Barefoot Tofino Compiler User Guide

This document explains the pragmas, primitives, and hash functions supported by the Barefoot Networks P4 Compiler for Tofino.

Document Revision History

Revision	Date	Description
10k-UG5-008	02/01/2019	Add new primitives execute_meter_with_or,execute_meter_from_hash_with_or,invalidate_digest,invalidate_clone. Add new pragmas pa_overlay_stage_separation,force_immediate,max_actions,egress_pkt_length_stage, field_list_field_slice,hash_algorithm,meter_profile,use_container_valid Updated pragmas meter_sweep_interval range value, phase0 default action Corrected typo in polynomial Hash Algorithms table.
10k-UG5-007EA	05/10/2018	Corrected `clone_egress_pkt_to_egress` description to state the clone is made of the packet as it *leaves* the egress pipeline, not as it enters.
10k-UG5-006EA	05/10/2018	Add documentation for pragmas including `pa_alias`, `phase0`, `force_match_dependency`, `stateful_field_slice`, and `optional_field`.
10k-UG5-005EA	10/12/2017	SDE 6.0.0 Release. Add new primitives, count_from_hash and execute_meter_from_hash. Add section, “Limitations on the Use of Primitives in Default Actions.” Clarify PHV container filling.
10k-UG5-004EA	07/27/2017	SDE 5.0.0 Release. Add `symmetric` pragma. Clarify `ways` pragma and `pack` pragma. Note limitation on constant argument to `subtract()`.
10k-UG5-003EA	07/25/2017	Early access release

Contents

1

Introduction: Using the Barefoot Networks P4 Compiler

In this document, we provide instructions for running P4 compiler for Tofino (the **`p4c-tofino`** compiler) and using the compiler pragmas, primitives, and hash functions supported in the compiler. The **`p4c-tofino`** compiler allows you to build P4 programs for the Tofino ASIC device and for the Tofino software model. The p4c-tofino compiler transforms a P4 program into a sequence of programming instructions that configure the Tofino forwarding pipeline to execute the packet processing algorithm you've specified. In typical usage, you will invoke the p4c-tofino compiler by running the P4-Build script, which is part of the Barefoot Networks Capilano SDE.

The information described in this document applies only to programs built for the Tofino ASIC. See the "Compatibility" section, below, for details on building P4 programs for other targets.

1.1 Getting the Compiler

The p4c-tofino compiler is packaged in the Barefoot Networks Capilano SDE, available from the Barefoot Networks support portal. For SDE installation instructions, see the support article, Building SDE in two easy steps. Once the SDE is installed, you can find the p4c-tofino compiler in `$SDE/install/bin`.

1.2 Compiler Output

Compiler output is placed in a file hierarchy rooted at the directory pointed to by the `-o` command line switch. By default, the output directory is the P4 program's name followed by `.tofino`.

The structure of this output directory is as follows:

- `api`: Auto-generated C code for interacting with Tofino at runtime (libpd)
- `cfg`: Device configuration, in JSON format
- `context`: Symbol tables and various address/symbol mappings, in JSON format
  - `mau.context.json`: Mappings for Match+Action tables and other pipeline objects
  - `parser.context.json`: Mappings for the parser
  - `deparser.context.json`: Mappings for the deparser
  - `phv.context.json`: Mappings between P4 fields and PHV addresses. Optionally, parser timing settings.
- `graphs`: Graph structures for the compiler-view of the program. This directory is created only when --create-graphs option is used.
  - `egress_graph.png`: The egress table control flow graph.
  - `egress_parser.png`: The egress parse graph.
  - `ingress_graph.png`: The ingress table control flow graph.
  - `ingress_parser.png`: The ingress parse graph.
- `logs`: Various plaintext logs generated during the compilation process
  - `asm.log`: Tofino assembly generation log
  - `mau.characterize.log`: Detailed Match+Action table packing information, such as efficiency
  - `mau.config.log`: Detailed information about static compile time device configuration.
  - `mau.gateway.log`: Detailed information about gateway table placement.
  - `mau.gw.log`: Detailed information about gateway encoding.
  - `mau.log`: Main Match+Action logging file detailing how resources are allocated.
  - `mau.resources.log`: Summary of all resources used in each Match+Action stage and per table.
  - `mau.sram.log`: Detailed information about SRAM allocation.
  - `mau.tcam.log`: Detailed information about TCAM allocation.
  - `mau.tp.log`: Detailed information about table placement in the Match+Action pipeline.
  - `pa.log`: Detailed PHV allocation log

        - `pa.results.log`: Detailed results about final PHV allocation.
        - `parde.config.log`: Final parser/deparser configuration information
        - `parde.error.log`: Parser/deparser errors
        - `parde.log`: Detailed parser/deparser processing log
    - `visualization`: Visualizations of the compiled program, in various non-plaintext formats
        - `mau.html`: Per-stage Match+Action resource usage
        - `parser.*.html`: Ingress/egress parser state machine configuration
        - `deparser.html`: Deparser configuration
        - `phv_allocation.html`: Per Match+Action stage PHV occupancy and usage
        - `table_placement.html`: High-level per-stage Match+Action resource usage
    - `out.tfa`: Device configuration, in Tofino assembly format

Depending on which command line flags are used (for example, the verbosity and libpd generation flags), some of these elements may not be present in the output directory.

1.3 Compiler Command Line Flags

Use the --help flag to print a description of the command line options:

```
root@localhost:~/doc/bf-sde-4.1.0.11/install/bin# ./p4c-tofino --help
usage: p4c-tofino [-h] [--create-graphs] [--disable-bridged-metadata]
                  [--disable-ha-allocation] [--dump_preprocessed]
                  [--extra-phv] [--extra-tphv] [--gen_asm]
                  [--no-dead-code-elimination] [--no-phase0]
                  [--number-mau-stages NUMBER_MAU_STAGES] [--output OUTPUT]
                  [--p4-14-spec] [--p4-name P4_NAME] [--p4-prefix P4_PREFIX]
                  [--parse-state-merge {True,False}] [--parser-timing-reports]
                  [--placement-order
{ingress_only,egress_only,ingress_before_egress,egress_before_ingress,ingress_and_egress}]
                  [--target {Tofino}] [--verbose {0,1,2}] [--version]
                  source
```

The Compiler Command Line Flags are listed below.

1.3.1 positional arguments:

`source`: A source file to include in the P4 program.

1.3.2 optional arguments:

`-h` or `--help`: show this help message and exit
`--create-graphs`: Creates parse and table flow graphs.
`--disable-bridged-metadata`: Turn off automatically generated bridged metadata.
`--disable-ha-allocation`: Removes resource allocation constraints associated with HA (high availability) mode.
`--dump_preprocessed` or `-E`: Dump pre-processed P4 code in `.i` file(s) and stop.
`--extra-phv`: Add 32 additional PHV containers to each PHV container group size. This option is for information-only compilation. No configuration will be produced.
`--extra-tphv`: Doubles the capacity of tagalong PHV containers. This option is for information-only compilation. No configuration will be produced.
`--gen_asm` or `-S`: Generate Tofino assembly code.
`--no-dead-code-elimination`: Turn off dead code elimination optimization.
`--no-phase0`: Turn off possibility of using ingress buffer for first ingress match table.

`--number-mau-stages NUMBER_MAU_STAGES`: Specify the number of match action stages per pipeline. This option is for information-only compilation. No configuration will be produced. By default, this is 12.
`--output OUTPUT, -o OUTPUT`: Produce compiler output in the specified directory. By default, output is placed in the current working directory in **`<program name>.tofino`**
`--p4-14-spec`: Turn on P4-14 specification compliance with respect to unspecified output port handling (send to port 0).
`--p4-name P4_NAME`: The name to use for the program. By default, the file name is used, minus the ".p4" extension.
`--p4-prefix P4_PREFIX`: The prefix to use for the runtime APIs.
`--parse-state-merge {True,False}`: Enable the parse state merging optimization. Default: **`True`**
`--parser-timing-reports`: Generate parser timing reports.
`--placement-order {ingress_only, egress_only, ingress_before_egress, egress_before_ingress, ingress_and_egress}`: The order in which to place tables. By default, **`ingress_before_egress`** is used.
`--verbose {0,1,2}, -vl {0,1,2}`: Display various levels of verbose compilation messages
`--version`: Print the compiler version and exit.

1.4 SDE Version

Information in this document applies to version 6.1.0 and later of the Barefoot Capilano SDE, unless otherwise noted.

1.5 Other Documentation

In addition to this guide, you will find the following other materials useful while writing P4 programs. These documents are available from the Barefoot Networks support portal:

- Barefoot's P4 Tutorial introduces P4 programming.
- The P4 Optimization Guide for 10k Series explains how to design your P4 program to make use of Tofino-specific features.

1.6 Compatibility

The pragmas described in this document apply only to programs built for the Tofino ASIC. The p4c-tofino compiler is not used when building for other targets such as the BMv2-Tofino (tofinobm) behavioral software model. (For information on building for various targets including the BMv2-Tofino model, see the Barefoot Networks support article, Compiling P4 Programs using P4-Build.)
The primitives, hash functions, and black boxes described in this document are available for the Tofino ASIC, the Tofino model, and the BMv2-Tofino model. They are not understood by other targets.

A note about targets: As mentioned above, this document describes the p4c-tofino compiler, which builds programs for the Tofino ASIC and Tofino software model.
The p4c-tofino compiler is *not* used when building for other targets such as the BMv2-Tofino (tofinobm) behavioral software model. (For information on building for various targets including BMv2-Tofino, see the KB article, Compiling P4 Programs using P4-Build.)
For initial development and debugging, you may find it useful to compile for the BMv2 software model. The behavioral model does not enforce resource constraints of the hardware. For details, see the Barefoot Networks support article, Differences in simulation models.

2

Compiling P4 Programs using P4-Build and p4c-tofino

Introduction

The building of a P4 program is both similar to and somewhat different from the building of a program written in a "traditional" programming language, such as C.

The similarity is obvious: in both cases, you need to run a compiler and pass it the name of the file that contains the program source code, as well as other optional parameters (the location of include file directories, compilation options, and so on). For example, you can compile a C program with the following command:

```
$ gcc -o hello hello.c
```

You can also compile a P4 program using a similar looking command:

```
$ p4c-tofino switch.p4
```

What is different is the *compiler output*. While the output of a C compiler is a single object file or a single executable, the output of P4 compiler is much more complex and consists of many files that can be grouped in the following categories:

1. The compiled program in a target-specific format for the Tofino ASIC (or for running on its register-accurate model) the compiler generates a binary file `tofino.bin`.
2. The code for Program Dependent (PD) API that the control plane can use in order to control the device that runs the compiled P4 program. We often refer to this as the autogenerated API. It consists of the following parts:
    a. Declarations and implementation of all PD API functions
    b. (optional) Interface definitions for the PD API functions in Thrift IDL
    c. (optional) Thrift PD API server code
    d. Thrift client bindings for PD API in C++ and Python
3. Additional data for program-independent components, contained in a file called `context.json`.
4. Logs
5. Visualizations

It is the autogenerated code described above in (2) that complicates matters, since after it has been generated, it needs to be compiled to create libraries that can then be used by the control plane. The p4-build framework automates this task and allows P4 programs to be built using the same GNU Coding Standard (GCS) process as used to build the rest of SDE components (as well as most free software).

GNU Coding Standard and Build System

The detailed description of GNU Coding Standards can be found in GNU documentation and in various books. We will concentrate on the build process consisting of the following steps:

1. `configure`
2. `make`
3. `make install`

During the first step, the user runs the script, called `configure`, an instance of which is shipped with every package. The script probes the environment, checks the presence of dependencies and finally configures the package by creating custom `Makefile`s that are ideally suited to the given build. Configure scripts have many standard parameters that allow the users to override the defaults; package authors usually extend these parameters with the package-specific ones.

During the second step, the `make` utility is invoked, and it uses the `Makefile`s, generated in the previous step to build the package according to the previously specified configuration. Some of the build parameters can be overridden if needed by specifying the values for the corresponding Makefile variables on the command line. The third step installs the results of the build into the chosen directory.

VPATH builds

There are two main approaches to the placement of the compilation results. The first, traditional approach, calls for placing the results of the compilation (such as `.o` object files) alongside with the source files. This is a very simple and easy-to-understand approach, and it is the most often used one.
To use the standard approach, the configure script should be run directly in the source directory, like so:

```
./configure [options]
```

However, this simple method has its weaknesses: if one wants to perform several builds (for example, to try different compile-time options or to compile for multiple different targets), they would need to compile the code, install the artifacts, clean up, build another configuration, and so on.
VPATH builds allow users to keep the source separate and intact. Instead, the user can use multiple build directories that contain the Makefiles and build artifacts only, one directory per build configuration.
GCS-compliant distributions (including SDE and p4-build) support VPATH builds. All you need to do is to create a separate directory for the build and invoke the configure script from that directory.
For example, suppose you want to build two versions of BMv2: one with debugging and logging enabled and another one without them (to achieve maximum switching performance). This can be done easily by creating two separate directories and running configure like this:

```
#### For the debuggable BMv2 ####
$ mkdir -p $SDE/build/bmv2-debug
$ cd $SDE/build/bmv2-debug
$ $SDE/pkgsrc/bmv2-*/configure --prefix=$SDE_INSTALL --enable-debugger CXXFLAGS='-O0 -g'

#### For the fastest BMv2 ####
$ mkdir -p $SDE/build/bmv2-fast
$ cd $SDE/build/bmv2-fast
$ $SDE/pkgsrc/bmv2-*/configure --prefix=$SDE_INSTALL --disable-logging-macros CXXFLAGS='-O2'
```

Now, it is possible to execute make in each of these directories and get two different executables, one compiled for debugging and the other compiled for maximum performance. Notice how both times we invoked *the same* configure script located in the source directory of the BMv2 package. The difference was only in the parameters.
The P4-build package is based on the same idea and takes it even further, allowing the build infrastructure to be kept in one place (the `p4-build` package itself), the P4 source code to be kept in another, and the build to happen in the third place.

Standard Makefile Variables

The GNU Make utility defines a number of standard Makefile variables that allow users to easily customize build process without changing the Makefiles. GNU Coding Standards-compliant Makefiles use these and define other well-known Makefile variables to control the build process.
The `CXXFLAGS=` parameter used in the example above is one such variable. It is a set of options that should be passed to each invocation of the C++ compiler. You can specify these variables both during the invocation of the configure script or later, during the make:

```
$ make CXXFLAGS="-O2 -g"
```

The values specified during the `make` will override the values specified during the `configure`.
P4-build takes advantage of this mechanism and defines several new variables for the programs, written in P4:

- P4C – the path to the P4 compiler
- P4FLAGS – the options to be passed to the P4 compiler
- P4PPFLAGS – the options to be passed to the C preprocessor as it is being invoked by P4C
- P4_PATH – the path to the main file of the program that should be compiled
- P4_NAME – the name of the program (default is the basename of the file)
- P4_PREFIX – the program name to be used when generating PD APIs (default is P4_NAME)

Using P4-Build

While there are multiple ways P4 build can be used, we will describe the approach in which we keep the P4 source code separate from the SDE, but the compilation happens within the SDE, meaning that:

- The build will be performed in a subdirectory of `$SDE/build`
- Logs will be placed in a subdirectory of `$SDE/logs`
- Compiled artifacts will be installed in `$SDE_INSTALL`

The main reason you would use this approach is that, if you do it in this way, you can reuse the SDE scripts to exercise your program. Another reason is that you will be able to keep multiple builds of your program for each SDE version without doing anything special.
Let's assume that you have a directory, for example `~myprog` that has the following structure:

- `~/myprog/p4src` contains your P4 program, where the main file is called `myprog.p4`
- Optionally, `~/myprog/ptf-scripts` can contain your tests scripts.

Step 1: Create a build directory

We will create a directory `$SDE/build/customer` and, below that, we will keep subdirectories for each of the projects. So in this case we will do:

```
$ mkdir -p $SDE/build/customer/myprog
```

Step 2: Configure your project using P4-build

Now, we need to descend into the newly created build directory and call the `configure` script, provided by the p4-build package. The p4-build package is located in `$SDE/pkgsrc,` as are all other SDE packages.

```
$ cd $SDE/build/customer/myprog
$ $SDE/pkgsrc/p4-build-*/configure          \
          --prefix=$SDE_INSTALL             \
          --with-tofino                     \
          enable_thrift=yes                 \
          P4_PATH=~/myprog/p4src/myprog.p4 \
          P4_NAME=myprog
```

After the `configure` script finishes working, it will create a full directory structure and the necessary Makefiles so that you can compile your program, automatically generate all the APIs, and compile them into libraries.

Step 3: Compile your program

The whole point of the configure script is to make sure you can compile your program with one simple command:

```
$ make
```

And that's the only command that is really required. Of course, you can supply additional values for the Makefile variables as described above, but they are not necessary by default

Step 4: Install the artifacts

To install all the artifacts into $prefix (which, by default, is $SDE_INSTALL) you need to execute one last command:

```
$ make install
```

You are done!

Exercising your program

Step 1: Creating a configuration file

To be able to exercise your program using the sample application (bf_switchd), you need to create a configuration JSON file and place it in `$SDE_INSTALL/share/p4/targets/tofino` directory. The easiest way to create such a file is to use an example file, `tofino_single_device.conf.in.` from the p4-examples package:

```
$ sed -e 's/TOFINO_SINGLE_DEVICE/myprog/'               \
     < $SDE/pkgsrc/p4-examples/tofino_single_device.conf.in \
     > $SDE_INSTALL/share/p4/targets/tofino/myprog.conf
```

Step 2: Running switchd

Running switchd is described in the main SDE README file. Because all the artifacts are installed in the same places where SDE expects them, you can use standard SDE scripts:

```
$ cd $SDE
$ ./run_switchd.sh -p myprog
```

Step 3: Using PTF

If you have created PTF tests for your program and placed them in `~/myprog/ptf-tests`, you can use a standard SDE script to run them:

```
$ ./run_p4_tests -p myprog -t ~/myprog/ptf-tests
```

Additional Notes

The P4-build package can also be used to compile P4 code for BMv2-based targets, *bmv2-simple_switch* (often simply called *bmv2*) and *bmv2-tofino* (also known as *tofinobm*).

To do that, add the following parameters to the configure command line (in Step 2):

- To add compilation for bmv2-simple_switch add `--with-bmv2`
- To add compilation for bmv2-tofino add `--with-tofinobm`

The p4-build package contains a README file that describes many useful technical details, especially directory layout.

Convenience script

Available on the Barefoot Networks support site is a simple script, `p4build.sh`, that allows you to easily execute all the required build steps and create a default configuration file in one simple step.
In the simplest case, all you need to do is to make sure your environment variables $SDE, $SDE_INSTALL and $PATH point to the selected SDE directories and then execute the script, while pointing it to the P4 program you want to compile. For example:

```
$ ./p4_build.sh ~/examples/p4calc/p4src/p4calc.p4
Using SDE /home/sde/bf-sde-3.2.2.40
Using SDE_INSTALL /home/sde/bf-sde-3.2.2.40/install
Your path contains $SDE_INSTALL/bin. Good
Configuring p4calc in /home/sde/bf-sde-3.2.2.40/build/p4_build/ ... DONE
   Building p4calc ... DONE
 Installing p4calc in /home/sde/bf-sde-3.2.2.40/install ... DONE
```

By default, the script builds the program for Tofino (ASIC or model) and builds the Thrift bindings. It also passes the rest of the arguments to the configure script, so you can easily add other parameters, such as `--tofinobm`, `P4PPFLAGS=`, and so on.

3

Supported Compiler Pragmas

This chapter describes the pragmas supported by the Barefoot Networks P4 compiler, p4c-tofino.

3.1 Frontend

3.1.1 dont_trim

```
@pragma dont_trim
+attached to any object
```

Prevents the attached object from being trimmed out for not being referenced anywhere else in the P4 program. This also prevents the compiler from eliminating an object that has no effect.

3.2 Parser/Deparser

3.2.1 parser_value_set_size

```
@pragma parser_value_set_size [max_values]
```

This pragma is applied to a P4 parser value set object, and it indicates the maximum number of unique values that can be programmed in the value set. This value is used to reserve hardware resources to be able to implement all values. The default value of the max_values parameter is 4. The maximum supported value is 64.

3.2.2 max_loop_depth

```
@pragma max_loop_depth [count]
+attached to P4 parser state
```

Allows the parser to infer the maximum number of loops that could happen in ingress or egress in a state. This is used to flatten the parse graph when other inference mechanisms fail.

3.2.3 not_deparsed

```
@pragma not_deparsed [ingress/egress]
+attached to P4 header instances
```

Prevents the attached header instance from being deparsed in the specified gress (ingress or egress). Note that this pragma will not prevent POV bits from being allocated and manipulated for the instance. It will not prevent the header and its fields from being accessed in the deparser either. It just prevents POV bits from being referenced in the deparser.
Note that PHV allocation also uses this to help shrink packet field liveness.

3.2.4 not_parsed

```
@pragma not_parsed [ingress/egress]
+attached to P4 header instances
```

Indicates the attached header instance, though appearing in the parse graph, will never actually be parsed. This pragma is used by the PHV allocator to reduce the lifetime of the packet fields to be from first use to the deparser. Normally, packet fields are considered live from the parser to the deparser. The intended use case for this pragma is for packet headers that will be added during processing. Note that the packet header must still be extracted in the parse graph in order to infer the correct deparse order of the header. This can be achieved using an invalid select condition to branch to a parse state where the extraction occurs.

3.2.5 force_shift

```
@pragma force_shift [ingress/egress] [bits]
+attached to P4 parser state
```

Forces the attached state of the ingress or egress parser to shift out the specified number of *bits*, regardless of the state's contents. The *bits* value must be divisible by eight. If the state must be split by the compiler, this pragma will cause compilation to fail.

3.2.6 terminate_parsing

```
@pragma terminate_parsing ingress/egress
+attached to P4 parser state
```

This pragma is applied to a state. It informs the ingress/egress parser to ignore the instructions, header extractions, and branching decisions performed in that state and unconditionally end the parsing process. The header fields extracted in this state are not considered for PHV allocation unless they are used in other states. It is the programmer's responsibility to ensure that these fields are not used in the MAU pipeline.

3.2.7 packet_entry

```
@pragma packet_entry
+attached to P4 parser states
```

Indicates the attached parser state is a special entry point for certain kinds of packets. Other than 'start', the currently supported entry points are:

- 'start_i2e_mirrored', for packets mirrored via the clone_ingress_pkt_to_egress() primitive
- 'start_e2e_mirrored', for packets mirrored via the clone_egress_pkt_to_egress() primitive
- 'start_coalesced', for packets built out of samples from the sample_e2e() primitive

These entry points are all used only by the egress parser, but you should write your parser program as if it were at the beginning of the pipeline entering the normal ingress control flow. In other words: return ingress, even though in reality your code will exit to egress.

3.2.8 not_critical

```
@pragma not_critical [ingress/egress]
+attached to P4 parser states
```

This pragma is applied to a parse state. The pragma indicates the parser state, though appearing on the parser's critical path (the path that writes the most number of bits), should not be considered as on the critical parse path. The PHV allocator heuristically attempts to allocate fields that appear on the critical path into containers such that a balance is maintained across the different container sizes available. Using this pragma on a parser state that does not appear on the critical path will have no effect.

3.3 PHV Allocation

The following pragmas govern how the compiler will map packet fields and metadata into Tofino's packet header vector (PHV). A PHV is a collection of 8-, 16-, and 32-bit containers that store header data and metadata. The compiler analyzes packet field and metadata usage and maps it into containers in the PHV. The P4 control flows are processed in Tofino match-action units (MAUs) in the match-action pipeline, which operates on PHV containers. After the PHV has been operated on by the MAUs in the pipeline, Tofino's deparser reassembles the modified packet from the PHV.

Warning! Each PHV allocation pragma must be accompanied by the `instance name.field name` of the header field it refers to. The pragma is silently ignored if you mis-type the name of the header or header field.

Warning! All header/field related pragmas need to be attached to the header object before the header is instantiated. Otherwise, the pragmas will be silently ignored.

3.3.1 pa_atomic

```
@pragma pa_atomic gress instance_name.field_name
+ attached to P4 header instances
```

Specifies that the indicated packet or metadata field cannot be split across containers. The gress value can be either *ingress* or *egress*. For example, an 8-bit field could be placed in one 8-bit container or one 16-bit container or one 32-bit container. A 16-bit field could be placed in one 16-bit container or one 32-bit container. A 24-bit field could be placed in one 32-bit container.

3.3.2 pa_solitary

```
@pragma pa_solitary gress instance_name.field_name
+ attached to P4 header instances
```

Specifies that the indicated packet or metadata field cannot share a container with any other field. It can be overlaid with other field(s) if they are mutually exclusive. The gress value can be either *ingress* or *egress*. Header fields can be solitary only if they are byte-aligned and if their width is a multiple of eight bits. A constraint error will be generated otherwise.

3.3.3 pa_no_tagalong

```
@pragma pa_no_tagalong gress instance_name.field_name
+ attached to P4 header instances
```

Specifies that the indicated packet or or metadata field cannot be allocated in tagalong space. The gress value can be either *ingress* or *egress*. The pragma is ignored for metadata field instances, since metadata will never be allocated to tagalong space.

3.3.4 pa_container_size

```
@pragma pa_container_size gress instance_name.field_name 32
+ attached to P4 header instances
```

Specifies that the indicated packet or metadata field should be allocated to containers of the indicated size.
If one container size is specified, and the field is larger than that container size, the field will only be allocated in containers of that size. The field will occupy multiple containers of the same size.
If multiple container sizes are specified, the first container you specify will hold the most significant bits, and the last container you specify will hold the least significant bits. At runtime, containers are filled from the least significant container to most significant container. The most significant container may be partially filled. For example, if you have:

```
     @pragma pa_container_size ingress hdr1.x_60 8 16 8 32
```

This will be allocated to containers like:

```
     phv8[7:0] = hdr1.x_60[59:56]
```

```
        phv16[15:0] = hdr1.x_60[55:40]
        phv8[7:0] = hdr1.x_60[39:32]
        phv32[31:0] = hdr1.x_60[31:0]
```

Allowed container sizes are 8, 16, and 32. The gress value can be either *ingress* or *egress*.

3.3.5 pa_container

```
@pragma pa_container gress instance_name.field_name 124
+ attached to P4 header instances
```

Specifies that the indicated packet or metadata field should be allocated to a given PHV container number or container numbers.
If one container is specified and the field width is larger than the container, an error will be thrown.
If multiple container addresses are specified, the first container you specify will hold the most significant bits, and the last container you specify will hold the least significant bits. At runtime, containers are filled from the least significant container to most significant container. The most significant container may be partially filled.
For example, if you have:

```
        @pragma pa_container ingress hdr1.x_60 64 128 65 2
```

This will be allocated to containers like:

```
        phv64[7:0] = hdr1.x_60[59:56]
        phv128[15:0] = hdr1.x_60[55:40]
        phv65[7:0] = hdr1.x_60[39:32]
        phv2[31:0] = hdr1.x_60[31:0]
```

Allowed container addresses are [0:223] for MAU-addressable registers and [256:367] for tagalong addressable registers. The gress value can be either *ingress* or *egress*.
Error checking is performed to ensure you do not violate MAU or deparser assignment constraints for a given gress.

3.3.6 pa_do_not_bridge

```
@pragma pa_do_not_bridge egress instance_name.field_name
+ attached to P4 header instances
```

Specifies that the indicated metadata field should not be included in the auto-inferred bridged metadata field list that is passed from the ingress pipeline to the egress pipeline. This pragma is ignored if the field is already not going to be bridged.

3.3.7 pa_mutually_exclusive

```
@pragma pa_mutually_exclusive gress inst_1.field_1 inst_2.field_2
+ attached to P4 header inst_1
```

Specifies that the two indicated fields can be considered mutually exclusive of one another. PHV allocation uses field exclusivity to optimize container usage by overlaying mutually exclusive fields in the same container. This pragma does not guarantee that the two fields will occupy the same container. It gives the compiler the option to do so. The gress value can be either *ingress* or *egress*.

3.3.8 pa_no_overlay

```
@pragma pa_no_overlay gress inst_1.field_1
+ attached to P4 header instances
```

Specifies that the indicated field cannot be overlayed with any other fields. The gress value can be either *ingress* or *egress*.

3.3.9 pa_allowed_to_share

```
@pragma pa_allowed_to_share gress inst_1.field_1 inst_2.field_2
+ attached to P4 header instances
```

Specifies that the two indicated fields are allowed to share a container with one another. This pragma does not guarantee that the two fields will occupy the same container. It gives the compiler the option to do so. The gress value can be either *ingress* or *egress*. This pragma removes all container-sharing constraint checking except the container gress assignment.

3.3.10 pa_do_not_share_with_mirrored

```
@pragma pa_do_not_share_with_mirrored gress inst_name.field_name
+ attached to P4 header instances
```

Specifies that the indicated field cannot share a container with any other field that is mirrored. This pragma is a more specific instance of *pa_solitary*. The gress value can be either *ingress* or *egress*.

3.3.11 pa_no_init

```
@pragma pa_no_init gress inst_name.field_name
+ attached to P4 header instances
```

Specifies that the indicated metadata field does not require initialization to 0 before its first use because the control flow guarantees that, in cases where this field's value is needed, it will always be written before it is read.

Use this pragma if your program contains logic to ensure that it only evaluates this field's value when executing control paths in which a meaningful value has been written to the field, whereas for other control paths your program may read from the field but does not care what value appears there. For such programs, the compiler cannot determine that the control plane program logic guarantees a write before a desired read. This indeterminacy occurs when some control paths exist that require reading from the field before writing to it, while other control paths result in the field being read even though it was not written with a meaningful value, but in these latter control paths the program does not care what value appears in the field.

This pragma is ignored if the field's allocation does not require initialization. This pragma is also ignored if it is attached to any field other than a metadata field. The gress value can be either *ingress* or *egress*.

Use this pragma with care. If that unexpected control flow path is exercised, the field will have an unknown value.

3.3.12 pa_alias

```
@pragma pa_alias gress inst_1.field_name_1 inst_2.field_name_2
+ attached to P4 header instances
```

Specifies that the two packet and/or metadata field instances are to be considered as aliases to one another. Use this pragma with care, as it merges the constraints of both fields. The gress value can be either ingress or egress. The two fields must be the same bit width. A field can only be aliased with one other field currently.

3.3.13 pa_overlay_new_container_stop

```
@pragma pa_overlay_new_container_stop gress header_name stop_point
```

Packet headers that are not on the critical parse path are allocated by trying to overlay their fields in containers that are occupied by packet headers on the critical path. The compilation phase that performs this allocation step searches a number of packing options to look for a solution that will open up the fewest new containers (i.e.

containers where data could not be overlaid.) By default, the search stops if the header can be completely overlaid (stop_point = 0). This pragma allows a higher threshold to be used as the stopping point.
Unlike most PHV allocation pragmas, this pragma uses a *header instance name* rather than a field instance name. The stopping point value can be an integer greater than or equal to zero. The gress value can be either ingress or egress. This pragma will be silently ignored if a packet header on the critical path is specified or if a metadata struct name is used.

3.3.14 pa_manual_bridge_header

```
@pragma pa_manual_bridge_header gress header_instance_name
```

Specifies that a particular header instance should be treated as a user-defined metadata bridge header (data that is the sent from the ingress pipeline to the egress pipeline). When this pragma is used, the compiler will no longer automatically infer any bridged metadata. The fields in the *ingress instance* of the header will be treated as live from their first use until the ingress deparser. The fields in the *egress instance* of the header will be treated as live from the egress parser to their last use.
Note that manually bridged headers must still appear in the parse graph, so that the ingress deparser can infer the correct deparse ordering. It is recommended to "parse" the header in an unreachable parse state.

3.3.15 pa_overlay_stage_separation

```
@pragma pa_overlay_stage_separation gress field_name value
```

One criteria for whether two fields can be overlaid is to check if their MAU table usage liveness range overlaps. Since PHV allocation runs before table placement, the liveness range for PHV allocation is determined by their first and last use in an idealized table placement. When two fields are overlaid, PHV allocation must add initialization instructions on table control flow paths where the later live field is read before written. ($P4_{14}$ requires that uninitialized fields have a value of 0.) This adds a dependency on any control flow path with initialization instructions. To avoid inducing table dependencies that could further move tables, especially if tables end up moving due to resource limitations, PHV allocation introduces the concept of a liveness range stage separation. PHV allocation will not overlay fields if their liveness range gap is not at least two by default.

For example, if we had the following fields:

- *meta.x* - live from stage 1 to 3
- *meta.y* - live from stage 4 to 7
- *meta.z* - live from stage 5 to 11

meta.x and *meta.y* would not be considered as overlayable, because their stage separation is only 1. *meta.x* and *meta.z* would be considered as overlayable, since their stage separation is 2.

In this example, adding the pragma:

```
@pragma pa_overlay_stage_separation ingress meta.y 1
```

would allow meta.x and meta.y to be overlaid, since it overwrites the default stage separation of 2 for field *meta.y*.

The pragma is expected to be added to the field that is live later in the pipeline. In the above example, adding the pragma to *meta.x* would not result in *meta.x* and *meta.y* being overlaid. (*meta.x* could be overlaid with fields live earlier to it with less stage separation.)

3.4 MAU

3.4.1 stage

```
@pragma stage 5
+attached to P4 match tables
```

Specifies that the associated match table must be placed in the indicated stage. Optionally, a second value denoting the number of entries of the table to place in the stage may also be specified. For example:

```
@pragma stage 5 1024
```

The stage number is required. If the number of entries is not specified, an attempt will be made to place all table entries in the indicated stage.

Multiple stage pragmas may be specified for a table, each with their own number of entries. Currently, only one of the stage pragmas attached to a table may skip specifying the number of entries. When a stage pragma does not specify the number of entries, it effectively means the remaining table entries will be placed in that stage.

For example:

```
@pragma stage 4 1024
@pragma stage 6 1024
@pragma stage 7
```

This instructs the compiler to allocate the rest of the table entries to stage 7.

Allowed stage numbers are 0 to 11. The sum of the specified entries must be less than or equal to the table size. The compiler verifies the constraints to ensure that the table can actually be placed in the specified stage. Otherwise, the compiler will generate an error.

3.4.2 ways

```
@pragma ways 4
+attached to P4 match tables
```

Specifies that the associated match table, if it is an exact match table, should use the given number of **`ways`**. On Tofino, exact-match tables are implemented using Cuckoo hashing. The **`ways`** and '**`pack`**' values specify how many slots are available in each Cuckoo hash table (a single match-action table created by your P4 program may consist of many Cuckoo hash tables in memory).

Allowed values are integers greater than or equal to 1. Depending on the requirements of the match table, a large **`ways`** value may prevent the table from being placed.

3.4.3 pack

```
@pragma pack 2
+attached to P4 match tables
```

Specifies that the associated match table, if it is an exact match table, should pack the given number of entries per wide table word. See the description of the **`ways`** pragma for an explanation of how the **`ways`** and **`pack`** settings are used.

Allowed values are integers from 1 to 9. Depending on the requirements of the match table, a large pack value may prevent the table from being placed.

3.4.4 immediate

```
@pragma immediate 1
+attached to P4 match tables
```

Specifies that some parameters from the match table's associated action data table should try to be packed in match overhead instead of in the action data table. A `1` tells the compiler to try to use this optimization. A `0` tells the compiler to turn off the optimization. All other values are ignored. This optimization, where available, can sometimes reduce the memory requirements. The default value is `1`. Note that action parameters cannot be packed in match overhead for indirectly referenced action data tables.

3.4.5 force_immediate

```
@pragma force_immediate 1
+attached to P4 match tables
```

Specifies that all action parameters from the match table's associated action data table must be packed in match overhead instead of in the action data table. A 1 tells the compiler to force this packing. All other values are ignored. The default value is 0. Note that if all action parameters cannot be packed in match overhead, compilation will fail. Also note that this pragma cannot be used when the action data table is indirectly addressed.

3.4.6 action_entries

```
@pragma action_entries 2048
+attached to P4 match tables
```

Specifies that the match table's associated action data table is only expecting to require the given number of entries. If this number of entries is less than the number of match table entries, the action data table will be indirectly referenced. If the given value is larger than or equal to the number of match entries, the action data table will be directly referenced.

3.4.7 ternary

```
@pragma ternary 1
+attached to P4 match tables
```

Specifies that the match table should be placed within TCAM resources. A `1` tells the compiler to use ternary resources. All other values are ignored.

3.4.8 selector_max_group_size

```
@pragma selector_max_group_size 512
+attached to P4 match tables
```

Specifies that the match table's selector should allocate enough space for the indicated maximum group size. Valid values are in the range [2:119040]. Invalid values are ignored. By default, the maximum group size is 120 members.

3.4.9 selector_num_max_groups

```
@pragma selector_num_max_groups 32
+attached to P4 match tables
```

Specifies that the match table's selector should allocate enough resources to support the indicated number of groups of maximum size. By default, the number of maximum groups is 1.

3.4.10 idletime_precision

```
@pragma idletime_precision 3
+attached to P4 match tables
```

Specifies, if the match table utilizes *idle time* counters, what the counter precision should be. Valid values are 6, 3, 2, and 1. All other values are ignored. The default value is 3.

3.4.11 idletime_two_way_notification

```
@pragma idletime_two_way_notification 1
+attached to P4 match tables
```

For a match table that uses *idle time* counters, this pragma specifies whether notifications to the CPU should occur for transitions from active to inactive *and* inactive to active. A value of 1 enables notifications for both transitions. A value of 0 disables two-way notification, indicating that the CPU will be notified only when a flow transitions from active to inactive. All other values are ignored. The default value is 1 for precisions greater than 1 bit. Otherwise, the default is 0.

3.4.12 idletime_per_flow_idletime

```
@pragma idletime_per_flow_idletime 1
+attached to P4 match tables
```

For a match table that uses *idle time* counters, this pragma specifies whether flows utilize per-flow *idle time*. A value of 1 enables per-flow idle time. A value of 0 disables per-flow idle time. All other values are ignored. The default value is 1 for precisions of 3 and 6 bits. The default value is 0 for precisions of 1 and 2 bits. Turning on *idle time* per flow means the state space for counters is reduced by 1. Per-flow idle time cannot be used with a precision of 1.

3.4.13 entries_with_ranges

```
@pragma entries_with_ranges <number>
+ attached to P4 match tables
```

If the match table uses ranges, this pragma specifies how many of the entries are expected to have ranges. Ranges require expanding the depth of the physical memory resources because one entry may require multiple physical memory words. The pragma value is expected to be a number from 0 to the table size, inclusive. Values larger than the specified table size are capped at the table size. The default value is 25% of the table size for tables that can use ranges.

3.4.14 use_hash_action

```
@pragma use_hash_action 1
+ attached to P4 match tables
```

Specifies that the match table should use the hash-action path. This forces the table to use hash action, even if it is less efficient. If the table cannot be implemented using hash action, the compiler will report an error. A value of 1 forces hash action. A value of 0 turns off the optimization, even if it is preferred. All other values are ignored.

3.4.15 table_counter

```
@pragma table_counter event-type
+ attached to P4 match tables
```

For the associated match table, this pragma turns on event logging for the type of event specified with the *event-type* argument. By default, *table_hit* event logging is performed. Only one type of event can be counted at a time. Allowed *event-type* values are:

- *table_miss* - count how many times the table misses
- *table_hit* - count how many times the table hits

- *gateway_miss* - count how many times the attached gateway table misses
- *gateway_hit* - count how many times the attached gateway table hits
- *gateway_inhibit* - count how many times the attached gateway table inhibits the match table

Note: A non-intuitive aspect of these counters is that they count irrespective of the table's predication status. For example, the *table_hit* counter is incremented each time a table hit occurs, even if the table has been predicated off. As another example, if predication deems that a MAU stage is bypassed completely, memory accesses for those tables are powered off, which means the table misses and the *table_miss* counter is incremented.

3.4.16 clpm_prefix

```
@pragma clpm_prefix ipv4.dstAddr
+attached to P4 match tables
```

Specifies that the associated match table should be implemented as a "chained LPM" table, where the chain consists of multiple logical tables, one per prefix length (or range of prefixes) specified. Chained LPM (CLPM) provides multi-stage LPM lookup with a separate table for each prefix length or range of lengths. To support LPM lookup in memory, the match key need only be as long as the prefix key, leading to better memory efficiency. The prefix length tables are exact match tables. The tables associated with prefix ranges are ternary tables. The field specified must have an *lpm* match type. All other fields in the match key must use the *exact* match type.

3.4.17 clpm_prefix_length

```
@pragma clpm_prefix_length <prefix size> <number of entries>
```

or

```
@pragma clpm_prefix_length <prefix size min> <prefix size max> <number of
entries>
```

Example:

```
@pragma clpm_prefix_length 16 1024
@pragma clpm_prefix_length 32 1024
@pragma clpm_prefix_length 17 24 4096
+attached to P4 match tables
```

Specifies what prefix lengths to use for a *CLPM* table and how many entries to allocate for that prefix. Prefixes are assumed to cover the most significant bits of the field. Multiple prefix lengths can be specified per table. The chain will be ordered so that the search begins with the longest prefix and proceeds to the shortest prefix. Any holes in the prefix space will NOT be covered by any table. In the above example, the first table in the chain would be a /32 exact-match table, followed by a ternary table that handles /17 to /24 prefix matches, followed by a /16 exact-match table. It is an error if no prefix lengths are specified but a prefix field is specified.

3.4.18 proxy_hash_width

```
@pragma proxy_hash_width 32
+ attached to P4 match tables
```

Specifies that an exact match table should be specified with a proxy hash result stored as the key. Two hash computations are performed. The first hash computes the entry location in memory. The second hash computed is stored in place of the exact match key. This pragma can only be used if the table can be implemented as an exact match table. Valid bit widths are in the range [1:52]. Proxy hashing is off by default.

3.4.19 proxy_hash_algorithm

```
@pragma proxy_hash_algorithm crc16
+ attached to P4 match tables
```

Specifies the hash algorithm to use to compute the proxy hash value to store in place of the match key. This is ignored if the pragma *proxy_hash_width* is not specified. By default, the hash algorithm used is *random*.

3.4.20 atcam_partition_index

```
@pragma atcam_partition_index atcam_meta.partition_index
+ attached to P4 match tables
```

Specifies the packet or metadata field to be used as the partition index for an algorithmic TCAM implementation. It is an error if this field is not part of the algorithmic TCAM table's match key. This pragma turns on the algorithmic TCAM implementation path.

3.4.21 atcam_number_partitions

```
@pragma atcam_number_partitions 1024
+ attached to P4 match tables
```

Specifies the number of partitions to use for an algorithmic TCAM implementation. By default, the value will be 2^n where n is the partition index bit width. The pragma value is ignored unless `atcam_partition_index` is specified. It is an error if this value is greater than 2^n (where n is the partition index bit width).

3.4.22 alpm

```
@pragma alpm 1
+ attached to P4 match tables
```

Specifies that the associated match table should be implemented as an algorithmic LPM table instead of using normal TCAM resources. A value of '1' turns the feature on. All other values are ignored. It is an error if the match key does not have one, and only one, field with a match type of `lpm`. Note that this implementation approach will infer a TCAM partition pre-classifier that sets a metadata field for a partition index. This implies that, at a minimum, any table implemented in this way will require two MAU stages. Setting the partition index requires a match dependency between the TCAM pre-classifier and the remainder of the table.

3.4.23 alpm_partitions

```
@pragma alpm_partitions 2048
+ attached to P4 match tables
```

If the associated match table has been specified to be implemented as an algorithmic LPM table (see the `alpm` pragma), the `alpm_partitions` pragma specifies how many partitions to use. The only supported values are 1024, 2048, 4096, 8192. The default is 1024.

3.4.24 alpm_subtrees_per_partition

```
@pragma alpm_subtrees_per_partition 2
+ attached to P4 match tables
```

If the associated match table has been specified to be implemented as an algorithmic LPM table (see the `alpm` pragma), the `alpm_subtrees_per_partition` pragma specifies how many subtrees are supported per partition. From a resource perspective, this indicates how many TCAM pre-classifier entries will be allocated. For example, if the subtrees per partition is 2 and the number of partitions is 1024, the TCAM pre-classifier will have 2048 entries. Allowed values are in the range [1:10]. The default value is 2.

3.4.25 lrt_enable

`@pragma lrt_enable [`**`1`**`|0]`

`+ attached to P4 statistics tables`

By default, Tofino performs automatic cache evictions of counter values from Tofino to the external CPU to prevent counter overflow. (Eviction is governed by the LR(t) algorithm, hence the name of this pragma.) A value of 0 disables LR(t) automatic eviction. A value of 1 enables LR(t) automatic eviction. You should only disable LR(t) automatic eviction if you're certain you will not overflow your on-chip counters or you will read and clear the counters faster than they can overflow. LR(t) messages cannot be used if full width counters (64-bits) are in use. By default, LR(t) is enabled for counter sizes less than 64 bits.

3.4.26 meter_sweep_interval

`@pragma meter_sweep_interval [1..20]`

`+ attached to P4 meter tables`

Specifies that the associated meter table should use the specified meter sweep interval adjustment. Allowed values are integers in the range [0:4]. For sweep interval adjustment values 0 through 4, the sweep interval, expressed in clock cycles, is calculated as:

$$sweep\ interval = 2^{(22+n)}$$

where n is the sweep interval adjustment.

Values 5 through 20 should only be used when there is a shifting time representation. The default value is 2, which corresponds to 2^{24} clock cycles.

3.4.27 bind_indirect_res_to_match

`@pragma bind_indirect_res_to_match <name of indirect resource (e.g. counter)>`

`+ attached to a P4 action profile`

By default, indirect resources (meter, counter, stateful) are logically bound to action entries, not match entries. This means that, by default, when using an indirect resource in conjunction with an action profile object, two match entries cannot share the same action data, but use different resource indices. This pragma helps overcome this limitation. It is used as follows:

```
action custom_action_1(dstAddr, idx) {
    modify_field(ipv4.srcAddr, ipAddr);
    count(cntDum, idx);
}
@pragma bind_indirect_res_to_match cntDum
action_profile custom_action_1_profile {
    actions { nop; custom_action_1; }
    size : 2048;
}

counter cntDum {
    type : packets;
    instance_count : 256;
}
```

In the above example, the `cntDum` entries will be bound to the match entries of whichever match table uses `custom_action_1_profile`.

The corresponding PD auto-generated entry add function will look like this:

```
p4_pd_status_t p4_pd_exm_indirect_1_exm_4ways_6Entries_add_entry(
    p4_pd_sess_hdl_t sess_hdl,
    p4_pd_dev_target_t dev_tgt,
    p4_pd_<prog_name>_<table_name>_match_spec_t *match_spec,
    p4_pd_mbr_hdl_t mbr_hdl,
    int cntDum_index,
    p4_pd_entry_hdl_t *entry_hdl
);
```

3.4.28 dynamic_table_key_masks

`@pragma dynamic_table_key_masks 1`
`+attached to P4 match tables`
Specifies that the associated match table, if it is an exact match table, will support runtime-programmable masks for fields in the table key. A value of 1 turns this feature on. All other values are ignored. It is an error if this pragma is applied to a table that must be implemented in TCAM resources or as an algorithmic TCAM. This pragma is supported in Capilano SDE 4.1.0 and later.

3.4.29 ignore_table_depenency

`@pragma ignore_table_dependency table_name`
`+attached to P4 match tables`
Specifies that the associated match table should ignore match and/or action dependencies to the indicated table. Use this pragma with care. It is assumed that the real dependency will be enforced by the control plane. Note that this pragma will be silently ignored if it is used on tables that belong to different gress values.

3.4.30 force_table_depenency

`@pragma force_table_dependency table_name`
`+attached to P4 match tables`
Specifies that the associated match table should force a match dependency to the indicated table. Adding an artificial dependency will prevent the two tables from being placed in the same stage. Note that this pragma will be silently ignored if it is used on tables that belong to different gress values.

3.4.31 egress_pkt_length_adjust_in_ternary

`@pragma egress_pkt_length_adjust_in_ternary 1`
`+attached to any P4 match table`
Specifies that the compiler-introduced table (called *__egress_pkt_length_update__*) that adjusts the *eg_intr_md.pkt_length* value based on whether it is a mirrored or non-mirrored packet should be placed within TCAM resources. A value of `1` instructs the system to use ternary resources. All other values are ignored. By default, the table is implemented as an exact match table if multiple adjustment values are possible. This pragma is ignored if this table is not required. It is also ignored if there is only one possible adjustment value, as this will be implemented in a gateway alone. Unlike most MAU pragmas, this one can be attached to any P4 match table object.

3.4.32 no_egress_length_correct

`@pragma no_egress_length_correct 1`
`+attached to any P4 match table`
Specifies that the compiler-introduced table, **`__egress_pkt_length_update__`** (which adjusts the **`eg_intr_md.pkt_length`**) is not needed. A value of 1 indicates that packet length will not be corrected. All other values are ignored. By default, the packet length correction is performed in all scenarios. This pragma is

ignored if this table is not required. Unlike most MAU pragmas, this one can be attached to any P4 match table object.

3.4.33 no_egress_length_correct_for_mirror

```
@pragma no_egress_length_correct_for_mirror 1
+attached to any P4 match table
```

Specifies that the compiler-introduced table, *__egress_pkt_length_update__* (which adjusts the *eg_intr_md.pkt_length*) does not need to consider mirror scenarios. A value of `1` indicates that mirror packet lengths will not be corrected. All other values are ignored. By default, the packet length correction is performed in all scenarios. This pragma is ignored if this table is not required. It is also ignored if there is only one possible adjustment value, as this will be implemented in a gateway alone. Unlike most MAU pragmas, this one can be attached to any P4 match table object.

3.4.34 symmetric

```
@pragma symmetric field_instance_1 field_instance_2
+ attached to P4 field list objects
```

Specifies that the two fields specified are to be considered symmetric when computing a hash result. This pragma is only available in the context of field list hash calculations used by action selectors. All other usages are silently ignored. It is an error if the two fields are not the same bit width.

3.4.35 phase0

```
@pragma phase0 1
+ attached to P4 match tables
```

Specifies that the indicated table must or must not be implemented as the “Phase 0” table. A ‘1’ indicates the table must be implemented as phase 0. A ‘0’ indicates the table should not be implemented as phase 0. All other values result in an error.
If the table is required to be implemented as phase 0 but cannot for other constraint reasons, compilation fails. A phase 0 table is implemented using the parser, by using per-port configuration data. Utilizing phase 0 is one mechanism to eliminate MAU table dependencies. Only one table can be implemented as the phase 0 table, and it is subject to following requirements and restrictions:

- The table must be the first table of the ***ingress*** control flow
- The table has no side effect tables other than action data
- The table has the following condition gating its execution: `(ig_intr_md.resubmit_flag == 0)`
- The table key is an exact match lookup on `ig_intr_md.ingress_port`
- The maximum table size is *288* entries
- There is only one action and it is also specified as the default action
- The single action can only perform assignment operations (i.e. `modify_field`) from action parameters or constants.
- The sum of the PHV containers width the fields written by the action reside in is less than or equal to 64 bits. Note that packing constraints may not allow all fields to be allocated in the same container or contiguously in a container.
- The action does not modify fields that would otherwise be written by the parser (i.e. written by a `set_metadata` call)

Here is an example P4 snippet:

```
action set_port_info(port_info, port_type) {
    modify_field(port_metadata.port_info, port_info);
    modify_field(port_metadata.port_type, port_type);
    modify_field(port_metadata.port_up, 1);
}

table ingress_port_mapping {
    reads {
        ig_intr_md.ingress_port : exact;
    }
    actions {
        set_port_info;
    }
    default_action : set_port_info;
    size : 288;
}

control ingress {
    if (ig_intr_md.resubmit_flag == 0) {
        apply(ingress_port_mapping);
    }

    // other apply statements
}
```

3.4.36 force_match_dependency

`@pragma force_match_dependency gress stage`

`+ attached to any P4 object`

Specifies that the indicated ingress or egress stage should be configured to have a match dependency to the previous stage. By using the `force_match_dependency` pragma instead of the command line argument, `--force-match-dependency`, you achieve more precise control on stage dependencies.

This pragma prevents a given stage from being executed concurrently with the previous one. The result is that the program latency increases, but energy consumption decreases.

The gress value can be either ingress or egress. The stage number can be an integer in the range [0:11].

3.4.37 stateful_field_slice

`@pragma stateful_field_slice field_name msb lsb`

`+ attached to stateful ALU black box definitions`

Specifies that the named field's usages in the black box are actually field slices. The syntax of stateful ALU blackboxes allows you to refer to metadata fields that are wider than the register size (1/8/16 or 32 bits, depending on the register and the blackbox). This pragma allows you to specify exactly which bits of the wider field (slice) should participate in the computation.

This pragma is intended to work around P4_14's lack of field slicing syntax. All usages of the named field in the blackbox will be interpreted as the field slice.

The least significant bit of the slice must begin at an eight-bit boundary (that is, at bit 0, at bit 8, at bit 16, at bit 24, or at another bit in this series).

Example:

```
@pragma stateful_field_slice ethernet.dstAddr 47 16
blackbox stateful_alu bbox {
    reg: reg1;
// This is interpreted to mean ethernet.dstAddr[47:16] != 0
    condition_lo: ethernet.dstAddr != 0;
    update_lo_1_value: register_lo + 1;
    output_value : alu_lo;
    output_dst : meta.x;
}
```

3.4.38 optional_field

```
@pragma optional_field <field name>
+ attached to P4 field lists
```

Specifies that the indicated field in a P4 field list can be optionally removed from participating in a hash calculation at run time.

3.4.39 all_fields_optional

```
@pragma all_fields_optional
+ attached to P4 field lists
```

Specifies that all the indicated fields in a P4 field list can be optionally removed from participating in a hash calculation at run time.

3.4.40 max_actions

```
@pragma max_actions 16
+ attached to P4 match tables
```

This pragma is intended for future-proofing against table layout changes caused by adding actions at a later date. The user specifies the maximum supported actions that will ever be allowed for this table. Normally, the compiler sizes the action instruction pointer based on how many actions can be run as a result of a table hit. For example, three actions requires two bits of action instruction pointer storage in match overhead. If the user indicated a maximum actions value of six, for example, this tells the compiler to allocate action instruction pointer overhead assuming six actions instead of three. This results in three bits of action instruction pointer space being used.

Any positive integer is allowed as the value, but an error will be thrown if the maximum actions value is less than the number of currently listed table actions.

3.4.41 egress_pkt_length_stage

```
@pragma egress_pkt_length_stage #
+attached to any P4 match table
```

Specifies that the compiler-introduced table (called *__egress_pkt_length_update__*) that adjusts the *eg_intr_md.pkt_length* value based on whether it is a mirrored or non-mirrored packet should be placed in a specific MAU stage. Values of 0 through 11 are supported. All other values, including invalid values, are ignored. Unlike most MAU pragmas, this one can be attached to any P4 match table object.

3.4.42 field_list_field_slice

```
@pragma field_list_field_slice <field name> <msb bit> <lsb bit>
+ attached to P4 field list definitions
```

Specifies that the named field's usages in the field list are actually field slices. This pragma is intended to work around P4_14's lack of field slicing syntax. All usages of the named field in the field list will be interpreted as the field slice. This pragma is only picked up when the field list is used to compute a hash value in the MAU. Only one slice definition is allowed per field.

Example:

```
@pragma field_list_field_slice ipv4.dstAddr 19 4
@pragma field_list_field_slice ipv4.srcAddr 27 20
field_list fl_1 {
    ipv4.dstAddr;
    ipv4.protocol;
    ipv4.srcAddr;
}

field_list_calculation hash_1 {
   input {
       fl_1;
   }
   algorithm {
       identity;
   }
   output_width : 32;
}
```

This will result in a 32-bit identity hash computed that is populated as:
{ipv4.dstAddr[19:4], ipv4.protocol, ipv4.srcAddr[27:20]}.

3.4.43 hash_algorithm

```
@pragma hash_algorithm <algorithm name>
+ attached to P4 match entry tables
```

Specifies that the indicated hash algorithm should be used as an exact match table's hash function. If the named algorithm does not provide enough hash bits, the algorithm is automatically converted to its 'extended' version (the output will be repeatedly concatenated until enough bits are available.)

3.4.44 meter_profile

```
@pragma meter_profile <profile number>
+ attached to P4 meter objects
```

Specifies what meter rate range profile to use for a color-based meter. The table below shows the different rate ranges that are supported for each meter profile. The profile only needs to be specified if very low rate metering is required. The default profile is 0.

Profile	Min Rate (Bytes/sec)	Max Rate (Bytes/sec)	Max Rate (Pkts/sec)
0	9.46	648970000000	1270000000
1	4.73	324485000000	635000000
2	2.37	162242500000	317500000
3	1.18	81121250000	158750000
4	0.59	40560625000	79375000
5	0.30	20280312500	39687500
6	0.15	10140156250	19843750
7	0.074	5070078125	9921875
8	0.037	2535039063	4960938
9	0.018	1267519531	2480468
10	0.0092	633759766	1240234
11	0.0046	316879883	620117
12	0.0023	158439941	310059
13	0.0012	79219970	155029
14	0.00058	39609985	77515
15	0.00029	19804992	38757

3.4.45 use_conainer_valid

```
@pragma use_container_valid <valid field name> <field name>
+ attached to P4 match tables (ternary only)
```

Specifies that the valid field name, *V*, indicated should be matched on using the container validity bit from field name, *F*. For ternary tables, this optimization enables reclaiming the two adjacent bits next to the version overhead field in the most significant nibble of a TCAM word. Both *V* and *F* must both be part of the table key. Additionally, *F* must be a parsed packet header field. This optimization cannot be used for metadata fields, because the P4_14 spec indicates that metadata fields are always valid (and will thus always be in a valid container).

This optimization is only available for match tables that will be implemented with ternary resources, but it cannot be used if the table has one or more fields that have the range match type. Attempting to use this pragma for exact match and algorithmic tcam tables will be silently ignored. At most two of these pragmas can be used per match table.

To enable this optimization, PHV allocation must ensure that F is not overlaid in a container, so an implicit no-overlay constraint is added to the field.

Example:
```
// This pragma prevents vlan.id from being overlaid in a container,
// which allows its container validity bit to be used instead of
vlan.valid.
@pragma use_container_valid vlan.valid vlan.id
@pragma use_container_valid ipv4 ipv4.diffserv
table t0 {
    reads {
      vlan.priority : exact;
      vlan.cfi : exact;
      vlan.id : exact;
      vlan.etherType : exact;
      ipv4.diffserv : ternary;
      vlan.valid : ternary;
      ipv4 : valid;
    }
    actions {
       do_nothing;
       set_p;
    }
}
```

Using this optimization allows us to pack the above table in a single TCAM, because the vlan and ipv4 valid bits are placed adjacent to the version bits (an overhead field).

4

Supported Compiler Primitives

4.1 Limitations on the Use of Primitives in Default Actions

Tofino device places certain restrictions on the primitives that use Hash Distribution Units. Due to the fact that some phases of their evaluation need to happen at the same time the match is happening, actions containing these primitives might not always be allowed to be used as default actions in the corresponding tables.
The following primitives cannot be used in a default action in a table with one or more keys (a table that has a `reads` block):

- o `count` (if the counter index comes from a header or metadata field)
- o `count_from_hash`
- o `execute_meter` (if the meter index comes from a header or metadata field or if it uses pre-color)
- o `execute_meter_from_hash`
- o `execute_meter_with_or` (if the meter index comes from a header or metadata field or if it uses pre-color)
- o `execute_meter_from_hash_with_or`
- o `execute_stateful_alu` (if the register index comes from a header or metadata field)
- o `execute_stateful_alu_from_hash`
- o `modify_field_rng_uniform`
- o `modify_field_with_hash_based_offset`

These primitives can still be used in a default action in a keyless table (a table without a `reads` block), except in cases where the table requires additional overhead or action data (including constants that cannot be encoded directly in the instructions), because these will generally prevent the compiler from placing a table in a gateway. Note that this restriction prevents `count()`/`execute_meter()`/`execute_stateful_alu()` with a constant or table-specified index from being used in a default action, even in a keyless table.

4.2 Arithmetic Primitives

4.2.1 add

4.2.1.1 P4 Syntax

```
add(dst, src1, src2)
```

4.2.1.2 Description

Performs the operation `dst = src1 + src2`.
The *dst* field determines whether the addition is signed or unsigned, and whether it is saturating.

4.2.1.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.1.4 Restrictions

- *dst*, *src1*, and *src2* all must be the same bit width.
- The maximum supported bit width is 64.

- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in *Table 1*, below.

In the table below, we show the limitations that apply when you pass specific combinations of operand types. In the table, we indicate the operand type using this notation:

- *field1* represents the first field passed, which will also receive the result value,
- *field2* and *field3* represent the second and third fields passed, if present
- *param1* and *param2* represent the first and second action parameters passed, if present
- *1* and *2* represent the first and second immediate (constant) values, if present

Not bullet.

Operation	**Supported?**	**Reason/Comment**
primitive(field1, field2, field3)	Yes	
primitive(field1, field2, param1)	Yes	
primitive(field1, param1, field2)	Yes	
primitive(field1, field2, 1)	Yes	When calling subtract() with a constant as the third operand, the constant is only allowed to be a value between -8 and 7, inclusive.
primitive(field1, 1, field2)	Yes	
primitive(field1, param1, param2)	No	Cannot use two action parameters at the same time.
primitive(field1, param1, 1)	No	Cannot use action parameter and immediate value at the same time.
primitive(field1, 1, param1)	No	Cannot use action parameter and immediate value at the same time.
primitive(field1, 1, 2)	No	Not supported. Use modify_field, instead.

Table 1: Type restrictions for primitives with two operands.

4.2.2 add_to_field

4.2.2.1 P4 Syntax

```
add_to_field(dst, src1)
```

4.2.2.2 Description

Performs the operation `dst = dst + src1`.
The dst field determines whether the addition is signed or unsigned. The dst field determines whether the operation is saturating.

4.2.2.3 Parameters

- *dst*: a packet or metadata field.

- *src1*: a packet field, metadata field, action parameter, or immediate value.

4.2.2.4 Restrictions

- dst and src1 must be the same bit width.
- The maximum supported bit width is 64.

4.2.3 bit_and

4.2.3.1 P4 Syntax

```
bit_and(dst, src1, src2)
```

4.2.3.2 Description

Performs the operation `dst = src1 & src2`.

4.2.3.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.3.4 Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 (in the "add" section, above).

4.2.4 bit_andca

4.2.4.1 P4 Syntax

```
bit_andca(dst, src1, src2)
```

4.2.4.2 Description

Performs the operation `dst = ~src1 & src2`.

4.2.4.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.4.4 Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 (in the "add" section, above).

4.2.5 bit_andcb

4.2.5.1 P4 Syntax

```
bit_andcb(dst, src1, src2)
```

4.2.5.2 Description

Performs the operation `dst = src1 & ~src2`.

4.2.5.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.5.4 Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the "add" section, above.

4.2.6 bit_nand

4.2.6.1 P4 Syntax

```
bit_nand(dst, src1, src2)
```

4.2.6.2 Description

Performs the operation `dst = ~(src1 & src2)`.

4.2.6.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.6.4 Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the "add" section, above.

4.2.7 bit_nor

4.2.7.1 P4 Syntax

```
bit_nor(dst, src1, src2)
```

4.2.7.2 Description

Performs the operation `dst = ~(src1 | src2)`.

4.2.7.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.7.4 Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the "add" section, above.

4.2.8 bit_not

4.2.8.1 P4 Syntax

```
bit_not(dst, src1)
```

4.2.8.2 Description

Performs the operation `dst = ~src1`.

4.2.8.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.

4.2.8.4 Restrictions

- dst and src1 must be the same bit width.

4.2.9 bit_or

4.2.9.1 P4 Syntax

```
bit_or(dst, src1, src2)
```

4.2.9.2 Description

Performs the operation `dst = src1 | src2`.

4.2.9.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.

- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.9.4 Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the "add" section, above.

4.2.10 bit_orca

4.2.10.1 P4 Syntax

```
bit_orca(dst, src1, src2)
```

4.2.10.2 Description

Performs the operation `dst = ~src1 | src2`.

4.2.10.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.10.4 Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the "add" section, above.

4.2.11 bit_orcb

4.2.11.1 P4 Syntax

```
bit_orcb(dst, src1, src2)
```

4.2.11.2 Description

Performs the operation `dst = src1 | ~src2`.

4.2.11.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.11.4 Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the "add" section, above.

4.2.12 bit_xnor

4.2.12.1P4 Syntax

```
bit_xnor(dst, src1, src2)
```

4.2.12.2Description

Performs the operation `dst = ~(src1 ^ src2)`.

4.2.12.3Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.12.4Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the "add" section, above.

4.2.13 bit_xor

4.2.13.1P4 Syntax

```
bit_xor(dst, src1, src2)
```

4.2.13.2Description

Performs the operation `dst = src1 ^ src2`.

4.2.13.3Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.13.4Restrictions

- dst, src1, and src2 all must be the same bit width.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the "add" section, above.

4.2.14 max

4.2.14.1P4 Syntax

```
max(dst, src1, src2)
```

4.2.14.2Description

Performs the operation `dst = max(src1, src2)`.
The dst field determines whether the operation is signed or unsigned. The dst field determines whether the operation is saturating.

4.2.14.3Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.14.4Restrictions

- dst, src1, and src2 all must be the same bit width.
- The maximum supported bit width is 32.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the “add” section, above.

4.2.15 min

4.2.15.1P4 Syntax

```
min(dst, src1, src2)
```

4.2.15.2Description

Performs the operation `dst = min(src1, src2)`.
The dst field determines whether the operation is signed or unsigned. The dst field determines whether the operation is saturating.

4.2.15.3Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field, metadata field, action parameter, or immediate value.

4.2.15.4Restrictions

- dst, src1, and src2 all must be the same bit width.
- The maximum supported bit width is 32.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the “add” section, above.

4.2.16 subtract

4.2.16.1P4 Syntax

```
subtract(dst, src1, src2)
```

4.2.16.2Description

Performs the operation `dst = src1 - src2`.
The dst field determines whether the subtraction is signed or unsigned. The dst field determines whether the operation is saturating.

4.2.16.3Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: a packet field or metadata field.

4.2.16.4Restrictions

- dst, src1, and src2 all must be the same bit width.
- The maximum supported bit width is 64.
- Only one of the sources can be an action parameter and/or immediate. The second source must be a packet or metadata field.
- Operand type restrictions, as specified in Table 1 in the "add" section, above.

4.2.17 subtract_from_field

4.2.17.1P4 Syntax

```
subtract_from_field(dst, src1)
```

4.2.17.2Description

Performs the operation `dst = dst - src1`.
The dst field determines whether the subtraction is signed or unsigned. The dst field determines whether the operation is saturating.

4.2.17.3Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field or metadata field.

4.2.17.4Restrictions

- dst and src1 must be the same bit width.
- The maximum supported bit width is 64.

4.3 Field Modification and Shift Primitives

4.3.1 shift_left

4.3.1.1 P4 Syntax

```
shift_left(dst, src1, src2)
```

4.3.1.2 Description

Performs the operation `dst = src1 << src2`.

4.3.1.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: an immediate value.

4.3.1.4 Restrictions

- dst and src1 must be the same bit width.
- The maximum supported bit width is 32.
- src2 must be an immediate value.

4.3.2 shift_right

4.3.2.1 P4 Syntax

```
shift_right(dst, src1, src2)
```

4.3.2.2 Description

Performs the operation `dst = src1 >> src2`.
The dst field determines whether the operation is signed or unsigned.

4.3.2.3 Parameters

- *dst*: a packet or metadata field.
- *src1*: a packet field, metadata field, action parameter, or immediate value.
- *src2*: an immediate value.

4.3.2.4 Restrictions

- dst and src1 must be the same bit width.
- The maximum supported bit width is 32.
- src2 must be an immediate value.

4.3.3 swap

4.3.3.1 P4 Syntax

```
swap(field_1, field_2)
```

4.3.3.2 Description

Performs the operation `field_1, field_2 = field_2, field_1`

4.3.3.3 Parameters

- *field_1*: a packet or metadata field.
- *field_2*: a packet or metadata field.

4.3.3.4 Restrictions

- *field_1* and *field_2* must be the same bit width.

4.3.4 modify_field

4.3.4.1 P4 Syntax

```
modify_field(dst, src, mask)
```

4.3.4.2 Description

Performs the operation `dst = (dst & ~mask) | (src & mask)`

4.3.4.3 Parameters

- *dst*: a packet or metadata field.
- *src*: a packet field, metadata field, action data parameter, or an immediate.
- *mask*: an optional integer specifying which bits of the destination field to modify. If a mask is not specified, the entire destination will be modified.

4.3.4.4 Restrictions

- There is limited support for assigning a larger destination from a smaller bit width source.

4.3.5 modify_field_conditionally

4.3.5.1 P4 Syntax

```
modify_field_conditionally(dst, condition, src)
```

4.3.5.2 Description

Performs the operation `dst = condition ? src : dst`

4.3.5.3 Parameters

- *dst*: a packet or metadata field.
- *condition:* action data parameter or an immediate.
- *src*: action data parameter or an immediate.

4.3.5.4 Restrictions

- Neither *condition* nor *src* can come from a phv.
- This is an expensive operation, because it requires at least twice the destination field's bit width storage in an action data table.
- *condition* cannot be used in any other primitives within the action function (temporary restriction)

4.3.6 modify_field_rng_uniform

4.3.6.1 P4 Syntax

```
modify_field_rng_uniform(dst, lower_bound, upper_bound)
```

4.3.6.2 Description

Performs the operation `dst = random(lower_bound, upper_bound)`, where the random number produced is chosen in the inclusive range from lower bound to upper upper bound.

4.3.6.3 Parameters

- *dst*: a packet or metadata field.
- *lower_bound*: an integer that must be 0.
- *upper_bound*: an integer that must be 2^n - 1, where *n* is allowed to be in the range [1:32]

4.3.6.4 Restrictions

- The maximum supported destination field bit width is 32.
- The lower bound must always be 0.
- The upper bound must be one less than a power of two.
- Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, "Limitations on the Use of Primitives in Default Actions."

4.3.7 modify_field_with_hash_based_offset

4.3.7.1 P4 Syntax

```
modify_field_with_hash_based_offset(dst, base, field_list_calc, size)
```

4.3.7.2 Description

Performs the operation `dst = hash(field_list) + base`, where the hash result is limited to the range `[0:size-1]`.

4.3.7.3 Parameters

- *dst*: a packet or metadata field.
- *base*: an integer indicating an offset to add to the computed hash.
- *field_list_calc*: A reference to a P4 hash field list calculation, which specifies a list of fields to perform a hash computation on and the hash algorithm to use.
- size: an integer indicating the maximum hash result value to use. Note that this must be a power of two.

4.3.7.4 Restrictions

- o Size must be an integer that is a power of two.
- o The maximum hash result width supported is 32 bits.
- o The sum of the hash result bit widths for primitives called in an action must be less than or equal to 32 bits.
- o Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, "Limitations on the Use of Primitives in Default Actions."

4.3.8 modify_field_with_shift

4.3.8.1 P4 Syntax

```
modify_field_with_shift(dst, src, shift, mask)
```

4.3.8.2 Description

Logically, performs the operation `dst = (src >> shift) & mask`.
Actually, performs the operation `dst[bw-1:0] = src[bw+shift-1:shift]`, where **`bw`** is the size of the mask.
For example, `modify_field_with_shift(x_16, y_16, 5, 0xFF)` maps to `x_16[7:0] = y_16[12:5]`, since **`bw`** = 8.

4.3.8.3 Parameters

- *dst*: a metadata field.
- *src*: a packet or metadata field.
- *shift*: an integer indicating the right shift distance.
- *mask*: an integer indicating how many bits to keep after the shift operation.

4.3.8.4 Restrictions

- Dst and src must be the same bit width and less than or equal to 32 bits each.
- Dst and src cannot be the same field.
- There is no way to clear the most significant bits of the destination with this operation, so dst must start at 0.
- Shift must be an integer, such that `0 <= shift <= src's bit width`.
- Mask must be of the form $(2^N) - 1$, where $0 <= N <=$ dst's bit width.

4.3.9 funnel_shift_right

4.3.9.1 P4 Syntax

```
funnel_shift_right(dst, msb_src, lsb_src, shift_distance)
```

4.3.9.2 Description

Performs the operation `dst = {msb_src, lsb_src} >> shift_distance`.

4.3.9.3 Parameters

- *dst*: a packet or metadata field.

- *msb_src*: a packet field or metadata field.
- *lsb_src*: a packet field or metadata field.
- *shift_distance*: an integer.

4.3.9.4 Restrictions

- dst, msb_src, and lsb_src all must be the same bit width.
- The maximum supported bit width is 32.
- The shift distance must be greater than or equal to 0.
- The shift distance must be less than or equal to min(dst.bit_width, 32).

4.4 Header Operations Primitives

4.4.1 add_header

4.4.1.1 P4 Syntax

```
add_header(hdr)
```

4.4.1.2 Description

Marks a header instance as valid. This method does not initialize the fields of the header instance. Ensure that your program initializes all fields in the header to appropriate values. The header will be output when the packet is deparsed at the end of the current gress pipeline.

4.4.1.3 Parameters

- *hdr*: a P4 header instance.

4.4.1.4 Restrictions

- The header instance must be part of the parsed representation.

4.4.2 copy_header

4.4.2.1 P4 Syntax

```
copy_header(hdr_dst, hdr_src)
```

4.4.2.2 Description

Copies all fields of the source header instance to the destination header instance. Sets the destination header instance's validity as valid.

4.4.2.3 Parameters

- *hdr_dst*: the destination P4 header instance.
- hdr_src: the source P4 header instance

4.4.2.4 Restrictions

- The destination and source header instances must be part of the parsed representation.
- The destination and source header instances must be of the same header type.

4.4.3 pop

4.4.3.1 P4 Syntax

```
pop(hdr_stack, cnt)
```

4.4.3.2 Description

Pops the specified number of header instances from a header tag stack by modifying the each header instance's validity bit and copying the fields to the appropriate location in the stack. For example, pop(stack1, 2) if stack1 is 3 elements will look like:

Index position	Before pop executes	After pop executes
0	stack1[0] (outer instance)	stack1[2]
1	stack1[1]	-
2	stack1[2] (innermost instance)	-

4.4.3.3 Parameters

- *hdr_stack*: a packet or metadata field.
- *cnt*: an optional integer specifying how many header instances to pop off the header stack. The default value is 1.

4.4.3.4 Restrictions

- The header stack must be in the parsed representation.
- A pop count of 0 is ignored.
- A pop count greater than the declared dimensions of the header stack is truncated to be the maximum depth of the header stack.

4.4.4 push

4.4.4.1 P4 Syntax

```
push(hdr_stack, cnt)
```

4.4.4.2 Description

Pushes the specified number of header instances onto a header tag stack by modifying each header instance's validity bit and copying the fields to the appropriate location in the stack. For example, push(stack1, 2) if stack1 is 3 elements will look like:

Index position	Before push executes	After push executes
0	stack1[0] (outer instance)	stack1[0] (new outer instance)
1	-	stack1[1] (new outer instance)

2	-	stack1[0] (original outer instance)

4.4.4.3 Parameters

- *hdr_stack*: a packet or metadata field.
- *cnt*: an optional integer specifying how many header instances to push onto the header stack. The default value is 1.

4.4.4.4 Restrictions

- The header stack must be in the parsed representation.
- A push count of 0 is ignored.
- A push count greater than the declared dimensions of the header stack is truncated to be the maximum depth of the header stack.

4.4.5 remove_header

4.4.5.1 P4 Syntax

```
remove_header(hdr)
```

4.4.5.2 Description

Marks a header instance as invalid. The header will not be output when the packet is deparsed at the end of the current pipeline.

4.4.5.3 Parameters

- *hdr*: a P4 header instance.

4.4.5.4 Restrictions

- The header instance must be part of the parsed representation.

4.5 Cloning, Recirculation, and Resubmission Primitives

4.5.1 bypass_egress

4.5.1.1 P4 Syntax

```
bypass_egress()
```

4.5.1.2 Description

Indicates that the packet will bypass the egress processing pipeline. Note that processing continues on the packet until the end of the ingress pipeline is reached. Sets the Tofino intrinsic metadata *ig_intr_md_for_tm.bypass_egress* to a value of one. This primitive will also invalidate any bridged metadata headers that were going to be communicated between the ingress and egress pipelines.

4.5.1.3 Restrictions

- Can only be called in the ingress pipeline.

4.5.2 clone_ingress_pkt_to_egress

4.5.2.1 P4 Syntax

```
clone_ingress_pkt_to_egress(mirror_id, field_list)
```

4.5.2.2 Description

Indicates that upon exiting the ingress pipeline, a clone of the packet as it originally entered the parser should be submitted to the egress pipeline with the values of the metadata in the field list preserved.
In the background, this primitive is equivalent to setting the metadata field *ig_intr_md_for_mb.ingress_mirror_id and* sending a digest to the CLONE_I2E_DIGEST_RCVR receiver. Therefore, it inherits all of the behaviors and restrictions from the generate_digest() primitive.

4.5.2.3 Parameters

- *mirror_id*: an integer representing a receiver ID
- *field_list*: an optional list of metadata fields to be carried over to the cloned packet.

4.5.2.4 Restrictions

- Can only be called from the ingress pipeline.
- A maximum of 8 unique field lists are supported by Tofino. Omitting the field list counts as one header.
- The field list cannot contain constant values.
- The field list cannot contain fields from headers or metadata that are written in the parser (for example, instances referenced in extract() and set_metadata() calls).
- The field list size has an upper bound of 7 bytes. Depending on underlying resource allocation for the field list data, the actual limit may be less.

4.5.3 clone_egress_pkt_to_egress

4.5.3.1 P4 Syntax

```
clone_egress_pkt_to_egress(mirror_id, field_list)
```

or, equivalently,

```
clone_e2e(mirror_id, field_list)
```

4.5.3.2 Description

Indicates that upon deparsing, a clone of the packet as it leaves the egress pipeline should be resubmitted to the egress pipeline with the values of the metadata in the field list preserved.
In the background, this primitive is equivalent to setting the metadata field *eg_intr_md_for_mb.egress_mirror_id and* sending a digest to the CLONE_E2E_DIGEST_RCVR receiver. Therefore, it inherits all of the behaviors and restrictions from the generate_digest() primitive.

4.5.3.3 Parameters

- *mirror_id*: an integer representing a receiver ID
- *field_list*: an optional list of metadata fields to be carried over to the cloned packet.

4.5.3.4 Restrictions

- Can only be called from the egress pipeline.
- Cannot be used in conjunction (on the same packet) with sample_e2e()
- A maximum of 8 unique field lists are supported by Tofino. Omitting the field list counts as one header. This resource is also shared with sample headers referenced by the sample_e2e() primitive.
- The field list cannot contain constant values.
- The field list cannot contain fields from headers or metadata that are written in the parser (for example, instances referenced in extract() and set_metadata() calls).
- The field list size has an upper bound of 7 bytes. Depending on underlying resource allocation for the field list data, the actual limit may be less.

4.5.4 recirculate

4.5.4.1 P4 Syntax

```
recirculate(port)
```

4.5.4.2 Description

Marks the packet to be recirculated through the ingress pipeline to the specified port. Sets the Tofino intrinsic metadata field *ig_intr_md_for_tm.ucast_egress_port* to be the value of the passed in port, appended with the pipe ID. The operation performed is:

```
ig_intr_md_for_tm.ucast_egress_port = (port & mask) | (ingress_port & ~mask)
```

where mask = 0x7f.

4.5.4.3 Parameters

- *port*: an integer or action data parameter.

4.5.4.4 Restrictions

- This primitive can only be called on packets while they are in the ingress pipeline.
- The port cannot come from a packet nor metadata field.

4.5.5 resubmit

4.5.5.1 P4 Syntax

```
resubmit(field_list)
```

4.5.5.2 Description

Marks the packet for resubmission. It will complete the ingress pipeline to generate any necessary metadata values. Then, the *original* packet data will be resubmitted to the ingress parser with values of the fields in *field_list* preserved from the ingress processing on the packet.
In the background, this is equivalent to sending a digest to the RESUBMIT_DIGEST_RCVR receiver, and therefore inherits all of the behaviors and restrictions from the generate_digest() primitive.

4.5.5.3 Parameters

- field_list: an optional list of metadata fields to preserve on the resubmitted packet.

4.5.5.4 Restrictions

- Can only be called from the ingress pipeline.
- The field list cannot contain constant values.
- The field list cannot contain fields from headers or metadata that are written in the parser (for example, instances referenced in extract() and set_metadata() calls).
- The field list size has an upper bound of 7 bytes. Depending on underlying resource allocation for the field list data, the actual limit may be less.
- A maximum of 8 unique field lists are supported by Tofino. Omitting the field list argument counts as one field list.

4.5.6 sample_e2e

4.5.6.1 P4 Syntax

```
sample_e2e(session_id, sample_length, sample_header)
```

4.5.6.2 Description

Indicates that a fragment of the packet should be sampled and appended to Tofino's coalescing buffer, until such a point that the buffer is flushed and a coalesced packet containing many such samples is sent through the egress pipeline. Additionally, the header instance pointed to by sample_header is inserted at the beginning of the sample before it is appended to the coalescing buffer.
The sample data comes from the original version of the packet that entered the egress pipeline, while the sample header is taken after egress pipeline modifications.
Later, when a coalesced packet is flushed from the buffer, it enters the egress parser into the special parser state marked with the `@pragma coalesced_packet_entry` pragma. This graph should return ingress as usual, even though the packet will skip directly to the egress pipeline.
In the background, this primitive is equivalent to sending a digest to the CLONE_E2E_DIGEST_RCVR receiver and setting the metadata fields *eg_intr_md_for_mb.egress_mirror_id* and *eg_intr_md_for_mb.coalesce_length*, and therefore inherits all of the behaviors and restrictions from the generate_digest() primitive.

4.5.6.3 Parameters

- *session_id*: an integer representing a receiver ID
- sample_length: a packet field, metadata field, integer constant, or action data parameter
    - a packet field, metadata field or integer input represents the number of ***bytes*** to sample from the packet
    - an action data parameter input represents the number of ***32-bit words*** to sample from the packet (Tofino cannot shift action data inputs.)
- sample_header: an optional header instance to prepend onto the packet sample. By default, no sample header is present.

4.5.6.4 Restrictions

- Can only be called from the egress pipeline.
- Cannot be used in conjunction (on the same packet) with clone_egress_pkt_to_egress()
- A maximum of 8 unique sample headers are supported by Tofino. Omitting the sample header counts as one header. This resource is also shared with field lists referenced by the clone_egress_pkt_to_egress() primitive.
- The sample header size has an upper bound of 8 bytes.
- The sample_length must be a multiple of 4 if it comes from a packet field, metadata field, or an immediate constant. Internally, the byte length is converted to the number of *full* 32-bit words. (for example, a packet length of 15 would be considered three 32-bit words.)

4.6 Utility and Other Primitives

4.6.1 count

4.6.1.1 P4 Syntax

```
count(stat_ref, index)
```

4.6.1.2 Description

Specifies that the indicated statistics table should perform a count operation at the given index. Note that the statistics table determines whether packet and/or packet bytes are being counted.

4.6.1.3 Parameters

- *stat_ref*: a P4 statistics table reference.
- index: a packet field, metadata field, action data table parameter, or constant that specifies the location in the statistics table to perform the count operation.

4.6.1.4 Restrictions

- A P4 action that uses a count primitive can only refer to a single statistics table.
- A P4 action can only call *count* once.
- Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, “Limitations on the Use of Primitives in Default Actions.”

4.6.2 count_from_hash

4.6.2.1 P4 Syntax

```
count_from_hash(stat_ref, field_list_calc)
```

4.6.2.2 Description

Specifies that the indicated statistics table should perform a count operation at an index computed from a hash calculation. Note that the statistics table determines whether packet and/or packet bytes are being counted.

4.6.2.3 Parameters

- *stat_ref*: a P4 statistics table reference.

- *field_list_calc*: a field list calculation object that specifies a hash algorithm, hash width, and field list over which to compute the hash.

4.6.2.4 Restrictions

- A P4 action that uses a *count_from_hash* primitive can only refer to a single statistics table.
- A P4 action can only call *count_from_hash* once.
- A P4 table cannot have actions that use both count and *count_from_hash*. (These require different hardware addressing schemes.)
- Only one field list calculation is allowed per P4 match table.
- Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, "Limitations on the Use of Primitives in Default Actions."

4.6.3 drop

4.6.3.1 P4 Syntax

```
drop()
```

4.6.3.2 Description

Marks the packet to be dropped once the packet reaches the end of the current processing pipeline. The entirety of the P4 Action that uses *drop()* will be performed. To force an immediate drop after the action completes, you must call both *drop()* and *exit()*.
If the drop call occurs in the ingress pipeline, it sets the Tofino intrinsic metadata *ig_intr_md_for_tm.drop_ctl*. if the drop call occurs in the egress pipeline, it sets the Tofino intrinsic metadata *eg_intr_md_for_oport.drop_ctl.*
The *drop_ctl* field is a 3-bit field consisting of:

- [2] disable mirroring
- [1] disable copy_to_cpu
- [0] disable unicast, multicast, and resubmit.

Specifying drop is the same as setting bit[0] in this *drop_ctl* field.

4.6.4 execute_meter

4.6.4.1 P4 Syntax

```
execute_meter(meter_ref, index, dst, pre_color)
```

4.6.4.2 Description

Specifies that the indicated meter table should perform a metering operation at the given index. The meter table determines the meter type, either standard, LPF, or RED. Note that in the P4_14 specification, this method is called "meter".

4.6.4.3 Parameters

- *meter_ref*: a P4 meter table reference.
- index: a packet field, metadata field, action data table parameter, or constant that specifies the location in the meter table to perform the count operation.

- dst: a packet or metadata field that specifies where to write the meter result. In normal mode, this will be the meter color. In LPF mode, the dst is written with the lpf result.
- pre_color: an optional packet or metadata field that specifies the minimum allowed color result. This can only be used in standard meter mode.

4.6.4.4 Restrictions

- A P4 action that uses an execute meter primitive can only refer to a single meter table.
- A P4 action can only call *execute_meter* once.
- The output result of the meter is 8 bits, which is currently encoded with values 0 for green, 1 for yellow, and 3 for red.
- The *pre_color* field must be at least two bits. If a field wider than two bits is used, only the least significant two bits will be examined. The encoding used is 0 for green, 1 for yellow, and 3 for red.
- If *index* is a packet of metadata field, the meter table must have a number of entries equal to 2^n where n is the bit width of index field. For example, if the index field is 10 bits, the meter table must specify 1024 entries.
- Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, "Limitations on the Use of Primitives in Default Actions."

4.6.5 execute_meter_with_or

4.6.5.1 P4 Syntax

```
execute_meter_with_or(meter_ref, index, dst, pre_color)
```

4.6.5.2 Description

Specifies that the indicated meter table should perform a metering operation at the given index and OR its color result with the destination field.
The operation performed is:
dst = dst | meter_ref[index]

4.6.5.3 Parameters

- *meter_ref*: a P4 meter table reference.
- index: a packet field, metadata field, action data table parameter, or constant that specifies the location in the meter table to perform the count operation.
- dst: a packet or metadata field that specifies where to write the meter result. In normal mode, this will be the meter color. In LPF mode, the dst is written with the lpf result.
- pre_color: an optional packet or metadata field that specifies the minimum allowed color result. This can only be used in standard meter mode.

4.6.5.4 Restrictions

- A P4 action that uses an execute meter primitive can only refer to a single meter table.
- A P4 action can only call *execute_meter* once.

- o The output result of the meter is 8 bits, which is currently encoded with values 0 for green, 1 for yellow, and 3 for red.
- o The *pre_color* field must be at least two bits. If a field wider than two bits is used, only the least significant two bits will be examined. The encoding used is 0 for green, 1 for yellow, and 3 for red.
- o If *index* is a packet of metadata field, the meter table must have a number of entries equal to 2^n where *n* is the bit width of index field. For example, if the index field is 10 bits, the meter table must specify 1024 entries.
- o Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, "Limitations on the Use of Primitives in Default Actions."

4.6.6 execute_meter_from_hash

4.6.6.1 P4 Syntax

```
execute_meter_from_hash(meter_ref, field_list_calc, dst, pre_color)
```

4.6.6.2 Description

Specifies that the indicated meter table should perform a metering operation at an index computed by a hash calculation. Note that the meter table determines the meter type, either standard, LPF, or RED.

4.6.6.3 Parameters

- o *meter_ref*: a P4 meter table reference.
- o *field_list_calc*: a field list calculation object that specifies a hash algorithm, hash width, and field list over which to compute the hash.
- o *dst*: a packet or metadata field that specifies where to write the meter result. In normal mode, this will be the meter color. In LPF mode, the *dst* is written with the lpf result.
- o *pre_color*: an optional packet or metadata field that specifies the minimum allowed color result. This can only be used in standard meter mode.

4.6.6.4 Restrictions

- o A P4 action that uses an execute meter from hash primitive can only refer to a single meter table.
- o A P4 action can only call *execute_meter_from_hash* once.
- o A P4 match table cannot have actions that use both *execute_meter* and *execute_meter_from_hash*. (These require different hardware addressing schemes.)
- o Only one field list calculation is allowed per P4 match table.
- o The output result of the meter is 8 bits, which is currently encoded with values 0 for green, 1 for yellow, and 3 for red.
- o The *pre_color* field must be at least two bits. If a field wider than two bits is used, only the least significant two bits will be examined. The encoding used is 0 for green, 1 for yellow, and 3 for red.
- o Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, "Limitations on the Use of Primitives in Default Actions."

4.6.7 execute_meter_from_hash_with_or

4.6.7.1 P4 Syntax

```
execute_meter_from_hash_with_or(meter_ref, field_list_calc, dst, pre_color)
```

4.6.7.2 Description

Specifies that the indicated meter table should perform a metering operation at an index computed by a hash calculation.

The operation performed is:
dst = dst | meter_ref[hash(field_list_calc)]

4.6.7.3 Parameters

- *meter_ref*: a P4 meter table reference.
- *field_list_calc*: a field list calculation object that specifies a hash algorithm, hash width, and field list over which to compute the hash.
- *dst*: a packet or metadata field that specifies where to write the meter result. In normal mode, this will be the meter color. In LPF mode, the *dst* is written with the lpf result.
- *pre_color*: an optional packet or metadata field that specifies the minimum allowed color result. This can only be used in standard meter mode.

4.6.7.4 Restrictions

- A P4 action that uses an execute meter from hash primitive can only refer to a single meter table.
- A P4 action can only call *execute_meter_from_hash* once.
- A P4 match table cannot have actions that use both *execute_meter* and *execute_meter_from_hash*. (These require different hardware addressing schemes.)
- Only one field list calculation is allowed per P4 match table.
- The output result of the meter is 8 bits, which is currently encoded with values 0 for green, 1 for yellow, and 3 for red.
- The *pre_color* field must be at least two bits. If a field wider than two bits is used, only the least significant two bits will be examined. The encoding used is 0 for green, 1 for yellow, and 3 for red.
- Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, "Limitations on the Use of Primitives in Default Actions."

4.6.8 execute_stateful_alu

4.6.8.1 P4 Syntax

```
execute_stateful_alu(index)
```

4.6.8.2 Description

Specifies that a stateful ALU interface should be run on a stateful table at the given index, if specified.

4.6.8.3 Parameters

- index: an optional packet field, metadata field, action data table parameter, or constant that specifies the location in the stateful table to perform the operation.

4.6.8.4 Restrictions

- A P4 action that uses an execute stateful alu primitive can only refer to a single stateful table.
- A P4 action can only call *execute_stateful_alu* or *execute_stateful_alu_from_hash* once.
- A P4 match table's action's cannot use both *execute_stateful_alu* and *execute_stateful_alu_from_hash*, because these require different addressing schemes.
- Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, "Limitations on the Use of Primitives in Default Actions."

4.6.9 execute_stateful_alu_from_hash

4.6.9.1 P4 Syntax

```
execute_stateful_alu_from_hash(field_list_calc)
```

4.6.9.2 Description

Specifies that a stateful ALU interface should be run on a stateful table with an index computed by a hash calculation.

4.6.9.3 Parameters

- field_list_calc: a field list calculation object that specifies a hash algorithm, hash width, and field list over which to compute the hash.

4.6.9.4 Restrictions

- A P4 action that uses an execute stateful alu primitive can only refer to a single stateful table.
- A P4 action can only call *execute_stateful_alu* or *execute_stateful_alu_from_hash* once.
- A P4 match table's actions cannot use both *execute_stateful_alu* and *execute_stateful_alu_from_hash*, because these require different addressing schemes.
- If the hash width has more addressable space than the stateful table, hash indexes outside the addressable range likely results in writing zero to the destination field, if any is specified.
- Using this primitive in an action can render that action ineligible to be a default action. See the earlier section, "Limitations on the Use of Primitives in Default Actions."

4.6.10 exit

4.6.10.1P4 Syntax

```
exit()
```

4.6.10.2Description

Terminates packet processing in the ***current*** pipeline upon completion of the action from which it is called. Note that if this is called in the ingress pipeline, packet processing will still occur in the egress pipeline (unless the packet has been dropped or if used in conjunction with *bypass_egress*.)

4.6.11 generate_digest

4.6.11.1P4 Syntax

```
generate_digest(digest_id, field_list)
```

4.6.11.2Description

Specifies that data specified by the field list should be sent to the "receiver" specified by digest_id. For each receiver, the compiler reserves a PHV container to indicate which field list to use. When the packet reaches the deparser, the data in the field list is sampled and sent to the hardware function that corresponds to this receiver. Note that the sampling occurs only when the packet reaches the deparser, and not at the time the generate digest call is made in the MAU pipeline.
Each receiver has a variety of different behaviors. Supported digest receivers are enumerated in the constants.p4 file of the Tofino library. Currently, only FLOW_LRN_DIGEST_RCVR is supported.

4.6.11.3Parameters

- *digest_id*: an integer representing the digest receiver. Currently, the generate_digest pragma requires that you set this argument to 0, which specifies that the FLOW_LRN_DIGEST_RCVR digest receiver will be used. See below for details.
- *field_list*: an optional list of fields to be sent to the digest receiver. By default, an empty field list is sent.

4.6.11.4Restrictions

- A maximum of 8 field lists per digest receiver per gress are supported by Tofino. Calls to generate_digest that do not include a field list are also counted against this total.
- Only one digest can be sent to a given receiver per gress. If generate_digest() is called with the same digest_id twice, only the last field list specified is actually used.
- The field list cannot contain constant values
- FLOW_LRN_DIGEST_RCVR is a field list representing a "learning quantum" to be sent to Tofino's learning filter. Observe these FLOW_LRN_DIGEST_RCVR restrictions:
    - The field list size has an upper bound of 47 bytes. Depending on underlying resource allocation for the field list data, the actual limit may be less.

4.6.12 invalidate

4.6.12.1P4 Syntax

```
invalidate(dst)
```

4.6.12.2Description

Invalidates a PHV container by setting the container's validity bit to 0 and clearing the container to all zeros. If the *dst* argument is a packet or metadata field, all PHV containers that contain all or part of the field will be invalidated. Note that invalidate works on a container granularity. If a field is invalidated that does not occupy the entire container, the whole container is still invalidated and cleared.

4.6.12.3Parameters

- *dst*: a packet or metadata field

4.6.13 invalidate_digest

4.6.13.1P4 Syntax

```
invalidate_digest()
```

4.6.13.2Description

Specifies that any learning digest that is active, as indicated by prior *generate_digest* usages, should be invalidated. No learning notifications will be produced, unless a subsequent call to *generate_digest* is made.

4.6.13.3Restrictions

- This primitive can only be called if the generate_digest primitive is used.
- This primitive can only be called in the ingress pipeline.

4.6.14 invalidate_clone

4.6.14.1P4 Syntax

```
invalidate_clone()
```

4.6.14.2Description

Specifies that any cloning digest that is active, as indicated by prior *clone_ingress_to_egress* or *clone_egress_to_egress* usages, should be invalidated. No clone will be produced, unless a subsequent call to a cloning primitive is made.

In the ingress pipeline, this primitive performs the operation:

- ig_intr_md_for_mb.ingress_mirror_id = 0
- compiler added field to specify the mirror field list index = 0

In the egress pipeline, this primitive performs this operation:

- Invalidates the container that holds the compiler added field to specify the mirror field list index.

4.6.14.3 Restrictions

- This primitive can only always be called in the ingress pipeline, since an ingress to egress session is reserved to implement drop functionality.
- This primitive can only be called in the egress pipeline if the *clone_egress_to_egress* primitive is used.

4.6.15 no_op

4.6.15.1 P4 Syntax

```
no_op()
```

4.6.15.2 Description

Performs no operation.

5

Supported Tofino Hash Algorithms

5.1 Using Hash Functions in P4 Programs

The hash algorithms listed in this chapter can be used for the following tasks in your P4 programs that run on Tofino:

- Hash-based addressing (e.g. *execute_stateful_alu_from_hash*)
- Copying hash result to a PHV container (e.g. *modify_field_with_hash_based_offset*)
- Selector hash inputs

Tofino hash functions support a maximum hash output width of 52 bits. Hash output widths larger than that will currently truncate to 52 bits. Tofino does not support using multiple hash functions to accomplish wider hashing.

5.2 Polynomial Hash Algorithms

The names listed in this table can be used in P4 programs to reference the indicated polynomial hash function.

Name (alternate name)	**poly**	**reversed**	**init**	**xout**
crc_8	0x107		0x00	0x00
crc_8_darc	0x139	X	0x00	0x00
crc_8_i_code	0x11D		0xFD	0x00
crc_8_itu	0x107		0x55	0x55
crc_8_maxim	0x131	X	0x00	0x00
crc_8_rohc	0x107	X	0xFF	0x00
crc_8_wcdma	0x19B	X	0x00	0x00
crc_16 (crc16)	0x18005	X	0x0000	0x0000
crc_16_buypass	0x18005		0x0000	0x0000
crc_16_dds_110	0x18005		0x800D	0x0000
crc_16_dect	0x10589		0x0001	0x0001
crc_16_dnp	0x13D65	X	0xFFFF	0xFFFF
crc_16_en_13757	0x13D65		0xFFFF	0xFFFF
crc_16_genibus	0x11021		0x0000	0xFFFF
crc_16_maxim	0x18005	X	0xFFFF	0xFFFF
crc_16_mcrf4xx	0x11021	X	0xFFFF	0x0000
crc_16_riello	0x11021	X	0x554D	0x0000
crc_16_t10_dif	0x18BB7		0x0000	0x0000
crc_16_teledisk	0x1A097		0x0000	0x0000
crc_16_usb	0x18005	X	0x0000	0xFFFF
x_25	0x11021	X	0x0000	0xFFFF
xmodem	0x11021		0x0000	0x0000
modbus	0x18005	X	0xFFFF	0x0000
kermit	0x11021	X	0x0000	0x0000
crc_ccitt	0x11021		0xFFFF	0x0000
crc_aug_ccitt	0x11021		0x1D0F	0x0000
crc_32 (crc32)	0x104C11DB7	X	0x00000000	0xFFFFFFFF
crc_32_bzip2	0x104C11DB7		0x00000000	0xFFFFFFFF
crc_32c	0x11EDC6F41	X	0x00000000	0xFFFFFFFF
crc_32d	0x1A833982B	X	0x00000000	0xFFFFFFFF
crc_32_mpeg	0x104C11DB7		0xFFFFFFFF	0x00000000
posix	0x104C11DB7		0xFFFFFFFF	0xFFFFFFFF
crc_32q	0x1814141AB		0x00000000	0x00000000

jamcrc	0x104C11DB7	X	0xFFFFFFFF	0x00000000
xfer	0x1000000AF		0x00000000	0x00000000
crc_64	0x1000000000000001B	X	0x0000000000000000	0x0000000000000000
crc_64_we	0x142F0E1EBA9EA3693		0x0000000000000000	0xFFFFFFFFFFFFFFFF
crc_64_jones	0x1AD93D23594C935A9	X	0xFFFFFFFFFFFFFFFF	0x0000000000000000

5.3 Named Hash Algorithms

The named hash functions are summarized in the table below and explained in the sections that follow.

Name	Maximum Bit Width	Comment
identity	any	Defaults to use the least significant N bits from the field list
identity_msb	any	Use the most significant N bits from the field list
identity_lsb	any	Use the least significant N bits from the field list
random	any	Random XORing of bits of the field list.

5.3.1 identity

Computes an identity hash by directly translating the input field list to output hash result.
Examples:

```
field_list my_field_list {
    h.w_8;
    h.x_8;
    h.y_8;
    h.z_8;
}
```

If producing 16 bit outputs:

```
identity(my_field_list) = {h.y_8, h.z_8}
identity_lsb(my_field_list) = {h.y_8, h.z_8}
identity_msb(my_field_list) = {h.w_8, h.x_8}
```

If producing 32 bit outputs:

```
identity(my_field_list) = {h.w_8, h.x_8, h.y_8, h.z_8}
```

5.3.2 random

Computes a random hash result by randomly XORing bits of the input field list.

5.4 User-defined Hash Algorithms

User-defined hash polynomials can be specified to produce 8-bit, 16-bit, 32-bit, or 64-bit results. Note that the *modify_field_with_hash_based_offset* primitive can only read at most 32 bits of hash result.
This is done by providing a string of the format:

```
algo_string ::= poly_<hex-number>{_<prop_name>[_<prop_value>]}*
```

where the supported properties (prop_names) are:

- `poly_<hex formatted string>` — the polynomial to use (Note: for the polynomial of the power N, bit N must be set. In other words, the bit width of the string must be at least N+1). **This property is mandatory.**
- `init_<hex formatted string>` — the initial value of the crc shift register. Default value is 0x0
- `not_rev` — indicates to not use a bit reverse algorithm. Default value is to use bit reverse algorithm
- `xout_<hex formatted string>` — a value to XOR with the final CRC computed. Default value is 0x0
- `lsb` —indicates to use the least significant bits of the hash output.
- `msb` — indicates to use the most significant bits of the hash output.
- `extend` — indicates to repeat the hash output width until the desired bit width is achieved.

Note: `lsb`, `msb`, and `extend` properties are mutually exclusive. No more than one of them can be used in a given definition.

Examples:

- The most frequently-used crc16 implementation (CRC16-CCITT) is represented in this format as: `poly_0x11021_not_rev`
- The most frequently-used crc32 implementation (CRC32-IEEE) is represented in this format as: `poly_0x104c11db7_xout_0xffffffff`

Only the polynomials of the powers 8, 16, 32, and 64 (represented by 9, 17, 33 and 65-bit-long numbers) are supported.

5.5 Appendix: Legacy Polynomial Hash Functions

Hash name	Maximum Bit Width	Comment
`crc16`	16	Defaults to use the least significant N bits from the field list, if N < 16
`crc16_msb`	16	Use the most significant N bits from the field list, where N < 16
`crc16_lsb`	16	Use the least significant N bits from the field list
`crc16_extend`	any	Repeats the calculated value every 16 bits until the required number of output bits is filled.
`crc32`	32	Defaults to use the least significant N bits from the field list, if N < 32
`crc32_msb`	32	Use the most significant N bits from the field list, where N < 32
`crc32_lsb`	32	Use the least significant N bits from the field list, where N < 32
`crc32_extend`	any	Repeats the calculated value every 32 bits until the required number of output bits is filled.

5.5.1 crc16

Computes a standard crc16 hash result over the provided input field list.

- Polynomial: 0x18005
- Reversed algorithm.
- Initial value: 0x0

- XOR out value: 0x0

5.5.2 crc32

Computes a standard crc32 hash result over the provided input field list.

- Polynomial: 0x104c11db7
- Reversed algorithm.
- Initial value: 0x0
- XOR out value: 0xffffffff

6

Error Codes

6.1 Parser Error Codes

The following error codes may be returned by the parser:

- PARSER_ERROR_OK (0x0000): No probems.
- PARSER_ERROR_NO_TCAM (0x0001): Unhandled case in select() statement that doesn't have a default clause.
- PARSER_ERROR_PARTIAL_HDR (0x0002): Ran out of data while trying to extract a header (packet too short).
- PARSER_ERROR_CTR_RANGE (0x0004): Should not occur, since we do not support variable length fields in the compiler.
- PARSER_ERROR_TIMEOUT_USER (0x0008): Infinite loop detected (too many states) while parsing a packet. Should not occur, since the compiler prevents it.
- PARSER_ERROR_TIMEOUT_HW (0x0010): Infinite loop detected (tto many clocks) while parsing a packet. Should not occur, since the compiler prevents it.
- PARSER_ERROR_SRC_EXT (0x0020): Invalid extraction source. Often occurs together with PARTIAL_HDR and indicates that a disallowed operation was attempted, such as trying to extract 16 bits of data into a 32-bit container.
- PARSER_ERROR_DST_CONT (0x0040): Should not occur.
- PARSER_ERROR_PHV_OWNER (0x0080): Should not occur.
- PARSER_ERROR_MULTIWRITE (0x0100): Should not occur. Compiler prevents arbitrary multiple writes to the same container unless it's a specific intention (such as POV bits).
- PARSER_ERROR_ARAM_SBE (0x0200): Single bit error detected and corrected in parser Action RAM.
- PARSER_ERROR_ARAM_MBE (0x0400): Multi-bit error detected (and not corrected) in parser Action RAM.
- PARSER_ERROR_FCS (0x0800): Bad FCS. This is a technical bit, since you won't typically see it -- it is sent ahead to make sure packet is properly discarded later.
- PARSER_ERROR_CSUM (0x1000): Bad Checksum.
- PARSER_ERROR_ARRAY_OOB (0xC000): Should not occur due to explicit parser loop unrolling by the compiler.