BIR (B Intermediate Representation)

Technical Report v1.1

Protocol Independent Forwarding (PIF) working group and open source software project

BIR is under development. This overview of BIR is provided for review by relevant stakeholders, including high-level packet processing language designers and compiler writers; target architecture specialists; and runtime configuration and operation API designers and implementers.

Editor: Gordon Brebner

Date: 9 March, 2016

Introduction

BIR (B Intermediate Representation, pronounced “Beer”) is a proposed intermediate representation for high-level packet processing programs being compiled to target packet processing architectures. It is a product of the Open Source SDN organization’s Protocol Independent Forwarding (PIF) project, which also operates as a working group within the Specifications area of the Open Networking Foundation.

The two main motivations for BIR are: (a) to avoid a proliferation of custom compilers for specific high-level language and target architecture combinations; and (b) to offer a neutral packet processing model that is not tied closely to specific languages and/or architectures but does not impede the efficient compilation of programs to targets.

BIR is the product of lengthy technical discussions and experiments in the PIF community. It is a successor to AIR (A Intermediate Representation, the initial PIF strawman proposal, pronounced “Air”) and was particularly influenced by existing ongoing research on intermediate representations for packet processing including NetASM (Princeton), OF Primitives (Google), POF (Huawei), and PX (Xilinx), by the emergent P4 high-level language, and by a broad spectrum of target architectures.

BIR represents programs using a three-layer packet processing model, as illustrated in Figure 1:

- Data Flow level (top level): a linear processing pipeline through which packets and associated metadata pass. Each pipeline stage is termed a (packet) processor. Some processors may have a programmatic description as a BIR Control Flow level component. Other processors may not have an explicit BIR description, and only present BIR packet and metadata input/output interfaces.

- Control Flow level (middle level): a state machine that operates on a packet and its associated metadata. Each state has a programmatic description as a BIR Basic Block level component that performs an unconditional sequence of instructions. Transitions between states are described using conditional jumps that are taken at the end of the Basic Block execution.

- Basic Block level (bottom level): an unconditional sequence of BIR Instructions that is executed within a particular context during the processing of a packet and its associated metadata. This local context optionally has two components: a header format at a particular offset within the packet; and a table providing lookup over entries maintained by the control and/or data plane.

The Control Flow and Basic Block levels (and the use of that terminology) are familiar from traditional general-purpose intermediate representations used by compilers, such as LLVM. The Data Flow level, the use of header and table contexts in the Basic Block level, and the provided Instruction repertoire, are familiar from domain-specific models of packet processing, such as the OpenFlow datapath.

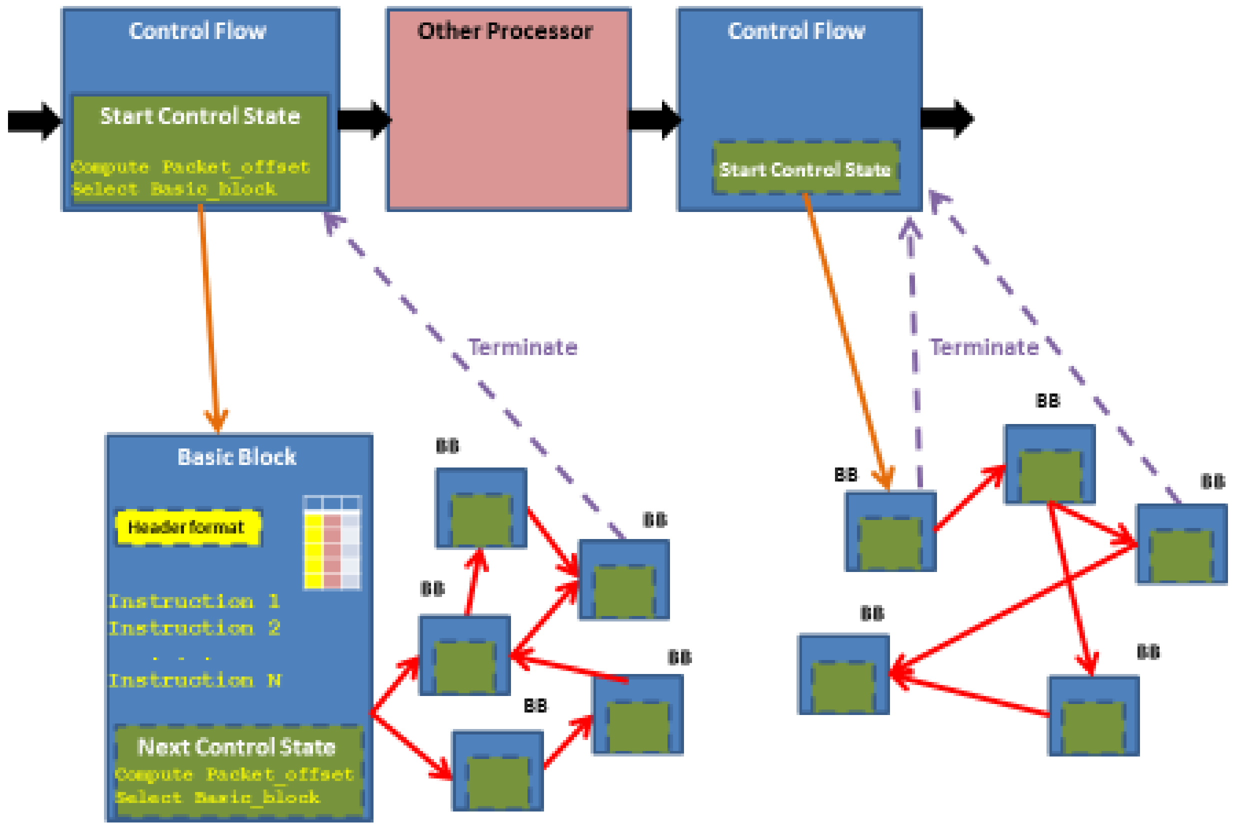

Figure 1: Example instance of BIR packet processing model

A summary of the various object types used in BIR-described systems, as introduced in the following sections, is given in Appendix 1. The only other construct used in BIR are **struct** types, used for defining structured data formats.

A summary of built-in library methods available via BIR M-type Instructions is given in Appendix 2.

An example BIR-described system, represented in YAML, is given in Appendix 3. This illustrates how the various BIR object types are used, and how their attributes are expressed. This YAML representation of BIR structs and object instances is used by the PIF IR modelling and simulation framework.

Appendix 4 lists some BIR topics that are under continuing discussion and development.

Data Flow level description

A Data Flow, which constitutes a described packet processing system instance, is represented by a BIR **processor_layout** type object. This object has one attribute: an ordered list of **processor** type objects. The ordered list represents the linear packet processing pipeline from start to finish.

A BIR **processor** type object is either a **control_flow** type object or an **other_processor** type object. The former is described in the next section. The latter type object is used to include a packet processor that is not explicitly described as a BIR Control Flow, which allows BIR system instances to include other standard library or user-provided packet processing components. An **other_processor** object has one attribute: the name of a processor class. This name is used by a compiler to access information about the Other Processor, for example the associated metadata that it touches, by searching a standard or user-provided directory tree. These Other Processors may, in general, store, clone, and reorder, packets, which are not capabilities of the BIR Control Flow processors as currently defined.

The example instance in Figure 1 shows a three-stage pipeline containing a Control Flow processor, followed by an Other Processor, followed by a Control Flow processor.

The pipeline may, in general, contain multiple packets and associated metadata being processed simultaneously, both across stages and within stages. Particular target architectures may place restrictions on the extent to which such parallel packet processing is allowed.

The familiar use case for a Data Flow is a packet forwarding datapath, where packets are classified and/or modified under the influence of tables that are read or written by a runtime control interface.

A matter for further study is whether a more general directed graph might be used rather than just a simple ordered list. For example, this could be a way of extending BIR's expressive capability towards describing conditional packet switching, or for allowing packet recirculation.

Control Flow level description

A Control Flow is represented by a BIR **control_flow** type object. This object has one attribute: a **control_state** type object. This object is used to describe the starting state for the Control Flow. The **control_state** type object is described in a later section.

The Control Flow behaves as a finite state machine, where the states are Basic Blocks with local contexts. In each state, the Basic Block is executed in order to perform unconditional packet processing. The initial state is determined using the Control Flow's **control_state** type object. Then state transitions occur as a Basic Block associated with the current state processes its own **control_state** type object to determine the Basic Block associated with the next state, as discussed in the next section. The final state is determined when a Basic Block processes its own **control_state** type object, and gets a termination result rather than a next Basic Block result. At this point, the Control Flow has completed its execution. As the state transitions occur, the offset into the packet is updated using the **control_state** type objects.

There are two familiar use cases for a Control Flow. The first is a packet parser or editor, where (some of) the states correspond to successive packet headers being parsed and/or edited. The second is a match-action table flow, where (some of) the states correspond to successive tables used.

A matter for further study is whether a Control Flow may have an optional action for cloning packets at either the entry to, or exit from, the Control Flow processor. At present, this action would have to be carried out in an Other Processor, and hence not be visible in the BIR description.

Basic Block Level description

A Basic Block is represented by a BIR **basic_block** type object. This object has four attributes. The first attribute is an optional **struct** type, which specifies the format of the local header in the packet being processed. The second attribute is an optional **table** type object, which specifies the local table. The third attribute is an ordered list of **instruction** objects, which represents the sequence of instructions executed. The fourth attribute is a **control_state** type object, which is used to describe the next state (or termination) in the enclosing Control Flow.

The Basic Block is where packet processing work is actually done. This work can be data plane driven, in that it operates on a localized header of the packet, and/or control plane driven, in that it has access to a single table with a control plane interface. A local header begins at the offset into the packet that is provided on entry to the Basic Block as part of the Control Flow state, and it has the header format that is associated with the Basic Block. Overall, instructions in the Basic Block can operate on the local header of the packet, all metadata associated with the packet, and the local table. Certain instructions may also access other state information maintained in the packet processing system, not described explicitly in the BIR representation. Header formats and Metadata, Tables, and Instructions, are described in later sections. Both local header format and local table are optional, their inclusion or not depending on the desired function of the Basic Block.

After the unconditional sequence of instructions has been executed, the next state of the enclosing Control Flow is selected. The **control_state** object is used to determine the next state for the Control Flow. If backward movement in the packet is required (offset reduction), this must be done by moving to a new Control Flow.

Packets, Header formats, and Metadata

Packets are the fundamental units operated upon by BIR packet processing system instances. They do not feature in the static BIR representation of a system, but their existence underpins all of the components of the BIR representation. Packets contain BIR-defined headers, which are fixed-size sections of the packet (not necessarily just at the beginning of the packet, despite the name). Packets are accompanied by additional BIR-defined metadata as they are processed by the system.

A Header format is represented by a BIR **struct** type. This describes an ordered list of the fields in a header. Header formats are used in Basic Blocks to interpret the raw bit string that constitutes a physical packet. This bit string begins at the bit offset in the control state for the Basic Block.

Metadata collections are used to hold per-packet state during processing. Each Metadata collection associated with a packet is represented by a BIR **metadata** type object. This object has three attributes: a **struct** type, which specifies the content of the collection of metadata, a visibility type string indicating whether it is supplied at the input and/or output of the packet processing system, and a set of unsigned integer initial values for the collection. The third attribute is included if and only if the collection is not supplied at the input. An instance of each defined **metadata** type object is associated with each packet that is being processed at any time.

Tables

Tables are the means by which packet processing is influenced by input from a control plane. A control interface can be used to add, modify, or delete, entries in a Table. Table lookups are performed during packet processing by Basic Blocks.

A Table is represented by a BIR **table** type object. This object has five attributes: a table type string indicating the style of lookup (for example, indexed, binary CAM, longest-prefix match, or TCAM), a positive integer depth giving the maximum number of entries, a **struct** type defining the lookup request format, a **struct** type defining the lookup response format, and a list of additional operations supported by the Table besides lookup (add entry, modify entry, delete entry, for example) from the data plane through Instructions in Basic Blocks. The request and response formats have no implicit semantics beyond that fact that a list of one or more values is supplied as a lookup request to the Table, and a list of one or more values is returned as a response from the Table.

Control States

Control States are the means for determining states within a Control Flow. The state captures the current offset within the packet being processed, and the current Basic Block processing the packet.

A Control State is represented by a BIR **control_state** type object. This object has two attributes: a selection construct for determining a packet offset value, and a selection construct for determining a **basic_block** type object. A selection construct consists of a list of $n \geq 1$ choices, the first n-1 being [conditional test, candidate] pairs and the final one being an unconditional candidate. For the first attribute, the candidates are expressions for computing a non-negative integer packet offset value. These expressions may include the special \$offset\$ token meaning the current packet offset value. For the second attribute, the candidates are identifiers of Basic Blocks, or the special \$done\$ token meaning termination of the control flow rather than a Basic Block. All selection conditions, and expressions for offset values, can use operands from the metadata associated with the packet, as well as constant values.

Instructions

Instructions are the means by which Basic Blocks perform packet processing. There are three instruction types: V (single Value assignment), O (built-in Operation), and M (user-provided Method call). An Instruction has three arguments, the exact nature of the arguments depending on the specific instruction type. The first argument is always the instruction type: V, O, or M.

A V-type Instruction has two further arguments. The first is the name of a header field within the local header for the Basic Block, or the name of a metadata field. This is the destination of the single value assignment. The second is an expression for computing the assigned value, with operands drawn from local header fields, metadata fields, or constants. Expressions are described in a later section.

An O-type Instruction has two further arguments. The first is the name of a built-in operation to be carried out. The second is a list of operands for the operation, the details depending on the operation. There are five built-in operations:

- Header insert, Hinsert([length]): insert a header at the current packet offset
- Header remove, Hremove([length]): remove a header at the current packet offset
- Table lookup, Tlookup(response, request): lookup request in the local Table
- Table insert, Tinsert(response, request, [,mask]): add an entry to the local Table
- Table remove, Tremove(request [,mask]): remove an entry from the local Table

More detail on these built-in operations is given in Appendix 2.

An M-type Instruction has two further arguments. The first is the name of a user-provided method to be called, which is looked up dynamically at compile time in a user-provided library collection. The second is a pair of metadata collections, the first being written with results from the method call, the second being read for parameters to the method call. An Other Module object is used to represent a collection of user-provided library methods. The **other_module** object has one attribute: a list of names of callable methods. The object's name and the method names are used by a compiler to access information about the methods' interpretation of the result and parameter metadata collections, by searching a standard or user-provided directory tree.

Struct

Structs are used to describe groups of values. A **struct** type consists of a list of fields. Each field has two components: an identifier string and a positive integer width in bits. In general, the list is regarded as unordered but, when a Struct describes a header format, the declared order is significant.

Identifiers

Identifiers are used for **struct** types, and for all object instances. An identifier starts with a letter A to Z or a to z or an underscore (_), followed by zero or more letters, underscores and digits (0 to 9). There are no reserved identifiers in BIR.

Expressions

Expressions are used in various contexts, as mentioned in preceding sections: for selecting and computing initial and next packet offset values, for selecting an initial or next Basic Block, and for computing values assigned by V-type instructions. BIR expressions are atomic, in the sense that the BIR representation does not explicitly decompose them further into individual operations to be carried out in some order. This matter is left to back-end compilation for particular targets. The operators usable within BIR expressions are the standard non-assignment arithmetic, bitwise, and logical, operators found in C. Parentheses may be used in expressions to override standard C operator precedence.

The operands depend on the context of the expression. These may be metadata fields or constants in all contexts. These may be the special operand \$offset\$ when computing next packet offset values, meaning the current offset, or the special operand \$done\$ when selecting an initial or next Basic Block, meaning the termination of the control flow rather than moving to a Basic Block. These may be packet header fields in Basic Blocks that have a local packet header specified.

Annotations

The danger of a common intermediate representation is that useful information in a higher-level language description may be lost and/or obfuscated from back-end compilation to target architectures. To avoid this, and to assist with debugging, BIR representations can contain annotations, falling into three broad categories: (i) source description information, such as filenames, line numbers, and column numbers; (ii) source structure information, such as hierarchical and modular naming and scoping; and (iii) function information, such as description of the packet processing capabilities of particular modules. The first two categories will have standard formats. The third category will have an evolving list of known capabilities, one example being packet parsing. The exact representation of annotations will be defined precisely in a future version of this specification.

Runtime APIs

This specification is concerned with the BIR representation of packet processing systems. An accompanying specification will concern the use of this representation when operating such systems. This requires a BIR-level API that can sit between a particular high-level language runtime API, and a particular target architecture runtime API. Notably, this must contain primitives for accessing Tables within the BIR representation, to supply the control interface to the packet processing. Runtime access of any other BIR components is a matter for future study.

APPENDIX 1: Summary of BIR object types

The following object types are used to form BIR representations of packet processing systems:

Name	*Description*	*Attributes*
processor_layout	Data flow pipeline	Ordered list of processors
processor	Pipeline processor	
control_flow	Control flow processor	Starting state (control_state)
other_processor	Non control flow processor	Name of processor class

basic_block	Basic Block	Local header format (optional);
		Local table (optional);
		Ordered list of instructions;
		Next state (control_state)

metadata	Metadata collection	Metadata collection format;
		External visibility;
		Initial values

table	Table	Table type;
		Table depth;
		Request format;
		Response format;
		Additional data plane operations

other_module	Collection of M-type methods	List of methods
control_state	Offset and Basic Block state	Packet offset calculation; Basic Block selection

instruction	Basic Block instruction	Instruction type: V, O, or M;
		Argument 1;
		Argument 2

In the YAML representation of BIR-described systems, as exemplified in Appendix 3, objects of all of the above types are separately declared with identifiers, except for the last two in the table: **control_state** and **instruction**. Objects of these two types are directly declared without identifiers as attributes of **control_flow** and **basic_block**, and **basic_block**, objects respectively.

APPENDIX 2: Summary of BIR built-in methods

The following built-in methods can be called from M-type instructions in Basic Blocks:

Name	*Description*	*Parameters*
hInsert	Insert header at current offset, with the format of the local header declared for the enclosing Basic Block. All references to the local header in the Basic Block refer to the new inserted header. If the overriding size parameter is given, that number of bits is inserted, rather than the total number of bits of the header format, with padding with zero bits where the insertee size is larger. *At most one hInsert or hRemove call instruction can occur in a Basic Block.*	Expression for an overriding size in bits (optional)
hRemove	Remove header at current offset, with the format of the local header declared for the enclosing Basic Block. All references to the local header in the Basic Block refer to the packet header before its removal. If the overriding size parameter is given, that number of bits is removed, rather than the total number of bits of the header format. *At most one hInsert or hRemove call instruction can occur in a Basic Block.*	Expression for an overriding size in bits (optional)
tLookup	Lookup the local Table. The request metadata collection contains a value for each member of the Table's request struct type. The values for each member of lookup's result struct type are placed into the response metadata collection.	Response metadata; Request metadata

tInsert	Insert entry into the local Table for the enclosing Basic Block. The request metadata collection contains a value for each member of the Table's request struct type. The mask metadata collection is given for table types such as TCAM, and contains a mask value for each member of the Table's request struct type. The response metadata collection contains a value for each member of the Table's response struct type.	Response metadata;
		Request metadata;
		Mask metadata (if required)

tRemove	Remove entry from the local Table for the enclosing Basic Block. The request metadata collection contains a value for each member of the Table's request struct type. The mask metadata collection is given for table types such as TCAM, and contains a mask value for each member of the Table's request struct type.	Request metadata; Mask metadata (if required)

APPENDIX 3: Example YAML representation of a P4-style packet processing system

```
# ----- ----- ----- ----- ----- ----- ----- -----
# Structs
# ----- ----- ----- ----- ----- ----- ----- -----
ethernet_t:
    type: struct
    fields:
        - dst:      48
        - src:      48
        - type_:    16

ipv4_t:
    type: struct
    fields:
        - version:  4
        - ihl:      4
        - tos:      8
        - len:      16
        - id:       16
        - flags:    3
        - frag:     13
        - ttl:      8
        - proto:    8
        - chksum:   16
        - src:      32
        - dst:      32

udp_t:
    type: struct
    fields:
        - sport:    16
        - dport:    16
        - len:      16
        - chksum:   16

metadata_t:
    type: struct
    fields:
        - eth_type_:    16
        - ipv4_proto:   8
        - udp_dport:    16
        - udp_chksum:   16


table_0_req_t:
    type: struct
    fields:
        - dport:        16
```

```
table_0_resp_t:
    type : struct
    fields:
        - hit:              1
        - p4_action:        2
        - action_0_arg0:    16
        - action_1_arg0:    16


# ----- ----- ----- ----- ----- ----- ----- -----
# Metadata Instances
# ----- ----- ----- ----- ----- ----- ----- -----
meta:
    type:       metadata
    values:     metadata_t
    visibility: none

table_0_req:
    type:       metadata
    values:     table_0_req_t
    visibility: none

table_0_resp:
    type:       metadata
    values:     table_0_resp_t
    visibility: none


# ----- ----- ----- ----- ----- ----- ----- -----
# Tables
# ----- ----- ----- ----- ----- ----- ----- -----
table_0:
    type:       table
    match_type: exact
    depth:      64
    request:    table_0_req_t
    response:   table_0_resp_t
    operations: []


# ----- ----- ----- ----- ----- ----- ----- -----
# Parser
# ----- ----- ----- ----- ----- ----- ----- -----
parse_eth:
    type:           basic_block
    local_header:   ethernet_t
    instructions:
        - [V, meta.eth_type_, type_]
    next_control_state:
        - $offset$ + 112
        - [type_ == 0x0800, parse_ipv4]
        - $done$
```

```
parse_ipv4:
    type:           basic_block
    local_header:   ipv4_t
    instructions:
        - [V, meta.ipv4_proto, proto]
    next_control_state:
        - $offset$ + 160
        - [proto == 17, parse_udp]
        - $done$

parse_udp:
    type:           basic_block
    local_header:   udp_t
    instructions:
        - [V, meta.udp_dport, dport]
        - [V, meta.udp_chksum, chksum]
        - [V, table_0_req.dport, dport]
    next_control_state:
        - $offset$ + 64
        - $done$

parser:
    type: control_flow
    start_control_state:
        - 0
        - parse_eth




# ----- ----- ----- ----- ----- ----- ----- -----
# Match+Action Pipeline
# ----- ----- ----- ----- ----- ----- ----- -----
bb_table_udp:
    type:           basic_block
    local_table:    table_0
    instructions:
        - [S, table_0_resp, table_0_req]
    next_control_state:
        - $offset$
        - [table_0_resp.hit == 1 && table_0_resp.p4_action == 1, bb_action_0]
        - [table_0_resp.hit == 1 && table_0_resp.p4_action == 2, bb_action_1]
        - $done$

bb_action_0:
    type:           basic_block
    instructions:
        - [V, meta.udp_dport, table_0_resp.action_0_arg_0]
    next_control_state:
        - $offset$
        - $done$
```

```
bb_action_1:
    type:           basic_block
    instructions:
        - [V, meta.udp_chksum, table_0_resp.action_1_arg_0]
    next_control_state:
        - $offset$
        - $done$

ingress_control:
    type: control_flow
    start_control_state:
        - 0
        - bb_table_udp



# ----- ----- ----- ----- ----- ----- ----- -----
# Deparser
# ----- ----- ----- ----- ----- ----- ----- -----
deparse_eth:
    type:           basic_block
    local_header:   ethernet_t
    instructions:   []
    next_control_state:
        - $offset$ + 112
        - [meta.eth_type_ == 0x0800, deparse_ipv4]
        - $done$

deparse_ipv4:
    type:           basic_block
    local_header:   ipv4_t
    instructions:   []
    next_control_state:
        - $offset$ + 160
        - [meta.ipv4_proto == 17, deparse_udp]
        - $done$

deparse_udp:
    type:           basic_block
    local_header:   udp_t
    instructions:
        - [V, dport, meta.udp_dport]
        - [V, chksum, meta.udp_chksum]
    next_control_state:
        - $offset$ + 64
        - $done$

deparser:
    type: control_flow
    start_control_state:
        - 0
        - deparse_eth
```

```
# ----- ----- ----- ----- ----- ----- ----- -----
# System
# ----- ----- ----- ----- ----- ----- ----- -----
a_p4_switch:
    type:       processor_layout
    format:     list
    implementation:
        - parser
        - ingress_control
        - deparser



# ----- ----- ----- ----- ----- ----- ----- -----
# Table Initialization
# ----- ----- ----- ----- ----- ----- ----- -----
table_initialization:
    - table_0:
        key:
            dport : 0x3333
        value:
            hit:            1
            p4_action:      1
            action_0_arg0:  32
            action_1_arg0:  0
    - table_0:
        key:
            dport : 0x4444
        value:
            hit:            1
            p4_action:      2
            action_0_arg0:  0
            action_1_arg0:  0x0007
```

APPENDIX 4: Issues for further investigation

The following topics are mentioned elsewhere in this document as areas for further investigation:

- Whether a Processor Layout should have an arbitrary directed graph structure, rather than just an ordered linear list structure.
- Whether a Control Flow should have an optional action to describe packet cloning on entry to, or exit from, the Control Flow processor.
- Definition of the scope, representation and semantics of annotations.
- Definition of the runtime API to control BIR components.

The following topics are additional areas for further investigation:

- Whether there should be another class of instructions for assigning complete structs, for example, with the local header or a metadata collection as the destination or assignee.
- Whether the response struct for Tables should always begin with a standard bit indicating hit/miss, or whether an extra Table attribute should indicate whether or not there is such a bit. In addition, whether hit/miss is enough or whether a hit count would be more useful.
- Whether Tables could have additional functions for maintaining state between packets (by behaving as indexed tables rather than lookup tables): having readable/updateable/writeable entries from the data plane via V-type Instructions; and/or being readable through a control interface.
- Concurrent access to stateful information between packets when there are multiple access points. These situations may arise through Other Processors in the data flow that store state, callable methods in the control flow that store state, and Tables. In the future, there may also be other – more explicit – stateful store components. A possible rule is to allow multiple read points, but only single write points.
- Precise definition of the representation and semantics of expressions.
- At the end of the example in Appendix 3, there is a draft YAML representation of Table initialization (inherited by BIR from AIR), which uses a special-case known identifier: table_initialization. This could be changed to use another, new, BIR object type, with a standard YAML representation. A further possibility is to have a new initialization attribute for Tables (though retaining separate initialization objects too is likely to be useful to decouple initialization as part of the functional code from initialization to provide test data).