Mistral documentation

Release 1.0

Dec 27, 2020

CONTENTS

1 The Cyclone V FPGA **1**
- 1.1 The FPGAs . . . 1
- 1.2 Bitstream stucture . . . 2
- 1.3 Logic blocks . . . 2
- 1.4 Routing network . . . 3
- 1.5 Programmable inverters . . . 3

2 CycloneV internals description **5**
- 2.1 Routing network . . . 5
- 2.2 Inner logic blocks . . . 6
- 2.3 Peripheral logic blocks . . . 6

3 CycloneV library usage **11**
- 3.1 Library structure . . . 11
- 3.2 Packages . . . 11
- 3.3 Model information . . . 12
- 3.4 pos, rnode and pnode . . . 13
- 3.5 Routing network management . . . 14
- 3.6 Logic block management . . . 15
- 3.7 Inverters management . . . 16
- 3.8 Pin/package management . . . 16
- 3.9 Options . . . 17
- 3.10 Bitstream management . . . 18

4 The mistral-cv command-line program **19**
- 4.1 models . . . 19
- 4.2 routes . . . 19
- 4.3 routes2 . . . 19
- 4.4 cycle . . . 19
- 4.5 bels . . . 20
- 4.6 decomp . . . 20
- 4.7 comp . . . 20
- 4.8 diff . . . 20

5 Mistral CycloneV library internals **21**
- 5.1 Structure . . . 21
- 5.2 Routing data . . . 21
- 5.3 Block muxes . . . 22
- 5.4 Logic blocks . . . 23
- 5.5 Inverters . . . 23

5.6 Forced-1 bits 23
5.7 Packages 23
5.8 Models 23

CHAPTER

ONE

THE CYCLONE V FPGA

1.1 The FPGAs

The Cyclone V is a series of FPGAs produced initially by Altera, now Intel. It is based on a series of seven dies with varying levels of capability, which is then derived into more than 400 SKUs with variations in speed, temperature range, and enabled internal hardware.

As pretty much every FPGA out there, the dies are organized in grids.

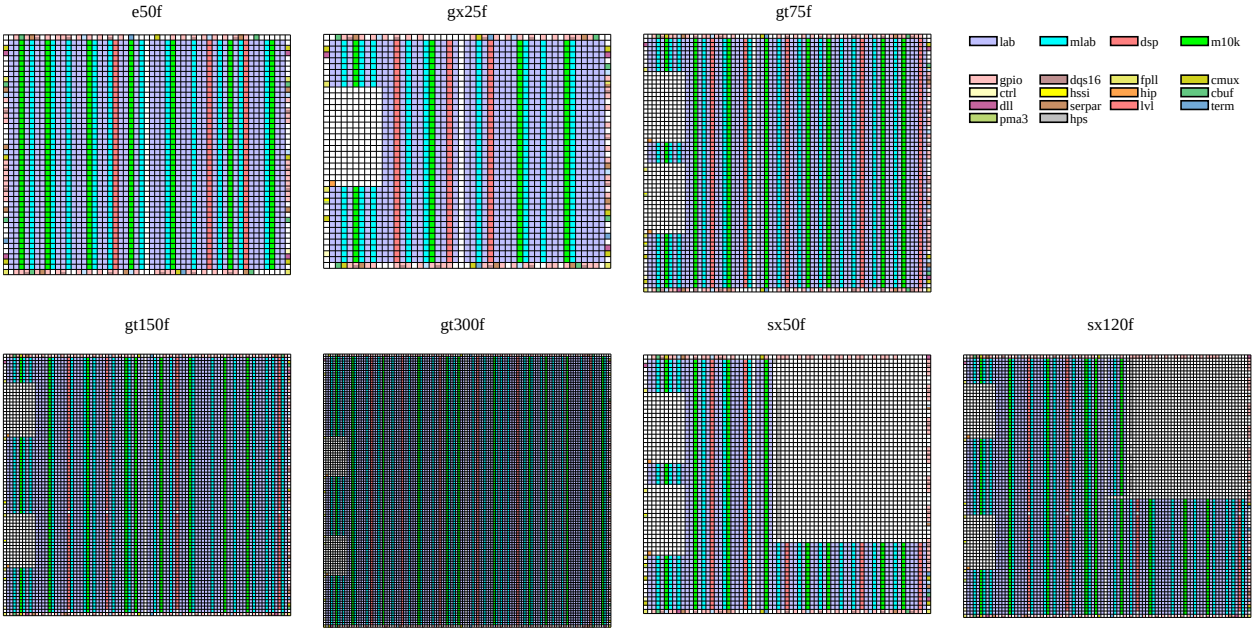

Fig. 1: Floor plan of the seven die types

The FPGA, structurally, is a set of logic blocks of different types communicating with each other either through direct links or through a large routing network that spans the whole grid.

Some of the logic blocks take visible floor space. Specifically, the notches on the left are the space taken by the high speed serial interfaces (hssi and pma3). Also, the top-right corner in the sx50f and sx120f variants is used to fit the hps, a dual-core arm.

1.2 Bitstream stucture

The bitstream is built from three rams:

- Option ram
- Peripheral ram
- Configuration ram

The option ram is composed of 32 blocks of 40 bits, of which only 12 are actually used. It includes the global configurations for the chip, such as the jtag user id, the programming voltage, the internal oscillator configuration, etc.

The peripheral ram stores the configuration of all the blocks situated on the borders of the chip, e.g. everything outside of labs, mlabs, dsps and m10ks. It is built of 13 to 16 blocks of bits that are sent through shift registers to the tiles.

The configuration ram stores the configuration of the labs, mlabs, dsps and m10ks, plus all the routing configuration. It also includes the programmable inverters which allows inverting essentially all the inputs to the peripheral blocks. It is organised as a rectangle of bits.

Die	Tiles	Pram	Cram
e50f	55x46	51101	4958x3928
gx25f	49x40	54083	3856x3412
gt75f	69x62	90162	6006x5304
gt150f	90x82	113922	7605x7024
gt300f	122x116	130828	10038x9948
sx50f	69x62	80505	6006x5304
sx120f	90x82	99574	7605x7024

1.3 Logic blocks

The logic blocks are of two categories, the inner blocks and the peripheral blocks. To a first approximation all the inner blocks are configured through configuration ram, and the peripheral blocks through the peripheral ram. It only matters where it comes to partial reconfiguration, because only the configuration ram can be dynamically modified. We do not yet support it though.

The inner blocks are:

- lab: a logic blocks group with 20 LUTs with 5 inputs and 40 Flip-Flops.
- mlab: a lab that can be reconfigured as 64*20 bits of ram
- dsp: a flexible multiply-add block
- m10k: a block of 10240 bits of dual-ported memory

The peripheral blocks are:

- gpio: general-purpose i/o, a block that controls up to 4 package pins
- dqs16: a block that manage differential input/output for 4 gpio blocks, e.g. up to 16 pins
- fpll: a fractional PLL
- cmux: the clock muxes that drive the clock part of the routing network
- ctrl: the control block with things like jtag
- hssi: the high speed serial interfaces

- hip: the pcie interfaces
- cbuf: a clock buffer for the dqs16
- dll: a delay-locked loop for the dqs16
- serpar: TODO
- lvl: TODO
- term: termination control blocks
- pma3: manages the channels of the hssi
- hmc: hardware memory controller, a block managing sdr/ddr ram interfaces
- hps: a series of 37 blocks managing the interface with the integrated dual-core arm

All of these blocks are configured similarly, through the setup of block muxes. They can be of 4 types: * Boolean * Symbolic, where the choice is between alphanumeric states * Numeric, where the choice is between a fixed set of numeric value * Ram, where a series of bits can be set to any value

Configuring that part of the FPGA consists of configuring the muxes associated to each block.

1.4 Routing network

A massive routing network is present all over the FPGA. It has two almost-disjoint parts. The data network has a series of inputs, connected to the outputs of all the blocks, and a series of outputs that go to data inputs of the blocks. The clock network consists of 16 global clocks signals that cover the whole FPGA, up to 88 regional clocks that cover an half of the FPGA, and when an hssi is present a series of horizontal peripheral clocks that are driven by the serial communications. Global and regional clock signals are driven by dedicated cmux blocks (not the fpll in particular, but they do have dedicated connections to the cmuxes).

These two networks join on data/clock muxes, which allow peripheral blocks to select for their clock-like inputs which network the signal should come from.

1.5 Programmable inverters

Essentially every output of the routing network that enters a peripheral block can optionally be inverted by activating the associated configuration bit.

CHAPTER

TWO

CYCLONEV INTERNALS DESCRIPTION

2.1 Routing network

The routing network follows a single-driver structure: a number of inputs are grouped together in one place, one is selected through the configuration, then it is amplified and used to drive a metal line. There is also usually one bit configuration to disable the driver, which can be all-off (probably leaving the line floating) or a specific combination to select vcc. The drivers correspond to a 2d pattern in the configuration ram. There are 70 different patterns, configured by 1 to 18 bits and mixing 1 to 44 inputs.

The network itself can be split in two parts: the data network and the clock network.

The data network is a grid of connections. Horizontal lines (H14, H6 and H3, numbered by the number of tiles they span) and vertical lines (V12, V4 and V2) helped by wire muxes (WM) connect to each over to ensure routing over the whole surface. Then at the tile level tile-data dispatch (TD) nodes allow to select between the available signals.

Generic output (GOUT) nodes then select between TD nodes to connect to logic blocks inputs. Logic block outputs go to Generic Input (GIN) nodes which feed in the connections. In addition a dedicated network, the Loopback dispatch (LD) connects some of the outputs from the labs/mlabs to their inputs for fast local data routing.

The clock network is more of a top-down structure. The top structures are Global clocks (GCLK), Regional clocks (RCLK) and Peripheral clocks (PCLK). They're all driven by specialized logic blocks we call Clock Muxes (cmux). There are two horizontal cmux in the middle of the top and bottom borders, each driving 4 GCLK and 20 RCLK, two vertical in the middle of the left and right borders each driving 4 GCLK and 12 RCLK, and 3 to 4 in the corners driving 6 RCLK each. The dies including an HPS (sx50f and sx120f) are missing the top-right cmux plus some of the middle-of-border-driven RCLK. That gives a total of 16 GCLK and 66 to 88 RCLK. In addition PCLK start from HSSI blocks to distribute serial clocks to the network.

The GCLK span the whole grid. A RCLK spans half the grid. A PCLK spans a number of tiles horizontally to its right.

The second level is Sector clocks, SCLK, which spans small rectangular zones of tiles and connect from GCLK, RCLK and PCLK. The on the third level, connecting from SCLK, is Horizontal clocks (HCLK) spanning 10-15 horizontal tiles and Border clocks (BCLK) rooted regularly on the top and bottom borders. Finally Tile clocks (TCLK) connect from HCLK and BCLK and distribute the clocks within a tile.

In addition the PMUX nodes at the entrance of plls select between SCLKs, and the GCLKFB and RCLKFB bring back feedback signals from the cmux to the pll.

Inner blocks directly connect to TCLK and have internal muxes to select between clock and data inputs for their control. Peripheral blocks tend to use a secondary structure composed from a TDMUX that selects one TD between multiple ones followed by a DCMUX that selects between the TDMUX and a TCLK so that their clock-like inputs can be driven from either a clock or a data signal.

Most GOUT and DCMUX connected to inputs to peripheral blocks are also provided with an optional inverter.

2.2 Inner logic blocks

2.2.1 LAB

The LABs are the main combinatorial and register blocks of the FPGA. A LAB tile includes 10 sub-blocks with 64 bits of LUT splitted in 6 parts, four Flip-Flops, two 1-bit adders and a lot of routing logic. In addition a common control subblock selects and dispatches clock, enable, clear, etc signals.

2.2.2 MLAB

A MLAB is a lab that can optionally be turned into a 640-bits RAM or ROM. The wiring is identical to the LAB, only some additional muxes are provided to select the RAM/ROM mode.

TODO: address/data wiring in RAM/ROM mode.

2.2.3 DSP

The DSP blocks provide a multiply-adder with either three 9x9, two 18x18 or one 27x27 multiply, and the 64-bits accumulator. Its large number of inputs and output makes it span two tiles vertically.

TODO: everything, GOUT/GIN/DCMUX mapping is done

2.2.4 M10K

The M10K blocks provide 10240 (256*40) bits of dual-ported rom or ram.

TODO: everything, GOUT/GIN/DCMUX mapping is done

2.3 Peripheral logic blocks

2.3.1 GPIO

The GPIO blocks connect the FPGA with the exterior through the package pins. Each block controls 4 pads, which are connected to up to 4 pins.

TODO: everything, GOUT/GIN/DCMUX mapping is done

2.3.2 DQS16

The DQS16 blocks handle differential signaling protocols. Each supervises 4 GPIO blocks for a total of 16 signals, hence their name.

TODO: everything

2.3.3 FPLL

The Fractional PLL blocks synthesize 9 frequencies from an input with integer or fractional ratios.

TODO: everything, GOUT/GIN/DCMUX mapping is done

2.3.4 CMUX

The Clock mux blocks drive the GCLK and the RCLK.

TODO: fpll feedback lines

2.3.5 CTRL

The Control block gives access to a number of anciliary functions of the FPGA.

TODO: everything, GOUT/GIN/DCMUX mapping is done

2.3.6 HSSI

The High speed serial interface blocks control the serializing/deserializing capabilities of the FPGA.

TODO: everything

2.3.7 HIP

The PCIe Hard-IP blocks control the PCIe interfaces of the FPGA.

TODO: everything

2.3.8 DLL

The Delay-Locked loop does phase control for the DQS16.

TODO: everything

2.3.9 SERPAR

Unclear yet.

TODO: everything

2.3.10 LVL

The Leveling Delay Chain does something linked to the DQS16.

TODO: everything

2.3.11 TERM

The TERM blocks control the On-Chip Termination circuitry

TODO: everything

2.3.12 PMA3

The PMA3 blocks control triplets of channels used with the HSSI.

TODO: everything

2.3.13 HMC

The Hardware memory controller controls sets of GPIOs to implement modern SDR and DDR memory interfaces. In the sx dies one of them is taken over by the HPS. They can be bypassed in favor of direct access to the GPIOs.

TODO: everything, and in particular the hmc-input -> GPIO input mapping when bypassed.

2.3.14 HPS

The interface between the FPGA and the Hard processor system is done through 37 specialized blocks or 28 different types.

TODO: everything. GOUT/GIN/DCMUX mapping is done except for HPS_CLOCKS.

HPS_BOOT

HPS_CLOCKS

HPS_CLOCKS_RESETS

HPS_CROSS_TRIGGER

HPS_DBG_APB

HPS_DMA

HPS_FPGA2HPS

HPS_FPGA2SDRAM

HPS_HPS2FPGA

HPS_HPS2FPGA_LIGHT_WEIGHT

HPS_INTERRUPTS

HPS_JTAG

HPS_LOAN_IO

HPS_MPU_EVENT_STANDBY

HPS_MPU_GENERAL_PURPOSE

HPS_PERIPHERAL_CAN

(2 blocks)

HPS_PERIPHERAL_EMAC

(2 blocks)

HPS_PERIPHERAL_I2C

(4 blocks)

HPS_PERIPHERAL_NAND

HPS_PERIPHERAL_QSPI

HPS_PERIPHERAL_SDMMC

HPS_PERIPHERAL_SPI_MASTER

(2 blocks)

HPS_PERIPHERAL_SPI_SLAVE

(2 blocks)

HPS_PERIPHERAL_UART

(2 blocks)

HPS_PERIPHERAL_USB

(2 blocks)

HPS_STM_EVENT

HPS_TEST

HPS_TPIU_TRACE

CHAPTER

THREE

CYCLONEV LIBRARY USAGE

3.1 Library structure

The library provides a CycloneV class in the mistral namespace. Information is provided to allow to choose a CycloneV::Model object which represents a sold FPGA variant. Then a CycloneV object can be created from it. That object stores the state of the FPGA configuration and allows to read and modify it.

All the types, enums, functions, methods, arrays etc described in the following paragraph are in the CycloneV class.

3.2 Packages

```
enum package_type_t;

struct CycloneV::package_info_t {
  int pin_count;
  char type;
  int width_in_pins;
  int height_in_pins;
  int width_in_mm;
  int height_in_mm;
};

const package_info_t package_infos[5+3+3];
```

The FPGAs are sold in 11 different packages, which are named by their type (Fineline BGA, Ultra Fineline BGA or Micro Fineline BGA) and their width in mm.

Enum	Type	Pins	Size in mm	Size in pins
PKG_F17	f	256	16x16	17x17
PKG_F23	f	484	22x22	23x23
PKG_F27	f	672	26x26	27x27
PKG_F31	f	896	30x30	31x31
PKG_F35	f	1152	34x34	35x35
PKG_U15	u	324	18x18	15x15
PKG_U19	u	484	22x22	19x19
PKG_U23	u	672	28x28	23x23
PKG_M11	m	301	21x21	11x11
PKG_M13	m	383	25x25	13x13
PKG_M15	m	484	28x28	15x15

3.3 Model information

```
enum die_type_t { E50F, GX25F, GT75F, GT150F, GT300F, SX50F, SX120F };

struct Model {
  const char *name;
  const variant_info &variant;
  package_type_t package;
  char temperature;
  char speed;
  char pcie, gxb, hmc;
  uint16_t io, gpio;
};

struct variant_info {
  const char *name;
  const die_info &die;
  uint16_t idcode;
  int alut, alm, memory, dsp, dpll, dll, hps;
};

struct die_info {
  const char *name;
  die_type_t type;
  uint8_t tile_sx, tile_sy;
  // ...
};

const Model models[];
CycloneV *get_model(std::string model_name);
```

A Model is built from a package, a variant and a temperature/speed grade. A variant selects a die and which hardware is active on it.

The Model fields are:

- name - the SKU, for instance 5CSEBA6U23I7
- variant - its associated variant_info
- package - the packaging used
- temperature - the temperature grade, 'A' for automotive (-45..125C), 'I' for industrial (-40..100C), 'C' for commercial (0..85C)
- speed - the speed grade, 6-8, smaller is faster
- pcie - number of PCIe interfaces (depends on both variant and number of available pins)
- gxb - ??? (same)
- hmc - number of Memory interfaces (same)
- io - number of i/os
- gpio - number of fpga-usable gpios

The Variant fields are:

- name - name of the variant, for instance se120b
- die - its associated die_info

- idcode - the IDCODE associated to this variant (not unique per variant at all)
- alut - number of LUTs
- alm - number of logic elements
- memory - bits of memory
- dsp - number of dsp blocks
- dpll - number of plls
- dll - number of delay-locked loops
- hps - number of arm cores

The Die usable fields are:

- name - name of the die, for instance sx120f
- type - the enum value for the die type
- tile_sx, tile_sy - size of the tile grid

The limits indicated in the variant structure may be lower than the theoretical die capabilities. We have no idea what happens if these limits are not respected.

To create a CycloneV object, the constructor requires a Model *. Either choose one from the models array, or, in the usual case of selection by sku, the CycloneV::get_model function looks it up and allocates one. The models array ends with a nullptr name pointer.

The get_model function implements the alias "ms" for the 5CSEBA6U23I7 used in the de10-nano, a.k.a MiSTer.

3.4 pos, rnode and pnode

```
using pos_t = uint16_t;          // Tile position

static constexpr uint32_t pos2x(pos_t xy);
static constexpr uint32_t pos2y(pos_t xy);
static constexpr pos_t xy2pos(uint32_t x, uint32_t y);
```

The type pos_t represents a position in the grid. xy2pos allows to create one, pos2x and pos2y extracts the coordinates.

```
using rnode_t = uint32_t;        // Route node id

enum rnode_type_t;
const char *const rnode_type_names[];
rnode_type_t rnode_type_lookup(const std::string &n) const;

constexpr rnode_t rnode(rnode_type_t type, pos_t pos, uint32_t z);
constexpr rnode_t rnode(rnode_type_t type, uint32_t x, uint32_t y, uint32_t z);
constexpr rnode_type_t rn2t(rnode_t rn);
constexpr pos_t rn2p(rnode_t rn);
constexpr uint32_t rn2x(rnode_t rn);
constexpr uint32_t rn2y(rnode_t rn);
constexpr uint32_t rn2z(rnode_t rn);

std::string rn2s(rnode_t rn);
```

A rnode_t represents a note in the routing network. It is characterized by its type (rnode_type_t) and its coordinates (x, y for the tile, z for the instance number in the tile). Those functions allow to create one and extract the different

components. rnode_types_names gives the string representation for every rnode_type_t value, and rnode_type_lookup finds the rnode_type_t for a given name. rn2s provides a string representation of the rnode (TYPE.xxx.yyy.zzzz).

The rnode_type_t value 0 is NONE, and a rnode_t of 0 is guaranteed invalid.

```
using pnode_t = uint64_t;        // Port node id

enum block_type_t;
const char *const block_type_names[];
block_type_t block_type_lookup(const std::string &n) const;

enum port_type_t;
const char *const port_type_names[];
port_type_t  port_type_lookup (const std::string &n) const;

constexpr pnode_t pnode(block_type_t bt, pos_t pos, port_type_t pt, int8_t bindex,␣
↪int16_t pindex);
constexpr pnode_t pnode(block_type_t bt, uint32_t x, uint32_t y, port_type_t pt, int8_
↪t bindex, int16_t pindex);
constexpr block_type_t pn2bt(pnode_t pn);
constexpr port_type_t  pn2pt(pnode_t pn);
constexpr pos_t        pn2p (pnode_t pn);
constexpr uint32_t     pn2x (pnode_t pn);
constexpr uint32_t     pn2y (pnode_t pn);
constexpr int8_t       pn2bi(pnode_t pn);
constexpr int16_t      pn2pi(pnode_t pn);

std::string pn2s(pnode_t pn);
```

A pnode_t represents a port of a logical block. It is characterized by the block type (block_type_t), the block tile position, the block number instance (when appropriate, -1 when not), the port type (port_type_t) and the bit number in the port (when appropriate, -1 when not). pn2s provides the string representation BLOCK.xxx.yyy(.instance):PORT(.bit)

The block_type_t value 0 is BNONE, the port_type_t value 0 is PNONE, and pnode_t 0 is guaranteed invalid.

```
rnode_t pnode_to_rnode(pnode_t pn) const;
pnode_t rnode_to_pnode(rnode_t rn) const;
```

These two methods allow to find the connections between the logic block ports and the routing nodes. It is always 1:1 when there is one.

3.5 Routing network management

```
void rnode_link(rnode_t n1, rnode_t n2);
void rnode_link(pnode_t p1, rnode_t n2);
void rnode_link(rnode_t n1, pnode_t p2);
void rnode_link(pnode_t p1, pnode_t p2);
void rnode_unlink(rnode_t n2);
void rnode_unlink(pnode_t p2);
```

The method rnode_link links two nodes together with n1 as source and n2 as destination, automatically converting from pnode_t to rnode_t when needed. rnode_unlink disconnects anything connected to the destination n2.

There are two special cases. DCMUX is a 2:1 mux which selects between a data and a clock signal and has no disconnected state. Unlinking it puts in in the default clock position. Most SCLK muxes use a 5-bit vertical configuration where up to 5 inputs can be connected and the all-off configuration is not allowed. Usually at least one input goes to vcc, but in some cases all five are used and unlinking selects the 4th input (the default in that case).

```
std::vector<std::pair<rnode_t, rnode_t>> route_all_active_links() const;
std::vector<std::pair<rnode_t, rnode_t>> route_frontier_links() const;
```

route_all_active_links gives all current active connections. route_frontier_links solves these connections to keep only the extremities, giving the inter-logic-block connections directly.

3.6 Logic block management

```
const std::vector<pos_t> &lab_get_pos()    const
[etc]
```

The numerous xxx_get_pos() methods gives the list of positions of logic blocks of a given type. The known types are lab, mlab, m10k, dsp, hps, gpio, dqs16, fpll, cmuxc, cmuxv, cmuxh, dll, hssi, cbuf, lvl, ctrl, pma3, serpar, term and hip. A vector is empty when a block type doesn't exist in the given die.

In the hps case the 37 blocks can be indexed by hps_index_t enum.

```
enum { MT_MUX, MT_NUM, MT_BOOL, MT_RAM };

enum bmux_type_t;
const char *const bmux_type_names[];
bmux_type_t bmux_type_lookup(const std::string &n) const;

struct bmux_setting_t {
  block_type_t btype;
  pos_t pos;
  bmux_type_t mux;
  int midx;
  int type;
  bool def;
  uint32_t s; // bmux_type_t, or number, or bool value, or count of bits for ram
  std::vector<uint8_t> r;
};

int bmux_type(block_type_t btype, pos_t pos, bmux_type_t mux, int midx) const;
bool bmux_get(block_type_t btype, pos_t pos, bmux_type_t mux, int midx, bmux_setting_
↪t &s) const;
bool bmux_set(const bmux_setting_t &s);
bool bmux_m_set(block_type_t btype, pos_t pos, bmux_type_t mux, int midx, bmux_type_t␣
↪s);
bool bmux_n_set(block_type_t btype, pos_t pos, bmux_type_t mux, int midx, uint32_t s);
bool bmux_b_set(block_type_t btype, pos_t pos, bmux_type_t mux, int midx, bool s);
bool bmux_r_set(block_type_t btype, pos_t pos, bmux_type_t mux, int midx, uint64_t s);
bool bmux_r_set(block_type_t btype, pos_t pos, bmux_type_t mux, int midx, const␣
↪std::vector<uint8_t> &s);

std::vector<bmux_setting_t> bmux_get() const;
```

These methods allow to manage the logic blocks muxes configurations. A mux is characterized by its block (type and position), its type (bmux_type_t) and its instance number (0 if there is only one). There are four kinds of muxes, symbolic (MT_MUX), numeric (MT_NUM), booolean (MT_BOOL) and ram (MT_RAM).

bmux_type looks up a mux and returns its MT_* type, or -1 if it doesn't exist. bmux_get reads the state of a mux and returns it in s and true when found, false otherwise. The def field indicates whether the value is the default. The bmux_set sets a mux generically, and the bmux_*_set sets it per-type.

The no-parameter bmux_get version returns the state of all muxes of the FPGA.

3.7 Inverters management

```
struct inv_setting_t {
  rnode_t node;
  bool value;
  bool def;
};

std::vector<inv_setting_t> inv_get() const;
bool inv_set(rnode_t node, bool value);
```

inv_get() returns the state of the programmable inverters, and inv_set sets the state of one. The field def is currently very incorrect.

3.8 Pin/package management

```
enum pin_flags_t : uint32_t {
  PIN_IO_MASK    = 0x00000007,
  PIN_DPP        = 0x00000001, // Dedicated Programming Pin
  PIN_HSSI       = 0x00000002, // High Speed Serial Interface input
  PIN_JTAG       = 0x00000003, // JTAG
  PIN_GPIO       = 0x00000004, // General-Purpose I/O

  PIN_HPS        = 0x00000008, // Hardware Processor System

  PIN_DIFF_MASK  = 0x00000070,
  PIN_DM         = 0x00000010,
  PIN_DQS        = 0x00000020,
  PIN_DQS_DIS    = 0x00000030,
  PIN_DQSB       = 0x00000040,
  PIN_DQSB_DIS   = 0x00000050,

  PIN_TYPE_MASK  = 0x00000f00,
  PIN_DO_NOT_USE = 0x00000100,
  PIN_GXP_RREF   = 0x00000200,
  PIN_NC         = 0x00000300,
  PIN_VCC        = 0x00000400,
  PIN_VCCL_SENSE = 0x00000500,
  PIN_VCCN       = 0x00000600,
  PIN_VCCPD      = 0x00000700,
  PIN_VREF       = 0x00000800,
  PIN_VSS        = 0x00000900,
  PIN_VSS_SENSE  = 0x00000a00,
};

struct pin_info_t {
  uint8_t x;
  uint8_t y;
  uint16_t pad;
  uint32_t flags;
  const char *name;
```

(continues on next page)

(continued from previous page)

```
  const char *function;
  const char *io_block;
  double r, c, l, length;
  int delay_ps;
  int index;
};

const pin_info_t *pin_find_pos(pos_t pos, int index) const;
```

The pin_info_t structure describes a pin with:

- x, y - its coordinates in the package grid (not the fpga grid, the pins one)
- pad - either 0xffff (no associated gpio) or (index << 14) | tile_pos, where index indicates which pad of the gpio is connected to the pin
- flags - flags describing the pin function
- name - pin name, like A1
- function - pin function as text, like "GND"
- io_block - name of the I/O block for power purposes, like 9A
- r, c, l - electrical characteristics of the pin-pad connection wire
- length - length of the wire
- delay_ps - usual signal transmission delay is ps
- index - pin sub-index for hssi_input, hssi_output, dedicated programming pins and jtag

The pin_find_pos method looks up a pin from a gpio tile/index combination.

3.9 Options

```
struct opt_setting_t {
  bmux_type_t mux;
  bool def;
  int type;
  uint32_t s; // bmux_type_t, or number, or bool value, or count of bits for ram
  std::vector<uint8_t> r;
};

int opt_type(bmux_type_t mux) const;
bool opt_get(bmux_type_t mux, opt_setting_t &s) const;
bool opt_set(const opt_setting_t &s);
bool opt_m_set(bmux_type_t mux, bmux_type_t s);
bool opt_n_set(bmux_type_t mux, uint32_t s);
bool opt_b_set(bmux_type_t mux, bool s);
bool opt_r_set(bmux_type_t mux, uint64_t s);
bool opt_r_set(bmux_type_t mux, const std::vector<uint8_t> &s);

std::vector<opt_setting_t> opt_get() const;
```

The options work like the block muxes without a block, tile or instance number. They're otherwise the same.

3.10 Bitstream management

```
void clear();
void rbf_load(const void *data, uint32_t size);
void rbf_save(std::vector<uint8_t> &data);
```

The clear method returns the FPGA state to all defaults. rbf_load parses a raw bitstream file from memory and loads the state from it. rbf_save generats a rbf from the current state.

CHAPTER

FOUR

THE MISTRAL-CV COMMAND-LINE PROGRAM

The mistral-cv command line program allows for a minimal interfacing with the library. Calling it without parameters shows the possible usages.

4.1 models

```
mistral-cv models
```

Lists the known models with their SKU, IDCODE, die, variant, package, number of pins, temperature grade and speed grade.

4.2 routes

```
mistral-cv routes <model> <file.rbf>
```

Dumps the active routes in a rbf.

4.3 routes2

```
mistral-cv routes <model> <file.rbf>
```

Dumps the active routes in a rbf where a GIN/GOUT/etc does not have a port mapping associated.

4.4 cycle

```
mistral-cv cycle <model> <file.rbf> <file2.rbf>
```

Loads the rbf in file1.rbf and saves is back in file2.rbf. Useful to test if the framing/unframing of oram/pram/cram works correctly.

4.5 bels

```
mistral-cv bels <model>
```

Dumps a list of all the logic elements of a model (only depends on the die in practice).

4.6 decomp

```
mistral-cv decomp <model> <file.rbf> <file.bt>
```

Decompiles a bitstream into a compilable source. Only writes down what is identified as not being in default state.

4.7 comp

```
mistral-cv comp <file.bt> <file.rbf>
```

Compiles a source into a bitstream. The source includes the model information.

4.8 diff

```
mistral-cv diff <model> <file1.rbf> <file2.rbf>
```

Compares two rbf files and identifies the differences in terms of oram, pram and cram. Useful to list mismatches after a decomp/comp cycle.

CHAPTER

FIVE

MISTRAL CYCLONEV LIBRARY INTERNALS

5.1 Structure

A large part of the library is generated code from information in the data directory. The exception is the routing data that is converter to compressed binary and put in the gdata directory. All the conversions are done with python programs and shell scripts in the tools directory.

5.2 Routing data

The routing data is stored in bzip2-compressed text files named <die>-r.txt.bz2. Each line describes a routing mux.

A mux description looks like that:

```
H14.000.032.0003 4:0024_2832 0:GIN.000.032.0005 1:GIN.000.032.0004 2:GIN.000.032.0001␣
↪3:GIN.000.032.0000
```

That line describes the mux for the rnode H14.000.032.0003. It uses the pattern 4 as position (24, 2832) and has four inputs connected to four GIN rnodes.

The chip uses a limited number of mux types, with a specific bit pattern in the cram controlling a fixed number of inputs and of bit set/unset values selecting them. There is a total of 70 different patterns, currently only described as C++ code in cv-rpats.cc. An additional 4 are added to store the variations of pattern 6 where the default is different.

The special case of pattern 6 looks like:

```
SCLK.014.000.0025 6.3:1413_0638 0:GCLK.000.008.0009 1:RCLK.000.004.0011 4:RCLK.000.
↪004.0003
```

The ".3" indicates that the default is on slot 3, e.g. value 0x08 or pattern 70+3.

The python script routes-to-bin.py loads this file and generated a compressed binary version in gdata which matches the rmux structure. The script mkroutes.sh generates it for all die types.

5.3 Block muxes

The lists of block muxes and options muxes are independant of the dies. They're in the block-mux.txt files. Each mux is described in these files using the following syntax:

```
g dft_mode m:3 21.42 20.40 20.43
  0 off
  1 on !
  7 dft_pprog
```

"g" indicates the subtype of mux, which is block-dependant, here "global". 'm' indicates a symbolic mux, 3 is the number of bits. It is followed by the bits coordinates, LSB first. Here it's an inner block, so the coordinates are 2D. Options are also 2D, and peripheral blocks are 1D.

In such a case of symbolic mux it is followed by the indented possible values of the mux (in hex) with the exclamation point indicating the default.

A numeric mux is similar but the type is 'n' and labels on the right have to be numeric.

Boolean muxes look like this:

```
g clk0_inv               b-   6.45
```

The 'b' indicates boolean, and '-' indicates the default is false, otherwise it is '+' for true. The boolean can be multi-bits, such as in the following example. Then all bits are set or unset.

```
g pr_en              b-:2 0.61 0.67
```

Finally ram muxes look like:

```
g cvpcie_mode                    r-:2    2.21 2.22
g clkin_0_src                    r2:4  760 761 762 763
```

In the second case the '2' between r and : indicates that the default value is 2.

Instanciated muxes can take two forms. For instance in fpll muxes of subtype 'c' are instanciated on the counter number, hence have 9 values. The mux is written as:

```
c cnt_in_src                     r2:2  600 601 | 602 603 | 604 605 | 606 607 | 608␣
↪609 | 610 611 | 612 613 | 614 615 | 616 617
c dprio0_cnt_hi_div              r1:8
  * 8 9 10 11 12 13 14 15
  * 24 25 26 27 28 29 30 31
  * 40 41 42 43 44 45 46 47
  * 56 57 58 59 60 61 62 63
  * 72 73 74 75 76 77 78 79
  * 88 89 90 91 92 93 94 95
  * 104 105 106 107 108 109 110 111
  * 120 121 122 123 124 125 126 127
  * 136 137 138 139 140 141 142 143
```

Either the bits are indicated on the same line separated by 'l', or they're set as one set per line start with an indented '*'.

The lab, mlab, m10k, mlab and hps_clocks target bits in the 2D cram by offsetting from a base position computed from the tile position (see the method pos2bit). opt targets bits in the oram. All the others with the exception of pma3-c target bits in the pram from a position found in <die>-pram.txt. pma3-c targets bits in the cram from the tables in pma3-cram.txt

mux_to_source.py enum <datadir> generates the file cv-bmuxtypes.ipp while mux_to_source.py mux <datadir> generates the file cv-bmux-data.cc. mkmux.sh does both calls.

5.4 Logic blocks

Blocks come from two sources, the files <die>-pram.txt indicates all the peripheral blocks with their pram address. The files <die>-<block>.txt where bock is cmux, ctrl, fpll, hmc, hps or iob has the information of the connections between the blocks and neighbouring blocks and the routing grid.

blocks_to_source.py generates the cvd-<die>-blk.cc file for a given die, abd mkblocks.sh calls it for every die.

5.5 Inverters

The list of inverters, their cram position and their default value (always 0 at this point) is in <die>-inv.txt. inv_to_source.py/mkinv.sh takes care of generating the cvd-<die>-inv.cc files.

5.6 Forced-1 bits

Five of the seven dies seem to have bits always set to 1. They are listed in the files <die>-1.txt. blocks_to_source.py takes care of it.

5.7 Packages

The file <die>-pkg.txt lists the packages and the pins of each package for each die. pkg_to_source.py/mkpkg.sh take cares of generating the cvd-<die>-pkg.cc files.

5.8 Models

models.txt includes all the information on variants and models. The cv-models.cc file is generated by models_to_source.py called by mkmodels.sh.