OpenFlow implementation on NetFPGA-10G

Design Document

Tatsuya Yabe

Index:

1. Introduction

1-1. About the design, the design document and the target audience

This is a design document for OpenFlow switch on NetFPGA-10G platform (OF-NF10G). It consists of a hardware portion of an OpenFlow switch implementation and its associated software. The main role of the hardware portion is to modify packet header fields and to forward it from one port to another/other port(s) at a line rate, with referring to flow tables residing in hardware.

The software portion, which is briefly discussed but is out of scope of this document, is in charge of exchanging OpenFlow Protocol messages with a controller, aggregating and reporting various statistics, as well as interfacing with hardware for setting up flow entries and handling packet_in, packet_out and unsupported packets by hardware.

Fig1.1 shows a simple structure of OF-NF10G switch and a controller.

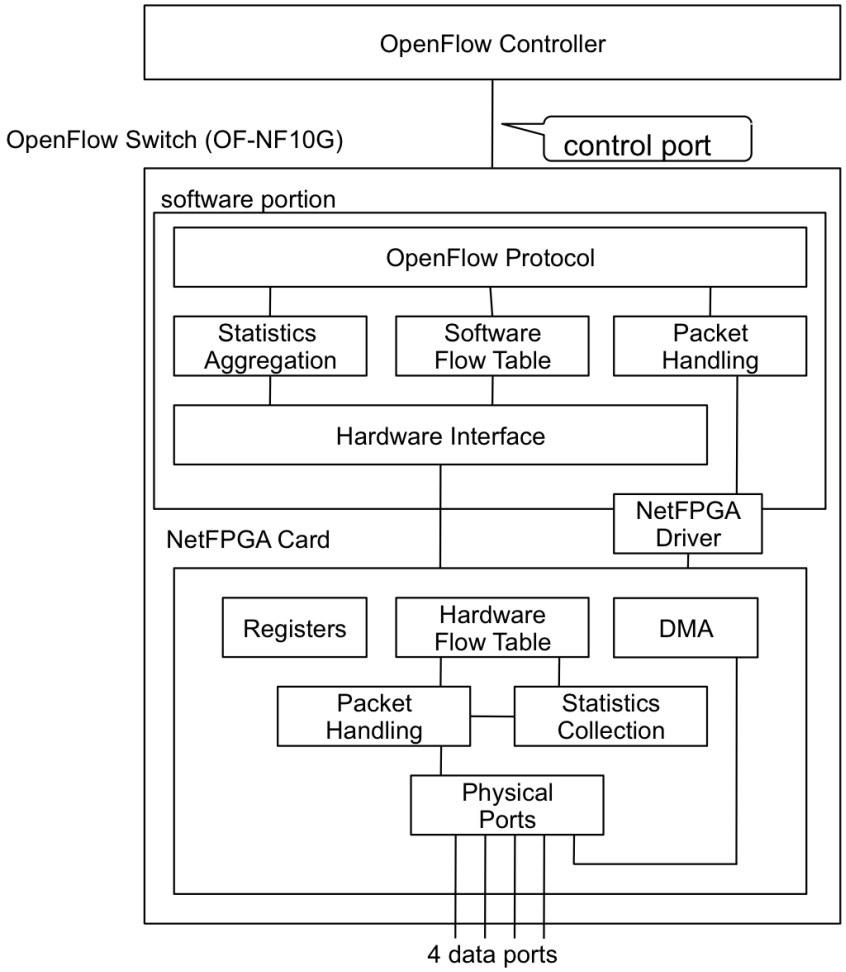


Fig1.1 Simple OpenFlow structure

The main target audience of this document is who wants to learn inside the hardware

implementation for modifying and/or adding functions of this implementation.

1-2. Supported functions and ongoing milestones

The current implementation has limited functionalities. The current status and future plans are shown in Fig1.2.

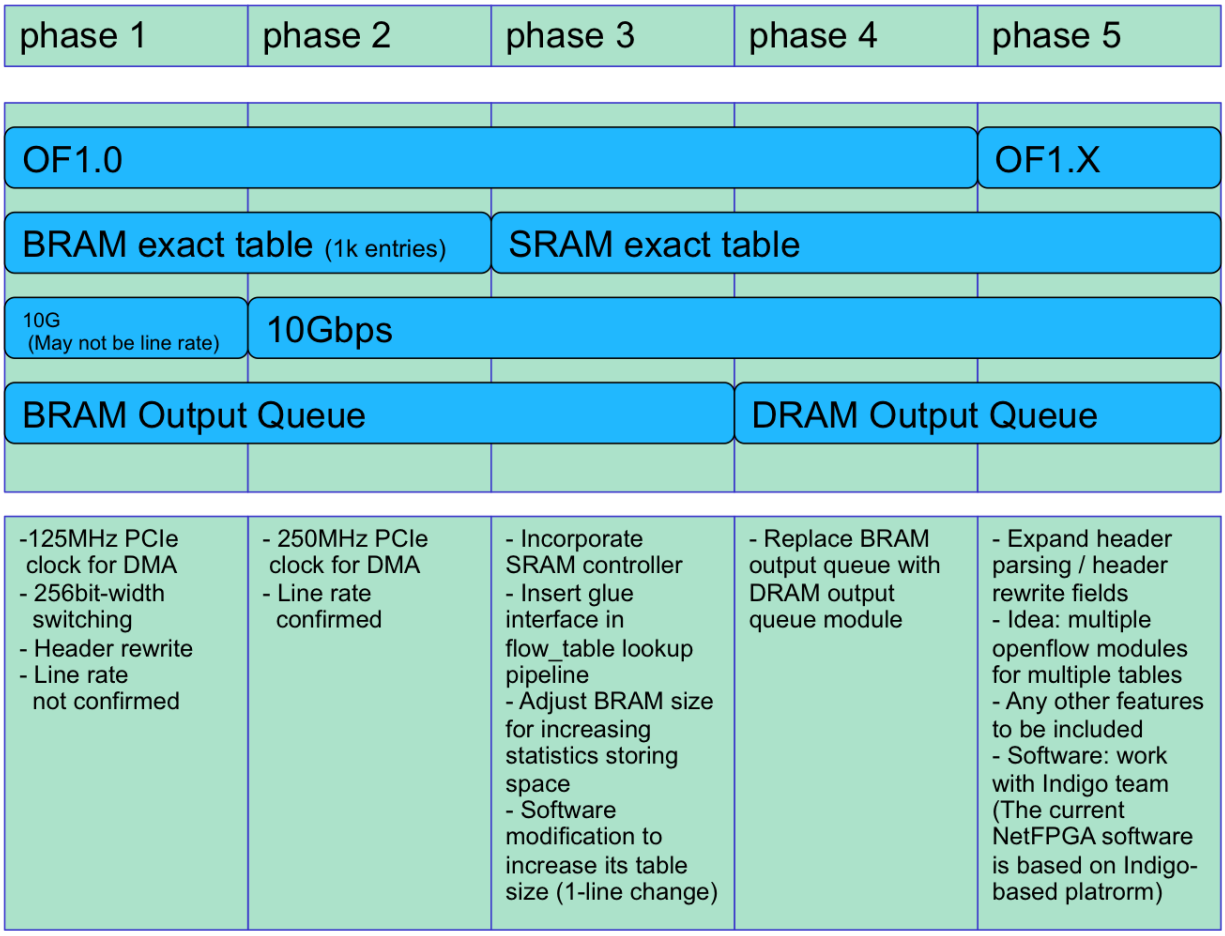


Fig1.2 Current status and the future plan

The current version/status is phase1, and it supports:

OpenFlow1.0

- w/ hardware assisted forwarding action
- Supported action: All the header rewrite actions and outport, except OFPAT_ENQUEUE.
- 32 wildcard entries --- 8 entries among them are reserved for hardware-to/from-CPU packet transfers, and practically, 24 wildcard entries are available for users.
- 1024 exact match entry points
- Hardware forwarding is theoretically line rate but not confirmed yet

- DMA transfer for packet_in and packet_out is potentially slow (clocking of DMA module is 125MHz instead of 250MHz)

Both exact match entries and wildcard match entries are stored inside BRAMs.

The later designs will/may support:
OpenFlow1.0

- Line rate confirmation
- Increasing number of entries (external SRAM for exact match)
- Supporting queuing function (w/ output queues on external DRAM)

OpenFlow1.X

Unless otherwise noted, this document will discuss phase1 of the design.

2. Comparison with NetFPGA-1G implementation

2-1. General comparison between NetFPGA-1G and NetFPGA-10G

See Table2.1 for spec comparison.

Table2.1 NetFPGA board comparison

Board	NetFPGA 1G	NetFPGA 10G
Network Interface	4 x 1Gbps Ethernet ports	4 x 10Gbps SFP+
Host Interface	PCI	PCI Express x8
FPGA	VirtexII-Pro50	Virtex5 TX240T
Logic Cells	53,136	239,616
Block RAMs	4176kbits	11664kbits
External Memories (SRAM)	4.5MB ZBT SRAM (72bitwidth, 36x2)	27MB QDRII SRAM (108bitwidth, 36x3)
External Memories (DRAM)	64MB DDR2 SDRAM (36bitwidth, 36x1)	288MB RLDRAM-II (144bitwidth, 72x2)

2-2. OpenFlow design on NetFPGA-1G

For a later comparison, Fig2.1 and Fig2.2 shows the top diagram of OpenFlow design on NetFPGA-1G. Not like NetFPGA-10G (mentioned in a later section), NetFPGA-1G implementation has eight ports (four MAC ports and four corresponding host ports) and only one data path pipeline for data from all the eight ports.

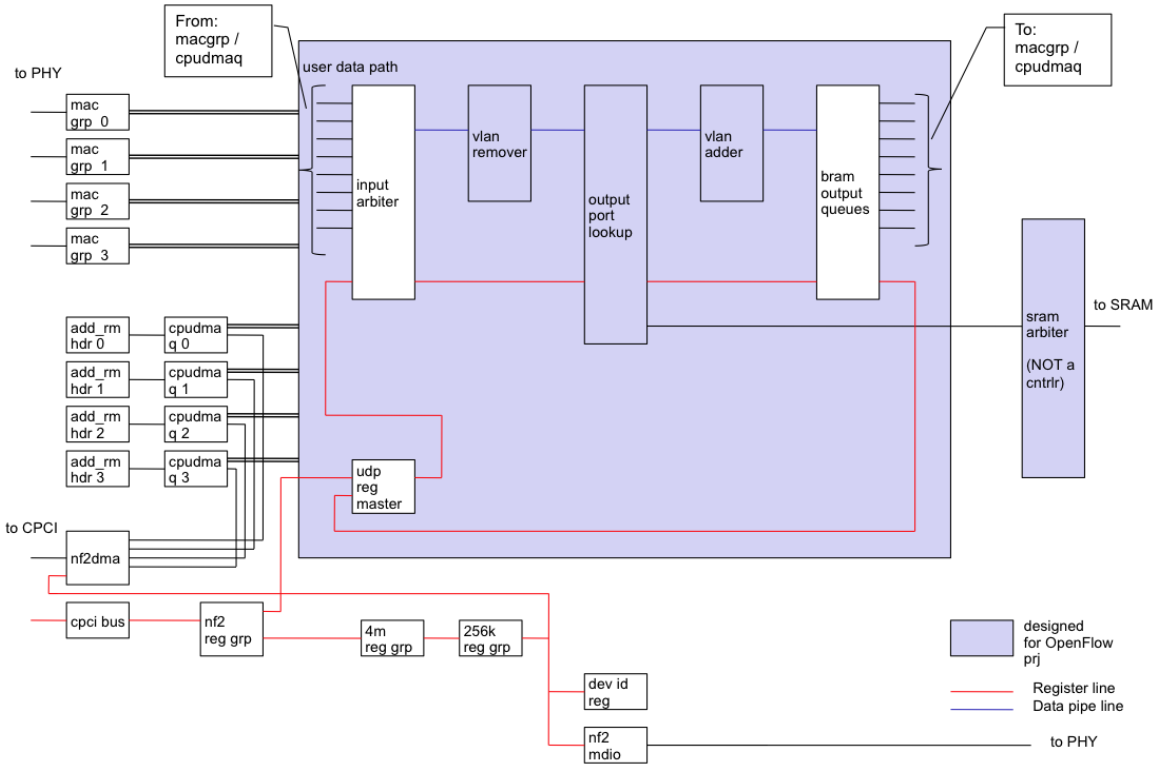


Fig2.1 OpenFlow on NetFPGA-1G

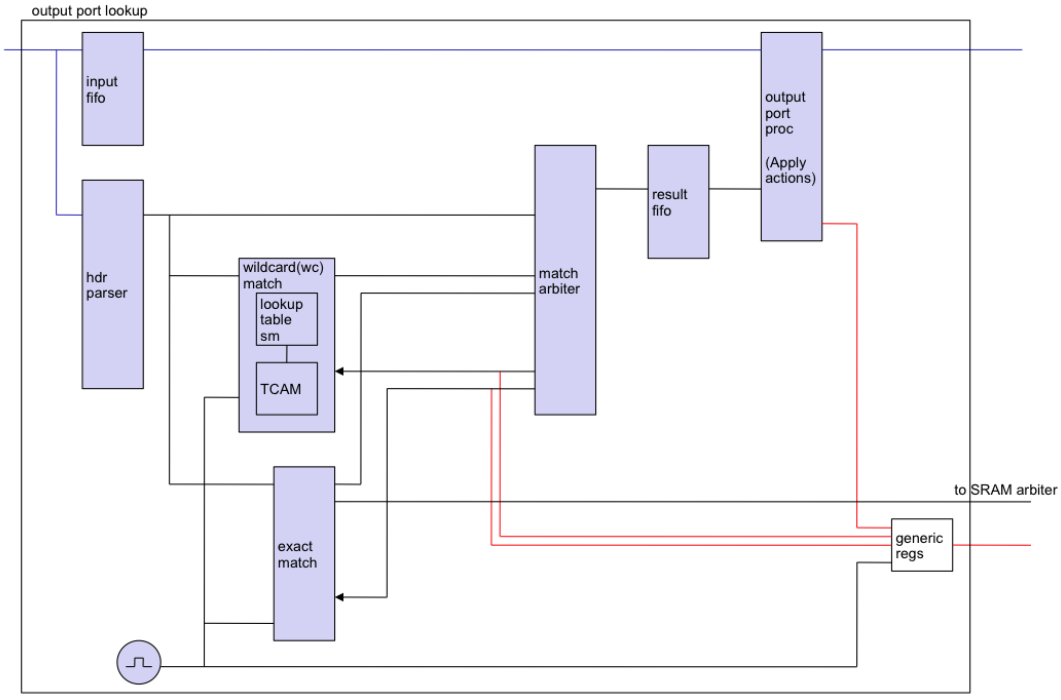


Fig2.2 OpenFlow on NetFPGA-1G (detailed)

3. Overall structure

3-1. Top level block diagram

Fig3.1 depicts the top level of OpenFlow implementation (hardware), and Fig3.2 shows how packets are switched to specified output port(s). The port-switching is done by using generic modules of input_arbiter, output_queues and AXI converters. Generic blocks out of OpenFlow pcore are abstracted and are out of scope of this document.

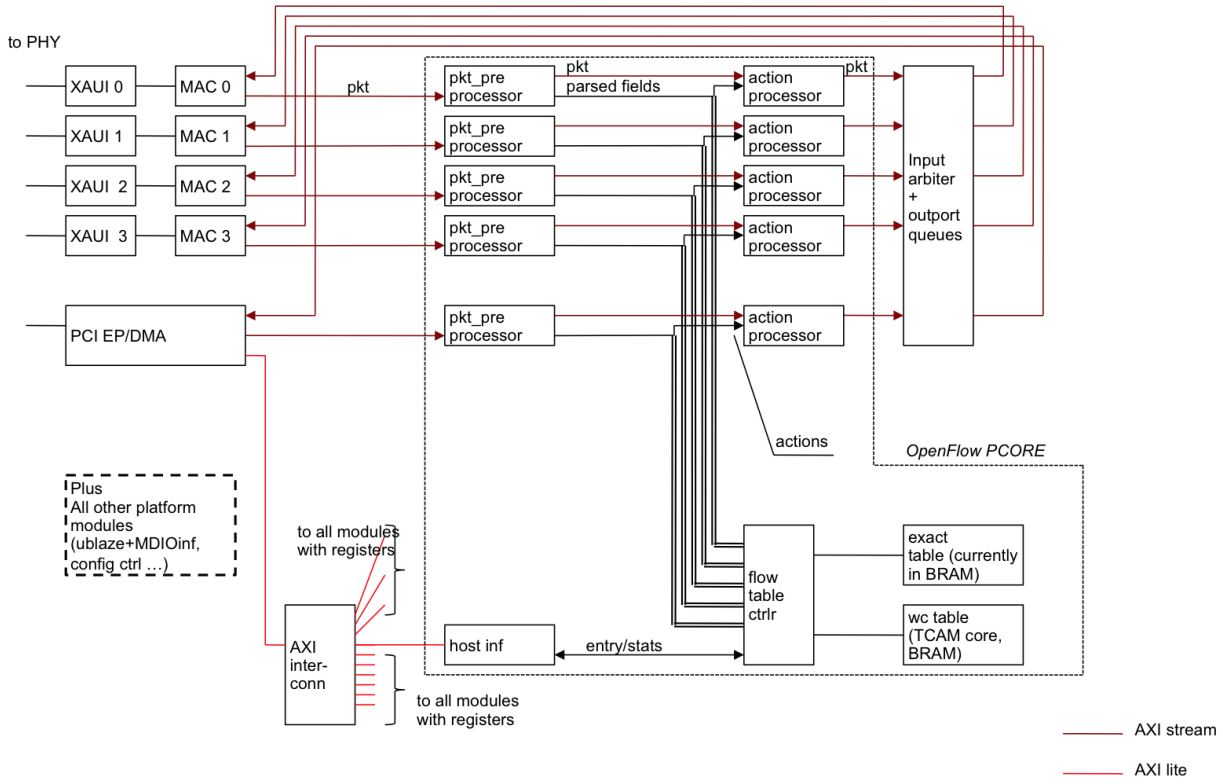


Fig3.1 Top level of OpenFlow implementation

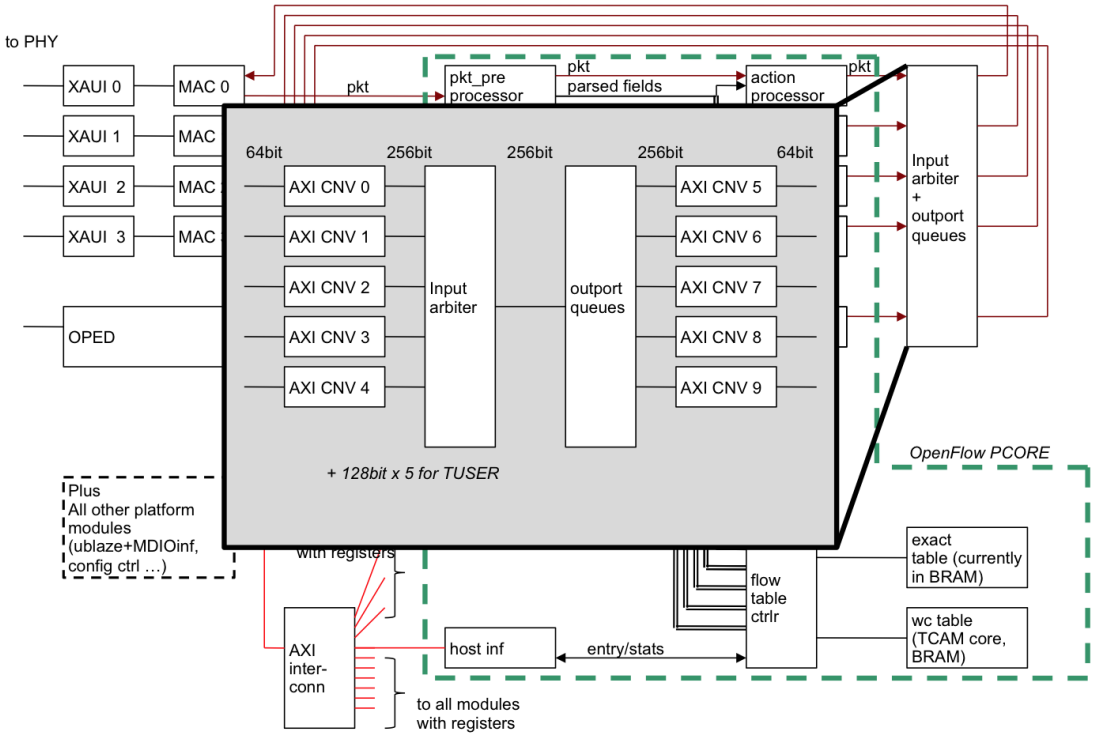


Fig3.2 Packet switching module

3-2. Modules inside OpenFlow pcore

OpenFlow pcore consists of: pkt_preprocessor, host_inf, flow_table_ctlr and action_processor. Followings are summaries of those modules:

pkt_preprocessor

Pkt_preprocessor is the first module in OpenFlow pcore for incoming packets. It parses headers of incoming packets and organizes the parsed fields so that it has as same bit order as 'match' field of a flow table entry bitmap, which is written by a host. Then pkt_preprocessor sends the bitmap to flow_table_ctrl with a handshake.
It forwards the packets themselves directly to action_processor with another handshake.

host_inf

Host_inf serves as an interface between a host and other modules inside OpenFlow pcore. In OpenFlow pcore, only host_inf module has AXI lite interface and all the necessary information is read/written via this module.

All the data in other modules which are to be read or written are wired to/from host_inf, and they can be accessed as registers in host_inf.

Flow table entries, whose actual destination is flow_table_ctlr, are also written and temporarily stored here. A host can access to a register window in this module with multiple register accesses. Once they are done, it sends a bitmap (same format as a bitmap which

pkt_preprocessor organizes) to flow_table_ctlr with a handshake. Flow entry statistics can also be read via this module.

flow_table_ctlr

Flow_table_ctlr checks if a queried flow entry bitmap (header information) from pkt_preprocessor matches any entries in its flow tables. If matched, it outputs corresponding 'actions' to queried port's action_processor. If not, it will send 'actions' with 'send_to_nowhere' information in it, so that action_prcoessor can modify the TUSER of the packet accordingly and eventually bram_output_queues can throw away the corresponding packet.

Flow table entry information is filled by a host via host_inf.

Flow_table_ctlr owns two flow tables; an exact match table and a wild card match table. In this version, both of them reside in FPGA, which has 1024 entry points and 32 entry points respectively. Future versions may use external memories such as SRAMs for tables.

action_processor

Action_processor rewrites forwarding port information in AXIS TUSER and rewrites packet header depending on 'action' list which is received from flow_table_ctlr.

Action_processor has interfaces to two preceding modules. The one is pkt_preprocessor, and action_processor receives packets from it, and the other is flow_table_ctrlr, and action_processor receives action list for the corresponding packet from it.

Action_processor is idle until it gets an action list, and once it has received the list, depending on the contents, it modifies the header of the corresponding packet and it also modifies 'dst_port' information in TUSER to the value of the port(s) to be forwarded. If the action is 'drop packet' then action_processor will clear (zero out) the 'dst_port' field in TUSER. Then finally it sends the packet out of OpenFlow pcore. The packet will be switched to the appropriate port(s) by the combination of input_arbiter module and output_queues module as Fig3.2 shows. If 'dst_port' is zero (drop packet), output_queues won't transfer the packet to anywhere, hence the packet is dropped.

3-3. Interfaces

Interface between OpenFlow pcore and others consists of five sets of AXI Stream slave/master interfaces for packet forwarding and an AXI Lite interface for host's register interface. Each of AXI Stream interfaces has 64-bit data width and 128-bit TUSER.

All the peripherals share the same clocks(axi_clk for AXI Stream and control_clk for AXI Lite) and a reset(peripheral_aresetn), where axi_clk is 160MHz and control_clk is 100MHz.

Infrastructure modules also use other clocks; PCIe and DMA modules use 250MHz (currently reduced to 125MHz) PCI clk and 10G MACs use 156MHz clock for communicating with outside signals.

3-4. Design choice

3-4-1. Block diagram

OpenFlow design has reordered the position of "input_arbiter" and removed "output_port_lookup" from NetFPGA-10G Reference NIC, instead it added "openflow_datapath" followed by "input_arbiter".

3-4-2. Multiple datapath

This design has *five* datapath pipelines with using an input_arbiter and output_queues as a switching facility in the end. Packets from each physical input port and packets from host port (aggregated) go through dedicated pipeline. Only header information bitmap will go to the centralized flow_table_control block.
Since OpenFlow design parses each field of a header and the header fields may be modified and tags may be added or subtracted, multiple short bit-width (64bit) pipelines are beneficial for executing many tasks per clock and also for successful FPGA implementation, even though reference_nic design suggests *one* long bit-width (256bit) pipeline.

3-4-3. Centralized flow_table_ctrl

Having multiple identical flow_table would be expensive in terms of memory utilization. Also the later release plans include using external memory chips (SRAMs) for flow tables. In order to control external memories, having single controller is more beneficial for maximizing their bandwidth usage.

4. Host interface

4-1. Prerequisite

As stated in Section 1, OpenFlow Switch on NetFPGA-10G consists of hardware(NetFPGA) and its associated software. So in order to make this design work, you will need a driver and a software on your host PC (See section 6 for installation).

You can use the same driver for a Stanford contributed version of reference NIC. The driver is needed for packet reception and transmission between NetFPGA and the host.
An associated software working with NetFPGA-10G talks OpenFlow protocol with a OpenFlow controller, and also manages NetFPGA's hardware flow tables (elapsed time checking/write/modify/erase).

4-2. Flow table handling overview

Associated software initializes hardware flow tables when it starts running. One of the processes is to write static entries to send packets between PHY port and CPU port as a last resort for OF-NF10G hardware.
When a packet is sent from NetFPGA to a host, it means the packet did not match against flow tables. All those packets are handled by the software. It could match the software table, as software won't set an entry to NetFPGA's flow table when its action(s) for the entry are not supported by NetFPGA. Or it could be because the number of rules has exceeded the number of entries supported by NetFPGA.
It may not exist even in a software table. In this case, in OpenFlow1.0, a software either a) drops the packet or b) sends it to the controller. The behavior is decided through a controller - switch negotiation but it is out of the scope of this document.

In order for a software to write a rule to a NetFPGA's flow table and read statistics, the software has to access to a register window in a certain way (key-hole method). See following subsections for details.
The reasons that this key-hole method has been introduced are (1) a flow table entry has more than 32 bit width and multiple software accesses are necessary. The entire bitmap needs to be updated at one time to avoid matching with intermediate incomplete entry and/or action (2) The signal interface to flow_table_ctlr must be the same as other 'pkt_preprocessor' modules.

4-3. Memory map

Apart from registers residing in NetFPGA-10G generic modules, OpenFlow pcore has a set of registers. The registers can be separated into two categories. The one is a flow entry register window mentioned in 4-2, containing match, mask, action and stats. The other is other miscellaneous registers including registers to control the flow entry register window and FPGA

level of packet forwarding statistics.

Fig4.1 shows the entire memory map which is viewable through a host. The addresses on the table are offset values. When you actually write or read, you need to add the OpenFlow pcore's module base address mentioned in system.mhs file. For instance, if OpenFlow pcore's module base address is 0x7a018000, the 'match' address will start at 0x7a018022.

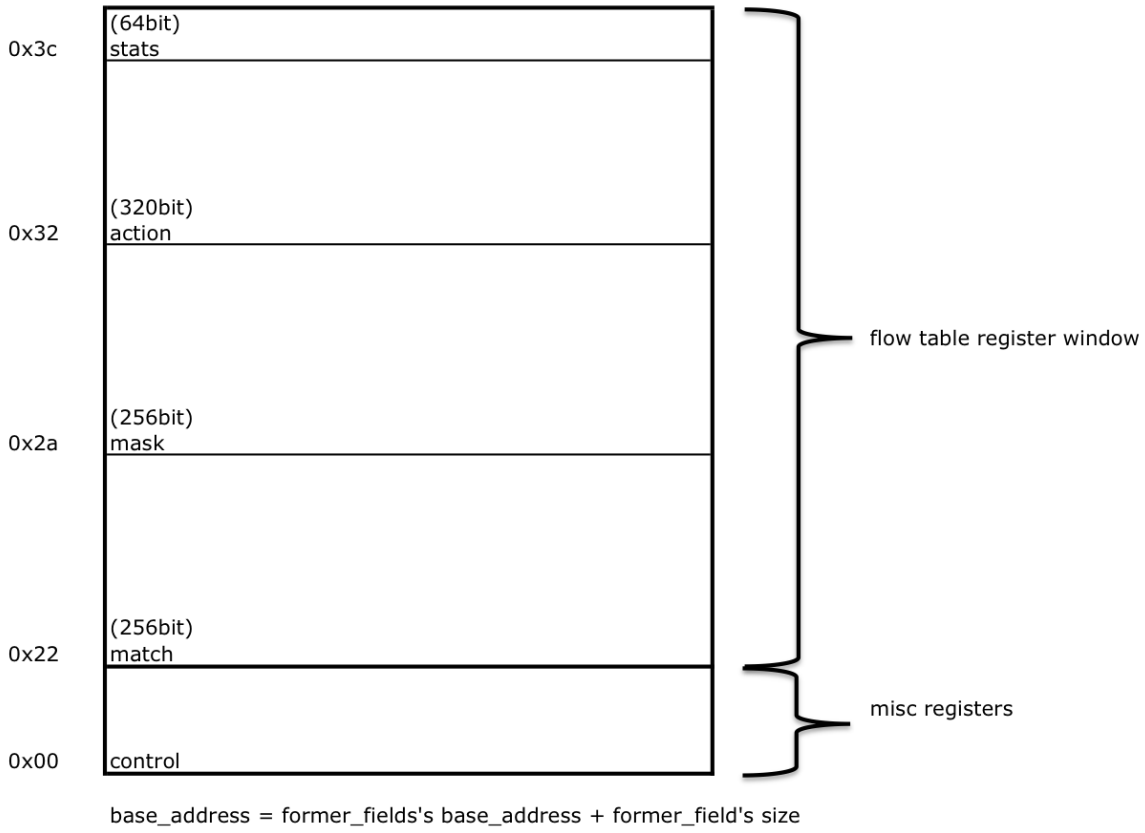


Fig4.1 Entire memory map of OpenFlow pcore

4-4. Flow table register window

Each "match", "mask", "action", "stats" in Fig4.1 is a bitmap of 256bit, 256bit, 320bit, 64bit width respectively. Fig4.2 shows how each bitmap is allocated in a set of registers.

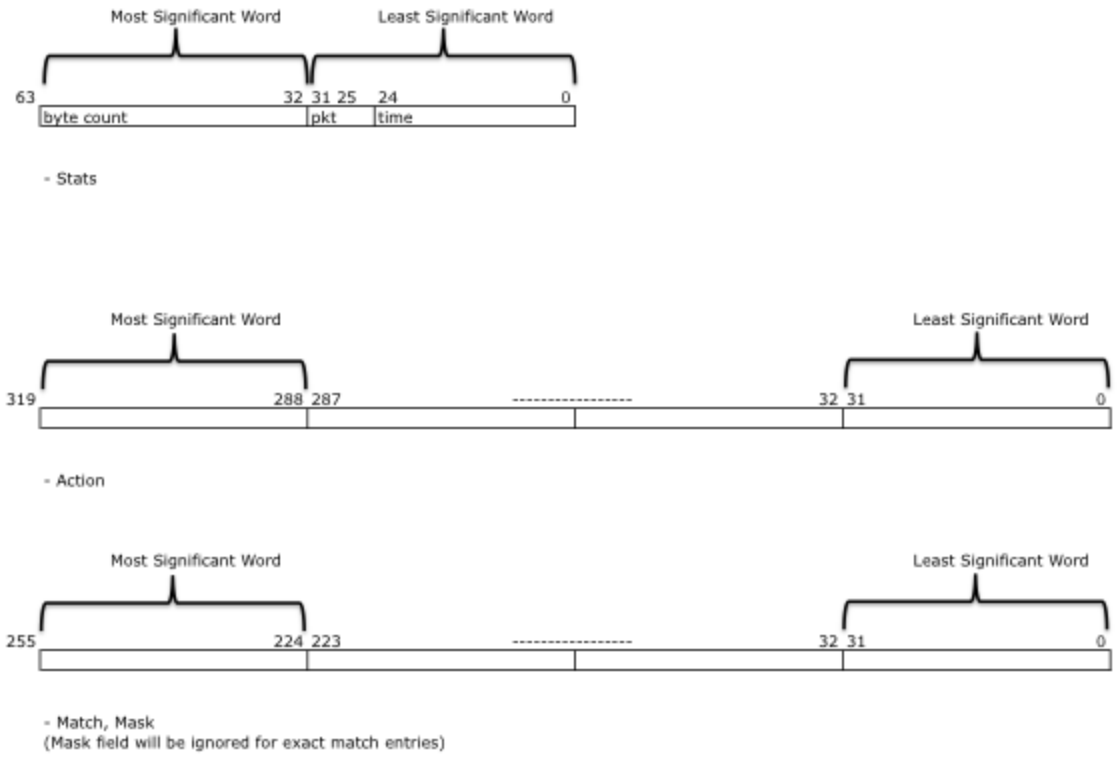


Fig4.2 Bitmap allocation

The excerpt from C header of an associated software shows how each field is allocated inside each entry. Both NetFPGA hardware and associated software have the same idea about a bit field allocation.

Match/Mask:

```
struct nf2_of_entry {
        uint16_t transp_dst;
        uint16_t transp_src;
        uint8_t ip_proto;
        uint32_t ip_dst;
        uint32_t ip_src;
        uint16_t eth_type;
        uint8_t eth_dst[6];
        uint8_t eth_src[6];
        uint8_t src_port;
        uint8_t ip_tos;
        uint16_t vlan_id;
        uint8_t pad;
```

};

Action:

```
struct nf2_of_action {
        uint16_t forward_bitmask;
        uint16_t nf2_action_flag;
        uint16_t vlan_id;
        uint8_t vlan_pcp;
        uint8_t eth_src[6];
        uint8_t eth_dst[6];
        uint32_t ip_src;
        uint32_t ip_dst;
        uint8_t ip_tos;
        uint16_t transp_src;
        uint16_t transp_dst;
        uint8_t reserved[18];
};
```

Stats:

```
struct nf2_of_counters {
        uint32_t pkt_count:25;
        uint8_t last_seen:7;
        uint32_t byte_count;
};
```

For each struct, the first field is put on Least Significant word (See Fig4.2).

Fig4.3 shows how src_port and forward_bitmask are assigned. The upper 8bits of forward_bitmask is currently not used.

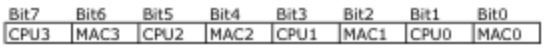


Fig4.3 Src_port and forward_bitmask assignment

Fig4.4 shows the bitmap of nf2_action_flag field. Each bit is assigned to an OpenFlow action, and if the value for a field is one(1), it means NetFPGA hardware is expected to perform this action.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0: OFPAT_OUTPUT
1: OFPAT_SET_VLAN_VID
2: OFPAT_SET_VLAN_PCP
3: OFPAT_POP_VLAN (OFPAT_STRIP_VLAN in 1.0)
4: OFPAT_SET_DL_SRC
5: OFPAT_SET_DL_DST
6: OFPAT_SET_NW_SRC
7: OFPAT_SET_NW_DST
8: OFPAT_SET_NW_TOS
9: OFPAT_SET_TP_SRC
10: OFPAT_SET_TP_DST
11: OFPAT_SET_NW_ECN (Feature in 1.1 and beyond (*))
12: OFPAT_PUSH_VLAN (OFPAT_ADD_VLAN in 1.0)
13: OFPAT_SET_NW_TTL (Feature in 1.1 and beyond (*))
14: OFPAT_DEC_NW_TTL (Feature in 1.1 and beyond (*))
15: reserved

(*) OF1.1+ actions are written but not simulated or physically tested

Fig4.4 nf2_action_flag

4-5. Miscellaneous registers

Fig4.4 shows the details of “control” registers.
As stated in section 4-3, the addresses on the list are offset values. When you actually write or read, you need to add the OpenFlow pcore’s module base address mentioned in system.mhs file. For instance, if OpenFlow pcore’s module base address is 0x7a018000, the ‘read_order’ address will start at 0x7a018020.

address	register name	Description	access type
0x21	read_order	Flow Table Access Control Register. See Flow Chart	write
0x20	mod_write_order	Flow Table Access Control Register. See Flow Chart	write
0x1f	wirte_order	Flow Table Access Control Register. See Flow Chart	write
0x1e	base_addr	Flow Table Access Control Register. See Flow Chart	write/read
0x1d	acc_rdy	Flow Table Access Control Register. See Flow Chart	read
0x1c	ip_tp_parse_cnt_4	Num of pkts parsed as a IP, TCP/IP or UDP/IP on CPU input port	read
0x1b	ip_tp_parse_cnt_3	Num of pkts parsed as a IP, TCP/IP or UDP/IP on Phy input port 3 (nf3)	read
0x1a	ip_tp_parse_cnt_2	Num of pkts parsed as a IP, TCP/IP or UDP/IP on Phy input port 2 (nf2)	read
0x19	ip_tp_parse_cnt_1	Num of pkts parsed as a IP, TCP/IP or UDP/IP on Phy input port 1 (nf1)	read
0x18	ip_tp_parse_cnt_0	Num of pkts parsed as a IP, TCP/IP or UDP/IP on Phy input port 0 (nf0)	read
0x17	arp_parse_cnt_4	Num of pkts parsed as an ARP on CPU input port	read
0x16	arp_parse_cnt_3	Num of pkts parsed as an ARP on Phy input port 3 (nf3)	read
0x15	arp_parse_cnt_2	Num of pkts parsed as an ARP on Phy input port 2 (nf2)	read
0x14	arp_parse_cnt_1	Num of pkts parsed as an ARP on Phy input port 1 (nf1)	read
0x13	arp_parse_cnt_0	Num of pkts parsed as an ARP on Phy input port 0 (nf0)	read
0x12	num_proc_done_4	Num of pkts processed and forwarded out of OpenFlow module (CPU input port)	read
0x11	num_proc_done_3	Num of pkts processed and forwarded out of OpenFlow module (input port 3)	read
0x10	num_proc_done_2	Num of pkts processed and forwarded out of OpenFlow module (input port 2)	read
0x0f	num_proc_done_1	Num of pkts processed and forwarded out of OpenFlow module (input port 1)	read
0x0e	num_proc_done_0	Num of pkts processed and forwarded out of OpenFlow module (input port 0)	read
0x0d	dl_parse_cnt_4	Num of pkts parsed as an L2 frame on CPU input port	read
0x0c	dl_parse_cnt_3	Num of pkts parsed as an L2 frame on Phy input port 3 (nf3)	read
0x0b	dl_parse_cnt_2	Num of pkts parsed as an L2 frame on Phy input port 2 (nf2)	read
0x0a	dl_parse_cnt_1	Num of pkts parsed as an L2 frame on Phy input port 1 (nf1)	read
0x09	dl_parse_cnt_0	Num of pkts parsed as an L2 frame on Phy input port 0 (nf0)	read
0x08	wildcard_misses	Num of pkts NOT matching any wildcard entries	read
0x07	wildcard_hits	Num of pkts matching a wildcard entry	read
0x06	exact_misses	Num of pkts NOT matching any exact match entries	read
0x05	exact_hits	Num of pkts matching an exact match entry	read
0x04	num_pkts_dropped_4	Num of pkts from CPU input port dropped in action_processor	read
0x03	num_pkts_dropped_3	Num of pkts from Phy input port 3 (nf3) dropped in action_processor	read
0x02	num_pkts_dropped_2	Num of pkts from Phy input port 2 (nf2) dropped in action_processor	read
0x01	num_pkts_dropped_1	Num of pkts from Phy input port 1 (nf1) dropped in action_processor	read
0x00	num_pkts_dropped_0	Num of pkts from Phy input port 0 (nf0) dropped in action_processor	read

Fig4.4 Register map

acc_rdy (read only)

1: “The block is ready to access to flow_table.”
If after read_order, it means that the data is available for read

0: “Don’t access to flow_table.”
If after read_order, it means that the data is not yet available for read

base_addr (read/write)

Address(row) to write an entry to, or read a stats from.
See memory map for detail.

write_order (write)

Writing any value to this address means 'Write the entry to flow_table'.
It includes statistics values and it is used for writing a new entry.
Once it is set, acc_rdy bit will turn to zero until writing process has been finished.

mod_write_order (write)

Writing any value to this address means 'Write the entry to flow_table
but don't overwrite statistics'.
It is used for modifying the existing entry. Statistics will be preserved.
Once it is set, acc_rdy bit will turn to zero until writing process has been finished.

read_order (write)

Writing any value to this address means 'start reading data from specified base_address'.
Once it is set, acc_rdy bit will turn to zero until reading process has been finished.
Wait until acc_rdy goes one before start reading the data(stats).

4-6. Flow entry setting sequence

Fig4.5 shows the idea of base addresses of exact_match table and wildcard_match table. Note that the current design has 1024 exact match entries and 32 wildcard match entries and only those areas are valid.

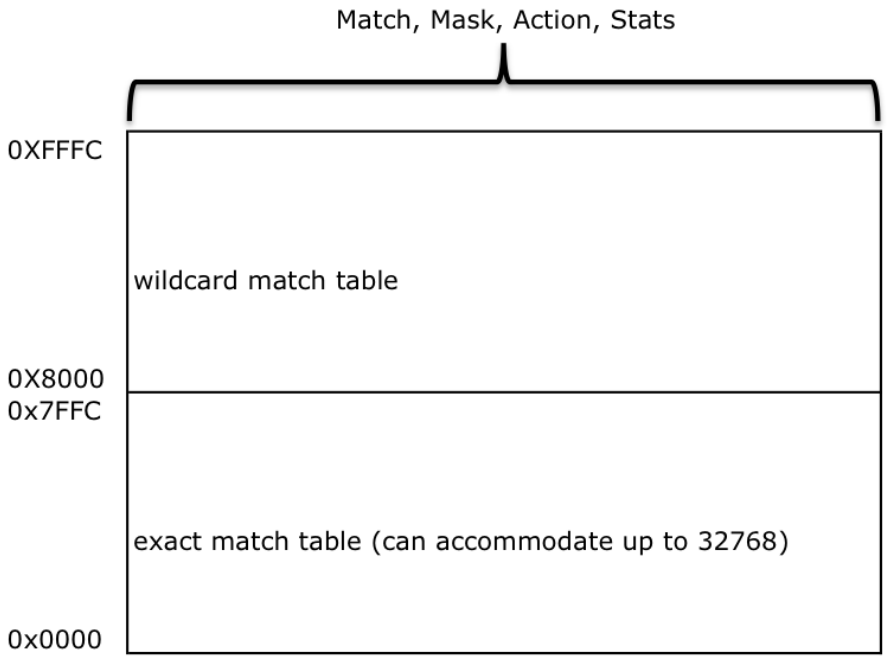


Fig4.5 Memory map for flow tables

In order to set a flow entry, a series of read and write accesses to registers is needed. See a flow chart on Fig4.6 for writing a flow entry (match, mask, action, stats) to a flow table, and see another flow chart on Fig4.7 for reading statistics from flow chart.

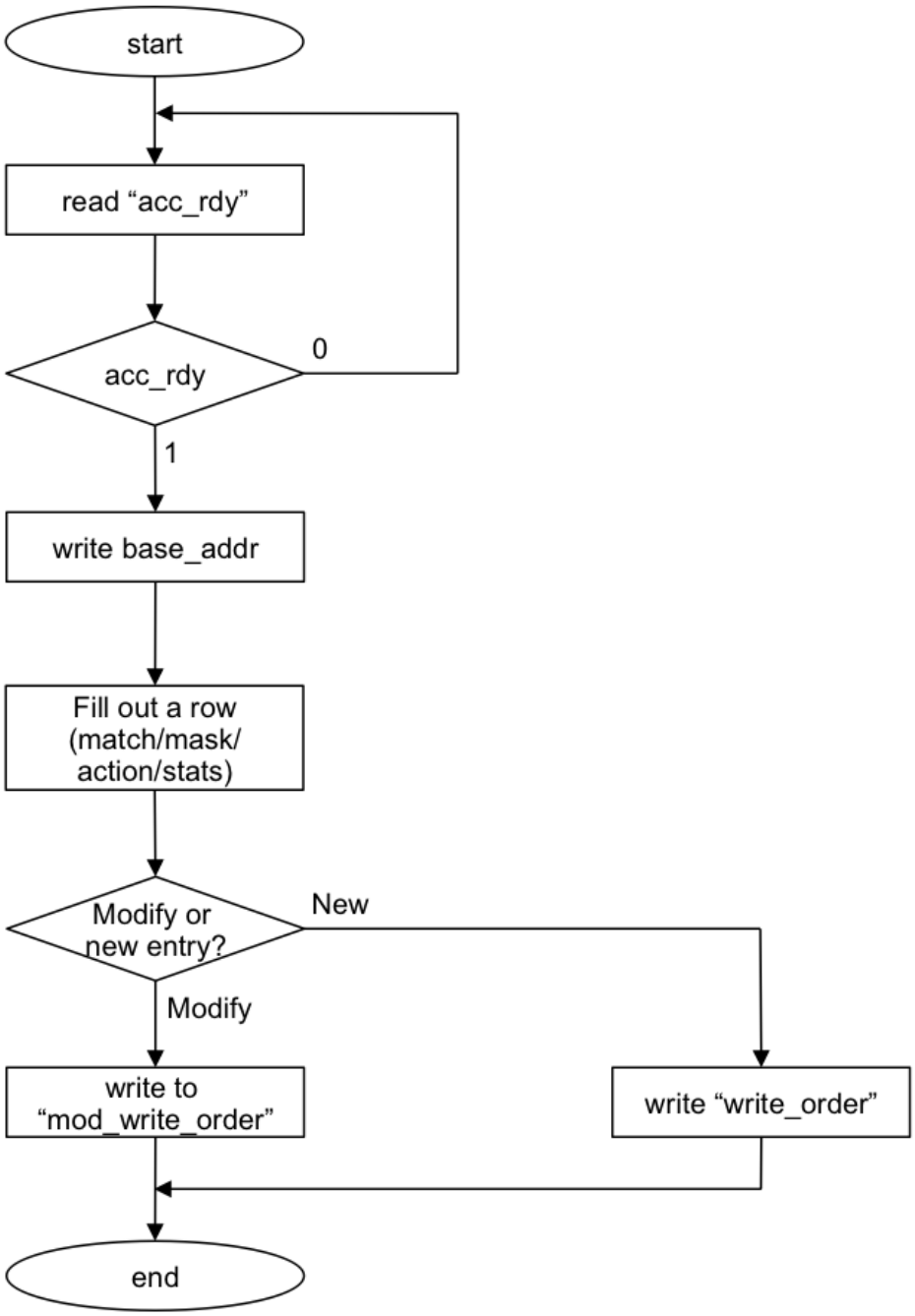


Fig4.6 Writing to a flow table

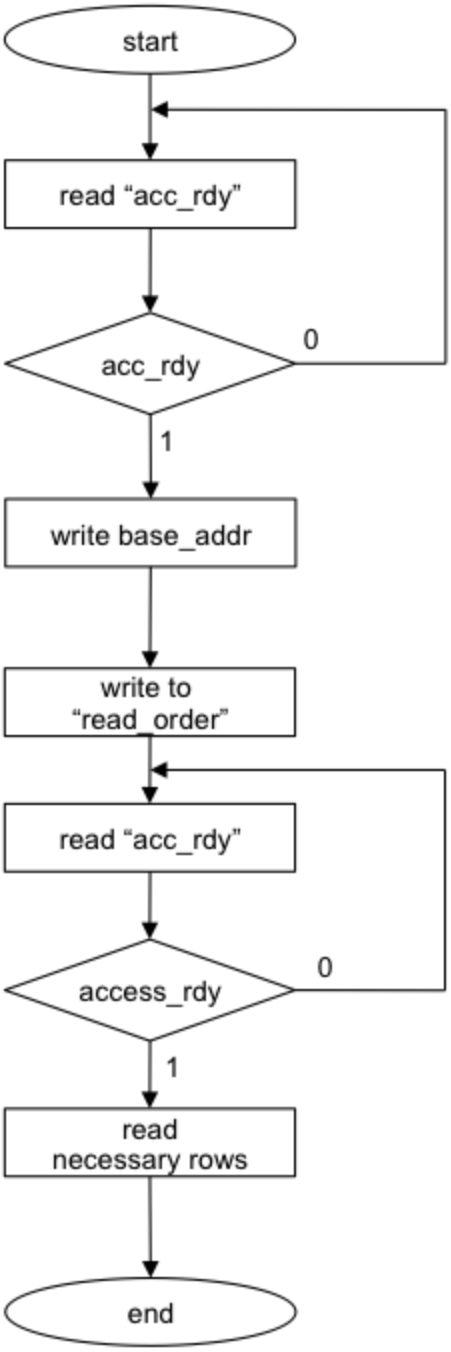


Fig4.7 Reading from a flow table

Note:
- Acc_rdy should be active within 1us after any physical access to a flow table.
- base_addr: one of the addresses in Fig4.5

5. Hardware modules inside OpenFlow pcore

This section describes the details of each module inside OpenFlow pcore.

5-1. pkt_preprocessor

5-1-1. Summary

This module parses packet headers and organizes them as a form of flow_entry bitmap, as a 'flow' is defined by its header. This module consists of packet buffers (input_fifo, output_pkt_buf), header parser module (header_parser) and lookup entry composer module (lu_entry_composer).

5-1-2. Block diagram

Fig5.1.1 shows a block diagram of the module (mpls_parser is coded but disabled and not simulated or tested).

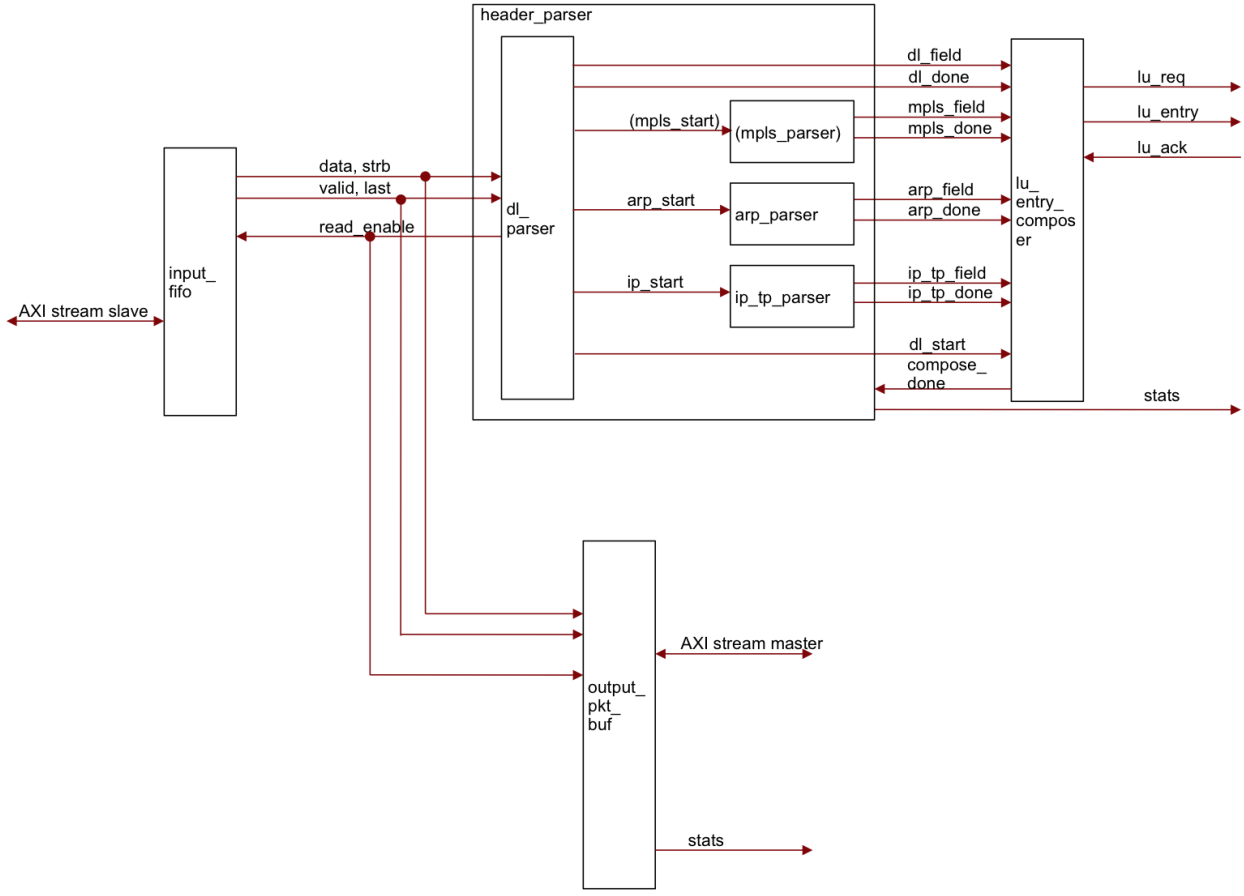


Fig5.1.1 pkt_preprocessor block diagram

An incoming packet as AXI Stream firstly goes to input_fifo; a set of buffers for data/tuser. The

buffered packet then goes to both header_parser and output_pkt_buf.
In the current version supporting OpenFlow1.0, the output from header_parser can be either:

- Ethernet w/wo VLAN tag(s) : It only parses the outermost tag
- IP
    - ICMP
    - TCP/UDP
    - other
- ARP
- Other

Once the parsing has been finished, lu_entry_composer organizes the result from header_parser and creates a flow_entry bitmap. Then it generates a 'request' with the bitmap to the following module (flow_table_ctlr). It waits for acknowledgement from flow_table_ctrl and once acknowledge is received, it clears out a 'request' and the bitmap.

Output_pkt_buf is used as a delay generator while it is waiting for a request signal to be generated. The expectation of this submodule is that the following module (action_processor) starts reading data from output_pkt_buf until the end of the packet (TLAST).

5-1-3. Interfaces

(1) AXI stream (slave)

- Bitwidth 64bit (fixed)
- TUSER 128bit

(2) Proprietary signals (between internal modules)

- lu_fields (out):
    - Flow entry for matching against flow table
    - The bitmap is organized by lu_entry_composer
    - Latched when a request to flow_table_ctrl (lu_req) is issued
    - Released when an acknowledgement from flow_table_ctrl (lu_ack) is received
- lu_req (out):
    - Lookup request generated by lu_entry_composer to flow_table_ctlr
    - Latched (staying 'high') until an acknowledgement from flow_table_ctlr (lu_ack) is received
    - Action_processor will also use the signal to start fetching packet itself (AXI stream data) out of output_pkt_buf
- lu_ack (in):
    - Generated by flow_table_ctlr module
    - When asserted, it means the table lookup request is 'accepted' (but not finished)
    - Used for releasing lu_req and lu_fields(flow_entry bitmap)
- Statistical information (out):
    - dl_parsed (incremented when parse is finished with l2 header)

    - mpls_parsed (incremented when parse is finished with mpls tag FOR FUTURE USE)
    - arp_parsed (incremented when parse is finished with arp header)
    - ip_tp_parsed (incremented when parse is finished with ip, icmp, udp, tcp, sctp header)

  All the counters above will roll over when it has reached the maximum value (0xFFFFFFFF).
- counter_reset (in): (THIS INTERFACE IS NOT YET IMPLEMENTED)
    - Reset all the counters

(3) AXI stream (master)
- Bitwidth 64bit (fixed)
- TUSER 128bit

5-1-4. Details

(1) input_fifo

Input_fifo is a common interface for many of NetFPGA pcores. it is used for reducing back pressure situation.
Input_fifo is filled when AXI TVALID is active and the fifo is not full. It has a minimum depth to reduce latency. Two input_fifos are implemented for packet itself and for TUSER which arrives at the first clock cycle of each packet. This module will latch it at the first clock for the latter module.

(2) header_parser

Fig5.1.2 shows bit positions for parsing each type of packet.

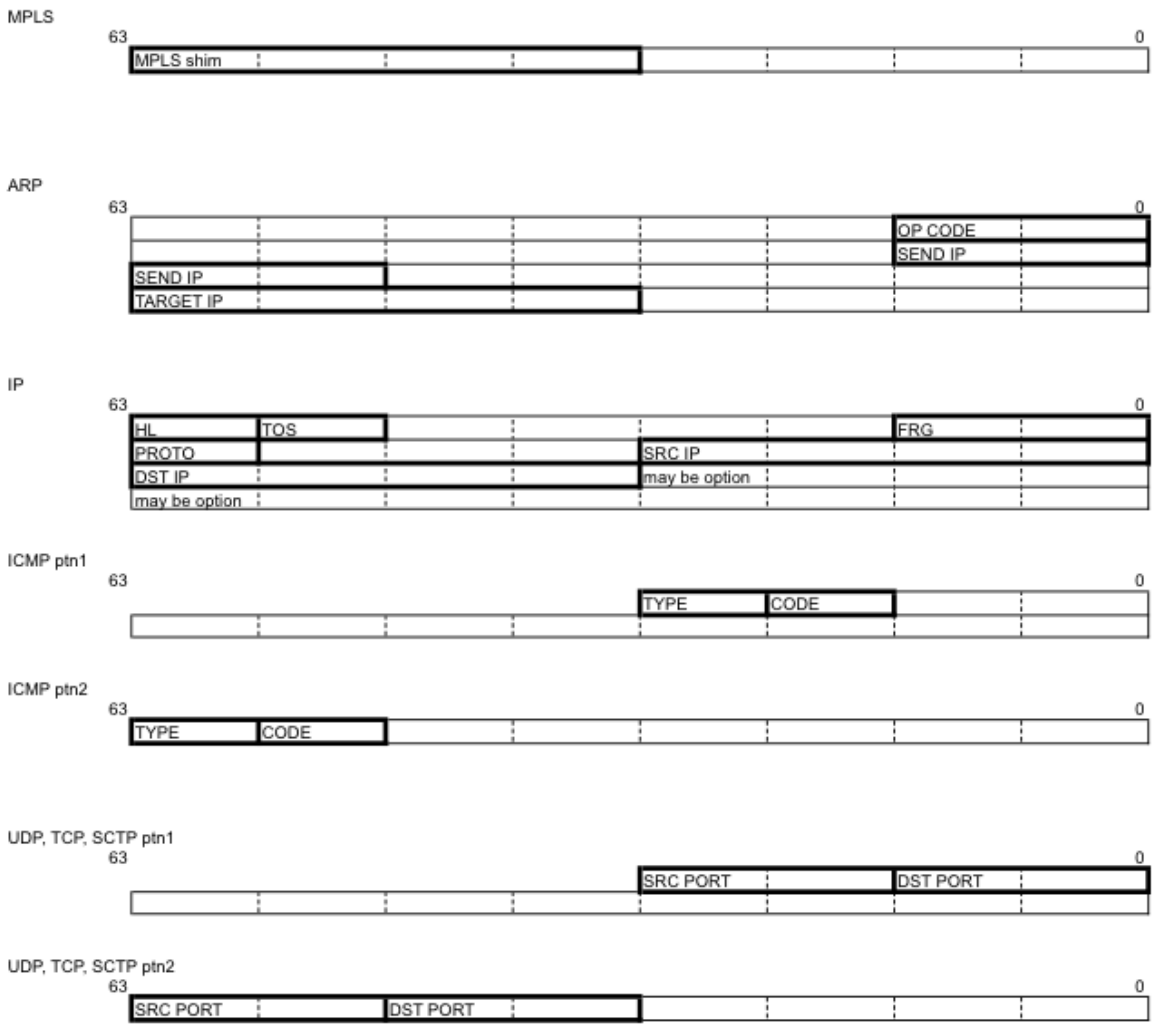


Fig5.1.2 Header fields to be parsed

Each parsing block has a state machine. Fig5.1.3 shows L2 parser state machine. This first state machine starts running when TVALID is asserted by a preceding module, and it goes back to the idle state when TLAST is received.

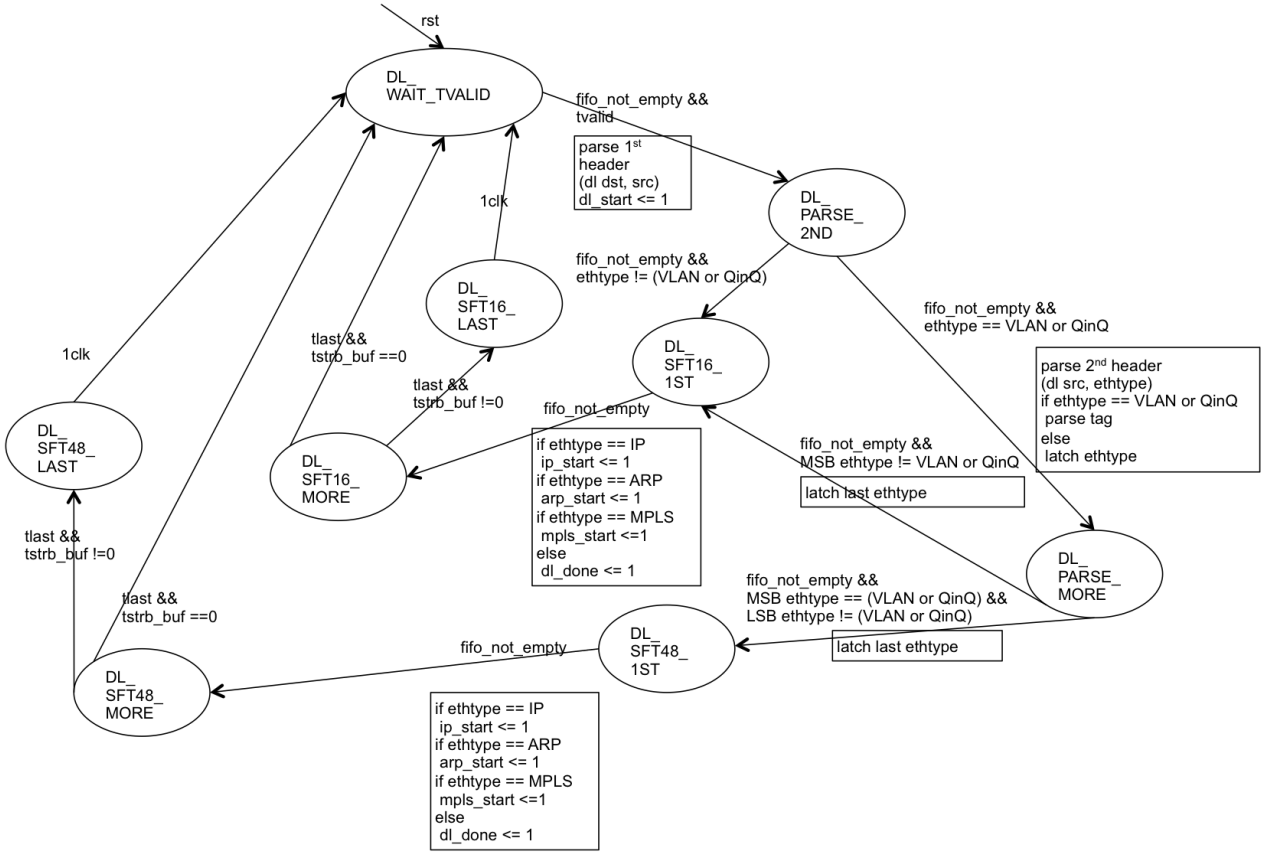


Fig5.1.3 dl_parser state machine

The following state machines start running when dl_parse (L2 parser, previous state machine) asserts 'done/start' pulse (1clk). They go back to the idle state when compose_done signal is received from lu_entry_composer sub module.

Fig5.1.4 shows mpls_parser state machine. This is for future use.

Fig5.1.5 shows arp_parser state machine.

Fig5.1.6 shows ip_tp_parser state machine.

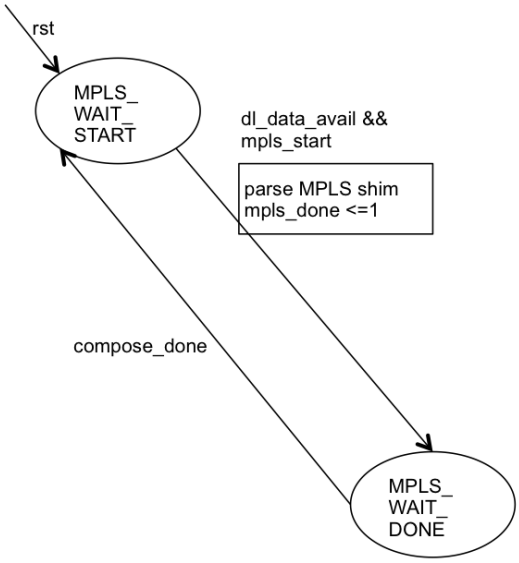


Fig5.1.4 mpls_parser state machine

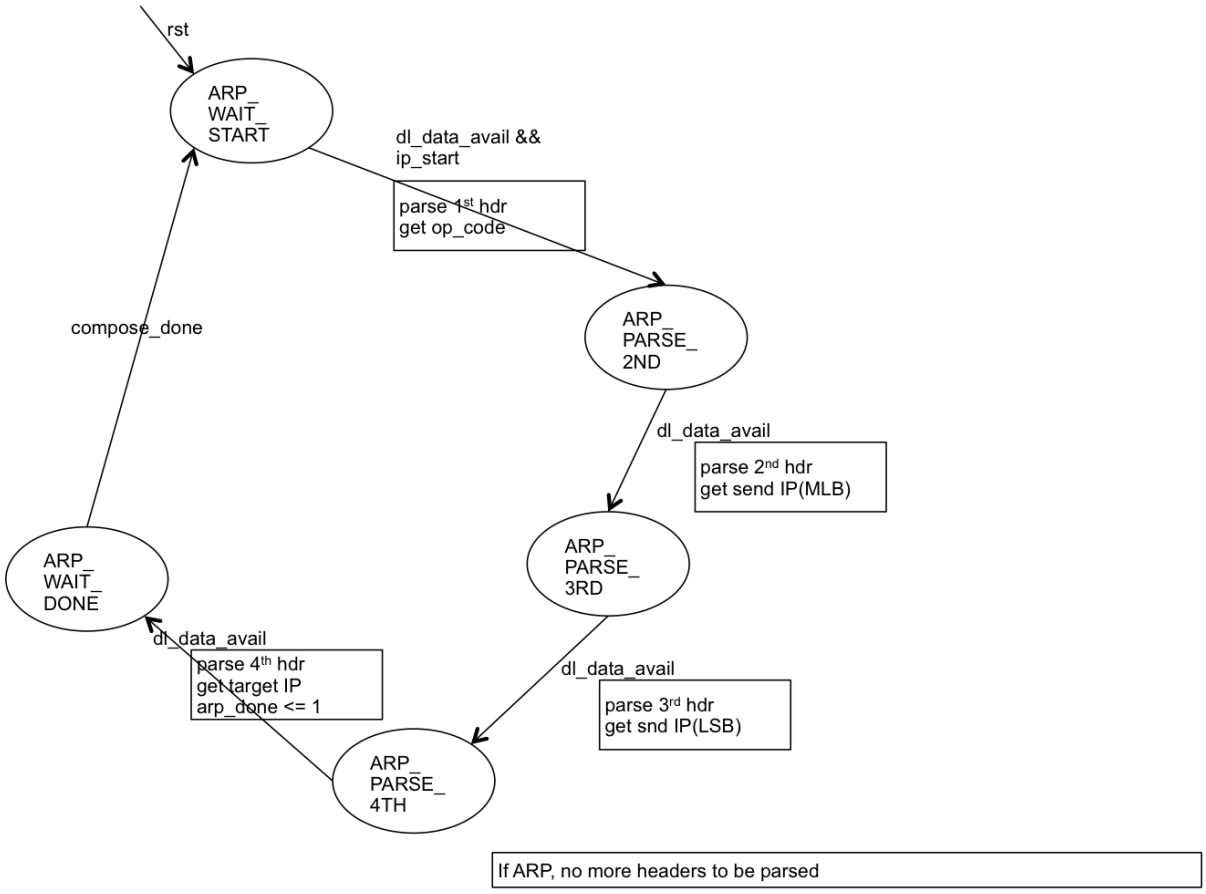


Fig5.1.5 arp_parser state machine

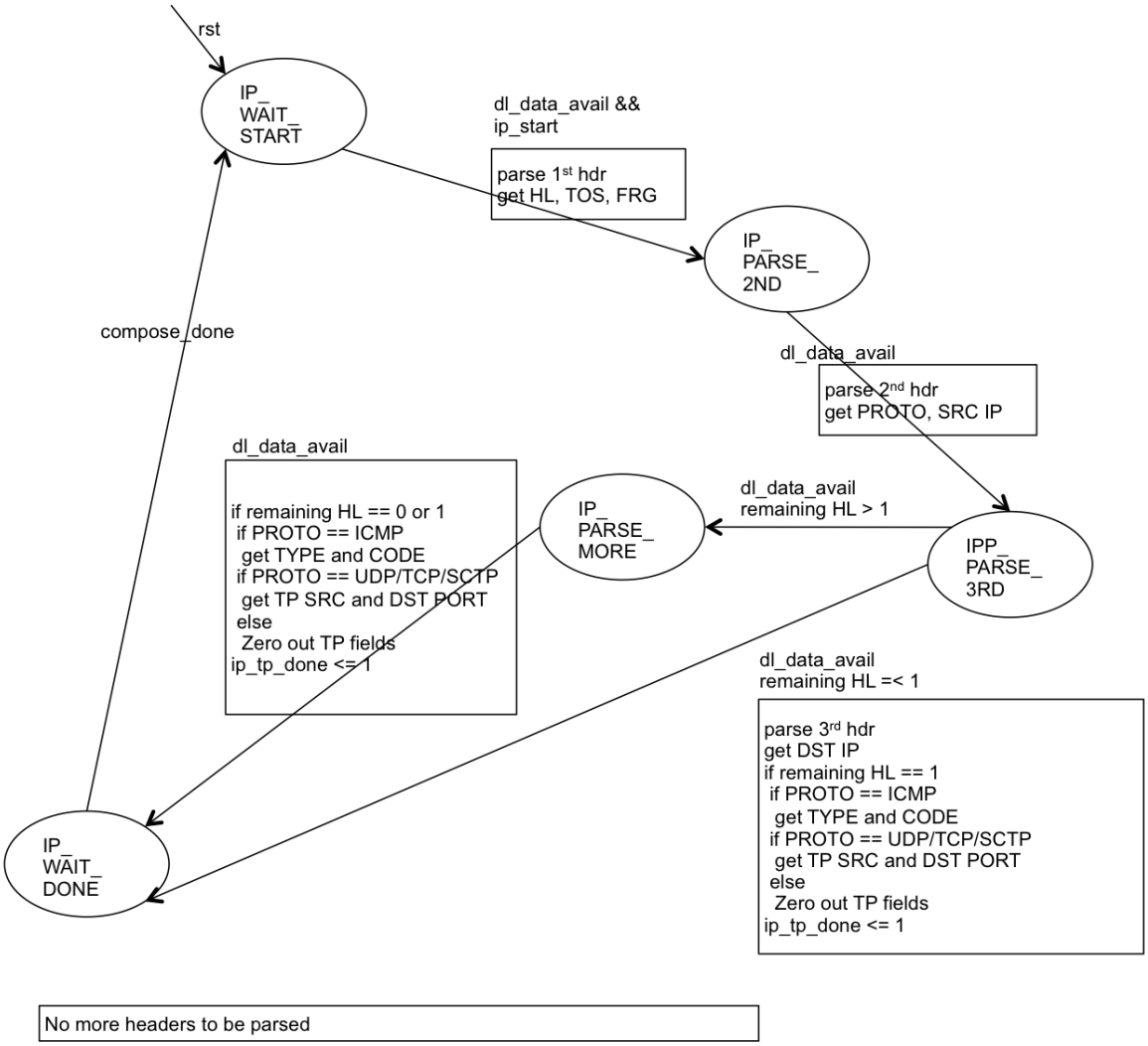


Fig5.1.6 ip_tp_parser state machine

As a design choice, parsing processes are separated into multiple small state machines. Main benefit is it can handle multiple unknown numbers of VLAN tags and/or IP header options. You can discard any numbers of tags in dl_parse state machine, then nw_tp_parse or any other state machines can always consider that the packet starts at the same bit position. Also a packet can be IP, ARP or raw ethernet and the fields to be parsed are different depending on the protocol.

(3) lu_entry_composer (lookup entry composer)

Lu_entry_composer composes the parsed fields into a specific format(flow_entry bitmap). Lu_entry_composer consists of three portions:
- Interface with preceding parsers
- Compose match field

- Interface with following flow_table_ctrl module
 Fig5.1.7 shows how each state machine works.

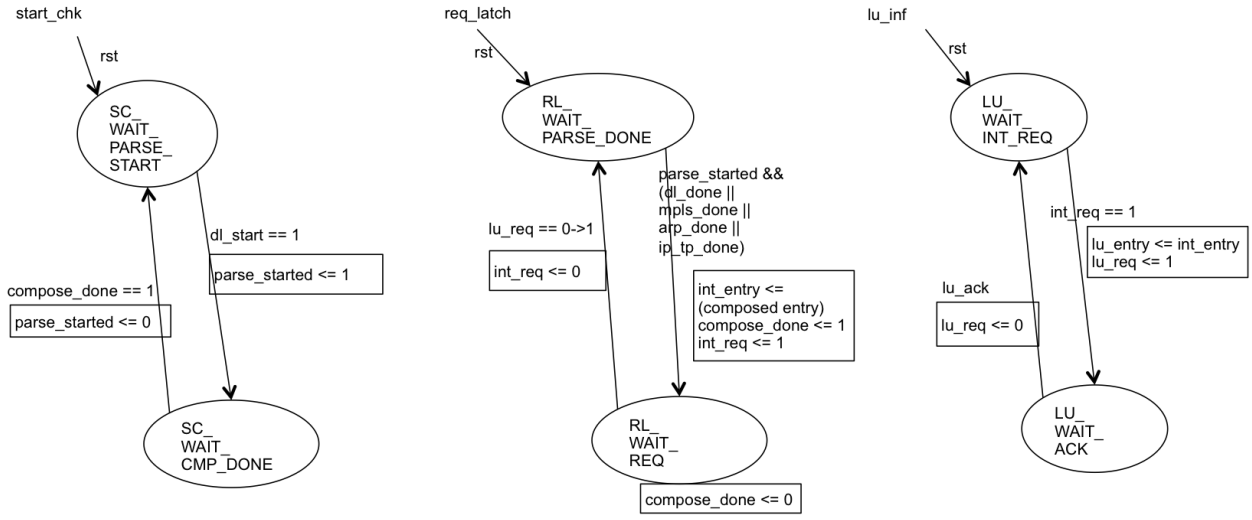


Fig5.1.7 lu_entry_processor state machines

(4) output_pkt_buf

Output_pkt_buf is used for syncing parsed header information and the packet itself. The depth of the buffer, 128byte (variable with a static parameter), is sufficient to store data until it finishes header parsing.
It is expected to be read by the following module when header_parser asserts 'lu_req'.
The necessary time period varies from pkt to pkt. Some may have multiple tags and IP options followed by TCP, and some may be ethernet frame with raw data.
With this depth, it can support UDP/TCP packet with two VLAN tags and 8 words of IP options.
If it is not enough, it emits 'overflow' error, but an error handling is not implemented yet.

5-2. host_inf

5-2-1. Summary

Host_inf is an interface point for a host to access to modules inside OpenFlow pcore. It is used for writing flow_entry, reading flow_entry statistics and accessing to other registers in OpenFlow pcore.
Host_inf buffers all the necessary information for one entry when a host writes to the ‘flow_table window’ in host_inf (see section 4) word by word. Once a host gives a trigger to host_inf, host_inf flushes the information (flow_entry) to flow_table_ctlr with the same handshake protocol as all other pkt_preprocessors and flow_table_ctlr.

5-2-2. Block diagram

Fig5.2.1 shows a block diagram of the module. The left-hand side (AXI_lite_write, AXI_lite_read) interfaces with a host via AXI_interconnect module, and the right-hand side interfaces with flow_table_ctlr and other modules in OpenFlow pcore. See following sections for details.

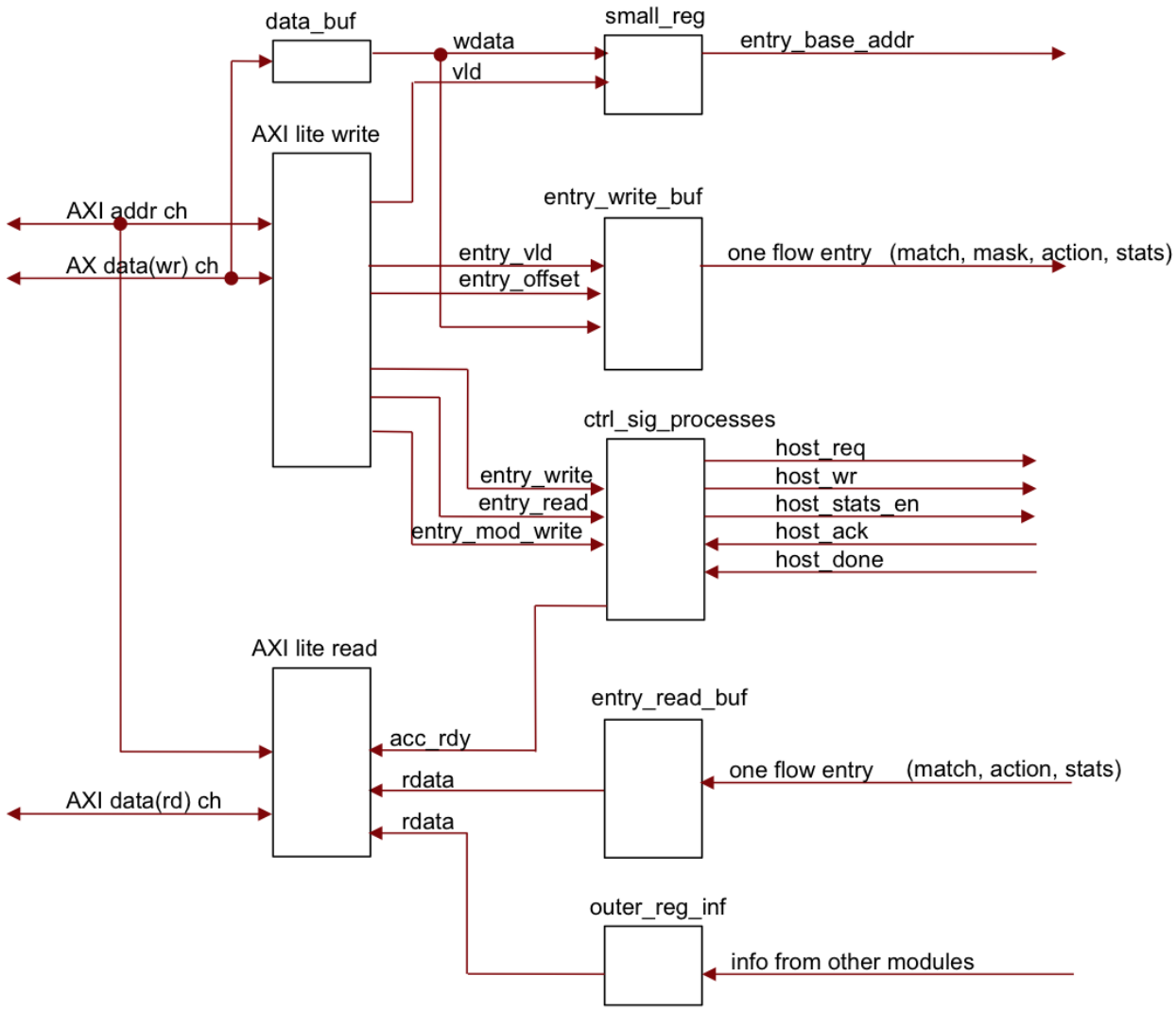


Fig5.2.1 host_inf block diagram

5-2-3. Interfaces

(1) AXI Lite (between AXI_interconnect module)

- AXI4lite address channel
- AXI4lite data channel (for input)
- AXI4lite data channel (for output)

(2) Proprietary signals

- host_req: (out)
    - Request for read/write from/to flow_table_ctrl
- host_ack: (in)
    - Asserted when request is acknowledged
- host_done: (in)
    - Asserted when write/read access is completed.
    - When the signal is for read, it means the data is ready to be read.
- host_wr: (out)
    - Indicates write access to flow_table_ctlr
- host_stats_en: (out)
    - Indicates the write/read access includes statistics portion of the entry
- host_rd: (out)
    - Indicates read access from flow_table_ctlr
- entry_base_address: (out)
    - Indicates a table ‘row’ to be accessed. See section 4 for the value for this signal bus
- flow_entry(match/mask/action/stats) : (out)
- flow_entry(stats): (in)
- registers residing other modules:
    - See section 4 for detail

(3) Clocks

Two clocks are used:

- Clock for AXI lite
    - Main clock in this module
- Clock for AXI stream
    - Used for interface with flow_table_ctrl

5-2-4. Details

(1) AXI_lite_write

AXI_lite_write portion is one of the interfaces with AXI interconnect, and it is responsible for getting data from a host. Fig5.2.2 shows a state machine of this portion.

- The implementation of this portion has been copied and modified from mdio_gen platform-block.
- AXI_lite clock domain

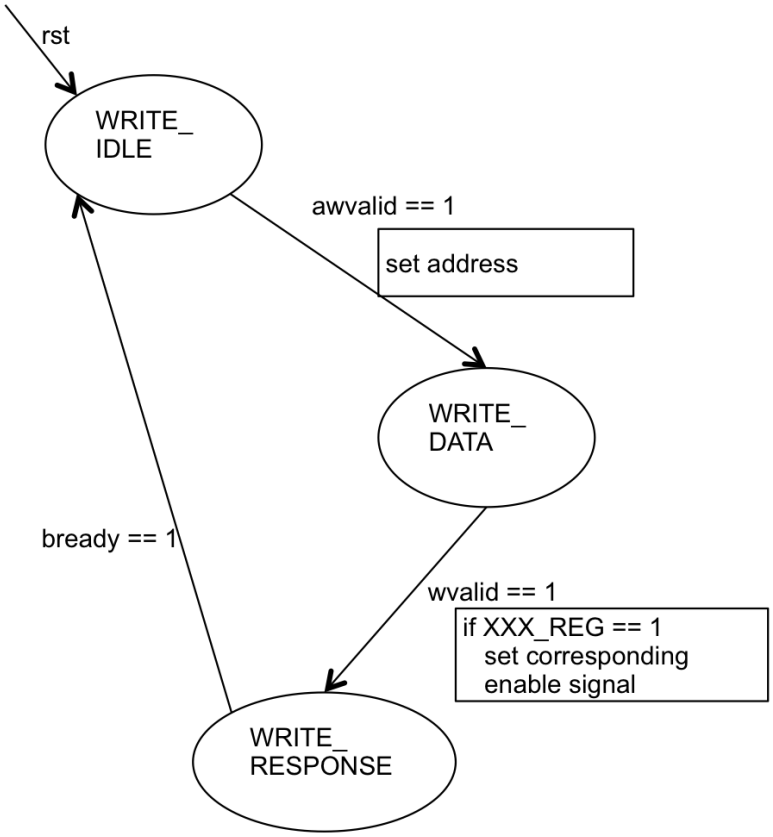


Fig5.2.2 AXI_lite_write State machine

(2) AXI_lite_read

AXI_lite_read portion is one of the Interfaces with AXI interconnect, and it is responsible for sending (presenting) data to a host. Fig5.2.3 shows a state machine of this portion.

- The implementation of this portion has been copied and modified from mdio_gen platform-block.
- AXI_lite clock domain

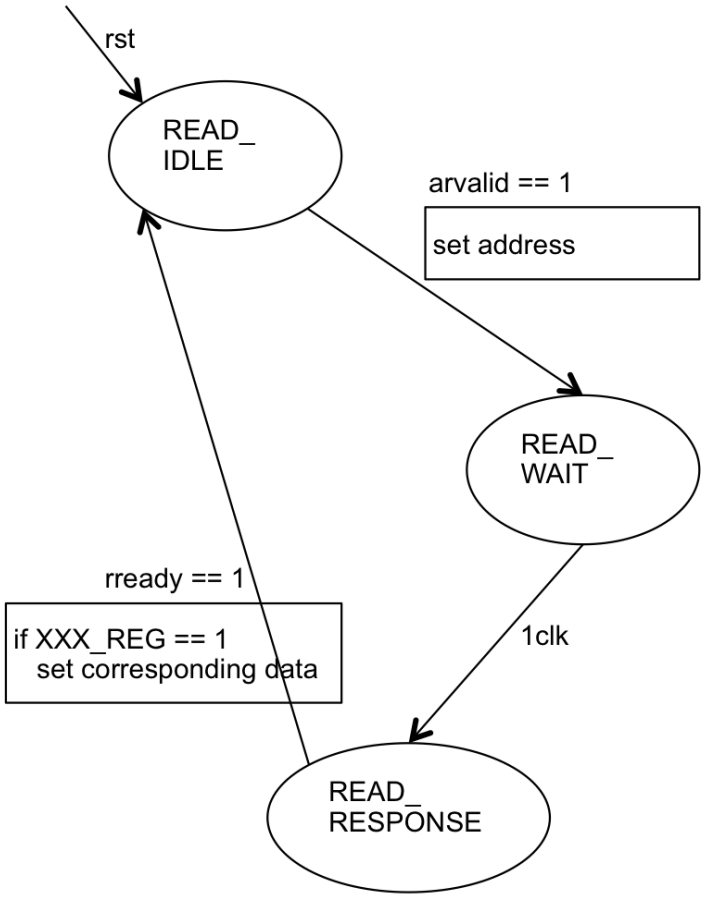


Fig5.2.3 AXI_lite_read State machine

(3) entry_write_buf

Entry_write_buf is a buffer that accommodates one entire entry(match/mask/action/statistics).
AXI_lite clock domain

(4) entry_read_buf

Entry_read_buf is a buffer that accommodates a statistics portion of an entry.
AXI_stream clock domain

(5) ctrl_sig_process

Ctrl_sig_process is responsible for exchanging control signals with flow_table_ctlr module.
When a host has written a value to ‘write’, ‘mod_write’ or ‘read’ register, and AXI_lite_write module has decoded it, ctrl_sig_process receives 'write', ‘mod_write’ or 'read' as a signal from it.
Ctrl_sig_process treats each of those signals as a trigger and starts handshaking with flow_table_ctlr.
At the same time it de-asserts 'access_enable' flag to zero. From a host, the status zero can be seen from acc_rdy register and it means host cannot read/write valid data.

Once the process with flow_table has been finished, it asserts back 'access_enable' flag to ‘one’. From a host, the status can been seen from acc_rdy register and it means host can read/write data.

Fig5.2.4 shows a state machine of this portion.

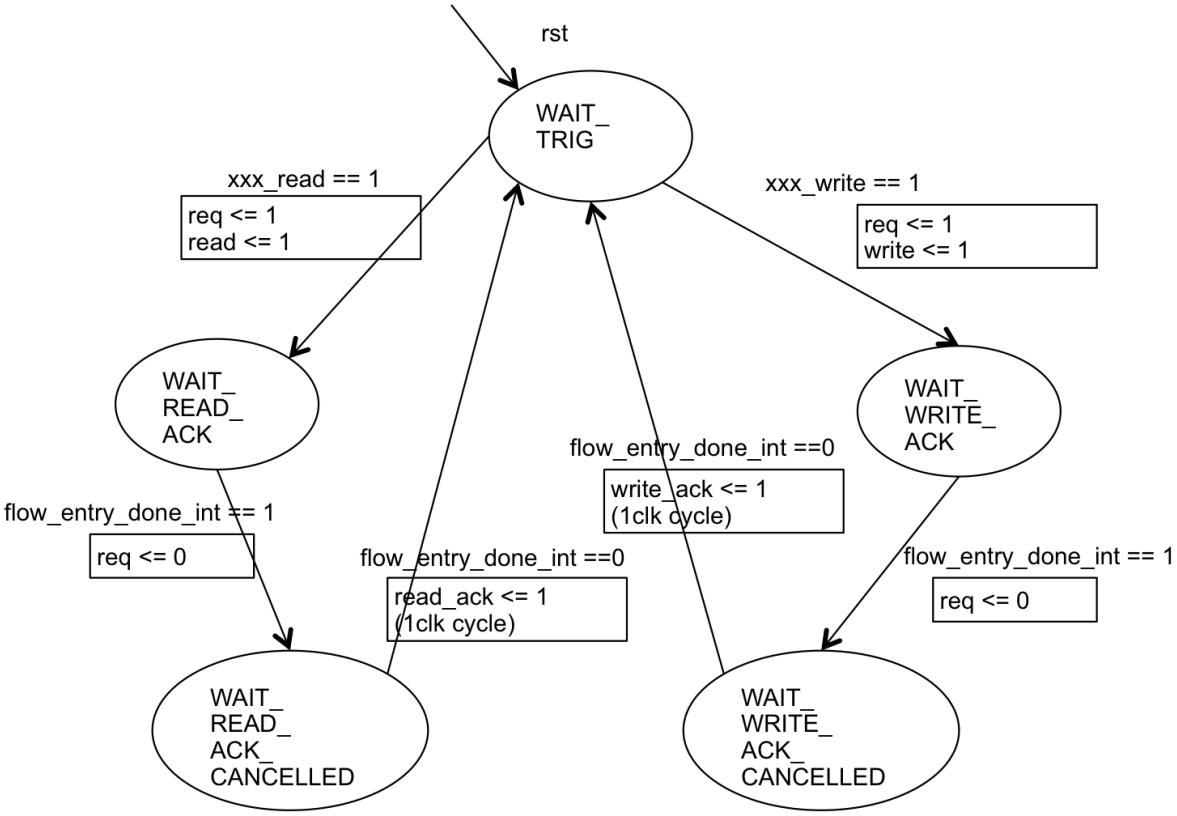


Fig5.2.4 Ctl_sig_process State Machine

5-3. flow_table_ctlr

5-3-1. Summary

Flow_table_ctlr interfaces flow tables and all the requesters. It has a single pipeline for each exact match and wildcard match table, and all the ports timeshare the pipeline.

When a port (pkt_preprocessor) queries if there is a matching field, it consults both exact match table and a wildcard match table. If it found a matching entry, it grabs a corresponding action from memory and sends it to the same port’s action_processor and updates a statistics.

If it matches both exact match table and wildcard match table, then it uses the action for exact match table.

5-3-2. Block diagram

Fig5.3.1 shows a block diagram of flow_table_ctlr. See 5.3.4 for descriptions.

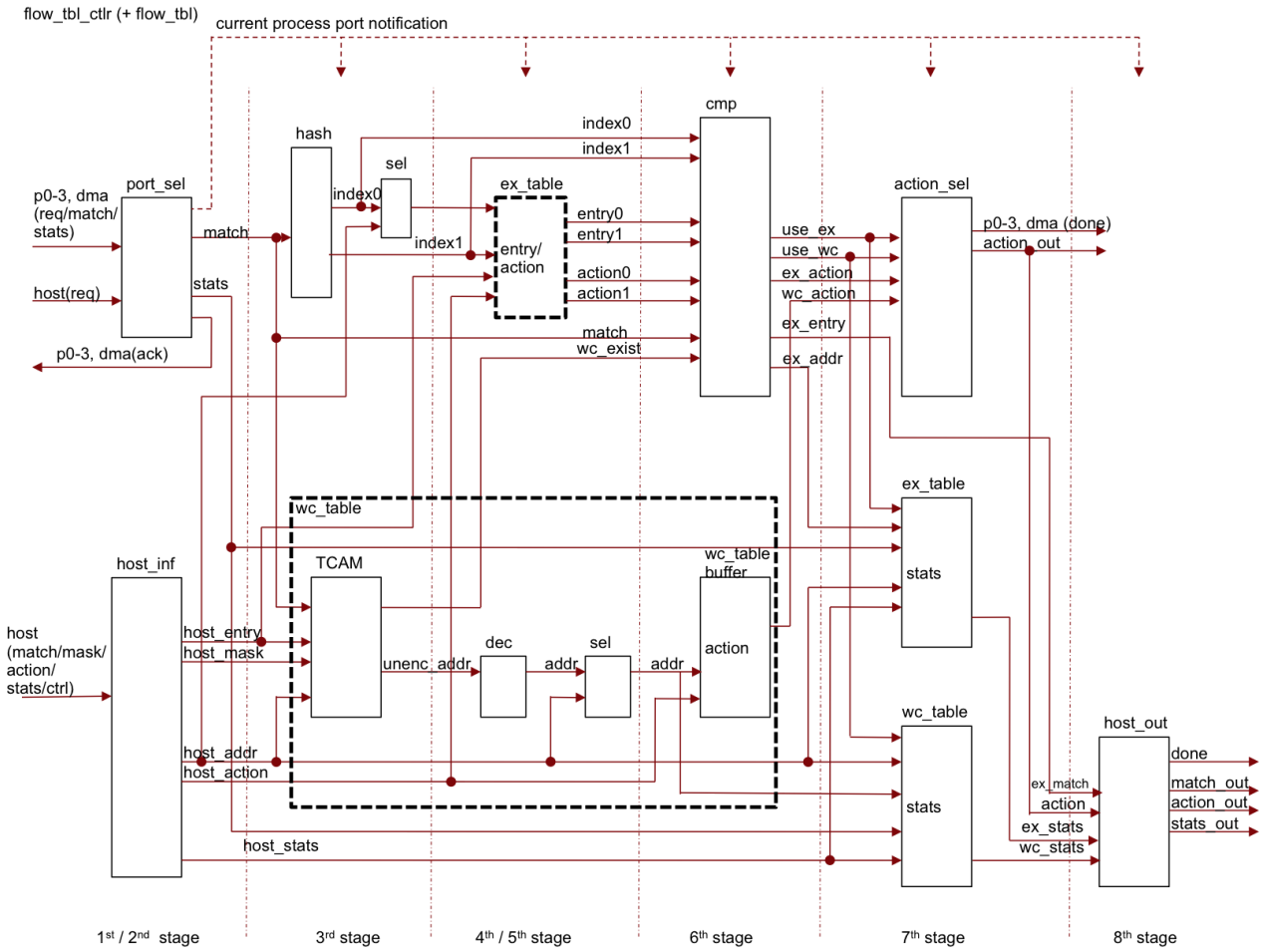


Fig5.3.1 flow_table_ctlr Block diagram

5-3-3. Interfaces

All the interfaces to/from flow_table_ctlr are proprietary.

(1) Flow match query/response

All the ports (port0, 1, 2, 3, DMA) have the same interface:

- p<4-0>_req: (in)
    - When asserted to one, it means this port has a parsed header for a query. It is set to zero when p<x>_ack is consumed by a requester(pkt_preprocessor).
- p<4-0>_match:(in)
    - Parsed header information to be matched against lookup tables in this module
- p<4-0>_ack (out):
    - Sent from this module when the request is accepted and p<x>_match is started

to be carried in the pipeline. It does NOT mean the process is done or action is ready.
    - This signal is asserted to one for one-clock period.
- p<4-0>_done (out):
    - Sent from this module when the process is done and action is ready.
    - This signal is asserted to one for one-clock period.
- action (out):
    - Sent from this module
    - Corresponding action information for requested header information IF it has a matching entry.
    - All zero IF it does NOT have a matching entry meaning ‘drop this packet’.
    - Valid only when p<x>_done is asserted to one.

(2) Host interface

When a host writes a flow entry information, the data is carried on the same pipeline. Therefore it has as same handshaking interface as (1).

- host_addr: (in)
    - Address (row) of tables (Lower half memory area is for exact match table, upper half memory area is for wildcard match table. See section4.)
- host_req: (in)
    - Same as p<x>_req
    - When asserted to one, it has an entry information to be registered
- host_ack: (out)
    - Same as p<x>_ack

  This signal is just for compatibility. Do NOT use it.
- host_done: (out)
    - Same as p<x>_done
    - If the request was ‘write’, then the preceding module (host_inf) can release a flow entry information (host_match_in, host_mask_in, host_action_in, host_stats_in)
    - This signal is asserted to one for one-clock period.
- host_write: (in)
    - When ‘one’, it means a host wants a write access
    - When ‘zero’, it means a host wants a read access
    - Keep asserting the same signal (0 or 1) until host_done is asserted
- host_stats_en: (in)
    - If 1 when host_write, write also host_stats_in in addition to host_match_in, host_mask_in and host_action_in. If not, do not write host_stats_in information
    - If read (host_write = 0), it will be ignored
- host_match_in: (in)
    - Flow entry information to be written
- host_mask_in: (in)
    - Flow entry information to be written
- host_action_in: (in)

    - Flow entry information to be written
- host_stats_in: (in)
    - Flow entry information to be written
    - Used when host_stats_en is active
- host_match_out: (out)
    - Flow entry information to be read
    - Not valid for wildcard table
- host_action_out: (out)
    - Flow entry information to be read
- host_stats_out: (out)
    - Flow entry information to be read

(3) Counters

- exact_hit: (out)
    - Incremented if there is a matching entry in exact match table
- exact_miss: (out)
    - Incremented if there is NOT a matching entry in wildcard match table
- wildcard_hit: (out)
    - Incremented if there is a matching entry in exact match table
- wildcard_miss: (out)
    - Incremented if there is NOT a matching entry in wildcard match table

(4) Misc

- openflow_timer: (in)
    - Counts at each second. It rolls over when it reached the limit.
    - Reference timer to measure the time difference between one packet to another. AXI streaming clock is used.

5-3-4. Registers

All the information in this module can be accessed via host_inf.

5-3-5. Details

In this module, a flow matching and an entry registration from a host are processed in a pipeline. The pipeline has eight stages and each stage consumes one clock cycle.
An exact match table and a wildcard match table are searched in parallel.
The current implementation uses

- Xilinx XAPP IP core TCAM for wildcard entry storage
- BRAM for exact match entry and action storage (phase1 implementation), wildcard action storage, and both statistics storage.

For collecting statistics, BRAM will be used even though exact match table is moved out to

SRAM, since it has to be accessed frequently (read, update and write, periodical read from host), and the data is relatively small (64bit / entry).

It consists of 7 stages for one query after the request is accepted. So the result (action) will be sent in 7 clock after the selection.

Refer to Fig5.3.1 for block name in the following descriptions

(1) 1st and 2nd stage
The port to be processed is selected on these two stages.
Port_sel block receives table_lookup requests from 4 physical ports and a CPU DMA port. Then it selects which port it should process on this clock, and it passes the selected information (flow_entry) to the next stage. The selection is made in a round robin way, and the most prioritized port will be shifted and changed clock by clock. Once one port is selected, it sends back 'ack' signal to the selected requester (port) at the next clock.
The request includes access from host for entry writing and statistics reading/writing (initializing). The access is treated in a same way as other ports.

(2) 3rd stage
Exact match table block and wildcard match table block start processing the information of the previous stage.

Exact_table_hash block generates two addresses from the given entry. It uses CRC-32 to create it. Software will use the same mechanism to choose the address for writing info.
Wildcard_cam is an XILINX IP core with 32 rows and 256bit width (32bit width x 8pcs are used for enhancing the speed to get a result). Unencoded format address will be used since we use multiple TCAMs and the entry which matches on all the TCAMs is needed. As mentioned, it decodes unencoded_address results from all the TCAMs and chooses one entry address. If it still hits multiple portions, the lowest row will be chosen.
Host block passes along address and all the data to the next stage. Nothing will be done on this stage for a host.

(3) 4th and 5th stage
On these stages, exact match block finds entries and actions for both two hashed addresses, and wildcard match block decodes its result and passes it to the next stage.

For exact_entries and exact_actions, the phase1 implementation introduces a dual-port BRAM (1K entries) to get two information at one time. If this stage is for host access, one of the dual-port ports is used to read/write entries. SRAM will be used in the future design and it may require some additional glue logics.
For wildcard process, the information from multiple TCAMs (see (2)) is collected and decoded for the next stage.

(4) 6th stage

In this stage, the results from exact match table (2 results) and wildcard table (1 result) are ready to be checked. If a matching entry is available, one of them is chosen to the next stage. If there's no matching entry, it sends 'nothing matched' information to the next stage.

Comparator checks the results. If both exact_match and wildcard_match have matching entries, exact entry is always chosen. Among the two entries of exact match, only one address should be matched maximum. ONLY IF both entries are matched, either one (portA) is used with incrementing an error counter (since it shouldn't happen).

For exact match, according to the decision between the two entries, it will choose which address to be used for status update. If this stage is for host access, host address will be chosen.

Same for wildcard match. The address will be passed to the next stage for status update, but if this stage is for host access, host address will be passed through.

In this stage the counters are updated. It has five counters that are: exact hit, exact miss, wildcard hit, wildcard miss and exact_error (meaning two exact entries are somehow matched).

Wildcard_action uses dual-port BRAM to get two information at one time. Depth is also 32 addresses. If this stage is for host access, one of the dual-port ports (portA) is used to read/write entries.

(5) 7th stage

On this stage it finally sends out a result to the requesting module. It also reads current status info from (still) both exact and wildcard tables. It may be a host access.

In action_sel block, one action among two exact match candidates and a wildcard match candidate is selected and it is sent out to the requested module, along with Done signal.

In this stage the statistics counter is also updated.

The current statistics data will be read for hit entry. Even if the particular match (either exact match or wildcard match) has not been selected and the data is not been used, statistics for both match table will be read anyway. In case of a host access for particular match address (either exact match or wildcard match), the information will be read regardless it is host read access or host write access.

(6) 8th stage

In this stage, if the entry hits either exact or wildcard match table, the particular table's statistics will be updated. If it is a host write access, the specified data will be written to the specified row of the specified table.

The updated information includes:

- packet count
- byte count

- current timer value

All the process is finished at this stage.

5-4 action_processor

5-4-1 Summary

The role of action_processor is to specify its forwarding ports and to update header fields and length of the packets, referring to ‘action’ received from flow_table_ctlr.

5-4-2 Block diagram

Fig5.4.1 shows a block diagram of action_processor. See 5.4.4 for descriptions.

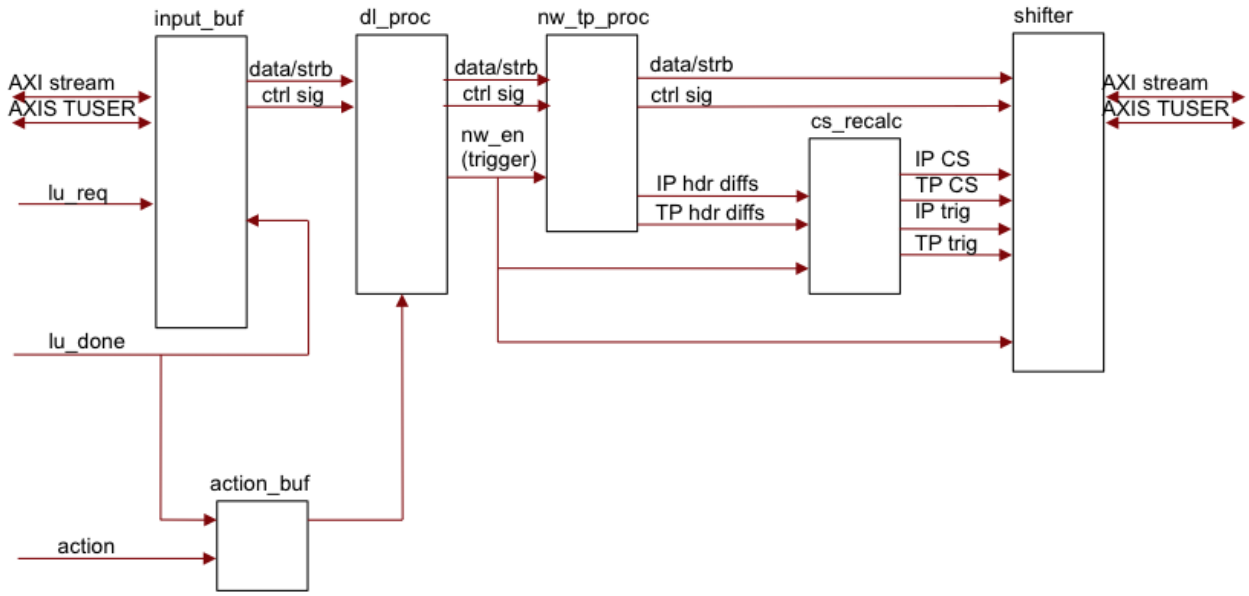


Fig5.4.1 action_processor Block diagram

5-4-3 Interfaces

(1) AXI stream (slave)

- Bitwidth 64bit (fixed)
- TUSER 128bit

(2) Flow match response

- lu_req: (in)
    - Sent from pkt_preprocessor
    - When asserted to one, it means this port has a parsed header for a query.

    - Used as a trigger to start fetching a packet from pkt_preprocessor.
- lu_done (in):
    - Sent from flow_table_ctlr when the process is done and action is ready.
    - When asserted to one, it will get ‘action’ and store it until the next ‘lu_done’.
    - This signal is asserted to one for one-clock period.
- action (in):
    - Sent from flow_table_ctlr
    - Corresponding action information for a query IF it has a matching entry.
    - All zero IF it does NOT have a matching entry, meaning ‘drop this packet’.
    - Valid only when lu_done is asserted to one.

(3) AXI stream (master)

- Bitwidth 64bit (fixed)
- TUSER 128bit

5-4-4 Details

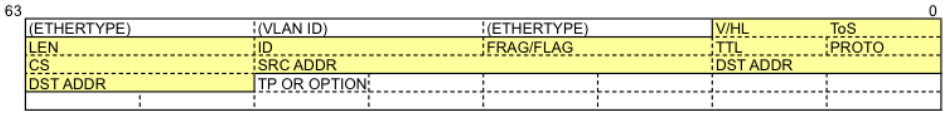
(1) Preprocessing blocks

When 'lu_req' is asserted by a previous module, it asserts a read signal to the previous block to get data from it. The data then is stored into input_buf. Input_buf has a minimum depth to reduce latency. Two fifos are implemented for packet itself and for TUSER which arrives at the first clock cycle of each packet. This module will latch it at the first clock for the latter module.

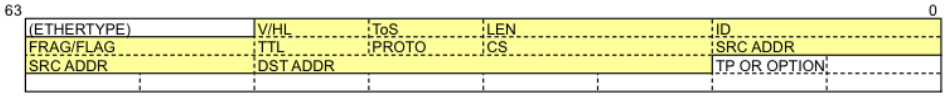
When 'lu_done' is asserted by a previous module (flow_table_ctlr), it latches the given ‘action’ into action_buf. Once it gets the ‘action’, it starts reading data from input_buf and start processing. It updates forwarding port field in TUSER, then it processes a packet layer by layer, including updating L2 fields, add/remove VLAN tag, modifying L3/L4 fields with updating its L3/L4 checksums.
Packet length is also updated after the DL process has been done when Vlan tags have been added or removed.

Fig5.4.2 shows a bit position to be processed for each type of packet.

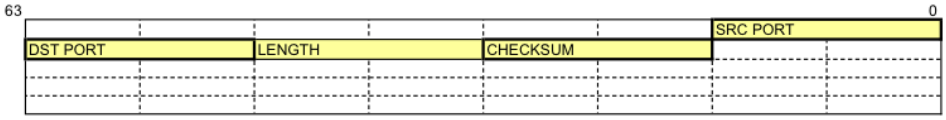
IP Packet
case1



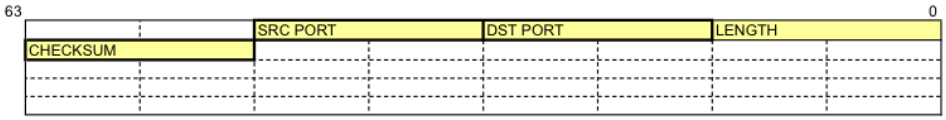
IP Packet
case2



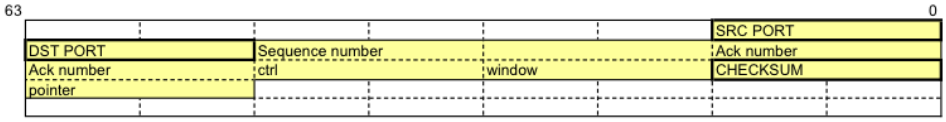
UDP Packet
case1



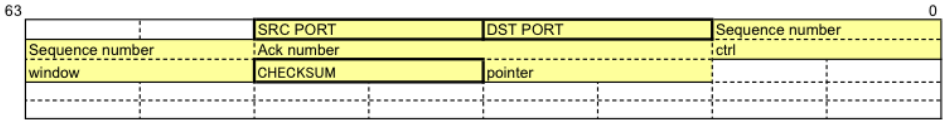
UDP Packet
case2



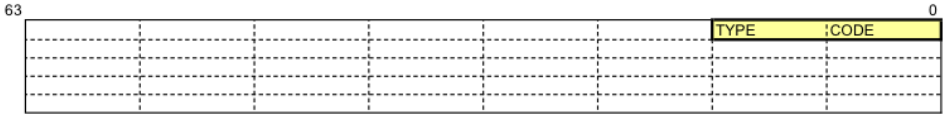
TCP Packet
case1



TCP Packet
case2



ICMP Packet
case1



ICMP Packet
case2

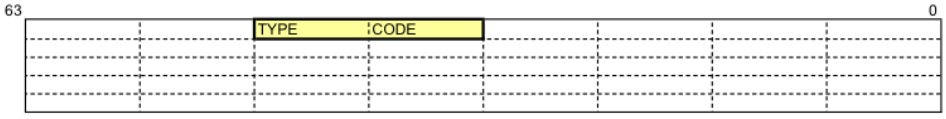


Fig5.4.2 Header fields to be processed

(2) dl_proc

Fig5.4.2 shows a state machine for Layer2 processing. It modifies MAC addresses and updates/adds/removes VLAN tags if they are specified in ‘action’.

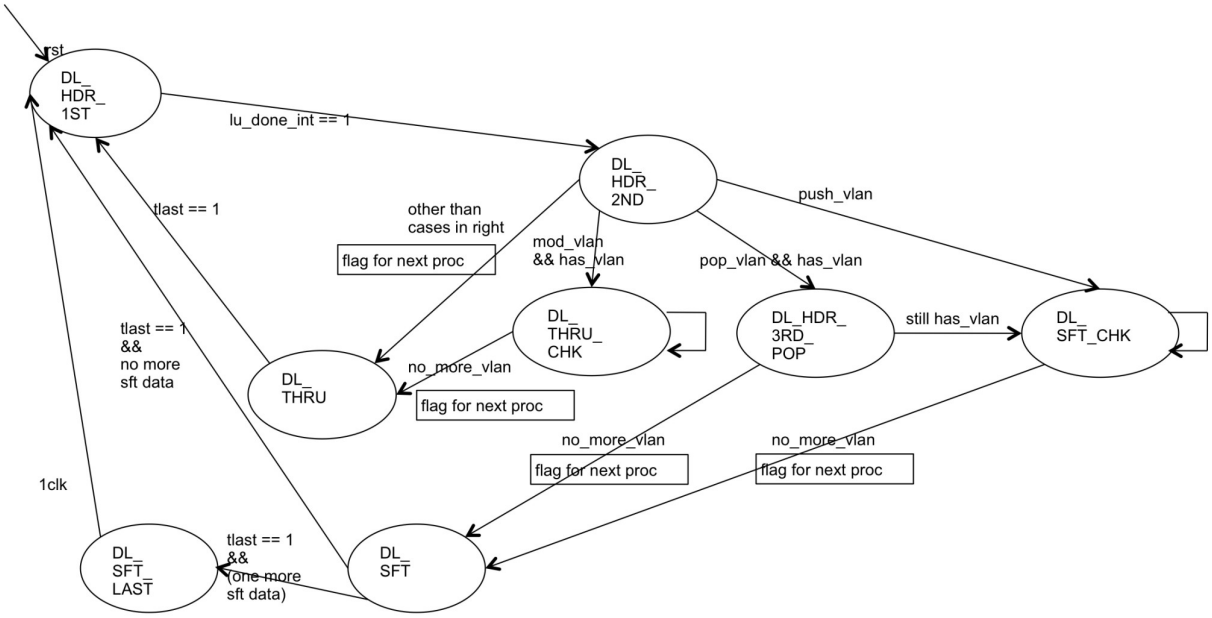


Fig5.4.2 DL_PROC state machine

VLAN tag processing starts at a certain bit position(bit 96). It processes the outer-most vlan.
- When the action is VLAN push:
1) Halt the packet flow, but not stopping receiving data on input_buf
2) If no data provided, copy the value from existing tag and add it
3) Shift all the remaining data
- When the action is VLAN pop:
1) Halt the packet flow, but not stopping receiving data on input_buf
2) Remove one tag and shift it until the next (tag) length has come
- When the action is modify VLAN
1) simply replace the value and not stopping receiving packet on input_fifo or halting a packet flow

Once all the process have been done, it adds a flag for the next process to indicate where (bit position) it should start processing.

(3) nw_tp_proc

Fig5.4.3 shows a state machine for Layer3/4 processing. It modifies IP addresses, TOS and TCP/UDP ports if they are specified in ‘action’.
The packet may have L2 (only VLAN, at this point) tags and bit position varies, so it has two

states (PTN1 and PTN2) to handling whichever cases.

During processing this state machine, it takes diffs for all the IP fields, even if the field has not been changed, to recalculate its checksum.

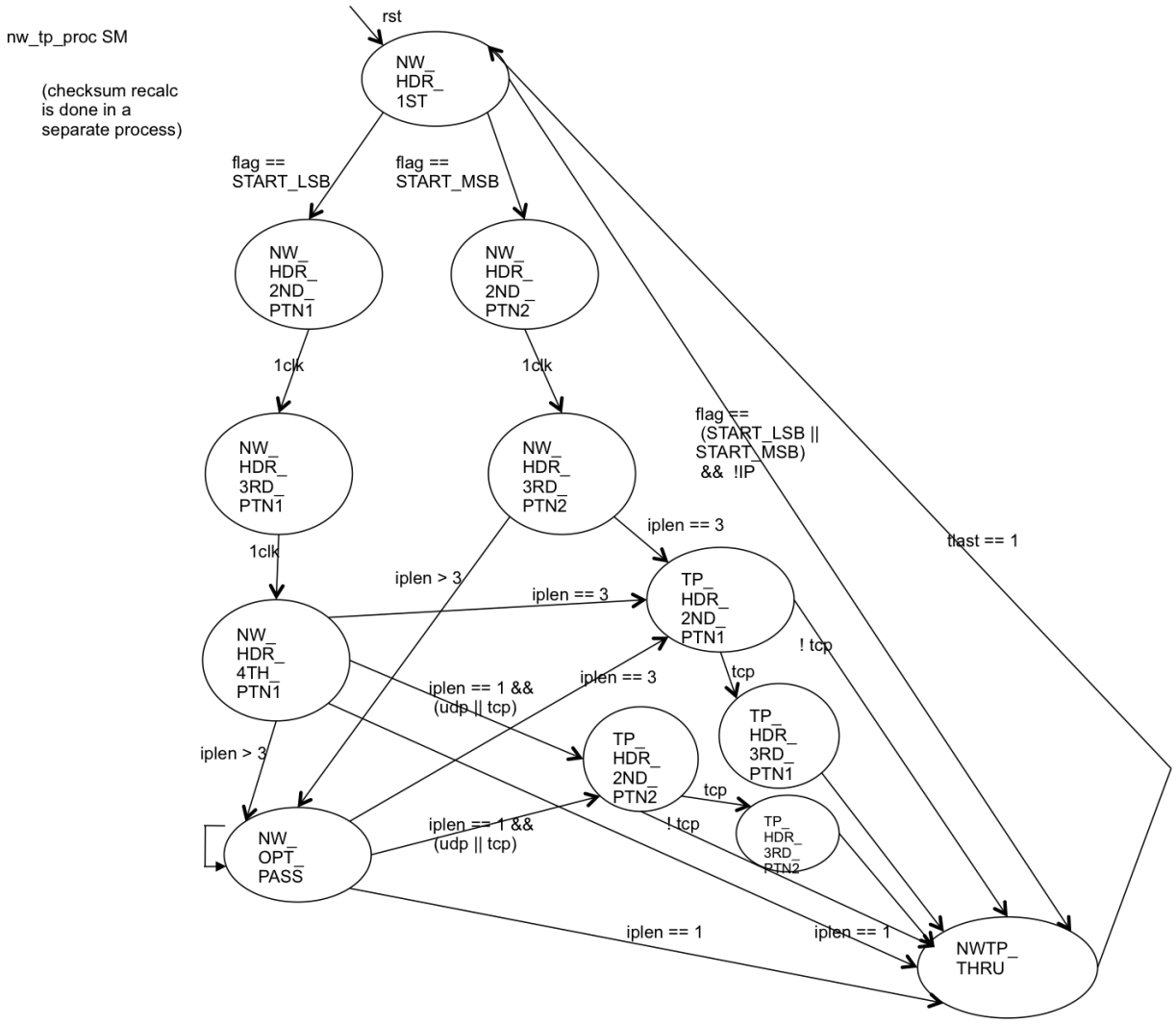


Fig5.4.3 NW_TP_PROC state machine

(4) cs_recalc

Fig5.4.4 shows a state machine for recalculating L3 checksum, and Fig5.4.5 show a state machine for L4 checksum recalculation.

RFC1624 is used for checksum recalculation.

NewChecksum =

~(~OldChecksum + (~OldData1 + NewData1) + ... + (~OldDataX + NewDataX))

It buffers a packet while it overwrites recalculated IP/UDP/TCP checksum, and it shifts the packet five times until it finishes rewriting a checksum.

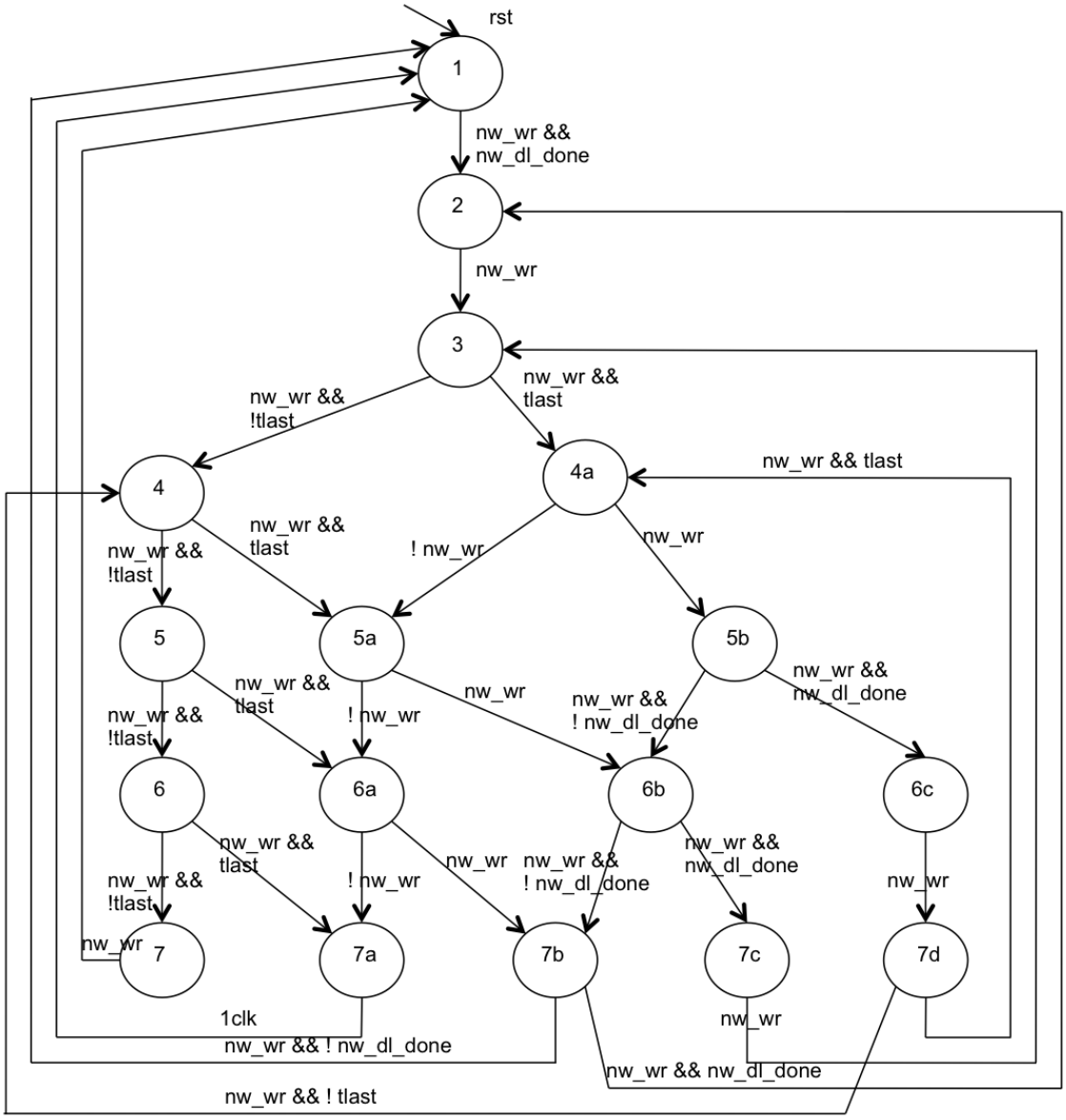


Fig5.4.4 L3 checksum recalculation state machine

tp_chksum_SM

(checksum recalculation)

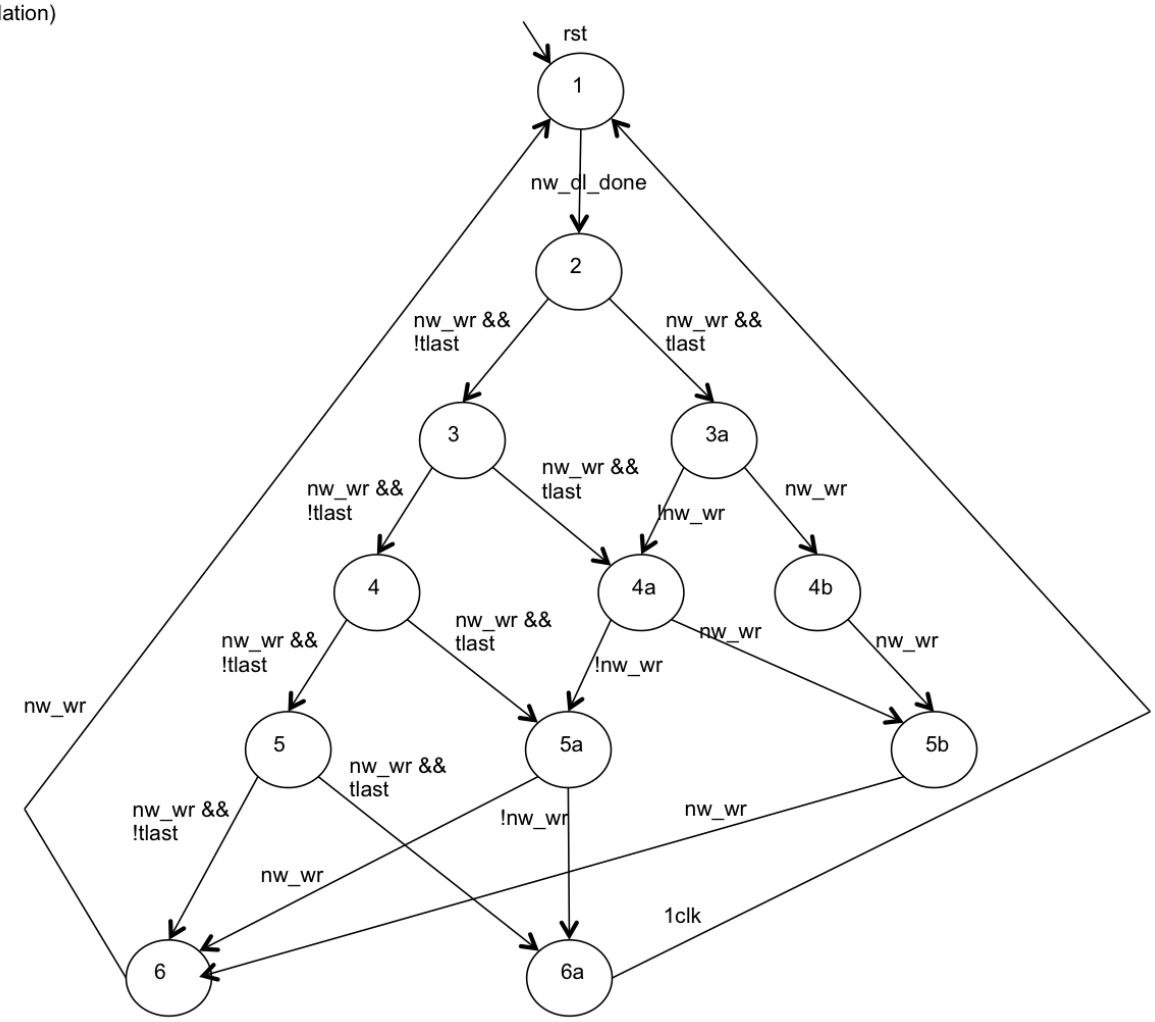


Fig5.4.5 L4 checksum recalculation state machine

6. Hardware development and running environment

6-1. OS

Developed on:

- Fedora16 64bit (3.3.5-2.fc16.x86_64)

Tested with NetFPGA-10G + software on:

- Fedora14 64bit (2.6.35.14-106.fc14.x86_64)

Note: The development environment can/should be Felora14. The reason why Fedora16 was used for develoment is only because it was the only development machine provided.

6-2. Tools and additional IP cores

- Xilinx ISE13.4
- Xilinx TCAM xapp1151_cam_v1_1.tar.gz

6-3. Installation and setup

- cd NetFPGA-10G-live/
- make
- cd contrib-projects/openflowswitch/hw/pcores/openflow_datapath_v1_00_a/
- make
- cd

6-4. TCAM installation

6-4-1. Xilinx Application

OpenFlow design requires XILINX TCAM module. Obtain ‘xapp1151’ application from XILINX website. The tested TCAM version is v1_1. As of the time of writing this document, the following link has the design files and its document of xapp1151.

http://www.xilinx.com/support/documentation/anmeminterfacestorelement_cam.htm

Please note that xilinx ISE must be installed to generate the core. Tested ISE version is 13.3.

6-4-2. Tool installation

On a directory you prefer, run:

- tar xfvz xapp1151_cam_v1_1.tar.gz

6-4-3. TCAM module generation

- cd xapp1151_cam_v1_1/implement
- xilperl CustomizeWrapper.pl

You need to specify several parameters. To choose those options, see follows:

```
**************************************************************
CAM Reference Design Customizer v1.1
**************************************************************

**************************************************************
Please input the following parameters for the CAM:
**************************************************************

Enter the FPGA base architecture to target
Valid options are virtex4, virtex5, virtex6, virtex6l, spartan3, spartan3e, spartan3a, spartan3adsp,
aspartan3, aspartan3e, spartan6:
virtex5

Input the CAM data width.  Valid values are 1-512:
32

Input the CAM depth.  Valid values are 16-4096:
32

Enter the CAM memory type to implement
(Choose SRL-based if you need a Ternary or Enhanced Ternary mode CAM)
0 = SRL-based, 1 = BRAM-based: 0

Choose the Ternary Mode setting.
0 = Ternary Mode Off, 1 = Standard Ternary Mode, 2 = Enhanced Ternary Mode: 1

NOTE: Enhanced Ternary Mode does not support MIF file memory initialization, thus a Write
Enable is required.
Setting C_MEM_INIT = 0 ...
Setting C_HAS_WE = 1 ...

Choose what type of encoding the MATCH_ADDR port will have.
0 = Binary Encoded, 1 = Single Match Unencoded (one-hot), 2 = Multi-match Unencoded: 2

Please select the optional features to be implemented:
Simultaneous Read/Write (y/n): y

Please select from the following optional input ports:
```

Enable (EN) (y/n): *n*

Please select from the following optional output ports:
Multiple Match Flag (MULTIPLE_MATCH) (y/n): *n*
Single Match Flag (SINGLE_MATCH) (y/n): *n*
Read Warning Flag (READ_WARNING) (y/n): *n*

After the selections the parameters should be as follows:

VHDL parameters were generated as follows:

```
================================================
C_FAMILY              : string  := virtex5;
C_MEM_TYPE                    : integer := 0;
C_WIDTH               : integer := 32;
C_DEPTH               : integer := 32;
C_ADDR_TYPE                   : integer := 2;
C_MATCH_RESOLUTION_TYPE   : integer := 0;
C_TERNARY_MODE            : integer := 1;
C_HAS_WE              : integer := 1;
C_MEM_INIT            : integer := 0;
C_HAS_CMP_DIN             : integer := 1;
C_REG_OUTPUTS             : integer := 0;
C_HAS_EN              : integer := 0;
C_HAS_MULTIPLE_MATCH: integer := 0;
C_HAS_SINGLE_MATCH    : integer := 0;
C_HAS_READ_WARNING   : integer := 0;

================================================
```

Then in the same directory, edit vhdl_xst.scr so that parameters are as follows:

```
-ifmt VHDL
-work_lib cam
-p  xc5vtx240t-ff1759-2
-write_timing_constraints No
-ifn vhdl_xst.prj
-iobuf NO
-max_fanout 100
-ofn ./results/cam.ngc
-ofmt NGC
-bufg 1
-bus_delimiter ()
```

-hierarchy_separator /
-case Maintain
-opt_mode Speed
-opt_level 1
-loop_iteration_limit 5000
-use_new_parser yes

Then run:

- xilperl RunXST.pl

6-4-4. Verilog file for simulation/implementation

After finishing above, run follows to create a verilog file for simulations:

- cd results
- netgen -sim -ofmt verilog cam.ngc

Edit the created cam.v file.

(1) Change the module name to '*cam*' from 'cam_wrapper'.

(2) Just after the port declaration, write a sentence:
// synthesis translate_off

(3) And before endmodule of 'cam_wrapper', write;
// synthesis translate_on
Please note the file has two modules therefore two 'endmodule'. Make sure to insert the line for the first module; cam_wrapper (now the module name is 'cam').

6-4-5. File copy

Copy cam.ngc in netlist directory of OpenFlow pcore:

- cp cam.ngc
  contrib-projects/openflowswitch/hw/pcores/openflow_datapath_v1_00_a/netlist/

Copy cam.v in verilog directory of OpenFlow pcore:

- cp cam.v
  contrib-projects/openflowswitch/hw/pcores/openflow_datapath_v1_00_a/hdl/verilog/

6-5. DMA customization (Optional)

In the initial version of OpenFlow design, DMA module runs at 125MHz instead of 250MHz. If you plan to run it at higher speed, the DMA module provides a 250MHz mode. To use that, perform following and set up the environment for it.

6-5-1. Choose 125MHz netlist

Cd to DMA pcore's netlist directory in the following path.

- lib/hw/contrib/pcores/dma_v1_00_a/netlist

Three files should be visible in the directory:

- dma_engine_125MHz.edf
- dma_engine_250MHz.edf
- dma_engine.edf

"dma_engine.edf" is a static link to one of the actual netlists. Initially it should be a link to dma_engine_125MHz.edf. Run follows to create a link to dma_engine_250MHz.edf.

- rm dma_engine.edf
- ln -s dma_engine_250MHz.edf dma_engine.edf

6-5-2. Modify XCO file

Cd to DMA pcore's xco directory.

- lib/hw/contrib/pcores/dma_v1_00_a/xco

Then edit the following file.

- vim endpoint_blk_plus_v1_15.xco

Find the line mentioning:

- CSET interface_freq=125

Change it to:

- CSET interface_freq=250_default

6-5-3. Remake a core

Cd to DMA pcore directory.

- lib/hw/contrib/pcores/dma_v1_00_a

Then run make

- make

6-6. Simulations

Flow_table setting script and input_packet_data are provided for each test case. *For each test, copy five packet files (stream_data_in_X.axi) and one register access file (reg_stim.axi)* from each directory to the following directory:

- NetFPGA-10G-live/contrib-projects/openflowswitch/hw/axi_simulation

In NetFPGA-10G-live/contrib-projects/openflowswitch/hw directory, run:

- make axisim

In the current environment, seeing the waveforms and performing ‘diff’ with provided stream_data_out_X.axi are the way to check the correctness.

The available tests are as follows:

	title	detail
1	exact_match_outport	Set up an exact match entry then send packets. Packets should be sent out the specified port.
2	exact_match_mod_dl_addr	Set up an exact match entry then send packets. Packets should be sent out the specified port and MAC address of the packets should be updated.
3	exact_match_add_vlan	Set up an exact match entry then send packets. Packets should be sent out the specified port and VLAN tag should be added.
4	exact_match_mod_nw_addr	Set up an exact match entry then send packets. Packets should be sent out the specified port and IP address of the packets should be updated.
5	exact_match_mod_nw_tos	Set up an exact match entry then send packets. Packets should be sent out the specified port and IP TOS field of the packets should be updated.
6	exact_match_mod_tcp_port	Set up an exact match entry then send packets. Packets should be sent out the specified port and TCP port of the packets should be updated.
7	exact_match_mod_udp_port	Set up an exact match entry then send packets. UDP packets should be sent out the specified port and UDP port of the packets should be updated.
8	wildcard_match_outport	Set up an wildcard match entry then send packets. Packets should be sent out the specified port.
9	wildcard_match_mod_dl_addr	Set up an wildcard match entry then send packets. Packets should be sent out the specified port and MAC address of the packets should be updated.
10	wildcard_match_add_vlan	Set up an wildcard match entry then send packets. Packets should be sent out the specified port and VLAN tag should be added.
11	wildcard_match_mod_nw_addr	Set up an wildcard match entry then send packets. Packets should be sent out the specified port and IP address of the packets should be updated.

12	wildcard_match_mod_nw_tos	Set up an wildcard match entry then send packets. Packets should be sent out the specified port and IP TOS field of the packets should be updated.
13	wildcard_match_mod_tcp_port	Set up an wildcard match entry then send packets. Packets should be sent out the specified port and TCP port of the packets should be updated.
14	wildcard_match_mod_udp_port	Set up an wildcard match entry then send packets. UDP packets should be sent out the specified port and UDP port of the packets should be updated.

6-7. Create a bitfile

- cd NetFPGA-10G-live/contrib-projects/openflowswitch/
- make

After all the process has been done and making sure the implementation has been successful,

- cd bitfiles

And you will see download.bit in the directory.

6-8. Download a bitfile

run:

- impact

Then select your bitfile and download it. You may need to reboot your machine after the download.

6-9. Build a DMA Driver

- cd NetFPGA-10G-live/projects/reference_nic/sw/host/driver
- make

6-10. Insert a DMA Driver kernel module

In NetFPGA-10G-live/projects/reference_nic/sw/host/driver directory,

- sudo insmod nf10.ko

6-11. Link-up NetFPGA interfaces

- sudo ifconfig nf0 up
- sudo ifconfig nf1 up
- sudo ifconfig nf2 up
- sudo ifconfig nf3 up

7. Software development and running environment

7-1. OS

Fedora14 64bit (2.6.35.14-106.fc14.x86_64)

Note: The OpenFlow software in the user space won't properly run on Fedora16(Kernel3.x) as it consumes VLAN tags and the OpenFlow software can't get them for manipulation.

7-2. Prerequisites

- User needs to have a sudo access.
- User needs at least two dedicated 10G NIC ports for testing.

7-3. OpenFlow software installation

7-3-1. Install necessary tools

- sudo yum -y install git automake pkgconfig libtool gcc make

7-3-2. Install OpenFlow-with-NetFPGA-10G software

- git clone git://github.com/eastzone/openflow.git
- cd openflow
- git checkout netfpga-10g
- ./boot.sh

Note: If you have modified the hardware platform, make sure you copy the latest xparameter.h from projects/openflowswitch/sw/host/include/ in the NetFPGA-10G tree to openflow/hw-lib/nf2/ in the openflow tree.

7-3-3. Build OpenFlow user-space switches with NetFPGA support

- cd openflow
- ./configure --enable-hw-lib=nf2
- make
- sudo make install

7-4. Testing tools installation

7-4-1. Setup testing environment

Stop avahi, to prevent it from polluting the tests with link discovery packets:

- sudo /sbin/chkconfig avahi-daemon off
- sudo /etc/rc.d/init.d/avahi-daemon stop

Disable IPv6, to prevent it from polluting the tests with link discovery packets: First, perform the following command

- sudo /sbin/chkconfig ip6tables off

Then create and edit disable-ipv6.conf file.

- sudo su
- echo 'options ipv6 disable=1' > /etc/modprobe.d/disable-ipv6.conf
- exit

Edit network file.

- sudo vim /etc/sysconfig/network

On the file, find the line mentioning "NETWORKING_IPV6=". If NETWORKING_IPV6=yes, change it to "no".

- NETWORKING_IPV6=no

7-4-3. Install necessary tools for OFTest

- sudo yum -y install python-setuptools
- sudo yum -y install scapy

7-4-4. Install OFTest

- git clone git://github.com/floodlight/oftest.git
- cd oftest
- git checkout 9130bcd # the version this bitfile is tested against

7-4-5. Modify a file in OFTest

Check your TWO nic ports. The following assumes the port names are eth1 and eth2.

- cd oftest/tests
- vim remote.py

Then find the lines:

```
remote_port_map = {
        23 : "eth2",
        24 : "eth3",
```

```
    25 : "eth4",
    26 : "eth5"
    }
```

And change it to:

```
remote_port_map = {
    1 : "eth1",
    2 : "eth2"
    }
```

7-4-6. Build OFTest

- cd oftest/tools/munger
- make install

7-5. Run OpenFlow Switch for OFTest

Assumptions:

- NetFPGA nf0 is wired to NIC eth1 and NetFPGA nf1 is wired to NIC eth2
- OpenFlow bitfile is downloaded to FPGA
- DMA driver is inserted on the machine
- NetFPGA interfaces are up

Open two terminals then on each terminal (all run as root):
Terminal 1:

- ofdatapath punix:/var/run/test -i nf0,nf1,nf2,nf3

Terminal 2:

- ofprotocol unix:/var/run/test tcp:127.0.0.1:6633

7-6. Run OFTest

Open another terminal. Then:

- cd oftest/tests
- ./oft --port-count=2 --platform=remote

To find the details about OFTest parameters, type:

- ./oft --help

7-7. Failing tests

You are testing it with two-port NIC environment, so you will see following test cases failing because it doesn't have enough ports for those tests:

- DirectMC
- DirectMCNonIngress
- DirectTwoPorts
- FloodMinusPort

8. Implementation results (Reference)

Following is the map result of phase1 implementation (w/ supporting all the actions)

```
Design Summary
--------------
Number of errors:      0
Number of warnings:  214
Slice Logic Utilization:
  Number of Slice Registers:          69,340 out of 149,760   46%
        Number used as Flip Flops:          69,265
        Number used as Latches:                  1
        Number used as Latch-thrus:             74
  Number of Slice LUTs:               63,781 out of 149,760   42%
        Number used as logic:               57,236 out of 149,760   38%
        Number using O6 output only:        52,182
        Number using O5 output only:         3,175
        Number using O5 and O6:              1,879
        Number used as Memory:              5,997 out of  39,360   15%
        Number used as Dual Port RAM:              861
        Number using O6 output only:         212
        Number using O5 output only:          17
        Number using O5 and O6:              632
        Number used as Shift Register:      5,136
        Number using O6 output only:        5,136
        Number used as exclusive route-thru:        548
  Number of route-thrus:              4,150
        Number using O6 output only:        3,598
        Number using O5 output only:          431
        Number using O5 and O6:               121

Slice Logic Distribution:
  Number of occupied Slices:          27,816 out of  37,440   74%
  Number of LUT Flip Flop pairs used:         90,337
        Number with an unused Flip Flop:    20,997 out of  90,337   23%
        Number with an unused LUT:          26,556 out of  90,337   29%
        Number of fully used LUT-FF pairs:  42,784 out of  90,337   47%
        Number of unique control sets:       3,709
        Number of slice register sites lost
        to control set restrictions:        7,885 out of 149,760  5%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
```

clock, reset, set, and enable signals for a registered element.
The Slice Logic Distribution report is not meaningful if the design is over-mapped for a non-slice resource or if Placement fails.
OVERMAPPING of BRAM resources should be ignored if the design is over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:

Number of bonded IOBs:	7 out of	680	1%
Number of LOCed IOBs:	7 out of	7	100%
IOB Flip Flops:	6		
Number of bonded IPADs:	58		
Number of LOCed IPADs:	10 out of	58	17%
Number of bonded OPADs:	48		

Specific Feature Utilization:

Number of BlockRAM/FIFO:	251 out of	324	77%
Number using BlockRAM only:	227		
Number using FIFO only:	24		
Total primitives used:			
Number of 36k BlockRAM used:	54		
Number of 18k BlockRAM used:	293		
Number of 36k FIFO used:	8		
Number of 18k FIFO used:	16		
Total Memory used (KB):	7,794 out of	11,664	66%
Number of BUFG/BUFGCTRLs:	17 out of	32	53%
Number used as BUFGs:	17		
Number of BUFDSs:	5 out of	24	20%
Number of GTX_DUALs:	12 out of	24	50%
Number of LOCed GTX_DUALs:	12 out of	12	100%
Number of PCIEs:	1 out of	1	100%
Number of PLL_ADVs:	2 out of	6	33%

Average Fanout of Non-Clock Nets: 3.78

Peak Memory Usage: 3242 MB
Total REAL time to MAP completion: 29 mins 46 secs
Total CPU time to MAP completion (all processors): 32 mins 54 secs

9. Status of OpenFlow functions

All the code with “OF1.1” comments have not been simulated or tested.