Chapter 7
Semi-custom EDA

Abstract In modern computing systems, FPGAs are used as dedicated programmable accelerators (Che et al. [1], Zhang et al. [2], Cong et al. [3]). General-purpose FPGAs are well optimized to fit a wide range of applications with a reasonable trade-off on performance, power, and area, but are seriously sub-optimal in application-specific contexts (Cong et al. [3], Neshatpour et al. [4]). In such case, customized FPGA architectures, which are highly tailored for a specific set of applications as well as seamless integration to other computing resources in the system, become a proper solution. However, developing a FPGA layout through full custom approaches is a time-consuming process even for industrial vendors, whose may take years to finalize (Greenhill et al. [5]). In addition, design tools such as mapping algorithms and bitstream generation have to be customized for different FPGA architectures, which lead to another time-consuming development task. Driven by the strong need, fast prototyping technology for customize FPGAs, especially semi-custom design approaches, has been insensitively researched in recent years. As such, development cycles of custom FPGAs can be comparable to modern ASICs, which opens the door to tightly integrating FPGAs to SoCs. In this section, we will first review existing EDA tools and then focus on critical EDA techniques that enable semi-custom designed FPGAs.

7.1 Overview

In the past two decades, fast prototyping techniques for customized FPGA architectures have been proven by many researches through semi-custom design flows [6–14]. These works share the same principles when generating FPGA layouts:

K. Tu et al., *FPGA EDA*, https://doi.org/10.1007/978-981-99-7755-0_7

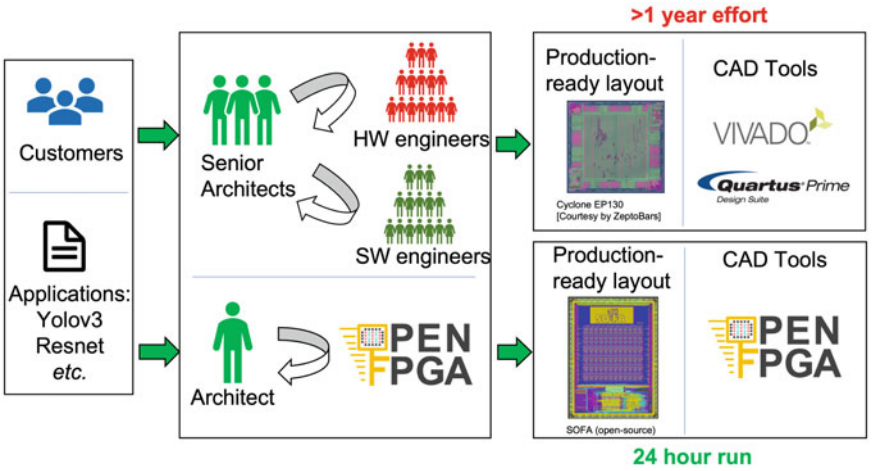


Fig. 7.1 An illustrative example that compares on engineering time and effort to prototype an FPGA using OpenFPGA (an open-source EDA tool that enables semi-custom approaches) and full-custom approaches

1. Model an FPGA architecture in synthesizable HDL netlists.
2. Use sophisticated ASIC design tools to implement the HDL netlists into physical layouts.

As illustrated in Fig. 7.1, the fast prototyping technology through semi-custom design flows accelerates and automates the development process of FPGAs.

Early works rely on handcrafted HDL netlists for FPGA architectures which even include low-level details down to transistor-level circuit designs [6, 7]. However, such methodology requires still significant manual effort, being inefficient in designing diverse FPGA fabrics targeting domain-specific applications. Moreover, early works focus only on developing fabric generators without associated compiler support, e.g., HDL-to-Bitstream generation [6, 7]. Recent works aim to build "FPGA generators" in the similar concept as the memory compilers in ASIC world [8–14]. The FPGA generators integrate both netlist generators and bitstream generators in a unified framework, on top of the well-known FPGA architecture exploration tool, e.g., VTR [15, 16]. Major technical features of existing FPGA generators are summarized in Table 7.1.

However, to implement production quality FPGA fabrics, layout generation is only a small part (Fig. 7.2), when compared to other essential aspects, such as testbench generator and bitstream support. For example, to verify the correctness of FPGA fabrics before taping out, design verification is a mandatory step. Note that design verification for FPGAs is mainly a software problem rather than a hardware problem, as functionality of an FPGA is determined by a bitstream file. Therefore, to ensure a high coverage in verification, a number of bitstream files are required to verify different operating modes and utilization rates of an FPGA device. As a result, a

Table 7.1 Comparison on EDA tools enabling semi-custom FPGA design

Tool/metric	Open source	Architecture language	Netlist generation	Bitstream generation	Testbench generation	SDC generation
Kuon et al. [6]	×	✓	Automatic[a]	×	×	×
Ova et al. [7]	×	×	Hand-crafted	×	×	×
Archipelago [10]	✓	×	Automatic	✓	×	×
Anderson et al. [8, 9]	×	✓	Automatic	✓	✓	✓
Mohan et al. [13]	×	✓	Automatic	✓	✓	✓
PRGA [11]	✓	✓	Automatic	✓	×	×
FABulous [12]	✓	✓	Automatic[b]	✓	✓	×
OpenFPGA [14]	✓	✓	Automatic	✓	✓	✓

[a]Only netlists of a tile is automatically generated
[b]Netlists of primitive circuits, e.g., LUT and routing multiplexers, have to be hand crafted

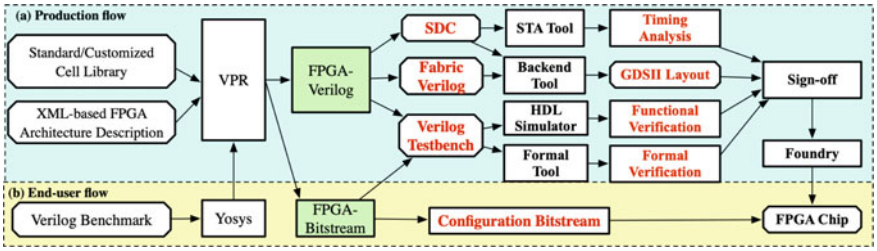


Fig. 7.2 Semi-custom design flow for FPGA fabrics: **a** production flow and **b** end-user flow

functional HDL-to-Bitstream generator is a required component, being as important as a netlist generator. In addition, a testbench generator is required to simulate the bitstream downloading w.r.t. a configuration circuits, as well as check the functional correctness of an FPGA under different I/O mapping and bitstreams. Actually, the complexity of a HDL-to-Bitstream flow is significantly higher than a netlist generator, which covers many NP hard problems in EDA, such as placement and routing. In recent years, with the growth of open-source HDL-to-Bitstream tools, design verification has been seriously considered and included in recent EDA tools, as shown in Table 7.1. In short, design verification for FPGA should not only validate the correctness of layout but also the correctness of associated software tool chains.

Beyond the essential components, to enable high-quality FPGA fabrics, timing constraints for physical design are critical. Nowadays, timing constraints are typically in the *Synopsys Design Constraints*(SDC) format, which are used to constrain timing paths when ASIC tools generate FPGA layouts. Without timing constraints, pin-to-pin delays, such as LUT delays and routing delay, may be too large to satisfy the target performance of an FPGA. Note that, a key difference between FPGAs and ASICs on timing paths is that an FPGA only has critical paths when mapped to a

specific HDL design. When implementing FPGA layouts, timing constraints cannot be biased to an HDL design because it may probably cause performance degradation on another HDL design. Therefore, the principle of the timing constraints is keep pin-to-pin delays on each timing path as uniform as possible, which indicates that every timing path is critical. Considering the large number of timing paths in a FPGA fabric, a SDC generator is required to avoid huge manual effort. Nowadays, to achieve high-performance FPGA fabrics, SDC generators are available in semi-custom EDA tool chains (Table 7.1).

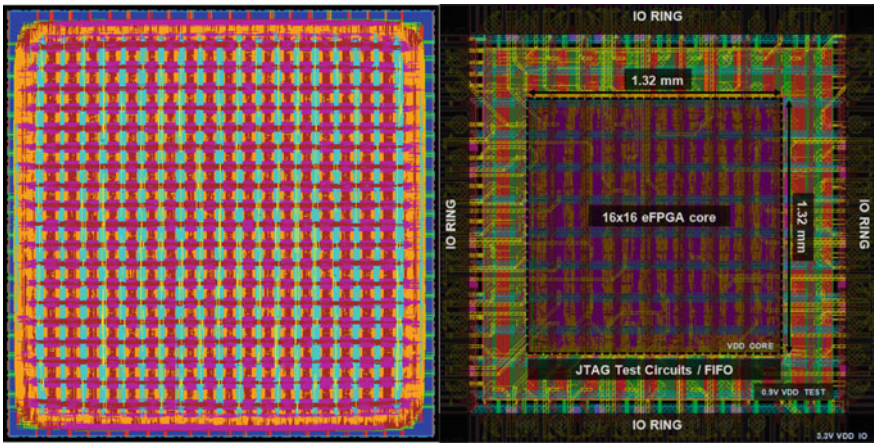
As architecture of FPGAs can be really different depending on their application context, a key value of FPGA generators is to support versatile FPGA architectures. Therefore, FPGA architecture description languages are needed to model complicated and large-scale FPGA device in compact and human readable representations. By leveraging the *University of Toronto FPGA Architecture Language*(UTFAL) [17], FPGA generators can convert a high-level FPGA description to synthesizable HDL netlists, and then implement layouts through ASIC design tools. Thanks to UTFAL's enriched syntax, FPGA generators can support a wide range of FPGA architectures. To unlock more possibility in device modeling, extended architecture description language (set of architecture guidance models) has been proposed [18], In this chapter, we focus on introducing the extended architecture description language while UTFAL has been covered in Chap. 2.

In short, a netlist generator, a bitstream generator and a testbench generator are three indispensable components in a basic semi-custom EDA framework for FPGA, with which designers can accomplish a functional FPGA fabric. However, as the growing needs of domain-specific FPGA fabrics, an expressive architecture language is now becoming important, because it is a must-have for designers to rapidly evaluate and prototype innovative FPGA architectures. As researchers have proven the feasibility of FPGA generators with silicon results (Fig. 7.3), future trends lie on improving PPA of the FPGA fabrics. This drives SDC generators to be an strategically important tool, which can constrain PPA of each segment in an FPGA fabric through semi-custom design tools w.r.t. performance goals.

7.2 Extended Architecture Description Language

In this part, we focus on the extended architecture description language(set of architecture guidance models conceptualized in Chap. 2) adopted by the OpenFPGA framework [18]. Other architecture description languages may have different syntax when modeling FPGA fabrics but share similar principles [11, 12]. Therefore, we focus more on general principles when designing an FPGA architecture description language than detailed syntax, with which we believe it is easier for readers to understand other architecture description languages.

UTFAL is designed for a detailed logical representation of FPGA architectures, providing sufficient information for EDA engines to perform packing, placement, and routing. However, to enable netlist generation and bitstream generation, a detailed



(a) A 20x20 FPGA fabric (courtesy by [8]) (b) A 16x16 FPGA chip (courtesy by [13])

Fig. 7.3 Showcase FPGA layouts through semi-custom design approaches

physical representation of complete FPGA fabric is required. The extended architecture description language is designed to provide supplementary information on top of the UTFAL. It fills the blank of UTFAL when modeling circuit-level implementation of programmable resources (see Sect. 7.2.1), physical mode of programmable blocks (see Sect. 7.2.2), and configuration scheme (see Sect. 7.2.2). Therefore, the extended architecture description language is complementary to UTFAL without overlapping in syntax and information. Similar to UTFAL, the extended architecture description language is XML-based. Full documentation about UTFAL and the extended architecture description language is available on [19, 20], respectively.

7.2.1 Circuit Modeling

As circuit design is a dominant factor impacting FPGA's PPA, the extended architecture description language provides enriched syntax to model circuit-level details of primitives in FPGAs, e.g., LUT, routing multiplexers. Figure 7.2 illustrates the different focus on modeling LUTs and routing multiplexers between UTFAL and the extended architecture description language. For EDA usage only, primitives can be treated as a black box with limited information, e.g., number of ports, port direction as well as pin-to-pin delays. However, to generate netlists, detailed circuit designs of primitives have to be modeled. On the other side, upon practical applications, hardware engineers may select various circuits to implement their FPGA fabrics. For instance, a ultra-low-power FPGA may be built with ultra-low-power circuit cells while a high-performance FPGA may use absolutely different circuit cells. As a result, the extended architecture description language is capable of modeling highly

```
<!-- Fracturable LUT modeled in XML -->
<circuit_model type="lut" name="frac_lut4" verilog_netlist="lut.v">
  <design_technology fracturable_lut="true"/>
  <pass_gate_logic circuit_model_name="tgate"/>
  <port type="input" prefix="lut_i" size="4" tri_state_map="---1"/>
  <port type="output" prefix="lut3_o" size="2" lut_frac_level="3"
lut_output_mask="0,1"/>
  <port type="output" prefix="lut4_o" size="1"
lut_output_mask="0"/>
  <port type="sram" prefix="sram" size="16"/>
  <port type="sram" prefix="mode" size="1" mode_select="true"/>
</circuit_model>
```

Fig. 7.4 Examples of extended XML syntax for LUTs

flexible circuit design topology even down to transistor level and allows designers to customize any component in an FPGA.

Among the programmable resources in an FPGA, there are two types of circuits whose structures have prominently impact on PPA and bitstream generator: LUTs and routing multiplexers. LUTs are used to implement logic functions while routing multiplexers are used to route signals between LUTs. In some FPGA devices, LUTs and routing multiplexers take 90% of chip area, critical path delays, and power consumption [21]. The choice of the circuit implementation may also impact the PPA of standalone circuit by 2× [22]. Therefore, the extended architecture description language provides fruitful syntax to support diverse circuit design topology and details for LUTs and routing multiplexers.

Table 7.3 lists the mainstream circuit topology for LUTs and routing multiplexers that are frequently used by modern FPGAs. Figure 7.4 shows an example about how the extended architecture description language models the internal structure of a fracturable 4-input LUT. Users can specify which inputs are disabled during fracturable mode in the XML property `tri_state_map`. The levels and positions of fracturable outputs can be freely defined through the XML properties `lut_frac_level` and `lut_output_mask`. To support mode switching of fracturable LUTs, the port map includes a special port `mode` rather than the regular configuration port. Figure 7.5 shows another example about how a tree-like 4-input routing multiplexer (see Table 7.2 for schematic) is modeled by the extended architecture description language. The multiplexing structures can be customized through an XML property `structure`. Note that both input, output and even intermediate buffers can be customized through XML syntax, which are needed for LUTs and routing multiplexers in different location of an FPGA. With these modeling, a netlist generator can output RTL and even gate-level netlists for the LUTs and routing multiplexers, meanwhile bitstream generator can decode configuration bits.

In addition to the detailed modeling, black-box modeling is also supported, where users can provide their own circuit implementation for primitives. When black-box modeling is adopt, the path to netlist should be defined through the XML property `verilog_netlist`, and only necessary information such as port list is required.

```
<!-- Tree-like MUX modeled in XML -->
<circuit_model type="mux" name="mux_tree"/>
  <design_technology structure="tree"/>
  <pass_gate_logic circuit_model_name="STD_CELL_MUX2"/>
  <port type="input" prefix="mux_in" size="4"/>
  <port type="output" prefix="mux_out" size="1"/>
  <port type="sram" prefix="mux_sram" size="2"/>
</circuit_model>
```

Fig. 7.5 Examples of extended XML syntax for MUXes

Such modeling is also frequently used as modern FPGAs are built with various third-party IPs, e.g., *Digital Signal Processor* (DSP), *Random Access Memory* (RAM) and *Serializer/Deserializer* (SerDes).

7.2.2 Physical Mode Modeling

To simplify EDA algorithms, UTFAL focus on compact description of *Logic Element* (LE) architectures instead of a complete schematic-level representation. For instance, a complex multi-mode LE in Fig. 7.6a is modeled by multiple abstract-level operating modes in Fig. 7.6b, c. The abstraction indeed eases the EDA algorithms in mapping to FPGA resources but hides important details required by netlist and bitstream generation for the physical LEs. For example, netlist generators cannot identify which mode in Fig. 7.6 denotes the physical implementation of the LE. Bitstream generators may miss configuration bits to be decoded in physical mode when the operating modes in Fig. 7.6b, c only include a part of programmable routing resources. Moreover, configuration bits of an operating mode should be properly reorganized for the physical mode. For example, the configuration bits of the two `3-LUT` in Fig. 7.6c should be mapped to the fracturable `4-LUT` in Fig. 7.6a. Without a detailed circuit-level implementation of the fracturable `4-LUT`, bitstream generators cannot even decode configuration bits of the two `3-LUT` from logic synthesis results.

Therefore, to enable both netlist and bitstream generators, extended syntax is developed to

1. distinguish between physical mode and operating modes;
2. link the components in the various operating modes to physical mode
3. establish the relationship between primitives in physical mode and their circuit-level modeling (see Sect. 7.2.1).

To be intuitive, we take the example of the multi-mode CLB shown in Fig. 7.6 and present XML description in Fig. 7.7. The physical implementation of the LE is specified to be the mode `phy`, through syntax `physical_mode_name`. The detailed architecture of the physical LE follows the same style as the UTFAL. Under the physical mode, users can link primitive blocks to circuit implementations using a XML property `circuit_model_name`. Figure 7.7 shows how a

Table 7.2 Different objectives between UTFAL and extended architecture description language: logical vs. physical modeling

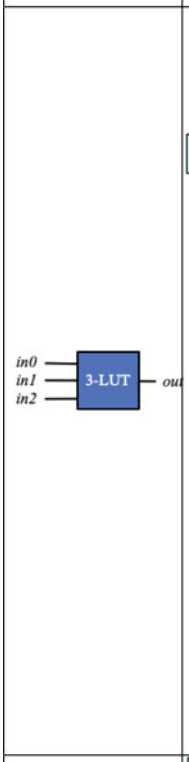
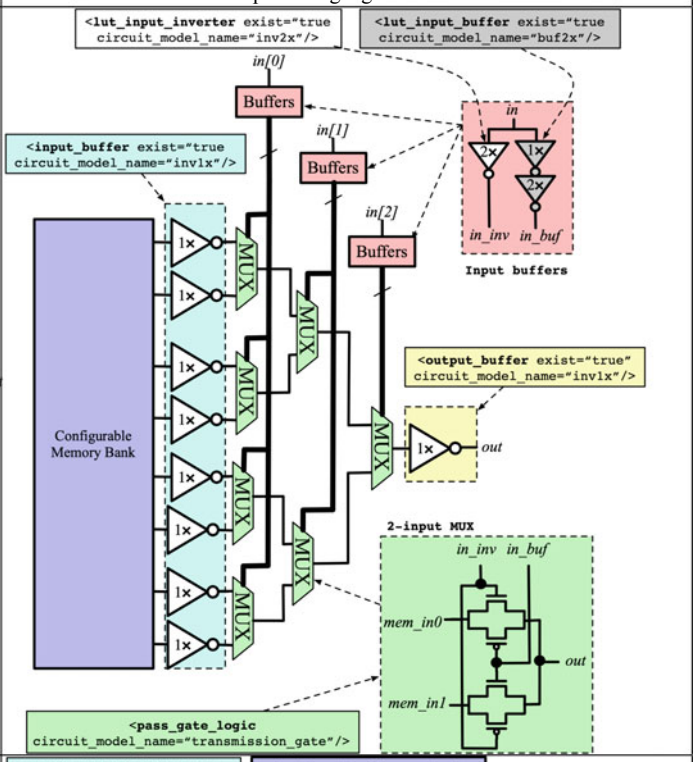
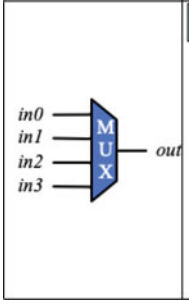
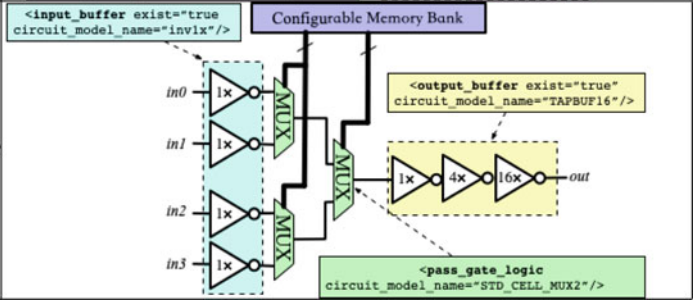
UTFAL	Extended architecture description language
	
	

Table 7.3 Various circuit designs of LUTs and routing multiplexers

Circuit	Design topology
LUT	1. Single-output LUTs
	2. Fracturable (multi-output) LUTs
	3. LUT with hard logic, e.g., carry
	4. LUT built with standard cells
	5. LUT with RAM/ROM
Routing multiplexer	1. One-level multiplexer
	2. Multi-level multiplexer
	3. Tree-like multiplexer
	4. Standard-cell multiplexer
	5. Multiplexer with local encoder
	6. Multiplexer with constant input

*Input and output buffering can be fully customized for both circuits

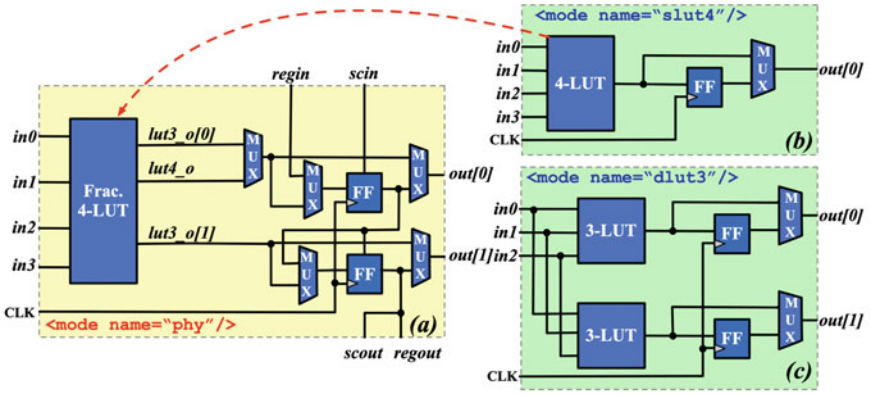


Fig. 7.6 **a** Physical implementation of a LE and **b**, **c** two operating modes

fracturable LUT `flut` is linked to a defined circuit model `frac_lut4` in Fig. 7.4. Under the operating modes, each virtual `pb_type` has to be linked to its physical implementation through XML properties `physical_pb_type_name` and `physical_mode_port`. Consider the example in Fig. 7.7, the operating modes `dlut3` and `slut4`, which correspond to the illustration in Fig. 7.6b, c, are linked to the physical mode `phy` which correspond to the illustration in Fig. 7.6a. The inputs `in` and outputs `out` of the pb_type `lut4` in mode `slut4` are linked to the inputs `in[0:3]` and outputs `lut4_o` of the pb_type `flut` in its physical mode `phy`, as highlighted by red dash lines in Fig. 7.6. XML syntax `mode_bits` allows users to customize the configuration bits applied to fracturable LUTs in any operating mode. For example, in Fig. 7.7, when the `lut4` is used, the `mode_bits="1"` will be applied to the port `mode` of its physical module `frac_lut4` in Fig. 7.4. As such, without modifying packing or synthesis engines, the XML syntax can map the con-

```
<!-- physical pb_type binding in logic element -->
<!-- Specify the physical mode -->
<pb_type name="le" physical_mode_name="phy"/>
<!-- Specify the circuit model for the primitives in physical
mode -->
<pb_type name="le[phy].flut" circuit_model_name="frac_lut4"/>
<pb_type name="le[phy].ff" circuit_model_name="sdff"/>
<!-- Bind operating modes to physical modes -->
<pb_type name="le[dlut3].lut3" physical_pb_type_name="le[phy].flut"
mode_bits="0">
  <port name="in" physical_mode_port="in[0:2]"/>
  <port name="out" physical_mode_port="lut3_o"/>
</pb_type>
<pb_type name="le[dlut3].ff" physical_pb_type_name="le[phy].ff"/>
<pb_type name="le[slut4].lut4" physical_pb_type_name="le[phy].flut"
mode_bits="1">
  <port name="in" physical_mode_port="in[0:3]"/>
  <port name="out" physical_mode_port="lut4_o"/>
</pb_type>
<pb_type name="le[slut4].ff" physical_pb_type_name="le[phy].ff"/>
```

Fig. 7.7 Examples of extended XML syntax for a LE

```
<configuration_protocol>
  <organization type="memory_bank" circuit_model_name="sram"
num_regions="4"/>
</configuration_protocol>
```

Fig. 7.8 Examples of memory-bank-based configuration protocol modeling

figuration bits from any operating mode to its physical implementation. In addition, such multi-mode modeling enable users to define a simplified BLE architecture in operating modes than physical mode, which reduces CPU time for packing.

7.2.3 Configuration Protocol

Programmable resources in an FPGA have to be configured through a protocol. However, configuration protocols are not modeled in UTFAL because they are well decoupled from packing, placement, and routing algorithms. Configuration scheme directly impacts bitstream generators, which is essential to a complete tool chain. More importantly, configuration protocol could be really different in FPGAs, depending on the application context. Extended architecture description language is developed to support versatile configuration protocols. Figure 7.8 shows an example of modeling a memory-bank-based configuration protocol, where other types of configuration protocol can be specified through XML property `type`. Through memory banks, each configuration memory cell can be accessed by enabling dedicated *Bit-Line* (BL) and *Word-Line* (WL). Note that the circuit implementation of a memory cell can be not

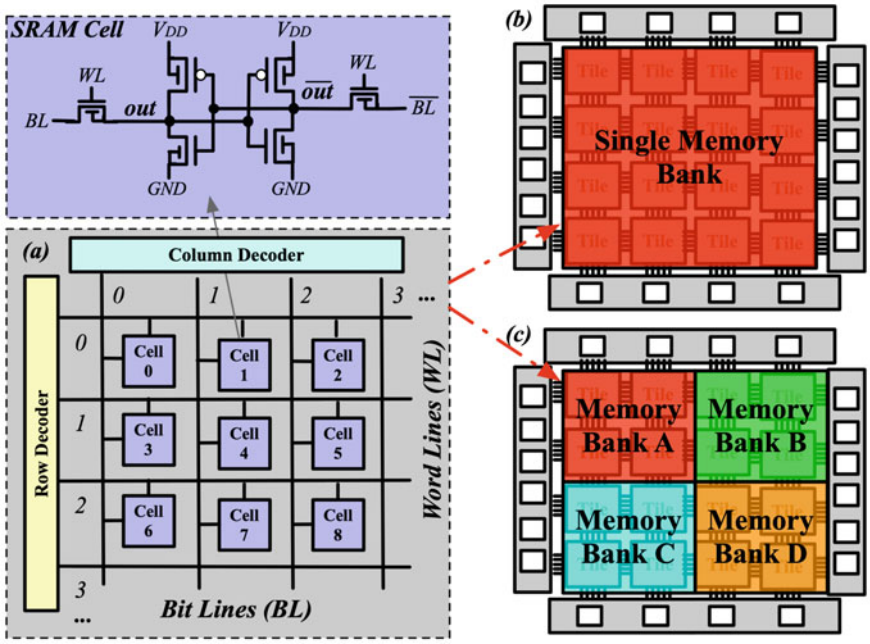


Fig. 7.9 Example of **a**, a memory organization using decoders; **b** single memory bank across the fabric; and **c** multiple memory banks across the fabric

limited to a SRAM, as shown in Fig. 7.9. For example, flip-flops or latches can also used as the fundamental cell in memory banks. The circuit model of configuration memory cell can be specified through XML property `circuit_model_name`. In addition, as FPGA size grows, multiple configuration regions are adapted to avoid long configuration time as well as challenges in physical design due to large parasitic in BL/WL interconnection. Figure 7.9b, c shows illustrative examples of single-region and 4-region memory banks, respectively. Therefore, the number of configuration regions can be customized through the XML property `num_regions`. Note that other configuration protocols, such as configuration chains and frame-based, are parameterized as memory banks, where different number of regions and various circuit implementation may also be applied.

In practice, configuration scheme for each tile or lower level primitive may need full customization. Take the example of memory bank, chip designer may need to customize which tiles to share BLs and WLs, in order to optimize in physical design and configuration time. Figure 7.10 shows an example file where designers can specify BL and WL sharing for each tile in each configuration region of an FPGA fabric. Two tiles share the same BL when their column index are same. Two tiles share the same WL when their row index are same. Consider the example in Fig. 7.10, the two tiles `grid_io_bottom_1__0_` and `grid_io_bottom_2__0_` are configured by the same WL but through two different BLs, where the BLs and WLs

```
<fabric_key>
  <region id="0">
    <<key id="0" alias="grid_io_bottom_1__0_" column="0" row="0"/>
    <<key id="0" alias="grid_io_bottom_2__0_" column="1" row="0"/>
  </region>
  <region id="1">
    <<key id="0" alias="grid_io_right_3__1_" column="2" row="1"/>
    <<key id="0" alias="grid_io_right_3__2_" column="2" row="2"/>
  </region>
  <region id="2">
    <<key id="0" alias="grid_clb_1__1_" column="0" row="1"/>
    <<key id="0" alias="grid_clb_1__2_" column="0" row="2"/>
  </region>
  <region id="3">
    <<key id="0" alias="grid_clb_2__1_" column="1" row="1"/>
    <<key id="0" alias="grid_clb_2__2_" column="1" row="2"/>
  </region>
</fabric_key>
```

Fig. 7.10 Examples of fabric key file modeling BL/WL sharing

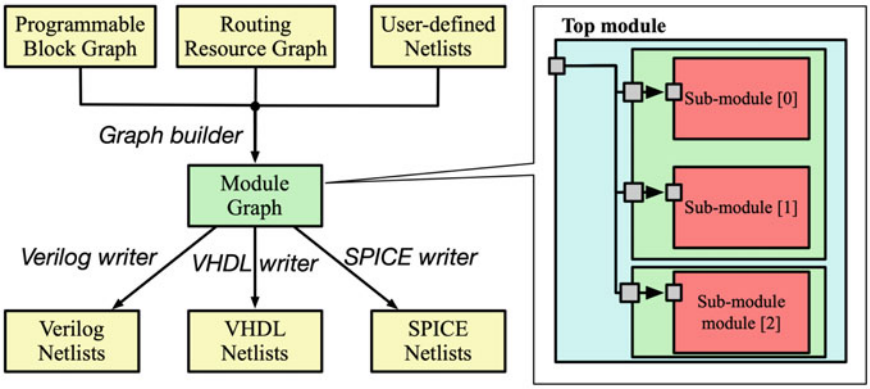


Fig. 7.11 Flowchart of netlist generator and graph-based modeling for modules

are controlled by `region 0`. For each region, different set of BLs and WLs are used to control the tiles under it. A tile can only be controlled by a configuration region. We refer interested reader to [20] for details.

7.3 Netlist Generator

As a cornerstone of the semi-custom design tools, netlist generators aim to translate a high-level architecture description to HDL netlists which can be adapted by ASIC tools to implement physical layouts. In early works, netlist generators is a simple

HDL code generator [6–10], which outputs internal device modeling to a synthesizable HDL format in a straightforward way. However, such native HDL translation of FPGA fabric imposes strong limitation when implementing physical layouts. For example, considering the HDL netlist which model a complete routing fabric as a flatten graph, the file sizes of netlists increase exponentially when FPGA size increases, which causes a long runtime in physical design. Furthermore, flatten netlists force a high design complexity when implementing an FPGA fabric, since a 4K-LUT FPGA may contain 8+ millions of logic gates. As a result, the physical design runtime of a medium sized FPGA is more than 24 h [8], while the physical design may fail for large sized FPGAs [23]. Modern netlist generators are designed to not only a simple code generator but also contain many features which make outputted netlists to be:

1. physical design friendly;
2. compatible with multiple HDL format and their standards;
3. human-readable, easy to debug and backtrace errors.

To enable these features, as depicted in Fig. 7.11, the implementation of the netlist generators is based on two steps:

1. Create a graph of modules which represent the complete FPGA fabric;
2. Build a number of netlist writers which output the module graph into selected file formats.

In the graph-based modeling, the whole FPGA fabric is represented as a tree of modules and their instances, as shown in Fig. 7.11. Modeling an FPGA fabric in a graph allows EDA tool to easily adjust hierarchy of netlists. For example, through graph merging, sub-modules can be merged which unlocks more opportunity in physical design optimization. It is also straightforward to profile the FPGA fabric, e.g., get the depth of netlists, count number of unique modules, etc., which can provide critical information for physical designers. A graph can be outputted to different file formats through various netlist writers, such as Verilog writer. As such, netlist writers consider a graph as an input, being decoupled from rest of engines in netlist generators. This can avoid massive code changes in core engine when developing a new netlist writer.

The auto-generated fabric netlists include both a programmable fabric with configuration protocol embedded. To be physical design friendly, netlist generators are capable of outputting netlist in different levels, e.g., *Register-Transfer Level* (RTL) and *Gate-level* (GL). Netlists at different levels of details unlock optimization opportunities through different design flows. As illustrated in Fig. 7.12, RTL (behavioral) netlists can be optimized through synthesis tools to standard cells and then physically implemented to layouts. Alternatively, GL (denoted as technology-mapped in Fig. 7.12) netlists are preferred as an direct input to physical design tool, when chip designers require specific standard cells to implement primitive circuits which are not synthesizable. The choice of design flows really depends on the PPA requirements and expertise of chip designers. For example, for ultra-high-performance FPGA, some specific cells are required in gate-level netlists and synthesis should be skipped.

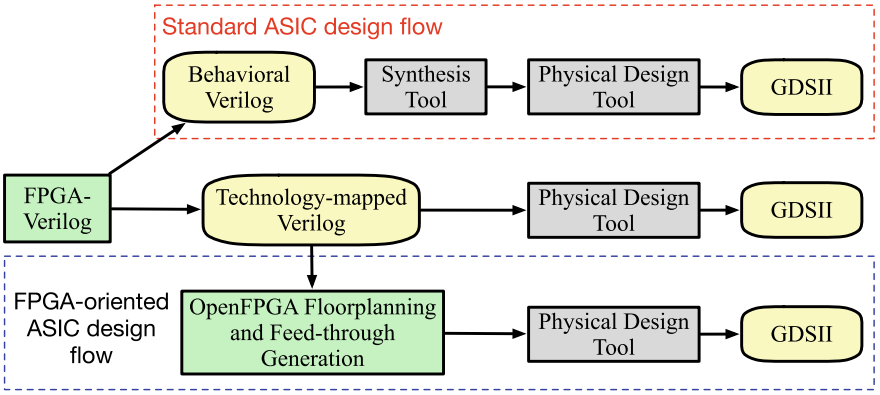


Fig. 7.12 An example of physical-design-friendly netlist generators

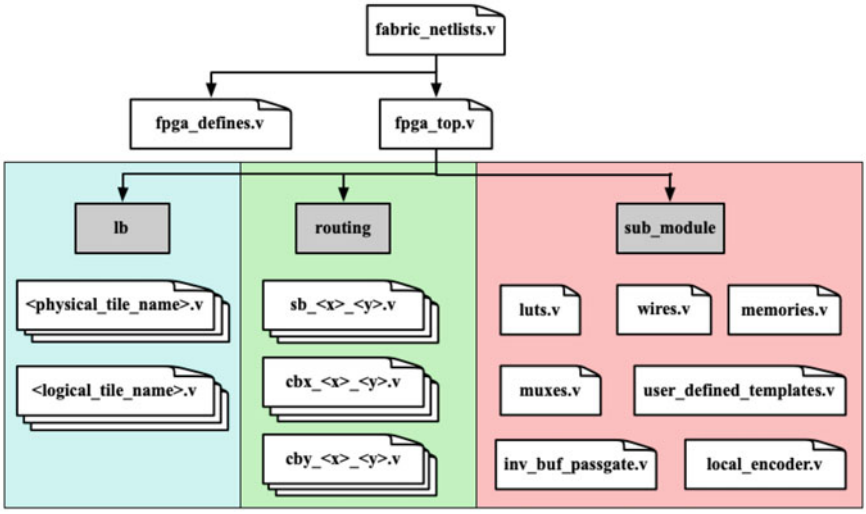


Fig. 7.13 An example of hierarchical Verilog netlists modeling a FPGA fabric

On the other side, the hierarchy of netlists also impact the physical design significantly. Figure 7.13 illustrates an example of Verilog netlists which are outputted by the OpenFPGA, which models a complete FPGA fabric in a hierarchical way. Note that highly hierarchical fabrics are generated, where large FPGAs can be built with a small number of repeatable tiles including routing blocks. Tiles and routing blocks are built with common primitive blocks, located in the `sub_module` directory, which can maximize the reuse of primitive netlists. Repeatable tiles can efficiently reduce the file sizes, total runtime, and design complexity of physical design flow. For example, in a physical design methodology, only unique tiles are placed and

```
<pb_type name="clb">
  <input name="I" num_pins="10" equivalent="full"/>
  <output name="O" num_pins="4" equivalent="none"/>
  <clock name="clk" num_pins="1"/>
<!-- child pb_type may be defined underneath -->
<pb_type>
```

(a) An examples of a CLB definition in UTFAL

```
module logical_tile_clb_mode_clb_(prog_clk, clb_I, clb_clk,
ccff_head, clb_O, ccff_tail);
  input [0:0] prog_clk;
  input [0:9] clb_I;
  input [0:0] clb_clk;
  input [0:0] ccff_head;
  output [0:3] clb_O;
  output [0:0] ccff_tail;
<!-- internal structure of clb is omitted -->
endmodule
```

(b) An example of a CLB in auto-generated Verilog netlists

Fig. 7.14 An example of auto-generated human-readable netlists corresponding to architecture definition

routed, while the top-level fabric is only an assemble of the tiles which are treated as black boxes [23].

Note that different physical design tools may require different HDL formats and their specific variants. Verilog is a popular HDL format for most physical design tools, while VHDL is more popular as a strict behavioral modeling for FPGA fabrics. Modern netlist generators include various netlist writers to convert a graph representation of FPGA fabric to the file format which meets downstream tool requirements. Even when considering Verilog format, various netlist styles may be demanded, in order to be compatible with latest Verilog standards. For instance, the syntax `default_nettype` is introduced to force strict wire definition in Verilog 2001. Supporting diverse syntax allows the auto-generated netlists to be more human readable and easier to back-trace errors for chip designers, especially when there are implementation errors during physical design flow. To further improve readability of outputted netlists, names of modules, ports, and nets should be human readable and correspond to architecture description. Figure 7.14 shows an example how the outputted netlist can be easy to correlated to the architecture description. In Fig. 7.14a, a programmable block `clb` with two input ports and one output is defined using the UTFAL. Figure 7.14b presents the Verilog codes which are outputted by OpenFPGA, corresponding to the programmable block `clb`. The port name and port size are consistent between the architecture description and the netlists, through which chip designer can backtrace the changes in netlists to a specific portion of architecture file. For instance, the port `I` of `clb` in Fig. 7.14a is named as `clb_I` in Fig. 7.14b.

We refer interested readers to [20] for a detailed implementation of netlist generator.

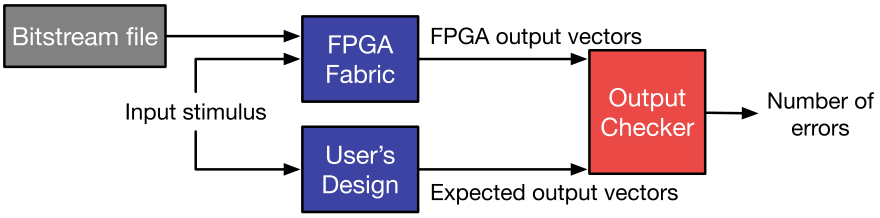


Fig. 7.15 Principles of Verilog testbenches: (1) using common input stimuli; (2) applying bitstream; (3) checking output vectors

Table 7.4 Auto-generated testbench features

Testbench	Runtime	Test vector	Test coverage
Full	Long	Random stimulus	Full fabric
Preconfigured	Short	Random stimulus/formal method	Programmable fabric only

7.4 Testbench Generator

It is essential to validate the correctness of FPGA fabrics before tape-out. However, a key difference between the design verification for FPGAs and ASICs lies on bitstreams. As highlighted in Fig. 7.15, an FPGA carries a specific functionality only when an associated bitstream is loaded. To ensure a high verification coverage, chip designers need a number of bitstream files, each of which is designed to validate a specific part of the FPGA. The bitstream files could be either synthetic (not synthesizable through HDL-to-bitstream tools) or based on a user's RTL design. To validate the various bitstream on an FPGA, testbenches have to be generated with dedicated I/O mapping for each configuration. Note that for most applications, only part of FPGA I/Os are used and for each application, each FPGA I/O may be used in a different way. Testbench generators assign the I/O mapping based on the results from HDL-to-Bitstream results. To enable self-testing, the FPGA and user's RTL design (simulated using an HDL simulator) are driven by the same input stimuli, and any mismatch on their outputs are reported as errors.

To trade-off runtime and coverage, as listed in Table 7.4, two types of testbenches are typically generated to validate the correctness of the fabric before tape-out: full and preconfigured. Full testbench aims at simulating an entire FPGA operating period, consisting of two phases:

1. the configuration phase, where the bitstream file is loaded to the programmable fabric through a configuration protocol, as highlighted by the green rectangle of Fig. 7.16;
2. the operating phase, where random input vectors are applied to drive both *Devices Under Test* (DUTs), as highlighted by the red rectangle of Fig. 7.16.

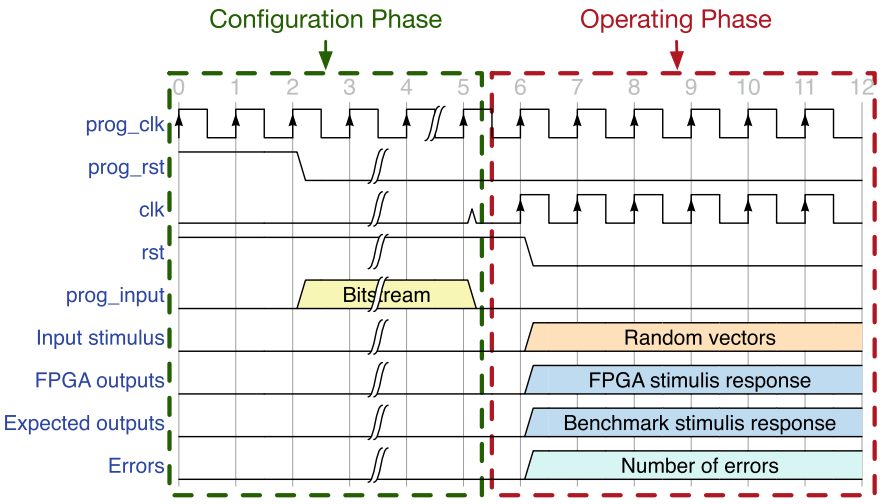


Fig. 7.16 Illustration on the waveforms in full testbench

Using the full testbench, chip designers can validate both the configuration circuits and programming fabric of an FPGA. However, the random testing vectors used in the full testbench may result in only a small set of functional coverage. On the other side, as the bitstream size increases exponentially with the FPGA size, the number of clock cycles required to load the bitstream becomes a dominating factor (more than 90%) in the verification runtime. For instance, HDL simulation of a full testbench including a 800k-bit bitstream consumes a 24-hour runtime when using a commercial state-of-the-art simulator. In short, even there are significant limitations, the full testbench remains a must-run verification, since it fully validates the configuration protocol.

To improve the coverage, the preconfigured testbench is proposed, which skips the time-consuming configuration phase and focus on the operating phase. As a result, sufficient number of testing vectors can be applied to ensure functional correctness of a mapped FPGA design, while simulation runtime is fairly small. To apply testing vectors to mapped I/Os of an FPGA, a preconfigured FPGA, which is instantiated with the user's bitstream, is encapsulated with the same port mapping as the user's RTL design, as illustrated in Fig. 7.17. Note that beyond the functional verification show in Fig. 7.15, the preconfigured FPGA module can be also fed to a formal tool for a 100% coverage formal verification against user's RTL design. Compared to the full testbench, the preconfigured testbench significantly accelerates the functional verification especially for large FPGAs.

We believe that with proper use of the two types of testbenches, the verification process for FPGAs can be significantly simplified or even automated.

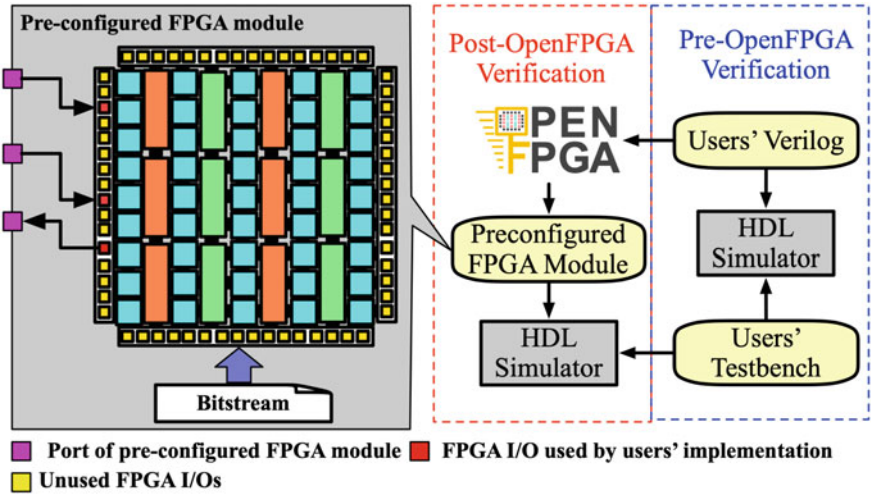


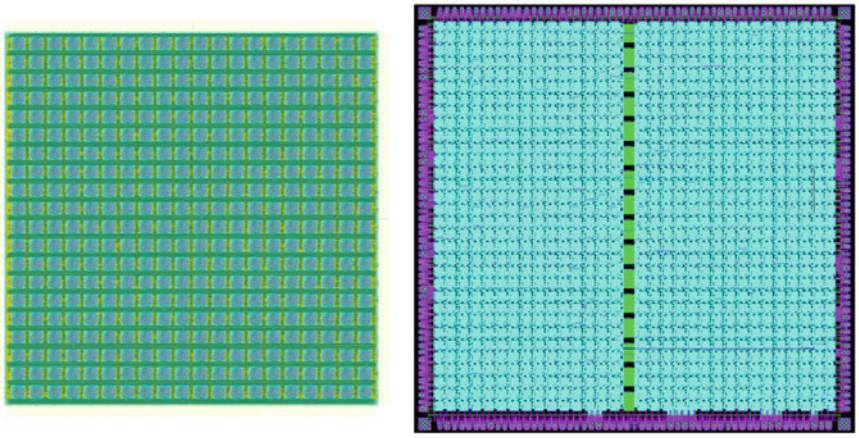
Fig. 7.17 Internal structure of a pre-configured FPGA module

7.5 Showcase

In this part, three FPGA fabrics produced by semi-custom EDA approaches are presented and then compared to a commercial baseline Stratix IV [24]:

1. a 20 × 20 homogeneous FPGA using a commercial 40 nm technology, built with standard cells only [8] (see layout in Fig. 7.3a);
2. a 20 × 20 homogeneous FPGA using a commercial 40 nm technology, built with standard cells only [18] (see layout in Fig. 7.18a);
3. a 32 × 32 heterogeneous FPGA using a commercial 12 nm technology, built with a mix of standard cells and custom cells [14] (see layout in Fig. 7.18b).

Note that through semi-custom approaches, the layout generation of the FPGA fabrics are within 24 h, but their architectures, technologies, and detailed methodologies are different. In all the FPGAs, each tile includes 10 *Logic Elements* (LEs) and a local routing architecture with 50% connectivity. The LE of homogeneous FPGAs consists of a 6-input fracturable LUT, a 4-input LUT, two 1-bit adders, and two flip-flops, which can operate in 6 different modes. The heterogeneous FPGA employs a simplified LE but without the 4-input LUT and also consists of a column of 512 Kb *Block RAMs* (BRAMs), generated by a foundry memory compiler. Full details about the showcased FPGA fabrics are listed in Table 7.5.



(a) Homogeneous (40nm) (b) Heterogeneous (12nm)

Fig. 7.18 Complete layout of FPGA fabrics

Table 7.5 Comparison on the FPGAs in Figs. 7.3a and 7.18

Resource/capacity	Standard homo [8]	Custom homo [18]	Standard hetero [14]
Array size	20 × 20	20 × 20	32 × 32
Tileable routing	×	×	✓
Fracturable 6-input LUTs	4 k [a]	4 k	9.92 k
4-input LUTs	N/A	8 k	N/A
1-bit full adder	8 k	8 k	19.84 k
Flip-flops	8k	8k	19.84 k
Block RAM	N/A	N/A	512 k bits
I/Os	N/A [b]	480	124
Routing channel width	300	300	200
Routing wires	87% L4	87% L4	L4
	13% L16	13% L16	
Fc_{in}	0.055	0.055	0.15
Routing multiplexer	tree-like	one/two-level	tree-like
Backend details	Standard homo [8]	Custom homo [18]	Standard hetero [14]
Tool	Cadence encounter v09.12	Cadence Innovus 19.1	Synopsys ICC2 2019.03
Layout area	16.89 mm^2	7 mm^2	9 mm^2
Flow type	Flatten	Two-step flatten	Hierarchical
Runtime (h)	20–24	24	12
Peak memory (GB)	64	60	215

[a] Each 6-input LUT contains 8 inputs
[b] Not reported

7.5.1 Methodologies

The homogeneous FPGA in [8] is generated by an in-house netlist generator based on VTR, while the rest of FPGA fabrics are generated by OpenFPGA [18]. Note that the netlists for the homogeneous FPGA in [8] were auto-generated in behavioral Verilog codes and optimized by Synopsys Design Compiler before physical design with a strategy to balance area and delay. The netlists auto-generated by OpenFPGA are technology mapped and directly used for physical design tools. Regarding circuit designs, the homogeneous FPGA in [8] and the heterogeneous FPGA in [14] is built with standard cells provided by a commercial 40nm technology, while the homogeneous FPGA in [18] adapts custom cells for routing multiplexers and configuration memory elements. Note that the homogeneous FPGA in [18] uses two-level structures for the multiplexers in *Connection Blocks* (CBs) and *Switch Blocks* (SBs) and local routing architecture, while one-level structure for those in LE. To guarantee high-performance, routing multiplexers are buffered at both inputs and outputs while LUTs are buffered at inputs, outputs, and every two intermediate stages.

The FPGA fabrics are implemented using three different physical design strategies. The homogeneous FPGA in [8] was implemented using a flatten backend flow with design constraints to force layout regularities. The homogeneous FPGA in [18] was implemented using a two-step backend flow where *Configurable Logic Blocks* (CLBs) are P&Red first and then instantiated at the top-level as hard macros. To leverage the symmetry of an FPGA fabric, the heterogeneous FPGA adopted a more hierarchical backend flow, where a library of hard macros for CLBs, CBs, and SBs is built and then assembled in the final layout. The hierarchical backend flow allows chip designers to optimize each hard macro with respect to the timing constraints generated by our tool with few combinational loops to be broken. Therefore, the heterogeneous FPGA is larger in array size, while its backend is 2× faster than the homogeneous. Commercial signoff tools are then used to ensure that all the fabrics are DRC-clean, and timing extraction is performed by using Synopsys PrimeTime.

7.5.2 Performance Evaluation

For a comprehensive analysis, the area, pin-to-pin delays, and the delays of the implemented benchmarks are considered when evaluating the FPGA fabrics. Table 7.6 compares the custom homogeneous FPGA in [18] to two baselines, a commercial Stratix-IV FPGA and the standard homogeneous FPGA in [8]. We believe it is a fair comparison since these FPGAs are similar in architecture and also implemented using 40nm technologies. The results prove the high value of using one-level and two-level multiplexing structures as well as an optimized cell library, which can improve the area by 42% and path delay by 30% when compared to a standard cell FPGA. Indeed, there are considerable gaps in area (60%) and path delays (20%) between the semi-custom-designed FPGAs and the full-custom-designed commercial FPGA.

Table 7.6 Area and delay comparison between [8, 14, 18] and Stratix IV

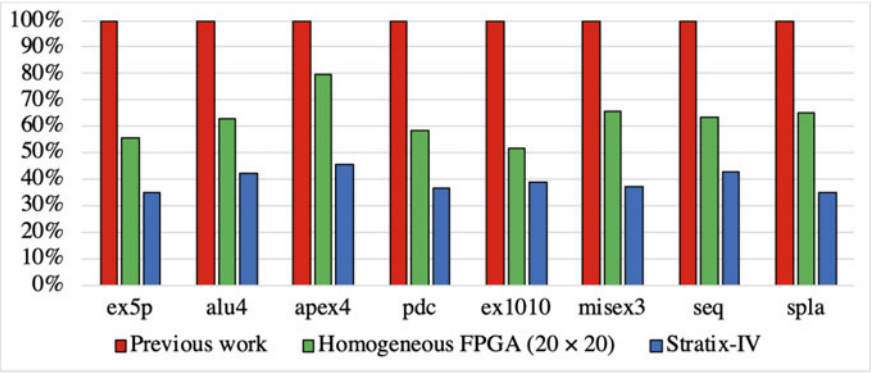
Generality	Standard homo [8]	Custom homo [18]	Standard hetero [14]	Stratix IV
Technology	40 nm	40 nm	40 nm	12nm
Cell Library	Standard	Custom[a]	Custom	Standard
Tile Area (μm^2)	30,625 (100%)	17,648 (-42%)	11,050 (-63%)	8,373 (-72%)
Path delay (ns)	Standard homo [8]	Custom homo [18]	Standard hetero [14]	Stratix IV
Process Corner	TT	SS	SS[b]	TT
6-LUT	0.5 (100%)	0.27 (−46%)	0.28 (−44%)	0.23 (−54%)
20-bit Adder[c]	1.63 (100%)	2.12 (+30%)	1.23 (-25%)	1.13 (−31%)
Local Routing[d]	0.27 (100%)	0.17 (−37%)	0.23 (-15%)	0.15 (−44%)
L4 track[e]	2.53 (100%)	0.82 (−67%)	0.59 (−76%)	0.75 (−70%)
Average	100%	−30%	−40%	(−50%)

[a] Use custom cells only in routing multiplexers and configuration chains
[b] The rest are standard cells. See details in [18]
[c] Consider the slow model in Quartus STA
[d] Local routing path starts from a BLE output and ends at a BLE input
[e] Lx track: FF→length-x wire→Local Routing→LUT→FF

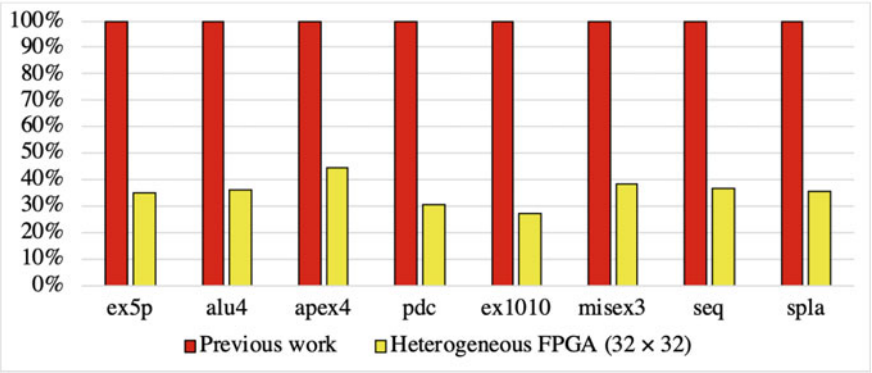
Even though there is an intrinsic PPA gap between standard-cell layouts and full-custom layouts, the performance gap can be reduced through a careful co-design between backend strategies and custom cell implementations [7].

For performance benchmarking, eight MCNC circuits are selected to fit all the 40nm FPGAs. Each benchmark is verified through the verification techniques in Sect. 7.4, using Mentor ModelSim and Synopsys Formality. Quartus 18.1.0 is used to implement the same benchmark set as the industry baseline, and the device model is set to the Stratix IV EP4S40G2F40C2. Figure 7.19a shows that FPGAs using custom cells is 2× slower on average than the Stratix IV. The gap comes from the hardware lags in performance, with an average of 20%. When critical paths consist of multiple routing paths listed in [Tab. 7.6], the delay difference will aggregate. The gap comes from sources:

1. the hardware lags in performance with an average of 10%. When critical paths consist of multiple routing paths listed in Table 7.6, the delay difference will aggregate. Therefore, the longer the critical path is, the larger the performance gap will be.



(a) Impact of custom cells and multiplexers on FPGAs at 40nm



(b) Standard cell FPGAs scaling from 40nm to 12nm

Fig. 7.19 Delay comparison between OpenFPGA and [8] (marked as previous works) using selected MCNC benchmarks

2. previous studies have shown a large gap between VPR CAD algorithms and commercial counterparts [25]. The performance gap may be as large as 55% on average, fully shadowing any efficiency on hardware.

This indicates that developing efficient CAD algorithms that can match industry quality should be a frontier for the open-source FPGA research community.

We compare the heterogeneous FPGA in Fig. 7.18b to the homogeneous FPGA [8], as both FPGAs are implemented by standard cells and also similar in architecture while using different technologies. Our results show that using semi-custom design approaches, FPGA architectures can be portable between different technology nodes and benefit significant performance improvements. In Table 7.6, the 12 nm FPGA is 72% smaller in area and 50% faster in path than the 40nm baseline. In Fig. 7.19b, the heterogeneous FPGA is 3× faster on average in benchmark delays than the 40 nm baseline.

7.6 Summary and Trends

Semi-custom design approaches have become a warm research topics in recent years, as different design methodology than commercial state-of-the-art FPGAs that are built through full custom approaches.

To enable semi-custom design approaches, innovative EDA tools have been developed as an unified framework for netlist generation, testbench generation and bitstream generation. Due to the automation in modern EDA tools, development cycle of FPGA layouts as well as engineering effort can be remarkably reduced. However, the semi-custom design approach is in its infancy stage, as we see non-negligible PPA gaps against commercial FPGAs.

Since most of the EDA tools are accessible in open-source community, future researches may focus on performance improvement on the design methodology, e.g., physical design techniques. In addition, being tightly integrated to architecture exploration tools, the EDA tools enable fast prototyping for innovative FPGA architectures. In other words, architecture exploration can achieve realistic PPA evaluation in a short development cycle, and effectiveness of architecture enhancements can be validated through layout-level results in a short period, as compared the full-custom approach. Also, with the expansion in open-source community for FPGAs, novel EDA algorithms, e.g., packing, placement and routing, may be studied and validated through physical FPGA fabrics using semi-custom design approach. Previously, the validation of EDA algorithms is typically based on hypothetical FPGA fabrics and high-level analysis methods, which has been proven to be inaccurate.

In short, semi-custom design approaches have changed the cost function to design, evaluate, and produce new FPGA fabrics, stimulating many research opportunitie in novel FPGA architecture, efficient physical design techniques, and novel EDA algorithms.

References

1. S. Che, J. Li, J.W. Sheaffer, K. Skadron, J. Lach, Accelerating compute-intensive applications with GPUS and FPGAs, in *2008 Symposium on Application Specific Processors* (2008), pp. 101–107
2. C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong, Optimizing FPGA-based accelerator design for deep convolutional neural networks, in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. (Association for Computing Machinery, New York, NY, USA, 2015), pp. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060
3. J. Cong, Z. Fang, M. Huang, L. Wang, D. Wu, CPU-FPGA coscheduling for big data applications. IEEE Design Test **35**(1), 16–22 (2018)
4. K. Neshatpour, H.M. Mokrani, A. Sasan, H. Ghasemzadeh, S. Rafatirad, H. Homayoun, Architectural considerations for FPGA acceleration of machine learning applications in mapreduce," in *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. SAMOS '18 (Association for Computing Machinery,

New York, NY, USA 2018), pp. 89–96. [Online]. Available: https://doi.org/10.1145/3229631.3229639
5. D. Greenhill, R. Ho, D. Lewis, H. Schmit, K.H. Chan, A. Tong, S. Atsatt, D. How, P. McElheny, K. Duwel, J. Schulz, D. Faulkner, G. Iyer, G. Chen, H.K. Phoon, H.W. Lim, W.-Y. Koay, T. Garibay, 3.3 a 14nm 1ghz FPGA with 2.5d transceiver integration, in *2017 IEEE International Solid-State Circuits Conference (ISSCC)* (2017), pp. 54–55
6. I. Kuon, A. Egier, J. Rose, Design, layout and verification of an FPGA using automated tools, in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '05 (Association for Computing Machinery, New York, NY, USA, 2005), pp. 215–226. [Online]. Available: https://doi.org/10.1145/1046192.1046220
7. Aken'Ova, V., Saleh, R., A "soft++" EFPGA physical design approach with case studies in 180 nm and 90 nm, in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)* (2006), pp. 6
8. J.H. Kim, J.H. Anderson, Synthesizable FPGA fabrics targetable by the verilog-to-routing (VTR) CAD flow, in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)* (2015), pp. 1–8
9. B. Grady, J.H. Anderson, Synthesizable heterogeneous FPGA fabrics, in *2018 International Conference on Field-Programmable Technology (FPT)* (2018), pp. 222–229
10. H.J. Liu, Archipelago - an open source FPGA with toolflow support (2014)
11. A. Li, D. Wentzlaff, Prga: an open-source FPGA research and prototyping framework, in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. (Association for Computing Machinery, New York, NY, USA, 2021), pp. 127–137. [Online]. Available: https://doi.org/10.1145/3431920.3439294
12. D. Koch, N. Dao, B. Healy, J. Yu, A. Attwood, Fabulous: an embedded FPGA framework, in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. (Association for Computing Machinery, New York, NY, USA, 2021), pp. 45–56. [Online]. Available: https://doi.org/10.1145/3431920.3439302
13. P. Mohan, O. Atli, O. Kibar, M. Zackriya, L. Pileggi, K. Mai, Top-down physical design of soft embedded FPGA fabrics, in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. (Association for Computing Machinery, New York, NY, USA, 2021), pp. 1–10. [Online]. Available: https://doi.org/10.1145/3431920.3439297
14. X. Tang, E. Giacomin, B. Chauviere, A. Alacchi, P.-E. Gaillardon, OpenFPGA: an open-source framework for agile prototyping customizable FPGAs. IEEE Micro **40**(4), 41–48 (2020)
15. J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K.B. Kent, J. Anderson, J. Rose, V. Betz, VTR 7.0: next generation architecture and cad system for FPGAs. ACM Trans. Reconfigurable Technol. Syst. **7**(2) (2014). [Online]. Available: https://doi.org/10.1145/2617593
16. K.E. Murray, O. Petelin, S. Zhong, J.M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A.G. Graham, J. Wu, M.J.P. Walker, H. Zeng, P. Patros, J. Luu, K.B. Kent, V. Betz, VTR 8: high-performance cad and customizable FPGA architecture modelling. ACM Trans. Reconfigurable Technol. Syst. **13**(2) (2020). [Online]. Available: https://doi.org/10.1145/3388617
17. J. Luu, Architecture-aware packing and cad infrastructure for field-programmable gate arrays. Ph.D. dissertation, University of Toronto (2014)
18. X. Tang, E. Giacomin, A. Alacchi, B. Chauviere, P.-E. Gaillardon, OpenFPGA: an opensource framework enabling rapid prototyping of customizable FPGAs, in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. (IEEE, 2019), pp. 367–374
19. V. to Routing, Verilog-to-routing documentation (2022) [Online]. Available: https://docs.verilogtorouting.org/en/latest/arch/
20. X. Tang, OpenFPGA documentation (2022). [Online]. Available: https://openfpga.readthedocs.io/en/master/
21. I. Kuon, R. Tessier, J. Rose (2008)
22. X. Tang, E. Giacomin, G. De Micheli, P.-E. Gaillardon, Circuit designs of high-performance and low-power rram-based multiplexers based on 4t(ransistor)1r(ram) programming structure. IEEE Trans. Circ. Syst. I: Regular Papers **64**(5), 1173–1186 (2017)

23. G. Gore, X. Tang, P.-E. Gaillardon, A scalable and robust hierarchical floorplanning to enable 24-hour prototyping for 100k-LUT FPGAs, in *Proceedings of the 2021 International Symposium on Physical Design*, ser. ISPD '21. (Association for Computing Machinery, New York, NY, USA, 2021), pp. 135–142. [Online]. Available: https://doi.org/10.1145/3439706.3447047
24. D. Lewis, E. Ahmed, D. Cashman, T. Vanderhoek, C. Lane, A. Lee, P. Pan, Architectural enhancements in Stratix-III™ and Stratix-IV™, in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09. (Association for Computing Machinery, New York, NY, USA, 2009), pp. 33–42. [Online]. Available: https://doi.org/10.1145/1508128.1508135
25. E. Hung, Mind the (synthesis) gap: examining where academic FPGA tools lag behind industry, in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)* (2015), pp. 1–4