

An Advanced NetFPGA OpenFlow Switch

Michael Laß
bevan@bi-co.net

Thomas Lücke
tloecke@mail.upb.de

Jörg Niklas
osjsn@niklasfamily.de

Felix Wallaschek
felix@elektronenversand.de

February 17, 2015

Abstract

As part of the project group “On-the-fly Networking for Big Data” we designed an OpenFlow switch using the NetFPGA-10G platform. Therefore we created a hardware design targeted on the Xilinx Virtex-5 that acts as a hardware accelerated dataplane for an Open vSwitch software switch. During our work we reengineered many parts of the NetFPGA framework to increase performance, flexibility and reliability. We also added new features, in particular our design utilizes the onboard SRAM chips and the available high speed interconnect ports. The result is a ready-to-use 10G OpenFlow switch that is well suited for test environments and further development.

Contents

1	Introduction	4
1.1	Architecture Overview	4
1.2	Features and Improvements	4
2	Getting Started	6
2.1	Prerequisites	6
2.2	Projects	7
2.3	Building the Hardware	7
2.4	Installing the Software	8
3	Basic Building Blocks	11
3.1	AXI-4 Stream Bus	11
3.2	Packet FIFO	14
3.3	QDR2-SRAM Interface	20
4	Hardware IP Cores	23
4.1	10G-Interface	23
4.2	Interconnect	26
4.3	Input-Arbiter	30
4.4	Output Queue	34
4.5	PCI Express Interface and DMA Engine	38
4.6	Layer2-Switch	40
4.7	OpenFlow Switch	42
4.8	TCAM	49
4.9	CAM	55
4.10	Further Cores	59
5	Software	60
5.1	The Linux Network Interface Card Driver	60
5.2	The Big Picture: How to Build an OpenFlow Switch	61
5.3	The Library <i>sdn_dataplane</i>	62
6	Notes for Future Work	68

1 Introduction

In the context of the project group “On-the-fly Networking for Big Data” [1] at the University of Paderborn (*UPB*) we built an OpenFlow switch based on the NetFPGA-10G platform [2]. While we used parts of the standard framework [3], most of the cores were developed from scratch to achieve certain goals. In addition to the hardware implementation we also integrated the result into Open vSwitch [4] to create a ready-to-use OpenFlow switch. This documentation gives an overview of the created framework, the developed hardware IP cores, the software implementation and the projects that we realized using this framework.

1.1 Architecture Overview

Figure 1.1 gives an overview over the basic architecture.

On the NetFPGA-10G card there are four 10G Ethernet ports available. Each of these ports is handled by an own instance of our 10G-Interface core. Additionally there are two Samtec high speed ports available, of which we use one port to allow interconnecting two cards. Similar to the 10G ports this port is handled by an instance of our Interconnect core. A third type of interface is the DMA that allows communication with the host PC via PCI express.

An input arbiter chooses data from an incoming port to be processed. The processing itself is done by the User Custom Logic, which can be modified to alter the behavior of the card. Outgoing data is stored in the output queue which acts as a buffer and distributes the data back to the appropriate ports.

While the basic architecture is very similar to the one of the official framework, it should be noted that the AXI4-Stream buses in both frameworks use the user definable bits (tuser) differently. Therefore compatibility between the two frameworks is not always given.

1.2 Features and Improvements

By designing a new framework, several new features could be implemented and functionality could be improved. The most important enhancements are listed below:

Jumbo Frame Support: Several buffers were eliminated that were too small to hold Jumbo Frames. The whole framework was designed with Jumbo Frames in mind, so there is full support for frames up to 9000 bytes.

Larger input- and output queues: The size of the input buffers were increased to be able to hold at least two Jumbo Frames. The output queue utilizes external SRAM to significantly increase the size of the output buffer.

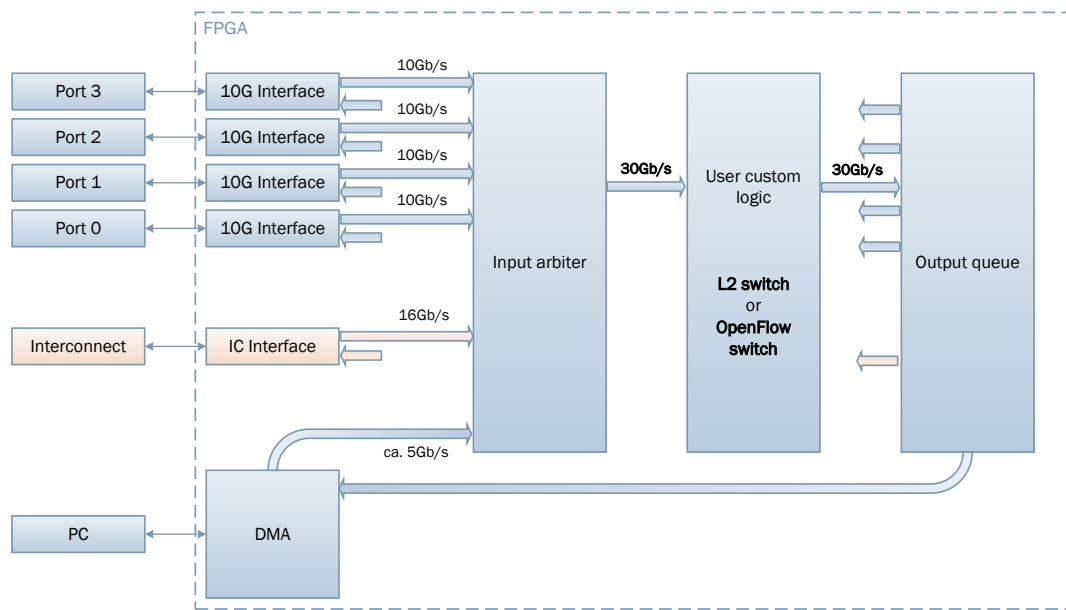


Figure 1.1: UPB Framework

Flow control: Depending on the filling level of the input queues, Ethernet pause frames are sent to implement flow control.

Driver ported to current Linux kernel techniques: The driver was ported to the Linux New API and can be compiled on all currently released kernel versions.

Support for multiple cards in one host PC: Installed NetFPGA-10G cards are now enumerated correctly by the Linux driver, so that installing more than one card is possible.

Support for the Samtec high speed port: Support for the two high speed connectors on the board was added which allows using them as a fast interconnect between multiple cards.

2 Getting Started

This sections describes the first steps to build and use the UPB NetFPGA framework and the provided projects.

2.1 Prerequisites

The framework and all projects were developed on a system using the following software versions. Building and using the hardware IP cores and the software might also be possible on different systems. However tests were only done with the listed versions.

- Ubuntu 14.04
- Linux Kernel 3.14
- Synplify Premier DP 2012.09-SP1
- Xilinx ISE 14.7
- Xilinx XPS 14.7

Additionally to the software listed above, licenses for the following IP cores are needed:

- Xilinx 10G MAC
- Xilinx Aurora

To be able to build the software part, the following libraries and tools are needed:

- A working build system with GNU make, gcc, g++
- Development version of the Boost C++ library 1.55
- Linux kernel header files
- To build the software documentation: Doxygen and graphviz

2.2 Projects

Two different projects using the UPB NetFPGA framework were realized and can be used in practice or for further development.

Layer-2 Learning Switch: The layer-2 learning switch acts as a simple Ethernet switch. When a packet arrives the MAC address of the sender is extracted from the packet and the combination of MAC address and port is stored in a hashmap, using one of the on-board SRAM chips. The destination port for a packet is looked up in this hashmap and the packet is forwarded accordingly. If a destination is not known, the packet is broadcasted to all ports.

This project is very well suited for new developers that want to experiment with the UPB NetFPGA framework. It is less complex than the OpenFlow project and therefore needs less time for synthesis and *Place and Route*. As a consequence it is also well suited for testing new or modified input and output cores or other parts of the datapath.

The project is located under `/contrib-projects/upb_l2switch`.

OpenFlow Switch: The OpenFlow switch consists of a hardware part, which provides a hardware accelerated dataplane for software defined networks, and a software part which implements support for this dataplane in Open vSwitch.

The project is located under `/contrib-projects/upb_openflowswitch`.

2.3 Building the Hardware

To build the hardware, first the environment has to be set up accordingly:

```
source /path/to/ISE/settings64.sh
export SYNPLIFY_CMD=/path/to/synplify/bin/synplify_premier_dp
export LM_LICENSE_FILE=port@license.server
```

After this you can run *make* in the baselfolder of the NetFPGA repository using the following targets:

make upb: Build all IP cores necessary to build the UPB projects and build both of the UPB projects afterwards. The resulting bitfiles can be found in `/contrib-projects/upb_l2switch/hw/bitfiles` and `/contrib-projects/upb_openflowswitch/hw/bitfiles`.

make upb_clean: Cleanup UPB projects and the IP cores.

make upb_cores: Build only the IP cores necessary to build the UPB projects.

make upb_cores_clean: Cleanup the IP cores necessary to build the UPB projects.

make upb_projects: Build only the UPB projects. It is necessary to build the cores before.

make upb_projects_clean: Cleanup the UPB projects.

The software part is always built whenever the IP cores are built.

The resulting bitfiles can be flashed using iMPACT. If you already have one of the projects and the corresponding Linux driver running, you can also flash the images into the flash chips of the NetFPGA card and configure the card to program itself from this chip. See the NetFPGA-10G wiki [5] for more details on this. The following example shows how to flash the OpenFlow image into flash A of the first NetFPGA card installed in the host PC.

```
lib/sw/contrib/drivers/nf10_upb_dma_v1_00_a/bin/nf10_configure \
/dev/nf10a -f a -b \
contrib-projects/upb_openflowswitch/hw/bitfiles/upb_openflowswitch.bin
```

2.4 Installing the Software

The software consists of two parts, the Linux NIC driver and Open vSwitch. This section describes how to build and install these components.

2.4.1 Loading and Installing the NIC Driver

The compiled driver can be found in the following directory:

```
lib/sw/contrib/drivers/nf10_upb_dma_v1_00_a/src/nic_driver
```

Loading the driver can be done by using *insmod*:

```
insmod nf10.ko
```

The Linux NIC driver can also be installed permanently using *make*:

```
make install
```

2.4.2 Building and Installing Open vSwitch

Before our extended version of Open vSwitch can be compiled, the environment variable `SDN_DATAPLANE_DIR` has to be set to the directory of the library *libsdn_dataplane*:

```
export NF_ROOT=/REPLACE_WITH_NETFPGA_ROOT_DIR
export SDN_DATAPLANE_DIR=\
${NF_ROOT}/lib/sw/contrib/drivers/nf10_upb_sdn_dataplane_v1_00_a
```

Next, change the directory to the root of the Open vSwitch source. To check if all dependencies for building are met, execute the following command:

```
dpkg-checkbuilddeps
```

If there is no output, all dependencies are met. Otherwise the missing pieces will be printed. Now you can build a *debian* package:


```
DEB_BUILD_OPTIONS='parallel=8' fakeroot debian/rules binary
```

This starts a parallel build using 8 processes. You can modify the concurrency to meet the number of CPU cores. To later clean the source directory after building a *debian* package, execute the following command:

```
fakeroot debian/rules clean
```

To install the datapath kernel module you additionally need the tool *dkms*. The installation of Open vSwitch is done as follows:

```
cd ..
dpkg -i \
openvswitch-common_2.3.1-1_amd64.deb \
openvswitch-switch_2.3.1-1_amd64.deb \
openvswitch-datapath-dkms_2.3.1-1_all.deb
```

This command can be used to uninstall Open vSwitch. It also removes all settings:

```
dpkg -P openvswitch-common openvswitch-switch openvswitch-datapath-dkms
```

After the successful installation and a reboot of the machine, Open vSwitch is started automatically. Open vSwitch stores its configuration in a persistent database. For the initial setup we used the following setup script:

```
#!/bin/bash

ovs-vsctl --may-exist add-br nf10a-bridge
ovs-vsctl --may-exist add-br nf10b-bridge

for i in {0..5}
do
    ovs-vsctl --may-exist add-port nf10a-bridge nf10av$i
    ovs-vsctl --may-exist add-port nf10b-bridge nf10bv$i
done

ovs-vsctl set-controller nf10a-bridge tcp:127.0.0.1:6633
ovs-vsctl set-controller nf10b-bridge tcp:127.0.0.1:6633
```

This setup assumes that two NetFPGA cards are installed and the OpenFlow controller is installed on the same machine. To use the NetFPGA's network interfaces with Open vSwitch, they have to be activated. If you like to do this manually, you can use the following script:

```
#!/bin/bash

for i in {0..5}
do
    ifconfig nf10av$i up
    ifconfig nf10bv$i up
done
```

After these steps, Open vSwitch should run with the NetFPGA acceleration activated. It makes sense to restart the Open vSwitch daemon as a regular process to see the log messages:

```
killall ovs-vswitchd  
ovs-vswitchd
```

You should see a lot of messages regarding the NetFPGA (e.g. CAM/TCAM found...). For our first tests we used the POX controller[6]. POX can easily be downloaded and installed:

```
git clone https://github.com/noxrepo/pox.git  
cd pox  
./pox.py openflow.of_01 --port=6633 log.level --DEBUG forwarding.l2_learning
```

3 Basic Building Blocks

In this chapter some basic building blocks are described that are used in several IP cores and projects.

3.1 AXI-4 Stream Bus

In our datapath we consistently use a custom AXI4-Stream bus. In table 3.1 the signals of this bus are shown as they are used by the master. Table 3.2 shows the slave interface.

Name	Direction	Description
m_axis_tdata[255:0]	out	The Ethernet data. The first byte which is transferred over the line is in [7:0].
m_axis_tkeep[31:0]	out	Byte valid signal. tkeep[0] belongs to tdata[7:0]. Associated bytes are valid in the stream. Only the last transfer may contain contiguous unused bytes in the upper part of the word.
m_axis_tvalid	out	Current transfer cycle contains valid data. tvalid will not be deasserted until the packet is completely transferred (the slave can expect a contiguous data stream for one packet).
m_axis_tready	in	Flow control from slave to master
m_axis_tlast	out	The Ethernet packet ends with this transfer cycle.
m_axis_tuser_in_port[2:0]	out	Incoming port (binary coded). This signal is valid during the whole packet transfer.
m_axis_tuser_in_vport[2:0]	out	Incoming virtual port (binary coded). Default vport is 0. This signal is valid during the whole packet transfer.
m_axis_tuser_out_port[7:0]	out	Bitfield of the outgoing port. Default value is 0. Multicasting is enabled by setting multiple bits. This signal is valid during the whole packet transfer.
m_axis_tuser_out_vport[7:0]	out	Bitfield of the outgoing virtual port. Ports without virtual port functionality ignore this value. Default value is 0. Multicasting is enabled by setting multiple bits. This signal is valid during the whole packet transfer.

m_axis_tuser_packet_length[13:0]	out	Packet length of the Ethernet packet in bytes. This signal is valid during the whole packet transfer.
----------------------------------	-----	-------------------------------------------------------------------------------------------------------

Table 3.1: AXI4-Stream bus signals (master interface)

Name	Direction	Description
s_axis_tdata[255:0]	in	See master interface.
s_axis_tkeep[31:0]	in	See master interface.
s_axis_tvalid	in	See master interface.
s_axis_tready	out	See master interface.
s_axis_tlast	in	See master interface.
s_axis_tuser_in_port[2:0]	in	See master interface.
s_axis_tuser_in_vport[2:0]	in	See master interface.
s_axis_tuser_out_port[7:0]	in	See master interface.
s_axis_tuser_out_vport[7:0]	in	See master interface.
s_axis_tuser_packet_length[13:0]	in	See master interface.

Table 3.2: AXI4-Stream bus signals (slave interface)

The *tvalid* and *tready* signals follow the standard AXI4-Stream logic. Because only contiguous parts of *tkeep* are allowed to be 0 and these have to be at the upmost position, a binary representation of “number of ones -1” is a more efficient encoding. This encoding is often used in our design when data has to be stored in FIFOs. Also the value is only allowed to contain zeros when also *tlast* is set, which also is often exploited for efficient storage of data.

Table 3.4 shows the bit positions of the ports used in our design in *tuser_out_port*. Because packets only arrive at a single port, *tuser_in_port* directly represents the corresponding port number.

Bit	Port Description
0	NetFPGA-10G Port 0 (close to the PCIe header)
1	NetFPGA-10G Port 1
2	NetFPGA-10G Port 2
3	NetFPGA-10G Port 3 (topmost port)
4	NetFPGA Interconnect port 0 (close to the slot bracket)
5	DMA (PC Ports; setting bits of “vport” is mandatory)
6	reserved
7	reserved

Table 3.3: Bit positions of ports in *tuser_out_port*

The DMA engine uses virtual port IDs to associate a packet to the network interfaces on the PC side. These virtual port (vport) numbers are used:

Bit	Port Description
0	/dev/nf10xv0
1	/dev/nf10xv1
2	/dev/nf10xv2
3	/dev/nf10xv3
4	/dev/nf10xv4
5	/dev/nf10xv5
6	/dev/nf10x (general purpose network interface)

Table 3.4: Bit positions of ports in *tuser_out_vport*

3.2 Packet FIFO

In our OpenFlow framework the bus width has to be increased from 64bit to 256bit in each 10G input core, as described later in this document. Because this introduces idle cycles on the bus, a FIFO is required on the input side of each port that is capable of holding complete packets to ensure a continuous data flow for each single packet in later stages of the pipeline. It also has to allow storage of per-packet metadata like the packet length. This data should only be stored once per packet and be available on the output side at all time. Another important requirement is the ability to drop packets after data was already fed into the FIFO. This is necessary because information about the integrity of a packet is only available after the whole packet was received and the switch needs the ability to drop corrupt packets.

In the following section the design of this packet-aware and transactional FIFO (short: PACKET_FIFO) and its usage will be described.

3.2.1 Design Overview

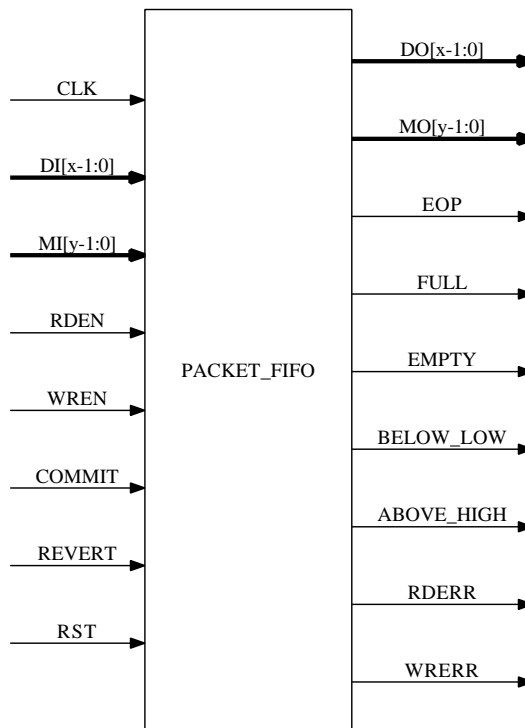


Figure 3.1: PACKET_FIFO Block

The PACKET_FIFO is designed to exactly fit the needs of our switch architecture, and for that it provides some special in- and outputs. Figure 3.1 shows our PACKET_FIFO block. The main differences to already existing FIFO primitives are:

- Data input (DI) and output (DO) width parameterizable.
- Additional meta data input (MI) and output (MO) for storage of one set of data for each packet. Input and output width is parameterizable.
- FIFO depth parameterizable separately for data and metadata storage.
- COMMIT and REVERT ports for transactional functionality.
- EOP port to signal end of packet.

When instantiating the PACKET_FIFO several attributes can be passed to configure the FIFO. Table 3.5 shows the available attributes and briefly describes their meaning.

Attribute Name	Type	Default	Notes
DATA_WIDTH	Integer	256	Specifies width of DI and DO. Internal width will be one bit larger for storage of <i>end of packet</i> information.
METADATA_WIDTH	Integer	16	Specifies width of MI and MO.
DATA_DEPTH	Integer	10	Sets the number of storable entries in the data part of the FIFO. When set to n the number of entries will be 2^n .
METADATA_DEPTH	Integer	10	Sets the number of storable entries in the metadata part of the FIFO. When set to n the number of entries will be 2^n . For optimal performance this should be scaled so that the data storage is filled up first.
LOW_THRESHOLD	Integer	256	Specifies how many entries have to be stored in the data part of the FIFO before BELOW_LOW is deasserted.
HIGH_THRESHOLD	Integer	768	Specifies how many entries have to be stored in the data part of the FIFO before ABOVE_HIGH is asserted.


Table 3.5: PACKET_FIFO Attributes

Table 3.6 gives an overview about all available ports of the PACKET_FIFO. Information about timing of these signals can be found in section 3.2.2 of this document.

Port Name	Direction	Signal Description
CLK	Input	Clock for read and write operations.
DI[$x-1:0$]	Input	Data input. Width x can be set via DATA_WIDTH.
MI[$y-1:0$]	Input	Meta data input. Width y can be set via METADATA_WIDTH. Data on this port is stored when and only when COMMIT and WREN are asserted.
RDEN	Input	Read enabled. This moves the read counter by one position and makes room for new data in the FIFO. Asserting this signal while EMPTY is high results in an error which is indicated by RDERR.
WREN	Input	Write enabled. When asserted data from DI is stored in the FIFO. Asserting this signal while FULL is high results in an error which is indicated by WRERR. In this case no data was stored. Has no effect when REVERT is asserted at the same time.
COMMIT	Input	Has to be asserted at the end of each packet together with WREN when writing the last part of the data. Content of MI gets stored in the FIFO and dropping it is not possible anytime after this operation. Must not be asserted at the same time as REVERT or will otherwise lead to an error which is indicated by WRERR.
REVERT	Input	Leads to a drop of the currently written packet. Must not be asserted at the same time as COMMIT. Input on WREN, DI and MI is ignored.
RST	Input	Synchronous reset signal.
DO[$x-1:0$]	Output	Data output.
MO[$y-1:0$]	Output	Metadata output. This output will always be valid while reading contents of a packet from DO.
EOP	Output	End of packet reached.
FULL	Output	The FIFO is full. Asserting WREN will lead to an error indicated by WRERR.
EMPTY	Output	The FIFO is empty. Asserting RDEN will lead on an error indicated by RDERR.
BELOW_LOW	Output	Asserted when less than LOW_THRESHOLD entries are stored in the data part of the FIFO.

ABOVE_HIGH	Output	Asserted when more than HIGH_THRESHOLD entries are stored in the data part of the FIFO.
RDERR	Output	An error occurred while trying to read data.
WRERR	Output	An error occurred while trying to write data.

Table 3.6: PACKET_FIFO Ports



BELOW_LOW / ABOVE_HIGH

BELOW_LOW and ABOVE_HIGH only represent the filling level of the block RAM that is used to store packet data. No information about the filling level of the metadata storage or additional output registers are taken into account. Therefore one must not use these ports to determine the exact amount of available space or available data in the FIFO. Under certain circumstances it can be used as a rough estimate though.

To save resources and keep the internal logic as simple as possible, the PACKET_FIFO has some limitations:

- First Word Fall Through (FWFT) mode only.
- COMMIT must not be asserted on the first chunk of data. This means each packet has to be at least two clock cycles long.
- No support of asynchronous operation.
- There is no ECC functionality.
- There are no WRCOUNT and RDCOUNT ports.

3.2.2 Timing

In this section some typical situations and the timing behavior of the PACKET_FIFO are shown.

Figure 3.2 shows writing the end of a packet to an empty FIFO. First data is written by setting WREN high while applying the data at DI. When writing the last part of data COMMIT is set high while applying the metadata on MI. Three clock cycles later the start of the written packet appears at DO, the metadata at MO and EMPTY is deasserted.

Figure 3.3 shows reading the last packet from the FIFO. While RDEN is asserted the output at DO changes each clock cycle. When the end of a packet is reached EOP is asserted. After confirming the read of the last data block via RDEN EMPTY is set high. At this point RDEN has to be set low before the next rising clock edge or otherwise RDERR will be asserted to signal a read error.

Figure 3.4 shows writing to the FIFO until its storage capacity is reached. D_{x+1} is the last set of data that is stored. At the same time FULL is set high. At this point either WREN has to be

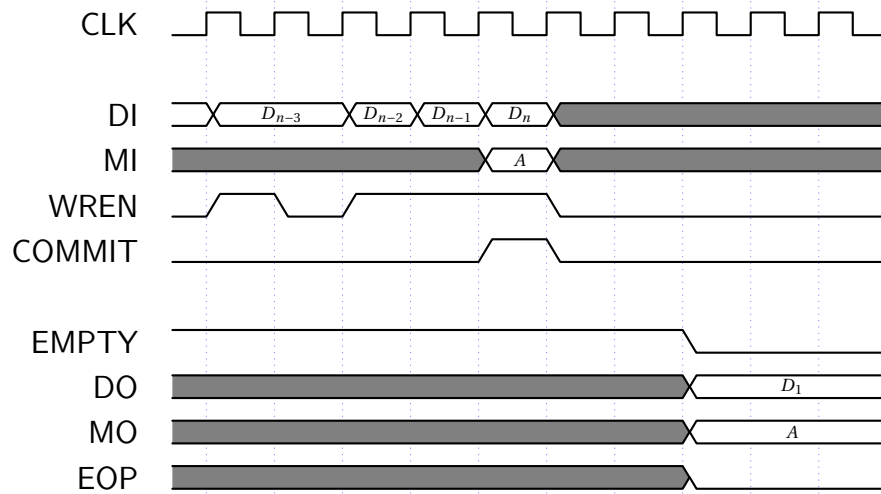


Figure 3.2: Writing end of packet to empty FIFO

set low or REVERT has to be set high before the next rising clock edge, otherwise WRERR will be asserted to signal a write error. When REVERT is asserted the data of the currently written packet is removed from the FIFO resulting in FULL to be set low again.

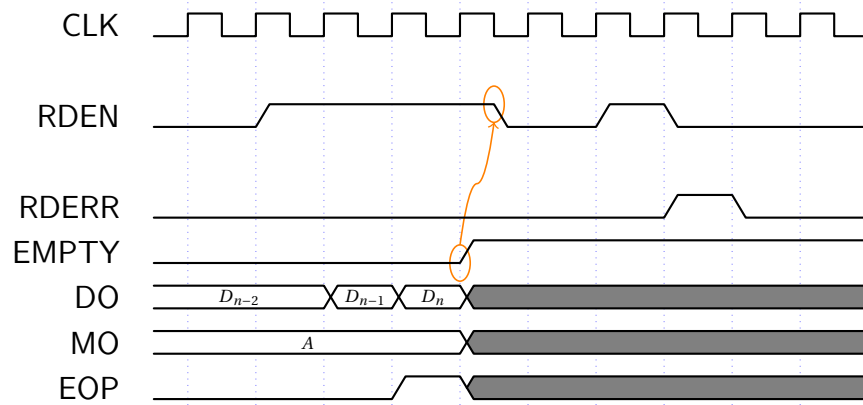


Figure 3.3: Reading last packet from FIFO

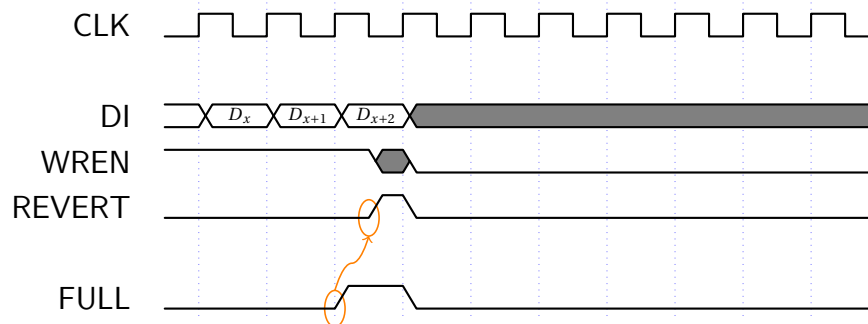


Figure 3.4: Filling up FIFO and revert operation

3.3 QDR2-SRAM Interface

The NetFPGA-10G board features 3 QDR II SRAM memory chips¹ with a capacity of 9 MBytes each (2 Mbit x 36). They can operate at 300 MHz. QDR II memories have independent read and write data ports which eliminates the need of bus turnaround cycles. It is possible to read and write to the memory in the same bus cycle. During each clock cycle two data bits are transferred (Double Data Rate - DDR). This means that a memory bandwidth of 21.6 Gb/s for read and write (43.2 Gb/s combined bandwidth) can be achieved per chip. In contrast to DRAM and also Reduced Latency DRAM (RL-DRAM) this also holds for random accesses and not only for bursts.

Due to these features, QDR II SRAM is specially well suited for look-up tables in typical network hardware devices. Sometimes it also makes sense to use these devices as Ethernet packet buffers, as they can also store smallest packets without a throughput penalty. DRAM designs would need large additional buffers which can absorb the access time to different memory locations in case of small packets. The overall design would be more complicated and more expensive, especially on FPGAs.

In our project we use one QDR II SRAM memory as a MAC table for the layer-2 switch and two memory devices as packet buffers in the output queue.

3.3.1 Features of the Core

- Support for 300 MHz (DDR)
- Fixed read latency of 5 clock cycles
- Automatic timing adjustment of the IO signals
- Comprehensive memory chip test

3.3.2 Design Details

Synchronous SRAM memories provide a simple interface which can be translated into a typical On-Chip Bus on the FPGA using mainly some combinatorial logic. The challenge is to initialize the FPGA core and set up a valid timing on the IO pins. Our design uses the serializers and deserializers (*ISERDES*, *OSERDES*) from the Virtex-5 FPGA together with the built-in switchable delay elements (*IODELAY*) in the IO blocks.

The FPGA controls the QDR II SRAM as a master. The 4:1 output serializers (*OSERDES*) of the FPGA are used to drive the clock and all other input signals of the memory chip with a fixed timing. The clock signal, which drives the SRAM memory, is phase shifted to the data signals. Here, the 4:1 serializers and a 90 degree phase shifted clock from a PLL are sufficient to provide a valid output timing.

QDR II SRAM memories have an echo clock output for the output data (the read data) called the *Echo Clock*. The first challenge is to find a valid phase alignment between the *Echo Clock*, which is also the capture clock for the input flip-flops on the FPGA, and the data output of

¹Cypress CY7C1515JV18-300BZXC

the SRAM (called q), such that the setup time of the input flip-flop on the FPGA is not violated. Second, we need to find a valid phase alignment between both previously named signal groups and the internal clock domain of the FPGA to get the read data synchronized with the On-Chip Bus.

To find a valid timing, the core uses a state machine which writes and reads different characteristic bit patterns while shifting the timing. By using some craftily chosen bit patterns, it is possible to eliminate the dependencies between different delay groups. The state machine first finds a valid phase alignment between the *Echo Clock* and q and afterwards the correct phase alignment between *Echo Clock*, q and the FPGA's main clock domain. During each of these two steps, a so-called *valid window* is searched. Inside this *valid window* the timing is properly working and the written data was successfully reread. After the window is detected, the middle of this valid window is chosen to set the delay elements. Using this timing-adjustment approach, it is possible to perfectly adapt to different chip and PCB versions. The setting of the timing is only done once after the FPGA was loaded. To compensate the *IODELAY* elements against temperature differences it is mandatory to use the *IODELAYCTRL* primitive. This is done inside the *Clock Generator* core which will be discussed later.

After adjusting the timing, a complete memory test with different patterns is executed. If this test is successful, the *ready* signal is asserted, otherwise the *error* signal is asserted.

3.3.3 Other Solutions

Another popular solution is the Xilinx Memory Interface Generator (MIG). This is a free software tool that generates a memory controller core. Unfortunately this tool only supports memory frequencies of up to 250 MHz for QDR II. The universal nature of this product (MIG also supports many other different memory types) has some drawbacks. The interfaces to the core are FIFO based. The minimum read latency is 18 clock cycles (not deterministic). In contrast, our approach provides a fast and slim solution with a deterministic timing which is better suited for low-latency table lookups and packet buffers.

3.3.4 Interface Description

Name	Direction	Description
clk	in	Clock input
clk2x	in	Clock input. The frequency has to be twice as high as <i>clk</i> .
clk2x90	in	Clock input. 90 degree shifted clock of <i>clk2x</i> .
reset	in	Active high reset
qdr	in/out	SystemVerilog interface to the QDR II memory chip
ready	out	Asserted as soon as the core is ready and the memory is functional
error	out	Asserted when there is a memory error detected during initialization

rd	in	Assert synchronously to r_addr to read from the memory
r_addr[18:0]	in	The address to read from
r_valid	out	Asserted as soon as the read data is available (not really needed as the timing is deterministic)
r_data[143:0]	out	The read data is available here 5 clock cycles after assertion of rd
wr	in	Assert synchronously to w_addr and w_data to write to memory
w_addr[18:0]	in	The address to write to
w_data[143:0]	in	The data to write

Table 3.7: QDR II memory controller core interface

To communicate with the memory chip, the user has to wait for *ready* or *error* to be asserted. When the *error* signal is asserted the memory cannot be accessed. As soon as *ready* is asserted, the user can read by asserting *rd* while setting the *r_addr* or write by asserting *wr* while setting *wr_addr* and *wr_data*. It is possible to issue a write and a read command to the same address in the same clock cycle. In that case the written data will already be reread and is available after 5 clock cycles.

3.3.5 Simulation

The QDR II interface core can also be simulated. The *Output Queue* core and the *layer-2 switch* core have example test benches which show how this can be done. We use a VHDL functional model from the vendor of the memory chips (Cypress Semiconductor) which we improved so that also the timing adjustment process can be simulated. It does not make sense to simulate the extensive memory test as this would take too much time. There is a parameter *simulation_speed_up* which can be set during the module instantiation. If it is set to '1' the simulation will run through the initialization routines in a more reasonable time.

3.3.6 Future Developments

The core can easily be adapted to other QDR II SRAM chips as all widths and other parameters are defined as SystemVerilog parameters. The default values are only valid for the NetFPGA-10G board. If you want to use the core for a different board with longer PCB traces, it might be necessary to increase the parameterizable read latency beyond 5 clock cycles to get a valid timing.

4 Hardware IP Cores

The UPB NetFPGA framework contains many new cores that were developed from scratch. In this section these cores, their basic design and their functionality are described.

4.1 10G-Interface

In our framework we are using the *Xilinx 10G MAC* (v10.3). To provide an AXI4-Stream interface like later core versions do, we have written a wrapper for this core. This is why our *10G interface* actually consists of two different cores: the wrapper for the *Xilinx 10G MAC* core and a *10G Input* core that processes incoming packets. Due to this, the MAC can easily be changed in future by just replacing the wrapper core (*10G interface*) with a new MAC supporting an AXI Stream interface.

When the *10G Input* core receives input from some MAC it converts the stream width to fit the internal stream width, stores the input in a *Packet FIFO* and forwards it to the input arbiter. In particular it makes use of the *Packet FIFOs* commit/revert functionality to discard incorrectly received packets and to limit the packet length in case of invalid packets.

To cope with congestion pause requests are used. These requests are based on the watermark signals of the *Packet FIFO*. Whenever a pause request is to be sent, the *flow_req* signal is asserted. If it is asserted, the signal *flow_value* contains a value according to IEEE 802.3.

The *10G Input* core is also able to forward packets coming from the output queue to a MAC or the described wrapper core. Therefore it reduces the width of the AXI4-Stream to a size of 64.

4.1.1 Design Overview

The *10G Input* core is equipped with four AXI streams as shown in Figure 4.1: two AXI4-Streams connecting to the *10G MAC* (one input and one output), one to send packets to the arbiter and one to receive packets from the output queue. The AXI4-Streams shared with the MAC and the AXI stream from the output queue operate with the same clock as the MAC (input via *clk156*). This clock is also made available to other cores via the *output_queue_clk* output. The AXI4-Stream to the arbiter operates with the clock from the *axi_aclk* input.

Table 4.1 shows the attributes with which the core can be configured.

Attribute Name	Type	Default	Notes
C_PORT_NUMBER	Integer	0	Physical port this core is connected to

C_INPORT_WIDTH	Integer	3	Width of the AXI tuser inport lanes (arbiter and output queue).
C_OUTPORT_WIDTH	Integer	8	Width of the AXI tuser outport lanes (arbiter and output queue).
C_PACKET_LENGTH_WIDTH	Integer	14	Width of the AXI tuser packet length lane (arbiter and output queue).
C_MAX_PACKET_LENGTH	Integer	10000	The maximal allowed packet length in bytes. Longer packets are dropped.
C_AXIS_DATA_WIDTH	Integer	256	Size of AXI tdata (arbiter and output queue). Should be 256, or another multiple of 64.
C_AXI_BASE_ADDR	Integer	32'h00000000	Core's base address when connected to AXI4-Lite.
C_AXI_HIGH_ADDR	Integer	32'hFFFFFFFF	Core's high address when connected to AXI4-Lite.

Table 4.1: 10g_input attributes

4.1.2 Frame Transmission

The *10G interface* (wrapper core) can be used to transmit frames via the *Xilinx 10G MAC*. For this purpose the *s_axis* interface is provided. Until the end of the transmission, valid data has always to be present at the input.

If at any time during the transmission *s_axis_tvalid* is deasserted, the transmission may be aborted.

To explicitly abort a transmission the *s_axis_tuser* bit is used. For a normal transmission it has to be set to 0 all the time. If asserted together with *s_axis_tvalid*, the transmission is aborted.

4.1.3 Frame Reception

The *10G interface* (wrapper core) can be used to receive frames via the *Xilinx 10G MAC*. As this core is not able to store any data, a connected core has to be able to accept data in any cycle. Precisely: *m_axis_tready* is assumed to be always 1.

The *m_axis_tuser* bit is used to transfer the good/bad signals from the MAC. Its value is only valid when *m_axis_tlast* is asserted. '0' encodes a good frame, '1' encodes a bad frame.

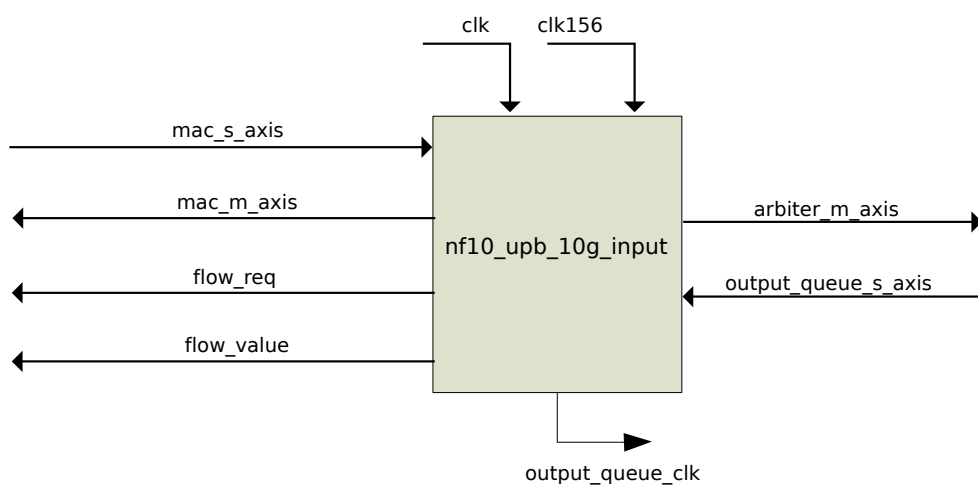


Figure 4.1: 10G Input Block

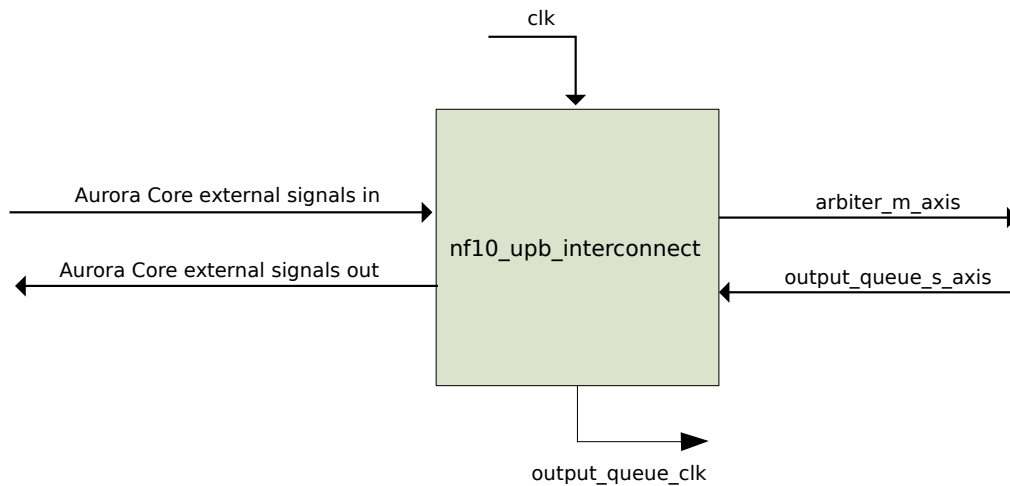


Figure 4.2: Interconnect Block

4.2 Interconnect

The NetFPGA-10G board is equipped with Samtec high speed connectors, which are attached to 10 GTX transceivers on the FPGA. Each transceiver is capable of 6.5 Gbps full duplex. This core is used to manage data exchange between two boards using one of these connectors (of each board). Internally the Xilinx Aurora 8B10B (v5.3) is used to set up the lanes.

To be integrated in a project with other UPB cores, two AXI4-Stream ports are provided (see Figure 4.2). A master stream which outputs received data and a slave stream which accepts data and transmits it to the other card. Like in the *10G Input* core, incoming packets (from the connector) are checked and in case of corruption deleted using the revert functionality of a *Packet FIFO*. To catch packets with bit errors, a 32 bit CRC value is used and sent with every data chunk. Furthermore, too small and too big packets are dropped. The *Interconnect* core will never drop packets due to congestion because of a flow control mechanism. Please note that the AXI slave stream is synchronous to a clock generated by the Aurora core. Like in the *10G Input* module this clock is named `output_queue_clk` as in our design the data comes from the *Output Queue*.

4.2.1 Design Overview

For a first impression of how the interconnect works see Figure 4.3.

As this core uses the Xilinx Aurora core it has to instantiate an Aurora core as well as several other modules. These are encapsulated in a core called *nf10_upb_aurora_input*. In addition to that, this core also calculates and checks the CRC values used for error detection. Since this calculation may cause timing issues, the data is delayed for some clock cycles. This way the retiming feature of Synplify is able to improve the timing significantly. Two lo-

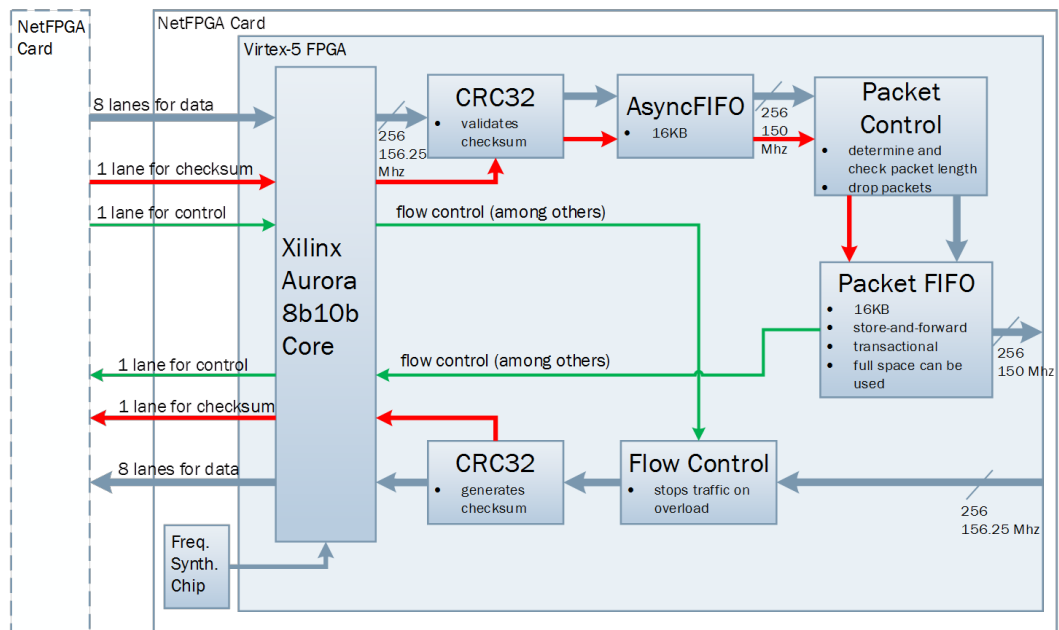


Figure 4.3: Interconnect Design Scheme

cal parameters *TX_DELAY_CYCLES* and *RX_DELAY_CYCLES* are used to control the length of the delay. The submodule *encodeInOneCycle* was introduced to improve timing. The whole *nf10_upb_aurora_input* module uses the clock generated by the Aurora Core.

In the top module received data is written into an asynchronous FIFO and from there read with our system clock. It is then written into a *Packet FIFO* and forwarded to the AXI master stream output signals. Data to transmit is directly fed from the AXI slave signals into the *nf10_upb_aurora_input* module.

In- and output signals besides the two AXI4-Streams and their clocks are connected to the Aurora Core.

Table 4.2 shows the attributes with which the module can be configured.

Attribute Name	Type	Default	Notes
C_PORT_NUMBER	Integer	0	Physical port this module is connected to.
C_INPORT_WIDTH	Integer	3	Width of the AXI tuser inport lanes (arbiter and output queue).
C_OUTPORT_WIDTH	Integer	8	Width of the AXI tuser outport lanes (arbiter and output queue).
C_PACKET_LENGTH_WIDTH	Integer	14	Width of the AXI tuser packet length lane (arbiter and output queue).
C_MAX_PACKET_LENGTH	Integer	10000	The maximal allowed packet length in bytes. Longer packets are dropped.
C_MIN_PACKET_LENGTH	Integer	33	The minimal allowed packet length in bytes. Shorter packets are dropped.
C_AXIS_DATA_WIDTH	Integer	256	Size of AXI tdata (arbiter and output queue). Should be 256.
SIM_GTXRESET_SPEEDUP	Integer	1	See Aurora Core documentation.
C_AXI_BASE_ADDR	Integer	32'h00000000	Module's base address when connected to AXI4-Lite.
C_AXI_HIGH_ADDR	Integer	32'hFFFFFFF	Module's high address when connected to AXI4-Lite.

Table 4.2: nf10_upb_interconnect attributes

4.2.2 Known Problems and Limitations

With our testbed we were only able to get the Aurora cores working reliably with a GTX rate of 2.5 Ghz. This results in a total bandwidth of 16 GBit/s (full duplex). Higher GTX rates led to a high amount of soft and even hard errors. This causes the Aurora core to constantly reset itself (if the channel had been set up) and makes transmissions impossible.

We noticed that some packets are dropped on account of a failing CRC check. At the same time there are no soft errors. This only happens rarely and could be the effect of bit errors during the transmission. But as there are no soft errors this means that all of these bit errors result in valid 8B10B code. However, this event is seldom enough to not cause any noticeable impact on the bandwidth.

4.3 Input-Arbiter

In our design of an OpenFlow switch an n:1 input arbiter is required to send data from multiple AXI4-Stream inputs to one user custom logic core. Although the current NetFPGA-10G project features a simple 4:1 input arbiter, we decided to design and implement one ourselves, as we need the flexibility to add more inputs, as well as more advanced scheduling strategies.

Our designed input arbiter can switch between an arbitrary number of inputs while loosing zero clock cycles per switch. It chooses inputs based on a round robin scheduler, while employing a configurable time-sharing strategy.

4.3.1 Design Overview

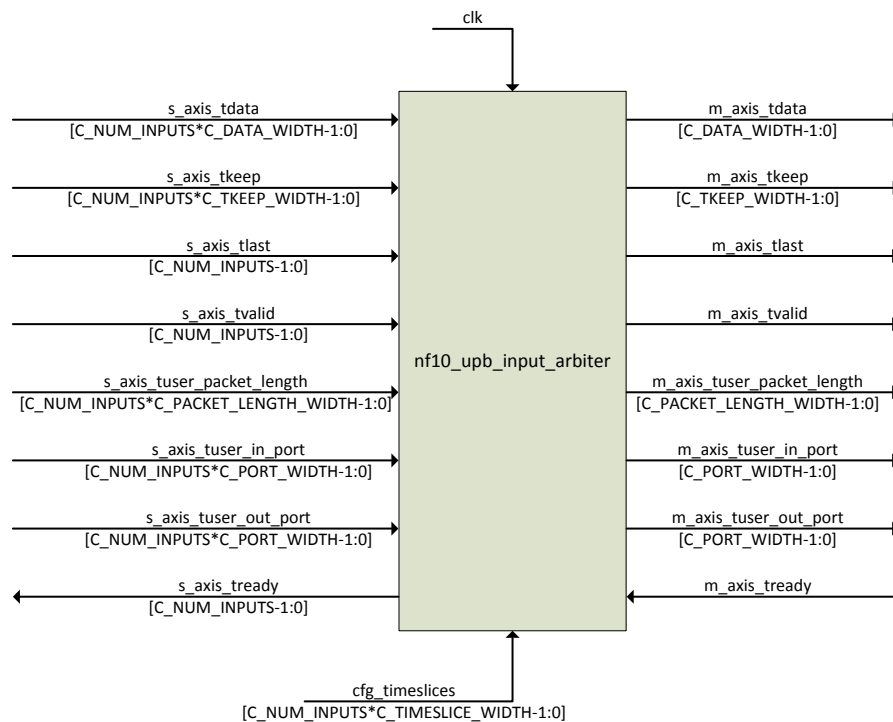


Figure 4.4: Input Arbiter Block

As seen in Figure 4.4, the core features AXI4-Stream slave and master ports. The n slave ports have a combined width of n times master port width, and are combined in one bit-vector. This is due to the verilog restriction which does not allow the number of in-/output ports to be defined by a parameter. A wrapper to split these vectors into different ports per slave stream, which is needed for XPS usage, is provided. In- and output ports of the input arbiter are defined by the AXI4-Stream interface defined for this project group. Information

about timing of these signals can be found in section 4.3.2 of this document. Table 4.3 shows the available attributes that can be used when instantiating the arbiter. The *cfg_timeslices* input is used to configure the timeslices on a per-input basis. A general rule of thumb for the configured timeslices is the logarithm of the bit-length of the greatest receivable packet size divided by the bit-width of the AXI4-Stream data lane.

Attribute Name	Type	Default	Notes
C_DATA_WIDTH	Integer	256	Width of the AXI data lane.
C_PACKET_LENGTH_WIDTH	Integer	14	Width of the AXI tuser packet length lane.
C_TKEEP_WIDTH	Integer	32	Width of the AXI tkeep lane. Should be equal to C_DATA_WIDTH/8.
C_IN_PORT_WIDTH	Integer	3	Width of the AXI tuser inport lanes.
C_OUT_PORT_WIDTH	Integer	8	Width of the AXI tuser outport lanes.
C_NUM_INPUTS	Integer	5	Number of the arbiters AXI slave ports.
C_TIMESLICE_WIDTH	Integer	9	Width of a single timeslice lane.

Table 4.3: input arbiter attributes

4.3.2 Timing

As shown in Figure 4.5, the arbiter introduces a two-cycle delay when forwarding data. If the current input does not provide valid data, the arbiter will change to an input that does (as seen in Figure 4.6) and resets the timeslice of the original input. However the arbiter must not change inputs if there is a packet in transit. In this case (Figure 4.7) the arbiter will wait for the tlast signal before changing inputs. Note that the arbiter will reduce the timeslice of the input while waiting for the tlast signal, however this will only trigger a change of inputs if the timeslice is consumed by the arrival of the tlast signal or if tvalid gets deasserted afterwards. If the selected input sends a continuous flow of packets, the arbiter will change to another input at the next arrival of a tlast signal after the timeslice of that input has been consumed. (See Figure 4.8)

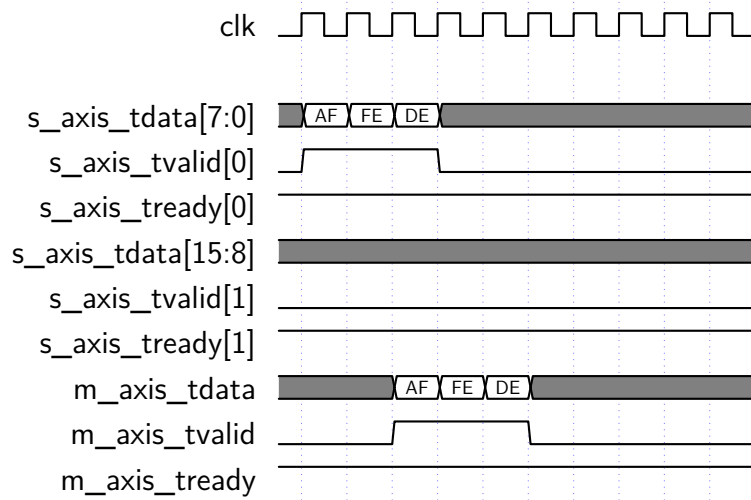


Figure 4.5: Forwarding data through the arbiter

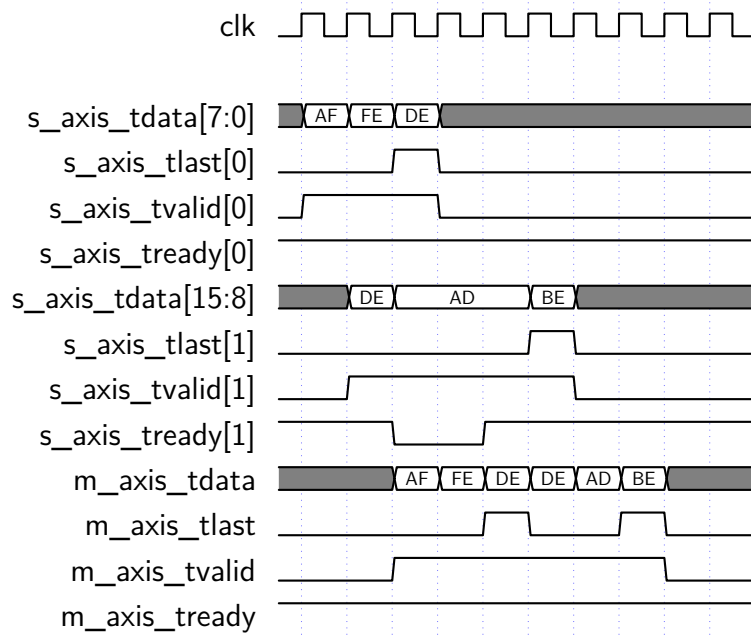


Figure 4.6: Arbiter changing input because of missing data

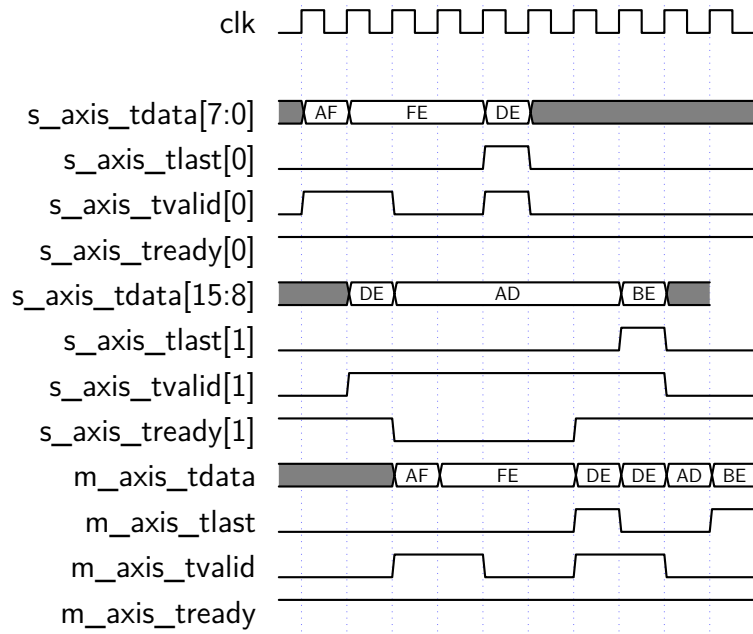


Figure 4.7: Arbiter waiting for tlast signal before changing input

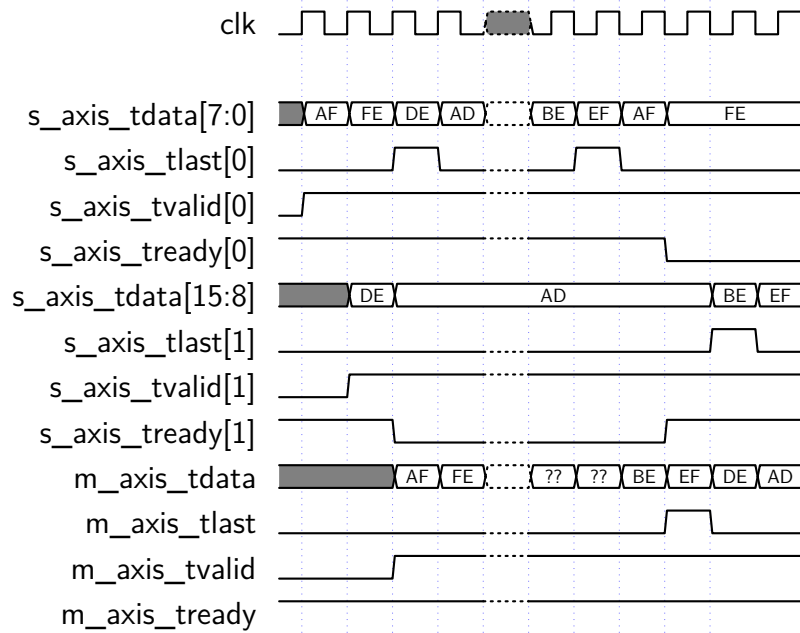


Figure 4.8: Arbiter changing input because of timeslice consumption

4.4 Output Queue

The *User Custom Logic* in the middle of the processing pipeline of the framework can choose to forward a packet to a set of output ports. This also means that congestion may happen at one of the output port queues, as the internal throughput of the framework is significantly higher than the throughput of most output ports. To absorb temporary packet bursts in the case of congestion, an output queue with sufficient buffering capabilities is mandatory.

4.4.1 Features of the Core

- Flexible number of egress ports supported (no limit by design)
- Uses two external QDR II memory chips
- Different buffers for unicasts and multicasts (due to memory bandwidth limitations)
- Unicast output buffer size of 16 MB (with 6 ports: 2.7 MB per port)
- Multicast output buffer parameterizable (default: 16 KB)

4.4.2 Design Details

The internal block RAMs of the FPGA cannot provide a sufficient buffer size for a high performance Ethernet switch. Instead we use two of the external QDR II SRAM chips for the output queue. Because of the strict synchronous design of the QDR II SRAM core, the external memory has exactly the same throughput as the processing pipeline of the framework. It is always possible to simultaneously write and read the internal data stream to and from the external memory.

At a first glance this sounds like the implementation will be very easy. But there are some more details which have to be solved. There may be multicast packets which arrive at the output queue and that have to be forwarded to more than just one port. At some point inside the output queue, the packets have to be duplicated. Either they have to be written to multiple positions in the external memory or they have to be read several times from external memory (which even makes it more complicated). Both solutions lead to the problem, that the memory bandwidth of the external memory is exceeded. On the other hand, multicast (and also broadcast) traffic might only produce a very little amount of traffic. This depends on the use case of the framework. So we decided to use different buffers for multicasts (also broadcasts) and unicasts. Unicasts go to external memory, multicasts are buffered in block RAM.

After separating unicasts and multicasts into different buffers, the correct order of the packets has to be recovered. Although there might not be so many situations where a reordering of unicast and multicast packets might lead to a problem, we did not want to programmatically allow this. In fact, obtaining the correct packet order is the most complex part of the output queue core.

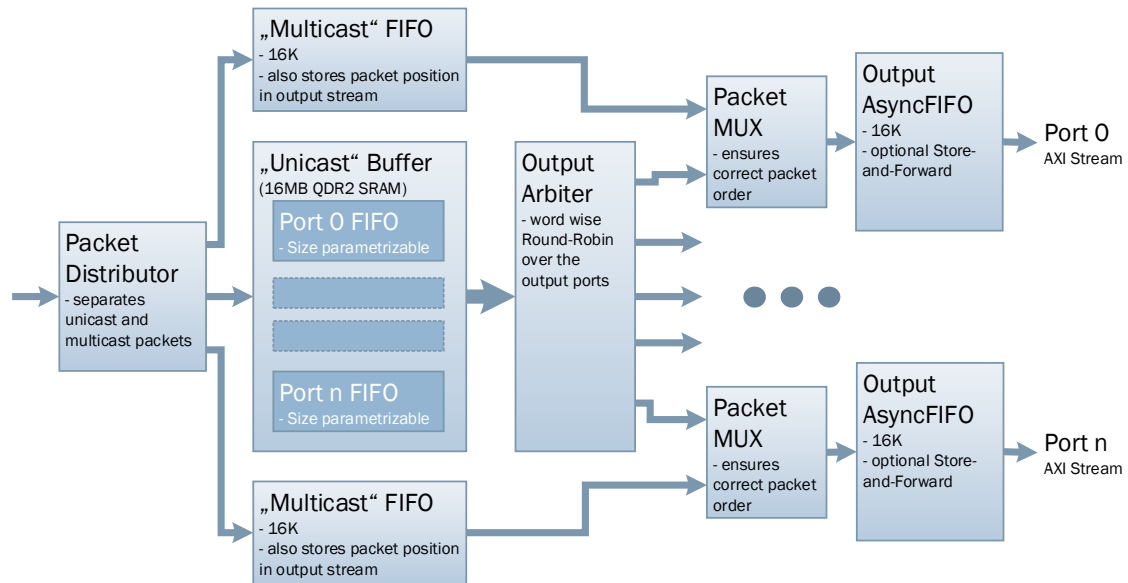


Figure 4.9: Design of the Output Queue

Figure 4.9 shows the structure of the output queue. Packets from the *User Custom Logic* arrive at the *Packet Distributor*. Here, unicast and multicast packets are separated and forwarded to the block RAM based “multicast FIFOs” and to the external memory based unicast buffers. Each egress port has an associated buffer area in the external memory and a separate multicast FIFO. The *Output Queue* core counts every incoming word from the *User Custom Logic*. The counter values are attached to every word fed into the multicast FIFOs. Only the smaller multicast FIFOs get this word count, not the large unicast buffer. This word numbering is needed to recover the correct ordering of the Ethernet packets (or rather the correct word order as we internally operate on a word basis). Before bringing unicast and multicasts together again, the *Output Arbiter* selects one external port’s buffer area in the external memory. The arbiter does this by word wise round robin selecting over the output port buffers. The *Packet MUX* entities also count every word they forward to the egress port. By comparing these counter values with the counter values delivered over the multicast FIFOs, the original packet order can be recovered. As the external memory has a larger latency than the internal block RAMs, there is a little penalty of a few clock cycles when switching back from a multicast to a unicast packet. The pipelines in the output stages of the *Output Queue* core always try to prefetch unicast packets and if it turns out that a multicast packet has to follow next, the prefetched data is dropped. This design concept follows the assumption that most packets are unicasts and thus it is always tried to prefetch data from external memory in bursts to prevent the output pipeline from stalling. Additional FIFOs between the *Output Arbiter* and the *Packet MUX*es could further absorb this switching penalty. We decided not to include additional FIFOs as the *Output Queue* core already consumes a large part of the FPGA block RAMs. The minimum size of one FIFO is 16 KB at 256 bit (8 block RAMs) and we

would need such a FIFO for every egress port.

The *Output Queue* also features asynchronous FIFOs which are connected to the ports. The port cores (e.g. 10G interface) have to provide the clock for this FIFO. The reason for this design is that we wanted to provide clean AXI4-Stream buses. For the functionality of the pipeline in the output stage we needed additional information about the filling level of the output FIFO (the “almost full” flag). So we included the output FIFOs in the *Output Queue* core.

4.4.3 Other Solutions

The default implementation of an output queue in the original NetFPGA project (*nf10_bram_output_queues*) is based on block RAMs. The buffer size per port is a few kilobytes. Due to these tiny buffers, Jumbo Frames do not fit into these buffers.

There exists an additional core which provides a four channel FIFO using the external QDR II SRAM as a buffer (*nf10_sram_fifo*). This core can be attached behind the *nf10_bram_output_queues* core as it is done in the *nic_sram* NetFPGA project. The FIFO core uses all 3 QDR II SRAM memory chips. The core talks to these chips using a Xilinx Memory Interface Generator (MIG) generated interface. The memory chips are asynchronously clocked at a frequency of 210 MHz. With this design it is only possible to buffer the Ethernet data of four ports. As the external memory here also does not have the sufficient bandwidth to store multicast packets under full load, packets get dropped as soon the the FIFOs run full. In this case, not only multicast packets get dropped. Instead, simply the next following packet which does not fit into a FIFO gets lost. This is a situation TCP flow control can not handle well. Single TCP flows might get a throughput break-in some time after multicasts exceed the external memory bandwidth.

4.4.4 Interface Description

Name	Default value	Description
output_ports	7	Number of output ports
multicast_queue_size_256b	2**9	Size of the multicast buffers as multiples of 256 bit
queue_sizes_256b [output_ports]	Memory equally shared	The size of each unicast buffer can be specified as multiples of 256 bit
store_and_forward_ports	7'b0001111	A '1' denotes a port which always needs a continuous data stream at its clock rate (e.g. 10G Interface)

Table 4.4: *Output Queue* core parameters

Name	Direction	Description
clk	in	Clock input
clk2x	in	Clock input. The frequency has to be twice as high as <i>clk</i> .
clk2x90	in	Clock input. 90 degree shifted clock of <i>clk2x</i> .
reset	in	Active high reset
qdr[2]	in/out	SystemVerilog interface to two of the QDR II memory chip
s_axis_*	in/out	AXI4-Stream slave interface (connected to the “ <i>User Custom Logic</i> ”)
m_axis_clk[output_ports]	in	Clock from the port core to clock the asynchronous output FIFO
m_axis_reset[output_ports]	in	Active high synchronous reset (port core clock domain)
m_axis_*[output_ports]	in/out	AXI4-Stream master interface (connected to the port core)

Table 4.5: *Output Queue* core interface

4.4.5 Simulation

An extensive test bench is provided, which verifies the behavior of the *Output Queue* core. Packets with random sizes and random destination ports are generated and the correctness of the packets and the packet order is verified at the outputs. Error messages inform if something goes wrong.

4.4.6 Future Developments

The *Output Queue* core was extensively tested during our work for this project group. Sometimes the complex logic for the recovery of the packet order caused the *Place and Route* FPGA tools to generate a bad placement. We had to add area constraints to limit the freedom of the placer. Floorplanning consumed a lot of time since the *Place and Route* process consumed about an hour on our PCs (OpenFlow switch project). It might be a good idea to temporarily remove the “packet order recovery” functionality of the *Output Queue* core to reduce *Place and Route* problems and maybe also reduce the translation time.

4.5 PCI Express Interface and DMA Engine

The NetFPGA-10G board is built as a PCI Express plug in card. The PCI Express connection supports 8 lanes (x8). The GTX transceivers in the Virtex-5 FPGA could theoretically support the PCI Express 2.0 speed of 5 GT/s. However, the PCI Express hard block on the Virtex-5, which is capable of handling the physical and data link layer, only supports PCI Express 1.0. As this hard block is used, the combined link rate is limited to 20 GT/s.

In our design we use the *dma_v1_00*¹ core from the original NetFPGA project. This core acts as a bridge between the PCI Express bus and the internal AXI4-Lite bus, which is used to access the registers and memory locations of our cores. In addition this core features two DMA channels (send and receive) which are used to transport the Ethernet data between the PC's main memory and the FPGA. Together with the Ethernet packets, the DMA engine can also transport some additional metadata which carries information about the data sink. This is used to forward Ethernet packets to the appropriate network devices on the PC side.

We extended the DMA to fit to our framework and fixed existing bugs.

4.5.1 Extensions to the DMA Core

We applied the same concepts we already used for the other network ports to the DMA core²: 64 bit to 256 bit AXI4-Streaming bus width conversion and the *Packet FIFO* as a store-and-forward buffer in front of the *Input Arbiter*. The new infrastructure provides a flawless connection to our processing pipeline and also supports Jumbo Frames.

The original DMA engine did not support Jumbo Frames. After we got a deeper understanding of the implementation of the DMA engine, we were able to remove the bottlenecks which prevented the use of Jumbo Frames.

4.5.2 Fixing Bugs

After the DMA engine was connected to our infrastructure, we got massive trouble with the stability of the DMA engine. Nearly everything else than sending pings over the network interfaces immediately froze the PC. Sometimes only power cycling the PC could bring it back to life. We started analyzing nearly every line of code of the DMA engine and installed *ChipScope* at countless points to find the bugs. All in all, it took a large amount of time to get the PCIe/DMA system reliably working.

This gives a short overview of the most important bugs we fixed:

- **Active State Power Management (ASPM) of the Virtex-5 hard block is incompatible with some newer mainboards**

The use of ASPM power management features by the operating system caused a complete freeze of the PC. We created a patch for the Xilinx *Core Generator* generated Verilog files that completely disables the advertisement of ASPM capabilities.

¹Source: lib/hw/contrib/pcores/nf10_upb_dma_v1_00_a

²Source: lib/hw/contrib/pcores/nf10_upb_dma_input_v1_00_a

- **Completion buffer overruns**

When sending Ethernet packets out of the PC, the DMA engine has to request the packet data from the PC's main memory. The DMA engine requested this data without caring about the level of the completion buffer. This led to a buffer overrun. We activated the advertisement of completion buffer credits. This solution does not conform to the PCI Express specification. End points have to advertise infinite credits for completions. But as PCIe bridges and switches are allowed to do so, most PC mainboards should support it.

Another solution we tested was to reduce the PCIe link width to x2. This also solved the problem as the DMA engine is always able to process the received data at that rate. Finally, we reduced the link width from x8 to x4. This reduction saves a few resources on the FPGA but has no impact on the throughput as the internal buses of the DMA engine are the bottleneck.

- **Missing constraints for signals crossing clock domains**

We identified some flaws in the implementation of entities that synchronize data between clock domains. We changed the DMA design to work synchronous with the clock from the PCIe hard block. The transition to the main clock domain of the framework is now done via asynchronous FIFOs which are provided by the FPGA. For the AXI4-Lite bus we use the asynchronous capabilities of the Xilinx *AXI Interconnect* core.

- **Invalid Ethernet packets made the DMA unusable**

Whenever the DMA engine received too small Ethernet packets, the DMA descriptor for this packet was not correctly processed. The DMA descriptors got out of sync between the driver on the PC and the DMA engine. The metadata (packet size and the destination port information) did not match to the Ethernet packet buffer anymore. This resulted in the reception of packets with wrong sizes at the wrong Linux network interfaces.

Of course invalid packets should be caught before they are passed to the DMA engine. But for advanced reliability we fixed this bug. Additionally we added a mechanism which detects when DMA descriptors get lost and thus get out of sync. In this case a warning message is written to the Linux kernel log.

There were also some bugs in the driver which caused the Linux kernel to crash on high traffic loads. More details can be found under the software section.

4.6 Layer2-Switch

This core for the UPB NetFPGA-10G framework implements a basic layer-2 switch. It is able to store up to 2^{19} MAC-Address - port relations in the external SRAM of the NetFPGA-10G card. It also invalidates entries five minutes after the last packet from that MAC address has been received from that port.

4.6.1 Design Overview

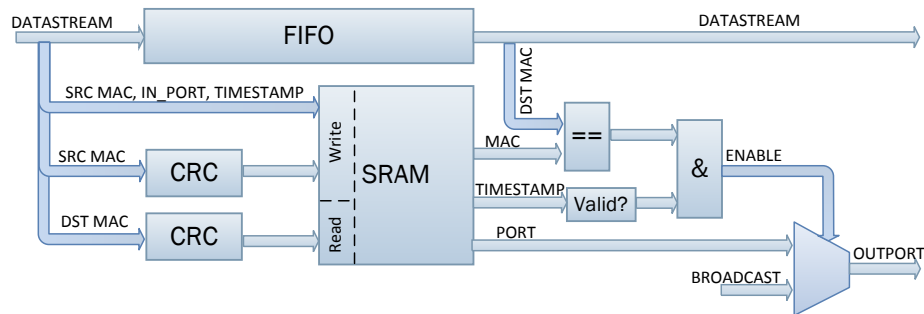


Figure 4.10: Switch Core

The general design of the core is shown in Figure 4.10. The core hashes the source and destination MAC addresses of incoming packets using a CRC algorithm and feeds them to the SRAM. The hashed source MAC address is used as the address to update the entry of this MAC address with the port number of the incoming packet and the current timestamp. The hash of the destination MAC address is used to obtain a possibly existing entry of this MAC address. The data of the entry is then compared to the real data of the packet, as there is a possibility for hash collisions. Also it is checked whether the timestamp is more than five minutes in the past. If this is the case, the packet is handled as if there was no entry at all and broadcasted to all ports except the incoming one. If there is a valid entry the packet is only forwarded to the port noted in the entry.

4.6.2 In-/Outputs and Parameters

Name	Direction	Description
clk	in	Clock signal
clk2x	in	Clock signal for SRAM
clk2x90	in	Clock signal for SRAM

reset	in	Reset signal
s_axis_*	in	AXI4-Stream slave bus (packet data)
m_axis_*	out	AXI4-Stream master bus (packet data)
qdr_c_*	in/out	Signals to QDR2-SRAM

Table 4.6: In- and Outputs of layer-2 switch core

Name	Default Value	Description
dma_port_id	5	Port ID of the DMA input core

Table 4.7: Parameters of layer-2 switch core

Table 4.6 lists the in- and outputs of the core, Table 4.7 does the same for available parameters.

4.6.3 Known Problems and Limitations

The core does not buffer any backpressure but forwards it to its feeding core, risking the overflow of the input queues. In the current framework this does not cause problems as the output queues do never deassert their ready signal.

4.7 OpenFlow Switch

This core for the UPB NetFPGA-10G framework implements the basic operations which are needed in hardware to implement an OpenFlow 1.0 switch.

However, the core does not implement a complete OpenFlow switch in hardware. A host computer which runs software to manage the switch and communicates with the controller is needed.

In the following section the design of the OpenFlow switch and its usage will be described.

4.7.1 Design Overview

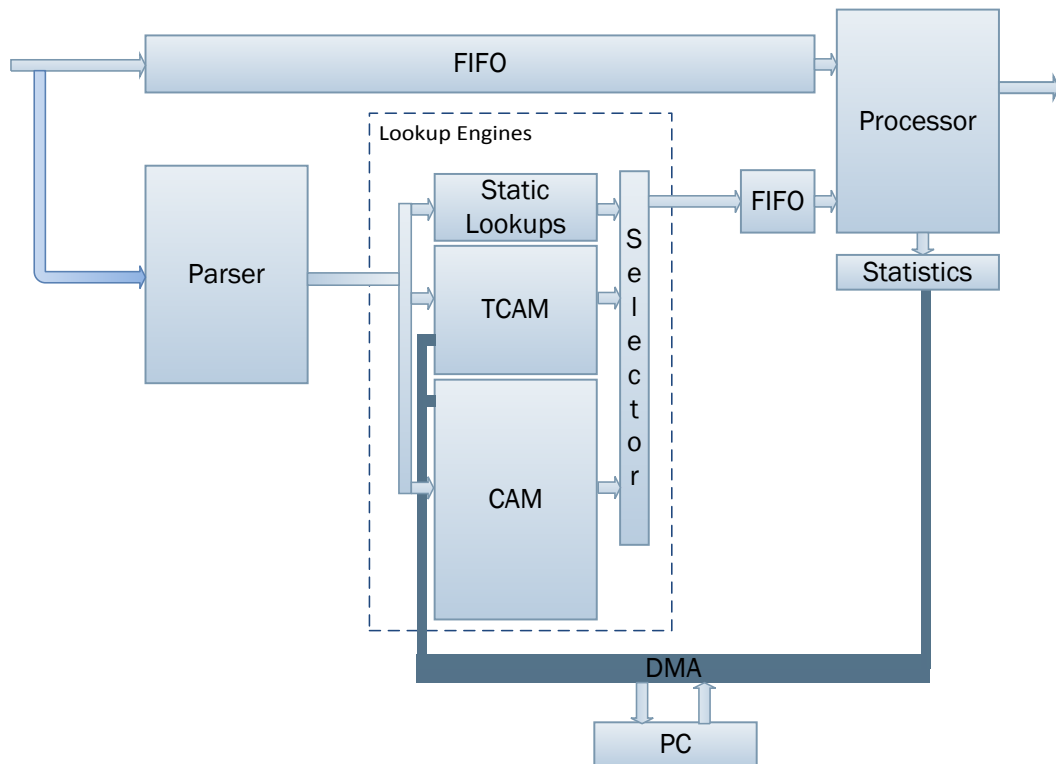


Figure 4.11: OpenFlow Core Framework

The general design of the core is shown in Figure 4.11 and consists of several submodules:

- FIFO
- Parser
- Lookup Engines

- Selector
- Action Processor

FIFO

The FIFO is used to delay the AXI4-Stream. It is read by the processor when an action for the packet has been computed.

Parser

Name	Width	Bit positions	Description
valid	1	243	Signals that tuple values are valid. Has to be asserted by the processor for exactly one clock cycle per packet.
port	3	242:240	Incoming port
vport	3	239:237	Incoming vport
dmac	48	236:189	Destination MAC Address
smac	48	188:141	Source MAC Address
typ	16	140:125	Ethernet type
vid	12	124:113	VLAN Id
pcp	3	112:110	VLAN PCP Field
sip	32	109:78	Source IP Address
dip	32	77:46	Destination IP Address
prot	8	45:38	IP Protocol
tos	6	37:32	IP ToS bits
tsp	16	31:16	Transport source port / ICMP Type
tdp	16	15:0	Transport destination port / ICMP Code

Table 4.8: OpenFlow Tuple (Parser output)

The parser extracts the header fields which are specified in the OpenFlow 1.0 standard [7]. It understands 802.3 Ethernet, 802.1q VLAN, ARP, IPv4 (including QoS according to RFC 2474 [8]), TCP, UDP and ICMP packets. As soon as an Ethernet packet is completely parsed, the *valid* signal is asserted for one clock cycle. In case of an parsing error, the *error* signal is additionally asserted. During clock cycles with a *valid* parsing result, all applicable fields are filled with the appropriate header values (see Table 4.8 for the extracted fields). Fields not applicable to the detected protocols are filled with zeros. This eliminates the need of unnecessary wildcards inside flow table entries.

Internally the parser is deeply pipelined. Before parsing the actual Ethernet data, the parser copies all potentially relevant bits into a register. These bits are called the *Bits of Interest*. This register has a width of 768 bit for OpenFlow 1.0. During the synthesis process, many unused bits get automatically removed by the tools. After these *Bits of Interest* are collected, the parser feeds the data into its parsing pipeline. During each pipeline step, the processed bits are removed before the remaining data is fed into the next pipeline stage. Additional signals carry information for the next parsing stages. This pipeline is optimized to reduce the *logic levels* for a 6-input LUT FPGA architecture. A very careful design is important here because the data buses are quite wide. Using too many *logic levels* here can easily cause timing problems, as the look-up tables are spread over a large area which significantly increases the delay on the routed signals.

The parsing pipeline has 6 stages. This means that the delay is always 6 clock cycles (after the *Bits of Interest* are ready). If some pipeline stage is not applicable to the current Ethernet packet, the data is simply copied to the next stage. This gives an overview of what is done in each stage:

1. Parse the Ethernet MAC headers
2. Parse the optional VLAN headers
3. Parse ARP or IP headers
4. Remove 0, 4, 8 or 12 IP option fields
5. Remove 0, 1, 2 or 3 IP option fields
6. Parse ICMP, TCP or UDP

The parser detects whenever a packet is too short for a detected protocol and asserts the *error* flag. There are also some plausibility checks for IP headers (IP version field and IP packet length). However, the parser does not verify any checksums. It is assumed that these are correct as the whole Ethernet packet already passed the *Frame Check Sequence* test before. If there is an implementation fault somewhere in the network and invalid TCP or UDP packets are forwarded, the receptor has to detect this and indicate some warning. The only drawback is that these wrong packets are counted as processed packets in the flow table statistics.

An extensive test bench is provided with the parser. We captured several Ethernet packets which include all relevant protocols. The test bench delivers these packets to the parser and checks if the result is as expected. For every protocol additional invalid packets are fed into the parser to validate the correct behavior.

Lookup Engines

Name	Width	Description
valid	1	Action signals are valid. Must be asserted for one clock cycle per packet.
match	1	Packet tuple is included in set of packets for which the action was defined.
type	2	Type of action. <i>00</i> describes FORWARD or DROP action, <i>01</i> IN_PORT action, <i>10</i> ALL action, <i>11</i> SEND_TO_CONTROLLER action.
port	C_OUT_PORT_WIDTH	Unary encoding of physical ports to which packet shall be forwarded. Only used for type == 00.
vport	C_OUT_PORT_WIDTH	Unary encoding of virtual ports to which packet shall be forwarded. Only used for type == 00.
match_addr	C_MATCH_ADDR_WIDTH	Address of match line in lookup engine. Will be forwarded to processor for statistics keeping.

Table 4.9: Action signals

The lookup engines provide the packet matching functionality of the core. Their basic functionality is to deliver OpenFlow actions for a defined set of packets, which is called a 'Flow'. The Flow is defined by an header field which is either static or configured at runtime. A lookup engine has to deliver a match or no-match signal within a fixed number of cycles. It has to be able to receive a lookup every 2 clock cycles as this is the minimum number of cycles needed to transfer an Ethernet packet at 256 bits per cycle. The match event is signalled by the assertion of both the *match* and the *valid* signals for one clock cycle, whereas the no-match event is indicated by the assertion of only the *valid* signal. The format of the OpenFlow action is shown in Table 4.9.

Currently implemented lookup engines in ascending priority are:

- Static Lookup
- TCAM Lookup
- CAM Lookup
- VPort Lookup

In general the Static Lookup and VPort Lookup engines implement the default behaviour of the OpenFlow core. Their task is to forward packets without any matches in CAM and TCAM

to the software part of the implementation and to allow the software part to send packets out of physical ports with highest priority.

As each of the lookup engines have their own address space, the match address values defined in Table 4.9 are not unique. To make them unique each lookup engine's match address is added with an offset which is computed by the sum of the possible match addresses of the lookup engines with lower priority.



Match address width

The sum of match address and offset might be wider than the original match address. Make sure to set the parameter `C_MATCH_ADDR_WIDTH` in file *parameters.v* accordingly.

Selector

The selector submodule selects one of the actions which were outputted by the lookup engines. The lookup engines are connected to the selector in order of their priority. The selected action is always the action of the lookup engine with the highest priority. All lookup engines have to provide the match or no-match signal at exactly the same time to the selector. To synchronize these signals a delay module is provided which can be used to delay the signals of a specific lookup engine for a fixed number of clock cycles.

Action Processor

The action processor matches actions to packets. For every action which is forwarded to the processor a packet is taken from the FIFO and handled accordingly to the action. Actions are performed on packets by setting the *outport* and *outvport* signals. The processor stores actions in a FIFO as the parser and lookup engines might be pipelined and might deliver new actions before the processor has finished handling a packet. The processor also does basic statistics gathering. It maintains 32 bit counters for every possible match address to count the number of packets, the number of bytes and the timestamp of the last time the action was applied. The system time of the processor is stored in a 32 bit vector and can be read using a dedicated address. The processor does not do any overflow handling of counters. Each possible match address is associated with three AXI addresses which can be used to poll statistics. The addresses are calculated as follows: The AXI base address of the processor is ored with the match address which is shifted by four bits. The last two bits are always zero to allow byte addressing. The third and fourth bit are used to index the different statistics. *00* is used to access the packet counter, *01* for the byte counter and *10* for the timestamp.



block RAM allocation for statistics

The processor allocates enough block RAM to store statistics for all possible $2^{C_MATCH_ADDR_WIDTH}$ match addresses. Because of this, the number of allocated block RAM primitives increases exponential to the value of the C_MATCH_ADDR_WIDTH parameter in file *parameters.v*. Make sure to set the parameter to the smallest value big enough to hold all match addresses which are used by the lookup engines plus their respective offsets.

4.7.2 In-/Outputs and Parameters

Name	Direction	Description
clk	in	Clock signal
reset	in	Reset signal
s_axis_*	in	AXI4-Stream slave bus (packet data)
m_axis_*	out	AXI4-Stream master bus (packet data)
s_axi_stats_*	in/out	AXI4-Lite slave bus for statistics (read-only)
s_axi_tcam_*	in/out	AXI4-Lite slave bus for TCAM lookup engine
s_axi_cam_*	in/out	AXI4-Lite slave bus for CAM lookup engine


Table 4.10: In- and Outputs of OpenFlow Switch core

Name	Default Value	Description
C_AXI_BASE_ADDR_TCAM	0xA0000000	AXI Base Address for TCAM lookup engine
C_AXI_HIGH_ADDR_TCAM	0xAFFFFFFF	AXI High Address for TCAM lookup engine
C_AXI_BASE_ADDR_CAM	0xB0000000	AXI Base Address for CAM lookup engine
C_AXI_HIGH_ADDR_CAM	0xBFFFFFFF	AXI High Address for CAM lookup engine
C_AXI_BASE_ADDR_STATS	0xC0000000	AXI Base Address for statistics
C_AXI_HIGH_ADDR_STATS	0xCFFFFFFF	AXI High Address for statistics
C_DMA_PORT	5	Port number of DMA input core

C_DMA_FIRST_EXTERNAL_PORT	0	Port number of first external input core
C_DMA_LAST_EXTERNAL_PORT	4	Port number of last external input core
TCAM_DEPTH	64	Number of match lines in TCAM lookup engine
CAM_DEPTH	2048	Number of match lines in CAM lookup engine

Table 4.11: Parameters of OpenFlow Switch core

Table 4.10 lists the in- and outputs of the core, Table 4.11 does the same for available parameters. Additional parameters which mostly define widths of vectors are defined in the file *parameters.v*.



Setting parameters

The parameters listed in Table 4.11 are defined in *nf10_upb_ofswitch.v*. However they are preset with obviously invalid values. The parameters need to be overwritten in the Synplify project file *nf10_upb_ofswitch.prj* before synthesis. Consider this when changing them. Also be sure to overwrite them when creating testbenches.

This does not hold for the parameters defined in file *parameters.v* which are set to their final value there.

4.7.3 Known Problems and Limitations

The core does not cope well with backpressure. If the ready signal of the slave core connected to the outgoing AXI4-Stream master bus is not set for some clock cycles the datastream delay FIFO and the action FIFO in the processor might overflow. In this case the core might loose packets or actions which results in actions being applied to the wrong packets. There is currently no way to recover from this state without resetting the core.

In the current framework this limitation does not cause problems as the output queues do never deassert their ready signal.

Our proposed solution to this problem is to check for the 'almost full' signal of both FIFOs in the *nf10_upb_ofswitch.v* file and deassert the ready signal of the AXI4-Stream slave bus if one of them is asserted.

The testbench of the core is not elaborate enough to detect errors in the execution of actions on packets. As of now it only detects missing or malformed packets. Errors in the handling of actions or statistics have to be detected by hand. The development of an elaborate testbench is proposed.

4.8 TCAM

In our design of an OpenFlow switch a TCAM is used to provide a lookup engine for wildcard flow entries. It needs to store header bitmaps of flows, where each bit can be either 0, 1 or X (don't care), and the corresponding action that is assigned to this flow. An interface has to be provided to add and remove flow entries from the TCAM during runtime. The TCAM also has to provide an interface for action lookups, where a given header bitmap is matched against the stored entries and the action stored for the first matching entry is returned.

There are several ready to use TCAM IP cores available. These IP cores have additional functionality like allowing to pass don't-care bits on lookup and provide convenient interfaces for filling the TCAM. The core we developed deliberately lacks these features and only provides the necessary functionality. The goal of this decision is to minimize hardware requirements and therefore allow a higher number of entries and overall better timing. Passing don't care bits on lookup is not needed for our task and the lack of a convenient interface can be compensated in software.

4.8.1 Design Overview

Figure 4.12 shows a block diagram of the `upb_tcam_lookup` core. When instantiating the `upb_tcam_lookup` core several attributes can be set. Table 4.12 shows the available attributes and briefly describes their meaning.

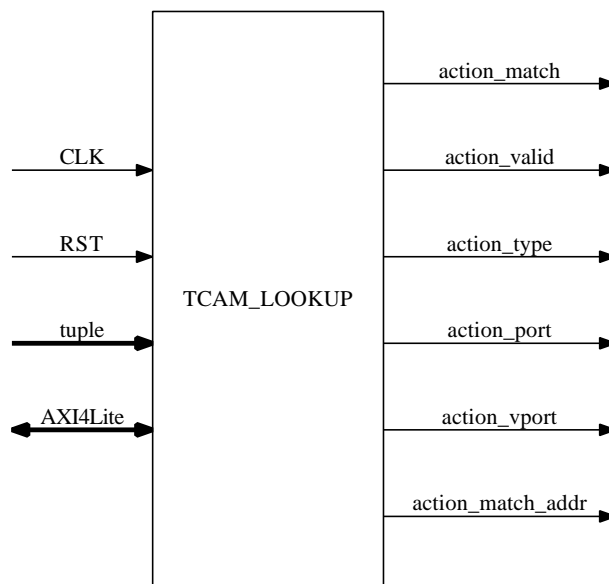


Figure 4.12: `upb_tcam_lookup` Block

Attribute Name	Type	Default	Notes
DELAY_MATCH	Integer	3	Delay generation of match signals by X clock cycles. This can be used to improve timing. Minimum: 1
DELAY_LOOKUP	Integer	3	Delay lookup of corresponding action by X clock cycles if a match was found. This can be used to improve timing. Minimum: 1
DELAY_WRITE	Integer	3	Delay writing changes by X clock cycles. This can be used to improve timing. Minimum: 1
FORCE_MUXCY	0/1	0	Instead of letting the tools implement the match logic, force the usage of the carry chain on current Xilinx devices. This can improve timing but impact depends on actual placement.
TCAM_DEPTH	Integer	32	Number of TCAM entries. Maximum: 1024
C_AXI_BASE_ADDR	Integer	0x00000000	Base address of this core for AXI4-Lite interface.
C_AXI_HIGH_ADDR	Integer	0xFFFFFFFF	High address of this core for AXI4-Lite interface.

Table 4.12: upb_tcam_lookup Attributes

Table 4.13 gives an overview about all available ports of the upb_tcam_lookup core.

Port Name	Direction	Signal Description
CLK	Input	Clock signal
RST	Input	Reset signal (active high)
tuple	Input	Flow tuple for which a lookup should be performed
action_match	Output	High if an action was found for the given tuple
action_valid	Output	High if output signals are valid
action_type	Output	Type of action (if found)
action_port	Output	Port that the packet should be forwarded to
action_vport	Output	Vport that the packet should be forwarded to

action_match_addr	Output	Address of the matching entry (if match was found)
s_axi_*	In-/Outputs	Standard AXI4-Lite interface (see chapter 4.8.3)

Table 4.13: upb_tcam_lookup Ports

4.8.2 Internal Design

Each TCAM entry is realized using 5-input LUTs. They are instantiated using 32bit shift registers (SRLC32E). To minimize the hardware utilization sequentially feeding data into these shift registers is not done in hardware but has to be done in software. To utilize the full 32bit width of the AXI4-Lite interface, data is fed into 32 registers in parallel.

The input tuple is split into parts of 5bit and each part is connected to the input of one LUT in each tcam entry. If all LUTs of an entry generate a match signal, a match was found. Therefore an AND between the outputs of all LUTs which belong to one entry is built. This can lead to timing issues, therefore another technique of generating the match signal can be chosen, by forcing the usage of the MUXCY multiplexers in the carry chain of current Xilinx devices. In practice this technique is flawed by the *Place and Route* tools that do not choose one contiguous carry chain but spread the parts over the FPGA.

If a match was found the corresponding action has to be looked up in a block RAM. Because there can be more than one match, the address for this lookup is determined by searching the first matching entry. This leads to the fact that the priority of the entries is encoded by their position in the TCAM.

4.8.3 AXI4-Lite Interface

The TCAM lookup core provides an AXI4-Lite interface that allows adding and removing entries from the TCAM. Also some information about the module can be read via this interface. All addresses in this section are relative to AXI_BASE_ADDR.

Table 4.14 shows the available addresses for read operations and the returned data.

Address	Description
0x00000000	Name of the lookup core ("TCAM")
0x00000004	Version of the core
0x00000008	Number of available entries in the TCAM

Table 4.14: AXI4-Lite readable addresses

The interface for adding entries to the TCAM, activating and deactivating them is realized via a direct interface to the basic elements inside the TCAM, namely SRL32 shift registers. The corresponding addresses can be seen in table 4.15.

Starting with the MSB these bits are fed into the SRLs sequentially. 32 SRLs are writable in parallel (1st SRL at LSB, 32nd SRL at MSB) using the AXI4-Lite interface (see table 4.15). This results in the following 32 write operations to the address 0x00004000:

00000000000000000000000000000000	0x0	(4.1)
00000000000000000000000000000000	0x0	(4.2)
00000000000000000000000000000000	0x0	(4.3)
00000000000000000000000000000000	0x0	(4.4)
00000000000000000000000000000000	0x0	(4.5)
00000000000000000000000000000000	0x0	(4.6)
00000000000000000000000000000000	0x0	(4.7)
00000000000000000000000000000000	0x0	(4.8)
00000000000000000000000000000000	0x0	(4.9)
00000000000000000000000000000000	0x0	(4.10)
00000000000000000000000000000000	0x0	(4.11)
00000000000000000000000000000000	0x0	(4.12)
00000000000000000000000000000000	0x0	(4.13)
00000000000000000000000000000000	0x0	(4.14)
00000000000000000000000000000000	0x0	(4.15)
00000000000000000000000000000000	0x0	(4.16)
00000000000000000000000000000000	0x0	(4.17)
00000000000000000000000000000000	0x0	(4.18)
00000000000000000000000000000000	0x0	(4.19)
00000000000000000000000000000000	0x0	(4.20)
00000000000000000000000000000001	0x2	(4.21)
00000000000000000000000000000001	0x2	(4.22)
00000000000000000000000000000000	0x0	(4.23)
00000000000000000000000000000000	0x0	(4.24)
00000000000000000000000000000000	0x0	(4.25)
00000000000000000000000000000000	0x0	(4.26)
00000000000000000000000000000000	0x0	(4.27)
00000000000000000000000000000000	0x0	(4.28)
00000000000000000000000000000001	0x1	(4.29)
00000000000000000000000000000000	0x0	(4.30)
00000000000000000000000000000001	0x1	(4.31)
00000000000000000000000000000000	0x0	(4.32)

Now the action has to be written and the entry has to be activated. Therefore 0xabc is written to 0x00008000 and 0x1 is written to 0x0000c000. Note that the last operation would disable entries 1–31. Therefore a bitstring representing all active entries in this range has to be written when en- or disabling an entry.

4.9 CAM

The CAM lookup engine is designed to fit into the OpenFlow core.

It provides the core with the ability to do action lookups for packets which are specified by their OpenFlow header field. The lookup engine is able to do one lookup every two clock cycles and provides a match or no-match signal in exactly one clock cycle. The general design of the lookup engine is shown in Figure 4.13

4.9.1 Design Overview

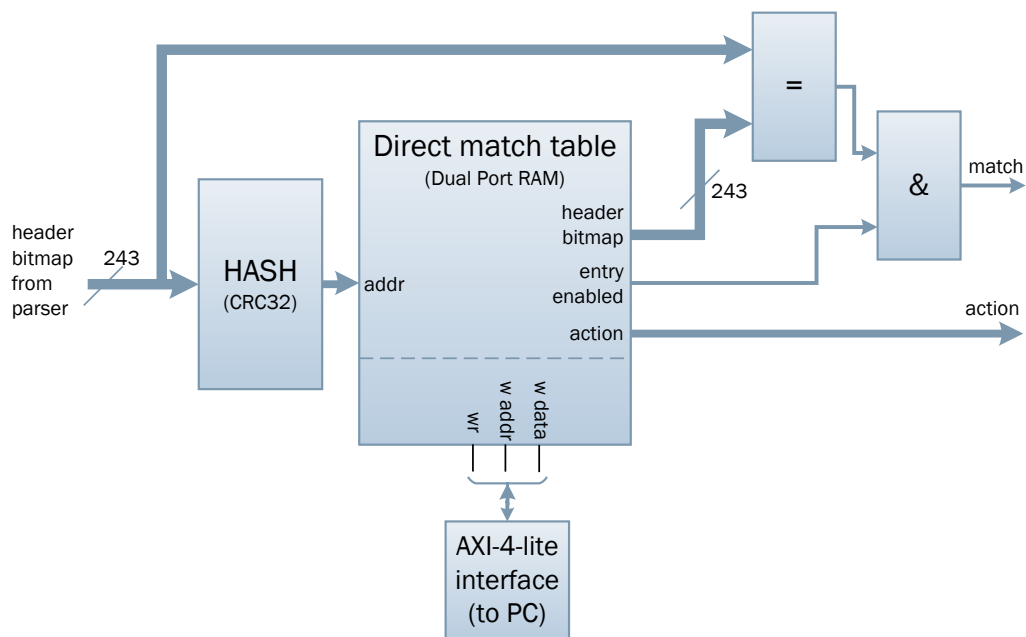


Figure 4.13: CAM lookup engine design

The design of the lookup engine is straight-forward and uses block RAM primitives to store data.

4.9.2 In-/Outputs and Parameters

Name	Direction	Description
CLK	in	Clock signal
RST	in	Reset signal

tuple	in	OpenFlow header field as described in Table 4.8.
action_*	out	Action as described in Table 4.9.
s_axi_*	in/out	AXI4-Lite slave bus to fill CAM and read properties.

Table 4.17: In- and Outputs of the CAM lookup engine

Name	Default Value	Description
CAM_DEPTH	2048	Number of match lines. Block RAM will be sized accordingly.
CAM_VERSION	1	Version number which can be requested by the host computer.
C_AXI_BASE_ADDR	0xB0000000	AXI4-Lite base address to fill CAM and read properties.
C_AXI_HIGH_ADDR	0xBFFFFFFF	AXI4-Lite high address to fill CAM and read properties.

Table 4.18: Parameters of CAM lookup engine

Table 4.17 lists the in- and outputs of the module, Table 4.18 does the same for available parameters.

4.9.3 Interfacing with the CAM

The CAM can be filled via the AXI4-Lite bus. It is important to know that the entries of the CAM are write-only. The only addresses which can be read are the ones described in Table 4.19.

The procedure to write cam entries is as follows:

1. Calculate the CRC of the tuple which is to be written into the cam. The polynomial to use is $1 + x^1 + x^2 + x^4 + x^5 + x^7 + x^8 + x^{10} + x^{11} + x^{12} + x^{16} + x^{22} + x^{23} + x^{26} + x^{32}$.
2. Take the $\log_2(CAM_DEPTH)$ least significant bits of the CRC and shift them five bits to the left (two bits to allow byte addressing, three bits to index eight different bytes).
3. OR the resulting value with the AXI base address to get the first address of the CAM entry.
4. Write the action as a 18 bit tuple (2 bit type, 8 bit vport, 8 bit port) to the address.
5. Write the 32 least significant bits of the tuple to the first address.

6. Write the following bits of the tuple in 32 bit chunks to the next seven byte addresses.

The first address is used twice in this procedure as the parity bits of the block RAM primitives can be used to store 36 bit words. As AXI only allows to write 32 bit words the first 32 bits of the first write is cached and written 4 bit at a time to the block RAM in the following write operations. This is done transparent to the AXI interface.

The general structure of the write addresses can be seen in Table 4.20.

In case of errors during a write operation the internal state machine of the write interface can either be reset by asserting the reset signal of the core or by writing zeros to an address ending in (bitwise) 00000 and 11100. Note that this will invalidate the action which is stored in that address.

Address	Description
0x00000000	Returns the string "CAM" encoded in ASCII
0x00000004	Version number
0x00000008	Returns CAM_DEPTH parameter

Table 4.19: Readable addresses of CAM lookup engine

Address	Description
0x00000000	Action field on first write access, first chunk of OpenFlow header field in second
0x00000004	Second chunk of OpenFlow header field
...	...
0x0000001c	Eighth chunk of OpenFlow header field
0x00000020	Action field on first write access, first chunk of OpenFlow header field in second
0x00000024	Second chunk of OpenFlow header field
...	...
0x0000003c	Eighth chunk of OpenFlow header field
...	...

Table 4.20: Structure of CAM write entries

4.9.4 Using the Userspace Tools

Together with the lookup engine there are three userspace tools provided to fill and test the engine. All of these are considered experimental and should be read carefully before being

used.

The *clear_cam.py* script takes the device node of a NetFPGA card as parameter and deletes all contents from the CAM.

The *tuple_calculator.py* script opens an interactive shell which provides functions to create tuple bitstrings and hash these according to the same polynomial as the lookup engine. It also is able to generate shell commands to write tuples to the CAM of a connected NetFPGA card.

The *cam_switch.py* script takes as arguments a list of the NetFPGA's virtual interfaces. It generates CAM entries for packets it receives which are broadcasted if it does not know the destination of a packet. The MAC addresses which are used to forward packets to the correct port are hardcoded in the script.

4.9.5 Known Problems and Limitations

The CAM currently uses block RAM primitives only. An engine which uses external SRAM would be able to store much more entries than the block RAM implementation. However this is no trivial task as the statistics for these entries would need to be stored in SRAM too.

The current implementation of the CAM uses the lowest 11 bits of a CRC32 hash of the OpenFlow tuple as address of an entry. Practical tests have shown that this method creates hash collisions rather fast. It is unknown whether this is to be expected due to the birthday problem [9] or might be optimizable.

4.10 Further Cores

There are some additional helper cores which are needed to operate the other cores and which can assist in debugging. We mainly want to name them here together with a very brief description.

4.10.1 Clock Generator

The *Clock Generator*¹ core provides the clock and reset signals for all other cores. The first plan was to operate the whole framework at a main clock frequency of 150 MHz (38.4 Gb internal throughput). With the growing complexity of our project it turned out that the effort, we had to spend into floorplanning, consumed too much time. So we decided to reduce the clock frequency to 120 MHz (30.72 Gb internal throughput). This can be done by setting the parameter *reduce_clk_to_120mhz* in the *Clock Generator* core.

Beside the main clock, the core also delivers the doubled clock and the 90 degree phase shifted doubled clock to the QDR II SRAM core. The core also manages the reset and startup procedure. The *reset* output signal is held active until the PLLs have locked, the DCI (*Device Controlled Impedance*) controller works stable and the IO delay controller (*IODELAYCTRL*) also works stable. After all these constraints are met, the reset signal is held active for further 20 μ s.

4.10.2 AXI4-Stream Bus Conformance Test

This core² is a helper core which checks the bus signals for conformance. Various error bits, which can be used to trigger *ChipScope*, inform about different problems.

4.10.3 ChipScope Debugging Cores

We have written cores^{3 4 5} that can be used to easily monitor AXI4-Lite and AXI4-Stream buses with ChipScope. The AXI4-Stream bus *ChipScope* monitor core also includes the *AXI4-Stream Bus Conformance Test* core.

These debugging cores can easily be added to the design using the *Xilinx Platform Studio* (XPS) and thus provide a convenient way to visualize the bus traffic at different point in the processing pipeline at the same time.

¹Source: lib/hw/contrib/pcores/nf10_upb_clock_generator_v1_00_a

²Source: lib/hw/contrib/pcores/nf10_upb_lib/hdl/verilog/axis_conform_check.v

³Source: lib/hw/contrib/pcores/nf10_upb_chipscope_icon_v1_00_a

⁴Source: lib/hw/contrib/pcores/nf10_upb_axi_lite_chipscope_v1_00_a

⁵Source: lib/hw/contrib/pcores/nf10_upb_axi_stream_chipscope_v1_00_a

5 Software

In the previous chapters we have shown how to build a data plane of a typical SDN network switch. The parser capabilities and the priorities of the CAM and TCAM lookup modules were designed to support OpenFlow 1.0. To build a complete OpenFlow switch, a large part of software is still missing.

In the following sections we will show how this switch is completed. A Linux driver and a library is used to make the connection to the data plane. Additionally we extended an already existing software OpenFlow switch to support our data plane: Open vSwitch.

5.1 The Linux Network Interface Card Driver

The driver¹ is based on the *reference_nic* driver from the original NetFPGA project.

5.1.1 Improvements and Bug Fixes

We extended the driver to support multiple cards in one PC. The naming scheme of the card had to be changed to accomplish this (*/dev/nf10* changed to */dev/nf10a*, *nf10b*, ...). For our concept we needed more than just 4 network interfaces. These interfaces are now called: *nf10a*, *nf10av0*, *nf10av1*, ..., *nf10av5* (for card *a*). *nf10a* is what we call a *general purpose* network interface and the *nf10av...* are so-called virtual ports. These are later used to forward packets without a flow table entry in the hardware to process them in software.

Additionally we fixed some bugs in the driver. At some points there was an incorrect usage of locks and memory barriers. Then we updated the driver design to the latest Linux kernel techniques. The *New API (NAPI)* is now used as an interrupt polling mechanism in case of frequent interrupts. Some deprecated function calls are now replaced by the current ones. The first tests showed that we nearly doubled the throughput of the DMA engine. This is caused by several improvements: Jumbo Frame support (MTU of 9000 bytes), improvements of the FPGA framework, driver improvements (no unnecessary locks, NAPI) and also the support of later Linux kernels. Our first throughput tests with the original *reference_nic* project showed a data rate of about 2.5 Gb. This was tested with a 3.8 Linux kernel as we could not get later kernels to work stable with this project. For our project we used Linux kernel 3.14.15 and reached about 5 Gb. These values were determined using an unidirectional *iperf* TCP test. If we do bidirectional TCP transfer tests, one direction only reached a few hundreds of megabits per second. This is due to the fact that the arbitration in the DMA engine does not equally assign its resources. It is obvious that all these values here are far away from what we expect from a 20 GT/s PCIe connection. We tried different approaches to further increase the

¹Source: `lib/sw/contrib/drivers/nf10_upb_dma_v1_00_a/src/nic_driver/`

throughput. Increasing the clock frequency of the DMA engine from 125 MHz to 250 MHz seems to be impossible on a Virtex-5. The delays of many paths in the design are far away from what is feasible for a frequency of 250 MHz. Increasing the buffers inside the DMA engine also did not help much.

The DMA engine together with the NIC driver are feasible to transport Ethernet packets to the software whenever flow entries are not installed in the data plane. However, using the NetFPGA as a general purpose 4-port 10G Ethernet NIC is not a good idea. For all our throughput tests we used separate 10G Ethernet cards (Emulex and Myricom).

During the last days of our project group we got the message that there is a new DMA engine and driver developed at the Cambridge University. It is used within their Open Source Network Tester (OSNT) project which also runs on the NetFPGA-10G. Maybe it's possible to port this driver to our framework. Unfortunately there was no time left to have a deeper look into this project.

5.1.2 Talking to the AXI4-Lite Bus

The cores that need to be controlled by the software are all connected to the *AXI4-Lite* bus on the FPGA. Inside the DMA engine there are some memory mapped registers which provide access to the FIFOs that are connected to this bus. As in the original NetFPGA project, this bus can be accessed via *IOCTL* calls to the driver. This means that every access from user space to the AXI bus causes a switch into kernel mode. We improved this by providing the functionality to map these AXI bus access registers directly into user space. The user space program can make a call to *mmap* to map one page with DMA registers to the user space. To make this secure, we changed the address space inside the DMA engine that only the mandatory registers are located on this page. The class *axi_bus_connector*¹ from the library *sdn_dataplane* uses the *mmap* feature of the driver to provide a convenient way to access the cores on the FPGA.

5.2 The Big Picture: How to Build an OpenFlow Switch

Figure 5.1 shows an overview of the complete OpenFlow switch. The two NetFPGA-10G cards are plugged into one PC. Both cards feature four 10G ports and are interconnected via the high speed Samtec connector. On the PC side, a Linux operating system, which has the NetFPGA NIC driver installed, is running. Each of the NetFPGA cards has 6 so-called virtual network interfaces: *nf10xv0*, *nf10xv1*, ..., *nf10xv5*. Inside the NetFPGA cards we see white boxes with dashed lines. These are the default forwarding rules of the OpenFlow core. Each of the virtual network interfaces (*nf10xv...*) is associated with one physical port, except *nf10xv5*. This interface is associated with the *general purpose* network interface *nf10x0*. This interface can be freely used under Linux, e.g. a IP address can be assigned.

Using the virtual network interfaces *nf10xv...* and a software OpenFlow switch attached to these interfaces, we already have a fully functional OpenFlow switch. For the software switch we use Open vSwitch [4]. This is a well established product that is widely used in the industry,

¹Source: lib/sw/contrib/drivers/nf10_upb_sdn_dataplane_v1_00_a/src/libsdn_dataplane/axi_bus_connector.hpp

for example in virtualization solutions. Inside Open vSwitch we have two so-called bridges. Each of these form an OpenFlow switch which then connects to a common OpenFlow controller. For our tests we used the well known *POX* [6] switch.

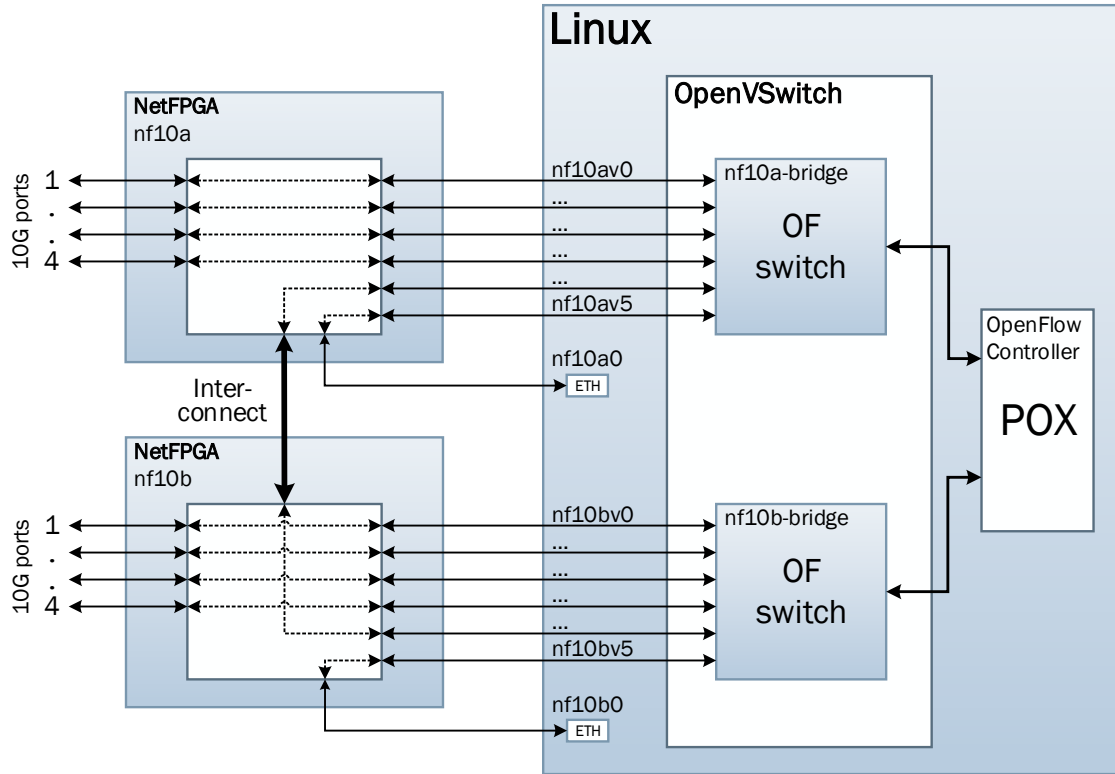


Figure 5.1: The big picture of an OpenFlow switch

Of course, we do not only want to provide a software switch solution. The next step is now to add an interface from *Open vSwitch* to the NetFPGA card that installs flow entries in the hardware tables. By copying flow table entries to the hardware, the default forwarding is overridden and the flow's packets do not arrive at the software switch anymore. They are simply hidden. This also means that the software switch additionally needs some information about what is going on in hardware. Otherwise *OpenFlow* idle timeouts would not work anymore.

5.3 The Library *sdn_dataplane*

The library *sdn_dataplane*¹ provides a high level access to the flow tables and the statistic counters on the FPGA. This library is written in C++11 and uses the Boost library [10]. An additional C wrapper provides an interface to other projects which are written in plain C. The

¹Source: lib/sw/contrib/drivers/nf10_upb_sdn_dataplane_v1_00_a/src/libsdn_dataplane/

Open vSwitch integration uses this wrapper. All objects in the library use the concept of C++ shared pointers (the Boost version of *shared_ptr*<> is used [11]). These are smart pointers which do reference counting. Whenever the reference count goes to zero, the object is destroyed. This concept is used instead of a garbage collector.

This documentation describes the concepts behind the *sdn_dataplane* library and gives an overview about the usage. There exists a separate reference manual where all classes, members and other functions are documented. The documentation is done using Doxygen. It is advised to use the HTML version. This is built with every *make* command that builds the cores and the software as long as doxygen and graphviz are available on the system. Additionally the source code has many comments which explain a lot of the functionality.

5.3.1 Automatic Flow Entry Management: The Class *sdn_dataplane*

To get a feeling of what the library¹ does, we start with a code example here. The Listing 5.1 shows how to get a connection to the data plane on the FPGA, install a flow entry with a *drop* action, modify that action, request the statistics associated with the flow entry and finally delete the flow entry.

```
1  try {
2      using namespace boost;
3
4      shared_ptr<sdn_dataplane> dp = make_shared<sdn_dataplane>("/dev/nf10a");
5
6      // create a new flow entry
7      shared_ptr<flow> f = make_shared<flow>();
8      flow::mac_t key = {{0x00, 0x60, 0xdd, 0x45, 0x42, 0xb9}};
9      flow::mac_t mask = {{0xff, 0xff, 0xff, 0xff, 0xff, 0xff}};
10     f->set_dst_mac(key, mask); // match the source MAC address
11
12     // add to flow table: Packets with given MAC will be DROPPED (default)
13     dp->add_flow(f);
14
15     // change action later: FORWARD to port 0 and 1
16     f->set_action(
17         flow::action_types::FORWARD,
18         port::egress_port(0)
19         | port::egress_port(1)
20     );
21
22     // request statistics
23     uint64_t packets, bytes;
24     posix_time::ptime utc_last_packet;
25     f->get_statistics(packets, bytes, utc_last_packet);
```

¹Source: lib/sw/contrib/drivers/nf10_upb_sdn_dataplane_v1_00_a/src/libsdn_dataplane/sdn_dataplane.hpp

```

26
27     f.reset(); // delete entry
28
29 }
30 catch (const errors &e) { /* process errors here */ }

```

Listing 5.1: Example using the library `sdn_dataplane`

The library has a built-in error system. Exceptions of the enumeration class *errors* carry the error code which can be translated into a text message. Inside the try...catch block an object of the class *sdn_dataplane* is created (Line 4). Internally, the class *sdn_dataplane* creates an instance of the class *axi_bus_connector* and stores a shared pointer. The connection is established to NetFPGA *nf10a* as denoted in the constructor call.

A new flow entry is created in line 7. The default action of this flow entry is to drop the packet. The flow entry object is modified to match only if the destination MAC address equals to 00:60:dd:45:42:b9 (Line 8-10). For every field in the flow entry, a mask can be specified. Every '1' bit in the mask means "this bit has to match". A '0' denotes a wildcard on that bit.

In line 13 the flow is added to the class *sdn_dataplane*. But this does not necessarily mean, that the flow really was installed in hardware. Internally the class *sdn_dataplane* holds objects of the classes *cam* and *tcam*. These classes provide the direct access to the hardware tables. During the instantiation, *cam* and *tcam* read out the table sizes of the tables on the FPGA and cleared the contents. When calling *add_flow(...)* of the class *sdn_dataplane*, it tries to install the flow into the "cheapest" hardware table which supports the flow entry. The "cheapest" table is always the largest table, in our case the CAM. As the example flow entry here has wildcards at most fields, it cannot be added to the CAM, instead *add_flow(...)* from the *tcam* class is called to put the flow entry there. If there is a free entry in the TCAM and there is no priority conflict, the flow entry will be installed in hardware. There are quite some reasons which can make it impossible to push a flow entry to the FPGA. These will be discussed in the next section.

The class *sdn_dataplane* does its best to always install flow entries into the largest ("cheapest") table, which can support this flow. But *sdn_dataplane* also automatically uninstalls flow entries from hardware when there are conflicts. The *flow* object can be queried if it is bound (installed) to a hardware table by calling *is_bound()*.

After the flow entry is added (line 13), the associated action can later be changed. This is done in line 16 where the action is changed to forward the packets simultaneously to port 0 and 1. If the flow entry is bound to hardware, handlers inside the flow object do the updates. To query the packet and byte counters or get the timestamp of the last matching packet, you can call *get_statistics(...)* as shown in line 25. The *Boost Date Time* library [12] is used to represent the time as a universal time (UTC).

Deleting a flow is done by destroying the shared pointer to the *flow* object. This can be done by explicitly calling *reset()* (line 27) or by letting the shared pointers go out of scope. If the flow entry was bound to hardware, it is automatically uninstalled there.

Priority conflicts and unsupported flow entries

For every flow entry, the library *sdn_dataplane* supports a 32 bit priority. It can be set by calling *set_priority(...)* on the flow object. The *OpenFlow* core on the other hand has fixed tables priorities. A matching entry in the CAM has always priority over the TCAM. This scheme was introduced in OpenFlow 1.0, where direct match entries always have highest priority. Later OpenFlow versions allow arbitrary 16 bit priorities. When you add a flow entry using *add_flow(...)* from the class *sdn_dataplane* it automatically chooses a compatible hardware table. The first flow entry pushed to that hardware table locks the given priority to that table. This limits the usable priorities for the remaining tables. For example, if a flow entry was added to the CAM with, let's say priority 1000, only priorities 0-999 remain available to be used by the TCAM, as every flow entry in the TCAM must have lower priority. If you add a wildcard entry with priority 1001 anyway, this results in a priority conflict. This flow can not be bound to hardware. Beyond this, the new flow entry with priority 1001 can not be simply ignored, as it might cause side effects. For example this entry could sort out some packets that should not be processed by flow entries with lower priorities. The solution is that the class *sdn_dataplane* has to unbind all flows with priority lower than 1001 from all hardware tables. This results in sending the packets that would have matched to those deactivated entries to the virtual network interfaces. The software switch has to handle those packets. As soon as such a conflict disappears (for example flow entry with priority 1001 was deleted), all those deactivated flow entries are tried to be rebound to hardware tables.

As long as you use the priority schemes from OpenFlow 1.0 such a conflict cannot appear. If you start using arbitrary priorities you should keep this in mind. If the logging severity level is set to *debug*, the library will print a warning message to *stdout* in such a case.

Using the same deactivation mechanism, it is also possible to add “*unsupported*” flow entries. Whenever a flow entry cannot be expressed with the capabilities of our parser (and consequently not with the methods of the class *flow*), you can set the flow entry to *unsupported* by calling *set_unsupported()* on the flow object. This causes all potentially conflicting flow entries to be unbound from the hardware tables, as long as this *unsupported* flow entry exists.

5.3.2 Manual Flow Entry Installation

The automatic flow entry management is useful whenever a software OpenFlow switch should be extended with hardware acceleration. We use these mechanisms to extend Open vSwitch. If you prefer to keep control over the installation of flows to the hardware, you can also directly use the classes *cam* and *tcam*. Both classes also have the method *add_flow(...)* which returns *true* or *false*, depending on the success of the installation.

5.3.3 The C-wrapper

Many projects are written in plain C. The main purpose of this C wrapper ¹ is to ease the integration into such projects.

The shared pointers of created objects are stored in data structures and are mapped to 64 bit IDs. So the user has to store these IDs in the C code instead of the shared pointers. This is less error prone as simply dealing with plain pointers. If a function is called with an invalid ID, this simply produces an error message instead of an undefined behavior. The wrapper catches exceptions and transforms the caught value of the *error* enumeration class to an error code, which is then the return code. All important methods of the classes *sdn_dataplane* and *flow* are wrapped.

5.3.4 Gathering of Statistics

The statistics core inside the *OpenFlow* core counts the received packets, the received bytes and stores a timestamp whenever a packet is received. This is done for every flow entry of all hardware flow tables. This means that we have one continuous address space that holds the statistics for all flows of all hardware tables. The class *flow_statistics*² attaches to the statistics core. This class is instantiated by the class *sdn_dataplane*. Inside the core, all counters have a width of 32 bit. In software all counters associated with active flow entries are periodically polled to detect counter changes before an overflow occurs and to virtually increase the counter width to 64 bit. The FPGA core has no functionality to reset these counters. Resetting is done by storing an offset value in software.

The core provides a free running 32 bit timestamp counter, which is also periodically polled by the software to extend it to 64 bit. By detecting increases of packet counters, the reception point in time can be ascribed to a 64 bit timestamp value. This timestamp is then converted to a UTC time which can be accessed by the user.

During the construction process of the class *sdn_dataplane*, the appropriate counter sets are attached to the instances of the classes *cam* and *tcam*. These classes in turn attach single counter sets to flow objects, whenever a flow was bound to hardware. This concept specially supports the demands of the TCAM where the priority of the flow entry is determined by the position inside the TCAM table. Whenever a flow entry has to be moved to a different position (e.g. because a flow with a higher priority was added), the statistics also have to be moved.

5.3.5 Integration into Open vSwitch

We used *Open vSwitch* 2.3.1, which is the latest long-term-support version at the time of writing this. This version fully supports *OpenFlow* 1.3. The integration of our library was done by reverse engineering the inner details of Open vSwitch. The supplied interfaces were not suitable for our purposes as we just wanted to extend a fully working software switch. We tried to change as less lines of code as possible in the sources that everything can easily be ported to later Open vSwitch versions. Instead we added an additional C file which does all

¹Source: lib/sw/contrib/drivers/nf10_upb_sdn_dataplane_v1_00_a/src/libsdn_dataplane/sdn_dp_cwrapper.h

²Source: lib/sw/contrib/drivers/nf10_upb_sdn_dataplane_v1_00_a/src/libsdn_dataplane/flow_statistics.h

the translations between their and our data structures.

During the initialization phase of Open vSwitch we check if the used network interfaces are virtual ports of the NetFPGA. If this is the case, we create instances of the class *sdn_dataplane* for every used NetFPGA. This is done by using the C wrapper.

Whenever a flow entry is added in Open vSwitch, we check if the involved ports reside on a NetFPGA card. If this is the case we try to transform the flow entry to our representation of a flow entry. This doesn't have to be necessarily possible. Open vSwitch supports OpenFlow 1.3. We do not prohibit OpenFlow controllers to use features of these later OpenFlow version we can not support in hardware. Whenever header fields, we do not support, do not just have a wildcard, we have to add an *unsupported* flow entry (with all those consequences discussed earlier). If there is just an unsupported action (we only support forwarding in hardware), this action can easily be transformed into a "do it in software" action.

If a flow entry inside Open vSwitch is to be modified or deleted, we check if this entry is also handled by our library. In that case we perform the same operations there. A statistics request for an accelerated flow is handled by combining the statistic data. Packet and byte counters are added and the latest timestamp of a received packet is used. Hard and soft timeouts for flow entries are handled by Open vSwitch. A timer periodically checks for flows timing out. We extended these checks to also query our library and combine the results.

5.3.6 Current Limitations

The OpenFlow standard specifies some statistics that have to be gathered for every port. This is completely implemented on the FPGA but it is currently not implemented in the software. However, most parts of the infrastructure to support this already exists.

6 Notes for Future Work

During our work we stumbled across several issues that people should be aware of if they plan to do further development using our framework.

- The *Place and Route* tools provided by Xilinx ISE 14.7 need some assistance by specifying area constraints for some cores to achieve good results. Otherwise parts of the design are often spread over large areas of the FPGA which leads to high signal delay and therefore failing timing constraints.
- Xilinx ISE 14.7 is not able to work with SystemVerilog code. As we decided to use SystemVerilog in our design we had to switch from Xilinx XST to Synopsys Synplify for synthesis.
- The interconnect produces data errors, especially when used with higher clock rates. This also depends on the cable used and sometimes can be influenced by wiggling the connector.
- The manufacturer of the interconnect cables specifies a lifetime of at most 50 plug cycles per cable. Frequent plugging should therefore be avoided.

Bibliography

- [1] University of Paderborn, Research Group Computer Networks, “PG On-the-fly Networking for Big Data,” date: 15.10.2014. [Online]. Available: <http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-rechnernetze/lehre/lehrveranstaltungen/pg-on-the-fly-networking-for-big-data.html>
- [2] NetFPGA, “NetFPGA-10G Information,” date: 15.10.2014. [Online]. Available: http://netfpga.org/10G_specs.html
- [3] NetFPGA Wiki, “Home_NetFPGA 10G,” date: 15.10.2014. [Online]. Available: https://github.com/NetFPGA/NetFPGA-public/wiki/Home_NetFPGA-10G
- [4] Open vSwitch, date: 15.10.2014. [Online]. Available: <http://openvswitch.org>
- [5] NetFPGA Wiki, “PCIE Programming,” date: 16.10.2014. [Online]. Available: <https://github.com/NetFPGA/NetFPGA-public/wiki/PCIE-Programming>
- [6] NoxRepo.org, “POX OpenFlow controller,” date: 15.10.2014. [Online]. Available: <https://http://www.noxrepo.org/pox/about-pox/>
- [7] Open Networking Foundation, “OpenFlow Switch Specification 1.0.0,” Dec 2009. [Online]. Available: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf>
- [8] K. Nichols, S. Blake, F. Baker, and D. Black, “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers,” RFC 2474 (Proposed Standard), Internet Engineering Task Force, Dec. 1998, updated by RFCs 3168, 3260. [Online]. Available: <http://www.ietf.org/rfc/rfc2474.txt>
- [9] Wikipedia, “Birthday Problem,” date: 15.10.2014. [Online]. Available: https://en.wikipedia.org/wiki/Birthday_problem
- [10] boost.org, “Boost C++ library,” date: 15.10.2014. [Online]. Available: <http://www.boost.org/>
- [11] —, “Boost Smart Ptr,” date: 15.10.2014. [Online]. Available: http://www.boost.org/doc/libs/release/libs/smart_ptr/smart_ptr.htm
- [12] —, “Boost Date Time,” date: 15.10.2014. [Online]. Available: http://www.boost.org/doc/libs/release/libs/date_time/