Project Trellis Documentation

SymbiFlow Team

Mar 27, 2019

LATTICE ECP5 ARCHITECTURE

Project Trellis documents the Lattice ECP5 architecture (and other related parts) to enable development of open-source tools. Our goal is to provide sufficient information to develop a free and open Verilog to bitstream toolchain for these devices.

CHAPTER

ONE

OVERVIEW

The ECP5 FPGA is arranged internally as a grid of *Tiles*. Each tile contains bits that configure routing and/or the tile's functionality.

Inside the ECP5 there is both *general routing*, which connects nearby tiles together (spanning up to 12 tiles) and *global routing*, which allows high fanout signals to connect to all tiles within a *quadrant* (such as clocks).

CHAPTER

TWO

TILES

The ECP5 FPGA is structured as a grid of tiles. This grid can be visualised by looking at the HTML output of `tile_html.py`.

Tiles have a name, for example *MIB_R13C6* and a type, for example **MIB_DSP2**. The name also encodes a position, in this case row 13, column 6. Multiple tiles may be located in the same grid square.

Tiles also have an offset in the bitstream in terms of frames and bits within frames, and a bitstream size in terms of frames and bits within frames.

2.1 Logic Tiles

All logic tiles are of type **PLC2**.

Logic Tiles contain 4 slices and routing fabric. The routing fabric in a logic tile can connect slices to general and global routing; and connect together general routing wires.

Each slice contains 2 LUTs, 2 flip-flops and fast carry logic. All slices can also be configured to function as distributed RAM.

2.2 Common Interconnect Blocks (CIBs)

Common Interconnect Blocks (CIBs) are used to connect special functions - everything other than slices - to the routing fabric. They are effectively a logic tile with the slices removed, and the signals that would connect to the slice are connected to inputs and outputs from those special functions. Special functions in this context includes IO, EBR, DSPs, PLLs, etc. Note that the names of these signals do not reflect the IO of the special function, but are always named as if they were connected to a slice. Part of Project Trellis will thus be determining this mapping.

There are several types of CIB depending on what they connect to and their location on the chip. For example:

- **CIB** tiles connect to top and bottom IO, and **CIB_LR** connect to left and right IO.
- **CIB_EBR** tiles connect to EBR.
- **CIB_DSP** tiles connect to the hard DSPs.
- **CIB_PLLx** tiles connect to the PLLs.
- **CIB_DCUx** tiles connect to the SERDES duals (DCUs).

Most CIBs contain a **CIBTEST** component, which has an unknown function in Lattice testing but is not used for user designs.

2.3 IO Tiles

There are several different types of IO tiles depending on the position in the device. In all cases, each IO pin is represented by two sites: an IO buffer (`PIO`) and IO registers/gearboxes (`IOLOGIC`).

Depending on the location in the device, IO pins are arranged in quads (A-D) or pairs (A-B).

- IO at the top of the device uses a total of four tiles for each pair of IO. **PIOT0** and **PIOT1** contain `PIOA` and `PIOB` split between them, and **PICT0** and **PICT1** contain `IOLOGICA` and `IOLOGICB`.
- IO at the left and right of the device use a total of three tiles for each quad of IO. Note that in this context *x* is **L** for left IO and **R** for right IO. **PICx1** contains all four PIO, **PICx0** contains `IOLOGICA` and `IOLOGICB`, and **PICx2** usually contains `IOLOGICC` and `IOLOGICD`. In some cases, `IOLOGICC` and `IOLOGICD` are placed inside a **MIB_CIB_LR** tile instead.
- IO at the bottom of the device uses a total of two tiles for each pair of IO. **PICB0** and **PICB1** contain both PIO and IOLOGIC split between them.

Additionally **BANKREFx** tiles contain per-bank IO configuration for reference voltages and VccIO.

2.4 Global Clock Tiles

Several different tiles have functions in global clock routing. See *Global Routing* for more information on exact connections and blocks involved.

- **TAP_DRIVE** and **TAP_DRIVE_CIB** tiles, arranged in several columns and present in all rows, selectively connect vertical global clocks coming in from the spine for the relevant quadrant to horizontal clock routing for the associated row section.
- **CMUX_xx** tiles form the central global clock muxes for each quadrant, selecting clocks from the **xMID** tiles, feeding the spine tiles. In some cases these are split and/or combined with another function (such as DSP or EBR). These also contain the two clock selectors (DCSs).
- **xMID_x** tiles select various input clocks and feed them to the central clock mux, providing a clock gate (DCC) for each clock.
- **ECLK_L** and **ECLK_R** select the IO edge clocks.
- **x_SPINE_xx** tiles selectively connect the outputs from the central clock muxes to the vertical wires feeding the **TAP_DRIVE** s. These are combined with another function, EBR or DSP and located in EBR/DSP rows.

2.5 Embedded Block RAM (EBR)

The EBR is distributed such that 9 columns contain 4 EBRs (each EBR is 18kbit). The tiles containing the EBR sites themselves, and the EBR configuration bits, are named **MIB_EBR0** to **MIB_EBR8**. The EBR is connected to the routing fabric using **CIB_EBR** tiles in the same row as the **MIB_EBRx** tiles. The meaning of *MIB* is unknown, it likely comes from Maco Interface Tile (Maco was a hybrid FPGA/ASIC technology previously used by Lattice).

The EBR configuration is split thus:

EBR	Tiles
0	**MIB_EBR0**, **MIB_EBR1**
1	**MIB_EBR2**, **MIB_EBR3**, **MIB_EBR4**
2	**MIB_EBR4**, **MIB_EBR5**, **MIB_EBR6**
3	**MIB_EBR6**, **MIB_EBR7**, **MIB_EBR8**

EBR initialisation is done using separate commands in the bitstream, not from within the EBR tiles themselves.

2.6 DSP Tiles

DSPs are distributed such that 9 columns contain 2 18x18 sysDSP slices. Two tiles per column (on the same row) contain the DSP sites themselves, and the DSP configuration bits, and are named **MIB_DSP0** to **MIB_DSP8** and **MIB2_DSP0** to **MIB2_DSP8**. The DSPs are connected to the routing fabric using **CIB_DSP** tiles in the same row as the **MIBx_DSPx** tiles.

2.7 System and Config Tiles

There are several tiles, usually only one of each per device, which contain miscellaneous system functions and global device settings. For example:

- **EFBx_PICBx** tiles (which are combined with bottom IO tiles) contain config port related settings such as which ports are enabled after configuration, whether TransFR is enabled, and the speed of the internal oscillator. They also contain configuration related to the global set/reset signal.
- **DTR** contains the Digital Temperature Readout function.
- **POR** contains a single bit to disable power-on reset.
- **OSC** contains the internal oscillator.
- **PVT_COUNTx** are related to Process/Voltage/Temperature compensation and contain a PVTTEST component for Lattice testing.

CHAPTER

THREE

GENERAL ROUTING

The ECP5's general routing is unidirectional and relatively straightforward. They are detailed below. The inputs and outputs of the routing inside tiles are:

- The inputs to BELs are named **A0**-**A7**, **B0**-**B7**, **C0**-**C7**, **D0**-**D7** (LUT inputs), **M0**-**M7** ("miscellaneous" inputs for muxes and other functions); **LSR0** and **LSR1** (local set/reset); **CLK0** and **CLK1** (clock) and **CE0**-**CE3** (clock enable), for logic tiles.
- The outputs are named **F0** to **F7** (LUT outputs) and **Q0** to **Q7** (FF outputs).
- CIB tiles have an identical routing configuration to logic tiles, regardless of what they connect to - effectively, the logic slices are replaced by the special function - however, all the netnames aboved are prefixed with **J**. Fixed arcs connect the CIB signals to the signals inside the special function tile.

Four types of routing resource are available:

- 8 ***X0** wires inside each tile (**H00L0x00**, **H00R0x00**, **V00T0x00**, and **V00B0x00**) do not leave a tile, but can be driven from a variety of internal and external signals; and all of the horizontal or vertical signals are inputs to all of the BEL input muxes.
- 8 **X1** "neighbour" wires originate, and terminate, in each tile (**H01E0x01**, **H01W0x00**, **V01S0x00** and **V01N0x01**). These connect together adjacent tiles in the specified direction, and can be driven by any of the LUT or FF outputs as well as a few other signals.
- 32 **X2** span-2 wires (**H02E0x01**, **H02W0x01**, **V02S0x01** and **V02N0x01**) originate in each tile, each connecting to the two next tiles in a given direction.
- 16 **X6** span-6 wires (**H06E0x03**, **H06W0x03**, **V06S0x03** and **V06N0x03**) originate in each tile connecting to two tiles, with a gap of 2 inbetween each, in a given direction.

In all cases, wires can only be driven inside one tile. **X2** and **X6** inputs only drive muxes in tiles other than the tile they originate in; whereas **X0** and **X1** also "bounce back" internally (this being the very purpose of **X0** tiles).

CHAPTER

FOUR

GLOBAL ROUTING

The ECP5's global clock routing is split into a number of parts. A high level overview of its structure is contained in Lattice Technical Note TN1263.

From a point of view of clock distribution, the device is split into four quadrants: upper left (UL), upper right (UR), lower left (LL) and lower right (LR). Throughout this document, *qq* refers to any quadrant named in this way, *s* refers to a side of the device (B, T, L, or R).

This document is a work in progress! Some information is still incomplete and subject to change.

4.1 Mid Muxes

The mid muxes are located in the middle of each edge of the device, and take clock inputs from a range of sources (clock input pins, PLL ouputs, CLKDIV outputs, etc) and feed them into the centre clock muxes.

Depending on the location, mid muxes have between 12 and 16 outputs to the centre muxes.

Between each mid mux multiplexer output and the centre mux is a Dynamic Clock Control component (DCC), which provides glitch free clock disable.

Inputs to the mid muxes are named as follows:

- **PCLKx_y** for the dedicated clock input pins
- **qqC_PCLKGPLLx_y** for the PLL outputs
- **s_CDIVX_x** for the clock divider outputs
- **qqQ_PCLKCIB_x** and **qqM_PCLKCIBx** have unknown function, most likely a connection to fabric (TODO: check)
- **PCSx_TXCLK_y** and **PCSx_RXCLK_y** are the SERDES transmit and receive clocks
- **SERDES_REFCLK_x** are the SERDES reference clocks

Outputs from the muxes themselves into the DCCs are named **sDCC00CLKI** through **sDCCnnCLKI**.

The outputs from the DCCs into the centre muxes are named **HPFE0000** through **HPFE1300** for the left side; **HPFW0000** through **HPFW1300** for the right side; **VPFN0000** through **VPFN1500** for the bottom side and **VPFS0000** through **VPFS1100** for the top side.

The left and right muxes are located in a single tile, **LMID_0** and **RMID_0** respectively. The top side is split between **TMID_0** and **TMID_1**, and the bottom side between **BMID_0** and **BMID_2V**.

4.2 Centre Muxes

The centre muxes select the 16 global clocks for each quadrant from the outputs of the mid muxes and from a few other sources. They also contain a total of two Dynamic Clock Select blocks, for glitch-free clock switching.

There are four fabric entries to the centre muxes from each of the four quadrants. These connect through DCCs in the same way as the mid mux outputs (but in this case the DCC is considered as part of the centre mux rather than mid mux).

The fabric entries come from nearby CIBs, but the exact net still need to be traced.

Inputs to the mid muxes (and DCSs) are named as follows:

- **HPFExx00**, **HPFWxx00**, **VPFNxx00** and **VPFSxx00** for the mid mux (via DCC) outputs
- **qqCPCLKCIB0** for the fabric clock inputs (via DCC)
- **DCSx** for the DCS outputs
- **VCC_INT** for constant one

Outputs from the centre muxes, going into the spine tiles, are named **qqPCLK0** through **qqPCLK15**.

Centre muxes also contain *CLKTEST*, a component with unknown function used in Lattice's testing.

The exact split of centre mux functionality between tiles varies depending on device, this is an example for the LFE5U-85F.

Tile	Tile Type	Functions	Notes
MIB_R22C66	EBR_CMUX_UL	DCS0 config, CLKTEST	Shared with EBR
MIB_R22C67	CMUX_UL_0	UL clocks mux, DCS0CLK0 input mux, DCC2	
MIB_R22C68	CMUX_UR_0	UR clocks mux, DCS0CLK1 input mux, DCC3	
MIB_R22C69	EBR_CMUX_UR	CLKTEST	Shared with EBR
MIB_R70C66	EBR_CMUX_LL	DCS1 config, CLKTEST	Shared with EBR
MIB_R70C67	CMUX_LL_0	LL clocks mux, DCS1CLK0 input mux, DCC0	
MIB_R70C68	CMUX_LR_0	LR clocks mux, DCS1CLK1 input mux, DCC1	
MIB_R70C69	EBR_CMUX_LR	CLKTEST	Shared with EBR

4.3 Spine Tiles

The outputs from the centre muxes go horizontally into each quadrant and connect to a few spine tiles (there are three spine tiles per quadrant in the 85k part). Spine tiles are located adjacent to a column of TAP_DRIVEs (see next section) and are combined with another function (EBR or DSP).

The purpose of a spine tile is to selectively connect the centre mux outputs to the vertical global wires feeding the adjacent column of TAP_DRIVEs through a buffer. There is one buffer per wire, with a 1:1 mapping - the only reason spine tiles are configurable is to disable unused buffers to save power, there is no other routing or selection capability in them.

The inputs to spine tiles from the quadrant's centre mux are named **qqPCLK0** through **qqPCLK15**.

The outputs are named **VPTX0000** through **VPTX15000**, which feed a column of tap drives.

4.4 TAP_DRIVE Tiles

TAP_DRIVE tiles are arranged in columns throughout the device.

The purpose of TAP_DRIVE tiles is to selectively connect the vertical global wires coming out of the adjacent spine tile to horizontal wires serving tiles to the left and right of the TAP_DRIVE. A TAP_DRIVE will typically serve a row of about 20 tiles, 10 to the left and 10 to the right.

Like spine tiles, TAP_DRIVE tiles have a 1:1 input-output mapping and only offer the ability to turn on/of buffers to save power.

The outputs are named **HPBX0000** through **HPBX15000**, with a net location on the left or right for the left or right outputs (signified as **L_** or **R_** in the Project Trellis database.

4.5 Non-Clock Global Usage

Inside PLBs, global nets can not only be connected to the clock signal, but also to clock enable, set/reset and general local wires. This does not seem to be commonly used by Diamond, which prefers to use general routing and the GSR signal.

Not all globals can be used for all functions, the allowed usage depending on net is shown below.

Global	CLK	LSR	CEN	Local
0	Y			Y
1	Y			Y
2	Y			Y
3	Y			Y
4	Y	Y		Y
5	Y	Y		Y
6	Y	Y		Y
7	Y	Y		Y
8	Y	Y		
9	Y		Y	
10	Y		Y	
11	Y		Y	
12	Y		Y	
13	Y		Y	
14	Y	Y	Y	
15	Y	Y	Y	

CHAPTER

FIVE

BITSTREAM FORMAT

Some documentation on the ECP5 bitstream format is published by Lattice themselves in th ECP5 sysCONFIG Usage Guide (TN1260).

5.1 Basic Structure

The ECP5 is primarily byte oriented and always byte aligned. Multi-byte words are always in big endian format.

Before the start of the bitstream itself is a comment section, which starts with FF 00 and 00 FF. Inside it are several null-terminated strings used as metadata by Lattice. The start of the bitstream is demarcated by a preamable, 0xFFFFBDB3. This is then followed by a 0xFFFFFFFF dummy section and then the start of functional commands.

At minimum, a bitstream command is an 8 bit command type field, then 24 bits of command information. The MSB of command information is set to 1 if a CRC16 follows the command. The other 23 bits are command-specific, but usually zeros. Then follows a command-specific number of payload bytes, and the CRC16 if applicable.

The CRC16 is accumulated over all commands until a CRC16 check is reached. It is not reset at the end of commands without a CRC16 check - except the `LSC_RESET_CRC` command, and after the actual bitstream payload (`LSC_PROG_INCR_RTI` or `LSC_PROG_INCR_CMP`).

The CRC16 is calculated using the polynomial 0x8005 with no bit reversal. This algorithm is sometimes known as "CRC16-BUYPASS".

> NB: Lattice's documents talk about the CRC16 being flipped. This is based on standard CRC16 code with reversal, effectively performing double bit reversal. CRC16 code with no bit reversal at all matches the actual format.

5.2 Control Commands

Control commands seen in a typical uncompressed, unencrypted bitstream are:

Command	Hex	Parameters	Description
Dummy	FF	N/A	Ignored, used for padding
LSC_RESET_CRC	3B	24 bit info: all 0	Resets the CRC16 register
VERIFY_ID	E2	• 24 bit info: all 0 • 32 bit device JTAG ID	This checks the actual device ID against the given value and fails if they do not match.
LSC_WRITE_COMP_DIC	02	• 24 bit info: all 0 • 8 bit Pattern7 • ... (6 more patterns) • 8 bit Pattern0	This stores the 8 most common bytes in the frames
LSC_PROG_CNTRL0	22	• 24 bit info: all 0 • 32 bit Ctl-Reg0 value	This sets the value of device control register 0 Normally 0x40000000
LSC_INIT_ADDRESS	46	24 bit info: all 0	Resets the frame address register
ISC_PROGRAM_SECURITY	CE	24 bit info: all 0	Program the security bit (prevents readback (?))
ISC_PROGRAM_USERCODE	C2	• CRC bit set, 23 bits 0 • 32 bit USER-CODE value	Sets the value of the USERCODE register
ISC_PROGRAM_DONE	5E	24 bit info: all 0	End of bitstream, set DONE high

5.3 Configuration Data

The FPGA configuration data itself is programmed by using command LSC_PROG_INCR_RTI (0x82) if no compression is used and command LSC_PROG_INCR_CMP (0xB8) when using compression. Following either of these commands, there are some setup bits:

- 1 bit: CRC16 comparison flag, normally set
- 1 bit: CRC16 comparison at end flag, normally cleared = CRC check after every frame
- 1 bit: dummy bits setting, normally cleared = include dummy bits in bitstream
- 1 bit: dummy byte setting, normally cleared = use following bits as number of dummy bytes
- 4 bits: number of dummy bytes between frames, usually 1
- 16 bits: number of configuration frames

This is then followed by a number of frames, each in the following format:

- The configuration frame itself (compressed in the case of the `LSC_PROG_INCR_RTI` command), such that bit 0 bit 0 of the first byte is the MSB of the frame, bit 7 of the first byte the MSB-7 and bit 0 of the last byte (if there are no dummy bits) being the LSB of the frame.
- Any dummy bits needed to pad the frame to a whole number of bytes.
- **If the second flag is cleared (see above) a CRC-16 checksum:**
  - For the first frame, this also covers any other commands sent before the programming command but after a CRC reset, and the programming command itself.
  - For subsequent frames, this excludes dummy bytes between frames
- Dummy 0xFF bytes, usually only 1

The highest numbered frame in the chip is sent first.

If the second flag is set there's no CRC sent in between frames but there's still one CRC-16 checksum after all the frames (this also covers any other commands sent before the programming command but after a CRC reset, and the programming command itself.).

Separate commands are used if EBR needs to be configured in the bitstream. EBR data can't use compression. `EBR_ADDRESS` (0xF6) is used to select the EBR to program and the starting address in the EBR; and `LSC_EBR_WRITE` (0xB2) is used to program the EBR itself using 72-bit frames. The specifics of these still need to be documented.

5.4 Compression Algorithm

- Before compression, the frame is left padded with zeroes (0) to make the data frame 64-bit bounded.
- After compressing the frame data, the resulting bits are right padded with zeroes (0) to make the data

frame byte bounded.

After padding, every byte in the bitstream is compressed by a simple prefix-free code with just 4 cases:

Code	Argument	Length	Encoded byte
0		1	zero
100xxx	bit position	6	byte with 1 bit set
101xxx	byte index	6	stored byte
11xxxxxxx	lit. byte	10	all others

- The first case is for the byte zero (`00000000`). That's just represented by a single zero bit (`0`).
- The second case is for bytes with just one bit set. After a `100` the set bit position is encoded in the following 3 bits. For example the byte `00100000` is encoded as `100101` because only the bit 5 is set (with 0 being the lsb and 7 the msb).
- The third case is for selecting one of the bytes stored by the `LSC_WRITE_COMP_DIC` instruction. Those bytes are selected as the 8 most common bytes (ignoring the zero bytes and the bytes with just one bit set, because those are encoded with the two previous cases). After a `101` the number of the selected pattern is encoded with 3 bits. For example to select pattern3 the code would be `101011`.
- The fourth case is for all remaining bytes. In that case after a `11` the complete byte is copied. For example byte `11001010` would be encoded as `1111001010`.

5.5 Device-Specific Information

Device	Device ID	Frames	Config Bits per Frame	Dummy Bits per Frame
LFE5U-25	0x41111043	7562	592	0
LFE5UM-25	0x01111043	7562	592	0
LFE5U-45	0x41112043	9470	846	2
LFE5UM-45	0x01112043	9470	846	2
LFE5U-85	0x41113043	13294	1136	0
LFE5UM-85	0x01113043	13294	1136	0

CHAPTER

SIX

GLOSSARY

Arc A programmable connection between two nodes. Most arcs are unidirectional buffers that connect in one direction only, a few are bidirectional switch transistors (however these are mostly used in situations where this is one useful direction anyway).

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.

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.

Database Contains information about programmable configuration bits, arc enable bits, and how wires are connected between tiles.

FF

Flip flop 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 A frame contains a row of bits used to configure the FPGA, and is the basic unit that a bitstream is split into. The number of bits per frame, and frames per device varies for different ECP5 chips.

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

General Routing Routing that connects together nearby *tiles* horizontally or vertically, spanning up to about 15 tiles at most. Called "shortwires" in Lattice Diamond, but this also refers to internal routing.

Global Routing Routing that connects to all *tiles* in a quadrant for logic, and all tiles on an edge for IO. Normally used for routing clocks, or occasionally other high fanout signals. Called "longwires" in Lattice Diamond.

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).

Internal Routing Routing that does not leave a single *tile*.

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.

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.

Quadrant The ECP5 FPGAs are arranged into four quadrants for the purpose of global signal distribution, upper left (UL), upper right (UR), lower left (LL) and lower right (LR).

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

Site Locations inside a tile that can contain an instance of a primitive.

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 programs included with Diamond to create debugging outputs.

Tile Fundamental unit of physical structure containing a single type of resource or function. The whole chip is a grid of tiles, however, multiple tiles may exist at one grid location.

Wire Physical wire within a *tile*.

CHAPTER

SEVEN

DATABASE DEVELOPMENT OVERVIEW

A targeted approach is used to construct bitstream databases as quickly as possible. The planned flow is for every routing mux or configuration setting we want to determine, to create post-place-and-route designs for all possibilities then run them through bitstream generation and compare the outputs.

7.1 NCL Files

NCL files are file format used by Lattice (originating from NeoCAD), which are a textual representation of the internal design database. Although there is little documentation on them, the format is relatively straightforward. There are tools included with Diamond to convert the design database to and from a NCL file (`ncd2ncl` and `ncl2ncd`). These are wrapped by the script `diamond.sh` included in Project Trellis, that allows two possibilities:

- If given a Verilog file, it will use Diamond for synthesis and PAR, and as well as a bitstream also dump the post-place-and-route design as a NCL file. This way you can inspect how the design maps to an NCL file, and the routing and configuration inside the tile.
- If given a NCL file, it will skip synthesis and PAR. It will convert the NCL file to a design database, then generate a bitstream from that.

In the planned fuzzing flow, we will first create a Verilog design for what we want to fuzz by hand, and convert it to an NCL file. Then we will manually create a template NCL file containing only the mux/config to be fuzzed. The Python fuzzer script will then subsitute this template file for each fuzz possibility.

A template file for LUT initialisation is shown as an example:

```
::FROM-WRITER;
design top
{
   device
   {
      architecture sa5p00;
      device LFE5U-25F;
      package CABGA381;
      performance "8";
   }

   comp SLICE_0
      [,,,,A0,B0,D0,C0,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,]
   {
      logical
      {
         cellmodel-name SLICE;
         program "MODE:LOGIC "
```

(continues on next page)

(continued from previous page)

```
                "K0::H0=${lut_func} "
                "F0:F ";
        primitive K0 i3_4_lut;
    }
    site R2C2A;
  }

}
```

The NCL file contains information about the device, components and routing (routing is not included in this example). As this was from a LUT initialisation fuzzer, ${lut_func} will be replaced by a function corresponding to the LUT init bit to be fuzzed (NCL files require an expression for LUT initialisation, rather than a series of bits).

7.2 Fuzzers

There are three types of fuzzer in use in Project Trellis. They are routing fuzzers, non-routing fuzzers and meta-fuzzers.

Routing fuzzers use the helper functions in `util/fuzz/nonrouting.py`. These will generally follow the following algorithm:

- Use the Tcl API to discover all of the netnames at the target location
- Use the Tcl API to discover the arcs associated with those netnames
- Apply filters to remove netnames and/or arcs not applicable to the current fuzzer
  - For example, when fuzzing a CIB_EBR you would ensure to include CIB signals but exclude EBR signals
  - When fuzzing a EBR you would conversely filter out the CIB signals and include EBR signals.
- Create a reference “empty” bitstream
- For every arc discovered above:
  - Create a bitstream containing the arc using a NCL file
  - Compare the bitstream against the empty bitstream:
    - If there was a change outside the tile(s) of interest, the arc is ignored
    - If there was a change to any of the tile(s) of interest, add a configurable mux arc to the database of the relevant tile
    - If there was no change at all, add the arc as a fixed connection to the database of the first specified tile

Note that routing fuzzers for special function tiles (such as PIO, EBR, etc) are primarily intended to find fixed connections to CIBs and within special functions rather than significant amounts of configurable routing, but the above algorithm is still used for consistency (and because it is not possible to know a priori whether an arc is configurable or fixed).

Non-routing fuzzers are intended to fuzz configuration such as LUT initialisation, flip-flop settings or IO type. It is possible to fuzz either words or enums. Words are for settings that naturally comprise of one or more bits, such as LUT initialisation. Enums are for settings with multiple textual values.

The algorithm for word settings is to create one bitstream for each word bit with that bit set, and compare to an all-zero bitstream to determine the change for that bit. These are also optionally compared against an empty bitstream with the setting/feature entirely missing to determine a default value, which is not always all-zeros.

The algorithm for enum settings is to create bitstreams with all possible enum values, in each case storing the CRAM of all tiles of interest. These are then compared to determine the set of bits affected by the enum, and in each case the bit values for each possible enum value.

Creating non-routing requires more work than routing fuzzers. The settings of interest, possible values, and how to create them in the Ncl or Verilog input must all be included in the fuzzer script.

Finally meta-fuzzers do not do any fuzzing but perform other necessary manipulations on the database during the fuzzing flow. For example, these may copy config bits from one tile to other tile types which have identical configuration in order to reduce the time needed for fuzzing.

CHAPTER

EIGHT

LIBTRELLIS OVERVIEW

libtrellis is a C++ library containing utilities to manipulate ECP5 bitstreams, and the databases that correspond tile bits to functionality (routing and configuration). Although libtrellis can be used as a standard shared library, its primary use in Project Trellis is as a Python module (called pytrellis), imported by the fuzzing and utility scripts. The C++ classes are bound to Python using Boost::Python.

8.1 Bitstream

This class provides functionality to read and write Lattice bitstream files, parse their commands, and convert them into a chip's configuration memory (in terms of frames and bits).

To read a bitstream, use `read_bit` to create a `Bitstream` object, then call `deserialise_chip` on that to create a `Chip`.

8.2 Chip

This represents a configured FPGA, in terms of its configuration memory (`CRAM`), tiles and metadata. You can either use `deserialise_chip` on a bitstream to construct a Chip from an existing bitstream, or construct a Chip by device name or IDCODE.

The `ChipInfo` structure contains information for a particular FPGA device.

8.3 CRAM

This class stores the entire configuration data of the FPGA, as a 2D array (frames and bits). Although the array can be accessed directly, many cases will use `CRAMView` instead. `CRAMView` provides a read/write view of a window of the CRAM. This is usually used to represent the configuration memory of a single tile, and takes frame and bit offsets and lengths.

Subtracting two `CRAMView` s, if they are the same size, will produce a `CRAMDelta`, a list of the changes between the two memories. This is useful for fuzzing or comparing bitstreams.

8.4 Tile

This represents a tile of the FPGA. It includes a `CRAMView` to represent the configuration memory of the tile.

8.5 TileConfig

This represents the actual configuration of a tile, in terms of arcs (programmable connections), config words (such as LUT initialisation) and config enums (such as IO type). It is the result of decoding the tile CRAM content using the bit database, and can be converted to a FASM-like format.

The contents of `TileConfig` are `ConfigArc` for connections, `ConfigWord` for non-routing configuration words (which also includes single config bits), `ConfigEnum` for enum configurations with multiple textual values, and `ConfigUnknown` for unknown bits not found in the database, which are simply stored as a frame, bit reference.

The contents of a tile's configuration RAM can be converted to and from a `TileConfig` by using the `tile_cram_to_config` and `config_to_tile_cram` methods on the `TileBitDatabase` instance for the tile.

8.6 TileBitDatabase

There will always be only one `TileBitDatabase` for each tile type, which is enforced by requiring calling the function `get_tile_bitdata` (in `Database.cpp`) to obtain a `shared_ptr` to the `TileBitDatabase`.

The `TileBitDatabase` stored the function of all bits in the tile, in terms of the following constructs:

- Muxes (`MuxBits`) specify a list of arcs that can drive a given node. Each arc (`ArcData`) contains specifies source, sink and the list of bits that enable it as a `BitGroup`.
- Config words (`WordSettingBits`) specify non-routing configuration settings that are arranged as one or more bits. Each config bit has an associated list of bits that enable it. This would be used both for single-bit settings and configuration such as LUT initialisation and PLL dividers.
- Config enums (`EnumSettingBits`) specify non-routing configuration settings that have a set of possible textual values, used for either modes/types (i.e. IO type) or occasionally "special" muxes not part of general routing. These are specified as a map between possible values and the bits that enable those values.

`TileBitDatabase` instances can be modified during runtime, in a thread-safe way, to enable parallel fuzzing. They can be saved back to disk using the `save` method.

They can also be used to convert between tile CRAM data and higher level tile config, as described above.

8.7 RoutingGraph

`RoutingGraph` and related structures are designed to store a complete routing graph for the Chip. `RoutingWire` represents wires, `RoutingArc` represents arcs between wires and `RoutingBel` represents a Bel (logic element). To reduce memory usage, `ident_t`, an index into a string store, is used instead of using strings directly. Use `RoutingGraph.ident` to convert from string to `ident_t`, and `RoutingGraph.to_str` to convert to a string.

8.8 DedupChipdb

This is an experimental part of libtrellis to "deduplicate" the repetition in the routing graph, by converting it to relative coordinates and then storing identical tile locations only once. The data produced is intended to be exported to a database for a place and route tool using the Python API.

8.9 ChipConfig

ChipConfig contains the high-level configuration for the entire chip, including all tiles and metadata. It can be directly converted to or from a high-level configuration text file. For more information see *Text Config Documentation*.

CHAPTER

NINE

TEXTUAL CONFIGURATION FORMAT

Project Trellis supports a simple text-based configuration format so that place-and-route, design manipulation and design analysis tools do not have to deal with bitstream specifics, nor link directly to libtrellis.

9.1 Overview

The text-based configuration format uses standard ASCII text files. `#` denotes a comment that continues until the end of the line. `.` at the start of the line followed by the command type denotes a command. Command options follow on the line, some commands may then have data on subsequent lines until the next command

9.2 Non-Tile Configuration

`.device <device name>` specifies the device type, for example `.device LFE5U-85F`. This should always be the first command in a file.

`.comment <comment>` marks the rest of a line as a comment that is included in the header of the bitstream. The FPGA ignores these, but they may be required if you wish to use vendor programming or deployment tools with the bitstream.

9.3 Tile Configuration

`.tile <tile name>` denotes the start of a tile. Note that Project Trellis tile names are the Lattice tile name followed by the colon and the tile type, for example `MIB_R22C5:MIB_DSP1`. This is because the Lattice tile names on their own are not unique.

Inside a tile there can be four entries: arcs, words, enums and unknown bits.

`arc: <sink> <source>` enables an arc inside the tile, using the same Trellis relative netnames as used in the database, for example `arc: S3_V06S0303 E1_H01W0100`.

`word: <name> <value>` sets the value of a configuration word (i.e. a configuration value that can reasonable be split into bits, such as LUT initialisation). The value is always in binary, MSB first. For example `word: SLICEC.K0.INIT 0101010101010101` configures LUT0 in SLICEC to be an inverter.

`enum: <name> <value>` sets the value of a configuration enum (i.e. a configuration value with multiple textual values). In all cases one of the values from the Trellis database must be used. For example `enum: PIOA.BASE_TYPE INPUT_LVCMOS25` configures PIOA as a LVCMOS25 input.

`unknown: F<frame>B<bit>` sets an unknown bit, specified by tile-relative frame and bit. For example `unknown: F95B0` sets unknown bit 0 in frame 95 in the tile.

Words and enums will not be included when converting a bitstream to textual configuration if they are at their default value (i.e. the value they would have in a bitstream generated from an empty design).

Beware that some configuration words and enums are split across multiple tiles in features such as IO, EBR and DSPs. To generate correct bitstreams, they must be included in every tile where they appear. Currently the value matcher, when converting bitstreams to config looks at each tile individually, so may pick a config that is correct in each tile but not overall. This will be fixed in the future.

9.4 Conversion

You can use `libtrellis/examples/bit_to_config.py <bitstream>` to convert a bitstream to a text config file, and `libtrellis/examples/config_to_bit.py <config> <bitstream>` to convert from config to a bitstream. C++ command line tools and an install script are currently being developed.

INDEX