# Project X-Ray Documentation

## *Release 0.0-2145-g85060dd*

**SymbiFlow Team**

**Apr 29, 2019**

# Xilinx 7-series Architecture

**1 Overview** **3**

**2 Configuration** **5**
- 2.1 Addressing . . . 5
- 2.2 Loading sequence . . . 6
- 2.3 Other . . . 7

**3 Bitstream format** **9**

**4 Interconnect PIPs** **11**
- 4.1 Fake PIPs . . . 11
- 4.2 Regular PIPs . . . 11
- 4.3 VCC Drivers . . . 12
- 4.4 Bidirectional PIPs . . . 12

**5 Distributed RAMs (DRAM / SLICEM)** **13**
- 5.1 Functions . . . 13
- 5.2 Configuration . . . 14

**6 Glossary** **17**

**7 References** **21**
- 7.1 Xilinx documents one should be familiar with: . . . 21
- 7.2 Other documentation that might be of use: . . . 22

**8 Project X-Ray** **23**

**9 Quickstart Guide** **25**
- 9.1 Step 1: . . . 25
- 9.2 Step 2: . . . 25
- 9.3 Step 3: . . . 25
- 9.4 Step 4: . . . 26
- 9.5 Step 5: . . . 26
- 9.6 Step 6: . . . 26
- 9.7 Step 7: . . . 26
- 9.8 Step 8: . . . 27
- 9.9 Step 9: . . . 27

**10 C++ Development 29**

**11 Process 31**
11.1 Parts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**12 Database 33**

**13 Current Focus 35**
13.1 TODO List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

**14 Contributing 37**
14.1 Sending . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
14.2 License . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
14.3 Code of Conduct . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
14.4 Sign your work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
14.5 Contributing to the docs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**15 Fuzzers 39**
15.1 Configurable Logic Blocks (CLB) . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
15.2 Block RAM (BRAM) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
15.3 Input / Output (IOB) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
15.4 Clocking (CMT, PLL, BUFG, etc) . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
15.5 Programmable Interconnect Points (PIPs) . . . . . . . . . . . . . . . . . . . . . . . 44
15.6 Hard Block Fuzzers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
15.7 Grid and Wire . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
15.8 Timing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
15.9 All Fuzzers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

**16 Minitests 57**
16.1 CLB_BUSED Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
16.2 clb-carry_cin_cyinit Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
16.3 clb-configs Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
16.4 CLB_MUXF8 Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
16.5 clkbuf Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
16.6 eccbits Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
16.7 FIXEDPNR Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
16.8 lvb_long_mux Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
16.9 nodes_wires_list Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
16.10 FASM Proof of Concept using Vivado Partial Reconfig flow . . . . . . . . . . . . . 59
16.11 Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
16.12 Using Vivado to generate .fasm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
16.13 PICORV32-v Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
16.14 PICORV32-y Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
16.15 pip-switchboxes Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
16.16 ROI_HARNESS Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
16.17 Quickstart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
16.18 How it works . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
16.19 tiles_wires_pips Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
16.20 util Minitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

**17 Tools 63**

**18 .db Files 65**
18.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
18.2 Segment bit positions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

18.3 segbits_*.db . . . . . . . . . . 66
18.4 ppips_*.db . . . . . . . . . . 67
18.5 mask_*.db . . . . . . . . . . 67
18.6 .bits example . . . . . . . . . . 67

**19 .json Files** **69**
19.1 Introduction . . . . . . . . . . 69
19.2 tilegrid.json . . . . . . . . . . 69
19.3 tileconn.json . . . . . . . . . . 71

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 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.

# CHAPTER 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.

## 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 |

### 2.1.1 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.

### 2.1.2 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.

### 2.1.3 CFG_CLB

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

## 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

## 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

## CHAPTER 3

# Bitstream format

---

**Todo:** Expand on rough notes

---

- Specific byte pattern at beginning of file to allow hardware to determine width of bus providing configuration data.
- Rest of file is 32-bit big-endian words
- All data before 32-bit synchronization word (0xAA995566) is ignored by configuration state machine
- Packetized format used to perform register reads/writes
  - Three packet header types
    - Type 0 packets exist only when performing zero-fill between rows
    - Type 1 used for writes up to 4096 words
    - Type 2 expands word count field to 27 bits by omitting register address
    - Type 2 must always be proceeded by Type 1 which sets register address
  - NOP packets are used for inserting required delays
  - Most registers only accept 1 word of data
  - Allowed register operations depends on interface used to send packets
    - Writing LOUT via JTAG is treated as a bad command
    - Single-frame FDRI writes via JTAG fail
- 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
- Xilinx BIT header
  - Additional information about how bitstream was generated
  - Unofficially documented at http://www.fpga-faq.com/FAQ_Pages/0026_Tell_me_about_bit_files.htm
  - Really does require NULL-terminated Pascal strings
  - Having this header is the distinction between .bin and .bit in Vivado
  - Is ignored entirely by devices

# CHAPTER 4

# Interconnect PIPs

## 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.

## 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 <INT>‘s 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*.

## 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.

## 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.

### CHAPTER 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.

## 5.1 Functions

### 5.1.1 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

### 5.1.2 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 |

## 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.

### 5.2.1 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 |

### 5.2.2 Ports for modes

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

| LUTs | 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] |

### 5.2.3 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 |

# CHAPTER 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.

CHAPTER 7

# References

## 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.

## 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

# CHAPTER 8

# Project X-Ray

Build Status Documentation Status License

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;

# CHAPTER 9

# Quickstart Guide

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

## 9.1 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.

## 9.2 Step 2:

Pull submodules:

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

## 9.3 Step 3:

Install CMake:

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

## 9.4 Step 4:

Build the C++ tools:

```
make build
```

## 9.5 Step 5:

(Option 1) - Install the Python environment locally

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

(Option 2) - Install the Python environment globally

```
sudo apt-get install python3-yaml
sudo 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. `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 and error about `sphinx_autodoc_typehints` but it should correctly build and install pyjson5. After this, run either option 1 or 2 again.

## 9.6 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
```

## 9.7 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)
```

## 9.8 Step 8:

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

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

## 9.9 Step 9:

Create HTML documentation:

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

## CHAPTER 10

# 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.

## CHAPTER 11

# 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.

## 11.1 Parts

### 11.1.1 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.

### 11.1.2 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.

### 11.1.3 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.

### 11.1.4 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.

### 11.1.5 Utils

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

### 11.1.6 Third Party

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

# CHAPTER 12

# 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.

# CHAPTER 13

# 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**!

## 13.1 TODO List

- [ ] Write a TODO list

# CHAPTER 14

# Contributing

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

## 14.1 Sending

All contributions should be sent as GitHub Pull requests.

## 14.2 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 `COPYING` file.

All new contributions must also be released under this license.

## 14.3 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.

## 14.4 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
```

## 14.5 Contributing to the docs

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

# CHAPTER 15

# Fuzzers

Fuzzers are things that 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 (no one is working on that yet), 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.

## 15.1 Configurable Logic Blocks (CLB)

### 15.1.1 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)

### 15.1.2 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?

### 15.1.3 clb-lutinit Fuzzer

#### NLUT.INIT

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

Sets the LUT6 INIT property

### 15.1.4 clb-n5ffmux Fuzzer

#### N5FFMUX

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

### 15.1.5 clb-ncy0 Fuzzer

#### CARRY4.NCY0

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

### 15.1.6 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.

### 15.1.7 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.

### 15.1.8 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.

### 15.1.9 clb-precyinit Fuzzer

#### PRECYINIT

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

### 15.1.10 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

## 15.2 Block RAM (BRAM)

### 15.2.1 bram-cascades Fuzzer

Missing README.md!

### 15.2.2 BRAM Configuration

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

### 15.2.3 BRAM Data

Solves for BRAM data bits

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

### 15.2.4 bram-fifo-config Fuzzer

Missing README.md!

### 15.2.5 bram36-config Fuzzer

Missing README.md!

## 15.3 Input / Output (IOB)

### 15.3.1 IOB Fuzzer

### 15.3.2 iob-ilogic Fuzzer

Missing README.md!

### 15.3.3 iob-ologic Fuzzer

Missing README.md!

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

### 15.4.1 clk-bufg-config Fuzzer

Missing README.md!

### 15.4.2 BUFG interconnect fuzzer

Solves pips located within the BUFG switch box.

### 15.4.3 clk-hrow-config Fuzzer

Missing README.md!

### 15.4.4 clk-hrow-pips Fuzzer

Missing README.md!

### 15.4.5 clk-rebuf-pips Fuzzer

Missing README.md!

### 15.4.6 HCLK_CMT interconnect fuzzer

Solves pips located within the HCLK_CMT switch box.

### 15.4.7 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).

### 15.4.8 Fuzzer for PIPs in HCLK titles

Run this fuzzer once.

It cannot solve HCLK.HCLK_CK_INOUT_* family

### 15.4.9 MMCM

`MMCME2_ADV` in UG953 lists the available attributes.

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

FIXME: Add description.

## 15.5 Programmable Interconnect Points (PIPs)

### 15.5.1 int-imux-gfan Fuzzer

Missing README.md!

### 15.5.2 int-piplist Fuzzer

Missing README.md!

### 15.5.3 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).

### 15.5.4 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).

### 15.5.5 Fuzzer for the ALT_FAN.*GFAN PIPs

This fuzzer solves the ALT_FAN.GFAN PIPs which had collisions with the GFAN PIPs.

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

Run this fuzzer once.

### 15.5.7 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).

### 15.5.8 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

### 15.5.9 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

### 15.5.10 piplist Fuzzer

Missing README.md!

### 15.5.11 ppips Fuzzer

Missing README.md!

## 15.6 Hard Block Fuzzers

### 15.6.1 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

## 15.7 Grid and Wire

### 15.7.1 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_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 (0x00000000:0xfffffff)

#### 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.

### 15.7.2 ordered_wires Fuzzer

Missing README.md!

### 15.7.3 get_counts Fuzzer

Missing README.md!

### 15.7.4 dump_all Fuzzer

Missing README.md!

## 15.8 Timing

### 15.8.1 Timing analysis fuzzer (timfuz)

WIP: 2018-09-10: this process is just starting together and is going to get significant cleanup. But heres the general idea

This runs various designs through Vivado and processes the resulting timing informationin order to create very simple timing models. While Vivado might have more involved models (say RC delays, fanout, etc), timfuz creates simple models that bound realistic min and max element delays.

Currently this document focuses exclusively on fabric timing delays.

### Quick start

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

This will take a relatively long time (say 45 min) and generate build/timgrid-v.json. You can do a quicker test run (say 3 min) using:

```
make PRJ=oneblinkw PRJN=1 -j$(nproc)
```

### Vivado background

Examples are for a XC750T on Vivado 2017.2.

TODO maybe move to: https://github.com/SymbiFlow/prjxray/wiki/Timing

### Speed index

Vivado seems to associate each delay model with a "speed index". The fabric has these in two elements: wires (ie one delay element per tile) and pips. For example, LUT output node A (ex: CLBLL_L_X12Y100/CLBLL_LL_A) has a single wire, also called CLBLL_L_X12Y100/CLBLL_LL_A. This has speed index 733. Speed models can be queried and we find this corresponds to model C_CLBLL_LL_A.

There are various speed model types:

- bel_delay
- buffer
- buffer_switch
- content_version
- functional
- inpin
- outpin
- parameters
- pass_transistor
- switch
- table_lookup
- tl_buffer
- vt_limits
- wire

IIRC the interconnect is only composed of switch and wire types.

Indices with value 65535 (0xFFFF) never appear. Presumably these are unused models. They are used for some special models such as those of type "content_version". For example, the "xilinx" model is of type "content_version".

There are also "cost codes", but these seem to be very course (only around 30 of these) and are suspected to be related more to PnR than timing model.

### Timing paths

The Vivado timing analyzer can easily output the following:

- Full: delay from BEL pin to BEL pin
- Interconnect only (ICO): delay from BEL pin to BEL pin, but only report interconnect delays (ie exclude site delays)

There is also theoretically an option to report delays up to a specific pip, but this option is poorly documented and I was unable to get it to work.

Each timing path reports a fast process and a slow process min and max value. So four process values are reported in total:

- fast_max
- fast_min
- slow_max
- slow_min

For example, if the device is end of life, was poorly made, and at an extreme temperature, the delay may be up to the slow_max value. Since ICO can be reported for each of these, fully analyzing a timing path results in 8 values.

Finally, part of this was analyzing tile regularity to discover what a reasonably compact timing model was. We verified that all tiles of the same type have exactly the same delay elements.

### Methodology

Make sure you've read the Vivado background section first

### Background

This section briefly describes some of the mathmatics used by this technique that readers may not be familiar with. These definitions are intended to be good enough to provide a high level understanding and may not be precise.

Numerical analysis: the study of algorithms that use numerical approximation (as opposed to general symbolic manipulations)

numpy: a popular numerical analysis python library. Often written np (import numpy as np).

scipy: provides higher level functionality on top of numpy

sympy ("symbolic python"): like numpy, but is designed to work with rational numbers. For example, python actually stores 0.1 as 0.1000000000000000055511151231257827021181583404541015625. However, sympy can represent this as the fraction 1/10, eliminating numerical approximation issues.

Least squares (ex: scipy.optimize.least_squares): approximation method to do a best fit of several variables to a set of equations. For example, given the equations "x = 1" and "x = 2" there isn't an exact solution. However, "x = 1.5" is a good compromise since its reasonably solves both equations.

Linear programming (ex: scipy.optimize.linprog aka linprog): approximation method that finds a set of variables that satisfy a set of inequalities. For example,

Reduced row echelon form (RREF, ex: sympy.Matrix.rref): the simplest form that a system of linear equations can be solved to. For example, given "x = 1" and "x + y = 9", one can solve for "x = 1" and "y = 8". However, given "x + y = 1" and "x + y + z = 9", there aren't enough variables to solve this fully. In this case RREF provides a best effort by giving the ratios between correlated variables. One variable is normalized to 1 in each of these ratios and is called the "pivot". Note that if numpy.linalg.solve encounters an unsolvable matrix it may either complain or generate a false solution due to numerical approximation issues.

### What didn't work

First some quick background on things that didn't work to illustrate why the current approach was chosen. I first tried to directly through things into linprog, but it unfairly weighted towards arbitrary shared variables. For example, feeding in:

- t0 >= 10
- t0 + t1 >= 100

It would declare "t0 = 100", "t1 = 0" instead of the more intuitive "t0 = 10", "t1 = 90". I tried to work around this in several ways, notably subtracting equations from each other to produce additional constraints. This worked okay, but was relatively slow and wasn't approaching nearly solved solutions, even when throwing a lot of data at it.

Next we tried randomly combining a bunch of the equations together and solving them like a regular linear algebra matrix (numpy.linalg.solve). However, this illustrated that the system was under-constrained. Further analysis revealed that there are some delay element combinations that simply can't be linearly separated. This was checked primarily using numpy.linalg.matrix_rank, with some use of numpy.linalg.slogdet. matrix_rank was preferred over slogdet since its more flexible against non-square matrices.

### Process

Above ultimately led to the idea that we should come up with a set of substitutions that would make the system solvable. This has several advantages:

- Easy to evaluate which variables aren't covered well enough by source data
- Easy to evaluate which variables weren't solved properly (if its fully constrained it should have had a non-zero delay)

At a high level, the above learnings gave this process:

- Find correlated variables by using RREF (sympy.Matrix.rref) to create variable groups
  - Note pivots
  - You must input a fractional type (ex: fractions.Fraction, but surprisingly not int) to get exact results, otherwise it seems to fall back to numerical approximation
  - This is by far the most computationally expensive step
  - Mixing RREF substitutions from one data set to another may not be recommended
- Use RREF result to substitute groups on input data, creating new meta variables, but ultimately reducing the number of columns
- Pick a corner
  - Examples assume fast_max, but other corners are applicable with appropriate column and sign changes
- De-duplicate by removing equations that are less constrained
  - Ex: if solving for a max corner and given:
  - t0 + t1 >= 10
  - t0 + t1 >= 12
  - The first equation is redundant since the second provides a stricter constraint

  - This significantly reduces computational time
- Use least squares (scipy.optimize.least_squares) to fit variables near input constraints
  - Helps fairly weight delays vs the original input constraints
  - Does not guarantee all constraints are met. For example, if this was put in (ignoring these would have been de-duplicated):
  - t0 = 10
  - t0 = 12
  - It may decide something like t0 = 11, which means that the second constraint was not satisfied given we actually want t0 >= 12
- Use linear programming (scipy.optimize.linprog aka linprog) to formally meet all remaining constraints
  - Start by filtering out all constraints that are already met. This should eliminate nearly all equations
- Map resulting constraints onto different tile types
  - Group delays map onto the group pivot variable, typically setting other elements to 0 (if the processed set is not the one used to create the pivots they may be non-zero)

### TODO, suggestions

Includes

- Consider removing rref
  - Intended to understand what can't be solved, maybe not useful in production
- Need more coverage
  - Consider instrumenting all fuzzers to output data to feed into timing anlayzer
  - Justification: we need a lot of weird cases, we have code that does that in the other fuzzers
- Tune performance parameters
  - Can we improve quality of results?
  - Do we have a good enough quality checker? (solve_qor.py)
  - Compare our vs Xilinx output on random designs
  - Does the solve take too long? What could speed it up?
- Investigate min corner
  - Tends to solve towards 0, making this not useful
  - Low priority: most designs just close timing with setup time
- Investigate characterizing full RC timing model
- Can we split pivot delays among elements instead of entirely into pivot?
- Consider breaking out timing analyzer into its own project / library so it can be re-used on other projects
- Review "–massage". Does this help?
- Review computed site delays vs published Xilinx numbers (DC and AC Switching Characteristics)
- Fabric delay models are RC, but are the site delay models RC as well or maybe just linear?
- Can we create antenna nets to get simpler solves?

- Can we get tcl timing analyzer to analyze a partial route?
    - Option says you should be able to do this
    - I could not actually get it to work

### Improve test cases

Test cases are somewhat random right now. We could make much more targetted cases using custom routing to improve various fanout estimates and such. Also there are a lot more elements that are not covered. At a minimum these should be moved to their own directory.

### ZERO models

Background: there are a number of speed models with the name ZERO in them. These generally seem to be zero delay, although needs more investigation.

Example: see pseudo pip item below

The timing models will probably significantly improve if these are removed. In the past I was removing them, but decided to keep them in for now in the spirit of being more conservative.

They include:

- _BSW_CLK_ZERO
- BSW_CLK_ZERO
- _BSW_ZERO
- BSW_ZERO
- _B_ZERO
- B_ZERO
- C_CLK_ZERO
- C_DSP_ZERO
- C_ZERO
- I_ZERO
- _O_ZERO
- O_ZERO
- RC_ZERO
- _R_ZERO
- R_ZERO

### Virtual switchboxes

Background: several low level configuration details are abstracted with virtual configurable elements. For example, LUT inputs can be rearranged to reduce routing congestion. However, the LUT configuratioon must be changed to match the switched inputs. This is handled by the CLBLL_L_INTER switchbox, which doesn't encode any physical configuration bits. However, this contains PIPs with delay models.

For example, LUT A, input A1 has node CLBLM_M_A1 coming from pip junction CLBLM_M_A1 has PIP CLBLM_IMUX7->CLBLM_M_A1 with speed index 659 (R_ZERO).

This might be further evidence on related issue that ZERO models should probably be removed.

#### Incporporate fanout

We could probably significantly improve model granularity by studying delay impact on fanout

#### Investigate RC delays

Suspect accuracy could be significantly improved by moving to SPICE based models. But this will take significantly more characterization

#### Characterize real hardware

A few people have expressed interest on running tests on real hardware. Will take some thought given we don't have direct access

#### Review approximation errors

Ex: one known issue is that the objective function linearly weights small and large delays. This is only recommended when variables are approximately the same order of magnitude. For example, carry chain delays are on the order of 7 ps while other delays are 100 ps. Its very easy to put a large delay on the carry chain while it could have been more appropriately put somewhere else.

## 15.9 All Fuzzers

### 15.9.1 dsp-mskpat Fuzzer

Missing README.md!

### 15.9.2 fifo-config Fuzzer

Missing README.md!

### 15.9.3 init-db Fuzzer

Missing README.md!

### 15.9.4 part-yaml Fuzzer

Missing README.md!

### 15.9.5 pins Fuzzer

Missing README.md!

## CHAPTER 16

# Minitests

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

## 16.1 CLB_BUSED Minitest

### 16.1.1 Purpose

Tests for BUSED bit

### 16.1.2 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

## 16.2 clb-carry_cin_cyinit Minitest

Missing README.md!

## 16.3 clb-configs Minitest

Missing README.md!

## 16.4 CLB_MUXF8 Minitest

### 16.4.1 Purpose

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

### 16.4.2 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

## 16.5 clkbuf Minitest

Missing README.md!

## 16.6 eccbits Minitest

Missing README.md!

## 16.7 FIXEDPNR Minitest

### 16.7.1 Purpose

### 16.7.2 Result

Preliminary result

## 16.8 lvb_long_mux Minitest

Missing README.md!

## 16.9 nodes_wires_list Minitest

Missing README.md!

## 16.10 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.

## 16.11 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
```

## 16.12 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.

## 16.13 PICORV32-v Minitest

### 16.13.1 Purpose

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

### 16.13.2 Result

## 16.14 PICORV32-y Minitest

### 16.14.1 Purpose

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

### 16.14.2 Result

## 16.15 pip-switchboxes Minitest

Missing README.md!

## 16.16 ROI_HARNESS Minitest

### 16.16.1 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;
- Zynq boards;

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
```

## 16.17 Quickstart

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

## 16.18 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

### 16.18.1 Result

## 16.19 tiles_wires_pips Minitest

Missing README.md!

## 16.20 util Minitest

Missing README.md!

## CHAPTER 17

# Tools

*SymbiFlow/prjxray/tools/*

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

# CHAPTER 18

# .db Files

## 18.1 Introduction

This section documents how prjxray represents the bitstream database. The databases are plain text files, using either simple line-based syntax or JSON. The databases are located in *database/<device_class>/*. The *settings.sh* file contains some configurations used by the tools that generate the database, including the region of interest (ROI, see [[Glossary]]).

**These ".db" files come in two common flavors:**

- *segbits_*.db*: encodes bitstream bits
- *mask_*.db*: which bits are used by a segment? Probably needs to be converted to tile

Also note: .rdb (raw db) is a convention for a non-expanded .db file (see below)

## 18.2 Segment bit positions

Bit positions within a segment are written using the following notation: A two digit decimal number followed by an underscore followed by a two digit decimal number. For example *26_47*.

The first number indicates the frame address relative to the base frame address for the segment and ranges from *00* to *35* for Atrix-7 CLB segments.

The second number indicates the bit position width.

> **Warning:** FIXME: Expand this section. We've had a couple questions around this, probably good to get a complete description of this that we can point people too. This is probably a good place to talk about tile grid and how it applies to segbit.

## 18.3 segbits_*.db

Tag files document the meaning of individual configuration bits or bit pattern. They contain one line for each pattern. The first word (sequence of non-whitespace characters) in that line is the *configuration tag*, the remaining words in the line is the list of bits for that pattern. A bit prefixed with a *!* marks a bit that must be cleared, a bit not prefixed with a *!* marks a bit that must be set.

No configuration tag may include the bit pattern for another tag as subset. If it does then this is an indicator that there is an incorrect entry in the database. Usually this either means that a tag has additional bits in their pattern that should not be there, or that *!<bit>* entries are missing for one or more tags.

These are created by segmatch to describe bitstream IP encoding.

**Example lines:**

- CLB.SLICE_X0.DFF.ZINI 31_58 * For feature CLB.SLICE_X0.DFF.ZINI * Frame: 31 * Word: 58 // 32 = 1 * Mask: 1 << (58 % 32) = 0x04000000 * To set an actual bitstream location, you will need to adjust frame and word by their tile base addresses
- CLBLL_L.SLICEL_X0.AOUTMUX.A5Q !30_06 !30_08 !30_11 30_07 * A multi bit entry. Bit 30_06 should be cleared to use this feature
- INT_L.BYP_BOUNCE5.BYP_ALT5 always * A pseudo pip: feaure always active => no bits required
- CLBLL_L.OH_NO.BAD.SOLVE <const0> * Internal only * Candidate bits exist, but they've only ever been set to 0
- CLBLL_L.OH_NO.BAD.SOLVE <const1> * Internal only * Candidate bits exist, but they've only ever been set to 1
- INT.FAN_ALT4.SS2END0 <m1 2> 18_09 25_08 * Internal only * segmatch -m (min tag value occurrences) was given, but occurrences are below this threshold * ie INT.FAN_ALT4.SS2END0 occcured twice, but this is below the acceptable level (say 5)
- INT.FAN_ALT4.SS2END0 <M 6 8> 18_09 25_08 * Internal only * segmatch -M (min tag occurrences) was given, but total occurrences are below this threshold * First value (6) is present=1, second value (8) is present=0 * Say -M 15, but there are 6 + 8 = 14 samples, below the acceptable threshold

**Related tools:**

- segmatch: solves symbolic constraints on a bitstream to produce symbol bitmasks
- dbfixup.py: internal tool that expands multi-bit encodings (ex: one hot) into groups. For example: * .rdb file with one hot: BRAM.RAMB18_Y1.WRITE_WIDTH_A_18 27_267 * .db: file expanded: BRAM.RAMB18_Y1.WRITE_WIDTH_A_18 !27_268 !27_269 27_267
- parsedb.py: valides that a database is fully and consistently solved * Optionally outputs to canonical form * Ex: complains if const0 entries exist * Ex: complains if symbols are duplicated (such as after a mergedb after rename)
- mergedb.sh: adds new bit entries to an existing db * Ex: CLB is solved by first solving LUT bits, and then solving FF bits

### 18.3.1 Interconnect PIP Tags

Tags for interconnect *PIPs* are stored in the *segbits_int_l.db* and *segbits_int_r.db* database files.

Tags that enable interconnect *PIPs* have the following syntax: *<tile_type>.<destination_wire>.<source_wire>*.

The *<tile_type>* may be *INT_L* or *INT_R*. The destination and source wires are wire names in that tile type. For example, consider the following entry in *segbits_int_l.db*: *INT_L.NL1BEG1.NN6END2 07_32 12_33*

**Warning:** FIXME: This is probably a good place to reference tileconn as the documentation that explains how wires are connected outside of switchboxes (which is what pips document).

This means that the bits *07_32* and *12_33* must be set in the segment to drive the value from the wire *NN6END2* to the wire *NL1BEG1*.

### 18.3.2 CLB Configurations Tags

Tags for CLB tiles use a dot-separated hierarchy for their tag names. For example the tag *CLBLL_L.SLICEL_X0.ALUT.INIT[00]* documents the bit position of the LSB LUT init bit for the ALUT for the slice with even X coordinate within a *CLBLL_L* tile. (There are 4 LUTs in a slice: ALUT, BLUT, CLUT, and DLUT. And there are two slices in a CLB tile: One with an even X coordinate using the *SLICEL_X0* namespace for tags, and one with an odd X coordinate using the *SLICEL_X1* namespace for tags.)

## 18.4 ppips_*.db

Pseudo *PIPs* are *PIPs* in the Vivado tool, but do not have actual bit pattern. The *ppips_*.db* files contain information on pseudo-*PIPs*. Those files contain one entry per pseudo-PIP, each with one of the following three tags: *always*, *default* or *hint*. 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). And 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.

## 18.5 mask_*.db

These are just simple bit lists

Example line: bit 01_256

See previous section for number meaning

For each segment type there is a mask file *mask_<seg_type>.db* that contains one line for each bit that has been observed being set in any of the example designs generated during generation of the database. The lines simply contain the keyword *bit* followed by the bit position. This database is used to identify unused bits in the configuration segments.

## 18.6 .bits example

Say entry is: bit_0002050b_002_05

2 step process: * Decode which segment * Decode which bit within that segment

We have: * Frame address 0x0002050b (hex) * Word #: 2 (decimal, 0-99) * Bit #: 5 (decimal, 0-31)

The CLB tile and the associated interconnect switchbox tile are configured together as a segment. However, configuration data is grouped by segment column rather than tile column. First, note this segment consists of 36 frames. Second, note there are 100 32 bit words per frame (+ 1 for checksum => 101 actual). Each segment takes 2 of those words meaning 50 segments (ie 50 CLB tiles + 50 interconnect tiles) are effected per frame. This means that the smallest unit that can be fully configured is a group of 50 CLB tile + switchbox tile segments taking 4 * 36 * 101 = 14544 bytes. Finally, note segment columns are aligned to 0x80 addresses (which easily fits the 36 required frames).

tilegrid.json defines addresses more precisely. Taking 0x0002050b, the frame base address is 0x0002050b & 0xFFFFFF80 => 0x00020500. The frame offset is 0x0002050b & 0x7F => 0x0B => 11.

So in summary: * Frame base address: 0x00020500 * Frame offset: 0x0B (11) * Frame word #: 2 * Frame word bit #: 5

So, with this in mind, we have frame base address 0x00020500 and word # 2. This maps to tilegrid.json entry SEG_CLBLL_L_X12Y101 (has “baseaddr”: [“0x00020600”, 2]). This also yields “type”: “clbll_l” meaning we are configuring a CLBLL_L.

**Warning:** FIXME: This example is out of date with the new tilegrid format, should update it.

Looking at segbits_clbll_l.db, we need to look up the bit at segment column 11, offset at bit 5. However, this is not present, so we fall back to segbits_int_l.db. This yields a few entries related to EL1BEG (ex: INT_L.EL1BEG_N3.EL1END0 11_05 13_05).

## CHAPTER 19

---

# .json Files

---

## 19.1 Introduction

**This section documents how prjxray represents FPGA fabric. Its primarily composed of two files:**

- tilegrid.json: list of tiles and how they appear in the bitstream
- tileconn.json: how tiles are connected together

**General notes:**

- prjxray created names are generally lowercase, while Vivado created names are generally uppercase
- _l and _r entries are generally identical, but probably represent different physical IP block layouts

## 19.2 tilegrid.json

The file *tilegrid.json* contains lists of all tiles in the device and the configuration segments formed by those tiles. It also documents the membership relationship of tiles and segments.

For each segment this contains the configuration frame base address, and the word offset within the frames, as well as the number of frames for the segment and number of occupied words in each frame.

**Warning:** FIXME: We should cross link to how to use the base address and word offset.

For each tile this file contains the tile type, grid X/Y coordinates for the tile, and sites (slices) within the tile.

This section assumes you are already familiar with the 7 series bitstream format.

**This file contains two elements:**

- segments: each entry lists sections of the bitstream that encode part of one or more tiles
- tiles: cores

### 19.2.1 segments

Segments are a prjxray concept.

**Each entry has the following fields:**

- baseaddr: a tuple of (base address, inter-frame offset)
- frames: how many frames are required to make a complete segment
- words: number of inter-frame words required for a complete segment
- tiles: which tiles reference this segment
- type: prjxray given segment type

Sample entry:

```
"SEG_CLBLL_L_X16Y149": {
        "baseaddr": [
                "0x00020800",
                99
        ],
        "frames": 36,
        "tiles": [
                "CLBLL_L_X16Y149",
                "INT_L_X16Y149"
        ],
        "type": "clbll_l",
        "words": 2
}
```

**Interpreted as:**

- Segment is named SEG_CLBLL_L_X16Y149
- Frame base address is 0x00020800
- For each frame, skip the first 99 words loaded into FDRI
- Since its 2 FDRI words out of possible 101, its the last 2 words
- It spans across 36 different frame loads
- The data in this segment is used by two different tiles: CLBLL_L_X16Y149, INT_L_X16Y149

Historical note: In the original encoding, a segment was a collection of tiles that were encoded together. For example, a CLB is encoded along with a nearby switch. However, some tiles, such as BRAM, are much more complex. For example, the configuration and data are stored in seperate parts of the bitstream. The BRAM itself also spans multiple tiles and has multiple switchboxes.

### 19.2.2 tiles

**Each entry has the following fields:**

- grid_x: tile column, increasing right
- grid_y: tile row, increasing down
- segment: the primary segment providing bitstream configuration
- sites: dictionary of sites name: site type contained within tile
- type: Vivado given tile type

Sample entry:

```
"CLBLL_L_X16Y149": {
        "grid_x": 43,
        "grid_y": 1,
        "segment": "SEG_CLBLL_L_X16Y149",
        "sites": {
                "SLICE_X24Y149": "SLICEL",
                "SLICE_X25Y149": "SLICEL"
        },
        "type": "CLBLL_L"
}
```

**Interpreted as:**

- Located at row 1, column 43
- Is configured by segment SEG_CLBLL_L_X16Y149
- Contains two sites, both of which are SLICEL
- A CLBLL_L type tile

## 19.3 tileconn.json

The file *tileconn.json* contains the information how the wires of neighboring tiles are connected to each other. 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.

**The file contains one large list. Each entry has the following fields:**

- grid_deltas: (x, y) delta going from source to destination tile
- tile_types: (source, destination) tile types
- wire_pairs: list of (source tile, destination tile) wire names

Sample entry:

```
{
        "grid_deltas": [
                0,
                1
        ],
        "tile_types": [
                "CLBLL_L",
                "HCLK_CLB"
        ],
        "wire_pairs": [
                [
                        "CLBLL_LL_CIN",
                        "HCLK_CLB_COUT0_L"
                ],
                [
```

(continues on next page)

(continued from previous page)

```
                            "CLBLL_L_CIN",
                            "HCLK_CLB_COUT1_L"
                    ]
            ]
}
```

**Interpreted as:**

- Use when a CLBLL_L is above a HCLK_CLB (ie 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

# Index

A

ASIC, 17

B

basic element, 17
basic logic element, 17
BEL, 17
Bitstream, 17
BLE, 17
Block RAM, 17

C

CFA, 17
CLB, 18
Clock, 17
Clock backbone, 17
Clock domain, 18
Clock region, 18
Clock spine, 17
Column, 18
Configurable logic block, 18

D

Database, 18

F

Fabric sub region, 18
FF, 18
Flip flop, 18
FPGA, 18
Frame, 18
Frame base address, 18
FSR, 18
Fuzzer, 18

H

Half, 18
HDL, 18
Horizontal clock row, 18
HROW, 18

I

I/O block, 18
INT, 18
Interconnect tile, 18

L

LUT, 19

M

MUX, 19

N

Node, 19

P

PIP, 19
Place and route, 19
PnR, 19
Programmable interconnect point, 19

R

Region of interest, 19
ROI, 19
Routing fabric, 19

S

Segment, 19
Site, 19
Slice, 19
Specimen, 19

T

Tile, 19

W

Wire, 19
Word, 19