1 Setting up the environment

To develop with Centurion, you first need to clone the repository from GitHub, which requires setting up SSH keys.

1. First, connect to the development machine. E.g:

   ```
   ssh <username>@elecpc306.its.york.ac.uk
   ```

   The VPN must be used to connect remotely, but it is not required when on campus.

2. Generate an SSH key by running

   ```
   ssh-keygen -t ed25519 -C "your York email address"
   ```

   and accepting the default options.

3. `cd` into `~/.ssh` and create or edit `config`. Assuming the key file is called `id_ed25519`, add this to the file:

   ```
   Host *
       AddKeysToAgent yes
       IdentityFile ~/.ssh/id_ed25519
   ```

4. Run the following commands to add the new key to the SSH agent.

   ```
   chmod 0600 ~/.ssh/config
   eval "$(ssh-agent)"
   ssh-add -k ~/.ssh/id_ed25519
   ```

5. Next, copy the contents of `id_ed25519.pub` and import it into your GitHub account. (Settings→SSH and GPG keys)

6. You can now clone the repo. From the directory where you would like the project to exist, run

   ```
   git clone git@github.com:York-Bio-inspired-Systems-and-Tech/Centurion-VC707.git
   ```

7. Finally, move into the repo with `cd Centurion-VC707`

There are two sides to the software: the code running on the host, and that running on the nodes. Eclipse is used for developing the host PC software, while Xilinx Development environment (which is also based on Eclipse) is used for the nodes.

To open Eclipse:

1. The SSH session must be setup with X-Forwarding, eg:

   ```
   ssh <username>@elecpc306.its.york.ac.uk -X
   ```

2. Open eclipse

   ```
   eclipse &
   ```

3. Eclipse will then ask for a workspace, which is where the host PC project will be stored. Choose `$REPO_DIR/sw/host_pc/`

4. Eclipse integrates well with git, but the first time the workspace is opened it will need to be told where to find the projects. Go to "File → Import" and choose "General → Existing Projects into Workspace". Press "Next" and set the root directory to `$REPO_DIR/sw/host_pc`

5. This should bring up a list of projects. For this tutorial we only need to import the `centurion_lib` and `centurion_example_Host_PC` project. Ensure that these projects are selected and click "Finish".

To open Xilinx Development environment:

1. Run `source /opt/Xilinx/Vivado/2018.3/settings64.sh` so Bash knows where to find the tools. This needs to be run from every new shell. Alternatively, add the command to `~/.bash_profile` so it is run automatically on login.

2. Open the IDE

   ```
   xsdk &
   ```

3. As with Eclipse, you will be asked for a workspace. This time, choose `$REPO_DIR/sw/node_sw`

4. Once again we have to tell Eclipse where to find the projects. Go to "File → Import" and choose "General → Existing Projects into Workspace". Press "Next" and set the root directory to `$REPO_DIR/sw/node_sw/`.
5. This should bring up a list of projects. We need to import the HW description project `centurion_node_hw`, the board support package `standalone_bsp_0` and the example project `centurion_example`. Ensure that these projects are selected and click "Finish".
6. The project may be showing in an error state. This is because the BSP is not built when it comes out of the repo. Build the BSP project by selecting "Project → Build All".

2 Programming the Centurion machine

The code for the nodes is compiled to an ELF file, which is then uploaded to the boards.

1. By default, XSDK builds the project automatically - this can be turned off from "Project → Build Automatically" if preferred. If this is turned off, just press the build icon in the toolbar.
2. The .elf file is output to the projects Debug folder, so from the command line, CD to this folder.
3. Load the ELF onto the Centurion machine (replace example with the project name):

   ```
   centurion_load_ELF centurion_example.elf
   ```
4. Lots of output will scroll past, as long as the command exits after programming node 63 the upload has been successful.

3 Monitoring node output

There are a few ways that the output can be observed.

The example program can be run on the host from the project's Debug directory: `./centurion_example_Host_PC`. This sends a message to a node, receives a message back and prints out the result, which is one way that output can be observed.

Each node has a debug register, which in the examples is memory mapped to the variable DEBUG_OUT. To see the debug output for all the nodes, run the command `./centurion_example_Host_PC`

Finally, the output of one node can be monitored, allowing for `xil_printf()`'s on the node to be observed. For example, to monitor node number 5, run the following three commands:

```
centurion_set_UART 5
stty -F /dev/ttyUSB0 speed 921600
cat < /dev/ttyUSB0
```

As this is annoying to type each time, I created a function in my `.bashrc` file:

```
monitornode(){
   centurion_set_UART $1
   stty -F /dev/ttyUSB0 speed 921600
   cat < /dev/ttyUSB0
}
```

That way, I can just type this in the terminal, e.g: `monitornode 5`

4 Random useful things

1. The rand() function works as expected on the nodes, except it can't be seeded from a clock. Instead, choose the seed on the host and send it to the nodes
2. There is no real time clock, but there is a counter that can be used for timing: `Read_RTC()` returns the current value of this counter.
3. Make sure that the initial barrier sync isn't removed and no code is before the initialisation - otherwise it isn't possible to reprogram the nodes.

4. Use the Xilinx library on the nodes, e.g `xil_printf` and `Xuint8` / `Xuint16` instead of `printf` and `int`.
5. I end up with multiple terminals open, so I added this function to my `.bashrc` file to help:

   ```
   title(){
      ORIG=$PS1
      TITLE="\e]2;$@\a"
      PS1=${ORIG}${TITLE}
   }
   ```

   This allows me to change the title of the terminal, e.g. `title myTerminal`