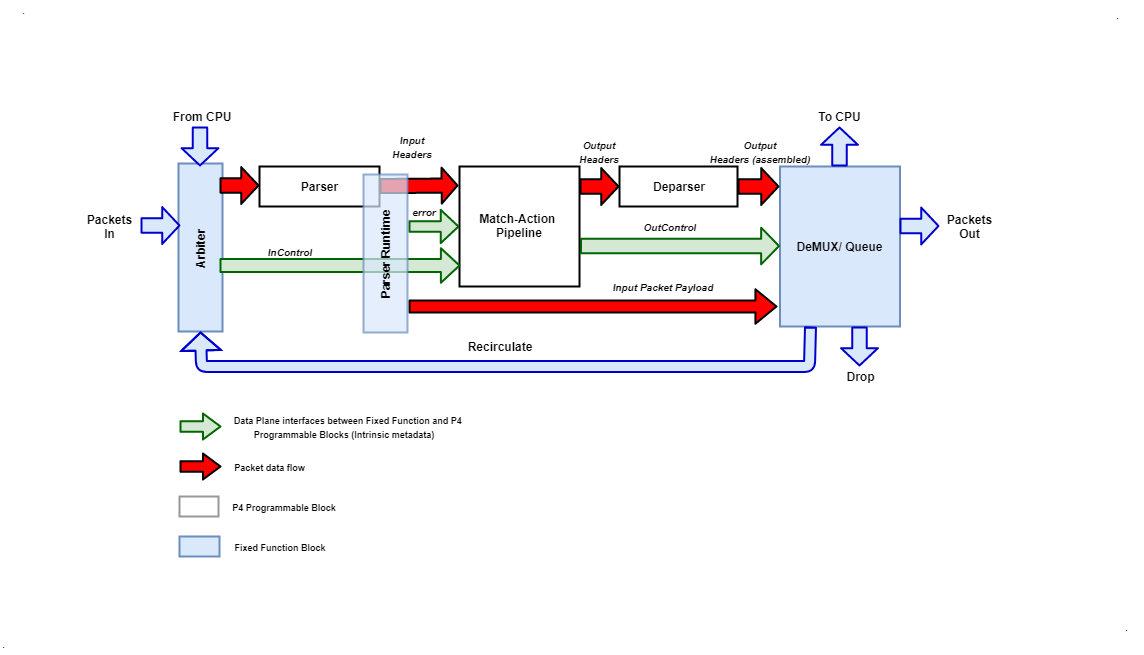
A very simple switch :

A switch is a device in a [computer network](https://en.wikipedia.org/wiki/Computer_network) that connects other devices together. Multiple data cables are plugged into a switch to enable communication between different networked devices. Switches manage the flow of data across a network by transmitting a received [network packet](https://en.wikipedia.org/wiki/Network_packet) only to the one or more devices for which the packet is intended. Each networked device connected to a switch can be identified by its [network address](https://en.wikipedia.org/wiki/Network_address), allowing the switch to direct the flow of traffic maximizing the security and efficiency of the network.

As and example to illustrate the features of archietecture consider implementing very simple switch in P4. We will first describe the architecture then write the P4 program that describes the data plane behavior of the switch.



**Figure 3**: Very Simple Switch Architecture

Archietecture Description

The “Very Simple Switch” (VSS) is the name given to the architecture. VSS receives packets through one of 8 input Ethernet ports, through a recirculation channel, or from a port connected directly to the CPU. VSS has one single parser, feeding into a single match-action pipeline, which feeds into a single deparser. After exiting the deparser, packets are emitted through one of 8 output Ethernet ports or one of 3 “special” ports:

* Packets sent to the “CPU port” are sent to the control plane
* Packets sent to the “Drop port” are discarded
* Packets sent to the “Recirculate port” are re-injected in the switch through a special input port

The white blocks in the figure are programmable, and the user must provide a corresponding P4 program to specify the behavior of each such block. The red arrows indicate the flow of user-defined data. The blue blocks are fixed-function components. The green arrows are data plane interfaces used to convey information between the fixed-function blocks and the programmable blocks—exposed in the P4 program as intrinsic metadata.VSS also has a number of fixed-function blocks whose behaviours are as follows:

1. Arbiter block

The input arbiter block performs the following functions:

* It receives packets from one of the physical input Ethernet ports, from the control plane, or from the input recirculation port.
* For packets received from Ethernet ports, the block computes the Ethernet trailer checksum and verifies it. If the checksum does not match, the packet is discarded. If the checksum does match, it is removed from the packet payload.
* After receiving a packet, the arbiter block sets the inCtrl.inputPort value that is an input to the parser with the identity of the input port where the packet originated. Physical Ethernet ports are numbered 0 to 7, while the input recirculation port has a number 13 and the CPU port has the number 14.

1. Parser runtime block

The parser runtime block works in concert with the parser. It provides an error code to the match-action pipeline, based on the parser actions, and it provides information about the packet payload (e.g., the size of the remaining payload data) to the demux block. As soon as a packet's processing is completed by the parser, the match-action pipeline is invoked with the associated metadata as inputs (packet headers and user-defined metadata).

1. DeMUX block

The core functionality of the DeMUX block is to receive the headers for the outgoing packet from the deparser and the packet payload from the parser, to assemble them into a new packet and to send the result to the correct output port. Payload is determined from the offset within the packet where parsing ended. The output port is specified by the value of outCtrl.outputPort, which is set by the match-action pipeline.

* Sending the packet to the drop port causes the packet to disappear.
* Sending the packet to an output Ethernet port numbered between 0 and 7 causes it to be emitted on the corresponding physical interface.
* Sending a packet to the output CPU port causes the packet to be transferred to the control plane. In this case, the packet that is sent to the CPU is the original input packet, and not the packet received from the deparser—the latter packet is discarded.
* Sending the packet to the output recirculation port causes it to appear at the input recirculation port. Recirculation is useful when packet processing cannot be completed in a single pass.
* If the outputPort has an illegal value (e.g., 9), the packet is dropped.

Architecture in P4 Language

The following P4 program provides a declaration of VSS in P4, as it would be provided by the VSS manufacturer. The declaration contains several type declarations, constants, and finally declarations for the three programmable blocks. The programmable blocks are described by their types; the implementation of these blocks has to be provided by the switch programmer.

// File : vss\_model.p4

# include <core.p4>

typedef bit<4> PortId;

const PortId REAL\_PORT\_COUNT = 4w8;

struct InControl {

    PortId inputPort;

}

const PortId RECIRCULATE\_IN\_PORT = 0xD;

const PortId CPU\_IN\_PORT = 0xE;

struct OutControl {

    PortId outputPort;

}

const PortId DROP\_PORT = 0xF;

const PortId CPU\_OUT\_PORT = 0xE;

const PortId RECIRCULATE\_OUT\_PORT = 0xD;

parser Parser<H>(packet\_in b, out H parsedHeaders);

control Pipe<H>(inout H headers, in error parseError, in InControl inCtrl, out OutControl outCtrl);

control Deparser<H>(inout H outputHeaders, packet\_out b);

package VSS<H>(Parser<H> p, Pipe<H> map, Deparser<H> d);

extern Checksum16 {

    Checksum16();

    void clear();

    void update<T>(in T data);

    void remove<T>(in T data);

    bit<16> get();

}

Element description of vss\_model.p4

* The included file core.p4 defines some standard data-types and error codes.
* bit<4> is the type of bit-strings with 4 bits.
* error is a built-in P4 type for holding error codes
* The declaration of a parser:

parser Parser<H>(packet\_in b, out H parsedHeaders);

This declaration describes the interface for a parser, but not yet its implementation, which will be provided by the programmer. The parser reads its input from a packet\_in, which is a pre-defined P4 extern object that represents an incoming packet, declared in the core.p4 library. The parser writes its output (the out keyword) into the parsedHeaders argument. The type of this argument is H, yet unknown—it will also be provided by the programmer.

* The declaration of a Match-Action pipeline named Pipe:

control Pipe<H>(inout H headers, in error parseError, in InControl inCtrl, out OutControl outCtrl);

The pipeline receives three inputs: the headers headers, a parser error parseError, and the inCtrl control metadata. The pipeline writes its outputs into outCtrl, and it must update in place the headers to be consumed by the deparser.

* The top-level package is called VSS; in order to program a VSS, the user will have to instantiate a package of this type (shown in the next section). The top-level package declaration also depends on a type variable H:

package VSS<H>

A type variable indicates a type yet unknown that must be provided by the user at a later time. In this case H is the type of the set of headers that the user program will be processing; the parser will produce the parsed representation of these headers, and the match-action pipeline will update the input headers in place to produce the output headers.

* The package VSS declaration has three complex parameters, of types Parser, Pipe, and Deparser respectively; which are exactly the declarations we have just described. In order to program the target one has to supply values for these parameters.
* In this program the inCtrl and outCtrl structures represent control registers. The content of the headers structure is stored in general-purpose registers.
* The extern Checksum16 declaration describes an extern object whose services can be invoked to compute checksums.

Implementation P4 Program for Very Simple Switch Architecture

# include <core.p4>

# include "vss\_model.p4"

typedef bit<48>  EthernetAddress;

typedef bit<32>  IPv4Address;

header Ethernet\_h {

    EthernetAddress dstAddr;

    EthernetAddress srcAddr;

    bit<16>         etherType;

}

header IPv4\_h {

    bit<4>       version;

    bit<4>       ihl;

    bit<8>       diffserv;

    bit<16>      totalLen;

    bit<16>      identification;

    bit<3>       flags;

    bit<13>      fragOffset;

    bit<8>       ttl;

    bit<8>       protocol;

    bit<16>      hdrChecksum;

    IPv4Address  srcAddr;

    IPv4Address  dstAddr;

}

struct Parsed\_packet {

    Ethernet\_h ethernet;

    IPv4\_h     ip;

}

// Parser section

error {

    IPv4OptionsNotSupported,

    IPv4IncorrectVersion,

    IPv4ChecksumError

}

parser TopParser(packet\_in b, out Parsed\_packet p) {

    Checksum16() ck;  // instantiate checksum unit

    state start {

        b.extract(p.ethernet);

        transition select(p.ethernet.etherType) {

            0x0800: parse\_ipv4;

            // no default rule: all other packets rejected

        }

    }

    state parse\_ipv4 {

        b.extract(p.ip);

        verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);

        verify(p.ip.ihl == 4w5, error.IPv4OptionsNotSupported);

        ck.clear();

        ck.update(p.ip);

        // Verify that packet checksum is zero

        verify(ck.get() == 16w0, error.IPv4ChecksumError);

        transition accept;

    }

}

// Match-action pipeline section

control TopPipe(inout Parsed\_packet headers,

                in error parseError, // parser error

                in InControl inCtrl, // input port

                out OutControl outCtrl) {

     IPv4Address nextHop;  // local variable

      action Drop\_action() {

          outCtrl.outputPort = DROP\_PORT;

      }

      action Set\_nhop(IPv4Address ipv4\_dest, PortId port) {

          nextHop = ipv4\_dest;

          headers.ip.ttl = headers.ip.ttl - 1;

          outCtrl.outputPort = port;

      }

     table ipv4\_match {

         key = { headers.ip.dstAddr: lpm; }  // longest-prefix match

         actions = {

              Drop\_action;

              Set\_nhop;

         }

         size = 1024;

         default\_action = Drop\_action;

     }

      action Send\_to\_cpu() {

          outCtrl.outputPort = CPU\_OUT\_PORT;

      }

     table check\_ttl {

         key = { headers.ip.ttl: exact; }

         actions = { Send\_to\_cpu; NoAction; }

         const default\_action = NoAction; // defined in core.p4

     }

      action Set\_dmac(EthernetAddress dmac) {

          headers.ethernet.dstAddr = dmac;

      }

      table dmac {

          key = { nextHop: exact; }

          actions = {

               Drop\_action;

               Set\_dmac;

          }

          size = 1024;

          default\_action = Drop\_action;

      }

       action Set\_smac(EthernetAddress smac) {

           headers.ethernet.srcAddr = smac;

       }

      table smac {

           key = { outCtrl.outputPort: exact; }

           actions = {

                Drop\_action;

                Set\_smac;

          }

          size = 16;

          default\_action = Drop\_action;

      }

      apply {

          if (parseError != error.NoError) {

              Drop\_action();  // invoke drop directly

              return;

          }

          ipv4\_match.apply(); // Match result will go into nextHop

          if (outCtrl.outputPort == DROP\_PORT) return;

          check\_ttl.apply();

          if (outCtrl.outputPort == CPU\_OUT\_PORT) return;

          dmac.apply();

          if (outCtrl.outputPort == DROP\_PORT) return;

          smac.apply();

    }

}

// deparser section

control TopDeparser(inout Parsed\_packet p, packet\_out b) {

    Checksum16() ck;

    apply {

        b.emit(p.ethernet);

        if (p.ip.isValid()) {

            ck.clear();              // prepare checksum unit

            p.ip.hdrChecksum = 16w0; // clear checksum

            ck.update(p.ip);         // compute new checksum.

            p.ip.hdrChecksum = ck.get();

        }

        b.emit(p.ip);

    }

}

// Instantiate the top-level VSS package

VSS(TopParser(), TopPipe(), TopDeparser()) main;

Command line options

The open source p4 compiler comes with many command line options but some of very useful commmand line options are below.

  -h, --help            show this help message and exit

  -b TARGET, --target TARGET

                        specify target device

  -a ARCH, --arch ARCH  specify target architecture

  -D PREPROCESSOR\_DEFINES

                        define a macro to be used by the preprocessor

  -E                    Only run the preprocessor

  -e                    Skip the preprocessor

  -I SEARCH\_PATH        Add directory to include search path

  -o PATH, --output PATH

                        Write output to the provided path

  --p4runtime-files P4RUNTIME\_FILES

                        Write the P4Runtime control plane API description

                        (P4Info) to the specified files (comma-separated

                        list); format is detected based on file suffix. Legal

                        suffixes are .txt, .json, .bin.

  --p4runtime-format {binary,json,text}

                        Choose output format for the P4Runtime API description

                        (default is binary). [Deprecated; use '--p4runtime-

                        files' instead]

  --std {p4-14,p4\_14,p4-16,p4\_16}

                        Treat subsequent input files as having type language.

In general, compiler command-line options can appear in any order in a single compiler invocation. However, the effects of some options depend on the order they appear in the command line and how they are combined with other related options.

The compiler enables you to use multiple options even where these might conflict.

You can use the environment variable to specify compiler command-line options. Options specified on the command line take precedence over options specified in the environment variable.

Steps to Compile a P4 program.

In the commands below, the -b option selects bmv2 (Behavioral Model Version 2) as the target, which is the software switch that we will use to run the P4 program.

The general command format :

p4c -b <target name> -I <include\_file> -o <output\_dir name>

p4c -b bmv2 test.p4 -o test.bmv2

This compiler generates a directory test.bmv2 which contains a file test.json which the generated “executable” code which is run by the software switch.

The P4 compiler also give us a command line options to get Runtime files for our P4 program. We will talk about P4 Runtime in the next section.

P4 Runtime

In the age of softwarization, SDN is unfolding its boundaries and industries are focusing to take networking control at top of the pyramid – from sophisticated hardware to software application. [P4](https://volansys.com/p4-programming-networks-forwarding-plane/)has already taken charge of the bottom of the pyramid (forwarding plane), by offering network programmability down to ASIC. Control Plane is the brain of networking devices. It has the various application that can learn and populate tables defined by data plane. In traditional network devices, the control plane or software which is used to configure the control movement of the packet was on the same device. Because of these, for any changes, the network admin has to configure the switch by separately connecting to each device. To add more complexity, each chip vendor provides their own proprietary interfaces/APIs for controlling data plane. To make a life of network admins easy, the goal was to separate the Control and Data Planes of networking devices. But still, due to different interface from different vendors, it was impossible to use single control plane APIs to control switch ASICs from different chip vendor. OpenFlow is the first standard open source communication protocol between the control plane and data plane by ONF (Open Networking Foundation) to promote adoption of SDN. OpenFlow is a solution to SDN networks which provides a standard interface between controller and switch in terms of flows. Flows contains match fields, priority and actions which are organized in tables. The APIs provided are generic, not dependent on target, thus making it suitable for any control plane software which are OpenFlow compatible.

There are various limitations with OpenFlow and SAI. Both were designed keeping in mind the fixed switches which makes it unscalable to new protocols in future. They are target independent but protocol dependent.

With P4 programmable data plane, there were no standard for control plane interface. Few vendors developed their own proprietary tools to auto-generate APIs which can populate tables from P4 program or JSON. But, there were no standard on how to define these APIs. P4 Runtime architecture makes it independent of protocols as well as underlying forwarding switch. The same API can be used to control different switches supporting different protocols.

P4 Runtime API have support for Managing behavior of data plane by adding, deleting, modifying, displaying entries in match-action tables. P4 compiler auto-generates the APIs that are needed to populate tables. P4 language can be used to describe the forwarding pipeline and P4 Runtime can be used control the forwarding and updating the forwarding logic. With P4, SDN controllers has gain ability to redefine tables, entries, parser, match actions, and packet processing logic. Thus giving complete control of the network.