P51 - Lab 1, Introduction to NetFPGA

Dr Noa Zilberman

Lent, 2018/19

The goal of this lab is to introduce you to the NetFPGA platform, and to provide hands on experience both in using the platform and developing for it. We begin with the Verilog based workflow, and follow with the P4 based workflow.

1 Background

The demand-led growth of cloud computing and datacenter networks has meant that many constituent technologies are beyond the budget of the research community. In order to make and validate timely, relevant research contributions, the wider research community requires accessible evaluation, experimentation and demonstration environments with specification comparable to the subsystems of the most massive datacenter networks.

The NetFPGA is an open platform enabling researchers and instructors to build high-speed, hardware-accelerated networking systems. The most prominent NetFPGA success is OpenFlow, which in turn has reignited the Software Defined Networking movement. NetFPGA enabled OpenFlow by providing a widely available open-source development platform capable of line-rate operation and was, until its commercial uptake, the reference platform for OpenFlow.

NetFPGA enables rapid prototyping of high bandwidth devices, using flexible, open-source IPs. Specifically, we use NetFPGA SUME, an open-source FPGA-based PCIe board, designed for the research community. NetFPGA SUME has I/O capabilities for 100Gbps operation as a networking device, stand alone computing unit or for test and measurement.

2 Development Machines

During the course you will use an assigned development machine. Each pair will be assigned different machines. All the machines are located in the Practical Classroom (SW02). This provides access to the machines, so you can change the physical connectivity according to the experiment.

You will interact with the machines via ssh:

1. On a computer in the Practical Classroom, log in using your own UIS credentials.
2. From Moodle, download the private key.
3. Limit the permissions of the private key: `chmod 600 p51_key`
4. `ssh -X root@<hostname>.nf.cl.cam.ac.uk -i p51_key`. Hosts ending in .cl.cam.ac.uk are permitted to ssh into these machines. `-X` enables X11 forwarding, allowing you to run graphical applications. `-i` is the private ssh authentication key.
   To ssh to the machines from outside the lab, follow the instructions on `https://www.cl.cam.ac.uk/local/sys/ssh/`.

Hostname	IP Address
nf-test103	128.232.82.63
nf-test104	128.232.82.64
nf-test102	128.232.82.62
nf-test108	128.232.82.68
nf-test110	128.232.82.70
nf-test111	128.232.82.71

Important: The IP addresses noted above should not be used for anything except for communication with the machines. The network interfaces assigned for the tests use different IP addresses.

3 Test Setup

Each of the development machines is equipped as follows:

- NetFPGA SUME board, used as the development platform
  NetFPGA network ports are marked *nf0* to *nf3*.
- Intel network interface cards (NIC) - X520-DA2
  NIC network ports are marked *intl0* and *intl1*.
- Optical fibers - duplex fibres with separate strands for transmitting and receiving.
- Optical 10G transceivers (SFP+) - converting electrical signals to optical signals and vice versa.

The network port markings noted above apply only to Figure 1. The name of the port on each machine may differ, (e.g.,*eth2, eth3*).
In a typical setup, each NIC port will be connected to a NetFPGA SUME port, e.g. intl0 to nf0, and intl1 to nf1.

Figure 1: Development Platform

4 Saving Your Work

Make sure to frequently back up your work.
The most recommended way to back up your work is using frequent commits to a git repository. Please do not push any changes, data or results directly to the NetFPGA repository. You can fork the repository to your own user and push changes there. If you would like to suggest a correction or an enhancement to the code, please follow the instructions in:
`https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/Contributing-Code`.

To copy a remote directory onto your local machine:
`sftp root@<hostname>.nf.cl.cam.ac.uk` and `get -r <directory>`.
There are also other ways to copy a remote directory, you are welcome to use those as well.

5 Using NetFPGA - Verilog Design Flow

This section provides step-by-step instructions how to access and test a NetFPGA design. To this end, we will be using the verilog Reference Switch design studied in class.

5.1 Accessing and programming the board

1. Login to the development machine:

   ```
   ssh -X root@<hostname>.nf.cl.cam.ac.uk -i p51\_key
   ```

2. ```
   cd ~/NetFPGA-SUME-live/tools/
   vim settings.sh
   ```

3. Make sure that NF_PROJECT_NAME is set to "reference_switch"

4. Load the environment setting (can be added to ~/.bashrc)s:

   ```
   source settings.sh
   ```

5. Compile the driver (one time only):

   ```
   cd $DRIVER_FOLDER
   make
   ```

6. Compile register access application (one time only):

   ```
   cd $APPS_FOLDER
   make
   ```

7. Program the NetFPGA board:

   ```
   cd $SUME_FOLDER/tools/scripts
   chmod +x run_load_image.sh
   ./run_load_image.sh $NF_DESIGN_DIR/bitfiles/reference_switch.bit
   ```

   Note that you may need to reset the machine if this is the first time the board is programmed after power up. This allows the PCIe bus to properly identify and enumerate the board. The first time the board is programmed after reset (not power up!), you may see a message "rmmod: ERROR: Module sume_riffa is not currently loaded". It can be ignored.

8. The board is now programmed and ready.
   ```

5.2 Simple Debug

1. Check if the board is recognized by the host:

   ```
   lspci -v |grep Xilinx
   ```

   The expected result is:

   ```
   01:00.0 Memory controller: Xilinx Corporation Device 7028
   Subsystem: Xilinx Corporation Device 0007
   ```

2. Check if simple register access works:

   ```
   cd $APPS_FOLDER
   ./rwaxi
   ```

   The expected result is:

   ```
   WARNING: using default test address 0x44020000
   READ  0x44020000 = 0x0001da02
   ```

   Note that the return value may change - 0x0001da02 is the ID of the output port lookup module. You module(s) may use a different value.

3. Reading all the design registers:

   ```
   cd $APPS_FOLDER
   make register_read.sh
   cd $NF_DESIGN_DIR/sw/host/apps/
   ./register_read.sh
   ```

4. Reading a specific register (e.g. address 0x44010004):

   ```
   cd $APPS_FOLDER
   ./rwaxi -a 0x44010008
   ```

5. Writing to a specific register (e.g. address 0x44010010, value 0xabcdabcd):

   ```
   cd $APPS_FOLDER
   ./rwaxi -a 0x44010010 -w 0xabcdabcd
   ```

6. Making sure that register access reaching the module:
   To make sure that a register access reaches the hardware and is not replied by some cache, most NetFPGA modules implement a "FLIP" register (typically at offset 0xc). The FLIP register returns the inverse of the value written to it. For example:

```
cd $APPS_FOLDER
./rwaxi -a 0x4401000C -w 0x55555555
./rwaxi -a 0x4401000C
```

Will return:

```
READ  0x4401000c = 0xaaaaaaaa
```

7. Making a file (e.g., sh) executable:

```
chmod +x <filename>
```

5.3 Building a project

The following steps are typically required when building a project:

1. Compiling an IP core:

```
cd $IP_FOLDER/<ip core name>
make
```

This step is required only for new IP cores or when changes are made to the tcl file of the core. There is no need to run make if only the HDL files were modified.

2. Compiling CAM/TCAM cores:
Follows the instructions on `https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-TCAM-IPs`
The CAM core is required for building the NetFPGA project. For this lab, there is no need to build the TCAM. You only need to run this step once. Step #1 can be skipped, as xapp1151 zip files were already downloaded to the folder.

3. Compiling all cores and building libraries:

```
cd $SUME_FOLDER
make
```

This step if typically required only once: after the git repository is cloned or pulled. It is also required if the *make clean* command was called.

4. Building a project:

```
cd $NF_DESIGN_DIR\
make
```

The result of this step is the programming (bit) file. *This step takes 45 minutes or more. Do not run it during class. You can run the next section independently.*

5.4 Testing a design

This section covers simple design testing of the NetFPGA platform, focusing on functionality. Performance testing will be discussed later in the course. There is no need to generate the bit file to do this section, a bit file was already generated for you.

For each NetFPGA design, functional tests need to be written. The tests are located under *$NF_DESIGN_DIR/test*. Each test has a dedicated folder called *hw/sim/both_major_minor*, for example *both_simple_broadcast* or *both_learning_sw*. The test itself is written in python, in a file called *run.py*. The NetFPGA test environment (python based as well) calls this file when invoked.

The NetFPGA Reference Switch is a *Learning Switch*, meaning that forwarding is done based on MAC addresses that the switch sees in incoming packets and associates with ports. For example, is a packet with source MAC address $aa:bb:cc:dd:ee:ff$ is received on Port 1, the switch will save in its lookup table an entry equivalent to "$aa:bb:cc:dd:ee:ff$, Port 1", and the next time a packet arrives with a destination MAC address $aa:bb:cc:dd:ee:ff$ it will know to send it to port 1. When the destination MAC address is not in the lookup table, the switch uses broadcast: it sends the packet on all ports (except for the incoming port).

The test *both_simple_broadcast* tests the switch's broadcast operation, and the test *both_learning_sw* tests both broadcast and learning. Read the file *run.py* under each test to learn how exactly it is done.

The following describes the steps for running a simulation, without GUI:

```
cd $SUME_FOLDER/tools/scripts
./nf_test.py sim --major learning --minor sw
```

The following describes the steps for running a simulation, with GUI (Vivado xsim):

```
cd $SUME_FOLDER/tools/scripts
./nf_test.py sim --major learning --minor sw --gui
```

Note that using xsim is not mandatory. You can also change the environment and use Modelsim, and a license for that is available. The following describes the steps for running a hardware test:

1. Connect port 0 of the NIC to port 0 of NetFPGA using a fibre.
2. Connect port 1 of the NIC to port 1 of NetFPGA using a fibre.
3. Check what are the interfaces names of your NIC (These can also be configures/modified, but it is not mandatory):

   ```
   ifconfig -a
   ```

4. Update interface names in the project test configuration files:

   ```
   vim $NF_DESIGN_DIR/test/global/setup
   vim $NF_DESIGN_DIR/test/connections/conn
   ```

Note that *connections/conn* file reflects the physical (external) connectivity between the NIC's ports and NetFPGA's ports.

5. Run the hardware test:

   ```
   cd $SUME_FOLDER/tools/scripts
   ./nf_test.py hw --major learning --minor sw
   ```

   If the hardware test fails, check your fibres and transceivers. Pay attention to any error messages.

The Reference Switch also works as a normal switch, enabling to connect multiple devices. For the following exercise you should work with another team, so two machines are used. We refer to those as Machine A and machine B.

1. On Machine A, Connect port 0 of the NIC to port 0 of NetFPGA using a fibre.
2. On Machine B, Connect port 1 of the NIC to port 1 of NetFPGA on Machine A using a fibre.
3. On Machine A, configure NIC port 0 (lets assume it is called eth1):

   ```
   ifconfig eth1 10.0.0.100 up
   ```

4. On Machine B, configure NIC port 0 (lets assume it is called eth1):

   ```
   ifconfig eth1 10.0.0.101 up
   ```

5. Check that the connectivity between both machines works. From Machine A ping Machine B:

   ```
   ping 10.0.0.101
   ```

6. You can also check the same in the opposite direction. From Machine B ping Machine A:

   ```
   ping 10.0.0.100
   ```

6 Using NetFPGA - P4 Design Flow

This section provides step-by-step instructions how to access and test a NetFPGA design. To this end, we will be using the P4 Simple SUME Switch design studied in class, combined with the calculator application.

6.1 Accessing and programming the board

1. Login to the development machine:

   ```
   ssh -X root@<hostname>.nf.cl.cam.ac.uk -i p51\_key
   ```

2. Edit bashrc to use Vivado 2018.2. This step needs to be done only the first time you connect to the machine:

   ```
   vim ~/.bashrc
   ```

   Go to line 103. change the line to:

   ```
   source /opt/Xilinx/Vivado/2018.2/settings64.sh\\
   ```

   Exit the ssh connection and connect again to the machine. Note that sourcing bashrc again (without exiting current session) will lead to problems.

3. ```
   cd ~/P4-NetFPGA/tools/
   vim settings.sh
   ```

4. Make sure that NF_PROJECT_NAME is set to "simple_sume_switch"

5. Make sure that P4_PROJECT_NAME is set to "switch_calc"

6. Load the environment settings (can be added to ~/.bashrc):

   ```
   source settings.sh
   ```

7. Compile the driver (one time only):

   ```
   cd $DRIVER_FOLDER
   make
   ```

8. Compile register access application (one time only):

   ```
   cd $APPS_FOLDER
   make
   ```

9. Program the NetFPGA board:

   ```
   cd $NF_DESIGN_DIR/bitfiles
   source program_switch.sh
   ```

   Note that you may need to reset the machine if this is the first time the board is programmed after power up. This allows the PCIe bus to properly identify and enumerate the board. The first time the board is programmed after reset (not power up!), you may see a message "rmmod: ERROR: Module sume_riffa is not currently loaded". It can be ignored.

10. The board is now programmed and ready.
   ```

6.2 Simple Debug

1. Check if the board is recognized by the host:

   ```
   lspci -v |grep Xilinx
   ```

   The expected result is:

   ```
   01:00.0 Memory controller: Xilinx Corporation Device 7028
   Subsystem: Xilinx Corporation Device 0007
   ```

2. Check if simple register access works:

   ```
   cd $APPS_FOLDER
   ./rwaxi
   ```

   The expected result is:

   ```
   WARNING: using default test address 0x44020000
   READ  0x44020000 = 0x0000032
   ```

   Note that the return value may change - 0x0000032 is the ID of the output port lookup module. You module(s) may use a different value.

3. Reading common (Verilog) design registers:

   ```
   cd $APPS_FOLDER
   make register_read
   cd $NF_DESIGN_DIR/sw/host/apps/
   ./register_read.sh
   ```

4. Reading a specific register (e.g. address 0x44010004):

   ```
   cd $APPS_FOLDER
   ./rwaxi -a 0x44010008
   ```

5. Writing to a specific register (e.g. address 0x44010010, value 0xabcdabcd):

   ```
   cd $APPS_FOLDER
   ./rwaxi -a 0x44010010 -w 0xabcdabcd
   ```

6. Making sure that register access reaching the module:
   To make sure that a register access reaches the hardware and is not replied by some cache, most NetFPGA modules implement a "FLIP" register (typically at offset 0xc). The FLIP register returns the inverse of the value written to it. For example:

```
cd $APPS_FOLDER
./rwaxi -a 0x4401000C -w 0x55555555
./rwaxi -a 0x4401000C
```

Will return:

```
READ  0x4401000c = 0xaaaaaaaa
```

7. Making a file (e.g., sh) executable:

```
chmod +x <filename>
```

8. To read registers within the P4 module, follow *Building a project* or *Testing a design* first (no need to regenerate the bit file).

6.3 Building a project

The following steps are typically required when building a project:

1. Compiling an IP core:

```
cd $IP_FOLDER/<ip core name>
make
```

This step is required only for new IP cores or when changes are made to the tcl file of the core. There is no need to run make if only the HDL files were modified.

2. Compiling CAM/TCAM cores:
Follows the instructions on `https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-TCAM-IPs`
The CAM core is required for building the NetFPGA project. You only need to run this step once. Step #1 can be skipped, as xapp1151 zip files were already downloaded to the folder.

3. Compiling all cores and building libraries:

```
cd $SUME_FOLDER
make
```

This step if typically required only once: after the git repository is cloned or pulled. It is also required if the *make clean* command was called.

4. Write your P4 program (there is no need to do it as part of the lab):

```
cd $P4_PROJECT_DIR/src
vim <filename>.p4
```

5. Add entries to the tables you defined in the P4 program:

   ```
   cd $P4_PROJECT_DIR/src
   vim commands.txt
   ```

6. Run the P4-SDNet compiler to generate the resulting HDL and an initial simulation framework:

   ```
   cd $P4_PROJECT_DIR && make
   ```

7. Generate the scripts that configure the table entries (current release requires that you run simulation first, see next section):

   ```
   cd $P4_PROJECT_DIR && make config_writes
   ```

8. Wrap SDNet output in wrapper module and install as a NetFPGA library core:

   ```
   cd $P4_PROJECT_DIR
   make uninstall_sdnet && make install_sdnet
   ```

9. Building a project:

   ```
   cd $NF_DESIGN_DIR
   make
   ```

   The result of this step is the programming (bit) file. *This step takes 45 minutes or more. Do not run it during class.*

6.4 Testing a design

This section covers simple design testing of the NetFPGA platform, focusing on functionality. Performance testing will be discussed later in the course.

For each NetFPGA design, functional tests need to be written. The tests are located under *$NF_DESIGN_DIR/test*. Each test has a dedicated folder called *hw/sim/both_major_minor*, for example *both_simple_broadcast* or *sim_switch_ctrlWrites*. The test itself is written in python, in a file called *run.py*. The NetFPGA test environment (python based as well) calls this file when invoked.

Contrary to the Verilog-based Reference Switch, the P4-NetFPGA Simple SUME Switch is **not** a *Learning Switch* (except for the learning switch program), meaning that forwarding is done based on entries that are written to the tables. These values are written in the commands.txt file.

As in P4-NetFPGA there is a shared architecture, with unique programs per design, program-specific python scripts are often used to test a design.

The following describes the steps for running a simulation of the P4 code in the SDNet environment, without GUI. The test is driven by gen_testdata.py:

```
cd $P4_PROJECT_DIR/nf_sume_sdnet_ip/SimpleSumeSwitch
./vivado_sim.bash
```

The following describes the steps for running a simulation of the P4 code in the SDNet environment, with GUI:

```
cd $P4_PROJECT_DIR/nf_sume_sdnet_ip/SimpleSumeSwitch
./vivado_sim_waveform.bash
```

The following describes the steps for running a simulation of the entire NetFPGA design:

1. Generate the scripts that configure table entries:

   ```
   cd $P4_PROJECT_DIR
   make config_writes
   ```

2. Wrap SDNet output in a wrapper module and install as a NetFPGA library core:

   ```
   cd $P4_PROJECT_DIR
   make uninstall_sdnet && make install_sdnet
   ```

3. Set up the simulation environment:

   ```
   cd $NF_DESIGN_DIR/test/sim_switch_default
   make
   ```

4. Run the simulation:

   ```
   cd $SUME_FOLDER/tools/scripts/
   ./nf_test.py sim --major switch --minor default
   ```

To run a simulation with GUI (Vivado xsim), replace the last stage with:

```
cd $SUME_FOLDER/tools/scripts
./nf_test.py sim --major switch --minor default --gui
```

Note that using xsim is not mandatory. You can also change the environment and use Modelsim, and a license for that is available.

The following describes the steps for running a hardware test:

1. Connect port 0 of the NIC to port 0 of NetFPGA using a fibre.
2. Connect port 1 of the NIC to port 1 of NetFPGA using a fibre.
3. Check what are the interfaces names of your NIC (These can also be configures/modified, but it is not mandatory):

```
ifconfig -a
```

4. Update interface names in the project test configuration files:

   ```
   vim $NF_DESIGN_DIR/test/global/setup
   vim $NF_DESIGN_DIR/test/connections/conn
   ```

   Note that *connections/conn* file reflects the physical (external) connectivity between the NIC's ports and NetFPGA's ports.

5. To run a P4 program test (e.g., switch calc), use help run_test to see test specific commands:

   ```
   cd $P4_PROJECT_DIR/sw/hw_test_tool
   ./switch_calc_tester.py
   help run_test
   run_test 2 + 3
   ```

6. To run a NetFPGA hardware test (generic test, not used in this example):

   ```
   cd $SUME_FOLDER/tools/scripts
   ./nf_test.py hw --major <name> --minor <name>
   ```

   If the hardware test fails, check your fibres and transceivers. Pay attention to any error messages.

7 Common Issues

- Problem: Vivado's GUI does not open.
  Solution: 1) make sure to ssh using -X.
  2) edit ~/.bashrc. comment the line *export DISPLAY=:0*.

- Problem: Receiving extra or unexpected packets in the hardware test.
  Solution: this is likely as the network manager is not disabled and/or the network is not configured properly. Follow the steps in `https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/Reference-Operating-System-Setup-Guide`, under the *Network Configuration Manager* section. In particular, make sure that your NIC's MAC address appears on the list of non managed interfaces (/etc/NetworkManager/NetworkManager.conf) and that **all** your NIC's host interfaces are set to manual (/etc/network/interfaces).

- Problem: SDNet simulation passes but SUME simulation fails.
  Make sure you followed the required steps:

```
cd $P4_PROJECT_DIR && make config_writes
cd $NF_DESIGN_DIR/test/sim_<major>_<minor> && make
cd $P4_PROJECT_DIR && make install_sdnet
```

Note that packet reorder may cause a failure. You should also consider running the simulation for a longer time if packets have not arrived. Make sure all configuration writes are completed before any packets are sent.

8 Useful links

- NetFPGA Repository: `https://github.com/NetFPGA/NetFPGA-SUME-live/`
- NetFPGA Wiki: `https://github.com/NetFPGA/NetFPGA-SUME-public/wiki`
- NetFPGA registration page: `https://netfpga.org/site/#/SUME_reg_form/`
- P4-NetFPGA Repository: `https://github.com/NetFPGA/P4-NetFPGA-live`
- P4-NetFPGA Wiki: `https://github.com/NetFPGA/P4-NetFPGA-public/wiki`
- P4-NetFPGA registration page: `https://goo.gl/forms/h7RbYmKZL7H4EaUf1`
- P4-NetFPGA online tutorial: `https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Tutorial-Assignments`