Lab: Virtual Function Telemetry Using TCP Timestamp Options

This lab illustrates a simple method for measuring the transit and processing time for packets handled in a PCIe Virtual Function (VF). This is achieved by creating a program to match traffic entering the Network Flow Processor from the physical network port, and sending the traffic to its PCIe interface (i.e. the network device driver in the kernel of the host OS) with a TCP TimeStamp (TS) option inserted. Conversely traffic received at the PCIe interface has the timestamp option removed, timestamp calculations executed and is then forwarded to the physical network.

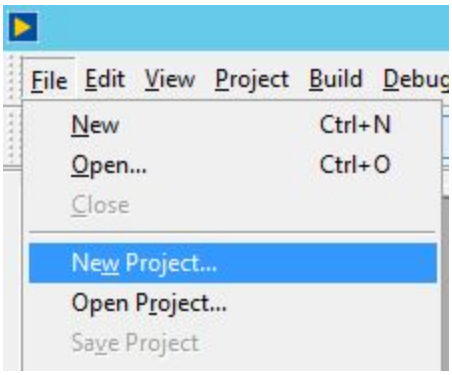
The program is constructed in three phases. We start with a 'boiler-plate' TCP/IP forwarding skeleton source. This includes all requisite code for verifying and calculating TCP checksums and doing basic forwarding. We then program parsing and inserting the timestamp header. Finally we add C sandbox code which performs the statistics processing and P4 code to remove the inserted timestamp option.

Part 1: Boilerplate TCP forwarding

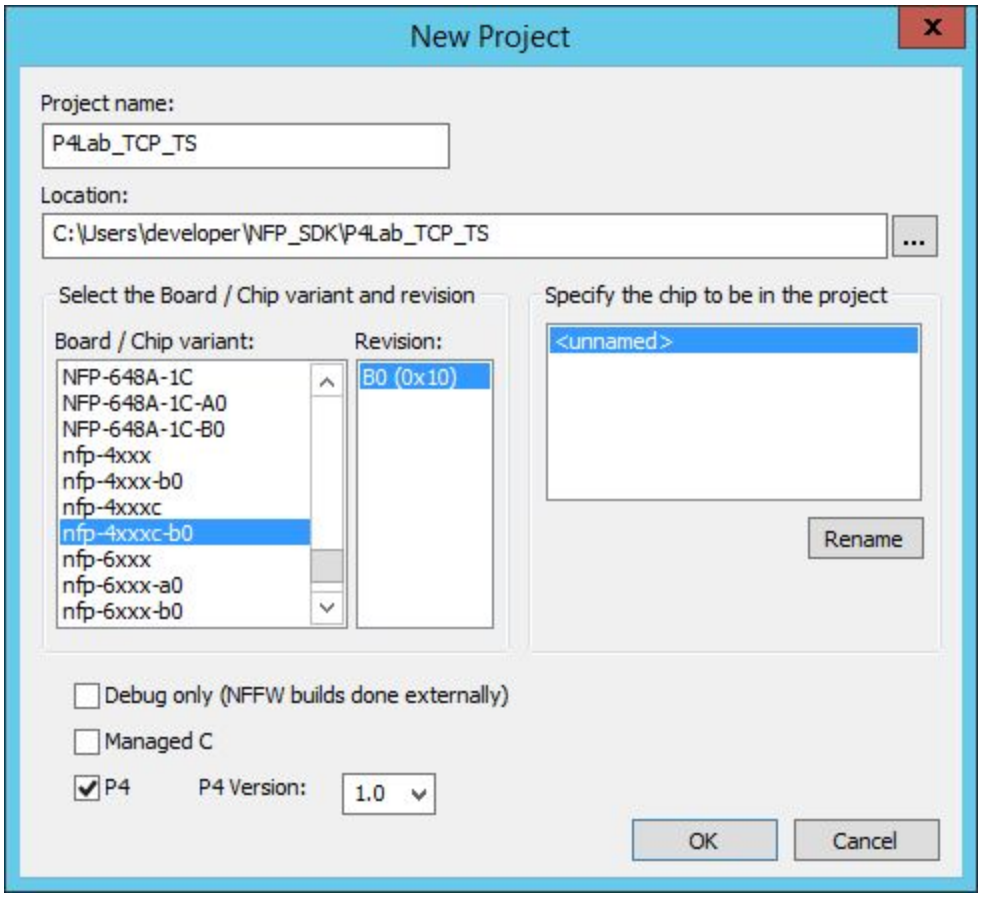
The first step is to create a skeleton Programmer Studio (PS) project and add the boilerplate TCP forwarding P4 source.

1. Open Programmer Studio by double clicking on the icon on your desktop.

2. Create a new project by choosing File -> New Project:

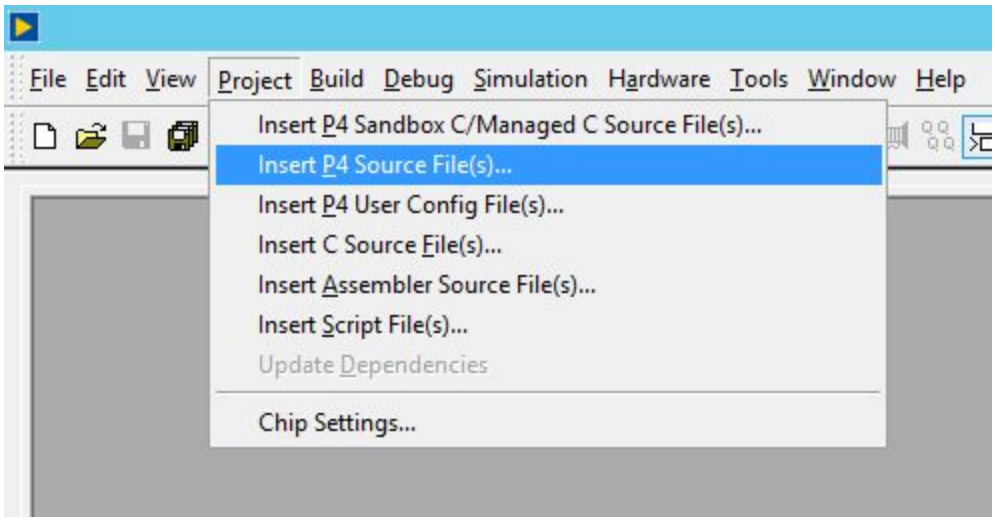


3. In the *New Project* dialogue choose a name for your project, select platform nfp-4xxx-b0 and the *P4* checkbox before pressing *OK*

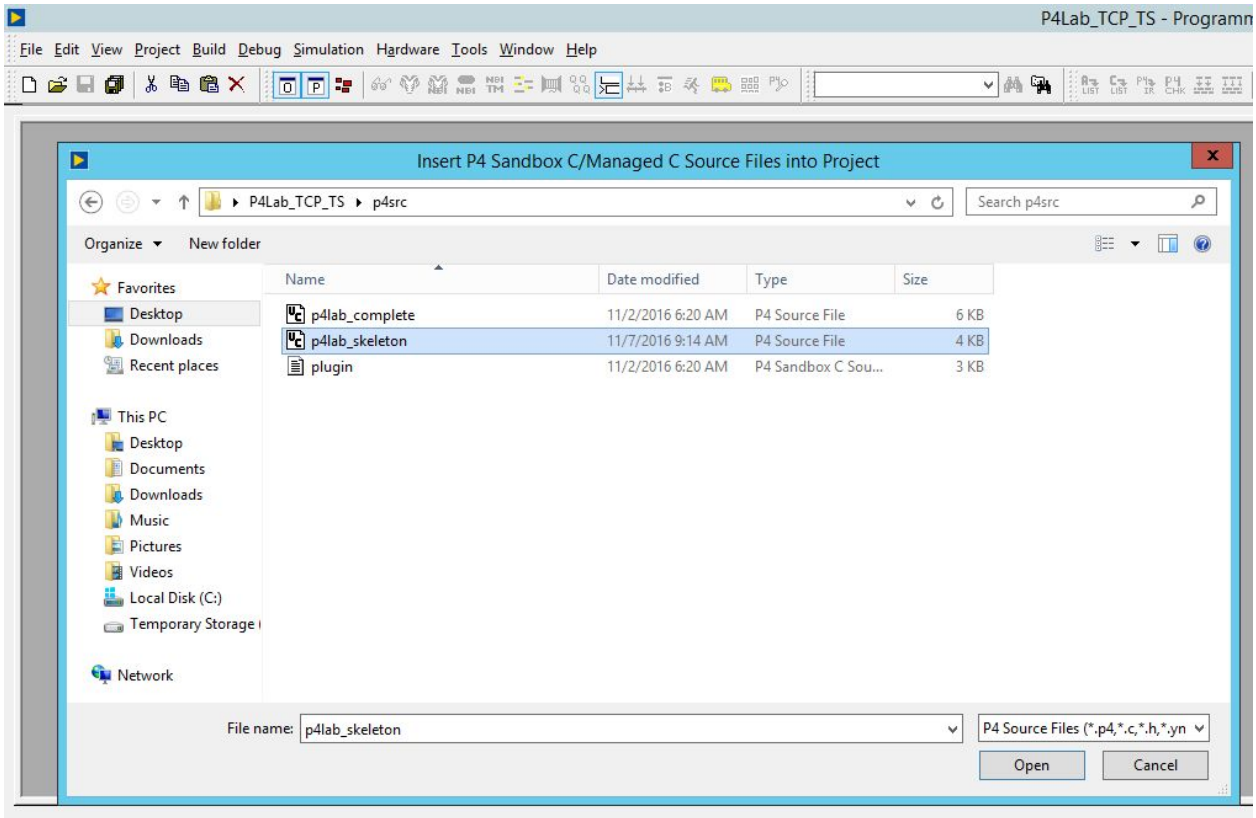


Note: Take care to select the P4 checkbox, as otherwise the project will not be using the P4 perspective. Also make sure you select 1.0 for the version.

4. Add the P4 source which will be found in the P4Lab_TCP_TS folder on the desktop (or whichever folder the demo resources were installed). To do this click *Project -> Insert P4 Source File(s)*.

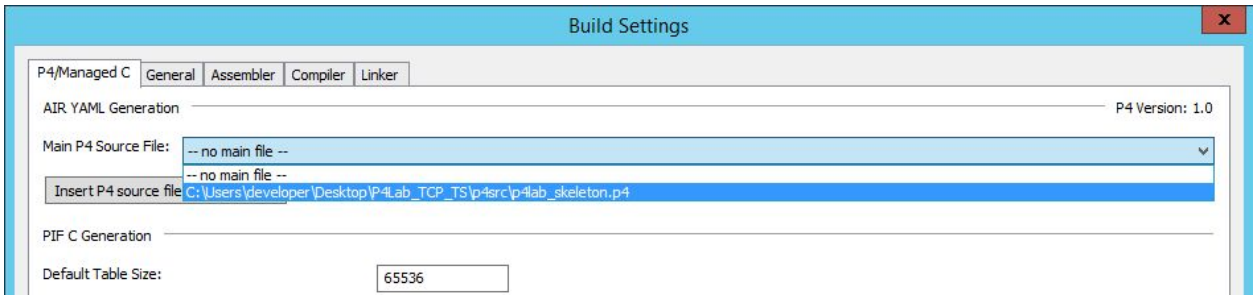


5. Now navigate to the *p4src* folder and select the file *p4lab_skeleton.p4*.

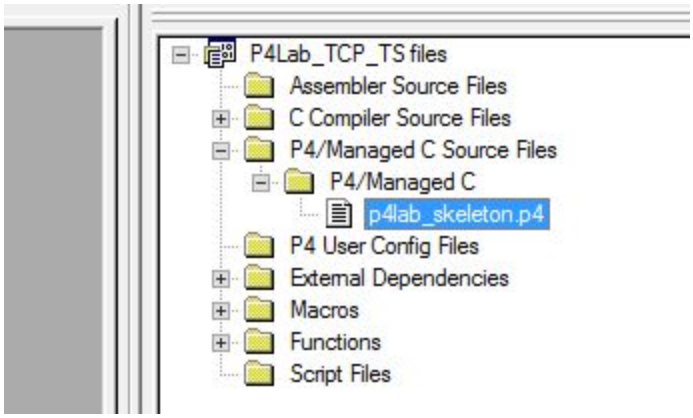


6. Select the source as the top-level P4 file by clicking *Build -> Settings…* then select the added

source from the drop down *Main P4 Source File*; click *OK*.



7. Now open the newly added source from the Programmer Studio File FileView window by double clicking *FileView->P4/Managed C Source Files->p4lab_skeleton.p4*



Now review the source; *p4lab_skeleton.p4* include the following:

- Header type definitions and instantiations for *Ethernet, IPv4 and TCP*
- ***Calculated Field*** definitions for IPv4 and TCP checksums
    - Calculated fields are the P4 language construct which facilitates automatic checksum verification on ingress and calculation on egress.
- Parsing for *Ethernet, IPv4 and TCP*
- A single table which matches on *standard_metadata.ingress_port* and executes a simple forwarding action on ingress
- An empty egress control flow

Before moving on let us review how calculated fields work by looking at the TCP checksum implementation. The first step is to define the list of fields that will be used in the calculation. This is done with a *field_list* construct shown below. A *field_list* is a collection of header fields, metadata fields and sized zero padding. Note the the keyword **payload** in the list indicates that all of the packet data beyond the *tcp.urgentPtr* field should be included in the calculation.

```
field_list tcp_ipv4_checksum_list {
```

```
        ipv4.srcAddr;
        ipv4.dstAddr;
        8'0;
        ipv4.protocol;
        tcp_ipv4_metadata.tcpLength;
        tcp.srcPort;
        tcp.dstPort;
        tcp.seqNo;
        tcp.ackNo;
        tcp.dataOffset;
        tcp.res;
        tcp.flags;
        tcp.window;
        tcp.urgentPtr;
        payload;
}
```

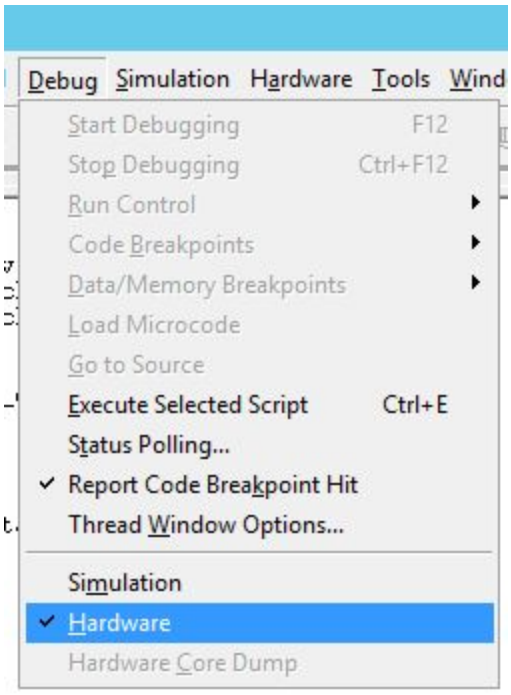
After the list of fields is defined we define the algorithm to apply to them this is done with a *field_list_calculation* construct as shown below:

```
field_list_calculation tcp_ipv4_checksum {
        input {
            tcp_ipv4_checksum_list;
        }
        algorithm : csum16;
        output_width : 16;
}
```

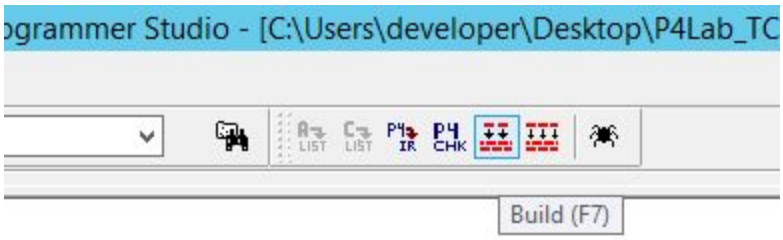
Finally we define which field contains the checksum and also whether and under what conditions the checksum should be checked and updated in the *calculated_field* construct as follows:

```
calculated_field tcp.checksum {
        verify tcp_ipv4_checksum if(valid(ipv4));
        update tcp_ipv4_checksum if(valid(ipv4));
}
```

8. Select hardware debug mode by clicking *Debug -> Hardware.* **THIS MUST BE DONE BEFORE BUILDING!**

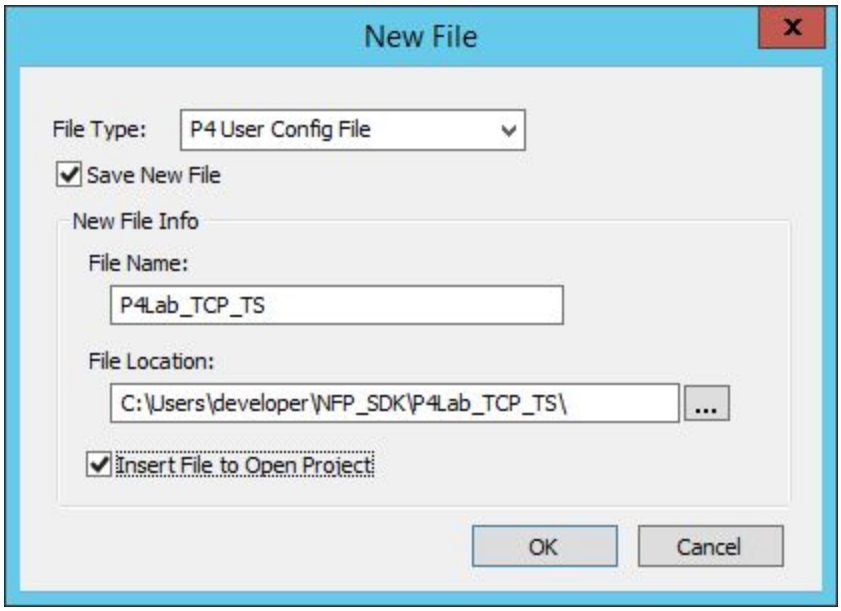


9. Build the source by clicking the Build button (or by pressing F7)

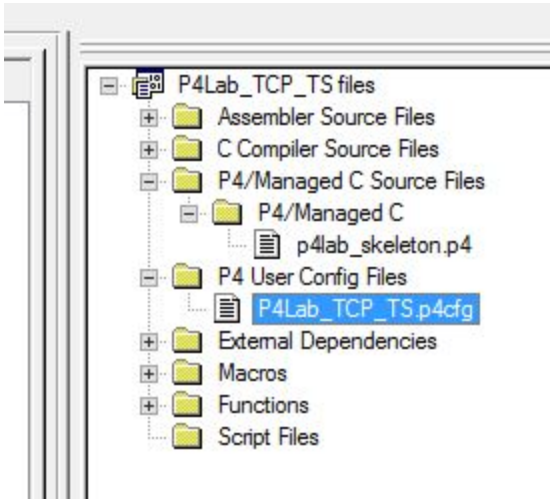


10. Create a new P4 user configuration by clicking *File->New* again.

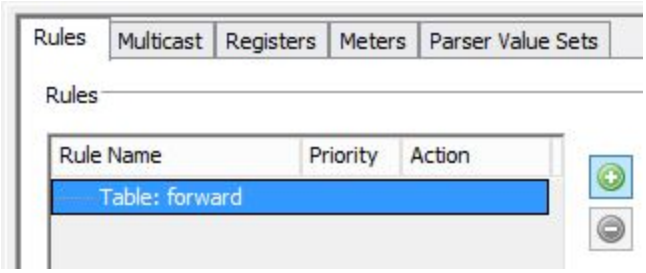
Now select *P4 User Configuration* from *File Type* drop down. Enter a name for the file in *File Name* textbox and click the *Insert into Open Project* checkbox.



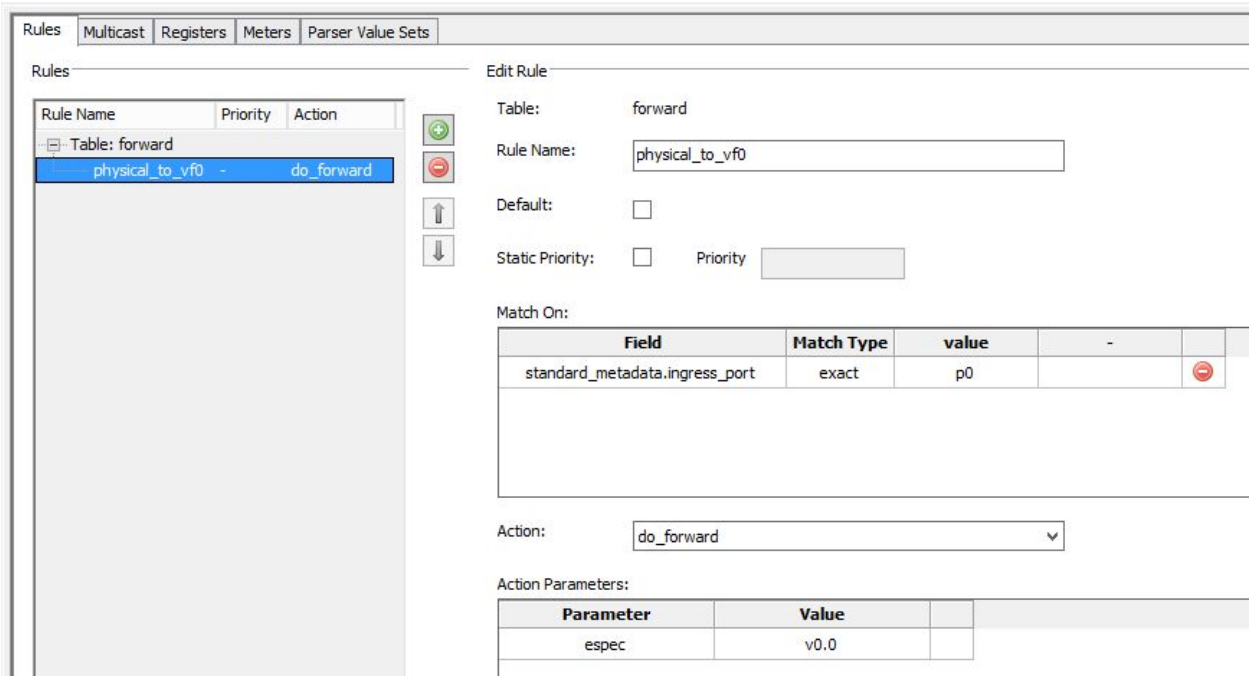
Open the rules configuration by clicking *FileView -> P4 User Config Files -> P4Lab_TCP_TS.p4cfg*



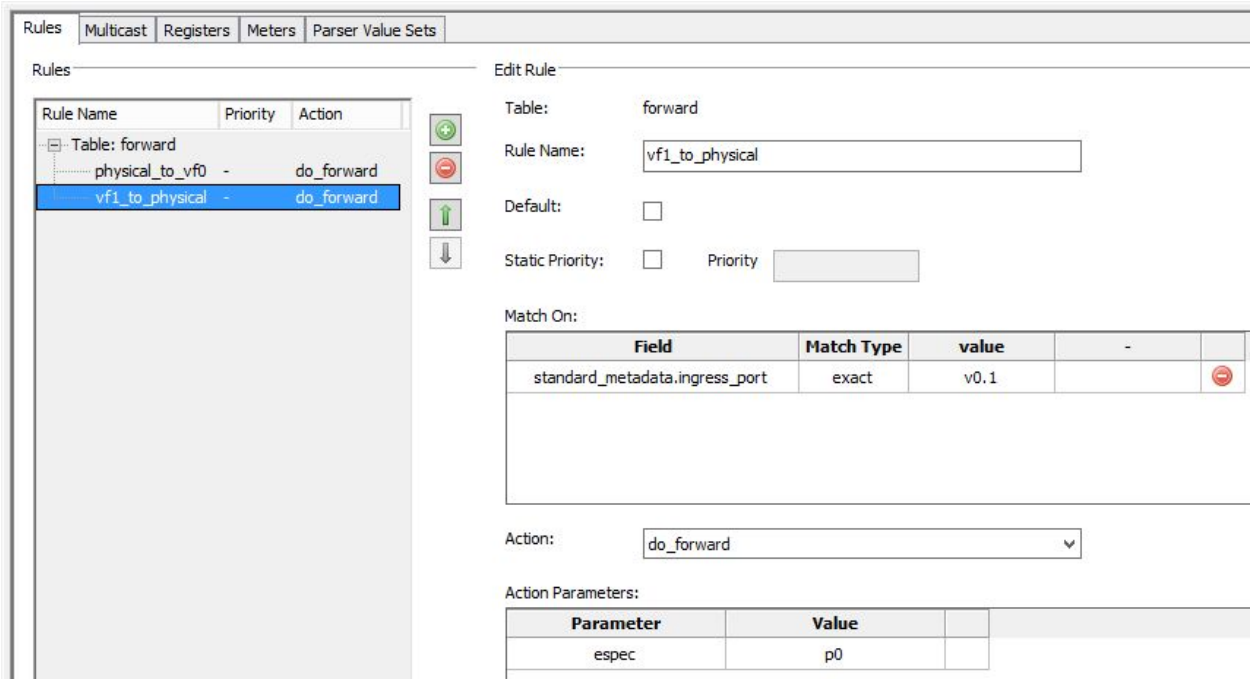
11. Now add a rule to forward traffic from the physical interface to PCIe VF 0. To add the rule first click on the *Table: forward* table within the *Rules* tab; then click on the add rule button:



Give the rule a name in the *Rule Name* textbox. Now populate the value of *match on* with *p0.* This will apply this rule to all packets from physical port 0 of the NIC. Now select the *do_forward* action from the *Action* dropdown. Set the action parameter *espec* to be *v0.0.* This will have the effect of forwarding all traffic with physical port 0 to PCIe VF 0.

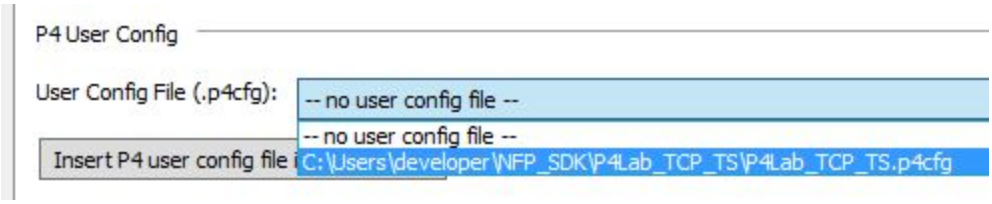


12. Now add a rule to forward traffic from PCIe VF 1 back to physical port 1. Create a new rule by clicking the add rule button again. Give the rule a new name. Similar to before enter *v0.1* in *value* field in the *match on* dialogue and *p0* in action parameter field.



13. Save by pressing CTRL-S

14. Now select your configuration as the project configuration by clicking *Build -> Settings…* then select your configuration from the *P4 User Config* dropdown.



Now we will add the TCP Timestamp option to the parser and create a table and action to insert the header.

15. Open the *p4lab_skeleton.p4* in the IDE editor. Create the header type definition for *tcp_tsopt_t* as below. Note that we will only support one TCP option so we include 16 bits of padding in the header definition; this makes the demo a lot simpler (note: the code to be added is shown below in **boldface**, the existing code is depicted in normal typeface):

```
header_type tcp_t {
      fields {
          srcPort : 16;
          dstPort : 16;
          seqNo : 32;
```

```
        ackNo : 32;
        dataOffset : 4;
        res : 4;
        flags : 8;
        window : 16;
        checksum : 16;
        urgentPtr : 16;
    }
}

header_type tcp_tsopt_t {
    fields {
        kind : 8;
        len : 8;
        ts_val : 32;
        ts_ect : 32;
        zero_padding : 16;
    }
}
```

16. Now instantiate the header:

```
header tcp_t tcp;
header tcp_tsopt_t tcp_tsopt;
```

17. Create the new *parse_tcp_tsopt* parser node which will extract the header:

```
parser parse_tcp {
    extract(tcp);
    set_metadata(tcp_ipv4_metadata.tcpLength, ipv4.totalLen -
(tcp_ipv4_metadata.scratch << 2));
    return ingress;
}

parser parse_tcp_tsopt {
    extract(tcp_tsopt);
    return ingress;
}
```

18. Modify the *parse_tcp* parser node to conditionally branch to the new parser node *parse_tcp_tsopt:*

```
parser parse_tcp {
      extract(tcp);
      set_metadata(tcp_ipv4_metadata.tcpLength, ipv4.totalLen -
(tcp_ipv4_metadata.scratch << 2));
      return select(latest.dataOffset, current(0,8)) {
          0x0808 : parse_tcp_tsopt;
          default : ingress;
      }
}
```

With the above change the *parse_tcp_tsopt* parser node will be entered if *tcp.dataOffset* is 8 (the expected offset when just the *tcp_tsopt* header is present) and the first 8-bits of packet data after the *tcp* header is 0x8 which is the option kind value for TCP TS (this is achieved using the *current* P4 construct). Note that this parser will ONLY work with a single timestamp option.

19. Declare the special intrinsic timestamp metadata type, this will automatically reflect the packet timestamp immediately before parsing:

```
header_type tcp_ipv4_metadata_t {
    fields {
        scratch : 16;
        tcpLength : 16;
    }
}

header_type intrinsic_metadata_t {
    fields {
        ingress_global_tstamp : 32;
    }
}
```

20. Instantiate the intrinsic timestamp metadata:

```
metadata tcp_ipv4_metadata_t tcp_ipv4_metadata;
metadata intrinsic_metadata_t intrinsic_metadata;
```

IMPORTANT: the spelling for the intrinsic metadata header definition and instantiation MUST be the same as above or the timestamp won't be populated.

21. Create an action to insert and populate the new *tcp_tsopt* header:

```
/*
 * Egress
 */

action do_insert_tsopt() {
      add_header(tcp_tsopt);

      modify_field(tcp_tsopt.kind, 8);
      modify_field(tcp_tsopt.len, 10);

      modify_field(tcp_tsopt.ts_ect, 0);
      modify_field(tcp_tsopt.zero_padding, 0);

      /* insert packet timestamp */
      modify_field(tcp_tsopt.ts_val,
                   intrinsic_metadata.ingress_global_tstamp);

      /* adjust sizes */
      add_to_field(tcp.dataOffset, 3);
      add_to_field(tcp_ipv4_metadata.tcpLength, 12);
      add_to_field(ipv4.totalLen, 12);
}

control egress {
}
```

22. Add a table to execute the insert action:

```
      add_to_field(ipv4.totalLen, 12);
}

table insert_tsopt {
    reads {
        standard_metadata.egress_port: exact;
    }
```

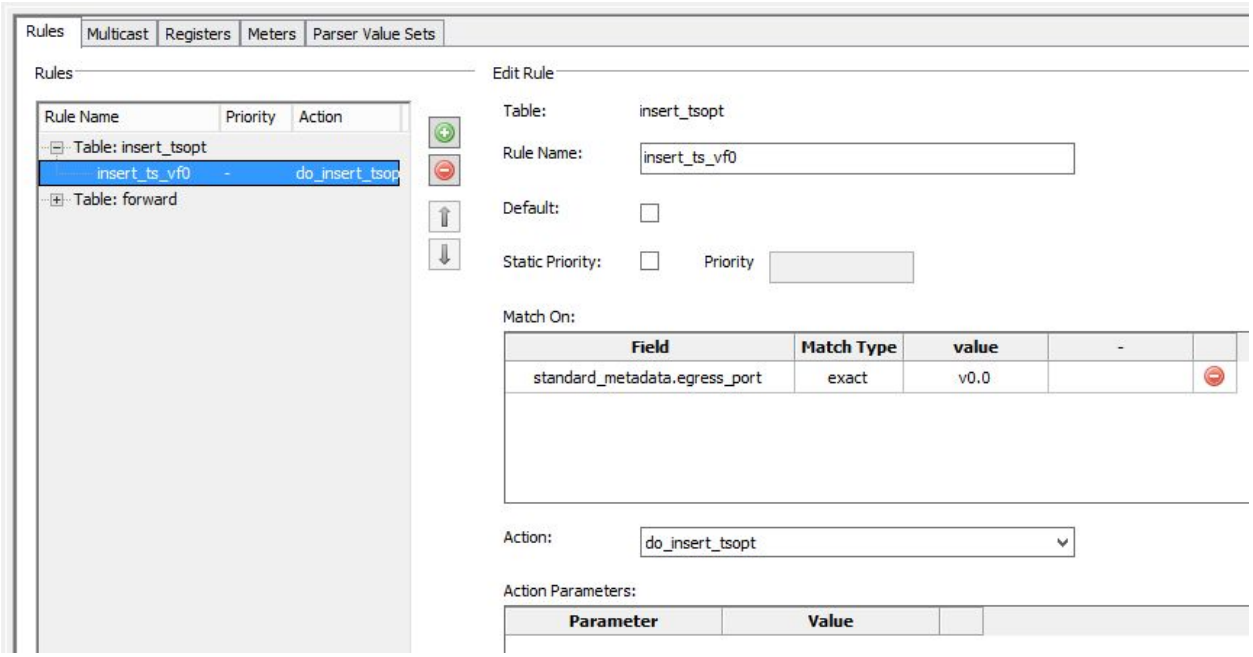
```
    actions {
        do_insert_tsopt;
    }
}

control egress {
}
```

23. Apply the table in the egress control flow conditionally if *tcp* is valid and *tcp_tsopt* is not valid:

```
table insert_tsopt {
    reads {
        standard_metadata.egress_port:  exact;
    }
    actions {
        do_insert_tsopt;
    }
}

control egress {
    if (valid(tcp) and not valid(tcp_tsopt)) {
        apply(insert_tsopt);
    }
}
```

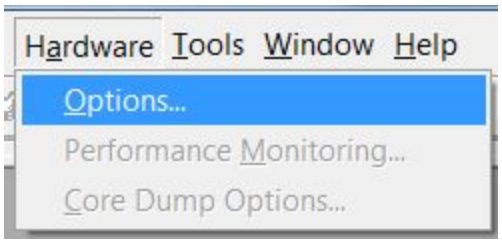
24. Save and build the project by pressing the build button as before or by pressing F7.

25. Similar to before add a rule to the *insert_tsopt* table to insert the *tcp_tsopt* header for packets destined to PCIe VF 0 and save your rules file. Note there are no action parameters for this rule:

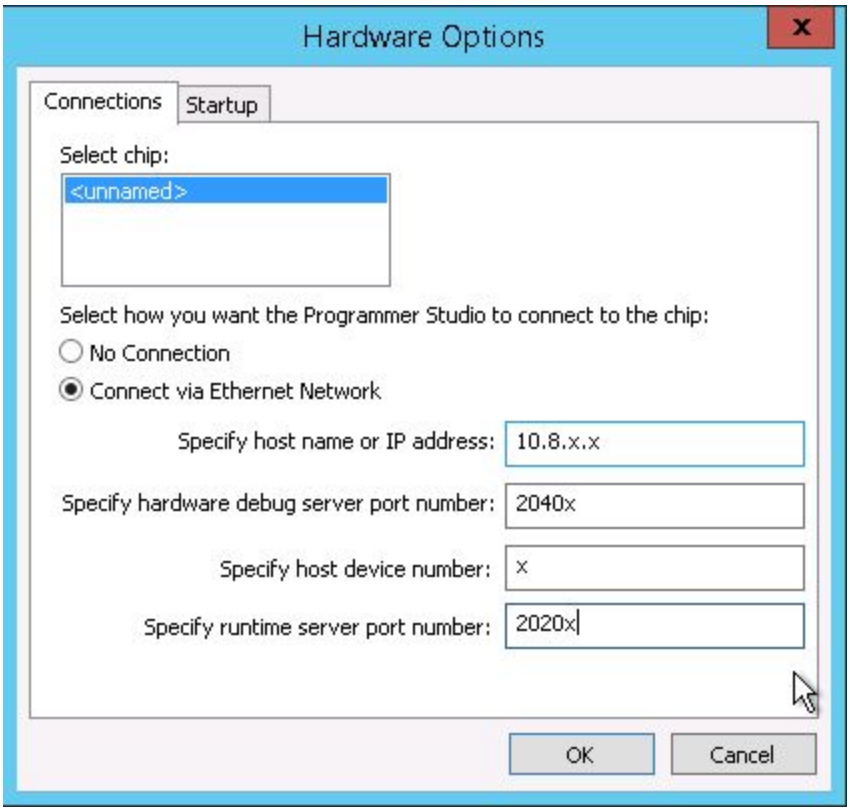


26. We now specify hardware debugging options.

a. Select the hardware debugging options dialog:



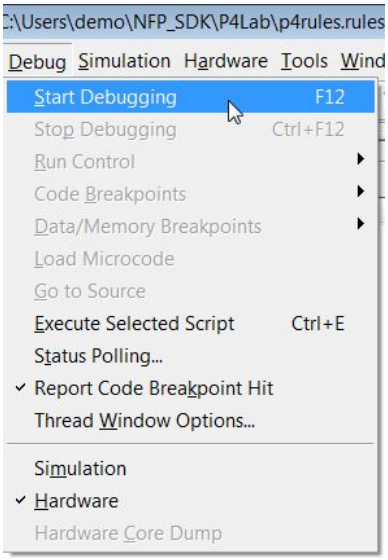
b. Specify the target hardware details:



Select “Connect via Ethernet Network”, type your designated details into the appropriate input fields and click OK.

27. Load your design:

a. Start debugging by pressing F12, or by using the menu:



b. Click on the *GO* toolbar button or press F5:

28. Open two putty sessions on the traffic generation server:

Log in to your assigned server as your designated user “sdkuser1” or “sdkuser2” with password “netronome”.

29. Inject a valid TCP packet

a. Start a tcpdump instance to view traffic on VF0 by entering:

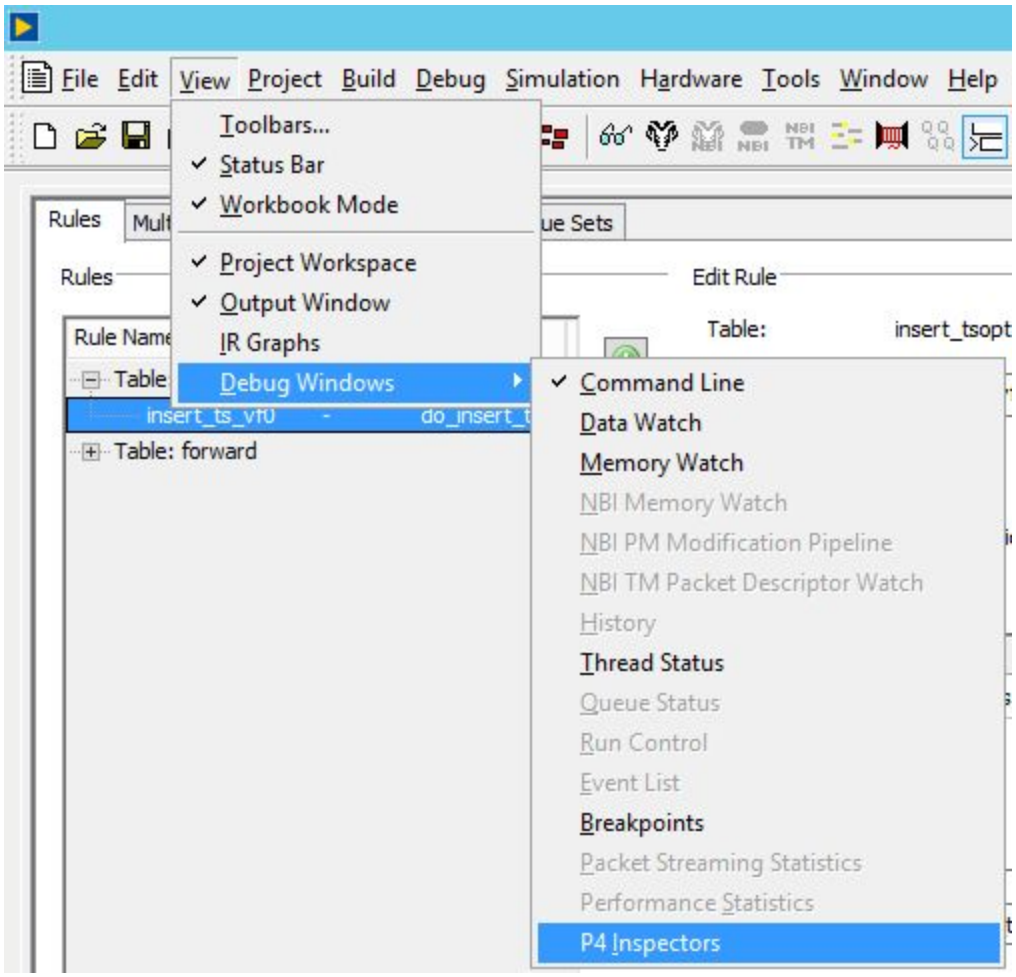
```
tcpdump -l -n -xx -vv -i v0.0
```

b. Now inject a packet using tcpreplay:

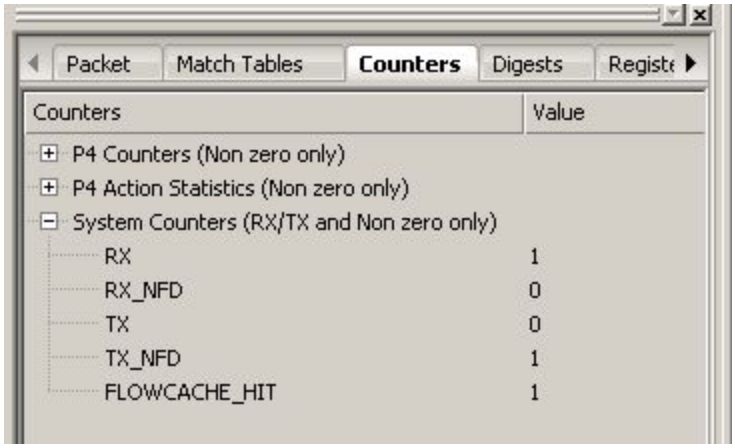
```
tcpreplay -i p1p1 /root/tcp.pcap
```

30. Open the *counters* tab in the PS P4 inspector window to view datapath statistics.

a. Open the P4 Inspectors window by clicking *View-> Debug Windows->P4 Inspectors*



b. Click on the *Counters* tab in the P4 Inspectors pane and note the RX counter reflects the packet arriving on the physical interface and TX_NFD reflects the packets sent to the PCIe VF:



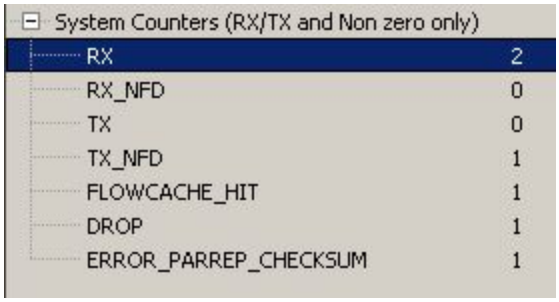
Note: you will need to right click on the pane and click on *Refresh* to update counters after sending packets into the system.

31. Inject an invalid tcp packet to the physical port by entering:

```
tcpreplay -i p1p1 /root/tcp_invalid.pcap
```

Note that the packet is not reflected in the tcpdump window.

32. The counters will now reflect a drop due to a failed checksum verification:



33. Now inject another valid tcp packet to the physical port by entering:

```
tcpreplay -i p1p1 /root/tcp.pcap
```

In the tcpdump output, note the additional TS option and that the checksum is correct:

```
     10.0.0.1.3000 > 10.0.0.100.4000: Flags [S], cksum 0xef52
(correct), seq 0:16, win 8192, options [TS val 1254929824 ecr 0,eol],
length 16
```

Resend packet using tcpreplay as above and note the incrementing timestamp value for every packet incrementing.

```
     10.0.0.1.3000 > 10.0.0.100.4000: Flags [S], cksum 0x8de7
(correct), seq 0:16, win 8192, options [TS val 1335169603 ecr 0,eol],
length 16
     10.0.0.1.3000 > 10.0.0.100.4000: Flags [S], cksum 0xaffe
(correct), seq 0:16, win 8192, options [TS val 1391717581 ecr 0,eol],
length 16
```

Note that this timestamp is a measurement in MicroEngine cycles; the conversion from cycles to microseconds will be covered in the next section.

34. Click the stop debugging toolbar button or press Ctrl + F12:

Click Yes to confirm the action and then Yes again to let the MicroEngines run.

Part 2: Processing timestamps and removing the TCP Timestamp option

We will now add statistics processing for timestamps and the ability to remove the Timestamp option.

1. The timestamp processing is done in a Sandbox C function. The function is declared by defining a custom action primitive:

```
/*
 * Ingress
 */


primitive_action tsopt_statistics();

action do_forward(espec) {
```

2. Define an action *do_process_tsopt()* which will call the custom primitive *tsopt_statistics* as well as removing an existing *tcp_tsopt* header and updating various lengths:

```
/*
 * Ingress
 */


primitive_action tsopt_statistics();

action do_process_tsopt() {
    tsopt_statistics();

    remove_header(tcp_tsopt);
    /* update lengths */
    subtract_from_field(tcp.dataOffset, 3);
    subtract_from_field(tcp_ipv4_metadata.tcpLength, 12);
    subtract_from_field(ipv4.totalLen, 12);
}
```

3. Now add a table, *process_tsopt,* which will call the *do_process_topt* action:

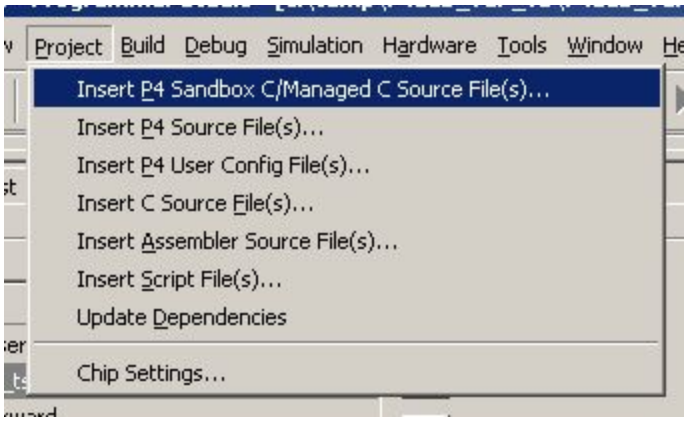
```
action do_process_tsopt() {
    tsopt_statistics();
}

table process_tsopt {
    actions {
        do_process_tsopt;
    }
}
```

4. Add the TCP timestamp option processing to the ingress control flow:

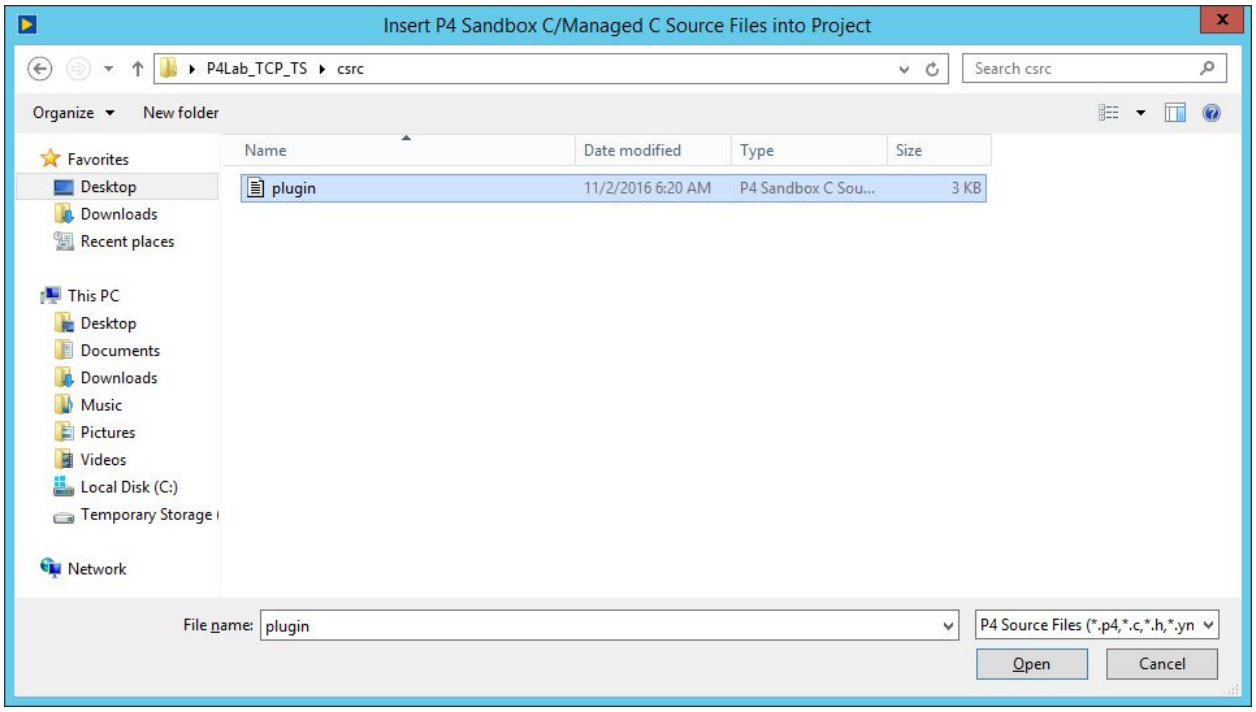
```
control ingress {
    if (valid(tcp_tsopt)) {
        apply(process_tsopt);
    }
    apply(forward)
}
```

Save the P4 source file.

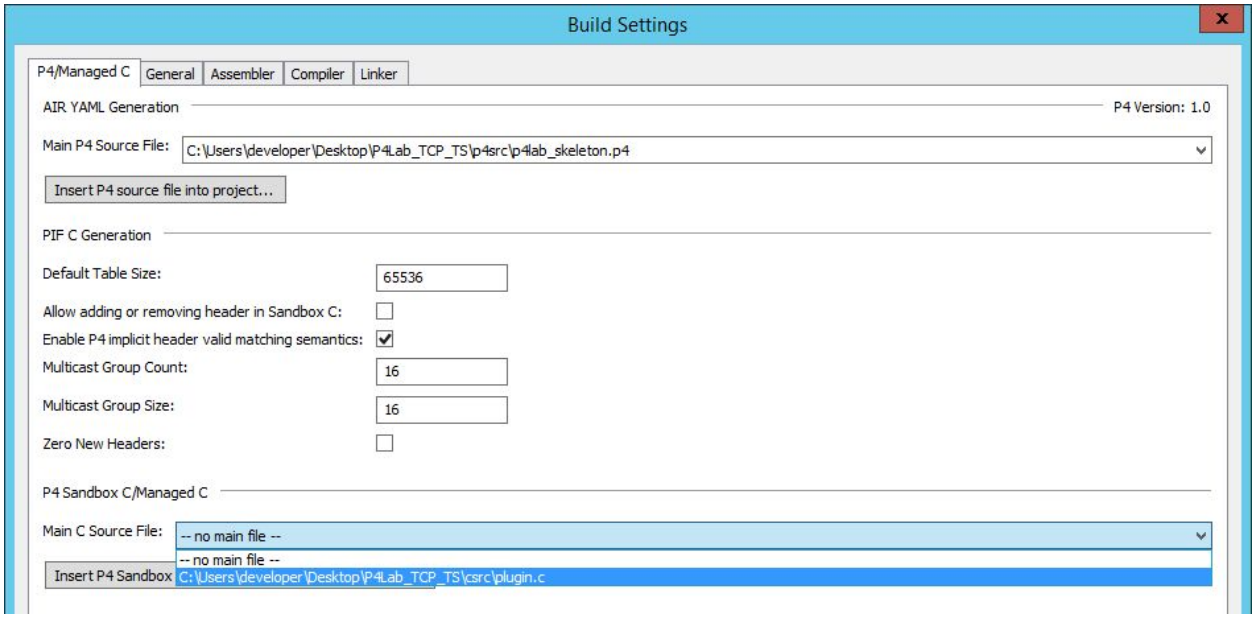
5. Add MicroC source which will be found in the P4Lab_TCP_TS folder on the desktop (or whichever folder the demo resources were installed). To do this click *Project -> Insert P4 Sandbox C / Managed C Source File(s)*:



Now navigate to the *csrc* folder and select the file *plugin.c*:



6. Open the build setting by clicking *Build -> Settings…* then select the *plugin.c* option from the *Main C Source File* drop down:



7. Open the *plugin.c* file in *FileView->P4/Managed C Source Files->P4 Managed C*

8. Review *plugin.c*; this file contains MicroC source that does the following:

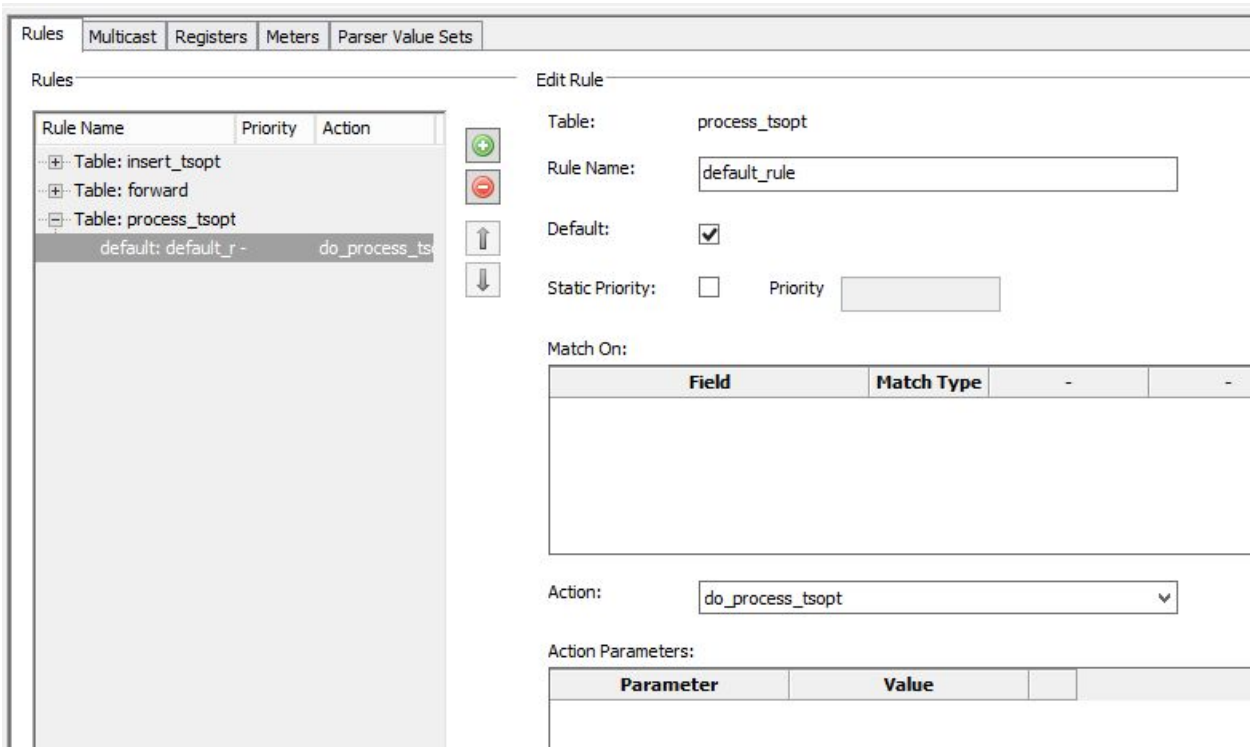
- Declares a statistics data structure per ingress port

    - This data structure is declared exported, this means we can access it from the x86 host which will be done later
- Reads P4 metadata for ingress port and packet timestamp
- Calculates the VF latency by subtracting the TCP Timestamp value from the timestamp for the current packet
- Stores the minimum, maximum, and accumulated latency per ingress port.

9. Build the project by either clicking the build button or by pressing *F7.*

10. Open the P4 user config file by clicking *Fileview->P4 User Config Files->P4Lab_TCP_TS.p4cfg*

11. Add a **default** rule for the *process_tsopt* table that executes the *do_insert_tsopt* action; this will unconditionally execute the *do_process_tsopt* action for every packet with the *tcp_tsopt* header present:



Save the rules.

12. Starting debugging by clicking the debug toolbar icon or by pressing *F12.* Remember to press the run button once the debug loading process is done.

13. From a traffic generation terminal create a network bridge between VF network devices *v0.0 and v0.1* as follows:

```
brctl addbr br0
brctl addif br0 v0.0 v0.1
ifconfig br0 up
```

Traffic entering *v0.0* will now be forwarded to *v0.1*.

14. Start tcpdump on the physical interface

```
tcpdump -l -n -xx -vv -i p1p1
```

15. Inject a valid tcp packet to the physical port by entering:

```
tcpreplay -i p1p1 /root/tcp.pcap
```

The tcpdump window will reflect both the packet entering the SmartNIC and the packet passed through the VFs, note that they should be the same.

16. Stop the tcpdump process by pressing *Control-C* and running the following command, which will dump the raw packet statistics:

```
watch -n1 -d /root/stats.sh
```

17. Insert multiple packets using tcpreplay as above and note the changing statistics values:

```
Every 1.0s: /root/stats.sh                  Tue Nov  8 18:33:09 2016

NFP: 0
SPEED: 700 MHz
PACKETS: 2
MAX: 0.156549 ms
MIN: 0.127749 ms
AVE: 0.142149 ms
```

19. As an exercise, experiment with introducing additional latency into the mock VNF by using netem to impair the egress VF and note the influence on the measured latency:

```
tc qdisc add dev v0.1 root netem delay 100ms
```

THE INFORMATION IN THIS DOCUMENT IS AT THIS TIME NOT A CONTRIBUTION TO P4.ORG.