Project X-Ray Documentation

Release 0.0-1037-gb0b9095

SymbiFlow Team

Nov 13, 2018

Xilinx 7-series Architecture

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

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.

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 5

Overview

5.1 SymbiFlow/symbiflow-arch-defs

This is where we describe the logical components in a device to VPR.

- VtR stands for Verilog to Routing,
- VPR stands for VtR Place and Route.
- VtR also has its own synthesis tool called ODIN-II, but we are using Yosys instead of that.

5.2 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.

5.2.1 Tilegrid Fuzzer

This fuzzer creates the tilegrid.json bitstream databaseThis 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 addressesA reference bitstream is generated and then a series of bitstreams are generated each with one LUT bit toggledThese are compared to find a toggled bit in the CLB segment columnThe resulting address is truncated to get the base frame addressFinally, 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 ROISee database for example values

XRAY_ROI_FRAMES

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

XRAY_ROI_GRID_*

Optionally these as a small performance optimization:

- XRAY_ROI_GRID_X1
- XRAY_ROI_GRID_X2
- XRAY_ROI_GRID_Y1
- XRAY_ROI_GRID_Y2

Which should, if unused, be set to -1, with this caveat:WARNING: CLB test generates this based on CLBs but implicitly includes INTTherefore, if you don't set an explicit XRAY_ROI_GRID_* it may failif you don't have a CLB*_L at left and a CLB*_R at right

5.2.2 FFConfig Fuzzer

Documents the following:

- FF clock inversion
- FF primitive mapping
- FF initialization value

Clock inversion is per slice (as BEL CLKINV)Vivado GUI is misleading as it often shows it per FF, which is not actually true

| |FFSYNC|LATCH|ZRST | |——|——|——|——| |Sample| 00_48|30_32|30_12| |FDPE | | | | |FDSE | X | | | |FDRE | X | | X | |FDCE | | | X | |LDCE | | X | X | |LDPE | | X | |

```
CLB.SLICE_X0.A5FF.ZINIT 31_06
CLB.SLICE_X0.A5FF.ZRESET 01_07
CLB.SLICE_X0.AFF.ZINIT 31_03
CLB.SLICE_X0.AFF.ZRESET 30_12
CLB.SLICE_X0.B5FF.ZINIT 31_22
CLB.SLICE_X0.B5FF.ZRESET 01_19
CLB.SLICE_X0.BFF.ZINIT 31_28
CLB.SLICE_X0.BFF.ZRESET 30_30
CLB.SLICE_X0.C5FF.ZINIT 31_41
CLB.SLICE_X0.C5FF.ZRESET 01_47
CLB.SLICE_X0.CFF.ZINIT 31_33
CLB.SLICE_X0.CFF.ZRESET 30_33
CLB.SLICE_X0.CLKINV 01_51
CLB.SLICE_X0.D5FF.ZINIT 31_51
CLB.SLICE_X0.D5FF.ZRESET 01_55
CLB.SLICE_X0.DFF.ZINIT 31_58
CLB.SLICE_X0.DFF.ZRESET 30_50
CLB.SLICE_X0.FFSYNC 00_48
CLB.SLICE_X0.LATCH 30_32
CLB.SLICE_X1.A5FF.ZINIT 31_05
CLB.SLICE_X1.A5FF.ZRESET 01_03
CLB.SLICE_X1.AFF.ZINIT 31_04
CLB.SLICE_X1.AFF.ZRESET 31_15
CLB.SLICE_X1.B5FF.ZINIT 31_23
CLB.SLICE_X1.B5FF.ZRESET 00_16
CLB.SLICE_X1.BFF.ZINIT 31_29
CLB.SLICE_X1.BFF.ZRESET 31_30
CLB.SLICE_X1.C5FF.ZINIT 31_42
CLB.SLICE_X1.C5FF.ZRESET 00_44
CLB.SLICE_X1.CFF.ZINIT 31_34
CLB.SLICE_X1.CFF.ZRESET 30_34
CLB.SLICE_X1.CLKINV 00_52
CLB.SLICE_X1.D5FF.ZINIT 31_52
CLB.SLICE_X1.D5FF.ZRESET 00_56
CLB.SLICE_X1.DFF.ZINIT 31_59
CLB.SLICE_X1.DFF.ZRESET 31_50
CLB.SLICE_X1.FFSYNC 01_31
CLB.SLICE_X1.LATCH 31_32
```

5.2.3 CLBn5FFMUX Fuzzer

Purpose

Document A5FFMUX family of CLB muxes

Algorithm

5FFMUXInputs can come from either the LUT6_2 NO5 output or the CLB NX inputTo perturb the CLB the smallest, want LUT6 always instantiatedHowever, some routing congestion that would require putting FFs in bypass(which turns out is actually okay, but didn't realize that at the time)Decided instead ot instantiate LUT8, but not use the outputTurns out this is okay and won't optimize things awaySo then, the 5FF D input is switched between the O5 output and an external CLB input

Outcome

Bits are one hot encoded per mux position

5.2.4 CLBnCY0 Fuzzer

Purpose

Document ACY0 family of CLB muxes

Algorithm

Outcome

5.2.5 FFSRCEMUX Fuzzer

Purpose

Document CEUSEDMUX, SRUSEDMUX muxes

Algorithm

Results

CEUSEDMUX: whether clock enable (CE) is used or clock always on

0: always on1: controlledCLB.SLICE_X0.CEUSEDMUX 00_39CLB.SLICE_X1.CEUSEDMUX <0 candidates>

SRUSEDMUX: whether FF can be reset or simply uses D value

(How used when SR?)0: never reset1: controlledCLB.SLICE_X0.SRUSEDMUX 00_35CLB.SLICE_X1.SRUSEDMUX <0 candidates>

5.2.6 CLBnFFMUX Fuzzer

Purpose

Document nFFMUX family of CLB muxes

Algorithm

Outcome

```
CLB.SLICE_X0.AFFMUX.B0 30_00
CLB.SLICE_X0.AFFMUX.B1 30_01
CLB.SLICE_X0.AFFMUX.B2 30_02
CLB.SLICE_X0.AFFMUX.B3 30_03
CLB.SLICE_X0.BFFMUX.B0 30_27
CLB.SLICE_X0.BFFMUX.B1 30_26
CLB.SLICE_X0.BFFMUX.B2 30_25
CLB.SLICE_X0.BFFMUX.B3 30_24
CLB.SLICE_X0.CFFMUX.B0 30_35
CLB.SLICE_X0.CFFMUX.B1 30_36
CLB.SLICE_X0.CFFMUX.B2 30_37
CLB.SLICE_X0.CFFMUX.B3 30_38
CLB.SLICE_X0.DFFMUX.B0 30_62
CLB.SLICE_X0.DFFMUX.B1 30_61
CLB.SLICE_X0.DFFMUX.B2 30_60
CLB.SLICE_X0.DFFMUX.B3 30_59
CLB.SLICE_X1.AFFMUX.B0 31_00
CLB.SLICE_X1.AFFMUX.B1 31_01
CLB.SLICE_X1.AFFMUX.B2 31_02
CLB.SLICE_X1.AFFMUX.B3 30_04
CLB.SLICE_X1.BFFMUX.B0 31_25
CLB.SLICE_X1.BFFMUX.B1 31_27
CLB.SLICE_X1.BFFMUX.B2 31_26
CLB.SLICE_X1.BFFMUX.B3 31_24
CLB.SLICE_X1.CFFMUX.B0 31_35
CLB.SLICE_X1.CFFMUX.B1 31_38
CLB.SLICE_X1.CFFMUX.B2 31_37
CLB.SLICE_X1.CFFMUX.B3 31_36
CLB.SLICE_X1.DFFMUX.B0 30_58
CLB.SLICE_X1.DFFMUX.B1 31_61
CLB.SLICE_X1.DFFMUX.B2 31_62
CLB.SLICE_X1.DFFMUX.B3 31_60
```

5.2.7 CLBnOUTMUX Fuzzer

Purpose

Document nOUTMUX family of CLB muxes

Algorithm

Outcome

```
CLB.SLICE_X0.AOUTMUX.B0 30_11
CLB.SLICE_X0.AOUTMUX.B1 30_08
CLB.SLICE_X0.AOUTMUX.B2 30_06
CLB.SLICE_X0.AOUTMUX.B3 30_07
CLB.SLICE_X0.BOUTMUX.B0 30_20
CLB.SLICE_X0.BOUTMUX.B1 30_21
CLB.SLICE_X0.BOUTMUX.B2 30_22
CLB.SLICE_X0.BOUTMUX.B3 30_23
CLB.SLICE_X0.COUTMUX.B0 30_45
CLB.SLICE_X0.COUTMUX.B1 30_44
CLB.SLICE_X0.COUTMUX.B2 30_40
```

(continues on next page)

(continued from previous page)

```
CLB.SLICE_X0.COUTMUX.B3 30_43
CLB.SLICE_X0.DOUTMUX.B0 30_56
CLB.SLICE_X0.DOUTMUX.B1 30_51
CLB.SLICE_X0.DOUTMUX.B2 30_52
CLB.SLICE_X0.DOUTMUX.B3 30_57
CLB.SLICE_X1.AOUTMUX.B0 31_09
CLB.SLICE_X1.AOUTMUX.B1 31_07
CLB.SLICE_X1.AOUTMUX.B2 31_10
CLB.SLICE_X1.AOUTMUX.B3 30_05
CLB.SLICE_X1.BOUTMUX.B0 31_20
CLB.SLICE_X1.BOUTMUX.B1 30_28
CLB.SLICE_X1.BOUTMUX.B2 31_21
CLB.SLICE_X1.BOUTMUX.B3 30_29
CLB.SLICE_X1.COUTMUX.B0 31_43
CLB.SLICE_X1.COUTMUX.B1 30_42
CLB.SLICE_X1.COUTMUX.B2 31_40
CLB.SLICE_X1.COUTMUX.B3 30_41
CLB.SLICE_X1.DOUTMUX.B0 31_56
CLB.SLICE_X1.DOUTMUX.B1 30_53
CLB.SLICE_X1.DOUTMUX.B2 31_57
CLB.SLICE_X1.DOUTMUX.B3 31_53
```

From manual O6 testing

```
30_11   X0 AOUTMUX O6
30_20   X0 BOUTMUX O6
30_45   X0 COUTMUX O6
30_56   X0 DOUTMUX O6
31_09   X1 AOUTMUX O6
31_20   X1 BOUTMUX O6
31_43   X1 COUTMUX O6
31_56   X1 DOUTMUX O6
```

5.2.8 CLBPRECYINIT Fuzzer

Purpose

Document PRECYINIT mux

Algorithm

Outcome

```
CLB.SLICE_X0.PRECYINIT.0 <0 candidates>
CLB.SLICE_X0.PRECYINIT.1 00_12
CLB.SLICE_X0.PRECYINIT.AX 30_14
CLB.SLICE_X0.PRECYINIT.CIN 30_13
CLB.SLICE_X1.PRECYINIT.0 <0 candidates>
CLB.SLICE_X1.PRECYINIT.1 01_11
CLB.SLICE_X1.PRECYINIT.AX 31_13
CLB.SLICE_X1.PRECYINIT.CIN 31_12
```

5.2.9 CLBRAM Fuzzer

Purpose

Solves SLICEM specific bits:

- Shift register LUT (SRL)
- Memory size
- RAM vs LUT
- Related muxes

Algorithm

Outcome

```
CLB.SLICE_X0.ALUT.RAM 31_16
CLB.SLICE_X0.ALUT.SMALL 00_04
CLB.SLICE_X0.ALUT.SRL 30_16
CLB.SLICE_X0.BLUT.RAM 31_17
CLB.SLICE_X0.BLUT.SMALL 00_24
CLB.SLICE_X0.BLUT.SRL 30_17
CLB.SLICE_X0.CLUT.RAM 31_46
CLB.SLICE_X0.CLUT.SMALL 00_28
CLB.SLICE_X0.CLUT.SRL 30_46
CLB.SLICE_X0.DLUT.RAM 31_47
CLB.SLICE_X0.DLUT.SMALL 01_59
CLB.SLICE_X0.DLUT.SRL 30_47
CLB.SLICE_X0.WA7USED 00_40
CLB.SLICE_X0.WA8USED 01_27
CLB.SLICE_X0.WEMUX.CE 01_23
```

5.2.10 NDI1MUX Fuzzer

See minitest for DI notes

5.2.11 Generic fuzzer for INT PIPs

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

5.2.12 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).

5.2.13 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).

5.2.14 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).

5.2.15 Fuzzer for INT PIPs driving the GFAN wires

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

5.2.16 Fuzzer for INT PIPs driving the GFAN wires with GND

Run this fuzzer once.

5.2.17 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.

5.2.18 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).

5.2.19 Fuzzer for PIPs in HCLK titles

Run this fuzzer once.

5.3 Minitests

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

5.3.1 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

5.3.2 CLB_FFCFG Minitest

Purpose

Tests all of the documented FF "primitives" (there are only 4 real ones)

Result

5.3.3 CLB_MUXF8 Minitest

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)

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

5.3.4 CLB_n5FFMUX Minitest

Purpose

Result

|DFFMUX|30_09|30_54| |——|——|——| | A | 1| 0| | B | 0| 1|

|CFFMUX|30_39|31_45| |——|——|——| |A | 0| 1| |B | 1| 0|

|BFFMUX|30_18|30_19| |——|——|——| |A | 0| 1| |B | 1| 0|

|AFFMUX|30_09|30_55| |——|——|——| |A | 1| 0| |B | 0| 1|

5.3.5 CLB_nCY0 Minitest

Purpose

Result

|DCY0|30_49| |—-|——| |O5 | 1| |AX | 0|

|CCY0|30_48| |—-|——| |O5 | 1| |AX | 0|

|BCY0|01_15| |—-|——| |O5 | 1| |AX | 0|

|ACY0|30_15| |—-|——| |O5 | 1| |AX | 0|

5.3.6 CLB_nDI1MUX Minitest

Purpose

Trying to set SLICEM LUT DI1 inputsThese exist for LUTA, LUTB, and LUTC onlyCan either be an external signal, another LUT's data input, or another LUT's carryNote: mux input pattern is irregular

Result

The following bits are set for NI but not NMC31:

```
bit 00_00 ADI1MUX.AI
bit 00_20 BDI1MUX.BI
bit 01_43 BDI1MUX.CI
```

Additionally, test with unknown DI mux bits don't appear near NI bitsThere is something strange going on

5.3.7 CLB_nFFMUX Minitest

Purpose

Result

|AFFMUX|30_00|30_01|30_02|30_03| |——|——|——|——|——| |F78 | 1| 1| | | |CY | 1| | 1| | |O5 | 1| | | 1| |AX | | 1| | | |XOR | | | 1| | |O6 | | | | 1|

5.3.8 CLB_RAM Minitest

Purpose

SLICEM RAM testLUT6 => 64 bitsFocus on 64 bit32 probably uses same O5/O6 stuff128 probably uses same MUX stuffWhy isn't there a 256?

Result

```
RAM128X1D 128-Deep by 1-Wide Dual Port Random Access Memory (Select RAM)
RAM128X1S 128-Deep by 1-Wide Random Access Memory (Select RAM)
RAM256X1S 256-Deep by 1-Wide Random Access Memory (Select RAM)
RAM32M 32-Deep by 8-bit Wide Multi Port Random Access Memory (Select RAM)
RAM32X1D 32-Deep by 1-Wide Static Dual Port Synchronous RAM
RAM32X1S 32-Deep by 1-Wide Static Synchronous RAM
RAM32X1S_1 32-Deep by 1-Wide Static Synchronous RAM with Negative-Edge Clock
RAM32X2S 32-Deep by 2-Wide Static Synchronous RAM

RAM64M 64-Deep by 4-bit Wide Multi Port Random Access Memory (Select RAM)
RAM64X1D 64-Deep by 1-Wide Dual Port Static Synchronous RAM
RAM64X1S 64-Deep by 1-Wide Static Synchronous RAM
RAM64X1S_1 64-Deep by 1-Wide Static Synchronous RAM with Negative-Edge Clock
```

5.3.9 FIXEDPNR Minitest

Purpose

Result

Preliminary result

| |00_48|30_12|31_03| |—-|—-|—-|—-| |FDPE| | | | |FDSE| X | | | |FDCE| | X | X | |FDRE| X | X | X |

5.3.10 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.

5.3.11 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
```

5.3.12 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.

5.3.13 PICORV32-v Minitest

Purpose

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

Result

5.3.14 PICORV32-y Minitest

Purpose

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

Result

5.3.15 ROI_HARNESS Minitest

Purpose

Creates an ROI with clk, inputs, and outputs to use as a partial reconfiguration test harness

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 shortIts 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

5.4 Tools

SymbiFlow/prjxray/tools/

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

5.5 SymbiFlow/prjxray/minitests/roi_harness

Shows how to use a bunch of tools together to patch an existing bitstream with hand-crafted FASM (FPGA assembler).

CHAPTER 6

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

CHAPTER 7

tilegrid.json

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: corres

7.1 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 requird 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"
        ],
```

(continues on next page)

(continued from previous page)

```
        "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.

7.2 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

CHAPTER 8

tileconn.json

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"
                ],
                [
                        "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