Corundum

Corundum contributors

Sep 23, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

Corundum is an open-source, high-performance *FPGA*-based *NIC* and platform for in-network compute. Features include a high performance datapath, 10G/25G/100G Ethernet, PCI express gen 3, a custom, high performance, tightly-integrated *PCIe DMA* engine, many (1000+) transmit, receive, completion, and event queues, scatter/gather DMA, *MSI*, multiple interfaces, multiple ports per interface, per-port transmit scheduling including high precision TDMA, flow hashing, *RSS*, checksum offloading, and native IEEE 1588 *PTP* timestamping. A Linux driver is included that integrates with the Linux networking stack. Development and debugging is facilitated by an extensive simulation framework that covers the entire system from a simulation model of the driver and PCI express interface on one side to the Ethernet interfaces on the other side.

Corundum has several unique architectural features. First, transmit, receive, completion, and event queue states are stored efficiently in block RAM or ultra RAM, enabling support for thousands of individually-controllable queues. These queues are associated with interfaces, and each interface can have multiple ports, each with its own independent scheduler. This enables extremely fine-grained control over packet transmission. Coupled with PTP time synchronization, this enables high precision TDMA.

Corundum also provides an application section for implementing custom logic. The application section has a dedicated PCIe BAR for control and a number of interfaces that provide access to the core datapath and DMA infrastructure.

The latest source code is available from the Corundum GitHub repository[1]. To stay up to date with the latest developments and get support, consider joining the mailing list[2] and Zulip[3].

Corundum currently supports devices from both Xilinx and Intel, on boards from several different manufacturers. Designs are included for the following FPGA boards; see *Device list* (page 131) for more details:

- Alpha Data ADM-PCIE-9V3 (Xilinx Virtex UltraScale+ XCVU3P)
- Dini Group DNPCIe_40G_KU_LL_2QSFP (Xilinx Kintex UltraScale XCKU040)
- Cisco Nexus K35-S (Xilinx Kintex UltraScale XCKU035)
- Cisco Nexus K3P-S (Xilinx Kintex UltraScale+ XCKU3P)
- Cisco Nexus K3P-Q (Xilinx Kintex UltraScale+ XCKU3P)
- Silicom fb2CG@KU15P (Xilinx Kintex UltraScale+ XCKU15P)
- NetFPGA SUME (Xilinx Virtex 7 XC7V690T)
- BittWare 250-SoC (Xilinx Zynq UltraScale+ XCZU19EG)
- BittWare XUP-P3R (Xilinx Virtex UltraScale+ XCVU9P)
- Intel Stratix 10 MX dev kit (Intel Stratix 10 MX 2100)
- Intel Stratix 10 DX dev kit (Intel Stratix 10 DX 2800)

[1] https://github.com/corundum/corundum
[2] https://groups.google.com/d/forum/corundum-nic
[3] https://corundum.zulipchat.com/

- Terasic DE10-Agilex (Intel Agilex F 014)
- Xilinx Alveo U50 (Xilinx Virtex UltraScale+ XCU50)
- Xilinx Alveo U200 (Xilinx Virtex UltraScale+ XCU200)
- Xilinx Alveo U250 (Xilinx Virtex UltraScale+ XCU250)
- Xilinx Alveo U280 (Xilinx Virtex UltraScale+ XCU280)
- Xilinx VCU108 (Xilinx Virtex UltraScale XCVU095)
- Xilinx VCU118 (Xilinx Virtex UltraScale+ XCVU9P)
- Xilinx VCU1525 (Xilinx Virtex UltraScale+ XCVU9P)
- Xilinx ZCU102 (Xilinx Zynq UltraScale+ XCZU9EG)
- Xilinx ZCU106 (Xilinx Zynq UltraScale+ XCZU7EV)

1.1 Publications

- A. Forencich, A. C. Snoeren, G. Porter, G. Papen, *Corundum: An Open-Source 100-Gbps NIC,* in FCCM'20. (FCCM Paper[4], FCCM Presentation[5])
- J. A. Forencich, *System-Level Considerations for Optical Switching in Data Center Networks*. (Thesis[6])

1.2 Citation

If you use Corundum in your project, please cite one of the following papers and/or link to the project on GitHub:

```
@inproceedings{forencich2020fccm,
    author = {Alex Forencich and Alex C. Snoeren and George Porter and George Papen},
    title = {Corundum: An Open-Source {100-Gbps} {NIC}},
    booktitle = {28th IEEE International Symposium on Field-Programmable Custom␣
↪Computing Machines},
    year = {2020},
}

@phdthesis{forencich2020thesis,
    author = {John Alexander Forencich},
    title = {System-Level Considerations for Optical Switching in Data Center Networks},
    school = {UC San Diego},
    year = {2020},
    url = {https://escholarship.org/uc/item/3mc9070t},
}
```

[4] https://www.cse.ucsd.edu/~snoeren/papers/corundum-fccm20.pdf
[5] https://www.fccm.org/past/2020/forums/topic/corundum-an-open-source-100-gbps-nic/
[6] https://escholarship.org/uc/item/3mc9070t

CHAPTER

TWO

GETTING STARTED

2.1 Join the Corundum community

To stay up to date with the latest developments and get support, consider joining the mailing list[7] and Zulip[8].

2.2 Obtaining the source code

The main upstream repository for Corundum[9] is located on GitHub[10]. There are two main ways to download the source code - downloading an archive, or cloning with git.

To clone via HTTPS, run:

```
$ git clone https://github.com/corundum/corundum.git
```

To clone via SSH, run:

```
$ git clone git@github.com:corundum/corundum.git
```

Alternatively, download a zip file:

```
$ wget https://github.com/corundum/corundum/archive/refs/heads/master.zip
$ unzip master.zip
```

Or a gzipped tar archive file:

```
$ wget https://github.com/corundum/corundum/archive/refs/heads/master.tar.gz
$ tar xvf master.tar.gz
```

There is also a mirror of the repository[11] on gitee[12], here are the equivalent commands:

```
$ git clone https://gitee.com/alexforencich/corundum.git
$ git clone git@gitee.com:alexforencich/corundum.git
$ wget https://gitee.com/alexforencich/corundum/repository/archive/master.zip
$ wget https://gitee.com/alexforencich/corundum/repository/archive/master.tar.gz
```

[7] https://groups.google.com/d/forum/corundum-nic
[8] https://corundum.zulipchat.com/
[9] https://github.com/corundum/corundum/
[10] https://github.com/
[11] https://gitee.com/alexforencich/corundum/
[12] https://gitee.com/

2.3 Setting up the FPGA development environment

Corundum currently uses Icarus Verilog[13] and cocotb[14] for simulation. Linux is the recommended operating system for a development environment due to the use of symlinks (which can cause problems on Windows as they are not supported by windows filesystems), however WSL may also work well.

The required system packages are:

- Python 3 (`python` or `python3`, depending on distribution)
- Icarus Verilog (`iverilog`)
- GTKWave (`gtkwave`)

The required python packages are:

- `cocotb`
- `cocotb-bus`
- `cocotb-test`
- `cocotbext-axi`
- `cocotbext-eth`
- `cocotbext-pcie`
- `pytest`
- `scapy`

Recommended additional python packages:

- `tox` (to run pytest inside a python virtual environment)
- `pytest-xdist` (to run tests in parallel with *pytest -n auto*)
- `pytest-sugar` (makes pytest output a bit nicer)

It is recommended to install the required system packages via the system package manager (`apt`, `yum`, `pacman`, etc.) and then install the required Python packages as user packages via `pip` (or `pip3`, depending on distribution).

2.4 Running tests

Once the packages are installed, you should be able to run the tests. There are several ways to do this.

First, all tests can be run by runing `tox` in the repo root. In this case, tox will set up a python virtual environment and install all python dependencies inside the virtual environment. Additionally, tox will run pytest as `pytest -n auto` so it will run tests in parallel on multiple CPUs.

```
$ cd /path/to/corundum/
$ tox
py3 create: /home/alex/Projects/corundum/.tox/py3
py3 installdeps: pytest == 6.2.5, pytest-xdist == 2.4.0, pytest-split == 0.4.0, cocotb␣
↪== 1.6.1, cocotb-test == 0.2.1, cocotbext-axi == 0.1.18, cocotbext-eth == 0.1.18,␣
↪cocotbext-pcie == 0.1.20, scapy == 2.4.5
py3 installed: attrs==21.4.0,cocotb==1.6.1,cocotb-bus==0.2.1,cocotb-test==0.2.1,
```

(continues on next page)

[13] http://iverilog.icarus.com/
[14] https://github.com/cocotb/cocotb

(continued from previous page)

```
↪cocotbext-axi==0.1.18,cocotbext-eth==0.1.18,cocotbext-pcie==0.1.20,execnet==1.9.0,
↪iniconfig==1.1.1,packaging==21.3,pluggy==1.0.0,py==1.11.0,pyparsing==3.0.7,pytest==6.2.
↪5,pytest-forked==1.4.0,pytest-split==0.4.0,pytest-xdist==2.4.0,scapy==2.4.5,toml==0.10.
↪2
py3 run-test-pre: PYTHONHASHSEED='4023917175'
py3 run-test: commands[0] | pytest -n auto
============================= test session starts ==============================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
cachedir: .tox/py3/.pytest_cache
rootdir: /home/alex/Projects/corundum, configfile: tox.ini, testpaths: fpga, fpga/app
plugins: forked-1.4.0, split-0.4.0, cocotb-test-0.2.1, xdist-2.4.0
gw0 [69] / gw1 [69] / gw2 [69] / gw3 [69] / gw4 [69] / gw5 [69] / gw6 [69] / gw7 [69] /␣
↪gw8 [69] / gw9 [69] / gw10 [69] / gw11 [69] / gw12 [69] / gw13 [69] / gw14 [69] / gw15␣
↪[69] / gw16 [69] / gw17 [69] / gw18 [69] / gw19 [69] / gw20 [69] / gw21 [69] / gw22␣
↪[69] / gw23 [69] / gw24 [69] / gw25 [69] / gw26 [69] / gw27 [69] / gw28 [69] / gw29␣
↪[69] / gw30 [69] / gw31 [69] / gw32 [69] / gw33 [69] / gw34 [69] / gw35 [69] / gw36␣
↪[69] / gw37 [69] / gw38 [69] / gw39 [69] / gw40 [69] / gw41 [69] / gw42 [69] / gw43␣
↪[69] / gw44 [69] / gw45 [69] / gw46 [69] / gw47 [69] / gw48 [69] / gw49 [69] / gw50␣
↪[69] / gw51 [69] / gw52 [69] / gw53 [69] / gw54 [69] / gw55 [69] / gw56 [69] / gw57␣
↪[69] / gw58 [69] / gw59 [69] / gw60 [69] / gw61 [69] / gw62 [69] / gw63 [69]
.....................................................................    [100%]
======================== 69 passed in 1534.87s (0:25:34) =======================
_____________________________________ summary _____________________________________
  py3: commands succeeded
  congratulations :)
```

Second, all tests can be run by running `pytest` in the repo root. Running as `pytest -n auto` is recommended to run multiple tests in parallel on multiple CPUs.

```
$ cd /path/to/corundum/
$ pytest -n auto
============================= test session starts ==============================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-0.13.1
rootdir: /home/alex/Projects/corundum, configfile: tox.ini, testpaths: fpga, fpga/app
plugins: split-0.3.0, parallel-0.1.0, cocotb-test-0.2.0, forked-1.3.0, metadata-1.11.0,␣
↪xdist-2.4.0, html-3.1.1, cov-2.12.1, flake8-1.0.7
gw0 [69] / gw1 [69] / gw2 [69] / gw3 [69] / gw4 [69] / gw5 [69] / gw6 [69] / gw7 [69] /␣
↪gw8 [69] / gw9 [69] / gw10 [69] / gw11 [69] / gw12 [69] / gw13 [69] / gw14 [69] / gw15␣
↪[69] / gw16 [69] / gw17 [69] / gw18 [69] / gw19 [69] / gw20 [69] / gw21 [69] / gw22␣
↪[69] / gw23 [69] / gw24 [69] / gw25 [69] / gw26 [69] / gw27 [69] / gw28 [69] / gw29␣
↪[69] / gw30 [69] / gw31 [69] / gw32 [69] / gw33 [69] / gw34 [69] / gw35 [69] / gw36␣
↪[69] / gw37 [69] / gw38 [69] / gw39 [69] / gw40 [69] / gw41 [69] / gw42 [69] / gw43␣
↪[69] / gw44 [69] / gw45 [69] / gw46 [69] / gw47 [69] / gw48 [69] / gw49 [69] / gw50␣
↪[69] / gw51 [69] / gw52 [69] / gw53 [69] / gw54 [69] / gw55 [69] / gw56 [69] / gw57␣
↪[69] / gw58 [69] / gw59 [69] / gw60 [69] / gw61 [69] / gw62 [69] / gw63 [69]
.....................................................................    [100%]
======================== 69 passed in in 2032.42s (0:33:52) ====================
```

Third, groups of tests can be run by running `pytest` in a subdirectory. Running as `pytest -n auto` is recommended to run multiple tests in parallel on multiple CPUs.

```
$ cd /path/to/corundum/fpga/common/tb/rx_hash
```

(continues on next page)

(continued from previous page)

```
$ pytest -n 4
============================= test session starts ==============================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-0.13.1
rootdir: /home/alex/Projects/corundum, configfile: tox.ini
plugins: split-0.3.0, parallel-0.1.0, cocotb-test-0.2.0, forked-1.3.0, metadata-1.11.0,␣
↪xdist-2.4.0, html-3.1.1, cov-2.12.1, flake8-1.0.7
gw0 [2] / gw1 [2] / gw2 [2] / gw3 [2]
..                                                                       [100%]
============================== 2 passed in 37.49s ==============================
```

Finally, individual tests can be run by runing `make`. This method provides the capability of overriding parameters and enabling waveform dumps in FST format that are viewable in gtkwave.

```
$ cd /path/to/corundum/fpga/common/tb/rx_hash
$ make WAVES=1
make -f Makefile results.xml
make[1]: Entering directory '/home/alex/Projects/corundum/fpga/common/tb/rx_hash'
echo 'module iverilog_dump();' > iverilog_dump.v
echo 'initial begin' >> iverilog_dump.v
echo '    $dumpfile("rx_hash.fst");' >> iverilog_dump.v
echo '    $dumpvars(0, rx_hash);' >> iverilog_dump.v
echo 'end' >> iverilog_dump.v
echo 'endmodule' >> iverilog_dump.v
/usr/bin/iverilog -o sim_build/sim.vvp -D COCOTB_SIM=1 -s rx_hash -P rx_hash.DATA_
↪WIDTH=64 -P rx_hash.KEEP_WIDTH=8 -s iverilog_dump -f sim_build/cmds.f -g2012   ../../
↪rtl/rx_hash.v iverilog_dump.v
MODULE=test_rx_hash TESTCASE= TOPLEVEL=rx_hash TOPLEVEL_LANG=verilog \
         /usr/bin/vvp -M /home/alex/.local/lib/python3.9/site-packages/cocotb/libs -m␣
↪libcocotbvpi_icarus   sim_build/sim.vvp -fst
     -.--ns INFO     cocotb.gpi                         ..mbed/gpi_embed.cpp:76   in set_
↪program_name_in_venv        Did not detect Python virtual environment. Using system-
↪wide Python interpreter
     -.--ns INFO     cocotb.gpi                         ../gpi/GpiCommon.cpp:99   in gpi_
↪print_registered_impl       VPI registered
     0.00ns INFO     Running on Icarus Verilog version 11.0 (stable)
     0.00ns INFO     Running tests with cocotb v1.7.0.dev0 from /home/alex/.local/lib/
↪python3.9/site-packages/cocotb
     0.00ns INFO     Seeding Python random module with 1643529566
     0.00ns INFO     Found test test_rx_hash.run_test
     0.00ns INFO     Found test test_rx_hash.run_test
     0.00ns INFO     Found test test_rx_hash.run_test
     0.00ns INFO     Found test test_rx_hash.run_test
     0.00ns INFO     Found test test_rx_hash.run_test
     0.00ns INFO     Found test test_rx_hash.run_test
     0.00ns INFO     Found test test_rx_hash.run_test
     0.00ns INFO     Found test test_rx_hash.run_test
     0.00ns INFO     running run_test (1/8)
     0.00ns INFO     AXI stream source
     0.00ns INFO     cocotbext-axi version 0.1.19
     0.00ns INFO     Copyright (c) 2020 Alex Forencich
     0.00ns INFO     https://github.com/alexforencich/cocotbext-axi
     0.00ns INFO     AXI stream source configuration:
```

(continues on next page)

(continued from previous page)

```
     0.00ns INFO       Byte size: 8 bits
     0.00ns INFO       Data width: 64 bits (8 bytes)
     0.00ns INFO     AXI stream source signals:
     0.00ns INFO       tdata width: 64 bits
     0.00ns INFO       tdest: not present
     0.00ns INFO       tid: not present
     0.00ns INFO       tkeep width: 8 bits
     0.00ns INFO       tlast width: 1 bits
     0.00ns INFO       tready: not present
     0.00ns INFO       tuser: not present
     0.00ns INFO       tvalid width: 1 bits
     0.00ns INFO     Reset de-asserted
     0.00ns INFO     Reset de-asserted
FST info: dumpfile rx_hash.fst opened for output.
     4.00ns INFO     Reset asserted
     4.00ns INFO     Reset asserted
    12.00ns INFO     Reset de-asserted
    12.00ns INFO     Reset de-asserted
    20.00ns INFO     TX frame: AxiStreamFrame(tdata=bytearray(b'\xda\xd1\xd2\xd3\xd4\
↪xd5ZQRSTU\x90\x00\x00'), tkeep=None, tid=None, tdest=None, tuser=None, sim_time_
↪start=20000, sim_time_end=None)
    28.00ns INFO     TX frame: AxiStreamFrame(tdata=bytearray(b'\xda\xd1\xd2\xd3\xd4\
↪xd5ZQRSTU\x90\x00\x00\x01'), tkeep=None, tid=None, tdest=None, tuser=None, sim_time_
↪start=28000, sim_time_end=None)
    36.00ns INFO     TX frame: AxiStreamFrame(tdata=bytearray(b'\xda\xd1\xd2\xd3\xd4\
↪xd5ZQRSTU\x90\x00\x00\x01\x02'), tkeep=None, tid=None, tdest=None, tuser=None, sim_
↪time_start=36000, sim_time_end=None)
    40.00ns INFO     RX hash: 0x00000000 (expected: 0x00000000) type: HashType.0␣
↪(expected: HashType.0)
    48.00ns INFO     TX frame: AxiStreamFrame(tdata=bytearray(b'\xda\xd1\xd2\xd3\xd4\
↪xd5ZQRSTU\x90\x00\x00\x01\x02\x03'), tkeep=None, tid=None, tdest=None, tuser=None, sim_
↪time_start=48000, sim_time_end=None)
    48.00ns INFO     RX hash: 0x00000000 (expected: 0x00000000) type: HashType.0␣
↪(expected: HashType.0)
    56.00ns INFO     RX hash: 0x00000000 (expected: 0x00000000) type: HashType.0␣
↪(expected: HashType.0)


################    skip a very large number of lines    ################


252652.01ns INFO     TX frame: AxiStreamFrame(tdata=bytearray(b'\xda\xd1\xd2\xd3\xd4\
↪xd5ZQRSTU\x08\x00E\x00\x00V\x00\x8b\x00\x00@\x06d\xff\n\x01\x00\x8b\n\x02\x00\x8b\x00\
↪x8b\x10\x8b\x00\x00\x00\x00\x00\x00\x00P\x02 \x00ms\x00\x00\x00\x01\x02\x03\x04\
↪x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\
↪x1c\x1d\x1e\x1f !"#$%&\'()*+,-'), tkeep=None, tid=None, tdest=None, tuser=None, sim_
↪time_start=252652007, sim_time_end=None)
252744.01ns INFO     RX hash: 0xa2a55ee3 (expected: 0xa2a55ee3) type: HashType.TCP|IPV4␣
↪(expected: HashType.TCP|IPV4)
252860.01ns INFO     TX frame: AxiStreamFrame(tdata=bytearray(b'\xda\xd1\xd2\xd3\xd4\
↪xd5ZQRSTU\x08\x00E\x00\x00V\x00\x8c\x00\x00@\x06d\xfc\n\x01\x00\x8c\n\x02\x00\x8c\x00\
↪x8c\x10\x8c\x00\x00\x00\x00\x00\x00\x00P\x02 \x00mo\x00\x00\x00\x01\x02\x03\x04\
```

(continues on next page)

(continued from previous page)

```
↪x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\
↪x1c\x1d\x1e\x1f !"#$%&\'()*+,-'), tkeep=None, tid=None, tdest=None, tuser=None, sim_
↪time_start=252860007, sim_time_end=None)
252952.01ns INFO     RX hash: 0x6308c813 (expected: 0x6308c813) type: HashType.TCP|IPV4␣
↪(expected: HashType.TCP|IPV4)
252960.01ns INFO     run_test passed
252960.01ns INFO    ␣
↪**************************************************************************************
                     ** TEST                            STATUS  SIM TIME (ns)  REAL TIME␣
↪(s)  RATIO (ns/s) **
                    ␣
↪**************************************************************************************
                     ** test_rx_hash.run_test           PASS       11144.00           1.
↪14       9781.95  **
                     ** test_rx_hash.run_test           PASS       44448.00           3.
↪80      11688.88  **
                     ** test_rx_hash.run_test           PASS       12532.00           1.
↪40       8943.27  **
                     ** test_rx_hash.run_test           PASS       49984.00           4.
↪42      11302.44  **
                     ** test_rx_hash.run_test           PASS       13088.00           1.
↪54       8479.38  **
                     ** test_rx_hash.run_test           PASS       52208.00           4.
↪62      11308.18  **
                     ** test_rx_hash.run_test           PASS       13940.00           1.
↪65       8461.27  **
                     ** test_rx_hash.run_test           PASS       55616.00           5.
↪03      11046.45  **
                    ␣
↪**************************************************************************************
                     ** TESTS=8 PASS=8 FAIL=0 SKIP=0             252960.01          25.
↪11      10073.76  **
                    ␣
↪**************************************************************************************

make[1]: Leaving directory '/home/alex/Projects/corundum/fpga/common/tb/rx_hash'
```

2.5 Setting up the FPGA build environment (Vivado)

Building FPGA configurations for Xilinx devices requires Vivado[15]. Linux is the recommended operating system for a build environment due to the use of symlinks (which can cause problems on Windows) and makefiles for build automation. Additionally, Vivado uses more CPU cores for building on Linux than on Windows. It is not recommended to run Vivado inside of a virtual machine as Vivado uses a significant amount of RAM during the build process. Download and install the appropriate version of Vivado. Make sure to install device support for your target device; support for other devices can be disabled to save disk space.

Licenses may be required, depending on the target device. A bare install of Vivado without any licenses runs in "WebPACK" mode and has limited device support. If your target device is on the WebPACK device list[16], then no Vivado license is required. Otherwise, you will need access to a Vivado license to build the design.

[15] https://www.xilinx.com/products/design-tools/vivado.html
[16] https://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html#architecture

Additionally, the 100G MAC IP cores on UltraScale and UltraScale+ require separate licenses. These licenses are free of charge, and can be generated for UltraScale[17] and UltraScale+[18]. If your target design uses the 100G CMAC IP, then you will need one of these licenses to build the design.

For example: if you want to build a 100G design for an Alveo U50, you will not need a Vivado license as the U50 is supported under WebPACK, but you will need to generate a (free-of-charge) license for the CMAC IP for UltraScale+.

Before building a design with Vivado, you'll have to source the appropriate settings file. For example:

```
$ source /opt/Xilinx/Vivado/2020.2/settings64.sh
$ make
```

2.6 Building the FPGA configuration

Each design contains a set of makefiles for automating the build process. To use the makefile, simply source the settings file for the required toolchain and then run `make`. Note that the repository makes significant use of symbolic links, so it is highly recommended to build the design under Linux.

For example:

```
$ cd /path/to/corundum/fpga/mqnic/[board]/fpga_[variant]/fpga
$ source /opt/Xilinx/Vivado/2020.2/settings64.sh
$ make
```

2.7 Building the driver

To build the driver, you will first need to install the required compiler and kernel source code packages. After these packages are installed, simply run `make`.

```
$ cd /path/to/corundum/modules/mqnic
$ make
```

Note that the driver currently does not support RHEL, centos, and related distributions that use very old and significantly modified kernels where the reported kernel version number is not a reliable of the internal kernel API.

2.8 Building the userspace tools

To build the driver, you will first need to install the required compiler packages. After these packages are installed, simply run `make`.

```
$ cd /path/to/corundum/utils
$ make
```

[17] https://www.xilinx.com/products/intellectual-property/cmac.html
[18] https://www.xilinx.com/products/intellectual-property/cmac_usplus.html

2.9 Setting up the PetaLinux build environment

Building PetaLinux projects for Xilinx devices requires PetaLinux Tools[19]. Linux is the recommended operating system for a build environment due to the use of symlinks (which can cause problems on Windows) and makefiles for build automation. Download and install the appropriate version of PetaLinux Tools. Make sure to install device support for your target device; support for other devices can be disabled to save disk space.

An example for a PetaLinux project in Corundum is accompanying the FPGA design using the Xilinx ZynqMP SoC as host system for mqnic on the Xilinx ZCU106 board. See *fpga/mqnic/ZCU106/fpga_zynqmp/README.md*.

Before building a PetaLinux project, you'll have to source the appropriate settings file. For example:

```
$ source /opt/Xilinx/PetaLinux/2021.1/settings.sh
$ make -C path/to/petalinux/project build-boot
```

2.10 Loading the FPGA design

There are three main ways for loading Corundum on to an FPGA board. The first is via JTAG, into volatile FPGA configuration memory. This is best for development and debugging, especially when complemented with a baseline design with the same PCIe interface configuration stored in flash. The second is via indirect JTAG, into nonvolatile on-card flash memory. This is quite slow. The third is via PCI express, into nonvolatile on-card memory. This is the fastest method of programming the flash, but it requires the board to already be running the Corundum design.

For a card that's not already running Corundum, there are two options for programming the flash. The first is to use indirect JTAG, but this is very slow. The second is to first load the design via JTAG into volatile configuration memory, then perform a warm reboot, and finally write the design into flash via PCIe with the `mqnic-fw` utility.

Loading the design via JTAG into volatile configuration memory with Vivado is straightforward: install the card into a host computer, attach the JTAG cable, power up the host computer, and use Vivado to connect and load the bit file into the FPGA. When using the makefile, run `make program` to program the device. If physical access is a problem, it is possible to run a hardware server instance on the host computer and connect to the hardware server over the network. Once the design is loaded into the FPGA, perform either a hot reset (via `pcie_hot_reset.sh` or `mqnic-fw -t`, but only if the card was enumerated at boot and the PCIe configuration has not changed) or a warm reboot.

Loading the design via indirect JTAG into nonvolatile memory with Vivado requires basically the same steps as loading it into volatile configuration memory, the main difference is that the configuration flash image must first be generated by running `make fpga.mcs` after using make to generate the bit file. Once this file is generated, connect with the hardware manager, add the configuration memory device (check the makefile for the part number), and program the flash. After the programming operation is complete, boot the FPGA from the configuration memory, either via Vivado (right click -> boot from configuration memory) or by performing a cold reboot (full shut down, then power on). When using the makefile, run `make flash` to generate the flash images, program the flash via indirect JTAG, and boot the FPGA from the configuration memory. Finally, reboot the host computer to re-enumerate the PCIe bus.

Loading the design via PCI express is straightforward: use the `mqnic-fw` utility to load the bit file into flash, then trigger an FPGA reboot to load the new design. This does not require the kernel module to be loaded. With the kernel module loaded, point `mqnic-fw` either to `/dev/mqnic<n>` or to one of the associated network interfaces. Without the kernel module loaded, point `mqnic-fw` either to the raw PCIe ID, or to `/sys/bus/pci/devices/<pcie-id>/resource0`; check `lspci` for the PCIe ID. Use `-w` to specify the bit file to load, then `-b` to command the FPGA to reset and reload its configuration from flash. You can also use `-t` to trigger a hot reset to reset the design.

Query device information with `mqnic-fw`, with no kernel module loaded:

[19] https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

```
$ sudo ./mqnic-fw -d 81:00.0
PCIe ID (device): 0000:81:00.0
PCIe ID (upstream port): 0000:80:01.1
FPGA ID: 0x04b77093
FPGA part: XCU50
FW ID: 0x00000000
FW version: 0.0.1.0
Board ID: 0x10ee9032
Board version: 1.0.0.0
Build date: 2022-01-05 08:33:23 UTC (raw 0x61d557d3)
Git hash: ddd7e639
Release info: 00000000
Flash type: SPI
Flash format: 0x00048100
Data width: 4
Manufacturer ID: 0x20
Memory type: 0xbb
Memory capacity: 0x21
Flash size: 128 MB
Write buffer size: 256 B
Erase block size: 4096 B
Flash segment 0: start 0x00000000 length 0x01002000
Flash segment 1: start 0x01002000 length 0x06ffe000
Selected: segment 1 start 0x01002000 length 0x06ffe000
```

Write design into nonvolatile flash memory with `mqnic-fw`, with no kernel module loaded:

```
$ sudo ./mqnic-fw -d 81:00.0 -w ../fpga/mqnic/AU50/fpga_100g/fpga/fpga.bit
PCIe ID (device): 0000:81:00.0
PCIe ID (upstream port): 0000:80:01.1
FPGA ID: 0x04b77093
FPGA part: XCU50
FW ID: 0x00000000
FW version: 0.0.1.0
Board ID: 0x10ee9032
Board version: 1.0.0.0
Build date: 2022-01-05 08:33:23 UTC (raw 0x61d557d3)
Git hash: ddd7e639
Release info: 00000000
Flash type: SPI
Flash format: 0x00048100
Data width: 4
Manufacturer ID: 0x20
Memory type: 0xbb
Memory capacity: 0x21
Flash size: 128 MB
Write buffer size: 256 B
Erase block size: 4096 B
Flash segment 0: start 0x00000000 length 0x01002000
Flash segment 1: start 0x01002000 length 0x06ffe000
Selected: segment 1 start 0x01002000 length 0x06ffe000
Erasing flash...
Start address: 0x01002000
```

(continues on next page)

(continued from previous page)

```
Length: 0x01913000
Erase address 0x02910000, length 0x00005000 (99%)
Writing flash...
Start address: 0x01002000
Length: 0x01913000
Write address 0x02910000, length 0x00005000 (99%)
Verifying flash...
Start address: 0x01002000
Length: 0x01913000
Read address 0x02910000, length 0x00005000 (99%)
Programming succeeded!
```

Reboot FPGA to load design from flash with `mqnic-fw`, with no kernel module loaded:

```
$ sudo ./mqnic-fw -d 81:00.0 -b
PCIe ID (device): 0000:81:00.0
PCIe ID (upstream port): 0000:80:01.1
FPGA ID: 0x04b77093
FPGA part: XCU50
FW ID: 0x00000000
FW version: 0.0.1.0
Board ID: 0x10ee9032
Board version: 1.0.0.0
Build date: 2022-01-05 08:33:23 UTC (raw 0x61d557d3)
Git hash: ddd7e639
Release info: 00000000
Flash type: SPI
Flash format: 0x00048100
Data width: 4
Manufacturer ID: 0x20
Memory type: 0xbb
Memory capacity: 0x21
Flash size: 128 MB
Write buffer size: 256 B
Erase block size: 4096 B
Flash segment 0: start 0x00000000 length 0x01002000
Flash segment 1: start 0x01002000 length 0x06ffe000
Selected: segment 1 start 0x01002000 length 0x06ffe000
Preparing to reset device...
Disabling PCIe fatal error reporting on port...
No driver bound
Triggering IPROG to reload FPGA...
Removing device...
Performing hot reset on upstream port...
Rescanning on upstream port...
Success, device is online!
```

2.11 Loading the kernel module

Once the kernel module is built, load it with `insmod`:

```
$ sudo insmod mqnic.ko
```

When the driver loads, it will print some debug information:

```
[ 1502.394486] mqnic 0000:81:00.0: mqnic PCI probe
[ 1502.394494] mqnic 0000:81:00.0:  Vendor: 0x1234
[ 1502.394496] mqnic 0000:81:00.0:  Device: 0x1001
[ 1502.394498] mqnic 0000:81:00.0:  Subsystem vendor: 0x10ee
[ 1502.394500] mqnic 0000:81:00.0:  Subsystem device: 0x9032
[ 1502.394501] mqnic 0000:81:00.0:  Class: 0x020000
[ 1502.394504] mqnic 0000:81:00.0:  PCI ID: 0000:81:00.0
[ 1502.394511] mqnic 0000:81:00.0:  Max payload size: 512 bytes
[ 1502.394513] mqnic 0000:81:00.0:  Max read request size: 512 bytes
[ 1502.394515] mqnic 0000:81:00.0:  Link capability: gen 3 x16
[ 1502.394516] mqnic 0000:81:00.0:  Link status: gen 3 x16
[ 1502.394518] mqnic 0000:81:00.0:  Relaxed ordering: enabled
[ 1502.394520] mqnic 0000:81:00.0:  Phantom functions: disabled
[ 1502.394521] mqnic 0000:81:00.0:  Extended tags: enabled
[ 1502.394522] mqnic 0000:81:00.0:  No snoop: enabled
[ 1502.394523] mqnic 0000:81:00.0:  NUMA node: 1
[ 1502.394531] mqnic 0000:81:00.0: 126.016 Gb/s available PCIe bandwidth (8.0 GT/s PCIe␣
↪x16 link)
[ 1502.394554] mqnic 0000:81:00.0: enabling device (0000 -> 0002)
[ 1502.394587] mqnic 0000:81:00.0: Control BAR size: 16777216
[ 1502.396014] mqnic 0000:81:00.0: Device-level register blocks:
[ 1502.396016] mqnic 0000:81:00.0:  type 0xffffffff (v 0.0.1.0)
[ 1502.396019] mqnic 0000:81:00.0:  type 0x0000c000 (v 0.0.1.0)
[ 1502.396021] mqnic 0000:81:00.0:  type 0x0000c004 (v 0.0.1.0)
[ 1502.396023] mqnic 0000:81:00.0:  type 0x0000c080 (v 0.0.1.0)
[ 1502.396025] mqnic 0000:81:00.0:  type 0x0000c120 (v 0.0.1.0)
[ 1502.396027] mqnic 0000:81:00.0:  type 0x0000c140 (v 0.0.1.0)
[ 1502.396029] mqnic 0000:81:00.0:  type 0x0000c150 (v 0.0.1.0)
[ 1502.396038] mqnic 0000:81:00.0: FPGA ID: 0x04b77093
[ 1502.396040] mqnic 0000:81:00.0: FW ID: 0x00000000
[ 1502.396041] mqnic 0000:81:00.0: FW version: 0.0.1.0
[ 1502.396043] mqnic 0000:81:00.0: Board ID: 0x10ee9032
[ 1502.396044] mqnic 0000:81:00.0: Board version: 1.0.0.0
[ 1502.396046] mqnic 0000:81:00.0: Build date: 2022-03-03 07:39:57 UTC (raw: 0x622070cd)
[ 1502.396049] mqnic 0000:81:00.0: Git hash: 8851b3b1
[ 1502.396051] mqnic 0000:81:00.0: Release info: 00000000
[ 1502.396056] mqnic 0000:81:00.0: IF offset: 0x00000000
[ 1502.396057] mqnic 0000:81:00.0: IF count: 1
[ 1502.396059] mqnic 0000:81:00.0: IF stride: 0x01000000
[ 1502.396060] mqnic 0000:81:00.0: IF CSR offset: 0x00080000
[ 1502.396065] mqnic 0000:81:00.0: Resetting Alveo CMS
[ 1502.613317] mqnic 0000:81:00.0: Read 4 MACs from Alveo BMC
[ 1502.624743] mqnic 0000:81:00.0: registered PHC (index 5)
[ 1502.624748] mqnic 0000:81:00.0: Creating interface 0
[ 1502.624798] mqnic 0000:81:00.0: Interface-level register blocks:
```

(continues on next page)

(continued from previous page)

```
[ 1502.624799] mqnic 0000:81:00.0:  type 0x0000c001 (v 0.0.2.0)
[ 1502.624801] mqnic 0000:81:00.0:  type 0x0000c010 (v 0.0.1.0)
[ 1502.624803] mqnic 0000:81:00.0:  type 0x0000c020 (v 0.0.1.0)
[ 1502.624804] mqnic 0000:81:00.0:  type 0x0000c030 (v 0.0.1.0)
[ 1502.624805] mqnic 0000:81:00.0:  type 0x0000c021 (v 0.0.1.0)
[ 1502.624806] mqnic 0000:81:00.0:  type 0x0000c031 (v 0.0.1.0)
[ 1502.624807] mqnic 0000:81:00.0:  type 0x0000c003 (v 0.0.1.0)
[ 1502.624811] mqnic 0000:81:00.0: IF features: 0x00000711
[ 1502.624812] mqnic 0000:81:00.0: Max TX MTU: 9214
[ 1502.624813] mqnic 0000:81:00.0: Max RX MTU: 9214
[ 1502.624816] mqnic 0000:81:00.0: Event queue offset: 0x00100000
[ 1502.624817] mqnic 0000:81:00.0: Event queue count: 32
[ 1502.624818] mqnic 0000:81:00.0: Event queue stride: 0x00000020
[ 1502.624822] mqnic 0000:81:00.0: TX queue offset: 0x00200000
[ 1502.624823] mqnic 0000:81:00.0: TX queue count: 8192
[ 1502.624824] mqnic 0000:81:00.0: TX queue stride: 0x00000020
[ 1502.624827] mqnic 0000:81:00.0: TX completion queue offset: 0x00400000
[ 1502.624828] mqnic 0000:81:00.0: TX completion queue count: 8192
[ 1502.624829] mqnic 0000:81:00.0: TX completion queue stride: 0x00000020
[ 1502.624832] mqnic 0000:81:00.0: RX queue offset: 0x00600000
[ 1502.624833] mqnic 0000:81:00.0: RX queue count: 256
[ 1502.624834] mqnic 0000:81:00.0: RX queue stride: 0x00000020
[ 1502.624838] mqnic 0000:81:00.0: RX completion queue offset: 0x00700000
[ 1502.624838] mqnic 0000:81:00.0: RX completion queue count: 256
[ 1502.624839] mqnic 0000:81:00.0: RX completion queue stride: 0x00000020
[ 1502.624841] mqnic 0000:81:00.0: Max desc block size: 8
[ 1502.632850] mqnic 0000:81:00.0: Port-level register blocks:
[ 1502.632855] mqnic 0000:81:00.0:  type 0x0000c040 (v 0.0.1.0)
[ 1502.632860] mqnic 0000:81:00.0: Scheduler type: 0x0000c040
[ 1502.632861] mqnic 0000:81:00.0: Scheduler offset: 0x00800000
[ 1502.632862] mqnic 0000:81:00.0: Scheduler channel count: 8192
[ 1502.632863] mqnic 0000:81:00.0: Scheduler channel stride: 0x00000004
[ 1502.632864] mqnic 0000:81:00.0: Scheduler count: 1
[ 1502.632866] mqnic 0000:81:00.0: Port count: 1
[ 1503.217179] mqnic 0000:81:00.0: Registered device mqnic0
```

The driver will attempt to read MAC addresses from the card. If it fails, it will fall back on random MAC addresses. On some cards, the MAC addresses are fixed and cannot be changed, on other cards they are written to use-accessible EEPROM and as such can be changed. Some cards with EEPROM come with blank EEPROMs, so if you want a persistent MAC address, you'll have to write a base MAC address into the EEPROM. And finally, some cards do not have an EEPROM for storing MAC addresses, and persistent MAC addresses are not currently supported on these cards.

2.12 Testing the design

To test the design, connect it to another NIC, either directly with a DAC cable or similar, or via a switch.

Before performing any testing, an IP address must be assigned through the Linux kernel. There are various ways to do this, depending on the distribution in question. For example, using `iproute2`:

```
$ sudo ip link set dev enp129s0 up
$ sudo ip addr add 10.0.0.2/24 dev enp129s0
```

You can also change the MTU setting:

```
$ sudo ip link set mtu 9000 dev enp129s0
```

Note that NetworkManager can fight over the network interface configuration (depending on the linux distribution). If the IP address disappears from the interface, then this is likely the fault of NetworkManager as it attempts to dynamically configure the interface. One solution for this is simply to use NetworkManager to configure the interface instead of iproute2. Another is to statically configure the interface using configuration files in `/etc/network/interfaces` so that NetworkManager will leave it alone.

One the card is configured, using `ping` is a good first test:

```
$ ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.221 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.109 ms
^C
--- 10.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1052ms
rtt min/avg/max/mdev = 0.109/0.165/0.221/0.056 ms
```

If `ping` works, then try `iperf`.

On the server:

```
$ iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 10.0.0.2, port 54316
[  5] local 10.0.0.1 port 5201 connected to 10.0.0.2 port 54318
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  2.74 GBytes  23.6 Gbits/sec
[  5]   1.00-2.00   sec  2.85 GBytes  24.5 Gbits/sec
[  5]   2.00-3.00   sec  2.82 GBytes  24.2 Gbits/sec
[  5]   3.00-4.00   sec  2.83 GBytes  24.3 Gbits/sec
[  5]   4.00-5.00   sec  2.82 GBytes  24.2 Gbits/sec
[  5]   5.00-6.00   sec  2.76 GBytes  23.7 Gbits/sec
[  5]   6.00-7.00   sec  2.63 GBytes  22.6 Gbits/sec
[  5]   7.00-8.00   sec  2.81 GBytes  24.2 Gbits/sec
[  5]   8.00-9.00   sec  2.73 GBytes  23.5 Gbits/sec
[  5]   9.00-10.00  sec  2.73 GBytes  23.4 Gbits/sec
[  5]  10.00-10.00  sec   384 KBytes  7.45 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
```

(continues on next page)

(continued from previous page)

```
[  5]   0.00-10.00  sec  27.7 GBytes  23.8 Gbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

On the client:

```
$ iperf3 -c 10.0.0.1
Connecting to host 10.0.0.1, port 5201
[  5] local 10.0.0.2 port 54318 connected to 10.0.0.1 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  2.74 GBytes  23.6 Gbits/sec    0   2.18 MBytes
[  5]   1.00-2.00   sec  2.85 GBytes  24.5 Gbits/sec    0   2.18 MBytes
[  5]   2.00-3.00   sec  2.82 GBytes  24.2 Gbits/sec    0   2.29 MBytes
[  5]   3.00-4.00   sec  2.83 GBytes  24.3 Gbits/sec    0   2.40 MBytes
[  5]   4.00-5.00   sec  2.82 GBytes  24.2 Gbits/sec    0   2.40 MBytes
[  5]   5.00-6.00   sec  2.76 GBytes  23.7 Gbits/sec    0   2.65 MBytes
[  5]   6.00-7.00   sec  2.63 GBytes  22.6 Gbits/sec    0   2.65 MBytes
[  5]   7.00-8.00   sec  2.81 GBytes  24.2 Gbits/sec    0   2.65 MBytes
[  5]   8.00-9.00   sec  2.73 GBytes  23.5 Gbits/sec    0   2.65 MBytes
[  5]   9.00-10.00  sec  2.73 GBytes  23.4 Gbits/sec    0   2.65 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  27.7 GBytes  23.8 Gbits/sec    0             sender
[  5]   0.00-10.00  sec  27.7 GBytes  23.8 Gbits/sec                  receiver

iperf Done.
```

Finally, test the PTP synchronization performance with `ptp4l` from `linuxptp`.

On the server:

```
$ sudo ptp4l -i enp193s0np0 --masterOnly=1 -m --logSyncInterval=-3
ptp4l[4463.798]: selected /dev/ptp2 as PTP clock
ptp4l[4463.799]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[4463.799]: port 0: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[4471.745]: port 1: LISTENING to MASTER on ANNOUNCE_RECEIPT_TIMEOUT_EXPIRES
ptp4l[4471.746]: selected local clock ec0d9a.fffe.6821d4 as best master
ptp4l[4471.746]: port 1: assuming the grand master role
```

On the client:

```
$ sudo ptp4l -i enp129s0 --slaveOnly=1 -m
ptp4l[642.961]: selected /dev/ptp5 as PTP clock
ptp4l[642.962]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[642.962]: port 0: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[643.477]: port 1: new foreign master ec0d9a.fffe.6821d4-1
ptp4l[647.478]: selected best master clock ec0d9a.fffe.6821d4
ptp4l[647.478]: port 1: LISTENING to UNCALIBRATED on RS_SLAVE
ptp4l[648.233]: port 1: UNCALIBRATED to SLAVE on MASTER_CLOCK_SELECTED
ptp4l[648.859]: rms 973559315 max 1947121298 freq -41295 +/- 15728 delay   643 +/-   0
ptp4l[649.860]: rms  698 max 1236 freq -44457 +/- 949 delay   398 +/-   0
ptp4l[650.861]: rms 1283 max 1504 freq -42099 +/- 257 delay   168 +/-   0
```

(continues on next page)

(continued from previous page)

```
ptp4l[651.862]: rms  612 max  874 freq -42059 +/-  85 delay   189 +/-   1
ptp4l[652.863]: rms  127 max  245 freq -42403 +/-  85
ptp4l[653.865]: rms   58 max   81 freq -42612 +/-  36 delay   188 +/-   0
ptp4l[654.866]: rms   21 max   36 freq -42603 +/-  12 delay   181 +/-   0
ptp4l[655.867]: rms    6 max   12 freq -42584 +/-   7 delay   174 +/-   1
ptp4l[656.868]: rms   14 max   26 freq -42606 +/-  12
ptp4l[657.869]: rms   19 max   23 freq -42631 +/-  11 delay   173 +/-   0
ptp4l[658.870]: rms   24 max   35 freq -42660 +/-  12 delay   173 +/-   0
ptp4l[659.870]: rms   23 max   35 freq -42679 +/-  16 delay   173 +/-   0
ptp4l[660.872]: rms   18 max   20 freq -42696 +/-   5 delay   170 +/-   0
ptp4l[661.873]: rms   18 max   30 freq -42714 +/-   8 delay   167 +/-   1
ptp4l[662.874]: rms   26 max   36 freq -42747 +/-  10 delay   168 +/-   0
ptp4l[663.875]: rms   18 max   21 freq -42757 +/-  10 delay   167 +/-   0
ptp4l[664.876]: rms   14 max   17 freq -42767 +/-   8 delay   167 +/-   1
ptp4l[665.877]: rms    9 max   12 freq -42741 +/-   7 delay   168 +/-   2
```

In this case, `ptp4l` has converged to an offset of well under 100 ns, reporting a frequency difference of about -43 ppm.

While `ptp4l` is syncing the clock, the kernel module will print some debug information:

```
[  642.943481] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 0
[  642.943487] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x400000000
[  647.860040] mqnic 0000:81:00.0: mqnic_start_xmit TX TS requested
[  647.860084] mqnic 0000:81:00.0: mqnic_process_tx_cq TX TS requested
[  648.090566] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 2795012
[  648.090572] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000b2e18
[  648.090575] mqnic 0000:81:00.0: mqnic_phc_adjtime delta: -1947115961
[  648.215705] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 3241067
[  648.215711] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000cf6da
[  648.340845] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 3199401
[  648.340851] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000ccc30
[  648.465995] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 3161092
[  648.466001] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000ca4f5
[  648.591129] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 3121946
[  648.591135] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000c7cdf
[  648.716275] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 3082853
[  648.716281] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000c54d7
[  648.841425] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 3048881
[  648.841431] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000c320e
[  648.966550] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 3012985
[  648.966556] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000c0d4c
[  649.091601] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 2980479
[  649.091607] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000bec03
[  649.216740] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 2950457
[  649.216746] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000bcd45
[  649.341844] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 2922995
[  649.341850] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000bb126
[  649.466966] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 2897658
[  649.466972] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000b9734
[  649.592007] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 2875145
[  649.592013] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000b8026
[  649.717159] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 2854962
[  649.717165] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000b6b7b
```

(continues on next page)

(continued from previous page)

```
[  649.776717] mqnic 0000:81:00.0: mqnic_start_xmit TX TS requested
[  649.776761] mqnic 0000:81:00.0: mqnic_process_tx_cq TX TS requested
[  649.842186] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 2813737
[  649.842191] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000b4144
[  649.967434] mqnic 0000:81:00.0: mqnic_phc_adjfine scaled_ppm: 2800052
[  649.967440] mqnic 0000:81:00.0: mqnic_phc_adjfine adj: 0x4000b3341
```

In this case, the core clock frequency is slightly less than 250 MHz. You can compute the clock frequency in GHz like so:

```
>>> 2**32/0x4000b3341
0.24998931910318553
```

CHAPTER

THREE

DEBUGGING

3.1 The server rebooted when configuring the FPGA

This is a common problem caused by the server management subsystem (IPMI, iLO, iDRAC, or whatever your server manufacturer calls it). It detects the PCIe device falling off the bus when the FPGA is reset, and does something in response. In some machines it just complains bitterly by logging an event and turning on an angry red LED. In other machines, it does that, and then immediately reboots the machine. Highly annoying. However, there is a simple solution for this that involves some poking around in PCIe configuration registers to disable PCIe fatal error reporting on the port that the device is connected to.

Run the `pcie_disable_fatal_err.sh` script before you configure the FPGA. Specify the PCIe device ID of the FPGA board as the argument (`xy:00.0`, as shown in `lspci`). You'll have to do a warm reboot after loading the configuration if the PCIe BAR configuration has changed. This will most likely be the case when going from a stock flash image that brings up the PCIe link to Corundum, but not from Corundum to Corundum unless something was changed in the PCIe IP core configuration. If the BAR configuration has not changed, then using the `pcie_hot_reset.sh` script to perform a hot reset of the device may be sufficient. The firmware update utility `mqnic-fw` includes the same functionality as `pcie_disable_fatal_err.sh` and `pcie_hot_reset.sh`, so if the card is running a Corundum design, then you can use `mqnic-fw -t` to disable fatal error reporting and reset the card before connecting to it via JTAG.

3.2 The link is down

Things to check, in no particular order:

- Try a hot reset of the card
- Try an unmodified "known good" design for your board, either corundum or verilog-ethernet
- Try using a direct attach copper cable to rule out issues with optical transceivers.
- Try a different link partner—try a NIC instead of a switch, or a different model NIC.
- txdisable/lpmode/reset pins—only applies if you're using optical transceivers or active optical cables. If these pins are pulled the wrong way, the lasers in the transceiver will not turn on, and the link will not come up. (No, I have never wasted an hour waiting for Vivado to do its thing after pulling lpmode the wrong way…on several different boards…)
- Optical module CDR settings—if you're trying to run a 25G or 100G optical transceiver or active optical cable at 10G or 40G, you may need to disable the module CDRs via I2C by writing 0x00 to MSA register 98 on I2C address 0x50 (CDR control). (No, I have not wasted several days trying to figure out why the electrical loopback works fine at 10G, but the optical transceiver only works at 25G…)

- Check settings at link partner—some devices are better about figuring out the proper configuration than others and need to have the correct settings applied manually (e.g. Mellanox NICs are quite good, but most packet switches can be rather bad about this and may only look at the line rate reported in the transceiver EEPROM instead of what's actually going on on the link). Also check to make sure the link partner doesn't have some sort of disagreement with the cable/transceiver - some devices, usually switches, are very picky about what the EEPROM says about who manufactured the cable.
- Check FEC settings—in general, 100G devices seem to require the use of RS-FEC, 10G and 25G usually run fine without FEC, but it may need to be manually disabled on the link partner.
- Serdes configuration—ensure the correct line rate, gearbox settings, etc. are correct. Some boards also have p/n swapped (e.g. ExaNIC X10/X25), so check tx/rx invert settings. (Yes, I managed to figure *that* one out after some head-scratching despite not having access to the schematic)
- Serdes site locations—make sure you're using the correct pins.
- Serdes reference clock configuration—make sure the reference clock matches the serdes configuration, and on some boards the reference clock needs to be configured in some way before use.

3.3 Ping and iperf don't work

Things to check, in no particular order:

- Check that the interface is up (`ip link set dev <interface> up`)
- Check that the interface has an IP address assigned (`ip addr`, `ip -c a`, `ip -c -br`, to check, `ip addr add 192.168.1.1/24 dev <interface>` to set)
- The corundum driver does not currently report the link status to the OS, so check for a link light (not all design variants implement this) and check the link partner for the link status (`ip link`, NO-CARRIER means the link is down at the PHY layer)
- Try hot resetting the card with the link partner connected (clear up possible RX DFE problem)
- Check tcpdump for inbound traffic on both ends of the link `tcpdump -i <interface> -Q in` to see what is actually traversing the link. If the TX direction works but the RX direction does not, there is a high probability it is a transceiver DFE issue that may be fixable with a hot reset.
- Check with `mqnic-dump` to see if there is anything stuck in transmit queues, transmit or receive completion queues, or event queues.

3.4 The device loses its IP address

This is not a corundum issue, this is NetworkManager or a similar application causing trouble by attempting to run DHCP or similar on the interface. There are basically four options here: disable NetworkManager, configure NetworkManager to ignore the interface, use NetworkManager to configure the interface and assign the IP address you want, or use network namespaces to isolate the interface from NetworkManager. Unfortunately, if you have a board that doesn't support persistent MAC addresses, it may not be possible to configure NetworkManager to deal with the interface correctly.

CHAPTER

FOUR

PERFORMANCE TUNING

Here are some tips and tricks to get the best possible performance with Corundum.

First, it's always a good idea to test with a commercial 100G NIC as a sanity check - if a commercial 100G NIC doesn't run near 100G line rate, then Corundum will definitely have issues.

Second, check the PCIe configuration with `lspci`. You'll want to make sure that the card is actually running with the full PCIe bandwidth. Some x16 PCIe slots don't have all of the lanes physically wired, and in many cases lanes can be switched between slots depending on which slots are used—for example, two slots may share 16 lanes, if the second slot is empty, the first slot will use all 16 lanes, but if the second slot has a card installed, both slots will run with 8 lanes.

This is what `lspci` reports for Corundum in a gen 3 x16 configuration in a machine with an AMD EPYC 7302P CPU:

```
$ sudo lspci -d 1234:1001 -vvv
81:00.0 Ethernet controller: Device 1234:1001
    Subsystem: Silicom Denmark Device a00e
    Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR-␣
↪FastB2B- DisINTx+
    Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >
↪SERR- <PERR- INTx-
    Latency: 0, Cache Line Size: 64 bytes
    Interrupt: pin ? routed to IRQ 337
    NUMA node: 1
    IOMMU group: 13
    Region 0: Memory at 20020000000 (64-bit, prefetchable) [size=16M]
    Capabilities: [40] Power Management version 3
        Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA PME(D0-,D1-,D2-,D3hot-,D3cold-)
        Status: D0 NoSoftRst+ PME-Enable- DSel=0 DScale=0 PME-
    Capabilities: [48] MSI: Enable+ Count=32/32 Maskable+ 64bit+
        Address: 00000000fee00000  Data: 0000
        Masking: 00000000  Pending: 00000000
    Capabilities: [70] Express (v2) Endpoint, MSI 00
        DevCap: MaxPayload 1024 bytes, PhantFunc 0, Latency L0s <64ns, L1 <1us
            ExtTag+ AttnBtn- AttnInd- PwrInd- RBE+ FLReset- SlotPowerLimit 75.000W
        DevCtl: CorrErr+ NonFatalErr+ FatalErr+ UnsupReq-
            RlxdOrd+ ExtTag+ PhantFunc- AuxPwr- NoSnoop+
            MaxPayload 512 bytes, MaxReadReq 512 bytes
        DevSta: CorrErr+ NonFatalErr- FatalErr- UnsupReq+ AuxPwr- TransPend-
        LnkCap: Port #0, Speed 8GT/s, Width x16, ASPM not supported
            ClockPM- Surprise- LLActRep- BwNot- ASPMOptComp+
        LnkCtl: ASPM Disabled; RCB 64 bytes, Disabled- CommClk+
            ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
```

(continues on next page)

(continued from previous page)

```
        LnkSta: Speed 8GT/s (ok), Width x16 (ok)
            TrErr- Train- SlotClk+ DLActive- BWMgmt- ABWMgmt-
        DevCap2: Completion Timeout: Range BC, TimeoutDis+ NROPrPrP- LTR-
             10BitTagComp- 10BitTagReq- OBFF Not Supported, ExtFmt- EETLPPrefix-
             EmergencyPowerReduction Not Supported, EmergencyPowerReductionInit-
             FRS- TPHComp- ExtTPHComp-
             AtomicOpsCap: 32bit- 64bit- 128bitCAS-
        DevCtl2: Completion Timeout: 50us to 50ms, TimeoutDis- LTR- OBFF Disabled,
             AtomicOpsCtl: ReqEn-
        LnkCap2: Supported Link Speeds: 2.5-8GT/s, Crosslink- Retimer- 2Retimers- DRS-
        LnkCtl2: Target Link Speed: 8GT/s, EnterCompliance- SpeedDis-
             Transmit Margin: Normal Operating Range, EnterModifiedCompliance-␣
↪ComplianceSOS-
             Compliance De-emphasis: -6dB
        LnkSta2: Current De-emphasis Level: -6dB, EqualizationComplete+␣
↪EqualizationPhase1+
             EqualizationPhase2+ EqualizationPhase3+ LinkEqualizationRequest-
             Retimer- 2Retimers- CrosslinkRes: unsupported
    Capabilities: [100 v1] Advanced Error Reporting
        UESta:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC-␣
↪UnsupReq- ACSViol-
        UEMsk:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC-␣
↪UnsupReq- ACSViol-
        UESvrt: DLP+ SDES+ TLP- FCP+ CmpltTO- CmpltAbrt- UnxCmplt- RxOF+ MalfTLP+ ECRC-␣
↪UnsupReq- ACSViol-
        CESta:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- AdvNonFatalErr+
        CEMsk:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- AdvNonFatalErr+
        AERCap: First Error Pointer: 00, ECRCGenCap- ECRCGenEn- ECRCChkCap- ECRCChkEn-
            MultHdrRecCap- MultHdrRecEn- TLPPfxPres- HdrLogCap-
        HeaderLog: 00000000 00000000 00000000 00000000
    Capabilities: [1c0 v1] Secondary PCI Express
        LnkCtl3: LnkEquIntrruptEn- PerformEqu-
        LaneErrStat: 0
    Kernel driver in use: mqnic
```

The device driver also prints out some PCIe-related information when it attaches to the device, to save the trouble of running `lspci`:

```
[  349.460705] mqnic 0000:81:00.0: mqnic PCI probe
[  349.460712] mqnic 0000:81:00.0:  Vendor: 0x1234
[  349.460715] mqnic 0000:81:00.0:  Device: 0x1001
[  349.460717] mqnic 0000:81:00.0:  Subsystem vendor: 0x1c2c
[  349.460719] mqnic 0000:81:00.0:  Subsystem device: 0xa00e
[  349.460721] mqnic 0000:81:00.0:  Class: 0x020000
[  349.460723] mqnic 0000:81:00.0:  PCI ID: 0000:81:00.0
[  349.460730] mqnic 0000:81:00.0:  Max payload size: 512 bytes
[  349.460733] mqnic 0000:81:00.0:  Max read request size: 512 bytes
[  349.460735] mqnic 0000:81:00.0:  Link capability: gen 3 x16
[  349.460737] mqnic 0000:81:00.0:  Link status: gen 3 x16
[  349.460739] mqnic 0000:81:00.0:  Relaxed ordering: enabled
[  349.460740] mqnic 0000:81:00.0:  Phantom functions: disabled
[  349.460742] mqnic 0000:81:00.0:  Extended tags: enabled
```

(continues on next page)

(continued from previous page)

```
[  349.460744] mqnic 0000:81:00.0:  No snoop: enabled
[  349.460745] mqnic 0000:81:00.0:  NUMA node: 1
[  349.460753] mqnic 0000:81:00.0: 126.016 Gb/s available PCIe bandwidth (8.0 GT/s PCIe␣
↪x16 link)
[  349.460767] mqnic 0000:81:00.0: enabling device (0000 -> 0002)
[  349.460802] mqnic 0000:81:00.0: Control BAR size: 16777216
[  349.462723] mqnic 0000:81:00.0: Configured 32 IRQs
```

Note that `lspci` reports `LnkSta: Speed 8GT/s (ok), Width x16 (ok)`, indicating that the link is running at the max supported speed and max supported link width. If one of those is reported as `(degraded)`, then further investigation is required. If `(ok)` or `(degraded)` is not shown, then compare `LnkSta` with `LnkCap` to see if `LnkSta` reports lower values. In this case, `lspci` reports `LnkCap: Port #0, Speed 8GT/s, Width x16`, which matches `LnkSta`. It also reports `MSI: Enable+ Count=32/32`, indicating that all 32 MSI channels are active. Some motherboards do not fully implement MSI and limit devices to a single channel. Eventually, Corundum will migrate to MSI-X to mitigate this issue, as well as support more interrupt channels. Also note that `lspci` reports `MaxPayload 512 bytes`—this is the largest that I have seen so far (on AMD EPYC), most modern systems report 256 bytes. Obviously, the larger, the better in terms of PCIe overhead.

Non-uniform memory access (NUMA) is another potential pitfall to be aware of. Systems with multiple CPU sockets will generally have at least one NUMA node associated with each socket, and some CPUs, like AMD EPYC, have internal NUMA nodes even with a single CPU. For best performance, any processes that access the NIC should be pinned to the NIC's local NUMA node. If packets are stored in memory located on a different NUMA node, then there will be a performance penalty associated with the NIC accessing that memory via QPI, UPI, etc. Use `numactl -s` to get a list of all physical CPUs and NUMA nodes on the system. If only one node is listed, then no binding is required. If you're running a CPU with internal NUMA nodes such as AMD EPYC, make sure that BIOS is set up to expose the internal NUMA nodes. The NUMA node associated with the network interface is shown both in the `lspci` and driver output output (`NUMA node: 3`), and it can also be read from sysfs (`/sys/class/net/<dev>/device/numa_node`). Use `numactl -l -N <node> <command>` to run programs on a specified NUMA node, for example, `numactl -l -N 3 iperf3 -s`. Recent versions of `numactl` also support automatically determining the NUMA node from the network device name, so in this case `numactl -l -N netdev:enp129s0 iperf3 -s` would run `iperf` on the NUMA node that `enp129s0` is associated with. It's important to make sure that both the client and the server are run on the correct NUMA node, so it's probably a better idea to manually run `iperf3 -s` under `numactl` than to run `iperf3` as a system service that could potentially run on any NUMA node. On Intel CPUs, PCM[20] can be used to monitor QPI/UPI traffic to confirm that processes are bound to the correct NUMA nodes.

It's also advisable to go into BIOS setup and disable any power-management features to get the system into its highest-performance state.

Notes on the performance evaluation for the FCCM paper: the servers used are Dell R540 machines with dual Intel Xeon 6138 CPUs and all memory channels populated, and `lspci` reports `MaxPayload 256 bytes`. The machines have two NUMA nodes, so only one CPU is used for performance evaluation to prevent traffic from traversing the UPI link. On these machines, a single `iperf` process would run at 20-30 Gbps with 1500 byte MTU, or 40-50 Gbps with 9000 byte MTU. The Corundum design for those tests was configured with 8192 TX queues and 256 RX queues.

[20] https://github.com/opcm/pcm

CHAPTER

FIVE

PORTING

This guide is a high-level overview for how to port Corundum to new hardware. In general, this guide only applies to FPGA families that are already supported by Corundum, new FPGA families can require significant interfacing changes, especially for the PCI express interface as this can vary significantly between different FPGA families.

The only interfaces that the Corundum datapath requires are the PCI express interface and the Ethernet interfaces. Ancillary features such as firmware updates, persistent MAC addresses, and optical module communication are optional—the core datapath will still function if these features are not implemented. In general the PCI express and Ethernet interfaces are dependent almost completely on the FPGA family, while ancillary features tend to be much more board-dependent.

5.1 Preparation

Before porting Corundum to a new board, it is recommended to create example designs for both verilog-ethernet and verilog-pcie for the target board. The verilog-ethernet design will bring up the Ethernet interfaces at 10 Gbps and ensures the transceivers, reference clocks, and module control pins are properly configured for the Ethernet interfaces to operate. Some boards may require additional code to configure clocking logic to supply the proper reference clocks to the transceivers on the FPGA, which will generally be one of 156.25 MHz, 161.1328125 MHz, 322.265625 MHz, or 644.53125 MHz. The verilog-pcie design brings up the PCI express interface, validating that all of the pin assignments and transceiver site locations are correct. Once both of these designs are working, then porting corundum is straightforward.

5.2 Porting Corundum

Start by making a copy of a Corundum design that targets a similar board. Priority goes to a chip in the same family, then similar ancillary interfaces.

5.2.1 Board ID

Each board should have a unique board ID specified in `mqnic_hw.h`. These IDs are used by the driver for any board-specific initialization and interfacing. These IDs are arbitrary, but making something relatively predictable is a good idea to reduce the possibility of collisions. Most of the current IDs are a combination of the PCIe vendor ID of the board manufacturer, combined with a board-specific portion. For example, the board IDs for ExaNICs are simply the original ExaNIC PCIe VID and PID, and the Xilinx board IDs are a combination of the Xilinx PCIe VID, the part series (7 for 7 series, 8 for UltraScale, 9 for UltraScale+, etc.) and the hex version of the board part number (VCU108 = 6c, VCU1525 = 5f5, etc.). Pick a board ID, add it to `mqnic_hw.h`, and set the `BOARD_ID` parameter in fpga_core.v.

5.2.2 FPGA ID

The FPGA ID is used by the firmware update tool as a simple sanity check to prevent firmware for a different board from being loaded accidentally. Set the FPGA_ID parameter in fpga_core.v to the JTAG ID of the FPGA on the board. The IDs are located in `fpga_id.h`/`fpga_id.c`. If you do not want to implement the firmware update feature, `FPGA_ID` can be set to 0.

5.2.3 PCIe interface

Ensure that the PCIe hard IP core settings are correct for the target board. In many cases, the default settings are correct, but in some cases the transceiver sites need to be changed. Edit the TCL file appropriately, or generate the IP in vivado and extract the TCL commands from the Vivado journal file. If you previously ported the verilog-pcie design, then the settings can be copied over, with the PCIe IDs, BARs, and MSI settings configured appropriately.

Check that the `BAR0_APERTURE` setting and PCIE_AXIS settings in `fpga.v` and `fpga_core.v` match the PCIe core configuration.

5.2.4 Ethernet interfaces

For 100G interfaces, use Xilinx CMAC instances. A free license can be generated on the Xilinx website. The cores must be configured for CAUI-4. Select the appropriate reference clock and transceiver sites for the interfaces on the board. It may be necessary to adjust the CMAC site selections depending on which transceiver sites are used. Implement the design, open the implemented design, check the relative positions of the transceiver sites and CMAC sites, and adjust as appropriate. You can actually look at any synthesized or implemented design for the same chip to look at the relative positions of the sites.

For 10G or 25G interfaces, you can either use the MAC modules from verilog-ethernet or Xilinx-provided MAC modules. For the included MACs, the main thing to adjust is the gtwizard instance. This needs to be set up to use the correct transceiver sites and reference clock inputs. The internal interface must be the 64 bit asynchronous gearbox. Check the connection ordering; the gtwizard instance is always in the order of the site names, but this may not match the board, and connections may need to be re-ordered to match. In particular, double check that the RX clocks are connected correctly.

Update the interfaces between `fpga.v` and `fpga_core.v` to match the module configuration. Update the code in `fpga_core.v` to connect the PHYs in `fpga.v` to the appropriate MACs in `fpga_core.v`. Also set `IF_COUNT` and `PORTS_PER_IF` appropriately in `fpga_core.v`.

5.2.5 I2C interfaces

MAC address EEPROMs and optical modules are accessed via I2C. This is highly board-dependent. On some boards, there is a single I2C interface and a number of I2C multiplexers to connect everything. On other boards, each optical module has a dedicated I2C interface. On other boards, the I2C bus sits behind a board management controller. The core datapath will work fine without setting up I2C, but having the I2C buses operational can be a useful debugging feature. If I2C access is not required, simply do not implement the registers and ensure that the selected board ID does not correspond to any I2C init code in `mqnic_i2c.c`.

All corundum designs that directly connect I2C interfaces to the FPGA pins make use of bit-bang I2C support in the Linux kernel. There are a set of registers set aside for controlling up to four I2C buses in `mqnic_hw.h`. These should be appropriately implemented in the NIC CSR register space in `fpga_core.v`. Driver code also needs to be added to `mqnic_i2c.c` to initialize everything appropriately based on the board ID.

5.2.6 Flash access

Firmware updates require access to the FPGA configuration flash. Depending on the flash type, this either requires connections to dedicated pins via specific device primitives, normal FPGA IO pins, or both. The flash interface is a very simple bit-bang interface that simply exposes these pins over PCIe via NIC CSR register space. The register definitions are in `mqnic_hw.h`. Take a look at existing designs that implement QSPI or BPI flash and implement the same register configuration in `fpga_core.v`. If firmware update support is not required, simply do not implement the flash register block.

5.2.7 Module control pins

Optical modules have several low-speed control pins in addition to the I2C interface. For DAC cables, these pins have no effect, but for AOC cables or optical modules, these pins are very important. Specifically, SFP+ and SFP28 modules need to have the correct level on `tx_disable` in order to turn the laser on. Similarly, QSFP+ and QSFP28 modules need to have the `reset` and `lpmode` pins set correctly. These pins can be statically tied off with the modules enabled, or they can be exposed to the driver via standard registers specified in `mqnic_hw.h` and implemented in the NIC CSRs in `fpga_core.v`.

CHAPTER

SIX

PERSISTENT MAC ADDRESSES

When registering network interfaces with the operating system, the driver must provide a MAC address for each interface. Ensuring that the configured MAC addresses are unique and consistent across driver reloads requires binding the addresses to the hardware in some way, usually through the use of some form of nonvolatile memory. It is relatively common for FPGA boards to provide small I2C EEPROMs for storing this sort of information. On other boards, the MAC address can be read out from the board management controller. If the driver fails to read a valid MAC address, it will fall back to using a randomly-generated MAC address. See the *Device list* (page 131) for a summary of how persistent MAC addresses are implemented on each board. Boards that have pre-programmed MAC addresses should work "out of the box". However, boards that include blank EEPROMs need to have a MAC address written into the EEPROM for this functionality to work.

6.1 Programming I2C EEPROM via kernel module

The driver registers all on-card I2C devices via the Linux I2C subsystem. Therefore, the MAC address EEPROM appears in sysfs, and a MAC address can easily be written using `dd`. Note that accessing the EEPROM is a little bit different on each board.

After loading the driver, the device can be accessed either directly (`/sys/bus/pci/devices/0000:xx:00.0/`) or from the corresponding network interface (`/sys/class/net/eth0/device/`) or miscdev (`/sys/class/misc/mqnic0/device/`). See the table below for the sysfs paths for each board. Note that the I2C bus numbers will vary. Also note that optical module I2C interfaces are registered as EEPROMs with I2C address 0x50, so ensure you have the correct EEPROM by dumping the contents with `xxd` or a hex editor before programming it.

After determining the sysfs path and picking a MAC address, run a command similar to this one to program the MAC address into the EEPROM:

```
echo 02 aa bb 00 00 00 | xxd -r -p - | dd bs=1 count=6 of=/sys/class/net/eth0/device/i2c-
↪4/4-0074/channel-2/7-0054/eeprom
```

After reloading the driver, the interfaces should use the new MAC address:

```
14: enp1s0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group␣
↪default qlen 1000
    link/ether 02:aa:bb:00:00:00 brd ff:ff:ff:ff:ff:ff
15: enp1s0d1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group␣
↪default qlen 1000
    link/ether 02:aa:bb:00:00:01 brd ff:ff:ff:ff:ff:ff
```

Manufacturer	Board	sysfs path [1]
Alpha Data	ADM-PCIE-9V3	`i2c-X/X-0050/eeprom` [3]
Exablaze	ExaNIC X10 [2]	`i2c-X/X-0050/eeprom` [3]
Exablaze	ExaNIC X25 [2]	`i2c-X/X-0050/eeprom` [3]
Xilinx	VCU108	`i2c-X/X-0075/channel-3/Y-0054/eeprom`
Xilinx	VCU118	`i2c-X/X-0075/channel-3/Y-0054/eeprom`
Xilinx	VCU1525	`i2c-X/X-0074/channel-2/Y-0054/eeprom`
Xilinx	ZCU106	`i2c-X/X-0074/channel-0/Y-0054/eeprom`

Notes: * [1] X and Y are i2c bus numbers that will vary * [2] Card should come pre-programmed with a base MAC address * [3] Optical module I2C interfaces may appear exactly the same way; confirm correct EEPROM by reading the contents with `xxd` or a hex editor.

CHAPTER

SEVEN

OPERATIONS

This is a list of all of the operations involved in sending and receiving packets, across both hardware and software.

7.1 Packet transmission

1. linux: The linux kernel calls `mqnic_start_xmit()` (via `ndo_start_xmit()`) with an `sk_buff` for transmission
2. `mqnic_start_xmit()` (`mqnic_tx.c`): The driver determines the destination transmit queue with `skb_get_queue_mapping`
3. `mqnic_start_xmit()` (`mqnic_tx.c`): The driver marks the `sk_buff` for timestamping, if requested
4. `mqnic_start_xmit()` (`mqnic_tx.c`): The driver generates the hardware IP checksum command and writes it into the descriptor
5. `mqnic_map_skb()` (`mqnic_tx.c`): The driver writes a reference to the `sk_buff` into `ring->tx_info`
6. `mqnic_map_skb()` (`mqnic_tx.c`): The driver generates DMA mappings for the `sk_buff` (`skb_frag_dma_map()`/`dma_map_single()`) and builds the descriptor
7. `mqnic_start_xmit()` (`mqnic_tx.c`): The driver enqueues the packet by incrementing its local copy of the producer pointer
8. `mqnic_start_xmit()` (`mqnic_tx.c`): At the end of a batch of packets, the driver writes the updated producer pointer to the NIC via MMIO
9. *queue_manager* (page 94) `s_axil_*`: The MMIO write arrives at the queue manager via AXI lite
10. *queue_manager* (page 94) `m_axis_doorbell_*`: The queue manager updates the producer pointer and generates a doorbell event
11. *tx_scheduler_rr* (page 98) `s_axis_doorbell_*`: The doorbell event arrives at the port schedulers
12. *tx_scheduler_rr* (page 98): The scheduler marks the queue as active and schedules it if necessary
13. *tx_scheduler_rr* (page 98): The scheduler decides to send a packet
14. *tx_scheduler_rr* (page 98) `m_axis_tx_req_*`: The scheduler generates a transmit request
15. *tx_engine* (page 97) `s_axis_tx_req_*`: The transmit request arrives at the transmit engine
16. *tx_engine* (page 97) `m_axis_desc_req_*`: The transmit engine issues a descriptor request
17. *desc_fetch* (page 43) `s_axis_desc_req_*`: The descriptor request arrives at the descriptor fetch module
18. *desc_fetch* (page 43) `m_axis_desc_dequeue_req_*`: The descriptor fetch module issues a dequeue request to the queue manager

19. *queue_manager* (page 94) `s_axis_dequeue_req_*`: The dequeue request arrives at the queue manager module
20. *queue_manager* (page 94): If the queue is not empty, the queue manager starts a dequeue operation on the queue
21. *queue_manager* (page 94) `m_axis_dequeue_resp_*`: The queue manager sends a response containing the operation status and DMA address
22. *desc_fetch* (page 43) `s_axis_desc_dequeue_resp_*`: The response arrives at the descriptor fetch module
23. *desc_fetch* (page 43) `m_axis_req_status_*`: The descriptor module reports the descriptor fetch status
24. *desc_fetch* (page 43) `m_axis_dma_read_desc_*`: The descriptor module issues a DMA read request
25. `dma_if_pcie_rd` `s_axis_read_desc_*`: The requst arrives at the DMA read interface
26. `dma_if_pcie_rd`: The DMA read interface issues a PCIe read request
27. `dma_if_pcie_rd`: The read data comes back in a completion packet and is written to the descriptor fetch local DMA RAM
28. `dma_if_pcie_rd` `m_axis_read_desc_status_*`: The DMA read interface issues a status message
29. *desc_fetch* (page 43) `m_axis_desc_dequeue_commit_*`: The descriptor fetch module issues a dequeue commit message
30. *queue_manager* (page 94): The queue manager commits the dequeue operation and updates the consumer pointer
31. *desc_fetch* (page 43) `dma_read_desc_*`: The descriptor fetch module issues a read request to its internal DMA module
32. *desc_fetch* (page 43) `m_axis_desc_*`: The internal DMA module reads the descriptor and transfers it via AXI stream
33. *tx_engine* (page 97): The descriptor arrives at the transmit engine
34. *tx_engine* (page 97): The transmit engine stores the descriptor data
35. *tx_engine* (page 97) `m_axis_dma_read_desc_*`: The transmit engine issues a DMA read request
36. `dma_if_pcie_rd` `s_axis_read_desc_*`: The requst arrives at the DMA read interface
37. `dma_if_pcie_rd`: The DMA read interface issues a PCIe read request
38. `dma_if_pcie_rd`: The read data comes back in a completion packet and is written to the interface local DMA RAM
39. `dma_if_pcie_rd` `m_axis_read_desc_status_*`: The DMA read interface issues a status message
40. *tx_engine* (page 97) `m_axis_tx_desc_*`: The transmit engine issues a read request to the interface DMA engine
41. *tx_engine* (page 97) `m_axis_tx_csum_cmd_*`: The transmit engine issues a transmit checksum command
42. *mqnic_interface_tx* (page 83) `tx_axis_*`: The interface DMA module reads the packet data from interface local DMA RAM and transfers it via AXI stream
43. *mqnic_egress* (page 82): egress processing
44. *tx_checksum* (page 97): The transmit checksum module computes and inserts the checksum
45. *mqnic_app_block* (page 43) `s_axis_if_tx`: data is presented to the application section
46. *mqnic_app_block* (page 43) `m_axis_if_tx`: data is returned from the application section
47. *mqnic_core* (page 59): Data passes enters per-interface transmit FIFO module and is divided into per-port, per-traffic-class FIFOs
48. *mqnic_app_block* (page 43) `s_axis_sync_tx`: data is presented to the application section
49. *mqnic_app_block* (page 43) `m_axis_sync_tx`: data is returned from the application section

50. *mqnic_core* (page 59): Data passes through per-port transmit async FIFO module and is transferred to MAC TX clock domain
51. *mqnic_app_block* (page 43) `s_axis_direct_tx`: data is presented to the application section
52. *mqnic_app_block* (page 43) `m_axis_direct_tx`: data is returned from the application section
53. *mqnic_l2_egress* (page 84): layer 2 egress processing
54. *mqnic_core* (page 59): data leaves through transmit streaming interfaces
55. The packet arrives at the MAC
56. The MAC produces a PTP timestamp
57. *tx_engine* (page 97): The PTP timestamp arrives at the transmit engine
58. *tx_engine* (page 97) `m_axis_cpl_req_*`: The transmit engine issues a completion write request
59. *cpl_write* (page 42): The completion write module writes the completion data into its local DMA RAM
60. *cpl_write* (page 42) `m_axis_cpl_enqueue_req_*`: The completion write module issues an enqueue request to the completion queue manager
61. *cpl_queue_manager* (page 42) `m_axis_enqueue_req_*`: The enqueue request arrives at the completion queue manager module
62. *cpl_queue_manager* (page 42): If the queue is not full, the queue manager starts an enqueue operation on the queue
63. *cpl_queue_manager* (page 42) `m_axis_enqueue_resp_*`: The completion queue manager sends a response containing the operation status and DMA address
64. *cpl_write* (page 42): The response arrives at the completion write module
65. *cpl_write* (page 42) `m_axis_req_status_*`: The completion write module reports the completion write status
66. *desc_fetch* (page 43) `m_axis_dma_write_desc_*`: The completion write module issues a DMA write request
67. `dma_if_pcie_wr` `s_axis_write_desc_*`: The requst arrives at the DMA write interface
68. `dma_if_pcie_wr`: The DMA write interface reads the completion data from the completion write module local DMA RAM
69. `dma_if_pcie_wr`: The DMA write interface issues a PCIe write request
70. `dma_if_pcie_wr` `m_axis_write_desc_status_*`: The DMA write interface issues a status message
71. *cpl_write* (page 42) `m_axis_desc_enqueue_commit_*`: The completion write module issues an enqueue commit message
72. *cpl_queue_manager* (page 42): The completion queue manager commits the enqueue operation and updates the producer pointer
73. *cpl_queue_manager* (page 42) `m_axis_event_*`: The completion queue manager issues an event, if armed
74. *cpl_write* (page 42): The event arrives at the completion write module
75. *cpl_write* (page 42): The completion write module writes the event data into its local DMA RAM
76. *cpl_write* (page 42) `m_axis_cpl_enqueue_req_*`: The completion write module issues an enqueue request to the completion queue manager
77. *cpl_queue_manager* (page 42) `s_axis_enqueue_req_*`: The enqueue request arrives at the completion queue manager module
78. *cpl_queue_manager* (page 42): If the queue is not full, the queue manager starts an enqueue operation on the queue

79. *cpl_queue_manager* (page 42) `m_axis_enqueue_resp_*`: The completion queue manager sends a response containing the operation status and DMA address
80. *cpl_write* (page 42) `s_axis_cpl_enqueue_resp_*`: The response arrives at the completion write module
81. *cpl_write* (page 42) `m_axis_req_status_*`: The completion write module reports the completion write status
82. *desc_fetch* (page 43) `m_axis_dma_write_desc_*`: The completion write module issues a DMA write request
83. `dma_if_pcie_wr` `s_axis_write_desc_*`: The requst arrives at the DMA write interface
84. `dma_if_pcie_wr`: The DMA write interface reads the event data from the completion write module local DMA RAM
85. `dma_if_pcie_wr`: The DMA write interface issues a PCIe write request
86. `dma_if_pcie_wr` `m_axis_write_desc_status_*`: The DMA write interface issues a status message
87. *cpl_write* (page 42) `m_axis_desc_enqueue_commit_*`: The completion write module issues an enqueue commit message
88. *cpl_queue_manager* (page 42): The completion queue manager commits the enqueue operation and updates the producer pointer
89. *cpl_queue_manager* (page 42) `m_axis_event_*`: The completion queue manager issues an interrupt, if armed
90. linux: The linux kernel calls `mqnic_irq_handler()`
91. `mqnic_irq_handler()` (`mqnic_irq.c`): The driver calls the EQ handler via the notifier chain (`atomic_notifier_call_chain()`)
92. `mqnic_eq_int()` (`mqnic_eq.c`): The driver calls `mqnic_process_eq()`
93. `mqnic_process_eq()` (`mqnic_eq.c`): The driver processes the event queue, which calls the appropriate handler (`mqnic_tx_irq()`)
94. `mqnic_tx_irq()` (`mqnic_tx.c`): The driver enables NAPI polling on the queue (`napi_schedule_irqoff()`)
95. `mqnic_eq_int()` (`mqnic_eq.c`): The driver rearms the EQ (`mqnic_arm_eq()`)
96. NAPI: The linux kernel calls `mqnic_poll_tx_cq()`
97. `mqnic_poll_tx_cq()` (`mqnic_tx.c`): The driver calls `mqnic_process_tx_cq()`
98. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver reads the completion queue producer pointer from the NIC
99. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver reads the completion record
100. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver reads the `sk_buff` from `ring->tx_info`
101. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver completes the transmit timestamp operation
102. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver calls `mqnic_free_tx_desc()`
103. `mqnic_free_tx_desc()` (`mqnic_tx.c`): The driver unmaps the `sk_buff` (`dma_unmap_single()`/`dma_unmap_page()`)
104. `mqnic_free_tx_desc()` (`mqnic_tx.c`): The driver frees the `sk_buff` (`napi_consume_skb()`)
105. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver dequeues the completion record by incrementing the completion queue consumer pointer
106. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver writes the updated consumer pointer via MMIO
107. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver reads the queue consumer pointer from the NIC

108. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver increments the ring consumer pointer for in-order freed descriptors
109. `mqnic_process_tx_cq()` (`mqnic_tx.c`): The driver wakes the queue if it was stopped (`netif_tx_wake_queue()`)
110. `mqnic_poll_tx_cq()` (`mqnic_tx.c`): The driver disables NAPI polling, when idle (`napi_complete()`)
111. `mqnic_poll_tx_cq()` (`mqnic_tx.c`): The driver rearms the CQ (`mqnic_arm_cq()`)

7.2 Packet reception

init:

1. `mqnic_activate_rx_ring()` (`mqnic_rx.c`): The driver calls `mqnic_refill_rx_buffers()`
2. `mqnic_refill_rx_buffers()` (`mqnic_rx.c`): The driver calls `mqnic_prepare_rx_desc()` for each empty location in the ring
3. `mqnic_prepare_rx_desc()` (`mqnic_rx.c`): The driver allocates memory pages (`dev_alloc_pages()`)
4. `mqnic_prepare_rx_desc()` (`mqnic_rx.c`): The driver maps the pages (`dev_alloc_pages()`)
5. `mqnic_prepare_rx_desc()` (`mqnic_rx.c`): The driver writes a pointer to the page struct in `ring->rx_info`
6. `mqnic_prepare_rx_desc()` (`mqnic_rx.c`): The driver writes a descriptor with the DMA pointer and length
7. `mqnic_refill_rx_buffers()` (`mqnic_rx.c`): The driver enqueues the descriptor by incrementing its local copy of the producer pointer
8. `mqnic_refill_rx_buffers()` (`mqnic_rx.c`): At the end of the loop, the driver writes the updated producer pointer to the NIC via MMIO

receive:

1. A packet arrives at the MAC
2. The MAC produces a PTP timestamp
3. *mqnic_core* (page 59): data enters through receive streaming interfaces
4. *mqnic_l2_ingress* (page 85): layer 2 ingress processing
5. *mqnic_app_block* (page 43) `s_axis_direct_rx`: data is presented to the application section
6. *mqnic_app_block* (page 43) `m_axis_direct_rx`: data is returned from the application section
7. *mqnic_core* (page 59): Data passes through per-port receive async FIFO module and is transferred to core clock domain
8. *mqnic_app_block* (page 43) `s_axis_sync_rx`: data is presented to the application section
9. *mqnic_app_block* (page 43) `m_axis_sync_rx`: data is returned from the application section
10. *mqnic_core* (page 59): Data passes enters per-interface receive FIFO module and is placed into per-port FIFOs, then aggregated into a single stream
11. *mqnic_app_block* (page 43) `s_axis_if_rx`: data is presented to the application section
12. *mqnic_app_block* (page 43) `m_axis_if_rx`: data is returned from the application section
13. *mqnic_ingress* (page 83): ingress processing
14. *rx_hash* (page 97): The receive hash module computes the packet flow hash
15. *rx_checksum* (page 97): The receive checksum module computes the packet payload checksum

16. *mqnic_interface_rx* (page 83): A receive request is generated
17. *rx_engine* (page 97): The receive hash arrives at the receive engine
18. *rx_engine* (page 97): The receive checksum arrives at the receive engine
19. *rx_engine* (page 97): The receive request arrives at the receive engine
20. *rx_engine* (page 97) `m_axis_rx_desc_*`: The receive engine issues a write request to the interface DMA engine
21. *mqnic_interface_rx* (page 83) `rx_axis_*`: The interface DMA module writes the packet data from AXI stream to the interface local DMA RAM
22. *rx_engine* (page 97) `m_axis_desc_req_*`: The receive engine issues a descriptor request
23. *desc_fetch* (page 43): The descriptor request arrives at the descriptor fetch module
24. *desc_fetch* (page 43) `m_axis_desc_dequeue_req_*`: The descriptor fetch module issues a dequeue request to the queue manager
25. *queue_manager* (page 94) `s_axis_dequeue_req_*`: The dequeue request arrives at the queue manager module
26. *queue_manager* (page 94): If the queue is not empty, the queue manager starts a dequeue operation on the queue
27. *queue_manager* (page 94) `m_axis_dequeue_resp_*`: The queue manager sends a response containing the operation status and DMA address
28. *desc_fetch* (page 43) `m_axis_desc_dequeue_resp_*`: The response arrives at the descriptor fetch module
29. *desc_fetch* (page 43) `m_axis_req_status_*`: The descriptor module reports the descriptor fetch status
30. *desc_fetch* (page 43) `m_axis_dma_read_desc_*`: The descriptor module issues a DMA read request
31. `dma_if_pcie_us_rd` `s_axis_read_desc_*`: The requst arrives at the DMA read interface
32. `dma_if_pcie_us_rd`: The DMA read interface issues a PCIe read request
33. `dma_if_pcie_us_rd`: The read data comes back in a completion packet and is written to the descriptor fetch local DMA RAM
34. `dma_if_pcie_us_rd` `m_axis_read_desc_status_*`: The DMA read interface issues a status message
35. *desc_fetch* (page 43) `m_axis_desc_dequeue_commit_*`: The descriptor fetch module issues a dequeue commit message
36. *queue_manager* (page 94): The queue manager commits the dequeue operation and updates the consumer pointer
37. *desc_fetch* (page 43) `dma_read_desc_*`: The descriptor fetch module issues a read request to its internal DMA module
38. *desc_fetch* (page 43) `m_axis_desc_*`: The internal DMA module reads the descriptor and transfers it via AXI stream
39. *rx_engine* (page 97): The descriptor arrives at the receive engine
40. *rx_engine* (page 97): The receive engine stores the descriptor data
41. *rx_engine* (page 97) `m_axis_dma_write_desc_*`: The receive engine issues a DMA write request
42. `dma_if_pcie_us_wr` `s_axis_write_desc_*`: The requst arrives at the DMA write interface
43. `dma_if_pcie_us_wr`: The DMA write interface reads the packet data from the interface local DMA RAM
44. `dma_if_pcie_us_wr`: The DMA write interface issues a PCIe write request
45. `dma_if_pcie_us_wr` `m_axis_write_desc_status_*`: The DMA write interface issues a status message
46. *rx_engine* (page 97) `m_axis_cpl_req_*`: The receive engine issues a completion write request

47. *cpl_write* (page 42): The completion write module writes the completion data into its local DMA RAM
48. *cpl_write* (page 42) `m_axis_cpl_enqueue_req_*`: The completion write module issues an enqueue request to the completion queue manager
49. *cpl_queue_manager* (page 42) `s_axis_enqueue_req_*`: The enqueue request arrives at the completion queue manager module
50. *cpl_queue_manager* (page 42): If the queue is not full, the queue manager starts an enqueue operation on the queue
51. *cpl_queue_manager* (page 42) `m_axis_enqueue_resp_*`: The completion queue manager sends a response containing the operation status and DMA address
52. *cpl_write* (page 42) `s_axis_cpl_enqueue_resp_*`: The response arrives at the completion write module
53. *cpl_write* (page 42) `m_axis_req_status_*`: The completion write module reports the completion write status
54. *desc_fetch* (page 43) `m_axis_dma_write_desc_*`: The completion write module issues a DMA write request
55. `dma_if_pcie_us_wr` `s_axis_write_desc_*`: The requst arrives at the DMA write interface
56. `dma_if_pcie_us_wr`: The DMA write interface reads the completion data from the completion write module local DMA RAM
57. `dma_if_pcie_us_wr`: The DMA write interface issues a PCIe write request
58. `dma_if_pcie_us_wr` `m_axis_write_desc_status_*`: The DMA write interface issues a status message
59. *cpl_write* (page 42) `m_axis_desc_enqueue_commit_*`: The completion write module issues an enqueue commit message
60. *cpl_queue_manager* (page 42): The completion queue manager commits the enqueue operation and updates the producer pointer
61. *cpl_queue_manager* (page 42) `m_axis_event_*`: The completion queue manager issues an event, if armed
62. *cpl_write* (page 42): The event arrives at the completion write module
63. *cpl_write* (page 42): The completion write module writes the event data into its local DMA RAM
64. *cpl_write* (page 42) `m_axis_cpl_enqueue_req_*`: The completion write module issues an enqueue request to the completion queue manager
65. *cpl_queue_manager* (page 42) `s_axis_enqueue_req_*`: The enqueue request arrives at the completion queue manager module
66. *cpl_queue_manager* (page 42): If the queue is not full, the queue manager starts an enqueue operation on the queue
67. *cpl_queue_manager* (page 42) `m_axis_enqueue_resp_*`: The completion queue manager sends a response containing the operation status and DMA address
68. *cpl_write* (page 42) `s_axis_cpl_enqueue_resp_*`: The response arrives at the completion write module
69. *cpl_write* (page 42) `m_axis_req_status_*`: The completion write module reports the completion write status
70. *desc_fetch* (page 43) `m_axis_dma_write_desc_*`: The completion write module issues a DMA write request
71. `dma_if_pcie_us_wr` `s_axis_write_desc_*`: The requst arrives at the DMA write interface
72. `dma_if_pcie_us_wr`: The DMA write interface reads the event data from the completion write module local DMA RAM
73. `dma_if_pcie_us_wr`: The DMA write interface issues a PCIe write request
74. `dma_if_pcie_us_wr` `m_axis_write_desc_status_*`: The DMA write interface issues a status message

75. *cpl_write* (page 42) `m_axis_desc_enqueue_commit_*`: The completion write module issues an enqueue commit message
76. *cpl_queue_manager* (page 42): The completion queue manager commits the enqueue operation and updates the producer pointer
77. *cpl_queue_manager* (page 42) `m_axis_event_*`: The completion queue manager issues an interrupt, if armed
78. linux: The linux kernel calls `mqnic_irq_handler()`
79. `mqnic_irq_handler()` (`mqnic_irq.c`): The driver calls the EQ handler via the notifier chain (`atomic_notifier_call_chain()`)
80. `mqnic_eq_int()` (`mqnic_eq.c`): The driver calls `mqnic_process_eq()`
81. `mqnic_process_eq()` (`mqnic_eq.c`): The driver processes the event queue, which calls the appropriate handler (`mqnic_rx_irq()`)
82. `mqnic_rx_irq()` (`mqnic_rx.c`): The driver enables NAPI polling on the queue (`napi_schedule_irqoff()`)
83. `mqnic_eq_int()` (`mqnic_eq.c`): The driver rearms the EQ (`mqnic_arm_eq()`)
84. NAPI: The linux kernel calls `mqnic_poll_rx_cq()`
85. `mqnic_poll_rx_cq()` (`mqnic_rx.c`): The driver calls `mqnic_process_rx_cq()`
86. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver reads the CQ producer pointer from the NIC
87. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver reads the completion record
88. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver fetches a fresh `sk_buff` (`napi_get_frags()`)
89. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver sets the `sk_buff` hardware timestamp
90. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver unmaps the pages (`dma_unmap_page()`)
91. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver associates the pages with the `sk_buff` (`__skb_fill_page_desc()`)
92. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver sets the `sk_buff` length
93. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver hands off the `sk_buff` to `napi_gro_frags()`
94. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver dequeues the completion record by incrementing the CQ consumer pointer
95. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver writes the updated CQ consumer pointer via MMIO
96. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver reads the queue consumer pointer from the NIC
97. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver increments the ring consumer pointer for in-order freed descriptors
98. `mqnic_process_rx_cq()` (`mqnic_rx.c`): The driver calls `mqnic_refill_rx_buffers()`
99. `mqnic_refill_rx_buffers()` (`mqnic_rx.c`): The driver calls `mqnic_prepare_rx_desc()` for each empty location in the ring
100. `mqnic_prepare_rx_desc()` (`mqnic_rx.c`): The driver allocates memory pages (`dev_alloc_pages()`)
101. `mqnic_prepare_rx_desc()` (`mqnic_rx.c`): The driver maps the pages (`dev_alloc_pages()`)
102. `mqnic_prepare_rx_desc()` (`mqnic_rx.c`): The driver writes a pointer to the page struct in `ring->rx_info`
103. `mqnic_prepare_rx_desc()` (`mqnic_rx.c`): The driver writes a descriptor with the DMA pointer and length
104. `mqnic_refill_rx_buffers()` (`mqnic_rx.c`): The driver enqueues the descriptor by incrementing its local copy of the producer pointer

105. `mqnic_refill_rx_buffers()` (`mqnic_rx.c`): At the end of the loop, the driver writes the updated producer pointer to the NIC via MMIO
106. `mqnic_poll_rx_cq()` (`mqnic_rx.c`): The driver disables NAPI polling, when idle (`napi_complete()`)
107. `mqnic_poll_rx_cq()` (`mqnic_rx.c`): The driver rearms the CQ (`mqnic_arm_cq()`)

CHAPTER

EIGHT

MODULES

8.1 Overview

Corundum has several unique architectural features. First, hardware queue states are stored efficiently in FPGA block RAM, enabling support for thousands of individually-controllable queues. These queues are associated with interfaces, and each interface can have multiple ports, each with its own independent transmit scheduler. This enables extremely fine-grained control over packet transmission. The scheduler module is designed to be modified or swapped out completely to implement different transmit scheduling schemes, including experimental schedulers. Coupled with PTP time synchronization, this enables time-based scheduling, including high precision TDMA.

The design of Corundum is modular and highly parametrized. Many configuration and structural options can be set at synthesis time by Verilog parameters, including interface and port counts, queue counts, memory sizes, etc. These design parameters are exposed in configuration registers that the driver reads to determine the NIC configuration, enabling the same driver to support many different boards and configurations without modification.

8.1.1 High-level overview

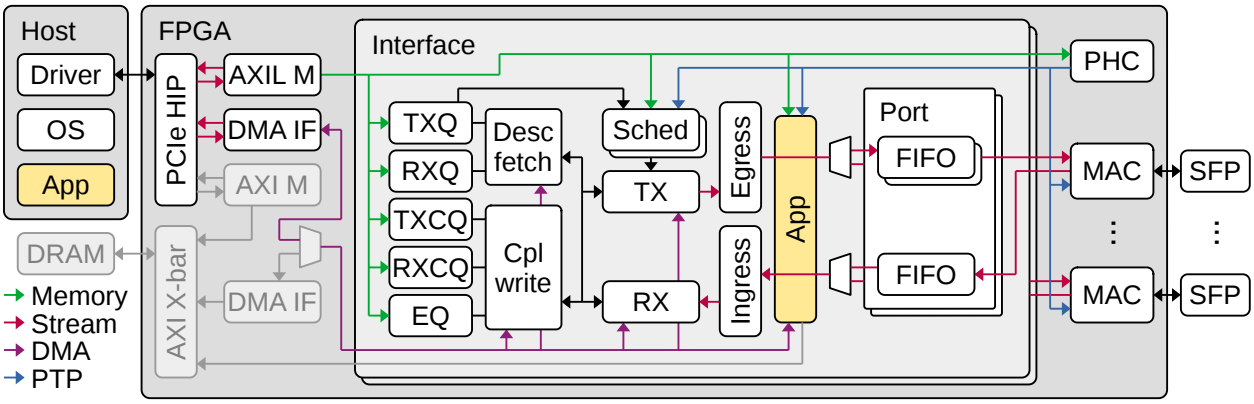


Fig. 8.1: Block diagram of the Corundum NIC. PCIe HIP: PCIe hard IP core; AXIL M: AXI lite master; DMA IF: DMA interface; AXI M: AXI master; PHC: PTP hardware clock; TXQ: transmit queue manager; TXCQ: transmit completion queue manager; RXQ: receive queue manager; RXCQ: receive completion queue manager; EQ: event queue manager; MAC + PHY: Ethernet media access controller (MAC) and physical interface layer (PHY).

A block diagram of the Corundum NIC is shown in Fig. 8.1. At a high level, the NIC consists of several hierarchy levels. The top-level module primarily contains support and interfacing components. These components include the PCI express hard IP core and Ethernet interface components including MACs, PHYs, and associated serializers, along with an instance of an appropriate *mqnic_core* (page 59) wrapper, which provides the DMA interface. This core module contains the PTP clock (*mqnic_ptp* (page 86)), application section (*mqnic_app_block* (page 43)), and one or more

mqnic_interface (page 83) module instances. Each interface module corresponds to an operating-system-level network interface (e.g. `eth0`), and contains the queue management logic, descriptor and completion handling logic, transmit schedulers, transmit and receive engines, transmit and receive datapaths, and a scratchpad RAM for temporarily storing incoming and outgoing packets during DMA operations. The queue management logic maintains the queue state for all of the NIC queues—transmit, transmit completion, receive, receive completion, and event queues.

For each interface, the transmit scheduler (*mqnic_tx_scheduler_block* (page 91)) in the interface module decides which queues are designated for transmission. The transmit scheduler generates commands for the transmit engine, which coordinates operations on the transmit datapath. The scheduler module is a flexible functional block that can be modified or replaced to support arbitrary schedules, which may be event driven. The default implementation of the scheduler in *tx_scheduler_rr* (page 98) is simple round robin. All ports associated with the same interface module share the same set of transmit queues and appear as a single, unified interface to the operating system. This enables flows to be migrated between ports or load-balanced across multiple ports by changing only the transmit scheduler settings without affecting the rest of the network stack. This dynamic, scheduler-defined mapping of queues to ports is a unique feature of Corundum that can enable research into new protocols and network architectures, including parallel networks and optically-switched networks.

In the receive direction, incoming packets pass through a flow hash module to determine the target receive queue and generate commands for the receive engine, which coordinates operations on the receive datapath. Because all ports in the same interface module share the same set of receive queues, incoming flows on different ports are merged together into the same set of queues.

An application block (*mqnic_app_block* (page 43)) is provided for customization, including packet processing, routing, and in-network compute applications. The application block has connections to several different subsystems.

The components on the NIC are interconnected with several different interfaces including AXI lite, AXI stream, and a custom segmented memory interface for DMA operations. AXI lite is used for the control path from the driver to the NIC. It is used to initialize and configure the NIC components and to control the queue pointers during transmit and receive operations. AXI stream interfaces are used for transferring packetized data within the NIC. The segmented memory interface serves to connect the PCIe DMA interface to the NIC datapath and to the descriptor and completion handling logic.

The majority of the NIC logic runs in the PCIe user clock domain, which is nominally 250 MHz for all of the current design variants. Asynchronous FIFOs are used to interface with the MACs, which run in the serializer transmit and receive clock domains as appropriate—156.25 MHz for 10G, 390.625 MHz for 25G, and 322.265625 MHz for 100G.

8.2 cpl_queue_manager

`cpl_queue_manager` implements

8.3 cpl_write

`cpl_write` manages operations associated with completion writeback. It is responsible for enqueuing completion and event records into the completion queue managers and writing records into host memory via DMA.

8.4 desc_fetch

desc_fetch manages operations associated with fetching descriptors. It is responsible for dequeuing descriptors from the queue managers and reading descriptors from host memory via DMA.

8.5 mqnic_app_block

mqnic_app_block is the top-level block for application logic. It is instantiated in *mqnic_core* (page 59). This is a pluggable module, intended to be replaced by a customized implementation via the build system. See .... for more details.

A number of interfaces are provided:

- Clock and reset synchronous with core datapath
- Dedicated AXI-lite master interface for application control (s_axil_app_ctrl)
- AXI-lite slave interface for access to NIC control register space (m_axil_ctrl)
- Access to DMA subsystem (*_axis_*_dma_*_desc, *_dma_ram)
- Access to PTP subsystem (ptp_*)
- Direct, MAC-synchronous, lowest-latency streaming interface (*_axis_direct_*)
- Direct, datapath-synchronous, low-latency streaming interface (*_axis_sync_*)
- Interface-level streaming interface (*_axis_if_*)
- Statistics interface (m_axis_stat)
- GPIO and JTAG passthrough (gpio, jtag)

Packet data from the host passes through all three streaming interfaces on its way to the network, and vise-versa. The three interfaces are:

1. *_axis_direct_*: Direct, MAC-synchronous, lowest-latency streaming interface. This interface is as close as possible to the main transmit and receive interfaces on *mqnic_core* (page 59), and is synchronous to the TX and RX clocks instead of the core clock. Enabled/bypassed via APP_AXIS_DIRECT_ENABLE in config.tcl.
2. *_axis_sync_*: Direct, datapath-synchronous, low-latency streaming interface. This interface handles per-port data between the main transmit and receive FIFOs and the async FIFOs, and is synchronous to the core clock. Enabled/bypassed via APP_AXIS_SYNC_ENABLE in config.tcl.
3. *_axis_if_*: Interface-level streaming interface. This interface handles aggregated interface-level data between the host and the main receive and transmit FIFOs, and is synchronous to the core clock. Enabled/bypassed via APP_AXIS_IF_ENABLE in config.tcl.

On the transmit path, data flows as follows:

1. *mqnic_interface_tx* (page 83): data is read from host memory via DMA
2. *mqnic_egress* (page 82): egress processing
3. s_axis_if_tx: data is presented to the application section
4. m_axis_if_tx: data is returned from the application section
5. Data passes enters per-interface transmit FIFO module and is divided into per-port, per-traffic-class FIFOs
6. s_axis_sync_tx: data is presented to the application section
7. m_axis_sync_tx: data is returned from the application section

8. Data passes through per-port transmit async FIFO module and is transferred to MAC TX clock domain
9. `s_axis_direct_tx`: data is presented to the application section
10. `m_axis_direct_tx`: data is returned from the application section
11. *mqnic_l2_egress* (page 84): layer 2 egress processing
12. *mqnic_core* (page 59): data leaves through transmit streaming interfaces

On the receive path, data flows as follows:

1. *mqnic_core* (page 59): data enters through receive streaming interfaces
2. *mqnic_l2_ingress* (page 85): layer 2 ingress processing
3. `s_axis_direct_rx`: data is presented to the application section
4. `m_axis_direct_rx`: data is returned from the application section
5. Data passes through per-port receive async FIFO module and is transferred to core clock domain
6. `s_axis_sync_rx`: data is presented to the application section
7. `m_axis_sync_rx`: data is returned from the application section
8. Data passes enters per-interface receive FIFO module and is placed into per-port FIFOs, then aggregated into a single stream
9. `s_axis_if_rx`: data is presented to the application section
10. `m_axis_if_rx`: data is returned from the application section
11. *mqnic_ingress* (page 83): ingress processing
12. *mqnic_interface_rx* (page 83): data is read from host memory via DMA

8.5.1 Parameters

`IF_COUNT`
Interface count, default `1`.

`PORTS_PER_IF`
Ports per interface, default `1`.

`SCHED_PER_IF`
Schedulers per interface, default `PORTS_PER_IF`.

`PORT_COUNT`
Total port count, must be set to `IF_COUNT*PORTS_PER_IF`.

`CLK_PERIOD_NS_NUM`
Numerator of core clock period in ns, default `4`.

`CLK_PERIOD_NS_DENOM`
Denominator of core clock period in ns, default `1`.

`PTP_CLK_PERIOD_NS_NUM`
Numerator of PTP clock period in ns, default `4`.

`PTP_CLK_PERIOD_NS_DENOM`
Denominator of PTP clock period in ns, default `1`.

PTP_TS_WIDTH

PTP timestamp width, must be `96`.

PTP_USE_SAMPLE_CLOCK

Use external PTP sample clock, used to synchronize the PTP clock across clock domains. Default `0`.

PTP_PORT_CDC_PIPELINE

Output pipeline stages on PTP clock CDC module, default `0`.

PTP_PEROUT_ENABLE

Enable PTP period output module, default `0`.

PTP_PEROUT_COUNT

Number of PTP period output channels, default `1`.

PTP_TS_ENABLE

Enable PTP timestamping, default `1`.

TX_TAG_WIDTH

Transmit tag signal width, default `16`.

MAX_TX_SIZE

Maximum packet size on transmit path, default `9214`.

MAX_RX_SIZE

Maximum packet size on receive path, default `9214`.

APP_ID

Application ID, default `0`.

APP_CTRL_ENABLE

Enable application section control connection to core NIC registers, default `1`.

APP_DMA_ENABLE

Enable application section connection to DMA subsystem, default `1`.

APP_AXIS_DIRECT_ENABLE

Enable lowest-latency asynchronous streaming connection to application section, default `1`

APP_AXIS_SYNC_ENABLE

Enable low-latency synchronous streaming connection to application section, default `1`

APP_AXIS_IF_ENABLE

Enable interface-level streaming connection to application section, default `1`

APP_STAT_ENABLE

Enable application section connection to statistics collection subsystem, default `1`

APP_GPIO_IN_WIDTH

Application section GPIO input signal width, default `32`

APP_GPIO_OUT_WIDTH

Application section GPIO output signal width, default `32`

DMA_ADDR_WIDTH

DMA interface address signal width, default `64`.

DMA_IMM_ENABLE

DMA interface immediate enable, default `0`.

DMA_IMM_WIDTH

DMA interface immediate signal width, default `32`.

DMA_LEN_WIDTH

DMA interface length signal width, default `16`.

DMA_TAG_WIDTH

DMA interface tag signal width, default `16`.

RAM_SEL_WIDTH

Width of select signal per segment in DMA RAM interface, default `4`.

RAM_ADDR_WIDTH

Width of address signal for DMA RAM interface, default `16`.

RAM_SEG_COUNT

Number of segments in DMA RAM interface, default `2`. Must be a power of 2, must be at least 2.

RAM_SEG_DATA_WIDTH

Width of data signal per segment in DMA RAM interface, default `256*2/RAM_SEG_COUNT`.

RAM_SEG_BE_WIDTH

Width of byte enable signal per segment in DMA RAM interface, default `RAM_SEG_DATA_WIDTH/8`.

RAM_SEG_ADDR_WIDTH

Width of address signal per segment in DMA RAM interface, default `RAM_ADDR_WIDTH-$clog2(RAM_SEG_COUNT*RAM_SEG_BE_WIDTH)`.

RAM_PIPELINE

Number of output pipeline stages in segmented DMA RAMs, default `2`. Tune for best usage of block RAM cascade registers.

AXIL_APP_CTRL_DATA_WIDTH

AXI lite application control data signal width, default `AXIL_CTRL_DATA_WIDTH`. Can be 32 or 64.

AXIL_APP_CTRL_ADDR_WIDTH

AXI lite application control address signal width, default `16`.

AXIL_APP_CTRL_STRB_WIDTH

AXI lite application control byte enable signal width, must be set to `AXIL_APP_CTRL_DATA_WIDTH/8`.

AXIL_CTRL_DATA_WIDTH

AXI lite control data signal width, default `32`. Must be 32.

AXIL_CTRL_ADDR_WIDTH

AXI lite control address signal width, default `16`.

AXIL_CTRL_STRB_WIDTH

AXI lite control byte enable signal width, must be set to `AXIL_CTRL_DATA_WIDTH/8`.

AXIS_DATA_WIDTH

Asynchronous streaming interface `tdata` signal width, default `512`.

AXIS_KEEP_WIDTH

Asynchronous streaming interface `tkeep` signal width, must be set to `AXIS_DATA_WIDTH/8`.

AXIS_TX_USER_WIDTH

Asynchronous streaming transmit interface `tuser` signal width, default `TX_TAG_WIDTH + 1`.

AXIS_RX_USER_WIDTH

Asynchronous streaming receive interface `tuser` signal width, default `(PTP_TS_ENABLE ? PTP_TS_WIDTH : 0) + 1`.

AXIS_RX_USE_READY

Use `tready` signal on RX interfaces, default `0`. If set, logic will exert backpressure with `tready` instead of dropping packets when RX FIFOs are full.

AXIS_SYNC_DATA_WIDTH

Synchronous streaming interface `tdata` signal width, default `AXIS_DATA_WIDTH`.

AXIS_SYNC_KEEP_WIDTH

Synchronous streaming interface `tkeep` signal width, must be set to `AXIS_SYNC_DATA_WIDTH/8`.

AXIS_SYNC_TX_USER_WIDTH

Synchronous streaming transmit interface `tuser` signal width, default `AXIS_TX_USER_WIDTH`.

AXIS_SYNC_RX_USER_WIDTH

Synchronous streaming receive interface `tuser` signal width, default `AXIS_RX_USER_WIDTH`.

AXIS_IF_DATA_WIDTH

Interface streaming interface `tdata` signal width, default `AXIS_SYNC_DATA_WIDTH*2**$clog2(PORTS_PER_IF)`.

AXIS_IF_KEEP_WIDTH

Interface streaming interface `tkeep` signal width, must be set to `AXIS_IF_DATA_WIDTH/8`.

AXIS_IF_TX_ID_WIDTH

Interface transmit streaming interface `tid` signal width, default `12`.

AXIS_IF_RX_ID_WIDTH

Interface receive streaming interface `tid` signal width, default `PORTS_PER_IF > 1 ? $clog2(PORTS_PER_IF) : 1`.

AXIS_IF_TX_DEST_WIDTH

Interface transmit streaming interface `tdest` signal width, default `$clog2(PORTS_PER_IF)+4`.

AXIS_IF_RX_DEST_WIDTH

Interface receive streaming interface `tdest` signal width, default `8`.

AXIS_IF_TX_USER_WIDTH

Interface transmit streaming interface `tuser` signal width, default `AXIS_SYNC_TX_USER_WIDTH`.

AXIS_IF_RX_USER_WIDTH

Interface receive streaming interface `tuser` signal width, default `AXIS_SYNC_RX_USER_WIDTH`.

STAT_ENABLE

Enable statistics collection subsystem, default `1`.

STAT_INC_WIDTH

Statistics increment signal width, default `24`.

STAT_ID_WIDTH

Statistics ID signal width, default `12`. Sets the number of statistics counters as `2**STAT_ID_WIDTH`.

8.5.2 Ports

clk

Logic clock. Most interfaces are synchronous to this clock.

Signal	Dir	Width	Description
clk	in	1	Logic clock

rst

Logic reset, active high

Signal	Dir	Width	Description
rst	in	1	Logic reset, active high

s_axil_app_ctrl

AXI-Lite slave interface (application control).

Signal	Dir	Width	Description
s_axil_app_ctrl_awaddr	in	AXIL_APP_CTRL_ADDR_WIDTH	Write address
s_axil_app_ctrl_awprot	in	3	Write protect
s_axil_app_ctrl_awvalid	in	1	Write address valid
s_axil_app_ctrl_awready	out	1	Write address ready
s_axil_app_ctrl_wdata	in	AXIL_APP_CTRL_DATA_WIDTH	Write data
s_axil_app_ctrl_wstrb	in	AXIL_APP_CTRL_STRB_WIDTH	Write data strobe
s_axil_app_ctrl_wvalid	in	1	Write data valid
s_axil_app_ctrl_wready	out	1	Write data ready
s_axil_app_ctrl_bresp	out	2	Write response status
s_axil_app_ctrl_bvalid	out	1	Write response valid
s_axil_app_ctrl_bready	in	1	Write response ready
s_axil_app_ctrl_araddr	in	AXIL_APP_CTRL_ADDR_WIDTH	Read address
s_axil_app_ctrl_arprot	in	3	Read protect
s_axil_app_ctrl_arvalid	in	1	Read address valid
s_axil_app_ctrl_arready	out	1	Read address ready
s_axil_app_ctrl_rdata	out	AXIL_APP_CTRL_DATA_WIDTH	Read response data
s_axil_app_ctrl_rresp	out	2	Read response status
s_axil_app_ctrl_rvalid	out	1	Read response valid
s_axil_app_ctrl_rready	in	1	Read response ready

m_axil_ctrl

AXI-Lite master interface (control). This interface provides access to the main NIC control register space.

Signal	Dir	Width	Description
m_axil_ctrl_awaddr	in	AXIL_CTRL_ADDR_WIDTH	Write address
m_axil_ctrl_awprot	in	3	Write protect
m_axil_ctrl_awvalid	in	1	Write address valid
m_axil_ctrl_awready	out	1	Write address ready
m_axil_ctrl_wdata	in	AXIL_CTRL_DATA_WIDTH	Write data
m_axil_ctrl_wstrb	in	AXIL_CTRL_STRB_WIDTH	Write data strobe
m_axil_ctrl_wvalid	in	1	Write data valid
m_axil_ctrl_wready	out	1	Write data ready
m_axil_ctrl_bresp	out	2	Write response status
m_axil_ctrl_bvalid	out	1	Write response valid
m_axil_ctrl_bready	in	1	Write response ready
m_axil_ctrl_araddr	in	AXIL_CTRL_ADDR_WIDTH	Read address
m_axil_ctrl_arprot	in	3	Read protect
m_axil_ctrl_arvalid	in	1	Read address valid
m_axil_ctrl_arready	out	1	Read address ready
m_axil_ctrl_rdata	out	AXIL_CTRL_DATA_WIDTH	Read response data
m_axil_ctrl_rresp	out	2	Read response status
m_axil_ctrl_rvalid	out	1	Read response valid
m_axil_ctrl_rready	in	1	Read response ready

m_axis_ctrl_dma_read_desc

DMA read descriptor output (control)

Signal	Dir	Width	Description
m_axis_ctrl_dma_read_desc_dma_addr	out	DMA_ADDR_WIDTH	DMA address
m_axis_ctrl_dma_read_desc_ram_sel	out	RAM_SEL_WIDTH	RAM select
m_axis_ctrl_dma_read_desc_ram_addr	out	RAM_ADDR_WIDTH	RAM address
m_axis_ctrl_dma_read_desc_len	out	DMA_LEN_WIDTH	Transfer length
m_axis_ctrl_dma_read_desc_tag	out	DMA_TAG_WIDTH	Transfer tag
m_axis_ctrl_dma_read_desc_valid	out	1	Request valid
m_axis_ctrl_dma_read_desc_ready	in	1	Request ready

s_axis_ctrl_dma_read_desc_status

DMA read descriptor status input (control)

Signal	Dir	Width	Description
s_axis_ctrl_dma_read_desc_status_tag	in	DMA_TAG_WIDTH	Status tag
s_axis_ctrl_dma_read_desc_status_error	in	4	Status error code
s_axis_ctrl_dma_read_desc_status_valid	in	1	Status valid

m_axis_ctrl_dma_write_desc

DMA write descriptor output (control)

Signal	Dir	Width	Description
m_axis_ctrl_dma_write_desc_dma_addr	out	DMA_ADDR_WIDTH	DMA address
m_axis_ctrl_dma_write_desc_ram_sel	out	RAM_SEL_WIDTH	RAM select
m_axis_ctrl_dma_write_desc_ram_addr	out	RAM_ADDR_WIDTH	RAM address
m_axis_ctrl_dma_write_desc_imm	out	DMA_IMM_WIDTH	Immediate
m_axis_ctrl_dma_write_desc_imm_en	out	1	Immediate enable
m_axis_ctrl_dma_write_desc_len	out	DMA_LEN_WIDTH	Transfer length
m_axis_ctrl_dma_write_desc_tag	out	DMA_TAG_WIDTH	Transfer tag
m_axis_ctrl_dma_write_desc_valid	out	1	Request valid
m_axis_ctrl_dma_write_desc_ready	in	1	Request ready

s_axis_ctrl_dma_write_desc_status

DMA write descriptor status input (control)

Signal	Dir	Width	Description
s_axis_ctrl_dma_write_desc_status_tag	in	DMA_TAG_WIDTH	Status tag
s_axis_ctrl_dma_write_desc_status_error	in	4	Status error code
s_axis_ctrl_dma_write_desc_status_valid	in	1	Status valid

m_axis_data_dma_read_desc

DMA read descriptor output (data)

Signal	Dir	Width	Description
m_axis_data_dma_read_desc_dma_addr	out	DMA_ADDR_WIDTH	DMA address
m_axis_data_dma_read_desc_ram_sel	out	RAM_SEL_WIDTH	RAM select
m_axis_data_dma_read_desc_ram_addr	out	RAM_ADDR_WIDTH	RAM address
m_axis_data_dma_read_desc_len	out	DMA_LEN_WIDTH	Transfer length
m_axis_data_dma_read_desc_tag	out	DMA_TAG_WIDTH	Transfer tag
m_axis_data_dma_read_desc_valid	out	1	Request valid
m_axis_data_dma_read_desc_ready	in	1	Request ready

s_axis_data_dma_read_desc_status

DMA read descriptor status input (data)

Signal	Dir	Width	Description
s_axis_data_dma_read_desc_status_tag	in	DMA_TAG_WIDTH	Status tag
s_axis_data_dma_read_desc_status_error	in	4	Status error code
s_axis_data_dma_read_desc_status_valid	in	1	Status valid

m_axis_data_dma_write_desc

DMA write descriptor output (data)

Signal	Dir	Width	Description
m_axis_data_dma_write_desc_dma_addr	out	DMA_ADDR_WIDTH	DMA address
m_axis_data_dma_write_desc_ram_sel	out	RAM_SEL_WIDTH	RAM select
m_axis_data_dma_write_desc_ram_addr	out	RAM_ADDR_WIDTH	RAM address
m_axis_data_dma_write_desc_imm	out	DMA_IMM_WIDTH	Immediate
m_axis_data_dma_write_desc_imm_en	out	1	Immediate enable
m_axis_data_dma_write_desc_len	out	DMA_LEN_WIDTH	Transfer length
m_axis_data_dma_write_desc_tag	out	DMA_TAG_WIDTH	Transfer tag
m_axis_data_dma_write_desc_valid	out	1	Request valid
m_axis_data_dma_write_desc_ready	in	1	Request ready

s_axis_data_dma_write_desc_status

DMA write descriptor status input (data)

Signal	Dir	Width	Description
s_axis_data_dma_write_desc_status_tag	in	DMA_TAG_WIDTH	Status tag
s_axis_data_dma_write_desc_status_error	in	4	Status error code
s_axis_data_dma_write_desc_status_valid	in	1	Status valid

ctrl_dma_ram

DMA RAM interface (control)

Signal	Dir	Width	Description
ctrl_dma_ram_wr_cmd_sel	in	RAM_SEG_COUNT*RAM_SEL_WIDTH	Write command select
ctrl_dma_ram_wr_cmd_be	in	RAM_SEG_COUNT*RAM_SEG_BE_WIDTH	Write command byte enable
ctrl_dma_ram_wr_cmd_addr	in	RAM_SEG_COUNT*RAM_SEG_ADDR_WIDTH	Write command address
ctrl_dma_ram_wr_cmd_data	in	RAM_SEG_COUNT*RAM_SEG_DATA_WIDTH	Write command data
ctrl_dma_ram_wr_cmd_valid	in	RAM_SEG_COUNT	Write command valid
ctrl_dma_ram_wr_cmd_ready	out	RAM_SEG_COUNT	Write command ready
ctrl_dma_ram_wr_done	out	RAM_SEG_COUNT	Write done
ctrl_dma_ram_rd_cmd_sel	in	RAM_SEG_COUNT*RAM_SEL_WIDTH	Read command select
ctrl_dma_ram_rd_cmd_addr	in	RAM_SEG_COUNT*RAM_SEG_ADDR_WIDTH	Read command address
ctrl_dma_ram_rd_cmd_valid	in	RAM_SEG_COUNT	Read command valid
ctrl_dma_ram_rd_cmd_ready	out	RAM_SEG_COUNT	Read command ready
ctrl_dma_ram_rd_resp_data	out	RAM_SEG_COUNT*RAM_SEG_DATA_WIDTH	Read response data
ctrl_dma_ram_rd_resp_valid	out	RAM_SEG_COUNT	Read response valid
ctrl_dma_ram_rd_resp_ready	in	RAM_SEG_COUNT	Read response ready

data_dma_ram

DMA RAM interface (data)

Signal	Dir	Width	Description
data_dma_ram_wr_cmd_sel	in	RAM_SEG_COUNT*RAM_SEL_WIDTH	Write command select
data_dma_ram_wr_cmd_be	in	RAM_SEG_COUNT*RAM_SEG_BE_WIDTH	Write command byte enable
data_dma_ram_wr_cmd_addr	in	RAM_SEG_COUNT*RAM_SEG_ADDR_WIDTH	Write command address
data_dma_ram_wr_cmd_data	in	RAM_SEG_COUNT*RAM_SEG_DATA_WIDTH	Write command data
data_dma_ram_wr_cmd_valid	in	RAM_SEG_COUNT	Write command valid
data_dma_ram_wr_cmd_ready	out	RAM_SEG_COUNT	Write command ready
data_dma_ram_wr_done	out	RAM_SEG_COUNT	Write done
data_dma_ram_rd_cmd_sel	in	RAM_SEG_COUNT*RAM_SEL_WIDTH	Read command select
data_dma_ram_rd_cmd_addr	in	RAM_SEG_COUNT*RAM_SEG_ADDR_WIDTH	Read command address
data_dma_ram_rd_cmd_valid	in	RAM_SEG_COUNT	Read command valid
data_dma_ram_rd_cmd_ready	out	RAM_SEG_COUNT	Read command ready
data_dma_ram_rd_resp_data	out	RAM_SEG_COUNT*RAM_SEG_DATA_WIDTH	Read response data
data_dma_ram_rd_resp_valid	out	RAM_SEG_COUNT	Read response valid
data_dma_ram_rd_resp_ready	in	RAM_SEG_COUNT	Read response ready

ptp

PTP clock connections.

Signal	Dir	Width	Description
ptp_clk	in	1	PTP clock
ptp_rst	in	1	PTP reset
ptp_sample_clk	in	1	PTP sample clock
ptp_pps	in	1	PTP pulse-per-second (synchronous to ptp_clk)
ptp_ts_96	in	PTP_TS_WIDTH	current PTP time (synchronous to ptp_clk)
ptp_ts_step	in	1	PTP clock step (synchronous to ptp_clk)
ptp_sync_pps	in	1	PTP pulse-per-second (synchronous to clk)
ptp_sync_ts_96	in	PTP_TS_WIDTH	current PTP time (synchronous to clk)
ptp_sync_ts_step	in	1	PTP clock step (synchronous to clk)
ptp_perout_locked	in	PTP_PEROUT_COUNT	PTP period output locked
ptp_perout_error	in	PTP_PEROUT_COUNT	PTP period output error
ptp_perout_pulse	in	PTP_PEROUT_COUNT	PTP period output pulse

direct_tx_clk

Transmit clocks for direct asynchronous streaming interfaces, one per port

Signal	Dir	Width	Description
direct_tx_clk	in	PORT_COUNT	Transmit clock

direct_tx_rst

Transmit resets for direct asynchronous streaming interfaces, one per port

Signal	Dir	Width	Description
direct_tx_rst	in	PORT_COUNT	Transmit reset

s_axis_direct_tx

Streaming transmit data from host, one AXI stream interface per port. Lowest latency interface, synchronous with transmit clock.

Signal	Dir	Width	Description
s_axis_direct_tx_tdata	in	PORT_COUNT*AXIS_DATA_WIDTH	Streaming data
s_axis_direct_tx_tkeep	in	PORT_COUNT*AXIS_KEEP_WIDTH	Byte enable
s_axis_direct_tx_tvalid	in	PORT_COUNT	Data valid
s_axis_direct_tx_tready	out	PORT_COUNT	Ready for data
s_axis_direct_tx_tlast	in	PORT_COUNT	End of frame
s_axis_direct_tx_tuser	in	PORT_COUNT*AXIS_TX_USER_WIDTH	Sideband data

`s_axis_direct_tx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
TX_TAG_WIDTH:1	tx_tag	TX_TAG_WIDTH	Transmit tag

m_axis_direct_tx

Streaming transmit data towards network, one AXI stream interface per port. Lowest latency interface, synchronous with transmit clock.

Signal	Dir	Width	Description
m_axis_direct_tx_tdata	out	PORT_COUNT*AXIS_DATA_WIDTH	Streaming data
m_axis_direct_tx_tkeep	out	PORT_COUNT*AXIS_KEEP_WIDTH	Byte enable
m_axis_direct_tx_tvalid	out	PORT_COUNT	Data valid
m_axis_direct_tx_tready	in	PORT_COUNT	Ready for data
m_axis_direct_tx_tlast	out	PORT_COUNT	End of frame
m_axis_direct_tx_tuser	out	PORT_COUNT*AXIS_TX_USER_WIDTH	Sideband data

`m_axis_direct_tx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
TX_TAG_WIDTH:1	tx_tag	TX_TAG_WIDTH	Transmit tag

`s_axis_direct_tx_cpl`

Transmit PTP timestamp from MAC, one AXI stream interface per port.

Signal	Dir	Width	Description
s_axis_direct_tx_cpl_ts	in	PORT_COUNT*PTP_TS_WIDTH	PTP timestamp
s_axis_direct_tx_cpl_tag	in	PORT_COUNT*TX_TAG_WIDTH	Transmit tag
s_axis_direct_tx_cpl_valid	in	PORT_COUNT	Transmit completion valid
s_axis_direct_tx_cpl_ready	out	PORT_COUNT	Transmit completion ready

`m_axis_direct_tx_cpl`

Transmit PTP timestamp towards core logic, one AXI stream interface per port.

Signal	Dir	Width	Description
s_axis_direct_tx_cpl_ts	out	PORT_COUNT*PTP_TS_WIDTH	PTP timestamp
s_axis_direct_tx_cpl_tag	out	PORT_COUNT*TX_TAG_WIDTH	Transmit tag
s_axis_direct_tx_cpl_valid	out	PORT_COUNT	Transmit completion valid
s_axis_direct_tx_cpl_ready	in	PORT_COUNT	Transmit completion ready

`direct_rx_clk`

Receive clocks for direct asynchronous streaming interfaces, one per port

Signal	Dir	Width	Description
direct_rx_clk	in	PORT_COUNT	Receive clock

`direct_rx_rst`

Receive resets for direct asynchronous streaming interfaces, one per port

Signal	Dir	Width	Description
direct_rx_rst	in	PORT_COUNT	Receive reset

`s_axis_direct_rx`

Streaming receive data from network, one AXI stream interface per port. Lowest latency interface, synchronous with receive clock.

Signal	Dir	Width	Description
s_axis_direct_rx_tdata	in	PORT_COUNT*AXIS_DATA_WIDTH	Streaming data
s_axis_direct_rx_tkeep	in	PORT_COUNT*AXIS_KEEP_WIDTH	Byte enable
s_axis_direct_rx_tvalid	in	PORT_COUNT	Data valid
s_axis_direct_rx_tready	out	PORT_COUNT	Ready for data
s_axis_direct_rx_tlast	in	PORT_COUNT	End of frame
s_axis_direct_rx_tuser	in	PORT_COUNT*AXIS_RX_USER_WIDTH	Sideband data

`s_axis_direct_rx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

`m_axis_direct_rx`

Streaming receive data towards host, one AXI stream interface per port. Lowest latency interface, synchronous with receive clock.

Signal	Dir	Width	Description
m_axis_direct_rx_tdata	out	PORT_COUNT*AXIS_DATA_WIDTH	Streaming data
m_axis_direct_rx_tkeep	out	PORT_COUNT*AXIS_KEEP_WIDTH	Byte enable
m_axis_direct_rx_tvalid	out	PORT_COUNT	Data valid
m_axis_direct_rx_tready	in	PORT_COUNT	Ready for data
m_axis_direct_rx_tlast	out	PORT_COUNT	End of frame
m_axis_direct_rx_tuser	out	PORT_COUNT*AXIS_RX_USER_WIDTH	Sideband data

`m_axis_direct_rx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

`s_axis_sync_tx`

Streaming transmit data from host, one AXI stream interface per port. Low latency interface, synchronous with core clock.

Signal	Dir	Width	Description
s_axis_sync_tx_tdata	in	PORT_COUNT*AXIS_SYNC_DATA_WIDTH	Streaming data
s_axis_sync_tx_tkeep	in	PORT_COUNT*AXIS_SYNC_KEEP_WIDTH	Byte enable
s_axis_sync_tx_tvalid	in	PORT_COUNT	Data valid
s_axis_sync_tx_tready	out	PORT_COUNT	Ready for data
s_axis_sync_tx_tlast	in	PORT_COUNT	End of frame
s_axis_sync_tx_tuser	in	PORT_COUNT*AXIS_SYNC_TX_USER_WIDTH	Sideband data

`s_axis_sync_tx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
TX_TAG_WIDTH:1	tx_tag	TX_TAG_WIDTH	Transmit tag

m_axis_sync_tx

Streaming transmit data towards network, one AXI stream interface per port. Low latency interface, synchronous with core clock.

Signal	Dir	Width	Description
m_axis_sync_tx_tdata	out	PORT_COUNT*AXIS_SYNC_DATA_WIDTH	Streaming data
m_axis_sync_tx_tkeep	out	PORT_COUNT*AXIS_SYNC_KEEP_WIDTH	Byte enable
m_axis_sync_tx_tvalid	out	PORT_COUNT	Data valid
m_axis_sync_tx_tready	in	PORT_COUNT	Ready for data
m_axis_sync_tx_tlast	out	PORT_COUNT	End of frame
m_axis_sync_tx_tuser	out	PORT_COUNT*AXIS_SYNC_TX_USER_WIDTH	Sideband data

`m_axis_sync_tx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
TX_TAG_WIDTH:1	tx_tag	TX_TAG_WIDTH	Transmit tag

s_axis_sync_tx_cpl

Transmit PTP timestamp from MAC, one AXI stream interface per port.

Signal	Dir	Width	Description
s_axis_sync_tx_cpl_ts	in	PORT_COUNT*PTP_TS_WIDTH	PTP timestamp
s_axis_sync_tx_cpl_tag	in	PORT_COUNT*TX_TAG_WIDTH	Transmit tag
s_axis_sync_tx_cpl_valid	in	PORT_COUNT	Transmit completion valid
s_axis_sync_tx_cpl_ready	out	PORT_COUNT	Transmit completion ready

m_axis_sync_tx_cpl

Transmit PTP timestamp towards core logic, one AXI stream interface per port.

Signal	Dir	Width	Description
s_axis_sync_tx_cpl_ts	out	PORT_COUNT*PTP_TS_WIDTH	PTP timestamp
s_axis_sync_tx_cpl_tag	out	PORT_COUNT*TX_TAG_WIDTH	Transmit tag
s_axis_sync_tx_cpl_valid	out	PORT_COUNT	Transmit completion valid
s_axis_sync_tx_cpl_ready	in	PORT_COUNT	Transmit completion ready

s_axis_sync_rx

Streaming receive data from network, one AXI stream interface per port. Low latency interface, synchronous with core clock.

Signal	Dir	Width	Description
s_axis_sync_rx_tdata	in	PORT_COUNT*AXIS_SYNC_DATA_WIDTH	Streaming data
s_axis_sync_rx_tkeep	in	PORT_COUNT*AXIS_SYNC_KEEP_WIDTH	Byte enable
s_axis_sync_rx_tvalid	in	PORT_COUNT	Data valid
s_axis_sync_rx_tready	out	PORT_COUNT	Ready for data
s_axis_sync_rx_tlast	in	PORT_COUNT	End of frame
s_axis_sync_rx_tuser	in	PORT_COUNT*AXIS_SYNC_RX_USER_WIDTH	Sideband data

`s_axis_sync_rx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

m_axis_sync_rx

Streaming receive data towards host, one AXI stream interface per port. Low latency interface, synchronous with core clock.

Signal	Dir	Width	Description
m_axis_sync_rx_tdata	out	PORT_COUNT*AXIS_SYNC_DATA_WIDTH	Streaming data
m_axis_sync_rx_tkeep	out	PORT_COUNT*AXIS_SYNC_KEEP_WIDTH	Byte enable
m_axis_sync_rx_tvalid	out	PORT_COUNT	Data valid
m_axis_sync_rx_tready	in	PORT_COUNT	Ready for data
m_axis_sync_rx_tlast	out	PORT_COUNT	End of frame
m_axis_sync_rx_tuser	out	PORT_COUNT*AXIS_SYNC_RX_USER_WIDTH	Sideband data

`m_axis_sync_rx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

s_axis_if_tx

Streaming transmit data from host, one AXI stream interface per interface. Closest interface to host, synchronous with core clock.

Signal	Dir	Width	Description
s_axis_if_tx_tdata	in	PORT_COUNT*AXIS_IF_DATA_WIDTH	Streaming data
s_axis_if_tx_tkeep	in	PORT_COUNT*AXIS_IF_KEEP_WIDTH	Byte enable
s_axis_if_tx_tvalid	in	PORT_COUNT	Data valid
s_axis_if_tx_tready	out	PORT_COUNT	Ready for data
s_axis_if_tx_tlast	in	PORT_COUNT	End of frame
s_axis_if_tx_tid	in	PORT_COUNT*AXIS_IF_TX_ID_WIDTH	Source queue
s_axis_if_tx_tdest	in	PORT_COUNT*AXIS_IF_TX_DEST_WIDTH	Destination port
s_axis_if_tx_tuser	in	PORT_COUNT*AXIS_IF_TX_USER_WIDTH	Sideband data

`s_axis_if_tx_tuser` bits, per interface

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
TX_TAG_WIDTH:1	tx_tag	TX_TAG_WIDTH	Transmit tag

m_axis_if_tx

Streaming transmit data towards network, one AXI stream interface per interface. Closest interface to host, synchronous with core clock.

Signal	Dir	Width	Description
m_axis_if_tx_tdata	out	PORT_COUNT*AXIS_IF_DATA_WIDTH	Streaming data
m_axis_if_tx_tkeep	out	PORT_COUNT*AXIS_IF_KEEP_WIDTH	Byte enable
m_axis_if_tx_tvalid	out	PORT_COUNT	Data valid
m_axis_if_tx_tready	in	PORT_COUNT	Ready for data
m_axis_if_tx_tlast	out	PORT_COUNT	End of frame
m_axis_if_tx_tid	out	PORT_COUNT*AXIS_IF_TX_ID_WIDTH	Source queue
m_axis_if_tx_tdest	out	PORT_COUNT*AXIS_IF_TX_DEST_WIDTH	Destination port
m_axis_if_tx_tuser	out	PORT_COUNT*AXIS_IF_TX_USER_WIDTH	Sideband data

`m_axis_if_tx_tuser` bits, per interface

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
TX_TAG_WIDTH:1	tx_tag	TX_TAG_WIDTH	Transmit tag

`s_axis_if_tx_cpl`

Transmit PTP timestamp from MAC, one AXI stream interface per interface.

Signal	Dir	Width	Description
s_axis_if_tx_cpl_ts	in	PORT_COUNT*PTP_TS_WIDTH	PTP timestamp
s_axis_if_tx_cpl_tag	in	PORT_COUNT*TX_TAG_WIDTH	Transmit tag
s_axis_if_tx_cpl_valid	in	PORT_COUNT	Transmit completion valid
s_axis_if_tx_cpl_ready	out	PORT_COUNT	Transmit completion ready

`m_axis_if_tx_cpl`

Transmit PTP timestamp towards core logic, one AXI stream interface per interface.

Signal	Dir	Width	Description
s_axis_if_tx_cpl_ts	out	PORT_COUNT*PTP_TS_WIDTH	PTP timestamp
s_axis_if_tx_cpl_tag	out	PORT_COUNT*TX_TAG_WIDTH	Transmit tag
s_axis_if_tx_cpl_valid	out	PORT_COUNT	Transmit completion valid
s_axis_if_tx_cpl_ready	in	PORT_COUNT	Transmit completion ready

`s_axis_if_rx`

Streaming receive data from network, one AXI stream interface per interface. Closest interface to host, synchronous with core clock.

Signal	Dir	Width	Description
s_axis_if_rx_tdata	in	PORT_COUNT*AXIS_IF_DATA_WIDTH	Streaming data
s_axis_if_rx_tkeep	in	PORT_COUNT*AXIS_IF_KEEP_WIDTH	Byte enable
s_axis_if_rx_tvalid	in	PORT_COUNT	Data valid
s_axis_if_rx_tready	out	PORT_COUNT	Ready for data
s_axis_if_rx_tlast	in	PORT_COUNT	End of frame
s_axis_if_rx_tid	in	PORT_COUNT*AXIS_IF_RX_ID_WIDTH	Source port
s_axis_if_rx_tdest	in	PORT_COUNT*AXIS_IF_RX_DEST_WIDTH	Destination queue
s_axis_if_rx_tuser	in	PORT_COUNT*AXIS_IF_RX_USER_WIDTH	Sideband data

`s_axis_if_rx_tuser` bits, per interface

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

m_axis_if_rx

Streaming receive data towards host, one AXI stream interface per interface. Closest interface to host, synchronous with core clock.

Signal	Dir	Width	Description
m_axis_if_rx_tdata	out	PORT_COUNT*AXIS_IF_DATA_WIDTH	Streaming data
m_axis_if_rx_tkeep	out	PORT_COUNT*AXIS_IF_KEEP_WIDTH	Byte enable
m_axis_if_rx_tvalid	out	PORT_COUNT	Data valid
m_axis_if_rx_tready	in	PORT_COUNT	Ready for data
m_axis_if_rx_tlast	out	PORT_COUNT	End of frame
m_axis_if_rx_tid	out	PORT_COUNT*AXIS_IF_RX_ID_WIDTH	Source port
m_axis_if_rx_tdest	out	PORT_COUNT*AXIS_IF_RX_DEST_WIDTH	Destination queue
m_axis_if_rx_tuser	out	PORT_COUNT*AXIS_IF_RX_USER_WIDTH	Sideband data

`m_axis_if_rx_tuser` bits, per interface

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

m_axis_stat

Statistics increment output

Signal	Dir	Width	Description
m_axis_stat_tdata	in	STAT_INC_WIDTH	Statistic increment
m_axis_stat_tid	in	STAT_ID_WIDTH	Statistic ID
m_axis_stat_tvalid	in	1	Statistic valid
m_axis_stat_tready	out	1	Statistic ready

gpio

Application section GPIO

Signal	Dir	Width	Description
gpio_in	in	APP_GPIO_IN_WIDTH	GPIO inputs
gpio_out	out	APP_GPIO_OUT_WIDTH	GPIO outputs

jtag

Application section JTAG scan chain

Signal	Dir	Width	Description
jtag_tdi	in	1	JTAG TDI
jtag_tdo	out	1	JTAG TDO
jtag_tms	in	1	JTAG TMS
jtag_tck	in	1	JTAG TCK

8.6 mqnic_core

`mqnic_core` is the core integration-level module for mqnic for all host interfaces. Contains the interfaces, asynchronous FIFOs, PTP subsystem, statistics collection subsystem, and application block.

For maximum flexibility, this module does not contain the actual host-facing DMA engine, so a wrapper is required to provide the DMA engine with the proper host-facing interface. The available wrappers are:

- *mqnic_core_pcie* (page 73) for PCI express
- *mqnic_core_axi* (page 71) for AXI

`mqnic_core` integrates the following modules:

- `stats_counter`: statistics aggregation
- *mqnic_ptp* (page 86): PTP subsystem
- *mqnic_interface* (page 83): NIC interface
- *mqnic_app_block* (page 43): Application block

8.6.1 Parameters

FPGA_ID
: FPGA JTAG ID, default is `32'hDEADBEEF`. Reported in *Firmware ID register block* (page 112).

FW_ID
: Firmware ID, default is `32'h00000000`. Reported in *Firmware ID register block* (page 112).

FW_VER
: Firmware version, default is `32'h00_00_01_00`. Reported in *Firmware ID register block* (page 112).

BOARD_ID
: Board ID, default is `16'h1234_0000`. Reported in *Firmware ID register block* (page 112).

BOARD_VER
: Board version, default is `32'h01_00_00_00`. Reported in *Firmware ID register block* (page 112).

BUILD_DATE
: Build date as a 32-bit unsigned Unix timestamp, default is `32'd602976000`. Reported in *Firmware ID register block* (page 112).

GIT_HASH
: 32 bits of the git commit hash, default is `32'hdce357bf`. Reported in *Firmware ID register block* (page 112).

RELEASE_INFO
: Additional release info, default is `32'h00000000`. Reported in *Firmware ID register block* (page 112).

IF_COUNT
: Interface count, default `1`.

PORTS_PER_IF
: Ports per interface, default `1`.

SCHED_PER_IF
: Schedulers per interface, default `PORTS_PER_IF`.

PORT_COUNT

Total port count, must be set to `IF_COUNT*PORTS_PER_IF`.

CLK_PERIOD_NS_NUM

Numerator of core clock period in ns, default `4`.

CLK_PERIOD_NS_DENOM

Denominator of core clock period in ns, default `1`.

PTP_CLK_PERIOD_NS_NUM

Numerator of PTP clock period in ns, default `4`.

PTP_CLK_PERIOD_NS_DENOM

Denominator of PTP clock period in ns, default `1`.

PTP_TS_WIDTH

PTP timestamp width, must be `96`.

PTP_CLOCK_PIPELINE

Output pipeline stages on PTP clock module, default `0`.

PTP_CLOCK_CDC_PIPELINE

Output pipeline stages on PTP clock CDC module, default `0`.

PTP_USE_SAMPLE_CLOCK

Use external PTP sample clock, used to synchronize the PTP clock across clock domains, default `0`.

PTP_SEPARATE_RX_CLOCK

Use `rx_ptp_clk` instead of `rx_clk` for providing current PTP time if set, default `0`.

PTP_PORT_CDC_PIPELINE

Output pipeline stages on PTP clock CDC module, default `0`.

PTP_PEROUT_ENABLE

Enable PTP period output module, default `0`.

PTP_PEROUT_COUNT

Number of PTP period output channels, default `1`.

EVENT_QUEUE_OP_TABLE_SIZE

Event queue manager operation table size, default `32`.

TX_QUEUE_OP_TABLE_SIZE

Transmit queue manager operation table size, default `32`.

RX_QUEUE_OP_TABLE_SIZE

Receive queue manager operation table size, default `32`.

TX_CPL_QUEUE_OP_TABLE_SIZE

Transmit completion queue operation table size, default `TX_QUEUE_OP_TABLE_SIZE`.

RX_CPL_QUEUE_OP_TABLE_SIZE

Receive completion queue operation table size, default `RX_QUEUE_OP_TABLE_SIZE`.

EVENT_QUEUE_INDEX_WIDTH

Event queue index width, default `5`. Sets the number of event queues on each interfaces as `2**EVENT_QUEUE_INDEX_WIDTH`.

TX_QUEUE_INDEX_WIDTH

Transmit queue index width, default `13`. Sets the number of transmit queues on each interfaces as `2**TX_QUEUE_INDEX_WIDTH`.

RX_QUEUE_INDEX_WIDTH

Receive queue index width, default `8`. Sets the number of receive queues on each interfaces as `2**RX_QUEUE_INDEX_WIDTH`.

TX_CPL_QUEUE_INDEX_WIDTH

Transmit completion queue index width, default `TX_QUEUE_INDEX_WIDTH`. Sets the number of transmit completion queues on each interfaces as `2**TX_CPL_QUEUE_INDEX_WIDTH`.

RX_CPL_QUEUE_INDEX_WIDTH

Receive completion queue index width, default `RX_QUEUE_INDEX_WIDTH`. Sets the number of receive completion queues on each interfaces as `2**RX_CPL_QUEUE_INDEX_WIDTH`.

EVENT_QUEUE_PIPELINE

Event queue manager pipeline length, default `3`. Tune for best usage of block RAM cascade registers for specified queue count.

TX_QUEUE_PIPELINE

Transmit queue manager pipeline stages, default `3+(TX_QUEUE_INDEX_WIDTH > 12 ? TX_QUEUE_INDEX_WIDTH-12 : 0)`. Tune for best usage of block RAM cascade registers for specified queue count.

RX_QUEUE_PIPELINE

Receive queue manager pipeline stages, default `3+(RX_QUEUE_INDEX_WIDTH > 12 ? RX_QUEUE_INDEX_WIDTH-12 : 0)`. Tune for best usage of block RAM cascade registers for specified queue count.

TX_CPL_QUEUE_PIPELINE

Transmit completion queue manager pipeline stages, default `TX_QUEUE_PIPELINE`. Tune for best usage of block RAM cascade registers for specified queue count.

RX_CPL_QUEUE_PIPELINE

Receive completion queue manager pipeline stages, default `RX_QUEUE_PIPELINE`. Tune for best usage of block RAM cascade registers for specified queue count.

TX_DESC_TABLE_SIZE

Transmit engine descriptor table size, default `32`.

RX_DESC_TABLE_SIZE

Receive engine descriptor table size, default `32`.

TX_SCHEDULER_OP_TABLE_SIZE

Transmit scheduler operation table size, default `TX_DESC_TABLE_SIZE`.

TX_SCHEDULER_PIPELINE

Transmit scheduler pipeline stages, default `TX_QUEUE_PIPELINE`. Tune for best usage of block RAM cascade registers for specified queue count.

TDMA_INDEX_WIDTH

TDMA index width, default `6`. Sets the number of TDMA timeslots as `2**TDMA_INDEX_WIDTH`.

PTP_TS_ENABLE

Enable PTP timestamping, default `1`.

TX_CPL_ENABLE

Enable transmit completions from MAC, default 1.

TX_CPL_FIFO_DEPTH

Depth of transmit completion FIFO, default 32.

TX_TAG_WIDTH

Transmit tag signal width, default $clog2(TX_DESC_TABLE_SIZE)+1.

TX_CHECKSUM_ENABLE

Enable TCP/UDP checksum offloading on transmit path, default 1.

RX_RSS_ENABLE

Enable receive side scaling, default 1. Requires RX_HASH_ENABLE to be set.

RX_HASH_ENABLE

Enable Toeplitz flow hashing for RX traffic, default 1.

RX_CHECKSUM_ENABLE

Enable TCP/UDP checksum offloading on receive path, default 1

TX_FIFO_DEPTH

Transmit FIFO depth in bytes, per output port, per traffic class, default 32768.

RX_FIFO_DEPTH

Receive FIFO depth in bytes, per output port, default 32768.

MAX_TX_SIZE

Maximum packet size on transmit path, default 9214.

MAX_RX_SIZE

Maximum packet size on receive path, default 9214.

TX_RAM_SIZE

Transmit scratchpad RAM size per interface, default 32768.

RX_RAM_SIZE

Receive scratchpad RAM size per interface, default 32768.

APP_ID

Application ID, default 0.

APP_ENABLE

Enable application section, default 0.

APP_CTRL_ENABLE

Enable application section control connection to core NIC registers, default 1.

APP_DMA_ENABLE

Enable application section connection to DMA subsystem, default 1.

APP_AXIS_DIRECT_ENABLE

Enable lowest-latency asynchronous streaming connection to application section, default 1

APP_AXIS_SYNC_ENABLE

Enable low-latency synchronous streaming connection to application section, default 1

APP_AXIS_IF_ENABLE

Enable interface-level streaming connection to application section, default `1`

APP_STAT_ENABLE

Enable application section connection to statistics collection subsystem, default `1`

APP_GPIO_IN_WIDTH

Application section GPIO input signal width, default `32`

APP_GPIO_OUT_WIDTH

Application section GPIO output signal width, default `32`

DMA_ADDR_WIDTH

DMA interface address signal width, default `64`.

DMA_IMM_ENABLE

DMA interface immediate enable, default `0`.

DMA_IMM_WIDTH

DMA interface immediate signal width, default `32`.

DMA_LEN_WIDTH

DMA interface length signal width, default `16`.

DMA_TAG_WIDTH

DMA interface tag signal width, default `16`.

IF_RAM_SEL_WIDTH

Width of interface-level select signal, default `1`.

RAM_SEL_WIDTH

Width of select signal per segment in DMA RAM interface, default `$clog2(IF_COUNT+(APP_ENABLE && APP_DMA_ENABLE ? 1 : 0))+IF_RAM_SEL_WIDTH+1`.

RAM_ADDR_WIDTH

Width of address signal for DMA RAM interface, default `$clog2(TX_RAM_SIZE > RX_RAM_SIZE ? TX_RAM_SIZE : RX_RAM_SIZE)`.

RAM_SEG_COUNT

Number of segments in DMA RAM interface, default `2`. Must be a power of 2, must be at least 2.

RAM_SEG_DATA_WIDTH

Width of data signal per segment in DMA RAM interface, default `256*2/RAM_SEG_COUNT`.

RAM_SEG_BE_WIDTH

Width of byte enable signal per segment in DMA RAM interface, must be set to `RAM_SEG_DATA_WIDTH/8`.

RAM_SEG_ADDR_WIDTH

Width of address signal per segment in DMA RAM interface, default `RAM_ADDR_WIDTH-$clog2(RAM_SEG_COUNT*RAM_SEG_BE_WIDTH)`.

RAM_PIPELINE

Number of output pipeline stages in segmented DMA RAMs, default `2`. Tune for best usage of block RAM cascade registers.

MSI_COUNT

Number of interrupt channels, default `32`.

AXIL_CTRL_DATA_WIDTH

AXI lite control data signal width, must be set to `32`.

AXIL_CTRL_ADDR_WIDTH

AXI lite control address signal width, default `16`.

AXIL_CTRL_STRB_WIDTH

AXI lite control byte enable signal width, must be set to `AXIL_CTRL_DATA_WIDTH/8`.

AXIL_IF_CTRL_ADDR_WIDTH

AXI lite interface control address signal width, default `AXIL_CTRL_ADDR_WIDTH-$clog2(IF_COUNT)`

AXIL_CSR_ADDR_WIDTH

AXI lite interface CSR address signal width, default `AXIL_IF_CTRL_ADDR_WIDTH-5-$clog2((PORTS_PER_IF+3)/8)`

AXIL_CSR_PASSTHROUGH_ENABLE

Enable NIC control register space passthrough, default `0`.

RB_NEXT_PTR

Next pointer of last register block in the NIC-level CSR space, default `0`.

AXIL_APP_CTRL_DATA_WIDTH

AXI lite application control data signal width, default `AXIL_CTRL_DATA_WIDTH`. Can be 32 or 64.

AXIL_APP_CTRL_ADDR_WIDTH

AXI lite application control address signal width, default `16`.

AXIL_APP_CTRL_STRB_WIDTH

AXI lite application control byte enable signal width, must be set to `AXIL_APP_CTRL_DATA_WIDTH/8`.

AXIS_DATA_WIDTH

Streaming interface `tdata` signal width, default `512`.

AXIS_KEEP_WIDTH

Streaming interface `tkeep` signal width, must be set to `AXIS_DATA_WIDTH/8`.

AXIS_SYNC_DATA_WIDTH

Synchronous streaming interface `tdata` signal width, default `AXIS_DATA_WIDTH`.

AXIS_IF_DATA_WIDTH

Interface streaming interface `tdata` signal width, default `AXIS_SYNC_DATA_WIDTH*2**$clog2(PORTS_PER_IF)`.

AXIS_TX_USER_WIDTH

Transmit streaming interface `tuser` signal width, default `TX_TAG_WIDTH + 1`.

AXIS_RX_USER_WIDTH

Receive streaming interface `tuser` signal width, default `(PTP_TS_ENABLE ? PTP_TS_WIDTH : 0) + 1`.

AXIS_RX_USE_READY

Use `tready` signal on RX interfaces, default `0`. If set, logic will exert backpressure with `tready` instead of dropping packets when RX FIFOs are full.

AXIS_TX_PIPELINE

Number of stages in transmit path pipeline FIFO, default `0`. Useful for SLR crossings.

AXIS_TX_FIFO_PIPELINE

Number of output pipeline stages in transmit FIFO, default `2`. Tune for best usage of block RAM cascade registers.

AXIS_TX_TS_PIPELINE

Number of stages in transmit path PTP timestamp pipeline FIFO, default `0`. Useful for SLR crossings.

AXIS_RX_PIPELINE

Number of stages in receive path pipeline FIFO, default `0`. Useful for SLR crossings.

AXIS_RX_FIFO_PIPELINE

Number of output pipeline stages in receive FIFO, default `2`. Tune for best usage of block RAM cascade registers.

STAT_ENABLE

Enable statistics collection subsystem, default `1`.

STAT_INC_WIDTH

Statistics increment signal width, default `24`.

STAT_ID_WIDTH

Statistics ID signal width, default `12`. Sets the number of statistics counters as `2**STAT_ID_WIDTH`.

8.6.2 Ports

clk

Logic clock. Most interfaces are synchronous to this clock.

Signal	Dir	Width	Description
clk	in	1	Logic clock

rst

Logic reset, active high

Signal	Dir	Width	Description
rst	in	1	Logic reset, active high

s_axil_ctrl

AXI-Lite slave interface (control). This interface provides access to the main NIC control register space.

Signal	Dir	Width	Description
s_axil_ctrl_awaddr	in	AXIL_CTRL_ADDR_WIDTH	Write address
s_axil_ctrl_awprot	in	3	Write protect
s_axil_ctrl_awvalid	in	1	Write address valid
s_axil_ctrl_awready	out	1	Write address ready
s_axil_ctrl_wdata	in	AXIL_CTRL_DATA_WIDTH	Write data
s_axil_ctrl_wstrb	in	AXIL_CTRL_STRB_WIDTH	Write data strobe
s_axil_ctrl_wvalid	in	1	Write data valid
s_axil_ctrl_wready	out	1	Write data ready
s_axil_ctrl_bresp	out	2	Write response status
s_axil_ctrl_bvalid	out	1	Write response valid
s_axil_ctrl_bready	in	1	Write response ready
s_axil_ctrl_araddr	in	AXIL_CTRL_ADDR_WIDTH	Read address
s_axil_ctrl_arprot	in	3	Read protect
s_axil_ctrl_arvalid	in	1	Read address valid
s_axil_ctrl_arready	out	1	Read address ready
s_axil_ctrl_rdata	out	AXIL_CTRL_DATA_WIDTH	Read response data
s_axil_ctrl_rresp	out	2	Read response status
s_axil_ctrl_rvalid	out	1	Read response valid
s_axil_ctrl_rready	in	1	Read response ready

`s_axil_app_ctrl`

AXI-Lite slave interface (application control). This interface is directly passed through to the application section.

Signal	Dir	Width	Description
s_axil_app_ctrl_awaddr	in	AXIL_APP_CTRL_ADDR_WIDTH	Write address
s_axil_app_ctrl_awprot	in	3	Write protect
s_axil_app_ctrl_awvalid	in	1	Write address valid
s_axil_app_ctrl_awready	out	1	Write address ready
s_axil_app_ctrl_wdata	in	AXIL_APP_CTRL_DATA_WIDTH	Write data
s_axil_app_ctrl_wstrb	in	AXIL_APP_CTRL_STRB_WIDTH	Write data strobe
s_axil_app_ctrl_wvalid	in	1	Write data valid
s_axil_app_ctrl_wready	out	1	Write data ready
s_axil_app_ctrl_bresp	out	2	Write response status
s_axil_app_ctrl_bvalid	out	1	Write response valid
s_axil_app_ctrl_bready	in	1	Write response ready
s_axil_app_ctrl_araddr	in	AXIL_APP_CTRL_ADDR_WIDTH	Read address
s_axil_app_ctrl_arprot	in	3	Read protect
s_axil_app_ctrl_arvalid	in	1	Read address valid
s_axil_app_ctrl_arready	out	1	Read address ready
s_axil_app_ctrl_rdata	out	AXIL_APP_CTRL_DATA_WIDTH	Read response data
s_axil_app_ctrl_rresp	out	2	Read response status
s_axil_app_ctrl_rvalid	out	1	Read response valid
s_axil_app_ctrl_rready	in	1	Read response ready

`m_axil_csr`

AXI-Lite master interface (passthrough for NIC control and status). This interface can be used to implement additional components in the main NIC control register space.

Signal	Dir	Width	Description
m_axil_csr_awaddr	in	AXIL_CSR_ADDR_WIDTH	Write address
m_axil_csr_awprot	in	3	Write protect
m_axil_csr_awvalid	in	1	Write address valid
m_axil_csr_awready	out	1	Write address ready
m_axil_csr_wdata	in	AXIL_CTRL_DATA_WIDTH	Write data
m_axil_csr_wstrb	in	AXIL_CTRL_STRB_WIDTH	Write data strobe
m_axil_csr_wvalid	in	1	Write data valid
m_axil_csr_wready	out	1	Write data ready
m_axil_csr_bresp	out	2	Write response status
m_axil_csr_bvalid	out	1	Write response valid
m_axil_csr_bready	in	1	Write response ready
m_axil_csr_araddr	in	AXIL_CTRL_ADDR_WIDTH	Read address
m_axil_csr_arprot	in	3	Read protect
m_axil_csr_arvalid	in	1	Read address valid
m_axil_csr_arready	out	1	Read address ready
m_axil_csr_rdata	out	AXIL_CTRL_DATA_WIDTH	Read response data
m_axil_csr_rresp	out	2	Read response status
m_axil_csr_rvalid	out	1	Read response valid
m_axil_csr_rready	in	1	Read response ready

ctrl_reg

Control register interface. This interface can be used to implement additional control registers and register blocks in the main NIC control register space.

Signal	Dir	Width	Description
ctrl_reg_wr_addr	out	AXIL_CSR_ADDR_WIDTH	Write address
ctrl_reg_wr_data	out	AXIL_CTRL_DATA_WIDTH	Write data
ctrl_reg_wr_strb	out	AXIL_CTRL_STRB_WIDTH	Write strobe
ctrl_reg_wr_en	out	1	Write enable
ctrl_reg_wr_wait	in	1	Write wait
ctrl_reg_wr_ack	in	1	Write acknowledge
ctrl_reg_rd_addr	out	AXIL_CSR_ADDR_WIDTH	Read address
ctrl_reg_rd_en	out	1	Read enable
ctrl_reg_rd_data	in	AXIL_CTRL_DATA_WIDTH	Read data
ctrl_reg_rd_wait	in	1	Read wait
ctrl_reg_rd_ack	in	1	Read acknowledge

m_axis_dma_read_desc

DMA read descriptor output

Signal	Dir	Width	Description
m_axis_dma_read_desc_dma_addr	out	DMA_ADDR_WIDTH	DMA address
m_axis_dma_read_desc_ram_sel	out	RAM_SEL_WIDTH	RAM select
m_axis_dma_read_desc_ram_addr	out	RAM_ADDR_WIDTH	RAM address
m_axis_dma_read_desc_len	out	DMA_LEN_WIDTH	Transfer length
m_axis_dma_read_desc_tag	out	DMA_TAG_WIDTH	Transfer tag
m_axis_dma_read_desc_valid	out	1	Request valid
m_axis_dma_read_desc_ready	in	1	Request ready

s_axis_dma_read_desc_status

DMA read descriptor status input

Signal	Dir	Width	Description
s_axis_dma_read_desc_status_tag	in	DMA_TAG_WIDTH	Status tag
s_axis_dma_read_desc_status_error	in	4	Status error code
s_axis_dma_read_desc_status_valid	in	1	Status valid

m_axis_dma_write_desc

DMA write descriptor output

Signal	Dir	Width	Description
m_axis_dma_write_desc_dma_addr	out	DMA_ADDR_WIDTH	DMA address
m_axis_dma_write_desc_ram_sel	out	RAM_SEL_WIDTH	RAM select
m_axis_dma_write_desc_ram_addr	out	RAM_ADDR_WIDTH	RAM address
m_axis_dma_write_desc_imm	out	DMA_IMM_WIDTH	Immediate
m_axis_dma_write_desc_imm_en	out	1	Immediate enable
m_axis_dma_write_desc_len	out	DMA_LEN_WIDTH	Transfer length
m_axis_dma_write_desc_tag	out	DMA_TAG_WIDTH	Transfer tag
m_axis_dma_write_desc_valid	out	1	Request valid
m_axis_dma_write_desc_ready	in	1	Request ready

s_axis_dma_write_desc_status

DMA write descriptor status input

Signal	Dir	Width	Description
s_axis_dma_write_desc_status_tag	in	DMA_TAG_WIDTH	Status tag
s_axis_dma_write_desc_status_error	in	4	Status error code
s_axis_dma_write_desc_status_valid	in	1	Status valid

dma_ram

DMA RAM interface

Signal	Dir	Width	Description
dma_ram_wr_cmd_sel	in	RAM_SEG_COUNT*RAM_SEL_WIDTH	Write command select
dma_ram_wr_cmd_be	in	RAM_SEG_COUNT*RAM_SEG_BE_WIDTH	Write command byte enable
dma_ram_wr_cmd_addr	in	RAM_SEG_COUNT*RAM_SEG_ADDR_WIDTH	Write command address
dma_ram_wr_cmd_data	in	RAM_SEG_COUNT*RAM_SEG_DATA_WIDTH	Write command data
dma_ram_wr_cmd_valid	in	RAM_SEG_COUNT	Write command valid
dma_ram_wr_cmd_ready	out	RAM_SEG_COUNT	Write command ready
dma_ram_wr_done	out	RAM_SEG_COUNT	Write done
dma_ram_rd_cmd_sel	in	RAM_SEG_COUNT*RAM_SEL_WIDTH	Read command select
dma_ram_rd_cmd_addr	in	RAM_SEG_COUNT*RAM_SEG_ADDR_WIDTH	Read command address
dma_ram_rd_cmd_valid	in	RAM_SEG_COUNT	Read command valid
dma_ram_rd_cmd_ready	out	RAM_SEG_COUNT	Read command ready
dma_ram_rd_resp_data	out	RAM_SEG_COUNT*RAM_SEG_DATA_WIDTH	Read response data
dma_ram_rd_resp_valid	out	RAM_SEG_COUNT	Read response valid
dma_ram_rd_resp_ready	in	RAM_SEG_COUNT	Read response ready

msi_irq

MSI request outputs

Signal	Dir	Width	Description
msi_irq	out	MSI_COUNT	Interrupt request

ptp

PTP clock connections.

Signal	Dir	Width	Description
ptp_clk	in	1	PTP clock
ptp_rst	in	1	PTP reset
ptp_sample_clk	in	1	PTP sample clock
ptp_pps	out	1	PTP pulse-per-second (synchronous to ptp_clk)
ptp_pps_str	out	1	PTP pulse-per-second (stretched) (synchronous to ptp_clk)
ptp_ts_96	out	PTP_TS_WIDTH	current PTP time (synchronous to ptp_clk)
ptp_ts_step	out	1	PTP clock step (synchronous to ptp_clk)
ptp_sync_pps	out	1	PTP pulse-per-second (synchronous to clk)
ptp_sync_ts_96	out	PTP_TS_WIDTH	current PTP time (synchronous to clk)
ptp_sync_ts_step	out	1	PTP clock step (synchronous to clk)
ptp_perout_locked	out	PTP_PEROUT_COUNT	PTP period output locked
ptp_perout_error	out	PTP_PEROUT_COUNT	PTP period output error
ptp_perout_pulse	out	PTP_PEROUT_COUNT	PTP period output pulse

tx_clk

Transmit clocks, one per port

Signal	Dir	Width	Description
tx_clk	in	PORT_COUNT	Transmit clock

tx_rst

Transmit resets, one per port

Signal	Dir	Width	Description
tx_rst	in	PORT_COUNT	Transmit reset

tx_ptp_ts

Reference PTP time for transmit timestamping synchronous to each transmit clock, one per port.

Signal	Dir	Width	Description
tx_ptp_ts_96	out	PORT_COUNT*PTP_TS_WIDTH	current PTP time
tx_ptp_ts_step	out	PORT_COUNT	PTP clock step

m_axis_tx

Streaming transmit data towards network, one AXI stream interface per port.

Signal	Dir	Width	Description
m_axis_tx_tdata	out	PORT_COUNT*AXIS_DATA_WIDTH	Streaming data
m_axis_tx_tkeep	out	PORT_COUNT*AXIS_KEEP_WIDTH	Byte enable
m_axis_tx_tvalid	out	PORT_COUNT	Data valid
m_axis_tx_tready	in	PORT_COUNT	Ready for data
m_axis_tx_tlast	out	PORT_COUNT	End of frame
m_axis_tx_tuser	out	PORT_COUNT*AXIS_TX_USER_WIDTH	Sideband data

`s_axis_tx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
TX_TAG_WIDTH:1	tx_tag	TX_TAG_WIDTH	Transmit tag

s_axis_tx_cpl

Transmit completion, one AXI stream interface per port.

Signal	Dir	Width	Description
s_axis_tx_cpl_ts	in	PORT_COUNT*PTP_TS_WIDTH	PTP timestamp
s_axis_tx_cpl_tag	in	PORT_COUNT*TX_TAG_WIDTH	Transmit tag
s_axis_tx_cpl_valid	in	PORT_COUNT	Transmit completion valid
s_axis_tx_cpl_ready	out	PORT_COUNT	Transmit completion ready

tx_status

Transmit link status inputs, one per port

Signal	Dir	Width	Description
tx_status	in	PORT_COUNT	Transmit link status

rx_clk

Receive clocks, one per port

Signal	Dir	Width	Description
rx_clk	in	PORT_COUNT	Receive clock

rx_rst

Receive resets, one per port

Signal	Dir	Width	Description
rx_rst	in	PORT_COUNT	Receive reset

rx_ptp_ts

Reference PTP time for receive timestamping synchronous to each receive clock, one per port. Synchronous to `rx_ptp_clk` if `PTP_SEPARATE_RX_CLOCK` is set.

Signal	Dir	Width	Description
rx_ptp_clk	in	PORT_COUNT	clock for PTP time
rx_ptp_rst	in	PORT_COUNT	reset for PTP time
rx_ptp_ts_96	out	PORT_COUNT*PTP_TS_WIDTH	current PTP time
rx_ptp_ts_step	out	PORT_COUNT	PTP clock step

s_axis_rx

Streaming receive data from network, one AXI stream interface per port.

Signal	Dir	Width	Description
s_axis_rx_tdata	in	PORT_COUNT*AXIS_DATA_WIDTH	Streaming data
s_axis_rx_tkeep	in	PORT_COUNT*AXIS_KEEP_WIDTH	Byte enable
s_axis_rx_tvalid	in	PORT_COUNT	Data valid
s_axis_rx_tready	out	PORT_COUNT	Ready for data
s_axis_rx_tlast	in	PORT_COUNT	End of frame
s_axis_rx_tuser	in	PORT_COUNT*AXIS_TX_USER_WIDTH	Sideband data

`s_axis_rx_tuser` bits, per port

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

`rx_status`

Receive link status inputs, one per port

Signal	Dir	Width	Description
rx_status	in	PORT_COUNT	Receive link status

`s_axis_stat`

Statistics increment input

Signal	Dir	Width	Description
s_axis_stat_tdata	in	STAT_INC_WIDTH	Statistic increment
s_axis_stat_tid	in	STAT_ID_WIDTH	Statistic ID
s_axis_stat_tvalid	in	1	Statistic valid
s_axis_stat_tready	out	1	Statistic ready

`app_gpio`

Application section GPIO

Signal	Dir	Width	Description
app_gpio_in	in	APP_GPIO_IN_WIDTH	GPIO inputs
app_gpio_out	out	APP_GPIO_OUT_WIDTH	GPIO outputs

`app_jtag`

Application section JTAG scan chain

Signal	Dir	Width	Description
app_jtag_tdi	in	1	JTAG TDI
app_jtag_tdo	out	1	JTAG TDO
app_jtag_tms	in	1	JTAG TMS
app_jtag_tck	in	1	JTAG TCK

8.7 `mqnic_core_axi`

`mqnic_core_axi` is the core integration-level module for mqnic for the AXI host interface. Wrapper around *mqnic_core* (page 59), adding the AXI DMA interface module.

`mqnic_core_axi` integrates the following modules:

- `dma_if_axi`: AXI DMA engine
- *mqnic_core* (page 59): core logic

8.7.1 Parameters

Only parameters implemented in the wrapper are described here, for the other parameters see *mqnic_core* (page 59).

AXI_DATA_WIDTH

AXI master interface data signal width, default `128`.

AXI_ADDR_WIDTH

AXI master interface address signal width, default `32`.

AXI_STRB_WIDTH

AXI master interface byte enable signal width, default `(AXI_DATA_WIDTH/8)`.

AXI_ID_WIDTH

AXI master interface ID signal width, default `8`.

AXI_DMA_MAX_BURST_LEN

AXI DMA maximum burst length, default `256`.

AXI_DMA_READ_USE_ID

Use ID field for AXI DMA reads, default `0`.

AXI_DMA_WRITE_USE_ID

Use ID field for AXI DMA writes, default `1`.

AXI_DMA_READ_OP_TABLE_SIZE

AXI read DMA operation table size, default `2**(AXI_ID_WIDTH)`.

AXI_DMA_WRITE_OP_TABLE_SIZE

AXI write DMA operation table size, default `2**(AXI_ID_WIDTH)`.

IRQ_COUNT

IRQ channel count, default `32`.

STAT_DMA_ENABLE

Enable DMA-related statistics, default `1`.

STAT_AXI_ENABLE

Enable AXI-related statistics, default `1`.

8.7.2 Ports

Only ports implemented in the wrapper are described here, for the other ports see *mqnic_core* (page 59).

m_axi

AXI master interface (DMA).

Signal	Dir	Width	Description
m_axi_awid	out	AXI_ID_WIDTH	Write ID
m_axi_awaddr	out	AXI_ADDR_WIDTH	Write address
m_axi_awlen	out	8	Write burst length
m_axi_awsize	out	3	Write burst size
m_axi_awburst	out	2	Write burst type
m_axi_awlock	out	1	Write lock
m_axi_awcache	out	4	Write cache

continues on next page

Table 8.1 – continued from previous page

Signal	Dir	Width	Description
m_axi_awprot	out	3	Write protect
m_axi_awvalid	out	1	Write valid
m_axi_awready	in	1	Write ready
m_axi_wdata	out	AXI_DATA_WIDTH	Write data data
m_axi_wstrb	out	AXI_STRB_WIDTH	Write data strobe
m_axi_wlast	out	1	Write data last
m_axi_wvalid	out	1	Write data valid
m_axi_wready	in	1	Write data ready
m_axi_bid	in	AXI_ID_WIDTH	Write response ID
m_axi_bresp	in	2	Write response status
m_axi_bvalid	in	1	Write response valid
m_axi_bready	out	1	Write response ready
m_axi_arid	out	AXI_ID_WIDTH	Read ID
m_axi_araddr	out	AXI_ADDR_WIDTH	Read address
m_axi_arlen	out	8	Read burst length
m_axi_arsize	out	3	Read burst size
m_axi_arburst	out	2	Read burst type
m_axi_arlock	out	1	Read lock
m_axi_arcache	out	4	Read cache
m_axi_arprot	out	3	Read protect
m_axi_arvalid	out	1	Read address valid
m_axi_arready	in	1	Read address ready
m_axi_rid	in	AXI_ID_WIDTH	Read response ID
m_axi_rdata	in	AXI_DATA_WIDTH	Read response data
m_axi_rresp	in	2	Read response status
m_axi_rlast	in	1	Read response last
m_axi_rvalid	in	1	Read response valid
m_axi_rready	out	1	Read response ready

msi_irq

Interrupt outputs

Signal	Dir	Width	Description
irq	out	IRQ_COUNT	Interrupt request

8.8 mqnic_core_pcie

`mqnic_core_pcie` is the core integration-level module for mqnic for the PCIe host interface. Wrapper around *mqnic_core* (page 59), adding PCIe DMA interface module and PCIe-AXI Lite masters for the NIC and application control BARs.

This module implements a generic PCIe host interface, which must be adapted to the target device with a wrapper. The available wrappers are:

- *mqnic_core_pcie_us* (page 79) for Xilinx Virtex 7, UltraScale, and UltraScale+
- *mqnic_core_pcie_s10* (page 77) for Intel Stratix 10 H-tile/L-tile

`mqnic_core_pcie` integrates the following modules:

- `dma_if_pcie`: PCIe DMA engine

- `pcie_axil_master`: AXI lite master module for control registers
- `stats_pcie_if`: statistics collection for PCIe TLP traffic
- `stats_dma_if_pcie`: statistics collection for PCIe DMA engine
- *mqnic_core* (page 59): core logic

8.8.1 Parameters

Only parameters implemented in the wrapper are described here, for the other parameters see *mqnic_core* (page 59).

TLP_SEG_COUNT
Number of segments in the TLP interfaces, default `1`.

TLP_SEG_DATA_WIDTH
TLP segment data width, default `256`.

TLP_SEG_STRB_WIDTH
TLP segment byte enable width, must be set to `TLP_SEG_DATA_WIDTH/32`.

TLP_SEG_HDR_WIDTH
TLP segment header width, must be `128`.

TX_SEQ_NUM_COUNT
Number of transmit sequence number inputs, default `1`.

TX_SEQ_NUM_WIDTH
Transmit sequence number width, default `5`.

TX_SEQ_NUM_ENABLE
Use transmit sequence numbers, default `0`.

PF_COUNT
PCIe PF count, default `1`.

VF_COUNT
PCIe VF count, default `0`.

F_COUNT
PCIe function count, must be `PF_COUNT+VF_COUNT`.

PCIE_TAG_COUNT
PCIe tag count, default `256`.

PCIE_DMA_READ_OP_TABLE_SIZE
PCIe read DMA operation table size, default `PCIE_TAG_COUNT`.

PCIE_DMA_READ_TX_LIMIT
PCIe read DMA transmit operation limit, default `2**TX_SEQ_NUM_WIDTH`.

PCIE_DMA_READ_TX_FC_ENABLE
Use transmit flow control credits in PCIe read DMA, default `0`.

PCIE_DMA_WRITE_OP_TABLE_SIZE
PCIe write DMA operation table size, default `2**TX_SEQ_NUM_WIDTH`.

PCIE_DMA_WRITE_TX_LIMIT

PCIe write DMA transmit operation limit, default `2**TX_SEQ_NUM_WIDTH`.

PCIE_DMA_WRITE_TX_FC_ENABLE

Use transmit flow control credits in PCIe write DMA, default `0`.

TLP_FORCE_64_BIT_ADDR

Force 64 bit address field for all TLPs, default `0`.

CHECK_BUS_NUMBER

Check bus number in received TLPs, default `1`.

MSI_COUNT

Number of MSI channels, default `32`.

STAT_DMA_ENABLE

Enable DMA-related statistics, default `1`.

STAT_PCIE_ENABLE

Enable PCIe-related statistics, default `1`.

8.8.2 Ports

Only ports implemented in the wrapper are described here, for the other ports see *mqnic_core* (page 59).

pcie_rx_req_tlp

TLP input (request to BAR)

Signal	Dir	Width	Description
pcie_rx_req_tlp_data	in	TLP_SEG_COUNT*TLP_SEG_DATA_WIDTH	TLP payload
pcie_rx_req_tlp_hdr	in	TLP_SEG_COUNT*TLP_SEG_HDR_WIDTH	TLP header
pcie_rx_req_tlp_bar_id	in	TLP_SEG_COUNT*3	BAR ID
pcie_rx_req_tlp_func_num	in	TLP_SEG_COUNT*8	Function
pcie_rx_req_tlp_valid	in	TLP_SEG_COUNT	Valid
pcie_rx_req_tlp_sop	in	TLP_SEG_COUNT	Start of packet
pcie_rx_req_tlp_eop	in	TLP_SEG_COUNT	End of packet
pcie_rx_req_tlp_ready	out	1	Ready

pcie_rx_cpl_tlp

TLP input (completion to DMA)

Signal	Dir	Width	Description
pcie_rx_cpl_tlp_data	in	TLP_SEG_COUNT*TLP_SEG_DATA_WIDTH	TLP payload
pcie_rx_cpl_tlp_hdr	in	TLP_SEG_COUNT*TLP_SEG_HDR_WIDTH	TLP header
pcie_rx_cpl_tlp_error	in	TLP_SEG_COUNT*4	Error
pcie_rx_cpl_tlp_valid	in	TLP_SEG_COUNT	Valid
pcie_rx_cpl_tlp_sop	in	TLP_SEG_COUNT	Start of packet
pcie_rx_cpl_tlp_eop	in	TLP_SEG_COUNT	End of packet
pcie_rx_cpl_tlp_ready	out	1	Ready

pcie_tx_rd_req_tlp

TLP output (read request from DMA)

Signal	Dir	Width	Description
pcie_tx_rd_req_tlp_hdr	out	TLP_SEG_COUNT*TLP_SEG_HDR_WIDTH	TLP header
pcie_tx_rd_req_tlp_seq	out	TLP_SEG_COUNT*TX_SEQ_NUM_WIDTH	TX seq num
pcie_tx_rd_req_tlp_valid	out	TLP_SEG_COUNT	Valid
pcie_tx_rd_req_tlp_sop	out	TLP_SEG_COUNT	Start of packet
pcie_tx_rd_req_tlp_eop	out	TLP_SEG_COUNT	End of packet
pcie_tx_rd_req_tlp_ready	in	1	Ready

s_axis_pcie_rd_req_tx_seq_num

Transmit sequence number input (DMA read request)

Signal	Dir	Width	Description
s_axis_pcie_rd_req_tx_seq_num	in	TX_SEQ_NUM_COUNT*TX_SEQ_NUM_WIDTH	TX seq num
s_axis_pcie_rd_req_tx_seq_num_valid	in	TX_SEQ_NUM_COUNT	Valid

pcie_tx_wr_req_tlp

TLP output (read request from DMA)

Signal	Dir	Width	Description
pcie_tx_wr_req_tlp_data	out	TLP_SEG_COUNT*TLP_SEG_DATA_WIDTH	TLP payload
pcie_tx_wr_req_tlp_strb	out	TLP_SEG_COUNT*TLP_SEG_STRB_WIDTH	TLP byte enable
pcie_tx_wr_req_tlp_hdr	out	TLP_SEG_COUNT*TLP_SEG_HDR_WIDTH	TLP header
pcie_tx_wr_req_tlp_seq	out	TLP_SEG_COUNT*TX_SEQ_NUM_WIDTH	TX seq num
pcie_tx_wr_req_tlp_valid	out	TLP_SEG_COUNT	Valid
pcie_tx_wr_req_tlp_sop	out	TLP_SEG_COUNT	Start of packet
pcie_tx_wr_req_tlp_eop	out	TLP_SEG_COUNT	End of packet
pcie_tx_wr_req_tlp_ready	in	1	Ready

s_axis_pcie_wr_req_tx_seq_num

Transmit sequence number input (DMA write request)

Signal	Dir	Width	Description
s_axis_pcie_wr_req_tx_seq_num	in	TX_SEQ_NUM_COUNT*TX_SEQ_NUM_WIDTH	TX seq num
s_axis_pcie_wr_req_tx_seq_num_valid	in	TX_SEQ_NUM_COUNT	Valid

pcie_tx_cpl_tlp

TLP output (completion from BAR)

Signal	Dir	Width	Description
pcie_tx_cpl_tlp_data	out	TLP_SEG_COUNT*TLP_SEG_DATA_WIDTH	TLP payload
pcie_tx_cpl_tlp_strb	out	TLP_SEG_COUNT*TLP_SEG_STRB_WIDTH	TLP byte enable
pcie_tx_cpl_tlp_hdr	out	TLP_SEG_COUNT*TLP_SEG_HDR_WIDTH	TLP header
pcie_tx_cpl_tlp_valid	out	TLP_SEG_COUNT	Valid
pcie_tx_cpl_tlp_sop	out	TLP_SEG_COUNT	Start of packet
pcie_tx_cpl_tlp_eop	out	TLP_SEG_COUNT	End of packet
pcie_tx_cpl_tlp_ready	in	1	Ready

pcie_tx_fc

Flow control credits

Signal	Dir	Width	Description
pcie_tx_fc_ph_av	in	8	Available posted header credits
pcie_tx_fc_pd_av	in	12	Available posted data credits
pcie_tx_fc_nph_av	in	8	Available non-posted header credits

config

Configuration inputs

Signal	Dir	Width	Description
bus_num	in	8	Bus number
ext_tag_enable	in	F_COUNT	Extended tag enable
max_read_request_size	in	F_COUNT*3	Max read request size
max_payload_size	in	F_COUNT*3	Max payload size

pcie_error

PCIe error outputs

Signal	Dir	Width	Description
pcie_error_cor	out	1	Correctable error
pcie_error_uncor	out	1	Uncorrectable error

msi_irq

MSI request outputs

Signal	Dir	Width	Description
msi_irq	out	MSI_COUNT	Interrupt request

8.9 mqnic_core_pcie_s10

`mqnic_core_pcie_s10` is the core integration-level module for mqnic for the PCIe host interface on Intel Stratix 10 GX, SX, MX, and TX series devices with H-tiles or L-tiles. Wrapper around *mqnic_core_pcie* (page 73), adding device-specific shims for the PCIe interface.

`mqnic_core_pcie_s10` integrates the following modules:

- `pcie_s10_if`: PCIe interface shim
- *mqnic_core_pcie* (page 73): core logic for PCI express

8.9.1 Parameters

Only parameters implemented in the wrapper are described here, for the other parameters see *mqnic_core_pcie* (page 73).

SEG_COUNT

TLP segment count, default 1.

SEG_DATA_WIDTH

TLP segment data signal width, default `256`.

SEG_EMPTY_WIDTH

TLP segment empty signal width, must be set to `$clog2(SEG_DATA_WIDTH/32)`.

TX_SEQ_NUM_WIDTH

Transmit sequence number width, default `6`.

TX_SEQ_NUM_ENABLE

Transmit sequence number enable, default `1`.

L_TILE

Tile select, `0` for H-tile, `1` for L-tile, default `0`.

8.9.2 Ports

Only ports implemented in the wrapper are described here, for the other ports see *mqnic_core_pcie* (page 73).

rx_st

H-Tile/L-Tile RX AVST interface

Signal	Dir	Width	Description
rx_st_data	in	SEG_COUNT*SEG_DATA_WIDTH	TLP data
rx_st_empty	in	SEG_COUNT*SEG_EMPTY_WIDTH	Empty
rx_st_sop	in	SEG_COUNT	Start of packet
rx_st_eop	in	SEG_COUNT	End of packet
rx_st_valid	in	SEG_COUNT	Valid
rx_st_ready	out	1	Ready
rx_st_vf_active	in	SEG_COUNT	VF active
rx_st_func_num	in	SEG_COUNT*2	Function number
rx_st_vf_num	in	SEG_COUNT*11	VF number
rx_st_bar_range	in	SEG_COUNT*3	BAR range

tx_st

H-Tile/L-Tile TX AVST interface

Signal	Dir	Width	Description
tx_st_data	out	SEG_COUNT*SEG_DATA_WIDTH	TLP data
tx_st_sop	out	SEG_COUNT	Start of packet
tx_st_eop	out	SEG_COUNT	End of packet
tx_st_valid	out	SEG_COUNT	Valid
tx_st_ready	in	1	Ready
tx_st_err	out	SEG_COUNT	Error

tx_fc

H-Tile/L-Tile TX flow control

Signal	Dir	Width	Description
tx_ph_cdts	in	8	Posted header credits
tx_pd_cdts	in	12	Posted data credits
tx_nph_cdts	in	8	Non-posted header credits
tx_npd_cdts	in	12	Non-posted data credits
tx_cplh_cdts	in	8	Completion header credits
tx_cpld_cdts	in	12	Completion data credits
tx_hdr_cdts_consumed	in	SEG_COUNT	Header credits consumed
tx_data_cdts_consumed	in	SEG_COUNT	Data credits consumed
tx_cdts_type	in	SEG_COUNT*2	Credit type
tx_cdts_data_value	in	SEG_COUNT*1	Credit data value

app_msi

H-Tile/L-Tile MSI interrupt interface

Signal	Dir	Width	Description
app_msi_req	out	1	MSI request
app_msi_ack	in	1	MSI acknowledge
app_msi_tc	out	3	MSI traffic class
app_msi_num	out	5	MSI number
app_msi_func_num	out	2	Function number

tl_cfg

H-Tile/L-Tile configuration interface

Signal	Dir	Width	Description
tl_cfg_ctl	in	32	Config data
tl_cfg_add	in	5	Config address
tl_cfg_func	in	2	Config function

8.10 mqnic_core_pcie_us

`mqnic_core_pcie_us` is the core integration-level module for mqnic for the PCIe host interface on Xilinx Virtex 7, UltraScale, and UltraScale+ series devices. Wrapper around *mqnic_core_pcie* (page 73), adding device-specific shims for the PCIe interface.

`mqnic_core_pcie_us` integrates the following modules:

- `pcie_us_if`: PCIe interface shim
- *mqnic_core_pcie* (page 73): core logic for PCI express

8.10.1 Parameters

Only parameters implemented in the wrapper are described here, for the other parameters see *mqnic_core_pcie* (page 73).

AXIS_PCIE_DATA_WIDTH

PCIe AXI stream `tdata` signal width, default `256`.

AXIS_PCIE_KEEP_WIDTH

PCIe AXI stream `tkeep` signal width, must be set to `(AXIS_PCIE_DATA_WIDTH/32)`.

AXIS_PCIE_RC_USER_WIDTH

PCIe AXI stream RC `tuser` signal width, default `AXIS_PCIE_DATA_WIDTH < 512 ? 75 : 161`.

AXIS_PCIE_RQ_USER_WIDTH

PCIe AXI stream RQ `tuser` signal width, default `AXIS_PCIE_DATA_WIDTH < 512 ? 62 : 137`.

AXIS_PCIE_CQ_USER_WIDTH

PCIe AXI stream CQ `tuser` signal width, default `AXIS_PCIE_DATA_WIDTH < 512 ? 85 : 183`.

AXIS_PCIE_CC_USER_WIDTH

PCIe AXI stream CC `tuser` signal width, default `AXIS_PCIE_DATA_WIDTH < 512 ? 33 : 81`.

RQ_SEQ_NUM_WIDTH

PCIe RQ sequence number width, default `AXIS_PCIE_RQ_USER_WIDTH == 60 ? 4 : 6`.

8.10.2 Ports

Only ports implemented in the wrapper are described here, for the other ports see *mqnic_core_pcie* (page 73).

s_axis_rc

AXI input (RC)

Signal	Dir	Width	Description
s_axis_rc_tdata	in	AXIS_PCIE_DATA_WIDTH	TLP data
s_axis_rc_tkeep	in	AXIS_PCIE_KEEP_WIDTH	Byte enable
s_axis_rc_tvalid	in	1	Valid
s_axis_rc_tready	out	1	Ready
s_axis_rc_tlast	in	1	End of frame
s_axis_rc_tuser	in	AXIS_PCIE_RC_USER_WIDTH	Sideband data

m_axis_rq

AXI output (RQ)

Signal	Dir	Width	Description
m_axis_rq_tdata	out	AXIS_PCIE_DATA_WIDTH	TLP data
m_axis_rq_tkeep	out	AXIS_PCIE_KEEP_WIDTH	Byte enable
m_axis_rq_tvalid	out	1	Valid
m_axis_rq_tready	in	1	Ready
m_axis_rq_tlast	out	1	End of frame
m_axis_rq_tuser	out	AXIS_PCIE_RQ_USER_WIDTH	Sideband data

s_axis_cq

AXI input (CQ)

Signal	Dir	Width	Description
s_axis_cq_tdata	in	AXIS_PCIE_DATA_WIDTH	TLP data
s_axis_cq_tkeep	in	AXIS_PCIE_KEEP_WIDTH	Byte enable
s_axis_cq_tvalid	in	1	Valid
s_axis_cq_tready	out	1	Ready
s_axis_cq_tlast	in	1	End of frame
s_axis_cq_tuser	in	AXIS_PCIE_CQ_USER_WIDTH	Sideband data

m_axis_cc

AXI output (CC)

Signal	Dir	Width	Description
m_axis_cc_tdata	out	AXIS_PCIE_DATA_WIDTH	TLP data
m_axis_cc_tkeep	out	AXIS_PCIE_KEEP_WIDTH	Byte enable
m_axis_cc_tvalid	out	1	Valid
m_axis_cc_tready	in	1	Ready
m_axis_cc_tlast	out	1	End of frame
m_axis_cc_tuser	out	AXIS_PCIE_CC_USER_WIDTH	Sideband data

s_axis_rq_seq_num

Transmit sequence number input

Signal	Dir	Width	Description
s_axis_rq_seq_num_0	in	RQ_SEQ_NUM_WIDTH	Sequence number
s_axis_rq_seq_num_valid_0	in	1	Valid
s_axis_rq_seq_num_1	in	RQ_SEQ_NUM_WIDTH	Sequence number
s_axis_rq_seq_num_valid_1	in	1	Valid

cfg_fc_ph

Flow control

Signal	Dir	Width	Description
cfg_fc_ph	in	8	Posted header credits
cfg_fc_pd	in	12	Posted data credits
cfg_fc_nph	in	8	Non-posted header credits
cfg_fc_npd	in	12	Non-posted data credits
cfg_fc_cplh	in	8	Completion header credits
cfg_fc_cpld	in	12	Completion data credits
cfg_fc_sel	out	3	Credit select

cfg_max_read_req

Configuration inputs

Signal	Dir	Width	Description
cfg_max_read_req	in	F_COUNT*3	Max read request
cfg_max_payload	in	F_COUNT*3	Max payload

cfg_mgmt_addr

Configuration interface

Signal	Dir	Width	Description
cfg_mgmt_addr	out	10	Address
cfg_mgmt_function_number	out	8	Function number
cfg_mgmt_write	out	1	Write enable
cfg_mgmt_write_data	out	32	Write data
cfg_mgmt_byte_enable	out	4	Byte enable
cfg_mgmt_read	out	1	Read enable
cfg_mgmt_read_data	in	32	Read data
cfg_mgmt_read_write_done	in	1	Write done

cfg_interrupt_msi_enable

Interrupt interface

Signal	Dir	Width	Description
cfg_interrupt_msi_enable	in	4	MSI enable
cfg_interrupt_msi_vf_enable	in	8	VF enable
cfg_interrupt_msi_mmenable	in	12	MM enable
cfg_interrupt_msi_mask_update	in	1	Mask update
cfg_interrupt_msi_data	in	32	Data
cfg_interrupt_msi_select	out	4	Select
cfg_interrupt_msi_int	out	32	Interrupt request
cfg_interrupt_msi_pending_status	out	32	Pending status
cfg_interrupt_msi_pending_status_data_enable	out	1	Pending status enable
cfg_interrupt_msi_pending_status_function_num	out	4	Pending status function
cfg_interrupt_msi_sent	in	1	MSI sent
cfg_interrupt_msi_fail	in	1	MSI fail
cfg_interrupt_msi_attr	out	3	MSI attr
cfg_interrupt_msi_tph_present	out	1	TPH present
cfg_interrupt_msi_tph_type	out	2	TPH type
cfg_interrupt_msi_tph_st_tag	out	9	TPH ST tag
cfg_interrupt_msi_function_number	out	4	MSI function number

status_error_cor

PCIe error outputs

Signal	Dir	Width	Description
status_error_cor	out	1	Correctable error
status_error_uncor	out	1	Uncorrectable error

8.11 mqnic_egress

`mqnic_egress` implements egress processing on the transmit side. This consists of:

1. Transmit checksum offloading

`mqnic_egress` integrates the following modules:

- *tx_checksum* (page 97): transmit checksum offloading

8.12 mqnic_ingress

`mqnic_ingress` implements ingress processing on the receive path. This consists of:

1. Receive checksum offloading
2. RSS flow hashing

`mqnic_ingress` integrates the following modules:

- *rx_checksum* (page 97): receive checksum offloading
- *rx_hash* (page 97): RSS flow hash computation

8.13 mqnic_interface

`mqnic_interface` implements one NIC interface, including the queue management logic, descriptor, completion, and event handling, transmit scheduler, and the transmit and receive datapaths.

`mqnic_interface` integrates the following modules:

- *queue_manager* (page 94): transmit and receive queues
- *cpl_queue_manager* (page 42): transmit and receive completion queues, event queues
- *desc_fetch* (page 43): descriptor fetch
- *cpl_write* (page 42): completion write
- *mqnic_tx_scheduler_block* (page 91): transmit scheduler
- *mqnic_interface_rx* (page 83): receive datapath
- *mqnic_interface_tx* (page 83): transmit datapath

8.14 mqnic_interface_rx

`mqnic_interface_rx` implements the host-side receive datapath.

`mqnic_interface_rx` integrates the following modules:

- *rx_engine* (page 97): receive engine
- *mqnic_ingress* (page 83): ingress datapath
- `dma_client_axis_sink`: internal DMA engine

8.15 mqnic_interface_tx

`mqnic_interface_tx` implements the host-side transmit datapath.

`mqnic_interface_tx` integrates the following modules:

- *tx_engine* (page 97): transmit engine
- `dma_client_axis_source`: internal DMA engine
- *mqnic_egress* (page 82): egress datapath

8.16 mqnic_l2_egress

`mqnic_l2_egress` contains layer 2 egress processing components, and operates synchronous to the MAC TX clock. Currently, this module is a placeholder, passing through streaming data without modification.

8.16.1 Parameters

`AXIS_DATA_WIDTH`

Streaming interface `tdata` signal width, default `512`.

`AXIS_KEEP_WIDTH`

Streaming interface `tkeep` signal width, must be set to `AXIS_DATA_WIDTH/8`.

`AXIS_USER_WIDTH`

Streaming interface `tuser` signal width, default `1`.

8.16.2 Ports

`clk`

Logic clock.

Signal	Dir	Width	Description
clk	in	1	Logic clock

`rst`

Logic reset, active high

Signal	Dir	Width	Description
rst	in	1	Logic reset, active high

`s_axis`

Streaming transmit data from host

Signal	Dir	Width	Description
s_axis_tdata	in	AXIS_DATA_WIDTH	Streaming data
s_axis_tkeep	in	AXIS_KEEP_WIDTH	Byte enable
s_axis_tvalid	in		Data valid
s_axis_tready	out		Ready for data
s_axis_tlast	in		End of frame
s_axis_tuser	in	AXIS_USER_WIDTH	Sideband data

`s_axis_tuser` bits

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

m_axis

Streaming transmit data towards network

Signal	Dir	Width	Description
m_axis_tdata	out	AXIS_DATA_WIDTH	Streaming data
m_axis_tkeep	out	AXIS_KEEP_WIDTH	Byte enable
m_axis_tvalid	out		Data valid
m_axis_tready	in		Ready for data
m_axis_tlast	out		End of frame
m_axis_tuser	out	AXIS_USER_WIDTH	Sideband data

`m_axis_tuser` bits

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

8.17 mqnic_l2_ingress

`mqnic_l2_ingress` contains layer 2 ingress processing components, and operates synchronous to the MAC RX clock. Currently, this module is a placeholder, passing through streaming data without modification.

8.17.1 Parameters

AXIS_DATA_WIDTH

Streaming interface `tdata` signal width, default `512`.

AXIS_KEEP_WIDTH

Streaming interface `tkeep` signal width, must be set to `AXIS_DATA_WIDTH/8`.

AXIS_USER_WIDTH

Streaming interface `tuser` signal width, default `1`.

AXIS_USE_READY

Use `tready` signal, default `0`. If set, logic will exert backpressure with `tready` instead of dropping packets when RX FIFOs are full.

8.17.2 Ports

clk

Logic clock.

Signal	Dir	Width	Description
clk	in	1	Logic clock

rst

Logic reset, active high

Signal	Dir	Width	Description
rst	in	1	Logic reset, active high

s_axis

Streaming receive data from network

Signal	Dir	Width	Description
s_axis_tdata	in	AXIS_DATA_WIDTH	Streaming data
s_axis_tkeep	in	AXIS_KEEP_WIDTH	Byte enable
s_axis_tvalid	in		Data valid
s_axis_tready	out		Ready for data
s_axis_tlast	in		End of frame
s_axis_tuser	in	AXIS_USER_WIDTH	Sideband data

`s_axis_tuser` bits

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

m_axis

Streaming receive data towards host

Signal	Dir	Width	Description
m_axis_tdata	out	AXIS_DATA_WIDTH	Streaming data
m_axis_tkeep	out	AXIS_KEEP_WIDTH	Byte enable
m_axis_tvalid	out		Data valid
m_axis_tready	in		Ready for data
m_axis_tlast	out		End of frame
m_axis_tuser	out	AXIS_USER_WIDTH	Sideband data

`m_axis_tuser` bits

Bit	Name	Width	Description
0	bad_frame	1	Invalid frame
PTP_TS_WIDTH:1	ptp_ts	PTP_TS_WIDTH	PTP timestamp

8.18 mqnic_ptp

`mqnic_ptp` implements the PTP subsystem, including PTP clock and period output modules.

`mqnic_ptp` integrates the following modules:

- *mqnic_ptp_clock* (page 88): PTP clock (*PTP hardware clock register block* (page 118))
- *mqnic_ptp_perout* (page 90): PTP period output (*PTP period output register block* (page 120))

8.18.1 Parameters

PTP_PERIOD_NS_WIDTH

PTP period ns field width, default `4`.

PTP_OFFSET_NS_WIDTH

PTP offset ns field width, default `32`.

PTP_FNS_WIDTH

PTP fractional ns field width, default `32`.

PTP_PERIOD_NS

PTP nominal period, ns portion `4'd4`.

PTP_PERIOD_FNS

PTP nominal period, fractional ns portion `32'd0`.

PTP_PEROUT_ENABLE

Enable PTP period output module, default `0`.

PTP_PEROUT_COUNT

Number of PTP period output channels, default `1`.

REG_ADDR_WIDTH

Register interface address width, default `7+(PTP_PEROUT_ENABLE ? $clog2((PTP_PEROUT_COUNT+1)/2) + 1 : 0)`.

REG_DATA_WIDTH

Register interface data width, default `32`.

REG_STRB_WIDTH

Register interface byte enable width, must be set to `(REG_DATA_WIDTH/8)`.

RB_BASE_ADDR

Base address of control register block, default `0`.

RB_NEXT_PTR

Address of next control register block, default `0`.

8.18.2 Ports

clk

Logic clock.

Signal	Dir	Width	Description
clk	in	1	Logic clock

rst

Logic reset, active high

Signal	Dir	Width	Description
rst	in	1	Logic reset, active high

reg

Control register interface

Signal	Dir	Width	Description
reg_wr_addr	in	REG_ADDR_WIDTH	Write address
reg_wr_data	in	REG_DATA_WIDTH	Write data
reg_wr_strb	in	REG_STRB_WIDTH	Write byte enable
reg_wr_en	in	1	Write enable
reg_wr_wait	out	1	Write wait
reg_wr_ack	out	1	Write acknowledge
reg_rd_addr	in	REG_ADDR_WIDTH	Read address
reg_rd_en	in	1	Read enable
reg_rd_data	out	REG_DATA_WIDTH	Read data
reg_rd_wait	out	1	Read wait
reg_rd_ack	out	1	Read acknowledge

ptp

PTP signals

Signal	Dir	Width	Description
ptp_pps	out	1	Pulse-per-second
ptp_ts_96	out	96	PTP timestamp
ptp_ts_step	out	1	PTP timestamp step
ptp_perout_locked	out	PTP_PEROUT_COUNT	Period output channel locked
ptp_perout_error	out	PTP_PEROUT_COUNT	Period output channel error
ptp_perout_pulse	out	PTP_PEROUT_COUNT	Period output channel pulse

8.19 mqnic_ptp_clock

`mqnic_ptp_clock` implements the PTP hardware clock. It wraps `ptp_clock` and provides a register interface for control, see *PTP hardware clock register block* (page 118).

8.19.1 Parameters

PTP_PERIOD_NS_WIDTH

PTP period ns field width, default `4`.

PTP_OFFSET_NS_WIDTH

PTP offset ns field width, default `32`.

PTP_FNS_WIDTH

PTP fractional ns field width, default `32`.

PTP_PERIOD_NS

PTP nominal period, ns portion `4'd4`.

PTP_PERIOD_FNS

PTP nominal period, fractional ns portion `32'd0`.

PTP_PEROUT_ENABLE

Enable PTP period output module, default `0`.

PTP_PEROUT_COUNT

Number of PTP period output channels, default `1`.

REG_ADDR_WIDTH

Register interface address width, default `7`.

REG_DATA_WIDTH

Register interface data width, default `32`.

REG_STRB_WIDTH

Register interface byte enable width, must be set to `(REG_DATA_WIDTH/8)`.

RB_BASE_ADDR

Base address of control register block, default `0`.

RB_NEXT_PTR

Address of next control register block, default `0`.

8.19.2 Ports

clk

Logic clock.

Signal	Dir	Width	Description
clk	in	1	Logic clock

rst

Logic reset, active high

Signal	Dir	Width	Description
rst	in	1	Logic reset, active high

reg

Control register interface

Signal	Dir	Width	Description
reg_wr_addr	in	REG_ADDR_WIDTH	Write address
reg_wr_data	in	REG_DATA_WIDTH	Write data
reg_wr_strb	in	REG_STRB_WIDTH	Write byte enable
reg_wr_en	in	1	Write enable
reg_wr_wait	out	1	Write wait
reg_wr_ack	out	1	Write acknowledge
reg_rd_addr	in	REG_ADDR_WIDTH	Read address
reg_rd_en	in	1	Read enable
reg_rd_data	out	REG_DATA_WIDTH	Read data
reg_rd_wait	out	1	Read wait
reg_rd_ack	out	1	Read acknowledge

ptp

PTP signals

Signal	Dir	Width	Description
ptp_pps	out	1	Pulse-per-second
ptp_ts_96	out	96	PTP timestamp
ptp_ts_step	out	1	PTP timestamp step

8.20 mqnic_ptp_perout

`mqnic_ptp_perout` implements the PTP period output functionality. It wraps `ptp_perout` and provides a register interface for control, see *PTP period output register block* (page 120).

8.20.1 Parameters

REG_ADDR_WIDTH

Register interface address width, default `6`.

REG_DATA_WIDTH

Register interface data width, default `32`.

REG_STRB_WIDTH

Register interface byte enable width, must be set to `(REG_DATA_WIDTH/8)`.

RB_BASE_ADDR

Base address of control register block, default `0`.

RB_NEXT_PTR

Address of next control register block, default `0`.

8.20.2 Ports

clk

Logic clock.

Signal	Dir	Width	Description
clk	in	1	Logic clock

rst

Logic reset, active high

Signal	Dir	Width	Description
rst	in	1	Logic reset, active high

reg

Control register interface

Signal	Dir	Width	Description
reg_wr_addr	in	REG_ADDR_WIDTH	Write address
reg_wr_data	in	REG_DATA_WIDTH	Write data
reg_wr_strb	in	REG_STRB_WIDTH	Write byte enable
reg_wr_en	in	1	Write enable
reg_wr_wait	out	1	Write wait
reg_wr_ack	out	1	Write acknowledge
reg_rd_addr	in	REG_ADDR_WIDTH	Read address
reg_rd_en	in	1	Read enable
reg_rd_data	out	REG_DATA_WIDTH	Read data
reg_rd_wait	out	1	Read wait
reg_rd_ack	out	1	Read acknowledge

ptp

PTP signals

Signal	Dir	Width	Description
ptp_ts_96	in	96	PTP timestamp
ptp_ts_step	in	1	PTP timestamp step
ptp_perout_locked	out	1	Period output locked
ptp_perout_error	out	1	Period output error
ptp_perout_pulse	out	1	Period output pulse

8.21 mqnic_tx_scheduler_block

`mqnic_tx_scheduler_block` is the top-level block for the transmit scheduler. It is instantiated in *mqnic_interface* (page 83). This is a pluggable module, intended to be replaced by a customized implementation via the build system. See .... for more details.

Two variations are provided:

- `mqnic_tx_scheduler_block_rr`: round-robin transmit scheduler (*tx_scheduler_rr* (page 98))
- `mqnic_tx_scheduler_block_rr_tdma`: round-robin transmit scheduler (*tx_scheduler_rr* (page 98)) with TDMA scheduler controller

8.21.1 Parameters

PORTS

Number of ports, default `1`.

INDEX

Scheduler index, default `0`.

REG_ADDR_WIDTH

Width of control register interface address in bits, default `16`.

REG_DATA_WIDTH

Width of control register interface data in bits, default `32`.

REG_STRB_WIDTH

Width of control register interface strb, must be set to `(REG_DATA_WIDTH/8)`.

RB_BASE_ADDR

Register block base address, default `0`.

RB_NEXT_PTR

Register block next pointer, default `0`.

AXIL_DATA_WIDTH

Width of AXI lite data bus in bits, default `32`.

AXIL_ADDR_WIDTH

Width of AXI lite address bus in bits, default `16`.

AXIL_STRB_WIDTH

Width of AXI lite wstrb (width of data bus in words), must be set to `AXIL_DATA_WIDTH/8`.

AXIL_OFFSET

Offset to AXI lite interface, default `0`.

LEN_WIDTH

Length field width, default `16`.

REQ_TAG_WIDTH

Transmit request tag field width, default `8`.

OP_TABLE_SIZE

Number of outstanding operations, default `16`.

QUEUE_INDEX_WIDTH

Queue index width, default `6`.

PIPELINE

Pipeline setting, default `3`.

TDMA_INDEX_WIDTH

Scheduler TDMA index width, default `8`.

PTP_TS_WIDTH

PTP timestamp width, default `96`.

AXIS_TX_DEST_WIDTH

AXI stream tdest signal width, default `$clog2(PORTS)+4`.

MAX_TX_SIZE

Max transmit packet size, default `2048`.

8.21.2 Ports

clk

Logic clock. Most interfaces are synchronous to this clock.

Signal	Dir	Width	Description
clk	in	1	Logic clock

rst

Logic reset, active high

Signal	Dir	Width	Description
rst	in	1	Logic reset, active high

ctrl_reg

Control register interface

Signal	Dir	Width	Description
ctrl_reg_wr_addr	in	REG_ADDR_WIDTH	Write address
ctrl_reg_wr_data	in	REG_DATA_WIDTH	Write data
ctrl_reg_wr_strb	in	REG_STRB_WIDTH	Write byte enable
ctrl_reg_wr_en	in	1	Write enable
ctrl_reg_wr_wait	out	1	Write wait
ctrl_reg_wr_ack	out	1	Write acknowledge
ctrl_reg_rd_addr	in	REG_ADDR_WIDTH	Read address
ctrl_reg_rd_en	in	1	Read enable
ctrl_reg_rd_data	out	REG_DATA_WIDTH	Read data
ctrl_reg_rd_wait	out	1	Read wait
ctrl_reg_rd_ack	out	1	Read acknowledge

s_axil

AXI-Lite slave interface. This interface provides access to memory-mapped per-queue control registers.

Signal	Dir	Width	Description
s_axil_awaddr	in	AXIL_ADDR_WIDTH	Write address
s_axil_awprot	in	3	Write protect
s_axil_awvalid	in	1	Write address valid
s_axil_awready	out	1	Write address ready
s_axil_wdata	in	AXIL_DATA_WIDTH	Write data
s_axil_wstrb	in	AXIL_STRB_WIDTH	Write data strobe
s_axil_wvalid	in	1	Write data valid
s_axil_wready	out	1	Write data ready
s_axil_bresp	out	2	Write response status
s_axil_bvalid	out	1	Write response valid
s_axil_bready	in	1	Write response ready
s_axil_araddr	in	AXIL_ADDR_WIDTH	Read address
s_axil_arprot	in	3	Read protect
s_axil_arvalid	in	1	Read address valid
s_axil_arready	out	1	Read address ready
s_axil_rdata	out	AXIL_DATA_WIDTH	Read response data
s_axil_rresp	out	2	Read response status
s_axil_rvalid	out	1	Read response valid
s_axil_rready	in	1	Read response ready

m_axis_tx_req

Transmit request output, for transmit requests to the transmit engine.

Signal	Dir	Width	Description
m_axis_tx_req_queue	out	QUEUE_INDEX_WIDTH	Queue index
m_axis_tx_req_tag	out	REQ_TAG_WIDTH	Tag
m_axis_tx_req_dest	out	AXIS_TX_DEST_WIDTH	Destination port and TC
m_axis_tx_req_valid	out	1	Valid
m_axis_tx_req_ready	in	1	Ready

s_axis_tx_req_status

Transmit request status input, for responses from the transmit engine.

Signal	Dir	Width	Description
s_axis_tx_req_status_len	in	LEN_WIDTH	Packet length
s_axis_tx_req_status_tag	in	REQ_TAG_WIDTH	Tag
s_axis_tx_req_status_valid	in	1	Valid

s_axis_doorbell

Doorbell input, for enqueue notifications from the transmit queue manager.

Signal	Dir	Width	Description
s_axis_doorbell_queue	in	QUEUE_INDEX_WIDTH	Queue index
s_axis_doorbell_valid	in	1	Valid

ptp_ts

PTP time input from PTP clock

Signal	Dir	Width	Description
ptp_ts_96	in	PTP_TS_WIDTH	PTP time
ptp_ts_step	in	1	PTP clock step

config

Configuration signals

Signal	Dir	Width	Description
mtu	in	LEN_WIDTH	MTU

8.22 queue_manager

`queue_manager` implements the queue management logic for the transmit and receive queues. It stores host to device queue state in block RAM or ultra RAM.

8.22.1 Operation

Communication of packet data between the Corundum NIC and the driver is mediated via descriptor and completion queues. Descriptor queues form the host-to-NIC communications channel, carrying information about where individual packets are stored in system memory. Completion queues form the NIC-to-host communications channel, carrying information about completed operations and associated metadata. The descriptor and completion queues are implemented as ring buffers that reside in DMA-accessible system memory, while the NIC hardware maintains the necessary queue state information. This state information consists of a pointer to the DMA address of the ring buffer, the size of

the ring buffer, the producer and consumer pointers, and a reference to the associated completion queue. The required state for each queue fits into 128 bits.

The queue management logic for the Corundum NIC must be able to efficiently store and manage the state for thousands of queues. This means that the queue state must be completely stored in block RAM (BRAM) or ultra RAM (URAM) on the FPGA. Since a 128 bit RAM is required and URAM blocks are 72x4096, storing the state for 4096 queues requires only 2 URAM instances. Utilizing URAM instances enables scaling the queue management logic to handle at least 32,768 queues per interface.

In order to support high throughput, the NIC must be able to process multiple descriptors in parallel. Therefore, the queue management logic must track multiple in-progress operations, reporting updated queue pointers to the driver as the operations are completed. The state required to track in-process operations is much smaller than the state required to describe the queue state itself. Therefore the in-process operation state is stored in flip-flops and distributed RAM.

The NIC design uses two queue manager modules: `queue_manager` is used to manage host-to-NIC descriptor queues, while `cpl_queue_manager` is used to manage NIC-to-host completion queues. The modules are similar except for a few minor differences in terms of pointer handling, fill handling, and doorbell/event generation. Because of the similarities, this section will discuss only the operation of the `queue_manager` module.

The BRAM or URAM array used to store the queue state information requires several cycles of latency for each read operation, so the `queue_manager` is built with a pipelined architecture to facilitate multiple concurrent operations. The pipeline supports four different operations: register read, register write, dequeue/enqueue request, and dequeue/enqueue commit. Register-access operations over an AXI lite interface enable the driver to initialize the queue state and provide pointers to the allocated host memory as well as access the producer and consumer pointers during normal operation.

A block diagram of the queue manager module is shown in Fig. 8.2. The BRAM or URAM array used to store the queue state information requires several cycles of latency for each read operation, so the `queue_manager` is built with a pipelined architecture to facilitate multiple concurrent operations. The pipeline supports four different operations: register read, register write, dequeue/enqueue request, and dequeue/enqueue commit. Register-access operations over an AXI lite interface enable the driver to initialize the queue state and provide pointers to the allocated host memory as well as access the producer and consumer pointers during normal operation.

Each queue has three pointers associated with it, as shown in Fig. 8.3—the producer pointer, the host-facing consumer pointer, and the shadow consumer pointer. The driver has control over the producer pointer and can read the host-facing consumer pointer. Entries between the consumer pointer and the producer pointer are under the control of the NIC and must not be modified by the driver. The driver enqueues a descriptor by writing it into the ring buffer at the index indicated by the producer pointer, issuing a memory barrier, then incrementing the producer pointer in the queue manager. The NIC dequeues descriptors by reading them out of the descriptor ring via DMA and incrementing the consumer pointer. The host-facing consumer pointer must not be incremented until the descriptor read operation completes, so the queue manager maintains an internal shadow consumer pointer to keep track of read operations that have started in addition to the host-facing pointer that is updated as the read operations are completed.

The dequeue request operation on the queue manager pipeline initiates a dequeue operation on a queue. If the target queue is disabled or empty, the operation is rejected with an *empty* or *error* status. Otherwise, the shadow consumer pointer is incremented and the physical address of the queue element is returned, along with the queue element index and an operation tag. Operations on any combination of queues can be initiated until the operation table is full. The dequeue request input is stalled when the table is full. As the read operations complete, the dequeue operations are committed to free the operation table entry and update the host-facing consumer pointer. Operations can be committed in any order, simply setting the commit flag in the operation table, but the operation table entries will be freed and host-facing consumer pointer will be updated in-order to ensure descriptors being processed are not modified by the driver.

The operation table tracks in-process queue operations that have yet to be committed. Entries in the table consist of an active flag, a commit flag, the queue index, and the index of the next element in the queue. The queue state also contains a pointer to the most recent entry for that queue in the operation table. During an enqueue operation, the operation table is checked to see if there are any outstanding operations on that queue. If so, the consumer pointer for the most recent operation is incremented and stored in the new operation table entry. Otherwise, the current consumer pointer is incremented. When a dequeue commit request is received, the commit bit is set for the corresponding entry.

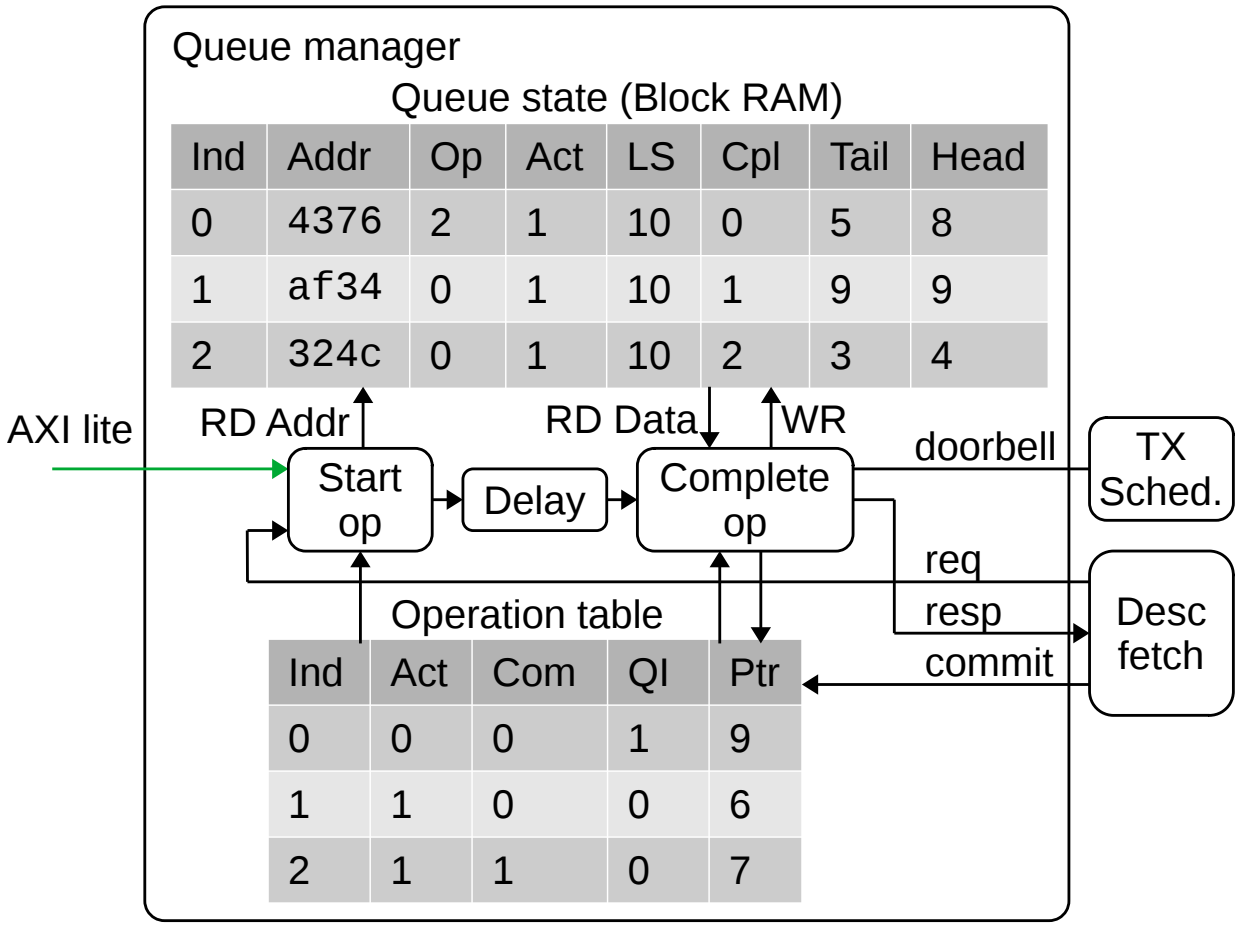


Fig. 8.2: Block diagram of the queue manager module, showing the queue state RAM and operation table. Ind = index, Addr = DMA address, Op = index in operation table, Act = active, LS = log base 2 of queue size, Cpl = completion queue index, Tail = tail or consumer pointer, Head = head or producer pointer, Com = committed; QI = queue index; Ptr = new queue pointer

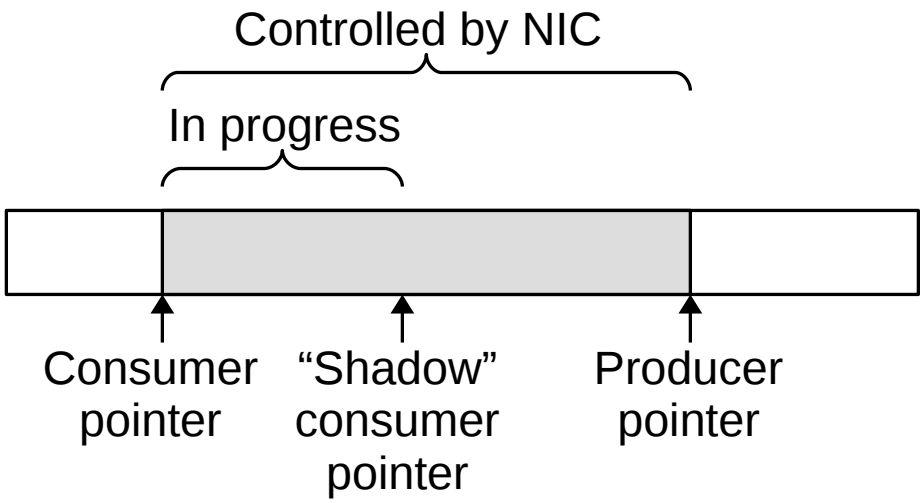


Fig. 8.3: Queue pointers on software ring buffers.

The entries are then committed in-order, updating the host-facing consumer pointer with the pointer from the operation table and clearing the active bit in the operation table entry.

Both the queue manager and completion queue manager modules generate notifications during enqueue operations. In a queue manager, when the driver updates a producer pointer on an enabled queue, the module issues a doorbell event that is passed to the transmit schedulers for the associated ports. Similarly, completion queue managers generate events on hardware enqueue operations, which are passed to the event subsystem and ultimately generate interrupts. To reduce the number of events and interrupts, completion queues also have an *armed* status. An armed completion queue will generate a single event, disarming itself in the process. The driver must re-arm the queue after handling the event.

8.23 rx_checksum

`rx_checksum` implements the receive checksum offloading support. It computes 16 bit checksum of Ethernet frame payload to aid in IP checksum offloading by the host network stack.

8.24 rx_engine

`rx_engine` manages receive datapath operations including descriptor dequeue and fetch via DMA, packet reception, data writeback via DMA, and completion enqueue and writeback via DMA. It also handles PTP timestamps for inclusion in completion records.

8.25 rx_hash

`rx_hash` implements flow hashing on the receive path. It extracts IP addresses and ports from packet headers and computes a 32-bit Toeplitz flow hash.

8.26 tx_checksum

`tx_checksum` implements the transmit checksum offloading support. It computes 16 bit checksum of frame data with specified start offset, then inserts computed checksum at the position specified by the host network stack.

8.27 tx_engine

`tx_engine` manages transmit datapath operations including descriptor dequeue and fetch via DMA, packet data fetch via DMA, packet transmission, and completion enqueue and writeback via DMA. It also handles PTP timestamps for inclusion in completion records.

8.28 tx_scheduler_rr

8.28.1 Operation

The default transmit scheduler used in the Corundum NIC is a simple round-robin scheduler implemented in the `tx_scheduler_rr` module. The scheduler sends commands to the transmit engine to initiate transmit operations out of the NIC transmit queues. The round-robin scheduler contains basic queue state for all queues, a FIFO to store currently-active queues and enforce the round-robin schedule, and an operation table to track in-process transmit operations.

Similar to the queue management logic, the round-robin transmit scheduler also stores queue state information in BRAM or URAM on the FPGA so that it can scale to support a large number of queues. The transmit scheduler also uses a processing pipeline to hide the memory access latency.

The transmit scheduler module has four main interfaces: an AXI lite register interface and three streaming interfaces. The AXI lite interface permits the driver to change scheduler parameters and enable/disable queues. The first streaming interface provides doorbell events from the queue management logic when the driver enqueues packets for transmission. The second streaming interface carries transmit commands generated by the scheduler to the transmit engine. Each command consists of a queue index to transmit from, along with a tag for tracking in-process operations. The final streaming interface returns transmit operation status information back to the scheduler. The status information informs the scheduler of the length of the transmitted packet, or if the transmit operation failed due to an empty or disabled queue.

The transmit scheduler module can be extended or replaced to implement arbitrary scheduling algorithms. This enables Corundum to be used as a platform to evaluate experimental scheduling algorithms. It is also possible to provide additional inputs to the transmit scheduler module, including feedback from the receive path, which can be used to implement new protocols and congestion control techniques. Connecting the scheduler to the PTP hardware clock can be used to support TDMA, which can be used to implement circuit-switched architectures.

The structure of the transmit scheduler logic is similar to the queue management logic in that it stores queue state in BRAM or URAM and uses a processing pipeline. However there are a number of significant differences. First, the scheduler logic is designed so that the scheduler does not stall when a queue is empty and a subsequent dequeue operation fails. Second, the scheduler contains a FIFO to enforce the round-robin schedule. The use of this FIFO requires an explicit reset routine to make the internal state (namely the scheduled flag bits) consistent after a reset. Third, the scheduler also contains logic to track the active state of each queue based on incoming doorbell requests and dequeue failures.

A block diagram of the transmit scheduler module is shown in Fig. 8.4. The transmit scheduler is built around a *scheduled queue* FIFO. This FIFO stores the indices of the currently-scheduled queues. An active queue is one that is presumed to have at least one packet available for transmission, an enabled queue is one that has been enabled for transmission, and a scheduled queue is one that has an entry in the scheduler FIFO. A queue will be scheduled (marked as scheduled and inserted into the FIFO) if it is both active and enabled. A queue will be descheduled when it reaches the front of the schedule FIFO, but is not enabled or not active. Queue enable states are controlled via three different enable bits per queue: queue enable, global enable, and schedule enable. The queue enable and global enable bits are writable via AXI lite, while the schedule enable bit is controlled from the scheduler control module via an internal interface. A queue is enabled when the queue enable bit and either the global enable or schedule enable bits are set. Queues become active when doorbell events are received, and queues become inactive when a transmit request fails due to an empty queue.

Tracking the queue active states must be done carefully for several reasons. First, the driver can update the producer pointer after enqueuing more than one packet, so the number of generated doorbell events does not necessarily correspond to the number of packets that were enqueued. Second, because the queues are shared among all ports on the same interface, multiple ports can attempt to send packets from the same queue, and the port transmit schedulers have no visibility into what the other schedulers are doing. Therefore, the most reliable method for determining that a queue is empty is to try sending from it, and flagging the failure. Note that the cost of an error is much higher when the queue is active than when the queue is empty. Attempting to send from an empty queue costs a few clock cycles and

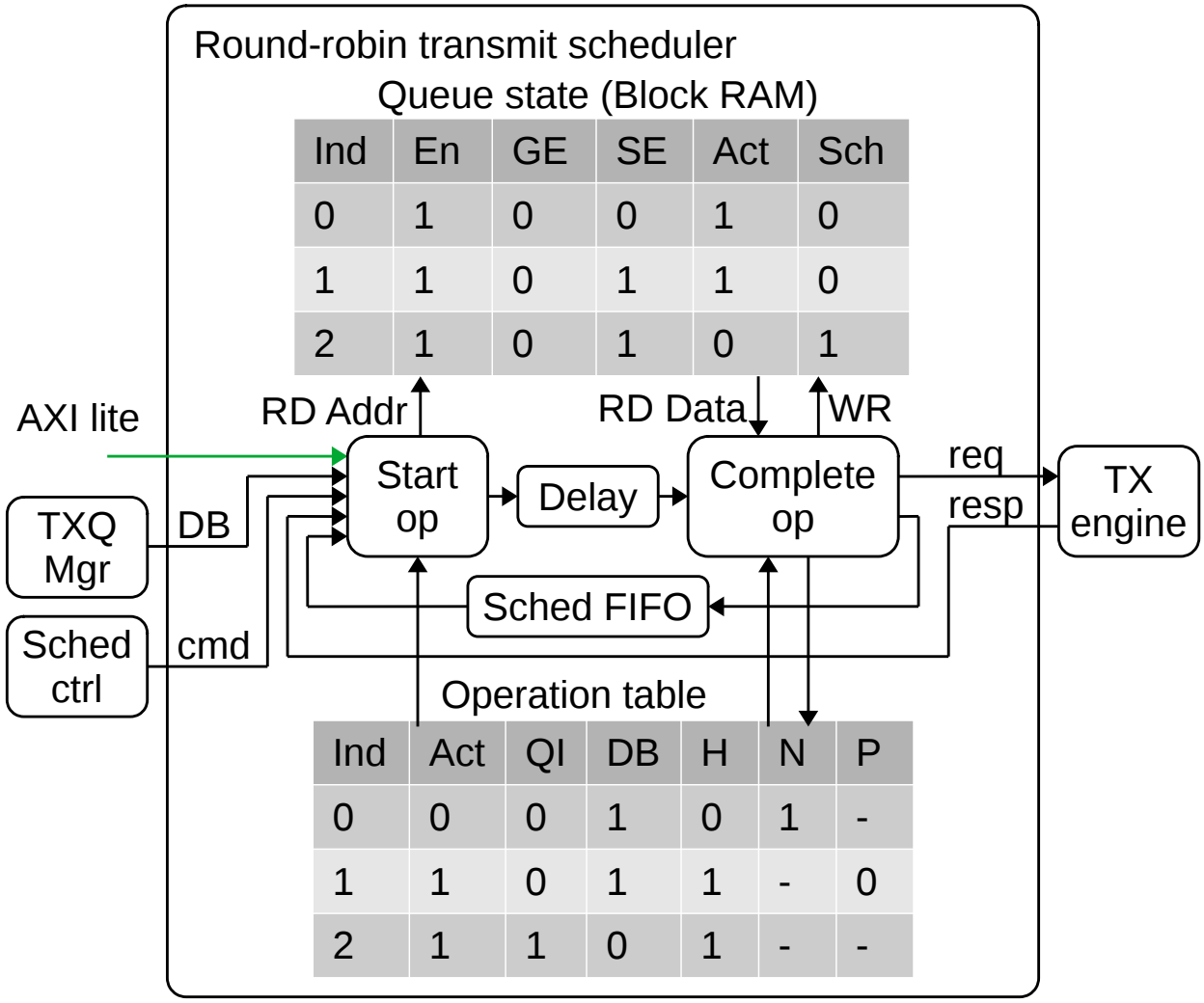


Fig. 8.4: Block diagram of the transmit scheduler module, showing queue state RAM and operation table. Ind = index, En = queue enable, GE = global enable, SE = schedule enable, Act = active, Sch = scheduled, QI = queue index, DB = doorbell, H = head, N = next, P = previous

temporarily occupies a few slots in corresponding operation tables. However, assuming a queue is empty when it is not will result in packets getting stuck in the queue. Fixing this stuck queue will not occur until the OS sends another packet on that queue and triggers another doorbell. Therefore, it is imperative to properly track doorbell events during transmit operations, as it is possible for a doorbell event to arrive after a dequeue attempt has failed, but before the failed transmit status arrives at the transmit scheduler module.

The pipeline in the transmit scheduler supports seven different operations: initialize, register read, register write, handle doorbell, transmit complete, scheduler control, and transmit request. The initialize operation is used to ensure the scheduler state is consistent after a reset. Register access operations over an AXI lite interface enable the driver to read all of the per-queue state and set the queue enable and global enable bits. The pipeline also handles incoming doorbell requests from the transmit queue manager module as well as queue enable/disable requests from the scheduler control module. Finally, the transmit request and transmit complete operations are used to generate transmit requests and handle the necessary queue state updates when the transmit operations complete.

Queues can become scheduled based on a register write that enables an active queue, a doorbell that activates an enabled queue, a scheduler operation that enables an active queue, and a transmit completion on an enabled queue that is either successful or has the doorbell bit set in the operation table. Queues can only be descheduled when the queue index advances to the front of the scheduler FIFO. If this occurs when the queue is both active and enabled, then the queue can be rescheduled and a transmit request generated. When the transmit operation completes, the transmit status response will be temporarily stored in a small FIFO and then processed by the pipeline to update the corresponding operation table entry and, if necessary, reschedule the queue.

The operation table tracks in-process transmit operations. Entries in the table consist of an active flag, the queue index, a doorbell flag, a head flag, a next pointer, and a previous pointer. The next and previous pointers form a linked list, enabling entries to be removed in any order while preserving the doorbell flag in the table. This prevents doorbells from getting 'lost' and the queue being mistakenly marked as inactive. A separate linked list is formed for each queue with active transmit operations. The operation table is implemented in such a way that it fits in distributed RAM.

8.28.2 Parameters

AXIL_DATA_WIDTH

Width of AXI lite data bus in bits, default `32`.

AXIL_ADDR_WIDTH

Width of AXI lite address bus in bits, default `16`.

AXIL_STRB_WIDTH

Width of AXI lite wstrb (width of data bus in words), must be set to `AXIL_DATA_WIDTH/8`.

LEN_WIDTH

Length field width, default `16`.

REQ_TAG_WIDTH

Transmit request tag field width, default `8`.

OP_TABLE_SIZE

Number of outstanding operations, default `16`.

QUEUE_INDEX_WIDTH

Queue index width, default `6`.

PIPELINE

Pipeline setting, default `3`.

8.28.3 Ports

clk

Logic clock. Most interfaces are synchronous to this clock.

Signal	Dir	Width	Description
clk	in	1	Logic clock

rst

Logic reset, active high

Signal	Dir	Width	Description
rst	in	1	Logic reset, active high

m_axis_tx_req

Transmit request output, for transmit requests to the transmit engine.

Signal	Dir	Width	Description
m_axis_tx_req_queue	out	QUEUE_INDEX_WIDTH	Queue index
m_axis_tx_req_tag	out	REQ_TAG_WIDTH	Tag
m_axis_tx_req_dest	out	AXIS_TX_DEST_WIDTH	Destination port and TC
m_axis_tx_req_valid	out	1	Valid
m_axis_tx_req_ready	in	1	Ready

s_axis_tx_req_status

Transmit request status input, for responses from the transmit engine.

Signal	Dir	Width	Description
s_axis_tx_req_status_len	in	LEN_WIDTH	Packet length
s_axis_tx_req_status_tag	in	REQ_TAG_WIDTH	Tag
s_axis_tx_req_status_valid	in	1	Valid

s_axis_doorbell

Doorbell input, for enqueue notifications from the transmit queue manager.

Signal	Dir	Width	Description
s_axis_doorbell_queue	in	QUEUE_INDEX_WIDTH	Queue index
s_axis_doorbell_valid	in	1	Valid

s_axis_sched_ctrl

Scheduler control input, to permit user logic to dynamically enable/disable queues.

Signal	Dir	Width	Description
s_axis_sched_ctrl_queue	in	QUEUE_INDEX_WIDTH	Queue index
s_axis_sched_ctrl_enable	in	1	Queue enable
s_axis_sched_ctrl_valid	in	1	Valid
s_axis_sched_ctrl_ready	out	1	Ready

s_axil

AXI-Lite slave interface. This interface provides access to memory-mapped per-queue control registers.

Signal	Dir	Width	Description
s_axil_awaddr	in	AXIL_ADDR_WIDTH	Write address
s_axil_awprot	in	3	Write protect
s_axil_awvalid	in	1	Write address valid
s_axil_awready	out	1	Write address ready
s_axil_wdata	in	AXIL_DATA_WIDTH	Write data
s_axil_wstrb	in	AXIL_STRB_WIDTH	Write data strobe
s_axil_wvalid	in	1	Write data valid
s_axil_wready	out	1	Write data ready
s_axil_bresp	out	2	Write response status
s_axil_bvalid	out	1	Write response valid
s_axil_bready	in	1	Write response ready
s_axil_araddr	in	AXIL_ADDR_WIDTH	Read address
s_axil_arprot	in	3	Read protect
s_axil_arvalid	in	1	Read address valid
s_axil_arready	out	1	Read address ready
s_axil_rdata	out	AXIL_DATA_WIDTH	Read response data
s_axil_rresp	out	2	Read response status
s_axil_rvalid	out	1	Read response valid
s_axil_rready	in	1	Read response ready

`control`

Control and status signals

Signal	Dir	Width	Description
enable	in	enable	Enable
active	out	enable	Active

CHAPTER

NINE

REGISTER BLOCKS

The NIC register space is constructed from a linked list of register blocks. Each block starts with a header that contains type, version, and next header fields. Blocks must be DWORD aligned in the register space. All fields must be naturally aligned. All pointers in the register blocks are relative to the start of the region. The list is terminated with a next pointer of 0x00000000. See Table 9.1 for a list of all currently-defined register blocks.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO -
RBB+0x04	Version	Major	Minor	Patch	Meta	RO -
RBB+0x08	Next pointer	Pointer to next register block				RO -

Type

The type field consists of a vendor ID in the upper 16 bits, and the sub type in the lower 16 bits. Vendor ID 0x0000 is used for all standard register blocks used by Corundum. See Table 9.1 for a list of all currently-defined register blocks.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Vendor ID		Type		RO -

Version

The version field consists of four fields, major, minor, patch, and meta. Version numbers must be changed when backwards-incompatible changes are made to register blocks. See Table 9.1 for a list of all currently-defined register blocks.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x04	Major	Minor	Patch	Meta	RO -

Next pointer

The next pointer field contains a block-relative offset to the start of the header of the next register block in the chain. A next pointer of 0x00000000 indicates the end of the chain.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x08	Pointer to next register block				RO -

Table 9.1: List of all currently-defined register blocks

Type	Version	Block
0x00000000	-	*Null register block* (page 117)
0xFFFFFFFF	0x00000100	*Firmware ID register block* (page 112)
0x0000C000	0x00000100	*Interface register block* (page 114)
0x0000C001	0x00000400	*Interface control register block* (page 115)
0x0000C002	0x00000200	port
0x0000C003	0x00000200	port_ctrl
0x0000C004	0x00000300	*Scheduler block register block* (page 123)
0x0000C005	0x00000200	application
0x0000C006	0x00000100	stats
0x0000C007	0x00000100	IRQ config
0x0000C010	0x00000100	*Event queue manager register block* (page 106)
0x0000C020	0x00000100	*Transmit queue manager register block* (page 122)
0x0000C021	0x00000100	*Receive queue manager register block* (page 121)
0x0000C030	0x00000100	*Transmit completion queue manager register block* (page 108)
0x0000C031	0x00000100	*Receive completion queue manager register block* (page 107)
0x0000C040	0x00000100	*Round-robin scheduler register block* (page 125)
0x0000C050	0x00000100	*TDMA scheduler controller register block* (page 124)
0x0000C060	0x00000100	*TDMA scheduler register block* (page 127)
0x0000C080	0x00000100	*PTP hardware clock register block* (page 118)
0x0000C081	0x00000100	*PTP period output register block* (page 120)
0x0000C090	0x00000100	RX queue map
0x0000C100	0x00000100	*GPIO register block* (page 113)
0x0000C110	0x00000100	*I2C register block* (page 114)
0x0000C120	0x00000200	*SPI flash register block* (page 111)
0x0000C121	0x00000200	*BPI flash register block* (page 110)
0x0000C140	0x00000100	*Alveo BMC register block* (page 104)
0x0000C141	0x00000100	*Gecko BMC register block* (page 105)
0x0000C150	0x00000100	*DRP register block* (page 109)

9.1 Alveo BMC register block

The Alveo BMC register block has a header with type 0x0000C140, version 0x00000100, and contains control registers for the Xilinx Alveo CMS IP[21].

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C140
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Address	Address				RW 0x00000000
RBB+0x10	Data	Data				RW 0x00000000

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Address

The address field controls the address bus to the CMS IP core. Writing to this register triggers a read of the corresponding address via the AXI-lite interface to the CMS IP.

[21] https://www.xilinx.com/products/intellectual-property/cms-subsystem.html

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Address				RW 0x00000000

Data

The data field controls the data bus to the CMS IP core. Writing to this register triggers a write to the address specified by the address register via the AXI-lite interface to the CMS IP.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Data				RW 0x00000000

9.2 Gecko BMC register block

The Gecko BMC register block has a header with type 0x0000C141, version 0x00000100, and contains control registers for the Silicom Gecko BMC.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C141
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Status	Status		Read data		RO 0x00000000
RBB+0x10	Data	Write data				RW 0x00000000
RBB+0x14	Command	Command				RW 0x00000000

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Status

The status field provides status information and the read data from the BMC.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Status		Read data		RO 0x00000000

Bit	Function
16	Done
18	Timeout
19	Idle

Data

The data field provides the write data to the BMC.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Write data				RW 0x00000000

Command

The command field provides the command to the BMC. Writing to the command field triggers an SPI transfer to the BMC.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Command				RW 0x00000000

9.3 Event queue manager register block

The event queue manager register block has a header with type 0x0000C010, version 0x00000100, and indicates the location of the event queue manager registers and number of event queues.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C010
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Offset	Offset to queue manager				RO -
RBB+0x10	Count	Queue count				RO -
RBB+0x14	Stride	Queue control register stride				RO 0x00000020

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Offset

The offset field contains the offset to the start of the event queue manager region, relative to the start of the current region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Offset to queue manager				RO -

Count

The count field contains the number of queues.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Queue count				RO -

Stride

The stride field contains the size of the control registers associated with each queue.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Queue control register stride				RO 0x00000020

9.3.1 Event queue manager CSRs

Each queue has several associated control registers, detailed in this table:

Address	Field	31..24	23..16	15..8	7..0	Reset value
Base+0x00	Base address L	Ring base address (lower 32)				RW -
Base+0x04	Base address H	Ring base address (upper 32)				RW -
Base+0x08	Control 1	Active			Size	RW -
Base+0x0C	Control 2	Arm		Int index		RW -
Base+0x10	Head pointer			Head pointer		RW -
Base+0x14	Tail pointer			Tail pointer		RW -

9.4 Receive completion queue manager register block

The receive completion queue manager register block has a header with type 0x0000C031, version 0x00000100, and indicates the location of the receive completion queue manager registers and number of completion queues.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C031
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Offset	Offset to queue manager				RO -
RBB+0x10	Count	Queue count				RO -
RBB+0x14	Stride	Queue control register stride				RO 0x00000020

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Offset

The offset field contains the offset to the start of the receive completion queue manager region, relative to the start of the current region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Offset to queue manager				RO -

Count

The count field contains the number of queues.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Queue count				RO -

Stride

The stride field contains the size of the control registers associated with each queue.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Queue control register stride				RO 0x00000020

9.4.1 Completion queue manager CSRs

Each queue has several associated control registers, detailed in this table:

Address	Field	31..24	23..16	15..8	7..0	Reset value
Base+0x00	Base address L	Ring base address (lower 32)				RW -
Base+0x04	Base address H	Ring base address (upper 32)				RW -
Base+0x08	Control 1	Active			Size	RW -
Base+0x0C	Control 2	Arm		Event index		RW -
Base+0x10	Head pointer			Head pointer		RW -
Base+0x14	Tail pointer			Tail pointer		RW -

9.5 Transmit completion queue manager register block

The transmit completion queue manager register block has a header with type 0x0000C030, version 0x00000100, and indicates the location of the transmit completion queue manager registers and number of completion queues.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C030
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Offset	Offset to queue manager				RO -
RBB+0x10	Count	Queue count				RO -
RBB+0x14	Stride	Queue control register stride				RO 0x00000020

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Offset

The offset field contains the offset to the start of the transmit completion queue manager region, relative to the start of the current region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Offset to queue manager				RO -

Count

The count field contains the number of queues.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Queue count				RO -

Stride

The stride field contains the size of the control registers associated with each queue.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Queue control register stride				RO 0x00000020

9.5.1 Completion queue manager CSRs

Each queue has several associated control registers, detailed in this table:

Address	Field	31..24	23..16	15..8	7..0	Reset value
Base+0x00	Base address L	Ring base address (lower 32)				RW -
Base+0x04	Base address H	Ring base address (upper 32)				RW -
Base+0x08	Control 1	Active			Size	RW -
Base+0x0C	Control 2	Arm		Event index		RW -
Base+0x10	Head pointer			Head pointer		RW -
Base+0x14	Tail pointer			Tail pointer		RW -

9.6 DRP register block

The DRP register block has a header with type 0x0000C150, version 0x00000100, and contains control registers for a Xilinx dynamic reconfiguration port (DRP).

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C150
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	DRP info	DRP info				RO -
RBB+0x10	Control	Control				RW 0x00000000
RBB+0x14	Address	Address				RW 0x00000000
RBB+0x18	Write data	Write data				RW 0x00000000
RBB+0x1C	Read data	Read data				RO 0x00000000

See *Register blocks* (page 103) for definitions of the standard register block header fields.

DRP info

The DRP info field contains identifying information about the component(s) accessible via the DRP interface.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	DRP info				RO -

Control

The control field is used to trigger read and write operations on the DRP interface.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Control				RW 0x00000000

Bit	Function
0	Enable
1	Write
8	Busy

To issue a read operation, set the address register and then write 0x00000001 to the control register. Wait for the enable and busy bits to self-clear, then read the data from the read data register.

To issue a write operation, set the address register and write data register appropriately, then write 0x00000003 to the control register. Wait for the enable and busy bits to self-clear.

Address

The address field controls the address for DRP operations. This address is directly presented on the DRP interface.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Address				RW 0x00000000

Write data

The write data field contains the data used for DRP write operations.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x18	Write data				RW 0x00000000

Read data

The read data field contains the data returned by DRP read operations.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x1C	Read data				RO 0x00000000

9.7 BPI flash register block

The BPI flash register block has a header with type 0x0000C121, version 0x00000200, and contains control registers for a BPI flash chip.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C121
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Format	Format				RO -
RBB+0x10	Address	Address				RW 0x00000000
RBB+0x14	Data	Data				RW 0x00000000
RBB+0x18	Control		REGION	DQ_OE	CTRL	RW 0x0000000F

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Format

The format field contains information about the type and layout of the flash memory. Bits 3:0 carry the number of segments. Bits 7:4 carry the index of the default segment that carries the main FPGA configuration. Bits 11:8 carry the index of the segment that contains a fallback FPGA configuration that is loaded if the configuration in the default segment fails to load. Bits 31:12 contain the size of the first segment in increments of 4096 bytes, for two-segment configurations with an uneven split. This field can be set to zero for an even split computed from the flash device size.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Format				RO -

bits	Configuration
3:0	Segment count
7:4	Default segment
11:8	Fallback segment
31:12	First segment size

Address

The address field controls the address bus to the flash chip.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Address				RW 0x00000000

Data

The data field controls the data bus to the flash chip.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Data				RW 0x00000000

`Control`

The control field contains registers to drive all of the other flash control lines, as well as registers for output enables.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x18		REGION	DQ_OE	CTRL	RW 0x0000000F

Bit	Function
0	CE_N
1	OE_N
2	WE_N
3	ADV_N
8	DQ_OE
16	REGION_OE

9.8 SPI flash register block

The SPI flash register block has a header with type 0x0000C120, version 0x00000200, and contains control registers for up to two SPI or QSPI flash chips.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C120
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Format	Format				RO -
RBB+0x10	Control 0		CS/CLK	OE	D	RW 0x00000000
RBB+0x14	Control 1		CS/CLK	OE	D	RW 0x00000000

See *Register blocks* (page 103) for definitions of the standard register block header fields.

`Format`

The format field contains information about the type and layout of the flash memory. Bits 3:0 carry the number of segments. Bits 7:4 carry the index of the default segment that carries the main FPGA configuration. Bits 11:8 carry the index of the segment that contains a fallback FPGA configuration that is loaded if the configuration in the default segment fails to load. Bits 31:12 contain the size of the first segment in increments of 4096 bytes, for two-segment configurations with an uneven split. This field can be set to zero for an even split computed from the flash device size.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Format				RO -

bits	Configuration
3:0	Segment count
7:4	Default segment
11:8	Fallback segment
31:12	First segment size

Control 0 and 1

The control 0 and 1 fields each control one SPI/QSPI flash interface. The second interface is only used in dual QSPI mode.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10		CS/CLK	OE	D	RW 0x00000000
RBB+0x14		CS/CLK	OE	D	RW 0x00000000

Bit	Function
0	D0
1	D1
2	D2
3	D3
8	OE for D0
9	OE for D1
10	OE for D2
11	OE for D3
16	CLK
17	CS_N

9.9 Firmware ID register block

The firmware ID register block has a header with type 0xFFFFFFFF, version 0x00000100, and carries several pieces of information related to the firmware version and build.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0xFFFFFFFF
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	FPGA ID	JTAG ID				RO -
RBB+0x10	FW ID	Vendor ID		Firmware ID		RO -
RBB+0x14	FW Version	Major	Minor	Patch	Meta	RO -
RBB+0x18	Board ID	Vendor ID		Board ID		RO -
RBB+0x1C	Board Version	Major	Minor	Patch	Meta	RO -
RBB+0x20	Build date	Build date				RO -
RBB+0x24	Git hash	Commit hash				RO -
RBB+0x28	Release info	Release info				RO -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

FPGA ID

The FPGA ID field contains the JTAG ID of the target device.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	JTAG ID				RO -

Firmware ID

The firmware ID field consists of a vendor ID in the upper 16 bits, and the firmware ID in the lower 16 bits.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Vendor ID		Firmware ID		RO -

Firmware version

The firmware version field consists of four fields, major, minor, patch, and meta.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Major	Minor	Patch	Meta	RO -

Board ID

The board ID field consists of a vendor ID in the upper 16 bits, and the board ID in the lower 16 bits.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x18	Vendor ID		Board ID		RO -

Board version

The board version field consists of four fields, major, minor, patch, and meta.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x1C	Major	Minor	Patch	Meta	RO -

Build date

The build date field contains the Unix timestamp of the start of the build as an unsigned 32-bit integer.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x20	Build date				RO -

Git hash

The git hash field contains the upper 32 bits of the git commit hash.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x24	Commit hash				RO -

Release info

The release info field is reserved for additional release information.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x28	Release info				RO -

9.10 GPIO register block

The GPIO register block has a header with type 0x0000C100, version 0x00000100, and contains GPIO control registers.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C100
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	GPIO in	GPIO in				RO -
RBB+0x10	GPIO out	GPIO out				RW -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

GPIO in

The GPIO in field reads the input signal states.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	GPIO in				RO -

GPIO out

The GPIO out field controls the output signal states.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	GPIO out				RW -

9.11 I2C register block

The I2C register block has a header with type 0x0000C110, version 0x00000100, and contains registers to control an I2C interface.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C110
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Control	Mux control		SDA	SCL	RW 0x00000303

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Control

The control field has bits to control SCL, SDA, and any associated multiplexers/switches.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Mux control		SDA	SCL	RW 0x00000303

Bit	Function
0	SCL in
1	SCL out
8	SDA in
9	SDA out

9.12 Interface register block

The interface register block has a header with type 0x0000C000, version 0x00000100, and indicates the number of interfaces present and where they are located in the control register space.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C000
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Offset	Offset to first interface				RO -
RBB+0x10	Count	Interface count				RO -
RBB+0x14	Stride	Interface stride				RO -
RBB+0x18	CSR offset	Interface CSR offset				RO -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Offset

The offset field contains the offset to the start of the first interface region, relative to the start of the current region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Offset to first interface				RO -

Count

The count field contains the number of interfaces.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Interface count				RO -

Stride

The stride field contains the size of the region for each interface.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Interface stride				RO -

CSR offset

The CSR offset field contains the offset to the head of the register block chain inside of each interface's region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x18	Interface CSR offset				RO -

9.13 Interface control register block

The interface control register block has a header with type 0x0000C001, version 0x00000400, and contains several interface-level control registers.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C001
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000300
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Features	Interface feature bits				RO -
RBB+0x10	Port count	Port count				RO -
RBB+0x14	Sched count	Scheduler block count				RO -
RBB+0x18	-	-				RO -
RBB+0x1C	-	-				RO -
RBB+0x20	Max TX MTU	Max TX MTU				RO -
RBB+0x24	Max RX MTU	Max RX MTU				RO -
RBB+0x28	TX MTU	TX MTU				RW -
RBB+0x2C	RX MTU	RX MTU				RW -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Features

The features field contains all of the interface-level feature bits, indicating the state of various optional features that can be enabled via Verilog parameters during synthesis.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Interface feature bits				RO -

Currently implemented feature bits:

Bit	Feature
0	RSS
4	PTP timestamping
8	TX checksum offloading
9	RX checksum offloading
10	RX flow hash offloading

Port count

The port count field contains the number of ports associated with the interface, as configured via Verilog parameters during synthesis.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Port count				RO -

Scheduler block count

The scheduler block count field contains the number of scheduler blocks associated with the interface, as configured via Verilog parameters during synthesis.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Scheduler block count				RO -

Max TX MTU

The max TX MTU field contains the maximum frame size on the transmit path, as configured via Verilog parameters during synthesis.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x20	Max TX MTU				RO -

Max RX MTU

The max RX MTU field contains the maximum frame size on the receive path, as configured via Verilog parameters during synthesis.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x24	Max RX MTU				RO -

TX MTU

The TX MTU field controls the maximum frame size on the transmit path.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x28	TX MTU				RW -

RX MTU

The RX MTU field controls the maximum frame size on the receive path.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x2C	RX MTU				RW -

9.14 Null register block

The null register block has a header with type 0x00000000 and no additional fields after the header.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x00000000
RBB+0x04	Version	Major	Minor	Patch	Meta	RO -
RBB+0x08	Next pointer	Pointer to next register block				RO -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

9.15 PTP hardware clock register block

The PTP hardware clock register block has a header with type 0x0000C080, version 0x00000100, and carries several control registers for the PTP clock.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C080
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Control	Control				RO -
RBB+0x10	Current time	Current time (fractional ns)				RO -
RBB+0x14	Current time	Current time (ns)				RO -
RBB+0x18	Current time	Current time (sec, lower 32)				RO -
RBB+0x1C	Current time	Current time (sec, upper 32)				RO -
RBB+0x20	Get time	Get time (fractional ns)				RO -
RBB+0x24	Get time	Get time (ns)				RO -
RBB+0x28	Get time	Get time (sec, lower 32)				RO -
RBB+0x2C	Get time	Get time (sec, upper 32)				RO -
RBB+0x30	Set time	Set time (fractional ns)				RW -
RBB+0x34	Set time	Set time (ns)				RW -
RBB+0x38	Set time	Set time (sec, lower 32)				RW -
RBB+0x3C	Set time	Set time (sec, upper 32)				RW -
RBB+0x40	Period	Period (fractional ns)				RW -
RBB+0x44	Period	Period (ns)				RW -
RBB+0x48	Nominal period	Nominal period (fractional ns)				RO -
RBB+0x4C	Nominal period	Nominal period (ns)				RO -
RBB+0x50	Adj time	Adj time (fractional ns)				RW -
RBB+0x54	Adj time	Adj time (ns)				RW -
RBB+0x58	Adj time count	Adj time cycle count				RW -
RBB+0x5C	Adj time act	Adj time active				RO -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

`Current time`

The current time registers read the current time from the PTP clock, with no double-buffering.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Current time (fractional ns)				RO -
RBB+0x14	Current time (ns)				RO -
RBB+0x18	Current time (sec, lower 32)				RO -
RBB+0x1C	Current time (sec, upper 32)				RO -

`Get time`

The get time registers read the current time from the PTP clock, with all values latched coincident with reading the fractional ns register.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x20	Get time (fractional ns)				RO -
RBB+0x24	Get time (ns)				RO -
RBB+0x28	Get time (sec, lower 32)				RO -
RBB+0x2C	Get time (sec, upper 32)				RO -

Set time

The set time registers set the current time on the PTP clock, with all values latched coincident with writing the upper 32 bits of the seconds field.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x30	Set time (fractional ns)				RW -
RBB+0x34	Set time (ns)				RW -
RBB+0x38	Set time (sec, lower 32)				RW -
RBB+0x3C	Set time (sec, upper 32)				RW -

Period

The period registers control the period of the PTP clock, with all values latched coincident with writing the ns field. The period value is accumulated into the PTP clock on every clock cycle.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x40	Period (fractional ns)				RW -
RBB+0x44	Period (ns)				RW -

Nominal period

The nominal period registers contain the nominal period of the PTP clock, which corresponds to zero frequency offset in the ideal case.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x48	Nominal period (fractional ns)				RO -
RBB+0x4C	Nominal period (ns)				RO -

Adjust time

The adjust time registers can be used to slew the clock over some time period. An adjustment can be specified with some amount of time added every clock cycle for N cycles.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x50	Adj time (fractional ns)				RW -
RBB+0x54	Adj time (ns)				RW -
RBB+0x58	Adj time cycle count				RW -
RBB+0x5C	Adj time active				RO -

9.16 PTP period output register block

The PTP period output register block has a header with type 0x0000C081, version 0x00000100, and carries several control registers for the PTP period output module.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C081
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Control	Control				RW -
RBB+0x10	Start time	Start time (fractional ns)				RW -
RBB+0x14	Start time	Start time (ns)				RW -
RBB+0x18	Start time	Start time (sec, lower 32)				RW -
RBB+0x1C	Start time	Start time (sec, upper 32)				RW -
RBB+0x20	Period	Period (fractional ns)				RW -
RBB+0x24	Period	Period (ns)				RW -
RBB+0x28	Period	Period (sec, lower 32)				RW -
RBB+0x2C	Period	Period (sec, upper 32)				RW -
RBB+0x30	Width	Width (fractional ns)				RW -
RBB+0x34	Width	Width (ns)				RW -
RBB+0x38	Width	Width (sec, lower 32)				RW -
RBB+0x3C	Width	Width (sec, upper 32)				RW -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

`Control`

The control register contains several control and status bits relating to the operation of the period output module.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Control				RW -

Bit	Function
0	Enable
8	Pulse
16	Locked
24	Error

The enable bit enables/disables output of the period output module. Note that this bit does not cause the module to lose lock when clear, only to stop generating pulses.

The pulse bit reflects the current output of the PTP period output module.

The locked bit indicates that the period output module has locked on to the current PTP time and is ready to generate pulses. The output is disabled while the period output module is unlocked, so it is not necessary to wait for the module to lock before enabling the output. The module will unlock whenever the start time, period, or width setting is changed.

The error bit indicates that the period output module came out of lock due to the PTP clock being stepped. The error bit is self-clearing on either reacquisition of lock or a setting change.

The period output module keeps track of the times for the next rising edge and next falling edge. Initially, it starts with the specified start time for the rising edge, and start time plus width for the falling edge. If the computed next rising edge time is in the past, the period will be added and it will be checked again, repeating this process

until the next rising edge is in the future. Note that the period is added once per clock cycle, so it is recommended to compute a start time that is close to the current time, particularly when using a small period setting, so that the period output module can lock quickly.

Start time

The start time registers determine the absolute start time for the output waveform (rising edge), with all values latched coincident with writing the upper 32 bits of the seconds field.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Start time (fractional ns)				RW -
RBB+0x14	Start time (ns)				RW -
RBB+0x18	Start time (sec, lower 32)				RW -
RBB+0x1C	Start time (sec, upper 32)				RW -

Period

The period registers control the period of the output waveform (rising edge to rising edge), with all values latched coincident with writing the upper 32 bits of the seconds field.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x20	Period (fractional ns)				RW -
RBB+0x24	Period (ns)				RW -
RBB+0x28	Period (sec, lower 32)				RW -
RBB+0x2C	Period (sec, upper 32)				RW -

Width

The width registers control the width of the output waveform (rising edge to falling edge), with all values latched coincident with writing the upper 32 bits of the seconds field.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x30	Width (fractional ns)				RW -
RBB+0x34	Width (ns)				RW -
RBB+0x38	Width (sec, lower 32)				RW -
RBB+0x3C	Width (sec, upper 32)				RW -

9.17 Receive queue manager register block

The receive queue manager register block has a header with type 0x0000C021, version 0x00000100, and indicates the location of the receive queue manager registers and number of queues.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C021
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Offset	Offset to queue manager				RO -
RBB+0x10	Count	Queue count				RO -
RBB+0x14	Stride	Queue control register stride				RO 0x00000020

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Offset

The offset field contains the offset to the start of the receive queue manager region, relative to the start of the current region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Offset to queue manager				RO -

Count

The count field contains the number of queues.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Queue count				RO -

Stride

The stride field contains the size of the control registers associated with each queue.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Queue control register stride				RO 0x00000020

9.17.1 Queue manager CSRs

Each queue has several associated control registers, detailed in this table:

Address	Field	31..24	23..16	15..8	7..0	Reset value
Base+0x00	Base address L	Ring base address (lower 32)				RW -
Base+0x04	Base address H	Ring base address (upper 32)				RW -
Base+0x08	Control 1	Active			Size	RW -
Base+0x0C	Control 2			CQ index		RW -
Base+0x10	Head pointer			Head pointer		RW -
Base+0x14	Tail pointer			Tail pointer		RW -

9.18 Transmit queue manager register block

The transmit queue manager register block has a header with type 0x0000C020, version 0x00000100, and indicates the location of the transmit queue manager registers and number of queues.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C020
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Offset	Offset to queue manager				RO -
RBB+0x10	Count	Queue count				RO -
RBB+0x14	Stride	Queue control register stride				RO 0x00000020

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Offset

The offset field contains the offset to the start of the transmit queue manager region, relative to the start of the current region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Offset to queue manager				RO -

Count

The count field contains the number of queues.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Queue count				RO -

Stride

The stride field contains the size of the control registers associated with each queue.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Queue control register stride				RO 0x00000020

9.18.1 Queue manager CSRs

Each queue has several associated control registers, detailed in this table:

Address	Field	31..24	23..16	15..8	7..0	Reset value
Base+0x00	Base address L	Ring base address (lower 32)				RW -
Base+0x04	Base address H	Ring base address (upper 32)				RW -
Base+0x08	Control 1	Active			Size	RW -
Base+0x0C	Control 2			CQ index		RW -
Base+0x10	Head pointer			Head pointer		RW -
Base+0x14	Tail pointer			Tail pointer		RW -

9.19 Scheduler block register block

The scheduler block register block has a header with type 0x0000C004, version 0x00000300, and indicates the offset to the scheduler block control registers.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C004
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000300
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Offset	Offset to scheduler block CSRs				RO -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

Offset

The offset field contains the offset to the start of scheduler block control registers, relative to the start of the current region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Offset to scheduler block CSRs				RO -

9.20 TDMA scheduler controller register block

The TDMA scheduler controller register block has a header with type 0x0000C050, version 0x00000100, and indicates the location of the scheduler controller in the register space, as well as containing some control, status, and informational registers.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C050
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Offset	Offset to scheduler				RO -
RBB+0x10	CH count	Channel count				RO -
RBB+0x14	CH stride	Channel stride				RO -
RBB+0x18	Control	Control				RW 0x00000000
RBB+0x1C	TS count	TS count				RW -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

`Offset`

The offset field contains the offset to the start of the scheduler, relative to the start of the current region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Offset to scheduler				RO -

`Channel count`

The channel count field contains the number of channels.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Channel count				RO -

`Channel stride`

The channel stride field contains the size of the region for each channel.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Channel stride				RO -

`Control`

The control field contains scheduler-related control bits.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x18	Control				RW 0x00000000

`Timeslot count`

The timeslot count register contains the number of time slots supported.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x1C	Timeslot count				RO -

9.20.1 TDMA scheduler controller CSRs

Each scheduler control channel has several associated control registers, detailed in this table:

Address	Field	31..24	23..16	15..8	7..0	Reset value
Base+0x00	Enable bits	Enable bits				RW -
Base+N	Enable bits	Enable bits				RW -

`Enable bits`

The enable bits field contains per-timeslot channel enable bits.

Address	31..24	23..16	15..8	7..0	Reset value
Base+0x00	Enable bits				RW 0x00000000

Bit	Function
0	Timeslot 0 enable
N	Timeslot N enable

9.21 Round-robin scheduler register block

The round-robin scheduler register block has a header with type 0x0000C040, version 0x00000100, and indicates the location of the scheduler in the register space, as well as containing some control, status, and informational registers.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C040
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	Offset	Offset to scheduler				RO -
RBB+0x10	CH count	Channel count				RO -
RBB+0x14	CH stride	Channel stride				RO 0x00000004
RBB+0x18	Control	Control				RW 0x00000000
RBB+0x1C	Dest	Dest				RW -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

`Offset`

The offset field contains the offset to the start of the scheduler, relative to the start of the current region.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Offset to scheduler				RO -

`Channel count`

The channel count field contains the number of channels.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Channel count				RO -

Channel stride

The channel stride field contains the size of the region for each channel.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Channel stride				RO 0x00000004

Control

The control field contains scheduler-related control bits.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x18	Control				RW 0x00000000

Bit	Function
0	Enable

Dest

The dest field controls the destination port and traffic class of the scheduler. It is initialized with the scheduler's index with traffic class 0.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x1C	Dest				RW -

9.21.1 Round-robin scheduler CSRs

Each scheduler channel has several associated control registers, detailed in this table:

Address	Field	31..24	23..16	15..8	7..0	Reset value
Base+0x00	Control	Control				RW 0x00000000

Control

The control field contains scheduler-related control bits.

Address	31..24	23..16	15..8	7..0	Reset value
Base+0x00	Control				RW 0x00000000

Bit	Function
0	Enable
1	Global enable
2	Control enable
16	Active
24	Scheduled

9.22 TDMA scheduler register block

The TDMA scheduler register block has a header with type 0x0000C060, version 0x00000100, and carries several control registers for the TDMA scheduler module.

Address	Field	31..24	23..16	15..8	7..0	Reset value
RBB+0x00	Type	Vendor ID		Type		RO 0x0000C060
RBB+0x04	Version	Major	Minor	Patch	Meta	RO 0x00000100
RBB+0x08	Next pointer	Pointer to next register block				RO -
RBB+0x0C	TS count	Timeslot count				RO -
RBB+0x10	Control	Control				RW -
RBB+0x14	Status	Status				RO -
RBB+0x20	Sch start	Sch start time (fractional ns)				RW -
RBB+0x24	Sch start	Sch start time (ns)				RW -
RBB+0x28	Sch start	Sch start time (sec, lower 32)				RW -
RBB+0x2C	Sch start	Sch start time (sec, upper 32)				RW -
RBB+0x30	Sch period	Sch period (fractional ns)				RW -
RBB+0x34	Sch period	Sch period (ns)				RW -
RBB+0x38	Sch period	Sch period (sec, lower 32)				RW -
RBB+0x3C	Sch period	Sch period (sec, upper 32)				RW -
RBB+0x40	TS period	TS period (fractional ns)				RW -
RBB+0x44	TS period	TS period (ns)				RW -
RBB+0x48	TS period	TS period (sec, lower 32)				RW -
RBB+0x4C	TS period	TS period (sec, upper 32)				RW -
RBB+0x50	Active period	Active period (fractional ns)				RW -
RBB+0x54	Active period	Active period (ns)				RW -
RBB+0x58	Active period	Active period (sec, lower 32)				RW -
RBB+0x5C	Active period	Active period (sec, upper 32)				RW -

See *Register blocks* (page 103) for definitions of the standard register block header fields.

`Timeslot count`

The timeslot count register contains the number of time slots supported.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x0C	Timeslot count				RO -

`Control`

The control register contains several control bits relating to the operation of the TDMA scheduler module.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x10	Control				RW -

Bit	Function
0	Enable

`Status`

The control register contains several status bits relating to the operation of the TDMA scheduler module.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x14	Status				RO -

Bit	Function
0	Locked
1	Error

Schedule start time

The schedule start time registers determine the absolute start time for the schedule, with all values latched coincident with writing the upper 32 bits of the seconds field.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x20	Sch start time (fractional ns)				RW -
RBB+0x24	Sch start time (ns)				RW -
RBB+0x28	Sch start time (sec, lower 32)				RW -
RBB+0x2C	Sch start time (sec, upper 32)				RW -

Schedule period

The schedule period registers control the period of the schedule, with all values latched coincident with writing the upper 32 bits of the seconds field.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x30	Sch period (fractional ns)				RW -
RBB+0x34	Sch period (ns)				RW -
RBB+0x38	Sch period (sec, lower 32)				RW -
RBB+0x3C	Sch period (sec, upper 32)				RW -

Timeslot period

The timeslot period registers control the period of each time slot, with all values latched coincident with writing the upper 32 bits of the seconds field.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x40	TS period (fractional ns)				RW -
RBB+0x44	TS period (ns)				RW -
RBB+0x48	TS period (sec, lower 32)				RW -
RBB+0x4C	TS period (sec, upper 32)				RW -

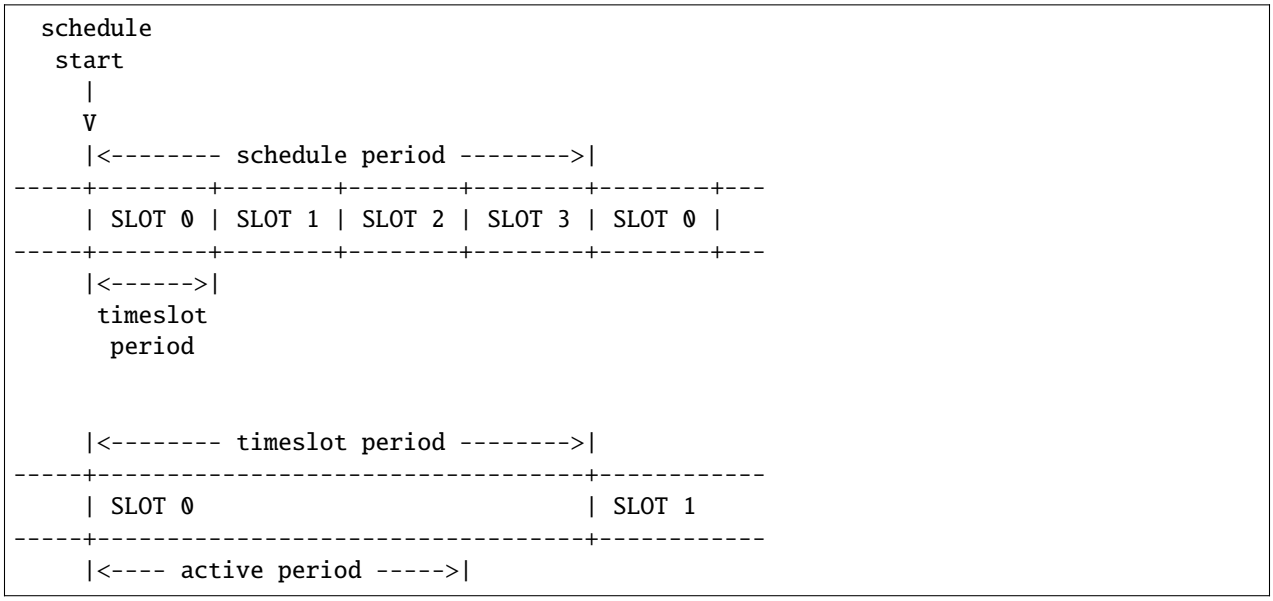
Active period

The active period registers control the active period of each time slot, with all values latched coincident with writing the upper 32 bits of the seconds field.

Address	31..24	23..16	15..8	7..0	Reset value
RBB+0x50	Active period (fractional ns)				RW -
RBB+0x54	Active period (ns)				RW -
RBB+0x58	Active period (sec, lower 32)				RW -
RBB+0x5C	Active period (sec, upper 32)				RW -

9.22.1 TDMA timing parameters

The TDMA schedule is defined by several parameters - the schedule start time, schedule period, timeslot period, and timeslot active period. This figure depicts the relationship between these parameters:



The schedule start time is the absolute start time. Each subsequent schedule will start on a multiple of the schedule period after the start time. Each schedule starts on timeslot 0, and advances to the next timeslot each timeslot period. The timeslot active period is the active period for each timeslot, forming a guard period at the end of the timeslot. It is recommended that the timeslot period divide evenly into the schedule period, but rounding errors will not accumulate as the schedule period takes precedence over the timeslot period. Similarly, the timeslot period takes precedence over the timeslot active period.

CHAPTER

TEN

DEVICE LIST

This section includes a summary of the various devices supported by Corundum, including a summary of board-specific features.

10.1 PCIe

This section details PCIe form-factor targets, which interface with a separate host system via PCI express as a PCIe endpoint.

Table 10.1: Summary of the various devices supported by Corundum.

Manufacturer	Board	FPGA	Board ID
Alpha Data	ADM-PCIE-9V3	XCVU3P-2FFVC1517I	0x41449003
Dini Group	DNPCIe_40G_KU_LL_2QSFP	XCKU040-2FFVA1156E	0x17df1a00
Cisco	Nexus K35-S	XCKU035-2FBVA676E	0x1ce40003
Cisco	Nexus K3P-S	XCKU3P-2FFVB676E	0x1ce40009
Cisco	Nexus K3P-Q	XCKU3P-2FFVB676E	0x1ce4000a
Silicom	fb2CG@KU15P	XCKU15P-2FFVE1760E	0x1c2ca00e
Digilent	NetFPGA SUME	XC7V690T-3FFG1761	0x10ee7028
BittWare	XUP-P3R	XCVU9P-2FLGB2104E	0x12ba9823
BittWare	250-SoC	XCZU19EG-2FFVD1760E	0x198a250e
Intel	DK-DEV-1SMX-H-A	1SM21BHU2F53E1VG	0x11720001
Intel	DK-DEV-1SMC-H-A	1SM21CHU1F53E1VG	0x11720001
Intel	DK-DEV-1SDX-P-A	1SD280PT2F55E1VG	0x1172a00d
Terasic	DE10-Agilex	AGFB014R24B2E2V	0x1172b00a
Xilinx	Alveo U50	XCU50-2FSVH2104E	0x10ee9032
Xilinx	Alveo U200	XCU200-2FSGD2104E	0x10ee90c8
Xilinx	Alveo U250	XCU250-2FIGD2104E	0x10ee90fa
Xilinx	Alveo U280	XCU280-L2FSVH2892E	0x10ee9118
Xilinx	VCU108	XCVU095-2FFVA2104E	0x10ee806c
Xilinx	VCU118	XCVU9P-L2FLGA2104E	0x10ee9076
Xilinx	VCU1525	XCVU9P-L2FSGD2014E	0x10ee95f5
Xilinx	ZCU106	XCZU7EV-2FFVC1156E	0x10ee906a

Table 10.2: Summary of available interfaces and on-board memory.

Board	PCIe IF	Network IF	DDR	HBM
ADM-PCIE-9V3	Gen 3 x16	2x QSFP28	16 GB DDR4 2400 (2x 1G x72)	-
DNPCIe_40G_KU_LL_2QSFP	Gen 3 x8	2x QSFP+	4 GB DDR4 2400 (512M x72)	-
Nexus K35-S	Gen 3 x8	2x SFP+	-	-
Nexus K3P-S	Gen 3 x8	2x SFP28	4 GB DDR4 (1G x32)	-
Nexus K3P-Q	Gen 3 x8	2x QSFP28	8 GB DDR4 (1G x72)	-
fb2CG@KU15P	Gen 3 x16	2x QSFP28	16 GB DDR4 2400 (4x 512M x72)	-
NetFPGA SUME	Gen 3 x8	4x SFP+	8 GB DDR3 1866 (2x 512M x64)	-
250-SoC	Gen 3 x16	2x QSFP28	4 GB DDR4 2400 (512M x72)	-
XUP-P3R	Gen 3 x16	4x QSFP28	4x DDR4 2400 DIMM (4x x72)	-
DK-DEV-1SMX-H-A	Gen 3 x16	2x QSFP28	8 GB DDR4 2666 (2x 512M x72)	8 GB
DK-DEV-1SMC-H-A	Gen 3 x16	2x QSFP28	8 GB DDR4 2666 (2x 512M x72)	16 GB
DK-DEV-1SDX-P-A	Gen 4 x16	2x QSFP28	2x 4GB DDR4 512M x72, 2x DIMM	-
DE10-Agilex	Gen 4 x16	2x QSFP-DD	4x 8GB DDR4 3200 DIMM (4x 72)	-
Alveo U50	Gen 3 x16	1x QSFP28	-	8 GB
Alveo U200	Gen 3 x16	2x QSFP28	64 GB DDR4 2400 (4x 2G x72)	-
Alveo U250	Gen 3 x16	2x QSFP28	64 GB DDR4 2400 (4x 2G x72)	-
Alveo U280	Gen 3 x16	2x QSFP28	32 GB DDR4 2400 (2x 2G x72)	8 GB
VCU108	Gen 3 x8	1x QSFP28	4 GB DDR4 2400 (2x 256M x80)	-
VCU118	Gen 3 x16	2x QSFP28	4 GB DDR4 2400 (2x 256M x80)	-
VCU1525	Gen 3 x16	2x QSFP28	64 GB DDR4 2400 (4x 2G x72)	-
ZCU106	Gen 3 x4	2x SFP+	2 GB DDR4 2400 (256M x64)	-

Table 10.3: Summary of support for various ancillary features.

Board	I2C [1]	MAC [2]	FW update
ADM-PCIE-9V3	N [3]	Y [5]	Y
DNPCIe_40G_KU_LL_2QSFP	Y	N [3]	Y
Nexus K35-S	N [3]	Y	Y
Nexus K3P-S	N [3]	Y	Y
Nexus K3P-Q	Y	Y	Y
fb2CG@KU15P	Y	Y	Y
NetFPGA SUME	Y	N [7]	N [8]
250-SoC	Y	N	N [9]
XUP-P3R	Y	Y	Y
DK-DEV-1SMX-H-A	N	N	N
DK-DEV-1SMC-H-A	N	N	N
DK-DEV-1SDX-P-A	N	N	N [10]
DE10-Agilex	Y	N	N
Alveo U50	N [4]	Y	Y
Alveo U200	Y	Y	Y
Alveo U250	Y	Y	Y
Alveo U280	N [4]	Y	Y
VCU108	Y	Y [5]	Y
VCU118	Y	Y [5]	Y
VCU1525	Y	Y [5]	Y
ZCU106	Y	Y [5]	N [9]

- [1] I2C access to optical modules
- [2] Persistent MAC address storage

- [3] Supported in hardware, driver support in progress
- [4] Limited read/write access via BMC pending driver support, full read/write access requires support in BMC firmware
- [5] Can read MAC from I2C EEPROM, but EEPROM is blank from factory
- [6] MAC available from BMC, but accessing BMC is not yet implemented
- [7] No on-board EEPROM
- [8] Flash sits behind board management controller, not currently exposed via PCIe
- [9] Flash sits behind Zynq SoC, not currently exposed via PCIe
- [10] Flash sits behind board management controller, inaccessible

Table 10.4: Summary of the board-specific design variants and some important configuration parameters.

Board	Design	IFxP	RXQ/TXQ	MAC	PTP	Sched
ADM-PCIE-9V3	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
ADM-PCIE-9V3	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
ADM-PCIE-9V3	mqnic/fpga_25g/fpga_tdma	2x1	256/256	25G	Y	TDMA
ADM-PCIE-9V3	mqnic/fpga_100g/fpga	2x1	256/8K	100G	Y	RR
ADM-PCIE-9V3	mqnic/fpga_100g/fpga_tdma	2x1	256/256	100G	Y	TDMA
DNPCIe_40G_KU_LL_2QSFP	mqnic/fpga/fpga_ku040	2x1	256/2K	10G	Y	RR
DNPCIe_40G_KU_LL_2QSFP	mqnic/fpga/fpga_ku060	2x1	256/2K	10G	Y	RR
Nexus K35-S	mqnic/fpga/fpga	2x1	256/2K	10G	Y	RR
Nexus K3P-S	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
Nexus K3P-S	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
Nexus K3P-Q	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
Nexus K3P-Q	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
fb2CG@KU15P	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
fb2CG@KU15P	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
fb2CG@KU15P	mqnic/fpga_25g/fpga_tdma	2x1	256/256	25G	Y	TDMA
fb2CG@KU15P	mqnic/fpga_100g/fpga	2x1	256/8K	100G	Y	RR
fb2CG@KU15P	mqnic/fpga_100g/fpga_tdma	2x1	256/256	100G	Y	TDMA
NetFPGA SUME	mqnic/fpga/fpga	1x1	256/512	10G	Y	RR
250-SoC	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
250-SoC	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
250-SoC	mqnic/fpga_100g/fpga	2x1	256/8K	100G	Y	RR
XUP-P3R	mqnic/fpga_25g/fpga	4x1	256/8K	25G	Y	RR
XUP-P3R	mqnic/fpga_25g/fpga_10g	4x1	256/8K	10G	Y	RR
XUP-P3R	mqnic/fpga_100g/fpga	4x1	256/8K	100G	Y	RR
DK-DEV-1SMX-H-A	mqnic/fpga_25g/fpga_1sm21b	2x1	256/1K	25G	Y	RR
DK-DEV-1SMC-H-A	mqnic/fpga_25g/fpga_1sm21c	2x1	256/1K	25G	Y	RR
DK-DEV-1SMX-H-A	mqnic/fpga_25g/fpga_10g_1sm21b	2x1	256/1K	10G	Y	RR
DK-DEV-1SMC-H-A	mqnic/fpga_25g/fpga_10g_1sm21c	2x1	256/1K	10G	Y	RR
DK-DEV-1SDX-P-A	mqnic/fpga_25g/fpga	2x1	256/1K	25G	Y	RR
DK-DEV-1SDX-P-A	mqnic/fpga_25g/fpga_10g	2x1	256/1K	10G	Y	RR
DE10-Agilex	mqnic/fpga_25g/fpga	2x1	256/1K	25G	Y	RR
DE10-Agilex	mqnic/fpga_25g/fpga_10g	2x1	256/1K	10G	Y	RR
DE10-Agilex	mqnic/fpga_100g/fpga	2x1	256/1K	100G	N	RR
Alveo U50	mqnic/fpga_25g/fpga	1x1	256/8K	25G	Y	RR
Alveo U50	mqnic/fpga_25g/fpga_10g	1x1	256/8K	10G	Y	RR

continues on next page

Table 10.4 – continued from previous page

Board	Design	IFxP	RXQ/TXQ	MAC	PTP	Sched
Alveo U50	mqnic/fpga_100g/fpga	1x1	256/8K	100G	Y	RR
Alveo U200	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
Alveo U200	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
Alveo U200	mqnic/fpga_100g/fpga	2x1	256/8K	100G	Y	RR
Alveo U250	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
Alveo U250	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
Alveo U250	mqnic/fpga_100g/fpga	2x1	256/8K	100G	Y	RR
Alveo U280	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
Alveo U280	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
Alveo U280	mqnic/fpga_100g/fpga	2x1	256/8K	100G	Y	RR
VCU108	mqnic/fpga_25g/fpga	1x1	256/2K	25G	Y	RR
VCU108	mqnic/fpga_25g/fpga_10g	1x1	256/2K	10G	Y	RR
VCU118	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
VCU118	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
VCU118	mqnic/fpga_100g/fpga	2x1	256/8K	100G	Y	RR
VCU1525	mqnic/fpga_25g/fpga	2x1	256/8K	25G	Y	RR
VCU1525	mqnic/fpga_25g/fpga_10g	2x1	256/8K	10G	Y	RR
VCU1525	mqnic/fpga_100g/fpga	2x1	256/8K	100G	Y	RR
ZCU106	mqnic/fpga_pcie/fpga	2x1	256/8K	10G	Y	RR

10.2 SoC

This section details SoC targets, which interface with CPU cores on the same device, usually via AXI.

Table 10.5: Summary of the various devices supported by Corundum.

Manufacturer	Board	FPGA	Board ID
Xilinx	ZCU102	XCZU9EG-2FFVB1156E	0x10ee9066
Xilinx	ZCU106	XCZU7EV-2FFVC1156E	0x10ee906a

Table 10.6: Summary of available interfaces and on-board memory.

Board	PCIe IF	Network IF	DDR	HBM
ZCU102	-	4x SFP+	2 GB DDR4 2400 (256M x64)	-
ZCU106	Gen 3 x4	2x SFP+	2 GB DDR4 2400 (256M x64)	-

Table 10.7: Summary of support for various ancillary features.

Board	I2C [1]	MAC [2]	FW update
ZCU102	Y	Y [3]	N
ZCU106	Y	Y [3]	N

- [1] I2C access to optical modules
- [2] Persistent MAC address storage
- [3] Can read MAC from I2C EEPROM, but EEPROM is blank from factory

Table 10.8: Summary of the board-specific design variants and some important configuration parameters.

Board	Design	IFxP	RXQ/TXQ	MAC	Sched
ZCU102	mqnic/fpga/fpga	2x1	32/32	10G	RR
ZCU106	mqnic/fpga_zynqmp/fpga	2x1	32/32	10G	RR

CHAPTER

ELEVEN

GLOSSARY

AXI
Advanced eXtensible Interface

BAR
Base Address Register

DMA
Direct Memory Access

FPGA
Field-Programmable Gate Array

JTAG
Joint Test Action Group

MAC
Media Access Control(ler)

MSI
Message-Signaled Interrupt

NIC
Network Interface Controller

PCI
Peripheral Component Interconnect

PCIe
PCI Express

PHY
PHYsical layer (interface)

PTP
Precision Time Protocol (IEEE 1588)

RSS
Receive Side Scaling

INDEX