# Project X-Ray Documentation

***Release 0.0-3297-gf9dda426***

**SymbiFlow Team**

**Dec 01, 2020**

# INTRODUCTION

Project X-Ray documents the Xilinx 7-Series FPGA architecture to enable development of open-source tools. Our goal is to provide sufficient information to develop a free and open Verilog to bitstream toolchain for these devices.

CHAPTER

ONE

# INTRODUCTION

Project X-Ray documents the Xilinx 7-Series FPGA architecture to enable the development of open-source tools. Our goal is to provide sufficient information to develop a free and open Verilog to bitstream toolchain for these devices.

The project is a part of SymbiFlow Toolchain. SymbiFlow uses the obtained information about the chip in Architecture Definitions project, which allows for creating bitstreams for many architectures including 7-Series devices.

## 1.1 Collected information

To allow the usage of Xilinx FPGAs in SymbiFlow toolchain we collect some important data about the Xilinx chips. The needed information includes:

- Architecture description:
  - chip internals
  - timings
- Bitstream format:
  - metadata (i.e. header, crc)
  - configuration bits

Final results are stored in the database which is further used by the Architecture Definitions project. The whole database is described in the dedicated *chapter*.

## 1.2 Methodology

The most important element of the project are fuzzers - scripts responsible for obtaining information about the chips. Their name comes from the fact that they use a similar idea to Fuzz testing. Firstly, they generate a huge amount of designs in which the examined chip property is either enabled or disabled. By comparing the differences in the final bitstream obtained from vendor tools, we can detect relations between bits in the bitstream and provided functionalities.

However, some of the fuzzers works differently, i.e. they just creating the database structure, the whole idea is similar and rely on the output produced by the vendor tools.

All fuzzers are described in the dedicated *chapter*.

## 1.3 Important Parts

The important parts of the *Project X-Ray* are:

- *minitests* - designs that can be viewed by a human in Vivado to better understand how to generate more useful designs.
- *experiments* - similar to *minitests* except for the fact that they are only useful for a short time.
- *tools & libs* - they convert the resulting bitstreams into various formats.
- *utils* - tools that are used but still require some testing

CHAPTER

TWO

# GETTING STARTED

## 2.1 Project X-Ray

Documentation Status License Build Status Tests

Database Generation Artix 7 Database Kintex 7 Database Zynq 7 Database

Documenting the Xilinx 7-series bit-stream format.

This repository contains both tools and scripts which allow you to document the bit-stream format of Xilinx 7-series FPGAs.

More documentation can be found published on prjxray ReadTheDocs site - this includes;

- Highlevel Bitstream Architecture
- Overview of DB Development Process

## 2.2 Quickstart Guide

Instructions were originally written for Ubuntu 16.04. Please let us know if you have information on other distributions.

### 2.2.1 Step 1:

Install Vivado 2017.2. If you did not install to /opt/Xilinx default, then set the environment variable XRAY_VIVADO_SETTINGS to point to the settings64.sh file of the installed vivado version, ie

```
export XRAY_VIVADO_SETTINGS=/opt/Xilinx/Vivado/2017.2/settings64.sh
```

Do not source the settings64.sh in your shell, since this adds directories of the Vivado installation at the beginning of your PATH and LD_LIBRARY_PATH variables, which will likely interfere with or break non-Vivado applications in that shell. The Vivado wrapper utils/vivado.sh makes sure that the environment variables from XRAY_VIVADO_SETTINGS are automatically sourced in a separate shell that is then only used to run Vivado to avoid these problems.

**Why 2017.2?** Currently the fuzzers only work on `2017.2`, see Issue #14 on prjxray.

**Is 2017.2 really required?** Yes, only `2017.2` works. Until Issue #14 is solved, **only** `2017.2` works and will be supported.

### 2.2.2 Step 2:

Pull submodules:

```
git submodule update --init --recursive
```

### 2.2.3 Step 3:

Install CMake:

```
sudo apt-get install cmake # version 3.5.0 or later required,
                           # for Ubuntu Trusty pkg is called cmake3
```

### 2.2.4 Step 4:

Build the C++ tools:

```
make build
```

### 2.2.5 Step 5:

Choose one of the following options:

(Option 1) - Install the Python environment locally

```
sudo apt-get install virtualenv python3 python3-pip python3-virtualenv python3-yaml
make env
```

(Option 2) - Install the Python environment globally

```
sudo apt-get install python3 python3-pip python3-yaml
sudo -H pip3 install -r requirements.txt
```

This step is known to fail with a compiler error while building the `pyjson5` library when using Arch Linux and Fedora. If this occurs, `pyjson5` needs one change to build correctly:

```
git clone https://github.com/Kijewski/pyjson5.git
cd pyjson5
sed -i 's/char \*PyUnicode/const char \*PyUnicode/' src/_imports.pyx
sudo make
```

This might give you an error about `sphinx_autodoc_typehints` but it should correctly build and install pyjson5. After this, run either option 1 or 2 again.

### 2.2.6 Step 6:

Always make sure to set the environment for the device you are working on before running any other commands:

```
source settings/artix7.sh
```

### 2.2.7 Step 7:

(Option 1, recommended) - Download a current stable version (you can use the Python API with a pre-generated database)

```
./download-latest-db.sh
```

(Option 2) - (Re-)create the entire database (this will take a very long time!)

```
cd fuzzers
make -j$(nproc)
```

### 2.2.8 Step 8:

Pick a fuzzer (or write your own) and run:

```
cd fuzzers/010-clb-lutinit
make -j$(nproc) run
```

### 2.2.9 Step 9:

Create HTML documentation:

```
cd htmlgen
python3 htmlgen.py
```

## 2.3 C++ Development

Tests are not built by default. Setting the PRJXRAY_BUILD_TESTING option to ON when running cmake will include them:

```
cmake -DPRJXRAY_BUILD_TESTING=ON ..
make
```

The default C++ build configuration is for releases (optimizations enabled, no debug info). A build configuration for debugging (no optimizations, debug info) can be chosen via the CMAKE_BUILD_TYPE option:

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
make
```

The options to build tests and use a debug build configuration are independent to allow testing that optimizations do not cause bugs. The build configuration and build tests options may be combined to allow all permutations.

## 2.4 Process

The documentation is done through a "black box" process were Vivado is asked to generate a large number of designs which then used to create bitstreams. The resulting bit streams are then cross correlated to discover what different bits do.

### 2.4.1 Parts

#### Minitests

There are also "minitests" which are designs which can be viewed by a human in Vivado to better understand how to generate more useful designs.

#### Experiments

Experiments are like "minitests" except are only useful for a short period of time. Files are committed here to allow people to see how we are trying to understand the bitstream.

When an experiment is finished with, it will be moved from this directory into the latest "prjxray-experiments-archive-XXXX" repository.

#### Fuzzers

Fuzzers are the scripts which generate the large number of bitstream.

They are called "fuzzers" because they follow an approach similar to the idea of software testing through fuzzing.

#### Tools & Libs

Tools & libs are useful tools (and libraries) for converting the resulting bitstreams into various formats.

Binaries in the tools directory are considered more mature and stable then those in the utils directory and could be actively used in other projects.

#### Utils

Utils are various tools which are still highly experimental. These tools should only be used inside this repository.

#### Third Party

Third party contains code not developed as part of Project X-Ray.

## 2.5 Database

Running the all fuzzers in order will produce a database which documents the bitstream format in the database directory.

As running all these fuzzers can take significant time, Tim 'mithro' Ansell me@mith.ro has graciously agreed to maintain a copy of the database in the prjxray-db repository.

Please direct enquires to Tim if there are any issues with it.

## 2.6 Current Focus

Current the focus has been on the Artix-7 50T part. This structure is common between all footprints of the 15T, 35T and 50T varieties.

We have also started experimenting with the Kintex-7 parts.

The aim is to eventually document all parts in the Xilinx 7-series FPGAs but we can not do this alone, **we need your help**!

### 2.6.1 Adding a new part to an existing family

We have written a detailed guide that walks through the process of adding a new part to an existing family.

### 2.6.2 TODO List

- [ ] Write a TODO list

## 2.7 Contributing

There are a couple of guidelines when contributing to Project X-Ray which are listed here.

### 2.7.1 Sending

All contributions should be sent as GitHub Pull requests.

### 2.7.2 License

All software (code, associated documentation, support files, etc) in the Project X-Ray repository are licensed under the very permissive ISC Licence. A copy can be found in the `LICENSE` file.

All new contributions must also be released under this license.

### 2.7.3 Code of Conduct

By contributing you agree to the *code of conduct*. We follow the open source best practice of using the Contributor Covenant for our Code of Conduct.

### 2.7.4 Sign your work

To improve tracking of who did what, we follow the Linux Kernel's "sign your work" system. This is also called a "DCO" or "Developer's Certificate of Origin".

**All** commits are required to include this sign off and we use the Probot DCO App to check pull requests for this.

The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as a open-source patch. The rules are pretty simple: if you can certify the below:

```
    Developer's Certificate of Origin 1.1

    By making a contribution to this project, I certify that:

    (a) The contribution was created in whole or in part by me and I
        have the right to submit it under the open source license
        indicated in the file; or

    (b) The contribution is based upon previous work that, to the best
        of my knowledge, is covered under an appropriate open source
        license and I have the right under that license to submit that
        work with modifications, whether created in whole or in part
        by me, under the same open source license (unless I am
        permitted to submit under a different license), as indicated
        in the file; or

    (c) The contribution was provided directly to me by some other
        person who certified (a), (b) or (c) and I have not modified
        it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

then you just add a line saying

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

using your real name (sorry, no pseudonyms or anonymous contributions.)

You can add the signoff as part of your commit statement. For example:

```
git commit --signoff -a -m "Fixed some errors."
```

*Hint:* If you've forgotten to add a signoff to one or more commits, you can use the following command to add signoffs to all commits between you and the upstream master:

```
git rebase --signoff upstream/master
```

### 2.7.5 Contributing to the docs

In addition to the above contribution guidelines, see the guide to *updating the Project X-Ray docs*.

CHAPTER

THREE

# XILINX 7-SERIES ARCHITECTURE

## 3.1 Overview

**Todo:** add diagrams.

Xilinx 7-Series architecture utilizes a hierarchical design of chainable structures to scale across the Spartan, Artix, Kintex, and Virtex product lines. This documentation focuses on the Artix and Kintex devices and omits some concepts introduced in Virtex devices.

At the top-level, 7-Series devices are divided into two *halves* by a virtual horizontal line separating two sets of global clock buffers (BUFGs). While global clocks can be connected such that they span both sets of BUFGs, the two halves defined by this division are treated as separate entities as related to configuration. The halves are referred to simply as the top and bottom halves.

Each half is next divided vertically into one or more *horizontal clock rows*, numbered outward from the global clock buffer dividing line. Each horizontal clock row contains 12 clock lines that extend across the device perpendicular to the global clock spine. Similar to the global clock spine, each horizontal clock row is divided into two halves by two sets of horizontal clock buffers (BUFHs), one on each side of the global clock spine, yielding two *clock domains*. Horizontal clocks may be used within a single clock domain, connected to span both clock domains in a horizontal clock row, or connected to global clocks.

Clock domains have a fixed height of 50 *interconnect tiles* centered around the horizontal clock lines (25 above, 25 below). Various function tiles, such as *CLBs*, are attached to interconnect tiles.

## 3.2 Configuration

Within an FPGA, various memories (latches, block RAMs, distributed RAMs) contain the state of signal routing, *BEL* configuration, and runtime storage. Configuration is the process of loading an initial state into all of these memories both to define the intended logic operations as well as set initial data for runtime memories. Note that the same mechanisms used for configuration are also capable of reading out the active state of these memories as well. This can be used to examine the contents of a block RAM or other memory at any point in the device's operation.

### 3.2.1 Addressing

As described in *Overview*, 7-Series FPGAs are constructed out of *tiles* organized into *clock domains*. Each tile contains a set of *BELs* and the memories used to configure them. Uniquely addressing each of these memories involves first identifying the *horizontal clock row*, then the tile within that row, and finally the specific bit within the tile.

*Horizontal clock row* addressing follows the hierarchical structure described in *Overview* with a single bit used to indicate top or bottom half and a 5-bit integer to encode the row number. Within the row, tiles are connected to one or more configuration busses depending on the type of tile and what configuration memories it contains. These busses are identified by a 3-bit integer:

| Address | Name | Connected tile type |
|---|---|---|
| 000 | CLB, I/O, CLB | Interconnect (INT) |
| 001 | Block RAM content | Block RAM (BRAM) |
| 010 | CFG_CLB | ??? |

Within each bus, the connected tiles are organized into *columns*. A column roughly corresponds to a physical vertical line of tiles perpendicular to and centered over the horizontal clock row. Each column contains varying amounts of configuration data depending on the types of tiles attached to that column. Regardless of the amount, a column's configuration data is organized into a multiple of *frames*. Each frame consists of 101 words with 100 words for the connected tiles and 1 word for the horizontal clock row. The 7-bit address used to identify a specific frame within the column is called the minor address.

Putting all these pieces together, a 32-bit frame address is constructed:

| Field | Bits |
|---|---|
| Reserved | 31:26 |
| Bus | 25:23 |
| Top/Bottom Half | 22 |
| Row | 21:17 |
| Column | 16:7 |
| Minor | 6:0 |

#### CLB, I/O, CLB

Columns on this bus are comprised of 50 directly-attached interconnect tiles with various kinds of tiles connected behind them. Frames are striped across the interconnect tiles with each tile receiving 2 words out of the frame. The number of frames in a column depends on the type of tiles connected behind the interconnect. For example, interconnect tiles always have 26 frames and a CLBL tile has an additional 12 frames so a column of CLBs will have 36 frames.

#### Block RAM content

As the name says, this bus provides access to the *block RAM* contents. Block RAM configuration data is accessed via the CLB, I/O, CLB bus. The mapping of frame words to memory locations is not currently understood.

#### CFG_CLB

While mentioned in a few places, this bus type has not been seen in any bitstreams for Artix7 so far.

### 3.2.2 Loading sequence

**Todo:** Expand on these rough notes.

- Device is configured via a state machine controlled via a set of registers
- CRC of register writes is checked against expected values to verify data integrity during transmission.
- Before writing frame data:
  - IDCODE for configuration's target device is checked against actual device
  - Watchdog timer is disabled
  - Start-up sequence clock is selected and configured
  - Start-up signal assertion timing is configured
  - Interconnect is placed into Hi-Z state
- Data is then written by:
  - Loading a starting address
  - Selecting the write configuration command
  - Writing configuration data to data input register
    - Writes must be in multiples of the frame size
    - Multi-frame writes trigger autoincrementing of the frame address
    - Autoincrement can be disabled via bit in COR1 register.
    - At the end of a row, 2 frames of zeros must be inserted before data for the next row.
- After the write has finished, the device is restarted by:
  - Strobing a signal to activate IOB/CLB configuration flip-flops
  - Reactivate interconnect
  - Arms start-up sequence to run after desync
  - Desynchronizes the device from the configuration port
- Status register provides detail of start-up phases and which signals are asserted

### 3.2.3 Other

- ECC of frame data is contained in word 50 alongside horizontal clock row configuration
- Loading will succeed even with incorrect ECC data
- ECC is primarily used for runtime bit-flip detection

## 3.3 Bitstream format

FPGAs are configured with a binary file called the bitstream. The bitstream carries the information on which logical elements on the fabric should be configured and how in order to implement the target design. Moreover, it contains vital information on how to perform the configuration. The format of the bitstream is architecture specific, although the formats for devices of the same vendor often share a number of features. This chapter covers some details of the bitstream format for FPGA architectures which are currently supported in SymbiFlow's bitstream manipulation tools.

### 3.3.1 Xilinx

Xilinx provides a big variety of architectures (Spartan6, Series7, UltraScale, UltraScale+) that differ in the list of features and size. Despite these differences the bitstream format is pretty much alike and differs in some small details.

#### Bitstream header

The bitstream header contains various information about the origin and content of the entire bitstream, such as generation timestamp, name of target part or length of the configuration data. The header itself is ignored by the device as only the subsequent words take part in the configuration process. The presence of the bit header provides the distinction between .bin and .bit for HDL software like `Vivado`. The data in the header is stored in the Tag-Length-Value(TLV) format.

#### Synchronization word

Before any configuration packet is processed by the FPGA the configuration logic needs to find the synchronization word. The so called `Sync word` for Xilinx devices is `0xAA995566`. It is used to allow the configuration logic to align at a 32-bit word boundary and requires the bus width to be detected successfully for parallel configuration mode beforehand.

#### Bus Width Auto Detection

A specific byte pattern is inserted at the beginning of the bitstream file to allow the hardware to determine the width of the bus providing the configuration data. The configuration logic checks the lower eight bits of the parallel bus and depending on the received sequence the appropriate external bus width is chosen. The pattern is inserted before the 32-bit synchronization word and is ignored by the configuration state machine.

### Configuration Packets

All words that follow the synchronization word are interpreted by the configuration logic. Depending on the architecture the words are interpreted as 16 or 32 bit big-endian words and form the configuration packets. There are three types of packets identified by the header which can contain three major commands: NOP, READ, WRITE. While NOP is used for inserting required delays in the configuration sequence the most common are read and write operations. These supported packet types are:

- Type 0 - these packets exist only when performing zero-fill between rows
- Type 1 - used for read/writes of a number of words specified by the word count portion of the header which differs between architectures
- Type 2 - this type of packets expands the word count field that Type 1 packets have. For Series-7 FPGAs Type 2 expands to 27 bits by omitting the register address, but has to follow a Type 1 packet which carries the address information. Spartan6's Type 2 packets still contain the frame address, however the configuration logic expects the header to be followed by two 16-bit words with the MSB in the first word.

### Configuration Registers

The addresses that are specified in the configuration packets are mapped to a set of registers that provide low-level control over the chip. Only some of them take part in the programming sequence whereas the rest controls various physical aspects of the configuration interface. Some of the key registers used during programming are:

- IDCODE - Before writing to the configuration memory, a 32-bit device ID code must be written to this register. Reads from the register return the attached device's ID code.
- CRC - When a packet is received by the device, it automatically updates an internal CRC calculation to include the contents of that packet. A write to the CRC register checks that the calculated CRC matches the expected value written to the register. This CRC check is only used to provide integrity checking of the packet stream, not the configuration memory contents, and are not required for programming. If you are modifying a bitstream, CRC writes can simply be removed instead of recalculating them.
- Command - Most of the programming sequence is implemented as a state machine that is controlled via one-shot actions. Writes to this register arm an action that, depending on the action requested, may be triggered immediately or delayed until some other condition is met. During autoincremented frame writes, the current command is rewritten during every autoincrement. This has the effect of rearming the action on every frame written.
- Frame Address Register (FAR) - Writes to this register set the starting address for the next frame read or write.
- FDRI - When a frame is written to FDRI, the frame data is written to the configuration memory address specified by FAR. If the write to FDRI contains more than one frame, FAR is autoincremented at the end of each frame.

For more information on the available configuration registers refer to the configuration guides for Series-7 (table 5.23) and Spartan6 (table 5.30).

### Configuration Sequence

The basic steps for configuring a Series-7 and a Spartan6 device are the same. The first steps are responsible for the setup and include power-up, clearing the configuration memory and sampling mode pins. Further steps are related to loading the bitstream which is done in the following stages:

- synchronization
- check device ID
- load configuration data frames

- CRC check
- startup sequence

More details on the configuration sequence can be found in the Series-7 configuration guide (page 84)

### Example programming sequence

A high-level overview of a programming sequence for a Series-7 device is presented in the table below:

| Command | Data | Description |
|---|---|---|
| Write TIMER | 0x00000000 | Disable the watchdog timer |
| Write WBSTAR | 0x00000000 | On the next device boot, start with the bitstream at address zero. This may be different if the bitstream contains a multi-boot configuration. |
| Write COMMAND | 0x00000000 | Switch to the NULL command. |
| Write COMMAND | 0x00000007 | Reset the calculated CRC to zero. |
| Write register 0x13 | 0x00000000 | Undocumented register. No idea what this does yet. |
| Write COR0 | 0x02003fd5 | Setup timing of various device startup operations such as which startup cycle to wait in until MMCMs have locked and which clock settings to use during startup. |
| Write COR1 | 0x00000000 | Writing defaults to various device options such as the page size used to read from BPI and whether continuous configuration memory CRC calculation is enabled. |
| Write IDCODE | 0x0362d093 | Tell the device that this is a bitstream for a XC7A50T. If the device is an XC7A50T, configuration memory writes will be enabled. |
| Write COMMAND | 0x00000009 | Activate the clock configuration specified in Configuration Option Register 0. Up to this point, the device was using whatever clock configuration the last loaded bitstream used. |
| Write MASK | 0x00000401 | Set a bit-wise mask that is applied to subsequent writes to Control 0 and Control 1. This seems unnecessary for programming but is used to toggle certain bits in those registers instead of using precomputed values. It might make more sense in a use case where the exact value of Control 0 or Control 1 is unknown but a bit needs to be flipped. |
| Write Control 0 | 0x00000501 | Due to the previous write to MASK, 0x401 is actually written to this register which is the default value. Mostly disable fallback boot mode and masks out memory bits in the configuration memory during readback. |
| Write MASK | 0x00000000 | Clear the write mask for Control 0 and Control 1 |
| Write Control 1 | 0x00000000 | Control 1 is officially undocumented. |
| Write FAR | 0x00000000 | Set starting address for frame writes to zero. |
| Write COMMAND | 0x00000001 | Arm a frame write. The write will occur on the next write to FDRI. |
| Write FDRI | <547420 words> | Write desired configuration to configuration memory. Since more than 101 words are written, FAR autoincrementing is being used. 547420 words is 5420 frames. Between each frame, COMMAND will be rewritten with 0x1 which re-arms the next write. Note that the configuration memory space is fragmented and autoincrement moves to the next valid address. |
| Write COMMAND | 0x0000000A | Update the routing and configuration flip-flops with the new values in the configuration memory. At this point, the device configuration has been updated but the device is still in programming mode. |
| Write COMMAND | 0x0000000D | Tell the device that the last configuration frame has been received. The device re-enabled its interconnect. |
| Write COMMAND | 0x00000005 | Run the device startup sequence. Documentation claims both a valid CRC check and a DESYNC command are required to trigger the startup. In practice, a bitstream with no CRC checks works just fine. |

#### Differences in the programming sequence between Xilinx architectures

As stated at the beginning of this chapter the bitstream formats for various Xilinx devices have a lot in common. However, there are some small differences which include:

- Device ID - the ID is not only architecture, but actually part specific.
- Configuration Frame Length - number of words in a configuration frame for Series7 is 101, UltraScale - 123, UltraScale+ - 93 and 65 for Spartan6.
- Configuration Registers - The registers and the corresponding addresses are shared among Series7, UltraScale and UltraScale+ architectures, Spartan6 however has a different set of these registers which has to be taken into account during the configuration sequence.

#### Other features

- CRC
  - Calculated automatically from writes: register address and data written
  - Expected value is written to CRC register
  - If there is a mismatch, error is flagged in status register
  - Writes to CRC register can be safely removed from a bitstream
  - Alternatively, replace with write to command register to reset calculated CRC value

## 3.4 Interconnect PIPs

### 3.4.1 Fake PIPs

Some *PIPs* are not "real", in the sense that no bit pattern in the bit-stream correspond to the PIP being used. This is the case for all the *PIPs* in the switchbox in a CLB tile (ex: CLBLM_L_INTER): They either correspond to buffers that are always on (i.e. 1:1 connections such as *CLBLL_L.CLBLL_L_AQ->CLBLL_LOGIC_OUTS0*), or they correspond to permutations of LUT input signals, which is handled by changing the LUT init value accordingly, or they are used to "connect" two signals that are driven by the same signal from within the CLB.

**Warning:** FIXME: Check the above is true.

The bit switchbox in an *INTs* tile also contains a few 1:1 connections that are in fact always present and have no corresponding configuration bits.

### 3.4.2 Regular PIPs

Regular *PIPs* correspond to a bit pattern that is present in the bit stream when the PIP is used in the current design. There is a block of up to 10-ish bits for each destination signal. For each configuration (i.e. source net that can drive the destination) there is a pair of bits that is set.

**Warning:** FIXME: Check if the above is true for PIPs outside of the INT switch box.

For example, when the bits 05_57 and 11_56 are set then SR1END3->SE2BEG3 is enabled, but when 08_56 and 11_56 are set then ER1END3->SE2BEG3 is enabled (in an *INT_L* tile paired with a CLBLL_L tile). A configuration in which all three bits are set is invalid. See *segbits_int_[lr].db* for a complete list of bit pattern for configuring *PIPs*.

### 3.4.3 VCC Drivers

The default state for a net is to be driven high. The *PIPs* that drive a net from *VCC_WIRE* correspond to the "empty configuration" with no bits set.

### 3.4.4 Bidirectional PIPs

Bidirectional *PIPs* are used to stitch together long traces (LV*, LVB*). In case of bidirectional *PIPs* there are two different configuration patterns, one for each direction.

## 3.5 Distributed RAMs (DRAM / SLICEM)

The SLICEM site can turn the 4 LUT6s into distributed RAMs. There are a number of modes, each element is either a 64x1 or a 32x2 distributed RAM (DRAM). The individual elements can be combined into either a 128x1 or 256x1 DRAM.

### 3.5.1 Functions

#### Modes

Some modes can be enabled at the single LUT level. The following modes are:

- 32x2 Single port (32x2S)
- 32x2 Dual port (32x2D)
- 64x1 Single port (64x1S)
- 64x1 Dual port (64x1D)

Some modes are SLICEM wide:

- 128x1 Single port (128x1S)
- 128x1 Dual port (128x1D)
- 256x1

#### Ports

Each LUT element when operating in RAM mode is a DPRAM64.

| Port name | Direction | Width | Description |
|---|---|---|---|
| WA | IN | 8 | Write address |
| A | IN | 6 | Read address |
| DI | IN | 2 | Data input |
| WE | IN | 1 | Write enable |
| CLK | IN | 1 | Clock |
| O6 | OUT | 1 | Data output 1 |
| O5 | OUT | 1 | Data output 2 |

### 3.5.2 Configuration

The configuration for the DRAM is found in the following segbits:

- ALUT.RAM
- ALUT.SMALL
- ADI1MUX.AI
- BLUT.RAM
- BLUT.SMALL
- BDI1MUX.BI
- CLUT.RAM
- CLUT.SMALL
- CDI1MUX.CI
- DLUT.RAM
- DLUT.SMALL
- WA7USED
- WA8USED

In order to use DRAM in a SLICEM, the DLUT in the SLICEM must be a RAM (e.g. DLUT.RAM). In addition the DLUT can never be a dual port RAM because the write address lines for the DLUT are also the read address lines.

#### Segbits for modes

The following table shows the features required for each mode type for each LUT.

| LUTs | 32x2S | 32x2D | 64x1S | 64x1D |
|---|---|---|---|---|
| D | DLUT.RAM<br>DLUT.SMALL | N/A | DLUT.RAM | N/A |
| C | CLUT.RAM<br>CLUT.SMALL<br>CDI1MUX.CI | CLUT.RAM<br>CLUT.SMALL | CLUT.RAM<br>CDI1MUX.CI | CLUT.RAM |
| B | BLUT.RAM<br>BLUT.SMALL<br>BDI1MUX.CI | BLUT.RAM<br>BLUT.SMALL | BLUT.RAM<br>BDI1MUX.CI | BLUT.RAM |
| A | ALUT.RAM<br>ALUT.SMALL<br>ADI1MUX.CI | ALUT.RAM<br>ALUT.SMALL | ALUT.RAM<br>ADI1MUX.CI | ALUT.RAM |

### Ports for modes

In each mode, how the ports are used vary. The following table show the relationship between the LUT mode and ports.

| LUT | 32x2S | 32x2D | 64x1S | 64x1D |
|---|---|---|---|---|
| D | WA[4:0] = A[4:0] = {D5,D4,D3,D2,D1}<br>DI[1:0] = {DX, DI} | N/A | WA[5:0] = A[5:0] = {D6,D5,D4,D3,D2,D1}<br>DI[0] = DI | N/A |
| C | WA[4:0] = A[4:0] = {C5,C4,C3,C2,C1}<br>DI[1:0] = {CX, CI} | WA[4:0] = {D5,D4,D3,D2,D1}<br>A[4:0] = {C5,C4,C3,C2,C1}<br>DI[1:0] = {CX,DI} | WA[5:0] = A[5:0] = {C6,C5,C4,C3,C2,C1}<br>DI[0] = CI | WA[5:0] = {D6,D5,D4,D3,D2,D1}<br>A[5:0] = {C6,C5,C4,C3,C2,C1}<br>DI[0] = DI |
| B | WA[4:0] = A[4:0] = {B5,B4,B3,B2,B1}<br>DI[1:0] = {BX, BI} | WA[4:0] = {D5,D4,D3,D2,D1}<br>A[4:0] = {B5,B4,B3,B2,B1}<br>DI[1:0] = {BX,DI} | WA[5:0] = A[5:0] = {B6,B5,B4,B3,B2,B1}<br>DI[0] = BI | WA[5:0] = {D6,D5,D4,D3,D2,D1}<br>A[5:0] = {B6,B5,B4,B3,B2,B1}<br>DI[0] = DI |
| A | WA[4:0] = A[4:0] = {A5,A4,A3,A2,A1}<br>DI[1:0] = {AX, AI} | WA[4:0] = {D5,D4,D3,D2,D1}<br>A[4:0] = {A5,A4,A3,A2,A1}<br>DI[1:0] = {AX,BLUT.DI[0]} | WA[5:0] = A[5:0] = {A6,A5,A4,A3,A2,A1}<br>DI[0] = AI | WA[5:0] = {D6,D5,D4,D3,D2,D1}<br>A[5:0] = {A6,A5,A4,A3,A2,A1}<br>DI[0] = BLUT.DI[0] |

### Techlib macros

The tech library exposes the following aggregate modes, which are accomplished with the following combinations.

| Macro | Option 1 | Option 2 | Option 3 | Option 4 |
|---|---|---|---|---|
| RAM32M | DLUT = 32x2S<br>CLUT = 32x2D<br>BLUT = 32x2D<br>ALUT = 32x2D | | | |
| RAM32X1D | DLUT = 32x2S<br>CLUT = 32x2D | BLUT = 32x2S<br>ALUT = 32x2D | | |
| RAM32X1S | DLUT = 32x1S | CLUT = 32x1S | BLUT = 32x1S | ALUT = 32x1S |
| RAM32X2S | DLUT = 32x2S<br>CLUT = 32x2D | BLUT = 32x2S<br>ALUT = 32x2D | | |
| RAM64M | DLUT = 64x1S<br>CLUT = 64x1D<br>BLUT = 64x1D<br>ALUT = 64x1D | | | |
| RAM64X1D | DLUT = 64x1S<br>CLUT = 64x1D | BLUT = 64x1S<br>ALUT = 64x1D | | |
| RAM64X1S | DLUT = 64x1S | CLUT = 64x1S | BLUT = 64x1S | ALUT = 64x1S |

## 3.6 Glossary

**ASIC** An application-specific integrated circuit (ASIC) is a chip that is designed and used for a specific purpose, such as video acceleration, machine learning acceleration, and many more purposes. In contrast to *FPGAs*, the programming of an ASIC is fixed at the time of manufacture.

**basic element**

**BEL**

**basic logic element**

**BLE** Basic elements (BELs) or basic logic element (BLEs) are the basic logic units in an *FPGA*, including carry or fast adders (*CFAs*), flip flops (*FFs*), lookup tables (*LUTs*), multiplexers (*MUXes*), and other element types. Note: Programmable interconnects (*PIPs*) are not counted as BELs.

BELs come in two forms:

- Basic BEL - A logic unit which does things.
- Routing BEL - A unit which is statically configured at routing time.

**Bitstream** Binary data that is directly loaded into an *FPGA* to perform configuration. Contains configuration *frames* as well as programming sequences and other commands required to load and activate same.

**Block RAM** Block RAM is inbuilt, configurable memory on an *FPGA*, able to store more data than the *flip flops*. The block RAM can function as dual or single-port memory. Xilinx 7 series devices offer a number of 36 Kb block RAMs, each with two independently controlled 18 Kb RAMs. The number of block RAMs available depends on the specific device.

**CFA** A carry or fast adder (CFA) is a logic element on the *FPGA* that performs fast arithmetic operations.

**Clock** A clock is a square-wave timing signal (50% on, 50% off) generated by an external oscillator and passed into the *FPGA*. The clock frequency drives the sequential logic elements in the FPGA, most importantly the *flip flops*. For example, the FPGA may use a 50 megahertz clock. An FGPA can use one or more clocks and can thus have one or more *clock domains*.

**Clock backbone**

**Clock spine** In Xilinx 7 series devices, the clock backbone or clock spine divides the *clock regions* on the device into two sides, the left and the right side.

**Clock domain** Portion of the device controlled by one *clock*. A clock domain is part of a *horizontal clock row* to one side of the global *clock spine*. The term also often refers to the *tiles* that are associated with these clocks.

**Clock region** Portion of a device including up to 12 *clock domains*. A clock region is situated to the left or right of the global clock spine, and is 50 *CLBs* tall on Xilinx 7 series devices. The clock region includes all synchronous elements in the 50 CLBs and one I/O bank, with a *horizontal clock row* at its center.

**Column** A term used in *bitstream* configuration to denote a collection of *tiles*, physically organized as a vertical line, and configured by the same set of configuration frames. Logic columns span 50 tiles vertically and 2 tiles horizontally (pairs of logic tiles and interconnect tiles).

**Configurable logic block**

**CLB** A configurable logic block (CLB) is the configurable logic unit of an *FPGA*. Also called a **logic cell**. A CLB is a combination of basic logic elements (*BELs*).

**Database** Text files containing meaningful labels for bit positions within *segments*.

**Fabric sub region**

**FSR** Another name for *clock region*.

**Flip flop**

**FF** A flip flop (FF) is a logic element on the *FPGA* that stores state.

**FPGA** A field-programmable gate array (FPGA) is a reprogrammable integrated circuit, or chip. Reprogrammable means you can reconfigure the integrated circuit for different types of computing. You define the configuration via a hardware definition language (*HDL*). The word "field" in *field-programmable gate array* means the circuit is programmable *in the field*, as opposed to during chip manufacture.

**Frame** The fundamental unit of *bitstream* configuration data consisting of 101 *words*. Each frame has a 32-bit frame address and 101 payload words, 32 bits each. The 50th payload word is an EEC. The 7 LSB bits of the frame address are the frame index within the configuration *column* (called *minor frame address* in the Xilinx documentation). The rest of the frame address identifies the configuration column (called *base frame address* in Project X-Ray nomenclature).

The bits in an individual frame are spread out over the entire column. For example, in a logic column with 50 tiles, the first tile is configured with the first two words in each frame, the next tile with the next two words, and so on.

**Frame base address** The first configuration frame address for a *column*. A frame base address has always the 7 LSB bits cleared.

**Fuzzer** Scripts and a makefile to generate one or more *specimens* and then convert the data from those specimens into a *database*.

**Half** Portion of a device defined by a virtual line dividing the two sets of global *clock* buffers present in a device. The two halves are referred to as the top and bottom halves.

**HDL** You use a hardware definition language (HDL) to describe the behavior of an electronic circuit. Popular HDLs include Verilog (inspired by C) and VHDL (inspired by Ada).

**Horizontal clock row**

**HROW** Portion of a device including 12 horizontal *clocks* and the 50 interconnect and function tiles associated with them. A *half* contains one or more horizontal clock rows and each half may have a different number of rows.

**I/O block** One of the configurable input/output blocks that connect the *FPGA* to external devices.

**Interconnect tile**

**INT** An interconnect tile (*INT_L*, *INT_R*) is used to connect other tiles to the fabric. It is also frequently called a switch box.

**LUT** A lookup table (LUT) is a logic element on the *FPGA*. LUTs function as a ROM, apply combinatorial logic, and generate the output value for a given set of inputs.

**MUX** A multiplexer (MUX) is a multi-input, single-output switch controled by logic.

**Node** A routing node on the device. A node is a collection of *wires* spanning one or more *tiles*. Nodes that are local to a tile map 1:1 to a wire. A node that spans multiple tiles maps to multiple wires, one in each tile it spans.

**PIP**

**Programmable interconnect point** A programmable interconnect point (PIP) is a connection point between two wires in a tile that may be enabled or disabled by the configuration.

**PnR**

**Place and route** Place and route (PnR) is the process of taking logic and placing it into hardware logic elements on the *FPGA*, and then routing the signals between the placed elements.

**Region of interest**

**ROI** Region of interest (ROI) is used in *Project X-Ray* to denote a rectangular region on the *FPGA* that is the focus of our study. The current region of interest is *SLICE_X12Y100:SLICE_X27Y149* on a *xc7a50tfgg484-1* chip.

**Routing fabric** The *wires* and programmable interconnects (*PIPs*) connecting the logic blocks in an *FPGA*.

**Segment** All configuration bits for a horizontal slice of a *column*. This corresponds to two ranges: a range of *frames* and a range of *words* within frames. A segment of a logic column is 36 frames wide and 2 words high.

**Site** Portion of a tile where *BELs* can be placed. The *slices* in a *CLB* tile are sites.

**Slice** Portion of a *tile* that contains *BELs*. A *CLBLL_L/CLBLL_R* tile contains two *SLICEL* slices. A *CLBLM_L/CLBLM_R* tile contains one *SLICEL* slice and one *SLICEM* slice. *SLICEL* and *SLICEM* are the most common types of slice, containing the *LUTs* and *flip flops* that are the basic logic units of the FPGA.

**Specimen** A *bitstream* of a (usually auto-generated) design with additional files containing information about the placed and routed design. These additional files are usually generated using Vivado TCL scripts querying the Vivado design database.

**Tile** Fundamental unit of physical structure containing a single type of resource or function. A container for *sites* and *slices*. The FPGA chip is a grid of tiles.

The most important tile types are left and right interconnect tiles (*INT_L* and *INT_R*) and left and right *CLB* logic/memory tiles (*CLBLL_L*, *CLBLL_R*, *CLBLM_L*, *CLBLM_R*).

**Wire** Physical wire within a *tile*.

**Word** 32 bits stored in big-endian order. Fundamental unit of *bitstream* format.

## 3.7 References

### 3.7.1 Xilinx documents one should be familiar with:

### UG470: 7 Series FPGAs Configuration User Guide

https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf

*Chapter 5: Configuration Details* contains a good description of the overall bit-stream format. (See section "Bitstream Composition" and following.)

### UG912: Vivado Design Suite Properties Reference Guide

http://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug912-vivado-properties.pdf

Contains an excellent description of the in-memory data structures and associated properties Vivado uses to describe the design and the chip. The TCL interface provides a convenient interface to access this information.

### UG903: Vivado Design Suite User Guide: Using Constraints

http://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug903-vivado-using-constraints.pdf

The fuzzers generate designs (HDL + Constraints) that use many physical contraints constraints (placement and routing) to produce bit-streams with exactly the desired features. It helps to learn about the available constraints before starting to write fuzzers.

### UG901: Vivado Design Suite User Guide: Synthesis

http://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug901-vivado-synthesis.pdf

*Chapter 2: Synthesis Attributes* contains an overview of the Verilog attributes that can be used to control Vivado Synthesis. Many of them are useful for writing fuzzer designs. There is some natural overlap with UG903.

### UG909: Vivado Design Suite User Guide: Partial Reconfiguration

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug909-vivado-partial-reconfiguration.pdf

Among other things this UG contains some valuable information on how to constrain a design in a way so that the items inside a pblock are strictly separate from the items outside that pblock.

### UG474: 7 Series FPGAs Configurable Logic Block

https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

Describes the capabilities of a CLB, the most important non-interconnect resource of a Xilinx FPGA.

### 3.7.2 Other documentation that might be of use:

Doc of .bit container file format: http://www.pldtool.com/pdf/fmt_xilinxbit.pdf

Open-Source Bitstream Generation for FPGAs, Ritesh K Soni, Master Thesis: https://vtechworks.lib.vt.edu/bitstream/handle/10919/51836/Soni_RK_T_2013.pdf

VTR-to-Bitstream, Eddie Hung: https://eddiehung.github.io/vtb.html

From the bitstream to the netlist, Jean-Baptiste Note and Éric Rannaud: http://www.fabienm.eu/flf/wp-content/uploads/2014/11/Note2008.pdf

Wolfgang Spraul's Spartan-6 (xc6slx9) project: https://github.com/Wolfgang-Spraul/fpgatools

Marek Vasut's Typhoon Cyclone IV project: http://git.bfuser.eu/?p=marex/typhoon.git

XDL generator/imported for Vivado: https://github.com/byuccl/tincr

## 3.8 Contributor Covenant Code of Conduct

### 3.8.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

### 3.8.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission

- Other conduct which could reasonably be considered inappropriate in a professional setting

### 3.8.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

### 3.8.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

### 3.8.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at atom@github.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

### 3.8.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4

## 3.9 Guide to updating the Project X-Ray docs

We welcome updates to the Project X-Ray docs. The docs are published on Read the Docs and the source is on the docs branch on GitHub.

The reason for using the `docs` branch is to avoid running the full CI test suite which triggers when merging anything to `master`. Ultimately of course the `docs` branch needs to be synchronized with `master`, but this can be done in bulk.

Updating the docs is a three-step process: Make your updates, test your updates, send a pull request.

### 3.9.1 1. Make your updates

The standard Project X-Ray *contribution guidelines* apply to doc updates too.

Follow your usual process for updating content on GitHub. See GitHub's guide to working with forks.

### 3.9.2 2. Test your updates

Before sending a pull request with your doc updates, you need to check the effects of your changes on the page you've updated and on the docs as a whole.

#### Check your markup

There are a few places on the web where you can view ReStructured Text rendered as HTML. For example: https://livesphinx.herokuapp.com/

#### Perform basic tests: make html and linkcheck

If your changes are quite simple, you can perform a few basic checks before sending a pull request. At minimum:

- Check that `make html` generates output without errors
- Check that `make linkcheck` reports no warnings.
- When editing, `make livehtml` is helpful.

To make these checks work, you need to install Sphinx. We recommend `pipenv`.

Follow the steps below to install `pipenv` via `pip`, run `pipenv install` in the `docs` directory, then run `pipenv shell` to enter an environment where Sphinx and all the necessary plugins are installed:

Steps in detail, on Linux:

1. Install pip:

```
sudo apt install python-pip
```

2. Install pipenv - see the pipenv installation guide:

```
pip install pipenv
```

3. Add pipenv to your path, as recommended in the pipenv installation guide. On Linux, add this in your `~/.profile` file:

```
export PATH=$PATH:~/.local/bin source ~/.profile
```

   Note: On OS X the path is different: `~/Library/Python/2.7/bin`

4. Go to the docs directory in the Project X-Ray repo:

```
cd ~/github-repos/prjxray/docs
```

5. Run pipenv to install the Sphinx environment:

```
pipenv install
```

6. Activate the shell:

```
pipenv shell
```

7. Run the HTML build checker, and check for *errors*:

```
make html
```

8. Run the link checker, and check for *warnings*:

```
make linkcheck
```

9. To leave the shell, type: `exit`.

#### Perform more comprehensive testing on your own staging doc site

If your changes are more comprehensive, you should do a full test of your fork of the docs before sending a pull request to the Project X-Ray repo. You can test your fork by viewing the docs on your own copy of the Read the Docs build.

Follow these steps to create your own staging doc site on Read the Docs (RtD):

1. Sign up for a RtD account here: https://readthedocs.org/
2. Go to your RtD connected services, click **Connect to GitHub**, and connect RtD to your GitHub account. (If you decide not to do this, you'll need to import your project manually in the following steps.)
3. Go to your RtD dashboard.
4. Click **Import a Project**.
5. Add your GitHub fork of the Project X-Ray project. Give your doc site a **name** that distinguishes it from the canonical Project X-Ray docs. For example, `your-username-prjxray`.
6. Make your doc site **protected**. See the RtD guide to privacy levels. Reason for protecting your doc site: If you leave your doc site public, it will appear in web searches. That may be confusing for readers who are looking for the canonical Project X-Ray docs.
7. Set RtD to build from your branch, rather than from `docs`. This ensures that the content you see on your doc site reflect your latest updates:
   - On your RtD dashboard, open **your project**, then go to **Admin > Advanced Settings.**
   - Add the name of your branch in **Default branch**. This is the branch that the "latest" build config points to. If you leave this field empty, RtD uses `master` or `trunk`.
8. RtD now builds your doc site, based on the contents in your Project X-Ray fork.
9. See the RtD getting-started guide for more info.

### 3.9.3 3. Send a pull request

Follow your standard GitHub process to send a pull request to the Project X-Ray repo. See the GitHub guide to creating a pull request from a fork.

CHAPTER

FOUR

# DATABASE DEVELOPMENT PROCESS

## 4.1 Contributing to Project X-Ray

There are a couple of guidelines when contributing to Project X-Ray which are listed here.

### 4.1.1 Sending

All contributions should be sent as GitHub Pull requests.

### 4.1.2 License

All software (code, associated documentation, support files, etc) in the Project X-Ray repository are licensed under the very permissive ISC Licence. A copy can be found in the `LICENSE` file.

All new contributions must also be released under this license.

### 4.1.3 Code of Conduct

By contributing you agree to the *code of conduct*. We follow the open source best practice of using the Contributor Covenant for our Code of Conduct.

### 4.1.4 Sign your work

To improve tracking of who did what, we follow the Linux Kernel's "sign your work" system. This is also called a "DCO" or "Developer's Certificate of Origin".

**All** commits are required to include this sign off and we use the Probot DCO App to check pull requests for this.

The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as a open-source patch. The rules are pretty simple: if you can certify the below:

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or
```

(continues on next page)

(continued from previous page)

```
    (b) The contribution is based upon previous work that, to the best
        of my knowledge, is covered under an appropriate open source
        license and I have the right under that license to submit that
        work with modifications, whether created in whole or in part
        by me, under the same open source license (unless I am
        permitted to submit under a different license), as indicated
        in the file; or

    (c) The contribution was provided directly to me by some other
        person who certified (a), (b) or (c) and I have not modified
        it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

then you just add a line saying

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

using your real name (sorry, no pseudonyms or anonymous contributions.)

You can add the signoff as part of your commit statement. For example:

```
git commit --signoff -a -m "Fixed some errors."
```

*Hint:* If you've forgotten to add a signoff to one or more commits, you can use the following command to add signoffs to all commits between you and the upstream master:

```
git rebase --signoff upstream/master
```

### 4.1.5 Contributing to the docs

In addition to the above contribution guidelines, see the guide to *updating the Project X-Ray docs*.

---

This file is generated from *README.md*, please edit that file then run the `./.github/update-contributing.py`.

## 4.2 Adding New Fuzzer

This chapter describes how to create a new fuzzer using a DSP as an example target primitive. The files that are generated with such fuzzer have been described in more detail in the *Database* chapter. The process of creating a new fuzzer consists of two elements, namely base address calculation and feature fuzzing.

### 4.2.1 Base Address Calculation

The base address calculation is based on segmatching (statistical constraint solver) the base addresses. A similar technique is used in most fuzzers for solving configuration bits.

#### Methodology

In this technique all IP blocks are changed in parallel. This means that log(N, 2) bitstreams are required instead of N to get the same number of base addresses. However, as part of this conversion, address propagation is also generally discouraged. So it is also recommended to toggle bits in all IP blocks in a column, not just one. In the CLB case, this means that every single CLB tile gets one bit set to a random value. If there are 4 CLB CMT columns in the ROI, this means we'd randomly set 4 * 50 bits in every bitstream. With 200 bits, it takes minimum floor(log(200, 2)) => 8 bitstreams (specimens) to solve all of them.

#### Calculating the base address

1. Find a tilegrid fuzzer to copy, e.g. "dsp"
2. Enter your copied directory
3. Edit *top.py*
    a. Refer to the Xilinx 7 Series Library guide and/or Vivado layout to understand the primitive you need to instantiate
    b. Find a single bit parameter that can be easily toggled, such as a clock inverter or a bulk configuration bit
    c. Find the correct site type in gen_sites()
    d. Instantiate the correct verilog library macro in top
    e. LOC it, if necessary. It's necessary to LOC it if there is more than one
4. Run make, and look at Vivado's output. Especially if you took shortcuts instantiating your macro (ex: not connecting critical ports) you may need to add DRC waivers to generate.tcl
5. Inspect the `build/segbits_tilegrid.tdb` to observe bit addresses, for example `DSP_L_X22Y0.DWORD:0.DFRAME:1b 0040171B_000_01`
    1. The `DFRAME` etc entries are deltas to convert this feature offset to the base address for the tile
    2. We will fix them in the subsequent step
6. Correct Makefile's `GENERATE_ARGS` to make it the section base address instead of a specific bit in that memory region
    1. Align address to 0x80: 0x0040171B => –dframe 1B to yield a base address of 0x00401700
    2. Correct word offset. This is harder since it requires some knowledge of how and where the IP block memory is as a whole
        i. If there is only one tile of this type in the DSP column: start by assuming it occupies the entire address range. In this step add a delta to make the word offset 0 (–dword 0) and later indicate that it occupies 101 words (all of them)
        ii. If there are multiple: compare the delta between adjacent tiles to get the pitch. This should give an upper bound on the address size. Make a guess with that in mind and you may have to correct it later when you have better information.
    3. Align bits to 0: 1 => –dbit 1

7. Run `make clean && make`
8. Verify `build/segbits_tilegrid.tdb` now looks resolved
    1. Ex: `DSP_L_X22Y0.DWORD:0.DFRAME:1b 0040171B_000_01`
    2. In this case there were several DSP48 sites per DSP column
9. Find the number of frames for your tile
    1. Run `$XRAY_BLOCKWIDTH build/specimen_001/design.bit`
    2. Find the base address you used above i.e. we used `0x00401700`, so use `0x00401700: 0x1B` (0x1C => 28)
    3. This information is in the part YAML file, but is not as easy to read
10. Return to the main tilegrid directory
11. Edit `tilegrid/add_tdb.py`
    1. Find `tdb_fns` and add an entry for your tile type e.g. `(dsp/build/segbits_tilegrid.tdb", 28, 10)`
    2. This is declared to be 28 frames wide and occupy 10 words per tile in the DSP column
12. Run `make` in the tilegrid directory
13. Look at `build/tilegrid.json`
    1. Observe your base address(es) have been inserted (look for bits `CLB_IO_CLK` entry in the `DSP_L_*` tiles)

### 4.2.2 Feature Fuzzing

The general idea behind fuzzers is to pick some element in the device (say a block RAM or IOB) to target and write a design that is implemented in a specific element. Next, we need to create variations of the design (called specimens) that vary the design parameters, for example, changing the configuration of a single pin and process them in Vivado in order to obtain the respective bitstreams. Finally, by looking at all the resulting specimens, the information which bits in which frame correspond to a particular choice in the design can be correlated. Looking at the implemented design in Vivado with "Show Routing Resources" turned on is quite helpful in understanding what all choices exist.

#### Fuzzer structure

Typically a fuzzer directory consists of a mixture of makefiles, bash, python and tcl scripts. Many of the scripts are shared among fuzzers and only some of them have to be modified when working on a new fuzzer.

- Makefile and a number of sub-makefiles contain various targets that have to be run in order to run the fuzzer and commit the results to the final database. The most important ones are:
    - *run* - run the fuzzer to generate the netlist, create bitstreams in Vivado, solve the bits and update the final database with the newly calculated results.
    - *database* - run the fuzzer without updating the final database

The changes usually done in the Makefile concern various script parameters, like number of specimen, regular expressions for inclusion or exclusion list of features to be calculated or maximal number of iterations the fuzzer should try to solve the bits for.

- *top.py* - Python script used to generate the verilog netlist which will be used by the fuzzer for all Vivado runs.

- *generate.tcl* - tcl script used by Vivado to read the base verilog design, if necessary tweak some properties and write out the specimen bitstreams
- *generate.py* - Python script that reads the generated bitstream and takes a parameterized description of the design (usually in the form of a csv file) in order to produce a file with information about which features are enabled and which are disabled in a given segment.

#### Creating the fuzzer

1. Open the *top.py* script and modify the content of the top module by instantiating a DSP primitive and specifying some parameters. Use LOC and DONT_TOUCH attributes to avoid some design optimization since the netlists are in many cases very artificial.
2. Make sure the *top.py* script generates apart from the top.v netlist, a csv file with the values of parameters used in the generated netlist.
3. Modify the *generate.tcl* script to read the netlist generated in step 1, apply, if necessary, some parameters from the csv file generated in step 2 and write out the bitstream
4. Modify the *generate.py* script to insert the tags, which signify whether a feature is disabled or enabled in a site, based on the csv parameters file generated in step 1

## 4.3 Fuzzers

Fuzzers are a set of tests which generate a design, feed it to Vivado, and look at the resulting bitstream to make some conclusion. This is how the contents of the database are generated.

The general idea behind fuzzers is to pick some element in the device (say a block RAM or IOB) to target. If you picked the IOB, you'd write a design that is implemented in a specific IOB. Then you'd create a program that creates variations of the design (called specimens) that vary the design parameters, for example, changing the configuration of a single pin.

A lot of this program is TCL that runs inside Vivado to change the design parameters, because it is a bit faster to load in one Verilog model and use TCL to replicate it with varying inputs instead of having different models and loading them individually.

By looking at all the resulting specimens, you can correlate which bits in which frame correspond to a particular choice in the design.

Looking at the implemented design in Vivado with "Show Routing Resources" turned on is quite helpful in understanding what all choices exist.

### 4.3.1 Configurable Logic Blocks (CLB)

#### clb-ffconfig Fuzzer

Documents FF configuration.

Note Vivado GUI is misleading in some cases where it shows configuration per FF, but its actually per SLICE

#### Primitive pin map

#### Primitive bit map

#### FFSYNC

Configures whether a storage element is synchronous or asynchronous.

Scope: entire site (not individual FFs)

#### LATCH

Configures latch vs FF behavior for the CLB

#### N*FF.ZRST

Configures stored value when reset is asserted

#### N*FF.ZINI

Sets GSR FF or latch value

#### CEUSEDMUX

Configures ability to drive clock enable (CE) or always enable clock

#### SRUSEDMUX

Configures ability to reset FF after GSR

TODO: how used when SR?

#### CLKINV

Configures whether to invert the clock going into a slice.

Scope: entire site (not individual FFs)

#### clb-ffsrcemux Fuzzer

#### CEUSEDMUX

Configures whether clock enable (CE) is used or clock always on

#### SRUSEDMUX

Configures whether FF can be reset or simply uses D value

XXX: How used when SR?

### clb-lutinit Fuzzer

#### NLUT.INIT

Sites: CLBL[LM]_[LR].SLICE[LM]_X[01] (all CLB)

Sets the LUT6 INIT property

### clb-n5ffmux Fuzzer

#### N5FFMUX

The A5FFMUX family of CLB muxes feed the D input of A5FF family of FFs

### clb-ncy0 Fuzzer

#### CARRY4.NCY0

The ACY0 family of CLB muxes feeds the CARRY4.DI0 family

### clb-ndi1mux Fuzzer

#### NDI1MUX

Configures the NDI1MUX mux which provides the DI1 input on CLB RAM.

Available selections varies by A/B/C/D, see db for details.

### clb-nffmux Fuzzer

#### NFFMUX

Configures the AFFMUX family of CLB muxes which feed the D input of the AFF series of FFs.

Available selections varies by A/B/C/D, see db for details.

#### clb-noutmux Fuzzer

#### [A-D]FFMUX

Configures the AOUTMUX family of CLB muxes which feed the AMUX family of CLB outputs

Available selections varies by A/B/C/D, see db for details.

#### clb-precyinit Fuzzer

#### PRECYINIT

Configures the PRECYINIT mux which provides CARRY4's first carry chain input

#### clb-ram Fuzzer

#### NLUT.RAM

Set to make a RAM* family primitive, otherwise is a SRL or LUT function generator.

#### NLUT.SMALL

Seems to be set on smaller primitives.

#### NLUT.SRL

Whether to make a shift register LUT (SRL). Set when using SRL16E or SRLC32E

#### WA7USED

Set to 1 to propagate CLB's CX input to WA7

#### WA8USED

Set to 1 to propagate CLB's BX input to WA8

#### WEMUX.CE

### 4.3.2 Block RAM (BRAM)

#### BRAM Configuration

Solves for BRAM configuration bits (18K vs 36K, etc)

#### BRAM Data

Solves for BRAM data bits

See workflow comments: https://github.com/SymbiFlow/prjxray/pull/180

#### RAMB36 features

This fuzzer emits features that only are used in the RAMB36E1 cell. There are 3 categories:

- ECC
- RAM extension
- Odd address modes

#### Odd address modes

Most RAMB36E1 address widths are expressed by configuring the underlying RAMB18E1 to handle half of the data. So `RAMB36.READ_WIDTH = 4` is expressed as `RAM18_Y0.READ_WIDTH = 2` and `RAM18_Y1.READ_WIDTH = 2`. However two address widths (1 and 9) are odd (e.g. not divisible by 2). In these cases, a RAMB36E1 specific feature is used. So `RAMB36.READ_WIDTH = 9` is expressed as:

- `RAMB18_Y0.READ_WIDTH_4`
- `RAMB18_Y1.READ_WIDTH_4`
- `RAMB36.BRAM36_READ_WIDTH_1`

and `RAMB36.READ_WIDTH = 1` is expressed as:

- `RAMB18_Y0.READ_WIDTH_1`
- `RAMB18_Y1.READ_WIDTH_1`
- `RAMB36.BRAM36_READ_WIDTH_1`

### 4.3.3 Input / Output (IOB)

#### IOB Fuzzer

### 4.3.4 Clocking (CMT, PLL, BUFG, etc)

#### HCLK_IOI interconnect fuzzer

This Fuzzer is a copy of the 047-hclk-ioi-pips fuzzer, but only solves IDELAYCTRL pips. It is separated from the original hclk-ioi-pips as these pips need different segmatch arguments to avoid mergedb conflicts. Indeed segmatch `-c` parameter is set to 3.

The IDEALYCTRL pips are in the following form:

```
HCLK_IOI3.HCLK_IOI_IDELAYCTRL_REFCLK.HCLK_IOI_LEAF_GCLK_((TOP)|(BOT))[0-9]
```

### BUFG interconnect fuzzer

Solves pips located within the BUFG switch box.

### BUFG interconnect fuzzer

Solves pips located within the BUFG switch box.

### HCLK_CMT interconnect fuzzer

Solves pips located within the HCLK_CMT switch box.

### HCLK_IOI interconnect fuzzer

Solves pips located within the HCLK_IOI switch box.

The segmatch `-c` argument is set to 2, as all the pips require at maximum 2 bits.

### Fuzzer for INT PIPs driving the CLK wires

Run this fuzzer a few times until it produces an empty todo.txt file (`make run` will run this loop).

### Fuzzer for PIPs in HCLK titles

Run this fuzzer once.

It cannot solve HCLK.HCLK_CK_INOUT_* family

### Fuzzer for the PIPs of CMT_TOP_[LR]_LOWER_B tiles.

The fuzzer instantiates a MMCM in each available site with 2/3 probability of using it. Once used it is connected randomly to various clock and logic resources.

For some nets a randomized "manual" route is chosen to cover as many routing scenarios as possible.

The information whether a MMCM is used or not is stored in a file (`"design.txt"`) along with the randomized route (`route.txt`)

After the design synthesis the `generate.py` sets fixed routes on some nets which is read from the `route.txt` file. The rest of the design is routed in the regular way. The script also dumps all used PIPs (as reported by Vivado) to the `design_pips.txt`.

The tag generation is done in the following way:

- If a MMCM site is occupied then tags for all active PIPs are emitted as 1s. No tags are emitted for inactive PIPs.
- When a MMCM site is not occupied (IN_USE=0) then tags for all PIPs for the CMT tile are emitted as 0s.
- The IN_USE tag is emitted directly.

The raw solution of tag bits is postprocessed via the custom script `fixup_and_group.py`. The script does two things:

- Clears all bits found for the IN_USE tag in all other tags. Those bits are common to all of them.

- Groups tags according to the group definitions read from the `tag_groups.txt` file. Bits that are common to the group are set as 0 in each tag that belongs to it (tags within a group are exclusive).

#### Clock Management Tile (CMT) - MMCM Fuzzer

Bits that are part of the dynamic configration register interface (see APPNOTE XAPP888) are handled specially.

#### Clock Management Tile (CMT) - Phase Lock Loop (PLL) Fuzzer

Bits that are part of the dynamic configration register interface (see APPNOTE XAPP888) are handled specially.

#### Fuzzer for the PIPs of CMT_TOP_[LR]_UPPER_T tiles.

The fuzzer instantiates a PLL in each available site with 2/3 probability of using it. Once used it is connected randomly to various clock and logic resources.

For some nets a randomized "manual" route is chosen to cover as many routing scenarios as possible.

The information whether a PLL is used or not is stored in a file (`"design.txt"`) along with the randomized route (`route.txt`)

After the design synthesis the `generate.py` sets fixed routes on some nets which is read from the `route.txt` file. The rest of the design is routed in the regular way. The script also dumps all used PIPs (as reported by Vivado) to the `design_pips.txt`.

The tag generation is done in the following way:

- If a PLL site is occupied then tags for all active PIPs are emitted as 1s. No tags are emitted for inactive PIPs.
- When a PLL site is not occupied (IN_USE=0) then tags for all PIPs for the CMT tile are emitted as 0s.
- The IN_USE tag is emitted directly.

The raw solution of tag bits is postprocessed via the custom script `fixup_and_group.py`. The script does two things:

- Clears all bits found for the IN_USE tag in all other tags. Those bits are common to all of them.
- Groups tags according to the group definitions read from the `tag_groups.txt` file. Bits that are common to the group are set as 0 in each tag that belongs to it (tags within a group are exclusive).

### 4.3.5 Programmable Interconnect Points (PIPs)

#### Fuzzer for bidirectional INT PIPs

Run this fuzzer a few times until it produces an empty todo.txt file (`make run` will run this loop).

### Fuzzer for the FAN_ALT*.BYP_BOUNCE PIPs

This fuzzer solves the FAN_ALT.BYP_BOUNCE PIPs which were occasionally solved incorrectly in 050-pip-seed or 056-pip-rem.

### Fuzzer for INT PIPs driving the CTRL wires

Run this fuzzer a few times until it produces an empty todo.txt file (`make run` will run this loop).

### Fuzzer for the FAN_ALT.*GFAN PIPs and BYP_ALT.*GFAN PIPs

This fuzzer solves the FAN_ALT.GFAN PIPs which had collisions with the GFAN PIPs as well as the BYP_ALT.GFAN PIPs.

### Fuzzer for INT PIPs driving the GFAN wires with GND

Run this fuzzer once.

### Fuzzer for INT LOGIC_OUTS -> IMUX PIPs

Run this fuzzer a few times until it produces an empty todo.txt file (`make run` will run this loop).

### Fuzzer for the remaining INT PIPs

Run this fuzzer a few times until it produces an empty todo.txt file (`make run` will run this loop).

This fuzzer occationally fails (depending on some random variables). Just restart it if you encounter this issue. The script behind `make run` automatically handles errors by re-starting a run if an error occurs.

### Solvability

Known issues:

- INT.CTRL0: goes into CLB's SR. This cannot be routed through

Jenkins build 3 (78fa4bd5, success) for example solved the following types:

- INT_L.EE4BEG0.LH12
- INT_L.FAN_ALT1.GFAN1
- INT_L.FAN_ALT4.BYP_BOUNCE_N3_3
- INT_L.LH0.EE4END3
- INT_L.LH0.LV_L9
- INT_L.LH0.SS6END3
- INT_L.LVB_L12.WW4END3
- INT_L.SW6BEG0.LV_L0

### Generic fuzzer for INT PIPs

Run this fuzzer a few times until it stops adding new PIPs to the database.

Sample runs:

- 78fa4bd5
  - jenkins 3, success
  - intpips: 1 iter, N=200, -m 5 -M 15
  - intpips todo final: N/A
  - intpips segbits_int_l.db lines: 3374
  - rempips todo initial: 279
  - rempips todo final (32): 9
- 20e09ca7
  - jenkins 21, rempips failure
  - intpips: 6 iters, N=48, -m 15 -M 45
  - intpips segbits_int_l.db lines: 3364
  - rempips todo initial: 294
  - rempips todo final (51): 294
- 1182359f
  - jenkins 23, intpips failure
  - inpips: 12 iters, N=48, -m 15 -M 45
  - intpips todo final: 495
  - inpips segbits_int_l.db lines: 5167
  - rempips todo: N/A

### const0

These show up in large numbers after a full solve. This means that it either has trouble solving these or simply cannot. Counts from sample run

Includes:

- INT.BYP_ALT*.LOGIC_OUTS* (24)
  - Ex: INT.BYP_ALT2.LOGIC_OUTS14
- INT.[NESW]*.LOGIC_OUTS* (576)
  - Ex: INT.EE4BEG2.LOGIC_OUTS2
  - Ex: INT.EL1BEG_N3.LOGIC_OUTS0
  - Ex: INT.WR1BEG3.LOGIC_OUTS2
- INT.IMUX*.* (1151)
  - Ex: INT.IMUX0.NL1END0
  - Ex: INT.IMUX0.FAN_BOUNCE7

  - Ex: INT.IMUX14.LOGIC_OUTS7

#### GFAN

Includes:

- Easily solves: INT.IMUX_L*.GFAN*
- Can solve: INT.BYP_ALT*.GFAN*
- Cannot solve: INT.IMUX*.GFAN* (solves as "<m1 0> ")

#### IMUX

- Okay: BYP_ALT*.VCC_WIRE
- Cannot solve: INT.IMUX[0-9]+.VCC_WIRE
- Cannot solve: INT.IMUX_L[0-9]+.VCC_WIRE

See https://github.com/SymbiFlow/prjxray/issues/383

### 4.3.6 Hard Block Fuzzers

#### XADC Fuzzer

As of this writing, this fuzzer is not in the ROI To use it, you must run tilegrid first with these options (artix7):

export XRAY_ROI_GRID_Y2=103 export XRAY_ROI="SLICE_X0Y100:SLICE_X35Y149 RAMB18_X0Y40:RAMB18_X0Y59 RAMB36_X0Y20:RAMB36_X0Y29 DSP48_X0Y40:DSP48_X0Y59 IOB_X0Y100:IOB_X0Y149 XADC_X0Y0:XADC_X0Y0" 005-tilegrid$ make monitor/build/segbits_tilegrid.tdb 005-tilegrid$ make

Then run this fuzzer

### 4.3.7 Grid and Wire

#### Tilegrid Fuzzer

This fuzzer creates the tilegrid.json bitstream database. This database contains segment definitions including base frame address and frame offsets.

#### Example workflow for CLB

generate.tcl LOCs one LUT per segment column towards generating frame base addresses.

A reference bitstream is generated and then:

- a series of bitstreams are generated each with one LUT bit toggled; then
- these are compared to find a toggled bit in the CLB segment column; then
- the resulting address is truncated to get the base frame address.

Finally, generate.py calculates the segment word offsets based on known segment column structure

#### Environment variables

##### XRAY_ROI

This environment variable must be set with a valid ROI. See database for example values

##### XRAY_EXCLUDE_ROI_TILEGRID

This environment variable must be set in case the part selected does not allow some tiles to be locked.

Error example (when using the artix 200T part): `ERROR: [Place 30-25] Component carry4_SLICE_X82Y249 has been locked to a prohibited site SLICE_X82Y249.`

To avoid this error, the `XRAY_EXCLUDE_ROI_TILEGRID` defines an ROI that is not taken into account when building the tilegrid, therefore excluding the problematic un-lockable sites.

As the resulting output file, `tilegrid.json`, is going to be checked against the one produced in the `074-dump_all` fuzzer, also the latter one needs to produce a reduced tilegrid, with the excluded tiles specified with the environment variable.

##### XRAY_ROI_FRAMES

This can be set to a specific value to speed up processing and reduce disk space If you don't know where your ROI is, just set to to include all values (0x00000000:0xffffffff)

##### XRAY_ROI_GRID_*

Optionally, use these as a small performance optimization:

- XRAY_ROI_GRID_X1
- XRAY_ROI_GRID_X2
- XRAY_ROI_GRID_Y1
- XRAY_ROI_GRID_Y2

These should, if unused, be set to -1, with this caveat:

WARNING: CLB test generates this based on CLBs but implicitly includes INT

Therefore, if you don't set an explicit XRAY_ROI_GRID_* it may fail if you don't have a CLB*_L at left and a CLB*_R at right.

### 4.3.8 All Fuzzers

#### cfg fuzzer

This fuzzer solves some of the bits in the CFG_CENTER_MID tile The tile contains sites of the following types: BSCAN, USR_ACCESS, CAPTURE, STARTUP, FRAME_ECC, DCIRESET and ICAP. DCIRESET and USR_ACCESS don't really have any parameters. The parameters on CAPTURE and FRAME_ECC don't toggle any bits in the bitstream.

#### PS7 verilog cell definition extractor

Extracts all pins of the PS7 bel from Vivado, groups them into ports, writes them to a JSON file.

## 4.4 Minitests

Minitests are experiments to figure out how things work. They allow us to understand how to better write new fuzzers.

### 4.4.1 CLB_BUSED Minitest

#### Purpose

Tests for BUSED bit

#### Result

However got this

```
seg SEG_CLBLL_R_X13Y101
bit 30_24

seg SEG_CLBLL_R_X13Y100
bit 30_24
```

which seems to indicate there is no such bit, or it was rolled into PIP stuff already

### 4.4.2 CLB_MUXF8 Minitest

#### Purpose

This tests an issue related to Vivado 2017.2 vs 2017.3 changing MUXF8 behavior The general issue is the LUT6_2 cannot be used with a MUXF8 (even if O5 is unused)

#### General notes:

- 2017.2: LUT6_2 works with MUXF8
- 2017.3: LUT6_2 does not work with MUXF8
- All: LUT6 works with MUXF8
- All: MUXF8 (even with MUXF7) can be instantiated unconnected
- 2017.4 seems to behave like 2017.3

### 4.4.3 FIXEDPNR Minitest

#### Purpose

#### Result

Preliminary result

### 4.4.4 Minitests for IDELAY

#### 1. basys3_idelay_var

A design for Basys3 board.

#### Description

This test generates a 50MHz square wave on an output pin which is then fed back to the FPGA IDELAY bel through another input pin. The delayed signal is then routed to yet another output pin which allows it to be compared with the input signal using an oscilloscope. The IDELAY is calibrated using 100MHz clock hence delays from 0 to 5ns can be achieved.

The switch `SW0` acts as reset. The switch `SW1` allows to change the delay value. One toggle of that switch increases the delay counter by one.

The `LED0` blinks continuously. The `LED1` indicates that the calibration of IDELAY has been completed (the `RDY` signal from IDELAYCTRL bel). Leds `LED11` through `LED15` indicate current delay setting (the `CNTVALUEOUT` of IDELAY bel).

#### Physical loopback

Consider the `JXADC` connector on the Basys3 board as seen when looking at the board edge:

```
 -- -- -- -- -- --
| 6| 5| 4| 3| 2| 1|
 -- -- -- -- -- --
|12|11|10| 9| 8| 7|
 -- -- -- -- -- --
```

- Pin1 - Signal output. Connect to CH1 of the oscilloscope.
- Pin2 - Delayed signal output. Connect to CH2 of the oscilloscope.
- Pin7 - Delay signal input, connect to Pin8.
- Pin8 - Signal output. Connect to Pin7.

**The oscilloscope must have bandwidth of at least 100MHz.**

#### 2. basys3_idelay_const

This design generates 32 independently shifted 50MHz square waves using constant delay IDELAY blocks. Delays between individual signals can be measured using an oscilloscope. Due to the fact that each delay step is about 100-150ps and the FPGA fabric + IOBs also introduce their own delays, actual delay values may be hard to measure.

#### 3. basys3_idelay_histogram

This design transmitts a pseudo-random data stream throught one output pin and then receives it through another one with the use of IDELAY. A physical loopback is required. The received data is compared with the transmitted one. Receive errors are counted separately for each one of 32 possible delay settings of the IDELAY bel. Values of 32 error counters are periodically pinted using the UART as an ASCII string.

There is a control state machine which performs the following sequence once per ~0.5s (adjustable).

1. Set delay of the IDELAY bel.
2. Wait for it to stabilize (a few clock cycles)
3. Compare received and transmitted data and count errors. Do it for some period of time (adjustable).
4. Repeat steps 1-4 for all 32 delay steps
5. Output error counters through the UART
6. Wait

The physical loopback has to be connected between `JXADC.7` and `JXADC.8` pins.

Example UART output:

```
...
0027F_000277_00026D_00025C_00027C_00028B_000265_000275_000265_000271_000275_000255_
↪00027A_000280_00027B_000265_00027B_00027A_00025D_000263_000256_00026F_000293_000268_
↪000286_000260_000269_000275_000266_00026D_000273_000272
00281_000271_000273_00026B_000273_000271_00025F_000279_00027D_000283_000266_000279_
↪000274_00025D_000261_000260_00026F_000287_00026E_000289_000261_000267_00027A_00026C_
↪00026D_000270_00026C_00027C_000251_000266_00027A_000283
00271_000255_00027D_000283_000283_00025B_00027E_000271_000263_000259_000262_000270_
↪00027E_00026F_00027D_000267_00026C_00026E_00026E_00027B_00026F_00026D_000279_000250_
↪00026E_00027E_000282_000267_000270_000262_000237_000284
...
```

There are 32 hex numbers separated by "_". Each one correspond to one error counter.

### 4.4.5 Minitests for ISERDES+IDELAY

#### 1. basys3_iserdes_idelay_histogram

This design transmitts a pseudo-random data stream throught one output pin and then receives it through another one with the use of IDELAY and ISERDES. A physical loopback is required. The received data is serialized again (internally) and the received bitstream compared with the transmitted one. This is agnostic to ISERDES configuration. Receive errors are counted separately for each one of 32 possible delay settings of the IDELAY bel. Values of 32 error counters are periodically pinted using the UART as an ASCII string.

There is a control state machine which performs the following sequence once per ~0.5s (adjustable).

1. Set delay of the IDELAY bel.

2. Wait for it to stabilize (a few clock cycles)
3. Compare received and transmitted data and count errors. Do it for some period of time (adjustable).
4. Repeat steps 1-4 for all 32 delay steps
5. Output error counters through the UART
6. Wait

The physical loopback has to be connected between `JXADC.7` and `JXADC.8` pins.

Consider the `JXADC` connector on the Basys3 board as seen when looking at the board edge:

```
 -- -- -- -- -- --
| 6| 5| 4| 3| 2| 1|
 -- -- -- -- -- --
|12|11|10| 9| 8| 7|
 -- -- -- -- -- --
```

- Pin1 - Received signal output, through IDELAY and ISERDES.O (for reference)
- Pin2 - Transmitted signal output (for reference).
- Pin3 - Serialized data clock that the ISERDES operates on (for reference)
- Pin7 - Physical loopback input, connect to Pin8
- Pin8 - Physical loopback output, connect to Pin7

**Important: Make the connection between Pin7 and Pin8 no longer than ~10cm (~4inch).** You can use cables of different length to see how it affects the signal delay.

Example UART output:

```
...
0027F_000277_00026D_00025C_00027C_00028B_000265_000275_000265_000271_000275_000255_
↪00027A_000280_00027B_000265_00027B_00027A_00025D_000263_000256_00026F_000293_000268_
↪000286_000260_000269_000275_000266_00026D_000273_000272
00281_000271_000273_00026B_000273_000271_00025F_000279_00027D_000283_000266_000279_
↪000274_00025D_000261_000260_00026F_000287_00026E_000289_000261_000267_00027A_00026C_
↪00026D_000270_00026C_00027C_000251_000266_00027A_000283
00271_000255_00027D_000283_000283_00025B_00027E_000271_000263_000259_000262_000270_
↪00027E_00026F_00027D_000267_00026C_00026E_00026E_00027B_00026F_00026D_000279_000250_
↪00026E_00027E_000282_000267_000270_000262_000237_000284
...
```

There are 32 hex numbers separated by "_". Each one correspond to one error counter.

An utility script `iserdes_idelay_histogram_receiver.p` can be found in the `utils` subdirectory. It reads and parses data received through UART and prints counter values in decimal.

### 4.4.6 ISERDES minitest for SDR and DDR

#### Description

This test allows to verify that ISEDRES is working on hardware. Tested modes are:

- NETWORKING / SDR
- NETWORKING / DDR

No chaining of two ISERDES bels.

The design serializes data using logic for all tested ISERDES modes. The data is presented onto selected pins. The same pins are used to receive the data which is then fed to ISERDES cells. No physical loopback is required. The clock is routed internally.

The received data is compared against transmitted internally. Errors are indicated using LEDs. The comparator module automatically invokes the bitslip feature of ISERDES (by brutaly testing all possible combinations).

LEDs indicate whether data is being received corectly. When a LED is lit then there is correct reception:

- LED0 - SDR, WIDTH=2
- LED1 - SDR, WIDTH=3
- LED2 - SDR, WIDTH=4
- LED3 - SDR, WIDTH=5
- LED4 - SDR, WIDTH=6
- LED5 - SDR, WIDTH=7
- LED6 - SDR, WIDTH=8
- LED7 - DDR, WIDTH=4
- LED8 - DDR, WIDTH=5
- LED9 - DDR, WIDTH=6
- LED10 - Blinking

The switch SW0 is used as reset.

#### Building

To build the project run the following command and the bit file will be generated.

```
make basys3_iserdes_sdr_ddr.bit
```

### 4.4.7 LiteX minitest

This folder contains minitest for various Litex configurations and target platforms. It is divided into two directories that differ in the CPU configuration.

- min - Minimal configuration - just a CPU + uart targeting Arty and Basys3 boards. The firmware is compiled into the bitstream i.e. the ROM and SRAM memories are instantiated and initialized on the FPGA (no DDR RAM controller).
- base - Linux capable SoC configuration with DDR and Ethernet targeting the Arty.

### Synthesis+implementation

For each variant and platform there are two variants: for Vivado only flow and for Yosys+Vivado flow. In order to run one of them enter the specific directory and run `make`. Once the bitstream is generated and loaded to the board, we should see the test result on the terminal connected to one of the serial ports.

### HDL code generation

The following instructions are for generation of the HDL code

#### 1. Install Litex

- Create an empty directory and clone there the following repos. Be sure to checkout the specific SHA given.
- If you do not want to install LiteX and Migen in your system, setup the Python virtualenv and activate it in the following way:

```
virtualenv litex-env
source litex-env/bin/activate
```

- Install LiteX and Migen packages from the previously cloned repos.

  Run the following command in each repo subdirectory:

```
./setup.py develop
```

- (optional) Hack LiteX HDL generation script to make it think that you have RISC-V toolchain installed (if you don't want to build and install it).
  - Open the file `litex/litex/soc/integration/cpu_interface.py` in your favorite editor.
  - Navigate to the line `53`.
  - Replace it with `("TRIPLE", "riscv32-unknown-elf")`

  This will allow you to generate the HDL code without bothering for compilation of the software.

#### 2. Install RISC-V toolchain

If you don't want to compile the software for the generated LiteX design then you may skip toolchain installation and just hack the LiteX to think that you have it. To do so follow instuctions in the previous point.

- Clone the repo

```
git clone https://github.com/crosstool-ng/crosstool-ng
cd crosstool-ng
git checkout afaf7b9a
```

- Create a file named `ct.config` and put the following content into it:

```
CT_CONFIG_VERSION="3"
CT_EXPERIMENTAL=y
CT_LOCAL_TARBALLS_DIR="${CT_TOP_DIR}/../dl"
CT_PREFIX_DIR="${CT_TOP_DIR}/${CT_TARGET}"
# CT_PREFIX_DIR_RO is not set
CT_ARCH_RISCV=y
```

(continues on next page)

(continued from previous page)

```
CT_ARCH_ARCH="rv32im"
CT_ARCH_ABI="ilp32"
CT_TARGET_VENDOR=""
CT_LIBC_NONE=y
# CT_CC_GCC_LDBL_128 is not set
CT_DEBUG_GDB=y
# CT_GDB_CROSS_PYTHON is not set
CT_ALLOW_BUILD_AS_ROOT=y
CT_ALLOW_BUILD_AS_ROOT_SURE=y
```

- Build the toolchain. Issue the following commands:

```
export DEFCONFIG=`realpath ct.config`
./bootstrap
./configure --enable-local
make -j`nproc`
./ct-ng defconfig
./ct-ng build.`nproc`
```

### 3. Generate the HDL code

If you have built the RISC-V toolchain then make the PATH point to its binaries:

```
export PATH="crosstool-ng/riscv32-unknown-elf/bin:$(PATH)"
```

The following command will generate HDL code for the LiteX SoC with DRAM and Ethernet support for the Arty board target:

```
cd litex/litex/boards/targets
./arty.py --cpu-type vexriscv --cpu-variant linux --with-ethernet --no-compile-
↪software --no-compile-gateware
```

You can choose which synthesis tool generate the design for. This can be done via the additional `--synth-mode` option of the `arty.py` script. The default is `vivado` but you can change it and specify `yosys`.

Generated code will be placed in the `litex/litex/boards/targets/soc_ethernetsoc_arty` folder.

### 4.4.8 LiteX Litex BaseSoC + LiteDRAM minitest

This folder contains a minitest for the Litex memory controller (LiteDRAM). For checking the memory interface we leverage the fact that the BIOS firmware performs a memory test at startup. The SoC is a Basic LiteX SoC configuration for the Arty board with the VexRiscv core.

#### Synthesis+implementation

There are two variants: for Vivado only flow and for Yosys+Vivado flow. In order to run one of them enter the specific directory and run `make`. Once the bitstream is generated and loaded to the board, we should see the test result on the terminal connected to one of the serial ports.

### 4.4.9 Minitest for OSERDES

#### Description

This test allows to verify that OSERDES is working on hardware. Tested modes are:

- SDR 2:1
- SDR 3:1
- SDR 4:1
- SDR 5:1
- SDR 6:1
- SDR 7:1
- SDR 8:1
- DDR 4:1
- DDR 6:1
- DDR 8:1

No chaining of two OSERDES bels.

A pseudo random sequence of words is generated by a LFSR. The sequence is then serialized by the OSERDES and the output bitstream goes to the output pin. The pin is using 3-state buffer which is constantly on. This allows to read serialized data from the same pin without the need of hardware pin loopback connection.

Simultaneously to the OSERDES operation, the word sequence is serialized internally by the FPGA logic. Both bitstreams are then compared and an error indication signal is generated. In order to mitigate for the OSERDES latency, the reference bitstream is delayed by a number of clock cycles which is adaptively changed.

LEDs indicate whether data is being received corectly. When a LED is lit then there is correct reception:

- LED0 - SDR 2:1
- LED1 - SDR 3:1
- LED2 - SDR 4:1
- LED3 - SDR 5:1
- LED4 - SDR 6:1
- LED5 - SDR 7:1
- LED6 - SDR 8:1

- LED7 - DDR 4:1
- LED8 - DDR 6:1
- LED9 - DDR 8:1
- LED10 - Blinking continuously

The switch SW0 is used as reset.

#### Building

To build the project run the following command and the bit file will be generated.

```
make basys3_oserdes_rates.bit
```

### 4.4.10 FASM Proof of Concept using Vivado Partial Reconfig flow

harness.v is a top-level design that routes a variety of signal into a black-box region of interest (ROI). Vivado's Partial Reconfiguration flow (see UG909 and UG947) is used to implement that design and obtain a bitstream that configures portions of the chip that are currently undocumented.

Designs that fit within the ROI are written in FASM and merged with the above harness into a bitstream with fasm2frame and xc7patch. Since writting FASM is rather tedious, rules are provided to convert Verilog ROI designs into FASM via Vivado.

#### Usage

make rules are provided for generating each step of the process so that intermediate forms can be analyzed. Assuming you have a .fasm file, invoking the %_hand_crafted.bit rule will generate a merged bitstream:

```
make foo.hand\_crafted.bit # reads foo.fasm
```

#### Using Vivado to generate .fasm

Vivado's Partial Reconfiguration flow can be used to synthesize and implement a ROI design that is then converted to .fasm. Write a Verilog module that *exactly* matches the roi blackbox model in the top-level design. Note that even the name of the module must match exactly. Assuming you have created that design in my_roi_design.v, 'make my_roi_design_hand_crafted.bit' will synthesize and implement the design with Vivado, translate the resulting partial bitstream into FASM, and then generate a full bitstream by patching the harness bitstream with the FASM. non_inv.v is provided as an example ROI design for this flow.

### 4.4.11 PICORV32-v Minitest

#### Purpose

Unknown bits CPU synthesis test (Vivado synthesis + Vivado PnR)

#### Result

### 4.4.12 PICORV32-y Minitest

#### Purpose

Unknown bits CPU synthesis test (Yosys synthesis + Vivado for PnR)

#### Result

### 4.4.13 PLLE2_ADV minitest

#### Description

This test verifies operation of the `PLLE2_ADV` primitive. The PLL is configured to output clocks using the following dividers:

- CLKOUT0: 16/16
- CLKOUT1: 16/32
- CLKOUT2: 16/48
- CLKOUT3: 16/64
- CLKOUT4: 16/80
- CLKOUT5: 16/96

The input clock can be swtched between 100MHz and 50MHz using the `sw[1]` switch. The 50MHz clock is generated using simple divider implemented in logic.

Clocks from the PLL are further divided by 2^21 and then fed to LEDs 0:5. PLL lock indicator is connected to LED 15. The switch `sw[0]` provides reset signal to the PLL.

#### Building

To build the project run the following command and the bit file will be generated.

```
make basys3_plle2_adv.bit
```

### 4.4.14 ROI_HARNESS Minitest

#### Purpose

Creates an harness bitstream which maps peripherals into a region of interest which can be reconfigured.

The currently supported boards are;

- Artix 7 boards;
- Basys 3
- Arty A7-35T
- Zynq boards;

- Zybo Z7-10

The following configurations are supported;

- SWBUT - Harness which maps a board's switches, buttons and LEDs into the region of interest (plus clock).
- PMOD - Harness which maps a board's PMOD connectors into the region of interest (plus a clock).
- UART - Harness which maps a board's UART

"ARTY-A7-SWBUT" # 4 switches then 4 buttons A8 C11 C10 A10 D9 C9 B9 B8 # 4 LEDs then 4 RGB LEDs (green only) H5 J5 T9 T10 F6 J4 J2 H6

```
    # clock
E3
```

"ARTY-A7-PMOD" # CLK on Pmod JA G13 B11 A11 D12 D13 B18 A18 K16 # DIN on Pmod JB E15 E16 D15 C15 J17 J18 K15 J15 # DOUT on Pmod JC U12 V12 V10 V11 U14 V14 T13 U13

"ARTY-A7-UART" # RST button and UART_RX C2 A9 # LD7 and UART_TX T10 D10 # 100 MHz CLK onboard E3

"BASYS3-SWBUT" # Slide switch pins V17 V16 W16 W17 W15 V15 W14 W13 V2 T3 T2 R3 W2 U1 T1 R2 # LEDs pins U16 E19 U19 V19 W18 U15 U14 V14 V13 V3 W3 U3 P3 N3 P1 L1

```
    # UART
    B18 # ins
    A18 # outs

    # 100 MHz CLK onboard
    W5
```

"ZYBOZ7-SWBUT" # J15 - UART_RX - JE3 # G15 - SW0 # K18 - BTN0 # K19 - BTN1 J15 G15 K18 K19

```
    # H15 - UART_TX - JE4
    # E17 - ETH PHY reset (active low, keep high for 125 MHz clock)
    # M14 - LD0
    # G14 - LD2
    # M15 - LD1
    # D18 - LD3

    # 125 MHz CLK onboard
    K17
```

### Quickstart

```
source settings/artix7.sh
cd minitests/roi_harness
source arty-swbut.sh
make clean
make copy
```

#### How it works

Basic idea:

- LOC LUTs in the ROI to terminate input and output routing
- Let Vivado LOC the rest of the logic
- Manually route signals in and out of the ROI enough to avoid routing loops into the ROI
- Let Vivado finish the rest of the routes

There is no logic outside of the ROI in order to keep IOB to ROI delays short Its expected the end user will rip out everything inside the ROI

To target Arty A7 you should source the artix DB environment script then source arty.sh

To build the baseline harness:

```
./runme.sh
```

To build a sample Vivado design using the harness:

```
XRAY_ROIV=roi_inv.v XRAY_FIXED_XDC=out_xc7a35tcpg236-1_BASYS3-SWBUT_roi_basev/fixed_
↪noclk.xdc ./runme.sh
```

Note: this was intended for verification only and not as an end user flow (they should use SymbiFlow)

To use the harness for the basys3 demo, do something like:

```
python3 demo_sw_led.py out_xc7a35tcpg236-1_BASYS3-SWBUT_roi_basev 3 2
```

This example connects switch 3 to LED 2

#### Result

### 4.4.15 Minitests for SRLs

This is a minitest for various SRL configurations.

Uses Yosys to generate EDIF which is then P&R'd by Vivado. The makefile also invokes bit2fasm and segprint

### 4.4.16 Timing minitest

This minitest uses Vivado to compile a design and extracts the relevant timing metadata from the design, e.g. what are the nets and how was the design routed.

For each clock path, the final timing is provided for each of the 4 corners of analysis.

From the timing metadata, `create_timing_worksheet_db.py` creates a worksheet breaking down the interconnect timing calculation and generating a final comparision between the reduced model implemented in prjxray and the Vivado timing results.

#### Model quality

The prjxray timing handles most nets +/- 1.5% delay. The large exception to this is clock nets, which appear to use a table lookup that is not understood at this time.

#### Running the model

The provided Makefile will by default compile all examples. It a specific design family is desired, the family name can be provided. If a specific design within a family is desired, use `<family name>_<iter>`.

Example:

```
# Build all variants of the DFF loopback test
make dff
# Build only DESIGN_NAME=dff ITER=63
make dff_63
```

### 4.4.17 Zynq7 EMIO minitest

This is a simple test of PS -> PL interface for Zynq7. Works on the ZYBO Z7 board with xc7z020 but should also work for xc7z010.

The PS firmware is bare metal. Upon start it enables MIO7 as output as well as PS <-> PL level shifters. Next it implements a blinking led on MIO7 and counter on GPIO bank 2. The bank 2 is connected to EMIOGPIOO[31:0] signals of the PS7 instance in the PL logic design.

The PL design "instantiates" the PS7 and connects EMIOGPIOO[3:0] to LEDs LD0-LD3 but through XOR gates controlled by push buttons BTN0-BTN3.

### 4.4.18 Building & loading

#### PS

Run `make firmware` to compile the firmware. Then run `make run` to upload it to Zybo. You must have Xilinx XSCT installed and pointed to in the environment. You can also just run `make` to execute those two above steps. Upon loading the LED LD4 should start blinking.

#### PL

Run `make top.bit` to generate the bitstream. Upload it to the Zybo AFTER uploading and running PS firmware. You can use eg. xc3sprog with the following command `xc3sprog -c jtaghs1 -p 1 top.bit`. Once done LEDs LD0-LD3 should begin dusplaying 4 LSBs of the counter.

## 4.5 Tools

*SymbiFlow/prjxray/tools/*

Here, you can find various programs to work with bitstreams, mainly to assist building fuzzers.

## 4.6 Guide to adding a new device to an existing family

This documents how to add support for a new device. The running example is the addition of the xc7a100t device to the Artix-7 family.

Adding a new device to an existing family is much simpler than adding a new family, since the building blocks (tiles) are already known. There are just more or fewer of them, arranged differently. No new fuzzers are needed. You just need to rerun some fuzzers for the new device to understand how the tiles are connected to each other and to IOs.

*If you are* **just** *adding a new package for a device that is already supported, you can skip Steps 2 through 4.*

Note: Since this guide was written, the xc7a100t has become the primary device in the database, not a secondary device as it was when it was originally added. Therefore the files currently in the repo don't match what is described here. But if you look at the original PRs, they match what is described in the examples here.

The main PR from the example is #1313. Followup fixes for problems revealed during testing are #1334 and #1336.

### 4.6.1 Step 0

Fork a copy of https://github.com/SymbiFlow/prjxray on GitHub (go to the page, click "Fork" button, select your own workspace).

Clone your fork, and make a new branch, with a name related to the new device/package:

```
git clone git@github.com:<yourUserID>/prjxray.git
cd prjxray
git checkout -b <new_branch_name>
```

### 4.6.2 Step 1

Follow the Project X-Ray developer setup instructions in the documentation, up through Step 7 and choose Option 1 (invoke the `./download-latest-db.sh` script). This script will clone the official prjxray-db database under `database/`. The following steps will make changes under this directory. You may want to put these changes on your own fork of `prjxray-db` for testing. This is explained at the end, under "Database Updates".

### 4.6.3 Step 2

Add a new settings file. Usually you will start with an existing settings file and modify it. Assuming you're in prjxray/,

```
cp settings/<baseline_device>.sh settings/<new_device>.sh
git add settings/<new_device>.sh
```

Example:

```
cp settings/artix7_200t.sh settings/artix7_100t.sh
git add settings/artix7_100t.sh
```

Update the following values in the new settings file:

- XRAY_PART – Important: choose a package that is fully bonded (typically the one with the largest number of pins). If the part that you're actually interested in is different (with fewer bonded pins), it will be handled later. In the running example, the actual part of interest was the xc7a100tcsg324, since that is on the Arty A7-100T board. But here, the xc7a100tfgg676 part is used; the xc7a100tcsg324 is handled later.
- XRAY_ROI_TILEGRID – modify the bounding boxes to be a tight fit on your new part.
- XRAY_IOI3_TILES – These tiles need special handling for an irregularity in Xilinx 7-series FPGAs. See the comments in the 005 fuzzer for more information.
- XRAY_PIN_00 – this must be a clock pin. You can look at the device in the Vivado GUI interactively (click on IOs and check their properties until you find one with IS_CLOCK=true), or run a small clocked design in Vivado and see which pin is assigned to 'clk'.
- XRAY_PIN_01 and on – these should be normal data pins on the device.

This is what the new settings file looked like in the example.

Source this new settings file:

```
source settings/<new_device>.sh
```

### 4.6.4 Step 3

Edit the top Makefile

- Update the Makefile by adding the new device to the correct list, so that the Makefile generates targets for the new device (used in Step 4). <new_device> is the basename of the new settings file that you just created.

```
<FAMILY>_PARTS=<existing_devices> <new_device>
```

- In our running example, we add artix7_100t to ARTIX_PARTS:

```
ARTIX_PARTS=artix7_200t artix7_100t
```

### 4.6.5 Step 4

Make sure you've sourced your new device settings file (see the end of step 2). Now it is time to run some fuzzers to figure out how the tiles on your new device are connected.

Make the following target, with <new_device> as above, and setting the parallelism factor -j<n> appropriate for the number of cores your host has. The make job can benefit from large numbers of cores.

```
make -j<n> MAX_VIVADO_PROCESS=<n> db-part-only-<new_device>
```

Again, <new_device> must match the base name of the new settings file that was added. For example,

```
make -j32 MAX_VIVADO_PROCESS=32 db-part-only-artix7_100t
```

- It should run fuzzers 000, 001, 005, 072, 073, 074, and 075.
- 005 will take a long time. Using multiple cores will help.
- 074 *will fail* the first time, since it hasn't been told to ignore certain wires.

  - After it fails, go to the build directory `cd fuzzers/074-dump_all/build_<XRAY_PART>` (this is the `XRAY_PART` from the new settings script; in our example, the build directory is `fuzzers/074-dump_all/build_xc7a100tfgg676-1/`).
  - Run `python3 ../analyze_errors.py --output_ignore_list > new-ignored`
  - Inspect and compare `new-ignored` against existing ignored wire files in `../ignored_wires/`.
  - If it looks good, copy it to an appropriately-named file: `cp new-ignored ../ignored_wires/artix7/<XRAY_PART>_ignored_wires.txt` (in our example, it is `../ignored_wires/artix7/xc7a100tfgg676-1_ignored_wires.txt`).
  - Add it: `git add ../ignored_wires/artix7/<XRAY_PART>_ignored_wires.txt`
- Return to prjxray/ directory, and clean up 074 to prepare for the rerun: `make -C fuzzers/074-dump-all clean`
- Rerun the top make command, e.g. `make -j32 MAX_VIVADO_PROCESS=32 db-part-only-artix7_100t`

### 4.6.6 Step 5

The next task is handling the extra parts – those not fully bonded out. These are usually the parts you actually have on the boards you buy.

- Add a new entry in the appropriate 'harness' section for any alternative packages (typically with fewer pins, in this example, 324 versus 676). If any `XRAY_PIN_<XX>` values you listed in the settings file are not bonded out on the new part, you must specify alternatives. In this example, we need to specify a new clock pin, `XRAY_PIN_00=N15`. Here, `XRAY_PART` is the extra part, and `XRAY_EQUIV_PART` is the original, fully-bonded version:

```
db-extras-artix7-harness:
    +source settings/artix7.sh && \
          XRAY_PIN_00=J13 XRAY_PIN_01=J14 XRAY_PIN_02=K15 XRAY_PIN_03=K16 \
          XRAY_PART=xc7a35tftg256-1 XRAY_EQUIV_PART=xc7a50tfgg484-1 \
          $(MAKE) -C fuzzers roi_only
+   +source settings/artix7_100t.sh && \
+         XRAY_PIN_00=N15 \
+         XRAY_PART=xc7a100tcsg324-1 XRAY_EQUIV_PART=xc7a100tfgg676-1 \
+         $(MAKE) -C fuzzers roi_only
    +source settings/artix7_200t.sh && \
          XRAY_PIN_00=V10 XRAY_PIN_01=W10 XRAY_PIN_02=Y11 XRAY_PIN_03=Y12 \
          XRAY_PART=xc7a200tsbg484-1 XRAY_EQUIV_PART=xc7a200tffg1156-1 \
          $(MAKE) -C fuzzers roi_only
```

Make the appropriate harness target (adjusting for your family):

```
make -j32 db-extras-artix7-harness
```

This target will make updates for the extra parts of all of the family devices, not just your new device.

### 4.6.7 Step 6

Do a spot check.

- Check that there are new part directories in the database under the family subdirectory, for example:

```
$ ll database/artix7/xc7a100*
database/artix7/xc7a100tcsg324-1:
total 19884
drwxrwxr-x  2 tcal tcal     4096 Apr 29 08:01 ./
drwxrwxr-x 13 tcal tcal    32768 Apr 29 08:00 ../
-rw-rw-r--  1 tcal tcal    10364 Apr 29 08:00 package_pins.csv
-rw-rw-r--  1 tcal tcal    32142 Apr 29 08:01 part.json
-rw-rw-r--  1 tcal tcal    22440 Apr 29 08:01 part.yaml
-rw-rw-r--  1 tcal tcal  8601612 Apr 29 08:01 tileconn.json
-rw-rw-r--  1 tcal tcal 11648042 Apr 29 08:01 tilegrid.json

database/artix7/xc7a100tfgg676-1:
total 19892
drwxrwxr-x  2 tcal tcal     4096 Apr 29 02:03 ./
drwxrwxr-x 13 tcal tcal    32768 Apr 29 08:00 ../
-rw-rw-r--  1 tcal tcal    16645 Apr 28 22:16 package_pins.csv
-rw-rw-r--  1 tcal tcal    32165 Apr 28 22:17 part.json
-rw-rw-r--  1 tcal tcal    22440 Apr 28 22:17 part.yaml
-rw-rw-r--  1 tcal tcal  8601612 Apr 29 02:03 tileconn.json
-rw-rw-r--  1 tcal tcal 11648042 Apr 28 22:37 tilegrid.json
```

In this case, the tile grid is the same size since it's the same chip, but the size of the package pins files differs, since there are different numbers of bonded pins.

Note: These changes/additions under `database/` do *not* get checked in. They are in the `prjxray-db` repo. This spot check is to make sure that your changes in `prjxray` will do the right thing when the official database is fully rebuilt. See "Database Updates" below for more information.

### 4.6.8 Step 7

Assuming everything looks good, commit to your `prjxray` fork/branch. You should have a new file under settings/, a new ignored_wires file, and a modified Makefile (see the initial PR of the example for reference).

```
git add Makefile settings/artix7_100t.sh
git status
git commit --signoff
```

### 4.6.9 Step 8

Push to GitHub:

```
git push origin <new_branch_name>
```

Then make a pull request. Navigate to the GitHub page for your `prjxray` fork/branch, and click the "New pull request" button. Making the pull request will kick off continuous integration tests. Watch the results and fix any issues.

### 4.6.10 Database Updates

The process above (steps 4 and 5) will create some new files and modify some existing files under database/, which is a different repo, `prjxray-db`.

To test these changes before the next official prjxray-db gets built (and even before your PR on prjxray is merged), you can put these changes on your own fork of prjxray-db, and then test them in the context of symbiflow-arch-defs.

To put the db updates on your own fork, create your fork of https://github.com/SymbiFlow/prjxray-db if you haven't already. Then follow one of the approaches suggested in the checked solution of this StackOverflow post.

You are NEVER going to send a pull request on *prjxray-db*. The database is always rebuilt from scratch. After your changes on prjxray are merged, they will reflected in the next prjxray-db rebuild. The changes submitted to your prjxray-db fork are only for your own testing.

To use your new repo/branch under symbiflow-arch-defs/third_party/prjxray-db/, you will need to change the submodule reference to point to your fork/branch of `prjxray-db`.

CHAPTER

FIVE

# DATABASE

This section documents how the bitstream database is represented in project X-Ray. The database is a collection of plain text files, using either simple line-based syntax or JSON format.

## 5.1 Description

The main goal of the X-Ray Project is to provide information about Xilinx 7-Series FPGA internals. All obtained chip data is stored in the project's database and is used by the Architecture Definitions project to produce a bitstream for the chosen 7-Series FPGA chip.

The database files are generated by the *fuzzers* and are located in the `database` directory. Each supported chip architecture has its own set of files, which are located in `database/<device_arch>/`. The database can be quite huge, however it consists only of a few file types. Some of them are common for the whole 7-Series architecture, but some of them are part specific.

Files common for whole 7-Series family:

- `mask_*`
- `ppips_*`
- `segbits_*`
- `site_type_*`
- `tile_type_*`
- `timings/*`

The files specific to a given part are located in a separate directory which is named after the FPGA part name i.e *xc7a35tcpg236-1* or *xc7a50tfgg484-1*.

Files specific for the particular FPGA part:

- `package_pins.csv`
- `part.json`
- `part.yaml`
- `tileconn.json`
- `tilegrid.json`

## 5.2 Common database files

This section contains a description of *database* files that are common for all Xilinx series 7 chip architectures.

### 5.2.1 segbits files

The *segbits files* are generated for every FPGA *tile* type. They store the information about the combinations of bits in the bitstream that are responsible for enabling different features inside the *tile*. The features can be related to enabling some part of the primitive, setting some initial state of the block, configuring pin pull-up on output pins, etc.

Besides the features provided in this file that can be enabled, the FPGA chip also has the default configuration. Due to that sometimes there is no need for affecting the default configuration.

#### Naming convention

The naming scheme for the segbits files is the following:

```
segbits_<tile>.db
```

Note that auxiliary `segbits_<tile>.origin_info.db` files provide additional information about the *fuzzer*, which produced the *database* file. This file is optional.

Every *tile* is configured at least by one of three configurational buses mentioned in the *Configuration Section*. The default bus is called `CLB_IO_CLK`. If the *tile* can also be configured by another bus, it has additional `segbits_<tile>.<bus_name>.db` related to that bus.

Example files:

- `segbits_dsp_r.db`
- `segbits_bram_l.db` (configured with default `CLB_IO_CLK` bus)
- `segbits_bram_l.block_ram.db` (configured with `BLOCK_RAM` bus)

#### File format

The file consists of lines containing the information about the feature and the list of bits that should be enabled/disabled to provide the feature's functionality:

```
<feature> <bit_list>
```

where:

- `<feature>` is of the form `<feature_name>.<feature_addr>`
- `<bit_list>` is the list of bits. Each bit is of the form `<frame_address_offset>_<bit_possition>`. If the bit has the `!` mark in front of it, that means it should be set to **0** for feature configuration, otherwise it should be set to **1**.

The names of the features are arbitrary. However, the naming convention allows for quick identifaction of the functionality that is being configured. The feature names are used during the generation of the FASM file.

### Feature naming conventions

#### PIPs

The <feature> names for interconnect *PIPs* are stored in the segbits_int_l.db and segbits_int_r.db database files. The features that enable interconnect *PIPs* have the following syntax:

```
<tile_type>.<destination_wire>.<source_wire>.
```

For example, consider the following entry in segbits_int_l.db:

```
INT_L.NL1BEG1.NN6END2 07_32 12_33
```

#### CLBs

The <feature> names for CLB tiles use a dot-separated hierarchy.

For example:

```
CLBLL_L.SLICEL_X0.ALUT.INIT[00]
```

This entry documents the initialization bits of the *LSB LUT* for the *ALUT* in the *SLICEL_X0* within a *CLBLL_L tile*.

#### Example

Below there is a part of the segbits_liob33_l.db file for the *artix7* architecture. The file describes the *CLBLL tile*:

```
<...>
LIOB33.IOB_Y0.IBUFDISABLE.I 38_82
LIOB33.IOB_Y0.IN_TERM.NONE !38_120 !38_122 !39_121 !39_123
LIOB33.IOB_Y0.IN_TERM.UNTUNED_SPLIT_40 38_120 38_122 39_121 39_123
LIOB33.IOB_Y0.IN_TERM.UNTUNED_SPLIT_50 38_120 38_122 !39_121 39_123
LIOB33.IOB_Y0.IN_TERM.UNTUNED_SPLIT_60 38_120 !38_122 !39_121 39_123
LIOB33.IOB_Y0.INTERMDISABLE.I 39_89
LIOB33.IOB_Y0.LVTTL.DRIVE.I24 38_64 !38_112 !38_118 38_126 39_65 39_117 39_119 !39_
↪125 !39_127
LIOB33.IOB_Y0.PULLTYPE.KEEPER 38_92 38_94 !39_93
LIOB33.IOB_Y0.PULLTYPE.NONE !38_92 38_94 !39_93
LIOB33.IOB_Y0.PULLTYPE.PULLDOWN !38_92 !38_94 !39_93
LIOB33.IOB_Y0.PULLTYPE.PULLUP !38_92 38_94 39_93
<...>
```

For example, the line:

```
LIOB33.IOB_Y0.PULLTYPE.PULLUP !38_92 38_94 39_93
```

means that the feature LIOB33.IOB_Y0.PULLTYPE.PULLUP will be set by clearing bit !38_92 and setting bits 38_94 and 39_93.

Generally, the <feature> name is linked with its functionality. For example, LIOB33.IOB_Y0.PULLTYPE.PULLUP means that in the LIOB33 *tile*, in IOB_Y0 site the *pull type* will be set to *PULLUP*. This simply means that all pins belonging to this particular IOB will be configured with pull-up.

### 5.2.2 site_type files

The *site_type files* are generated for every FPGA *site* type. They store the information about the pins and *PIPs* of the *site*.

#### Naming convention

The naming scheme for the *site* type files is the following:

```
site_type_<site>.json
```

Example files:

- `site_type_IDELAYE2.json`
- `site_type_PLLE2_ADV.json`
- `site_type_SLICEL.json`

#### File format

The *site* type files are JSON files with the following scheme:

```
{
    "site_pins": {
        "<PIN_NAME>": {
            "direction": "<DIR>"
        },
        <...>
    },
    "site_pips": {
        "<PIP_NAME>": {
            "from_pin": "<PIN_NAME>",
            "to_pin": "<PIN_NAME>"
        }
    },
    "type": "<TYPE>"
}
```

where:

- <*PIN_NAME*> - specifies the *site* pin name
- <*PIP_NAME*> - specifies the *site pip* name
- <*DIR*> - is a direction of a pin (either **IN** or **OUT**)
- <*TYPE*> - specifies the *site* type

The `"site_pins"` section describes the input pins of a *site* and its directions. The `"site_pips"` describes the *PIPs* inside the *site* and which wires they can connect.

### Example

Below there is a part of `site_type_SLICEL.json` file for the *artix7* architecture:

```
{
   "site_pins": {
      "A": {
         "direction": "OUT"
      },
      "A1": {
         "direction": "IN"
      },
      "A2": {
         "direction": "IN"
      },
      "A3": {
         "direction": "IN"
      },
      "A4": {
         "direction": "IN"
      },
      "A5": {
         "direction": "IN"
      },
      "A6": {
         "direction": "IN"
      },
      <...>
   },
   "site_pips": {
      "A5FFMUX:IN_A": {
         "from_pin": "IN_A",
         "to_pin": "OUT"
      },
      "A5FFMUX:IN_B": {
         "from_pin": "IN_B",
         "to_pin": "OUT"
      },
      "A5LUT:A1": {
         "from_pin": "A1",
         "to_pin": "O5"
      },
      "A5LUT:A2": {
         "from_pin": "A2",
         "to_pin": "O5"
      },
      "A5LUT:A3": {
         "from_pin": "A3",
         "to_pin": "O5"
      },
      "A5LUT:A4": {
         "from_pin": "A4",
         "to_pin": "O5"
      },
      "A5LUT:A5": {
         "from_pin": "A5",
         "to_pin": "O5"
      },
```

(continues on next page)

(continued from previous page)

```
        <...>
    },
    "type": "SLICEL"
}
```

Compare the description with the Xilinx documentation of that *site*.

### 5.2.3 tile_type files

The *tile_type files* are generated for every FPGA *tile* type. They store the information about the *tile* configuration, its *PIPs*, *sites*, wires and their properties.

#### Naming convention

The naming scheme for the segbits files is the following:

```
tile_type_<tile>.json
```

Example files:

- `tile_type_INT_L.json`
- `tile_type_BRAM_L.json`
- `tile_type_HCLK_CLB.json`

#### File format

The *tile* type files are JSON files with the following shape:

```
{
    "pips": {
        "<PIP_NAME>": {
            "can_invert":' "<BOOL>",
            "dst_to_src": {
                "delay": [
                    "<FAST_MIN>",
                    "<FAST_MAX>",
                    "<SLOW_MIN>",
                    "<SLOW_MAX>"
                ],
                "in_cap": "<IN_CAPACITANCE>",
                "res": "<RESISTANCE>"
            },
            "dst_wire": "<WIRE_NAME>",
            "is_directional": "<BOOL>",
            "is_pass_transistor": <BOOL>,
            "is_pseudo": "0",
            "src_to_dst": {
                "delay": [
                    "<FAST_MIN>",
                    "<FAST_MAX>",
                    "<SLOW_MIN>",
                    "<SLOW_MAX>"
```

(continues on next page)

(continued from previous page)

```
                ],
                "in_cap": "<IN_CAPACITANCE>",
                "res": "<RESISTANCE>"
            },
            "src_wire": "<WIRE_NAME>"
        },
    },
    "sites": [
        {
            "name": "<SITE_NAME>",
            "prefix": "<SITE_PREFIX>",
            "site_pins": {
                "<SITE_PIN_NAME>": {
                    "cap": "<CAPACITY>",
                    "delay": [
                        "<FAST_MIN>",
                        "<FAST_MAX>",
                        "<SLOW_MIN>",
                        "<SLOW_MAX>"
                    ],
                    "wire": "<WIRE_NAME>"
                },
                <...>
    ],
    "tile_type": "<TILE_TYPE>",
    "wires": {
        "<WIRE_NAME>": {
            "cap": "<WIRE_CAPACITY>",
            "res": "<WIRE_RESISTANCE>"
        },
        <...>
    },
}
```

### "pips" section

The "pips" section describes all *PIPs* in the *tile*. Every *PIP* has its name - `"<PIP_NAME>"` and may be characterized by the following attributes:

- `can_invert` - takes a value which can be either **1** or **0**. It defines whether the *PIP* has an inverter on its output or not.
- `dst_to_src` - information about the connection in the direction from destination to source. It describes the following properties of the connection:
  - `delay` - a four-element list, which contain information about the delay of pins. First two elements are related to the *fast corner* of the technological process, the second two elements to the *slow corner*. The first element of the pair is the minimum value of the corner, the second describes the maximum value. They are given in us (nanoseconds).
  - `in_cap` - the input capacitance of the *PIP* in uF (microfarads).
  - `res` - the resistance of the *PIP* in m (miliohms).
- `dst_wire` - the destination wire name
- `is_directional` - contains the information whether *PIP* is directional.

- `is_pass_transisstor` - contains the information whether *PIP* acts as a pass transistor
- `is_pseudo` - contains the information whether *PIP* is a pseudo-PIP
- `src_to_dst` - contains the information about the connection in the direction from source to destination. It is described by the same set of properties as `dst_to_src` section.

### "sites" section:

The "sites" section describes all *sites* in the *tile*. Every *site* may be characterized by the following attributes:

- `name` - location in the *tile* grid
- `prefix` - the type of the *site*
- `site_pins` - describes the pins that belong to the *site*. Every pin has its name - `<PIN_NAME>` and may be described by the following attributes:
  - `cap` - pin capacitance in uF (microfarads).
  - `delay` - a four-element list, which contain information about the delay of pins. First two elements are related to the *fast corner* of the technological process, the second two elements to the *slow corner*. The first element of the pair is the minimum value of the corner, the second describes the maximum value. They are given in us (nanoseconds).
  - `wire` - wire associated with the pin
- `type` - indicates the type of the site
- `x_coord` - describes *x* coordinate of the site position inside the tile
- `y_coord` - describes the *y* coordinate of the site position inside the tile

### "wires" section

The "wires" section describes the wires located in the *tile*. Every wire has its name - `<WIRE_NAME>` and may be characterized by the following attributes:

- `cap` - wire capacitance in uF (microfarads)
- `res` - wire resistance in m (miliohms).

### Other

- `tile_type` - indicates the type of the tile

### Example

Below there is a part of `tile_type_BRAM_L.json` for the *artix7* architecture:

```
{
   "pips": {
      "BRAM_L.BRAM_ADDRARDADDRL0->>BRAM_FIFO18_ADDRATIEHIGH0": {
         "can_invert": "0",
         "dst_to_src": {
             "delay": [
                 "0.038",
```

(continues on next page)

(continued from previous page)


```
                "0.046",
                "0.111",
                "0.134"
            ],
            "in_cap": "0.000",
            "res": "737.319"
        },
        "dst_wire": "BRAM_FIFO18_ADDRATIEHIGH0",
        "is_directional": "1",
        "is_pass_transistor": 0,
        "is_pseudo": "0",
        "src_to_dst": {
            "delay": [
                "0.038",
                "0.046",
                "0.111",
                "0.134"
            ],
            "in_cap": "0.000",
            "res": "737.319"
        },
        "src_wire": "BRAM_ADDRARDADDRL0"
    },
    <...>
    "BRAM_L.BRAM_IMUX12_1->BRAM_IMUX_ADDRARDADDRU8": {
        "can_invert": "0",
        "dst_to_src": {
            "delay": null,
            "in_cap": null,
            "res": "0.000"
        },
        "dst_wire": "BRAM_IMUX_ADDRARDADDRU8",
        "is_directional": "1",
        "is_pass_transistor": 1,
        "is_pseudo": "0",
        "src_to_dst": {
            "delay": null,
            "in_cap": null,
            "res": "0.000"
        },
        "src_wire": "BRAM_IMUX12_1"
    },
    <...>
},
"sites": [
    {
        "name": "X0Y0",
        "prefix": "RAMB18",
        "site_pins": {
            "ADDRARDADDR0": {
                "cap": "0.000",
                "delay": [
                    "0.000",
                    "0.000",
                    "0.000",
                    "0.000"
                ],
```


(continues on next page)

(continued from previous page)


```
                "wire": "BRAM_FIFO18_ADDRARDADDR0"
            },
            <...>
            "WRERR": {
                "delay": [
                    "0.000",
                    "0.000",
                    "0.000",
                    "0.000"
                ],
                "res": "860.0625",
                "wire": "BRAM_RAMB18_WRERR"
            },
            <...>
        },
        "type": "RAMB18E1",
        "x_coord": 0,
        "y_coord": 1
    }
],
"tile_type": "BRAM_L",
"wires": {
    "BRAM_ADDRARDADDRL0": null,
    "BRAM_ADDRARDADDRL1": null,
    "BRAM_ADDRARDADDRL2": null,
    "BRAM_ADDRARDADDRL3": null,
    "BRAM_EE2A0_0": {
        "cap": "60.430",
        "res": "268.920"
    },
    <...>
    "BRAM_EE2A0_1": {
        "cap": "60.430",
        "res": "268.920"
    },
    <...>
 }
```


### 5.2.4 ppips files

The *ppips files* are generated for every FPGA *tile* type. They store the information about the pseudo-PIPs, inside the tile.

Programable Interconnect Point (*PIP*) is a connection inside the *tile* that can be enabled or disabled. Pseudo PIPs appear as standard *PIPs* in the Vivado tool, but they do not have actual configuration bit pattern in segbits files (they are not configurable).

The *ppips files* contains the information which *PIPs* <*PIP*> do not have configuration bits, which allows the tools to not generat error in that situation. On the other hand this information is used to indicate that the connection between wires is always on.

### Naming convention

The naming scheme for the PPIPs files is the following:

```
ppips_<tile>.db
```

For example:

- `ppips_dsp_l.db`
- `ppips_clbll_l.db`
- `ppips_bram_int_interface_l.db`

### File format

The file contains one entry per pseudo-PIP, each with one of the following three tags: `always`, `default` or `hint`. The entries are of the form::

```
<ppip_location> <tag>
```

The tag `always` is used for pseudo-PIPs that are actually always-on, i.e., that are permanent connections between two wires.

The tag `default` is used for pseudo-PIPs that represent the default behavior if no other driver has been configured for the destination net (all default pseudo-PIPs connect to the VCC_WIRE net).

The tag `hint` is used for PIPs that are used by Vivado to tell the router that two logic slice outputs drive the same value, i.e., behave like they are connected as far as the routing process is concerned.

### Example

Below there is a part of artix7 `ppips_clbll_l.db` file:

```
<...>
CLBLL_L.CLBLL_L_A.CLBLL_L_A6 hint
CLBLL_L.CLBLL_L_AMUX.CLBLL_L_A hint
CLBLL_L.CLBLL_L_AX.CLBLL_BYP0 always
CLBLL_L.CLBLL_L_B.CLBLL_L_B1 hint
CLBLL_L.CLBLL_L_B.CLBLL_L_B2 hint
CLBLL_L.CLBLL_L_B.CLBLL_L_B3 hint
CLBLL_L.CLBLL_L_B.CLBLL_L_B4 hint
<...>
```

The `<ppip_location>` name is arbitrary. However, the naming convention is similar to the one in the Vivado tool, which allows for quick identification of their role in the FPGA chip.

### 5.2.5 mask files

The *mask files* are generated for every FPGA *tile* type. They store the information which bits in the bitstream can configure the given *tile* type.

#### Naming convention

The naming scheme for the mask files is the following:

```
mask_<tile>.db
```

Note that auxiliary `mask_<tile>.origin_info.db` files provide additional information about the *fuzzer*, which produced the *database* file. This file is optional.

Every *tile* is configured at least by one of three configurational buses mentioned in the *Configuration Section*. The default bus is called `CLB_IO_CLK`. If the *tile* can also be configured by another bus, it has an additional `mask_<tile>.<bus_name>.db` related to that bus.

For example:

- `mask_dsp_r.db`
- `mask_bram_l.db` (configured with default `CLB_IO_CLK` bus)
- `mask_bram_l.block_ram.db` (configured with `BLOCK_RAM` bus)

#### File format

The file consists of records that describe the configuration bits for the particular *tile* type. Each entry inside the file is of the form:

```
bit <frame_address_offset>_<bit_position>
```

This means that the *tile* can be configured by a bit located in the *frame* at the address `<base_frame_addr> + <frame_address_offset>`, at position `<tile_offset> + <bit_position>`. Information about `<base_frame_address>` and `<tile_offset>` can be taken from part specific `tilegrid.json` file.

#### Example

Below there is a part of artix7 `mask_clbll_l.db` file describing a FPGA *CLBLL* *tile*:

```
<...>
bit 00_61
bit 00_62
bit 00_63
bit 01_00
bit 01_01
bit 01_02
<...>
```

The line `bit 01_02` means that the *CLBL_LL* *tile* can be configured by the bit located in the *frame* at the address `<base_frame_address> + 0x01`, at position `<tile_offset> + 0x2`.

The `tilegrid.json` is a file specific to a given chip package. For *xc7a35tcpg236-1* we can find an exemplary *CLBLL_L* entry:

```
"CLBLL_L_X2Y0": {
    "bits": {
        "CLB_IO_CLK": {
            "baseaddr": "0x00400100",
            "frames": 36,
            "offset": 0,
            "words": 2
        }
    },
    "clock_region": "X0Y0",
    "grid_x": 10,
    "grid_y": 155,
    "pin_functions": {},
    "sites": {
        "SLICE_X0Y0": "SLICEL",
        "SLICE_X1Y0": "SLICEL"
    },
    "type": "CLBLL_L"
},
```

The `<base_frame_addr>` can be found as a argument of the *"baseaddr"* key and for *CLBLL_L_X2Y0* *tile* it is equal to `0x00400100`. The `<tile_offset>` on the other hand is an argument of the *"offset"* key. Here it is equal to 0.

Finally, we are able to compute the bit location associated with the `bit 01_02` entry.

The configuration bit for this record can be found in the following *frame* address:

```
0x00400100 + 0x01 = 0x00400101
```

Located at the bit position:

```
0x0 + 0x2 = 0x2
```

More about the configuration process and the meaning of the *frame* can be found in the *Configuration Section*.

## 5.3 Part specific database files

This section contains a description of *database* files that are part (chip) specific:

### 5.3.1 tilegrid file

The `tilegrid.json` is a list of all *tiles* in the device. This information is used at various stages of the flow i.e. for *database* generation or creating a *bitstream*. The most important parts of the file are related to *frame* addressing within the *bitstream*, grid and *clock region* location, list of underlying *sites*, or the type of the *tile* itself.

Before diving into this section, it is advised to familiarize yourself with the 7-Series *Bitstream Format* chapter and *Configuration* chapter.

### File format

The file consists of the entries describing every *tile* in the FPGA chip. The file is of the form:

```
{
    "<TILE_NAME>": {
        "bits": {
            "<CONFIGURATION_BUS>": {
                "baseaddr": "<BASE_ADDR>,
                "frames": 28,
                "offset": 97,
                "words": 2
            },
            <...>
    },
    "clock_region": <CLOCK_REGION>,
    "grid_x": <GRID_X>,
    "grid_y": <GRID_Y>,
    "pin_functions": {
        "<PIN_NAME">: "<PIN_FUNCTION>",
        <...>
    },
    "prohibited_sites": [
        "<SITE_TYPE>",
        <...>
    ],
    "sites": {
        "<SITE_NAME>": <SITE_TYPE>,
        <...>
    },
    "type": "INT_R"
}
```

The <TILE_NAME> indicates the name of the *tile* described in the entry. The naming convention matches Vivado.

Each *tile* entry in the file has the following fields:

- "bits" - contains the data related to *tile* configuration over the <CONFIGURATION_BUS>. There are three types of the configuration buses in 7-Series FPGAs: CLB_IO_CLK, BLOCK_RAM and CFG_CLB. Every <CONFIGURATION_BUS> has the following fields:
  - baseaddr - Basic address of the configuration *frame*. Every configuration *frame* consist of 101 of 32bit *words*. Note that a single *frame* usually configures a bunch of *tiles* connected to the single configuration bus.
  - "frames" - Number of *frames* that can configure the *tile*.
  - offset - How many words of offset is present in the *frame* before the first *word* that configures the *tile*.
  - words - How many 32bit *words* configure the *tile*.
- clock_region - indicates to which *clock region* the *tile* belongs to.
- grid_x - *tile* column, increasing right
- grid_y - *tile* row, increasing down
- pin_functions - indicates the special functions of the *tile* pins. Usually it is related to IOB blocks and indicates i.e. differential output pins.
- prohibited_sites - Indicates which *site* types cannot be used in the *tile*

- `sites` - dictionary which contains information about the *sites* which can be found inside the *tile*. Every entry in the dictionary contains the following information:
  - `"<SITE_NAME>"` - The unique name of the *site* inside the *tile*.
  - `"<SITE_TYPE>` - The type of the *site*
- `type` - The type of the *tile*

### Examples

```
"CLBLL_L_X16Y149": {
    "bits": {
        "CLB_IO_CLK": {
            "baseaddr": "0x00020800",
            "frames": 36,
            "offset": 99,
            "words": 2
        }
    },
    "clock_region": "X0Y2",
    "grid_x": 43,
    "grid_y": 1,
    "pin_functions": {},
    "sites": {
        "SLICE_X24Y149": "SLICEL",
        "SLICE_X25Y149": "SLICEL"
    },
    "type": "CLBLL_L"
}
```

Interpreted as:

- *Tile* is named `CLBLL_L_X16Y149`
- *Frame* base address is `0x00020800`
- For each *frame*, skip the first 99 words loaded into FDRI
- Since it's 2 FDRI words out of possible 101, it's the last 2 words
- It spans across 36 different *frame* loads
- Located in *clock region* `X0Y2`
- Located at row 1, column 43
- Contains two *sites*, both of which are SLICEL
- Is a `CLBLL_L` type *tile*

More information about *frames* and the FPGA configuration can be found in the *Configuration* chapter. Example of absolute *frame* address calculation can be found in the *mask file* chapter.

### 5.3.2 tileconn file

The `tileconn.json` file contains the information on how the wires of neighboring tiles are connected. It contains one entry for each pair of tile types, each containing a list of pairs of wires that belong to the same node.

**Warning:** FIXME: This is a good place to add the tile wire, pip, site pin diagram.

This file documents how adjacent tile pairs are connected. No directionality is given.

#### File format

The file contains one large list:

```
[
    {
        "grid_deltas": [
            <DELTA_X>,
            <DELTA_Y>
        ],
        "tile_types": [
            "<SOURCE_TILE>",
            "<DESTINATION_TILE>"
        ],
        "wire_pairs": [
            [
                "<SOURCE_TILE_WIRE>",
                "<DESTINATION_FILE_WIRE>"
            ],
            <...>
        ],
    },
    <...>
]
```

Each entry has the following fields:

- `grid_deltas` - indicates the position (`<DELTA_X>`, `<DELTA_Y>`) of the source tile relative to the destination_file
- `tile_types` - contains the information about both `<SOURCE_TILE_TYPE>` and `<DESTINATION_TILE_TYPE>`
- `wire_pairs` - contains the names of both `<SOURCE_TILE_WIRE>` and `<DESTINATION_TILE_WIRE>`

#### Example

```
{
        "grid_deltas": [
                0,
                1
        ],
        "tile_types": [
                "CLBLL_L",
                "HCLK_CLB"
```

(continues on next page)

(continued from previous page)

```
        ],
        "wire_pairs": [
                [
                        "CLBLL_LL_CIN",
                        "HCLK_CLB_COUT0_L"
                ],
                [
                        "CLBLL_L_CIN",
                        "HCLK_CLB_COUT1_L"
                ]
        ]
}
```

Interpreted as:

- Use when a `CLBLL_L` is above a `HCLK_CLB` (i.e. pointing south from `CLBLL_L`)
- Connect `CLBLL_L.CLBLL_LL_CIN` to `HCLK_CLB.HCLK_CLB_COUT0_L`
- Connect `CLBLL_L.CLBLL_L_CIN` to `HCLK_CLB.HCLK_CLB_COUT1_L`
- A global clock tile is feeding into slice carry chain inputs

### 5.3.3 part files

Both the `part.json` and `part.yaml` files contain information about the configuration resources of the FPGA chip. The files include information about bus types and the number of *frames* that are available for the given configurational *column*.

Additionally, the file stores information about the device ID and available *IO BANKS*.

#### File format

Both files contain the same information, but since the `part.yaml` is less accessible, the description will be based on the `part.json` file.

The `part.json` file is of the following form:

```
{
    "global_clock_regions": {
        "bottom": {
            "rows": {
                "<ROW_NUMBER>" : {
                    "configuration_buses": {
                        "<CONFIGURATION_BUS>": {
                            "configurational_columns": {
                                "<COLUMN_NUMBER>": {
                                    "frame_count": <FRAME_COUNT>
                                }
                                <...>
                            }
                        }
                        <...>
                    }
                }
                <...>
```

(continues on next page)

(continued from previous page)

```
            }
        },
        "top": {
            "rows": {
                "<ROW_NUMBER>" : {
                    "configuration_buses": {
                        "<CONFIGURATION_BUS>": {
                            "configurational_columns": {
                                "<COLUMN_NUMBER>": {
                                    "frame_count": <FRAME_COUNT>
                                }
                                <...>
                            }
                        }
                        <...>
                    }
                }
                <...>
            }
        },
        }
    },
    "idcode" : <IDCODE>,
    "iobanks" : {
        "<BANK_ID>": <BANK_POSITION>",
        <...>
    }
}
```

The file contains three main entries:

- `"global_clock_regions"` - Contains the information about the configurational resources of the FPGA chip. The 7-Series FPGAs are divided into two *halves* - `top` and `bottom`. This explains the origin of those entries in the file.

  Every half contains a few `rows` associated with the global *clock regions*. The particular row of the global *clock regions* is indicated by the `<ROW_NUMBER>`. Since every row can be configured by one of three configurational buses: `CLK_IO_CLKB`, `BLOCK_RAM` or `CFG_CLB`, the appropriate bus is indicated by the `<CONFIGURATION_BUS>`.

  There are many *columns* connected to a single bus. Each column is described by appropriate `<COLUMN_NUMBER>` entry which contains the information about the number of frames (`<FRAME_COUNT>`) which can be used to configure the particular column.

- `"idcode"` - ID of the given chip package
- `"iobanks"` - a dictionary that contains the *IO Bank* ID (`<BANK_ID>`) and their position in the FPGA grid (`<BANK_POSITION>`).

### Examples

```
{
    global_clock_regions": {
        "bottom": {
            "rows": {
                "0": {
                    "configuration_buses": {
                        "BLOCK_RAM": {
                            "configuration_columns": {
                                "0": {
                                    "frame_count": 128
                                },
                                "1": {
                                    "frame_count": 128
                                },
                                "2": {
                                    "frame_count": 128
                                }
                            }
                        },
                        "CLB_IO_CLK": {
                            "configuration_columns": {
                                "0": {
                                    "frame_count": 42
                                },
                                "1": {
                                    "frame_count": 30
                                },
                                "2": {
                                    "frame_count": 36
                                },
                                <...>
                            }
                        }
                    <...>
                },
            },
        "top" : {
            <...>
        }
    },
    "idcode": 56803475,
    "iobanks": {
        "0": "X1Y78",
        "14": "X1Y26",
        "15": "X1Y78",
        "16": "X1Y130",
        "34": "X113Y26",
        "35": "X113Y78"
    }
}
```

### 5.3.4 package_pins file

The `package_pins.csv` is a simple file that describes the pins of the particular FPGA chip package.

#### File format

Every row in the file represents a single pin. Each of the pins is characterized by:

- `pin` - The package pin name
- `bank` - The ID of *IO BANK* to which the pin is connected. It should match with the data from the *part file*
- `site` - The *site* to which the pin belongs
- `tile` - The *tile* to which the pin belongs
- `pin_function` - The function of the pin

#### Example

```
A1,35,IOB_X1Y97,RIOB33_X43Y97,IO_L1N_T0_AD4N_35
```

This line means that the pin `A1` which belongs to *IO BANK* `35`, of `IOB_X1Y97` *site* in `RIOB33_X43Y97` *tile* has `IO_L1N_T0_AD4N_35` function.

# INDEX