcp=1 metadata.new_vlan_cfi=1
metadata.new_vlan_vid=ae8 ;

# metadata.new_vlan_pcp=2 metadata.new_vlan_cfi=0
metadata.new_vlan_vid=5c0
metadata.new_vlan_pcp=6 metadata.new_vlan_cfi=0
metadata.new_vlan_vid=335 ;

@ 4 metadata.new_vlan_pcp=3 metadata.new_vlan_cfi=1
metadata.new_vlan_vid=9
metadata.new_vlan_pcp=4 metadata.new_vlan_cfi=1
metadata.new_vlan_vid=dc1 ;
metadata.new_vlan_pcp=5 metadata.new_vlan_cfi=0
metadata.new_vlan_vid=5d9 ;
metadata.new_vlan_pcp=5 metadata.new_vlan_cfi=1
metadata.new_vlan_vid=410 ;
metadata.new_vlan_pcp=5 metadata.new_vlan_cfi=1
```

```
metadata.new_vlan_vid=c04 ;
```

```
@ * metadata.new_vlan_pcp=3 metadata.new_vlan_cfi=1  
metadata.new_vlan_vid=99
```

## Output Files

The behavioral model outputs modified Packet Data and Metadata which can be verified and used as golden data in simulation and board testing. The output files use the same formats as the text input files.

## Running the Behavioral Model

The behavioral model uses the .json output provided by the P4 compiler and provides two applications:

### **p4bm-vitisnet**

This is an application which models the data plane. It consumes the .json file and implements the packet processing behavior defined by the corresponding P4 program.

### **p4bm-vitisnet-cli**

This is an application which models the control plane. It provides a command line interface (CLI) used to manage the data plane (p4bm-vitisnet).

These applications are used together to carry out behavioral modeling. Typical usage is as follows:

- The p4bm-vitisnet process is launched (usually as a background task), specifying a .json file as an input.
- The p4bm-vitisnet-cli process is launched, which connects to p4bm-vitisnet via a socket interface.
- Management commands are issued at the CLI to interact with the model, for example, add table entries, trigger traffic flow, etc.

To simplify usage and enable the use of scripts for controlling the model, a third application called run-p4bm-vitisnet is also provided. This application is simply a wrapper around p4bm-vitisnet and p4bm-vitisnet-cli that also accepts a script containing commands which are submitted

to p4bm-vitisnet-cli.

For the socket connections involved, the default port used is 9090, which can be overridden with a user-specified value.

#### p4bm-vitisnet Application

The p4bm-vitisnet application models the data plane behavior of a P4 program. Its primary input is the .json file produced by the P4 Vitis Networking P4 compiler.

The key features of p4bm-vitisnet are as follows:

- Supports the Vitis Networking P4 Architecture.
- Packet data is inserted into the model from files only, of which both PCAP and the AMD packet text file format are supported (see [Packet File](#) for more information).
- Input packet metadata can optionally be inserted into the model and output packet metadata is always emitted to a file (see [Metadata File](#) for more information).
- Dropped packets can optionally be emitted as zero-length packets, enabling the metadata associated with them to be captured if desired.
- Packet flow is controlled purely from the associated p4bm-vitisnet-cli, which allows table management to be interleaved with packet flow if desired.

When launched, p4bm-vitisnet parses the provided .json file and uses it to allocate and configure all of the resources needed to perform the modeling (parsers, tables, etc.). It then waits for commands to be received from p4bm-vitisnet-cli on the specified socket interface. The model uses *run to completion* semantics; all commands are executed in full in the order they are received.

The application supports the command line parameters identified in the following table.

**Table: Optional Parameters**

Parameter	Description
-h [--help]	Prints a list of possible parameters and a short



Parameter	Description
	description.
--thrift-port arg	TCP port on which to run the Thrift runtime server (socket connection for CLI).
--log-console	Enable logging on stdout.
--log-file arg	Enable logging to given file.
-L [ --log-level ] arg	Set log level, supported values are 'trace', 'debug', 'info', 'warn', 'error', 'off'; default is 'trace'.
--log-flush	If used with '--log-file', the logger flushes to disk after every log message.
--dump-packet-data arg	Specify how many bytes of packet data to dump upon receiving and sending a packet. The logger is used to dump the packet data, with log level 'info'. Ensure the log level you have set does not exclude 'info' messages; default is 0, which means that nothing is logged.
-v [ --version ]	Display version information.
--json-version	Display VNP4 JSON version supported in the format <major>.<minor>.
--load-modules arg	Comma separated list of runtime loadable modules (such as extern modeling).

The p4bm-vitisnet-cli application provides the control plane functionality needed to operate the p4bm-vitisnet data plane.

When launched, p4bm-vitisnet-cli attempts to establish a connection to an already running p4bm-vitisnet process, using the port specified. If a connection cannot be established, it retries for a user-specified number of attempts before giving up.

Upon successful connection, you are presented with an interactive command line which includes built-in help for all supported commands. The list of commands can be obtained by entering help and usage information for each command can be obtained by entering help <command>. For example: help table\_add.

The following table describes the most important commands supported by the CLI.

**Table: CLI Supported Commands**

Parameter	Description
- help	Display the list of commands and provides access to usage information for each command
- show_tables	List all tables present
- table_add	Add an entry to a specified table
- table_delete	Remove an entry from a specified table
- table_clear	Reset a specified table
- table_dump	Display the contents of a specified table
- counter_read	Read a counter value from a specified counter extern
- counter_write	Update a counter value of a specified counter extern
- counter_reset	Reset all counter values of a specified counter extern

Parameter	Description
- run_traffic	Define the source of packet data, and optionally metadata (see below for further details)
- exit	Terminate both the CLI and the model (causes p4bm-vitisnet to exit)

The `run_traffic` command takes a single parameter which is used as a base from which the source/input and destination/output files for packet data and metadata are derived by appending some extra text. For instance command `run_traffic` causes the following to take place:

1. The p4bm-vitisnet application attempts to open the `example_in.pcap` file for reading.
2. If step 1 succeeded, it then opens `example_out.pcap` for writing.
3. If step 1 failed, the p4bm-vitisnet application attempts to open `example_in.user` for reading.
4. If step 3 succeeded, it then opens `example_out.user` for writing.
5. The p4bm-vitisnet application then attempts to open `example_in.meta` for reading
6. The p4bm-vitisnet application finally attempts to open `example_out.meta` for writing.

These steps describe how the model selects between the PCAP file and the AMD packet text file format, for example, PCAP is given priority if present. The file selected as input determines the format of the output, for example, if a PCAP input is found, the output is also PCAP. The process also shows that the input metadata file is optional and can be omitted if desired. When omitted, all metadata is initialized to zero by the model. The application supports the command line parameters identified in the following table.

**Table: Optional Parameters**

Parameter	Description
-h [--help]	Prints a list of possible

Parameter	Description
	parameters and a short description.
--thrift-port THRIFT_PORT	Thrift server port for table updates (socket connection to model).

#### run-p4bm-vitisnet Application

The run-p4bm-vitisnet application is a helper utility included to improve the ease of use of both p4bm-vitisnet and p4bm-vitisnet-cli applications by abstracting the user from launching both processes and providing the ability to apply a set of commands automatically from a file.

The program accepts two mandatory parameters - the .json file (used by p4bm-vitisnet) and a test-case text file containing the CLI commands (passed into p4bm-vitisnet-cli). When invoked, the application launches both p4bm-vitisnet and p4bm-vitisnet-cli as sub-processes. Once both processes have started and their connection has been established, run-p4bm-vitisnet opens the specified test-case text file and reads its contents line by line. Blank lines and comment lines (those which start with the '#' character) are ignored. All other lines are passed into the p4bm-vitisnet-cli sub-process. When the end of the test-case file is reached, run-p4bm-vitisnet sends the exit command to p4bm-vitisnet-cli, causing both sub-processes to terminate gracefully.

The application supports the command line parameters described in the following table. The following snippet shows an example of the help command line parameter.

```
run-p4bm-vitisnet --help
usage: run-p4bm-vitisnet [-h] [-p THRIFT_PORT] [-l LOG_FILE]
-j JSON [-s SCRIPT] [-m LOAD_MODULES] [-d] [-c
CONNECT_COUNT]
```

**Table: run-p4bm Vitis Networking P4 Supported Commands**

Parameter	Description
-----------	-------------

Parameter	Description
-h, --help	Shows this help message response (see above example)
-p THRIFT_PORT, --thrift-port THRIFT_PORT	Thrift server port for simulation
-l LOG_FILE, --log-file LOG_FILE	Name for simulation output logs
-j JSON, --json JSON	JSON description of P4 program to simulate
-s SCRIPT, --script SCRIPT	File containing commands to run through simulator
-d, --drop-zero-length	Emit dropped packets as zero length packets
-c CONNECT_COUNT, --connect-count CONNECT_COUNT	Number of times to attempt thrift connection (0 => unlimited)
-m LOAD_MODULES, --load-modules LOAD_MODULES	Comma separated list of runtime loadable modules (such as extern modeling)

### Behavioral Model Use Case

To verify the functionality of a P4 program <testname>.p4 (along with the input packet data <packetname>\_in.pcap|user) using the behavioral model, complete the following steps:

1. Compile the P4 program using p4c-vitisnet. This generates a JSON file that can be used in the behavioral model:

```
p4c-vitisnet <testname>.p4 -o <json_name>.json
```

2. Create a <testcase>.txt text file containing a command to run the traffic data, ensuring that the input packet data file follows the <name>\_in.pcap|user naming convention. If there are any tables in the P4 program, the table entries should be added to this testcase file (before the run\_traffic command):

```
echo "run_traffic <packetname>" > <testcase>.txt
```

The format of a `table_add` command looks as follows:

```
table_add <table_name> <action_name> <key0> <key1>  
<key2>... => <response0> <response1>...
```

For example:

```
table_add FiveTuple InsertVLAN 0x6e39cb7c 0x945e726d 0x06  
0xc4aa 0x8ca2 => 0x3 0x1 0x060
```

3. Run the `run-p4bm-vitisnet` application to generate outputs from the model:

```
run-p4bm-vitisnet -j <json_name>.json -s <testcase>.txt
```

Both the compiler and the behavioral model CLI programs need to be executed from within a Vivado context. The following example illustrates this using the supplied `FiveTuple` example design with the command:

```
vivado -mode batch -source cli_example.tcl
```

where the Tcl script `cli_example.tcl` is:

```
exec p4c-vitisnet $::env(XILINX_VIVADO)/data/ip/xilinx/  
vitis_net_p4_v1_0/example_design/examples/five_tuple/  
fiveTuple.p4 -o cli_example.json
```

```
exec cp $::env(XILINX_VIVADO)/data/ip/xilinx/  
vitis_net_p4_v1_0/example_design/examples/five_tuple/  
traffic_in.user .
```

```
exec run-p4bm-vitisnet -p 9091 -s $::env(XILINX_VIVADO)/data/  
ip/xilinx/vitis_net_p4_v1_0/example_design/examples/  
five_tuple/cli_commands.txt -j cli_example.json
```

```
exec ls -l traffic_out.user traffic_out.meta
```

Warnings

A warning is displayed by the model when it detects a read or write access to a header field that is invalid:

```
Detected read access of field in an invalid header udp. In P4
this behaviour is undefined. You should modify your P4
program to eliminate this access
```

When this message is displayed, the behavior is unexpected, possibly leading to the behavioral model and RTL reading/writing different values. An example of P4 code that might create such a warning is:

```
meta.ipv4_version = hdr.ipv4.version
```

To avoid the warning occurring with the above code, the code could be modified to:

```
if (hdr.ipv4.isValid()) {
    meta.ipv4_version = hdr.ipv4.version;
}
```

Another example of P4 code that might create such a warning is:

```
hdr.ipv4.version = 4;
hdr.ipv4.setValid();
```

To avoid the warning occurring with the above code, the code could be modified to:

```
hdr.ipv4.setValid();
hdr.ipv4.version = 4;
```

## Exceptions

If the model attempts to extract a header that exceeds the *Header Depth Limit* packet offset of 8191 bytes, the following message is displayed:

```
Exception while parsing: HeaderDepthLimitExceeded
```

# Vivado Design Suite Hardware Flow

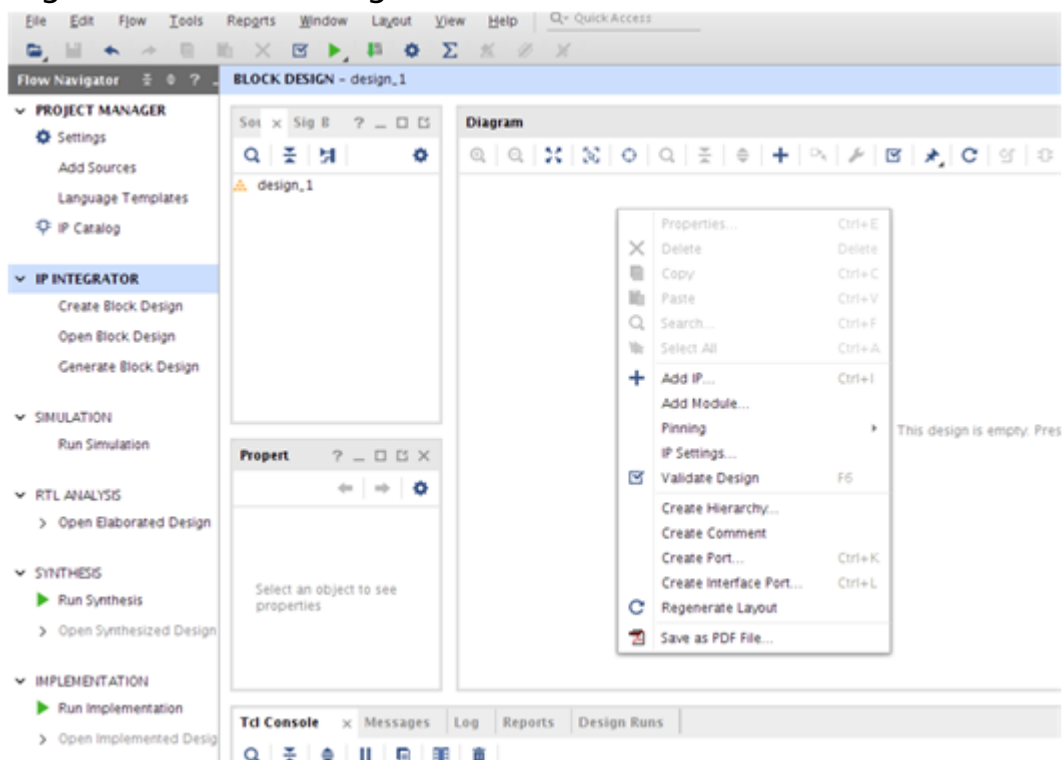
AMD Vivado™ Design Suite 2018.3 or later is used for this flow.

Instantiating the AMD Vitis™ Networking P4 IP in the Vivado tool can be done in two different ways, using IP integrator or using IP Catalog. Vitis Networking P4 (and older SDNet versions) must be used with the correct version of the Vivado tool. For example, SDNet version 2018.3 must be paired with Vivado version 2018.3, VNP4 2020.2 with Vivado version 2020.2 etc.

## IP Integrator

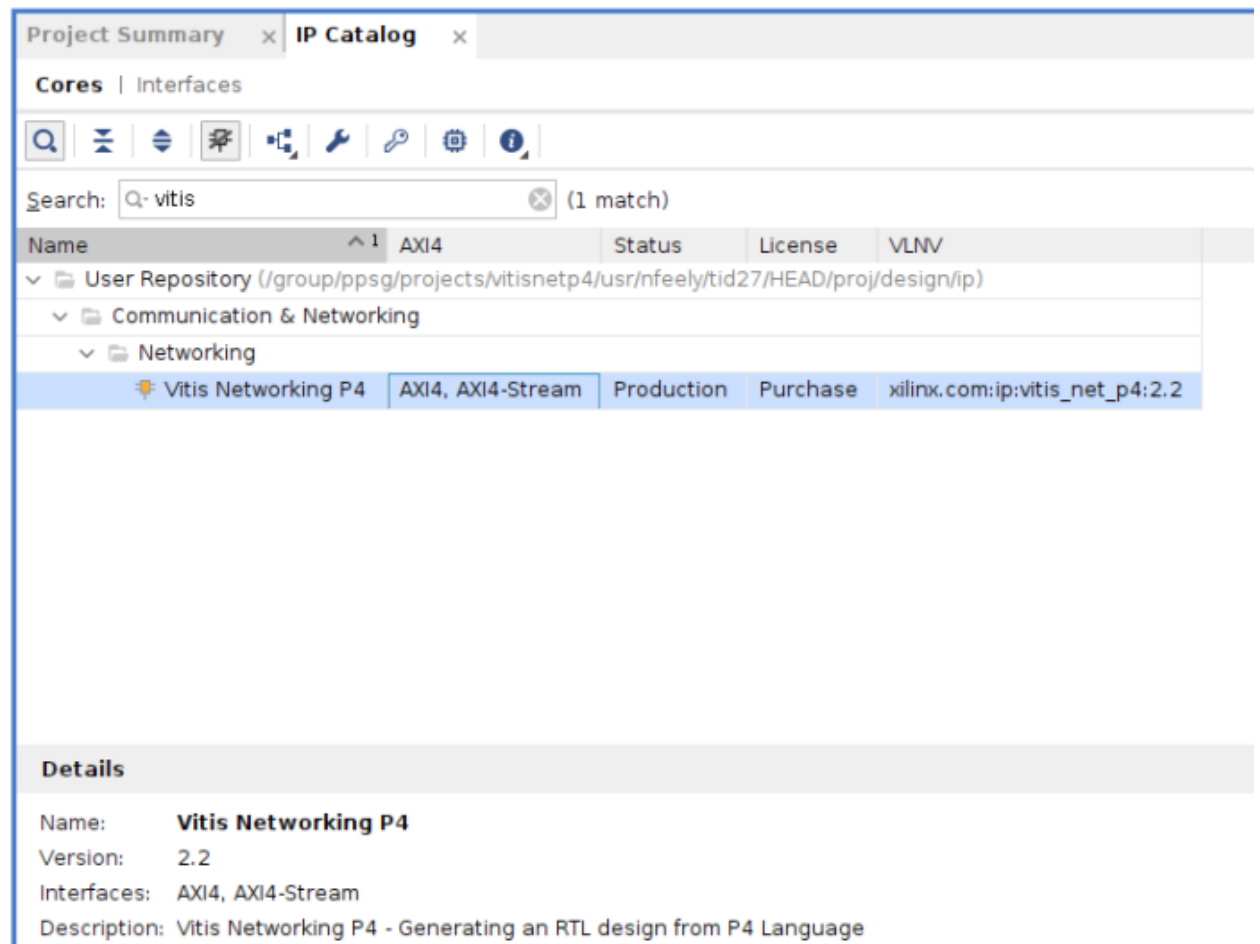
To instantiate the Vitis Networking P4 IP in Vivado Design Suite using IP integrator, follow these steps:

1. Open a project in the Vivado tool.
2. Click Create Block Design in the panel on the left.
3. Right-click in the Diagram window and select Add IP.



4. Search for "Vitis Networking P4" in the search box of the window that appears and select Vitis Networking P4.





## IP Catalog

To instantiate the Vitis Networking P4 IP via IP Catalog, follow these steps:

1. Open a project in the Vivado tool.
2. Click IP Catalog in the panel on the left. Search for *vitis* in the search box of the IP catalog window and select:

### **Vitis Networking P4**

for 2021.1/2021.2/2022.1/2022.2/2023.1/2023.2/2024.1/2024.2

### **vitisnetp4\_v1\_0**

for 2020.2

### **sdnet\_2\_2**

for 2020.1

### **sdnet\_2\_1**

for 2019.2

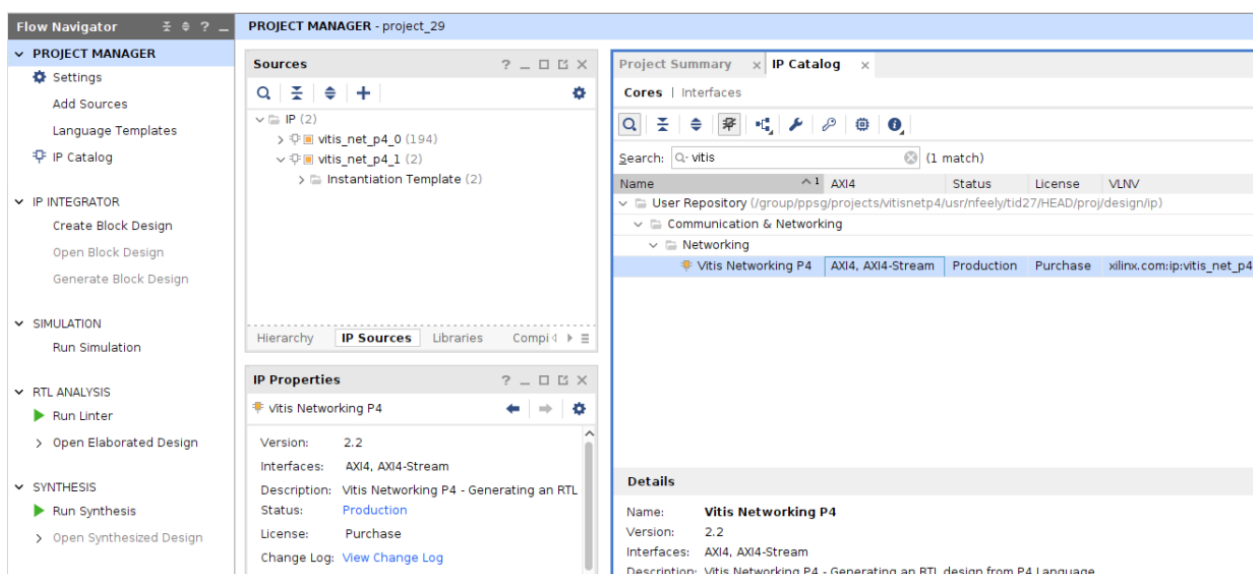
### **sdnet\_2\_0**

for 2019.1

## sdnet\_1\_1

for 2018.3

3. The Vitis Networking P4 IP can be configured by double-clicking the instantiated IP.



## Vitis Networking P4 IP

This section contains information and instructions for using and customizing the Vitis Networking P4 IP.

---

**Recommended:** Other than selecting an input P4 file and modifying the Packet Data Bus Width, all other configuration settings are advanced settings that can normally be left at their default.

---

The Vitis Networking P4 IP GUI includes six tabs, four of which are not visible until a P4 file is selected from the main Top Level Settings tab:

- Top level settings
- Tables
- Metadata
- Clocking
- Advanced Options
- About Vitis Networking P4

### Top Level Settings

The Top Level Settings Tab is shown in the following figure.

**Figure: Top Level Settings Tab**

Component Name

Top Level Settings | Tables | Externs | Metadata | Clocking | Advanced Options | About VitisNetP4

**AXI Stream Configuration**

Packet Data Bus Width


Packet Rate (Mp/s)

**P4 File**

☒ Select P4 Example

P4 File Example

**Re-Compile P4**

 Vitis

Compiled when P4 file Configuration panel. d by the user, then re-compile the P4.

Estimated Latency = 0 nanoseconds (0 clock cycles)

## Packet Data Bus Width

AXI4-Stream Packet Data Bus Width: can be set to 32-bit, 64-bit, 128-bit, 256-bit, 512-bit, or 1024-bit.

## Packet Rate

Packet rate in million packets/second. This parameter is relevant to any CAM tables that might be in your P4 program. A lower packet rate can result in a lower resource utilization for the CAMs. Vitis Networking P4 applies back-pressure on the `s_axis` interface if necessary to avoid exceeding the packet rate at the CAMs.

## P4 File

Select Select P4 Example to allow the selection of P4 example designs from the P4 Example drop-down menu.

Alternatively, browse to select a P4 file in P4 File Location then click OK. The VNP4 IP invokes the P4C-VitisNetP4 compiler, automatically compiling the P4 file once it is selected, and unlocks the other four tabs.

## P4 Compile

Re-compiles the P4 for use when the selected P4 file has been edited by the user.

## Calculate Latency

Calculates the total latency of the VNP4 system. This button appears, for convenience, on each tab. See [Latency](#) for additional information on the latency calculation.

## Tables

The Tables tab is shown in the following figure (which is taken from an AMD Versal™ design).

**Figure: Tables**

**Top Level Settings** | **Tables** | Externs | Metadata | Clocking | Advanced Options | About VitisNetP4

**Global Settings For Tables**

CAM Table RAM Style: Auto

**Table Settings Summary**

ID	Instance Name	Mode	Key Width	Response Width	Total Entries	Memory Resources
1	FiveTuple	BCAM	104	30	8192	60 (BRAM36)

**Table Implementation Settings**

☒ Show Detailed Settings

Instance ID (FiveTuple): 1

RAM Style: Global Setting

Clock Source: CAM Mem Clock

Lookup Rate: 300.0 [1.0 - 600.0]

CAM Optimization Mode: NONE

## Settings for Tables

Settings for Tables defines the generic settings for all *Direct* and *CAM*

tables in the design. These global settings are the default settings for all tables. Each table has an individual *Ram Style* parameter which can be used to override the RAM Style global settings. This section is only shown one table at a time, for the table selected with the 'Table ID' dropdown menu. Configuration parameters will be displayed accordingly based on the table type.

#### Direct Table Configuration

##### **Ram Style**

Selects the RAM style to be applied on a Direct table instance basis. The options are:

##### **Global Setting**

Uses the global RAM style as defined under Global Table RAM Style.

##### **Block RAM**

Table implemented in Block RAM.

##### **Ultra RAM**

Table implemented in URAM.

#### CAM Table Configuration

If the Show More Detail box is checked, more information is displayed per table (key width, resp width, num entries, num masks, and mode).

##### **Ram Style**

Selects the RAM style to be applied for each CAM instance. The options are:

**Global Setting**

Uses the global CAM RAM style as defined under Global Setting for Tables.

**Block RAM**

Table implemented in Block RAM.

**Ultra RAM**

Table implemented in URAM.

**DDR**

Table implemented in Double Data Rate memory.

**HBM**

Table implemented in High Bandwidth Memory (only supported for BCAM).

**Clock Source**

Sets the clock source to be either CAM Memory clock or AXI4-Stream clock. The CAM Memory clock allows for the CAMs to run at a higher rate (up to 600 MHz) to reduce resource utilization.

**HW Update**

This tick box appears if a Versal part is selected, Ram Style is set to either DDR or HBM and Mode is set to BCAM.

If this box is unchecked, table management (inserts/deletes/updates) is performed by software and requires a shadow memory. The performance of the inserts/deletes/updates depends on the processor.

If this box is checked, no shadow memory is required and table management performance is independent of the processor, but extra hardware resources are needed. However, insert/delete/update operations are still triggered by software in hardware mode.


**Cache Entries**

The drop-down menu beside Cache Entries appears if a Versal part is selected, Ram Style is set to either DDR or HBM and Mode is set to BCAM. Cache support can be disabled by selecting NONE.


**Lookup rate**

The lookup rate of a CAM table can be individually tuned to the required value.

 **Note:** The lookup rate cannot exceed the configured packet rate.

 **Note:** If the selected configuration results in too many memory units to be generated by the CAM, an error appears in the Vivado console as shown in the following figure. This error should be cleared before proceeding.

### Figure: Too Many Memory Units Error

 [xilinx.com:ip:vitis\_net\_p4:1.0-0] vitis\_net\_p4\_0: CAM ERROR: CAM\_ERROR\_TOO\_MANY\_UNITS

## Externs

The Externs tab is shown in the following figure.

### Figure: Externs Tab

Component Name vitis\_net\_p4\_0

Top Level Settings

Tables

Externs

Metadata

Clocking

Advanced Options

About VitisNetP4

User Externs

ID	Instance Name	Input Width	Output Width
1	round_trip_field	3	3

Checksum Externs

ID	Instance Name	Hash Algorithm	Data Width
1	ck_crc16	CRC16	32
2	cksum	ONES_COMPLEMENT16	16


Counter Externs

ID	Instance Name	Count Type	Data Width	Total Counters
1	PacketCounter	packets	64	1

Register Externs

ID	Instance Name	Data Width	Total Registers	Compute Latency
1	reg_ext	128	8	3

All externs found in the specified p4 file are displayed in this tab only for reference. There are four individual sub-sections, one per extern type. In each sub-section a table is displayed with some key configuration values, including instance name, width, type, etc.

 **Note:** Checksum data width and Register latency are only displayed after clicking the Calculate Latency button.

## Metadata

The Metadata tab is shown in the following figure.

**Figure: Metadata Tab**

**Top Level Settings** | **Tables** | **Externs** | **Metadata** | **Clocking** | **Advanced Options** | **About VitisNetP4**

**Metadata Settings**

☐ Enable Metadata Output for Dropped Packets

**User Metadata I/O**

Metadata Field	Input Connected	Output Connected
metadata.port	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

All metadata fields present are displayed. You can select/deselect all metadata fields. By default, all metadata fields are provided as input and output ports, but they can be deselected if not needed as an I/O to reduce resource utilization.

The width of the metadata input and output ports remain fixed to the number of metadata fields defined in the P4 file, regardless of what metadata fields are selected/deselected. However, internally in the Vitis Networking P4 IP, optimization is performed based on what has been selected/deselected. The generated `<design_name>_pkg.sv` file indicates the bit positions of the metadata fields that have been selected (see [Generated Files](#) for more information).

### Enable Metadata Output for Dropped Packets

Can be selected to enable metadata output for packets which have been dropped.

### Enable All IO

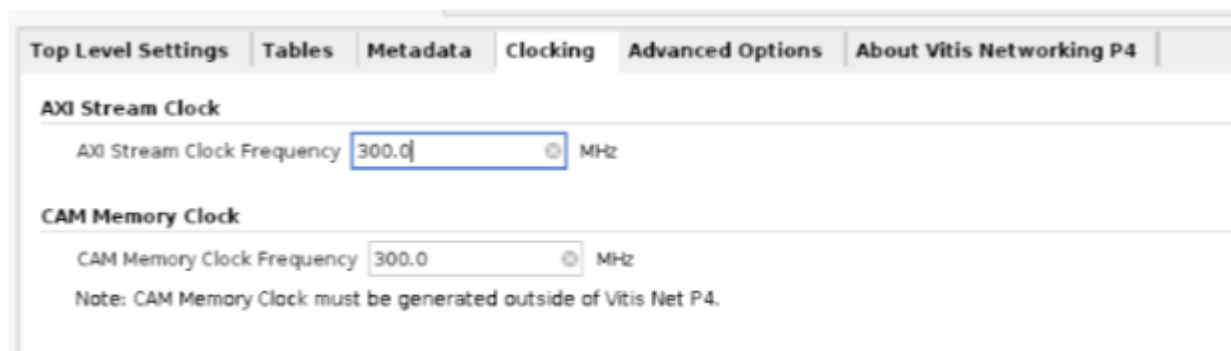
Can be selected to restore all check boxes in the table to the selected state.

## Clocking

The Clocking tab is shown in the following figure.

**Figure: Clocking Tab**





The screenshot shows a web application interface with a tabbed menu at the top: 'Top Level Settings', 'Tables', 'Metadata', 'Clocking', 'Advanced Options' (selected), and 'About Vitis Networking P4'. Under the 'Advanced Options' tab, there are two sections. The first section, 'AXI Stream Clock', contains a label 'AXI Stream Clock Frequency' followed by a text input field containing '300.0' and a unit selector set to 'MHz'. The second section, 'CAM Memory Clock', contains a label 'CAM Memory Clock Frequency' followed by a text input field containing '300.0' and a unit selector set to 'MHz'. Below these fields is a note: 'Note: CAM Memory Clock must be generated outside of Vitis Net P4.'

Values entered are used to evaluate the impact of the clock frequency on latency within the system and also for implementation.

### AXI Stream Clock

Enter the AXI4-Stream clock frequency in MHz (up to a maximum of 600 MHz). For reliable timing closure use 300 MHz or below.

### CAM Memory Clock

Enter the CAM Memory Clock frequency in MHz (up to a maximum of 600 MHz). The CAM Memory Clock allows for the CAMs to run at a higher rate than the AXI4-Stream clock to reduce resource utilization. There are no requirements on the ratio between the two clocks, but they must both be derived from the same original clock source. The ratio between the CAM Memory Clock and the PKT\_RATE parameter determines the TDM\_FACTOR which can help achieve potential resource savings. Refer to the CAM product guides listed in [References](#) for details on the TDM\_FACTOR to achieve potential resource savings.

## Advanced Options

The Advanced Options tab is shown in the following figure.


### Figure: Advanced Options Tab

Top Level Settings	Tables	Externs	Metadata	Clocking	Advanced Options	About VitisNetP4
<b>Memory Error Correction</b>						
<input type="checkbox"/> Enable ECC on all BRAM/URAM NOTE: CAM Tables always have ECC Enabled.						
<b>Statistic &amp; Control Registers</b>						
<input type="checkbox"/> Enable Registers						
<b>Debug I/O Signal Logging</b>						
<input type="checkbox"/> Dump AXI & AXIS I/O signals						
<b>CAM Tables Debug Flags</b>						
<input type="checkbox"/> CAM_DEBUG_NO_ERROR_MSG <input checked="" type="checkbox"/> CAM_DEBUG_SKIP_MEM_INIT <input type="checkbox"/> CAM_DEBUG_HW_WR						
<input type="checkbox"/> CAM_DEBUG_HW_LOOKUP						

### ECC Enable

ECC is supported and enabled on the logic CAMs by default and cannot be disabled. ECC is disabled by default on Direct Tables, Counter Externs and re-alignment FIFOs but you can choose to enable it by selecting the associated box. This only applies where these are implemented in Block RAM or URAM, ECC for LUTRAM is not supported. Any single-bit ECC errors are detected and corrected when data is read from the RAMs. Double-bit ECC errors are detected but cannot be automatically corrected. ECC errors are flagged to software via an interrupt port `irq` (see [Vitis Networking P4 Tool Interface](#)). The Statistics register, if enabled, includes ECC debug/status registers (see [Register Map](#)). There are also ECC error-scrubbing mechanisms included to read all entries of the BRAMs and URAMs every 1 ms for early detection of ECC errors. In case of single-bit error detection, the corrected data is then automatically written back to the RAM to remove the error.

---

 **Note:** ECC error-scrubbing mechanisms do not apply to FIFOs, where data is not expected to remain in the RAM for longer than 1 ms.

---

### Vitis Networking P4 Statistics Registers

The Statistics/Control registers can be enabled (see [Register Map](#)) by selecting the associated check box.

### Debug I/O Capture Enable

You can choose to enable I/O data capture by selecting the associated check box. When this is enabled, the following three log files are created upon simulation of the system:

- `/<proj_location>/<proj_name>/<proj_name>.sim/sim_1/behav/<simulator_name>/<component name>_stream_in.log`
  - This contains the `s_axis` and input user metadata interfaces along with a timestamp.
  - The format of this file is: `tready, tvalid, tlast, tdata, tkeep, metadata_valid, metadata` and timestamp (tab separated values).
- `/<proj_location>/<proj_name>/<proj_name>.sim/sim_1/behav/<simulator_name>/<component name>_stream_out.log`
  - This contains the `m_axis` and output user metadata interfaces along with a timestamp.
  - The format of this file is: `tready, tvalid, tlast, tdata, tkeep, metadata_valid, metadata` and timestamp (tab separated values).
- `/<proj_location>/<proj_name>/<proj_name>.sim/sim_1/behav/<simulator_name>/<component name>_control_if.log`
  - This contains the AXI4-Lite interface along with a timestamp.
  - The format of this file is: `s_axi_awaddr, s_axi_awvalid, s_axi_awready, s_axi_wdata, s_axi_wstrb, s_axi_wvalid, s_axi_wready, s_axi_bresp, s_axi_bvalid, s_axi_bready, s_axi_araddr, s_axi_arvalid, s_axi_arready, s_axi_rdata, s_axi_rresp, s_axi_rvalid, s_axi_rready` and timestamp (tab separated values).

## CAM HW Debug

You can optionally enable/disable the following CAM HW Debug options, resulting in extra information being displayed in the Vivado Design Suite console when a simulation is run.

### **CAM\_DEBUG\_NO\_ERROR\_MSG**

Disables printout of software error messages (disabled by default).

### **CAM\_DEBUG\_SKIP\_MEM\_INIT**

Skips memory initialization, useful for speedup of simulations (enabled by default).

## **CAM\_DEBUG\_HW\_WR**

Hardware prints write operations (disabled by default). Example:

```
# ** Info:
example_top.DUT.inst.match_action_engine_inst.table2_MyProcessing_FiveTuple_inst.hcam_table_inst.HW_WR address =
0x0047f200 data = 0xaa20101124116cc1 1407770
```

## **CAM\_DEBUG\_HW\_LOOKUP**

Hardware prints look-up operations (disabled by default). Example:

```
# ** Info:
example_top.DUT.inst.match_action_engine_inst.table2_MyProcessing_FiveTuple_inst.hcam_table_inst.HW_LOOKUP key =
0xfd5f0bc89aaa20101124116cc1 response = 0x06222
MATCH                               4183326
```

## About Vitis Networking P4

The About Vitis Networking P4 tab displays a list of Vitis Networking P4 document references, the IP version information, and a legal notice.

## Setting the IP Parameters from the Command Line (Vivado Console)

All of the IP Parameters can optionally be applied from the command line in the Vivado console, as an alternative to opening the IP Customization GUI. This is typically preferable for a scripted approach to re-generating a P4 design in Vivado. The easiest way to determine the syntax for these command line parameters is to use the IP Customization GUI first and then copy the resulting commands from the Vivado console or the Vivado Journal file (.jou).

For some of the per-table parameters, there are a collection of indexed parameters to facilitate the display and setting of these per-table parameters in the IP Customization GUI. However these indexed parameters can be ignored from the command line perspective. Instead, there is a consolidated parameter in the format of a Tcl Dictionary which is more convenient to modify from the command line.

The first step is to make sure that the P4\_FILE parameter has been set, so that the P4 program is already compiled and the other parameter

values are automatically populated with the values based on the P4 program. Next, the Tcl dictionary value can be read from the parameter. Then the individual per-table settings can be updated in that Tcl dictionary value before re-applying the updated parameter back to the IP. Here is an example of updating the ram\_style parameters per table in the example design forward.p4:

```
# Get the complete TCL dictionary value of parameters for the CAMs
set cam_table_params [get_property CONFIG.CAM_TABLE_PARAMS
[get_ips <p4_instance>]]
# Modify the specific parameters per-table as required
dict set cam_table_params MyProcessing.forwardIPv4 ram_style
URAM
dict set cam_table_params MyProcessing.forwardIPv6 ram_style
BRAM
# Apply the updated parameter settings
set_property CONFIG.CAM_TABLE_PARAMS $cam_table_params
[get_ips <p4_instance>]
```

The IP Customization GUI can then be opened to verify that the new parameter settings are in place as intended. Details of the other Tcl Dictionary format parameters that can be set from the command line in a similar way are given in the following table.

**Table: Tcl Dictionary Format Parameters**

Parameter Name	Identifier	Sub-parameter	Allowed Values	Notes
DIRECT_TABLE_PARAMS	control_name	ram_style	GLOBAL, URAM, BRAM, DRAM	Default is GLOBAL. Note: DRAM refers to Distributed RAM (LUTRAM)
CAM_TABLE_PARAMS	control_name	ram_style	GLOBAL, URAM, BRAM,	Default is GLOBAL

Parameter Name	Identifier	Sub-parameter	Allowed Values	Notes
			DRAM	
		clock	CAM, AXIS	Default is CAM
		mode	BCAM, STCAM, TCAM	Read-only (determined by P4 program)
		opt	GLOBAL, RAM, LOGIC	Default is GLOBAL
		cue_enable	true, false	Default is false.
		var_rate	true, false	Variable rate. Only applicable to STCAM. Default is true for STCAM
		phased	true, false	Phased-lookup. Only applicable to HBM/DDR BCAMs. Default is true for HBM/DDR BCAMs.
		cache_mode	NONE, 512, 1k,	Default is 8k

Parameter Name	Identifier	Sub-parameter	Allowed Values	Notes
			2k, 4k, 8k, 16k	
USER_METADATA_ENABLE	enable_data_field_name	input	true, false	Default is true
		output	true, false	Default is true


Refer to the CAM product guides listed in [References](#) for more details on the CAM table parameters.

## Example Designs

Vitis Networking P4 IP is delivered with the following example designs contained in the <Vivado\_install\_area>/data/ip/xilinx/vitis\_net\_p4\_v1\_0/example\_design/examples directory:

- Default

---

 **Note:** Synthesis and implementation of the default P4 is not supported.

---

- Echo
- FiveTuple
  - The table found in this example (BCAM mode) can also be implemented using HBM or DDR (see [Example Design Use \(with HBM/DDR BCAM\)](#)).
- FiveTuple\_tinycam
- Forward
- Forward\_tinycam
- Calculator
- Advanced Calculator
- Tcp\_fsm

The Vitis Networking P4 example designs contain the following blocks:

### Control

Based on the P4 file, determines the IPs in the design and programs any necessary initializations.

## Stimulus

Takes input from the pcap|user text file, converts it to an .axi file format using C DPI functions, and sends in packets.

## Checker

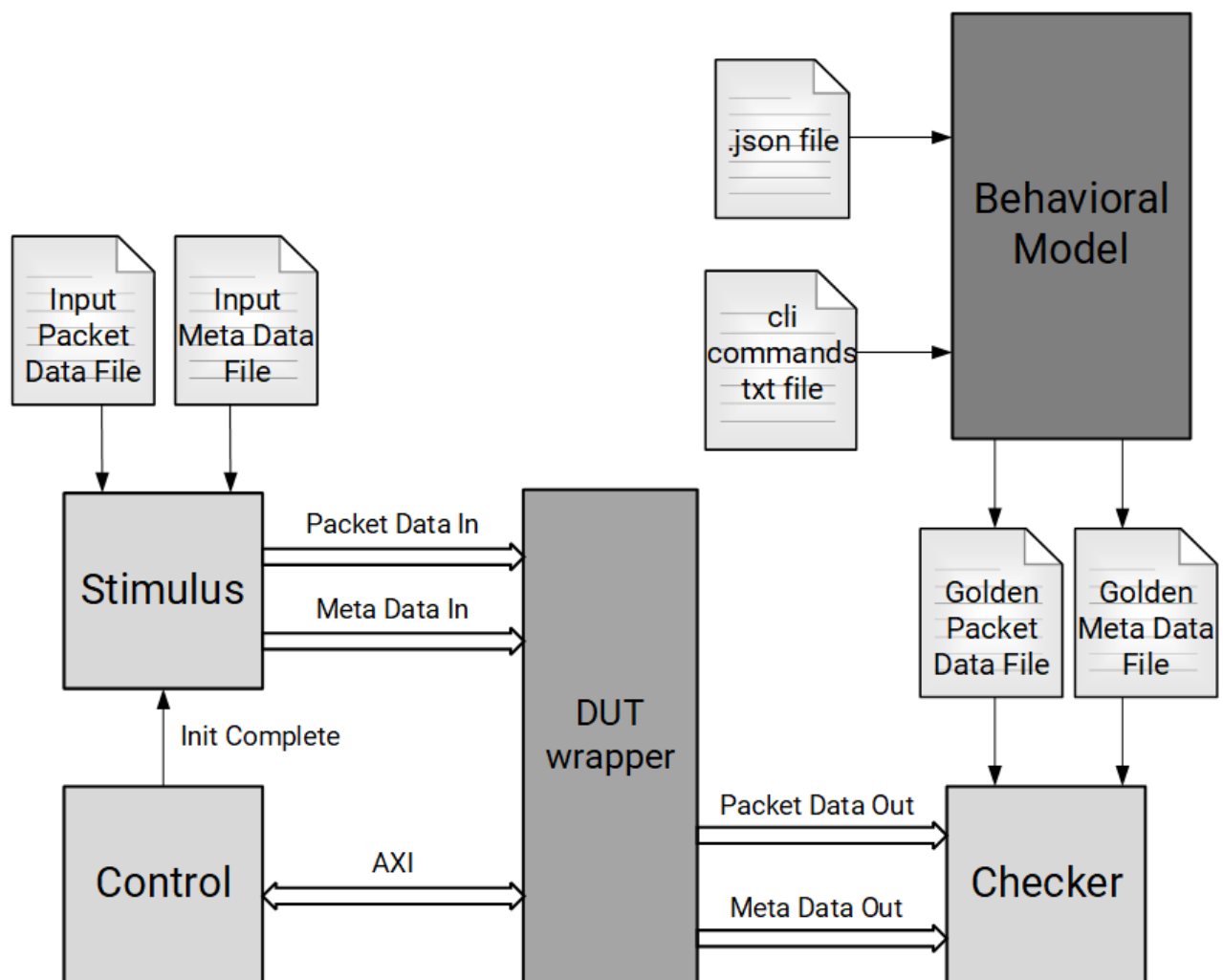
Converts the expected packet file to an .axi file format using C DPI functions and checks against expected packets produced by the Behavioral Model.

## DUT Wrapper

Contains all of the design related modules - for example, VNP4, HBM, User Externs.

The following figure shows a block diagram of the example design.

**Figure: Example Design Block Diagram**



X29369-061124



Each of the example designs (except Echo) provided with Vitis Networking P4 contains one or more tables. In the generated VNP4 IP, these tables are implemented using the table hardware that has been described previously in [Target Architecture](#). Programming these tables with a set of key-response entries is required to exercise these example designs. The CAMs can only be programmed by using their control plane software drivers written in C. The test bench provided with the example designs makes use of a SystemVerilog direct programming interface (DPI) library, which allows those control plane software drivers to be used from inside a SystemVerilog simulation environment. [DPI Simulation](#) describes in detail how DPI is used within the test bench provided with the example designs.

### cli\_commands.txt File Contents

In addition to input packet data (and optional input meta data), all of the example designs have an input cli\_commands.txt file. This file must, at a minimum, contain the run\_traffic command. For example, run\_traffic <packetname>. (Refer to the previous section, [p4bm-vitisnet-cli Application](#), for more information on the run\_traffic command).

It is also recommended that the command exit be included at the end of every cli\_commands.txt file, although this is not mandatory.

The cli\_commands.txt file can contain the following commands:

#### **table\_add**

Add an entry to the specified table

#### **table\_modify**

Modify an existing entry on the specified table

#### **table\_delete**

Delete an entry from the specified table

#### **table\_clear**

Delete all entries from the specified table

#### **reset\_state**

Delete all entries from all tables

#### **counter\_read**

Read a counter value from a specified counter extern

**counter\_write**

Update a counter value of a specified counter extern

**counter\_reset**

Reset all counter values of a specified counter extern

**run\_traffic**

Load traffic and metadata for the example

The format of each of the commands is as follows:

- `table_add` <table name> <action name> <match fields> => <action parameters> [priority]
- `table_modify` <table name> <entry handle> [action parameters]
- `table_delete` <table name> <entry handle>
- `table_clear` <table name>
- `reset_state`
- `counter_read` <counter name> <counter index>
- `counter_write` <counter name> <counter index> <byte count> <packet count>
- `counter_reset` <counter name>
- `run_traffic` <filename>

For the `run_traffic` command, the `filename` parameter specifies the input stimulus packet data and optional metadata files, where packet data is stored in the <filename>\_in.pcap|user text file and metadata is stored in the <filename>\_in.meta file. The command parameters are defined as follows:

**table\_name**

The name given to the table in the P4 program. It is case sensitive.

**entry\_handle**

The value given to reference an entry when it is added to the table

**action\_name**

The name given to the action in the P4 program. It is case-sensitive.

**match\_fields**

A space-separated list that appears in the same order as the table search key fields as specified in the P4 program. The format of the individual field depends on the match kind used for the field:

**ternary**

Fields are specified as a hexadecimal value followed by '&&&' followed by a hexadecimal mask (no white space between the numbers and the &&&). Bit positions of the mask equal to 1 are exact match bit positions and bit positions of the mask equal to 0 are wildcard bit positions.

**field\_mask**

Fields are specified in a similar way as ternary fields, but with the restriction that the only mask values permitted are all bits set to 0 or all bits set to 1.

**lpm**

Fields are specified by a hexadecimal value followed by a '/' character followed by the prefix length (0x0a010203/24 for example). The prefix length must always be in decimal (hexadecimal is not supported). A prefix length of 0 can be used, and such an entry matches any value of that search key field.

**range**

Fields are specified as a minimum hexadecimal value followed by '->' followed by a maximum hexadecimal value (with no white spaces), for example: 0->0xffffffff for a 32-bit field.

**exact**

Fields are specified using a hexadecimal value.

**unused**

Fields are specified using a hexadecimal value.

**action parameters**

A space-separated list of values that provides the parameters for the specified action. They must appear in the same order as they appear in the P4 program.

**priority**

In CAMs where multiple entries can match, the lowest priority value of the entry determines the winning response. It must only be specified when the table mode/type is STCAM or TCAM, otherwise it is omitted.

An example of a `table_add` for the following table with `exact`, `lpm`, and `field_mask` `match_fields` is as follows:

```
table test_table {
    key = {meta.port : exact;
    hdr.eth.dmac : lpm;
    hdr.eth.smac : field_mask;
    ...}
```

```
table_add test_table update_flow 0x5 0x296f5d24202a/48
0xeebb6da832d8&&0xffffffffffff...
```

Other examples of `cli_commands.txt` file entries are further explained in the following sections describing the individual example designs.

## Generated Files

The following files are generated when any of the example designs are created:

- `<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/sources_1/ip/vitis_net_p4_0/src/verilog/vitis_net_p4_0_pkg.sv`
  - This file contains details in System Verilog on general configuration settings, including table structures, actions and memory map base addresses. It also includes import definitions for the DPI functions.
- `<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/sources_1/ip/vitis_net_p4_0/src/sw/drivers/target/src/vitis_net_p4_0_defs.c`
  - This file contains details in C on general configuration settings, including table structures, actions and memory map base addresses.
- `<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/sources_1/ip/vitis_net_p4_0/src/sw/drivers/target/inc/vitis_net_p4_0_defs.h`
  - This is the C header file.

Generally, in any project where Vitis Networking P4 is instantiated within IP Integrator, the same files are generated in the following locations:

- `<project_name>/<project_name>.gen/sources_1/bd/<bd_name>/ip/<inst_name>/src/verilog/<inst_name>_pkg.sv`
- `<project_name>/<project_name>.gen/sources_1/bd/<bd_name>/ip/<inst_name>/src/sw/drivers/target/src/<inst_name>_defs.c`
- `<project_name>/<project_name>.gen/sources_1/bd/<bd_name>/ip/<inst_name>/src/sw/drivers/target/inc/<inst_name>_defs.h`

Similarly, in any project where Vitis Networking P4 is instantiated within IP catalog, the same files are generated in the following locations:

- `<project_name>/<project_name>.gen/sources_1/ip/<inst_name>/src/verilog/<inst_name>_pkg.sv`
- `<project_name>/<project_name>.gen/sources_1/ip/<inst_name>/src/sw/drivers/target/src/<inst_name>_defs.c`
- `<project_name>/<project_name>.gen/sources_1/ip/<inst_name>/src/sw/drivers/target/inc/<inst_name>_defs.h`

## Default Example Design

The Default example is a simple pass-through design. It is the default P4 file selected when a new VNP4 instance is created.

## Echo Example Design

The Echo example is like a UDP echo server. It does nothing more than send back whatever packets are sent to it. No control plane software is required. The UDP port is set up to listen to port 0x1234, packets with a different destination port remain unmodified.

The Echo example is the only example design that uses an input metadata file, `traffic_in.meta`. The UDP destination port is set in `traffic_in.meta` as follows:

```
% Packet #1
metadata.udp_dst_port=1234 ;
```

## FiveTuple Example Design

The FiveTuple example shows how to implement the standard 5-tuple network application in P4 language. A 5-tuple system is commonly used to identify key requirements for creating a secure and bidirectional TCP/IP or UDP/IP network connection between two or more machines. This application uses the source IP address and TCP/UDP port number, destination IP address and TCP/UDP port number and IP protocol to perform an exact-match based classification.

In the FiveTuple test, eight packets (contained in traffic\_in.user) are input to the Vitis Networking P4 Parser. The FiveTuple.p4 file describes one BCAM table with a key width of 104 bits.

The FiveTuple example also instantiates two counter externs - PacketCounter and ByteCounter. The corresponding cli\_commands.txt file contains some examples of these counter externs being read (with clear-on-read being enabled by default).

cli\_commands.txt File

The cli\_commands.txt file contains the keys and responses corresponding to the first four packets, the first entry is shown in the following example:

```
# Table FiveTuple, Entry 1
# key:[ ipv4.src=fd5f0bc8 ipv4.dst=9aaa2010 ipv4.protocol=11
src_port=2411 dst_port=6cc1 ]
# response:[ pcp=1 cfi=1 vid=111 action=InsertVLAN ]
table_add FiveTuple InsertVLAN 0xfd5f0bc8 0x9aaa2010 0x11
0x2411 0x6cc1 => 0x1 0x1 0x111
```

where FiveTuple represents the table name and InsertVLAN represents the action name. The fields of the key are listed in the same order as that of the table defined in FiveTuple.p4:

```
table FiveTuple {
    key
        = { hdr.ipv4.src      : exact;
            hdr.ipv4.dst      : exact;
            hdr.ipv4.protocol : exact;
            table_key_sport   : exact;
            table_key_dport   : exact; }
```

The same applies to the response data:

```
action InsertVLAN(bit<3> pcp, bit<1> cfi, bit<12> vid)
```

When the Example Design is created, the following file is produced:

```
<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/sources_1/ip/  
vitis_net_p4_0/src/verilog/vitis_net_p4_0_pkg.sv
```

This contains details of the table structures used in the test. It includes a list of associated actions per table, for example:

```
        // Table 'FiveTuple' Action list  
const XilVitisNetP4Action  
XilVitisNetP4TableActions_FiveTuple[] =  
    '{  
        XilVitisNetP4Action_InsertVLAN,  
        XilVitisNetP4Action_NoAction  
    };}
```

## C Driver

The Example Design calls a C driver function called `XilVitisNetP4BcamInsert` (from within the `table_add` task in `<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/sources_1/ip/vitis_net_p4_0/src/verilog/vitis_net_p4_0_pkg.sv`) to add the entry to the table.

## Function

The Match-Action block modifies the header based on the following:

- If there is a match in a UDP or TCP packet and it does not already contain a VLAN header, a VLAN header is inserted.
- If there is a match in a UDP or TCP packet and it already contains a VLAN header, a QinQ header is inserted.

In the FiveTuple example, packets 1 and 2 have VLAN headers inserted, packets 3 and 4 have QinQ headers inserted and packets 5-8 are not modified. Golden reference output data from the Behavioral Model is stored in a file (`traffic_out.user`) and is compared against the output of the Deparser.

## FiveTuple\_tinycam Example Design



The FiveTuple\_tinycam example is the same as the FiveTuple example, except for the size of the table (32 for FiveTuple\_tinycam versus 8192 for FiveTuple), which results in a TinyBCAM table being used instead of a regular BCAM table.

## Forward Example Design

The Forward example exemplifies the implementation of the core of an IPv4/IPv6 network switch. The IP destination address is used to perform an LPM search to determine the port to where the packet needs to be redirected. The IPv6 table is setup to be implemented with a ternary CAM, the IPv4 table with a semi-ternary CAM.

In the case of IPv4 packets, the incoming IPv4 checksum is verified using the Checksum Extern. If there is a hit in the IPv4 table, then the metadata port field is updated and TTL header field is decremented. The IPv4 checksum is then updated based on the change to the TTL field value using a second Checksum Extern instance.

In the case of IPv6 packets, the hop\_limit field is decremented in the packet.

If there are any error conditions (parsing errors, non-IP or incomplete packets, IPv4 packets with invalid checksum, IPv4/6 packets with null hop limit, or IPv4/6 packets with an unsupported destination address), the packet is dropped.

In the Forward example, fifteen packets (contained in traffic\_in.user) are input to the Vitis Networking P4 Parser. The Forward.p4 file describes two tables:

- TCAM mode with a key width of 32 bits and num\_masks of 16. This gets implemented as an STCAM table (see [Compiler Table Selection](#)).
- STCAM mode with a key width of 128 bits. This gets implemented as a TCAM table (see [Compiler Table Selection](#)).

### cli\_commands.txt File

The cli\_commands.txt file contains the priorities, masks, keys, and responses corresponding to three IPv4 packets and three IPv6 packets respectively. The following is an example of one IPv6 entry:

```
# Table forwardIPv6, Entry 1
```

```
# key:[ ipv6.dst=9f2f01913d8b6437864e0b6791a8ba64 ]
# response:[ port=14 action=forwardPacket ]
table_add forwardIPv6 forwardPacket
0x9f2f01913d8b6437864e0b6791a8ba64/128 => 0x14 0
```

where forwardIPv6 represents the table name and forwardPacket represents the action name. The notation <key>/<size of mask> is used to represent which bits of the key are maskable. In the example above all 128 bits are masked. The field following the response field corresponds to the priority field, which is set to 0 in the above example.

In this example there is only one field per key, as defined in Forward.p4:

```
table forwardIPv6 {
    key                = { hdr.ipv6.dst : lpm; }
```

The same applies to the response data:

```
action forwardPacket(bit<9> port)
```

When the Example Design is created, the following file is produced:

```
<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/
sources_1/ip/vitis_net_p4_0/src/verilog/vitis_net_p4_0_pkg.sv
```

This contains details of the table structures used in the test. It includes a list of associated actions per table, for example:

```
// Table 'forwardIPv6' Action list
const XilVitisNetP4Action
XilVitisNetP4TableActions_forwardIPv6[] =
    '{
        XilVitisNetP4Action_forwardPacket,
        XilVitisNetP4Action_dropPacket
    };
```

## C Driver

The Example Design calls two C driver functions called XilVitisNetP4StcamInsert and XilVitisNetP4TcamInsert (from within the table\_add task in <proj\_location>/vitis\_net\_p4\_0\_ex/vitis\_net\_p4\_0\_ex.gen/sources\_1/ip/vitis\_net\_p4\_0/src/verilog/vitis\_net\_p4\_0\_pkg.sv) to add the entries to the table.

## Function

The Match-Action block forwards the packet based on the following:

- If there is a match in an IPv4 packet, the packet is forwarded to the Deparser.
- If there is a match in an IPv6 packet, the packet is forwarded to the Deparser.
- If there is no match in the tables, or the packet is not IPv4 nor IPv6, then the packet is dropped by Vitis Networking P4.

In the Forward example, packets 1, 3 and 9 are forwarded IPv4 packets, and packets 4, 6 and 13 are forwarded IPv6 packets, resulting in the golden reference output file from the Behavioral Model (traffic\_out.user) containing six packets in total.

In the Forward example, a 9-bit wide output metadata port called port is defined. This metadata port is set from the response of the table entries. Golden reference output packet metadata from the Behavioral Model is stored in a file (traffic\_out.meta).

## Forward\_tinycam Example Design

The Forward\_tinycam example is the same as the Forward example, except for the size of the tables (32 for both Forward\_tinycam tables versus 1024 for both Forward tables), which results in two TinyCAM tables being used instead of a TCAM and STCAM table.

## Calculator Example Design

The Calculator example implements a simple calculator based on a custom protocol carried over Ethernet. The device uses the operation ID to perform an exact-match (with direct matching) and execute the requested operation. Packets with an unknown operation ID are dropped. The packet containing the result of the operation is sent back out of the same port it came in on, while swapping the Ethernet source and destination addresses. The supported operations are: arithmetic addition, subtraction and multiplication, as well as bit-wise AND, OR and XOR. In the Calculator test, nine packets (contained in traffic\_in.user) are input to the Vitis Networking P4 Parser. The Calculator.p4 file describes one direct table.

## cli\_commands.txt File

The cli\_commands.txt file contains the keys and responses corresponding to the first six packets. The following is an example of one entry:

```
# Table calculate, Entry 1
# key:[ p4calc.operation=6 ]
# response:[ action=operation_add ]
table_add calculate operation_add 0x6 =>
```

where calculate corresponds to the table name and operation\_add corresponds to the action name.

In this example there is only one field in the key, as defined in calculator.p4:

```
table calculate {
    key = {
        hdr.p4calc.operation : direct;
    }
}
```

When the example design is created, the following file is produced:

<proj\_location>/vitis\_net\_p4\_0\_ex/vitis\_net\_p4\_0\_ex.gen/sources\_1/ip/vitis\_net\_p4\_0/src/verilog/vitis\_net\_p4\_0\_pkg.sv

This contains details of the table structures used in the test. It includes a list of associated actions per table, for example:

```
// Table 'calculate' Action list
XilVitisNetP4Action XilVitisNetP4TableActions_calculate[]
=
'{
    XilVitisNetP4Action_operation_add,
    XilVitisNetP4Action_operation_sub,
    XilVitisNetP4Action_operation_mult,
    XilVitisNetP4Action_operation_and,
    XilVitisNetP4Action_operation_or,
    XilVitisNetP4Action_operation_xor,
    XilVitisNetP4Action_operation_drop
};
```

## C Driver

The Example Design calls a C driver function called `XilVitisNetP4DcamInsert` (from within the `table_add` task in `<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/sources_1/ip/vitis_net_p4_0/src/verilog/vitis_net_p4_0_pkg.sv`) to add the entry to the table.

## Function

The Match-Action block performs the following:

- If an unknown action is specified or the header is invalid, the packet is dropped.
- If there is a match, the source and destination addresses are swapped.
- If there is a match, the requested operation is performed on Operand A and Operand B and the result field is updated.

In the Calculator example, packets 1 to 6 have source/destination addresses swapped, actions performed and results inserted. Packets 7, 8, and 9 are dropped because of an unknown action/invalid header. Golden reference output data from the Behavioral Model output is stored in a file (`traffic_out.user`) and is compared against the output of the Deparser.

## Advanced Calculator Example Design

The Advanced Calculator example is based on the Calculator Example, but with two additional operations implemented with user externs: arithmetic division and square-root.

There are user extern structures defined in the generated SystemVerilog package file in Vitis Networking P4:

`<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/sources_1/ip/vitis_net_p4_0/src/verilog/vitis_net_p4_0_pkg.sv`

These structures can facilitate the connections of the user extern I/O ports, as can be seen in

`/<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex/imports/example_top.sv`

For example:

```
vitis_net_p4_0_pkg::USER_EXTERN_IN_T    user_extern_in;  
vitis_net_p4_0_pkg::USER_EXTERN_VALID_T user_extern_in_valid;
```

```
vitis_net_p4_0_pkg::USER_EXTERN_OUT_T    user_extern_out;  
vitis_net_p4_0_pkg::USER_EXTERN_VALID_T  
user_extern_out_valid;
```

The different fields of the user extern I/O structures can then be accessed by name (based on the naming in `calculator_extended.p4` file), as can be seen when connecting to the `calc_divide_ip` module:

```
calc_divide_ip calc_divide_ip_inst (  
    .aclk (s_axis_aclk),  
    .s_axis_divisor_tvalid  
    (user_extern_out_valid.calc_divide),  
    .s_axis_divisor_tdata  
    (user_extern_out.calc_divide.divisor),  
    .s_axis_dividend_tvalid  
    (user_extern_out_valid.calc_divide),  
    .s_axis_dividend_tdata  
    (user_extern_out.calc_divide.dividend),  
    .m_axis_dout_tvalid      (user_extern_in_valid.calc_divide),  
    .m_axis_dout_tdata  
    ({user_extern_in.calc_divide.quotient,  
  
    user_extern_in.calc_divide.reminder}))  
);
```

Alternatively, instead of using the aforementioned structures, parameters defined in `vitis_net_p4_0_pkg.sv` can be used to access the relevant bits of the user extern I/O ports, for example:

```
localparam USER_EXTERN_OUT_CALC_SQUARE_ROOT_WIDTH = 32;  
localparam USER_EXTERN_OUT_CALC_SQUARE_ROOT_MSB   = 95;  
localparam USER_EXTERN_OUT_CALC_SQUARE_ROOT_LSB   = 64;
```

## Tcp\_fsm Example Design

This example design is an over simplification of a TCP server's FSM, where communication is always initiated by the client and never by the server. The intention of this P4 file is to showcase stateful packet processing with the use of the register extern, tables and other p4 elements.

See [RFC9293](#) for more information about the TCP protocol.

The TCP FSM example design uses a table (clientDB) to store the IP and MAC address of registered clients, which can be added via the control plane. The server only replies to registered clients and only to packets used to establish and/or close the connection. The server is set up to drop incoming invalid packets or packets from any unregistered clients. When a connection is established, the server is set up to only forward incoming packet payload to metadata (CONFIG.OUTPUT\_METADATA\_FOR\_DROPPED\_PKTS must be 'true').

This example instantiates a register extern (sockReg), to store connection information about registered clients, such as connection timestamp, connection state and sent packets count.

cli\_commands.txt

The cli\_commands.txt file contains the table keys and responses corresponding to a single registered client and register reads to retrieve the stored traffic and connection information.


Function

The match-action is composed of multiple sub-control blocks:

### **TcpProcessSegment\_p**

Processes incoming TCP packet headers. It uses TCP flags to implement a state machine defined in [RFC9293](#).

---

 **Note:** Only listen, syn\_rcvd, established and last\_ack states are included.

---

### **TcpServerMain\_p**

Implements the main server's process, which in this case, is assigning the incoming packet payload of established connections to metadata.

In the example traffic input provided, only a single client is registered. It sends corresponding packets to establish the connection, sends traffic through to the server when the connection is established and closes the connection. In this example, packets from unregistered clients can also be found, and these are expected to be dropped.

## Daisy-Chain Example Design

It is possible to run an example design simulation with two VNP4 instances in a daisy-chain sequence. To run, for example, the FiveTuple Example Design in this way, the P4 file needs to be renamed to `fiveTuple_daisy_chain.p4`. When the example design is then opened (following instructions in [Example Design Use](#)), the P4 is instantiated twice in a daisy-chain sequence.

When run in this way, the traffic is passed through the behavioral model twice to generate the expected data. Any table entries are applied to both instances.


## Example Design Use

The steps to run the FiveTuple Example Design using the Vivado Simulator are as follows:

1. Open a Vivado project.
2. Instantiate the Vitis Networking P4 IP: Click IP Catalog and double-click Vitis Networking P4.
3. Select a P4 file from the FiveTuple directory `<Vivado_install_area>/data/ip/xilinx/vitis_net_p4_v1_0/example_design/examples/five_tuple`.
4. Right-click `vitis_net_p4_0` in the Sources window and click Open IP Example Design.
5. In the new Example Design Vivado project, click Run Simulation under Simulation in the panel on the left. This launches XSim simulator, the default simulator used in Vivado Design Suite.
6. The simulation is run for a maximum of 10  $\mu$ s.

The Echo, FiveTuple\_tinycam, Forward, Forward\_tinycam, Calculator, and Advanced Calculator example designs can be simulated in a similar fashion.

---

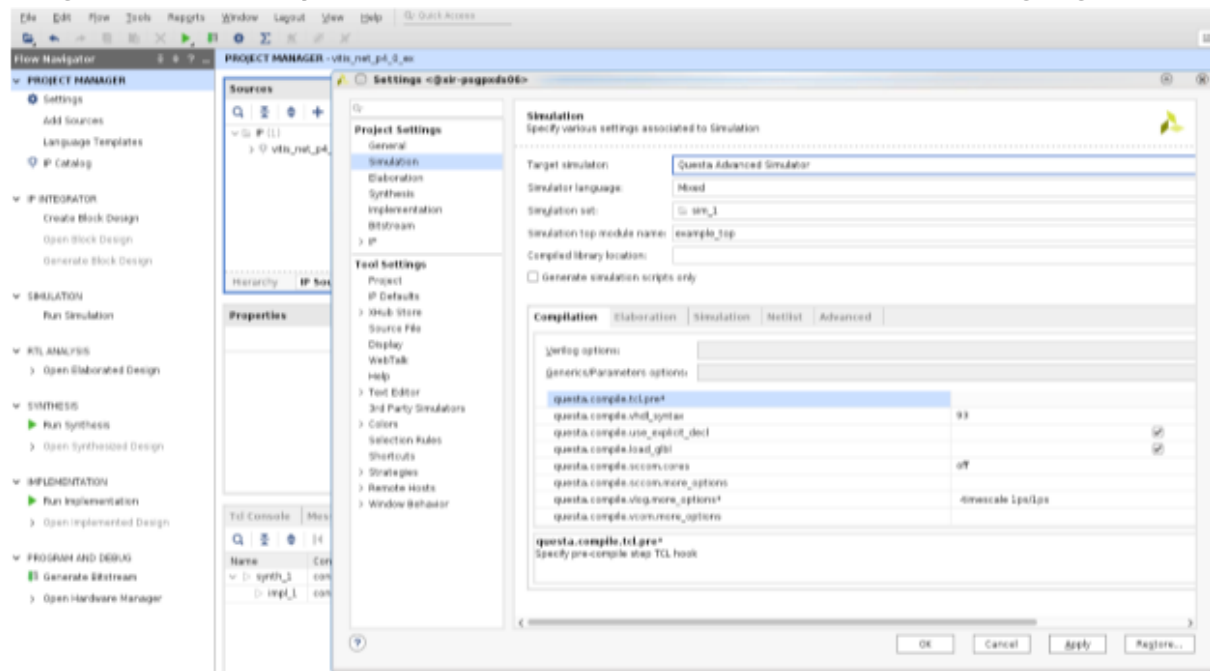
 **Note:** To run the above example design simulation using Questa, Xcelium or VCS simulator, the following two steps should be completed between steps 4 and 5:

---

1. In the new Example Design Vivado project, change the target simulator: Click Settings under PROJECT MANAGER in the panel on



the left, select Simulation under Project Settings, and select the target simulator you want to use, as shown in the following figure.



2. Verify that the compiled library location is set correctly to point to the correct pre-compiled simulation libraries. See *Vivado Design Suite User Guide: Logic Simulation (UG900)* for instructions on how to compile simulation libraries.

### Example Design Use (with HBM/DDR BCAM)


The steps to run the FiveTuple example design using HBM BCAM with the Questa simulator are as follows:

1. Open a Vivado project and select a device that has HBM DRAM.
2. Instantiate the Vitis Networking P4 IP: Click IP Catalog, then double-click Vitis Networking P4.
3. Select a P4 file from the FiveTuple directory <Vivado\_install\_area>/data/ip/xilinx/vitis\_net\_p4\_v1\_0/example\_design/examples/five\_tuple.
4. In the Tables tab drop-down menu under Ram Style, select HBM. The Memory Resources entry should update to show the number of HBM pseudo channels used.
5. Right-click vitis\_net\_p4\_0 in the Sources window and click Open IP Example Design.
6. In the new Example Design Vivado project, change the target simulator: Click Settings under PROJECT MANAGER in the panel on the left, select Simulation under Project Settings, and select the


target simulator you want to use.

7. Verify that the compiled library location is set correctly to point to the correct pre-compiled simulation libraries. See the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for instructions on how to compile simulation libraries.
8. Click Run Simulation under Simulation in the panel on the left. This launches the Questa simulator.
9. The simulation is run for a maximum of 10 us.


---

 **Note:** To run this example design simulation using VCS simulator, select the Verilog Compiler simulator (VCS) instead of the Questa simulator.

---

 **Note:** To run this example design simulation using DDR instead of HBM, in step 4, select 'DDR' instead of 'HBM'.

---

 **Note:** If running HBM example designs using the Questa simulator, the following property needs to be set: `set_property -name {questa.elaborate.vopt.more_options} -value {-inlineFactor=0} -objects [get_filesets sim_1]`

---

## Launch Simulation

Simulations should be run as normal for any IP block once the RTL output for the design has been generated. For information on running simulations on the Vivado Design Suite project, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

## Run Synthesis and Implementation

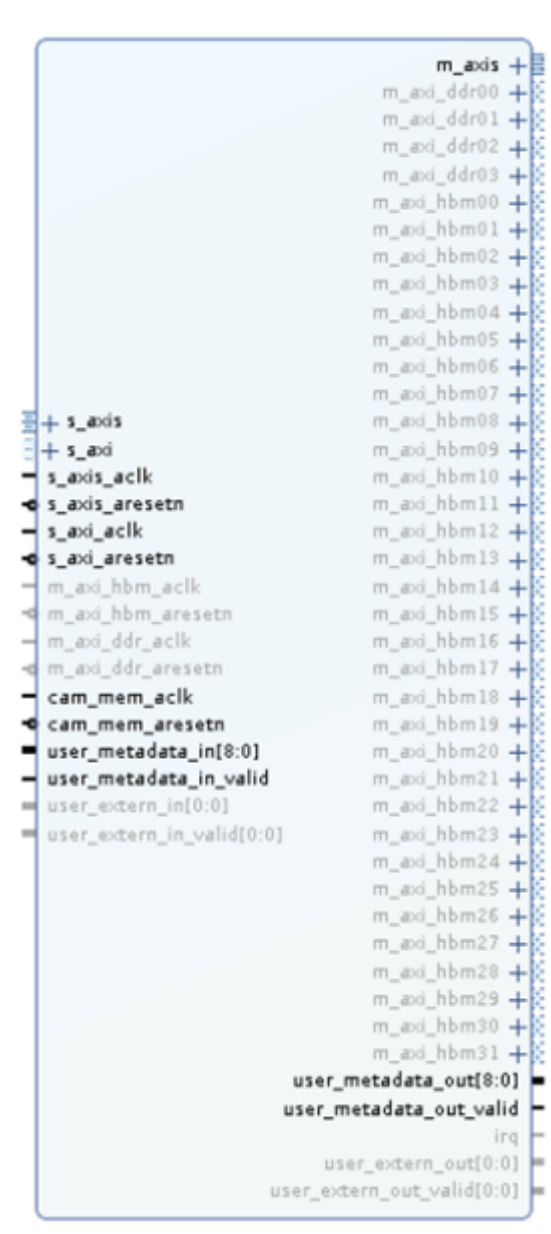
Synthesis and implementation should be run as normal for any IP block. For information on running synthesis and implementation on the Vivado Design Suite project, see the *Vivado Design Suite User Guide: Synthesis* (UG901) and the *Vivado Design Suite User Guide: Implementation* (UG904).

## Vitis Networking P4 Tool Interface

The AMD Vitis™ Networking P4 Tool default interfaces are shown in the

following figure and described in the following table.

**Figure: Vitis Networking P4 Tool Interface**



**Table: Vitis Networking P4 Tool Interface**

Name	I/O	Description
s_axis	I/O	Slave AXI4-Stream interface carrying input packet data. Data bus width is configurable. Details can be found in <a href="#">s_axis Interface</a> .
s_axi	I/O	AXI4-Lite bus interface from the Control Plane, used for register access.

Name	I/O	Description
s_axis_aclk	I	Slave AXI4-Stream clock.
s_axis_aresetn	I	Active Low slave AXI4-Stream reset.
cam_mem_aclk	I	CAM memory clock.
cam_mem_aresetn	I	Unconnected.
s_axi_aclk	I	AXI4-Lite clock.
s_axi_aresetn	I	Active Low AXI4-Lite reset.
m_axi_hbm_aclk	I	Master AXI4-Stream clock.
m_axi_hbm_aresetn	I	Active Low HBM reset.
m_axi_ddr_aclk	I	Master AXI4-Stream clock
m_axi_ddr_aresetn	I	Active Low DDR reset.
user_metadata_in	I	Input user metadata bus. Bus width varies depending on the P4 program selected.
user_metadata_in_valid		Input user metadata valid. Asserted with first word of the corresponding packet. If not asserted, user_metadata_in is treated as all zeroes for that packet. If the axis_tuser/axis_tid/axis_tdest keywords are used, the user_metadata_in_valid port is ignored and the signal will be auto-generated internally for each packet based on the S_AXIS signals.
user_extern_in	I	Data input from user Extern interface(s). This vector is extended, depending on the number of user Extern instances.
user_extern_in_valid	I	Data in valid from user Extern interface(s). This vector is extended, depending on the number of user Extern instances (for example, a single valid bit


Name	I/O	Description
		per user Extern interface).
user_extern_in_ready	O	Data in ready to user Extern interface(s). This vector is only present if an extern in 'variable latency' mode is found in the p4 file. This vector is extended, depending on the number of User Extern instances.
m_axis	I/O	Master AXI4-Stream interface carrying output packet data. Data bus width is configurable. Details can be found in <a href="#">m_axis Interface</a> .
m_axi_hbmXX	I/O	Master AXI HBM interface(s), where XX represents the interface number in the range of 0 to 31.
m_axi_ddrXX	I/O	Master AXI DDR interface(s), where XX represents the interface number in the range of 0 to 3.
user_metadata_out	O	Output user metadata bus. Bus width varies depending on the P4 program selected.
user_metadata_out_valid	O	Output user metadata valid. Asserted with first word of the corresponding packet.
user_extern_out	O	Data output to user Extern interface(s). This vector is extended, depending on the number of user Extern instances.
user_extern_out_valid	O	Data output valid to user Extern interface(s). This vector is extended, depending on the number of user Extern instances (for example, a single valid bit per user Extern interface).

Name	I/O	Description
user_extern_out_ready		Data out ready from user Extern interface(s). This vector is only present if an extern in 'variable latency' mode is found in the p4 file. This vector is extended, depending on the number of User Extern instances.
irq	O	Interrupt signal asserted when any internal ECC interrupts from BRAM/URAM (CAMs or FIFOs) are triggered. Active High on rising-edge of s_axis_clk domain.

## s\_axis Interface

The following table shows the breakdown of the s\_axis interface signals.

**Table: s\_axis Interface**

Name	I/O	Description
s_axis_tdata		Packet data bus of configurable width.
s_axis_tvalid	I	Indicates that the master is driving a valid transfer. A transfer takes place when both s_axis_tvalid and s_axis_tready are asserted.
s_axis_tkeep	I	Indicates whether the content of the associated byte of s_axis_tdata is processed as part of the data stream. <hr/>  <b>Note:</b> This value must be set to all 1s for all words of a packet except the TLAST word, where the valid bytes must be right-justified (no invalid byte gaps). The all 0s value is not currently supported on this input interface, except for a null packet length where there are no headers to be inserted into the packet. <hr/>
s_axis_tlast	I	Indicates the boundary of the packet.


Name	I/O	Description
s_axis_tuserl		User-defined sideband information transmitted alongside the data stream. Only present if the axis_tuser keyword appears as a field name within the user metadata structure of a P4 program. It is only valid for the first word of an incoming packet to the Vitis Networking P4 IP, validated by s_axis_tvalid.
s_axis_tdestl		Provides routing information for the data stream. It is only present if the axis_tdest keyword appears as a field name within the user metadata structure of a P4 program. It is only valid for the first word of an incoming packet to the Vitis Networking P4 IP, validated by s_axis_tvalid.
s_axis_tid	I	Data stream identifier that indicates different streams of data. It is only present if the axis_tid keyword appears as a field name within the user metadata structure of a P4 program. It is only valid for the first word of an incoming packet to the Vitis Networking P4 IP, validated by s_axis_tvalid.
s_axis_tready	O	Indicates that the slave can accept a transfer in the current cycle.

## m\_axis Interface

The following table shows the breakdown of the m\_axis interface signals.

**Table: m\_axis Interface**

Name	I/O	Description
m_axis_tdataO		Packet data bus of configurable width.
m_axis_tvalidO		Indicates that the master is driving a valid transfer. A transfer takes place when both m_axis_tvalid and m_axis_tready are asserted.

Name	I/O	Description
m_axis_tkeepO		Indicates whether the content of the associated byte of s_axis_tdata is processed as part of the data stream. For metadata output of dropped packets, m_axi_tkeep = 0. This is described in the <i>User Metadata Interface section</i> . <hr/>  <b>Note:</b> There no are invalid byte gaps in the middle of the data bus. <hr/>
m_axis_tlast O		Indicates the boundary of the packet.
m_axis_tuser O		User-defined sideband information transmitted alongside the data stream. It is only present if the axis_tuser keyword appears as a field name within the user metadata structure of a P4 program. It is valid beginning with the first word of an outgoing packet from the Vitis Networking P4 IP. The field remains at the same value for the duration of each packet.
m_axis_tdest O		Provides routing information for the data stream. It is only present if the axis_tdest keyword appears as a field name within the user metadata structure of a P4 program. It is valid beginning with the first word of an outgoing packet from the Vitis Networking P4 IP. The field remains at the same value for the duration of each packet.
m_axis_tid O	O	Data stream identifier that indicates different streams of data. It is only present if the axis_tid keyword appears as a field name within the user metadata structure of a P4 program. It is valid beginning with the first word of an outgoing packet from the Vitis Networking P4 IP. The field remains at the same value for the duration of each packet.
m_axis_tready		Indicates that the slave can accept a transfer in the current cycle.

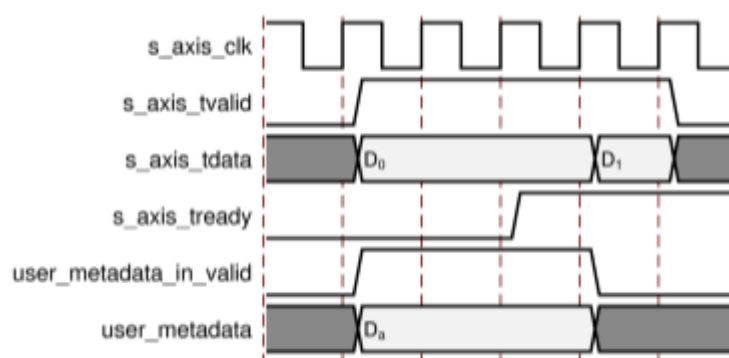


# User Metadata Interface

There are no separate back-pressure signals for the user metadata interfaces. These interfaces follow the corresponding packet TREADY signals of the packet AXIS interfaces, for example:

If `s_axis_tvalid` and `user_metadata_in_valid` are asserted for the first word of a packet, and `s_axis_tready` = 0, then `s_axis_tvalid` and `user_metadata_in_valid` should remain asserted until at least the cycle after `s_axis_tready` = 1, as shown in the following timing diagram.

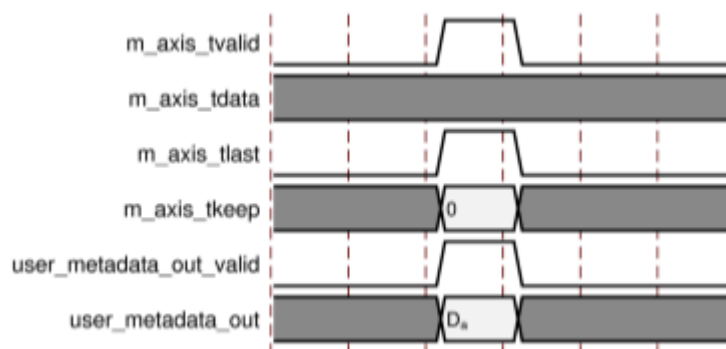
**Figure: TREADY Timing Diagram**



Similarly, if `m_axis_tready` = 0 during the first word of a packet, then `m_axis_valid` and `user_metadata_out_valid` should remain asserted until the cycle after `m_axis_tready` = 1, at which time `user_metadata_out_valid` is deasserted if there are further data words for the same packet.

There is an option to allow metadata to be output for dropped packets. When this option is enabled, the dropped packet becomes a “zero-length” or “null” packet. This is a single-beat packet with `m_axis_tvalid` = 1, `m_axis_tlast` = 1 and `m_axis_tkeep` = 0. The metadata output shall be aligned with this single-beat packet as shown in the following timing diagram.

**Figure: Metadata Output Timing Diagram**



If this option is disabled, there are no `m_axis_tvalid` or `user_metadata_out_valid` signals for the dropped packet.

If the keywords `axis_tuser`, `axis_tid` or `axis_tdest` are used in the metadata structure, the signals are mapped to/from the TUSER/TID/TDEST ports. In this case there are still placeholders for these fields in the `user_metadata_in` structure but these are ignored by Vitis Networking P4. The values are output in both the `user_metadata_out` structure and the TUSER/TID/TDEST ports. These keywords can be used with simple bit-string definitions or with a sub-struct definition, to further breakdown the TUSER/TID/TDEST into multiple sub-fields. This is demonstrated in the following example:

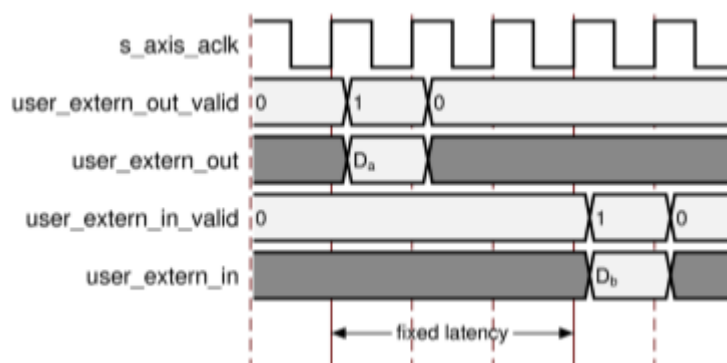
```
struct tuser_format {
    bit<8>  field_a;
    bit<16> field_b;
    bit<16> field_c;
}

// User metadata structure
struct metadata {
    bit<9>      dummy3;
    bit<29>     axis_tid;
    bit<7>      dummy2;
    bit<13>     axis_tdest;
    bit<5>      dummy1;
    tuser_format axis_tuser; // still using the same
    "axis_tuser" keyword here, but the type is a struct instead
    of a bit<> format
    bit<1>      dummy0;
}
```

## User Extern Interface

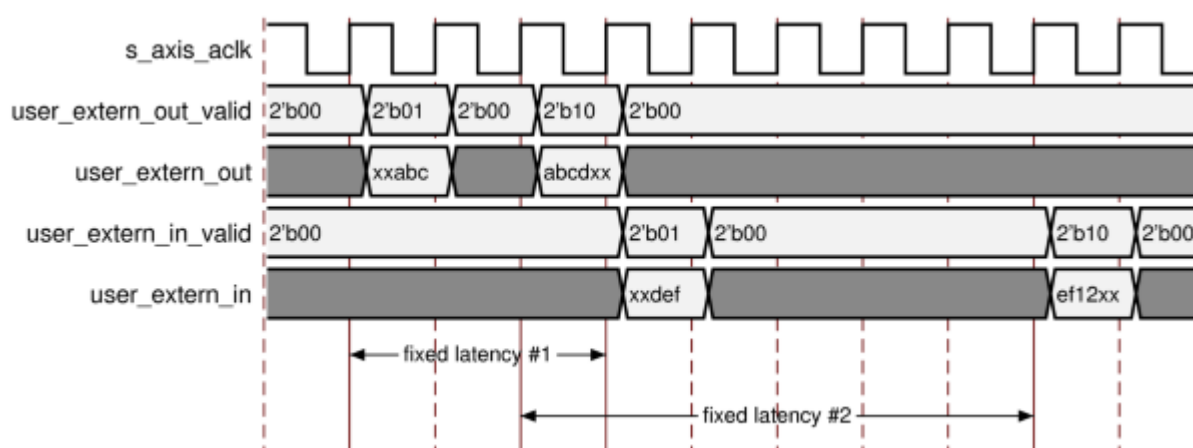
The user extern interface provides a simple data and valid signal for input and output, synchronous to the `s_axis_aclk` domain. A single user extern waveform (in fixed latency mode) is shown in the following figure.

**Figure: Single User Extern Waveform**



These vector signals are extended in width to cover all user extern instances from the user's P4 program (for example, a single valid bit per user extern instance). The generated SystemVerilog package file includes a set of constants and a structure definition to interpret the bit definition of these signals based on the P4 program names (see [Generated Files](#) for more information). A waveform displaying two user externs (in fixed latency mode) is shown in the following figure.

**Figure: Two User Externs Waveform**



For every packet that executes the `apply` method of a user Extern instance, there is a corresponding assertion of the `user_extern_out_valid` signal for one clock cycle. Then after a fixed number of clock cycles (as specified in the `fixed_latency` value of the User Extern instance in the P4 program), the corresponding bit of the

`user_extern_in_valid` signal is expected to be asserted for one clock cycle.

There is support for back-pressure signaling on the User Extern interface only when the defined p4 instance is defined as 'variable latency'. Also note that the inter-packet gap might not be maintained from the `S_AXIS` interface to the user Extern interface. For example, it is possible to get back-to-back assertions on `user_extern_out_valid` even if the worst case is a new packet every two cycles on the `S_AXIS` interface. This is because of variable latency through the Parser.

## Clocks

The frequencies of the Vitis Networking P4 IP interface clocks are not fixed, they can be defined by the user. The following are some guidelines to follow:

- Set the AXI4-Lite clock (`s_axi_aclk`) frequency to 100 MHz or below, to ensure reliable timing closure.
- Set the AXI4-Stream clock (`s_axis_aclk`) frequency to 300 MHz or below, to ensure reliable timing closure.
- The CAM Memory clock (`cam_mem_aclk`) must be derived from the same clock source as the AXI4-Stream clock. It can be any value up to a maximum of 600 MHz.

## Resets

The following are the reset signals used in the Vitis Networking P4 IP:

### **`s_axis_aresetn`**

Active-Low slave AXI4-Stream reset.

### **`s_axi_aresetn`**

Active-Low AXI4-Lite reset. It resets the AXI4-Lite interface signaling but it does not reset any of the register maps (such as Statistics Registers and table entries).

### **`m_axi_hbm_aresetn`**

Active-Low HBM reset. This is the DRAM memory interface reset. It resets all FSMs that manage memory access, pending and uncompleted memory transactions.

# Back-Pressure

Vitis Networking P4 could issue back-pressure on the S\_AXIS interface by bringing the TREADY signal Low. This can happen in the following scenarios:

- TREADY is Low at the M\_AXIS interface.
- Back-pressure is generated within the Deparser because of header insertions extending a packet's length.
- Back-pressure is generated by the CAMs if they are using TDM and if the specified Packet Rate is exceeded and causing excessive table lookup requests (see [Top Level Settings](#) for a description of the Packet Rate parameter).
- Back-pressure is generated by the Packet Rate Limiter (when enabled) to prevent the input stream exceeding the specified packet rate.
- Back-pressure is generated for a few clock cycles post `s_axis_aresetn`.

When there are CAMs present in the design, it takes a few clock cycles to stop back-pressure after it has been applied. If CAMs are not present, back-pressure stops instantly.

## Latency

Vitis Networking P4 can have a variable latency because of these factors:


- Back-pressure (see [Back-Pressure](#)).
- Varying levels of parsing.
  - If a packet has a small number of headers to be parsed it can have a lower latency through Vitis Networking P4.
  - If a packet has a larger number of headers to be parsed it can have a higher latency through Vitis Networking P4.

The calculated latency figure (described in [Vitis Networking P4 IP](#)) is based on the worst-case latency of the Parser, but the best case situation in terms of back-pressure. A preview of the latency in the system is available in the GUI before RTL generation or synthesis.

An example of how to calculate the number of clock cycles required to

flush the Vitis Networking P4 pipeline of all packets is as follows:

---

 **Note:** In this example the assumption is made that `m_axis_tready` is held High to flush out the pipeline and that `s_axis_tvalid` is held Low to avoid new data entering the pipeline. In this case, the calculated latency value can be treated as the maximum latency, the only exception being any header insert operations.

---

If the P4 program is inserting new headers into the outgoing packets, some clock cycles need to be added to the calculated latency value. For each header inserted, the header size is divided by the packet bus width and rounded up to the nearest integer. This is *Calculation A* in the following calculation.

The number of packets that can be in the pipeline at any given time (based on the calculated latency, the minimum packet size, the maximum packet rate, etc.) should also be considered. This is *Calculation B* in the following calculation.

This number is then multiplied by the extensions to the latency per packet due to header inserts to give the total extensions to the latency due to header inserts. This is *Calculation C* in the following calculation. The final result is achieved by adding the calculated latency to the result of Calculation C as shown in Calculation D.

For example:

- Calculated Latency from the Vitis Networking P4 GUI = 20 clock cycles
- Packet Bus Width = 64-bit
- Packet Rate = AXIS Clock Frequency
- Minimum Packet Size = 64 bytes
- P4 program can insert two new headers to the outgoing packets
  - Header A = 48-bit
  - Header B = 136-bit

### Calculation A

Calculate the possible extensions to the latency per packet due to header inserts:

- Header A =  $\text{ceil}(48/64) = 1$  clock cycle
- Header B =  $\text{ceil}(136/64) = 3$  clock cycles
- Total =  $3+1 = 4$  clock cycles

### Calculation B

Calculate number of packets (worst case) in the pipeline at a given time:

- $\text{ceil}(\text{calculated latency} / (\text{Minimum Packet Size} / \text{Packet Bus Width})) = \text{ceil}(20 / (64 * 8 / 64)) = 3 \text{ packets in progress}$

### Calculation C

Total Extensions to the latency due to header inserts =  $3 * 4 = 12$  clock cycles


### Calculation D

In this example, the maximum time to flush out the pipelines =  $20 + 12 = 32$  clock cycles.

## Register Map

The Vitis Networking P4 IP register map is located in the generated file, `<inst_name>_defs.h`. See [Generated Files](#) for the path to this generated file. The memory contents of the Direct Tables and TinyBCAM/TinyTCAM can be updated by means of AXI4-Lite accesses, that of the other CAMs cannot. The other CAM's updates are handled by the CAM driver. The Tiny CAM and Direct table AXI4-Lite interfaces support read and write operations to add, update and delete table entries. These register maps are shown in the following tables, followed by the register maps for the Counter and Register Externs.

---

 **Note:** Memory map details of the other CAMs are not published. Due to the algorithmic nature of these CAMs, the correct translation to memory mapped registers can only be accessed using their software APIs.

---

**Table: Direct Tables Register Map**

Register	Offset Range	Description
0x00	[31:0]	Control Register:

Register	Offset Range	Description
		<ul style="list-style-type: none"> <li>• RD flag (bit 0): when asserted, a read operation is triggered. Automatically cleared when the read has completed and data is available in the data register(s).</li> <li>• WR flag (bit 1): when asserted, a write operation is triggered. Automatically cleared when the write has completed.</li> <li>• Reset All (bit 2): when asserted, all entries in the table have their EntryInUse flag cleared to 0. Automatically cleared when the reset has completed.</li> <li>• Debug mode (bit 29): <ul style="list-style-type: none"> <li>◦ '0' = allows normal use of the table entry read/write registers.</li> <li>◦ '1' = causes the table entry read/write registers to update automatically with every data plane lookup operation.</li> </ul> </li> <li>• Debug captured (bit 30): <ul style="list-style-type: none"> <li>◦ '1' = indicates that a data plane lookup has occurred while debug mode was enabled such that the details are captured in the table entry read/write registers.</li> <li>◦ '0' = no data plane lookup operations have been captured yet.</li> </ul> </li> <li>• Hit/miss flag (bit 31): must be set to 1 when an entry is added or updated and set to 0 when an entry is deleted. Read and Write access. In debug mode, this register is automatically updated with each data plane lookup to reflect the hit/miss status. Default value: 0.</li> <li>• Bits from 3 to 28 are reserved.</li> </ul>
0x04	[31:0]	Address Register. This register must be updated prior to the assertion of a read or write flag. In




Register	Offset Range	Description
		debug mode, this register is automatically updated with each data plane lookup key.
0x08	[31:0]	<p>ECC Control Register</p> <ul style="list-style-type: none"> <li>• [0]: Inject Single-bit error</li> <li>• [1]: Inject Double-bit error</li> <li>• [2]: Disable ECC Scrubbing</li> <li>• [31]: ECC Enable (read-only)</li> </ul> <p>All other bits are reserved.</p>
0x0C	[31:0]	Lookup count. Clear on read.
0x10	[31:0]	Hit count. Clear on read.
0x14	[31:0]	Miss count. Clear on read.
0x18	[31:0]	Single bit error count. Clear on read.
0x1C	[31:0]	Double bit error count. Clear on read.
0x20	[31:0]	Last ECC errored address (Single-bit error).
0x24	[31:0]	Last ECC errored address (Double-bit error)
0x28-0x3F	[31:0]	Reserved
0x40	[31:0]	<p>Data Register (part 1). This register must be updated prior to the assertion of the write flag. The contents of this register are automatically updated after a read operation. In debug mode, this register is automatically updated with the response value for any data plane lookup hit. If RESPONSE_WIDTH value is greater than 32 bits, consecutive registers need to be used. This set of registers extend to the required total response width.</p>
0x44	[31:0]	Data register (part 2).

Register Offset	Range	Description
...	[31:0]	...
0x1FFC	[31:0]	Data register (part 2032).

### Table: TinyBCAM/TinyTCAM Register Map

Register	Offset	Access	Description
0x00	[31:0]	RW	Control Register:

Register	Offset	Access	Description
			<ul style="list-style-type: none"> <li>• RD flag (bit 0): when asserted, a read operation is triggered. Automatically cleared when the read has completed and the data is available in the data register(s).</li> <li>• WR flag (bit 1): when asserted, a write operation is triggered. Automatically cleared when the write has completed.</li> <li>• Reset flag (bit 2): when asserted, all CAM entries in a table are reset.</li> <li>• Bits from 3 to 28 are reserved.</li> <li>• Debug mode (bit 29): <ul style="list-style-type: none"> <li>◦ '0' = allows normal use of the table entry read/write registers.</li> <li>◦ '1' = causes the table entry read/write registers to be updated automatically with every data plane lookup operation.</li> </ul> </li> <li>• Debug captured (bit 30): <ul style="list-style-type: none"> <li>◦ '1' = indicates that a data plane lookup has occurred while debug mode was enabled such that the details are captured in the table entry read/write registers.</li> <li>◦ '0' = no data plane lookup operations have been captured yet.</li> </ul> </li> <li>• EntryInUse flag (bit 31): must be set to 1 when an entry is added or updated and set to 0 when an entry is deleted. Read and Write access. In debug mode, this register is automatically updated with each data plane lookup to reflect the EntryInUse status. Default value: 0.</li> </ul>
0x04	[31:0]	RW	Entry ID. This register must be updated prior to the assertion of a read or write flag.

Register	Offset	Access	Description
			<p> <b>Note:</b> The Entry ID equals the entry position in the table.</p> <p>In debug mode, this register is automatically updated with each data plane lookup hit to reflect the corresponding Entry ID.</p>
0x08	[31:0]	RO	Emulation Mode Register: Used by Software to confirm correct TinyCAM driver is instantiated i.e., TinyBCAM or TinyTCAM driver.
0x0C	[31:0]	RO	Lookup count. Clear-on-read.
0x10	[31:0]	RO	Hit count. Clear-on-read.
0x14	[31:0]	RO	Miss count. Clear-on-read.
0x18-0x3f	[31:0]	RO	Reserved.
0x40	[31:0]	RW	<p>Data Register (part 1). This register must be updated prior to the assertion of the write flag. The contents of this register are automatically updated after a read operation.</p> <p>In debug mode, this register is automatically updated with the key value for each data plane lookup request. The response value will only be updated if there is a hit.</p> <p>If the ENTRY_WIDTH value is greater than 32 bits, then consecutive registers need to be used. This set of registers extend to the required total response width.</p>
0x44	[31:0]	RW	Data Register (part 2).
...	[31:0]	RW	...
0x1FFC	[31:0]	RW	Data Register (part 2032).

**Table: Counter Extern Register Map**

Register Offset	Bit Range	Access	Description
0x00	[31:0]	RW	Control - a read/write operation is initiated with each write to this register: <ul style="list-style-type: none"> <li>• [24] Write Enable <ul style="list-style-type: none"> <li>◦ 1: Write Operation</li> <li>◦ 0: Read Operation</li> </ul> </li> <li>• [22:16] Burst Size: This field contains the burst size minus 1 e.g., value of 0 corresponds to burst size of 1</li> <li>• [15:0] Start Address</li> </ul>
0x04	[31:0]	R0	Status <ul style="list-style-type: none"> <li>• [0] Burst in progress flag</li> </ul>
0x08	[31:0]	RW	ECC Control <ul style="list-style-type: none"> <li>• [4] ECC Scrub Disable</li> <li>• [3] Collection RAM Injectdbiterr</li> <li>• [2] Collection RAM Injectsbiterr</li> <li>• [1] Counter RAM Injectdbiterr</li> <li>• [0] Counter RAM Injectsbiterr</li> </ul>
0x0C	[31:0]	RW	Read Mode Control <ul style="list-style-type: none"> <li>• [0] Disable Clear-on-Read <ul style="list-style-type: none"> <li>◦ 1: Read without clear</li> <li>◦ 0: Clear on read (default)</li> </ul> </li> </ul>
0x10	[31:0]	RW	ECC Single Bit Error Count
0x14	[31:0]	RW	ECC Double Bit Error Count
0x20	[31:0]	RW	Write Value LSB: Least Significant half of counter write value [31:0]

Register Offset	Bit Range	Access	Description
0x24	[31:0]	RW	Write Value MSB: Most Significant half of counter write value [63:32]
0x800	[31:0]	RO	Collection RAM: Counter (Start Address + 0) value[31:0]
0x804	[31:0]	RO	Collection RAM: Counter (Start Address + 0) value[63:32]
0x808	[31:0]	RO	Collection RAM: Counter (Start Address + 1) value[31:0]
0x80C	[31:0]	RO	Collection RAM: Counter (Start Address + 1) value[63:32]
...			
0xBF8	[31:0]	RO	Collection RAM: Counter (Start Address + 127) value[31:0]
0xBFC	[31:0]	RO	Collection RAM: Counter (Start Address + 127) value[63:32]

**Table: Register Extern Register Map**

Register Offset	Bit Range	Access	Description
0x00	[31:0]	RW	Table Write Index Index that shall be written to by a set operation, a SW write to this register triggers a set operation to the specified table index.
0x04	[31:0]	RW	Table Read Index Index that shall be read from by a read operation, a SW write to this register triggers a read operation to the specified table index.

Register Offset	Bit Range	Access	Description
0x100	[31:0]	RO	Table read status If read_busy='1', there is an outstanding read being executed and SRD registers are not valid.
0x108	[31:0]	RO	Version number.
0x10C	[31:0]	RO	ID of table.
0x110	[31:0]	RO	Largest table index that can be used
0x118	[31:0]	RO	Entry width (bits).
0x200	[31:0]	RO	For a table set operation, this register group defines the value that will be written to the specified index. It consists of 128 32b registers. Register[0] starts at 0x200, register[1] at 0x204 etc.
0x400	[31:0]	RO	For a table read operation, this register group holds the final read data that should only be read when read_busy = '0'. It consists of 128 32b registers. Register[0] starts at 0x400, register[1] at 0x404 etc.

The Statistics and Control register map can be found in the generated file, <inst\_name>\_defs.h. See [Generated Files](#) for the path to this generated file.

The Statistics and Control Register AXI4-Lite interface supports read and write operations for these registers. This includes Interrupt Registers in relation to ECC errors. These registers are split into “IP Components” (which are specific to the target architecture) and “P4 Elements” (which correspond to tables and counter externs in the P4 program). For each component/element, there are two register bits to distinguish between ECC Single-bit and Double-bit error functionality. The ECC Capabilities registers indicate which components/elements support ECC functionality because this is not always obvious – it can be dictated by the RAM-style setting which is sometimes automatically determined by the tool, for example, there is no ECC supported when targeting Distributed RAM.

**Table: Statistics and Control Register Map**

Register	Bit Range	Reset Value	Access	Description
Configuration Registers				
0x000	[31:0]		Read-only	VNP4 Version (SDNV) <ul style="list-style-type: none"> <li>• [31:24]: Reserved</li> <li>• [23:16]: IP Core Revision number</li> <li>• [15:8]: IP Core Minor Version number</li> <li>• [7:0]: IP Core Major Version number</li> </ul>
0x004	[31:0]		Read-only	VNP4 Instance Configuration (SDNC) <ul style="list-style-type: none"> <li>• [31:30]: Reserved</li> <li>• [29:20]: AXI Stream Clock in MHz</li> <li>• [19:10]: CAM Memory Clock in MHz</li> <li>• [9:0]: Packet Rate in Mp/s</li> </ul>
0x008	[31:0]		Read-only	ECC Capabilities Register for IP Components (IP_ECCC) Bit indexing is allocated as follows: <ul style="list-style-type: none"> <li>• Bit[0]: Packet FIFO ECC Single-bit Error Functionality</li> <li>• Bit[1]: Packet FIFO ECC Double-bit Error Functionality</li> <li>• Bit[2]: Metadata FIFO ECC Single-bit Error Functionality</li> <li>• Bit[3]: Metadata FIFO ECC Double-bit Error Functionality</li> <li>• Bits[31:4]: Reserved</li> </ul> <p>The values of each bit have the following meaning:</p>



Register	Bit Range	Reset Value	Access	Description
				<ul style="list-style-type: none"><li>• 1: ECC is supported for this element/component</li><li>• 0: ECC is not supported for this element/component</li></ul>
0x00C	[2n-1:0]		Read-only	<p>ECC Capabilities Register for P4 Elements (P4_ECCC)</p> <p>Bit indexing is allocated as follows for all tables and counter externs (up to max n=128) in the P4 program, where m is the number of tables:</p>

Register	Bit Range	Reset Value	Address	Description
				<ul style="list-style-type: none"> <li>• Bit[0]: First Table ECC Single-bit Error Functionality</li> <li>• Bit[1]: First Table ECC Double-bit Error Functionality</li> <li>• Bit[2]: Second Table ECC Single-bit Error Functionality</li> <li>• Bit[3]: Second Table ECC Double-bit Error Functionality</li> <li>• ...</li> <li>• Bit[2m-2]: m'th Table ECC Single-bit Error Functionality</li> <li>• Bit [2m-1]: m'th Table ECC Double-bit Error Functionality</li> <li>• Bit [2m]: First Counter Extern ECC Single-bit Error Functionality</li> <li>• Bit [2m+1]: First Counter Extern ECC Double-bit Error Functionality</li> <li>• ...</li> <li>• Bit [2n-2]: (n-m)'th Counter Extern ECC Single-bit Error Functionality</li> <li>• Bit [2n-1]: (n-m)'th Counter Extern ECC Double-bit Error Functionality</li> </ul> <p>The values of each bit have the following meaning:</p> <ul style="list-style-type: none"> <li>• 1: ECC is supported for this element/component</li> <li>• 0: ECC is not supported for this element/component</li> </ul>
Interrupt Registers				

Register	Bit Range	Reset Value	Access	Description
0x400 0x404	[31:0] [2n-1:0]	0x0	Read-only	<p>Interrupt Status Register (ISR)</p> <p>The bit indexing aligns with the IP_ECCC and P4_ECCC registers above.</p> <p>The values of each bit have the following meaning:</p> <ul style="list-style-type: none"> <li>• 1: Pending interrupt</li> <li>• 0: Interrupt cleared</li> </ul>
0x424 0x428	[31:0] [2n-1:0]	0x0	Read/Write	<p>Interrupt Enable Register (IER)</p> <p>The bit indexing aligns with the IP_ECCC and P4_ECCC registers above.</p> <p>The values of each bit have the following meaning:</p> <ul style="list-style-type: none"> <li>• 1: Interrupt enabled</li> <li>• 0: Interrupt disabled</li> </ul>
0x448 0x44C	[31:0] [2n-1:0]	0x0	Clear-on-Write	<p>Interrupt Clear Register (ICR)</p> <p>The bit indexing aligns with the IP_ECCC and P4_ECCC registers above.</p> <p>The values of each bit have the following meaning:</p> <ul style="list-style-type: none"> <li>• 1: Clear interrupt</li> <li>• 0: Ignored</li> </ul>
Error Registers				
0x1018	[31:0]	0x0	Clear-on-Read	Packet FIFO Single-bit ECC Error Counter (PFSE)
0x101C	[31:0]	0x0	Clear-on-	Packet FIFO Double-bit ECC Error Counter (PFDE)

Register	Bit Range	Reset Value	Access	Description
			Read	
0x1020	[31:0]	0x0	Clear-on-Read	Vector FIFO Single-bit ECC Error Counter (VFSE)
0x1024	[31:0]	0x0	Clear-on-Read	Vector FIFO Double-bit ECC Error Counter (VFDE)
Control Registers				
0x1400 [31:0] 0x1404 [2n-1:0]		0x0	Read/Write	Inject ECC Bit Error Register (IEBE)The bit indexing aligns with the IP_ECCC and P4_ECCC registers above. The values of each bit have the following meaning: <ul style="list-style-type: none"> <li>• 1: Start injecting ECC errors</li> <li>• 0: Stop injecting ECC errors</li> </ul>
0x1424	[31:0]	0xFFFF	Read/Write	Packet Rate Limiter Margin (PRLM) <ul style="list-style-type: none"> <li>• [31:16]: Reserved</li> <li>• [15:0]: Maximum number of packets per 1000 AXIS clock cycles</li> </ul>

## Runtime Drivers

### Overview

The runtime drivers provide an API that allows control plane management of both P4 and IP specific elements in an instance of the AMD Vitis™ Networking P4 IP, where P4 elements are elements described by the P4 program and IP specific elements provide additional sources of information and control for an AMD Vitis™ Networking P4 IP instance.

The runtime drivers supported P4 elements are:

- Tables from Match-Action control blocks of a P4 program

The runtime drivers supported IP specific elements are:

- Build Information Reader
- Interrupt Controller
- Control

The P4 table elements from Match-Action control blocks present in a P4 program are implemented using Look-up engines, where the Look-up engines are implemented by a variety of addressable tables (see [Match-Action Engine](#) for more information). The principle operation of the Look-up engine is where a look-up value (called a key) is provided to the Look-up engine (table), which then searches its memory for a match. The Look-up engine then generates an output (called a response), the contents of which depend on whether a match was found.

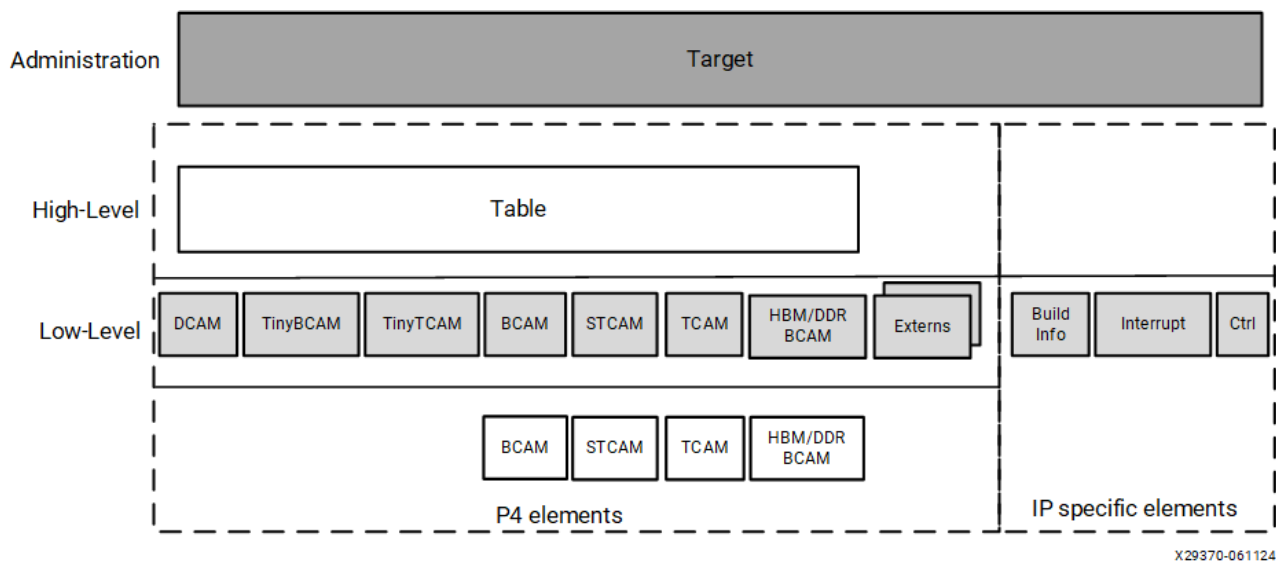
The runtime drivers provide an API that allows control plane management of tables present in an instance of the Vitis Networking P4 IP. A primary responsibility of the runtime drivers is to populate the contents of the Look-up engine with entries constructed from key and response values specified in the tables of the Match-Action control blocks of a P4 program. The key can be constructed from user defined header input, metadata input and/or internally defined match-action scalars. The response is constructed from an action identifier and action parameters. The action identifier is a binary value used to select the action named in the action list of the match-action control block and the action parameters are specified in the associated action declaration.

The build information reader element allows the control plane to read settings used when configuring the instance of the Vitis Networking P4 IP plus the version of the IP instance.

The interrupt element provides control of triggering inputs to detect ECC related errors for P4 table elements plus internal FIFOs.

The control element allows setting of the packet rate limit margin, control of ECC error injection plus soft reset of Vitis Networking P4 engines within the FPGA.

The runtime drivers are organized in a layered architecture to provide different levels of access for applications. The layers are depicted in the following figure:

**Figure: Runtime Driver Layer**

Detailed information on the driver's public interfaces can be found in the Runtime Software Driver Documentation located at <{XILINX\_VIVADO}/doc/vitis\_net\_p4/drivers/doc. The functionality offered by the high level API is summarized as follows:

1. Initialization/configuration
2. Management of table contents
  - a. Table entry inserts
  - b. Updates
  - c. Deletes
3. Monitoring of table health (such as ECC error information)
4. Interrupt control for ECC error information
5. Build Information Reader to allow reading
  - a. IP Version
  - b. Settings used to configure the Vitis Networking P4 instance
6. Control of
  - a. Setting the packet rate limit margin
  - b. ECC error injection
  - c. Soft reset of Vitis Networking P4 engines
7. Utility functions for ease of use (such as fetch a handle to a given table instance by name)

The administration layer comprises the target driver, the purpose of which is to simplify the initialization and management of all control plane accessible elements in the Vitis Networking P4 design. It achieves this by

consuming a configuration data structure that is automatically generated when Vitis Networking P4 is run. The data structure describes the configuration of all control plane elements present in the Vitis Networking P4 design (such as tables). Using this information, the target driver is able to initialize a driver context structure instance for every control plane accessible element in the design. Furthermore, it provides an API that allows these structures to be retrieved by specifying the name of the corresponding control plane element. After one of these context structures has been retrieved (for example, a table), the functions offered by its associated driver can be used.

Use of these APIs is illustrated in [Runtime Drivers](#). The following are key points for the operation of the target driver:

- When Vitis Networking P4 is run in AMD Vivado™ Design Suite, a configuration data structure is generated that describes all tables present in the design, this structure is an input to the target driver.
- During initialization, the target driver consumes the generated configuration structure and uses it to initialize all control plane accessible elements present (both P4 and IP specific elements).
- During initialization the table driver internally creates an instance of the appropriate low-level CAM driver based on the provided configuration data structure.
- The set of operations supported for a given table varies depending on the underlying CAM IP with which it was implemented.
- The table driver functions internally concatenate the action parameters and the action identifier provided by the caller to produce a response. It is this response value that is ultimately passed down into the lower layers.

The low-level CAM drivers provide the same style of interface as the table driver, but with a lower level of abstraction. That is, these drivers require that the action parameters and the action identifier have already been concatenated into a response parameter by the caller. Also, for the Direct (DCAM), TinyBCAM and TinyTCAM tables, APIs have been included as a debug feature to handle storing last key and response values.

## Location

The runtime software driver is located within a Vivado Design Suite

project in the directory `<project_name>/<project_name>.gen/sources_1/ip/<inst_name>/src/sw/drivers`, and is composed of the following:

**common**

Definitions shared between multiple drivers.

**cam\_top**

Low-level drivers.

**table**

High-level table drivers.

**target**

Top-level utility driver for managing both P4 and IP specific elements present. Includes generated Vitis Networking P4 configuration data.

**target\_mgmt**

Low-level drivers to manage a target (Vitis Networking P4 instance) via IP specific elements.

**cam\_obf**

Obfuscated CAM drivers.

**Makefile**

Top-level Makefile.

**xilinx\_vitisnetp4\_drivers.mak**

Makefile snippet that configures the top-level Makefile with the source code directories. Includes path dependencies.

**x86.mak**

Makefile snippet that configures the top-level Makefile with details of the platform, for example, indication of which tool chain to use.

## Building

The build system provided with the drivers is configured to compile the code into both static and shared libraries for a x86 Linux host using GCC. The build system is configured to use that platform by means of the `x86.mak` file, which consists of a set of variables that specify details such as the toolchain to use, compilation flags, and the installation directory. The design of the build system is such that it can easily be targeted to other platforms by defining a custom Makefile snippet with the details for



the platform in question. The top-level Makefile can be directed to use a different platform when invoking the make command:

```
make PLATFORM=<customer_platform_file>.mak
```

This causes the x86.mak file to be ignored and the specified file to be used instead. Additionally, the top-level Makefile can also pick up the PLATFORM if it is set as an environment variable. This provides a convenient alternative to specifying it at the command line. If PLATFORM is defined both as an environment variable and as a parameter to make, the latter takes precedence. Several of the variables defined in the Makefile snippet can be overridden with environment variables or at the command line, if so desired.

## Runtime Driver Examples

The following example programs demonstrate valid initialization and use of the runtime drivers. These are provided for reference purposes only, the build system provided with the drivers does not compile this program into an executable. The programs have no interface to an FPGA and instead use a set of stub functions for register access. To run this program, see [Building](#) for details on compiling the drivers for their platform and [Porting to Platform](#) for details on how to connect the drivers to their platform's register access functions.

### Table Example

This section summarizes the Target API functions to control a table P4 element. This example C program is based on the FiveTuple example design and is provided at the following location:

```
<project_name>/<project_name>.gen/sources_1/ip/<inst_name>/src/sw/  
drivers/target/examples/five_tuple_example.c
```

This program demonstrates target driver function calls to achieve the following functionality:

- Initialize the driver using `XilVitisNetP4TargetInit()`
- Obtain a handle for table using `XilVitisNetP4TargetGetTableByName()`
- Obtain an action identifier using `XilVitisNetP4TableGetActionId()`
- Perform the following table entry operations:
  - insert using `XilVitisNetP4TableInsert()`
  - update using `XilVitisNetP4TableUpdate()`
  - delete using `XilVitisNetP4TableDelete()`
- Exit driver using `XilVitisNetP4TargetExit()`

The example application uses the packed byte arrays “FiveTupleKeyArray” and “FiveTupleActionParamsArray”. These arrays use a big-endian format and are packed in the order that they are declared in the P4 file. In the FiveTuple P4 file the table is defined as follows:

```
table FiveTuple {
    key
        = { hdr.ipv4.src      : exact;
            hdr.ipv4.dst      : exact;
                hdr.ipv4.protocol : exact;
                table_key_sport  : exact;
                table_key_dport  : exact; }

    actions
        = { InsertVLAN;
            NoAction; }

    size
        = 8192;

    default_action = NoAction;
}
```

with the field sizes defined as follows:

- `hdr.ipv4.src`: 32-bit field
- `hdr.ipv4.dst`: 32-bit field
- `hdr.ipv4.protocol`: 8-bit field
- `table_key_sport`: 16-bit field
- `table_dport`: 16-bit field

The fields for each key in the “FiveTupleKeyArray” are ordered as shown in the following figure:

**Figure: FiveTuple Key Packed Byte Array**

0	31 32	63 64	71 72	87 88	103
hdr.ipv4.src	hdr.ipv4.dst	hdr.ipv4.protocol	table_key_sport	table_key_dport	

In the example, all the keys result in a "InsertVLAN" action which is defined as follows:

```
action InsertVLAN(bit<3> pcp, bit<1> cfi, bit<12> vid)
```

resulting in a packed action parameter byte array ordered as shown in the following figure:

**Figure: InsertVLAN Action Packed Byte Array**

0	2	3	4	15
pcp	cfi	vid		

## Control Example

The example summarizes Target and Control API functions to set the packet rate limit margin, and injecting ECC errors. This example C program is provided at the following location: `<project_name>/<project_name>.gen/sources_1/ip/<inst_name>/src/sw/drivers/target/example/ctrl_example.c`

This program demonstrates target driver function calls to achieve the following functionality:

- Initialize the driver using `XilVitisNetP4TargetInit()`
- Obtain a context for the control driver using `XilVitisNetP4TargetGetCtrlDrv()`
- Reset all of the Vitis Networking P4 engines using `XilVitisNetP4TargetCtrlSoftResetEngine()`
- Set the packet rate limit margin using `XilVitisNetP4TargetCtrlSetPacketRateLimitMargin()`
- Enable and Disable ECC error injection for a single Vitis Networking P4 Component using `XilVitisNetP4TargetCtrlIpComponentEnableInjectEccError()` and `XilVitisNetP4TargetCtrlIpComponentDisableInjectEccError()`
- Enable and Disable ECC error injection for a single P4 Element using `XilVitisNetP4TargetCtrlP4ElementEnableInjectEccError()` and `XilVitisNetP4TargetCtrlP4ElementDisableInjectEccError()`

## Interrupt Example

This example summarizes Target and Interrupt API functions to control and monitor the ECC interrupts for the table present in the FiveTuple example. This example C program is provided at the following location:  
<project\_name>/<project\_name>.gen/sources\_1/ip/<inst\_name>/src/sw/drivers/target/example/interrupt\_example.c

This program demonstrates target and interrupt driver function calls to achieve the following functionality:

- Initialize the driver using `XilVitisNetP4TargetInit()`
- Obtain the context for the interrupt driver using `XilVitisNetP4TargetGetInterruptDrv()`
- Obtain the P4 element ID for the “FiveTuple” table using `XilVitisNetP4TargetGetTableElementIdByName()`
- Enable both single and double bit ECC error interrupts using `XilVitisNetP4TargetInterruptEnableElementEccErrorById()`
- Read interrupt status to determine if either a single or double bit ECC error interrupt has occurred using `XilVitisNetP4TargetInterruptGetElementEccErrorStatusById()`
- Clear the single or double bit ECC error interrupts if present using `XilVitisNetP4TargetInterruptClearElementEccErrorStatusById()`
- Disable the single bit ECC error interrupt using `XilVitisNetP4TargetInterruptDisableElementEccErrorById()`
- Exit driver using `XilVitisNetP4TargetExit()`

## Driver Memory Usage Estimate

The stack size is dependent on the type of CAM selected to implement a table (see [Compiler Table Selection](#) for more information). The stack size required per CAM is shown on the following table:

**Table: Stack Size Requirements**

CAM Type	Stack Size
DCAM	4 KB
BCAM	128 KB
STCAM	128 KB
TCAM	1 MB
Tiny BCAM	4 KB
Tiny TCAM	4 KB
HBM/DDR BCAM (with HW Update enabled)	4 KB

CAM Type	Stack Size
HBM/DDR BCAM (with HW Update disabled) (default)	128 KB

The stack size used in an Vitis Networking P4 IP instance should be greater than or equal to the largest stack size of the CAMs present in that instance. For example:

- If a Vitis Networking P4 IP instance contains a DCAM, stack size is equal to or greater than 4 KB.
- If a Vitis Networking P4 IP instance contains several DCAMs and a BCAM, stack size is equal to or greater than 128 KB.
- If a Vitis Networking P4 IP instance contains a BCAM and a TCAM, stack size is equal to or greater than 1 MB.

## Porting to Platform

The drivers provided by Vitis Networking P4 are deliberately decoupled from platform-specific functionality such as FPGA register access. This is done to maximize portability, but requires that customer applications using the drivers provide this functionality during initialization.

To use the drivers on a user platform an environment interface structure must be provided for the specific platform. The environment interface structure provides the driver with the platform specific implementations for the following:

- Read and write functions for 32 bit registers (mandatory)
- Logging functions for information and error logging (mandatory)
- A context data structure supporting the implementation of the above functions (optional)

The environment interface structure `XilVitisNetP4EnvIf` is defined as follows in the `vitisnetp4_common.h` file:

```
struct XilVitisNetP4EnvIf
{
    XilVitisNetP4UserCtxType    UserCtx;    /**< Pointer to
application-specific                                context needed by
```

```

callbacks (set to
NULL if not
needed)*/
XilVitisNetP4WordWrite32Fp WordWrite32; /**< Callback for
function to perform
a 32-bit write to
a hardware
register*/
XilVitisNetP4WordRead32Fp WordRead32; /**< Callback for
function to perform
a 32-bit read from
a hardware
register*/
XilVitisNetP4LogFp LogError; /**< Callback for
function to emit
error messages*/
XilVitisNetP4LogFp LogInfo; /**< Callback for
function to emit
informational
messages*/
};

```

A reference implementation is provided by function `XilVitisNetP4StubEnvIf()` implemented in file `vitisnetp4_common.c`. This reference is a stubbed implementation of the environment structure interface which does not perform any register reads or writes.

## Driver Requirements

When porting the runtime drivers, be aware of the following characteristics:

- It is written to comply with a C99 compiler
- It requires a math library
- It performs floating point operations
- It is designed to run in userspace context and not a kernel context such as Linux kernel context

## Example

The following example provides an implementation of an environment interface structure suitable for standalone (bare metal) software

applications on AMD processors. The implementation reuses the stubbed implementation for logging functions and replaces the register read and write functions with versions suitable for this platform. Also demonstrated is the use of the user context initialization data to store the base address of the Vitis Networking P4 IP. The example has the following steps:

1. Define the user context.
  2. Implement the register read and write functions.
  3. Implement functions to create and destroy the example environment interface structure.
- Use of the example environment interface structure:
    - The following provides a structure that can contain the address of the Vitis Networking P4 IP core:

```
typedef struct ExampleUserContext
{
    XilVitisNetP4AddressType VitisNetP4IpAddress;
} ExampleUserContext;
```



- Implement the register read and write functions:

- The functions use the `Xil_Out32` and the `Xil_In32` functions from the AMD hardware abstraction layer of the standalone library, see *BSP and Libraries Document Collection* ([UG643](#)) for more information.

```
XilVitisNetP4ReturnType
ExampleVitisNetP4WordWrite32Baremetal(XilVitisNetP4Env
vIf *EnvIfPtr, XilVitisNetP4AddressType Address,
uint32_t WriteValue)
{
    ExampleUserContext *UserCtxPtr;
    if (EnvIfPtr == NULL)
    {
        return XIL_VITISNETP4_GENERAL_ERR_NULL_PARAM;
    }
    if (EnvIfPtr->UserCtx == NULL)
    {
        return
XIL_VITISNETP4_GENERAL_ERR_INTERNAL_ASSERTION;
    }
    UserCtxPtr = (ExampleUserContext *)EnvIfPtr-
>UserCtx;
    /* Currently have base address fixed, move to
UserCtx */
    Xil_Out32(UserCtxPtr->VitisNetP4IpAddress +
Address, WriteValue);
    return XIL_VITISNETP4_SUCCESS;
}
XilVitisNetP4ReturnType
ExampleVitisNetP4WordRead32Baremetal(XilVitisNetP4Env
If *EnvIfPtr, XilVitisNetP4AddressType Address,
uint32_t *ReadValuePtr)
{
    ExampleUserContext *UserCtxPtr;
    if (EnvIfPtr == NULL)
    {
        return XIL_VITISNETP4_GENERAL_ERR_NULL_PARAM;
    }
    if (ReadValuePtr == NULL)
    {
        return XIL_VITISNETP4_GENERAL_ERR_NULL_PARAM;
```

```

    }
    if (EnvIfPtr->UserCtx == NULL)
    {
        return
XIL_VITISNETP4_GENERAL_ERR_INTERNAL_ASSERTION;
    }
    UserCtxPtr = (ExampleUserContext *)EnvIfPtr-
>UserCtx;
    *ReadValuePtr = Xil_In32(UserCtxPtr-
>VitisNetP4IpAddress + Address);
    return XIL_VITISNETP4_SUCCESS;
}

```

- Functions to create and destroy the environment interface structure:

```

int ExampleCreateBaremetalEnvIf(XilVitisNetP4EnvIf
*EnvIfPtr, XilVitisNetP4AddressType VitisNetP4IpAddress)
{
    XilVitisNetP4ReturnType Result;
    ExampleUserContext *UserCtxPtr;
    UserCtxPtr = calloc(1, sizeof(ExampleUserContext));
    if (UserCtxPtr == NULL)
    {
        printf("ERROR: Failed to allocate memory\n\r");
        return -1;
    }
    UserCtxPtr->VitisNetP4IpAddress = VitisNetP4IpAddress;
    /*
    * Reusing the Stub and updating the Read and Write
function
    */
    Result = XilVitisNetP4StubEnvIf(EnvIfPtr);
    if (Result == XIL_VITISNETP4_SUCCESS)
    {
        /* Install stubs for each platform-specific function
    */
        (*EnvIfPtr)->WordWrite32 =
ExampleVitisNetP4WordWrite32Baremetal;
        (*EnvIfPtr)->WordRead32 =
ExampleVitisNetP4WordRead32Baremetal;
        (*EnvIfPtr)->UserCtx =

```

```

(XilVitisNetP4UserCtxType)UserCtxPtr;
    }
    return 0;
}
void ExampleDestoryBaremetalEnvIf(XilVitisNetP4EnvIf
*EnvIfPtr)
{
    free(EnvIfPtr->UserCtx);
}

```

- Use of the example environment interface structure:

```

XilVitisNetP4EnvIf EnvIf;
int Result;
Result = ExampleCreateBaremetalEnvIf(&EnvIf,
XPAR_VITISNETP4_0_BASEADDR);
if (Result != XIL_VITISNETP4_SUCCESS)
{
    return -1;
}
ExampleDestoryBaremetalEnvIf(&EnvIf);

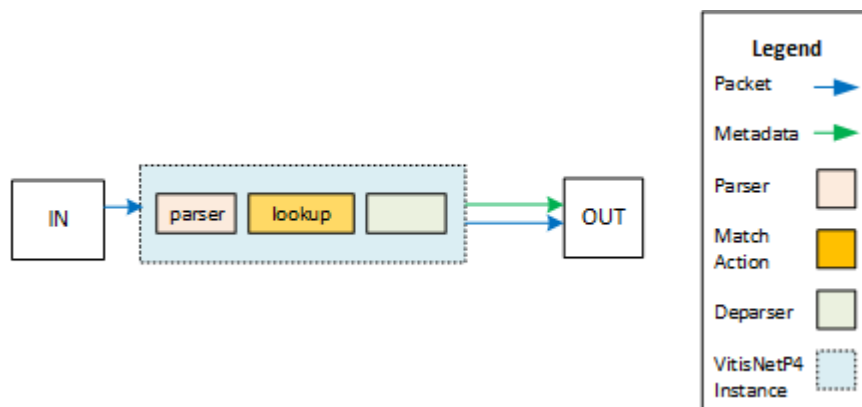
```

## Examples of Basic Architectures

This section provides examples to illustrate how AMD Vitis™ Networking P4 can be used. Note that an empty box in the diagrams represents a "null" engine to illustrate that a particular engine is not required.

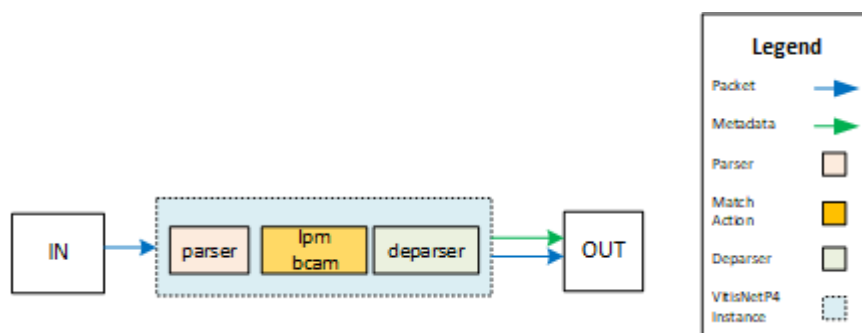
The following figure illustrates a trivial topology of an OpenFlow classifier consisting of a Parser and a Look-up engine. The Look-up engine is of type TCAM in this example.

### Figure: Trivial Topology of an Openflow Classifier



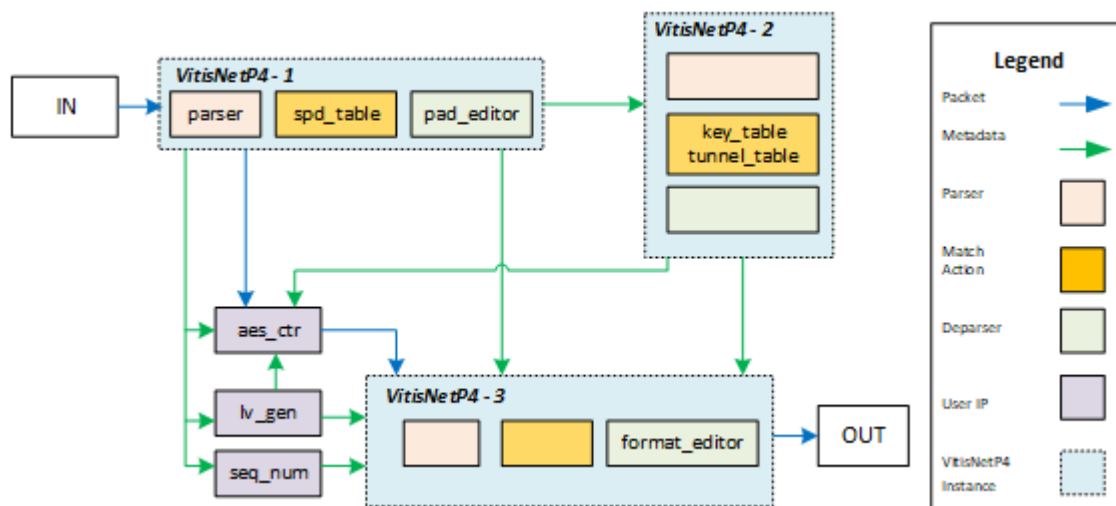
The following figure illustrates a router example that utilizes two different types of Look-up engines: exact match (BCAM) and longest prefix match (LPM).

**Figure: Router Example**



The following figure illustrates an example of performing the encryption path of an IPSec implementation. The system consists of three Vitis Networking P4 instances and user IP. The user IP in this example implements other functions such as counters and also the encryption function.

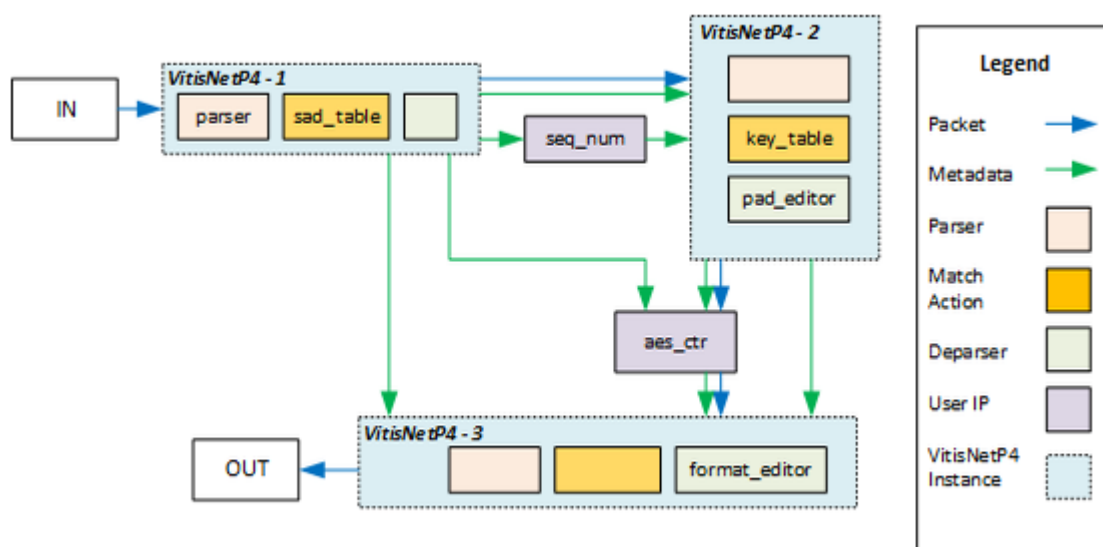
**Figure: User IP Performing Encryption Path of an IPSec Implementation**



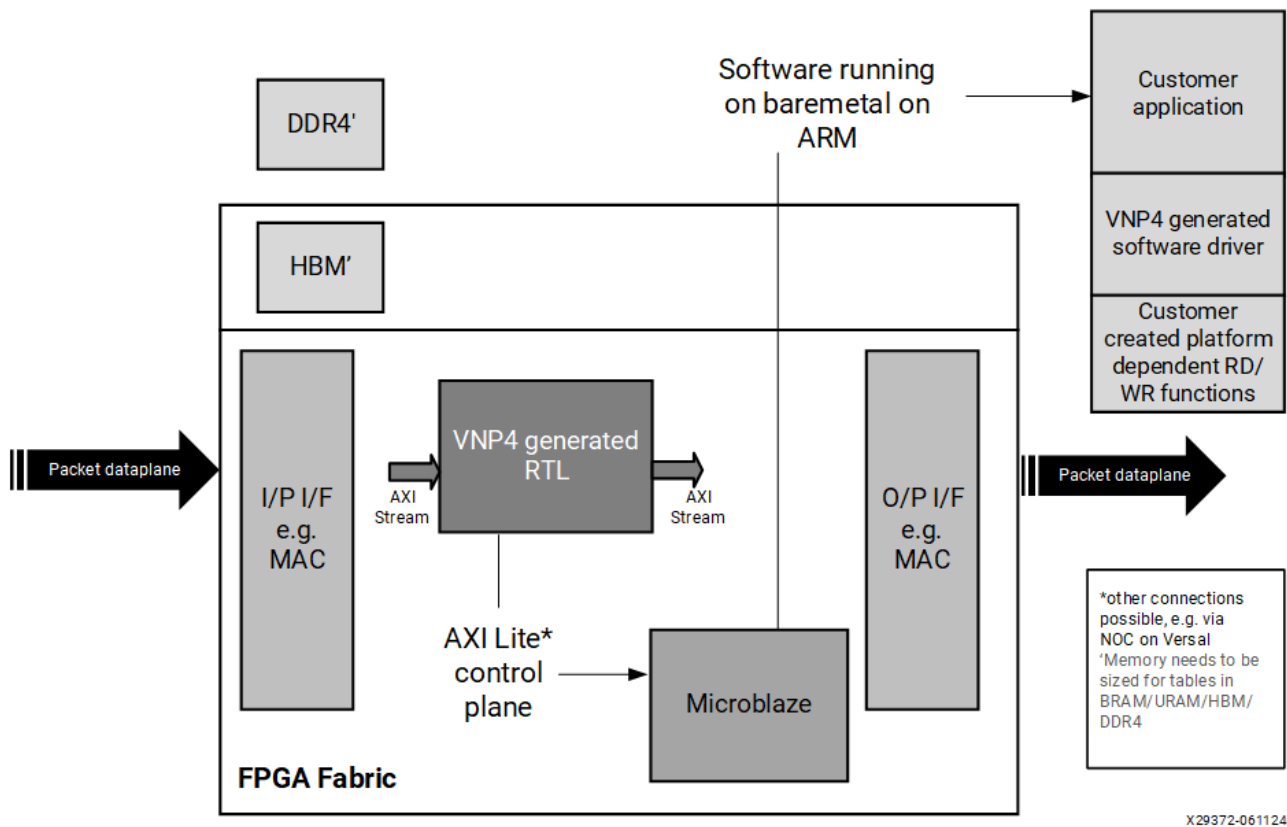
The following figure illustrates another example showing the decryption path of an IPsec implementation.

The following figure illustrates a P4 system with an embedded AMD MicroBlaze™ processor.

**Figure: Decryption Path of an IPsec Implementation**

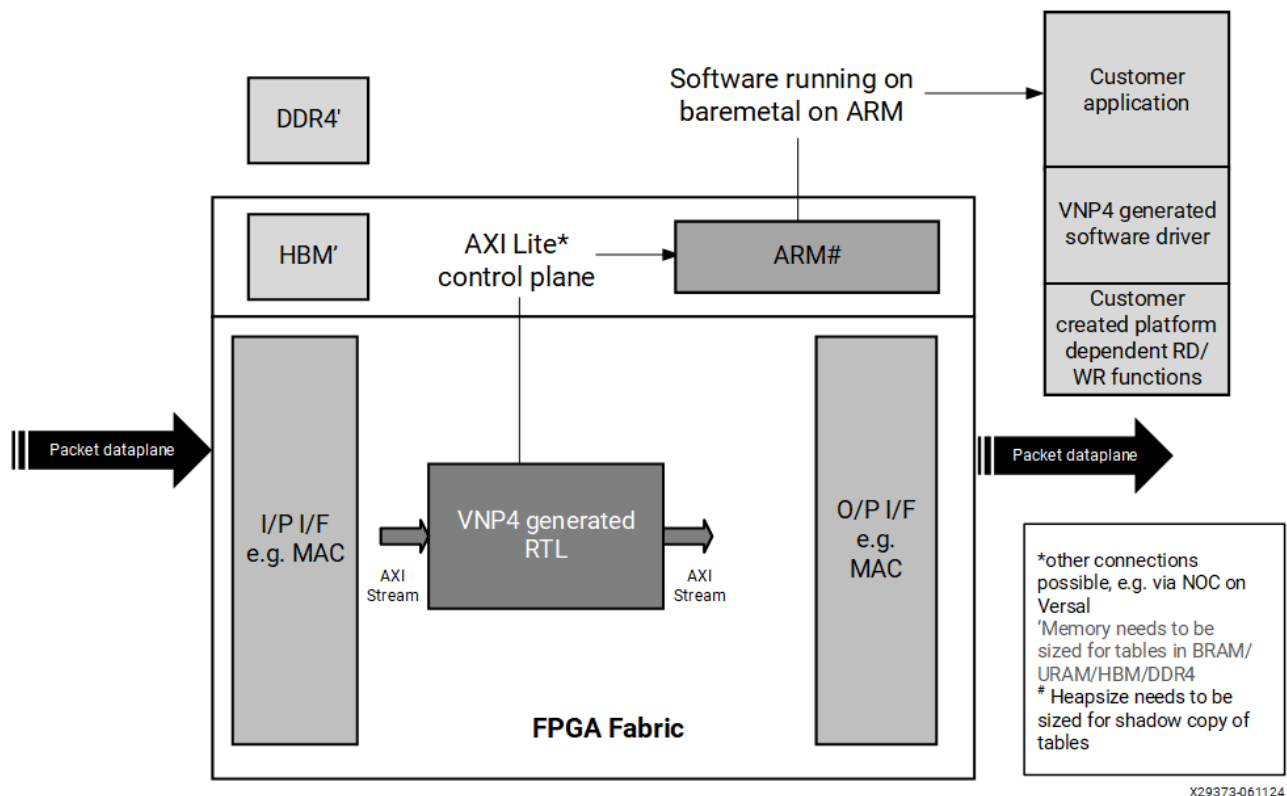


**Figure: P4 System with Embedded MicroBlaze Processor**

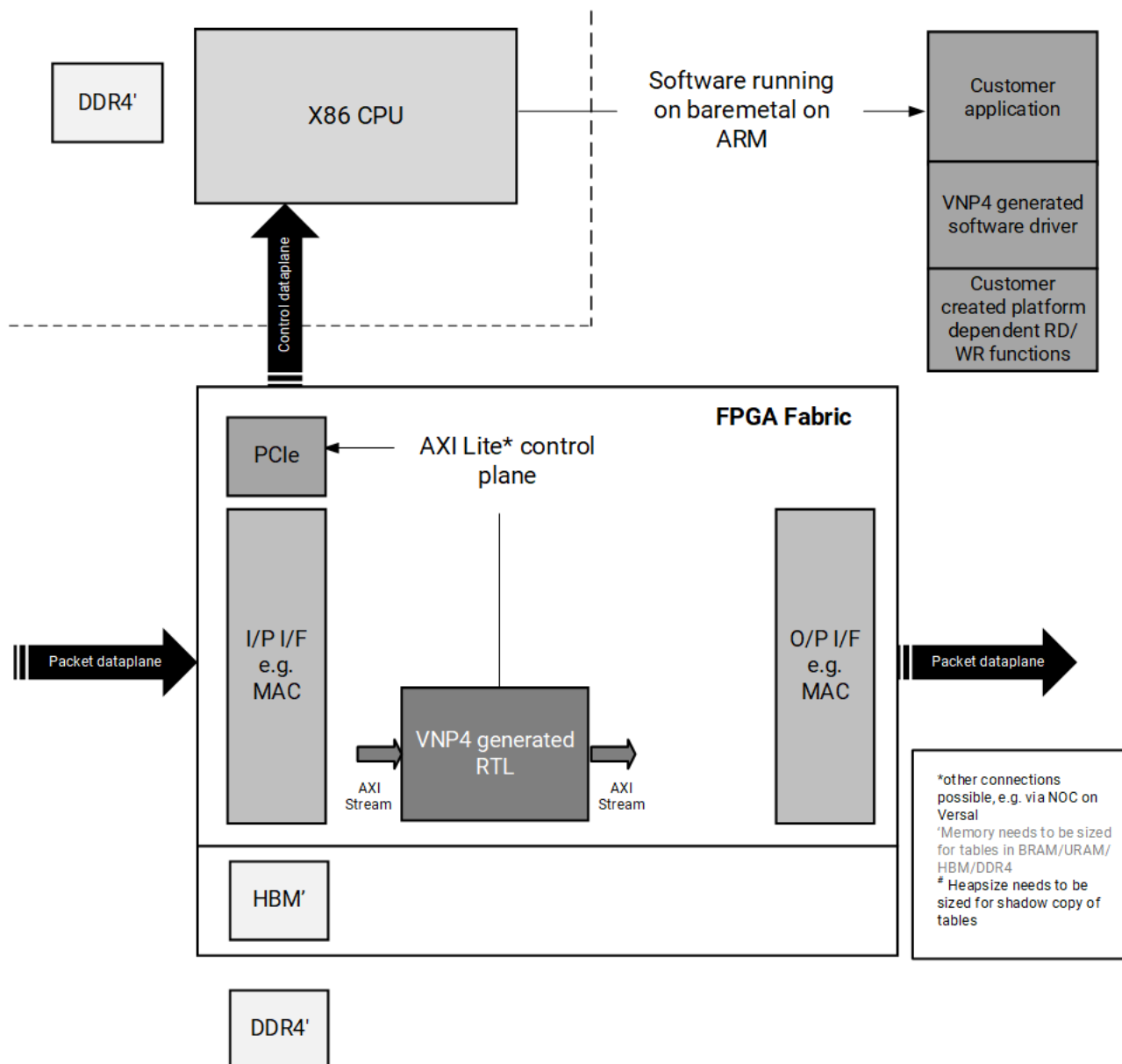


The following figure illustrates a P4 system with an embedded Arm® processor.

**Figure: P4 System with Embedded Arm Processor**



The following figure illustrates a P4 system with external CPU.

**Figure: P4 System with External CPU**

X29374-061124

## Sample P416 Program

The following is an example of a P4<sub>16</sub> program (FiveTuple).

**Note:** Metadata structure names and metadata field names must not use any SystemVerilog keywords, otherwise errors occur in synthesis, for example, `meta.output.eth_dmac` and `meta.input.skip_bytes` are not allowed because `input` and `output` are keywords.

```
typedef bit<48>  MacAddr;
typedef bit<32>  IPv4Addr;
```



```

const bit<16> QINQ_TYPE = 0x88A8;
const bit<16> VLAN_TYPE = 0x8100;
const bit<16> IPV4_TYPE = 0x0800;

const bit<8> TCP_PROT = 0x06;
const bit<8> UDP_PROT = 0x11;

//
*****
***** //
// ***** H E A D E R S
***** //
//
*****
***** //

header eth_mac_t {
    MacAddr dmac; // Destination MAC address
    MacAddr smac; // Source MAC address
    bit<16> type; // Tag Protocol Identifier
}

header vlan_t {
    bit<3> pcp; // Priority code point
    bit<1> cfi; // Drop eligible indicator
    bit<12> vid; // VLAN identifier
    bit<16> tpid; // Tag protocol identifier
}

header ipv4_t {
    bit<4> version; // Version (4 for IPv4)
    bit<4> hdr_len; // Header length in 32b words
    bit<8> tos; // Type of Service
    bit<16> length; // Packet length in 32b words
    bit<16> id; // Identification
    bit<3> flags; // Flags
    bit<13> offset; // Fragment offset
    bit<8> ttl; // Time to live
    bit<8> protocol; // Next protocol
    bit<16> hdr_chk; // Header checksum
    IPv4Addr src; // Source address

```

```

        IPv4Addr dst;          // Destination address
    }

    header ipv4_opt_t {
        varbit<320> options; // IPv4 options - length =
        (ipv4.hdr_len - 5) * 32
    }

    header tcp_t {
        bit<16> src_port;    // Source port
        bit<16> dst_port;    // Destination port
        bit<32> seqNum;      // Sequence number
        bit<32> ackNum;      // Acknowledgment number
        bit<4>  dataOffset;  // Data offset
        bit<6>  resv;        // Offset
        bit<6>  flags;       // Flags
        bit<16> window;      // Window
        bit<16> checksum;    // TCP checksum
        bit<16> urgPtr;      // Urgent pointer
    }

    header tcp_opt_t {
        varbit<320> options; // TCP options - length =
        (tcp.dataOffset - 5) * 32
    }

    header udp_t {
        bit<16> src_port;    // Source port
        bit<16> dst_port;    // Destination port
        bit<16> length;      // UDP length
        bit<16> checksum;    // UDP checksum
    }

    //
    *****
    ***** //
    // ***** S T R U C T U R E S
    ***** //
    //
    *****
    ***** //

```

```

// header structure
struct headers {
    eth_mac_t    eth;
    vlan_t       new_vlan;
    vlan_t       vlan;
    ipv4_t       ipv4;
    ipv4_opt_t   ipv4opt;
    tcp_t        tcp;
    tcp_opt_t    tcptopt;
    udp_t        udp;
}

// User metadata structure
struct metadata {
    // empty
}

// User-defined errors
error {
    InvalidIPpacket,
    InvalidTCPpacket
}

//
*****
***** //
// ***** P A R S E R
***** //
//
*****
***** //

parser MyParser(packet_in packet,
                 out headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t smeta) {

    state start {
        transition parse_eth;
    }

    state parse_eth {

```

```
        packet.extract(hdr.eth);
        transition select(hdr.eth.type) {
            VLAN_TYPE : parse_vlan;
            IPV4_TYPE : parse_ipv4;
            default    : accept;
        }
    }

    state parse_vlan {
        packet.extract(hdr.vlan);
        transition select(hdr.vlan.tpid) {
            IPV4_TYPE : parse_ipv4;
            default    : accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        verify(hdr.ipv4.version == 4 && hdr.ipv4.hdr_len >=
5,
            error.InvalidIPpacket);
        packet.extract(hdr.ipv4opt,
(((bit<32>)hdr.ipv4.hdr_len - 5) * 32));
        transition select(hdr.ipv4.protocol) {
            TCP_PROT   : parse_tcp;
            UDP_PROT   : parse_udp;
            default     : accept;
        }
    }

    state parse_tcp {
        packet.extract(hdr.tcp);
        verify(hdr.tcp.dataoffset >= 5,
error.InvalidTCPpacket);
        packet.extract(hdr.tcptopt,
(((bit<32>)hdr.tcp.dataoffset - 5) * 32));
        transition accept;
    }

    state parse_udp {
        packet.extract(hdr.udp);
        transition accept;
```

```

    }
}

//
*****
***** //
// ***** P R O C E S S I N G
***** //
//
*****
***** //

control MyProcessing(inout headers hdr,
                    inout metadata meta,
                    inout standard_metadata_t smeta) {

    bit<16> table_key_sport;
    bit<16> table_key_dport;
    bool hit = false;

    action InsertVLAN(bit<3> pcp, bit<1> cfi, bit<12> vid) {
        hdr.new_vlan.setValid();
        hdr.new_vlan.pcp = pcp;
        hdr.new_vlan.cfi = cfi;
        hdr.new_vlan.vid = vid;
        hdr.new_vlan.tpid = hdr.eth.type;
    }

    table FiveTuple {
        key          = { hdr.ipv4.src      : exact;
                        hdr.ipv4.dst      : exact;
                        hdr.ipv4.protocol : exact;
                        table_key_sport   : exact;
                        table_key_dport   : exact; }
        actions      = { InsertVLAN;
                        NoAction; }
        size          = 8192;
        default_action = NoAction;
    }

    apply {

```

```
        if (hdr.udp.isValid()) {
            table_key_sport = hdr.udp.src_port;
            table_key_dport = hdr.udp.dst_port;
            hit = FiveTuple.apply().hit;
        } else if (hdr.tcp.isValid()) {
            table_key_sport = hdr.tcp.src_port;
            table_key_dport = hdr.tcp.dst_port;
            hit = FiveTuple.apply().hit;
        }

        if (hit) {
            if (hdr.vlan.isValid())
                hdr.eth.type = QINQ_TYPE;
            else
                hdr.eth.type = VLAN_TYPE;
        }
    }
}

//
*****
***** //
// ***** D E P A R S E R
***** //
//
*****
***** //

control MyDeparser(packet_out packet,
                    in headers hdr,
                    inout metadata meta,
                    inout standard_metadata_t smeta) {
    apply {
        packet.emit(hdr.eth);
        packet.emit(hdr.new_vlan);
        packet.emit(hdr.vlan);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.ipv4opt);
        packet.emit(hdr.tcp);
        packet.emit(hdr.tcptopt);
        packet.emit(hdr.udp);
    }
}
```

```

}

//
*****
***** //
// ***** M A I N
***** //
//
*****
***** //


XilinxPipeline(
    MyParser(),
    MyProcessing(),
    MyDeparser()
) main;

```

## Supported P416 Language Features

The following table identifies supported and not supported P4<sub>16</sub> language features.

---

 **Note:** Conditional statements are not supported inside actions.

---

**Table: P4<sub>16</sub> Language Features**

Context	Supported	Not Supported
Data Types	<b><i>void</i></b> (extern functions only) <b><i>match_kind</i></b> (architecture defined) <b><i>error</i></b> (architecture and user defined) Integers (unsigned: <b><i>bit</i></b> and <b><i>varbit</i></b> ) <b><i>bool</i></b>	<b><i>void</i></b> (others) <b><i>match_kind</i></b> (user defined) <b><i>ParserTimeout error</i></b> String ( <b><i>@name "..."</i></b> ) Integers (others)
Derived Types	<b><i>enum</i></b> (architecture specific) <b><i>header</i></b> (and header	<b><i>enum</i></b> (user defined) <b><i>header</i></b> (push and pop methods)

Context	Supported	Not Supported
	stack) <b>struct</b> <b>package</b> (architecture specific) <b>typedef</b> <b>const</b>	<b>header_union</b> <b>tuple</b> <b>package</b> (user defined)
Expressions	Cast Variables Operators (!, ~, -, +, *, >>, <<, &, ^,  )	Operators (/, %, ?)
Statements	<b>assign</b> <b>if, else</b> <b>block</b>	<b>exit</b> <b>return</b> <b>switch</b>
Parsing	<b>state</b> <b>extract</b> (fixed width extraction) <b>extract</b> (variable width extraction) <b>transition</b> (user defined) <b>transition</b> (accept and reject) <b>select</b> (no overlap with mask/range) <b>verify</b>	<b>lookahead</b> <sup>1</sup> (extract to underscore, or advance) <b>length</b> <b>select with</b> <b>overlapsskipping bits</b> (mask/range) <b>sub-parsers</b>
Control	<b>action</b> <b>table</b> (key, actions, default_action, entries, size and implementation statement) <b>emit</b> <b>apply</b> <b>extern</b> (user) <b>Externs</b> (Built-in)	<b>Other Externs</b>



Context	Supported	Not Supported
	Counter, Checksum, InternetChecksum, Register)	
<p>1. A potential workaround for the “lookahead” functionality can be achieved by using the “extract” method. This involves re-defining headers, to split them into a shorter “generic” header which can be extracted first, before deciding what the next transition select should be and the remainder of the header parsing.</p>		

## Nulling Blocks

The following code shows examples of how to "null" each engine.

```
// TRANSLATED TO P4 BY XILINX
#include <core.p4>
#include <xsa.p4>

//
*****
***** //
// ***** S T R U C T U R E S
***** //
//
*****
***** //

// header structure
struct headers {
}

// User metadata structure
struct metadata {

}
```

```
//
*****

***** //
// ***** P A R S E R
***** //

//
*****

***** //

parser NullParser(packet_in packet,
                    out headers hdr,
                    inout metadata meta,
                    inout standard_metadata_t smeta) {

    state start {
        transition accept;
    }
}

//
*****

***** //
// ***** M A T C H   A C T I O N   E N G I N E
***** //

//
*****

***** //

control NullMAPipe(inout headers hdr,
                   inout metadata meta,
                   inout standard_metadata_t smeta) {

    apply {
    }
}

//
*****

***** //
// ***** D E P A R S E R
***** //
```

```
//
*****

***** //

control NullDeparser(packet_out packet,
                      in headers hdr,
                      inout metadata meta,
                      inout standard_metadata_t smeta) {
    apply {
    }
}

//
*****

***** //
// ***** M A I N
***** //

//
*****

***** //

XilinxPipeline(
    NullParser(),
    NullMAPipe(),
    NullDeparser()
) main;
```

If no packet editing is required, it is recommended to emit all extracted headers again in the original order that they were extracted in the Parser, even in cases where the output packet bus is not used. This results in optimal resource utilization.

If the results of the Parser are not used in any practical way, there is potential for the whole design to be optimized away.

## P4 References and Guidelines

### P4 Cheat Sheet

### Data Types

```
// Basic data Types:
bool flag; // two values: true & false
int sigData; // signed integer
bit bitData; // unsigned int (bit-string)
varbit varData; // variable-size bit-str.

// typedef: introduces alternate type name
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

// const: constant value definition
const bit<16> IPV4_TYPE = 0x0800;
const bit<8> UDP_PROT = 0x11;

// headers: ordered collection of members
// operations test and set validity bits:
// isValid(), setValid(), setInvalid()
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> type;
}

// struct: unordered collection of members
// use to define header structures
struct headers_t {
    ethernet_t ethernet;
    vlan_t[2] vlan; // stack
}

// structs can also be used to define
// metadata structures
struct meta_out_t {
    macAddr_t dstAddr;
    bit<16> offset;
}

// error: contains opaque values that
// can be used to signal errors
error {
    InvalidEthernetType,
```

```
IPv4VersionNotSupported,  
}
```

## Compiler Directives

```
// define macros (without arguments)  
#define NULL 0  
  
// undefine existing macros  
#undef NULL  
  
// conditional directives  
#if #else #endif #ifdef #ifndef #elif  
  
// include contents from other files  
#include <file.p4> // can use quotes ""
```

## Statements and Expressions

```
// local metadata declaration  
bit<16> tmp1, tmp2, tmp3, tmp4;  
bit<16> tmp5 = 0; // default value  
  
// variable assignment and member access  
tmp1 = hdr.ethernet.type;  
  
// bit slicing, concatenation (++)  
tmp2 = tmp1[7:0] ++ tmp1[15:8];  
  
// data type casting  
tmp3 = (bit<16>)tmp2;  
  
// arithmetic operators:  
// +, -, *, <<, >>  
tmp4 = 2*tmp1 + tmp3 - (bit<16>)tmp1[7:0];  
  
// bitwise operators:  
// and, or, not, &, |, ^, ~  
tmp5 = (~tmp1 & tmp2) | (tmp3 ^ tmp4);
```

```
// conditional statement
// expression output requires a bool
// logical operators: ==, !=, >, >=, <, <=
if (next_hop == 0) {
    metadata.next_hop = hdr.ipv4.dst;
} else {
    metadata.next_hop = next_hop;
}
```

## Tables

```
table ipv4_lpm {
    // key match kinds: exact, ternary,
    // lpm, range, field_mask & unused
    key = {
        hdr.ipv4.dstAddr : lpm;
        hdr.ipv4.srcAddr : exact;
    }
    // actions that can be invoked
    actions = {
        ipv4_forward;
        drop_packet;
    }
    // table properties:
    // maximum number of entries
    size = 1024;
    // enable direct match
    // (for exact keys only)
    direct_match = true;
    // number of supported unique masks:
    // ternary and lpm only
    num_masks = 256;
    // action invoked when table fails
    // to find a match for the key used
    // default_action = drop_packet();
}
```

## Actions

```
// action declaration:
action ipv4_forward(bit<9> port) {
    // local variable and assignment
    bit<48> tmpAddr = hdr.ipv4.srcAddr;

    // in/out values from/to data plane
    hdr.ipv4.srcAddr = hdr.ipv4.dstAddr;
    hdr.ipv4.dstAddr = tmpAddr;

    // inputs provided by control plane
    meta.port = port; // stored in table
}

// explicit action invocation:
ipv4_forward(0x123); // constant arg. value
```

## Parsing

```
// parser: must always begin with the
// special state "start"
state start {
    transition parse_ethernet;
}

// User-defined parser state
state parse_ethernet {
    // fixed length extraction
    packet.extract(hdr.ethernet);
    // assignment to metadata
    meta.eth_dmac = hdr.ethernet.dstAddr;
    // select transition to next state
    transition select(hdr.ethernet.type) {
        0x0800 : parse_ipv4;
        0x8100 : parse_vlan;
        default : accept;
    }
}

// header stack extraction
state parse_vlan {
```

```
    packet.extract(hdr.vlan.next);
    transition select(hdr.vlan.last.tpid) {
        VLAN_TYPE : parse_vlan; // loop
        IPV4_TYPE : parse_ipv4;
        default   : accept;
    }
}

// advanced parsing
state parse_ipv4 {
    // fixed length extraction
    packet.extract(hdr.ipv4);
    // custom form of error handling
    // false causes transition to reject
    verify(hdr.ipv4.version == 4,
           error.IpVersionNotSupported);
    // variable length extraction
    packet.extract(hdr.ipv4opt,
                  ((bit<32>)hdr.ipv4.hdr_len - 5) * 32);
    // constant transition (end of packet)
    transition accept; // fixed transition
}
```

## Deparsing

```
// the inverse of parsing is deparsing,
// or packet reconstruction
apply {
    // insert headers into packet(if valid)
    packet.emit(hdr.ethernet);
    packet.emit(hdr.vlan); // stack
    packet.emit(hdr.ipv4);
    packet.emit(hdr.ipv4opt); //varbit
}
```

## Processing

```
// all variables, tables and actions
// definitions outside apply methodapply {
```



```
// unconditional table search
ipv6_lpm.appply();

// branch on header validity
// conditional table search
if (hdr.ipv4.isValid()) {
    ipv4_lpm.apply();
}

// branch on table hit result
if (local_ip_table.apply().hit) {
    send_to_cpu();
} else {
    drop_packet();
}

// branch on error
if (smeta.parser_error ==
    error.InvalidEthernetType)
    drop_packet();

// header manipulation
hdr.new_vlan = hdr_vlan;
hdr.vlan.setInvalid();
if (hdr.ipv4.isValid())
    hdr.new_vlan.tpid = 0x0800
}
```

## Vitis Networking P4 Architecture

```
// standard metadata format
struct standard_metadata_t {
    bit<1>  drop;
    bit<64> ingress_timestamp;
    bit<16> parsed_bytes;
    error   parser_error;
}

// Pipeline elements
parser Parser<H, M>(
```

```
        packet_in b,  
        out H hdr,  
        inout M meta,  
        inout standard_metadata_t smeta  
    );  
  
    control MatchAction<H, M>(  
        inout H hdr,  
        inout M meta,  
        inout standard_metadata_t smeta  
    );  
  
    control Deparser<H, M>(  
        packet_out b,  
        in H hdr,  
        inout M meta,  
        inout standard_metadata_t smeta  
    );  
  
    // architecture's main package  
    package XilinxPipeline<H, M>(  
        Parser<H, M> p,  
        MatchAction<H, M> ma,  
        Deparser<H, M> dep  
    );
```

## Counter Extern

```
const bit<32> NUM_COUNTERS = 8192; // up to 65536 supported  
typedef bit<13> CounterIndex_t; // CounterIndex type should  
correspond with the number of counters  
  
// Instantiation of Counter Externs  
// 3 modes supported, as shown here  
Counter<bit<64>, CounterIndex_t>(NUM_COUNTERS,  
CounterType_t.PACKETS) PacketCounter;  
Counter<bit<64>, CounterIndex_t>(NUM_COUNTERS,  
CounterType_t.BYTES) ByteCounter;  
Counter<bit<64>, CounterIndex_t>(NUM_COUNTERS,  
CounterType_t.PACKETS_AND_BYTES) ComboCounter;
```

```
// The "count" method can only be called once per counter
instance per packet
PacketCounter.count(counter_index);
ByteCounter.count(counter_index);
ComboCounter.count(counter_index);
```

## Checksum Extern

```
Checksum<bit<16>>(HashAlgorithm_t.ONES_COMPLEMENT16) cksum;
cksum.apply(
{
  hdr.ipv4.version,
  hdr.ipv4.hdr_len,
  hdr.ipv4.tos,
  hdr.ipv4.length,
  hdr.ipv4.id,
  hdr.ipv4.flags,
  hdr.ipv4.offset,
  hdr.ipv4.ttl,
  hdr.ipv4.protocol,
  hdr.ipv4.src,
  hdr.ipv4.dst
},
  hdr.ipv4.hdr_chk
);
```

## InternetChecksum Extern

```
InternetChecksum() cksum;
cksum.clear();
cksum.subtract({hdr.ipv4.ttl, hdr.ipv4.protocol});
hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
cksum.add({hdr.ipv4.ttl, hdr.ipv4.protocol});
cksum.get(hdr.ipv4.hdr_chk);
```

## User Extern

```
// Structure Definitions
struct divider_input {
    bit<32> divisor;
    bit<32> dividend;
}

struct divider_output {
    bit<32> remainder;
    bit<32> quotient;
}

divider_input div_in;
divider_output div_out;

// Instantiation of UserExtern Object
UserExtern<divider_input, divider_output>(34) calc_divide;

// Interface to User Extern
calc_divide.apply(div_in, div_out);
```

## Register Extern

```
const bit <32> L2_NUM_REGS = 512; // up to 65536 supported
const bit <32> L2_NUM_FLOWS = 9; // reg index width (2**9 =
512)
typedef bit<L2_NUM_FLOWS> l2_num_flows_t;
typedef bit<32> seqNo_t;

// Instantiate Register extern
Register<seqNo_t, l2_num_flows_t>(L2_NUM_REGS) seqNo_reg;

// Register index and value variables
seqNo_t init_seqNo;
l2_num_flows_t flowID = hdr.tcp.dst_port[L2_NUM_FLOWS-1:0];

// The register extern methods can only be called once per
instance per packet
seqNo_reg.read(flowID, init_seqNo); // READ
if (hdr.tcp.flags[SYN_POS:SYN_POS] == 1)
{
```

```
    init_seqNo = hdr.tcp.seqNo;           // MODIFY
}
seqNo_reg.write(flowID, init_seqNo);     // WRITE
```

## Guidelines for Porting P4 Code to the AMD Architecture

There are some architecture-specific changes required when porting P4 code from other targets. Here is an example to show the changes required when porting P4 code from the V1Switch architecture:

- Replace the included architecture file:
  - `#include <v1model.p4>` should be replaced with `#include <xsa.p4>`
- The Vitis Networking P4 Architecture has only three components: *Parser*, *Match-Action*, and *Deparser*. A P4 program needs to be modified to fit into this architecture. For example, moving from V1Model to XSA:
  - V1Switch should be renamed to XilinxPipeline.
  - VerifyChecksum and ComputeChecksum must be removed as these controls are not supported in XSA.
  - Ingress and Egress pipelines must be combined into one Match-Action pipeline.
- Deparser control has metadata and `standard_metadata_t` inouts, similar to the Parser and Match-Action pipelines. These must be added to the Deparser control declaration (even if they are not used in the P4 program).
- The standard metadata structure is different and only has a few fields. Any use of other standard metadata fields from the V1Model must be removed, or replaced with user metadata.
- Supported externs are different in XSA compared to the V1Model. Check the latest XSA to see which externs are supported and the syntax involved.

## P4 References

- P4 Standards Organization
  - <https://p4.org>
- Specifications
  - [P4<sub>16</sub>](#)
  - [Portable Switch Architecture](#)
- P4 Language
  - [Language tutorial](#)
- Tutorial Examples
  - [basic](#)
  - [basic\\_tunnel](#)
  - [load\\_balance](#)
  - [MRI](#)
  - [P4runtime](#)
  - [source\\_routing](#)

## DPI Simulation

This section describes how and why DPI is used within the test bench provided with the example designs. DPI is used to enable the combined testing of the C code of the software drivers with the SystemVerilog RTL of the hardware design. DPI is described here in brief as an aid, it is recommended that you familiarize yourself with Section 35 of IEEE 1800-2017 to gain a more complete understanding of its functionality.

## Important Files

There are three files from the example design test bench that are relevant to DPI:

- Precompiled DPI library binary
- VitisNetP4 instance (config dependent) SystemVerilog package file
- VitisNetP4 DPI (non-config dependent) SystemVerilog package file
- Test bench Control block

When the example design is generated, the precompiled DPI library can be found in the following location:

`/<proj_location>/vitis_net_p4_0_ex/imports/vitis_net_p4_drv_dpi.so`

This precompiled shared object (.so) file contains:

- The control plane drivers for tables and externs, for example, `XilVitisNetP4BcamInsert()`.
- A set of utility functions specific to supporting the use of the control plane drivers inside a SystemVerilog simulation. The names of all such functions start with "XilVitisNetP4Dpi", for example, `XilVitisNetP4DpiCreateEnv()`.

The following SystemVerilog package files are also generated and can be found at the following locations:

`/<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/sources_1/ip/vitis_net_p4_0/src/hw/simulation/vitis_net_p4_dpi_pkg.sv`

`/<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex.gen/sources_1/ip/vitis_net_p4_0/src/verilog/vitis_net_p4_0_pkg.sv`

The first package contains a declaration of all the DPI constants, parameters, structs, and functions provided by the library, as well as all of the imports for the functions found in the DPI library binary. The second package has the specific configurations from the P4 program, which varies depending on the p4 file and configuration used. These package files should be referenced to understand which functions are present in the DPI library and what parameters they accept.

The control block of the test bench is where the DPI library is used in practice, and it is found at the following location:

`/<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex/imports/example_control.sv`

The control block is a useful reference for users who wish to develop their own test bench, as it illustrates in practice how to use the functions provided by the DPI library, which is described in further detail in the next sections.

## Interaction Between SystemVerilog and C

The main interaction between the SystemVerilog and C code can be

summarized as follows:

- When the SystemVerilog test bench wants to modify a table's state (for example, insert an entry), it calls the associated control plane driver function from the DPI library.
  - SystemVerilog code is now causing C code to execute
- As the control plane driver function executes, it performs a series of register reads and writes by triggering associated tasks in the SystemVerilog test bench.
  - C code is now causing SystemVerilog code to execute

To understand how this handshaking is achieved, it is important to first understand some of the features of DPI which influence how it is used in the example designs, and also to have a basic understanding of how the control plane drivers function.

## AXI Interfacing Tasks

For the DPI library to be able to trigger the execution of a SystemVerilog task, the interface to the task must be known to the DPI library. In practice, this means that it must be hard-coded in both the SystemVerilog test bench and the DPI library.

For the example designs, two tasks called `axi_lite_wr` and `axi_lite_rd` (located in `example_control.sv`) are defined and can be called from the DPI library. These tasks contain the logic needed to issue writes and reads to the AMD Vitis™ Networking P4 IP's AXI interface. The `axi_lite_wr` task takes two integer inputs (address and data), the `axi_lite_rd` task takes one integer input (address) and one integer output (data). You are encouraged to develop your own custom test bench. The DPI library provided with Vitis Networking P4 can be used in any such test bench, provided that the test bench defines these two tasks as described in this section.

## Hierarchical Path to AXI Interface

For the DPI library to be able to trigger the `axi_lite_wr` and `axi_lite_rd` tasks, the DPI library must be provided with the location of those tasks in the hierarchy of the simulation. This is accomplished by calling a utility function named `XilVitisNetP4DpiCreateEnv()`. This



function accepts a string parameter that specifies the hierarchical path to the module that implements these tasks. In the case of the example designs, this path is specified in:

```
/<proj_location>/vitis_net_p4_0_ex/vitis_net_p4_0_ex/imports/  
example_control.sv  
as example_top.example_control
```

## Preparing for Use of Control Plane Drivers

`XilVitisNetP4DpiCreateEnv()` produces a return value, the data type of which is *chandle*. This data type is used by SystemVerilog to hold pointers to memory in the DPI library. In the case of the example designs, the memory being pointed to is a data structure needed for successful initialization of the control plane drivers by the test bench. The test bench must call the `XilVitisNetP4DpiCreateEnv()` function once for each VNP4 instance, and store the return values in variables.

## Simulating with Multiple VNP4 Instances

The example design can be extended to support multiple VNP4 IP instances. There are a few steps involved:

1. Update `example_top.sv` to set the `NUM_M_AXI` parameter to the number of VNP4 IP instances. Then instantiate all those instances and connect the data-plane signals as appropriate in `example_dut_wrapper.sv`.
2. Update `example_control.sv`:
  - a. Add ``include` at the top for the package files of any other VNP4 IP instances.
  - b. Add further calls to the `run_cli_commands` task for each VNP4 IP instance (following the commented out instructions).
  - c. Remove `run_traffic` command from all but last `cli_commands` file, so that traffic is only run after all table entries have been setup.
3. Update to `run-p4bm-vitisnet.tcl` script (or equivalent script) for behavioral modeling of multiple P4 instances with potentially other custom modules in between.

The `example_control` module uses the same `axi_lite_wr` and `axi_lite_rd` tasks for the AXI-Lite signaling sequence, but it multiplexes those signals to the corresponding VNP4 IP instance according to the call to the `run_cli_commands` task.

Alternative setups are also possible. For example, the multiplexing could be replaced with the addition of different offset addresses for the different VNP4 IP instances, in cases where a single AXI-Lite I/F exists on the DUT and the address decoding to the different VNP4 IP instances happens within the DUT.

See [Daisy-Chain Example Design](#) for more information.

## Additional Utilities

A new set of helper functions and tasks have been added to `vitis_net_p4_dpi_pkg.sv` to ease the table and register DPI calls from a SystemVerilog test bench. For example:

- `XilVitisNetP4DPIinit()`
- `XilVitisNetP4DPIexit()`
- `XilVitisNetP4DPItableAdd()`
- ...

These functions/tasks were part of `vitis_net_0_pkg.sv` on previous releases (named differently). These are optional wrapper functions/tasks that can be used instead of the previously mentioned DPI control plane functions, if so desired. Note that these new helper functions/tasks require the use of the new `XilVitisNetP4DPIHandle` struct to hold various pointers and other memory and driver related information.

In addition to the control plane drivers, the DPI binary library provides the following optional utility functions:

- `XilVitisNetP4DpiByteArrayCreate()`
- `XilVitisNetP4DpiStringToByteArray()`
- ...

These functions help the test bench consume key-response entries from a text file and convert them into byte arrays before passing them to a control plane driver function such as `XilVitisNetP4BcamInsert()`. It is possible to implement this code solely in SystemVerilog, if so desired.

Any function described above with a name ending in `*Create()` has a counterpart that ends with `*Destroy()`. The `*Create()` functions allocate memory inside the DPI library and the `*Destroy()` counterparts release the memory. It is good practice to call the `*Destroy()` functions after the memory that has been allocated is no longer required, such as towards the end of the simulation's execution.

## Custom Control Plane Driver Usage

For users who wish to write their own custom control plan application/test bench, precompiled DPI binary control plane functions can be called manually for every sub-module found in the p4 design (tables, counters, externs etc). Note that for every `XilVitisNetP4*Init()` called, a `XilVitisNetP4*Exit()` needs to be called before exiting the simulation and the names of these functions do not contain the text 'DPI'. These functions are imported directly from their C version via import "DPI-C" (located in `vitis_net_p4_dpi_pkg.sv`).

## Resource Optimizations

### CAMS

If a higher-rate CAM Memory clock signal can be provided at multiples of the required packet rate, that will allow for TDM within the CAM IP and potentially reduce logic and RAM resources.

Reducing the required Packet Rate parameter can also help to reduce the logic and RAM resources of the CAMs.

Other approaches to reduce the resource utilisation of CAM tables include:

- Reducing the size (number of entries supported)
- Reducing the `num_masks` parameter value (where applicable)

### P4 Coding Styles

In general, it is recommended to avoid heavily nested "if" conditionals and long sequences of "if/else if/else if/..." conditionals. These can

increase the design latency and buffering/pipelining resources.

## Parser

For Variable-length extracts, it is strongly recommended to make the granularity as coarse as possible to minimize utilization. For example, many header formats are in multiples of 16, 32 or 64 bits. The variable length extract should be coded explicitly with this multiplier in place. This is particularly important when the variable length information is coming from input metadata. e.g.

```
packet.extract(hdr.ipv4opt, (((bit<32>)hdr.ipv4.hdr_len - 5)
* 32));
```

## Deparser

If there is no packet editing required, it is recommended to emit all extracted headers again in the original order that they were extracted in the Parser, even in cases where the output packet bus will not be used. This will result in optimal resource utilization.

## User Metadata

By default, all the fields of the User Metadata structure are provided as input and output ports of the VNP4 IP. However in some cases not all of these fields are needed for input/output.

- Fields may be used for input-only and are ignored at the output.
- Fields may be output-only, where the input is ignored and the signal value is always assigned within the P4 program.
- Fields may be internal-only, e.g. for signalling between the parser and the match-action.

Where the input functionality is not required for certain fields, it is recommended to assign those fields to a default value early in the P4 program (e.g. in the start state of the parser, or at the beginning of the match-action pipeline). This allows the tool to optimize away the unused pipelining from the input port.

Similarly where output functionality is not required for certain fields, it is recommended to assign those fields to a default value at the end of the match-action pipeline. This allows the tool to optimize away the unused pipelining to the output port.

The User Metadata structure is very flexible and can be extended to a large number of fields which can help during the debug stages of a design to provide extra visibility. It should be noted though that very wide metadata structures can lead to sub-optimal resource utilization due to the amount of pipelining/buffering involved. Once the design is functioning as expected, it is recommended to reduce the size of the User Metadata structure as much as possible to optimise the resources.

## Header Insertion

Where a new header is to be inserted using the `setValid()` method, it is generally more optimal to include a check for the previous header being valid (assuming that the insertion should only happen under this condition). A simple example of this check is highlighted below for inserting a new VLAN header following an ethernet header.

```
if (new_vlan_insert == 1 && hdr.eth.isValid())  
{  hdr.new_vlan.setValid(); }
```

This can help the tool narrow down the possible locations in the packet where the header insertion can happen, optimizing the logic resources.

# Debugging

## Debug Tools

There are many tools available to address design issues. It is important to know which tools are useful for debugging various situations.

### Vivado Design Suite Debug Feature

The AMD Vivado™ Design Suite debug feature inserts logic analyzer and virtual I/O cores directly into your design. The debug feature also allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature in the Vivado IDE is used for logic debugging and validation of a design running in AMD devices.

The Vivado logic analyzer is used to interact with the logic debug LogiCORE IP cores, including:

- ILA 2.0 (and later versions)
- VIO 2.0 (and later versions)

See the *Vivado Design Suite User Guide: Programming and Debugging* ([UG908](#)).

## Simulation Debug

For VNP4, there are CAM debug flags which relate to hardware and some which relate to software. For a description of the hardware debug flags, see [CAM HW Debug](#). For a full list of both hardware and software CAM debug flags, see the *Simulation Debug* section of the respective CAM product guide.

When using STCAM, TCAM or BCAM it is recommended to enable debug by calling `XilVitisNetP4CamSetDebugFlags` prior to calling the driver initialization functions listed in [DPI Simulation](#). Note that this is not supported for TinyBCAM, TinyTCAM or Direct Tables.

### Example of CAM Configuration Debug

```
debug_flag = CAM_DEBUG_ARGS | CAM_DEBUG_CONFIG |  
CAM_DEBUG_CONFIG_ARGS  
debug_flags = 0x2042
```

Before a CAM Configuration API call, set the debug flags in the VNP4 drivers:

```
void XilVitisNetP4CamSetDebugFlags(uint32_t flags)
```

or via the CAM driver directly:

```
void cam_arg_set_debug_flags(cam_arg_t *cam_arg, uint32_t  
debug_flags);
```

Ensure that you set the `debug_flags` back to zero when finished debugging the configuration.

## Table Programming

There are a few common pitfalls related to table programming:

- It is important to always re-compile the software drivers following any changes to the P4 program. This includes using the re-generated \*\_defs.c/\*\_defs.h files.
- It is important to always re-compile the software drivers following an upgrade to a different version of VNP4.
- The 32-bit register read/write functions must be defined for the target design setup (see [Porting to Platform](#) for details). This should include any base addressing to the beginning of the VNP4 instance.
- The AXI-Lite address width is dependent on the number of control-plane elements in the P4 design (e.g. the number of tables). If the P4 program is changed after the AXI-Lite interface connections have been made in the hardware design, then it is possible that a wider AXI-Lite address width and address space is required (e.g. if new tables were added to the P4 program, or if the statistics registers are enabled). In some cases the Address Editor or AXI Interconnect needs to be updated to account for an increase in address space, or else some of the tables may not be accessible for programming.

## User Metadata

The flexibility of the User Metadata structure can be helpful for debugging a VNP4 design. It is possible to effectively “probe” different parts of the P4 program by assigning variables of interest to new fields of the user\_metadata structure at various stages of the P4 program, so that they become visible at the output ports of the VNP4 IP. Those field ports can then be viewed in an RTL simulation waveform or they can be captured by an ILA if running in hardware. These user\_metadata fields can later be removed after debug is complete for more efficient utilization.

# Additional Resources and Legal Notices

## Finding Additional Documentation

## Technical Information Portal


The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to <https://docs.amd.com>.

## Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select Help > Documentation and Tutorials.
- On Windows, click the Start button and select Xilinx Design Tools > DocNav.
- At the Linux command prompt, enter docnav.

---

 **Note:** For more information on DocNav, refer to the *Documentation Navigator User Guide* ([UG968](#)).

---

## Design Hubs

AMD 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 DocNav, click the Design Hubs View tab.
- Go to the [Design Hubs](#) web page.

## Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Support](#).

## References

These documents provide supplemental material useful with this guide:



1. *P4 Language Consortium website* (<http://p4.org>)
2. *P4<sub>16</sub> Language Specification v1.0.0* (<http://p4.org/specs>)
3. *P4<sub>16</sub> Portable Switch Architecture (PSA) v1.0* (<http://p4.org/specs>)
4. *Binary CAM Search LogiCORE IP Product Guide* (PG317)
5. *Ternary CAM Search LogiCORE IP Product Guide* (PG318)
6. *Semi-Ternary CAM Search LogiCORE IP Product Guide* (PG319)
7. *Cached DRAM Binary CAM LogiCORE IP Product Guide* (PG427)
8. *WireShark website* (<http://wireshark.org>)
9. *Vitis Networking P4 Installation Guide and Release Notes* (UG1307)
10. *Vitis Networking P4 Getting Started Guide* (UG1373)
11. *BSP and Libraries Document Collection* (UG643)
12. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
13. *Vivado Design Suite User Guide: Synthesis* (UG901)
14. *Vivado Design Suite User Guide: Implementation* (UG904)
15. *Low-Level Driver Software Documentation: {XILINX\_VIVADO}/doc/vitis\_net\_p4/drivers/doc*
16. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
17. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
18. *RFC 9293 Transmission Control Protocol*: (<http://datatracker.ietf.org/doc/html/rfc9293>)

## Revision History

The following table shows the revision history for this document.

Section	Revision Summary
11/13/2024 Version 2024.2	
General updates	Updated content for Vivado Design Suite 2024.2.
07/09/2024 Version 2024.1	
General updates	Updated content for Vivado Design Suite 2024.1.
10/18/2023 Version 2023.2	

Section	Revision Summary
General updates	Updated content for Vivado Design Suite 2023.2.
05/06/2023 Version 2023.1	
General updates	Updated content for Vivado Design Suite 2023.1.
12/08/2022 Version 2022.2	
General updates	Updated content for Vivado Design Suite 2022.2.
08/24/2022 Version 2022.1	
Initial release	N/A

## Please Read: Important Legal Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes. THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR

PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

## 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

© Copyright 2018-2024 Advanced Micro Devices, Inc. AMD, the AMD Arrow logo, UltraScale, UltraScale+, Versal, Vitis, Vivado, and combinations thereof are trademarks of Advanced Micro Devices, Inc. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the US and/or elsewhere. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.