Chapter 3
Design (Application Design) Modeling

Abstract Application design is the bridge between end user's idea and FPGA's functional units. Modeling it will build up application design data structure—the ballast stone of any EDA engine in this stage. This chapter dives into the principles and implementations of FPGA design (application design) modeling, showing that how these models are classified and described.

3.1 Design Description Levels

3.1.1 Abstract Levels

FPGA application design's abstract levels are quite similar to CPU's: natural level [1], high level [2], low level [3], machine level [4] and physical level (Fig. 3.1).

In the field of CPU (scalar computing) application design, machine language is a series of instruction sequences composed of "0" and "1", directly interacting with the hardware at the bottom layer (for FPGA application design, the binary bitstream system); assembly language use abbreviated identifiers in its instructions to operate on the hardware (for FPGA application design, hardware description language is widely used to describe hardware circuits, requiring developers to have a considerable degree of low-level hardware knowledge); high-level language is more or less independent to a particular type of computing architecture and has already been the first choice for most computer programmers (for FPGA application design, it is also getting more and more commonly used); natural language is the way of communication between humans and considered to be the ultimate way to communicate with computing engines; there has been lots of research works in processing it (for FPGA application design, there is still a long way to go).

Just like chip design models (Sect. 2.1), application design models can also be theoretically described at similar abstract levels (Fig. 3.2). Related formats and standards have been intensively studied in this field to improve the design productivity.

K. Tu et al., *FPGA EDA*, https://doi.org/10.1007/978-981-99-7755-0_3

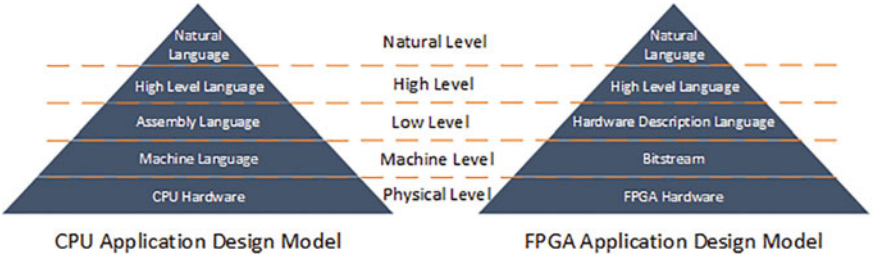

Fig. 3.1 Abstract levels of application design

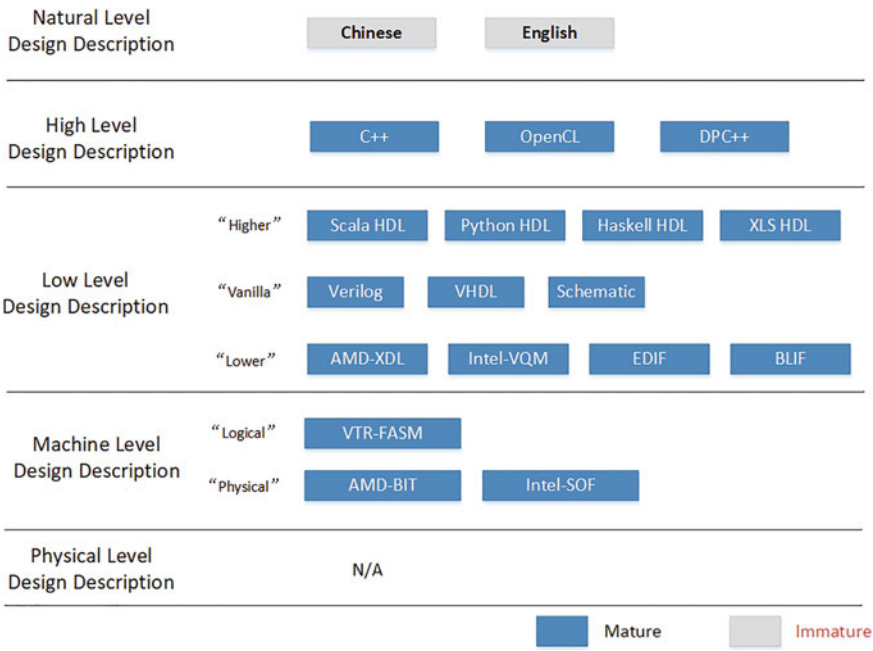

Fig. 3.2 FPGA design description abstract levels

1. Machine-Level Description
   The machine-level description of an application design refers to the expression of the bitstream (generally in binary), which directly controls the hardware behavior of the FPGA. Similar to device models, the "logical" description presents the correlation between every programmable bits in the bitstream and hardware resources (such as the FASM format file [5]); the "physical" description then is the final value of each configuration bits in a physical order determined by the configuration protocols (such as AMD's BIT format file and Intel's SOF format file).

2. Low-Level Description
   Identical to device models, three sub-levels are shown here:
   At "vanilla" level, traditional hardware description (such as Verilog, VHDL, or Schematics) is widely used to build the design model.
   At "higher" level, design descriptions with higher abstraction (such as Scala HDL, Python HDL, Haskell HDL, XLS HDL) could be a powerful complement to traditional descriptions.
   At "lower" level, more detailed hardware descriptions (such as AMD-XDL, Intel-VQM, BLIF, EDIF) is used to specify lower units in the FPGA design.
3. High-Level Description
   The high-level description for FPGA application design refers to software-oriented languages (such as C/C++, OpenCL, SystemC, DPC++).
   Inspired by the Open Computing Language (OpenCL) programming for heterogeneous systems, Intel has defined the Data Parallel C++ (DPC++) design language as its cross-architecture (CPU, GPU, FPGA) programming.
4. Natural-Level Description
   The automatic conversion of natural language into a language that FPGA can "understand" is also the future research direction of the academic community.

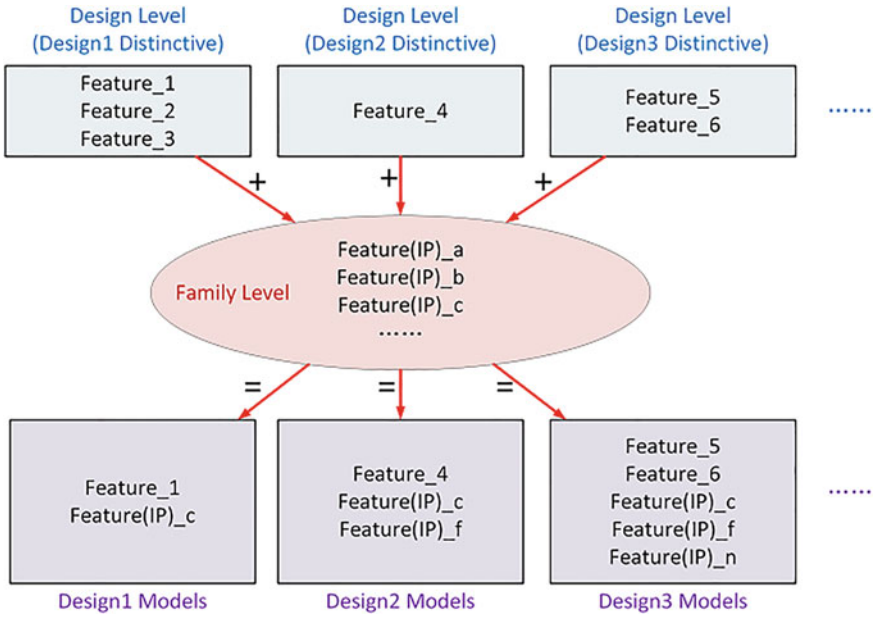

Fig. 3.3 FPGA design description reuse levels

3.1.2 Reuse Levels

From application design perspective, IP-based design methodology is the mainstream way of increasing reusability. IP (both soft ones and hard ones embedded in the FPGA) is generally family shared, which means it can be called when using any device under the supported families (Fig. 3.3). In modern FPGA application design EDA tools, IP integrator is an standard function that will not be absent, enabling users to get fast access to these predefined units.

3.2 Design Model Classifications

Similar to device (chip design) information, the design (application design) information of an FPGA can also be organized in classes: primary class, constraint class and report class. The "design checkpoint", built from these models, contains all the EDA information related to the application design.

The primary information is the torso of the design, the constraint information is set to direct the working strategy of EDA engines, then the report information shows the concerned metrics, helping designers to better analyze the current situation. If the reported results are not satisfactory, the design will be modified and then recurrently approaches the optimized goal.

3.2.1 Primary Class

Identical to device models, the primary models of application design EDA also include logical resource structure model and configuration bit structure model. Nevertheless, the substantial contents of them are quite different from the previous chapter. Again, the same with device models, we introduce design primary models at low abstract level (Fig. 3.1) for the same reason.

1. Logical Resource Structure (LRS) Model
   The LRS of an FPGA application design is usually presented by netlist—a term that describes the components and connectivity of the design. A simplified hierarchy of the design logic resource model is shown in (Fig. 3.4).
   The design core logic resources in the netlist can be divided into *clusters*(will accommodate in *tiles* in the device logic resources), each *cluster* is composed of *molecules*(will accommodate in *sites* in the device logic resources), and each *molecule* is composed of *atoms*(will accommodate in *primitives* in the device logic resource). Similarly, *atom* is also composed of gate-level units.
   The design interconnect logic resources in the netlist is composed of *nets*, and a *net* represents the connections between FPGA units (the edges of the netlist

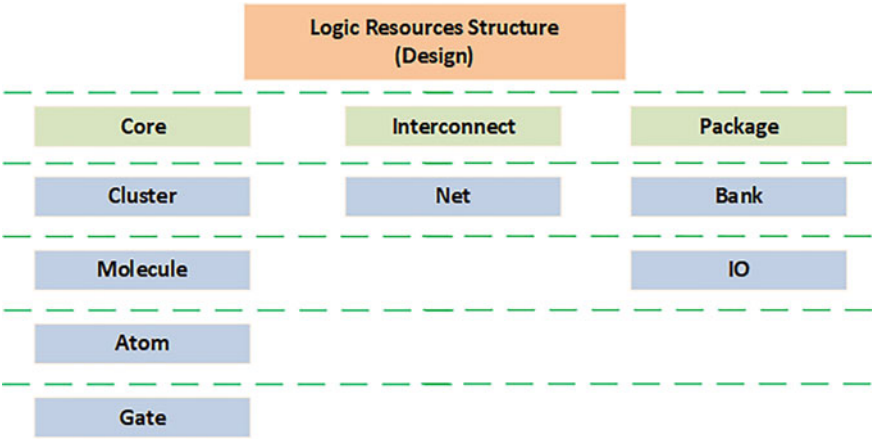

Fig. 3.4 FPGA design logical resource structure level

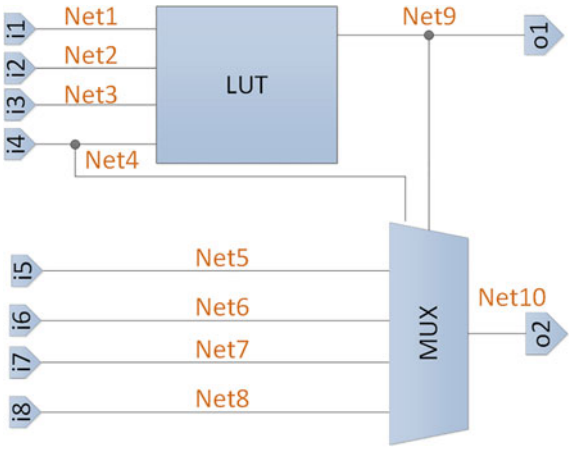

Fig. 3.5 FPGA application design netlist example

hyper-graph). Each net has a single driver pin, and a set of sink pins (will accommodate in *wires/switches* in the device logic resource).
The design IO will accordingly accommodate in Pad units in the device logic resource.
Take (Fig. 3.5) as an example, there are 12 *atoms*(1 LUT, 1 MUX, 8 inputs and 2 outputs) and 10 *nets* joining them altogether.

2. Configuration Bit Structure (CBS) Model
The CBS of an FPGA application design can also be defined from two perspectives: logical and physical.
Logical bit structure collects every active configuration bit's "logical address" of the design, that is, which logic resource it belongs to (Fig. 3.6).

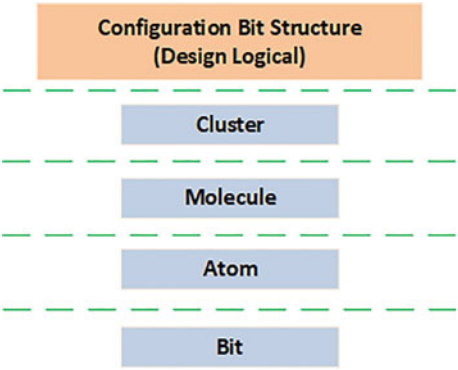

Fig. 3.6 FPGA design configuration bit structure level (logical)

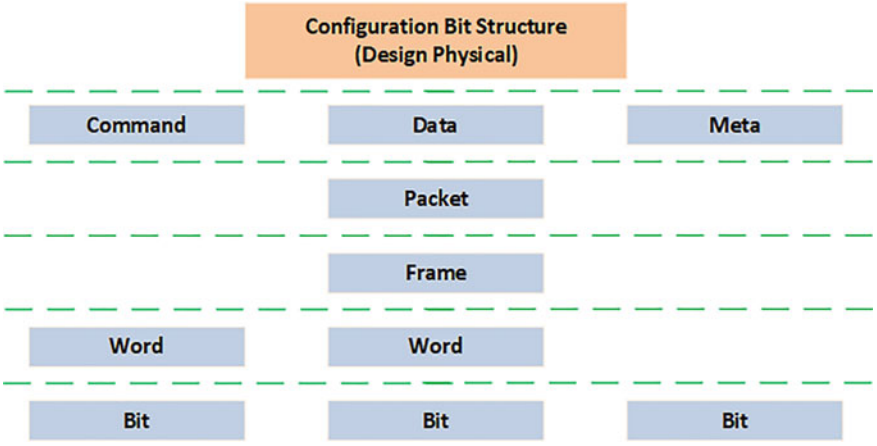

Fig. 3.7 FPGA design configuration bit structure level (physical)

Physical bit structure collects every active configuration bit's "physical address", that is, which position it lies in the final bitstream according to the programming protocol (Fig. 3.7).

After the logical and physical structure are properly identified, the configuration data can be outputted as the desired bitstream format (Fig. 3.8).

3.2.2 Constraint Class

FPGA application design constraints work at specific stage of the design flow, for example, routing constraints are used during the routing stage. Over-constraining or under-constraining the design both may cause sign-off difficulties.

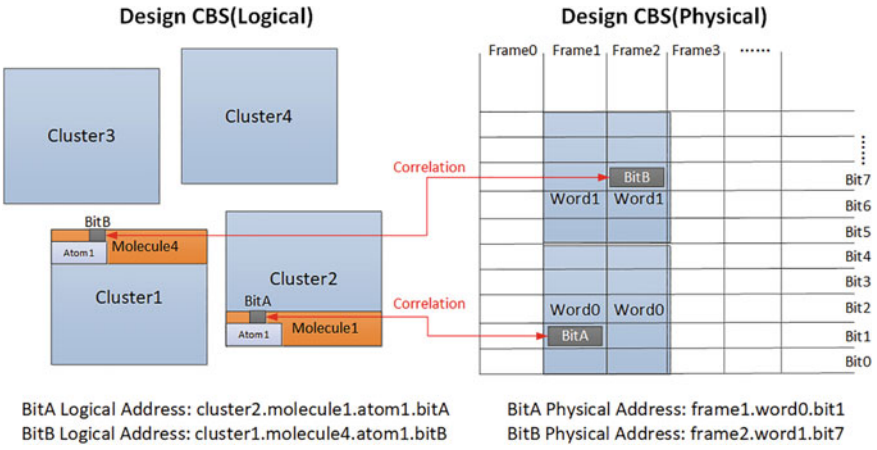

Fig. 3.8 FPGA design configuration bit correlation

TCL (Tool Command Language), pronounced "tickle", is an easy-to-learn scripting language and can run by scripts from either the Windows or Linux command-line. The language is easily extended with new function calls and has been expanded to support new tools and technology since its inception and adoption in the early 1990s. It has been adopted as the standard application programming interface, or API, among most EDA vendors to control and extend their applications.

Most of the FPGA vendors have adopted TCL as the design constraint format for their application EDA tools, as it is easily mastered by designers who are familiar with this industry standard language. The TCL interpreter inside the tool provides the full power and flexibility of TCL to control the flow or set the constraints.

Modern FPGA application design constraints have the following properties:

1. Inherit from industry standard SDC (Synopsys Design Constraint) commands and have its own expansions.
2. They are not simple strings, but are commands that follow the TCL semantic.
3. They can be interpreted like any other TCL command by the TCL interpreter.
4. They are read in and parsed sequentially the same as other TCL commands.

3.2.3 Report Class

Based on the objective (or EDA process) it addressed, the design reports can be divided into many categories: high-level synthesis report, logic synthesis report, physical implementation (packing/placement/routing...) report, analysis (timing/power/resource...) report, bitstream configuration (generation/download) report, and so on.

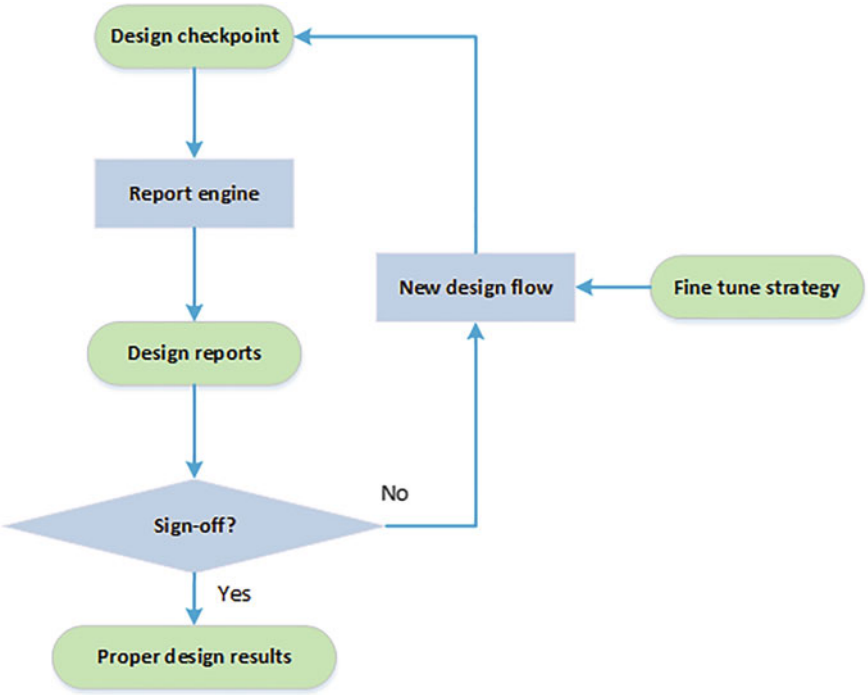

Fig. 3.9 FPGA application design report helps designers to sign-off properly

Design report offers information in human readable format from a specific perspective to help designers focus on the metrics they concern, if any sign-off requirement is not met, iterative modifications can be done until getting the proper solution (Fig. 3.9).

3.3 Design Model Implementations

The previous section listed all the design model classes: primary class, constraint class, and report class. In this section, we will present typical implementation practices of each model (Table 3.1).

3.3.1 Logic Resource Structure Model

In FPGA application design flow, the design netlist carries different information at different EDA stages. At logic synthesis stage, elaboration process turn the design

Table 3.1 Comparison of FPGA design model implementations

Model name	Abstract level	Reuse level	Class
Logic Resource Structure	High/Low	Design	Primary
Configuration Bit Structure	Machine	Design	Primary
Constraint	High/Low	Design	Constraint
Report	High/Low	Design	Report

Table 3.2 FPGA application design netlist formats and the EDA information they could carry ([a] is closed source)

Format	Generic netlist	Synthesized netlist	Implemented netlist	Adopter
RTLIL	Yes	/	/	Yosys
BLIF	Yes	Yes	/	Academia
GTECH[a]	Yes	/	/	Synplify
EDIF	Yes	Yes	/	Industry
VQM	/	Yes	/	Quartus
XDL	/	Yes	Yes	ISE
XDEF[a]	/	Yes	Yes	Vivado
VPR-Verilog	/	/	Yes	VPR
F4PGA-JASON	/	Yes	Yes	NextPnR

into gate-level representation (Generic Netlist), mapping process turn the design into atom-level representation (Synthesized Netlist); at physical implementation stage, cluster-level representation (Implemented Netlist) is generated.

There is no universal FPGA netlist format that can be used throughout the whole EDA process by the time this book is written, however, (Table 3.2) still listed the most popular legacy netlist formats and the EDA stages they could go through.

Implementation example: BLIF [6]

Berkeley Logic Interchange Format (BLIF) aimed to describe a logic-level hierarchical circuit in textual form.

Implementation example: EDIF [7]

Electronic Design Interchange Format (EDIF) is a format that could capture all features of circuit design. It has been accepted as a communications medium to manufacturing equipment and an interchange format between EDA systems.

Implementation example: Intel/Altera VQM [8]

Verilog Quartus Mapping (VQM) is the Intel/Altera version text file that contains a atom-level netlist. VQM files are typically generated by Intel/Altera Quartus.

Implementation example: AMD/Xilinx XDL [9, 10]

Xilinx Design Language (XDL) is the AMD/Xilinx version text file that represents a design netlist after mapping to the FPGA primitives. XDL files are typically generated by AMD/Xilinx ISE.

Table 3.3 SDC Syntax

Command	Supported arguments
Mostly [Verb]_[Noun]	Object / [-argument object]

```
<!-- Post-synthesis design report in XML -->
<atom_usage_report>
  <atoms num="<int>">
    <atom type="<atom_type_name_0>" usage="<int>"></atom>
    <atom type="<atom_type_name_1>" usage="<int>"></atom>
    ......
    <atom type="<atom_type_name_n>" usage="<int>"></atom>
  <input_pins num="<int>"></input_pins>
  <output_pins num="<int>"></output_pins>
  <nets num="<int>"></nets>
</atom_usage_report>
```

Fig. 3.10 Example XML syntax for post-synthesis design report

3.3.2 Configuration Bit Structure Model

1. Logical CBS information
   Implementation example: VTR-FASM [11]
   FPGA Assembly (FASM) is a textual representation of a bitstream. By assigning a symbolic name to each configurable thing in the FPGA, the resulting FASM file shows what features are specifically configured "on". These files provide an easy way to write programs that manipulate bitstreams. Modifying a textual FASM file is far easier than trying to modify a binary bitstream.
2. Physical CBS information
   Implementation example: AMD/Xilinx-BIT [12, 13]
   BIT files are AMD/Xilinx FPGA configuration files containing configuration information. In this file, each four bytes is a packet (analogous to CPU instruction). The packet could be a special header, or only carrying normal data. The header packet follows a simple assembly-like instruction set to dictate the configuration process.

3.3.3 Constraint Model

Synopsys's design constraint model (SDC) (Table 3.3) is the heart of all modern FPGA application design constraint models.

Implementation example: xDC ("x" represents the vendor)

FPGA vendors usually extend their constraint syntax based on SDC (because SDC cannot cover some FPGA specific syntax, such as physical constraints).

```
<!-- Packing report in XML -->
<packing_report>
  <cluster num="<int>">
    <unit name="<cluster_name_0>" accommodation_type="tile_type0"/>
    <unit name="<cluster_name_1>" accommodation_type="tile_type1"/
    ......
  </cluster>
  <input_pins num="<int>"/>
  <output_pins num="<int>"/>
  <nets num="<int>"/>
</packing_report>
```

Fig. 3.11 Example XML syntax for packing report

```
<!-- Placement report in XML -->
<placement_report>
  <unit name="cluster_name_0" accommodation_address="x0_y0_z0"/>
  <unit name="cluster_name_1" accommodation_address="x1_y1_z1"/>
  ......
</placement_report>
```

Fig. 3.12 Example XML syntax for placement report

```
<!-- Routing report in XML -->
<routing_report>
  <net name="net_name_0">
    <unit start="pin_0" end="pin_1"/>
    <unit start="pin_2" end="pin_3"/>
    <branch>
      <unit start="pin_0" end="pin_1"/>
      <unit start="pin_2" end="pin_3"/>
    </branch>
    <branch/>
    ......
  </net>
  <net name="net_name_1"/>
  ......
</routing_report>
```

Fig. 3.13 Example XML syntax for routing report

Universal FPGA constraint syntax still needs time to be standardized across vendors.

3.3.4 Report Model

Each FPGA vendor or academic organization has its own reporting style. Universal FPGA report syntax still needs time to emerge.

```
<!-- Power report in XML -->
<power_report>
  <power_models name="final"/>
  <total_power value="<float>"/>
  <transceiver_static_power value="<float>"/>
  <transceiver_dynamic_power value="<float>"/>
  <core_static_power value="<float>"/>
  <core_dynamic_power value="<float>"/>
  ......
</power_report>
```

Fig. 3.14 Example XML syntax for power report

```
<!-- Timing report in XML -->
<timing_report>
  <path start="atom_0" end="atom_1" type="setup/hold"
slack="<float>"/>
  <path start="atom_2" end="atom_3" type="setup/hold"
slack="<float>"/>
  ......
</timing_report>
```

Fig. 3.15 Example XML syntax for timing report

1. Post-synthesis report
   Implementation example: (Fig. 3.10)
2. Post-implementation report
   Implementation example: (Figs. 3.11, 3.12 and 3.13)
3. Power report
   Implementation example: (Fig. 3.14)
4. Timing report
   Implementation example: (Fig. 3.15)

References

1. Wikipedia, Natural language (2022). https://en.wikipedia.org/wiki/Natural_language
2. Wikipedia, High level language (2022). https://en.wikipedia.org/wiki/High-level_programming_language
3. Wikipedia, Assembly language. (2022) https://en.wikipedia.org/wiki/Assembly_language
4. Wikipedia, Machine code (2022). https://en.wikipedia.org/wiki/Machine_code
5. F4PGA, FPGA assembly (FASM) (2021). https://fasm.readthedocs.io/en/latest/
6. U. of California Berkeley, Berkeley logic interchange format (1992). https://people.eecs.berkeley.edu/~alanmi/publications/other/blif.pdf
7. H.J. Kahn, R.F. Goldman, The electronic design interchange format EDIF: present and future, in *Proceedings of the 29th ACM/IEEE Design Automation Conference*, Series DAC '92 (IEEE Computer Society Press, Washington, DC, USA, 1992), pp. 666–671
8. A. QUIP, VQM extractor and language functional description (2005)

9. C. Beckhoff, D. Koch, J. Torresen, The Xilinx design language (XDL): tutorial and use cases, in *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)* (2011), pp. 1–8
10. B.J.P. Tomas, Xilinx design language (2012). http://www.ee.unlv.edu/~selvaraj/ecg707/lecture/XilinxDesignLanguage.pdf
11. B.J.P. Tomas, FPGA assembly (FASM). https://fasm.readthedocs.io/en/latest/
12. AMD/Xilinx, Xilinx bit bitstream files. http://www.pldtool.com/pdf/fmt_xilinxbit.pdf
13. Y. Shan, FPGA bitstream explained. http://lastweek.io/fpga/bitstream/