$P4_{16}$ Portable Switch Architecture (PSA)

(draft)

The P4.org language consortium

May 15, 2017

Abstract

P4 is a language for expressing how packets are processed by the data plane of a programmable network forwarding element. P4 programs specify how the various programmable blocks of a target architecture are programmed and connected. The Portable Switch Architecture (PSA) is target architecture that describes common capabilities of network switch devices which process and forward packets across multiple interface ports.

1. Target Architecture Model

The Portable Switch Architecture (PSA) Model has six programmable P4 blocks and two fixed-function blocks, as shown in Figure 1. Programmable blocks are hardware blocks whose function can be programmed using the P4 language. The Packet buffer and Replication Engine (PRE) and the Buffer Queueing Engine (BQE) are target dependent functional blocks that may be configured for a fixed set of operations.

Incoming packets are parsed and have their checksums validated and are then passed to an ingress match action pipeline, which makes decisions on where the packets should go. After the ingress pipeline, the packet may be buffered and/or replicated (sent to multiple egress ports). For each such egress port, the packet passes through an egress match action pipleline and a checksum update calculation before it is deparsed and queued to leave the pipeline..

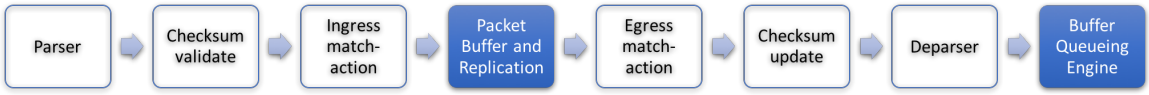

Figure 1. Portable Switch Pipeline

A programmer targetting the PSA is required to instantiate objects for the programmable blocks that conform to these APIs. Note that the programmable block APIs are templetized on user defined headers and metadata. The PSA offers flexibility in defining the user metadata, however, one may chose to use a common metadata types for subsets of the objects.

When instantiating the `main` `package` object, the instances corresponding to the programmable blocks are passed as arguments.

2. PSA Data types

2.1. PSA type definitions

These types need to be defined before including the architecture file and the macro protecting them should be defined.

```
typedef bit<unspecified> PortId_t;
typedef bit<unspecified> MulticastGroup_t;
typedef bit<unspecified> PacketLength_t;
typedef bit<unspecified> EgressInstance_t;
typedef bit<unspecified> ParserStatus_t;
typedef bit<unspecified> ParserErrorLocation_t;
typedef bit<unspecified> entry_key;           /// for DirectCounters

const   PortId_t         PORT_CPU = unspecified;
```

2.2. PSA supported metadata types

```
typedef enum { NORMALINSTANCE_, CLONE_INSTANCE } InstanceType_t;

struct psa_parser_input_metadata_t {
  PortId_t                ingress_port;
  InstanceType_t          instance_type;
}

struct psa_ingress_input_metadata_t {
  PortId_t                ingress_port;
  InstanceType_t          instance_type;  /// Clone or Normal
  /// set by the runtime in the parser, these are not under programmer control
  ParserStatus_t          parser_status;
  ParserErrorLocation_t   parser_error_location;
}

struct psa_ingress_output_metadata_t {
  PortId_t                egress_port;
}

struct psa_egress_input_metadata_t {
  PortId_t                egress_port;
  InstanceType_t          instance_type;  /// Clone or Normal
  EgressInstance_t        instance;       /// instance coming from PRE
}
```

2.3. Match kinds

Additional supported match_kind types

```
match_kind {
    range,   /// Used to represent min..max intervals
    selector /// Used for implementing dynamic_action_selection
}
```

2.4. Cloning methods

```
enum CloneMethod_t {
  /// Clone method         Packet source             Insertion point
  Ingress2Ingress,  /// original ingress,           Ingress parser
  Ingres2Egress,    /// post parse original ingres, Buffering queue
  Egress2Ingress,   /// post deparse in egress,     Ingress parser
  Egress2Egress     /// inout to deparser in egress, Buffering queue
}
```

3. PSA Externs

3.1. Packet Replication Engine

The `PacketReplicationEngine` extern represents the non-programmable part of the PSA pipeline.

Even though the PRE can not be programmed using P4, it can be configured both directly using control plane APIs and by setting intrinsic metadata. In this specification we opt to define

the operations available in the PRE as method invocations. A target backend is responsible for mapping the PRE extern APIs to the appropriate mechanisms for performing these operations in the hardware.

The PRE is instantiated by the architecture and a P4 program can use it directly. It is an error to instantiate the PRE multiple times. The PRE is made available to the Ingress programmable block using the same mechanism as packet_in. A corresponding Buffering and Queueing Engine (BQE) extern is defined for the Egress pipeline (@see BufferingQueueingEngine).

Note: some of these operations may not be implemented as primitive operations on a certain target. However, All of the operations can be implemented as a combination of other operations. Applications that rely on non-primitive operations may incur significant performance penalties, however, they should be functionally correct.

Semantics of behavior for multiple calls to PRE APIs

The semantics of calling the PRE APIs is equivalent to setting intrinsic metadata fields/bits and assumming that the PRE looks up the fields in the following order: drop, truncate, multicast, clone, output_port.

Following this semantics, examples of the behaviors are:

- any call to drop in the pipeline will cause the packet, and all potential clone copies (see below) to drop.
- any call to truncate, will cause the packet (and its clones \todo: check) to be truncated.
- multiple calls to send_to_port – the last call in the ingress pipeline sets the output port.
- multiple calls to multicast – the last in the ingress pipeline sets the multicast group
- interleaving send_to_port and multicast – the semantics of multicast is defined as below (https://github.com/p4lang/tutorials/issues/22): if (multicast_group != 0)

  ```
  multicast_to_group(multicast_group);
  ```

  else

  ```
  send_to_port(output_port);
  ```

  From this, it follows that if there is a call that sets the multicast_group, the packet will be multicast to the group that last set the multicast group. Otherwise, the packet will be sent to the port set by send_to_port.
- multiple clone invocations will cause the packet to be cloned to the corresponding port. Any drop call in the pipeline will cause the packet to drop, and no clone will be created (following the analogy with intrinsic metadata bit fields, drop bits are processed before clone bits are looked up).
- resubmit
- recirculate

\TODO: finalize the semantics of calling multiple of the PRE APIs

```
extern PacketReplicationEngine {

  // PacketReplicationEngine(); /// No constructor. PRE is instantiated
                                /// by the architecture.
```

3.1.1. PRE Methods

3.1.1.1. Unicast operation Sends packet to a port.

Targets may implement this operation by setting the appropriate intrinsic metadata or through some other mechanism of configuring the PRE.

The port parameter is the output port. If the port is PORT_CPU the packet will be sent to CPU.

```
void send_to_port (in PortId_t port);
```

3.1.1.2. Multicast operation Sends packet to a multicast group or a port.

Targets may implement this operation by setting the appropriate intrinsic metadata or through some other mechanism of configuring the PRE.

The multicast_group parameter is the multicast group id. The control plane must program the multicast groups through a separate mechanism.

```
void multicast (in MulticastGroup_t multicast_group);
```

3.1.1.3. Drop operation Do not forward the packet.

The PSA implements drop as an operation in the PRE. While the drop operation can be invoked anywhere in the ingress pipeline, the semantics supported by the PSA is that the drop will be at the end of the pipeline (ingress or egress).

```
void drop      ();
```

3.1.1.4. Clone operation Create a copy of the packet and send it to the specified port.

The PSA specifies four types of cloning, with the packet sourced from different points in the pipeline and sent back to ingress or to the buffering queue in the egress (@see CloneMethod_t). \TODO: needs both source and destination data

Parameters:

- clone_method The type of cloning.
- port The port to send the cloned packet to.

```
void clone     (in CloneMethod_t clone_method, in PortId_t port);
```

3.1.1.5. Clone with extra data operation Create a copy of the packet with additional data and send it to the specified port.

The PSA specifies four types of cloning, with the packet sourced from different points in the pipeline and sent back to ingress or to the buffering queue in the egress (@see CloneMethod_t). \TODO: needs both source and destination data

Parameters:

- clone_method The type of cloning.
- port The port to send the cloned packet to.
- data additional header data attached to the packet

```
void clone<T>  (in CloneMethod_t clone_method, in PortId_t port, in T data);
```

3.1.1.6. Resubmit operation Send a packet to the ingress port with additional data appended.

This operation is intended for recursive packet processing. \TODO: needs both source and destination data

Parameters:

- data A header definition that can be added to the set of packet headers.
- port The input port at which the packet will be resubmitted.

```
void resubmit<T>(in T data, in PortId_t port);
```

3.1.1.7. Recirculate operation Send a post deparse packet to the ingress port with additional data appended.

This operation is intended for recursive packet processing. \TODO: needs both source and destination data

Parameters:

- data A header definition that can be added to the set of packet headers.
- port The input port at which the packet will be resubmitted.

```
void recirculate<T>(in T data, in PortId_t port);
```

3.1.1.8. Truncate operation Truncate the outgoing packet to the specified length.

The length parameter represents the packet length.

```
void truncate(in bit<32> length);
```

3.2. Buffering Queueing Engine

The BufferingQueueingEngine extern represents the the other non-programmable part of the PSA pipeline (after Egress).

Even though the BQE can not be programmed using P4, it can be configured both directly using control plane APIs and by setting intrinsic metadata. In this specification we opt to define the operations available in the BQE as method invocations. A target backend is responsible for mapping the BQE extern APIs to the appropriate mechanisms for performing these operations in the hardware.

The BQE is instantiated by the architecture and a P4 program can use it directly. It is an error to instantiate the BQE multiple times. The BQE is made available to the Egress programmable block using the same mechanism as packet_in. A corresponding Packet Replication Engine (PRE) extern is defined for the Igress pipeline (@see PacketReplicationEngine).

Note: some of these operations may not be implemented as primitive operations on a certain target. However, All of the operations can be implemented as a combination of other operations. Applications that rely on non-primitive operations may incur significant performance penalties, however, they should be functionally correct.

The ordering semantics of multiple calls to BQE APIs is identical to the semantics ordering of PRE invocations, for the subset of functions supported in the BQE.

```
extern BufferingQueueingEngine {

  // BufferingQueueingEngine(); /// No constructor. BQE is instantiated
                                /// by the architecture.
```

3.2.1. BQE Methods

3.2.1.1. Unicast operation Sends packet to port.

Targets may implement this operation by setting the appropriate intrinsic metadata or through some other mechanism of configuring the BQE.

The port parameter is the output port. If the port is PORT_CPU the packet will be sent to CPU.

```
void send_to_port (in PortId_t port);
```

3.2.1.2. Drop operation

Do not forward the packet.

The PSA implements drop as an operation in the BQE. While the drop operation can be invoked anywhere in the ingress pipeline, the semantics supported by the PSA is that the drop will be at the end of the pipeline (ingress or egress).

```
void drop       ();
```

3.2.1.3. Truncate operation

Truncate the outgoing packet to the specified length

The length parameter represents the packet length.

```
void truncate(in bit<32> length);
```

3.3. Hashes

Supported hash algorithms:

```
enum HashAlgorithm {
  crc32,
  crc32_custom,
  crc16,
  crc16_custom,
  random,           /// are random hash algorithms useful?
  identity
}
```

3.3.1. Hash function

Example usage:

```
parser P() {
  Hash<crc16, bit<56>> h();
  bit<56> hash_value = h.getHash(16, buffer, 100);
}
```

Parameters:

- Algo The algorithm to use for computation (@see HashAlgorithm).
- O The type of the return value of the hash.

```
extern Hash<Algo, O> {
  /// Constructor
  Hash();

  /// compute the hash for data
  O getHash<T, D, M>(in T base, in D data, in M max);
}
```

3.4. Checksum computation

Checksums and hash value generators are examples of functions that operate on a stream of bytes from a packet to produce an integer. The integer may be used, for example, as an integrity check for a packet or as a means to generate a pseudo-random value in a given range on a packet-by-packet or flow-by-flow basis.

Parameters:

- W The width of the checksum

```
extern Checksum<W> {
  Checksum(HashAlgorithm hash);          /// constructor
  void clear();              /// prepare unit for computation
  void update<T>(in T data); /// add data to checksum
  void remove<T>(in T data); /// remove data from existing checksum
  W    get();               /// get the checksum for data added since last clear
}
```

3.5. Counters

Counters are a simple mechanism for keeping statistics about the packets that trigger a table in a Match Action unit.

Direct counters fire when the count method is ivoked in an action, and have an instance for each entry in the table.

3.5.1. Counter types

```
enum CounterType_t {
    packets,
    bytes,
    packets_and_bytes
}
```

3.5.2. Counter

```
extern Counter<W, S> {
  Counter(S n_counters, W size_in_bits, CounterType_t counter_type);
  void count(in S index, in W increment);

  /*
  @ControlPlaneAPI
  {
    W    read<W>      (in S index);
    W    sync_read<W> (in S index);
    void set          (in S index, in W seed);
    void reset        (in S index);
    void start        (in S index);
    void stop         (in S index);
  }
  */
}
```

3.5.3. Direct Counter

```
extern DirectCounter<W> {
  DirectCounter(CounterType_t counter_type);
  void count();

  /*
  @ControlPlaneAPI
  {
    W    read<W>      (in entry_key key);
```

```
    W    sync_read<W> (in entry_key key);
    void set          (in W seed);
    void reset        (in entry_key key);
    void start        (in entry_key key);
    void stop         (in entry_key key);
  }
  */
}
```

3.6. Meters

Meters (RFC 2698) are a more complex mechanism for keeping statistics about the packets that trigger a table. The meters specified in the PSA are 3-color meters.

3.6.1. Meter types

```
enum MeterType_t {
    packets,
    bytes
}
```

3.6.2. Meter colors

```
enum MeterColor_t { RED, GREEN, YELLOW };
```

3.6.3. Meter

```
extern Meter<S> {
  Meter(S n_meters, MeterType_t type);
  MeterColor_t execute(in S index, in MeterColor_t color);

  /*
  @ControlPlaneAPI
  {
    reset(in MeterColor_t color);
    setParams(in S committedRate, in S committedBurstSize
              in S peakRate, in S peakBurstSize);
    getParams(out S committedRate, out S committedBurstSize
              out S peakRate, out S peakBurstSize);
  }
  */
}
```

3.6.4. Direct Meter

```
extern DirectMeter {
  DirectMeter(MeterType_t type);
  MeterColor_t execute(in MeterColor_t color);

  /*
  @ControlPlaneAPI
  {
```

```
    reset(in entry_key entry, in MeterColor_t color);
    void setParams<S>(in entry_key entry,
                      in S committedRate, in S committedBurstSize
                      in S peakRate, in S peakBurstSize);
    void getParams<S>(in entry_key entry,
                      out S committedRate, out S committedBurstSize
                      out S peakRate, out S peakBurstSize);
  }
  */
}
```

3.7. Registers

Registers are stateful memories whose values can be read and written in actions. Registers are similar to counters, but can be used in a more general way to keep state.

Although registers cannot be used directly in matching, register.read may be used as the RHS of an assignment operation, allowing the current value of the register to be copied into metadata and be available for matching in subsequent tables.

A simple usage example might be to verify that a "first packet" was seen for a particular type of flow. A register cell would be allocated to the flow, initialized to "clear". When the protocol signalled a "first packet", the table would match on this value and update the flow's cell to "marked". Subsequent packets in the flow could would be mapped to the same cell; the current cell value would be stored in metadata for the packet and a subsequent table could check that the flow was marked as active.

```
extern Register<T, S> {
  Register(S size);
  T    read  (in S index);
  void write (in S index, in T value);

  /*
  @ControlPlaneAPI
  {
    T    read<T>      (in S index);
    void set          (in S index, in T seed);
    void reset        (in S index);
  }
  */
}
```

3.8. Random

The random extern provides a reliable, target specific number generator in the min .. max range.

The set of distributions supported by the Random extern. \TODO: should this be removed in favor of letting the extern return whatever distribution is supported by the target?

```
enum RandomDistribution {
  PRNG,
  Binomial,
  Poisson
}

extern Random<T> {
  Random(RandomDistribution dist, T min, T max);
```

```
  T read();

  /*
  @ControlPlaneAPI
  {
    void reset();
    void setSeed(in T seed);
  }
  */
}
```

3.9. Action Profile

Action profiles are used as table implementation attributes.

Action profiles implement a mechanism to populate table entries with actions and action data. The only data plane operation required is to instantiate this extern. When the control plane adds entries (members) into the extern, they are essentially populating the corresponding table entries.

```
extern ActionProfile {
  /// Construct an action profile of 'size' entries
  ActionProfile(bit<32> size);

  /*
  @ControlPlaneAPI
  {
     entry_handle add_member    (action_ref, action_data);
     void         delete_member (entry_handle);
     entry_handle modify_member (entry_handle, action_ref, action_data);
  }
  */
}
```

3.10. Action Selector

Action selectors are used as table implementation attributes.

Action selectors implement another mechanism to populate table entries with actions and action data. They are similar to action profiles, with additional support to define groups of entries. Action selectors require a hash algorithm to select members in a group. The only data plane operation required is to instantiate this extern. When the control plane adds entries (members) into the extern, they are essentially populating the corresponding table entries.

```
extern ActionSelector {
  /// Construct an action selector of 'size' entries
  /// @param algo hash algorithm to select a member in a group
  /// @param size number of entries in the action selector
  /// @param outputWidth size of the key
  ActionSelector(HashAlgorithm algo, bit<32> size, bit<32> outputWidth);

  /*
  @ControlPlaneAPI
  {
     entry_handle add_member        (action_ref, action_data);
     void         delete_member     (entry_handle);
     entry_handle modify_member     (entry_handle, action_ref, action_data);
```

```
    group_handle create_group      ();
    void         delete_group      (group_handle);
    void         add_to_group      (group_handle, entry_handle);
    void         delete_from_group (group_handle, entry_handle);
  }
  */
}
```

3.11. Packet Generation

\TODO: is generating a new packet and sending it to the stream or is it adding a header to the current packet and sending it to the stream (copying or redirecting).

```
extern Digest<T> {
  Digest(PortId_t receiver); /// define a digest stream to receiver
  void emit(in T data);      /// emit data into the stream

  /*
  @ControlPlaneAPI
  {
  // TBD
  // If the type T is a named struct, the name should be used
  // to generate the control-plane API.
  }
  */
}
```

4. Programmable blocks

The following declarations provide a template for the programmable blocks in the PSA. The P4 programmer is responsible for implementing controls that match these interfaces and instantiate them in a package definition.

The current implementation uses the same user-defined metadata structure for all the controls. An alternative design is to split the user-defined metadata into an input parameter and an output parameter for each block. The compiler will have to check that the out parameter of a block matches the in parameter of the subsequent block.

```
parser Parser<H, M>(packet_in buffer, out H parsed_hdr, inout M user_meta,
                    in psa_parser_input_metadata_t istd);

control VerifyChecksum<H, M>(in H hdr, inout M user_meta);

control Ingress<H, M>(inout H hdr, inout M user_meta,
                      PacketReplicationEngine pre,
                      in  psa_ingress_input_metadata_t  istd,
                      out psa_ingress_output_metadata_t ostd);

control Egress<H, M>(inout H hdr, inout M user_meta,
                     BufferingQueueingEngine bqe,
                     in  psa_egress_input_metadata_t  istd);

control ComputeChecksum<H, M>(inout H hdr, inout M user_meta);
```

```
control Deparser<H>(packet_out buffer, in H hdr);

package PSA_Switch<H, M>(Parser<H, M> p,
                         VerifyChecksum<H, M> vr,
                         Ingress<H, M> ig,
                         Egress<H, M> eg,
                         ComputeChecksum<H, M> ck,
                         Deparser<H> dep);
```