# CASPER Toolflow

*Release 0.1*

Collaboration for Astronomy Signal Processing and Electronics R

Oct 19, 2020

## Setup

Welcome to the documentation for `mlib_devel`, the CASPER Toolflow!

CHAPTER 1

# What is mlib_devel?

The `mlib_devel` repository contains a set of FPGA DSP libraries and programming tools developed and maintained by the Collaboration for Astronomical Signal Processing and Electronics Research (CASPER). Within the collaboration, this collection of software is affectionately referred to as *The Toolflow*.

The CASPER toolflow allows you to generate signal processing designs using MATLAB's graphical programming tool `Simulink`. These designs can be turned into FPGA bitstreams and loaded onto a variety of supported hardware platforms to perform real-time digital signal processing systems. CASPER also provides a Python software library for interacting with running designs: casperfpga.

# CHAPTER 2

# Using mlib_devel

For more information about installing and using the CASPER Toolflow, see the project's documentation.

CASPER also maintain a set of tutorials, designed to introduce new users to the toolflow.

CHAPTER 3

# Updating an Existing Toolflow Installation

You can always update your installation of *mlib_devel* by pulling updated code from this repository. If you do this, chances are you'll need to update your Simulink models to match your new *mlib_devel* libraries. A script is provided to automate this process. With your model open and active, in your MATLAB prompt, run

```
update_casper_blocks(bdroot)
```

This script will resynchronize every CASPER block in your design with its latest library version. Depending on the size of your model, it may take many minutes to complete! As always, back up your designs before attempting such a major operation. And, if you experience problems, please raise Github issues!

CHAPTER 4

# *mlib_devel* directory structure

***casper_library*** Simulink DSP libraries

***xps_library*** Simulink libraries for tool-flow supported modules (ADC interfaces, Ethernet cores, etc.)

***xps_base*** HDL code and Xilinx EDK wrappers used in older (ROACH2 and earlier) versions of the toolflow.

***docs*** Sphinx documentation for the software in this project.

***jasper_library*** Python and MATLAB scripts required to drive the compilation process. Also platform-dependent configuration information and source-code for IP modules used by the toolflow in the following directories.

- ***platforms*** YAML files defining the compile parameters and physical constraints of CASPER-supported FPGA platforms.
- ***golden*** Golden boot images for FPGA platforms which require them.
- ***hdl_sources*** HDL source files for all toolflow-suppled modules (eg. ADC interfaces, Ethernet cores, etc.).
- ***sw*** Codebase for embedded software processors used by the toolflow
- ***yellow_blocks*** Python classes for each yellow block in the simulink *xps_library*.

## 4.1 Setup

The software stack you will require to use the toolflow will depend what hardware you are targeting. Older hardware (ROACH2 and earlier) use the older Xilinx software (ISE) which forces the use of different tools.

The current compatibility matrix is below:

(Note that official support for ROACH plaforms is no longer provided, however this version of *mlib_devel* contains all ROACH related documentation and ROACH tutorials can be found here)

| Hardware | Operating System | Matlab Version | Xilinx Version | mlib_devel branch / commit | Python Version |
|---|---|---|---|---|---|
| ROACH1/2 | Ubuntu 14.04 | 2013b | ISE 14.7 | branch: *roach* | Python 2.7 |
| SKARAB | Ubuntu 16.04 | 2018a | Vivado 2019.1.1 | branch: *master* | Python 3 |
| SNAP | Ubuntu 16.04 | 2018a | Vivado 2019.1.1 | branch: *master* | Python 3 |
| Red Pitaya | Ubuntu 16.04 | 2018a | Vivado 2019.1.1 | branch: *master* | Python 3 |
| VCU118 | Ubuntu 16.04 | 2018a | Vivado 2019.1.1 | branch: *master* | Python 3 |
| VCU128 | Ubuntu 16.04 | 2018a | Vivado 2019.1.1 | branch: *master* | Python 3 |
| ZCU111 | Ubuntu 16.04 | 2018a | Vivado 2019.1.1 | branch: *master* | Python 3 |
| SNAP2 | Ubuntu 16.04 | 2016b | Vivado 2016.4 | branch: *master* | Python 3 |

The recommended OS is Ubuntu as it is what the majority of the collaboration are using. This makes it easier for us to support you. If you are so inclined, you could also use Red Hat, but we definitely do not support Windows. You are welcome to try but you will be on your own. You could always run Linux in a VM although this will increase your compile times.

Please refer to the setup links below for more information on setting up the toolflow.

### 4.1.1 Setup Links

1. *Installing the Toolflow*
2. *Installing Matlab*
3. *Installing Xilinx Vivado*
4. *Configuring the Toolflow*
5. *Running the Toolflow*

#### Installing the Toolflow

This page explains how to install the CASPER tools and what supporting software is required to run them.

#### Getting the right versions

The toolflow is very sensitive to mis-matching software versions. The current compatibility matrix of software versions is below:

*(Note that official support for ROACH plaforms is no longer provided, however this version of `mlib_devel` contains all ROACH related documentation and ROACH tutorials can be found here)*

Other software combinations may work, but these are the tested configurations. The master branch is usually updated once a year. Between updates, code with newer features can be found in the `casper-astro-soak-test` branch. This branch can usually be used in place of the `master` branch for platforms which support `master`. However, be aware that `casper-astro-soak-test` is likely to be less stable. Please report any bugs you encounter via github's issue tracker.

### Pre-requisites

1. MATLAB

   MATLAB installation instructions are available here, or, contact whoever manages your software installations. You will need to install both MATLAB and Simulink.

2. Xilinx Vivado

   This is available from xilinx.com and will require a license. If you are part of an academic institution you may be eligible for free licenses via the Xilinx University Program. Vivado install instructions are available here.

3. Python

   Compiling for supported platforms requires Python 3 and `pip3`. If you don't have these already you can probably install them in Ubuntu environments by opening a terminal and running the command `apt-get install python3 python3-pip`.

   We thoroughly recommend using a virtual environment to separate the version of Python and its libraries the toolflow uses from the rest of your system.

   To create a Python 3 virtual environment:

   ```
   # change directory to where you want the virtual environment to live
   cd /home/user/work
   # install virtualenv using pip3
   sudo pip3 install virtualenv
   # create a Python 3 virtual environment
   virtualenv -p python3 casper_venv
   # to activate the virtual environment:
   source casper_venv/bin/activate
   # to deactivate the virtual environment:
   deactivate
   ```

### Obtaining the Toolflow

Clone the toolflow from the mlib_devel git repository.

```
# Clone the mlib_devel repository. Replace <branch_name> with the branch
# supported by your chosen platform.
# Eg. for master you should run:
# git clone -b master https://github.com/casper-astro.mlib_devel
git clone -b <branch_name> https://github.com/casper-astro/mlib_devel
```

This could take a while – the repository is several hundred megabytes. If you want, you can save some time by adding the `--depth=1` flag to the above command. This will only download the current version of the repository, rather than its full git history.

Next, move into the `mlib_devel` repository you have just created, activate your virtual environment (if using one) and download any Python dependencies you need by installing the requirements.txt file. The downloaded dependencies will be installed within the virtual environment separate to the rest of your system.

```
cd mlib_devel
source /home/user/work/casper_venv/bin/activate
pip3 install -r requirements.txt
```

You may need to run the `pip3 install` command as an administrator if you are using the system-maintained python installation instead of a virtual environment.

### Configuring the toolflow

You now have all the software you need to start building your designs. However, you'll still need to specify some local configuration details which will depend on how you carried out your installation. See Configuring the Toolflow for more details.

### How to install Matlab

This section explains How To install Matlab R2013b and R2016b.

### How to Install R2013b

1. OS Required/suggested: Ubuntu 14.04 LTS
2. Ubuntu 14.04. Using Nautilius, click on R2013b_UNIX.iso and extract to "Installs/Matlab2013b".
3. Ubuntu 14.04. Open a terminal < ctrl + alt + T>. You will need to install the JRE (Java Runtime Environment) if you don't have it. Type `sudo apt-get install openjdk-7-jre` at the prompt and press enter.
4. Terminal: You will now need to backup file "libstdc++.so.6" and link to file "libstdc++.so.6.0.13". Type in `cd ~/Installs/Matlab2013b/bin/glnxa64` and press enter.
5. Terminal: Type `sudo mv libstdc++.so.6 libstdc++.so.6_bu` and enter. The file should now be backed up.
6. Terminal: Type `sudo ln -s libstdc++.so.6.0.13 libstdc++.so.6` and press enter. The file should now be linked.
7. Terminal: set the matlab environment variable to call java 7. Type `export MATLAB_JAVA="/usr/lib/jvm/java-7-openjdk-amd64/jre` and press enter. Type `echo $MATLAB_JAVA` to make sure the new path is set.
8. Terminal: Make sure the java is executable. Type `cd ~/Installs/Matlab2013b` and press enter. Type `chmod +x sys/java/jre/glnxa64/jre/bin/java` and press enter.
9. Terminal: You will now need to invoke the installer. Type `sudo ./install -javadir /usr/lib/jvm/java-7-openjdk-amd64/jre` at the prompt and press enter.
10. The MathWorks Installer GUI should pop up. Select "Install without using the Internet" and select "Next".
11. You will be requested to sign the "License Agreement" page. Click "Yes" and then click "Next".
12. You will be requested to fill in the file installation key for your license. The Matlab Administrator should of provided a license and file installation key. If not, make sure you get one from him/her. Type in the file installation key and press "Next". I choose to install my license file under "~/Matlab".
13. You will then be requested for the "Installation Type". Click on the "Typical" radio button and press "Next".
14. You will then need to specify the installation folder. I choose "/opt/Matlab/R2013b". Press "Next". If the folder does not exist then click "Yes" to create it.
15. You will then be required to confirm your installation settings. If happy then press "Install" else press "Back" and then return to this step when happy.
16. You will be informed that your installation may require additional configuration skips. This can be ignored. Click "Next".
17. You will be informed that the installation is complete. Make sure Activate Matlab is ticked and click "Next".

18. It is now time to Activate MathWorks Software. A "MathWorks Software Activation" window will pop up. Click on the "Activate manually without the internet" and press "Next".
19. Click on the "Enter the full path to your license file, including the file name:" and browse to the license file (*.lic) and click "Select". Then press "Next". If all goes well then you will receive a message that says "Activation is complete.". Click "Finish".
20. Open another terminal and navigate to the "opt" folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> Matlab -R`
21. Terminal: Navigate to the "home" folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> .matlab -R`
22. It will be a good idea to create an Matlab R2013b startup script file on your Desktop with the following lines:

```
#!/bin/bash
cd /opt/Matlab/R2013b/bin/
./matlab
```

```
NB: Make sure the file is executable and that the nautilius documentation
navigator is set to run the script.
```

1. Run the script and the Matlab IDE will launch. You can now select the required Matlab m files (*.m) and continue.
2. In order to run the ISE-flow of the CASPER tools, please see here

### How To Install R2016b

1. OS Required/suggested: Ubuntu 14.04 LTS, Ubuntu 16.04 LTS (with tweaks), Red Hat 6.6 (Santiago).
2. Ubuntu 14.04. Using Nautilius, click on "R2016b_glnxa64_dvd1.iso" and extract to "Installs/Matlab2016b".
3. Ubuntu 14.04. Open a terminal < ctrl + alt + T> and type `cd ~/Installs` and then type `chmod +w Matlab2016b/ -R`. This will give all the files in the Matlab2016b folder write access.
4. Ubuntu 14.04. Using Nautilius, click on "R2016b_glnxa64_dvd2.iso" and extract to "Installs/Matlab2016b".
5. Open a terminal <ctrl + alt + T> and type `cd  ~/Installs/Matlab2016b/`,then `sudo ./install` and enter.
6. The MathWorks Installer GUI should pop up. Select "Use a File Installation Key" and select "Next".
7. You will be requested to sign the "License Agreement" page. Click "Yes" and then click "Next".
8. You will be requested to fill in the file installation key for your license. The Matlab Administrator should of provided a license and file installation key. If not, make sure you get one from him. Type in the file installation key and press "Next". I choose to install my license file under "~/Matlab".
9. You will then need to specify the installation folder. I choose "/opt/Matlab/R2016b". Press "Select" and then "Next".
10. You will then see a "Product Selection" window. Make sure that all products are ticked and select "Next".
11. You must then decide where you want the symbolic links to your Matlab scripts to be stored. I chose the default location "/usr/local/bin".
12. You will then be required to confirm your installation settings. If happy then press "Install" else press "Back" and then return to this step when happy.
13. You will be informed that your installation may require additional configuration skips. This can be ignored. Click "Next".

14. You will be informed that the installation is complete. Make sure Activate Matlab is ticked and click "Next".
15. It is now time to Activate MathWorks Software. A "MathWorks Software Activation" window will pop up. Click on the "Activate manually without the internet" and press "Next".
16. Click on the "Enter the full path to your license file, including the file name:" and browse to the license file (*.lic) and click "Select". Then press "Next". If all goes well then you will receive a message that says "Activation is complete.". Click "Finish".
17. Open another terminal and navigate to the "opt" folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> Matlab -R`
18. Terminal: Navigate to the "home" folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> .matlab -R`
19. It will be a good idea to create an Matlab R2016b startup script file on your Desktop with the following lines:

```
#!/bin/bash
cd /opt/Matlab/R2016b/bin/
./matlab
```

NB: Make sure the file is executable and that the nautilius documentation navigator is set to run the script.

1. Run the script and the Matlab IDE will launch. You can now select the required Matlab m files (*.m) and continue.
2. In order to run Matlab with the Casper tools please look at the CASPER read the docs page: https://casper-toolflow.readthedocs.io/en/latest/jasper_documentation.html.

### How To Install R2018a

1. OS Required/suggested: Ubuntu 14.04 LTS/Ubuntu 16.04 LTS.
2. Ubuntu 16.04. Using Nautilius, click on "R2018a_glnxa64_dvd1.iso" and extract to "Installs/Matlab2018a".
3. Ubuntu 16.04. Open a terminal < ctrl + alt + T> and type `cd ~/Installs` and then type `chmod +w Matlab2018a/ -R`. This will give all the files in the Matlab2018a folder write access.
4. Ubuntu 16.04. Using Nautilius, click on "R2018a_glnxa64_dvd2.iso" and extract to "Installs/Matlab2018a".
5. Open a terminal <ctrl + alt + T> and type `cd  ~/Installs/Matlab2018a/`,then `sudo ./install` and enter.
6. The MathWorks Installer GUI should pop up. Select "Use a File Installation Key" and select "Next".
7. You will be requested to sign the "License Agreement" page. Click "Yes" and then click "Next".
8. You will be requested to fill in the file installation key for your license. The Matlab Administrator should of provided a license and file installation key. If not, make sure you get one from him. Type in the file installation key and press "Next". I choose to install my license file under "~/Matlab".
9. You will then need to specify the installation folder. I choose "/opt/Matlab/R2018a". Press "Select" and then "Next".
10. You will then see a "Product Selection" window. Make sure that all products are ticked and select "Next".
11. You must then decide where you want the symbolic links to your Matlab scripts to be stored. I chose the default location "/usr/local/bin".
12. You will then be required to confirm your installation settings. If happy then press "Install" else press "Back" and then return to this step when happy.

13. You will be informed that your installation may require additional configuration skips. This can be ignored. Click "Next".
14. You will be informed that the installation is complete. Make sure Activate Matlab is ticked and click "Next".
15. It is now time to Activate MathWorks Software. A "MathWorks Software Activation" window will pop up. Click on the "Activate manually without the internet" and press "Next".
16. Click on the "Enter the full path to your license file, including the file name:" and browse to the license file (*.lic) and click "Select". Then press "Next". If all goes well then you will receive a message that says "Activation is complete.". Click "Finish".
17. Open another terminal and navigate to the "opt" folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> Matlab -R`
18. Terminal: Navigate to the "home" folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> .matlab -R`
19. It will be a good idea to create an Matlab R2016b startup script file on your Desktop with the following lines:

```
#!/bin/bash
cd /opt/Matlab/R2018a/bin/
./matlab
```

NB: Make sure the file is executable and that the nautilius documentation navigator is set to run the script.

1. Run the script and the Matlab IDE will launch. You can now select the required Matlab m files (*.m) and continue.
2. Install the R2018a update pack: "r2018a-update-6.tar.gz" by unpacking the tar.gz file and following the install instructions "r2018a-updates-install-instructions.pdf" for linux.
3. In order to run Matlab with the Casper tools please look at the CASPER read the docs page: https://casper-toolflow.readthedocs.io/en/latest/jasper_documentation.html.

### How to install Xilinx Vivado

This section explains How To install Vivado 2016.2, 2016.4 and 2018.2.

### How to Install 2016.x

1. OS Required/suggested: Ubuntu 14.04 LTS, Ubuntu 16.04 LTS (with tweaks) and Red Hat 6.6 (Santiago). There was an issue using Ubuntu 12.04 LTS, which caused the DocNav utility to crash.
2. Click/double click on the `Xilinx_Vivado_SDK_2016.2_0605_1.tar.gz` file in the Ubuntu Nautilius document navigator and choose a folder to extract the files to. I use "home/Installs‘‘ in this document. If you use something different then remember to replace "Installs" with your directory name.
3. Ubuntu 14.04. Open a terminal < ctrl + alt + T>. Change directory to the following folder: `cd Installs/Xilinx_Vivado_SDK_2016.2_0605_1`
4. Terminal: Type `sudo ./xsetup` and press enter. This application needs to be installed with root privileges otherwise the installation will not install properly. You will be prompted for the sudo password. Enter this and press enter.
5. The Vivado Installer GUI will pop up. The GUI might explain that there is a new version available, but ignore that and press "Continue" and then "Next" to commence with the installation process.
6. Read the terms and conditions page and when happy tick "I agree" for all three tick boxes. Then click "Next".

7. Select the "Vivado HL_System Edition" radio button and select "Next".
8. You will then be required to select which tools you want to install with the Vivado Design Edition. I selected "Software Development Kit", "Ultrascale+" and "Zynq UltraScale + MPSoC". The rest of the boxes were ticked (except Cable Drivers),so I have decided to install the complete set of tools available. Make sure that "DocNav" is ticked if you want access to the documentation that Xilinx has provided. This is highly recommended, as the documentation is part of the Ultrafast design methodology. Press "Next".
9. Select where you want to install the Vivado tool set. I am using the default `opt/Xilinx` folder. I have also ticked the "Create program group entries" and "create desktop shortcuts" buttons. This is not necessary though. Press "Next".
10. A window will pop up offering to create the `opt/Xilinx` directory if it does not exist. Select "Yes".
11. A window with the "Installation Summary" will be displayed showing what tools will be installed and where they will be stored on your drive. If you are happy press "Install", otherwise press "Back" and edit your previous settings.
12. Wait until the Xilinx Software Install window states that the "Installation completed successfully" and select "OK".
13. Open another terminal and navigate to the `opt/` folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> Xilinx -R`
14. Terminal: Navigate to the `home` folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> .Xilinx -R`
15. DocNav will not work unless you follow the steps highlighted in the text file: set_up_vivado_2015.1_on_ubuntu_14.04 (Doc Nav section only). Install the i386 architecture and then install the missing libraries. All the commands are highlighted in the attached file.
16. DocNav will now open, but you won't be able to open the documentation until you have made the following link. Using the terminal type in `cd /opt/Xilinx/Vivado/2016.2/ids_lite/ISE/lib/lin64` and press enter.
17. Using the terminal, type in `mv libstdc++.so.6 libstdc++.so.6bu` and press enter. Now type in `ln -s /usr/lib/x86_64-linux-gnu/libstdc++.so.6 ./libstdc++.so.6` and pressenter. The `libstdc++.so.6` file will now be properly linked and DocNav should work.
18. It will be a good idea to create a vivado startup script file on your Desktop with the following lines:

```
#!/bin/bash
cd /opt/Xilinx/Vivado/2016.2/bin/
./vivado
```

    NB: Make sure the file is executable and that the nautilius documentation navigator is set to run the script.
19. Run the script and the Vivado IDE will launch. You can now select the required Xilinx Vivado project file (*.xpr) and continue.
20. It is now time to install the license for Vivado. Create a `Xilinx` folder in your home directory using the nautilius documentation navigator: `home/<user name>/Xilinx` and copy the vivado license file provided by your administrator to this location.
21. Load the license using the Vivado License Manager". If not already open click on "Help" -> "Manage License. . . ". Click "Load License" and then click on "Copy License. . . ". Navigate to the license file (*.lic) in the `home/<user name>/Xilinx` folder. Press "Open" and when the license installation was successful then press "OK".
22. To confirm that the license file was successful, click on "View License Status" and make sure a list of Tools/IP is read back and that the license is still valid. Once this is done then close the "Vivado License Manager" by

clicking on the red cross at the top left of the window. You will be prompted if you want to close the "Vivado License Manager". Click "Yes".

### Optional: Install USB Drivers for JTAG

*Note: this will only be used by toolflow/yellow block developers and is not required for standard use of the toolflow*

The most reliable way to install the JTAG cable drivers is to use the drivers provided with ISE.

A folder containing all the files required has been uploaded in the same folder as this document: linux_jtag_cable_drivers.tar.gz. This also includes a useful installation script that prepares the files and places them in the correct directories.

Instructions:

- Extract the contents of the file linux_jtag_cable_drivers.tar.gz
- Run: `sudo ./install.sh` (NB!: Must run as sudo)

It may be a good idea to power your PC/lap top down and then up again as the USB drivers may not take affect until this happens. In my case, I plugged a stick drive into the USB and then ejected that and connected the Xilinx Platform Cable USB module. Once this was done then the status LED illuminated and I was able to configure the FPGA via JTAG.

### How to Install 2018.x

1. OS Required/suggested: Ubuntu 16.04 LTS.
2. Click/double click on the `Xilinx_Vivado_SDK_2018.2_0614_1954.tar.gz` file in the Ubuntu Nautilius document navigator and choose a folder to extract the files to. I use "home/Installs‘‘ in this document. If you use something different then remember to replace "Installs" with your directory name.
3. Ubuntu 16.04. Open a terminal < ctrl + alt + T>. Change directory to the following folder: `cd Installs/Xilinx_Vivado_SDK_2018.2_0614_1954`
4. Terminal: Type `sudo ./xsetup` and press enter. This application needs to be installed with root privileges otherwise the installation will not install properly. You will be prompted for the sudo password. Enter this and press enter.
5. The Vivado Installer GUI will pop up. The GUI might explain that there is a new version available, but ignore that and press "Continue" and then "Next" to commence with the installation process.
6. Read the terms and conditions page and when happy tick "I agree" for all three tick boxes. Then click "Next".
7. Select the "Vivado HL_System Edition" radio button and select "Next".
8. You will then be required to select which tools you want to install with the Vivado Design Edition. I selected "Software Development Kit", "Ultrascale+" and "Zynq UltraScale+ MPSoC". The rest of the boxes were ticked (except Cable Drivers), so I have decided to install the complete set of tools available. Make sure that "DocNav" is ticked if you want access to the documentation that Xilinx has provided. This is highly recommended, as the documentation is part of the Ultrafast design methodology. Press "Next".
9. Select where you want to install the Vivado toolset. I am using the default `opt/Xilinx` folder. I have also ticked the "Create program group entries" and "create desktop shortcuts" buttons. This is not necessary though. Press "Next".
10. A window will pop up offering to create the `opt/Xilinx` directory if it does not exist. Select "Yes".

11. A window with the "Installation Summary" will be displayed showing what tools will be installed and where they will be stored on your drive. If you are happy press "Install", otherwise press "Back" and edit your previous settings.
12. Wait until the Xilinx Software Install window states that the "Installation completed successfully" and select "OK".
13. Open another terminal and navigate to the `opt/` folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> Xilinx -R`
14. Terminal: Navigate to the `home` folder and remember to change user and group to your username with the following command: `sudo chown <username>:<username> .Xilinx -R`
15. If DocNav does not work, then try follow the steps highlighted in the text file: set_up_vivado_2015.1_on_ubuntu_14.04 (Doc Nav section only). Install the i386 architecture and then install the missing libraries. All the commands are highlighted in the attached file.
16. DocNav may open, but it is possible you won't be able to read the documentation until you have made the following link - first try and read the documentation via DocNav though. Using the terminal type in `cd /opt/Xilinx/Vivado/2016.2/ids_lite/ISE/lib/lin64` and press enter.
17. If you still can't access the documentation via DocNav: Using the terminal, type in `mv libstdc++.so.6 libstdc++.so.6bu` and press enter. Now type in `ln -s /usr/lib/x86_64-linux-gnu/libstdc++.so.6 ./libstdc++.so.6` and pressenter. The `libstdc++.so.6` file will now be properly linked and DocNav should work.
18. It will be a good idea to create a vivado startup script file on your Desktop with the following lines:

    ```
    #!/bin/bash
    cd /opt/Xilinx/Vivado/2018.2/bin/
    ./vivado
    ```

    NB: Make sure the file is executable and that the nautilius documentation navigator is set to run the script.
19. Run the script and the Vivado IDE will launch. You can now select the required Xilinx Vivado project file (*.xpr) and continue.
20. It is now time to install the license for Vivado. Create a `Xilinx` folder in your home directory using the nautilius documentation navigator: `home/<user name>/Xilinx` and copy the vivado license file provided by your administrator to this location.
21. Load the license using the Vivado License Manager". If not already open click on "Help" -> "Manage License…". Click "Load License" and then click on "Copy License…". Navigate to the license file (*.lic) in the `home/<user name>/Xilinx` folder. Press "Open" and when the license installation was successful then press "OK".
22. To confirm that the license file was successful, click on "View License Status" and make sure a list of Tools/IP is read back and that the license is still valid. Once this is done then close the "Vivado License Manager" by clicking on the red cross at the top left of the window. You will be prompted if you want to close the "Vivado License Manager". Click "Yes".

### Optional: Install USB Drivers for JTAG

*Note: this will only be used by toolflow/yellow block developers and is not required for standard use of the toolflow*

A folder containing all the files required has been uploaded in the same folder as this document: linux_jtag_cable_drivers.tar.gz. This also includes a useful installation script that prepares the files and places them in the correct directories.

Instructions:

- Extract the contents of the file linux_jtag_cable_drivers.tar.gz
- Run: `sudo ./install.sh` (NB!: Must run as sudo)

It may be a good idea to power your PC/lap top down and then up again as the USB drivers may not take affect until this happens. In my case, I plugged a stick drive into the USB and then ejected that and connected the Xilinx Platform Cable USB module. Once this was done then the status LED illuminated and I was able to configure the FPGA via JTAG.

### Configuring the Toolflow

If you have successfully installed the toolflow and its dependencies, it is now time to configure the flow to suit your specific environment. The toolflow needs to know where dependencies like MATLAB and Xilinx tools have been installed. Other site-dependent parameters may also need to be defined.

#### The `startsg` script

A startup script – `startsg` – is provided as part of the toolflow repository. This script can be used in two ways:

- If *executed* (i.e. `/path/to/mlib_devel/startsg`): start MATLAB with the correctly defined library paths.
- If *sourced* (i.e. `source /path/to/mlib_devel/startsg`): configure software paths without starting MATLAB.

The former method is what you should do if you want to start a Simulink design, or open an existing one.

The latter method is useful if you want to run parts of the toolflow outside of MATLAB (eg. `exec_flow.py`) or run Xilinx tools (eg. `vivado`) directly from the command line.

#### Specifying local details

The `startsg` script is generic. You should not need to modify it. The script does not require that the Matlab and Xilinx tools be installed in specific locations, but it does require that you provide it with a few details about your local installation. This is done by creating a `startsg.local` file that defines a few key variables needed by `startsg`. Two essential variables are:

- `MATLAB_PATH` - the path to the directory where MATLAB was installed
- `XILINX_PATH` - the path to the directory where Xilinx tools were installed

Optional variables:

- `PLATFORM` - Used by the Xilinx tools to select suitable runtime binaries for your system. If not specified, it will be defaulted to `lin64`, indicating a 64-bit Linux operating system. This is the only configuration the collaboration tests.
- `XILINXD_LICENCE_FILE` - The path to your Xilinx software license if it exists in a non-standard location.
- `JASPER_BACKEND` - the type of Xilinx tools you want to use to implement your design. Supported options are `vivado` or `ise`. The default is `vivado`, which is correct for all CASPER-supported platforms. *(Note: `ise` is the Xilinx tool used for ROACH1/ROACH2 designs, however official support for ROACH platforms is no longer provided).*
- `CASPER_PYTHON_VENV_ON_START` - The path to your Python virtual environment (if one is being used). This will activate the virtual environment on load.

Other variables: Depending on your operating system, and MATLAB / Xilinx quirks, you may need to specify other generic OS variables. For example, with MATLAB 2018a and Ubuntu 16.04, it is necessary to over-ride the default MATLAB libexpat library to a newer version. To do this you can set the `LD_PRELOAD` variable.

Here is a sample `startsg.local` file:

```
export XILINX_PATH=/opt/Xilinx/Vivado/2019.1
export MATLAB_PATH=/usr/local/MATLAB/R2018a
export PLATFORM=lin64
export JASPER_BACKEND=vivado

# over-ride the MATLAB libexpat version with the OS's one.
# Using LD_PRELOAD=${LD_PRELOAD}:"..." rather than just LD_PRELOAD="..."
# ensures that we preserve any other settings already configured
export LD_PRELOAD=${LD_PRELOAD}:"/usr/lib/x86_64-linux-gnu/libexpat.so"

# Activate a custom python environment on load
export CASPER_PYTHON_VENV_ON_START=/home/user/work/casper_venv
```

Since this configuration refers to your specific installation environment, in general it shouldn't be commited to the `mlib_devel` repository. In fact, the repository is configured to ignore changes to any files with names beginning `startsg.`. If you really want to commit your local configuration file, you can do this, but it's helpful to call it something other than `startsg.local`, (eg. `startsg.local.example` or `startsg.local.my-server-name`) so as not to conflict with other users, all of whom will have similar files with different contents.

### Using `startsg`

By default, executing (or sourcing) the `startsg` script will use variables defined in the configuration file `startsg.local` residing in the same directory as `startsg`. However, you can use a specific configuration by specifying one as an argument to `startsg`. This can be useful if you want to store configurations for multiple versions of MATLAB / Xilinx tools.

For example:

```
    $ ./startsg                         # Uses startsg.local if one exists in the␣
↪current directory

    $ ./mlib_devel/startsg              # Uses startsg.local if one exists in ./mlib_
↪devel/

    $ ./startsg startsg.local.use_vivado_2016  # Uses the startsg.local.use_vivado_16␣
↪configuration file
```

### Symlink for convenience

Running `startsg` from the `mlib_devel` directory (where it lives) will start MATLAB with `mlib_devel` as the current directory. Hopefully you store your models somewhere outside `mlib_devel` (which should contain only the CASPER *libraries*), in which case after running `startsg` you will need to navigate within MATLAB to the directory where your model files live. To avoid this minor annoyance, you can create a symbolic link to `startsg` in your application directory (i.e. where your model file lives). When running `startsg` via this symlink, MATLAB will start up with your application directory as the current directory and also run the optional `casper_startup.m` file if one exists.

To configure such a symlink you should run

```
# Go to the directory you store your models in.
# You should place a startsg.local file here.
cd /directory/where/my/models/are
# Create the symbolic link
ln -s /directory/where/mlib_devel/is/startsg startsg
# Run startsg from your model directory
./startsg my.startsg.local
```

This model of operating works particularly well when using git submodules to store a copy of `mlib_devel` alongside your models. Using submodules in this way ensures that whoever downloads your models can also easily obtain the version of `mlib_devel` they were originally compiled against.

In this case, your directory structure will look something like:

```
my_spectrometer/
├── my_spectrometer.slx
├── startsg.local
├── startsg (symlink -> ./mlib_devel/startsg)
└── mlib_devel (submodule)
    ├── startsg
    ├── casper_library
    ├── ...
    ├── ...
    └── ...
```

### Running the Toolflow

1. There are two ways of working with the Vivado-based CASPER toolflow. You can do this initially with the MATLAB GUI to compile the front end and then handle the middleware and backend generation using Python or you can run everything in Python. The former stage is more for design and debugging (steps 2-11) and the later stage (steps 12-15) is for the final tested and working design. This How-to will cover both methods.
2. **Matlab/Python method:** Using the terminal, type the following under "mlib_devel": `./startsg` It is currently not possible to compile for Altera or Lattice FPGAs. This script will source all the relevant Matlab and Xilinx paths, run matlab and start the system generator. Wait until the Matlab GUI has opened and Matlab is ready.
3. **Matlab/Python method:** In the Matlab command window, type the following: `simulink`. This will start simulink. Wait until the Simulink window has opened.
4. **Matlab/Python method:** In the Simulink Library Browser, click on the "open model or library" icon in the tab and select where your desired simulink file is (*.slx). There are some test files under "jasper_library/test_models". I use "test_snap.slx" for this How To. Once the file has been selected, click "Open". The "test_snap" design should open in the Simulink window.
5. **Matlab/Python method:** Click the simulink design window ("test_snap") and press the following: "Ctrl + D". This will update the simulink model and check for warnings or errors. Make sure there are no errors or warnings. A window should pop up if this is the case.
6. **Matlab/Python method:** In the Matlab command window terminal, type the following: `jasper_frontend`. This will generate the yellow block peripheral file and run the system generator. Wait until the "XSG generation complete. Complete. Run 'exec_flow.py -m . . . .'" message is displayed.
7. **Matlab/Python method:** In the Matlab Command Window, cut the following text from it: `python ../exec_flow.py -m ...  --middleware --backend --software ...`. The matlab generation process is now complete and now it is time to switch to Python.

8. **Matlab/Python method:** Open a new terminal <CTRL+ALT+T>, and source the following files from the "mlib_devel" directory: `startsg` This is an important step, because the Xilinx and Matlab paths will not be specified properly and "exec_flow.py" will fail to run if this is not done.
9. **Matlab/Python method:** Using the terminal, paste the "python exec_flow.py. . . ." command that was cut earlier from Matlab, in the terminal: `python exec_flow.py -m ... --middleware --backend --software`. This command will execute the middleware, which calls the yellow block constructors, creates the top.v file and generates the yaml file, which contains all the parameters needed for the backend to compile. The backend reads the yaml file and builds a list of sources, constraints, generates the constraints file and the tcl file. This tcl file is used by Vivado to compile the top.v file and all other relevant source files. This generates a bit and binary file, which is used to configure the FPGA. The software reads the binary file and generates a bof and fpg file. The arguments passed to exec_flow.py will be explained in more detail below when dealing with the Python method.
10. **Matlab/Python method:** Using the terminal, wait until the design has finished compiling. Vivado compiles should indicate that there are no timing violations. Check the slack times for the setup and hold reports. They should not be negative. If they are then your design is not meeting timing and some changes will need to be made to your design.
11. **Matlab/Python method:** The output directories are generated where the *.slx file sits. For example, building for `test_snap.slx` results in the following directories being generated under `jasper_library/test_models/test_snap/`:
    - `sysgen/`: contains the system generator files,
    - `outputs/`: contains the bof and fpg files, and
    - `myproj/`: contains the Vivado projects files, source files, synth results and implementation results. The bin and bit files are also stored here.

    **NB:** Instead of running "jasper_frontend" from the Matlab command window, you can run "jasper", which will do all the above steps from 6) to now, but all display output will be routed through the Matlab Command window.
12. **Python method:** Before I explain this method it is important to explain how the "exec_flow" command works and the arguments that are passed to it.
    - The `exec_flow`, which stands for "execution flow" can either run the whole flow or just parts of the flow depending on the needs of the user.
    - The Vivado compile is done using project mode only.
    - I have already explained the `--middleware`, `--backend` and `--software` arguments in step 9) above.
    - There is also a `--perfile` and `--frontend` argument, which is not needed in the Matlab/Python method, but is required for the Python method.
    - The `--perfile` and `--frontend` arguments run the yellow block peripheral file generation and the system generator compile, respectively. It is identical to running `jasper_frontend` from the command window in Matlab - see Matlab/Python method above.
    - Below is a list of the `exec_flow` arguments.
      - `--perfile` - Runs the front end peripheral file generation. If not specified, then it won't generate the peripheral file.
      - `--frontend` - This compiles the front end IP, which basically runs the system generator. If not specified, then the compile will not be run.
      - `--middleware` - This runs the toolflow middle process. If not specified, then this process will not be run.

- `--backend` - This runs the backend compilation i.e. Xilinx Vivado. If not specified, then this process will not be run.
- `--software` - This runs the software compilation - generates a *.bof and *.fpg file. If not specified, then this process will not be run.
- `--be` - This specifies the type of backend to be run. This is "–be vivado", but provision has been made for other backends. If this is not specified, then the default is the Vivado backend.
- `--jobs` - The number of processor cores to run the compile with. If this is not specified, the default is 4. You need to make sure that your processor has at least 4 threads if this is to work.
- `-m` - The absolute path and filename of the *.slx file (Simulink model) to compile. If not specified, the default is "/tools/mlib_devel/jasper_library/test_models/test.slx". I would suggest always specifying this.
- `-c` - This is the build directory. The default is the same directory as the *.slx file (Simulink model). I don't normally specify this.
- `--synth_strat` - Specify a Synthesis Strategy for your compile. The options are as follows, as provided by Vivado 2019.1.1:
  * Flow_AreaOptimized_high
  * Flow_AreaOptimized_medium
  * Flow_AreaMultThresholdDSP
  * Flow_AlternateRoutability
  * FFlow_PerfOptimized_high
  * Flow_PerfThresholdCarry
  * Flow_RuntimeOptimized
- `--impl_strat` - Specify an Implementation Strategy for your compile. The options are as follows, as provided by Vivado 2019.1.1:
  * Performance_Explore
  * Performance_ExplorePostRoutePhysOpt
  * Performance_ExploreWithRemapx
  * Performance_WLBlockPlacement
  * Performance_WLBlockPlacementFanoutOpt
  * Performance_EarlyBlockPlacement
  * Performance_NetDelay_high
  * erformance_NetDelay_low
  * Performance_Retiming
  * Performance_ExtraTimingOpt
  * Performance_RefinePlacement
  * Performance_SpreadSLLs
  * Performance_BalanceSLLs
  * Performance_BalanceSLRs
  * Performance_HighUtilSLRs

- Congestion_SpreadLogic_high
- Congestion_SpreadLogic_medium
- Congestion_SpreadLogic_low
- Congestion_SSI_SpreadLogic_high
- Congestion_SSI_SpreadLogic_low
- Area_Explore
- Area_ExploreSequential
- Area_ExploreWithRemap
- Power_DefaultOpt
- Power_ExploreArea
- Flow_RunPhysOpt
- Flow_RunPostRoutePhysOpt
- Flow_RuntimeOptimized
- Flow_Quick

Here are some examples of how to run the command:

This will run the whole process, except will not generate a fpg and bof file for programming.

```
python .../exec_flow.py -m /home/<username>/mlib_devel/jasper_library/test_models/
↪test_snap.slx --perfile --frontend --middleware --backend
```

This will run the whole process.

```
python .../exec_flow.py -m /home/<username>/mlib_devel/jasper_library/test_models/
↪test_snap.slx --perfile --frontend --middleware --backend --software
```

This will run the front end peripheral file generation and IP compile process using the Vivado system generator.

```
python .../exec_flow.py -m /home/<username>/mlib_devel/jasper_library/test_models/
↪test_snap.slx --perfile --frontend
```

13. **Python method:** Open a new terminal <CTRL+ALT+T>, and source the following files from the `mlib_devel` directory:

    - `source startsg startsg.local`
    - This is an important step, because the Xilinx and Matlab paths will not be specified properly and `exec_flow.py` will fail to run.

14. **Python method:** Using the terminal, run the complete “exec_flow” command:

```
python .../exec_flow.py -m /home/<username>/mlib_devel/jasper_library/test_models/
↪test_snap.slx --perfile --frontend --middleware --backend --software
```

Feel free to add or remove arguments as you wish or need. The design should run through the toolflow generation process to completion. Once complete, the Vivado compile should report any errors, e.g. timing issues. The Vivado compile will determine if timing is met or not and display this to the screen. The user will need to monitor the slack time variable to see whether the compile has met timing or not. If the slack time is negative then timing is not met and if the slack time is positive for both setup and hold timing then the design has met the timing requirements.

15. **Python method:** The output directories are generated where the *.slx file sits. I used "test_snap.slx", hence the following directories were generated under `jasper_library/test_models/test_snap/`:
    - `sysgen/`: contains the system generator files,
    - `outputs/`: contains the bof and fpg files, and
    - `myproj/`: contains the Vivado projects files, source files, synthesis results and implementation results. The bin and bit files are also stored here.

## 4.2 Documentation

- CASPER Tutorials
- *Block Documentation*
- *Toolflow Documentation*
- *Toolflow Sourcecode*
- casperfpga Sourcecode

### 4.2.1 Block Documentation

**Contents**

#### Signal Processing Blocks

*adder_tree* (Adder Tree)

*barrel_switcher* (Barrel Switcher)

*bit_reverse* (Bit Reverser)

*cmult_4bit_br** (Conjugating Complex 4-bit Multiplier Implemented in BlockRAM)

*cmult_4bit_br* (Complex 4-bit Multiplier Implemented in BlockRAM)

*cmult_4bit_em** (Conjugating Complex 4-bit Multiplier Implemented in Dedicated Multipliers)

*cmult_4bit_em* (Complex 4-bit Multiplier Implemented in Embedded Multipliers)

*cmult_4bit_sl** (Conjugating Complex 4-bit Multiplier Implemented in Slices)

*cmult_4bit_sl* (Complex 4-bit Multiplier Implemented in Slices)

*complex_addsub* (Complex Adder/Subtractor)

*c_to_ri* (Complex to Real/Imaginary)

*DDS* (Direct Digital Synthesizer)

*dec_fir* (Decimating FIR Filter)

*delay_bram_en_plus* (Enabled Delay in BlockRAM))

*delay_bram_prog* (Programmable Delay in BlockRAM)

*delay_bram* (Delay in BlockRAM)

*delay_complex* (Complex Delay)

*delay_slr* (Delay in SLRs)

*delay_wideband_prog* (Programmable Wideband Delay Implemented in BlockRAM)

*dram_vacc* (DRAM Vector Accumulator)

*dram_vacc_tvg* (DRAM Vector Accumulator Test Vector Generator)

*edge* (Edge Detect Block)

*fft_biplex_real_2x* (Real-sampled Biplex FFT, with Output Demuxed by 2)

*fft_biplex_real_4x* (Real-sampled Biplex FFT, with Output Demuxed by 4)

*fft* (Complex FFT)

*fft_wideband_real* (Real-sampled Wideband FFT)

*finedelay_fstop_prog* (Programmable Fine delay along with Fringe Stop)

*finedelay_fstop_prog_cordic* (Programmable Fine delay with Fringe Stop using CORDIC block)

*fir_col* (PFB FIR Column)

*fir_dbl_col* (PFB FIR Double Column)

*fir_tap* (PFB FIR Tap)

*freeze_cntr* (Freeze Counter)

*lo_const* (DC Local Oscillator)

*lo_osc* (Local Oscillator)

*mixer* (Mixer)

*negedge* (Negative Edge Detector)

*partial_delay* (Partial Delay)

*pfb_fir_real* (Real-sampled Polyphase FIR Filter Frontend for PFB)

*pfb_fir* (Polyphase FIR Filter Frontend for PFB)

*posedge* (Positive Edge Detector)

*power* (Complex Data Power Calculator)

*pulse_ext* (Pulse Extender)

*rcmult* (Real/Complex Multiplier)

*reorder* (Arbitrary Reorderer)

*ri_to_c* (Real/Imaginary to Complex)

*square_transposer* (Square Transposer)

*stopwatch* (Stopwatch)

*sync_delay_en* (Enabled Sync Delay)

*sync_delay_proc* (Programmable Sync Delay) *sync_gen* (Parameterized Sync Generator)

*win_x_engine* (Windowed X-Engine)

*xeng_tvg* (X-Engine Test Vector Generator)

## Adder Tree

**Block:** Adder Tree (`adder_tree`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

Sums all inputs using a tree of adds and delays.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| No. of inputs. | n_inputs | The number of inputs to be summed. |
| Add Latency | latency | The latency of each stage through the adder tree. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | in | Boolean | Indicates the next clock cycle containing valid data |
| din | in | Inherited | A number to be summed. |

### Description

Sums all inputs using a tree of adds and delays. Total latency is $ceil(log_2(n_inputs)) * latency$.

### Barrel Switcher

**Block:** Barrel Switcher (`barrel_switcher`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

#### Contents

#### Summary

Maps a number of inputs to a number of outputs by rotating In(N) to Out(N+M) (where M is specified on the sel input), wrapping around to Out1 when necessary.

#### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Number of inputs | n_inputs | The number of parallel inputs (and outputs). |

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | in | Boolean | Indicates the next clock cycle contains valid data |
| In | in | Inherited | The stream(s) to be transposed. |
| sync_out | out | Boolean | Indicates that data out will be valid next clock cycle. |
| Out | out | Inherited | The transposed stream(s). |

#### Description

Maps a number of inputs to a number of outputs by rotating In(N) to Out(N+M) (where M is specified on the sel input), wrapping around to Out1 when necessary.

### Bit Reverser

**Block:** Bit reverser (`bit_reverse`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

#### Summary

Reverses the bit order of the input. Input must be unsigned with binary point at position 0. Costs nothing in hardware.

#### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| No. of bits. | n_bits | Specifies the width of the input. |

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | UFix_x_0 | The input signal. |
| out | out | UFix_x_0 | The output. |

#### Description

Reverses the bit order of the input. Input must be unsigned with binary point at position 0. Costs nothing in hardware.

### Conjugate Complex 4-bit Multiplier BRAM

**Block:** Conjugating Complex 4-bit Multiplier Implemented in Block RAM (`cmult_4bit_br*`)
**Block Author**: ?
**Document Author**: ?

### Contents

### Summary

Perform a conjugating complex multiplication $(a + bi)(c \;\; di) = (ac + bd) + (bc \;\; ad)i$. Implements the logic in Block RAM.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Multiplier Latency | mult_latency | The latency through a multiplier. |
| Add Latency | add_latency | The latency through an adder. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| a | in | Inherited | The real component of input 1. |
| b | in | Inherited | The imaginary component of input 1. |
| c | in | Inherited | The real component of input 2. |
| d | in | Inherited | The imaginary component of input 2. |
| real | out | Inherited | ac+bd |
| imag | out | Inherited | -ad+bc |

### Description

Perform a conjugating complex multiplication $(a + bi)(c \;\; di) = (ac + bd) + (bc \;\; ad)i$. Implements the logic in Block RAM. Each 4 bit real multiplier is implemented as a lookup table with 4b+4b=8b of address.

### Complex 4-bit Multiplier BRAM

**Block:** Complex 4-bit Multiplier Implemented in Block RAM (`cmult_4bit_br`)
**Block Author**: Block Author
**Document Author**: Document Author

#### Contents

### Summary

Perform a complex multiplication $(a + bi)(c \;\; di) = (ac \;\; bd) + (ad + bc)i$. Implements the logic in Block RAM.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Multiplier Latency | mult_latency | The latency through a multiplier. |
| Add Latency | add_latency | The latency through an adder. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| a | in | Inherited | The real component of input 1. |
| b | in | Inherited | The imaginary component of input 1. |
| c | in | Inherited | The real component of input 2. |
| d | in | Inherited | The imaginary component of input 2. |
| real | out | Inherited | ac-bd |
| imag | out | Inherited | ad-bc |

### Description

Each 4 bit real multiplier is implemented as a lookup table with 4b+4b=8b of address.

## Conjugate Complex 4-bit Multiplier, Dedicated Multipliers

**Block:** Conjugating Complex 4-bit Multiplier Implemented in Dedicated Multipliers. (`cmult_4bit_em*`)
**Block Author**: ?
**Document Author**: Vinayak Nagpal

### Contents

### Summary

Perform a conjugating complex multiplication $(a + bi)(c \ \ di) = (ac + bd) + (bc \ \ ad)i$. Implements the logic in dedicated multipliers.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Multiplier Latency | mult_latency | The latency through a multiplier. |
| dd Latency | add_latency | The latency through an adder. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| a | in | Inherited | The real component of input 1. |
| b | in | Inherited | The imaginary component of input 1. |
| c | in | Inherited | The real component of input 2. |
| d | in | Inherited | The imaginary component of input 2. |
| real | out | Inherited | ac+bd |
| imag | out | Inherited | -ad+bc |

### Description

Perform a conjugating complex multiplication $(a + bi)(c \ \ di) = (ac + bd) + (bc \ \ ad)i$. Implements the logic in dedicated multipliers. Each 4 bit real multiplier is implemented as a lookup table with 4b+4b=8b of address.

## Complex 4-bit Multiplier, Embedded Multipliers

**Block:** Complex 4-bit Multiplier Implemented in Embedded Multipliers (`cmult_4bit_em`)
**Block Author**: ?
**Document Author**: ?

### Contents

### Summary

Perform a complex multiplication $(a + bi)(c \ \ di) = (ac \ \ bd) + (ad + bc)i$. Implements the logic in embedded multipliers.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Multiplier Latency | mult_latency | The latency through a multiplier. |
| dd Latency | add_latency | The latency through an adder. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| a | in | Inherited | The real component of input 1. |
| b | in | Inherited | The imaginary component of input 1. |
| c | in | Inherited | The real component of input 2. |
| d | in | Inherited | The imaginary component of input 2. |
| real | out | Inherited | ac-bd |
| imag | out | Inherited | ad+bc |

### Description

Perform a complex multiplication $(a + bi)(c \quad di) = (ac \quad bd) + (ad + bc)i$. Implements the logic in embedded multipliers. Each 4 bit real multiplier is implemented as a lookup table with 4b+4b=8b of address.

## Conjugate Complex 4-bit Multiplier, Slices

**Block:** Conjugating Complex 4-bit Multiplier Implemented in Slices (`cmult_4bit_sl*`)
**Block Author**: Aaron Parsons
**Document Author**: Vinayak Nagpal

### Contents

### Summary

Perform a conjugating complex multiplication $(a + bi)(c \quad di) = (ac + bd) + (bc \quad ad)i$. Implements the logic in Slices.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Multiplier Latency | mult_latency | The latency through a multiplier. |
| Add Latency | add_latency | The latency through an adder. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| a | in | Inherited | The real component of input 1. |
| b | in | Inherited | The imaginary component of input 1. |
| c | in | Inherited | The real component of input 2. |
| d | in | Inherited | The imaginary component of input 2. |
| real | out | Inherited | ac+bd |
| imag | out | Inherited | -ad+bc |

### Description

Perform a conjugating complex multiplication $(a + bi)(c \; di) = (ac + bd) + (bc \; ad)i$. Implements the logic in Slices.

## Complex 4-bit Multiplier, Slices

**Block:** Complex 4-bit Multiplier Implemented in Slices (`cmult_4bit_sl`)
**Block Author**: Aaron Parsons
**Document Author**: Vinayak Nagpal

### Contents

### Summary

Perform a complex multiplication $(a + bi)(c \; di) = (ac \; bd) + (ad + bc)i$. Implements the logic in Slices.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Multiplier Latency | mult_latency | The latency through a multiplier. |
| Add Latency | add_latency | The latency through an adder. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| a | in | Inherited | The real component of input 1. |
| b | in | Inherited | The imaginary component of input 1. |
| c | in | Inherited | The real component of input 2. |
| d | in | Inherited | The imaginary component of input 2. |
| real | out | Inherited | ac-bd |
| imag | out | Inherited | ad+bc |

### Description

Perform a complex multiplication $(a + bi)(c \;\; di) = (ac \;\; bd) + (ad + bc)i$. Implements the logic in Slices.

## Complex Adder/Subtractor

**Block:** Complex Adder/Subtractor (`complex_addsub`)
**Block Author**: Aaron Parsons
**Document Author**: Ben Blackman

### Contents

### Summary

This block does a complex addition and subtraction of 2 complex numbers, `a` and `b`, and spits out 2 complex numbers, `a+b` and `a-b`.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Bit Width | BitWidth | The number of bits in its input. |
| Add Latency | add_latency | The latency of the adders/subtractors. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| a | IN | 2*BitWidth Fixed point | The first complex number whose higher BitWidth bits are its real part and lower BitWidth bits are its imaginary part. |
| b | IN | 2*BitWidth Fixed point | The second complex number whose higher BitWidth bits are its real part and lower BitWidth bits are its imaginary part. |
| a+b | OUT | 2*BitWidth Fixed point | Upper BitWidth bits are real(`a`)+real(`b`). Lower BitWidth bits are imaginary(`a`)-imaginary(`b`). |
| a-b | OUT | 2*BitWidth Fixed point | Upper BitWidth bits are imaginary(`a`)+imaginary(`b`). Lower BitWidth bits are real(`b`)-real(`a`). |

### Description

#### Usage

The top output, `a+b`, is a complex output whose real part equals the sum of the real parts of `a` and `b`. The imaginary part of `a+b` equals the difference of the imaginary parts of `a` and `b`. The bottom output, `a-b`, is a complex output whose real part equals the sum of the imaginary parts of `a` and `b`.The imaginary part of `a-b` equals the difference of the real parts of `b` and `a`. The latency of this block is 2*`add_latency`.

### Complex to Real-Imag

**Block:** Complex to Real-Imag Block (`c_to_ri`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

#### Contents

### Summary

Outputs real and imaginary components of a complex input. Useful for simplifying interconnects. See also ri_to_c.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Bit Width | n_bits | Specifies width of real/imag components. Assumed equal for both components. |
| Binary Point | bin_pt | Specifies the binary point location in the real/imaginary components. Assumed equal for both components. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| c | in | UFix_x_0 | Complex input, real in MSB, imaginary in LSB. |
| r | out | Fix_x_y | Real signed output, binary point specified by parameter. |
| i | out | Fix_x_y | Imaginary signed output, binary point specified by parameter. |

### Description

Outputs real and imaginary components of a complex input. Useful for simplifying interconnects. See also ri_to_c.

## DDS

**Block:** DDS (`dds`)
**Block Author**: Aaron Parsons
**Document Author**: Ben Blackman

### Contents

### Summary

Generates sines and cosines of different phases and outputs them in parallel.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Frequency Divisions (M) | freq_div | Denominator of the frequency. |
| Frequency (? /M * $2$ * pi) | freq | Numerator of the frequency. |
| Parallel LOs | num_lo | Number of parallel local oscillators. |
| Bit Width | n_bits | Bit width of the outputs. |
| Latency | latency | Description |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sinX | OUT | Fix_(n_bits)_(n_bits-1) | Sine output corresponding to the Xth local oscillator. |
| cosX | OUT | Fix_(n_bits)_(n_bits-1) | Cosine output corresponding to the Xth local oscillator. |

### Description

### Usage

There are `sin` and `cos` outputs each equal to the minimum of `num_lo` and `freq_div`. If `num_lo` > = `freq_div`/`freq`, then the outputs will be `lo_consts`. Otherwise each output will oscillate depending on the values of `freq_div` and `freq`. If the outputs oscillate, then there will be a latency of `latency` and otherwise there will be zero latency.

### Decimating FIR Filter

**Block:** Decimating FIR Filter (`dec_fir`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons, Ben Blackman

#### Contents

### Summary

FIR filter which can handle multiple time samples in parallel and decimates down to 1 time sample. If coefficiencts are symmetric, it will automatically fold before multiplying.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Number of Parallel Streams | n_inputs | The number of time samples which arrive in parallel. |
| Coefficients | coeff | The FIR coefficients. If this vector is symmetric, the FIR will automatically fold before multiplying. |
| Bit Width Out | n_bits | The number of bits in each real/imag sample of the complex number that is output. |
| Quantization Behavior | quantization | The quantization behavior used in converting to the output bit width. |
| Add Latency | add_latency | The latency of adders/converters. |
| Mult Latency | mult_latency | The latency of multipliers. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync_in | IN | boolean | Takes an impulse 1 cycle before input is valid. |
| realX | IN | Fix_(n_bits)_(n_bits-1) | Real input X |
| inagX | IN | Fix_(n_bits)_(n_bits-1) | Imaginary input X |
| sync_out | OUT | boolean | Will be high the clock cycle before `dout` is valid. |

### Description

### Usage

User specifies the number of parallel streams to be decimated to one complex number. Inputs are multiplied by the coefficients and added together to form `dout`. Bit Width Out specifies the widths of the real and imaginary components of the complex number to be output (Ex. if Bit Width Out = 8, then dout will be 16 bits, 8 for the real and imaginary components).

## Enabled Delay in BRAM

**Block:** The Enabled Delay in BRAM Block (`delay_bram_en_plus`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

A delay block that uses BRAM for its storage and only shifts when enabled. However, BRAM latency cannot be enabled, so output appears bram_latency clocks after an enable.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Enabled Delays | DelayLen | The length of the delay. |
| Extra (unenabled) delay for BRAM Latency | bram_latency | The latency of the underlying storage BRAM. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | ??? | The signal to be delayed. |
| en | in | ??? | To be asserted when input is valid. |
| out | out | ??? | The delayed signal. |
| valid | out | ??? | Asserted when output is valid. |

### Description

A delay block that uses BRAM for its storage and only shifts when enabled. However, BRAM latency cannot be enabled, so output appears bram_latency clocks after an enable.

## Programmable Delay in BRAM

**Block:** The Programmable Delay in BRAM Block (`delay_bram_prog`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

A delay block that uses BRAM for its storage and has a run-time programmable delay. When delay is changed, some randomly determined samples will be inserted/dropped from the buffered stream.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Max Delay ($2^?$) | MaxDelay | The maximum length of the delay (i.e. the BRAM Size). |
| BRAM Latency | bram_latency | The latency of the underlying storage BRAM. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| din | in | ??? | The signal to be delayed. |
| delay | in | ??? | The run-time programmable delay length. |
| dout | in | ??? | The delayed signal. |

### Description

A delay block that uses BRAM for its storage and has a run-time programmable delay. When delay is changed, some randomly determined samples will be inserted/dropped from the buffered stream.

## Delay in BRAM

**Block:** The Delay in BRAM Block (`delay_bram`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

A delay block that uses BRAM for its storage.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Delay Length | DelayLen | The length of the delay. |
| BRAM Latency | bram_latency | The latency of the underlying storage BRAM. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | ??? | The signal to be delayed. |
| out | out | ??? | The delayed signal. |

### Description

A delay block that uses BRAM for its storage.

## Complex Delay

**Block:** Complex Delay (`delay_complex`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

A delay block that treats its input as complex, splits it into real and imaginary components, delays each component by a specified amount, and then re-joins them into a complex output. The underlying storage is user-selectable (either BRAM or SLR16 elements). The reason for this is wide (36 bit) delays make adjacent multipliers in multiplier-bram pairs unusable.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Delay Depth | delay_depth | The length of the delay. |
| Bit Width | n_bits | Specifies the width of the real/imaginary components. Width of each component is assumed equal. |
| Use BRAM | use_bram | Set to 1 to implement the delay using BRAM. If 0, the delay will be implemented using SLR16 elements. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | ??? | The complex signal to be delayed. |
| out | out | ??? | The delayed complex signal. |

### Description

A delay block that treats its input as complex, splits it into real and imaginary components, delays each component by a specified amount, and then re-joins them into a complex output. The underlying storage is user-selectable (either BRAM or SLR16 elements). The reason for this is wide (36 bit) delays make adjacent multipliers in multiplier-bram pairs unusable.

## Delay in Slices

**Block:** The Delay in Slices Block (`delay_slr`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

A delay block that uses slices (SLR16s) for its storage.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Delay Length | DelayLen | The length of the delay. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | ??? | The signal to be delayed. |
| out | out | ??? | The delayed signal. |

### Description

A delay block that uses slices (SLR16s) for its storage.

### Programmable Wideband Delay

**Block:** Programmable Wideband Delay (`delay_wideband_prog`)
**Block Author**: Jason Manley, Mekhala Muley
**Document Author**: Jason Manley, Mekhala Muley

#### Contents

#### Summary

A delay block that uses single port BRAM for its storage and has a run-time programmable delay for simultaneous inputs.

#### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Max Delay | max_delay | The maximum length of delay which can be provided (in sample clock cycles). |
| Number of simultaneous inputs (2^?) | n_inputs_bits | Number of sequential time series inputs (specified in power of 2) required to the delay block. |
| BRAM Latency | bram_latency | The latency of the underlying storage BRAM. |
| Select type of BRAM | bram_type | Selects the type of BRAM (Single or Dual Port) to be used by the delay module. |

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| delay | in | Unsigned Integer | The runtime programmable delay value. |
| sync | in | Boolean | Sync pulse to synchronize this delay block with the other blocks in the design. |
| data_in | in | ??? | The simultaneous signals to be delayed. |
| sync_out | out | boolean | Synchronizing output pulse from the delay block. |
| data_out | out | inherited | The delayed simultaneous outputs. |

### Description

A delay block that uses single port BRAM for its storage and has a run-time programmable delay for sequential time series inputs. The block incurs a minimum delay as specified at the bottom of the block name. By default this is added to the user's requested delay.

Maximum delay should be in terms of powers of 2, if not, the block converts the maximum delay provided by user to the nearest power of 2.

Single port BRAM introduces glitches in the output if the programmable runtime delay is increased campared to the last entry. The minimum acceptable BRAM latency (Single and Dual Port) is 1, by default kept at 4.

### DRAM Vector Accumulator

**Block:** DRAM Vector Accumulator (`dram_vacc`)
**Block Author**: Arash Parsa
**Document Author**: Jason Manley

#### Contents

#### Summary

A vector accumulator for very large vector lengths using the BEE2's DRAM.

#### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| ??? | ??? | ??? |

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| Port Name | Port Direction | Port data type | Port Description |
| Port Name | in | ufix_x_y | Port Description |
| Port Name | in | inherited | Port Description |

### Description

A vector accumulator for very large vector lengths using the BEE2's DRAM.

### DRAM Vector Accumulator Test Vector Generator

**Block:** DRAM Vector Accumulator Test Vector Generator (`dram_vacc_tvg`)
**Block Author**: Jason Manley, Arash Parsa
**Document Author**: Jason Manley

#### Contents

#### Summary

Comprehensive TVG for the DRAM Vector Accumulator.

#### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Number of Vectors | len | |

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| tvg_sel | in | boolean | |
| sync_in | in | boolean | |
| data_in | in | ufix_32_0 | |
| valid_in | in | boolean | |
| sync_out | out | boolean | |
| data_out | out | ufix_32_0 | |
| valid_out | out | boolean | |

#### Description

Comprehensive TVG for the DRAM Vector Accumulator.

### Edge Detect

**Block:** The Edge Detect Block (`edge`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

**Contents**

#### Summary

Outputs true if a boolean input signal is not equal to its value during the last clock.

#### Mask Parameters

None.

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | Boolean | Input boolean signal. |
| out | out | Boolean | Edge detected output boolean signal. |

#### Description

Outputs true if a boolean input signal is not equal to its value during the last clock.

### Real-sampled Biplex FFT (demuxed by 2)

**Block:** Real-sampled Biplex FFT (with output demuxed by 2) (`fft_biplex_real_2x`)
**Block Author**: Aaron Parsons
**Block Maintainer**: Andrew Martens
**Document Author**: Aaron Parsons, Andrew Martens

### Contents

## Summary

Computes the real-sampled Fast Fourier Transform using the standard Hermitian conjugation trick to use a complex core to transform a two real streams. Thus, a biplex core (which can do 2 complex FFTs) can transform 4 real streams. Twiddle factor, and other logic sharing, allows multiples of 4 input streams to be processed simultaneously with minimal resource increases. Only positive frequencies are output (negative frequencies are the mirror images of their positive counterparts). Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^{N\ 1}$ 1. Real inputs 0 and 2 share one output port (with the data for 0 coming first, then the data for 2), likewise for inputs 1 and 3, and so on.

Please note that this documentation refers to the latest version of this block and may not be valid for older versions, please look in the history for older versions of this documentation.

### Mask Parameters

| Parameter | Variable | Description | Recommended Value |
|---|---|---|---|
| Number simultaneous inputs (4*?) | n_inputs | The number of inputs the FFT is to process as a multiple of 4. | |
| Size of FFT: (2^?) | FFTSize | The number of channels computed in the complex FFT core. The number of channels output for each real stream is half of this. | |
| Input bit width | input_bit_width | The number of bits in each real and imaginary sample as they are input to the FFT. If bit growth is not chosen, each FFT stage will round numbers back down to this number of bits after performing a butterfly computation. If bit growth is chosen, the number of bits will increase by one with every FFT stage up to the maximum specified. | To make optimal use of BRAMs => 18 For low FFT noise => 25 |
| Input binary point | bin_pt | The position of the binary point in the input data | |
| Coefficient Bit Width | coeff_bit_width | The number of bits used in the real and imaginary part of the twiddle factors at each stage. | 18 |
| Asynchronous operation | async | Whether valid data is input on every clock cycle or is flagged via the en input port. | |
| Quantization Behavior | quantization | Specifies the rounding behaviour used at the end of each twiddle and butterfly computation to return to the number of bits if bit growth is not enabled or to keep the number of bits at the maximum specified. | NOT Truncate. |
| Overflow Behavior | overflow | Indicates the behaviour of the FFT core when the value of a sample exceeds what can be expressed in the specified bit width. | |
| Add Latency | add_latency | Latency through adders in the FFT. | 1 |
| Mult Latency | mult_latency | Latency through multipliers in the FFT. | 2 |
| BRAM Latency | bram_latency | Latency through BRAM in the FFT. | 2 For designs aimed at > 200MHz => 3 |
| Convert Latency | conv_latency | Latency through blocks used to reduce bit widths after twiddle and butterfly stages. | 1 For designs aimed at > 180Mhz => 2 |
| Number bits above which to store stage's coefficients in BRAM (2^? bits) | coeffs_bit_limit | Determines the threshold at which the twiddle coefficients in a stage are stored in BRAM. Below this threshold distributed RAM is used. | 8 (ensures at least 2^8=256 bits out of 18432 bits of BRAM used) |
| Number bits above which to implement stage's delays in BRAM (2^? bits) | delays_bit_limit | Determines the threshold at which data delays in a stage are stored in BRAM. Below this threshold distributed RAM is used. | 8 (ensures at least 2^8=256 bits out of 18432 bits of BRAM used) |
| BRAM sharing in coeff storage | coeff_sharing | Real and imaginary components of twiddle factors can be generated from the same set of coefficients, reducing BRAM use at the cost of some logic. | |
| Store a fraction of coeff factors where useful | coeff_decimation | The full set of twiddle factors can be generated from a smaller set, reducing BRAM use at the cost of the some logic. | |
| Maximum | max_fanout | The maximum fanout the twiddle factors are allowed to experience be- | |

### Ports

| Port | Dir | Data Type | Description | Recommended Use |
|---|---|---|---|---|
| sync | in | Boolean | sync is used to indicate the last data word of a frame of input data. When the block is in asynchronous operating mode an active signal is aligned with en being active. When the block is in synchronous operating mode, a an active pulse is aligned with the clock cycle before the first valid data of a new input frame. | Ensure the sync period complies with the memo describing correct use. |
| shift | in | Unsigned | Sets the shifting schedule through the FFT to prevent overflow. Bit 0 specifies the behavior of stage 0, bit 1 of stage 1, and so on. If a stage is set to shift (with bit = 1), then every sample is divided by 2 at the output of that stage. | |
| pol | in | Signed consisting of one (Input Bit Width) width signals per input. | The time-domain stream(s) to be channelised. | Data amplitude should not exceed 0.5 (divide data by 2 pre-FFT) |
| en | in | Boolean | When asynchronous operation is chosen, this port indicates that valid input data is available on all input data ports. | |
| sync_out | out | Boolean | Indicates that data out will be valid next clock cycle when in synchronous mode, or when dvalid is next active. | |
| pol_out | out | Inherited | The frequency channels. | |
| of | out | Unsigned, one bit per 4 inputs | Indication of internal arithmetic overflow. Not time aligned with data. The most significant bit is the flag for pol0_in, pol1_in, pol2_in and pol3_in etc. | |
| dvalid | out | Boolean | Indicates that valid data is available on all output data ports. | |

### Description

Computes the real-sampled Fast Fourier Transform using the standard Hermitian conjugation trick to use a complex core to transform a two real streams. Thus, a biplex core (which can do 2 complex FFTs) can transform 4 real streams. Twiddle factor, and other logic sharing, allows multiples of 4 input streams to be processed simultaneously with minimal resource increases. Only positive frequencies are output (negative frequencies are the mirror images of their positive counterparts). Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^{N\ 1}$ 1. Real inputs 0 and 2 share one output port (with the data for 0 coming first, then the data for 2), likewise for inputs 1 and 3, and so on.

### Real-sampled Biplex FFT (demuxed by 4)

**Block:** Real-sampled Biplex FFT (with output demuxed by 4) (`fft_biplex_real_4x`)
**Block Author**: Aaron Parsons
**Block Maintainer**: Andrew Martens
**Document Author**: Aaron Parsons, Andrew Martens

## Contents

### Summary

Computes the real-sampled Fast Fourier Transform using the standard Hermitian conjugation trick to use a complex core to transform a two real streams. Thus, a biplex core (which can do 2 complex FFTs) can transform 4 real streams. Twiddle factor, and other logic sharing, allows multiples of 4 input streams to be processed simultaneously with minimal resource increases. All frequencies (both positive and negative) are output (negative frequencies are the mirror images of their positive counterparts). Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^N$ 1.

### Mask Parameters

| Parameter | Variable | Description | Recommended Value |
|---|---|---|---|
| Number simultaneous inputs (4*?) | n_inputs | The number of inputs the FFT is to process as a mutliple of 4. | |
| Size of FFT: (2^?) | FFTSize | The number of channels computed in the complex FFT core. The number of channels output for each real stream is half of this. | |
| Input bit width | input_bit_width | The number of bits in each real and imaginary sample as they are input to the FFT. If bit growth is not chosen, each FFT stage will round numbers back down to this number of bits after performing a butterfly computation. If bit growth is chosen, the number of bits will increase by one with every FFT stage up to the maximum specified. | To make optimal use of BRAMs => 18 For low FFT noise => 25 |
| Input binary point | bin_pt | The position of the binary point in the input data | |
| Coefficient Bit Width | coeff_bit_width | The number of bits used in the real and imaginary part of the twiddle factors at each stage. | 18 |
| Asynchronous operation | async | Whether valid data is input on every clock cycle or is flagged via the en input port. | |
| Quantization Behavior | quantization | Specifies the rounding behavior used at the end of each twiddle and butterfly computation to return to the number of bits specified above. | NOT Truncate. |
| Overflow Behavior | overflow | Indicates the behavior of the FFT core when the value of a sample exceeds what can be expressed in the specified bit width. | Wrap as Saturate will not make overflow corruption better behaved. |
| Add Latency | add_latency | Latency through adders in the FFT. | 1 |
| Mult Latency | mult_latency | Latency through multipliers in the FFT. | 2 |
| BRAM Latency | bram_latency | Latency through BRAM in the FFT. | 2 For designs aimed at > 200MHz => 3 |
| Convert Latency | conv_latency | Latency through blocks used to reduce bit widths after twiddle and butterfly stages. | 1 For designs aimed at > 180Mhz => 2 |
| Number bits above which to store stage's coefficients in BRAM (2^? bits) | coeffs_bit_limit | Determines the threshold at which the twiddle coefficients in a stage are stored in BRAM. Below this threshold distributed RAM is used. | 8 (ensures at least 2^8=256 bits out of 18432 bits of BRAM used) |
| Number bits above which to implement stage's delays in BRAM (2^? bits) | delays_bit_limit | Determines the threshold at which data delays in a stage are stored in BRAM. Below this threshold distributed RAM is used. | 8 (ensures at least 2^8=256 bits out of 18432 bits of BRAM used) |
| BRAM sharing in coeff storage | coeff_sharing | Real and imaginary components of twiddle factors can be generated from the same set of coefficients, reducing BRAM use at the cost of some logic. | |
| Store a fraction of coeff factors where useful | coeff_decimation | The full set of twiddle factors can be generated from a smaller set, reducing BRAM use at the cost of the some logic. | |

### Ports

| Port | Dir | Data Type | Description | Recommended Use |
|---|---|---|---|---|
| sync | in | Boolean | sync is used to indicate the last data word of a frame of input data. When the block is in asynchronous operating mode an active signal is aligned with en being active. When the block is in synchronous operating mode, a an active pulse is aligned with the clock cycle before the first valid data of a new input frame. | Ensure the sync period complies with the memo describing correct use. |
| shift | in | Unsigned | Sets the shifting schedule through the FFT to prevent overflow. Bit 0 specifies the behavior of stage 0, bit 1 of stage 1, and so on. If a stage is set to shift (with bit = 1), then every sample is divided by 2 at the output of that stage. | |
| pol | in | Signed consisting of one (Input Bit Width) width signals per input. | The time-domain stream(s) to be channelised. | Data amplitude should not exceed 0.5 (divide data by 2 pre-FFT) |
| en | in | Boolean | When asynchronous operation is chosen, this port indicates that valid input data is available on all input data ports. | |
| sync_out | out | Boolean | Indicates that data out will be valid next clock cycle. | |
| pol_out | out | Inherited | The frequency channels. | |
| of | out | Unsigned, one bit per 4 inputs | Indication of internal arithmetic overflow. Not time aligned with data. The most significant bit is the flag for pol0_in, pol1_in, pol2_in and pol3_in etc. | |

### Description

Computes the real-sampled Fast Fourier Transform using the standard Hermitian conjugation trick to use a complex core to transform a two real streams. Thus, a biplex core (which can do 2 complex FFTs) can transform 4 real streams. Twiddle factor, and other logic sharing, allows multiples of 4 input streams to be processed simultaneously with minimal resource increases. All frequencies (both positive and negative) are output (negative frequencies are the mirror images of their positive counterparts). Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^N$ 1.

### FFT

**Block:** FFT (`fft`)
**Block Author**: Aaron Parsons
**Block Maintainer**: Andrew Martens
**Document Author**: Aaron Parsons, Andrew Martens

**Contents**

## Summary

Computes the Fast Fourier Transform with $2^N$ channels for time samples presented $2^P$ at a time in parallel. Uses a biplex FFT architecture under the hood which has been extended to handled time samples in parallel. For P = 0, this block accepts two independent, parallel streams (labelled as pols) and computes the FFT of each independently (the biplex architecture provides this for free). Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^N$ 1 (which can be interpreted as channel -1). When multiple time samples are presented in parallel on the input, multiple frequency samples are output in parallel.

### Mask Parameters

| Parameter | Variable | Description | Recommended Value |
|---|---|---|---|
| Number simultaneous streams | n_streams | The number of input data streams to be processed in parallel. Each stream consists of a set of parallel inputs set by another parameter (see Number of Simultaneous Inputs) | |
| Size of FFT: (2^?) | FFTSize | The number of channels computed in the complex FFT core. The number of channels output for each real stream is half of this. | |
| Input Bit Width | input_bit_width | The number of bits in each real and imaginary sample as they are carried through the FFT. Each FFT stage will round numbers back down to this number of bits after performing a butterfly computation if bit growth is not enabled. | To make optimal use of BRAM => 18 For low FFT noise => 25 |
| Input binary point | bin_pt | The position of the binary point in the input data. | |
| Coefficient Bit Width | coeff_bit_width | The number of bits used in the real and imaginary part of the twiddle factors at each stage. | 18 |
| Number of Simultaneous Inputs: (2^?) | n_inputs | The number of parallel time samples which are presented to the FFT core each clock. This must be at least $2^2$. The number of output ports is half of this value. | |
| Unscramble output (ie, put channels in canonical order) | unscramble | The FFT inherently produces data in an order that requires unscrambling before being used by many algorithms. This requires resources and can limit performance and so should be disabled if not necessary. | |
| Asynchronous operation | async | Whether valid data is input on every clock cycle or is flagged via the en input port. | |
| Quantization Behavior | quantization | Specifies the rounding behavior used at the end of each twiddle and butterfly computation to return to the number of bits specified above. | NOT Truncate. |
| Overflow Behavior | overflow | Indicates the behavior of the FFT core when the value of a sample exceeds what can be expressed in the specified bit width. | Wrap as Saturate will not make overflow corruption better behaved. |
| Add Latency | add_latency | Latency through adders in the FFT. | 1 |
| Mult Latency | mult_latency | Latency through multipliers in the FFT. | 2 |
| BRAM Latency | bram_latency | Latency through BRAM in the FFT. | 2 For designs aimed at > 200MHz => 3 |
| Convert Latency | conv_latency | Latency through blocks used to reduce bit widths after twiddle and butterfly stages. | 1 For designs aimed at > 180Mhz => 2 |
| Number bits above which to store stage's coefficients in BRAM (2^? bits) | coeffs_bit_limit | Determines the threshold at which the twiddle coefficients in a stage are stored in BRAM. Below this threshold distributed RAM is used. | 8 (ensures at least 2^8=256 bits out of 18432 bits of BRAM used) |
| Number bits above which to implement stage's delays | delays_bit_limit | Determines the threshold at which data delays in a stage are stored in BRAM. Below this threshold distributed RAM is used. | 8 (ensures at least 2^8=256 bits out of |

### Ports

| Port | Dir | Data Type | Description | Recommended Use |
|---|---|---|---|---|
| sync | in | Boolean | sync is used to indicate the last data word of a frame of input data. When the block is in asynchronous operating mode an active signal is aligned with en being active. When the block is in synchronous operating mode, a an active pulse is aligned with the clock cycle before the first valid data of a new input frame. | Ensure the sync period complies with the memo describing correct use. |
| shift | in | Unsigned | Sets the shifting schedule through the FFT to prevent overflow. Bit 0 specifies the behavior of stage 0, bit 1 of stage 1, and so on. If a stage is set to shift (with bit = 1), then every sample is divided by 2 at the output of that stage. | |
| in<stream><input> | in | Signed consisting of one (Input Bit Width) width signals per input. | The time-domain stream(s) to be channelised. | Data amplitude should not exceed 0.5 (divide data by 2 pre-FFT) |
| en | in | Boolean | When asynchronous operation is chosen, this port indicates that valid input data is available on all input data ports. | |
| sync_out | out | Boolean | Indicates that data out will be valid next clock cycle. | |
| out<stream><output> | out | Inherited | The frequency channels. | |
| of | out | Unsigned, one bit per input stream | Indication of internal arithmetic overflow. Not time aligned with data. The most significant bit is the flag for input stream 0 etc. | |

### Description

Computes the Fast Fourier Transform with $2^N$ channels for time samples presented $2^P$ at a time in parallel. Uses a biplex FFT architecture under the hood which has been extended to handled time samples in parallel. For P = 0, this block accepts two independent, parallel streams (labelled as pols) and computes the FFT of each independently (the biplex architecture provides this for free). Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^N$ 1 (which can be interpreted as channel -1). When multiple time samples are presented in parallel on the input, multiple frequency samples are output in parallel.

### Real-sampled Wideband FFT

**Block:** Real-sampled Wideband FFT (`fft_wideband_real`)

**Block Author**: Aaron Parsons

**Document Author**: Aaron Parsons

## Contents

### Summary

Computes the real-sampled Fast Fourier Transform using the standard Hermitian conjugation trick to use a complex core to transform a single real stream using half the normal resources (this requires at least 4 time samples in parallel). Only positive frequencies are output (negative frequencies are the mirror images of their positive counterparts), so there the number of output ports is half the number of input ports. Uses a biplex FFT architecture under the hood which has been extended to handle time samples in parallel. Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^{N\ 1}$ 1.

### Mask Parameters

| Parameter | Variable | Description | Recommended Value |
|---|---|---|---|
| Number simultaneous streams | n_streams | The number of input data streams to be processed in parallel. Each stream consists of a set of parallel inputs set by another parameter (see Number of Simultaneous Inputs) | |
| Size of FFT: (2^?) | FFTSize | The number of channels computed in the complex FFT core. The number of channels output for each real stream is half of this. | |
| Input Bit Width | input_bit_width | The number of bits in each real and imaginary sample as they are carried through the FFT. Each FFT stage will round numbers back down to this number of bits after performing a butterfly computation if bit growth is not enabled. | To make optimal use of BRAM => 18 For low FFT noise => 25 |
| Input binary point | bin_pt | The position of the binary point in the input data. | |
| Coefficient Bit Width | coeff_bit_width | The number of bits used in the real and imaginary part of the twiddle factors at each stage. | 18 |
| Number of Simultaneous Inputs: (2^?) | n_inputs | The number of parallel time samples which are presented to the FFT core each clock. This must be at least $2^2$. The number of output ports is half of this value. | |
| Unscramble output (ie, put channels in canonical order) | unscramble | The FFT inherently produces data in an order that requires unscrambling before being used by many algorithms. This requires resources and can limit performance and so should be disabled if not necessary. | |
| Asynchronous operation | async | Whether valid data is input on every clock cycle or is flagged via the en input port. | |
| Quantization Behavior | quantization | Specifies the rounding behavior used at the end of each twiddle and butterfly computation to return to the number of bits specified above. | NOT Truncate. |
| Overflow Behavior | overflow | Indicates the behavior of the FFT core when the value of a sample exceeds what can be expressed in the specified bit width. | Wrap as Saturate will not make overflow corruption better behaved. |
| Add Latency | add_latency | Latency through adders in the FFT. | 1 |
| Mult Latency | mult_latency | Latency through multipliers in the FFT. | 2 |
| BRAM Latency | bram_latency | Latency through BRAM in the FFT. | 2 For designs aimed at > 200MHz => 3 |
| Convert Latency | conv_latency | Latency through blocks used to reduce bit widths after twiddle and butterfly stages. | 1 For designs aimed at > 180Mhz => 2 |
| Number bits above which to store stage's coefficients in BRAM (2^? bits) | coeffs_bit_limit | Determines the threshold at which the twiddle coefficients in a stage are stored in BRAM. Below this threshold distributed RAM is used. | 8 (ensures at least 2^8=256 bits out of 18432 bits of BRAM used) |
| Number bits above which to implement stage's delays | delays_bit_limit | Determines the threshold at which data delays in a stage are stored in BRAM. Below this threshold distributed RAM is used. | 8 (ensures at least 2^8=256 bits out of |

### Ports

| Port | Dir | Data Type | Description | Recommended Use |
|---|---|---|---|---|
| sync | in | Boolean | sync is used to indicate the last data word of a frame of input data. When the block is in asynchronous operating mode an active signal is aligned with en being active. When the block is in synchronous operating mode, a an active pulse is aligned with the clock cycle before the first valid data of a new input frame. | Ensure the sync period complies with the memo describing correct use. |
| shift | in | Unsigned | Sets the shifting schedule through the FFT to prevent overflow. Bit 0 specifies the behavior of stage 0, bit 1 of stage 1, and so on. If a stage is set to shift (with bit = 1), then every sample is divided by 2 at the output of that stage. | |
| in<stream><input> | in | Signed consisting of one (Input Bit Width) width signals per input. | The time-domain stream(s) to be channelised. | Data amplitude should not exceed 0.5 (divide data by 2 pre-FFT) |
| en | in | Boolean | When asynchronous operation is chosen, this port indicates that valid input data is available on all input data ports. | |
| sync_out | out | Boolean | Indicates that data out will be valid next clock cycle. | |
| out<stream><output> | out | Inherited | The frequency channels. | |
| of | out | Unsigned, one bit per input stream | Indication of internal arithmetic overflow. Not time aligned with data. The most significant bit is the flag for input stream 0 etc. | |

### Description

Computes the real-sampled Fast Fourier Transform using the standard Hermitian conjugation trick to use a complex core to transform a single real stream using half the normal resources (this requires at least 4 time samples in parallel). Only positive frequencies are output (negative frequencies are the mirror images of their positive counterparts), so there the number of output ports is half the number of input ports. Uses a biplex FFT architecture under the hood which has been extended to handled time samples in parallel. Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^{N\ 1}$ 1.

### Programmable Fine Delay w/ Fringe stop

**Block:** Programmable fine delay with fringe stop (`finedelay_fstop_prog`)
**Block Author**: Mekhala Muley, GMRT, India.
**Document Author**: Mekhala Muley, GMRT, India.

Contents

## Summary

This block performs the fine delay correction along with the fringe stop. It accepts the simultaneous stream of data from the FFT module and has a run time programmable fine delay correction along with the fringe stopping.

**Note:** This block is specifically compatible with the "fft_wideband_real" module. For other FFT modules changes will be required in this block depending upon output of the FFT module used in the design.

## Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Number of simultaneous inputs | n_input | Number of simultaneous inputs (in frequency domain) from the FFT module. |
| Number of FFT channels | fft_len | Number of channels in the FFT |
| FFT input bitwidth | fft_bits | Number of bits in each real and imaginary samples obtained from the FFT module. |
| Sine-Cos LUT input bitwidth | theta_bits | Address bitwidth required for the SineCos LUT and hence decides the resolution of the phase. Allowable bitwidth for Sine-Cos LUT ranges from 3- 16. |
| Sine-Cos LUT output data width | sine_cos_bits | Data width of the sine cos LUT. |
| Maximum number of FFT cycles (Rate of change of fringe = 2^?) | fft_cycle_bits | Number of FFT cycles after which the rate of change of fringe needs to be applied. The number of FFT cycles are specified in terms of powers of 2. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | In | Boolean | Indicates the next clock cycle which contains valid data. |
| theta_fract | In | Unsigned | Sets the integer number required for fine delay correction.The bitwidth is equal to the address width for SineCos LUT. The method of calculating the range of integer required for the fine delay correction is explained in Description. |
| theta_fs | In | Unsigned | Sets the integer value equivalent to the initial phase value for fringe stop . The bitwidth is equal to the address width for SineCos LUT. Hence the possible range for initial phase varies from 0 to 2^theta_bits. |
| fft_fs | In | Unsigned | Sets the number of FFT cycles after which fringe update rate need to be applied. |
| en_theta_fs | In | Unsigned | This is the one bit control signal required to upload the new initial phase required for fringe stop. The new initial phase value gets loaded only at the posedge of this signal. |
| pol_in | In | Inherited | The frequency domain stream from the FFT module. |
| sync_out | Out | Boolean | Indicates that data out will be valid next clock cycle. |
| out | Out | Inherited | The fine delay and fringe stop corrected frequency channels. |

### Description

This block performs the fine delay correction along with the fringe stop. This block accepts the simultaneous stream of data from the FFT module and has a run time programmable fine delay correction along with the fringe stopping.

**Note:** This block is specifically compatible with the “fft_wideband_real” module. For other FFT modules changes will be required in this block depending upon the way in which the data is output from the FFT module.

**Fine Delay Correction:**

Masking parameters like *theta_bits* and *sine_cos_bits* decides the resolution of the phase required for fine delay and fringe stop.

For eg. theta_bits = 14 will generate a SineCos LUT with a depth of 2^14=16K, hence the resolution is of 0.02197 degrees. Consider the correlator design with specifications like bandwidth = 300MHz and no. of FFT channels = fft_len = 1024, then the maximum integer value of theta_fract (i.e. max fine delay = 1 clk cycle)will be x = (2^ theta_bits)/ (fft_len/2) = 32

Thus 1/32th of the clock cycle delay can be compensated with the above parameters.

**Fringe Stop:**

Masking parameter *fft_cycle_bits* determines the maximum number of FFT cycles after which the fringe phase will be incremented.

For eg. Let the sync period is of 2^27 clks and number of FFT points be 2^10 then the maximum number of FFT cycles for incrementing the fringe phase by amount of resolution set for the Sine-Cos LUT = 2^27 / 2^10 = 2^17

It means that minimum rate of incrementing fringe phase by 0.02197 degrees is after 2^17 FFT cycles.

### Programmable Fine Delay w/ Fringe stop, CORDIC

**Block:** Programmable fine delay with fringe stop using CORDIC block (`finedelay_fstop_prog_cordic`)
**Block Author**: Mekhala Muley, GMRT, India.
**Document Author**: Mekhala Muley, GMRT, India.

#### Contents

#### Summary

This block performs the fine delay correction along with the fringe stop. This block accepts the simultaneous stream of data from the FFT module and has a run time programmable fine delay correction along with the fringe stopping. This block uses CORDIC block which is compatible with Virtex 5 FPGA. Hence this block can be used for designs on Virtex5 FPGA of ROACH board.

**Note 1:** This block is specifically compatible with the "fft_wideband_real" module. For other FFT modules changes will be required in this block depending upon the way in which the data is output from the FFT module.

**Note 2:** This block requires System Generator Version 11.1 to make it compatible with Virtex 5.

**Note 3:** Currently the block is able to correct delay for One clock or less than one clock. It does not correct delays which are more than one clock.

#### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Number of simultaneous inputs | n_input | Number of simultaneous inputs (in frequency domain) from the FFT module. |
| Number of FFT channels | fft_len | Number of channels in the FFT |
| FFT input bitwidth | fft_bits | Number of bits in each real and imaginary samples obtained from the FFT module. |
| CORDIC Sine-Cos input bitwidth | theta_bits | Input bitwidth required for the CORDIC SineCos block and hence decides the resolution of the phase. |
| CORDIC Sine-Cos input binary bitwidth | theta_binary_bits | Input binary bitwidth of the CORDIC sine cos block. |
| Maximum number of FFT cycles (Rate of change of fringe = 2^?) | fft_cycle_bits | Number of FFT cycles after which the rate of change of fringe needs to be applied. The number of FFT cycles are specified in terms of powers of 2. |
| Sync Period | sync_period | Duration of the sync pulse. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | In | Boolean | Indicates the next clock cycle which contains valid data. |
| theta_fract | In | Unsigned | Sets the integer number required for fine delay correction.The bitwidth is equal to the address width for SineCos LUT. The method of calculating the range of integer required for the fine delay correction is explained in Description. |
| theta_fs | In | Unsigned | Sets the integer value equivalent to the initial phase value for fringe stop . The bitwidth is equal to the address width for SineCos LUT. Hence the possible range for initial phase varies from 0 to 2^theta_bits. |
| fft_fs | In | Unsigned | Sets the number of FFT cycles after which fringe update rate need to be applied. |
| en_theta_fs | In | Unsigned | This is the one bit control signal required to upload the new initial phase required for fringe stop. The new initial phase value gets loaded only at the posedge of this signal. |
| pol_in | In | Inherited | The frequency domain stream from the FFT module. |
| sync_out | Out | Boolean | Indicates that data out will be valid next clock cycle. |
| out | Out | Inherited | The fine delay and fringe stop corrected frequency channels. |

### Description

This block performs the fine delay correction along with the fringe stop. This block accepts the simultaneous stream of data from the FFT module and has a run time programmable fine delay correction along with the fringe stopping.This block uses CORDIC block which is compatible with Virtex 5 FPGA. Hence this block can be used for designs on Virtex5 FPGA of ROACH board.

**Note:** This block is specifically compatible with the “fft_wideband_real” module. For other FFT modules changes will be required in this block depending upon the way in which the data is output from the FFT module.

**Fine Delay Correction:**

Masking parameters like theta_bits and theta_binary_bits decides the resolution of the phase required for fine delay and fringe stop.

Resolution = (2^theta_bits * 2 * pi) / 8

For eg. theta_bits = 20 will generate a CORDIC SineCos block with a resolution of 0.000437 degrees.

Let bandwidth = 300MHz and no. of FFT channels = fft_len = 1024 then the maximum integer value of theta_fract (i.e. max fine delay = 1 clk cycle)will be

num = (2^theta_bits * pi) / 8

x = (num * 2)/(fft_len/2) = 1608

Thus 1/1608th of the clock cycle delay can be compensated with the above parameters.

**Fringe Stop:**

Masking parameter fft_cycle_bits determines the maximum number of FFT cycles after which the fringe phase will be incremented.

For eg. Let the sync period is of 2^27 clks and number of FFT points be 2^10 then the maximum number of FFT cycles for incrementing the fringe phase by amount of resolution set for the Sine-Cos LUT = 2^27 / 2^10 = 2^17

It means that minimum rate of incrementing fringe phase by 0.000437 degrees is after 2^17 FFT cycles.

## FIR Column

**Block:** FIR Column (`fir_col`)
**Block Author**: Aaron Parsons
**Document Author**: Ben Blackman

### Contents

### Summary

Takes in real and imaginary numbers to be multiplied by the coefficients and then the filter sums the real and imaginary parts separately. Then both sums are output as well as a delayed version of the unchanged inputs.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Inputs | n_inputs | The number of real inputs and the number of imaginary inputs. |
| Coefficients | coeff | A vector of coefficients of this FIR. Should be the same number of coefficients as inputs. |
| Add Latency | add_latency | The latency of the internal adders. |
| Mult Latency | mult_latency | The latency of the internal multipliers. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| realX | IN | Inherited | This is real input X. Its data type is inherited from the previous block. |
| imagX | IN | Inherited | This is imaginary input X. Its data type is inherited from the previous block. |
| real_outX | OUT | Inherited | This output is `realX` delayed by 1 cycle. |
| imag_outX | OUT | Inherited | This output is `imagX` delayed by 1 cycle. |
| real_sum | OUT | Inherited | This is the sum of all the `realX` * coefficient X. |
| imag_sum | OUT | Inherited | This is the sum of all the `imagX` * coefficient X. |

### Description

#### Usage

This block takes in a number of inputs in parallel and outputs a delayed version of them and also multiplies the inputs by the coefficients. Then `real_sum` and `imag_sum` are computed and are delayed due to the latency in the adders which depends both on the `add_latency` and the number of inputs.

## FIR Double Column

**Block:** FIR Double Column (`fir_dbl_col`)
**Block Author**: Aaron Parsons
**Document Author**: Ben Blackman

### Contents

### Summary

Takes in real and imaginary numbers to be multiplied by the coefficients and then the filter sums the real and imaginary parts separately. Then both sums are output as well as a delayed version of the unchanged inputs.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Inputs | n_inputs | The number of real inputs and the number of imaginary inputs. |
| Coefficients | coeff | A vector of coefficients of this FIR. Should be the same number of coefficients as inputs. |
| Add Latency | add_latency | The latency of the internal adders. |
| Mult Latency | mult_latency | The latency of the internal multipliers. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| real | IN | Inherited | This real input is to be multiplied by one of the coefficients. |
| imag | IN | Inherited | This imaginary input is to be multiplied by one of the coefficients. |
| real_back | IN | Inherited | These real inputs correspond to the second half of the input stream. They get added to one of the `real` inputs before being multiplied by the coefficient. |
| imag_back | IN | Inherited | These imaginary inputs correspond to the second half of the input stream. They get added to one of the `imag` inputs before being multiplied by the coefficient. |
| real_out | OUT | Inherited | This output is `real` delayed by 1 cycle. |
| imag_out | OUT | Inherited | This output is `imag` delayed by 1 cycle. |
| real_back_out | OUT | Inherited | This output is `real_back` delayed by 1 cycle. |
| imag_back_out | OUT | Inherited | This output is `imag_back` delayed by 1 cycle. |
| real_sum | OUT | Inherited | This is the sum of all the multiplications between `real` and `real_back` and their corresponding coefficients. |
| imag_sum | OUT | Inherited | This is the sum of all the multiplications between `imag` and `imag_back` and their corresponding coefficients. |

### Description

### Usage

This block takes in a number of inputs in parallel and outputs a delayed version of them and also multiplies the inputs by the coefficients. Then `real_sum` and `imag_sum` are computed and are delayed due to the latency in the adders which depends both on the `add_latency` and the number of inputs. For example, if you choose the number of inputs to be 2, then there will be 2 `real` and 2 `real_back` input ports along with 2 `imag` and 2 `imag_back` input ports. The FIR Double Column blocks takes advantage of the symmetric filter tap coefficients by adding the first and last inputs together before multiplying by the coefficient. This results in a more efficient FIR filter column.

## FIR Tap

**Block:** FIR Tap (`fir_tap`)
**Block Author**: Aaron Parsons
**Document Author**: Ben Blackman

## Contents

## Summary

This block multiplies both inputs by `factor` and outputs the result immediately after the multiply and outputs a delayed copy of the input by 1 cycle,

## Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Factor | factor | The value that multiplies both inputs. |
| Mult latency | latency | The latency of the multiplier. |

## Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| a | IN | Inherited | The first number to be multiplied by `factor`. It usually is the real component of an input. |
| b | IN | Inherited | The second number to be multiplied by `factor`. It usually is the imaginary component of an input. |
| a_out | OUT | Inherited | The input `a` delayed by 1 cycle. |
| b_out | OUT | Inherited | The input `b` delayed by 1 cycle. |
| real | OUT | Inherited | The result of the multiplication of `a` with `factor`. |
| imag | OUT | Inherited | The result of the multiplication of `b` with `factor`. |

## Description

### Usage

`a_out` and `b_out` are 1 cycle delayed versions of `a` and `b`, respectively. `real` and `imag` are the results of `a` * `factor` and `b` * `factor`, respectively. The delay from `a` to `real` or `b` to `imag` is equal to `latency`.

## Freeze Counter Block

**Block:** Freeze Counter Block (`freeze_cntr`)

**Block Author**: Aaron Parsons

**Document Author**: Aaron Parsons

## Contents

### Summary

A freeze counter is an enabled counter which holds its final value (regardless of enables) until it is reset.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Counter Length ($2^?$) | CounterBits | Specifies the number of bits (and the final count output of $2^{bits\ 1}$). |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| en | in | ??? | Step the counter by 1 unless addr=$2^{bits\ 1}$. |
| rst | in | ??? | Reset counter to 0. |
| addr | out | ??? | Current output of the counter. |
| we | out | Boolean | Outputs boolean true just before addr is incremented. |
| done | out | Boolean | Outputs boolean true when a final en is asserted and addr=$2^{bits\ 1}$. |

### Description

A freeze counter is an enabled counter which holds its final value (regardless of enables) until it is reset. Thus, a $2^5$ freeze counter will count from 0 to 31 on 31 enables, but will hold 31 thereafter until a reset occurs. This block is useful for writing data in a single pass to memory without looping.

## Local Oscillator Constant

**Block:** Local Oscillator Constant (`lo_const`)
**Block Author**: Aaron Parsons
**Document Author**: Ben Blackman

### Contents

### Summary

Gives the sine and cosine of a desired constant phase.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Output Bitwidth | n_bits | Bitwidth of the outputs. |
| Phase (0 to 2*pi) | phase | The phase value for which the sine and cosine are evaluated. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sin | OUT | Fix_(n_bits)_(n_bits-1) | The sine of the given phase value. |
| cos | OUT | Fix_(n_bits)_(n_bits-1) | The cosine of the given phase value. |

### Description

#### Usage

This block gives the sine and cosine of a user-specified, constant phase value with a user-specified bitwidth.

### Local Oscillator

**Block:** Local Oscillator (`lo_osc`)
**Block Author**: Aaron Parsons
**Document Author**: Ben Blackman

### Contents

### Summary

Generates an oscillating sine and cosine.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Output Bitwidth | n_bits | Bitwidth of the outputs. |
| Counter Step | counter_step | Step size of the internal counter. |
| Counter Start Value | counter_start | Initial value of the internal counter. |
| Counter Bitwidth | counter_width | Bitwidth of the internal counter. |
| Latency | latency | The latency of the block. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sin | OUT | Fix_(n_bits)_(n_bits-1) | Sine of the current phase, which is given by the counter. |
| cos | OUT | Fix_(n_bits)_(n_bits-1) | Cosine of the current phase, which is given by the counter. |

### Description

#### Usage

This block generates the sine and cosine of an oscillator with user-defined spacing (based on `counter_step` and `counter_width`) and bitwidth.

### Mixer

**Block:** Mixer (`mixer`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons, Ben Blackman

### Contents

### Summary

Digitally mixes an input signal (which can be several samples in parallel) with an LO of the indicated frequency (which is some fraction of the native FPGA clock rate).

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Frequency Divisions | freq_div | The (power of 2) denominator of the mixing frequency. |
| Mixing Frequency | freq | The numerator of the mixing frequency. |
| Number of Parallel Streams | nstreams | The number of samples that arrive in parallel. |
| Bit Width | n_bits | The bitwidth of LO samples. |
| BRAM Latency | bram_latency | The latency of sin/cos lookup table. |
| MULT Latency | mult_latency | The latency of mixing multipliers. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | IN | boolean | Takes in an impulse the cycle before the `dins` are valid. |
| dinX | IN | Fix_8_7 | Input X to be mixed and output on `realX` and `imagX`. |
| sync_out | OUT | boolean | This signal will be high the cycle before the data coming out is valid. |
| realX | OUT | Fix_(n_bits)_(n_bits-1) | Real output of mixed `dinX`. |
| imagX | OUT | Fix_(n_bits)_(n_bits-1) | Imaginary output of mixed `dinX`. |

### Description

### Usage

`Mixer` mixes the incoming data and produces both real and imaginary outputs.

$M$ = Frequency Divisions
$F$ = Mixing Frequency

M and F must both be integers, and M must be a power of 2. The ratio F/M should equal the ratio f/r where r is the data rate of the sampled signal. For example, an F/M of 3/16 would downmix an 800Msps signal with an LO of 150MHz.

### Negative Edge Detect

**Block:** Negative Edge Detect Block (`negedge`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

**Contents**

#### Summary

Outputs true if a boolean input signal is currently false, but was true during the last clock.

#### Mask Parameters

None.

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | Boolean | Input boolean signal. |
| out | out | Boolean | Negative-edge detected output boolean signal. |

#### Description

Outputs true if a boolean input signal is currently false, but was true during the last clock.

### Partial Delay

**Block:** Partial Delay Block (`partial_delay`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

For a set of parallel inputs which represent consecutive time samples of the same input signal, this block delays the stream by a dynamically selectable number of samples between 0 and (n_inputs-1).

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| No. of inputs. | n_inputs | The number of parallel inputs. |
| Mux Latency | latency | The latency of each mux block. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | ??? | ??? | Indicates the next clock cycle containing valid data |
| din | in | ??? | A number to be summed. |

### Description

Ex.: Mapping of 4 parallel input samples to output for delay = 2.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| … | 4 | 0 | … | $\longrightarrow$ | 6 | 2 | … | … |
| … | 5 | 1 | … | $\longrightarrow$ | 7 | 3 | … | … |
| … | 6 | 2 | … | $\longrightarrow$ | … | 4 | 0 | … |
| … | 7 | 3 | … | $\longrightarrow$ | … | 5 | 1 | … |

## Polyphase Real FIR Filter

**Block:** Polyphase Real FIR Filter (`pfb_fir_real`)
**Block Author**: Henry Chen
**Document Author**: Ben Blackman

## Contents

## Summary

This block, combined with an FFT, implements a real Polyphase Filter Bank which uses longer windows of data to improve the shape of channels within a spectrum.

## Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Size of PFB ($2^?$ pnts) | PFBSize | The number of channels in the PFB (this should also be the size of the FFT which follows). |
| Total Number of Taps | TotalTaps | The number of taps in the PFB FIR filter. Each tap uses 2 real multiplier cores and requires buffering the real and imaginary streams for $2^{*PFBSize*}$ samples. |
| Windowing Function | WindowType | Which windowing function to use (this allows trading passband ripple for steepness of rolloff, etc). |
| Number of Simultaneous Inputs ($2^?$) | n_inputs | The number of parallel time samples which are presented to the FFT core each clock. The number of output ports are set to this same value. |
| Make Biplex | MakeBiplex | Double up the inputs to match with a biplex FFT. |
| Input Bitwidth | BitWidthIn | The number of bits in each real and imaginary sample input to the PFB. |
| Output Bitwidth | BitWidthOut | The number of bits in each real and imaginary sample output from the PFB. This should match the bit width in the FFT that follows. |
| Coefficient Bitwidth | CoeffBitWidth | The number of bits in each coefficient. This is usually chosen to match the input bit width. |
| Use Distributed Memory for Coeffs | CoeffDistMem | Store the FIR coefficients in distributed memory (if = 1). Otherwise, BRAMs are used to hold the coefficients. |
| Add Latency | add_latency | Latency through adders in the FFT. |
| Mult Latency | mult_latency | Latency through multipliers in the FFT. |
| BRAM Latency | bram_latency | Latency through BRAM in the FFT. |
| Quantization Behavior | quantization | Specifies the rounding behavior used at the end of each butterfly computation to return to the number of bits specified above. |
| Bin Width Scaling (normal=1) | fwidth | PFBs give enhanced control over the width of frequency channels. By adjusting this parameter, you can scale bins to be wider (for values > 1) or narrower (for values < 1). |

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | IN | Boolean | Indicates the next clock cycle contains valid data |
| pol_in | IN | Inherited | The (real) time-domain stream(s). |
| sync_out | OUT | Boolean | Indicates that data out will be valid next clock cycle. |
| pol_out | OUT | Inherited | The (real) PFB FIR output, which is still a time-domain signal. |

#### Description

#### Usage

This block, combined with an FFT, implements a real Polyphase Filter Bank which uses longer windows of data to improve the shape of channels within a spectrum.

### Polyphase FIR Filter

**Block:** Polyphase FIR Filter (frontend for a full PFB) (`pfb_fir`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

#### Contents

#### Summary

This block, combined with an FFT, implements a Polyphase Filter Bank which uses longer windows of data to improve the shape of channels within a spectrum.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Size of PFB: ($2^?$) | PFBSize | The number of channels in the PFB (this should also be the size of the FFT which follows). |
| Total Number of Taps: | TotalTaps | The number of taps in the PFB FIR filter. Each tap uses 2 real multiplier cores and requires buffering the real and imaginary streams for $2^{\mathrm{PFBSize}}$ samples. |
| Windowing Function | WindowType | Which windowing function to use (this allows trading passband ripple for steepness of rolloff, etc). |
| Number of Simultaneous Inputs: ($2^?$) | n_inputs | The number of parallel time samples which are presented to the FFT core each clock. The number of output ports are set to this same value. |
| Make Biplex | MakeBiplex | Double up the inputs to match with a biplex FFT. |
| Input Bit Width | BitWidthIn | The number of bits in each real and imaginary sample input to the PFB. |
| Output Bit Width | BitWidthOut | The number of bits in each real and imaginary sample output from the PFB. This should match the bit width in the FFT that follows. |
| Coefficient Bit Width | CoeffBitWidth | The number of bits in each coefficient. This is usually chosen to match the input bit width. |
| Use Distributed Memory for Coefficients | CoeffDistMem | Store the FIR coefficients in distributed memory (if = 1). Otherwise, BRAMs are used to hold the coefficients. |
| Add Latency | add_latency | Latency through adders in the FFT. |
| Mult Latency | mult_latency | Latency through multipliers in the FFT. |
| BRAM Latency | bram_latency | Latency through BRAM in the FFT. |
| Quantization Behavior | quantization | Specifies the rounding behavior used at the end of each butterfly computation to return to the number of bits specified above. |
| Bin Width Scaling (normal = 1) | fwidth | PFBs give enhanced control over the width of frequency channels. By adjusting this parameter, you can scale bins to be wider (for values > 1) or narrower (for values < 1). |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | in | Boolean | Indicates the next clock cycle contains valid data |
| pol_in | in | Inherited | The (complex) time-domain stream(s). |
| sync_out | out | Boolean | Indicates that data out will be valid next clock cycle. |
| pol_out | out | Inherited | The (complex) PFB FIR output, which is still a time-domain signal. |

### Description

This block, combined with an FFT, implements a Polyphase Filter Bank which uses longer windows of data to improve the shape of channels within a spectrum.

## Positive Edge Detect

**Block:** Positive Edge Detect Block (`posedge`)

**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

**Contents**

#### Summary

Outputs true if a boolean input signal is true this clock and was false during the last clock.

#### Mask Parameters

None.

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | Boolean | Input boolean signal. |
| out | out | Boolean | Positive-edge detected output boolean signal. |

#### Description

Outputs true if a boolean input signal is true this clock and was false during the last clock.

### Power

**Block:** Power (`power`)
**Block Author**: Aaron Parsons
**Document Author**: Ben Blackman

### Contents

### Summary

Computes the power of a complex number.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Bit Width | BitWidth | The number of bits in its input. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| c | IN | 2*BitWidth Fixed point | A complex number whose higher BitWidth bits are its real part and lower BitWidth bits are its imaginary part. |
| power | OUT | UFix_(2*BitWidth)_(2*BitWidth-1) | The computed power of the input complex number. |

### Description

### Usage

The power block typically has a latency of 5 and will compute the power of its input by taking the sum of the squares of its real and imaginary components.

### Pulse Extender

**Block:** Pulse Extender Block (`pulse_ext`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

#### Summary

Extends a boolean signal to be high for the specified number of clocks after the last high input.

#### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Length of Pulse | pulse_len | Specifies number of clocks after the last high input for which the output is held high. |

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | Boolean | Input boolean signal. |
| out | out | Boolean | Pulse-extended boolean signal. |

#### Description

Extends a boolean signal to be high for the specified number of clocks after the last high input. If a new in pulse (input high) occurs, the counter determining the output pulse length is reset.

### RC Multiplier

**Block:** RC Multiplier (`rcmult`)
**Block Author**: Aaron Parsons
**Document Author**: Ben Blackman

### Contents

### Summary

Takes an input and sine and cosine value and gives out both real and imaginary outputs.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Latency | latency | The latency of the multipliers and of the `rcmult` block. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| d | IN | Inherited | The input to be multiplied by sine and cosine values. |
| sin | IN | Inherited | The sine value used to multiply `d` and generate the `imag` output. |
| cos | IN | Inherited | The cosine value used to multiply `d` and generate the `real` output. |
| real | OUT | Inherited | The result of multiplying `d` with `cos`. |
| imag | OUT | Inherited | The result of multiplying `d` with `sin`. |

### Description

#### Usage

This `rcmult` block takes an input value, `d`, and computes the real and imaginary components by multiplying by the `cos` and `sin`, respectively. The block has a delay of `latency` associated with it.

## Reorder

**Block:** Reorder (`reorder`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

## Summary

Permutes a vector of samples to into the desired order.

## Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Output Order | map | Assuming an input order of 0, 1, 2, . . . , this is a vector of the desired output order (e.g. [0 1 2 3]). |
| No. of inputs. | n_inputs | The number of parallel streams to which this reorder should be applied. |
| BRAM Latency | bram_latency | The latency of the BRAM buffer. |
| Map Latency | map_latency | The latency allowed for the combinatorial logic required for mapping a counter to the desired output order. If your permutation can be acheived by simply reordering bits (as is the case for bit reversed order, reverse order, and matrix tranposes with power-of-2 dimensions), a map latency of 0 is appropriate. Otherwise, 1 or 2 is a good idea. |
| Double Buffer | double_buffer | By default, this block uses single buffering (meaning it uses a buffer only the size of the vector, and permutes the data order in place). You can override this by setting this parameter to 1, in which case 2 buffers are used to permute the vector (saving logic resources at the expense of BRAM). |

## Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | in | Boolean | Indicates the next clock cycle contains valid data |
| en | in | Boolean | Indicates the current input data is valid. |
| din | in | Inherited | The data stream(s) to be permuted. |
| sync_out | out | Boolean | Indicates that data out will be valid next clock cycle. |
| valid | out | Boolean | Indicates the current output data is valid. |
| dout | out | Inherited | The permuted data stream(s). |

### Description

Permutes a vector of samples into the desired order. By default, this block uses a single buffer to do this. As vectors are permuted, the data placement in memory will go through several orders before it repeats. For large orders ( > 16) you should consider using double buffering, but otherwise, this block saves BRAM resources with only a modest increase in logic resources.

## Real-Imag to Complex

**Block:** Real-Imag to Complex Block (`ri_to_c`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

Concatenates real and imaginary inputs into a complex output. Useful for simplifying interconnects. See also c_to_ri.

### Mask Parameters

None.

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| r | in | Fix_x_y | Real data |
| i | in | Fix_x_y | Imaginary signed output, binary point specified by parameter. |
| c | out | UFix_x_0 | Complex input, real in MSB, imaginary in LSB. |

### Description

Conveniently combines real and imaginary components of a number into a single wire. See also c_to_ri.

## Square Transposer

**Block:** Square Transposer (`square_transposer`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

Presents a number of parallel inputs serially on the same number of output lines.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Number of inputs | n_inputs | The number of parallel inputs (and outputs). |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | in | Boolean | Indicates the next clock cycle contains valid data |
| In | in | Inherited | The stream(s) to be transposed. |
| sync_out | out | Boolean | Indicates that data out will be valid next clock cycle. |
| Out | out | Inherited | The transposed stream(s). |

### Description

(Out1, Out2, etc.) appear aligned:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| In1 | d12 | d8 | d4 | d0 | ⟶ | d3 | d2 | d1 | d0 | Out1 |
| In2 | d13 | d9 | d5 | d1 | ⟶ | d7 | d6 | d5 | d4 | Out2 |
| In3 | d14 | d10 | d6 | d2 | ⟶ | d11 | d10 | d9 | d8 | Out3 |
| In4 | d15 | d11 | d7 | d3 | ⟶ | d15 | d14 | d13 | d12 | Out4 |

### Stopwatch

**Block:** Stopwatch (`stopwatch`)
**Block Author**: Jason Manley
**Document Author**: Jason Manley

#### Contents

#### Summary

Counts the number of clocks between a start and stop pulse.

#### Mask Parameters

None.

#### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| start | in | boolean | Start counting |
| stop | in | boolean | Stop counting and hold value until reset received |
| reset | in | boolean | Reset back to zero. |
| count_out | out | ufix_32_0 | Number of clocks elapsed since start pulse received. |

#### Description

This block counts the number of clocks between a start and stop pulse. This value is held until a reset is received. If another start pulse is received before the reset, counting resumes from where it left-off. If a reset is received mid-way through a count (ie before a stop pulse) then the stopwatch will be reset and await another start pulse before it restarts counting.

### Enabled Sync Delay

**Block:** Enabled Sync Delay (`sync_delay_en`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

Delay an infrequent boolean pulse by the specified number of enabled clocks.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Delay Length | DelayLen | The length of the delay. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| in | in | boolean | The boolean signal to be delayed. |
| en | in | boolean | To be asserted when input is valid. |
| out | out | boolean | The delayed boolean signal, output 1 clock after en. |

### Description

Delay an infrequent boolean pulse by the specified number of enabled clocks. If the input pulse repeats before the output pulse is generated, an internal counter resets and that output pulse is never generated.

## Programmable Sync Delay

**Block:** Programmable Sync Delay (`sync_delay_prog`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons

### Contents

### Summary

Delay an infrequent boolean pulse by a run-time programmable number of enabled clocks. If the input pulse repeats before the output pulse is generated, an internal counter resets and that output pulse is never generated. When delay is changed, some randomly determined samples will be inserted/dropped from the buffered stream.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Max Delay ($2^?$) | MaxDelay | The maximum length of the delay. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync | in | ??? | The boolean signal to be delayed. |
| delay | in | ??? | The run-time programmable delay length. |
| sync_out | out | ??? | The delayed boolean signal. |

### Description

Delay an infrequent boolean pulse by a run-time programmable number of enabled clocks. If the input pulse repeats before the output pulse is generated, an internal counter resets and that output pulse is never generated. When delay is changed, some randomly determined samples will be inserted/dropped from the buffered stream.

## Sync Pulse Generator

**Block:** Sync Pulse Generator (`sync_gen`)
**Block Author**: Mark Wagner, Suraj Gowda
**Document Author**: Suraj Gowda, Billy Mallard

### Contents

### Summary

Generates a sync pulse of an appropriate period for a design.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Simulation Accumulation Length | gen_acc_len | The accumulation length that will be used for generation. |
| FFT Size | fft_size | The size of the FFT. |
| Simultaneous Inputs (FFT) | fft_simult_inputs | The number of data inputs into the FFT. |
| Taps in pfb_fir | pfb_fir_taps | The number of PFB filter taps. If your design does not use a pfb_fir, set this to 1. |
| Reorder Orders | reorder_vec | A vector of the orders of the reorder blocks inside the FFT. In your model, the reorder blocks are in fft/fft_biplex/biplex_cplx_unscrambler. The orders should be displayed under the blocks. |
| Scale | scale | The amount to scale the sync period by. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sync_out | out | Boolean | The generated sync pulse. |

### Description

The `sync_gen` block computes the sync pulse period as:

$$SyncPeriod = (acc\_num)*(pfb\_fir\_taps)*LCM(reorder\_vec)*(\frac{fft\_size}{fft\_simult\_inputs})$$

This formula is derived in Memo #25.

### Windowed X-Engine

**Block:** Windowed X-Engine (`win_x_engine`)

**Block Author**: Jason Manley, Aaron Parsons, Terry Filiba
**Document Author**: Jason Manley

### Contents

### Summary

CASPER X engine with added internal valid data masking functionality. Based on Aaron Parsons' design.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Number of antennas | n_ants | Number of antennas to process. |
| Bit width of samples in | n_bits | Bit width of each input sample number. Usually set to 4, resulting in 16 bit input numbers (2 polarizations, complex numbers). |
| Accumulation length | acc_len | Specified per antenna. |
| Adder latency | add_latency | Used to set the latency of internal adders. |
| Multiplier latency | mult_latency | Used to set the latency of internal multipliers. |
| BRAM latency | bram_latency | Used to set the latency of internal BRAMs. |
| Implementation: Multiplier type | use_ded_mult | Select the type of multipliers to use. Can be a single number or array - see below. |
| Implementation: Delay type | use_bram_delay | Selects the type of delays to implement. Single number configures all internal taps. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| ant | in | variable width. see below. | Input port for incoming antenna data. |
| sync_in | in | boolean | Synchronization pulse. New window begins clock cycle after sync received. |
| win-dow_valid | in | boolean | Indicates incoming antenna data is valid. Must remain constant for acc_len*n_ants. |
| acc | out | variable width. see below. | Output data. |
| valid | out | boolean | Indicates data on acc is valid. |
| sync_out | out | boolean | Passthrough for sync pulses. |

### Description

### Introduction

The CASPER X engine is a streaming architecture block where complex antenna data is input and accumulated products (for all cross-multiplications) are output in conjugated form. Because it is streaming with valid data expected on every clock cycle, data is logically divided into windows. These windows can either be valid (in which case the computation yields valid, outputted results) or invalid (in which case computation still occurs, but the results are ignored and not presented to the user).

### Input format

Data is input serially: `antenna A, antenna B, antenna C` etc. Each antenna's data consists of dual polarization, complex data. The bit width of each component number can be set as a parameter, `n_bits`. The X-engine thus expects these four numbers of `n_bits` to be concatenated into a single, unsigned number. CASPER convention dictates that complex numbers are represented with higher bits as real and lower bits as imaginary. The top half of the input number is polarization one and the lower half polarization two.

The internals of the block are reset with the reception of a sync pulse. A new window begins on the very next clock cycle. Each window is `int_len` * `n_ants` clock cycles long. The data for each antenna is input for `acc_len` clock cycles.

For example, for `n_bits` of 4 and `acc_len` of 2, the input to the X-engine would be 16 bits every clock cycle mapped as follows:

| ... | $t_4$ | $t_3$ | $t_2$ | $t_1$ | $t_0$ | $\longrightarrow$ |
|---|---|---|---|---|---|---|
| ... | $C_{1\text{real}}$ | $B_{1\text{real}}$ | $B_{1\text{real}}$ | $A_{1\text{real}}$ | $A_{1\text{real}}$ | most_sig 4b$\longrightarrow$ |
| ... | $C_{1\text{imag}}$ | $B_{1\text{imag}}$ | $B_{1\text{imag}}$ | $A_{1\text{imag}}$ | $A_{1\text{imag}}$ | 4b$\longrightarrow$ |
| ... | $C_{2\text{real}}$ | $B_{2\text{real}}$ | $B_{2\text{real}}$ | $A_{2\text{real}}$ | $A_{2\text{real}}$ | 4b$\longrightarrow$ |
| ... | $C_{2\text{imag}}$ | $B_{2\text{imag}}$ | $B_{2\text{imag}}$ | $A_{2\text{imag}}$ | $A_{2\text{imag}}$ | least_sig 4b$\longrightarrow$ |

X-engine input with `acc_len` of 2.

The `window_valid` line is expected to remain constant for the duration of each window. If it is high, the output is considered valid and captured into the output FIFO buffer. With the close of that window, the output will be presented to the user as valid data on every second clock pulse. If `window_valid` was held low, the data is ignored.

With the close of one window, anther begins directly afterwards. Data can thus be streamed in and out continuously, while a sync pulse will force the start of a new window.

## Output Format

The windowed X-engine will produce $num_baselines = n_ants \times \frac{n_ants+1}{2}$ valid outputs. The unwindowed x engine produces $num_baselines = n_ants \times (\frac{n_ants}{2} + 1)$ results. The extra valids are a result of the algorithm employed and are masked out by the internal `x_engine_mask`.

Generally, the output of the X-engine configured for `N` antennas can be mapped into a table with $\frac{n_ants}{2} + 1$ columns and *N* rows as follows:

| | | | | | | |
|---|---|---|---|---|---|---|
| 1st | 0 × 0 | 0 × N | 0 × (N-1) | 0 × (N-2) | … | ⟶ |
| 2nd | 1 × 1 | 0 × 1 | 1 × N | 1 × (N-1) | … | ⟶ |
| 3rd | 2 × 2 | 1 × 2 | 0 × 2 | 2 × N | … | ⟶ |
| 4th | 3 × 3 | 2 × 3 | 1 × 3 | 0 × 3 | … | ⟶ |
| 5th | 4 × 4 | 3 × 4 | 2 × 4 | 1 × 4 | … | ⟶ |
| 6th | 5 × 5 | 4 × 5 | 3 × 5 | 2 × 5 | … | ⟶ |
| … | … | … | … | … | … | ⟶ |

Each table entry represents a valid output. Data is read out right to left, top to bottom. Bracketed values are from previous window.

As an example, consider the output for a 4 antenna system (with antennas numbered A through D):

| | | | |
|---|---|---|---|
| 1st | **AA** | prev win DA | prev win CA |
| 2nd | **BB** | **AB** | prev win BD |
| 3rd | **CC** | **BC** | **AC** |
| 4th | next win AA | **CD** | **BD** |
| 5th | next win BB | next win AB | **DB** |

Boldfaced type represents current valid window of data. Data is read out right to left, top to bottom. Non-boldfaced data is masked.

Thanks to the inclusion of the `x_engine_mask` block, X-engine output duplicates (observed in rows 5 and 6 above) are automatically removed. The output of a 4 antenna windowed X-engine is thus `AA, AB, BB, AC, BC, CC, BD, CD, DD, DA`.

## X-Engine TVG

**Block:** X-Engine TVG (`xeng_tvg`)
**Block Author**: Jason Manley
**Document Author**: Jason Manley

## Contents

### Summary

Basic test vector generator for CASPER X-engines.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Number of Antennas ($2^n$) | ant_bits | Bitwidth of the number of antennas in the system. |
| Bitwidth of Samples in | bits_in | Bitwidth of component of the input. |
| X integration length ($2^n$) | x_int_bits | Bitwidth of X-engine accumulation length. |
| Sync Pulse Period ($2^n$) | sync_period | Bitwidth of number of valids between sync pulses. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| tvg_sel | in | ufix_2_0 | TVG selection. 0=off (passthrough), 1-3=TVG select. |
| data_in | in | inherited: bits_in*4 | Data in for passthrough. |
| valid_in | in | boolean | Valid in made available for passthrough. |
| sync_in | in | boolean | Sync in made available for passthrough. |
| data_out | out | inherited: bits_in*4 | ??? |
| sync_out | out | boolean | ??? |
| valid_out | out | boolean | ??? |

### Description

This block generates data in a format suitable for input to a CASPER X-engine. The `tvg_sel` line selects the TVG. If set to zero, it is configured for passthrough and all input signals are propagated to the output (TVG is off). Values one through three select a TVG pattern. In this case, sync pulses are generated internally and valid data is output all the time. The three patterns are as follows:

1. Inserts a counter representing the antenna number. All real values count up from zero and imaginary values counting down from zero. ie., antenna four would have the value $4\ \ 4i$ inserted.
2. Inserts the same constant for all antennas: $Pol_{1real} = 0.125$, $Pol_{1imag} = \ 0.75$, $Pol_{2real} = 0.5$ and $Pol_{2imag} = \ 0.25$
3. User selectable values for each antenna. Input registers named `tv0` through `tv7` are input cyclically. Each value is input for `x_int_bits` clocks.

### Communication Blocks

*ten_gbe* (10GbE Transceiver)

*XAUI* (XAUI Transceiver)

### 10 GbE Transceiver

**Block:** 10GbE Transceiver (`ten_gbe`)
**Block Author**: David George
**Document Author**: David George

#### Contents

#### Summary

This block sends and receives UDP frames (packets). It accepts a 64 bit wide data stream with user-determined frame breaks. The data stream is wrapped in a UDP frame for transmission. Incoming UDP packets are unwrapped and the data presented as a 64 bit wide stream.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Port | port | Selects the physical CX4 port. |
| Shallow RX Fifo | rx_dist_ram | Selects shallow distributed RAM rather than BRAM for the RX FIFO. This option should only be used if the application clock is faster than 156.25 MHz. Constant overruns will most likely occur for clock frequency lower than this. |
| Pre-emphasis | pre_emph | Selects the default pre-emaphasis to use over the physical link. Currently unused. |
| Differential Swing | swing | Selects the size of the differential swing to use in mV. Currently unused. |
| Enable Fabric on Startup | fab_en | This will enable the 10Ge interface on start-up, negating the requirement of software intervention. |
| Fabric MAC Addresss | fab_mac | Sets the default MAC, usually configured by software. |
| Fabric IP Addresss | fab_ip | Sets the default IP, usually configured by software. |
| Fabric UDP Port | fab_port | Sets the default UDP Port, usually configured by software. |
| Fabric Gateway | fab_gate | Sets the default gateway address, usually configured by software. |
| Enable CPU RX | cpu_rx_en | Optionally disable the CPU receive interface to save 2 BRAMS, if that interface is not required. |
| Enable CPU TX | cpu_tx_en | Optionally disable the CPU trasmit interface to save 2 BRAMS, if that interface is not required. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| rst | in | boolean | Resets the transmit and receive memories when asserted |
| tx_data | in | UFix_64_0 | The data stream to be transmitted |
| tx_valid | in | boolean | The core accepts the data on `tx_data` into the buffer while this line is high |
| tx_dest_ip | in | UFix_32_0 | Selects the IP address of the receiving device |
| tx_dest_port | in | UFix_16_0 | Selects the listening port of the receiving device (UDP port) |
| tx_end_of_frame | in | boolean | Signals the transceiver to begin transmitting the buffered frame (ie signals end of the frame). This signal must pulse high on the same clock cycle as the final tx_valid signal |
| rx_ack | in | boolean | Used to acknowledge reception of the data currently on rx_data and signals the transceiver to produce the next 64 bits from the receiver FIFO. |
| rx_overrun_ack | in | boolean | Used to reset the RX state machine when a RX overrun occurs. |
| led_up | out | boolean | Indicates a link on the port |
| led_rx | out | boolean | Represents received traffic on the port |
| led_tx | out | boolean | Represents transmitted traffic on the port |
| tx_afull | out | boolean | Indicates that the TX FIFO is almost full |
| tx_overflow | out | boolean | Indicates that the TX FIFO has overflowed |
| rx_data | out | UFix_64_0 | The received data stream. |
| rx_valid | out | boolean | Indicates that the data on rx_data is valid (indicates a packet, or partial packet is in the RX buffer). |
| rx_source_ip | out | UFix_32_0 | Represents the IP address of the sender of the current packet. |
| rx_source_port | out | UFix_16_0 | Represents the sender's UDP port of the current packet. |
| rx_end_of_frame | out | boolean | Goes high to indicate the end of the received frame. |
| rx_bad_frame | out | UFix_1_0 | Indicates a CRC error on the frame immediately received when asserted simultaneously with `rx_end_of_frame`. |
| rx_overrun | out | UFix_1_0 | Indicates that an RX buffer overrun when asserted simultaneously with `rx_end_of_frame`. |

### Description

This document is a draft and requires verification.

### Configuration

This core will typically be configured by software by setting the following OPB registers: MAC, IP, UDP port, ARP table, gateway and fabric enable. However, there are currently parameters to set these values (except for the ARP table) at build time.

### Transmitting

To transmit, data is clocked into the TX buffer through `tx_data` in 64 bit wide words using `tx_valid`. When sending the final word of data in the packet, simultaneously pulse the `tx_end_of_frame` line; the transceiver will add a UDP wrapper addressed to `tx_dest_ip:tx_dest_port` and begin transmission immediately. Note that tx_dest_ip and tx_dest_port are only looked at when tx_end_of_frame is pulsed high, thus the value must be valid for that clock period but is irrelevant otherwise.

The transmit FIFO can only handle frame sizes of up to 8704 bytes, or 1088 64 bit words. As the TX FIFO approaches full, the `tx_afull` signal will be asserted. This is an indication that the application should stop entering data into the core. If the application overflows the TX buffer, the `tx_overflow` signal will be asserted. When the overflow is asserted the TX interface will block, requiring a reset (using `rst`) to return to full functionality.

### Receiving

Upon receipt of a packet, `rx_valid` will go high, with the received data presented on `rx_data` in 64 bit wide words. You acknowledge receipt of this data using `rx_ack`, at which point the next data word will be presented. When the end of the packet is reached, `rx_end_of_frame` will go high. If the a CRC error occurred when receiving the packet, the `rx_bad_frame` signal will be asserted.

If the RX data overran the RX FIFO, the `rx_overrun` signal will be asserted along with `rx_end_of_frame`. When this occurs the RX state machine will block until the `rx_overrun_ack` is asserted. It is recommended that the `rx_overrun` be directly connected to the `rx_overrun_ack` signal.

There is no hard maximum frame size for RX. However, care must be taken not to overrun the rx buffer.

### Addressing

To transmit, the IPv4 address is represented as a 32 bit binary number (whereas it's usually represented as four 8 bit decimal numbers). For example, if you wanted to send all packets to `192.168.1.1`, you would enter

$$192 \times 2^{24} + 168 \times 2^{16} + 1 \times 2^{8} + 1 = 3232235777$$

as the IP address. The port is represented by a 16 bit number, allowing full addressing of the UDP port range. Ports below 1024 are generally reserved for Linux kernel and Internet functions. Ports 1024 - 49151 are registered for specific applications and may not be used without IANA registration. To ensure inter-operability and compatibility, we recommend using dynamic (private) ports 49152 through 65535.

### LED Outputs

The LED lines indicate port activity and can be connected to external GPIO LED interfaces. Bear in mind that even if no packets are being transmitted or received through the Simulink interface block, miscellaneous configuration packets are still sent and may be received by the microprocessor core. This activity will also be reflected on the activity LEDs.

### Operation

Apart from configuring the block, the processor is also used to map the routing tables. ARP requests and responses are handled by the CPU.

### XAUI

**Block:** XAUI Transceiver (`XAUI`)
**Block Author**: Pierre Yves Droz, Henry Chen
**Document Author**: Jason Manley

## Contents

## Summary

XAUI block for sending and receiving point-to-point, streaming data over the BEE2 and iBOB's CX4 connectors. NOTE: A new version of this block is in development.

## Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Demux | demux | Selects the width of the data bus. 1 for 64 bits, 2 for 32 bits. |
| Port | port | Selects the physical CX4 port on the iBOB or BEE2. The iBOB has two ports; the BEE2 has two for the control FPGA and four for each of the user FPGAs. CORR is not used by CASPER. |
| Pre-emphasis | pre_emph | Selects the pre-emaphasis to use over the physical link. Default: 3 (see Xilinx documentation) |
| Differential Swing | swing | Selects the size of the differential swing to use in mV. Default: 800 (see Xilinx documentation) |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| rx_get | in | boolean | Used to request the next data word from the RX buffer. |
| rx_reset | in | boolean | Resets the receive subsystem. |
| tx_data | in | ufix_64_0 or ufix_32_0 | Accepts the next data word (64 or 32 bits) to be transmitted. |
| tx_outofband | in | ufix_8_0 or ufix_4_0 | Accepts the next data word (8 bits if demux=1, 4 bits if demux=2) to be transmitted through the out-of-band channel. |
| tx_valid | out | boolean | Clocks the transmit data into the transceiver. Data is clocked into the buffer while this line is high. |
| rx_data | out | ufix_64_0 | Outputs the received data stream. |
| rx_outofband | out | ufix_8_0 or ufix_4_0 | Outputs the out-of-band received data stream. |
| rx_empty | out | boolean | Indicates that the receive buffer is empty. |
| rx_valid | out | boolean | Indicates that data has been received. |
| rx_linkdown | out | boolean | Indicates that the link is down (eg. faulty cable). |
| tx_full | out | boolean | Indicates the transmit buffer is full. |
| rx_almost_full | boolean | inherited | Indicates the receive buffer is full. |

### Description

### Demux

Perhaps a misnomer, this parameter describes the width of the data bus rather than a selection of two muxed streams on one channel. At 156MHz XAUI clock, the maximum transmission speed is 64bits * 156.25 MHz = 10Gbit/s. For BEE or iBOB designs clocked at rates above 156MHz, clocking-in 64 bit data on every clock cycle would cause the XAUI block's FIFO buffers to overflow. The `demux` option is provided which halves the input data bus width to 32 bits and enables data to be clocked-in on every FPGA clock cycle. Along with the data bus width, the `out of band` bus width is also halved to 4 bits.

### Out of band signals

Out of band signals are guaranteed to arrive at the same time as the data word with which they were sent. Out-of-band data is only transmitted across the physical link if the input to `tx_outofband` changes and is clocked in as valid (`tx_valid`). In other words, if you keep `tx_outofband` constant, no additional bandwidth is consumed (the in-band signals are transmitted as normal). When data is clocked into the transmitter, it will appear out the receiver as if the `tx_outofband` and `tx_data` arrived simultaneously. Care should be taken to ensure that the data clocked into `tx_outofband` and `tx_data` does not exceed the XAUI's maximum transmission rate (approximately 10Gbps for 156.25MHz clock). Each change of `tx_outofband` (be it one bit or eight bits) requires 64 bits (a full word) to transmit. This bus width is 8 bits if `demux` is not selected (set to 1), and 4 bits if it is set to 2.

### System Blocks

*adc* (ADC)

*x64_adc* (64 Channel, 12 bit ADC: 64ADCx64-12)

*dac* (DAC)

*dram* (DRAM)

*gpio_bidir* (Bi-directional GPIO)

*gpio* (GPIO)

*qdr* (QDR)

*snapshot* (Snapshot Capture)

*snap* (Snapshot Capture)

*snap64* (64-Bit Snapshot Capture)

*software_register* (Software Register)

*sram* (SRAM)

*XSG core config* (XSG Core Config)

*Gaussian Random Number Generator* (Gaussian Random Number Generator)

*Correlation Control Block* (CCB)

## ADC

**Block:** ADC (`adc`)
**Block Author**: Pierre Yves Droz
**Document Author**: Ben Blackman

### Contents

### Summary

The ADC block converts analog inputs to digital outputs. Every clock cycle, the inputs are sampled and digitized to 8 bit binary point numbers in the range of [-1, 1) and are then output by the adc.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| ADC board | adc_brd | Select which ADC port to use on the IBOB. |
| ADC clock rate (MHz) | adc_clk_rate | Sets the clock rate of the ADC, must be at least 4x the IBOB clock rate. |
| ADC interleave mode | adc_interleave | Check for 1 input, uncheck for 2 inputs. |
| Sample period | sample_period | Sets the period at which the adc outputs samples (ie 2 means every other cycle). |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sim_in | IN | double | The analog signal to be digitized if interleave mode is selected. Note: For simulation only. |
| sim_i | IN | double | The first analog signal to be digitized if interleave mode is unselected. Note: For simulation only. |
| sim_q | IN | double | The second analog signal to be digitized if interleave mode is unselected. Note: For simulation only. |
| sim_sync | IN | double | Takes a pulse to be observed at the output to measure the delay through the block. Note: For simulation only. |
| sim_data_valid | IN | double | A signal that is high when inputs are valid. Note: For simulation only. |
| oX | OUT | Fix_8_7 | A signal that represents sample X+1 (Ex. o0 is the 1st sample, o7 is the 8th sample). Used if interleave mode is on. |
| iX | OUT | Fix_8_7 | A signal that represents sample X+1 (Ex. i0 is the 1st sample, o3 is the 4th sample). Used if interleave mode is off. |
| qX | OUT | Fix_8_7 | A signal that represents sample X+1 (Ex. q0 is the 1st sample, q3 is the 4th sample). Used if interleave mode is off. |
| outofrangeX | OUT | boolean | A signal that represents when samples are outside the valid range. |
| syncX | OUT | boolean | A signal that is high when the sync pulse offset by X if interleave mode is unselected, or 2X if interleave mode is selected is high (Ex. sync2 is the pulse offset by 2 if interleave is off or offset by 4 if interlave is on). |
| data_valid | OUT | boolean | A signal that is high when the outputs are valid. |

### Description

### Usage

The ADC block can take 1 or 2 analog input streams. The first input should be connected to input i and the second to input q if it is being used. The inputs will then be digitized to `Fix_8_7` numbers between [-1, 1). For a single input, the `adc` samples its input 8 times per IBOB clock cycle and outputs the 8 samples in parallel with o0 being the first sample and o7 the last sample. For 2 inputs, the `adc` samples both inputs 4 times per IBOB clock cycle and then outputs them in parallel with i0-i3 corresponding to input i and q0-q3 corresponding to input q. In addition to having 2 possible inputs, each IBOB can interface with 2 `adc`s for a total of 4 inputs or 2 8-sample inputs per IBOB.

### Connecting the Hardware

To hook up the ADC board, attach the clock SMA cable to the clk_i port, the first input to the I+ port, and the second input to the Q+ port. Check the hardware on the ADC board near the input pins. There should be for 4 square chips in a straight line. If there are only 3, the second input, Q+, may not work. Note that if you chose `adc0_clk`, make sure to plug the ADC board in to the adc0 port. The same applies if you chose `adc1_clk` to plug the board into adc1 port. If you are using both ADCs, then you need to plug a clock into both clk_i inputs and you should probably run them off of the same signal generator.

### ADC Background Information

The ADC board was designed to mate directly to an IBOB board through ZDOK connectors for high-speed serial data I/O. Analog data is digitized using an Atmel AT84AD001B dual 8-bit ADC chip which can digitize two streams at 1 Gsample/sec or a single stream at 2 Gsample/sec. This board may be driven with either single-ended or differential inputs.

## X64 ADC

**Block:** x64_adc (`x64_adc`)
**Block Author**: Jack Hickish, David George
**Document Author**: Jack Hickish

### Contents

### Summary

The x64_ADC block converts 64 analog inputs to digital outputs. Every clock cycle, the inputs are sampled and digitized to 12 bit binary point numbers in the range of [-1, 1).

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| ADC clock rate (MHz) | adc_clk_rate | Sets the rate of the ADC sampling clock. The ROACH clock is derived from the ADC clock and should be 4x the clock rate entered here. |
| Include SPI interface? | spi | If checked, includes the ports and logic necessary to set the ADC control registers via SPI. |
| GPIO bank | ctrl_gpio | The ADC SPI and reset interfaces are not routed through the ZDOK connector. This parameter selects which of the ROACH GPIO banks to use to control them. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| sim[0:15] | IN | double | sim<n> should be connected to analogue channels 4n:4n+3 to be digitized. Note: For simulation only. |
| sim_sync | IN | double | Input should be high when channel 4n is present on input sim<n>. Note: For simulation only. |
| adc_rst | IN | Bool | Active high reset signal, used to reset FIFOs and adc chip. |
| sdata | IN | UFix_8_0 | Data to be written to the ADC registers over SPI. sdata[7:4] represents the register address, sdata[3:0] represents the new value to be written. Present only when SPI interface is enabled. |
| spi_strb | IN | Bool | SPI write strobe. When a positive edge is detected on this port an SPI write is initiated using the data and address presented on input sdata. Present only when SPI interface is enabled. |
| dout[0:15] | OUT | Fix_12_F | Four consecutive values of the signal represent a single time sample from four channels, with dout<n> representing channels 4n:4n+3. |
| chan_sync | OUT | UFix_8_0 | A signal which goes high when a sample from channel 4n is present on output dout<n>. The 8 bits of the signal are used to sync the 8 individual ADC chips on the ADC card. |

### Description

### Background

The x64_adc block is an interface to the 64ADCx64-12 board developed by Rick Raffanti. The board is based around 8 Texas Instruments ADS5272 chips, each digitizing 8 signals at 12 bits resolution and up to 65MSa/s. The ADCs can be clocked by an on-board 50MHz oscillator, or an external clock source.

### Connecting the ADC

The 64ADCx64-12 is a twin Z-DOK card. Only one can be connected to a ROACH board.

### Clock Selection

The x64_adc card includes an on-board 50MHz oscillator, but can also be driven by an external clock.

Header J4 controls selection of the ADC clock source. Leave J4 open to use the on-board 50MHz oscillator, or jumper J4 to use an external clock applied across resistor R9 via pins 36 and 38 of header J13.

### Reset signal

The reset pin is located on pin 4 of J13. This signal is active low, and should be held high for normal operation. The yellow block interface is configured to drive the ADC reset pin via GPIO<A|B>_0, depending on the block parameter specified by the user. Note that the reset on the yellow block interface is active HIGH. When the ADC is held in reset, data output on the yellow block data lines will be the value -1 for all channels, in Fix_12_11 format.

### SPI Interface

Various ADC features (including test patterns) can be activated by using a Serial Peripheral Interface to set the registers as defined in the ADS5272 data sheet. Physical connections are as follows:
**ADC pins**

- SCLK: pin 36, J12
- SDATA: pin 38, J12
- nCS: (ADC chip *n*): pin (20 + 2*n*), J12

**ROACH pins**

- SCLK: GPIO<A|B>_2
- SDATA: GPIO<A|B>_1
- nCS: GPIO<A|B>_3

Where the GPIO port to use is determined by user-specified mask parameter. To write SPI registers, the "include SPI interface" option should be activated in the x64_adc yellow block parameters. This should expose the input ports "sdata" and "spi_strb" to the user. When a positive edge is detected on spi_strb, the data on sdata[3:0] is written to address sdata[7:4]. Currently, due to limited ROACH GPIO and lack of requirement, only one nCS signal is used by the yellow block. This can be connected to all ADC nCS pins, to write registers on all chips simultaneously.

### Usage

The x64_adc block can take 64 analog input streams. The inputs are digitized to `Fix_12_11` numbers between [-1, 1).

### ADC Synchronization

The ADC card uses 8 separate chips, each providing its own clock over ZDOK to the FPGA. Rather than use all 8 clocks (some of which are not connected to clock enabled FPGA pins), a single clock is used, and the software calibration script File:X64 adc cal.txt is run to ensure that data from all ADC chips is properly aligned.

Note: There may be a problem while running this script as is. In case there is a problem While running the version available in the repository and throws the following error: unknown'x64_adc_ctrl' variable, it can be fixed by adding the following line to the core_info.tab in the local repository (mlib_devel/blob/master/xps_base/XPS_ROACH_base/core_info.tab)

1. IF# strcmp(get(b,'type'),'xps_x64_adc')#x64_adc_ctrl 3 10000 100

Also, the following line of calibration script which is no longer supported by the 'corr' package can be removed (it seems this line was just meant for debugging).

fclk_sampled = self.bit_string((val0&0x0fff),12)

### Data Output

The 64 channels digitized by the ADC are presented to the user as 16 data output signals. Each signal will cycle through four multiplexed channels every four clock cycles. For example, in four consecutive clock cycles a sample from channels 0,1,2,3 will appear on output "dout0". In the following four clock cycles, the next time sample will appear. Output dout<n> is responsible for samples from channels 4*n*, 4*n*+1, 4*n*+2 and 4*n*+3. Physically, ADC chip *m* is responsible for channels 8*m*, 8*m*+1, . . . , 8*m*+7.

It is possible to identify the channels presented on each output by observing the chan_sync output, which is high when sample 4*n* is present on output dout<n>. The 8 bits of chan_sync give the sync flag associated with each of the 8 ADC chips. Proper calibration should ensure that all chips are synchronized. In this case, the chan_sync output should output zero, with the value 255 appearing once every four clocks.

### Data Input

Data can be input for simulation using the sim<n> and sim_sync inputs. These inputs are passed straight to the dout<n> and chan_sync outputs, and should be controlled accordingly, taking into account the data output details above.

### 64ADCx64-12

ADC64 Block Diagram

PXS Test Jig

PXSADC Top Level

PXS Spreadsheet

The 64 input 12 bit ADC board was developed by Rick Raffanti. the board has been tested at 65 Msps using Rick's verilog interface, but the simulink yellow block has only been tested at 50 Msps so far.

### DAC

**Block:** DAC (`dac`)
**Block Author**: Henry Chen
**Document Author**: Ben Blackman

**Contents**

### Summary

The DAC block converts 4 digital inputs to 1 analog output. The `dac` runs at 4x FPGA clock frequency, outputting analog converted samples 0 through 3 each FPGA clock cycle.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| DAC board | dac_brd | Select which IBOB port to run this `dac`. |
| DAC clock rate (MHz) | dac_clk_rate | The clock rate to run the `dac`. Must be 4x FPGA clock rate. |
| Sample period | sample_period | Sets the period at which the `dac` outputs samples (ie 2 means every other cycle). |
| Show Implementation Parameters | show_param | Allows the user to set the implementation parameters. |
| Invert output clock phase | invert_clock | When unchecked, the `dac` samples the data aligned with the clock. When checked, the `dac` samples the data aligned with an inverted clock. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| dataX | IN | Fix_9_8 | One of 4 digital inputs to be converted to analog. |
| sim_out | OUT | double | Analog output of `dac`. Note: For simulation only. |

### Description

### Usage

The `dac` takes 4 `Fix_9_8` inputs and outputs an analog stream. The `dac` runs at 4x the FPGA clock speed.

To be updated.

## DRAM

**Block:** DRAM (`dram`)
**Block Author**: Pierre Yves Droz (BEE2), David George(ROACH)
**Document Author**: Jason Manley, Laura Spitler

### Contents

### Summary

This block interfaces to the BEE2+ROACH's 1GB DDR2 ECC DRAM modules. Commands that are clocked-in are executed with an unknown delay, however, execution order is maintained. The underlying controller for the BEE2 and the ROACH are different and not all features are supported across both platforms (see below for details).

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| DIMM | dimm | Selects which physical DIMM to use (four per user FPGA). |
| Data Type | arith_type | Inform Simulink how it should interpret the stored data. |
| Data binary point | bin_pt | Inform Simulink how it should interpret the stored data - specifically, the bit position in the word where it should place the binary point. |
| Datapath clock rate (MHz) | ip_clock | Clock rate for DRAM. Default: 200MHz (400DDR). |
| Sample period | sample_period | Is significant for clocking the block. Default: 1 |
| Simulate DRAM using ModelSim | use_sim | Requires the addition of the ModelSim block at the top level of the design. Used to simulate DRAM block only. |
| Lesser Simulation Address Width | ??? | If the ModelSim simulation is disabled a very basic simulation using BRAMs will be performed. This parameter selects the address width to the bram memory and cannot exceed 20 (or so) bits. |
| Enable bank management | bank_mgt | *Advise leave off for BEE2.* Allows multiple banks to be open at the same time. *Always enabled on ROACH (setting ignored).* |
| Use wide data bus (288 bits) | wide_data | Burst writes require 288 bits. If not selected, provide a 144 bit bus which needs to be supplied with data in consecutive clock cycles to form the 288 bits. 288 bit bus can make for challenging routing! *Not implemented on ROACH.* |
| Use half-burst | half_burst | Only store 144 bits per burst (wastes half capacity as the second 144 bits are unusable). If enabled, requires at least two clock cycles to store 144 bits. Second clock cycle's data is forfeited. *Not implemented on ROACH.* |
| Use BRAM FIFOs *(ROACH only)* | bram_fifos | Use blockRAM FIFO's in DRAM controller. This is required only if the application clock rate is less than the dram clock rate to avoid overflows on the read interface. By default distributed RAM will be used which exhibits better timing performance and reduces BRAM resources. |
| Include CPU Interface *(ROACH only)* | use_sniffer | Includes the CPU interface which allows direct DRAM access from software. Including this may introduce timing issues at very high DRAM controller frequencies. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| rst | in | boolean | Resets the block when pulsed high |
| address | in | UFix_32_0 | A signal which accepts the address. See below for details. |
| data_in | in | 144 or 288 bit unsigned | Accepts data to be saved to DRAM. |
| wr_be | in | UFix_18_0 or UFix_36_0 | Selects bytes for writing (write byte enable). It is normally 18 bits wide for a 144 bit data bus, but if 288 bit data bus is selected, this becomes a 36 bit variable. |
| RWn | in | boolean | Selects read or not-write. `1` for read, `0` for write. |
| cmd_tag | in | UFix_32_0 | Accepts a user-defined tag for labelling entered commands. *Not implemented on ROACH.* |
| cmd_valid | in | boolean | Clocks data into the command buffer. |
| rd_ack | out | boolean | Used to acknowledge that the last `data_out` value has been read. |
| cmd_ack | out | boolean | Acknowledges that the last command was accepted (when buffer is full, will not accept additional commands). ROACH: Pin HI unless an attempt to clock in a command failed |
| data_out | out | UFix_144_0 | Outputs data from DRAM, 144 bits at a time. Reads are in groups of 288 bits (ie, 2 clocks). |
| rd_tag | out | UFix_32_0 | Outputs the identifier for the data on `data_out` (as submitted on `cmd_tag` when the command was issued). *Not implemented on ROACH.* |
| rd_valid | out | boolean | Indicates that the data on `data_out` is valid. |

### Description

### BEE2 Specific Info

Core details about the BEE2 memory interface can be found at the (static) BEE2 wiki:

http://bee2.eecs.berkeley.edu/wiki/Bee2Memory.html

### Addressing

The 1GB storage DIMMs have 18 512Mbit chips each. They are arranged as 64Mbit x 8 (bus width) x 9 (chips per side/rank) x 2 (sides/ranks). Two ranks (sides) per module with the 9 memory ICs connected in parallel, each holding 8 bits of the data bus width (72 bits). Each IC has four banks, with 13 bits of row addressing and 10 bits for column addressing. Normally, each address would hold 64 bits + parity (8 bits), however, the BEE2 uses the parity space as additional data storage giving a capacity of 1.125 GB per DIMM module.

From Micron's datasheet on the *MT47H64M8CD-37E* (as used by CASPER in its Crucial 1GB *CT12872AA53E* modules): The double data rate architecture is essentially a 4n-prefetch architecture, with an interface designed to transfer two data words per clock cycle at the I/O balls. A single read or write access effectively consists of a single 4n-bit-wide, one-clock-cycle data transfer at the internal DRAM core and four corresponding n-bit-wide, one-half-clock-cycle data transfers at the I/O balls.

Reads and writes must thus occur four-at-a-time. 4 x 72bits = 288 bits. Although the mapping of the logical to physical addressing is abstracted from the user, it is useful to know how the DRAM block's address bus is derived, as it impacts performance:

| Addressing | Assignment |
|---|---|
| Column | $12 \longrightarrow 3$ |
| Rank | 13 |
| Row | $27 \longrightarrow 14$ |
| Bank | $29 \longrightarrow 28$ |
| not used | $31 \longrightarrow 30$ |

Address bit assignments

Each group of 8 addresses selects a 144 bit logical location (the lowest 3 bits are ignored). For example, address `0x00` through `0x7` all address the same 144 bit location. To address consecutive locations, increment the address port by eight. There are thus a total of $2^{27}$ possible addresses. The block supports 2GB DIMMs (UNCONFIRMED) since 14 bits of addressing are reserved for row selection. The 1GB DIMMs using Micron 512Mb chips, however, only use 13 bits for row selection which results in $2^{26}$ possible address locations. Care should be taken when addressing the 1GB DIMMS as bit 27 of the address range is not valid. However, bits 28 and 29 are mapped. Since bit 27 is ignored, it results in overlapping memory spaces.

### Data bus width

The BEE2 uses ECC DRAM, however, the parity bits are used for data storage rather than parity storage. Thus, the data bus is 72 bits wide instead of the usual 64 bits.

The memory module has a DDR interface requiring two reads or writes per RAM clock cycle (~200MHz), thus requiring the user to provide 144 bits per clock cycle. Furthermore, as outlined above, data has to be captured in batches of 288 bits. This can be done in one of two ways: in two consecutive blocks of 144 bits, or over a single 288 bit-wide bus. This is selectable as a mask parameter. If half-burst is selected, only a 144 bit input is required. 288 bits are still written to DRAM, but the second 144 bits are not specified. Thus, half of the DRAM capacity is unusable.

### ROACH Specific Info

The ROACH DRAM infrastructure currently doesn't support half burst and wide data modes. Bank management is always enabled. Tag buffers are not implemented. The DRAM controller clock rate can be one of the following: 150, 200, 266, 300 or 333. If a frequency other than these is provided the default of 266 will be used. The dram controller has been known to work at 300MHz.

### Interfacing Details

To write data into the DRAM, 'RWn' is held low, 'cmd_valid' is held high for a minimum of two FPGA clocks, and the 'address' port is held constant for both clock cycles. For example, to write into addresses 0x00 and 0x01, keep the address at 0x00 for both clocks. To read data out of the DRAM, hold 'RWn' high, keep the address constant for two FPGA clock cycles, and toggle the 'cmd_valid' pin every clock. Note that a new word will be available on the 'data_out' pin on every clock cycle. 'rd_valid' will frame valid output data some indeterminate number of clock cycles after the read 'cmd_valid' toggles. 'cmd_ack' is high unless an attempt to write a command into the input FIFO failed, at which point it will go low synchronously with the issuing of the failed command.

Many ROACHs have been shipped with 1 GB dual rank DIMMs by default. The current DRAM controller is not able to handle multiple ranks, so when a dual-rank DIMM is installed on the board, only half the memory is available. In order to use the full 1 GB, a single rank DIMM is needed, or in principle a dual rank 2 GB module.

Note that on the ROACH all of the oddities of the DRAM addressing specified above for the BEE2 version are taken care of for you, so you can just directly address locations 0 to (2^30 / 16) = 2^26 in the hardware.

### DRAM CPU interface

If the block mask was set to include the CPU interface, the DRAM can be accessed by bytes through BORPH through 'dram_memory'. The width of the CPU interface is only 128 bits (16 bytes), which results in discrepancy between hardware and CPU address. After every 64 bits, there are 8 ECC bits not visible to the CPU. For example bytes 0x00-0x07 in the DRAM are seen as 0x00-0x07 in the CPU, byte 0x08 in the DRAM is not visible to the CPU, and byte 0x09 in the DRAM is seen as byte 0x08 in the CPU.

Only 64MB of DRAM can be mapped into the 'dram_memory' register at any given time. You can select which 64MB segment is mapped into the 'dram_memory' register though the first 32-bit word of the 'dram_controller' register. For example, to access the first 64MB chunk of DRAM write 0x0 into this register and for the second 0x1.

The DRAM is most easily accessed using the KATCP function "read_dram".

The second 32-bit word in the 'dram_controller' register indicates the DRAM controller ready flag. This value stores will be 0x1 if the controller is operational. If it is not your DRAM will not operate at all. Typical problems causing this would include using an unsupported RDIMM.

### Example models

1) David George's million channel ROACH spectrometer ("buf" block): rmspec.mdl

2) Laura Spitler's simple design that reads and writes a counter into the DRAM: Dram roach rwramp.mdl

3) Jason Manley's DRAM counter example: Dram counter test 10 1.gz

4) Tim Madden's DRAM streaming output design (April 2015) https://github.com/argonnexraydetector/RoachFirmPy

### Performance Tips

The performance of the DRAM block is dependent on the relative location of the addressed data and whether or not the mode (read/write) is changed. For example, consecutive column addresses can be written without delay, however, changing rows or banks incur delay penalties. See above for the address bit assignment.

To obtain optimum performance, it is recommended that the least significant bits be changed first (ie address the memory from `0x0000000` through to address `0x20000000` on the BEE2). This will increment column addresses first, followed by rank change, both of which incur little delay. Changing rows or banks can take twice as long. Further information can be found in the DRAM module's datasheet (Micron *MT47H64M8* on the BEE2).

Changing the mode(read/write) results in large delays, so it is recommended that read and writes be done in bursts into consecutive addresses. For a fabric clock speed of 200 MHz and DRAM speed of 266 MHz, a burst length of at least 32 words is recommended.

Bank management allows for three banks to be open simultaneously, reducing the overhead when switching between these banks. This feature is always enabled on ROACH, but YMMV with the BEE2 controller.

### Bi-directional GPIO

**Block:** Bi-directional GPIO (`gpio_bidir`)
**Block Author**: Brian Bradford
**Document Author**: Brian Bradford

### Contents

### Summary

The gpio_bidir block provides an Bi-diectional GPIO interface.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| I/O group | io_group | Selects the GPIO header on the board. |
| Data bitwidth | bitwidth | Specifies data bitwidth. |
| GPIO bit index | bit_index | Specifies the pin on the selected GPIO header. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| din | in | inherited | Data input (size set by Data bitwidth parameter in bits) |
| in_not_out | in | inherited | The control signal, 1 for input and 0 for output. |
| dout | out | inherited | The data output. |

### Notes

When using bitwidths greater than one, you should specify a vector of bit indices to use. GPIO bit index should have the same number of elements as the I/O bitwidth.

Example: If you set Data bitwidth to 4, you might want you use GPIO bit indices [0, 1, 2, 3].

## GPIO

**Block:** GPIO (`gpio`)
**Block Author**: Pierre-Yves Droz
**Document Author**: Billy Mallard

## Contents

## Summary

The gpio block provides access to GPIO pins on any board that has GPIO headers.

## Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| I/O group | io_group | Selects the board architecture and the GPIO header on that board. |
| I/O direction | io_dir | Chooses whether the pin sends data `out` of or `in` to the FPGA. |
| Data Type | arith_type | Specifies data type of register. |
| Data bitwidth | bitwidth | Specifies data bitwidth. |
| Data binary point | bin_pt | Specifies the binary point position of data. |
| GPIO bit index | bit_index | Specifies the pin on the selected GPIO header. |
| Sample period | sample_period | Specifies sample period of interface. |
| Use DDR | use_ddr | ??? |
| Pack register in the pad | reg_iob | ??? |
| Register clock phase | reg_clk_phase | `0`, `90`, `180`, `270`. |
| Termination method | termination | `None`, `Pullup`, `Pulldown`. |

## Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| reg_out | in | inherited | Output from design to GPIO pin. Only in I/O `in` mode. |
| sim_out | out | double | Simulation output of pin value. Only in I/O `in` mode. |
| sim_in | in | double | Simulation input of pin value. Only in I/O `out` mode. |
| reg_in | out | inherited | Input from GPIO pin to design. Only in I/O `out` mode. |

## Description

### ROACH Specific Information

*Main article:* ROACH FPGA Interfaces

Each GPIO header has one direction selector that controls the direction of all 8 of its pins. You must set a direction. So, in addition to your normal `gpio` block (configured for I/O group `ROACH:gpio[ab]`), add another `gpio` block, and set the following parameters:

- I/O group = `ROACH:gpio[ab]_oe_n`
- I/O direction = `out`
- bitwidth = `1`
- bit_index = `0`

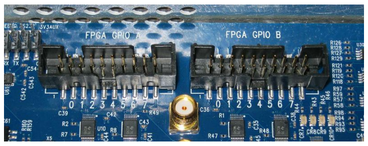
The direction selector takes a boolean value as its input, so wire it to a Xilinx constant block:

- `0` - output from ROACH
- `1` - input to ROACH

The two SMA connectors on the back of the board are wired directly to GPIO pins. Specifically, J11 and J10 are wired to pins 6 and 7 on GPIO A.

### Notes

The order of the FPGA GPIOs have been changed from "3 2 1 0 7 6 5 4" to now match the order printed on the PCB "0 1 2 3 4 5 6 7".



Old mapping: 3 2 1 0 7 6 5 4

New mapping: 0 1 2 3 4 5 6 7

## QDR

**Block:** QDR (`qdr`)
**Block Author**: David George
**Document Author**: David George

### Contents

### Summary

This block interfaces to the QDR SRAM devices on ROACH boards. Commands are executed at the rate they are applied, with synchronous and fixed timing. Data is always presented 10 cycles after a read is issued. Read and write data ports have 100% duty cycles.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| QDR Chip | which_qdr | Selects which physical QDR device to use (Two on ROACH V1). |
| Simulation QDR Address Width | qdr_awidth | Specifies the width of the address bus of the simulation model. (Limited to 18) |
| Use CPU Interface | use_cpu | Specify whether or not to include the QDR CPU interface, the removal of which may improve timing performance. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| rd_en | in | boolean | Asserting this signal issues a read command. See below for details on issuing commands. |
| wr_en | in | boolean | Asserting this signal issues a write command. See below for details on issuing commands. |
| be | in | UFix_4 | Selects bytes for writing (write byte enable). See below for behaviour relating to bursting and ECC bits. |
| address | in | UFix_32 | Signal used as the QDR address. See below for behaviour relating to addressing. |
| wr_data | out | UFix_36 | The data to written into the QDR. Bits 35:32 are ECC bits and are cleared when the processor writes to the corresponding byte. See below for behaviour relating to bursting and ECC bits. |
| data_valid | out | boolean | An active high signal indicating that the read data is valid. |
| ack | out | boolean | A signal indicating that the CPU interface is not accessing the memory. |
| phy_ready | out | boolean | A signal indicating that the QDR PHY has completed calibration, which takes very roughly 100us. |
| cal_fail | out | boolean | A signal indicating that the PHY calibration has failed. |

### Description

This document is a draft and requires verification.

### Issuing Commands

There are two types of commands: reads and writes. They are issued by the rd_en and wr_en signals respectively. The QDR address is presented on the same cycle that the command is issues. One type of command cannot be issued in two consecutive cycles. When this happens, the second issue will be ignored. This is due to QDR supporting bursts to reduce data rates on the control signals. Further, if a read and write are issued at the same time the write will be ignored. However, if the previous command was a valid read, the current read will be ignored and the write will take preference.

#### Bursting

When issuing reads and writes, data is presented on the respective data ports for two cycles. When issuing a write command, the `data_in` and `be` ports must be set for both the issuing cycle and the following cycle. During a read response, data is issued on the same cycle that the data_valid is asserted and on the following cycle.

#### Addressing

The address presented when a command is issued addresses a full burst worth of memory i.e. 72 bits of data.

#### ECC Bits

In hardware the QDR word is composed of four 9 bit components which are masked by the byte_enable signal. Each of these component include 8 data bits and a single ECC bit. This clashes with the byte-enable on the processor, which mask only 8 bits. For this reason the ECC bit gets cleared when the CPU writes to a byte of QDR memory. With this yellow block, the QDR data_in and data_out ports have the ECC bits on lines 35:32. This allows the four processor bytes to cleanly map to bits 31:0 of the data_in and data_out ports. This leads to a side-effect in the byte-enable behaviour as follows: be[0] masks data_in bits [7:0] and [32], be[1] mask data_in bits[15:8] and [33] etcetera.

### Snapshot

**Block:** Snapshot (`snapshot`)
**Block Author**: Andrew Martens
**Document Author**: Andrew Martens

#### Contents

#### Summary

The snapshot block is configurable block for capturing blocks of data with a standard interface supporting generic software drivers. It combines functionality from (and deprecates) the snap, sc, snap_64 and snap_circ blocks.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Storage medium | storage | Specifies whether to store the data in BRAM or DRAM |
| DRAM dimm | dram_dimm | Specifies which dimm to use if using DRAM as storage medium. |
| DRAM clock rate | dram_clock | Specifies the DRAM controller clock rate if using DRAM as a storage medium. |
| No. of Samples ($2^?$) | nsamples | Specifies the maximum depth of the data storage buffer |
| Data width | data_width | The bit width of the input data |
| Start delay support | offset | Option to support inserting a programmable number of samples between the trigger for the start of capture, and data capture itself. |
| Circular capture support | circap | Option to support continual capture until a signal to stop is received. |
| Extra value capture support | value | Option to support the capture of a value to a register as the first data item is captured. |
| Use DSP48s to implement counters | use_dsp48 | Option to use DSP48s to implement various internal counters to save logic. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| din | IN | unsigned_8_0 OR unsigned_16_0 OR unsigned_32_0 OR unsigned_64_0 OR unsigned_128_0 | The data to be captured. Data is stored with oldest data in the lowest addresses and in little endian format. |
| we | IN | boolean | After a trigger is begun, enables a write to the data buffer. |
| trig | IN | boolean | When high, triggers the beginning of a data capture. Thereafter, every enabled data is written to the data buffer. If offset capture is enabled, capture can be delayed by a configurable number of samples after the trigger. If circular capture is enabled, capture continues until the input to the stop port goes high. |
| stop | IN | boolean | Triggers the end of data capture when in circular capture mode. |
| vin | IN | unsigned_32_0 | When extra value capture is enabled the value on this port is captured on the same sample as the first data item captured. |
| ready | OUT | boolean | When using DRAM as a storage buffer, signals that the DRAM controller has finished calibration and is ready to receive data. |

### Software interface

| Name | Dir | Data Type |
|---|---|---|
| ctrl | Write | unsigned_32_0 |
| trig_offset | Write | unsigned_32_0 |
| val | Read | unsigned_32_0 |
| status | Read | unsigned_32_0 |
| tr_en_cnt | Read | unsigned_32_0 |
| bram | Read | unsigned_32_0 |
| dram | Read | unsigned_32_0 |

### Description

### Usage

Under TinySH/BORPH, this device will have 3 sub-devices: `ctrl`, `bram`, and `addr`. `ctrl` is an input register. Bit 0, when driven from low to high, enables a trigger/data capture to occur. Bit 1, when high, overrides `trig` to trigger instantly. Bit 2, when high, overrides `we` to always write data to bram. `addr` is an output register and records the last address of `bram` to which data was written. `bram` is a 32 bit wide Shared BRAM of the depth specified in `Parameters`.

## Snapshot Capture

**Block:** Snapshot Capture (`snap`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons, Ben Blackman

### Contents

### Summary

The snap block provides a packaged solution for capturing data from the FPGA fabric and making it accessible from the CPU. Snap captures to a 32 bit wide shared BRAM.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| No. of Samples ($2^?$) | nsamples | Specifies the depth of the Shared BRAM(s); i.e. the number of 32bit samples which are stored per capture. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| din | IN | unsigned_32_0 | The data to be captured. Regardless of type, the bit-level representation of these numbers are written as 32bit values to the Shared BRAM. |
| trig | IN | boolean | When high, triggers the beginning of a data capture. Thereafter, every enabled data is written to the shared BRAM until it is full. |
| we | IN | boolean | After a trigger is begun, enables a write to Shared BRAM. |

### Description

#### Usage

Under TinySH/BORPH, this device will have 3 sub-devices: `ctrl`, `bram`, and `addr`. `ctrl` is an input register. Bit 0, when driven from low to high, enables a trigger/data capture to occur. Bit 1, when high, overrides `trig` to trigger instantly. Bit 2, when high, overrides `we` to always write data to bram. `addr` is an output register and records the last address of `bram` to which data was written. `bram` is a 32 bit wide Shared BRAM of the depth specified in `Parameters`.

### 64 Bit Snapshot

**Block:** 64 Bit Snapshot (`snap64`)
**Block Author**: Aaron Parsons
**Document Author**: Aaron Parsons, Ben Blackman

#### Contents

### Summary

The snap block provides a packaged solution for capturing data from the FPGA fabric and making it accessible from the CPU. Snap64 captures to 2x32 bit wide shared BRAMs to effect a 64 bit capture.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| No. of Samples ($2^?$) | nsamples | Specifies the depth of the Shared BRAM(s); i.e. the number of 64bit samples which are stored per capture. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| din | IN | unsigned_64_0 | The data to be captured. Regardless of type, the bit-level representation of these numbers are written as 64bit values to the Shared BRAMs. |
| trig | IN | boolean | When high, triggers the beginning of a data capture. Thereafter, every enabled data is written to the shared BRAM until it is full. |
| we | IN | boolean | After a trigger is begun, enables a write to Shared BRAM. |

### Description

### Usage

Under TinySH/BORPH, this device will have 3 sub-devices: `ctrl`, `bram_msb`, `bram_lsb`, and `addr`. `ctrl` is an input register. Bit 0, when driven from low to high, enables a trigger/data capture to occur. Bit 1, when high, overrides `trig` to trigger instantly. Bit 2, when high, overrides `we` to always write data to bram. `addr` is an output register and records the last address of bram to which data was written. `bram_msb` and `bram_lsb` are 32 bit wide Shared BRAMs of the depth specified in `Parameters`. `bram_msb` holds the upper 32 bits of `din` while `bram_lsb` holds the lower 32 bits of `din`.

### Software Register

**Block:** Software Register (`software register`)
**Block Author**: Pierre-Yves Droz
**Document Author**: Henry Chen

### Contents

### Summary

Inserts a unidirectional 32-bit register shared between the FPGA design and the PowerPC bus.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| I/O direction | io_dir | Chooses whether register writes `To Processor` or reads `From Processor`. |
| Data Type | arith_type | Specifies data type of register. |
| Data bitwidth | bitwidth | Specifies data bitwidth. Hard-coded at 32 bits. |
| Data binary point | bin_pt | Specifies the binary point position of data. |
| Sample period | sam-ple_period | Specifies sample period of interface. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| reg_out | in | inherited | Output from design to processor bus. Only in `To Processor` mode. |
| sim_out | out | double | Simulation output of register value. Only in `To Processor` mode. |
| sim_in | in | double | Simulation input of register value. Only in `From Processor` mode. |
| reg_in | out | inherited | Input from processor bus to design. Only in `From Processor` mode. |

### Description

A software register is a `shared` interface, meaning that it is attached to both the FPGA fabric of the System Generator design as well as the PowerPC bus. The registers are unidirectional; the user must choose at design-time whether the register is in `To Processor` mode (written by the FPGA fabric and read by the PowerPC) or in `From Processor` mode (written by the PowerPC and read by the FPGA fabric).

The bitwidth is fixed at 32 bits, as it is attached to a 32-bit bus, but the Simulink interpretation of the data type and binary point is controllable by the user. The data type and binary point parameters entered into the mask are enforced by the block; the block will cast to the specified data type and binary point going in both directions.

## SRAM

**Block:** SRAM (`sram`)
**Block Author**: Pierre Yves Droz, Henry Chen
**Document Author**: Ben Blackman

### Contents

### Summary

The sram block represents a 36x512k SRAM chip on the IBOB. It stores 36-bit words and requires 19 bits to access its address space.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| SRAM | sram | Selects which SRAM chip this block represents. |
| Data Type | arith_type | Type to which the data is cast on both the input and output. |
| Data binary point (bitwidth is 36) | bin_pt | Position of the binary point of the data. |
| Sample period | sample_period | Sets the period with reference to the clock frequency. |
| Simulate SRAM using ModelSim | use_sim | Turns ModelSim simulation on or off. |

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| we | IN | boolean | A signal that when high, causes the data on data_in to be written to address. |
| be | IN | unsigned_4_0 | A signal that enables different 9-bit bytes of data_in to be written. |
| address | IN | unsigned_19_0 | A signal that specifies the address where either data_in is to be stored or from where data_out is to be read. |
| data_in | IN | arith_type_36 | A signal that contains the data to be stored. |
| data_out | OUT | arith_type_36 | A signal that contains the data coming out of address. |
| data_valid | OUT | boolean | A signal that is high when data_out is valid. |

### Description

### Usage

The SRAM block is 36x512k, signifying that its input and output are 36-bit words and it can store 512k words. Each clock cyle, if `we` is high, then each bit of be determines whether each 9-bit chunk will be written to address. `be` is 4 bits with the highest bit corresponding to the most significant chunk (so if `be` is 1100, only the top 18 bits will be written). If `we` is low, then the SRAM block ignores `data_in` and be and reads the word stored at `address`.

## XSG Core Config

**Block:** XSG Core Config (`XSG core config`)
**Block Author**: Pierre-Yves Droz
**Document Author**: Henry Chen

### Contents

### Summary

The XSG Core Config block is used to configure the System Generator design for the `bee_xps` toolflow. Settings here are used to configure the Xilinx System Generator block parameters automatically, and control toolflow script execution. It needs to be at the top level of all designs being compiled with the `bee_xps` toolflow.

### Mask Parameters

| Parameter | Variable | Description |
|---|---|---|
| Hardware Platform | hw_sys | Selects the board/chip to compile for. |
| Include Linux add-on board support | ibob_linux | Includes BORPH-capable Linux for IBOB. |
| User IP Clock source | clk_src | Selects the clock on which to run the System Generator circuit. |
| GPIO Clock Pin I/O group | gpio_clk_io_group | Selects GPIO type to use as clock input if using user clock on an IBOB. |
| GPIO Clock Pin bit index | gpio_clk_bit_index | Selects GPIO pin to use as clock input if using user clock on an IBOB. |
| User IP Clock rate (MHz) | clk_rate | Generates timing constraints for the design. |
| Sample Period | sample_period | Sample period for Simulink simulations. |
| Synthesis Tool | synthesis_tool | Selects the tool to use for synthesizing the design’s netlist. |

### Ports

None.

### Description

The function of the XSG Core Config block is to set parameters for the toolflow scripts. It supercedes the use of the Xilinx System Generator block and has supplemental options for board-level parameters. Although a System Generator block is still needed in all designs, the XSG Core Config block automatically changes the System Generator block settings based on its own parameters.

The settings in the XSG Core Config block are used to determine the system-level conditions of the SysGen design. It sets which of the toolflow-supported boards the design is being compiled for, from which it determines what FPGA to target, as well as clocking options like clock source and timing constraints. The Sample Period and Synthesis Tool parameters are included in the block so that all system-level options available in the System Generator block could be handled by this single block.

## Gaussian Random Number Generator

**Block:** Gaussian Random Number Generator (`Gaussian Random Number Generator`)
**Block Author**: Kaushal D. Buch
**Document Author**: Kaushal D. Buch

### Contents

### Summary

This is a low foot-print Gaussian noise source for testing CASPER based hardware designs during development. It contains a pair of uncorrelated noise and each noise data is available as four parallel output streams, which are directly compatible to the iADC outputs.

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| reset | IN | boolean | Reset signal to initialize the noise sources. Reset is synchronous active high signal. |
| Noise_Out11 to Noise_Out14 | OUT | 8-bit signed | Four uncorrelated streams of noise samples. |
| Noise_Out21 to Noise_Out24 | OUT | 8-bit signed | Four uncorrelated streams of noise samples. These are also uncorrelated with respect to Noise_Out11 to Noise_Out14 signals. |

### Description

The noise source blocks do not use any specific resources like BRAMs, multipliers etc. The Kurtosis value of these sources is around 2.9 to 2.93. Both of these sources can be used as individual noise source or collectively as uncorrelated noise sources. The resource utilization is around 2 to 3 % for a single noise source on Virtex-5 SX95 FPGA.

Note: The seeds of the individual noise sources can be changed. But the seeds need to be mutually uncorrelated for a particular source i.e. considering only one source say Noise Source -1 ,all the fourteen seeds within that source have to mutually uncorrelated.

### Test Results

Test was carried out by integrating these uncorrelated sources with a pocket correlator design running at 600MHz ADC clock and having 1 second integration time, implemented on ROACH. The results show a flat spectrum across all the FFT channels and a normalized cross-correlation of about 0.001.

### Correlation Control Block

**Block:** Correlation Control Block (`CCB`)
**Block Author**: Kaushal D. Buch, GMRT, India
**Document Author**: Kaushal D. Buch, GMRT, India

#### Contents

### Summary

Correlation control block takes a pair of uncorrelated digital noise sources in the input and generates a pair of output noise with correlation. The amount of correlation can be selected from a set of pre-defined values. This block is an extension to the Gaussian Random Number block in the CASPER library.

### Ports

| Port | Dir | Data Type | Description |
|---|---|---|---|
| noise_in1 to noise_in4 | IN | 8-bit signed | Four uncorrelated streams from the output of the Gaussian Random Number Generator. |
| noise_in5 to noise_in8 | IN | 8-bit signed | Four uncorrelated streams from the output of the Gaussian Random Number Generator. |
| corr_sel_in | IN | 3-bit un-signed | Selection of the amount of correlation coefficient at the output. :: Selection - 0 - uncorrelated (~0%) 1 - 5% correlation 2 - 10% correlation 3 - 20% correlation 4 - 50% correlation 5 - 100% correlation |
| corr_noise_out1 to corr_noise_out4 | OUT | 8-bit signed (Fix8_7) | Four streams of output digital noise. |
| corr_noise_out5 to corr_noise_out8 | OUT | 8-bit signed (Fix8_7) | Four streams of output digital noise. |

### Description

Correlation Control Block (CCB) is an extension to the existing library block called Gaussian Random Number Generator (GRNG). CCB can be used along with GRNG block to get variable correlation between two input noise channels.

The correlation control block uses an uncorrelated noise source whose coupling to the two input channels is varied to control the correlation. By varying the ratio of the variance of common noise source (Pc) to the variance of input channels (P1 and P2) we get the correlation coefficient as Pc / (P+Pc) (Note: We assume that P1 = P2 = P, i.e. components from input channels have same variance).

Currently, there is a facility to select the following values of correlation through software register - 0% (uncorrelated), 5%, 10%, 20%, 50% and 100% (correlated).

### Test Results

The variable correlation digital noise source design was tested with the GRNG for a 300MHz BW PoCo with 0.89s integration on ROACH.

## 4.2.2 The CASPER Toolflow

The current Python-based CASPER toolflow (sometimes called *jasper*, to distinguish it from the MATLAB-based flow which supported ROACH2 and earlier platforms) was designed to enable users to quickly and effectively turn high-level DSP designs into FPGA bitcode, without having to worry about low-level implementation details.

While the ultimate goal of the toolflow is to support (or at least not prohibit) a wide range of mechanisms to input DSP designs, and be agnostic about the target FPGA's vendor, in practice the toolflow pipeline is as follows:

- A user specifies a DSP design using MATLAB's graphical programming tool, `Simulink`.
- I/O to/from the DSP pipelines are specified using CASPER's `xps_library` Simulink blocks.
- DSP functionality is specified using CASPER's `casper_library` Simulink blocks.
- A user invokes the CASPER toolflow, which generates a Vivado project, complete with appropriate infrastructure and constraints to support the user's I/O requirements.
- Vivado compiles the toolflow-supplied project into a bitstream.
- The toolflow appends meta-data to this bitstream, describing and run-time, software-accessible, components in the firmware design.
- The bitstream and meta-data are delivered to the user in the form of a `.fpg` file.
- This file is programmed on to a compatible board using the `casperfpga` Python library.
- `casperfpga` provides Python methods for interacting with a running board (configuring registers, etc.)

### Goals of the CASPER Toolflow

The ultimate goal of the tooflow is simple: make FPGA programming easy, so researchers (largely radio astronomers) can quickly develop and deploy instrumentation to meet their scientific requirements. More concretely, the CASPER toolflow aims to:

- Make complex interfaces (1/10/40/100 Gb Ethernet, SRAM, DRAM, ADCs, DACs, etc.) available as sources and sinks in a DSP design without requiring knowledge of their low-level implementation.
- Facilitate easy porting of designs between FPGA platforms, by abstraction of these underlying interfaces.
- Provide a straight-forward integration with control and monitoring software.

While the *jasper* toolflow described in this documentation is (relatively) new, it builds on CASPER's original MATLAB-based flow, which was designed over a decade ago, and has quietly powered many scientific instruments over its lifetime.

### Toolflow Terminology

When discussing the CASPER toolflow, it helps to first define some terminology commonly used in the collaboration.

**Frontend** : The part of the CASPER software stack used for design-entry. In practice, this means *Simulink*, though there are perpetual aspirations to support other entry methods, particularly non-graphical ones.

**Backend** : The vendor-specific part of the CASPER software stack used to compile a toolflow-generated project. In practice, this means *Xilinx Vivado*, but could also in principle be *ISE* or *Altera Quartus*.

**Middleware / Toolflow-proper** : The Python-based part of the CASPER software stack used to turn the design information entered into the *Frontend* into a complete specification for a *Backend* compile.

**Yellow Blocks** : Interface blocks (ADCs/DACs/Ethernet/Memory) which are used by the *Frontend* to give the user access to a platform's peripherals. Yellow Blocks in the Frontend are supported by Python classes in the toolflow middleware. These classes ultimately determine what source files, constraints, and HDL incantations are required to instantiate a *Yellow Block* in hardware.

**Software Register** : A *Software Register* is a yellow block encapsulating a 32-bit register in a design which can be accessed at runtime, usually via the `casperfpga` FPGA control library. These registers are usually used to feed control signals (MUX switch controls / enable signals / etc.) into a design.

**Platform YAML** : A YAML file defining the physical properties of an FPGA hardware platform.

**jasper.per** : A YAML file output by the toolflow *Frontend* which specifies what was designed by the user. In principle, any tool which can generate such a file is a viable toolflow Frontend candidate.

**castro** : A well-meaning attempt to provide an abstract, well-defined, platform-independent interface between the tooflow *Middleware* and the platform-/vendor-specific *Backend*. This interface is a Python `Castro` object, dumped as a YAML file! In practice, developers have subverted the original definitions of the `Castro` class to make it easier to get platform-specific data between stages of the toolflow, and somewhat undermined its purpose.

## Parts of the Toolflow

There are several components to the toolflow which are key to its functionality. All toolflow related files (except those used by the *Frontend*) are found in the `jasper_library` subdirectory of the toolflow repository.

Some documentation of the Pythonic parts of the toolflow can be found in the auto-generated docs. Here we focus on giving an overview of the purpose of different parts of the toolflow, as well as providing some (hopefully) instructive examples of their usage.

## Peripherals file

The `jasper.per` peripherals file is output by the toolflow *Frontend*. As an example, a design for the SNAP board, which contains two software registers – one called `a`, which feeds a value into the DSP pipeline, and one called `b`, which reads a value out of the DSP pipeline – is shown below:

```yaml
yellow_blocks:
  snap_tut_intro/SNAP1:
    name: SNAP1
    fullpath: snap_tut_intro/SNAP1
    tag: xps:xsg
    hw_sys: 'SNAP:xc7k160t'
    clk_src: sys_clk
    clk_rate: 100
    use_microblaze: off
    sample_period: 1
    synthesis_tool: XST
  snap_tut_intro/a:
    name: a
    fullpath: snap_tut_intro/a
    tag: xps:sw_reg
    io_dir: From Processor
    io_delay: 0
    sample_period: 1
    names: reg
    bitwidths: 32
    bin_pts: 0
    arith_types: 0
    sim_port: on
    show_format: off
  snap_tut_intro/b:
    name: b
    fullpath: snap_tut_intro/b
    tag: xps:sw_reg
    io_dir: To Processor
    io_delay: 0
    sample_period: 1
    names: reg
```

(continues on next page)

(continued from previous page)

```
    bitwidths: 32
    bin_pts: 0
    arith_types: 0
    sim_port: on
    show_format: off
user_modules:
  snap_tut_intro:
    clock: clk
    ports:
      - snap_tut_intro_a_user_data_out
      - snap_tut_intro_b_user_data_in
    sources:
      - /foo/snap_tut_intro/sysgen/hdl_netlist/snap_tut_intro.srcs/sources_1/imports/
↪sysgen
      - /foo/snap_tut_intro/sysgen/hdl_netlist/snap_tut_intro.srcs/sources_1/ip/*.coe
      - /foo/snap_tut_intro/sysgen/hdl_netlist/snap_tut_intro.srcs/sources_1/ip/*/*.
↪xci
```

This YAML structure has two keys.

- `yellow_blocks` is a hierarchical dictionary of all yellow blocks in the user's design, along with their parameters. The `tag` field of each entry in this dictionary indicates what type of block this is. The `yellow_blocks` dictionary always contains a block with the tag `xps:xsg`, which is a special yellow block which contains information of the platform the user is targeting. In this case, the target is a *SNAP* board.
- `user_modules` is a dictionary listing the module(s) in a design which have been compiled by the *Frontend*. For a Simulink design, this is everything in the user's model which is not a yellow block. Each module's dictionary should define the name of its clock port, other data ports, and it's source file locations. In the case of a Simulink design, the latter are the outputs of the Xilinx System Generator compiler.

### Platforms

`jasper_library/platforms/<platform>.yaml` contains YAML files describing the physical properties of a hardware platform. In order for a board to be supported by the CASPER toolflow it must have a Platform YAML file present in this directory. The name of this file should match the name of the compile platform, as understood by the toolflow *Frontend*. For example, the head of the YAML file `snap.yaml` which defines the SNAP platform is shown below.

```
name: snap
manufacturer: Xilinx
fpga: xc7k160tffg676-2
backend_target: vivado
sources: []
constraints: []
provides:
  - sfp0
  - sfp1
  - zdok0
  - HAD1511_0
  - HAD1511_1
  - HAD1511_2
  - lmx2581
pins:
  sys_clk_p:
    loc: E10
```

(continues on next page)

(continued from previous page)

```
    iostd: LVDS_25
  sys_clk_n:
    loc: D10
    iostd: LVDS_25
  led:
    iostd: LVCMOS25
    loc:
      - C13
      - C14
      - D13
      - D14
      - E12
      - E13
  zdok0:
    iostd: LVCMOS25
    loc:
      - AA23
      - AB24
      - Y25
      - Y26
      - U24
      - U25
      - lots more pins....
```

The required fields in this YAML file are:

- **name**: The name of the platform, as understood by the tooflow (i.e. matching the name in the `hw_sys` parameter of the yellow block tagged `xps:xsg`). This should match the name of the file `<name>.yaml`.
- **manufacturer**: A string indicating the platform's FPGA manufacturer. This can be used to make implementation decicions as the toolflow builds a user's design. Currently, only `Xilinx` is supported.
- **fpga**: The FPGA model the platform uses. This should be a string in the form understood by the FPGA vendor's compile tools. For example, for Xilinx platforms, it should be the appropriate part for a `create_project ... -part <fpga>` `tcl` command call.
- **backend_target**: A string indicating the *Backend* compile tool to use. Currently the only supported target is `vivado`, which corresponds to the `VivadoBackend` class. Once upon a time `ise` was also supported, as a proof-of-concept experiment in compiling for pre-7-series Xilinx FPGAs. This probably no longer works.
- **sources**: A list defining source files which should be included in compiles for this platform. Ideally this should be an empty list `[]`, since a more toolflow-friendly way of adding files is via the platform-specific MSSGE Yellow block.
- **constraints**: A list defining constraints files which should be included in compiles for this platform. Ideally this should be an empty list `[]`, since a more toolflow-friendly way of adding files is via the platform-specific MSSGE Yellow block.
- **provides**: A list of strings detailing the capabilities of the board. These are used for loose consistency checks when compiling, as they are matched against `requires` strings defined by `YellowBlock` instances. For example, a 10Gb Ethernet yellow block might *require* `sfp0` - if the platform does not *provide* this, the compile fill fail consistency checks.
- **pins**: The bulk of the platform file contains pin location and iostandard definitions, in a a dictionary of the form `{<pin name>: {loc: <loc>, iostd: <iostd>}`. Either `loc` or `iostd` entries may be lists. If they are both lists, their lengths should match. These friendly pin names are used by the toolflow to perform platform-agnostic definitions of external port connections.

### The `VerilogModule` Class

The `VerilogModule` class is a Pythonic encapsulation of a verilog module. It provides simple Python methods to build a module by adding ports, signals, assignment statements, and sub-module instantiations. Code generation methods allow the `VerilogModule` to output valid verilog describing itself.

The `VerilogModule` class also provides the ability to instantiate sub-modules with Wishbone interfaces, and will quietly manage their address spaces and arbitration.

While relatively simple, the `VerilogModule` class is central to the functionality of the toolflow, which is, ultimately, just a code-generator. Adding support for new hardware to the toolflow entails heavy use of the `VerilogModule` class by `YellowBlock` objects.

### Yellow Blocks

A yellow block is a *Frontend* module (so-called because in Simulink these modules are, literally, yellow blocks) whose instantiation in a user's design triggers the toolflow to insert some code into the generated code at compile time. A dedicated tutorial has been written to explain how to add new yellow blocks to the tooflow *Frontend* and *Middleware*.

In the case of the *Frontend*, adding a yellow block means putting a new block in the Simulink `xps_library` and creating a GUI (or *mask* in Simulink parlance) with which users can set the block's parameters. For example, for a 10Gb Ethernet yellow block, at a minimum the block mask should allow the user to pick the physical port on their board they with which then want the Ethernet block to be associated. The mask might also allow parameters to be configured, such as the MAC address of the underlying Ethernet core, or sizes of transmit/receive buffers.

In the case of the *Middleware*, adding a yellow block means creating a new `YellowBlock` subclass. This subclass defines how the presence of a yellow block in the *Frontend* impacts the constaints and verilog delivered to the *Backend* compiler.

Current `YellowBlock` subclasses can be found in the toolflow repository at `jasper_library/yellow_blocks` and alongside the yellow block tutorial serve as instructive examples. Unless a block is unusually simple, it probably will implement at least three methods:

- `initialize()`: Configure parameters of the yellow block, such as the source files it requires.
- `modify_top(top)`: Modify the`VerilogModule` instance, `top`, which describes the top-level of the user's design. Here you can make use of `VerilogModule` methods to instantiate sub-modules, and connect them to stuff.
- `gen_constraints()`: Return a list of constraint objects, defining external pin connections, clock rates, or other constraints the toolflow *Backend* knows how to deal with.

### How it all fits together

The toolflow is executed via the `exec_flow.py` script. A complete compile is invoked with

```
python exec_flow.py --perfile --frontend --middleware --backend --software -m <model_
↪name>
```

Where the final argument indicates the *Frontend* file (Simulink model) which should be used as the starting point of the compile. Each of the flags triggers a different stage of the toolflow compile process, and following these stages gives an idea of how the toolflow fits together.

### Peripheral file generation / Frontend compile

The `--perfile` flag causes the toolflow's *Frontend* to output a toolflow-standard `jasper.per` peripherals file, which contains information about all the yellow blocks in the design, and the locations of source files which the *Frontend* is responsible for compiling. The `--frontend` flag causes the frontend to compile any user IP it is responsible for synthesizing. In the case of the Simulink Frontend, this is essentially a call to *Xilinx System Generator*, triggering a compile of all the DSP blocks in the user's Simulink model.

These actions correspond to the toolflow methods:

```
ToolflowFrontend.gen_periph_file()
ToolflowFrontend.compile_user_ip()
```

In practice, since these methods ultimately invoke MATLAB calls, while the toolflow can call them via Python methods, usually they are run directly in MATLAB, via the command `jasper_frontend`. After this stages of the compile, a `jasper.per` file has been generated, which serves as the input to future compile stages. This file contains information about all the yellow blocks in the *Frontend* model, as well as the locations of any synthesized code which needs to be included in the final project.

### Middleware Project Building

The `--middleware` flag invokes the core toolflow methods which build an FPGA project. These methods are:

```
Toolflow.gen_periph_objs()
Toolflow.build_top()
Toolflow.generate_hdl()
Toolflow.generate_consts()
Toolflow.write_core_info()
Toolflow.write_core_jam_info()
Toolflow.constraints_rule_check()
Toolflow.dump_castro()
```

In chronological order:

1. `gen_periph_objs()` reads `jasper.per` to figure out which yellow blocks are in a user's design. It then constructs the associated `YellowBlock` objects, and calls their `initialize()` methods.
2. `build_top()` creates a `VerilogModule` instance to represent the top-level of the user's design in HDL.
3. `generate_hdl()` instantiates relevant yellow block code in this top-level module, by calling each `YellowBlock` sub-class's `modify_top` method. It also instantiates the user's DSP IP (i.e., the blocks compiled by *System Generator*. At the end of this method the fully populated `VerilogModule` instance is turned into a verilog source file, and added to the project the toolflow is constructing.
4. `generate_consts()` gathers the constraints required by each yellow block via their `gen_constraints` methods. Where applicable, symbolic contraints (such as the LOCs) are turned into physical constraints via the pin mappings in the relevant platform's YAML configuration file.
5. `write_core_info()` / `write_core_jam_info()` collects information about the runtime-accessible registers in the design, and writes them to file(s).
6. `constraints_rule_check()` checks for ports in the top-level verilog which are missing associated constraints.
7. `dump_castro()` dumps a description of the now complete project specification to disk, as a YAML dump of a `Castro` instance.

### Backend compiling

The `--backend` flag triggers instantiation of a `ToolflowBackend` object (in practice, this will invariably be a `VivadoBackend`. Two methods are run against this object:

```
ToolflowBackend.import_from_castro()
ToolflowBackend.compile()
```

The first of these, `import_from_castro`, reads the output of the toolflow *Middleware*, essentially copying attributes of the `Castro` objects to internal attributes. The second, `compile()`, takes the imported pythonic representation of the project and delivers a bitstream. In the `VivadoBackend` case, everything in the project is elaborated into a `tcl` script, which is then run by Vivado in batch mode.

### Software generation

When the *Backend* finishes compiling, it will (hopefully!) have generated a viable FPGA bitstream. All that remains is to append meta data to this bitstream, which will tell the `casperfpga` software library what registers are present in the bitstream. This concatenation of bitstream and metadata is generated by

```
ToolflowBackend.mkfpg()
```

Which delivers a custom-CASPER-format `.fpg` file.

Users can load this file onto a CASPER-supported FPGA platform with the `casperfpga` library:

```
import casperfpga
myfpga = casperfpga.CasperFpga(<hostname>)
myfpga.upload_to_ram_and_program(<fpgfile>.fpg)
# Read and write registers:
myfpga.registers.reg_a = 0xdeadbeef
#...etc.
```

### Supporting New Hardware

Here we briefly summarize the steps required to add support for a new hardware platform or peripheral to the toolflow.

### Adding a New Platform

Depending on the level of support required, adding a new hardware platform to the toolflow is actually quite straightforward.

### Adding a Platform to the Toolflow Frontend

First, a *Platform Block* for the new platform needs to be added to the `xps_library` Simulink blockset. This library can be found in the toolflow repository at `xps_library/xps_library.slx`. However, to aid version control, this library is automatically generated from the model files in `xps_library/xps_models`, each of which contains one library block.

Platform blocks live in `xps_library/xps_models/Platforms` – it is suggested that new platforms are added by copying one of these models to a new file, whose name reflects the new platform being added.

Once the new model file has been created, open it in Simulink, and modify the mask parameters of the block as appropriate. Probably, this means removing and clock sources which aren't valid for your platform, and/or hardcoding/parameterizing the allowed clock rates the user may enter.

Your platform yellow block should have an initialization function which looks for a *Xilinx System Generator* block in your design and configures it appropriately. For example, the SNAP-board initialization function is `xps_xsg_snap_conf_mask.m`, and contains:

```
if ~strcmp(bdroot, 'xps_library')
    sysgen_blk = find_system(gcs, 'SearchDepth', 1,'FollowLinks','on','LookUnderMasks
↪','all','Tag','genX');
    if length(sysgen_blk) == 1
        xsg_blk = sysgen_blk{1};
    else
        error('XPS block must be on the same level as the Xilinx SysGen block');
    end

    [hw_sys, hw_subsys] = xps_get_hw_plat(get_param(gcb,'hw_sys'));
    clk_src = get_param(gcb, 'clk_src');
    %clk_src = get_param(gcb, [hw_sys, '_clk_src']);
    syn_tool = get_param(gcb, 'synthesis_tool');

    %set_param(gcb, 'clk_src', clk_src);

    ngc_config.include_clockwrapper = 1;
    ngc_config.include_cf = 0;

    xlsetparam(xsg_blk,'xilinxfamily', 'Kintex7',...
        'part', hw_subsys,...
        'speed', '-2',...
        'testbench', 'off',...
        'package', 'ffg676');

    xlsetparam(xsg_blk,...
        'sysclk_period', num2str(1000/clk_rate),...
        'synthesis_language', 'VHDL');

    if strcmp(syn_tool, 'Leonardo Spectrum')
        xlsetparam(xsg_blk, 'synthesis_tool', 'Spectrum');
    else
        xlsetparam(xsg_blk, 'synthesis_tool', syn_tool)
    end

    xlsetparam(xsg_blk,'clock_loc','d7hack')
end
```

You should create your own initialization function in `xps_library/` and point your block to use it. At a minimum, it should appropriately set the `xilinxfamily`, `part`, `speed` and `package` entries of the Xilinx System Generator block via a command similar to:

```
xlsetparam(xsg_blk,'xilinxfamily', 'Kintex7',...
    'part', hw_subsys,...
    'speed', '-2',...
    'testbench', 'off',...
    'package', 'ffg676');
```

You may have to manually open a Xilinx System Generator block to figure out what the correct `xilinxfamily` specification is for your FPGA. If you wish, you can place design rule checking (eg. maximum / minimum allowed clock rates) within this initialization function.

Once you have added and saved your new platform block, you can include it in `xps_library.slx` by regenerating the library with the MATLAB command `xps_build_new_library`.

### Adding a Platform to the Toolflow Middleware

The new platform block needs to be backed up with support from the toolflow middleware.

First, a platform YAML file needs to be added to `jasper_library/platforms/` meeting the spec explained *earlier*.

Second, a platform yellow block python class is required. The class name should match your platform name (lower case), and the class should be stored in the file `jasper_library/yellow_blocks/<platform>.py`.

The `YellowBlock` class for the SNAP board is:

```
from yellow_block import YellowBlock
from constraints import ClockConstraint, PortConstraint, RawConstraint

class snap(YellowBlock):
    def initialize(self):
        self.add_source('infrastructure')
        self.add_source('wbs_arbiter')
        # 32-bit addressing => second half of 32 MByte memory. See UG470 v1.11 Table␣
↪7.2, Note 1
        self.usermemaddr = 0x800000  >> 8
        self.golden = False

    def modify_top(self,top):
        inst = top.get_instance('snap_infrastructure', 'snap_infrastructure_inst')
        inst.add_port('sys_clk_buf_n', 'sys_clk_n', parent_port=True, dir='in')
        inst.add_port('sys_clk_buf_p', 'sys_clk_p', parent_port=True, dir='in')
        inst.add_port('sys_clk0      ', 'sys_clk    ')
        inst.add_port('sys_clk180    ', 'sys_clk180')
        inst.add_port('sys_clk270    ', 'sys_clk270')
        inst.add_port('clk_200       ', 'clk_200    ')
        inst.add_port('sys_rst       ', 'sys_rst    ')
        inst.add_port('idelay_rdy    ', 'idelay_rdy')

        top.add_signal('sys_clk90')
        top.assign_signal('sys_clk90', '~sys_clk270')

    def gen_children(self):
        children = [YellowBlock.make_block({'tag':'xps:sys_block', 'board_id':'12',
↪'rev_maj':'12', 'rev_min':'0', 'rev_rcs':'32'}, self.platform)]
        if self.use_microblaze:
            children.append(YellowBlock.make_block({'tag':'xps:microblaze'}, self.
↪platform))
        else:
            children.append(YellowBlock.make_block({'tag':'xps:spi_wb_bridge'}, self.
↪platform))
            # XADC is embedded in the microblaze core, so don't include another one␣
↪unless we're not microblazin'
            children.append(YellowBlock.make_block({'tag':'xps:xadc'}, self.platform))
        return children

    def gen_constraints(self):
```

(continues on next page)

(continued from previous page)

```
        cons =[
            PortConstraint('sys_clk_n', 'sys_clk_n'),
            PortConstraint('sys_clk_p', 'sys_clk_p'),
            ClockConstraint('sys_clk_p', period=5.0),
            RawConstraint('set_property CONFIG_VOLTAGE 2.5 [current_design]'),
            RawConstraint('set_property CFGBVS VCCO [current_design]'),
            RawConstraint('set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_
↪design]'),
            RawConstraint('set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_
↪design]'),
            RawConstraint('set_property BITSTREAM.CONFIG.SPI_32BIT_ADDR Yes [current_
↪design]'),
            RawConstraint('set_property BITSTREAM.CONFIG.TIMER_CFG 2000000 [current_
↪design]'), # about 10 seconds
        ]
        if self.golden:
            #cons += [RawConstraint('set_property BITSTREAM.CONFIG.NEXT_CONFIG_ADDR 0x
↪%.7x [current_design]' % self.usermemaddr),]
            pass
        else:
            cons += [RawConstraint('set_property BITSTREAM.CONFIG.CONFIGFALLBACK␣
↪ENABLE [current_design]'),]
        return cons

    def gen_tcl_cmds(self):
        tcl_cmds = {}
        # After generating bitstream write PROM file
        # Write both mcs and bin files. The latter are good for remote programming␣
↪via microblaze. And makes sure the
        # microblaze code makes it into top.bin, and hence top.bof
        tcl_cmds['promgen'] = []
        tcl_cmds['promgen'] += ['write_cfgmem  -format mcs -size 32 -interface SPIx4 -
↪loadbit "up 0x0 ./myproj.runs/impl_1/top.bit " -checksum -file "./myproj.runs/impl_
↪1/top.mcs" -force']
        tcl_cmds['promgen'] += ['write_cfgmem  -format mcs -size 32 -interface SPIx4 -
↪loadbit "up 0x%.7x ./myproj.runs/impl_1/top.bit " -checksum -file "./myproj.runs/
↪impl_1/top_0x%x.mcs" -force' % (self.usermemaddr, self.usermemaddr)]
        tcl_cmds['promgen'] += ['write_cfgmem  -format bin -size 32 -interface SPIx4 -
↪loadbit "up 0x0 ./myproj.runs/impl_1/top.bit " -checksum -file "./myproj.runs/impl_
↪1/top.bin" -force']
        return tcl_cmds
```

This class instantiates a `snap_infrastructure` module in a design's top-level verilog file (via `modify_top`), and requires the source files `jasper_library/hdl_sources/wbs_arbiter/*` and `jasper_library/hdl_sources/infrastructure/*` (via `initialize`). You'll probably want to make an infrastructure block which is appropriate to your platform – i.e., takes in an external clock signal, and generates a buffered system clock which other modules in a design can use.

There are a few rules which platform `YellowBlock` subclasses must adhere to in order not to break the toolflow:

- They should add the source `wbs_arbiter` (this is a silly requirement which should be mitigated)
- They should generate a buffered clock called `sys_clk`, and its 90/180/270 degree counterparts `sysclk90`, `sysclk180` and `sysclk270`.
- `sys_clk` should be 100 MHz. (This is not strictly a requirement, but other modules assume `sys_clk` is relatively slow, and therefore appropriate for use in high combinatorial-logic paths)

The SNAP block attaches the external `sys_clk_n` and `sys_clk_p` ports to ports with the same names in the platforms YAML file via the `PortConstraint` instances returned by `gen_constraints`. It is not necessary for the two arguments of the `PortConstraint` to match – i.e., it is acceptable to connect the top-level port `sys_clk_n` to the platform pin designated `foo_clk` – but maintaining some consistency with existing platforms is recommended where possible.

A `ClockConstraint` should also be defined in `gen_constraints`, which appropriately sets the rate of the clock being used to derive the 100 MHz system clock.

The SNAP example above also uses a variety of other features which may be useful for new platforms.

- `RawConstraint` instances are used to pass bitstream configuration information to the downstream compiler.
- An optional method `gen_tcl_cmds` is included, which adds tcl commands directly to the final compilation script. In this case, these commands are used to generate a SNAP-appropriate prom file from the compiled bitstream.
- An optional method `gen_children` is used to instantiate other yellow blocks in the design as if the user had placed them in their Simulink model.

#### Compiling

With the above changes, you should be able to compile a design for your new platform using the toolflow!

During the development process, it is recommended that you look at the generated `jasper.per` and `top.v` files to check how definitions in python classes and YAML configuration files are being turned into code. The Yellow Block tutorial will probably prove helpful in further understanding the machinations of the toolflow.

Of course, you can only use yellow blocks in your design which your platform understands. Some will be automatically supported – software registers and shared bram blocks have no top-level ports or platform-specific HDL, and so are sompletely supported. Others can be supported by simply ensuring that your platform's YAML configuration file has appropriate entries for all the pins needed by a yellow block. This is the case, for example, with the `spi_wb_bridge` block, which allows a design's on-chip wishbone bus to be driven via an external SPI interface. However, most more complex yellow blocks (Ethernet / Memory interfaces) will require some level of customization to be supported.

#### Adding a New Peripheral

Adding a new peripheral is well covered in the Yellow Block tutorial

### 4.2.3 jasper_library

#### castro

**class** `castro.`**`Castro`**(*design_name*, *src_files*, *ips=[]*, *mm_slaves=[]*, *temp_fpga_model=”*)

Stores complete generic structure design information

**`__init__`**(*design_name*, *src_files*, *ips=[]*, *mm_slaves=[]*, *temp_fpga_model=”*)

x.__init__(. . . ) initializes x; see help(type(x)) for signature

**`dump`**(*filename*)

saves this class object to a yaml file

**static load**(*filename*)

loads this class object from a yaml file and assert that it is of type Castro

**class** castro.**ClkConstraint**(*portname*, *period_ns*, *freq_mhz=100*, *clkname=None*, *waveform_min_ns=None*, *waveform_max_ns=None*, *port_en=True*, *virtual_en=False*)
Class to hold a clock constraint

**__init__**(*portname*, *period_ns*, *freq_mhz=100*, *clkname=None*, *waveform_min_ns=None*, *waveform_max_ns=None*, *port_en=True*, *virtual_en=False*)
x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**ClkGrpConstraint**(*clknamegrp1=None*, *clknamegrp2=None*, *clkdomaintype=None*)
Class to hold a clock group constraint.

**__init__**(*clknamegrp1=None*, *clknamegrp2=None*, *clkdomaintype=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**FalsePthConstraint**(*sourcepath=None*, *destpath=None*)
Class to hold a false path constraint.

**__init__**(*sourcepath=None*, *destpath=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**GenClkConstraint**(*pinname*, *clkname=None*, *divide_by=None*, *clksource=None*)
Class to hold a generated clock constraint.

**__init__**(*pinname*, *clkname=None*, *divide_by=None*, *clksource=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**InDelayConstraint**(*clkname=None*, *consttype=None*, *constdelay_ns=None*, *add_delay_en=None*, *portname=None*)
Class to hold an Input Delay constraint.

**__init__**(*clkname=None*, *consttype=None*, *constdelay_ns=None*, *add_delay_en=None*, *portname=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**MaxDelayConstraint**(*sourcepath=None*, *destpath=None*, *constdelay_ns=None*)
Class to hold a Max Delay constraint.

**__init__**(*sourcepath=None*, *destpath=None*, *constdelay_ns=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**MinDelayConstraint**(*sourcepath=None*, *destpath=None*, *constdelay_ns=None*)
Class to hold a Min Delay constraint.

**__init__**(*sourcepath=None*, *destpath=None*, *constdelay_ns=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**MultiCycConstraint**(*multicycletype=None*, *sourcepath=None*, *destpath=None*, *multicycledelay=None*)
Class to hold a multi cycle constraint.

**__init__**(*multicycletype=None*, *sourcepath=None*, *destpath=None*, *multicycledelay=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**OutDelayConstraint**(*clkname=None*, *consttype=None*, *constdelay_ns=None*, *add_delay_en=None*, *portname=None*)
Class to hold an Output Delay constraint.

**__init__**(*clkname=None*, *consttype=None*, *constdelay_ns=None*, *add_delay_en=None*, *portname=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**PinConstraint**(*portname*, *symbolic_name*, *portname_indices=None*, *symbolic_indices=None*, *location="*, *drive_strength=0*, *slew_rate=0*, *io_standard="*, *termination="*)

Class to hold a pin constraint

**__init__**(*portname*, *symbolic_name*, *portname_indices=None*, *symbolic_indices=None*, *location="*, *drive_strength=0*, *slew_rate=0*, *io_standard="*, *termination="*)

x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**RawConstraint**(*raw*)

Class to hold raw constraints. These are really against the spirit of castro, since they are tool-specific. But, being pragmatic, sometimes they are necessary to encode simple constraints, for highly technology-specific features. The contents of these is not defined by castro.

**__init__**(*raw*)

x.__init__(...) initializes x; see help(type(x)) for signature

**class** castro.**Synthesis**(*platform_name="*, *fpga_manufacturer="*, *fpga_model="*, *synth_tool="*, *pin_map=[]*, *vendor_constraints_files="*, *pin_constraints=[]*, *clk_constraints=[]*, *gen_clk_constraints=[]*, *clk_grp_constraints=[]*, *input_delay_constraints=[]*, *output_delay_constraints=[]*, *max_delay_constraints=[]*, *min_delay_constraints=[]*, *multi_cycle_constraints=[]*, *false_path_constraints=[]*, *raw_constraints=[]*, *temp_fpga_model="*, *temp_quartus_qsf_files=[]*)

A class to specify all the synthesis specific attributes of the design

**__init__**(*platform_name="*, *fpga_manufacturer="*, *fpga_model="*, *synth_tool="*, *pin_map=[]*, *vendor_constraints_files="*, *pin_constraints=[]*, *clk_constraints=[]*, *gen_clk_constraints=[]*, *clk_grp_constraints=[]*, *input_delay_constraints=[]*, *output_delay_constraints=[]*, *max_delay_constraints=[]*, *min_delay_constraints=[]*, *multi_cycle_constraints=[]*, *false_path_constraints=[]*, *raw_constraints=[]*, *temp_fpga_model="*, *temp_quartus_qsf_files=[]*)

x.__init__(...) initializes x; see help(type(x)) for signature

**resolve_constraint**(*constraint*)

Ensure constraint targets existing platform constraints

**class** castro.**mm_slave**(*name*, *mode*, *base_address*, *span*)

JASPER: A list of elements of this class gets merged with the bitsream

**__init__**(*name*, *mode*, *base_address*, *span*)

x.__init__(...) initializes x; see help(type(x)) for signature

### constraints

**class** constraints.**ClockConstraint**(*signal=None*, *name=None*, *freq=None*, *period=None*, *port_en=True*, *virtual_en=False*, *waveform_min=0.0*, *waveform_max=None*)

A clock constraint – simply holds the name of the clock signal, clock name, whether clock source is get_ports or get_pins, whether a virtual clock, waveform parameters for duty cycle and the corresponding clock freq and period.

This assigns the clock timing constraint on the clock port in user_const.xdc, for example:

`ClockConstraint('A','A', period=6.4, port_en=True, virtual_en=False, waveform_min=0.0, waveform_max=3.2))` is translated to `create_clock -period 6.400 -name A -waveform {0.000 3.200} [get_ports {A}]` in the xdc file.

This tells Vivado which ports should be clocks.

**__init__**(*signal=None*, *name=None*, *freq=None*, *period=None*, *port_en=True*, *virtual_en=False*, *waveform_min=0.0*, *waveform_max=None*)

Construct a ClockConstraint instance.

**Parameters**

- **signal** (*str*) – The signal name of the clock port
- **name** (*str*) – The name of the clock
- **freq** (*float*) – The clock frequency in MHz (no need to specify period if the frequency is specified)
- **period** (*float*) – The period of the clock in ns (no need to specify frequency if the period is specified)
- **port_en** (*boolean*) – If True then the clock port is enabled. If False then the clock port is bypassed for the case of a virtual clock.
- **virtual_en** (*bool*) – This is set to True when using a virtual clock, otherwise it is False.
- **waveform_min** (*float*) – This parameter is used to determine the duty cycle of the clock in ns. Typically 0ns.
- **waveform_max** (*float*) – This parameter is used to determine the duty cycle of the clock in ns. Typically half the period of the clock for a 50% duty cycle.

**class** constraints.**ClockGroupConstraint**(*clock_name_group_1=None*, *clock_name_group_2=None*, *clock_domain_relationship=None*)

A clock group constraint – simply holds the name of both clock domains and the domain relationship e.g. asynchronous

This assigns the clock group timing constraint on two or more clock groups in user_const.xdc, for example:

`ClockGroupConstraint('A', 'B', 'asynchronous')` is translated to `set_clock_groups -asynchronous -group [get_clocks A] -group [get_clocks B]` in the xdc file.

This constraint is used to cut the clock relationship between two or more clock groups.

**__init__**(*clock_name_group_1=None*, *clock_name_group_2=None*, *clock_domain_relationship=None*)

Construct a ClockGroupConstraint instance.

**Parameters**

- **clock_name_group_1** (*str*) – The clock name of the first group e.g. the clock port name or virtual clock name
- **clock_name_group_2** (*str*) – The clock name of the second group e.g. the clock port name or virtual clock name
- **clock_domain_relationship** (*str*) – This specifies the relationship between the two clock name groups. Typically this is set to `asynchronous` which tells the Vivado timing analyzer to ignore the timing relationship between these two clock domains, as the clocks are asynchronous.

**class** constraints.**FalsePathConstraint**(*sourcepath=None*, *destpath=None*)

A false path constraint - simply holds the source and destination paths.

This assigns the false path timing constraint in user_const.xdc, for example:

`FalsePathConstraint(destpath='[get_ports {A}]')` is translated to `set_false_path -to [get_ports {A}]` in the xdc file.

Any path that appears in the FalsePathConstraint is ignored by the Vivado timing analyzer.

**`__init__`**(*sourcepath=None*, *destpath=None*)
Construct a FalsePathConstraint instance.

**Parameters**

- **`sourcepath`** (*`str`*) – The source path that the constraint is applied to - includes path and port names.
- **`destpath`** (*`str`*) – The destination path that the constraint is applied to - includes path and port names.

**`class`** `constraints.`**`GenClockConstraint`**(*signal*, *name=None*, *divide_by=None*, *clock_source=None*)
A clock generation constraint – simply holds the name of the clock signal, clock name, clock source and divide by value.

This assigns the generated clock timing constraint on a non global clock port in user_const.xdc, for example:

`GenClockConstraint(signal='sub/Q', name='sub/CLK', divide_by=16, clock_source='sub/C')` is translated to `create_generated_clock -name sub/CLK -source [get_pins {sub/C}] -divide_by 16 [get_pins {sub/Q}]` in the xdc file.

This constraint is used to assign a clock to signals that are not inferred by Vivado naturally and should be.

**`__init__`**(*signal*, *name=None*, *divide_by=None*, *clock_source=None*)
Construct a GenClockConstraint instance.

**Parameters**

- **`signal`** (*`str`*) – The signal name that is required to be a clock
- **`name`** (*`str`*) – The name of the generated clock
- **`divide_by`** (*`int`*) – The value to divide the clock_source by in order to determine the clock frequency of the generated clock in MHz
- **`clock_source`** (*`str`*) – This is the clock source (input) of the generated clock. The clock_source and the divide_by value determined the generated clock out frequency in MHz: generated clock in MHz = clock_source*divide_by

**`class`** `constraints.`**`InputDelayConstraint`**(*clkname=None*, *consttype=None*, *constdelay_ns=None*, *add_delay_en=None*, *portname=None*)
An input delay constraint - simply holds the name of the reference clock, constraint type (min or max), constraint delay value (ns), whether an existing constraint exists and a new one needs to be added and the port name that the constraint applies to.

This assigns the clock input delay timing constraint in user_const.xdc, for example: `InputDelayConstraint(clkname='A', consttype='min', constdelay_ns=1.0, add_delay_en=True, portname='B')` is translated to `set_input_delay -clock [get_clocks A] -min -add_delay 1.000 [get_ports {B}]` in the xdc file.

This constraint is used to assign input constraints referenced to the clock.

**`__init__`**(*clkname=None*, *consttype=None*, *constdelay_ns=None*, *add_delay_en=None*, *portname=None*)
Construct a InputDelayConstraint instance.

**Parameters**

- **`clkname`** (*`str`*) – The clock name which the port name is referenced to
- **`consttype`** (*`str`*) – This is constraint type: either be a `min` (hold) or `max` (setup).

- **constdelay_ns** (*float*) – This is the constraint delay in ns - takes into account the Tco, clock skew and board delay.
- **add_delay_en** (*bool*) – If more than one constraint is needed on the portname then this is True, else set it to False.
- **portname** (*str*) – The port name of the signal that needs to be constrained.

**class** constraints.**MaxDelayConstraint**(*sourcepath=None*, *destpath=None*, *constdelay_ns=None*)

A set max delay constraint - simply holds the source, destination paths and the constraint delay value (ns).

This assigns the max delay timing constraint in user_const.xdc, for example: `MaxDelayConstraint(destpath='[get_ports {A}]', constdelay_ns=1.0)` is translated to `set_max_delay 1.0 -to [get_ports {A}]` in the xdc file.

This constraint is used when there is no clock reference.

**__init__**(*sourcepath=None*, *destpath=None*, *constdelay_ns=None*)

Construct a MaxDelayConstraint instance.

**Parameters**

- **sourcepath** (*str*) – The source path that the constraint is applied to - includes path and port names.
- **destpath** (*str*) – The destination path that the constraint is applied to - includes path and port names.
- **constdelay_ns** (*float*) – This is the constraint delay in ns - takes into account the Tsu, clock skew and board delay.

**class** constraints.**MinDelayConstraint**(*sourcepath=None*, *destpath=None*, *constdelay_ns=None*)

A set min delay constraint - simply holds the source, destination paths and the constraint delay value (ns).

This assigns the min delay timing constraint in user_const.xdc, for example:

`MinDelayConstraint(destpath='[get_ports {A}]', constdelay_ns=1.0)` is translated to `set_min_delay 1.0 -to [get_ports {A}]` in the xdc file.

This constraint is used when there is no clock reference.

**__init__**(*sourcepath=None*, *destpath=None*, *constdelay_ns=None*)

Construct a MinDelayConstraint instance.

**Parameters**

- **sourcepath** (*str*) – The source path that the constraint is applied to - includes path and port names.
- **destpath** (*str*) – The destination path that the constraint is applied to - includes path and port names.
- **constdelay_ns** (*float*) – This is the constraint delay in ns - takes into account the Th, clock skew and board delay.

**class** constraints.**MultiCycleConstraint**(*multicycletype=None*, *sourcepath=None*, *destpath=None*, *multicycledelay=None*)

A multi cycle constraint - simply holds the multi cycle type (steup or hold), source, destination paths and multi cycle delay value in clock cycles.

This assigns the multicycle timing constraint in user_const.xdc, for example:

`MultiCycleConstraint(multicycletype='setup',sourcepath='get_clocks B', destpath='get_ports A', multicycledelay=4)` is translated to `set_multicycle_path -setup -from [get_ports A] -to [get_clocks B] 4` in the xdc file.

This tells the Vivado timing analyzer that the signal will take more than one clock cycle to propagate through the logic.

**__init__**(*multicycletype=None*, *sourcepath=None*, *destpath=None*, *multicycledelay=None*)
Construct a MultiCycleConstraint instance.

**Parameters**

- **multicycletype** (*str*) – The type of multicycle constraint: either `setup` or `hold`
- **sourcepath** (*str*) – The source path that the constraint is applied to - includes path and port names.
- **destpath** (*str*) – The destination path that the constraint is applied to - includes path and port names.
- **multicycledelay** (*int*) – This represents the number of clock cycles to delay.

**class** constraints.**OutputDelayConstraint**(*clkname=None*, *consttype=None*, *constdelay_ns=None*, *add_delay_en=None*, *portname=None*)
An output delay constraint - simply holds the name of the reference clock, constraint type (min or max), constraint delay value (ns), whether an existing constraint exists and a new one needs to be added and the port name that the constraint applies to.

This assigns the clock output delay timing constraint in user_const.xdc, for example:

`OutputDelayConstraint(clkname='A', consttype='min', constdelay_ns=1.0, add_delay_en=True, portname='B')` is translated to `set_output_delay -clock [get_clocks A] -min -add_delay 1.000 [get_ports {B}]` in the xdc file.

This constraint is used to assign output constraints referenced to the clock.

**__init__**(*clkname=None*, *consttype=None*, *constdelay_ns=None*, *add_delay_en=None*, *portname=None*)
Construct a OutputDelayConstraint instance.

**Parameters**

- **clkname** (*str*) – The clock name which the port name is referenced to
- **consttype** (*str*) – This is constraint type: either be a `min` (hold) or `max` (setup).
- **constdelay_ns** (*float*) – This is the constraint delay in ns - takes into account the Tsu, Th, clock skew and board delay.
- **add_delay_en** (*bool*) – If more than one constraint is needed on the portname then this is True, else set it to False.
- **portname** (*str*) – The port name of the signal that needs to be constrained.

**class** constraints.**PortConstraint**(*portname, iogroup, port_index=[], iogroup_index=[0], loc=None, iostd=None*)
A class to facilitate constructing abstracted port constraints.

Eg, adc_data[7:0] <=> zdok0[7:0] which can later be translated into physical constraints by providing information about a target platform.

This assigns the port LOC and voltage constraints to user_const.xdc, for example:

`PortConstraint('A', 'A')` is translated to `set_property PACKAGE_PIN BC27 [get_ports A]` and `set_property IOSTANDARD LVCMOS18 [A]` in the xdc file. The “BC27”

LOC and "LVCMOS18" is determined by the platform yaml file, which contains all the platform top level ports and LOC assignments.

**`__init__`** (*portname, iogroup, port_index=[], iogroup_index=[0], loc=None, iostd=None*)
Construct a PortConstraint instance.

**Parameters**

- **`portname`** (`str`) – The name (in verilog) of the port
- **`port_index`** (`int`) – Specify an offset of the port index to attach to iogroup[index]. This feature was added so that we can do (eg.) myport[3:0] <=> gpioA[3:0], myport[7:4] <=> gpioB[3:0]
- **`iogroup`** (`str`) – The abstract name of the ports physical connection (eg. zdok0, zdok1, gpioa)
- **`iogroup_index`** (`int or list`) – The index of the abstract name to which the HDL port should connect
- **`loc`** (`list`) – Specify a loc to construct a physical constraint, forgoing the abstract names. Experimental.
- **`iostd`** – Specify an iostd to construct a physical constraint, forgoing the abstract names. Experimental.

**`gen_physical_const`** (*platform*)
Set the LOC and IOSTDs of an abstract constraint for a given platform.

**Parameters** **`platform`** (`Platform`) – The platform instance against which to evaluate the constraint(s).

**`class`** `constraints.`**`RawConstraint`** (*const*)
A class for raw constraints – strings to be dumper unadulterated into a constraint file.

This assigns any raw constraints (set_property, pblock etc) in user_const.xdc, for example:

`RawConstraint('set_property OFFCHIP_TERM NONE [get_ports A]')` is translated to `set_property OFFCHIP_TERM NONE [get_ports A]` in the xdc file.

Any constraint not handled in the above classes can be added using the raw constraints.

**`__init__`** (*const*)
Construct a RawConstraint instance.

**Parameters** **`const`** (`str`) – This represents the path with port names that the constraint is applied to

## exec_flow

`exec_flow.`**`shell_source`** (*script*)
Sometime you want to emulate the action of "source" in bash, settings some environment variables. Here is a way to do it.

## helpers

`helpers.`**`to_int_list`** (*s*)
take a string like `[0,1,2,3]` and return a list of integers

`helpers.`**`write_file`** (*fn, str*)
write string `str` to filename `fn`

## memory

**class** memory.**Register**(*name*, *nbytes=4*, *offset=0*, *mode='r'*, *default_val=0*, *ram=False*, *ram_size=-1*, *data_width=32*)

**__init__**(*name*, *nbytes=4*, *offset=0*, *mode='r'*, *default_val=0*, *ram=False*, *ram_size=-1*, *data_width=32*)

A class to encapsulate a register's parameters. This is used when instantiating a device with a large address space, but it is desirable to be able to address sub-spaces of this memory with separate names.

For example (see sys_block.py):

```
class sys_block(YellowBlock):
    def initialize(self):
        self.typecode = TYPECODE_SYSBLOCK
        self.add_source('sys_block')
        # the internal memory_map
        self.memory_map = [
            Register('sys_board_id',   mode='r',  offset=0),
            Register('sys_rev',        mode='r',  offset=0x4),
            Register('sys_rev_rcs',    mode='r',  offset=0xc),
            Register('sys_scratchpad', mode='rw', offset=0x10),
            Register('sys_clkcounter', mode='r',  offset=0x14),
        ]
    def modify_top(self,top):
        inst = top.get_instance('sys_block', 'sys_block_inst')
        inst.add_parameter('BOARD_ID', self.board_id)
        inst.add_parameter('REV_MAJ', self.rev_maj)
        inst.add_parameter('REV_MIN', self.rev_min)
        inst.add_parameter('REV_RCS', self.rev_rcs)
        inst.add_port('user_clk', 'user_clk')
        inst.add_wb_interface('sys_block', mode='r', nbytes=64, memory_
↪map=self.memory_map, typecode=self.typecode)
```

**Parameters**

- **name** (*String*) – The name of this register
- **nbytes** (*Integer*) – Number of bytes this register occupies
- **offset** (*Integer*) – Location, in bytes, where this register resides in memory, relative to the base address of the device.
- **mode** (*String*) – Read/write permission for this register. 'r' (readable), 'w' (writable), 'rw' (read/writable)
- **default_val** (*Integer*) – Default value for register to be reset to and initialized.

## platform

This module tries to retrieve as much platform-identifying data as possible. It makes this information available via function APIs.

If called from the command line, it prints the platform information concatenated as single string to stdout. The output format is useable as part of a filename.

platform.**architecture**(*executable='/home/docs/checkouts/readthedocs.org/user_builds/casper-toolflow/envs/stable/bin/python'*, *bits=''*, *linkage=''*)

Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple (bits,linkage) which contains information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If bits is given as '', the sizeof(pointer) (or sizeof(long) on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system's "file" command to do the actual work. This is available on most if not all Unix platforms. On some non-Unix platforms where the "file" command does not exist and the executable is set to the Python interpreter binary defaults from _default_architecture are used.

`platform.`**`dist`**(*distname='', version='', id='', supported_dists=('SuSE', 'debian', 'fedora', 'redhat', 'centos', 'mandrake', 'mandriva', 'rocks', 'slackware', 'yellowdog', 'gentoo', 'UnitedLinux', 'turbolinux')*)

Tries to determine the name of the Linux OS distribution name.

The function first looks for a distribution release file in /etc and then reverts to _dist_try_harder() in case no suitable files are found.

Returns a tuple (distname,version,id) which default to the args given as parameters.

`platform.`**`java_ver`**(*release='', vendor='', vminfo=('', '', ''), osinfo=('', '', '')*)

Version interface for Jython.

Returns a tuple (release,vendor,vminfo,osinfo) with vminfo being a tuple (vm_name,vm_release,vm_vendor) and osinfo being a tuple (os_name,os_version,os_arch).

Values which cannot be determined are set to the defaults given as parameters (which all default to '').

`platform.`**`libc_ver`**(*executable='/home/docs/checkouts/readthedocs.org/user_builds/casper-toolflow/envs/stable/bin/python', lib='', version='', chunksize=2048*)

Tries to determine the libc version that the file executable (which defaults to the Python interpreter) is linked against.

Returns a tuple of strings (lib,version) which default to the given parameters in case the lookup fails.

Note that the function has intimate knowledge of how different libc versions add symbols to the executable and thus is probably only useable for executables compiled using gcc.

The file is read and scanned in chunks of chunksize bytes.

`platform.`**`linux_distribution`**(*distname='', version='', id='', supported_dists=('SuSE', 'debian', 'fedora', 'redhat', 'centos', 'mandrake', 'mandriva', 'rocks', 'slackware', 'yellowdog', 'gentoo', 'UnitedLinux', 'turbolinux'), full_distribution_name=1*)

Tries to determine the name of the Linux OS distribution name.

The function first looks for a distribution release file in /etc and then reverts to _dist_try_harder() in case no suitable files are found.

supported_dists may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If full_distribution_name is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from supported_dists is used.

Returns a tuple (distname,version,id) which default to the args given as parameters.

`platform.`**`mac_ver`**(*release='', versioninfo=('', '', ''), machine=''*)

Get MacOS version information and return it as tuple (release, versioninfo, machine) with versioninfo being a tuple (version, dev_stage, non_release_version).

Entries which cannot be determined are set to the parameter values which default to ''. All tuple entries are strings.

`platform.`**`machine`**`()`

Returns the machine type, e.g. 'i386'

An empty string is returned if the value cannot be determined.

`platform.`**`node`**`()`

Returns the computer's network name (which may not be fully qualified)

An empty string is returned if the value cannot be determined.

`platform.`**`platform`**`(`*aliased=0*, *terse=0*`)`

Returns a single string identifying the underlying platform with as much useful information as possible (but no more :).

The output is intended to be human readable rather than machine parseable. It may look different on different platforms and this is intended.

If "aliased" is true, the function will use aliases for various platforms that report system names which differ from their common names, e.g. SunOS will be reported as Solaris. The system_alias() function is used to implement this.

Setting terse to true causes the function to return only the absolute minimum information needed to identify the platform.

`platform.`**`popen`**`(`*cmd*, *mode='r'*, *bufsize=None*`)`

Portable popen() interface.

`platform.`**`processor`**`()`

Returns the (true) processor name, e.g. 'amdk6'

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for machine(), e.g. NetBSD does this.

`platform.`**`python_branch`**`()`

Returns a string identifying the Python implementation branch.

For CPython this is the Subversion branch from which the Python binary was built.

If not available, an empty string is returned.

`platform.`**`python_build`**`()`

Returns a tuple (buildno, builddate) stating the Python build number and date as strings.

`platform.`**`python_compiler`**`()`

Returns a string identifying the compiler used for compiling Python.

`platform.`**`python_implementation`**`()`

Returns a string identifying the Python implementation.

**Currently, the following implementations are identified:** 'CPython' (C implementation of Python), 'IronPython' (.NET implementation of Python), 'Jython' (Java implementation of Python), 'PyPy' (Python implementation of Python).

`platform.`**`python_revision`**`()`

Returns a string identifying the Python implementation revision.

For CPython this is the Subversion revision from which the Python binary was built.

If not available, an empty string is returned.

`platform.`**`python_version`**`()`

Returns the Python version as string 'major.minor.patchlevel'

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to 0).

platform.**python_version_tuple**()

Returns the Python version as tuple (major, minor, patchlevel) of strings.

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to 0).

platform.**release**()

Returns the system's release, e.g. '2.2.0' or 'NT'

An empty string is returned if the value cannot be determined.

platform.**system**()

Returns the system/OS name, e.g. 'Linux', 'Windows' or 'Java'.

An empty string is returned if the value cannot be determined.

platform.**system_alias**(*system*, *release*, *version*)

Returns (system,release,version) aliased to common marketing names used for some systems.

It also does some reordering of the information in some cases where it would otherwise cause confusion.

platform.**uname**()

Fairly portable uname interface. Returns a tuple of strings (system,node,release,version,machine,processor) identifying the underlying platform.

Note that unlike the os.uname function this also returns possible processor information as an additional tuple entry.

Entries which cannot be determined are set to ''.

platform.**version**()

Returns the system's release version, e.g. '#3 on degas'

An empty string is returned if the value cannot be determined.

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)

## toolflow

A python-based toolflow to build a vivado project from a simulink design, using the CASPER xps library.

A work in progress.

**class** toolflow.**ISEBackend**(*plat=None*, *compile_dir='/tmp'*)

**__init__**(*plat=None*, *compile_dir='/tmp'*)

**Parameters**

- **plat** –
- **compile_dir** –

**add_compile_cmds**(*cores=8*, *plat=None*)

add the tcl commands for compiling the design, and then launch vivado in batch mode

**compile**(*cores*, *plat*)

**static format_clock_const**(*c*)

**static format_const**(*attribute*, *val*, *port*, *index=None*)

Generate a tcl syntax command from an attribute, value and port (with indexing if required)

**gen_constraint_file**(*constraints*)
Pass this method a toolflow-standard list of constraints which have already had their physical parameters calculated and it will generate a contstraint file and add it to the current project.

**get_ucf_const**(*const*)
Pass a single toolflow-standard PortConstraint, and get back a tcl command to add the constraint to a vivado project.

**class** `toolflow.`**SimulinkFrontend**(*compile_dir='/tmp'*, *target='/tmp/test.slx'*)

**__init__**(*compile_dir='/tmp'*, *target='/tmp/test.slx'*)

**Parameters**

- **compile_dir** –
- **target** –

**compile_user_ip**(*update=False*)
Compile the users simulink design. The resulting netlist should end up in the location already specified in the peripherals file.

**Parameters** **update** (`bool`) – Update the simulink model before running system generator

**gen_periph_file**(*fname='jasper.per'*)
generate the peripheral file.

i.e., the list of yellow blocks and their parameters.

It also generates the `design_info.tab` file which is used to populate the fpg file header

**Parameters** **fname** (`str`) – The full path and name to give the peripheral file.

**write_git_info_file**(*fname='git_info.tab'*)
Get the git info for mlib_devel and the model file. :param fname: :return:

**class** `toolflow.`**Toolflow**(*frontend='simulink'*, *compile_dir='/tmp'*, *frontend_target='/tmp/test.slx'*, *jobs=8*)
A class embodying the main functionality of the toolflow. This class is responsible for generating a complete top-level verilog description of a project from a 'peripherals file' which encodes information about which IP a user wants instantiated.

The toolflow class can parse such a file, and use it to generate verilog, a list of source files, and a list of constraints. These can be passed off to a toolflow backend to be turned into some vendor-specific platform and compiled. At least, that's the plan…

**__init__**(*frontend='simulink'*, *compile_dir='/tmp'*, *frontend_target='/tmp/test.slx'*, *jobs=8*)
Initialize the toolflow.

**Parameters**

- **frontend** (`str`) – Name of the toolflow frontend to use. Currently only `simulink` is supported
- **compile_dir** – Compile directory where build files and logs should go.

**build_top**()
Copies the base top-level verilog file (which is platform dependent) to the compile directory. Constructs an associated VerilogModule instance ready to be modified.

**check_attr_exists**(*thing*, *generator*)
Lots of methods in this class require that certain attributes have been set by other methods before proceeding. This is probably a symptom of the code being terribly structured. This method checks if an attribute

exists and throws an error message if not. In principle it could automatically run the necessary missing steps, but that seems pretty suspect.

**Parameters**

- **`thing`** (*`str`*) – Attribute to check.
- **`generator`** (*`str`*) – Method which can be used to set thing (used for error message only)

**`constraints_rule_check`**()
Check pin constraints against top level signals. Warn about missing constraints.

**`dump_castro`**(*filename*)
Build a 'standard' Castro object, which is the interface between the toolflow and the backends.

**`exec_flow`**(*gen_per=True*, *frontend_compile=True*)
Execute a compile.

**Parameters**

- **`gen_per`** (*`bool`*) – Have the toolflow frontend generate a fresh peripherals file
- **`frontend_compile`** (*`bool`*) – Run the frontend compiler (eg. System Generator)

**`gen_periph_objs`**()
Generate a list of yellow blocks from the current peripheral file.

Internally, calls:

- `_parse_periph_file`: parses .per file
- `_extract_plat_info`: instantiates platform instance

Then calls each yellow block's constructor. Runs a system-wide drc before returning.

**`generate_consts`**()
Compose a list of constraints from each yellow block. Use platform information to generate the appropriate physical realisation of each constraint.

**`generate_hdl`**()
Generates a top file for the target platform based on the peripherals file.

Internally, calls:

- `instantiate_periphs`: call each yellow block's mod_top method
- `instantiate_user_ip`: add ports to top module based on port entries in peripheral file
- `regenerate_top`: rewrite top.v

**`generate_xml_ic`**(*memory_map*)
Generate xml interconnect file that represent top-level AXI4-Lite interconnect for Oxford's xml2vhdl.

**`generate_xml_memory_map`**(*memory_map*)
Generate xml memory map files that represent each AXI4-Lite interface for Oxford's xml2vhdl.

**`regenerate_top`**()
Generate the verilog for the modified top module. This involves computing the wishbone interconnect / addressing and generating new code for yellow block instances.

**`write_core_info`**()

**`write_core_jam_info`**()

**xml2vhdl**()

Function to call Oxford's python code to generate AXI4-Lite VHDL register interfaces from an XML memory map specification.

Obtained from: https://bitbucket.org/ricch/xml2vhdl/src/master/

**class** toolflow.**ToolflowBackend**(*plat=None*, *compile_dir='/tmp'*)

**__init__**(*plat=None*, *compile_dir='/tmp'*)

**Parameters**

- **plat** –
- **compile_dir** –

**add_const_file**(*constfile*)

Add a constraint file to the project. via a tcl incantation. In non-project mode, it is important to note that copies are not made of files. The files are read from their source directory. Project mode copies files from their source directory and adds them to the a new compile directory.

**Parameters constfile** –

**add_source**(*source*, *plat*)

Add a sourcefile to the project. Via a tcl incantation. In non-project mode, it is important to note that copies are not made of files. The files are read from their source directory. Project mode copies files from their source directory and adds them to the a new compile directory.

**static calculate_checksum_using_bitstream**(*bitstream*, *packet_size=8192*)

Summing up all the words in the input bitstream, and returning a `Checksum` - Assuming that the bitstream HAS NOT been padded yet

**Parameters**

- **bitstream** – The actual bitstream of the file in question
- **packet_size** – max size of image packets that we pad to

**Returns** checksum

**compile**(*core*, *plat*)

**Parameters**

- **core** –
- **plat** –

**gen_constraint_file**(*constraints*)

Pass this method a toolflow-standard list of constraints which have already had their physical parameters calculated and it will generate a constraint file and add it to the current project.

**import_from_castro**(*filename*)

**initialize**(*plat*)

**Parameters plat** –

**mkfpg**(*filename_bin*, *filename_fpg*)

This function makes the fpg file header and the final fpg file, which consists of the fpg file header (core_info.tab, design_info.tab and git_info.tab) and the compressed binary file. The fpg file is used to configure the ROACH, ROACH2, MKDIG and SKARAB boards.

**Parameters**

- **`filename_bin`** (*`str`*) – This is the path and binary file (top.bin) that contains the FPGA programming data.
- **`filename_fpg`** (*`str`*) – This is the output time stamped fpg file name

**class** `toolflow.`**`ToolflowFrontend`**(*compile_dir='/tmp'*, *target='/tmp/test.slx'*)

**`__init__`**(*compile_dir='/tmp'*, *target='/tmp/test.slx'*)

**Parameters**

- **`compile_dir`** –
- **`target`** –

**`compile_user_ip`**()

Compile the user IP to a single HDL module.

Return the name of this module.

Should be overridden by each FrontEnd subclass.

**`gen_periph_file`**(*fname='jasper.per'*)

Call upon the frontend to generate a jasper-standard file defining peripherals (yellow blocks) present in a model.

This method should be overridden by the specific frontend of choice, and should return the full path to the peripheral file.

Use `skip = True` to just return the name of the file, without bothering to regenerate it (useful for debugging, and future use cases where a user only wants to run certain steps of a compile)

**`write_git_info_file`**(*fname='git_info.tab'*)

Call upon the frontend to generate a git info file, which contains the git repo information, which is used for the header for the fpg file. This function is overwritten by the SimulinkFrontEnd Class

**class** `toolflow.`**`VivadoBackend`**(*plat=None*, *compile_dir='/tmp'*, *periph_objs=None*)

**`__init__`**(*plat=None*, *compile_dir='/tmp'*, *periph_objs=None*)

**Parameters**

- **`plat`** –
- **`compile_dir`** –
- **`periph_objs`** –

**`add_compile_cmds`**(*cores=8*, *plat=None*, *synth_strat=None*, *impl_strat=None*)

Add the tcl commands for compiling the design, and then launch vivado in batch mode

**`add_const_file`**(*constfile*)

Add a constraint file to the project. via a tcl incantation. In non-project mode, it is important to note that copies are not made of files. The files are read from their source directory. Project mode copies files from their source directory and adds them to the a new compile directory.

**Parameters** **`constfile`** –

**`add_ip`**(*ip*)

Add an ip core from a library

**`add_library`**(*path*)

Add a library at <path>

**add_source**(*source*, *plat*)
Add a sourcefile to the project. Via a tcl incantation. In non-project mode, it is important to note that copies are not made of files. The files are read from their source directory. Project mode copies files from their source directory and adds them to the a new compile directory.

**add_tcl_cmd**(*cmd*, *stage='pre_synth'*)
Add a command to the tcl command list with a trailing newline.

**compile**(*cores*, *plat*, *synth_strat=None*, *impl_strat=None*)

**Parameters**

- **cores** –
- **plat** –
- **impl_strat** – Implementation Strategy to use when carrying out the implementation run 'impl'

**eval_tcl**()

**static format_cfg_const**(*attribute*, *val*)
Generate a configuration tcl syntax command from an attribute and value

**static format_clock_const**(*c*)

**static format_clock_group_const**(*c*)

**static format_const**(*attribute*, *val*, *port*, *index=None*)
Generate a tcl syntax command from an attribute, value and port (with indexing if required)

**static format_false_path_const**(*c*)

**static format_gen_clock_const**(*c*)

**static format_input_delay_const**(*c*)

**static format_max_delay_const**(*c*)

**static format_min_delay_const**(*c*)

**static format_multi_cycle_const**(*c*)

**static format_output_delay_const**(*c*)

**gen_add_compile_dir_source_tcl_cmds**()
Run each blocks add_compile_dir_source functions and add them to the projects sources

**gen_constraint_file**(*constraints*)
Pass this method a toolflow-standard list of constraints which have already had their physical parameters calculated and it will generate a constraint file and add it to the current project.

**gen_yellowblock_custom_hdl**()
Create each yellowblock's custom hdl files and add them to the projects sources

**gen_yellowblock_tcl_cmds**()
Compose a list of tcl commands from each yellow block. To be added to the final tcl script.

**get_tcl_const**(*const*)
Pass a single toolflow-standard PortConstraint, and get back a tcl command to add the constraint to a vivado project.

**initialize**(*plat*)

**Parameters** **plat** –

## verilog

Lots of code in this file could be shared between methods and the VerilogInstance/Module classes. Maybe distill at some point.

**class** verilog.**AXI4LiteDevice**(*regname*, *nbytes*, *mode*, *hdl_suffix=''*, *hdl_candr_suffix=''*, *memory_map=[]*, *typecode=255*, *data_width=32*)

A class to encapsulate the parameters (name, size, etc.) of a AXI4-Lite slave device.

**__init__**(*regname*, *nbytes*, *mode*, *hdl_suffix=''*, *hdl_candr_suffix=''*, *memory_map=[]*, *typecode=255*, *data_width=32*)

Class constructor.

**Parameters**

- **regname** (*String*) – Name of register (this name is the string used to access the register from software)
- **nbytes** (*Integer*) – Number of bytes in this slave's memory space.
- **mode** (*String*) – Permissions ('r': readable, 'w': writable, 'rw': read/writeable)
- **hdl_suffix** (*String*) – Suffix given to wishbone port names. Eg. if *hdl_suffix = foo*, ports have the form *wbs_dat_i_foo*
- **hdl_candr_suffix** (*String*) – Suffix given to wishbone clock and reset port names. Eg. if *hdl_suffix = foo*, ports have the form *wbs_clk_i_foo*
- **memory_map** (*list*) – A list or *Register* instances defining the contents of sub-blocks of this device's memory.
- **typecode** (*Integer*) – Typecode number (0-255) identifying the type of this block. See *yellow_block_typecodes.py*
- **data_width** (*Integer*) – Width of the data to be stored in this device

**base_addr = None**

Start (lowest) address of the memory space used by this device, in bytes.

**high_addr = None**

End (highest) address of the memory space used by this device, in bytes.

**class** verilog.**ImmutableWithComments**

A class which you can add attributes to, but you can't change them once they're set. You are allowed to try and set them to the same value again. The `comment` attribute is special. Each time you try to set it, the comment string is appended to the existing comment attribute.

**__init__**()

x.__init__(...) initializes x; see help(type(x)) for signature

**class** verilog.**Parameter**(*name*, *value*, *comment=None*)

A simple class to hold parameter attributes. It is immutable, and will throw an error if its attributes are changed after being set.

**__init__**(*name*, *value*, *comment=None*)

Create a `Parameter` instance.

**Parameters**

- **name** (*str*) – Name of this parameter
- **value** (*Varies*) – Value this parameter should be set to.
- **comment** (*str*) – User-assisting comment string to attach to this parameter.

**update_attrs**(*name*, *value*, *comment=None*)
Update the attributes of this block.

**Parameters**

- **name** (*str*) – Name of this parameter
- **value** (*Varies*) – Value this parameter should be set to.
- **comment** (*str*) – User-assisting comment string to attach to this parameter.

**class** verilog.**Port**(*name*, *signal=None*, *parent_port=False*, *parent_sig=True*, ***kwargs*)
A simple class to hold port attributes. It is immutable, and will throw an error if multiple manipulation attempts are incompatible.

**__init__**(*name*, *signal=None*, *parent_port=False*, *parent_sig=True*, ***kwargs*)
Create a `Port` instance.

**Parameters**

- **name** – Name of the port
- **signal** (*str*) – Signal to which this port is attached
- **parent_port** (*bool*) – When module 'A' instantiates the module to which this port is attached, should this port be connected to a similar port on 'A'.
- **parent_sig** (*bool*) – When module 'A' instantiates the module to which this port is attached, should 'A' also instantiate a signal matching the one connected to this port.
- **kwargs** – Other keywords which should become attributes of this instance.

**update_attrs**(*name*, *signal=None*, *parent_port=False*, *parent_sig=True*, ***kwargs*)
Update the attributes of this block.

**Parameters**

- **name** – Name of the port
- **signal** (*str*) – Signal to which this port is attached
- **parent_port** (*bool*) – When module 'A' instantiates the module to which this port is attached, should this port be connected to a similar port on 'A'.
- **parent_sig** (*bool*) – When module 'A' instantiates the module to which this port is attached, should 'A' also instantiate a signal matching the one connected to this port.
- **kwargs** – Other keywords which should become attributes of this instance.

**class** verilog.**Signal**(*name*, *signal="*, *width=0*, ***kwargs*)
A simple class to hold signal attributes. It is immutable, and will throw an error if its attributes are changed after being set.

**__init__**(*name*, *signal="*, *width=0*, ***kwargs*)
Create a 'Signal' instance.

**Parameters**

- **name** (*str*) – Name of this signal
- **signal** (*int*) – Name of this signal
- **width** – Bitwidth of this signal
- **kwargs** – Other keywords which should become attributes of this instance.

**update_attrs**(*name*, *width=0*, ***kwargs*)

**class** verilog.**VerilogModule**(*name=”*, *topfile=None*, *comment=”*)

A Python object which knows how to represent itself in Verilog.

**__init__**(*name=”*, *topfile=None*, *comment=”*)

Construct a new module, named `name`. You can either start with an empty module and add ports/signals/instances to it, or you can specify an existing top-level file topfile, which will be modified. If doing the latter, the construction of wishbone interconnect demands that the topfile has a `localparam N_WB_SLAVES`, which specifies the number of wishbone slaves in the un-modified topfile. And `SLAVE_BASE` and `SLAVE_HIGH` localparams definiting the slave addresses.

Eg:

```
localparam N_WB_SLAVES = 2;

localparam SLAVE_BASE = {
32'h00010000, // slave_1
32'h00000000  // slave_0
};

localparam SLAVE_HIGH = {
32'h00010003, // slave_1
32'hFFFFFFFF  // slave_0
};

// This module will only tolerate
// i/o declarations like:

module top (
    input sysclk_n,
    input sysclk_p,
    ...
    );

// I.e, NOT

module top(
    sysclk_n,
    sysclk_p,
    ...
    );
    input sysclk_n;
    input sysclk_p;
    ...

// YMMV if your topfile doesn't use linebreaks as
// shown above. I.e., for best chance of success don't do

module top( sysclk_n,
sysclk_p);

localparam SLAVE_BASE = {32'h00000000};
```

**Parameters**

- **name** (*str*) – Name of this module
- **topfile** (*str or None*) – The filename of an existing verilog file, if any, to which this module should add.

- **comment** (*str*) – A user-friendly comment to be inserted in Verilog where this module is instantiated.

**add_axi4lite_interface**(*regname*, *mode*, *nbytes=4*, *default_val=0*, *suffix=”*, *candr_suffix=”*, *memory_map=[]*, *typecode=255*, *data_width=32*)

Add the ports necessary for a AXI4-Lite slave interface.

This function returns the AXI4LiteDevice object, so the caller can mess with it’s memory map if they so desire.

Added the (optional) data_width parameter to make provision for variable-size BRAMs

**add_localparam**(*name*, *value*, *comment=None*)

Add a parameter to the entity, with name `parameter` and value `value`.

You may add a comment that will end up in the generated verilog.

**add_parameter**(*name*, *value*, *comment=None*)

Add a parameter to the entity, with name `parameter` and value `value`.

You may add a comment that will end up in the generated verilog.

**add_port**(*name*, *signal=None*, *parent_port=False*, *parent_sig=True*, ***kwargs*)

Add a port to the module. Only the parameter `name` is compulsory. Others may be required when instantiating this module in another.

E.g., an instance of this module needs all ports to have a defined `signal` value.

However, if this module is at the top level, this isn’t necessary. Similarly, a port featuring in an instantiated module need not have a width or direction specified, but if you want to instantiate the module and propagate the port to the parent, the parent won’t know what to do unless these port parameters are specified.

**Parameters**

- **name** – name of the port
- **signal** – name of the signal to connect port to. Can include bit indexing, e.g. `my_signal[15:8]`
- **dir** – direction of signal
- **width** – width of signal
- **parent_port** – When instantiating this module, promote this port to a port of the parent
- **parent_sig** – When instantiating this module, add a signal named `signal` to the parent
- **comment** – Use this to add a comment string which will end up in the generated verilog

**add_raw_string**(*s*)

**add_signal**(*name*, *width=0*, ***kwargs*)

Add an internal signal to the entity, with name `signal` and width `width`.

You may add a comment that will end up in the generated verilog.

**add_sourcefile**(*file*)

**add_wb_interface**(*regname*, *mode*, *nbytes=4*, *suffix=”*, *candr_suffix=”*, *memory_map=[]*, *typecode=255*)

Add the ports necessary for a wishbone slave interface. Wishbone ports that depend on the slave index are identified by a parameter that matches the instance name. This parameter must be given a value in a higher level of the verilog code!

This function returns the WbDevice object, so the caller can mess with it’s memory map if they so desire.

**assign_signal**(*lhs*, *rhs*, *comment=None*)
Assign one signal to another, or one signal to a port.

i.e., generate lines of verilog like: `assign lhs = rhs;`

`lhs` and `rhs` are strings that can represent port or signal names, and may include verilog-style indexing, eg `[15:8]`

You may add a comment that will end up in the generated verilog.

**assign_wb_interface**(*name*, *id=0*, *suffix=”*, *candr_suffix=”*, *sub_arb_id=0*)
Add the ports necessary for a wishbone slave interface. Wishbone ports that depend on the slave index are identified by a parameter that matches the instance name. This parameter must be given a value in a higher level of the verilog code!

**axi4lite_memory_map**(*base_addr=65536*, *alignment=4*)
This function is only to be called by the ‘top’ verilog module after all other yellow blocks have called ‘modify_top’, but before the axi4lite_interconnect yellow block class has called ‘modify_top’ as that class requires the memory map this creates.

**Parameters**

- **base_addr** (*int*) – The address from which indexing of instance axi4lite interfaces will begin. Any memory space required by the template verilog file should be below this address.
- **alignment** (*int*) – Alignment required by all memory start addresses.

memory map: keys: name of AXI4-Lite interfaces. values:

- ‘memory_map’: internal memory map for this interface
- ‘size’: size of internal memory map in bytes
- ‘absolute_address’: actual address in memory determined by base_addr
- ‘relative_address’: address relative to base_addr
- ‘axi4lite_devices’: List of AXI4LiteDevice objects for core_info backwards compatibility

**gen_assignments_ascii_art**()

**Returns** Pretty ascii art “Assignments” string.

**gen_assignments_str**()
Generate the verilog code required to assign a port or signal to another signal

**gen_cur_blk_comment**(*cur_blk*, *dict*)
This helper function returns the current block string, if the dictionary is not empty and the current block is not `default`.

**gen_default_nettype_str**()

**gen_endmod_str**()

**gen_instance_verilog**(*instname*)
Generate a string corresponding to the instantiation of this instance, with instance name `instname`

**gen_instances_ascii_art**()

**Returns** Pretty ascii art “Instances” string.

**gen_instances_dec_str**()
Generate the verilog code required to instantiate the instances in this module

**gen_localparams_dec_str**()
Generate the verilog code required to declare localparams

**gen_mod_dec_str**()
Generate the verilog code required to start a module declaration.

**gen_module_file**(*filename=None*)

**gen_params_dec_str**()
Generate the verilog code required to declare parameters

**gen_port_list**()
Generate the verilog code required to declare ports

**gen_ports_dec_str**()
Generate the verilog code required to declare ports with special attributes, eg LOCS, etc.

**gen_signals_ascii_art**()

**Returns** Pretty ascii art "Signals" string.

**gen_signals_dec_str**()
Generate the verilog code required to declare signals

**gen_top_mod**()
Return the code that needs to go in a top level verilog file to incorporate this module.

I.e., everything except the module port declaration headers and endmodule lines.

TODO: This is almost identical to write_new_module_file(). Combine?

**get_base_wb_slaves**()
Look for the pattern `localparam N_WB_SLAVES` in this module's topfile, and use it to extract the number of wishbone slaves in the module. Update the base_wb_slaves attribute accordingly. Also extract the addresses. Names are auto-generated

**get_instance**(*entity*, *name*, *comment=None*)
Instantiate and return a new instance of entity `entity`, with instance name `name`.

You may add a comment that will end up in the generated verilog.

**has_instance**(*name*)
Check if this module has an instance called <name>. If so return True

**instantiate_child_ports**()
Add ports and signals associated with child instances

**rewrite_module_file**(*filename=None*)
Rewrite the intially supplied verilog file to include instance, signals, ports, assignments and wishbone interfaces added programmatically.

The initial verilog file is backed up with a '.base' extension.

**search_dict_for_name**(*dict*, *name*)
This helper function searches each top level dictionary to see if it contains `name` and returns the key that does.

**set_cur_blk**(*cur_blk*)
Set the name of the block currently driving code generation. This is useful for grouping and commenting the ports / instances / signals associated with particular instances, so that the output Verilog is prettier.

**Parameters** **cur_blk** (*str*) – The name of the current block driving code generation.

**wb_compute**(*base_addr=65536*, *alignment=4*)

Compute the appropriate wishbone address limits, based on the current wishbone-using instances instantiated in the module.

Will NOT take into account wishbone memory space used by the template verilog file (but see base_addr, below)

**Parameters**

- **base_addr** (*int*) – The address from which indexing of instance wishbone interfaces will begin. Any memory space required by the template verilog file should be below this address.
- **alignment** (*int*) – Alignment required by all memory start addresses.

**write_new_module_file**(*filename=None*)

Write a verilog file from scratch, based on the programmatic additions of instances / signals / etc. to the VerilogModule instance.

The jasper toolflow has been using `rewrite_module_file()` rather than this method, so it may or may not still work correctly. It used to, at least…

**class** verilog.**WbDevice**(*regname*, *nbytes*, *mode*, *hdl_suffix=''*, *hdl_candr_suffix=''*, *memory_map=[]*, *typecode=255*)

A class to encapsulate the parameters (name, size, etc.) of a wishbone slave device.

**__init__**(*regname*, *nbytes*, *mode*, *hdl_suffix=''*, *hdl_candr_suffix=''*, *memory_map=[]*, *typecode=255*)

Class constructor.

**Parameters**

- **regname** (*str*) – Name of register (this name is the string used to access the register from software)
- **nbytes** (*int*) – Number of bytes in this slave's memory space.
- **mode** (*str*) – Permissions ('r': readable, 'w': writable, 'rw': read/writeable)
- **hdl_suffix** (*str*) – Suffix given to wishbone port names. Eg. if `hdl_suffix = foo`, ports have the form `wbs_dat_i_foo`
- **hdl_candr_suffix** (*str*) – Suffix given to wishbone clock and reset port names. Eg. if `hdl_suffix = foo`, ports have the form `wbs_clk_i_foo`
- **memory_map** (*list*) – A list or `Register` instances defining the contents of sub-blocks of this device's memory.
- **typecode** (*int*) – Typecode number (0-255) identifying the type of this block. See `yellow_block_typecodes.py`

**base_addr = None**

Start (lowest) address of the memory space used by this device, in bytes.

**high_addr = None**

End (highest) address of the memory space used by this device, in bytes.

**sub_arb_id = None**

If using multiple bus arbiters, which arbiter should this slave attach to?

verilog.**gen_wbs_master_arbiter**(*arbiters*, *max_devices_per_arb=32*)

Deliver a string defining the top level of a hierarchical Wishbone arbiter. This can be written to a file and then imported into an HDL project. Ideally (maybe) this instantiation would be made via a VerilogModule class.

`verilog.`**`instantiate_wb_arb_module`**`(`*module*, *n_slaves*, *n_sub_arbs=None*`)`

Instantiate a Wishbone Arbiter into a module.

**Parameters**

- **`module`** (*`VerilogModule instance`*) – Module into which the arbiter should be instantiated.
- **`n_slaves`** (*`int`*) – Number of slaves this arbiter is connected to.
- **`n_sub_arbs`** (*`int or None`*) – Number of sub-arbiters beneath the arbiter being instantiated here. If None, a non-hierarchical arbiter will be used.

## yellow_blocks

### adc

**`class`** `yellow_blocks.adc.`**`adc`**`(`*blk*, *platform*, *hdl_root=None*`)`

**`gen_constraints`**`()`

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**`()`

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**`(`*top*`)`

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### adc16

**`class`** `yellow_blocks.adc16.`**`adc16`**`(`*blk*, *platform*, *hdl_root=None*`)`

**`gen_constraints`**`()`

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**`()`

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### adc20g

**class** `yellow_blocks.adc20g.`**adc20g**(*blk*, *platform*, *hdl_root=None*)

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**initialize**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### adc5g

**class** `yellow_blocks.adc5g.`**adc5g**(*blk*, *platform*, *hdl_root=None*)

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**initialize**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### bram

**class** `yellow_blocks.bram.`**bram**(*blk*, *platform*, *hdl_root=None*)

**initialize**()

This function is called by YellowBlocks __init__ method. We could override __init__ here, but this seems a little bit more user friendly.

**modify_top**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### clock_passthrough

**class** `yellow_blocks.clock_passthrough.`**clock_passthrough**(*blk*, *platform*, *hdl_root=None*)

**gen_constraints**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**initialize**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**modify_top**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### dcp

**class** `yellow_blocks.dcp.`**dcp**(*blk*, *platform*, *hdl_root=None*)

**initialize**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

## forty_gbe

**class** yellow_blocks.forty_gbe.**forty_gbe**(*blk*, *platform*, *hdl_root=None*)

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**gen_tcl_cmds**()
Generate a dictionary of tcl command lists, to be executed at compile time. Allowed keys are: `init`, `pre_synth`, `synth`, `post_synth`, `pre_impl`, `impl`, `post_impl`, `pre_bitgen`, `bitgen`, `post_bitgen`, `prom_gem`. The key used determines at what stage the tcl commands will be run.

Eg.:

```
{
    'pre_synth': ["first pre-synthesis tcl command", "second pre-synthesis␣
↪tcl command"],
    'prom_gen' : ["A tcl command to generate a prom file after bit gen"],
}
```

**Returns** Dictionary of tcl command lists. Default {}

**initialize**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**instantiate_fgbe**(*top*, *num=None*)

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## gpio

**class** yellow_blocks.gpio.**gpio**(*blk*, *platform*, *hdl_root=None*)

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**initialize**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters `top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### gpio_bidir

**class** `yellow_blocks.gpio_bidir.`**`gpio_bidir`**(*blk*, *platform*, *hdl_root=None*)

**`gen_constraints`**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters `top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### hmc

**class** `yellow_blocks.hmc.`**`hmc`**(*blk*, *platform*, *hdl_root=None*)

**`gen_constraints`**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`instantiate_hmcc`**(*top*, *num=None*)

**`modify_top`**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### i2c_interface

**class** `yellow_blocks.i2c_interface.`**`i2c_interface`**(*blk*, *platform*, *hdl_root=None*)

**`gen_constraints`**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()

This function is called by YellowBlocks __init__ method. We could override __init__ here, but this seems a little bit more user friendly.

**`modify_top`**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### ip

**class** `yellow_blocks.ip.`**`ip`**(*blk*, *platform*, *hdl_root=None*)

**`initialize`**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

### lmx2581

**class** `yellow_blocks.lmx2581.`**`lmx2581`**(*blk*, *platform*, *hdl_root=None*)

**`gen_constraints`**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## microblaze

**class** `yellow_blocks.microblaze.`**microblaze**(*blk*, *platform*, *hdl_root=None*)

**static factory**(*blk*, *plat*, *hdl_root=None*)

**initialize**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**class** `yellow_blocks.microblaze.`**microblaze_k7**(*blk*, *platform*, *hdl_root=None*)

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**gen_tcl_cmds**()
Generate a dictionary of tcl command lists, to be executed at compile time. Allowed keys are: `init`, `pre_synth`, `synth`, `post_synth`, `pre_impl`, `impl`, `post_impl`, `pre_bitgen`, `bitgen`, `post_bitgen`, `prom_gem`. The key used determines at what stage the tcl commands will be run.

Eg.:

```
{
    'pre_synth': ["first pre-synthesis tcl command", "second pre-synthesis␣
↪tcl command"],
    'prom_gen' : ["A tcl command to generate a prom file after bit gen"],
}
```

**Returns** Dictionary of tcl command lists. Default {}

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

**class** `yellow_blocks.microblaze.`**microblaze_ku7**(*blk*, *platform*, *hdl_root=None*)

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**gen_tcl_cmds**()
Generate a dictionary of tcl command lists, to be executed at compile time. Allowed keys are: `init`, `pre_synth`, `synth`, `post_synth`, `pre_impl`, `impl`, `post_impl`, `pre_bitgen`, `bitgen`, `post_bitgen`, `prom_gem`. The key used determines at what stage the tcl commands will be run.

Eg.:

```
{
    'pre_synth': ["first pre-synthesis tcl command", "second pre-synthesis␣
↪tcl command"],
    'prom_gen' : ["A tcl command to generate a prom file after bit gen"],
}
```

**Returns** Dictionary of tcl command lists. Default {}

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## onegbe

**class** yellow_blocks.onegbe.**onegbe**(*blk*, *platform*, *hdl_root=None*)

**static factory**(*blk*, *plat*, *hdl_root=None*)

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

**class** yellow_blocks.onegbe.**onegbe_casia_k7**(*blk*, *platform*, *hdl_root=None*)

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**initialize**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

**class** `yellow_blocks.onegbe.`**`onegbe_snap`**(*blk*, *platform*, *hdl_root=None*)

**`gen_constraints`**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

**class** `yellow_blocks.onegbe.`**`onegbe_vcu118`**(*blk*, *platform*, *hdl_root=None*)

**`gen_constraints`**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

**class** `yellow_blocks.onegbe.`**`onegbe_vcu128`**(*blk*, *platform*, *hdl_root=None*)

**`gen_constraints`**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## skarab

**class** `yellow_blocks.skarab.`**`skarab`**(*blk*, *platform*, *hdl_root=None*)

**`gen_children`**()
The toolflow will try to allow blocks to instantiate other blocks themselves, by calling this method. Override it in your subclass if you need to use this functionality.

**Returns** A list of child YellowBlock instances

**`gen_constraints`**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## snap

**class** `yellow_blocks.snap.`**`snap`**(*blk*, *platform*, *hdl_root=None*)

**`gen_children`**()
The toolflow will try to allow blocks to instantiate other blocks themselves, by calling this method. Override it in your subclass if you need to use this functionality.

**Returns** A list of child YellowBlock instances

**`gen_constraints`**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**gen_tcl_cmds**()
Generate a dictionary of tcl command lists, to be executed at compile time. Allowed keys are: `init`, `pre_synth`, `synth`, `post_synth`, `pre_impl`, `impl`, `post_impl`, `pre_bitgen`, `bitgen`, `post_bitgen`, `prom_gem`. The key used determines at what stage the tcl commands will be run.

Eg.:

```
{
    'pre_synth': ["first pre-synthesis tcl command", "second pre-synthesis␣
↪tcl command"],
    'prom_gen' : ["A tcl command to generate a prom file after bit gen"],
}
```

**Returns** Dictionary of tcl command lists. Default {}

**initialize**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## snap2

**class** yellow_blocks.snap2.**snap2**(*blk*, *platform*, *hdl_root=None*)

**gen_children**()
The toolflow will try to allow blocks to instantiate other blocks themselves, by calling this method. Override it in your subclass if you need to use this functionality.

**Returns** A list of child YellowBlock instances

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**gen_tcl_cmds**()
Generate a dictionary of tcl command lists, to be executed at compile time. Allowed keys are: `init`, `pre_synth`, `synth`, `post_synth`, `pre_impl`, `impl`, `post_impl`, `pre_bitgen`, `bitgen`, `post_bitgen`, `prom_gem`. The key used determines at what stage the tcl commands will be run.

Eg.:

```
{
    'pre_synth': ["first pre-synthesis tcl command", "second pre-synthesis␣
↪tcl command"],
    'prom_gen' : ["A tcl command to generate a prom file after bit gen"],
}
```

**Returns** Dictionary of tcl command lists. Default {}

**`initialize`**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## snap_adc

**class** `yellow_blocks.snap_adc.`**`snap_adc`**(*blk*, *platform*, *hdl_root=None*)

**`gen_children`**()
The first instance of this adc adds the required clock controller module

**`gen_constraints`**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

**`reorder_ports`**(*port_list*, *wb_bitwidth=32*)
Reorder output ports of ADCs to arrange sampling data in correct order in wb_bram

wb_bitwidth stands for the bit width of data in/out port of wishbone bus

```
reorder_ports(['a1','a2','a3','a4'])
when self.adc_data_width == 8, return {a1,a2,a3,a4}
when self.adc_data_width == 16, return {a3,a4,a1,a2}
when self.adc_data_width == 32, return {a4,a3,a2,a1}
```

### spi_wb_bridge

**class** `yellow_blocks.spi_wb_bridge.`**`spi_wb_bridge`**(*blk*, *platform*, *hdl_root=None*)

**`gen_constraints`**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### sw_reg

**class** `yellow_blocks.sw_reg.`**`sw_reg`**(*blk*, *platform*, *hdl_root=None*)

**`initialize`**()

This function is called by YellowBlocks __init__ method. We could override __init__ here, but this seems a little bit more user friendly.

**`modify_top`**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### sw_reg_sync

**class** `yellow_blocks.sw_reg_sync.`**`sw_reg_sync`**(*blk*, *platform*, *hdl_root=None*)

**`initialize`**()

This function is called by YellowBlocks __init__ method. We could override __init__ here, but this seems a little bit more user friendly.

**`modify_top`**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## sys_block

**class** `yellow_blocks.sys_block.`**`sys_block`**(*blk*, *platform*, *hdl_root=None*)

**`initialize`**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## ten_gbe

**class** `yellow_blocks.ten_gbe.`**`ten_gbe`**(*blk*, *platform*, *hdl_root=None*)

**static factory**(*blk*, *plat*, *hdl_root=None*)

**`instantiate_ktge`**(*top*, *num=None*)

**class** `yellow_blocks.ten_gbe.`**`tengbaser_xilinx_k7`**(*blk*, *plat*, *hdl_root*, *use_gth=False*)

**`__init__`**(*blk*, *plat*, *hdl_root*, *use_gth=False*)

Class constructor. Set up the initial values for block attributes, by copying key/val pairs from the `blk` dictionary. Call the class's `initialize()` method, where the user should set compile parameters and override this class's default attributes. Finally, call the class's `check_support()` method, to verify that the block and platform chosen are compatible.

**Parameters**

- **`blk`** – A jasper-standard dictionary containing block information. Key/value pairs in this dictionary are copied to attributes of this instance.
- **`platform`** – A Platform object representing the platform type.
- **`hdl_root`** (*`Optional`*) – The path to a directory containing all hdl code necessary to instantiate this block. This root directory is used as a base from which block's source files are defined. If None (default), will default to the system's *HDL_ROOT* environment variable.

**`gen_children`**()

The mx175 clocks the gth from a clock which is passed through the FPGA and through a jitter cleaner (si5324) back into the GTH clock port. The first ten gig core needs to make sure this pass through is instantiated.

**`gen_constraints`**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**initialize**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**instantiate_infra**(*top*, *num*)

**instantiate_phy**(*top*, *num*)

**modify_top**(*top*)

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

**class** yellow_blocks.ten_gbe.**tengbaser_xilinx_ku7**(*blk*, *plat*, *hdl_root*, *use_gth=False*)

**__init__**(*blk*, *plat*, *hdl_root*, *use_gth=False*)

Class constructor. Set up the initial values for block attributes, by copying key/val pairs from the `blk` dictionary. Call the class's `initialize()` method, where the user should set compile parameters and override this class's default attributes. Finally, call the class's `check_support()` method, to verify that the block and platform chosen are compatible.

**Parameters**

- **blk** – A jasper-standard dictionary containing block information. Key/value pairs in this dictionary are copied to attributes of this instance.
- **platform** – A Platform object representing the platform type.
- **hdl_root** (*Optional*) – The path to a directory containing all hdl code necessary to instantiate this block. This root directory is used as a base from which block's source files are defined. If None (default), will default to the system's *HDL_ROOT* environment variable.

**gen_children**()

The mx175 clocks the gth from a clock which is passed through the FPGA and through a jitter cleaner (si5324) back into the GTH clock port. The first ten gig core needs to make sure this pass through is instantiated.

**gen_constraints**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**initialize**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**instantiate_infra**(*top*, *num*)

**instantiate_phy**(*top*, *num*)

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

**class** yellow_blocks.ten_gbe.**tengbe_v2_xilinx_v6**(*blk*, *platform*, *hdl_root=None*)

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**initialize**()
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### vcu118

**class** yellow_blocks.vcu118.**vcu118**(*blk*, *platform*, *hdl_root=None*)

**gen_children**()
The toolflow will try to allow blocks to instantiate other blocks themselves, by calling this method. Override it in your subclass if you need to use this functionality.

**Returns** A list of child YellowBlock instances

**gen_constraints**()
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**gen_tcl_cmds**()
Generate a dictionary of tcl command lists, to be executed at compile time. Allowed keys are: `init`, `pre_synth`, `synth`, `post_synth`, `pre_impl`, `impl`, `post_impl`, `pre_bitgen`, `bitgen`, `post_bitgen`, `prom_gem`. The key used determines at what stage the tcl commands will be run.

Eg.:

```
{
    'pre_synth': ["first pre-synthesis tcl command", "second pre-synthesis␣
↪tcl command"],
    'prom_gen' : ["A tcl command to generate a prom file after bit gen"],
}
```

**Returns** Dictionary of tcl command lists. Default {}

**`initialize`**`()`
This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**`modify_top`**`(`*top*`)`
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### xadc

**`class`** `yellow_blocks.xadc.`**`xadc`**`(`*blk*, *platform*, *hdl_root=None*`)`

**`gen_constraints`**`()`
Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**`initialize`**`()`
This function is called by YellowBlocks __init__ method. We could override __init__ here, but this seems a little bit more user friendly.

**`modify_top`**`(`*top*`)`
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

### xsg

**`class`** `yellow_blocks.xsg.`**`xsg`**`(`*blk*, *platform*, *hdl_root=None*`)`
An xsg YellowBlock class, representing a CASPER "platform" block. I.e., the block which specifies which platform you are compiling for, and which clock (or other) compile-time settings you want to use.

Uses 2 attributes from the front end block configuration (in addition to the "platform" attribute which all YellowBlocks have access to): "clk_rate" (float): The clock rate, in MHz, the compile is targeting for the DSP pipeline. "clk_src" (string): The clock source (eg. "sys_clk", "adc0_clk", etc.) used to run the DSP pipeline.

This block is a little unlike other YellowBlocks – it has 3 jobs:

1) Add a requirement – "self.requires.append(...)" – for a block in the design to provide – "self.provides.append(...)" – four clock phases:

   self.clk_src self.clk_src + "90" self.clk_src + "180" self.clk_src + "270"

   Probably, these clocks will be provided by the platform-specific block instantiated as a child of this block (see 3. below). Whichever block provides them must create signals with these names in the top-level verilog.

2) Create wires in the top-level verilog desing, and assign the above clock signals to the new names: "user_clk" "user_clk90" "user_clk180" "user_clk270" This allows modules in the design to use these clock names, without requiring any knowledge about where they are coming from (eg. sys_clk, an ADC clock, etc). NOTE: Since this block instantiates clocks with these names, you MUST NOT use user_clkX signals elsewhere in your design.

3) Instantiate a child YellowBlock, with identical parameters to this block, but with the class name self.platform.name. This is probably the instance you want to use to generate your custom clocks (eg, sys_clk, sys_clk90, . . . ), so this block should add these signals to top-level verilog, and also "provide" them using the YellowBlock.provides mechanism.

**`gen_children`**`()`

The toolflow will try to allow blocks to instantiate other blocks themselves, by calling this method. Override it in your subclass if you need to use this functionality.

**Returns** A list of child YellowBlock instances

**`initialize`**`()`

Things the toolflow has to know. eg, clocks needed/provided

**`modify_top`**`(`*top*`)`

Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **`top`** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

## yellow_block

**`class`** `yellow_blocks.yellow_block.`**`YellowBlock`**`(`*blk*, *platform*, *hdl_root=None*`)`

A yellow block object encodes all the information necessary to instantiate a piece of IP in an existing HDL base package.

- which verilog modules need to be instantiated.
- which instances need to be connected by signals
- which ports of the instance need to be promoted to top-level
- what is the type of these ports (for the constraints file)
- is the device a slave on a CPU bus
- if so, how much address space does it need?
- what features does this block provide the rest of the system, e.g. clock sources
- what fixed resources does this block use (e.g. QDR chip / ZDOK interface)

All the HDL related stuff is dealt with by the verilog module class, so we just need to add bus / memory space requirements and define what resources the block uses and provisions.

**`__init__`**`(`*blk*, *platform*, *hdl_root=None*`)`

Class constructor. Set up the initial values for block attributes, by copying key/val pairs from the `blk` dictionary. Call the class's `initialize()` method, where the user should set compile parameters and override this class's default attributes. Finally, call the class's `check_support()` method, to verify that the block and platform chosen are compatible.

**Parameters**

- **`blk`** – A jasper-standard dictionary containing block information. Key/value pairs in this dictionary are copied to attributes of this instance.

- **`platform`** – A Platform object representing the platform type.
- **`hdl_root`** (*`Optional`*) – The path to a directory containing all hdl code necessary to instantiate this block. This root directory is used as a base from which block's source files are defined. If None (default), will default to the system's *HDL_ROOT* environment variable.

**`add_build_dir_source`**()

This function is neccessary as yellow blocks dont have access to the build directory when they want to add a source file that is not in hdl_lib this function can be used. Generate a list of dictionaries containing files/directories relative to the build_dir, which will be added to the sources of the project. to the project.

Eg.: []

{'files': 'xml2vhdl_hdl_output/', – this can be a directory or a file 'library' : 'work'} – this is only used if the file needs to be included under a library (vhdl only) for verilog use ''

]

**Returns** Dictionary of tcl command lists. Default {}

**`add_source`**(*path*)

Add a source file to the list of files required to compile this yellow block. The path given should be relative to the root directory `hdl_root`. Globbing is supported.

**Parameters `path`** – Path of file required for compilation. Eg "/some/source/file.v" or "/some/files*.v"

**`blk = None`**

Stores the *blk* parameter, passed into this block's constructor.

**`check_support`**()

Check the platform being used is supported by this block. Relies on subclasses to set the `platform_support` attribute appropriately in their `initialize()` methods. The default of the YellowBlock class is `platform_support = 'all'`.

Throw an error if the platform appears unsupported.

**`copy_attrs`**()

Grab the dictionary entries of self.blk and turn them into attributes of this YellowBlock instance.

**`drc`**()

Perform block-specific design rule checks. This method should be overridden by subclasses if any custom design checks are required.

**`exc_requires = None`**

"Exclusive requirements". A list of strings, e.g. "zdok0", "sfp2", detailing a resources this block needs in order to compile. If another block tries to require the same resource, the compile will fail error checking.

**`gen_children`**()

The toolflow will try to allow blocks to instantiate other blocks themselves, by calling this method. Override it in your subclass if you need to use this functionality.

**Returns** A list of child YellowBlock instances

**`gen_constraints`**()

Generate a list of Constraint objects, appropriate for this block. This method should be over-ridden by sub-classes to return a list of constraints as defined in `constraints.py`

**Returns** A list of Constraint instances. Default is []

**gen_custom_hdl**()

Generate a dictionary of custom hdl, to be saved as a file and added to the sources of the generated project. The key is the file name and the value is a string of HDL code to save in to that file. Eg.: {

'my_hdl.vhdl': ["<HDL code>"], 'my_2nd_hdl.vhdl' : ["<More HDL code>"],

}

**Returns** Dictionary of hdl files. Default {}

**gen_tcl_cmds**()

Generate a dictionary of tcl command lists, to be executed at compile time. Allowed keys are: `init`, `pre_synth`, `synth`, `post_synth`, `pre_impl`, `impl`, `post_impl`, `pre_bitgen`, `bitgen`, `post_bitgen`, `prom_gem`. The key used determines at what stage the tcl commands will be run.

Eg.:

```
{
    'pre_synth': ["first pre-synthesis tcl command", "second pre-synthesis␣
↪tcl command"],
    'prom_gen' : ["A tcl command to generate a prom file after bit gen"],
}
```

**Returns** Dictionary of tcl command lists. Default {}

**hdl_root = None**

The base directory from which source file's locations are specified

**i_am_the_first = None**

A boolean, which is *True* if *self.inst_id == 0*

**initialize**()

This function is called by the `__init__()` method. It is meant to be overridden by subclasses.

It should over-ride instance attributes to configure the block.

Common attributes which might be manipulated are: `requires`, `exc_requires`, `provides`, `ips`, `sources`, `platform_supports`

**inst_id = None**

The ID of this block within all the instances of this block's class

**ips = None**

A list of IP dictionaries defining user-supplied IP to include with this block Dictionaries in this list have keys *path* (the path to the library) *name* (the name of the IP) *module_name* (the name of the HDL module this block defines) *vendor*, *library*, *version* (strings used by the backend to instantiate the IP)

**logger = None**

The *jasper.yellowblock* logger

**static make_block**(*blk*, *platform*, *hdl_root=None*)

A builder function to return an instance of the correct `YellowBlock` subclass for a given type of block and target platform.

**Parameters**

- **blk** – A jasper-standard dictionary containing block information
- **platform** – A Platform object representing the platform type.

**Optional keyword param hdl_root** The path to a directory containing all hdl code necessary to instantiate this block. This root directory is used as a base from which block's source files are defined.

**modify_top**(*top*)
Modify the VerilogModule instance top (so as to instantiate this module's HDL) This method should be overridden by subclasses implementing their custom HDL requirements.

**Parameters** **top** – A VerilogModule instance, defining the top-level of an HDL design into which this block should instantiate itself.

**name = None**
A friendly name for this block, generated from the *tag* entry in the *self.blk* dictionary and *self.inst_id*. Eg. "sw_reg5", or "ten_gbe0"

**platform = None**
Stores the *platform* parameter, passed into this block's constructor

**platform_support = None**
A list of platform names this block supports, or, the string "all", indicating the block is platform agnostic.

**provides = None**
A list of strings, eg. "zdok0", "sfp1", detailing a resource this block provides to the design. These will me matched against *self.requires* and *self.exc_requires* of all the blocks in the design to determine if the compile is viable.

**requires = None**
A list of strings, eg. "zdok0", "sfp1", detailing a resource this block needs to compile. To pass rule-checking, every entry here must be matched with an entry in *self.provides* of another block, or the target platform

**sources = None**
A list of source files (paths relative to *self.hdl_root*) required by this module

**throw_error**(*message*)
Raise an exception, showing the input message, but prefixing with a human-readable yellow block name.

**typecode = None**
A unique typecode indicating the type of yellow block this is. See *yellow_block_typecodes.py*. This code gets baked into a memory-map in the FPGA binary, and allows embedded software to figure out what type of devices are on the CPU bus.

### yellow_block_typecodes

Here lie device typecode definitions. These codes get baked into toolflow-generate firmware can be used by embedded software to determine what types of devices are present on the CPU bus at different memory locations.

For example, a typical use case would be:

1. MicroBlaze wakes up on programming a board
2. Microblaze looks for an Ethernet core in the running firmware by searching for something in the firmware's memory map with typecode TYPECODE_ETHCORE.
3. MicroBlaze manipulates this device so as to talk to the outside world.

Values chosen are non-critical, but should be unique, and <256.

## 4.3 Get Involved

If you are a CASPER collaborator, or you're just interested in what we're up to, feel free to join our mailing list by sending a blank email here.

If would like to get involved in the development of the tools, please join our dev mailing list by sending a blank email here..

# Python Module Index

# Index