

Aarno Lindqvist

DEVELOPING LOGIC SYNTHESIS FLOW FOR NVDLA IP

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Prof. Timo Hämäläinen
M.Sc. Saman Payvar
May 2022

ABSTRACT

Aarno Lindqvist: Developing Logic Synthesis Flow for NVDLA IP
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
May 2022

Modern digital devices require high computing performance; thus, markets have a huge demand for SoC. The most powerful SoC are implemented on ASIC chips since, it is the most cost-efficient technology when production volumes are high. An important step on ASIC design process is the logic synthesis. By utilizing dedicated software tool, it transfers RTL code into gate-level netlist. The logic synthesis process is executed multiple times alongside the RTL code development to meet the desired specifications for the chip.

This thesis project used the NVDLA IP as a use case to execute logic synthesis. NVDLA is an open-source deep learning accelerator developed by NVIDIA. The design is able to execute CNNs making it efficient. Each component in the NVDLA can be configured independently, which make it flexible and cost effective. NVDLA software ecosystem has extensive cover of software features. NVDLA is divided into five partitions according to their functionality. Each partition is an individual top-level synthesis hierarchy.

The target of this thesis is to develop a logic synthesis flow for NVDLA in the company design environment. This was achieved by exploiting NVDLA design environment, company internal memory wrapper, and Synopsys Design Compiler and IC Compiler 2 tools to execute logic synthesis for TSMC 7 nm standard cell technology. All the used RTL codes and scripts were downloaded from NVDLA GitHub webpage. The memory wrapper was created by the company memory wrapper tool. It connects the NVDLA design and the RAM instances. The Design Compiler tool was used to generate the initial netlist for NVDLA partitions. The IC Compiler 2 tool was used to create individual floorplans for each partition. The generated DEF file was used for second pass synthesis to obtain the final logic synthesis results. The results demonstrate that the company design environment can be used to run synthesis for open-source IP blocks. Further, the developed flow provides a platform to exploit different kind of open-source IP's on industrial development environment since, it can generate synthesis results for 7 nm standard cell technology quickly.

Keywords: SoC, ASIC, Logic Synthesis, NVDLA, STA

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Aarno Lindqvist: Logiikka synteesi vuon kehittäminen NVDLA lohkolle
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Toukokuu 2022

Nykyaikaiset digitaaliset laitteet tarvitsevat paljon laskentatehoa, näin ollen markkinoilla on suuri tarve järjestelmäpiireille. Kaikista tehokkaimmat järjestelmäpiirit toteutetaan ASIC piiriteknologialla, koska se on kaikista kustannustehokkain piiriteknologia vaihtoehto kun tuotanto määrät ovat suuria. Tärkeä vaihe ASIC kehitys prosessissa on logiikka synteesi. Hyödyntämällä räätälöityjä ohjelmistotyökaluja, se muuttaa RTL koodin porttitason piirikuvaukseksi. Logiikka synteesi prosessi toistetaan useasti RTL koodin kehityksen rinnalla, jotta piirille asetetut tekniset tavoitteet täytyisivät mahdollisimman nopeasti.

Tämä diplomityö suoritti logiikka synteessin NVDLA prosessori lohkolle. NVDLA on avoimeen lähdekoodiin perustuva laitteistonkiihdytin, joka on tarkoitettu suorittamaan koneoppimisalgoritmeja. Tämä tekee siitä hyvin tehokkaan prosessori lohkon. Jokainen NVDLA:n sisäinen osa voidaan konfiguroida itsenäisesti, mikä tekee siitä joustavan ja kustannustehokkaan. Sen ohjelmisto ekosysteemi kattaa suuren määrän ohjelmisto-ominaisuuksia. NVDLA on jaettu viiteen osaan niiden toiminnallisuuden perusteella. Jokainen osa on itsenäinen ylätasen synteesi hierarkia.

Tämän diplomityön tavoitteena oli kehittää logiikka synteesi vuo NVDLA prosessori lohkolle kohde yrityksen suunnitteluympäristössä. Tämä saavutettiin hyödyntämällä NVDLA suunnitteluympäristöä, yrityksen sisäistä muistikärettä, ja Synopsyksen Design Compiler ja IC Compiler 2 työkaluja. Näiden avulla logiikka synteesi suoritettiin TSMC:n 7 nanometrin standardi soluteknologialle. Käytetyt RTL koodit ja skriptit ladattiin NVDLA GitHub verkkosivulta. Muistikääre luotiin yrityksen sisäisellä muistikääre työkalulla. Se yhdistää NVDLA lohkon RAM muisteihin. Design Compiler työkalua käytettiin ensimmäisen piirikuvauksen muodostamiseen. IC Compiler 2 työkalua käytettiin yksilöllisten pohjapiirrosten luomiseen jokaiselle osalle. Luotua DEF tiedosta hyödynnettiin toisella synteesi kerralla lopullisten synteesi tulosten saamiseksi. Tulokset osoittavat, että yrityksen suunnitteluympäristöä voidaan käyttää synteesi tulosten tuottamiseen avoimen lähdekoodin prosessori lohkoille. Näin ollen, kehitettyä vuota voidaan hyödyntää alustana erilaisten avoimen lähdekoodin prosessori lohkojen vertailuun teollisessa kehitysympäristössä. Sen avulla voidaan luoda synteesi tulokset 7 nanometrin standardi soluteknologialle nopeasti.

Avainsanat: Järjestelmäpiiri, ASIC, Logiikka synteesi, NVDLA, STA

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

PREFACE

This thesis work was done in the SoC organization of Nokia during 2021 and early 2022.

First, I would like to thank my supervisor Juha Lias for the opportunity to do the thesis at Nokia. I want to thank Prof. Timo Hämäläinen and M.Sc. Saman Payvar for their valuable feedback during the writing process. Especially I want to thank my colleague Ajay Ganesha for the excellent mentoring throughout the thesis project. Many thanks also to my colleagues Syed Gillani and Vishnu Shibu for their support in the project work.

Finally, I want to thank my family and friends for supporting me during my University studies.

Tampere, 12th May 2022

Aarno Lindqvist

TABLE OF CONTENTS

1. INTRODUCTION	1
2. SYSTEM ON A CHIP DESIGN	4
2.1 SoC	4
2.2 ASIC	5
2.2.1 Abstraction levels	7
2.3 Logic synthesis	9
2.3.1 EDA tool limitations	9
2.3.2 Logic synthesis flow	10
2.3.2.1 RT-level synthesis	11
2.3.2.2 Module generator	12
2.3.2.3 Gate-level synthesis	13
2.3.2.4 Cell-level synthesis	14
2.3.3 Efficient use of synthesis tool	16
2.3.4 Timing considerations	16
2.3.5 Static timing analysis (STA)	21
2.4 Design Compiler (DC) synthesis flow	22
2.5 Synthesis in practise with DC	26
2.5.1 DC commands used during synthesis	27
2.6 Design optimization using DC	28
2.6.1 Design rule constraints (DRC)	30
2.6.2 Optimization constraints	30
2.6.3 Design optimization strategies	31
3. LITERATURE REVIEW	34
4. NVDLA IMPLEMENTATION FLOW	39
4.1 NVDLA	39
4.1.1 Hardware architecture	43
4.1.2 Software design	47
4.2 Implementation flow	50
4.3 Detailed system description	55
4.3.1 Partition_m	56
4.3.2 Partition_o	56
4.3.3 Partition_p	57
4.3.4 Partition_a	58
4.3.5 Partition_c	59
4.3.6 Memory wrapper	59
5. SYNTHESIS RESULTS	61
5.1 Power	61
5.2 Area	64
5.3 Timing	67
5.4 Comparison to literature	72
5.5 Analysis of the developed synthesis flow	73
6. CONCLUSIONS	74
REFERENCES	77

LIST OF FIGURES

Figure 1.	<i>Basic ASIC design flow [21].</i>	2
Figure 2.	<i>Basic design elements of a SoC [1].</i>	5
Figure 3.	<i>Digital ICs [2].</i>	6
Figure 4.	<i>Logic synthesis flow [7].</i>	11
Figure 5.	<i>Modest hypothetical standard-cell library for ASIC technology [7].</i>	15
Figure 6.	<i>Synthesis iterations in an area-delay space [7].</i>	20
Figure 7.	<i>A basic synthesis flow. [9].</i>	23
Figure 8.	<i>Two different NVDLA systems [6].</i>	41
Figure 9.	<i>Headless NVDLA architecture [6].</i>	44
Figure 10.	<i>NVDLA system software dataflow [6].</i>	48
Figure 11.	<i>Visualization of portability layers in the NVDLA system [6].</i>	50
Figure 12.	<i>The directory structure for logic synthesis [6].</i>	51
Figure 13.	<i>Output files of a synthesis run [6].</i>	53
Figure 14.	<i>The floorplan exploration flow from DC to ICC 2 [9].</i>	54
Figure 15.	<i>NVDLA top-level structure [10, 28].</i>	55
Figure 16.	<i>Layout picture of partition m.</i>	56
Figure 17.	<i>Layout picture of partition o.</i>	56
Figure 18.	<i>Layout picture of partition p.</i>	57
Figure 19.	<i>Layout picture of partition a.</i>	58
Figure 20.	<i>Layout picture of partition c.</i>	59
Figure 21.	<i>Memory wrapper.</i>	60

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial intelligence
AMBA	Advanced microcontroller bus architecture
API	Application programming interface
ARM	Advanced RISC machines
ASIC	Application specific integrated circuit
ASSP	Application specific standard product
AXI	Advanced extensible interface
BDMA	Bridge direct memory access
CAD	Computer aided design
CDP	Cross-channel data processor
CE	Communication element
CLPD	Complex programmable logic device
CNN	Convolutional neural network
CPU	Central processing unit
CSB	Configuration space bus
CTS	Clock tree synthesis
DBBIF	Data backbone interface
DC	Design compiler
DFT	Design for test
DL	Deep learning
DLA	Deep learning accelerator
DNN	Deep neural network
DRAM	Dynamic random-access memory
DRC	Design rule constraint
DSP	Digital signal processor
EDA	Electronic design automation
FIFO	First in first out
FPGA	Field programmable gate array
RTOS	Real time operating system kernel
HDL	Hardware description language
I/O	Input/Output
I2C	Inter integrated circuit
IC	Integrated circuit
IoT	Internet of things
IP	Intellectual property
IRQ	External interrupt interface
KMD	Kernel mode-driver
LCD	Liquid-crystal display
LRN	Local response normalization
MAC	Multiply-accumulate
NLR	No local reuse
NoC	Network on chip
NRE	Non-recurring engineering
NVDLA	Nvidia Neep learning accelerator
OCP	Open core protocol
PCI	Peripheral component interconnect
PDP	Planar data processor
PE	Processing element
PLD	Programmable logic devices
PreLU	Parametric rectified linear unit
RAM	Random access memory
RC	Resistor-capacitor

ReLU	Rectified linear unit
ROM	Read only memory
RTL	Register transfer level
SATA	Serial AT attachment
SDP	Single data processor
SIMD	Single instruction, multiple data
SoC	System on a chip
SOP	Sum of products
SPI	Serial peripheral interface
SRAM	Static random-access memory
STA	Static timing analysis
TCL	Tool command language
TSCM	Taiwan semiconductor manufacturing company
UMD	User mode-driver
USB	Universal serial bus
VHDL	Very high-speed hardware description language

<i>V</i>	Voltage
<i>nm</i>	nanometre
<i>mm</i>	millimetre
<i>mW</i>	milliwatt

1. INTRODUCTION

The target of the thesis project is to develop a logic synthesis flow for NVDLA in the company design environment. The flow can be used to review different kind of IP's quickly. The idea is to utilize both publicly available and internal scripts and RTL designs as much as possible to develop the flow. This will be achieved by utilizing NVDLA design environment, company internal memory wrapper, and Synopsys Design Compiler and IC Compiler 2 tools to run logic synthesis for TSMC 7 nm standard cell technology. All the needed codes and scripts can be downloaded from NVDLA GitHub webpage. The company memory wrapper tool is utilized to generate a memory wrapper which is able to connect the NVDLA design with needed RAM instances. The Design Compiler is used to generate the initial netlists for NVDLA partitions. To improve the results a floorplan will be created for each partition with IC Compiler 2 tool. The generated DEF file is then used for second pass synthesis to obtain the final synthesis results.

The challenge in developing the flow is to integrate all the needed parts together: the NVDLA design, the memory wrapper and the memory instances with the synthesis scripts. Also, meeting the timing constraints is an issue which requires synthesis setup and floorplan explorations. As mentioned earlier the RTL codes of the NVDLA design and the reference synthesis scripts were readily available at the GitHub webpage. The principle of re-use was exploited. However, the scripts needs to be modified to be able to use them in this project. The memory wrapper will be created for this project. Memory instances has been acquired by the company from a vendor and will be utilized in this project. Each synthesis partitions need a floorplan; they will be created manually in this project. By implementing the above tasks, the result is a flow that can be used to evaluate different IP blocks for 7 nm standard cell technology. Further by optimizing the developed flow the synthesis results of NVDLA can be improved.

First, the ASIC design flow will be explained to get an idea about the field of the thesis project. The basic ASIC design flow is shown in Figure 1.

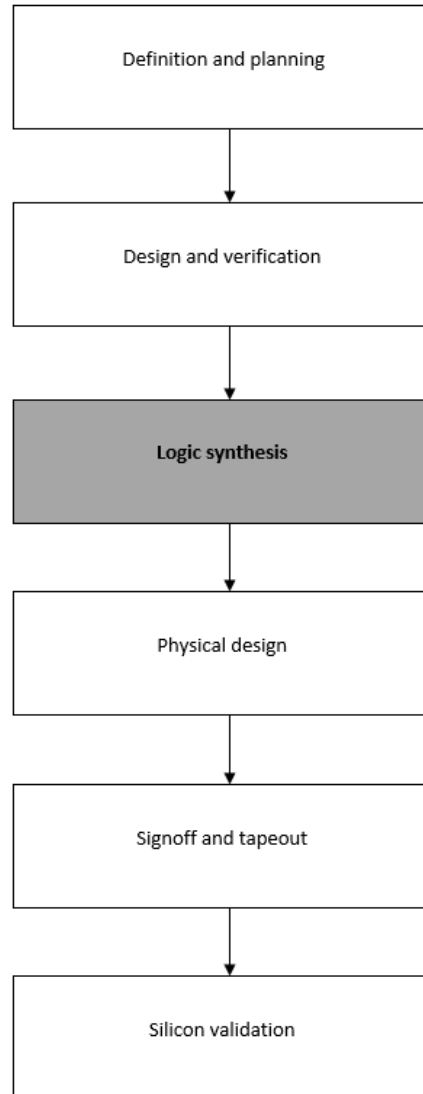


Figure 1. Basic ASIC design flow [21].

The focus area of this thesis project is the logic synthesis. It is important part of the ASIC design flow and is highlighted with grey background colour in Figure 1. The ASIC design process starts from marketing research and architecture specification based on the market requirements. Then the RTL code is designed and verified. The logic synthesis step transforms the RTL code into netlist which can be then processed to the form of real circuit in the physical design. After that the circuit is finalised and send to vendor for fabrication in the signoff and tapeout step. When the produced chip comes back from vendor, it is tested in silicon validation. After that the chip should be completed product.

Thesis is organized into 6 main sections 1. Introduction, 2. System on a Chip Design, 3. Literature review, 4. NVDLA Implementation flow, 5. Synthesis results, and 6. Conclusion. Section 2. contains theoretical background of System on Chip, ASICs, logic synthesis flow and related timing, Design compiler synthesis flow, Synthesis in practice

with DC, and design optimization with DC. Section 3. contains review of the related literature. Section 4. contains discussions about the NVDLA hardware architecture and software design, the implementation flow, and detailed system description. Section 5. present the synthesis results from power, area, and timing views. It also compares the results to literature and analyses the developed synthesis flow. Finally, section 6. concludes the thesis and discusses about the future works.

2. SYSTEM ON A CHIP DESIGN

On the highest level this thesis is related to the system on a chip (SoC) design, which is implemented by utilizing ASIC technology. Modern SoCs are basically all around us in every day devices. Therefore, SoCs are studied first. After that ASIC technology is introduced and then the logic synthesis process is discussed in detail. Logic synthesis is the actual topic of this thesis project.

2.1 SoC

A SoC is set of processors, memories, and interconnections [3]. It is a computer system which is designed to implement an application specific functionality for some given application domain [3]. Most of the SoC designs are executed on ASIC or FPGA devices. The huge demand on markets like consumer electronics, automobiles and telecommunication has led to a need for more and more advanced electronic devices. At the same time the development of IC technology on the past 40 years has given the solution to that demand, thus the empirical law of Moore [2,4]. Historically, the concept of a computer has been a single processor combined with a memory on a board [3]. On modern IC chips there can be up to 10 billion transistors which allows the integration of complete complex systems on a chip [2]. Usually, a SoC contains signal processing elements, hardware accelerator, microprocessor or microcontroller cores and memories [2]. The function of signal processor or hardware accelerator is to take care of heavy computing whereas the microprocessor or controller takes care of the process and some low performance computing [2]. The memories store the software codes and data. In most of the cases there is also on-chip analogue interface, processing circuits for pre- and post-processing and many wireless interfaces [2]. The development of complex systems like the SoC might require working time of thousands of man-years [2]. Commonly, used design elements of a SoC are shown in Figure 2.

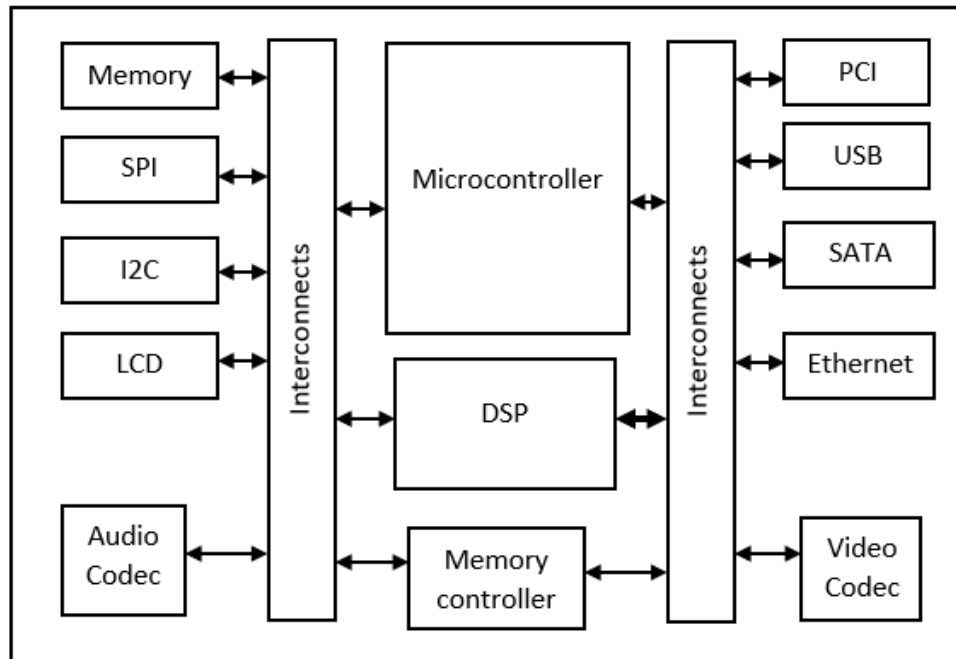


Figure 2. Basic design elements of a SoC [1].

The difference between general-purpose computer on a board and SoC is the design target. When designing a SoC, the specific application is known so all the components of the system can be optimized for that purpose. Highlighting the customisation is what distinguishes a system architecture from a computer architecture. [3, 25].

2.2 ASIC

ASIC is abbreviation for Application Specific Integrated Circuit. Integrated circuits (IC) are electronic circuits that are made of silicon wafer. A silicon wafer can consist of up to thousands of dies. One ASIC die can have few billion transistors. As the name implies ASICs are integrated circuits that are designed for some specific application. ASICs are tailored directly for the customers use case with specific requirements. ASICs are used in real-time computing and digital signal processing systems that require high computing performance and low power consumption. Here is some market segments and applications where ASICs could be used: Tire pressure monitor for automotive industry, 5G radio for telecommunication industry, Patient monitoring for medical industry, UHD TV for display industry, Smart phone for digital consumer industry, Robotics for manufacturing industry and Radar processing for military industry. [1,2].

The class of PLDs (Programmable Logic Devices) include FPGAs (Field-Programmable Gate Arrays) and CLPDs (Complex Programmable Logic Devices). PLD chips contain programmable logic elements and wires which connects all the logic elements together. Semi-custom ASICs are ICs that contains some finished metal layers and contact and

some custom designed metal layers and contact, thus the time-to-market is shorter compared to custom ASICs. One example of semi-custom ASIC is a gate array. Custom ASICs are the most tailored ICs. In these all the metal layers are specially designed for a client's application. Custom ASIC can be also design by using IP (Intellectual Property) cores. The purpose of using IP cores is to reuse commonly used part of IC design that are already finished to make the time spend to the design process shorter. [2].

IP cores can be represented in three forms soft cores, firm cores, and hard cores. Soft cores are synthesisable HDL codes. Soft cores are flexible, but their performance is not that predictable because they are not yet fully tested and optimized. Firm cores are already optimised for area and performance. Hard cores are finished parts of the chip that execute some functionality. They are optimised for performance, size and power and they are mapped to some specific technology. Microprocessors like Intel Itanium and ARM are examples of IPs also FPGA-based accelerators like decoders and encoders are IPs. NVDLA is a soft IP core. [2].

In the industry most popular digital IC technologies are FPGAs and cell-based custom ASICs. Usually, big custom ASICs are design by using standard cells, macro-cells, and IP cores. These can be designed in-house, or they can be purchased from an external supplier. Figure 3 visualizes the most relevant digital IC technologies. There are also other digital IC technologies that are not presented in Figure 3.

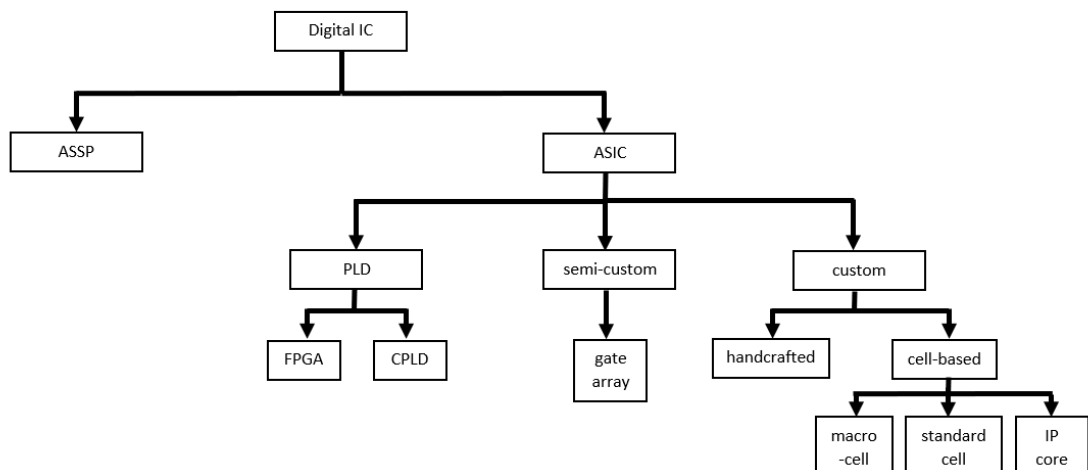


Figure 3. Digital ICs [2].

The most common ways to implement NVDLA would be to use FPGA or cell-based technologies. This thesis project focuses on standard-cell ASIC's, for that reason let's study them little bit more. In standard-cell ASIC technology, a circuit is created from a set of predefined logic components referred as standard cells. The cells are ready-made

components whose functionality and layouts are validated and tested. The benefit of standard-cell ASIC technology is significantly simplified development process. It enables the work to be done at the gate level in preference to the transistor level. The building blocks are normally delivered by the device manufacturer as a library of standard cells. Usually, the library contains basic logic gates, such as AND, OR, and NAND gates, combinational components, such as 2-to-1 multiplexer and 1-bit full adder, and memory elements, such as latch and D flip-flop. Some of these libraries might also include blocks with more sophisticated functioning, such as an adder, and random-access memory (RAM). [7, 19].

2.2.1 Abstraction levels

When the number of transistors in the chip rises to hundreds of millions, a human or a computer cannot deal with the quantity of information directly. The complexity needs to be managed; therefore, a system is characterized with abstraction levels. Single abstraction level shows only the chosen elements of the system and pays no attention to the related details. Hence, the amount of information is reduced to show only the decisive information i.e., it scales down the system to a more manageable level. High abstraction level contains only vitally important information about the system, while low abstraction level contains profound information which was previously ignored. The low abstraction level is complex but models the system at the real circuit level. Usually, the development process flow from high abstraction level towards the low abstraction level. [7].

A digital system is divided into 4 abstraction levels:

1. Transistor level
2. Gate level
3. Register transfer level (RTL)
4. Processor level

Different representations of the abstraction levels do exist such as division into 6 abstraction levels [2]. The division is made by the size of the elementary building components, starting from the smallest, these are the transistors, logic gates, functional units, and processors. Another dimension of the system are the views [7]. Next, the 4 abstraction levels are studied independently.

The *transistor-level abstraction* is the lowest. The basic electrical components are utilized at this level such as, resistors, inductors, and transistors. The behaviour is modelled with differential equations or with voltage diagrams. Input-output characteristics can be examined with analog simulation software tools. At this level, a digital circuit behaves as

an analog system i.e., the signals are time dependent and has continuous value. The final physical layout of components and interconnects are constructed at the transistor level. It is the result of the design process. [7].

The *gate-level abstraction* is one level above the transistor level. The basic logic gates are utilized at this level for example, AND, OR, and memory components, such as, flip-flop. Signals are described as logic 1 or logic 0 depending on the set voltage threshold. System with only two values can be presented by Boolean algebra equations thus, complicated differential equations are not needed anymore. Practically, the abstraction changes a continuous system to a discrete system. It is good to notice that the signal is still continuous, it is just interpreted with a pre-set voltage threshold. The timing information is also easier to interpret by just using the *propagation delay* which is the time needed to generate a valid output value. At this level the physical view expresses the placement and the routing of the logic gates and the wires. Also, the number of gates in a system can be counted, it is called as *gate count*. The gate count tells the area of a circuit based on the area of two-input NAND gate. With this method the equivalent gate count is independent from the device technology. [7]. The synthesis results for NVDLA are presented on this abstraction level.

The next abstraction level is the *register-transfer-level abstraction*. At this level logic gates form components that are used to create functional units such as, adders, registers, and multiplexers. The RT level is more abstract than the lower levels. Signals are interpreted with a particular data type for example as an unsigned integer. The behaviour of data is described with finite state machines (FSM). An elementary property at this level is that the storage elements utilize common *clock signal*. The clock signal is used to synchronize the data input into the storage elements at the rising or falling edge of the clock signal. In an appropriate system, all the data signals should stabilize during a clock period. Since, the timing can be examined in clock cycles, the variations in propagation delays and signal glitches has no effect to the functioning and can be disregarded. At this level the physical features are presented as the floorplan. The floorplan is useful tool, when defining the clock period because it visualizes the longest path. [7].

Finally, the highest abstraction level is the *processor-level abstraction*. The intellectual property (IP) blocks such as, processors, memory, and buses are utilized at this level. The behaviour is expressed as computation steps and communication processes. The signals are formed to a set and specified with different data types. The computation steps forms the time measure. Multiple computations can be executed in parallel, and the data

is changed between components with communication protocol. At this level, the physical layout is expressed as floorplan. Naturally, with larger components than on RT level. [7].

2.3 Logic synthesis

In the ASIC design flow logic synthesis step is the process which converts an RTL code into a technology specific gate level netlist [1]. The output can be also referred to as cell-level netlist [7]. The synthesis step can be executed after the verification of the RTL code is done and the design met the coverage goals. Synthesis is performed by the EDA tool which inputs are RTL code, design constraints and the standard cell library. The output of synthesis is optimized gate level netlist which is created from the basis of inputs. The most common logic synthesis tools are Synopsys Design Compiler and Cadence Genus. The synthesis tool takes into consideration power, performance, and area as the most important factors to create the gate level netlist. The goal of synthesis process is to meet specified constraints by considering costs for different implementations. Gate level netlist is structural representation of the design presented as standard cells. Gate level verification is performed for the netlist to check that the functionality of the design is correct after the synthesis. After that pre layout STA (Static timing analysis) is performed to check possible timing violations in the design. In this stage STA is executed without using the parasitic (RC) effect. The goal is to repair any setup timing violations and to enhance the total performance of the design. The hold time violations are usually repaired after the CTS (Clock tree synthesis) and routing. Finally, before physical implementation the DFT (Design for Testability) is performed for the gate level netlist. This is performed by using DFT tool and the goal is to find possible faults in the design. To make this stage more convenient the RTL should be made DFT friendly. The benefit of this is faster scan chain insertion and it enables the total fault coverage for the design. [1].

2.3.1 EDA tool limitations

The design process of digital circuit is not easy by any means. Completing the process involves many challenging tasks, requiring lots of data processing by complicated algorithms thus, computers are utilized to execute it. Therefore, one could ask whether the whole synthesis process can be automated. Then the engineer would only design a high-level behavioral model and EDA tools take care of the rest of the process i.e., the logic synthesis and placement and routing. However, the EDA tools have fundamental limitations that make this impossible on a full scale, this emerge from the theories of computational algorithms such as, the graph theory. [7].

The logic synthesis is an intractable problem and there is no polynomial-time algorithm to solve it. The logic synthesis process can be seen as a searching problem. The resulting circuit has $O(2^n)$ solutions thus, the optimal solution is a result of exhaustive global search. That is why, in actual synthesis tools, the search space is limited to local search and clever tactics and heuristics are utilized to lead the search into desired direction. The HDL code sets the starting point for the search hence, a good HDL code is essential. Since, the local search might not get an efficient solution from bad starting point. [7].

Synthesis and also other design steps comprise computationally hard problems. EDA tools have this theoretical limitation by nature. Heuristic algorithms can find desired solutions but may not suit for all types of inputs. These limitation will also remain in the future and therefore a design engineer's expertise will be needed. [7].

2.3.2 Logic synthesis flow

Logic synthesis is the process where RTL code is realized into RTL descriptions by utilizing elementary logic cells from the target technology library. The process is usually divided into steps to make it more comprehensible. These are RT-level synthesis, gate-level synthesis, and cell-level synthesis or technology mapping. As a first step, this list could include high-level synthesis where an algorithm is transformed into a system architecture containing of systems data path and control path which could be further transformed into HDL code. However, it is essentially different compared to other steps and it is performed by different software tools, which are not of interest to this thesis. Thus, it will not be discussed further in this thesis, although it is an important area of research today. [7].

The synthesis flow is visualized in Figure 4. The final circuit is formed level by level. It is an iterative process where first an RT-level netlist is transformed and optimized to a gate-level netlist which is finally transformed and optimized to a cell-level netlist. Complex RT-level components are generally processed with a module generator, these components include for example adder and comparator. [7].

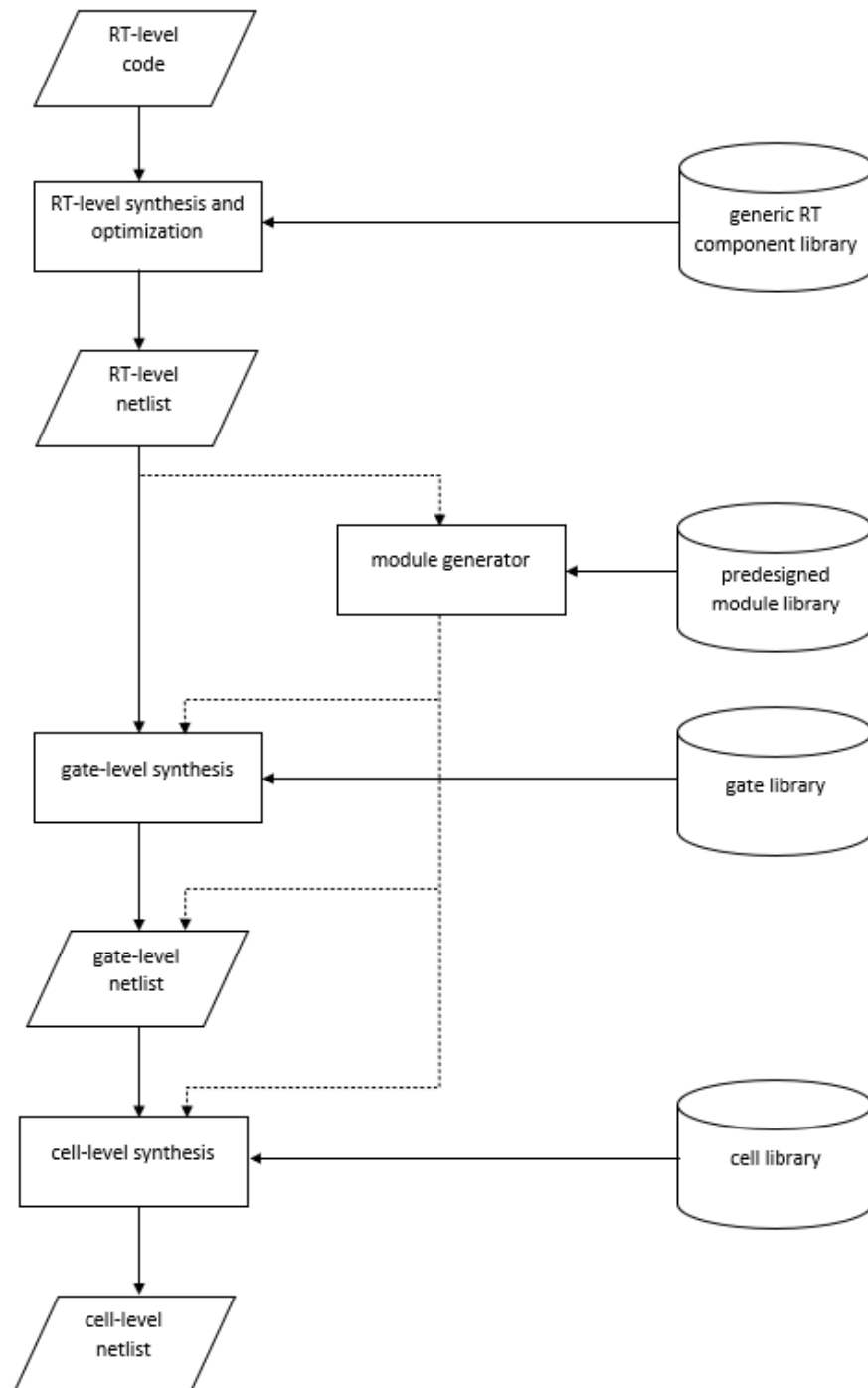


Figure 4. Logic synthesis flow [7].

2.3.2.1 RT-level synthesis

In *RT-level synthesis* a behavioral RTL description is transformed into a circuit build from components provided by a *generic RT-level library*. The generic RT component library contains technology independent components. Thus, they are universal to all technologies. Generally, the components are categorized into three classes: functional units, routing units, and storage units. *Functional units* will be utilized to produce the

logic, relational, and arithmetic operators confronted in RTL code. *Routing units* are different kind of multiplexers utilized to build the routing structure of an RTL description. *Storage units* refers to registers and latches, which are utilized in sequential circuits to storage data at intermediate stages of RTL logic. [7].

An *RT-level netlist* is derivated and optimized during RT-level synthesis. This is the process where RTL statements are transformed into equivalent structural implementations. The usage of optimization techniques may decrease the complexity of circuit and improve performance. General optimization techniques include operator sharing, common code elimination and constant propagation. The scope of RT-level optimization is highly restricted compared to gate- and cell-level synthesis. Thus, it is only executed temporarily. At this stage the importance of good design is highlighted because it may amend the RT-level structure significantly. Thus, software tools can derive better implementation. [7].

2.3.2.2 Module generator

The RT-level synthesis creates a netlist of generic RT-level components from the initial RTL descriptions. These components have to be amended into lower abstraction level, to process them further towards the final circuit. RT-level components include logical operator, multiplexers, adder, subtractor, incrementor, decrementor, comparator, shifter, and multiplier. Some of the components are uncomplicated and can be mapped straight into gate-level implementation, these are called as *random logic* component. Typically, these embodies less regularity and enables optimization subsequently in gate-level synthesis phase. Examples of random logic components are logical operators and multiplexers. The other components are referred as *regular logic*. A *module generator* is used for these more complex components. It is a specialised software which is able to derive the components for the gate-level implementation phase. Typically, regular logic has some kind of recurring structure and is designed in advance. As shown in the Figure 4. a module generator has capabilities to generate modules in various abstraction levels: Gate-level behavioral description, presynthesized gate-level netlist, presynthesized cell-level netlist. [7].

A gate-level behavioral description presents RTL code in a way where it utilizes only simple signal assignment and logical operators. This form allows it to be mapped to a gate-level netlist without much effort. At gate-level the description is general and technology independent. A single *gate-level netlist* is created by flattening the description and merging it with the random logic. The netlist combination is synthesized as a whole in *gate-level synthesis*. [7].

As stated earlier the regular logic components has the regular and repetitive features which allow further explorations of their properties and even manual derivation and synthesis of the netlist at the gate- or even cell-level. Thus, the regular logic and the random logic are dealt with separately. This approach might result a more efficient implementation compared to gate-level synthesis. In this process a presynthesized gate- or cell-level netlist is neither flattened nor merged with the random logic. Gate-level synthesis and even cell-level synthesis is performed separately for the random logic. Once these two separate netlists have been processed, they can be combined. [7].

The non-flattened approach has two benefits. Firstly, it may use highly optimized modules because they can be designed in advance. Secondly, the modules are separated from the other logic modules meaning that the remaining part is smaller requiring less effort to process and optimize it. However, the weakness of this approach is decreased possibilities for additional optimization due to isolation of the non-flattened modules and the random logic. There is no clear rule whether the flattened or the non-flattened approach is more effective. Users can enable or disable this feature in some of the synthesis tools. [7].

2.3.2.3 Gate-level synthesis

The gate-level synthesis process produces elementary gate-level components, for example a NOT-, AND-, NAND-, OR-, and NOR gates. The number of the components is optimized forming the so-called structural view of a design. The generic technology independent components do not possess specific information about the size or the propagation delay. These components are the operators of Boolean algebra. Hence, a design can be presented as a Boolean function. Gate-level synthesis may be performed in two different ways: two-level synthesis and multilevel synthesis. [7].

An example of two-level synthesis is the sum-of-products expression. In that the logic is formed from AND gates and OR gates. Where AND gates are at the first level and OR gates at the second level. Any two-level logic construction may be created with the sum-of-products expression. Two-level synthesis is thus intended to find an optimal sum-of-product expression for Boolean function. This is achieved by minimizing the number of AND gates and the total number of fan-ins to these AND gates. For a maximum of about five input circuits this is done manually by using the Karnaugh map technique. In real applications circuits may have hundreds of inputs, as a consequence optimization cannot be done manually. Thus, it is an intractable problem. However, good algorithms have been developed to create suboptimal but still efficient circuits. Two-level synthesis is good at processing and manipulating logic statements. It scales down the amount of

information required to represent a function and therefore could be used as a preliminary step for multilevel processing. [7].

Multilevel synthesis uses multiple gate levels to represent a Boolean function. With multiple levels the process is not so strict, and it gains more freedom, resulting more efficient and flexible outcome. The design may be explored from area or delay optimization perspective or even from an optimal trade-off point for both the area and delay. Compared to two-level circuit it lowers the number of gates as well as the number of fan-ins. The foundation of contemporary application technology is in small cells with a constrained fan-in count. Making it more suitable for multilevel synthesis. [7].

Processing and optimizing a multilevel logic requires more effort. Usually, it is performed with heuristic methods by following a database of circuit rules. Due to the limited number of restrictions multilevel synthesis may generate substantially different results. A very small change in original description might lead to an entirely different implementation. [7].

2.3.2.4 Cell-level synthesis

Gate-level synthesis process produces an optimized netlist of generic components. *Cell-level synthesis* also known as *technology mapping* transforms the gate-level netlist into technology dependent cell-level netlist. To achieve this the process uses the target technology library. The resulting components are usually called as *cells*. Generally, in ASIC technology a semiconductor vendor delivers the technology library and fabricates the device. A cell is defined by its function and by a set of physical parameters including, for example area, delay, and input and output capacitance load, each cell is also linked to the physical layout. Cell-level synthesis can be performed simply by just transforming generic components into logic cells without any further explorations. However, the generated circuit will not be very efficient. Hence, the functionalities, areas and delays of the cells are worth to exploit. The cell-level synthesis is a hard process, involving intractable problems. Again, heuristic algorithms are applied to find solution. Next, an example on a standard-cell library is discussed. [7].

A *standard-cell technology library* includes hundreds of cells, such as combinational, sequential and interface cells. Some examples of combinational cells are AND-, OR-, NAND-, NOR-, and XOR gates. Also, more complex circuits might be included, like 1-bit full adder and 1-bit 2-to-1 multiplexer. Example of standard-cell library is visualized in Figure 5. The library in the Figure 5. was invented for a sake of example and does not represent any real library. [7].

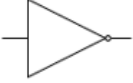
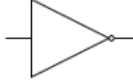
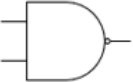
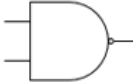
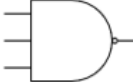
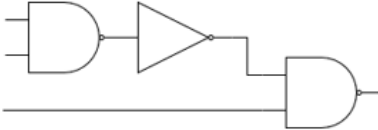
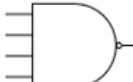
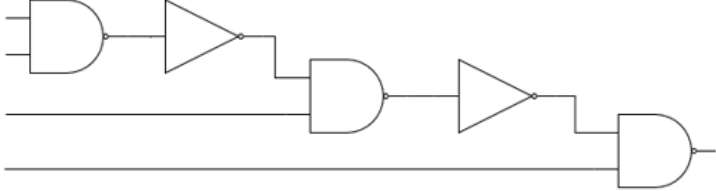

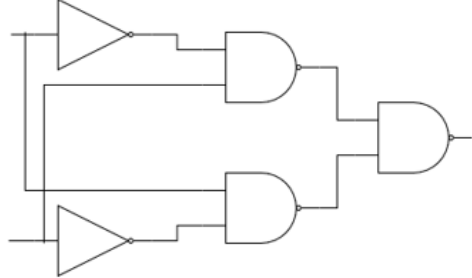
Cell name (Cost)	Symbol	NAND-NOT Representation
NOT (2)		
NAND2 (3)		
NAND3 (4)		
NAND4 (5)		
XOR (4)		

Figure 5. Modest hypothetical standard-cell library for ASIC technology [7].

The columns represent the cell name and the relative area cost of that cell, its logic gate symbol, and the NAND-NOT representation of that cell. In NAND-NOT representation cells are implemented by utilizing 2-input NAND gates and NOT gates, this simplifies the cell-level synthesis process. [7].

A technology library contains cells which are optimized and fine-tuned for specific technology. This is done by manually designing each cell from the transistor level in preference to simple logic gate level. An example of this is the XOR gate, if it would be implemented with NAND- and NOT gates the area would be 13, which is over 4 times the area of the NAND2 cell. While, with the transistor level implementation the area of the XOR gate is only 4, which is 2 times the area of the NAND2 cell. Hence, the standard-cell library contains various elementary cells. Additionally, a single logic function may have several cells implementing different area-delay trade-offs. [7].

To summarize, the cell-level synthesis or the mapping is an iterative process, where the initial mapping is a one-to-one gate-to-cell translation, it is not the most optimal solution. To make the solution better, NAND-NOT representations are replaced by corresponding more optimal logic gates. Finally, to find the most optimal area-delay trade-off different kind of transistor level implementations are explored. Thus, cell-level synthesis plays a major role in the logic synthesis process, while its complexity is very high. [7].

2.3.3 Efficient use of synthesis tool

Although, synthesis software has some limitations, it is a vigorous and essential tool. It automates various design tasks and executes intractable and recurring computations. Designers should be aware of the features and constraints of software, to be able to adjust it and make compromises when needed. [7].

In general, the performance of logic synthesis tool is highly effective for circuit sized around 2 000 000 instances, this is achieved by studies of algorithms. Even small design includes hundreds of gates, making the use of synthesis tool practical. The synthesis is straightforward from logic operators to gate-level components. Because the mapping can be done directly at this level, there is no need to be concerned about the sharing and optimization of logical operators in an RTL code description. [7].

The RT level optimization includes complicated arithmetic and dependent operators and routing structure. These demand human handling, to determine the wanted design in an RTL description. Recurring small improvements on code can enhance circuit efficiency essentially. Thus, designer's insight and expertise of a circuit has great importance to the end result. [7].

The "data" flow from circuit inputs to outputs through the system demonstrates *routing structure*, which indirectly specifies the layout of the physical circuit. The data flow is described in the RTL code, which forms the initial layout. However, this will be reshaped by the placement and routing process to realize the circuit on a two-dimensional silicon chip. Hence, the resulting implementation will be smaller and faster if the RTL code resembles the structure of the silicon chip. This is due to the fact that synthesis tool cannot make significant global changes. The RTL coding technique creates a basis for synthesis, affecting the result much more than the optimization done by the synthesis tool. [7].

2.3.4 Timing considerations

Even though a digital circuit is fast the respond is not instant. Meaning that the output is a function of time. The *propagation delay* represents the time between an input state

change and a valid output value. As a time-domain characteristic for a circuit, it is a major design criterion. Also, a *hazard* is time-domain phenomenon, it is the unwanted state change of an output signal. It is a transient event but might result malfunction in a poorly designed circuit. Next, subsection will discuss the propagation delay and hazard in more detail. [7].

A digital circuit requires some time to generate a valid output response to the change of an input. In digital system, the time needed to propagate a signal from input port through the circuit to output port is called as *propagation delay* or just *delay*. Multiple input-output delays may exist due to the large number of ports, but propagation delay usually refers to the worst-case delay i.e., it indicates how fast a digital system can operate. Hence, it symbolizes the *performance* or the *speed* of the digital system. For a digital system the two most important design metrics are the speed and area i.e., the circuit size. The system propagation delay can be computed by first figuring out the delays of individual components and by searching all the input-output paths. Then we can add up the delays from all the components for each path and with that define the propagation delay. Now we can notice that the system delay is based on its components and depends on the information they possess. Hence, the most accurate delay estimate will be obtained at the cell level. The netlist contains the detailed physical and electrical information of the cells. In contrast, the RT level offers least information about the physical and the electrical characteristics since at that level the components require further processing. [7].

In order to characterize the propagation delay, the cell level time-domain behavior is studied and analysed at the transistor level, where the modelling is based on transistors, resistors, and capacitors functioning. Main reason for the delay is parasitic capacitance, coming from two overlapping metal layers. A state change in a transistor makes capacitors to charge or discharge which causes the delay. Performing the cell analysis at this level is very complex thus it is applied only at a small scale. Consequently, it gives basic information for the modelling. A simplified linear model is common approach to perform timing analysis at the cell level. [7]. Also, nonlinear delay model, polynomial delay model, and current source delay model can be used [20]. The simplified linear model is unsophisticated model but with it the cell level complexity can be managed. In this model, all parasitic capacitances are summed and represented as one capacitor, which in turn allows consideration of only first-order effects. The delay can be represented by an equation [7].

$$\text{delay} = d_{intrinsic} + r \times C_{load}, \quad (2)$$

The circuit inside the cell is expressed by the first term, $d_{intrinsic}$, it models the transistor state changing time for example changing from off state to on state. The cell drives external circuit, which is expressed by the second term in the equation. The parameter C_{load} relates to the path from output of the current cell to the input of driven cell, it is the total capacitive load. It is defined as the sum of parasitic capacitance from interconnected wires and the input capacitance from driven cells. The parameter r models the output impedance, which expresses the driving power of the cell. High driving power can be achieved with small impedance, since it enables higher current. Thus, the capacitive load is charged or discharged in a shorter time. Consequently, with small r value the delay is shorter. Accordingly, bigger transistor shortens the delay. [7].

The cell-level delay calculation are dependent on the accuracy of few factors such as the used parameters and the model. The values of parameters $d_{intrinsic}$, r , and input capacitance are defined by the manufacturer and can be considered reasonably accurate. The total input capacitive load can be computed from the netlist, after technology mapping has been completed and fan-out of all the cells are available. Whereas, the wire capacitance is determined by the real length and location of wires, which are not specified at the synthesis process. However, synthesis tools can generate a rough estimate based on a statistical model. The wire capacitance can be determined accurately once place and route step is performed. At cell-level, wiring has a great effect on delay estimation. The linear cell-level model disregards higher-order effects in circuit, which might have an impact to the accuracy of the model. Hence, the impact on the functioning of the circuit may be significant. However, it is possible to use more advanced models. For example, a simple lumped RC model can be replaced with more advanced RC circuits to get more accurate estimation. [7].

With comparatively large transistors the above-mentioned factors have a minor impact on the total delay and can be excluded from the considerations, because of that the synthesis process can generate accurate timing data. However, on nanometre technology the given factors emerge, which complicates the design process. Accurate timing data will be received only after placement and routing. Inherently, the delay of a cell cannot be managed precisely. The delay is also impacted by the manufacturing process and operating environment. A technology vendor can only provide the boundary values, usually this means the maximal propagation delay. [7].

When delays of individual cells are known, path delays can be defined. A path delay is the sum of cell delays on the path. Usually, a digital system contains a great number of different paths, flowing from input to output ports with different delays. The *system delay*

is the longest path, referring to the worst-case scenario. Generally, it is also known as the *critical path*. [7].

To find the critical path, the netlist can be handled as a graph. Thereby all the possible paths in the netlist can be determined and the longest of them can be found quite easily. Hence the critical path is defined by the topology of the system, and it is also referred as the *topologically critical path*. Sometimes this approach may lead to overestimating the critical path due to the existence of a *false path*. It is a path that exist but according to defined logic, data cannot pass along it. The topographically critical path can be a false path and can be ignored from true system delay calculation. Therefore, both the topography and the internal logic operations must be considered, which makes the critical path determination difficult task. However, it can be determined with software. Hence, synthesis tool is used to execute cell-level timing analysis. Synthesis tool also provides a feature to manually rule out possible false paths. [7].

The same principles apply also at the RT-level and therefore the propagation delay can be already analysed and calculated at this stage. At RT-level the result of the calculation rely on the used components. This implies that if an RT-level design includes mostly simple logical operators being mainly random logic, these will experience lots of transformations and optimization in the logic synthesis phase. Meaning that the initial RT-level delay calculations does not necessarily reflect the complete synthesized circuit at all. There is also another side, if an RT-level design includes mostly complex operators and function blocks, they will have the greatest impact on delay calculations. Since, they are designed in advance and optimized, logic synthesis will not change their delay features notably. Therefore, this type of circuit could have realistic delay calculation already at the RT-level. Consequently, the critical path and thus the performance are known. This will facilitate the RTL code design and ultimately will result more efficient circuit with the preferred area-delay features. [7].

The two most important design specifications are the chip area and the system delay. Usually, the final design is trade-off between these two. In practice, a fast circuit requires more area, and a small circuit is not the most efficient. One application can be implemented with multiple different area-delay characteristics. Multilevel logic synthesis is flexible, it is able to add gates in the circuit to decrease delay. This is useful in a situation where the circuit area is optimized but the system delay still need to be reduced. Intrinsically, the optimal trade-off can only be achieved within a certain range. The area cannot become smaller and the performance better endlessly. The impact of RTL improvements to synthesis iterations is shown in Figure 6 [7].

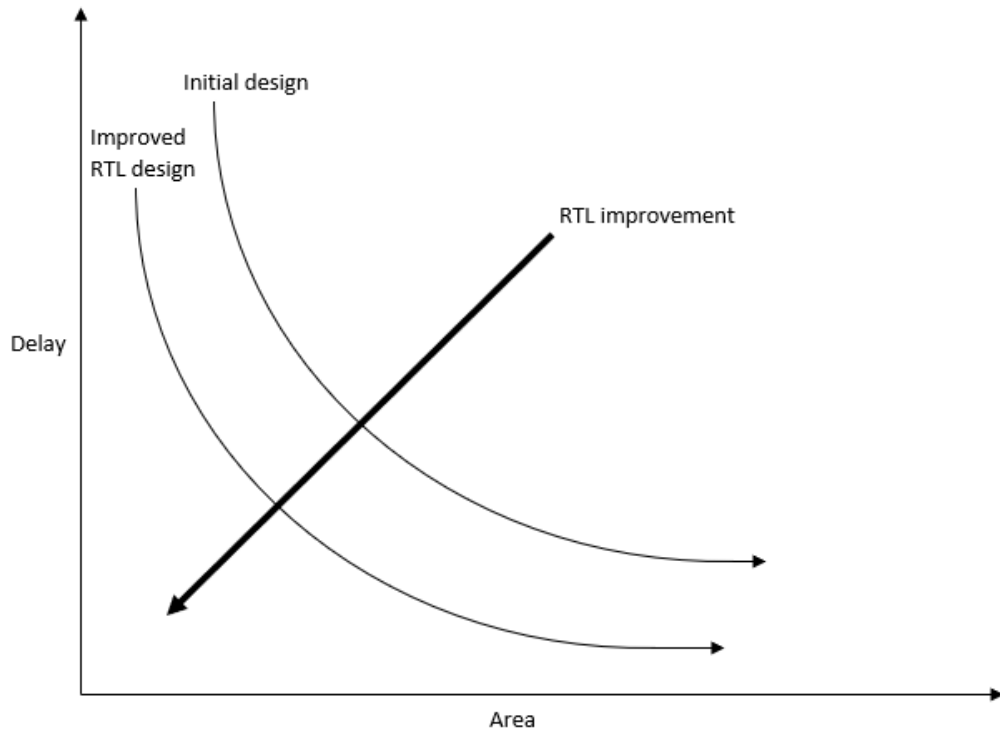


Figure 6. Synthesis iterations in an area-delay space [7].

The logic synthesis is iterative process. The process introduced in section 2.3.2 finds the optimal area for the circuit. Usually, synthesis is performed for separated IP blocks or partitions of the whole system. To achieve the performance targets, logic synthesis needs timing constraints. A circuit cannot be synthesized exactly to achieve a certain propagation delay. However, the maximal accepted threshold value for propagation delay is set in the timing constraints. As stated previously, the critical path defines the system delay. Therefore, not all paths should be blindly optimized. The synthesis processing sequence is iterative. As a first step synthesis usually generates implementation with the smallest achievable area. Next, the resulting netlist will be analysed to obtain the critical path and the system delay. If the system delay is longer than what the constraints allow, synthesis adds gates in the circuit to make the critical path faster. The regenerated implementation introduces the new critical path, which is assumed to be the second longest path of the initial implementation. The new system delay will be explored to identify if it met the timing constraints. The process iterates through until the resulting implementation is within desired constraints. [7].

The previously described processing sequence is executed at the gate or cell level and is too laborious for human but can be utilized on RT level. The schematics show the placement of complex RT-level components and the preliminary routing configuration. System delay is dominated by the complex components, thus the path through them will

indicate the critical path. The analysis is beneficial for architectural explorations and ultimately will result a more efficient circuit. The designer's knowledge about the system can induce global optimization. Naturally, this is more effective than local optimization at gate- or cell-level, performed by synthesis tool. [7].

The propagation delay indicates the needed time for a system to generate a valid output state. Oscillations at the output port within the transient period are called as *timing hazards*. In digital systems, multiple paths might lead to a single output port. Since, path's delays can vary, signals might reach the output port at different points of time. Consequently, the output port might oscillate before a steady-state is generated. The oscillations are one or more unexpected alternations at the output port, they are called as *glitches*. If a circuit is able to generate glitches it has a timing hazard. There are two types of timing hazards, the next paragraphs will describe them in more detail and explain how to handle them. [7].

A *static hazard* refers to a glitch at output port when assumed to be at a steady-state. Furthermore, it includes two types of hazards, static-1 hazard, and static-0 hazard. A *static-1 hazard* happens when Boolean algebra analysis of a circuit shows that output should be at a steady-state '1' but circuit generates a '0' glitch. Correspondingly, a *static-0 hazard* happens when analysis show that the output should be at a steady-state '0' but a '1' glitch is generated. A *dynamic hazard* refers to a glitch at the time period when a circuit is switching state for example from '0' to '1'. The source of these two hazards is too fast paths i.e., the different propagation delays on each path. [7].

2.3.5 Static timing analysis (STA)

The STA calculates time period when certain signal have to be available [11]. The goal is to check that a circuit can operate at the required frequency [11]. Commonly, a sequential circuit is utilized in a digital system, it is a circuit containing memory elements. Further, a synchronous sequential circuit has global clock signal which controls all the memories in the circuit thus, this method makes the design process simpler. The most popular memory element is D-type flip-flop, it has 3 main timing parameters: [7]

1. *clock-to-q delay*: The delay on path between the d input and the q output occurring after the edge of the clock signal.
2. *setup time*: Indicates the time period when the d signal must be stable **before** the edge of the clock signal.
3. *hold time*: Indicates the time period when the d signal must be stable **after** the edge of the clock signal.

The clock-to-q delay is about the same as the delay caused by a combinational element. Setup time and hold time are timing constraints specified for a circuit. They define the

time period around the clock edge when the d signal must be stable. A change in the d signal during the time period is known as *setup time violation* if it happens before the clock edge or *hold time violation* if it happens after the clock edge. These timing violations can create a *metastable state* for the D flip-flop, where the q output is in an unknown state and a circuit does not function correctly. [7].

The timing of a combinational circuit is mainly based on the longest path in it i.e., the propagation delay. While the timing of a sequential circuit is primarily determined by the timing constraints specified by the memory components. Where the most important timing parameter is the *maximal clock rate*, which actually contains the propagation delay, the clock-to-q delay, and the setup time constraint. The main objective in designing a sequential circuit is to meet the setup and hold time constraints. [7].

A sequential circuit containing memory elements and combinational logic has a limit for the maximal clock rate ($T_{c(max)}$) to avoid setup time violation, it is determined as

$$T_{c(max)} = T_{cq} + T_{comb(max)} + T_{setup}, \quad (3)$$

Where T_{cq} is clock-to-q delay, $T_{comb(max)}$ is the maximum propagation delay of the combinational logic, and T_{setup} is the setup time constraint. The hold time constraint (T_{hold}) has some differences to the setup time constraint. To avoid hold time violation, a sequential circuit has to satisfy the inequality

$$T_{hold} < T_{cq} + T_{comb(min)}, \quad (4)$$

Where T_{cq} is the clock-to-q delay and $T_{comb(min)}$ is the minimum propagation delay of the combinational logic. [7].

2.4 Design Compiler (DC) synthesis flow

Design compiler has its own logic synthesis flow. Usually, it is used on design exploration and design implementation phases. Flowchart of the synthesis flow is presented in Figure 7. [9].

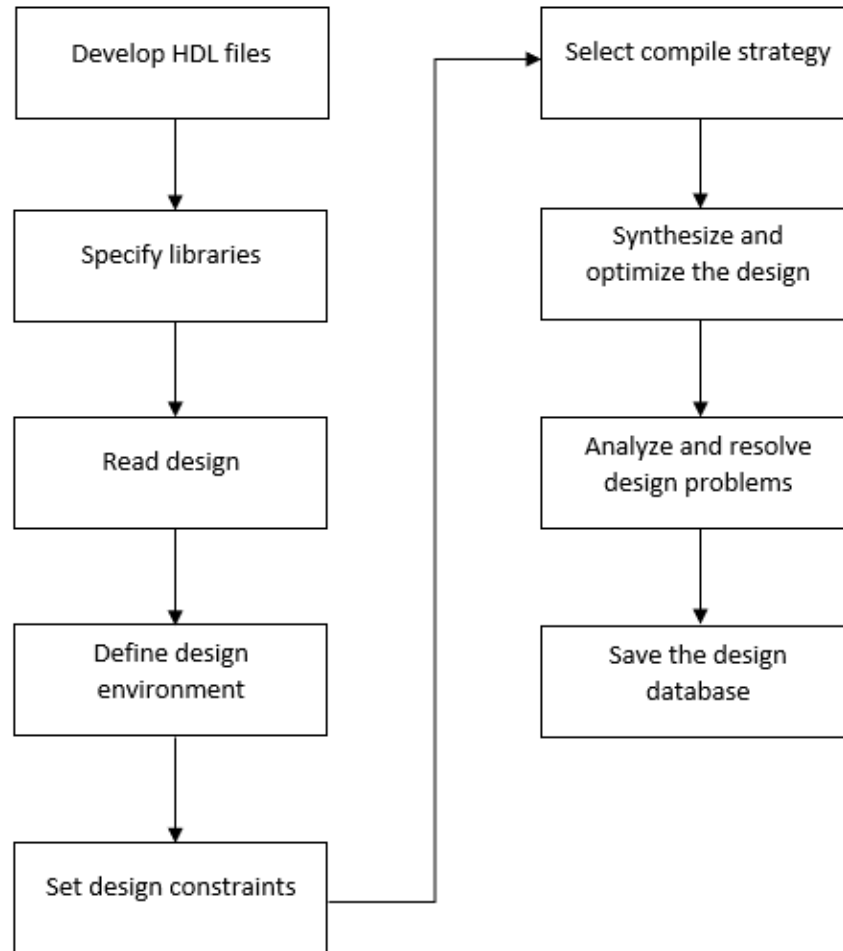


Figure 7. A basic synthesis flow. [9].

The following paragraphs discuss design compiler synthesis flow in more detail.

Develop HDL files: Hardware description language (HDL) files are the input for design compiler, these can be created by using Verilog or VHDL. In the HDL code preparation phase, you have to take to account design data management, design partitioning, and HDL coding style. Design files contain the design descriptions. Naming of these files must be unique. When preparing design for synthesis, managing design file data is important, with appropriate strategy data is not lost. Selected HDL coding strategy is also important because HDL coding is the basis for synthesis. A good method to manage the design data is to organize it systematically. Effective partitioning of a design can improve the synthesis results, decrease compilation time, and simplify the script files and constraints. Partitioning have an effect on design block size thus, block size control must be done carefully. [9].

Design cannot function correctly without constraints; thus, they are very important. They are statements that determine the design goals in measurable circuit characteristics.

These can be timing, area, power, and capacitance. The logic library is crucial for design, it defines the implicit design rule constraints. Explicit optimization constraints such as physical constraints can be also defined. They help in optimization. [9].

Specify libraries: Next in the synthesis flow comes specify libraries step. In this step logic, symbol, and DesignWare libraries are used for design function implementation and for showing synthesis outcome graphically. The mapped logic libraries are called target libraries which are produced in optimization. Target libraries have the cells in them which are used to produce the netlist and definitions for the design's operating conditions. A design is compiled or translated by using the link libraries, even more specifically used libraries are the target libraries which are the subset of the link libraries. Link libraries contain the delay models information that are used to compute timing values and path delays. A pseudo library called ALIB can be used to explore trade-offs between area and delay. ALIB is created by characterizing the target library. ALIB contains real gates mapped from the Boolean functional circuits in the target library. The symbolic representations of the cells are in the symbol library. The DesignWare libraries contain reusable circuits which are used as a building elements within the synthesis run to optimize the speed and area. The physical libraries have to be specified if DC is used in topographical mode. These include the Milkyway reference library, holding physical information about the standard cells and macros. [9].

Read design: RTL design and gate-level netlist are used as an input for design compiler. It utilizes HDL compiler to interpret Verilog and VHDL designs and gate-level netlists. Also, netlist in ddc format can be read in the tool. DC can handle multiple designs when read into memory, even design modification can be made when in memory. [9].

Define design environment: The expected design environment has to be specified to enable optimization of the design. This means the operating condition, system interface features, and wire load models (if synthesis is not run-in topographical mode). Typical operating condition are temperature, voltage, and process fluctuations. System interface features comprise input drives, I/O loads, and fanout loads. Consequently, the design environment setup have an impact on the synthesis outcome. The design environment is specified with commands such as, *set_operating_conditions*, *set_drive*, *set_load*, *set_driving_cell*, *set_wire_load_model*, and *set_fanout_load*. [9].

Set design constraints: Constraints determine the design objectives in circuit characteristics for example, time, area, and capacitance. These are used in design optimization during synthesis run. There are two main constraint types design rule constraints and optimization constraints. Design rule constraints (DRC) are located in

the logic library, a design needs them to operate properly. DRC have a priority over optimization constraints. The design rule constraint types involve for example maximum transition time, maximum fanout, maximum capacitance, minimum capacitance, cell degradation, and connection class. Optimization constraints are defined by the designer. DC tries to meet optimization constraints within optimization phase, but cannot break the design rules thus, optimization rules have to be set realistically. The optimization constraints involve for example input and output delays, minimum and maximum delay, maximum area, and power optimization. Constraints are defined on the command line or in a file. Both of the constraints are tried to be met however, the design rule constraints are given priority. [9].

Select compile strategy: The compile strategy has to be set for hierarchical designs. The top-down compilation or the bottom-up compilation can be used for different parts of the design. The top-down compilation compiles the top-level design and the sub designs in conjunction. The bottom-up compilation compiles the sub designs individually from the lowest hierarchy level to the highest level i.e., the top-level design. Also, mixed compile is possible, it utilizes both of the strategies depending on what is the best option for the particular sub design. [9].

Synthesize and optimize the design: In synthesis run the optimization maps the design to best possible structure according to functional, speed, and area specifications. The design synthesis and optimization is incorporated to the compile process which is launch with the *compile_ultra* or the *compile* command. A technology-specific circuit is created in the optimization process. It is executed on three levels architectural optimization, logic-level optimization, and gate-level optimization. The architectural optimization operate on the HDL expressions performing common subexpression sharing, resource sharing, selection of DesignWare components, operator reorganization, and identifies arithmetic expressions to run data path synthesis. The logic-level optimization operate on the technology independent netlist, it performs two procedures structuring and flattening. The gate-level optimization operates on the technology independent netlist to create the technology-dependent netlist. It performs mapping, delay optimization, design rule fixing, and area optimization. [9].

Analyze and resolve design problems: Inherently, the synthesis and optimization results need to be analysed and it is done with area, constraint, and timing reports extracted from DC. These will help to sort out possible issues and to enhance the final outcome of synthesis. Reports can be created during the synthesis run before and after the compilation. [9].

Save the design database: DC does not save the design automatically. Therefore, the design have to be saved manually into ddc, Verilog, svsim, VHDL, or Milkyway format. This can be done for the whole design and for the sub designs whenever needed by using the *write_file* command. [9].

2.5 Synthesis in practise with DC

Synopsys Design Compiler is vigorous synthesis tool. It is used to execute the logic synthesis for ASIC designs. Logic synthesis is run with the help of special DC commands. This section will discuss about the important commands. [8].

When DC is launched it reads the starting file *synopsys_dc.setup* located in the working directory. Two starting files are needed: one in the working directory and another one in the root directory, where the DC installation is also located. Also, couple of parameters must be setup to make the tool usable:

1. *search_path*: Parameter utilized on the synthesis run to search the target technology libraries [8]. On this project it is used for the RTL file paths.
2. *target_library*: The logic cells are located in the *target_library*. DC uses this library to map the logic cells on the cell-level synthesis. [8]. On this project the 7 nm technology file paths are provided with this parameter.
3. *link_library*: This parameter is used to link the logic cells in the target technology libraries with the libraries containing referenced components and designs [9]. This is used for the memory instance file paths.

After these three parameters are setup, the tool can be used from the command prompt [8].

2.5.1 DC commands used during synthesis

A digital system is designed by using synthesizable HDL, the design is input for DC. Table 1 introduces some important DC commands utilized during synthesis run. [8].

Table 1. *DC commands [8].*

Line	Commands	Description
1	read -format <format_type> <filename>	Read the design in DC.
2	analyze -format <format_type> <list of file names>	Used to analyse the syntax and translation prior the generic design is built.
3	elaborate -format <list of module names>	Elaborates the design.
4	check_design	Check the issues in the design, such as shorts and connections.
5	create_clock -name <clock_name> -period <clock_period> <clock_pin_name>	Generates the clock for the design.
6	set_clock_skew <rising_clock_skew> <falling_clock_skew> <clock_name> -rise_delay fall_delay	Set the clock skew for the design.
7	set_input_delay <input_delay> <input_port> -clock <clock_name>	Define the input port delay.
8	set_output_delay <output_delay> <output_port> -clock <clock_name>	Define the output port delay.
9	compile -map_effort <map_effort_level>	Compile the design. Map effort level can be set to low, medium, or high.
10	write -format <format_type> -output <file_name>	Save the synthesis output.
11	set_false_path -from [get_ports <port list>] -to [get_ports <port list>]	Define the false path.
12	set_multicycle_path <period> -from [get_cells] -to [get_cells] -setup	Enable multicycle paths for design setup timing.
13	set_multicycle_path <period> -from [get_cells] -to [get_cells] -hold	Enable multicycle paths for design hold timing.
14	set_clock_uncertainty	Set the estimated network skew value.
15	set_clock_latency	Set the estimated source and network latency values.
16	set_clock_transition	Set the estimated clock skew value.
17	set_dont_touch	Disable optimization of the gates that are already mapped.

Commands related to reading the design in DC include analyse and elaborate which pass needed parameters during elaboration. The actual read design command is utilized to pass pre-compiled design in DC. The check design command is utilized after reading to check potential problems in the design. Previously mentioned commands are part of

the basic execution script that DC is running in synthesis run. Hence, this is utilized in this project. The create clock command specifies a clock for the design which is also used as a reference in timing analysis. The clock is linked to the clock pin in the design. If clock pin does not exist, synthesis creates virtual clock for the design. This is set in the constraint files that can be downloaded from GitHub webpage. Clock skew needs to be handled in synthesis; it is the difference between signal arrivals into different flip-flops. Positive clock skew adds margin for the setup timing. Since, the clock signal arrives late to the destination flip-flop compared to the source flip-flop. By contrast, negative clock skew adds margin for the hold timing. Since, the clock signal arrives late to the source flip-flop compared to the destination flip-flop. The clock skew is used to model the propagation delay in the clock tree in order to synthesize the design. It is set in the constraints file with the command `set_clock_transition`. Input and output delays are determined with command in Table 1 at lines 7 and 8. If needed the delays can be determined as minimum or/and maximum delays for both input and output. To do that following commands are utilized: [8]

1. `set_input_delay -clock <clock_name>-max <delay> <input_port>`
2. `set_input_delay -clock <clock_name>-min <delay> <input_port>`
3. `set_output_delay -clock <clock_name>-max <delay> <output_port>`
4. `set_output_delay -clock <clock_name>-min <delay> <output_port>`

The synthesis is executed with the compile command. This can be done with different levels of effort (low, medium, and high) and is specified with the command in Table 1 at line 9. This project uses newer command `compile_ultra -no_seq_output_inversion -gate_clock -spg -scan`. The resulting synthesis output is saved with the write command. The output can be saved in different formats such as VHDL or database format (ddc). This project saves it in verilog and ddc formats. Possible false paths in the design can be specified with the set false path command. Some know false paths are set in the provided constraints files. Multicycle paths in the design can be specified with commands in Table 1 at lines 12 and 13 to check setup and hold timing. This was used in order to improve setup timing. [8].

2.6 Design optimization using DC

SoC design can have multiple functional blocks which is actually good thing for the synthesis. Block-level constraints can be specified to satisfy area, speed, and power targets. The blocks in the design can have different clock and power domains and the functionality can be different. To optimize the design in synthesis, it can be divided into partitions according to the functionality. Hence, the block-level constraints need to be

specified in the sdc files. If block-level constraints are not met, the RTL code or the architecture may need to be changed. [8]. The synthesis tool optimizes the design, but it can usually improve the performance only by 10-20 percentage [22]. NVDLA is divided into 5 partitions based on the functionality, thus synthesis is driven for these individually. Constraints need to be specified also for the chip-level synthesis. The top-level design integrates different functional blocks together. The top-level constraints need to be specified properly to obtain required speed, area, and power numbers. A key factor is synchronization between the blocks. The don't touch attributes can be set for the verified clocks. Top-level constraints or top-level synthesis model are not specified for NVDLA. Main optimization and design rule constraints (DRC) are shown in Table 2. [8].

Table 2. *Optimization and design rule constraints [8].*

Line	Commands	Type	Description
1	set_max_transition	Design rule constraint	Specifies the maximum transition time.
2	set_max_fanout	Design rule constraint	Defines the maximum fanout.
3	set_max_capacitance	Design rule constraint	Defines the maximum capacitance
4	set_min_capacitance	Design rule constraint	Defines the minimum capacitance
5	set_operating_conditions	Optimization constraint	Specifies the PVT conditions
6	set_load	Optimization constraint	Utilized for load modelling on output
7	set_clock_uncertainty	Optimization constraint	Sets estimate for the network skew
8	set_clock_latency	Optimization constraint	Sets estimate for the source and network delays
9	set_clock_transition	Optimization constraint	Sets estimate for the input skew
10	set_max_dynamic_power	Optimization constraint	Defines the maximum dynamic power
11	set_max_leakage_power	Power constraint	Defines the maximum leakage power
12	set_max_total_power	Power constraint	Defines the maximum total power
13	set_dont_touch	Power constraint	Prevents the optimization of placed gates

Commands in Table 2 are discussed in following sections. NVDLA environment provides design rules constraints for the partitions, these were utilized in this project.

2.6.1 Design rule constraints (DRC)

The most important DRC are fanout, capacitance, and transition. However, usually all of these are not specified in the first synthesis iterations. They have greater importance within synthesis than optimization constraints. The max fanout command obtains the load driving capabilities of a port. The max transition command sets the maximum transition time for state change from '0' to '1' or from '1' to '0'. DC can set it for a particular net or for the whole design, by ignoring the library value. The max capacitance command defines the maximum net capacitance. DC utilizes it to handle violation within the compilation phase of the synthesis run. The capacitance derive from the cell characteristics. [8]. These are not specified for NVDLA in this project.

2.6.2 Optimization constraints

The area and speed constraints are utilized within the logic synthesis to optimize the design. While the power constraints are taken care of within physical synthesis. The speed is crucial for the performance of the design thus, the speed optimization is performed first followed by the area optimization. The set don't use command can be used if a cell from the technology library should be ignored. The set don't touch command can be used if some of the functional block are already optimized and does not require further processing. The set prefer command can be used when there is a need to map the design for a new technology library. The design can be hierarchical or flattened, if needed the design can be flattened during synthesis by utilising set flattened command. The set structure command can be used to enhance the area or gate count. This can be performed by using either Boolean structuring or timing driven structuring. The group and ungroup commands can be utilized to create or remove hierarchy in design respectively. [8]. In this project the set_clock_transition was used for the clocks. It is specified in the constraint files.

2.6.3 Design optimization strategies

To get an idea of synthesis in practise, lets study how an imaginary optimization strategy for a large SoC design could proceed: [8]

1. At first, run synthesis at block-level and generate report of all the violations.
2. Check the timing and area reports and try to adjust constraint at the block-level. Also report violating paths.
3. If the timing needs to be enhanced, utilize asynchronous path grouping and after that the compilation strategies.
4. If some block already met the timing use the don't touch command.
5. If the area need to be reduced, utilize grouping and resource sharing with the set max area command.
6. Generate report of the multicycle paths. This helps the timing analyser to check the setup and hold timing.
7. generate report of the false paths in the block-level design.
8. If block-level timing and area constraints are met, the top-level design constraints can be utilized to begin the optimizations at the top-level.
9. If individual blocks met the timing but the top-level does not, the following actions can be made to fulfil the top-level requirements:
 - a. Check timing violations and if they exist adjust the RTL to speed up data arrival at the partition boundary.
 - b. If partition boundary is in order, utilize register balancing technique to get rid of the setup time violations.
 - c. If the hold time violations emerge from the timing report, the data arrives to block interface too quickly and strategies to balance the path can be applied for example additional buffers.
 - d. If the timing is not met after these actions, the RTL or architecture can be adjusted. Finally, as a last resort the constraints can be relaxed.

This strategy is targeted for the top-level synthesis; thus, it was not that applicable for this thesis project. However, some of the steps between 1 and 7 were executed on the project.

Synthesis has 2 alternative compilation strategies top-down or bottom-up. Both have their pros and cons, and the designer can choose preferred approach. The top-down compilation works with the whole design i.e., on the top-level. It utilizes the top-level design constraints during the compilation. The pros of this approach include complete paths, since the optimization is executed for the whole design, results usually the most optimal design, does not need many iterations, constraints are simple, and straightforward information management. The cons, in turn include requires more and longer runtime and needs larger memory. [8].

The bottom-up approach starts the compilation at the block-level and proceeds from there to the top-level. This requires the usage of the set don't touch attribute to avoid

recompilation of already processed blocks. Also, the input and output timing information has to be set for all the block individually. The pros of this approach include relatively fast runtime, smaller amount of processing needed for a run, and smaller amount of memory needed. The cons, in turn include optimization is executed on block-level i.e., not necessarily useful for the full design, needs multiple iterations, and has to manage multiple hierarchies. [8]. Specific compilation strategy was not determined in this project since, the synthesis was executed for each partition individually. On the other hand, this can be seen as bottom-up compilation strategy if the project would have proceeded to the top-level synthesis.

Area optimization techniques can be applied to minimize the total area of the design. However, the priority is to optimize the timing and subsequently the area. The area can be reduced by adjusting the RTL level. When using DC to minimize area, a few recommendations are worth to follow do not create individual combinational logic blocks, avoid using glue logic to connect two designs, and define the `set_max_area` attribute for the design during synthesis run. [8]. The `set_max_area` attribute was utilized in the synthesis runs. In this project it was set to 0,0 to get optimal synthesis results.

The usage of individual combinational logic blocks can lead to poor partitioning which will hinder the optimization during synthesis run since, the hierarchy is fixed. DC is unable to optimize the port interfaces, meaning that it cannot edit the design hierarchy if the blocks are separated. Another example of poor partitioning is the usage of glue logic. Glue logic is for example a logic gate between two combinational logic elements. Therefore, DC cannot optimize the design as a whole because the glue logic separates the elements. This can be avoided by using the `group` command to assort the logic gate to either element. Minimal area can be achieved by utilizing the `set_max_area` attribute during the synthesis. The optimization priorities are DRC, timing, power, and area. With the attribute area optimization can be started after the timing is met. Design area can be reported with command `report_area`. [8].

The timing has the highest priority compared to other circuit characteristics i.e., power and area. Initially the optimization checks DRC violations, followed by the timing violations, the power constraints, and eventually the area constraints. DC has some timing optimization commands; those are studied next. mapping can be executed with different effort levels. One approach is to start initial compilation with the command `map_effort_medium` since, it reduces the compilation time. If timing constraints are not met, the `map_effort_high` command can be utilized, it should enhance the performance. If the timing constraints are not met with high mapping effort, the `group_path` command can be utilized to intensify the compilation. When critical paths are identified from the

timing report, the compilation time can be increased by giving a weight factor for the paths with the `group_path` command. [8]. In this project the compilation was executed with high mapping effort. Further, the `group_path` command was utilized in the project to enhance timing on violating NVDLA blocks.

Important timing analysis parameters include the setup and hold slack, those can be extracted from the timing reports. The setup slack is defined as the difference between the data required time and the data arrival time and it should be positive. If it is not positive, the data path is usually too slow and needs RTL adjustments. The hold slack is defined as the difference between the data arrival time and the data required time and likewise it should be positive, meaning that there is no transitions during the hold time period. If it is not positive, the data path is usually too fast and needs RTL adjustments. [8]. The setup and hold slacks are defined in the post-synthesis STA to get an idea how to improve the design if needed. The STA analysis for NVDLA synthesis results are presented later on the thesis.

A SoC design can have multiple hierarchies and even though each sub-design met the constraints the top-level design might not. The *characterize* command can be utilized on this kind of situation. It enables the boundary conditions for the sub-modules in the top-level hierarchy environment. With this the compilation and characterization can be executed for the submodules independently. [8]. Top-level synthesis was not executed in this project.

Design compiler is able to perform register balancing. Register balancing is highly effective method to improve the performance of pipelined circuit architecture. It moves combinational logic from one pipelined stage to another pipelined stage, reducing the delay on slower register to register path. Hence, the performance is improved because it is defined by the slowest register to register path. The functionality of the total path is not changed because the combinational logic is just re-placed to different pipeline stage on the same path. Register balancing is executed by using the command *balance_register*. [8]. Register balancing was not utilized in this project.

A pipelined design can have a structure that requires more than one clock cycle to execute the functionality. This is known as the multicycle path. Multicycle paths need to be reported to execute the setup and the hold checks correctly. Multicycle path is defined by using the *set_multicycle_path* command. [8]. Multicycle paths were set for some of the NVDLA blocks in the constraint files in this thesis project.

3. LITERATURE REVIEW

This section presents literature review from related studies. Exactly same kind of studies as this thesis are not found. However, the review include other studies which used NVDLA and some important studies from hardware accelerator field.

First, a paper *Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks* by Y. H. Chen, J. Emer, & V. Sze is studied. Convolutional neural networks (CNNs) are becoming more and more popular in contemporary AI systems, because of their excellent precision in many application fields. However, CNNs have high computational complexity occurring from convolutional computations which are executed in parallel fashion. Thus, the data movement is high in the process requiring lots of energy. Minimizing data movement is the key factor to improve energy efficiency in CNN's. Chen et al. presents a new dataflow to achieve that, it is called as RowStationary (RS). The presented architecture utilizes local data reuse on filter weights and pixel maps, and decreases data movement on computations such as partial sum accumulation. The RS dataflow can be applied to many kind of CNN configurations, since it lowers data movement by using processing engine (PE) local storage, direct inter-PE communication and parallelism. RS dataflow test on the AlexNet CNN prove that it is more energy efficient than other dataflows on CNN field. RS dataflow is implemented on a real chip confirming the analysis of energy efficiency. [18]. A paper presenting the implementation is studied next.

A paper *Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks* by Y. H. Chen, T. Krishna, J. S. Emer, & V. Sze presents the implementation of RS dataflow. The Eyeriss is a hardware accelerator for CNN applications. Key benefit of Eyeriss is the improved energy efficiency of the whole system. This is achieved by minimizing both on-chip and off-chip data movement which is more energy consuming than the actual computing, for example data is reused locally and communication to DRAM is reduced. Hence, Eyeriss utilizes the RS dataflow architecture with 168 PE's. Further, techniques such as compression and data gating are used to minimize energy consumption. The chip is fabricated to TSMC 65 nm CMOS technology, chip size is 16 mm², it contains 1176k 2-input NAND gates, and operates on 250 MHz maximum clock rate. For AlexNet the performance is 35 frames/s and 0,0029 DRAM access at 278 mW. For VGG-16 the performance is 0,7 frames/s and 0,0035 DRAM access at 236 mW. [24].

A paper *Origami: A 803-GOp/s/W Convolutional Network Accelerator* by L. Cavigelli & L. Benini is studied next. The paper presents new architecture for Convolutional networks, discusses about its implements on real ASIC, and performs silicon measurements to that ASIC. Origami is designed for object recognition applications. The architecture contains 4 primary parts: Image Window SRAM and Image Bank, Filter Bank, Sum-of-product units (SoP), and Channel summer units (ChSum). Image window SRAM and image bank receives new images and sends those to SoP units. Filter bank receives filter weights and sends them to SoP units. When SoP units have received data from previous stages, they calculate the inner product of an image patch and a filter kernel. These results are then send to ChSum units. ChSum unit calculates total for inner products and outputs the processed pixel stream. The architecture has 2 clocks: Image window SRAM, Image bank and filter bank are operator with one clock. SoP units and ChSum units are operated with another clock which is 2 times faster than the initial clock. The Origami is implemented on UMC 65 nm CMOS technology. It uses 250 MHz clock for I/O and SRAM and 500 MHz clock for SoP and ChSum units. Operating voltages are 0,8 V and 1,2 V respectively. The core area is 3,09 mm². Total power consumptions are 237 mW at 0,8 V and 654 mW at 1,2 V. Technology scaled results show that Origami is area efficient and has the lowest power consumption compared to other designs. [23].

Article *CoNNA – Compressed CNN Hardware Accelerator* by R. Struharik, B. Vukobratovic, A. Erdeljan, & D. Rakanovic is explored next. Article present new CNN hardware accelerator architecture; it is called as CoNNA. It is used to accelerate trimmed and quantized CNNs. The design is a coarse-grained architecture which can be reconfigured dynamically within the processing. The architecture contain following elements: Reconfigurable Computing Unit (RCU) which executes all calculations, Input Stream Manager (ISM), it delivers compressed data from memories to RCU, and Output Stream Manager (OSM), formats, compresses and delivers output data back to the memories. The data movement with related systems is handled with four AXI interfaces: Input Stream Interface which supplies all input data to RCU, Output Stream Interface connects to OSM and supplies output data to the memories, CNN Description Interface supplies CNN data in to CoNNA. The above has been implemented as AXI-Full interfaces. Lastly, Configuration Interface implemented as AXI-Lite interface controls and configures the CoNNA architecture. CoNNA is tested against Eyeriss, NullHop, and NVDLA CNN accelerators, when running AlexNet, VGG-16, VGG-19, GoogleNet, and ResNet-50 CNNs. To compare the results CoNNA was configured to match MAC unit count and operating frequency in the earlier designs. The experiments show that CoNNA executes CNNs faster than the reference designs. [16].

Article *The Implementation of LeNet-5 with NVDLA on RISC-V SoC* by S. Feng, J. Wu, S. Zhou & R. Li is explored next. This article studies the performance of NVDLA when it is added on RISC-V core and runs LeNet-5 CNN. The implemented SoC contains two interesting components the main CPU which is RISC-V core and the hardware accelerator which is small NVDLA core. Communication between the components in the system is handled with ARM Advanced Peripheral Bus (APB) and AXI4 Bus. The APB connects to NVDLA with special adapter. The AXI4 is used to connect NVDLA to external DRAM. The system is compared to Intel i5 9300H (2,4 GHz) and K210 (400 MHz) processors and the LeNet-5 is ran with INT8 precision. The conclusion of the article is that NVDLA (500 MHz) provides proper acceleration for the CPU on the computations with low power consumption. [30].

A paper *Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim* by F. Farshchi, Q. Huang, & H. Yun analyses NVDLA performance. This experiment utilizes FireSim simulator on the Amazon cloud FPGA. On this environment NVDLA is integrated with a RISC-V SoC and the system runs YOLOv3 object-detection algorithm. The used NVDLA configuration is *nv_large* which is wrapped to manage the communication between NVDLA, and RISC-V. Communication with rest of the system is handled through CSB, DBB, and IRQ interfaces. The used platform is a cost-effective and flexible solution to carry out the research on large SoC's. The results show that NVDLA has better acceleration performance than GPU and CPU solutions, performance can be even enhanced by utilizing larger cache and/or hardware prefetcher, and shared memory between CPU and NVDLA causes major disturbance to the process. [28].

Article *Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices* by Y. H. Chen, T. J. Yang, J. S. Emer, & V. Sze is studied next. The article present new deep neural network (DNN) architecture for mobile devices. These devices require low energy consumption and has recourse limitations. Therefore, compact and sparse DNNs are needed. The problems are solved with a new NoC and PE designs. The NoC is a hierarchical mesh, which can handle a broad range of bandwidths to keep the PEs working and the processing efficient. The PE utilizes the sparse nature of weights and activations on multiple DNN layers to enhance performance and reduce energy consumption. Eventually, these two are combined in Eyeriss v2 to form the DNN hardware accelerator. The design was implemented on 65 nm TSMC LP 1P9M technology. Place-and-route step is performed for the design and the reported results are from post-layout cycle-accurate gate-level simulation. The logic only gate count in Eyeriss v2 is 2695k NAND-2 gates, maximum clock rate is 200 MHz, and number of MACs is 384. Compared to original Eyeriss the performance and energy efficiency of

Eyeriss v2 are improved by 42,5× and 11,3× respectively, when running AlexNet. When running MobileNet performance and energy efficiency are improved by 12,6× and 2,5× respectively, compared to original Eyeriss. [31].

A paper *ENVISION: A 0,26-to-10TOPS/W Subword-Parallel Dynamic-Voltage-Accuracy-Frequency-Scalable Convolutional Neural Network Processor in 28 nm FDPOI* by B. Moons, R. Yutterhoeven, W. Dehaene, & M. Verhelst is studied next. This paper present the hierarchical recognition processing idea on Envision CNN architecture to be able to develop always-on applications. Envision contains 3 power and body-bias domains, 16-bit SIMD RISC, 2D- and 1D-SIMD arrays, a scalar unit, 64×2kB single-port SRAMs, and Huffman DMA. The design was implemented on 28 nm FDSOI technology, core area is 1,87 mm², with 1V clock rate is 200 MHz, gate count is 1,95M (NAND-2), and maximum count of MACs is 1024 (scalable). The average power consumptions when running AlexNet and VGG-16 are 44 mW and 26 mW, respectively. The results show that Envision is more efficient than the Origami or the Eyeriss designs on recognition tasks. [32].

A paper *A 1,06-to-5,09 TOPS/W Reconfigurable Hybrid-Neural-Network Processor for Deep Learning Applications* by S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, L. Liu, & S. Wei present hybrid neural network processor. Commonly, deep learning architectures such as AlexNet are combinations of Convolutional Neural Network (CNN), Full Connection Network (FCN), and Recurrent Neural Network (RNN). The foundation of presented processor is on a heterogenous PE array. It can be spatially partitioned, and it can form variant bit-width data path. The architecture contains Two PE arrays: general PE and super PE, a controller, two on-chip memories, a sharing weight buffer, and an adaptive interface. The design is implemented on 65 nm LPCMOS technology, and at 1,2V clock rate is 200 MHz. When running AlexNet and LRCN the chip performance reaches 1,27 TOPS/W for 105 fps and 1,28 TOPS/W for 83, respectively. Compared to other designs the presented processor gains better energy efficiency by 5,2×. [33].

A paper *UNPU: A 50,6 TOPS/W Unified Neep Neural Network Accelerator with 1b -to 16b Fully-Variable Weight Bit-Precision* by J. Lee, C. Kim, S. Kang, D. Shin, S Kim, & H. J. Yoo is explored next. This paper introduces a Unified Neural Processing Unit (UNPU), it can process convolutional layers, fully-connected layers, and recurrent layers from 1 to 16 bit -precision. Usually, all of these are not supported. The architecture contains 4 DNN cores, an aggregation core, a 1 D SIMD core, and RISC controller. Communication between these elements is managed with an on-chip network. The UNPU is implemented on 65 nm CMOS process node, die area is 16 mm², it operates up to 1,1 V with maximum clock rate of 200 MHz, and with these condition it consumes power 297 mW. UNPU

operates with 3,08 TOPS/W, 11,6 TOPS/W, and 50,6 TOPS/W performance on 16-, 4-, and 1-bit weights, respectively. The fabricated chip is able execute facial expression recognition and dialogue generation tasks. These functionalities have been proven with FER2013 and the Twitter dialogue database. [34].

To sum up the chapter, it can be stated that a comprehensive set of related studies was found from literature. Two type of paper were review. Some of the paper were studied because they had similar results to this project, which could be then compared [18, 24, 23, 31, 32, 33, 34]. Rest of the paper did not have comparable results but those studied also NVDLA which made them interesting for this project [16, 30, 28,]. However, all of the papers were studying CNN hardware accelerators but from slightly different points of view.

4. NVDLA IMPLEMENTATION FLOW

The implementation flow is discussed next. The goal of this thesis project is to execute logic synthesis phase of the ASIC design flow for the NVDLA open-source hardware accelerator. NVDLA source codes can be downloaded from the GitHub [14]. The logic synthesis was performed with Synopsys Design compiler and the floorplan was created with IC Compiler 2. The target technology node is 7 nm. Actual logic synthesis run was executed on external LSF computing server. The studied design was NVDLA version 1.0 [14]. The `nv_dla1 (nv_full)` is a non-configurable “full-precision” version of the NVDLA. It is fixed at 2048 8-bit MACs [14]. The NVDLA architecture and software design are discussed next.

4.1 NVDLA

The NVIDIA Deep Learning Accelerator (NVDLA) is a standardized open architecture hardware design to meet the demand for computational inference. The NVDLA design is scalable and very configurable. At the same time, it is also flexible and simplifies integration. Its execution is mostly based on four mathematical operations which are convolutions, activations, pooling, and normalization. Generally, computations in deep learning inference are based on these. This is because they have characteristics that make them good match for special-purpose hardware implementations. These are highly predictable memory access patterns, and they are easy to parallelize. [6]. Furthermore, CNN hardware accelerators extend memory, explore parallelism, and reduce data movement when implemented on FPGA or ASIC devices [29].

NVDLA hardware inference acceleration solution is simple and agile. It is applicable for many performance levels and can be used in from cost-sensitive Internet of Things (IoT) devices to large high performance IoT devices. In practice NVDLA is a set of IP-core models built with an open industry standards. NVDLA is implemented in RTL form as Verilog code, it is a synthesis and simulation model. NVDLA also includes the TLM SystemC simulation model which is made for software development, system integration, and testing. The NVDLA is open-source project and project management is implemented as an open, directed community. [6]. Other similar kind of convolutional neural network hardware accelerators are CoNNA and MIT Eyeriss CNN accelerator [16, 18]. The Eyeriss is also much more flexible design than NVDLA [26]. However, NVDLA dataflow performance is better than of NLR (No Local Reuse) strategy [27].

NVDLA architecture is designed to make configuration, integration, and portability easier. It uncovers the building blocks of deep learning inference operations acceleration. NVDLA hardware includes 5 main components which are:

1. Convolution Core
2. Single Data Processor
3. Planar Data Processor
4. Channel Data Processor
5. Dedicated Memory and Data Reshape Engines

All these blocks are separated and can be configured independently. For example, if a system does not need so much convolutional performance the performance of the convolutional block can be scaled down or vice versa if it needs additional convolutional performance, it can be scaled up. This can be done without editing other blocks in the accelerator. CPU or co-processor handles operations scheduling for each block. Blocks operate on tight scheduling boundaries independently. This kind of scheduling management can be made in two ways on high level these are the “headed” and “headless” implementation. In “headed” implementation this is done by adding management coprocessor in the design as part the NVDLA sub-system. In “headless” implementation this functionality is merged to the main processor by using higher level driver implementation. The benefit of this is that NVDLA hardware architecture can be used in different size implementations. [6].

Interfacing is handled with common practises between the NVDLA hardware and rest of the system. In the design a control channel realizes interface for register file and interrupt. Two standard AXI bus interfaces enable the interface with memory. The main memory interface connects to the broader memory system which includes system DRAM, and this memory interface will be common with CPU and I/O peripherals in the system. The optional second memory interface enable a connection for higher-bandwidth memory dedicated to NVDLA or some external subsystem. The benefit of this optional heterogeneous memory interface is additional flexibility to scale between various host systems. [6].

The inferencing flow starts with processor transmitting down the hardware configuration of a layer together with an “active” command. In a “headed” implementation this executed by a microcontroller and in a “headless” implementation by the main CPU. It is also possible to transmit down multiple layers to different engines and activate them at the same time if this is not excluded by data dependencies. To ensure smooth execution all, the engines have a double-buffer for their configuration registers with this they can get the configuration of a second layer and start execution instantly after the active layer has

finished. After a hardware engine completes its active job, it transmits an interrupt to the processor to inform the completion and the processor can begin the same process again. previously described command-execute-interrupt flow iterates until the inference is completed for the whole network. Figure 8 shows the two NVDLA system models. On the left side is the small “headless” implementation which is intended for more cost-sensitive devices. On the right side is the large headed implementation which have the additional coprocessor and high-bandwidth SRAM for the NVDLA sub-system. The intended use of large system are high-performance IoT devices which may execute multiple jobs simultaneously. [6].

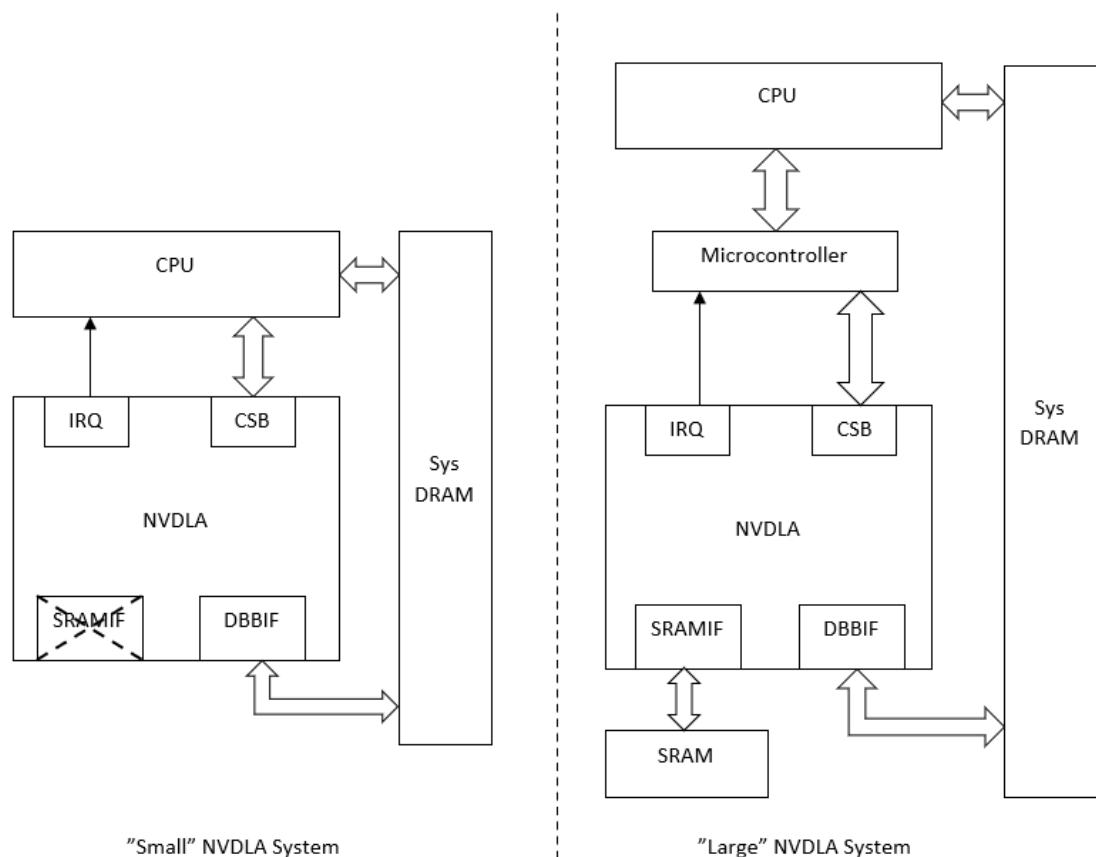


Figure 8. Two different NVDLA systems [6].

Purpose of the *small-NVDLA model* is to enable usage of deep learning technologies in areas where it has not been feasible before. It is targeted for cost-sensitive Internet of Things (IoT) devices with AI and automation systems which operations are well-defined and have cost, area, and power as the main drivers. Flexibility in configurable resources in the NVDLA is the key factor to achieve saving in cost, area, and power. Neural network models in the NVDLA allow pre-compilation and performance adjustment in a way that enables size reduction and scales down load complexity in larger models. With these adjustments the whole NVDLA implementation is smaller whereupon models use less

storage and are quicker for system software to load and process. [6]. Even the small-NVDLA has high performance and low power when compared to normal processors [31].

Typically, these kind of systems execute one task at a time and due to that system performance is not a big concern while NVDLA is operating. Usually, processor architectural choices and usage of task management systems like FreeRTOS are the result for reasonable amount of NVDLA interrupts in the main processor. Resulting that extra microcontroller is not needed because the main processor is able to perform all the rough scheduling, memory allocation and the more fine-grained management of NVDLA. [6].

Systems implemented as the small-NVDLA model the system performance is usually not a priority which is the reason why it is not using the optional second memory interface. In this case not having a high-speed memory path is unlikely to impact to the system performance. These system usually use DRAM as the system memory because it consumes less power than SRAM. Further the system memory is used as a computation cache which makes the system more power-efficient. [6].

When the primary objectives are high performance and versatility *the large-NVDLA model* is a preferable choice. High performance IoT systems might execute inference on various network topologies due to that flexibility is important for these kind of systems. These systems may execute multiple task at the same time, so it is important that they don't use too much processing power on the host. The large-NVDLA hardware has a second memory interface made for high-bandwidth SRAM. It is optional and can be used if needed. It will be used to interface with coprocessor to help the main processor with the interrupt load. [6].

If the system is implemented with a high-bandwidth SRAM, it is connected to a fast-memory bus interface port on NVDLA. When included NVDLA will use this SRAM as a cache, further it can be distributed by other high-performance computing components on the system. The purpose of this functionality is to decrease communication load to the main system memory. [6].

General purpose processors like ARM Cortex-M and Cortex-R processor or alternatively inhouse microcontroller designs usually fill the requirements for the NVDLA coprocessor. Even though coprocessor is in use some tasks are handled by the main processor to manage NVDLA properly. The tasks can be divided as follows, the coprocessor is responsible of scheduling and fine-grained programming, while the main processor is responsible of rough scheduling on the NVDLA hardware, it has also the responsibility of IOMMU mapping for memory access in NVDLA, memory allocation of input data, fixed

weight arrays on the hardware, and synchronization of other system components and tasks executed on NVDLA hardware. [6].

4.1.1 Hardware architecture

The NVDLA architecture has two operation modes. The options that can be programmed are independent mode, and fused mode. In *independent mode* functional blocks are configured according to what and when it executes, so every block is working with assigned task. This means that blocks in NVDLA are performing operations in memory-to-memory fashion, the data goes in and out of systems main memory or SRAM memory if enabled. *Fused mode* is quite similar to independent mode. The difference in operation is that some of the blocks may be linked together as a pipeline. Benefit from this is improved performance because data doesn't have to go to memory and again back to block. Instead of this communication between blocks is handled with small FIFOs. Here is an example of this, data can be sent straight from the convolution core to the Single Data Point Processor and from there to the Planar Data Processor, and then pass to the Cross-channel Data Processor. Figure 9 represents the internal architecture of headless NVDLA core. [6].

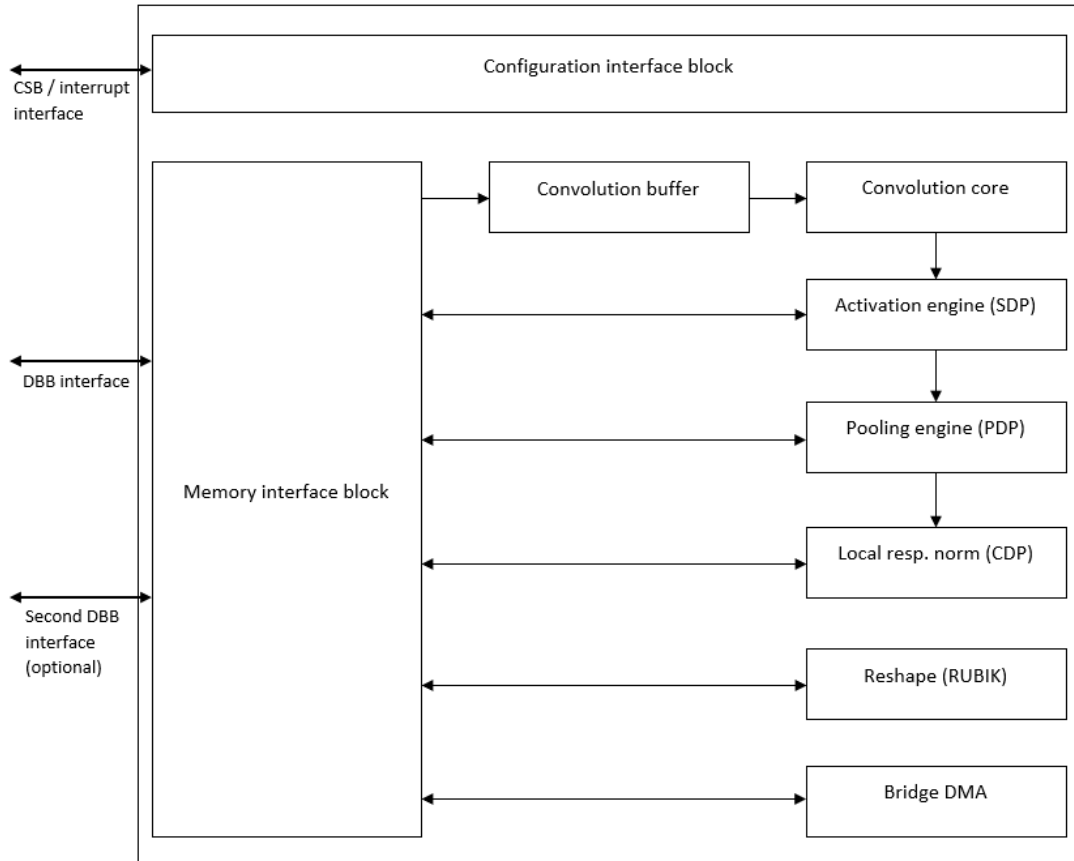


Figure 9. Headless NVDLA architecture [6].

NVDLA connects to the rest of the system with three connections which are Configuration Space Bus interface (CSB), Interrupt interface, and Data Backbone interface (DBB). *The CSB* is a synchronous interface, it has low-bandwidth, low-power, and 32-bit control bus that is made for a CPU to access configuration registers in the NVDLA. Working principle on the CSB interface is that NVDLA function as a slave. Interface protocol on CSB implementation is simple which gives flexibility to convert to AMBA, OCP or some other system bus. *Interrupt interface* is for a 1-bit level driven interrupt. The interrupt line in NVDLA hardware is used on two cases, when task is completed or if an error occurs. *The DBB* interface is the main connection between NVDLA and the main system memory subsystem. It is setup to be a synchronous, high-speed, and highly configurable data bus. This bus can be configured based on the systems requirements, it can have different address sizes, data sizes, and issue different sizes of requests. The DBB is a simple interface alike to AXI and can be used with AXI-compliant-systems. There is also second connection for the DBB interface, it is an optional connection which may be used when a second memory path is available. The second connection is similar to the primary Data Backbone interface and is designed to be used with an on-chip SRAM which can raise up the throughput and bring down access

latency. The second interface is not needed to get NVDLA working, by removing it systems can save area. [6].

Components in the NVDLA architecture are design specifically for deep neural networks to assist inference operations on it. NVDLA has 6 key functional blocks which are Convolution operations block, Single Data Point Processor, Planar Data Processor, Cross-channel Data Processor, Data Reshape Engine, and Bridge DMA. NVDLA hardware supports some deep learning frameworks like TensorFlow. [6]. These CNN layers are the reason why modern CNNs have high performance [18].

Convolution operations are handled in *Convolution core* and *buffer* blocks. They work with 2 data sets “weights” and “features”. Weights are offline-trained, and they are constant between different inference runs. Feature data set is the input data which changes depending on the network’s input. The convolutional hardware engine can be used with different parameters to operate with different size convolutions and to keep high efficiency. It is implemented as a naive convolution engine to optimize and upgrade performance. Optimizations include support for sparse weight compression to save memory bandwidth. Winograd convolution support enhances computing efficiency for different filter sizes. Batch convolution may save some extra memory bandwidth with reused weights when multiple inferences are executed in parallel. The NVDLA convolution engine has its own internal RAM, it is for storing weights and input features and it is called as the “convolution buffer”. The benefit of this is better memory efficiency which is obtained by eliminating the need to send a request to the system memory controller every time a weight or feature is required. [6].

The *Single Data Point Processor (SDP)* or *Activation engine* block makes possible the usage of linear and non-linear functions onto single data points. In Convolutional Neural Networks this is usually used instantly after the convolution operation. For non-linear functions the Single Data Point Processor possess a lookup table and for linear functions it uses plain bias and scaling. Combined together they can execute basic activation functions and other element-wise operations such as ReLU, PReLU, precision scaling, batch normalization, bias addition with other sophisticated non-linear functions, like a sigmoid and a hyperbolic tangent. [6].

The *Planar Data Processor (PDP)* or *Pooling engine* block exists in order to run specific spatial operations which are commonly used in Convolution Neural Network applications. It can be configured during a run to aid various pool group sizes, and to aid three pooling functions which are maximum-pooling, minimum-pooling, and average-pooling. [6].

The *Cross-channel Data Processor* (CPD) or *Local resp. norm* block works with multi-plane operations. It is a customized component made to utilize the local response normalization function (LRN). It is a normalization function operating especially on channel dimensions, unlike to the spatial dimensions. [6].

The *data reshape engine* (RUBIK) runs data format transformations here are a few examples splitting or slicing, merging, contraction, and reshape-transpose. The inference process on a convolution network usually requires reconfigured or reshaped data in memory. This means that different features or spatial areas of a picture may detached with “slice” operations and operations like “reshape-transpose” generates a larger output data than the input dataset. and general deconvolutional network operations like “reshape-transpose” generate output data with larger dimensions the input dataset. [6].

The *Bridge DMA* (BDMA) module acts as an engine for copying data to transmit it between the system DRAM and the high-speed memory interface, when used. Meaning that this module provides connection between the system DRAM and high-speed memory interface to accelerate data transfer. [6].

NVDLA is highly configurable in order to fit it in different applications. Another reconfigurable CNN design is the Compressed CNN hardware accelerator (CoNNA) [16]. This flexibility is achieved by extensive set of configurable hardware parameters which balance area, power, and performance. Some of these are listed below. [6].

1. *Data types*. (NVDLA is set to support multiple data types in its operational parts. Area may be saved by choosing limited subset of these. Available data types are binary; int4; int8; int16; int32; fp16; fp32; and fp64.) [6].
2. *Input image memory formats*. (Different image modes may be enabled or disabled to save area. Options include planar images, semi-planar images, or in general other packed memory formats.) [6].
3. *Weight compression*. (Memory bandwidth may be reduced in NVDLA, this can be done by sparsely storing convolution weights. Weight compression can be turned off to save area.) [6].
4. *Winograd convolution*. (The Winograd is an optimization algorithm for convolution in some of its dimensions. It can be enabled or disabled depending of the NVDLA implementation.) [6].
5. *Batched convolution*. (Batching is convolution operation that can save memory bandwidth. It can be enabled or disabled depending of the NVDLA implementation.) [6].
6. *Convolution buffer size*. (The convolution buffer is constructed from several banks. The number of banks can be set from 2 to 32 and the size of these can set from 4KiB to 8 KiB. The total amount of convolution buffer memory can be defined by multiplying the above parameters together.) [6].

7. *MAC array size.* (NVDLA contains the multiply-accumulate engine which is formed in two dimensions. The width dimension is market as “C”, it can have a value from 8 to 64. The depth dimension is market as “K”, it can have a value from 4 to 64. The total amount of multiply-accumulates can be defined by multiplying the above parameters together.) [6].
8. *Second memory interface.* (NVDLA has optional second memory interface designed for high-speed access. It can use only one interface but when needed this second interface can be enabled.) [6].
9. *Non-linear activation functions.* (Nonlinear activation functions can be removed from the lookup table if there is need to save area. These functions include for example sigmoid and tanh.) [6].
10. *Activation engine size.* (Activation outputs per cycle can be set from 1 to 16.) [6].
11. *Bridge DMA engine.* (This module is not needed in all implementations so it can be disabled to save area.) [6].
12. *Data reshape engine.* (The data reshape engine is optional and can be disabled to save area.) [6].
13. *Pooling engine presence.* (The pooling engine is optional and can be disabled to save area.) [6].
14. *Pooling engine size.* (Size of the pooling engine can be set to generate 1 to 4 outputs per cycle.) [6].
15. *Local response normalization engine presence.* (This functionality can be disabled to save area.) [6].
16. *Local response normalization engine size.* (The local response normalization engine can be set to generate 1 to 4 outputs per cycle.) [6].
17. *Memory interface bit width.* (Bit width of the memory interface depends of the width of the external memory interface. Size of the internal buffers can be set with relation to the external memory.) [6].
18. *Memory read latency tolerance.* (The number of cycles between read request and read data return is the memory latency time. This can be set to different latency times which in turn affects the internal latency buffer size of read DMA engines.) [6].

4.1.2 Software design

NVDLA software ecosystem has extensive cover of software features. The related software is divided into 2 groups. These are the compilation tools and the runtime environment. The compilation tools implement model conversion, and the runtime environment is run-time software which enables networks loading and execution on NVDLA. Figure 10 visualizes NVDLA system software functioning. It is the general dataflow inside of NVDLA system software. [6].

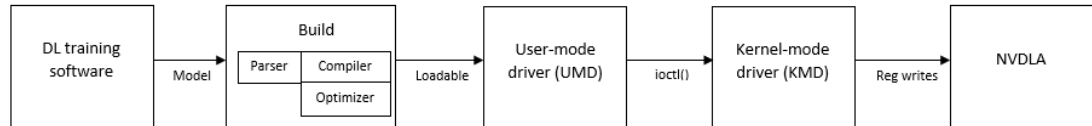


Figure 10. NVDLA system software dataflow [6].

Compilation tools contain compiler and parser. Compiler generates a set of optimized hardware layers according to the specific NVDLA configuration. By optimizing model size of hardware layers network, load and run times are reduced which in turn improves performance. Compilation is a multi-step process that may be divided into 2 basic components which are parsing and compiling. The parser may be implemented in a simple way, its functionality can only be based on reading a pre-trained Caffe model and creating an “intermediate representation” of a network which is then passed on to the following step of compilation. Inputs for the compiler are the parsed data and the hardware configuration of specific NVDLA design from these it creates a network of hardware layers. Previous steps are executed offline and can be executed on target device for which the NVDLA design is implemented. [6].

Knowledge about configuration in particular NVDLA hardware design is helpful. It helps the compiler to create suitable layers for the available features. This means that the compiler can select most suitable convolution operation mode for the design for example Winograd convolution or basic convolution, it can even split convolution operations into small pieces depending on the size of the convolution buffer. This step also handles model quantization to lower precision and memory region allocation for appropriate weights. The same compiler tool will be able to create an operation list for numerous distinctive NVDLA configurations. [6].

Going further in the dataflow next there the runtime environment. It runs an inference model on NVDLA hardware. The runtime environment is divided into two functional layers which are user mode driver and kernel mode driver. User mode driver and user-mode programs are the primary interfaces. As the data goes forward from parsing the network will be compiled layer by layer and converted into “NVDLA loadable” which is a file format type in NVDLA design, these tasks are performed by the neural network compiler. As soon as loadable is available user mode runtime driver loads it and issue inference task for kernel mode driver to continue the process. Kernel mode driver contains drivers and firmware that schedule layer operations on NVDLA, it is also responsible of programming the NVDLA registers to define all the functional units. [6].

Let’s take a closer look of previously described drivers. When the runtime execution begins, there exist a representation of the network so called “NVDLA loadable”

illustration which is stored in memory. From the perspective of a loadable NVDLA design is made of functional blocks that are represented in a form of layers in software. All the layers have data about their dependencies. Each single layer knows the tensors it will use as inputs as well as outputs in memory. It even has the information about the specific configuration of single block to run particular operation on it. The link between layers is a dependency graph, it is utilized to schedule operations by kernel mode driver. The NVDLA loadable is standardized format for all compiler and user mode driver implementations. Designs made in accordance with the NVDLA standard should comprehend any NVDLA loadable illustration. Inference can be performed with the loadable illustration, even if the implementation does not have all the required features. [6].

In user mode driver “loadable” illustrations are processed through a standard application programming interface (API), it also handles input and output tensors binding to memory locations and performing inference. The purpose of this layer is to load the network into memory and pass it on to the kernel mode driver as defined by the implementation. e.g., Linux kernel uses *ioctl()* to transmit information between the user mode driver and the kernel mode driver. This can be done by a simple function call if it is performed on a single-process system, where the kernel mode driver operates in the same environment with the user mode driver. [6].

The primary entry point in kernel mode driver receives an inference task in memory, it selects the task to be scheduled from multiple options and then issues it to the core engine scheduler. The above referred to functioning of multi-process system. Core engine scheduler responsibilities include processing interrupts from NVDLA hardware, managing the network by scheduling layers on functional blocks, and updating dependencies between layers if for instance next layer needs updates based on the task done on a previous layer. The scheduler determines when the subsequent layer is ready to be scheduled by using data from the dependency graph. The benefit of this function is optimized scheduling of layers. It also prevents performance deviation between various implementations of kernel mode driver. Portability layers in the NVDLA are shown in Figure 11. [6].

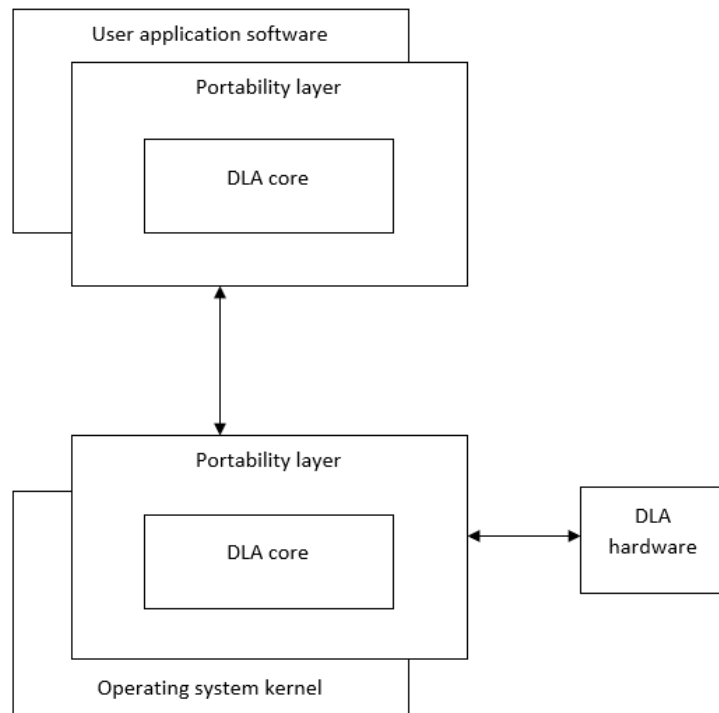


Figure 11. Visualization of portability layers in the NVDLA system [6].

The system is configured in a way where the user mode driver pile and the kernel mode driver pile exist as defined APIs. They are supposed to be wrapped to a system by portability layer. Core implementations should sustain in the portability layer quite easily without major changes and facilitate efforts to execute an NVDLA software-pile on many platforms if needed. The similar core implementations should compile equally easily on Linux as on FreeRTOS when portability layers are properly in place. Correspondingly, on designs implemented with a combined microcontroller as “headed” NVDLA implementation, the portability layer enables to execute the same low-level software on the microcontroller as what would be executed on the main CPU if the design is implemented as a “headless” implementation without coprocessor. [6]. This concludes the discussions about the NVDLA design.

4.2 Implementation flow

To work with the NVDLA the design environment needs to be setup. The final Verilog RTL code requires three build to obtain the desired configuration according to a feature specification, in this case the `nv_full`. The RTL is built by executing commands `hw/make` and after that `./tools/bin/tmake -build vmod`. [6].

The downloaded NVDLA design also contains a reference synthesis setup [6]. It was utilized in this project to execute logic synthesis with Design Compiler in Topographical mode. The directory structure for synthesis is visualized in Figure 12.

SCRIPTS

```

├── dc_app_vars.tcl
├── dc_interactive.tcl
├── #dc_run.tcl#
├── dc_run.tcl
├── dc_run.tcl_bkp_1207
├── dc_run.tcl_org
├── default_config.sh
├── dont_use.txt
├── RAM_1C_W0_R0_synthesis_file_set.txt
├── RAM_1C_W0_R0_synthesis_file_set.txt_old
└── syn_launch.sh

```

CONSTRAINTS

```

├── NV_NVDLA_partition_a.macroplacement.tcl
├── NV_NVDLA_partition_a.sdc
├── NV_NVDLA_partition_c.macroplacement.tcl
├── NV_NVDLA_partition_c.path_group.tcl
├── NV_NVDLA_partition_c.sdc
├── NV_NVDLA_partition_m.sdc
├── NV_NVDLA_partition_o.macroplacement.tcl
├── NV_NVDLA_partition_o.path_group.tcl
├── NV_NVDLA_partition_o.sdc
├── NV_NVDLA_partition_o.tcl
├── NV_NVDLA_partition_p.macroplacement.tcl
└── NV_NVDLA_partition_p.sdc

```

Figure 12. The directory structure for logic synthesis [6].

In the provided scripts the design is partitioned into 5 independent sub-designs to execute synthesis. NV_NVDLA-partition_* are a top-level synthesis hierarchies thus the design is compiled accordingly. [6]. The provided synthesis partition setup was exploited in this project to fasten the implementation process.

To run synthesis the synthesis configuration needs to be setup. The *default_config.sh* file can be used as a reference. The most important design option variables include:

1. TOP_NAMES: It is a list of top-level partitions to be synthesized such as, NV_NVDLA_partition_a NV_NVDLA_partition_c. [6].
2. RTL_SEARCH_PATH: Indicates the directory location of the whole RTL design. [6].

3. DEF: Directory path to .def files incorporating the floorplans. These are created with the ICC2 tool and named with according to the TOP_NAMES for example as NV_NVDLA_partition_p.def. [6].
4. CONS: Directory path to .sdc files. Files in SDC format are named according to the TOP_NAMES for example as NV_NVDLA_partition_m.sdc. Constraints can be specified in .tcl format to guide the synthesis run. [6]. Synthesis runs were performed with both default constraints and additional constraints in TCL format to relax setup timing.
5. TARGET_LIB: Directory path to a standard cell library utilized during the mapping [6].
6. LINK_LIB: Directory path to the libraries utilized when the design is linked. Should also contain the target library and RAM compiler timing libraries. [6].
7. NDM_LIB: Directory path to the files on new data model format. These are libraries containing technology specific information about the logical and physical characteristics of the particular technology. NDM mode can be utilized in the Design Compiler NXT topographical mode. [9].
8. TF_FILE: Directory path to the Milkyway Technology File which is utilized to generate the Milkyway models for the physical libraries [6].
9. TLUPLUS_FILE: Directory path to the TLUPlus files needed for the RC estimations [6].
10. TLUPLUS_MAPPING_FILE: Directory path to the Tech2ITF file. It is used to connect layer names between the Milkyway Tech file and the interconnect technology format file. [6].
11. MIN_ROUTING_LAYER: Indicates the bottom routing layer that can be used for the signal nets [6].
12. MAX_ROUTING_LAYER: Indicates the top routing layer that can be used for the signal nets [6].
13. HORIZONTAL_LAYERS: Involves a list of layers primarily dedicated for horizontal routing [6].
14. VERTICAL_LAYERS: Involves a list of layers primarily dedicated for vertical routing [6].
15. DONT_USE_LIST: Involves a list of common cell expressions that are not desired to be mapped to the design [6].
16. COMMAND_PREFIX: Variable containing a string. It manages the synthesis runs on LSF server. [6]. The used string is "bsub -q b_soc_rh7 -R "rusage[mem=16000]"".

Synthesis constraints include clock constraints and physical constraints. The clock constraints are specified in SDC file for each partition, which are set to the CONS variable. Additional constraints can be also set in TCL format. On the actual synthesis run the provided default constraints were utilized with additional constraints specified for this project. Physical constraints are used in physical synthesis run and they are specified in DEF file. Physical constraints contain placement information about RAM and I/O defined by the designer. These are set to the DEF variable. [6].

On NVDLA environment synthesis is run by using the `syn_launch.sh` script file. Run options include:

1. `-config`: Directory path to the configuration file which contains all the required variables for libraries utilized during synthesis run. This is the `default_config.sh` file. [6].
2. `-mode`: Set the used synthesis tool. Options include `wlm` which is used for DC non-topographical and non-physical synthesis, `dct` which is used for DC-Topographical, `dcb` which is used for DC-Graphical together with additional `-spg` argument in the compile command, and `de` which is used for DC Explorer. [6]. the `dct` mode was used in actual synthesis runs on the project.
3. `-build`: Creates output directory for a synthesis run as `nvdla_syn_<timestamp>` [6].
4. `-modules`: A list of modules i.e., `TOP_NAMES` that will be processed in synthesis run [6].

To run physical synthesis by DC-Topographical the following string was executed `${NVDLA_ROOT}/syn/dc/scripts/syn_launch.sh -mode dct -config /path/to/default_config.sh`. Also at least the next variables have to be setup in `default_config.sh` file `TARGET_LIB`, `LINK_LIB`, `MW_LIB`, `DC_PATH`, `TF_FILE`, `TLUPLUS_FILE`, `TLUPLUS_MAPPING_FILE`, `MIN_ROUTING_LAYER`, and `MAX_ROUTING_LAYER`. Furthermore, `HORIZONTAL_LAYERS` and `VERTICAL_LAYERS` may be setup if the physical libraries need that information. Eventually, the NVDLA synthesis environment creates output file structure which is shown in Figure 13.

```
<BUILD>
|--- fv
|   `--- NV_NVDLA_partition*
|         `--- NV_NVDLA_partition*.svf
|--- net
|   |--- NV_NVDLA_partition*.gv (Mapped Netlist)
|   |--- NV_NVDLA_partition*.full.def (complete output DEF)
|   `--- NV_NVDLA_partition*.sdc (Output SDC)
|--- db
|   `--- NV_NVDLA_partition*.ddc (Synthesis design Database)
|       (There are also a few intermediate design databases here)
`--- report
    |--- NV_NVDLA_partition*.check_design
    |--- NV_NVDLA_partition*.check_timing
    `--- NV_NVDLA_partition*.final.report
        (Detailed timing/QoR information)
        (There are also reports generated at intermediate stages)
```

Figure 13. Output files of a synthesis run [6].

The `fv` directory contains `.svf` files for each partition to run formal verification with the Synopsys Formality tool. The `net` directory contains for example the mapped netlist. The

db directory contains the final and intermediate synthesis design database files for each partition. The *report* directory contains the essential report files of the synthesis run for all the partitions. The *.final.report* files contain the Quality of Report (QoR) information which are examined in the results section of this thesis. [6]. This concludes the synthesis flow discussions related to NVDLA environment.

To present comprehensive description of the implementation flow the IC Compiler 2 tool is considered on discussions. The executed flow is shown in Figure 14.

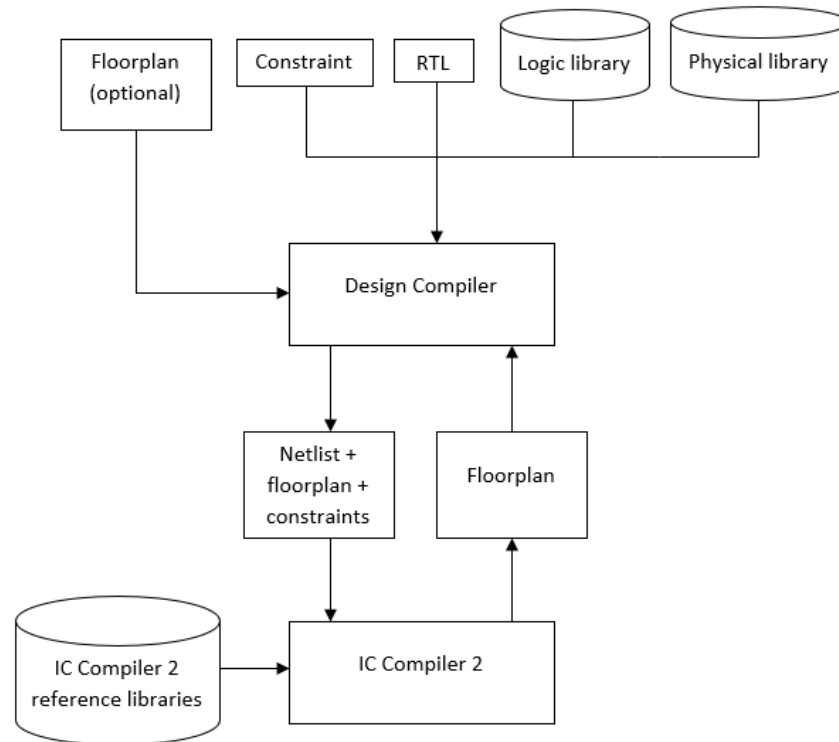


Figure 14. The floorplan exploration flow from DC to ICC 2 [9].

The initial logic synthesis run requires the constraints, RTL, logic library, and physical library as an input. The floorplan is not yet created so it is optional. DC performs the logic synthesis process and generates the netlist, floorplan, and constraints. At this stage the automatically generated floorplan is not very efficient thus the floorplan is created manually with IC Compiler 2. This means placing the macros and I/O ports and storing the placement information in a .def file. Now the floorplan can be included into the synthesis run and design compiler can perform the synthesis process again from a better starting point. Hence, the output netlist is more efficient compared to the initial output. The described flow was performed in the thesis project. Next, the System is discussed in more detail.

4.3 Detailed system description

The NVDLA system is divided into five partitions each performing dedicated functionality. The basic top-level structure and data exchange between partitions is shown in Figure 15.

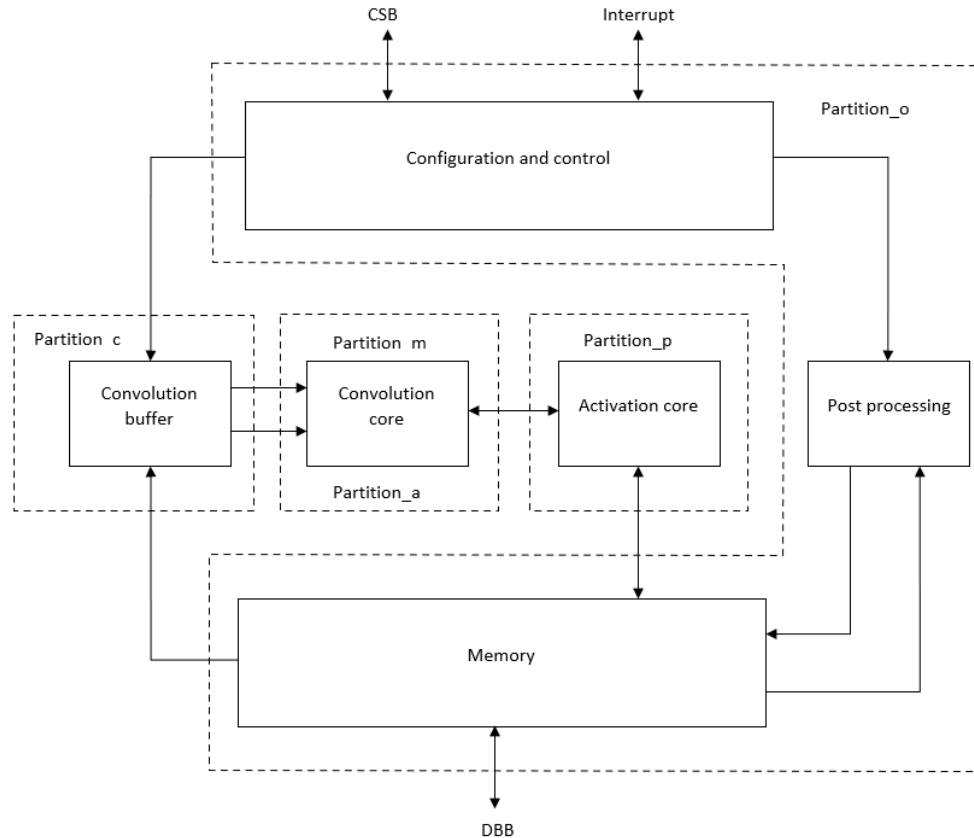


Figure 15. NVDLA top-level structure [10, 28].

The partitions include:

1. Partition_o: This partition manages the communication in the system between the processing components. The functionality includes *CSB master*, *AXI interfaces*, *bridge DMA*, *Rubik engine*, *Cross-channel data processor*, *Planar data processor* and *Global unit*. [14].
2. Partition_c: This partition controls some convolution kernel operations. The functionality includes *Convolution DMA*, *Convolution buffer* and *Convolution sequence controller*. [14].
3. Partition_m: This partition executes multiplication and addition calculations. The functionality includes *Convolution MAC array*. [14].
4. Partition_a: This partition collects all the partial sums generated in the MAC array and evaluates the outcomes prior passing them to the following activations step. The functionality includes *Convolution accumulator*. [14].
5. Partition_p: This partition executes some linear and non-linear operations. The functionality includes *Single data processor*. [14].

Next sections describes the physical structure of the partitions in detail.

4.3.1 Partition_m

The partition m is part of the convolution core. The post-synthesis layout is visualized in Figure 16.

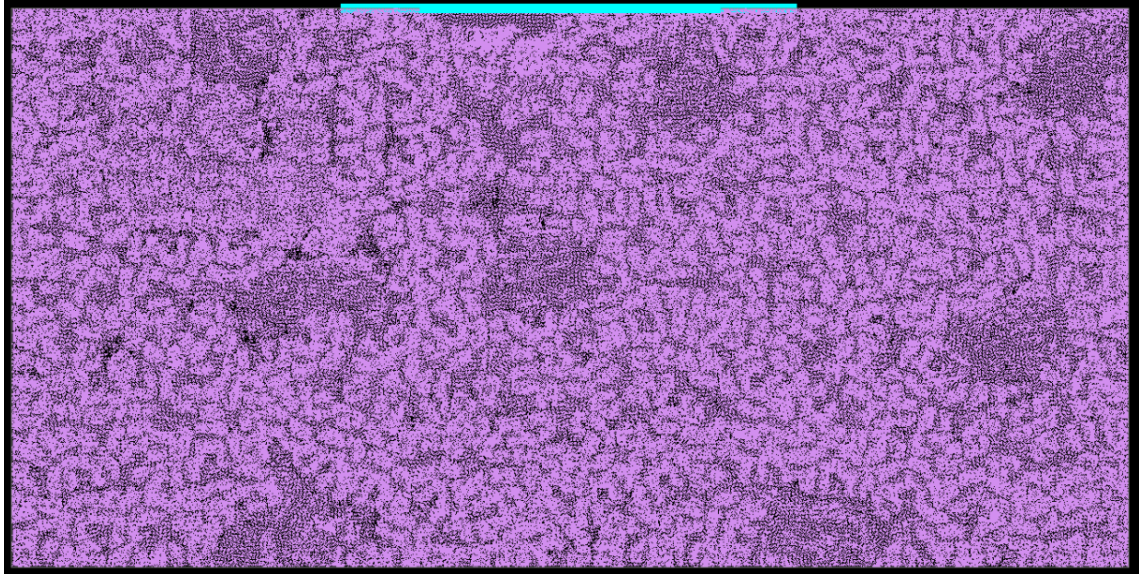


Figure 16. *Layout picture of partition m.*

On the layout the I/O's are placed on the top of the core shown as narrow turquoise rectangle in the Figure 16. In this way the rest of the core area is left for standard cells which are shown as purple fog. The leaf cell count is 894 106 and there are no macros i.e., RAMs in partition m. The design area is 133 097,7002 μm^2 . Next, the layout of partition_o is presented.

4.3.2 Partition_o

Partition_o is the controlling part of the design. The post-synthesis layout is visualized in Figure 17.

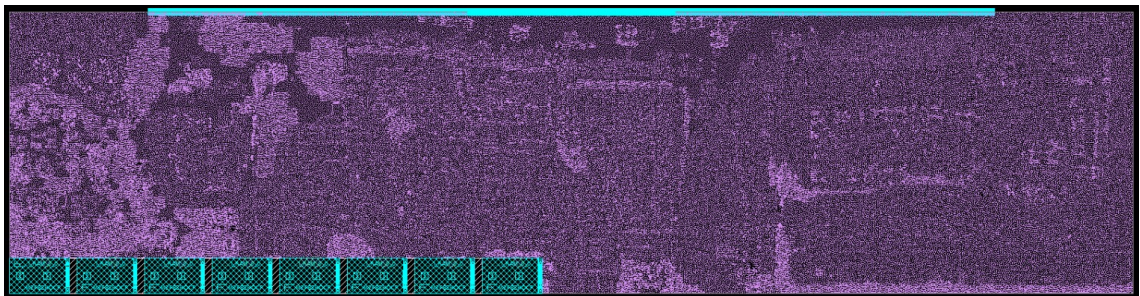


Figure 17. *Layout picture of partition o.*

Similar to partition_m the I/O's are placed on top of the core also on partition_o. The difference is that partition_o has 8 macro instances, they are the turquoise boxes at the bottom left corner. Usually, in industry the memory macros are placed on a U-shape to

give as much uniform space for the standard cells as possible. The placement is started from the bottom left corner and the macros are placed together hierarchically; in this case the macro count is low, therefore they don't form the U-shape. However, this way the routing between the logic elements is usually better. For the same reason, buffer only blockages are set on the gaps between the memories visible in Figure 17. Also, hard blockages are placed on top of the memories to make sure that standard-cells are not placed on top of the memories during synthesis run. Standard-cells are the purple fog. Just by looking at the Figure 17 we can say that partition_o has less cells than partition_m. The leaf cell count in partition_o is only 291 905 and the design area is 68339,4530 μm^2 . Next, the layout of partition_p is presented.

4.3.3 Partition_p

Partition_p is the activation core. The post-synthesis layout is visualized in Figure 18.

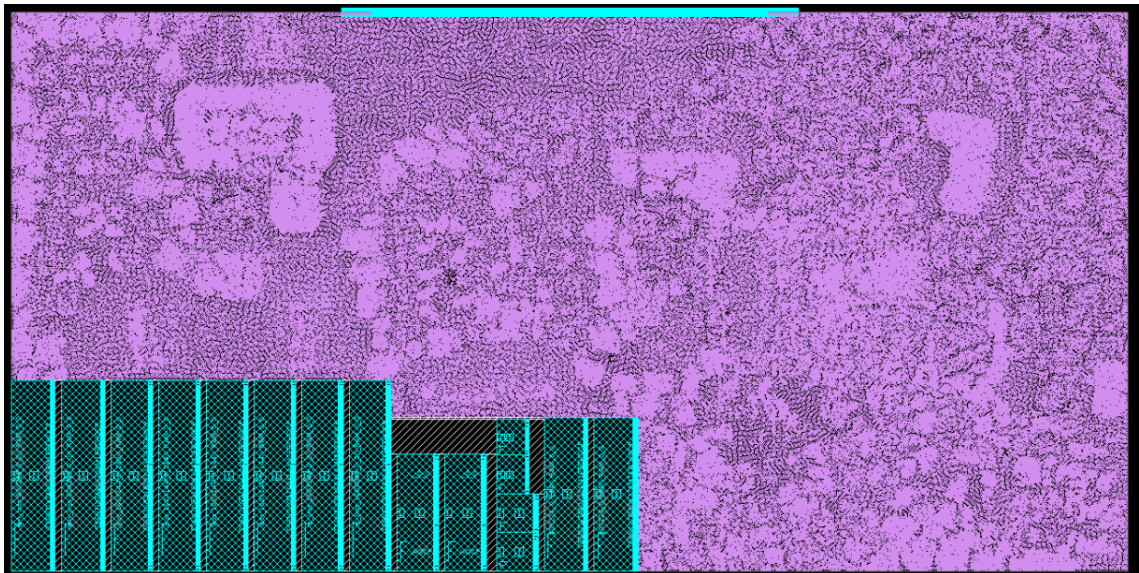


Figure 18. *Layout picture of partition p.*

The applied placing principles are the same as on partition_m and partition_o. I/O's are on top, memory placement is started from bottom left corner (blockages in between and on top), and standard-cells are in between I/O's and memories. Leaf cell count is 859 129 and it has 16 macro instances. Design area is 194 218,8532 μm^2 . Next, the layout of partition_a is presented.

4.3.4 Partition_a

Partition_a is the other part of the convolution core. The post-synthesis layout is visualized in Figure 19.

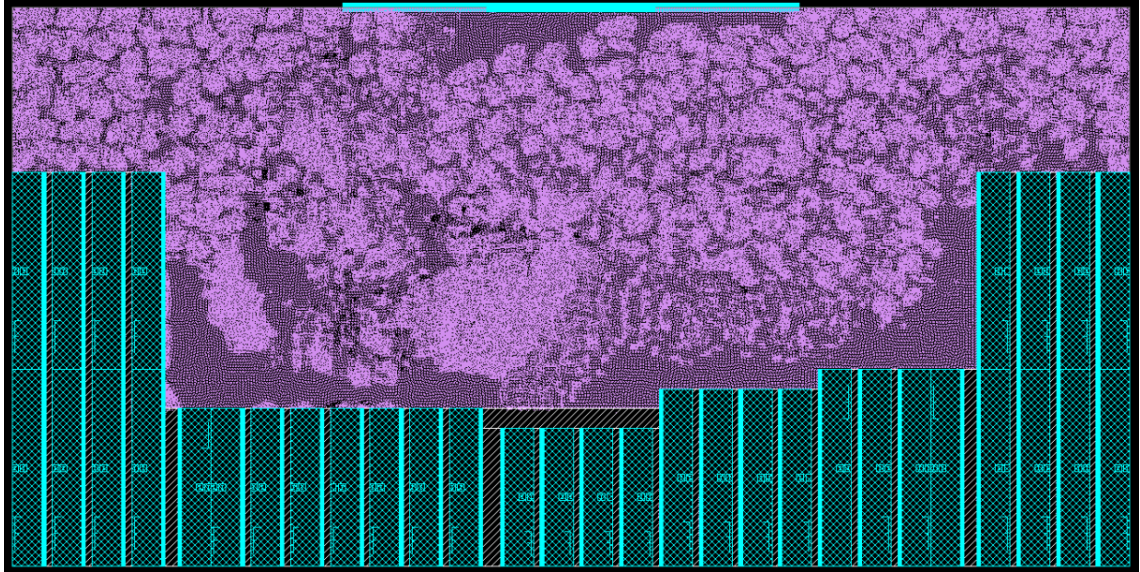


Figure 19. *Layout picture of partition a.*

The placement on partition_a is implemented with same principles as on the previous partitions. Nonetheless, the difference is that on partition_a the U-shape placement of the memories is visible in Figure 19. Leaf cell count is 470 830 and it contains 36 macro instances. Design area is 190 734,8399 μm^2 . Finally, the layout of partition_c is presented in next section.

4.3.5 Partition_c

Partition_c is the convolution buffer. The post-synthesis layout is visualized in Figure 20.

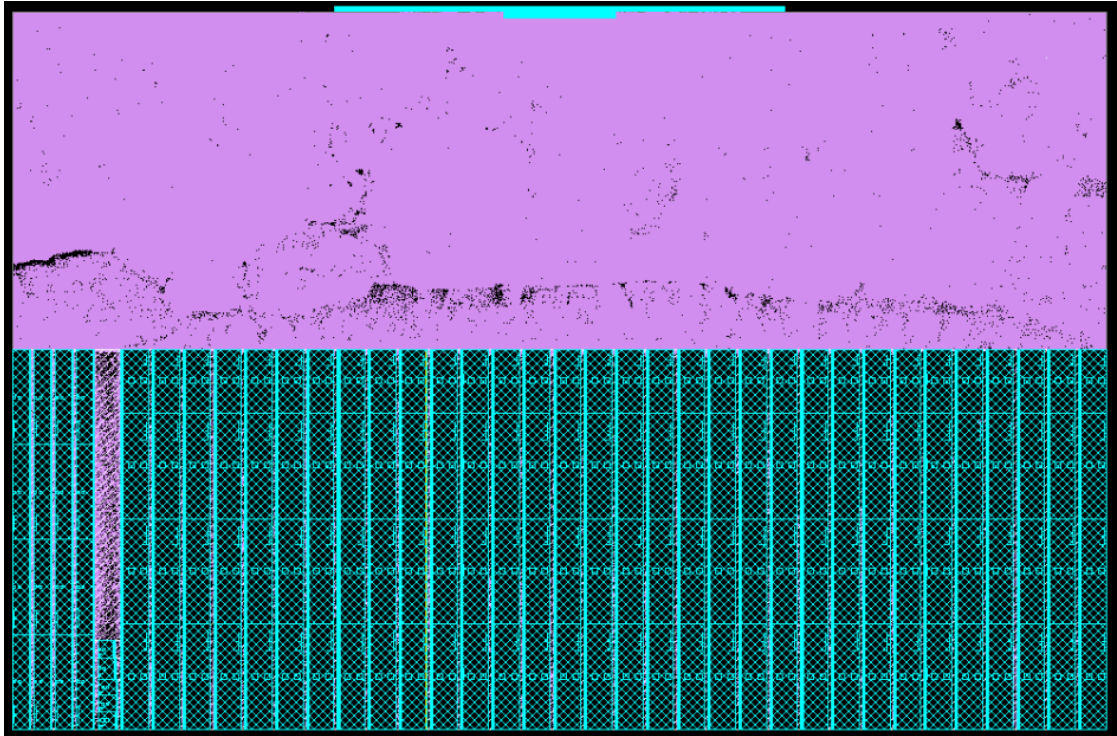


Figure 20. *Layout picture of partition c.*

Previously explained placements principles are applied in partition_c as well. However, it is different to other partitions, it is memory dominated with 149 macro instances. From Figure 20 we can also see that the purple standard-cell area does not look foggy in partition_c, because cells are placed much denser compared to other partitions. Since, the memories require more space. Leaf cell count is 931 135. Design area is 863205,5468 μm^2 of which memory macros cover 724 145,7469 μm^2 . Next, the used memory wrapper is discussed.

4.3.6 Memory wrapper

The NVDLA core and TSMC RAMs need to be connected with some method to enable communication between them. To achieve this a memory wrapper is created with company internal tool. The tool is able to generate a memory wrapper for single-port, two-port or dual-port 7 nm RAMs [13]. Basic configuration of a memory wrapper is visualized in Figure 21.

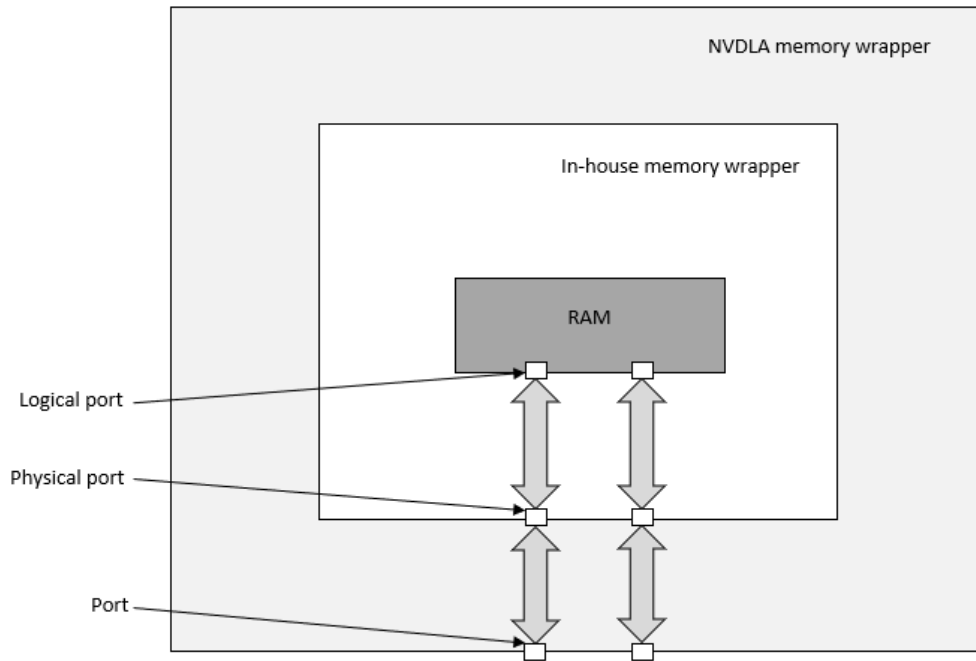


Figure 21. *Memory wrapper.*

In general, a memory wrapper system is a set of virtual elements which are interconnected through channels. The wrapper integrates the memory interface to the communication network. The system is setup so that from the physical point of view memory accesses are divided into logical accesses. To integrate the logical and the physical interface and to access the behaviour, the interface contains two types of ports: logical and physical ports. Logical ports enable the logical access to the memory and physical ports enables the physical access to the memory. This allow the distinct behaviour and interface which increases the flexibility level in the design. Since, the memory implementation can be decided later in the design process. [12].

5. SYNTHESIS RESULTS

This section discusses the post synthesis results. In this analysis results include power, area, and timing numbers. The results are presented independently for all the NVDLA partition since, they are independent synthesis top-level hierarchies. The top-level integration model is missing; thus, no attempt is made to combine the partitions in the results. The Global operating voltage was 0,675 V in the synthesis run. First, the power results are presented.

5.1 Power

This section reports the power results for all the partitions individually. It summarizes the internal power, switching power and leakage power numbers of all the functional elements in the partition. Power results of partition_a are shown in Table 3.

Table 3. *Power results of partition_a.*

Partition_a Power Group	Internal Power mW	Switching Power mW	Leakage Power mW	Total Power mW	%
io_pad	0,0000	0,0000	0,0000	0,0000	0,00 %
memory	4,2592	5,0204	0,0098	9,2894	54,01 %
black_box	0,0000	0,0000	0,0000	0,0000	0,00 %
clock_network	0,3631	0,9654	0,0000	1,3285	7,72 %
register	4,3067	0,9273	0,0008	5,2348	30,44 %
sequential	0,0000	0,0000	0,0000	0,0000	0,00 %
combinational	0,1495	1,1939	0,0023	1,3457	7,82 %
Total	9,0784	8,1070	0,0130	17,1984	

From Table 3 we can see that memory consumes most of the total power in partition_a. If we further add registers to this consideration, they consume together over 84 percentage of the total power. The least power-hungry element is the clock network with combinational logic being in the same numbers. As one can expect most of the power is consumed in the dynamic power category. In the partition_a the internal power consumption and the switching power consumption are almost the same. The leakage power consumption is low in partition_a. The total power consumption of this partition is about 17,2 mW. Power results of partition_c are shown in Table 4.

Table 4. *Power results of partition_c.*

Partition_c Power Group	Internal Power mW	Switching Power mW	Leakage Power mW	Total Power mW	%
io_pad	0,0000	0,0000	0,0000	0,0000	0,00 %
memory	10,7914	12,4233	0,0558	23,2705	34,93 %
black_box	0,0000	0,0000	0,0000	0,0000	0,00 %
clock_network	0,8698	2,0236	0,1587	3,0521	4,58 %
register	10,1113	1,0236	0,0034	11,1383	16,72 %
sequential	0,0000	0,0000	0,0000	0,0000	0,00 %
combinational	3,8412	25,3195	0,0073	29,1680	43,78 %
Total	25,6137	40,9486	0,0667	66,6290	

From Table 4 we can see that in partition_c the combinational logic consumes most of the power almost 44 percentage of the total power. On contrast, the clock_network consumes the least amount of power. As in partition_a in partition_c most of the power is consumed as dynamic power. In partition_c the switching power is the highest power consumer. The leakage power consumption is minimal in partition_c. The total power consumption of the partition is about 66,6 mW which is the highest compared to other partitions. Power results of partition_m are shown in Table 5.

Table 5. *Power results of partition_m.*

Partition_m Power Group	Internal Power mW	Switching Power mW	Leakage Power mW	Total Power mW	%
io_pad	0,0000	0,0000	0,0000	0,0000	0,00 %
memory	0,0000	0,0000	0,0000	0,0000	0,00 %
black_box	0,0000	0,0000	0,0000	0,0000	0,00 %
clock_network	0,9224	1,6369	0,0001	2,5594	30,62 %
register	4,1849	0,3489	0,0010	4,5348	54,25 %
sequential	0,0000	0,0000	0,0000	0,0000	0,00 %
combinational	0,3006	0,9559	0,0088	1,2653	15,14 %
Total	5,4080	2,9417	0,0098	8,3595	

From Table 5 we can see that majority of the power is consumed in registers which is about 54 percentage of the total power consumption in partition_m. This partition does not have own memories which was the highest power user in previous partitions, but it performs MAC calculations thus the registers power usage is high. Combinational logic consumes the least amount of power in this partition. Internal operations cover most of the power consumption whereas leakage power is minimal also in partition_m. The total power consumption of this partition is about 8,4 mW and it is the lowest compared to other partitions. Power results of partition_o are shown in Table 6.

Table 6. *Power results of partition_o.*

Partition_o Power Group	Internal Power mW	Switching Power mW	Leakage Power mW	Total Power mW	%
io_pad	0,0000	0,0000	0,0000	0,0000	0,00 %
memory	0,0524	0,0201	0,0008	0,0733	0,63 %
black_box	0,0000	0,0000	0,0000	0,0000	0,00 %
clock_network	0,5376	0,6655	0,0000	1,2031	10,28 %
register	7,4854	0,9434	0,0023	8,4311	72,03 %
sequential	0,0000	0,0000	0,0000	0,0000	0,00 %
combinational	0,4878	1,5088	0,0012	1,9978	17,07 %
Total	8,5632	3,1378	0,0043	11,7053	

The results in Table 6 show that in partition_o registers power consumption is very high, those consumes over 72 percentage of the total power. Even though partition_o controls the memory, its own memory does not consume much power. As noted previously, also in this partition most of the power is consumed in internal cell operations and the leakage power consumption is minimal which is a good thing since energy is not wasted. The total power consumption of partition_o is about 11,7 mW. Finally, power results of partition_p are shown in Table 7.

Table 7. *Power results of partition_p.*

Partition_p Power Group	Internal Power mW	Switching Power mW	Leakage Power mW	Total Power mW	%
io_pad	0,0000	0,0000	0,0000	0,0000	0,00 %
memory	6,6264	1,3276	0,0043	7,9583	18,00 %
black_box	0,0000	0,0000	0,0000	0,0000	0,00 %
clock_network	0,7985	3,6920	0,0000	4,4905	10,15 %
register	20,4380	1,1150	0,0024	21,5554	48,74 %
sequential	0,0000	0,0000	0,0000	0,0000	0,00 %
combinational	1,3909	8,8235	0,0047	10,2191	23,11 %
Total	29,2538	14,9581	0,0114	44,2232	

The results in Table 7 show that registers cover about 49 percentage of the power consumption. On contrast, clock_network uses the least amount of power, although some electric current flow through it constantly. Internal operations consume most of the power while leakage power consumption is minimal. This partition has comparatively high total power consumption over 44 mW.

To summarize, we can note that convolution operations in the partition_c requires lots of power effecting to the total power consumption of the NVDLA core. Further, the single data processor in the activation core implemented in partition_c has the biggest effect on the total power consumption. The convolution core is not too power-hungry element

in the design. It is implemented in partition_a and partition_m and the summed total power consumption is only, just over 25 mW. By just looking the Figure 15 you could expect that managing the system requires lots of power but actually as the power numbers show partition_o has only 11,7 mW total power consumption, which is actually quite obvious since it does not perform heavy computations. The leakage power consumption is managed properly in all the partitions i.e., it is low in each partition. Most of the power is consumed as of dynamic power where the cell internal power is the higher consumer of energy in most of the partition except in the partition_c where the switching power has the highest number. The next section discusses the area results of the NVDLA partitions.

5.2 Area

This section discusses the area related results of each partition. All the area and cell count results are shown in following tables. Area results are collected to Table 8.

Table 8. *Area results of all the partitions.*

Area / μm^2	Partition_a	Partition_c	Partition_m	Partition_o	Partition_p
Combinational Area	40325,00	88075,21	113601,18	16444,84	83260,54
Non-combinational Area	27038,85	50984,59	19496,52	44834,09	54212,83
Buf/Inv Area	4068,36	17733,08	6353,46	1200,95	6046,25
Total Buffer Area	681,15	10506,35	1659,11	793,44	1650,29
Total Inverter Area	3387,21	7226,73	4694,35	407,51	4395,96
Macro/Black Box Area	123370,99	724145,75	0,00	7060,52	56745,48
Net Area	0,00	0,00	0,00	0,00	0,00
Net XLength	4808959,50	14996478,00	4519846,00	2566439,50	4771879,00
Net Ylength	3838447,75	17214730,00	3422268,00	1208921,25	3583880,50
Cell Area	190734,84	863205,55	133097,70	68339,45	194218,85
Design Area	190734,84	863205,55	133097,70	68339,45	194218,85
Net Length	8647407,00	32211208,00	7942114,00	3775360,75	8355759,50

Partition_a is mid-sized compared to other partitions. The combinational area is 40325 μm^2 , non-combinational area is 27038,85 μm^2 and macro area is 123370,99 μm^2 together these form the cell area 190734,84 μm^2 which is actually the overall design area. This area also includes the buffer and inverter area. The total inverter area in partition_a is 3387,21 μm^2 and the total buffer area is 681,15 μm^2 thus the buf/inv area is 4068,36 μm^2 . Partition_c is the largest of the partitions. In partition_c the combinational area is 88075,21 μm^2 , non-combinational area is 50984,59 μm^2 and macro area is

724145,75 μm^2 together these form cell area of 863205,55 μm^2 . The high area number is consequence of RAMs in the partition as we can see from the macro area number. It has also the largest area of inverters and buffers, the total inverter area is 7226,73 μm^2 and the total buffer area is 10506,35 μm^2 thus the buf/inv area is 17733,08 μm^2 . Partition_m is one of the smaller partitions with cell area of 133097,7 μm^2 . It does not have any RAMs hence the combinational area is the largest (113601,18 μm^2) compared to other partitions. As a consequence, the non-combinational area is the smallest (19496,52 μm^2). The major difference to other partition is that partition_m does not have any RAMs hence the macro area is 0 μm^2 . In partition_m the total inverter area is 4694,35 μm^2 and the total buffer area is 1659,11 μm^2 thus the buf/inv area is 6353,46 μm^2 . Partition_o is the smallest partition with 68339,45 μm^2 cell area. The small size results from the combinational area which is only 16444,84 μm^2 . Non-combinational area is 44834,09 μm^2 and macro area is 7060,52 μm^2 which is the smallest compared to partitions that have RAMs. Also, total inverter area (407,51 μm^2) and total buffer area (793,44 μm^2) are the smallest in partition_o hence the buf/inv area is the smallest 1200,95 μm^2 . Lastly, the mid-sized partition_p has cell area of 194218, 85 μm^2 . This is the outcome of combinational area (83260, 54 μm^2), non-combinational area (54212,83 μm^2) which highest compared to other partitions, and macro area (56745, 48 μm^2). Total inverter area is 4395,96 μm^2 and total buffer area is 1650,29 μm^2 thus the buf/inv area is 6046,25 μm^2 . Next, the cell count results are discussed. The results are show in Table 9.

Table 9. *Cell count results of all the partitions.*

Cell Count	Partition_a	Partition_c	Partition_m	Partition_o	Partition_p
Hierarchical Cell Count	1067	2759	1865	1348	2552
Hierarchical Port Count	4268	11036	7460	5392	10208
Leaf Cell Count	470830	931135	894106	291905	859129
Buf/Inv Cell Count	68921	204188	104069	16809	98248
Buf Cell Count	8256	100533	20045	9605	19411
Inv Cell Count	60665	103655	84024	7204	78837
CT Buf/Inv Cell Count	0	0	0	0	0
Combinational Cell Count	381252	768265	829455	144298	688702
Sequential Cell Count	89578	162870	64651	147607	170427
Macro Count	36	149	0	8	16

First, cell count results of partition_a are examined. Combinational cell count is 381252 and sequential cell count is 89578 together resulting leaf cell count of 470830 cells which

is not a big number compared to other partitions. As the area results already indicated partition_a has much more inverters (60665) than buffers (8256) resulting buf/inv cell count to be 68921 which is also quite small number compared to other partitions. Macro count in partition_a is 36, this is visualized in Figure 19. Partition_c is the densest from the cell count perspective. It has the highest leaf cell count (931135) resulting from combinational cell count of 768265 cells and from sequential cell count of 162870 cells. It has also the highest inverter cell count (103655) and buffer cell count (100533) thus the buf/inv cell count is the highest 204188 cells. Partition_c has 149 macros making it macro dominated from the layout perspective. This is visualized in Figure 20. Partition_m has almost as high leaf cell count (894106) as partition_c. This is mainly the outcome of combinational cell count (829455) which is the highest compared to other partitions. On contrast, the sequential cell count (64651) is the lowest in the design. As noted in the area section partition_m does not have any RAMs thus the macro count is 0. This is visualized in Figure 16. Partition_o has the lowest leaf cell count (291905), it also has the lowest combinational cell count (144298), and the sequential cell count is 147607. Accordingly, the inverter cell count (7204) and the buffer cell count (9605) are the lowest compared to other partition thus the buf/inv cell count (16809) is the lowest. The macro count of partition_o is 8 which is also shown in Figure 17. Lastly, the cell count numbers of partition_p are represented. Partition_p has high leaf cell count (859129) from these combinational cells are 688702 and sequential cells are 170427. The sequential cell count is the highest in partition_p compared to other partitions. Inverter cell count is 78837 and buffer cell count is 19411 thus buf/inv cell count is 98248. Partition_p has 16 macros which are shown in Figure 18. This concludes the area section of the results. Next, the timing results are discussed.

5.3 Timing

This section discusses about the timing results, reported after successful synthesis run. The setup and hold timing results of all the partitions are shown in this section. After that summary of design rule count is presented. Lastly, all the timing path groups of all the partitions are examined individually. First from Table 10 we can examine the setup timing and the hold timing of all the partitions.

Table 10. *Setup and hold timing results of all the partitions.*

Timing ns	Partition a	Partition c	Partition m	Partition o	Partition p
Design WNS (Setup)	0,0000	0,1462	0,0000	0,0000	0,0000
TNS	0,0000	1148,8319	0,0000	0,0000	0,0000
Number of Violating Paths	0	28367	0	0	0
Design (Hold) WNS	0,1527	0,1127	0,0000	0,8553	0,1195
TNS	305,9194	15,8890	0,0000	2071,0925	2,8882
Number of Violating Paths	5669	1207	0	2647	37

More specifically Table 10 shows the worst negative slack (WNS), the total negative slack (TNS), and the number of violating paths in the design for both setup and hold timing. The upper half of Table 10 concerns the setup timing and the bottom half the hold timing. As shown the setup part contains only zeros for all the other partitions for WNS, TNS, and count of violating paths, the exception is partition_c which will be analysed later on upcoming sections. The zeros indicate that the synthesis run was successful i.e., the setup time constraints are met, and the design process can now proceed to the place and route phase. Even if, the hold timing is not met, except for the partition_m. The slacks and the count of violating paths for the hold timing part need to be decreased to zero to be sure that the design operates as desired. However, these can be ignored for now thus they will change on the place and route phase. If hold time violations still exist after the place and route, then those have to be fixed. Types of design rule violations are summarized in Table 11.

Table 11. *Design rule violation types of all the partitions.*

Design Rule Violation Types	Partition_a	Partition_c	Partition_m	Partition_o	Partition_p
Total Number of Nets	488940	967816	1009711	295966	907475
Nets with Violations	21	287	93	2	48
Max Trans Violations	21	287	92	2	44
Max Cap Violations	0	0	1	0	9

In Table 11 we can see total number of nets, number of violations in the nets, and the type of violation either transition or capacitance violation. Partition_m has the highest number of nets whereas partition_o has the lowest number of nets. All the partitions have

some kind of violations. The highest violation count is in partition_c and the lowest in partition_o. Violations related to transitions exist in all the partitions. Whereas capacitance related violations exist only in partition_m and partition_p. Next, factors related to the timing paths and the clocks in the design are discussed. The results include for example number of logic levels, critical path characteristics, and violation characteristics of all the timing paths in the NVDLA. Timing paths in partition_a are shown in Table 12.

Table 12. *Partition_a: timing paths.*

Partition_a, ns	Timing Path Group: clock_gating_default	Timing Path Group: nvdla_core_clk
Levels of Logic	2	34
Critical Path Length	0,7574	0,9069
Critical Path Slack	0,0169	0,0000
Critical Path Clk Period	0,9000	0,9000
Total Negative Slack	0,0000	0,0000
No. of Violating Paths	0	0
Worst Hold Violation	0,0000	-0,1527
Total Hold Violation	0,0000	-305,9188
No. of Hold Violations	0	5669

From Table 12 we can see that it has 2 timing path groups from which the *nvdla_core_clk* is the actual clock signal of the partition and the *clock_gating_default* is additional path group to help the synthesis meet the timing constraints. *Clock_gating_default* has 2 logic levels, and it is clear of violations. We can also determine the maximum clock rate from the critical path length (0,7574 ns), it is $f = 1 / T = 1,3203$ GHz. The *nvdla_core_clk* has 34 logic levels and it has hold violations, but these can be ignored at this point. However, to analyse the hold violations a bit we can see that worst hold violation and also the total hold violation has a negative number implying that the data path is slow compared to clock signal which is 0,9 ns. The critical path is 0,9024 ns also indicating the slow path; thus, it needs to be reduced. Timing paths in partition_c are shown in Table 13.

Table 13. *Partition_c: timing paths.*

Partition_c, ns	Timing Path Group: clock_gating_default	Timing Path Group: nvdla_core_clk	Timing Path Group: from_mem	Timing Path Group: to_mem	Timing Path Group: clk_gaters
Levels of Logic	40	43	6	31	15
Critical Path Length	0,9894	1,0370	0,9416	0,8012	0,5619
Critical Path Slack	-0,1455	-0,1462	-0,0364	-0,0408	-0,1207
Critical Path Clk Period	0,9000	0,9000	0,9000	0,9000	0,9000
Total Negative Slack	-38,0323	-995,4911	-56,1600	-56,3569	-2,9106
No. of Violating Paths	566	20697	4120	2955	43
Worst Hold Violation	0,0000	0,0000	0,0000	-0,1127	0,0000
Total Hold Violation	0,0000	0,0000	0,0000	-15,8890	0,0000
No. of Hold Violations	0	0	0	1207	0

From Table 13 we can see that partition_c contains 5 timing path groups, which are *clock_gating_default*, *nvdla_core_clk*, *from_mem*, *to_mem*, and *clk_gaters*. The *nvdla_core_clk* is the main clock timing group and the other 4 path groups are segregated to help the synthesis to optimize timing based on priorities. As Table 13 shows all the timing path groups have setup timing violations but only the *to_mem* path involves hold time violations. Generally, hold time violations are not considered during synthesis because those will change in place & route stage and usually those are also fixed at that stage. However, setup time violations should be fixed already at synthesis stage. Timing path groups in partition_c contains in total 28367 paths under setup violations. To understand the reasons behind violations, the layout is visualized in Figure 20. Partition_c is memory intensive block; it has 149 RAM instances integrated to the design through the memory wrappers. The desired configurations for memory layout is U-shape (commonly used in industry), to give the standard cells more space as possible. However, the memory hierarchies have to be placed together thus the layout is not really a U-shape in partition_c. Hence, one of the reasons to the existence of setup violations is related to the memory placement. By analysing timing paths from the QoR report we can find out that the worst violators are paths travelling to the memory instances on the bottom of the layout. This issue could be fixed by exploring different kind of memory placements, which can be time consuming process i.e., more iterative thus it is not performed on this project. Another reason behind setup violations in the main clock path group can be due to incompatible RTL design to the 7 nm technology. Since, the NVDLA is initially targeted to 22 nm standard-cell technology. On this project it is implemented to 7 nm standard-cell technology, the RTL model does not always fit directly to different technology. Also because of this, the memory instances (7nm) are divided into multiple instances to match the design memory configurations that was originally present in the design (22nm). This may cause the long timing paths which are of course not expected in the initial NVDLA model. Looking at the *from_mem* path group we see that about 70

percentage of the time in the clock period in the timing are used up inside the memories. Hence, the timing could be improved by utilizing high speed memories from the technology library. Remaining path group is about `clk_gaters`, the worst critical path slack is usually analysed after clock tree synthesis stage in the Place and route and iterated there to fix those timing paths. At synthesis there isn't much that one can do to fix these clock gaters as clocks are in their ideal stage (not propagating). Timing paths in `partition_m` are reported in Table 14.

Table 14. *Partition_m: timing paths.*

Partition_m, ns	Timing Path Group: clock_gating_default	Timing Path Group: nvdla_core_clk
Levels of Logic	4	36
Critical Path Length	0,7943	0,8955
Critical Path Slack	0,0000	0,0000
Critical Path Clk Period	0,9000	0,9000
Total Negative Slack	0,0000	0,0000
No. of Violating Paths	0	0
Worst Hold Violation	0,0000	0,0000
Total Hold Violation	0,0000	0,0000
No. of Hold Violations	0	0

Results in Table 14 show that `clock_gating_default` in `partition_m` has 4 logic levels. It is clear of violations and the critical path is 0,7943 ns from which we can derive the maximum clock rate to be 1,2590 GHz. In `partition_m` the `nvdla_core_clk` contains 36 levels of logic and does not have any violations varying from previous partition. The critical path is 0,8955 and thus the maximum clock rate is 1,1167 GHz. Timing paths in `partition_o` are reported in Table 15.

Table 15. *Partition_o: timing paths.*

Partition_o, ns	Timing Path Group: clock_gating_default	Timing Path Group: u_NV_NVDL A_cvif	Timing Path Group: nvdla_core_clk	Timing Path Group: nvdla_falcon_clk
Levels of Logic	41	29	29	3
Critical Path Length	0,8456	0,9069	0,7471	0,3499
Critical Path Slack	0,0000	0,0000	0,0000	1,1332
Critical Path Clk Period	0,9000	0,9000	0,9000	1,5160
Total Negative Slack	0,0000	0,0000	0,0000	0,0000
No. of Violating Paths	0	0	0	0
Worst Hold Violation	0,0000	-0,8553	-0,8553	-0,6837
Total Hold Violation	0,0000	-43,2998	-2024,5063	-3,2931
No. of Hold Violations	0	54	2588	5

Results in Table 15 show that `partition_o` has 4 timing path groups. From these the `nvdla_core_clk` and the `nvdla_falcon_clk` are actual clock signals. The difference to other

partitions is the `nvdla_falcon_clk` which does not exist in other partitions, it is used for the system control purposes which are performed in `partition_o`. The other 2 timing path groups are the `clock_gating_default` and `u_NV_NVDLA_cvif` which are additional path groups to help the synthesis to meet the timing constraints. The `clock_gating_default` has 41 logic levels, and is clear of violations. the critical path is 0,8456 ns and the maximum clock rate is 1,1826 GHz. The `u_NV_NVDLA_cvif` has 29 logic levels. It has hold violations which can be ignored as stated before. The critical path length is 0,9069 ns exceeding the clock period length which is 0,9 ns. The `nvdla_core_clk` has 29 logic levels, and has some hold violations which are ignored. The critical path is 0,7471 ns and the maximum clock rate is 1,3385 GHz. `Partition_o` contains also the `nvdla_falcon_clk` it has 3 logic levels and some hold violations which are ignored. The critical path length is 0,3499 ns and thus the maximum clock rate is 2,8579 GHz. The actual clock period in `nvdla_falcon_clk` is set to 1,5160 ns differing from the `nvdla_core_clock`. Finally, timing paths in `partition_p` are shown in Table 16.

Table 16. *Partition_p: timing paths.*

Partition_p, ns	Timing Path Group: clock_gating_default	Timing Path Group: nvdla_core_clk
Levels of Logic	23	24
Critical Path Length	0,8453	0,8908
Critical Path Slack	0,0000	0,0000
Critical Path Clk Period	0,9000	0,9000
Total Negative Slack	0,0000	0,0000
No. of Violating Paths	0	0
Worst Hold Violation	0,0000	-0,1195
Total Hold Violation	0,0000	-2,8882
No. of Hold Violations	0	37

From Table 16 we can note that `clock_gating_default` has 23 logic levels, and it is clear of violations. The critical path is 0,8453 ns and the derived maximum clock rate is 1,1830 GHz. The `nvdla_core_clk` has 24 levels of logic and involves hold violations which does not require further analysis at this point of the design process. The critical path is 0,8908 ns and thus the maximum clock rate is 1,1226 GHz.

To summarize the timing result we can say that the target to have design clear of setup violations after synthesis has been achieved for all the other partitions except for the `partition_c`. The layout of `partition_c` would need more explorations to get it clear of setup violations. Also, the hold timing part needs to be improved, all the other partitions have hold timing violations except the `partition_m`. However, on industrial ASIC process the hold time violations are usually fixed on the place & route phase, which is executed after the logic synthesis. This concludes the timing report part.

5.4 Comparison to literature

NVDLA synthesis results are compared to 5 other SoC designs from the literature. The results cannot be compared directly to other studies, since NVDLA is just a hardware accelerator, and the reference designs are complete SoC. Also, the results are presented only for the NVDLA partitions since the top-level synthesis model is not created on this project. However, they provide an idea about the magnitude of the results and may provide important information for the further development in future. Summary of the synthesis results is presented in Table 17.

Table 17. *Summary of NVDLA partitions.*

	Partition_a	Partition_c	Partition_m	Partition_o	Partition_p
Technology	7 nm	7 nm	7 nm	7 nm	7 nm
Area	0,19mm ² (Cell Area)	0,86mm ² (Cell Area)	0,13mm ² (Cell Area)	0,07mm ² (Cell Area)	0,19mm ² (Cell Area)
On-chip SRAM	-	-	-	-	-
Max core frequency	1,32 GHz	-	1,26 GHz	1,18 GHz	1,18 GHz
Bit precision	8b	8b	8b	8b	8b
Num. of MACs	2048 Total in NVDLA (8b)	2048 Total in NVDLA (8b)	2048 Total in NVDLA (8b)	2048 Total in NVDLA (8b)	2048 Total in NVDLA (8b)
Power consumption	17 mW	67 mW	8 mW	12 mW	44 mW

From Table 17. the benefit of the smaller technology can be seen on all the metrics. Especially the max core frequency is significantly greater on NVDLA than on other designs in Table 18.

Table 18. *Reference designs.*

	Eyeriss [24]	ENVISION [32]	Thinker [33]	UNPU [34]	Eyeriss v2 [31]
Technology	65 nm	28 nm	65 nm	65 nm	65 nm
Area	12,25 mm ² (Core Area)	1,87 mm ² (Core Area)	19,35 mm ² (Die Area)	16 mm ² (Die Area)	2695k gates (NAND-2)
On-chip SRAM (kB)	181,5	144	348	256	246
Max core frequency	200 MHz	200 MHz	200 MHz	200 MHz	200 MHz
Bit precision	16b	4b/8b/16b	8b/16b	1b-16b	8b
Num. of MACs	168 (16b)	512 (8b)	1024 (8b)	13824 (bit-serial)	384 (8b)
Power consumption	236 mW	290 mW	447 mW	297 mW	460 mW

When comparing NVDLA results to the reference designs from Table 18. we can see that NVDLA is better on almost all the metrics. However, as noted earlier, the results are not directly comparable.

5.5 Analysis of the developed synthesis flow

The presented synthesis results show that the developed flow works, and it can generate good synthesis results. The generated results show detailed information about the characteristics of the used design which can be used to compare different IP blocks on ASIC design process. The integration of all the required elements NVDLA design, source scripts and constraints, RAM memory wrapper, and 7 nm standard cell technology files was accomplished after multiple synthesis iterations by utilizing the floorplan exploration flow from DC to ICC2. This proves that the company design environment can be used to run synthesis for open-source IP blocks. The developed flow provides a platform to exploit different kind of open-source IP's on industrial environment since, it can produce synthesis results for 7 nm standard cell technology node quickly. This concludes the results section.

6. CONCLUSIONS

Contemporary digital devices require high computing performance; thus, markets have a huge demand for SoC. However, big digital systems are very complex thus the systems are considered from different views. The most powerful SoCs are implemented on ASIC-chips. Usually, ASICs are designed with standard cells. ASIC is the most cost-efficient technology when production volumes are high. An important step on ASIC design process is the logic synthesis. By utilizing dedicated software tool, the logic synthesis process transfers RTL code into gate-level netlist, which is then used to design the actual chip from physical point of view. The process consists of 4 sub processes: RT-level synthesis, Module generator, Gate-level synthesis, and Cell-level synthesis. Usually, This is executed multiple times for a design at different RTL maturity phases. Since, it gives information about the performance of RTL design and can help to meet the specifications. Usually, the goal is to design a chip that is able operate on certain frequency. This is proven by using static timing analysis. The synthesis tool is run with TCL scripts. The scripts guide the tool to use correct optimization algorithms with the intended options i.e., switches. In DC this is performed in synthesize, optimize the design, analyse, and resolve design problems steps. By setting the optimization variables or design rule constraints properly the synthesis tool can improve the performance of the design up to 20 percentage.

The used design was NVDLA, it is an open-source deep learning accelerator developed by NVIDIA. It can execute CNNs which make it efficient. The main components in the design are Convolution core, single data processor, planar data processor, channel data, dedicated memory, and data reshape engines. Each component can be configured independently which make it very flexible. The system can be implemented as small- or large NVDLA system depending on the target application. This makes the system more cost effective. NVDLA software ecosystem has extensive cover of software features. The software includes the compilation tools and the runtime environment. NVDLA system software dataflow cover DL training, build (parsing, compiling, and optimizing), User-mode driver, and Kernel-mode driver.

To run synthesis the synthesis configuration needs to be setup. This includes variables such as, TOP_NAMES, RTL_SEARCH_PATH, DEF, CONS, TARGET_LIB, LINK_LIB etc., also run options need to be setup with appropriate switches. When all the constraints, RTL, logic library, and physical library files are setup the synthesis run can be executed. Usually, large system are divided into partitions. This is the case with

NVDLA, it has five partitions separated by their functionality. Partition_m is part of the convolution core. Partition_o contains the controlling functionalities. Partition_p is the activation core. Partition_a contains the other part of the convolution core. Partition_c is the convolution buffer. Each partition is an individual top-level synthesis hierarchy. Therefore, floorplan was created for each partition. To enable communication between NVDLA core and RAM instances a memory wrapper was generated.

The post synthesis results were extracted from QoR files. The power results show that convolution operation require lots of power and most of the power is consumed as of dynamic power. The area results show that the macro area has the biggest effect to the total area. This can be seen in partition_c which has 149 macro instances forming a macro area of 0,72 mm². From timing perspective, the goal is to have setup timing clean of violations at post synthesis phase. This is achieved for all the other partitions except for the partition_c. Usually, hold timing violations are not fixed at logic synthesis phase. All the other partitions have hold violations except the partition_m. The results are compared to other designs from literature: Eyeriss, ENVISION, Thinker, UNPU, and Eyeriss v2. The obtained results with 7 nm technology are promising but cannot be compared directly with the reference SoC designs.

The target of the thesis was to develop a logic synthesis flow for NVDLA in the company design environment. This was achieved by utilizing NVDLA design environment, company internal memory wrapper, and Synopsys Design Compiler and IC Compiler 2 tools to run logic synthesis for TSMC 7 nm standard cell technology. All the needed codes and scripts were downloaded from NVDLA GitHub webpage. The company memory wrapper tool was utilized to generate a memory wrapper which is able to connect the NVDLA design with needed RAM instances. The Design Compiler was used to generate the initial netlists for NVDLA partitions. To improve the results a floorplan was created for each partition with IC Compiler 2 tool. The generated DEF file was then used for second pass synthesis to obtain the final synthesis results. This proves that the company design environment can be used to run synthesis for open-source IP blocks. The developed flow provides a platform to exploit different kind of open-source IP's on industrial development environment since, it can produce synthesis results for 7 nm standard cell technology node quickly.

Future work concerning the thesis could include exploring different layouts for partition_c to improve the setup timing. After that the ASIC backend design process could be continued, to create a finished DLA hard macro. Also, the developed flow could be repeated for more advanced technology nodes. Concerning NVDLA, it could be

integrated into SoC to study its acceleration capabilities on different application fields.
This could be targeted to FPGA or ASIC device.

REFERENCES

- [1] V. Taratee, Digital logic design using Verilog: Coding and RTL synthesis, Springer (India) private limited, New Delhi, 2016, p. 255–322.
- [2] J.M. Veendrick H., Nanometer CMOS ICs from basics to ASICs, Springer international publishing, 2nd ed., Cham, 2017, p. 321–398.
- [3] M.J. Flynn & W. Luk, Computer system design: System-on-Chip, John Wiley & Sons, Inc., Publication, New Jersey, 2011, p. 1–58.
- [4] M. Hubner & J. Becker, Multiprocessor System-on-Chip: Hardware design and tool integration, Springer Science+Business Media, New York, USA, 2011, p. 1–54.
- [5] D.D. Gajski, S. Abdi, A Gerstlauer & G. Schirner, Embedded System Design: Modeling, Synthesis and Verification, Springer Science+Business Media, New York, USA, 2009, p. 1–124.
- [6] NVIDIA Deep Learning Accelerator, NVDLA Open Source Project, NVIDIA Corporation, website. Available (accessed on 14.5.2021): [NVDLA Primer — NVDLA Documentation](#)
- [7] P. P. Chu, RTL hardware design using VHDL: Coding for Efficiency, Portability, and Scalability, John Wiley & Sons, Inc., Hoboken, New Jersey, 2006, p. 3–423.
- [8] V. Taratee, Logic Synthesis and SOC Prototyping: RTL Design using VHDL, Springer Nature Singapore Pte Ltd., 2020, p. 109–179.
- [9] Synopsys, Design Compiler user guide, website, Available (Accessed on 23.11.2021): [Design Compiler User Guide \(synopsys.com\)](#)
- [10] S. Ramakrishnan, Implementation of a Deep Learning Inference Accelerator on the FPGA, Master's thesis, Lund University, March 2020, 27 p. Available: <https://lup.lub.lu.se/luur/download?func=downloadFile&recordId=9007070&fileId=9007133>
- [11] S. Churiwala, S. Garg, Principles of VLSI RTL Design: A Practical Guide, Springer Science+Business Media, New York, USA, 2011, p. 43–94.
- [12] F. Gharsalli, S. Meftali, F. Rousseau & A. A. Jerraya, Automatic Generation of Embedded Memory Wrapper for Multiprocessor SoC, Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324), New Orleans, Louisiana, USA, June 10-14, 2002. IEEE Electronic Library (IEL), pp. 596–601.
- [13] V. Shibu, Company internal memory wrapper tool.

- [14] NVDLA source code repository in GitHub, website, Available (accessed on 2.9.2021): [hw/vmod/nvdla at nvdlav1 · nvdla/hw · GitHub](https://github.com/nvdla/nvdla)
- [15] A. Kulmala, Scalable Multiprocessor System-on-chip Architecture Design on FPGA, dissertation, Tampere University of Technology, Publication 793, 2009, 17 p., Available: <http://urn.fi/URN:NBN:fi:tyy-200903131044>
- [16] R. Struharik, B. Vukobratovic, A. Erdeljan, & D. Rakanovic, CoNNA – Compressed CNN Hardware Accelerator, 2018 21st Euromicro Conference in Digital System Design (DSD) 2018-08, Prague, Czech Republic, August 29-31, 2018. IEEE Electronic Library (IEL), pp. 365–372.
- [17] D. A. Pham & B. C. Lai, Dataflow and microarchitecture co-optimisation for sparse CNN on distributed processing element accelerator, IET circuits, devices & systems, 2020-11, Vol.14 (8), HERTFORD: The Institution of Engineering and Technology, pp. 1185–1194.
- [18] Y. Chen, J. Emer & V. Sze, Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks, 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, August 24, 2016, pp. 367–379.
- [19] V. A. Chandrasetty, VLSI Design: A Practical Guide for FPGA and ASIC Implementations, Springer New York Dordrecht Heidelberg London, New York, USA, 2011, pp. 57.
- [20] K. Golshan, Physical design essentials: An ASIC Design Implementation Perspective, 2007 Springer Science+Business Media, LLC, Newport Beach, California, USA, 2007, pp. 25–26.
- [21] A. Teman, Digital VLSI Design: lecture 1, Bar-Ilan University, Faculty of Engineering, website, Available (Accessed on 22.11.2021): [Digital VLSI Design Lecture 1: Introduction \(biu.ac.il\)](https://www.biu.ac.il/~eng/dvlsi/lecture1/)
- [22] Digital ASIC Design: A Tutorial on the Design Flow, Digital ASIC Group, Lund University, 20 October, 2005, website, Available (Accessed on 16.8.2021): [dasic.dvi \(lth.se\)](https://www.dasic.dvi.lth.se)
- [23] L. Cavigellini, & L. Benini, Origami: A 803-GOp/s/W Convolution Network Accelerator, IEEE Transactions on Circuits and Systems for Video Technology, vol. 27, no. 11, Nov. 2017, July 18, 2016, pp. 2461–2475.
- [24] Y. Chen, T. Krishna, J. S. Emer, & V. Sze, Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Network, IEEE Journal of Solid-State Circuits, vol. 52, no. 1, Jan. 2017, November 8, 2016, pp. 127–138.
- [25] J. Nurmi, Processor Design System-On-Chip Computing for ASICs and FPGAs, 1st edition, 2007, Dordrecht: Springer Netherlands, 2007, Web, pp. 7–26.

- [26] H. Kwon, M. Pellauer, A. Parashar & T. Krishna, Flexion: A Quantitative Metric for Flexibility in DNN Accelerators, in *IEEE Computer Architecture Letters*, vol. 20, no. 1, 1 Jan.-June 2021, December 14, 2020, pp. 1–4.
- [27] H. Bolhasani, & S. J. Jassbi, Deep learning accelerators: a case study with MAESTRO, *Journal of big data*, 2020-11-12, Vol.7 (1), pp. 1–11.
- [28] F. Farshchi, Q. Huang & H. Yun, Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim, 2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2), 2019, pp. 21–25.
- [29] Q. Zhang, M. Zhang, T. Chen, Z. Sun, Y. Ma & B. Yu, Recent advances in convolutional neural network acceleration, *Neurocomputing (Amsterdam)*, 2019-01-05, Vol.323, Elsevier B.V., September 17, 2018, pp. 37–51.
- [30] S. Feng, J. Wu, S. Zhou and R. Li, The Implementation of LeNet-5 with NVDLA on RISC-V SoC, 2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS), 2019, March 19, 2020, pp. 39–42.
- [31] Y. -H. Chen, T. -J. Yang, J. Emer and V. Sze, Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices, in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, June 2019, pp. 292–308.
- [32] B. Moons, R. Uytterhoeven, W. Dehaene and M. Verhelst, 14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI, 2017 IEEE International Solid-State Circuits Conference (ISSCC), 2017, pp. 246–247.
- [33] S. Yin et al., A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications, 2017 Symposium on VLSI Circuits, 2017, pp. C26-C27.
- [34] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim and H. -J. Yoo, UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision, 2018 IEEE International Solid - State Circuits Conference - (ISSCC), 2018, pp. 218-220.